Daniel Luiz Alves Madeira

Uma Estrutura Baseada em Hash Table para Buscas Otimizadas em Octree em GPU

Niterói

02 de Março de 2010

Daniel Luiz Alves Madeira

Uma Estrutura Baseada em Hash Table para Buscas Otimizadas em Octree em GPU

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Mestre. Área de Concentração: Computação Visual e Interfaces.

Orientador:
Esteban Walter Gonzales Clua

Universidade Federal Fluminense

Niterói

02 de Março de 2010

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Mestre. Área de concentração: Computação Visual e Interfaces

Aprovada por:

Prof. Dr. Esteban Walter Gonzales Clua
Orientador

Prof. Dr. Anselmo Antunes Montenegro
Universidade Federal Fluminense

Prof. Dr. Thomas Lewiner
Pontifícia Universidade Católica do Rio de Janeiro

”Se a princípio a idéia não é absurda, então não há esperança para ela”

Albert Einstein

Aos meus familiares e amigos, base de tudo o que eu faço.

Agradecimentos

À Deus, por me dar forças para perseguir meu objetivos.

Aos meus pais, pelo apoio incondicional em todos esses meus anos de vida.

Aos meus irmãos Rafael e Marcell, e à toda a minha família, por toda a amizade, companheirismo e diversão durante todos esses anos.

À Marcelle, que se tornou tão especial desde que nos conhecemos, pelos puxões de orelha para que eu nunca deixasse de me esforçar. "Desistir não é nobre. E arduamente, não desistimos."

Também a todos os grandes amigos que fiz na UFJF, que sempre terão lugar reservado, mesmo quando a distância é grande e poucas são as conversas: Sara, Davi, Noara, Douglas, Crys, Marco, Marcelo Caniato, Renata, Luiz Guilherme.

Ao Instituto de Computação da UFF e todos seus professores e funcionários, pela oportunidade de desenvolver meus estudos.

Ao professor Esteban, pela orientação e ajuda durante todo o mestrado. Também aos professores Anselmo e Thomas, pela grande ajuda durante este trabalho.

Ao pessoal do Medialab, pelo excelente ambiente de trabalho. Farei jus ao broche!

Aos meus antigos companheiros de república em Niterói, pelos bons momentos de confraternização.

À todas as pessoas cujos nomes não estão aqui, mas que fizeram parte desta caminhada.

À CAPES pela ajuda financeira, sem a qual não poderia ter terminado este trabalho.

Resumo

Estruturas de octree são amplamente utilizadas em aplicações gráficas para acelerar a computação de relações geométricas de proximidade. Com o aumento do poder do hardware gráfico, tarefas de processamento estão sendo progressivamente portadas para estas arquiteturas. Todavia, octrees são essencialmente estruturas hierárquicas, e buscas em octrees são processos sequenciais, não ideais para uma implementação na GPU. De um lado, várias estratégias foram propostas para a estrutura de octree na GPU, a maioria utilizando buscas hierárquicas. Por outro lado, trabalhos recentes introduziram buscas otimizadas, que evitam travessias hierárquicas na estrutura.

Neste trabalho, é proposta uma octree em GPU que permite as buscas otimizadas que utiliza o streaming da GPU para buscar um grande conjunto de pontos em paralelo. Também propõe-se uma paralelização da busca otimizada para melhorar o tempo de busca de um único ponto.

Finalmente, a estrutura proposta se utuliza das recentes arquiteturas de hardware para otimizar a estrutura da octree na GPU.

Palavras-chave: *Estrutura de dados*; *Árvore Octária*; *Busca Geométrica*; *Busca Otimizada*; *GPU*

Abstract

Octree structures are widely used in graphic applications to accelerate the computation of geometric proximity relations. With the increasing power of graphics hardware, processing tasks are progressively ported of to those architectures. However, octrees are essentially hierarchical structures, and octree searches are mainly sequential processes, which is not suited for GPU implementation. On one side, several strategies have been proposed for GPU octree data structure, most of them use hierarchical searches. On the other side, recent works introduced optimized searches which avoid hierarchical traversals.

In this work, we propose a GPU octree that allows for those optimized searches, which uses the GPU streaming to search for large set of points at once. Moreover, we propose a parallelization of those optimized search to speed up the single point search.

Finally, the proposed structure takes advantage of the recent graphics hardware architectures to improve the GPU octree data structure.

Keywords: *Data Structure*; *Octree*; *Geometric Search*; *Optimized Search*; *GPU*

Sumário

Lista de Figuras

1 Introdução

Uma grande quantidade de algoritmos gráficos dependem da proximidade espacial dos dados: detecção de colisão, aproximação de superfícies geométricas, simulação de fluidos baseadas em partículas, renderização baseada em raios entre outros. Realizar a busca com força bruta leva o algoritmo à uma complexidade quadrática $O\left(n^2\right)$, o que é proibitivo para grandes volumes de dados. Então, várias otimizações foram propostas para acelerar tais buscas, a maioria baseada em divisão-e-conquista: dividir o espaço em blocos menores que podem ser processados independentemente. Esta divisão precisa ser armazenada em memória, o que leva a um *trade-off* entre consumo de memória e aceleração do procedimento de busca (2).

Entre estas, a octree é uma das mais difundidas, sendo utilizada em áreas como processamento de imagens (3), modelagem geométrica (4, 5), imagens médicas (6), detecção de colisão (7), renderização baseada em pontos (8), visualização de isosuperfícies (9) e modelagem volumétrica (10), entre vários outros campos.

Com o poder de processamento crescente do *hardware* (GPU), muitas aplicações gráficas e não-gráficas foram portadas para esta arquitetura paralela, tarefa esta delicada, porém produtiva (11, 12).

A situação ideal seria manipular todo o estágio de aplicação, além de todos os estágios de geometria e rasterização na GPU. Porém, várias partes de uma aplicação típica, como interação com o usuário ou travessias puramente hierárquicas em árvores, são essencialmente sequenciais, exigindo uma implementação em CPU.

Isto provoca uma contínua comunicação entre CPU e GPU, o que constitui um gargalo, principalmente quando uma estrutura de dados precisa ser passada entre estes dois dispositivos.

Em particular, em muitas aplicações de octrees, que utilizam buscas hierárquicas (divisão-e-conquista), as relações de proximidade são mantidas em CPU. Este trabalho propõe uma estrutura de dados em GPU para representação de uma octree que permite buscas otimizadas (13). Esta estrutura é totalmente mantida em GPU, reduzindo o tráfego de dados entre CPU e GPU.

1.1 Contribuições

A principal contribuição deste trabalho é uma nova estrutura de Octree adaptada para a GPU que permite buscas otimizadas. As buscas otimizadas evitam a travessia hierárquica da Octree, otimizando o tempo total da busca. Para isso, a estrutura da Octree deve permitir acesso em tempo constante a qualquer um de seus nós, motivo pelo qual a Octree foi implementada como uma tabela hash. Além disso, o nível do ponto a ser buscado deve ser inferido. Dessa maneira, uma otimização estatística é utilizada para inferir com maior precisão este nível.

1.2 Motivação

Octrees são amplamente utilizadas em várias áreas. Um exemplo é em algoritmos de recorte de superfícies, utilizados em aplicações como ray-tracing, entre outras. Ao adaptar uma Octree ao modelo, através de buscas simples e da travessia da Octree, podemos eliminar uma grande quantidade de nós que não interessam (por estar fora do campo de visão, por exemplo). Em algoritmos de detecção de colisão, a Octree auxilia permitindo que se busque com maior facilidade as faces do modelo que estão passíveis de uma colisão com algum outro objeto. Assim, ao invés de fazer testes para todas as faces, pode-se eliminar grande parte das faces, tornando o algoritmo mais eficiente.

1.3 Estrutura

Este trabalho está dividido da seguinte maneira: no capítulo 2 é realizada a revisão bibliográfica do tema. Conceitos e estruturas de Octrees e tabelas hash são apresentados. No mesmo capítulo, os métodos de buscas são apresentados. No capítulo 3 é apresentada a estrutura criada para este trabalho e os algoritmos de buscas desenvolvidos. No capítulo 4 é apresentada a implementação da hashed octree e dos algoritmos de busca citados no capítulo anterior. O capítulo 5 traz os testes e seus resultados e, por fim, o capítulo 6 traz as conclusões e sugestões de trabalhos futuros.

2 Revisão Bibliográfica

2.1 Octrees

Uma octree é uma estrutura de dados hierárquica baseada na decomposição de uma região 3D em sub-espaços, na forma de cubos. Enquanto a raiz desta árvore representa todo o espaço, cada nó da árvore representa um sub-espaço da região, na forma de um sub-cubo. O cubo de um nó não-folha pode ser dividido em 8 octantes, gerando assim 8 filhos. Os dados da geometria sendo representada geralmente são armazenados somente nas folhas. Consideramos aqui as octrees completas, onde cada nó pode então ou ser um nó-folha ou possuir 8 filhos (2).

