3
Algoritmos para visualização volumétrica baseados na GPU

Ao longo dos últimos anos foram utilizadas três principais classes de algoritmos para a renderização direta de volumes de malhas não-estruturadas acelerada por placas gráficas:

- traçado de raios (*ray-tracing*);
- projeção de células (*cell-projection*);
- "fatiamento" (*slicing*).

Nos algoritmos baseados em traçado de raios (Garrity, 1990; Bunyk et al., 1997), um raio é traçado para cada posição (*pixel*) do plano de projeção, e as contribuições do volume ao longo do raio são compostas para a geração da imagem final.

No caso da projeção de células (Shirley & Tuchman, 1990), cada célula do volume é projetada uma de cada vez, e sua contribuição é composta com o resultado das anteriores. Dependendo do modelo óptico utilizado, é necessário que as células sejam desenhadas segundo determinada ordem (normalmente, de trás para frente ou de frente para trás) de forma a garantir a composição correta de volumes semi-transparentes. Há diversos algoritmos para realizar essa ordenação, e alguns são apresentados resumidamente na seção (3.1.1).

Os algoritmos de fatiamento (Yagel et al., 1996; Chopra & Meyer, 2002) utilizam planos perpendiculares a uma determinada direção para "cortar" as células do volume, extraindo, assim, as "fatias" que serão compostas para a geração da imagem final. A qualidade da imagem gerada depende do número de fatias, que deve ser suficientemente alto para garantir uma amostragem adequada dos dados visualizados.

Até recentemente, os algoritmos cujos melhores desempenhos foram reportados na visualização interativa de malhas não-estruturadas eram os baseados em projeção de tetraedros (Moreland & Angel, 2004). O grande sucesso desses

algoritmos se deve à facilidade com que são adaptados para explorar os recursos de rasterização de polígonos das placas gráficas atuais.

Entretanto, utilizando os novos recursos oferecidos pelas placas gráficas programáveis, Weiler et al. (2003a) desenvolveram um algoritmo de traçado de raios que apresenta desempenho competitivo com os algoritmos de projeção de tetraedros, o que mostra o potencial dessas placas para desenvolver novos algoritmos interativos, ou ainda revisitar idéias antigas.

Os resultados reportados na literatura para os algoritmos de fatiamento (Chopra & Meyer, 2002) ainda se apresentam muito distantes dos obtidos com a projeção de células e com o algoritmo de traçado de raios de Weiler et al. (2003a). Também, até o momento, não parece haver uma forma direta de mapeá-los para as placas gráficas programáveis. Dessa forma, as abordagens estudadas e implementadas neste trabalho são baseadas na projeção de tetraedros, que é discutida na seção (3.1), e no algoritmo de traçado de raios de Weiler et al. (2003a), apresentado na seção (3.2).

3.1. Projeção de tetraedros

Um dos primeiros algoritmos para a visualização de malhas não-estruturadas a explorar a aceleração das placas gráficas para a rasterização de polígonos foi proposto por Shirley & Tuchman (1990), e é chamado de *Projected Tetrahedra* (ou Tetraedros Projetados). Esse algoritmo se baseia na projeção de tetraedros lineares, que, primeiramente, são classificados de acordo com o perfil projetado (Figura 16) relativo ao observador. Cada tetraedro projetado é, então, decomposto em triângulos, aos quais são associados valores de propriedades e que são enviados para a placa gráfica para serem desenhados. Recentemente, Wylie et al. (2002) utilizaram um programa por vértice para encapsular as computações dependentes do observador na placa gráfica. Isso proporciona o suporte para a visualização de uma primitiva TETRAEDRO, da mesma forma que as primitivas PONTO, LINHA e TRIÂNGULO, que são suportadas pelas placas gráficas atuais.

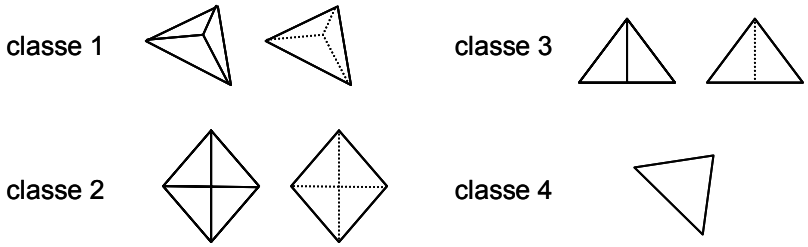


Figura 16 – Classificação de tetraedros projetados de acordo com o número de triângulos necessários para a renderização.

Explorando a programação de placas gráficas, Weiler et al. (2002) propuseram um novo algoritmo para projeção de tetraedros lineares de forma independente do observador, permitindo, também, suportá-lo diretamente pela placa gráfica como uma primitiva TETRAEDRO. Esse algoritmo foi chamado *View-Independent Cell Projection (*VICP*)*, e, devido às limitações das placas existentes na época do seu desenvolvimento, a implementação se restringia à projeção ortográfica. Posteriormente, Weiler et al. (2003b) implementaram o VICP para projeção em perspectiva, utilizando programação por fragmento. Aproveitando a flexibilidade e o desempenho cada vez maiores oferecidos pela programação por fragmentos das placas gráficas mais modernas, o VICP será o algoritmo de projeção de tetraedros utilizado no presente trabalho.

A idéia do VICP se baseia em uma técnica similar ao algoritmo de traçado de raios, mas restrito a apenas um tetraedro. A Figura 17 ilustra um raio que parte do observador e atravessa o tetraedro: s_f e s_b representam, respectivamente, os valores de um campo escalar na posição de entrada e de saída do raio, e l é a distância percorrida pelo raio no interior do tetraedro. Com o campo escalar do volume associado aos vértices do tetraedro, s_f e s_b podem ser obtidos pela interpolação linear desses valores, em relação às faces de entrada e de saída do raio. Se for utilizada pré-integração da função de transferência, (s_f, s_b, l) serão as coordenadas da textura 3D.

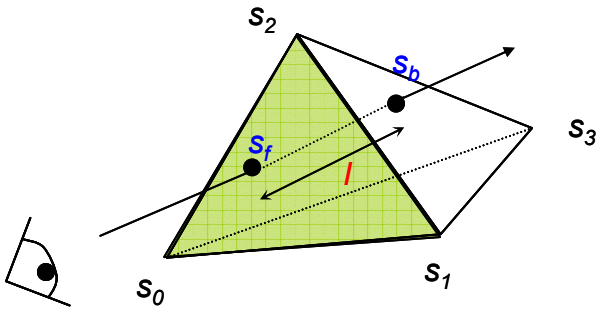


Figura 17 – Raio partindo do observador que atravessa um tetraedro linear.

