3.1 Triangle-Edge Data Structure

The data structure presented in this section is designed to represent the topology and order of a three-dimensional triangulation. The oldest and most widely used representation for the topological structure of a cell complex is the incidence graph (see for example [21]). This graph contains a node for every cell (of any dimension) in the complex, with arcs connecting nodes if the corresponding cells are incident to each other and their dimensions differ by one. This approach stores the cells *explicitly* but does not provide any direct access to ordering information. A fairly different approach is to represent the cells *implicitly* by a set of basic elements and operators. This idea was introduced with the quad-edge data structure by Guibas and Stolfi [42], proposed for modeling two-dimensional manifolds and their duals. Faces and vertices are represented implicitly by rings of directed and oriented edges. A set of traversal functions defines an abstract edge algebra. The edge-facet structure due to Dobkin and Laszlo [17] extends this idea to three-dimensions. Brisson [5] generalized this idea to subdivided manifolds of arbitrary dimensions.

The *triangle-edge data structure* presented here is a simplification of Dobkin and Laszlo's edge-facet structure. By reducing the scope to exactly what is needed, namely, simplicial complexes without duals, the compactness as well as the response time of the structure can be improved dramatically.

Data structure. The triangle-edge data structure is triangle based. Assume a unique index f for each triangle in a triangulation $\mathcal{T}$ of a set of points $S = \{p_1, p_2, \ldots, p_n\} I\!R^3$. The data structure's atomic unit is the so-called *triangle-edge* pair $a = \langle f, v \rangle$, with $0 \leq v \leq 5$, which identifies one of six versions of triangle $f = \sigma_{\{p_i, p_j, p_k\}}$. Each version corresponds to one of the six directed edges incident to f (see figure 3.1). Sometimes it will be convenient to associate edges or triangles with triangle-edge pairs. We write $a = \langle f, v \rangle \equiv e_{i,j} \equiv f_{i,j,k}$, where $e_a = e_{i,j}$ is the directed edge from p_i to p_j, and $f_{i,j,k}$ is the oriented triangle determined by the sequence of vertices (p_i, p_j, p_k). Let $f_{i,j,k}$ correspond to $a = \langle f, v \rangle$. The *positive halfspace* of a triangle-edge a, denoted by a^+, is the set of all points $x \in I\!R^3$ which see the endpoints of $f_{i,j,k}$ make a counterclockwise turn; that is, (x, p_i, p_j, p_k) has positive direction.

Each triangle f defines two *edge rings*, one traversing the edges of f in counterclockwise direction, and one in clockwise direction, as seen from $\langle f, 0 \rangle^+$. Similarly, each edge e defines two *triangle rings* traversing the triangles incident to e in the two opposite directions. Each triangle-edge a belongs to exactly one edge ring and exactly one triangle ring (see figures 3.1 and 3.2).

The following basic *triangle-edge functions* are defined:

- $a.\texttt{org} = i$ — if p_i is the origin of the directed edge e_a.
- $a.\texttt{sym} = b$ — if $e_a = e_{i,j}$ and $e_b = e_{j,i}$, that is, e_a and e_b are the two directed versions of the same undirected edge.
- $a.\texttt{enext} = b$ — if e_a and e_b are both in the same edge ring and e_b is the successor of e_a.
- $a.\texttt{fnext} = b$ — if e_a and e_b are in the same triangle ring and e_b is the successor of e_a.

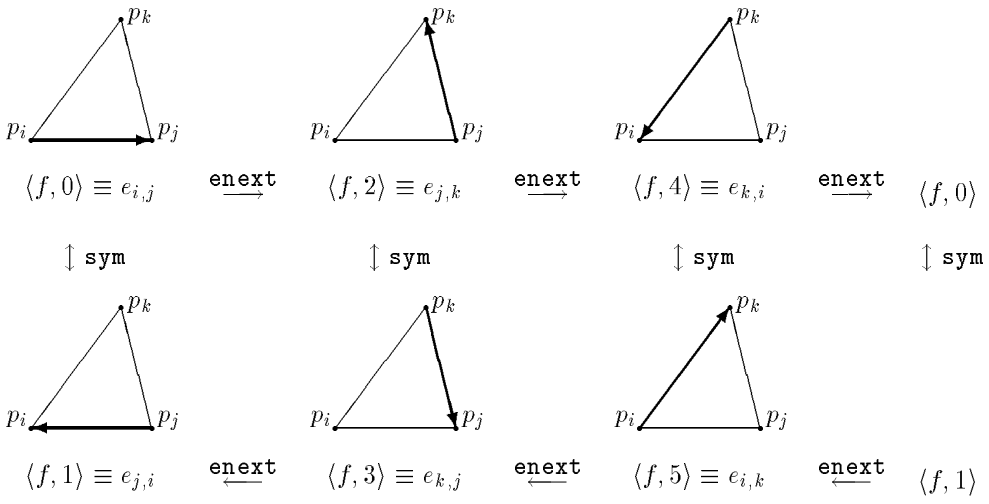

Figure 3.1: The six versions of triangle $f_{i,j,k}$, connected in two edge rings.

