Representação de Subdivisões Espaciais com Aplicação na Triangulação de Delaunay

MARCELO EUGÊNIO KALLMANN

Laboratório de Computação Gráfica - COPPE-Sistemas / UFRJ
21945-970, Rio de Janeiro, RJ, Brasil, Caixa Postal 68511
kallmann@cos.ufrj.br

Abstract. There are many algorithms for computing a Delaunay triangulation of a point set in three dimensional space but less considerations are done about the data structure that describe the relations of the generated tetrahedrals. Such a data structure is proposed here along with a set of operators necessary to make it useful for an implementation of the incremental algorithm of Watson (1981).

1. Introdução

A técnica de subdivisão espacial é utilizada em diversas aplicações na Computação Gráfica. A modelagem de sólidos heterogêneos, a análise de propriedades feita através de elementos finitos e a visualização de sólidos CSG são alguns exemplos destas aplicações.

Diversas estruturas de dados tem sido propostas para a descrição de subdivisões espaciais. Weiler (1986) e Roma (1992) descrevem a estrutura chamada de *radial-edge*. Esta estrutura é formada por listas encadeadas de entidades topológicas formando uma hierarquia: Modelos, Regiões, Cascas (*Shells*), Faces, Laços (*Loops*), Arestas e Vértices. Pode-se dizer que é uma extensão tridimensional da *half-edge* descrita em Mäntylä (1988). É uma estrutura complexa, pois tem-se todas as relações de adjacência de maneira explícita e direta, e o conjunto de operadores que manipulam a estrutura (*operadores de Euler*) se torna bem extenso. Em Bruzzone (1990) temos a descrição de duas estruturas bem mais compactas: *tetrahedral data structure* e *3D symmetric data structure*. A primeira descreve somente subdivisões onde cada região é um tetraedro e a segunda descreve regiões poliedrais de faces triangulares. Porém, a restrição de que as faces sejam triangulares não é muito desejável. Dobkin (1989) descreve a estrutura *facet-edge* que é uma extensão tridimensional da *quad-edge* descrita em Guibas (1985). A *facet-edge* é uma estrutura que descreve qualquer subdivisão poliedral, podendo ser manipulada por um conjunto mínimo de 3 operadores elementares, um de criação de um elemento de aresta e outros dois *splices* (similar ao *splice* em Guibas (1985)). Porém esta estrutura apenas fornece as relações de adjacência entre os elementos de aresta, não mantendo listas de regiões, faces, arestas ou vértices explicitamente (estas redundâncias são úteis, para guardar atributos de regiões, faces, arestas e vértices).

Com a preocupação de definir um conjunto mínimo e suficiente de operadores, Joe (1991) propõe a construção de uma triangulação tridimensional de Delaunay usando transformações locais a partir das combinações possíveis de triangular 5 pontos. Lawson (1986) classifica estas possíveis transformações locais em triangulações de qualquer dimensão.

Os operadores de Euler apresentam uma forma natural de manipular a estrutura e são bem poderosos, porém, algumas vezes não são suficientemente flexíveis. O operador *splice* no caso tridimensional, se divide em dois: *splice-edges* e *splice-faces*. Estes operadores apresentam boa flexibilidade mas sua utilização é pouco natural, sendo necessário o uso de operadores de mais alto nível que utilizem os *splices*.

A estrutura proposta neste trabalho é chamada de *half-face* e é uma generalização tridimensional da *half-edge*, porém simplificada; o que resulta em um subconjunto da *radial-edge*. Assim, a *half-face* é uma estrutura compacta que descreve todas as adjacências da subdivisão, simplificando as aplicações onde é necessário fazer muitos cálculos geométricos, como na visualização e construção de triangulações 3D, em particular na triangulação de Delaunay. Descrevemos também, dois conjuntos de operadores de Euler para a manutenção da estrutura, um geral e um específico para quando as regiões são tetraedros.

Por fim, analisamos as diferenças entre algoritmos 2D e 3D para triangulações de Delaunay e mostramos uma implementação do algoritmo incremental de Watson (1981) para uma *tetraedração* de Delaunay (utilizaremos este termo *tetraedração* mais vezes).

2. Definições e Notações

Os elementos topológicos existentes em uma subdivisão espacial são: vértices, arestas, faces e regiões. Pode-se fazer uma generalização da nomenclatura chamando cada um destes elementos por uma célula-k, onde esta célula-k é um subespaço do R^3 homeomorfo a R^k. Desta forma teremos uma célula-0 como um vértice, uma célula-1 como uma aresta, célula-2 como uma face e célula-3 como uma região.

Dizemos então que uma célula-*k* é descrita por uma coleção finita de células-*(k-1)*. Ou seja, uma região é definida pelo conjunto de suas faces, cada face é definida pela lista de suas arestas e cada aresta é definida por dois vértices. Nota-se que as coleções podem ser ordenáveis ou não. Para descrever as faces, é fundamental manter suas arestas em ordem de percorrimento pela fronteira mantendo sempre o mesmo sentido de orientação.

