5
A Estrutura de Dados CHF

Neste capítulo descreveremos as características dos quatro níveis da estrutura de dados CHF. Abordaremos também a construção de cada um deles a partir dos níveis anteriores.

5.1
Nível 0: Sopa de Tetraedros

O primeiro nível proposto para a composição da CHF é o *nível* 0. Nele armazenamos a menor quantidade de informação possível para a representação de uma malha de tetraedros. Apenas as informações geométricas e os tetraedros da malha são armazenados. Sua principal aplicação é a visualização da malha e assim, nenhuma informação que diz respeito à vizinhança dos tetraedros é armazenada. Podemos imaginá-lo como uma "sopa" de tetraedros (figura 5.1).

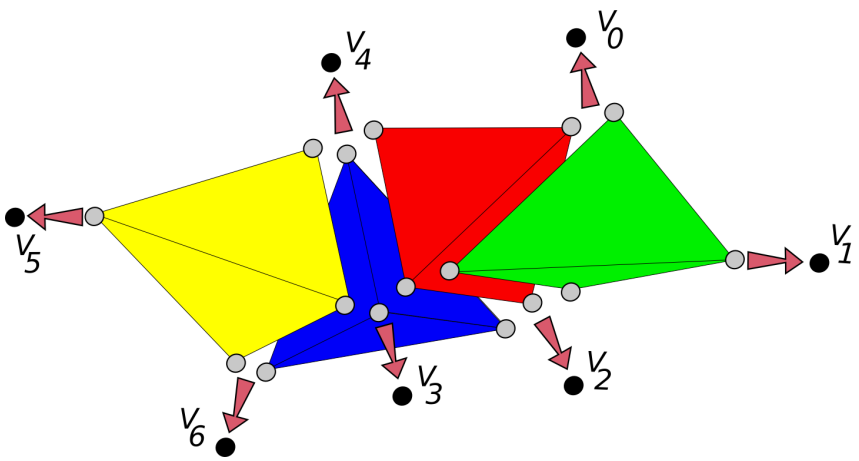

Figura 5.1: Nível 0 da CHF: Sopa de Tetraedros.

Utilizaremos o conceito de *half–face* (Lopes and Tavares 1997) para acessar os elementos de um tetraedro.

Definição 5.1 Half–face: Uma *half–face* é a associação de um tetraedro a um dos seus triângulos de bordo ou, equivalentemente, a associação deste triângulo ao vértice oposto em um tetraedro (veja figura 5.2).

Na estrutura de dados CHF, os vértices, as *half–faces* e os tetraedros são indexados por inteiros não–negativos. Cada tetraedro é representado por quatro *half-faces* consecutivas que definem sua orientação. Por exemplo, as *half–faces* 0, 1, 2 e 3 correspondem ao primeiro tetraedro, as *half–faces* 4, 5, 6 e 7 pertencem ao segundo tetraedro, e assim por diante.

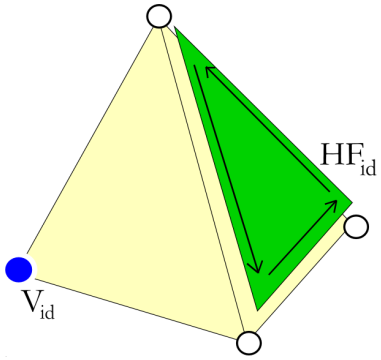

Figura 5.2: Uma *half–face* de um tetraedro.

Contêiner Geometry: As informações geométricas existentes na CHF ficam armazenadas no contêiner do tipo vector chamado Geometry, denotado por G[]. Para cada vértice do modelo, guardamos suas coordenadas, vetor normal e outros atributos geométricos convenientes para a aplicação. Assim, se quisermos acessar a geometria de um vértice de índice V_{id} v devemos buscá–la através do contêiner G[] acessando a posição G[v]. O contêiner G[] tem tamanho $\mathbf{n}_0$.

Regras para Half–Faces: Uma *half–face* de índice HF_{id} hf pertence ao tetraedro $\lfloor \mathsf{hf}/4 \rfloor$. Ainda o índice das quatro *half–faces* que pertencem ao tetraedro de índice T_{id} t são $4\mathsf{t}, 4\mathsf{t}+1, 4\mathsf{t}+2$, e $4\mathsf{t}+3$. As *half–faces next*, *middle* e *previous* de uma dada *half–face* HF_{id} hf no tetraedro $\lfloor \mathsf{hf}/4 \rfloor$ são definidas como:

$$\begin{aligned} \mathrm{next}_{\mathsf{hf}}(\mathsf{hf}) &:= 4\lfloor \mathsf{hf}/4 \rfloor + (\mathsf{hf}+1)\%4, \\ \mathrm{mid}_{\mathsf{hf}}(\mathsf{hf}) &:= 4\lfloor \mathsf{hf}/4 \rfloor + (\mathsf{hf}+2)\%4, \\ \mathrm{prev}_{\mathsf{hf}}(\mathsf{hf}) &:= 4\lfloor \mathsf{hf}/4 \rfloor + (\mathsf{hf}+3)\%4. \end{aligned}$$

Note que as operações aritméticas acima podem ser escritas eficientemente como operações binárias: $4\mathsf{t} := \mathsf{t} \ll 2$, $\lfloor \mathsf{hf}/4 \rfloor := \mathsf{hf} \gg 2$, $\mathsf{hf}\%4 := \mathsf{hf}\&3$ e $4\lfloor \mathsf{hf}/4 \rfloor := \mathsf{hf}\&(\sim 3)$. Logo as expressões das *half–face next*, *middle* e *previous* podem ser reescritas na forma:

$$\begin{aligned} \mathrm{next}_{\mathsf{hf}}(\mathsf{hf}) &:= \mathsf{hf}\&(\sim 3) \mid (\mathsf{hf}+1)\&3, \\ \mathrm{mid}_{\mathsf{hf}}(\mathsf{hf}) &:= \mathsf{hf}\&(\sim 3) \mid (\mathsf{hf}+2)\&3, \\ \mathrm{prev}_{\mathsf{hf}}(\mathsf{hf}) &:= \mathsf{hf}\&(\sim 3) \mid (\mathsf{hf}+3)\&3. \end{aligned}$$

Contêiner Vertex: Na CHF cada *half–face* HF_{id} hf é associada ao vértice oposto a *half–face* no tetraedro incidente. Tal associação é armazenada em um contêiner de inteiros do tipo vector, chamado Vertex e denotado por V[]. Assim, o número inteiro v = V[hf] é o índice do vértice oposto a hf (figura 5.3). O tamanho de V[] é igual a $4\mathbf{n}_3$, e cada entrada de V[] varia de 0 a $\mathbf{n}_0 - 1$.

