3
Representação e Reconstrução de ADFs em GPU

A estrutura clássica de uma ADF [24, 17] consiste de uma *octree* baseada em ponteiros, que armazena oito *voxels* por célula. Cada *voxel* corresponde a um valor de distância amostrado em um dos vértices da célula. Esta abordagem tem dois grandes problemas para uma implementação em GPU:

1. Uma vez que cada vértice da *octree* é compartilhado por várias células, a estrutura acaba por armazenar muitos *voxels* redundantes. Como a memória das GPUs é bastante limitada, redundâncias podem impedir que uma ADF caiba em memória de vídeo.

2. Percorrer *octrees* baseadas em ponteiros na GPU é complexo e ineficiente. Visto que as GPUs não têm suporte a recursão e nem operações de escrita livre em memória, é impraticável utilizar uma pilha para percorrer a *octree*. Mesmo que fosse viável implementar uma pilha, provavelmente o desempenho viria a sofrer por conta da baixa coerência do algoritmo.

Para extrair o máximo do poder de processamento paralelo das GPUs, as próximas seções propõem soluções para reduzir a redundância de *voxels* e permitir que a *octree* seja percorrida sem uma pilha.

PUC-Rio - Certificação Digital Nº 0611941/CA

3.1
A Textura de Voxels

A replicação de *voxels* pode fazer com que ADFs minimamente complexas excedam a capacidade de memória das GPUs modernas (em torno de 768 MB). Para controlar a redundância, é preciso desacoplar o armazenamento de *voxels* da estrutura da *octree*. Há várias soluções possíveis para este problema, com variados níveis de complexidade e eficiência. Uma solução muito simples que elimina toda redundância foi sugerida por Bærentzen [2]: utilizar uma tabela de dispersão 3-D para armazenar os *voxels* indexados pela posição do vértice. Porém, nessa solução cada célula acessada exigiria oito consultas à tabela de dispersão, o que afligiria o desempenho da estrutura e provavelmente tornaria seu uso inviável em aplicações interativas. Como o acesso aos *voxels* tem papel */crucial/* no desempenho, ele deve ser tão simples e coerente quanto possível.

Este trabalho propõe uma abordagem alternativa: armazenar os *voxels* em uma textura separada da *octree*, e fazer com que as células guardem ponteiros para os seus *voxels*. Apesar de soar como uma solução óbvia, na realidade é *muito* difícil empacotar todos os *voxels* em uma textura compacta sem replicá-los. Naturalmente, não faz sentido armazenar oito ponteiros em cada célula, já que os ponteiros tomariam quase tanta memória quanto oito amostras de distância. Para que esta abordagem faça sentido, os *voxels* pertencentes a uma célula precisam ser organizados em *blocos* contíguos, de maneira que cada célula possa guardar apenas um ponteiro (para o bloco) ao invés de oito. Em última análise, busca-se nesta abordagem:

1. Empacotar todos os *voxels* em uma textura 3-D compacta, com um único componente do tipo *float*.

2. Garantir que os *voxels* pertencentes à mesma célula estejam adjacentes em um bloco de $2 \times 2 \times 2$ *texels*.

3. Minimizar a replicação de *voxels* sempre que possível, empacotando juntos os *voxels* vizinhos de mesmo nível.

A construção de uma textura de *voxels* pelos critérios acima não é uma tarefa trivial. Apesar de existirem algoritmos baseados em heurísticas capazes de produzir resultados com pouca redundância, encontrar a solução ótima é notoriamente um problema combinatorial NP-difícil, uma vez que o problema se trata de uma variante complexa do problema da mochila.

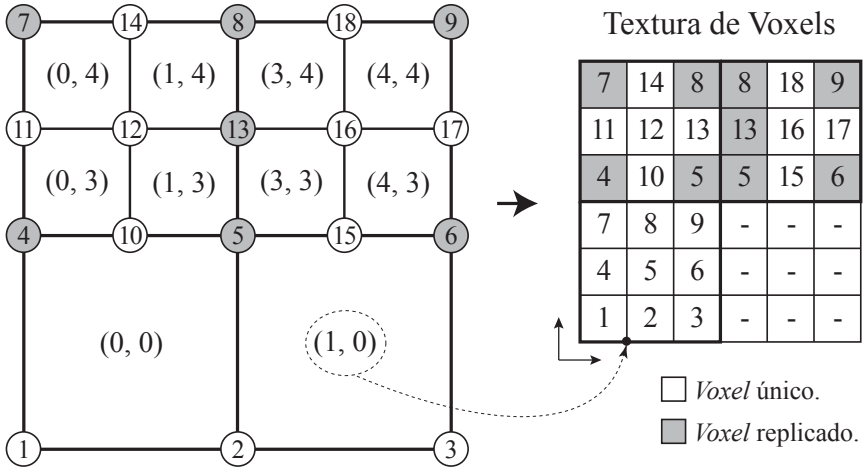

Figura 3.1: Uma *quadtree* (esquerda) e a sua textura de voxels (direita).

Por simplicidade, a solução adotada nesta dissertação elimina apenas a redundância entre células irmãs. Isto é feito facilmente, armazenando os *voxels* de cada grupo de oito células irmãs em um bloco de 3x3x3 *texels*.

