A Generic Query-Modify Framework for Volumetric Mesh Processing

Guillaume Damiand[a,*], Vincent Nivoliers[a] and Romain Pascual[b]

[a] *CNRS, UCBL, INSA Lyon, LIRIS, UMR5205, F-69622, Villeurbanne, France*
[b] *MICS, Centralesupelec, Université Paris-Saclay, 9 rue Joliot Curie, 91192, Gif-sur-Yvette Cedex, France*

ARTICLE INFO

Keywords:
Combinatorial maps
Mesh processing
Query-modify framework
Tetrahedral recombination

ABSTRACT

We introduce a query-modify framework for automating volumetric mesh processing. Our method enables flexible and efficient modifications of geometric structures composed of multiple volumes with minimal user-implemented code. Modifications are provided as rules consisting of a query mesh and a target mesh representing structural information to be extracted and replaced. The rules enable both localized queries to be matched with a portion of an input mesh and targeted modifications on the matched portion of the input mesh. Our approach generalizes standard mesh manipulations and adapts to various applications, including geometric modeling, remeshing, and topology-aware transformations. We showcase our framework on several use cases, including the first complete implementation of a tetrahedral recombination method based on 171 cases, exhaustively classifying all possible recombinations. Our framework allows for arbitrarily connected collections of volumes as queries, enabling automated and application-driven mesh modifications.

1. Introduction

Many mesh processing algorithms follow a common workflow: first, identify specific portions of the mesh and then locally rewrite its combinatorics by creating or removing vertices, edges, or faces. A simple example is mesh decimation, where adjacent triangle pairs are identified and removed, and the resulting disconnected edges are reconnected. Implementing such a procedure is straightforward when the identified portions are simple, such as detecting pairs of adjacent triangles. However, more complex procedures might involve many cases, making manual identification tedious and error-prone. For instance, converting tetrahedral meshes into hexahedral ones requires combining neighboring tetrahedra into hexahedra, which means knowing all possible groups of tetrahedra that form a hexahedron. Meshkat and Talmor [31] identified six ways to decompose a cube into tetrahedra, leading to the design of an ad hoc procedure that automatically generates search trees to find matches automatically. Sokolov et al. [38] later extended this to ten cases to account for numerical precision. More recently, Pellerin et al. [35] demonstrated that a generic hexahedron (rather than a cube) admits 174 decompositions into tetrahedra, of which 171 have a valid geometric realization. However, no algorithm currently exploits these 171 cases for transforming tetrahedral meshes into hexahedral or hex-dominant meshes.

Abstractly, the local rewriting of the mesh combinatorics can be described as a two-step process. First, a query is performed on the *input* mesh to identify relevant substructures, and then a modification is applied to transform the matched pattern. In this work, we propose a generic framework for query-based mesh operations relying on recording and reproducing traversals of combinatorial maps. Our method relies on a volumetric query mesh encoded as a combinatorial map called the *query* and the associated modifications encoded in a second map called the *target*. Arbitrary query and target patterns provide a rich and flexible solution that can formally be described via graph rewriting [14, 21]. However, expressing an arbitrary transformation as a query-target pair requires specifying a precise mapping of all parts of both patterns from a so-called shared interface. Additionally, when designing a dedicated framework, rather than relying on general-purpose graph rewriting techniques or tools, one can leverage domain-specific properties, i.e., properties of the graphs being modified, to improve efficiency.

This work identifies three specific restrictions that cover a wide range of standard operations in geometric modeling. These restrictions assume that query and target patterns are connected combinatorial maps. The first, called query-extend, exploits query patterns included in the target patterns to extend the current object. The second, query-delete, is symmetric and assumes the target pattern to be included in the query pattern to delete parts of the object. The third, query-replace, requires both patterns to have isomorphic boundaries, enabling replacement of the interior of the query pattern. It directly extends the query-replace framework of Damiand and Nivoliers [12] by supporting multi-cell queries.

These three possibilities build upon the core idea of identifying a pattern (the query) for modification. Besides allowing a shared algorithm for performing the query and, down the line, a shared implementation, it also streamlines the specification of modeling operation for the user that only needs to provide this pattern. As a fallback for cases that cannot be easily expressed using these three approaches, we introduce the query-apply operation, which allows custom code to be written.

