# COMPUTAÇÃO GRÁFICA

Unidade 10 – Visibilidade e Recorte

Ivan Nunes da Silva


1

## O Problema de Visibilidade

- Numa cena tridimensional, normalmente não é possível ver todas as superfícies de todos os objetos;
- Não queremos que objetos ou partes de objetos não visíveis apareçam na imagem;
- Problema importante que tem diversas ramificações:
  - Descartar objetos que não podem ser vistos (*culling*);
  - Recortar objetos de forma a manter apenas as partes que podem ser vistas (*clipping*);
  - Desenhar apenas partes visíveis dos objetos:
    - Em aramado (*hidden-line algorithms*)
    - Superfícies (*hidden surface algorithms*)
  - Sombras (visibilidade a partir de fontes luminosas).

2

## Espaço do Objeto x Espaço da Imagem

- Métodos que trabalham no **espaço do objeto**:
  - 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.
- Métodos que trabalham no **espaço da imagem**:
  - 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
    - Scan-line

3

## Recorte (*Clipping*)

- **Problema Definido Por:**
  - Geometria a ser recortada: pontos, retas, planos, curvas, superfícies.
  - Restrições de recorte: Janela (2D).
- **Recorte de Segmento de Reta x Retângulo**
  - Entrada:
    - Segmento de reta $P_1$ - $P_2$
    - Janela alinhada com eixos (*xmin*, *ymin*) x (*xmax*, *ymax*).
  - Saída:
    - Segmento recortado.
  - Principais Métodos:
    - Cohen-Sutherland.
    - Liang-Barsky.

