Pontifícia Universidade Católica
do Rio de Janeiro

Karin Sulamita Leão Lisowski

Método da Oclusão Implícita e suas aplicações em visualização

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Matemática Aplicada do Departamento de Matemática da PUC–Rio

Orientador: Prof. Sinesio Pesco

Rio de Janeiro
Março de 2007

Pontifícia Universidade Católica
do Rio de Janeiro

Karin Sulamita Leão Lisowski

Método da Oclusão Implícita e suas aplicações em visualização

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Matemática Aplicada do Departamento de Matemática do Centro Técnico Científico da PUC–Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Sinesio Pesco
Orientador
Departamento de Matemática — PUC–Rio

Prof. Luiz Henrique de Figueiredo
IMPA

Prof. Geovan Tavares
Departamento de Matemática – PUC - Rio

Prof. Thomas Lewiner
Departamento de Matemática – PUC - Rio

Prof. Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico – PUC - Rio

Rio de Janeiro, 27 de Março de 2007

Karin Sulamita Leão Lisowski

Graduou–se em Licenciatura em Matemática na Universidade do Estado do Rio de Janeiro - UERJ (Rio de Janeiro, Brasil)

Ficha Catalográfica

Lisowski, Karin

Método da Oclusão Implícita e suas aplicações em visualização / Karin Sulamita Leão Lisowski; orientador: Sinesio Pesco. — Rio de Janeiro : PUC–Rio, Departamento de Matemática, 2007.

v., 79 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Matemática.

Inclui referências bibliográficas.

1. Matemática – Tese. 2. Extração de Isosuperfícies. 3. Descarte por oclusão. 4. Visualização Volumétrica. I. Pesco, Sinesio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Matemática. III. Título.

CDD: 510

Aos amantes da matemática e de suas aplicações

Agradecimentos

Neste momento especial da minha vida, agradeço ao Maior Matemático e Físico do universo: Deus, que nos dá capacidade e inteligência de estudar suas obras grandiosas.

Aos meus pais Fabiano e Esther por todo o amor, apoio e incentivo durante toda a minha vida. Aos meus irmãos Sara e Cleber, bem como minha cunhada Ana, e um agradecimento especial ao meu cunhado Edimar (engenheiro), que me ajudou a adquirir o gosto pela matemática.

Aos meus colegas do Tecgraf, em especial: André, Aurélio e Gui- lherme, pois com eles, aprimorei minhas habilidades de programação. A todos os meus colegas do Departamento de Matemática da PUC, pessoas que entendem que sozinho não se chega a lugar algum, e que todos nós somos na verdade, uma grande equipe.

A todos os meus professores: Hélio Lopes, Marcelo Dreux (Engenharia Mecânica), Geovan Tavares, Carlos Tomei e Marcos Craizer, que sem exceção, foram maravilhosos e muito pacientes comigo.

À CAPES, pelos auxílios concedidos. Aos funcionários do departamento de Matemática da PUC. Ao professor Paulo Rogério Sabini da UERJ, bem como sua esposa Aruquia Barbosa, por terem me incentivado a cursar o mestrado, e pelo apoio dado durante todos esses anos. Um agradecimento muito especial ao meu orientador Sinésio Pesco, pela paciência na orientação desta dissertação, e pelo grande conhecimento transmitido durante esses 2 anos.

Resumo

Lisowski, Karin; Pesco, Sinesio. **Método da Oclusão Implícita e suas aplicações em visualização**. Rio de Janeiro, 2007. 79p. Dissertação de Mestrado — Departamento de Matemática, Pontifícia Universidade Católica do Rio de Janeiro.

Neste trabalho aplicamos o método de oclusão implícita para acelerar o tempo de cálculo e renderização de isosuperfícies em dados volumétricos regulares. Dado um campo escalar contínuo f sobre um domínio D e um isovalor w, a oclusão implícita explora a continuidade de f para determinar os limites de visibilidades sem a necessidade de calcular a isosuperfície explicitamente. Aplicamos esta técnica para obter também as silhuetas visíveis das isosuperfícies.

Palavras–chave

Extração de Isosuperfícies. Descarte por oclusão. Visualização Volumétrica.

Abstract

Lisowski, Karin; Pesco, Sinesio. **Implicit Occluder Method and visualization applications**. Rio de Janeiro, 2007. 79p. MsC Thesis — Departament of Mathematics, Pontifícia Universidade Católica do Rio de Janeiro.

In this work we apply the Implicit Occluders method for optimizing the computation and rendering of isosurfaces in regular volumetric data. Given a continuous scalar field f over a domain D and an isovalue w, Implicit Occluders exploits the continuity of f to determine visibility bounds without the need for computing the isosurface explicitly. We apply this technique to obtain also the visible silhouettes of isosurfaces.

Keywords

Isosurface extraction. Occlusion Culling. Volume visualization.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

1.1
Motivação

Existem diversas técnicas de cálculo e renderização de isosuperfícies. Entre elas, podemos destacar o algoritmo de Marching Cubes (10), que utiliza amostras de uma função escalar f em um grid volumétrico e usa uma tabela de combinações possíveis entre as mudanças de sinal dos vértices do grid, criando assim, triângulos que aproximam a isosuperfície.

No entanto, o algoritmo de Marching Cubes é oneroso computacionalmente, principalmente para dados volumétricos grandes. Por isso, muitos trabalhos buscam detectar regiões que não são de interesse no volume, com o objetivo de evitar aplicar o algoritmo Marching Cubes em todo o dado. Uma das estratégias é aplicar testes de visibilidade para determinar quais regiões são visíveis e devem ser exploradas.

A determinação da visibilidade tem sido um problema fundamental na computação gráfica. Vários algoritmos de remoção de áreas escondidas têm sido desenvolvidos com o objetivo de determinar porções visíveis da imagem. O principal objetivo deste trabalho é apresentar uma técnica de visibilidade capaz de reconstruir uma isosuperfície de um dado volumétrico em menos tempo, se comparado aos métodos tradicionais, buscando para isso gerar apenas as porções visíveis da isosuperfície.

1.2
Objetivo

Nesta dissertação apresentaremos uma técnica que otimiza a computação e renderização de isosuperfícies em um dado volumétrico. Dado um campo escalar contínuo f sobre um domínio D (onde D é convexo) e um isovalor w, nossa técnica explora a continuidade de f para determinar os limites de visibilidade sem a necessidade de computar a isosuperfície $f^{-1}(w)$.

Decompondo o domínio hierarquicamente (via uma octree, por exemplo), classificamos cada nó como positivo, negativo ou zero conforme o seu campo escalar seja maior, menor ou contenha o isovalor w respectivamente. A partir

daí, para cada posição do observador, geramos a região de oclusão da primeira troca de sinal entre os nós. A idéia-chave é que as eventuais isosuperfícies atrás desta região não são visíveis (ver figura 1.1). Em seguida, verificamos a visibilidade de cada nó zero frente ao mapa gerado e as classificamos como visíveis ou não. Aplicamos então o algoritmo de Marching Cubes apenas nas células visíveis, evitando assim, aplicar o algoritmo em todo o dado volumétrico.

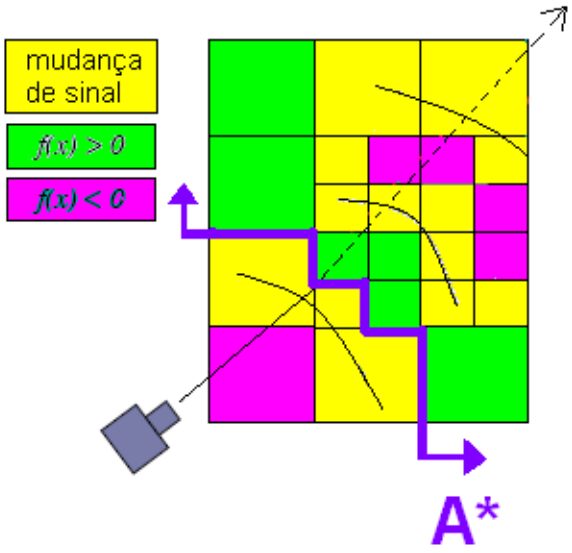


Figura 1.1: O método da oclusão implícita estuda a primeira mudança de sinal de positivo para negativo ou vice versa.

Neste trabalho estamos considerando dados volumétricos como um conjunto regular de amostras de campos escalares que podem ser interpretados como uma amostragem da função implícita F.

Uma das contribuições é a aplicação do método da oclusão implícita no cálculo da silhueta exterior de uma figura. DeCarlo et al. (1) propôs uma técnica de renderização da silhueta. DeCarlo menciona que desenhar apenas a silhueta externa é muito mais dispendioso do que calcular todas as linhas da silhueta. No entanto, em nosso método, utilizando a oclusão implícita podemos ocultar boa parte das linhas não-visíveis, dependendo da qualidade do ocluder gerado.

1.3 Trabalhos anteriores

O método proposto neste trabalho é uma continuidade dos trabalhos de Pesco et al. (12) e Wilhelms e Van Gelder (14), que proporam técnicas que buscam orimizar a renderização de isosuperfícies em dados volumétricos.

Este último propõe uma octree e guarda as informações dos valores máximo e mínimo de cada nó da octree, e verifica em quais nós estariam ocorrendo as mudanças de sinal de f numa vizinhança da isosuperfície. Já no método de oclusão implícita, criamos a região de oclusão realizando um estudo da primeira mudança de sinal dos nós da octree. A geração do mapa de oclusão está relacionada com a posição do observador, sendo portanto denominada uma técnica dependente do observador, como veremos no capítulo 3.

1.4 Organização da dissertação

Esta dissertação foi organizada da seguinte maneira: o capítulo 2 apresenta os conceitos preliminares sobre visibilidade. O capítulo 3 mostra alguns trabalhos anteriores realizados na área de visualização e descreve o método da oclusão implícita. O capítulo 4 dá detalhes sobre a implementação: estrutura de dados, montagem da octree, obtenção da região de oclusão, teste de visibilidade e renderização do objeto. O capítulo 5 apresenta os resultados e as comparações. O capítulo 6 descreve o algoritmo de Marching Lines para o cálculo da silhueta do objeto. Finalmente, o capítulo 7 conclui e apresenta propostas para trabalhos futuros.

2
Preliminares

2.1
Técnicas básicas de visibilidade

A determinação da visibilidade tem sido um problema fundamental em computação gráfica. Entre as técnicas de visibilidade mais conhecidas estão: back-face culling, view-frustum culling e occlusion culling. Faremos um breve comentário sobre cada uma dessas técnicas.

2.1.1
Back-face culling

Consiste em remover as faces ocultas de um objeto dada a posição do observador, evitando assim, o processamento desnecessário de faces que não são visíveis, como nos mostra a figura 2.1.

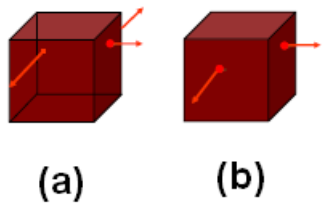


Figura 2.1: (a) imagem sem back-face culling (b) imagem utilizando o back face culling

A implementação do back-face culling consiste no cálculo do produto interno entre a normal da face em questão e a direção do observador. Seja N o vetor normal à face e V o vetor de visão (ver figura 2.2). Então, se:

- V.N = 0, os vetores são perpendiculares, e a face não é visível.
- V.N > 0, o ângulo entre os vetores é menor que 90 ° e a face está virada para trás e não é visível.
- V.N < 0, o ângulo entre os vetores é maior que 90 ° e a face está virada para frente.

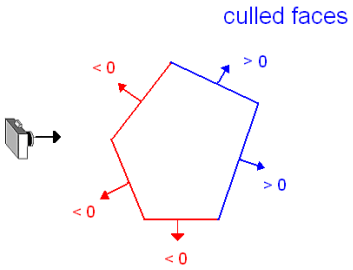


Figura 2.2: A face não é visível se o produto interno entre a normal da face e o vetor de visão for menor que zero.

A biblioteca do OpenGL possui uma opção para habilitar o back-face culling através do glEnable(GL_CULL_FACE).

2.1.2 View-frustum culling

Chamamos de viewing frustum o volume limitado entre a posição do observador e os vértices da janela em questão. Qualquer geometria que se encontrar fora deste volume estará fora do campo de visão do observador, logo, não será renderizada. Ver figura 2.3.

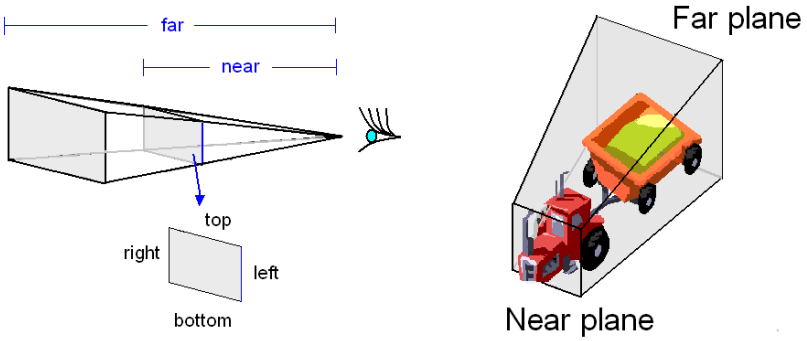


Figura 2.3: View frustum culling: qualquer parte do objeto fora do volume de visualização não é visível.

2.1.3 Occlusion culling

Outra técnica que evita cálculos desnecessários é o occlusion culling, cujo objetivo é não renderizar a geometria escondida por outro objeto. Na figura 2.4, o poliedro oculta parcialmente a visualização do cubo.

A figura 2.5 ilustra o back-face, view-frustum e occlusion culling.

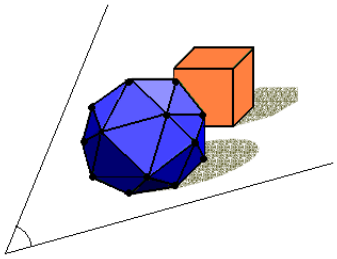


Figura 2.4: Occlusion culling: remove a geometria escondida por outro objeto.

2.2 Critérios para algoritmos de visibilidade

Nesta seção discutimos alguns critérios utilizados na classificação das principais técnicas de visibilidade.

2.2.1 Ponto versus Região

- *from point versus from region* – O algoritmo *from point* calcula a visibilidade considerando o observador posicionado em um único ponto, enquanto que o *from region* calcula a visibilidade que seja válida em qualquer ponto de uma certa região.

2.2.2 Espaço objeto versus Espaço imagem

- *object space versus image space* – Os algoritmos *object space* utilizam objetos em si para o cálculo da visibilidade. Por outro lado, os algoritmos *image space* operam na representação discreta de objetos quando fragmentados durante o processo de rasterização. Em geral, os algoritmos de imagem (*image space*) são mais eficientes e funcionam normalmente por aproximação, já os algoritmos de objeto (*object space*) possuem solução exata pois verificam a parte do objeto que está sendo encoberta por outro objeto qualquer.

2.2.3 Conservativo versus Aproximado

- *consevative versus aproximate* – o algoritmo *conservativo* é aquele que superestima o conjunto visível. Partes do objeto que não são visíveis são classificados como visíveis. O algoritmo *aproximado* calcula a visibilidade aproximada e não garante encontrar todas as primitivas visíveis possíveis.

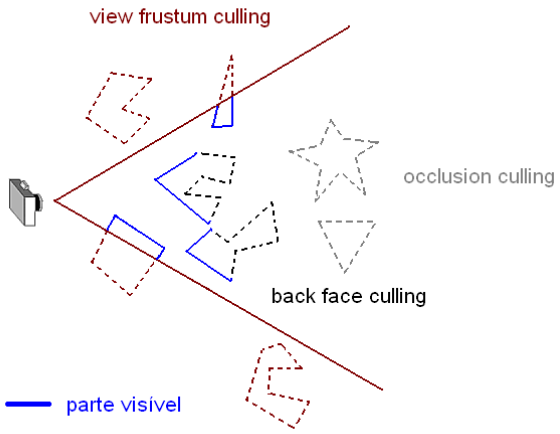


Figura 2.5: 3 técnicas de remoção de áreas escondidas: back face culling, view frustum culling e occlusion culling.

2.3 Recursos do OpenGL

O OpenGL (GL Graphics Library) é um sistema gráfico desenvolvido pela Silicon Graphics voltada para aplicações gráficas 3D.

Entre os vários recursos do OpenGL utilizados na implementação deste trabalho, destacamos dois que são fundamentais na geração do ocluder implícito: depth buffer e stencil buffer.

Depth buffer

O depth buffer é responsável pelo armazenamento da profundidade de cada pixel. Este valor varia de 0 a 1, onde o zero indica que o objeto está próximo do observador e 1 indica que o objeto está no infinito. É usado para o occlusion culling. O depth buffer também é conhecido como z-buffer.

A função glDepthFunc(GLenum func) especifica a função usada para comparar o valor z de cada pixel com o valor z presente no depth buffer. O default é GL_LESS, que significa que um fragmento passa pelo teste quando seu valor em z for menor do que o valor já armazenado no depth buffer. Outros parâmetros são GL_NEVER, GL_ALWAYS, GL_LEQUAL, GL_EQUAL, GL_NOTEQUAL e GL_GREATER. Tal comparação só é feita se o teste de profundidade estiver ativado pelo glEnable(GL_DEPTH_TEST).

Stencil buffer

Utilizamos o stencil buffer para restringir uma determinada porção da tela para que objetos não sejam desenhados, assim como um pintor utiliza um estêncil de papelão com spray para pintar uma superfície. Tal mascaramento

de regiões é realizado marcando os pixels que devem ou não ser desenhados por 1 ou 0.

O stencil buffer é ativado pela função glEnable(GL_STENCIL_TEST), e o teste do stencil é feito através da função glStencilFunc() que possui 3 parâmetros: (GLenum func, GLint ref, GLuint mask). O primeiro parâmetro pode ser: GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, ou GL_NOTEQUAL. O teste glStencilFunc() realiza a comparação (func) do valor de referência (ref & mask) com o valor no stencil buffer (stencil & mask). Ver comentários na seção 4.5.2.

A função glStencilOp() especifica como o conteúdo da máscara do stencil buffer será modificado quando um fragmento passar ou falhar no teste do stencil.Tem como entrada de dados: (GLenum fail, GLenum zfail, GLenum zpass). O primeiro parâmetro indica o que acontece quando o teste do stencil falha. O segundo é se o teste do stencil não falhar, mas o z-buffer falhar, e por fim, o terceiro parâmetro indica se os 2 testes (do stencil e do z-buffer) não falharem. Os 3 parâmetros podem ser: GL_KEEP, GL_ZERO, GL_REPLACE GL_INCR, GL_DECR, ou GL_INVERT, onde:

- GL_KEEP - mantém o atual conteúdo do stencil para esse fragmento.
- GL_ZERO - zera o conteúdo para esse fragmento.
- GL_REPLACE - Troca o conteúdo do stencil para o valor indicado no campo ref do glStencilFunc.
- GL_DECR - decrementa o conteúdo.