Para descrever todas estas coleções explicitamente, vamos considerar mais dois elementos: *meia-face* e *uso-de-aresta*. Assim, iremos chamar de *elementos da estrutura* qualquer um dos seis elementos seguintes: região, face, meia-face, aresta, uso-de-aresta e vértice.

A noção de meia-face vem direto do fato de que cada face é adjacente a duas regiões distintas. Assim, desejamos descrever cada região separadamente, como se cada região fosse um sólido, dando uma cópia da face (ou uma meia-face) para cada região. Desta forma, cada meia-face vai conter uma lista de usos-de-aresta. O uso-de-aresta representa que a aresta está sendo utilizada por uma dada região, porém pode estar sendo utilizada por outras *n* regiões. Em 2D, o conceito dado em Mäntylä (1988) é de meia-aresta. Em 3D, a meia-aresta se torna uso-de-aresta e a face se divide em duas meias-faces.

A figura 1 mostra alguns dos ciclos de usos-de-aresta necessários para se descrever uma subdivisão de três regiões: dois cubos e a região externa. As três regiões ficam completamente definidas pelas descrições de suas faces. Olhando-se de fora para dentro de cada região, cada face possui um ciclo de orientação anti-horária de quatro usos-de-aresta. Pode-se notar que cada uso-de-aresta está associado a uma aresta e a um vértice, que é o vértice de origem do vetor que o representa. Assim, cada aresta está associada a pares de usos-de-aresta onde cada par pertence a uma região distinta. Os dois usos-de-aresta de cada par são chamados de *mates*. Por exemplo, u_2 e e_1 são incidentes à mesma aresta e pertencem à mesma região, então u_2 é mate de e_1 e vice-versa. Também u_8 é mate de e_2 e e_3 é mate de e_4. Neste trabalho, o conceito de mate é igual ao conceito usado em Mäntylä (1988). Em Roma (1992), o conceito usado é o que chamamos de *face-mate*, que será descrito mais a frente.

Duas meias-faces são chamadas de *mates* quando estão associadas à mesma face. Na figura 1, As meias-faces hf_1 e hf_2 são mates e cada uma possui seu ciclo de usos-de-aresta. hf_1 possui o ciclo [u_1 u_2 u_3 u_4] e hf_2 possui o ciclo [u_5 u_6 u_7 u_8]. Os usos-de-arestas u_1 e u_5, por exemplo, possuem como aresta associada a mesma aresta a_1, porém pertencem à regiões distintas. Assim, dizemos que u_1 e u_5 são *face-mates*, ou seja u_1 é face-mate de u_5 e vice-versa. Da mesma forma temos e_1 face-mate de e_3 e e_2 face-mate de e_4. Como exemplo, a aresta a_2 possui os pares de usos-de-aresta [u_2, e_1], [e_3, e_4] e [e_2, u_8], cada par pertencendo a uma das regiões.

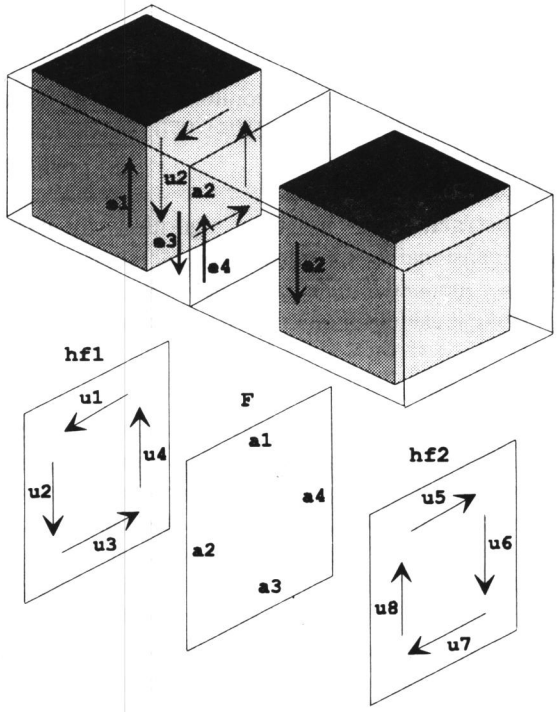


Figura 1 : As meias-faces hf1 e hf2 estão associadas à face F e são *mates*, logo possuem os ciclos de usos-de-aresta com orientações opostas. Olhando de fora para dentro, a orientação é sempre anti-horária.

3. Nomenclatura da Implementação

A figura 2 descreve uma implementação simplificada para os elementos da estrutura. Foi utilizada a linguagem orientada ao objeto C++, usando-se classes que manuseiam listas genéricas, descarregando o tratamento de muitas listas do código da estrutura.

A nomenclatura segue um padrão utilizado em várias bibliotecas de C++ onde cada nome de classe começa com a letra "T", enfatizando que o nome descreve um Tipo de dados. Assim, chamamos de TRegion a classe que representa uma região, TFace para face, THalfFace para meia-face, TEdge para aresta, TEdgeUse para uso-de-aresta e TVertex para vértice.