Neste algoritmo, apenas as faces frontais (*front faces*) são desenhadas. Para manter a independência do observador, todas as faces podem ser enviadas para a placa gráfica, que descartará as outras (*back faces*) automaticamente. O valor de s_f é obtido através da interpolação linear realizada pela placa gráfica durante a rasterização da face, enquanto s_b pode ser calculado com a seguinte equação, onde $\vec{g}$ é o gradiente do campo escalar no tetraedro e $\hat{d}$ é a direção normalizada do raio:

$$s_b = s_f + (\vec{g}.\hat{d})l \qquad \textbf{(3.1)}$$

Em um tetraedro linear, o gradiente é constante e pode ser calculado em pré-processamento.

Se for utilizada projeção ortográfica, a direção do raio será constante para todos os tetraedros. No caso de perspectiva, a direção pode ser facilmente calculada para cada vértice no programa por vértice, subtraindo-se a posição do observador da posição do vértice. O vetor resultante será, então, interpolado automaticamente pela placa gráfica resultando na direção, que deve ser normalizada, para cada fragmento.

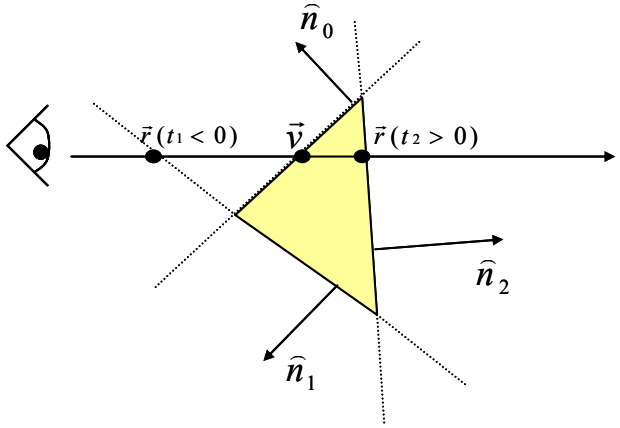


Figura 18 – Projeção lateral de um tetraedro que é atravessado por um raio que parte do observador.

Ainda resta calcular a distância *l* do raio no interior do tetraedro, que é igual à distância entre as posições de entrada e saída do raio. A posição de entrada $\vec{v}$ (Figura 18) é determinada automaticamente pela interpolação linear dos vértices da face frontal desenhada, enquanto que a posição de saída pode ser calculada pela interseção do raio com a face pela qual o raio sai do tetraedro. Considerando a equação do plano de uma face f_i, com normal $\hat{n}_i$, e um ponto $\vec{p}_i$ pertencente ao plano, além da equação paramétrica do raio $\vec{r}$:

$$(\vec{p}_i - \vec{q}).\hat{n}_i = 0 \qquad \textbf{(3.4)}$$

$$\vec{r} = \vec{v} + t\hat{d} \qquad \textbf{(3.5)}$$

o parâmetro do raio t_i, relativo à interseção $\vec{q}$ do raio com a face f_i será:

$$t_i = \frac{(\vec{p}_i - \vec{v}).\hat{n}_i}{\hat{d}.\hat{n}_i} \qquad \textbf{(3.6)}$$

que é igual à distância *l*. Assim, t_i pode ser calculado para as outras três faces do tetraedro, e o problema agora se reduz a descobrir em qual das faces o raio sai do tetraedro. Se $t_i < 0$ (Figura 18), então o raio corta o plano da face f_i antes da posição onde o raio se inicia (no caso, $\vec{v}$). Logo, o raio não pode estar saindo do tetraedro. Como conseqüência, a face de saída será a que tiver o menor t_i positivo, o que significa que f_i é a primeira face cortada pelo raio na direção $\hat{d}$.

Esse algoritmo pode ser implementado utilizando-se a programação por vértice e por fragmento. Para cada um dos 3 vértices das 4 faces do tetraedro, o que resulta em 12 vértices por tetraedro, os seguintes atributos devem ser enviados para a placa gráfica:

- posição do vértice ($\vec{v}$);
- valor do campo escalar no vértice (s);
- normais das outras faces ($\hat{n}_a, \hat{n}_b, \hat{n}_c$);
- posição do vértice oposto à face, pertencente às outras três ($\vec{p}_i$);
- gradiente do tetraedro ($\vec{g}$).

A memória necessária para cada vértice, juntamente com seus atributos, é igual a $3+1+3*3+3+3=19\ floats = 76\ bytes$, considerando-se que cada valor é representado por um número de ponto flutuante (*float*) de 4 *bytes*. Como cada tetraedro requer 12 vértices, o custo por tetraedro é igual a 912 *bytes*.

Como atualmente não é possível compartilhar informações entre os vértices de uma mesma face (ou tetraedro) nos programas por vértice, é necessário que algumas informações sejam enviadas de forma redundante. Cada atributo do vértice será interpolado automaticamente durante a rasterização da face. Dessa forma, para cada fragmento, estarão disponíveis a posição $\vec{v}$ de entrada do raio e o escalar sf nessa posição, além dos outros atributos, que são iguais para todos os vértices da face. Com essas informações, as eq. (3.6) e (3.1) podem finalmente ser resolvidas no programa por fragmento.

Devido aos atributos de um vértice serem diferentes para as 3 faces de que o vértice faz parte, não é possível utilizar uma "faixa" (*strip*) de triângulos para desenhar as faces do tetraedro, o que reduziria o número de vértices enviados para 6 por tetraedro. De fato, como reportam Weiler et al. (2002), o grande volume de dados transferidos para a placa gráfica representa o "*gargalo*" do VICP.