PUC-Rio - Certificação Digital Nº 0611941/CA

Uma vez que as *octrees* das ADFs são árvores cheias, todas as células, exceto a raiz, pertencem a um grupo de oito irmãs. As redundâncias permanecem entre outros tipos de células adjacentes (*e.g.* células primas) e entre células descendentes, mas esta solução simples reduz consideravelmente (em $\sim 57\%$) o tamanho da textura de *voxels*, e é suficiente para permitir que ADFs razoavelmente complexas caibam em memória de vídeo. Além disso, já que os *voxels* das células são organizados em blocos de 2x2x2 *texels*, é possível explorar a unidade de filtragem de textura das GPUs para obter — praticamente de graça — valores de distância interpolados trilinearmente. A Figura 3.1 ilustra esta solução em 2-D, com uma *quadtree* e a sua textura de *voxels* correspondente. Os *voxels* são numerados de um a dezoito, e *voxels* replicados são sombreados em cinza. A posição do bloco de *voxels* correspondente de cada célula é indicada entre parênteses, no centro das células.

3.2 Visão Geral da Representação das ADFs

O espaço de uma ADF é um cubo unitário com o primeiro vértice na origem, como ilustrado na Figura 3.2. A reconstrução do campo de distância e o endereçamento de células é sempre feito neste espaço. A mesma convenção é usada para o espaço local das células da *octree*.

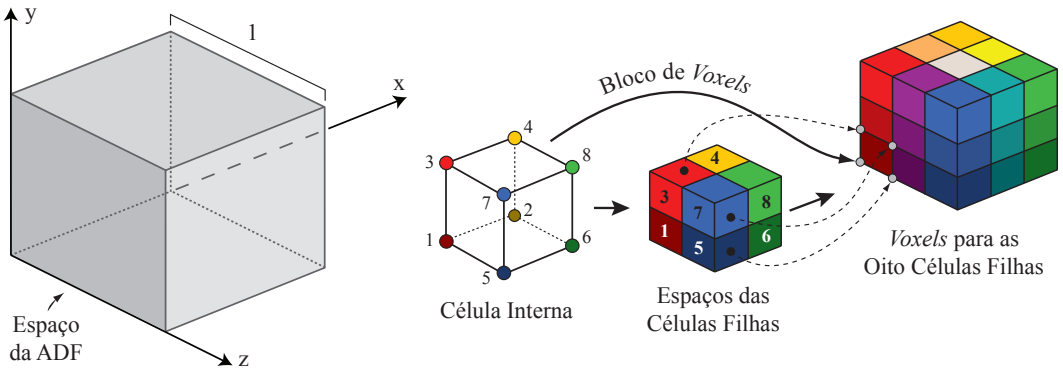

Figura 3.2: Visão geral da representação de ADF utilizada neste trabalho.

As ADFs propostas neste trabalho são representadas em memória de vídeo através de três texturas tridimensionais:

Textura de *Voxels*: armazena os *voxels* de cada grupo de oito células irmãs em blocos de 3x3x3 *texels*, como detalhado na Seção 3.1. A textura garante que os oito *voxels* de cada célula estão sempre juntos, em blocos de 2x2x2 *texels*.

PUC-Rio - Certificação Digital Nº 0611941/CA

Textura de Células: armazena cada célula da ADF em um *texel*. Cada *texel* guarda duas informações: o identificador da célula, e o endereço para o bloco de *voxels* da célula, na textura de *voxels*.

Textura de Dispersão: descreve a hierarquia das células. As Seções 3.3 e 3.4 fornecem mais detalhes sobre esta textura.

Como as *octrees* de ADFs são árvores cheias, as células-folha podem ser mantidas *implícitas*, o que reduz drasticamente (em até 87%) o tamanho da textura de células. Neste esquema, a textura de células armazena somente as células não-folha. Cada célula, ao invés de guardar um ponteiro para seu próprio bloco de *voxels*, passa a guardar um ponteiro para um bloco de 3x3x3 *texels*, que contém os *voxels* dos seus oito filhos (Figura 3.2).

3.3 Octrees Baseadas em Dispersão Espacial

Tipicamente, as *octrees* de ADFs são implementadas com ponteiros. Porém, na arquitetura atual das GPUs, percorrer *octrees* com ponteiros é complexo e ineficiente. Para superar estas limitações, propõe-se o uso de uma *octree* que permita *acesso aleatório* às células. Dado um nível ℓ da *octree* e um ponto $\mathbf{p}$ pertencente ao domínio, a célula de nível ℓ que contém o ponto $\mathbf{p}$ deve ser obtida em tempo constante. Esta representação permite que a *octree* seja percorrida sem uma pilha, e com maior coerência.

A solução baseia-se na idéia de uma *octree* definida implicitamente em uma textura 3-D. Semelhante à maneira como uma *árvore binária completa* pode ser representada implicitamente em um vetor 1-D, uma *octree completa* pode ser representada implicitamente em um vetor 3-D. É possível, então, formular uma função que calcula em tempo $O(1)$ a posição de qualquer célula dentro do vetor 3-D. Evidentemente, o custo para se representar ADFs com *octrees completas* é impraticável, já que as ADFs são inerentemente esparsas e o espaço tomado pelas *octrees completas* é exponencial na altura da árvore. Como solução, um mecanismo de dispersão espacial 3-D é usado para *compactar* o vetor 3-D esparso onde a *octree* implícita é definida.

