INF 2063 – Visualização de Modelos Massivos

Trabalho de Pesquisa

Descarte por Oclusão CHC++ e FastV

Vitor Barata R. B. Barroso
vbarata@tecgraf.puc-rio.br

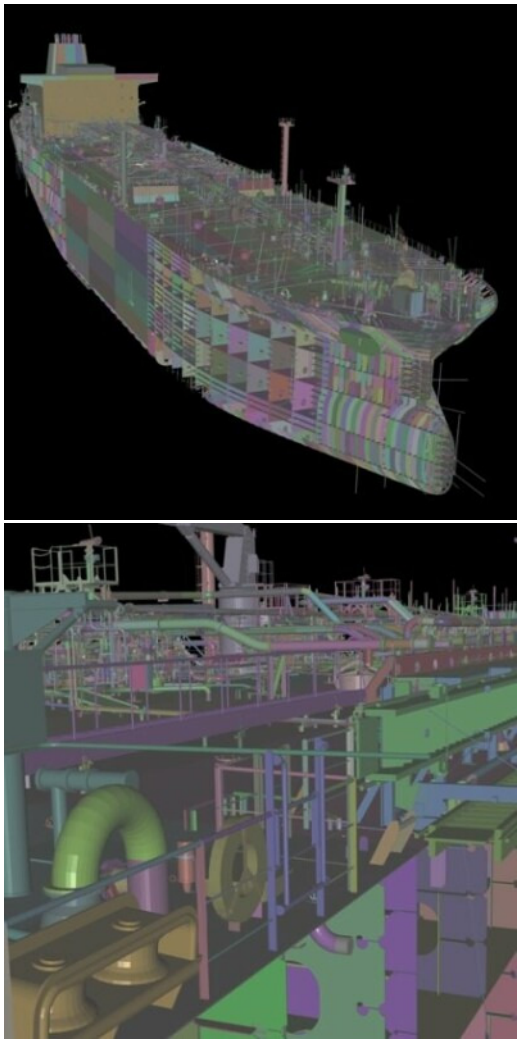
Motivação – Descarte por Visibilidade

- Renderização de cenas complexas em tempo real
  - Modelos CAD detalhados
  - Formas escaneadas
  - Resultados de simulações físicas, como de fluidos
  - Cenas complexas em geral

- Objetivo
  - Processar e enviar à placa gráfica apenas o conjunto de primitivas necessário para formar a imagem final (PVS – *Potentially Visible Set*)
  - Descartar, com testes rápidos, toda ou a maior parte da geometria que não contribui para o resultado final

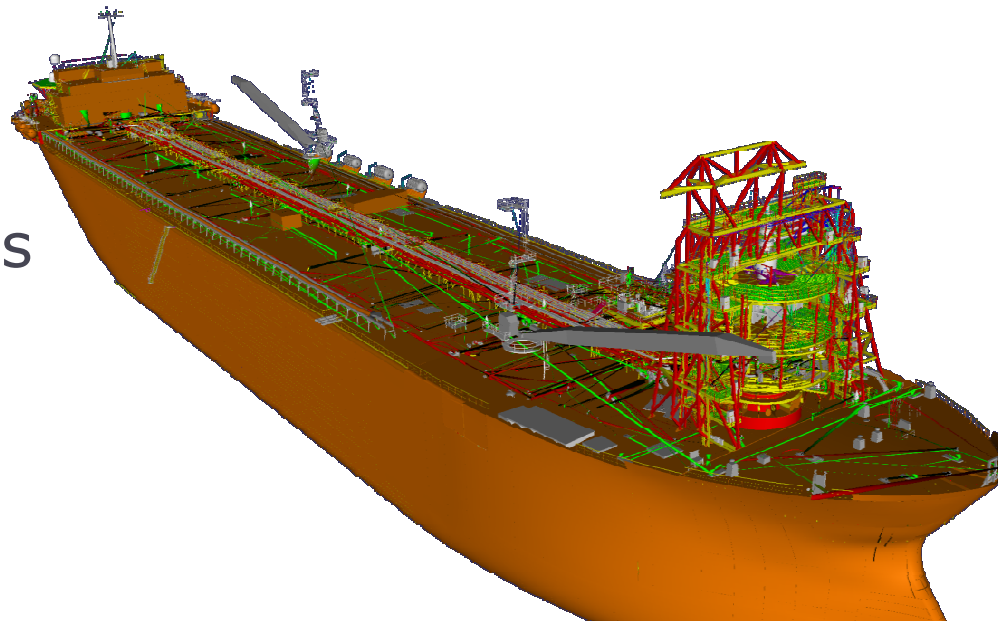
Exemplo de Modelo CAD Complexo

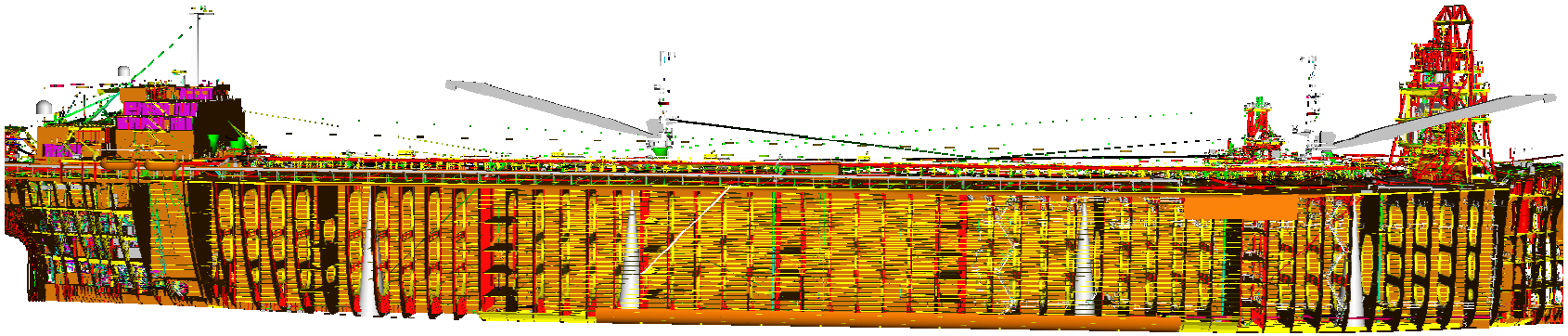
Double Eagle Tanker
(fonte: NVNews - NVIDIA GeForce3 Preview)



