5
Sistema Proposto

O sistema proposto foi construído para funcionar inteiramente no processador gráfico. Não há necessidade de transferência de dados entre a CPU e a GPU. Para isso, o sistema requer uma série de funcionalidades presentes apenas nos processadores gráficos mais modernos (*buffer objects*, texturas em ponto flutuante, *multiple rendering targets*, acesso a textura no programa de vértices e *occlusion queries*).

O sistema evolui em passos discretos. Um passo de simulação é composto pelos seguintes estágios:

- Integração numérica
- Construção da estrutura de *grid*
- Detecção e resposta à colisão entre partículas
- Aplicação das restrições
- Detecção e resposta à colisão com o ambiente

Depois de completar um passo de simulação e todas as partículas estarem com suas posições atualizadas, pode-se opcionalmente começar a fase de desenho: as partículas são enviadas através do *pipeline* gráfico para compor a imagem correspondente.

5.1.
Simulação

5.1.1.
Integração Numérica

O primeiro estágio do passo de simulação é responsável por atualizar a posição das partículas no tempo, integrando a equação de movimento. Este estágio também é responsável por gerenciar o nascimento e a morte de partículas, se um emissor de partículas estiver ativo.

O estado das partículas é armazenado numa textura *RGBA-float*, chamada *textura de estado*. A posição é armazenada nos canais RGB. Dois valores adicionais são compactados no canal alfa: o raio da partícula, usado no estágio de colisão; e o tempo de vida da partícula, usado para a gerência de nascimento e morte da partícula. O tempo de vida da partícula, *L*, é atualizado a cada passo de acordo com a seguinte equação:

$$L_{t+\Delta t} = L_t - \Delta t \qquad (8)$$

A integração numérica é feita em uma passada. Um programa de fragmentos é responsável por computar a nova posição da partícula de acordo com a Equação 7 e por atualizar o tempo de vida de acordo com a Equação 8. O programa de fragmentos acessa duas texturas de entrada, o estado atual e o estado anterior, e gera uma textura de saída, implementando assim um esquema de buffer triplo.

Para diminuir a computação nos próximos estágios, este programa de fragmentos também computa a célula a qual a partícula, já atualizada, pertence. Esta informação é armazenada em outra textura RGBA, chamada *textura info-cel*: a posição da célula no *grid* (I, J, K) usa os canais RGB e o ID da célula usa o canal alfa. Nos próximos estágios, o ID da célula é usado para acessar os dados da célula e a posição da célula (I, J, K) é usada para acessar a vizinhança.

5.1.2. Construção da Estrutura de *Grid*

O método mais intuitivo de detecção de colisão entre partículas resulta num algoritmo quadrático, o que é inaceitável para aplicações interativas com uma grande quantidade de partículas. O sistema proposto segue a idéia de subdividir o espaço em um *grid* regular, com o tamanho das células igual ao diâmetro da maior partícula. As partículas são inseridas numa lista mantida na célula que contém seus centros. Com isso, uma dada partícula só colide com partículas inseridas na mesma célula ou com partículas inseridas nas células vizinhas.

A montagem do *grid* é feita em múltiplas passadas de renderização. A computação é feita num programa de vértices, pois é necessária a capacidade de realizar operações de armazenamento randômico (*scatter*): cada partícula é transformada para a posição correta na textura de saída. A textura de saída

compõe o *grid* e o vetor auxiliar numa textura 2D, chamada *textura de grid*, que representa conceitualmente um vetor unidimensional. Se *ncells* e *nparticles* representam, respectivamente, o número de células e o número de partículas, a dimensão deste vetor unidimensional é *ncell* + *nparticles*: as primeiras *ncell* posições da textura são usadas para armazenar o *grid* e as outras posições para armazenar o vetor auxiliar com as listas encadeadas.

Na primeira passada a *textura info-cel*, gerada no passo de integração, é usada como entrada e uma *textura de grid* é usada como saída. As partículas são desenhadas como pontos. Para cada ponto correspondente a uma partícula, o programa de vértices consulta a *textura info-cel* e usa o ID da célula para transformar o ponto corretamente. O ponto é mapeado para a posição da célula na *textura de grid*. Nesta primeira passada, se duas ou mais partículas são mapeadas para uma mesma célula, apenas uma delas permanece na *textura de grid*.