Como foram utilizados objetos para manutenção de listas genéricas, não aparecem, explicitamente na figura 2, os ponteiros de ligação das listas utilizadas. Porém, cada elemento da estrutura pertence a uma lista duplamente encadeada e pode-se dizer que cada um possui os ponteiros *next* e *prior* que apontam para o próximo elemento e o anterior, formando a lista.

```
class TRegion
 { TList hfList;
 };

class TFace
 { THalfFace *hf1, *hf2;
 };

 class TEdge
 { TEdgeUse *eu;
 };

class TVertex
 { TEdgeUse *eu;
   TPoint3D point;
 };

class THalfFace
 { TRegion   *region;
   TFace     *face;
   TCircList  euList;
 };

class TEdgeUse
 { TVertex   *vertex;
   THalfFace *hf;
   TEdgeUse  *mate;
   TEdge     *edge;
 };

class TSubdivision
 { TList regionList;
   TList faceList;
   TList edgeList;
   TList vertexList;
 };
```

Figura 2 : Uma implementação simplificada em C++ dos *elementos da estrutura* e de TSubdivision, que "manuseia" a subdivisão.

4. Adjacências

Para extrair a informação de adjacência desejada da estrutura, iremos usar funções de percorrimento, usando como elemento básico o uso-de-aresta. Podemos obter todas as relações apenas "caminhando" entre os usos-de-aresta. Isto é garantido pois a cada uso-de-aresta temos somente uma região, uma face, uma aresta e um vértice associado, tornando possível distinguir qualquer conjunto de entes topológicos por um conjunto de usos-de-aresta.

Utilizamos quatro funções básicas para o percorrimento: *next()*, *prior()*, *mate()* e *faceMate()*. Estas funções são implementadas como *funções membro* de TEdgeUse, assim, seu uso é similar ao uso de operadores pós-fixados. Ex.: *e=e->next()->mate()*. Algumas vezes chamaremos estas funções de percorrimento de operadores.

Os operadores next() e prior(), percorrem pela lista de usos-de-arestas da meia-face associada. O next() retorna o próximo uso-de-aresta no sentido anti-horário enquanto o prior() retorna o anterior. O operador mate() retorna o seu uso-de-aresta mate, e o faceMate() retorna seu face-mate.

Pode-se criar outros operadores a partir destes básicos, como por exemplo *radial()* e *rotate()*. O rotate() é o mesmo que mate()->next() e permite "rodar" em torno de um vértice sem mudar de região. O radial() é o mesmo que faceMate()->mate() e permite "rodar" pelos usos-de-aresta que são associados à mesma aresta, mudando a região associada.

O único operador que permite mudar de região é o operador faceMate(). Ele é obtido percorrendo a lista de usos-de-aresta da meia-face mate até encontrar o uso-de-aresta que possui a mesma aresta associada. Os demais operadores são realizados apenas consultando ponteiros já existentes na estrutura.

5. Operadores de Manutenção

Foram implementadas duas classes de operadores, seguindo a linha dos *operadores de Euler* em Mäntylä (1988) e em Bruzzone (1990): os elementares para subdivisões gerais e os específicos para tetraedrações, que são implementados usando os elementares.

Os operadores de construção funcionam colando pedaços de regiões topologicamente consistentes, mas com volume interior vazio, em outras regiões. Então opera-se sobre as faces destas regiões dando à elas a geometria desejada. Cada operador de construção possui seu respectivo operador inverso.

Cada operador possui sua lista de argumentos, mas a maioria deles recebe apenas um uso-de-aresta como parâmetro e devolve um outro uso-de-aresta como valor de retorno, mantendo um padrão entre os operadores.

5.1. Operadores Elementares

As variáveis *r*, *e*, *e1* e *e2* representam um ponteiro para um TEdgeUse e as variáveis *P*, *P1* e *P2* são pontos tridimensionais contendo 3 coordenadas *(x,y,z)*.

O operador **init** inicializa a estrutura criando uma aresta a partir de dois vértices. Seu inverso: **destroy**. Sintaxe : r = init (P1, P2); destroy ();.

O **mev** (*make edge and vertex*) opera em uma face incluindo um vértice e uma aresta. Inverso: **kev** (*kill edge and vertex*). Sintaxe: r = mev(e,P); e = kev(r,P);.

O **mef** (*make edge and face*) cria uma nova face e uma nova aresta. Seu inverso: **kef** (*kill edge and face*). Sintaxe: r = mef (e1, e2); e1 = kef (e1, e2);.

O **mfr** (*make face and region*) cria uma região entre duas regiões já existentes que compartilham uma face. A região criada tem um volume nulo, como se fosse uma folha de papel dobrada ao meio que foi "esprimida" entre duas regiões. Seu inverso: **kfr** (*kill face and region*). Sintaxe : r = mfr (e); e = kfr (r);.

O **mfr2** (*make face and region 2*) é análogo ao operador anterior, porém a região nova é criada entre

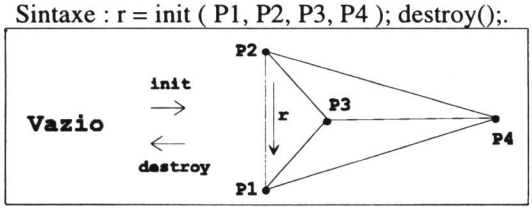
regiões que compartilham duas faces com uma aresta em comum. Seu Inverso: **kfr2** (*kill face and region 2*). Sintaxe: r = mfr2 (e); e = kfr2 (r);.