Dados uma *octree completa* no cubo $\mathbf{C} = [0,1)^3$, um nível ℓ e um ponto $\mathbf{p} \in \mathbf{C}$, a função

$$S(\ell, \mathbf{p}) = 2^\ell + \left\lfloor 2^\ell \mathbf{p} \right\rfloor \tag{3-1}$$

retorna o índice 3-D da célula de nível ℓ que contém o ponto $\mathbf{p}$, e assim define implicitamente a *octree* no espaço $\mathbf{U} \subset \mathbb{N}^3$

$$\mathbf{U} = \left[1, 2^{(\sup \ell + 1)} - 1\right]^3 \tag{3-2}$$

PUC-Rio - Certificação Digital Nº 0611941/CA

onde $\sup \ell$ é o nível máximo da *octree*.

Uma *octree* baseada em dispersão espacial é criada combinando-se a função $S(\ell, \mathbf{p})$ com uma função de dispersão tridimensional $H(\mathbf{k} \in \mathbf{U})$. O acesso a uma célula é sempre feito em dois passos:

1. A função $S(\ell, \mathbf{p})$ é usada para gerar uma "chave" $\mathbf{k} = S(\ell, \mathbf{p})$ para a célula que deseja-se acessar.

2. A função $H(\mathbf{k})$ recebe como entrada a "chave" e calcula o endereço real da célula em um vetor compacto, usando como base um algoritmo de dispersão espacial. Esta função também pode indicar um erro, caso a chave não corresponda a nenhuma célula da *octree*.

Já que a função $S(\ell, \mathbf{p})$ é claramente $O(1)$, o acesso às células poderá ser feito em tempo constante se $H(\mathbf{k})$ também for $O(1)$. A Seção 3.4 mostra como é possível garantir que a função de dispersão espacial seja de fato $O(1)$.

A Figura 3.3 ilustra os passos necessários para que os *voxels* nos vértices de uma célula sejam acessados. Inicialmente, a função $S(\ell, \mathbf{p})$ gera uma chave $\mathbf{k} \in \mathbf{U}$, que identifica a célula de nível ℓ que contém o ponto $\mathbf{p}$. Esta chave é passada à função de dispersão 3-D $H(\mathbf{k})$, que utiliza uma "textura de dispersão" auxiliar para calcular a posição real da célula em uma "textura de células" compacta. A partir do *texel* da célula obtém-se a posição $\mathbf{v}$ do bloco de *voxels* da célula na "textura de *voxels*". Finalmente, o acesso aos *voxels* é representado na figura pela função $A(\mathbf{v})$.

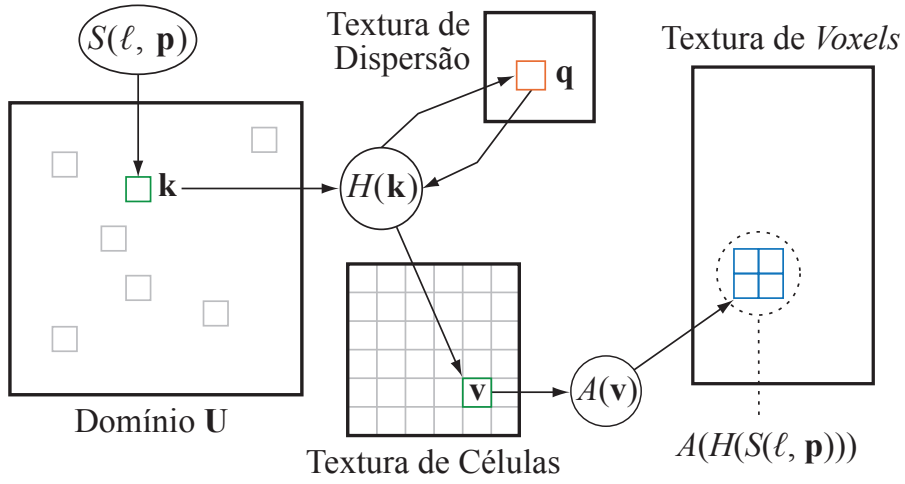

Figura 3.3: Ilustração em 2-D do acesso aos *voxels* nos vértices de uma célula de uma *octree* baseada em dispersão espacial.

Qualquer algoritmo que precise percorrer a *octree* deve trabalhar com o conceito de uma posição $\mathbf{p}$ e um nível ℓ correntes. Através da função $S(\ell, \mathbf{p})$, esses valores indicam qual é a *célula* corrente. Desse modo, cada passo de um algoritmo de percorrimento pode:

PUC-Rio - Certificação Digital Nº 0611941/CA

a) Pular para uma célula adjacente, somando-se $\pm 2^{-\ell}$ a um ou mais componentes do vetor 3-D $\mathbf{p}$.

b) Subir ou descer na hierarquia através do decremento ou incremento de ℓ, respectivamente.

Por exemplo, a chamada

$$S\left(\ell, \mathbf{p} + (-2^{-\ell}, 0, 0)\right)$$

fornece a chave para a célula à esquerda da célula corrente.

3.4 Dispersão Espacial Perfeita

A escolha do algoritmo de dispersão 3-D é determinante na eficiência de uma *octree* baseada em dispersão espacial. Infelizmente, os algoritmos de dispersão tradicionais são inapropriadas para arquiteturas SIMD (*Single-Instruction, Multiple-Data*) como as GPUs.