Exemplo de Modelo CAD Complexo

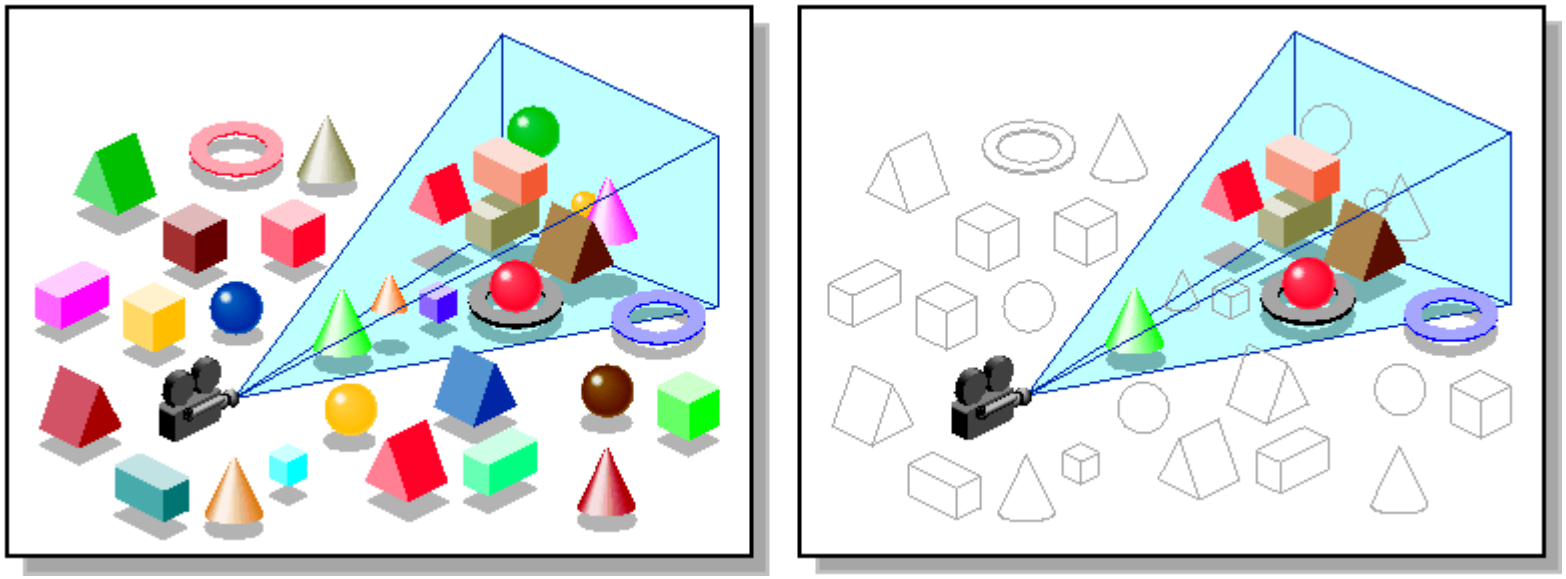
- P-38
  - 90.871 objetos
  - 4.717.749 vértices





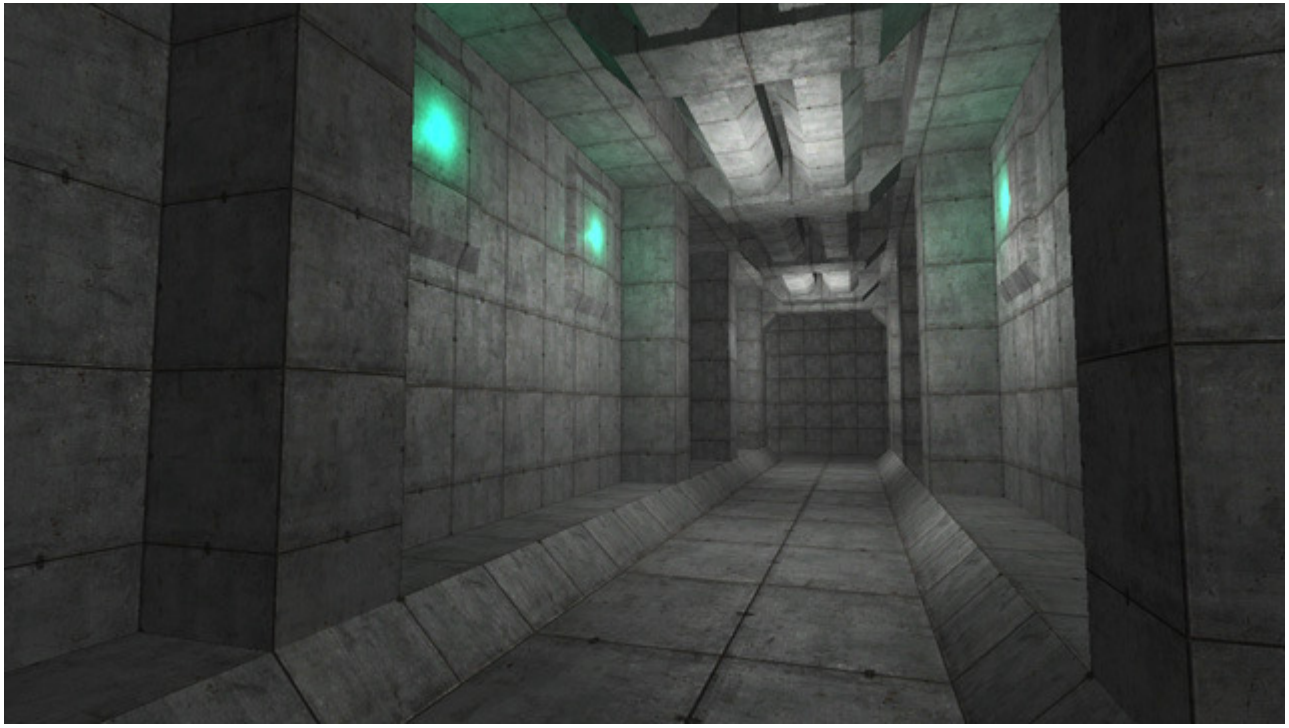
Descarte por Visibilidade

- Principais tipos de descarte
  - Contra o volume de visão: primitivas fora do campo visual
  - Por oclusão: primitivas escondidas atrás de outras primitivas