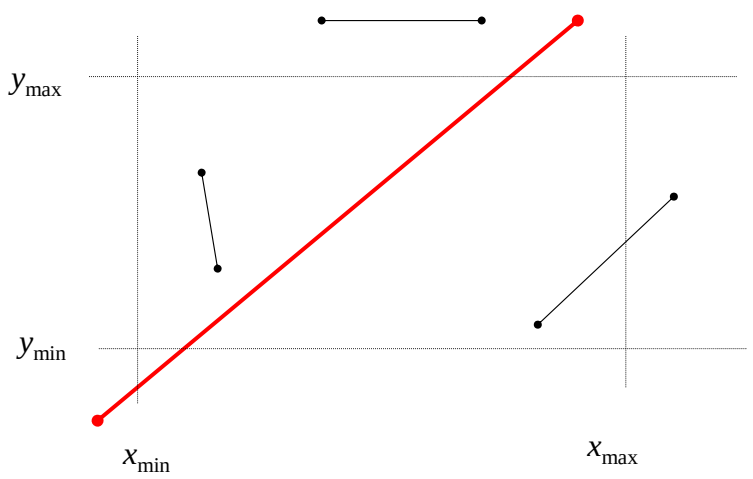
4

## Cohen-Sutherland

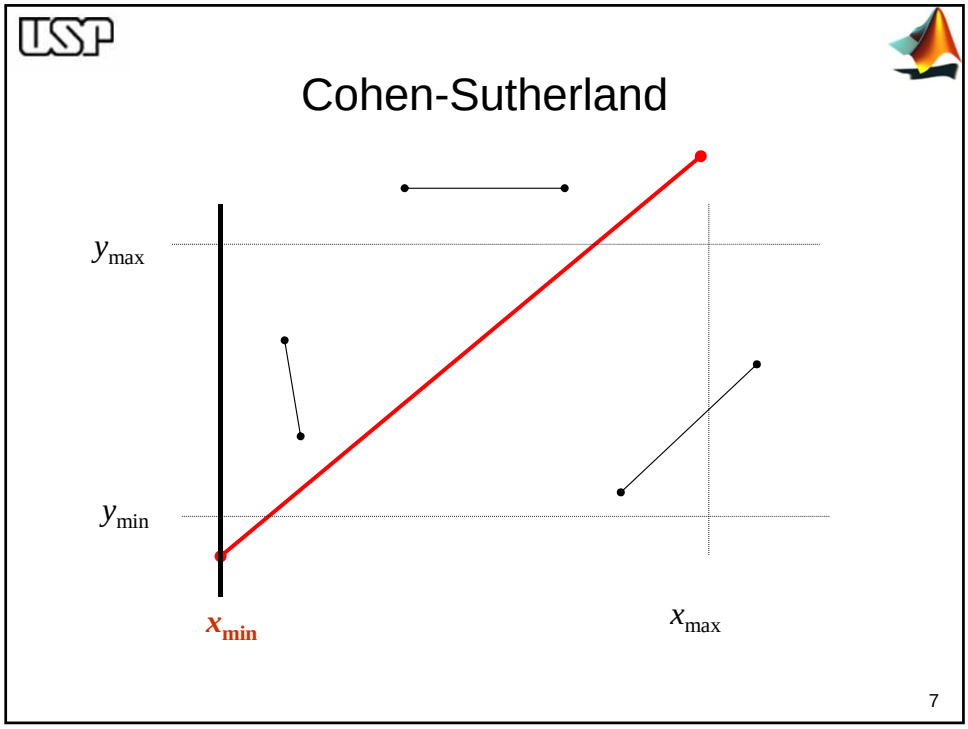
- Vértices do segmento são classificados com relação a cada semi-espaço plano que delimita a janela:
  - $x \geq x_{min}$ e $x \leq x_{max}$ ; $y \geq y_{min}$ e $y \leq y_{max}$
- Se ambos os vértices são classificados como fora, descartar o segmento (totalmente invisível).
- Se ambos são classificados como dentro, testar o próximo semi-espaço.
- Se um vértice dentro e outro fora, computar o ponto de interseção $Q$ e continuar o algoritmo com o segmento recortado ($P_1$-$Q$ ou $P_2$-$Q$).

