Tratamento de linhas e superfícies escondidas

Professor: Luciano Ferreira Silva, Dr.

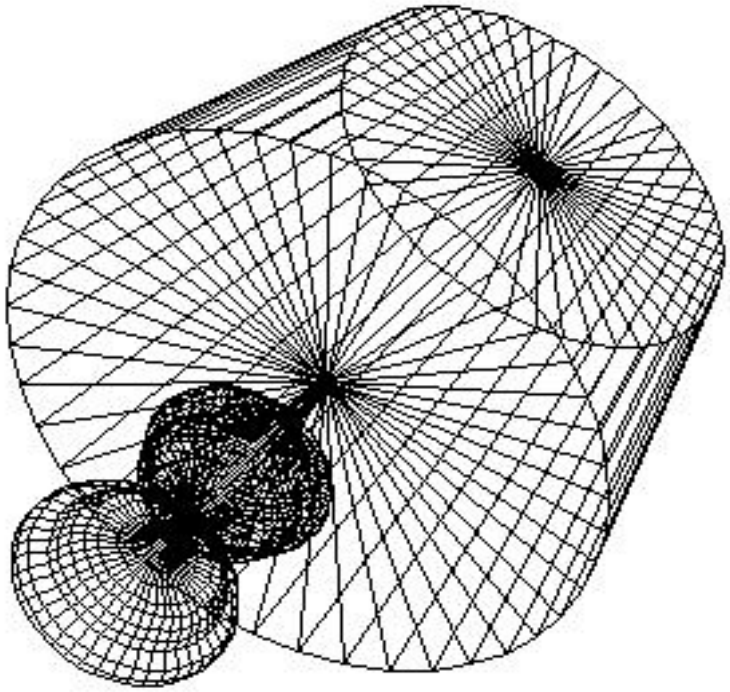
Problema

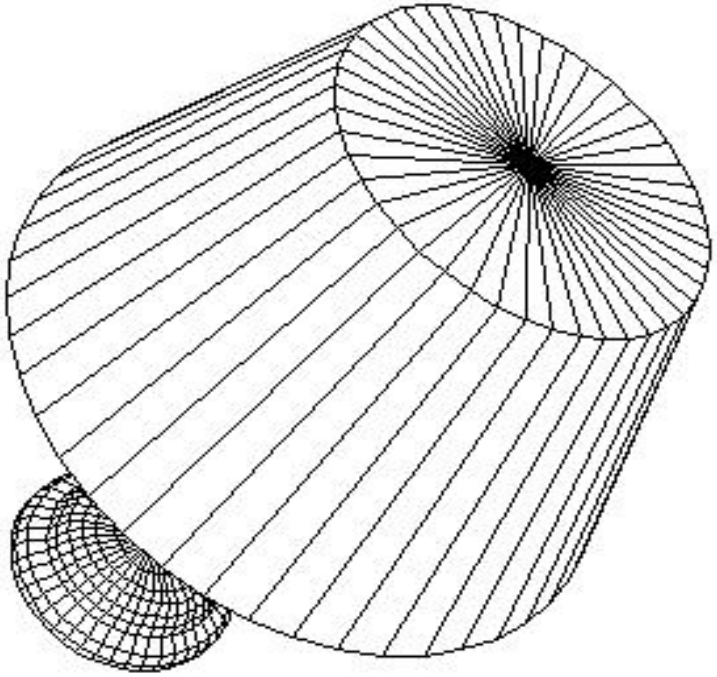
- **Dado um objeto 3D e uma especificação de visualização definindo o tipo de projeção, o problema é:**
  - ✓ Definir que linhas/superfícies do objeto são visíveis para o centro de projeção - proj. perspectiva;
  - ✓ Ou definir que linhas/superfícies do mesmo são visíveis ao longo da direção de projeção - proj. paralela.

Tratamento de linhas e superfícies escondidas

- A solução eficiente de problemas de visibilidade é o principal passo do processo de criação de cenas realísticas.
- Esse problema lida freqüentemente com a determinação da visibilidade de linhas e superfícies.
- Essa fase foi denominada de eliminação de polígonos ou faces escondidas (*culling back-faces*).