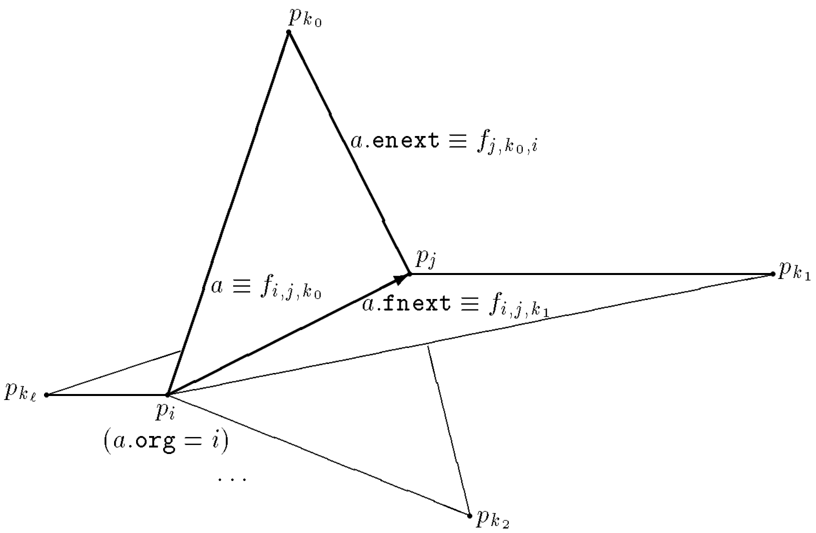

Figure 3.2: The basic triangle-edge functions.
The eyes of the reader are in half-space a^+. The triangle ring of a is $\langle\langle f_{i,j,k_0}, f_{i,j,k_1}, f_{i,j,k_2}, \ldots, f_{i,j,k_\ell}\rangle\rangle$. The triangle ring of $a.\texttt{sym} \equiv f_{j,i,k_0}$ is in opposite direction: $\langle\langle f_{j,i,k_0}, f_{j,i,k_\ell}, \ldots, f_{j,i,k_2}, f_{j,i,k_1}\rangle\rangle$.

Figure 3.2 illustrates these functions. The postfix form "a.func" and its equivalent functional form "func(a)" are used as synonyms throughout this text. Exponents are used to express functional concatenation; for example, "$a.\mathtt{enext}^2$" is equivalent with "a.enext.enext." Also, "a.dest" is used as an abbreviation for "a.sym.org."

Consistency. Table 3.1 lists some of the axioms that hold for the edge-triangle functions. These relations can be used to test the data structure for consistency. Of course, axioms involving enext traversals are trivially true since, as we will see below, our implementation has the edge rings "hard-wired." However, all axioms concerning fnext must hold if the facet rings are properly linked. These relations, together with, for example, Euler formulas and their derivatives, are useful to check whether the current state of the data structure is indeed a valid representation of a triangulation. For example, checking axioms (T13) and (T14) makes sure that the facets of all tetrahedra are properly linked.

Implementation. The internal representation of the triangle-edge functions takes advantage of the fact that the structure of all edge rings is identical. Therefore, it can be implemented by a global lookup table $ve[0..5]$ with six entries $ve[v]$, $0 \leq v \leq 5$, such that, $a.\mathtt{enext} = \langle t, ve[v] \rangle$, for any triangle-edge pair $a = \langle t, v \rangle$. However, actual pointers are needed for the successor relations in the

(T0)	$a \in \mathcal{T} \iff a.\mathtt{org} \in S$ and $a.\mathtt{sym}, a.\mathtt{enext}, a.\mathtt{fnext} \in \mathcal{T}$
(T1)	$a.\mathtt{sym}^2 = a$
(T2)	$a.\mathtt{enext}^3 = a$
(T3)	$a.\mathtt{enext}^i \neq a$ for $i = 1, 2$
(T4)	$a.\mathtt{sym}.\mathtt{enext}^i \neq a$ for $i = 1, 2, 3$
(T5)	$a.\mathtt{sym}.\mathtt{enext}.\mathtt{sym}.\mathtt{enext} = a$
(T6)	$a.\mathtt{fnext}^i = a$ for some $i < \infty$
(T7)	$a.\mathtt{sym}.\mathtt{fnext}^i \neq a$ for all i
(T8)	$a.\mathtt{sym}.\mathtt{fnext}.\mathtt{sym}.\mathtt{fnext} = a$
(T9)	$a.\mathtt{org} \neq a.\mathtt{sym}.\mathtt{org}$
(T10)	$a.\mathtt{enext}.\mathtt{org} = a.\mathtt{sym}.\mathtt{org}$
(T11)	$a.\mathtt{enext}^2.\mathtt{org} \neq a.\mathtt{org}$
(T12)	$a.\mathtt{fnext}.\mathtt{org} = a.\mathtt{org}$
(T13)	$a^+ \cap \lvert\mathcal{T}\rvert \neq \emptyset \Longrightarrow a = a.\mathtt{turn}^3$, with $a.\mathtt{turn} = a.\mathtt{enext}.\mathtt{sym}.\mathtt{fnext}.\mathtt{enext}$
(T14)	$a^+ \cap \lvert\mathcal{T}\rvert \neq \emptyset \Longrightarrow p_{a.\mathtt{fnext}.\mathtt{enext}^2.\mathtt{org}} \in a^+$
(T15)	$x \in a^+ \Longrightarrow$ sequence $(x, p_{a.\mathtt{org}}, p_{a.\mathtt{enext}.\mathtt{org}}, p_{a.\mathtt{enext}^2.\mathtt{org}})$ has positive orientation

Table 3.1: Some axioms for the triangle-edge functions.

triangle rings. Note that there are six versions of a triangle but only three vertices (that is, possible origins). To save memory, a second lookup table $vo[0..5]$ is used to map the indices corresponding to the origins of the six different versions of a triangle f to its base triangle $\langle f, 0\rangle$. See table 3.2 for the internal declaration of the data structure, and the pseudocode of the basic triangle-edge functions.