Se for utilizada projeção ortográfica, o número de parâmetros por vértice pode ser reduzido. Para um raio que entra em um tetraedro pela face f_3 (Figura 19), a face de saída do raio deve ser f_0, f_1 ou f_2. Weiler et al. (2002) observam que, para projeção ortográfica, a distância t_i, entre o ponto de entrada do raio e a interseção com a face f_i, pode ser interpolada linearmente ao longo da face de entrada, para cada possível face de saída f_i. Por exemplo, as distâncias do vértice v_1 para as faces f_0, f_1 e f_2 na direção do raio são, respectivamente, 0, t_1 e 0, pois esse vértice é compartilhado pelas faces f_0 e f_2 (e também f_3). O equivalente ocorre para os outros dois vértices, v_0 e v_2, da face f_0. Logo, a distância t_i de qualquer posição na face f_3 para as outras faces pode ser determinada pela interpolação linear das distâncias associadas aos vértices de f_3 (Figura 19).

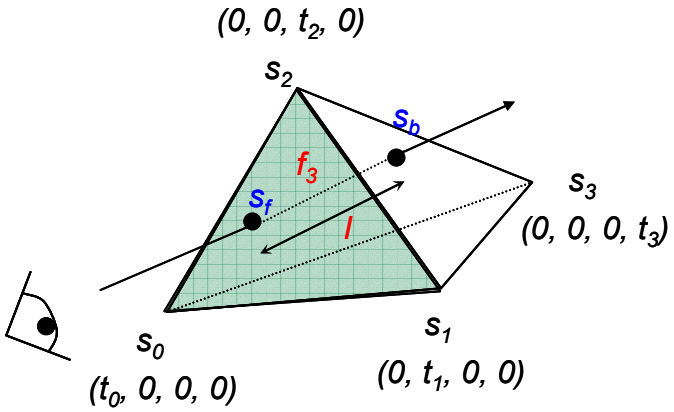


Figura 19 – Raio que parte do observador e atravessa um tetraedro. Cada vértice do tetraedro possui um vetor (t_0, t_1, t_2, t_3) das respectivas distâncias às faces f_0, f_1, f_2 e f_3, na direção do raio.

Dessa forma, para cada vértice v_i, podemos enviar como parâmetro a equação do plano $\vec{o}_i$ da face oposta e calcular a distância t_i no programa por vértice. Os atributos necessários para cada vértice são:

- posição do vértice ($\vec{v}$);
- valor do campo escalar no vértice (s);
- equação do plano da face oposta ao vértice ($\vec{o}_i$);
- gradiente do tetraedro ($\vec{g}$).

Uma vantagem é que, além de reduzir o número de atributos de cada vértice, essa otimização permite utilizar uma faixa de triângulos para desenhar um tetraedro, pois os atributos não dependem da face que está sendo desenhada, mas apenas do vértice.

O custo de armazenamento por vértice será, então, igual a 11 *floats* = 44 *bytes*. Se forem utilizadas faixas de triângulos, o custo por tetraedro será igual a 264 *bytes*.

Em seu sistema para visualização volumétrica, Moreland & Angel (2004), baseando-se no modelo óptico de emissão e absorção e na abordagem de Weiler et al. (2002), restrita à projeção ortográfica, utilizam as idéias do VICP mas defendem uma abordagem dependente do observador. Os algoritmos de projeção de células requerem uma ordenação de visibilidade dependente do observador. Essa ordenação deve ser realizada a cada vez em que a malha é desenhada, o que

implica em enviar novamente os vértices, ou seus índices, para a placa gráfica, o que é reportado por Weiler et al. (2002) como o gargalo do algoritmo. Dessa forma, não haveria custo adicional significativo em se realizarem mais cálculos na CPU. Moreland & Angel (2004) apostam nisso para reduzir a quantidade de atributos enviados por vértice, computando o parâmetro t_i de cada vértice na CPU e enviando-o como atributo do vértice. Além disso, é enviado o valor do campo escalar s_{bi} na posição determinada por t_i, o que permite eliminar o gradiente do tetraedro como um atributo do vértice. O índice local do vértice no tetraedro também é necessário, de forma que o programa de fragmentos possa representar t_i e s_{bi} como tuplas que serão interpoladas. A interpolação de s_{bi} é equivalente à de t_i (Figura 19). Os atributos por vértice são:

- posição do vértice ($\vec{v}$);
- valor do campo escalar no vértice (s)
- distância do vértice para a face oposta, na direção do raio (t_i);
- valor do campo escalar na interseção do raio com a face oposta ao vértice (s_{bi});
- índice local do vértice no tetraedro ($i \in [0,3]$).

Dessa forma, Moreland & Angel (2004) reduzem o custo de memória por vértice para 7 *floats* = 28 *bytes*, e conseqüentemente, com o uso de faixas de triângulos, para 168 *bytes* por tetraedro.

Entretanto, neste trabalho, a restrição à projeção ortográfica não foi uma opção. Assim, nos baseamos na abordagem de Weiler et al. (2003b) e aumentamos o esforço do programa por fragmento para remover o gargalo do VICP, como apresentado na seção (4.1.1).

3.1.1. Ordenação de células

Como mencionado, além de uma forma de desenhar cada uma das células, os algoritmos baseados em projeção de células requerem uma *ordenação de visibilidade*, considerando o modelo óptico de emissão e absorção.

Williams (1992) define que "*uma ordenação de visibilidade de um conjunto de objetos, a partir de um observador, é uma ordenação tal que, se o objeto a obstrui o objeto b, então b precede a na ordenação*".

Diversos algoritmos de ordenação de visibilidade para malhas de poliedros foram desenvolvidos, sendo que os mais populares são baseados no algoritmo *Meshed Polyhedra Visibility Ordering* (MPVO), apresentado por Williams (1992). Esse algoritmo ordena um conjunto de células convexas e de faces planas, de uma malha também convexa e acíclica (i.e., sem ciclos de visibilidade (Figura 20)). A grande vantagem do MPVO é ser capaz de realizar a ordenação em tempo e espaço de armazenamento lineares (*O(n)*) em relação ao número *n* de células, além de sua simplicidade.

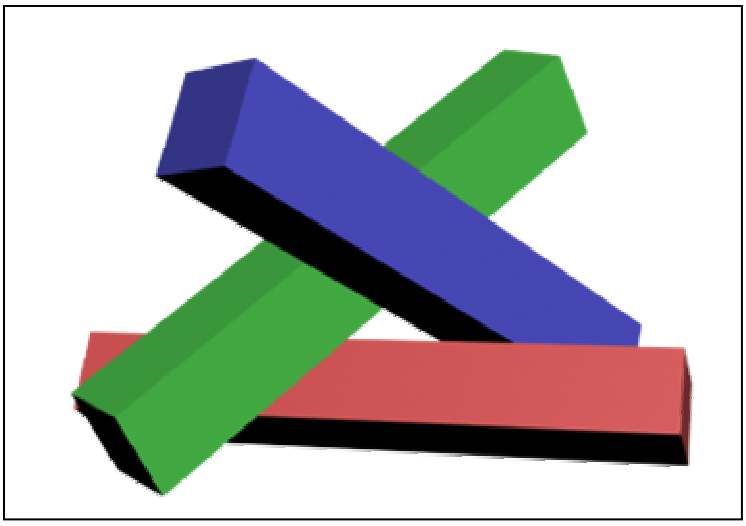


Figura 20 – Exemplo de um ciclo de visibilidade.

