



Introdução à Computação Gráfica

Visibilidade e Recorte

Adaptação: João Paulo Pereira

António Costa

Autoria: Claudio Esperança

Paulo Roma Cavalcanti



O Problema de Visibilidade

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

Espaço do Objecto x Espaço da Imagem

- Métodos que trabalham no espaço do objecto
 - ◆ 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 rectas
 - Recorte de polígonos
 - Algoritmos de visibilidade que utilizam recorte
 - Algoritmo do pintor
 - BSP-trees
 - Algoritmo de recorte sucessivo
 - Volumes de sombra

Espaço do Objecto x Espaço da Imagem

- Métodos que trabalham no espaço da imagem
 - ◆ Entrada é vectorial e saída é matricial
 - ◆ Dependente da resolução da imagem
 - ◆ Visibilidade determinada apenas em pontos (pixels)
 - ◆ Podem aproveitar aceleração por hardware
 - ◆ Exemplos:
 - Z-buffer
 - Algoritmo de Warnock
 - Mapas de sombra

Recorte (*Clipping*)

- Problema definido por
 - ◆ Geometria a ser recortada
 - Pontos, rectas, planos, curvas, superfícies
 - ◆ Restrições de recorte
 - Janela (2D)
 - Volume de visibilidade
 - Frustum (tronco de pirâmide)
 - Paralelipípedo
 - Polígonos
 - Convexos
 - Genéricos (côncavos, com buracos, etc)
- Resultado depende da geometria
 - ◆ Pontos: valor booleano (visível / não visível)
 - ◆ Rectas: segmento de recta ou coleção de segmentos de recta
 - ◆ Planos: polígono ou coleção de polígonos

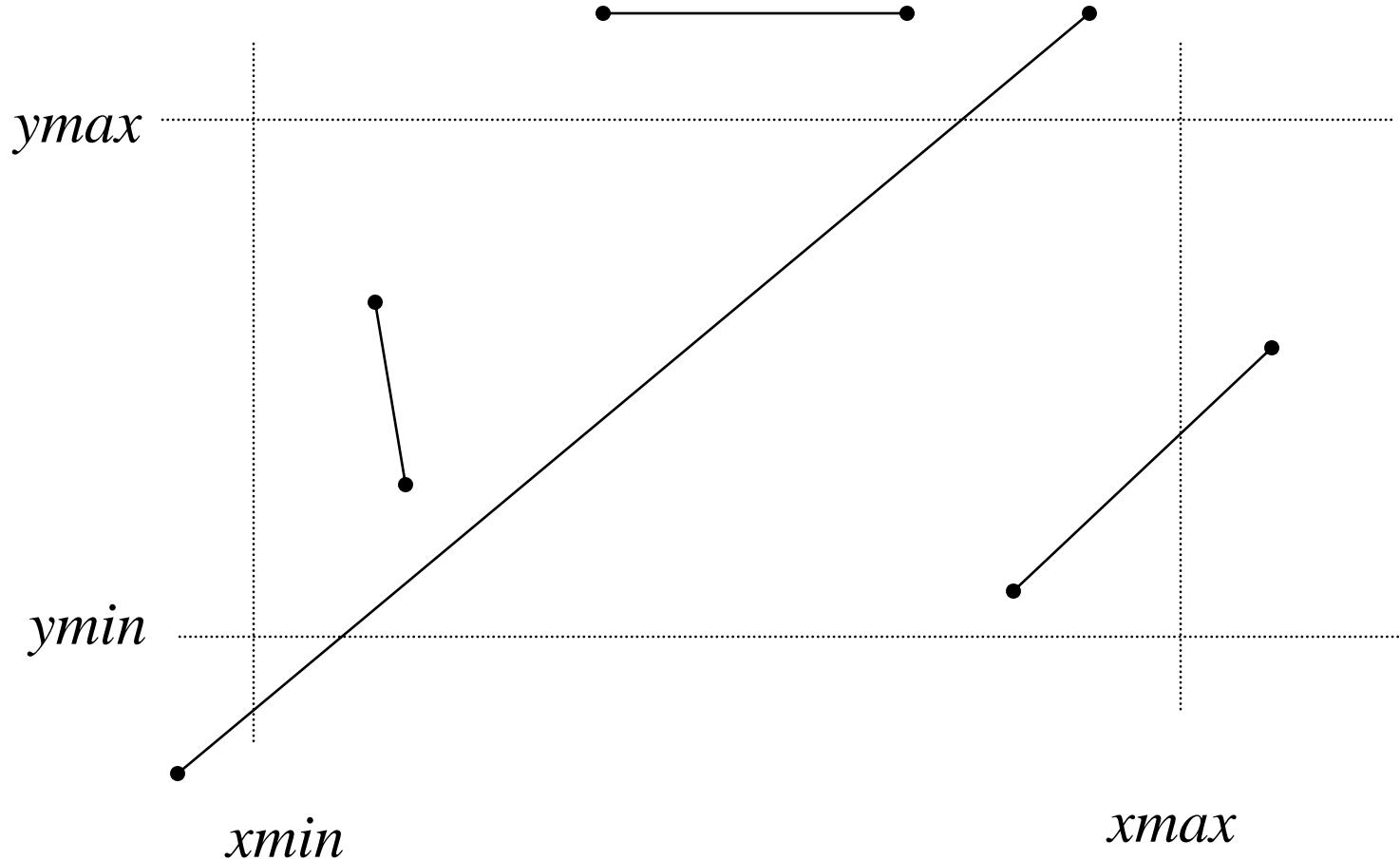
Recorte de Segmento de Recta x Rectângulo

- Problema clássico 2D
- Entrada:
 - ◆ Segmento de recta $P_1 - P_2$
 - ◆ Janela alinhada com eixos $(xmin, ymin) - (xmax, ymax)$
- Saída: Segmento recortado (possivelmente nulo)
- Variantes
 - ◆ Cohen-Sutherland
 - ◆ Liang-Barksy / Cyrus-Beck
 - ◆ Nicholl-Lee-Nicholl

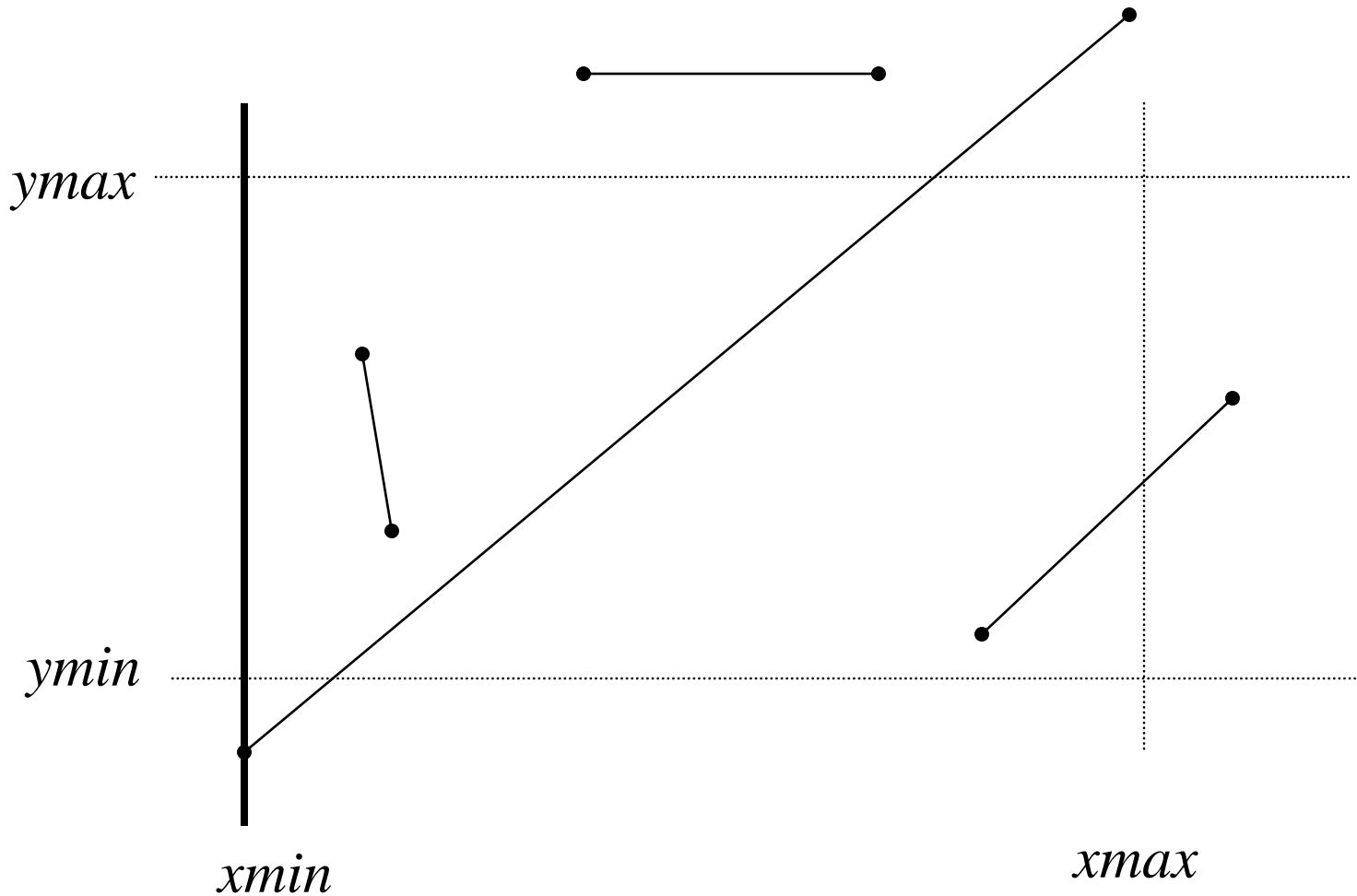
Cohen-Sutherland

- Vértices do segmento são classificados com relação a cada semi-espaco plano que delimita a janela
 - ◆ $x \geq xmin$ e $x \leq xmax$ e $y \geq ymin$ e $y \leq ymax$
- Se ambos os vértices classificados como fora, descartar o segmento (totalmente invisível)
- Se ambos classificados como dentro, testar o próximo semi-espaco
- Se um vértice dentro e outro fora, computar o ponto de intersecção Q e continuar o algoritmo com o segmento recortado (P_1-Q ou P_2-Q)