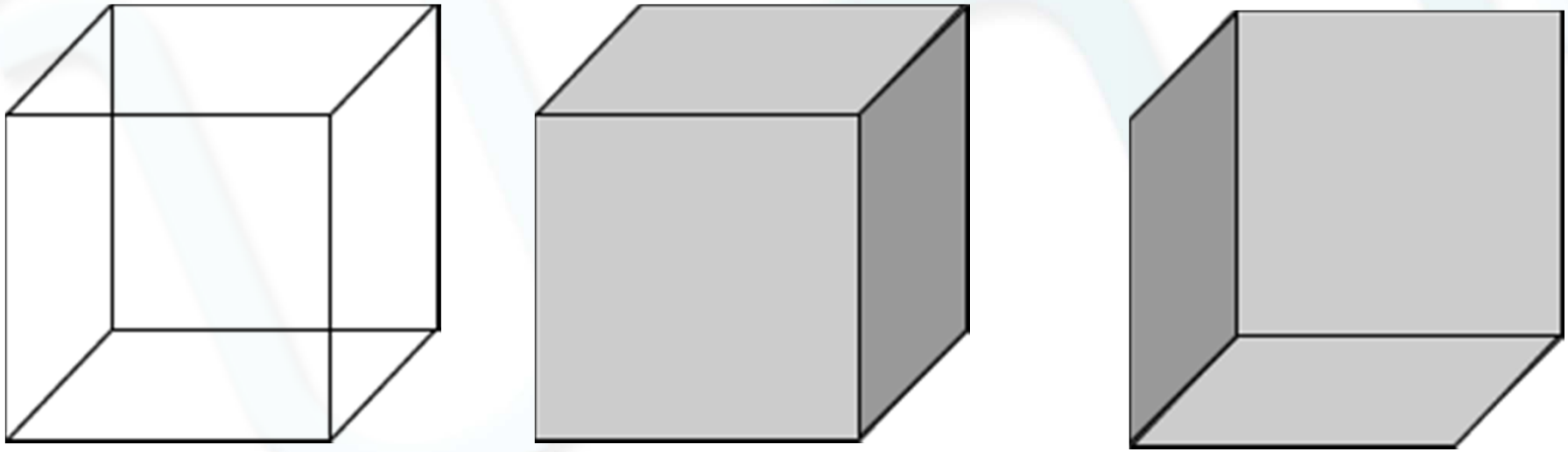
Tratamento de linhas e superfícies escondidas





Tratamento de linhas e superfícies escondidas

- Ilusões de profundidade - cubo de Necker;
- Representação aramada em A e suas possíveis interpretações em B e C:



A B C

Tratamento de linhas e superfícies escondidas

- **Classificação dos Algoritmos: *Object-space* e *Image-space;***
- ***Object-space*:**
  - ✓ Trabalham diretamente com as definições do objeto;
    - Ou seja, é a representação por um subconjunto de retas e curvas, extraídas dos contornos do objeto, de maneira a representá-lo;
  - ✓ A descrição matemática dessas retas e curvas é normalmente usada para gerar o objeto na memória do sistema sua exibição;

Tratamento de linhas e superfícies escondidas

- ***Imagem-space:***
  - ✓ Propõe a representação do sólido através de uma série de faces ou superfícies conectadas apropriadamente;
  - ✓ Uma aproximação interna simples pode ser utilizada para descrever cada face ou superfície, e a exibição do objeto é produzida como o agregado dessas faces ou superfícies;

Tratamento de linhas e superfícies escondidas

- **Em suma os algoritmos:**
  - ✓ *Object-space*: trabalham diretamente com as definições do objeto - compara objetos e partes dos objetos entre si, para definir que partes estão visíveis - cálculos geométricos com maior precisão!!
  - ✓ *Image-space*: trabalha com as imagens projetadas: a visibilidade é decidida ponto a ponto em cada posição do pixel sobre o plano de projeção!!