O **mfr3** (*make face and region 3*) cria uma região nova entre regiões que compartilham três faces com um vértice em comum. Inverso: **kfr3** (*kill face and region 3*). Sintaxe: r = mfr3 (e); e = kfr3 (r);.

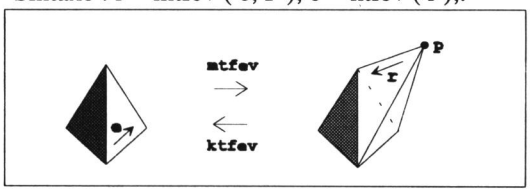
O operador **split** - divide uma região em duas, colocando uma face de divisão de acordo com um conjunto de arestas indicado pelo parâmetro. Seu inverso: **join**. Sintaxe : r = split (e); e = join (r);.

5.2. Operadores para Tetraedrações

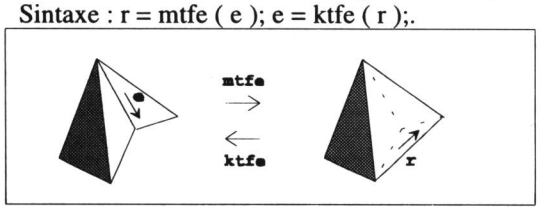
init - inicializa com um tetraedro.
Inverso : **destroy** - deleta toda a estrutura.
Sintaxe : r = init (P1, P2, P3, P4); destroy();.



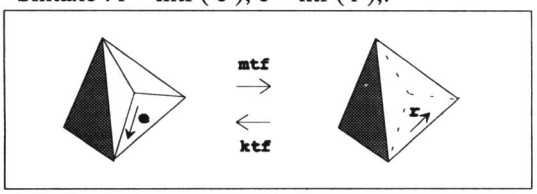
mtfev - *make tetrahedral, faces, edges and vertex.*
Inverso : **ktfev** - *kill tetra., face, edge and vertex.*
Sintaxe : r = mtfev (e, P); e = ktfev (r);.



mtfe - *make tetrahedral, faces and edge.*
Inverso : **ktfe** - *kill tetrahedral, face and edge.*
Sintaxe : r = mtfe (e); e = ktfe (r);.

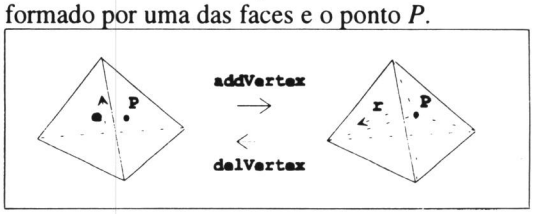


mtf - *make tetrahedral and face.*
Inverso : **ktf** - *kill tetrahedral and face.*
Sintaxe : r = mtf (e); e = ktf (r);.



addVertex - Insere um vértice em tetraedro.
Inverso : **delVertex** - retira vértice inserido.
Sintaxe : r = addVertex (e,P); e = delVertex (r);.

O tetraedro que recebeu o vértice inserido em seu interior se transforma em quatro novos tetraedros, cada um formado por uma das faces e o ponto *P*.



6. O Algoritmo para a Tetraedração de Delaunay

A seguir apresentamos algumas definições para introduzir a tetraedração de Delaunay. Uma discussão mais ampla é feita em Lawson (1986) e Watson (1981).

Definição 1: Dado um conjunto de pontos tridimensionais *S*, dizemos que um tetraedro *T*, cujos vértices são pontos de *S*, satisfaz ao *Critério de Delaunay* quando a esfera cuja fronteira contém os quatro vértices de *T* (a circunsfera de *T*) não contém nenhum ponto de *S* em seu interior.

Definição 2: Dada uma tetraedração *D*, dizemos que *D* é uma *Tetraedração de Delaunay* quando todos os tetraedros de *D* atendem ao critério de Delaunay.

Definição 3: Dado um tetraedro *T*, e um ponto tridimensional *v*, dizemos que *T* e *v* satisfazem o *Teste da Esfera* se a circunsfera de *T* não contém *v*.

Por Dey (1992), os únicos cálculos numéricos realizados no algoritmo de Delaunay são: o produto vetorial para decidir em que lado um ponto está em relação a uma face e o teste da esfera. Estes dois cálculos são baseados em determinantes e a robustez e precisão deles ditam a robustez e precisão do algoritmo.

O algoritmo implementado foi proposto primeiro por Watson (1981) e tem sido usado amplamente para duas e três dimensões: Dado um conjunto *S* de pontos tridimensionais, a tetraedração de Delaunay começa criando um tetraedro inicial que contenha todos os pontos de *S*, que são inseridos incrementalmente à subdivisão, fazendo-se as "modificações locais necessárias". No final, deleta-se o tetraedro criado inicialmente. A criação deste tetraedro inicial é feita para evitar a inserção de novos pontos que estejam fora do fecho convexo da tetraedração corrente.