Uma octree é uma árvore, onde cada nó que não é folha possui interligação com mais outros oitos nós da estrutura de dados, hierarquicamente abaixo do mesmo. Existem várias maneiras de implementar tal interligação de nós, sendo a mais comum através de ponteiros. Este capítulo apresenta as representações mais comuns de uma octree. Neste trabalho se utiliza uma árvore de resolução variável, onde nós-folha existem em vários níveis da árvore, representada por uma tabela hash para permitir buscas otimizadas (13).

Quando se trata de Octree sempre refere-se ao espaço tridimensional. Enquanto as Octrees podem se comportar adequadamente para uma divisão de um espaço 3D, existe uma variação para o espaço 2D (imagens e planos). Esta variação é denominada Quadtree.

2.1.1 Construção da Octree

Para se construir uma octree, deve-se inicialmente definir uma bounding box, ou caixa envolvente, que englobe o espaço contido na Octree. Dentro dos limites desta caixa está a primitiva ou cenário 3D que se deseja representar. O espaço é então dividido em oito partes iguais, verificando-se em seguida se existe intersecção desta primitiva ou cenário com cada um dos hexaedros resultantes da divisão.

Caso não ocorra intersecção o cubo resultante permanece em branco ou vazio. Se houver intersecção há duas possibilidades: caso o hexaedro esteja completamente inserido na primitiva ou cenário, marca-se o hexaedro para que seja exibido na tela; caso o hexaedro esteja parcialmente inserido na primitiva, divide-se este hexaedro em mais oito partes menores iguais, repetindo-se então o algoritmo em questão. Estes passos se repetem até que a profundidade máxima da árvore seja alcançada ou a primitiva ou cenário sejam representados por completo.

O algoritmo a seguir mostra a criação de uma octree.

```
Procedimento CriaArvore
  Entrada: Solido, Octree *raiz, nivel
  C ← Classifica(Solido, raiz →min, raiz →max)
  se C = CHEIO então
  |  raiz ←CHEIO
  fim
  senão se C = VAZIO então
  |  raiz ←VAZIO
  fim
  senão se C = PARCIAL então
  |  raiz ←CHEIO
  |  subdivide(raiz);
  |  para cada filho F de raiz faça
  |  |  CriaArvore(Solido, F, nivel + 1)
  |  fim
  fim
```

Figura 1: Procedimento de criação da Octree

A figura Figura 2 ilustra o processo. No primeiro nível, tem-se o espaço inicial. No segundo nível, o espaço inicial é dividido em 8 octantes. No terceiro nível, um dos octantes

é novamente subdividido. A figura 3 exemplifica a adaptação da Octree a um modelo

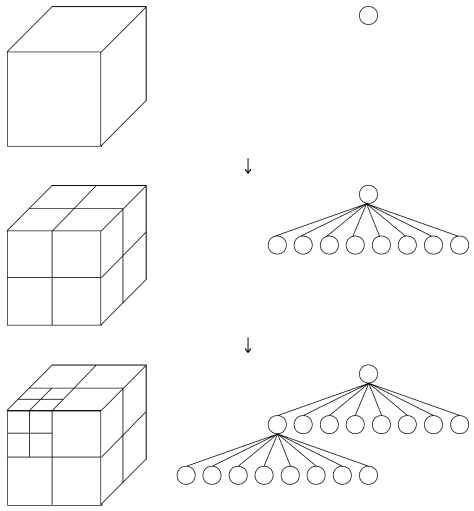

Figura 2: Construção de uma Octree. O espaço é sempre subdividido em 8 partes.

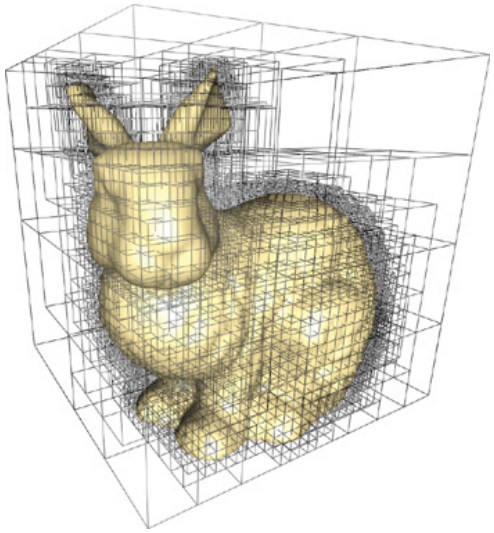

Figura 3: Exemplo de uma Octree adaptada a um modelo. Imagem retirada de (1).

2.1.2 Principais Representações

Existem diversas estruturas de dados capazes de representar uma octree. A diferença entre estas está no modo que as interligações entre os nós serão definidas, podendo ser através de ponteiros, vetores lineares ou mesmo tabelas hash, cada um com suas vantagens e desvantagens. A implementação através da tabela hash foi utilizada neste trabalho.

Octree por ponteiros

Nesta representação, cada nó não-folha possui 8 ponteiros para seus 8 respectivos filhos e cada nó-folha não possui ponteiros. Assim, a travessia da árvore se dá somente pela travessia dos ponteiros, gerando uma maneira recursiva e simples de percorrer toda a árvore. Dentro desta categoria, duas representações são mais comuns: a pai-filho e a filho-irmão.

A representação pai-filho é a mais clássica, sendo que cada nó armazena 8 ponteiros, um para cada filho, além dos seus próprios dados. Nos nós-folha, os ponteiros são nulos, enquanto nos nós-intermediários, os dados são nulos (2). Um ponteiro do filho para o pai também pode ser adicionado na estrutura de dados, de forma a facilitar a travessia das folhas para a raiz (Figura 4).

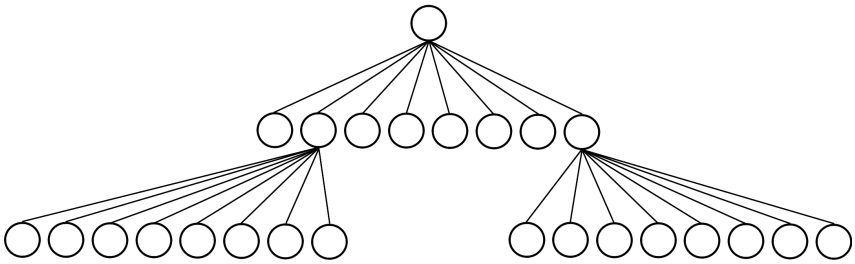

Figura 4: Representação pai-filho de uma octree.

Na representação filho-irmão, cada nó possui apenas um ponteiro para o primeiro dos seus filhos e para o irmão à direita (Figura 5). Nesta representação, a busca pelos filhos de um determinado nó é facilitada, porém o tempo de travessia total é maior.

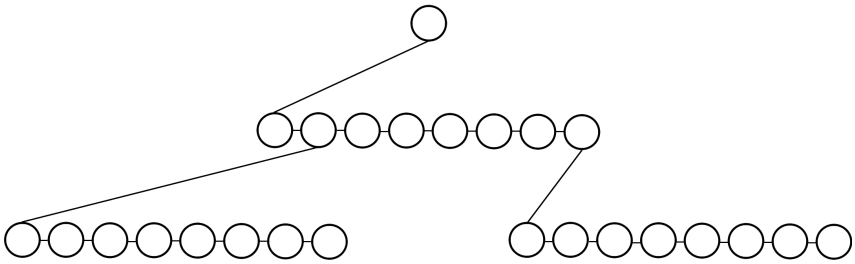

Figura 5: Representação filho-irmão de uma octree.

Octree linear

Esta representação utiliza somente um vetor para armazenar a octree. Cada nó possui oito índices, indicando a localização dos seus filhos no vetor, eliminando o uso de ponteiros. Os índices são calculados automaticamente por $(8 * indicePai + offsetFilho)$ para cada filho. Desta forma é possivel que o acesso para cada nó da octree seja feito em tempo constante.

Para calcular o tamanho necessário na criação do vetor é necessário conhecer o nível máximo da octree. Esta solução é eficiente se a octree for cheia, ou seja, todos os seus nós folhas estão no mesmo nível da árvore. Caso a octree possua resolução variável o vetor terá muitas posições vazias, gerando assim um desperdício grande de memória.

Hashed Octree

É possível trocar os ponteiros por manipulaçao de índices, mesmo em octrees com resolução variável. Neste caso, a referência aos filhos devem ser substituídas por um cálculo a partir do índice do pai e os nós armazenados em uma tabela de índices (Figura 6).

Esta estrutura é mais compacta que uma octree linear, porém, como depende do cálculo do índice do filho, pode gerar tempos de busca maior que as octrees por ponteiros, que só dependem da desreferenciação destes. Todavia, permitem acesso direto em tempo constante a qualquer nó da octree, dado seu índice, enquanto as octrees baseadas em ponteiros só permitem acesso direto à raiz.

