

Visibilidad

# O Problema de Visibilidade

- Numa cena tri-dimensional, 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)

## Motivação

- Dispositivos matriciais sobre-escrevem os objetos (aparecem os desenhados por último, quando há sobreposição).
  - ◆ Em 3D, gera uma imagem incorreta, se nada for feito para corrigir a ordem de desenho.
- Os algoritmos de visibilidade estruturam os objetos da cena, de modo a que sejam exibidos corretamente.

# 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
    - Algoritmos de visibilidade que utilizam recorte
      - Algoritmo do pintor
      - BSP-trees
      - Algoritmo de recorte sucessivo
      - Volumes de sombra

# Espaço do Objeto x Espaço da Imagem

- 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
    - Algoritmo de Warnock
    - Mapas de sombra

# Algoritmos de Visibilidade

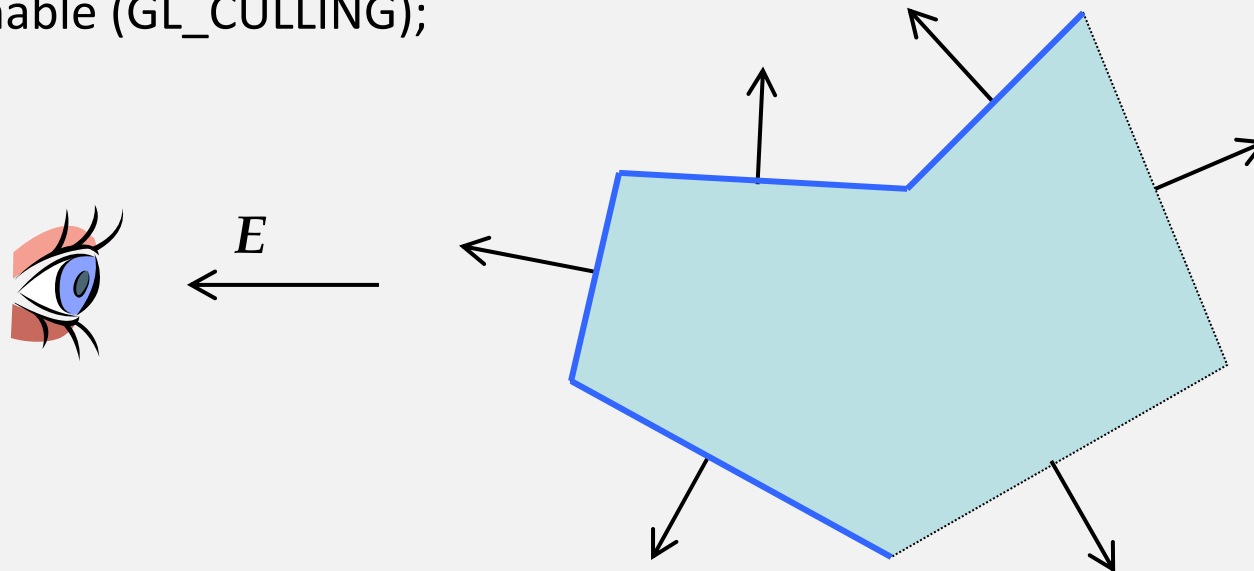
- Visibilidade é um problema complexo que não tem *uma* solução “ótima”
  - ♦ O que é ótima?
    - Pintar apenas as superfícies visíveis?
    - Pintar a cena em tempo mínimo?
  - ♦ Coerência no tempo?
    - Cena muda?
    - Objetos se movem?
  - ♦ Qualidade é importante?
    - Antialiasing
  - ♦ Aceleração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

# 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 / *depth* buffer
    - Menos importante se rasterização é feita por hardware
  - ◆ Número de objetos
    - Técnicas de “*culling*”
    - Células e portais
    - Recorte pode aumentar o número de objetos

# 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?
  - ♦  $N \cdot E > 0 \rightarrow$  Face da frente
  - ♦  $N \cdot E < 0 \rightarrow$  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)
    - Atualizar 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
    - Ex: `glutInitDisplayMode (GLUT_RGB|GLUT_DEPTH);`
    - 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
    - Ex: `glDepthFunc (GL_GREATER);`

# Z-Buffer

- Vantagens:
  - ♦ Simples e comumente implementado em Hardware
  - ♦ Objetos 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-serrilhado
    - É preciso ter informações sobre os vários polígonos que cobrem cada pixel

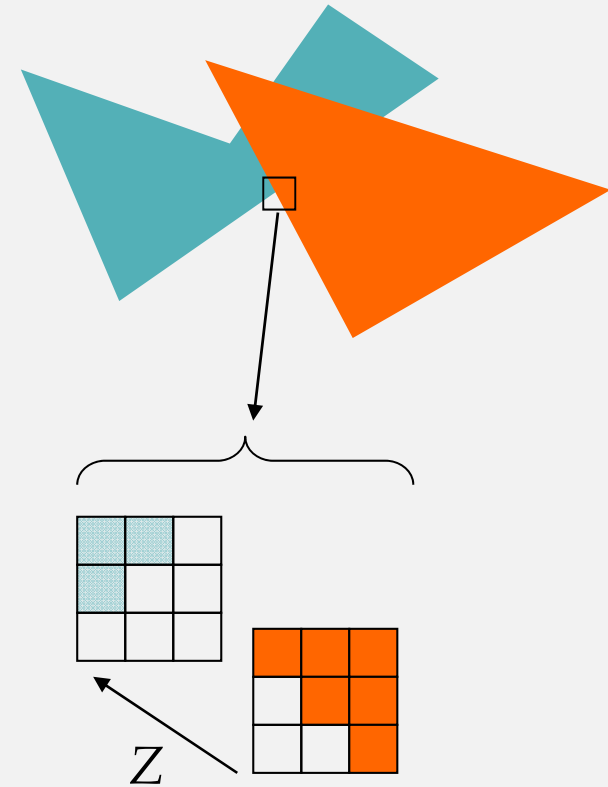
# Z-Buffer e Transparência

- 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
  - ♦ Pintar de trás para frente → Algoritmo do pintor
    - Necessário de qualquer maneira, para realizar transparência com *blending* (canal alfa)



# A-Buffer

- Melhoramento da idéia 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
  - ◆ Idéia 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”

