Maintaining mesh connectivity using a simplex-based data structure

Han-Wen Nienhuys

A. Frank van der Stappen

institute of information and computing sciences, utrecht university

technical report UU-CS-2003-018

www.cs.uu.nl

Maintaining mesh connectivity using a simplex-based data structure

Han-Wen Nienhuys* and A. Frank van der Stappen
Institute of Information and Computing Sciences, Utrecht University,
PO Box 80089, 3508 TB Utrecht, The Netherlands
{hanwen,frankst}@cs.uu.nl

Abstract

Many applications in visualization and computer graphics use meshes composed of triangles or tetrahedra, and need to store the connectivity of these meshes. In some applications meshes must be changed at run-time. In this case, it is necessary to update mesh connectivity too. This paper presents a data structure with two operations that can express any mesh change. The prime advantage of this approach is that all low-level code to maintain mesh connectivity is contained in these two routines, which promotes a modular program design. 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. It applies to simplicial meshes of any dimension.

1 Introduction

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 [4], and the quad-edge structure [6]. These structures all store the connectivity in slightly different ways at slightly different memory costs. Memory usage is important when manipulating large meshes, so Campagna et al. [2] 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 [5] 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 [8] uses a simplified version of the facet-edge structure for maintaining the connectivity of tetrahedral meshes.

Brisson [1] 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: $\mathsf{switch}_k(t)$ 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

*corresponding author

discussed above, the edge algebra discussed by Guibas and Stolfi [6] and various other mesh data structures [7, 10] 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 paper we first discuss the underlying abstract mesh representation. This representation uses abstract oriented simplexes, a basic concept in algebraic topology [3]. 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.

2 Abstract oriented simplexes

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 $\operatorname{conv}\{\mathbf{a}_1, \mathbf{a}_2\}$, the convex hull of $\mathbf{a}_1$ and $\mathbf{a}_2$, is an edge, $\operatorname{conv}\{\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\}$ is a triangle, and $\operatorname{conv}\{\mathbf{a}_1, \ldots, \mathbf{a}_4\}$ 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 $\operatorname{conv}\{\mathbf{a}_1, \ldots, \mathbf{a}_k\}$ and $\operatorname{conv}\{\mathbf{b}_1, \ldots, \mathbf{b}_l\}$ are properly joined if

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

where $S \subset \{\mathbf{a}_1, \ldots, \mathbf{a}_k, \mathbf{b}_1, \ldots, \mathbf{b}_l\}$. Properly joined simplexes are demonstrated in Figure 1.

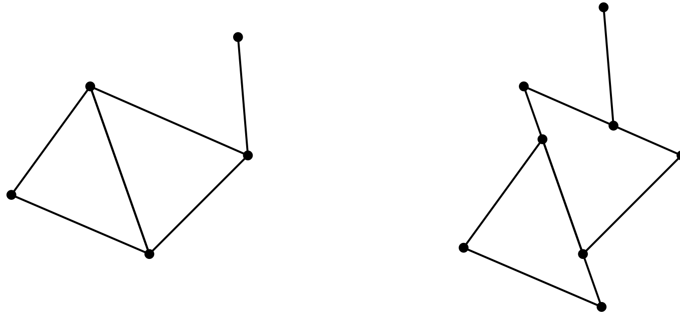

Figure 1: Properly joined simplexes (left), and improperly joined ones (right)

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

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.

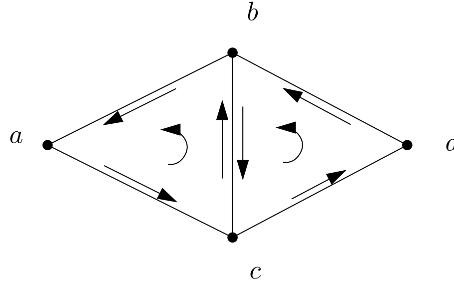

Figure 2: A simple 2-dimensional pseudo-manifold. We have $T = \{abc, bdc\}$ (leaving out the angled brackets in the notation of simplexes), and $K = T \cup \{ab, bc, ca, cb, bd, dc, a, b, c, d, -a, -b, -c, -d\}$. The boundary of K is formed by $\{ab, ca, bd, dc\}$; the other edges (bc and $-bc$) form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

3 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 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.

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. The following lists the most important operations that can be defined and implemented for the Simplex type.

- σ.subn (j) returns $(-1)^j q\langle a_0, \ldots, a_k \backslash a_j \rangle$, the jth subset of σ.
- 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.

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];
};
```

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

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

$$t.\mathsf{faces}[i].\mathsf{simplex} = t.\mathsf{simplex}.\mathsf{subn}(i), \qquad i = 0, \ldots, d. \tag{1}$$

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

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

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

$$f \in f.\mathsf{element}.\mathsf{faces}, \tag{2}$$

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

$$f.\mathsf{mate} = \mathsf{null} \vee f.\mathsf{mate}.\mathsf{simplex} = -f.\mathsf{simplex}. \tag{3}$$

The connectivity of a simplicial mesh then is a collection of Element and Face objects such that invariants (1) to (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

$$t.\mathsf{simplex} = u.\mathsf{simplex} \implies t = u. \tag{4}$$

4 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

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

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

$$\mathsf{boundary}(T) = \{(\sigma, f) : f.\mathsf{simplex} = \sigma, \wedge -\sigma \notin \mathsf{faces}(T) \wedge \sigma \in \mathsf{faces}(T)\} \tag{5}$$

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 ← new Element(τ)
   mesh.elements ←mesh.elements ∪ {e}
   for j in 0, ..., d:
      σ ← τ.subn(j)
      f ← new Face(σ)
      f.element ←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
```

By using the complex boundary and mate fields, the mesh connectivity can be maintained analogously during element removal.