O índice pode ser gerado *ad hoc* para uma octree estática com o objetivo de reduzir a tabela de índices, como proposto na GPU via hash perfeito (Seção 2.2.1). Todavia, qualquer mudança significativa na estrutura da octree implica um completo recálculo dos índices, e o cálculo dos índices dos filhos é substituído por mais espaço de memória.

O índice também pode ser gerado sistematicamente a partir da posição geométrica do nó ou pela posição do nó na hierarquia da octree. O desempenho na recuperação de

000	001	010	011	100	101	110	111
10000	1	10010	10011	100	101	110	111
1111000	10001	1111010	1111011	1001100	1001101	1001110	1001111
	1111001			11100	11101	11110	11111

Figura 6: Representação de uma quadtree via hash. A função de hash utiliza os 3 últimos bits de seus códigos (linha superior) para agrupar os nós da árvore (linhas inferiores).

dados numa hashed octree, assim como uma tabela hash comum, depende fortemente da qualidade da função de hash utilizada. No capítulo 2.2 serão apresentadas estratégias de implementação da tabela hash e a função de hash utilizada neste trabalho.

Branch-on-need Octree

Esta estrutura, proposta por Wilhelms e Gelder em (14), é uma representação por ponteiros incompleta que subdivide o espaço não uniformemente. O objetivo é criar uma árvore com o mínimo de subárvores vazias nos nós internos. Dessa maneira, diminui-se o número de nós na árvore e economiza-se espaço. Porém, como se divide o espaço de uma maneira não uniforme, múltiplos pais para um filho podem surgir em níveis mais baixos, tornando a travessia da árvore um pouco mais complexa. Além disso, um sistema de hash baseado somente nos sistemas de coordenadas não é aplicável a este método.

Octree em texturas

Representar as octrees através de texturas foi a primeira adaptação para a implementação das octrees em GPUs (15) e (1). Quando as primeiras GPUs programáveis foram lançadas, apenas era possível usar seu recurso computacional por meio dos processadores de shaders (pixel e vertex). Entretanto, para tal tarefa era necessário transformar qualquer espaço de problema em uma imagem, passo muitas vezes não-trivial. Somente

após este passo seria possível processá-lo e depois convertê-lo novamente para o espaço do problema original.

Para facilitar esta codificação e também a implementação das octrees na GPU, Lefohn et al. em (16) propuseram uma biblioteca de templates, como a biblioteca STL do C++. As novas arquiteturas de GPUs, como o CUDA e o FireStream, permitem a eliminação desta conversão para textura, tornando o seu uso mais facilitado para estes fins.

2.2 Tabelas Hash em Octrees

Uma tabela hash é uma estrutura de dados que utiliza uma função de dispersão para eficientemente mapear identificadores ou chaves em valores associados. A função de dispersão é usada para transformar a chave no índice da tabela, onde o dado final é recuperado.

O desempenho desta estrutura depende da função de dispersão (função hash) utilizada. Esta função deve, idealmente, fornecer um mapeamento de 1 para 1, onde cada chave é mapeada em somente um índice, e cada índice é associado a somente uma chave. Tais funções são chamadas de hashs perfeitos. É uma situação rara, o que em geral obriga a tratar as colisões na tabela, que consistem em situações onde duas ou mais chaves são mapeadas em um mesmo índice.

Numa tabela hash bem dimensionada, o custo médio de buscas de chaves é independente do número de elementos armazenados ($O(1)$). Mais ainda: as inserções e remoções também podem ser feitas em tempo médio constante.

2.2.1 Hash Perfeito

Uma função de hash perfeita (17, 18) cria um mapeamento único de chaves, sem nenhuma colisão. Implementar uma função deste tipo geralmente implica em conhecer previamente todas as chaves possíveis do problema. Uma funcão de hash perfeita f para um determinado conjunto de chaves K é definida formalmente por (19):

$$\forall j,k \in Kf(j) = j(k) \rightarrow j = k \tag{2.1}$$

A utilização de funções de hash perfeitas é recomendada para grandes volumes de dados que são pouco atualizados e acessados com bastante frequência, pois uma inserção de chave na tabela pode implicar no recálculo da função de hash perfeita, o que é geralmente demorado.

2.2.2 Tratamento de colisões

Quando a função de hash utilizada não fornece um mapeamento perfeito, diferentes chaves podem ser mapeadas para o mesmo índice, o que provoca colisões. Diferentes métodos de tratamento de colisões serão apresentados nesta seção.

Encadeamento Aberto

No encadeamento aberto, as colisões são resolvidas armazenando as chaves que se colidiram numa lista. Em cada índice do vetor da tabela hash há uma lista de chaves colididas. Ao inserir uma nova chave, o algoritmo aplica a função de hash para calcular o índice e caso haja colisão, o insere no final desta lista. A busca então envolve calcular o índice através da função hash e então iterar sobre esta lista, até encontrar a chave procurada.

Caso a função de hash não seja eficiente e mapeie muitas chaves para um mesmo índice, observa-se uma grande perda de performance, pois o tempo de busca depende diretamente do tamanho destas listas. No pior caso, todas as chaves são mapeadas para um único índice e o tempo de busca sobe para $O(N)$, onde N é a quantidade de chaves mapeadas.

Em CPU, estas listas são facilmente implementadas como listas encadeadas. Na GPU, não se pode trabalhar com ponteiros, proibindo então o uso de listas encadeadas. A alternativa é utilizar listas sequenciais, pré-alocadas, mas para isso deve-se em todas as

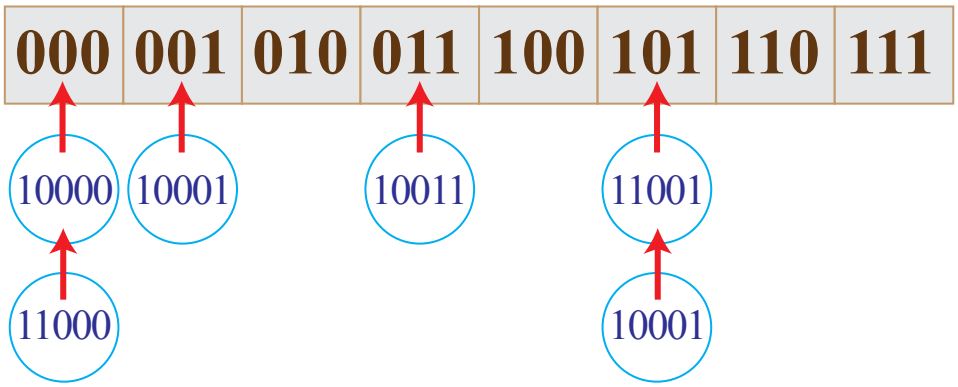

Figura 7: Encadeamento aberto: chaves colididas são armazenadas numa lista na posição do seu índice.

posições da tabela hash alocar uma lista grande o suficiente para absorver todas as colisões ocorridas, o que facilmente leva a um grande desperdício de memória, que é bem limitada na GPU.

Encadeamento Fechado

No encadeamento fechado, quando há uma colisão, uma segunda função de hash é utilizada para calcular uma nova posição para a chave. Esta operação é chamada de *rehashing*. Esta segunda função $c(h)$ deve ser escolhida sistematicamente para assegurar que será possível recuperar todos os outros nós. Buscar por uma chave n se resume a olhar diretamente a posição mapeada pela função de hash. Se a posição estiver vazia, a chave n não pertence à tabela hash. Se a posição não estiver vazia e a chave contida ali não for a chave procurada, é aplicada a segunda função de hash e a busca é refeita com o novo índice gerado (Figura 8).

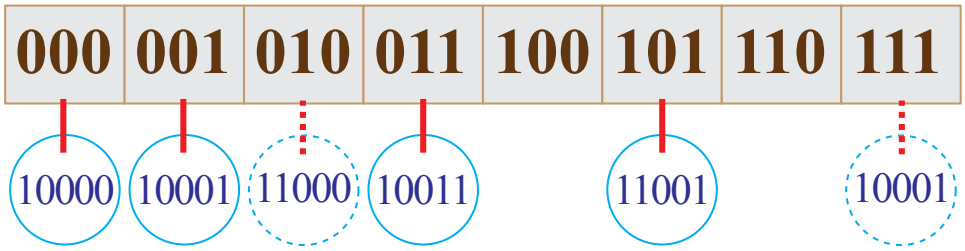

Figura 8: Encadeamento fechado: uma chave colidida é enviada para uma nova posição. Neste exemplo, é realizado um *shift* de 2 bits para calcular a nova posição.

Neste trabalho foi utilizado o encadeamento fechado para resolver as colisões. As

particularidades da implementação em GPU serão tratadas no capítulo 4.