O MPVO é composto por três etapas. A primeira (pré-processamento) consiste em construir um grafo de adjacências entre as células (Figura 21) e computar as equações dos planos de todas as faces internas da malha. Na segunda etapa, o algoritmo (assumindo que a malha é acíclica) transforma o grafo de adjacências em um grafo orientado acíclico (DAG), atribuindo direções às arestas do grafo de adjacências (Figura 21a). Para isso, avalia-se a equação do plano de cada face interna e a aresta do grafo é marcada como "entrando", "saindo" ou "nenhum", se a posição do observador estiver à frente, atrás ou sobre o plano, respectivamente (para projeção ortográfica, a direção de visualização é que será considerada). Assim, para duas células que compartilham uma face, se em uma a face estiver marcada como "entrando", na outra estará "saindo", ou ambas estarão marcadas como "nenhum", o que implica que não existe uma conexão no DAG para a face. Finalmente (Figura 21b), a terceira etapa realiza uma ordenação

topológica do DAG, utilizando busca em largura (BFS) ou profundidade (DFS), o que resulta em uma ordenação de visibilidade das células. No caso da busca em profundidade, as células que não possuem arestas no DAG marcadas como "saindo" são colocadas em uma lista *L*, durante a segunda etapa. Para cada célula da lista, todas as suas adjacências que estejam marcadas como "entrando" são, então, visitadas recursivamente.

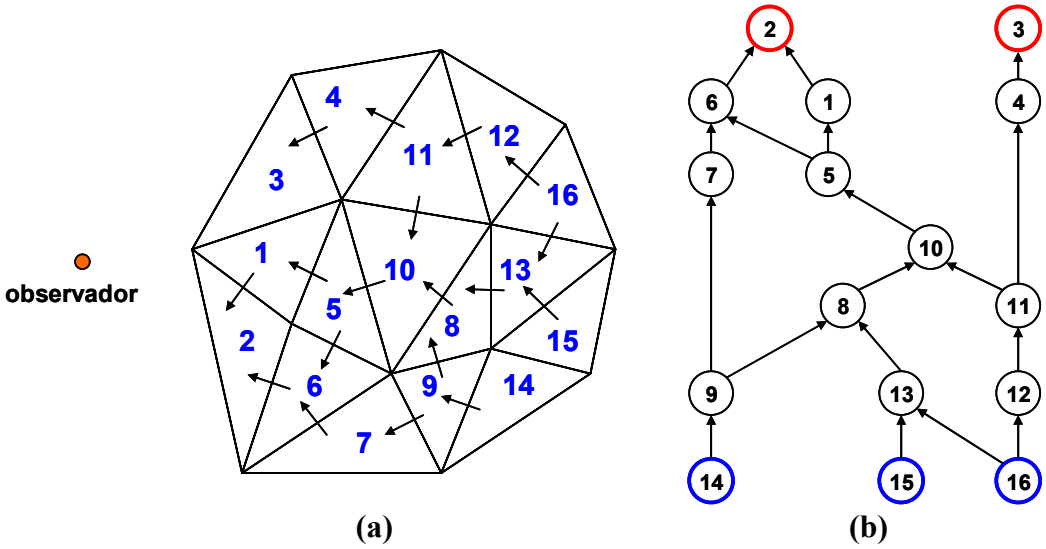


Figura 21 – Ilustração do algoritmo MPVO. (a) As setas representam a classificação das faces internas como "entrando", "saindo" ou "nenhum", em relação ao observador. (b) Grafo orientado criado a partir da classificação das faces.

Adicionalmente, Williams (1992) apresenta uma extensão do MPVO para a ordenação de visibilidade de malhas não-convexas, chamada MPVONC, com complexidade $O(n + b \log b)$, sendo n o número total de células e b o número de células externas (da fronteira da malha). O MPVONC é uma heurística, o que implica que, em alguns casos, podem ocorrer falhas na ordenação de uma malha. Posteriormente, outras extensões foram propostas para o MPVO. A primeira a conseguir ordenar malhas não-convexas foi apresentada por Silva et al. (1998) e utiliza o paradigma de plano de varredura (de Berg et al., 2000) para estender o DAG da segunda etapa do MPVO, com relações adicionais entre as faces externas de uma malha não-convexa (Figura 22). Comba et al. (1999) utilizaram árvores BSP para determinar essas relações adicionais. Kraus & Ertl (2001) apresentam uma extensão para o MPVO que permite ordenar malhas com ciclos de visibilidade. Para isso, observam que um ciclo de visibilidade pode ser

representado como componentes fortemente conexas de um grafo orientado. Assim, modificam a terceira etapa do MPVO para extrair e ordenar esses ciclos, e dividem as células envolvidas para que elas possam ser visualizadas corretamente. Recentemente, Cook et al. (2004) utilizaram o espaço da imagem projetada para determinar as relações adicionais ao MPVO, reportando os melhores tempos entre as extensões do MPVO que ordenam malhas não-convexas. Para criar as relações adicionais entre as faces externas, a placa gráfica é utilizada para desenhá-las, armazenado o resultado em cada elemento (*pixel*) da imagem gerada em uma estrutura similar a um *A-Buffer* (Carpenter, 1984), que mantém as contribuições de cada pixel de forma ordenada. Finalmente, o resultado final de cada pixel é utilizado para criar as relações adicionais.

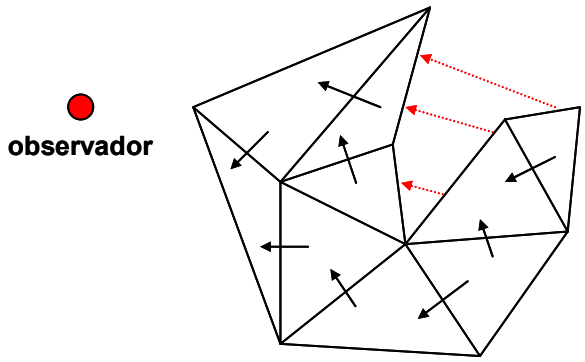


Figura 22 – Extensão das relações de adjacência do MPVO, para malhas não-convexas, representada pelas setas vermelhas.

Outras abordagens para a ordenação de células podem ser encontradas nos trabalhos de Carpenter (1984), Wittenbrink (2001), Cignoni & Floriani (1998), Farias et al. (2000) e Stein et al. (1994).

Neste trabalho, o MPVONC foi utilizado para a ordenação das células. Essa heurística apresentou resultados satisfatórios, tanto quanto ao desempenho como quanto à qualidade das imagens (apesar de ser uma heurística) para as malhas de elementos finitos que foram testadas.

3.2. Traçado de raios na placa gráfica

O algoritmo proposto por Weiler et al. (2003a) é baseado no paradigma de traçado de raios. Com praticamente todos os cálculos sendo realizados através da

programação por fragmento na placa gráfica, Weiler et al. (2003a) conseguiram implementar um sistema interativo para a visualização de malhas de tetraedros lineares, com desempenho competitivo com os algoritmos de projeção de células. Esse algoritmo estende o algoritmo VICP, que utiliza traçado de raios apenas no interior do tetraedro que será projetado, para percorrer uma seqüência de tetraedros ao longo de cada raio que parte do observador.

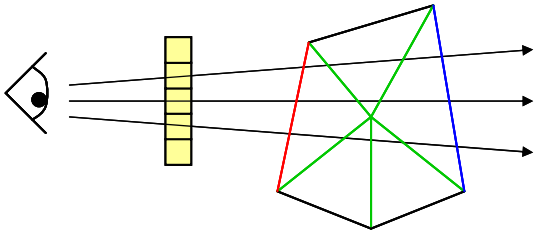


Figura 23 – Propagação de raios que partem do observador e atravessam a malha de tetraedros. A cada passo de um raio, é atravessado um tetraedro.

Seguindo a idéia geral dos algoritmos de traçado de raios (Figura 23), cada elemento (*pixel*) da imagem projetada corresponde a um raio que parte do observador e passa pela posição do pixel. O valor final é igual ao resultado da composição das contribuições de cada tetraedro interceptado ao longo do raio.

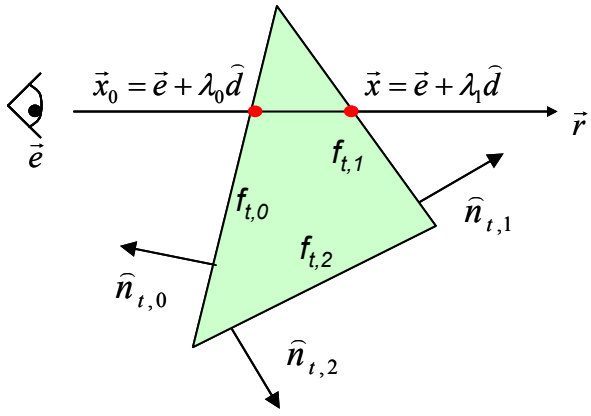


Figura 24 – Interseção de um raio com as faces de um tetraedro.

Inicialmente, é preciso encontrar a primeira interseção, que estará localizada em uma face da fronteira externa da malha. Essa face representa a posição de entrada $\vec{x}_0$ do raio em um tetraedro (Figura 24), assim como no VICP. Como conseqüência, a próxima interseção será a posição de saída $\vec{x}$, que, por sua vez, é

igual à posição de entrada do raio em um tetraedro adjacente, e assim sucessivamente (Figura 23), até que o raio atinja a posição de saída da malha, que será uma face sem um tetraedro adjacente. A cada passo, as contribuições de cor e opacidade de cada tetraedro ao longo do raio são compostas com o resultado anterior, da frente para trás. Assim, para cada raio, as etapas do algoritmo são:

1. Encontre a primeira interseção.
2. Enquanto o raio estiver dentro da malha:
   2.1. Determine a posição de saída do tetraedro.
   2.2. Calcule a contribuição do tetraedro.
   2.3. Componha com o resultado anterior.
   2.4. Avance para o tetraedro adjacente.

Cada passo representa um tetraedro que é atravessado pelo raio, como na Figura 24. Considerando a eq. (3.7), do plano da *i-ésima* face ($f_{t,i}$) do tetraedro *t*, com normal $\hat{n}_{t,i}$ e um vértice $\vec{v}_{t,3-i}$ da face, além da equação paramétrica do raio $\vec{r}$ (eq. 3.8), que parte do observador $\vec{e}$ na direção normalizada $\hat{d}$, temos:

$$(\vec{v}_{t,3-i} - \vec{q}).\hat{n}_{t,i} = 0 \quad \textbf{(3.7)}$$

$$\vec{r} = \vec{e} + \lambda\hat{d} \quad \textbf{(3.8)}$$

O parâmetro do raio λ_i, relativo à interseção $\vec{q}$ do raio com a face $f_{t,i}$ será, então:

$$\lambda_i = \frac{(\vec{v}_{t,3-i} - \vec{e}).\hat{n}_{t,i}}{\hat{d}.\hat{n}_{t,i}} \quad \textbf{(3.9)}$$

Se a face de entrada for conhecida, o parâmetro λ_i deve ser calculado para as outras três faces, e a posição de saída será correspondente ao menor λ_t que seja maior que o valor relativo à posição de entrada.

O valor do campo escalar $s(\vec{x})$ na posição de saída $\vec{x}$ do tetraedro pode ser calculado da seguinte forma:

$$s(\vec{x}) = \vec{g}_t.(\vec{x} - \vec{x}_0) + s(\vec{x}_0) = \vec{g}_t.\vec{x} + (s(\vec{x}_0) - \vec{g}_t.\vec{x}_0) \quad \textbf{(3.10)}$$

onde $\vec{g}_t$ é o gradiente do campo escalar no tetraedro *t*, e $\vec{x}_0$, neste caso, pode ser uma posição qualquer no interior ou em uma face do tetraedro. Definindo $\hat{g}_t = (s(\vec{x}_0) - \vec{g}_t.\vec{x}_0)$, a eq. (3.10) pode ser reescrita como:

$$s(\vec{x}) = \vec{g}_t.\vec{x} + (s(\vec{x}_0) - \vec{g}_t.\vec{x}_0) = \vec{g}_t.\vec{x} + \hat{g}_t \quad \textbf{(3.11)}$$

Para que seja possível propagar o raio, cada passo deve receber como parâmetros o índice *t* do tetraedro, o parâmetro λ, relativo à posição de entrada do raio no tetraedro e a cor e opacidade compostas até o passo anterior. Assim, as informações podem ser representadas como uma tupla (*t, λ, R, G, B, A*). Além dos parâmetros, deve-se ter acesso a uma estrutura de dados contendo as informações geométricas e topológicas da malha de tetraedros.