3
Método da oclusão implícita

Dados volumétricos de grandes dimensões estão presentes em vários campos como medicina, sísmica, etc. A geração e visualização de isosuperfícies para esses dados pode ser impraticável em algumas situações, pois a isosuperfície gerada pode conter milhões de polígonos sobrecarregando o processamento gráfico. Além disso, a busca de todas as células do dado que contém a isosuperfície e a consequente geração da triangulação resultam em um alto consumo de tempo.

Uma proposta para viabilizar esse processo procura explorar o fato de que, para alguns dados volumétricos grandes, porções da isosuperfície desejada são internas, estando portanto, ocultas pelas porções visíveis da isosuperfície. Assim, se gerarmos apenas porções visíveis da isosuperfície, reduziremos o número de faces geradas e visualizadas resultando em um processo mais rápido.

Este é o princípio básico de uma classe de algoritmos denomidado view-dependent (dependência do observador) (8), no qual baseia-se o método da oclusão implícita que mencionaremos a seguir.

Inicialmente descreveremos alguns trabalhos anteriores relacionados.

3.1
Trabalhos anteriores

Livnat (8) propõe uma clasificação das principais estratégias utilizadas para acelerar a extração de isosuperfícies em três categorias: decomposição geométrica do espaço, decomposição do espaço valor e decomposição do espaço imagem.

3.1.1
Decomposição geométrica do espaço

Especificamente para grids volumétricos estruturados obtemos uma organização espacial dos dados, imposta pela própria estrutura ortogonal do grid. Sendo assim, os métodos baseados na geometria do espaço procuram explorar a coerência entre células adjacentes.

Um primeiro exemplo é o algoritmo Marching Cubes (10), criado em 1987 com o objetivo de renderizar imagens médicas em tomografias computadorizadas e ressonâncias magnéticas. A idéia básica do algoritmo é aproximar uma isosuperfície em um dado volumétrico através de triângulos. O algoritmo calcula os vértices do triângulo utilizando interpolação linear dos vértices de cada cubo do dado volumétrico, daí o nome "Marching Cubes", pois o algoritmo "marcha"em todos os cubos do volume de dados. O Marching Cubes não se preocupa em localizar de uma forma eficiente porções do grid que não contenham a isosuperfícies. Visto que os dados volumétricos são na maioria dos casos de grandes dimensões, muito se têm pesquisado na direção de algoritmos que melhorem a eficiência do Marching Cubes.

Nessa direção, Wilhelms e Van Gelder (14) realizaram um trabalho que utiliza uma octree para otimizar a renderização de um dado volumétrico. Inicialmente o dado volumétrico regular é ajustado a octree de forma que cada nó da octree delimita um subconjunto do grid. Em seguida, dado um isovalor (ou isolevel) w, classifica-se cada nó da octree como: positivo, negativo ou zero, segundo o critério: se o valor da função em todo o subconjunto do grid contido nesse nó for maior do que w, o nó é "positivo". Caso o valor da função em todos os vértices for menor que w, o nó é classificado como "negativo", e se houver mudança de sinal, dizemos que o nó é "zero"(ver figura 3.1). O objetivo de classificar os nós da octree, é aplicar o algoritmo Marching Cubes apenas nos nós "zero"da octree, ou seja, nós que contém a isosuperfície, evitando assim, percorrer regiões do grid que não contenham a isosuperfície. Este algoritmo não é view-dependent, pois a quantidade de nós "zero"da octree independe da posição do observador.

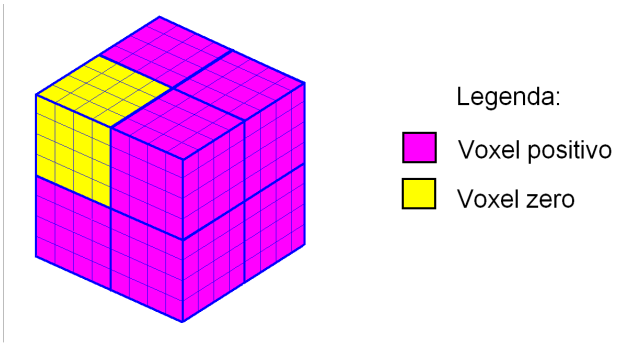


Figura 3.1: Octree de nível 1 onde há 1 nó "zero"e 7 nós "positivos", Logo, aplica-se o algoritmo de Marching Cubes em apenas 1/8 de todo o volume de dados.

3.1.2
Decomposição do espaço valor

A decomposição do espaço valor considera a decomposição dos valores do campo escalar f definido sobre o grid, em geral, através de intervalos cujos extremos correspondem respectivamente ao mínimo e máximo que f atinge em um dado nó.

Assim, através de estratégias que organizem e ordenem os intervalos é possível acessar eficientemente os nós que contenham regiões da isosuperfície. Usando esta técnica, Livnat et al. (8) definem a noção do span space e utilizam uma kd-tree para organizar os intervalos. Cignoni et al. (4) utilizam o span-space para reduzir a complexidade na fase de busca dos nós. Gao e Shen (5) aplicam o conceito de span space para desenvolver um processamento em paralelo.

Para a extração de isosuperfícies em dados de dimensões arbitrárias, Chiang et. al. (3) e (2) propõe a utilização de memória externa (out-of-core).

3.1.3
Decomposição do espaço imagem

Nesta classificação inclui-se os métodos que atuam principalmente na geração da imagem final, buscando evitar a extração da isosuperfície em regiões que não são visíveis.

Parker et al. (11) propõe um ray tracing em tempo real cujo princípio é gerar imagens da isosuperfície sem uma representação explícita dos polígonos da isosuperfície.

Em (8) Livnat et al. introduziram uma classe de algoritmos deno- minados view-dependent (dependência do observador), baseado em um trabalho anterior de Greene (6). O objetivo é reduzir os tempos de busca, geração e renderização selecionando, para isso, somente células que contenham porções visíveis da isosuperfície para uma dada posição do observador. O princípio básico do trabalho proposto por Livnat et al. (9) é construir regiões de oclusão (via espaço imagem) por intermédio da própria isosuperfície extraída incrementalmente. Para isso utiliza uma estrutura hierárquica com percorrimento de frente para trás.

3.1.4
Apresentação do trabalho

Nessa dissertação prosseguimos com o trabalho proposto em Pesco et al. (12) que, seguindo a classificação proposta, corresponde a um algoritmo view-dependent. A principal diferença entre Pesco at al. (12) e Livnat (7) é que a

construção da região de oclusão é feita sem calcular a isosuperfície. A geração da região de oclusão é feita apenas com a informação do campo escalar.

3.2
Método da oclusão implícita

Considere um campo escalar contínuo $f : D \rightarrow \Re$ definido sobre um domínio convexo $D \subset \Re^3$ e um isovalor w. O objetivo é determinar ocluders para a isosuperfície $f^{-1}(w)$, ou seja, determinar regiões que delimitam porções visíveis e não visíveis da isosuperfície.

Explorando a continuidade de f podemos gerar ocluders sem a necessidade de calcular a isosuperfície $f^{-1}(w)$. Para isso, estudamos as trocas de sinal de $f^*(x) = f(x) - w$ de positivo para negativo (ou vice versa) numa vizinhança da isosuperfície. Sem perda de generalidade, vamos considerar $w = 0$. Caso $w \neq 0$, basta considerar a função f^*.

3.2.1
Descrição do método

Considere $A \subset D$ uma região onde f é sempre positivo e $B \subset D$ outra região em que f é sempre negativo conforme a figura 3.2.

Desde que D seja convexo, qualquer reta r conectando um ponto em A a um ponto em B está totalmente contida em D. Então, se o valor de f sobre r muda continuamente de positivo para negativo, segundo o teorema do valor intermediário, existe um zero de f entre os 2 valores, que é exatamente a interseção da reta r com a isosuperfície.

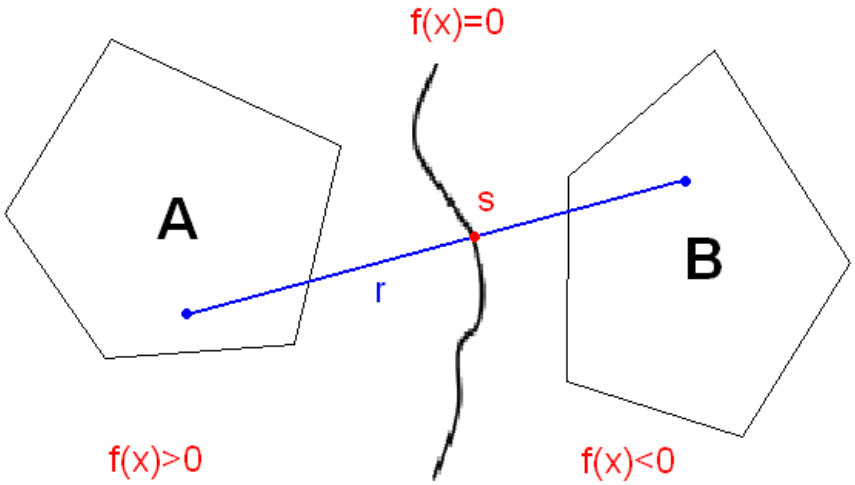


Figura 3.2: A reta r conectando o conjunto A ao conjunto B intercepta a isosuperfície $f^{-1}(0)$.

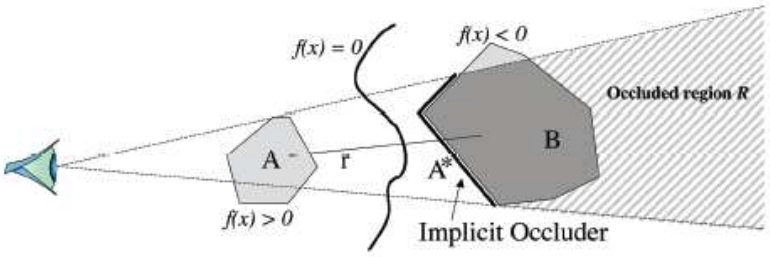


Figura 3.3: A idéia fundamental na oclusão implícita é explorar a continuidade do campo escalar f para definir uma região de oclusão, sem a necessidade de computar a isosuperfície.

Em seguida, considerando uma direção particular do observador projetamos a região A sobre o bordo da região B (ou vice versa) obtendo A^* conforme indicado na figura 3.3.

Assim qualquer raio partindo do obsevador e alcançando A^* deve obrigatoriamente ter interseção com a isosuperfície. Podemos, portanto, concluir que a região atrás de A^* é não visível e eleger A^* como um ocluder.

3.2.2 Observações sobre o método

Algumas observações sobre o método:

- A determinação de A^* depende de uma posição particular do observador. Isso justifica a classificação desse método como view-dependent.
- A determinação de A^* é feita através do estudo das trocas de sinal de f, não sendo necessário calcular a isosuperfície. Por esta razão a denominação oclusão "implícita".
- A região entre a isosuperfíce $f^{-1}(0)$ e A^* é considerada visível pelo método, embora não seja pois está atrás da isosuperfície. Dessa forma o método define um conjunto visível *conservativo* (ver figura 3.4), *from point*, pois calcula a visibilidade a partir da posição da câmera e *image space*, pois verificamos a visibilidade (discreta) de porções do objeto.
- Para ser efetiva, qualquer implementação do método precisa detectar a primeira mudança de sinal na direção do raio para um melhor aproveitamento do ocluder (veja figura 3.5)

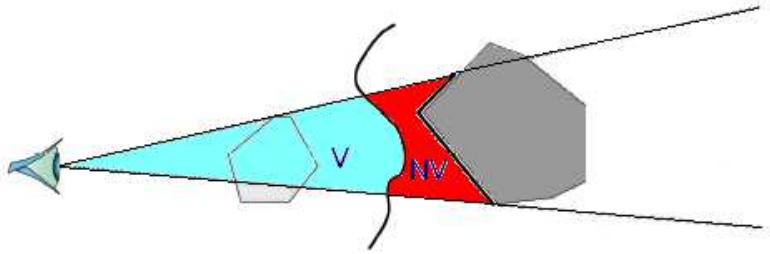


Figura 3.4: O conjunto conservativo de visibilidade é a união dos dois conjuntos indicados V e NV, pois inclui toda a porção visível e possivelmente uma porção não visível.

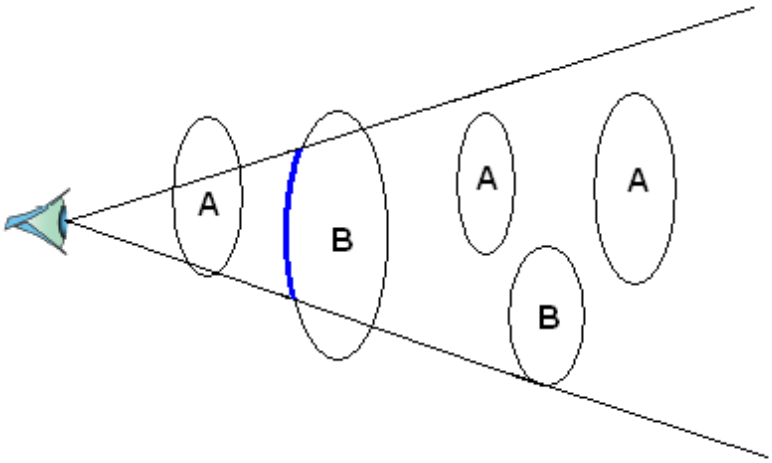


Figura 3.5: A idéia do occluder é detectar a primeira mudança de sinal.

4
Implementação da oclusão implícita

A implementação da oclusão implícita (12) segue, em vários aspectos, a estrutura proposta em Wilhelms e Van Gelder (14) e Livnat e Hansen (8).

O algoritmo está separado em 4 etapas básicas:

1. Leitura do dado volumétrico e construção da octree.
2. Geração dos ocluders.
3. Teste de visibilidade para detectar os nós visíveis da octree.
4. Cálculo e visualização da isosuperfície.

Leitura do dado volumétrico e construção da octree. Primeiramente, lemos as informações do dado volumétrico, montamos uma octree como fase de pré-processamento, ajustamos cada nó da octree de modo a coincidir com a subdivisão do grid do volume de dados. Guardamos numa estrutura de dados as informações dos valores máximo e mínimo de cada nó, e com estas informações podemos classificar cada nó da octree como "positivo", "negativo"ou "zero".

Geração dos ocluders. Os nós da octree que estão acima e abaixo da isosuperfície (positivos e negativos) serão os possíveis occluders. A região de oclusão é construída no espaço imagem como resultado da renderização dos nós positivos e negativos seguindo uma estratégia que será apresentada nas seções seguintes. O objetivo principal dessa estratégia de renderização dos nós é encontrar a primeira mudança de sinal e para isso utilizamos o depth buffer e stencil buffer do OpenGL.

Teste de visibilidade. Montada a região de oclusão, realizamos um teste de visibilidade por intermédio dos testes de oclusão presentes em recentes placas gráficas (hardware occlusion culling query). Isto é feito da seguinte forma:

– Enviamos o bounding box do nó para que seja verificada sua visibilidade contra a região de oclusão previamente gerada.

– Se algum pixel for visível, então nó é classificado como visível.

A eficiência e rapidez na classificação da visibilidade de um nó é fundamental no algoritmo proposto. Um nó marcado como não visível não será visitado e consequentemente todos os seus filhos também, resultando em uma potencial economia de tempo, o que justificaria o tempo investido na geração do occluder e no próprio teste de visibilidade. Por outro lado, os nós visíveis terão esse mesmo tempo acrescido do tempo de cálculo e renderização da isosuperfície.

Rendering. Por fim, calculamos e renderizamos a isosuperfície apenas nos nós "zero"da octree que foram marcados como visíveis, utilizando o Marching Cubes (10).

Nas próximas seções apresentaremos alguns detalhes de implementação de cada uma dessas etapas.

4.1 Leitura do dado volumétrico e construção da octree

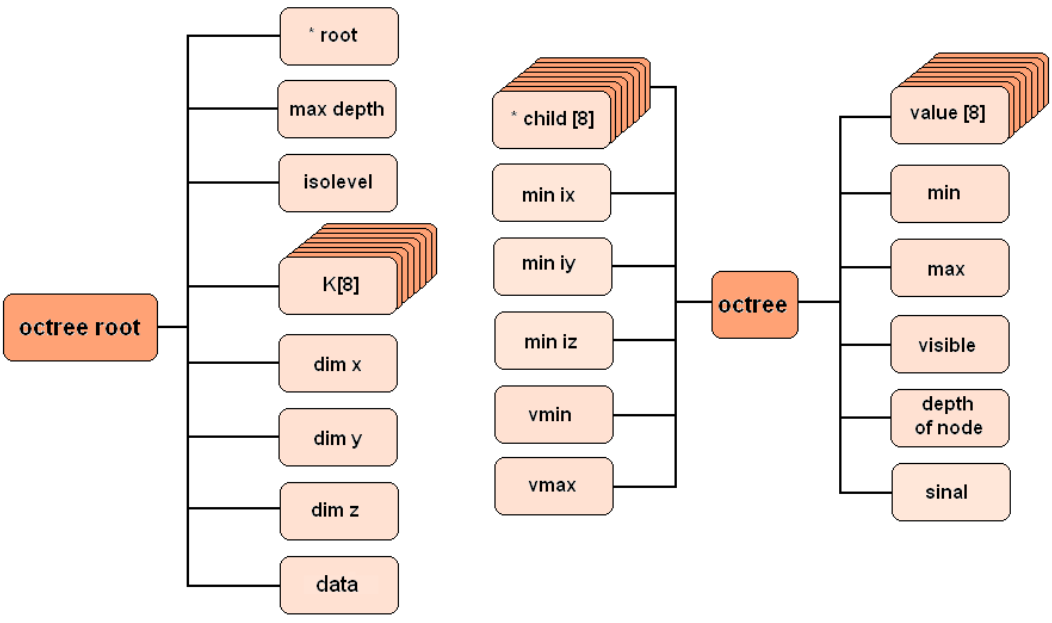


Figura 4.1: Organização das classes Octree e OctreeRoot.

4.1.1 Estrutura de dados

Para implementação da oclusão implícita utilizamos uma octree, seguindo a mesma estrutura proposta em Wilhelm e Van Gelder (14). A octree foi implementada através das classes OctreeRoot e Octree.

Estrutura de dados da OctreeRoot

- (Octree *) root // ponteiro para a raiz da octree.
- (unsigned) max_depth // profundidade máxima.
- (double) isolevel // valor do isolevel.
- (int) K[8] // ordem de renderização dos filhos de cada voxel.
- (unsigned) _dimx, _dimy, _dimz //dimensões do grid volumétrico.
- (Grid*) data // dado volumétrico.

Estrutura de dados da Octree

- (double) _value[8] // valor da função nos vértices de cada nó.
- (octree *) _child[8] // ponteiro para os filhos.
- (double) _vmin, _vmax // valor mínimo e máximo de cada nó.
- (unsigned) _sinal // (0) se o nó for negativo, (1) para positivo e (2) para mudança de sinal.
- (ponto3D) _min, _max // coordenadas do voxel.
- (bool) _visible // indica se um nó é visível ou não.
- (unsigned) _depth_of_node // profundidade em que se encontra o nó.
- (int) _min_ix, _min_iy, _min_iz, _max_ix, _max_iy, _max_iz // índices associados ao grid volumétrico correspondente a cada nó.