Para lidar com duas ou mais partículas mapeadas na mesma célula, passadas adicionais são feitas. Nestas passadas, a última *textura de grid* produzida é também usada como entrada. Então, para cada partícula, o programa de vértices consulta a *textura info-cel* para acessar o ID da célula e então, usando a última *textura de grid* gerada, checa qual partícula está armazenada nesta célula. Se a partícula armazenada for a mesma partícula que está sendo considerada, nada deve ser feito. Se na primeira passada uma partícula diferente foi armazenada, seu ID é usado para acessar o vetor auxiliar, e mais uma vez checar se há alguma partícula armazenada nesta posição. Este vetor continua sendo percorrido até que a própria partícula seja encontrada ou um espaço vazio seja achado, onde a partícula corrente será colocada (como já mostrado na Figura 4).

Passadas adicionais são feitas até que todas as partículas sejam armazenadas. Para sabermos se uma passada adicional é necessária, toda vez que uma partícula é inserida no vetor auxiliar, seu valor de z é definido como um. Após cada passada, um consulta de oclusão (*occlusion query*) é feita para saber se algum pixel é igual a um. Se a região do *z-buffer* correspondente ao vetor auxiliar contiver apenas zeros, todas as partículas já foram mapeadas corretamente para suas posições, e não precisamos mais de nenhuma passada. A Figura 8 ilustra uma construção e *grid* em três passadas, mostrando o resultado do *grid* e o *z-buffer* correspondente.

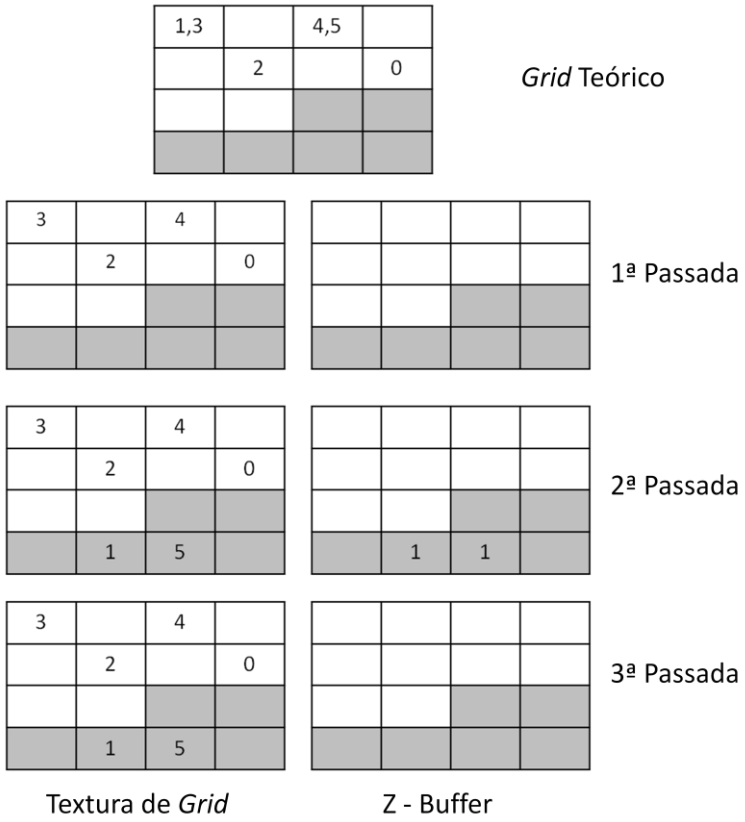

Figura 8 Ilustração de uma montagem de *grid* em três passadas. As células em cinza representam as listas. Na primeira passada as partículas são mapeadas para sua posição no grid. Na segunda passada, a partícula 1 é inserida na posição 3 do vetor auxiliar e a partícula 5 na posição 4, definindo o *Z-Buffer n*essas posições como um. Na terceira passada, como todas as partículas já foram inseridas, o *Z-Buffer* permanece apenas com zeros, o que indica o fim do processo.

5.1.3. Detecção e resposta a colisão entre partículas

Depois da estrutura de *grid* montada, a detecção e a resposta a colisão entre as partículas pode ser eficientemente computada. Isto é feito por um programa de fragmentos que recebe como entrada o estado das partículas atualizado e gera outro estado atualizado. A resposta a colisão é baseada numa colisão totalmente inelástica, sem atrito. O objetivo é deslocar cada partícula até que não exista mais interpenetração. Isso é obtido iterativamente aplicando o método de relaxação [8]. Idealmente, deveríamos encontrar o primeiro par de partículas que estão colidindo e aplicar o deslocamento, metade para cada partícula, necessário para eliminar a

interpenetração. Então, procurar o próximo par, já considerando a posição atualizada do primeiro par detectado. Porém, este algoritmo não se encaixa no modelo de computação paralela da GPU. Então, o seguinte método foi usado: para cada partícula, são detectadas todas as outras partículas que estão colidindo com ela, calcula-se o deslocamento para separar cada par, soma-se metade desses deslocamentos e atualiza-se a posição da partícula corrente (Figura 9). Para acessar as partículas nas células vizinhas, o programa de fragmentos também recebe como entrada a *textura info-cell* e a *textura de grid*.