No artigo [20], Lefebvre e Hoppe apontam que os algoritmos de dispersão costumam resolver colisões realizando uma seqüência de acessos (*probes*) na tabela de dispersão. O número desses acessos pode variar para cada consulta, em função do número de colisões ocorridas para cada chave. Este tipo de tratamento para as colisões é ineficiente nas GPUs, porque o paralelismo SIMD faz com que todos os dados de um mesmo bloco esperem sempre pelo maior número de acessos. Por exemplo, em um bloco de 16 chaves que devem ser consultadas, se 15 das 16 consultas fizerem menos de 3 acessos e apenas uma consulta fizer 10 acessos, todas as 16 consultas levarão o tempo de 10 acessos.

Nas GPUs, o paralelismo SIMD ocorre no processamento de *fragmentos*, após a rasterização. Geralmente a tela é dividida em blocos com tamanhos que variam entre 4x4 até 16x16 fragmentos, e cada bloco é processado de forma independente por um grupo de processadores. Até a introdução das *GeForce 8*, no final de 2006, todas as GPUs eram equipadas com processadores vetoriais especializados no processamento de fragmentos. Desde então, as novas GPUs abandonaram os processadores vetoriais e adotaram uma arquitetura escalar baseada em *processadores de fluxo* (*stream processors*) [18]. A nova arquitetura traz um aumento de desempenho de até dez vezes, ao permitir maior paralelismo. No entanto, o modelo de processamento para fragmentos continua sendo SIMD.

Para tirar bom proveito do paralelismo SIMD, todas as consultas devem realizar o mesmo número de acessos à tabela de dispersão. Isto pode ser conseguido se uma função de dispersão for *pré-calculada* de forma a garantir

PUC-Rio - Certificação Digital Nº 0611941/CA

que não existam colisões entre as chaves de uma *octree*. Neste caso, todas as consultas seriam feitas com um único acesso, através de uma *função de dispersão perfeita*.

As funções de dispersão perfeitas são funções *injetoras* para um certo domínio (*e.g.* conjunto de chaves). Quando a cardinalidade do contradomínio (a tabela de dispersão) é igual à cardinalidade do domínio, a função de dispersão perfeita é *mínima*. Infelizmente, as funções de dispersão perfeitas mínimas são *extremamente* raras. Por isso, na prática utilizam-se funções não-mínimas com um contradomínio de tamanho aceitável, que desperdice pouca memória.

Este trabalho utiliza dispersão 3-D perfeita para representar *octrees* através da técnica exposta na Seção 3.3. Neste esquema, a função $H(\mathbf{k})$ é gerada para um conjunto *estático* de células, de forma que cada chave $\mathbf{k}$ seja mapeada para um *texel* diferente da textura de células. As subseções seguintes oferecem uma visão geral de como as funções de dispersão 3-D perfeitas são geradas, e como elas são utilizadas em GPU.

PUC-Rio - Certificação Digital Nº 0611941/CA

3.4.1 Função de Dispersão Espacial Perfeita

O esquema de dispersão espacial perfeita utilizado neste trabalho é baseado no método proposto por Lefebvre e Hoppe [20]. Este método, projetado especialmente para GPUs, permite armazenar um conjunto de dados esparsos em uma textura compacta através de uma função de dispersão 2-D ou 3-D. A função funciona com um número fixo de operações aritméticas e um único acesso a textura, o que a torna extremamente eficiente. No artigo [20], Lefebvre e Hoppe relatam que o método é capaz de compactar 562.912 elementos espalhados por uma textura de 512^3 em uma textura densa de 45^3.

As funções de dispersão espacial são definidas em um domínio $\mathbf{U}$ — uma grade d-dimensional com $u = \bar{u}^d$ elementos, denotada por $\mathbb{Z}_{\bar{u}}^d = [0, \bar{u} - 1]^d$. Toda função de dispersão perfeita $h(\mathbf{p})$ é restrita a um subconjunto $\mathbf{S} \subset \mathbf{U}$ de n elementos, onde cada posição $\mathbf{p} \in \mathbf{S}$ possui um valor associado $D(\mathbf{p})$. O objetivo da função de dispersão é mapear o conjunto de valores esparsos $D(\mathbf{p})$ em uma textura compacta $\mathbf{H}[h(\mathbf{p})]$, onde:

- A textura $\mathbf{H}$ é um vetor d-dimensional, de tamanho $m = \bar{m}^d \geq n$, que contém todos os valores $\{D(\mathbf{p}) : \mathbf{p} \in \mathbf{S}\}$.
- A função $h(\mathbf{p}) : \mathbf{U} \rightarrow \mathbf{H}$ é injetora quando restrita a $\mathbf{S}$. Ou seja, cada ponto $\mathbf{p} \in \mathbf{S}$ é mapeado para uma posição única em $\mathbf{H}$.

A função de dispersão espacial perfeita proposta por Lefebvre e Hoppe [20] é dada por

$$h(\mathbf{p}) = (h_0(\mathbf{p}) + \mathbf{\Phi}\,[h_1(\mathbf{p})]) \mod \bar{m} \tag{3-3}$$

onde:

- A textura $\mathbf{\Phi}$, denominada *textura de solução*, é um vetor d-dimensional, de tamanho $r = \bar{r}^d$, que contém vetores d-dimensionais.
- A função $h_0(\mathbf{p})$ mapeia o ponto $\mathbf{p}$ do domínio $\mathbf{U}$ para a textura $\mathbf{H}$, utilizando apenas *endereçamento modular* ($\mathbf{p} \mod \bar{m}$).
- A função $h_1(\mathbf{p})$ mapeia o ponto $\mathbf{p}$ do domínio $\mathbf{U}$ para a textura $\mathbf{\Phi}$, também utilizando endereçamento modular ($\mathbf{p} \mod \bar{r}$).