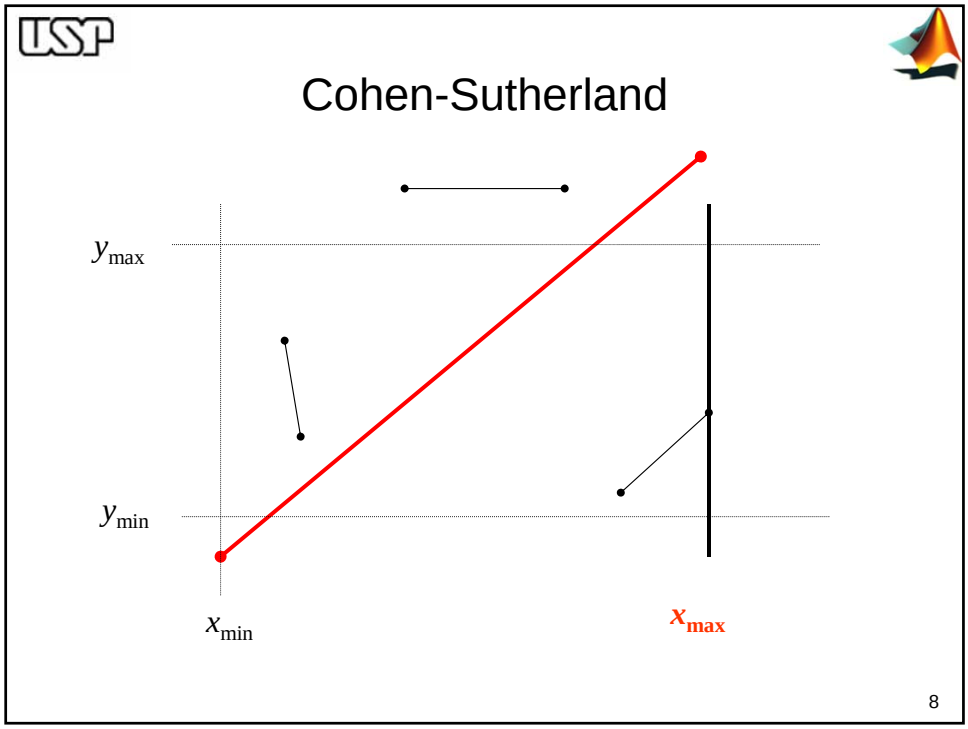
5

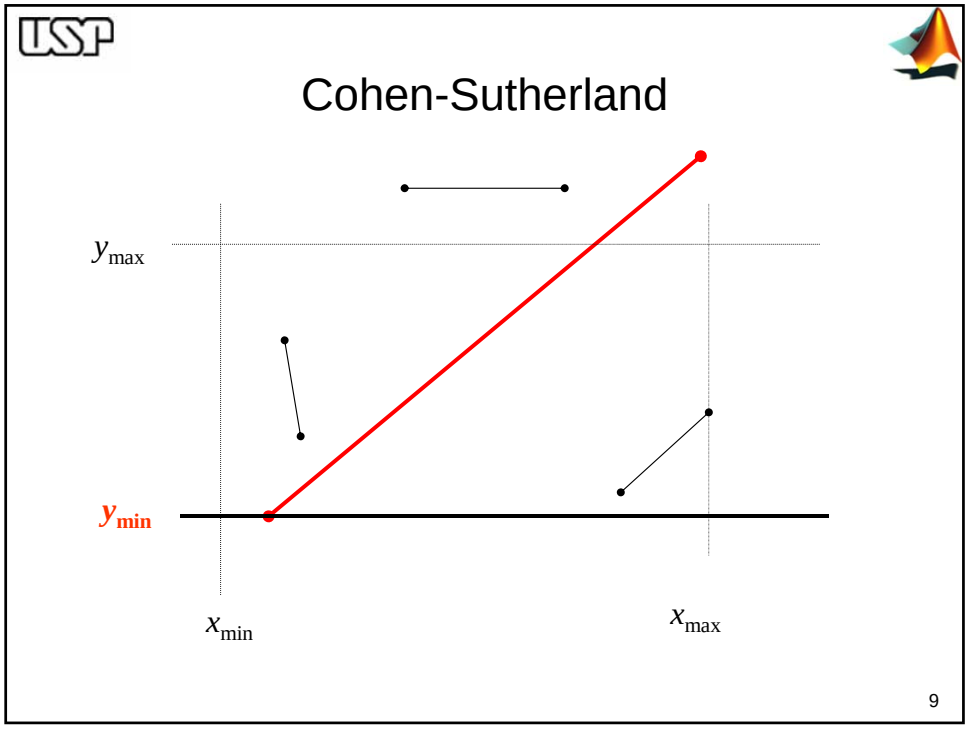
## Cohen-Sutherland

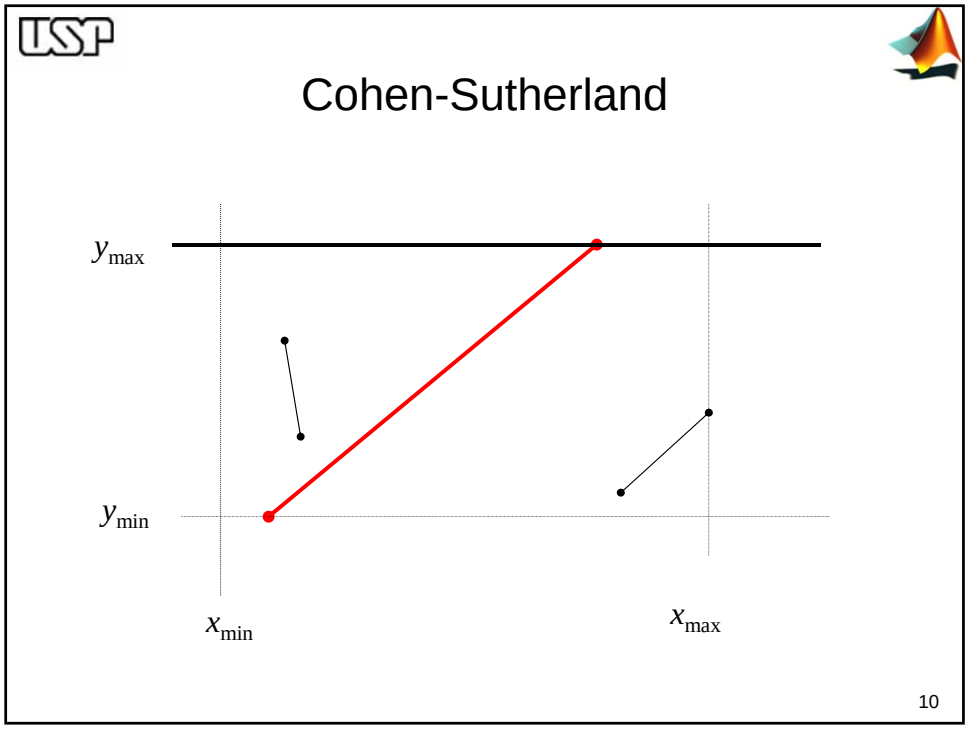



6


Cohen-Sutherland
$y_{max}$
$y_{min}$
$x_{min}$
$x_{max}$
7


Cohen-Sutherland
$y_{max}$
$y_{min}$
$x_{min}$
$x_{max}$
8


Cohen-Sutherland
$y_{max}$
$y_{min}$
$x_{min}$
$x_{max}$
9


Cohen-Sutherland
$y_{max}$
$y_{min}$
$x_{min}$
$x_{max}$
10

## Algoritmo de Liang-Barsky

- É mais eficiente visto que não precisamos computar pontos de interseção irrelevantes.
- O funcionamento deste algoritmo consiste em realizar vários testes comparativos de domínio antes de se calcular os pontos de interseção.
- Refinamento que consiste em representar a reta em forma paramétrica, ou seja:

$$x = x_1 + t \cdot (x_2 - x_1) = x_1 + t \cdot \Delta x, \quad \text{onde } \Delta x = x_2 - x_1$$
$$y = y_1 + t \cdot (y_2 - y_1) = y_1 + t \cdot \Delta y, \quad \text{onde } \Delta y = y_2 - y_1$$

- Porção da reta não recortada deve satisfazer:

$$x_{\min} \le x_1 + t \cdot \Delta x \le x_{\max}$$
$$y_{\min} \le y_1 + t \cdot \Delta y \le y_{\max}$$

11

## Algoritmo de Liang-Barsky

- A partir dessas duas expressões: $\begin{cases} x_{\min} \le x_1 + t \cdot \Delta x \le x_{\max} \\ y_{\min} \le y_1 + t \cdot \Delta y \le y_{\max} \end{cases}$