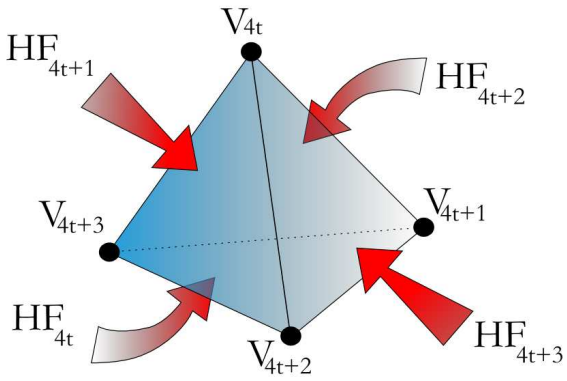

Figura 5.3: Relação entre *half–faces* e vertices.

A tabela 5.1 e a figura 5.4 definem a orientação de cada *half–face* induzidas pela orientação do tetraedro de índice t.

Half–Face	< Orientação >
HF_{id} 4t	< V[4t + 1], V[4t + 2], V[4t + 3] >
HF_{id} 4t + 1	< V[4t + 2], V[4t] , V[4t + 3] >
HF_{id} 4t + 2	< V[4t + 3], V[4t] , V[4t + 1] >
HF_{id} 4t + 3	< V[4t] , V[4t + 2], V[4t + 1] >

Tabela 5.1: Orientação das *half–faces* de um tetraedro T_{id} t.

Half–Edges: Assim como na estrutura de dados *Handle–Face*, cada *half–face* da CHF é limitada por um ciclo de três *half–edges* (figura 5.2) implicitamente representadas.

Definição 5.2 Half–edge: Uma *half–edge* da CHF é a associação de uma *half–face* e um de seus vértices.

A orientação do ciclo de *half–edges* de uma *half–face* HF_{id} hf é induzida a partir da própria orientação da *half–face* hf. Logo a partir da tabela 5.1 e da figura 5.4 podemos também descobrir todos os ciclo de *half–edges* de um tetraedro de índice T_{id} t.

Por exemplo, o ciclo de *half–edges* na *half–face* de índice HF_{id} 4t é dado por V[4t + 1] → V[4t + 2] , V[4t + 2] → V[4t + 3], e V[4t + 3] → V[4t + 1], o ciclo de *half–edges* da *half–face* HF_{id} 4t + 1 é V[4t + 2] → V[4t] , V[4t] → V[4t + 3], e V[4t + 3] → V[4t + 2] e assim por diante.

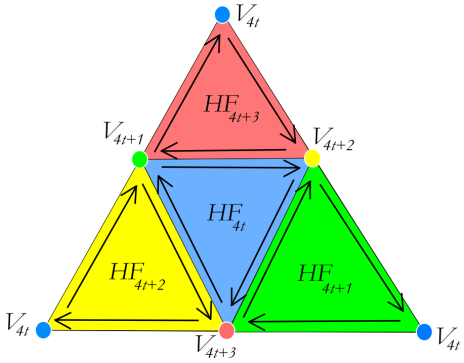

Figura 5.4: Orientação das *half–faces* de um tetraedro T_{id} t.

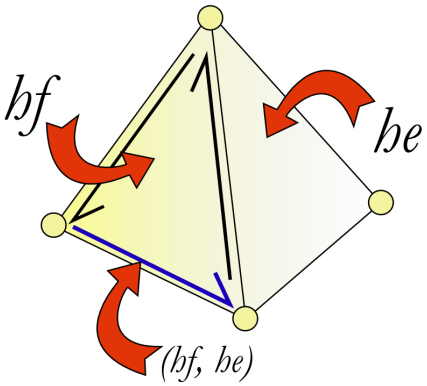

Figura 5.5: Índice de uma *half–edge*.

Identificamos uma *half–edge* de uma *half–face* HF_{id} hf através do par (hf, he), onde he é o índice da *half–face* oposta ao vértice inicial da *half–edge* (figura 5.5). Podemos obter as *half–edges* anterior e posterior (chamadas de *next* e *previous* respectivamente) de uma *half–edge* com índice (hf, he), no ciclo de *half–edges* da *half–face* hf, através das seguintes regras:

$$\begin{aligned}\text{next}_{\mathsf{he}}(\mathsf{hf},\mathsf{he}) &:= (\mathsf{hf}, \mathsf{hf}\&(\sim 3)|N[\mathsf{he}\%4][\mathsf{hf}\%4]),\\ \text{prev}_{\mathsf{he}}(\mathsf{hf},\mathsf{he}) &:= (\mathsf{hf}, \mathsf{hf}\&(\sim 3)|P[\mathsf{he}\%4][\mathsf{hf}\%4]).\end{aligned}$$

$$onde : N = \begin{bmatrix} - & 3 & 1 & 2 \\ 2 & - & 3 & 0 \\ 3 & 0 & - & 1 \\ 1 & 2 & 0 & - \end{bmatrix} \; e \; P = N^t.$$