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
```

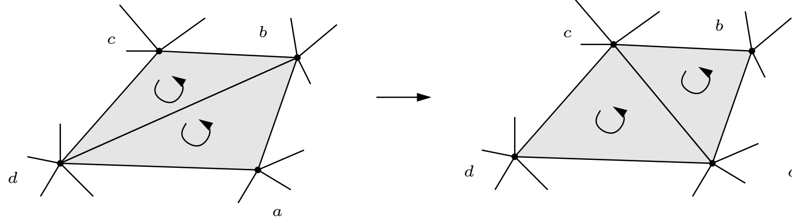

Figure 3: An edge flip changes the edge db to ad. This can be encoded as replacing the 2-simplexes $\{bcd, abd\}$ by $\{abc, acd\}$. The elements on the left contain edges ab, bc, cd, and da which are also present after the flip.

This code is still not optimal. For example, in the edge flip from Figure 3, 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 4, 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 $\pi : \mathcal{V} \rightarrow \mathcal{V}$ 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

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

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. In practice, a vertex substitution would be effected by the substituted method of the Simplex class.

```
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
```

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 a program, 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

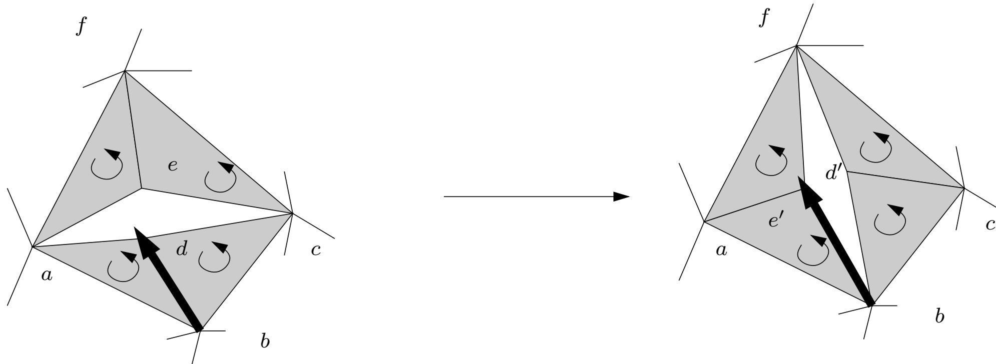

Figure 4: Cutting and stitching can be achieved using replace-triangles. The operation shown here can be effected as replacing $\{abd, bcd, cfe, aef\}$ with $\{abe', ae'f, d'cf, bcd'\}$. The operation can also be written as a vertex substitution, e.g. substitute $d := d'$ in abd. By specifying the operation like this, objects can be made persistent. Then bd (left) and be' (right) are represented by the same object, marked in bold.

replace-elements are guaranteed to deliver valid data structures, unless extra conditions are given on their arguments. Checking these conditions requires storing more information of the mesh, making change operations more expensive. Fortunately, Conditions (1) to (5) can be checked automatically in a validation routine. Such a validation routine is expensive, but could be called when a special debugging option is switched on. Less expensive checks include verifying that no key occurs twice when forming newfaces in the replace-elements algorithm. Such checks may also be done for production versions of a program.

The algorithms presented have been implemented in various prototypes of surgery simulation applications [9]. Source code for these programs is available from

`http://www.cs.uu.nl/groups/AA/virtual/`

The data structure favors reliability and ease of use over performance. The algorithms presented also 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.

Only d- and $(d-1)$-dimensional mesh features are identified with objects. If lower-dimensional simplexes must be tracked too, lists of these must be maintained separately.

References

[1] Erik Brisson. Representing geometric structures in d dimensions: Topology and order. *Discrete and Computational Geometry*, 9:387–426, 1993.

[2] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges — A scalable representation for triangle meshes. *Journal of Graphics Tools*, 3(4):1–12, 1998.

[3] Fred H. Croom. *Basic Concepts of Algebraic Topology*. Undergraduate Texts in Mathematics. Springer-Verlag, 1978.

[4] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*, chapter 9. Springer-Verlag, 2000.

[5] David P. Dobkin and Michael J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4:3–32, 1989.

[6] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[7] Martti Mäntylä. *An introduction to solid modeling.* Computer Science Press, College Park, MD, 1988.

[8] Ernst Peter Mücke. *Shapes and Implementations in Three-Dimensional Geometry.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1993.

[9] Han-Wen Nienhuys. *Cutting in deformable objects.* PhD thesis, Utrecht University, 2003.

[10] K. J. Weiler. *Topological Structures for Geometrical Modeling.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1986.