  obtêm-se as 4 desigualdades seguintes, as quais representam as fronteiras de seus Semi-Espaços (SE):

  1) $-t_1 \cdot \Delta x \le x_1 - x_{\min}$, referente ao SE "Esquerdo"
  2) $t_2 \cdot \Delta x \le x_{\max} - x_1$, referente ao SE "Direito"
  3) $-t_3 \cdot \Delta y \le y_1 - y_{\min}$, referente ao SE "Inferior"
  4) $t_4 \cdot \Delta y \le y_{\max} - y_1$, referente ao SE "Superior"

- Linha infinita intercepta semi-espaços planos para os seguintes valores do parâmetro *t*:

$$t_k = \frac{q_k}{p_k} \quad \begin{cases} k = 1 \Rightarrow p_1 = -\Delta x \; ; \; q_1 = x_1 - x_{\min} \\ k = 2 \Rightarrow p_2 = \Delta x \; ; \; q_2 = x_{\max} - x_1 \\ k = 3 \Rightarrow p_3 = -\Delta y \; ; \; q_3 = y_1 - y_{\min} \\ k = 4 \Rightarrow p_4 = \Delta y \; ; \; q_4 = y_{\max} - y_1 \end{cases}$$

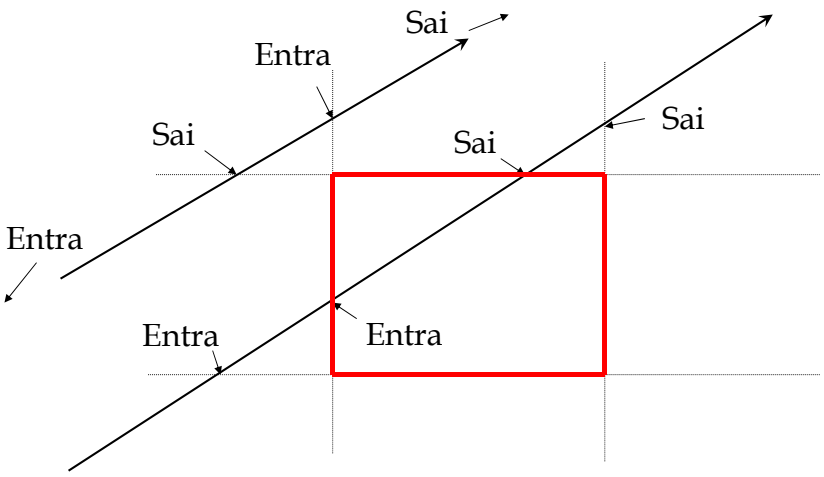
12

## Algoritmo de Liang-Barsky

- Se $p_k < 0$, à medida que $t$ aumenta, reta **entra** no semi-espaço plano;
- Se $p_k > 0$, à medida que $t$ aumenta, reta **sai** do semi-espaço plano;
- Se $p_k = 0$, reta é paralela ao semi-espaço plano (recorte é trivial);
- Se existe um segmento da reta dentro do retângulo, classificação dos pontos de interseção deve ser "**entra, entra, sai, sai**".