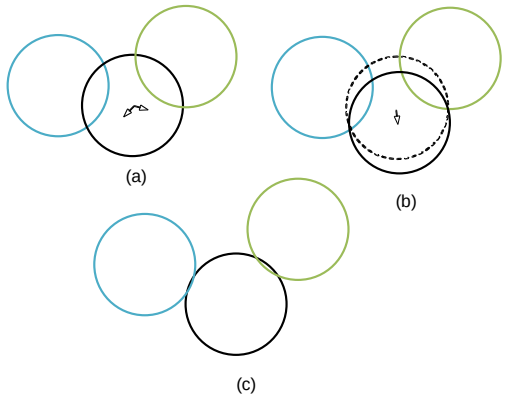

Figura 9 Resposta à colisão entre partículas: (a) para a partícula corrente, calcula o deslocamento necessário para separar cada par de partículas; (b) soma metade dos deslocamentos e atualiza a partícula; e (c) no fim da iteração, todas as partículas são deslocadas.

Para melhorar a convergência do método da relaxação, o deslocamento feito nas partículas é feito em três passadas. Na primeira passada, cada partícula é deslocada considerando apenas as 8 vizinhas que compartilham um vértice com a célula da partícula corrente. Na segunda passada, o deslocamento é feito considerando as 12 vizinhas que compartilham uma aresta com a célula. Este deslocamento já leva em conta as posições atualizadas pela primeira passada. Finalmente, na terceira passada, as 6 células que compartilham uma face com a célula da partícula corrente, mais a própria célula, são consideradas. Esta estratégia de três passadas reduz o problema de mapear o método de relaxação para a GPU. O ganho em convergência compensa o pequeno impacto no desempenho pelas passadas adicionais.

Após cada passo a estrutura de *grid* não é atualizada, com isso podemos deixar de detectar potenciais colisores. Estas três passadas são apenas uma iteração do método de relaxação, e todas as iterações são feitas sem a atualização

do *grid*. Na prática, as colisões não detectadas não representam um problema com taxas de quadros por segundo altas, pois o *grid* será recomputado no próximo passo de simulação. Na verdade, o método de relaxação funciona até com um número pequeno de iterações, pois o processo continua no próximo passo. Em alguns experimentos conseguimos estabilidade usando apenas uma iteração.

5.1.4. Aplicação das restrições

O sistema permite ainda que a aplicação adicione restrições às partículas. Até agora, foram implementada restrições de barra rígida. Este tipo de restrição mantém duas partículas separadas por uma dada distância. Isso permite criar aproximações para partículas não esféricas usadas por algumas aplicações, como simulação de materiais granulares [3]. O formato não esférico é obtido pelo agrupamento de várias partículas ligadas por restrições de barra rígida. A Figura 10 mostra algumas das configurações testadas. Outras restrições, como manter uma partícula a uma distância fixa de um plano ou restringir uma partícula a mover-se ao longo de uma trajetória também podem ser implementadas seguindo o mesmo modelo.

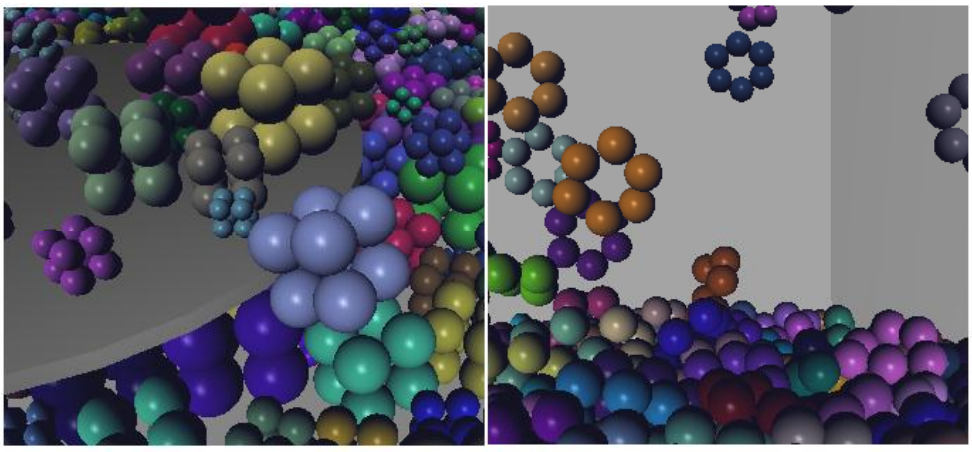

Figura 10 Criando partículas não esféricas por meio de agrupamento.

