ESQ: Editable SQuad representation for triangle meshes

Luca Castelli Aleardi
LIX, Ecole Polytechnique
Palaiseau, France
amturing@lix.polytechnique.fr

Olivier Devillers
Projet Geometrica, INRIA
Sophia-Antipolis, France
olivier.devillers@inria.fr

Jarek Rossignac
Georgia Institute of Technology
USA
http://www.cc.gatech.edu/~jarek/

***Abstract*—We consider the problem of designing space efficient solutions for representing the connectivity information of manifold triangle meshes. Most mesh data structures are quite redundant, storing a large amount of information in order to efficiently support mesh traversal operators. Several compact data structures have been proposed to reduce storage cost while supporting constant-time mesh traversal. Some recent solutions are based on a global re-ordering approach, which allows to implicitly encode a map between vertices and faces. Unfortunately, these compact representations do not support efficient updates, because local connectivity changes (such as edge-contractions, edge-flips or vertex insertions) require re-ordering the entire mesh. Our main contribution is to propose a new way of designing compact data structures which can be dynamically maintained. In our solution, we push further the limits of the re-ordering approaches: the main novelty is to allow to re-order vertex data (such as vertex coordinates), and to exploit this vertex permutation to easily maintain the connectivity under local changes. We describe a new class of data structures, called *Editable SQuad (ESQ)*, offering the same navigational and storage performance as previous works, while supporting local editing in amortized constant time. As far as we know, our solution provides the most compact dynamic data structure for triangle meshes. We propose a linear-time and linear-space construction algorithm, and provide worst-case bounds for storage and time cost.**

***Keywords*-triangle meshes; compact representations; dynamic data structures;**

I. Introduction

Many graphics, animation and modeling applications use triangle meshes. Different data structures have been proposed for representing them [1], [2], [3], [4], [5]. In general one associates consecutive positive integer IDs with the different vertices, store the vertex data (locations, normals, textures) in an array, and use arrays to store incident relations between faces and vertices. In some cases the *traversability* (the ability of accessing neighboring cells at a constant time cost) is not required: for example, for applications such as rasterization and integral properties evaluation, the information directly available from the indexed face set suffices. However, many mesh processing functions (such as identifying silhouette edges, rendering with on-the-fly adaptive subdivision, or walking along the intersection with a plane or another mesh) require access to adjacent elements. Furthermore, some applications require a set of operators in order to locally update the mesh structure: this occurs, for example, in the incremental computation of Delaunay triangulation [6], [7]. Others require changing the connectivity dynamically (such as adaptive mesh refinement [8]). Usual mesh representations contain redundant information in order to achieve the prescribed requirements. Finally, the *compactness* has been considered more recently: the data structure should take as little storage as possible, so as to reduce page faults and cache misses. Although some applications store a large amount of information associated to mesh elements (such as vertex or faces), most applications only store vertex coordinates (often quantized to 16 bits or less) and possibly normal and texture coordinates (which can also be quantized). In general the cost of connectivity is really expensive and always dominates storage. For example, in the case of a triangle mesh, there are typically twice as many triangles as vertices: therefore the storage cost of connectivity in common data structures (such as *Corner Table* or *Half-edge*), ranges from $((6 \times 2) + 1) \times 32 = 416$ to $((3 \times 6) + 1) \times 32 = 608$ bits per vertex, while the storage of the quantized vertex locations is usually between $3 \times 16 = 48$ and $3 \times 32 = 96$ bits per vertex.

A. Traversable, modifiable and compact meshes

Two natural requirements for a mesh data structure are the accessability and navigability: *traversable meshes* should provide a basic set of operators allowing efficient (possibly constant-time) access and navigation between mesh elements. For example, the implementation of mesh traversals requires to perform the walk on the mesh around a face or to retrieve the neighbors of a vertex. Data structures should be also *indexable*. The application should be able to associate consecutive integer IDs to all triangles, and to access the IDs of their vertices in constant time, so as to support various marking and bookkeeping functionalities. The *reverse indexability* may also be important to applications where one wishes to obtain the IDs of all triangles that are incident upon a given vertex. *Modifiable* data structures have the ability to dynamically maintain the mesh under local updates. Most applications need to perform local connectivity changes (attach/remove a triangle, insert or split a vertex, collapse or flip an edge) in constant time. A further requirement is the *simplicity*: a data structure should be simple to implement, so as to facilitate the development and testing of operators for mesh traversal and modification. The operators for changing the connectivity

and for indexing or traversing the mesh should be fast in practice, so that their cost is negligible when compared to the geometric or photometric processing cost performed by typical applications. Because of the increasing complexity of surface meshes used in applications, a number of works propose *compact* data structures: the *compactness* of a mesh representation is a quality of measure, which can be defined as the average number of references stored per vertex. For dealing with meshes having huge size one has to eliminate, as much as possible, the redundancies of data structures: this is a crucial point, to reduce overall storage and hence memory thrashing.

B. Prior art: mesh encodings and data structures

Mesh compression: As already observed, representing a triangulation in a basic way induces a lot of redundancies. There exists several schemes allowing to efficiently compress the connectivity of a surface mesh into a few bits per vertex: this is the case, for example, of the *Edgebreaker* scheme [9], [10], which encodes a planar triangulation of size n with at most $3.67n$ bits. A recent scheme [11] is even more compact, allowing to compress into the optimal number of bits $(3.24n)$, matching asymptotically Tutte's entropy bound [12]. Although compressed formats reduce connectivity storage, they are not useful for mesh processing, since to support traversal operators they require full (or at least partial) decompression.