13

## Algoritmo de Liang-Barsky

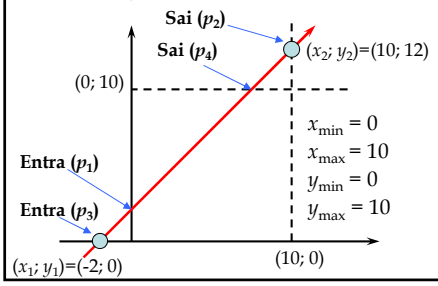



14

# Algoritmo de Liang-Barsky

$$t_k = \frac{q_k}{p_k} \begin{cases} k=1 \Rightarrow p_1 = -\Delta x \;;\; q_1 = x_1 - x_{\min} & \text{(Esquerdo)} \\ k=2 \Rightarrow p_2 = \Delta x \;;\; q_2 = x_{\max} - x_1 & \text{(Direito)} \\ k=3 \Rightarrow p_3 = -\Delta y \;;\; q_3 = y_1 - y_{\min} & \text{(Inferior)} \\ k=4 \Rightarrow p_4 = \Delta y \;;\; q_4 = y_{\max} - y_1 & \text{(Superior)} \end{cases}$$

- Uma linha paralela a um dos lados da *viewport* tem $p_k$=0.
- Quando $p_k < 0$, a linha está entrando na *viewport*.
- Quando $p_k > 0$, a linha está saindo da *viewport*.
- Para $p_k \neq 0$, tem-se que o ponto de interseção é dado por $t_k = q_k/p_k$.




$\Delta x = x_2 - x_1 = 10 - (-2) = 12$ ;
$\Delta y = y_2 - y_1 = 12 - 0 = 12$

$p_1 = -12 \Rightarrow q_1 = -2$ (Entra no SE esquerdo)
$p_2 = 12 \Rightarrow q_2 = 12$ (Sai do SE direito)
$p_3 = -12 \Rightarrow q_3 = 0$ (Entra no SE inferior)
$p_4 = 12 \Rightarrow q_4 = 10$ (Sai do SE superior)

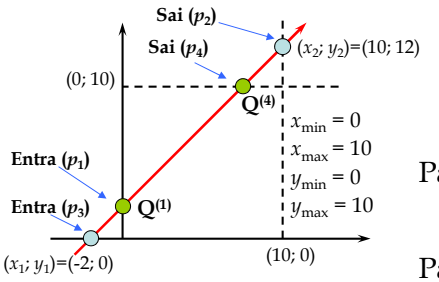
Assim, a reta corta a *viewport*, pois tem-se a sequência **"entra, entra, sai, sai"**

15

# Algoritmo de Liang-Barsky

- Para o exemplo, calcula-se então os parâmetros $t_k$ que possibilitam obter os pontos de interseção da reta com a respectiva *viewport*.
- Neste caso, utiliza-se os valores de $t_k$ correspondentes as interseções $p_1$ e $p_4$ .

Para $k = 1$ ➔ $t_1 = q_1/p_1 = (x_1\text{-}x_{min})/\text{-}\Delta x = \text{-}2/\text{-}12$ ➔ $t_1 = 1/6$
Para $k = 4$ ➔ $t_4 = q_4/p_4 = (y_{max}\text{-}y_1)/\Delta y = 10/12$ ➔ $t_4 = 5/6$




$\Delta x = x_2 - x_1 = 10 - (-2) = 12$
$\Delta y = y_2 - y_1 = 12 - 0 = 12$

$$x = x_1 + t \cdot \Delta x \Rightarrow x = -2 + 12 \cdot t$$
$$y = y_1 + t \cdot \Delta y \Rightarrow y = 0 + 12 \cdot t$$