A propagação deve ser feita para cada pixel da tela de projeção, o que sugere a utilização de um programa por fragmento como forma de implementação do algoritmo na placa gráfica. Para isso, pode-se desenhar um retângulo do tamanho da tela, apenas como uma forma de permitir que o programa seja executado para todos os *pixels*. Se a placa gráfica suportar fluxo dinâmico (com estruturas do tipo *while ... do ... end*), a propagação completa de um raio pode ser realizada no programa por fragmento, que precisará ser executado apenas uma vez para cada pixel.

Entretanto, na maioria das placas gráficas disponíveis atualmente, essa facilidade ainda não se encontra disponível. Assim, uma forma de realizar a propagação do raio consiste em desenhar vários retângulos do tamanho da tela, de modo que, a cada retângulo desenhado, seja executado um passo da propagação. Os retângulos são desenhados em uma textura 2D do tamanho da tela de projeção, que será utilizada no passo seguinte como uma forma de transferir dados entre passos subseqüentes.

A posição de entrada de um raio na malha de tetraedros pode ser determinada simplesmente desenhando-se as faces externas. Já para descobrir se o raio saiu da malha, basta comparar o índice *t* do tetraedro atual com um valor inválido (o índice *0* (zero) pode ser convenientemente utilizado). Em caso positivo, o raio não deve continuar sendo propagado, o que implica em não mais executar o programa por fragmento para o respectivo pixel. Se for utilizado fluxo dinâmico no programa por fragmento, a implementação é direta, mas, no caso contrário, o problema se torna um pouco mais complicado. Na verdade, dois casos devem ser tratados: *um raio saiu da malha* e *todos os raios saíram da malha*.

Quanto ao primeiro, mesmo que o raio não deva mais ser propagado, é necessário continuar desenhando retângulos para permitir a propagação dos outros

que ainda não deixaram a malha. Enquanto estiverem sendo desenhados retângulos, o resultado final do raio deve ser comunicado para o passo seguinte.

Nas placas gráficas, a determinação da visibilidade de um pixel é realizada utilizando-se o algoritmo de *z-buffer* (Foley et al., 1997). Assim, uma maneira de descobrir se todos os raios saíram da malha, o que significa que se deve parar de desenhar retângulos, consiste em executar um programa por fragmento para colocar no z-buffer um valor mínimo quando o raio não estiver mais na malha, de modo que, no passo seguinte, o fragmento não será mais desenhado. Isso pode ser implementado como um passo adicional, executado após um determinado número de passos. Esse passo é responsável por "marcar" o z-buffer, cumulativamente, a cada vez em que é executado, até que nenhum fragmento passe mais pelo teste de profundidade. A contagem do número de fragmentos desenhados (*occlusion query*) é uma facilidade suportada de forma relativamente eficiente pelas placas gráficas modernas.

Em malhas não-convexas (Figura 25), às vezes é necessário que um raio que deixou a malha entre novamente. Uma limitação do algoritmo de Weiler et al. (2003a) é que, quando o raio sai da malha, não é mais possível explorar a coerência entre tetraedros adjacentes para permitir que ele entre novamente. Assim, na abordagem original, somente malhas convexas são suportadas.

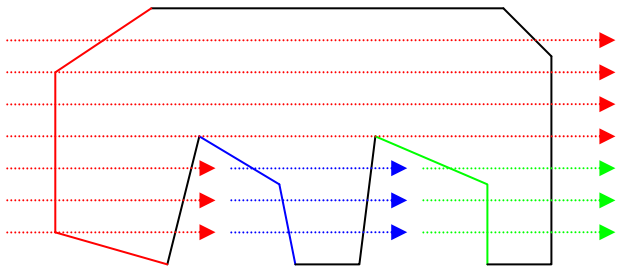


Figura 25 – Os raios de cores diferentes representam as camadas necessárias para a visualização de uma malha não-convexa.

Enquanto Weiler et al. (2003a) tornam as malhas convexas adicionando células "inativas", Weiler et al. (2004), e paralelamente Bernardon et al. (2003), propõem utilizar a técnica de "descamação" (*depth-peeling*) (Everitt, 2001). Como ilustrado na Figura 25, para cada vez que raios entram na malha, é desenhada uma nova "camada" de faces frontais para determinar as novas

posições de entrada, que serão relativas às faces frontais mais próximas do observador, desconsiderando-se as camadas anteriores. A implementação pode ser realizada, considerando o algoritmo de z-buffer, por um programa por fragmento utilizado para comparar a profundidade de um fragmento projetado com o valor relativo à camada anterior, armazenado em uma textura 2D, de forma a permitir somente os valores de profundidade maiores. Isso é complementado com o teste de profundidade convencional da placa gráfica, configurado para permitir apenas os valores de fragmentos menores do que os armazenados no z-buffer. Dessa forma, a cada vez em que é determinado que todos os raios saíram da malha, é necessário desenhar novamente as faces externas para descobrir se há raios que voltam a entrar na malha. O algoritmo termina quando a contagem do número de fragmentos desenhados for igual a zero.

Assim, as etapas do algoritmo modificado são:

1. Desenhe uma camada.
2. Se há fragmentos desenhados:
   2.1. Enquanto o raio estiver dentro da malha:
      2.1.1. Determine a posição de saída do tetraedro.
      2.1.2. Calcule a contribuição do tetraedro.
      2.1.3. Componha com o resultado anterior.
      2.1.4. Avance para o tetraedro adjacente.
   2.2. Vá para o item 1.

