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

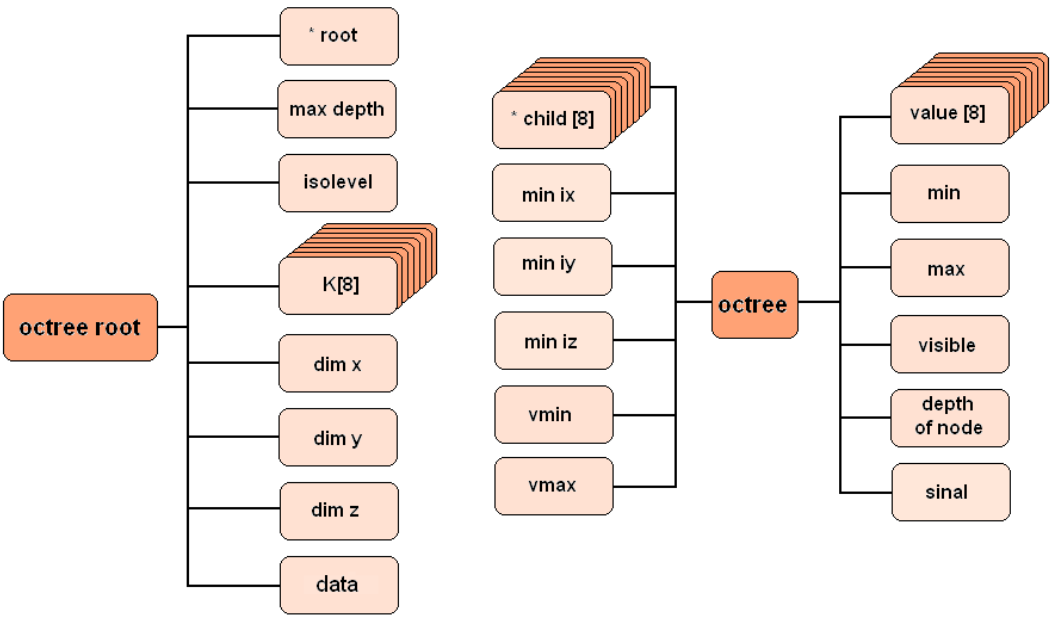

Figura 4.1: Organização das classes Octree e OctreeRoot.

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.

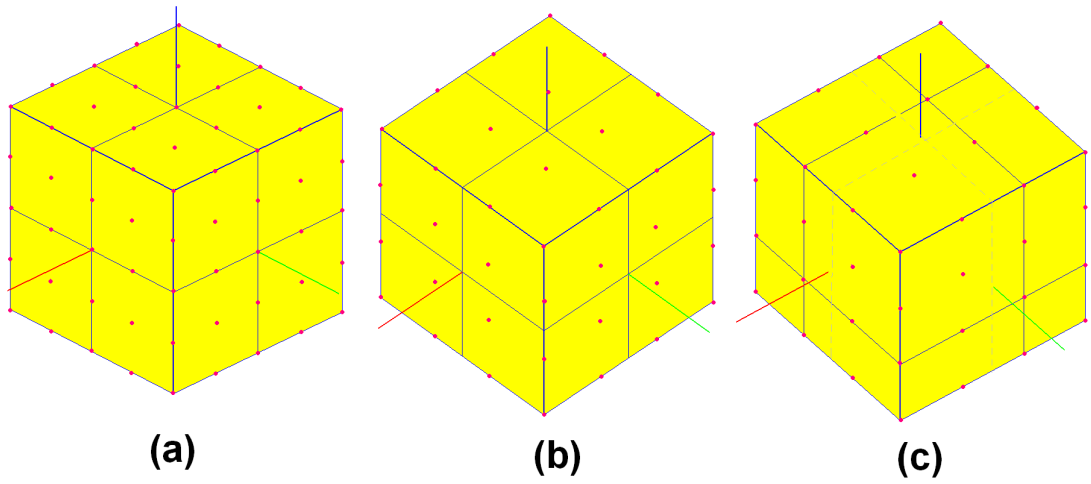

Figura 4.2: (a) vértice do voxel coincidente com a subdivisão do grid. (b) vértice do voxel não coincidente com a subdivisão do grid. (c) correção do nó da octree de modo a coincidir com o grid.

