Query-replace operations for topologically controlled 3D mesh editing

Guillaume Damiand, Vincent Nivoliers

Query-replace operations for topologically controlled 3D mesh editing

Guillaume Damiand, Vincent Nivoliers

Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS, UMR5205, F-69622 Villeurbanne, France

ARTICLE INFO

Article history:
Received July 8, 2022

3D Volumic Mesh, Transformation Operation, Topological Validity, Query-Replace.

ABSTRACT

We propose a generic framework to describe modifications on 3D objects (surfaces and volumes) based on a query and replace mechanism. These modifications are described in terms of rewriting rules: a set of patterns is provided, each one defining a possible replacement. The patterns are expressed using 3D combinatorial maps, ensuring the global topological validity of the transformed mesh. At the core of the framework, we use topological signatures to efficiently query and match patterns on the input. Our formalism can generically describe many different transformations like subdivision, smoothing, topological correction, or remeshing. Its interest is twofold. It provides an easy implementation for such operations, especially when many cases arise: we provide an example with over 300 cases. It is also able to detect corner cases that were not provided and are required to ensure the topological correctness of the output.

1. Introduction

Many operations can be designed on volumetric meshes to build or modify them, such as the generation of conformal hexahedral meshes using adaptive refinement [1], hexahedral meshes based on symmetric moving frames [2] or octrees [3] and hybrid mesh generation in parallel based on refinement and coarsening [4] for instance. Non-conventional meshes made of a wider class of polygons rather than just triangles are also getting used in computer graphics in applications based on tilings such as texture generation, sampling theory, remeshing, and generation of decorative patterns [5]. New methods are developed to mesh surfaces with such meshes [6] or exploit their aperiodic structure in procedural generation [7].

These applications on various types of meshes (hexahedral meshes, mixed polygonal or polyhedral elements) emphasize the need for operations to transform these meshes. Many of the transformation operations in the above mentioned contributions – adaptive refinement or tiling for instance – can be seen as *query-replace* operation: a *target* cell is transformed by replacing its interior by another subdivision given by a *pattern*. The usual pipeline to perform such operations is first to identify for each element the corresponding case, and then to insert the replacement. Detecting cases depends on the problem, and is generally performed by indexing the cases and designing an algorithm computing for each input cell the corresponding index. Adding the replacements requires identifying the entities (vertices, edges, faces) to be matched and connected for each case. With many cases, this becomes a tedious task, and factoring the code for this identification on multiple cases is error prone.

Our main contribution in this work is the definition of a generic query-replace framework that allows to: (1: query) quickly identify portions of the input mesh compatible with a set of desirable modifications and (2: replace) locally remesh the interior of the identified portion of the input mesh by the interior of the provided pattern, ensuring a topologically valid output. In our work, we focus on replacing single faces or volume elements of the mesh. Faces can have any number of vertices and volume elements can have any connected manifold as boundary. The problem is especially complex for volume elements where applying a replacement requires matching arbitrary complex manifolds.

Our method has several advantages that will be detailed in the following: (1) it is generic (works for patterns and targets with any topology); (2) it is easy to use: patterns can be designed with a 3D modeler in which case there is no code to write; (3) it produces a mesh with a valid topology avoiding topological cracks; (4) the query from a set of patterns is efficient thanks to the use of topological signatures; (5) the query tool can be used in order to prove the validity and completeness of a set of patterns, to avoid missing corner cases. To the best of our knowledge, our solution is the first operation of this kind providing all these properties.

1.1. Related Work

Several works about mesh generation and transformation exist based on the idea of using transformation rules. For example, in procedural modeling, 3D objects can be generated automatically, using some randomness to add variety to the final result (see for example [7]). But such techniques usually are only able to generate a mesh and do not allow to transform an existing mesh.

Shape grammars [8] and L-systems [9] are methods allowing to generate meshes based mainly on grammar and rewriting rules

*Corresponding author
e-mail: guillaume.damiand@cnrs.fr (Guillaume Damiand)

(several variants exist based on the same principle). These methods allow mainly to create a mesh. Due to their recursive nature, they can be easily used to create fractal objects. For this reason, they were used for example to describe plants and other branching structures [10] or architectural models of buildings of different styles [11]. The grammar is defined based on a number of symbols that represent components of the mesh, and symbols are transformed according to rewriting rules. [12] proposes an extension of L-systems to 3G-maps (a variant of 3D combinatorial maps) which is also based on symbols associated with volumes and faces of the mesh. The transformation rules of these methods use symbols and the validity of the produced mesh depends on the validity of the rules defined by users. Moreover, the definition of the rules can be a complicated and long task requiring an high level of expertise.

Jerboa [13] uses a different approach. This is a topological modeler based of G-maps, where all operations are defined through particular graph transformation rules. These rules are defined graphically in the modeler, and a specific formalism must be learned before to be able to define a new rule. A rule can be applied if a morphism exists between the left part of the rule and the graph.

Our approach uses also the principle of rewriting rules. The main differences between these previous works are: (1) our rules are defined with patterns given as meshes. They can be designed with a 3D modeler, and there is no code to write, no new formalism to learn. (2) we guarantee the topological validity of the result whatever the rules defined. (3) searching a rule among a huge set is fast thanks to the use of our topological signature.

1.2. Outline

Preliminary notions are introduced in Section 2 before providing an introductive example in Section 3 to generally outline our method. We then define in Section 4 the query part of the operation and in Section 5 the replace part. Section 6 states and proves our claims about the properties and strengths of our method and Section 7 finally provides several experiments to demonstrate its use in practical applications.

2. Preliminaries and Related Work

2.1. Mesh Data Structures

Various data structures exist to describe meshes, depending on the desired complexities. Winged edges [14], Doubly Connected Edge List [15], half-edge data structure [16] or Surface Meshes [17] are based on edges and allow for efficient queries for most traversal operations. Corner Tables [18] are based on vertices and are especially interesting for real time rendering with a data layout that more closely matches that of the classical rendering pipeline. In this work, we use Combinatorial maps [19, 20] which can be seen of a generalization of half-edge meshes which can handle higher dimensional cell complexes. Comparisons for these various data structures can be found in [21] and [20].

Our main reason for using combinatorial maps is that it allows for the description of volumetric cell complexes with a fine grain control over the topology of the complex. 2D faces and 3D cells are not limited to triangles, tetrahedrons or even convex shapes. With such a data structure, our query and replace operations can precisely identify the regions of the input map where our replacement patterns can be applied without topological errors. Reconnecting our replacements can be done efficiently and without ambiguities, leaving a properly connected output map.

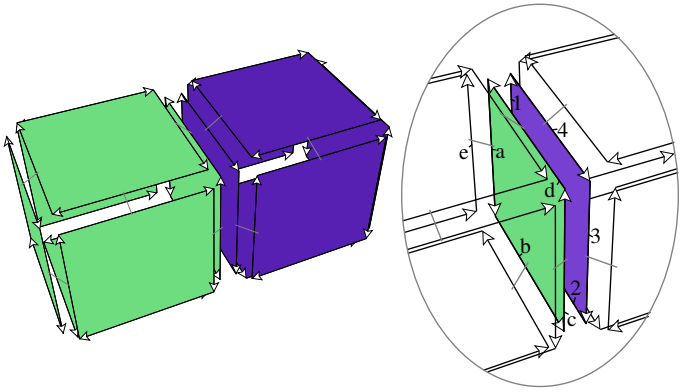


Fig. 1: Example of a 3-map encoding two adjacent cubes. Faces are drawn with small gaps between them, in order to differentiate the darts. A detailed zoom of the interface between the two cubes is provided on the right.

2.2. Combinatorial Maps

Combinatorial maps use *darts* as their core primitives. Darts are attached to the edges of the cell complex, but duplicated for each volumetric cell and each polygonal face incident to the edge. The connectivity between these darts is encoded by functions mapping darts to others. In terms of implementation, these functions can be stored as pointers or indices.

Definition 1 (3D Combinatorial map [19]). A $3D$ combinatorial map (3-map) is defined by a tuple $M = (D, \beta_1, \beta_2, \beta_3)$ where

- D is a finite set of darts;
- β_1 is a *permutation* on D, *i.e.*, a one-to-one mapping from D to D;
- β_2 and β_3 are *involutions* on D, *i.e.*, a one-to-one mapping from D to D such that $\beta_i = \beta_i^{-1}$;
- $\beta_1 \circ \beta_3$ is an *involution* on D.

Intuitively, β_1 connects the darts in order to circulate clockwise around the 2D face containing the dart. It is a permutation because the boundary of every face is a cycle. We note β_0 for β_1^{-1}, allowing to turn around faces with respect to the opposite orientation. β_2 connects darts associated to the same edge on adjacent faces of the same volumetric cell. It therefore allows to circulate the surface corresponding to the boundary of a single volumetric cell. β_3 connects darts corresponding to the same edge on the opposite face of an adjacent 3D cell. Using β_3 therefore allows to circulate between volumetric cells. A dart d is said to be i-sewn with another dart d' if $d = \beta_i(d')$.

Comparing with half-edges, β_0 is equivalent to *previous*, β_1 is equivalent to *next*, β_2 is equivalent to *opposite*, and β_3 is a new relation similar to *opposite* but for volumes instead of faces.

In some cases, it may be useful to allow some β_i to be partially defined, thus leading to open combinatorial maps. The idea is to add an element ϵ to the set of darts, and to allow darts to be i-sewn with ϵ. By definition, $\forall 0 \leq i \leq 3, \beta_i(\epsilon) = \epsilon$. A dart d is said i-free if $\beta_i(d) = \epsilon$. In this work, we restrict this possibility to

β_3 for volumetric meshes, prohibiting this for β_1 and β_2 in order to avoid degenerated configurations, while allowing meshes with boundary, i.e. some volumes can have no adjacent volume along some of their faces. A dart d 3-free belongs to the boundary of the mesh: the face containing d is incident to only one volume. For surface meshes without β_3, we can allow β_2 to be null, thus allowing meshes with boundaries.

An example of a 3-map is provided in Fig. 1. It contains 48 darts representing two adjacent cubes. The closeup shows the face separating the two cubes to illustrate the different dart operators. We have for example $\beta_0(a) = d, \beta_1(a) = b, \beta_2(a) = e$, and $\beta_3(a) = 1$.

2.3. Combinatorial Maps Isomorphism

A combinatorial map can be seen as a graph with darts as vertices, and connected by the β functions. For our query and replace algorithm, we need a tool to match maps. In terms of graphs, we therefore need to compute graph isomorphisms.

Definition 2 (Map Isomorphism [19]). Two 3-maps $M = (D,\beta_1,\beta_2,\beta_3)$ and $M' = (D',\beta'_1,\beta'_2,\beta'_3)$ are isomorphic if there exists a one-to-one mapping $f : D \to D'$, called *isomorphism function*, such that $\forall d \in D, \forall i \in \{1,2,3\}\ f(\beta_i(d)) = \beta'_i(f(d))$.

This definition has been extended to open maps in [22] by adding that $f(\epsilon) = \epsilon$, thus enforcing that when a dart is i-sewn with ϵ the dart matched to it by f is also i-sewn with ϵ.

In the general case, computing graph isomorphisms is a hard problem, but efficient algorithms exist for planar graphs [23, 24]. The main idea is that on such graphs, an order can be defined on the neighbors of each vertex thus allowing for a deterministic traversal only depending on the starting vertex. This generalizes for combinatorial maps in any dimension [25], ordering the neighboring darts by increasing subscript on the β functions: from a given dart d, if $i < j$, then $\beta_i(d)$ will be handled before $\beta_j(d)$. Using any deterministic traversal (DFS of BFS for instance), the darts of the first map can be indexed and the indexing only depends on the starting vertex chosen for the traversal. On the second map, an identical traversal can be started for every dart as a starting point. If the maps are isomorphic, there exists at least one starting vertex on the second map which will lead to a matching indexing of the darts. Traversals shall therefore be performed for one dart d of M, and all starting darts d' of M', yielding a time complexity of $O(|D|^2)$ (we can assume that $|D| = |D'|$ as the contrary trivially prevents the maps to be isomorphic).

2.4. Signatures of Combinatorial Maps

The goal of map signatures [26] is to define a canonical traversal for each map (when the map is automorphic, this canonical traversal can be obtained from multiple starting darts). To do so, a word is built from the traversal, using a labeling of the darts and using their neighborhoods. These words can be seen as some sort of perfect hashing for the graphs. Checking for graph isomorphism therefore reduces to ensure that their corresponding words are identical. This allows for signature precomputation, and the use of Hash tables with signatures as keys.

Definition 3 (Labelling [26]). Given a 3-map $M = (D,\beta_1,\beta_2,\beta_3)$, a labeling of M is a bijective function $l : D \cup \{\epsilon\} \to \{0,\dots,|D|\}$ such that $l(\epsilon) = 0$.

Algorithm 1: *BFL(M,d)*

Input: $M = (D,\beta_1,\beta_2,\beta_3)$: a connected 3-map;
$d \in D$: a dart.
Output: a labeling $l : D \cup \{\epsilon\} \to \{0,\dots,|D|\}$

```
for each d' ∈ D do l(d') ← −1;
l(ε) ← 0; l(d) ← 1; l ← 2;
let Q be an empty queue;
add d at the end of Q;
while Q is not empty do
    remove d' from the head of Q;
    for i in {1,2,3} do
        if l(β_i(d')) = −1 then
            l(β_i(d')) ← l; l ← l + 1 ;
            add β_i(d') at the end of Q;
return l;
```

Labeling a 3-map can be done through classical graph traversal algorithms, Algo. 1 for instance uses a breadth first traversal and labels the nodes with complexity $O(|D|)$, since darts are connected to at most three other darts. The labeling will vary depending on the traversal algorithm, the initial dart used to start the traversal and the order in which the neighboring darts are considered. Here neighboring darts are considered in a deterministic order: β_1 then β_2 and finally β_3. The key insight of the graph isomorphism algorithm is that labellings obtained with graph traversals encode the topology of the graph.

Definition 4 (Word [26]). Given a connected 3-map $M = (D,\beta_1,\beta_2,\beta_3)$ and a labeling $l : D \cup \{\epsilon\} \to \{0,\dots,|D|\}$ the word associated with (M,l) is the sequence

$$W(M,l) =< w_{1\cdot1}, w_{2\cdot1}, w_{3\cdot1}, w_{1\cdot2}, \dots, w_{3\cdot|D|} >$$

such that $\forall i \in \{1,2,3\}$, $\forall k \in \{1,\dots,|D|\}$, $w_{i\cdot k} = l(\beta_i(d_k))$ where d_k is the dart labeled with k, i.e., $d_k = l^{-1}(k)$.

In the following, we only consider the breadth first labeling, and we denote by $W(M,d)$ the word associated with the breadth first labeling of map M, starting from dart d. $W(M,d)$ can be computed on the fly during the breadth first search algorithm by enumerating the three labels of $\beta_i(d)$ for each dart considered in the main loop (and labeling the neighboring darts with no label so far).

Theorem 1 (Word Isomorphism [26]). *Two connected 3-maps $M = (D,\beta_1,\beta_2,\beta_3)$ and $M' = (D',\beta'_1,\beta'_2,\beta'_3)$ are isomorphic iff there exists two darts $d \in D$ and $d' \in D'$ such that $W(M,d) = W(M',d')$.*

One pair of darts is enough to ensure that two maps are isomorphic. When they are, the set of all possible words that can be obtained by traversing M from its darts is equal to that of M'. Since words can be totally ordered using lexicographical order, a canonical word can therefore be defined as the minimum word for all starting darts.

Definition 5 (Map Signature [26]). The signature $S(M)$ of a connected $3-map$ $M = (D,\beta_1,\beta_2,\beta_3)$ is the word $W(M,d)$ for $d \in D$ such that $W(M,d) \leq W(M,d')$ for every $d' \in D$.

Two maps are therefore isomorphic provided their signatures are the same. In terms of time complexity, computing a map signature requires $O(|D|^2)$ elementary operations.

Using map signatures is interesting when one map will be tested many times against other maps. This is the case for our query and replace formalism, where the queries are maps to be tested against portions of the input map. We therefore precompute the signatures of our queries. In addition, when testing portions of the input map against our queries, computing the signature of the map portions allows for testing against every query without additional traversals.

3. Introductive Example

Mesh modifications are described in our formalism as a sequence of patterns. Each pattern describes modifications to be applied on matching faces of volumes of the input mesh. This matching is efficiently performed using signatures computed for each pattern and each face and volume of the input mesh. Fig. 2 illustrates this process to carve grooves is a cube element. First, the desired edges are flagged and cut in three. Then a first set of patterns splits the faces of the cube. One such pattern is provided in Fig. 3. Another pattern is used for faces with two opposite split edges. The volume cell is then split according to the pattern depicted on Fig. 5. The final result is then the cube with the groove carved.

Our formalism ensures that the combinatorial structure remains valid at each step with darts properly connected. Matching signatures provide matching labelings of the darts on the replaced piece of input map and the pattern. Each dart in the pattern can then be connected properly according to the labeling. In some situations however, relying on the sole signature is not enough due to automorphisms. In Fig. 3 for instance, the boundary of the pattern is a cycle with 6 vertices. Using only this topological information, the replacement can be applied in six different ways by rotating it in the cycle such as the cases provided in Fig. 4. To provide more control over how replacements should be applied, we therefore augment our pattern description with flags (depicted as red crosses) integrated in the signatures. These ensure that matched darts are identically flagged. When flagged darts are split, we use as a convention that the resulting dart starting at the same vertex as the original dart inherits the flag. This is illustrated on Fig. 2b.

4. Query

Our query replace operations are defined using a set of *patterns*. Each pattern is described by a 3-map. The boundary of the pattern is the portion of input mesh that will be searched for. When a cell of the input mesh is found with a matching boundary, the cell is replaced by the interior of the pattern. We distinguish *e-patterns* which will act on the input edges, *f-patterns* on the input faces and *v-patterns* on the input volumetric cells.

Depending on the application, the set of patterns may be big. Our idea is therefore to compute signatures for the boundaries of the patterns and store them in a hash map. For each cell of the input mesh, we can compute its signature and look for matches in the hash map. Since our patterns only aim at replacing single elements of the input map, there is no overlap between the patterns and the order in which the elements are processed does not influence the result.

To provide additional control over how the patterns should be applied, we added a flag mechanism integrated into the signatures. Flags can be assigned automatically based on geometrical or topological criteria or interactively. With flags, input elements and patterns can only be matched when a map isomorphism exists and preserves the flags.

4.1. e-pattern

The only useful operation on an input edge is to subdivide it into several edges. An *e*-pattern is therefore a path of k edges. In this case, we do not need to use signatures, the operation being totally defined by the number k. We remind that the operation that adds a vertex into an edge modifies all the darts of the edge in order to ensure the global validity of the combinatorial map (cf. [20] for all details).

4.2. f-pattern

4.2.1. Definition

An f-pattern is a set of connected faces, like the example given in Fig. 3. The boundary of the pattern will be matched against the boundaries of the input mesh faces. Since a face has to be a topological disc, its boundary is a cycle of darts. The boundary of an f-pattern therefore has to be a cycle of 2-free darts to be able to match some input cell.

We define the f-signature S_{FP} of an f-pattern as the number of edges in its boundary, and the f-signature S_F of a face as number of edges as well: this number of edges characterizes the topology of the boundary. Querying for an f-pattern therefore reduces to finding input faces with the right number of edges.

Counting the number of edges on the boundary of a face is simply done by starting on any dart of the face, and using β_1 to circulate around the boundary until the initial dart is reached again. To traverse the boundary of an f-pattern, we start from a 2-free dart of the pattern, and find the next 2-free dart on the boundary by circulating around the vertex at the tip of the current dart using $\beta_1 \circ \beta_2$ until another 2-free dart is found.

4.2.2. Flagged Version

Dart cycles are automorphic to themselves and replacements can therefore be performed in several ways. In the example of Fig. 3, the pattern applies on a face having 6 edges. The desired behavior would be to match dart b on the boundary of the face with dart 4 on the f-pattern. Topology alone is however not enough to distinguish dart b from any other dart of the cycle. Without additional control, dart 4 can therefore be matched with any dart on the boundary of the face leading to undesired replacement (although topologically valid) as shown on Fig. 4.

We therefore propose an optional flag system integrated to the signatures to enforce more control over how darts should be matched between the boundary of the f-pattern and the face. Such flags are marked with red crosses on the figures. When flags are set, flagged darts are always matched together (and non-flagged darts together).

To achieve this goal, we extend the f-signature from a single number to a *sequence* of numbers, each number being the length of a path starting from a flagged dart and ending to the dart before

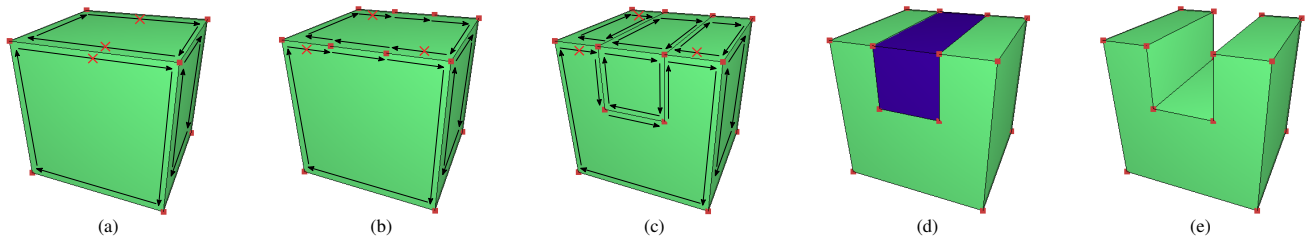


Fig. 2: Steps to create a slot on a cube with our formalism. (*a*) Starting cube. (*b*) Flagged edges are cut in three. (*c*) Three faces are updated with face query-replaces. (*d*) The initial volume is cut in two by one volume query-replace. (*e*) The new volume is removed to retrieve the initial volume with a slot.

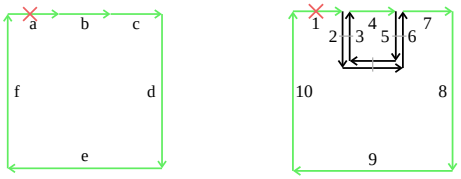


Fig. 3: Example of a face (left) matching an f-pattern (right). The sequence of darts (1,4,7,8,9,10) is the boundary of the f-pattern. The face and the pattern boundary both have 6 as their signatures

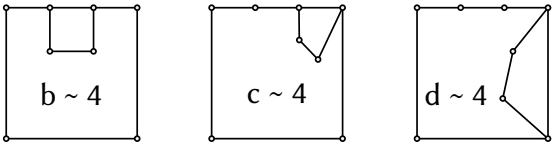


Fig. 4: Without flags, applying the f-pattern described in Fig. 3 can be done in several ways, since the boundary is a cycle automorphic to itself. Matching dart 4 with dart b in Fig. 3 is the desired behavior, but without flags, dart 4 could be matched with darts c or d, leading to undesired replacements. Adding flags and constraining flagged edges to be matched enforces the desired replacement.

the next flagged dart along the cycle. Signatures must however be independent of the starting dart. We therefore normalize the sequence using the smallest sequence according to the lexicographic order among all the sequences obtained from all flagged darts. When no dart of the face is flagged, the signature is (0,k) where k is the length of the dart cycle. This allows us to differentiate the case of Fig. 3 with (6) as a signature, and a hexagonal face without flags, with a signature of (0,6).

When a face f and an f-pattern pf have the same flagged signature, the border of pf has the same length as f since the sum of the integers in the sequence encodes that length. When flags are present, both starting darts d and d' associated with the signatures of f and pf are flagged and the sequence of numbers ensure that matched darts have the same state.

4.3. v-pattern

4.3.1. Definition

A v-pattern is a set of connected volumes having one boundary surface, like the example given in Fig. 5. If the boundary is not a surface, it cannot match the boundary of a single volume element of the input mesh.

The boundary of a v-pattern cannot be characterized by a single number as for an f-pattern. We use a variant of the signatures of Gosselin et al. [26] summarized in Section 2.4.

We start by labeling all the 3-free darts of the v-pattern. These form the boundary of the 3-map. Iteration over these darts is

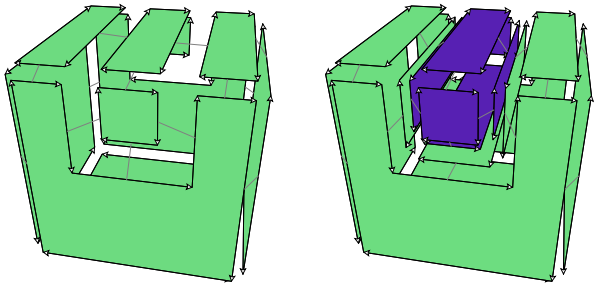


Fig. 5: Example of a volume (left) and a v-pattern (right).

performed using a variant of a breadth-first traversal of the 3-map (cf. Section 2.4) described in Algo. 2. The use of β_2 in the original BFS is replaced by $\beta_2 \circ \beta_3$ on non 3-free darts in order to circulate around boundary edges from one 3-free dart to the next without considering inner faces. The resulting labeling is uniquely defined for a given starting dart since the order of the traversed darts is unique.

Like for the definition of the signature of a 3-map, we encode the traversal of the boundary of a v-pattern from dart d as a word listing for each dart of the boundary (sorted by label) the label sequence of its adjacent darts (β_1 then β_2). Note however that since we only traverse the boundary, the β_2 neighbor of a dart is set to be the opposite dart *on the boundary* rather than in the adjacent face on the same volumetric cell. In Algo. 2 this corresponds to the additional loop in line 11.

Definition 6 (Word of a v-pattern). Given a v-pattern $vp = (D,\beta_1,\beta_2,\beta_3)$, and a labeling $l : D' \rightarrow \{1,\ldots,|D'|\}$ (with $D' = \{d \in D | d \text{ is 3-free}\}$). The word associated with (vp,l) is the sequence

$$W_{VP}(vp,l) =< w_1,\ldots,w_{2\cdot|D'|} >$$

such that $\forall k \in \{1,\ldots,|D'|\}$, $w_{2\cdot k} = l(\beta_1(d_k))$, and $w_{2\cdot k+1} = l(d'')$, where d_k is the dart labeled with k, i.e., $d_k = l^{-1}(k)$, and $d'' = (\beta_2 \circ \beta_3)^j(\beta_2(d_k))$, with j the smallest positive integer such that d'' is 3-free.

We denote by $W_{VP}(vp,d)$ the word associated with v-pattern vp, for the labeling obtained by Algo. 2 starting from dart d. The v-signature of vp, $S_{VP}(vp)$ is the smallest word $W_{VP}(vp,d)$ for each 3-free dart d, with respect to the lexicographical order.

Signatures for volumetric cells of the input mesh are computed in a similar way, the only difference being that there is no internal face and thus there is no need to use $\beta_2 \circ \beta_3$ to jump over inner faces. The algorithm iterating through all the darts of the volume is thus the same as Algo. 2, removing lines 11 to 12. Similarly,

Algorithm 2: Label the border of a v-pattern

Input: $vp = (D, \beta_1, \beta_2, \beta_3)$: a v-pattern;
$d \in D$: a 3-free dart.
Result: Labels for all darts of the border of vp.

let Q be an empty queue;
add d at the end of Q;
let `label` be an empty map associating an index to a dart ;
`label`$[d] \leftarrow 1$; $l \leftarrow 2$;
while Q *is not empty* **do**
  remove d' from the head of Q;
  `// circulate around the face of` d'
  if $\beta_1(d')$ *is not mapped to an index in* `label` **then**
    `label`$[\beta_1(d')] \leftarrow l$; $l \leftarrow l + 1$;
    add $\beta_1(d')$ at the end of Q;
  `// find the adjacent boundary face at` d'
  $d'' \leftarrow \beta_2(d')$;
  while d'' *is not 3-free* **do**
    $d'' \leftarrow \beta_2 \circ \beta_3(d'')$;
  if d'' *is not mapped to an index in* `label` **then**
    `label`$[d''] \leftarrow l$; $l \leftarrow l + 1$;
    add d'' at the end of Q;
return `label`;

Algorithm 3: Replace the interior of a face by the interior of a compatible f-pattern

Input: $M = (D, \beta_1, \beta_2, \beta_3)$: a target 3-map;
$fp = (D', \beta'_1, \beta'_2, \beta'_3)$: an f-pattern;
$f : D' \to D$: a map isomorphism from the boundary of D' to a subset of D.
Result: the interior darts of fp are inserted and connected in D in the matching face.

foreach $d' \in D'$ **do**
  if d' *is 2-free* **then**
    `// d' is a boundary dart`
    if $\beta'_1(d')$ *is not 2-free* **then**
      `// its next dart is an interior dart`
      `// 1-sew the matching dart in D to it`
      1-sew$(f(d'), \beta'_1(d'))$;
      $d'' \leftarrow d'$;
      do
        $d'' \leftarrow \beta'_1 \circ \beta'_2(d'')$;
      while $\beta_1(d'')$ *is not 2-free*;
      `// d'' is 1-sewn to the next 2-free dart`
      1-sew$(d'', \beta_1(f(d')))$;
  else
    `// d' is an interior dart`
    $D \leftarrow D \cup \{d'\}$

the definition of the word of a volume W_V is the same as that provided in Def. 6 but using $w_{2\cdot k+1} = l(\beta_2(d_k))$.

Computing such signatures amounts to computing the map signature of an imaginary 2-map coating the boundary of the pattern or the volumetric cell. It therefore follows from Gosselin et al. [26] that when the signatures of the v-pattern and the input volumetric cell match, they are isomorphic. The matched input volumetric cell can therefore be replaced by the interior of the pattern without creating topological issues.

4.3.2. Flagged Version

Like for f-patterns, darts can be optionally flagged on a v-pattern and on the input mesh to restrict set of darts than can be matched, providing more control to the user on the possible matches. The flags are integrated in the signatures by modifying the definition of words for a v-patterns and a volume. A bit field is appended at the end of the signature with one bit per dart. The k^{th} bit indicates whether d_k is flagged.

Without the appended bit field, a signature is a regular word associated with a traversal of the map. Two maps with the same word are therefore isomorphic. Matching bit fields along with the words ensures that darts with identical labels have identical flags. Therefore, patterns will only be matched with input cells exhibiting compatible flags.

5. Replace

Given one target t input cell and a pattern p with a matching signature, the replace operation rewrites the interior of the target by the interior of the pattern. A matching signature ensures that the border of the pattern is isomorphic to that of the target: a map isomorphism f maps every dart in the boundary of the target cell to the dart on the boundary of the pattern with the same label. From the initial dart of the traversal producing the signatures, the input cell and the pattern boundaries can be traversed simultaneously to build this mapping.

Pattern replacement requires addressing two issues: (1) updating of the mesh connectivity and partitioning the target into cells; (2) specifying the positions of the interior vertices of the pattern from the shape of the boundary of the target boundary vertices.

5.1. Mesh Connectivity

5.1.1. f-pattern

Algorithm 3 gives the method to replace a face (given by one of its darts) by a compatible f-pattern. This algorithm links the interior darts of the pattern with the darts of the face. Note that most of the interior darts of the pattern can be inserted as is without changing their connections to other darts. The only modified darts are along the boundary where multiple new inner faces share a vertex. For each such vertex, two darts need to be updated to take into account the fan of inner edges at the vertex. This is illustrated in Fig. 3, the isomorphism matches darts (a,b,c,d,e,f) respectively with darts (1,4,7,8,9,10). Dart 1 is 2-free and $\beta'_1(1)$ is not 2-free, meaning that the vertex at the tip of 1 is shared by several faces. The dart a of the target – matched to 1 in the pattern – therefore has to be 1-sewn with 2, and the inner dart 3 needs to be 1-sewn with dart b (the next boundary dart). These modifications insert locally the edge {2,3} of the pattern. In the algorithm, $\beta'_1 \circ \beta'_2$ is used to circulate around a boundary vertex shared by multiple faces. Performing such insertions for all interior edges will integrate the interior of the pattern to replace the target face. All the 2-sewn darts of the pattern are added into M.

If the target face is between two different volumes, each dart of the face is 3-sewn to another dart on the opposite face of the

Algorithm 4: Replace the interior of a volume by the interior of a compatible v-pattern

Input: $M = (D, \beta_1, \beta_2, \beta_3)$: a 3-map;
$vp = (D', \beta'_1, \beta'_2, \beta'_3)$: a v-pattern;
$f : D' \to D$: a map isomorphism from the boundary of D' to a subset of D.

Result: the interior darts of vp are inserted and connected in D in the matching volume.

```
foreach d' ∈ D' do
    if d' is 3-free then
        if β'2(d') is not 3-free then
            2-sew(f(d'), β'2(d'));
            2-sew(β'2(d'), f(d'));
    else D ← D ∪ {d'};
```

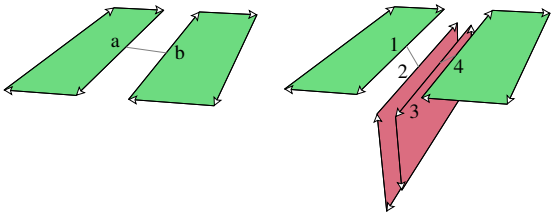


Fig. 6: Zoom on the two upper faces of the volume shown in Fig. 5 (left), and on the two faces that are inserted locally (right, in red).

neighboring volume. In such a case, the replacement has to be done similarly for the two opposite faces, but with reverse orientations (swapping β_0 and β_1) on one side to preserve the correct orientation of the volumes around the face. Each inner dart in the replacement of one of the faces is 3-sewn with its reversed counterpart in the replacement of the opposite face.

5.1.2. *v-pattern*

Algorithm 4 details the replacement of the interior of a volume by a compatible v-pattern. Interior faces are inserted by using 2-sew operations. An interior dart of the pattern 2-sewn to a dart d' on the boundary of the pattern has to be 2-sewn to the dart matching d' in the target map. Since β'_2 is an involution, such edges are found using β'_2 on the edges of the boundary of the pattern, identified as the 3-free darts in the pattern. The involution also requires 2-sewing the pattern and the target darts in both ways. Fig. 6 zooms on the two upper faces of the volume shown in Fig. 5 to illustrate the required 2-sewing operations. One internal face (drawn in red in Fig. 6 right) must be inserted between two faces. When considering the 3-free dart 1 in the pattern, dart $a = f(1)$ will be 2-sewn with dart $2 = \beta'_2(1)$, and vice versa. A second modification will be done when considering 3-free dart 4: dart $b = f(4)$ will be 2-sewn with dart $3 = \beta'_2(4)$ and vice versa.

5.2. *Geometry*

In the previous section, we showed how the interior of a face is replaced by the interior of a pattern, but only for the mesh connectivity part. The positions of the *external vertices* on the border of the pattern (drawn in blue in Fig. 7) are constrained to those of their corresponding vertices on the target face. When a pattern has *internal vertices* (drawn in red in Fig. 7), the positions of these vertices must be interpolated as a function of the positions of the vertices of the target face.

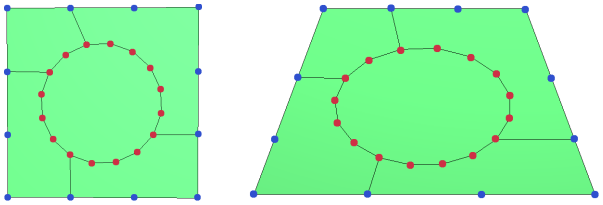


Fig. 7: Example of the replacement of a face with 12 edges by an f-pattern containing an inner circle. (left) The pattern boundary is provided with a squared shape. (right) The target face has a trapezoidal shape.

The main idea is to compute generalized barycentric coordinates for the internal vertices of the pattern as a function of the external vertices, for each pattern. When a pattern is used, positions of the internal vertices are interpolated using these barycentric coordinates and the positions of the external vertices of the target. Such barycentric coordinates can be defined in 2D [27] for f-patterns and 3D [28] for v-patterns. Fig. 7 illustrates the process for an f-pattern.

6. Properties and Strengths

6.1. *Genericity and Simplicity*

The operation is *generic*: any face/volume can be used as a pattern and as a target. This enables many different uses of this operation for different mesh transformation methods. In addition, the query-replace operation is *easy to use*: it is sufficient to provide specific sets of f-patterns and v-patterns to define a new modification operation. These patterns can be made by a 3D modeler, and in this case there is no code to write to define a new operation. These various advantages are illustrated in our experiments detailed in Section 7.

6.2. *Topological Validity*

A 3-map represents a 3D quasi-manifold (combinatorial equivalent of manifold, see [20] for all definitions), and thus is by definition a *topologically valid 3D object*. The query-replace operation takes two 3-maps as input: a target $M = (D, \beta_1, \beta_2, \beta_3)$ being the object to transform, and a pattern $p = (D', \beta'_1, \beta'_2, \beta'_3)$. The target is modified using either Algo. 3 for an f-pattern or Algo. 4 for a v-pattern. Proving that these two algorithms produce a valid 3-map as output implies the topological validity of the query-replace operation: the modified object is a valid 3D quasi-manifold.

Theorem 2. *The 3-map resulting from the replacement of an input volume element with the interior of a v-pattern is topologically valid: β_1 is a permutation and β_2, β_3 and $\beta_1 \circ \beta_3$ are involutions.*

Proof. Let M be the 3-map undergoing the query-replace operation with pattern p. There are two modifications of M: (1) non 3-free darts of p are copied into M; (2) some darts of the target are 2-sewn with some darts of the pattern and vice versa. Let d be a dart in the resulting 3-map.

1. If d is an original dart of M, then $\beta_1(d)$ and $\beta_3(d)$ are also original darts because β_1 and β_3 are not modified by Algo. 4;

2. If d is a dart copied from p, then from Algo. 4 d was not 3-free in the pattern. It therefore belongs to a face in the interior of the pattern, and $\beta_1(d)$ as well. $\beta_1(d)$ was therefore not 3-free in p, and is thus also present in M after the modification. Since d is not 3-free and β_3 is an involution, $\beta_3(d)$ is not 3-free as well and was also copied to M. As a conclusion, β_1, β_3 and $\beta_1 \circ \beta_3$ are still involutions after the modification since it was the case in p, they were not modified, and all the concerned darts were copied to M.

β_1 and β_3 are therefore preserved, β_1 is still a permutation, β_3 and $\beta_1 \circ \beta_3$ are still involutions. To prove that β_2 is an involution there are several cases.

1. If $\beta_2(d)$ was modified by the algorithm, then the modification in Algo. 4 is done in both ways which enforces that β_2 is an involution on these darts;

2. If $\beta_2(d)$ is not modified:

   - If d is an interior edge of the pattern, or an edge of M which is not related to the replacement, the β_2 neighbors are not modified as well, and are present in the modified map, the involution is thus preserved;
   - If d is a dart on the boundary of the volume replaced by the pattern, it is matched to some 3-free dart d' on the boundary of the pattern. In Algo. 4, the fact that $\beta_2(d)$ is not modified means that $d'' = \beta'_2(d')$ is 3-free as well. The signature matching implies a map isomorphism, and therefore d'' is matched with $\beta_2(d)$. Since d'' and d' are 3-free $\beta_2(\beta_2(d))$ will not be modified.

β_2 is therefore an involution. □

A similar proof can be done for f-patterns for Algo. 3. The key point of this second proof is the fact that the same modification is applied similarly, but in reverse orientation, on both half-faces for non 3-free darts.

6.3. Computation Time Complexity

The use of signatures gives us a very good computation time complexity for the query replace operation, particularly when the number of patterns is big. Indeed, we can start by computing the signatures of all the patterns, and store them in an associative array using the signatures as keys, and the patterns as values. Then, for each face or volume element of the target (depending on the type of patterns), computing its signature and querying the associative array directly provides a compatible pattern if any.

The complexity of the access is in amortized constant time if a hash-table is used. The complexity of the replace algorithms is linear in the number of darts of the target n_t. The complexity to compute the signature of a target element is quadratic in the number of darts forming the element: from each dart, a linear traversal is performed (cf. Section 2.4). This gives an overall amortized complexity in n_t^2, which does not depend on the number of patterns.

The same operation can be done without using signatures. In this case, all the patterns have to be iterated over, and for each one an isomorphism test is performed with the target element. The complexity of the isomorphism test is also quadratic in the number of darts of the target (cf. Section 2.3). Applying this test for each pattern however gives an overall complexity in $\#p \times n_t^2$, with $\#p$ the number of patterns. When this number is small, the difference between the computation times of the two methods is also small. It becomes significant when the number of patters increases, as we will see in our experiments.

6.4. Proof of the exhaustiveness of a set of configurations

Signatures are a good tool to prove the exhaustiveness of a set of patterns. This problem is complicated, as illustrated by the well known topological errors of the initial marching cube method [29] and the numerous following papers proposing corrections and additional cases.

Given a set of patterns, two important questions are: (1) are there two patterns with an isomorphic boundary? (2) do the patterns cover all the possible cases? The first question can be addressed with an associative array of signatures. For each pattern its signature is computed and queried in the array. If it already belongs to the array, another pattern has an isomorphic boundary. Otherwise, the signature can be added in the array. The second question can be tackled by first generating all possible configurations of the studied algorithm. For each configuration, its signature is computed and queried against the pattern signature array. If no pattern matches, a pattern is missing and the corresponding configuration is not handled by the set of patterns.

Note that depending on the use cases, it can be on purpose that two patterns have isomorphic boundaries. Even in such cases, the signatures allow a user to detect and control these configurations. When a target is compatible with several patterns, additional information is required in order to find which one must be used (based on geometry, values associated with vertices...).

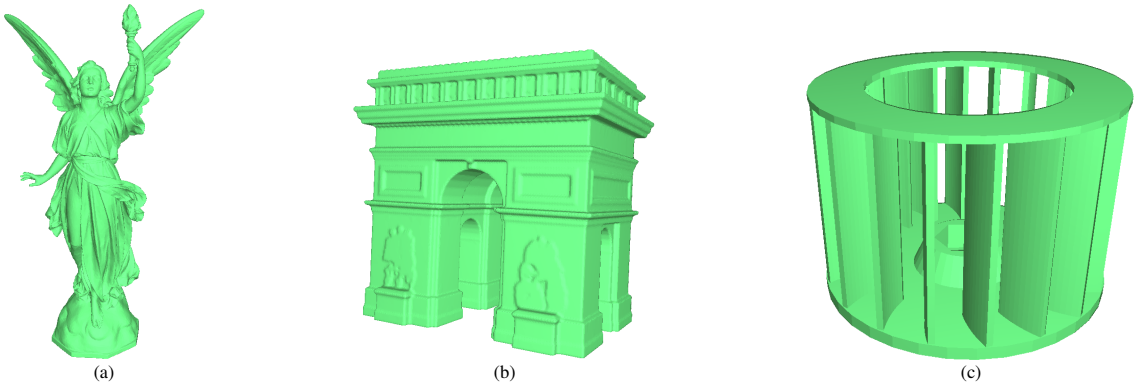
7. Experiments

We have implemented our query/replace operation, both for faces and volumes, based on the CGAL implementation of combinatorial maps [30] and the linear cell complex additional layer [31], which represents the geometry. All experiments were run on an Intel®i7-1165G7 GPU @ 2.80GHz with 32 GB RAM. The code and the data to generate the results are available on the following repository https://gitlab.liris.cnrs.fr/gdamiand/3d-query-replace.

In our experiments, we used the 3 meshes shown in Fig. 8 as input, coming from the Thingi10K 3D mesh dataset [32].

We implemented four different mesh transformation methods based on our query-replace operation. In each case, a preliminary step inserts some vertices on some/all edges on a mesh. Depending on the application, a post-processing step removes or merges some cells in order to obtain the final mesh. In all cases, the core of the transformation used the query-replace for f-patterns then for v-patterns. These patterns were created using the Moka 3D modeler [33], except for the conforming mesh experiment where they were imported from existing vtk files.

For all these experiments, flags were automatically set on input darts and darts of f-patterns and v-patterns having a corner as origin (the 8 vertices of a cube for instance, but not the vertices in the middle of its edges or faces). We use the convention that when a dart is split, the flag is transferred to the output dart with the same origin. Using flags in these experiments therefore required no manual user intervention.



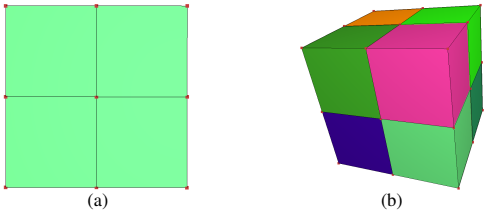
(a) (b) (c)

Fig. 8: The 3 meshes used in our experiments (the names are the ones used in Thingi10K). (*a*) `S1` is "Stanford's Lucy", 24,975 vertices, 74,919 edges and 49,946 faces. (*b*) `S2` is "Famous Paris buildings", 174,066 vertices, 522,192 edges and 348,128 faces. (*c*) `S3` is "Involute Blower", 7,824 vertices, 1,268 edges and 2,608 faces.

Algorithm 5: Hexahedral Subdivision

Input: S: a 3D surface;
`lmax`: a maximal subdivision level.
Output: M: a hexadral approximation of S at level `lmax`.

```
Create one hexahedron in M having the geometry of the
bounding box of S;
for i ← 1 to lmax do
    H ← all hexahedra to subdivide of M;
    Split all edges of H in two;
    Query-replace all faces of H using pattern Fig. 9a;
    Query-replace all volumes of H using pattern Fig. 9b;
return M
```



(a) (b)

Fig. 9: (*a*) f-pattern to transform a square, with each edge cut in two, into 2×2 squares. (*b*) v-pattern to transform a cube, with each face cut in 4, into $2 \times 2 \times 2$ cubes.

7.1. Hexahedral Subdivision

Our first experiment implements a method to approximate a 3D surface by a set of hexahedra at a desired resolution. The main principle of the method is given in Algo. 5.

Starting from a 3D surface and a maximal subdivision level, the method first determines the set of all hexahedra to subdivide H. These hexahedra are those intersected by the surface, and those having more than one subdivision level of difference with respect to at least one of their neighbors. All the edges of these hexahedra are then split in two (by inserting a new vertex in the middle of the edge). The rest of the method only consists in using twice our query-replace operation: first for all faces of H with four subdivided edges using the f-pattern shown in Fig. 9a, which subdivides such a face into 4 squares; second for all volumes of H using the v-pattern shown in Fig. 9b, which subdivides a volume into 8 hexahedra.

In this algorithm, we only use one f-pattern and one v-pattern.

Mesh	#volumes	#fqr	#vqr	time-qr	time-direct
`S1`	116,865	88,031	23,103	9.04s	4.54s
`S2`	233,877	167,260	43,129	18.22s	11.40s
`S3`	279,604	203,436	54,741	39.05s	30.87s

Table 1: **Number of elements and timings for Hexahedral Subdivision** for the three meshes given in Fig. 8, with `lmax`= 7. #volumes is the number of volumes obtained at the end of the subdivision; #fqr is the number of query-replace of faces done, and #vqr the number of query-replace of volumes; time-qr is the total computation time of the method using the query-replace operations; time-direct is the total computation time of the same method using a direct hexahedral subdivision.

In addition, by construction, we know that each face of H is compatible with the f-pattern, and that each volume of H is compatible with the v-pattern. For this reason, we can avoid the query-part of the query-replace operation since we know it is always satisfied, and use only the replace part of the operation.

We can see in Table 1 the number of volumes obtained by this method for our three input surface meshes, using 7 as maximal subdivision level, and the number of query-replace used (for faces and for volumes). We compared the computation time of our method using Algo. 5 and the query-replace operations with a direct implementation of the hexahedral subdivision. The latter is obviously faster since it is optimized for the subdivision, creating only the necessary darts and updating only the minimal information. We observed an overhead of about 36% in average for the query-replace method, which is significant. The main advantage of the query-replace method is its simplicity: there is almost no code to develop and thus few risk of bugs comparing to the direct implementation. Another advantage is the simplicity to change the subdivision scheme, by only modifying the two patterns using a 3D modeler. Note that if the computation time is critical, query-replace method can be used as a validation tool to verify that the direct method produces the correct result.

Fig. 10 shows a partial representation of the result obtain for mesh `S1`. We did not draw all hexahedra in order to visualize the interior of the final map, showing hexahedra at different levels of subdivision.

7.2. Marching-Cubes

In this second experiment, we implemented a Marching-Cube method [29] aiming at extracting a 3D surface from a hexahedral mesh. This hexahedral mesh can be obtained from various inputs

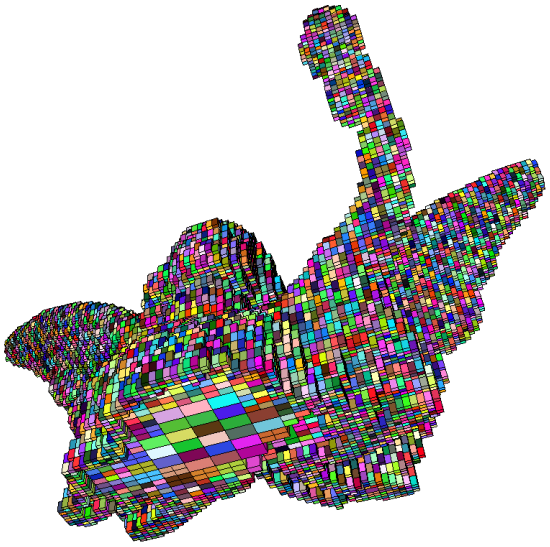


Fig. 10: Result of the hexahedral subdivision obtained for mesh `S1`, with `lmax`= 7 (partial representation to visualize the interior of the mesh with different levels of subdivision).

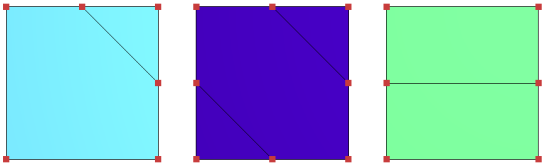


Fig. 11: The 3 f-patterns used for the marching cubes implementation.

(like for example a 3D scalar field or a signed distance function). The only requirement is to be able to test whether a vertex is inside or outside the object. The original paper defines 15 different configurations (exploiting rotations and symmetries), but it has topological inconsistencies (cracks appearing for ambiguous cases). Several papers proposed solutions to this problem, until obtaining 33 different configurations [34, 35], which has been proved to be complete.

Algorithm 6: Marching-Cube

Input: M: a 3D hexahedral mesh.
Result: Transform M into a 3D surface approximating the initial hexahedral mesh.

1 Split in two all edges of M connecting one vertex inside and one vertex outside;
2 Query-replace all faces of M using the 3 f-patterns Fig. 11;
3 Query-replace all volumes of M using the 29 v-patterns, 3 being drawn in Fig. 12;
4 Remove all faces incident to one old vertex.

Algorithm 6 formalizes our method to build a marching-cube surface from a 3D hexahedral mesh using our query-replace operation. First, a new vertex is inserted on each edge crossing the surface (i.e. one vertex is inside and the other one is outside, note that we consider as inside vertices *on* the surface). We then use the query-replace operation for all faces and all volumes.

There are 3 different possible cases for faces, and thus 3 different f-patterns shown in Fig. 11.

For v-patterns, we initially created the 15 original configurations [29] using a 3D modeler. We then used the validation method presented in Section 6.4 to progressively identify missing v-patterns: we generated all possible configurations for the inside and outside status of the 8 vertices of a cube, and for each one tested if an existing pattern matched. When it was not the case, we created a new corresponding pattern and went on with the next configuration. We obtained 29 v-patterns (3 of them being drawn in Fig. 12) which is the minimal set of required configurations in order to consider all possibles cases. We did not obtain the 33 different configurations of [34, 35] because we did not create configurations with isomorphic boundaries but different internal subdivisions.

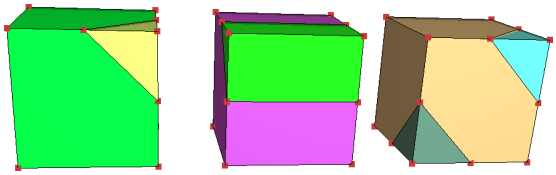


Fig. 12: The first 3 v-patterns (among the 29 total ones) used for the marching cube experiment.

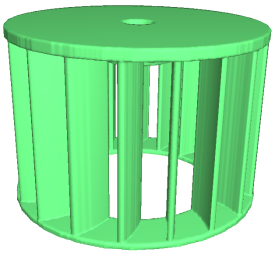


Fig. 13: Result of the marching cube method applied on the hexahedral mesh obtained from mesh `S3`, with `lmax`= 7.

After all the query-replace operations, the surface is extracted by removing all the faces having at least one initial vertex among their vertices. This is done in the last step of Algo. 6.

We applied this algorithm for three input hexahedral meshes (obtained from the three input surfaces of Fig. 8 with a maximal subdivision level equal to 7). The computation time of our method is 7.79s (resp. 15.76s and 23.34s) to generate triangular meshes with 121,809 triangles (resp. 208,855 and 260,004). Fig. 13 shows the results obtained from the `S3` case.

7.3. Conformal Mesh

The goal of this third experiment was to generate a conformal 3D mesh, using hexahedra, tetrahedra, prisms and pyramids unit elements. A mesh is conformal if the pairwise intersection of any two cells is either a single lower-dimensional cell or is empty. Visually, the mesh has no T-junctions. Jaillet and Lobos [36][1] constructed a set of 325 cases to transform a hexahedral mesh with at most one level of subdivision between adjacent cells into a conformal mesh. We therefore translated these cases into v-patterns and applied our query and replace method.

We first perform a query-replace on faces, and ensured with our exhaustiveness test that the five f-patterns given in Fig. 14

[1] Thanks to Fabrice Jaillet for providing us with the VTK files of the 325 patterns.

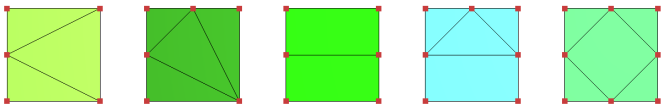


Fig. 14: The 5 f-patterns used for the conforming mesh experiment.

Algorithm 7: Conformal Mesh Generation

Input: M: a 3D hexahedral mesh.
Result: Transform M into a conformal mesh.
1 Query-replace all faces of M using the 5 f-patterns Fig. 14;
2 Query-replace all volumes of M using the 325 v-patterns, some of which are drawn in Fig. 15.

are sufficient to handle all cases. Once the faces subdivided according to these patterns, a second query-replace acts on all volumes using the 325 v-patterns, 3 of which are depicted in Fig. 15. The resulting mesh is a conformal mesh.

This experiment illustrates perfectly the great interest of our query-replace operation with a great number of cases (325 v-patterns). Manually implementing a test to identify the proper cases to apply would have been a tedious task prone to bugs in the developed code. With our method, no additional code is required. In addition, we proved the exhaustiveness of the set of 325 configurations using the method presented in Section 6.4.

We tested the process for the 3 hexahedral meshes obtained for our 3 surfaces with a maximal subdivision level equal to 7. The distribution volume types in the resulting mesh is provided in Table 2. The column time-qr provides the efficient computation time of our method: 6.59s to transform 257,642 hexahedra for the second mesh.

In a second experiment, we demonstrated the importance of using signatures to handle the size of the set of patterns. As a comparison, we used the isomorphism test directly to retrieve for each target its compatible pattern. This second method is much more expensive in terms of computation time, since the 325 patterns need to be tested in the worst case for every target hexahedron until a compatible one is found. This computation time is given in the time-qrns column in Table 2. It is in average 18 times slower than the method with signatures.

7.4. Mesh Transformation

In this last experiment, we implemented a mesh transformation method that transforms a tetrahedral mesh into a hexahedral one. Starting from a 3D surface, we used TetGen [37] to generate a 3D tetrahedral mesh. We then split each edge in two by inserting a vertex in the middle of the edge, and used our query-replace for each face, then for each volume, using the two patterns shown in Fig. 16. Fig. 17 shows (partially) the result of this transformation starting from mesh `S1`.

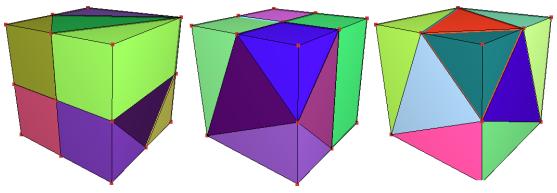


Fig. 15: 3 v-patterns (among the 325 total ones) used for the conformal mesh experiment.

Mesh	#h	#t	#pr	#py	time-qr	time-qrns
`S1`	108,754	41,159	6,659	58,935	2.18s	29.86s
`S2`	207,165	97,410	16,626	143,606	6.59s	131.9s
`S3`	272,164	37,772	4,972	46,908	2.54s	46.42s

Table 2: **Number of elements and timings for Conformal Mesh Generation** for the three hexahedral meshes obtained from the surfaces given in Fig. 8, with `lmax`= 7. #h is the number of hexahedra obtained at the end of the generation (#t tetrahedra, #pr prisms and #py pyramids); time-qr is the total computation time of the method using the query-replace operations; time-qrns is the total computation time of the same method using the query-replace operations but without using signatures.

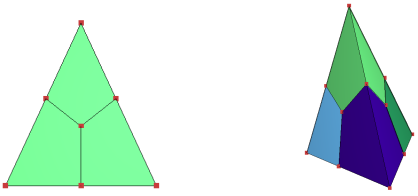


Fig. 16: (Left) f-pattern to transform a triangle, with each edge cut in two, into 3 quadrilaterals. (Right) v-pattern to transform a tetrahedron, with each face cut in 3, into 4 hexahedra.

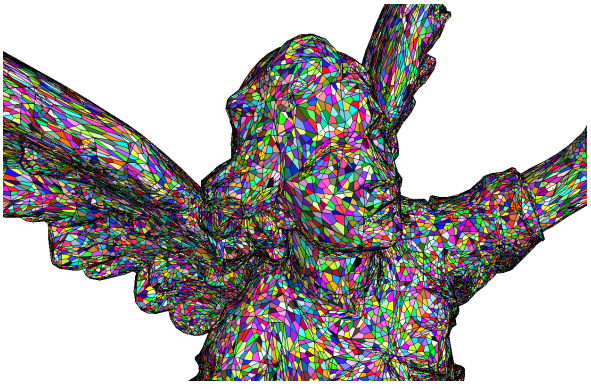


Fig. 17: Result of the transformation of a tetrahedral mesh into a hexahedral one for mesh `S1` (zoom).

8. Conclusion

In this paper, we defined a query-replace operation for faces and volumes of 3D meshes. Our operation is generic, simple to use, topologically valid, and can be used to prove the exhaustiveness of a set of patterns. Moreover, the query of one pattern from a set of patterns is fast thanks to the use of topological signatures. We provide the formal definition of the basic operation, along with a flagged version to augment the signatures and provide more control when necessary. The efficiency and versatility of our operation is demonstrated by several experiments reproducing existing mesh generation methods in our formalism and illustrating the efficiency of our solution.

Note that our appoach can be used for different data structures when it is possible to define and compute signatures and replace operations. This is for example possible for half-edge data structures, but obviously only for e-patterns and f-patterns since it is not possible to represent groups of volumes in this case.

This work opens many possible future work. First, our query and replace operation is currently limited to the replacement of single elements of the target mesh. This is fine to encode many mesh generation techniques, but other operations like edge contraction or flips cannot be expressed in our formalism. We therefore would like to be able to handle the replacement of portions of a mesh made of multiple elements. This would allow more possible transformations, like for example the method of [38] which regroups several tetrahedra into hexahedra. Secondly, we would like to extend this work to meshes with possibly 1-free and 2-free darts, allowing to consider surfaces with boundaries. Lastly, we would like to study if other operations could be defined by using a similar principle of rewriting rules to create new cells or to remove existing ones. The goal would be to define a generic language to describe mesh portions or graphs, in a way similar to that of regular expressions on one dimensional words. We would also like to explore the expressiveness of our approach for procedural generation of patterns and automatic addition of details to a rough initial structure.

References

[1] Pitzalis, L, Livesu, M, Cherchi, G, Gobbetti, E, Scateni, R. Generalized adaptive refinement for grid-based hexahedral meshing 2021;40(6).

[2] Corman, E, Crane, K. Symmetric moving frames. ACM Trans Graph 2019;38(4).

[3] Gao, X, Shen, H, Panozzo, D. Feature preserving octree-based hexahedral meshing. Computer Graphics Forum ????;38(5):135–149.

[4] Parallel hybrid mesh adaptation by refinement and coarsening. Graphical Models 2020;111:101084.

[5] Kaplan, CS. Introductory tiling theory for computer graphics. In: Introductory Tiling Theory for Computer Graphics. 2009,.

[6] Meekes, M, Vaxman, A. Unconventional patterns on surfaces. ACM Transactions on Graphics (TOG) 2021;40:1 – 16.

[7] Peytavie, A, Galin, E, Grosjean, J, Mérillou, S. Procedural Generation of Rock Piles Using Aperiodic Tiling. Computer Graphics Forum 2009;28(7):1801–1809.

[8] Stiny, G, Gips, J. 'shape grammars and the generative specification of painting and sculpture'. vol. 71. 1971, p. 1460–1465.

[9] Lindenmayer, A. Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. Journal of Theoretical Biology 1968;18(3):300–315.

[10] Tobler, RF, Maierhofer, S, Wilkie, A. Mesh-based parametrized l-systems and generalized subdivision for generating complex geometry. Int J Shape Model 2002;8:173–191.

[11] Marvie, JE, Perret, J, Bouatouch, K. Fl-system: A functional l-system for procedural geometric modeling. The Visual Computer 2005;21:329–339.

[12] Bohl, E, Terraz, O, Ghazanfarpour, D. Modeling fruits and their internal structure using parametric 3Gmap L-systems ????;.

[13] Belhaouari, H, Arnould, A, Le Gall, P, Bellet, T. Jerboa: A graph transformation library for topology-based geometric modeling. In: Giese, H, König, B, editors. Graph Transformation. Cham: Springer International Publishing. ISBN 978-3-319-09108-2; 2014, p. 269–284.

[14] Baumgart, B. A polyhedron representation for computer vision. In: Proc. of AFIPS National Computer Conference; vol. 44. 1975, p. 589–596.

[15] Muller, D, Preparata, F. Finding the intersection of two convex polyhedra. Theoretical Computer Science 1978;7(2):217–236.

[16] Weiler, K. Edge-based data structures for solid modelling in curved-surface environments. Computer Graphics and Applications 1985;5(1):21–40.

[17] Sieger, D, Botsch, M. Design, implementation, and evaluation of the surface_mesh data structure. In: Quadros, WR, editor. Proc. of 20th International Meshing Roundtable. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-24734-7; 2012, p. 533–550.

[18] Rossignac, J. 3D compression made simple: Edgebreaker with zipandwrap on a corner-table. In: Proc. of International Conference on Shape Modeling and Applications. 2001, p. 278–283.

[19] Lienhardt, P. N-Dimensional generalized combinatorial maps and cellular quasi-manifolds. Inte J of Computational Geometry and Applications 1994;4(3):275–324.

[20] Damiand, G, Lienhardt, P. Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing. A K Peters/CRC Press; 2014.

[21] Botsch, M, Kobbelt, L, Pauly, M, Alliez, P, Lévy, B. Polygon Mesh Processing. AK Peters; 2010.

[22] Damiand, G, De La Higuera, C, Janodet, JC, Samuel, E, Solnon, C. Polynomial algorithm for submap isomorphism: Application to searching patterns in images. In: Proc. of 7th Workshop on Graph-Based Representation in Pattern Recognition (GBR); vol. 5534 of *Lecture Notes in Computer Science*. Venice, Italy: Springer Berlin/Heidelberg; 2009, p. 102–112.

[23] Hopcroft, JE, Wong, JK. Linear time algorithm for isomorphism of planar graphs. In: STOC. ACM; 1974, p. 172–184.

[24] Cori, R. Un code pour les graphes planaires et ses applications. In: Astérisque; vol. 27. Paris, France: Soc. Math. de France; 1975,.

[25] Solnon, C, Damiand, G, de la Higuera, C, Janodet, JC. On the complexity of submap isomorphism and maximum common submap problems. Pattern Recognition (PR) 2015;48(2):302–316.

[26] Gosselin, S, Damiand, G, Solnon, C. Efficient search of combinatorial maps using signatures. Theoretical Computer Science (TCS) 2011;412(15):1392–1405.

[27] Hormann, K, Floater, MS. Mean value coordinates for arbitrary planar polygons. ACM Trans Graph 2006;25(4):1424–1441.

[28] Ju, T, Schaefer, S, Warren, J. Mean value coordinates for closed triangular meshes. New York, NY, USA: Association for Computing Machinery. ISBN 9781450378253; 2005,.

[29] Lorensen, WE, Cline, HE. In: Stone, MC, editor. SIGGRAPH. ????,.

[30] Damiand, G. Combinatorial maps. In: CGAL User and Reference Manual; 3.9 ed. 2011,http://www.cgal.org/Pkg/CombinatorialMaps.

[31] Damiand, G. Linear Cell Complex. In: CGAL User and Reference Manual; 4.0 ed. 2012,http://www.cgal.org/Pkg/LinearCellComplex.

[32] Zhou, Q, Jacobson, A. Thingi10k: A dataset of 10,000 3d-printing models. arXiv preprint arXiv:160504797 2016;.

[33] Vidil, F, Damiand, G, Dexet-Guiard, M, Guiard, N, Ledoux, F, Fousse, A, et al. Moka: 3d topological modeler. 2002. http://moka-modeller.sourceforge.net/.

[34] Chernyaev, E. Marching cubes 33: Construction of topologically correct isosurfaces. Tech. Rep.; 1995.

[35] Lewiner, T, Lopes, H, Vieira, AW, Tavares, G. Efficient implementation of marching cubes' cases with topological guarantees. Journal of Graphics Tools 2003;8(2):1–15.

[36] Jaillet, F, Lobos, C. Fast Quadtree/Octree adaptive meshing and remeshing with linear mixed elements. Engineering with Computers 2021;.

[37] Si, H. Tetgen, a delaunay-based quality tetrahedral mesh generator 2015;41(2).

[38] Identifying combinations of tetrahedra into hexahedra: A vertex based strategy. Computer-Aided Design 2018;105:1–10.

Appendix

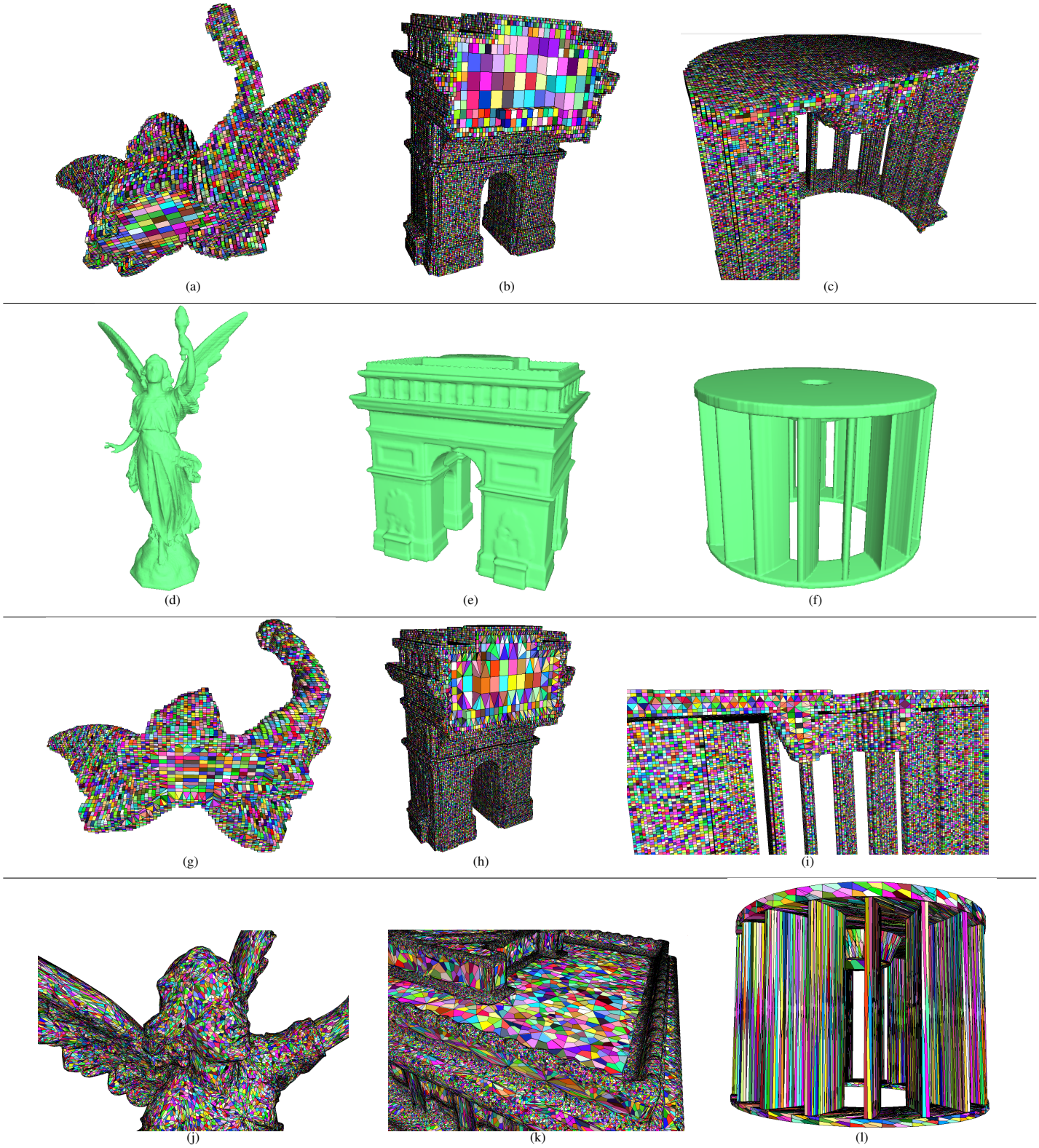


Fig. 18: **Visual results of all methods applied to the three meshes used in the experiments in Section 7**. First line: Hexahedra Subdivision; Second line: Marching-Cubes; Third line: Conformal Mesh; Fourth line: Mesh Transformation. Results for Marching-cubes are surfacic meshes; for other methods they are volumic meshes, drawn sometimes only partially to see the interior.