In total, there are nine fields per record for a triangle f: three `origin` fields for the indices, i, j, and k, where $\langle f, 0\rangle \equiv f_{i,j,k}$, and six `next_f` fields for the pointers to the next triangle in the triangle rings of $\langle f, v\rangle$, for $0 \leq v \leq 5$. For convenience, assume a (dynamically allocated) linear array $tr[0..*]$ to

```
type Triangle = record
  origin: array [0..2] of Integer;
  next_f: array [0..5] of Integer;
end;

const
  vo: array [0..5] of Integer ← {0, 1, 1, 2, 2, 0};
  ve: array [0..5] of Integer ← {2, 5, 4, 1, 0, 3};

var tr: dynamic array [0..*] of Triangle;

function org(a) is
  assert a = ⟨f, v⟩;
  return tr[f].origin[vo[v]];
end;

function sym(a) is
 return a + (if a is odd then −1 else 1);
end;

function enext(a) is
  assert a = ⟨f, v⟩;
  return ⟨f, ve[v]⟩;
end;

function fnext(a) is
  assert a = ⟨f, v⟩;
  return tr[f].next_f[v];
end;
```

Table 3.2: The basic triangle-edge functions.

store all the triangle records. Moreover, we map triangle-edge pairs to single integers, which allows us to handle all references to triangle pairs, and not just triangles, as integers: $\langle f, v\rangle \equiv 8f + v$. Note how this conversion formula enables the usage of fast shift operations instead of multiplication. This would not be possible for the otherwise more natural conversion formula $6f + v$.

Since lemma 2.4.1 states that triangulations of S have $O(n^2)$ triangles in the worst case, it makes sense to assign only half as many bits for each `origin` field than for each `next_f` field.[1] This gives a total of 30 bytes for a `Triangle` record, assuming the usual 32 bits integers for edge-triangle pairs. Using 32-bit `origin` fields increases the record size to 36 bytes. This lifts the restriction on the maximum number of vertices for all practical purposes, provided that the complexity of the triangulation does not show quadratic worst-case behavior.

The sign bits of the triangle-edge fields can be used to encode markers. For the time being, assume only the sign bits of the `origin` fields to be used for that purpose. This gives the user the possibility to mark each triangle with up to eight different "colors," where three of them are predefined: `BLANK` for unmarked triangles, and `HULL` or `SYMHULL` for triangles f bounding the convex hull of S, depending on whether $\langle f, 0\rangle^+ \cap \mathrm{conv}(S)$ or $\langle f, 1\rangle^+ \cap \mathrm{conv}(S)$ is $\emptyset$.

Creating and deleting triangles is easy when the records of the linear array $tr[0..*]$ are managed in a freelist. To delete a triangle, mark its record as "not active" by simply clearing its `origin` fields, and add it to the freelist. To create a new triangle $f_{i,j,k}$, retrieve an unused index f from the list, set its `origin` fields to i, j, and k, and initialize all its triangle rings to be *trivial*, that is, set the `next_f` fields such that $\langle f, v\rangle.\texttt{fnext} = \langle f, v\rangle$, for $0 \leq v \leq 5$.

Ring manipulation. A most useful operation is `fsplice`, which modifies triangle rings. It takes two triangle-edge pairs a and b as arguments and returns no value. However, assuming that $\langle\!\langle a, a.\texttt{fnext}, a.\texttt{fnext}^2, \ldots, a.\texttt{fnext}^{\ell-1}\rangle\!\rangle$ denotes the triangle ring of a, with $a.\texttt{fnext}^{\ell} = a$, the call of operation $\texttt{fsplice}(a, b)$ has the following side effects on the rings of a and b:

$$\langle\!\langle \underbrace{\ldots\ldots}_{i}, a, \overbrace{\ldots\ldots\ldots}^{ii}, b, \underbrace{\ldots\ldots}_{iii} \rangle\!\rangle \quad \overset{\texttt{fsplice}(a,b)}{\Longleftrightarrow} \quad \langle\!\langle a, \underbrace{\ldots\ldots}_{iii}, \underbrace{\ldots\ldots}_{i} \rangle\!\rangle \not\equiv \langle\!\langle b, \overbrace{\ldots\ldots\ldots}^{ii} \rangle\!\rangle$$

(a) If the two rings are distinct, it combines them into one ring.

(b) If the rings are identical, it breaks them into two distinct rings.

[1] Let M be the maximum value for `Integer` numbers. To be save also in the worst case, n has to be restricted so that $8(n^2 - 3n) + 5 \leq M$. For $M = 2^{31} - 1$ (integers), this gives $n \leq 16385$. For $M = 2^{32} - 1$ (unsigned integers), this is $n \leq 23171$. Both bounds are smaller than $2^{15} - 1$.

It is worthwhile to mention, that `fsplice` is the inverse to itself. The implementation of `fsplice` can be found in table 3.3. Also, observe how `fmerge`(a, b) = `fsplice`$(a, b$.sym.fnext.sym$)$ combines