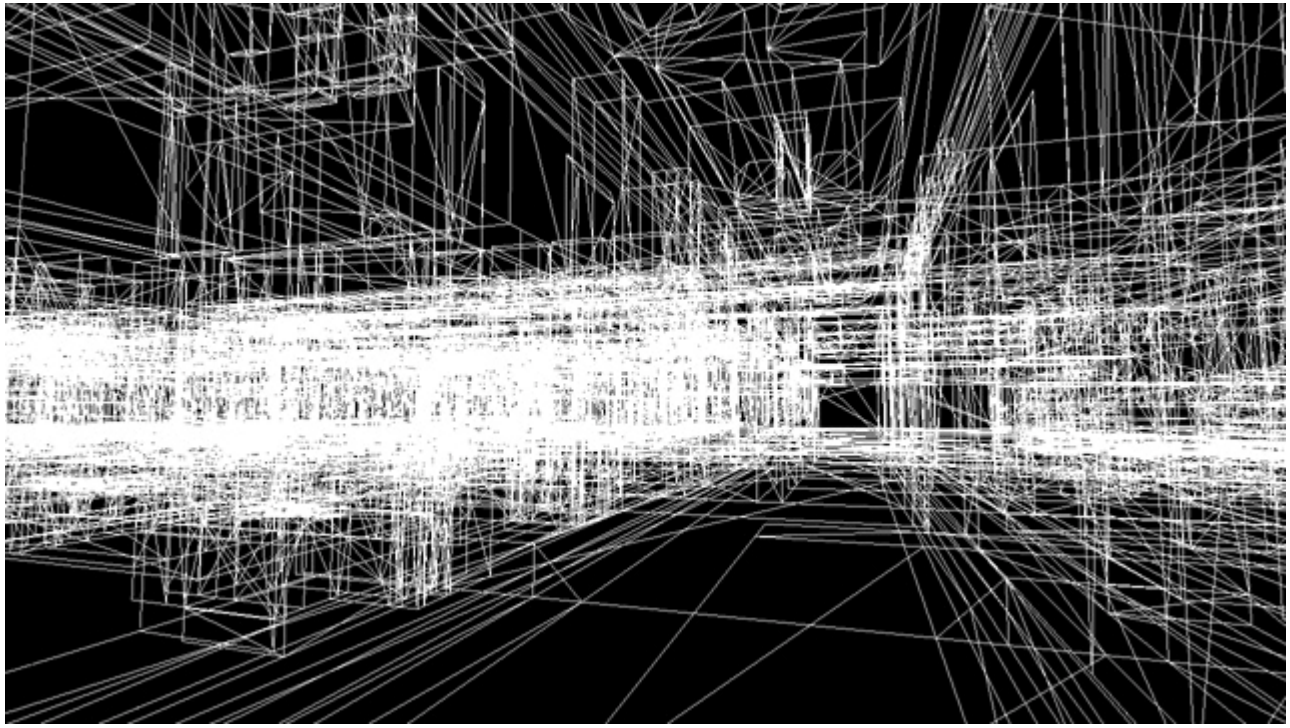
Descarte por visibilidade contra volume de visão e por oclusão
(fonte: OpenGL optimizer programmer's guide: an open API for large-model visualization)

Exemplo de Motor de Jogo (CHC++)



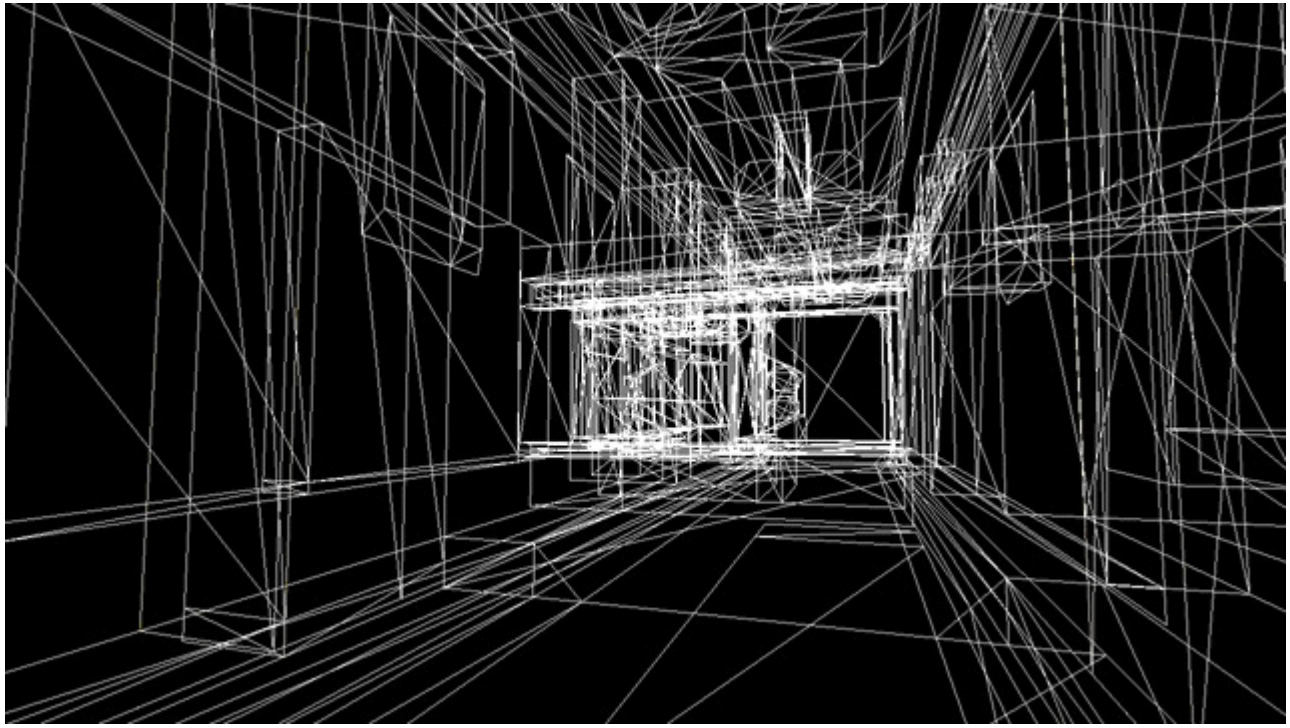
Unknown Worlds - Natural Selection 2
(http://www.unknownworlds.com/ns2)

Exemplo de Motor de Jogo (CHC++)



Unknown Worlds - Natural Selection 2
(http://www.unknownworlds.com/ns2)

Exemplo de Motor de Jogo (CHC++)



Unknown Worlds - Natural Selection 2
(http://www.unknownworlds.com/ns2)

Descarte por Oclusão

- Processamento
  - *Online* (por ponto): O PVS é determinado dinamicamente, em tempo de execução, para o ponto onde se encontra o observador no momento.
  - *Offline* (por região): Em pré-processamento, determina-se o PVS para vários pontos-de-vista espalhados pela cena. Em tempo de execução, escolhe-se e utiliza-se o PVS do ponto mais próximo.
- Abordagem
  - Exata: Determinam-se exatamente as primitivas visíveis ao observador.
  - Conservativa: Primitivas oclusas podem ser incluídas no PVS para economizar cálculos.
  - Agressiva: Primitivas visíveis podem ser excluídas do PVS.
- Espaço
  - Do objeto: o PVS é determinado a partir de testes envolvendo a geometria dos objetos da cena em seu espaço original. (Fast-V)
  - Da imagem: consideram-se os objetos após a rasterização, ou seja, uma amostragem de suas distâncias até o observador, como a fornecida pelo próprio z-buffer. (CHC++)

Teste de Oclusão em Hardware

- Permite perguntar à placa gráfica quantos pixels seriam rasterizados numa operação de desenho qualquer
  - Considera teste de z com z-buffer atual
- Operação assíncrona
  - Chamada retorna imediatamente
  - Resultado demora algum tempo para ficar disponível
  - Pode-se consultar rapidamente se o resultado já está pronto
- Problema: ociosidade da CPU e da GPU
  - Se o resultado for requisitado cedo demais, CPU espera até que esteja disponível
  - Falta de novos dados para a GPU deixa-a subutilizada