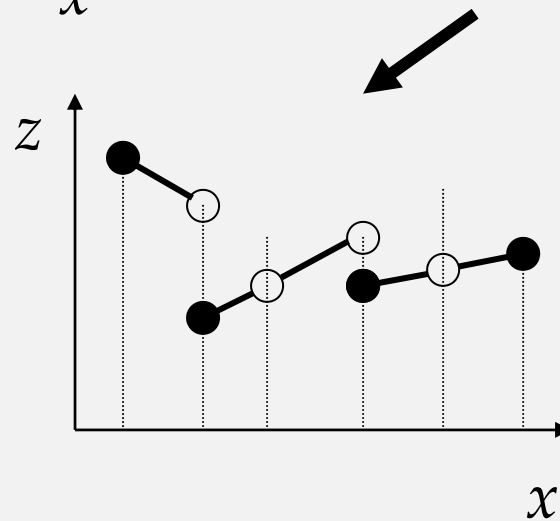
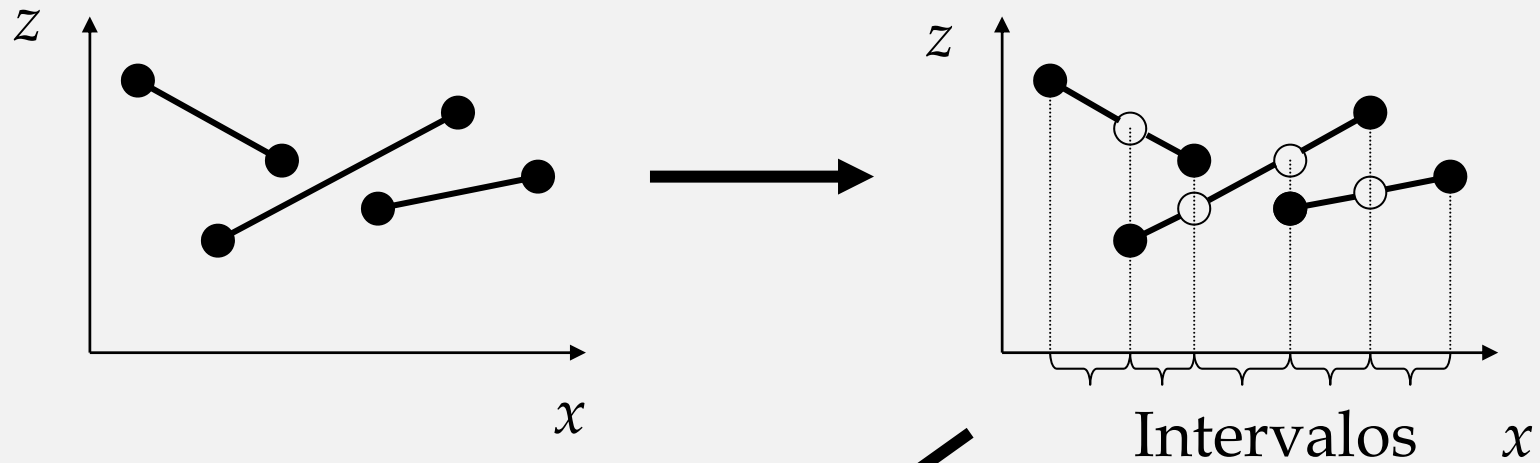
- Idéia é aplicar o algoritmo de rasterização de polígonos a todos os polígonos da cena simultaneamente
- Explora coerência de visibilidade
- Em sua concepção original requer que polígonos se interceptem apenas em vértices ou arestas
  - ♦ Pode ser adaptado para lidar com faces que se interceptam
  -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



# Algoritmo “Scan-Line”

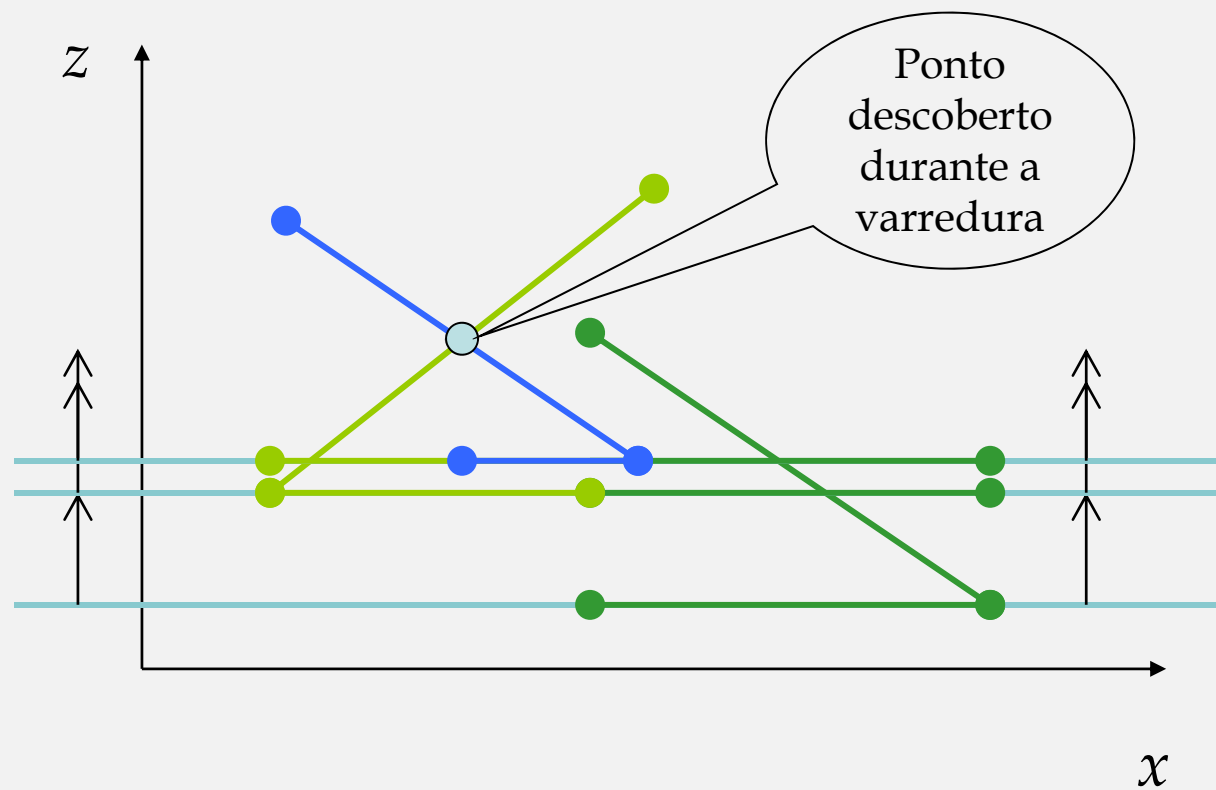
- Ordena-se todas as arestas de todos os polígonos por  $y_{min}$
- Para cada plano de varredura  $y$ 
  - ♦ Para cada polígono
    - Determinar intervalos  $x_i$  de interseção com plano de varredura
  - ♦ Ordenar intervalos de interseção por  $z_{min}$
  - ♦ Para cada linha de varredura  $z$ 
    -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