3.2.1. Estruturas de dados

As estruturas de dados utilizadas por este algoritmo devem estar armazenadas na memória da placa gráfica, de modo que o programa por fragmento possa acessá-las e, assim, realizar as operações necessárias à propagação do raio ao longo da malha de tetraedros. Portanto, devem ser representadas como texturas 1D, 2D ou 3D, que são atualmente as únicas formas possíveis de acesso a grandes volumes de dados estáticos por um programa por fragmento. Como a dimensão de cada coordenada de uma textura ainda é limitada (as placas atuais suportam, em média, até 4096 posições), as texturas 1D só

podem ser utilizadas para pequenos volumes de dados. Dessa forma, as texturas 2D são as mais utilizadas, transformando-se o índice *i* de uma posição de um vetor unidimensional em um par de coordenadas de textura *(u,v)*, como ilustrado na Figura 26. O índice *t* de um tetraedro, por exemplo, pode ser decomposto em dois índices: t_u e t_v, representados como coordenadas de uma textura 2D contendo as informações do tetraedro.

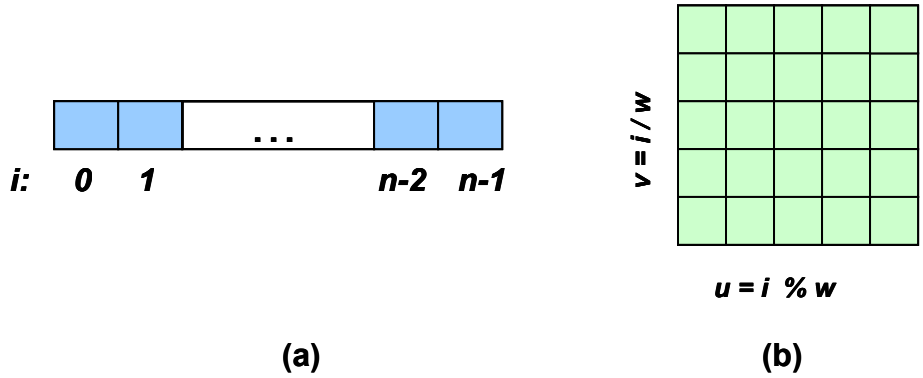


Figura 26 – (a) Vetor unidimensional indexado pelo índice *i*. (b) Textura 2D equivalente ao vetor. O índice *i* é decomposto em duas coordenadas de textura, *u* e *v*. As dimensões da textura são *w* e *h*.

As informações que são passadas de um passo do algoritmo para o seguinte podem ser codificadas em uma textura 2D, em função das coordenadas *(x, y)* dos pixels da tela de projeção, como na Tabela 1. Para cada posição da textura, os dados (*t, λ, R, G, B, A*) são armazenados nas componentes de cor e opacidade *(r, g, b, a)*. As placas gráficas modernas (NVIDIA, 2004b) permitem criar texturas com valores de ponto flutuante (*float*) de 32 *bits* por componente, além de compactar dois valores de "meia-precisão" (*half*), de 16 bits, em um *float* de 32 bits. Segundo NVIDIA (2004b), inteiros no intervalo [-2048, 2048] podem ser exatamente representados como um valor *half*. Assim, informações tais como o índice do tetraedro atual, podem ser representadas por valores do tipo *half* e compactadas em *floats*, para serem armazenadas na textura. As outras informações também podem ser compactadas da mesma forma, permitindo alguns erros de precisão que, em geral, não representam um problema. No caso da cor associada e da opacidade, a precisão de 16 *bits* ainda é melhor do que os 8 *bits* utilizados tradicionalmente.

Tabela 1 – Dados armazenados na textura 2D utilizada para a passagem de parâmetros para um passo da propagação de um raio.

Coordenadas		**Dados**					
u	**v**	**r**	**g**	**b**		**a**	
x	y	t	λ	R	G	B	A

O programa por fragmento também precisa acessar as informações geométricas e topológicas dos tetraedros da malha. Devido às limitações das placas gráficas, quando o algoritmo original foi desenvolvido, Weiler et al. (2003a) utilizaram uma estrutura de dados simples na qual, para cada tetraedro de índice t, são armazenados as posições dos vértices, as normais das faces e o gradiente do campo escalar. As únicas informações topológicas são os índices dos tetraedros adjacentes às faces de cada tetraedro. Essa estrutura pode ser implementada com texturas 3D, como mostrado na Tabela 2. Bernardon et al. (2004) decompõem cada textura 3D em texturas 2D, cujo acesso é normalmente mais eficiente.

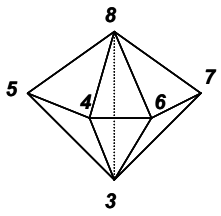
Tabela 2 – Estrutura de dados utilizada para o Traçado de Raios na placa gráfica, originalmente usada por Weiler et al. (2003a).

Textura	**Coordenadas**			**Dados**			
	u	**v**	**w**	**r**	**g**	**b**	**a**
Vértices	t		i	$\vec{v}_{t,i}$			
Normais	t		i	$\hat{n}_{t,i}$			$f_{t,i}$
Gradientes	t			$\vec{g}_t$			$\hat{g}_t$
Adjacências	t		i	$a_{t,i}$			

Na tabela acima, $\hat{n}_{t,i}$ é a normal da face com índice local i no tetraedro t, $\vec{v}_{t,i}$ é o vértice oposto à face, $a_{t,i}$ é o índice do tetraedro adjacente, $f_{t,i}$ é o respectivo índice local da face no tetraedro adjacente, $\vec{g}_t$ é o gradiente do tetraedro e $\hat{g}_t$ é o termo escalar da eq. (3.11). Na abordagem original, o índice local da face de entrada do tetraedro também é passado como parâmetro de cada passo da propagação do raio.

O espaço de memória necessário para a estrutura de dados é igual a $(4*3)T + (4*4)T + 4T + 4T = 36T$, sendo T o número de tetraedros. Isso corresponde, por tetraedro, a *36 floats = 144 bytes*, o que representa um alto custo de memória para grandes malhas. Entretanto, Weiler et al. (2003a) observaram que, futuramente, com a evolução das placas gráficas, o espaço poderia ser reduzido, por exemplo, para *16 bytes* por vértice e *64 bytes* por tetraedro com otimizações como índices para normais e vértices, redução da precisão da normal para o tipo *half* e armazenamento de apenas uma normal para uma face compartilhada por dois tetraedros.