Duas outras *half–edges* especiais, úteis para as operações R_{1*}, podem ser definidas: A *half–edge mate* de uma *half–edge* (hf, he), pertence a *half–face* oposta ao vértice inicial de $\text{prev}_{\mathsf{he}}(\mathsf{hf},\mathsf{he})$ (figura 5.6), já a *half–edge radial* pertence à *half–face oposta* a hf, que será definida no nível 1 (figura 5.7). Tais *half–edges* podem ser obtidas da seguinte forma:

$$\begin{aligned}\text{mate}_{\mathsf{he}}(\mathsf{hf},\mathsf{he}) &= (\text{prev}_{\mathsf{he}}(\mathsf{hf},\mathsf{he}), \text{next}_{\mathsf{he}}(\mathsf{hf},\mathsf{he})),\\ \text{radial}_{\mathsf{he}}(\mathsf{hf},\mathsf{he}) &= (\quad \mathsf{O}[\mathsf{hf}] \quad , \text{next}_{\mathsf{he}}(\mathsf{hf},\mathsf{he})).\end{aligned}$$

Com as quatro regras descritas acima, podemos escrever algoritmos similares aos propostos por (Weiler 1986) para percorrer todas as *half–faces* em torno de uma aresta da malha, permitindo desta forma implementações eficientes para o cálculo da estrela de uma aresta. Tais algoritmos serão descritos com detalhes na seção 5.6.

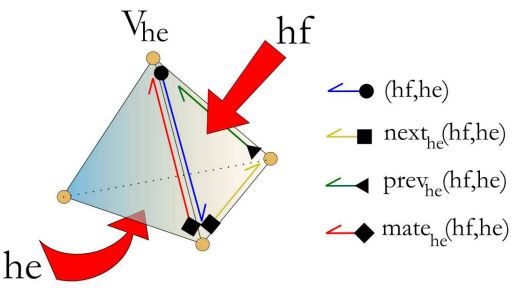

Figura 5.6: *Half–edge mate*

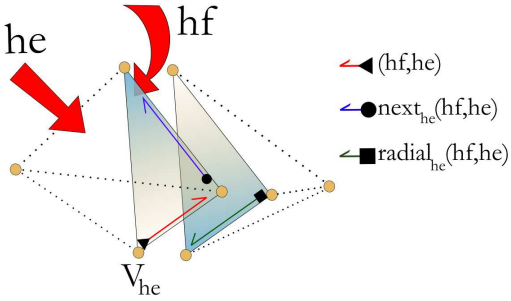

Figura 5.7: *Half–edge radial*

5.2
Nível 1: Adjacência entre Tetraedros

Para muitas aplicações é importante que saibamos percorrer os tetraedros da malha eficientemente, e para que isso seja possível, o *nível* 1 da CHF adiciona a estrutura informações sobre a vizinhança de cada tetraedro, ou seja, deixamos de tratar a malha como uma "sopa" de tetraedros e passamos a vê–la como um conjunto de tetraedros conectados (figura 5.8).

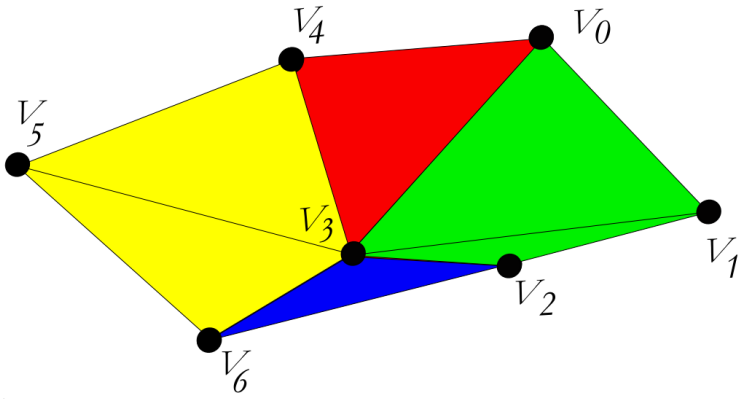

Figura 5.8: Nível 1: Adjacência entre Tetraedros.

Como estamos trabalhando com 3–variedades, cada *half–face* da malha é incidente a um ou dois tetraedros. As faces de uma 3–variedades terão duas *half–faces* incidentes quando forem de interior e apenas uma quando forem de bordo.

A informação de adjacência entre dois tetraedros vizinhos é implicitamente representada através de duas *half–faces* incidentes a uma mesma face. Em outras palavras representaremos a adjacência entre tetraedros através das *half–faces opostas*, definidas a seguir:

Definição 5.3 Half–face oposta: Uma *half–face* é *oposta* a *half–face* HF_{id} hf quando tem os mesmos vértices de hf mas orientação oposta, e por isso compartilham a mesma face.

Adicionamos a CHF um contêiner do tipo vector chamado Opposite, denotado por O[], que associa para cada *half–face*, sua *half–face* oposta (5.9).

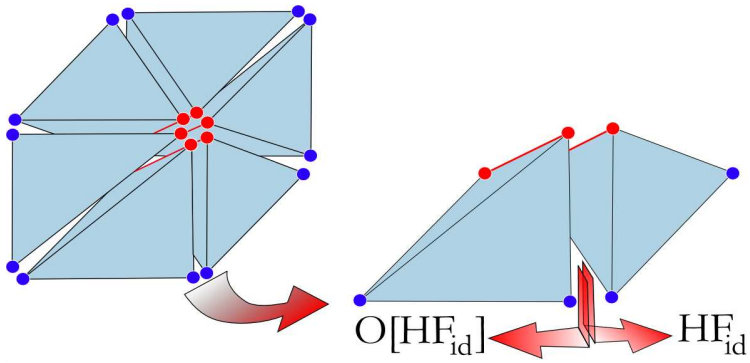

Figura 5.9: Half–face oposta a half-face HF_{id}.

Se uma *half–face* HF_{id} hf está no bordo da 3–variedade, ela não terá uma *half–face* oposta, e codificaremos seu oposto por O[hf] $= -1$. Com isso, o valor armazenado em O[hf] nos proporciona uma forma direta de classificação de uma *half–face* entre *half–edge* de bordo e de interior. O tamanho do contêiner O[] é $4\mathbf{n}_3$, e suas entradas variam de -1 até $4\mathbf{n}_3 - 1$.