A dificuldade está nestas "modificações locais necessárias". Uma abordagem simples que se utiliza de um processo recursivo é proposto em Green (1978) para o caso bidimensional: Acha-se o triângulo *T* que contém o ponto *v* a inserir. Criam-se três triângulos novos no lugar de *T*, ligando-se *v* aos vértices de *T*. Então, para cada triângulo novo, aplica-se o teste do círculo com o vértice oposto do triângulo adjacente. Se o teste falhar, é feita uma troca de arestas internas (um *swap*) e o teste

continua a ser feito recursivamente até que seja aceito. A figura 3 ilustra este algoritmo.

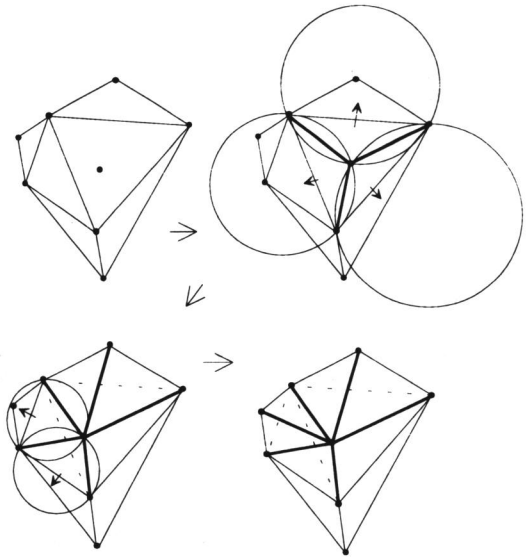


Figura 3 : Ilustração do algoritmo recursivo para o caso bidimensional.

Porém, o caso tridimensional não pode ser tratado analogamente. Este *swap* 2D de arestas, em 3D, fica sendo a transformação de dois tetraedros com uma face em comum em três que compartilham uma mesma aresta, fazendo uma troca de face por aresta.

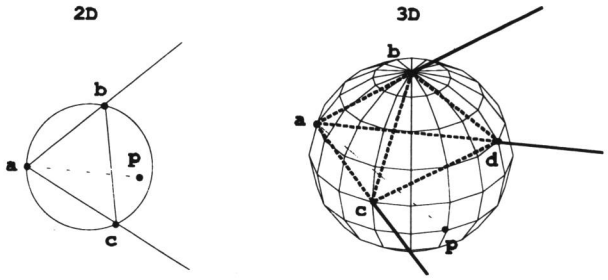


Figura 4 : Em 2D, sempre que o teste do círculo falha para um ponto p é possível trocar a aresta bc pela ap, pois ap está parcialmente contida no triângulo abc. Já no caso tridimensional, o teste da esfera pode falhar para um ponto p tal que ap esteja totalmente fora do tetraedro $abcd$.

Então é possível obter uma configuração em que o teste da esfera falha ao mesmo tempo em que não é possível realizar a troca face-aresta. A figura 4 exemplifica esta ocasião.

Assim, em 3D, este processo não apresenta a simplicidade esperada de operar basicamente com um único operador de troca de arestas e então o problema é abordado de outra forma.

Quando um novo ponto v é inserido, todos os tetraedros cuja circunsfera contenha v são deletados, criando um poliedro estrelado P que contém v em seu núcleo. Assim, basta ligar v a todos os vértices de P para formar a tetraedração atendendo ao critério de Delaunay. A figura 5 ilustra este processo para o caso 2D, sua generalização 3D é totalmente análoga.

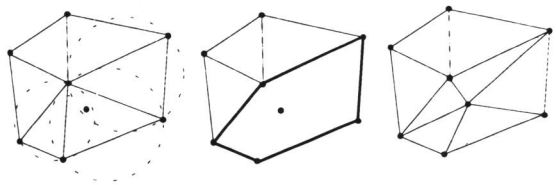


Figura 5 : Ilustração do processo de inserção de pontos novos no caso bidimensional.

7. Implementação

A implementação do algoritmo foi feita criando-se uma nova *classe* descendente de TSubdivision chamada de TTetra. Esta nova classe contém as funções membro necessárias para a manutenção de tetraedrações, como os operadores específicos e as funções para a construção da tetraedração de Delaunay.

A função principal é a função insertPoint() de inserção de pontos novos. Ela se divide em três partes: 1°-encontra o tetraedro que contém o novo ponto. 2°-cria o poliedro estrelado, retirando todos os tetraedros que não satisfazem o critério de Delaunay. 3°-liga todos os vértices do poliedro estrelado ao novo ponto, criando os novos tetraedros de Delaunay.

Antes de analisar o pseudo-código da figura 6, vejamos as seguintes definições.

Definição 4 : Seja T uma triangulação de pontos no plano. Dizemos que os triângulos de T são *unicamente percorridos* quando visitamos todos os triângulos de T somente uma vez e sempre que um dado triângulo t for visitado, ou t é o primeiro triângulo visitado ou existe um outro triângulo da triangulação já visitado t', onde t' e t possuem uma aresta em comum.

