Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

Recorte

Projeção

Rasterização

Visibilidade

A história até aqui…

Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

Recorte

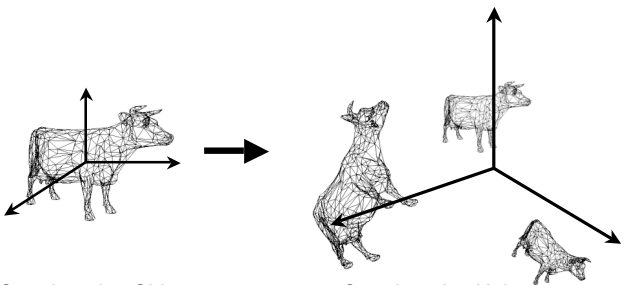
Projeção

Rasterização

Visibilidade

✓Objetos definidos no seu próprio sistema de coordenadas

✓Transformações de modelagem orientam os modelos geométricos num sistema comum de coordenadas (UNIVERSO)



Coordenadas Objeto Coordenadas Universo

Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

Recorte

Projeção

Rasterização

Visibilidade

✓Vértices iluminados de acordo com as propriedades geométricas e de material

✓Modelo de Iluminação Local

Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

Recorte

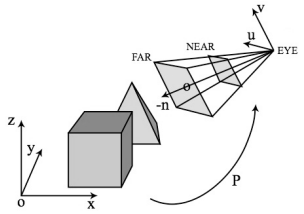
Projeção

Rasterização

Visibilidade

Resumo

✓Mapeamento de coordenadas de Universo para câmera



Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

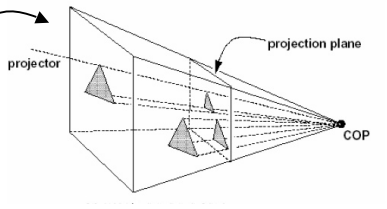
Recorte

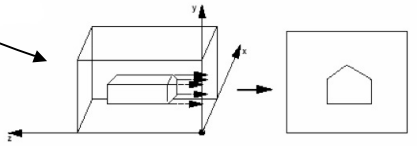
Projeção

Rasterização

Visibilidade

✓Escolha da projeção: perspectiva ou ortográfica





✓Os vértices são projetados para coordenadas da janela (Window)

Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

Recorte

Projeção

Rasterização

Visibilidade

Aula de hoje…

Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

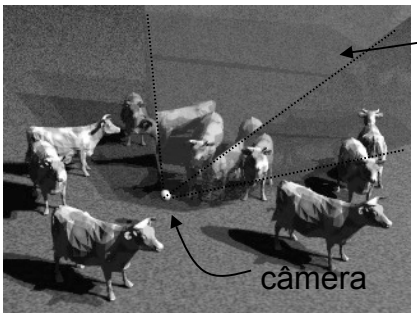
Recorte

Projeção

Rasterização

Visibilidade

✓Eliminação de objetos inteiros e partes de objeto que estão fora do *Frustum*

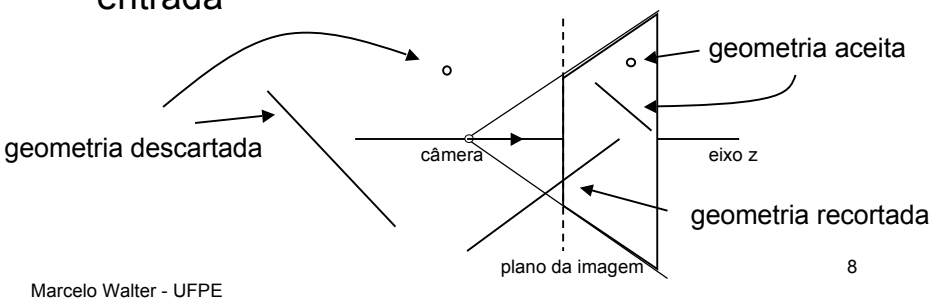


Universo

Imagem final

Estratégias para Recorte

- Recortar durante a rasterização (livro do Foley chama de *scissoring,* recorte em 2D, não usaremos)
- Recorte analítico: altera a geometria 3D de entrada



Marcelo Walter - UFPE

8

Importância do Recorte

- Etapa de otimização
- Preprocessamento para determinação de visibilidade
  - Não exibe primitivas ‘atrás’ da câmera
- Garante que somente primitivas potencialmente visíveis serão rasterizadas

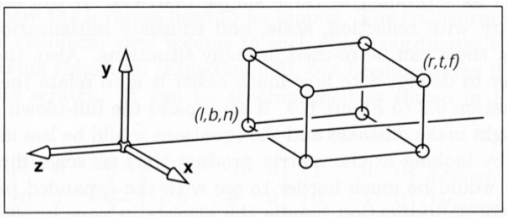
MAS ANTES DE RECORTAR…

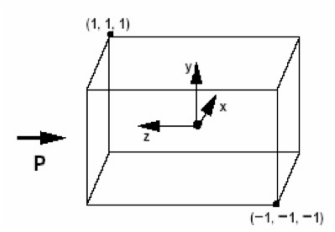
Coordenadas Normalizadas

- Vantagens
  - Recorte mais eficiente para um volume retangular, alinhado com os eixos (*Volume-padrão*)
  - *Frustum* tem geometria arbitrária
- Desvantagens
  - Como todos os pontos são transformados antes do recorte, provavelmente estaremos transformando pontos que posteriormente serão eliminados pelo recorte