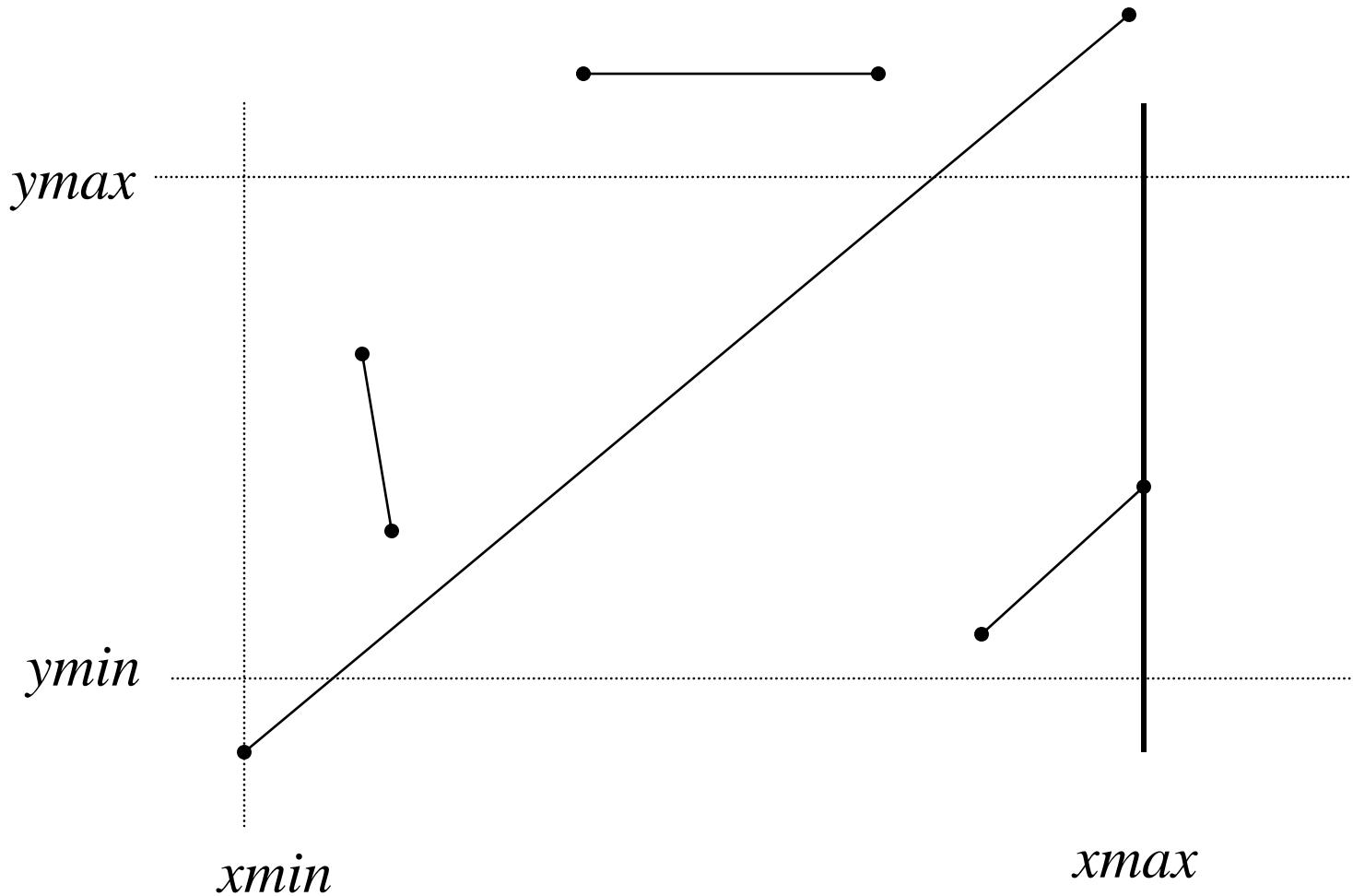
Cohen-Sutherland



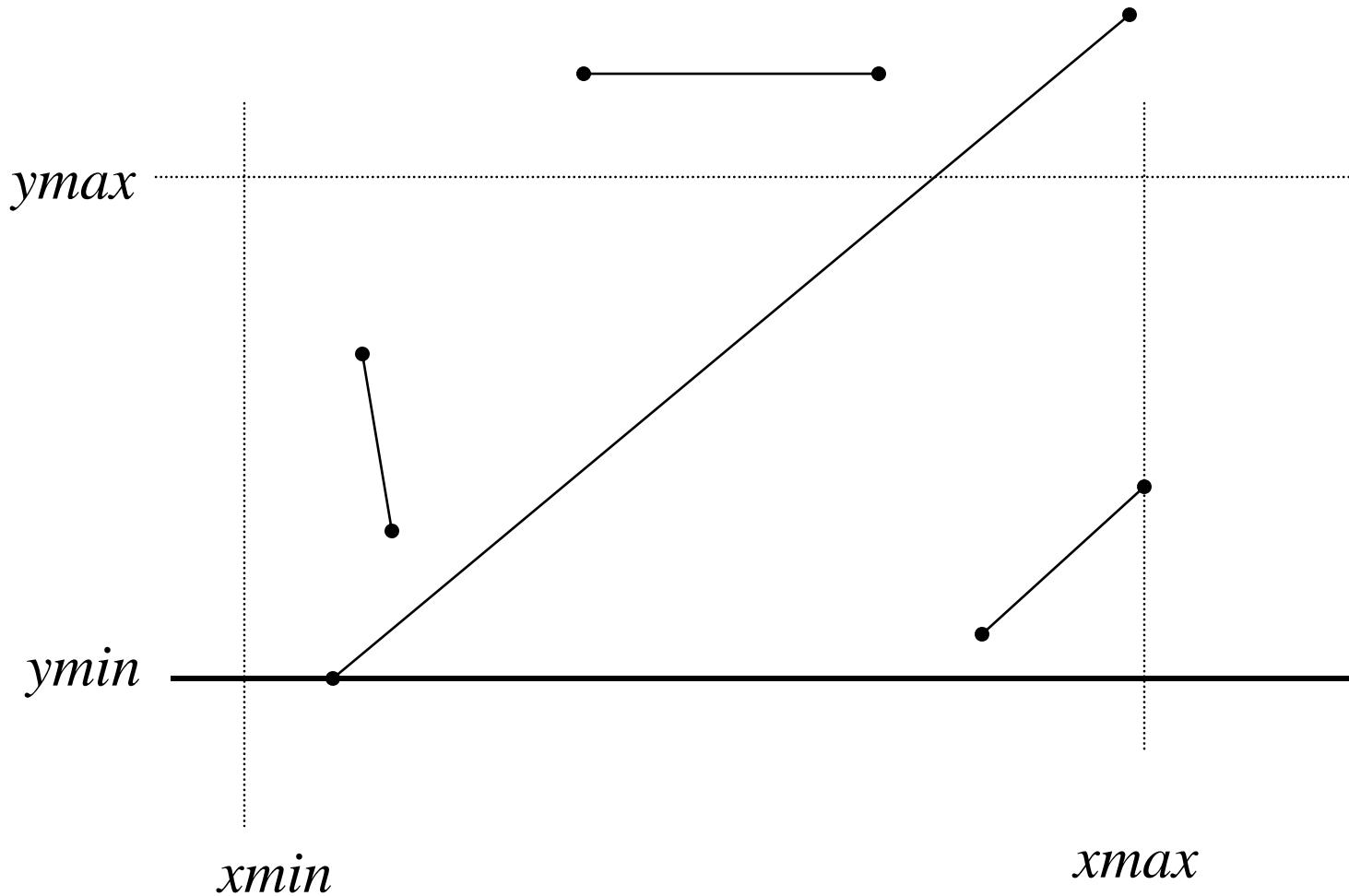
Cohen-Sutherland



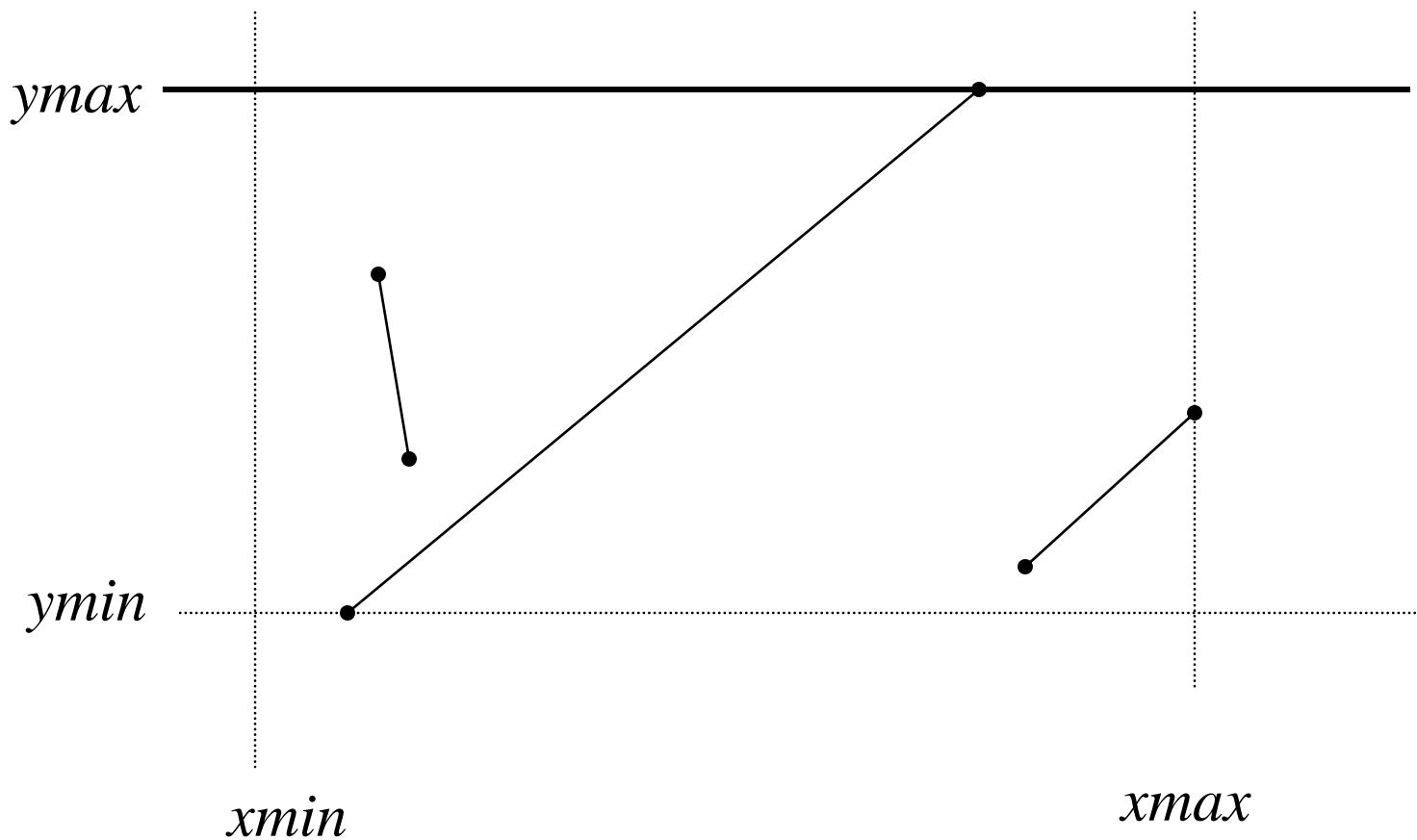
Cohen-Sutherland



Cohen-Sutherland

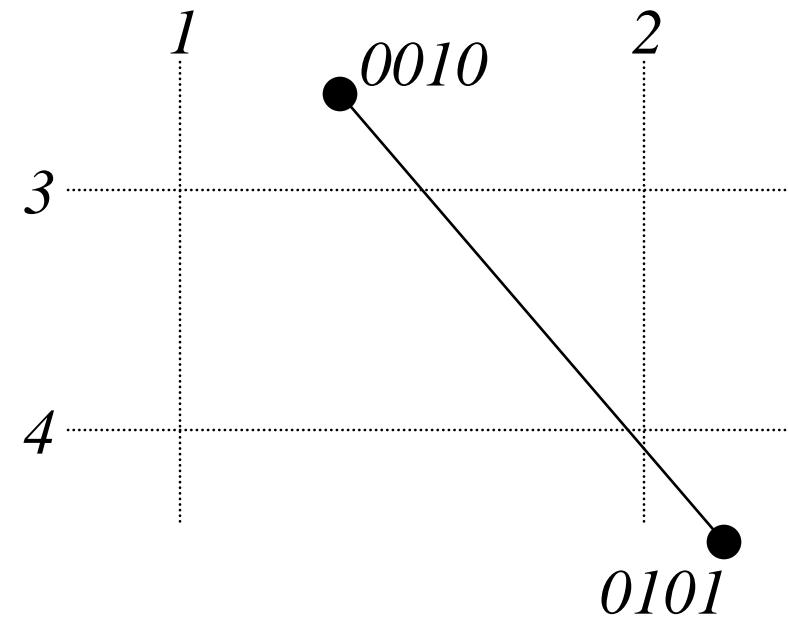


Cohen-Sutherland



Cohen-Sutherland - Detalhes

- Recorte só é necessário se um vértice dentro e outro fora
- Classificação de cada vértice pode ser codificada em 4 bits, um para cada semi-espaco
 - ◆ Dentro = 0 e Fora = 1
- Rejeição trivial:
 - ◆ $\text{Classif}(P_1) \& \text{Classif}(P_2) \neq 0$
- Aceitação trivial:
 - ◆ $\text{Classif}(P_1) \mid \text{Classif}(P_2) = 0$
- Intersecção com quais semi-espacos?
 - ◆ $\text{Classif}(P_1) \wedge \text{Classif}(P_2)$



Algoritmo de *Liang-Barsky*

- Refinamento que consiste em representar a recta na forma paramétrica
- É mais eficiente visto que não é preciso computar pontos de intersecção irrelevantes
- Porção da recta não recortada deve satisfazer

$$x_{\min} \leq x_1 + t \Delta x \leq x_{\max} \quad \Delta x = x_2 - x_1$$

$$y_{\min} \leq y_1 + t \Delta y \leq y_{\max} \quad \Delta y = y_2 - y_1$$

Algoritmo de *Liang-Barsky*

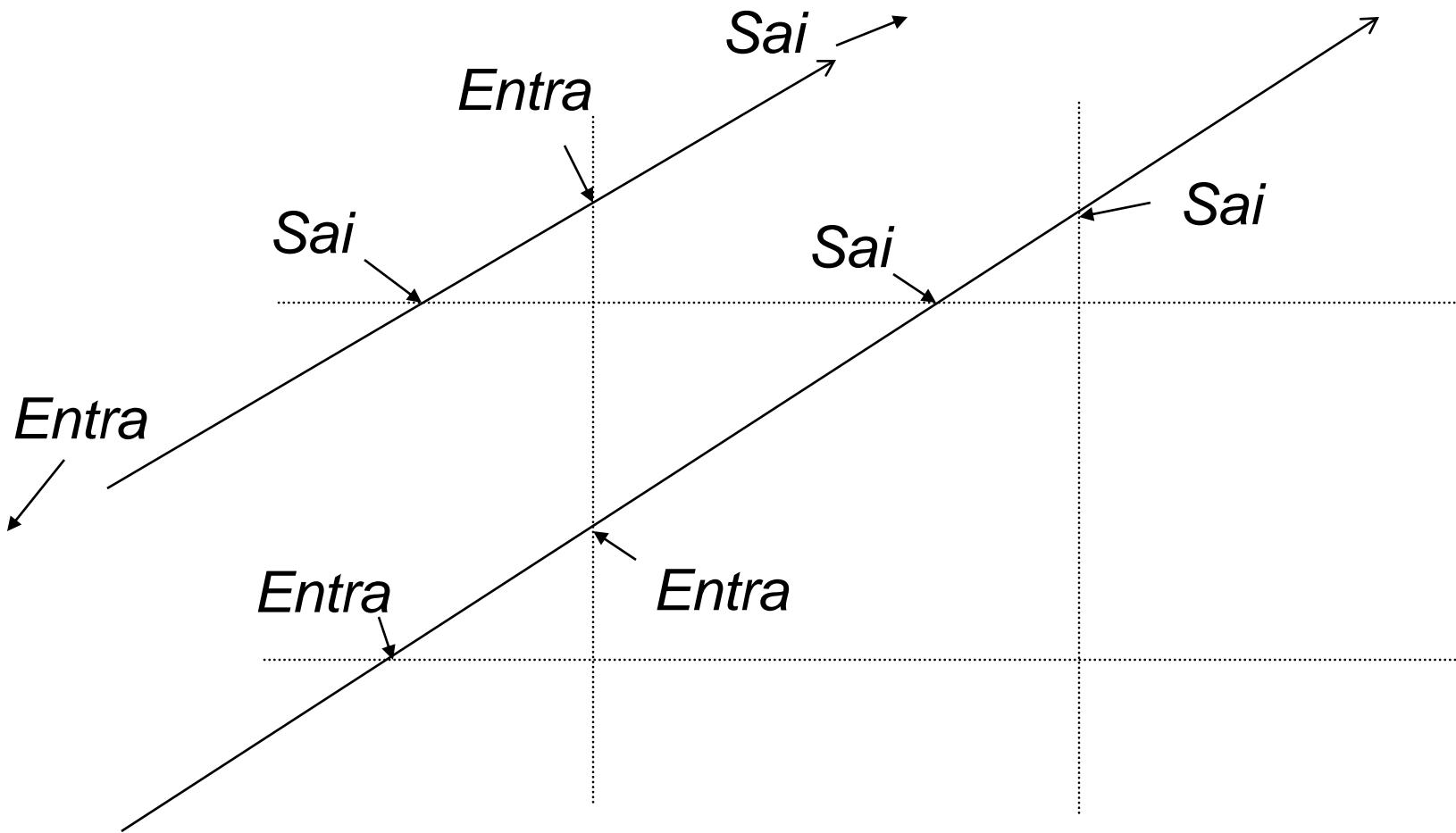
- Linha infinita intercepta semi-espacos planos para os seguintes valores do parâmetro t :

$$t_k = \frac{q_k}{p_k} \quad \text{onde} \quad \begin{aligned} p_1 &= -\Delta x & q_1 &= x_1 - x_{\min} \\ p_2 &= \Delta x & q_2 &= x_{\max} - x_1 \\ p_3 &= -\Delta y & q_3 &= y_1 - y_{\min} \\ p_4 &= \Delta y & q_4 &= y_{\max} - y_1 \end{aligned}$$

Algoritmo de *Liang-Barsky*

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

Algoritmo de *Liang-Barsky*



Liang-Barsky – Pseudo-código

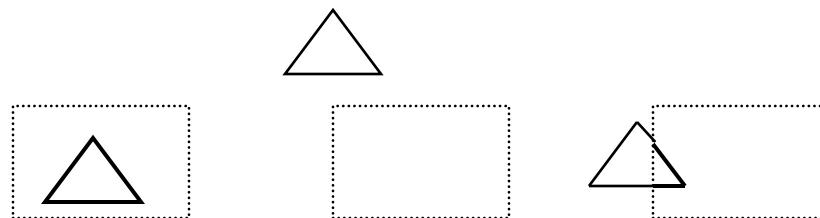
- Computar valores de t para os pontos de intersecção
- Classificar pontos em **entra** ou **sai**
- Vértices do segmento recortado devem corresponder a dois valores de t :
 - ◆ $t_{min} = \max(0, t's \text{ do tipo } \mathbf{entra})$
 - ◆ $t_{max} = \min(1, t's \text{ do tipo } \mathbf{sai})$
- Se $t_{min} < t_{max}$, segmento recortado é não nulo
 - ◆ Computar vértices substituindo os valores de t
- Na verdade, o algoritmo calcula e classifica valores de t um a um
 - ◆ Rejeição precoce
 - Ponto é do tipo **entra** mas $t > 1$
 - Ponto é do tipo **sai** mas $t < 0$

