Chapter 7

Mesh representation

All implementations in the previous chapters also manipulate meshes of triangles and tetrahedra. Since the focus of our work has been on *changing* meshes, a data structure has been developed where change operations are easy to specify and implement. In this chapter we will discuss the data structure, and explain how other parts of the program are grouped around it. It only applies to simplicial meshes of any dimension, for example, triangle and tetrahedron meshes. The data structure makes a distinction between how the connectivity of the mesh—objects connected with pointers—is stored, and its abstract definition—ordered sequences of vertices, so-called *simplexes*. Such simplexes are also used to describe triangulations abstractly in the field of algebraic topology [29].

Subdivisions, be them triangulations, tetrahedralizations or more general complexes of polyhedral cells, are usually represented by objects connected with pointers. Many such data structures exist for storing subdivisions of the plane, for example the doubly connected edge list [31], and the quad edge structure [48]. These structures all store the connectivity in slightly different ways at slightly different memory costs. Memory usage is an important issue when manipulating large meshes, so Campagna et al. [23] propose a triangle mesh representation where the choice between computation costs and memory costs can be made at compile time.

For 3D subdivisions, Dobkin and Laszlo [39] describe a data structure that can represent general complexes of cells. The cells can have any shape and may be infinite; the only restriction is that they must meet properly. The central notion of their data structure is the facet-edge: it represents the combination of a facet (a 2-dimensional cell) and an edge (a 1-dimensional cell). A cell complex is stored as a set of objects, each representing a single facet-edge. Every object contains references to the four adjacent facet-edge objects: the next and previous edge of the same face, and the next and previous facet that is incident with the same edge. Since neighboring facet-edges are stored explicitly, it is very easy and efficient to traverse all the facets incident to an edge, and all edges contained in a facet. Mücke [68] uses a simplified version of the facet-edge structure for maintaining the connectivity of tetrahedral meshes.

Brisson [16] proposes a generalization of the concept of facet-edge to d dimensions: mesh features are represented by so-called *cell tuples*. A cell tuple is a tuple $(c_0, \ldots, c_d)$,

where each c_j represents a j-dimensional mesh feature, and $c_i \subset c_{i+1}$. In 3D, a cell-tuple represents a vertex as part of a specific edge and a specific face. For $k = 0, \ldots, d$ the switch operator is defined: $\text{switch}_k(t)$ is the unique cell tuple that agrees with t except in its kth component. For example, in 3D, the switch_3 operator moves from a vertex of a volumetric cell to the same vertex as part of the same edge and face, but from a neighboring volumetric cell. Data structures such as the facet-edge structure discussed above, the edge algebra discussed by Guibas and Stolfi [48] and various other mesh data structures [61, 99] can be expressed in terms of cell tuples.

In summary, meshes are typically represented by objects connected by pointers. Relations between objects, such as incidence, inclusion and neighborhood, are maintained by storing pointers between these objects. This structure allows for efficient traversal of the mesh: jumping between mesh features is a matter of following pointers. In this sense, our data structure resembles much of the previous work. However, we have chosen to make explicit what the mesh connectivity objects represent. This makes it possible to specify correctness of the data structure, prove algorithms dealing with meshes correct, and formally specify what change operations should do. Moreover, implementing such operations is easy. Two operations are provided to change the mesh connectivity, change-elements and replace-elements. These are generic operations, and can be used to implement high level mesh operations. All code for maintaining mesh connectivity is concentrated in these two routines, enhancing the modularity of the total system.

In this chapter we first discuss the underlying abstract mesh representation. This representation uses abstract oriented simplexes, a basic concept in algebraic topology [29]. Then we discuss how connectivity is stored in the program, and how it can be modified, in other words, how change-elements and replace-elements are implemented. Finally, we show how the rest of the system interfaces with mesh changes.

7.1 Abstract oriented simplexes

Domains with general shapes in the Finite Element Method are usually represented using *unstructured* meshes. These meshes consist of triangles (in 2D) or tetrahedra (in 3D). In *conforming* Finite Element Methods, shape functions should be admissible as solutions to the original, continuous problem. In the case of elastic problems, this implies that the functions should be piecewise continuously differentiable. This continuity condition (also known as *compatibility condition*), implies that the common interface of two elements should also be a mesh feature. In other words, mesh elements should be *properly joined.*

In a triangulated or tetrahedral mesh, mesh features are formed by convex hulls of vertices, so-called *geometric simplexes*. For example, let $\mathbf{a}_1, \ldots, \mathbf{a}_4$ be an affinely independent set of points in $\mathbb{R}^d$, then $\text{conv}\{\mathbf{a}_1, \mathbf{a}_2\}$, the convex hull of $\mathbf{a}_1$ and $\mathbf{a}_2$, is an edge, $\text{conv}\{\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\}$ is a triangle, and $\text{conv}\{\mathbf{a}_1, \ldots, \mathbf{a}_4\}$ is a tetrahedron. Proper joining of simplexes can be expressed as follows. Let $\mathbf{a}_1, \ldots, \mathbf{a}_k \in \mathbb{R}^d$ and $\mathbf{b}_1, \ldots, \mathbf{b}_l \in \mathbb{R}^d$, then $\text{conv}\{\mathbf{a}_1, \ldots, \mathbf{a}_k\}$ and $\text{conv}\{\mathbf{b}_1, \ldots, \mathbf{b}_l\}$ are properly joined if

$$\text{conv}\{\mathbf{a}_1, \ldots, \mathbf{a}_k\} \cap \text{conv}\{\mathbf{b}_1, \ldots, \mathbf{b}_l\} = \text{conv}\, S,$$

where $S \subset \{a_1, \ldots, a_k, b_1, \ldots, b_l\}$. Properly joined simplexes are demonstrated in Figure 7.1.