O algoritmo 11 mostra uma maneira eficiente de construirmos o contêiner O[] a partir de V[]. Utilizaremos um contêiner genérico do tipo map para auxiliar a construção do container O[].

Algorithm 11 Construção do Contêiner Opposite

```
1:  O[i] ← −1 {Inits the container}
2:  map{V_id × V_id × V_id → HF_id} adjacency
3:  for HF_id hf ∈ {0 ... 4n_3 − 1} do
4:     {Gets the vertices tuple of hf}
       v_0 ← V[next_hf(hf)] ; v_1 ← V[mid_hf(hf)] ;
       v_2 ← V[prev_hf(hf)] ; v^3 ← sort(v_0, v_1, v_2)
5:     if adjacency.find(v^3) then
6:        O[hf] ← adjacency [v^3]
7:        O[O[hf]] ← hf {Found opposite half–face}
8:        adjacency.erase(v^3)
9:     else {Temporarily stores half–face}
10:       adjacency [v^3] ← hf
11:    end if
12: end for
```

5.3 Nível 2: Representação das Células

Como observamos em 4.3, é extremamente útil que tenhamos uma representação explicita de cada célula do complexo simplicial. Com esse objetivo, no *nível* 2 da CHF adicionamos à estrutura uma representação explícita para as células da malha (figura 5.10).

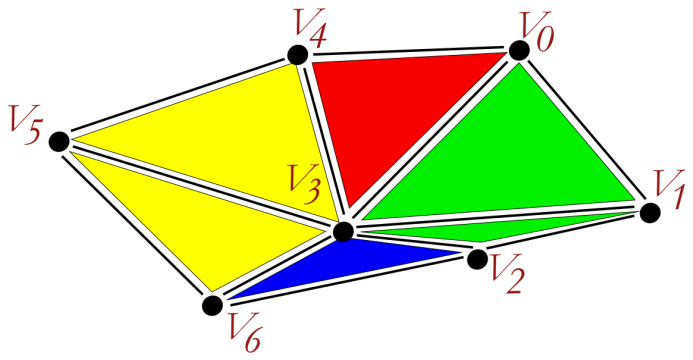

Figura 5.10: Nível 2: Representação das Células.

Tal representação é feita pro meio de três novos contêineres: Vertex Half–Face, Edge Map e Face Map, que serão descritos com mais detalhes a seguir.

Face Map: Identificaremos as faces da malha através de suas *half–faces* incidentes. Uma face de interior será representada por sua *half–face* de menor índice $F_{id} := HF_{id}$, uma vez que a outra pode ser obtida através do contêiner O[]. Uma face de bordo será identificada através de sua única *half–face*.

Podemos relacionar o índice de cada face da malha a atributos usando o contêiner associativo do tipo map Face Map, denotado por FH[].

É importante observar que os vértices das faces podem ser facilmente acessados através da *half-face* que a identifica. A map FH[] tem tamanho $\mathbf{n}_2$ e pode ser alocada em tempo $O(\mathbf{n}_2 log(\mathbf{n}_2))$ usando o algoritmo 12, já que como vimos em 2.8 podemos recuperar um objeto armazenado em um contêiner do tipo map com $\mathbf{n}_2$ objetos em tempo $O(log(\mathbf{n}_2))$.

Algorithm 12 Construção da Face Map

```
1: map{F_id → property} FH
2: for HF_id hf ∈ {0...4n_3 − 1} do
3:    {Gets the opposite of hf}
      hf_0 ← hf ; hf_1 ← O[hf] ;
      h ← min(hf_0, hf_1) ≥ 0
4:    if HE.find(h) then
5:       continue; {Face already found}
6:    else {Stores the face}
7:       FH[h] ← property
8:    end if
9: end for
```

Podemos otimizar a representação das faces se dividirmos a Face Map em duas maps diferentes: a primeira com as faces de bordo, e a segunda com as faces de interior. Desta maneira, a procura por faces em operações no bordo da variedade ganham em eficiência já quem em um contêiner map, buscas são

realizada em tempo $O(log(N))$, e neste caso N é o número de faces do modelo.

Edge Map: Identificaremos uma aresta por um par ordenado de inteiros $E_{id} :=< \mathsf{v}_1, \mathsf{v}_2 >$, onde $\mathsf{v}_1 < \mathsf{v}_2$ são os índices dos vértices da aresta. Desta maneira, obtemos uma representação única para cada aresta do modelo, independente de orientação.

Através de um novo contêiner do tipo map chamado Edge Map, denotado por EH[], relacionamos cada aresta às suas propriedades. Ainda, para cada aresta, armazenamos o índice de uma de suas *half–faces* incidentes. Se a aresta é uma aresta de bordo, a *half–face* armazenada será a *half–face* de bordo que tem a aresta orientada de v_1 para v_2 (veja figura 5.11). Assim ganhamos uma maneira direta de classificar as arestas como arestas de interior ou arestas de bordo.

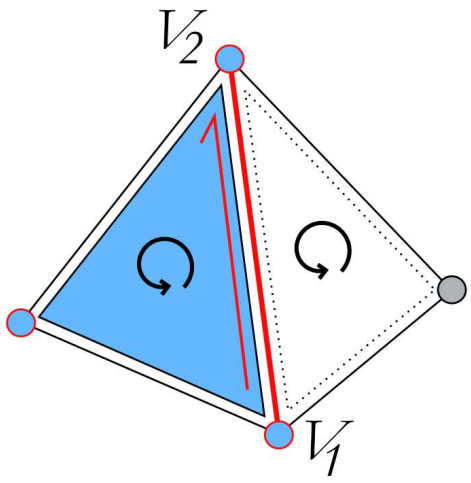

Figura 5.11: Half–face de bordo incidente a uma aresta de bordo.