As restrições também são resolvidas por meio de relaxação. Em cada passada, após aplicar o deslocamento provocado pela colisão entre partículas, uma passada extra é feita para satisfazer as restrições de barra rígida.

Para cada partícula é armazenada uma lista de todas as restrições associadas a ela. As restrições são armazenadas em duas texturas. A primeira textura tem o mesmo tamanho da *textura de estado* e armazena dois valores: o número de

restrições anexadas à partícula e a posição da primeira restrição na segunda textura. A segunda textura possui os dados das restrições. Para a restrição de barra rígida, apenas dois valores são necessários: o tamanho da barra e o ID da outra partícula conectada por esta restrição.

Estas duas texturas, junto com a última *textura de estado*, são acessadas por um programa de fragmentos que retorna uma *textura de estado* atualizada, resolvendo as restrições. Para cada partícula, o programa de fragmentos soma os deslocamentos aplicados para que todas as restrições sejam atendidas. De novo, são necessárias apenas algumas iterações para alcançar resultados satisfatórios, já que a relaxação continua no passo seguinte.

5.1.5. Detecção e resposta a colisão com o ambiente

Para cada iteração do método de relaxação, passadas adicionais são feitas para lidar com a colisão das partículas com o ambiente. O ambiente é modelado instanciando obstáculos. Cada primitiva de obstáculo representa um objeto simples, para que a distância de uma partícula até a superfície do obstáculo possa ser facilmente calculada. Mesmo com primitivas simples, podemos modelar ambientes relativamente complexos, graças a um modelo de composição flexível. Também existe suporte a colisão contra terrenos, expressados por meio de mapas de alturas armazenados em textura.

Para cada obstáculo, um programa de fragmentos verifica se a posição da partícula é inválida. Se for, a partícula é projetada para uma posição válida, gerando assim uma *textura de estado* atualizada. A projeção é feita na direção normal da superfície computada na posição corrente da partícula. Para terrenos, a normal é explicitamente armazenada na textura. Os canais RGB são usados para armazenar a normal e o canal alfa para armazenar o mapa de altura.

5.2. Modelagem do Ambiente

O ambiente onde a simulação ocorre é caracterizada pela instanciação de emissores de partículas e pela modelagem dos obstáculos.

5.2.1.
Emissores de Partículas

A CPU é usada, em pré-processamento, para gerar a posição inicial das partículas. Quando um emissor de partículas é instanciado, é gerada uma textura chamada *textura de emissão*. Esta textura é passada como entrada para o estágio de integração numérica, que também é responsável pela gerencia de nascimento e morte das partículas. A seguinte metodologia é seguida: no começo da simulação, o nascimento das partículas é controlado por uma única variável uniforme no shader. As partículas são armazenadas na *textura de estado* ordenada pelo seu tempo de nascimento. Esta variável controla a quantidade máxima de partículas em um determinado quadro. A variável é incrementada, aumentando o número máximo de partículas na simulação até que todas as partículas na textura sejam consideradas. O tempo de vida de cada partícula é decrementado a cada passo de simulação, até chegar a um valor negativo. Neste momento, a partícula morre e renasce em sua posição original.

5.2.2.
Modelagem dos Obstáculos

Uma funcionalidade importante do sistema proposto é a de criar obstáculos a partir de objetos simples, sem ter que recodificar os *shaders*. Um obstáculo simples é caracterizado por uma superfície simples ou discretizada, como um plano, um cone, uma esfera, um cilindro, uma caixa ou um terreno. A cena pode ser modelada combinando esses diferentes tipos de obstáculos.

Para dar mais poder de expressão ao processo de modelagem, foi adicionada a opção de cada obstáculo possuir uma lista de *condições*. Condições são representadas por um conjunto de outras superfícies que indicam se um dado obstáculo deve ou não ser considerado. Esta decisão é tomada partícula a partícula, de acordo com suas posições. Como exemplo, pode-se construir um obstáculo como uma esfera com um furo embaixo. Para isso, basta definir uma esfera como um obstáculo sujeito a uma condição de plano. A esfera é considerada um obstáculo somente para partículas que estão, por exemplo, no lado positivo do plano.