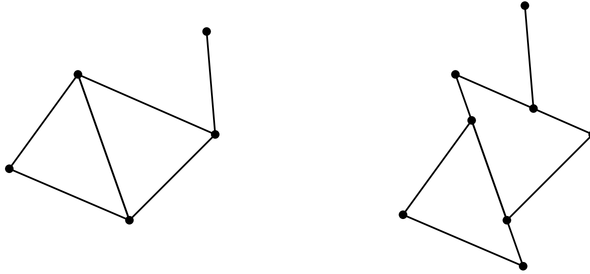

Figure 7.1: Properly joined simplexes (left), and improperly joined ones (right)

We can see that properly joined geometric simplexes are characterized by their sets of vertices. Hence, for reasoning with simplexes, it suffices to consider the discrete set of their vertices, and disregard the continous nature of convex subsets of $\mathbb{R}^d$. If we only consider sets of vertices, then the type of the vertices themselves is not relevant. Therefore, we will assume for the remainder of the chapter that vertices come from some set $\mathcal{V}$, which is left unspecified.

Simplexes can have orientations. For example, an edge can have two directions, and a triangle can have a normal pointing in two directions. This orientation is related to ordering of the vertices: if two vertices in a triangle are swapped, the direction of the normal is flipped. The orientation of a simplex can also be defined in terms of swaps. Let $a_0, \ldots, a_k \in \mathcal{V}$ be a sequence of $k+1$ vertices, for $k \geq 1$. A permutation π of these vertices may be decomposed into a number of swaps. If this number is even, then π is an *even* permutation, otherwise it is an *odd* permutation. The positive simplex $\langle a_0, \ldots, a_k \rangle$ is formed by the equivalence class of all even permutations of $a_0, \ldots, a_k$, i.e.,

$$\langle a_0, \ldots, a_k \rangle = \{\pi(a_0, \ldots, a_k) : \pi \text{ is an even permutation}\}.$$

Analogously, the equivalence class of uneven permutations forms the other orientation

$$-\langle a_0, \ldots, a_k \rangle = \{\pi(a_0, \ldots, a_k) : \pi \text{ is an odd permutation}\}.$$

The number k is also called the *dimension* of the simplex. 1-simplexes correspond to edges, 2-simplexes to triangles and 3-simplexes to tetrahedra. The above definition requires $k > 0$. For simplexes of one vertex, we simply assume that they exist in two orientations.

Containment of oriented abstract simplexes is defined with help of the subsimplex operation. This operation is defined as follows.

$$\mathsf{subsimplex}_{a_j} \langle a_0, \ldots, a_k \rangle = (-1)^j \langle a_0, \ldots, a_k \backslash a_j \rangle, \quad a_0, \ldots, a_k \in \mathcal{V}, k \geq 1.$$

The notation $a_0, \ldots, a_k \backslash \nu$ means the sequence $a_0, \ldots, a_k$ with ν removed. This definition is independent of the representative chosen. This operation implies an inclusion relation. We have $\sigma \subset \tau$, if $\sigma = \tau$ or when there is some $\nu \in \tau$ such that $\sigma \subset \mathsf{subsimplex}_\nu \tau$.

We call the set of abstract simplexes K an oriented d-dimensional pseudo-manifold, or a *simplicial mesh* if the following conditions hold:

1. if τ in K and $\sigma \subset \tau$ then σ in K
2. Every σ in K is a subsimplex of some $\tau \in K$, where τ has dimension d.
3. if $\sigma \in K$ is a $(d-1)$-simplex, then it is subsimplex of only one d-simplex.

Two d-simplexes τ_1 and τ_2 are neighbors if there is a $(d-1)$-simplex $\rho \subset \tau_1$ such that $-\rho \subset \tau_2$. The boundary of a simplicial mesh K, denoted by ∂K is formed by the set of $d-1$ simplexes whose opposite orientation is not part of K. Since all simplexes in a pseudo-manifold are part of some d-simplex, we can characterize the structure by its set of d-simplexes.

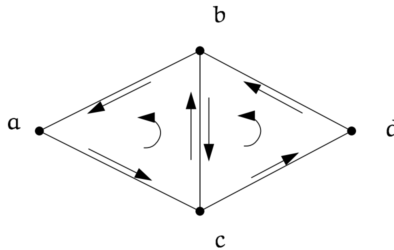

Figure 7.2: A simple 2-dimensional pseudo-manifold. We have $T = \{abc, bdc\}$ (leaving out the angled brackets in the notation of simplexes), and $K = T \cup \{ab, bc, ca, cb, bd, dc, a, b, c, d, -a, -b, -c, -d\}$. The boundary of K is formed by $\{ab, ca, bd, dc\}$; the other edges (bc and $-bc$) form a pair that connect abc and bcd. The orientation of triangles and edges are indicated with arrows.