Classical data structures: Most popular geometric data structures store a large number of references in order to describe incidence relations. Their classical implementations, in most programming environments are *explicit* representations: references allow to navigate in the data structure through address indirection. For example, in edge-based representations such as *Half-edge* [2], for each basic element (the *half-edge*) one stores 2 references to incident half-edges (the next one in ccw order in the same face, and the opposite one, lying in the neighboring face), plus a reference to a bounding vertex. Moreover, each vertex stores an incident half-edge, which leads to $3e + n$ references for a general manifold mesh having e edges and n vertices. Thus, in the case of triangle meshes, the *Half-edge* data structure uses $19\,rpv$ (*references per vertex*): other edge-based representations, such as *Quad-edge* [5] or *Winged-edge* [3], have similar storage requirements. Face-based representations (*Corner Table*, or triangle DS [1]) have slightly smaller requirements. They store, for each triangle, only 3 references to the neighboring faces (or opposite corners) and 3 references to the bounding vertices, plus a reference to an incident face for each vertex: this leads to a storage cost of $(2 \times (3+3)) + 1 = 13\,rpv$. Classical data structures are modifiable as they support updates in constant time: implementing a local modification (such as a triangle split or an edge flip) in the mesh involves a constant number of memory access and reference updates.

Compact data structures: The main goal of compact (practical) data structures ([13], [14], [15], [16], [17], [18], [19], [20]) is to reduce the number of references, while still guaranteeing an efficient implementation of navigational operations as in usual representations. This goal can be achieved, for example, by grouping adjacent mesh elements (as neighboring triangles in [14]): this allows to save some internal references, while still maintaining the representation under local modifications. A more recent approach consists in re-ordering mesh elements (typically faces, or edges) according to a given permutation. Adopting this strategy, the *SOT data structures* [16] allows to efficiently navigate in the mesh, using at most $6\,rpv$. Combining this idea with a triangle pairing and a matching between triangles and vertices, it is possible to obtain a more compact version (*SQUAD* data structure [17]) which uses about $4\,rpv$ in practice (the *LR* data structure is even more compact, requiring about $2.16\,rpv$ for tested meshes). Performing a re-ordering of input points, a recent edge-based representation [15] (referred to as *sorted TRIPOD*) guarantees the same navigation performances, providing a space bound of $4\,rpv$ in the worst case (as in the *TRIPOD* data structure [20], the main ingredient are Schnyder woods [21]).

Theoretical solutions: succinct representations: For completeness, we also mention that *succinct representations* [22], [23], [24] are successful in matching optimal asymptotic bounds for many class of meshes, running under the well known *word-Ram model*. In this case a memory word can store an arbitrary (small) number of references (or service bits) of tiny size: typically, one may stores up to $O(\frac{\lg n}{\lg \lg n})$ sub-words of length $l = O(\lg \lg n)$ each. Succinct data structures are of theoretical interest, mainly because the amount of auxiliary bits required, even if asymptotically negligible, remains important for triangulations of practical size.

Modifiable compact solutions: Star vertices [25] uses $7\,rpv$ to the price of navigation in $O(d)$ time, where d is the degree of the vertex involved. This structure is dynamic, as the mesh is editable: update operations can be performed in $O(d^2)$ time, where d is the degree of involved vertices. The catalog-based representation described in [14] has a storage cost slightly larger (the most compact version requires $7.67\,rpv$), while being fully dynamic: standard local modifications can be performed in $O(1)$ time (as for [1], the number of reference updates is still constant). This catalog-based approach is somehow a practical version of [24], a succinct dynamic representation allowing to encode a triangulation having f faces with at most $2.17f + o(f)$ bits, while supporting local modifications in $O(\log^2 f)$ amortized time.

Our contributions

Most recent data structures for the connectivity of manifold triangle meshes are either compact, modifiable, or traversable, but not all three. Our contribution is to introduce new data structures achieving all the requirements above. The main idea is based on a new sorting strategy, making use of a matching from vertices to triangles: our novel approach for storing connectivity could lead others to invent new interesting dynamic mesh encoding schemes. We are able to describe a new class of mesh data structures supporting standard local edits in constant

	traversable/modifiable			compact and traversable				compact/travers/modif		**our results**		
Data structure	Edge Based	Triangle DS Corner Table	Directed Edges	SOT	SQUAD	LR	sorted TRIPOD	2D Catalogs	Star Vertices	ESQ C_1	ESQ C_2	ESQ C_3
size (rpv)	19	13	13	6	≈ 4	≈ 2.16	4	7.67	7	6	6	4.8
navigation	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(d)$	$O(1)$	$O(1)$	$O(1)$
vertex access	$O(1)$	$O(1)$	$O(1)$	$O(d)$	$O(d)$	$O(d)$	$O(d)$	$O(1)$	$O(1)$	$O(d)$	$O(d)$	$O(d)$
triangle split	$O(1)$	$O(1)$	$O(1)$	-	-	-	-	$O(1)$	$O(d^2)$	$O(1)$	$O(1)$	$O(1)$
edge flip	$O(1)$	$O(1)$	$O(1)$	-	-	-	-	$O(1)$	$O(d^2)$	$O(1)$	$O(1)$	$O(1)$
vertex delete	$O(1)$	$O(1)$	$O(1)$	-	-	-	-	$O(1)$	$O(d^2)$	$O(d)$	$O(1)$	$O(d)$

TABLE I: Comparison between data structures for triangle meshes. Memory requirements are expressed in terms of references per vertex (*rpv*) and hold in the worst case (at the exception of *SQUAD* and *LR*, whose performances come from experimental tests). Navigation and vertex access time hold in the worst case. The update cost is given in terms of the number of reference updates. We denote by d the degree of involved vertices.