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

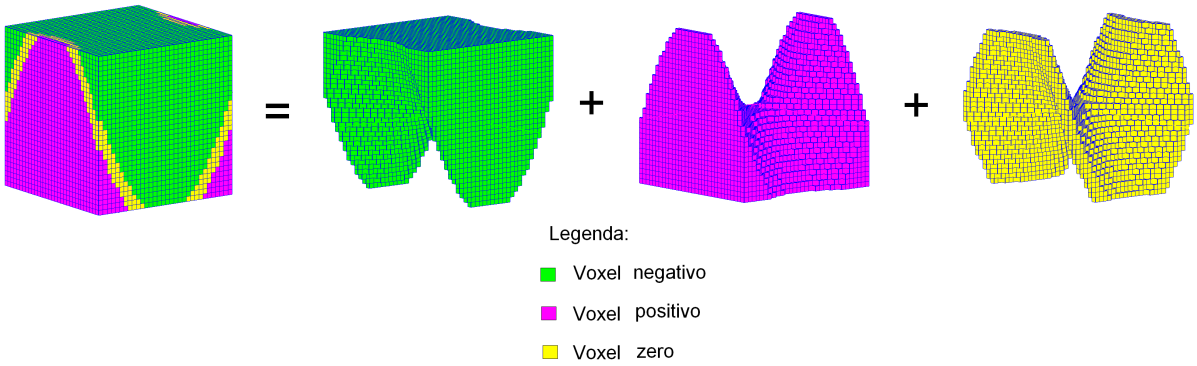

Figura 4.3: Octree: composta de nós positivos, negativos, e nós zero (onde há mudança de sinal).

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.

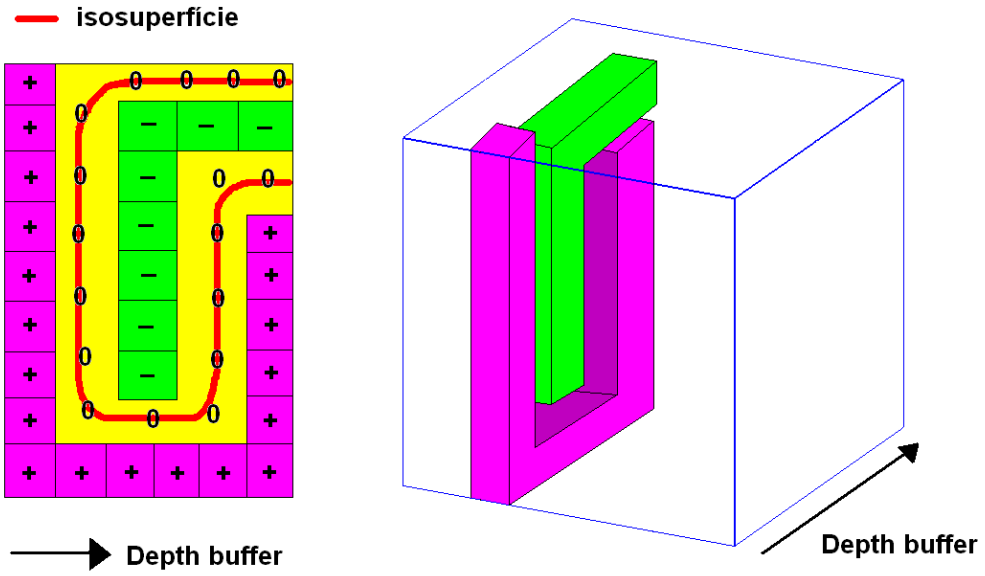

Figura 4.4: Disposição dos nós no depth buffer.

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).

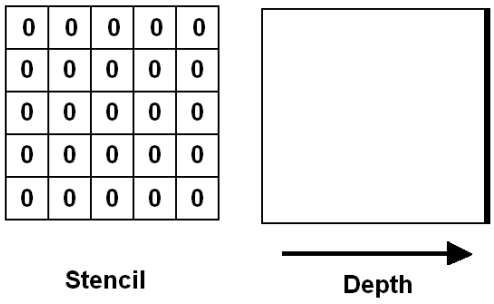

Figura 4.5: Inicializamos o stencil buffer com 0 e o depth buffer no infinito, ou seja, com valor 1.

