Pontifícia Universidade Católica
do Rio de Janeiro

Marcos de Oliveira Lage Ferreira

Estruturas de Dados Topológicas Escalonáveis para Variedades de dimensão 2 e 3

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Matemática Aplicada do Departamento de Matemática da PUC–Rio

Orientador: Prof. Hélio Côrtes Vieira Lopes

Rio de Janeiro
fevereiro de 2006

Pontifícia Universidade Católica
do Rio de Janeiro

Marcos de Oliveira Lage Ferreira

Estruturas de Dados Topológicas Escalonáveis para Variedades de dimensão 2 e 3

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Matemática Aplicada do Departamento de Matemática do Centro Técnico Científico da PUC–Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Hélio Côrtes Vieira Lopes
Orientador
Departamento de Matemática — PUC–Rio

Prof. Luiz Carlos Pacheco R. Velho
IMPA

Prof. Waldemar Celes
Departamento de Informática – PUC–Rio

Prof. Geovan Tavares dos Santos
Departamento de Matemática – PUC–Rio

Prof. Sinésio Pesco
Departamento de Matemática – PUC–Rio

Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 14 de fevereiro de 2006

Marcos de Oliveira Lage Ferreira

Graduou–se em Licenciatura em Matemática na Universidade do Estado do Rio de Janeiro – UERJ

Ficha Catalográfica

Ferreira, Marcos de Oliveira Lage

Estruturas de Dados Topológicas Escalonáveis para Variedades de dimensão 2 e 3 / Marcos de Oliveira Lage Ferreira; orientador: Hélio Côrtes Vieira Lopes. — Rio de Janeiro : PUC–Rio, Departamento de Matemática, 2006.

v., 94 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Matemática.

Inclui referências bibliográficas.

1. Matemática – Tese. 2. Estruturas de Dados Topológicas. 3. Topologia Computacional. 4. Geometria Computacional. I. Lopes, Hélio Côrtes Vieira. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Matemática. III. Título.

CDD: 510

Agradecimentos

Ao meu orientador professor Hélio Lopes pelo apoio e amizade durante a realização deste trabalho.

À CAPES e à FAPERJ pelos auxílios concedidos, sem os quais não poderia ter realizado este trabalho.

Aos meus pais Marcos e Lúcia, minha irmã Cynthia e minha namorada Viviane pelo carinho de sempre.

Aos meus colegas da PUC-Rio, em especial ao Thomas Lewiner.

Aos professores do departamento de Matemática, em especial ao Geovan Tavares, Paul Schweitzer, Marcos Craizer e Sinésio Pesco. Ao professor Marcos Alexandrino da USP e ao professor Luiz Velho do IMPA.

Aos funcionários do departamento de Matemática pela ajuda de sempre, em especial à Creuza.

Resumo

Ferreira, Marcos de Oliveira Lage; Lopes, Hélio Côrtes Vieira. **Estruturas de Dados Topológicas Escalonáveis para Variedades de dimensão** 2 **e** 3. Rio de Janeiro, 2006. 94p. Dissertação de Mestrado — Departamento de Matemática, Pontifícia Universidade Católica do Rio de Janeiro.

Pesquisas na área de estrutura de dados são fundamentais para aumentar a generalidade e eficiência computacional da representação de modelos geométricos. Neste trabalho, apresentamos duas estruturas de dados topológicas escalonáveis, uma para superfícies trianguladas, chamada CHE (*Compact Half–Edge*), e outra para malhas de tetraedros, chamada CHF (*Compact Half–Face*). Tais estruturas são compostas de diferentes níveis, que nos possibilitam alterar a quantidade de dados armazenados com objetivo de melhorar sua eficiência computacional. O uso de APIs baseadas no conceito de objeto, e de herança de classes, possibilitam uma interface única para cada função em todos os níveis das estruturas. A CHE e a CHF requerem pouca memória e são simples de implementar já que substituem o uso de ponteiros pelo de contêineres genéricos e regras aritméticas.

Palavras–chave

Estruturas de Dados Topológicas. Topologia Computacional. Geometria Computacional.

Abstract

Ferreira, Marcos de Oliveira Lage; Lopes, Hélio Côrtes Vieira. **Scalable topological data–structures for 2 and 3 manifolds**. Rio de Janeiro, 2006. 94p. MsC Thesis — Department of Matemática, Pontifícia Universidade Católica do Rio de Janeiro.

Research in data structure area are essential to increase the generality and computational efficiency of geometric models' representation. In this work, we present two new scalable topological data structures, one for triangulated surfaces, called CHE (*Compact Half – Edge*), and the another for tetrahedral meshes, called CHF (*Compact Half – Face*). Such structures are composed of different levels, that enable us to modify the amount of data stored with the objective to improve its computational efficiency. The use of APIs based in the object concept and class inheritance, makes possible an unique interface for each function at any level. CHE and CHF requires very few memory and are simple to implement since they substitute the use of pointers by generic containeres and arithmetical rules.

Keywords

Topological Data–Structures. Computational Topology. Computational Geometry.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Motivação: Complexos celulares são amplamente utilizados para representar objetos multi-dimensionais em vários tipos de aplicações. Em particular, complexos simpliciais se destacam pois suas propriedades combinatórias os fazem mais fáceis de ser entendidos, representados e manipulados do que complexos celulares gerais.

Com freqüência, sistemas de modelagem geométrica ou de visualização científica precisam lidar com malhas simpliciais extremamente grandes, e com isso, o estudo na área de estruturas de dados topológicas se torna fundamental para otimizar a relação entre a flexibilidade, generalidade e eficiência computacional (tanto no uso de memória quanto na performance) da representação de modelos geométricos.

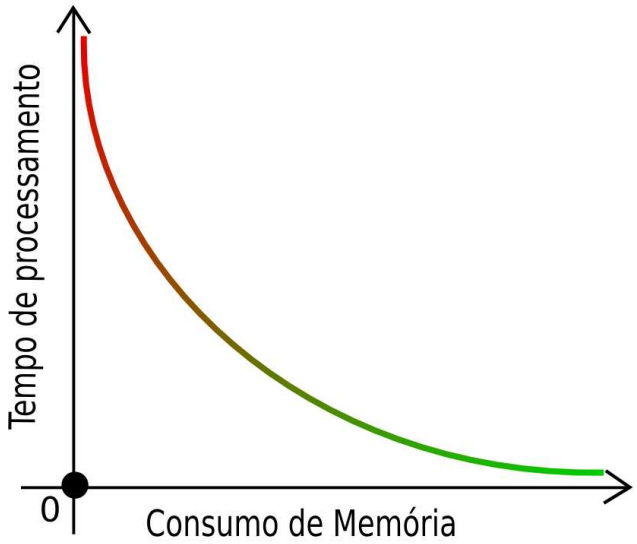

Figura 1.1: Gráfico *performance X memória*.

Ainda, é natural que exista uma relação direta entre o uso de memória e a eficiência da representação (figura 1.1). Quanto mais informação é armazenada, mais memória é requerida, e mais eficiente é a busca das informações topológicas desejadas. Infelizmente, memória é uma limitação física de siste-

mas computacionais e muito embora a performance seja também um aspecto de extrema importância, o tempo é, muitas vezes, uma limitação tolerável.

Por exemplo, quando dobramos a quantidade de dados de entrada que desejamos processar é aceitável que se espere o dobro do tempo do processamento anterior (até mesmo pouco mais do que isso) para que se obtenha o resultado. Entretanto, dobrar a quantidade de dados de entrada pode fazer com que a estrutura de dados não possa ser totalmente armazenada na memória de acesso direto disponível no sistema (RAM), e assim precisaríamos trabalhar com memória virtual para armazenar os dados, o que certamente causaria uma limitação severa que acarretaria na perda significativa de performance (disk thrashing).

Contribuição: Observando este cenário, propomos neste trabalho a implementação de duas estruturas de dados, uma para superfícies e outra para 3–variedades, chamadas de CHE (*Compact Half–Edge*), e CHF (*Compact Half–Face*) (Lage *et al.* 2005) respectivamente. Tais estruturas têm como principal característica o balanceamento entre o uso de memória e a eficiência da representação topológica do modelo.

Em outros termos, apresentaremos duas estruturas de dados *escalonáveis*, no sentido que estas podem utilizar, desde que haja disponibilidade física, uma quantidade adicional de memória para obter melhor desempenho no acesso às informações topológicas, ou por outro lado diminuir a performance da estrutura com o objetivo de aumentar a complexidade dos modelos que podem ser representados.

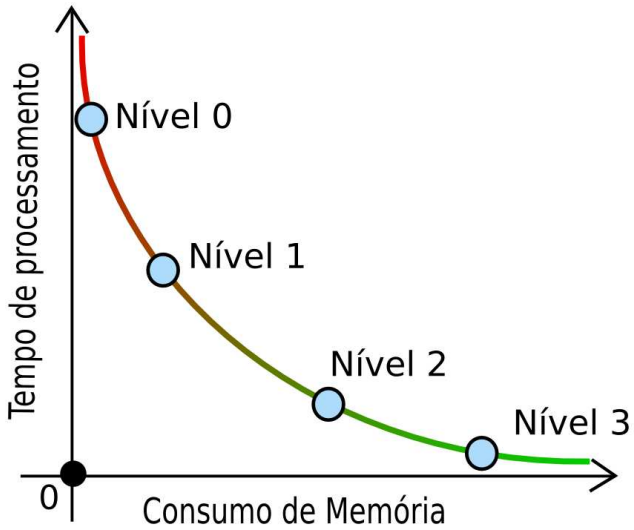

Figura 1.2: Níveis da CHE e da CHF no gráfico *performance X memória*.

Tanto a CHE quanto a CHF são compostas de quatro níveis (figura 1.2), cada um adicionando novas informações ao nível anterior, com o objetivo de

tornar a estrutura mais eficiente na obtenção de informações topológicas. Fica claro no entanto que, a cada novo nível, é preciso alocar uma quantidade maior de memória que no nível anterior.

Na prática, os níveis da estrutura de dados são implícitos ao programador através do uso de herança virtual. *Heranças virtuais* são um recurso da linguagem de programação C++ e, da programação orientada a objetos, que faz com que o programador não se preocupe com qual nível da estrutura ele está trabalhando. O compilador utiliza a seguinte técnica: se um objeto pelo menos uma função virtual, um ponteiro oculto, denominado *v–pointer* é criado. Tal ponteiro aponta para uma tabela global denominada *v–table*. O compilador cria uma *v–table* para cada classe que possui uma ou mais funções virtuais. Na *v–table* são armazenados os endereços de cada função virtual. Durante a execução de uma função virtual, o compilador segue o *v–pointer* do objeto para acessar o elemento apropriado da *v–table* da classe. Assim obtemos uma interface única para cada função da estrutura independe do nível em que estamos trabalhando.

Visão Geral: Dividimos esta dissertação da seguinte forma: no capítulo 2, definimos alguns objetos combinatórios que compõem a base teórica que precisaremos no decorrer do trabalho. No capítulo 3 fazemos uma breve revisão dos trabalhos anteriores em estrutura de dados. No capítulo 4, descrevemos nossa proposta de estrutura de dados escalonável para superfícies, a CHE. No capítulo 5 propomos uma extensão da CHE para variedades de dimensão 3, chamada CHF. No capítulo 6 comparamos o consumo de memória da CHE e da CHF com o de estruturas clássicas da literatura. Por fim, no capítulo 7 apresentamos algumas conclusões e propomos trabalhos futuros.

2
Preliminares

Neste capítulo, revisaremos algumas noções combinatórias básicas relacionadas a complexos simpliciais em dimensão arbitrária, introduziremos as relações topológicas existentes entre as células de um complexo e, por fim, apresentaremos algumas noções básicas de programação genérica.

2.1
Simplexos

Definição 2.1 Simplexo: Um *simplexo* σ de dimensão k, é o fecho convexo de $k+1$ pontos $\{v_0, \ldots, v_k\}, v_i \in \mathbb{R}^m$, em posição geral, isto é, os vetores $v_1 - v_0, v_2 - v_0, \ldots, v_k - v_0$ são linearmente independentes.

Por convenção, chamamos um simplexo de dimensão k de k–simplexo e, quando o contexto for entendido, chamaremos um 0–simplexo de *vértice*, um 1–simplexo de *aresta*, um 2–simplexo de *triângulo* e um 3–simplexo de *tetraedro*(figura 2.1). Os pontos $v_0, \ldots, v_k$ de um k–simplexo σ são chamados de *vértices* de σ.

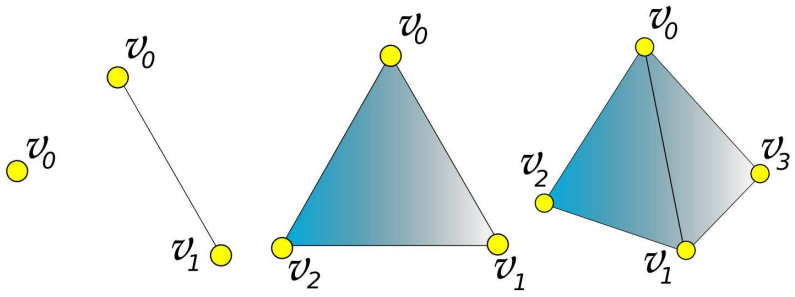

Figura 2.1: Simplexos de dimensão 0, 1, 2 e 3 em $\mathbb{R}^m$.

Dado um k–simplexo σ, denotamos sua dimensão por $dim(\sigma) = k$ e o conjunto dos seus vértices por V_σ.

Definição 2.2 p**–Face:** Um p–simplexo γ gerado a partir de um subconjunto $V_\gamma \subseteq V_\sigma$, dos vértice de um k–simplexo σ, com $p \leq k$, é chamado de uma p*–face* de σ(figura 2.2).

Quando não houver ambigüidade, a dimensão de γ será omitida e diremos apenas que γ é uma *face* de σ. Se γ é face de σ dizemos também que σ é *incidente* a γ e que σ é uma *co–face* de γ.

Um simplexo γ é uma *face própria* de um simplexo σ se $dim(\gamma) < dim(\sigma)$. O simplexo gerado a partir do subconjunto vazio $V_\gamma = \emptyset$ é, por convenção, uma (-1)–face de todo k–simplexo, com $k \geq 0$.

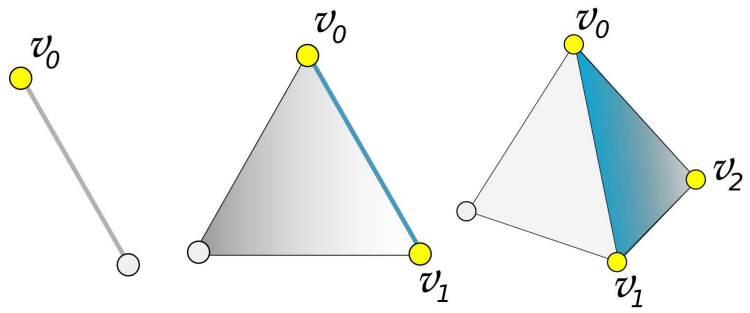

Figura 2.2: Exemplos de faces dos simplexos de dimensão 1, 2 e 3 em $\mathbb{R}^m$

Definição 2.3 Bordo de um Simplexo: O *bordo* de um p–simplexo σ, denotado por $\partial\sigma$, é a coleção de todas as faces próprias de σ(figura 2.3).

O *interior* de um simplexo σ, é definido como $Int(\sigma) = \sigma - \partial\sigma$.

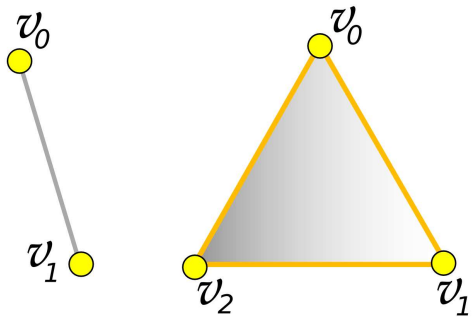

Figura 2.3: Bordo e interior de simplexos de dimensão 1 e 2 em $\mathbb{R}^m$.

2.2 Complexos Simpliciais

Definição 2.4 Complexo Simplicial: Um *complexo simplicial* Σ é um conjunto finito de simplexos tais que:

1. Se $\sigma \in \Sigma$, então todas as faces de σ pertencem a Σ.
2. Se $\sigma, \gamma \in \Sigma$, então $\sigma \cap \gamma$ é uma face própria de σ e γ.

A condição 2 da definição (2.4) impede que existam interseções indevidas entre simplexos de um complexo simplicial.

Por exemplo, 3–simplexos podem ter interseção com outros simplexos apenas em vértices, arestas ou triângulos comuns, 2–simplexos apenas em arestas e vértices comuns, e assim por diante (figuras 2.4 e 2.5).

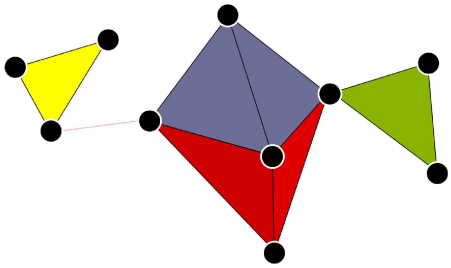

Figura 2.4: Complexo simplicial

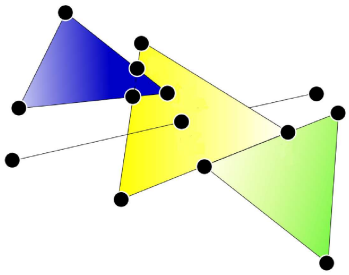

Figura 2.5: Complexo inválido

Definimos a dimensão de um complexo simplicial Σ como o número inteiro $d = max\{dim(\sigma) \mid \sigma \in \Sigma\}$, e dizemos que Σ é um *complexo simplicial d–dimensional*, ou simplesmente de *d–complexo simplicial*.

O *poliedro* de um d–complexo simplicial Σ imerso em $\mathbb{R}^m$, com $0 \leq d \leq m$, denotado por $|\Sigma|$, é o subconjunto de $\mathbb{R}^m$ definido pela união, como conjunto de pontos, de todos os simplexos de Σ. Um *subcomplexo simplicial* de Σ é qualquer subconjunto Σ^* composto por simplexos de Σ tal que Σ^* é também um complexo simplicial.

2.3 Relações entre Simplexos

Definição 2.5 **Estrela:** A *estrela* de um simplexo $\sigma \in \Sigma$, denotada por $\text{star}(\sigma, \Sigma)$, é a união de todos os simplexos $\gamma \in \Sigma$ que são co–face de σ.

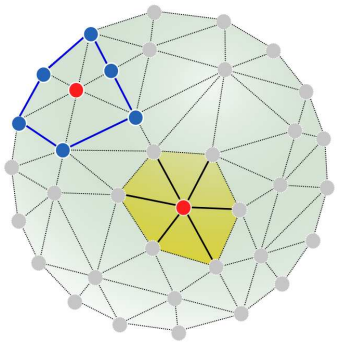

Figura 2.6: Estrela e Elo de vértices de uma esfera.

Definição 2.6 Elo: O *Elo* de um simplexo $\sigma \in \Sigma$, denotado por $\text{link}(\sigma, \Sigma)$, é o conjunto de simplexos $\gamma \in \Sigma$ tais que:

1. γ é face de algum simplexo $\psi \in \text{star}(\sigma, \Sigma)$.
2. $\gamma \notin \text{star}(\sigma, \Sigma)$.

A figura 2.6 ilustra a estrela (em amarelo) e o elo (em azul) de um dos vértices de um 2–complexo em $\mathbb{R}^3$.

Um simplexo $\sigma \in \Sigma$ é chamado de *simplexo topo* se $\text{star}(\sigma, \Sigma) = \{\sigma\}$. Se um d–complexo Σ é tal que todos os seus simplexos topo são d–simplexos, então Σ é um complexo *regular* (figuras 2.8, 2.7).

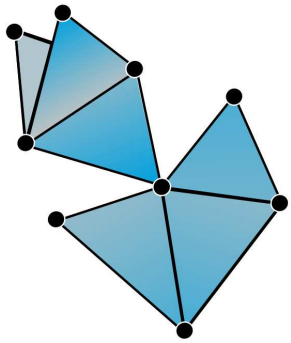

Figura 2.7: 2–complexo regular

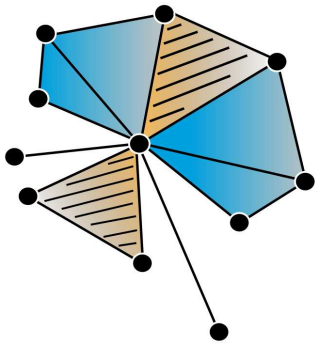

Figura 2.8: 3–complexo não regular

Definição 2.7 Simplexos p–Adjacentes: Dois simplexos σ e γ são *p–adjacentes* quando existe uma p–face comum a eles (figuras 2.10, 2.9).