Para que este método de dispersão espacial funcione, é preciso encontrar uma textura $\mathbf{\Phi}$ de tamanho $r = \bar{r}^d$ que resolva todas as colisões de $h(\mathbf{p}), \mathbf{p} \in \mathbf{S}$. Como ilustrado na Figura 3.4 (à direita), cada elemento de $\mathbf{\Phi}$ guarda uma translação que é aplicada aos pontos $\{h_1^{-1}(\mathbf{q}) \subset \mathbf{S}\}$. Para uma textura $\mathbf{\Phi}$ suficientemente grande, essa translação pode resolver todas as colisões da função de dispersão. A idéia é que o tamanho $\bar{m}$ da textura $\mathbf{H}$ seja sempre o menor possível para acomodar os n elementos de $\mathbf{S}$, mas que a textura $\mathbf{\Phi}$ possa crescer até um tamanho que permita uma solução para todas as colisões. Como um *texel* de $\mathbf{\Phi}$ tem normalmente um custo de memória menor que um *texel* de $\mathbf{H}$, é geralmente preferível que $\bar{r}$ cresça, e não $\bar{m}$. Não obstante, quase sempre a textura $\mathbf{\Phi}$ permanece menor que $\mathbf{H}$.

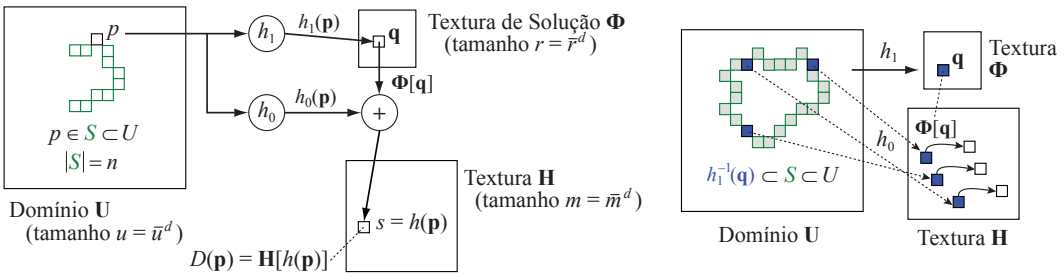

Figura 3.4: Método de dispersão perfeita proposto por Lefebvre e Hoppe [20].

A implementação da função de dispersão $h(\mathbf{p})$ em GPU requer um número fixo de instruções e um único acesso a textura. Todas as operações de módulo podem vir de graça se as texturas forem configuradas para o modo de endereçamento "*wrap*". Observe que, neste esquema, a "textura de células" referida na Seção 3.3 seria $\mathbf{H}$, enquanto que a "textura de dispersão" seria $\mathbf{\Phi}$.

Apesar de funcionar relativamente bem para a compactação de texturas — onde os domínios são geralmente menores que 512^3 — este método não

PUC-Rio - Certificação Digital Nº 0611941/CA

escala bem para domínios maiores, que possam conter *qualquer* inteiro. Em particular, esta limitação compromete o uso do método para a representação de *octrees* (Seção 3.3), já que o domínio de uma *octree* baseada em dispersão espacial cresce muito rápido — *e.g.* 1024^3 para uma *octree* de altura 10 e 4096^3 para uma *octree* de altura 12.

A razão da baixa escalabilidade da função de dispersão proposta por Lefebvre e Hoppe é que, sempre que dois pontos $\mathbf{p}_1, \mathbf{p}_2 \in \mathbf{S}$ forem divisíveis simultaneamente por $\bar{m}$ e $\bar{r}$, não existirá instância de $\mathbf{\Phi}$ com tamanho r que torne a função injetora. Nesses casos, a estratégia proposta seria aumentar $\bar{r}$, mas para domínios grandes e com muitos elementos, é comum que não exista valor de $\bar{r}$ aceitável que satisfaça este critério. Lefebvre e Hoppe mencionam no trabalho [20] que domínios grandes, onde $\bar{u} > \bar{m}\bar{r}$, podem gerar problemas; mas argumentam que, na prática, $\bar{r}$ é sempre grande o suficiente para que $\bar{u} \leq \bar{m}\bar{r}$. No entanto, este problema foi observado na prática ao se tentar gerar funções de dispersão perfeitas para *octrees* esparsas de altura superior a 8. Para a grande maioria das *octrees*, o algoritmo despendia horas procurando uma instância válida de $\mathbf{\Phi}$, sem sucesso.

Esta dissertação propõe duas alterações simples na função $h(\mathbf{p})$, para permitir que ela funcione mesmo quando duas chaves $\mathbf{p}_1, \mathbf{p}_2 \in \mathbf{S}$ forem simultaneamente divisíveis por $\bar{m}$ e $\bar{r}$. A saber:

- A textura de solução $\mathbf{\Phi}$ passa a armazenar um valor w (escalar) a mais por *texel*. No caso 3-D, a textura passaria a ter quatro canais.
- A função $h_0(\mathbf{p})$, responsável por mapear $\mathbf{p} \in \mathbf{U}$ em $\mathbf{H}$ utilizando $(\mathbf{p} \bmod \bar{m})$, é substituída pela operação $(\mathbf{p} \bmod w)$.