A funcionalidade de adicionar condições aos obstáculos, junto com a funcionalidade de combinar diferentes obstáculos, mostrou ser bastante flexível na criação de diferentes cenas. A Figura 11 mostra alguns exemplos. Na esquerda, dois obstáculos, uma caixa e um cilindro, foram combinados para criar uma caixa com cantos arredondados. No centro, um cilindro com um furo foi criado anexando um pequeno cilindro e um plano como condições a um cilindro maior. Na direita, uma ampulheta foi criada combinando dois troncos de cone, que foram criados submetendo um cone a uma condição de plano.

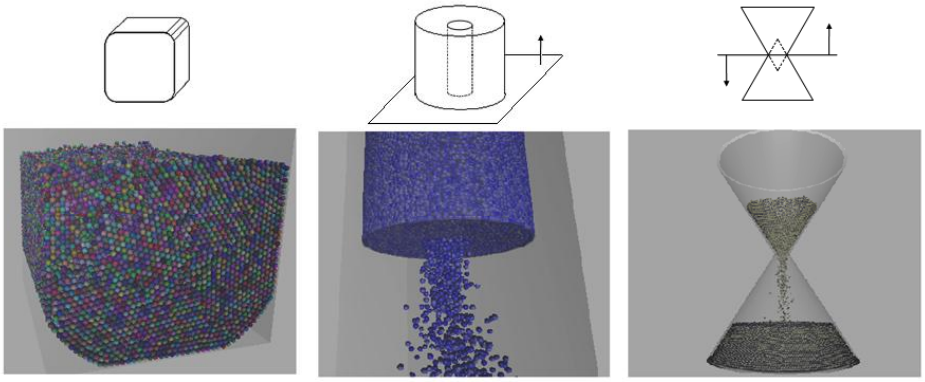

Figura 11 Exemplos de obstáculos e condições.

A vantagem dos obstáculos e condições, como proposto, é que o tratamento de colisão entre partículas e o ambiente é facilmente alcançado com um conjunto de passadas. Cada obstáculo do ambiente é composto pelo próprio obstáculo mais um conjunto de condições. Na primeira passada, um programa de fragmentos valida a posição de todas as partículas de acordo com o obstáculo considerado. Cada partícula do lado inválido do obstáculo é projetada para uma posição válida através da normal da superfície. Este programa de fragmentos gera uma nova *textura de estado*. Se nenhuma condição for anexada ao obstáculo, esta textura representa o estado atualizado e é enviada ao próximo obstáculo, se este existir, até que todos os obstáculos sejam considerados. Se existir alguma condição anexada ao obstáculo, novas passadas de *rendering* são feitas. Um programa de fragmentos tem como entrada a *textura de estado* antes do obstáculo ser considerado. Este programa escreve na mesma textura usada na passada anterior (sem limpá-la, então cada texel possui a posição atualizada considerando o obstáculo). Se a condição não for satisfeita, o estado de entrada é copiado para a saída, sobrescrevendo a posição calculada na primeira passada (preservando a

posição original). Se a condição for satisfeita, o fragmento é descartado, validando então a posição computada pelo obstáculo considerado.

5.3. Fluxograma e Análise de Memória

A Figura 12 ilustra o fluxograma completo do sistema. As caixas brancas centrais representam os programas de fragmentos, exceto *Grid construction*, que representa um programa de vértices. Objetos de textura são representados pelas caixas coloridas.

