4
A Estrutura de Dados CHE

Neste capítulo descreveremos os quatro níveis da estrutura de dados CHE, abordando suas principais característica e a construção de cada um deles a partir do nível anterior.

4.1
Nível 0: Sopa de Triângulos

O primeiro nível que propomos para a construção da CHE é o *nível* 0. Nele armazenamos apenas as informações essenciais para a representação de uma malha de triângulos: a geometria dos vértices e os triângulos que a compõem. Sua principal aplicação é a visualização de malhas, não havendo a intenção de representar direta e eficientemente relações de incidência e adjacência. Assim, no nível 0, o modelo é encarado como uma "sopa" de triângulos (figura 4.1).

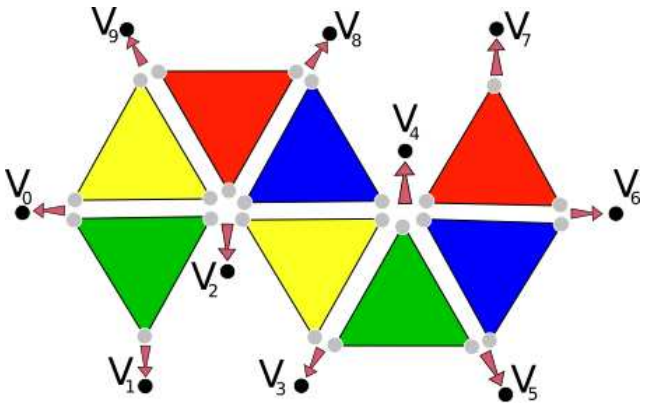

Figura 4.1: Nível 0 da CHE: Sopa de Triângulos.

Para acessar os elementos de um triângulo trabalharemos com o conceito de *half–edge*, proposto inicialmente por (Mäntylä 1988).

Definição 4.1 Half–edge: Uma *half–edge* é uma aresta dotada de uma orientação induzida por um de seus triângulos incidentes. Analogamente, uma *half–edge* é associação de um triângulo a uma de suas arestas de bordo (figura 4.2).

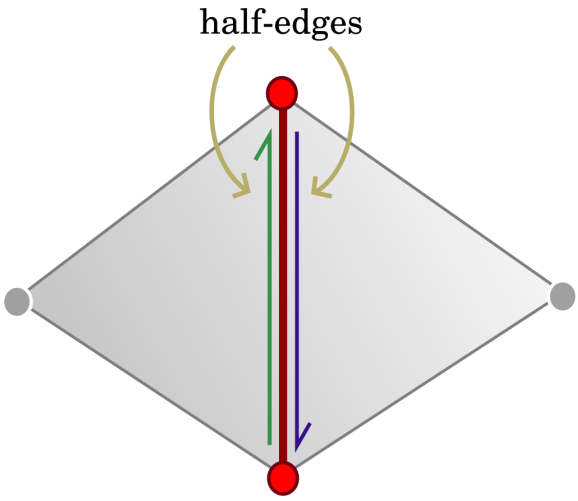

Figura 4.2: Half–edges de uma aresta em seus triângulos incidentes.

Na CHE, vértices, *half–edges* e triângulos são indexados por números inteiros não-negativos. Cada triângulo é representado por 3 *half–edges* consecutivas, que formam seu bordo orientado. Assim, as *half–edges* 0, 1 e 2 pertencem ao triângulo 0, as *half–edges* 3, 4 e 5 são as *half–edges* do triângulo 1, e assim sucessivamente.

Contêiner Geometry: As informações geométricas carregadas pela CHE ficam armazenadas no contêiner do tipo vector chamado Geometry e denotado por G[]. Para cada vértice da malha, armazenamos suas coordenadas, vetor normal etc. Assim se desejamos acessar as informações geométricas de um vértice de índice V_{id} v devemos buscá-las através do contêiner G[], acessando a posição G[v]. O contêiner G[] tem tamanho igual a $\mathbf{n}_0$.

Regras para Half–Edges: Como descrito acima, vértices, *half–edges* e triângulos são indexados por inteiros não–negativos, e de devido à construção adotada, podemos escrever algumas regras aritméticas para obter informações sobre como tais entidades se relacionam.