Coordenadas Normalizadas

Projeção Ortográfica





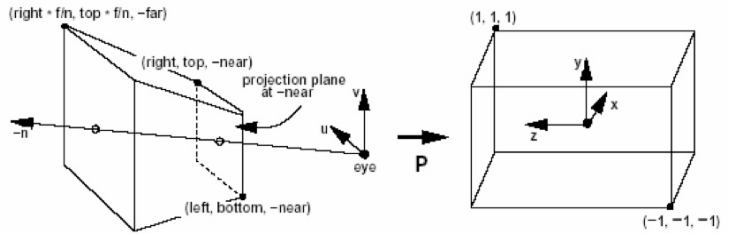
matriz de conversão*

*A obtenção desta matriz é um bom exercício de Álgebra :-)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & \frac{-(\text{right}+\text{left})}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{bottom}-\text{top}} & 0 & \frac{-(\text{bottom}+\text{top})}{\text{bottom}-\text{top}} \\ 0 & 0 & \frac{2}{\text{far}-\text{near}} & \frac{-(\text{far}+\text{near})}{\text{far}-\text{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Coordenadas Normalizadas

Projeção Perspectiva



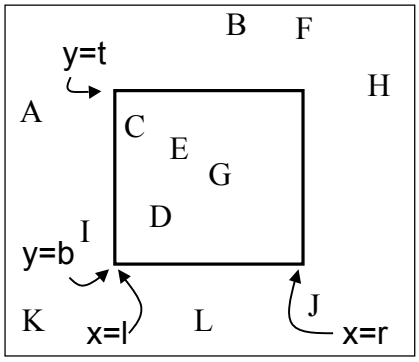
matriz de conversão

$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \begin{bmatrix} \frac{2 \cdot \text{near}}{\text{right}-\text{left}} & 0 & \frac{-(\text{right}+\text{left})}{\text{right}-\text{left}} & 0 \\ 0 & \frac{2 \cdot \text{near}}{\text{bottom}-\text{top}} & \frac{-(\text{bottom}+\text{top})}{\text{bottom}-\text{top}} & 0 \\ 0 & 0 & \frac{\text{far}+\text{near}}{\text{far}-\text{near}} & \frac{-2 \cdot \text{near} \cdot \text{far}}{\text{far}-\text{near}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

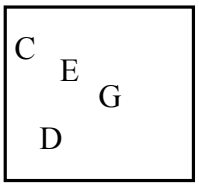
Algoritmos de recorte

(veremos em 2D primeiro e depois estendemos para 3D)

Recorte de pontos



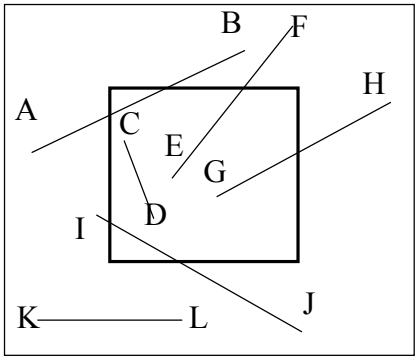
Qual a condição a ser satisfeita?

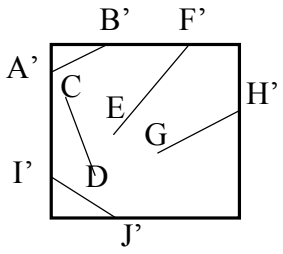


l <= x <= r
b<= y <= t

Algoritmos de recorte

Recorte de linhas

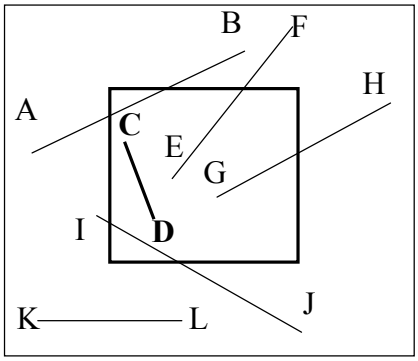




Saída desejada

Algoritmos de recorte

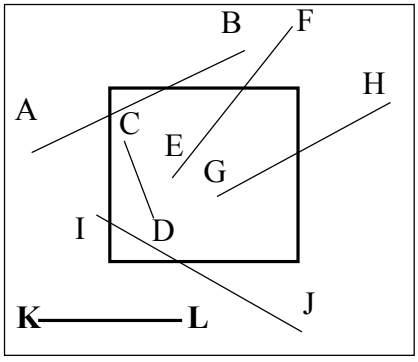
Recorte de linhas: *Trivialmente aceito*



Pontos estão dentro da janela

Algoritmos de recorte

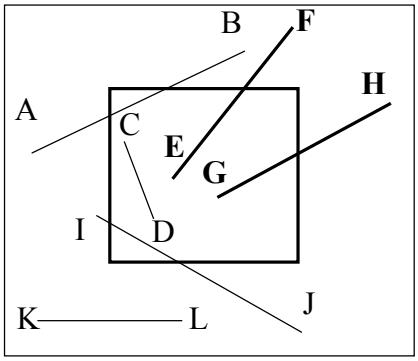
Recorte de linhas: *Trivialmente recusado*