(amortized) time. As in prior works ([15], [17], [16], [18]), local navigation operators are supported in worst case $O(1)$ time, while the access to vertex data can require $O(d)$ time (for a degree d vertex). Our data structures are simple to implement and provided with an analysis of worst case storage bounds. The simplest solution uses $6\,rpv$, while supporting updates operations in $O(1)$ amortized time: this is obtained with a re-ordering of input data which allows to encode the map from triangles to vertices. Our most compact data structure makes use of a grouping strategy between adjacent triangles, and uses only $4.8\,rpv$, while still supporting efficient navigation and update operations. We provide experimental results concerning storage requirements and time-cost performances, obtained with implementations, which confirm the practical interest of our approach (experimental comparisons are reported in Table II, while Table I shows the space requirements and worst-case performances for traversal and update operators of existing mesh representations).

II. Preliminaries

Meshes and triangulations

Here we consider manifold triangle meshes, whose combinatorics correspond to simple triangulations embedded on a surface of arbitrary genus (without loops and multiple edges). Given a triangulation $\mathcal{T}$, we denote by n (resp. by f) the number of its vertices (resp. faces). Let us denote by $V = \{v_0, v_1, \ldots, v_{n-1}\}$ the set of its vertices, and by $F = \{\triangle_0, \triangle_1, \ldots, \triangle_{f-1}\}$ the set of its triangles.

Dynamic data structures

As in previous works [26], [24], [27], [14] we assume that the programming environment provides a system memory manager which allows to allocate and free memory. The problem of designing and implementing efficient dynamic data structures is of both theoretical and practical interest. From the design and analysis point of view *resizable arrays* [26] provide worst case $O(1)$ time random access to data (reading and writing operations), while supporting updates (shrinking and growing operators) in amortized $O(1)$ time, where additional storage cost is only $O(\sqrt{n})$ (where n is the number of elements stored). Similar time-cost performances are also matched by practical implementations provided in classic programming languages (such as Java or C++), the wasted storage due to deallocation being of order $\Theta(n)$. In the rest of this work we assume to have an implementation of an *abstract data type* (ADT) for dynamic arrays, supporting the following operators with the performances above:

- $\texttt{read}(T, i)$: return the value stored at $T[i]$,
- $\texttt{write}(T, i, e)$: store element e at position i in T,
- $\texttt{grow}(T)$: increase the size of T,
- $\texttt{shrink}(T)$: decrease the size of T.

As usual, we assume elements have indices between 0 and $n-1$ and are all of equal length, coinciding with the size of memory words (32 or 64 bits in practice).

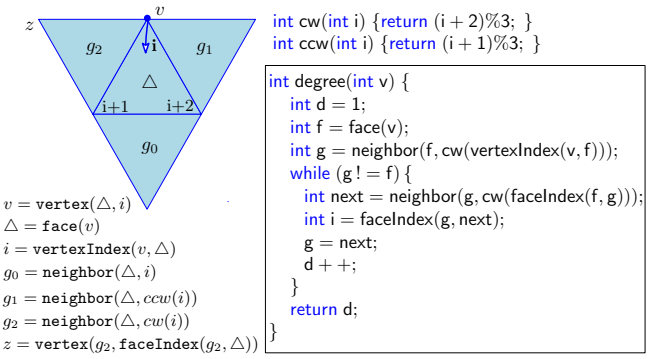

Fig. 1: Abstract data type for triangle meshes: our representations support the same operators as in [1], [14].

Abstract data type for dynamic meshes

Our representations are *face-based* and implement the same interface as in [14] and [1] (the *Corner Table* data structure provides a slightly different interface). More precisely, we have the following operators:

- $\texttt{neighbor}(\triangle, i)$: retrieve the i-th neighbor of $\triangle$,
- $\texttt{vertex}(\triangle, i)$: retrieve the vertex with index i of $\triangle$,
- $\texttt{face}(v)$: access one triangle incident to v.

With a combination of these operators it is also possible to define other functions, such as:

- $\texttt{faceIndex}(\triangle_1, \triangle_2)$: return the index of $\triangle_1$ among the neighbors of $\triangle_2$.
- $\texttt{vertexIndex}(v, \triangle)$: return the index of vertex v in face $\triangle$.

The combination of the operators listed above allows to turn around a face in cw (or ccw) order, and thus to compute the degree of a vertex or to list its neighbors (as illustrated

in Fig. 1). The mesh can be dynamically modified under the following update operators: defining a complete set of operators for editing a mesh, as in previous works:
- `split`$(\triangle, p)$: subdivide a given triangle $\triangle$, into three new sub-triangles by inserting a new point p,
- `flip`(e): perform the flip of a given edge e,
- `delete`(v): remove a degree 3 vertex v, together with its three incident triangles.

III. Dynamic compact meshes: general scheme

Overview of our approach

As in previous compact representations, we exploit a re-ordering approach to reduce the number of references representing the map from vertices to faces and the map from faces to vertices. Here, we further explore the power of such re-ordering strategies. Instead of re-ordering mesh elements (typically faces [17] or edges [15]), we rather exploit a permutation of geometric data associated with vertices. We avoid the indirection between triangles and vertices, by storing the vertex coordinates together with the data associated with one of its incident triangles: in this way we save 3 references for each triangle. The main novelty is to show that a coherent numbering of triangles and vertices can be maintained efficiently after local changes. Furthermore, as suggested in SQUAD [17] and [14], we also perform a grouping of few adjacent triangles in patches: this further reduces the number of references between neighboring elements.

Scheme description

Given a (triangular) mesh $\mathcal{T}$ having f faces, we consider a partition of its faces into *patches*. A patch is a face-connected triangulation with some marked vertices, namely a set of adjacent triangles whose dual graph is connected: in practice we will deal with patches which are simply connected and with a simple boundary. Given a face partition of $\mathcal{T}$ into patches, we say that such a partition is *valid* if the two following conditions are satisfied. Any face belongs to one and only one patch: patches can only share edges and vertices. A vertex $v \in \mathcal{T}$ may belong to several distinct patches, but is *matched* only once, and is marked in the corresponding patch.Two patches have the same shape if there is a bijection between their vertices that preserve triangles and marks. Two patches are of the same *type* (s, b, m), if they have the same number s of triangles, the same number m of marked vertices and boundaries of the same size b (b is the number of boundary edges). A *catalog* is the set of distinct (types of) patches defining the partition of the mesh: the size $|\mathcal{C}|$ of the catalog is the number of (different) patches it contains.