This paper defines the four query-modify operations alongside their corresponding algorithms. These operations

*Corresponding author
guillaume.damiand@cnrs.fr (G. Damiand); vincent.nivoliers@univ-lyon1.fr (V. Nivoliers); romain.pascual@centralesupelec.fr (R. Pascual)
ORCID(s): 0000-0003-1580-5517 (G. Damiand); 0000-0001-5242-1585 (V. Nivoliers); 0000-0003-1282-1933 (R. Pascual)

constitute our first main contribution: a new generic query-modify (Q& M) framework that handles connected sets of volumes as a query or target. We show four experiments to illustrate various uses of our method for computer graphics and mesh processing. The first and main application is the first implementation of a tetrahedral recombination method exhaustively using all 171 geometrically valid decompositions of a hexahedron into tetrahedra [35]. This application is our second main contribution. The second and third experiments show how to use our framework to modify existing meshes: the second edits a volumetric model of a house built from an IFC file, and the last modifies two surface meshes imported from Blender. The final experiment is a toy example that generates a terrain mesh with small houses, illustrating the flexibility of the query-apply operation.

The paper is organized as follows. Section 2 discusses related work on procedural modeling, graph transformations, and tetrahedral recombination. Section 3 introduces the definitions needed for this work, focusing on combinatorial maps, isomorphisms, and the previous query-replace method. Section 4 provides a high-level overview of our approach, which is then detailed in section 5 for the query part and section 6 for the modification part. Section 7 presents our four applications, while section 9 concludes and gives directions for future work.

2. Related work

2.1. Procedural modeling

Automatic mesh generation and editing have been extensively studied, particularly in procedural generation. This process can be seen as a rewriting system that transforms an input mesh using predefined rules. Several frameworks express such rules, such as L-systems [28] and shape grammars [40]. L-systems use formal grammars to generate sequences of symbols, or words, interpreted as geometric objects [37, II.5 and III.6]. They have been widely used for modeling plants [36] and urban landscapes [29]. Shape grammars, introduced in [40], operate directly on geometric shapes by recursively applying transformation rules. Initially developed for paintings and sculptures [40], they have been widely used in architecture (e.g., CGA shape [32]) and computer-aided engineering [19]. For a broader overview, see Cagan's retrospective [6]. L-systems and shape grammars excel at generating data from scratch but require internal access to the encoding of both rules and objects to edit the ruleset. Thus, designing new rules to be applied to existing data is difficult as it means producing the matching words from generic geometric objects, essentially reverse-engineering a procedural representation from raw meshes. Moreover, their high-level rule formulation abstracts the low-level description of the modification. As a result, each rule has its own implementation. In contrast, our approach directly manipulates objects using example-based rules defined as small mesh fragments. Users specify transformations visually – by providing before and after patterns – without coding or accessing the internal data representations. This no-code workflow lowers the entry barrier and supports rapid prototyping, making it well-suited for mesh editing tasks that go beyond pure procedural generation.

2.2. Graph transformations

We represent geometric objects using combinatorial maps, introduced by Lienhardt [27], which extend the half-edge data structure [41] to arbitrary dimensions. Combinatorial maps can be interpreted as graphs, enabling Pascual et al. [33] to define modeling operations via graph transformations, ensuring the topological correctness of the resulting objects. Arnould et al. [1] extended this formalism to handle attributes such as vertex coordinates, allowing modifications to affect both the mesh geometry and its combinatorics. The Jerboa library [3] implements these techniques, offering generic rules applicable to existing data. Designing new modeling operations in Jerboa still requires some effort.

Querying a geometric object to locate patterns relates to the graph isomorphism problem, which is tractable for combinatorial maps. Isomorphism can be solved by checking for identical words [17], a method applied in [12] for query-replace operations. Damiand and Nivoliers [12] introduced the notion of signature as a speed-up technique. A signature is a special word, i.e., a sequence of symbols, that unequivocally encodes the topology of a cell. Thus, using signatures inherently limits queries to a single cell: a single volume, face, or edge. Generalizing their method to arbitrary queries is nontrivial due to combinatorial explosion. Our method removes signatures and records query traversals reproduced on the input to identify matching regions.