Os dois pontos estão fora do retângulo e linha não cruza janela

Algoritmos de recorte

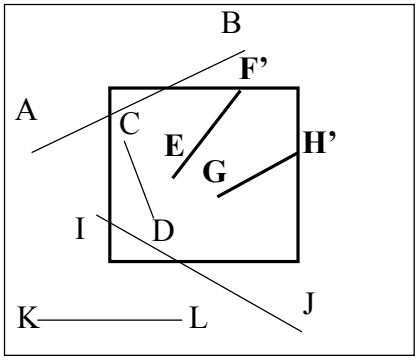
Recorte de linhas: *Cálculos de recorte*



Um dos pontos da linha está fora e outro está dentro da janela

Algoritmos de recorte

Recorte de linhas: *Cálculos de recorte*

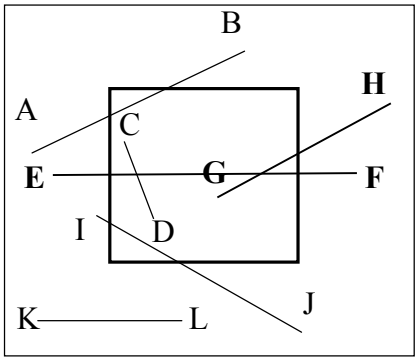


Um dos pontos da linha está fora e outro está dentro da janela

Linha deve ser recortada

Algoritmos de recorte

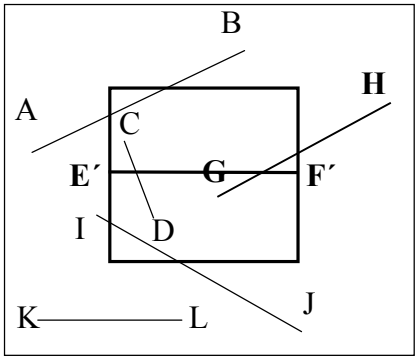
Recorte de linhas: *Cálculos de recorte*



Os dois pontos estão fora

Algoritmos de recorte

Recorte de linhas: *Cálculos de recorte*

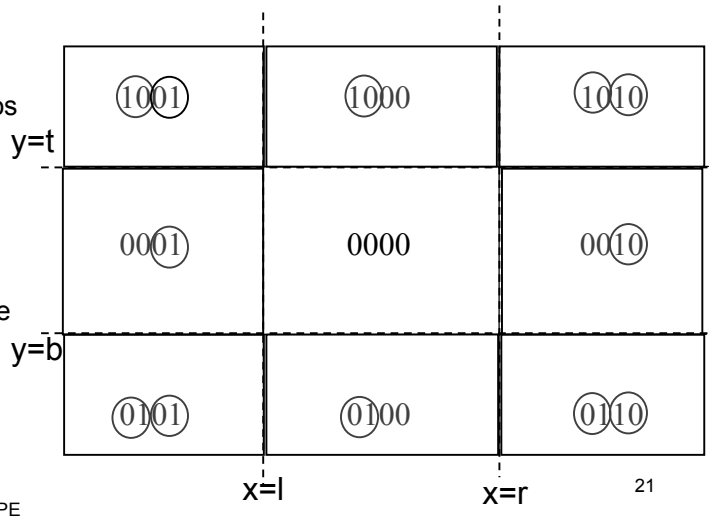


Os dois pontos estão fora

Linha deve ser recortada

Algoritmo de Cohen-Sutherland

- Maneira eficiente de determinar os diferentes casos
- Divide a região em 9 subespaços
- Atribuição de códigos aos espaços



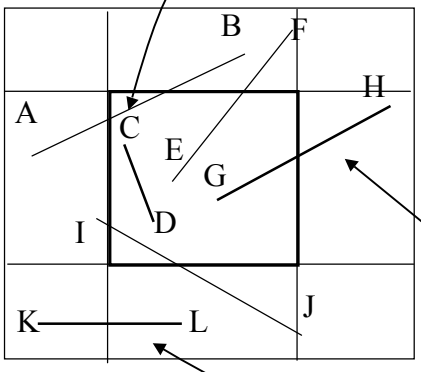
Cohen-Sutherland
Outcodes

If y > t ➔ seta primeiro bit em 1
If y < b ➔ seta segundo bit em 1
If x > r ➔ seta terceiro bit em 1
If x < l ➔ seta quarto bit em 1

http://www.cs.princeton.edu/~min/cs426/jar/clip.html

↖ Applet exemplificando o algoritmo

Cohen-Sutherland
Usando os outcodes

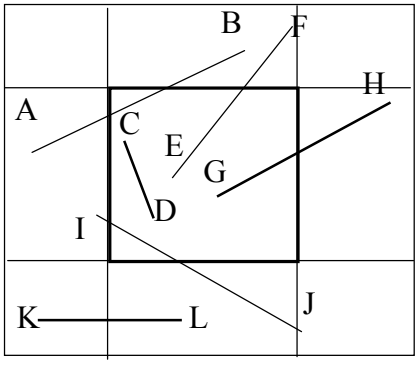


• Como devem ser os outcodes dos pontos trivialmente aceitos?

•Dos pontos trivialmente recusados?

• O que fazer com os que não são nem aceitos nem recusados?

Cohen-Sutherland
Usando os outcodes



• Como devem ser os outcodes dos pontos trivialmente aceitos?
0000

•Dos pontos trivialmente recusados?
AND bitwise diferente de 0!