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

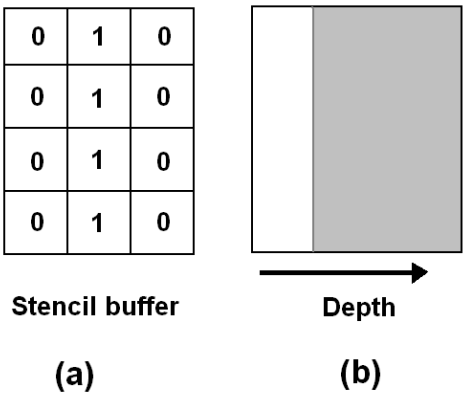

Figura 4.6: Inicializamos o stencil buffer com 0 e o depth buffer no infinito, ou seja, com valor 1.

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:

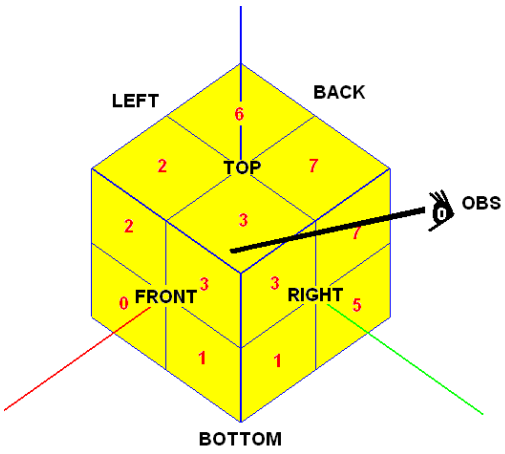

Figura 4.7: Ordenação dos nós octree.

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).

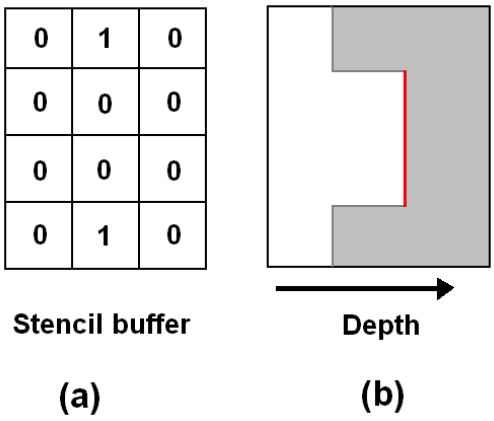

Figura 4.8: Renderiza negativo

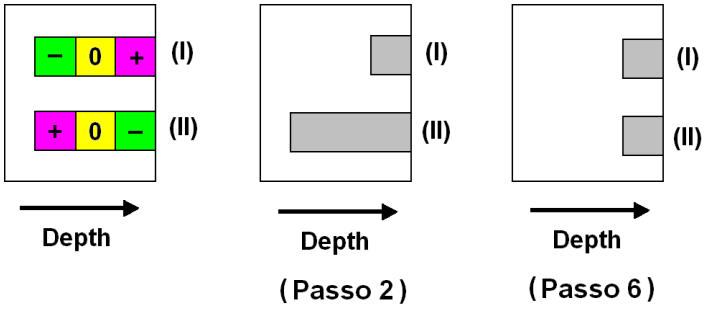

Figura 4.9: Passo 2: Após a renderização dos nós positivos. Passo 6: Após a renderização dos nós negativos.

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:

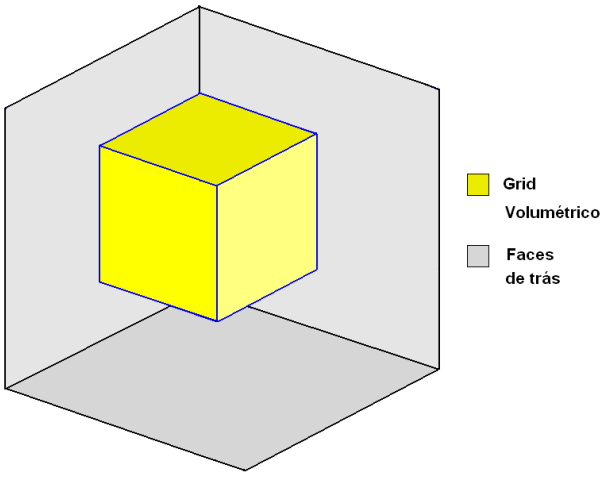

Figura 4.10: Correção da profundidade dos pixels através da renderização de 3 faces de um cubo posicionado atrás do grid volumétrico.