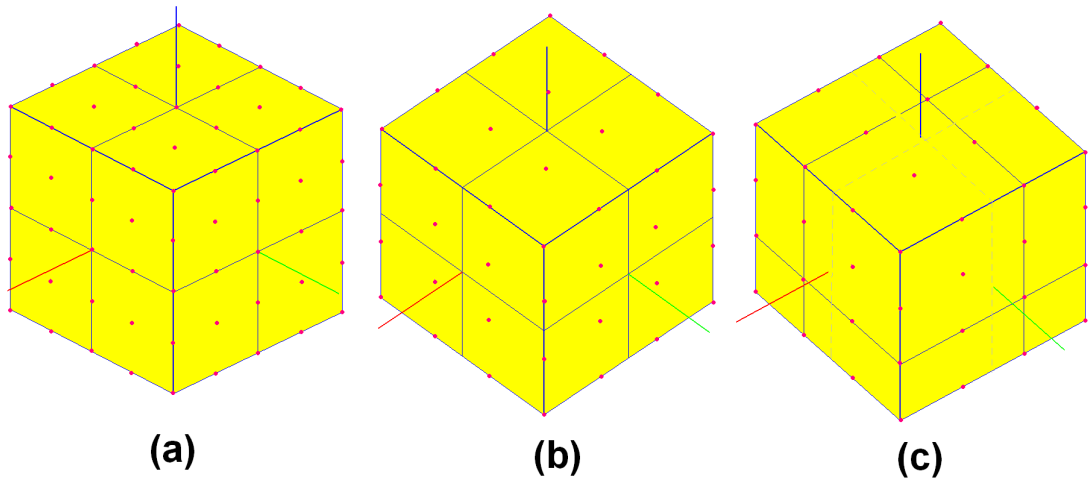


Figura 4.2: (a) vértice do voxel coincidente com a subdivisão do grid. (b) vértice do voxel não coincidente com a subdivisão do grid. (c) correção do nó da octree de modo a coincidir com o grid.

4.1.2
Ajuste do grid à octree

Em nosso trabalho elaboramos uma classe chamada GRID que guarda as informações do dado volumétrico:

- dimx, dimy e dimz // dimensões do dado volumétrico.
- data[i][j][k] // valor escalar associado.

Quando ajustamos o grid do dado volumétrico na octree, pode ocorrer do nó não coincidir com o grid em alguma das direções. Isto pode ocorrer quando o grid tem dimensão par. Observe a figura 4.2 o exemplo de quando a subdivisão do nó coincide ou não com a subdivisão do grid.

Neste caso, ajustamos os nós ao grid de forma a compensar essa diferença somente no último nó da octree, conforme nos mostra o ítem (c) da figura 4.2.

4.1.3
Valores máximo e mínimo de um nó

Depois de criarmos a octree, precisamos calcular os valores máximo e mínimo de cada nó. Para isso, utilizaremos a rotina SetMinMax( ), que compara os valores da função escalar em cada vértice do cubo e retorna o maior e o menor valor (_vmin e _vmax) desde o último nível da octree. Os valores máximo e mínimo do nó pai são os valores máximo e mínimo considerando-se os 8 nós filhos.

Pseudo código da rotina SetMinMax()

```
double maxchild[8], minchild[8];
int indice, indice2;
- Se o nó não é folha:
   - Para (i=0; i<8; i++)
      SetMinMax() do nó_child[i];
      minchild[i] = mínimo de nó_child[i]
      maxchild[i] = máximo de nó_child[i]
   - Fim do para
   - Para (indice=1, índice<8, indice++)
      -Se (maxchild[indice] > maxchild[maiorIndice])
         maiorIndice = indice;
      -Fim do se
   -Fim do para
   -vmax = maxchild[maiorIndice]
   - Para (indice2=1, indice2<8,indice2++)
      -Se (minchild[indice2] < minchild[menorIndice])
         menorIndice = indice2;
      -Fim do se
   -Fim do para
   - vmin = minchild[menorIndice]
- Senão:
   - Calcula mínimo do nó
   - Calcula máximo do nó
-Fim do se
```

Tabela 4.1: Pseudo codigo da rotina SetMinMax

4.1.4 Obtenção do sinal do nó

Encontrados os valores _vmin e _vmax de cada nó, utilizamos a rotina SetSinal(isolevel) para calcularmos o sinal do nó. Definimos os sinais de um nó como: 0– positivo, 1 – negativo, 2 – onde há mudança de sinal. Observe que a determinação do sinal do nó deve ser recalculada sempre que houver alteração do isolevel. Ver figura 4.3.

4.2 Geração do mapa de oclusão

É importante destacar que até este ponto, foi descrito na seção 4.1 a etapa de pré-processamento, que inclui a leitura do dado, geração da octree e o cálculo do máximo/mínimo. Nesta seção e nas seções 4.3 e 4.4 estaremos descrevendo as rotinas que são processadas cada vez que um evento de rendering da cena é chamado.

Pseudo código da rotina SetSinal(isolevel)
- Se _vmax < isolevel O sinal do nó é negativo (0) ; - Senão - Se _vmin > isolevel O sinal do nó é positivo (1); - Senão Há mudança de sinal (2);

Tabela 4.2: Pseudo codigo da rotina SetSinal

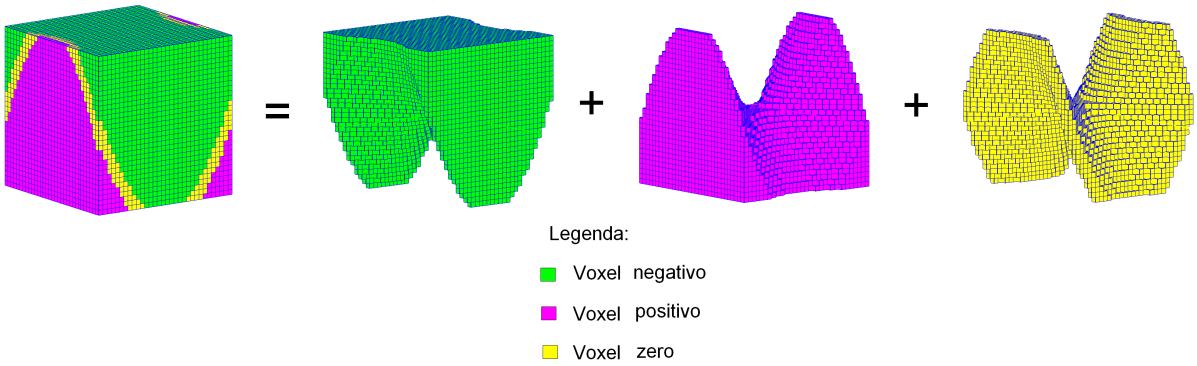


Figura 4.3: Octree: composta de nós positivos, negativos, e nós zero (onde há mudança de sinal).

Conforme mencionado anteriormente, para obtermos a região de oclusão não será necessário calcular a isosuperfície. Neste momento, nosso principal objetivo é construir a região de oclusão através da projeção dos nós positivos sobre os negativos (ou vice-versa), preservando a profundidade na primeira troca de sinal. Para isso utilizaremos as bibliotecas do OpenGL. São necessários 2 buffers: o depth buffer e o stencil buffer (apenas 1 bit).

A geração do ocluder implícito é feita em 2 etapas. Na primeira etapa, o bounding box de todos os nós negativos são renderizados seguindo a ordem de frente para trás para corrigir a profundidade da oclusão. O stencil buffer é utilizado para determinar quais pixels foram cobertos na primeira passada e para garantir que na segunda passada, a profundidade seja corrigida apenas uma vez para cada pixel. Vejamos os detalhes da implementação ilustrando através do exemplo da figura 4.4. O algoritmo pode ser separado em 7 passos principais:

1. Inicialização do Depth e Stencil buffers.

2. Configuração dos testes do Stencil e Depth buffers.

3. Renderizar os nós positivos.
4. Reconfigurar os testes do Stencil e Depth buffers.
5. Ordenação frente-para-trás dos nós da octree.
6. Renderizar os nós negativos.
7. Correção da profundidade para os pixels onde não houve troca de sinal.

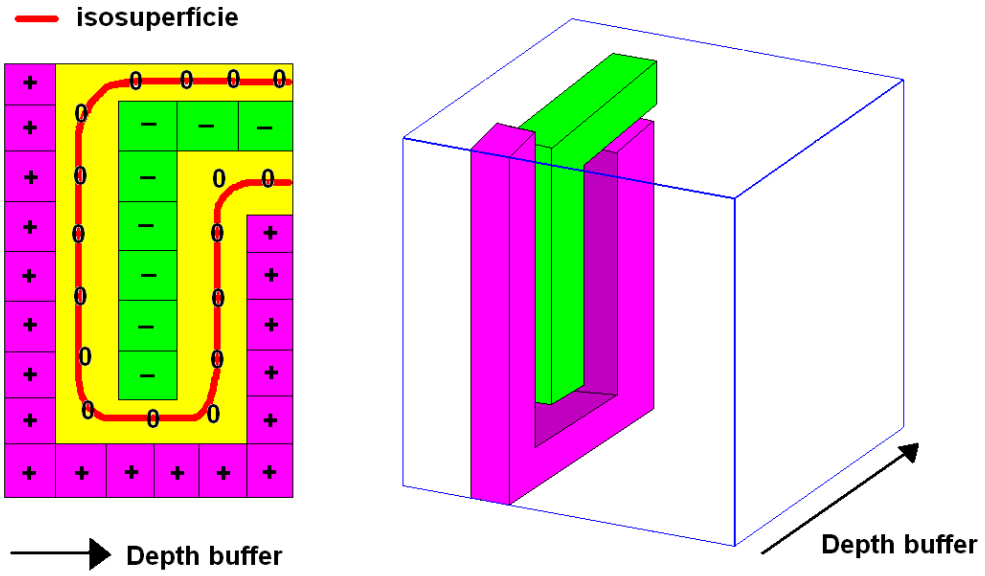


Figura 4.4: Disposição dos nós no depth buffer.

4.2.1 Inicialização do Depth buffer e Stencil buffer

Em nosso algoritmo o primeiro passo é zerar o stencil buffer e inicializar o depth buffer no infinito (ver figura 4.5).

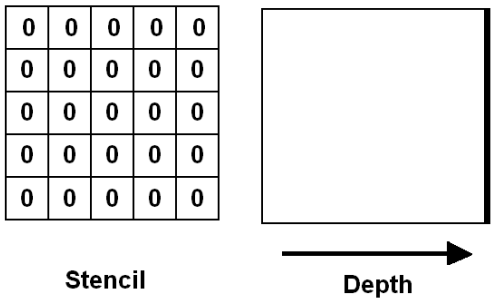


Figura 4.5: Inicializamos o stencil buffer com 0 e o depth buffer no infinito, ou seja, com valor 1.

Para isso, utilizamos as seguintes funções:

glClearDepth(1); // inicializa o depth com 1
glClearStencil(0); //inicializa o stencil com 0
glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT); //inicializa o depth buffer e stencil buffer

4.2.2 Configuração dos testes do Stencil e Depth buffers

No caso do stencil, na primeira etapa do algoritmo, marcaremos 1 para todo pixel resultante da renderização dos nós positivos da octree. Para marcar no stencil a região a ser desenhada na primeira parte do algoritmo, utilizamos a função glStencilFunc(). Definimos então as funções que irão colocar (ou não) dados na área do stencil, dependendo do resultado dos testes. Iniciamos com:

Referencia = 1;
glStencilFunc (GL_ALWAYS, Referencia, 1);
glStencilOp (GL_KEEP, GL_REPLACE, GL_REPLACE);

Em outras palavras: o teste do stencil sempre passa (GL_ALWAYS) e portanto a operação definida no 2º e 3º parâmetros do glStencilOp serão executados conforme o teste do depth buffer falhe ou passe. Como ambas são GL_REPLACE, então o resultado é que para cada fragmento gerado na renderização dos nós positivos, corresponderá o valor 1 (o conteúdo do stencil será trocado pelo valor indicado no campo Referência, que é 1) no stencil.

4.2.3 Renderizar os nós positivos

Para reduzir o tempo de geração do ocluder, na renderização dos nós positivos (ou negativos) da octree, estamos renderizando apenas os 3 lados visíveis do "bounding box"do nó conforme a posição do observador.

Outra economia na renderização é o fato de que não estamos renderizando todos os nós folha positivos, pois se um voxel pai é positivo, seus filhos também o serão.

Observe na tabela 4.3 o algoritmo de renderização do occluder positivo.

4.2.4 Reconfigurar os testes do Stencil e Depth buffers

Inicialmente alteramos o teste do depth buffer para glDepthFunc (GL_GREATER), pois desejamos guardar a maior profundidade referente aos

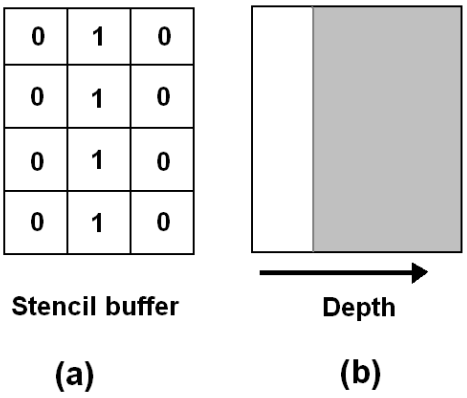


Figura 4.6: Inicializamos o stencil buffer com 0 e o depth buffer no infinito, ou seja, com valor 1.

Pseudo código da rotina Ocluder_Positivo(octree father)

```
- Se (father) é positivo:
     renderiza os 3 lados visíveis do voxel
- Senão
    -Se (father) não é nó folha e há mudança de sinal:
        -Para i variando de 1 a 8
           Occluder_Positivo(father− >child[i])
```

Tabela 4.3: Pseudo codigo da rotina OcluderPositivo

nós negativos.

O teste do depth buffer irá funcionar se a nova profundidade for maior do que a armazenada anteriormente, caso contrário, o conteúdo do depth buffer para esse fragmento fica como está.

Agora mudamos nossa função teste do stencil para:

```
glStencilFunc(GL_NOTEQUAL, 0, 1);
glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE);
```

Assim, se o stencil correspondente a um dado pixel não é igual (GL_NOTEQUAL) ao valor de referência (zero), ou seja, é 1, dizemos que o teste do stencil passou. O teste do stencil estará localizando as regiões que foram anteriormente renderizadas pelos nós positivos. Neste caso, estaremos substituindo o valor (GL_REPLACE) do stencil para 0. Na prática, estamos estabelecendo que na próxima etapa, durante a renderização dos nós negativos, somente as regiões anteriormente marcadas pelos nós positivos serão cobertas pelos nós negativos, porém aqui há um ponto importante dessa implementação: ao trocarmos o valor do stencil para zero em um dado fragmento, estamos

garantindo que o teste do stencil irá falhar para qualquer outro fragmento na mesma posição. Com a ordenação de frente para trás é garantido que apenas a primeira troca de sinal será registrada. Porém para que a profundidade do depth buffer fique correta, precisamos ordenar a octree de frente-para-trás de forma que o nó negativo mais próximo do observador seja renderizado primeiro. Este procedimento será descrito a seguir.

4.2.5 Ordenação frente-para-trás dos nós da octree

A ordenação de frente-para-trás da octree é um algoritmo básico que depende unicamente da posição do observador. Faremos uma rápida descrição do algoritmo utilizando o OpenGL. A posição do observador pode ser obtida diretamente da matriz ModelView do OpenGL:

$$\begin{pmatrix} V[0] & V[1] & V[2] = obs.x & 0 \\ V[4] & V[5] & V[6] = obs.y & 0 \\ V[8] & V[9] & V[10] = obs.z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
double V[16];
glGetDoublev(GL_MODELVIEW_MATRIX,V);
obs.x=V[2];
obs.y=V[6];
obs.z=V[10];
```

Através da posição do observador, podemos obter uma ordenação para os nós da octree (variável K[8] da estrutura de dados mencionada na seção 4.2). É necessário sabermos primeiramente em que octante o observador se encontra. Para isto, basta compararmos o sinal das coordenadas x, y e z do observador. Por exemplo: se as coordenadas x, y e z do observador forem positivas, o observador se encontra no octante 0 e a ordem de renderização dos nós será: 3,7,1,5,2,6,0,4 conforme indica a figura 4.7.

Dado o octante do em que se encontra o observador, ordenamos os nós da octree através de uma matriz $a[i][j]$ onde a linha i representa o octante, e a coluna j representa a ordem $k[i]$ desejada:

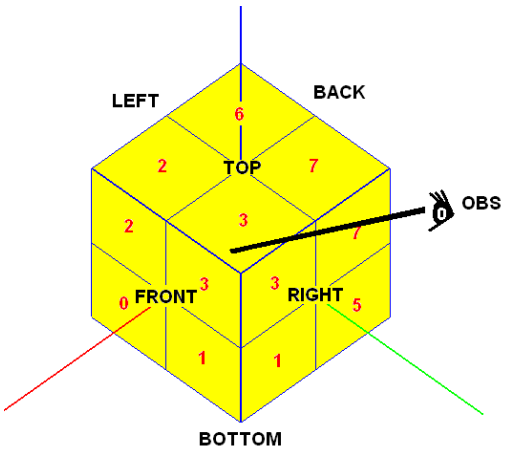


Figura 4.7: Ordenação dos nós octree.

Octante	Ordenação K[i]
0	3 7 1 5 2 6 0 4
1	7 3 5 1 6 2 4 0
2	2 6 0 4 3 7 1 5
3	6 2 4 0 7 3 5 1
4	1 5 3 7 0 4 2 6
5	5 1 7 3 4 0 6 2
6	0 4 2 6 1 5 3 7
7	4 0 6 2 5 1 7 3

Esta ordenação foi obtida observando os nós que estão à frente do observador e logo em seguida, os que estão atrás (front-to-back). Consideramos "frente"aquele voxel que esconde uma face de um outro voxel. A ordem de renderização é importante para a montagem da região de oclusão, pois isso garantirá que a informação da profundidade seja armazenada corretamente no z-buffer.

4.2.6 Renderizar os nós negativos

Nesta etapa renderizamos os nós negativos seguindo a ordem estabelecida na etapa anterior. As figuras 4.8 (a) e 4.8 (b) ilustram qual seria o resultado para o exemplo proposto.

Observe que é indiferente a ordem em que aparecem os nós positivos e negativos para o resultado final, como está ilustrado na figura 4.9.

No caso (I), o passo 6 não altera o depth, pois o depth-test foi selecionado para GL_GREATER, logo ficamos com a profundidade do maior (do nó positivo).

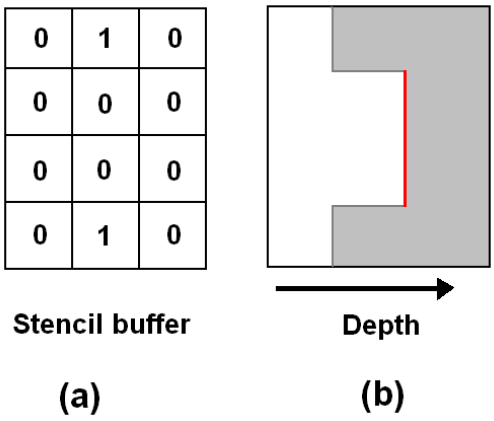


Figura 4.8: Renderiza negativo

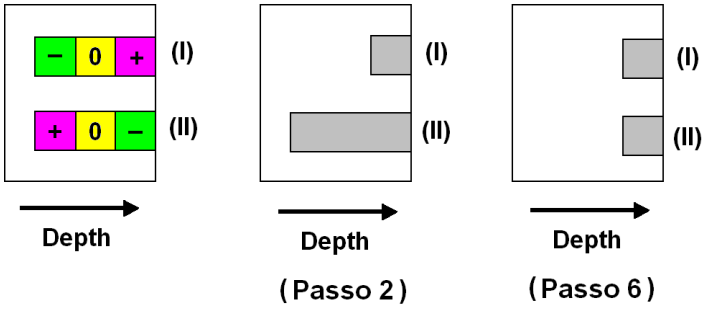


Figura 4.9: Passo 2: Após a renderização dos nós positivos. Passo 6: Após a renderização dos nós negativos.

No caso (II), o passo 6 altera o depth pois o nó negativo tem maior profundidade.

4.2.7
Correção da profundidade para os pixels onde não houve troca de sinal

Uma última correção é necessária nos fragmentos onde não houve troca de sinal e que não deve haver oclusão. Observe que no exemplo da figura 4.6, os nós positivos mais acima estariam cobrindo uma parte visível da isosuperfície. Para resolver este problema, basta renderizar as 3 faces de trás de um cubo que contenha todo o grid volumétrico (ver figura 4.10).

Isto corrige apenas a profundidade dos fragmentos ainda não alterados (marcados com 1 no stencil). Veja as figuras 4.11 (a) e 4.11 (b).

O occluder implícito buscado seria a região destacada na figura 4.11 (b).

4.2.8
Algoritmo em C++

Em resumo, o algoritmo em C++ é dado na tabela 4.4:

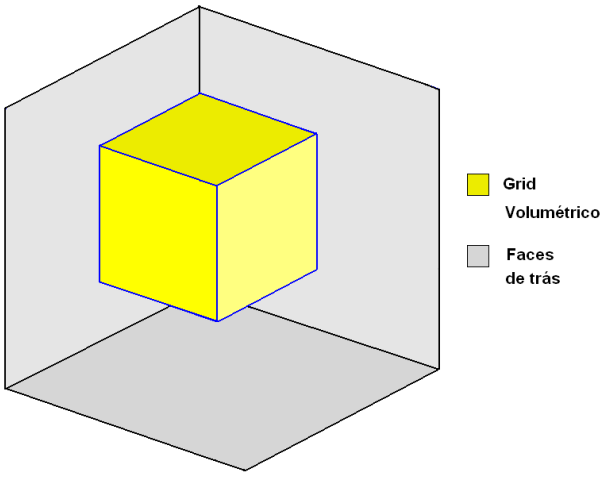


Figura 4.10: Correção da profundidade dos pixels através da renderização de 3 faces de um cubo posicionado atrás do grid volumétrico.

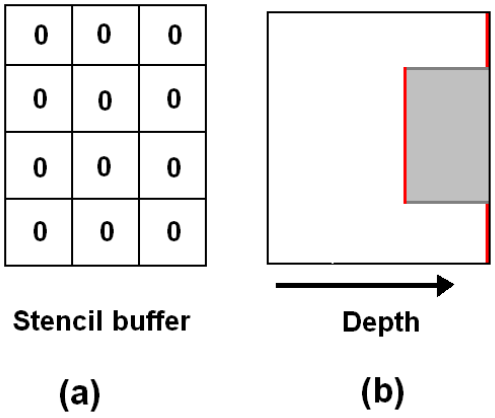


Figura 4.11: Mapa de oclusão implícito.

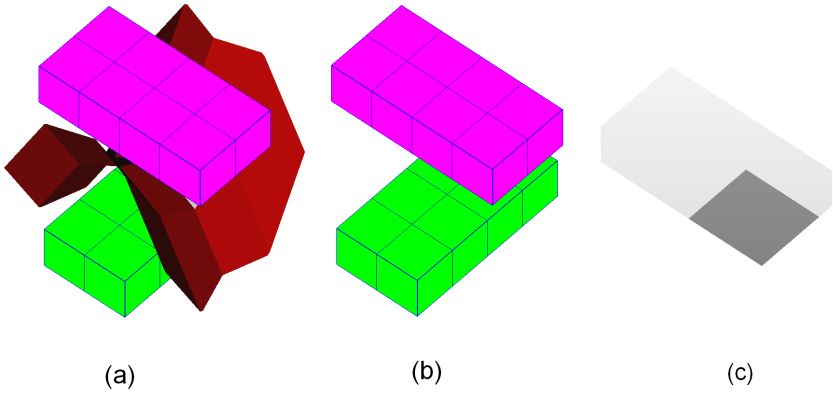


Figura 4.12: Parabolóide hiperbólico numa octree de nível 2: (a) Nós positivos, negativos e isosuperfície (b) Apenas os nós positivos e negativos (c) região de oclusão.

Podemos visualizar a região de oclusão através da função display depth buffer, que colore os pixels com tons de cinza de acordo com a profundidade

Algoritmo do stencil e depth buffers:

```
glEnable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
glClearDepth(1);
glClearStencil(0);
glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE);
glDepthFunc(GL_LESS);
Occluder_Positivo();
glDepthFunc(GL_GREATER);
glStencilFunc(GL_NOTEQUAL, 0, 1);
Occluder_Negativo();
Renderiza_Voxel_Back();
glDisable(GL_STENCIL_TEST);
glDepthFunc(GL_LESS);
```

Tabela 4.4: Algoritmo do stencil e depth buffers.

marcada no depth buffer.

Note os exemplos das figuras 4.13 e 4.14 de regiões de oclusão desenhadas através do display depth buffer: (Veja mais exemplos no capítulo 5)

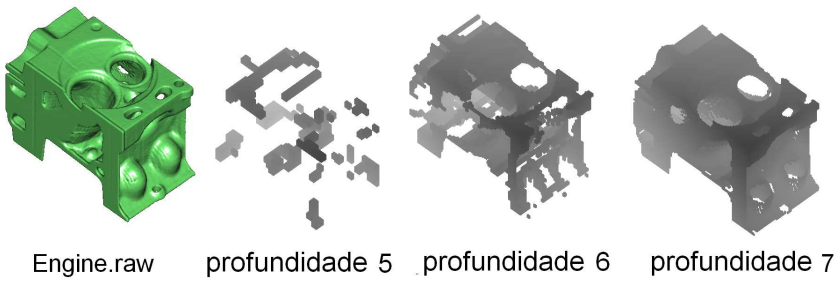


Figura 4.13: Regiões de oclusão do dado volumétrico Engine selecionando-se octrees com profundidades 5, 6 e 7.

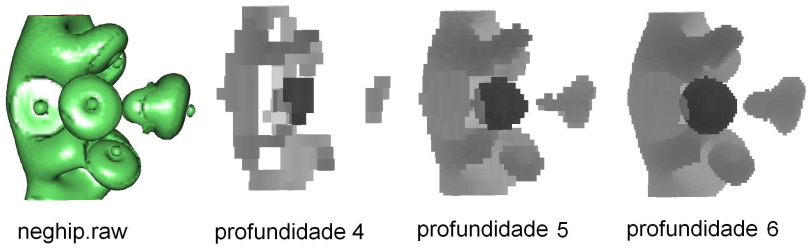


Figura 4.14: Regiões de oclusão do dado volumétrico Neghip selecionando-se octrees com profundidades 5, 6 e 7.

4.3 Teste de visibilidade do nó da octree

Utilizaremos recursos da placa gráfica para verificar se um nó da octree é visível ou não contra o mapa de oclusão. Occlusion queries podem ser usados para obter o número exato de fragmentos que passam pelo teste de profundidade (depth test).

Basicamente, este teste consiste nos seguintes passos:

1. Criar a região de oclusão.
2. Desenhar o bounding Box dos objetos "candidatos"à oclusão.
3. Finalizar o teste de oclusão.
4. Verificar o número de pixels do bounding Box que passaram no teste de profundidade.

Antes de utilizarmos os recursos da placa, desabilitaremos as máscaras de profundidade e de cor:

glDepthMask(GL_FALSE); //desabilita a escrita no buffer de profundidade

glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE); //desabilita a escrita no buffer de cor

Occlusion queries são utilizados para evitar renderizar objetos completamente oclusos. Isto pode resultar numa grande redução de geometria na cena.

Utilizamos a extensão NVIDIA NV_OCCLUSION_QUERY que consiste nas seguintes funções:

- *glBeginOcclusionQueryNV(uint id)* – Inicia o occlusion query. Tal comando zera o contador do pixel e o resultado do teste de oclusão. O occlusion test é inicializado como FALSE. Logo após utilizarmos o comando glBeginOcclusionQueryNV, podemos renderizar o bounding Box do objeto (nó da octree).
- *glEndOcclusionQueryNV(void)* – Com este comando finalizamos o occlusion query.
- *glGenOcclusionQueriesNV(sizei n, uint *ids)* – Gera o occlusion queries, ou seja, a lista de objetos a serem determinados como oclusos ou não. *n* é o tamanho da lista a ser gerada e *ids* é o nome do occlusion query. Estes nomes são marcados como visitados, mas nenhum objeto é associado a

eles até que o BeginOcclusionQueryNV seja chamado. Cada occlusion query contém um contador de pixels, e tal contador é inicializado com zero quando o objeto é criado.

– *glGetOcclusionQueryuivNV(uint id, enum pname, uint *params)* – Informa o número de fragmentos que passam no teste de profundidade.

– *glDeleteOcclusionQueriesNV(sizei n, const uint *ids)* – Deleta o occlusion queries.

Observe que um nó será testado apenas se houver mudança de sinal. Os nós positivos (ou negativos) são obrigatoriamente não visíveis.

Partindo do nó raiz, iniciamos o processo de renderização do bounding box dos nós filhos da raiz.

Se não houverem pixels na contagem de um nó filho, este não é visível. Se houver ao menos 1 pixel na contagem, o nó é dito visível. E como o occlusion test foi realizado nos 8 filhos e chamamos recursivamente a função para os filhos deste nó, podemos dizer que se nenhum pixel foi encontrado nos 8 filhos, concluímos que o nó pai também não é visível.

Segue o pseudo código das 2 funções que verificam a visibilidade do nó da octree.

4.3.1
Pseudo código do mark occluder:

Pseudo código do mark_occluder(octree * root, int profundidade)

GLuint occlusionQueries[8];
glGenOcclusionQueriesNV(8, occlusionQueries);
glDisable(GL_STENCIL_TEST);
glDepthMask(GL_FALSE);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
mark_occluder_node(node,depth,occlusionQueries);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
glDeleteOcclusionQueriesNV(8, occlusionQueries);

Tabela 4.5: Pseudo codigo da rotina Mark_Occluder

4.3.2
Pseudo código do mark occluder node

Em alguns casos, é caro verificar a visibilidade até o nó folha da octree. Podemos reduzir o tempo nesta etapa verificando a visibilidade de um nó da octree até um determinado nível de profundidade estipulado pelo usuário. Observe no capítulo 6 os resultados obtidos com diferentes profundidades para este teste de visibilidade.

4.4
Cálculo e render da isosuperfície

Em nosso trabalho, utilizamos o algoritmo de Marching Cubes (10) para gerar a triangulação da isosuperfície. As normais em cada vértice são obtidas por interpolação do gradiente do dado volumétrico.

Conforme mencionamos anteriormente, criamos uma octree onde ajustamos cada voxel ao grid do dado volumétrico, e por fim, aplicamos Marching Cubes apenas na porção do grid associado ao nó da octree onde há mudança de sinal e foi marcado como visível pelo teste da seção anterior (ver figura 4.15).

A figura 4.16 ilustra o resultado do teste de visibilidade utilizando a oclusão implícita gerada com uma octree de profundidade 6.

No dado volumétrico Engine, a isosuperfície no isolevel 120 possui 643.580 faces. Aplicando-se a oclusão implícita o número de faces renderizadas decai conforme a qualidade da região de oclusão, conforme mostra a tabela seguinte.

Profundidade 5	**Profundidade 6**	**Profundidade 7**
642.294	468.549	256.726

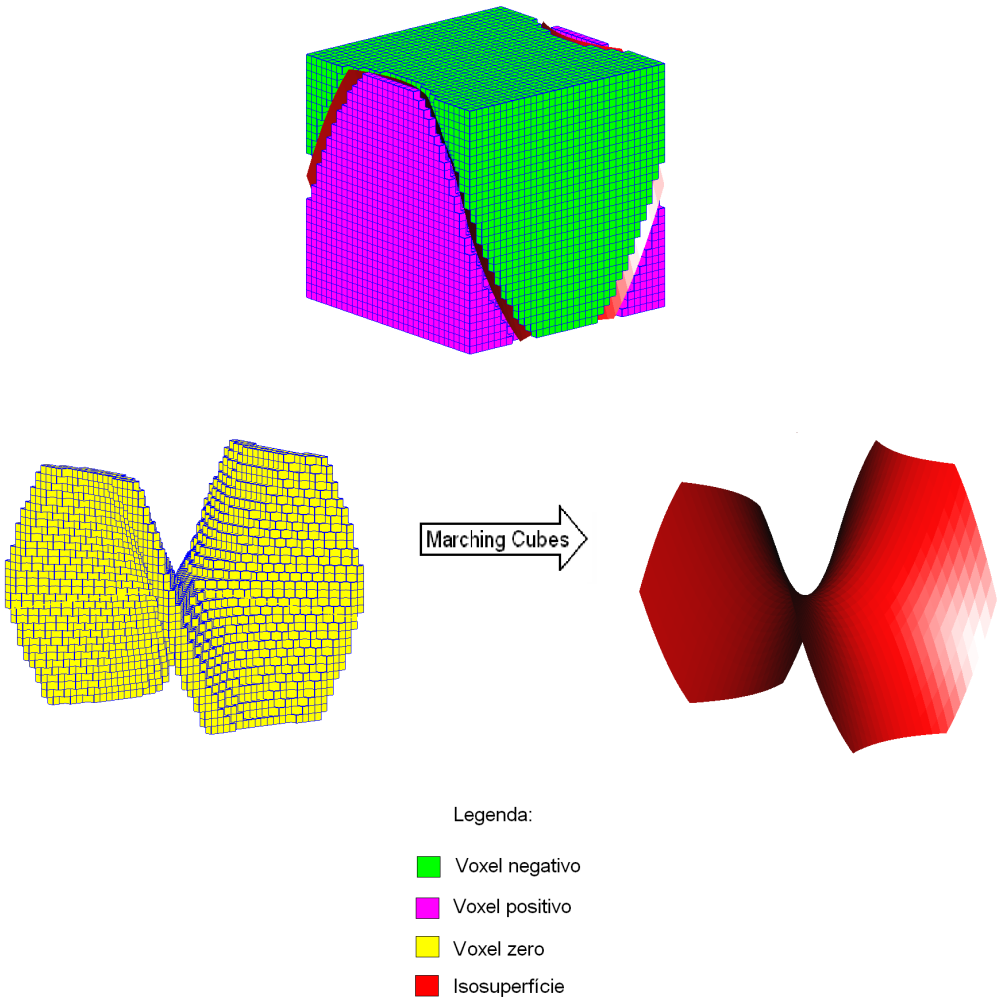


Figura 4.15: Aplicação do algoritmo de Marching Cubes apenas nos nós onde há mudança de sinal e que são visíveis.

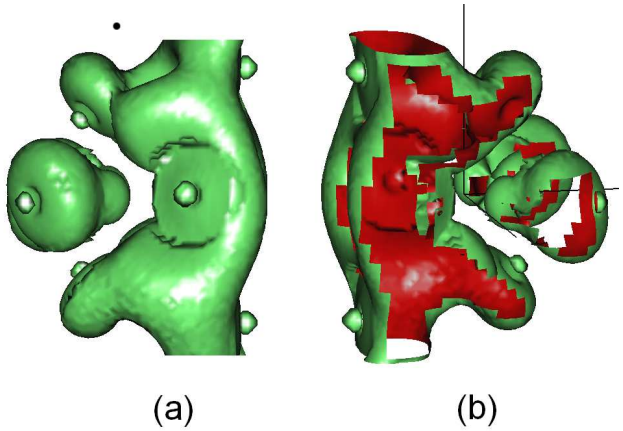


Figura 4.16: Dado volumétrico Neghip.raw: (a) isosuperfície renderizada com occluder (b) Teste de visibilidade interrompido para mostrar quantas faces deixaram de ser renderizadas aplicando-se a oclusão implícita.

Pseudo código do mark_occluder_node (octree *node, unsigned depth, GLuint *occlusionQueries)

```
- Se o nó não é folha
   - Se há mudança de sinal
      node− >visible = true;

      -Se (prof > prof_do_nó)
         GLuint pixel_count[8]

       -Para i variando de 1 a 8
          glBeginOcclusionQueryNV(occlusionQueries[i]);
          renderiza as 6 faces do voxel child na ordem K[i]
          glEndOcclusionQueryNV();
       -Fim do para
       bool any_visible = false;
       -Para i variando de 1 a 8
          glGetOcclusionQueryuivNV(occlusionQueries[i], GL_PIXEL_COUNT_NV,
                        pixel_Count[i]);
       -Fim do para

       -Para i variando de 1 a 8
         -Se pixel_count[i]>0
            Mark_occluder_node(child[k[i]],depth,occlusionQueries);
            any_visible = true;
         -Senão (pixel_count = 0)
             Node− >child[i]− >visible = false
       -Fim do para

       -Se (any_visible = false)
         node− >visible = false; // se nenhum pixel foi encontrado nos filhos,
                     //o pai não é visível
       -Fim do se

     -Senão //(prof = prof_do_nó)
        -Para i variando de 1 a 8
           mark_occluder_node(child[i], depth, occlusionQueries);
        -Fim do para