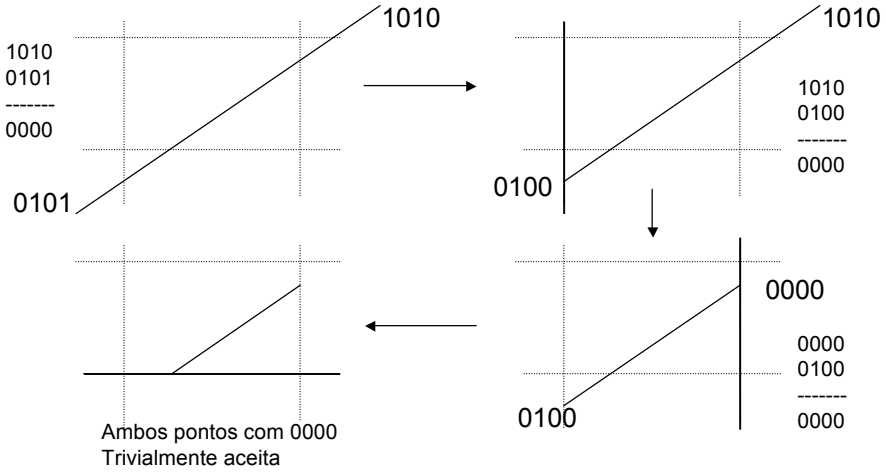
• O que fazer com os que não são nem aceitos nem recusados? Recorte por segmentos

Passos

1. Pontos são verificados para aceite e rejeição trivial utilizando os *outcodes*
2. Caso a linha precise ser recortada, divide em segmentos pelo limite da janela
3. Recorte iterativo até que a linha passe o teste de trivialmente aceita ou trivialmente rejeitada

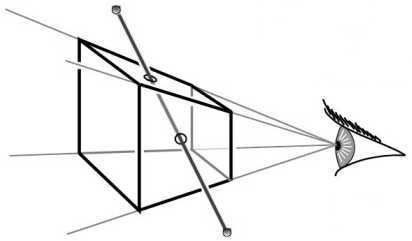
Cohen-Sutherland

Exemplo (ordem arestas l,r,b,t)



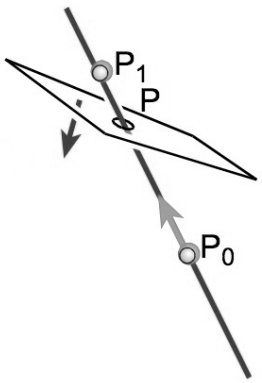
Recorte 3D

- Extensão de Cohen-Sutherland para 3D
- 6 outcodes ao invés de 4
  - bit 1: ponto está acima do volume
  - bit 2: ponto está abaixo do volume
  - bit 3: ponto está à direita do volume
  - bit 4: ponto está à esquerda do volume
  - bit 5: ponto está atrás do volume
  - bit 6: ponto está à frente do volume
- Os casos permanecem os mesmos



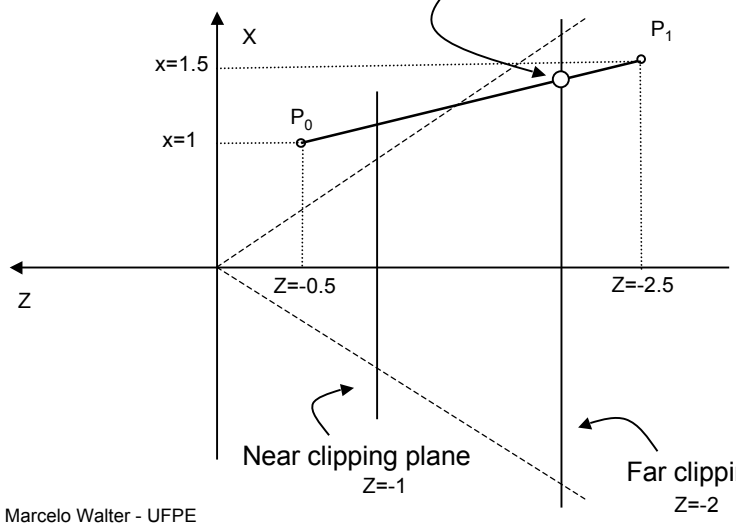
Calculando as intersecções

- Utilizamos a representação paramétrica de uma reta que passa por 2 pontos P_0 e P_1
- $L(t) = (1 - t) . P_0 + t . P_1$
- Substituimos esta equação na equação do plano que estamos recortando contra (genericamente $Ax+By+Cz+D=0$)



Exemplo

Qual o ponto de intersecção entre o segmento de reta dado por P_0P_1 e o plano z=-2?



- Eq paramétrica da reta
  - $L(t) = (1 - t)\ P_0 + t\ P_1$
  - $L_x(t) = (1 - t)\ 1 + 1.5\ t$ *
  - $L_z(t) = (1 - t)\ (-0.5) + -2.5t$ **
- Eq do plano z = -2
- Subs eq do plano em ** temos:

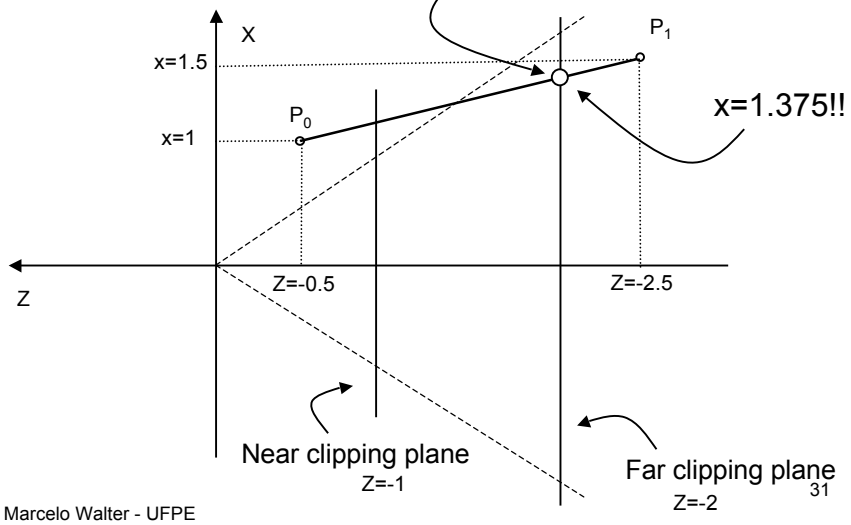
  -2 = -0.5 +0.5t - 2.5t

  t = 1.5/2 = 0.75
- Fazendo t=0.75 em * temos

  $L_x(0.75) = (1-0.75)\ 1 + 1.5\ (0.75) = 1.375$

Exemplo

Qual o ponto de intersecção entre o segmento de reta dado por P_0P_1 e o plano z=-2?



Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

Recorte

Projeção

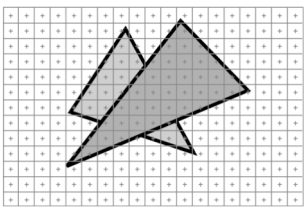
Rasterização

Visibilidade

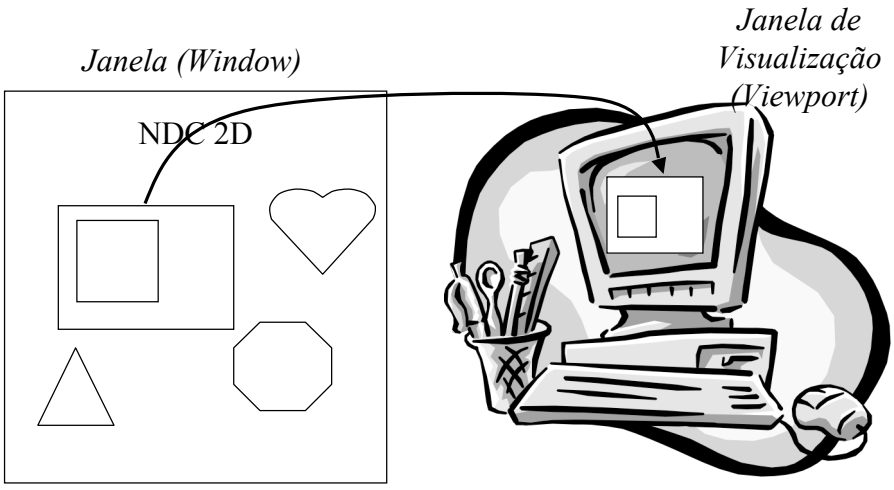
✓Rasterização das linhas em pixels e possível preenchimento

✓Expressão das coordenadas em coordenadas de tela (*Viewport*)

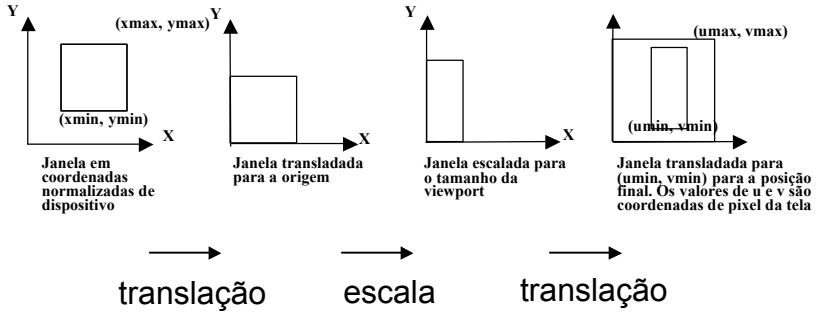
✓Interpolação dos valores (profundidade, normal, cor) dos vértices conforme necessidade



Onde estamos na tela?

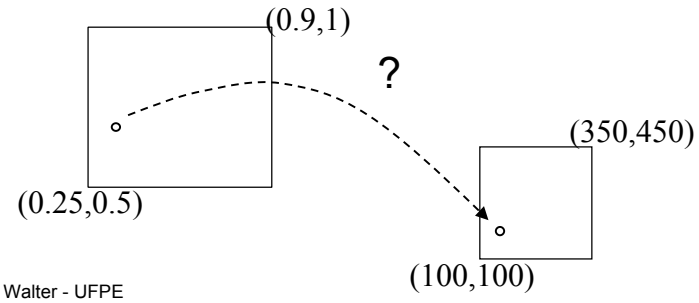


Transformação Window-to-Viewport



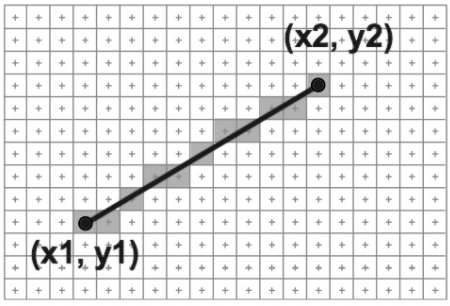
Exemplo

- O ponto (0.3,0.65) vai para qual pixel da janela? E o ponto (0.575,0.75)?
- Verifique que as respostas são (119,205) e (225,275)