Métodos

Image Space:

- ✓ Z-buffer;
- ✓ Linhas de varredura;
- ✓ Sub-divisão por áreas;
- ✓ Ray Casting;

Object Space:

- ✓ Equação do plano;
- ✓ Produto escalar - back face removal;

Object Space

Características básicas:

- ✓ Entrada e saída são dados geométricos
- ✓ Independente da resolução da imagem
- ✓ Menos vulnerabilidade a *aliasing*
- ✓ Rasterização ocorre *depois*
- ✓ Exemplos:
  - Maioria dos algoritmos de recorte e *culling*
    - Recorte de segmentos de retas
    - Recorte de polígonos
  - Algoritmos de visibilidade que utilizam recorte
    - Algoritmo do pintor

Image Space

Características Básicas:

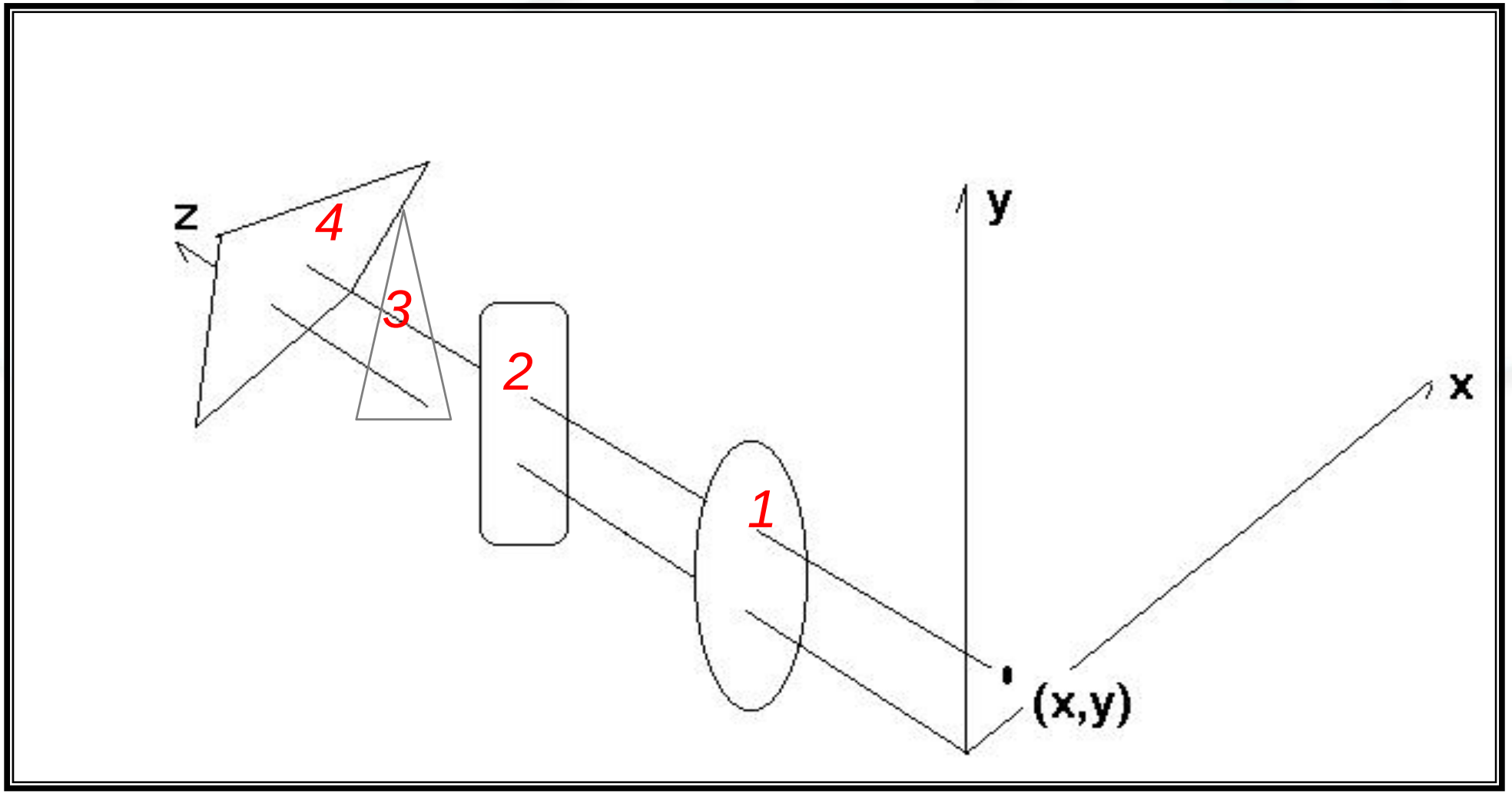
- ✓ Entrada é vetorial e saída é matricial
- ✓ Dependente da resolução da imagem
- ✓ Visibilidade determinada apenas em pontos (pixels)
- ✓ Podem aproveitar aceleração por hardware
- ✓ Exemplos:
  - Z-buffer
  - Linhas de varredura