Recorte de Polígono contra Rectângulo

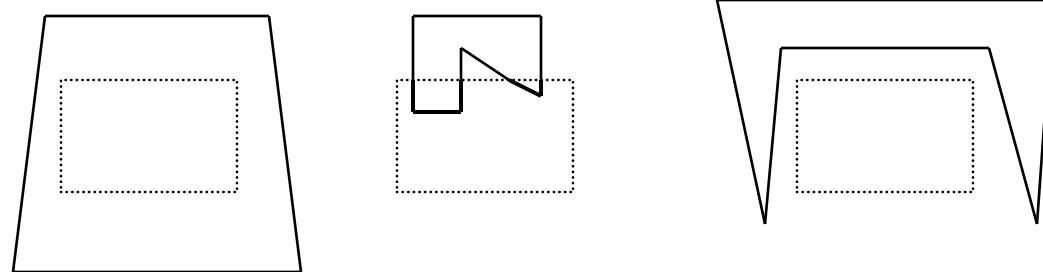
- Inclui o problema de recorte de segmentos de recta
 - ◆ Polígono resultante tem vértices que são
 - Vértices da janela,
 - Vértices do polígono original, ou
 - Pontos de intersecção aresta do polígono/aresta da janela
- Dois algoritmos clássicos
 - ◆ Sutherland-Hodgman
 - Figura de recorte pode ser qualquer polígono convexo
 - ◆ Weiler-Atherton
 - Figura de recorte pode ser qualquer polígono

Recorte de Polígono contra Retângulo

- Casos Simples

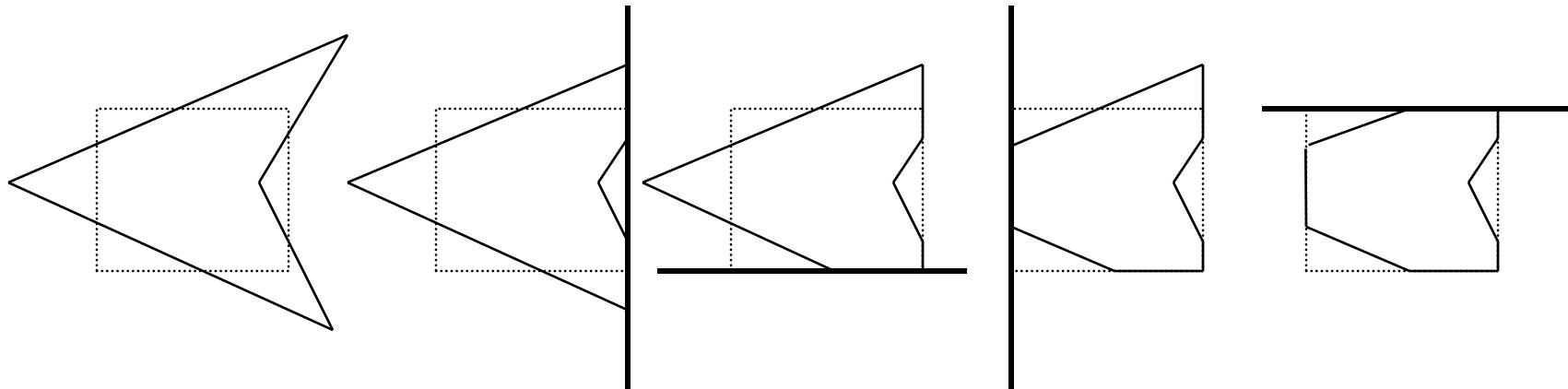


- Casos Complicados



Algoritmo de *Sutherland-Hodgman*

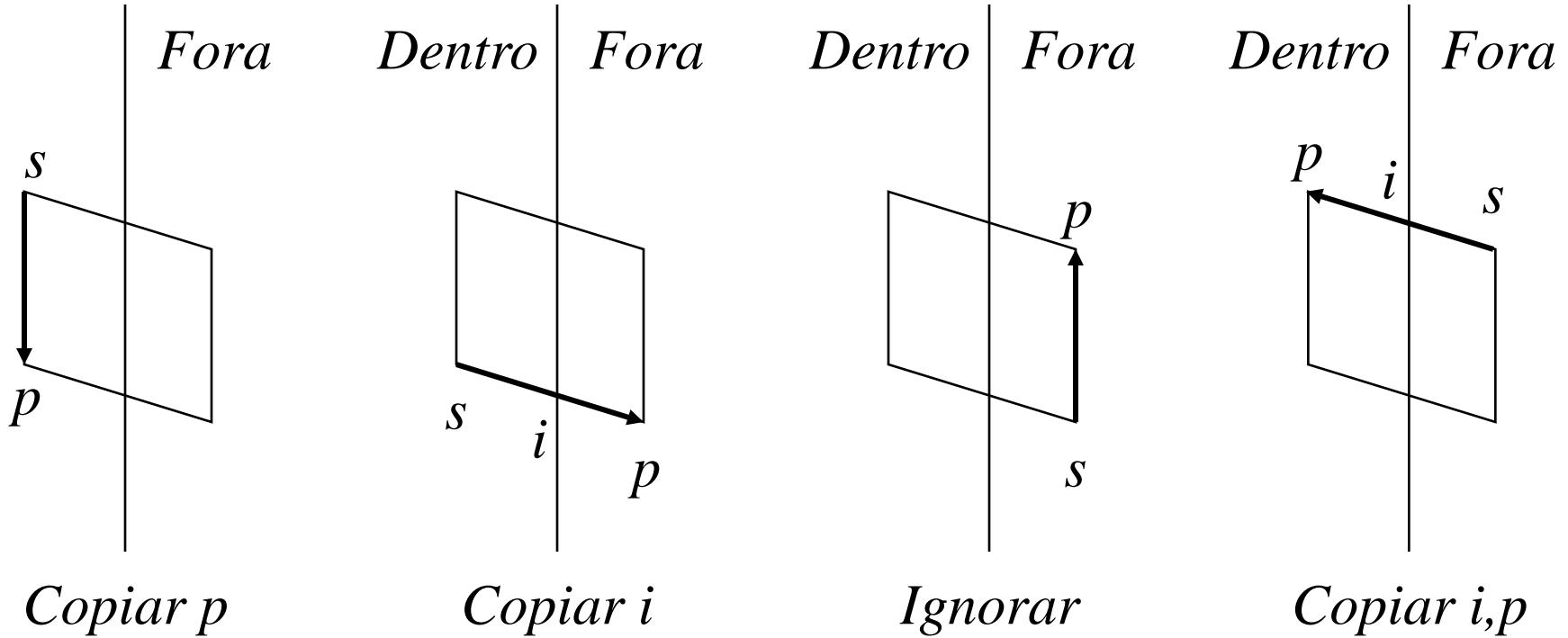
- Ideia é semelhante à do algoritmo de Sutherland-Cohen
 - ◆ Recortar o polígono sucessivamente contra todos os semi-espacos planos da figura de recorte



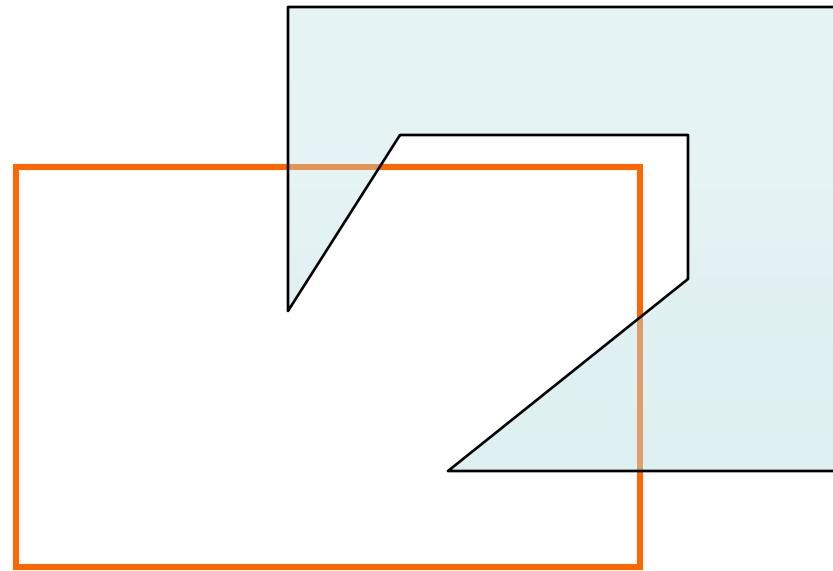
Algoritmo de *Sutherland-Hodgman*

- Polígono é dado como uma lista circular de vértices
- Vértices e arestas são processados em sequência e classificados contra o semi-espaco plano corrente
 - ◆ Vértice:
 - Dentro: copiar para a saída
 - Fora: ignorar
 - ◆ Aresta
 - Intercepta semi-espaco plano (vértice anterior e posterior têm classificações diferentes) : Copiar ponto de intersecção para a saída
 - Não intercepta: ignorar

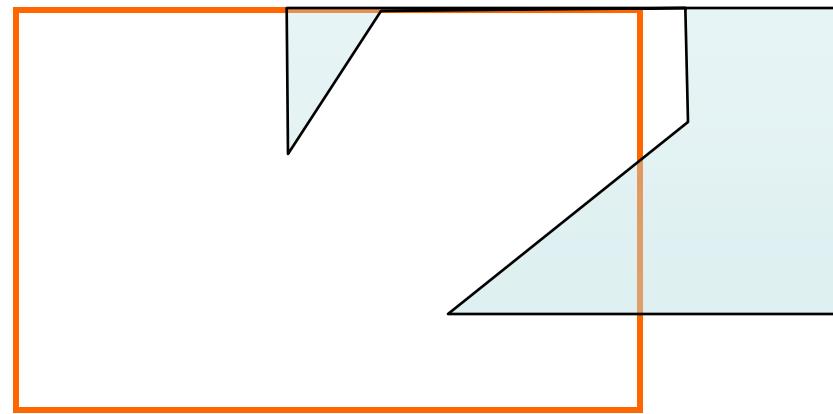
Algoritmo de Sutherland-Hodgman



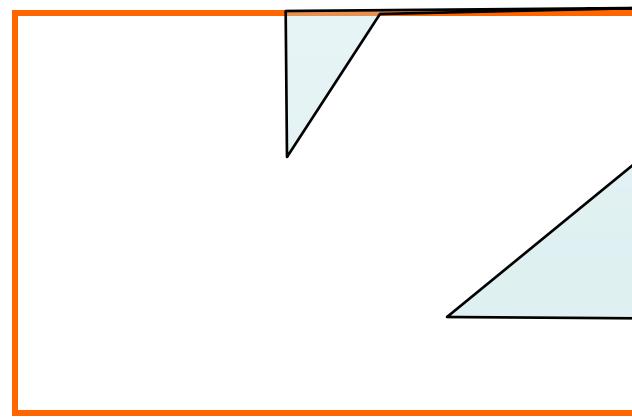
Sutherland-Hodgman – Exemplo



Sutherland-Hodgman – Exemplo

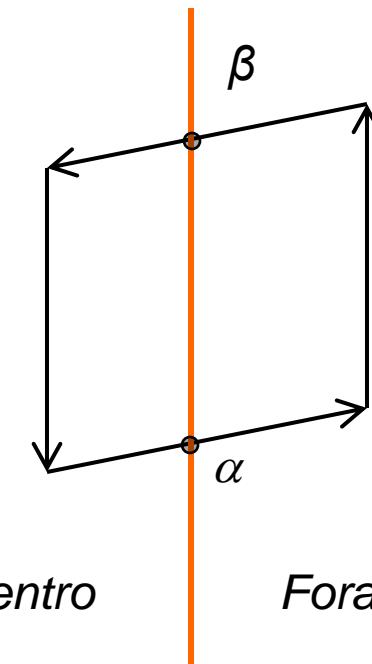


Sutherland-Hodgman – Exemplo

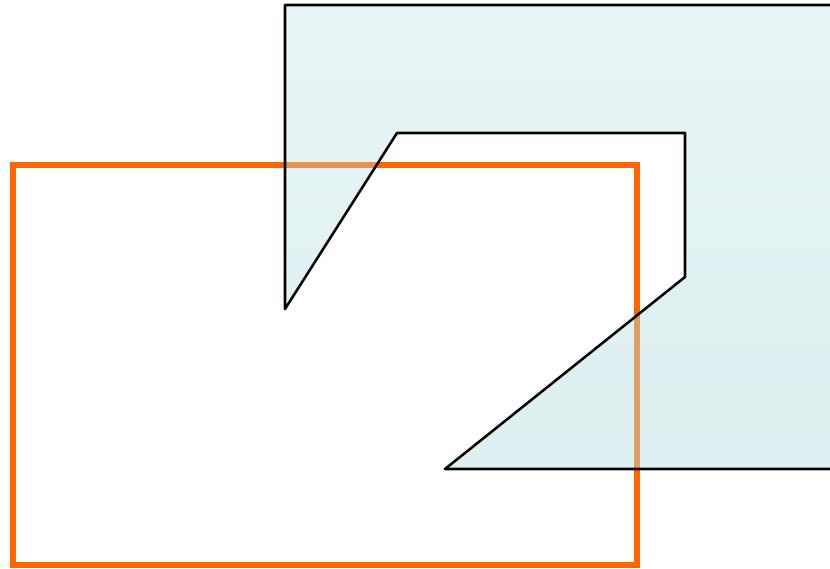


Sutherland Hodgman – Eliminando Arestas Fantasmas

- Distinguir os pontos de intersecção gerados
 - ◆ De dentro para fora: rotular como do tipo α
 - ◆ De fora para dentro: rotular como do tipo β
- Iniciar o percurso de algum vértice “fora”
- Ao encontrar um ponto de intersecção α , ligar com o último β visto
- Resultado pode ter mais de uma componente conexa



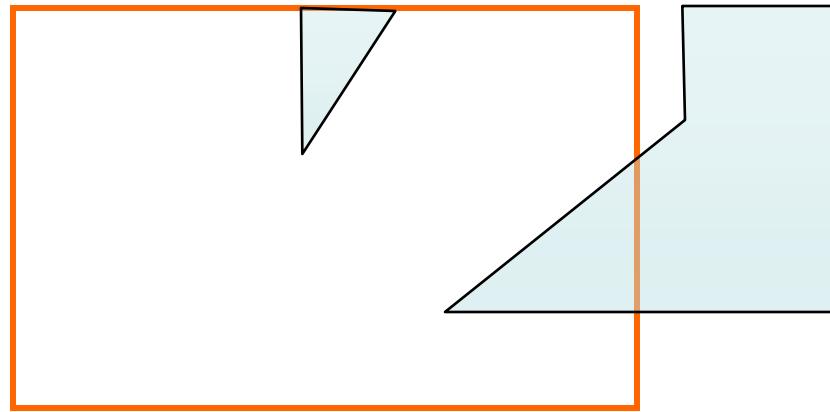
Sutherland Hodgman – Eliminando Arestas Fantasmas – Exemplo