Definição 5 : Seja T uma triangulação no plano. Dizemos que um percorrimento único dos triângulos de T é do tipo que *não fecha ciclos* se, a cada visita do percorrimento único, o conjunto formado pela união de todos os triângulos já visitados é um conjunto simplesmente conexo (não contém "buracos").

Em diversas ocasiões é interessante fazer um percorrimento único por face triangulares sem fechar ciclos. Em nosso algoritmo de inserção de pontos na tetraedração usaremos este percorrimento duas vezes. Uma na função joinTetra(), que cria o poliedro estrelado e outra na função constructTetra(), que reconstrói os tetraedros dentro do poliedro estrelado.

```
void TTetra::insertPoint ( TPoint3D p )
{
 // Encontra região que contém o p
 r = findRegion ( p );

 // Cria poliedro estrelado por Delaunay
 enfileira as meias-faces da região r;
 while ( fila não vazia )
 { f = meia-face da frente da fila;
   if ( f não é meia-face da fronteira e
        falha teste da esfera de f->mate() e p )
     joinTetra ( uso-de-aresta de f );
 }

 // Reconstrói ligando as faces ao ponto p
 e = um uso-de-aresta do poliedro estrelado;
 mtfev ( e, p );  // Cria primeiro tetraedro
 enfileira meias-faces adjacentes à face de e;
 while ( fila não vazia )
 { f = meia-face da frente da fila;
   constructTetra ( uso-de-aresta de f );
 }
}

void TTetra::joinTetra ( TEdgeUse *e )
{
 if ( uma face em comum entre as regiões )
  { if ( face de e fecha um ciclo )
      { re-enfilera meia-face de e; retorna; }
    join (e);
    enfileira meias-faces adjac. à face de e;
  }
 if ( duas faces em comum )
  { retira da fila referências às faces;
    u = uso-de-aresta adjacente às duas faces.
    kef ( u->next(), u->mate()->next() );
    join (e);
    enfileira as duas outras meias-faces;
  }
}

void TTetra::constructTetra ( TEdgeUse *e )
{
 if ( meia-face de e está marcada ) retorna;

 // Escolhe o que fazer de acordo com quantos
 // usos-de-arestas em volta da meia-face
 // de e estão marcados.

 if ( os três usos-de-aresta estão marcados )
   marca face de e;
 if ( dois usos-de-arestas marcados )
  { u = uso-de-aresta não marcado;
    marca face de e e aresta de u;
    enfileira meia-face de u->mate();
    mtf ( u->next()->faceMate() );
  }
 if ( um uso-de-aresta marcado )
  { if ( fecha um ciclo )
      { re-enfilera meia-face de e; retorna; }
    u1, u2 = usos-de-arestas não marcados;
    marca face de e e arestas de u1 e u2;
    enfileira meia-face de u1-mate();
    enfileira meia-face de u2->mate();
    mtfe ( e->faceMate() );
  }
}
```

Figura 6 : Pseudo-código resumido das funções principais.

A figura 7 exemplifica um percorrimento que não fecha ciclos e um outro que fecha ciclos.

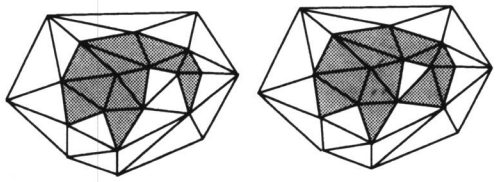


Figura 7 : Nos dois casos, os triângulos pintados representam triângulos já visitados de um percorrimento único, porém, somente o percorrimento da esquerda não fecha ciclos.

A função joinTetra() vai unindo o tetraedro que contém o ponto a inserir com os tetraedros vizinhos, formando o poliedro estrelado. Para isso usa o operador join(). Existem dois casos a considerar. O caso em que existe somente uma face comum entre o poliedro e o tetraedro a unir é tratado simplesmente com a utilização do operador join() e então as outras três faces que restaram do tetraedro que foi unido são enfileiradas dando continuidade ao processo. Porém, deve-se tomar cuidado para que, ao unir este tetraedro não se forme um ciclo no percorrimento das faces do poliedro, o que causaria uma inconsistência.

A maneira de se detectar se o percorrimento vai fechar um ciclo ou não é feita através de marcações nas faces triangulares e nos vértices percorridos. A figura 8 mostra como evitar os ciclos a partir das marcações de vértice e faces percorridas.

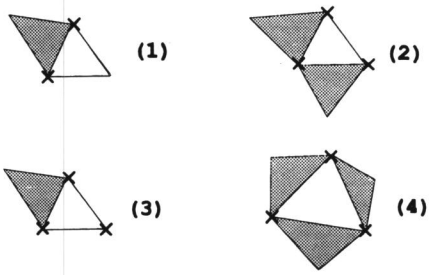


Figura 8 : Nos casos (1) e (2) o triângulo não hachurado pode ser visitado que nenhum ciclo se formará. O caso (3) é quando não se pode visitar o triângulo pois o vértice oposto à aresta da face marcada está marcado. Evitando que o caso (3) ocorra, nunca obteremos um triângulo a visitar como no caso (4), a não ser que este seja o último triângulo à visitar, isto considerando que os pontos da triangulação estão sobre a casca de uma esfera.