Teste de Oclusão em Hardware

- Idéia básica
  - Desenhar os objetos da cena do mais próximo para o mais distante
  - Fazer uma *occlusion query* enviando o volume envolvente do objeto
  - Se o resultado não indicar a rasterização de nenhum fragmento, a geometria correspondente está oclusa e não precisa ser desenhada
- Pontos positivos
  - Todos os objetos da cena podem ser oclusores e oclusos, independente de sua complexidade
  - Fusão de oclusores: um objeto pode ser escondido por mais de um oclusor em locais diferentes, uma característica que é capturada automaticamente
  - Utilização do poder de processamento paralelo da GPU
  - Facilidade de integração a algoritmos de rendering

Teste de Oclusão em Hardware

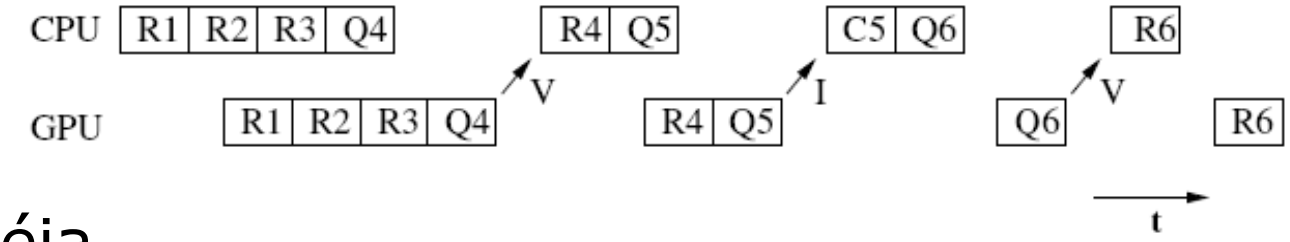
- Cuidados básicos
  - Volume envolvente deve ser rápido de se desenhar
    - preferir uma caixa a uma esfera
  - Volume envolvente deve se ajustar bem à geometria
    - Volumes grandes demais tornam o teste ultra-conservativo
    - Minimizar tamanho do PVS
  - Desenho do mais próximo para o mais distante
    - Calcular a distância de cada objeto ao observador
    - Organizar uma fila de prioridades

Algoritmo Hierárquico

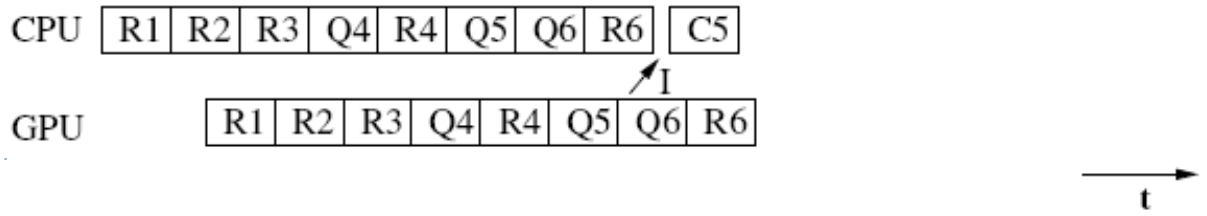
- Explorar a coerência espacial da cena
  - Organizar a cena numa hierarquia
    - Subdivisão espacial (grid regular, octree, kd-tree)
    - Hierarquia de volumes envolventes (AABB, OBB)
    - Critério de subdivisão do espaço ou de agrupamento de primitivas
  - Percorrer os nós da hierarquia
    - Testar contra o volume de visão
    - Testar por oclusão e esperar o resultado
    - Se o nó estiver visível
      - Percorrer os filhos do mais próximo ao mais distante
      - Desenhar a geometria das folhas
    - Se o nó estiver invisível
      - Interromper o percorrimento deste ramo da hierarquia

Descarte Coerente Hierárquico (CHC)

- Ociosidade do CPU e da GPU



- Idéia
  - Explorar a coerência temporal da classificação de visibilidade
    - Visível tende a continuar visível e vice-versa
  - Reorganizar as chamadas
    - Intercalar testes de oclusão com outras operações
    - Enquanto espera, CPU continua processando e enviando dados à GPU

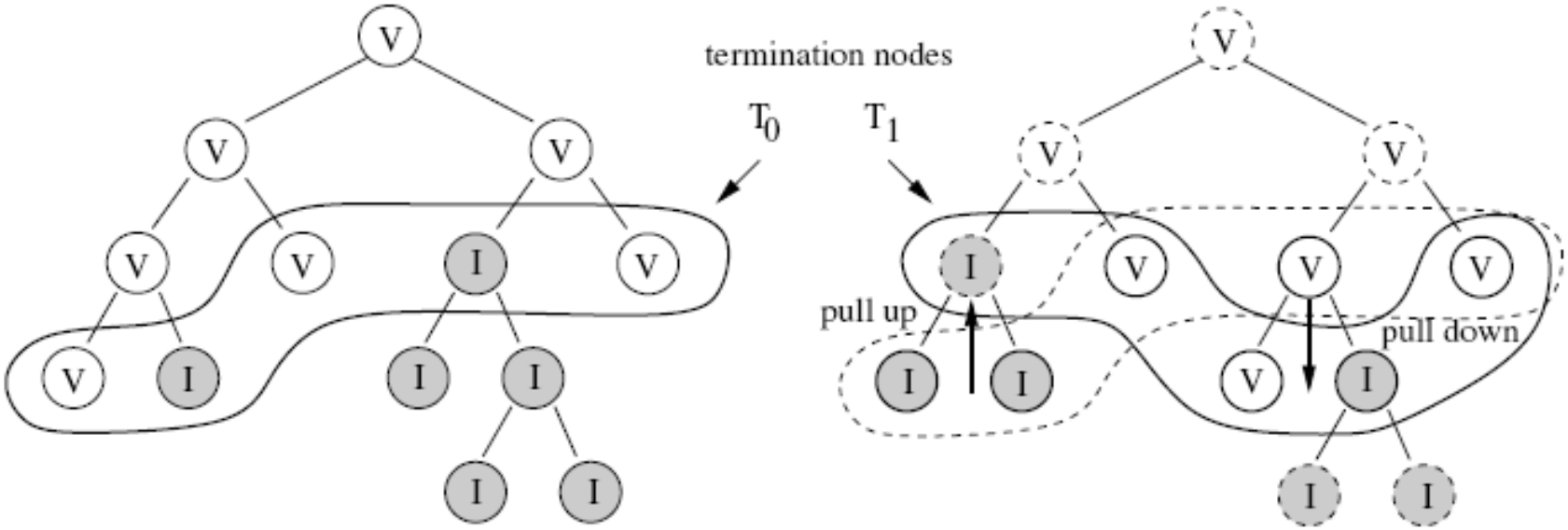


Descarte Coerente Hierárquico (CHC)

- Nós de término de um percorrimento
  - Nós internos invisíveis
  - Folhas, visíveis ou não, com pai visível
- Algoritmo
  - Fila de testes
    - Testes requisitados são armazenados numa fila até que seus resultados estejam disponíveis
  - Percorrimento começa nos nós de término do quadro anterior
    - Nós internos anteriormente visíveis são pulados
    - Nós previamente invisíveis: requisitar o teste e armazenar na fila
    - Folhas previamente visíveis
      - requisitar e armazenar o teste
      - desenhar imediatamente assumindo que a visibilidade é mantida
      - o resultado do teste será usado no quadro seguinte

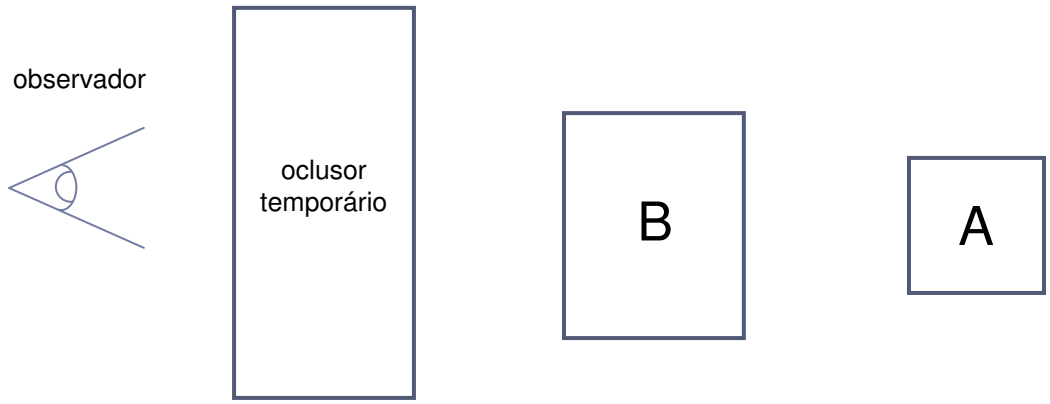
Descarte Coerente Hierárquico (CHC)

- Sempre que houver um resultado de teste disponível
  - Se visível, processam-se os filhos ou desenha-se a geometria
  - Se invisível, descarta-se toda a sub-árvore com raiz no nó ocluso
- Propagação de visibilidade
  - No percorrimento, marcamos todo nó visitado como invisível
  - Quando um nó visível é detectado, propagamos a mudança para os ancestrais
  - Permanecem invisíveis apenas nós cujos filhos sejam todos invisíveis



Descarte Coerente Hierárquico (CHC)

- Inconsistência
  - Imagine dois objetos A e B, ambos invisíveis em certo quadro
  - No quadro seguinte, o teste de oclusão de A pode ser iniciado antes que se termine o de B
  - Se B ficou visível, somos incapazes de detectar uma possível oclusão de A por B
  - Isso “apenas” torna o algoritmo mais conservativo!



Descarte Coerente Hierárquico (CHC)

- Otimizações
  - Objetos visíveis tendem a continuar visíveis
    - Podemos refazer seus testes de oclusão apenas a cada n_{av} quadros
  - Tolerância em pixels para visibilidade
    - Consideramos visíveis apenas objetos que rasterizem pelo menos n_{vp} fragmentos
  - Espera nula da CPU
    - Sempre que formos forçados a esperar o resultado de um teste de oclusão, podemos aproveitar e já desenhar algum objeto da fila
      - Escolher sempre os objetos mais leves para não desperdiçar trabalho demais caso ele esteja oculto afinal de contas

Algoritmo CHC++

- Problemas da abordagem anterior
  - Desempenho
    - Grande número de testes de oclusão
    - Grande número de mudanças de estado da renderização
    - Existência ainda de algum tempo de CPU em espera
    - Desenho muito conservador, de mais geometria do que o necessário
  - Aplicabilidade
    - Impossibilidade de integração com algoritmos de renderização otimizados, com coleta de objetos separados por materiais e *shaders*

Algoritmo CHC++

- Testes de visibilidade em lote
  - Minimizar mudanças de estado da renderização
  - Grupos (lotes) de testes são requisitados conjuntamente

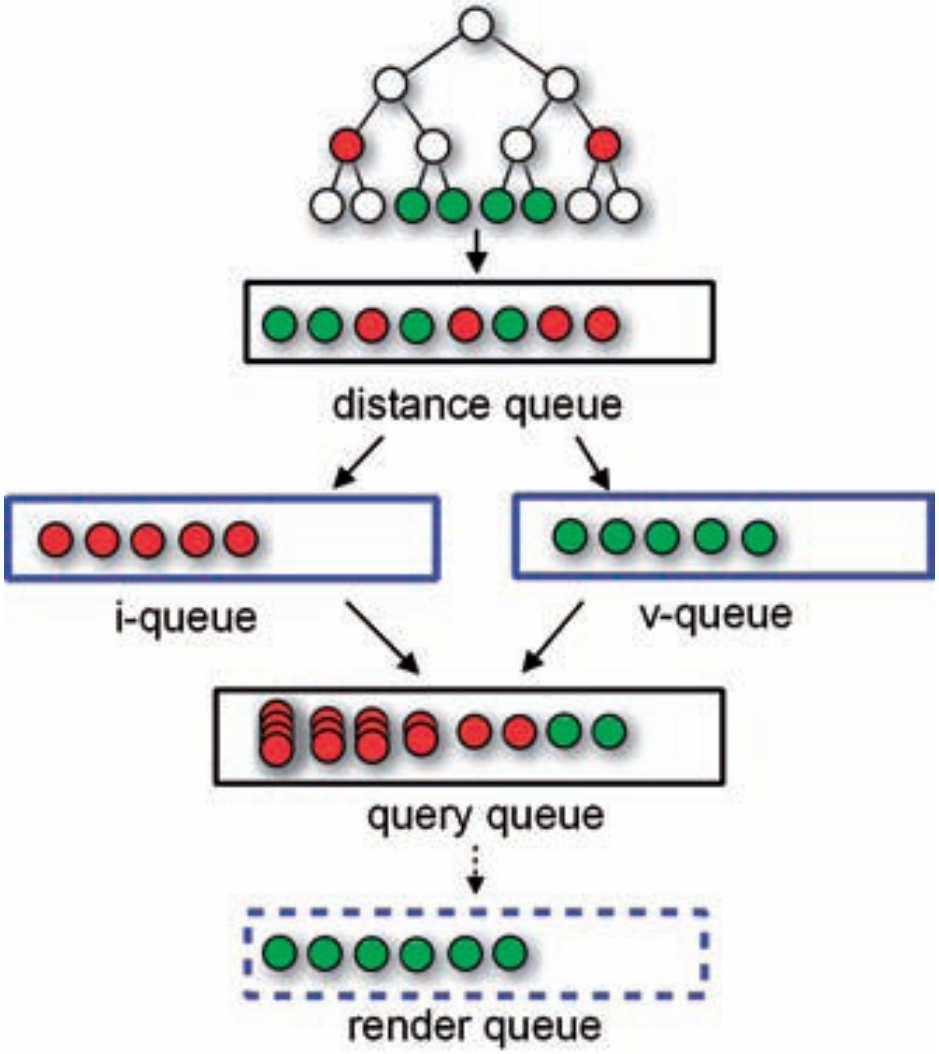
- Lotes de nós previamente invisíveis
  - Em vez de requisitar imediatamente, colocar na "fila-i"
  - Quando o número de nós na fila chegar a um valor *b*, fazem-se todas as requisições de uma vez
  - *Tradeoff*
    - Menos mudanças de estado x Atraso nos resultados dos testes
    - Algoritmo não é muito sensível, bons resultados para $20 < b < 80$