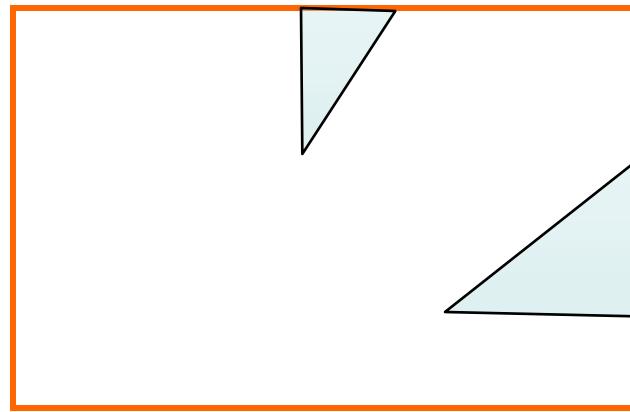
Sutherland Hodgman –

Eliminando Arestas Fantasmas –

Exemplo



Sutherland Hodgman – Eliminando Arestas Fantasmas – Exemplo



Sutherland-Hodgman - Resumo

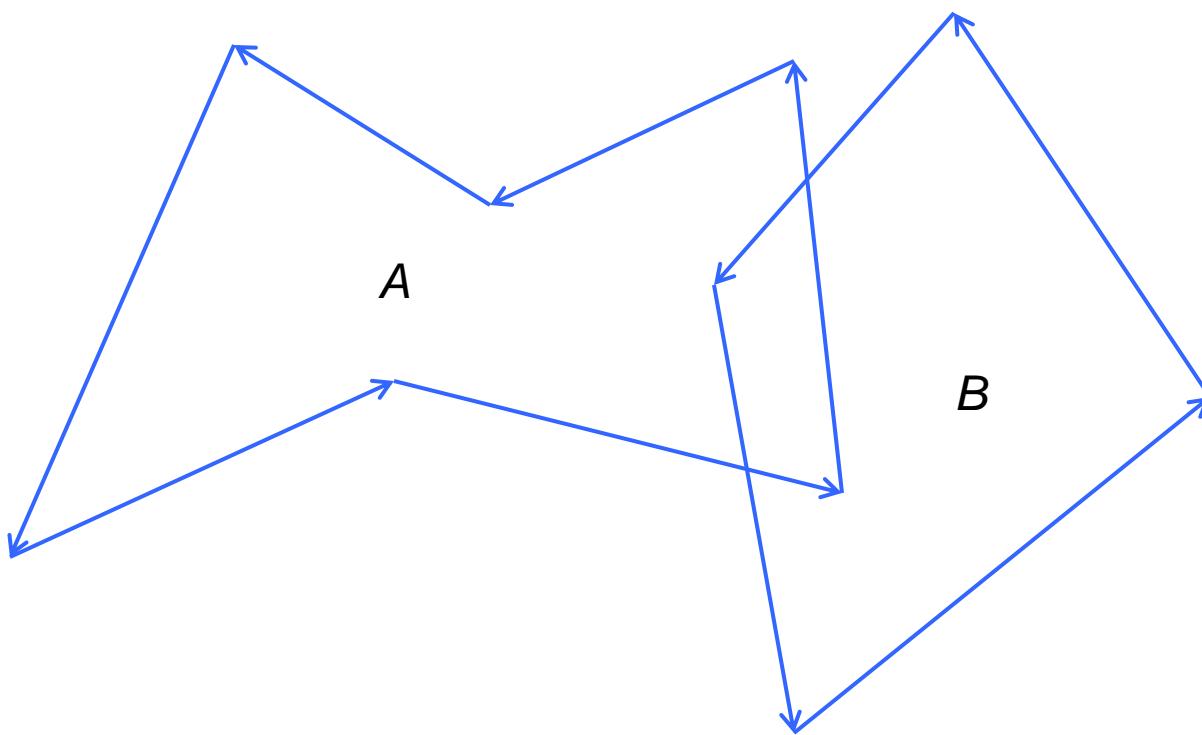
- Facilmente generalizável para 3D
- Pode ser adaptado para implementação em hardware
 - ◆ Cada vértice gerado pode ser passado pelo pipeline para o recorte contra o próximo semi-espaco plano
- Pode gerar arestas “fantasma”
 - ◆ Irrelevante para fins de desenho
 - ◆ Podem ser eliminadas com um pouco mais de trabalho

Algoritmo de Weiler-Atherton

- Recorta qualquer polígono contra qualquer outro polígono
- Pode ser usado para computar operações de conjunto com polígonos
 - ◆ União, Intersecção, Diferença
- Mais complexo que o algoritmo de Sutherland-Hodgman
- Idéia:
 - ◆ Cada polígono divide o espaço em 3 conjuntos
 - Dentro, fora, borda
 - ◆ Borda de cada polígono é “duplicada”
 - Uma circulação corresponde ao lado de dentro e outra ao lado de fora
 - ◆ Nos pontos de intersecção, é preciso “costurar” as 4 circulações de forma coerente