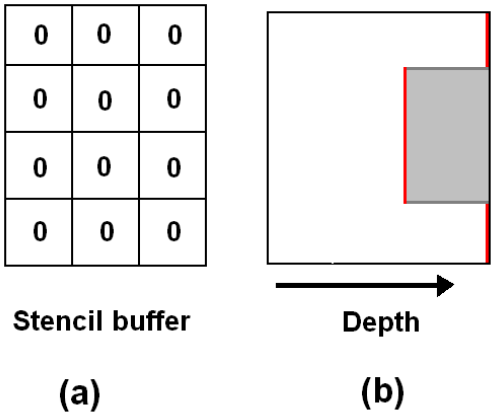

Figura 4.11: Mapa de oclusão implícito.

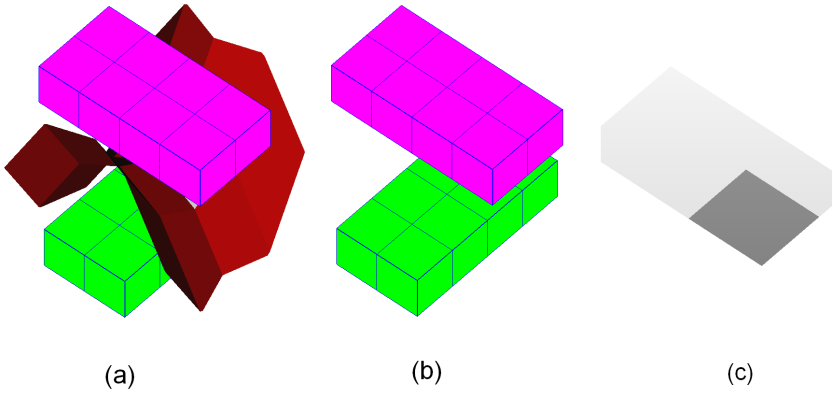

Figura 4.12: Parabolóide hiperbólico numa octree de nível 2: (a) Nós positivos, negativos e isosuperfície (b) Apenas os nós positivos e negativos (c) região de oclusão.

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)

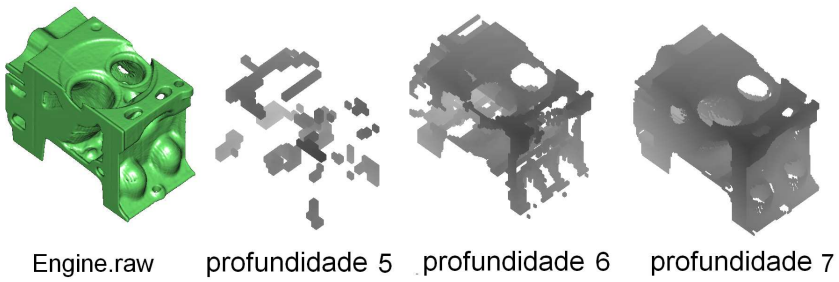

Figura 4.13: Regiões de oclusão do dado volumétrico Engine selecionando-se octrees com profundidades 5, 6 e 7.

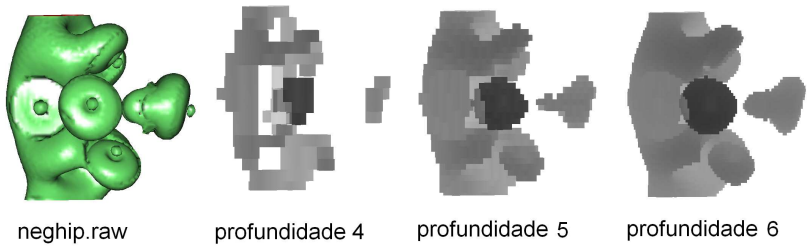

Figura 4.14: Regiões de oclusão do dado volumétrico Neghip selecionando-se octrees com profundidades 5, 6 e 7.

