UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Visualização em Tempo Real de Dados Volumétricos Dinâmicos usando Hardware Gráfico

por

ALÉCIO PEDRO DELAZARI BINOTTO

Dissertação submetida à avaliação como
requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. João Luiz Dihl Comba
Orientador

Profa. Dra. Carla Maria Dal Sasso Freitas
Co-Orientadora

Porto Alegre, março de 2003.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Binotto, Alécio Pedro Delazari

Visualização em Tempo Real de Dados Volumétricos Dinâmicos Usando Hardware Gráfico / por Alécio Pedro Delazari Binotto. – Porto Alegre: PPGC da UFRGS, 2003.

73f.: il.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2003. Orientador: Comba, João Luiz Dihl. Co-Orientadora: Freitas, Carla Maria Dal Sasso.

1. Visualização Volumétrica. 2. Compressão de Dados. 3. *Hardware* Gráfico. 4. Texturas 3D. 5. Visualização Científica. I. Comba, João Luiz Dihl. II. Freitas, Carla Maria Dal Sasso. III. Título.

“Se buscas resultados distintos, nunca haja sempre da mesma maneira.”
Albert Einstein (1879-1955)

Dedico este trabalho aos meus pais, Jurandyr e Mercedes, e a Maria Clara

Agradecimentos

Agradeço aos meus pais, Jurandyr e Mercedes, e toda a minha família pelo imensurável e incessante apoio durante esta jornada.

A Maria Clara pelo incentivo e completa compreensão durante este período de extrema dedicação.

A Deus, por ter me guiado e me concedido muita saúde.

Agradeço de maneira veemente aos meus orientadores, Prof. João Comba e Profa. Carla Freitas, pela disponibilidade, presença atuante, incentivo, conhecimento e pela convivência durante estes anos, onde obtive oportunidade de crescer assimilando suas inúmeras lições não só profissionais, mas também de vida. Obrigado realmente por propiciarem este amadurecimento.

Agradeço a todos os professores do Grupo de Computação Gráfica pelos conselhos e questionamentos.

A todos os meus colegas do Grupo de Computação Gráfica, em especial a Wilson Gavião, Isabel Siqueira, Carlos Dietrich, Stefan Zanona e Christian Azambuja pelas dicas técnicas.

A todos os meus amigos, em especial Bruno, Carolina, Cláudio, Clóvis, Délcio, Fábio Braga, Fábio Lobo, César, Dalton, Otávio, Fabrício e Vinícius, muito obrigado.

Agradeço ao CNPq pelo apoio financeiro recebido.

Sumário

Lista de Abreviaturas

API	Application Program Interface
CFD	Computational Fluid Dynamics
Cg	C for Graphics
GPU	Graphics Processing Unit
RGBA	Sistema de cor Red Green Blue Alpha

Lista de Figuras

Lista de Tabelas

Resumo

Dados volumétricos temporais são usados na representação de fenômenos físicos em várias aplicações de visualização científica, pois tais fenômenos são complexos, alteram-se com o tempo e não possuem uma forma de representação definida. Uma solução é usar amostragens sobre um espaço de forma geométrica simples que contém o fenômeno (um cubo, por exemplo), discretizado ao longo de uma grade em células de mesmo formato e usualmente chamado de volume de amostragem. Este volume de amostragem representa um instante da representação do fenômeno e, para representar dados temporais, simplesmente enumera-se tantos volumes quanto forem as diferentes instâncias de tempo. Esta abordagem faz com que a representação seja extremamente custosa, necessitando de técnicas de representação de dados para comprimir e descomprimir os mesmos.

Este trabalho apresenta uma nova abordagem para compressão de volumes de dados temporais que permite a visualização em tempo real destes dados usando *hardware* gráfico. O método de compressão usa uma representação hierárquica dos vários volumes de dados dentro da memória do *hardware* gráfico, referenciados pelo *hardware* como texturas 3D. O método de compressão tem melhor desempenho para dados volumétricos esparsos e com alto grau de coerência (espacial e temporal). A descompressão destes dados é feita por programas especiais que são executados no próprio *hardware* gráfico.

Um estudo de caso usando o método de compressão/descompressão proposto é apresentado com dados provenientes do Projeto MAPEM (Monitoramento Ambiental em Atividades de Perfuração Exploratória Marítima). O objetivo do projeto é propor uma metodologia para o monitoramento dos efeitos das descargas de materiais no ecossistema marinho durante a perfuração de um poço de petróleo. Para estimar certos descarregamentos de fluidos, o projeto usa um simulador CFD que permite mostrar tais descarregamentos, gerando grades planares e uniformes 2D ou 3D em qualquer instante de tempo durante a simulação.

Palavras-Chaves: visualização volumétrica, compressão de dados, *hardware* gráfico, texturas 3D, visualização científica.

TITLE: "REAL-TIME VISUALIZATION OF DYNAMIC VOLUMETRIC DATA USING GRAPHICS HARDWARE"

Abstract

Temporal volumetric data are used in many scientific visualization applications for representing physical phenomena, which may have complex shapes, change with time and do not have a closed representation form. Sampling is one solution that can be used in these cases, using a simple spatial shape that contains the phenomena (e. g. cube), discretized along an uniform grid. This sampling volume represents one instance of the phenomena, and to represent temporal data, one approach is to enumerate as many volumes as time instances. This approach leads to a costly representation, demanding data representation techniques to compress and decompress these data.

This work presents a novel approach to compress temporal volumetric data suitable for real-time volume rendering using graphics hardware. The compression technique uses an hierarchical representation of data inside the graphics board memory (referenced as 3D textures). The compression method has better performance in sparse and highly coherent (spatial or temporal) data sets. The decompression is done by special programs that run inside the graphics board.

A case study using the compression/decompression scheme is presented using data from the MAPEM Project (Environment Monitoring of Off-Shore Oil Prospecting Activities and Exploration). The project goal is to evaluate the impact in the marine ecosystem of cuttings discharged during oil well drilling activities. In order to estimate a drilling behavior, the project uses a CFD simulator that samples the dispersion of discharged fluids and materials, producing uniform 2D or 3D grid planar sections at any time instance of the simulation.

Keywords: volume rendering, data compression, graphics hardware, 3D textures, scientific visualization.

1 Introdução

O estudo de fenômenos físicos é de grande importância em várias áreas da ciência. Modelos computacionais matemáticos são desenvolvidos com base em equações que tentam aproximar da realidade, na melhor forma possível, cada fenômeno. A representação de escoamento de fluidos, em particular, é feita por vários grupos de pesquisa da área de modelagem de fluidos computacionais (CFD - *Computational Fluid Dynamics*). O comportamento dos fluidos é usualmente governado por um conjunto de equações diferenciais denominadas equações de Navier-Stokes, mas a representação geométrica destes dados é complexa, não possuem uma forma definida e, sobretudo, alteram-se com o tempo.

Técnicas de amostragem de dados são usadas freqüentemente em casos onde a função de modelagem é complexa. Um espaço de forma geométrica simples, que contenha o fenômeno, é escolhido (um cubo, por exemplo) e discretizado ao longo de uma grade contendo células de mesmo formato. Cada um desses volumes de amostragem, como são usualmente chamados, corresponde a um instante da representação do fenômeno. Para representar dados temporais, são enumerados tantos volumes quantas forem as diferentes instâncias de tempo. Assim, o objetivo deste trabalho é desenvolver uma forma de representar esses dados eficientemente, para um processo de visualização interativa.

Um exemplo da modelagem de escoamento de fluidos encontra-se no Projeto MAPEM (Monitoramento Ambiental em Atividades de Perfuração Exploratória Marítima)[1]. Neste projeto, que será o estudo de caso deste trabalho, analisa-se o comportamento e o impacto do descarregamento de resíduos da perfuração de poços de petróleo em um ecossistema marinho. Em uma de suas fases, as perfurações envolvem o uso de um fluido não-aquoso (prejudicial ao meio ambiente) e os cascalhos descarregados, apesar de lavados previamente, estão impregnados com certa quantidade desse fluido. Tendo em vista a importância da preservação do meio ambiente, as empresas de extração de petróleo em alto mar devem se preocupar com o monitoramento das substâncias e materiais descarregados durante os processos de perfuração exploratória para extração de petróleo. As altas multas decorrentes de contaminação aumentam consideravelmente a importância desse monitoramento.

Nesse projeto, um simulador denominado OOC (*Offshore Operators Commitee Mud and Produced Water Discharge Model*) é usado para prever a dispersão e a deposição no fundo do mar dos materiais descarregados durante as perfurações de petróleo (Brandsma e Smith, 1999). Os dados gerados por este simulador são escalares e representam concentrações de material por unidade de área em grades 2D (seções planares), 3D (espaciais) ou 4D (espaço-temporais). Para um melhor entendimento destes resultados, uma apresentação visual de tais dados é feita (Comba *et al.*, 2002) usando técnicas de Visualização Científica (Brodlie *et al.*, 1992) ou, mais especificamente, Visualização Volumétrica (Kaufman, 1991 e Brodlie e Wood, 2001).

A representação eficiente de tais dados torna-se crucial quando aplicada em conjunto com uma técnica de visualização volumétrica. Obviamente, a visualização de grades 4D é mais complexa que grades 2D ou 3D, sendo alvo de pesquisas recentes (Shen *et al.*, 1999, Ellsworth *et al.*, 2000 e Kraus e Ertl, 2002). O foco do presente

[1] MAPEM é um projeto financiado pelo programa FINEP/CTPetro e será abordado em maior detalhe no capítulo 5.

trabalho consiste em apresentar uma nova abordagem para a representação comprimida destes dados, visualizando-os em tempo real usando *hardware* gráfico.

O desenvolvimento de *hardware* gráfico nos últimos anos tem resultado em melhorias de performance e em novas ou mais genéricas características em relação às gerações de *hardware* anteriores. Recentemente, a ênfase no projeto de *hardware* gráfico tem focado o desenvolvimento de novas funcionalidades, dado que os níveis de performance têm atingido patamares extremamente satisfatórios. Como resultado, efeitos especiais mais complexos podem ser obtidos, aumentado o realismo das imagens geradas.

A geração da cor final associada a cada *pixel* de tela é a tarefa crucial dos algoritmos de visualização. Isto não é simples de ser realizado, pois a cor final representa a contribuição de diferentes componentes como, por exemplo, as componentes difusa, especular e ambiente no modelo de iluminação de Phong (Foley e Dam, 1992). Dentre as funcionalidades adicionadas às placas (*hardware*) gráficas, o mapeamento de texturas permitiu a especificação de novos componentes para a geração da cor final do *pixel*. A idéia é usar a memória da placa gráfica para armazenar os valores da função correspondente aos dados, chamada de textura, e usar uma outra função que permite mapear um dado *pixel* a esta representação. Inicialmente, uma única textura, usualmente bi-dimensional, gerada proceduralmente ou obtida diretamente a partir de uma imagem 2D estava disponibilizada por *pixel*. Mais recentemente, têm sido usadas texturas tri-dimensionais, bem como a existência de múltiplas texturas por *pixel*.

Esta nova abordagem também se mostrou útil na visualização de determinados volumes. O método descrito por Engel *et al.* (2001) consiste em representar os dados volumétricos em uma textura 3D e visualizá-la por uma seqüência de planos paralelos entre si e ortogonais a direção de visualização. Desta forma, cada *pixel* gerado a partir de cada um destes planos será mapeado para a textura 3D, obtendo o valor do dado volumétrico amostrado. O uso de transparência e o desenho dos planos ordenados pela distância do observador permitem a visualização volumétrica em tempo real de um dado volume (Engel *et al.*, 2001).

A extensão desta técnica para mais de um volume pode ser feita através da criação de uma textura 3D para cada instância de tempo. Entretanto, a capacidade de memória do *hardware* gráfico e de memória do computador são limitações para esta solução. Como volumes pequenos usam uma quantidade considerável de memória (128^3 com 4 bytes por entrada resulta em 8MB), o número de instâncias a ser usado nesta abordagem é reduzido.

Uma outra característica recente da técnica de mapeamento de texturas é usar o resultado de um mapeamento de textura para definir o acesso a uma textura subseqüente. Esta técnica, denominada de "texturas dependentes", tem mostrado um potencial enorme para estender ainda mais os efeitos especiais a serem aplicados.

Este trabalho apresenta uma nova forma de compactar as informações volumétricas temporais a qual supera as limitações de memória existentes, com melhor desempenho quando os dados são esparsos e possuem alto grau de coerência (tanto espacial quanto temporal). Para descomprimir tal representação, utiliza-se a funcionalidade de texturas dependentes para recuperar a informação original. A Figura 1.1 ilustra, em uma visão geral, os processos envolvidos na técnica, desde os dados a serem comprimidos até sua visualização.

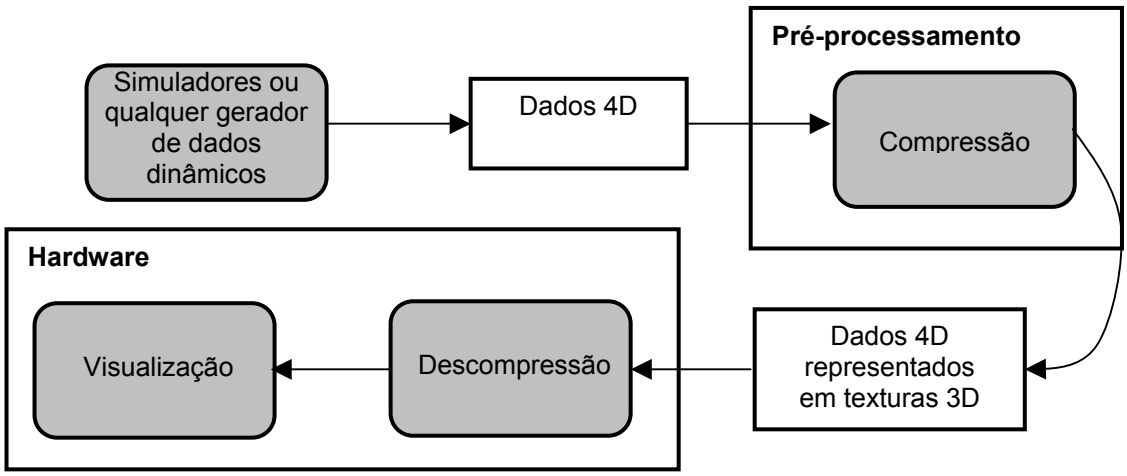


FIGURA 1.1 - Processos envolvidos na compressão e visualização de dados 4D

O método de compressão proposto torna possível a visualização em tempo real de simulações que apresentam comportamento dinâmico e cujos resultados são representados por vários volumes de dados, correspondendo a instâncias de tempo. Ao fim deste trabalho, o método de compressão / visualização será aplicado e validado com os dados provenientes do Projeto MAPEM. Entretanto, a aplicabilidade do método e sua visualização não são específicas para os dados desse projeto, mas para qualquer aplicação cujos dados sejam esparsos e apresentem coerência espaço-temporal.

Formando a base para o entendimento do problema, o capítulo 2 apresentará os conceitos e técnicas de visualização volumétrica e compressão de dados volumétricos ou de grandes dimensões.

O capítulo 3 revisa a arquitetura do *hardware* gráfico atual. Neste capítulo, são apresentadas as características do *hardware* e suas propriedades. Será dada também uma ênfase maior às peculiaridades da estrutura que engloba as operações com unidades de textura, as quais serão base para o método de compressão / visualização proposto.

Formada a base das técnicas de visualização volumétrica, compressão e das propriedades do *hardware*, o capítulo 4 detalha o mecanismo de compressão de dados volumétricos dinâmicos proposto. Será explorada não só sua idéia estrutural, mas também sua viabilidade, ou seja, a implementação.

O capítulo 5 apresenta os resultados obtidos, através de um estudo de caso com dados oriundos do Projeto MAPEM. São analisados resultados obtidos com a técnica de visualização que se tem hoje em dia e com a aplicação deste método de compressão/descompressão. São elaboradas, também, análises de eficiência do método, o qual atingiu uma visualização dinâmica (animação) e interativa de mais de 30 *frames* por segundo, caracterizando tempo real.

Finalmente, o capítulo 6 conclui o trabalho proposto. Trabalhos futuros também serão abordados neste capítulo como sugestão para continuação e ampliação deste método de representação e visualização.

2 Visualização Volumétrica e Compressão

Este capítulo aborda os conceitos fundamentais sobre visualização volumétrica e compressão de dados volumétricos. Para os algoritmos de visualização, são descritos os trabalhos básicos na área, sendo um artigo recente (Brodlie e Wood, 2001), que faz uma análise crítica das diferentes técnicas e suas extensões, a principal fonte bibliográfica desta parte. Na seção de compressão de dados volumétricos, são apresentadas técnicas recentes de compressão para dados estáticos e dinâmicos, utilizando algumas das técnicas de visualização apresentadas. Esta revisão é importante, pois o último tipo de compressão será o alvo deste trabalho.

2.1 Introdução

A visualização de dados visa fornecer aos usuários ferramentas para a identificação de características significativas nos dados ali representados, usando diversas técnicas de análise, exibição e exploração. Os dados são de naturezas diversas. Por exemplo, uma forma de obter diretamente dados referentes a fenômenos físicos é através de instrumentos de aquisição, como sensores, *scanners* ou até mesmo satélites. Por outro lado, representações podem ser criadas através da modelagem matemática de um fenômeno físico, estando tal abordagem presente em áreas como medicina, geologia, meteorologia, bioquímica, etc. Em todas estas áreas e aplicações, a visualização de dados tem papel importante.

Uma classe específica de dados interessante a este trabalho refere-se a enumerações regulares do espaço. No caso tri-dimensional, tais dados são ditos volumétricos, definidos em grades que contém valores escalares ou vetoriais associados. Estes dados volumétricos são representados por uma matriz de elementos de volume, onde cada elemento básico é chamado de *voxel (volume element)* (Figura 2.1). A existência de um ou mais valores associados a uma posição de amostragem depende das propriedades dos dados que o volume representa. Este tipo de representação é usado vastamente para dados provenientes de fenômenos físicos ou para modelagens geométricas complexas de problemas. Por exemplo, a análise por elementos finitos ou por dinâmica de fluidos é utilizada vastamente para simular eventos da natureza, constituindo uma fonte geradora de dados volumétricos. No caso de dinâmica de fluidos é comum haver mais de um valor escalar associado à mesma posição, tais como temperatura, pressão e densidade, bem como dados vetoriais.

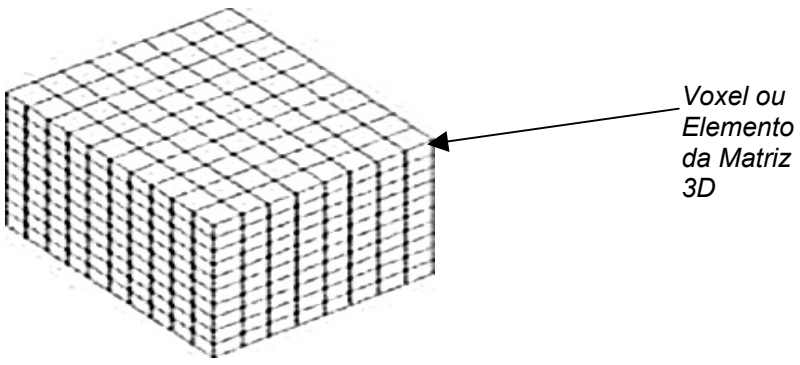


FIGURA 2.1 - Grade ou matriz tridimensional representando um volume

Uma forma de analisar estes dados é usar a técnica de Computação Gráfica chamada Visualização Volumétrica, de modo a comparar os dados oriundos de simulações com resultados numéricos derivados de experimentos ou coletados da natureza.

Um dos principais problemas desta área está relacionado ao tamanho dos conjuntos de dados volumétricos, em geral da ordem de vários *megabytes*, os quais necessita, de compressão. Estes volumes de dados são ditos multidimensionais, onde cada ponto do conjunto de dados é representado por uma *n*-upla, onde cada elemento desta *n*-upla representa uma coordenada em um espaço *n*-dimensional, com *n* maior ou igual a três. Entretanto, o presente trabalho destina-se apenas a dados que variam ao longo do tempo e, conseqüentemente, uma destas dimensões será o tempo.

A Figura 2.2 mostra um exemplo de visualização volumétrica com dados do projeto MAPEM, correspondendo ao resultado da simulação do descarregamento de cascalhos em alto mar ao longo de três instantes de tempo. Este exemplo será explorado posteriormente, no capítulo 5.

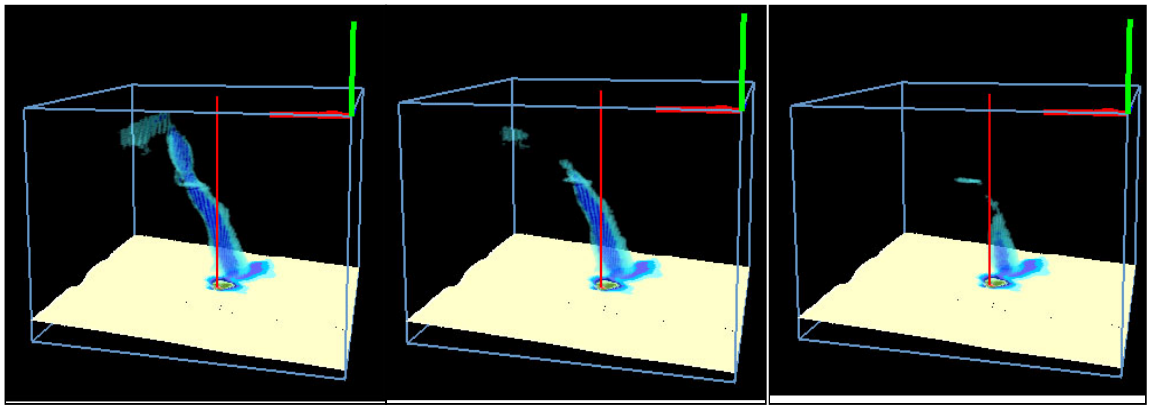


FIGURA 2.2 - Visualização de três instantes de tempo de dados volumétricos

2.2 Técnicas de Visualização Volumétrica

Na literatura, vários termos são utilizados para caracterizar as diferentes classes de técnicas de visualização de volumes. Neste trabalho são utilizados os termos definidos por Kaufman (1991), que considera duas abordagens básicas para a solução do problema de visualização volumétrica: extração de superfícies (*surface rendering*) e visualização direta de volumes (*volume rendering*).

Estas classes diferem basicamente pela utilização ou não de representações intermediárias dos dados volumétricos para a geração da visualização adequada à aplicação. Enquanto na visualização direta de volumes a projeção para formar a imagem é realizada diretamente a partir dos dados volumétricos, na extração de superfícies os dados volumétricos são convertidos para uma representação geométrica (polígonos), a partir da qual são usados os métodos tradicionais de visualização de malhas de polígonos. A Figura 2.3 apresenta o esquema básico da possível conexão entre estas duas classes de técnicas.

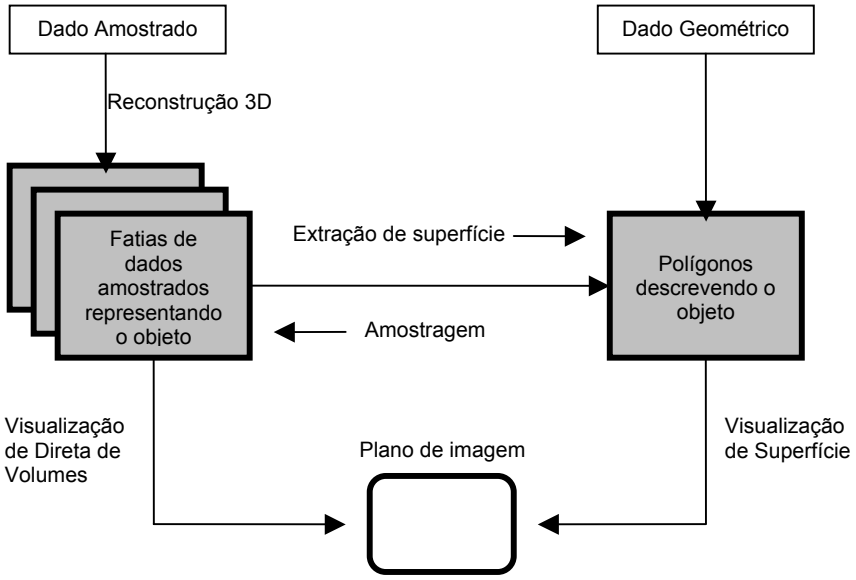


FIGURA 2.3 - Representação esquemática das técnicas de visualização volumétrica (Kaufman, 1991)

As técnicas envolvidas no processo de visualização volumétrica (Figura 2.3) podem ser resumidas, segundo Kaufman (1991) na execução de cinco passos básicos, definidos na Figura 2.4. No entanto, a visualização é realizada através da implementação apenas dos três últimos passos, pois é considerada somente a visualização de volumes já pré-processados.

No final do processo de aquisição do volume, os dados são reconstruídos, gerando um volume com dimensões proporcionais. Uma vez fornecido um volume já determinado, pode-se dar início ao processo de visualização. Para isto, inicia-se a etapa de classificação, que está relacionada à identificação do material representado em cada *voxel*. Esta etapa possibilita a seleção de características dos dados, segundo um critério quantitativo, definindo a região do volume que se deseja explorar. Em técnicas de extração de superfícies, classificar significa definir o valor de limiarização (*threshold*) utilizado para identificar a superfície que deve ser poligonalizada, para posterior visualização. Já para técnicas de visualização direta de volumes, a etapa de classificação envolve a definição da relação entre os valores dos dados do volume e os valores de cor e opacidade que serão utilizados no algoritmo de exibição, ou seja, a definição das funções de transferência.