2.2.3 Codificação das chaves

A tabela hash utiliza uma função para codificar uma chave em um índice do vetor que contem os dados. Castro utiliza o código de Morton para tal codificação. O código de Morton (21), ou curva de ordem-Z, é amplamente utilizado para transformar coordenadas bidimensionais ou de mais dimensões em códigos de uma única dimensão. Uma vez que os códigos foram calculados, percorrê-los em ordem crescente equivale a uma busca em profundidade numa árvore, evidenciando que o código mantém a ordem hierárquica da Octree. Assim, este código é muito utilizado em quadtrees e octrees. Neste trabalho implementa-se esta estratégia para a etapa de busca da Octree.

O índice pode ser calculado diretamente a partir da hierarquia da árvore, recursivamente. Para tanto, caminha-se pela árvore (Figura 4). O índice da raiz é sempre 1, e o índice de cada filho é a concatenação do índice do seu pai com a direção de seu octante, codificado em 3 bits. A travessia inversa da árvore também é permitida, já que para encontrar o índice do pai basta truncar os últimos 3 bits do índice do filho.

O índice pode ser equivalentemente computado a partir da posição geométrica do cubo contido no nó e pelo seu tamanho. Considerando que o cubo da raiz é único e denotando por (x,y,z) as coordenadas do centro do cubo e por 2^{-l} o tamanho do cubo, o código de Morton pode ser gerado por:

$$1x_l y_l z_l x_{l-1} y_{l-1} z_{l-1} x_{l-2} y_{l-2} z_{l-2} \ldots x_1 y_1 z_1 \ , \tag{2.2}$$

onde $x_l x_{l-1} \ldots x_1$ é a decomposição binária de $\lfloor 2^l x \rfloor$. A figura Figura 9 exemplifica esta operação.

A geração deste índice pode ser acelerada utilizando dilatação e contração de inteiros (22). Este processo auxilia a concatenação dos bits, alterando os bits através de

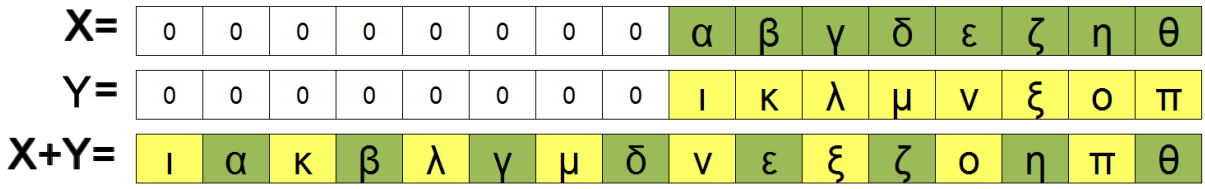

Figura 9: Geração do código de Morton. Os bits de X e Y são concatenados um a um, gerando o código final.

máscaras, até que cada bit original do número esteja separado por 0. A figura Figura 10 exemplifica a dilatação de um número inteiro.

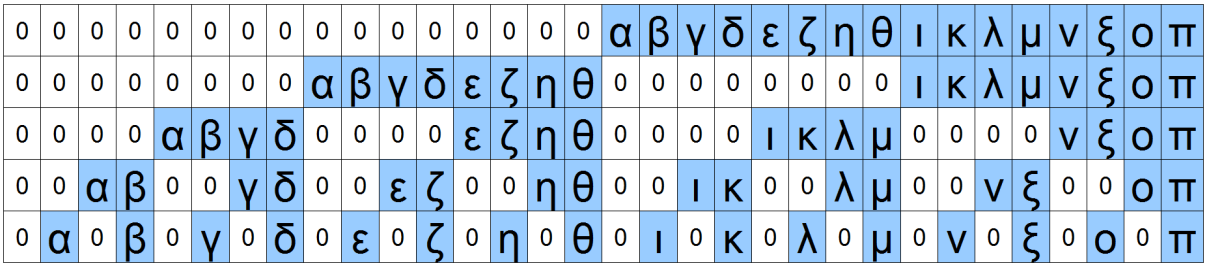

Figura 10: Passoa da dilatação de um número inteiro de 16 bits. Os bits são separados e bits 0 são inseridos entre eles.

2.3 Buscas

Como definido por Samet, uma $2^d - tree$ que permite acesso em tempo constante aos seus nós é chamada de $2^d - tree$ *aberta*. Gargantini definiu em (23) uma quadtree linear, onde os nós folhas são armazenados em um vetor ordenado. Glassner em (24) e depois Warren e Salmon em (25) desenvolveram otimizações sobre esta estrutura. O primeiro utilizou uma tabela hash para armazenar os nós folhas e o segundo trabalho paralelizou uma octree utilizando uma tabela hash paralela. Este capítulo apresenta os algoritmos clássicos de busca em octrees, bem como a otimização estatística proposta por Castro em (20).

2.3.1 Busca clássica

A forma clássica de se realizar buscas numa octree envolve percorrer toda a hierarquia da árvore, a partir da raiz, até encontrar o nó desejado. A partir da raiz, pergunta-se a cada filho se o valor pesquisado faz parte de sua descendência. Caso positivo, repete-se a mesma verificação aos filhos daquele nó, recursivamente, até que se atinja um nó folha, onde geralmente está o valor procurado. Caso o valor não exista na octree, é retornado um ponteiro vazio. O algoritmo na figura 11 ilustra este procedimento.

```
Função BuscaValor
  Entrada: Valor v, Octree *raiz
  Saída: Nó da Octree contendo v
  se v ∉ raiz então
  |  retorna NULL
  fim
  senão se v ∈ raiz e raiz == FOLHA então
  |  retorna raiz
  fim
  senão
  |  para cada filho F de raiz faça
  |  |  BuscaValor(v, F)
  |  fim
  fim
```

Figura 11: Procedimento clássico de busca na Octree

2.3.2 Busca otimizada

O algoritmo de busca direta proposto por Castro utiliza a seguinte idéia: ao buscar um ponto p qualquer, é gerada a chave $k(p)$ de p para um nível estimado l. Assim, três situações podem ocorrer ao acessar a entrada correspondente na tabela hash:

1. O nó $n_l(p)$ é folha e é retornado pelo algoritmo.

2. O nó $n_l(p)$ não é folha. Isto implica que o nível estimado é pouco profundo e l é incrementado até $n_l(p)$ ser uma folha.

3. O nó $n_l(p)$ não exite. Isto implica que o nível estimado é muito profundo e l é decrementado até $n_l(p)$ ser uma folha.

O algoritmo da busca otimizada (Figura 12) estima um nível $\hat{l}$. Com este nível, é gerado o código de Morton para o ponto que está sendo buscado. Em seguida, dois loops executam as condições 2 e 3 do algoritmo, testando iterativamente se o código de Morton não existe ou se ele existe, porém sem pertencer a um nó-folha. Ao final destes dois loops, o último nó n válido é retornado.

Procedimento BuscaOtimizada
Entrada: Ponto p
Saída: Nó n
computa código de Morton m_{max} de p na tabela hash
computa código de Morton m no nível $l = \hat{l}$ a partir de m_{max}
acessa o nó n correspondente a m na tabela hash
// Caso 1 → n existe na tabela hash
enquanto *n existe na tabela hash* **faça**
  incrementa a profundidade l, adicionando em m 3 bits de m_{max}
  acessa o nó n correspondente a m na tabela hash
fim
// Caso 2 → n não existe na tabela hash
enquanto *n não existe na tabela hash* **faça**
  decrementa a profundidade l, retirando 3 bits de m
  acessa o nó n correspondente a m na tabela hash
fim
retorna *último nó n válido*

Figura 12: Procedimento de busca otimizada

Se o nível estimado for 0, a busca se resume à busca clássica, iniciando-se da raiz. Além disso, a chave não precisa ser recalculada a cada passo, pois pode ser deduzida da chave anterior. Este é o pior caso do algoritmo, pois sabe-se que os nós folhas são os mais distantes da raiz.

2.3.3 Otimização estatística

O algoritmo da seção anterior utiliza a posição do ponto e um nível estimado $\hat{l}$ para compor a chave e realizar a busca na tabela hash. Porém a única informação que se possui

é o ponto que se deseja procurar. Falta ainda criar uma estimativa para o nível da árvore. Este nível estimado indica em qual nível há uma folha que contenha o ponto procurado.

De acordo com Castro, o custo médio de busca depende do número de vezes em que, na média, cada folha da árvore foi procurada. Supuseram que este custo está relacionado somente ao nível da árvore. Assim, as buscas na tabela hash não tem nenhuma relação com a geometria do objeto.

O custo médio de busca é então dado por:

$$custo\left(\hat{l}\right) = \sum_{l} p_l \times f_l \times custo_{\hat{l}}(l) \tag{2.3}$$

onde p_l é o número de folhas num determinado nível l, f_l é o número de vezes em que cada folha é procurada no nível l e $custo_{\hat{l}}(l)$ é o custo da busca no nível l.

Na busca otimizada, existem três casos para analisar: o caso onde a folha é diretamente encontrada, o caso onde o nível estimado foi pouco profundo e o caso onde o nível estimado foi muito profundo.