# Algoritmo "Scan-Line"



Onde projeções  
dos polígonos se  
interceptam,  
desenhar o da  
frente

# Algoritmo "Scan-Line"



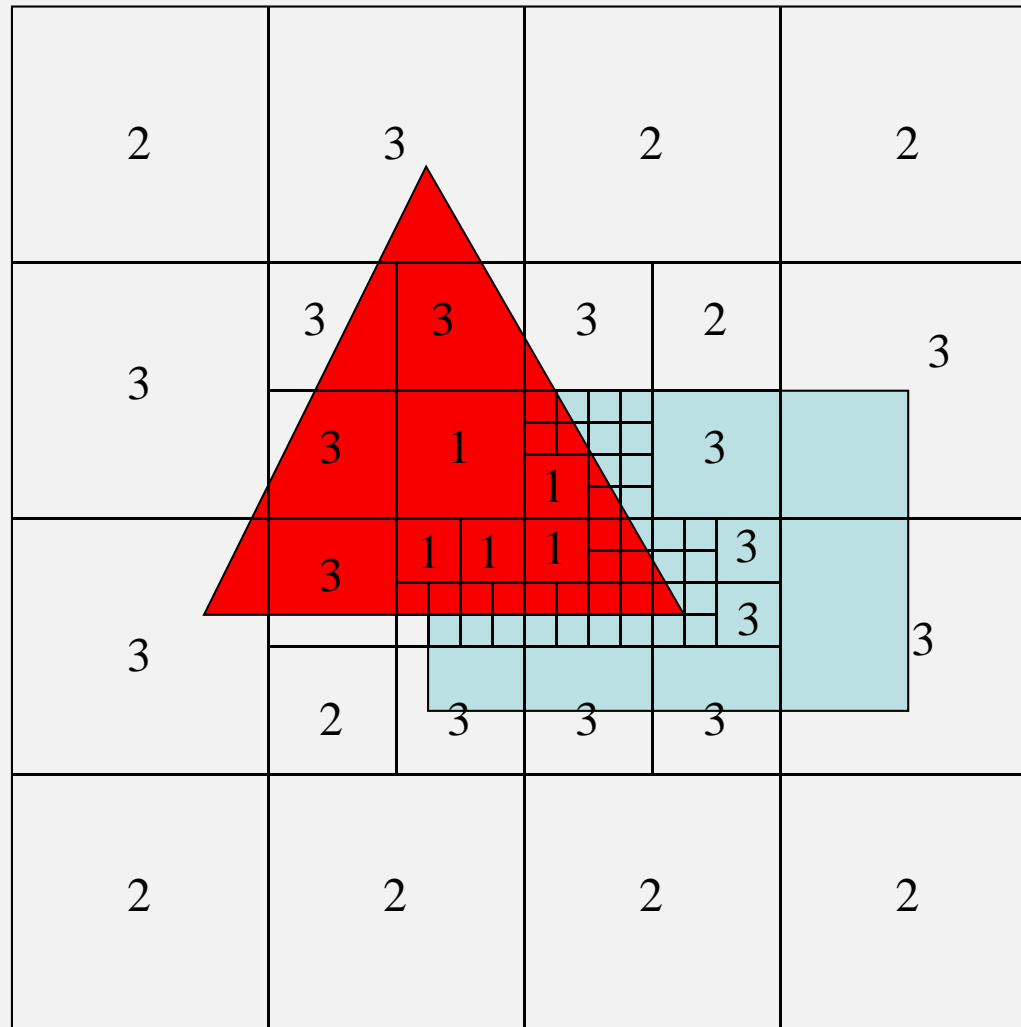
# Algoritmo “Scan-Line”

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

# 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

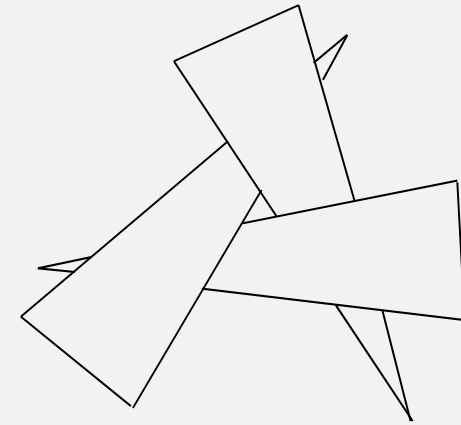


# Algoritmo de Warnock

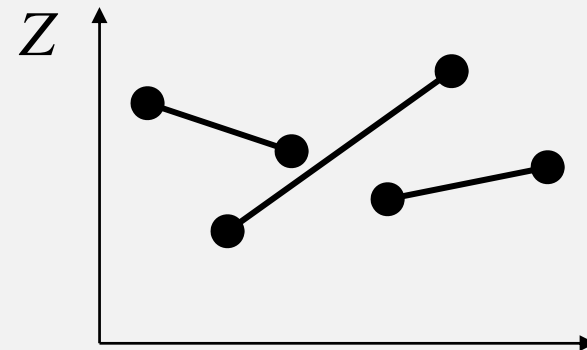
- Vantagens
  - ♦ Explora coerência de área
    - Apenas áreas que contêm arestas precisam ser subdivididas até o nível de pixel
  - ♦ Pode ser adaptado para suportar transparência
  - ♦ Levando a recursão até tamanho de subpixel, pode-se fazer filtragem de forma elegante
  - ♦ Pinta cada pixel apenas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

# Algoritmo do Pintor

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



Não há ordem possível



Que ponto usar para determinar ordem?