Com as alterações, a função fica da seguinte forma:

$$\begin{aligned} h(\mathbf{p}) &= ((\mathbf{p} \bmod w) + (x, y, z)) \bmod \bar{m}, \\ &\quad \text{onde } (x, y, z, w) = \mathbf{\Phi}\left[h_1(\mathbf{p})\right] \end{aligned}$$

Esta nova função concede um "grau de liberdade" adicional para o mecanismo de dispersão. A chance de encontrar uma função injetora aumenta consideravelmente, enquanto que o custo da função mantém-se praticamente o mesmo. Quando dois pontos $\mathbf{p}_1, \mathbf{p}_2 \in \mathbf{S}$ forem simultaneamente divisíveis por $\bar{m}$ e $\bar{r}$, o valor w pode ser escolhido de forma que $\mathbf{p}_1$ e $\mathbf{p}_2$ não colidam. Com este novo método, o tempo médio de geração para as funções de dispersão perfeitas das *octrees* baixou, de horas, para menos de um minuto.

PUC-Rio - Certificação Digital Nº 0611941/CA

3.4.2
Geração da Função de Dispersão Espacial Perfeita

A geração de uma função de dispersão espacial perfeita pelo método descrito na Seção 3.4.1 envolve encontrar tamanhos $\bar{m}$ e $\bar{r}$, e valores adequados para a textura $\mathbf{\Phi}$, de maneira a tornar a função $h(\mathbf{p})$ injetora quando restrita a $\mathbf{S}$. Naturalmente, deseja-se que $\bar{m}$ e $\bar{r}$ sejam tão pequenos quanto possível.

A estratégia sugerida por Lefebvre e Hoppe [20] é que $\mathbf{H}$ seja sempre a menor textura regular capaz de comportar os n elementos de $\mathbf{S}$; ou seja, o menor tamanho $\bar{m}$ tal que $m = \bar{m}^d \geq n$. Já o tamanho da textura $\mathbf{\Phi}$ é estabelecido como o menor valor $\bar{r}$ para o qual o *algoritmo de geração* (da função de dispersão perfeita) consegue encontrar uma função injetora.

O problema de encontrar valores para a textura $\mathbf{\Phi}$ é equivalente ao problema de *compressão de matrizes esparsas*, que é NP-completo. Por isso, o algoritmo proposto por Lefebvre e Hoppe [20] baseia-se em uma heurística gulosa e probabilística. A geração de uma textura $\mathbf{\Phi}$ de tamanho $\bar{r}$ pode requerer várias tentativas, especialmente quando $\bar{r}$ é próximo do tamanho ótimo. Uma abordagem possível para a geração da textura $\mathbf{\Phi}$ consiste em começar com um valor de $\bar{r}$ pequeno, fazer múltiplas tentativas para cada $\bar{r}$, e caso uma textura de solução válida não seja encontrada, selecionar um valor de $\bar{r}$ maior e repetir o processo.

O algoritmo guloso sugerido por Lefebvre e Hoppe [20] para preencher a textura $\mathbf{\Phi}$ — ainda sem considerar o componente adicional w, proposto nesta dissertação — segue aos seguintes passos:

1. Uma grade d-dimensional $\mathbf{\Phi}'$, com o mesmo tamanho da textura $\mathbf{\Phi}$, é criada. Cada célula guarda uma lista de pontos e uma translação.

2. Todos os pontos $\mathbf{p} \in \mathbf{S}$ são mapeados para $\mathbf{\Phi}'$ através da função $h_1(\mathbf{p})$.

3. As células da grade $\mathbf{\Phi}'$ são ordenadas de acordo com o número de pontos mapeados para cada uma delas; ou seja, $|h_1^{-1}(\mathbf{q})|$.

4. Uma grade d-dimensional $\mathbf{H}'$, com o mesmo tamanho da textura $\mathbf{H}$, é criada. Cada célula guarda um booleano que indica se ela já foi ocupada. Todas as células são inicialmente marcadas 'vazias'.

5. Para cada célula de $\mathbf{\Phi}'$, em ordem decrescente de "pontos mapeados", busca-se uma translação $\mathbf{t}$ que, quando aplicada ao pontos $\{h_1^{-1}(\mathbf{q})\}$ através da função $h(\mathbf{p})$, mapeie-os para células 'vazias' de $\mathbf{H}'$.

   – A busca começa por um $\mathbf{t}$ aleatório e depois prossegue linearmente.

PUC-Rio - Certificação Digital Nº 0611941/CA

- Caso uma translação válida seja encontrada, ela é salva na célula. As células de $\mathbf{H}'$ ocupadas pelos pontos são marcadas 'ocupadas'.
- Caso nenhuma translação seja válida, o algoritmo falha.

6. A textura $\mathbf{\Phi}$ é preenchida com as translações salvas em $\mathbf{\Phi}'$.

A idéia deste algoritmo guloso é tentar resolver os casos mais difíceis primeiro; ou seja, achar valores para as células de $\mathbf{\Phi}'$ que têm mais pontos. No último estágio, quando $|h_1^{-1}(\mathbf{q})| = 1$, a solução é trivial e o algoritmo não pode mais falhar. Esta heurística aumenta consideravelmente as chances de sucesso, apesar de o algoritmo ainda depender da sorte para que as translações escolhidas nos primeiros passos "se encaixem". Observe que, em geral, ao aumentar $\bar{r}$, o número de pontos mapeados em cada célula de $\mathbf{\Phi}'$ diminui, e a chance de sucesso do algoritmo aumenta.