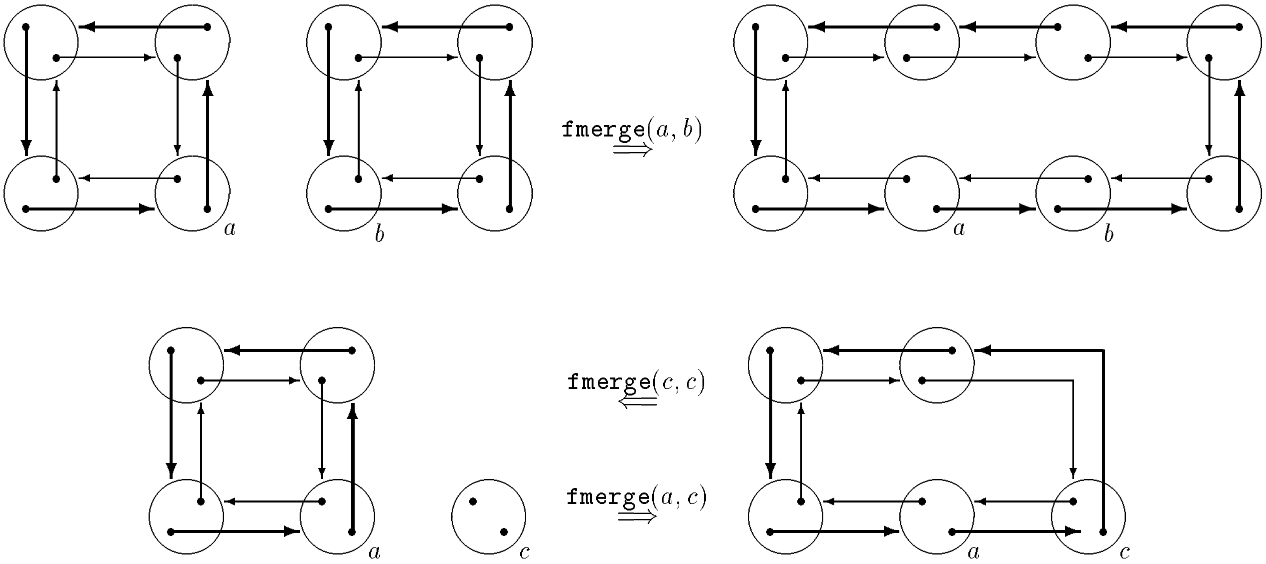

Figure 3.3: The effects of `fmerge`(a, b) = `fsplice`$(a, b$.sym.fnext.sym$)$.

procedure `fsplice`(a, b) **is**
  swap pointers a, b;
  swap pointers a.`fnext.sym`, b.`fnext.sym`;
end;

refinement swap pointers a, b **is**
  assert $a = \langle f_a, v_a \rangle$ **and** $b = \langle f_b, v_b \rangle$;
  $tr[f_a]$.`next_f`$[v_a] \;\leftrightarrow\; tr[f_b]$.`next_f`$[v_b]$;
end;

procedure `delete`(a) **is**
  assert $a = \langle f, v \rangle$ **and** $b = a$.`enext` **and** $c = a$.`enext`^{2};
  `fmerge`(a, a);
  `fmerge`(b, b);
  `fmerge`(c, c);
  for $v = 0, 1, 2$ **do**
    $tr[f].origin[v] \leftarrow 0$
  end for;
  add t to freelist;
end;

Table 3.3: More pseudocode for triangle-edge functions.

the two rings of a and b if they are different (see figure 3.3). When `fmerge` is applied to two identical arguments, like in `fmerge`(a, a), where a denotes some triangle-edge in a nontrivial triangle ring, it unlinks a from this ring. This fact is used in the code of `delete`, which can be found in table 3.3. On the other hand, if $a = \langle f, v \rangle$ denotes a trivial triangle ring, and b is some other ring, then `fmerge`(a, b) adds the triangle f to the ring of b.

procedure `collect`$(p_h, a, a', \mathcal{L})$ **is**
  assert $a = \langle f, v \rangle \equiv f_{i,j,k}$ **and** (p_h, p_i, p_j, p_k) has positive orientation;
  if f is marked with `HULL` or `SYMHULL` **then**
    unmark f;
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{a\}$;
    `collect_dfs`$(p_h, a$`.fnext.sym`$, a', \mathcal{L})$;
    `collect_dfs`$(p_h, a$`.enext.fnext.sym`$, a', \mathcal{L})$;
    `collect_dfs`$(p_h, a$`.enext`^{2}`.fnext.sym`$, a', \mathcal{L})$;
  else
    $\mathcal{L} \leftarrow \{a, a$`.fnext.sym`$, a$`.enext.fnext.sym`$, a$`.enext`^{2}`.fnext.sym`$\}$;
  end if;
end

procedure `collect_dfs`$(p_h, a, a', \mathcal{L})$ **is**
  assert $a = \langle f, v \rangle \equiv f_{i,j,k}$;
  if f is marked with `HULL` or `SYMHULL` **then**
    if (p_h, p_i, p_j, p_k) has positive orientation **then**
      unmark f;
      $\mathcal{L} \leftarrow \mathcal{L} \cup \{a\}$;
      `collect_dfs`$(p_h, a$`.enext.fnext.sym`$, a', \mathcal{L})$;
      `collect_dfs`$(p_h, a$`.enext`^{2}`.fnext.sym`$, a', \mathcal{L})$;
    else
      $a' \leftarrow a$;
    end if;
  end if;
end

Table 3.4: Collecting triangles visible from point p_h.

Usage. This section closes with a couple of sample programs using the triangle-edge functions of the data structure. These procedures will be used for the construction of Delaunay triangulations in section 4.1. The idea is to add a point p_h, which is not yet a vertex of the current triangulation $\mathcal{T}$, to the triangulation by either attaching it to the outside of the convex hull or by inserting it inside the appropriate tetrahedron. We assume that the current data structure is a valid representation of $\mathcal{T}$, and that all triangles bounding (the convex hull of) $\mathcal{T}$ are properly marked with either `HULL` or `SYMHULL`; call such a triangle a *hull triangle*. Furthermore, a facet $a = \sigma_T$ is said to be *visible* from p_i, if $\sigma_{T \cup \{p_i\}}$ does not intersect any triangle of $\mathcal{T}$, and $p_i \in a^+$, the positive side of a.