7.2 Representing the mesh

The mesh representation discussed in the previous section can be directly implemented. In the next two sections, we show how this is done, both using class declarations (in C++ syntax) and pseudo-code. In this pseudo-code, we will refer to simplexes with the greek letters σ and τ. The variable t always refers to an Element object representing a d-simplex, and the variable f always refers to a Face object representing a $(d-1)$-simplex. In general variables are denoted by words printed in italic. In the pseudo-code we will equate maps, search structures that store a value v for some keys k, with a set of key/value tuples. This is done for the sake of notational convenience. In practice, such search structures will typically be implemented by balanced trees. We assume that sets of key/value tuples can be indexed, and that it supports the method keys that returns all keys in the map, and the method erase, that removes a single (key,value) tuple from the set. Examples of the use of these maps are given here.

$m \leftarrow \{(k_1, v_1), (k_2, v_2)\}$
m.keys() (* *returns* $\{k_1, k_2\}$ *)
$m[k_1]$ (* *returns* v_1 *)
$m[k_1] \leftarrow w_1$ (* *changes the value corresponding to* k_1 *)
$m[k_3] \leftarrow v_3$ (* *adds the tuple* (k_3, v_3) *)
m.erase(k_2) (* *removes* (k_2, v_2) *)

If $\mathcal{V}$ is totally ordered, then we can define a canonical representation for each simplex. Let $a_0, \ldots, a_k \in \mathcal{V}$. We can sort the vertices in a oriented simplex, while counting the number of swaps s, and take $(-1)^s \operatorname{sort}(a_0, \ldots, a_k)$ as the representative of $\langle a_0, \ldots, a_k \rangle$. In effect, this the canonical representation translates a k-simplex in a $k+2$ tuple, consisting of the $k+1$ vertices and the value of s. Since the elements of each tuple can be ordered, the tuples themselves can also be ordered, e.g. by the lexicographic order. This implies that the canonical representation can be used as a key in a lookup structure. In this way, we can create tables of objects with simplexes as keys.

The canonical representation of a simplex can be used to implement it. Assuming that there is some type Vertex representing vertices, the data of the Simplex type may expressed (in C++ syntax) as follows.

```
class Simplex {
  Vertex vertices[MAXDIMENSION+1];
  int dimension;
};
```

Let us assume for the remainder that a Simplex object can be created from a sign $q \in \{-1, 1\}$ and a sequence of vertices $b_0, \ldots, b_k$, yielding the canonical representation $p\langle a_0, \ldots, a_k \rangle$ with $a_0 < \cdots < a_k$ and $p \in \{-1, 1\}$. Let σ and τ be Simplex objects, j an integer from $0, \ldots, k$, and v and w Vertex objects. Then the following operations can be defined and implemented for the Simplex type.

- σ.count() returns $k+1$, the number of vertices in σ.
- σ.dimension() returns k, the dimension of the simplex.
- σ.index (v) returns j such that $a_j = v$.
- σ.sign () returns p.
- σ.vertex(j) returns a_j.
- σ.subn (j) returns $(-1)^j q\langle a_0, \ldots, a_k \backslash a_j \rangle$, the jth subset of σ.
- σ.subv (v) returns $\text{subsimplex}_v \sigma$.
- σ.mate() returns $-\sigma$.
- σ.substituted (v, w) returns σ with v replaced by w in the vertices of the simplex.
- compare (σ, τ) is a signed comparison of σ and τ. It can be implemented by lexicographic ordering.

- σ.sup (ν) returns $p\langle \nu, a_0, , \ldots, a_k\rangle$, the unique $(k+1)$-simplex containing both the vertex ν and the simplex σ.

Since a d-dimensional pseudo-manifold is characterized by a set of d-simplexes, a simplicial mesh can be succinctly specified as a set of Simplex objects of dimension d. However, traversing the elements of a mesh cannot be done efficiently with this representation. Therefore, d-simplexes and $(d-1)$-simplexes are also represented as objects, i.e. chunks of memory with a unique identity that can be referenced to by means of pointers. The base class for both objects is Mesh-feature. It contains the simplex that it is supposed to represent. One derived class represents d-simplexes, and is called Element, by analogy with naming of Finite Elements. Objects of the class Face represent $(d-1)$-simplexes.[1] The definition of Mesh-feature in C++ notation is as follows.

```
class Mesh_feature {
  Simplex simplex;
};
```

Each d-simplex contains $d+1$ faces, so the Element object has $d+1$ pointers to Face objects.

```
class Element : public Mesh_feature {
  Face * faces[MAXDIMENSION+1];
};
```

A face of an element is obtained by removing one vertex from its simplex. Faces and vertices in an element are related. This relation is used in opposite-vertex and opposite-face methods of Element objects. The method opposite-vertex for an Element object e takes a face f from e.faces, and returns a vertex from e.simplex such that

$$f.\mathsf{simplex} = e.\mathsf{simplex.subv}(e.\mathsf{opposite\text{-}vertex}(f))$$

Similarly, the function opposite-face, takes a vertex ν from e.simplex, and returns a Face object from e.faces such that

$$e.\mathsf{opposite\text{-}face}(\nu).\mathsf{simplex} = e.\mathsf{simplex.subv}(\nu)$$

Both functions are also illustrated in Figure 7.3.