Método z-buffer

- **Algoritmo simples;**
- **Armazena, para cada ponto da tela:**
  - ✓ Registro de profundidade dos objetos
  - ✓ Intensidade dos objetos de cena
- **Requer duas arrays**
- **Início da renderização**
  - ✓ *Buffer* de cor = cor de fundo
  - ✓ *z-buffer* = profundidade máxima
- **Elevado custo de armazenamento**

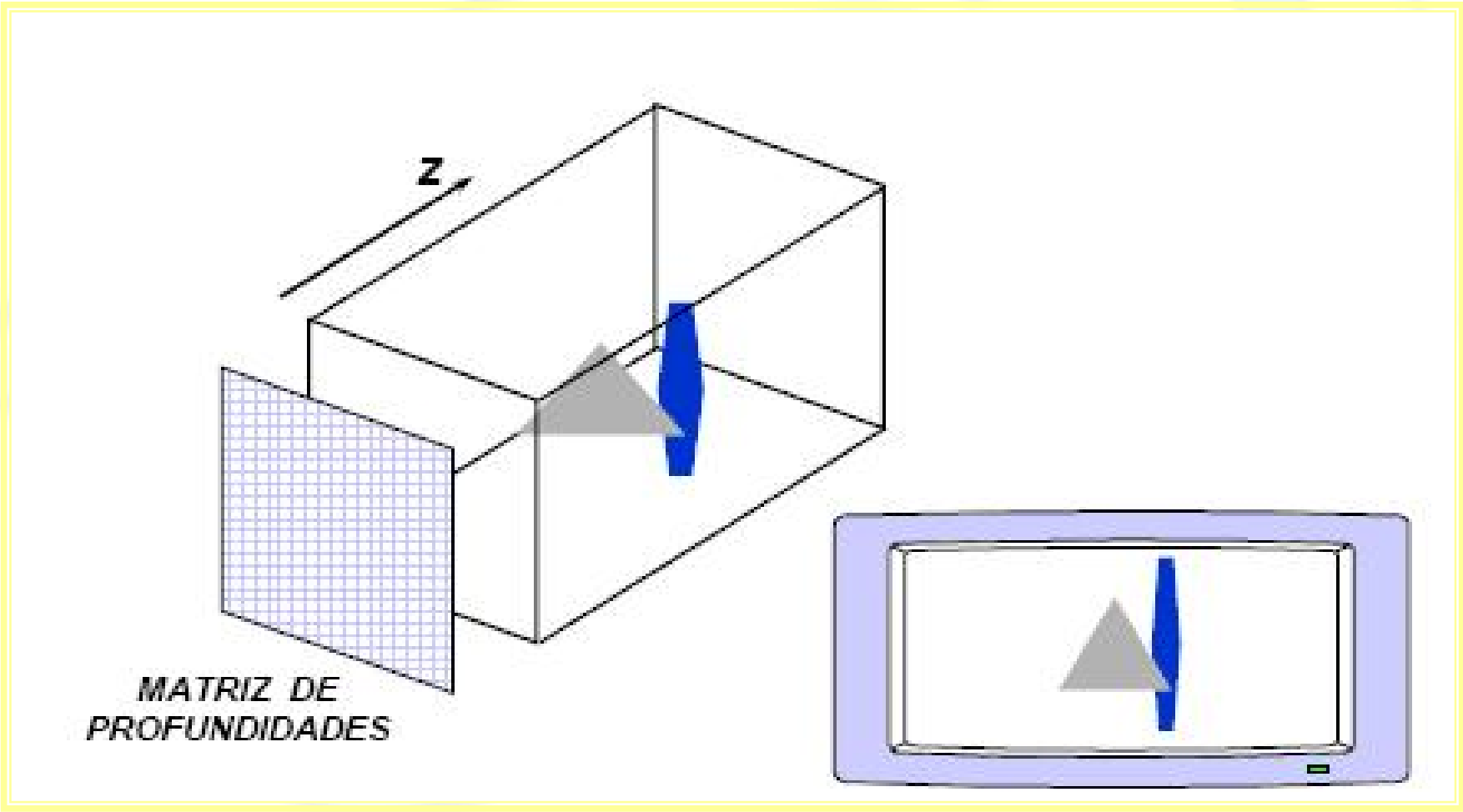
Método z-buffer

Lógica da análise:



Método z-buffer

- **Idéia Básica:**



Algoritmo: z- buffer

- Para todo pixel da tela, faça depth [x,y]=Max e intensidade [x,y]= valor de fundo da tela (*background*);
- Para cada polígono, encontre os pixel (x,y) que estão associados aos limites do polígono quando projetados na tela.
- Para cada um destes pixel:
  - Calcular o valor da profundidade z do polígono para posição (x,y);
  - Se z < depth [x,y] => coloque z (x,y) no z-buffer ou melhor, faça depth [x,y] = z e faça intensidade (x,y) igual à intensidade do polígono, caso contrário:
  - se z >= depth [x,y] => não tome nenhuma ação
- Depois de processados todos os polígonos, a array "intensidade" conterá a solução

z- buffer

- **Vantagens:**
  - ✓ Simples e comumente implementado em *Hardware;*
  - ✓ Objetos podem ser desenhados em qualquer ordem;

Z-Buffer

- **Desvantagens:**
  - ✓ Rasterização independe de visibilidade