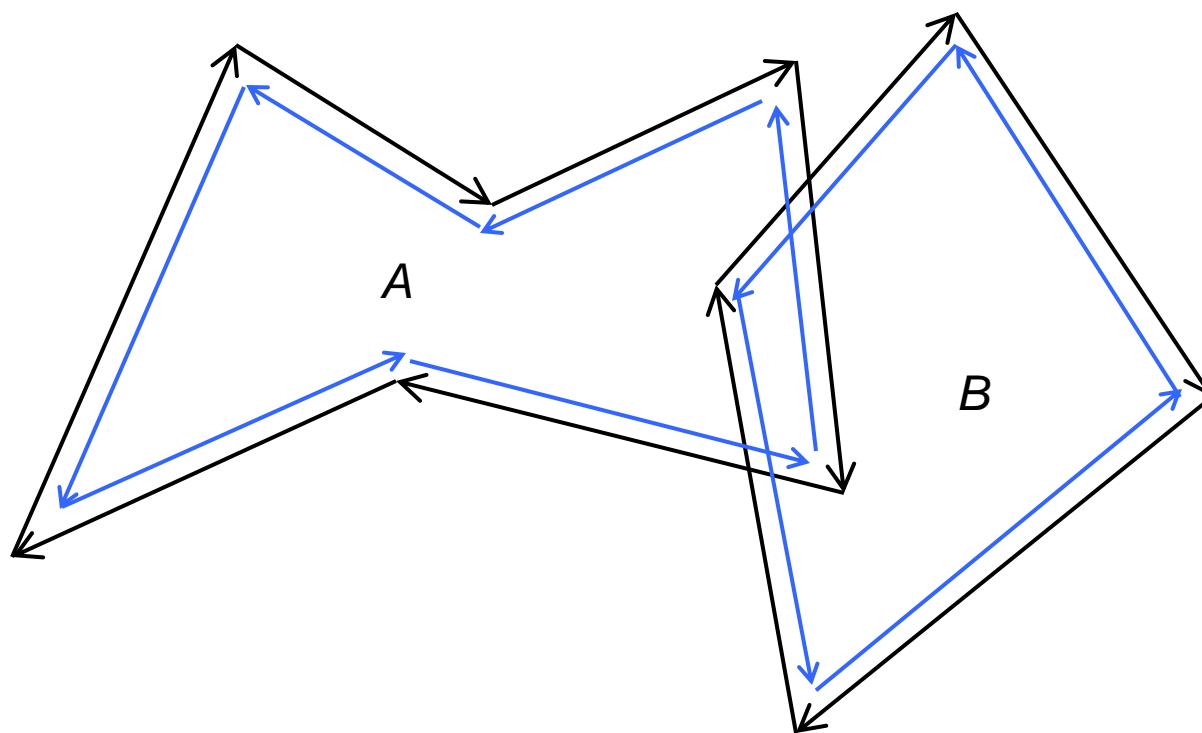
Algoritmo de Weiler-Atherton

*Interior do polígono à esquerda da seta
(circulação anti-horária)*



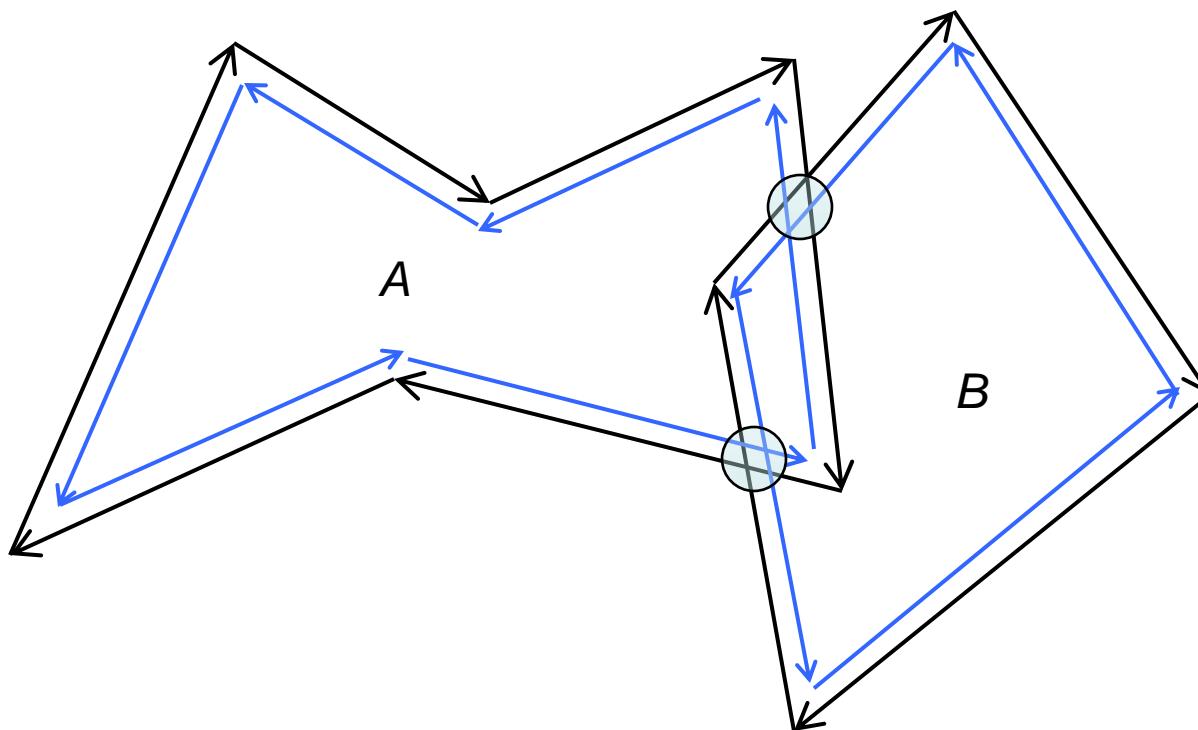
Algoritmo de Weiler-Atherton

*Exterior do polígono à direita da seta
(circulação horária)*



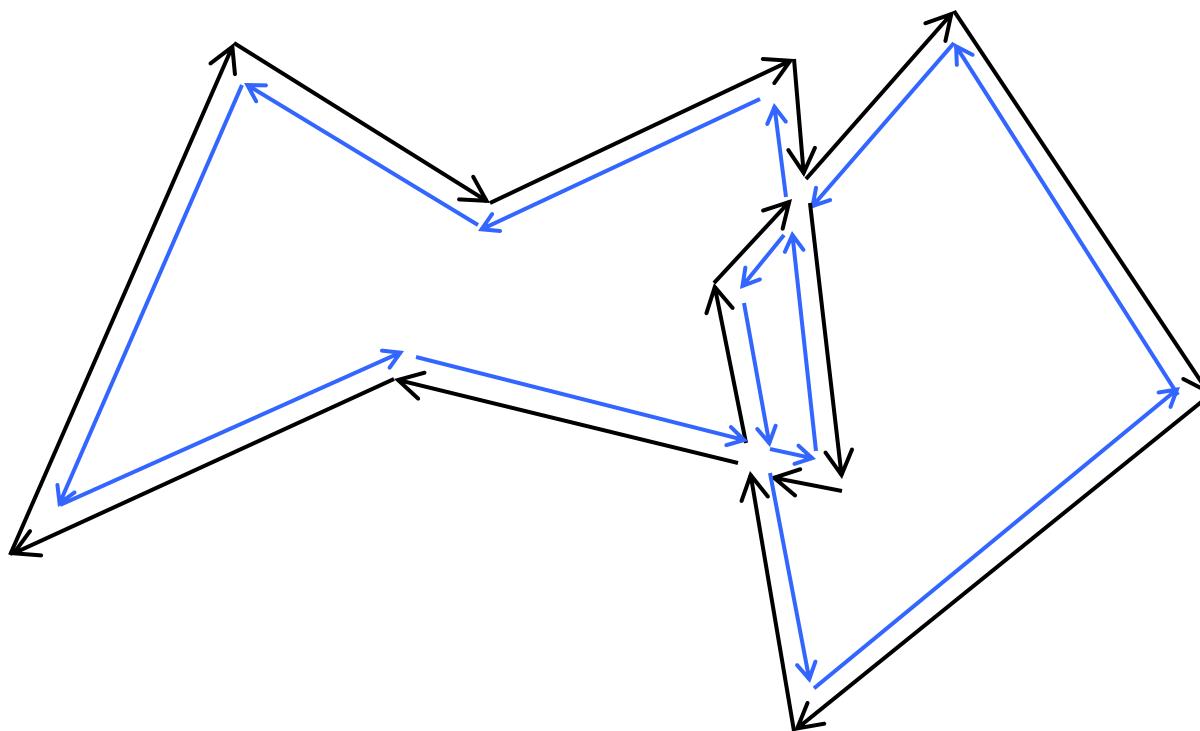
Algoritmo de Weiler-Atherton

Pontos de intersecção são calculados



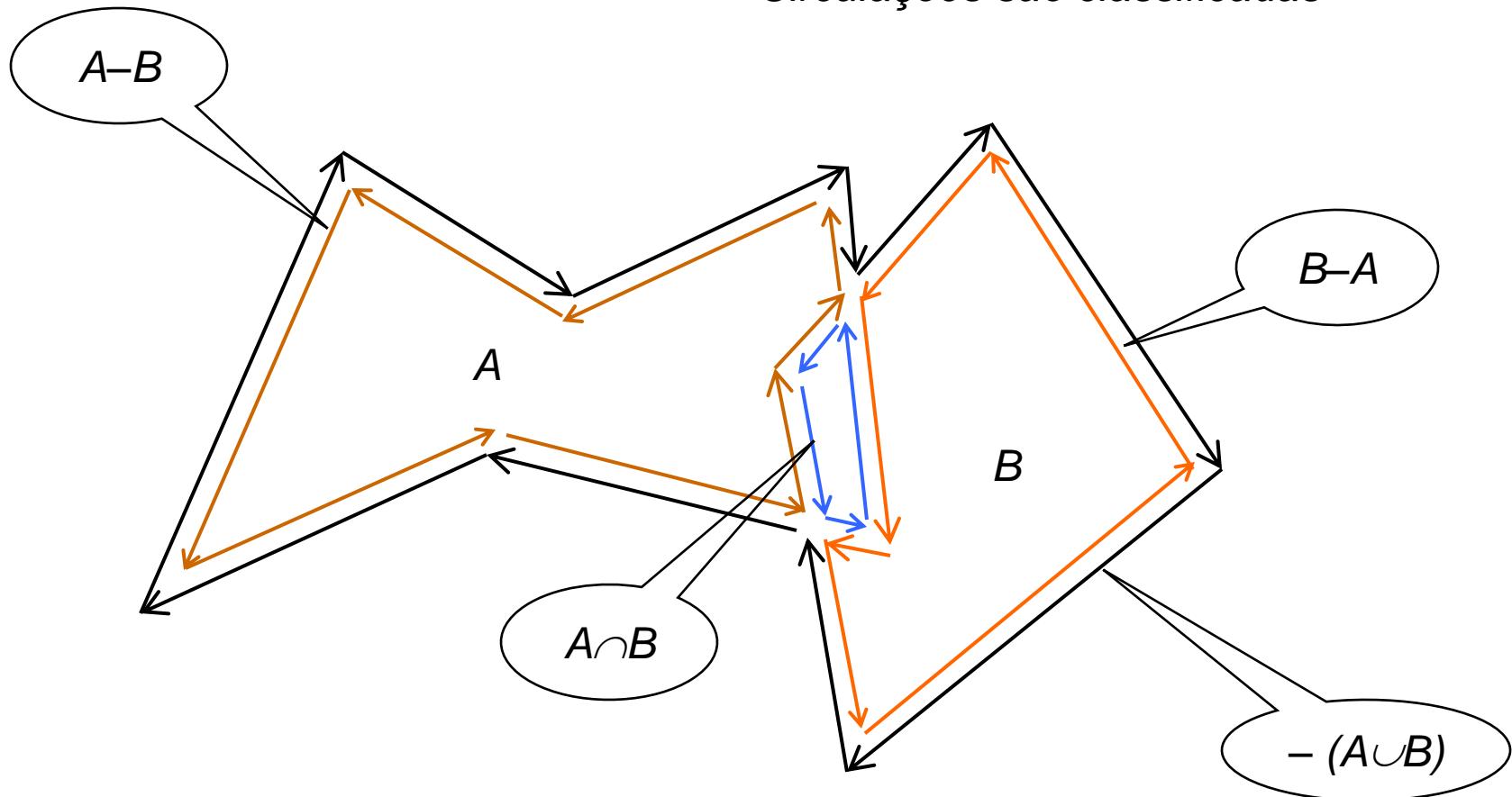
Algoritmo de Weiler-Atherton

Circulações são costuradas



Algoritmo de Weiler-Atherton

Circulações são classificadas



Algoritmos de Visibilidade

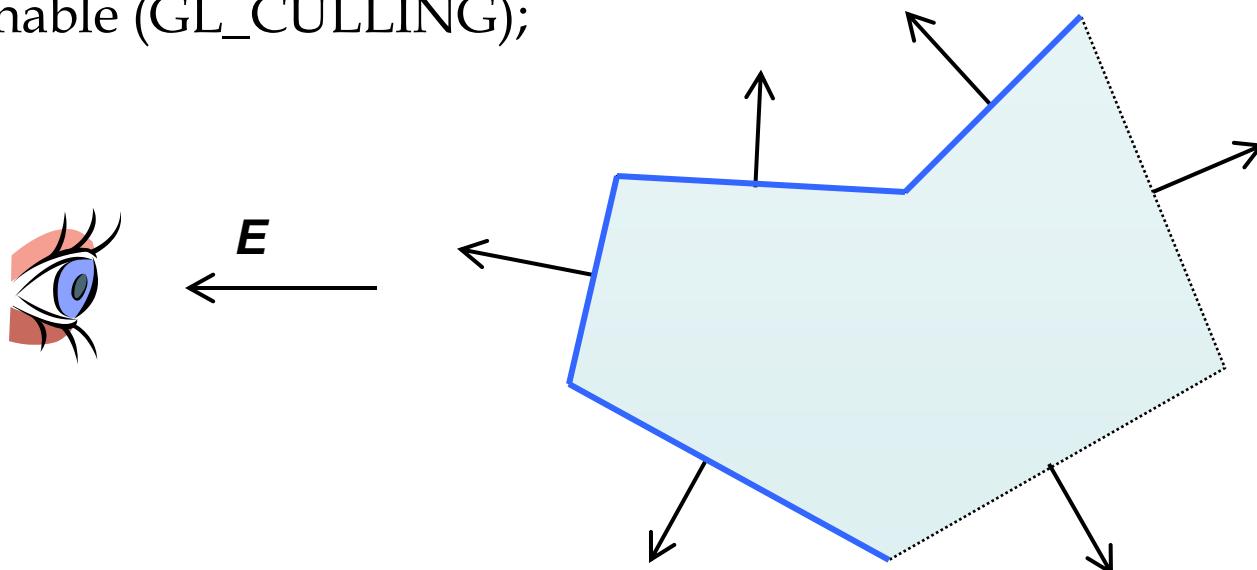
- Visibilidade é um problema complexo que não tem *uma* solução “óptima”
 - ◆ O que é óptima?
 - Pintar apenas as superfícies visíveis?
 - Pintar a cena em tempo mínimo?
 - ◆ Coerência no tempo?
 - Cena muda?
 - Objectos movem-se?
 - ◆ Qualidade é importante?
 - *Antialiasing*
 - ◆ Aceleração por Hardware?

Complexidade do Problema

- Factores que influenciam o problema
 - ◆ Número de pixels
 - Em geral procura-se minimizar o número total de pixels pintados
 - Resolução da imagem / *depth* buffer
 - Menos importante se rasterização é feita por hardware
 - ◆ Número de objectos
 - Técnicas de “*culling*”
 - Células e portais
 - Recorte pode aumentar o número de objectos

Backface Culling

- Hipótese: cena é composta de objectos poliédricos fechados
- Podemos reduzir o número de faces aproximadamente a metade
 - ◆ Faces de trás não precisam ser pintadas
- Como determinar se a face é de trás?
 - ◆ $N \cdot E > O \rightarrow$ Face da frente
 - ◆ $N \cdot E < O \rightarrow$ Face de trás
- OpenGL
 - ◆ `glEnable(GL_CULLING);`



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*
 - Pintar o pixel (atualizar o buffer de cor)
 - Actualizar o buffer de profundidade
 - ◆ Caso contrário, ignorar

Z-Buffer

- OpenGL:
 - ◆ Habilitar o z-buffer:
`glEnable(GL_DEPTH_TEST);`
 - ◆ Não esquecer de alocar o z-buffer → GLUT
 - Número de bits por pixel depende de implementação / disponibilidade de memória
 - ◆ Ao gerar um novo quadro, limpar também o z-buffer:
`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`
 - ◆ Ordem imposta pelo teste de profundidade pode ser alterada
`glDepthFunc(...)`

Z-Buffer

- Vantagens:
 - ◆ Simples e comunmente implementado em Hardware
 - ◆ Objectos podem ser desenhados em qualquer ordem
- Desvantagens:
 - ◆ Rasterização independente 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

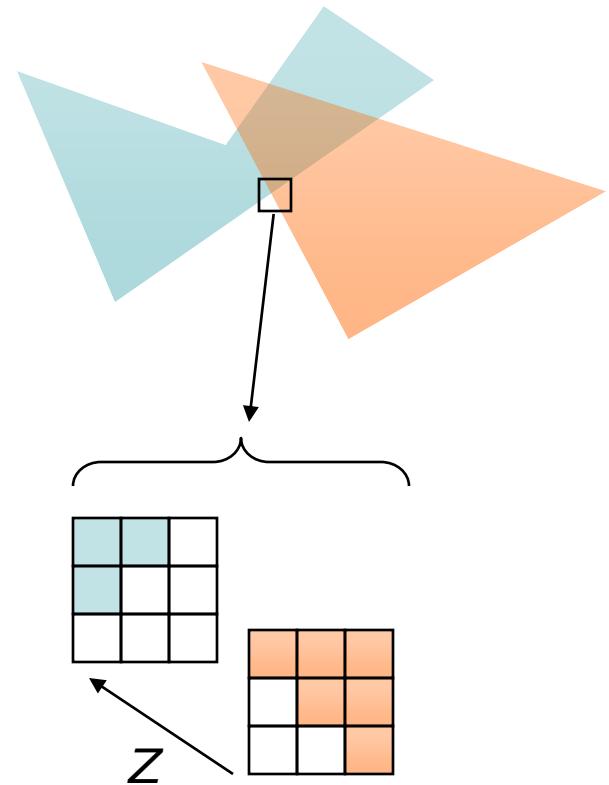
Z-Buffer e Transparência

- Se há objectos semi-transparentes, a ordem de renderização é importante
- Após a renderização de um objecto transparente, actualiza-se o z-buffer?
 - ◆ Sim → novo objecto por trás não pode mais ser renderizado
 - ◆ Não → z-buffer fica incorreto
- Soluções
 - ◆ Extender o z-buffer → A-buffer
 - ◆ Pintar de trás para frente → Algoritmo do pintor
 - Necessário de qualquer maneira, para realizar transparência com *blending* (canal alfa)



A-Buffer

- Melhoramento da ideia do z-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

- Fase 1: Polígonos são rasterizados
 - ◆ Se pixel completamente coberto por polígono e polígono é opaco
 - Inserir na lista removendo polígonos mais profundos
 - ◆ Se o polígono é transparente ou não cobre totalmente o pixel
 - Inserir na lista
- Fase 2: Geração da imagem
 - ◆ Máscaras de subpixels são misturadas para obter cor final do pixel

A-Buffer

- Vantagens
 - ◆ Faz mais do que o z-buffer
 - ◆ Ideia da máscara de subpixels pode ser usada com outros algoritmos de visibilidade
- Desvantagens
 - ◆ Implementação (lenta) por software
 - ◆ Problemas do z-buffer permanecem
 - Erros de quantização em z
 - Todos os polígonos são rasterizados

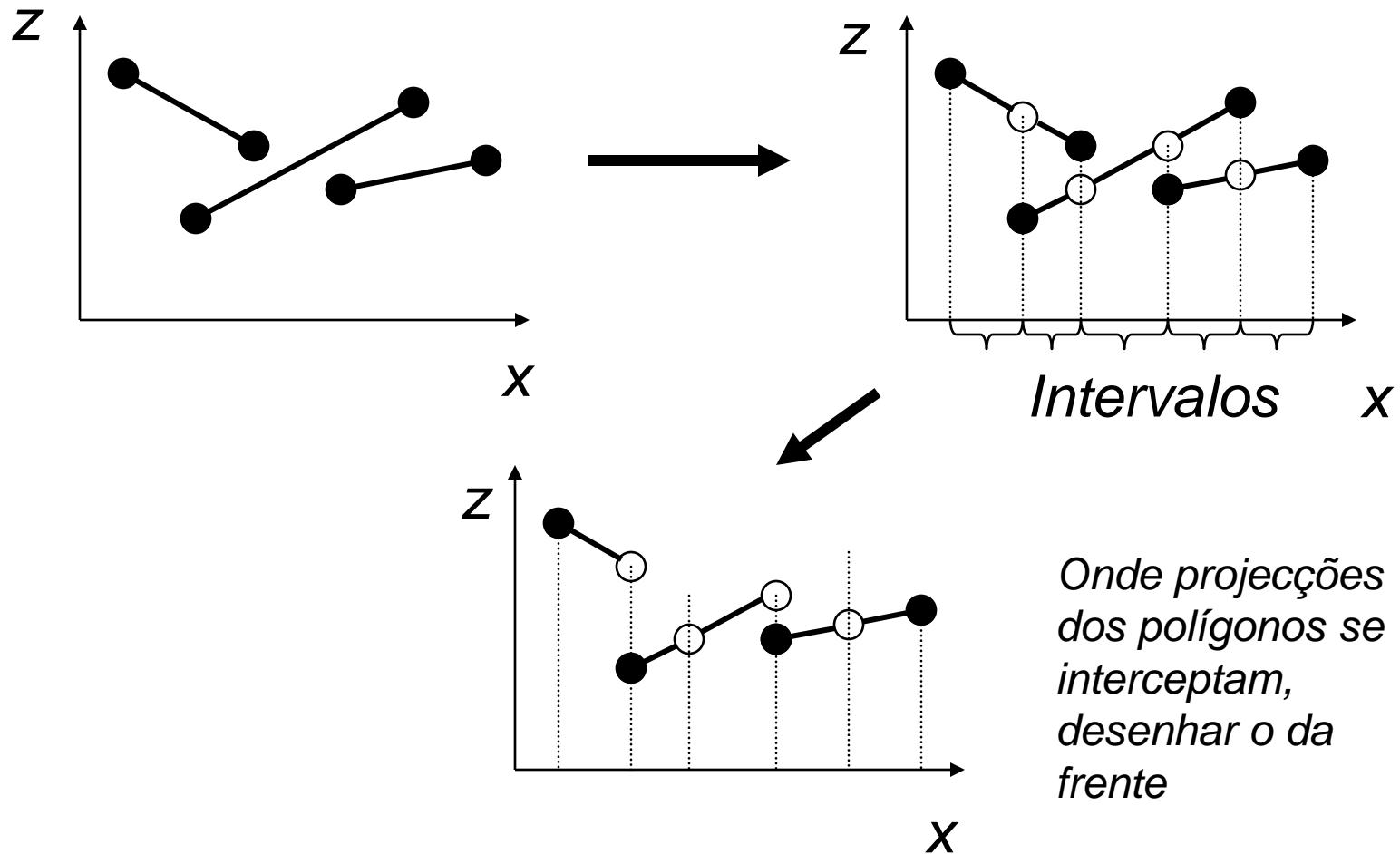
Algoritmo Scan-Line

- Idéia 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 mesmo ser estendido para rasterizar sólidos CSG