Procedure `collect`$(p_h, a, a', \mathcal{L})$ is to collect all triangles that are visible from p_h, and to store them in a set $\mathcal{L}$. Table 3.4 lists the code. We assume that an oracle provided an initial triangle-edge a which corresponds to one such triangle. In other words, $a \equiv f_{i,j,k}$, the sequence (p_h, p_i, p_j, p_k) has positive orientation, and one of the following two cases applies.

1. $p_h \notin |\mathcal{T}|$.
   Then a lies on the boundary of the convex hull, and `collect` will use depth-first search to add all visible hull triangles to $\mathcal{L}$. As a side effect, their hull bits will be deleted. This will tear a hole into the boundary of $|\mathcal{T}|$, which has to be fixed afterwards. For that purpose, we maintain a hull triangle a' which is not visible from p_h, but lies on the boundary to the region covered by the triangles of $\mathcal{L}$ (see below).
2. $p_h \in |\mathcal{T}|$.
   In this case, p_h lies inside the tetrahedron $\sigma_T = t_{a^+}$, that is $\{p_i, p_j, p_k\} \subset T$ and $\sigma_T \cap a^+$ is nonempty. In this case, the output parameter a' will be undefined, and $\mathcal{L}$ will simply consist of the triangles of σ_T.

In order to insert the new vertex p_h into the current triangulation, `collect`$(p_h, a, a', \mathcal{L})$ is used in conjunction with a procedure `connect`$(p_h, a', \mathcal{L})$ which adds the tetrahedron $t_{h,i,j,k}$ to $\mathcal{T}$for every triangle $f_{i,j,k} \in \mathcal{L}$. The full pseudocode of this procedure is omitted; however, some crucial details are provided in the following.

For the time being, assume that $p_h \notin |\mathcal{T}|$. Furthermore, let the triangles of $\mathcal{L}$ be processed in the same order as they were gathered by `collect` (that is, in depth-first order). With this, $f_{i,j,h} \notin \mathcal{T}$ conveniently implies that $f_{j,k,h} \notin \mathcal{T}$ and $f_{k,i,h} \notin \mathcal{T}$. Consulting figure 3.4, five case can be distinguished:

A. $f_{i,j,h} \notin \mathcal{T}$ (which implies $f_{j,k,h} \notin \mathcal{T}$ and $f_{k,i,h} \notin \mathcal{T}$).
All three missing triangles have to be added to $\mathcal{T}$. Also, we need to perform appropriate modifications to the corresponding triangle rings.

B. $f_{i,j,h} \in \mathcal{T}$ and $f_{j,k,h} \notin \mathcal{T}$ and $f_{k,i,h} \notin \mathcal{T}$.
Two missing triangles have to be added, and their triangle rings modified.

C. $f_{i,j,h} \in \mathcal{T}$ and $f_{j,k,h} \in \mathcal{T}$ and $f_{k,i,h} \notin \mathcal{T}$.
One new triangle has to be created, with the appropriated triangle rings.

D. $f_{i,j,h} \in \mathcal{T}$ and $f_{j,k,h} \notin \mathcal{T}$ and $f_{k,i,h} \in \mathcal{T}$.
This case is the mirror image of case C.

E. $f_{i,j,h} \in \mathcal{T}$ and $f_{j,k,h} \in \mathcal{T}$ and $f_{k,i,h} \in \mathcal{T}$.
In this case, all adjacent triangles have already been created at an earlier stage. However, it is possible that the triangle rings are not yet properly closed. In some sense, only *after* this is done, the tetrahedron $t_{h,i,j,k}$ is really added to $\mathcal{T}$.

Obviously, it is not enough to just create the new triangles, they also have to be linked into the data structure by appropriate modifications to the corresponding triangle rings. Since the details of this procedure are tedious and repetitive, only case B will be considered as an example. Table 3.5 lists the appropriate code segment. Two triangles, $f_{j,k,h}$ and $f_{k,i,h}$, have to be created, and five triangle rings, corresponding the edges $e_{j,k}$, $e_{k,i}$, $e_{i,h}$, $e_{j,h}$, and $e_{k,h}$, have to be altered. Note that the triangle rings of the edges of the new triangles are initially trivial. Thus, x.`sym.fnext.sym` $= x$, and calls to `fmerge` in table 3.5 can be replaced by calls to `fsplice` with the same parameters.