Catalog-based representation: As in previous array-based representations [15], [16], [17], [18], [13], our data structure represents the mesh connectivity by storing a given number of references describing incident relations between mesh elements. Let us consider a partition of $\mathcal{T}$ based on a catalog $\mathcal{C}$ with patches of k different types. We represent the connectivity of $\mathcal{T}$ storing the references in k tables $T_0, T_1, \ldots T_{k-1}$: table T_i stores the connectivity of a patch with s_i triangles and m_i marked vertices. More precisely T_i stores, for each patch of type (s_i, b_i, m_i), a block of b_i integers in consecutive memory words: they represent the b_i references to faces in neighboring patches. Geometric information (and other information associated to vertices or triangles), are stored in a collection $\{P_j\}_{0<j\leq k}$ of tables containing vertex coordinates, each associated to a table T_j. P_j has $|T_j|$ rows (one for each row in T_j) and m_j columns (one for each vertex marked in a patch stored in T_j, if $m_j = 0$ the table may be empty). The entries of P_j are in correspondence with the patches stored in the associated table T_j. More precisely, the a-th row $P_j[a]$ stores the coordinates corresponding to vertices $w_1, \ldots w_{m_j}$ which are matched by the a-th patch in table T_j. When required, additional informations associated to the faces can also be stored in T_j.

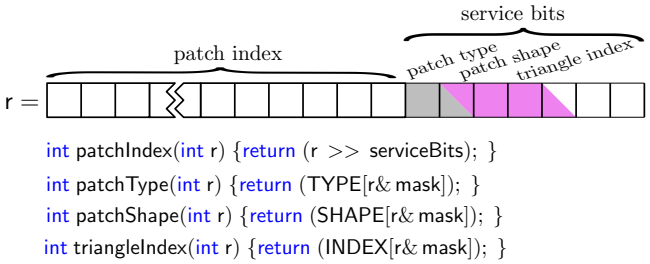

Fig. 2: References encoding.

References encoding: Recall that, according to our decomposition, each triangle belongs to exactly a patch, and each vertex is associated to a patch. Thus, in order to be able to refer to a given triangle $\triangle$, a *reference* is composed of the following data:
- a *patch type*: a code for indicating the type of patch p containing $\triangle$,
- a *patch shape*: a code for indicating the shape of patch p containing $\triangle$,
- a *triangle index*: a code for indicating the face $\triangle$ among the triangles contained in p,
- a *patch index*: the index of p in table T_j (T_j being the table containing the patches of same type as p).

A reference is encoded as an integer value r (typically on 32 bits): first bits (also referred to as *service bits*) are reserved for storing the three first codes, while remaining bits encode the patch index. Since in our representations the number of arrays, as well as the number of triangles per patch, are fixed, the first three codes require a constant number of bits (between 1 and 4 bits suffice in practice). The main part of a reference (representing the index of patch p, stored in T_j), requires $\lceil \log_2 |T_j| \rceil$ bits. Retrieving service bits and patch indices can be performed with a combination of *bit shift* and *bit mask* operations (see Fig. 2). Observe that we do not need to store references to vertex data, which actually can be retrieved by the correspondence between vertices and matched patches.

Limitations: As one can observe, our approach apply to meshes for which we can design an efficient matching correspondence (between vertices and patches) that can be

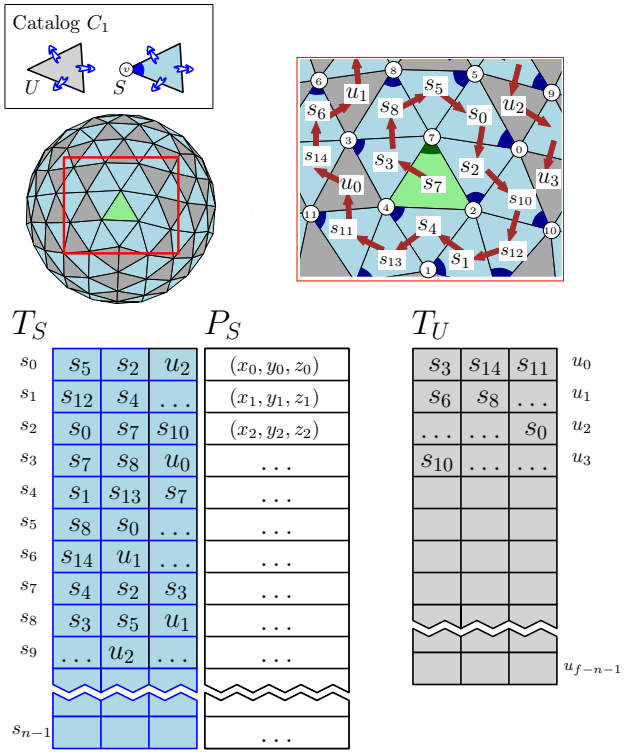

Fig. 3: Construction of the ESQ data structure (catalog $\mathcal{C}_1$). Triangles and vertices are matched according to a *Edgebreaker* traversal, starting from the *seed face* (green triangle). Patches of type S (blue faces) are stored in table T_S: they are ordered according to the index of the matched vertex (table P_S stores vertex coordinates). Table T_U stores the unmatched patches (gray faces).

```
int neighbor(int r, int i) {
    if (patchType(r) == S)
        return tableS[patchIndex(r) * 3 + i];
    else
        return tableU[patchIndex(r) * 3 + i];
}
```

```
int vertex(int r, int i) {
    if (patchType(r) == S && i == 0)
        return patchIndex(r);
    int f = neighbor(r, cw(i));
    int j = faceIndex(r, f);
    while (f != r) {
        if (j == 1 && patchType(f) == S)
            return patchIndex(f);
        int next = neighbor(f, ccw(j));
        j = faceIndex(f, next);
        f = next;
    }
}
```

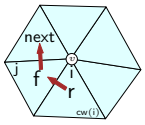

Fig. 4: Implementation of navigational operators.

easily maintained under dynamic updates. In the next sections we describe a few examples of data structures instantiating the general scheme described above, for the case of triangle meshes without boundaries (the case of boundaries is more involved and discussed in the last section). The choice of different catalogs of patches has a deep impact on the performances of mesh representations: for example, a small catalog (such as $\mathcal{C}_1$ consisting of few types of patches) is simpler to implement and faster in practice compared to a bigger catalog (such as $\mathcal{C}_2$) with better time-cost guarantees.

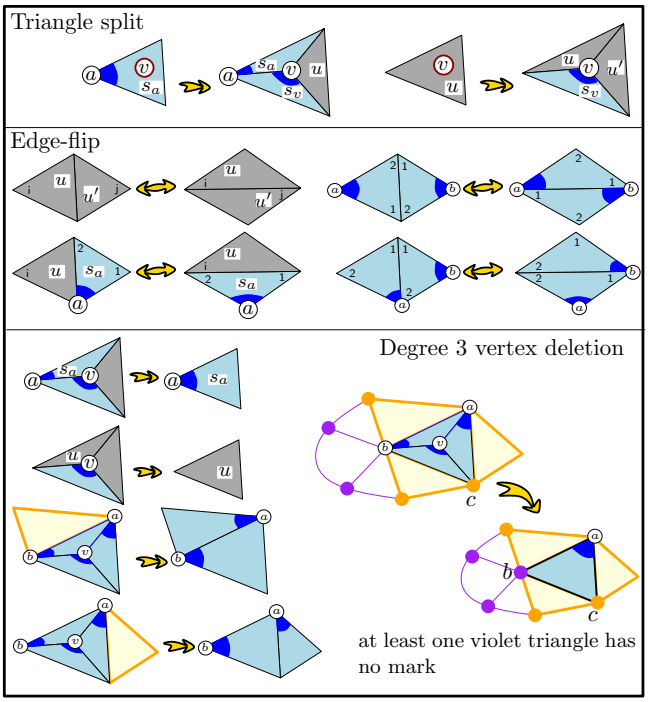

Fig. 5: Update operations for the catalog $\mathcal{C}_1$.

IV. Instantiating the General Scheme

A. The simplest catalog

The smallest catalog, denoted by $\mathcal{C}_1$, contains two types of patches, having only one triangle with zero or one matched vertex: we call U the patch with one triangle and no matched vertex, and S the triangle associated with one vertex (refer to Fig. 3).

Storage requirements: In such a case we have exactly two arrays of size n and $f - n$ respectively, one for patches of type U and one for patches of type S: with some abuse of notation, we denote them by T_U and T_S. Since the triangles in T_U are unmatched, we need only one table P_S to store vertex coordinates: P_S has one column and n rows. $P_S[i]$ stores the coordinates of v_i, while $T_S[i]$ stores the 3 neighbors of the triangle (denoted by s_i) matched with v_i. As each patch stores 3 references to adjacent triangles, the cost for connectivity is $6\,rpv$ (recall that there about $2n$ faces in a triangulation). Finally, observe that for this simple catalog we need only one service bit, for distinguishing between patches of type U and S: no need of more service bits, since all patches consist of exactly one triangle.

Matching triangles and vertices: The construction of our data structure relies on a matching between faces and vertices. This matching is dynamically maintained by the update operation described later, but if a triangulation is already given, then this matching can be computed as in [16] visiting the faces, according to an *Edgebreaker* traversal of the dual graph. The algorithm visits triangles (and vertices) in a depth-first manner (visiting right neighbor first), starting from a *seed face* (green face in Fig. 3): each time an unvisited vertex

v is reached, it is *matched* to the current incident triangle. As for the Edgebreaker coder, the traversal ends when all faces and vertices have been visited, producing a valid partition (for more details see [16]).

Implementing navigational operators: For the sake of completeness, we briefly sketch how to implement the navigational operators. Let us denote by r (resp. v) a reference to a given triangle (resp. vertex): recall that v ranges between 0 and $n-1$, while r is an integer whose first bit (service) describes the type of patch (the remaining bits encode the triangle index, which ranges between 0 and at most n). As one could expect, the implementation of `neighbor` operator is straightforward and fast, as it requires only one access in tables T_U or T_S. The implementation of `vertex` operator is more involved (as in [15], [16], [17], [18]): as we did not explicitly stored references to vertices, we have to iteratively turn around a given vertex until the corresponding matched triangle is reached (the pseudocode is shown in Fig. 4).

Update operators and performance analysis: Update operations for catalog $\mathcal{C}_1$ are described in Fig. 5. Most update operations are quite easy to implement, involving a constant number of references updates in arrays T_U and T_S: the only exception concerns the case of vertex deletion.

The only difficult operation is the deletion of a degree 3 vertex, when all the three incident triangles have a marked vertex. In one triangle the marked vertex is the deleted one v, but after the deletion two vertices a and b are unmarked and we have a single triangle abc to assign it. The first workaround is found if one of the neighboring triangle has no marked vertex (Fig 5-bottom-left), then this triangle can be assigned one of the two orphan vertices. If the three neighboring triangles already have a marked vertex, only one of the triangles incident to edges bc and ca can have c as a marked vertex, assume without loss of generality that c is not marked in the triangle incident to bc; then a is marked in abc and among all the triangles incident to b at least one has to be free of mark (since c is marked in a triangle not incident to b) thus b can be assigned to that triangle (Fig 5-bottom-right).

Although this last operation cannot be guaranteed to be in constant time and may require a time linear in the degree of b, it can be observed that this search for an unmarked triangle incident to b can be stopped as soon as such a triangle is found. Since half of the triangles are unmarked this exploration in average will not need to explore all the neighbors of b even if the degree of b is large.

B. A catalog with guaranteed constant time vertex removal

A slight modification of catalog C_1 allows to obtain a data structure that ensures constant time vertex removal, to the price to a less efficient edge flip (but still in constant time) (refer to Figure 6). We add a new type patch to the catalog, called D, consisting of one triangle with two incident matched vertices. We obtain a catalog C_2 having three types of patches (thus implemented with 3 tables T_U, T_S and T_D), consisting each of 1 triangle and having at most two matched vertices. Two service bits are needed by Catalog C_2.

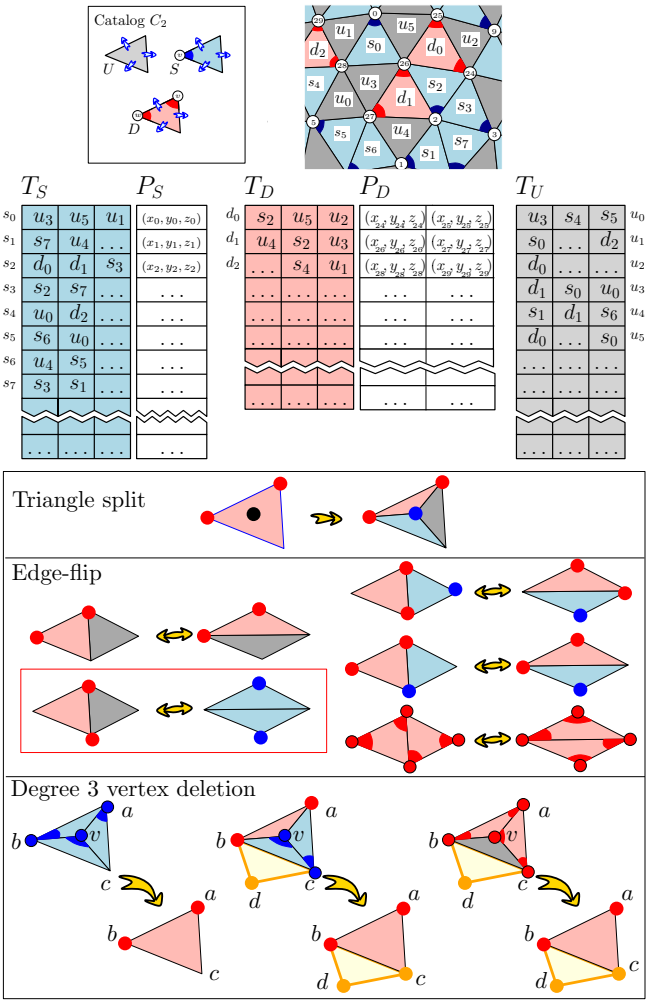

Fig. 6: All updates operations can be perform in amortized $O(1)$ time, with the slightly richer catalog $\mathcal{C}_2$.

Performance analysis: Since each patch contains one triangle, the storage analysis is the same as for catalog C_1, leading to a cost of $6\,rpv$. Triangle split can be easily performed as before. The main differences concern the support of edge flip and vertex deletion.

Edge flips: As for catalog $\mathcal{C}_1$, the correctness relies on an exhaustive case analysis. In addition to the cases already considered in Fig. 5, we have to deal with few more configurations, involving patches of type D. As depicted in Fig. 6, edge flips can always be supported with a constant number of memory changes. In most cases we need only some few `read` and `write` operations, which can be performed in worst case $O(1)$ time on dynamic arrays. Unfortunately, there remains one case (refer to Fig. 6), where the edge flip implies the creation of two new patches of type S, and two removals (of patches D and U respectively): it still leads amortized $O(1)$ time (the cost for updating tables T_S, T_D and T_U) but is clearly less efficient compared to catalog C_1 where the patches are just modified in place in the tables T_U and T_S.

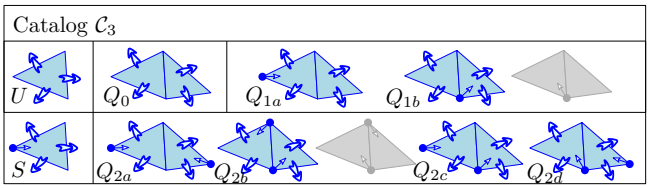

Fig. 7: The triangle/quad catalog $\mathcal{C}_3$ requires at most $4.8rpv$.

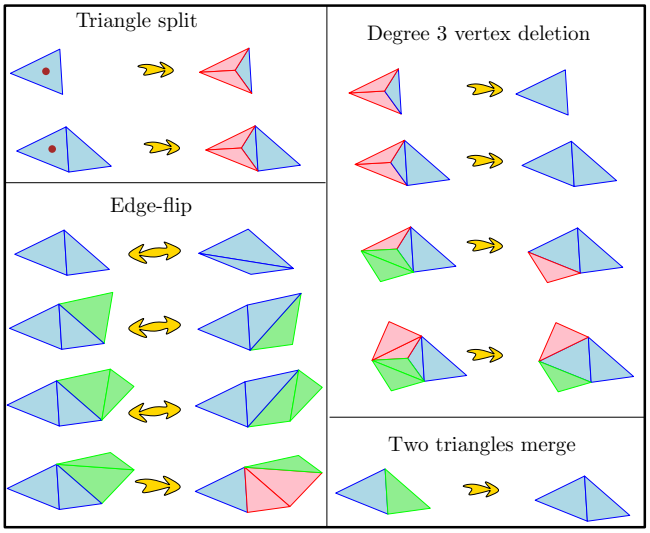

Fig. 8: Update operations for the catalog $\mathcal{C}_3$. We omit to mark vertices (we use colors to distinguish neighboring patches).

Vertex removals: The advantage of catalog $\mathcal{C}_2$ is in the way we can perform vertex deletions more efficiently. As depicted in Fig. 6, we can use an adjacent triangle (say triangle (c, b, d)), in order to distribute the remaining vertices. For example, vertices a and b can be associated to triangle (a, b, c) of type D, while the remaining vertex c is matched by triangle (c, b, d): observe that such triangle is originally of type U or S, and becomes of type S or D after the removal of v. The removal/addition of a constant number of patches in tables T_S, T_U and T_D can be supported in amortized $O(1)$ time.

V. Triangles and quads: more compact catalogs

Here we consider catalogs containing patches which are simple triangles or *quads* (groups of paired adjacent triangles): as before they may have zero or more marked vertices.

Catalog $\mathcal{C}_3$: guaranteed upper bound: The catalog $\mathcal{C}_3$ (see Fig. 7) contains triangles with zero or one marked vertex and quads (groups of two triangles with zero or one marked vertex each). We need five catalogs T_U, T_S, T_{Q_0}, $T_{Q\cdot 1}$, and T_{Q_2}. For the type Q_1 and Q_2 there are several possible shapes depending on the position of the marked vertices with respect to the quad's diagonal and the triangle matched by the marked vertices. The gray pictures on Fig. 7 can be avoided yielding 16 different triangles on that pictures allowing 4 service bits to distinguish all cases. Update operations are sketched in Fig. 8. The way the vertices are marked is not described and is identical to the one in Section IV-A.

Performance analysis: If we enforce that two neighboring triangles are always merged in a quad, then we can lower bound the number of quads by $\frac{3}{5}n$ and upper bound the number of isolated triangles by $\frac{4}{5}n$ and get that we need only $4.8\,rpv$, as shown in [14].

VI. Experimental results

In order to evaluate practical performances, we have written Java array-based implementations of the triangle data structure [1] (TDS) and of our ESQ data structure (catalog $\mathcal{C}_1$). Table II reports the experimental results obtained by TDS and ESQ representations on the tested 3D models: we compare both navigational and update operators. Our tests confirm the practical interest of ESQ approach: time-cost performances of ESQ are close to the ones of TDS, while the storage gain is significant (TDS is not compact, using $13rpv$).

Navigational operators: As one could expect, the ESQ implementation of the `vertex` operator is slower than TDS (about 12 times slower): recall that to retrieve a bounding vertex we have to turn around a vertex until the matched triangle is found (while TDS needs a memory access). Quite surprisingly, ESQ is (slightly) faster than TDS when comparing `degree` operator. The main reason is that ESQ can take advantage of the matching correspondence between triangles and faces. Given a vertex v we know that its index in s_v (the matched triangle) is 0, so the clock-wise neighboring face around v is given by `neighbor`$(s_v, 2)$: which allows to save some computations compared to the TDS implementation.

Update operators: When evaluating the `split` operators we only consider the combinatorial cost: we do not perform geometric calculations, and we split triangles just performing connectivity changes. As shown by our experiments, ESQ has (slightly) better performances than TDS: observe that both data structures perform the same reference updates to neighboring faces, while TDS has to update also vertex references (which do not exist in ESQ). Concerning the `flip` operator, ESQ is slower than TDS (about 1.7 times slower, in our tests). This comes from the number of cases to consider (as illustrated in Fig. 5): in order to distinguish all different cases ESQ has to perform a number of tests and bit-wise operations (to retrieve the patch type of the two neighboring faces).

Construction: The preprocessing time for our ESQ data structure is also shown: it includes the matching phase, as well as the construction step (memory allocation and references setting). In all our tests, vertices and faces are accessed consecutively. We run our experiments on a Dell XT2, equipped with a Core 2 Duo 1.6GHz, Java 1.6 (the JVM using 1GB heap memory), under Windows 7 (32 bits).

VII. Concluding remarks and extensions

We have proposed compact representations for triangle meshes supporting local editing operations. Our data structures are provided with guaranteed bounds (between $4.8\,rpv$ and $6\,rpv$) and experimental evaluation. Further improvements and extensions are possible, according to the remarks below.

	statistics			preprocessing (seconds)		vertex (ns)		degree (ns)		split (ns)		flip (ns)	
3D model	vertices	faces	genus	matching	ESQ construction	TDS	**ESQ**	TDS	**ESQ**	TDS	**ESQ**	TDS	**ESQ**
Bague	2652	5.3K	1	0.01	0.02	12	132	318	303	2005	1749	335	591
Aphrodite	46096	92K	0	0.08	0.06	12	134	347	330	796	655	331	575
Feline	49864	99K	2	0.11	0.07	10	138	354	324	774	645	324	578
Camille's hand	195557	391K	0	0.28	0.64	11	151	445	364	799	612	383	601
Eros	476596	953K	0	0.51	0.55	11	129	322	294	701	581	336	576
Pierre's hand	773465	1.54M	0	0.8	1.96	10	117	285	274	783	589	348	583

TABLE II: Experimental results. We compare the time cost performances of our ESQ data structure (Catalog $\mathcal{C}_1$, using $6\,rpv$), with the triangle based data structure [1] (TDS), which is not compact and requires $13\,rpv$. Timings are expressed in terms of nanoseconds(ns) per operation. The preprocessing time (in seconds) for the construction of ESQ is also reported.

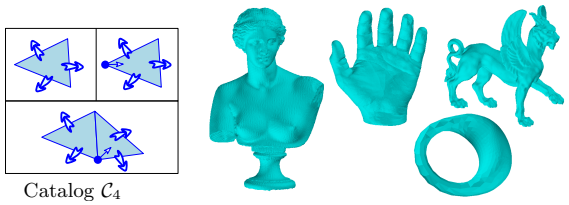

Fig. 9: Left: triangle-quad catalog $\mathcal{C}_4$. Right: our tested meshes are manifold with no boundary (genus 0, 1 and 2).

Meshes with boundaries: Our results hold for triangle meshes which are manifold and without boundaries (and with arbitrary fixed genus). Dealing with boundaries is more involved and requires to design new matching rules (for assigning vertices to faces after local changes). Furthermore, our bounds for the triangle-quad catalogs should be updated to take into account boundaries of arbitrary size (relying on a slightly different counting argument).

Total vs. partial pairing: In the static setting, one way to obtain better bounds (for the triangle-quad catalog) is to compute a perfect matching between pairs of adjacent triangles. Unfortunately, it is not clear how to maintain such a pairing when local changes are performed (without re-processing the entire mesh). A better trade-off between time-cost performances and lower memory occupancy could be achieved, in practice, with a smaller triangle-quad catalog $\mathcal{C}_4$ (which would be simple to implement, involving only three types of patches, S, U single triangles and Q for quads). According to experimental tests (see [17]), the matching and pairing strategy of the *SQuad* representation allows to regroup most of triangles into quads, leading to a cost of about $4\,rpv$.

Acknowledgment

This work is supported by ERC (agreement ERC StG 208471 - ExploreMap).

References

[1] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec, "Triangulations in CGAL," *Comp. Geometry*, vol. 22, pp. 5–19, 2002.
[2] B. G. Baumgart, "Winged-edge polyhedron representation," Stanford, Tech. Rep., 1972.
[3] ——, "A polyhedron representation for computer vision," in *AFIPS National Computer Conference*, 1975, pp. 589–596.
[4] S. Campagna, L. Kobbelt, and H. P. Seidel, "Direct edges - a scalable representation for triangle meshes," *Journal of Graphics tools*, vol. 3, no. 4, pp. 1–12, 1999.
[5] L. J. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and computation of voronoi diagrams," *ACM Trans. Graph.*, vol. 4, no. 2, pp. 74–123, 1985.
[6] C. L. Lawson, "Software for C^1 surface interpolation," in *Math. Software III*, J. R. Rice, Ed. New York, NY: Academic Press, 1977, pp. 161–194.
[7] O. Devillers, "The Delaunay hierarchy," *Internat. J. Found. Comput. Sci.*, vol. 13, pp. 163–180, 2002.
[8] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Lévy, *Polygon Mesh Processing*. AK Peters, 2010.
[9] J. Rossignac, "Edgebreaker: Connectivity compression for triangle meshes," *Trans. Vis. and Comput. Graph.*, vol. 5, pp. 47–61, 1999.
[10] T., H. Lopes, J. Rossignac, and A. W. Vieira, "Efficient edgebreaker for surfaces of arbitrary topology," in *Sibgrapi*, 2004, pp. 218–225.
[11] D. Poulalhon and G. Schaeffer, "Optimal coding and sampling of triangulations," *Algorithmica*, vol. 46, pp. 505–527, 2006.
[12] W. Tutte, "A census of planar triangulations," *Canadian Journal of Mathematics*, vol. 14, pp. 21–38, 1962.
[13] T. J. Alumbaugh and X. Jiao, "Compact array-based mesh data structures," in *Proc. of 14th Int. Meshing Roundtable (IMR)*, 2005, pp. 485–503.
[14] L. Castelli Aleardi, O. Devillers, and A. Mebarki, "Catalog-based representation of 2D triangulations," *Int. J. Comput. Geometry Appl.*, vol. 21, no. 4, pp. 393–402, 2011.
[15] L. Castelli Aleardi and O. Devillers, "Explicit array-based compact data structures for triangulations," in *Proc. 22th Ann. Internat. Sympos. Algorithms Comput.*, ser. LNCS, vol. 7074, 2011, pp. 312–322.
[16] T. Gurung and J. Rossignac, "*SOT*: compact representation for tetrahedral meshes," in *Proc. of the ACM Symp. on Solid and Physical Modeling*, 2009, pp. 79–88.
[17] T. Gurung, D. E. Laney, P. Lindstrom, and J. Rossignac, "SQuad: Compact representation for triangle meshes," *Comput. Graph. Forum*, vol. 30, no. 2, pp. 355–364, 2011.
[18] T. Gurung, M. Luffel, P. Lindstrom, and J. Rossignac, "LR: compact connectivity representation for triangle meshes," *ACM Trans. Graph.*, vol. 30, no. 4, p. 67, 2011.
[19] M. Lage, T. Lewiner, H. Lopes, and L. Velho, "CHF: A scalable topological data structure for tetrahedral meshes," in *Sibgrapi*, 2005, pp. 349–356.
[20] J. Snoeyink and B. Speckmann, "Tripod: a minimalist data structure for embedded triangulations," in *Workshop on Comput. Graph Theory and Combinatorics*, 1999.
[21] W. Schnyder, "Embedding planar graphs on the grid," in *SODA*, 1990, pp. 138–148.
[22] L. Castelli Aleardi, O. Devillers, and G. Schaeffer, "Succinct representations of planar maps," *Theor. Comput. Sci.*, vol. 408, no. 2-3, pp. 174–187, 2008.
[23] ——, "Succinct representation of triangulations with a boundary," in *WADS*. Springer, 2005, pp. 134–145.
[24] ——, "Dynamic updates of succinct triangulations," in *CCCG*, 2005, pp. 135–138.
[25] M. Kallmann and D. Thalmann, "Star-vertices: a compact representation for planar meshes with adjacency information," *Journal of Graphics Tools*, vol. 6, pp. 7–18, 2002.
[26] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick, "Resizable arrays in optimal time and space," in *WADS*, 1999, pp. 37–48.
[27] V. Raman and S. S. Rao, "Succint dynamic dictionaries and trees," in *ICALP*. Springer, 2003, pp. 357–366.