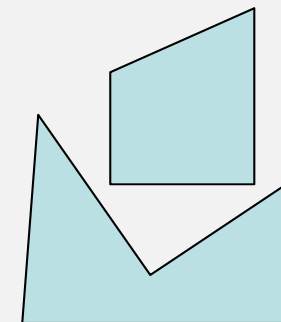
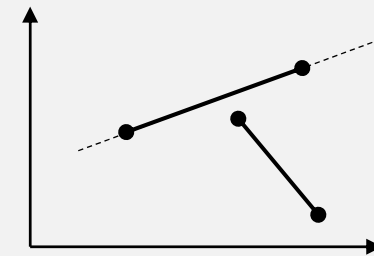
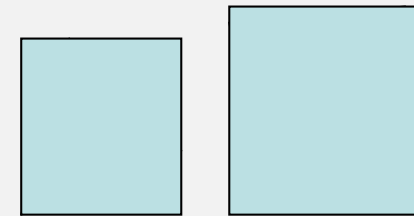
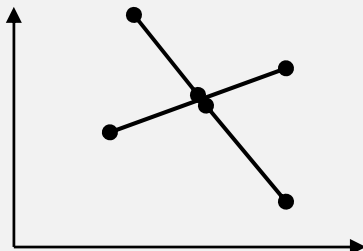


# Algoritmo do Pintor

- 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. Projeções 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  $M + P$

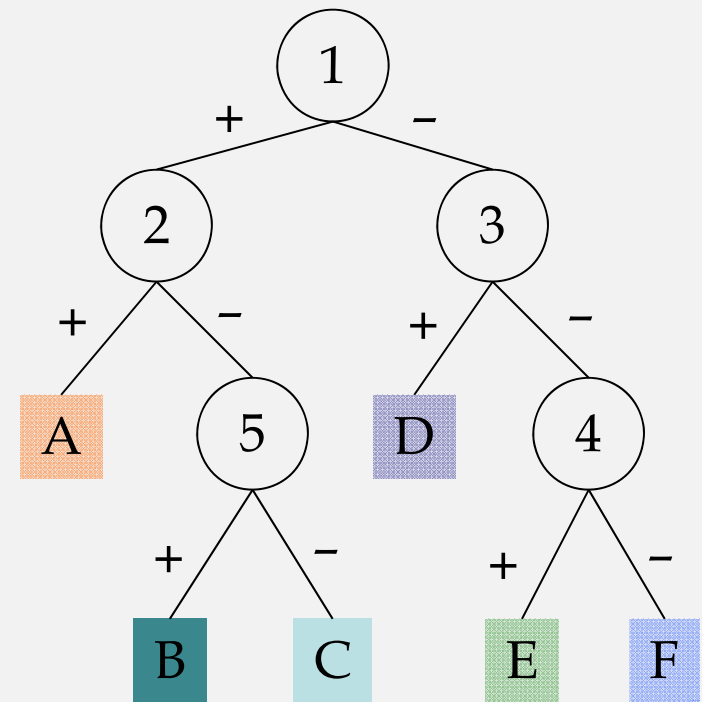
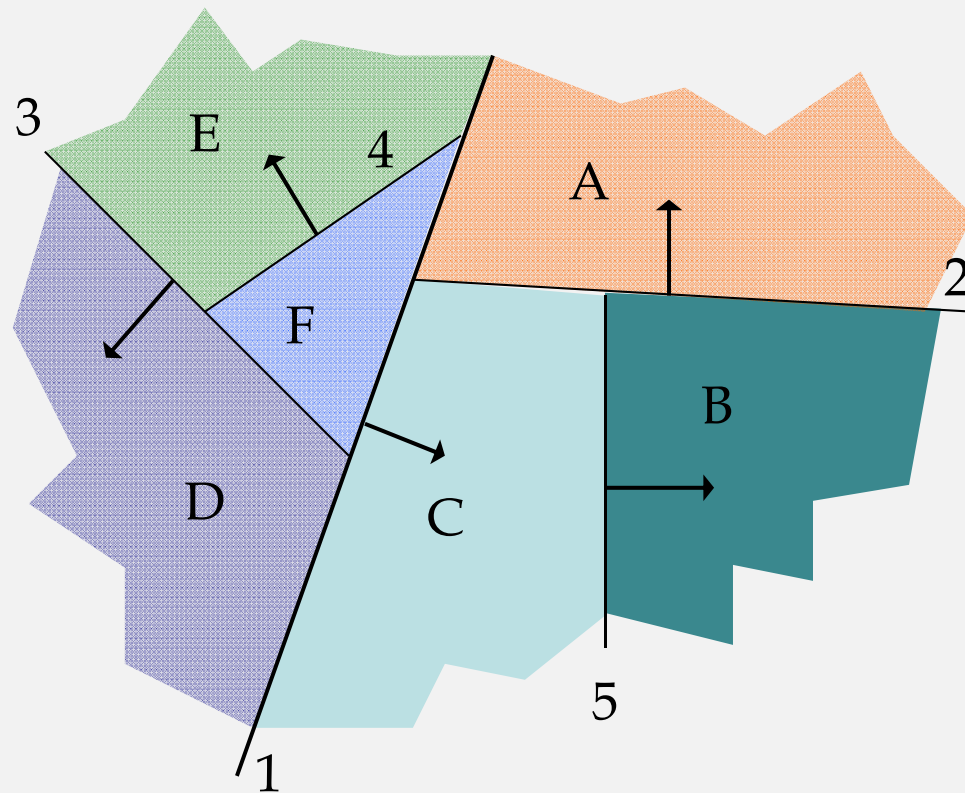
# Algoritmo de Recorte Sucessivo

- Vantagens
  - ♦ Trabalha no espaço do objeto
    - Independe 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 coleções de objetos busca-se uma divisão aproximadamente eqüânime
  - ♦ Se estamos partindo polígonos
    - (2D), normalmente usa-se a direção de alguma aresta como suporte para o plano
    - (3D), normalmente usa-se a orientação do plano de suporte do de algum polígono
  - ♦ Se os objetos têm extensão, é importante escolher planos que interceptem o menor número possível de objetos

# 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 (arquitetura, museus, jogos)
- Diversas variantes
  - ♦ Desenhar de trás para frente (algoritmo do pintor)
  - ♦ Desenhar de frente para trás (algoritmo de recorte recursivo)
  - ♦ Outras ...