Em seguida, para visualização direta de volumes, é aplicado um modelo de iluminação (etapa de mapeamento) que, com base nas propriedades do material e de cada *voxel* e nas condições de iluminação externas, calcula a tonalidade de cor em cada ponto do volume. Nas técnicas de extração de superfícies, esta etapa realiza a remoção de áreas ou faces ocultas, introduzindo um modelo de iluminação para o cálculo da cor final dos polígonos.

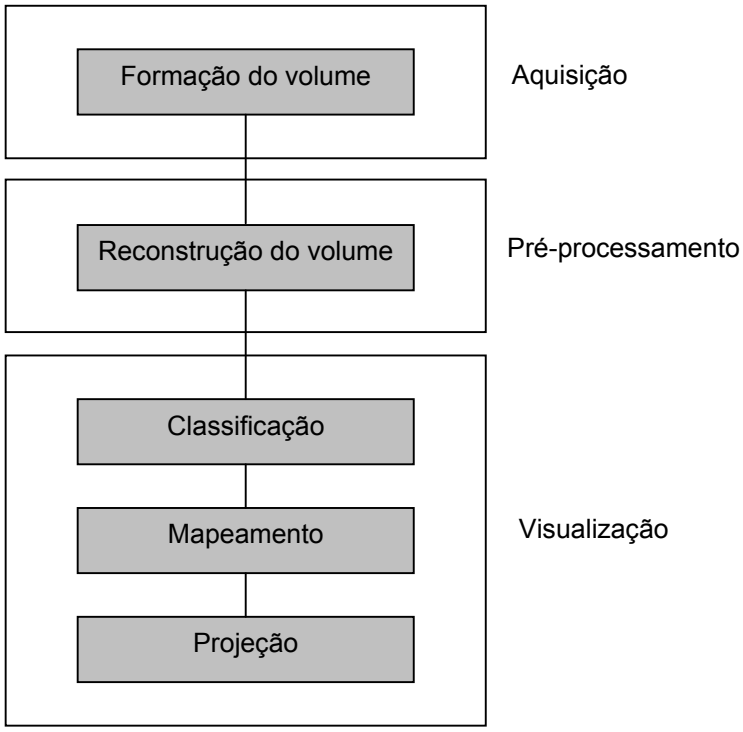


FIGURA 2.4 - Processo para visualização volumétrica

A última etapa do processo de visualização envolve a projeção dos *voxels* ou polígonos mapeados na superfície de visualização e a conseqüente composição para determinar a imagem a ser visualizada. Nos algoritmos de visualização direta de volumes é realizada a projeção dos *voxels* e o cálculo da composição dos mesmos sobre o plano da imagem.

A seguir, encontra-se a descrição das técnicas mais conhecidas destas duas abordagens. Entretanto, o escopo deste trabalho encontra-se em uma técnica específica da visualização direta de volumes, o Mapeamento de Textura 3D, à qual será dada uma ênfase maior.

2.2.1 Extração de Superfícies

Esses métodos produzem a visualização através da geração de representações geométricas dos dados de modo a isolar um determinado objeto que está representado nos dados volumétricos. Em razão disto, a quantidade de dados manipulados é reduzida quando da formação da imagem. Esta classe de algoritmos utiliza técnicas convencionais de Computação Gráfica para a visualização de polígonos e do re-processamento do volume para se extrair novamente o objeto toda vez que uma alteração da característica do volume a ser visualizada for solicitada.

Resumidamente, estes algoritmos tipicamente ajustam uma superfície construída por polígonos a pontos de isovalor dentro dos dados volumétricos. As grandes vantagens destas técnicas são a velocidade para a geração e exibição da imagem final e o pequeno espaço de armazenamento requerido. No entanto, não são apropriadas quando existem isosuperfícies complexas nos dados.

Entre os algoritmos de extração de superfícies, destacam-se dois que são descritos a seguir: conexão de contornos, proposto por Keppel (1975) e adaptado por Fuchs *et al.* (1977), e cubos marchantes (*marching cubes*), idealizado por Lorensen e Cline (1987).

*2.2.1.1 Conexão de Contornos (*Contour-Connecting*)*

A idéia básica deste algoritmo é determinar uma isolinha em cada fatia de dados e, posteriormente, conectar as fatias adjacentes. O método opera inicialmente em cada fatia de dados individualmente, seguindo uma determinada ordem dos objetos.

Após o valor de limiarização ser especificado, uma curva fechada conecta estes valores para cada fatia de dados (linha de isovalores). Uma vez determinado o contorno de cada fatia, o problema volta-se para encontrar a conexão ótima, geralmente através de triângulos, interligando as curvas de fatias adjacentes (Figura 2.5).

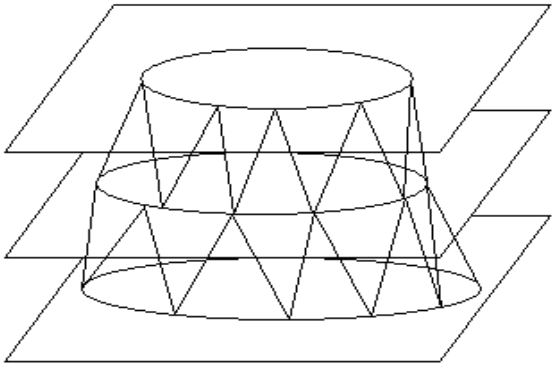


FIGURA 2.5 - Algoritmo de conexão de contornos (Keppel, 1975)

Quando uma fatia contém mais de um contorno que pode ser conectado a um dos contornos da fatia seguinte, é usado, em geral, um critério de proximidade para definir os pares de contornos a serem conectados, sendo este o problema central da técnica. A Figura 2.6 ilustra esta dificuldade.

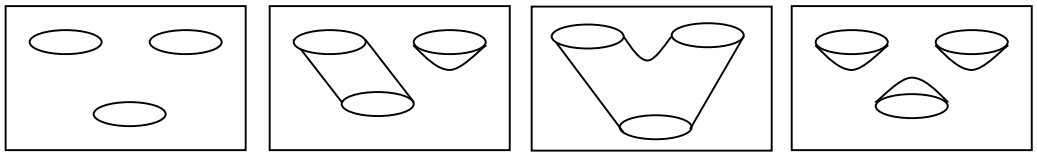


FIGURA 2.6 - Modelos que podem ser gerados para um mesmo par de fatias (Fuchs *et al.*, 1977)

Sem entrar em detalhes no problema apresentado na Figura 2.6, trata-se da conexão dos contornos definidos em duas fatias adjacentes, onde o objetivo é encontrar a conexão ótima. Considera-se um contorno como um ciclo fechado de segmentos de reta ligando pontos de mesma propriedade em um plano particular, sendo todos os planos paralelos entre si. Considera-se também dois contornos situados em planos adjacentes, definidos como, por exemplo, ($P_0, P_1, P_2 \ldots P_{m-1}, P_0$) e ($Q_0, Q_1, Q_2 \ldots Q_{n-1}, Q_0$).

Existem dois problemas básicos para a construção de uma superfície a partir de conjuntos estruturados de contornos planos: correspondência e acoplamento. O problema de correspondência ocorre quando um segmento de um contorno no plano é criado e existem dois ou mais contornos próximos no mesmo plano. Algumas soluções para este problema foram sugeridas por Meyers (1992). Uma solução emprega cilindros elípticos que produzem contornos ajustados por elipses; outra solução usa árvores geradoras de grafos que ligam os contornos em seções adjacentes, sendo que o grafo gerado é usado para encontrar a melhor conexão de contornos.

O problema de acoplamento está relacionado à maneira como os contornos serão unidos em duas fatias adjacentes para formar os triângulos que geram a superfície. Cada triângulo é formado por dois pontos do contorno P e um do contorno Q, ou vice versa. Isto pode ser escrito como: (P_i,P_j,Q_k) ou (Q_i,Q_j,P_k). O problema é decidir a orientação do próximo triângulo. Fuchs *et al.* (1977) determinam o triângulo seguinte utilizando as seguintes regras:

- cada segmento de contorno será usado por um único triângulo;
- o lado esquerdo de um determinado triângulo será usado como lado direito do seguinte.

A decisão do próximo triângulo a ser conectado é feita com base na teoria dos grafos, usando a seguinte hipótese: "toda superfície aceitável definida entre dois contornos pode ser associada com certos ciclos em um grafo toroidal direcionado" (Meyers, 1992). A superfície ótima corresponde ao caminho mínimo no grafo toroidal. O custo é a soma dos arcos percorridos, sendo que várias heurísticas são propostas para calcular este caminho mínimo.

*2.2.1.2 Cubos Marchantes (*Marching Cubes*)*

O algoritmo de cubos marchantes foi introduzido por Lorensen e Cline (1987) com uma versão similar de Wyvill *et al.* (1986) e consiste em uma técnica clássica, sendo uma das mais utilizadas para a extração de superfícies e visualização de dados amostrados.

O algoritmo assume que os dados estão armazenados em uma grade estruturada e cada célula é processada individualmente. As células na grade estruturada são topologicamente equivalentes a cubos e o método foca na extração de superfície ao longo de um único cubo. O processo completo de extração é realizado percorrendo todos os cubos, um após o outro, derivando o nome de cubos marchantes.

Cada vértice de um cubo pode ser maior ou menor que um determinado limiar k. Assim, 256 (2^8, pois um cubo possui 8 vértices) cenários diferentes são possíveis. Uma função $f(x,y,z)$ pode ser construída como uma interpolação tri-linear dos valores de cada vértice de um cubo, gerando uma isosuperfície. As interseções da isosuperfície $f(x,y,z) = k$ com as arestas do cubo são fácil e corretamente calculadas por interpolação linear inversa. A Figura 2.7 ilustra esta função $f(x,y,z) = k$, onde os vértices do cubo que estão abaixo do plano apresentam valores menores que o limiar k e os que estão acima apresentam valores maiores que k.

Como o comportamento de $f(x,y,z) = k$ dentro do cubo é o de uma superfície cúbica, uma simples estimativa de f ao longo do cubo pode ser realizada unindo os pontos de intersecção em um conjunto de triângulos.

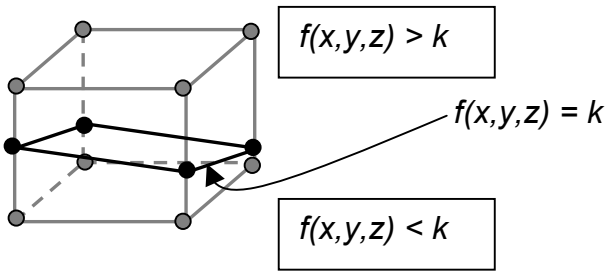


FIGURA 2.7 - Função de interpolação para cada vértice

Encerrado o processamento de um cubo, o algoritmo prossegue (“marcha”) para a próxima célula ou cubo. Depois que todas as células forem “visitadas”, a isosuperfície, definida em termos de uma malha poligonal, está pronta para ser visualizada.

Um ponto central do algoritmo é a definição de uma tabela que determina a topologia da superfície dentro do cubo para cada caso possível. No trabalho de Lorensen e Cline (1987), os 256 casos dessa tabela são reduzidos a 15 casos básicos, descritos na Figura 2.8, através de considerações de simetria e complementaridade, onde a topologia da superfície é invariante a rotações do cubo e a inversão dos valores dos vértices é equivalente ao caso original.

É importante ressaltar que recentemente, Cignoni *et al*. (2000) mostraram que os casos de ambigüidade se estendem de 256 casos para 798 casos, mas apenas 88 são de configurações distintas. Outro trabalho anterior (Chernayev, 1995) também identificou outros 33 casos básicos.

Um problema do *marching cubes* é que existem casos ambíguos na tabela de casos. Uma análise cuidadosa dos casos 7, 10, 12 e 13, na Figura 2.8, como casos subseqüentes, mostra que existem estados nos quais uma célula pode ser “cortada” de mais de uma maneira. Essa ambigüidade ocorre na face de um *voxel* quando vértices em arestas adjacentes estão em estados diferentes, mas vértices diagonais estão no mesmo estado (todos dentro ou todos fora da superfície). A escolha de um dos dois casos precisa ser consistente ao longo do processamento. Por exemplo, vejamos o caso 3 e o complementar do caso 6 (casos complementares são obtidos substituindo-se os vértices que estão dentro pelos que estão fora ou vice-versa). Se dois casos ambíguos são implementados de forma independente um do outro, ocorre a formação de um buraco na superfície.

No *marching cubes*, a solução adotada consiste em estender os 15 casos originais para incluir os casos complementares, sendo que estes casos são projetados para serem compatíveis com os casos vizinhos e evitam a criação de buracos.

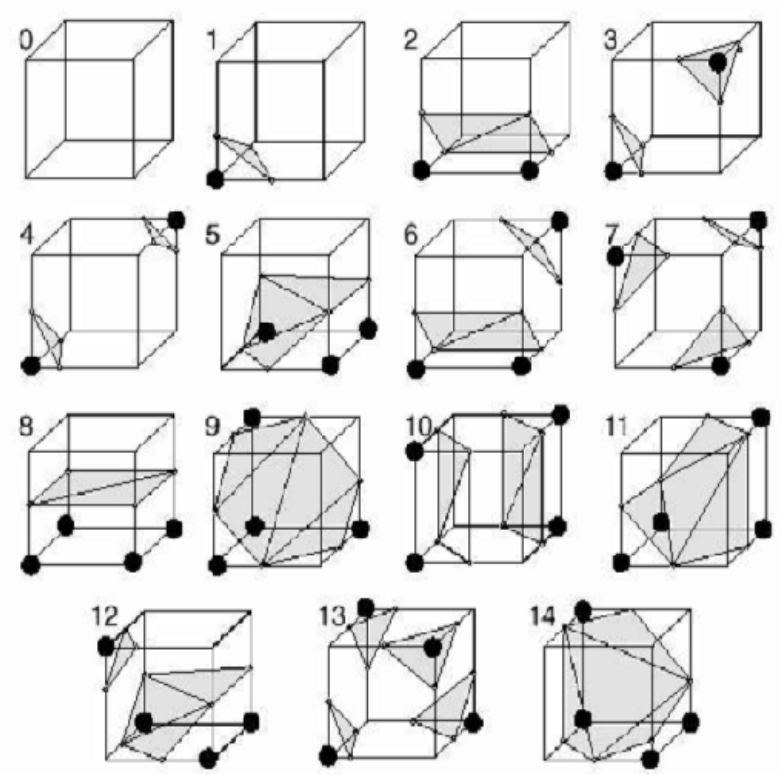


FIGURA 2.8 - Os 15 casos básicos do algoritmo de *marching cubes* (Lorensen e Cline, 1987)

Posteriormente, Cline *et al*. (1998) descobriram também que o tamanho de alguns triângulos gerados era menor que o tamanho de um *pixel*. O algoritmo denominado divisão de cubos (*dividing cubes)* foi, então, desenvolvido para tirar vantagem desta observação. O algoritmo efetua a projeção das células na tela, verificando somente aquelas em que a projeção é maior que um *pixel*. Se isto ocorrer, a célula é dividida em sub-células, cada uma das quais determinando um ponto. Se não, a célula toda é visualizada como um ponto.

2.2.2 Visualização Direta de Volumes

A visualização direta de volumes oferece uma solução mais completa para a visualização de volumes, a qual tem o objetivo de retratar o volume inteiro ao invés de uma pequena amostra do mesmo. Este conjunto de técnicas tem sido, tradicionalmente, considerado mais custoso computacionalmente do que a extração de superfícies. Entretanto, essa situação tem sido recentemente minimizada por novos desenvolvimentos em *hardware*.

Este método consiste em tomar o volume constituído por *voxels* e projetá-los diretamente em *pixels*, dispensando o uso de primitivas geométricas ou representação intermediária. Neste caso, numa etapa de classificação, é utilizada uma função de transferência que corresponde ao mapeamento dos valores dos *voxels* para propriedades visuais, tais como cor e opacidade. Estas funções de transferência podem, por exemplo, utilizar-se de estimativas de gradiente, onde se pode combinar a diminuição da cor e

opacidade nas áreas de baixo gradiente com o aumento destas características nas áreas de alto gradiente.

A visualização das estruturas de interesse dentro do volume é realizada a partir da "visita", em uma certa ordem, a todos os *voxels* (ou quase todos, dependendo do algoritmo) e da aplicação da função de transferência para a construção da imagem. Esta ordem pode ser classificada, segundo Kaufman (1991), em ordem dos objetos (*object-order* ou *forward projection*), quando os *voxels* são projetados diretamente no plano de imagem, e ordem da imagem (*image-order* ou *backward projection*), que determina para cada *pixel* do plano da imagem quais são as amostras que contribuem no cálculo da sua intensidade.

Estas técnicas possuem um alto custo computacional, pois normalmente envolvem uma interpolação de valores nos pontos ao longo da direção de visualização. Por outro lado, produzem imagens de excelente qualidade, uma vez que todos os *voxels* podem ser usados na síntese das imagens, possibilitando a visualização interior dos objetos.

Os principais algoritmos que se destacam neste grupo são: *ray casting*, *splatting* e *shear-warp*. Essas técnicas são descritas a seguir juntamente com a técnica de mapeamento de textura, considerada também muito importante na visualização direta de volumes e usada no método proposto neste trabalho.

2.2.2.1 Algoritmo de Ray Casting

Resumidamente, o algoritmo de *ray casting* (Levoy, 1988 e Levoy, 1990) baseia-se em raios lançados do ponto de vista do observador, passando através de cada *pixel* da tela e interceptando o volume. Se o raio interceptá-lo, o conteúdo do volume ao longo do raio é amostrado, transformando-se o valor acumulado em cor e opacidade, atribuídos ao *pixel*. Este algoritmo é o mais usado para a visualização de volumes quando se necessita de imagens de alta qualidade (Elvins, 1992).

Os parâmetros de visualização tradicionalmente especificam a posição do observador, o tipo, a direção e o plano de projeção e os planos de corte 3D. No contexto da visualização direta de volumes, a direção de projeção $\overrightarrow{DOP}$ determina a orientação dos raios lançados no volume. O plano de projeção é mapeado para uma janela de visualização I no plano da imagem, que se refere aos valores de intensidade associados às posições dos *pixels* (Figura 2.9).

A intensidade para cada *pixel* da imagem é determinada pelo lançamento de um raio R_p deste *pixel* ao seguindo a direção de visualização. Ao longo de cada raio são processadas s amostras, em geral com um espaçamento constante. O processamento de uma amostra corresponde ao cálculo de um valor de cor e um valor de opacidade dependentes das posições de amostragem. Caso a posição da amostra não coincida com as posições discretas de amostragem do volume, o cálculo é feito a partir da cor e opacidade dos *voxels* vizinhos.

Na Figura 2.9, considerou-se que o processamento do algoritmo de *ray casting* é realizado na ordem de frente para trás (*front-to-back*), ou seja, as amostras são processadas a partir do primeiro ponto de intersecção a do raio com o volume até o segundo ponto de intersecção b. Entretanto, também é possível realizar o processamento na ordem de trás para frente (*back-to-front*), ou seja, de b para a.

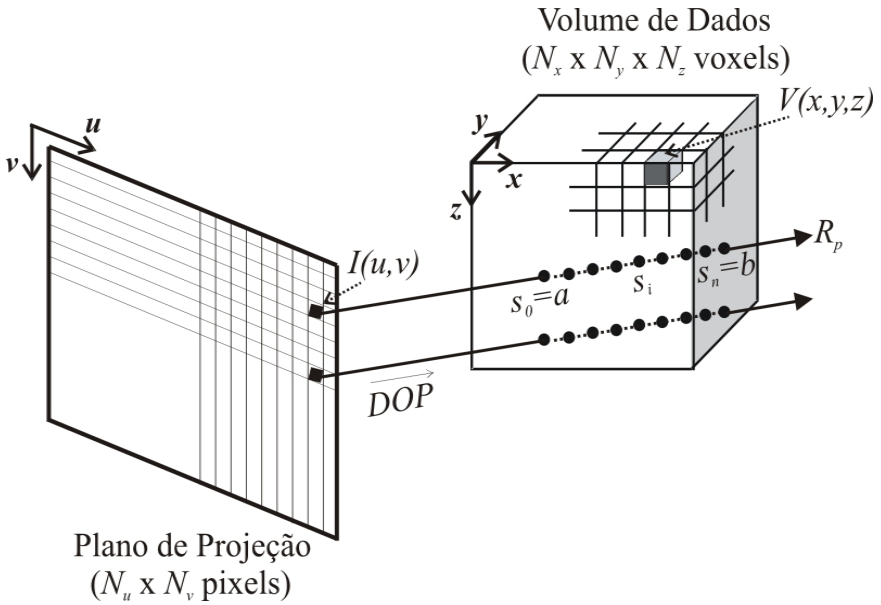


FIGURA 2.9 - Esquema genérico do algoritmo de *ray casting front-to-back* (extraída de Manssour e Freitas, 2002)

2.2.2.2 *Algoritmo de* Splatting

Splatting (Westover, 1989 e Westover, 1990) é um algoritmo de visualização do tipo *image-order* usado com volumes de dados regulares, mas que podem não possuir o mesmo espaçamento nas três dimensões da grade. É inspirado na estrutura do *pipeline* de visualização de polígonos, onde cada primitiva passa ao longo de vários estágios por vez.

Cada elemento é mapeado no plano da tela e, em seguida, através de um processo de acumulação, tem sua contribuição à formação da imagem calculada. A contribuição de um *voxel* é maior perto do centro de sua projeção e menor quando mais distante do centro. Pode-se fazer uma analogia como atirar (*splat*) uma bola de neve contra um muro. A contribuição da neve no centro do impacto será maior e decai à medida que se distancia do centro. O algoritmo termina quando todas as primitivas tiverem sido mapeadas na tela.

O algoritmo é definido por quatro etapas principais: transformação, tonalidade, reconstrução e visibilidade. Inicialmente, a amostra é processada através de sua transformação de coordenadas do volume para o espaço da tela. Em seguida é feita a tonalização da amostra, que engloba classificação e iluminação, usando apenas a informação local.

A reconstrução é realizada para todas as amostras em um plano do volume de dados, definido como um plano paralelo ao plano da imagem. Após a determinação da porção da imagem que a amostra influencia, através da função *footprint*, sua contribuição é adicionada em um acumulador do plano. Quando todas as amostras de um plano do volume de dados são processadas, o conteúdo do acumulador do plano é combinado na imagem através de um operador de composição. Depois que todas as amostras forem processadas, é, então, obtida a imagem final.

A função *footprint* é calculada para cada posição de observação do volume de dados. Como cada *voxel* projetado (*splat*) é representado por um núcleo (*kernel*) de interpolação simétrico, primeiro é calculada a extensão do espaço de tela onde o *kernel* será projetado. Essa projeção é chamada de um *footprint*. Para uma projeção ortográfica, o núcleo mais comum é um Gaussiano esférico, ilustrado na Figura 2.10.

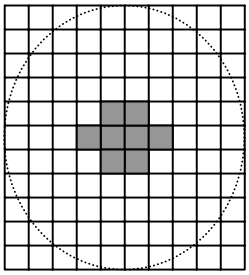


FIGURA 2.10 - Footprint, onde áreas escuras indicam maior intensidade

Todos os *pixels* cujos centros se encontram na região onde o *kernel* é projetado podem ser afetados pela amostra. A partir do *kernel*, são ponderados os valores dos *voxels*. A tabela que representa a função *footprint* é determinada em uma etapa de pré-processamento, acelerando o método (Westover, 1990).

Um algoritmo semelhante, denominado *v-buffer*, foi apresentado por Upson e Keeler (1998). Este algoritmo apresenta como diferença o fato de se basear em células 2D e não em células 3D (*voxels*). O algoritmo *v-buffer* percorre o interior da célula interpolando os valores dos vértices e projetando cada valor interpolado no plano de visualização. Tanto o método de *splatting* como o *v-buffer*, são facilmente paralelizáveis.

2.2.2.3 Algoritmo de Shear-Warp

A idéia do algoritmo de *shear-warp*, desenvolvido por Lacroute e Levoy (1994) com base, entre outros, no trabalho de Cameron e Undrill (1992), é pré-transformar os dados em uma orientação de maneira que a visualização seja rápida.

A base desta técnica está na fatoração da matriz de visualização, onde há a decomposição da transformação de projeção em duas etapas. Esta decomposição gera uma transformação de cisalhamento (*shear*) e uma de correção (*warping*).

Inicialmente, o volume é transformado para um sistema de coordenadas intermediário de modo que cada fatia fique paralela ao plano da imagem e perpendicular aos raios. Assim, com este cisalhamento, os raios de visão percorrem de maneira trivial as fatias do volume, gerando uma imagem intermediária distorcida. A outra parcela resultante da decomposição é uma transformação de correção da deformação da imagem intermediária, que gera a imagem final do objeto tridimensional. Estas fases podem ser visualizadas na Figura 2.11.

Resumidamente, então, o algoritmo é composto das seguintes etapas: (i) transformação do volume de dados para o espaço de cisalhamento, transladando e reamostrando cada fatia de acordo com a translação; (ii) composição das fatias reamostradas, projetando o volume em uma imagem 2D intermediária; (iii)

transformação da imagem intermediária para o espaço da imagem final, corrigindo a deformação.

