Determinação de Superfícies Visíveis

M.C.F. de Oliveira

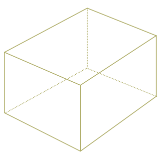
Fontes: Hearn & Baker, Cap. 9
Curso CG, University of Leeds (Ken Brodlie):
http://www.comp.leeds.ac.uk/kwb/gi21/lectures.html

Rendering de Polígonos

- Por eficiência e realismo, só queremos renderizar as faces poligonais que são visíveis para a câmera
  - Back-face culling (´rejeição trivial´)
  - Z-buffer (remoção de faces ocultas por outros objetos)

Back Face Culling

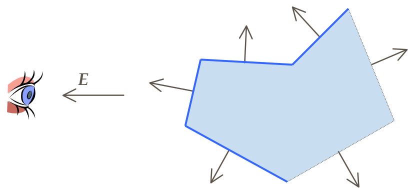
- Se faces pertencem a um objeto sólido (um poliedro, por exemplo), não é necessário renderizar as faces de trás (não visíveis)



Apenas 3 faces precisam ser traçadas

Faces ´de trás´ podem ser removidas do pipeline

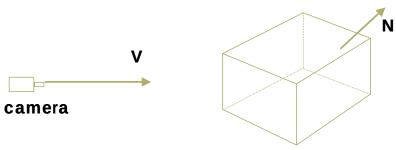
Back Face Culling



Nota: processo assume que cena é composta por objetos poliédricos fechados

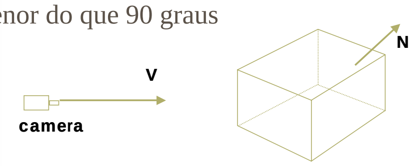
Back Face Culling

- Como descobrir quais são as faces ´de trás´??



Back Face Culling

- Um polígono tem sua face externa voltada para o lado oposto ao observador (é face de trás) se o ângulo entre o vetor normal à face (**N**) e o vetor direção de observação (**V**) é menor do que 90 graus



Back Face Culling

- Eficiente fazer esse teste no sistema de coordenadas de visualização
  - Vetor direção de observação paralelo ao eixo z_v
  - ou seja $\mathbf{V} = (0,0,V_z)$, e $\mathbf{V} \bullet \mathbf{N} = V_z * N_z$
- Portanto, para fazer o teste $\mathbf{V} \bullet \mathbf{N} > 0$ basta verificar o sinal da componente z do vetor normal à face

7

Back Face Culling

- Muito importante para *rendering* mais eficiente (simplica muito a cena)
  - em geral é o primeiro passo do processo

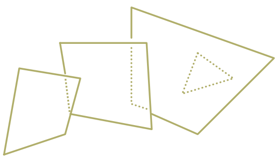
```
glEnable(GL_CULL_FACE)
glCullFace(GL_BACK)
```

- Com isso, restam apenas os polígonos/faces potencialmente visíveis para a câmera…

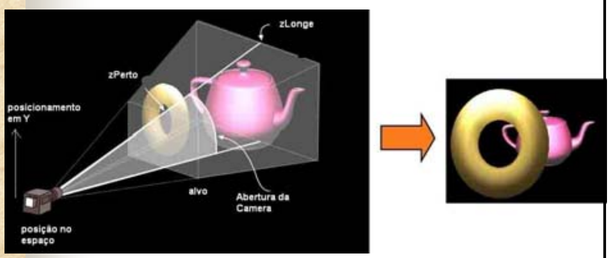
8

Outro Problema: Faces Ocultas

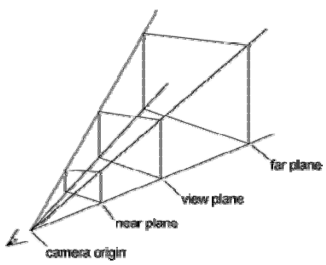
- Algumas faces da cena ficam ocultas atrás de faces de outros objetos: só queremos desenhar (renderizar) as faces (ou partes delas) realmente visíveis



9



10

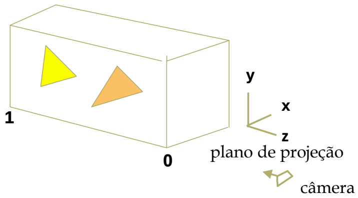


11

Solução – Algoritmo Z Buffer

- Ed Catmull, 1973
  - Considere que as faces passaram pela transformação de projeção, e tiveram suas coordenadas z retidas (informação de profundidade) – suponha valores de z normalizados no intervalo 0 a 1

Para cada pixel (x,y), queremos traçar a face mais próxima da câmera, i.e., com menor valor de z



12

Cenário – Algoritmo Z Buffer

- Faces poligonais projetadas no plano
  - Em geral, face = triângulo
- Cada face processada em separado
- Ordem de processamento das faces é arbitrária
- Cada face é rasterizada independentemente das demais

Algoritmo Z Buffer

- Algoritmo usa dois buffers
  - frame buffer armazena os valores RGB que definem a cor de cada pixel... tipicamente 24 bits, mais 8 bits para transparência (alfa)
  - z-buffer para manter informação de profundidade associada a cada pixel... tipicamente 16, 24 ou 32 bits
- Inicialização

  frame buffer inicializado com a cor de fundo da cena

  colour (x,y) = (IRED, IGREEN, IBLUE)$_{fundo}$

  z-buffer inicializado com 1 (*far clipping plane*, ou profundidade máxima, em relação ao observador)

  depth (x,y) = 1

Algoritmo Z Buffer

- Para cada face sendo renderizada (*scan converted* e Gouraud ou Phong *shaded*):

  calcula (se necessário) profundidade z para cada pixel projetado (x,y) da face

  if z < depth(x,y) then

  depth (x,y) = z;

  colour (x,y) = (IRED, IGREEN, IBLUE)gouraud/phong
- Note que esse algoritmo assume observador olhando para o eixo z positivo (sistema de coordenadas da mão esquerda)
  - valores maiores de z correspondem a profundidades maiores (mais distantes do observador)