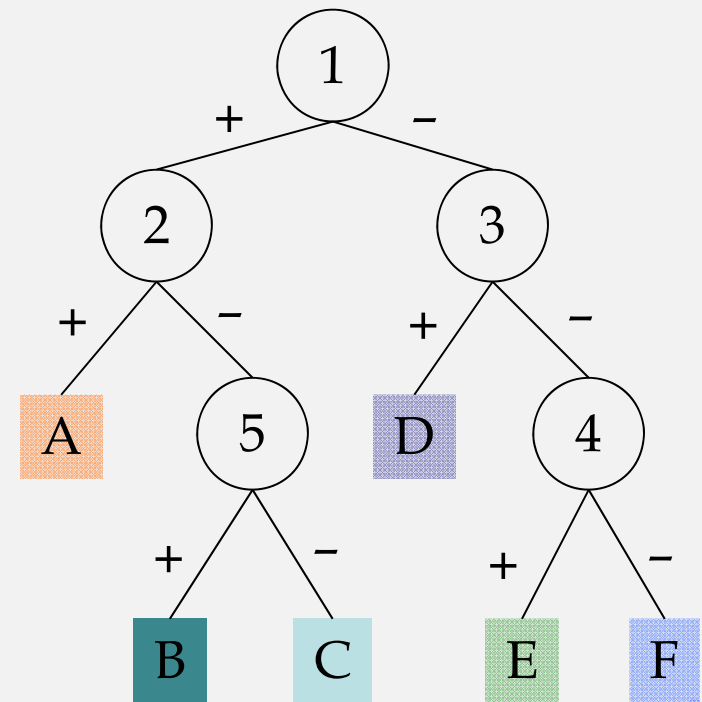
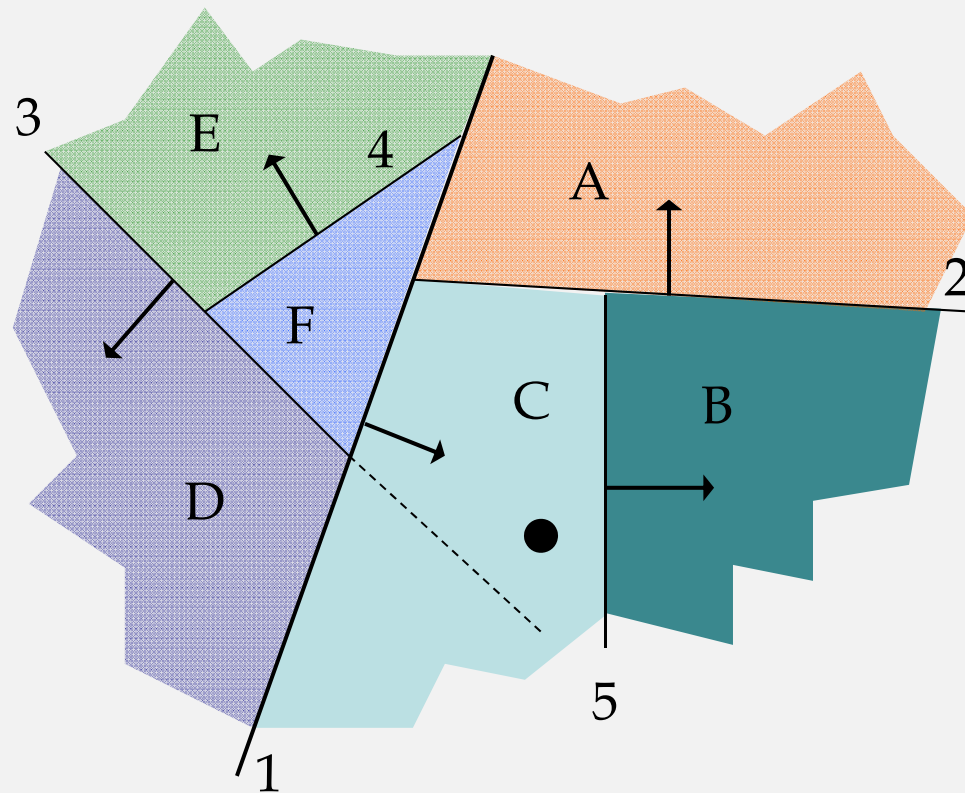
## BSP-trees - Construção

- Escolhe-se um dos polígonos da coleção presente na célula (ao acaso?)
  - ♦ Não existe algoritmo ótimo
  - ♦ Algumas heurísticas (ex.: minimum stabbing number)
- Divide-se a coleção em duas sub-coleções (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 frustum de visão