The faces variable must contain Face objects. The following invariant specifies in what order they are stored.

$$t.\mathsf{faces}[i].\mathsf{simplex} = t.\mathsf{simplex.subn}(i), \qquad i = 0, \ldots, d. \tag{7.1}$$

In a pseudo-manifold, each face is in exactly one d-simplex. Hence we may store pointers from Face objects to Element objects. The Face object also stores a pointer to its mate, the Face object with the opposite orientation

[1] This is in contrast with traditional terminology for simplicial complexes, where simplexes of all dimensions are called “faces.”

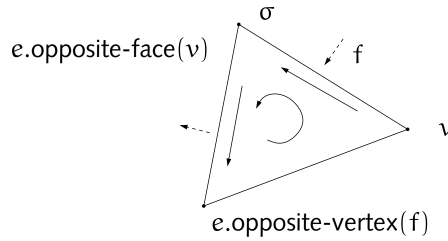

Figure 7.3: The body of the cycle-around algorithm illustrated: an element e is entered through f (dotted arrow), and left through e.opposite-face (ν) (other dotted arrow). In this case, σ is a 0-simplex, i.e. a vertex. The orientation of the triangle and the edges are indicated with arrows.

```
class Face : public Mesh_feature {
  Element *element;
  Face *mate;
};
```

The element pointer in a Face object f satisfies the following invariant

$$f \in \text{f.element.faces}, \tag{7.2}$$

where we treat the array faces as a set. The mate field of a Face object f satisfies

$$\text{f.mate} = \text{null} \vee \text{f.mate.simplex} = -\text{f.simplex}. \tag{7.3}$$

The connectivity of a simplicial mesh then is a collection of Element and Face objects such that invariants (7.1) to (7.3) are satisfied, and each d and $(d-1)$-simplex is represented by exactly one Element and Face object respectively, i.e., for all Mesh-feature objects t, u in the mesh we have

$$\text{t.simplex} = \text{u.simplex} \implies t = u. \tag{7.4}$$

This connectivity information is sufficient to traverse the mesh. We give the example of traversing Element objects incident with one particular $(d-2)$-simplex. In 2D, this routine traverses all triangles incident with a vertex, and in 3D all tetrahedra incident with an edge. It takes a Face object entry as argument, and a number $j \in \{0, \ldots, d-1\}$. It returns a set of d-simplexes that contain entry.simplex.subn(j). Termination and correctness of the algorithm can be proved using the integrity of the data structure, and properties of the simplicial mesh.

procedure cycle-around (entry: Face, $j : \{0, \ldots, d-1\}$)
  star $\leftarrow \emptyset$
  f $\leftarrow$ entry
  $\nu \leftarrow$ entry.simplex.vertex(j)
  while f $\neq$ null

```
    e ← f.element
    exit ← e.opposite-face(v)
    v ← e.opposite-vertex(f)
    star.add (e)
    f ← exit.mate
    if f = entry:
      f ← null
  return star
```

This code is also illustrated in Figure 7.3.

This routine builds a set of Element objects that contain $\sigma = \text{entry.simplex.subn}(j)$. This can be seen by considering the following loop invariant.

$$f = \text{null} \vee \text{subsimplex}_v(\text{f.simplex}) = \sigma.$$

If $f \neq \text{null}$, then $\text{f.simplex} = \sigma.\text{sup}(v)$, and $\text{e.simplex} = \sigma.\text{sup}(v).\text{sup}(w)$ for some vertex w. Hence $\text{exit.simplex} = -\sigma.\text{sup}(w)$. In the next step, either the loop exits because $\text{exit.mate} = \text{null}$, or f and v are changed such that the invariant holds again.

In addition, we see that the loop adds a sequence of Element objects with d-simplexes $(\tau_1, \tau_2, \ldots)$ to star. These d-simplexes are of the form

$$\sigma.\text{sup}(p_1).\text{sup}(p_2), \sigma.\text{sup}(p_2).\text{sup}(p_3), \sigma.\text{sup}(p_3).\text{sup}(p_4), \ldots$$

for a sequence of vertices $(p_1, p_2, \ldots)$. All simplexes of the sequence are unique. To see this, suppose that $\tau_j = \tau_i$ for some $j \leq i$. In other words

$$\sigma.\text{sup}(p_j).\text{sup}(p_{j+1}) = \sigma.\text{sup}(p_i).\text{sup}(p_{i+1}).$$