Algoritmo CHC++

- Lotes de folhas previamente visíveis
  - Geometria desenhada imediatamente, independente do teste
  - Em vez de requisitar imediatamente, colocar o teste na fila-v
  - Resultados só serão úteis no próximo quadro!
    - Requisitar testes da fila-v apenas quando não houver mais nós da hierarquia a percorrer ou testes já prontos para processar
    - Ao final do quadro, requisitar os testes restantes na fila-v

- Integração a algoritmos de renderização
  - Em vez de desenhar imediatamente, colocar numa fila de renderização
  - Renderizar toda a fila logo antes de qualquer teste de oclusão
  - Fila pode ser ordenada por materiais, *shaders*, ou como desejado

Algoritmo CHC++



Algoritmo CHC++

- Multitestes
  - Minimizar número de testes de oclusão
  - Se toda uma parte da cena está invisível, basta um único teste para saber se ela continua invisível
    - Ex.: cena e observador estáticos
  - Idéia:
    - Agrupar nós previamente invisíveis que têm a mesma probabilidade de permanecerem assim no quadro atual
    - Fazer apenas um teste desenhando todos os nós
      - Se invisível, mantém todos invisíveis e economiza vários testes
      - Se visível, testa um por um (faz apenas o primeiro teste a mais)
    - Probabilidade:
      - Quanto mais tempo um nó se mantiver sem alterar sua classificação, mais provável é que ele continue se mantendo assim

Algoritmo CHC++

- Custo-benefício de multitestes
  - Número de quadros que um nó N mantém sua classificação inalterada:

    i_N (persistência de visibilidade)
  - Probabilidade de o nó continuar mantendo sua classificação como invisível:

    $p_{keep}(i) \approx 0{,}99 - 0{,}7 \cdot e^{-i}$ (ajustado a medidas empíricas)
  - Probabilidade de o multiteste M falhar (algum nó $N \in M$ se tornar visível):

    $p_{fail}(M) = 1 - \Pi_{\forall N \in M} [p_{keep}(i_N)]$
  - Custo de um multiteste: número de testes esperado

    $C(M) = 1 + p_{fail}(M) \cdot |M|$
  - Benefício de um multiteste: número de nós

    $B(M) = |M|$
  - Custo-benefício:

    $V(M) = B(M) / C(M)$

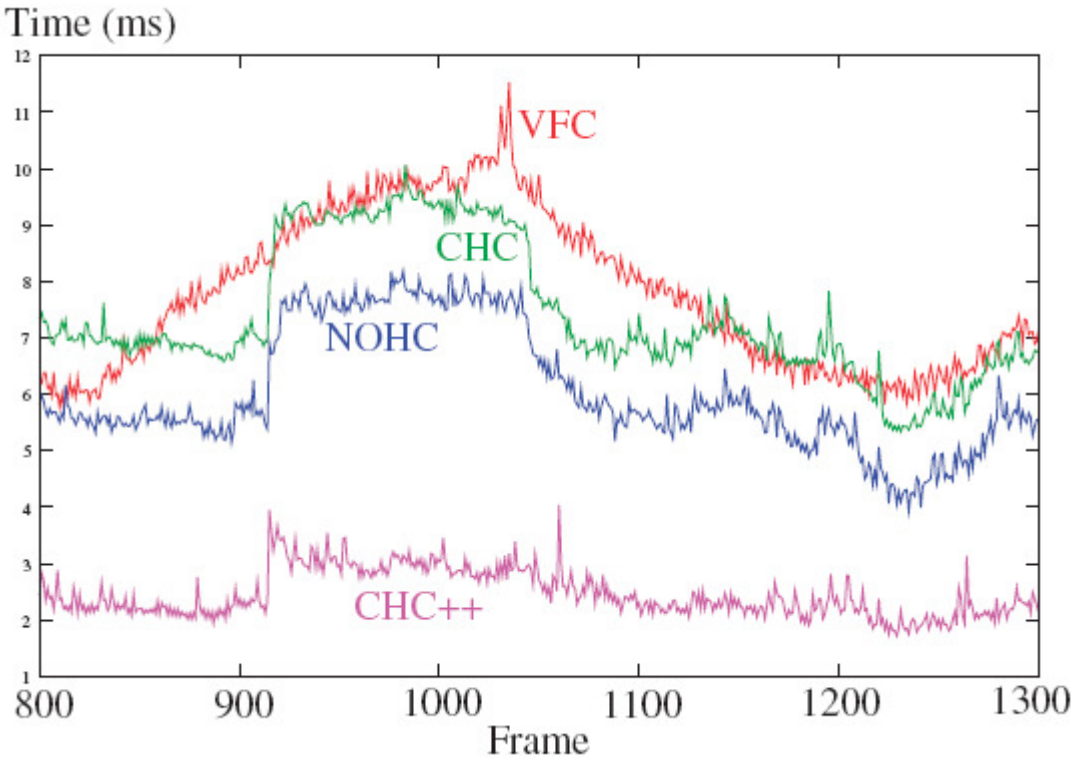
- Estratégia gulosa de agrupamento:
  - Agrupar os nós seguindo a ordem de $p_{keep}(i_N)$, até achar V(M) máximo
  - Produz grupos maiores para p_{keep} mais alto!

Algoritmo CHC++

- CHC: objetos visíveis tendem a continuar visíveis
  - Refazer seus testes de oclusão apenas a cada n_{av} quadros
  - Alinhamento de testes de oclusão quando objetos se tornam visíveis simultaneamente!
    - Quando um objeto se torna visível, a espera até o primeiro teste de oclusão será de r quadros, com r aleatório entre 0 e n_{av}
- Volumes envolventes mais justos
  - Uso de AABBs simples
  - Para nós internos, volume é a união dos volumes dos filhos!
    - Refinamos o volume até um nível d_{max} abaixo do pai
    - Paramos de refinar se os volumes filhos tiverem muita sobreposição

Algoritmo CHC++

- Resultado:
  - Estado-da-Arte atual em descarte por oclusão por ponto!