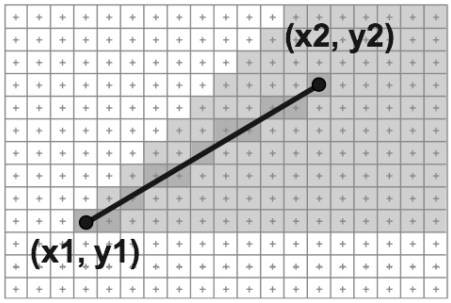
Algoritmos de rasterização

- *Objetivo*
  Aproximar primitivas matemáticas descritas através de vértices em *coordenadas reais* por meio de um conjunto de pixels em *coordenadas inteiras*



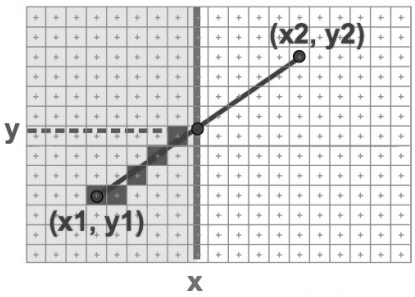
Algoritmos de rasterização

- Assumiremos
  m = dy/dx
  0 < m <= 1
- Exatamente um pixel por coluna



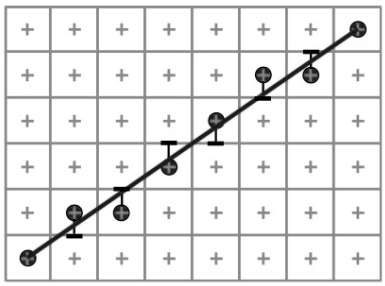
Algoritmo Incremental

- Calcula y em função de x
  - X sempre avança 1
  - Y acompanha de acordo com eq da reta
  - $y = y_1 + m\ (x - x_1)$
  - $y = y_1 + m$ Conversão para inteiro
  - $y_{tela} = floor(y+0.5)$



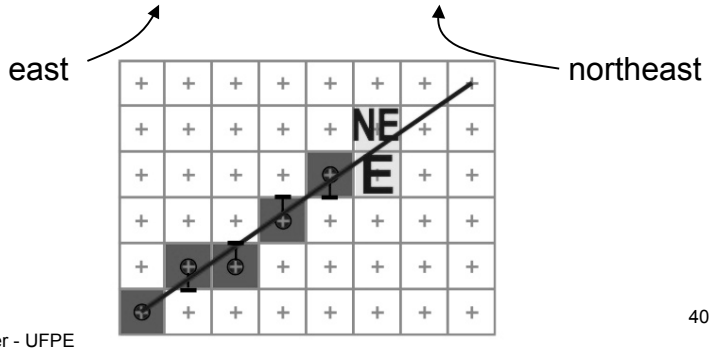
Algoritmo do Ponto Médio

- Bresenham - 1962
  - Atrativo porque usa somente operações aritméticas (não usa *round* ou *floor*)
  - Obtém mesma seqüência do algoritmo anterior
  - É incremental (usa o último resultado)



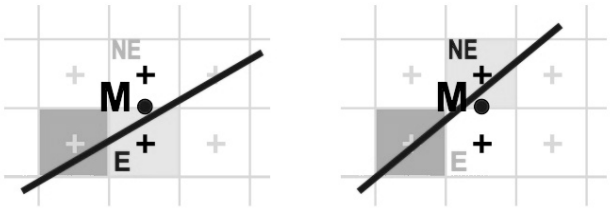
Bresenham
Algoritmo do Ponto Médio

- Observação chave
  Se estivermos no pixel (x_i,y_i) o próximo pixel ou é E (x_i+1, y_i) ou NE (x_i+1,y_i+1)



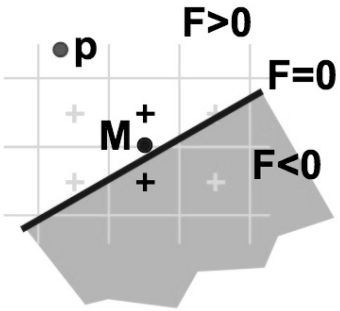
Qual pixel escolher? E ou NE?

- E se o segmento passar *abaixo* ou *sobre* o ponto médio M
- NE se o segmento passar *acima* do ponto médio M



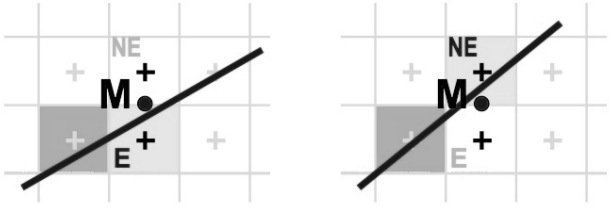
Como saber onde passa o segmento?

- Eq. da reta
  y = mx + b → y-mx-b = 0
  F(x,y) = y - mx - b
- O valor de F(x,y) nos diz onde o ponto está em relação à reta verticalmente
  - F(x,y) = 0 → Ponto está NA linha
  - F(x,y) > 0 → Ponto está ACIMA da linha
  - F(x,y) < 0 → Ponto está ABAIXO da linha



Decision Function

- Qual as coordenadas do ponto médio?
  (x+1, y + 0.5)
- D(x,y) = y - mx - b
  - D(x,y) < 0 → escolhe NE
  - D(x,y) > 0 → escolhe E
  - D(x,y) = 0 → arbitrariamente escolhe uma possibilidade (consistentemente)