# BSP-Tree



Ordem de desenho: D E F A B C

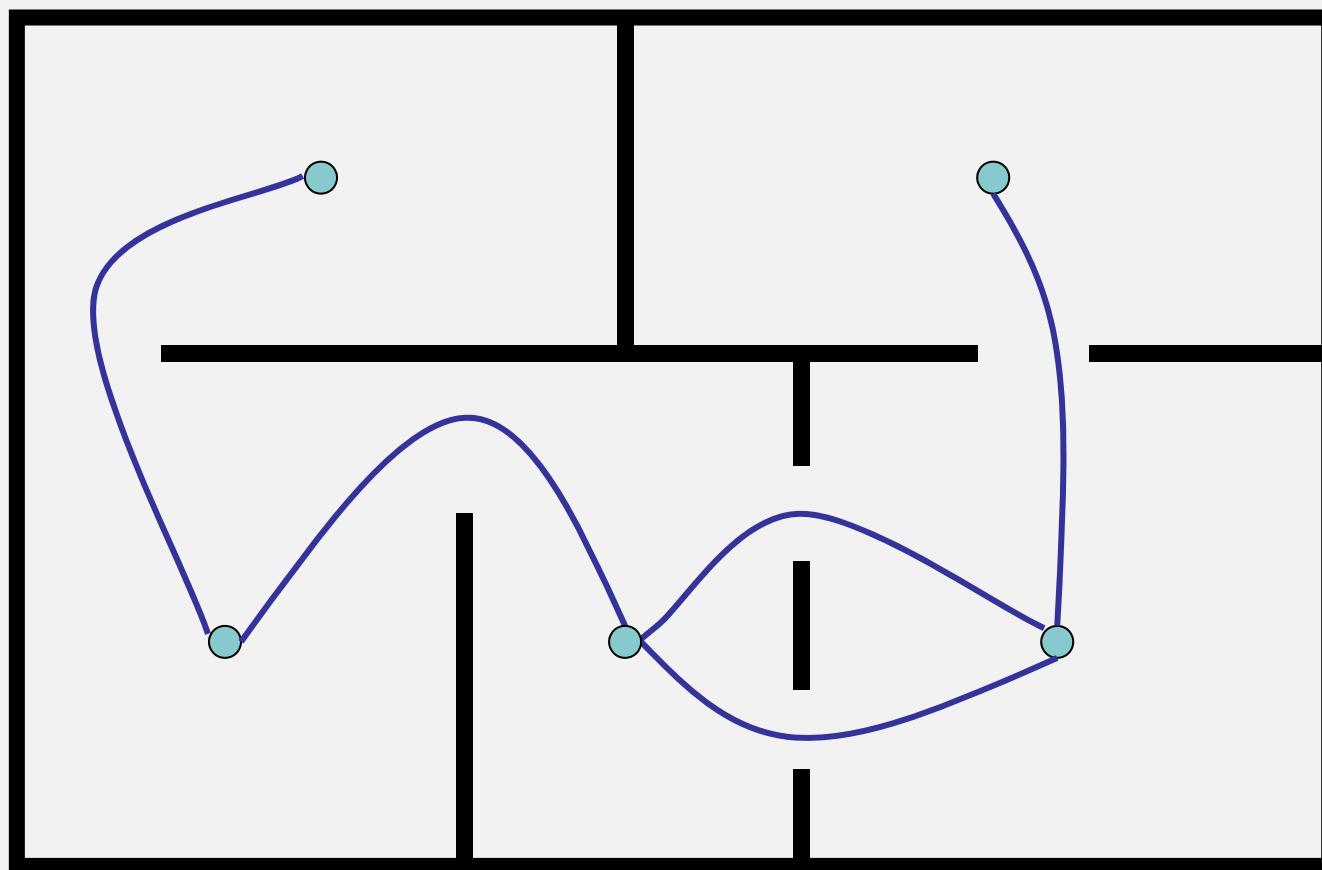
# BSP-trees

- 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

- Idéia usada em aplicações de caminhada (*walkthrough*) por ambientes virtuais do tipo arquitetônico
  - ♦ Cena composta de diversos compartimentos (quartos, salas, etc)
- Visibilidade é determinada convencionalmente dentro de cada compartimento (célula)
- Visibilidade entre células requer que luz atravessasse 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

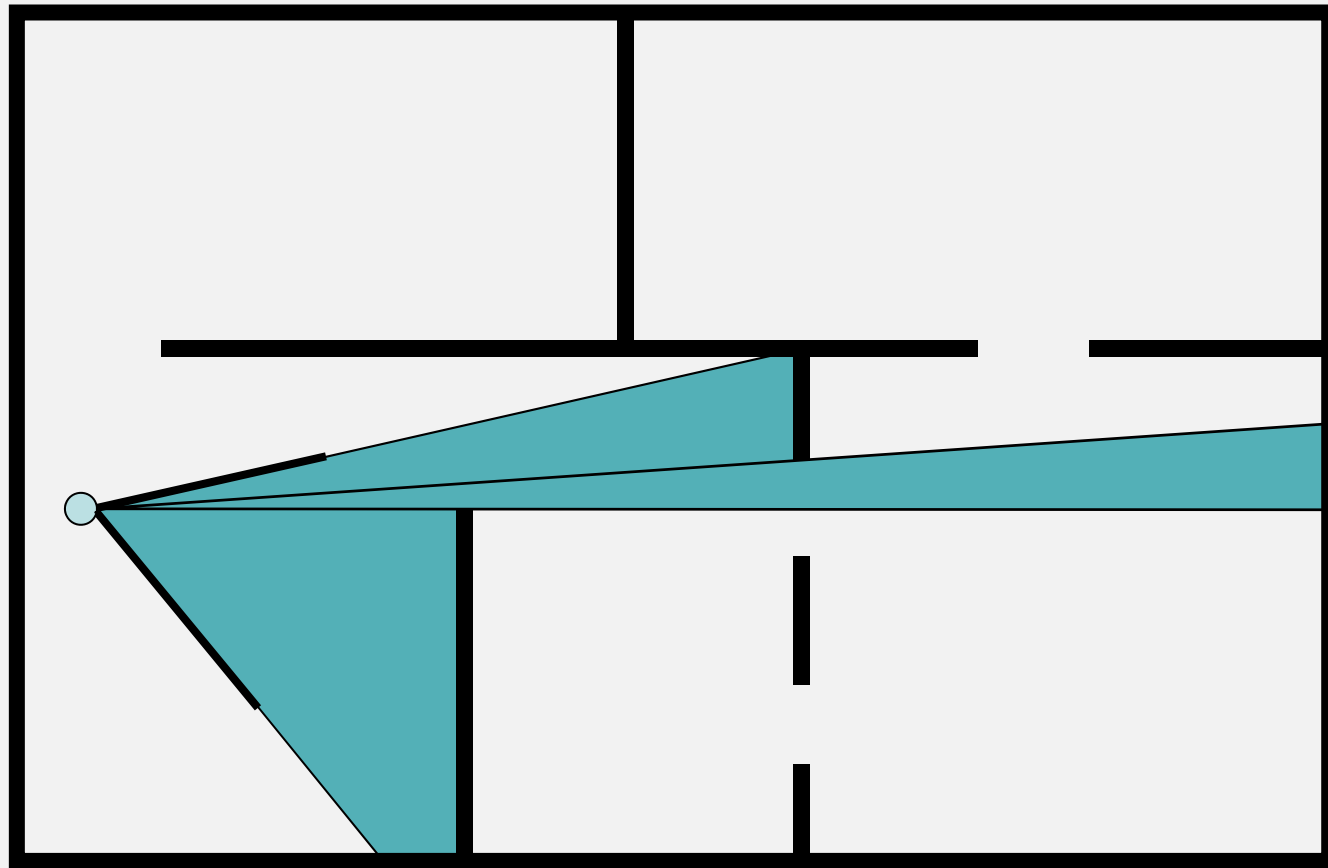
## Células e Portais



## Células e Portais - Algoritmo

- Desenhar célula  $C$  (paredes, objetos) 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