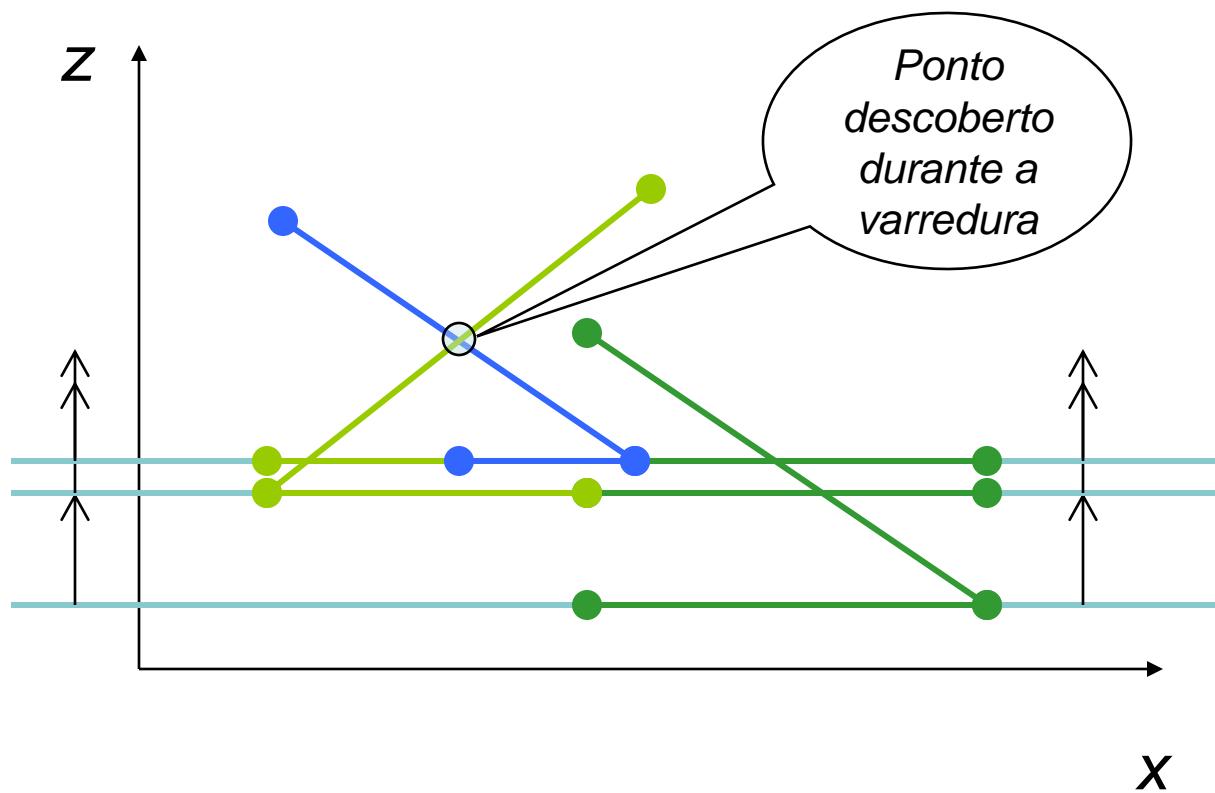
Algoritmo Scan-Line

- Ordenam-se todas as arestas de todos os polígonos por y_{min}
- Para cada plano de varrimento y
 - ◆ Para cada polígono
 - Determinar intervalos x_i de intersecção com plano de varredura
 - ◆ Ordenar intervalos de intersecção por z_{min}
 - ◆ Para cada linha de varriente z
 - Inserir arestas na linha de varrimento respeitando inclinação z/x
 - ◆ Renderizar resultado da linha de varrimento

Algoritmo Scan-Line



Algoritmo Scan-Line



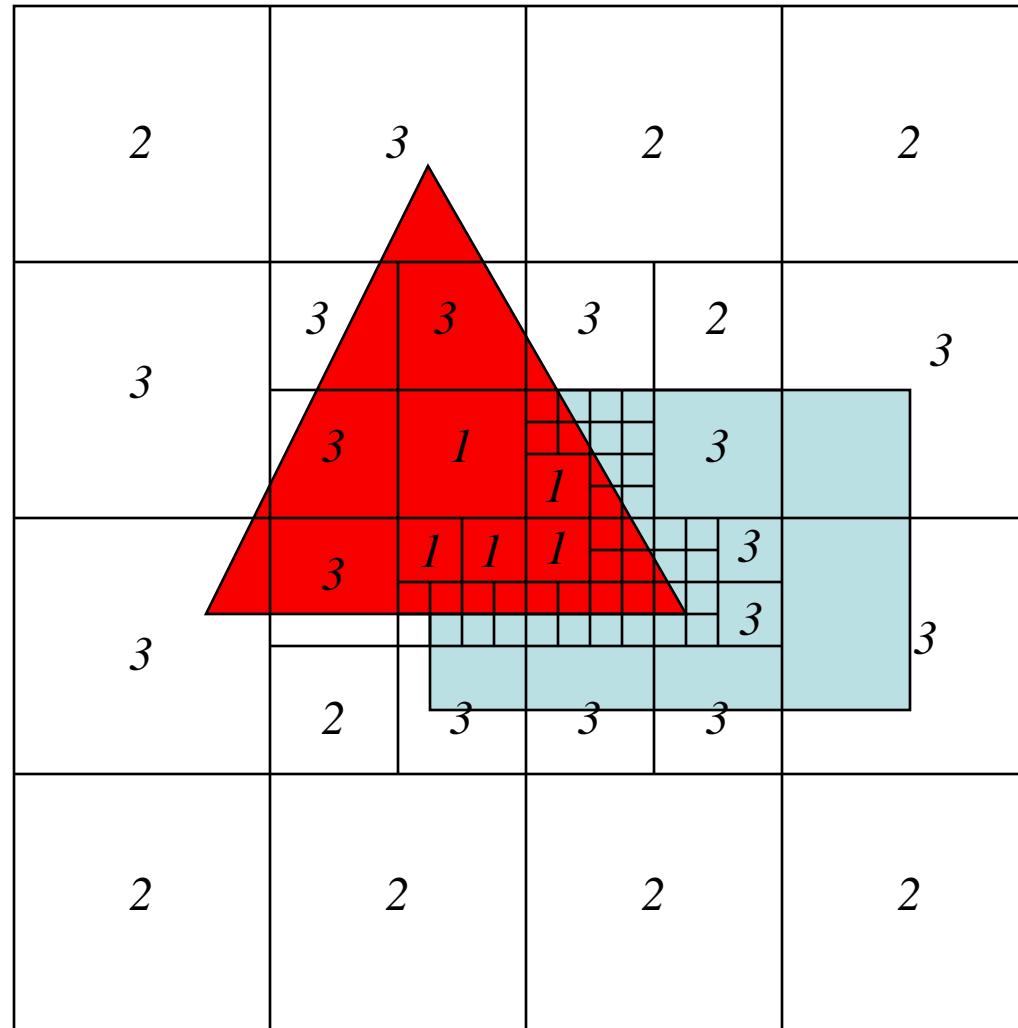
Algoritmo Scan-Line

- Vantagens
 - ◆ Algoritmo flexível que explora a coerência entre pixels de uma mesma linha de varrimento
 - ◆ Razoável independência da resolução da imagem
 - ◆ Filtragem e *anti-aliasing* 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 varrimento não é explorada
 - Polígonos invisíveis são descartados múltiplas vezes
 - ◆ Relativa complexidade
 - ◆ Não muito próprio para implementação em hardware

Algoritmo de Warnock

- Usa subdivisão do espaço da imagem para explorar coerência de área
- Sabemos como pintar uma determinada sub-região da imagem se:
 1. Um polígono cobre a região totalmente e em toda região é mais próximo que os demais
 2. Nenhum polígono a intercepta
 3. Apenas um polígono a intercepta
- Se a sub-região não satisfaz nenhum desses critérios, é subdividida recursivamente à maneira de uma quadtree
 - ◆ Se sub-região se reduz a um pixel, pintar o polígono com menor profundidade

Algoritmo de Warnock

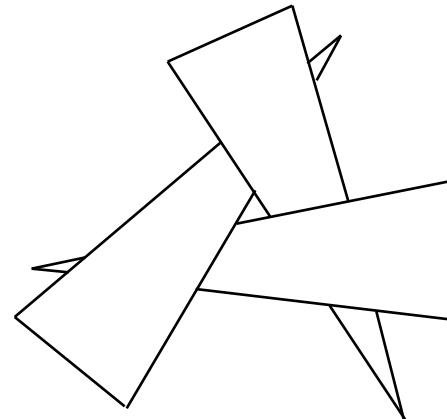


Algoritmo de Warnock

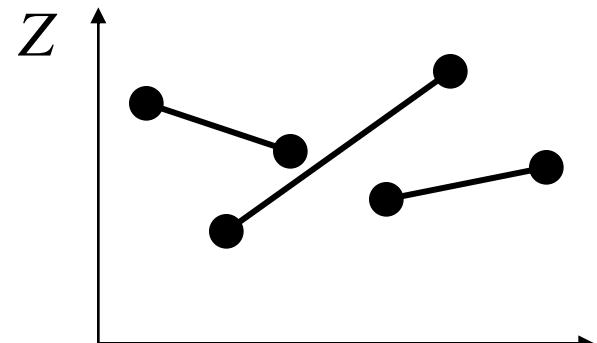
- Vantagens
 - ◆ Explora coerência de área
 - Apenas áreas que contêm arestas precisam ser subdivididas até o nível de pixel
 - ◆ Pode ser adaptado para suportar transparência
 - ◆ Levando a recursão até tamanho de subpixel, pode-se fazer filtragem de forma elegante
 - ◆ Pinta cada pixel apenas uma vez
- Desvantagens
 - ◆ Testes são lentos
 - ◆ Aceleração por hardware improvável

Algoritmo do Pintor

- Também conhecido como algoritmo de prioridade em Z (*depth priority*)
- Ideia é pintar objectos mais distantes (*background*) antes de pintar objectos próximos (*foreground*)
- Requer que objectos sejam ordenados em Z
 - ◆ Complexidade $O(N \log N)$
 - ◆ Pode ser complicado em alguns casos
 - ◆ Na verdade, a ordem não precisa ser total se projecções dos objectos não se interceptam



Não há ordem possível



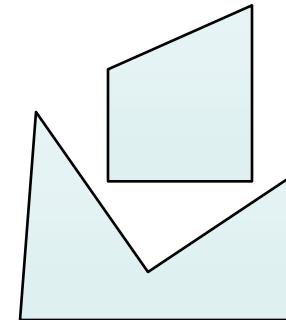
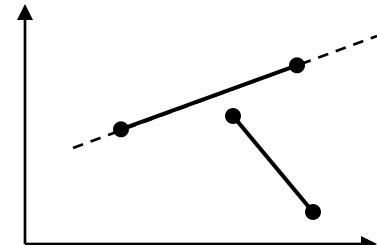
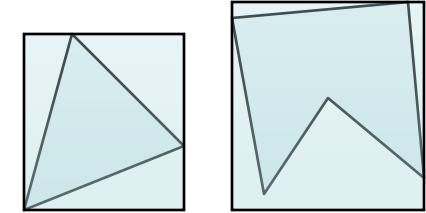
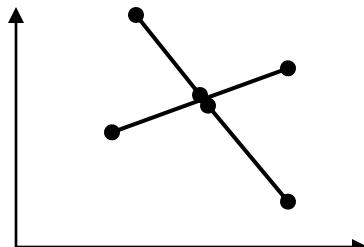
Que ponto usar para determinar ordem?

Algoritmo do Pintor

- Ordenação requer que se determine, para cada par de polígonos A e B :
 - ◆ A precisa ser pintado antes de B
 - ◆ B precisa ser pintado antes de A
 - ◆ A ordem de pintura é irrelevante
- Pode-se usar um algoritmo simples baseado em troca. Ex.: *Bubble Sort*
- Como a ordem a ser determinada não é total, pode-se usar um algoritmo de ordenação parcial
Ex.: *Union-Find* (Tarjan)

Algoritmo do Pintor

- Ordem de pintura entre A e B determinada por testes com níveis crescentes de complexidade
 1. Caixas limitantes de A e B não se interceptam
 2. A atrás ou na frente do plano de B
 3. B atrás ou na frente do plano de A
 4. Projecções de A e B não se interceptam
- Se nenhum teste for conclusivo, A é substituído pelas partes obtidas recortando A pelo plano de B (ou vice-versa)



Algoritmo de Recorte Sucessivo

- Pode ser pensado como um algoritmo do pintor ao contrário
- Polígonos são pintados de frente para trás
- É mantida uma máscara que delimita quais porções do plano já foram pintadas
 - ◆ Máscara é um polígono genérico (pode ter diversas componentes conexas e vários “buracos”)
- Ao considerar cada um novo polígono P
 - ◆ Recortar contra a máscara M e pintar apenas $P - M$
 - ◆ Máscara agora é $M + P$

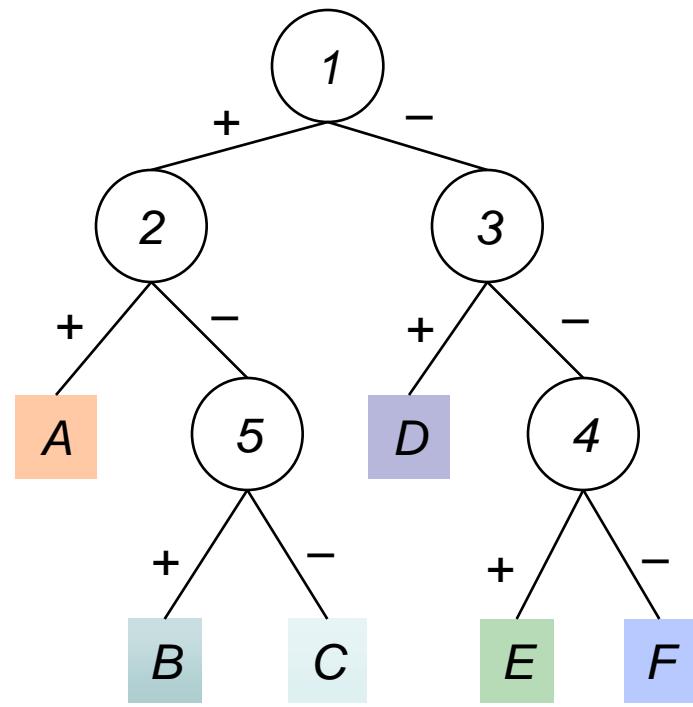
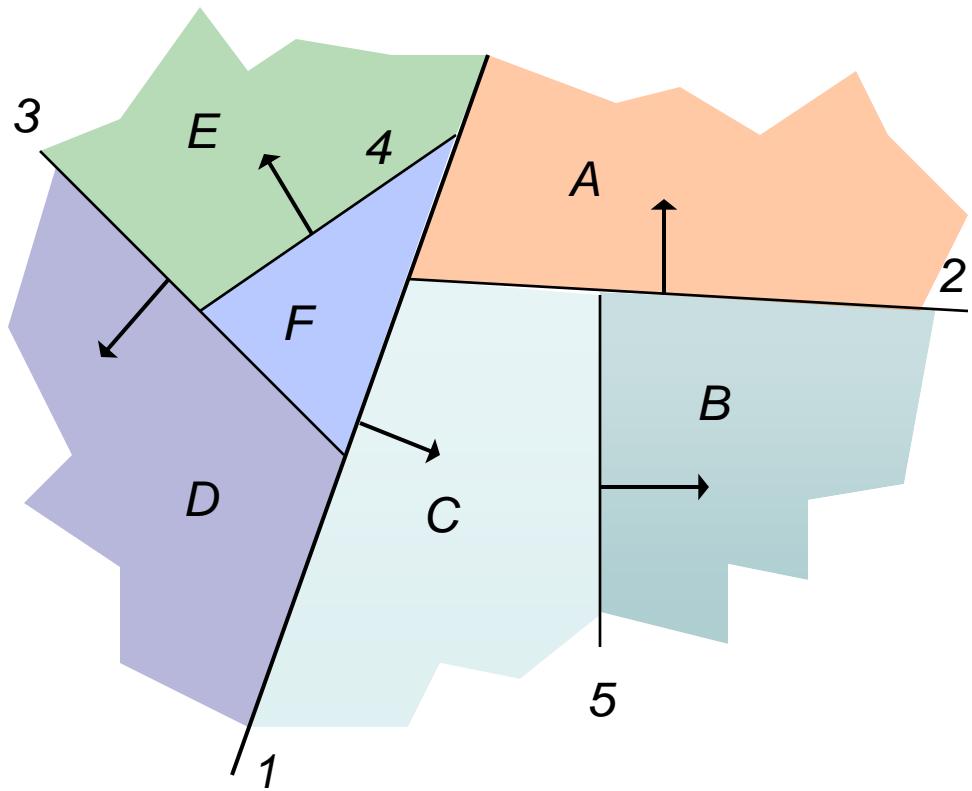
Algoritmo de Recorte Sucessivo

- Vantagens
 - ◆ Trabalha no espaço do objecto
 - Independente da resolução da imagem
 - Não tem problemas de quantização em z
 - ◆ Pinta cada pixel uma vez apenas
- Desvantagem
 - ◆ Máscara pode-se tornar arbitrariamente complexa
 - ◆ Excessivamente lento

BSP-Trees

- São estruturas de dados que representam uma partição recursiva do espaço
- Muitas aplicações em computação gráfica
- Estrutura multi-dimensional
- Cada célula (começando com o espaço inteiro) é dividida em duas por um plano
 - ◆ *Binary Space Partition Tree*
- Partição resultante é composta de células convexas (politopos)

BSP-Tree



BSP-Trees

- A orientação dos planos de partição depende da aplicação e é um dos pontos mais delicados do algoritmo de construção
 - ◆ Ao partir colecções de objectos visa-se uma divisão aproximadamente equitativa
 - ◆ Se estamos partindo polígonos
 - 2D - normalmente usa-se a direcção de alguma aresta como suporte para o plano
 - 3D - normalmente usa-se a orientação do plano de suporte do de algum polígono
 - ◆ Se os objectos têm extensão, é importante escolher planos que interceptem o menor número possível de objectos

BSP-Trees e Visibilidade

- BSP-trees permitem obter uma ordem de desenho baseada em profundidade
 - ◆ Vantagem: se o observador se move, não é preciso reordenar os polígonos
 - ◆ Bastante usada em aplicações de caminhada em ambientes virtuais (arquitectura, museus, jogos)
- Diversas variantes
 - ◆ Desenhar de trás para frente (algoritmo do pintor)
 - ◆ Desenhar de frente para trás (algoritmo de recorte recursivo)
 - ◆ Outras...