    - Lento, se o número de polígonos é grande;
  - ✓ Erros na quantização de valores de profundidade podem resultar em imagens inaceitáveis;
  - ✓ Dificulta o uso de transparência ou técnicas de anti-aliasing;
    - É preciso ter informações sobre os vários polígonos que cobrem cada pixel;

z- buffer e transparência

- **Se há objetos semi-transparentes, a ordem de renderização é importante;**
- **Após a renderização de um objeto transparente, atualiza-se o z-buffer?**
  - ✓ Sim → novo objeto por trás não pode mais ser renderizado
  - ✓ Não → z-buffer fica incorreto
- **Soluções**
  - ✓ Estender o **z-buffer** → **A-buffer**

A-buffer

- **Permite implementação de transparência e de filtragem (*anti-aliasing*);**
- **Para cada pixel manter lista ordenada por z onde cada nó contém:**
  - Máscara de subpixels ocupados
  - Cor ou ponteiro para o polígono
  - Valor de z (profundidade)

A-buffer

- **1º Passo: Rasterização dos Polígonos:**
  - ✓ Se pixel totalmente coberto por um dado polígono e este polígono é opaco:
    - Inserir na lista removendo polígonos mais profundos
  - ✓ Se o polígono em questão é transparente ou não cobre totalmente o pixel
    - Inserir na lista
- **2º Passo: Geração da imagem**
  - ✓ Máscaras de subpixels são misturadas para obter cor final do pixel