Outros tipos de retas

- Caso for horizontal ou vertical tratar como caso especial
- Caso $x_1 > x_2$ inverter ordem dos pontos
- Para as outras inclinações de segmentos generalização do caso visto
  (não veremos aqui…)

Transformações Modelagem

Iluminação (*Shading*)

Transformação Câmera

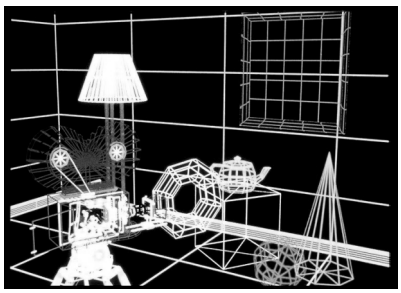
Recorte

Projeção

Rasterização

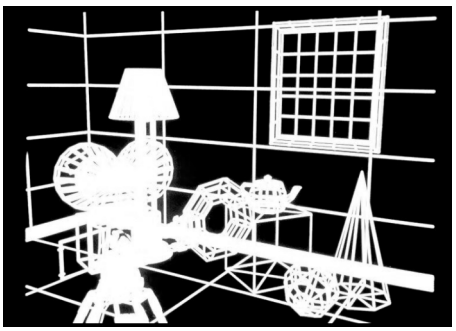
Visibilidade

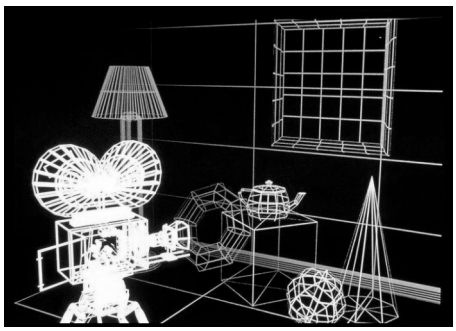
✓Desenhar apenas os polígonos visíveis



Objetivo Principal

Aumento do realismo





O que foi feito aqui?

46

Marcelo Walter - UFPE

Duas Abordagens

- A primeira abordagem determina quais dos ***n*** objetos são visíveis em cada pixel da imagem (precisão de imagem)

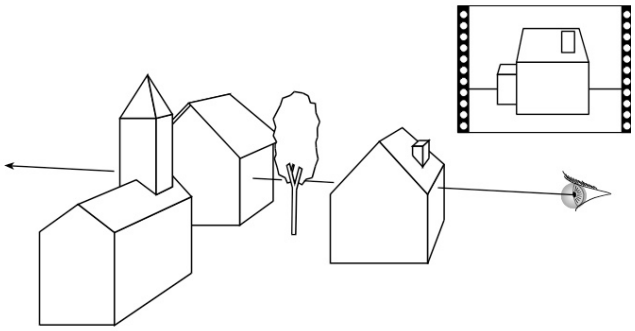
```
para( cada pixel na imagem)
{
          determinar o objeto mais próximo do observador;
          desenhar o pixel na cor apropriada;
}
```

- A segunda abordagem compara objetos diretamente uns com os outros, eliminando objetos inteiros ou porções deles que não são visíveis (precisão de objeto)

```
para( cada objeto no mundo)
{
          determinar as partes do objeto cuja visão não está obstruída
                    por outras partes dele ou de qualquer outro objeto;
          desenhar essas partes na cor apropriada;
}
```


Primeira Abordagem

- Ray Casting*



*veremos com mais detalhes nas próximas aulas

Segunda Abordagem

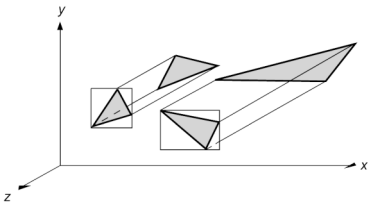
- Mais genérica e resolve o problema de forma exata, independentemente da resolução da imagem
- Dados n objetos complexidade n^2
- Testar cada objeto com todos os outros

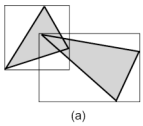
```
para( cada objeto no mundo)
{
          determinar as partes do objeto cuja visão não está obstruída
                    por outras partes dele ou de qualquer outro objeto;
          desenhar essas partes na cor apropriada;
}
```


Simplificações

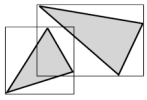
Bounding Box

- Baseado no princípio de coerência de objetos
  - Se um objeto está inteiramente separado de outro, basta comparar os dois objetos, e não cada face de cada objeto





(a)



(b)

Simplificações

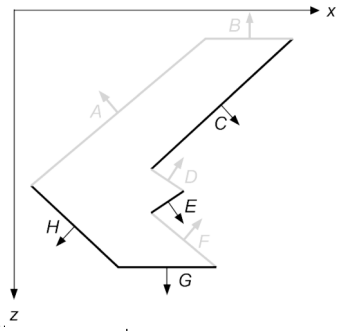
Backface Culling

- Eliminar as faces de um objeto obstruídas por ele mesmo
- Considera somente as faces de um único objeto
- Assumindo-se que o observador não está no interior de nenhum dos objetos da cena
- Para verificarmos se um polígono é *backface*, basta verificarmos se a normal do polígono não está apontando para o observador

Simplificações

Backface Culling

- Se N.V > 0, para objetos sólidos isso significa que essa face nunca será vista pelo observador
- V é um vetor que parte do olho para o objeto