A map EH[] tem $\mathbf{n}_1$ entradas que podem ser alocadas em tempo $O(\mathbf{n}_2 log(\mathbf{n}_2))$ usando o algoritmo 13.

Algorithm 13 Construção da Edge Map

```
1: map{V_id × V_id → HF_id} EH
2: for T_id t ∈ {0 ... n_3 − 1} do
3:   for v^2 ∈ {(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)} do
4:     a ← (4t | v^2[0]), b ← (4t | v^2[1])
5:     v^2 ← (V[a], V[b]), sort(v^2) {Gets and sorts the edges' vertices.}
6:     if EH.find(v^2) = EH.end() then
7:       Gets one incident half–face (a boundary one if avaiable).
8:     else if O[EH[v^2]] ≠ −1 then
9:       Tries to change the current half–face to a boundary one .
10:    end if
11:  end for
12: end for
```

Também podemos otimizar a representação das arestas se dividirmos a Edge Map em duas maps diferentes: a primeira com as arestas de bordo, e a segunda com as arestas de interior. Desta maneira, a procura por arestas em operações no bordo da variedade ganham em eficiência já que em um contêiner map, uma busca é realizada em tempo $O(log(N))$, onde neste cado N é o número de arestas armazenadas.

Vertex Half–Face: Para muitas aplicações é importante que consigamos calcular a estrela do vértice de maneira eficiente. Para isso, é útil armazenarmos mais um contêiner do tipo vector, chamaremos de Vertex Half–Face e denotaremos por VH[], que armazena, para cada vértice v, o índice de uma de suas *half–faces* incidentes. Se o vértice for um vértice de bordo, a *half–face* armazenada deve ser uma *half–face* de bordo e desta forma podemos classificar eficientemente um vértice como vértice de bordo ou vértice de interior.

O contêiner VH[] tem tamanho $\mathbf{n}_0$ e pode ser alocado em tempo $O(4\mathbf{n}_3)$ usando o algoritmo 14.

Algorithm 14 Construção do contêiner Vertex Half–Face

```
1: VH[i] ← −1; {Inits the container}
2: for HF_id hf ∈ {0 ... 4n_3 − 1} do
3:    if O[hf] = −1 then
4:       {Stores the boundary half–face}
         VH[V[next_hf(hf)]] ← hf;
         VH[V[ mid_hf(hf)]] ← hf;
         VH[V[prev_hf(hf)]] ← hf;
5:    else if VH[∗] = −1 then
6:       {Stores the first half–face}
         VH[∗] ← hf;
7:    end if
8: end for
```

5.4
Nível 3: Representação das Superfícies de Bordo

Como citamos no capítulo 2 o bordo de uma 3–variedade com bordo é uma 2–variedade sem bordo. (Lopes and Tavares 1997) mostraram a importância de termos uma manipulação eficiente das células de bordo para construir e destruir variedades de dimensão 3 controlando sua topologia. Para isso, as relações de incidência e adjacência das células de bordo devem ser explicitamente representadas. A CHF utiliza a CHE, descrita no capítulo 4 para realizar tal representação.

De fato, a CHF implementa o segundo nível da CHE, isto é, ela usa dois contêineres de inteiros do tipo vector, o contêiner bV[] e o contêiner bO[] para armazenar os vértices iniciais e a *half–edge* oposta de cada *half–edge* da

superfície de bordo. Os vértices da CHE e da CHF tem os mesmos índices e desta maneira não é preciso armazenar um novo contêiner com informações geométricas para o nível 0 da CHE. Os contêineres bV[] e bO[] podem ser obtidos em tempo linear através de V[] e O[].

5.5 Exemplo de Construção da CHF

A seguir mostraremos um exemplo simples de construção da CHF. A 3–variedade escolhida para o exemplo é composta por apenas dois tetraedros que compartilham uma de suas faces. A figura 5.12 ilustra o modelo, e a representação de cada um de seus vértices e half–faces, além dos contêineres que compõem cada nível da estrutura de dados. Observe que o modelo é formado por 5 vértices e 2 tetraedros.

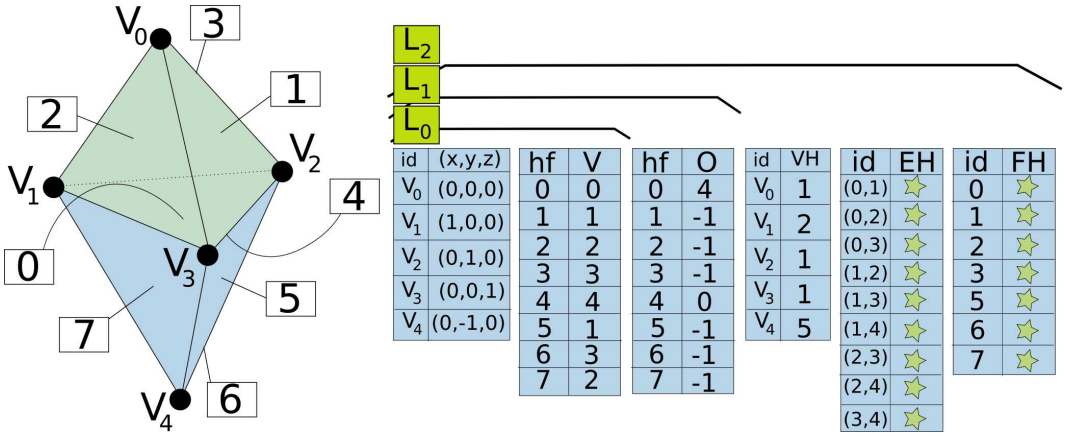

Figura 5.12: Exemplo de construção da CHE .

Para o modelo em questão, o nível 0 da CHF é composto pelos contêineres G[] e V[] com 5 e 8 entradas respectivamente.

No nível 1 adicionamos o contêiner O[] também composto por 8 entradas. Ainda, representamos os opostos de half–faces de bordo pelo inteiro -1.