Para $t = t_1 = 1/6$ $\begin{cases} x = 0 \\ y = 2 \end{cases}$ ⇨ $Q^{(1)} = (0; 2)$

Para $t = t_4 = 5/6$ $\begin{cases} x = 8 \\ y = 10 \end{cases}$ ⇨ $Q^{(4)} = (8; 10)$

## Comparação

- Cohen-Sutherland
  - *Clipping* repetitivo, alto custo computacional.
  - Melhor utilização quando a maioria das linhas se encaixam nos casos triviais de aceitação e rejeição.
- Liang-Barsky
  - Cálculo de *t* para as interseções (baixo custo computacional).
  - Computação dos pontos (*x*,*y*) de corte é feita apenas uma vez.
  - Melhor usado quando a maioria das linhas precisam ser recortadas.

17

## Algoritmos de Visibilidade

- Visibilidade é um problema complexo que não tem *uma* solução "ótima".
  - O que é ótima?
    - Pintar apenas as superfícies visíveis?
    - Pintar a cena em tempo mínimo?
  - Coerência no tempo?
    - Cena muda?
    - Objetos se movem?
  - Qualidade é importante?
    - *Antialiasing*
  - Aceleração por Hardware?

18

## Complexidade do Problema

- Fatores que influenciam o problema:
  - Número de pixels
    - Em geral, procura-se minimizar o número total de pixels pintados.
    - Resolução da imagem.
  - Número de objetos
    - Técnicas de "*culling*" descarta objetos desnecessários.
    - Recorte pode aumentar o número de objetos.

19

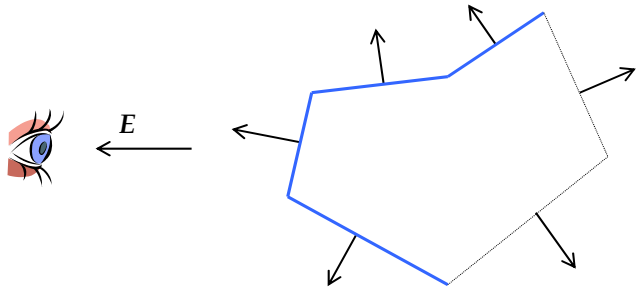
## *Backface Culling*

- Hipótese: cena é composta de objetos poliédricos fechados.
- Podemos reduzir o número de faces aproximadamente à metade.
  - Faces de trás não precisam ser pintadas.
- Como determinar se a face é de trás?

$\vec{N} \cdot \vec{E} < 0 \Rightarrow$ Face da Trás

$\vec{N} \cdot \vec{E} > 0 \Rightarrow$ Face da Frente

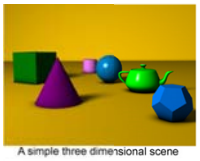
$$\cos(\theta) = \frac{\vec{N} \cdot \vec{E}}{|\vec{N}| \cdot |\vec{E}|}$$

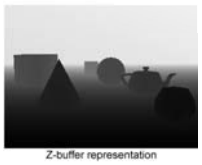



20

# Z-Buffer

- Método que opera no espaço da imagem.
- Manter para cada pixel um valor de profundidade (*z-buffer* ou *depth buffer*).
- Início da renderização:
  - *Buffer* de cor = cor de fundo
  - *z-buffer* = profundidade máxima
- Durante a rasterização de cada polígono, cada pixel passa por um *teste de profundidade.*
  - Se a profundidade do pixel for menor que a registrada no *z-buffer,* então:
    - Pintar o pixel e atualizar o buffer de cor
    - Atualizar o buffer de profundidade
  - Caso contrário, ignorar.