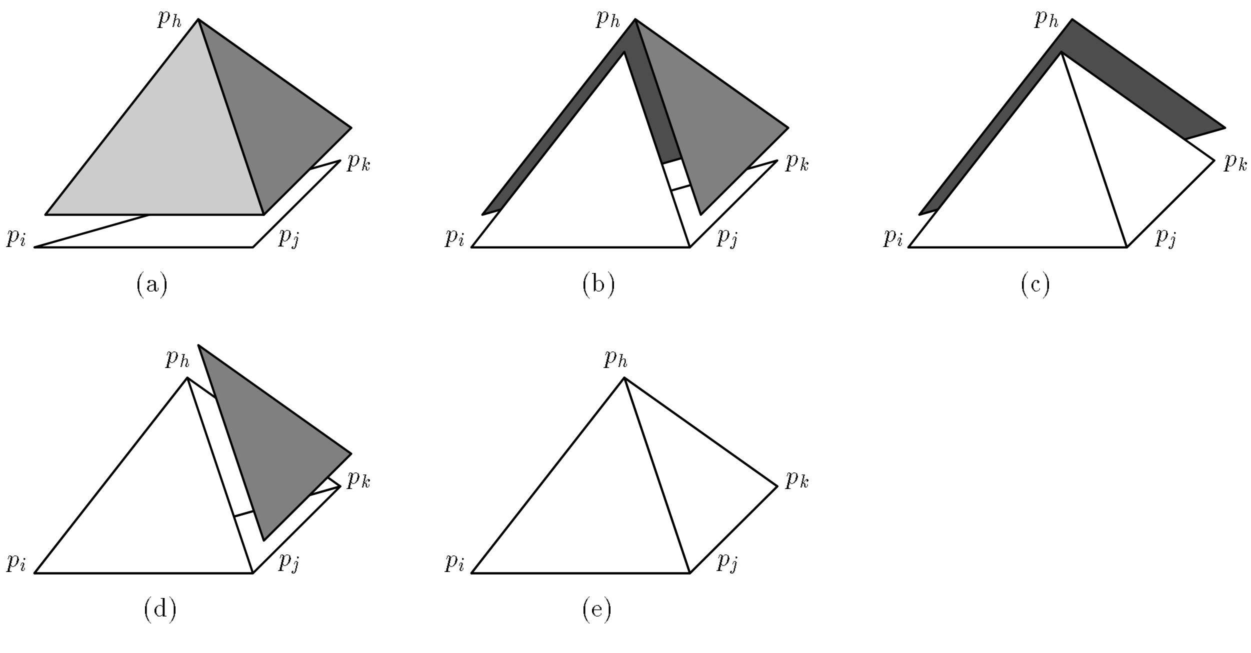

Figure 3.4: The five cases when connecting p_h to $a \equiv f_{i,j,k}$.

To reiterate, `connect`$(p_h, a', \mathcal{L})$ scans through $\mathcal{L}$, the list of triangles visible from p_h, and processes them in the same order as they were generated by `collect`$(p_h, a, a', \mathcal{L})$, starting with some visible hull triangle a. For every visited triangle, it then distinguishes the five cases of figure 3.4 and executes a series of `fmerge` or `fsplice` operations, following the pattern of table 3.5. After this, the facets of every tetrahedron $t_{h,i,j,k}$, for $f_{i,j,k} \in \mathcal{L}$, are properly added to the triangulation $\mathcal{T}$. Since we were adding the point p_h outside $|\mathcal{T}|$ we still have to fix the hull bits of the new triangles on the convex hull. A simple depth-first traversal starting at the hull triangle a', which was provided by `collect` as not visible from p_h, but initially adjacent to some triangle in $\mathcal{L}$, can do the job.

Finally, observe that the above procedure works independently of whether $p_h \notin |\mathcal{T}|$ or $p_h \in |\mathcal{T}|$. In the latter case, p_h lies inside some tetrahedron, and $\mathcal{L}$, as generated by procedure `connect`, consist of its four triangles. They are processed by `connect`, which adds the new triangles after encountering the cases A, B, C, and E. Overall, the time complexity of `collect` and `connect` is proportional to the number of triangles visible from p_h.

This concludes the discussion of the edge-triangle data structure. This structure and its traversal functions will be used in section 4.1, where we construct three-dimensional Delaunay triangulations.

`connect` **refinement** case B(p_h, a) **is**
  assert $a \equiv f_{i,j,k} \in \mathcal{L}$ **and** $f_{i,j,h} \in \mathcal{T}$ **and** $f_{j,k,h} \notin \mathcal{T}$ **and** $f_{k,i,h} \notin \mathcal{T}$;
  $b \leftarrow f_{j,k,h}$;
  $c \leftarrow f_{k,i,h}$;
  `fmerge`$(a.$`enext`$, b)$; $\{ e_{j,k} \}$
  `fmerge`$(a.$`enext`$^2, c)$; $\{ e_{k,i} \}$
  `fmerge`$(a.$`fnext.sym.enext`$, c.$`enext`$)$; $\{ e_{i,h} \}$
  `fmerge`$(a.$`fnext.enext.sym`$, b.$`enext`$^2)$; $\{ e_{j,h} \}$
  `fmerge`$(b.$`enext`$, c.$`enext`$^2.$`sym`$)$; $\{ e_{k,h} \}$
end

Table 3.5: Connecting p_h to $a \equiv f_{i,j,k}$; case B.

3.2 Alpha Shape File and Data Structure

According to chapter 2, it is possible to represent the whole family of α-shapes of S by the faces or simplices of the Delaunay triangulation $\mathcal{D}$, augmented by a set of intervals. How to store these intervals is the topic of this section. Figure 3.5 shows the overall layout of the data structure, which is discussed in more detail below.

Since it is often convenient to have the interior of a shape triangulated, we represent α-complexes, rather than α-shapes; see section 2.2. As already discussed, there is a single interval for each simplex $\sigma_T \in \mathcal{D}$ so that σ_T belongs to the α-complex $\mathcal{C}_\alpha$ if and only if α is contained in this interval. Each face is classified as either interior, regular, or singular, which breaks down the interval for each face into three (possibly empty) subintervals; see table 2.2, section 2.3.

Face indexing. The data structure of section 3.1, which is used to store $\mathcal{D}$, is triangle based; vertices, edges, and tetrahedra are only represented implicitly. However, some sort of indexing for faces of all dimensions is needed in order to conveniently store additional information for each face (that is, the above mentioned intervals) in linear arrays. For vertices and triangles this is easy. SoS already provides a unique index for each point or vertex. Triangles can be indexed as in the triangle-edge data structure.