No primeiro caso, a chave é gerada em tempo constante e o nível foi estimado corretamente, ou seja $\hat{l} = l$. Neste caso, a busca é realizada em tempo constante c. Já nos casos onde o nível não foi estimado corretamente, são geradas $|l - \hat{l}|$ novas chaves, ao adaptar o nível estimado até o nível correto da árvore. Estas chaves também são geradas em tempo constante. Assim, o custo da busca é dado pela soma da constante c mais a sobrecarga da diferença $|l - \hat{l}|$.

Para otimizar o custo então, é preciso encontrar uma função que minimize o mesmo. Castro em (20) mostra que a função que minimiza este custo é uma função mediana dos níveis das folhas ponderada pelo número de vezes em que são procuradas. Considera-se $p(l)$ como uma função de probabilidade para p_l e $f(l)$ como uma função de probabilidade para f_l. A construção de $p(l)$ é feita definindo-a por partes em cada intervalo inteiro $[l, l+1]$, da seguinte forma: cada parte é uma parábola de área $p(l)$ cancelando nas

bordas do intervalo. Fazendo uma construção similar para $f(l)$, pode-se escrever a função custo em sua forma contínua:

$$custo\left(\hat{l}\right) = c + \int_0^{\hat{l}} p(l) \cdot f(l) \cdot \left(\hat{l} - l\right) dl + \int_{\hat{l}}^{\infty} p(l) \cdot f(l) \cdot \left(\hat{l} - l\right) dl \tag{2.4}$$

Derivando a equação, temos:

$$\frac{d}{d\hat{l}}\left(custo\left(\hat{l}\right)\right) = \int_0^{\hat{l}} p(l) \cdot f(l)\, dl - \int_{\hat{l}}^{\infty} p(l) \cdot f(l)\, dl \tag{2.5}$$

Como o mínimo é único, temos que este deve estar perto do mínimo da função discreta. O valor ótimo é dado então pela mediana dos níveis das folhas da Octree ponderada de vezes que foram procuradas. Esta média é dinamicamente calculada mantendo um pequeno histograma dos níveis das folhas.

3 Modelagem da Hashed Octree em Arquitetura de GPUs

A arquitetura da GPU impõe uma série de limitações que forçam algumas adaptações na modelagem e na implementação da hashed octree e da busca otimizada. O presente trabalho tem como principal contribuição uma proposta de modelagem computacional da hashed octree e da busca otimizada adequada para este tipo de arquitetura. Neste capítulo esta modelagem será apresentada.

3.1 Limitações da GPU

A NVIDIA, desde a série 8 da sua linha de GPUs GeForce, unificou os pixel shaders e os vertex shaders em uma só unidade, chamada de *stream processor* (processador de fluxo). Esta unidade é um processador de ponto flutuante, mais simples e muito mais flexível que os shaders. Um fluxo nada mais é que um conjunto de dados que requerem computação similar, provida pelo kernel, função executada para cada elemento dos dados. A partir desta série de GPUs, a NVIDIA desenvolveu o CUDA (Compute Unified Device Architecture), sua arquitetura de programação paralela. Utilizando CUDA, as GPUs se tornaram efetivamente arquiteturas abertas, como as CPUs. Todavia, diferentemente da CPU, a GPU oferece uma arquitetura com um elevado número de núcleos, onde cada um pode executar milhares de threads simultaneamente. Quando uma aplicação pode ser modelada para este tipo de arquitetura, a GPU pode oferecer grande ganho em desempenho, chegando a uma ou mais ordens de grandeza (26).

Ainda recente, a arquitetura CUDA apresenta algumas limitações, listadas abaixo:

- Utiliza um subset da linguagem C, sem suporte a recusão e a ponteiros de função, com algumas extensões simples.
- Renderização de textura não é suportado.
- Precisão dupla só é suportada em versões mais novas de GPUs e contém alguns desvios da norma 754 do IEEE.
- A largura de banda e a latência entre CPU e GPU podem ser um gargalo.
- Threads devem rodar em grupos de pelo menos 32 para melhor desempenho, com o número total de threads em centenas. Desvios no código do programa não causam grande impacto no desempenho, desde que todas as 32 threads em um mesmo grupo sigam o mesmo fluxo de execução, o que não é uma tarefa trivial.
- Algumas operações, como o módulo (operador % em C), são extremamente custosas para a GPU, devido à simplificação do processador de fluxo.

Estas limitações pouco a pouco estão sendo eliminadas, devido à constante evolução do CUDA, porém cada melhoria geralmente só é implementada em GPUs mais recentes. Portanto, o programador CUDA deve estar atento a qual GPU utilizar, para alcançar desempenho máximo.

3.2 Modelagem da Hashed Octree

A modelagem da tabela hash a ser utilizada para a hashed octree envolve escolher adequadamente a função de hash e o tratamento de colisão adequado. Para o caso da GPU, algumas destas decisões são críticas, já que além de limitar as opções, para o completo aproveitamento do poder de processamento da GPU, a estrutura deve permitir que várias execuções em paralelo aconteçam. A modelagem correta desta estrutura permite que o

algoritmo da busca otimizada seja mais eficiente, já que o tempo total do algoritmo de busca depende do tempo de busca da octree.

O primeiro passo é saber como resolver os problemas de colisão. Para isso, deve-se definir como será o encadeamento da tabela hash. O encadeamento aberto, onde uma chave colidida é inserida numa lista pertencente ao índice da tabela hash, é a opção que gera maior economia de memória, pois uma nova posição só é alocada quando um novo nó é inserido.

Sua utilização, porém, requer o uso de ponteiros, o que ainda é proibido na GPU. Assim, a opção que melhor atende é a de encadeamento fechado. O encadeamento fechado armazena os nós colididos em sua própria estrutura, devendo, portanto, ser pré-alocada com tamanho suficiente para absorver todos os nós. Além disso, tabelas hash sofrem perdas de desempenho quando operam muito cheias. Portanto, para evitar estes problemas, o ideal é alocá-la com pelo menos o dobro do tamanho necessário.

Neste trabalho, o encadeamento fechado foi escolhido exatamente por ser melhor adaptável à GPU. Uma investigação sobre a influência do tamanho da tabela hash sobre o tempo de busca foi realizada e será apresentada no capítulo 5.

A função hash h para uma chave k a ser utilizada é $h(k) = k \mod m$, onde m é o número de índices da tabela hash.

Para o rehashing, uma segunda função é necessária. Existem duas opções clássicas para a função rehash: incremento linear e incremento quadrático. Em ambas as funções, deve-se garantir que ela seja capaz de percorrer todas as posições da tabela hash. Por exemplo, numa tabela de tamanho 4, uma função de incremento linear $h'(k) = h(k) + 2$ impede que, dada uma chave k, todas as posições disponíveis sejam investigadas antes da inserção falhar, retornando que a tabela hash está cheia.

Neste trabalho, para impedir os ciclos na função rehash, foi utilizada uma função $h'(k) = h(k) + 1$. Ainda que existam funções mais adequadas, a escolha desta foi devido à sua simplicidade.

3.3 Modelagem das Buscas

Para adaptar o algoritmo de busca para a GPU, deve-se considerar como uma aplicação típica que utiliza Octrees funciona. Basicamente, são feitas várias consultas à Octree, buscando a informação contida nas folhas. Tomando por exemplo o cálculo de colisões, vários pontos são buscados e a partir do resultado de cada busca independente, um processamento é feito. O resultado da busca por um ponto não influi nem depende dos resultados da execução do mesmo algoritmo em outros pontos, mostrando sua independência.

Este fato nos leva a uma adaptação mais simples do algoritmo de busca otimizada (Figura 13. O algoritmo de busca se torna um kernel para a GPU. Um kernel é o programa que executa em cada thread da GPU. Cada thread tem seu fluxo de execução completamente independente e realiza a busca independentemente.

Procedimento Busca de N Pontos
Entrada: VERTICE *pontosBusca*[], HASHITEM *tabelaHash*[]
para *todo ponto p em* ∗*pontosBusca* **faça em paralelo**
  busca ponto *p* em ∗*tabelaHash*
  retorna face *f* encontrada
fim

Figura 13: Modelagem da busca otimizada na GPU

Procedimento Busca de um Único Ponto
Entrada: VERTICE *p*, HASHITEM *tabelaHash*[]
para *todo nível l da Octree* **faça em paralelo**
  busca ponto *p* no nível *l* em ∗*tabelaHash*
  se *nó n encontrado é folha* **então**
    retorna face *f* encontrada
  fim
fim

Figura 14: Modelagem da busca otimizada de um único ponto na GPU

O segundo algoritmo de busca foi pensado para os casos em que uma única busca precisa ser realizada. Para o caso da busca de somente um ponto, lançar somente um thread subutiliza o hardware. Para este caso, o algoritmo a seguir é mais efetivo que o anterior ao realizar paralelamente a busca em todas os níveis da árvore (Figura 14).