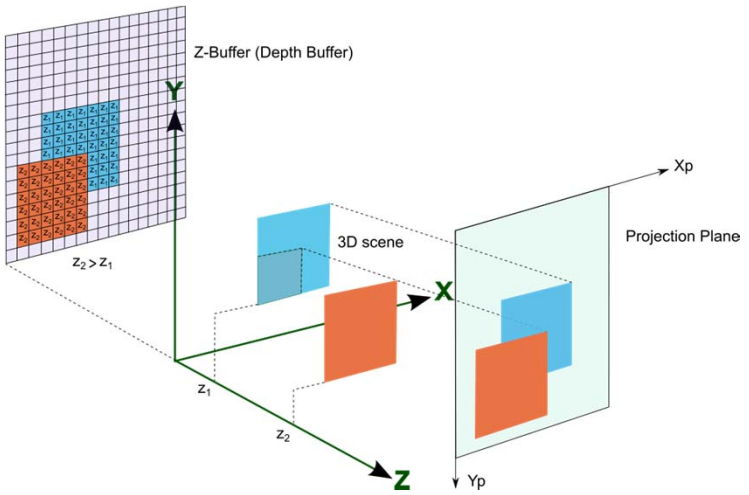

A simple three dimensional scene


Z-buffer representation



# Z-Buffer

- Representação:



## Z-Buffer

- **Vantagens:**
  - Simples e muito comumente implementado em Hardware.
  - Objetos podem ser desenhados em qualquer ordem.
- **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-serrilhado.
    - É preciso ter informações sobre os vários polígonos que cobrem cada pixel.
    - Z_buffer_Aplicado.avi
    - Z_buffer_Conceito.avi



## Algoritmo “Scan-Line”

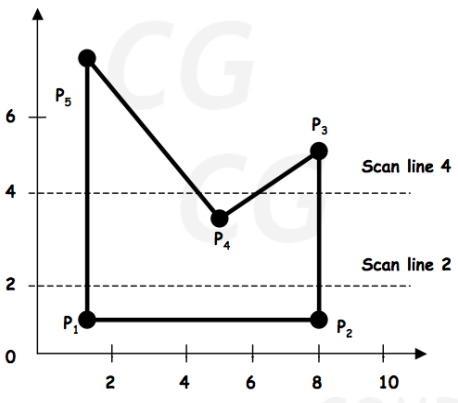
- Ideia é aplicar o algoritmo de rasterização de polígonos a todos os polígonos da cena simultaneamente.
- Explora coerência de visibilidade.
- Em sua concepção original requer que polígonos se interceptem apenas em vértices ou arestas:
  - Pode ser adaptado para lidar com faces que se interceptam.
  - Pode até mesmo ser estendido para rasterizar sólidos.

## Algoritmo “Scan-Line”

- Ordena-se todas as arestas de todos os polígonos por $y_{min}$ .
- Para cada plano de varredura $y$, fazer:
  - Para cada polígono, fazer:
    - Determinar intervalos $x_i$ de interseção com plano de varredura.
  - Renderizar resultado da linha de varredura.

25

## Algoritmo “Scan-Line”




**SCAN LINE 2**

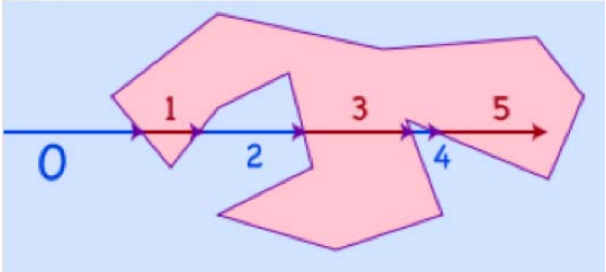
*x < 1 → fora do polígono*
*1 ≤ x ≤ 8 → dentro do polígono*
*x > 8 → fora do polígono*

**SCAN LINE 4**

*x < 1 → fora do polígono*
*1 ≤ x ≤ 4 → dentro do polígono*
*4 < x < 6 → fora do polígono*
*6 ≤ x ≤ 8 → dentro do polígono*
*x > 8 → fora do polígono*

26

## Algoritmo “Scan-Line”




Quando o número de arestas do polígono interceptadas é ímpar, está dentro; quando é par, está fora.

27

## Algoritmo “Scan-Line”

- Vantagens
  - Algoritmo flexível que explora a coerência entre pixels de uma mesma linha de varredura.
  - Razoável independência da resolução da imagem.
  - Filtragem e *antialiasing* podem ser incorporados com um pouco de trabalho.
  - Pinta cada pixel apenas uma vez.
  - Razoavelmente imune a erros de quantização em *z*.
- Desvantagens
  - Coerência entre linhas de varredura não é explorada.
    - Polígonos invisíveis são descartados múltiplas vezes.
  - Relativa complexidade.
  - Não muito próprio para implementação em Hardware.

28