Hash tables are used to index edges and tetrahedra. For each edge and tetrahedron, the hash function provides a unique index between 1 and $m_e > n_e = |F_1|$ and between 1 and $m_t > n_t = |F_3|$, respectively.[2] Note that there will be unused indices, corresponding to empty records in the hash tables. The number of unused indices depends on the given load factor (for example, the load factor may be set to $\frac{n}{m} = 0.8$).

Let $E(\sigma_T)$ be the set of all edge-triangle pairs that correspond to a simplex $\sigma_T \in \mathcal{D}$. For a tetrahedron $\sigma_T \equiv t_{a^+}$ these are the corresponding four edge-triangle pairs; and for an edge $\sigma_T \equiv e_a$, we set $E(\sigma_T) = \{a.\mathtt{fnext}^i, a.\mathtt{sym.fnext}^i \mid i = 1, 2, 3, \ldots\}$; that is, the edge-triangle pairs of the facets in the two triangle rings (in opposite direction) corresponding to e_a. We define the *minimum edge-triangle pair* representing an edge $\sigma_T \equiv e_a$, or a tetrahedron $\sigma_T \equiv t_{a^+}$, as $a_{\min} = \min\{\langle f, v\rangle \mid \langle f, v\rangle \in E(\sigma_T)\}$.[3]

[2] The actual ranges of the hash indices for edges and tetrahedra are $0 \ldots m_e - 1$ and $0 \ldots m_t - 1$. For convenience, we add 1 to hash indices to get face indices starting at 1.

[3] It is easy to order the edge-triangle pairs. After all, we are already representing them as integers: $\langle f_1, v_1\rangle < \langle f_2, v_2\rangle$ if and only if $8f_1 + v_1 < 8f_2 + v_2$; compare this with section 3.1.

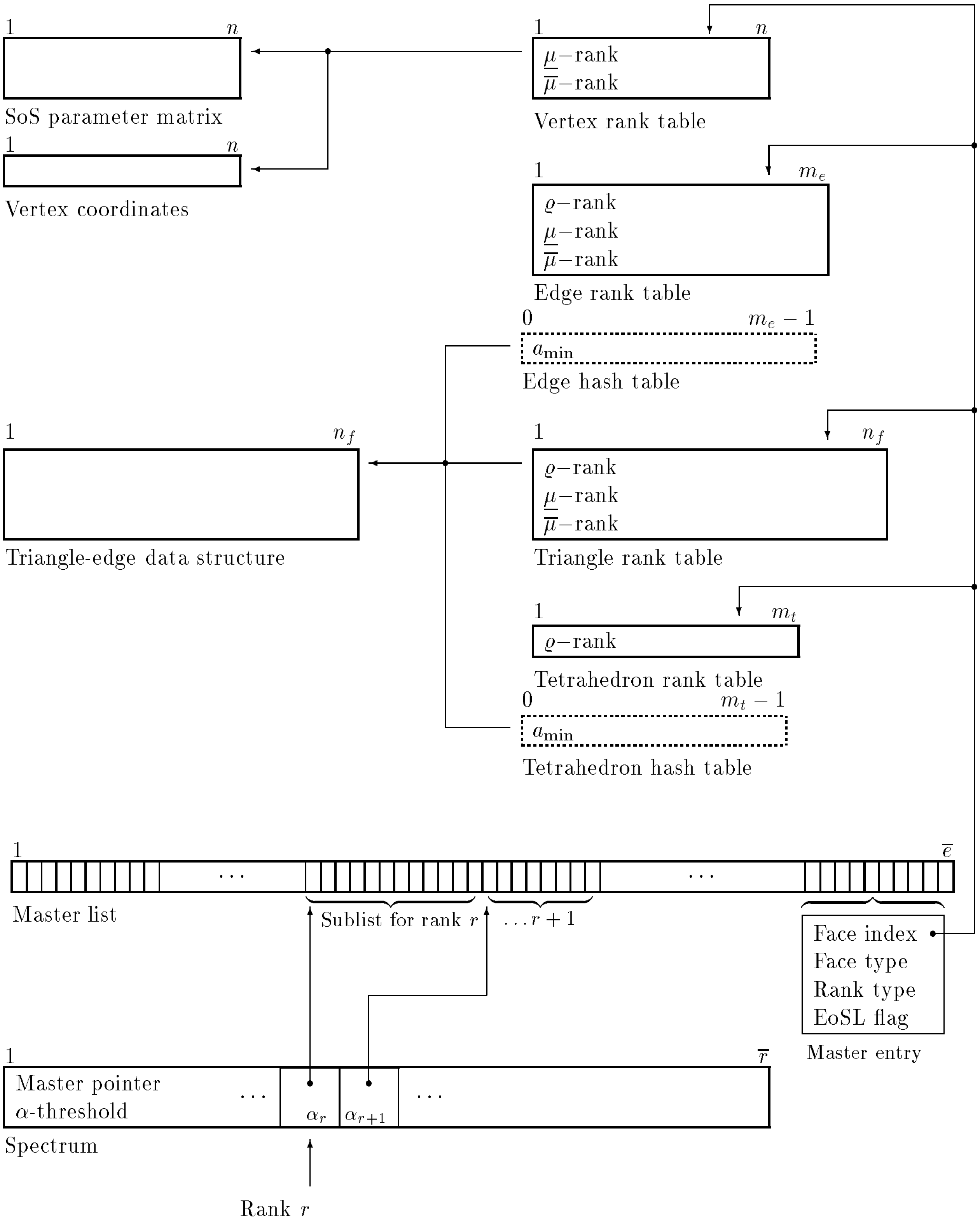

Figure 3.5: The α-shape file and data structure.