Quando o componente w é adicionado à textura $\mathbf{\Phi}$, um novo passo precisa ser acrescentado ao algoritmo que gera $\mathbf{\Phi}$. Para que as translações de $\mathbf{\Phi}'$ sejam calculadas, as células já devem ter um valor de w associado. Por isso, o novo passo deve ser introduzido imediatamente após o passo 3. Para cada célula, o valor de w deve ser determinado pelo seguinte algoritmo:

Algoritmo 3.1 Cálculo do componente w da textura $\mathbf{\Phi}$.

```
 1: function COMPUTE_W(C_Φ)                          ▷ calcula w para uma célula C_Φ
 2:     if not CONTAIN_COLLISIONS(C_Φ, m̄) then        ▷ testa um w neutro
 3:         return m̄                                  ▷ a célula não contém colisões
 4:     end if
 5:     w ← 1                ▷ procura por um valor de w que resolva as colisões
 6:     repeat
 7:         w ← NEXT_COPRIME(w, m̄, r̄)
 8:         if w > m̄ then
 9:             ERROR("nenhum valor de w pode resolver as colisões")
10:         end if
11:     until not CONTAINS_COLLISIONS(C_Φ, w)
12:     return w
13: end function
```

A função CONTAINS_COLLISIONS(C_Φ, w) testa se, utilizando um certo valor de w na função de dispersão, uma certa célula de $\mathbf{\Phi}'$ contém colisões. Já a função NEXT_COPRIME(c, a, b) retorna o primeiro número maior que c que não seja divisível nem por a nem por b. Os valores de w que podem resolver as colisões dentro de uma célula são sempre co-primos de $\bar{m}$ e $\bar{r}$; por isso, a função só testa valores que satisfaçam este critério.

PUC-Rio - Certificação Digital Nº 0611941/CA

3.4.3
Reconhecimento de Células Inexistentes

A função de dispersão espacial $h(\mathbf{p})$, definida na Seção 3.4.1, é injetora somente quando restrita ao conjunto de pontos para o qual ela foi gerada. Caso a função seja usada com qualquer outro ponto $\mathbf{v} \notin \mathbf{S}$, o mapeamento é feito para a posição de um ponto $\mathbf{p} \in \mathbf{S}$ "aleatório". Este esquema é suficiente quando se tem garantia de que as chaves consultadas serão sempre "válidas". Porém, na implementação de uma *octree* baseada em dispersão espacial, deve ser possível consultar se uma determinada célula da *octree* existe ou não. Infelizmente, a função $h(p)$ não dispõe de informação suficiente para gerar esta resposta.

Este problema pode ser solucionado de forma simples se os elementos do domínio forem armazenados no contradomínio da função. Ou seja, se os pontos $\mathbf{p} \in \mathbf{S}$ forem armazenados nos *texels* correspondentes da textura $\mathbf{H}$. Desta forma, é possível verificar se um ponto $\mathbf{p}$ consultado pertence ou não a $\mathbf{S}$, testando se o *texel* $\mathbf{H}[h(\mathbf{p})]$ contém $\mathbf{p}$.

Na implementação de uma *octree* baseada em dispersão espacial, a solução análoga seria armazenar as chaves das células, geradas pela função $S(\ell, \mathbf{p})$, na "textura de células". Porém, para poupar memória, ao invés de armazenar as chaves diretamente, a solução adotada armazena *identificadores* que são gerados a partir das chaves das células. Esta técnica requer 4 *bytes* por célula da *octree*, ao invés dos 12 necessários para armazenar uma chave.

3.5
Reconstrução do Campo de Distância

O primeiro passo no sentido de reconstruir o campo de distância em um certo ponto $\mathbf{p}$ é encontrar a menor célula da ADF que contenha $\mathbf{p}$. Uma vez encontrada esta célula, o valor de distância no ponto pode ser obtido trivialmente através de interpolação trilinear com os *voxels* da célula. Esta interpolação pode ser feita praticamente de graça, pela GPU, se a opção de filtragem trilinear estiver habilitada para a textura de *voxels*.

Encontrar a menor célula da ADF que contenha $\mathbf{p}$ envolve uma busca na *octree*. Graças à representação baseada em dispersão espacial — que permite acesso aleatório às células — esta busca pode ser feita de múltiplas formas. Em todos os casos, a função $S(\ell, \mathbf{p})$ é usada para obter a célula de nível ℓ que contém $\mathbf{p}$. O objetivo é encontrar o maior valor de ℓ para o qual exista uma célula. Três alternativas foram testadas:

1. Busca linear.

2. Busca binária.

PUC-Rio - Certificação Digital Nº 0611941/CA

3. Busca linear com coerência.

Abaixo encontram-se pseudo-códigos para estes algoritmos. Por questões de simplicidade, os códigos desprezam o fato de que as células-folha da *octree* são mantidas implicitamente.