Weiler et al. (2004) propõem a codificação das malhas em faixas de tetraedros (Figura 27) como forma de reduzir o custo de armazenamento. Nesta codificação, a adjacência de uma face de um tetraedro é determinada por duas informações: o *índice da faixa (strip) onde está o tetraedro vizinho* e o *índice do primeiro vértice do tetraedro vizinho na respectiva* faixa. Assim, (2,3) corresponde ao terceiro vértice da segunda faixa. No exemplo da Figura 27, essas informações estão armazenadas no primeiro vértice v_k de cada tetraedro t, e (0,0) representa uma face sem adjacência. Observando que, em um tetraedro do interior de uma faixa, as adjacências a_0 e a_3 pertencem à mesma faixa do tetraedro, a_1 e a_2 podem ser deslocados para o próximo vértice e a_0 e a_3 compactados em a_1 e a_2, como ilustrado na Figura 27c. Para a_0 e a_3, é associado um valor igual a 1, para os vizinhos da mesma faixa ("vizinhos implícitos"), ou 0, para os de outra faixa ("vizinhos explícitos").



faixa (*strip*) 2						
v_k	5	4	3	8	6	7
a_0	2,2	2,3	1,6	-	-	-
a_1	0,0	1,3	3,5	-	-	-
a_2	1,1	3,3	0,0	-	-	-
a_3	3,1	2,1	2,2	-	-	-
	1	2	3	4	5	6

faixa (*strip*) 2 (compactada)						
v_k	5	4	3	8	6	7
a_0	-	1	1	0	-	-
a_1	-	0,0	1,3	3,5	1,6	-
a_2	3,1	1,1	3,3	0,0	-	-
a_3	-	0	1	1	-	-
	1	2	3	4	5	6

(a) **(b)** **(c)**

Figura 27 – (a) Faixa com 3 tetraedros. (b) Representação da faixa, onde v_k é o índice global de um vértice e a_0-a_3 são as adjacências de um tetraedro da faixa, armazenadas na posição do primeiro vértice do tetraedro. (c) Representação compactada da faixa.

Cada uma das adjacências a_1 ou a_2, da faixa compactada, pode ser armazenada em um *float* de 32 *bits*, utilizando 16 *bits* para os índices da faixa e do tetraedro na faixa. Os valores (0 ou 1) de a_0 e a_3 podem ser compactados nos *bits* mais significativos de a_1 e a_2, que serão efetivamente armazenadas, juntamente com o índice do vértice v_k. Assim, para uma faixa cujo comprimento tende ao infinito, o custo de memória por tetraedro, relativo à faixa, é aproximadamente igual a 3 *floats*. As texturas utilizadas para a implementação dessa estrutura de dados são apresentadas na Tabela 3.

Tabela 3 – Texturas 2D utilizadas para o armazenamento de faixas de tetraedros.

Textura	**Coordenadas**		**Dados**			
	u	**v**	**r**	**g**	**b**	**a**
Faixas (*strips*)	t		$v_{t,0}$	$a_{t,1}$	$a_{t,2}$	
Vértices	k		$\vec{v}_k$			s_k
Normais	j		$\hat{n}_j$			o_j
Gradientes	t		$\vec{g}_t$			$\hat{g}_t$
Índice das normais	t		$n_{t,0}$	$n_{t,1}$	$n_{t,2}$	$n_{t,3}$

Na tabela acima, t é o índice do tetraedro, k é o índice de um vértice e j é o índice do vetor normal ao plano de uma face; $v_{t,0}$ é o índice do primeiro vértice de um tetraedro da faixa, enquanto que $a_{t,1}$ e $a_{t,2}$ são as informações de adjacência compactadas; $\vec{v}_k$, $(\hat{n}_j, o_j)$ e $\vec{g}_t$ são a posição de um vértice, a equação do plano de uma face e o gradiente de um tetraedro, respectivamente; s_k é o escalar associado ao vértice, e $\hat{g}_t$ é o termo escalar da eq. (3.11) . Finalmente, $n_{t,i}$ é o índice da equação do plano da *i-ésima* face do tetraedro, na textura de normais.

Apenas as texturas de Faixas e de Vértices são necessárias, segundo observam Weiler et al. (2004), pois o valor do campo escalar $s(\vec{x})$ em qualquer posição do tetraedro pode ser interpolado utilizando-se coordenadas baricêntricas. As equações dos planos das faces do tetraedro também podem ser calculadas a partir da posição dos vértices. Dessa forma, as outras três texturas são opcionais, e são utilizadas para reduzir o tamanho do programa por fragmento. Se apenas as duas primeiras texturas forem utilizadas (*Faixas* e *Vértices*), e assumindo que o

número de vértices V é aproximadamente 1/5 do número de tetraedros T (Beall & Shephard, 1997), então o custo por tetraedro é $3 + 4/5 \approx 3{,}8\, floats \approx 15\, bytes$.

Por outro lado, quando as texturas dos planos das normais são utilizadas, a textura de vértices não é necessária, já que as interseções do raio podem ser computadas diretamente pelas equações desses planos. Considerando as texturas de normais e gradientes, e cada normal compartilhada por dois tetraedros sendo armazenada apenas uma vez, então o custo por tetraedro será aproximadamente $3 + 2 * 4 + 4 + 4 = 19\, floats = 76\, bytes$, o que representa quase metade do custo da estrutura de dados original, utilizada por Weiler et al. (2003a).

A utilização de faixas de tetraedros representa uma alternativa interessante para reduzir a memória utilizada. Entretanto, também introduz alguma complexidade. O problema da minimização do número de faixas para uma malha é NP-completo (Weiler et al., 2004), o que implica na aplicação de heurísticas. Em comparação com a utilização de 4 índices de vértices e 4 de adjacências por tetraedro, Weiler et al. (2004) reportam uma taxa da compressão de até 56% para a heurística que foi utilizada. Também determinam um limite inferior teórico de aproximadamente 37% para o tamanho da malha compactada em relação ao original.

Devido às limitações impostas pelas placas gráficas ao tamanho das texturas, devem ser utilizadas texturas 2D para armazenar as faixas de tetraedros. O método usado por Weiler et al. (2004) consiste em colocar toda uma faixa em apenas uma linha da textura, para que os vértices da faixa possam ser acessados seqüencialmente. Isto resulta em desperdício de memória quando uma faixa não ocupa a linha inteira. Assim, utilizam um algoritmo guloso para tentar maximizar a utilização de cada linha da textura, armazenando mais de uma faixa por linha.