2.3. Tetrahedra recombination

We illustrate our method in hex-dominant meshing, where a classical pipeline involves laying out vertices on a grid pattern aligned with a predefined direction field, connecting them with Delaunay tetrahedra, and merging adjacent tetrahedra into hexahedra [31]. To identify such combinations, Meshkat and Talmor [31] developed an automatic procedure that checks whether a set of tetrahedra matches a pattern by enumerating traversals of the graph of adjacent tetrahedra for six identified cases using two operations. `LINK` merges tetrahedra and `QUAD` merges tetrahedral facets into quads. Sokolov et al. [38] extended this to ten cases by accounting for slivers in Delaunay tetrahedra. Pellerin et al. [35] showed that up to 171 different tetrahedral combinations can form a hexahedron and proposed a method that selects merges using a maximal independent set on a constraint graph. Still, the method for identifying the matching set of tetrahedra is based on a dedicated algorithm that does not exploit the full 171 cases by lack of a generic framework to express how to find these cases. Our framework solves this problem by providing a generic way to express the query and replace operations from user-specified patterns. While this approach is dedicated to merging tetrahedra, it already contains the idea of reproducing a recorded traversal. We use combinatorial maps rather than a set of tetrahedra with face adjacencies to obtain finer control over traversal order, independent of how adjacent tetrahedra are retrieved.

3. Foundations

In this section, we introduce the model of combinatorial maps that we use to represent objects.

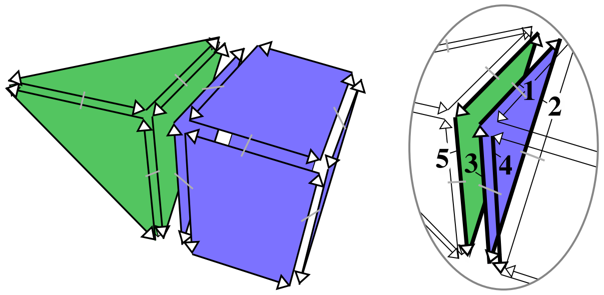
3.1. Why 3D combinatorial maps?

Our approach processes the combinatorial description of the topology of a volumetric object made of polyhedral cells. Among representations for such objects, combinatorial maps are lightweight data structures that fully encode the topological relations between the cells without restriction to specific complexes. The motivation for 3-maps instead of a more classical mesh data structure is threefold: first, it is ordered, allowing the definition of topological words (see defs. 4 and 5); second, it allows handling volumetric meshes with arbitrary topology but explicit relations between the volumes; and third, it encodes the topological validity of the mesh such that the topological soundness of Q&M framework can be assessed by showing that operations transform valid 3-map into valid 3-map. Essentially, performing queries and replacements on combinatorial maps allows for a precise description of the topological parts to be modified and ensures that the operation does not introduce any topological error.

The core context of a *topological word* introduced in this work can be defined on any *ordered* data structure. The order is needed as it provides a unique way to iterate through elements, immediately ruling out adjacency- or incidence-list-based structures. Therefore, the half-edge data structure would also support the definition of words and the global method described here, but only in 2D. Indeed, half-edge data structures describe isolated volumes, meaning that a connected set of volumes cannot be queried. Actually, 3-maps are simply a generalization of half-edges that encode volumetric relations: β_0 is *previous*, β_1 is *next*, and β_2 is *opposite*, while β_3 is a new relation providing a volumetric *opposite*.

The volumetric aspect of 3-map is needed to model buildings with multiple connected volumes. Even in the case of surface meshes, transformations may involve intermediate configurations with non-manifold faces, which 3-maps can handle directly. Compared to tetrahedral-based data structures, 3-maps can represent and combine cells of arbitrary topology, which, for instance, allows converting a tetrahedral mesh into a hexahedral one within the same data structure with intermediate states where the hexahedra are triangulated and therefore, no longer real hexahedra.