No nível 2 incluímos os contêineres VH[], EH[] e FH[]. Observe que o número de entradas de VH[] é igual 5, ou seja igual ao número de vértices do modelo. Ainda, na map EH[], identificamos cada aresta pelo par ordenado formado pelos índices dos vértices da aresta. O contêiner EH[] tem tamanho igual a 9, que é o número de arestas do modelo. Por fim, na map FH[], identificamos cada face do modelo pela half–face incidente de menor índice. O contêiner FH[] tem 7 entradas, que é o número de faces do modelo.

Não estamos representado neste exemplo o nível 3 da CHF, já que este é composto pela CHE da superfíceie de bordo, e já vimos um exemplo de construção da CHE na seção 4.5.

5.6 Interrogações Topológicas na CHF

Nesta seção discutiremos a performance das *funções de resposta às informações topológicas do tipo* R_{pq} em cada nível da CHF, ainda, iremos propor algoritmos para o cálculo de cada uma delas nos diferentes níveis da estrutura.

$\mathbf{R_{0*}$ – Estrela do Vértice:} A Estrela do vértice (R_{0*}) é particularmente essencial para modelagem volumétrica. A cada nível, com o aumento da quantidade de informações armazenadas, a CHF obtém de maneira mais eficiente os simplexos pertencentes a estrela de um vértice de índice V_{id} v.

A CHF responde a operação R_{0*} em tempo $O(\mathbf{n}_3)$ no nível 0, já que, por não existirem informações sobre a adjacência dos tetraedros da malha, o contêiner V[] precisa ser atravessado, em uma busca das *half–faces* incidentes ao vértice. O algoritmo descrito em 15, propõem a implementação das funções R_{0*} no nível 0.

Algorithm 15 R_{0*}(v), level 0

```
1: container< σ* > R0*
2: for HFid hf ∈ {0 ... 4n3 − 1} do {all half–faces}
3:   if v = V[hf] then
4:     Tid t ← ⌊hf/4⌋; {gets the incident tetrahedron.}
5:     get all incident σ* in t; {get the incident ∗–simplexes.}
6:     R0*.insert(σ*); {add all ∗–simplexes}
7:     continue
8:   end if
9: end for
```

No nível 1, o contêiner V[] só precisa ser atravessado até que a primeira *half–face* incidente ao vértice V_{id} v seja descoberta. Em seguida, utilizando o contêiner O[] e o conjunto de regras descritas na seção 5.1, a estrela do vértice é obtida em tempo $O(deg(\mathsf{v}))$, onde $deg(\mathsf{v})$ é o número de tetraedros incidentes a v. Assim no pior caso, R_{0*} tem complexidade $O(\mathbf{n}_3)$, mas, em média, o algoritmo é $deg(\mathsf{v})$ vezes mais rápido que o nível 0. O algoritmo usado para a obtenção dos simplexos na estrela de uma vértice V_{id} v, no nível 1 é uma combinação das idéias contidas nos algoritmos 15 e 16.

Por fim, nos níveis 2 e 3 o tempo gasto para calcularmos a estrela de um vértice v se reduz a $O(deg(\mathsf{v}))$ já que, através do contêiner VH[], podemos obter diretamente a primeira *half–face* incidente ao vértice v, e em seguida novamente através do contêiner O[] e das regras binárias, obtemos os simplexos restantes. O cálculo das funções R_{0*} nos níveis 2 e 3 da CHF pode ser feito através do algoritmo 16.

Algorithm 16 $R_{0*}(\mathsf{v}, \mathsf{v}_0 \leftarrow \mathsf{VH}[\mathsf{v}])$, level 2,3

```
1: stack< T_id > st; container< σ* > R_0*;
2: st.push(⌊v_0/4⌋); {get the first incident tetrahedron.}
3: repeat
4:    T_id t ← st.pop(); {get the next tetrahedron.}
5:    if t was not visited then
6:       get all incident σ*; {get all incident ∗–simplexes.}
7:       R_0*.insert(σ*); {add all incident ∗–simplexes.}
8:       st.push(t_ng); {add neighbors tetrahedrons}
9:    end if
10: until st not empty
```

$\mathbf{R_{1*}}$ – Estrela da Aresta Na CHF, identificamos uma aresta por um par ordenado de inteiros que representam os índices dos vértices da aresta. Sendo assim, a relação de incidência R_{10} é diretamente respondida em todos os níveis da estrutura.

No nível 0, o tempo gasto pela CHF para responder as relações R_{1*} é $O(\mathbf{n}_3)$, pois já que não temos conhecimento da relação de adjacência dos tetraedros, o método precisa atravessar todo o contêiner V[] em busca da aresta $\mathsf{e} =< \mathsf{v}_1, \mathsf{v}_2 >$. O algoritmo utilizado para o cálculo das operações R_{1*} é descrito a seguir no algoritmo 17.

Algorithm 17 R_{1*} $(E_{id} < \mathsf{v}_1, \mathsf{v}_2 >)$, level 0

```
1: container< σ* > R_1*
2: for HF_id hf ∈ {0 ... 4n_3 − 1} do {all half–faces}
3:    if v_1 = V[hf] and
      (v_2 = V[next_hf(hf)] or v_2 = V[mid_hf(hf)] or v_2 = V[prev_hf(hf)]) then
4:       T_id t ← ⌊hf/4⌋; {get the incident tetrahedron.}
5:       get all incident σ* in t; {get all incident ∗–simplexes.}
6:       R_13.insert(σ*) ; {add all simplexes.}
7:       continue
8:    end if
9: end for
```

No nível 1, novamente precisamos atravessar o contêiner V[] em busca da primeira *half–face* incidente a aresta $\mathsf{e} =< \mathsf{v}_1, \mathsf{v}_2 >$ e em seguida com o auxilio do contêiner O[] e das regras definidas anteriormente, o ciclo de *half–faces* em torno da aresta é obtido em tempo $O(deg(\mathsf{e}))$, onde $deg(\mathsf{e})$ é igual ao número de tetraedros incidentes a aresta e. No pior caso, o level 1 obtém a estrela de uma aresta em tempo $O(\mathbf{n}_3)$ mas em média o cálculo é feito $O(deg(\mathsf{e}))$ vezes mais rápido que no level 0. O algoritmo 17 e 18 combinados nos dá um meio de calcular as operações R_{1*} no nível 1 da CHF.