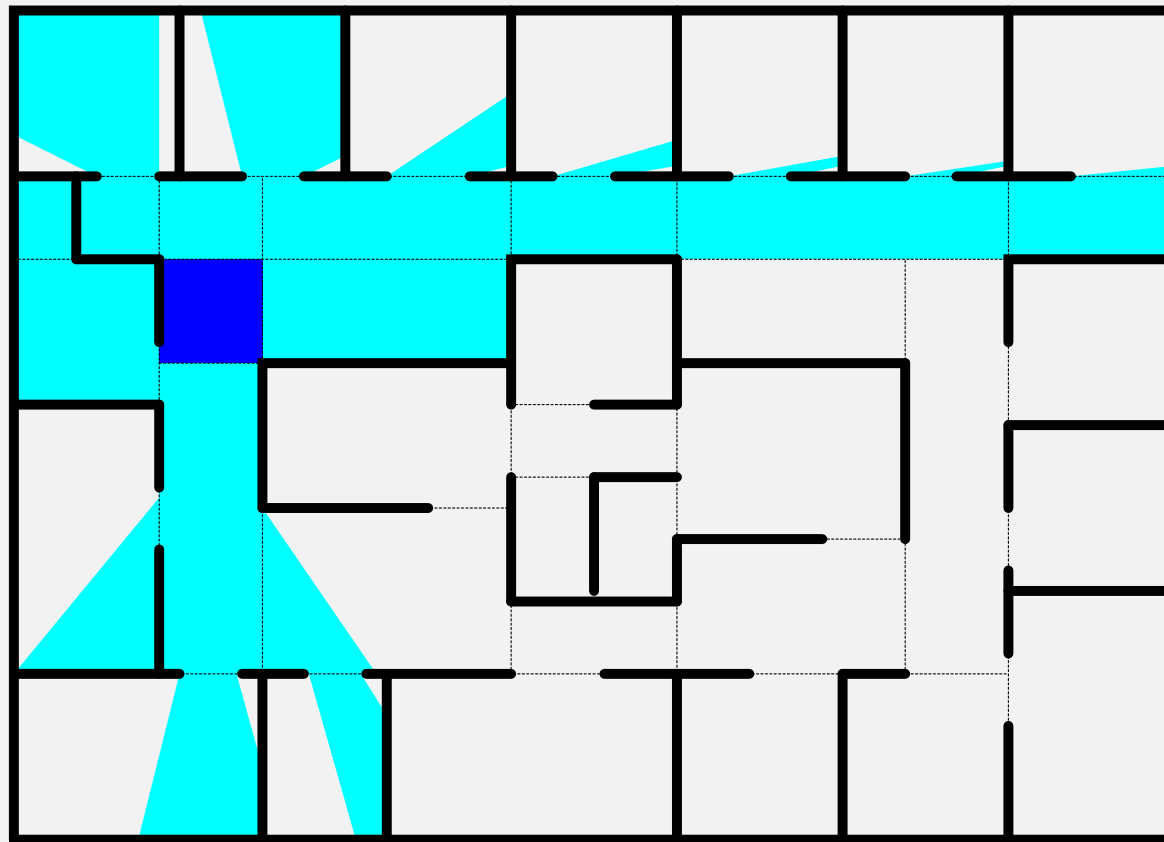




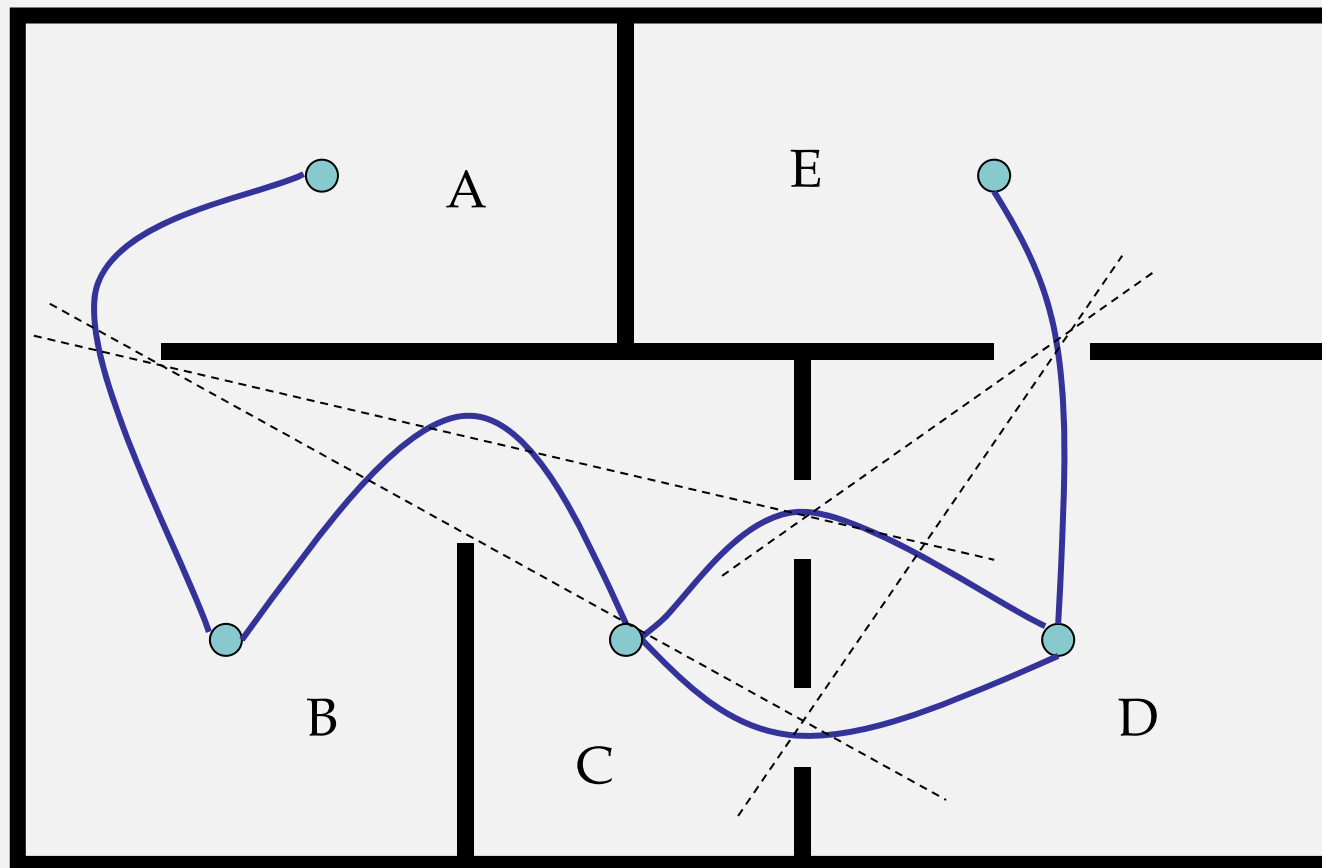
# Células e Portais – Visibilidade Pré-Computada