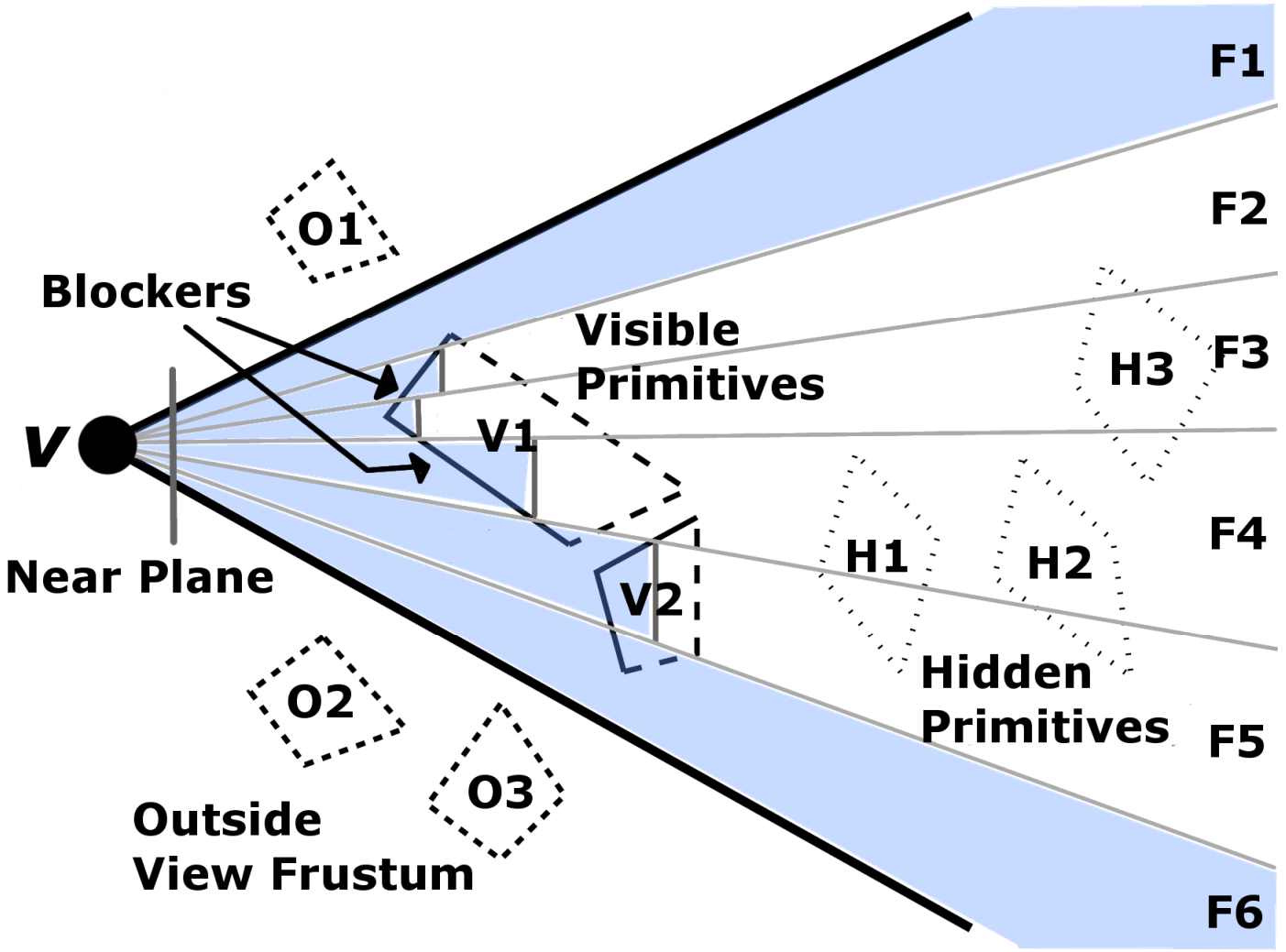


Algoritmo Fast-V

- Idéia básica
  - Subdividir o volume de visão em um grande número de pequenos *frusta*
    - Subdivisão uniforme ou adaptativa
  - Encontrar um oclusor, composto por triângulos conectados, que bloqueie cada *frustum*
  - Ajustar o plano *far* e calcular o PVS de cada *frustum*
  - O PVS final é dado pela união dos PVS individuais

- Espaço do objeto
  - Calcula o subconjunto de primitivas que são atingidas por qualquer raio traçado pelo interior do volume de visão
    - Pode calcular PVS muito conservativo, com objetos através de frestas menores que um pixel!
  - Oclusores devem ser homeomorfos a discos

Algoritmo Fast-V



Algoritmo Fast-V

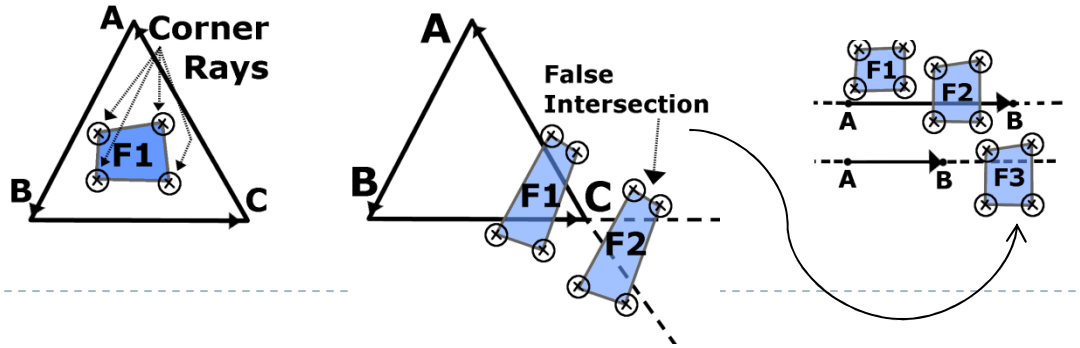
- Pontos positivos
  - Aplicável a quaisquer objetos triangulados, independente de seu tamanho ou disposição
  - Primeiro algoritmo no espaço do objeto a ter desempenho suficiente para ser aplicável em tempo real
  - PVS conservativo, que converge para o exato conforme aumenta a subdivisão de *frusta*, com precisão a nível de primitivas
  - Escalável em multiprocessadores, cada *frustum* é independente
- Pontos negativos
  - Não faz fusão de oclusores (PVS maior que o necessário)
  - Otimizado apenas para modelos com informações topológicas
  - Não tem extensão para modelos massivos *out-of-core*
  - Mais lento e difícil de implementar do que CHC
  - Parece convergir para espaço da imagem com um *frustum* por pixel

Algoritmo Fast-V

- Representação de um *frustum*
  - 4 raios, partindo do observador, por cada aresta da pirâmide
  - Planos *near* (fixo) e *far* (ajustável) paralelos
- Hierarquia de volumes envolventes (AABB)
  - Interseção: o *frustum* está sendo totalmente/parcialmente bloqueado pela AABB?
- Triângulos nas folhas
  - Representados por suas 3 arestas orientadas
  - Interseção: O *frustum* está sendo totalmente/parcialmente bloqueado por dado triângulo?