4 Implementação

A implementação para a GPU apresenta algumas particularidades, causadas pelas limitações da própria GPU (Seção 3.1). Neste capítulo será apresentada a implementação da estrutura e da busca, descritas no capítulo anterior.

4.1 Implementando a Hashed Octree

Nesta seção será apresentada a implementação da hashed octree. A construção da octree foi feita em CPU, sendo copiada posteriormente para a GPU. Assim, a implementação funciona tanto em CPU quanto em GPU.

Primeiramente, fora criadas duas estruturas básicas para armazenar os vértices e as faces. As estruturas utilizadas foram as seguntes:

```
typedef struct VERTICE
{
  float x, y, z;
};

typedef struct FACE
{
  int v1, v2, v3;
};
```

A estrutura VERTICE armazena as coordanadas de um ponto no espaço, relativo a um vértice. A estrutura FACE armazena três índices inteiros que apontam para as posições dos vértices do triângulo num vetor de vértices.

Duas outras estruturas adicionais foram utilizadas para auxiliar o processo de criação da octree:

```
typedef struct BBOX
{
  float centro[3], raio;
};

typedefstruct NODE
{
  FACE* faces;
  int qtdeFaces;
  BBOX bbox;
  int isFolha;
  int nivel;
};
```

A estrutura BBOX armazena a caixa envolvente do nó da octree, armazenando a coordenada do centro e seu raio. Já a estrutura NODE armazena o nó da octree, possuindo um vetor de faces, com todas as faces contidas em sua caixa envolvente, um campo para indicar se o nó é uma folha ou um nó interno e seu nível, além de sua própria caixa envolvente.

A criação da octree passa pelo carregamento do modelo. Foram utilizados modelos armazenados em formato PLY (Polygon File Format, também conhecido como Stanford File Format). Os arquivos carregados geravam uma lista de vértices e uma lista de faces.

Após o carregamento, foi implementado o algoritmo apresentado na figura 1, porém inserindo o nó atual na tabela hash antes da recursão referente à geração dos filhos.

A implementação da tabela hash é o ponto principal da estrutura proposta. A implementação correta desta estrutura permite que o algoritmo da busca otimizada seja mais eficiente, já que o tempo total do algoritmo de busca depende do tempo de busca da octree.

Algumas restrições foram observadas durante a implementação da tabela hash. A mais importante é a limitação da GPU quanto ao uso de ponteiros. Uma vez que a GPU não permite que se utilizem ponteiros, há duas alternativas para a implementação da tabela hash. A primeira delas consiste em utilizar encadeamento aberto sem utilizar ponteiros, criando uma vetor de tamanho N, pré-definido, e em cada posição deste vetor, criar outro vetor de tamanho pré-definido, para o caso de colisão. A segunda é utilizar o encadeamento fechado na tabela.

Como não é possível determinar previamente a quantidade de nós que vão colidir, cada posição do vetor da primeira opção deveria manter outro vetor suficientemente grande para armazenar todo o modelo a ser utilizado. Assim, esta solução gera um grande desperdício de memória. Desta forma, neste trabalho foi utilizada a técnica de encadeamento interno, alocando assim um vetor de tamanho mínimo de 2 vezes o número total de nós na octree.

Definido qual encadeamento a ser utilizado, o próximo passo é decidir qual função de rehash utilizar. Pela simplicidade, foi utilizada uma função linear $h'(k) = h(k) + 1$.

Para armazenar os dados necessários de cada entrada da tabela hash, a seguinte estrutura foi criada:

```
typedef struct HASHITEM
{
  UINT64 mortonCode;
  FACE face;
```

```
    int isFolha;
};
```

Esta estrutura armazena o código de Morton do nó, que é a chave de busca utilizada, a face contida no nó folha e um flag para indicar se aquele nó é folha ou não. Este flag será importante para a implementação da busca. O algoritmo de busca otimizada em CPU testa um acesso inválido para decidir se o nó existe ou não. Na GPU, este campo deve ser explicitamente declarado, pois este teste não é possível.

Com isso, a função de inserção na tabela hash foi a seguinte:

```
void insert(NODE no){
  UINT64 m = mortonCode(no.bbox.centre, no.level);
  int probe = hash(m);
  HASHITEM item;
  item.mortonCode = m;
  item.face = value.faces[0];
  item.isLeaf = value.isLeaf;
  while (hashTable[probe].isLeaf != EMPTY){
    probe = (probe + 1) % HASH_SIZE;
  }
  hashTable[probe] = item;

  if((item.isLeaf == LEAF)||(item.isLeaf == EMPTYLEAF))
    histograma[value.level]++;
}
```

A estrutura na GPU é composta de um vetor de HASHITEM, que compõem a tabela hash, mais o vetor de faces e o de vértices. Ao final da adaptação da Octree na CPU,

estas estruturas são copiadas da CPU para a GPU, ficando prontas para a execução das buscas.

4.2 Implementando a Busca Otimizada

Baseado no algoritmo de busca otimizada proposto por Castro, dois algoritmos foram propostos para este trabalho, sendo um para realizar a busca de vários pontos em paralelo e um responsável por paralelizar a busca nos níveis da árvore, permitindo a busca de um único ponto em todos os níveis da árvore em paralelo.

Nesta seção serão apresentados os dois algoritmos e suas principais particularidades.

4.2.1 Busca de vários pontos

O primeiro algoritmo proposto visa aproveitar o fato que as buscas de cada ponto são independentes e, em aplicações reais (por exemplo, colisões), muitas buscas são realizadas simultaneamente. Assim, o paralelismo da GPU pode ser aproveitado explorando vários pontos ao mesmo tempo. O algoritmo criado foi o seguinte:

O algoritmo recebe o vetor de pontos a serem buscados, a quantidade de pontos e a tabela hash. O kernel calcula o identificador do thread e acessa a posição correspondente no vetor de pontos, caso seja um acesso válido. A partir deste ponto, realiza-se a busca, como na CPU. Cada thread então armazena o resultado da busca num vetor de saída, na posição relacionada ao identificador da thread. Ao final da execução do kernel, o vetor de saída é descarregado para a CPU.

4.2.2 Busca de um único ponto

O primeiro algoritmo busca aproveitar o fato de várias buscas poderem ser feitas simultaneamente em uma aplicação típica. Porém, em alguns casos, somente uma única busca é realizada. Para aproveitar o paralelismo da GPU nesses casos, um novo algoritmo foi proposto.

Procedimento Kernel para Busca de N Pontos

Entrada: VERTICE $*pontosBusca$, int $qtdePontosBusca$, HASHITEM $*tabelaHash$

Saída: FACE $*facesEncontradas$

$thread\_id \leftarrow$ índice thread

$codMortonMax \leftarrow$ código de Morton no nível máximo

se $thread < qtdePontosBusca$ **então**

  $p \leftarrow pontosBusca[thread\_id]$

  $\hat{l} \leftarrow$ nível estimado

  $codMortonNivel =$ código de Morton do ponto p no nível estimado $\hat{l}$

  $n \leftarrow$ nó correspondente a $codMortonNivel$ na tabela hash

  enquanto n *não existe na tabela hash* **faça**

    decrementa o nível l, retirando 3 bits de $codMortonNivel$

    $n \leftarrow$ nó pai de n, correspondente a $codMortonNivel$ na tabela hash

  fim

  enquanto n *existe na tabela hash* **faça**

    incrementa o nível l, adicionando 3 bits de $codMortonMax$ a $codMortonNivel$

    $n \leftarrow$ nó filho de n, correspondente a $codMortonNivel$ na tabela hash

  fim

fim

$facesEncontradas[thread\_id] \leftarrow$ face contida no último nó folha n encontrado

Figura 15: Busca otimizada na GPU

No algoritmo anterior, cada thread realiza a busca uma vez em cada nível da árvore. Já neste algoritmo, dispara-se uma thread para cada nível da árvore, fazendo assim que um único ponto seja buscado na tabela hash em todos os níveis da octree simultaneamente.

Há uma versão mais otimizada deste algoritmo, para o caso do número de threads serem limitados. Desta forma, disparam-se as threads idealmente perto do nível estimado $\hat{l}$, onde a probabilidade da busca ser realizada com sucesso é maior. Aqui, para evitar esforços repetidos, a travessia para cima (caso 2) é realizada por metade das threads, subindo $\frac{g-1}{2}$ níveis ao invés de 1, e as travessias para baixo.