- Operações de recorte são complexas
  - ♦ Volume recortado pode ter um grande número de faces
- Idéia: 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 direção
- Informação de visibilidade
  - ♦ Célula a Região (estimativa exata)
  - ♦ Célula a Célula (estimativa grosseira)
  - ♦ Célula a Objeto (estimativa fina)

## Visibilidade Célula a Região



# Visibilidade Célula a Célula

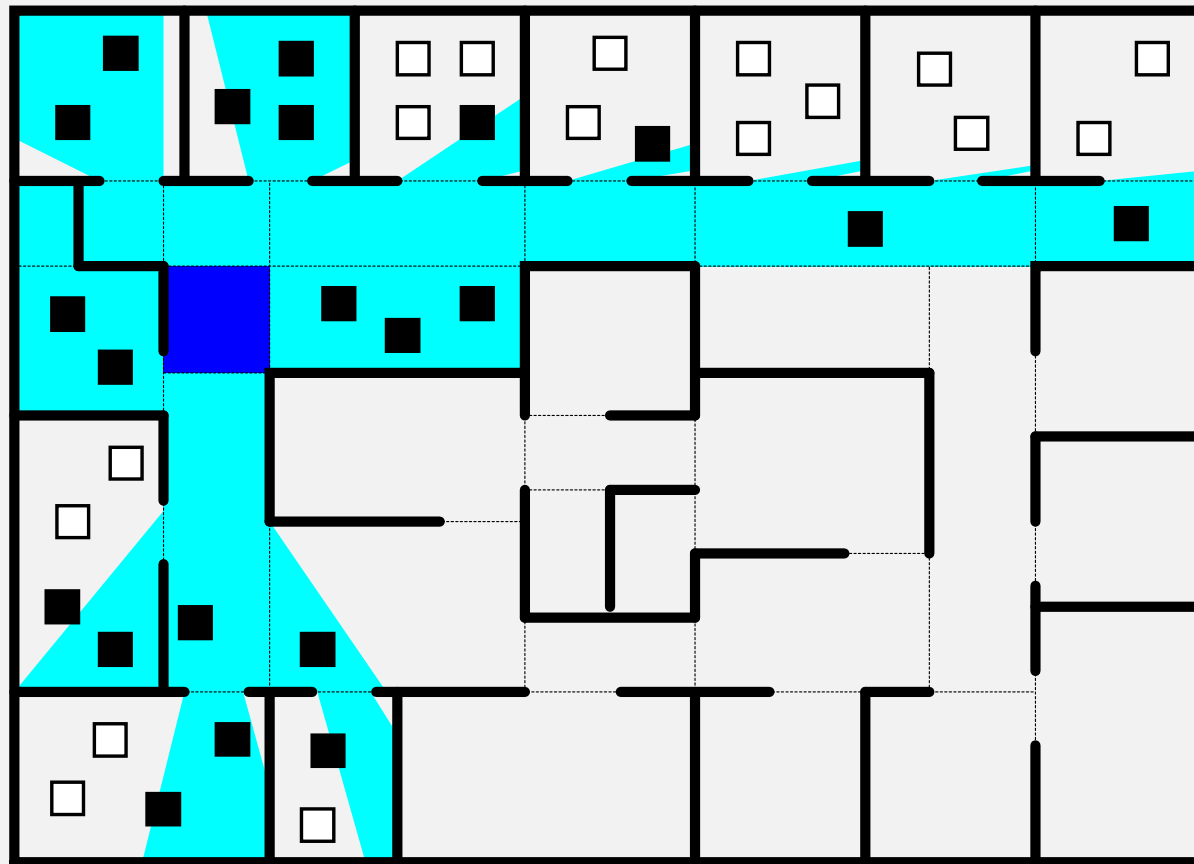


**1 Portal**  
AB,BC,CD,DE

**2 Portais**  
AC,BD,CE

**3 Portais**  
AD

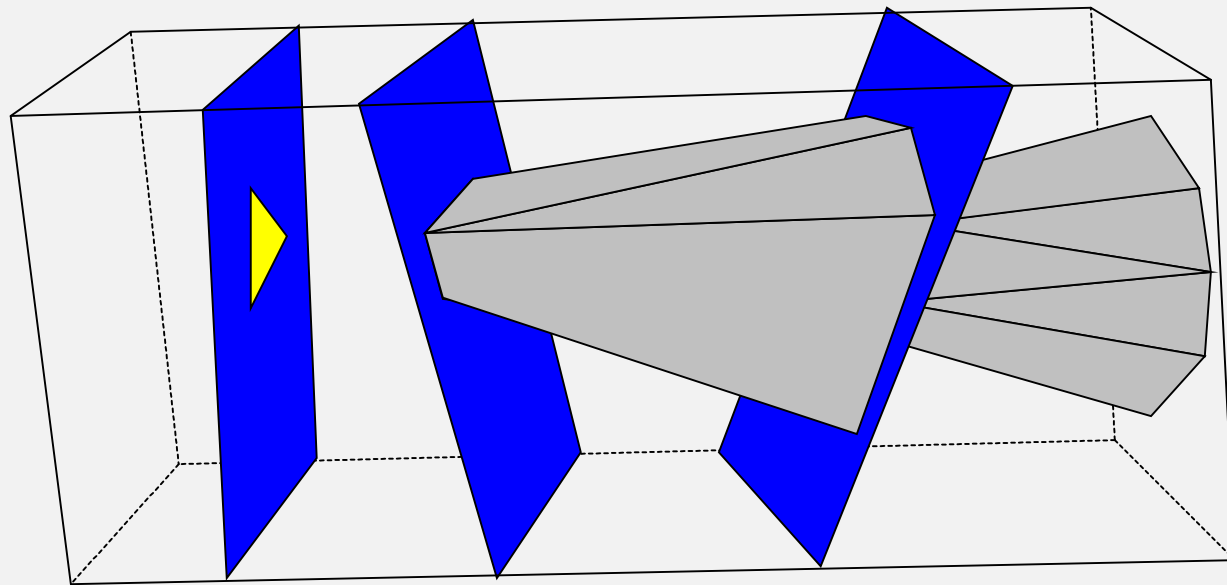
# Visibilidade Célula a Objeto



## 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 seqüência de portais
  - ◆ Usar z-buffer
  - ◆ Se dados de visibilidade célula-a-objeto estiver disponível, desenhar apenas os objetos 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 objetos 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