Uma *half–edge* de índice HE_{id} he pertence ao triângulo $\lfloor \mathsf{he}/3 \rfloor$. Por outro lado, os índices das 3 *half–edges* de um triângulo F_{id} t são $3\mathsf{t}, 3\mathsf{t}+1$ e $3\mathsf{t}+2$. Dada uma *half–edge* de índice HE_{id} he, definimos suas *half–edges next* e *previous* como as *half–edges* anterior e posterior no ciclo de *half–edges* do triângulo incidente (figura 4.3). Para obter seus índices, utilizamos as seguintes expressões:

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

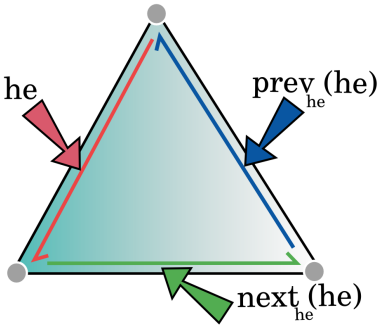

Figura 4.3: Half–edges $\text{next}_{\mathsf{he}}(\mathsf{he})$ e $\text{prev}_{\mathsf{he}}(\mathsf{he})$.

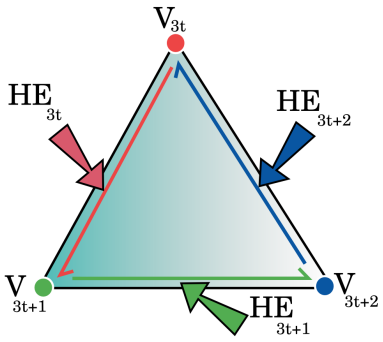

Figura 4.4: Relação entre vértices e half–edges.

Contêiner Vertex: Para cada *half–edge* armazenamos o índice de seu vértice inicial em um contêiner do tipo vector chamado Vertex e denotado por V[]. Ou seja, o número $\mathsf{v} = \mathsf{V}[\mathsf{he}]$ corresponde ao índice do vértice inicial de uma *half–edge* he (figura 4.4). O contêiner V[] tem tamanho $3\mathbf{n}_2$ e cada entrada varia de 0 a $\mathbf{n}_0 - 1$.

4.2 Nível 1: Adjacência entre Triângulos

O *nível* 1 da CHE deixa de tratar a malha como uma "sopa" de triângulos e passa a vê-la como um conjunto de triângulos conectados (figura 4.5). Para isso, adicionamos, neste nível, informações sobre as relações de adjacência entre os triângulos da malha. Em muitas aplicações precisamos atravessar os triângulos de maneira eficiente, e isso seria impossível sem o conhecimento de suas vizinhanças.

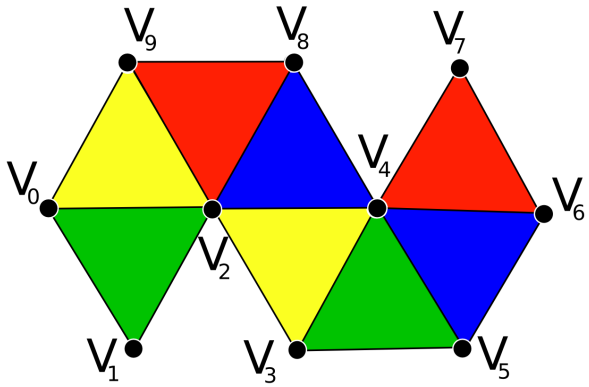

Figura 4.5: Nível 1 da CHE: Adjacência entre os triângulos.

Em uma variedade combinatória de dimensão 2, cada aresta é incidente a no máximo dois triângulos. Assim cada aresta da malha terá duas *half–edges* incidentes quando for de interior e uma quando estiver no bordo. Quando

conhecemos o par de *half–edges* incidentes a uma determinada aresta, temos implicitamente que seus triângulos de origem são adjacente. Formalizaremos esta idéia através da definição de *half–edges* opostas.

Definição 4.2 Half–edge oposta: Uma *half–edge* é *oposta* a *half–edge* HE_{id} he se tem os mesmos vértices de he mas orientação oposta (figura 4.6).

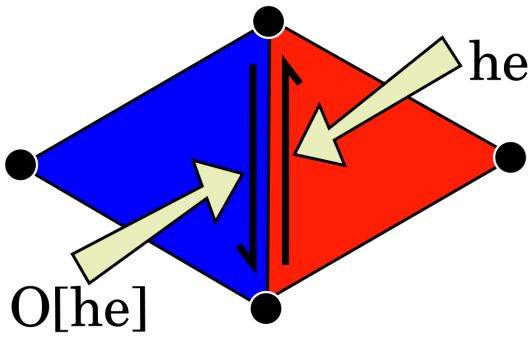

Figura 4.6: Half–edge oposta a *half–edge* HE_{id} he.

Assim as relações de adjacência, serão representadas através do contêiner do tipo vector chamado Opposite, denotado por O[], onde armazenaremos a *half–edge* oposta de cada *half–edge* da malha. Quando uma *half–edge* HE_{id} he for incidente a uma aresta de bordo, ela não terá uma *half–edge* oposta e diremos que $\mathsf{O}[\mathsf{he}] = -1$, desta maneira ganhamos uma forma direta para saber se uma *half–edge* é ou não de bordo. O contêiner O[] tem tamanho $3\mathbf{n}_2$ e suas entradas variam de -1 a $3\mathbf{n}_2 - 1$.

O algoritmo 1 descreve uma maneira eficiente de construirmos o contêiner O[] a partir do contêiner V[]. Utilizaremos um contêiner genérico do tipo map para auxiliar a construção do contêiner O[].

Algorithm 1 Construção do Contêiner Opposite

```
1:  O[i] ← −1 {Inits the container}
2:  map{V_id × V_id → HE_id} adjacency
3:  for HE_id he ∈ {0 ... 3n_2 − 1} do
4:      {Gets the vertices of he}
        v_0 ← V[he] ; v_1 ← V[next_he(he)] ;
        v^2 ← sort(v_0, v_1)
5:      if adjacency.find(v^2) then
6:          O[he] ← adjacency [v^2]
7:          O[O[he]] ← he {Found opposite half–edge}
8:          adjacency.erase(v^2)
9:      else {Temporarily stores half–edge}
10:         adjacency [v^2] ← he
11:     end if
12: end for
```

4.3 Nível 2: Representação das Células

Em muitos momentos, é importante que a estrutura de dados nos possibilite representar explicitamente cada simplexo da variedade combinatória, pois assim, podemos associar a cada um deles um conjunto de atributos que podem variar de acordo com a aplicação. Por exemplo, em algoritmos de simplificação, podemos atribuir a cada simplexo um certo custo para sua remoção. Seguindo esse pensamento, no *nível* 2 da CHE acrescentamos à estrutura uma representação explícita para cada célula da malha (figura 4.7).

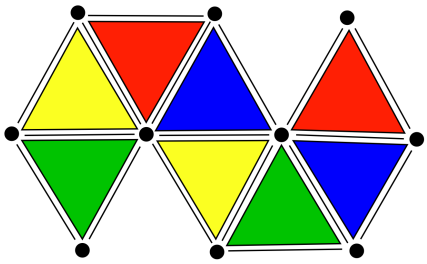

Figura 4.7: Nível 2 da CHE: Representação das células.

Tal representação é feita por meio de dois novos contêineres: Vertex Half–Edge e Edge Map. A seguir descreveremos cada um deles assim como a maneira eficiente de construí-los.

Edge Map: Identificaremos uma aresta por uma de suas *half–edges* incidentes $E_{id} := \mathsf{he}$, já que a outra *half–edge* pode ser recuperada através do contêiner O[]. Assim através de um contêiner associativo do tipo map, que chamaremos de Edge Map, e denotaremos por EH[], relacionamos o índice de cada aresta com suas propriedades.

Algorithm 2 Construção da Edge Map

```
1: map{HE_id → property} EH
2: for HE_id he ∈ {0 ... 3n_2 − 1} do
3:    {Gets the opposite of he}
      he_0 ← he ; he_1 ← O[he] ;
      h ← min(he_0, he_1) ≥ 0
4:    if EH.find(h) then
5:       continue {Edge already found};
6:    else {Stores the edge}
7:       EH[h] ← property
8:    end if
9: end for
```

Vale observar que os vértices que compõem a aresta são facilmente descobertos através dos vértices iniciais de suas *half–edges*. O contêiner EH[] tem tamanho igual a $\mathbf{n}_1$, e pode ser alocado através do algoritmo 2.

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 caso N é o número de arestas armazenadas.

Vertex Half–Edge: É de extrema importância para muitas aplicações que a estrutura de dados possa acessar de maneira eficiente a estrela de um vértice. Por isso é muito útil armazenarmos um novo contêiner chamado de Vertex Half–Edge e denotado por VH[] que para cada vértice, guarda o índice de uma de suas *half–edges* incidentes.

Podemos simplificar ainda mais o cálculo da estrela se adotarmos o seguinte critério para a escolha da *half–edge* armazenada: Se o vértice é de bordo, então a *half–edge* escolhida será a *half–edge* de bordo que tem o vértice como vértice inicial. Se o vértice for de interior, a *half–face* armazenada será qualquer *half–edge* incidente cujo vértice é o ponto inicial (figura 4.8).

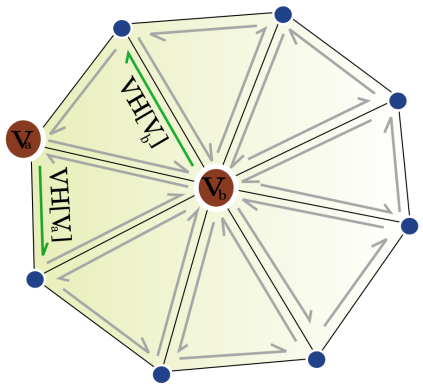

Figura 4.8: Escolha das *half–edges* armazenadas no contêiner VH[].

Dessa maneira, ganhamos uma forma bastante simples de classificar os vértices da malha como vértices de interior ou vértices de bordo. Dado um vértice de índice V_{id} v basta checarmos se O[VH[v]] $= -1$ e caso afirmativo saberemos que o vértice v é de bordo.

O contêiner VH[] tem tamanho $\mathbf{n}_0$ e suas entradas variam de 0 a $3\mathbf{n}_2 - 1$. O algoritmo 3 descreve uma maneira eficiente de alocarmos o contêiner VH[].

4.4 Nível 3: Representação das Curvas de Bordo

Como apresentamos em 3.1, (Lopes and Tavares 1997) mostraram a importância de termos uma representação eficiente do bordo de variedades para maior controle da topologia do objeto. Ainda, algoritmos de triangulação

Algorithm 3 Construção do contêiner Vertex Half–Edge

```
1: VH[i] ← −1; {Inits the container}
2: for HE_id he ∈ {0 ... 3n_2 − 1} do
3:   if O[he] = −1 then
4:     VH[V[he]] ← he; {Stores the boundary half–edge}
5:   else
6:     if VH[V[he]] = −1 then
7:       VH[V[he]] ← he; {Stores the first incident half–edge}
8:     end if
9:   end if
10: end for
```

usando avanço de fronte são exemplos significantes de aplicações que necessitam de uma representação explícita das curvas de bordo da superfície. Assim, no *nível* 3 da CHE adicionamos a estrutura uma representação explícita para as curvas de bordo de uma superfície (figura 4.9).

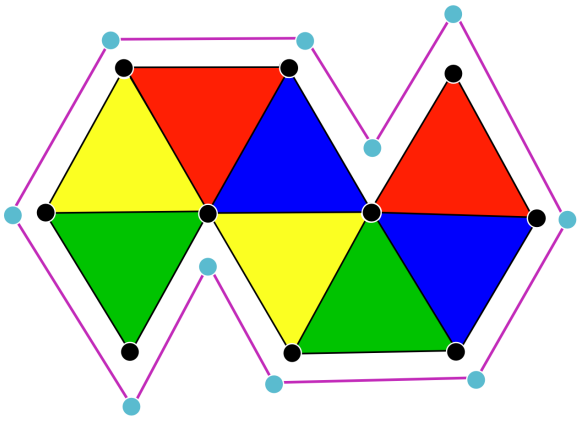

Figura 4.9: CHE nível 3: Representação das curvas de bordo.

Contêiner Boundary: Cada curva de bordo é representada por uma de suas *half–edges* incidentes já que, a partir destas, todas as outras podem ser obtidas eficientemente através das regras aritméticas e do contêiner O[], descritos anteriormente. Para isto, criamos um novo contêiner vector chamado Contêiner Boundary e denotado por CH[], que armazena a *half–edge* representante de cada curva de bordo da superfície. Logo, o tamanho de CH[] é igual a $\mathbf{n}_\partial$, definido como o número de curvas de bordo existentes na superfície.

Ainda, para otimizarmos a localização de arestas nas curvas de bordo, armazenamos no contêiner O[] um atributo que informa o índice da componente de bordo a que pertence cada *half–edge* de bordo. Por exemplo, se uma *half–edge* de bordo he pertence à componente de número 2, então O[he] = −2. O algoritmo 4 descreve a criação do contêiner CH[].

Algorithm 4 Construção do Contêiner Boundary

```
1: container<bool> Visited ← false;
2: for HE_id he ∈ {0 ... 3n_2 − 1} do
3:   if O[he] = −1 then
4:     HE_id he_0 ← he, CH[].insert(he_0)
5:     repeat {Visits the half–edges in the boundary curve}
6:       Visited[he_0] = true;
7:       while O[next_he(he_0)] ≠ −1 do
8:         he_0 ← O[next_he(he_0)]; {Goes to the next half–edge.}
9:       end while
10:      he_0 ← next_he(he_0)
11:    until Visited[he_0] = false
12:  end if
13: end for
```

4.5 Exemplo de Construção da CHE

A seguir mostraremos um exemplo simples de construção da CHE. A superfície escolhida para o exemplo é um tetraedro sem uma de suas faces. A figura 4.10 ilustra a planificação do modelo, e a representação de cada um de seus vértices e half–edges, além dos contêineres que compõem cada nível da estrutura de dados. Observe que o modelo é formado por 4 vértices e 3 faces.

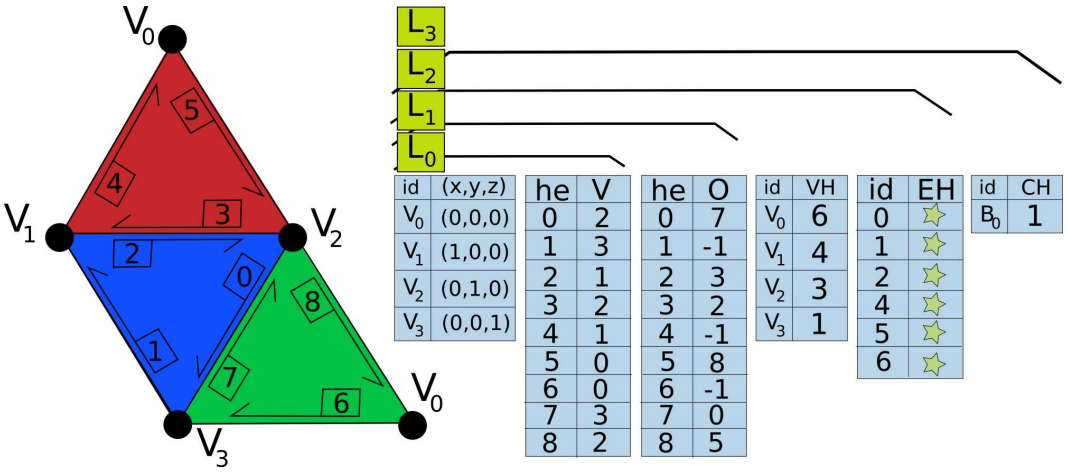

Figura 4.10: Exemplo de construção da CHE .

Para o modelo em questão, o nível 0 da CHE é composto pelos contêineres G[] e V[] com 4 e 9 entradas respectivamente.

No nível 1 adicionamos o contêiner O[] também composto por 9 entradas. Estamos codificando os opostos de half–edges de bordo com o inteiro -1.

No nível 2 incluímos os contêineres VH[] e EH[]. Observe que o número de entradas de VH[] é igual 4, ou seja igual ao número de vértices do modelo.

Ainda, na map EH[], identificamos cada aresta pela half–edge incidente de menor índice. O contêiner EH[] tem tamanho igual a 6, que é o número de arestas do modelo.

Por fim, no nível 3, adicionamos a representação da curva de bordo gerada pela ausência de uma das faces do tetraedro. Como o modelo contém apenas uma curva de bordo, o contêiner CH[] tem apenas uma entrada.

4.6 Interrogações Topológicas na CHE

Nesta seção discutiremos a performance das *funções de resposta às interrogações topológicas do tipo* R_{pq} em cada nível da CHE. Além disso iremos propor a implementação de cada função nos diferentes níveis da estrutura. Desta maneira, veremos com detalhes como cada nova informação adicionada aos níveis da CHE influencia na eficiência dos algoritmos.

$\mathbf{R_{0*}}$ – Estrela do Vértice: O cálculo eficiente da estrela do vértice é extremamente importante para diversas aplicações. Com isso, uma estrutura de dados otimizada precisa responder com rapidez as funções de resposta do tipo R_{0*}. A CHE calcula a estrela do vértice com mais eficiência a medida que aumentamos a quantidade de memória utilizada.

Para calcularmos a estrela de um vértice V_{id} v no nível 0, precisamos percorrer o contêiner V[], em busca do vértice v, já que não temos nenhuma informação sobre as relações de adjacência e incidência da malha. Com isso, a estrela de um vértice pode ser obtida apenas em tempo $O(\mathbf{n}_2)$. O algoritmo 5 descreve a implementação das funções R_{0*} no nível 0 da CHE.

Algorithm 5 R_{0*} (V_{id} v), nível 0

```
1: container< σ* > R0*
2: for HE_id he ∈ {0 ... 3n2 − 1} do {for all half–edges.}
3:   if v = V[he] then
4:     F_id t ← ⌊he/3⌋; {get the incident triangle.}
5:     get all incident σ* in t; {get the incident ∗–simplexes.}
6:     R0*.insert(σ*); {add all incident ∗–simplexes.}
7:     continue;
8:   end if
9: end for
```

No nível 1 da estrutura, atravessamos o contêiner V[] até que a primeira half–edge incidente a v seja encontrada, em seguida utilizamos o contêiner O[] e as regras aritméticas previamente descritas para obter os demais elementos da estrela de v. Localizada a primeira *half–edge*, o algoritmo pode ser calculado

em tempo $O(deg(\mathsf{v}))$, onde $deg(\mathsf{v})$ é o número de triângulos na estrela de v. Logo no pior caso, o cálculo de R_{0*} no nível 1 da CHE tem custo $O(\mathbf{n}_2)$, e em média ele é $deg(\mathsf{v})$ vezes mais rápido que no nível 0 já que temos probabilidade $1/\mathbf{n}_0$ de encontrar o vértice de índice v no contêiner V[].

O algoritmo 6, propõe a implementação da função no nível 1 da CHE. Observe que, para simplificar a escrita do algoritmo, estamos assumindo que a primeira *half–edge* incidente ao vértice v é uma *half–edge* de bordo, no caso em que v é um vértice de bordo, mas que mesmo sem tal restrição conseguimos escrever um algoritmo análogo e de mesma complexidade para o cálculo da operação R_{0*} no nível 1.

Algorithm 6 R_{0*} (V_{id} v), nível 1

```
1: container< σ* > R0*
2: for HE_id he ∈ {0 ... 3n2 − 1} do {for all half–edges}
3:    if v = V[he] then {get the first half–edge.}
4:       HE_id he0 ← he; HE_id h ← he0; break;
5:    end if
6: end for
7: repeat
8:    F_id t ← ⌊h/3⌋; {get the incident triangle.}
9:    get all incident σ* in t; {get the incident ∗–simplexes.}
10:   R0*.insert(σ*); {add all incident simplexes.}
11:   h ← next_he(O[h]); {go to the next of the opposite.}
12: until h ≠ −1 and h ≠ he0
```

Por fim, nos níveis 2 e 3, com o auxílio do contêiner VH[], conseguimos reduzir o tempo gasto no cálculo das funções R_{0*} para $deg(\mathsf{v})$, já que não precisamos mais procurar a primeira *half–edge* incidente ao vértice v. Assim, através da *half–edge* armazenada em HE_{id} $\mathsf{he} = \mathsf{VH}[\mathsf{v}]$, do contêiner O[], e das regras aritméticas, obtemos todos os simplexos da estrela de v. O algoritmo 7, descrito a seguir, propõe a implementação da função nos níveis 2 e 3 da estrutura de dados.

Algorithm 7 R_{0*} (V_{id} v, VH[v]), níveis 2,3

```
1: container< σ* > R0*
2: HE_id he0 ← VH[v]; HE_id h ← he0; {get the first half–edge.}
3: repeat
4:    F_id t ← ⌊h/3⌋; {get the incident triangle.}
5:    get all incident σ* in t; {get the incident ∗–simplexes.}
6:    R0*.insert(σ*); {add all incident ∗–simplexes.}
7:    h ← next_he(O[h]); {go to the next of the opposite.}
8: until h ≠ −1 and h ≠ he0
```

$\mathbf{R_{1*}}$ – Estrela da Aresta: A análise do desempenho da CHE no cálculo da estrela de uma aresta em uma superfície é análoga à feita para o cálculo das estrelas de vértices. Vale observar que uma aresta é representada por uma de suas *half–edges* incidentes, ou seja E_{id} e = he, e que R_{10} é uma relação de incidência que retorna os vértices incidentes a aresta e. Pela representação adotada, os vértices de e, são diretamente obtidos através do contêiner V[] e são eles: $\mathsf{v}_a = \mathsf{V}[\mathsf{he}]$ e $\mathsf{v}_b = \mathsf{V}[\text{next}_{\mathsf{he}}(\mathsf{he})]$.

No nível 0 da CHE, dada uma aresta E_{id} e = he devemos percorrer o contêiner V[], em busca da outra *half–edge* incidente a aresta e, caso ela exista. Sendo assim, a CHE responde as relações R_{1*}, com $* \geq 1$, em tempo $O(\mathbf{n}_2)$. O algoritmo 8 descreve o cálculo das funções R_{1*} no nível 0 da CHE.

Algorithm 8 R_{1*} (E_{id} e), nível 0

```
1: container< σ* > R1*
2: Vid va ← V[e]; Vid vb ← V[next_he(e)]
3: get all incident σ*; R1*.insert(σ*);
4: for HEid he ∈ {0 ... 3n2 − 1} do {for all half–edges}
5:    if vb = V[he] and va = V[next_he(he)] then
6:       Fid t ← ⌊he/3⌋; {get the incident triangle.}
7:       get all incident σ* in t; {get the incident *–simplexes.}
8:       R1*.insert(σ*); {add all incident *–simplexes.}
9:       break;
10:   end if
11: end for
```

Nos níveis 1, 2, 3 a complexidade do algoritmo reduz para apenas $O(1)$, já que, se quisermos calcular a estrela de uma aresta E_{id} e, podemos usar a tabela O[] e descobrir a segunda *half–edge* incidente à aresta em questão.

$\mathbf{R_{2*}}$ – Adjacências e Incidências dos Triângulos: Todas as relações de incidência de triângulos podem ser respondidas em tempo constante em todos os níveis da CHE. Dado o índice F_{id} t de um triângulo, sabemos que suas *half–edges* tem índices 3t, 3t + 1, 3t + 2, logo os vértices de t são aqueles de índices V[3t], V[3t + 1], V[3t + 2] e as arestas podem ser representadas através de suas *half–edges*.

Para calcular os triângulos adjacentes a um triângulo F_{id} t (função de resposta R_{22}) no nível 0 da CHE, novamente devemos percorrer o contêiner V[] a procura dos mesmos. Com isso, o algoritmo é calculado em tempo $O(\mathbf{n}_2)$ neste nível. O algoritmo 9 propõe a implementação da função R_{22} no nível 0.

Nos níveis 1, 2, 3 da CHE, conseguimos descobrir os triângulos adjacentes a um triângulo de índice F_{id} t em tempo linear através do contêiner O[].

Algorithm 9 R_{22} (F_{id} t), nível 0

```
1: container< F_id t > R22
2: V_id v_a ← V[3t]; V_id v_b ← V[3t + 1]; V_id v_c ← V[3t + 2]
3: for HE_id he ∈ {0 ... 3n_2 − 1} do {for all half–edges}
4:    if (v_a = V[he] and v_c = V[next_he(he)]) or
         (v_c = V[he] and v_b = V[next_he(he)]) or
         (v_b = V[he] and v_a = V[next_he(he)]) then
5:       get incident F_id t_i ← ⌊he/3⌋;
6:       R22.insert(t_i);
7:    end if
8: end for
```

Percorrimento das Curvas de Bordo: Podemos implementar algoritmos eficientes para o percorrimento das curvas de bordo de um modelo a partir do nível 1 da CHE, pois no nível 0, não temos informações sobre a adjacência dos simplexos da malha, logo classificar uma *half–edge* como de bordo ou de interior é uma tarefa cara que invibializa o percorrimento eficiente das componentes de bordo.

Nos níveis 1 e 2, com o auxílio do contêiner O[], procuramos a primeira *half–edge* HE_{id} he de bordo do modelo e, a partir desta, visitamos todas as *half–edges* incidentes a componente de bordo que contém he. Para garantir que todas as componentes de bordo foram percorridas, verificamos se todas as *half–edges* de bordo foram visitadas. Tal algoritmo tem complexidade $O(\mathbf{n}_2)$.

Algorithm 10 Percorrimento das curvas de bordo, nível 3

```
1: container< HE_id he > Bd
2: for HE_id he ∈ CH[] do {for all boundary representatives}
3:    HE_id he_0 ← he
4:    repeat {Half–edge found}
5:       Bd.insert(he);
6:       repeat {Searches the next half–edge}
7:          he ← next_he(O[next_he(he)])
8:       until O[he] = −1
9:    until he ≠ he_0
10: end for
```

No nível 3, através do Contêiner Curve, podemos obter a primeira *half–edge* incidente de cada curva de bordo da malha, e a partir destas, percorrer todas as *half–edges* de bordo do modelo. Neste nível o algoritmo é calculado em tempo $O(\partial\mathbf{n}_1)$, onde $\partial\mathbf{n}_1$ é o número de arestas de bordo do modelo.

No algoritmo 10 sugerimos a implementação do algoritmo de percorrimento das curvas de bordo para o nível 3 da CHE.

Resumo: Por fim, na tabela 4.1, temos uma visão geral da complexidade de todos os algoritmos propostos neste capítulo para a implementação das *funções resposta às 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*} e R_{1*} ($* \geq 1$) com complexidade $O(\mathbf{n}_2)$, mas em média tais algoritmos são $deg(\sigma)$ vezes mais rápidos que no nível 0 (onde $deg(\sigma)$ 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 aresta ou uma face nos cálculos de R_{1*} e R_{2*}. Em outras palavras, se desejarmos calcular a estrela de uma aresta cujos vértices tem índices (v_a, v_b) ou da uma face cujos vértices são (v_a, v_b, v_c) precisamos primeiro identificar qual a *half–edge* que representa a aresta ou o índice da face. Uma vez identificado o índice da aresta ou da face é feita a análise da complexidade topológica dos algoritmos.

	Nível 0	*Nível* 1	*Nível* 2	*Nível* 3
R_{00}	$O(\mathbf{n}_2)$	$O(\mathbf{n}_2)$	$O(deg(\mathsf{v}))$	$O(deg(\mathsf{v}))$
R_{01}	$O(\mathbf{n}_2)$	$O(\mathbf{n}_2)$	$O(deg(\mathsf{v}))$	$O(deg(\mathsf{v}))$
R_{02}	$O(\mathbf{n}_2)$	$O(\mathbf{n}_2)$	$O(deg(\mathsf{v}))$	$O(deg(\mathsf{v}))$
R_{10}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{11}	$O(\mathbf{n}_2)$	$O(1)$	$O(1)$	$O(1)$
R_{12}	$O(\mathbf{n}_2)$	$O(1)$	$O(1)$	$O(1)$
R_{20}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{21}	$O(1)$	$O(1)$	$O(1)$	$O(1)$
R_{22}	$O(\mathbf{n}_2)$	$O(1)$	$O(1)$	$O(1)$

Tabela 4.1: Complexidade das funções resposta na CHE.