   -Senão //(sinal é + ou -)
      node− >visible = false

-Senão //(o nó é folha)
   -Se há mudança de sinal
      node− >visible = true
   -Senão
     node− >visible = false
```

Tabela 4.6: Pseudo codigo da rotina Mark_Occluder_Node

5
Resultados da extração de isosuperfícies

Neste capítulo apresentaremos os resultados do método de oclusão implícita aplicada na extração de isosuperfícies.

Um dos objetivos principais do método de oclusão implícita é reduzir o número de faces geradas e renderizadas de uma isosuperfície, buscando também reduzir o tempo total na visualização. Assim, nos resultados obtidos apresentaremos esses dados e discutiremos o custo/benefício em cada caso. Em particular, a oclusão implícita envolve três custos que devem ser avaliados:

1. Construção da região de oclusão.
2. Teste de visibilidade.
3. Geração e renderização.

Construção da região de oclusão - Na geração do ocluder, devemos selecionar qual a profundidade da octree que será utilizada. Quanto menor a profundidade selecionada, mais rápido a região de oclusão é construída, com economia de tempo. Por outro lado, menor será a área coberta, com consequente redução na qualidade do ocluder. Para isosuperfícies muito refinadas, a utilização de ocluders de maior profundidade pode resultar em uma boa performance, como veremos nos exemplos. Comparamos os tempos para octrees de profundidades 5, 6 e 7.

Teste de visibilidade - Assim como na geração do ocluder, podemos também selecionar o nível máximo de profundidade para o teste de visibilidade (capítulo 4, seção 4.3), o que pode significar também uma economia de tempo. Isto ocorre porque muitas vezes o teste de visibilidade até a máxima profundidade pode ser mais caro do que calcular e renderizar um número adicional de faces invisíveis. Nos exemplos a seguir comparamos tempos para profundidades do nível 5 até a profundidade máxima.

Cálculo e renderização - Para cada nó classificado como visível, é feita a geração e rendering da isosuperfície deste nó via Marching Cubes. A princípio as faces geradas não são armazenadas na memória para futuros

acessos, reduzindo o consumo de memória. Também, alterando-se o isovalor não há necessidade de remontarmos a octree, já que os valores máximo e mínimo são os mesmos, bastando apenas verificar o sinal dos nós da octree.

Comparamos os tempos para alguns dados volumétricos com ocluder e sem ocluder. O caso sem ocluder corresponde ao algoritmo proposto po Wilhelms e Van Gelder (14).

Explicitamos o tempo de cada etapa do algoritmo: construção da região de oclusão, teste de visibilidade do nó e tempo de renderização via Marching Cubes.

Para cada exemplo exibimos a região de oclusão para diferentes profundidades. Como resultado final exibimos a isosuperfície obtida, bem como a comparação entre uma silhueta renderizada com o algoritmo de Marching Lines com/sem ocluder.

Implementamos nosso trabalho em C++, e como biblioteca gráfica, utilizamos o OpenGL. Nossos resultados foram obtidos em um equipamento com processador Pentium D, 3.0 Ghz, 3,25 GB de RAM e placa de vídeo NVIDIA GFORCE FX 7800.

Neste trabalho estamos utilizando exemplos de dados volumétricos que na maior parte foram obtidos no site www.volvis.org.

5.1 Stent

O dado volumétrico stent (174 x 512 x 512) foi extraído de uma tomografia computadorizada do abdomem e pélvis. Inicialmente escolhemos a isosuperfície de nível $w = 600$ com originalmente 2.624.244 faces.

A tabela a seguir exibe os resultados obtidos para o exemplo da figura 5.1 com uma octree de profundidade 7.

Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,187	
Teste de visibilidade:	0,140	
Marching Cubes:	2,734	9,421
Total:	3,062	9,421
Número de faces renderizadas:	780.408	2.624.244
Redução do tempo total:	67%	
Redução do número de faces:	71%	

Utilizando o método de oclusão implícita para este exemplo, renderizamos apenas 29 % do total de triângulos, de modo que deixamos de calcular 1.843.836 faces para a posição de câmera mencionada na tabela.

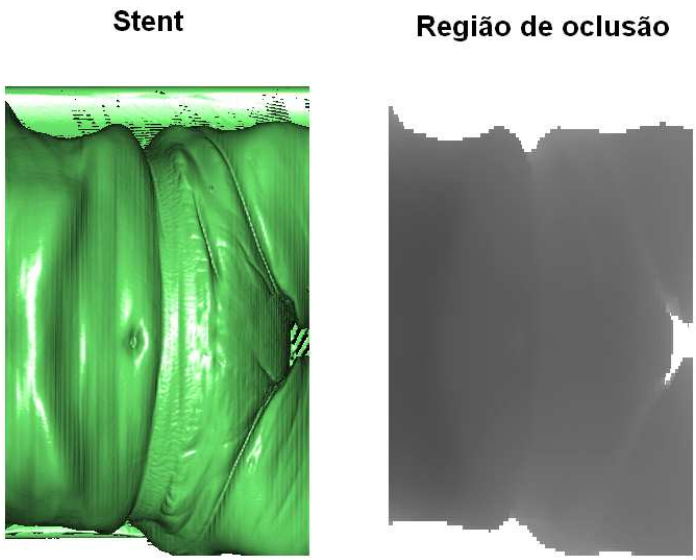


Figura 5.1: Isosuperfície com isovalor 600 e sua respectiva região de oclusão.

A figura 5.2 (a) compara os tempos gastos com/sem ocluder para diferentes posições de câmera do stent. A figura 5.2 (b) exibe o número de triângulos gerados em cada caso.

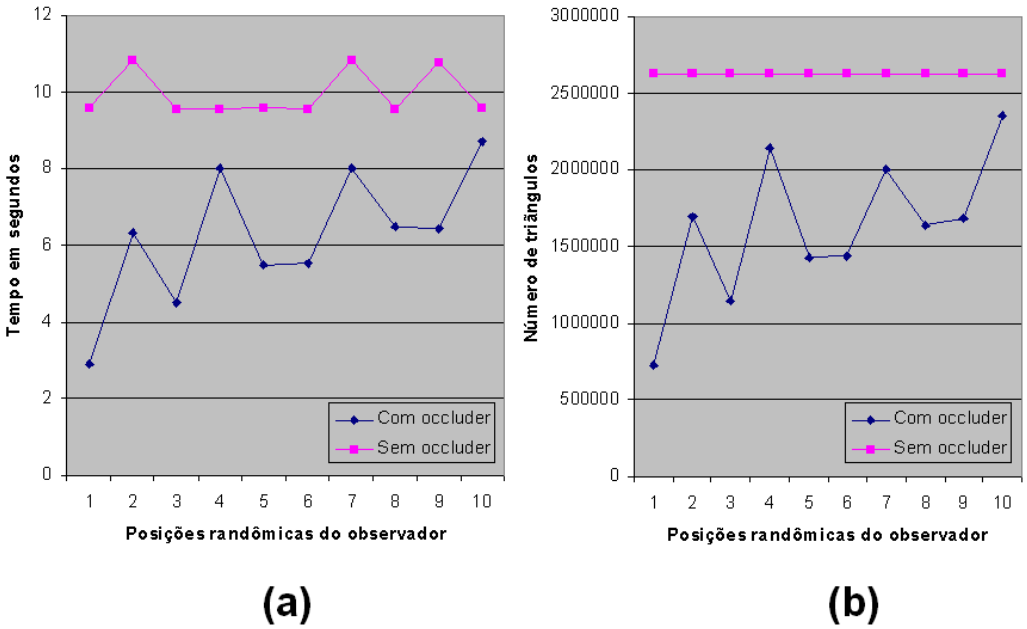


Figura 5.2: Gráficos do dado volumétrico Stent de profundidade 7 e isolevel 600. (a) Comparação dos tempos totais com/sem occluder. (b) Comparação do número de triângulos com/sem occluder.

A figura 5.3 apresenta a composição do tempo usando a oclusão implícita nas três partes: geração do ocluder, teste de visibilidade e geração/rendering.

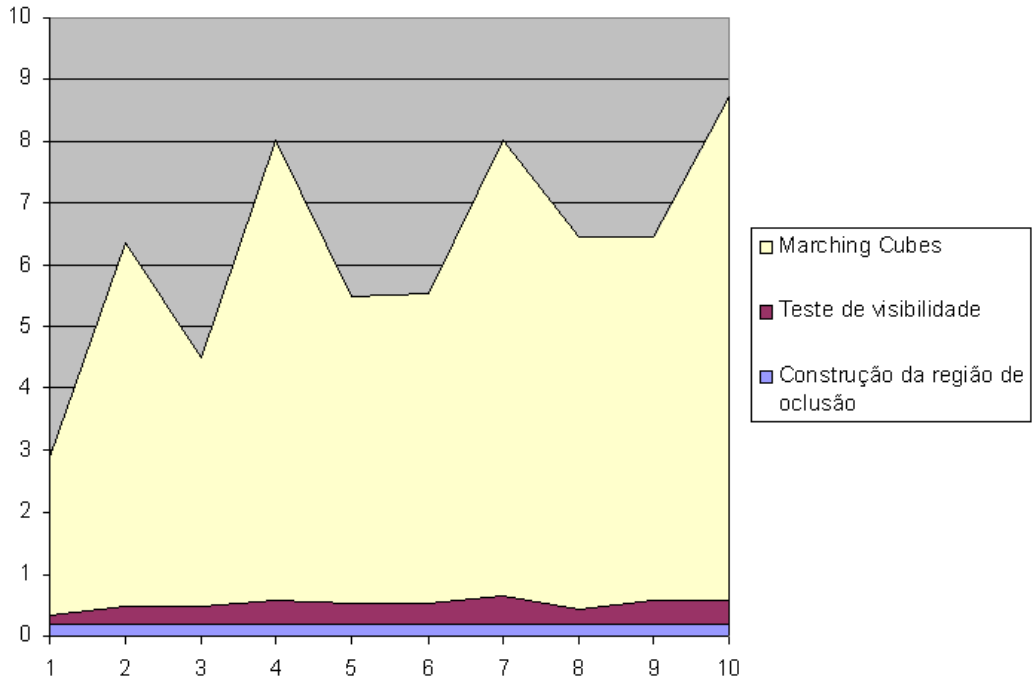


Figura 5.3: Gráfico do dado volumétrico Stent de profundidade 7 e isolevel 600. Comparações dos tempos de criação da região de oclusão, teste de visibilidade e tempo de renderização.

A tabela a seguir exibe os resultados obtidos para mapas de oclusão gerados com profundidade 7 e 6 e testes de visibilidade de 5 até a profundidade máxima.

Para o exemplo do stent, obtivemos o melhor tempo de renderização numa octree de profundidade 7 com verificação da visibilidade até o último nível da octree, que foi de 3,062 segundos, conforme nos mostra a tabela 5.1.

Mapa de oclusão		**Teste de visibilidade**	
Profundidade 7	**Nível 7**	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,187	0,187	0,187
Teste de visibilidade:	0,140	0,078	0,032
Marching Cubes:	2,734	3,187	4,186
Total:	**3,062**	3,453	4,406
Triângulos:	780.408	912.041	1.198.859
Profundidade 6	**Nível 6**	**Nível 5**	
Construção da reg. de ocl.:	0,046	0,046	
Teste de visibilidade:	0,047	0,016	
Marching Cubes:	3,562	4,531	
Total:	3,655	4,592	
Triângulos:	953.495	1.222.965	

Tabela 5.1: Comparação dos tempos para diferentes níveis de visibilidade do Stent

A figura 5.4 exibe os mapas de oclusão para diferentes profundidades.

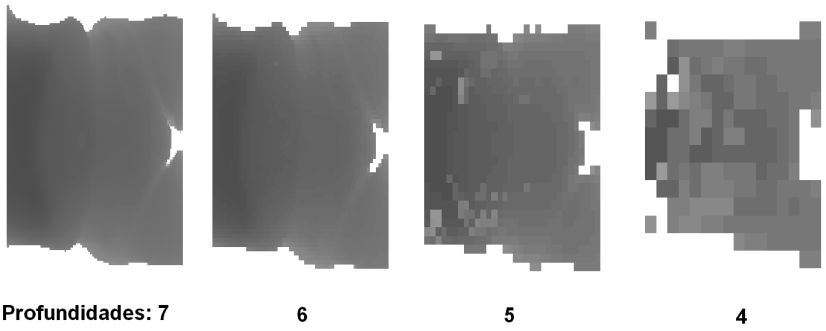


Figura 5.4: Regiões de oclusão do Stent para as profundidades 7,6,5 e 4.

Conforme mencionado anteriormente, uma das vantagens de utilizarmos o método da oclusão implícita, é o fato de que podemos visualizar um dado com diferentes isovalores, sem a necessidade de remontarmos a octree. Na figura 5.5 temos a imagem do Stent com 2 isovalores: 600 e 1300. No entanto, o occluder não se mostra eficiente no isovalor 1300, pois não há quase região de oclusão. A tabela seguinte exibe os resultados obtidos neste caso.

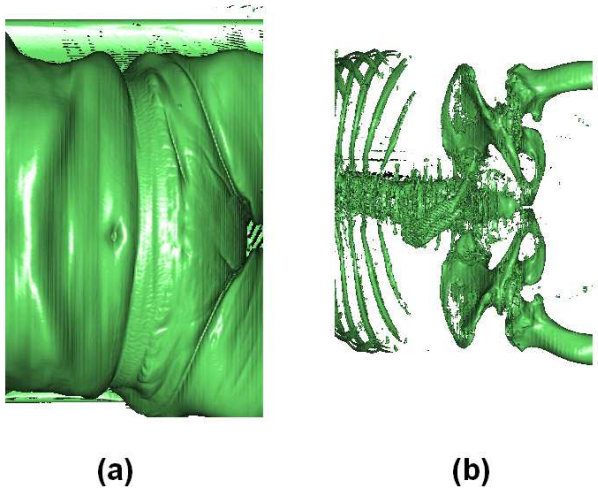
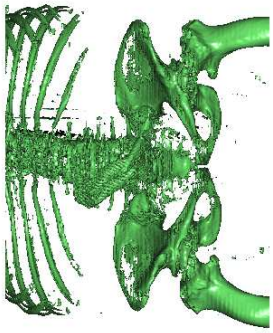


Figura 5.5: Stent16.raw: (a) com isovalor 600 referente à pele (b) isovalor 1300 referente aos ossos.

Stent com isovalor 1300	**Região de oclusão**
tempo com occluder	4,421
tempo sem occluder	4,250
n° de faces sem occluder	1.198.084
n° de faces com occluder	1.198.084

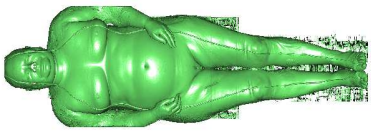
Stent com isovalor 1300	Região de oclusão
	

tempo com occluder	4,421
tempo sem occluder	4,250
nº de faces sem occluder	1.198.084
nº de faces com occluder	1.198.084

Figura 5.6: Para o isovalor 1300, a quantidade de faces com/sem ocluder é a mesma.

5.2 Visible Woman

O dado volumétrico da Visible Woman é extenso e tem 887 MB. Neste dataset é possível visualizar pele e ossos dependendo do valor da isosuperfície. Observe nas tabelas a seguir as imagens da Visible Woman, a região de oclusão gerada e seus respectivos tempos de renderização.

Figura	Região de oclusão
	

Arquivo:	Visible_woman.raw
Dimensão:	1734x512x512
Câmera:	-450,450,90
Profundidade:	7
Isolevel:	600

Figura 5.7: Dados da Visible Woman

Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,187	
Teste de visibilidade:	0,219	
Marching Cubes:	17,889	59,00
Total:	18,297	59,00
Número de faces renderizadas:	4.801.044	15.514.608
Redução do tempo total:	31%	
Redução do número de faces:	31%	

Para este exemplo, foram renderizados apenas 4.801.044 de faces, que correspondem 31% do total de faces para a posição de câmera mencionada na tabela. Note que 10.713.564 de triângulos deixaram de ser renderizados com o método da oclusão implícita.

Obtivemos o melhor tempo de renderização de 18,297 segundos com uma octree de profundidade 7, verificando a visibilidade dos nós até o último nível, conforme nos mostra a tabela 5.2.

Mapa de oclusão		**Teste de visibilidade**	
Profundidade 7	**Nível 7**	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,187	0,172	0,187
Teste de visibilidade:	0,219	0,078	0,031