Procedimento Kernel para Busca de Único Ponto

Entrada: VERTICE *ponto*, int *nMaxOctree*, HASHITEM $*tabelaHash$
Saída: FACE $faceEncontrada$
$thread\_id \leftarrow$ índice thread
$codMortonMax \leftarrow$ código de Morton no nível máximo
se $thread < nMaxOctree$ **então**
  $codMortonNivel =$ código de Morton do ponto p no nível $thread\_id$
  $n \leftarrow$ nó correspondente a $codMortonNivel$ na tabela hash
  se n *é folha* **então**
    $faceEncontrada \leftarrow$ face contida em n
  fim
fim

Figura 16: Busca de um único ponto na GPU

Procedimento Busca de um único ponto

computa código de Morton $m_{\max}$ de p na profundidade máxima
para $iter \in 1..iter_{\max}$ **faça**
  para cada *thread* $t_i \in (0 \cdots g-1)$ **faça**
    se $t_i \leq \frac{g-1}{2}$ **então**
      $l_i \leftarrow \hat{l} + t_i + iter \cdot \frac{g-1}{2}$
    senão
      $l_i \leftarrow \hat{l} + t_i - (iter+2) \cdot \frac{g-1}{2}$ - 1
    fim
    computa código m de p na profundidade l_i a partir de $m_{\max}$
    accessa o nó n correspondente a m na tabela hash
    se n *é uma folha* **então**
      retorna n
    fim
  fim
fim

Figura 17: Busca de um único ponto p na GPU, dado um número g limitado de threads

5 Resultados

Este capítulo descreve os resultados obtidos com os algoritmos apresentados no capítulo 4, adaptando o algoritmo apresentado na seção 2.3.2. Comparações com o algoritmo otimizado na CPU também foram feitas. Os modelos utilizados para teste e a metodologia do teste das buscas são apresentados abaixo. A Octree implementada foi descrita também no capítulo 4 e de forma mais geral na seção 2.1.2. Para todos os testes foram utilizados um Core 2 Duo de 2,53GHz e uma GeForce 9600M GT com 32 processadores de fluxo, organizados em 4 multiprocessadores, e 512 MB de RAM. Os processadores da GPU utilizada operam a uma velocidade de 1,25GHz.

5.1 Parâmetros de Teste

A hashed octree utilizada para os testes foi adaptada num conjunto de pontos da seguinte maneira: os nós foram subdivididos até conterem somente um único ponto ou a octree atingir o nível máximo. O nível máximo foi definido como 21, pois a chave utilizada possui 64 bits. Com o nível limitado em 21, a combinação das três coordenadas utiliza 63 bits, mais o bit de controle.

Os dados utilizados para adaptar a Octree foram modelos em formato ply, de diferentes resoluções (Figura 18). Foram testadas a adaptação da estrutura para vários modelos, porém a limitação de memória da GPU impede que modelos muito grandes sejam utilizados. Em (20), a estrutura para o modelo Thai Statue, por exemplo, ocupou 535 MB de memória, enquanto a grande maioria das GPU trabalham com 512 MB ou menos.

(a) Modelo Ant

(b) Modelo Armadillo

(c) Modelo Bunny

(d) Modelo Drill Zipped e VRip

Figura 18: Modelos utilizados para testes na Octree

As resoluções dos modelos apresentados (quantidade de vértices e faces) estão na figura 19.

Modelo	**# vértices**	**# faces**
Ant	486	912
Armadillo	172974	345944
Bunny	35497	69451
Drill vrip	1961	3855
Drill zipped	881	1281

Figura 19: Resolução dos modelos utilizados para os testes

5.2 Testes Realizados

Para validar a estrutura e a busca na GPU, três testes foram realizados. Para a estrutura, um teste variando o tamanho da estrutura foi criado. Desta maneira, podemos

ver se seu tamanho tem impacto no tempo de busca em si. Já para validar o algoritmo de busca, foram realizadas buscas de vários pontos nos modelos citados na seção anterior e também a busca de um único ponto. Para comparar os tempos de busca, uma comparação com uma implementação em CPU foi realizada.

Os pontos utilizados para as buscas foram pontos randomicamente escolhidos dentro da caixa envolvente do objeto. Assim, as buscas vão ocorrer sempre em uma área próxima ao objeto; área esta bem mais subdividida na octree.

O primeiro teste, que envolve variar o tamanho da tabela hash da hashed octree, foi realizado com o modelo Ant.

Tamanho da tabela hash (em número de elementos)

	2,000,000	**1,000,000**	**500,000**	**250,000**	**100,000**
CPU	5 s	2 s	1 s	1 s	1 s
GPU	0.33 s	0.29 s	0.28 s	0.28 s	0.27 s

Figura 20: Tempo de execução (em segundos) para o modelo Ant, com diferetes tamanhos de tabela hash.

Na tabela 20 pode-se observar a pouca influência do tamanho da tabela hash nos tempos de busca, enquanto se mantém grande o suficiente para armazenar o modelo. O tempo apresentado é uma média aritmética dos tempos de execução. Cada algoritmo (CPU e GPU) foi executado 20 vezes, e envolveu a busca de 50 pontos no modelo.

A pouca influência do tamanho da tabela hash nos tempos de busca implica que a tabela hash funciona bem, mesmo quando grandes modelos são utilizados. Vale citar que o desempenho da tabela hash cai caso sua ocupação seja maior que 50%.

O segundo teste envolve a busca de um diferente número de pontos (de 10 a 300) na hashed octree, nos modelos citados. Novamente o tempo de cada busca é uma média aritmética de 20 execuções de cada algoritmo, tanto para CPU quanto para GPU.

A comparação de tempo absoluto está na figura 23. Pode-se observar que o algoritmo proposto teve um bom desempenho, sendo executado no mínimo 3 vezes mais rápido, em buscas menores, e executando em média 45 vezes mais rápido, na busca por 300 pontos.

Nas figuras 21 (CPU) e 22 (GPU), podemos observar que o aumento do número de pontos implica em um aumento linear com uma proporção grande na busca na CPU, enquanto os resultados na GPU evidenciam o benefício da paralelização no algoritmo proposto, já que o incremento do tempo é menor. O speedup conseguido é exibido na figura 23.

# pontos	**10**	**50**	**100**	**200**	**300**
Ant	1	2	5	10	14
Armadillo	1	4	8	17	25
Bunny	1	3	7	14	21
Drill vripped	1	2	4	6	10
Drill zip	1	3	7	15	21

Figura 21: Tempo total de execução (em segundos) para a CPU.

# pontos	**10**	**50**	**100**	**200**	**300**
Ant	0.27	0.28	0.36	0.39	0.43
Armadillo	0.29	0.29	0.38	0.41	0.44
Bunny	0.28	0.35	0.37	0.4	0.37
Drill vripped	0.28	0.35	0.37	0.34	0.37
Drill zip	0.28	0.28	0.21	0.37	0.38

Figura 22: Tempo total de execução (em segundos) para a GPU.

# pontos	**10**	**50**	**100**	**200**	**300**
Ant	3,70	7,14	13,89	25,64	32,56
Armadillo	3,45	13,79	21,05	41,46	56,82
Bunny	3,57	8,57	18,92	35,00	56,76
Drill vripped	3,57	5,71	10,81	17,65	27,03
Drill zip	3,57	10,71	33,33	40,54	55,26

Figura 23: Speed-up do algoritmo para os casos de teste. O algoritmo proposto teve tempo de execução pelo menos 3 vezes menor que o algritmo em CPU.

Na figura 25, podemos observar que o tempo de execução cresce num padrão linear, porém bem lentamente, diferente da CPU, mostrado na figura 24.

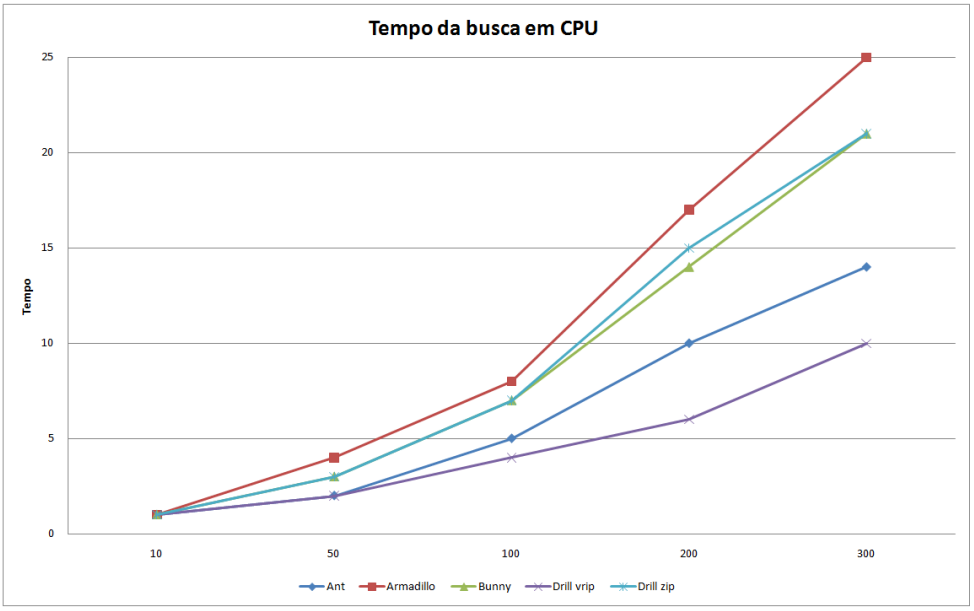

Figura 24: Tempos de execução do algoritmo da CPU (Figura 21). O gráfico exibe o alto crescimento de tempo quando o tamanho da busca aumenta.