Por fim, nos níveis 2 e 3 a complexidade do cálculo da estrela de uma aresta E_{id} e se reduz a $O(deg(\mathsf{e}))$ já que através do contêiner EH[] podemos

Algorithm 18 R_{1*} ($E_{id} < \mathsf{v}_1, \mathsf{v}_2, (\mathsf{hf}_0, \mathsf{he}_0) >$), level 2,3

container$< \sigma^* > R_{1*}$
T_{id} $\mathsf{t}_0 \leftarrow \lfloor \mathsf{hf}_0/4 \rfloor$; {get the first tetrahedron.}
HF_{id} $\mathsf{hf} \leftarrow \mathsf{hf}_0$; HE_{id} $\mathsf{he} \leftarrow \mathsf{he}_0$; T_{id} $\mathsf{t} \leftarrow \mathsf{t}_0$
repeat
  get all incident σ^* in t; {get all incident $*$–simplexes.}
  R_{1*}.insert(σ^*) {add all incident $*$–simplexes.}
  $(\mathsf{hf}, \mathsf{he}) \leftarrow \text{mate}_{\mathsf{he}}(\text{radial}_{\mathsf{he}}(\mathsf{hf}, \mathsf{he}))$; {go to the mate of the radial.}
  $\mathsf{t} \leftarrow \lfloor \mathsf{hf}/4 \rfloor$; {get the incident tetrahedron.}
until $\mathsf{t} \neq -1$ and $\mathsf{t} \neq \mathsf{t}_0$

obter diretamente a primeira half–face incidente à aresta e, e em seguida novamente através do contêiner $\mathsf{O}[]$ e das regras binárias, obtemos os simplexos restantes. O algoritmo descrito em 18 podem ser usados para a obtenção dos simplexos na estrela de uma aresta e nos níveis 2 e 3 da CHF.

$\mathbf{R_{2*}}$ – Estrela da Face Obter os simplexos incidentes a uma face é bastante simples. Como identificas uma face por uma de suas *half–faces* hf, podemos obter seus vértices diretamente fazendo $\mathsf{v}_0 = \mathsf{V}[\text{next}_{\mathsf{hf}}(\mathsf{hf})]$, $\mathsf{v}_1 = \mathsf{V}[\text{mid}_{\mathsf{hf}}(\mathsf{hf})]$, $\mathsf{v}_2 = \mathsf{V}[\text{prev}_{\mathsf{hf}}(\mathsf{hf})]$, respondendo assim a operação R_{20} em tempo constante em todos os níveis. Ainda, é fácil perceber que as arestas de R_{21} são $\mathsf{v}_0\mathsf{v}_1$, $\mathsf{v}_1\mathsf{v}_2$, $\mathsf{v}_2\mathsf{v}_0$ e assim podem ser descobertas facilmente, respondendo a operação R_{21} também em tempo constante.

Para calcularmos as operações R_{2*} no nível 0, precisamos atravessar todo o contêiner $\mathsf{V}[]$ em busca da *half–face* oposta à aquela que identifica a face, caso ela exista. Logo a complexidade do algoritmo é $O(\mathbf{n}_3)$ para as operações R_{2*} no nível 0. O algoritmo 19 descreve o cálculo da estrela de uma face no nível 0.

Algorithm 19 R_{2*} (F_{id} f), level 0

container$< \sigma^* > R_{2*}$
V_{id} $\mathsf{v}_1 \leftarrow \mathsf{V}[\text{next}_{\mathsf{hf}}(\mathsf{f})]$; V_{id} $\mathsf{v}_2 \leftarrow \mathsf{V}[\text{mid}_{\mathsf{hf}}(\mathsf{f})]$; V_{id} $\mathsf{v}_3 \leftarrow \mathsf{V}[\text{prev}_{\mathsf{hf}}(\mathsf{f})]$
for HF_{id} $\mathsf{hf} \in \{0 \ldots 4\mathbf{n}_3 - 1\}$ **do** {all half–faces}
  if $\mathsf{v}_1 = \mathsf{V}[\mathsf{hf}]$ **and**
  $(\mathsf{v}_2 = \mathsf{V}[\text{next}_{\mathsf{hf}}(\mathsf{hf})]$ **or** $\mathsf{v}_2 = \mathsf{V}[\text{mid}_{\mathsf{hf}}(\mathsf{hf})]$ **or** $\mathsf{v}_2 = \mathsf{V}[\text{prev}_{\mathsf{hf}}(\mathsf{hf})])$ **and**
  $(\mathsf{v}_3 = \mathsf{V}[\text{next}_{\mathsf{hf}}(\mathsf{hf})]$ **or** $\mathsf{v}_3 = \mathsf{V}[\text{mid}_{\mathsf{hf}}(\mathsf{hf})]$ **or** $\mathsf{v}_3 = \mathsf{V}[\text{prev}_{\mathsf{hf}}(\mathsf{hf})])$ **then**
    T_{id} $\mathsf{t} \leftarrow \lfloor \mathsf{hf}/4 \rfloor$; {get the incident tetrahedron.}
    get all incident σ^* in t; {get all incident $*$–simplexes.}
    R_{2*}.insert(σ^*) ; {add all incident $*$–simplexes.}
    continue;
  end if
end for

Do nível 1 em diante, a complexidade é reduzida para $O(1)$, pois dada a *half–face* identificadora de uma face, recuperarmos a outra diretamente através do contêiner $\mathsf{O}[]$ e conseqüentemente conseguimos obter todos os $*$–simplexos pertencentes a estrela da face.

$\mathbf{R_{3*}$ – Adjacências e Incidências de Tetraedros Todas as relações de incidência e adjacência de tetraedros são respondidas em tempo constante em todos os níveis da CHF, exceto pelo nível 0, onde a pergunta por tetraedros adjacentes é respondida em $O(\mathbf{n}_3)$.

Dado um tetraedro de índice T_{id} t, pela construção da CHF, sabemos que suas *half–faces* tem índices $4\mathsf{t}$, $4\mathsf{t}+1$, $4\mathsf{t}+2$ e $4\mathsf{t}+3$, logo determinamos diretamente todos os vértices, arestas e faces incidentes t.

O algoritmo 20 descreve o cálculo dos tetraedros adjacentes a um tetraedro. Como todos os outros algoritmos implementados no nível 0, o método é baseado em buscas na tabela $\mathsf{V}[]$.

Algorithm 20 R_{33} (T_{id} t), level 0

```
1: container< V_id > R_33;
2: V_id v_0 ← V[4t];      V_id v_1 ← V[4t + 1]
   V_id v_2 ← V[4t + 2]; V_id v_3 ← V[4t + 3];
3: for HF_id h ∈ {0 ... 4n_3 − 1} do {all half–faces}
4:    V_id p_1 = V[next_hf(h)]; V_id p_2 = V[mid_hf(h)]; V_id p_3 = V[prev_hf(h)];
5:    if (v_1 = p_0 and v_2 = p_1 and v_3 = p_2) or
         (v_1 = p_0 and v_2 = p_2 and v_3 = p_3) or
         (v_1 = p_0 and v_2 = p_1 and v_3 = p_3) or
         (v_1 = p_1 and v_2 = p_2 and v_3 = p_3) then
6:       R_13.insert(⌊h/4⌋) ;{add tetrahedron}
7:       continue
8:    end if
9: end for
```

Percorrimento das Superfícies de Bordo: Podemos escrever algoritmos eficientes para percorrimento de células de bordo a partir do nível 1 da CHF, já que nele são incluídas informações sobre as relações de adjacência entre os tetraedros do modelo. Entretando é no nível 2, com a representação explícita das células de bordo através dos contêineres adicionais propostos, e principalmente no nível 3, com a representação explícita da superfície de bordo, que conseguimos responder com mais eficiência questões como o cálculo de estrela de vértices da superfície de bordo.

Para tal, no nível 2, a divisão proposta para os contêineres edge Map e Face Map, que separa as células de bordo e interior de cada um deles, aumenta

bastante a eficiência dos algoritmos, já que a recuperação de um elemento em um contêiner do tipo map é feita em tempo $O(log(N))$, com N igual ao número de elementos no contêiner.

No nível 3, podemos trabalhar com os algoritmos de interesse propostos na seção 4.6, para a resposta de interrogações topológicas da CHE.

Revisão: Na tabela 5.2, temos uma visão geral da complexidade de todos os algoritmos propostos neste capítulo para a implementação das *funções resposta à interrogações topológicas do tipo* R_{pq}.

Podemos observar que no nível 1, apesar do uso do container O[], ainda obtemos, no pior caso, R_{0*}, R_{1*} ($* \geq 1$) com complexidade $O(\mathbf{n}_3)$, mas em média tais algoritmos são $deg(\sigma)$ vezes mais rápidos no do nível 0 ($deg(\sigma)$ é o número de simplexos incidentes ao $*$–simplexo σ). Ainda a partir do nível 2 obtemos os algorítmos em tempo ótimo.

Vale citar, no entanto, que não levamos em conta na análise de complexidade, o tempo gasto para identificar uma face ou um tetraedro nos cálculos de R_{2*} e R_{3*}. Em outras palavras, se desejarmos calcular a estrela de uma face cujos vértices tem índices (v_a, v_b, v_c) ou da um tetraedro cujos vértices são (v_a, v_b, v_c, v_d) precisamos primeiro identificar qual a *half–face* que representa a face ou o índice do tetraedro. Uma vez identificado o índice da face ou do tetraedro é feita análise da complexidade topológica dos algoritmos.

	Nível 0	*Nível* 1	*Nível* 2	*Nível* 3
R_{00}	$O(\mathbf{n}_3)$	$O(\mathbf{n}_3)$	$O(deg(\mathsf{v}))$	$O(deg(\mathsf{v}))$
R_{01}	$O(\mathbf{n}_3)$	$O(\mathbf{n}_3)$	$O(deg(\mathsf{v}))$	$O(deg(\mathsf{v}))$
R_{02}	$O(\mathbf{n}_3)$	$O(\mathbf{n}_3)$	$O(deg(\mathsf{v}))$	$O(deg(\mathsf{v}))$
R_{03}	$O(\mathbf{n}_3)$	$O(\mathbf{n}_3)$	$O(deg(\mathsf{v}))$	$O(deg(\mathsf{v}))$
R_{10}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{11}	$O(\mathbf{n}_3)$	$O(\mathbf{n}_3)$	$O(deg(\mathsf{e}))$	$O(deg(\mathsf{e}))$
R_{12}	$O(\mathbf{n}_3)$	$O(\mathbf{n}_3)$	$O(deg(\mathsf{e}))$	$O(deg(\mathsf{e}))$
R_{13}	$O(\mathbf{n}_3)$	$O(\mathbf{n}_3)$	$O(deg(\mathsf{e}))$	$O(deg(\mathsf{e}))$
R_{20}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{21}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{22}	$O(\mathbf{n}_3)$	$O(1)$	$O(1)$	$O(1)$
R_{23}	$O(\mathbf{n}_3)$	$O(1)$	$O(1)$	$O(1)$
R_{30}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{31}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{32}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{33}	$O(\mathbf{n}_3)$	$O(1)$	$O(1)$	$O(1)$

Tabela 5.2: Complexidade das funções resposta na CHF.