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

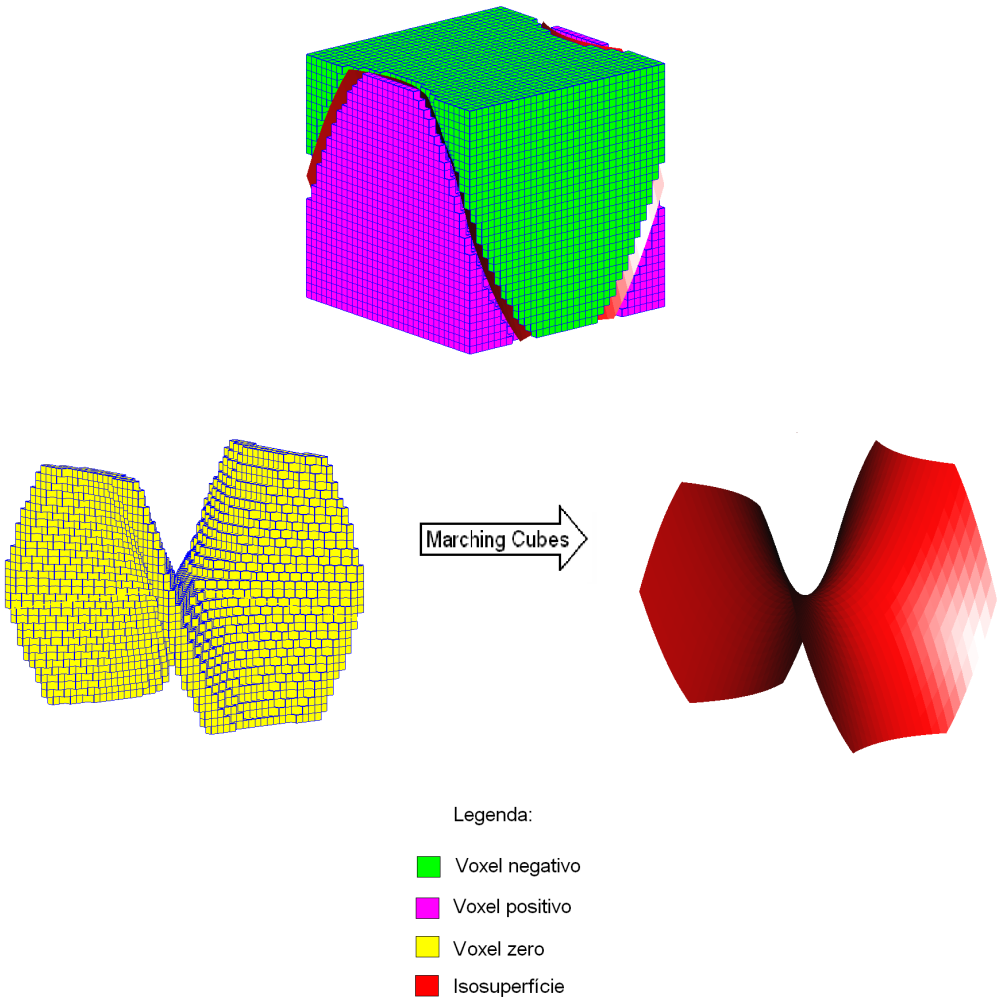

Figura 4.15: Aplicação do algoritmo de Marching Cubes apenas nos nós onde há mudança de sinal e que são visíveis.

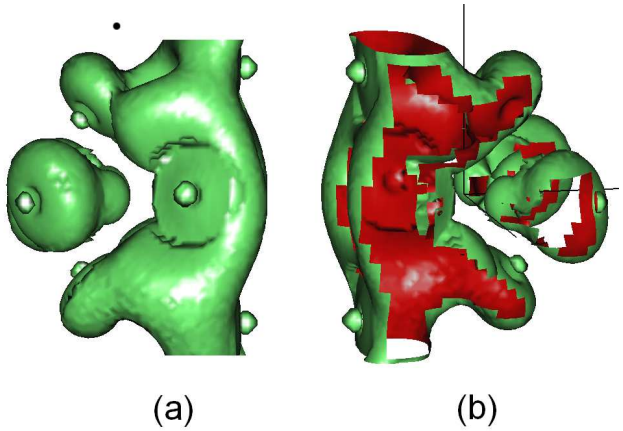

Figura 4.16: Dado volumétrico Neghip.raw: (a) isosuperfície renderizada com occluder (b) Teste de visibilidade interrompido para mostrar quantas faces deixaram de ser renderizadas aplicando-se a oclusão implícita.

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