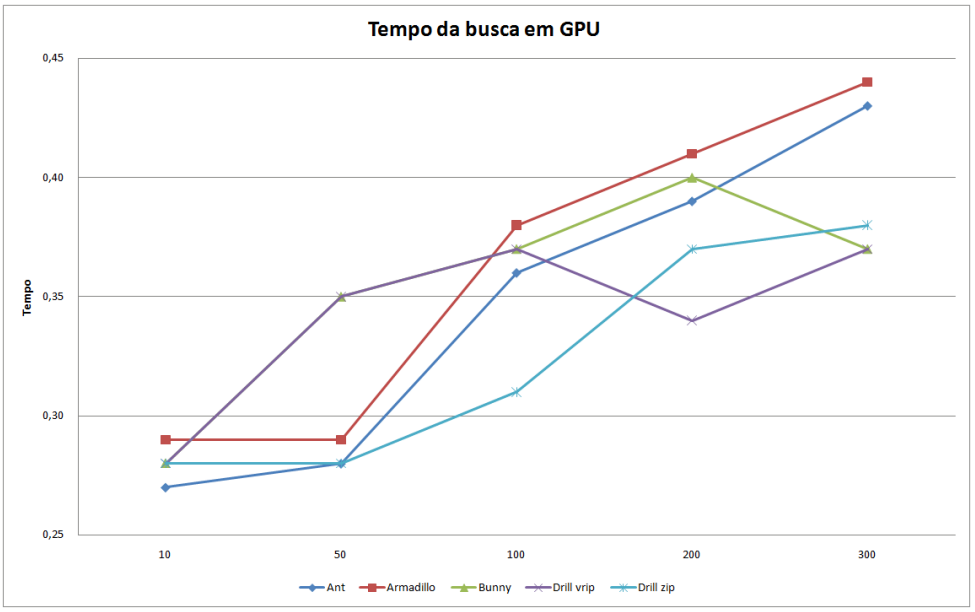

Figura 25: Tempos de execução do algoritmo da GPU (Figura 22). O gráfico exibe o padrão linear do algoritmo proposto.

O terceiro teste envolve a busca de um único ponto. Os tempos obtidos estão na figura 26. Pode-se observar que os tempos foram parecidos com a busca de 10 pontos do algoritmo anterior. A busca de um único ponto leva pouca carga de trabalho para a GPU, fazendo com que o overhead de processamento da mesma apareça mais. Ainda assim, é um bom método, pois permite que a estrutura da hashed octree seja mantida sempre em GPU, seja para buscas de muitos ou apenas um ponto.

	Tempo (s)
Ant	0,25
Armadillo	0,24
Bunny	0,24
Drill vripped	0,21
Drill zip	0,23

Figura 26: Tempo total de execução da busca de um único ponto na GPU.

6 Conclusões e Trabalhos Futuros

Esta dissertação apresentou uma estrutura de octree para a GPU que permite buscas otimizadas. Os algoritmos propostos otimizaram o tempo de busca em pelo menos 3 vezes, melhorando o tempo de busca em média em 22 vezes. Os algoritmos foram comparados com uma implementação em CPU, para uma comparação de tempo de execução. Como a estrutura utilizada tanto na CPU quanto na GPU foi a mesma, não houve gasto extra de memória. Além dos tempos de busca conseguidos, um outro ganho da adaptação da estrutura da hashed octree é permitir sua utilização completa na GPU. A estrutura é completamente mantida em GPU, reduzindo o gargalo que existe na transferência de dados entre CPU e GPU.

Este trabalho pode ser aprimorado com a construção e a atualização da hashed octree diretamente na GPU. Novas funções de hash e um método eficiente de ajuste do tamanho da tabela hash também podem ser criadas, para melhorar o desempenho das buscas na tabela hash.

Como sugestão de trabalhos futuros, estão a criação e atualização da octree diretamente na GPU, além da adaptação de outras estruturas como as k-d trees e BSPs. Além disso, a pesquisa de outras funções estatísticas que se adaptem melhor à outras estruturas ou até mesmo a casos específicos de utilização da octree. A criação de um aplicativo de demonstração do uso da estrutura apresentada também é interessante.

A aplicação deste trabalho em aplicações específicas, como um sistema de visualização de Ray-tracing em tempo real, simulador de fluidos, entre outras, também é uma sugestão interessante de trabalhos futuros.

Referências Bibliográficas

1 LEFEBVRE, S.; HORNUS, S.; NEYRET, F. Octree textures on the GPU. In: *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. [S.l.]: Addison Wesley, 2005. cap. 37, p. 595–613.

2 SAMET, H. *The design and analysis of spatial data structures*. [S.l.]: Addison-Wesley, 1990.

3 CHEN, H. H.; HUANG, T. S. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing*, v. 43, n. 3, p. 409–431, 1988.

4 PUECH, C.; YAHIA, H. Quadtrees, octrees, hyperoctrees: a unified analytical approach to tree data structures used in graphics, geometric modeling and image processing. In: *SCG '85: Proceedings of the first annual symposium on Computational geometry*. [S.l.]: ACM, 1985. p. 272–280.

5 TANG, Z. Octree representation and its applications in CAD. *Journal of Computer Science and Technology*, Springer, v. 7, n. 1, p. 29–38, 1992.

6 COATRIEUX, J.; BARILLOT, C. A survey of 3D display techniques to render medical data. *3D Imaging in Medicine, Algorithms, Systems and Applications*, p. 175–195, 1990.

7 JIMENEZ, P.; THOMAS, F.; TORRAS, C. 3d collision detection: a survey. *Computers & Graphics*, v. 25, n. 2, p. 269–285, 2001.

8 KOBBELT, L.; BOTSCH, M. A survey of point-based techniques in computer graphics. *Computers & Graphics*, Elsevier, v. 28, n. 6, p. 801–814, 2004.

9 NEWMAN, T. S.; YI, H. A survey of the marching cubes algorithm. *Computers & Graphics*, v. 30, n. 5, p. 854–879, 2006.

10 KNOLL, A. A Survey of Octree Volume Rendering Methods. In: *IRTG Workshop*. [S.l.: s.n.], 2006.

11 BUCK, I. et al. Brook for GPUs: stream computing on graphics hardware. In: *Siggraph*. [S.l.: s.n.], 2004. p. 777–786.

12 ZAMITH, M. et al. The GPU used as a math co-processor in real time applications. *Journal of Computer in Entertainment: CIE*, v. 6, p. 1–19, 2008.

13 CASTRO, R. et al. Statistical optimization of octree searches. *Computer Graphics Forum*, v. 27, p. 1557–1566, 2008.

14 WILHELMS, J.; GELDER, A. V. Octrees for faster isosurface generation. In: *ACM Transactions on Graphics*. [S.l.: s.n.], 1992. v. 11, p. 201–227.

15 BENSON, D.; DAVIS, J. Octree textures. In: *Siggraph*. [S.l.: s.n.], 2002. v. 21, n. 3, p. 785–790.

16 LEFOHN, A. et al. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, v. 25, n. 1, p. 60–99, jan. 2006. Disponível em: <http://graphics.cs.ucdavis.edu/ lefohn/work/glift/>.

17 LEFEBVRE, S.; HOPPE, H. Perfect spatial hashing. In: *Siggraph*. [S.l.: s.n.], 2006. p. 579–588.

18 BASTOS, T.; CELES, W. GPU-accelerated Adaptively Sampled Distance Fields. In: *Shape Modeling and Applications, 2008*. [S.l.: s.n.], 2008. p. 171–178.

19 DIETZFELBINGER, M. et al. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 23, n. 4, p. 738–761, 1994. ISSN 0097-5397.

20 CASTRO, R. *Otimização Estatística de Buscas para Estruturas Hierárquicas Espaciais*. Tese (Doutorado) — PUC-Rio, mar. 2008.

21 MORTON, G. M. *A computer oriented geodetic data base and a new technique in file sequencing*. [S.l.], 1966.

22 STOCCO, L.; SCHRACK, G. Integer dilation and contraction for quadtrees and octrees. In: *Pacific Rim Conference on Communications, Computers, and Signal Processing, 1995*. [S.l.: s.n.], 1995. p. 426–428.

23 GARGANTINI, I. Linear octrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, v. 4, n. 20, p. 365–374, 1982.

24 GLASSNER, A. Space subdivision for fast ray tracing. *Computer Graphics & Applications*, v. 4, n. 10, p. 15–22, 1984.

25 WARREN, M. S.; SALMON, J. K. A parallel hashed octree n-body algorithm. *Supercomputing, IEEE*, p. 12–21, 1993.

26 VASILIADIS, G. et al. Gnort: High performance network intrusion detection using graphics processors. In: *Recent Advances In Intrusion Detectiom (RAID)*. [S.l.: s.n.], 2008.