Marching Cubes:	17,889	20,187	25,485
Total:	**18,297**	20,437	25,702
Triângulos:	4.801.044	5.420.342	6.906.328
Profundidade 6	**Nível 6**	**Nível 5**	
Construção da reg. de ocl.:	0,047	0,047	
Teste de visibilidade:	0,031	0,015	
Marching Cubes:	25,406	31,656	
Total:	25,485	31,733	
Triângulos:	6.041.543	7.662.143	

Tabela 5.2: Comparação dos tempos para diferentes níveis de visibilidade da Visible Woman

Na figura 5.8 são apresentados os tempos gastos com/sem occluder para diferentes isovalores da Visible Woman. Observe que a partir do isolevel 900, os tempos com/sem occluder começam a coincidir, devido à redução da região de oclusão.

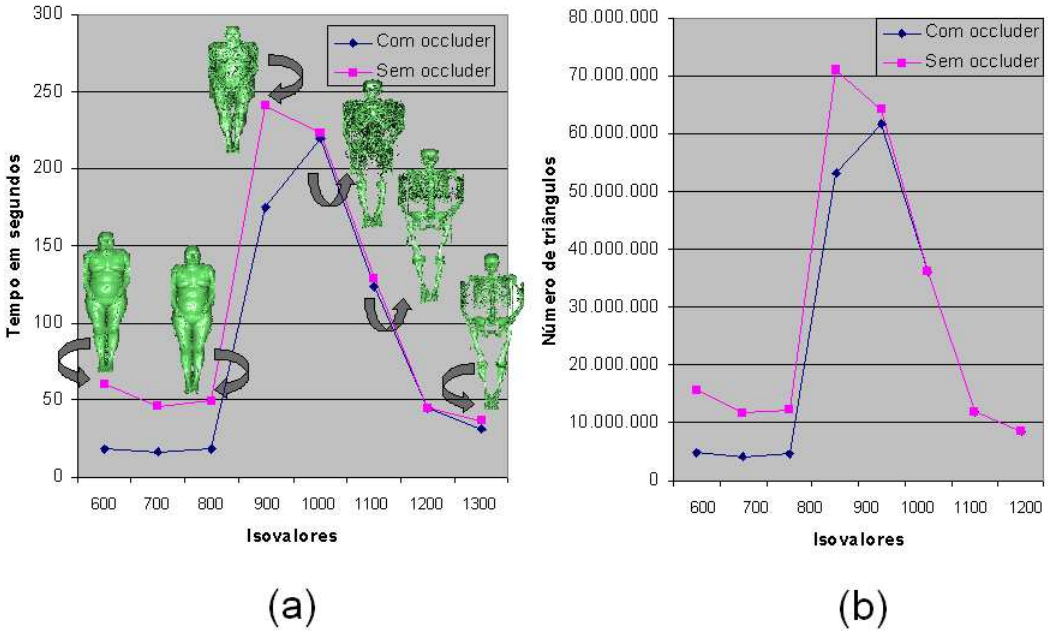


Figura 5.8: (a) gráfico com os tempos em segundos da renderização da Visible Woman para diferentes isovalores. (b) gráfico do número de faces renderizadas para diferentes isovalores.

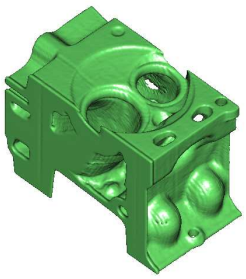
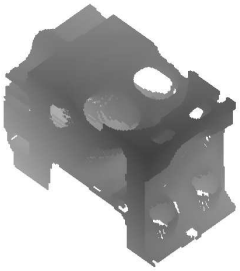
5.3 Engine

Imagem de dois cilindros de um bloco de motor. No arquivo Engine fixamos uma determinada posição de câmera e obtivemos uma redução de 392.854 faces, sendo renderizados apenas 40% do total de faces.

Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,172	
Teste de visibilidade:	0,141	
Marching Cubes:	0,936	2,342
Total:	1,25	2,342
Número de faces renderizadas:	256.726	643.580
Redução do tempo total:	60%	
Redução do número de faces:	60%	

A figura 5.10 nos mostra os tempos gastos com/sem occluder para diferentes posições de câmera do engine, bem como o número de faces renderizadas. Observe também na figura 5.11 os tempos gastos para cada etapa do algoritmo: criação da região de oclusão, teste de visibilidade e render.

O melhor tempo de renderização foi obtido numa octree de profundidade 7, verificando a visibilidade dos nós até o último nível, conforme nos mostra a

Figura	Região de oclusão
	

Arquivo :	engine.raw
Dimensão:	128x256x256
Câmera:	135,45,90
Profundidade:	7
Isolevel:	120

Figura 5.9: Dados do Engine.

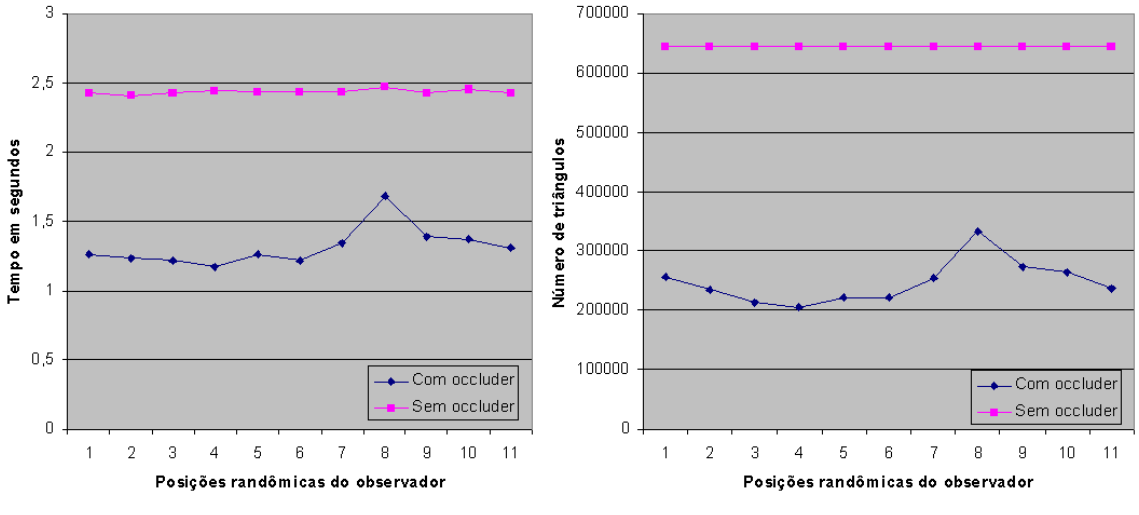


Figura 5.10: Gráficos do dado volumétrico Engine de profundidade 7 e isolevel 120. (a) Comparação dos tempos totais com/sem occluder. (b) Comparação do número de triângulos com/sem occluder.

tabela 5.3.

5.4 Foot

Imagem de um pé humano obtido através de uma radiografia. Tecidos e ossos estão presentes no dataset.

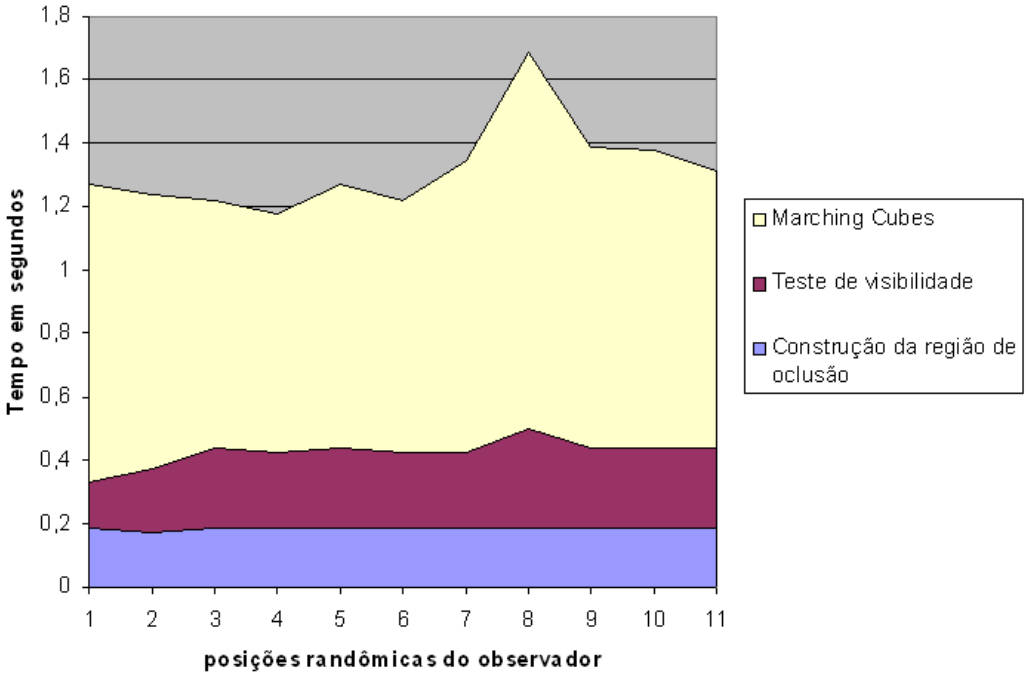


Figura 5.11: Gráfico do dado volumétrico engine de profundidade 7 e isolevel 120. Comparações dos tempos de criação da região de oclusão, teste de visibilidade, tempo de renderização.

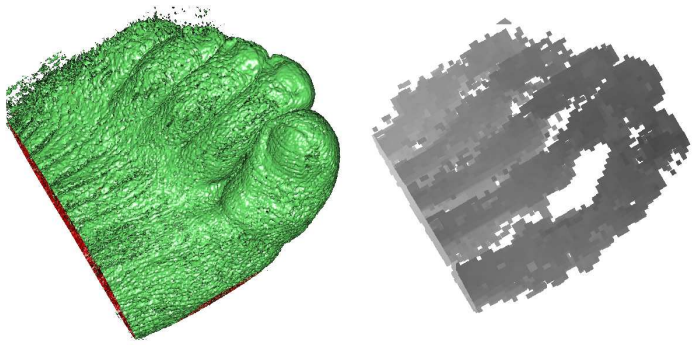
Mapa de oclusão		**Teste de visibilidade**	
Profundidade 7	**Nível 7**	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,172	0,174	0,172
Teste de visibilidade:	0,141	0,093	0,046
Marching Cubes:	0,936	1,11	1,485
Total:	**1,25**	1,375	1,702
Triângulos:	256.726	321.182	426.191
Profundidade 6	**Nível 6**	**Nível 5**	
Construção da reg. de ocl.:	0,031	0,031	
Teste de visibilidade:	0,047	0,032	
Marching Cubes:	1,672	1,983	
Total:	1,75	2,046	
Triângulos:	468.549	556.783	

Tabela 5.3: Comparação dos tempos para diferentes níveis de visibilidade do Engine.

Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,296	
Teste de visibilidade:	0,141	
Marching Cubes:	6,311	9,312
Total:	6,75	9,312
Número de faces renderizadas:	1.864.009	2.577.814
Redução do tempo total:	28%	
Redução do número de faces:	28%	

Podemos verificar que o melhor tempo de renderização do foot foi obtido

Figura	Região de oclusão



Arquivo :	foot.raw
Dimensão:	256x256x256
Câmera:	150,185,90
Profundidade:	6
Isolevel:	15

Figura 5.12: Dados do Foot.

numa octree de profundidade 7, verificando a visibilidade até o nível 6, conforme nos mostra a tabela 5.4.

Mapa de oclusão		**Teste de visibilidade**	
Profundidade 7	**Nível 7**	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,296	0,296	0,296
Teste de visibilidade:	0,375	0,141	0,078
Marching Cubes:	6,25	6,311	7,031
Total:	6,921	**6,75**	7,406
Triângulos:	1.739.143	1.864.009	2.061.062
Profundidade 6	**Nível 6**	**Nível 5**	
Construção da reg. de ocl.:	0,047	0,062	
Teste de visibilidade:	0,078	0,032	
Marching Cubes:	7,453	8,203	
Total:	7,578	8,296	
Triângulos:	2.184.169	2.388.112	

Tabela 5.4: Comparação dos tempos para diferentes níveis de visibilidade do Foot.

5.5 Ppm

O dado volumétrico Ppm é extenso e possui 1,179 GB. Utilizamos duas posições de câmera onde podemos visualizar o Ppm de frente e de lado, e mostrar a eficiência do ocluder para este exemplo.

5.5.1 Frente

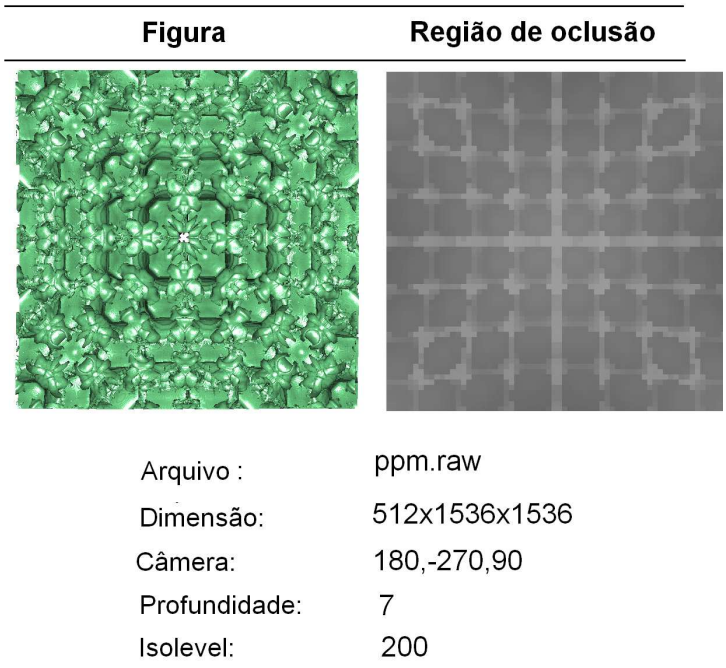


Figura 5.13: Dados do Ppm (frente).

Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,391	
Teste de visibilidade:	0,405	
Marching Cubes:	127,235	198,483
Total:	128,031	198,483
Número de faces renderizadas:	30.178.366	49.181.578
Redução do tempo total:	36%	
Redução do número de faces:	39%	

O melhor tempo obtido para este modelo foi numa octree de profundidade 7, verificando a visibilidade até o último nível, conforme nos mostra a tabela 5.5.

Mapa de oclusão		**Teste de visibilidade**	
Profundidade 7	**Nível 7**	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,391	0,391	0,391
Teste de visibilidade:	0,405	0,171	0,109
Marching Cubes:	127,235	151,766	193,656
Total:	**128,031**	152,328	194,156
Triângulos:	30.178.366	36.205.689	46.560.371
Profundidade 6	**Nível 6**	**Nível 5**	
Construção da reg. de ocl.:	0,078	0,063	
Teste de visibilidade:	0,094	0,031	
Marching Cubes:	200,000	230,906	
Total:	200,172	231,00	
Triângulos:	42.107.976	48.391.508	

Tabela 5.5: Comparação dos tempos para diferentes níveis de visibilidade do Ppm.

5.5.2 Lado

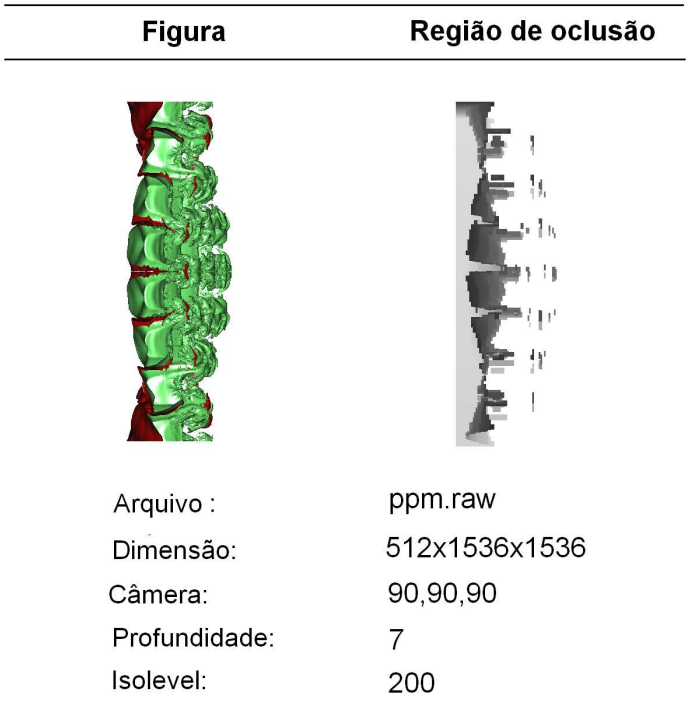


Figura 5.14: Dados do Ppm (Lado).

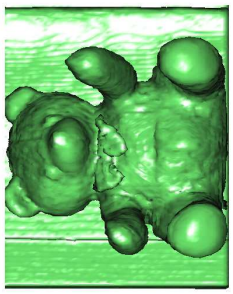
Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,36	
Teste de visibilidade:	0,671	
Marching Cubes:	152,266	198,406
Total:	153,297	198,406
Número de faces renderizadas:	36.855.781	49.181.578
Redução do tempo total:	23%	
Redução do número de faces:	26%	

Mapa de oclusão		**Teste de visibilidade**	
Profundidade 7	**Nível 7**	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,36	0,359	0,359
Teste de visibilidade:	0,671	0,202	0,093
Marching Cubes:	152,266	169,141	190,860
Total:	**153,297**	169,702	190,312
Triângulos:	36.855.781	42.956.213	48.420.702
Profundidade 6	**Nível 6**	**Nível 5**	
Construção da reg. de ocl.:	0,062	0,079	
Teste de visibilidade:	0,109	0,046	
Marching Cubes:	213,610	220,563	
Total:	213,781	220,688	
Triângulos:	47.375.548	48.887.798	

Tabela 5.6: Comparação dos tempos para diferentes níveis de visibilidade do Ppm (lado).

5.6 Teddy Bear

Imagem de um urso de pelúcia com dimensões 64 x 128 x 128.

Figura	Região de oclusão
	

Arquivo :	tedybear.raw
Dimensão:	64x128x128
Câmera:	270,-270,90
Profundidade:	6
Isolevel:	50

Figura 5.15: Dados do Teddy Bear.

Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,032	
Teste de visibilidade:	0,031	
Marching Cubes:	0,515	0,639
Total:	0,579	0,639
Número de faces renderizadas:	127.604	168.904
Redução do tempo total:	10%	
Redução do número de faces:	24%	

Note que para a profundidade 7, o tempo gasto no Marching Cubes com/sem ocluder foi de 0,469 e 0,828 respectivamente. No entanto, o tempo gasto na região de oclusão e no teste de visibilidade, fez com que o tempo total com ocluder ficasse maior do que o tempo sem ocluder. Na tabela 5.7, podemos verificar que o melhor tempo de renderização do TeddyBear foi obtido numa octree de profundidade 6, verificando a visibilidade até o nível 5.

Podemos notar pela figura 5.16 que já na profundidade 6 temos uma boa região de oclusão, não sendo necessário refinar o ocluder.

Mapa de oclusão		**Teste de visibilidade**	
Profundidade 7	**Nível 7**	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,204	0,217	0,202
Teste de visibilidade:	0,264	0,094	0,063
Marching Cubes:	0,469	0,500	0,561
Total:	0,938	0,811	0,828
Triângulos:	95.841	106.104	122.210
Profundidade 6	**Nível 6**	**Nível 5**	
Construção da reg. de ocl.:	0,047	0,032	
Teste de visibilidade:	0,078	0,031	
Marching Cubes:	0,469	0,515	
Total:	0,593	**0,579**	
Triângulos:	116.603	127.604	

Tabela 5.7: Comparação dos tempos para diferentes níveis de visibilidade do Teddy Bear.

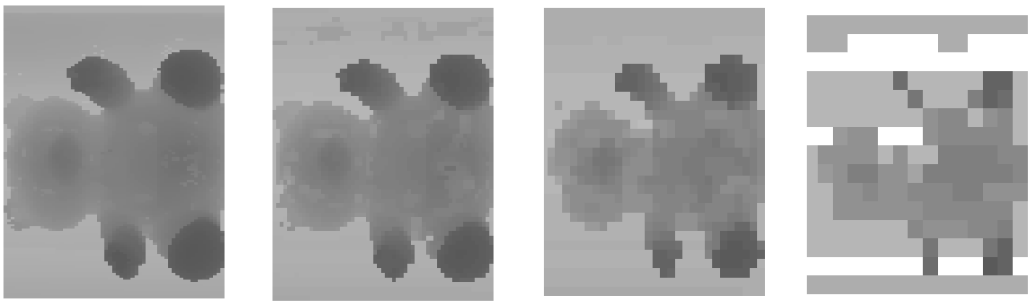


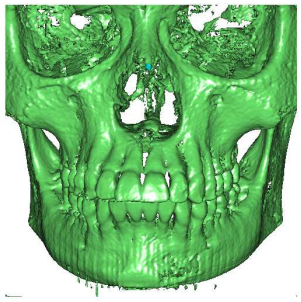
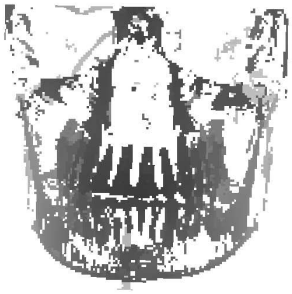
Figura 5.16: Regiões de oclusão para os níveis 7,6,5 e 4 respectivamente.

5.7 Skull