Busca Linear: começa pelo primeiro nível da *octree* ($\ell = 1$) e vai descendo um nível de cada vez, até encontrar a menor célula que contenha $\mathbf{p}$.

```
1: function QUERY(p)                    ▷ busca a menor célula que contém p
2:     v ← ∅                            ▷ menor célula encontrada até o momento
3:     for ℓ ← 1, ∞ do
4:         k ← S(ℓ, p)                  ▷ chave para a célula de nível ℓ que contém p
5:         v' ← H(k)                    ▷ acesso à textura de células
6:         if not EXISTS(v', k) then    ▷ verifica se a célula existe
7:             break                    ▷ caso não existam mais células, pára
8:         end if
9:         v ← v'                       ▷ atualiza a menor célula
10:    end for
11:    return v                         ▷ retorna a menor célula
12: end function
```

Este é o mais simples dos três algoritmos testados. Alternativamente, este algoritmo poderia começar a busca pelo nível máximo da *octree*, e retornar a primeira célula encontrada de baixo para cima.

Busca Binária: realiza uma busca binária entre o primeiro nível e o nível máximo da octree, em busca do nível da menor célula.

```
1: function QUERY(p)                    ▷ busca a menor célula que contém p
2:     v ← ∅                            ▷ menor célula encontrada até o momento
3:     ℓ_min ← 1                        ▷ menor nível válido da ADF
4:     ℓ_max ← maxlevel                 ▷ maior nível válido da ADF
5:     while ℓ_min ≤ ℓ_max do
6:         ℓ ← ½(ℓ_min + ℓ_max)
7:         k ← S(ℓ, p)                  ▷ chave para a célula de nível ℓ que contém p
8:         v' ← H(k)                    ▷ acesso à textura de células
9:         if EXISTS(v', k) then        ▷ verifica se a célula existe
10:            v ← v'                   ▷ célula existe; atualiza a menor célula
11:            ℓ_min ← ℓ + 1            ▷ busca por células abaixo do nível ℓ
12:        else                         ▷ nenhuma célula encontrada neste nível
13:            ℓ_max ← ℓ − 1            ▷ busca por células acima do nível ℓ
14:        end if
```

PUC-Rio - Certificação Digital Nº 0611941/CA

```
15:     end while
16:     return v                                   ▷ retorna a menor célula
17: end function
```

A busca binária pode oferecer bom desempenho para árvores muito altas. Porém, como as ADFs dificilmente ultrapassam o nível 12, constatou-se que não há vantagem na busca binária. Testes com ADFs de nível 8 mostraram que, em CPU, a busca binária é tão rápida quanto a busca linear. Em GPU, porém, a busca binária foi mais lenta que a linear.

Busca Linear com Coerência: realiza uma busca linear a partir de um nível ℓ qualquer. Se uma célula existir no nível ℓ, a busca continua a descer até encontrar a menor célula. Caso contrário, a busca sobe nível a nível e retorna a primeira célula encontrada de baixo para cima.

```
1: function QUERY(p, ℓ)
2:     k ← S(ℓ, p)              ▷ chave para célula de nível ℓ que contém p
3:     v ← H(k)                                  ▷ acesso à textura de células
4:     if EXISTS(v, k) then                      ▷ verifica se a célula existe
5:         while ℓ < maxlevel do   ▷ a célula existe; tenta descer mais
6:             k ← S(ℓ + 1, p)     ▷ tenta obter a célula um nível abaixo
7:             v′ ← H(k)
8:             if EXISTS(v′, k) then
9:                 ℓ ← ℓ + 1                  ▷ atualiza último nível válido
10:                v ← v′                          ▷ atualiza menor célula
11:            else
12:                break                ▷ não existem mais células abaixo
13:            end if
14:        end while
15:    else                                        ▷ a célula não existe
16:        while ℓ > 0 do   ▷ busca a menor célula de baixo para cima
17:            ℓ ← ℓ − 1          ▷ tenta obter uma célula um nível acima
18:            k ← S(ℓ, p)
19:            v ← H(k)
20:            if EXISTS(v, k) then
21:                break                      ▷ menor célula encontrada
22:            end if
23:        end while
24:    end if
25:    return v, ℓ       ▷ retorna a menor célula e o último nível válido
26: end function
```

PUC-Rio - Certificação Digital Nº 0611941/CA

Esta técnica permite que os algoritmos explorem coerência espacial. Por exemplo, considere um algoritmo de lançamento de raios para visualização volumétrica que precise amostrar o campo de distância em intervalos regulares ao longo de um raio. Este algoritmo pode guardar o nível da célula utilizada em cada passo, e no passo seguinte recomeçar a busca a partir desse nível. Para ADFs de nível 9, esta estratégia mostrou ser em torno de 30% mais rápida que uma busca linear comum. A implementação real desta função é feita em GLSL (*OpenGL Shading Language*) e encontra-se disponível nos apêndices.

As funções acima retornam o endereço $\mathbf{v}$ do bloco de *voxels* da menor célula que contém um ponto $\mathbf{p}$. Esse endereço é um vetor 3-D, e indica a posição central do primeiro *texel* do bloco de *voxels* da célula. Com a opção de filtragem trilinear habilitada para a textura de *voxels*, o seguinte comando GLSL pode ser utilizado para reconstruir o valor de distância no ponto $\mathbf{p}$:

$$\texttt{texture3D}\left(voxelTexture, \frac{\mathbf{v} + \mathbf{p}_{cell}}{\mathbf{voxelTextureDim}}\right) \tag{3-4}$$

onde $\mathbf{p}_{cell}$ é o ponto $\mathbf{p}$ transformado para o espaço local da célula, e **voxelTextureDim** são as dimensões da textura de voxels.

PUC-Rio - Certificação Digital Nº 0611941/CA