This implies $p_i = p_j$ and $p_{j+1} = p_{i+1}$. Their predecessors in the sequence are $\tau_{j-1} = \sigma.\text{sup}(p_{j-1}).\text{sup}(p_j)$ and $\tau_{i-1} = \sigma.\text{sup}(p_{i-1}).\text{sup}(p_j)$ respectively. Since τ_{j-1} and τ_{i-1} both contain the face $-\sigma.\text{sup}(p_j)$ they must be equal, implying that $p_{j-1} = p_{i-1}$. This argument can be continued inductively, until we have $\tau_1 = \tau_{i-j+1} = \sigma.\text{sup}(p_1).\text{sup}(p_2)$. The integrity of the data structure implies that entry is the only Face object whose simplex is $\sigma.\text{sup}(p_1)$. Therefore, if $i < j$ the if statement would have aborted the loop before τ_i is added in the i-th step. Therefore $i = j$. Since the mesh only contains finitely many d-simplexes, the loop must terminate.

7.3 Changing the mesh

In this section we show how mesh connectivity objects can changed in a generic fashion. This is done by two routines, replace-elements and change-elements. First we show how replace-elements can be implemented. In situations where the number of elements does not change, a different routine with additional desirable properties can be used. This is the change-elements routine. Finally, we show how cuts can be expressed with change-elements.

Every change in the mesh can be encoded as removing existing elements, exposing more of the boundary of the mesh, and attaching new elements to the boundary. The actual connectivity information is stored in Face objects, since their mate fields link neighboring elements. To update these fields properly, it is necessary to store the boundary of the pseudo-manifold. This is done with the following data structure for the mesh connectivity.

```
class Mesh_connectivity {
  set<Element*> elements;
  map<Simplex, Face*> boundary;
};
```

This definition uses the generic types `set` and `map`. The variable elements is a set of Element objects. The variable boundary maps simplexes to their Face objects for all boundary faces.

Let the $(d-1)$-simplexes of a set of d-simplexes T be given by

$$\mathsf{faces}(T) = \{\, \sigma : \sigma = \mathsf{subsimplex}_\nu(\tau), \nu \in \tau, \tau \in T \,\}.$$

Then, the boundary map of a simplicial mesh formed by T may be characterized as

$$\mathsf{boundary}(T) = \{(\sigma, f) : f.\mathsf{simplex} = \sigma, \wedge - \sigma \notin \mathsf{faces}(T) \wedge \sigma \in \mathsf{faces}(T)\} \qquad (7.5)$$

The primary mesh change operation is replacing elements. The simplest way to implement it is by removing elements one-by-one, and adding new elements one-by-one. This is achieved by the following procedure.

procedure replace-elements (mesh: Mesh-topology,
  old-objects: set of Element, new-simplexes: set of Simplex):
 for e **in** old-objects:
  remove-element (mesh, e)
 for τ **in** new-simplexes:
  add-element (mesh,τ)

When a single element is added, the connectivity can be maintained by removing boundary faces that attach to the new element, and adding other new faces of the element to the boundary.

procedure add-element (mesh: Mesh-topology, τ: Simplex)
 $e \leftarrow$ new Element(τ)
 mesh.elements $\leftarrow$mesh.elements $\cup \{e\}$
 for j **in** $0, \ldots, d$:
  $\sigma \leftarrow \tau$.subn(j)
  f $\leftarrow$ new Face(σ)
  f.element $\leftarrow e$

```
    e.faces[j] ←f
    if −σ ∈ mesh.boundary.keys():
      f.mate ←mesh.boundary[−σ]
      f.mate.mate ←f
      mesh.boundary.erase (σ)
    else :
      mesh.boundary[f.simplex] ←f
      f.mate ← null
```

Similarly, the boundary can be updated during element removal.

```
procedure remove-element (mesh, e)
  for f in e.faces:
    if f.mate:
      mesh.boundary[−f.simplex] ← f.mate
      f.mate.mate ← null
      f.mate ← null
    else :
      mesh.boundary.erase(f.simplex)
  mesh.elements ← mesh.elements\{e}
```

This code maintains mesh connectivity, but is not very efficient and replaces all Face objects, even the ones that were not changed. The following improvements solve these problems. First, in some cases, a mesh change modifies elements, but may leave certain faces in place. In the code shown below, these faces are maintained, so that pointers to these faces remain valid after the change. It achieves this by remembering old faces, and reusing those that also occur in the new configuration.

```
procedure replace-elements (mesh: Mesh-topology, old-objects: set of Element,
    new-simplexes: set of Simplex):
  oldfacemap ← { (f.simplex, f) : f ∈ e.faces, e ∈ old-objects }
  newfacemap ← {(σ, f) : σ = τ.subn(j), j = 1, . . . , d, τ ∈ new-simplexes,
    f = (if σ ∈ oldfacemap.keys()) : oldfacemap[σ] else: null)}
  discard ← oldfacemap\newfacemap
  (*)
  for (σ, f) in discard:
    (**)
    if f.mate:
      f.mate.mate ← null
      f.mate ← null
      mesh.boundary[f.simplex] ← f.mate
    else :
      boundary.erase (f)
  mesh.elements ← mesh.elements\old-objects
```

```
for τ in new-simplexes:
  e←new Element (τ)
  mesh.elements ← mesh.elements ∪ {e}
  for j in 0, . . . , d:
    (σ, f) ← newfacemap[τ.subn(j)]
    if f = null:
      f ← new Face(σ)
      if −σ ∈ mesh.boundary.keys():
        f.mate ← mesh.boundary[−σ]
        f.mate.mate ← f
        mesh.boundary.erase (-σ)
      else :
        mesh.boundary[σ] ← f
    f.element ← e
    e.faces[j] ← f
```