Assim, o percorrimento sem formação de ciclos pode ser realizado da seguinte forma: começa-se com um triângulo inicial qualquer. Ele é visitado, marcado e seus três vértices também são marcados. Os três

triângulos adjacentes por aresta ao inicial são enfileirados. Agora começa-se a retirar os triângulos do início da fila até a fila ficar vazia. Para cada triângulo retirado, verificamos se ele se enquadra no caso (3) da figura 8. Caso afirmativo, não visitamos este triângulo, apenas o colocamos no final da fila para ser retirado em outra ocasião em que não ocorra o caso (3).

Ao desistir de visitar os triângulos que se enquadram no caso (3), impedimos que se formem duas componentes conexas no conjunto de triângulos não visitados, fato que significa que está se formando um ciclo. Pode-se ver que, para cada triângulo rejeitado t, haverá um conjunto de triângulos, que, quando forem visitados, mudarão as marcações de t, tornando-o visitável. E sempre haverá algum triângulo possível de ser retirado da fila, garantindo que a fila se esvaziará e que todos os triângulos serão visitados.O caso em que o tetraedro a unir possui duas faces em comum com o poliedro, apresenta duas considerações: uma é que, para unir as regiões precisamos, antes de usar o operador join(), usar o operador kef() para transformar as duas faces em uma. A outra consideração é que temos que checar na fila se existe alguma referência à alguma das faces deletadas. Como está sendo feito um percorrimento único, a face associada ao uso-de-aresta passado como parâmetro não precisa ser checada na fila, mas a outra face deletada precisa. Se for encontrada na fila, teremos que retirar este elemento pois, senão, quando esta referência de face for retirada da fila em outra ocasião, teremos um ponteiro que aponta para uma face já deletada. Para resolver o caso pode-se fazer uma busca linear na fila, já que esta não cresce muito. Uma solução mais rápida, porém que gasta mais memória, seria incluir nas meias-faces ponteiros para as suas referências na fila.

A função constructTetra() é chamada para construir os tetraedros formados ligando-se o novo ponto inserido aos vértices do poliedro estrelado resultante do processo em joinTetra(). Primeiro cria-se um tetraedro inicial, usando-se mtfev(), e enfileira-se as suas três meias-faces associadas às faces comuns ao poliedro estrelado. Então vai-se retirando da fila as meias-faces até a fila ficar vazia. Em seguida, constructTetra() pega cada meia-face e verifica quantas arestas desta meia-face já foram visitadas. Se apenas uma aresta, checa se ao visitar esta meia-face algum ciclo será formado e, se não, cria o tetraedro correspondente à meia-face com o operador mtfe(). Se duas arestas foram visitadas, usa-se o operador mtf(). Se as três arestas foram visitadas, não é preciso fazer nada pois esta é a última face do percorrimento e então o poliedro estrelado já foi reduzido a um tetraedro e nada é feito, apenas marcamos a meia-face como visitada e a fila naturalmente se esvaziará terminando o processo.

Desta forma, pode-se notar que todo o processo feito em insertPoint() é de caráter local e a complexidade não depende do número de pontos da tetraedração, e sim do número de vezes que o teste da esfera falha. Assim, pode-se dizer que a cada ponto inserido, o tempo gasto é aproximadamente constante, isto sem levar em conta o tempo que a função findRegion() pode gastar.

Esta função percorre os tetraedros da subdivisão até encontrar um que contenha o ponto p a inserir. A procura pelo tetraedro é feita da seguinte forma: começa-se com um tetraedro inicial t qualquer. Para cada face de t, verifica-se a posição relativa de p. Se p estiver do lado de dentro de t com relação à todas as suas faces então o tetraedro contém p. Se não, passa-se a um tetraedro vizinho t', adjacente por uma face que indique p do lado de fora de t e então repete-se o processo fazendo $t=t'$.

Para pontos distribuídos uniformemente em um cubo, este processo é bem eficiente e o processo total de inserção de pontos fica constante, levando-nos à complexidade de $O(n)$ para tetraedrar n pontos.

Porém, para o pior caso, findRegion() pode gastar um tempo linear. Se tivermos uma subdivisão alongada de maneira que começamos o processo pelo extremo oposto ao tetraedro que procuramos, iremos percorrer praticamente toda a estrutura. Imagine que temos uma tetraedração de pontos que foram gerados na casca de um "tubo" fino e alongado, de forma que começamos a procurar por um ponto que está em um tetraedro do final do tubo por um tetraedro do início do tubo. A figura 9 ilustra esta circunstância no caso bidimensional, que é inteiramente análogo.

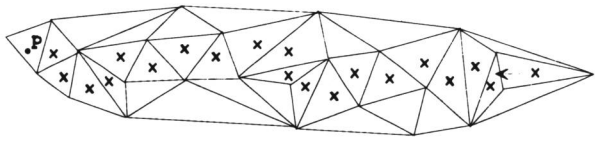


Figura 9 : Exemplo de uma triangulação bidimensional de pontos em que a procura pelo triângulo que contém o ponto p pode gastar tempo linear, dependendo do triângulo inicial da busca.