Imagem gerada através da radiografia de um crânio humano.

Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,250	
Teste de visibilidade:	0,375	
Marching Cubes:	6,578	7,282
Total:	7,203	7,282
Número de faces renderizadas:	1.812.836	1.990.867
Redução do tempo total:	2%	
Redução do número de faces:	9%	

Neste caso, o ocluder obtido na profundidade 7 não é de boa qualidade e a redução das faces foi mínima. Já na profundidade 6, não há região de oclusão e na tabela 5.8 podemos verificar que a quantidade de triângulos renderizados

Figura	Região de oclusão
	

Arquivo :	skull.raw
Dimensão:	256x256x256
Câmera:	540,-180,90
Profundidade:	7
Isolevel:	40

Figura 5.17: Dados do Skull

é exatamente o total de faces.

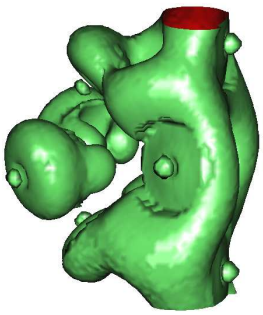
Mapa de oclusão		**Teste de visibilidade**	
Profundidade 7	**Nível 7**	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,25	0,25	0,234
Teste de visibilidade:	0,375	0,171	0,094
Marching Cubes:	6,578	7,079	7,203
Total:	**7,203**	7,5	7,531
Triângulos:	1.812.836	1.954.789	1.989.478
Profundidade 6	**Nível 6**	**Nível 5**	
Construção da reg. de ocl.:	0,047	0,047	
Teste de visibilidade:	0,078	0,046	
Marching Cubes:	7,203	7,188	
Total:	7,328	7,281	
Triângulos:	1.990.867	1.990.867	

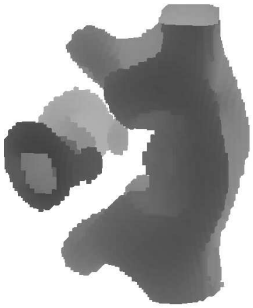
Tabela 5.8: Comparação dos tempos para diferentes níveis de visibilidade do Skull.

5.8 Neghip

Imagem de uma simulação da distribuição espacial dos elétrons numa molécula de proteína.

Figura	Região de oclusão





Arquivo :	neghip.raw
Dimensão:	64x64x64
Câmera:	350x440x90
Profundidade:	6
Isolevel:	20

Figura 5.18: Dados do Neghip

Com ocluder		**Sem ocluder**
Construção da reg. de ocl.:	0,047	
Teste de visibilidade:	0,016	
Marching Cubes:	0,125	0,18
Total:	0,202	0,18
Número de faces renderizadas:	33.392	44.520

Mapa de oclusão	**Teste de visibilidade**	
Profundidade 6	**Nível 6**	**Nível 5**
Construção da reg. de ocl.:	0,032	0,047
Teste de visibilidade:	0,062	0,016
Marching Cubes:	0,141	0,125
Total:	0,235	**0,202**
Triângulos:	29.521	33.392

Tabela 5.9: Comparação dos tempos para diferentes níveis de visibilidade do Neghip.

Para este exemplo, obtivemos o melhor tempo numa octree de profundidade 6, verificando a visibilidade até o nível 5, no entanto, este tempo foi maior do que sem ocluder. Devido ao tamanho do dado e à pouca região de

oclusão, os tempos de construção da região de oclusão e do teste de visibilidade oneraram o tempo total com ocluder.

6
Aplicação da oclusão implícita em silhuetas

Silhuetas desempenham um importante papel na compreensão de um dado volumétrico. Neste capítulo apresentaremos uma aplicação da oclusão implícita na geração de silhuetas.

6.1
Introdução

A silhueta de uma superfície S é formada pelos pontos $p \in S$ tal que $V.N_p = 0$ onde V é o vetor observador e N_p é a normal à superfície no ponto p.

Seguindo DeCarlo et al. (1), para o dado volumétrico podemos considerar uma superfície silhueta formada pelo conjunto de curvas silhuetas de todas as possíveis isosuperfícies para um observador fixo. A superfície silhueta seria a isosuperfície associada ao isovalor 0 da função $c(i, j, k) = -\nabla f(i, j, k).v(i, j, k)$. Assim, podemos encontrar a silhueta de uma superfície particular através do algoritmo de Marching Lines (13), que consiste no cálculo da interseção de duas funções implícitas: o da isosuperfície calculada pelo marching Cubes e a superfície de contorno, conforme ilustra a figura 6.1 de De Carlo et al. (1).

A desvantagem do método é que, se calcularmos todos os pontos onde $n.v = 0$, teremos não apenas a silhueta exterior da figura, mas também a interior. DeCarlo et al. (1) propõe um método que evita renderizar porções não visíveis da silhueta, que consiste em traçar um raio do observador até cada vértice gerado na silhueta e verificar se há interseção com a isosuperfície ou não, examinando a interseção do raio com as faces do grid volumétrico. O autor menciona que o teste de visibilidade usado neste caso é oneroso.

Utilizando o método de oclusão implícita podemos reduzir o tempo de obtenção da silhueta exterior, já que estaremos calculando Marching Lines somente nos nós visíveis da octree. Visto que nosso algoritmo é conservativo, teremos silhuetas interiores na imagem, porém, quanto maior a profundidade da octree, melhor a qualidade do ocluder e portanto menos silhuetas internas serão visíveis, aproximando-se assim da silhueta real do objeto.

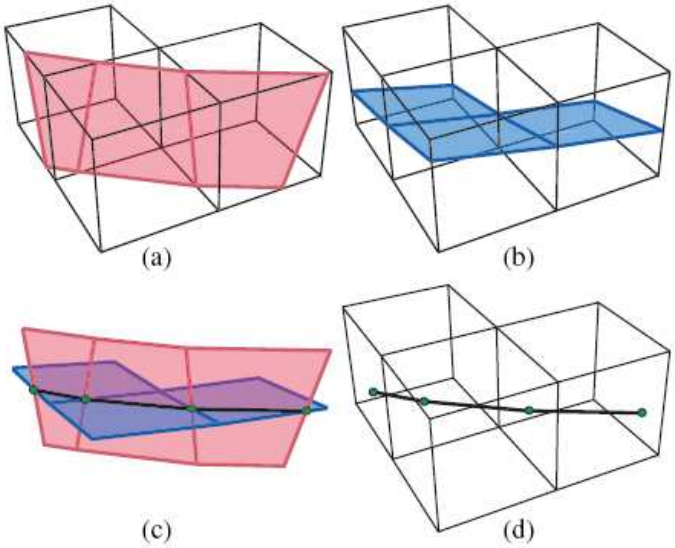


Figura 6.1: Marching Lines: (a) isosuperfície (b) superfície de contorno (c) interseção entre isosuperfície e superfície de contorno (d) silhueta.

6.2 Marching Lines

Para utilizarmos o algoritmo de Marching Lines, precisamos calcular primeiramente a isosuperfície através do algoritmo de Marching Cubes: Na aresta do cubo (P_i, P_j) em que houver mudança de sinal, utilizamos interpolação linear para encontrar o ponto P_{ij} que aproxima a isosuperfície:

$P_{ij} = P_i.t_{ij} + (1-t).P_j$, onde t_{ij} varia de [0,1].

Guardamos a informação do t_{ij} de cada aresta em que há isosuperfície. O próximo passo é descobrir o valor de $g = V.N_p$ (onde V é o vetor da posição do observador e N_p é o vetor normal) em cada ponto P_{ij} do triângulo gerado pelo Marching Cubes, utilizando o parâmetro t_{ij} :

$g_{ij} = g_i.t_{ij} + (1-t).g_j$, onde g_i e g_j são os valores de $V.N_p$ na aresta (i , j) em que há isosuperfície.

Verificamos em cada aresta do triângulo se há mudança de sinal de g. Se houver, podemos encontrar pontos em que $g = 0$ através da interpolação linear. Repetimos o processo para as 3 arestas de cada triângulo da isosuperfície e assim, determinamos uma aresta que pertence a silhueta, conforme ilustra a figura 6.2.

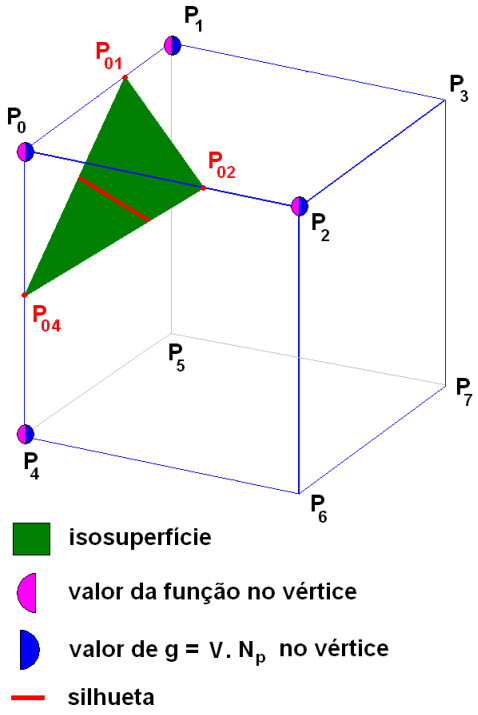


Figura 6.2: Verificamos em cada aresta do triângulo se há mudança de sinal de g.

7
Resultados da silhueta

De Carlo et al. (1) propõe um algoritmo que percorre a curva silhueta, uma vez obtida a semente inicial. Neste trabalho optamos por extrair a silhueta em cada nó considerado visível, ao invés de percorrer toda a silhueta. Porém a aplicação da oclusão implícita pode ser feita em ambas as estratégias.

A figura 7.1 exibe o resultado obtido para o engine com um mapa de oclusão gerado por uma octree de profundidade 7. Em 7.1 (a) temos a silhueta sem ocluder e em 7.1 (b) o resultado utilizando-se a oclusão implícita.

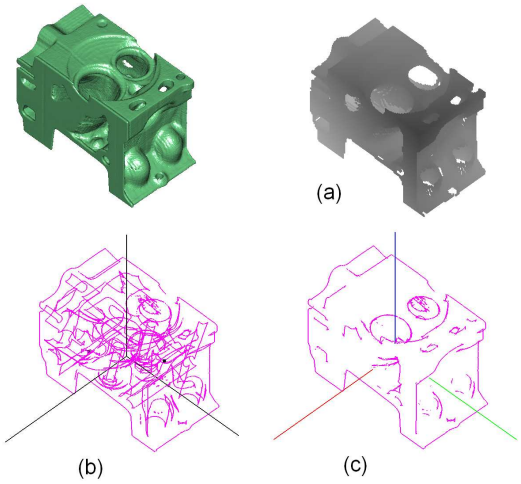


Figura 7.1: Arquivo "engine.raw"com profundidade 7: (a) região de oclusão (b) silhueta sem occluder (c) silhueta com occluder.

A tabela 7.1 resume a composição dos tempos obtidos comparando-se a silhueta com e sem ocluder. No caso da profundidade 7, uma vez que o mapa de oclusão é de boa qualidade, há uma economia de 27% do tempo se comparado com a silhueta com ocluder, uma vez que o algoritmo de Marching Lines só é aplicado nos nós visíveis.

A figura 7.2 exibe uma comparação entre a silhueta obtida com a oclusão implícita e a silhueta exterior real (obtida renderizando o objeto sem iluminação sobre a silhueta). Para a octree de profundidade 7, observa-se que boa parte da silhueta interna foi removida.

Mapa de oclusão	**Prof. 7**	**Prof. 6**	**Prof. 5**
Construção da reg. de ocl.:	0,202	0,152	0,031
Teste de visibilidade:	0,248	0,2	0,079
Marching Lines:	0,535	1,358	1,956
Total:	0,985	1,71	2,066

Tabela 7.1: Comparação dos tempos de renderização da silhueta para profundidades diferentes do Engine.

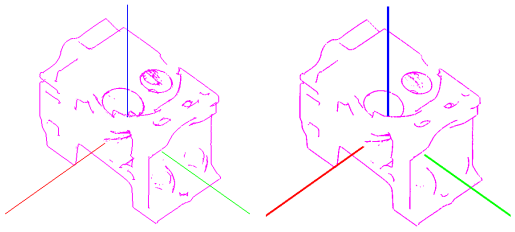


Figura 7.2: Engine, profundidade 7, isovalor 120: (a) silhueta com ocluder (b) silhueta real.

A tabela seguinte resume a composição dos tempos obtidos nesse exemplo.

Engine, prof 7, isov.: 120	**tempo do cálculo da silhueta:**
com ocluder:	0,985 segundos
sem ocluder:	1,343 segundos

A qualidade do ocluder influencia diretamente na remoção das silhuetas não-visíveis. Neste exemplo, se reduzirmos a profundidade da octree para 6 e 5, obteremos as silhuetas da figura 7.3. Observe que na profundidade 5 quase não há ocluder.

Em uma visão completa da Visible Woman, o cálculo da silhueta exterior foi obtido em 36% do tempo sem ocluder, como mostra a figura 7.4.

Em exemplos que geram bons mapas de oclusão, obtemos na maioria dos casos uma sensível eliminação das silhuetas não visíveis. Isto pode ser observado também nos exemplos do Stent (figura 7.5), Statueleg (figura 7.6), Neghip (figura 7.7), Bonsai (figura 7.8), Ppm frente (figura 7.9) e PPm lado (figura 7.10).