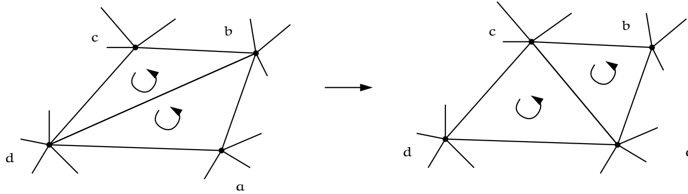

Figure 7.4: An edge flip changes the edge db to ad. This can be encoded as replacing the 2-simplexes $\{bcd, abd\}$ by $\{abc, acd\}$. The elements on the left contain edges ab, bc, cd, and da which are also present after the flip.

This code is still not optimal. For example, in the edge flip from Figure 7.4, the boundary does not change, while orientations of bd and ad are temporarily added to and removed from the boundary. When the boundary is large, these temporary changes may be expensive. They can be prevented by adding matched pairs to newfacemap before processing it. If the following code is added in the place marked with $(*)$ in the previous algorithm, then these unnecessary updates are prevented.

```
for σ in newfacemap.keys ():
  if −σ ∈ newfacemap.keys () ∧ σ.sign() = 1 ∧
      newfacemap[σ] = null ∧ newfacemap[−σ] = null:
    f1 ← new Face(σ)
    f2 ← new Face(−σ)
    f2.mate ← f1
    f1.mate ← f2
    newfacemap[σ] ← f1
    newfacemap[−σ] ← f2
```

Similarly, the updates of the boundary, following $(**)$ in the pseudo-code, only have to be performed when $-\sigma \notin$ discard.keys().

Some types of mesh modifications do not change the number of mesh elements, only their connectivity. For example, in Figure 7.5, two faces are dissected and two other are glued together at the same time, leaving the number of faces and elements invariant. It is possible to implement this operation with replace-elements. However, there is a one-to-one correspondence for every face and element before and after the change, and this correspondence is lost when replace-elements is used. Therefore, we propose a second operation, change-elements that maintains this correspondence. Its argument is a substitution, that is applied to a number of elements. Abstractly speaking, a substitution s is a set of (τ, π)-tuples, where τ is a d-simplex, and $\pi : \mathcal{V} \to \mathcal{V}$ is a substitution on the vertices. If T is set of d-simplexes in the original mesh, then applying the substitution s entails forming the mesh

$$K' = \{\sigma : \sigma \subset \tau, \tau \in T'\} T' = \{\tau \in T : \tau \notin \text{s.keys}()\} \cup \{\pi(\tau) : (\tau, \pi) \in s\}.$$

When implementing this operation, the substitution takes the form of a set of tuples (t, π), where t is an Element object, and π a vertex substitution.

```
procedure change-elements (mesh: Mesh-connectivity,
    substitution: element/node-substitution map):
  oldfaces ←{ t.faces[j] : (t, π) ∈ substitution, j = 0, . . . , d }
  for f in oldfaces:
    if f.mate:
      mesh.boundary[f.simplex] ← f
      f.mate ← 0
      f.mate.mate ← 0
    else :
      mesh.boundary.erase(f.simplex)
  for (e, π) in substitution:
    τ ← e.simplex
    τ′ ← π(τ)
    newfaces ←{ (πσ, f) : f = e.face(j), σ = τ.subn(j), j = 0, . . . , d }
    e.simplex ←τ′
    for j in 0, . . . , d:
      σ′ ← τ′.subn(j)
      f ← newfaces[σ′]
      e.faces[j] ←f
      f.simplex ← σ′
      if −σ ∈ mesh.boundary.keys():
        f.mate ← mesh.boundary[−σ′]
        f.mate.mate ← f
        mesh.boundary.erase[−σ′]
      else :
        mesh.boundary[σ′] ← f
```

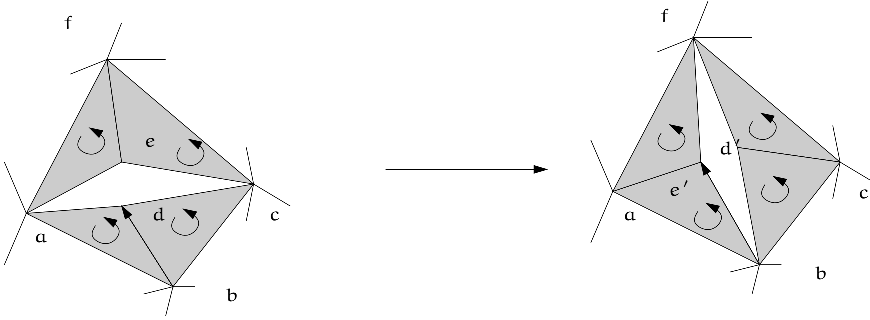

Figure 7.5: Cutting and stitching can be achieved using replace-triangles. The operation shown here can be effected as replacing $\{abd, bcd, cfe, aef\}$ with $\{abe', ae'f, d'cf, bcd'\}$. The operation can also be written as a vertex substitution, e.g. substitute $d := d'$ in abd. By specifying the operation like this, objects can be made persistent. Then bd (left) and be' (right) are represented by the same object, marked in bold.