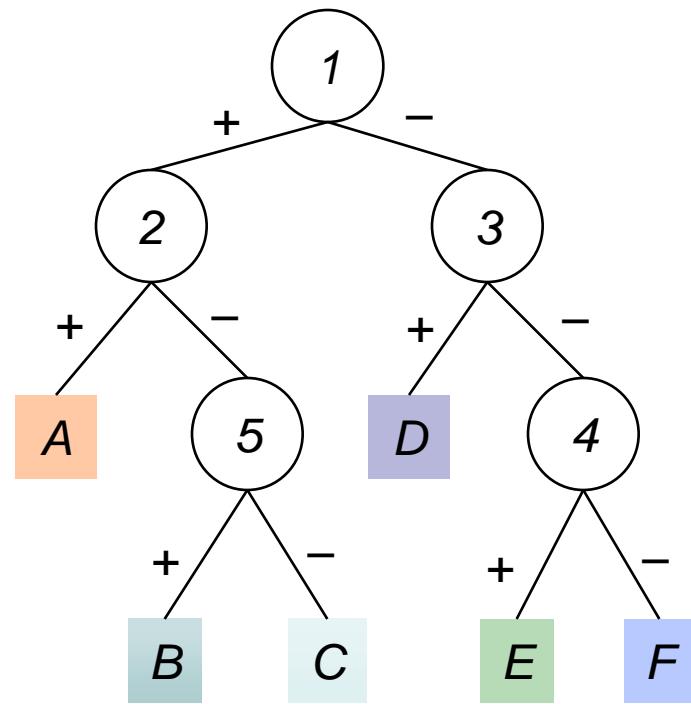
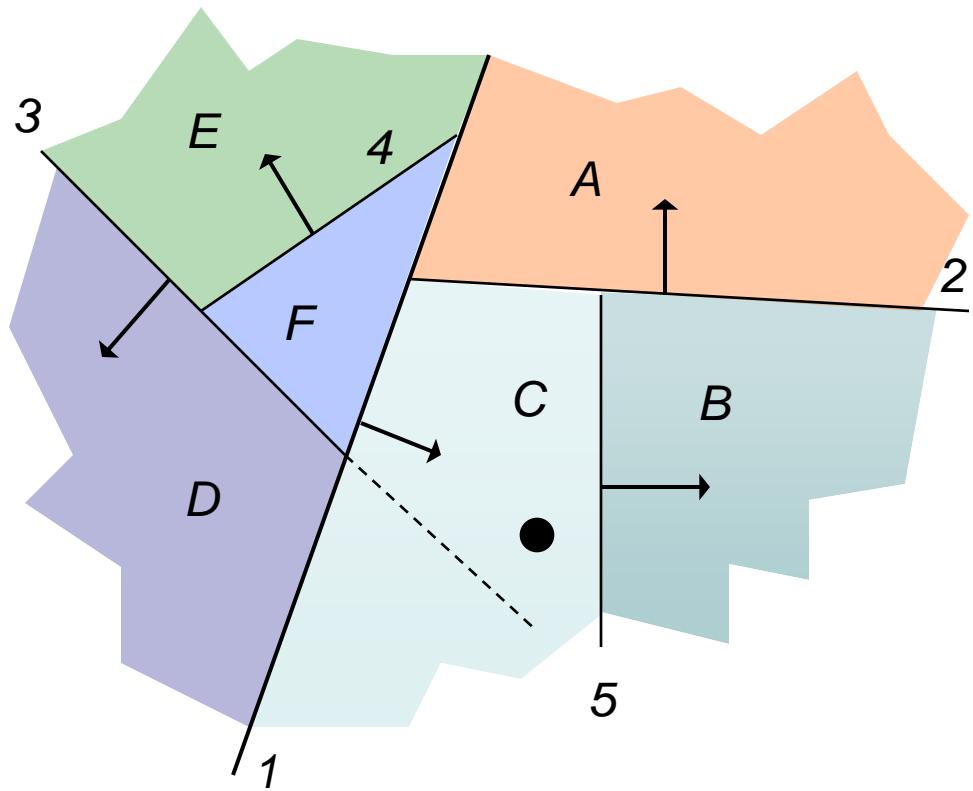
BSP-Trees - Construção

- Escolhe-se um dos polígonos da colecção presente na célula (ao acaso?)
 - ◆ Não existe algoritmo ótimo
 - ◆ Algumas heurísticas (ex.: *minimum stabbing number*)
- Divide-se a colecção em duas sub-colecções (além do próprio polígono usado como suporte)
 - ◆ Polígonos na frente do plano
 - ◆ Polígonos atrás do plano
- Divisão pode requerer o uso de recorte
- Partição prossegue recursivamente até termos apenas um polígono por célula

BSP-Trees - Desenho

- Se observador está de um lado do plano de partição, desenha-se (recursivamente)
 - ◆ Os polígonos do lado oposto
 - ◆ O próprio polígono de partição
 - ◆ Os polígonos do mesmo lado
- Pode-se ainda fazer *culling* das células fora do volume de visão

BSP-Tree



Ordem de desenho: D E F A B C

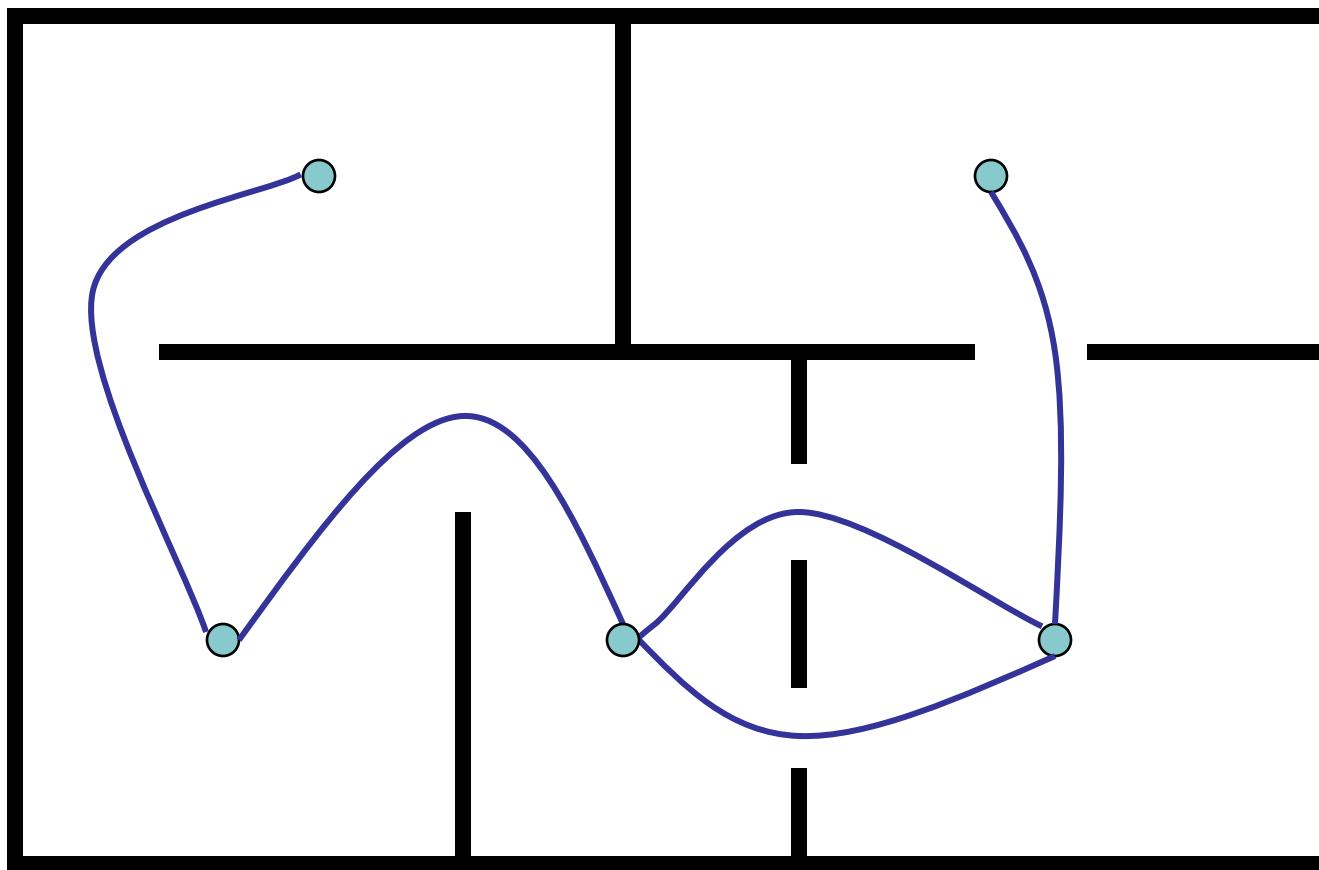
BSP-Trees

- Vantagens
 - ◆ Pode ser usado para caminhadas
 - ◆ Filtragem e *anti-aliasing* suportados com facilidade (desenho de trás para a frente)
 - ◆ Algoritmo de frente para trás usado em jogos
- Desvantagens
 - ◆ Desenha mesmo pixel várias vezes
 - ◆ Número de polígonos pode crescer muito

Células e Portais

- Ideia usada em aplicações de caminhada (*walkthrough*) por ambientes virtuais do tipo arquitectônico
 - ◆ Cena composta de diversos compartimentos (quartos, salas, etc)
- Visibilidade é determinada convencionalmente dentro de cada compartimento (célula)
- Visibilidade entre células requer que luz atravesse partes vasadas das paredes tais como janelas, portas, etc (portais)
- Modelo de células e portais pode ser entendido como um grafo
 - ◆ Células = vértices
 - ◆ Portais = arestas

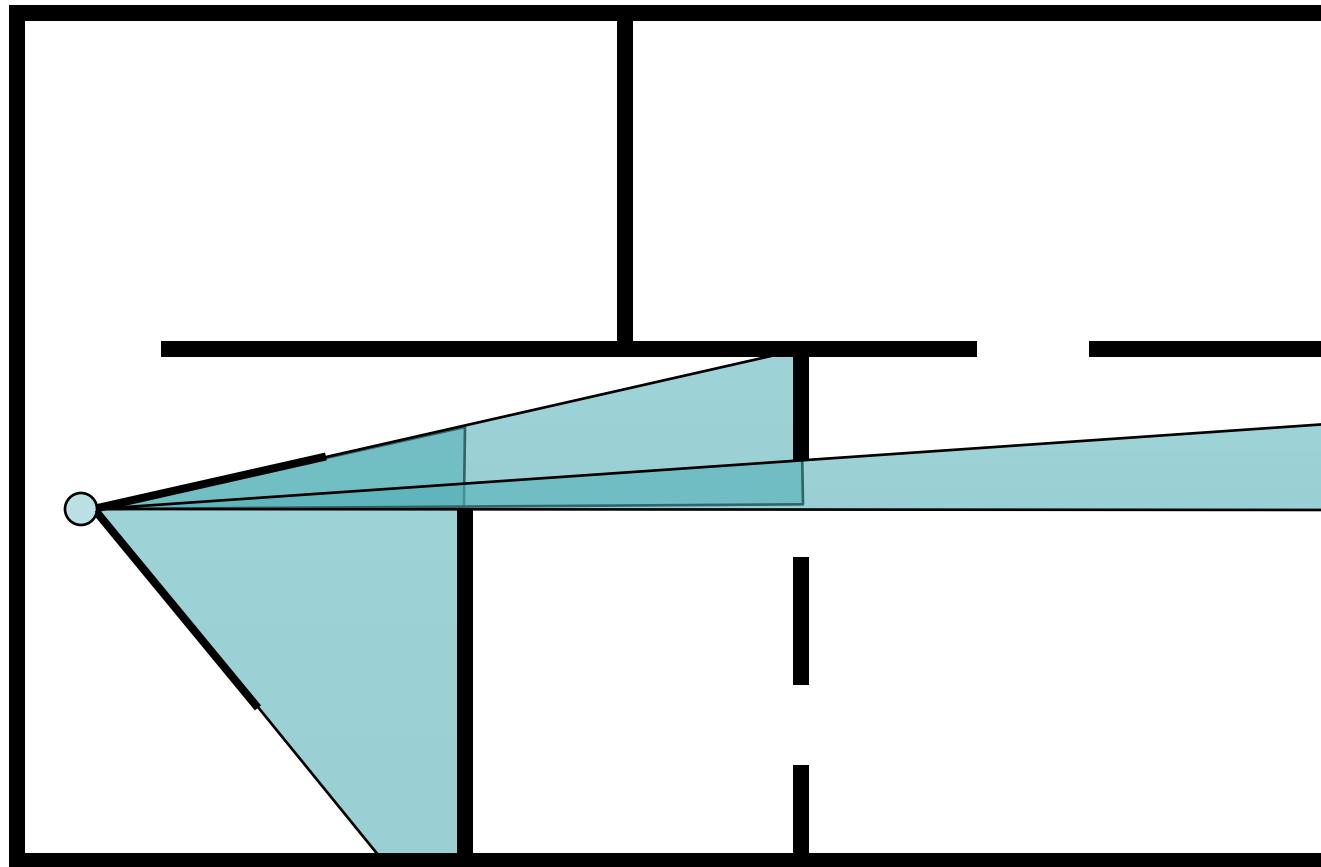
Células e Portais



Células e Portais - Algoritmo

- Desenhar célula C (paredes, objectos) onde o observador está
- Para cada célula V_i vizinha à célula do observador por um portal, recortar o volume de visão pelo portal
- Se volume recortado não for nulo,
 - ◆ Desenhar célula vizinha restrita à região não recortada do volume de visão
 - ◆ Repetir o procedimento recursivamente para as células vizinhas de V_i