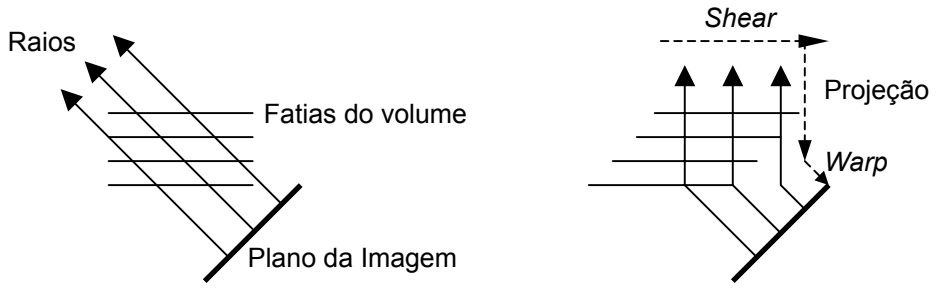


FIGURA 2.11 - Esquema do algoritmo *shear-warp* (Lacroute e Levoy, 1994)

O algoritmo explora a coerência do volume de dados usando o método de compactação *Run-Lenght Encoding* para os *voxels*, de tal forma que não ocorre o processamento dos *voxels* transparentes.

O método também pode ser paralelizável e sua performance é apresentada em Lacroute (1996).

2.2.2.4 Mapeamento de Textura

O desenvolvimento de *hardware* específico para computação gráfica motivou o trabalho de Akeley (1993) em apresentar a visualização de uma textura sólida usando o primeiro *hardware* da série *SGI Reality Engine*. Esse trabalho foi o precursor dentre as abordagens baseadas em texturas usando *hardware* para PCs. Posteriormente, Cabral *et al*. (1994) apresentou uma representação do volume de dados utilizando mapeamento de texturas com opacidade, tanto 2D como 3D. Esta abordagem foi significativamente expandida por Westermann e Ertl (1998), que introduziram um algoritmo para visualização de isosuperfícies sombreadas. Em seguida, Meiβner *et al.* (1999) desenvolveu um método de iluminação difusa para visualização direta de volumes semi-transparentes.

A seguir, são apresentadas duas abordagens baseadas em mapeamento de texturas para visualização direta de volumes. A primeira explora uma abordagem simplificada (bidimensional) e a outra aplica realmente o conceito em três dimensões.

2.2.2.4.1 Textura 2D

Especificamente, as texturas 2D podem ser usadas para a visualização volumétrica de dados regulares estruturados. Para isso, uma coleção de polígonos paralelos entre si é especificada ortogonal a cada eixo principal do sistema de coordenadas (*object-aligned*), como mostrado na Figura 2.12. Estes planos poderão ser desenhados tanto na ordem de frente para trás (*front-to-back*) como de trás para frente (*back-to-front*) e a especificação destas três coleções de planos se deve completamente à direção de visualização.

Dada uma direção de visualização que segue a mesma direção de qualquer um dos eixos, seria suficiente especificar apenas a coleção de planos que estivesse ortogonal a esta direção. Entretanto, a especificação de apenas um destes conjuntos de planos não é interessante quando se utiliza diferentes câmeras para visualização. Por exemplo, quando uma coleção é desenhada no plano XY e a direção de visualização se move para

o plano YZ, o usuário estará visualizando diretamente ao longo das fatias e, conseqüentemente, a imagem desaparecerá. Outro problema que poderia ocorrer com a visualização de uma única seqüência de planos é que o usuário também perceberia que o volume é realmente constituído de fatias, se os intervalos entre as mesmas forem consideráveis, já que a interpolação é bi-linear.

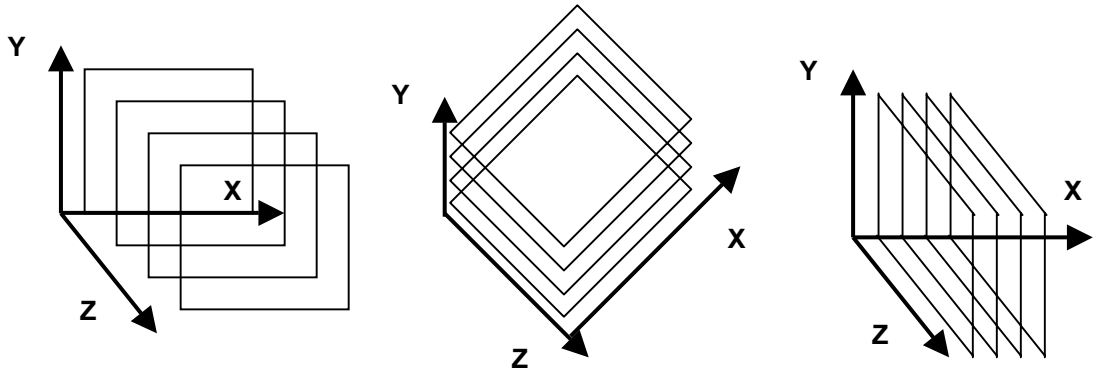


FIGURA 2.12 - Fatias ortogonais a cada eixo

Usando os três conjuntos de fatias, este problema seria evitado, pois, seguindo o exemplo, o plano YZ estaria na direção de visualização. Entretanto, a visualização destas três pilhas de fatias ao mesmo tempo seria muito custosa e causaria uma visualização errônea do objeto, já que, na realidade, três objetos seriam visualizados ao mesmo tempo. Para prevenir este possível problema, caso os objetos não sejam simétricos, apenas o conjunto de polígonos mais alinhado à direção de visualização é desenhado, tornando-se invisíveis os outros dois conjuntos. À medida que o ponto de visualização se movimenta, o sistema seleciona o conjunto de fatias apropriado, ou seja, o mais perpendicular ao observador é desenhado.

Solucionados tais problemas, o mapeamento de textura e da função de transferência de cor e opacidade é simples e pode ser abstraído no esquema da Figura 2.13.

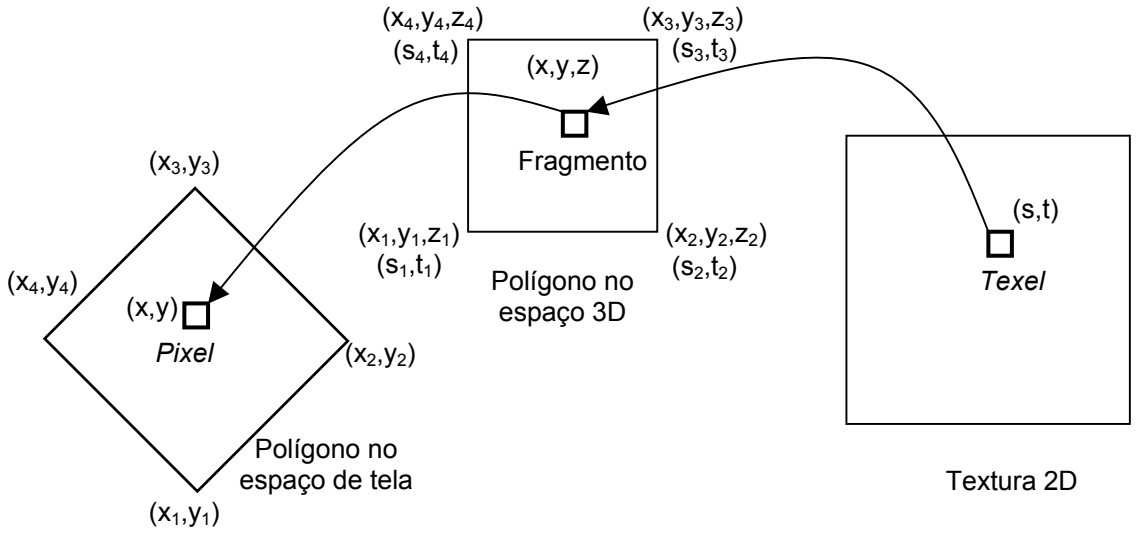


FIGURA 2.13 - Mapeamento de textura 2D

Apesar destas soluções propostas, o usuário ainda perceberá claramente as fatias quando estiver se aproximando de um ângulo de 90 graus em relação a qualquer um dos planos, necessitando de uma correção baseada no algoritmo de *shear-warp*.

2.2.2.4.2 Textura 3D

Este método surgiu basicamente com o avanço recente do *hardware* para a aceleração do processo de visualização, tornando-se rápido o bastante e sendo uma alternativa ao mapeamento de textura 2D para dados volumétricos. Com o avanço da tecnologia, o *hardware* gráfico acelerou todas as etapas do *pipeline* 3D.

A idéia desta técnica parte do princípio de uma pré-conversão do volume de dados para um mapa 3D de texturas, usando um método de classificação baseado em tabelas que representam a função de transferência de cor e opacidade dos *voxels*, onde cada fatia do volume de dados será representada em uma fatia do mapa de textura. A conversão dos mapas de textura é realizada apenas uma vez, sendo uma etapa de pré-processamento.

Uma vez criada a textura 3D, é necessário aplicá-la a uma coleção de polígonos (Yagel *et al.*, 1996) localizados em planos paralelos ao plano da imagem, como exemplificado na Figura 2.14. A interpolação agora será trilinear, fazendo com que o volume seja considerado na composição da cor de cada *pixel*. Na realidade, o volume é apresentado usando um artifício que é criar polígonos que são exibidos e recebem fatias da textura.

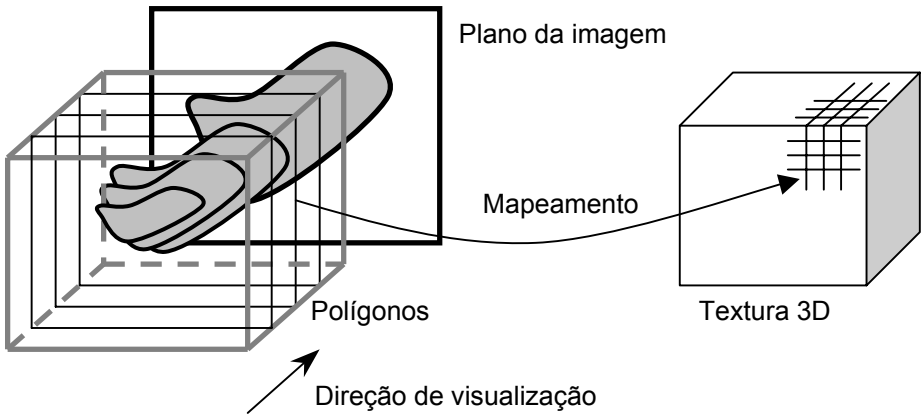


FIGURA 2.14 - Aplicação da textura 3D em uma seqüência de polígonos

Os planos podem ser desenhados de trás para frente (*back-to-front*) ou de frente para trás (*front-to-back*) e, de acordo com Engel *et al.* (2001), se a câmera se movimentar, os planos devem acompanhá-la de forma que fiquem sempre ortogonais à direção de visualização (*view-aligned*), como na Figura 2.15. Esta abordagem soluciona o problema apresentado pelo mapeamento de texturas 2D em dados volumétricos.

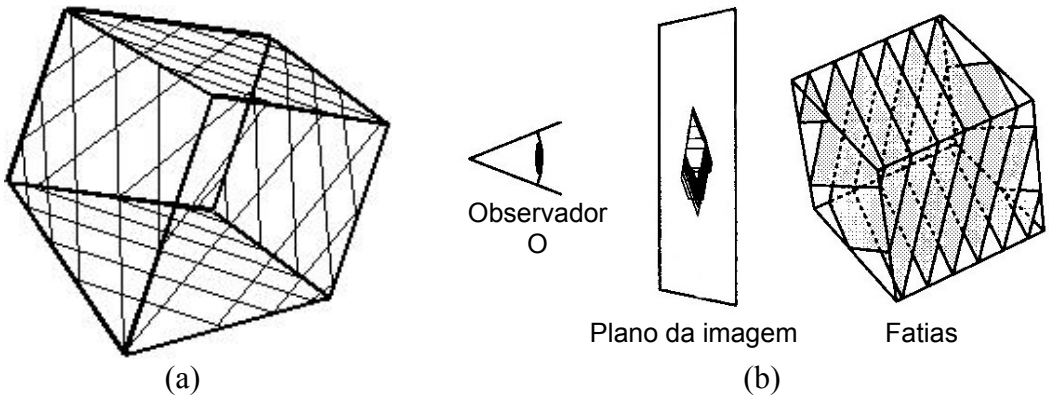


FIGURA 2.15 - Visualização usando textura 3D: (a) planos ortogonais ao olhar do leitor (paralelos à folha) e (b) planos ortogonais a direção de observação de O (Kraus e Ertl, 2002)

O número de planos utilizados é determinado levando em consideração o compromisso entre qualidade e velocidade. Quanto mais planos são utilizados, mais precisos os resultados e maior o tempo gasto pelo algoritmo. O número de planos exigidos para apresentar os dados adequadamente em uma dada direção é função da amostragem dos dados, podendo ser até uma relação de um para um.

Então, basicamente, o método de mapeamento de textura 3D resume-se a uma simples correspondência entre as coordenadas (x,y) de cada plano e uma unidade da textura 3D (*texel*) a ser interpolada (s,t,r). A Figura 2.16 apresenta de modo geral o mapeamento de um único polígono.

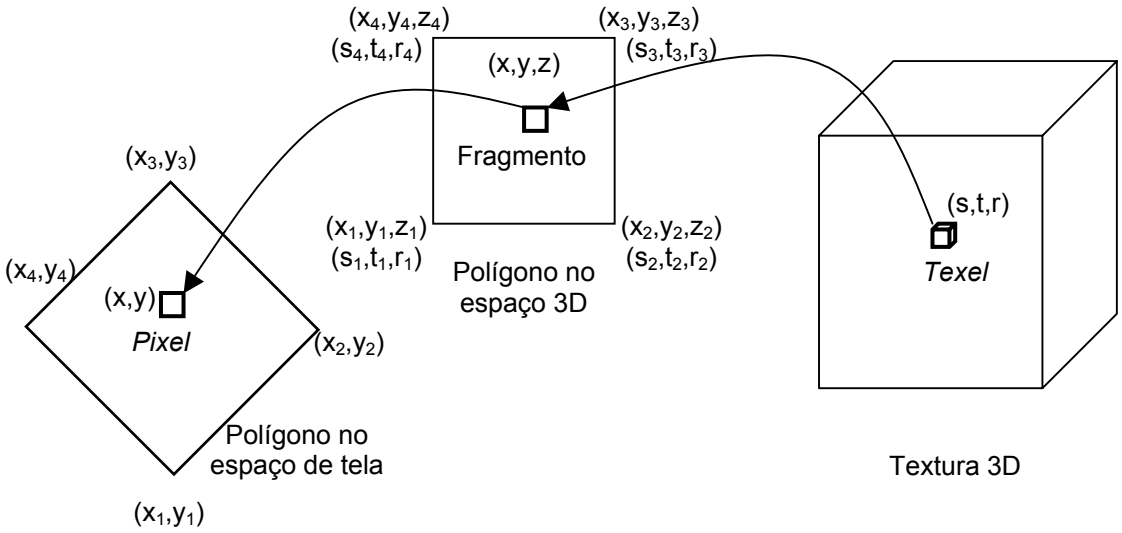


FIGURA 2.16 - Mapeamento de um único polígono a textura 3D

2.3 Compressão de Dados Volumétricos

Diversos trabalhos recentes em visualização volumétrica abordam técnicas de compressão de dados volumétricos de grandes dimensões para diminuir o volume a ser transmitido e, conseqüentemente, o tempo de visualização destes dados. Várias destas técnicas utilizam abordagens hierárquicas na visualização, enquanto outras focam em armazenar e transmitir eficientemente os dados volumétricos.

Por exemplo, Benson e Davis (2002) e DeBry *et al.* (2002) desenvolveram, independentemente, abordagens semelhantes para representação de texturas em *octrees*, onde apenas a parte do volume (nodos) que intercepta uma superfície do modelo geométrico é armazenada.

Todas estas técnicas de compressão de dados volumétricos estáticos são baseadas em uma propriedade denominada "coerência espacial", ou seja, no fato de que *voxels* adjacentes têm valores semelhantes de intensidade de cor ou outra propriedade qualquer. Para dados dinâmicos, a coerência temporal é levada em consideração. As principais e mais recentes técnicas de compressão de dados volumétricos são descritas a seguir.

2.3.1 Visualização Volumétrica com Multiresolução Baseada em *Octrees* como Representação de Texturas

Para grandes volumes de dados, isto é, volumes maiores que a capacidade de armazenamento de textura em memória, uma decomposição em sub-volumes, a qual pode implicar em um esquema de compressão, é aplicada por Boada *et al.* (2001). Estes autores desenvolveram uma técnica de visualização direta de volumes usando texturas 3D, onde a textura é obtida através de uma representação hierárquica, do tipo *octree*, do volume de dados usando dois critérios: homogeneidade e importância dos dados.

A Figura 2.17 apresenta a idéia básica desta técnica, onde a construção da *octree* é realizada numa etapa de pré-processamento do volume de dados, e seu caminhamento na etapa de visualização leva em conta algumas definições do usuário. Um conjunto de nodos desta *octree*, chamado de corte, também pode ser selecionado pelo usuário durante o caminhamento da árvore, determinando um sub-volume a ser visualizado.

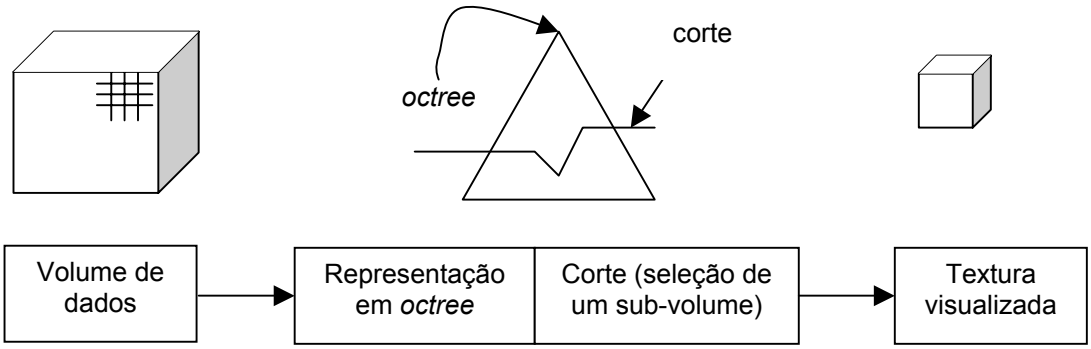


FIGURA 2.17 - Esquema geral do método de visualização volumétrica com multiresolução baseada em *octree* (extraído de Boada *et al.*, 2001)

Para a construção da *octree*, é necessário um critério para determinar quando uma região do volume (sub-volume) é importante. O critério considerado por Boada *et al.* (2001) foi a homogeneidade (coerência espacial) dos dados, com uma seleção *on-line* de regiões de grande interesse, as quais serão representadas em máxima resolução.

Durante a construção da estrutura, a homogeneidade dos dados refere-se aos valores de intensidade das amostras do volume que estão associadas aos nodos da estrutura e que satisfazem um dado intervalo de valores. Para cada nodo, um erro é calculado, representando o grau de homogeneidade das amostras, ou seja, as diferenças entre os valores reais dos *voxels* de uma região representada por um nodo na *octree* e o valor obtido por uma interpolação trilinear dos mesmos. Este erro representa o grau de homogeneidade de um sub-volume coberto pelo nodo.

O caminhamento na *octree* é de cima para baixo (*top-down*) para selecionar os nodos que compõem o corte. Um corte será válido quando o sub-volume resultante garantir que todas as regiões do conjunto de dados correspondentes aos nodos participantes do corte tenham uma representação no volume total, satisfazendo um erro determinado pelo usuário. Em contrapartida, o grau de compressão pode ser baixo considerando apenas a homogeneidade dos sub-volumes e, por essa razão, o usuário tem de definir, além de um limiar de erro aceitável, um valor de importância ao nodo. A importância do nodo apenas determina se está dentro ou fora da região de interesse e pode ser definida pelo usuário através da seleção de uma sub-região 3D do volume de dados.

Finalmente, a visualização do sub-conjunto de nodos (textura 3D) é realizada na ordem de trás para frente, onde um cubo, envoltório do objeto e centrado no nodo central da estrutura, é formado por um conjunto de polígonos paralelos ao plano de projeção.

2.3.2 Mapeamento de Texturas Adaptativas

Esta técnica é fruto do recente trabalho de Kraus e Ertl (2002) e apresenta um método de compressão de dados (2D, 3D e campos de iluminação 4D) para representar mapeamento de texturas adaptativas (*adaptive texture maps*), isto é, mapeamento de texturas com resoluções locais adaptativas e com fronteiras adaptativas. O método trabalha em duas etapas, onde uma decomposição do volume de dados é realizada e o resultado é armazenado em duas texturas. Na primeira etapa, uma textura armazena um fator de escala (resolução) e índices para os dados que são realmente armazenados em outra textura, caracterizando a segunda etapa.

A Figura 2.18 apresenta um exemplo, em duas dimensões, do método. A Figura 2.18(a) representa a primeira etapa, onde os dados são subdivididos em células, as quais armazenam uma coordenada de origem e um fator de escala dos dados que são armazenados em outra textura (Figura 2.18(b)).

Este fator de escala é calculado a partir da célula resultante da decomposição e representa o quão semelhante esta célula é em relação a uma célula vazia. A célula será armazenada na segunda textura seguindo a resolução ou escala calculada. O processo de descompressão é realizado em *hardwar*e, acessando a textura da primeira etapa que resultará em um outro acesso (dependente) à textura da segunda etapa. Os acessos a texturas, principalmente os acessos dependentes, utilizando *hardware* ficarão mais claros a partir do próximo capítulo.

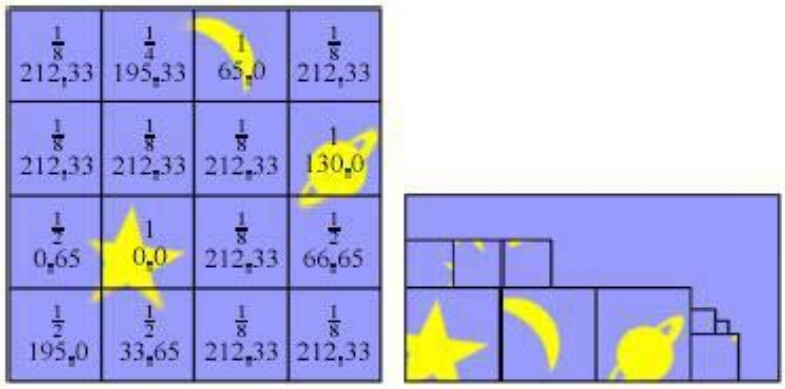


FIGURA 2.18 - Ilustração em 2D do mapeamento de texturas adaptativas. (a) Índice: fator de escala e coordenadas para os dados comprimidos. (b) Dados comprimidos (Kraus e Ertl, 2002).

No processo de visualização do mapeamento de texturas adaptativas, o acesso à textura através de coordenadas (*s, t*) é efetivado realizando os seguintes passos (Figura 2.19):

- determinação da célula na textura de índices que contem o ponto (*s ,t*);
- cálculo das coordenadas (s_0 ,t_0) que correspondem a origem desta célula;
- acesso à textura de índices que corresponde a esta célula para recuperar o fator de escala *m* e a origem (s_0',t_0') correspondente a esta célula na textura de dados comprimidos;
- cálculo das coordenadas (*s',t'*) na textura de dados comprimidos correspondente a (*s,t*) na textura de índices, levando em conta o fator de escala *m*;
- acesso à textura de dados comprimidos através de (*s', t'*) e interpolação.

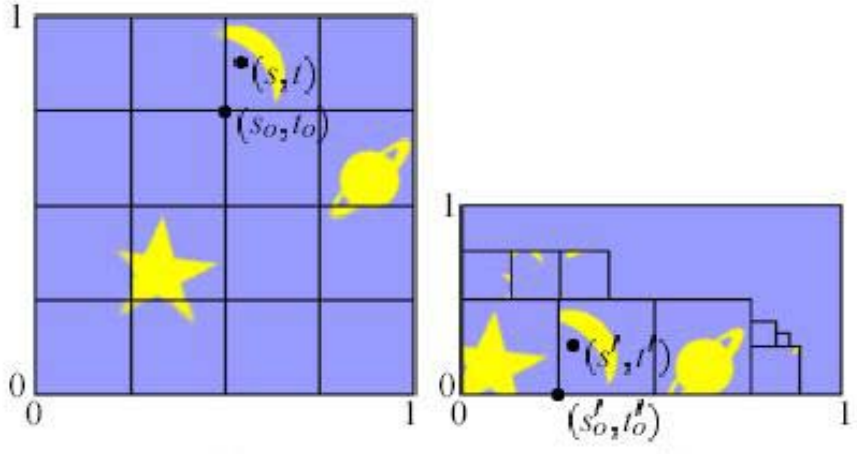
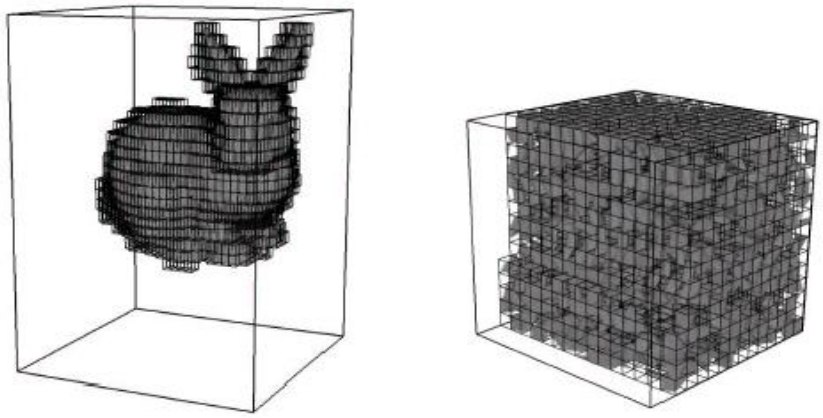


FIGURA 2.19 - Visualização de texturas na técnica de mapeamento adaptativo de texturas (Kraus e Ertl, 2002)

A Figura 2.20 mostra um exemplo para este método com dados volumétricos, onde (a) ilustra a textura de índices e (b) a textura contendo os dados comprimidos.



(a) **(b)**

FIGURA 2.20 - Textura 3D de índices (a) e textura 3D com dados comprimidos (b) (Kraus e Ertl, 2002)

O método também apresentou, em abordagem semelhante aos dados 3D, um exemplo de campos de luz (*light fields*), representados por dados de quatro dimensões (Levoy e Hanrahan, 1996). Para tal, utilizou-se duas texturas 3D para armazenamento dos dados.

É importante ressaltar que este trabalho de Kraus e Ertl (2002) é semelhante à técnica a ser apresentada nesta dissertação, mesmo tendo sido desenvolvidos totalmente independentes. Entretanto, embora este método seja eficiente na visualização de grandes volumes de dados, assim como outros métodos descritos em Cabral *et al.* (1994) e LaMar *et al.* (1999), não aborda a visualização de vários volumes que representam instantes de tempo, ou seja, não aborda visualização volumétrica de dados dinâmicos.

2.3.3 Árvores TSP

Enquanto um volume de dados estático normalmente é caracterizado pela propriedade da coerência espacial, volumes que variam com o tempo apresentam também coerência temporal, ou seja, os valores de intensidade dos *voxels* tendem a não mudar drasticamente de um instante de tempo para o próximo. Embora estas características de coerência sejam um ponto crucial para a exploração de técnicas de compressão de dados, somente recentemente estão sendo exploradas por um pequeno número de pesquisas.

A técnica proposta por Shen *et al.* (1999) apresenta uma estrutura hierárquica chamada árvore TSP (*Time-Space Partitioning*) para visualização de uma série temporal de dados volumétricos, representando-os tanto no domínio espacial quanto temporal. A base desta estrutura é praticamente uma *octree*, mas com o adicional de conter não só informações espaciais, mas também temporais. Para armazenar a informação temporal, cada nodo da árvore TSP é uma árvore binária que divide também, recursivamente, o tempo representado no conjunto volumétrico de dados (Figura 2.21). As informações armazenadas nos nodos da árvore binária são o valor médio dos *voxels* componentes do sub-volume e um erro espacial e temporal referentes a um dado período de tempo.

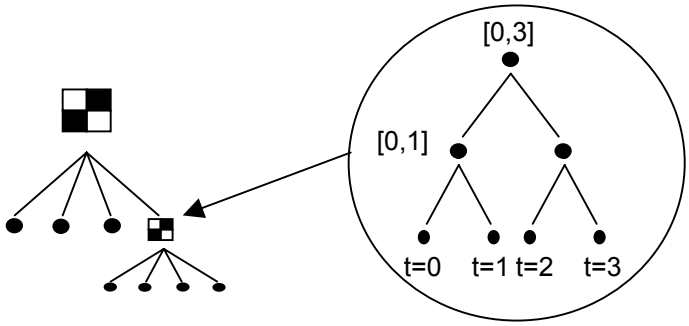


FIGURA 2.21 - Exemplo de árvore TSP para dados 2D, representado 4 instantes de tempo (Shen *et al.*, 1999)

A visualização desta estrutura, utilizando a técnica de *Ray Casting*, é realizada percorrendo apenas os nodos contidos no instante de tempo atual, onde os sub-volumes são visualizados em imagens independentes que são armazenadas em memória *cache*.

Na continuação do trabalho envolvendo árvores TSP (Ellsworth *et al.*, 2000), foi desenvolvida a visualização utilizando texturas 3D em *hardware*, mas com duas alterações em relação à abordagem de mapeamento de texturas 3D padrão (seção 2.2.2.4.2). Uma vez gerados os sub-volumes próprios à visualização, tais sub-volumes serão visualizados seguindo um fator de tolerância de erro espacial: os sub-volumes que estiverem dentro da faixa de erro são visualizados usando uma cor constante enquanto que os outros são visualizados usando uma textura. Neste último caso, os sub-volumes que estiverem na faixa de tolerância temporal e representem um período de tempo são visualizados usando a textura que representa o determinado instante de tempo. Com isso, a técnica permite o compartilhamento de texturas entre os instantes de tempo.

3 *Hardware* Gráfico

Uma das grandes dificuldades da visualização interativa de dados volumétricos é a quantidade de dados envolvida em relação às limitações de capacidade de armazenamento e processamento dos computadores. Apesar do desenvolvimento de algoritmos otimizados, apenas o avanço recente da tecnologia de *hardware*, principalmente em placas gráficas aceleradoras, possibilitou uma grande expansão e utilização destas aplicações.

As placas gráficas aceleradoras vêm incorporando muitos aspectos que antes eram realizados por software e que permitem inúmeros efeitos gráficos complexos, além de conseguir processar cenas compostas por milhões de triângulos em tempo real. Além disto, uma característica importante e inovadora é a possibilidade de programação das funcionalidades existentes na arquitetura do *pipeline* gráfico da placa.

Determinados processos do *pipeline* gráfico são mais suscetíveis ao aspecto de programação do que outros. A unidade ou processo do *pipeline* que trata os vértices dos objetos gráficos, por exemplo, é flexível a operações de programação. A unidade de programação de vértices (*vertex shaders*) poderá realizar várias operações e cálculos com os atributos dos vértices (posição, cor e normal, por exemplo) para produzir efeitos, dentre outros, de iluminação, *skinning*, *morphing* e *bump mapping*.

Outro processo importante do *pipeline* que pode ser programado é aquele que gera a cor final de um *pixel* na tela. A unidade de programação de *pixels* ou fragmentos (*pixel* ou *fragment shader*) opera com os atributos de cada *pixel* para a composição final da cor de cada fragmento. Esta unidade de programação produz efeitos muito interessantes, já que se pode fazer acessos a várias texturas próprias do *hardware* para combinar o resultado destes acessos aos próprios atributos dos *pixels*. Estes acessos a texturas vêm sendo explorados por pesquisas utilizando texturas 3D na visualizaçào de dados volumétricos.

A Figura 3.1 apresenta um fluxo de dados geral da GPU (Unidade Gráfica de Processamento) do *hardware*, contendo os processos programáveis e os não-programáveis.

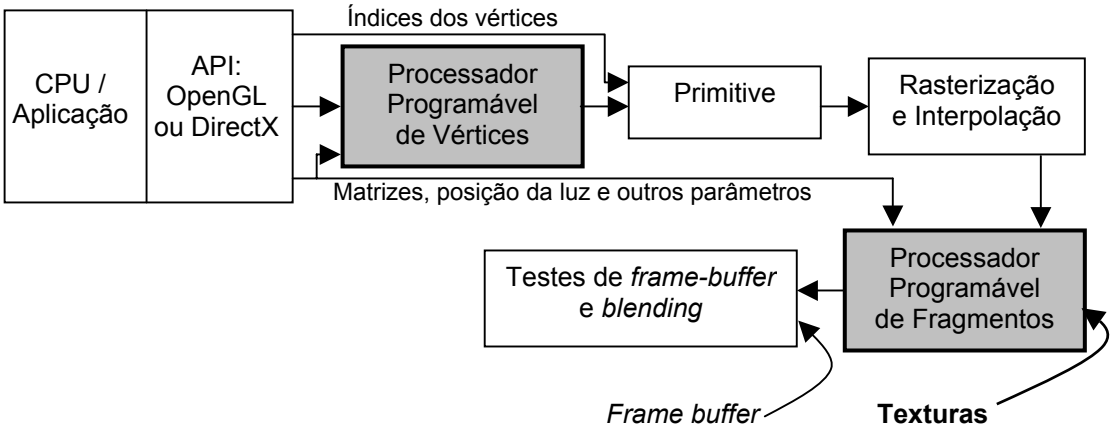


FIGURA 3.1 - Fluxo de dados e unidades programáveis do *hardware* (NVIDIA, 2003)

Para programar estas unidades do *pipeline* gráfico do *hardware*, uma API (*Application Program Interface*) foi desenvolvida pela empresa *NVIDIA Coorporation*, denominada *C for Graphics* (Cg). A linguagem Cg permite um alto nível de abstração do *hardware* gráfico semelhante à linguagem C, tanto em sua sintaxe quanto no próprio nível de abstração, podendo ser compilada para gerar código em aplicações que utilizam DirectX ou OpenGL. Ao longo deste capítulo, exemplos de códigos de *vertex* ou *fragment shaders* serão apresentados em Cg admitindo que o leitor tenha um prévio conhecimento da linguagem, que pode ser obtido através de um tutorial apropriado (NVIDIA, 2003).

A seguir, estas duas unidades principais de programação em *hardware* são apresentadas juntamente com exemplos descritos na linguagem de programação Cg. Entretanto, uma ênfase maior será abordada para as características da unidade *fragment shader*, pois este trabalho utilizará fundamentalmente as propriedades das texturas 3D presentes no *hardware*.

3.1 Unidade de Programação de Vértices (*vertex shader*)

Primitivas simples, como triângulos, são usadas para representar diversas formas ou objetos. As principais informações destas primitivas estão relacionadas com seus vértices. Tais informações dos vértices são enviadas a GPU e representam, por exemplo, posições (x,y,z,w), normais (x,y,z), cores (r,g,b,a) ou coordenadas de textura (s,q,r,t). Em seguida, transformações geométricas, cálculo de iluminação e geração de coordenadas de textura são executadas pela unidade de *vertex shaders*.

Desta forma, para esta unidade programável do *pipeline* gráfico do *hardware,* um programa é composto por uma seqüência de instruções que executa operações a partir de parâmetros compostos por características dos vértices dos dados e resulta em outros parâmetros de saída referentes a cada vértice. Uma representação esquemática do modelo de *vertex shader* está descrita na Figura 3.2. Estes programas acessam quatro tipos de unidades de memória: atributos dos vértices (apenas leitura), parâmetros de entrada para o programa (apenas leitura), registradores temporários (leitura e gravação) e registradores de saída (apenas gravação).

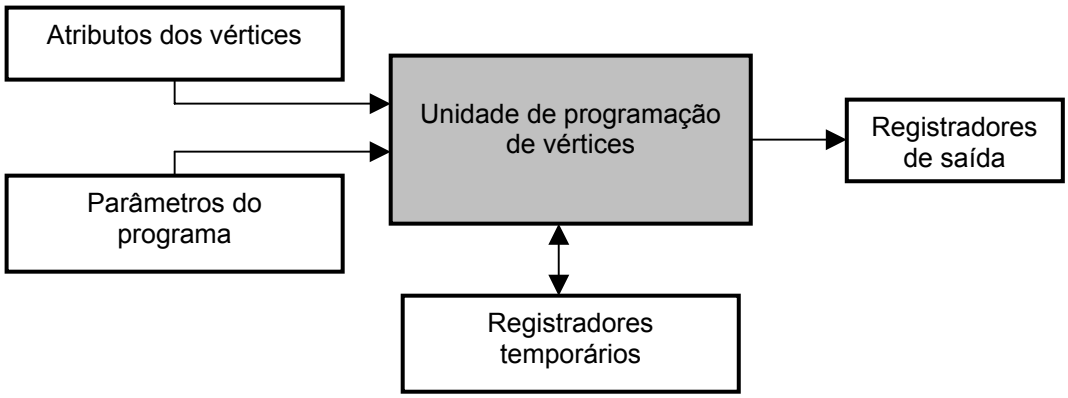


FIGURA 3.2 - Modelo do *vertex shader*

Os dados referentes a cada vértice de entrada são armazenados nos registradores de atributos dos vértices. Estes dados representam, entre outros parâmetros, posição,

cor, normal e coordenadas de textura. A aplicação também pode enviar parâmetros que não têm ligação direta com os vértices, proporcionando a criação de efeitos especiais. Estes parâmetros são, usualmente, matrizes de transformação, parâmetros de iluminação, valores pré-computados, etc.

A partir destes dados fornecidos pela aplicação, a unidade de programação executa as operações especificadas e pode armazenar nos registradores temporários resultados intermediários durante a execução do programa para uma operação posterior. Após a execução completa do programa, os resultados são armazenados nos registradores de saída.

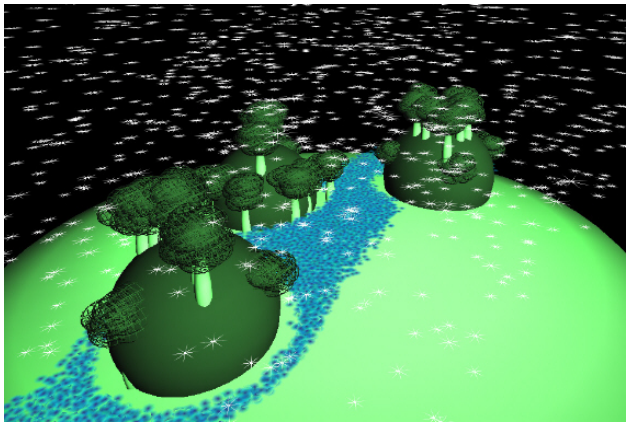
3.1.1 Efeitos Especiais

Muitos efeitos especiais podem ser obtidos usando a programação em *hardware* de vértices. Abaixo, estão descritos alguns efeitos publicados (NVIDIA, 2001):

- ***Skinning***: a animação de esqueletos requer a aplicação de pesos aos vértices do modelo, proporcionais ao impacto de cada osso. Esta técnica também requer a aplicação de inúmeras matrizes de transformação (movimentação) para cada vértice;
- **Deformação dinâmica de superfícies**: a mudança de posição dos vértices permite com que os *vertex shaders* executem deformações no formato de um objeto. Esse movimento dos vértices pode ser regido por uma simples função ao longo do tempo;
- **Modelagem procedural**: semelhante a deformações, a modelagem procedural pode ser usada para descrever o formato de um objeto utilizando a programação em *hardware* de vértices. Sistemas de partículas também podem ser criados usando o mesmo conceito;
- ***Morphing***: determinados valores de intensidade representando pesos são definidos para cada vértice e utilizados para transformar um objeto em outro;
- **Distorção**: a criação de matrizes de transformação permite um efeito de distorção diferente dos efeitos criados por um *pipeline* fixo de vértices. Um exemplo desta técnica são os efeitos do tipo “olho de peixe”;
- **Ambiente**: efeitos de neblina podem ser obtidos mudando em tempo real as coordenadas da neblina, que dependerão de sua altura ou elevação.

3.1.2 Exemplo em Cg

Um exemplo muito interessante utilizando *vertex shaders* foi desenvolvido no grupo de Computação Gráfica da UFRGS (Dietrich e Comba, 2003), denominado originalmente *Repulsive Potencial Fields*. Este trabalho implementa os conceitos de modelagem procedural e sistemas de partículas para representar o movimento de neve caindo em um ambiente composto pelo fluxo de um rio e alguns obstáculos. Todas as partículas (neve e água do rio) desviarão destes obstáculos a partir de cálculos de repulsão realizados com os vértices dos elementos que formam tais partículas. A Figura 3.3(a) mostra uma imagem resultante deste exemplo e (b) ilustra o código em Cg para o *vertex shader*.



(a)

```
struct app2vert : application2vertex {
        float4 position : POSITION;
        float3 normal   : NORMAL;
        float4 color    : DIFFUSE;
        float4 disloc   : SPECULAR;
};
struct vert2frag : vertex2fragment {
        float4 HPOS : POSITION;
        float4 COL0 : COLOR0;
};
vert2frag main( app2vert IN,
                uniform float4x4 ModelViewProj, uniform float4x4 control,
                uniform float4x4 obstacle     , uniform float4   clock ) {
        vert2frag OUT; float t, it; float4 P, O;

        //> FLUXO DO RIO
        t   = IN.disloc.w + clock.x / 6000.0f; // flow
        t   = (t > 1.0f) ? t - 1.0f : t;
        it  = 1.0f - t;
        P.x = it * it * it;       // P0
        P.y = 3.0f * t * it * it; // P1
        P.z = 3.0f * t * t * it;  // P2
        P.w = t * t * t;          // P3
        OUT.HPOS.x = dot( P, float4(control[0][0], control[1][0], control[2][0],
                                    control[3][0]) );
        OUT.HPOS.y = dot( P, float4(control[0][1], control[1][1], control[2][1],
                                    control[3][1]) );
        OUT.HPOS.z = dot( P, float4(control[0][2], control[1][2], control[2][2],
                                    control[3][2]) );
        OUT.HPOS   = OUT.HPOS + IN.disloc;
        OUT.HPOS.w = 1.0f;
        //< FLUXO DO RIO
        //> REPULSIVIDADE DO RIO E DA NEVE AO LONGO DO OBSTACULO
        float g, d, s;  float4 N;

        N   = OUT.HPOS - obstacle[0]; N.w = 0.0f;
        d   = length( N );
        N   = normalize( N );
        s   = 1.0f / (obstacle[0].w * obstacle[0].w);
        g   = exp(-d * d * s) * obstacle[0].w; // obstacle power
        N   = N * g;
        O   = OUT.HPOS + N + IN.position;
        //< REPULSIVIDADE DO RIO E DA NEVE AO LONGO DO OBSTACULO

        O.w      = 1.0f;
        OUT.HPOS = mul( ModelViewProj, O );
        OUT.COL0 = IN.color;
        return OUT;
}
```

(b)

FIGURA 3.3 - Exemplo do efeito especial (a) e seu respectivo código em Cg (b) para *vertex shader*: *Repulsive Potential Fields* (Dietrich e Comba, 2003)

Embora o exemplo da Figura 3.3 não seja aqui detalhado minuciosamente, já que não é o objetivo principal deste trabalho, vale ressaltar que cada vértice contém quatro propriedades: posição, normal, cor e um valor de deslocamento. Além destes parâmetros oriundos das características dos vértices (IN), mais quatro parâmetros de entrada ao programa foram definidos: matriz de projeção (ModelViewProj), matriz com pontos controle de uma curva *Bèzier* (control) que representa a trajetória do rio, matriz representando a posição e o raio do obstáculo (obstacle) e um valor que representa o tempo (clock). A partir destes dados, cálculos são realizados no procedimento principal e os resultados produzidos (posição e cor), referentes a cada vértice, são gravados nos registradores de saída. Este procedimento principal, por sua vez, divide-se em duas etapas: execução do cálculo do fluxo das partículas do rio na trajetória da curva estipulada e cálculo da repulsão das partículas de água e neve pelo obstáculo.

3.2 Unidade de Programação de Fragmentos (*pixel* ou *fragment shader*)

Embora a programação em *hardware* com vértices ofereça grande flexibilidade para cálculos e interpolação de dados em baixa freqüência (por vértices), a programação por fragmento ou por *pixel* é necessária para informações de alta freqüência, isto é, informações que mudam muito mais rapidamente e que não conseguem ser capturadas pelos vértices.

Este controle de dados no nível de *pixel* evoluiu não só com o desenvolvimento do *hardware*, mas também com a evolução do *pipeline* gráfico. Enquanto algumas *placas gráficas* ainda disponibilizam apenas uma textura (1D, 2D ou 3D) para mapeamento, os avanços recentes introduziram múltiplas texturas. Através da unidade de programação por fragmento, efeitos especiais interessantes podem ser obtidos através de múltiplas texturas. A cor final de um *pixel*, por exemplo, pode ser determinada através da combinação de vários acessos às diferentes texturas disponíveis com outros atributos, como valores de iluminação difusa e especular.

Para aumentar ainda mais o poder de combinação dos dados, o *pipeline* gráfico pode ainda manipular os valores de acesso às texturas. Conectando acessos seqüenciais às texturas disponíveis, coordenadas de acesso a uma determinada textura podem depender diretamente do resultado de um acesso prévio a outra textura. Isto quer dizer que o resultado do acesso a uma textura, com coordenadas de textura (s,t,r,q), pode não representar uma cor no sistema RGBA (r,g,b,a) e sim novas coordenadas de textura (s',t',r',q') para um próximo acesso. A este processo deu-se o nome de "texturas dependentes" e para a unidade programável que envolve os acessos a texturas e cálculos com seus resultados, chamou-se *texture shader* ou unidade de programação de texturas. Esta unidade do *hardware* desempenha papel importante neste trabalho.

3.2.1 Unidade de Programação de Texturas (*texture shader*)

A funcionalidade das texturas dependentes permite uma grande flexibilidade e um alto poder computacional em nível de *pixel*, já que as texturas poderão conter não apenas cor e sim informações codificadas que podem ser trabalhadas na execução de uma função, gerando a cor final do *pixel*. A Figura 3.4 exemplifica o *pipeline* referente ao hardware da GeForce4, que permite a implementação do conceito de texturas dependentes.

Na Figura 3.4, os *pixels* oriundos da unidade rasterização passam por uma seqüência de até três acessos a texturas dependentes e as cores resultantes são armazenadas nos registradores de saída, que podem ser combinadas a outros atributos ou cores armazenadas em um *buffer* (*frame buffer*) para geração da cor final do *pixel*.

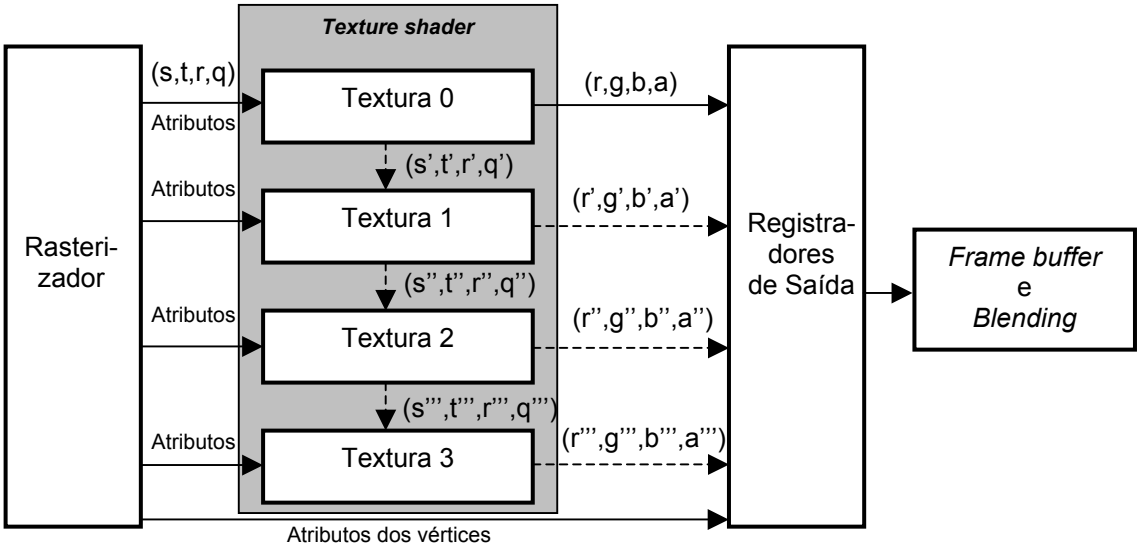


FIGURA 3.4 - *Fragment shader* do *hardware* GeForce4

Assim, a unidade de programação de texturas ou simplesmente *texture shader* consiste, no *hardware* da GeForce4, em até quatro estágios de manipulação de acessos a texturas (Figura 3.4). Cada estágio executa determinadas operações nas coordenadas de textura em questão para produzir tanto um novo conjunto de coordenadas de textura para o próximo estágio (texturas dependentes) quanto a cor final do *pixel* no estágio atual. O programa da unidade *texture shader* determina, então, o comportamento do acesso à textura seguindo uma possível dependência implícita em relação aos estágios anteriores e uma possível combinação ou manipulação dos valores de cor obtidos.

Um exemplo da aplicabilidade das texturas dependentes encontra-se no trabalho de Engel *et al.* (2001). Este trabalho apresenta dois algoritmos de visualização volumétrica baseados em textura, um para textura 2D e outro para textura 3D, ambos utilizando o conceito de textura dependente presente no *hardware* GeForce3. Uma determinada textura (a ser utilizada como dependente), em ambos algoritmos, contém valores escalares obtidos a partir de uma pós-classificação (aplicação de uma função de transferência após interpolação) do conjunto de volumes de dados. Durante a visualização dos polígonos a serem mapeados pela textura principal, dois pares de valores escalares são amostrados de dois planos (polígonos) adjacentes (Figura 3.5), onde cada *pixel* de um plano corresponde à contribuição de cor e opacidade ao longo de um segmento de raio. Como cada *pixel* destes dois polígonos adjacentes corresponde ao mesmo raio, uma composição destes planos em um único valor é realizada. Assim, estes dois pares de valores escalares são usados para realizar um acesso dependente à textura criada pela pós-classificação, a qual contém os valores de cor e opacidade pré-calculados.

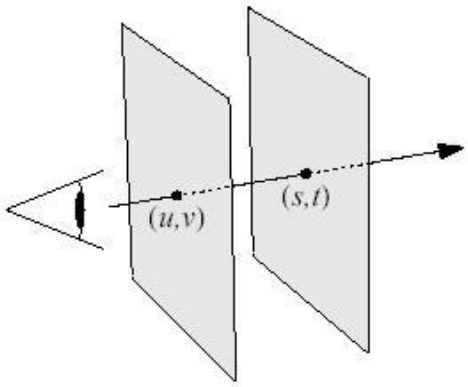


FIGURA 3.5 - Par de planos adjacentes (Engel *et al.*, 2001)

3.2.2 Efeitos Especiais

Vários efeitos especiais podem ser obtidos usando *fragment shaders*. Abaixo, estão descritos alguns efeitos publicados (NVIDIA, 2001):

- ***Depth sprites***: visualização de objetos tridimensionais através de uma operação de mapeamento de textura em uma superfície plana. O valor de profundidade dos *pixels* componentes da superfície são trocados por valores provenientes de um acesso a textura que contém um mapa de alturas. O *texture shader* é usado na troca destes valores de profundidade e outras unidades do *fragment shader* são usados na composição da iluminação da geometria resultante destes *pixels*;
- ***Bump mapping* especular e difuso usando a iluminação de Phong**: utiliza *texture shader* para o cálculo dos parâmetros que compõem a equação de iluminação especular e difusa;
- **Iluminação para *shadow mapping***: combina o uso de *vertex shader* e *texture shader* para calcular iluminação difusa e especular de cada *pixel* com utilização de texturas contendo informações de sombras;
- **Reflexão e refração**: utiliza o *texture shader* para visualização de reflexão e refração de superfícies em *bump mapping*;
- **Visualização de superfícies transparentes**: a visualização da sobreposição de superfícies transparentes é realizada na ordem caracterizada de trás para frente (*back to front*). O uso de *shadow mapping* em *hardware* combinado com *texture shader* (manipulação com valores *alpha*) propicia a correta visualização destas superfícies.

3.2.3 Exemplo em Cg

Um exemplo simples de um programa em Cg para *texture shader* foi extraído de NVIDIA (2003) e é apresentado na Figura 3.6. Este exemplo combina a técnica de mapeamento de ambiente (*environment mapping*) com reflexão, transformando a normal, extraída do mapa de normais, no espaço que representa os dados de ambiente (textura do ambiente), refletindo esta normal a partir do vetor de visualização. Finalmente, se extrai os dados da textura de ambiente para obter a cor final do *pixel*, gerando um efeito de reflexão do ambiente em que se encontra o objeto em questão.

Os atributos de entrada do programa são: coordenadas de textura pré-calculadas (IN) para acesso a textura de ambiente, textura 2D contendo as normais (normalMap), textura contendo os dados do ambiente externo (environmentMap) e o vetor de visualização (eyeVector). Na primeira etapa, um acesso à textura de normais é realizado para recuperar, obviamente, o vetor normal do *pixel*. A segunda etapa utiliza a seguinte instrução: *texCUBE_reflect_eye_dp3x3*. Esta instrução utiliza a normal calculada, já recuperada no espaço da textura de ambiente, juntamente com o vetor de visualização e as coordenadas de textura, para calcular o vetor de reflexão. A mesma instrução utiliza este vetor de reflexão para, automaticamente, acessar a textura contendo os dados de ambiente e retornar cor de *pixel*.

(a)

```
struct vert2frag
{
        // Coordenadas de textura transformadas no espaço da textura cube map
        float4 tangentToCubeSpace0: TEXCOORD1;
        float4 tangentToCubeSpace1: TEXCOORD2;
};
struct frag2frame
{
        float4 color : COLOR0;
};

frag2frame main ( vert2frag IN,
                        uniform sampler2D normalMap,
                        uniform samplerCUBE environmentMap,
                        uniform float3 eyeVector
                ) : COLOR
{
        frag2frame OUT;

        // Acesso a textura que contém os valores de normais
        float4 normal = tex2D(normalMap);
        // Acesso a textura de CUBE MAP
        OUT.color = texCUBE_reflect_eye_dp3x3(environmentMap,
                                              IN.tangentToCubeSpace0,
                                              IN.tangentToCubeSpace1,
                                              normal,
                                              eyeVector);

        return OUT;
}
```

(b)

FIGURA 3.6 - Exemplo do efeito especial (a) e seu respectivo código em Cg para *fragment shader*: *environment mapping*, extraído de NVIDIA (2003)

4 Compressão e Visualização de Dados Volumétricos Dinâmicos Usando Texturas 3D

Este capítulo apresenta a contribuição essencial deste trabalho mostrando o desenvolvimento de um mecanismo de compressão e descompressão (para visualização) de dados volumétricos dinâmicos. O método é dividido em duas etapas:

- decomposição dos dados volumétricos dinâmicos e armazenamento dos sub-volumes, resultantes da decomposição, em texturas 3D (etapa de compressão);
- visualização das texturas 3D geradas usando características do *hardware* gráfico (etapa de descompressão).

4.1 Compressão

A etapa de compressão de informação codifica os múltiplos volumes de dados e os armazena para visualização posterior. O método de compressão proposto no presente trabalho é baseado em alguns pressupostos ou propriedades fundamentais dos dados, descritos a seguir.

4.1.1 Propriedades Básicas dos Dados

Dados volumétricos dinâmicos podem apresentar inúmeras informações associadas a cada ponto amostrado, dependendo da aplicação (velocidade, vorticidade, temperatura, salinidade, viscosidade, entre outros). Também podem apresentar outras características mais abstratas que não estão associadas diretamente a um determinado ponto amostrado e sim ao conjunto completo dos dados.

Uma destas características, que é de grande importância para um método de compressão, é o fato dos dados serem esparsos e, além disso, se concentrarem em apenas algumas regiões da área total amostrada, com um comportamento semelhante ao de uma matriz esparsa. Para estas matrizes, existem vários métodos de compressão descritos na literatura, que variam de acordo com o quão esparso é o conjunto de dados e sua localização ou concentração na matriz. Porém, os métodos de compressão de matrizes esparsas são usados na solução de sistemas lineares que possuam propriedades específicas (matrizes diagonais, triangulares, etc). Portanto, tais métodos não se aplicam diretamente para dados volumétricos esparsos que não possuem tais propriedades.

Outra característica fundamental é a coerência, muito explorada por Shen *et al.* (1999) e Ellsworth *et al.* (2000), conforme já mencionado na seção 2.3 do capítulo 2. Os dados dinâmicos possuem um alto grau de coerência, tanto espacial como temporal. A Figura 4.1 representa uma idéia desta propriedade, em duas dimensões, para instantes de tempo a partir de t_n.

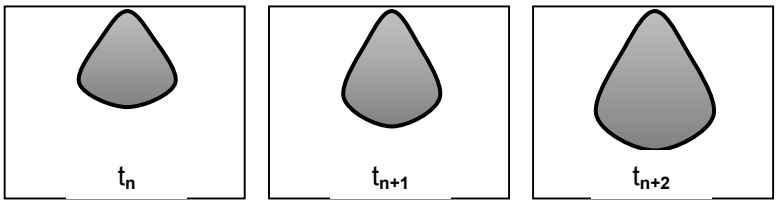


FIGURA 4.1 - Exemplo de coerência espacial e temporal

A coerência espacial está bastante presente em dados de dinâmica de fluidos (CFD), por exemplo, onde a velocidade e direção regem a posição do material ao longo do volume de dados, sendo que os valores das células vizinhas em relação ao valor de uma determinada célula da grade não mudam abruptamente. Já o outro tipo de coerência, a temporal, é freqüentemente encontrada em uma série qualquer de volumes de dados, cada volume representando um instante de tempo, onde o valor de uma determinada célula também não muda abruptamente de um instante (portanto, um volume) para outro.

Usada apropriadamente, a coerência pode ser explorada para acelerar consideravelmente o desempenho da visualização, que ainda é muito difícil de ser executada em tempo real para visualização direta de volumes. Outra possibilidade de aceleração através desta propriedade é reduzir o excesso de I/O (*input/output*), pois volumes iguais ou muito semelhantes podem ser armazenados uma única vez e reutilizados.

Estas duas propriedades (dados esparsos e coerência) permitem compactar a informação, o que é extremamente interessante, pois tais dados tendem a ser volumosos. Este capítulo apresenta um novo mecanismo de compactação que se beneficia das características do *hardware* gráfico descrito no capítulo 3, utilizando as propriedades das texturas 3D dependentes para acelerar o processo de visualização.

Feito isso, será possível realizar a visualização volumétrica em tempo real de dados dinâmicos, resultantes de simulações geradas por modelos em CFD ou por qualquer outra área que contenha dados variantes ao longo do tempo.

4.1.2 Dados de Entrada para o Mecanismo

Assume-se, então, que os dados de entrada consistam de diferentes instâncias de tempo, cada qual representada por um volume de dados esparsos. A dimensão de cada volume não é limitada, mas por restrições de memória do *hardware* gráfico atual serão utilizados volumes com 128^3 células.

Cada volume passa por uma etapa de classificação de valores, onde cada célula corresponde a um valor escalar de intensidade que pode variar de 0 a 255 no sistema RGBA, como ilustrado na Figura 4.2.

Todos os *voxels* de cada volume de dados serão submetidos, então, ao método de compressão denominado ***Gritex,*** descrito a seguir.

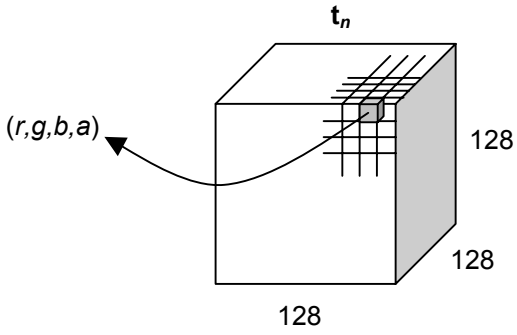


FIGURA 4.2 - Volume de dados

4.1.3 O Método de Compressão *Gritex*

A abordagem usada para este método consiste em um mecanismo de indexação dividido em duas etapas dependentes, pois é baseado nas texturas dependentes descritas no capítulo 3.

Resumindo o método, para um entendimento preliminar (ver Figura 4.3), a primeira etapa consiste em uma decomposição regular de cada volume de entrada em vários sub-volumes, até um determinado nível estipulado, correspondendo a uma subdivisão hierárquica dos dados. Neste passo, o objetivo é separar claramente as regiões que necessitam de um maior detalhamento daquelas que não precisam de refinamento. A primeira etapa resulta no armazenamento em uma primeira textura 3D, de valores correspondentes aos sub-volumes plenamente "resolvidos", pois são homogêneos, ou referências para uma segunda textura 3D, onde serão armazenados os sub-volumes heterogêneos que precisam ser refinados.

Na segunda etapa, para cada sub-volume onde o refinamento é necessário, tais refinamentos são criados, ou seja, prossegue a subdivisão até a unidade de *voxel*. Todos estes sub-volumes refinados serão armazenados na segunda ou em outras unidades de textura 3D, conforme necessário.

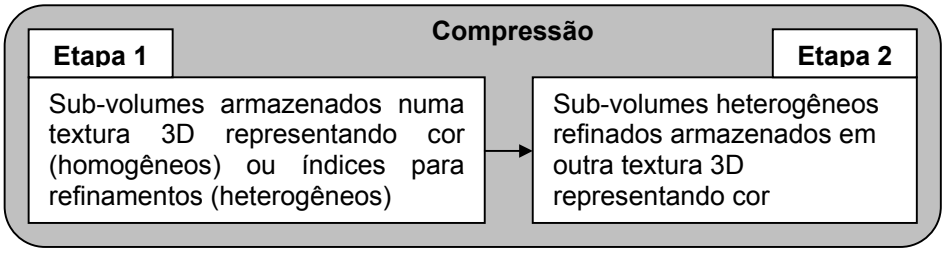


FIGURA 4.3 - Esquema geral do método de compressão

4.1.3.1 Primeira Etapa: Índice

Em geral, estruturas de dados hierárquicas convencionais são apropriadamente usadas para caracterizar a homogeneidade dos dados no domínio espacial. Com isso, nesta primeira etapa, cada volume de dados é decomposto através de um processo

hierárquico um pouco similar à técnica *octree* (Mäntylä, 1988). Em *octrees*, um volume é dividido em 8 sub-volumes (cubos) de igual tamanho chamados de nodos, que poderão conter valores constantes ou variáveis. Os cubos que contiverem valores constantes são denominados cheios (valores maiores que zero) ou vazios (valor igual a zero), e são considerados **homogêneos**. Caso contrário, são chamados de mistos ou **heterogêneos**, requerendo subdivisões adicionais até que seja alcançada a homogeneidade. A Figura 4.4 mostra esta decomposição, onde os nodos homogêneos são caracterizados pelas cores branca (vazio) e preta (cheio) e os heterogêneos pela cor cinza.

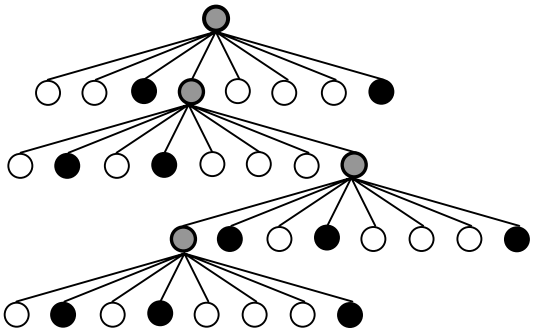


FIGURA 4.4 - Estrutura de uma *octree*

Entretanto, a natureza adaptativa da *octree* não é diretamente aproveitada pelas texturas 3D em *hardware*, tanto por sua irregularidade como também pelo fato de não ser preciso armazenar informações relacionadas à hierarquia em cada nodo. Para forçar esta regularidade indispensável ao uso das texturas 3D em *hardware* e para facilitar o processo de visualização, constrói-se uma estrutura semelhante à *octree* completa (com todos os nodos filhos) até uma certa profundidade ou nível, ou seja, um *grid*. Este nível, obviamente, tem de ser menor que a profundidade máxima e é definido de acordo com as características dos volumes de dados.

A diferença clara entre esta estrutura e uma *octree* é verificada quando a *octree* encontra um nodo homogêneo, finalizando a subdivisão. Num *grid*, existem subdivisões sucessivas até o nível pré-determinado, independente da homogeneidade do nodo em questão. Outra diferença presente é que um *grid* não guarda as informações hierárquicas da estrutura de dados.

A Figura 4.5 ilustra a estrutura do *grid*, apresentando a divisão do volume em sub-volumes até o segundo nível da *octree*.

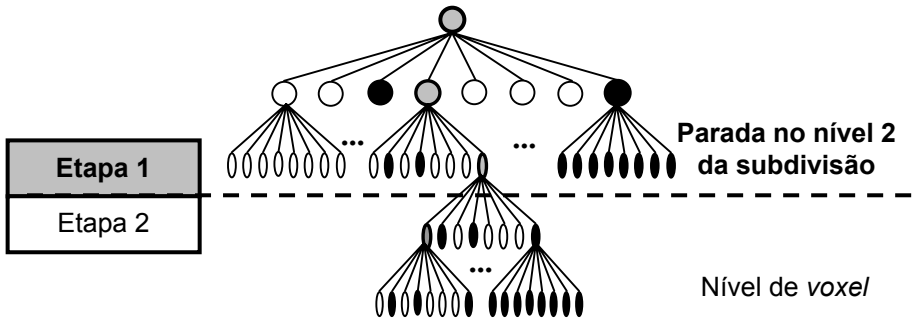


FIGURA 4.5 - Estrutura de um *grid* para o nível 2

É muito importante frisar que, dependendo das características dos dados de entrada (mais ou menos esparsos e concentrados em apenas algumas regiões da amostra), é possível estimar o melhor ponto de parada do processo de subdivisão na primeira etapa. Isto significa dizer que este ponto de parada não é fixo e pode ser alterado adequadamente.

Na segunda etapa, como será visto, apenas os sub-volumes heterogêneos serão novamente subdivididos até o nível de *voxel*, independente da homogeneidade dos nodos.

Para os dados atuais de entrada (volumes de dimensão 128^3) e os exemplos ao longo de todo este trabalho, a subdivisão será processada até o nível quatro da árvore, um nível considerado intermediário. A Figura 4.6 apresenta esta estrutura, sendo mostrado, do lado esquerdo a dimensão de cada cubo ou célula e do lado direito, o nível da subdivisão, juntamente com o número total de cubos resultantes. Na figura, para facilitar a visualização, as oito subdivisões para cada nodo estão representadas apenas por duas e todos os valores estão descritos em potência de dois.

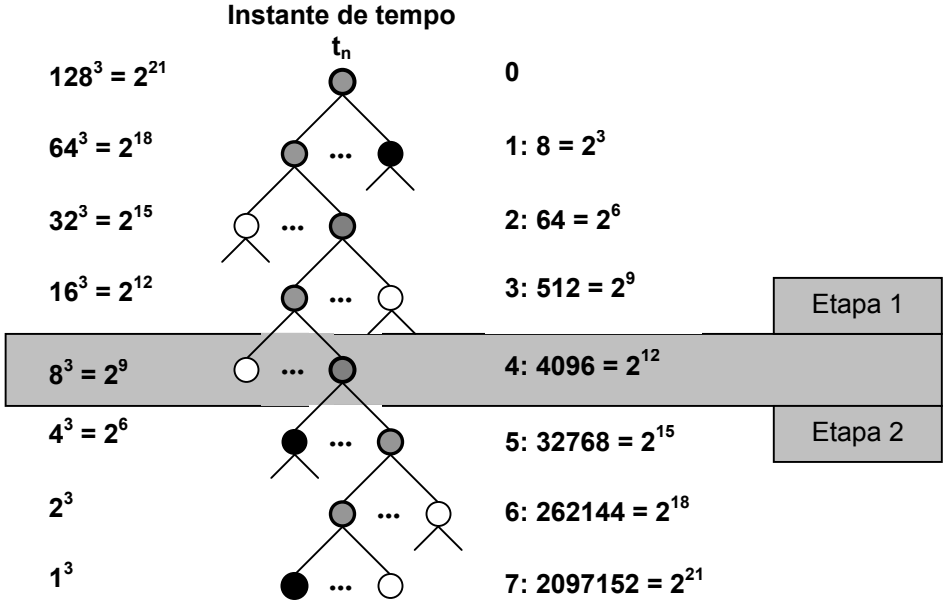


FIGURA 4.6 - *Grid* para um volume de 128^3. No lado esquerdo da árvore tem-se o tamanho de cada célula e no lado direito o nível da subdivisão juntamente com a quantidade máxima comportável de sub-volumes

Ao final da primeira etapa, o *grid* é armazenado na primeira textura 3D da seguinte forma:

- **sub-volumes homogêneos**: os valores armazenados na textura são os próprios valores de intensidade de cor e opacidade no sistema RGBA, onde a componente *alpha* varia de 0 a 0,9;
- **sub-volumes heterogêneos**: um índice de acesso ao refinamento é calculado e armazenado nessa textura, iniciando a segunda etapa do processo de compressão. Os valores armazenados correspondem a (*ox*, *oy*, *oz, T*), onde esse índice corresponderá à origem (*ox*, *oy*, *oz*) do sub-volume e a identificação da textura (*T*), dita "textura de refinamento", que

armazenará o sub-volume. Neste caso, a componente *alpha* representa a identificação da textura e assume um valor maior ou igual a 1.

Os valores da origem no caso de sub-volumes heterogêneos são determinados simplesmente como a primeira posição (*x*,*y*,*z*) livre na textura de refinamento e todos os valores referentes a este refinamento serão armazenados a partir desta origem. A informação da origem na textura de índices, representando uma posição inicial do sub-volume na textura de refinamento, é armazenada como valores *r*, *g* e *b*, e a informação de identificação da textura é armazenada como canal alfa na representação RGBA.

Exemplificando, o nível quatro do *grid* apresenta 4096 nodos, que correspondem ao primeiro índice de um dado instante de tempo. Esta informação é armazenada em uma das texturas 3D disponíveis no *hardware*. Como estas texturas também apresentam dimensão 128^3, será possível armazenar 512 (128^3/4096) destes *grids*, que representam 512 instâncias de tempo. Cada *grid* neste nível terá dimensões 16^3, o que corresponde a ter 16 sub-volumes de 8^3, considerando a dimensão de 128^3 $(128/8)^3$. Este armazenamento caracteriza a primeira etapa da indexação do método de compressão. A Figura 4.7 ilustra esta primeira etapa da compressão. Nota-se também que este armazenamento de *grids* em texturas origina o nome do método, *gritex*.

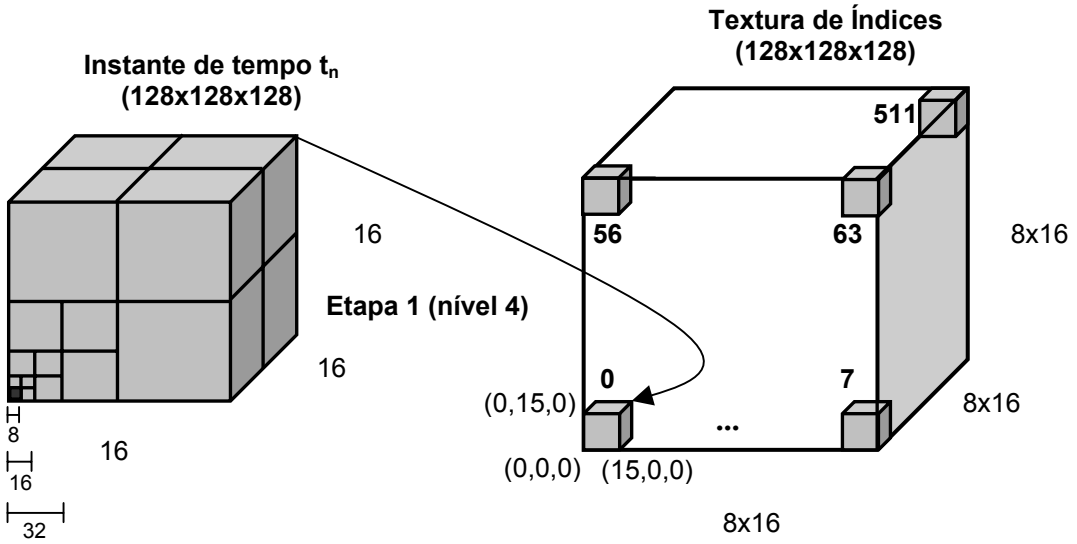


FIGURA 4.7 - Esquema de armazenamento da primeira etapa de compressão

4.1.3.2 Segunda Etapa: Refinamento

Na segunda etapa do método *Gritex*, desenvolve-se a subdivisão dos sub-volumes heterogêneos, isto é, um refinamento ou subdivisão completa dos cubos é criado. O resultado deste refinamento será armazenado em outra textura 3D, caracterizando a segunda etapa do método de compressão, de acordo com a Figura 4.8. Esta textura, por sua vez, será acessada a partir da primeira textura (textura de índices), através do conceito de dependência de texturas.

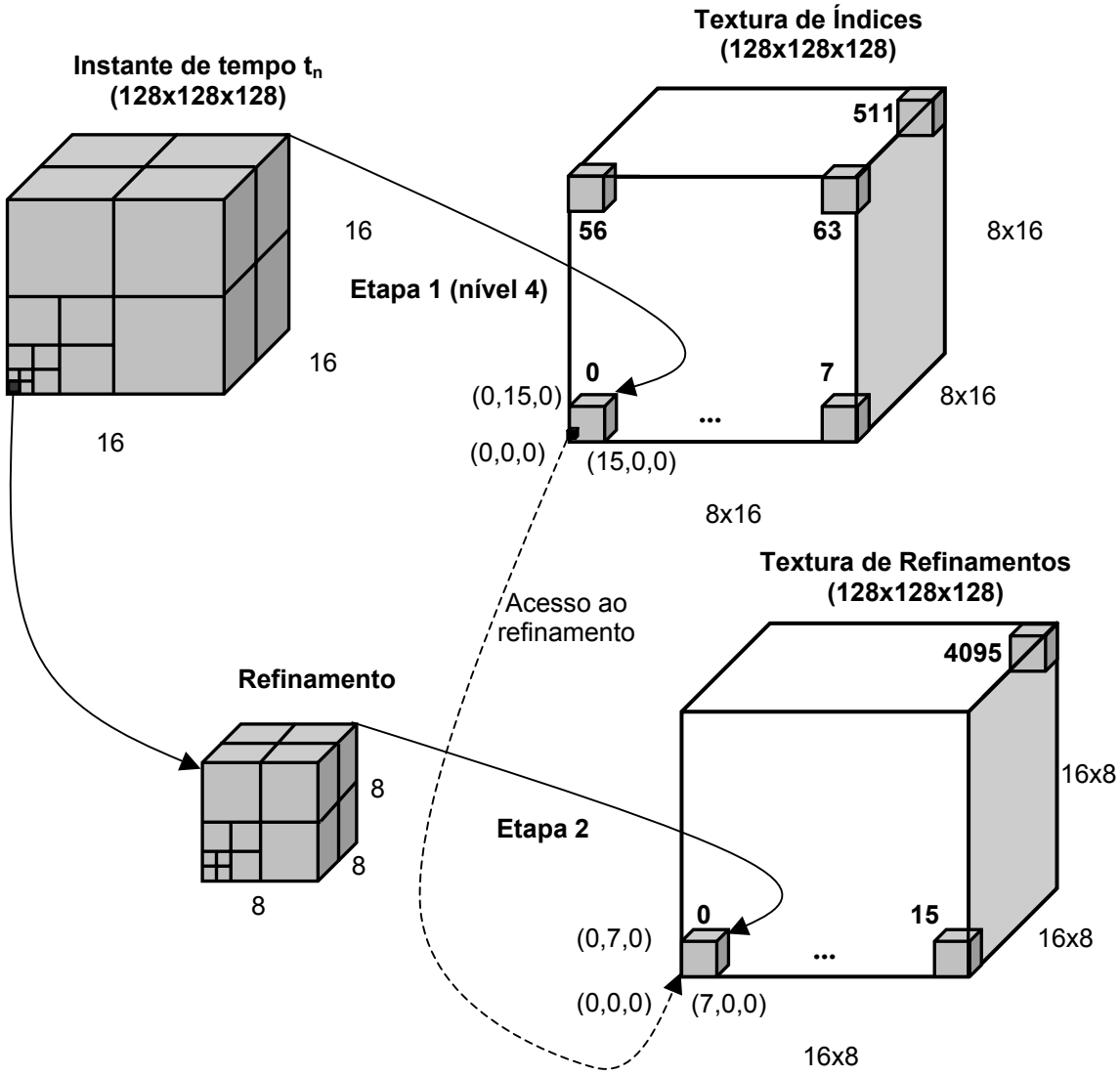


FIGURA 4.8 - Esquema geral de armazenamento da compressão

O *hardware* gráfico atual disponibiliza quatro texturas 3D e o método de compressão está utilizando, até o momento, apenas duas, restando outras duas. Com isso, estas outras duas texturas 3D poderão comportar mais refinamentos, caso o espaço de armazenamento da segunda textura se esgote.

De forma a explorar os conceitos de coerência espacial e temporal vistos anteriormente, tenta-se re-usar ao máximo a informação de refinamento, armazenando os valores já alocados em uma tabela de *hash*. Esta abordagem aumenta a capacidade de armazenamento dos refinamentos, comparando cada refinamento novo com refinamentos criados previamente e apenas armazena-se os refinamentos distintos.

Caso haja um refinamento igual a um já armazenado (duplicado), ou seja, caso ocorra uma "colisão", haverá o re-uso dos índices (origem e identificador da textura) relativos a este refinamento na textura de índices. A função *hash* utilizada neste trabalho é extremamente simples e corresponde à soma dos valores escalares RGBA de todas as células que compõem um sub-volume. Todavia, outras funções mais elaboradas também podem ser utilizadas e testadas.

A Figura 4.9 apresenta este esquema de *hash* implementado através de uma lista encadeada. Cada *voxel* do refinamento apresenta valores escalares para cada canal de intensidade de cor RGBA que são somados entre si e também aos valores RGBA de todos os outros *voxels* componentes do refinamento. Este valor obtido é dividido pelo escalar 4195, estipulado como um bom número para uma função *hash* utilizando o método da divisão, e o resto da divisão representa o índice na tabela *hash*. Em seguida, o sub-volume é armazenado na lista encadeada referente ao resultado da função *hash*.

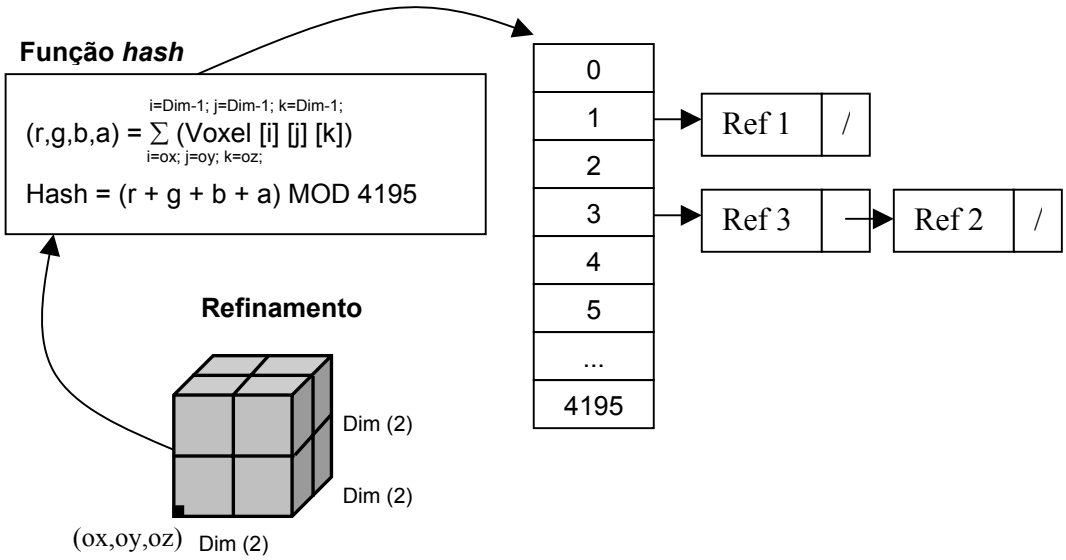


FIGURA 4.9 - Esquema *hash* utilizado

Nota-se, neste caso, que uma colisão não é definida apenas pelo valor de retorno da função *hash*, pois sub-volumes diferentes podem apresentar o mesmo valor de soma de suas células. A colisão é definida apenas para aqueles sub-volumes que realmente são idênticos. Assim, uma comparação volume a volume, envolvendo *voxel* a *voxel*, é realizada entre o sub-volume em questão e os sub-volumes já armazenados na lista encadeada referente ao valor da função *hash* obtido.

Completado o processo de compressão, é necessário realizar o caminho inverso, ou seja, a descompressão. A descompressão é utilizada para visualização dos volumes e está detalhada na seção a seguir.

4.2 Descompressão e Visualização

A visualização volumétrica baseada em texturas, neste trabalho, se baseia no uso de texturas mapeadas para uma seqüência de polígonos ou planos alinhados de acordo com a direção de visualização. Isto pode ser observado re-visitando a seção 2.2.2.4.1 do capítulo 2. Entretanto, o método de compressão *Gritex* requer uma etapa de descompressão, usando o *hardware* gráfico, com operações que vão além de um simples mapeamento de textura 3D, pois os dados amostrados estão armazenados em mais de uma textura 3D. O acesso as diferentes texturas, como já mencionado, será realizado utilizando a característica de texturas dependentes em *hardware*.

Sendo assim, a visualização de um determinado tempo *t* é realizada mapeando a "porção" referente a este tempo na textura 3D que contém os dados obtidos na primeira

etapa do processo de compressão, mas com um adicional: um conjunto de instruções será informado à unidade *texture shader* do *hardware* para verificar se os valores de cor e opacidade (RGBA) a serem visualizados são os próprios armazenados nesta textura de índices ou se estes valores servirão de índice a valores armazenados na(s) textura(s) de refinamento, os quais deverão ser buscados.

A Figura 4.10 mostra a equação que resulta nas origens de um tempo *t* contido na textura 3D de índices, onde *refinamentoDim* é a dimensão do volume que representa o refinamento (no caso do nível de parada ser 4, *refinamentoDim* é igual a 8), *tempoDim* é a dimensão do volume que representa cada tempo (para nível de parada 4, *tempoDim* é igual a 16), MOD retorna o resto da divisão de dois valores inteiros e DIV retorna o quociente da divisão. Nota-se uma ressalva para não confundir a origem (*ox*,*oy*,*oz*) de um refinamento com esta origem (*Ox*,*Oy*,*Oz*) de um tempo na textura de índice.

Ox = (*t* MOD refinamentoDim) * tempoDim
Oy = (*t* DIV refinamentoDim) * tempoDim
Oz = (*t* DIV refinamentoDim^2) * tempoDim

FIGURA 4.10 - Cálculo para a recuperação da origem de um tempo na textura de índices

Em seguida, a Figura 4.11 ilustra o papel da unidade *texture shader* na visualização dos dados comprimidos pela técnica *Gritex*. A unidade de rasterização gera todos os fragmentos pertencentes à área delimitada pelos vértices estabelecidos dos polígonos ou planos e os envia (fragmentos ou *pixels*), um por um, à unidade de *fragment shader*. As texturas de índice e refinamento também são informadas ao *fragment shader*.

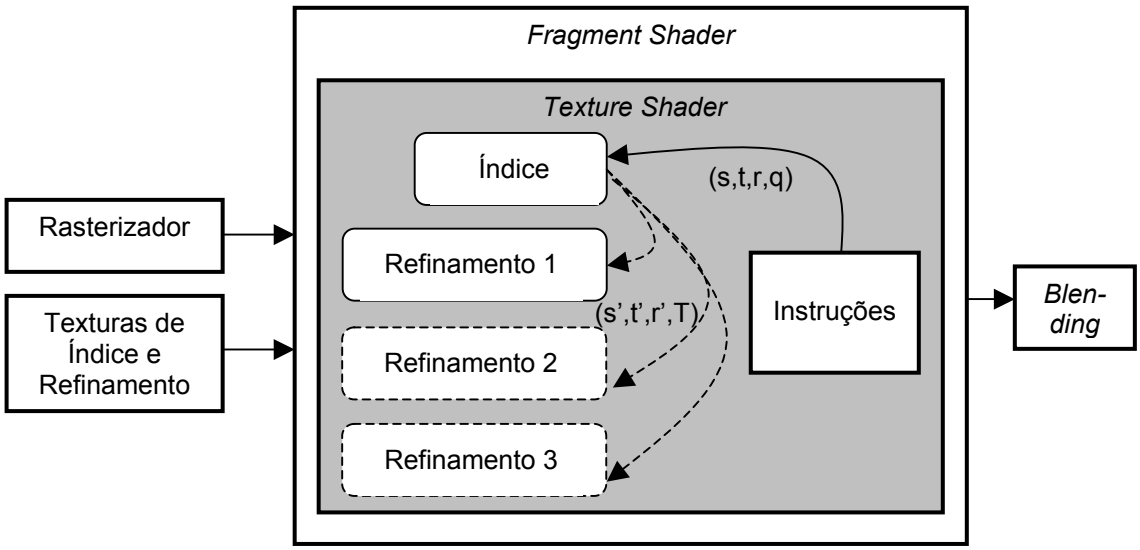


FIGURA 4.11 - Esquema báscio do *texture shader* para o método de compressão

O conjunto de instruções definidos na unidade de *texture shader* executará um acesso à textura de índices, através das coordenadas (*s*,*t*,*r*,*q*) dos polígonos, e analisará

os valores RGBA obtidos. Caso estes valores signifiquem uma intensidade de cor e opacidade (este teste é detalhado na próxima seção), são exibidos (*Blending*). Caso contrário, estes valores assumirão o papel de novas coordenadas (*s'*,*t'*,*r'*, *T*) para acesso (dependente) ao refinamento, onde *s'*, *t'* e *r'* definem a origem do sub-volume que contém o fragmento em questão dentro da textura de refinamento e *T* caracteriza essa textura. É importante ressaltar que estes valores recuperados pelo acesso à primeira textura podem não ser utilizados diretamente para um acesso dependente (como é o caso deste método), mas também manipulados para a geração de um novo índice.

A partir desta origem e da dimensão do sub-volume, todos os *voxels* contidos neste sub-volume são mapeados às suas referentes coordenadas de textura, obtendo-se, assim, a cor e opacidade de cada fragmento ou *voxel* do sub-volume. Para tal, um deslocamento a partir desta origem é calculado de acordo com a Figura 4.12. Na figura, o cálculo do deslocamento utiliza as coordenadas de textura X, Y e Z referentes a cada fragmento para realizar uma operação de MOD (função que retorna o resto de uma divisão de inteiros) com as dimensões dos sub-volumes. A cor final do *pixel* é, então, calculada através de um acesso à textura de refinamento utilizando, como novas coordenadas, o deslocamento somado à sua origem. Em seguida, esta cor é enviada para o processo de composição de cores para posterior visualização (*blending*).

dx = Coordenada X da textura MOD dimensão X do sub-volume

dy = Coordenada Y da textura MOD dimensão Y do sub-volume

dz = Coordenada Z da textura MOD dimensão Z do sub-volume

Cor = Refinamento [**ox** + **dx**] [**oy** + **dy**] [**oz** + **dz**]

FIGURA 4.12 - Cálculo para recuperação dos refinamentos

Resumindo, a Figura 4.13 apresenta o processo geral de visualização do método *Gritex* para um instante de tempo t_n. Nota-se que as coordenadas de textura (*s,t,r*) são divididas pela dimensão do sub-volume (8, no caso) no acesso à textura de índices, pois cada unidade dessa textura representa um sub-volume de 8^3.

A próxima seção apresenta os detalhes de implementação da descompressão. Esta descompressão baseia-se no conjunto de instruções, apresentado na Figura 4.11, implementado na unidade *texture shader* e executado pelo *hardware* gráfico.

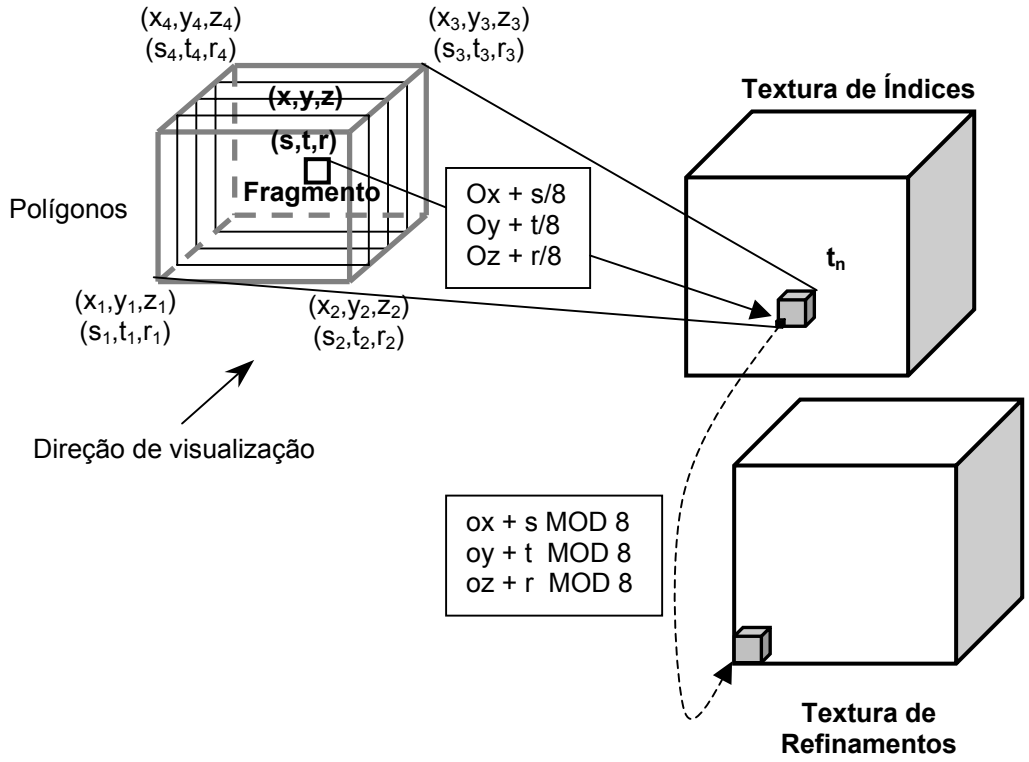


FIGURA 4.13 - Processo geral de visualização para um instante de tempo t_n

4.2.1 Detalhes de implementação

As instruções foram codificadas na linguagem Cg, abordada no capítulo 3, e compiladas para gerar código geral *assembler* de forma a ser aproveitado integralmente por um *hardware* de outro fabricante, *Radeon 9700* da empresa *ATI*. É importante ressaltar que não foi utilizada outra placa gráfica com características semelhantes ou com maior poder de armazenamento e processamento, como o *hardware GeForce FX* da empresa *NVIDIA*, pois este ainda não se encontra disponível no mercado.

Uma vez armazenadas todas as unidades de textura (índice e refinamento), um mapeamento simples, referente a um instante de tempo em questão, da textura 3D de índices é realizado. Nesta etapa de acesso à textura, entra em ação a unidade *fragment shader* e, por sua vez, sua sub-unidade *texture shader*. Esta unidade executa o conjunto de instruções, previamente compilado, para atribuição de cor e opacidade a cada *pixel*. A Figura 4.14 ilustra o conjunto de instruções adotadas para descompressão e visualização com apenas uma textura 3D representando os refinamentos. Esta simplificação se deve ao fato do *hardware ATI Radeon 9700* trabalhar com no máximo 64 instruções em *assembler* para um *fragment shader* e o código em Cg da Figura 4.14 resulta, em sua compilação, 60 instruções, ou seja, quase o número máximo possível de instruções.

As etapas para a visualização/descompressão presentes no programa da Figura 4.14 estão numeradas a seguir:

1. os valores *Ox*, *Oy* e *Oz* são pré-calculados através da equação da Figura 4.10, os quais determinam o instante de tempo a ser acessado na textura de índices;
2. os seguintes parâmetros de entrada ao programa são, então, informados: coordenadas de textura (*IN*), origem do sub-volume (*Ox, Oy* e *Oz*) que representa um tempo na textura de índices, dimensão do sub-volume de refinamento (*nodeSide*) estabelecido na Figura 4.12, textura 3D com os índices armazenados (*texIndex*) e textura 3D contendo os refinamentos (*texRefinement*);
3. calcula-se as novas coordenadas de textura (*index*) referente ao instante de tempo desejado para acesso na textura de índices. Estas novas coordenadas de textura são obtidas através da divisão das coordenadas de texturas passadas como parâmetro (*IN*) pela dimensão do refinamento (*nodeSide*) e seu resultado somado a origem (*Ox*, *Oy*, *Oz*). Assim, é realizado o mapeamento correto da "porção" de textura que representa o tempo em questão;
4. acesso à textura de índices com as coordenadas armazenadas em *index*;
5. um teste é realizado com a informação retornada do acesso a textura de índices para verificar se a mesma representa uma cor, relativa a um sub-volume homogêneo, ou uma origem na textura de refinamento. O teste é realizado levando em consideração a componente alfa do valor retornado (*r,g,b,a*), já que a implementação da compressão foi realizada adotando valor igual a 1 para um sub-volume que necessite de refinamento e valores menores que 1 para sub-volumes homogêneos. Este teste é realizado através de uma instrução *if-else*. Deve-se observar que, no código, os valores RGBA recuperados da textura estão sendo representados por (*x,y,z,w*);
6. caso *a* seja igual a 1, um acesso dependente é realizado na textura de refinamento, mas com novas coordenadas de textura calculadas em *refIndex*. Estas novas coordenadas (*ox+dx*, *oy+dy*, *oz+dz*) são calculadas seguindo a fórmula apresentada na Figura 4.12, com algumas modificações:
    a. as origens são multiplicadas por 2, pois a estrutura RGBA utilizada assume valores entre 0 e 255, mas quando se representa uma origem nesta estrutura, armazena-se valores referentes à dimensão da textura 3D usada, ou seja, entre 0 e 127. O *hardware* gráfico faz um mapeamento dos valores de RGBA para valores entre 0 e 1 e, assim, o valor 127 seria mapeado para 0,5 e não para 1 (dimensão máxima da textura) como se deseja;
    b. a segunda modificação também leva em conta este mapeamento para valores entre 0 e 1. Como a função MOD retorna o resto da divisão entre dois valores inteiros e os valores de coordenadas de textura também são mapeados para valores entre 0 e 1, estes valores de coordenadas de textura devem ser transformados para inteiros. A solução é multiplicar estes valores pela dimensão da textura (128^3), realizar o cálculo da função MOD e, finalmente, dividir o resultado pela dimensão da textura. Deve-se observar que o mapeamento das

coordenadas de textura não é realizado com valores entre 0 a 255 e sim com valores entre 0 a 127, para valores entre 0 a 1;

7. a cor final do *pixel* é, finalmente, definida como sendo o resultado do acesso à textura de índices, caso *a* (alfa) seja diferente de 1 ou como sendo o resultado do acesso à textura de refinamento, caso *a* seja igual a 1.

```
struct vert2frag
{
	float4 texCoord : TEX0;
};

struct frag2frame
{
	float4 color : COLOR0;
};

frag2frame
main( vert2frag IN,		// Coordenada de textura
	uniform float Ox,	// Origens na textura de índices
	uniform float Oy,	//
	uniform float Oz,	//
	uniform float nodeSide, // Dimensão do refinamento
	uniform sampler3D texIndex : texUnit0,	// Textura de índices
	uniform sampler3D texRefinement : texUnit1 // Textura de refinamentos
	) : COLOR
{
	// Cor de saída do pixel
	frag2frame OUT;

	//> Origem de um tempo t na textura de índices + deslocamento
	float3 index;
	index.x = Ox + IN.texCoord.x / nodeSide;
	index.y = Oy + IN.texCoord.y / nodeSide;
	index.z = Oz + IN.texCoord.z / nodeSide;
	//< Origem de um tempo t na textura de índices + deslocamento

	// Acesso a textura de índices. Origin = (ox,oy,oz)
	float4 origin = f4tex3D(texIndex, index);

	//> Origem de um refinamento na textura de refinamentos + deslocamento
	float3 refIndex;
	refIndex.x = origin.x*2 + fmod(IN.texCoord.x*128, nodeSide)/128;
	refIndex.y = origin.y*2 + fmod(IN.texCoord.y*128, nodeSide)/128;
	refIndex.z = origin.z*2 + fmod(IN.texCoord.z*128, nodeSide)/128;
	//< Origem de um refinamento na textura de refinamentos + deslocamento

	// Teste para saber se há refinamentos
	if (origin.w == 1.0) {
		// Acesso a textura de refinamento
		float4 ref = f4tex3D(texRefinement, refIndex);
		// Cor final do pixel é a cor recuperada de um refinamento
		OUT.color = ref;
	}
	else {
		// Cor final do pixel é a cor recuperada no índice
		OUT.color = origin;
	}

	// Retorno da cor final do pixel
	return OUT;
}
```

FIGURA 4.14 - Código em Cg para visualização (descompressão)

5 Resultados

Este capítulo apresenta os resultados da aplicação do método proposto a um estudo de caso, o Projeto MAPEM. O estudo de caso corresponde à visualização dos dados provenientes da simulação do descarregamento de material produzido na perfuração prospectiva de um poço de petróleo.

5.1 Geração dos Dados Volumétricos Dinâmicos

O Modelo OOC foi desenvolvido pela empresa *Exxon Mobil Production Research* (Brandsma e Smith, 1999) para simular o comportamento do descarregamento no mar de materiais (cascalhos impregnados de fluidos de perfuração, entre outros) gerados pela perfuração de poços de petróleo (Figura 5.1). O objetivo final da simulação é prever a acumulação destes cascalhos no fundo do mar. Um estudo desse modelo foi realizado (Binotto, 2001) e serão abordados aqui apenas os aspectos necessários para o entendimento geral do processo.

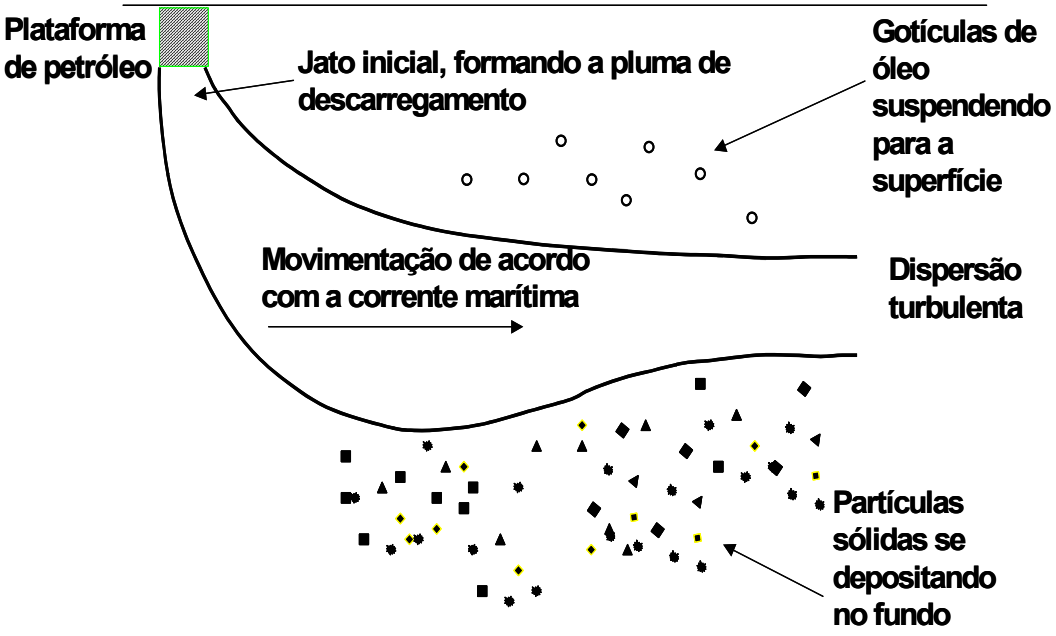


FIGURA 5.1 - Fases do descarregamento (Brandsma e Smith, 1999)

Os dados de entrada para o simulador descrevem:

- **Grade** – região a ser estudada na simulação (Figura 5.2). Esta área é decomposta de células, formando uma malha; à cada célula serão associados valores de concentração de material descarregado.
- **Ambiente** – variáveis do ambiente, entre as quais, os perfis de correntes marítimas em diversas profundidades.
- **Descarregamento** – parâmetros referentes à taxa de descarga, dimensão do tubo de descarga, orientação do tubo, entre outros.

- **Saída** – descrição da organização esperada do resultado da simulação, sendo os mais importantes itens: a definição de profundidades ao longo da coluna de água em que se deseja obter uma grade com a concentração do material e definição de instantes de tempo em que se deseja obter tais informações.

O modelo produz, basicamente, resultados numéricos para análise, sem nenhuma saída gráfica. Estes resultados representam concentrações do material descarregado por célula da grade. Pode-se gerar apenas um único plano representando o acúmulo total no fundo do mar de cada classe de partículas sólidas individuais ou do conjunto de todos os constituintes sólidos, como também uma série de planos ou camadas ao longo da coluna de água, representando fatias de um volume de dados (Figura 5.2).

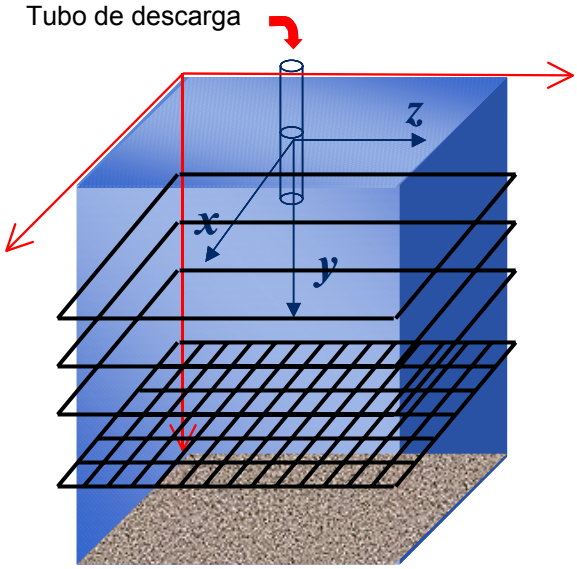


FIGURA 5.2 - Estrutura da grade e seqüência de planos de dados gerados pelo simulador

Após a obtenção dos dados de simulação, estes passam por uma etapa de pós-processamento para a obtenção de valores de espessura da camada (virtual) de cascalho que seria acumulada no fundo do mar. Assim, o resultado desse pós-processamento pode ser tanto as grades originais de concentração como grades contendo “espessuras” em diferentes profundidades, para o cômputo da acumulação de fundo.

5.2 Visualização dos Dados Volumétricos

Obtidos os dados de espessura, a visualização dos dados resultantes com o Modelo OOC foi desenvolvida em duas etapas.

A primeira etapa é um processo de mapeamento de cores. Para isso, tais dados serão normalizados em valores escalares de cor no sistema RGBA, variando de 0 a 255. O resultado desta etapa consistirá em uma textura a ser visualizada na etapa seguinte. Por exemplo, para uma grade que representa a acumulação do fundo do mar, uma textura 2D será criada. Entretanto, reunindo as várias grades que representam camadas dos dados pós-processados, uma textura 3D pode ser criada. Como o simulador gera,

para cada instante de tempo, um conjunto de camadas como as da Figura 5.2, várias texturas 3D podem ser criadas.

Passando pela etapa de mapeamento de cores, os dados já podem ser visualizados como texturas. Para o caso da visualização de um único instante de tempo, ou apenas da acumulação no fundo, um programa interativo para visualização 3D simples de texturas 2D e 3D, baseado numa câmera OpenGL, é utilizado. Para a visualização de uma série temporal de dados volumétricos, é utilizada a técnica de compressão/visualização desenvolvida neste trabalho. A utilização de ambos permitirá uma avaliação do método proposto através da visualização de instantes de tempo específicos.

5.2.1 Visualização Usando Texturas 2D e 3D Simples

Na Figura 5.3, é apresentada a visualização da acumulação total de materiais decorrente da simulação de uma perfuração[2]. Foram gerados os dados volumétricos de diferentes camadas e apenas a espessura acumulada na camada de fundo é exibida.

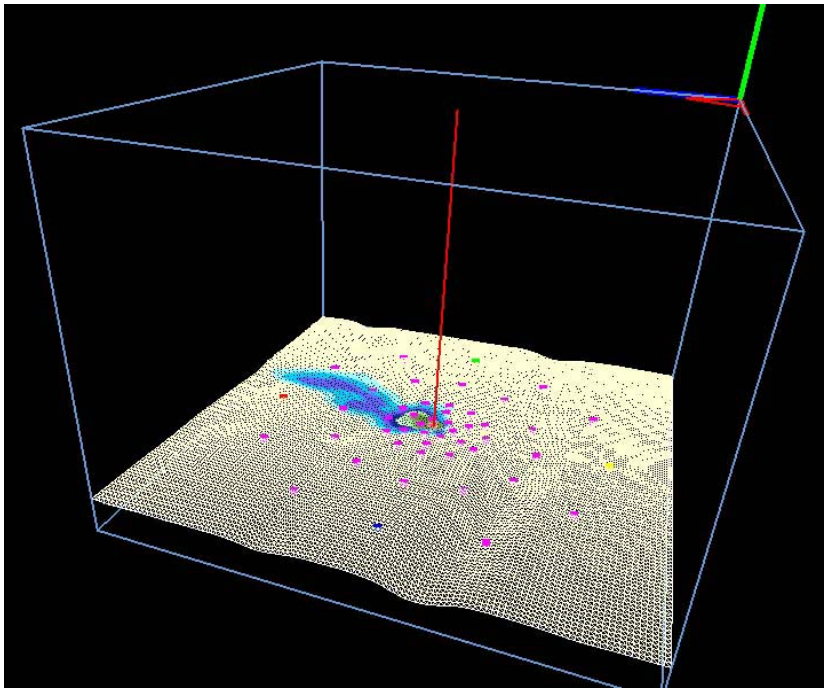


FIGURA 5.3 - Visualização acumulação total, representada por uma textura 2D

Na Figura 5.3, o cubo (volume de amostragem) representa a coluna de água do mar correspondente à grade modelada e a linha vertical, no centro do cubo, o local da perfuração. A batimetria do fundo do mar é exibida em aramado e a textura 2D, nela mapeada, representa a espessura de acumulação do material; os pontos demarcados correspondem às estações de coleta de material *in* loco pelos diferentes grupos do Projeto MAPEM.

A utilização dos dados volumétricos diretamente como texturas 3D permite visualizar o comportamento do fluxo de descarga do material ao longo do tempo, desde a superfície até o fundo do oceano. Para isso, é necessário utilizar um *hardware* gráfico

[2] Perfuração do poço *Eagle*, localizado na Bacia de Campos, de 2 a 12 de Junho de 2002.

como GeForce3 ou superior. Além disso, é necessário trocar a textura 3D, para cada instante de tempo a ser visualizado, e processar a visualização com a técnica de mapeamento de texturas 3D (seção 2.2.2.4.2 do capítulo 2). A Figura 5.4 ilustra a visualização de quatro tempos (dois iniciais e dois intermediários) da mesma simulação exibida na Figura 5.3. Cada tempo, ou seja, cada volume de dados correspondente a um tempo, é representado numa textura 3D.

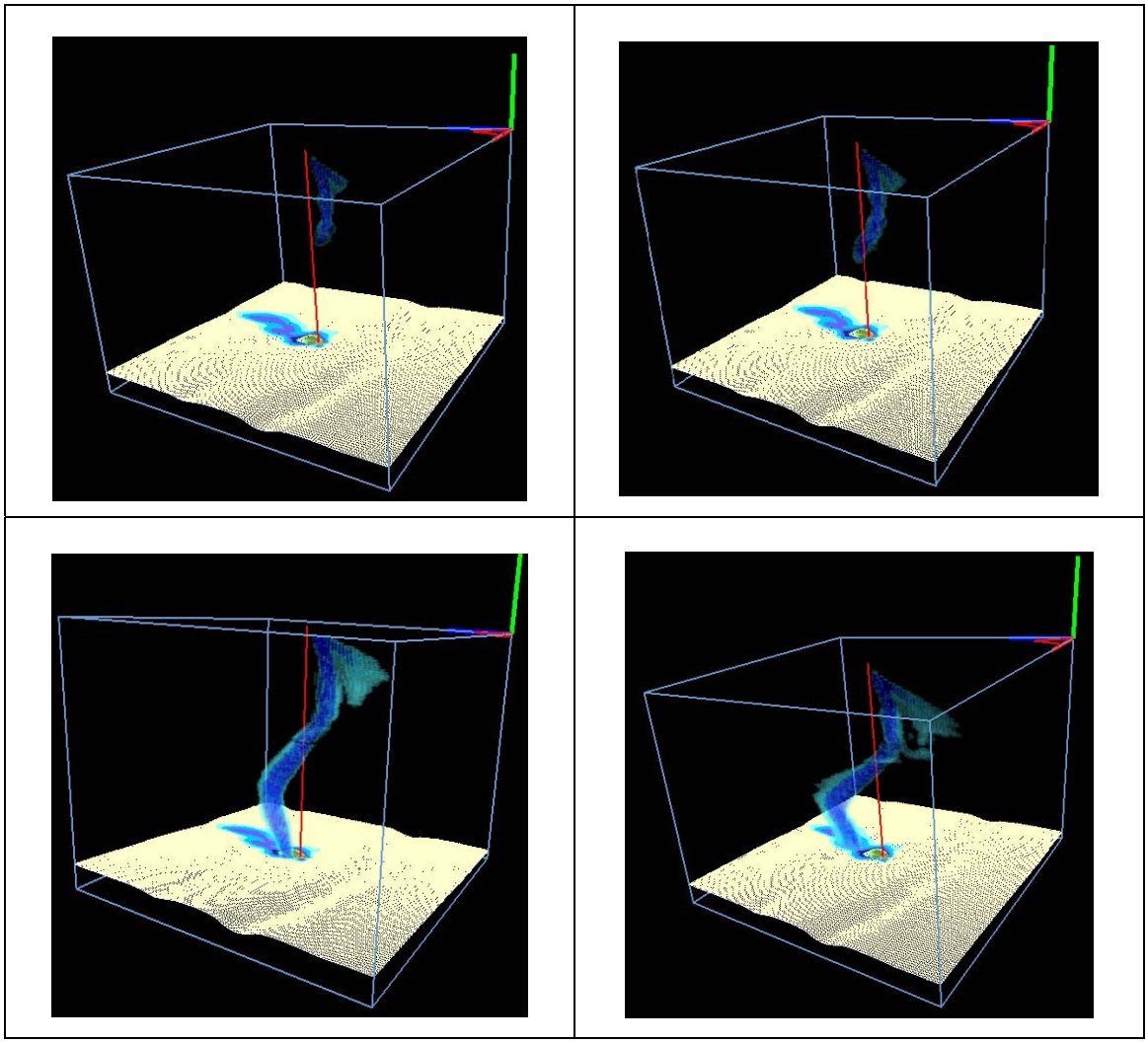


FIGURA 5.4 - Visualização individualizada de quatro instantes de tempo, com texturas 3D

Mesmo com o uso do *harwdare* gráfico, esta visualização torna-se lenta quando se deseja exibir seqüencialmente diferentes volumes (de 2 a 3 segundos para atualizar a imagem, usando um computador com a configuração descrita na próxima seção). Desta forma, a solução para visualizar o descarregamento em tempo real é aplicar ao conjunto de dados volumétricos a técnica de compressão e visualização de dados *Gritex* e utilizar as texturas 3D resultantes da técnica e as propriedades (dependência de texturas) em *hardware* para visualização dos vários volumes.

5.2.2 Visualização Usando a Técnica de Compressão *Gritex*

A técnica *Gritex*, proposta no capítulo anterior, obteve um ótimo desempenho de compressão com os volumes resultantes do estudo de caso. Conseguiu-se realizar em tempo real a visualização dos 36 instantes de tempo que representam uma das simulações da perfuração do poço *Eagle*.

O nível quatro foi escolhido como ponto de parada da primeira etapa de compressão, podendo comportar ainda mais 476 (512-36) instantes de tempo. O método de compressão resultou em 143.894 sub-volumes ou nodos homogêneos de dimensão 8^3 e 3.562 sub-volumes de 8^3, que necessitaram de refinamento (Tabela 5.1). A função *hash* identificou 126 colisões, ou seja, 3.436 (3.562-126) sub-volumes de 8^3 foram armazenados na textura de refinamento, restando ainda 660 (4.096-3.436) posições livres.

O computador utilizado para o pré-processamento de compressão foi um PC Pentium IV com 1.6 GHz e 256 MB de memória RAM. O tempo de compressão destes 36 instantes de tempo foi em torno de 6 minutos, ou seja, média de 10 segundos para cada volume (instante de tempo) de 128^3.

A Figura 5.5 ilustra a comparação da visualização de três instantes de tempo (instantes 5, 21 e 35) usando o método de mapeamento de texturas 3D simples (visto na seção 2.2.2.4.2) à esquerda e o método de compressão e descompressão (visualização) *Gritex* à direita. Ressalta-se que na Figura 5.5 são visualizados apenas os dados do descarregamento, sem dados de acumulação no fundo do mar. Pode-se observar a similaridade entre as imagens a2, b2 e c2 e a1, b1 e c1, respectivamente.

Um aspecto interessante para ter idéia do volume armazenado é visualizar a textura 3D de índices e de refinamentos, resultantes da compressão *Gritex*. A Figura 5.6(a) e 5.6(b) exibem, respectivamente, a textura de índices e de refinamentos. A visualização da textura de índices é um pouco curiosa, pois os índices que referenciam os refinamentos também são mapeados para cores. Como a implementação do processo de armazenamento está no formato (z,y,x) e não (x,y,z), os valores mapeados para cores estarão no formato (b,g,r) e não (r,g,b). Isso explica a cor azul predominante inicialmente, seguindo o tom de verde e, finalmente, o vermelho. As cores na textura de refinamento são as próprias cores dos sub-volumes refinados.

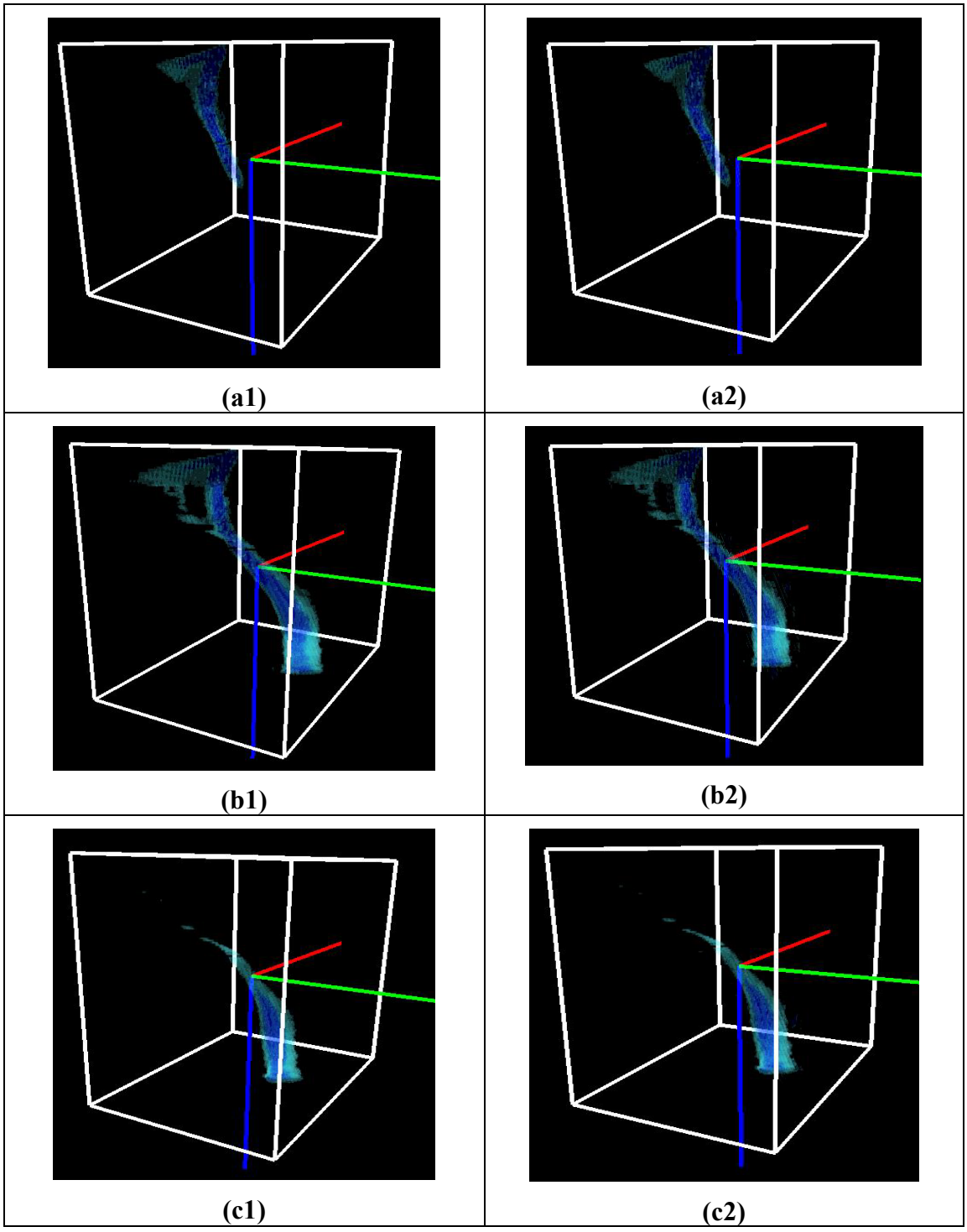


FIGURA 5.5 - Comparação entre visualizações usando texturas 3D simples e usando a técnica *Gritex*: são ilustrados os instantes de tempo 5 , 21 e 35, com as imagens à esquerda sendo geradas com texturas 3D simples e as da direita usando o método *Gritex*

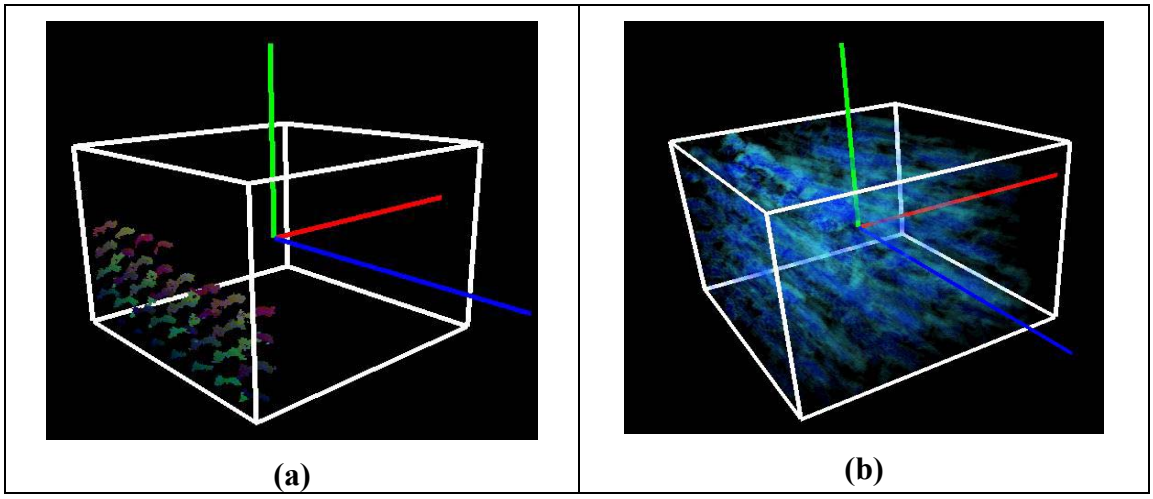


FIGURA 5.6 - Texturas 3D de índice e refinamento produzidas por *Gritex* com nível 4 de parada

Um outro nível de parada da primeira etapa do método de compressão *Gritex*, nível 5, foi utilizado com o mesmo conjunto de dados. Neste nível, os sub-volumes resultantes obtém dimensão de 4^3 e ocupam um espaço na textura de índices correspondente a 32^3 $((128/4)^3)$. A textura de índices poderá, então, conter até 64 $((128/32)^3)$ instantes de tempo.

A compressão dos 36 instantes de tempo ocorreu, também, em torno de 6 minutos, e resultou em 1.165.507 sub-volumes ou nodos homogêneos de dimensão 4^3, e 14.141 sub-volumes de 4^3, que necessitaram de refinamento (Tabela 5.1). A função *hash* identificou 1.385 colisões, ou seja, 12.756 (14.141-1.385) sub-volumes de 4^3 foram armazenados na textura de refinamento, restando ainda 20.012 (32.768-12.756) posições livres.

A descompressão produziu os mesmos resultados da Figura 5.5 mas, obviamente, as texturas 3D de índices e refinamentos (Figura 5.7) são diferentes das da Figura 5.6.

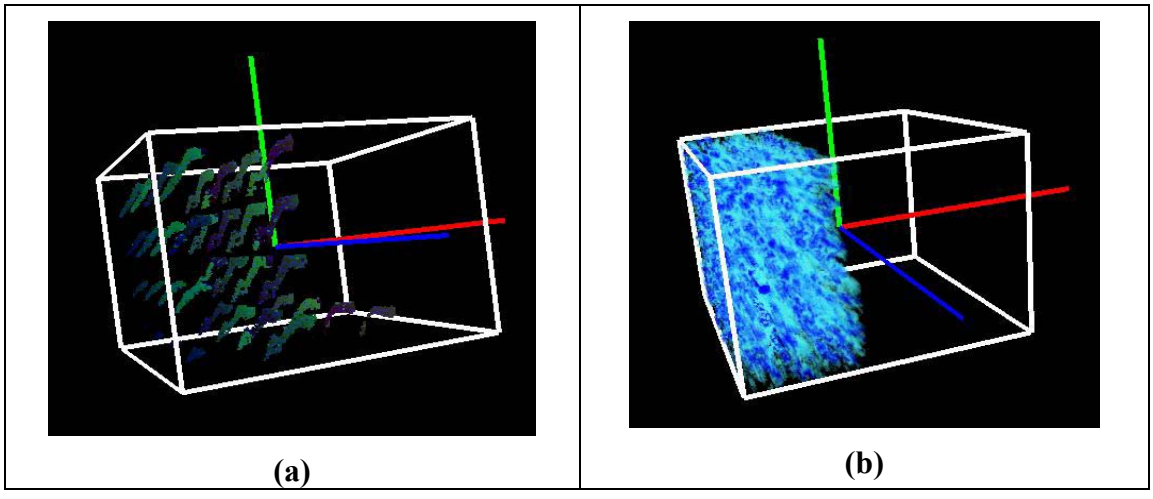


FIGURA 5.7 - Texturas 3D de índice e refinamento produzidas por *Gritex* com nível 5 de parada

Resumidamente, a Tabela 5.1 mostra a comparação dos resultados da técnica de compressão para três níveis de parada (relacionados a este estudo de caso), apresentando a capacidade máxima de instâncias de tempo e refinamentos armazenados, o número resultante de sub-volumes homogêneos, heterogêneos e idênticos (duplicados). Também é apresentado o número de texturas 3D (128^3) utilizadas para armazenar as 36 instâncias de tempo e seus respectivos refinamentos.

TABELA 5.1 - Comparação dos resultados do método de compressão

Nível	Máx. Instâncias de tempo (Índices)	Máx. Refina-mentos	Sub-volumes homogê-neos	Sub-volumes heterogê-neos	Sub-volumes duplicados	Índices utilizados	Refina-mentos utilizados
3	4.096	512	17.355	1.077	0	0,009	2,104
4	512	4.096	143.894	3.562	126	0,070	0,839
5	64	32.768	1.165.507	14.141	1.385	0,562	0,390

Outra análise de grande importância é apresentada na Tabela 5.2 para comparação do desempenho da visualização. Esta tabela mostra a média da variação da taxa de *frames* por segundo (FPS) em relação à janela de visualização e ao número de polígonos desenhados para obter a imagem do volume.

TABELA 5.2 - Análise de desempenho de visualização em quadros por segundo

Polígonos	Dimensão da janela de visualização	FPS
200	128^2	33,76
200	256^2	8,65
200	512^2	4,23
500	128^2	14,66
500	256^2	4,00
500	512^2	1,30

6 Conclusão

Este trabalho apresentou um novo método para visualização direta de volumes dinâmicos (representados por uma função 4D) em tempo real usando texturas 3D e características de *hardware* gráfico. O método *Gritex* seguiu uma abordagem em duas etapas, sendo a primeira para compressão (em pré-processamento) de vários volumes de dados, representando instâncias de tempo, para *grids* armazenados em texturas 3D, e a segunda, para descompressão e visualização dos dados usando o conceito de texturas dependentes presente no *hardware* gráfico.

O método apresentou, em alguns casos, uma relação de quadros por segundo maior que 30, o que torna a visualização não apenas interativa e sim em tempo real. Vale ressaltar que esta relação também leva em conta o número de planos utilizados para mapear a textura 3D e a dimensão da janela de visualização.

O grande custo computacional do método se concentra na etapa de compressão dos volumes de dados. Mas este custo não chega a ser considerado desvantagem, já que é parte de um pré-processamento dos dados e não do processo de visualização propriamente dito.

Analisando o método *Gritex* em relação aos demais trabalhos descritos na literatura, apenas o trabalho de Kraus e Ertl (2002) se assemelha a este método. Além de apresentar uma abordagem 2D e 3D, Kraus e Ertl (2002) também apresentaram, como visto no capítulo 2, uma abordagem de compressão de dados 4D referentes a uma função de campos de iluminação (Levoy e Hanrahan, 1996), utilizando um esquema em dois níveis de índices, armazenados em texturas 3D, e baseado no conceito de dependência de texturas em *hardware*. O método também não apresenta uma abordagem hierárquica flexível para a determinação de sub-volumes homogêneos e heterogêneos e não há detalhes sobre um possível re-uso de dados de sub-volumes homogêneos, como desenvolvido em *Gritex*.

O método *Gritex* é genérico para a utilização de quantas texturas forem necessárias para o armazenamento dos dados. A limitação do número de instruções para a unidade *texture shader* (64 instruções com o *hardware* utilizado) foi o único fator determinante para o uso de apenas duas texturas na atual implementação. O relaxamento dessa limitação propicia a utilização de mais de duas texturas para o armazenamento dos dados volumétricos dinâmicos. Isto quer dizer que mais de uma textura 3D poderá armazenar tanto os índices da primeira etapa quanto seus refinamentos.

Possíveis trabalhos diretamente derivados deste podem surgir aplicando o método com outros volumes de diferentes simulações ou oriundos de outros domínios, para uma análise do impacto de diferentes taxas de coerência espacial e temporal, assim como de diferentes níveis de esparsidade.

Anexo

(Artigo publicado no I Workshop de Teses e Dissertações do XV Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens e na Revista *Scientia* da Unisinos)

Real-Time Volume Rendering of Dynamic Data Using Graphics Hardware

ALÉCIO P. D. BINOTTO, JOÃO L. D. COMBA, CARLA M. D. S. FREITAS

II/UFRGS – Instituto de Informática da Universidade Federal do Rio Grande do Sul, Caixa Postal 15.064, 91501-970 Porto Alegre, RS, Brasil
{abinotto, comba, carla}@inf.ufrgs.br

Abstract. Many applications in computational fluid dynamics work with time-varying volumetric data (4D). Although high-dimensional, it is possible to compress these data when they are sparse or have coherence (spatial or temporal). In this work we are developing a new compression scheme that benefits from 3D textures of recent graphics hardware, and we antecede it will be possible to render it in real time. Tests will be performed using the data produced in MAPEM project.

1 Introduction

The visualization and representation of physical phenomena is very important to many scientific areas. For example, the MAPEM project (Environment Monitoring of Off-Shore Activities and Exploration) evaluates the impact in the marine ecosystem of the discharge of drilling cuts produced during oil perforation. Seabed samples are collected before and after drilling, and pass through a complete analysis by several groups (chemistry, statistics, geosciences, etc). The visualization of the discharge process is very useful to help understand the results obtained in this analysis, and this is the main goal of this work.

The representation of the discharge phenomenon can become very complex (see Brodlie et al. [5]). Solutions to this type of problems are studied in the Computational Fluid Dynamics (CFD) field, where it is common to decompose the domain in uniform cells (grids), with each cell containing an approximation of the function. This approach produces volumetric datasets (3D) for each instance of time, which represents a 4D dataset if we consider all time instances.

In the MAPEM project, approximations to the discharge process are obtained using a simulator developed by the Exxon Mobil Corporation called **OOC** (The Offshore Operators Comittee Mud and Produced Water Discharge Model, Brandsma and Smith [7]). The simulator allows the user to specify the parameters that describe the discharge process (location, pipe orientation, salinity, fluid concentration, currents, etc). The discharge process is approximated using a mathematical model, which produces an estimated dispersion of discharged fluids and materials during oil drilling. These data is represented in a uniform 3D grid or 2D planar sections, and can be sampled at any instance of time during the simulation. In order to interpret the generated results, scientific visualization techniques are used.

Volume Rendering is the classic Computer Graphics technique to render volumetric data as described. Traditionally, it is computationally very expensive, requires faster computers and even special hardware, which limited the widespread use of the technique. Recently, the advance of graphics hardware is changing this scenario. Several new features, such as support to multi-texturing and 3D textures, can be used by volumetric rendering algorithms to speed-up computations and make it real-time.

In this paper we present a novel approach to perform volume rendering using recent graphics hardware. The paper is organized as follows. First, we review past and related work. Next, we present the new features of graphics hardware, specially

the 3D and dependent textures, which are important to understand the technique proposed in Section 4. The description of the visualization model used for the proposed technique is presented in Section 5. Section 6 presents the current stage of the work, followed by conclusions and future directions.

2 Related Work

Four-dimensional data composed of several instances of volumetric data have been widely used in flow visualization. In Shen et al. [2], an efficient structure called the TSP-tree (Time-Space Partitioning) was created to visualize temporal series of volumetric data. Ellsworth et al. [1] followed extended this work with a way to represent TSP trees in hardware, using 3D textures of a SGI Infinite Reality 2.

Engel et al. [4] proposed an approach that explores the 3D textures of the GeForce3 graphics board (NVIDIA Corporation). Volume rendering using 3D textures is done by rendering a stack of 3D textured slices (polygons) orthogonal to the viewing direction position. Each slice is processed according to its distance to the viewer, starting with the nearest slice. Each pixel of a slice corresponds to the contribution of one ray segment that is emitted by the viewer and the accumulation is computed considering the impact of the correspondent pixel in the previous slice processed. The access to each pixel value, as well as the combination of several pixel colors, is computed using the GeForce3 pixel shader, which we will review in the next section.

3 Review of the GeForce3 Pixel Shader

Nowadays, graphic cards have been incorporating many interesting features that were previously only implemented by software. Besides allowing innumerous complex effects, these features also make it possible to render scenes composed by millions of triangles at very high fill-rates.

One of the features that can produce incredible effects is the use of textures. In the GeForce3 card (64MB of memory), it is possible to process simultaneously for each fragment a maximum of four textures (1D, 2D or 3D). Textures can be seen as tables that are indexed by texture coordinates, producing a 32-bit integer and represented as *rgba* colors.

The first attempts to allow more than one texture access to each pixel (called multi-texturing) were implemented by individual accesses to each one of the textures. In the OpenGl 1.2 specification [10], some level of dependency between texture accesses was introduced by allowing the result of a texture access to be passed on to the following texture unit. In the GeForce3, a more elaborate dependency between texture units was proposed, allowing other ways to re-use the result of a texture unit (rgba). This approach is called *dependent textures*. The available operations to combine these results with the next input texture coordinates were still limited, but even so they are able to obtain innumerous effects. For example, Comba and Bastos [3] mentioned that a pixel color could be an input parameter for the following texture, by changing the depth texture coordinate value (z) of a fragment for the color value obtained.

Once all texture accesses were performed, the final task of the pixel shader is to produce the final color of the pixel. A single pixel can be associated with different colors, coming from either the interpolated colors generated during rasterization or colors obtained by accesses to texture units. In the GeForce3, a dedicated hardware called the Register Combiners allows the different colors of a pixel to be combined and produce its final color.

The next section discusses the proposed structure to visualize in real time the fluid dynamic data using these hardware features.

4 Data Storage using 3D Textures

Some CFD data are extremely sparse, such as in liquid and gaseous fluids simulation. In addition, dynamic data usually possess high degrees of coherence, either spatial or temporal. Using the above properties, we designed a compression mechanism that can be implemented using the graphics hardware, specially the properties of 3D textures. Our solution is initially target to the GeForce3, which has four 3D textures and a defined set of dependent operations, but can be applied to future boards with more general designs and more resources. Once this is done, our intention is to make it possible the real-time visualization of the CFD simulations.

We assume that the input data consists of different time instances, each represented by a sparse volumetric data. Due to the graphics card limitations, the dimension of each volume is at most 128^3, which is a reasonable value in volume

rendering. Each cell corresponds to an intensity scalar value varying from 0 to 255.

The approach consists in creating a **two-step indexed dependent** mechanism, like the dependent textures. In the first step, a regular decomposition of each volume in sub-volumes is performed, corresponding to an initial hierarchical subdivision of the data. In this step, our goal is to separate the regions that need to be detailed from the ones that are homogeneous. A hierarchical subdivision in few levels usually suffices for this. In the second step, we add refinements to each non-homogeneous sub-volumes obtained in the first step. All of these values are storage in different textures units (the first phase uses a 3D texture and the second another one).

In the first step, a hierarchically process decomposes the volumetric data in a structure similar to an *octree* (Mäntylä [6]), where the original volume is divided in eight sub-volumes (cubes). In octrees, when a cube contains a constant value it is called *black* (if the value is higher than zero) or *white* (if the value is zero). In contrast, if the value is not constant it is called *gray* and requires additional subdivisions. However, the adaptive nature of octrees creates irregular structures most of the time not suited to be stored directly in 3D textures. To force the indispensable regularity, we build a complete octree (with all nodes) until a certain depth, which must be less than the maximum depth (seven for a 128x128x128 volume). Note that a complete octree is an uniform partition of a volume, or in other words, a *grid*. Note that the octree nature of the structure remains in the fact that we only refine the sub-volumes that are not homogeneous.

Depending on the data properties (coherence and sparcity), it is possible to estimate the best breakpoint of the subdivision process, allowing it to be changed according to the data. In our current experiments, we are using four levels of subdivision, as illustrated in Figure 1.

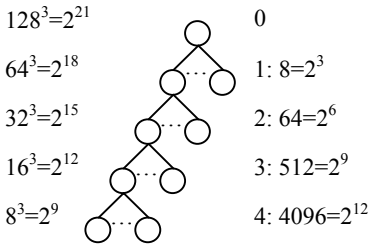


Figure 1 Subdivision until the fourth level. In the left we have the size of the sub-volume, and in the right the number of cubes in the corresponding level.

There are 4096 nodes at the fourth level of the grid corresponding to the first index of a given time instance. This information is stored in one of the 3D textures available, defined at the current tests due to the graphics memory limitations at 128^3, which has room to 512 (128^3/4096) of these fourth level grids.

Only gray nodes need to be refined, and will be stored in an additional index. We chose to use a two-level indexing scheme, therefore the entire sub-volume corresponding to a gray node needs to be represented elsewhere in other textures. Because we have three more 3D textures available, we could store in the first index the necessary information to locate the sub-volume in other textures, such as the texture identification (2, 3, or 4), and the origin of the sub-volume inside the texture (*ox*, *oy*, *oz*), and use dependent operations to retrieve this data. Due to limitations on the dependent texture operations available today (specific dot products), only one 3D texture can be used. This limitation is soon going to be removed as NVIDIA antecedes a fully programmable fragment shader with more powerful dependent operations [9].

The second texture will contain all refinements of gray nodes obtained from the grids stored in the first texture. In order to explore spatial and temporal coherence, we look to re-use refinement information as much as possible using a hashing scheme that speeds-up comparisons of each newly created refinement to the ones previously created. Collision cases allow us to re-use indexes, increasing the compression of the data. More elaborate hash functions can be proposed and we are using a very simple strategy that takes the remainder to the table size of the sums of the scalar values of each cell in the sub-volume, and so far the results are satisfactory.

Figure 2 shows the two-index model scheme proposed. Each volume, representing a time instance, is divided until a user-defined level (four in this case). For these parameters, the initial volume contains 16^3 volumes of size 8^3, for a total of 512 different time instances. All stored in the first 3D texture. Each gray sub-volume has size 8^3, refined and stored in the second texture.

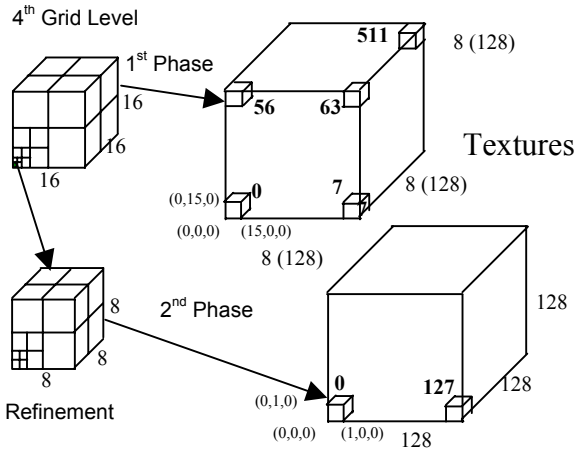


Figure 2 Two-indexed storage technique graphic scheme.

5 Data Visualization

The visualization of the dynamic data stored in the two-level index structure will follow the approach proposed by Engel et al. [4]. The visualization works by rendering a sequence of parallel slices orthogonal to the viewing direction, indexed to the 3D texture. Here, we will have a different way to retrieve information from the textures, using the method proposed in the previous section.

During rendering, each slice is processed according to its distance in relation to the viewer, from the nearest to the farthest (front-to-back). Pixel accumulation uses two adjacent slices (slab), which takes into account the impact of the correspondent pixel in the already processed front slice (sf). The accumulation is mapped onto one slice (either the front or the back slice) as depicted in Figure 3. Color and opacity will be linearly interpolated for each pixel by the GeForce3 rasterization engine.

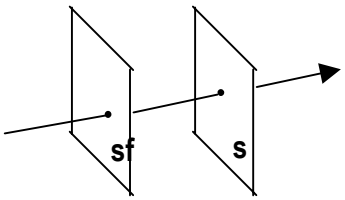


Figure 3 Pair of polygons (slab).

Temporal coherence can also be explored if the volume data do not change in a certain period of time. In this case, volume visualization will be executed just once, allowing images to be re-used for further frames in an animation sequence. Shen et al. [2] describes good results using this approach, decreasing volume rendering time and I/O time.

6 Current Stage

The implementation is under way. The construction of the two-index structure is already concluded for a general case. The volume rendering of a single instance of a 3D structure is done, remaining to integrate both parts using the dependent operations of the GeForce3 board. We are also experimenting with the new NVIDIA shading language CG (C for Graphics, [8]), which will be probably be the standard for graphics hardware programming.

Preliminary results obtained by the two-index implementation illustrate the potential of our approach to CFD data, specially the ones in the MAPEM project. For three time instances of dynamic data (128^3x3) produced by the OOC simulator using the two-index approach until with breakpoint at the fourth level, we reached a well distributed hash table (at most with two nodes in each linked list of the hash table) with a total of 1459 homogeneous nodes and 77 nodes that need to be refined.

The implementation reads 3D textures as a collection of 2D slices, corresponding grids of a given time instance. For the MAPEM project, we wrote a visualization tool that allows other types of visualization of the discharge process (visualization of bathymetry data and accumulations at the bottom of sea, pipe position and orientation, etc), and we will be integrating into this program the visualization using the two-index approach. The visualization of the discharge process as described by the simulation data will help the analysts of the MAPEM project to better understand the effects of the discharge in the marine ecosystem, as well as serve as a tool to debug the simulator itself.

7 Conclusions and Future Work

The main contribution of this research is to define a new approach for real-time volume rendering of dynamic data that are sparse and presents high levels of spatial and temporal coherence. The existence of 3D textures in hardware, together with powerful ways to combine values stored in textures, allowed to design a new way to compress and access volumetric data, in such way that it can be rendered in real time.

The new indexed model will be widely tested in many general CFD functions, including the

specific generated by the MAPEM project. Other volumetric functions or high-dimensional functions, such as light-fields may be considered.

Also, as graphics hardware is changing so much recently, we antecede that we will need to follow the new boards released by NVIDIA. We believe that the appearance of a new shading language as CG will make our task of programming the hardware a lot easier.

Acknowledgements

We would like to thank Cnpq for the support to the MAPEM project (Finep/CTPETRO), which has been motivating this work.

References

[1] D. Ellsworth, L. J. Chiang and H. W. Shen, "Accelerating Time-Varying Hardware Volume Rendering Using TSP Trees and Color-Based Error Metrics", In *Volume Visualization*, 2000.

[2] H. W. Shen, L. J. Chiang and L. K. Ma, "A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space (TSP) Tree", In *Volume Visualization*, 1999.

[3] J. L. D. Comba and R. Bastos, "Special Effects with Current Graphics Hardware". *Revista de Informática Teórica e Aplicada* 8 (2001), nº 2, 69--88.

[4] K. Engel, M. Kraus and T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading", In *Graphics Hardware Symposium*, 2001.

[5] K. W. Brodlie et al., "Scientific Visualization Techniques and Applications", Springer-Verlag, 1992.

[6] M. Mäntylä, "An Introduction to Solid Modeling", Computer Science Press, 1988.

[7] M. G. Brandsma and J. P. Smith, "Offshore Operators Committee Mud and Produced Water Discharge Model – Report and User Guide", ExxonMobil Upstream Research Company, 1999.

[8] NVIDIA Corporation, nVidia CG specifications, http://developer.nvidia.com/cg.

[9] NVIDIA Corporation, nVidia CineFx specifications, http://developer.nvidia.com.

[10] Silicon Graphics Inc., OpenGl 1.2 specifications, http://www.opengl.org/developers/documentation/OpenGL12.html.

[11] D. Benson and J. Davis, "Octree Textures", *Proceedings of SIGGRAPH 02*, 2002.

[12] D. DeBry, J. Gibbs, D. D. Petty and N. Robins, "Painting and Rendering Textures on Unparameterized Models", *Proceedings of SIGGRAPH 02*, 2002.

[13] I. Boada, I. Navazo and R. Scopigno, "Multiresolution Volume Visualization with a Texture-Based Octree", *The Visual Computer* 17 (2001), nº 3, 185—197.

Bibliografia

AKELEY, K. RealityEngine Graphics. **Computer Graphics**, New York, p. 109-116, Aug. 1993. Trabalho apresentado na SIGGRAPH Conference, 1993.

BENSON, D.; DAVIS, J. Octree Textures. **ACM Transactions on Graphics**, New York, v.21, n.3, July 2002. Trabalho apresentado na SIGGRAPH, 2002.

BINOTTO, A. P. D. **Visualização da Simulação de Escoamentos**. 2001. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BOADA, I.; NAVAZO, I.; SCOPIGNO, R. Multiresolution Volume Visualization with a Texture-Base Octree. **The Visual Computer**, New York, v.17, 2001.

BRANDSMA, M.; SMITH, J. **Offshore Operators Committee Mud and Produced Water Discharge Model**: report and user guide. Houston: ExxonMobil Upstream Research Company, 1999.

BRODLIE, K. et al. **Scientific Visualization**: techniques and applications. Berlim: Springer-Verlag, 1992.

BRODLIE, K. ; WOOD, J. Recent Advances in Volume Visualization. **Computer Graphics Forum**, Amsterdam, v.20, n.2, p. 125-148, 2001.

CABRAL, B.; CAM, N.; FORAN, J. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In: ACM/IEEE SYMPOSIUM ON VOLUME VISUALIZATION, 1994. **Proceedings…** New York: IEEE Press, 1994. p. 91-98.

CAMERON, G.; UNDRILL, P. Rendering Volumetric Medical Image Data on a SIMD-Architecture Computer. In: SIGGRAPH, 1994. **Proceedings…** Orlando: [s.n.], 1994. p. 451-458.

CHERNAYEV, E. **Marching Cubes 33**: construction of topologically correct and adaptive trilinear surfaces. Cidade: CERN, 1995. (Relato técnico CN/95-17). Disponível em: <http://wwwinfo.cern.ch/asdoc/psdir/mc.ps.gz>. Acesso em:18 dez. 2002.

CIGNONI, P. et al. Reconstruction of Topologically Correct and Adaptive Trilinear Surfaces. **Computer Graphics Forum**, Amsterdam, v.24, n.3, p. 399-418, 2000.

CLINE, H. et al. Two Algorithms for Three-Dimensional Reconstruction of Tomograms. **Medical Physics**, [S.l.], v.15, n.3, p. 320-327, 1988.

COMBA, J.L.D.; BINOTTO, A.P.D.; FREITAS, C.M.D.S. Visualização tridimensional interativa dos resultados da simulação da perfuração do poço Eagle. In: RELATÓRIO do Projeto MAPEM. Porto Alegre: Instituto de Geociências, 2003.

DEBRY, D. et al. Painting and Rendering Textures on Unparameterized Models. **ACM Transactions on Graphics**, New York, v.21, n.3, July 2002. Trabalho apresentado na SIGGRAPH, 2002.

DIETRICH, C.; COMBA, J. **Repulsive Potential Fields**. Disponível em: <http://www.cgshaders.org/contest/contest-results-03.php>. Acesso em: 19 fev. 2003.

DOI, A.; KOIDE, A. An Efficient Method of Triangulating Equi-valued Surfaces by Using Tetrahedral Cells. **IEICE Trans. Commun. Elec. Inf. Syst.**, [S.l.], v.74, n.1, p. 214-224, 1991.

ELLSWORTH, D.; CHIANG, L.; SHEN, H. Accelerating Time-Varying Hardware Volume Rendering Using TSP Trees and Color-Based Error Metrics. In: ACM/IEEE SYMPOSIUM ON VOLUME VISUALIZATION, 2000. **Proceedings…** New York: IEEE Press, 2000. p. 119-128.

ELVINS, T. T. A Survey of Algorithms for Volume Visualization. **Computer Graphics Forum**, [S.l.], v.26, n.3, p. 194-201, 1992.

ENGEL, K.; KRAUS, M.; ERTL, T. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In: EUROGRAPHICS/SIGGRAPH WORKSHOP ON GRAPHICS HARDWARE, 2001. **Proceedings…** Los Angeles: Addison-Wesley Publishing Company Inc, 2001. p. 9.

FOLEY, J. D. et al. **Computer Graphics**: Principles and Practice. Washington: Addison-Wesley, 1992.

FUCHS, H.; ZEDEM, Z. M.,; USELTON, S. P. Optimal Surface Reconstruction from Planar Contours. **Communications of the ACM**, New York, v. 20, n. 10, p. 693-702, 1977.

KAUFMAN, A. E. **Volume Visualization**. Los Alamitos, CA: IEEE Computer Society Press, 1991.

KEPPEL, E. Approximating Complex Surfaces by Triangulation of Contour Lines. **IBM Journal of Research and Development**, [S.l.], v. 19, n. 1, p. 2-11, 1975.

KRAUS, M.; ERTL, T. Adaptive Texture Maps. In: EUROGRAPHICS/SIGGRAPH WORKSHOP ON GRAPHICS HARDWARE, 2002. **Proceedings…** San Antonio: IEEE Press, 2002. p. 1-10.

LACROUTE, P. Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization. **IEEE Transactions on Visualization and Computer Graphics**, [S.l.], v. 2, n. 3, p. 218-231, 1996.

LACROUTE, P.; LEVOY, M. Fast Volume Rendering Using a Shear-Warp Transformation of the Viewing Transformation. In: SIGGRAPH, 1994. **Proceedings…** Orlando: [s.n.], 1994. p. 451-458.

LAMAR, E.; HAMANN, B.; JOY, K. Multiresolution techniques for Interactive Texture-Based Volume Visualization. In: IEEE CONFERENCE ON VISUALIZATION, 10., 1999, San Francisco, CA. **Visualization'99**: proceedings. New York: ACM, 1999. p. 355-361.

LEVOY, M. Volume Rendering – Display of Surfaces from Volume Data. **Computer Graphics and Applications**, Los Alamitos, v. 8, n. 3, p. 29-37, 1988.

LEVOY, M. Efficient Ray Tracing of Volume Data. **ACM Transaction on Graphics**, New York, v. 9, n. 3, p. 245-261, 1990.

LEVOY, M.; HANRAHAN, P. Light Field Rendering. In: SIGGRAPH, 1996. **Proceedings…** New York: ACM Press, 1996. p.31-42.

LORENSEN, W. E.; CLINE, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: SIGGRAPH, 1987. **Proceedings…** New York: ACM Press, 1987. p. 163-169.

MANSSOUR, I. H.; FREITAS, C. M. D. S. Visualização Volumétrica. **Revista de Informática Teórica e Aplicada**, Porto Alegre, v. 9, n. 2, p. 97-126, 2002.

MÄNTYLÄ, M. **An Introduction to Solid Modeling**. New York: Computer Science Press, 1988.

MEIβNER, M.; HOFFMANN, U.; STRAβER, W. Enabling Classification and Shading for 3D Texture Based Volume Rendering Using OpenGL and Extensions. In: IEEE CONFERENCE ON VISUALIZATION, 10., 1999, San Francisco, CA. **Visualization'99**: proceedings. Los Alamitos: IEEE Press, 1999. p. 207-214.

MEYERS, D.; SKINNER, S.; SLOAN, K. Surfaces from Contours. **ACM Transactions on Graphics**, New York, v. 11, n. 3, p. 228-258, 1992.

NVIDIA COORPORATION. **CG Toolkit**: a developer's guide to programmable graphics. Disponível em: <http://developer.nvidia.com/Cg>. Acesso em: 19 fev. 2003.

NVIDIA COORPORATION. Disponível em: <http://www.nvidia.com/developer>. Acesso em: 01 ago. 2001.

SHEN, H.; CHIANG, L.; MA, K. A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree. In: ACM/IEEE SYMPOSIUM ON VOLUME VISUALIZATION, 1999. **Proceedings…** New York: IEEE Press, 1999. p. 371-377.

UPSON, C.; KEELER, M. V-Buffer: Visible Volume Rendering. In: SIGGRAPH, 1988. **Proceedings…** New York: ACM Press, 1988. p. 59-64.

WESTERMANN, R.; ERTL, T. Efficiently Using Graphics Hardware in Volume Rendering Applications. In: SIGGRAPH, 1998. **Proceedings…** New York: ACM Press, 1998. p. 169-177.

WESTOVER, L. Interactive Volume Rendering. In: WORKSHOP ON VOLUME VISUALIZATION, 1989. **Proceedings…** North Carolina: University of North Carolina Press, 1989. p. 9-16.

WESTOVER, L. Footprint Evaluation for Volume Rendering. In: SIGGRAPH, 1990. **Proceedings…** New York: ACM Press, 1990. p. 367-376.

WYVILL, G.; MCPHEETERS, C.; WYVILL, B. Data Structure for Soft Objects. **The Visual Computer**, [S.l.], v. 2, p. 227-234, 1986.

YAGEL, R. et al. Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. In: ACM/IEEE SYMPOSIUM ON VOLUME VISUALIZATION, 1996. **Proceedings…** New York: IEEE Press, 1996. p. 55-62.