The primary example of the change-elements operation is the dissect operation, also discussed in Chapter 3, which produces cuts along faces in a simplicial mesh. We show how a vertex substitution for a cut along a surface C can be defined. Let us assume that a simplicial mesh is given as T, a set of oriented d-simplexes, satisfying the conditions for a simplicial mesh, and K is the complex induced by T, i.e.

$$K = \{\sigma : \sigma \subset \tau, \tau \in T\}.$$

We assume that the cut is specified by a set of faces $C \subset \text{faces}(T)$, such that

$$\sigma \in C \implies -\sigma \in C.$$

In other words, C is a set of face pairs from K. The star of a vertex v is the set of all elements incident with v, in other words,

$$\text{star}(v) = \{\tau \in T : v \in \tau\}.$$

Let v be a vertex, and let τ and τ' be elements from $\text{star}(v)$. We say that τ and τ' are (v, C)-connected if there are d-simplexes $\tau = \tau_1, \ldots, \tau_k = \tau'$ in T such that all τ_i contain v, and all τ_i and τ_{i+1} are neighbors for $i = 1, \ldots, k-1$ and

$$\text{faces}(\tau_i) \cap (-\text{faces}(\tau_{i+1})) \notin C, \qquad i = 1, \ldots, k-1.$$

In other words, $\tau_1, \ldots, \tau_k$ is a chain of elements containing v that does not cross C.

The notion of (v, C)-connectedness is an equivalence relation on $\text{star}(v)$, so for each vertex v of T, we may partition $\text{star}(v)$ into equivalence classes. Assume that these classes are given by $S_{v,1}, \ldots, S_{v,l}$ for some $l \geq 1$. Let us assume that a unique vertex $w_{v,i}$ for each equivalence class $S_{v,i}$ is given. In practice this may be a 'copy' of v with a different number, or perhaps a vertex that is slightly displaced with respect to v. Let τ

be a d-simplex, then we define for $\nu \in \tau$ the node substitution

$$\varphi_\tau(\nu) = \begin{cases} \nu & \text{if star}(\nu) \text{ is the single } (\nu, C)\text{-equivalence class,} \\ w_{\nu,i} & \text{if } \tau \in S_{\nu,i}, \text{ for some } 1 \le i \le k \text{ and } l > 1. \end{cases}$$

By construction, the mapping $\nu \mapsto \varphi_{\tau,S}(\nu)$ is injective. We can define a complex K' induced by a set of d-simplexes T' as follows.

$$K' = \{ \sigma : \sigma \subset \tau, \tau \in T' \},$$
$$T' = \{ \langle \varphi_\tau(a_0), \dots, \varphi_\tau(a_k) \rangle : \tau = \langle a_0, \dots, a_k \rangle \in T \}.$$

Since $\nu \mapsto \varphi_\tau(\nu)$ is injective, all elements of T' are d-simplexes, and all $(d-1)$-simplexes are unique within faces(T'). Hence K' is a d-dimensional pseudo-manifold.

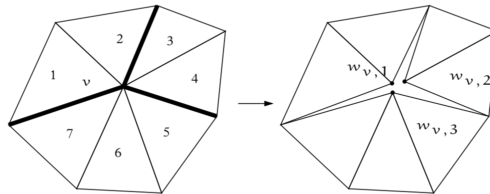

Figure 7.6: A dissection can be expressed as a node substitution. In the above example, the cut surface C (bold) partitions the 2-simplexes incident with ν in 3 sets. Elements 5, 6 and 7 are (ν, C) connected, but 1 and 7 are not. In the result ν is substituted by three different nodes $w_{\nu,1}$, $w_{\nu,2}$ and $w_{\nu,3}$. Elements 5, 6 and 7 share the node $w_{\nu,3}$.

The dissect operation leaves faces that are not in C joined together. If $\sigma \in K$ and $-\sigma \in K$ but $\sigma \notin C$, then the d-simplexes τ_1 and τ_2 containing σ and $-\sigma$ respectively, obviously are (ν, C)-connected for all vertices ν of σ, hence both σ and $-\sigma$ are mapped to corresponding faces σ' and $-\sigma'$. However, not all faces in C also have to end up on the boundary of K'. An example is given in Figure 7.7.

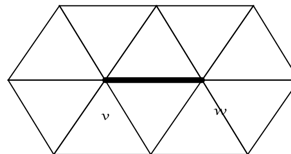

Figure 7.7: Face νw is not dissected by the cut surface marked in bold, since the surface does not split the elements around ν and w into different components.