Desta forma, se findRegion pode levar $O(n)$, podemos chegar a um tempo $O(n^2)$ para a tetraedração total, que é o tempo ótimo para o pior caso de uma tetraedração de Delaunay, já que pode-se ter configurações de n pontos que geram $O(n^2)$ tetraedros de Delaunay, como no caso da tetraedração dos pontos mostrados na figura 10.

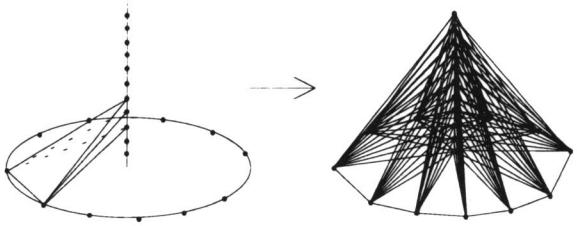


Figura 10 : Uma configuração de pontos que gera $O(n^2)$ tetraedros em uma tetraedração de Delaunay. Metade dos pontos estão na reta vertical e metade sobre o círculo da base.

8. Um Visualizador para Subdivisões 3D

Outro aspecto importante a considerar é como visualizar uma subdivisão tridimensional. Uma triangulação planar é trivialmente visualizada identificando o plano da triangulação com a tela do monitor do computador. Mas como fazer em três dimensões, sem dispor de periféricos especiais?

A maneira encontrada foi fazer um visualizador que permite "caminhar" entre os tetraedros pelas adjacências de faces. Desta forma, pode-se marcar um conjunto determinado de tetraedros para permitir uma visualização localizada, ressaltando os tetraedros marcados dos demais através da utilização de cores.

Esta visualização parcial se torna indispensável, pois já com um conjunto pequeno de tetraedros desenhados em aramado, não conseguimos identificar suas ligações corretamente devido à confusão gerada pelo número excessivo de linhas.

9. Conclusões

A técnica de subdivisão espacial, principalmente a subdivisão em tetraedros, é muito utilizada. A tetraedração de Delaunay se destaca dentre as demais por manter a boa forma dos tetraedros gerados. Ela é utilizada em diversas aplicações, como na reconstrução de formas tridimensionais em Boissonat (1984).

Visando atender diretamente estas aplicações, propomos neste trabalho uma estrutura topológica para descrever subdivisões espaciais que busca ser simples e concisa em sua definição, porém completa na descrição das subdivisões representadas.

A estrutura proposta é mais completa do que as mostradas em Bruzzone (1990) e Dobkin (1989) e é menos extensa do que estruturas mais gerais, como a de Weiler (1986) e Roma (1992), porém, continua descrevendo todas as relações de adjacência necessárias para simplificar os diversos cálculos geométricos.

A estrutura não exige que as faces sejam triangulares nem que as regiões sejam tetraedros, não restringindo qualquer aplicação. Durante o algoritmo implementado, por exemplo, é necessário criarmos uma região que não é um tetraedro.

O programa que gera e visualiza a tetraedração foi implementado em um IBM-PC e consegue manipular subdivisões de, em média, 500 tetraedros apenas com a memória de 640 Kbytes.

10. Referências

Boissonat J. D. (1984), "Geometric Structures for Three-Dimensional Shape Representation". ACM Transaction on Graphics, 3, 266-286.

Bruzzone, E. e De Floriani, L. (1990), "Two Data Structures for Building Tetrahedralizations". The Visual Computer, 6, 266-283.

Dey, T. K.; Sugihara, K. e Bajaj, C. L. (1992), "Delaunay Triangulations in Three Dimensions with Finite Precision Arithmetic". Computer Aided Geometric Design, 9, 457-470.

Dobkin, D. P. e Laszlo M. J. (1989), "Primitives for the Manipulation of Three-Dimensional Subdivisions". Algorithmica, 4, 3-32.

Green, P. J. e Sibson, R. (1978), "Computing Dirichlet Tesselations in the Plane". The Computer Journal, 21, 168-173.

Guibas, L. e Stolfi J. (1985), "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams". ACM Transaction on Graphics, 4, 75-123.

Joe, B. (1991), "Construction of Three-Dimensional Delaunay Triangulations Using Local Transformations". Computer Aided Geometric Design, 8, 123-142.

Lawson, C. L. (1986), "Properties of n-Dimensional Triangulations". Computer Aided Geometric Design, 3, 231-246.

Mäntylä, M. (1988), "An Introduction to Solid Modeling". Computer Science Press, Maryland, 1988.

Roma, P. C. (1992), "Criação e Manutenção de Subdivisões do Espaço". Tese de Doutorado, PUC-Rio, Departamento de Informática.

Watson, D. F. (1981), "Computing the n-Dimensional Delaunay Tessellation with Application to Voronoi Polytopes". The Computer Journal, 24, 167-172.

Weiler, K. (1986), "Topological Structures for Geometric Modeling". PhD Thesis, Rensselaer Polytecnic Institute, Troy, NY.