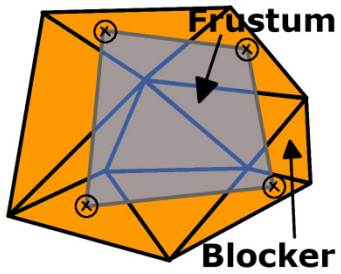
Algoritmo Fast-V

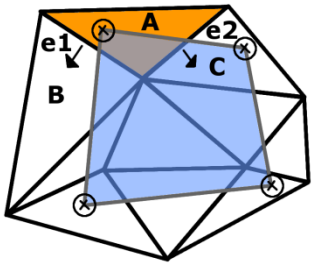
- Coordenadas de *Plücker*
  - Uma linha orientada no espaço 3D pode ser caracterizada por 6 coordenadas homogêneas usando-se dois vetores:
    - **d** – Vetor entre dois pontos **x** e **y** da linha, dados em relação à origem
    - **m** – Produto vetorial (**x**×**y**)
  - Dadas duas linhas orientadas (d_1:m_1) e (d_2:m_2), o sinal de ($d_1 \cdot m_2 + m_1 \cdot d_2$) indica em qual sentido – ou orientação – uma linha precisa girar para se tornar coplanar à outra
  - Testes rápidos (e conservativos) de interseção
    - Um *frustum* está dentro de um triângulo se suas 4 arestas têm a mesma orientação em relação às 3 arestas do triângulo
    - Um *frustum* intercepta uma aresta se suas 4 arestas têm orientações diferentes em relação a ela

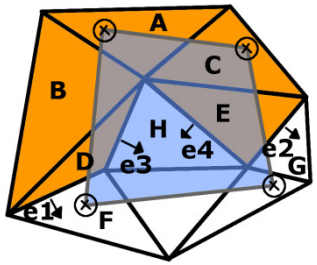


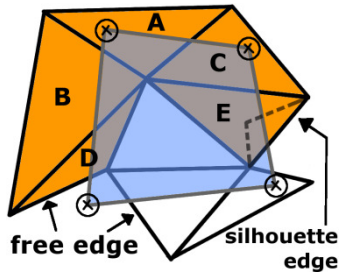
Algoritmo Fast-V

- Bloqueador de um *frustum*
  - Conjunto de triângulos conectados que bloqueia <u>todos</u> os raios internos ao *frustum*
  - Se um triângulo bloqueia parcialmente um *frustum*:
    - Encontramos as arestas interceptadas
    - Caminhamos pelos triângulos vizinhos adjacentes a essas arestas
    - Achamos um bloqueador se o caminhamento fechar um ciclo
    - Desistimos conservativamente se encontrarmos:
      - uma aresta livre (apenas um triângulo incidente)
      - uma aresta de silhueta (triângulos incidentes com normais opostas)



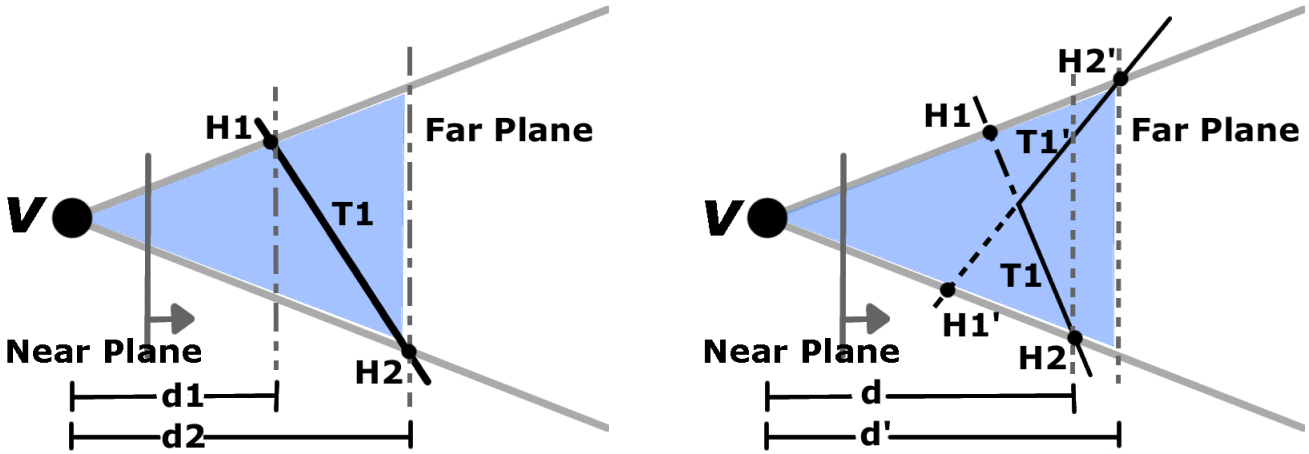






Algoritmo Fast-V

- Atualização do plano *far*
  - Para cada triângulo do bloqueador
    - Tomamos a distância, até o plano *near,* de cada ponto de interseção dos raios do *frustum* com o plano do triângulo
  - Escolhemos para o plano *far* a maior distância encontrada



Algoritmo Fast-V

- Subdivisão do volume de visão a partir do plano *near*
  - Caso mais simples: grid uniforme
  - Distribuição desigual de geometria pela cena: grid adaptativo
    - Estrutura de quadtree
    - *Frusta* com PVS muito grandes são subdivididos
      - os filhos são computados com base na área de oclusores parciais, numa tentativa de torná-los oclusores totais dos *sub-frusta*
  - Modelo sem topologia
    - Subdividir repetidamente os *frusta* até que cada um seja bloqueado por apenas um triângulo

- Implementação multi-core
  - Simples e escalável, já que o processamento de cada *frustum* é independente dos demais

Conclusão

- Fast-V é o primeiro algoritmo viável de descarte por oclusão por ponto no espaço do objeto
- Algoritmos no espaço da imagem ainda oferecem várias vantagens
  - Desempenho superior
  - Facilidade de implementação
  - Facilidade de incorporação a renderizadores
  - Fusão de oclusores e poder da GPU ganhos de graça
  - Independência da geometria da cena
    - Qualquer coisa que marque o z-buffer!
- Projeto:
  - Implementar CHC++
  - Incorporar ao grafo de cena SG do Tecgraf

Referências

[1] Notas de aula

[2] P. Santos, **Técnicas de Otimização para Visualização de Modelos Massivos** – Relatório Final de Projeto, PUC-Rio, 2006

[3] J Bittner,M Wimmer, H. Piringer, W. Purgathofer, **Cooherent Hierarchical Culling: Hardware Occlusion Made Useful** – Proceedings of Eurographics 2004

[4] O. Mattausch, J. Bittner, M. Wimmer, **CHC++: Coherent Hierarchical Culling Revisited** – Proceedings of Eurographics 2008

[5] A Chandak, L. Antani, M. Taylor, D. Manocha, **Fast-V: From-point Visibility Culling on Complex Models** – Proceedings of Eurographics 2009