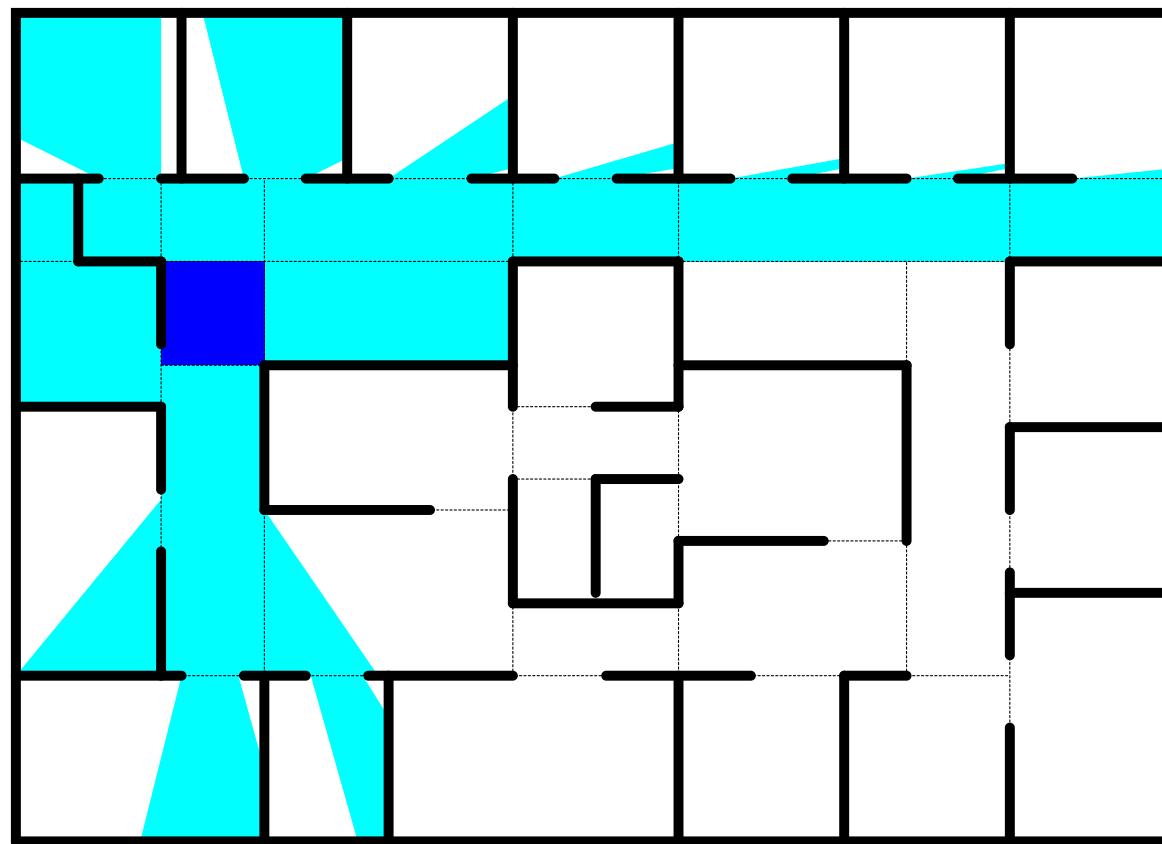
Células e Portais - Exemplo



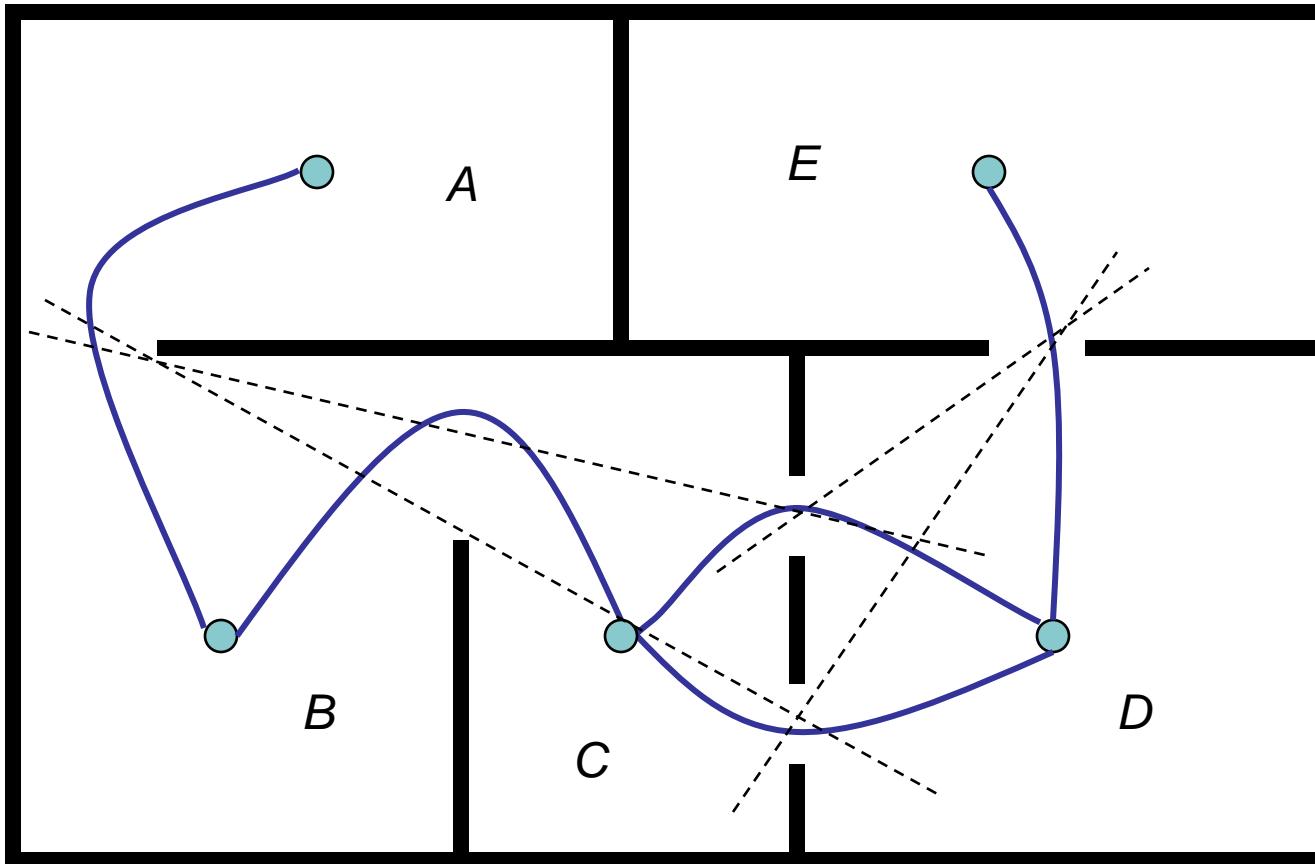
Células e Portais – Visibilidade Pré-Computada

- Operações de recorte são complexas
 - ◆ Volume recortado pode ter um grande número de faces
- Ideia: Pré-computar dados de visibilidade
- Conceito de observador genérico
 - ◆ Observador que tem liberdade para se deslocar para qualquer ponto da célula e olhar em qualquer direcção
- Informação de visibilidade
 - ◆ Célula a Região (estimativa exata)
 - ◆ Célula a Célula (estimativa grosseira)
 - ◆ Célula a Objecto (estimativa fina)

Visibilidade Célula a Região



Visibilidade Célula a Célula



1 Portal

AB, BC, CD, DE

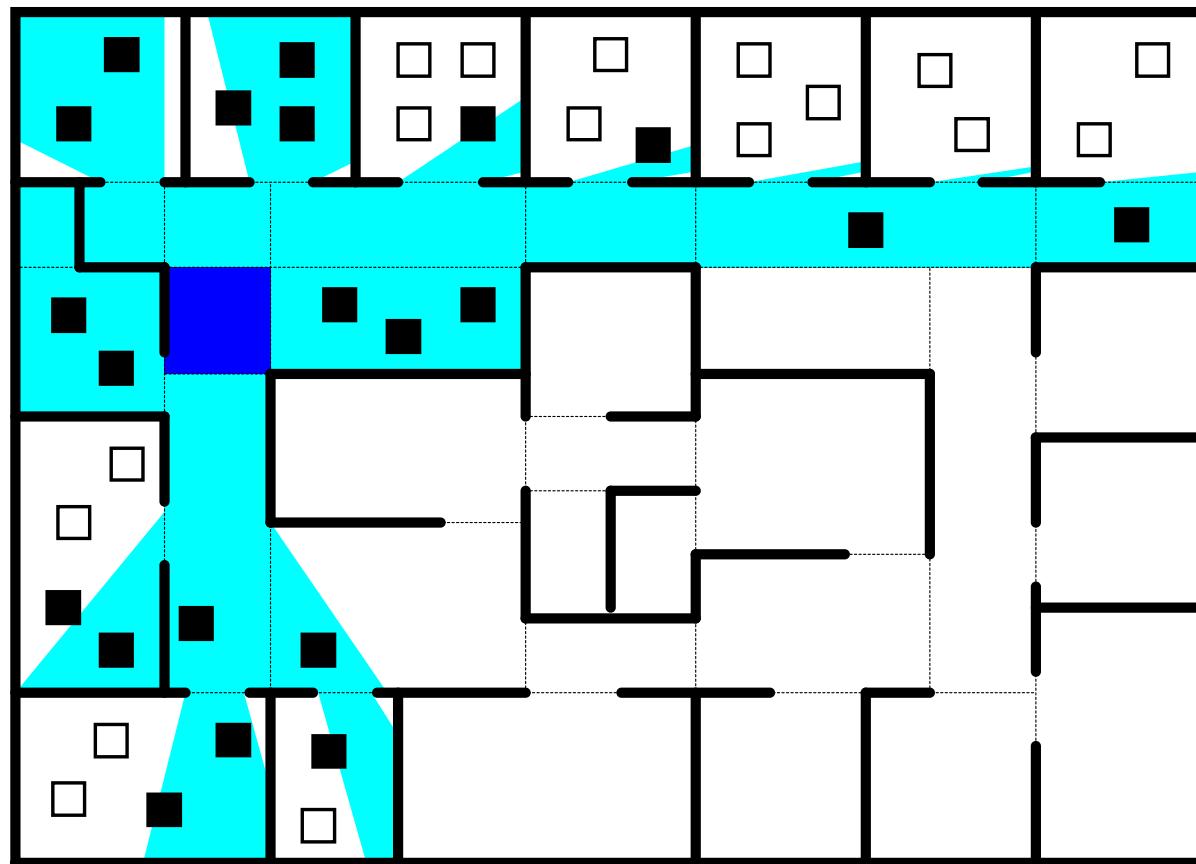
2 Portais

AC, BD, CE

3 Portais

AD

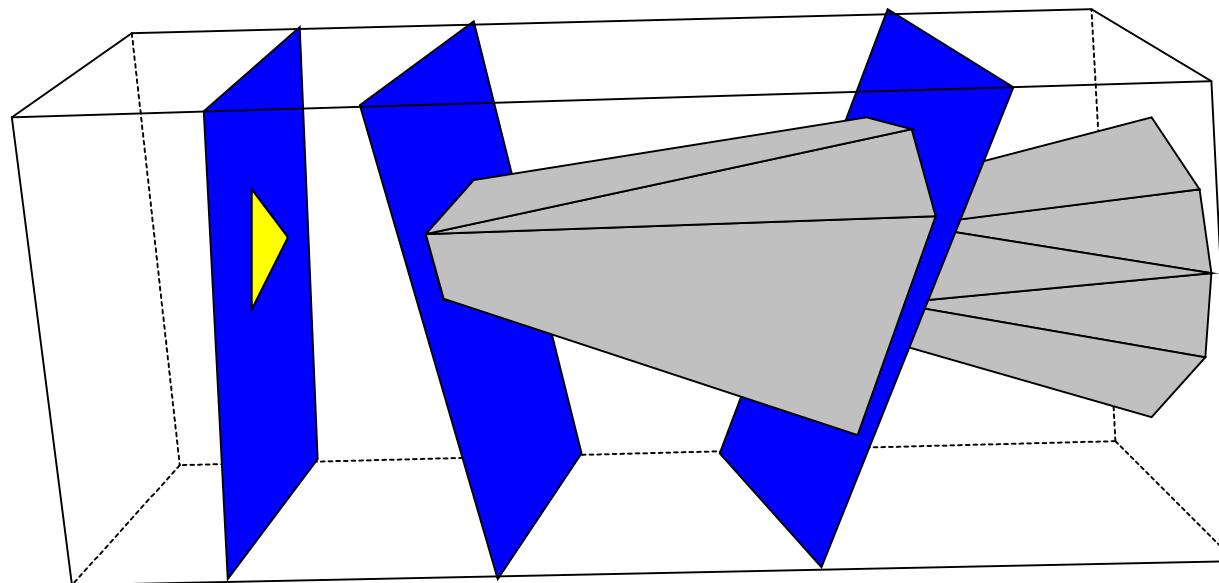
Visibilidade Célula a Objecto



Células e Portais

- Uma vez computada a visibilidade célula-a-região, os demais dados de visibilidade são obtidos trivialmente
- Em 3D, o cálculo exato dos volumes de visão pode ser bastante complexo (faces quádricas)
 - ◆ Na prática, usa-se aproximações conservadoras desses volumes (faces planas)
 - ◆ Paper Eurographics 2000: “*Efficient Algorithms for Computing Conservative Portal Visibility Information*” Jiménez, Esperança, Oliveira

Estimativa Conservadora de Volumes de Visão



Células e Portais – Algoritmo com Visibilidade Pré-Computada

- Desenhar célula C do observador
- Desenhar todas as células no Conjunto de Visibilidade de C
 - ◆ Células com visibilidade não nula através de uma sequência de portais
 - ◆ Usar z-buffer
 - ◆ Se dados de visibilidade célula-a-objecto estiverem disponíveis, desenhar apenas os objectos visíveis

Células e Portais - Resumo

- Versão mais utilizada requer que se pré-compute dados de visibilidade
 - ◆ Antecede a fase de caminhada
 - ◆ Visibilidade é aproximada
 - ◆ Requer método auxiliar para determinação de visibilidade
- Vantagens
 - ◆ Bastante eficiente em ambientes complexos com alta probabilidade de oclusão
 - ◆ Reduz o número de objectos a serem desenhados em algumas ordens de grandeza
- Desvantagens
 - ◆ Pré-processamento
 - ◆ Não tem grande utilidade em alguns tipos de cena
 - Ex. ambientes ao ar livre