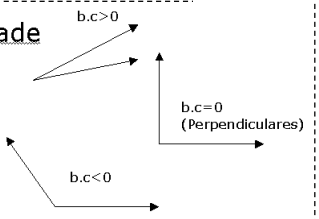


Sinal de b.c e Perpendicularidade

$\cos\theta > 0 \text{ se } |\theta| < 90^\circ \rightarrow \text{b.c} > 0$

$\cos\theta = 0 \text{ se } |\theta| = 90^\circ \rightarrow \text{b.c} = 0$

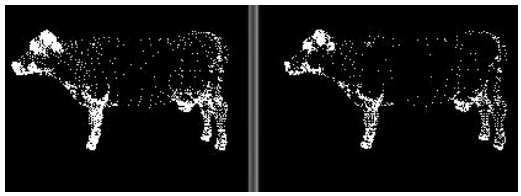
$\cos\theta < 0 \text{ se } |\theta| > 90^\circ \rightarrow \text{b.c} < 0$



Simplificações

Backface Culling

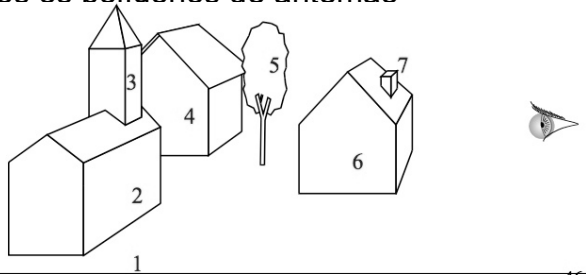
- Não é uma solução completa:
  - Resolve apenas as faces de um mesmo objeto e não sua relação com os demais
  - Se os polígonos possuem ambos os lados visíveis (eles não formam um volume), o algoritmo não pode ser aplicado
- Teste barato computacionalmente. Na média metade dos polígonos serão descartados
- OpenGL - `glEnable(GL_CULL_FACE)`



Com backface culling

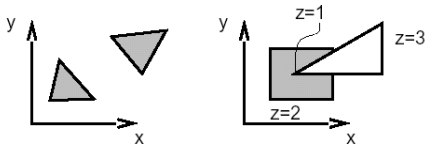
Algoritmo do Pintor

- Desenhar os polígonos assim como um pintor a óleo faz. O mais distante primeiro
  - Ordenar polígonos de acordo com z
  - Resolver ambigüidades onde z's se sobrepõem
  - Desenhar a partir do maior z até o menor
  - Visto que o mais próximo é desenhado por último, ele estará no topo (e por isso ele será visto)
  - Precisa de todos os polígonos de antemão

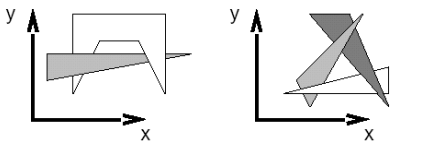


Algoritmo do Pintor

- Necessidade de pré-ordenação
- Quando z's se sobrepõem
  - Alguns casos triviais:



  - Outros muito complexos



- Necessidade de se subdividir polígonos

Z-Buffer

Depth Buffer

- Criado por Catmull (1974)
- Idéia:
  - Na etapa de rasterização, considerar z, além de x e y
  - Além do *framebuffer*, teremos um *depth buffer* (z buffer), onde estarão armazenados os z's
  - Inicialmente, setamos todo o z-buffer para infinito (ou para o *far clipping plane*)
  - *Framebuffer* é todo setado para cor de fundo

Z-Buffer

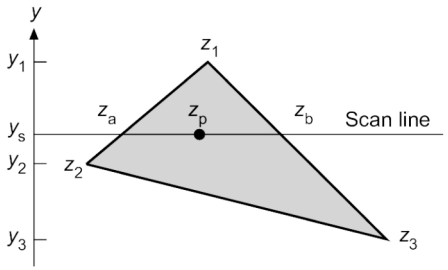
- Realizar a etapa de rasterização, usando a seguinte rotina para preencher o *framebuffer*

```
WritePixel(int x, int y, float z, colour)
if ( z < zbuf[x][y] ) then
    zbuf[x][y] = z;
    frambuffer[x][y] = colour;
end
```

- Ou seja, o buffer armazena o pixel mais próximo até então em (x,y)
- Não renderiza um pixel se ele tem a profundidade maior que z[x][y]

Z-Buffer

- Determinando profundidades



$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

Z-Buffer

- Vantagens
  - Fácil de implementar
  - Simples de implementar em hardware
  - Objetos não precisam ser necessariamente polígonos
- Desvantagens
  - Consome muita memória
    - Se diminuir memória, podem ocorrer erros
      - faces A = 0.89485, B = 0.89501 / Round: A = 0.895, B = 0.895
      - Renderiza A, depois B (Fica B, errado!, pois A é mais próxima)
  - Dependente do dispositivo
  - O algoritmo pode desenhar um pixel muitas vezes

Z-Buffer em OpenGL/GLUT

```
glutInitDisplayMode(GLUT_DEPTH| ...);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GLenum func); //func: GL_NEVER, GL_ALWAYS, GL_LESS, GL_EQUAL,
                          //, GL_LEQUAL, GL_GEQUAL,
                          //, GL_GREATER, GL_NOTEQUAL
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
...
drawObjects()
```

Comparação de Algoritmos

- *Backface Culling* rápido, porém insuficiente por si só
- Algoritmo do pintor é independente de dispositivo, mas com muitos detalhes, além de lento
- Z-buffer é rápido e fácil de implementar , é dependente do dispositivo e consome muita memória (Porém é o escolhido para HW)

Perguntas?