Em alguns momentos omitiremos o grau de adjacência entre simplexos para tornar a notação mais simples. Por exemplo, dois k–simplexos que compartilham uma $(k-1)$–face e dois vértices incidentes a uma mesma aresta serão chamados apenas de *adjacentes*.

Dois simplexos que não são nem incidentes e nem adjacentes são denominados *disjuntos*.

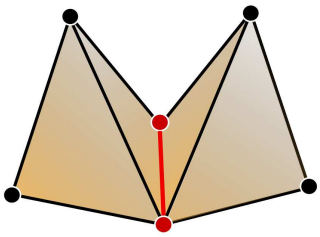

Figura 2.9: Simplexos 1–adjacentes

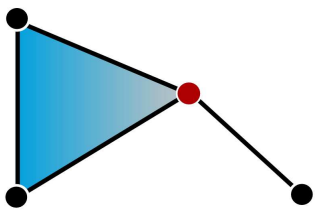

Figura 2.10: Simplexos 0–adjacentes

Definição 2.8 Simplexos h–Conectados: Dois simplexos ψ e γ são *h–conectados* se e somente se existe uma seqüência de simplexos $(\sigma_i)_{i=0}^{n}$ tal que:

1. Dois simplexos consecutivos σ_{i-1}, σ_i são h–adjacentes;
2. ψ e γ são faces de σ_0 e σ_n respectivamente.

A figura 2.11 mostra exemplos de simplexos 0–conectados e 1–conectados.

2.4 Componentes Conexas

Um subcomplexo Σ^* de um complexo simplicial Σ é *h–conexo* se e somente se todos os seu vértices são h–conectados. Um subcomplexo maximal em Σ^*, h–conexo, é chamado *h–componente conexa* de Σ.

A figura 2.11 mostra um complexo simplicial composto de uma 0–componente conexa.

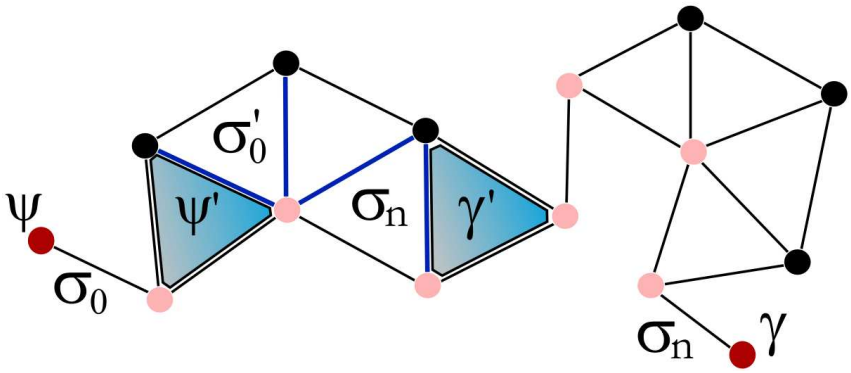

Figura 2.11: ψ e γ são 0–conectados e $\psi^{'}$ e $\gamma^{'}$ são 1–conectados. O complexo simplicial Σ é uma 0–componente conexa.

Definição 2.9 Componente Conexa: Seja Σ^* um subcomplexo simplicial de Σ, dizemos que Σ^* é uma *componente conexa* de Σ se e somente se Σ^* é uma 0–componente conexa de Σ.

2.5 Variedades

Um $(d-1)$–simplexo σ de um d–complexo simplicial Σ é um *$(d-1)$–simplexo variedade*(figura 2.12) se e somente se existem no máximo dois d–simplexos em Σ incidentes a σ. Caso σ não seja um simplexo variedade, o chamamos de *simplexo não–variedade*(figura 2.13).

Definição 2.10 Pseudo–Variedade Combinatória: Um d–complexo Σ, $|\Sigma| \subset \mathbb{R}^m$, é uma *pseudo–variedade combinatória* de dimensão d se, e só se:

1. Σ é um complexo regular $(d-1)$–conectado.
2. Todo $(d-1)$–simplexo $\sigma \in \Sigma$ é um $(d-1)$–simplexo variedade.

Para simplificar a notação, chamaremos uma pseudo–variedade combinatória de dimensão de d de *pseudo–d–variedade.* A figura 2.14 mostra um exemplo de pseudo–2–variedade. Um m–complexo simplicial Σ, imerso em $\mathbb{R}^m$ tem as seguintes propriedades:

1. Σ é uma pseudo–m–variedade.
2. Os simplexos topo incidentes a um $(d-2)$–simplexo σ podem ser ordenados em torno de σ.

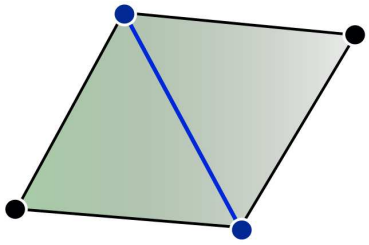

Figura 2.12: Simplexos variedade

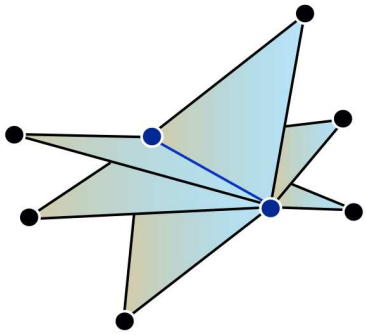

Figura 2.13: Simplexos não–variedade

Observe que se Σ é um d–complexo simplicial imerso em $\mathbb{R}^m$, $m > d$, então Σ não é necessariamente uma pseudo–variedade, pois podem existir muitos d–simplexos na estrela de um $(d-1)$–simplexo $\in \Sigma$. Entretanto, a propriedade 2 continua valendo para todos $(d-1)$–simplexos.

Definição 2.11 Variedade Combinatória: Uma pseudo–k–variedade Σ, $|\Sigma| \subset \mathbb{R}^m$, tal que, para todo vértice $v \in \Sigma$, a star(v, Σ) é homeomorfa a $\mathbb{R}^k$ ou $\mathbb{R}^k_+$ é chamada de *Variedade Combinatória de dimensão k.*

Por simplicidade, chamaremos uma variedade combinatória de dimensão k de k–variedade (figura 2.15).

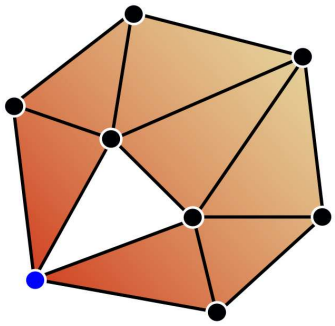

Figura 2.14: Pseudo–2–variedade

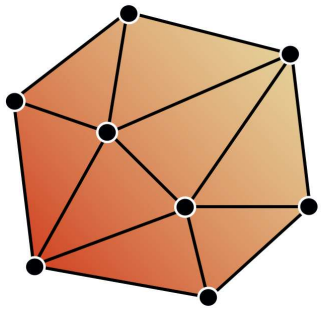

Figura 2.15: 2–variedade

Definição 2.12 Orientabilidade: Seja Σ uma k–variedade combinatória. A orientação de dois k–simplexos adjacentes σ e γ pertencentes a Σ é *coerente* se o $(k-1)$–simplexo ψ que compartilham tem orientação oposta em cada um dos simplexos. A variedade Σ é *orientável* se podemos escolher uma orientação coerente para todos os seus simplexos (figura 2.16).

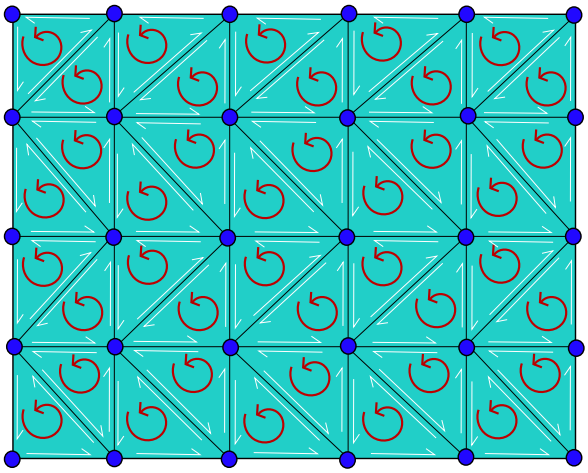

Figura 2.16: 2–variedade combinatória orientada no sentido anti–horário.

Deste ponto em diante, uma k–variedade Σ sempre significará uma k–variedade combinatória orientada, e denotaremos por $\mathbf{n}_k$ o número de k–simplexos existentes em Σ.

2.6 Bordo de Variedades

Definição 2.13 Simplexos de Bordo: Um $(k-1)$–simplexo de uma k–variedade incidente a apenas um k–simplexo é chamado de *simplexo de bordo.*

Todas as faces de um simplexo de bordo também são simplexos de bordo. Os simplexos que não são de bordo são chamados *simplexos interiores.*

Definição 2.14 Bordo de Variedades: O *bordo* de uma k–variedade Σ, denotado por $\partial\Sigma$, é a união de todos os seus simplexos de bordo.

Observe que o bordo de uma k–variedade é uma $(k-1)$–variedade sem bordo. A figura 2.17 ilustra uma 3–variedade e seus simplexos de bordo e interior.

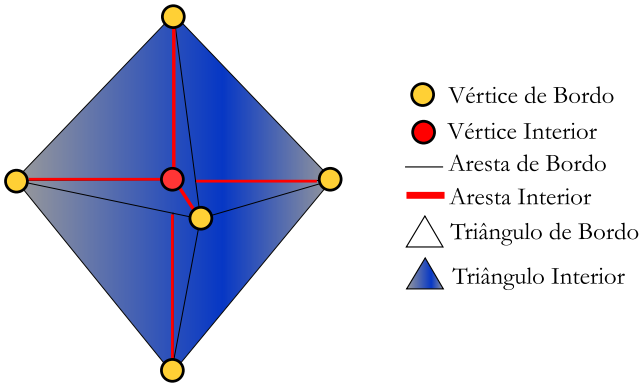

Figura 2.17: Simplexos de bordo e interior de uma 3–variedade.

2.7 Relações Topológicas em Variedades

De acordo com (Floriani and Hui 2003), uma estrutura de dados otimizada deve apresentar soluções eficientes para o cálculo de relações topológicas entre simplexos. Tais operações podem ser representadas através das *funções de resposta*.

Definição 2.15 Funções de resposta às interrogações topológicas: Seja Σ um m–complexo e $\sigma \in \Sigma$ um p–simplexo. As *funções de resposta às interrogações topológicas do tipo* $R_{pq}(\sigma)$ são relações topológicas que retornam, todos os q–simplexos γ de Σ que não são disjuntos de σ.

Por exemplo, $R_{00}(\sigma)$ retorna todos os vértices σ^* tais que σ, σ^* são incidentes a uma mesma aresta de Σ (figura 2.18).

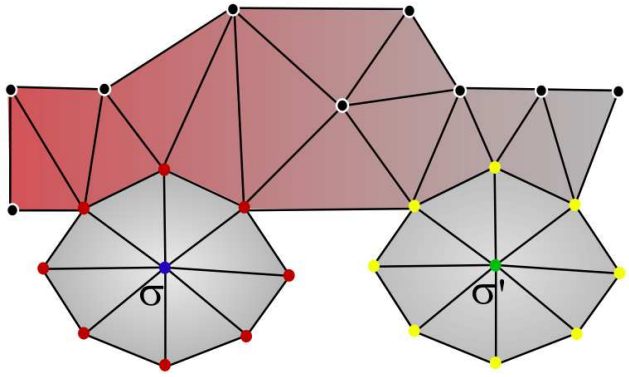

Figura 2.18: $R_{00}(\sigma)$ e $R_{00}(\sigma^{'})$

De maneira mais geral, quando $p < q$, $R_{pq}(\sigma)$ representa o conjunto de q–simplexos pertencentes a star(σ). Quando $p > q$, $R_{pq}(\sigma)$ consiste no conjunto dos q–simplexos que são face de σ. Por fim, quando $p = q$, $R_{pq}(\sigma)$ obtém o conjunto dos p–simplexos adjacentes ao simplexo σ.

A figura 2.19 mostra todos os casos possíveis para a função de retorno de uma aresta em uma 2–variedade.

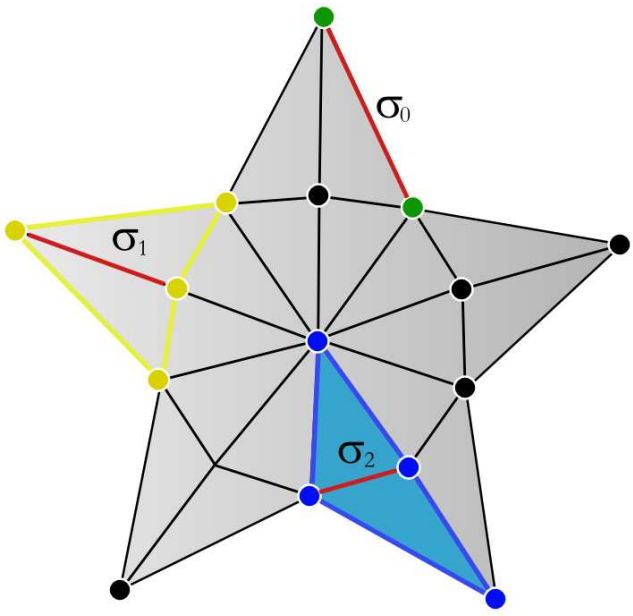

Figura 2.19: $R_{10}(\sigma_0)$, $R_{11}(\sigma_1)$ e $R_{12}(\sigma_2)$

2.8 Programação Genérica

Utilizaremos conceitos de programação genérica na construção das estruturas de dados propostas neste trabalho. O conceito mais importante que abordaremos é o de contêiner. Em termos gerais, um *contêiner* é um objeto que contém e pode organizar outros objetos. Existem vários tipos de contêineres conhecidos, que permitem trabalhar com os objetos de diversas maneiras. Cada tipo de contêiner deve fornecer recursos e procedimentos que nos permitam manipular os dados armazenados. Neste trabalho, utilizamos a implemetação de contêineres genéricos da biblioteca STL (*Standart Template Library*) da linguagem C++.

A seguir faremos uma breve descrição alguns tipos de contêiner:

- **Vector:** Um vetor é um *array* unidimensional cujos dados podem ser acessados aleatoriamente.
- **Deque:** Uma deque é uma fila com dois finais, ou seja, permite adicionar e remover elementos de ambas as extremidades. Uma vantagem de uma deque é que esta suporta o acesso aleatório de elementos.
- **Queue:** Um contêiner queue é uma fila de elementos que permite que adicionemos objetos em seu final, e que elementos do início sejam

removidos. Não podemos acessar elementos de uma fila aleatóriamente, assim como não é possível seu percorrimento.

- **Stack:** Uma stack é uma construção de programação que opera de forma análoga a uma pilha de pratos: elementos são inseridos e removidos do topo da pilha.
- **List:** Um contêiner List é uma lista de elementos e pode ser simplesmente ou duplamente encadeada. No caso de ser simplesmente encadeada, cada elemento da lista é ligado ao próximo elemento do conjunto. Já no caso de uma lista duplamente encadeada, cada elemento é ligado ao elemento anterior e posterior na lista. Uma lista duplamente encadeada suporta a remoção e inserção de elementos em tempo constante, independente do seu tamanho. Já numa lista simplesmente encadeada apenas a inserção é realizada em tempo constante.
- **Heap:** Uma priority queue é uma fila cujos elementos são ordenados de acordo com um critério de prioridade pré–definido.
- **Set:** Um set se assemelha a um conjunto matemático de objetos, sem repetição de elementos. Operações como união, interseção, diferença, etc são alguns dos recursos proporcionados pelo contêiner.
- **Multiset:** Um multiset é também tratado como um conjunto matemático, mas não existe a restrição de que os elementos sejam únicos.
- **Map:** Um contêiner map permite armazenar dados em pares chave/valor onde a chave é utilizada para acessar o valor associado. Em um contêiner map, cada chave é única.
- **Multimap:** Um contêiner multimap é análogo a um map, mas permite a associação de mais de um valor por chave.

Detalharemos as características técnicas do contêiner map já que ele será utilizado em vários algoritmos durante a construção das estruturas CHE e CHF.

Contêiner Map: Um map é um contêiner que nos permite associar um objeto conhecido como chave, a outro objeto chamado de valor. O objeto chave é utilizado como identificador do objeto valor, e assim não é permitida a utilização de chaves iguais para dois ou mais valores. O contêiner map é, internamente, uma *RB–Tree (Red–Black Tree)*. As *Red–Black Trees* são árvores binárias balanceadas de forma a garantir que operações como a busca, remoção e inserção de nós sejam feitas, no pior caso, em tempo $log(N)$, onde N é o número de nós que compõem a árvore.

Em cada nó de uma *RB–Tree* armazenamos uma cor: vermelho (*red*) ou preto (*black*). De acordo com as regras impostas na coloração dos nós, *RB–Trees* garantem que nenhum caminho, de uma mesma raiz até uma folha, é mais de duas vezes maior que qualquer outro.

Ainda, os nós de uma *RB–Tree* contém as informações sobre a chave, o filho à esquerda, o filho à direita e o pai. Se um filho ou o pai de um nó não existir, o ponteiro do campo correspondente aponta para *null*. Supondo que cada informação armazenada em um nó da árvore ocupe 4 bytes de memória, seu custo total é de 20N bytes, onde N é o número de nós que compõem a árvore.

Uma árvore binária é uma *RB–Tree* se satisfaz as seguintes propriedades:

1. Todo nó é vermelho ou preto.
2. Toda folha(*null*) é preta.
3. Se um nó é vermelho, então seus dois filhos são pretos.
4. Todo caminho simples de um dado nó até uma folha contém o mesmo número de nós preto.

A figura 2.20 mostra o exemplo de uma *RB–Tree*. Repare que cada nó da árvore é vermelho ou preto, toda folha (*null*) é preta, os filhos de um nó vermelho são pretos e todo caminho simples de um nó a uma folha contém o mesmo número de nós pretos.

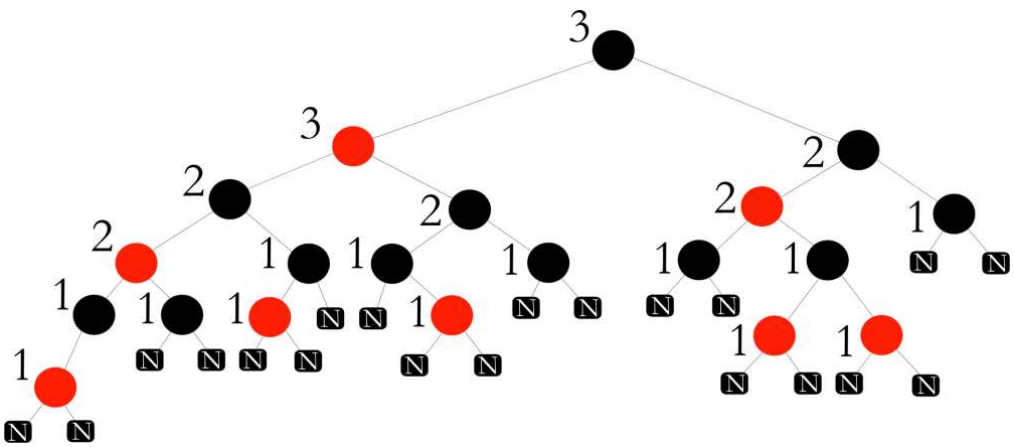

Figura 2.20: Exemplo de uma *RB–Tree*.

Podemos encontrar um estudo mais detalhado sobre *RB–Trees* em (Cormen *et al.* 1990).

3
Trabalhos Anteriores

Neste capítulo faremos uma revisão das principais estruturas de dados topológicas existentes na literatura. Classificaremos os trabalhos de acordo com a classe de objetos que estes podem representar. Na seção 3.1 citaremos estruturas para superfícies em $\mathbb{R}^3$, na seção 3.2 estruturas desenvolvidas para a representação de variedades de dimensão 3, na seção 3.3 estruturas para variedades de dimensão n e, por fim, em 3.4 estruturas para não–variedades.

Alguns trabalhos citados neste capítulo serão descritos com mais detalhe, já que, no capítulo 6, os compararemos com as estruturas CHE e CHF.

3.1
Estruturas de Dados para Superfícies em $\mathbb{R}^3$

A estrutura *Winged–Edge* de (Baumgart 1975) foi um dos primeiros trabalhos propostos para a representação de superfícies em $\mathbb{R}^3$. Muitas modificações desta estrutura foram desenvolvidas com o objetivo de aumentar a variedade de objetos a serem modelados. Entre tais modificações, os trabalhos mais importantes são o de (Braid *et al.* 1980), que introduziu o conceito de *loop*, e o de (Mäntylä 1988), que criou a estrutura de dados *Half–Edge*.

Em seguida, (Guibas and Stolfi 1985), com a *Quad–Edge*, também generalizaram a *Winged–Edge*, permitindo a modelagem de variedades de dimensão 2 não orientáveis. Ainda, (Castelo *et al.* 1992) (Lopes 1996) definiram a *Handle–Edge*: uma estrutura de dados que representa explicitamente o bordo da superfície combinatória, o que permite uma descrição direta da conectividade de suas células de bordo.

Mais recentemente, (Rossignac *et al.* 2001) propuseram uma estrutura de dados extremamente concisa chamada *Corner–Table* que utiliza apenas vetores de inteiros e um conjunto de regras para representar superfícies trianguladas.

A primeira estrutura de dados que usou o conceito de escalabilidade, foi batizada como *Directed–Edges*, e proposta por (Campagna *et al.* 1998).

Escolhemos as estruturas *Handle–Edge*, *Corner–Table* e *Directed–Edges* para serem descritas com mais detalhes, já que em cada delas observamos características semelhantes às características da CHE.

Handle–Edge A *Handle–Edge* é uma extensão da estrutura de dados *Half–Edge* desenvolvida com o objetivo de se obter uma implementação eficiente dos operadores de alça (Lopes 1996). Para tal a *Handle–Edge* propõe uma representação explícita das curvas de bordo da superfície. A estrutura trabalha com o conceito de *half–edge*, definido por (Mäntylä 1988). Uma *half–edge* é uma aresta orientada em um triângulo da malha. A implementação originalmente proposta para a *Handle–Edge* utiliza ponteiros, e cada entidade é representada por um tipo de nó. A seguir descreveremos cada nó da estrutura de dados:

- **Nó Surface:** O *nó surface* representa a instância de uma superfície. Este nó é conectado aos primeiros elementos das listas duplamente encadeadas de vértices, faces, arestas e curvas de bordo que compõem a superfície (figura 3.1).

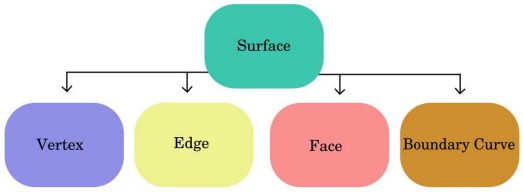

Figura 3.1: Nó surface.

- **Nó Face:** O *nó face* representa um triângulo (que também chamaremos de *face*) da superfície. Toda face é delimitada por um ciclo de *half–edges*. Cada face é ligada a uma de suas *half–edges* incidentes e à superfície da qual faz parte. As faces são elementos de uma lista composta por todas as faces da superfície (figura 3.2).

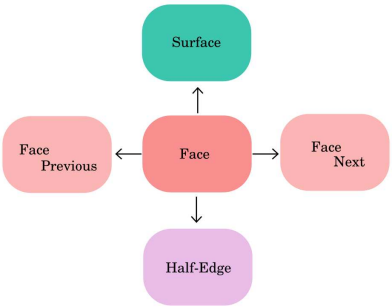

Figura 3.2: Nó face

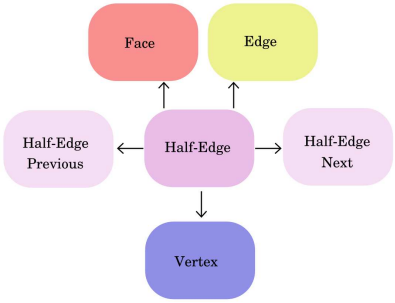

Figura 3.3: Nó half–edge

- **Nó Half–Edge:** Um *nó half–edge* representa, explicitamente, a orientação de uma aresta na face a que pertence, induzida a partir da orientação da face (figuras 3.4, 3.5). Cada nó *half–edge* é ligado à face incidente e ao vértice inicial da *half–edge*. Ainda, guardamos referências

da aresta incidente e das *half–edges* seguinte e anterior (chamadas de *next* e *previous*) no ciclo de *half–edges* da face incidente (figura 3.3).

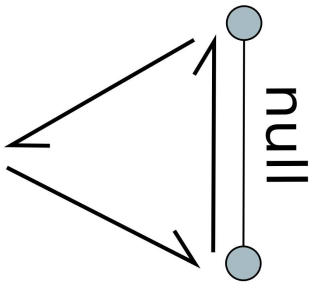

Figura 3.4: Half–edge de uma aresta de bordo.

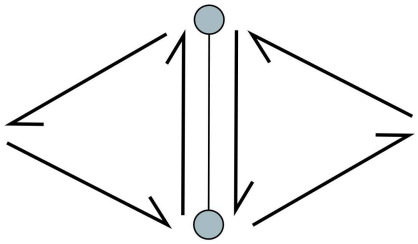

Figura 3.5: Half–edges de uma aresta de interior.

- **Nó Boundary Curve:** Um *nó boundary curve* representa uma componente conexa do bordo da superfície através de uma de suas arestas incidentes (figura 3.6). O nó *boundary curve* faz parte de uma lista composta por todas as curvas de bordo da superfície.

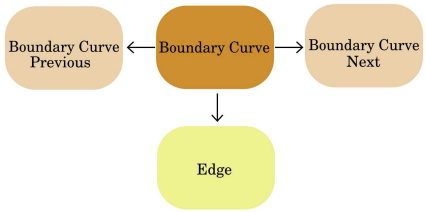

Figura 3.6: Nó Boundary Curve.

- **Nó Edge:** Um *nó edge* pode representar tanto uma aresta de bordo quanto uma aresta de interior (figura 3.7). Cada aresta é ligada a um trio ordenado $(HE_1, HE_2, \partial S)$ composto de dois nós *half–edge* e um nó *boundary curve*, e construído de acordo com as seguintes regras:
  - Se a aresta é de interior, o trio tem a configuração $(HE_1, HE_2, \emptyset)$, onde HE_1 e HE_2 são as duas *half–edges* incidentes a aresta.
  - Se a aresta é de bordo, a configuração do trio é $(HE_1, \emptyset, \partial S)$, onde HE_1 é sua única *half–edge* e ∂S é a curva de bordo a que pertence a aresta.

  O nó edge também faz parte de uma lista composta por todas as arestas de uma superfície.
- **Nó Vertex:** O *nó vertex* pode representar tanto vértices de bordo quanto vértices de interior (figura 3.8). Cada vértice é associado a um par ordenado de *half–edges* que obedece as seguintes regras:

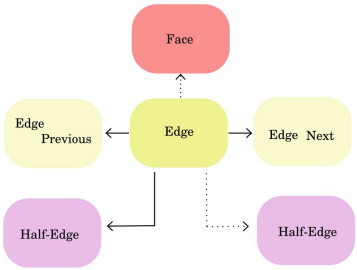

Figura 3.7: Nó edge

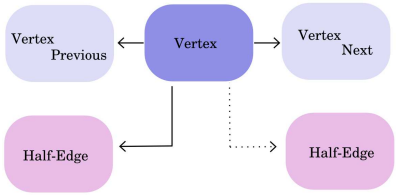

Figura 3.8: Nó vertex

– Se o vértice for de interior, a primeira *half–edge* armazena é uma das *half–edges* incidentes ao vértice, e a segunda aponta para *null* (figura 3.9).
– Se o vértice é um vértice de bordo, as duas *half–edges* de bordo incidentes são armazenadas. A primeira corresponde a aresta que tem o vértice como ponto inicial, e a segunda a aresta que tem o vértice como ponto final (figura 3.10).

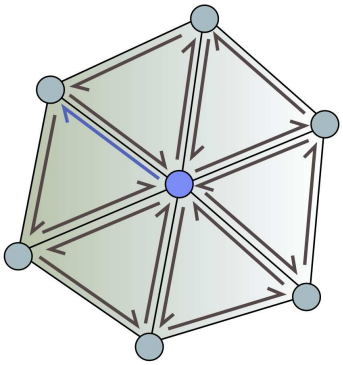

Figura 3.9: Vértice de interior.

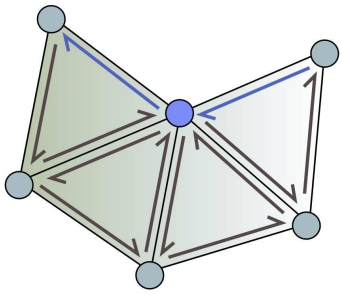

Figura 3.10: Vértice de bordo.

Corner–Table A *Corner–Table* é uma estrutura de dados concisa para a representação de malhas triangulares que trabalha com um conceito equivalente ao de *half–edge*: o corner. Um *corner* é a associação de um triângulo a um de seus vértices, enquanto uma *half–edge* é a associação de um triângulo e uma de suas arestas. As duas definições são equivalentes pois, para cada *corner* em um triângulo, sempre existe uma única *half–edge* oposta a ele (figura 3.11).

Cada triângulo é representado por três *corners* consecutivos, indexados segundo sua orientação. Logo, uma malha com $\mathbf{n}_2$ triângulos terá $3\mathbf{n}_2$ *corners* e, por definição, o triângulo i será representado pelos *corners* $3i$, $3i+1$, $3i+2$.

Desse modo, podemos escrever expressões simples para o cálculo dos

corners "vizinhos" a um *corner* de índice c no triângulo $\lfloor c/3 \rfloor$ [1], chamados de *next* de c e *previous* de c (denotados por $next(c)$ e $prev(c)$) [2]:

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

Os atributos geométricos da *Corner–Table* são guardados em um vetor chamado G[]. Para cada vértice da malha, associamos atributos como coordenadas, vetores normal etc. A dimensão de G[] é igual a $\mathbf{n}_0$.

As informações topológicas carregadas pela estrutura, ficam armazenadas em dois vetores de inteiros batizados por V[] e O[], ambos de dimensão $3\mathbf{n}_2$. O vetor V[] armazena o índice do vértice associado a um determinado corner. Para construir o vetor O[] temos que definir o conceito de *corner* oposto. Diremos que dois *corners* de índices c e o são *opostos* se $\mathsf{V}[next(c)] = \mathsf{V}[prev(o)]$ e $\mathsf{V}[prev(c)] = \mathsf{V}[next(o)]$ (figura 3.12). O vetor O[] guarda o *corner* oposto de cada corner.

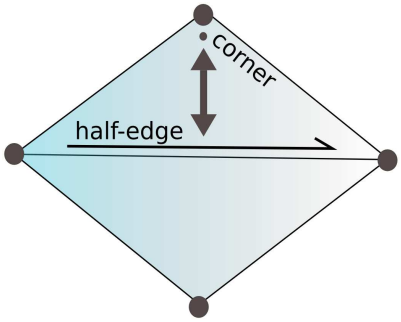

Figura 3.11: Relação corner/half–edge.

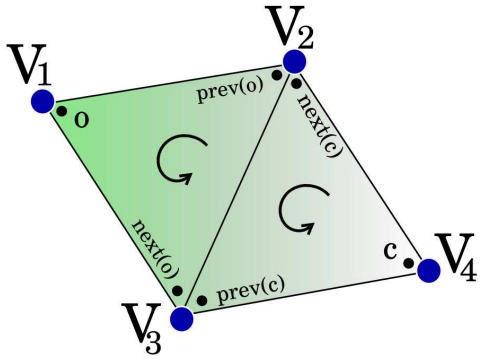

Figura 3.12: Corners opostos c e o.

Apenas com as tabelas V[] e O[] e as duas regras aritméticas descritas é possível atravessar eficientemente os triângulos de uma malha representada pela estrutura. Assim a vizinhança de um vértice pode ser obtida a partir de um de seus *corner* incidente. Porém, a localização de um *corner* a partir do índice de um vértice tem custo linear já que é preciso atravessar a tabela V[] a sua procura. Uma solução simples para este problema, proposta por (Vieira 2003), é armazenar, para cada vértice, o índice de um de seus *corners* incidentes.

Directed–Edges A Estrutura de dados *Directed–Edges* é uma extensão escalonável da *Half–Edge*. Existem 3 níveis chamados de *full–sized*, *medium–sized* e *small–sized*. A estrutura é baseada no conceito de *directed–edge*. Uma

[1] O símbolo $\lfloor * \rfloor$ representa a parte inteira do valor de $*$. Ou seja, um *corner* c pertence ao triângulo de índice igual a parte inteira da divisão de c por 3.

[2] O símbolo $a\%b$ nas expressões dos *corners next* e *previous* representa o resto da divisão de a por b. Neste caso estamos trabalhando com o resto da divisão por 3.

directed–edge e^d de uma aresta e é uma ocorrência de e no bordo de um de seus triângulos incidentes. Vale observar, que o conceito de *directed–edge* é também equivalente aos conceitos de *half–edge* e *corner* utilizados pelas estruturas apresentadas anteriormente (figura 3.13).

Na estrutura *Directed–Edges*, triângulos e arestas não são explicitamente representados, entretanto são implicitamente relacionados com suas *directed–edges* incidentes: o triângulo com índice i, denotado por f^i, é descrito pelas *directed–edges* $3i$, $3i+1$ e $3i+2$, que formam o bordo orientado de f^i, já uma aresta geométrica é descrita pelo par de *directed–edges* incidentes.

Para cada *directed–edge*, podemos armazenar referências para seus vértices, que chamamos de v^a e v^b, para sua *directed-edge* vizinha, isto é aquela incidente à mesma aresta geométrica, denotada por e^{ng}, e para as outras *directed–edges* no bordo do triângulo incidente, identificadas por e^{pv} e e^{nx} (figura 3.14). A memória utilizada para armazenar algumas dessas referências pode ser substituída por uma quantidade constante de cálculos adicionais.

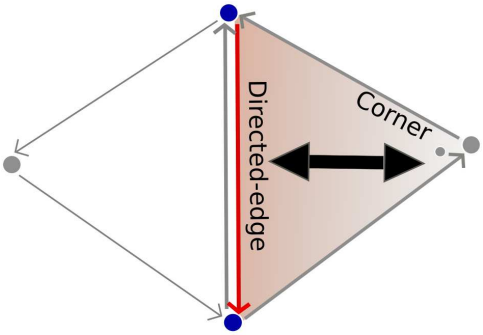

Figura 3.13: Relação corner/directed–edge.

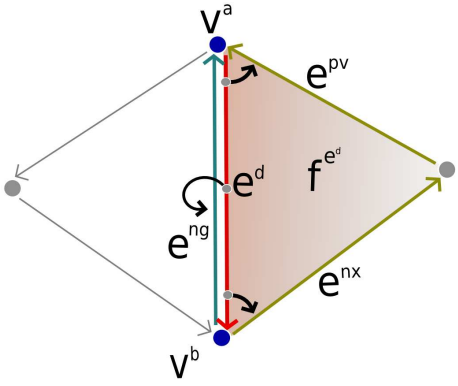

Figura 3.14: Informações armazenadas pela directed–Edge e^d.

Os níveis da estrutura codificam, para cada *directed–edge*:

- *Small–Sized*: O vértice final v^b e a *directed–edge* vizinha e^{ng}.
- *Medium–Sized*: O vértice final v^b, a *directed–edge* vizinha e^{ng} e a *directed–edge* anterior e^{pv}, seguindo a orientação do triângulo incidente f^i.
- *Full–Sized*: Os vértices inicial e final v^a e v^b, a *directed–edge* vizinha e^{ng} e as *directed–edges* anterior e posterior e^{nx} e e^{pv}, seguindo a orientação do triângulo incidente f^i.

Para otimizar a estrutura podemos armazenar, para cada vértice, uma de suas *directed–edges* incidentes. Ainda, podemos representar arestas e vértices não variedade armazenando uma *directed–edge* incidente para cada componente de bordo da estrela do simplexo não variedade.

3.2 Estruturas de Dados para 3–Variedades

Existem várias estruturas de dados desenvolvidas para a representação do domínio de uma 3–variedades através de uma decomposição simplicial. A abordagem descrita em (Guibas and Stolfi 1985) foi estendida por (Dobkin and Laszlo 1989), através de uma nova proposta chamada *Facet–Edge*, que tinha como objetivo representar complexos celulares que são subdivisões de uma esfera tridimensional. Posteriormente, o conceito de *half–edge* foi estendido por (Lopes and Tavares 1997), com a estrutura de dados *Handle–Face*, que representava explicitamente a superfície de bordo de 3–variedades, permitindo o controle de sua topologia.

A seguir, estudaremos com mais detalhes a estrutura de dados *Handle–Face* pois esta apresenta características bastante parecidas com as da CHF.

Handle–Face A *Handle–Face* é uma extensão da *Handle–Edge* para variedades de dimensão 3 que utiliza o conceito de *half–face*. Uma *half–face* é um 2–simplexo dotado de uma orientação induzida por um de seus 3–simplexos incidentes. A implementação da *Handle–Face* utiliza ponteiros e cada entidade é representada por um nó.

O grande diferencial da estrutura é a introdução do *nó boundary surface*, que nos dá uma representação explícita para o bordo da 3–variedade. Quando a superfície de bordo é explicitamente representada, o acesso às relações de incidências e adjacências nas faces de bordo, arestas de bordo e vértices de bordo é otimizado, permitindo uma implementação eficiente dos operadores de alça.

- **Nó 3–Manifold:** Um *nó 3–manifold* é representado por uma lista de superfícies (que são bordo de volumes), uma lista de superfícies de bordo, uma lista de faces de interior, uma lista de aresta de interior, e por fim uma lista de vértices de interior (figura 3.15).

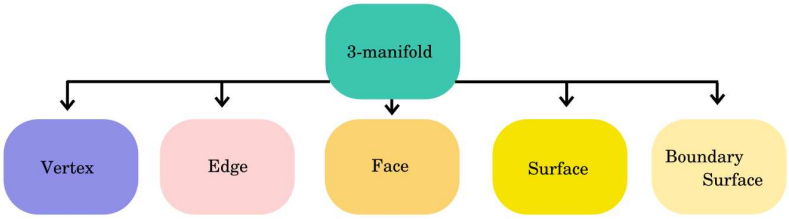

Figura 3.15: Nó 3–Manifold.

- **Nó Surface:** Um *nó surface* corresponde a um sólido da variedade (tetraedro). Nele guardamos referências para a lista de *half–faces*, a lista de arestas e a lista de vértices do sólido. Ainda o nó aponta para a 3–variedade a que pertence o sólido (figura 3.16).
- **Nó Boundary Surface:** Um *nó boundary surface* corresponde a uma componente conexa da superfície de bordo da 3–variedade. Cada superfície é representada pelos seus simplexos incidentes. São armazenadas referências para as listas de faces, arestas e vértices da componente de bordo. O nó aponta para a 3–variedade a que pertence (figura 3.17).

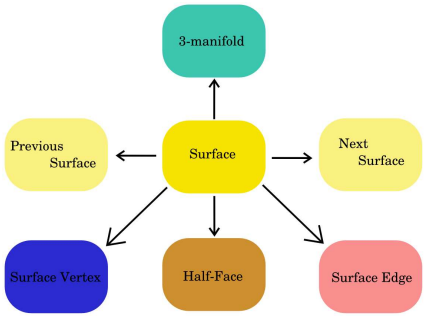

Figura 3.16: Nó Surface.

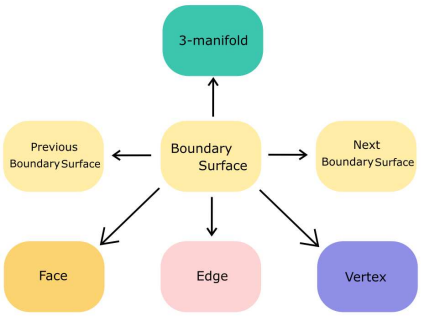

Figura 3.17: Nó Boundary Surface.

- **Nó Face:** Um *nó face* pode representar tanto uma face de interior quanto uma face de bordo (figura 3.18).

  No caso de uma face de interior, o nó é conectado às suas duas *half–faces* incidentes. Caso contrário, o nó representa uma face de bordo e é conectado a sua única *half–face* e a superfície de bordo que contém a face.

- **Nó Half–Face:** Um *nó half–face* representa uma *half–face* de uma célula de dimensão 3, através de seu ciclo de *half–edges*. Ainda, ele armazena informações sobre a face e o sólido incidentes à *half–face* (figura 3.19).

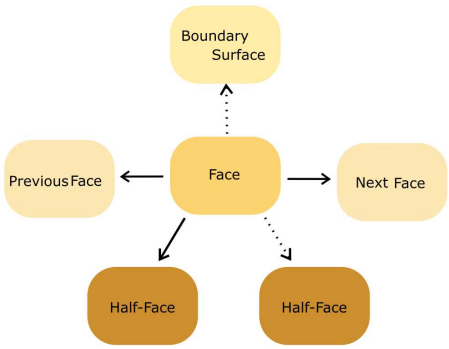

Figura 3.18: Nó Face.

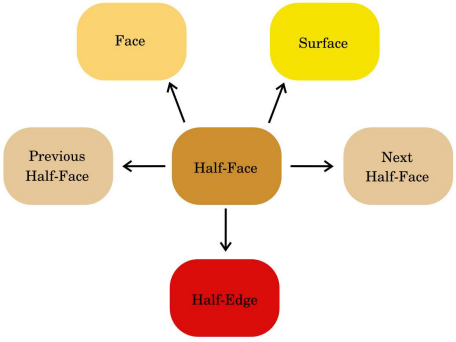

Figura 3.19: Nó Half–Face.

- **Nó Edge:** Um *nó edge* pode corresponder a uma aresta de interior ou a uma aresta de bordo (figura 3.20).

  Se a aresta for de bordo, o nó tem uma representação direta das duas *half–edges* de bordo correspondentes a esta aresta. Por outro lado, se a aresta for interior à variedade, armazenamos apenas uma de suas *half–edges* incidentes para que sejamos capazes de descobrir o ciclo de *half–edges* em torno de uma aresta.

- **Nó Surface Edge:** O *nó surface edge* representa uma aresta na superfície de uma célula de dimensão 3. Cada aresta superfície tem duas *half–edges* incidentes e uma referência para a aresta correspondente da 3–variedade. Ainda, armazenamos as *half–edges* oposta e radial de cada *half–edges* incidentes (figura 3.21).

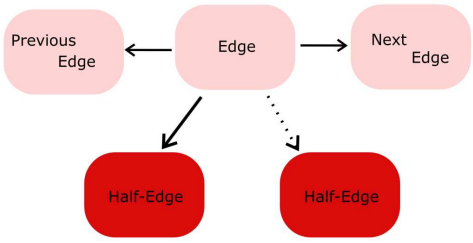

Figura 3.20: Nó Edge.

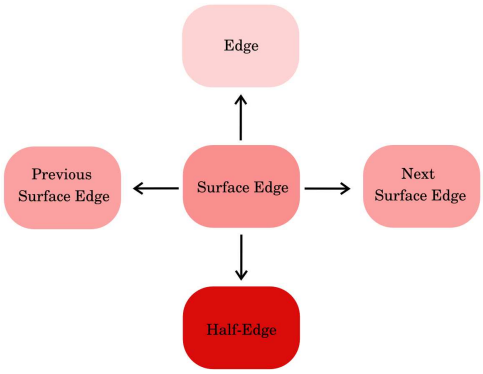

Figura 3.21: Nó Surface Edge.

- **Nó Half–Edge:** Um *nó half–edge* armazena referências diretas para as *half–edges next* e *previous*, para o vértice inicial, para a *half–face* incidente e para a aresta incidente de uma *half–edge* (figura 3.22).

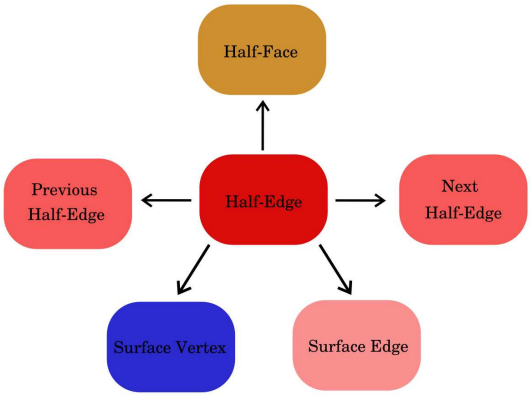

Figura 3.22: Nó Half–Edge.

- **Nó Vertex:** O *nó vertex* representa um vértice da variedade e armazena uma lista com todos os vértices superfície incidentes a ele. Se o vértice estiver no bordo da 3–variedade armazenamos também uma de suas *half–edges* de bordo incidentes (aquela cujo ponto inicial é o vértice) (figura 3.23).

- **Nó Surface Vertex:** O *nó surface vertex* tem uma ligação direta com o vértice original da 3–variedade. Ainda é armazenada uma *half–edge* da superfície que tem o vértice como ponto inicial para que seja possível o cálculo eficiente da estrela na superfície correspondente (figura 3.24).

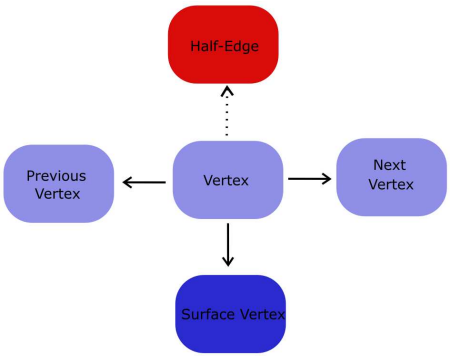

Figura 3.23: Nó Vertex.

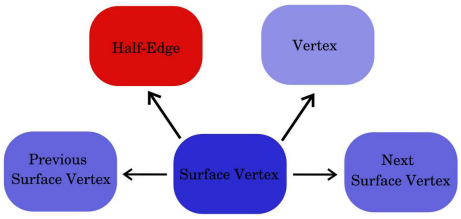

Figura 3.24: Nó Surface Vertex.

A figura 3.25 representa uma visão global do relacionamento entre os nós da *Handle–Face*.

3.3 Estruturas de Dados para n–Variedades

De acordo com (Brisson 1993), dentre todas as estruturas de dados citadas anteriormente, apenas a *Quad–Edge* e a *Facet–Edge* não representam explicitamente cada célula da estrutura combinatória. Por outro lado, a *Winged–Edge*, a *Half–Face* e a *Handle–Edge* são exemplos de estruturas de dados que fazem tal representação explícita das células.

Generalizando a idéia da *Quad–edge* e da *Facet–Edge*, alguns trabalhos significativos foram introduzidos com o objetivo de representar n–variedades. Entre eles, as estruturas *Cell–Tuple* de (Brisson 1993), *the n–generalized Maps for simplicial quasi–manifolds* de (Lienhardt 1994) e *Hypermaps* de (Bertrand and Dufourd 1994) merecem ser citadas.

Um outro exemplo de estrutura, independente da dimensão, é a *Indexed data structure with adjacencies* proposta por (Paoluzzi *et al.* 1993).

Apresentaremos também com mais detalhes a estrutura de dados *Indexed data structure with adjacencies* pois, suas características bastante parecidas com as da CHF.

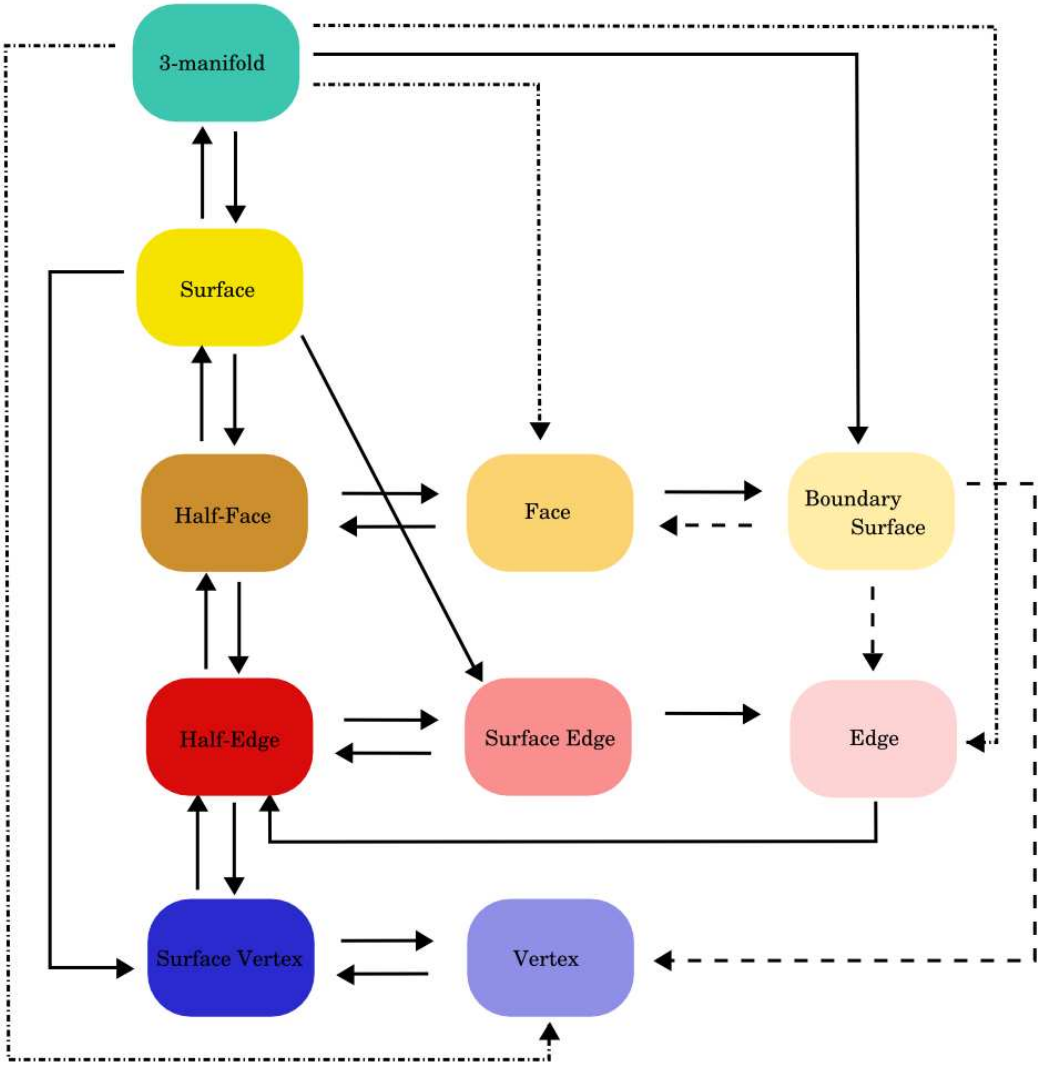

Figura 3.25: Nós da Handle–Face.

Indexed Data Structure with Adjacencies A *indexed data–structure with adjacencies*, ou simplesmente *Ia*, é uma estrutura de dados compacta para representação de variedades de dimensão arbitrária. A estrutura representa apenas o conjunto dos simplexos de dimensão máxima no complexo simplicial, e as relações de adjacência entre tais simplexos.

Na implementação, a *Ia* utiliza um vetor de tamanho $\mathbf{n}_0$ para a representação das coordenadas de cada vértice, e um par de vetores, cada um de tamanho $d \cdot \mathbf{n}_{d\text{-}1}$, para a armazenar os vértices e relação topológica entre as $(d-1)$–faces dos d–simplexos.

Observamos facilmente que as relações de bordo podem ser recuperadas eficientemente, entretanto a estrutura de dados não suporta navegação eficiente pelas arestas e vértices da malha.

3.4 Estruturas de Dados para não–Variedades

A modelagem de não–variedades é fundamental em muitas aplicações. Por exemplo, em sistemas de modelagem, precisamos ter uma representação eficiente de não–variedades já que o conjunto das variedades não é fechado para operadores booleanos.

A primeira proposta de estrutura de dados para este tipo de objeto foi feita por (Weiler 1986), com a estrutura *Radial–Edge*. Em seguida, várias alterações da estrutura foram desenvolvidas com objetivo de adaptá-la a novas aplicações. Algumas contribuições substanciais para a representação de não–variedades são os trabalhos de (Wu 1989, Wu 1992, Yamaguchi and Kimura 1995, Gursoz *et al.* 1990, Rossignac and O'Connor 1990, Cavalcanti *et al.* 1997, Lee and Lee 2001).

Ainda, (Pesco *et al.* 2003) propuseram a estrutura *Handle–Cell*, uma extensão da *Handle–Face*, que faz o tratamento de 2–complexos celulares gerais. Por fim, (Nonato *et al.* 2001) propuseram outra extensão para a *Handle–Face* para tratar casos especiais de singularidade, a *Singular Handle–Face*.

Mais recentemente (Floriani and Hui 2003) apresentaram uma estrutura bastante concisa para a manipulação de não–variedades chamada *NMIA*, que é uma extensão da *Indexed data structure with adjacencies*.

Apresentaremos também com mais detalhes a estrutura de dados *NMIA* pois, no trato de 3–variedades, suas características bastante parecidas com as da CHF.

NMIA A *Non–Manifold Indexed data structure with adjacencies* (*NMIA*) é uma estrutura de dados para a representação de não–variedades de dimensão 3. Sua proposta é representar os simplexos topo de dimensões 1 e 2, chamados de *wire–edges* e *dangling–faces* (figuras 3.26 e 3.27). Mais ainda, é preciso que a estrutura possa descrever as estrelas de um vértice ou de uma aresta nos casos em que estas contêm mais de uma componente conexa. Resumindo, podemos agrupar as singularidades tratadas pela estrutura em quatro casos:

1. Existência de vértices cuja estrela é formada por mais de uma componente conexa, chamados de *nm–vértice* (figura 3.26).
2. Existência de *wire–edges* (figura 3.26).
3. Existência de arestas cuja estrela é formada por mais de uma componente conexa, chamadas de *nm–aresta* (figura 3.27).
4. Existência de *dangling–faces* (figuras 3.26 e 3.27).

Para trabalhar com os casos de singularidade listados, um novo grupo de *funções de resposta* precisa ser definido: as relações $R_{*clusters}$. A função $R_{0clusters}$

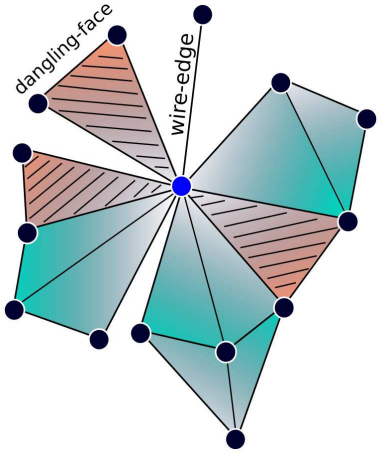

Figura 3.26: Estrela de um *nm–*vértice não-variedade

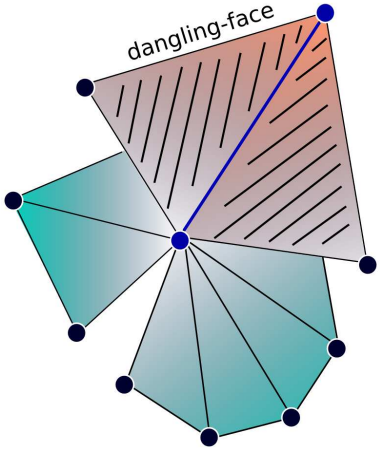

Figura 3.27: Estrela de uma *nm–*aresta não-variedade

retorna, para um vértice v, o conjunto de simplexos topo formado pelos representantes de cada componente conexa da estrela de v. A função $R_{2clusters}$ retorna, para cada aresta e_i de uma *dangling–face*, o grupo de simplexos topo formado pelos representantes de cada componente conexa da estrela de e_i, ordenados no sentido anti–horário. A função $R_{3clusters}$ é análoga a $R_{2clusters}$ para as arestas de um tetraedro .

As informações topológicas armazenadas pela estrutura de dados são:

- Para cada *tetraedro*: as relações R_{30}, R_{33} e $R_{3clusters}$.
- Para cada *dangling–face*: a relação R_{20} e $R_{2clusters}$.
- Para cada *wire–edge*: a relação R_{10}.
- Para cada *vértice*: a relação $R_{0clusters}$.

Assim, dado um complexo simplicial Σ a *NMIA* codifica todos os vértices, *wire–edges*, *dangling–faces* e tetraedros de Σ. A estrutura de dados não representa explicitamente as arestas e faces do complexo simplicial.

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).

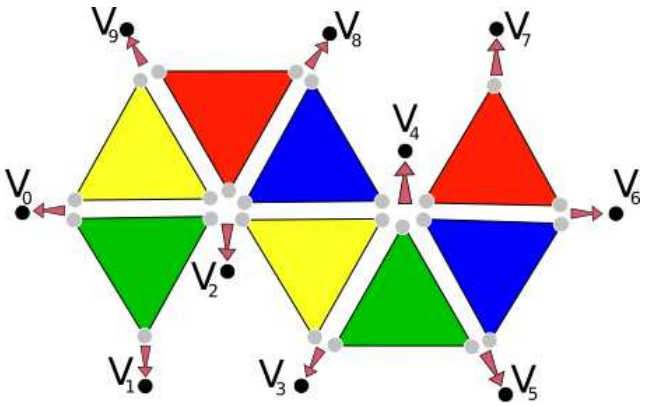

Figura 4.1: Nível 0 da CHE: Sopa de Triângulos.

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).

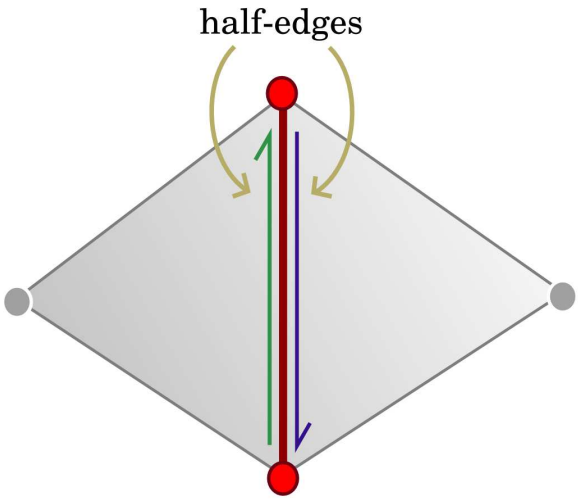

Figura 4.2: Half–edges de uma aresta em seus triângulos incidentes.

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:

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

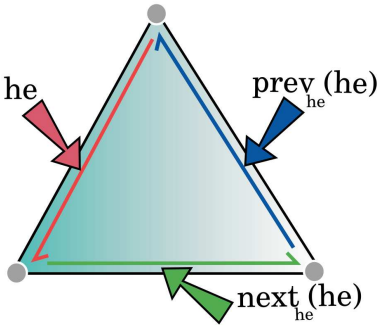

Figura 4.3: Half–edges $\text{next}_{\mathsf{he}}(\mathsf{he})$ e $\text{prev}_{\mathsf{he}}(\mathsf{he})$.

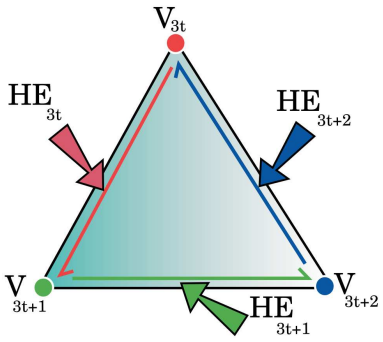

Figura 4.4: Relação entre vértices e half–edges.

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.

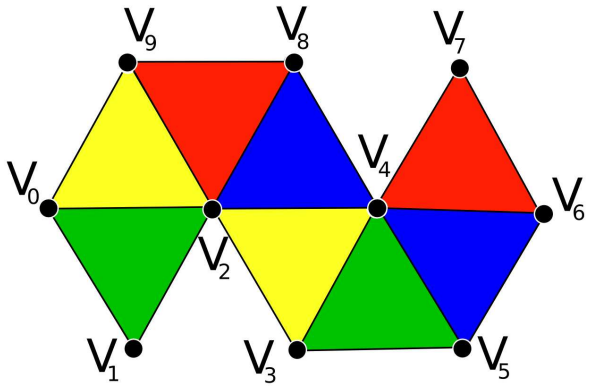

Figura 4.5: Nível 1 da CHE: Adjacência entre os triângulos.

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).

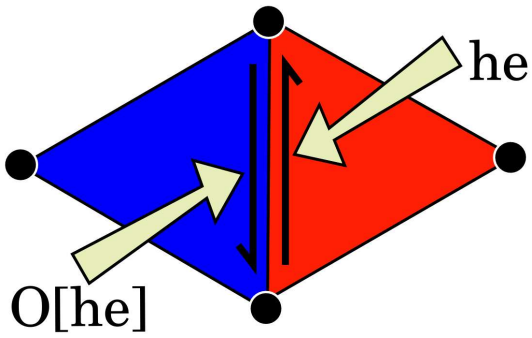

Figura 4.6: Half–edge oposta a *half–edge* HE_{id} he.

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 $\mathsf{O[he]} = -1$, 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).

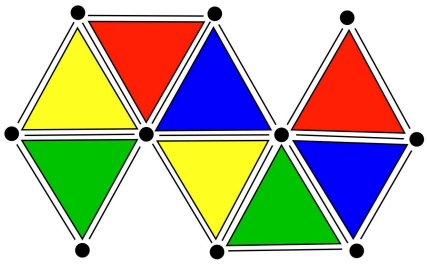

Figura 4.7: Nível 2 da CHE: Representação das células.

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).

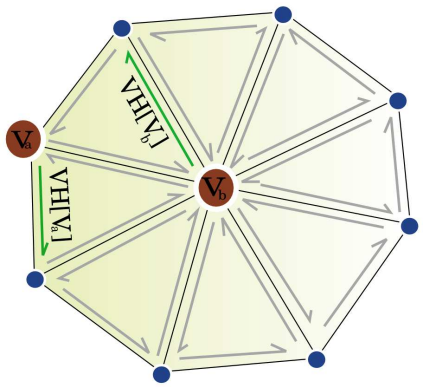

Figura 4.8: Escolha das *half–edges* armazenadas no contêiner VH[].

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).

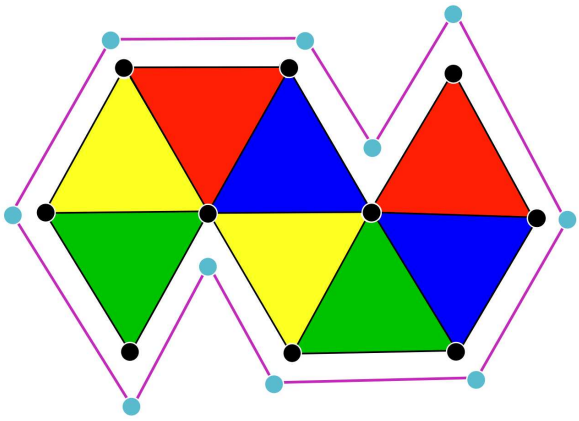

Figura 4.9: CHE nível 3: Representação das curvas de bordo.

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.

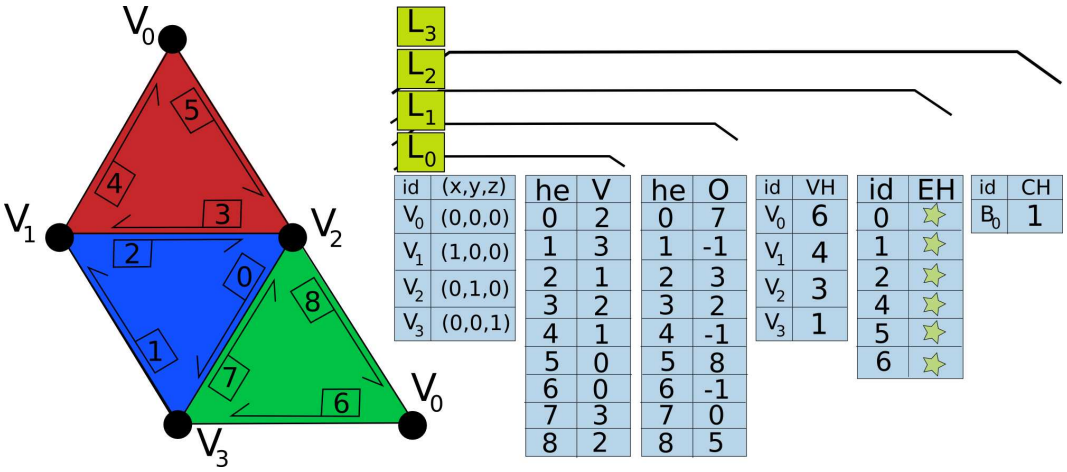

Figura 4.10: Exemplo de construção da CHE .

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 $\mathsf{V}[]$.

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 $\mathsf{VH}[]$, 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 $\mathsf{O}[]$, 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} $\mathsf{v}, \mathsf{VH}[\mathsf{v}]$), 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} $\mathsf{e} = \mathsf{he}$, 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 $\mathsf{V}[]$ 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} $\mathsf{e} = \mathsf{he}$ devemos percorrer o contêiner $\mathsf{V}[]$, 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 $\mathsf{O}[]$ 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 $3\mathsf{t}$, $3\mathsf{t}+1$, $3\mathsf{t}+2$, logo os vértices de t são aqueles de índices $\mathsf{V}[3\mathsf{t}]$, $\mathsf{V}[3\mathsf{t}+1]$, $\mathsf{V}[3\mathsf{t}+2]$ 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 $\mathsf{V}[]$ 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 $\mathsf{O}[]$.

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.

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).

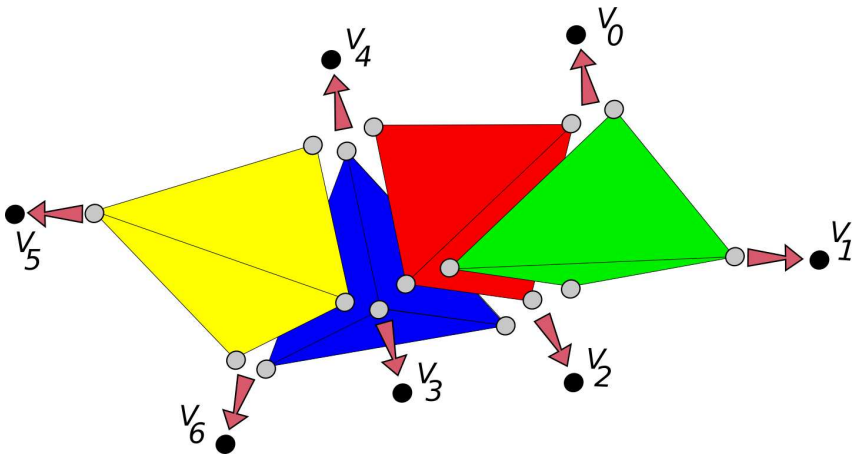

Figura 5.1: Nível 0 da CHF: Sopa de Tetraedros.

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.

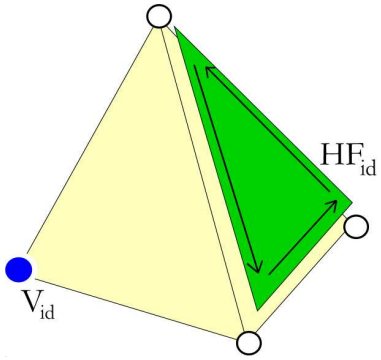

Figura 5.2: Uma *half–face* de um tetraedro.

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 $\mathsf{v} = \mathsf{V}[\mathsf{hf}]$ 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 $\mathbf{n}_0 - 1$.

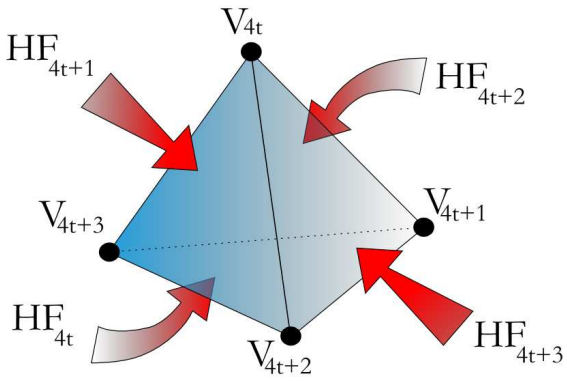

Figura 5.3: Relação entre *half–faces* e vertices.

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 $\mathsf{V}[4\mathsf{t}+1] \rightarrow \mathsf{V}[4\mathsf{t}+2]\,, \mathsf{V}[4\mathsf{t}+2] \rightarrow \mathsf{V}[4\mathsf{t}+3]$, e $\mathsf{V}[4\mathsf{t}+3] \rightarrow \mathsf{V}[4\mathsf{t}+1]$, o ciclo de *half–edges* da *half–face* HF_{id} $4\mathsf{t}+1$ é $\mathsf{V}[4\mathsf{t}+2] \rightarrow \mathsf{V}[4\mathsf{t}]\,, \mathsf{V}[4\mathsf{t}] \rightarrow \mathsf{V}[4\mathsf{t}+3]$, e $\mathsf{V}[4\mathsf{t}+3] \rightarrow \mathsf{V}[4\mathsf{t}+2]$ e assim por diante.

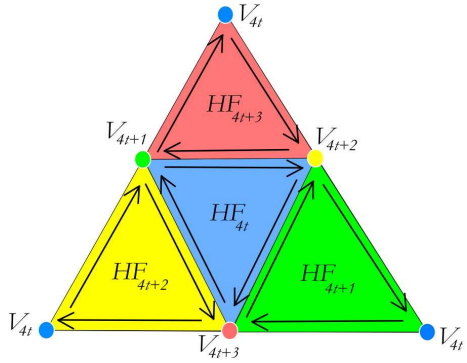

Figura 5.4: Orientação das *half–faces* de um tetraedro T_{id} t.

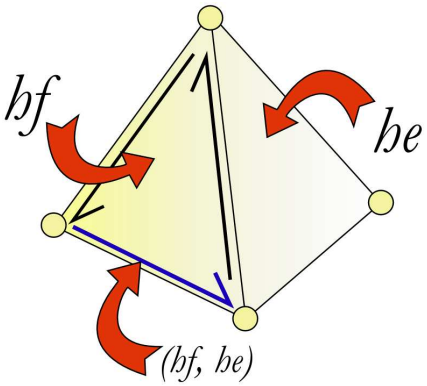

Figura 5.5: Índice de uma *half–edge*.

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.

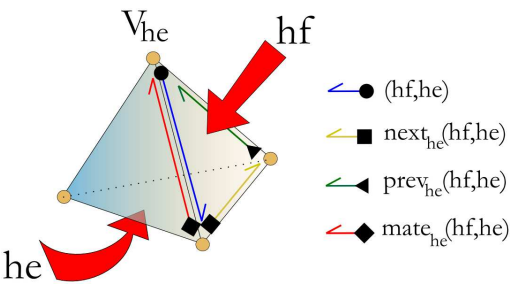

Figura 5.6: *Half–edge mate*

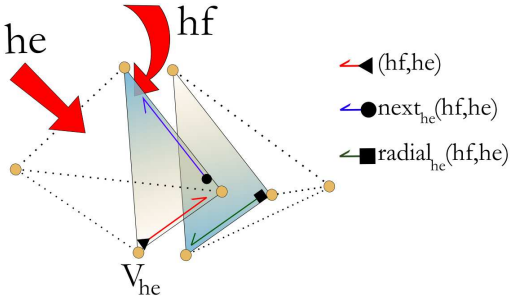

Figura 5.7: *Half–edge radial*

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).

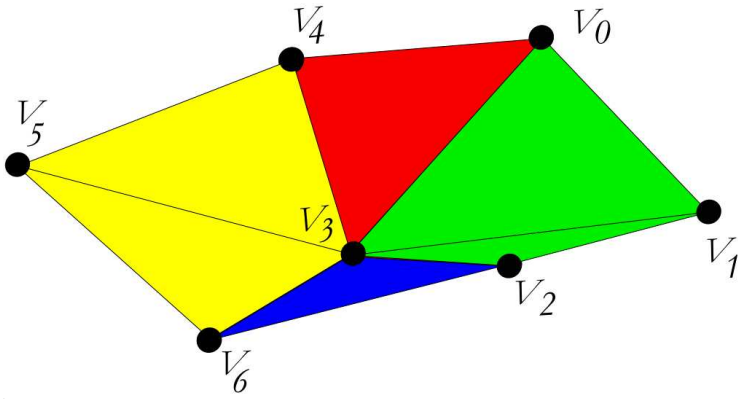

Figura 5.8: Nível 1: Adjacência entre Tetraedros.

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).

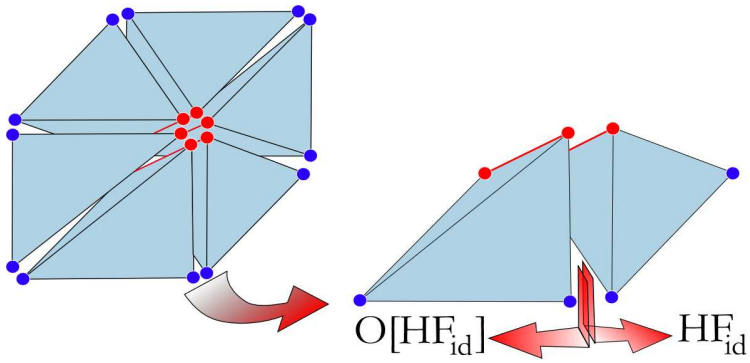

Figura 5.9: Half–face oposta a half-face HF_{id}.

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).

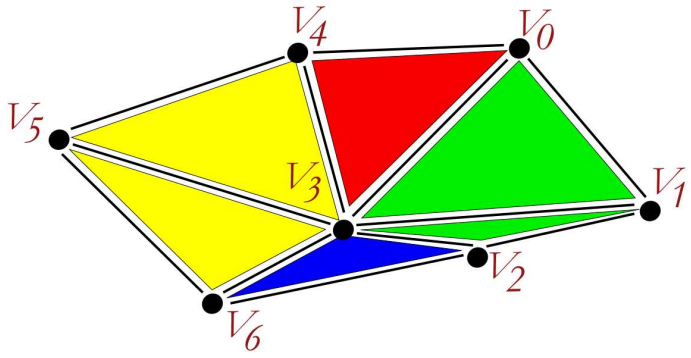

Figura 5.10: Nível 2: Representação das Células.

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.

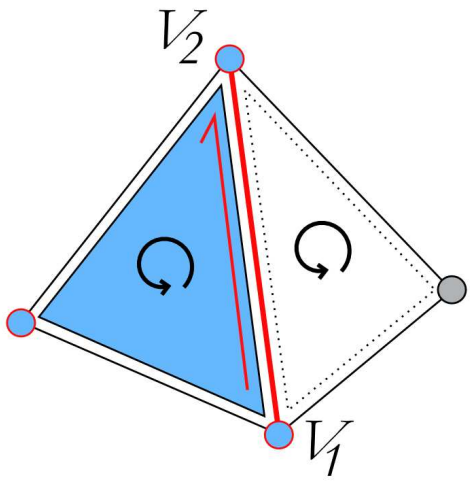

Figura 5.11: Half–face de bordo incidente a uma aresta de bordo.

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.

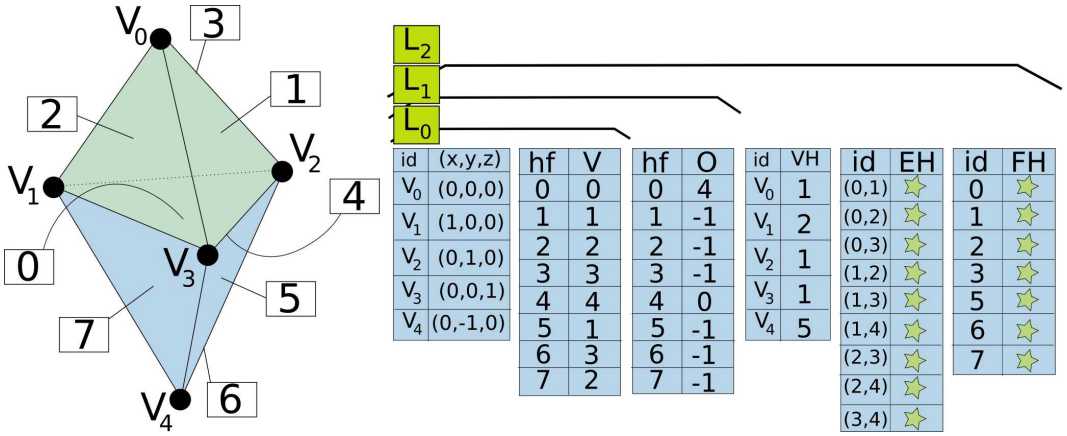

Figura 5.12: Exemplo de construção da CHE .

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 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 4t, 4t + 1, 4t + 2 e 4t + 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 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.

6
Comparações

Neste capítulo analisaremos o consumo de memória em cada nível da CHE e da CHF. Ainda, complementaremos o estudo comparando o consumo de memória de cada nível com o custo de estruturas clássicas, apresentadas anteriormente no capítulo 2. Dividiremos o capítulo em duas seções. Na primeira, analisaremos o consumo de memória nas estruturas clássicas para superfícies e as compararemos com a CHE. Em seguida estudaremos o consumo das estruturas para 3–variedades compararemos com a CHF.

6.1
Estruturas de Dados para Superfícies em $\mathbb{R}^3$

Handle–Edge: Para representar a topologia de uma superfície triangulada, a estrutura de dados *Handle–Edge* requer 4 ponteiros por face, 13 ponteiros por aresta e 4 ponteiros por vértice. Para estimar o espaço de memória consumido pela com a *Handle–Edge* em função do número de vértices do modelo, podemos utilizar a fórmula de Euler para superfícies: $\mathbf{n}_0 - \mathbf{n}_1 + \mathbf{n}_2 = \chi(\mathsf{S})$, onde $\chi(\mathsf{S})$ é a característica de Euler da superfície S. Se assumirmos que o número de genus de uma superfície é pequeno se comparado à $\mathbf{n}_0$, então $\mathbf{n}_0 - \mathbf{n}_1 + \mathbf{n}_2 \approx 0$. Ainda, supondo que o número de arestas de bordo é pequeno e sabendo que cada aresta de interior é compartilhada por duas faces, então $\mathbf{n}_1 \approx \frac{3}{2}\mathbf{n}_2$. Como a valência de um vértice é em média igual a 6, assumindo um espaço de 4 bytes por referência, o consumo de memória da *Handle–Edge* é aproximadamente $4 \cdot 8\mathbf{n}_0 + 4 \cdot 39\mathbf{n}_0 + 4 \cdot 4\mathbf{n}_0 \approx 204\mathbf{n}_0$ bytes.

Corner–Table: A estrutura de dados *Corner–Table* é bastante concisa. Para armazenar as informações topológicas de uma malha de triângulos, supondo que cada número inteiro ocupa 4 bytes, a estrutura consome $4 \cdot 6\mathbf{n}_2 \approx 24\mathbf{n}_2$ bytes de memória já que, para cada triângulo armazenamos o índice do vértice inicial e o *corner* oposto de cada um de seus 3 *corners*.

Ainda, se adicionarmos a estrutura um *array* que para cada vértice do modelo armazena o índice de um de seus *corners* incidentes, como proposto

por (Vieira 2003), precisamos alocar um espaço extra de memória igual a $4 \cdot \mathbf{n}_0$ bytes. Novamente utilizando a fórmula de Euler podemos expressar o consumo de memória em função no número de vértices de um modelo. Estaremos supondo, novamente, que o número de genus e o número de arestas de bordo são pequenos, e assim teremos $\mathbf{n}_1 \approx \frac{3}{2}\mathbf{n}_2$, logo $\mathbf{n}_2 \approx 2\mathbf{n}_0$. Com isso a quantidade de memória consumida pela *Corner–Table* é aproximadamente $24\mathbf{n}_2 + 4\mathbf{n}_0 \approx 24 \cdot 2\mathbf{n}_0 + 4\mathbf{n}_0 \approx 52\mathbf{n}_0$.

Directed–Edges: Como vimos anteriormente, a *Directed–Edges* é uma estrutura de dados escalonável composta de 3 níveis: *small–sized*, *medium–sized*, e *full–sized*.

No primeiro nível, o *small–sized*, a estrutura armazena a menor quantidade de informações topológicas necessária para que possamos implementar as funções de resposta às informações topológicas de forma eficiente. Cada *directed–edge* armazena uma referência para o seu vértice inicial e para sua *directed–edge* vizinha e assim, supondo que cada referência ocupa 4 bytes, precisamos $4 \cdot 2 \cdot 3\mathbf{n}_2 \approx 4 \cdot 2 \cdot 6\mathbf{n}_0 \approx 48\mathbf{n}_0$ bytes para carregar o nível *small–sized* da estrutura.

No nível *medium–sized*, adicionamos uma referência para a directed–edge *previous* de cada *directed–edge* da malha, sendo assim precisamos de $4 \cdot 3 \cdot 3\mathbf{n}_2 \approx 4 \cdot 3 \cdot 6\mathbf{n}_0 \approx 72\mathbf{n}_0$ bytes de memória para utilizar o nível *medium–sized*.

Por fim, no nível *full–sized* adicionamos uma referência para o vértice final e uma referência para a *directed–edge next* de cada *directed–edge* da malha. Logo armazenamos no total 5 referências explicitas de outras entidades do modelo para cada *directed–edge*, e como cada triângulo é composto de 3 *directed–edges*, precisamos de $4 \cdot 5 \cdot 3\mathbf{n}_2 \approx 4 \cdot 5 \cdot 6\mathbf{n}_0 \approx 120\mathbf{n}_0$ bytes para carregar a topologia de uma malha no nível *full–sized*.

CHE: Como apresentamos no capítulo 4, CHE é também uma estrutura de dados escalonável para superfícies, composta pelos níveis 0, 1, 2 e 3.

No nível 0, a CHE não armazena nenhuma informação sobre a adjacência dos triângulos da malha, portanto o espaço ocupado pela estrutura em memória é extremamente pequeno. Neste nível guardamos apenas o índice do vértice inicial de cada *half–edge* no contêiner V[]. Logo, para alocar o nível 0, supondo que cada valor inteiro ocupa 4 bytes, utilizamos apenas $4 \cdot 3\mathbf{n}_2$ bytes de memória e, usando a aproximação deduzida a partir da fórmula de Euler, o nível 0 da CHE consome aproximadamente $4 \cdot 6\mathbf{n}_0 \approx 24\mathbf{n}_0$ bytes de memória.

No nível 1, adicionamos a CHE informações sobre a vizinhança de cada triângulo através do contêiner O[], ou seja, precisamos armazenar a *half–edge* oposta de cada *half–edge* da malha. Com isso, a quantidade de memória que precisamos para o nível 1 da CHE é igual a $4 \cdot 3\mathbf{n}_2 + 4 \cdot 3\mathbf{n}_2 = 24\mathbf{n}_2$, e colocando o custo em função do número de vértices do modelo, precisamos de aproximadamente $48\mathbf{n}_0$ bytes de memória. Com isso dobramos a quantidade de memória utilizada pela estrutura do nível 0 para o nível 1, mas, em contrapartida, aumentamos bastante a eficiência do cálculo das funções de resposta, como vimos na seção 4.6.

O nível 2 da CHE adiciona uma representação explícita para cada célula do modelo. Desta maneira podemos associar a cada uma delas atributos significativos para uma dada aplicação onde deseja-se utilizar a estrutura. Propomos neste nível a associação de cada vértice a uma de suas *half–edges* incidentes através do contêiner Vertex Half–Edge, já que assim otimizamos bastante o cálculo da estrela do vértice, e ainda a criação da Edge Map para a representação explícita de arestas. Com isso, para construir o nível 2 da CHE precisamos alocar ao menos $4\mathbf{n}_0$ bytes de memória para o contêiner Vertex Half–Edge e $20\mathbf{n}_1 \approx 60\mathbf{n}_0$ bytes para a Edge Map. Assim seu custo total do nível 2 da CHE é, aproximadamente, $64\mathbf{n}_0 + 48\mathbf{n}_0 \approx 112\mathbf{n}_0$ bytes.

No último nível da CHE, adicionamos uma representação explícita para cada componente de bordo da superfície. Tal representação é feita armazenando uma *half–edge* incidente a cada curva de bordo da superfície. Assim precisamos de $\mathbf{n}_\partial$ de memória para armazenar a *half–edge* representante de cada curva de bordo. Logo, precisamos para o nível 3, um total de memória de aproximadamente $\mathbf{n}_\partial + 112\mathbf{n}_0$. Como na maioria das vezes a quantidade de componentes de bordo é pequena, o espaço de memória necessário é de aproximadamente $112\mathbf{n}_0$ bytes.

Comparações: Cada estrutura de dados apresentada trata de forma diferente o balanceamento entre o uso de memória e o custo computacional. Dentre as estruturas estudadas, a CHE é a mais flexível já que propõe uma implementação escalonável que se adapta a uma larga escala de aplicações.

Mesmo quando comparada às estruturas mais compactas, como a *Corner–Table*, a CHE se mostra bastante eficiente já que no nível 2, por exemplo o custo da CHE e da *Corner–Table* são os mesmos.

Em relação a estruturas mais sofisticadas como a *Handle–Edge* e a *Directed–Edges* a CHE também apresenta vantagens. A CHE consegue tratar o conceito de escalabilidade de maneira mais ampla e flexível que a *Directed–Edges*, uma vez que a mudança de nível proporciona muito mais que um

equilíbrio no uso de memória e processador, ela também nos dá acesso a novas informações sobre a malha, não conseguidas diretamente a partir dos níveis anteriores. Ainda, comparada a *Handle–Edge* a CHE representa um grande ganho no uso de memória já que o nível 3 da CHE e a *Handle–Edge* apresentam a mesma característica, que é a representação explícita das curvas de bordo, mas o nível 2 da CHE é bem mais compacto que a *Handle–Edge*.

Uma desvantagem da CHE em relação à *Handle–Edge* é que, a inserção e remoção de células é menos eficiente, já que, tendo em vista que a CHE trabalha com contêineres de índices inteiros, a mudança no número de células requer a reorganização dos índices. Em contrapartida, a *Handle–Edge*, por trabalhar com ponteiros, necessita apenas do redirecionamento de referências, o que sem dúvida requer menos esforço do que a reorganização de índices da CHE.

Na tabela 6.2 fazemos um quadro comparativo dos gastos de memória de cada estrutura.

	Custo em Bytes
Handle–Edge	$204\mathbf{n}_0$
Corner–Table	$52\mathbf{n}_0$
Directed–Edges *Small*	$48\mathbf{n}_0$
Directed–Edges *Medium*	$72\mathbf{n}_0$
Directed–Edges *Full*	$120\mathbf{n}_0$
CHE *Nível* 0	$24\mathbf{n}_0$
CHE *Nível* 1	$48\mathbf{n}_0$
CHE *Nível* 2	$112\mathbf{n}_0$
CHE *Nível* 3	$112\mathbf{n}_0$

Tabela 6.1: Custo de memória para representação da topologia.

6.2 Estruturas de Dados para 3–Variedades

Nesta seção, analisaremos o consumo de memória das estrutura de dados para variedades de dimensão 3 apresentadas neste trabalho e os compararemos com o custo da CHF. Assim como fizemos na seção anterior, expressaremos tais custos em função do número de vértices da malha, e para isso mais utilizaremos a equação de Euler para variedades de dimensão 3: $\mathbf{n}_0 - \mathbf{n}_1 + \mathbf{n}_2 - \mathbf{n}_3 = \chi(\mathsf{M})$, onde $\chi(\mathsf{M})$ é a característica de Euler da 3–variedade M.

Supondo que a característica de Euler de uma 3–variedade é pequena se comparada ao número de vértices, e ainda imaginando que a quantidade de faces de bordo é pequena se comparada ao número de faces de interior (Gumhold *et al.* 1999) mostra que podemos trabalhar com as seguintes aproximações: $\mathbf{n}_0 : \mathbf{n}_1 : \mathbf{n}_2 : \mathbf{n}_3 \approx 1 : 6.5 : 11 : 5.5$.

Por padrão, adotaremos que o espaço necessário para armazenar um valor inteiro em memória é de 4 bytes.

Handle–Face: Para representar a topologia de uma 3–variedade, a estrutura de dados *Handle–Face* necessita de 136 ponteiros por tetraedro, 5 ponteiros por face, 4 ponteiros por aresta e 3 ponteiros por vértice. Para estimar o espaço de memória utilizado por uma modelo carregado com a *Handle–Face*, utilizaremos as aproximações deduzidas a partir da formula de Euler e apresentadas em (Gumhold *et al.* 1999) e descritas anteriormente. Desta forma, como cada ponteiro requer 4 bytes, a *Handle–Face* precisará de $4 \cdot (136\mathbf{n}_3 + 5\mathbf{n}_2 + 4\mathbf{n}_1 + 3\mathbf{n}_0) \approx 4 \cdot (748\mathbf{n}_0 + 55\mathbf{n}_0 + 26\mathbf{n}_0 + 3\mathbf{n}_0) \approx 3328\mathbf{n}_0$ bytes de memória.

Indexed Data–Structure with Adjacency: A estrutura de dados *Indexed data–structure with adjacency*, a *Ia*, é bastante concisa. Como a estrutura armazena apenas o k–esqueleto e a relação de adjacência entre os k–simplexos da malha, precisamos apenas de $4 \cdot 8\mathbf{n}_3 \approx 32\mathbf{n}_3$ bytes para armazenar as informações todas as relações topológicas da estrutura.

Novamente utilizando a fórmula de Euler podemos expressar o consumo de memória em função no número de vértices do modelo. Estaremos supondo, novamente, que o número de genus e o número de faces de bordo são pequenos, e, assim, teremos $\mathbf{n}_3 \approx 5.5\mathbf{n}_0$, e a quantidade de memória consumida pela *Ia*, será, aproximadamente, $176\mathbf{n}_0$ bytes.

NMIA: A *Non–manifold indexed data structure with adjacencies*, ou *NMIA* por simplicidade, foi desenvolvida para representar não–variedades de dimensão 3, mas proporcionando uma boa adaptatividade na representação de 3–variedades. A *NMIA* estende a *Ia* codificando as múltiplas componentes conexas existentes na estrela de vértices ou arestas não–variedade.

O custo em memória para construirmos a *NMIA* é de $8\mathbf{n}_3 + 3\mathbf{n}_2^t + 2\mathbf{n}_1^t + C_e + C_v$, onde C_e representa o número total de clusters em arestas não–variedades, C_v representa o número total de componentes conexas nas estrelas dos vértices, $\mathbf{n}_2^t$ representa o número de *dangling–faces*, e $\mathbf{n}_1^t$ o número de *wire–edges* do modelo.

Para compará-la com a CHF, analisaremos o consumo de memória da *NMIA* apenas para malhas de tetraedros que são 3–variedades. Como complexos que são variedades não possuem *dangling–faces*, *wire edges*, e as estrelas de vértices e de arestas tem apenas uma componente conexa, $\mathbf{n}_2^t = \mathbf{n}_1^t = C_e = 0$ e $C_v = \mathbf{n}_0$. Lodo o custo em bytes de armazenamento da estrutura será igual ao da Ia, ou seja aproximadamente $4 \cdot 44\mathbf{n}_0 \approx 176\mathbf{n}_0$.

CHF: Como apresentamos no capítulo 5, a CHF é uma estrutura de dados escalonável para 3–variedades, composta por 4 níveis.

No nível 0, a CHF não armazena informações sobre a adjacência dos tetraedros da malha, logo o espaço ocupado pela estrutura em memória

é extremamente pequeno. Como armazenamos apenas o vértice oposto a cada *half–face*, no tetraedro incidente, utilizamos apenas de $4 \cdot 4\mathbf{n}_3$ bytes de memória neste nível. Ainda, como $\mathbf{n}_3 \approx 5.5\mathbf{n}_0$, o nível 0 da CHF consome aproximadamente $88\mathbf{n}_0$ bytes de memória.

No nível 1, adicionamos a CHF informações sobre a vizinhança de cada tetraedro, ou seja, armazenamos a *half–face* oposta de cada *half–face* da malha. Assim, a quantidade de memória que precisamos para o nível 1 da CHF é igual a $4 \cdot 4\mathbf{n}_3 + 4 \cdot 4\mathbf{n}_3 = 32\mathbf{n}_3$ bytes, e colocando em função do número de vértices do modelo, precisamos de aproximadamente $176\mathbf{n}_0$ bytes de memória. Com isso dobramos a quantidade de memória utilizada pela estrutura do nível 0 para o nível 1, mas em contrapartida aumentamos bastante a eficiência do cálculo das funções de resposta, como vimos na seção 5.6.

O nível 2 da CHF adiciona uma representação explícita para cada célula do modelo. Desta maneira podemos associar a cada uma delas atributos significativos para uma dada aplicação onde deseja-se utilizar a estrutura. Propomos neste nível a associação de cada vértice e aresta a uma de suas *half–faces* incidentes através dos contêineres Vertex Half–Face e Edge Map, já que assim otimizamos bastante o cálculo das estrelas de vértices e arestas. Propomos também, neste nível, a criação do contêiner Face Map, para a representação explícita de faces. Como vimos na seção 2.8, precisamos de aproximadamente 20 bytes por aresta e face para a construção de contêineres do tipo map, logo o nível 2 da CHF precisa alocar $4\mathbf{n}_0 + 20\mathbf{n}_1 + 20\mathbf{n}_2$ bytes a mais que o nível 1, e seu custo total em memória é de aproximadamente $4\mathbf{n}_0 + 20\mathbf{n}_1 + 20\mathbf{n}_2 + 176\mathbf{n}_0 \approx 4\mathbf{n}_0 + 130\mathbf{n}_0 + 220\mathbf{n}_0 + 176\mathbf{n}_0 \approx 530\mathbf{n}_0$ bytes.

No último nível da CHF, adicionamos uma representação explícita para as superfícies de bordo da 3–variedade. Tal representação é feita implementando o nível 1 da CHE. Assim precisamos de $48\partial\mathbf{n}_0$ bytes a mais que no nível 2 da CHF, onde $\partial\mathbf{n}_0$ é o número de vértices da superfície de bordo. Logo, precisamos para o nível 3, um total de memória de aproximadamente $530\mathbf{n}_0 + 48\partial\mathbf{n}_0$.

Comparações: Cada estrutura de dados gerencia de forma diferente o uso de memória e o custo computacional. A CHF mostrou-se mais flexível que as demais, já que é a única que propõe uma implementação escalonável que se adapta a uma larga escala de aplicações.

Mesmo quando comparada às estruturas mais compactas, como a Ia ou a NMIA restrita a variedades, a CHF é bastante competitiva, já que no nível 1, por exemplo, o custo da CHF é o mesmo destas estruturas.

A CHF consegue tratar o conceito de escalabilidade de maneira bastante ampla e flexível pois, assim como na CHE, a mudança de nível proporciona

	Custo em Bytes
Handle–Face	$3328\mathbf{n}_0$
IA/ NMIA *Manifold*	$176\mathbf{n}_0$
CHF *Nível* 0	$88\mathbf{n}_0$
CHF *Nível* 1	$176\mathbf{n}_0$
CHF *Nível* 2	$530\mathbf{n}_0$
CHF *Nível* 3	$530\mathbf{n}_0 + 48\partial\mathbf{n}_0$

Tabela 6.2: Custo de memória para representação da topologia.

muito mais que o equilíbrio entre o uso de memória e processador, ela também nos dá acesso a novas informações sobre a malha, não conseguidas diretamente nos níveis anteriores.

Em relação a estruturas mais sofisticadas como a *Handle–Face* a CHF representa um grande ganho no uso de memória já que o nível 3 da CHF e a *Handle–Face* apresentam a mesma característica principal, que é a representação explícita das superfícies de bordo, mas o nível 3 da CHF é bem mais compacto que a *Handle–Face*.

Uma desvantagem da CHF em relação à *Handle–Face* é que, a inserção e remoção de células é menos eficiente, já que, tendo em vista que a CHF trabalha com contêineres de índices inteiros, a mudança no número de células requer a reorganização dos índices de tempos em tempos via um procedimento de "recolhimento de lixo" (*Garbage Collection*). Em contrapartida, a *Handle–Face*, por trabalhar com ponteiros, necessita apenas do redirecionamento de referências, o que sem dúvida requer menos esforço.

Na tabela 6.2 fazemos um quadro comparativo dos gastos de memória de cada estrutura.

7
Conclusões e Trabalhos Futuros

Neste capítulo apresentaremos alguns resultados obtidos com a implementação das CHE e da CHF. Em seguida faremos algumas conclusões e indicaremos os trabalhos futuros.

Resultados: A estrutura CHE foi implementada utilizando a linguagem de programação C+. Com o suporte da linguagem à herança virtual de classes, a implementação e o uso da CHE se tornam simples já que a mesma interface pode ser usada para uma função em todos os níveis da estrutura. Assim, cada nível da estrutura de dados é implementado em uma nova classe, que herda todas as características do nível anterior, adiciona novas funcionalidades à estrutura e otimiza a busca de informações topológicas.

A seguir, mostraremos alguns exemplos de superfícies representadas por diferentes níveis da CHE. Calcularemos o custo de armazenamento de cada uma delas no diferentes níveis da estrutura.

A figura 7.1 mostra uma malha de triângulos, do modelo do Stanford Bunny. Nesta versão, o modelo é composto por 100.000 vértices e 199.322 triângulos. Na tabela 7.1 mostramos o custo aproximado de armazenamento das informações topológicas do modelo do Stanford Bunny em cada nível da CHE.

	Custo em Bytes
CHE *Nível* 0	2.400.000
CHE *Nível* 1	4.800.000
CHE *Nível* 2	11.200.000
CHE *Nível* 3	11.200.000

Tabela 7.1: Custo de memória para representação do modelo Stanford Bunny.

A figura 7.2 mostra uma malha de triângulos, do modelo do Happy Buddha, também obtido no repositório de modelos 3D da universidade de Stanford, carregada com o nível 2 da CHE.

Como, neste nível, existe uma representação explícita das arestas da malha, a visualização em *wireframe* do modelo é bastante eficiente, pois cada aresta é desenhada apenas uma vez. Em estruturas baseadas apenas na

representação de triângulos e suas adjacências (como a *Corner–Table* ou a *Directed–Edges*) desenhamos cada aresta duas vezes, uma vez por triângulo incidente.

O modelo é composto por 543.652 vértices e 1.087.716 triângulos. Na tabela 7.2 mostramos o custo aproximado de armazenamento das informações topológicas do modelo do Happy Buddha em cada nível da CHE.

	Custo em Bytes
CHE *Nível* 0	13.047.648
CHE *Nível* 1	26.095.296
CHE *Nível* 2	60.889.024
CHE *Nível* 3	60.889.024

Tabela 7.2: Custo de memória para representação do modelo Happy Buddha.

A figura 7.3 mostra uma malha de triângulos, gerada a partir de um modelador CSG, e carregada com o nível 1 da CHE. O modelo é composto por 82.020 vértices e 164.036 triângulos. Na tabela 7.3 mostramos o custo aproximado de armazenamento das informações topológicas do modelo CSG em cada nível da CHE.

	Custo em Bytes
CHE *Nível* 0	1.968.480
CHE *Nível* 1	3.936.960
CHE *Nível* 2	9.186.240
CHE *Nível* 3	9.186.240

Tabela 7.3: Custo de memória para representação do modelo CSG.

A figura 7.4 mostra o modelo 3D da cabeça da estátua de David, obtido à partir de um scanner 3D, pelo projeto Michelangelo da universidade de Stanford. Neste exemplo estamos visualizando o modelo com o nível 3 da CHE.

Neste nível, existe uma representação explícita das curvas de bordo do modelo, que utilizamos neste exemplo para visualizar o bordo do pescoço da estátua de David.

Nesta versão, o modelo é composto por 50.329 vértices e 100.458 triângulos. Na tabela 7.4 mostramos o custo aproximado de armazenamento das informações topológicas do modelo David em cada nível da CHE.

	Custo em Bytes
CHE *Nível* 0	1.207.896
CHE *Nível* 1	2.415.792
CHE *Nível* 2	5.636.848
CHE *Nível* 3	5.636.852

Tabela 7.4: Custo de memória para representação do modelo David.

A figura 7.5 mostra uma malha de triângulos, do modelo do Dragon da universidade de Stanford. O modelo é composto por 437.645 vértices e 871.414 triângulos. Na tabela 7.2 mostramos o custo aproximado de armazenamento das informações topológicas do modelo do Dragon em cada nível da CHE.

	Custo em Bytes
CHE *Nível* 0	10.503.480
CHE *Nível* 1	21.006.960
CHE *Nível* 2	49.016.204
CHE *Nível* 3	49.016.204

Tabela 7.5: Custo de memória para representação do modelo Dragon.

Assim como na implementação da CHE, desenvolvemos a CHF em C++, e como havíamos dito, cada nível da estrutura foi criado utilizando o conceito de heranças virtuais da linguagem, com os níveis representados por classes hierarquicamente relacionadas.

A figura 7.6 mostra um modelo volumétrico do Stanford Bunny, com 48.810 vértices e 270.660 tetraedros, representado pelo nível 1 da CHF.

O custo aproximado de memória para armazenar as informações topológicas do bunny em cada um dos níveis da CHF está expresso na tabela 7.6.

	Custo de memória
CHF *Nível* 0	4.295.280
CHF *Nível* 1	8.590.560
CHF *Nível* 2	25.826.300
CHF *Nível* 3	26.202.332

Tabela 7.6: Custo de memória para representação do modelo Stanford Bunny volumétrico.

As figuras 7.7 e 7.8, ilustram o bunny representado pelo nível 2 da CHF. Com a estratégia de armazenar uma *half–face* incidente para cada vértice e aresta do modelo, e escolhendo tal *half–face* de forma que esta seja uma *half–face* de bordo no caso de arestas e vértices de bordo, temos uma classificação direta das células da malha como células de interior e células de bordo.

A figura 7.9 mostra a representação do bordo do modelo volumétrico do Stanford bunny, através da CHE no nível 3 da CHF.

A figura 7.10, mostra o modelo Hand com 28.793 vértices e 125.127 tetraedros, e um campo escalar definido nos vértices. O custo aproximado de armazenamento das informações topológicas do modelo Hand em cada um dos níveis da CHF estão descritos na tabela 7.7. A figura 7.11, mostra o modelo Blunto com 40.921 vértices e 187.318 tetraedros. O custo aproximado

	Custo de memória
CHF *Nível* 0	2.533.784
CHF *Nível* 1	5.067.568
CHF *Nível* 2	15.260.290
CHF *Nível* 3	15.989.506

Tabela 7.7: Custo de memória para representação do modelo Hand.

de armazenamento das informações topológicas do modelo Hand em cada um dos níveis da CHF estão descritos na tabela 7.8.

	Custo em Bytes
CHF *Nível* 0	3.601.048
CHF *Nível* 1	7.202.096
CHF *Nível* 2	21.688.130
CHF *Nível* 3	22.010.738

Tabela 7.8: Custo de memória para representação do modelo Blunto.

A figura 7.12, mostra o modelo Gargoyle com 48.553 vértices e 258.229 tetraedros. O custo aproximado de armazenamento das informações topológicas do modelo Gargoyle em cada um dos níveis da CHF estão descritos na tabela 7.9.

	Custo em Bytes
CHF *Nível* 0	4.272.664
CHF *Nível* 1	8.545.328
CHF *Nível* 2	25.733.090
CHF *Nível* 3	26.306.978

Tabela 7.9: Custo de memória para representação do modelo Gargoyle.

Por fim, nos gráficos 7.13, 7.14 temos um resumo do tempo de carregamento de cada modelo apresentando nesta seção, utilizando as estruturas de dados convenientes. Como não podia deixar de acontecer, quando carregamos os últimos níveis da CHE ou da CHF, precisamos de mais tempo de processamento para gerar as informações topológicas extras que são armazenadas. Entretando, como discutimos durante a dissertação, depois de carregados os níveis mais altos da CHE e CHF podem responder questões topológicas de forma muito mais eficiente que nos primeiros níveis.

Conclusões: A principal contribuição deste trabalho é projetar e descrever a implementação de duas novas estruturas de dados topológicas bastante eficientes, chamadas de CHE e CHF. A CHE foi desenvolvida para representar superfícies com ou sem bordo, através de uma malha de triângulos, já a

CHF pode representar 3–variedades com ou sem bordo através de malhas de tetraedros.

As duas estruturas utilizam o conceito de escalabilidade para gerenciar o uso de memória e o tempo de processamento de maneira conveniente, ou seja, caso exista espaço suficiente em memória, a maior quantidade possível de informações topológicas é armazenada, e como conseqüência operações tais como as funções de resposta às informações topológicas podem ser calculadas de maneira mais eficiente. Na prática o conceito de escalabilidade pode ser obtido através do uso de heranças virtuais da linguagem de programação C++.

Tanto a CHE quanto a CHF são compostas de 4 níveis que trabalham de maneira diferente o balanceamento entre uso de memória e processamento, com isso ambas podem ser aplicadas a uma larga escala de aplicações.

Para título de comparação, descrevemos outras estruturas topológicas clássicas para superfícies e 3–variedades. Tanto a CHE quanto a CHF se mostraram bastante compactas e eficientes, já que em seus primeiros níveis o consumo de memória é equivalente ao consumo das estruturas mais compactas presentes na literatura e com mesma eficiência nos algoritmos para o cálculo das funções de resposta. Ainda, comparadas a estruturas mais complexas, como a *Handle–Edge* ou a *Directed–Edges* para superfícies, e a *Handle–Face* para 3–variedades, as estruturas de dados propostas apresentaram um ganho bastante significativo em termos de memória, sem a perda das características das estruturas originais.

A CHE e a CHF são extremamente simples de serem implementadas, e fáceis de serem usadas. Na prática a escolha do nível a ser utilizado pode ser feita de várias maneiras, entre elas:

- O programador pode escolher um determinado nível que será usado durante todo o programa.
- Um conjunto de regras podem ser definidas para que seja decidido dinamicamente qual nível apresenta um melhor gerenciamento de uso de memória e tempo de processamento.
- De acordo com a demanda de processamento, um novo nível pode ser temporariamente alocado, conseguindo uma melhora de processamento temporária, de acordo com uma certa tolerância pré definida.

Por fim, o uso de contêineres em lugar de arrays estáticos ou ponteiros faz com que a implementação da CHE e da CHF sejam bastante genéricas.

Trabalhos Futuros: Uma extensão natural deste trabalho é explorar ainda mais o conceito de escalabilidade, proposto para o desenvolvimento da CHE e da CHF.

Primeiramente pensamos em adicionar um novo nível às estruturas que possa tratar de maneira eficiente com não–variedades. Esta extensão pode ser feita de forma natural, se adotarmos a estratégia de armazenar, para cada vértice ou aresta não–variedade do modelo, uma *half–face* ou *half–edge* incidente em cada componente conexa de suas estrelas.

Desejamos também desenvolver uma generalização da estrutura para variedades de dimensão n. Seguindo a estratégia atual, poderíamos criar uma estrutura com 4 níveis que tivessem as seguintes características: no nível 0, representaríamos apenas os n–simplexos da n–variedade, no nível 1 representaríamos a relação de adjacência entre os n–simplexos, através de suas $(n-1)$–faces, no nível 2 adicionaríamos uma representação explícita para as m–células da variedade, $m \leq n$ e por fim no nível 3, representaríamos a $(n$–$1)$–variedade de bordo.

Outra generalização possível para as estruturas é o tratamento de malhas com células que não são simplexos.

Por fim, desejamos implementar operadores topológicos eficientes para a contrução, desconstrução e modificação local de dimensão 2 e 3 como os de (Lopes 1996) para a *Handle–Edge* e a *Handle–Face*, baseados em operadores de alça e operadores, que tirem proveito das características da CHF.

Por fim, com os operadores estelares e de alça implementados podemos investigar a implementação de algoritmos de simplificação, como o de (Vieira 2003), e subdivisão nos diversos níveis tanto da CHE quanto da CHF.

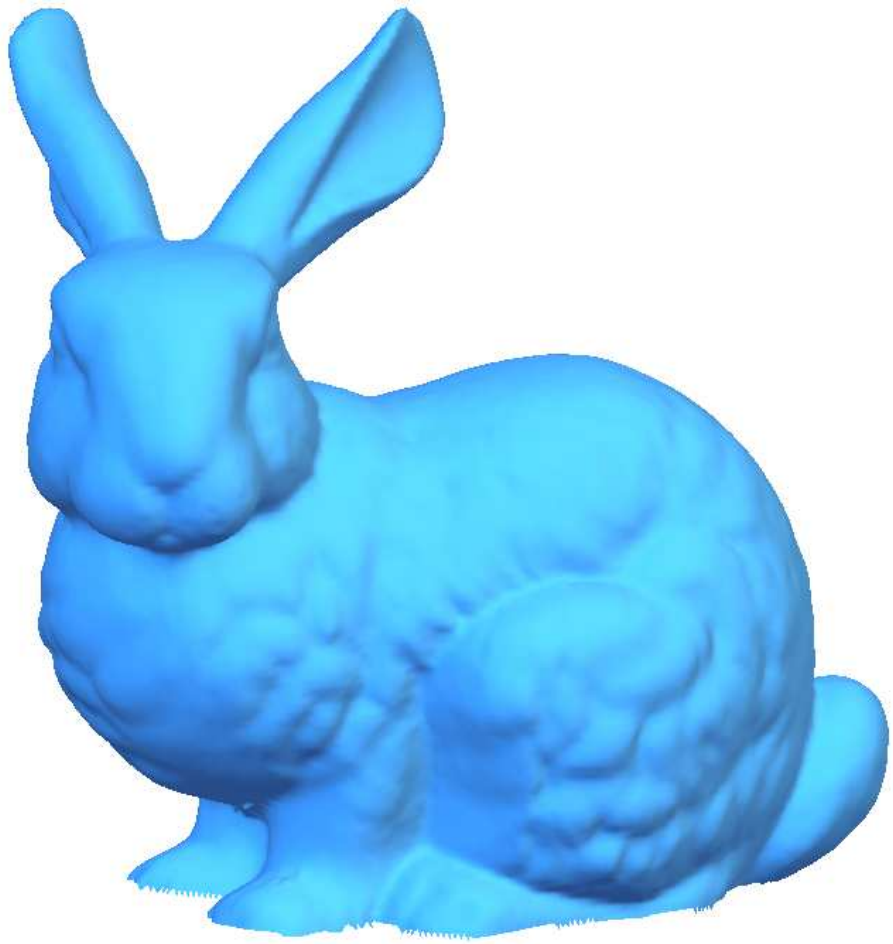

Figura 7.1: Stanford Bunny, 100.000 vértices, 199.322 triângulos, CHE nível 2.

Figura 7.2: Happy Buddha, 543.652 vértices, 1.087.716 triângulos, CHE nível2.

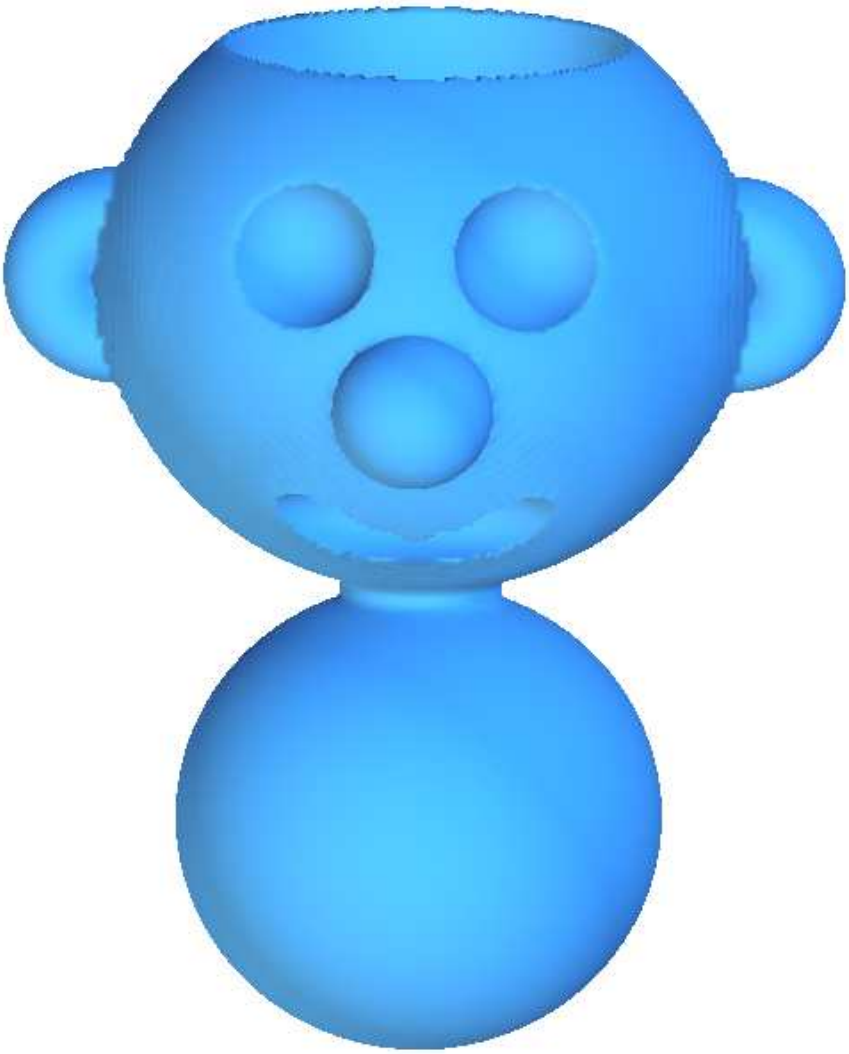

Figura 7.3: Modelo CSG, 82.020 vértices, 164.036 triângulos, CHE nível 1.

Figura 7.4: David, 50.329 vértices, 100.458 triângulos, CHE nível 3.

Figura 7.5: Dragon, 437.645 vértices, 871.414 triângulos, CHE nível 3.

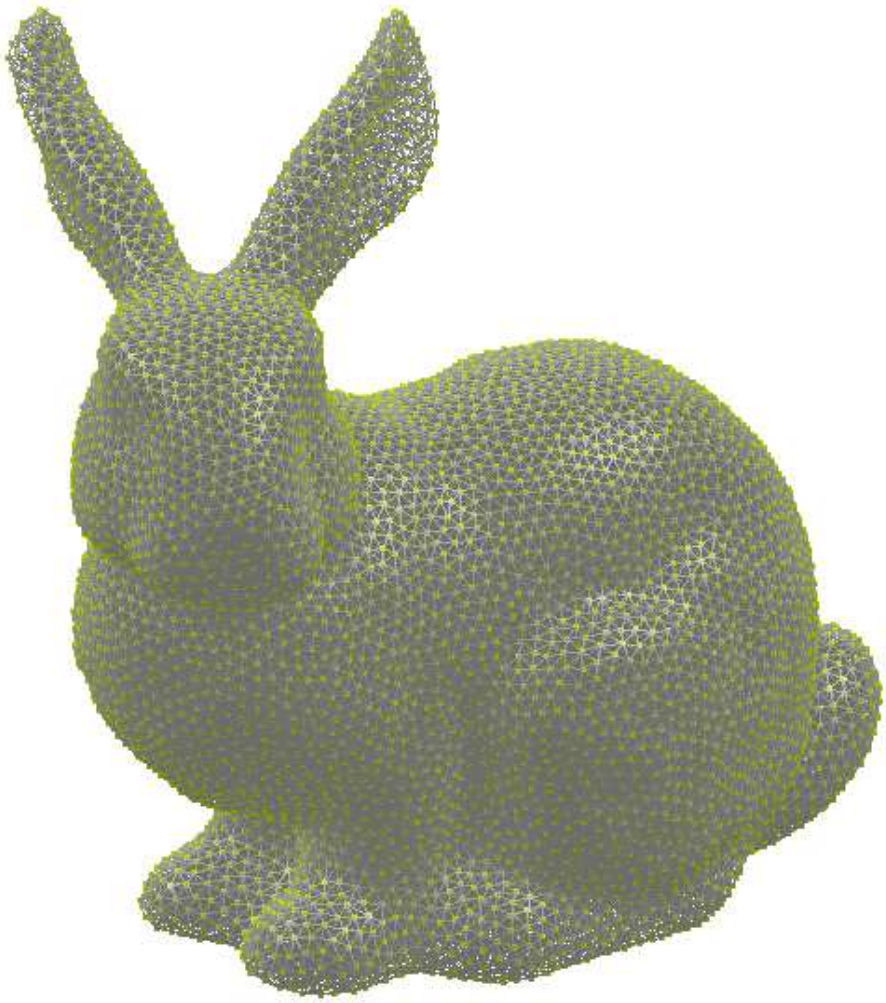

Figura 7.6: Stanford Bunny, 48.810 vértices, 273.660 tetraedros, CHF nível 1.

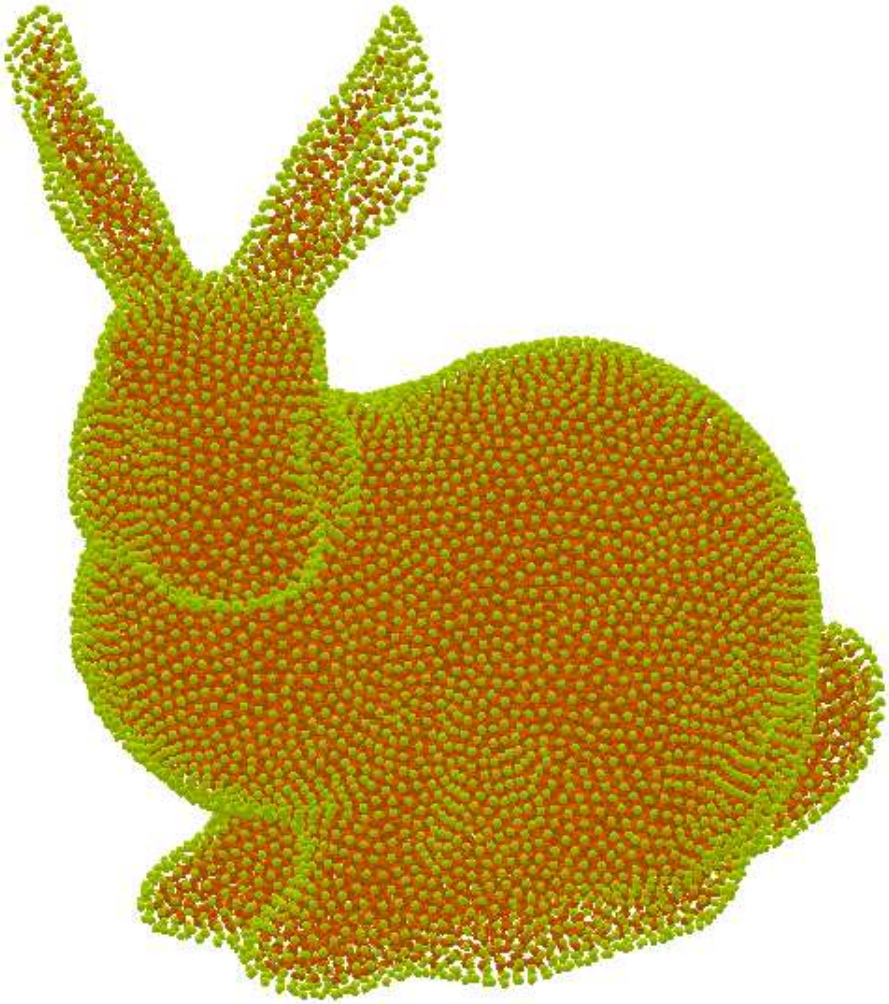

Figura 7.7: Stanford Bunny, classificação dos vértices CHF nível 2.

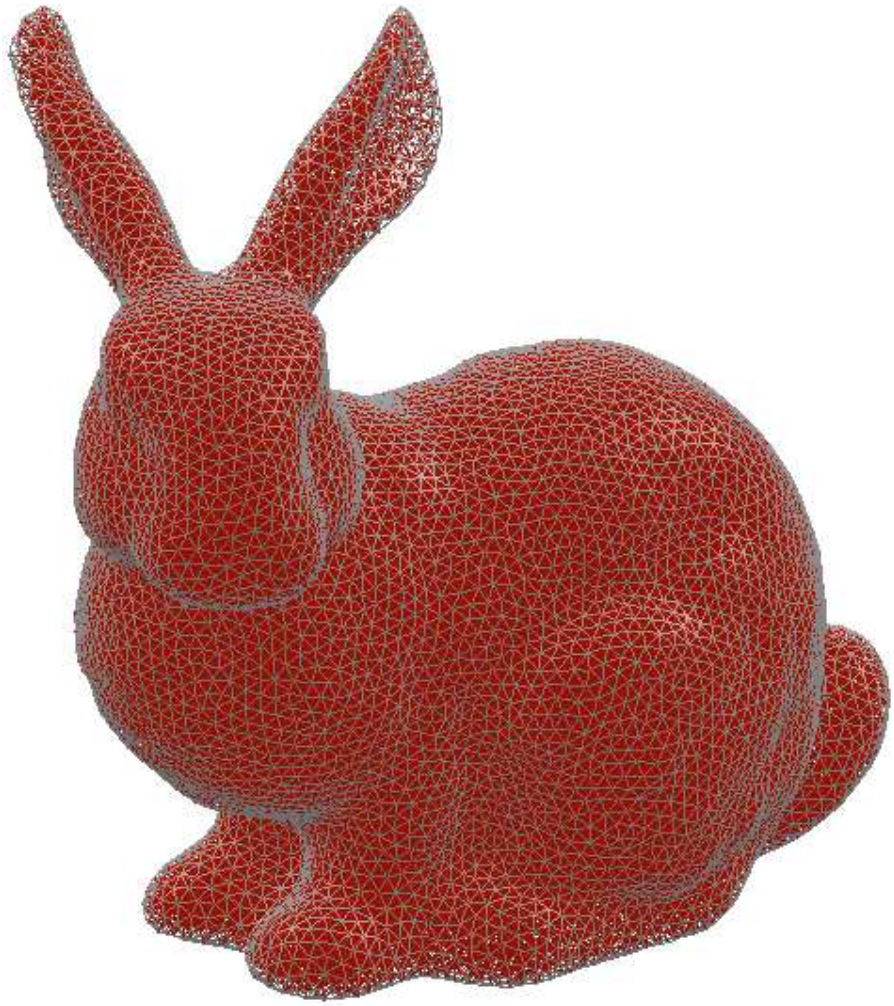

Figura 7.8: Stanford Bunny, classificação das arestas CHF nível 2.

Figura 7.9: Stanford Bunny, superfície de bordo CHF nível 3.

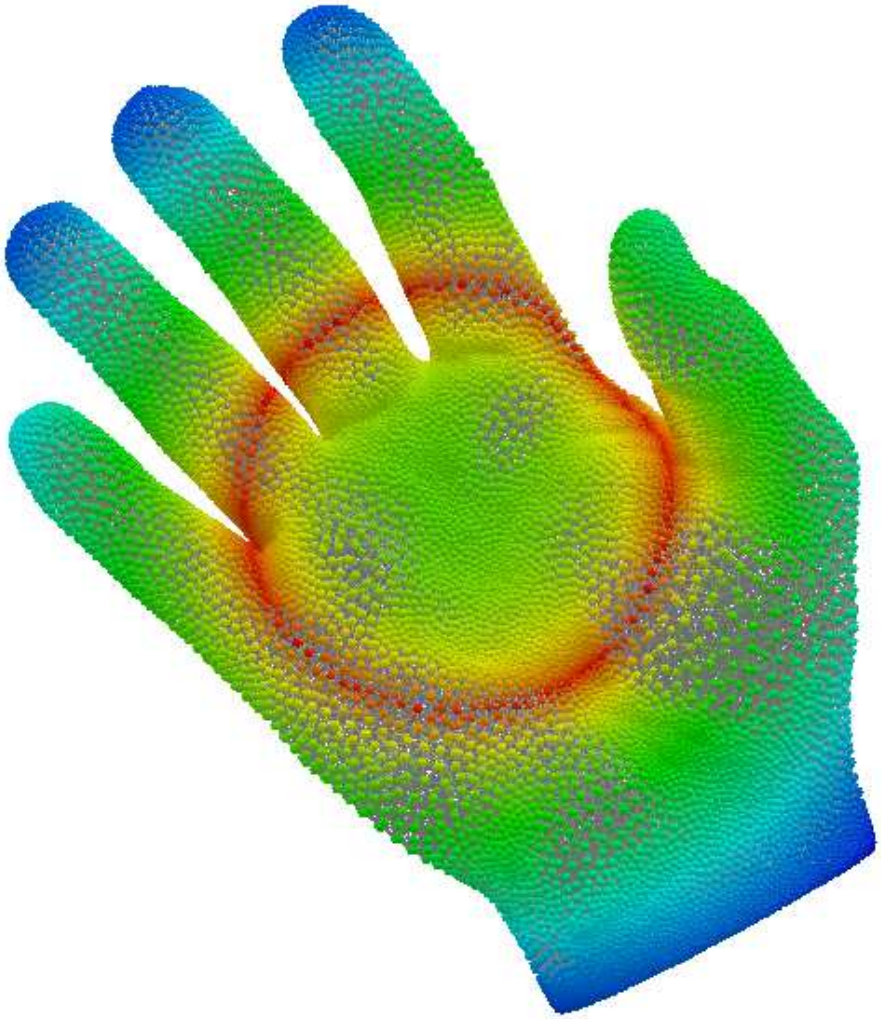

Figura 7.10: Hand, 28.793 vértices, 125.127 tetraedros, Campo Escalar.

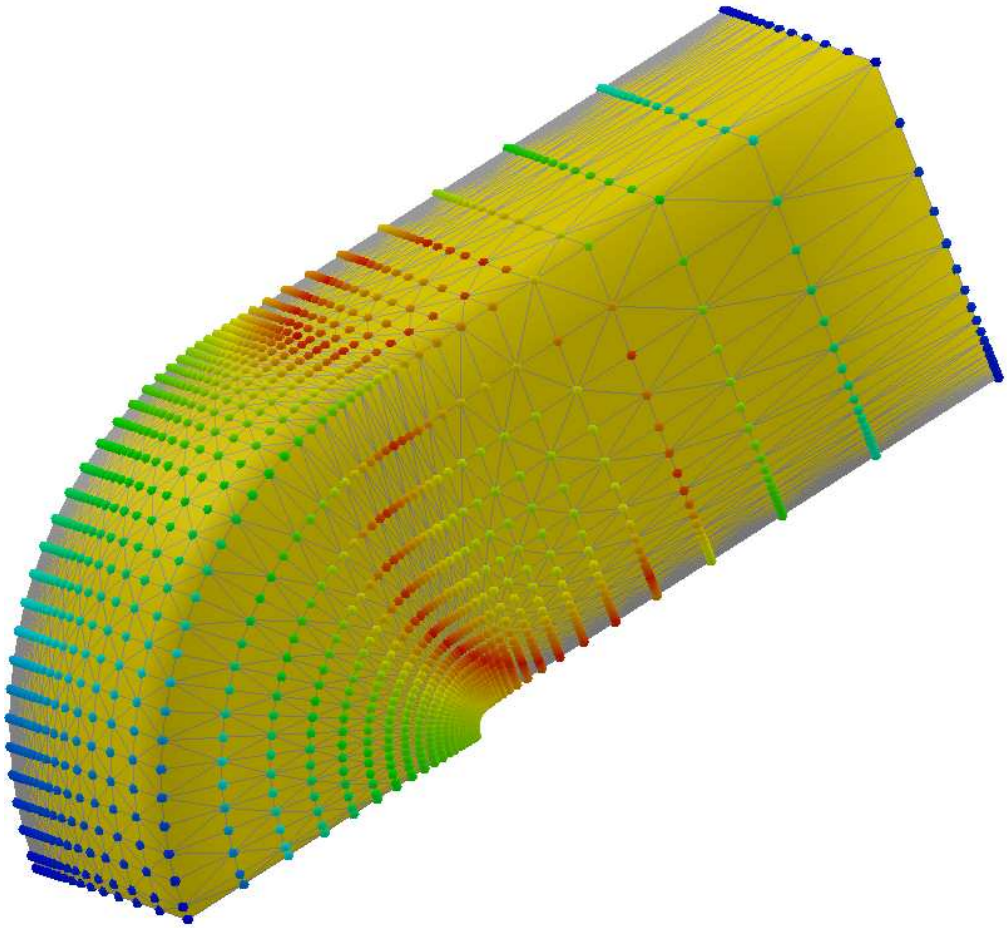

Figura 7.11: Blunto, 40.921 vértices, 187.318 tetraedros, CHF nível 3.

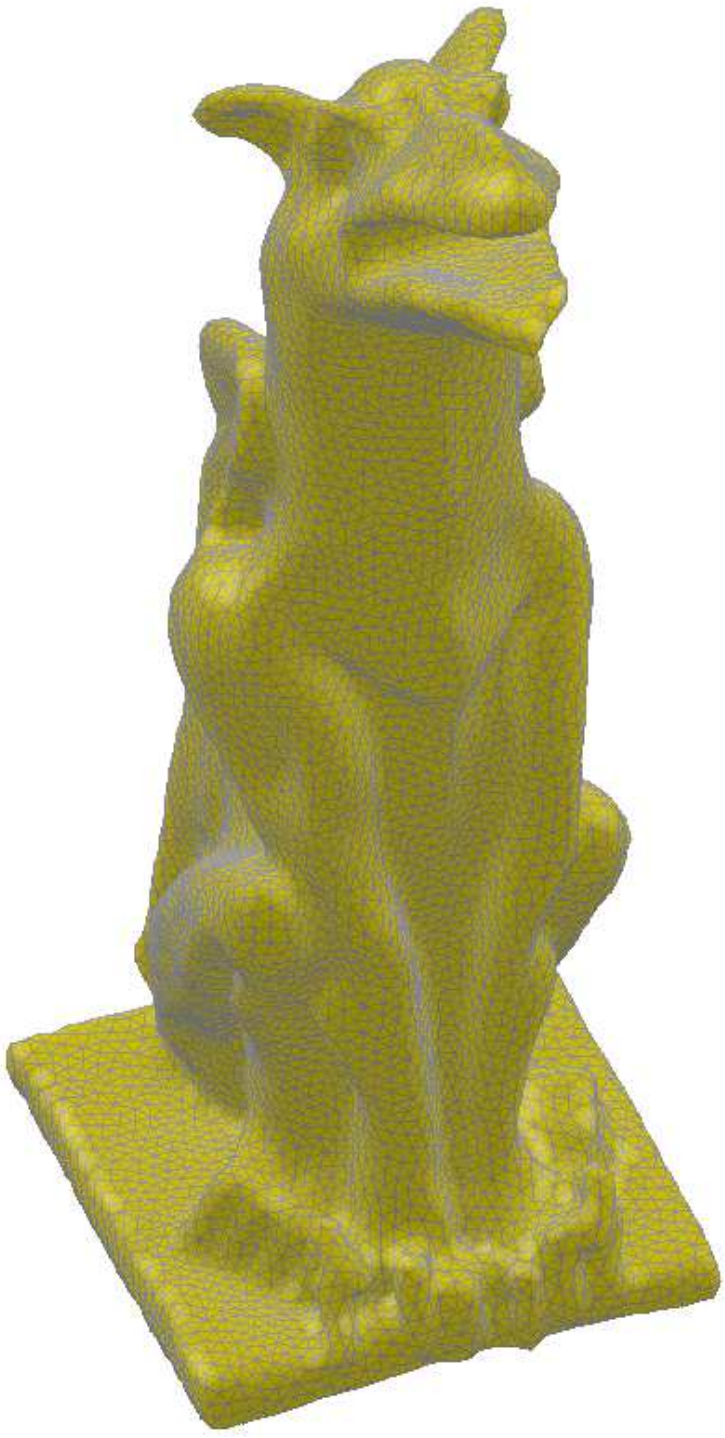

Figura 7.12: Gargoyle, 48.553 vértices, 258.229 tetraedros, CHF nível 3.

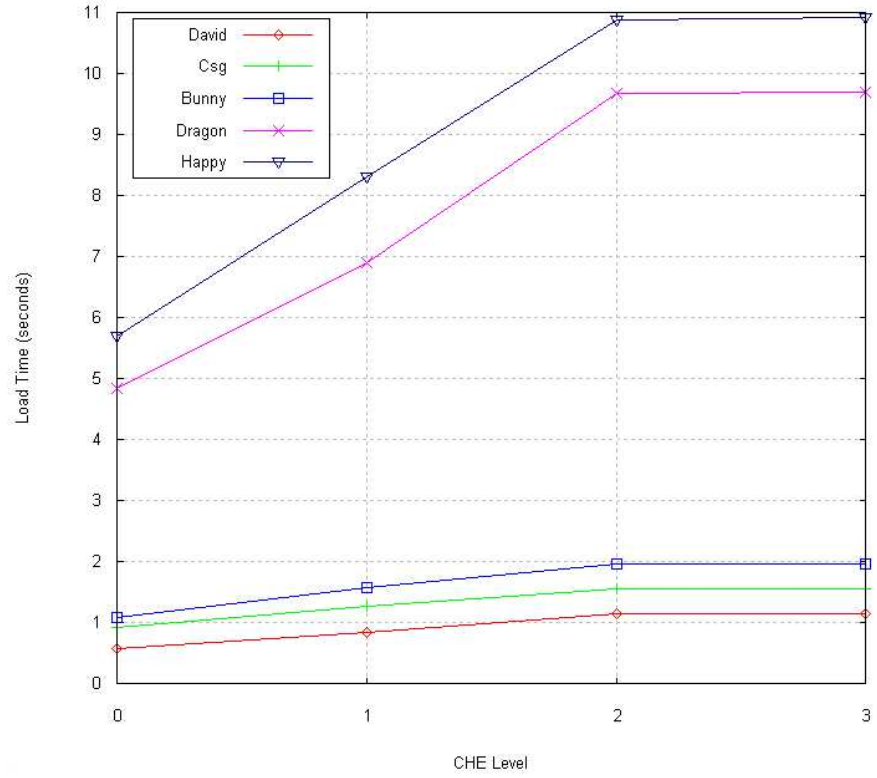

Figura 7.13: Tempo de carregamento dos níveis CHE

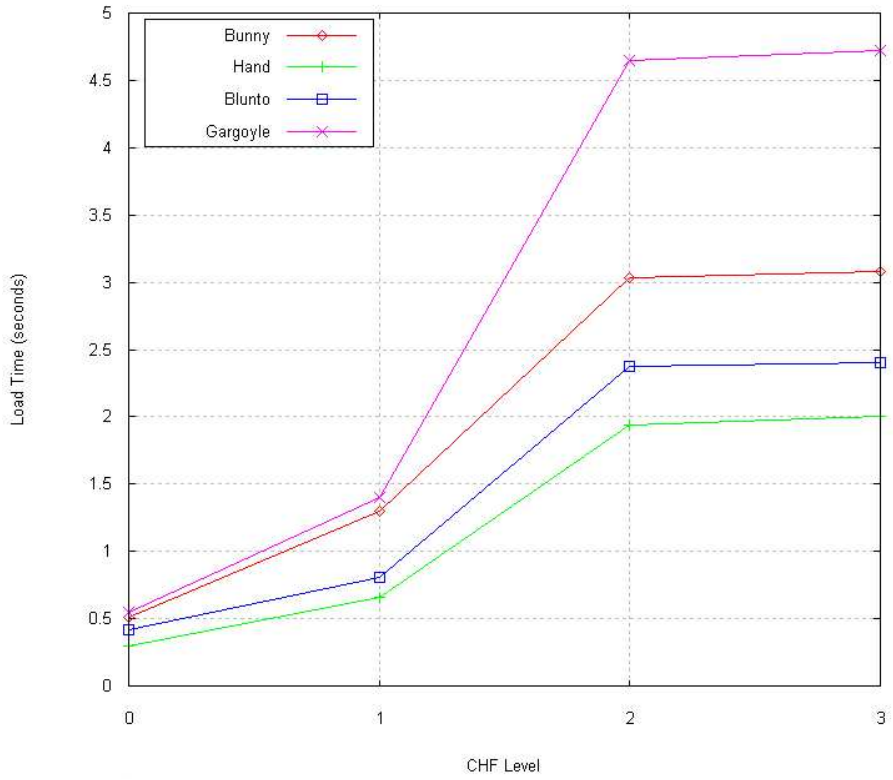

Figura 7.14: Tempo de carregamento dos níveis CHF

Referências Bibliográficas

[Baumgart 1975] BAUMGART, B. G.. **A Polyhedron Representation for Computer Vision**. AFIPS National Computer Conference, 44:589–596, 1975. 3.1

[Bertrand and Dufourd 1994] BERTRAND, Y.; DUFOURD, J.-F.. **Algebraic Specification of a 3D–Modeler Based on Hypergraphs**. CVGIP Graphical Models and Image Processing, 56:29–60, 1994. 3.3

[Braid *et al.* 1980] BRAID, I. C.; HILLYARD, R. C. ; STROUD, I. A.. **Stepwise construction of polyhedra in geometric modeling**. In: Brodlie, K. W., editor, MATHEMATICAL METHODS IN COMPUTER GRAPHICS AND DESIGN, p. 123–141. Academic Press, 1980. 3.1

[Brisson 1993] BRISSON, E.. **Representing Geometric Structures in d Dimensions: Topology and Order**. Discrete and Computational Geometry, 9:387–426, 1993. 3.3

[Campagna *et al.* 1998] CAMPAGNA, S.; KOBBELT, L. ; SEIDEL, H.-P.. **Directed edges: A scalable representation for triangle meshes**. Journal of Graphics Tools, 3(4):1–11, 1998. 3.1

[Castelo *et al.* 1992] CASTELO, A.; LOPES, H. ; TAVARES, G.. **Handlebody Representation for Surfaces and Morse Operators**. In: CURVES AND SURFACES IN COMPUTER VISION GRAPHICS III, p. 270–283, 1992. 3.1

[Cavalcanti *et al.* 1997] ROMA CAVALCANTI, P.; CARVALHO, P. C. P. ; MARTHA, L. F.. **Non–Manifold modelling: an approach based on spatial subdivision**. Computer–Aided Design, 29(3):209–220, 1997. 3.4

[Cormen *et al.* 1990] CORMEN, T. H.; LEISERSON, C. H. ; RIVEST, R. L.. **Introduction to Algorithms**. McGraw–Hill, New York, 1990. 2.8

[Dobkin and Laszlo 1989] DOBKIN, D. P.; LASZLO, M. J.. **Primitives for the manipulation of three–dimensional subdivisions**. Algorithmica, 4:3–32, 1989. 3.2

[Floriani and Hui 2003] DE FLORIANI, L.; HUI, A.. **A scalable data structure for three–dimensional non–manifold objects**. In: SYMPOSIUM ON GEOMETRY PROCESSING, p. 72–82. ACM, 2003. 2.7, 3.4

[Guibas and Stolfi 1985] GUIBAS, L. J.; STOLFI, J.. **Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams**. Transactions on Graphics, 4:74–123, 1985. 3.1, 3.2

[Gumhold *et al.* 1999] GUMHOLD, S.; GUTHE, S. ; STRAŠER, W.. **Tetrahedral mesh compression with the cut–border machine**. In: VISUALIZATION, p. 51–58. IEEE, 1999. 6.2, 6.2

[Gursoz *et al.* 1990] GURSOZ, E. L.; CHOI, Y. ; PRINZ, F. B.. **Vertex–Based Representation of Non–Manifold Boundaries**. In: Turner, J. U.; Wozny, M. J. ; Preiss, K., editors, GEOMETRIC MODELING FOR PRODUCT ENGINEERING, p. 107–130. Elsevier, 1990. 3.4

[Lage *et al.* 2005] LAGE, M.; LEWINER, T.; LOPES, H. ; VELHO, L.. **CHE: A scalable topological data structure for triangular meshes**. Technical report, PUC — Rio de Janeiro, 2005. 1

[Lee and Lee 2001] LEE, S.; LEE, K.. **Partial Entity Structure: A Compact Non–Manifold Boundary Representation Based on Partial Topological Entities**. In: SOLID MODELING AND APPLICATIONS, p. 159–170. ACM, 2001. 3.4

[Lienhardt 1994] LIENHARDT, P.. **N–dimensional Generalized Combinatorial Maps and Cellular Quasi–Manifolds**. Journal of Computational Geometry & Applications, 4:275–324, 1994. 3.3

[Lopes 1996] LOPES, H.. **Algorithm to build and unbuild 2 and 3 dimensional manifolds**. PhD thesis, Department of Mathematics, PUC–Rio, 1996. 3.1, 3.1, 7

[Lopes and Tavares 1997] LOPES, H.; TAVARES, G.. **Structural operators for modeling 3–manifolds**. In: Hoffman, C.; Bronsvort, W., editors, SOLID MODELING AND APPLICATIONS, p. 10–18. ACM, 1997. 3.2, 4.4, 5.1, 5.4

[Mäntylä 1988] MÄNTYLÄ, M.. **An Introduction to Solid Modeling**. Computer Science Press, Rockville, 1988. 3.1, 3.1, 4.1

[Nonato *et al.* 2001] NONATO, G.; MINGHIM, R.; DE OLIVEIRA, M. C. F. ; TAVARES, G.. **A Novel Approach for Delaunay 3D Reconstruction with a comparative analysis in the Light of Applications**. Computer Graphics Forum, 20(2):161–174, 2001. 3.4

[Paoluzzi *et al.* 1993] PAOLUZZI, A.; BERNARDINI, F.; CATTANI, C. ; FERRUCCI, V.. **Dimension–independent modeling with simplicial complexes**. Transactions on Graphics, 12(1):56–102, 1993. 3.3

[Pesco *et al.* 2003] PESCO, S.; LOPES, H. ; TAVARES, G.. **A Stratification Approach for Modeling 2–cell complexes**. Computers & Graphics, 28(2):235–247, 2004. 3.4

[Rossignac and O'Connor 1990] ROSSIGNAC, J.; O'CONNOR, M. A.. **SGC : A Dimension Independent Model for Pointsets with Internal Structures and Incomplete Boundaries**. In: Turner, J. U.; Wozny, M. J. ; Preiss, K., editors, GEOMETRIC MODELING FOR PRODUCT ENGINEERING, p. 145–180. Elsevier, 1990. 3.4

[Rossignac *et al.* 2001] ROSSIGNAC, J.; SAFONOVA, A. ; SZYMCZAK, A.. **3D Compression Made Simple: Edgebreaker on a Corner–Table**. In: SHAPE MODELING INTERNATIONAL, p. 278–283. IEEE, 2001. 3.1

[Vieira 2003] VIEIRA, A. W.. **A topologocal approach for mesh simplification**. Master's thesis, Department of Mathematics, PUC–Rio, 2003. 2, 6.1, 7

[Weiler 1986] WEILER, K. J.. **Topological Structures for Geometric Modeling**. PhD thesis, Rensselaer Polytechnic Institute, New York, USA, 1986. 3.4, 5.1

[Wu 1989] WU, S. T.. **A new combinatorial model for boundary representation**. Computers & Graphics, 13(4):477–486, 1989. 3.4

[Wu 1992] WU, S. T.. **Non–manifold data models: implementation issue**. In: MICAD, Computer Graphcis and Computer Aided Technologies, p. 37–56, 1992. 3.4

[Yamaguchi and Kimura 1995] YAMAGUCHI, Y.; KIMURA, F.. **Non–Manifold Topology Based on Coupling Entities**. Computers & Graphics, 15(1):42–50, 1995. 3.4