Finally, the dissect operation implies a connectedness condition. It is reasonable to assume that for $C = \emptyset$, the dissect operation does nothing. This implies that for every vertex ν, star(ν) forms a $(\nu, \emptyset)$-connected component. In other words all elements containing ν should be connected to each other via $(d-1)$-faces. This is a desirable property, for star(ν) can then be found by traversing the mesh starting from an arbitrary $\tau \in$ star(ν).

7.4 Interfacing with mesh connectivity

Both change-elements and replace-elements are characterized by changes to the set of elements and the boundary. Such changes are signaled to other parts of the simulation by the following mechanism. A Mesh-connectivity object maintains a list of Mesh-connectivity-watchers. These are objects that to take some special action upon changes to the connectivity. They can be characterized by a C++ class declaration as follows.

```
class Mesh_connectivity_watcher {
  virtual void process_changed_element (Element*);
  virtual void process_changed_boundary (Face*);
  virtual void init_elements (set<Element*> const*);
  virtual void init_boundary (map<Simplex,Face*> const*);
};
```

When a Mesh-connectivity-watcher is added to a Mesh-connectivity object, it is initialized with the current list of all Element and Face objects. After this initialization, the virtual functions process-changed-boundary and process-changed-element are called for every change to the mesh boundary and every element removed or added.

An example of a Mesh-connectivity-watcher is the following routine

procedure process-changed-boundary (f):
  $\mathbf{n} \leftarrow$ normal of f
  if (f.mate = null) $\wedge$ ($\mathbf{n} \cdot \mathbf{e}_3 < 0$):
    for v in f.simplex.vertices:
      deformation-constraints.fix-node (v)

This code assures that the deformable object is always fixed on one side. All boundary faces pointing in $\mathbf{e}_3$ direction have their faces fixed.

In our implementation, the set of vertices simply is given by the positive integers, with natural ordering. For a linear FEM discretization, simplex vertices and nodes (interpolation conditions) coincide. This fact is exploited by taking vertex numbers as array indices for nodal quantities. For example, in a mesh with m vertices in 3D, nodal quantities like force and displacement are vectors from $\mathbb{R}^{3m}$. Such a vector is stored as an array of floating point numbers. The entries corresponding to a vertex $v \in \mathbb{N}$ are stored at locations $3v$ to $3v + 2$ in the array.

7.5 Discussion

We have presented a data structure for maintaining the connectivity of simplicial meshes in an arbitrary spatial dimension. The data structure can be specified in terms of *abstract simplexes*, ordered sequences of vertices. These simplexes can be represented directly in the computer, and are also used to specify and implement operations changing the mesh connectivity. Properties of the mesh and the integrity of the data structure, which are crucial in proving traversal algorithms correct, can be verified automatically.

We have discussed two operations to change mesh connectivity, replace-elements and change-elements, and have shown how to implement them. Unfortunately, neither change-elements nor replace-elements are guaranteed to deliver valid data structures, unless extra conditions are given on their arguments. An example is in Figure 7.8, where a mesh change is shown that violates Condition (7.4). Catching these mistakes requires storing more information of the mesh, making change operations more expensive. Fortunately, Conditions (7.1) to (7.5) can be checked automatically in a validation routine. Such a validation routine is expensive, but it can help program debugging. Less expensive checks can also help catching errors. For example, some errors can be caught by checking that no key occurs twice when forming newfaces in the replace-elements algorithm.

The algorithms presented do not have optimal performance. For example, the change-elements contains spurious updates of the boundary. Another source of overhead are updates of mesh.elements in replace-elements. During invocations of this routine old Element objects are removed from the mesh, and new ones introduced. The advantage is that it is easy to catch some programming errors: when an Element or Face object is discarded, it may be flagged as "invalid". Bugs caused by using invalid objects can thus be caught automatically. The disadvantage is that every replace-elements call—even if it does not change the number of elements—changes mesh.elements, and will cost $\mathcal{O}(\log(n))$ time, where n is the number of elements in the mesh. This overhead could be eliminated by reusing old Element objects.

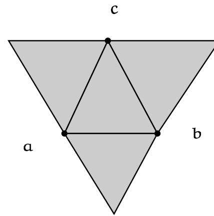

Figure 7.8: Not all invalid mesh changes can be caught. When peforming replace-elements$(\emptyset, \{abc\})$ on the mesh shown above, the d-simplex abc and its faces will represented by two objects.

Only d- and $(d-1)$-dimensional mesh features are identified with objects. This limits its applicability; both for higher order FEM discretizations and for certain relaxation techniques it is necessary to also track edges, which are $(d-2)$-dimensional for $d = 3$, across mesh changes. In a higher order FEM discretization, nodes, i.e. interpolation conditions, are also located on edges of the elements. These nodes are shared by all elements containing that edge, so each edge must be uniquely identified. For a linear FEM interpolation, most off-diagonal entries of the stiffness matrix correspond to force/displacement relations of two nodes connected by an edge. Certain relaxation algorithms exploit matrix structure by traversing the matrix column by column or row by row, for example the Gauss-Seidel iteration [47]. A matrix-free implementation of this algorithm must maintain lists of edges incident to each node. It possible to tracking these edges using a Mesh-topology-watcher instance.