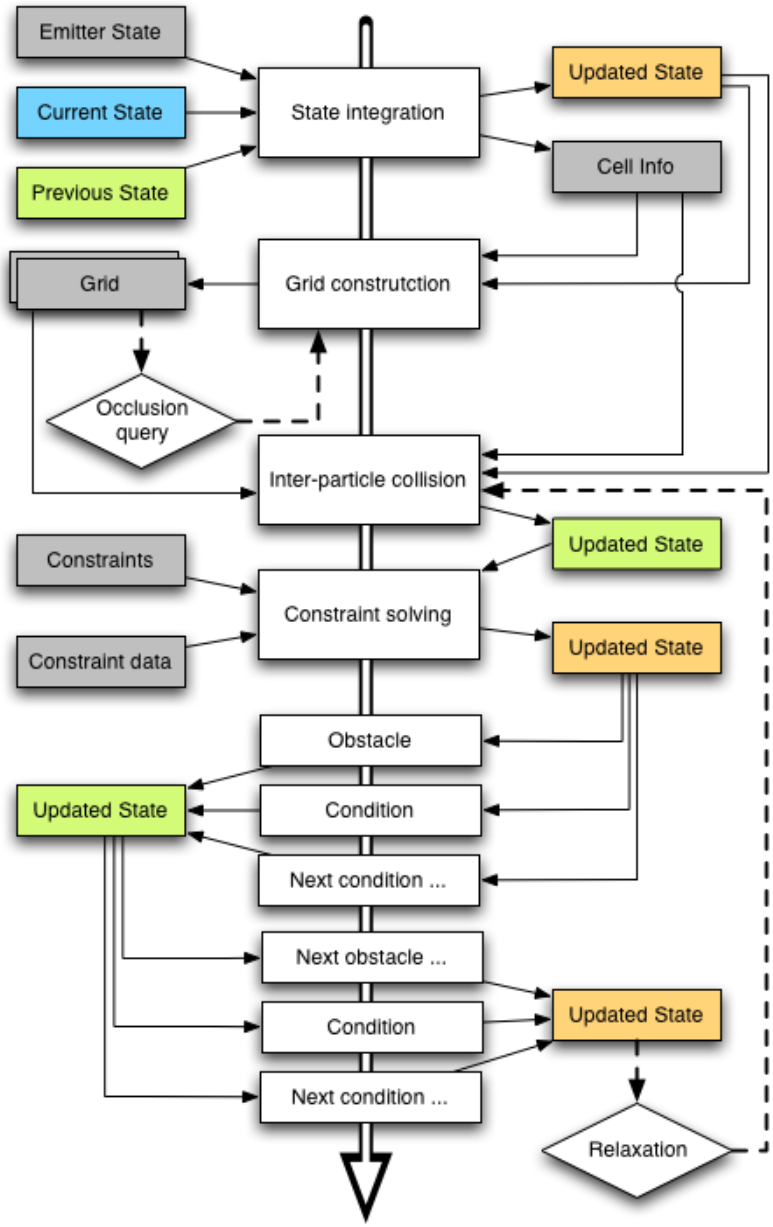

Figura 12 Fluxograma do sistema.

As cores mostradas nas texturas de estados ajudam a visualizar a necessidade de três objetos de textura na simulação. O primeiro programa de fragmentos recebe duas texturas de estado como entrada e gera uma terceira. A partir deste ponto, cada programa de fragmentos alterna entre a terceira textura e a textura usada para representar o estado anterior. No fim, o estado corrente vira o anterior, o último estado atualizado se torna o corrente e a outra textura fica como saída para o primeiro estágio do próximo passo, implementando assim um esquema de buffer triplo.

O espaço de memória utilizado pode ser medido pelo conjunto de texturas usado para armazenar os dados usados ao longo da simulação. O sistema requer o uso de 9 objetos de textura. Como descrito anteriormente, são usados três texturas de estado para fazer a evolução das partículas. Além disso, são usadas três texturas constantes: uma para armazenar o estado inicial gerado pelo emissor de partículas e duas para armazenar as informações das restrições. O sistema também usa três texturas para a construção do *grid*: uma com a informação das células para cada partícula e duas para armazenar a estrutura do *grid* (uma como entrada e outra como saída do programa de vértices). A Tabela 1 mostra todas as texturas necessárias, com os dados armazenados e suas dimensões.

Tipo	**Nome**	**Dados**	**Dimensão**
Constante	Emissão	*x, y, z, r/L*	# partículas
	Restrição	*index, #bars*	# partículas
	Dados de Barra	*len, particleID*	# restrições
Estado	Anterior	*x, y, z, r/L*	# partículas
	Corrente	*x, y, z, r/L*	# partículas
	Atualizado	*x, y, z, r/L*	# partículas
Cell *grid*	Cell info	*I, J, K, cellID*	# partículas
	Grid (entrada)	*particleID*	# células + # partículas
	Grid (saida)	*particleID*	# células + # partículas

Tabela 1 Objetos de textura usados pelo sistema.

A maior demanda de memória vem das texturas usadas para armazenar a estrutura do *grid*, especialmente quando o domínio considerado é muito grande. Uma alternativa eficiente para o armazenamento do *grid* é armazenar as células em uma tabela de dispersão (hash) [6].