Nos casos em que temos baixa qualidade do ocluder, não há remoção da silhueta não visível suficiente para justificar o tempo de geração e teste de oclusão como o exemplo do Skull.

Em nossa implementação, existe a possibilidade de utilizarmos 2 isovalores para verificar a silhueta de 2 isosuperfícies. Com isso, a rotina SetSinal mencionada no capítulo 4 sofre uma pequena alteração, descrita na tabela 7.2.

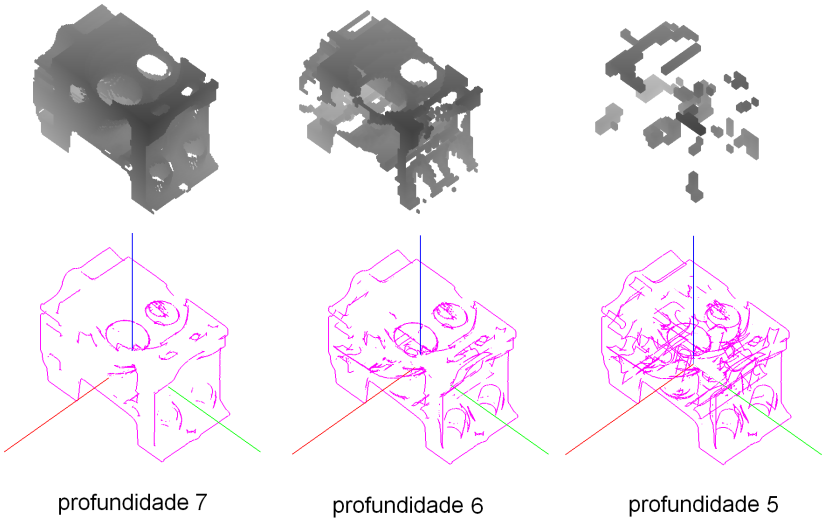


Figura 7.3: Comparação das silhuetas do engine para as profundidades 7, 6 e 5 com suas respectivas regiões de oclusão.

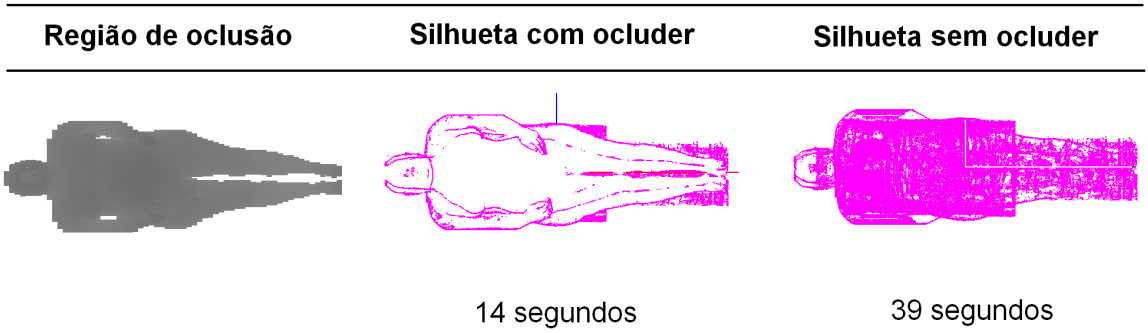


Figura 7.4: Tempos da silhueta com/sem ocluder da Visible Woman. Para este exemplo, o tempo de renderização com ocluder corresponde à 36% do tempo de renderzação sem ocluder, logo, obtivemos uma economia de 64% do tempo através do método da oclusão implícita.

Pseudo código da rotina SetSinal(isolevel1, isolevel2)
- Se _vmax < isolevel1 e _vmax < isovalor2 O sinal do nó é negativo (0) ; - Senão - Se _vmin > isolevel1 e _vmin > isovalor2 O sinal do nó é positivo (1); - Senão Há mudança de sinal (2);

Tabela 7.2: Pseudo codigo da rotina SetSinal

A região de oclusão diminui se escolhermos um intervalo para os isovalores. Note a diferença da região de oclusão para o caso em que isovalor1 =

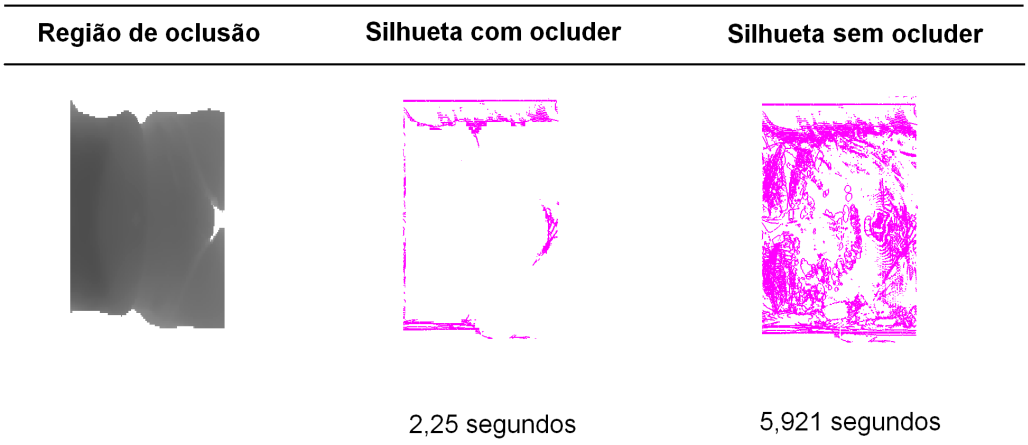


Figura 7.5: Tempos da silhueta com/sem ocluder do Stent. Para este exemplo obtivemos uma economia de 62% do tempo através do método da oclusão implícita

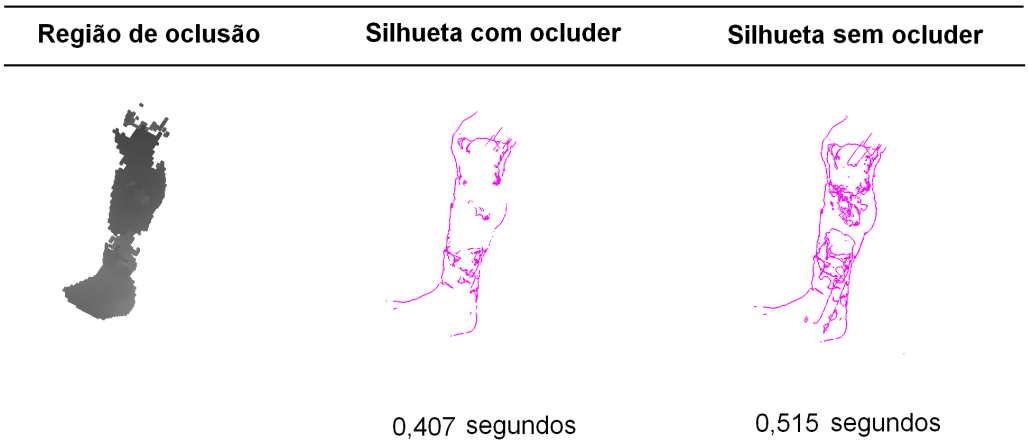


Figura 7.6: Tempos da silhueta com/sem ocluder do dado Statueleg. Para este exemplo, obtivemos uma economia de 21% do tempo sem ocluder.

isovalor2 = 5, para o caso em que o isovalor1 = 5 e isovalor2 = 70 nas figuras 7.12 e 7.13.

Observe o exemplo da figura 7.14 do dado volumétrico do pé da visible woman de profundidade 6 e isovalores: 600 (azul) referente à pele e 1200 (cinza) referente aos ossos.

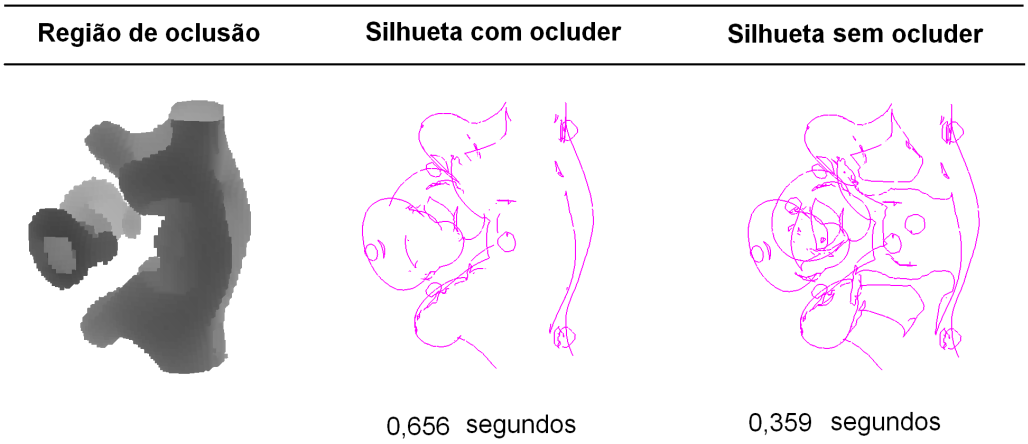


Figura 7.7: Tempos da silhueta com/sem ocluder do Neghip. Devido ao tamanho pequeno do dado e à pouca região de oclusão, o tempo gasto com a construção da região de oclusão e com o teste de visibilidade acabou onerando o tempo total com ocluder.

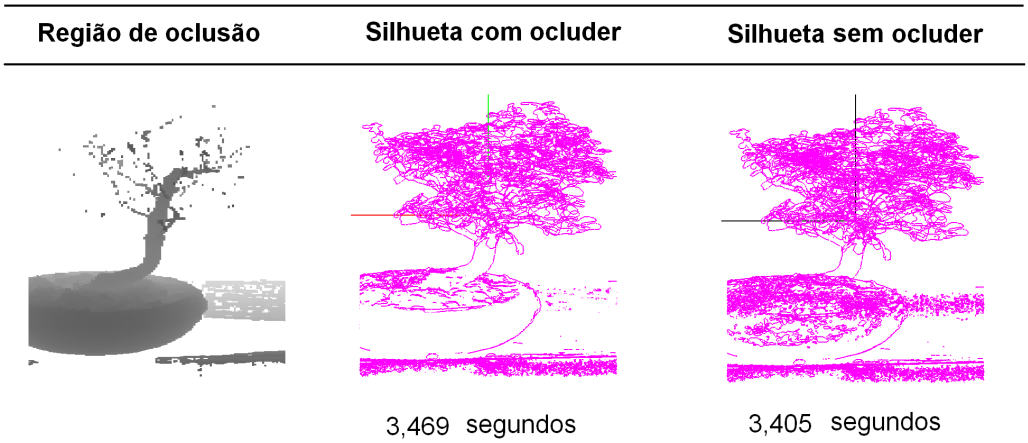


Figura 7.8: Tempos da silhueta com/sem ocluder do Bonsai. Para este exemplo, também não houve economia de tempo para o cálculo da silhueta com ocluder, devido à pouca região de oclusão do dado.

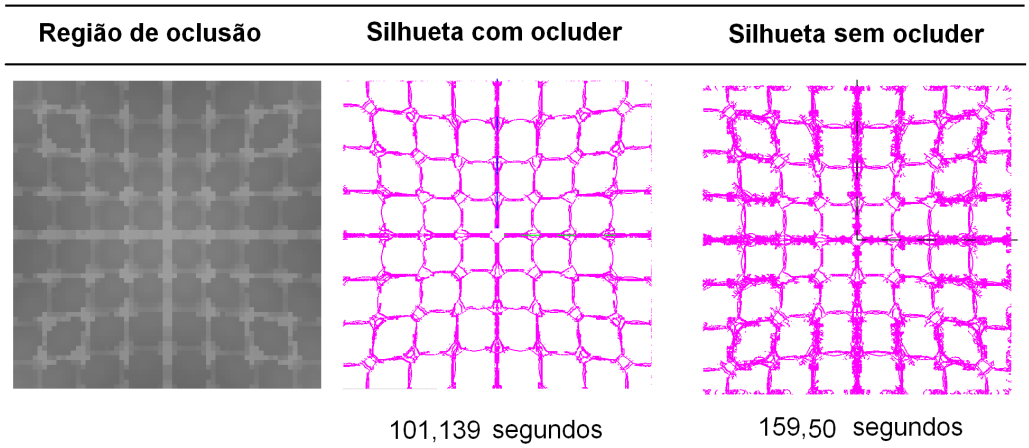


Figura 7.9: Tempos da silhueta com/sem ocluder do Ppm (frente). Para o dado do Ppm de frente, obtivemos uma economia de 37% do tempo sem ocluder.

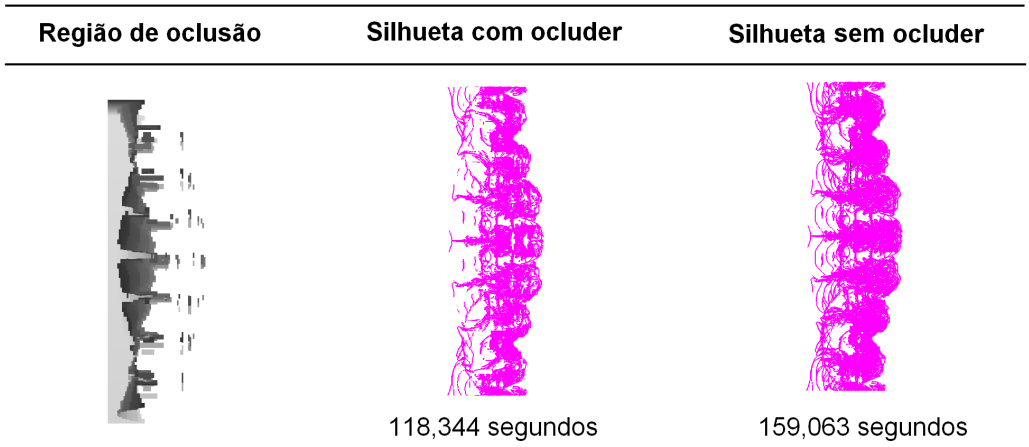


Figura 7.10: Tempos da silhueta com/sem ocluder do Ppm (lado). Para o dado do Ppm de lado, obtivemos uma economia de 26% do tempo sem ocluder.

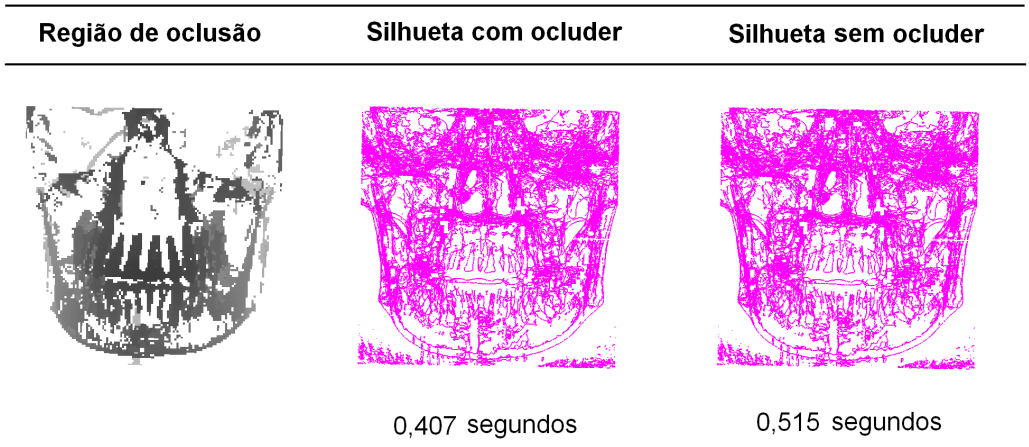


Figura 7.11: Tempos da silhueta com/sem ocluder do Skull. Para este exemplo, obtivemos uma economia de 21% do tempo sem coluder.

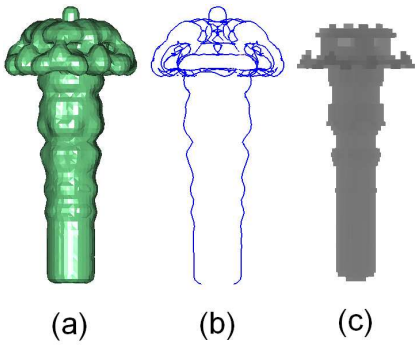


Figura 7.12: (a) isosuperfície (b) silhueta (c) região de oclusão para isovalor1 = 5 e isovalor2 = 5

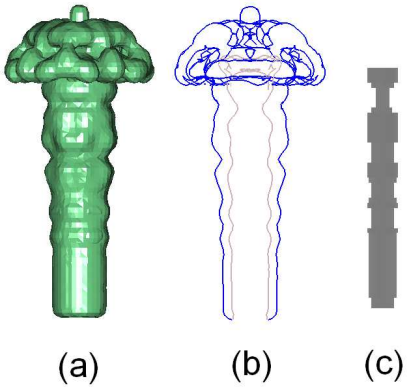


Figura 7.13: (a) isosuperfície (b) silhueta (c) região de oclusão para isovalor1 = 5 e isovalor2 = 70

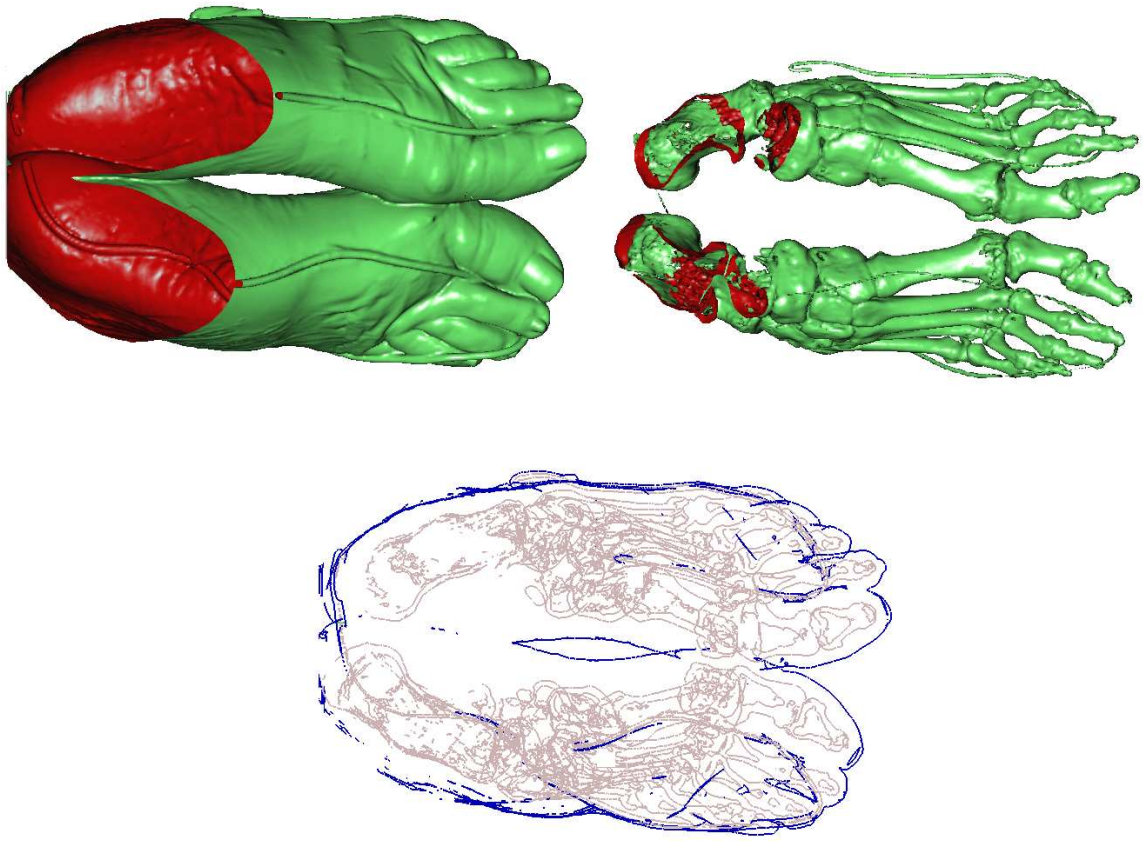


Figura 7.14: Silhuetas do pé da visible woman para isovalores 600 (referente à pele) e 1200 (referente aos ossos).

8
Conclusão e trabalhos futuros

Esta dissertação apresentou o método de oclusão implícita para aceleração da extração e rendering de isosuperfícies e sua aplicação na visualização da silhueta exterior do objeto.

Conforme vimos no capítulo 5, na maioria dos exemplos descritos houve redução no tempo, principalmente nos modelos em que o mapa de oclusão é grande, como no caso do Ppm (frente) e Visible Woman, onde obtivemos uma redução de 43% e 69% do tempo de renderização respectivamente. Nesses dados, como o dado volumétrico é razoavelmente grande, havendo muitas faces na isosuperfície desejada (15 milhões no isovalor 600 da Visible Woman, por exemplo), o tempo de geração do mapa de oclusão e o teste de visibilidade serão praticamente mínimos se comparados ao tempo gasto pelo Marching Cubes. O tempo de geração do ocluder (e teste) está limitado à profundidade da octree, uma vez que corresponde a renderização de uma fração do número máximo de nós da octree.

O ocluder se mostra eficiente à medida que o observador se aproxima do objeto. A cada aumento do zoom, diminui a região visível (já excetuando a região eliminada pelo view-frustum culling), diminuindo assim, o número de faces a serem renderizadas.

No entanto, existem situações em que a oclusão implícita não se mostra eficiente. Dependendo do dado e do isovalor, a região de oclusão pode não ser significativa (esqueleto do Stent), e nesses casos a geração da oclusão e teste significam custos extras a serem adicionados.

Com relação à silhueta, podemos concluir que a oclusão implícita se mostrou adequada na maioria dos exemplos, para a remoção de silhuetas não visíveis. A vantagem é o baixo custo da geração do ocluder e do teste de visibilidade em dados volumétricos relativamente grandes (quando comparados aos custos envolvidos nesses casos).

Para dados muito maiores, que não podem ser carregados em memória, a oclusão implícita poderia determinar regiões potencialmente visíveis para um modelo out-of-core de visualização, como trabalhos futuros. Também, nosso algoritmo foi testado para grids regulares, mas pode ser extendido para

grids não regulares (tetraedros). Neste caso, seria necessário tratar o caso não convexo.

Referências Bibliográficas

[1] BURNS, M.; KLAWE, J.; RUSINKIEWICZ, S.; FINKELSTEIN, A. ; DECARLO, D.. **Line drawings from volume data**. ACM Transactions on Graphics (Proc. SIGGRAPH), 24(3):512–518, Aug. 2005. 1.2, 6.1, 6.1, 7

[2] CHIANG, Y.; SILVA, C.. **External memory techniques for isosurface extraction in scientific visualization**. In: Abello, J.; Vitter, J., editors, EXTERNAL MEMORY ALGORITHMS AND VISUALIZATION, volumen 50 de **DIMACS Book Series**, p. 247–277. American Mathematical Society, 1999. 3.1.2

[3] CHIANG, Y.-J.; SILVA, C. T.. **I/o optimal isosurface extraction (extended abstract).** In: IEEE VISUALIZATION, p. 293–300, 1997. 3.1.2

[4] CIGNONI, P.; MONTANI, C.; PUPPO, E. ; SCOPIGNO, R.. **Optimal isosurface extraction from irregular volume data**. In: VVS '96: PROCEEDINGS OF THE 1996 SYMPOSIUM ON VOLUME VISUALIZATION, p. 31–38, Piscataway, NJ, USA, 1996. IEEE Press. 3.1.2

[5] GAO, J.; SHEN, H.-W.. **Parallel view-dependent isosurface extraction using multi-pass occlusion culling**. In: PVG '01: PROCEEDINGS OF THE IEEE 2001 SYMPOSIUM ON PARALLEL AND LARGE-DATA VISUALIZATION AND GRAPHICS, p. 67–74, Piscataway, NJ, USA, 2001. IEEE Press. 3.1.2

[6] GREENE, N.. **Hierarchical polygon tiling with coverage masks**. In: SIGGRAPH '96: PROCEEDINGS OF THE 23RD ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 65–74, New York, NY, USA, 1996. ACM Press. 3.1.3

[7] LIVNAT, Y.. **Accelerated isosurface extraction approaches**. In: Hansen, C.; Johnson, C., editors, THE VISUALIZATION HANDBOOK, p. 39–55. Elsevier, 2005. 3.1.4

[8] LIVNAT, Y.; HANSEN, C.. **View dependent isosurface extraction**. In: IEEE VISUALIZATION '98, p. 175–180, Oct 1998. 3, 3.1, 3.1.2, 3.1.3, 4

[9] LIVNAT, Y.; SHEN, H. ; JOHNSON, C.. **A near optimal isosurface extraction algorithm for structured and unstructured grids**. IEEE Transactions on Visual Computer Graphics, 2(1):73–84, 1996. 3.1.3

[10] LORENSEN, W. E.; CLINE, H. E.. **Marching cubes: A high resolution 3d surface construction algorithm**. In: SIGGRAPH '87: PROCEEDINGS OF THE 14TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, volumen 21, p. 163–169, New York, NY, USA, July 1987. ACM Press. 1.1, 3.1.1, 4, 4.4

[11] PARKER, S.; SHIRLEY, P.; LIVNAT, Y.; HANSEN, C. ; SLOAN, P.-P.. **Interactive ray tracing for isosurface extraction**. In: IEEE VISUALIZATION '98, p. 233–238, October 1998. 3.1.3

[12] SINESIO PESCO, PETER LINDSTROM, V. P.; SILVA, C. T.. **Implicit occluders**. 2004. 1.3, 3.1.4, 4

[13] THIRION, J.-P.; GOURDON, A.. **Marching lines algorithm: new results and proofs**. April 1993. 6.1

[14] WILHELMS, J.; GELDER, A. V.. **Octrees for faster isosurface generation**. ACM Trans. Graph., 11(3):201–227, 1992. 1.3, 3.1.1, 4, 4.1.1, 5

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo