4
Modificações realizadas

Neste capítulo, apresentaremos as modificações realizadas e as escolhas que fizemos para as implementações dos algoritmos VICP (projeção de células) e Traçado de Raios, discutidos no capítulo anterior.

Na seção (4.2.2), é proposta uma forma de utilizar a integração de segmentos lineares, discutida na seção (2.3.2), no algoritmo de Traçado de Raios.

4.1.
Estruturas de dados na placa gráfica

Esta seção trata das estruturas de dados que utilizamos para a implementação dos algoritmos do Capítulo 3. Para o VICP, propomos uma estruturação para que os dados de uma malha de tetraedros sejam armazenados na placa gráfica. O objetivo é eliminar o gargalo na transferência de dados, reportado por Weiler et al. (2002). No caso do Traçado de Raios, que requer estruturas de dados na placa gráfica, utilizamos uma variação das que foram apresentadas na seção (3.2.1), visando obter um balanceamento entre desempenho e custo de armazenamento, com menor complexidade e maior facilidade para realizar modificações do que a estrutura de dados que utiliza faixas de tetraedros.

4.1.1.
Projeção de células (VICP)

Weiler et al. (2003b) reportam que o gargalo do VICP está na transferência de dados para a placa gráfica. Assim, como um trabalho futuro, sugerem que uma das formas de eliminá-lo seria armazenando os dados diretamente na placa gráfica, utilizando, por exemplo, texturas. Isso permitiria inclusive utilizar faixas de triângulos para reduzir o número de vértices enviados por tetraedro. Por outro lado, Moreland & Angel (2004) utilizam a abordagem de Weiler et al. (2002), restrita à projeção ortográfica, e realizam mais cálculos na CPU para reduzir o volume de dados enviados para a placa gráfica (seção 3.1).

Neste trabalho, optamos por explorar mais a programação por fragmentos, armazenando a malha de tetraedros na placa gráfica, conforme sugerem Weiler et al. (2003b). Dessa forma, acreditamos em uma evolução mais rápida dos processadores gráficos (GPU) em relação à CPU, como tem ocorrido recentemente (Lefohn et al., 2004).

A abordagem mais simples seria empregar um vetor contendo, para cada tetraedro, os 12 vértices que devem ser enviados para a placa gráfica (Weiler et al., 2003b). Para desenhar um tetraedro, basta acessar a posição relativa ao primeiro vértice e enviar seqüencialmente os 12 vértices do tetraedro. Um *vertex buffer object* (OpenGL ARB, 2005) pode ser usado para armazenar esse vetor na memória de vídeo, e apenas os índices de cada tetraedro precisam ser enviados para a placa gráfica. Entretanto, o custo de armazenamento dessa abordagem é muito alto.

Por isso, combinamos a utilização de *vertex buffer objects* com estruturas de dados armazenadas em texturas 2D (Tabela 4) na placa gráfica, que podem ser acessadas pelos programas por vértice e por fragmento. É importante notar que isso não exclui uma estrutura de dados na CPU, pois ainda é necessário ordenar os tetraedros de acordo com o observador, o que é feito em CPU.

Os dados mínimos necessários para a renderização de um tetraedro consistem nas posições de seus vértices e nos valores do campo escalar associado. Porém, a ordenação de visibilidade dos tetraedros de uma malha requer uma estrutura de dados que permita extrair eficientemente outras informações, como o conjunto das faces externas ou dos tetraedros adjacentes às faces de um determinado tetraedro. Na literatura, podem ser encontradas diversas estruturas de dados (Garimella, 2002) que atendem a esses requisitos. Celes et al. (2004), por exemplo, propõem uma estrutura de dados para malhas de elementos finitos que é ao mesmo tempo compacta e eficiente. Essa estrutura é a utilizada neste trabalho para o armazenamento de dados na CPU.

Tabela 4 – Texturas 2D utilizadas para o armazenamento de uma malha de tetraedros na placa gráfica, para projeção de células.

Textura	**Coordenadas**		**Dados**			
	u	**v**	**r**	**g**	**b**	**a**
Vértices	k		$\vec{v}_k$			s_k
Normais0	t		$\hat{n}_{t,0}$			$o_{t,0}$
Normais1	t		$\hat{n}_{t,1}$			
Normais2	t		$\hat{n}_{t,2}$			
Normais3	t		$\hat{n}_{t,3}$			$o_{t,3}$
Gradientes	t		$\vec{g}_t$			

Na tabela acima, t é o índice do tetraedro e k é o índice de um vértice; $\vec{v}_k$ e s_k são, respectivamente, a posição e o escalar associado ao vértice k; $\hat{n}_{t,i}$ é a normal da *i-ésima* face do tetraedro, enquanto $o_{t,i}$ é o índice de um vértice pertencente a essa face (apenas dois índices são necessários, pois cada vértice pertence simultaneamente a três faces do tetraedro); e $\vec{g}_t$ é o gradiente do tetraedro.

O vetor de atributos (*vertex buffer object*) dos vértices deverá conter, em cada posição, apenas:

- índice do tetraedro (t);
- índice do vértice (k).

A textura contendo a posição do vértice pode ser acessada no programa por vértice para calcular a direção do raio, necessária no caso da projeção em perspectiva. Os outros dados podem ser acessados pelo programa por fragmento para realizar os cálculos da mesma forma que na abordagem original. Agora, a interseção do raio deve ser computada para todas as quatro faces do tetraedro, pois não é mais possível descobrir qual a face que está sendo desenhada. Por outro lado, como as informações de cada vértice não dependem mais dessa face, pode ser utilizada uma faixa de triângulos para desenhar o tetraedro.

Essa estrutura de dados pode ser facilmente modificada, adicionando-se mais duas texturas de normais para permitir a visualização de hexaedros com

faces planas e campo escalar constante (ou seja, com gradiente nulo) no seu interior. Isto pode ser útil, por exemplo, para a visualização de reservatórios naturais de petróleo, simulados por diferenças finitas.

A memória de textura necessária por cada vértice, assumindo-se o número de vértices como 1/5 do número de tetraedros (Beall & Shephard, 1997), é igual a $(4/5)+4+3+3+4+3=17{,}75$ *floats* $=71$ *bytes* . O espaço por tetraedro para o *vertex buffer object*, considerando-se que serão usadas faixas de triângulos, é igual a 6*2 = 12 *floats* = 48 *bytes*. O custo total é, então, igual a 119 *bytes* por tetraedro, o que ainda consome muita memória, mas muito menos do que os 912 *bytes* necessários quando utilizado apenas o *vertex buffer object* .

4.1.2. Traçado de Raios

Para o algoritmo de Traçado de Raios, optamos por uma variação da estrutura de dados utilizada originalmente por Weiler et al. (2003a). Buscamos uma estrutura de dados mais compacta do que a anterior e, ao mesmo tempo, eficiente e conceitualmente menos complexa do que as faixas de tetraedros (Weiler et al., 2004). A estrutura de dados utilizada, representada como texturas 2D, é ilustrada na Tabela 5.

Tabela 5 – Estrutura de dados utilizada para a implementação do algoritmo de Traçado de Raios.

Textura	**Coordenadas**		**Dados**			
	u	**v**	**r**	**g**	**b**	**a**
Normais0	t		$\hat{n}_{t,0}$			$o_{t,0}$
Normais1	t		$\hat{n}_{t,1}$			$o_{t,1}$
Normais2	t		$\hat{n}_{t,2}$			$o_{t,2}$
Normais3	t		$\hat{n}_{t,3}$			$o_{t,3}$
Gradientes	t		$\vec{g}_t$			$\hat{g}_t$
Adjacências	t		$a_{t,0}$	$a_{t,1}$	$a_{t,2}$	$a_{t,3}$

Na tabela, t é o índice do tetraedro, $(\hat{n}_{t,i}, o_{t,i})$ é a equação do plano da *i-ésima* face do tetraedro, $\vec{g}_t$ e $\hat{g}_t$ são, respectivamente, o gradiente e o termo escalar da eq. (3.11), e $a_{t,i}$ é o índice do tetraedro adjacente à *i-ésima* face do tetraedro. O custo de armazenamento por tetraedro é, então, igual a $4*4+4+4=24\ floats = 96\ bytes$, contra os 144 *bytes* da estrutura de dados original, mas ainda superior aos 76 *bytes* necessários para as faixas de tetraedros, considerando-se que os planos das faces são armazenados.

4.2. Integração de segmentos lineares

Para computar a contribuição de um raio no interior de um tetraedro, Weiler et al. (2003a, 2004) utilizam uma função de transferência pré-integrada, armazenada em uma textura 3D. Moreland & Angel (2004) aplicam a integração de segmentos lineares ao algoritmo VICP, baseando-se na abordagem de Weiler et al. (2002), restrita à projeção ortográfica. Para cada "fatia" de um tetraedro, definida por dois pontos de controle da função de transferência (seção 2.3.2), este é enviado para a placa gráfica com os valores dos pontos de controle como parâmetros adicionais. Assim, as "fatias" são determinadas na CPU e cada tetraedro pode ser enviado diversas vezes para a placa gráfica, que é responsável por "cortá-lo" de acordo com os pontos de controle.

Nesta seção, propomos uma adaptação para realizar a integração de segmentos lineares com todos os cálculos realizados diretamente na GPU. Isso é particularmente interessante para o algoritmo de Traçado de Raios, que requer as estruturas de dados na GPU. Nesta proposta, combinamos a técnica de iso-superfícies em GPU, de Roettger et al. (2000) (seção 2.3.3), com a formulação de Moreland & Angel (2004) (seção 2.3.2) para a resolução da integral de renderização de volume para um segmento linear. Ao aplicarmos essas técnicas ao Traçado de Raios, buscamos obter uma melhor qualidade de imagem, em relação à pré-integração, e permitir a modificação interativa da função de transferência.

Em uma função de transferência composta por segmentos lineares, cada *ponto de controle* representa uma iso-superfície ao longo de um raio que atravessa um tetraedro. Para detectar as iso-superfícies, utilizamos uma textura 2D, como a da seção (2.3.3), mas contendo, em cada posição, apenas duas componentes.

Considerando um campo escalar normalizado ($s \in [0,1]$), os dados de uma posição da textura são:

- o valor da primeira iso-superfície atravessada pelo raio (s_{iso}), ou -1;
- o valor da próxima iso-superfície ($prox\_s_{iso}$), ou -1.

Se, por exemplo, forem consideradas três iso-superfícies: s_{iso1} = 0,25, s_{iso2} = 0,5 e s_{iso3} = 0,75, a textura será, então, codificada como ilustrado na Figura 28.

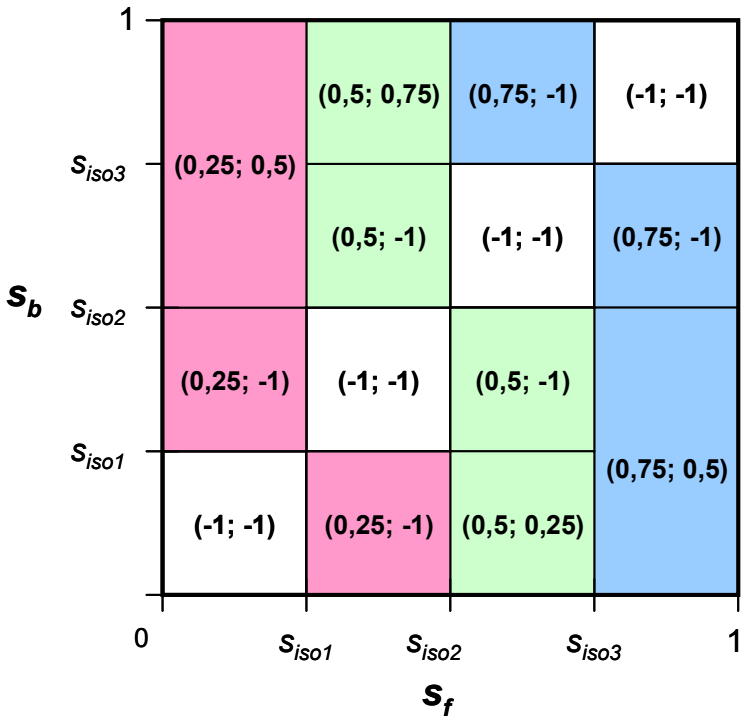

Figura 28 – Textura 2D utilizada para a determinação das iso-superfícies definidas pelos pontos de controle da função de transferência. Em cada posição são armazenados *(s_{iso}; $prox\_s_{iso}$)*, relativos à primeira interseção de um raio entre s_f e s_b, e o valor da próxima iso-superfície. Se s_{iso} for igual a -1, então nenhuma iso-superfície é atravessada.

Os valores dos escalares s_f e s_b, das posições de entrada ($\vec{x}_0$) e saída ($\vec{x}$) do tetraedro, são as coordenadas da textura. Se o valor de s_{iso}, na posição acessada na textura, for -1, então nenhuma iso-superfície foi atravessada pelo raio, o que implica que s_f e s_b estão dentro de um mesmo segmento linear da função de transferência e o raio pode ser diretamente integrado para o segmento. Porém, no caso de haver uma iso-superfície ($s_{iso} \neq -1$), o segmento deverá ser integrado entre s_f e s_{iso}, e os parâmetros passados para o próximo passo do algoritmo devem ser relativos à posição de s_{iso}, ao invés de s_b. Assim, considerando que os

parâmetros recebidos pelo passo atual do raio são (t, λ, R, G, B, A), os parâmetros (t', λ', R', G', B', A'), do próximo passo, podem ser calculados da seguinte forma, onde l é a distância entre $\vec{x}_0$ e $\vec{x}$, e (r,g,b,a) são o resultado da integração do raio entre s_f e s_{iso}:

$t' = t$ (raio permanece no tetraedro atual);

$$\lambda' = \lambda + l\frac{s_{iso} - s_f}{s_b - s_f};$$

$$R' = A * R + (1 - A) * r$$
$$G' = A * G + (1 - A) * g$$
$$B' = A * B + (1 - A) * b$$
$$A' = A + (1 - A) * a$$

O algoritmo pode, então, ser expresso pelo seguinte pseudo-código:

```
1.  siso, prox_siso = Textura2D(sf, sb)
2.  if (siso >= 0)  // Corta iso-superfície.
3.  {
4.    if (sf == siso)
5.      siso = prox_siso;
6.
7.    if (abs(sb - sf) > abs(siso - sf))
8.    {
9.      t' = t;   // Continua no mesmo tetraedro.
10.     l  = l*(siso - sf)/(sb - sf);
11.     lambda' = lambda + l;
12.     sb = siso;
13.   }
14. }
15. r, g, b, a = Integra_Segmento_Linear(sf, sb, l);
16  R', G', B', A' = A*(R, G, B, 1) + (1-A)*(r, g, b, a);
```

Supondo que $s_f = 0{,}9$ e $s_b = 0{,}3$, e considerando as iso-superfícies ilustradas na Figura 28, o resultado da textura será: $s_{iso} = 0{,}75$ e $prox\_s_{iso} = 0{,}5$. O raio deve ser, então, integrado entre 0,9 e 0,75. No passo seguinte, $s_f = 0{,}75$ e $s_b = 0{,}3$. Como s_f já é igual ao valor de uma iso-superfície (linha 4), é preciso avançar para a próxima, que é 0,5. O raio será, então, cortado e integrado entre 0,75 e 0,5. No próximo passo, $s_f = 0{,}5$ e $s_b = 0{,}3$. Novamente, é necessário avançar para a próxima iso-superfície, que é 0,25. Entretanto, $s_b = 0{,}3$. Dessa forma, a iso-supefície não será cortada e o raio será integrado entre 0,5 e 0,3. Isso é determinado pela linha 7 do pseudo-código, que compara os valores absolutos das distâncias $(s_b - s_f)$ e $(s_{iso} - s_f)$. Como $abs(0{,}3 - 0{,}5) < abs(0{,}25 - 0{,}5)$, o raio não

será mais cortado e poderá avançar para o próximo tetraedro. É importante notar que esse teste também funciona para os casos em que, ao avançar para a próxima iso-superfície, o valor seja -1.