Algoritmo Z Buffer

- Implementação eficiente: algoritmo pode explorar *coerência* de diversas maneiras…
- Depois de todas as faces processadas, o *depth buffer* contém a profundidade das superfícies visíveis, e o *frame buffer* contém as cores dessas superfícies
  - Cena pronta para ser exibida

Z Buffer

- Maior vantagem: simplicidade!
- Desvantagens
  - Quantidade de memória necessária (menos impacto no custo hoje, mas desempenho ainda é um aspecto crítico)
  - Alguns cálculos desnecessários… porque?
  - Precisão limitada para o cálculo de profundidade em cenas complexas pode ser um problema: quantização de valores de profundidade pode introduzir artefatos
- Placas gráficas otimizam operações no z-buffer
  - Clearing, z-buffers hierárquicos, etc.

Z Buffer

- OpenGL
  - habilita z-buffer

    glEnable (GL_DEPTH_TEST);
  - aloca z-buffer
    - Ex: glutInitDisplayMode (GLUT_RGB|GLUT_DEPTH);
    - Número de bits por pixel depende de implementaçã0o / disponibilidade de memória
  - Ao gerar novo quadro, limpar também o z-buffer

    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

Transparencia e alpha

- Para superfícies translúcidas, é preciso considerar como a luz se comporta **através de vários materiais**: face transparente não oculta outra…
- Alpha
  - Especificação de opacidade (ou transparência)
    - Alpha = 1 ➔ objeto opaco (opacidade máxima)
    - Alpha = 0 ➔ objeto totalmente transparente
    - 0 < Alpha < 1 ➔ objecto translúcido

Alpha blending (A = alpha)

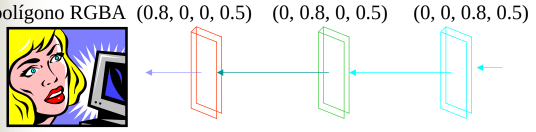
• Para compor a cor do pixel combina as cores e opacidades das superfícies visíveis neste pixel

R = Af Rf + (1 - Af) Rb
G = Af Gf + (1 - Af) Gb
B = Af Bf + (1 - Af) Bb
A = Af + (1 - Af) Ab

sendo (Rf, Gf, Bf, Af): cor e opacidade da superfície na frente
(Rb, Gb, Bb, Ab): cor e opacidade da superfície atrás
(R, G, B, A): cor resultante da combinação de ambas

Exemplo

Cor polígono RGBA (0.8, 0, 0, 0.5) (0, 0.8, 0, 0.5) (0, 0, 0.8, 0.5)



Cor percebida (0.4, 0.2, 0.1, 0.875) (0, 0.4, 0.2, 0.75) (0, 0, 0.4, 0.5)

Transparência

- Solução melhor
  - Renderiza todas as superfícies opacas e o *background* em qualquer ordem, com
    - Teste de profundidade (*depth*) abilitado
    - Atualizações no *depth buffer* abilitadas
    - *Alfa blending* desabilitado
  - Renderiza as superfícies transparentes de trás para frente, com
    - Teste de profundidade (*depth*) abilitado
    - Atualizações no *depth buffer* desabilitadas
    - Alfa *blending* abilitado

22

Sombras

- Z buffers também oferecem um excelente mecanismo para implementar sombras
- z buffer é usado para determinar o que é visível para a câmera (compara profundidades em relação ao observador)
- O processo de calcular sombras requer determinar o que é ‘visível’ para a fonte de luz

23

Shadow Z Buffer

- Um segundo z-buffer é usado, chamado *shadow z-buffer* (ou *shadow map*)
- Algoritmo em 2 passos
  - Cena é ‘renderizada’ utilizando como ponto de observação a posição da fonte de luz, com informação de profundidade armazenada no *shadow z-buffer* (não é necessário calcular as intensidades): informa a posição mais próxima da fonte de luz…
  - Cena é ‘renderizada’ utilizando como ponto de observação a posição da câmera, usando o modelo de iluminação e tonalização Gouraud ou Phong, com o algoritmo z-buffer... é necessário ajustar a cor caso o ponto esteja na sombra

24

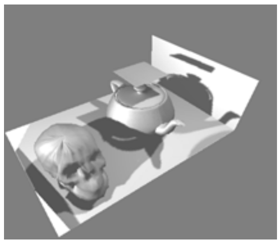
Shadow Z Buffer

- Para determinar se ponto está na sombra
  - Toma sua posição (x_O, y_O, z_O) na visão da câmera, e transforma na posição correspondente (x_O', y_O', z_O') na visão da fonte de luz
  - Recupera o valor z, digamos z_L , no *shadow z-buffer* na posição (x_O', y_O')
  - se z_L está mais próximo da fonte de luz do que z_O', significa que algum objeto está mais próximo da fonte e, portanto, este ponto está na sombra… nesse caso, apenas o termo de iluminação ambiente do modelo de iluminação deve ser atribuído ao ponto
- Resolução do shadow z-buffer é crítica
  - Porquê?

25

Exemplo Sombras

- Múltiplas sombras de múltiplos objetos e fontes de luz!
- Pergunta: quantas vezes um polígono é renderizado se existem duas fontes de luz?



Chris Bentley, WPI

26