Note that $a_{\min}$ can be computed in constant time for each tetrahedron t_a^+, and in constant amortized time for each edge e_a in $\mathcal{D}$. Since $a_{\min}$ is unique for each face, it can be used as a key for the hash functions. Moreover, it also allows the implementation of inverse indexing: given a face index i, we can obtain an edge-triangle pair representing the face; either by the $a_{\min}$ reference in the $(i-1)$-st entry of the corresponding hash table, or, in the case of triangles, by directly referring to the i-th triangle record. Thus, we achieve a unique (and reversible) indexing scheme for all faces of $\mathcal{D}$. This will be used in the following to build the linear arrays storing the intervals for the face classification of section 2.3.

Alpha ranks. Recall that the set of α-thresholds consists of all end points of the intervals in table 2.2. The sorted sequence of α-thresholds, $\alpha_1, \alpha_2, \ldots, \alpha_{\overline{r}}$, with $\alpha_r < \alpha_{r+1}$ for $1 \leq r \leq \overline{r}$, forms the α-spectrum. For convenience, we always assume $\alpha_1 = 0$ and $\alpha_{\overline{r}} = \infty$; if 0 and ∞ do not occur in the set of α-thresholds, these values are added to the spectrum. We say the threshold α_r is of *rank* r. Each rank stands for an end point of a certain value. This is used to store the intervals of the faces of $\mathcal{D}$ in four so-called *rank tables*, one for each dimension; see figure 3.5. For edges and triangles we need to store the ranks of the ϱ, $\underline{\mu}$, and $\overline{\mu}$ values, for vertices the ranks of $\underline{\mu}$ and $\overline{\mu}$, and for tetrahedra only the rank of ϱ; compare this with the intervals in table 2.2.

Note that certain intervals are empty. A "void" rank 0 is used to encode these; again, compare this with table 2.2.

- Attached edges and triangles have ϱ-rank $= 0$.
- Faces bounding conv(S) have $\underline{\mu}$-rank $\neq 0$ and $\overline{\mu}$-rank $= 0$.
- Furthermore, redundant (or dumped) vertices and empty entries in the edge and tetrahedra rank tables are marked with $\underline{\mu}$-rank $= \overline{\mu}$-rank $= 0$.[4]

Master list. In general, end points of more than one α-interval will be mapped to the same α-threshold in the spectrum. For many application, such as computing "Betti numbers" of α-complexes [15], or computing the volume, surface, and other metric features of α-diagrams or space filling diagrams [25], it is desirable to be able to retrieve, given a rank r, all faces whose intervals have α_r as an end point. In other words, for each α-threshold α_r in the spectrum, we need to store the list of faces whose intervals have this threshold value as an end point. For convenience, we merge these lists in order of increasing rank to a list of sublists, the so-called *master list*; see figure 3.5. Each entry stores the corresponding face index, the face type (that is, the dimension of the face), the rank type (the entry may have type ϱ, $\underline{\mu}$, or $\overline{\mu}$), and a flag indicating the end of a sublist. The faces in each sublist are sorted in order of non-decreasing dimension.

[4] The concept of "redundant" points or vertices occurs in the context of weighted α-shapes and regular triangulations; see section 4.1.1. Even in the case of Delaunay triangulations, our actual implementation chooses to dump duplicate points. Neither redundant nor dumped vertices occur in $\mathcal{D}$.

The master list can be implemented as a linear array, provided that the spectrum stores an index for each threshold α_r, pointing to the corresponding sublist within the array. Note that, if we scan the spectrum and master list in order of decreasing ranks, we get the faces in the same order as they are removed from the α-complex when "shrink wrapping" the α-shape from the convex hull conv(S), for $\alpha_{\overline{r}} = \infty$, to the point set S itself, for $\alpha_1 = 0$. Also notice that the sublists of $\alpha_1 = 0$ and $\alpha_{\overline{r}} = \infty$ are empty in the nondegenerate case. Furthermore, observe that we do not explicitly store the $\varrho = 0$ values of the vertices. Thus, for $\alpha = 0$, the set of vertices is initialized with S, by default.[5]

Section 4.3 will list the formulas needed to compute the α-thresholds, and section 5.2 will discuss their robust implementation using the SoS perturbation scheme. Here, we only mention that it is neither necessary nor desirable to maintain the perturbation for the values of the α-spectrum: simple floating-point values will do. In fact, SoS uses exact arithmetic, and the resulting α-thresholds are rationals formed as ratios of long-integer numbers. These exact threshold values are used to sort the spectrum consistently. After this, however, they are discarded, and mapped to floating-point values. This is done so that

- different actual values (that is, rationals) for α-thresholds with the same floating-point value are mapped to different ranks, while
- α-thresholds, which are only different with respect to the perturbation, are mapped to the same rank.

To conclude the discussion on the data structures used to represent the family of α-shapes for a point set $S \in \mathbb{R}^3$, it should be noted that interval trees (see for example [67]) can be used as an alternative to (or in addition to) the master list. In particular, rendering shapes of arbitrary α will be more efficient with interval trees. However, the master list is more compact, and thus better suited for storage purposes; besides, it is always possible to reconstruct the interval trees from the master list, if desired.

[5] This is no longer valid for weighted α-shapes. There, the ϱ-values of vertices are not necessarily all zero and their actual values have to be stored in the vertex rank table as well. The sublist for $\alpha_1 = 0$ is no longer necessarily empty in the non-degenerate case.