Another main interest of using 3-maps is to guarantee the topological soundness of our approach. Indeed, soundness follows from the same arguments as in [33] since our method relies on query and target maps with isomorphic functions and preserved regions. In practice, all modifications are performed using sewing, unsewing, removal, and the query-replace operation of [12]. These high-level primitives were already proven sound, i.e., they preserve the topological validity of a mesh, ensuring that the three target-based modification operations (and any query-apply operation based on them) are sound.



(a) (b)

Figure 1: (a) An example of a 3-map showing a triangular prism sharing a face with a tetrahedron. The tetrahedron contains 12 darts, 3 per face and the triangular prism contains 18 darts, 3 per triangular face and 4 per quad face. (b) Zoom in on the face between the two volumes. The orientation of the arrows explicit the β_1 function: $\beta_1(1) = 2$, and $\beta_0(1) = 3$. For the other dimensions, the involution changes the cell, e.g., $\beta_2(3) = 4$, and $\beta_3(3) = 5$.

3.2. Combinatorial maps

A combinatorial map consists of darts and mappings between them. A dart represents a copy of an edge for a specific face and the volume incident to the edge. The mappings sew darts together to describe the adjacency between the object's cells.

Definition 1 (3D Combinatorial map [27]). *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 *(a one-to-one mapping from* D *to* D*);*
- β_2 *and* β_3 *are* involutions *on* D *(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.

Combinatorial maps generalize the half-edge data structure where β_1 corresponds to 'next' and β_2 to 'opposite.' An example is provided in fig. 1. The permutation β_3 essentially encodes a volumetric 'opposite,' while the missing 'previous' is given by β_1^{-1}, usually written β_0. When we want to encode meshes with boundaries, we add a special dart ϵ and extend the permutations β_i to functions $D \cup \{\epsilon\} \to D \cup \{\epsilon\}$ with the convention that $\beta_i(\epsilon) = \epsilon$ for $i \in \{0, \dots, 3\}$. Any dart d such that $\beta_i(d) = \epsilon$ is said to be i-free as it encodes the boundary of an i-cell. Otherwise, the dart is i-sewn. The cells of the mesh (vertices, edges, faces, and volumes) are orbits of the darts under the action of some specific permutations β_i, which can be retrieved via a breadth-first search algorithm (see [11] for all the precise definitions).

A combinatorial map can be reinterpreted as a graph, where darts are the nodes and β_i represents i-labeled arcs. Thus, querying and replacing a part of a map can be explained via graph rewriting similar to the construction developed in [33]. Although this theoretical framework ensures the validity of our query-replace approach, our motivation is practical. We only require a fragment of the theory, namely a tool to compute a match of a query map into an input map, i.e., a subgraph isomorphism.

3.3. Isomorphism

Matching a query in an input mesh is a partial isomorphism problem: a subset of the input must exactly match the pattern. Such isomorphisms can easily be expressed with 3-maps.

Definition 2 (Map Isomorphism [27]). *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 was extended in [9] to deal with i-free darts by requiring that $f(\epsilon) = \epsilon$, ensuring that the boundary is mapped onto the boundary. While computing arbitrary graph isomorphism is known to be a hard problem [15, 18], efficient algorithms exist for specific families of graphs. For instance, planar graphs [22], graphs with polyhedral embeddings [24], or an embedding on a surface of fixed genus [4] admit efficient isomorphism and subgraph isomorphism testing. In these approaches, the key idea is to consider an ordering of the edges around a node to obtain a deterministic graph traversal. The same technique can be used with combinatorial maps, exploiting the natural ordering given by the dimension of the β_i functions [39]. Additionally, combinatorial maps are rigid in the sense of [13], which ensures that we can greedily check for isomorphisms. As a result, isomorphism can be checked by mapping a single dart and indexing all remaining darts via the same traversal performed on the two maps. Then, it suffices to check the pairing of any dart in the first map with all darts from the second map. Standard speed-up techniques can also be performed, e.g., comparing the number of darts.

We use a similar technique where the traversal of the first map is encoded as a word to be read while analyzing the second map. To build this word, we first abstract the traversal of the map as a labeling of the darts.

Definition 3 (Labeling [17]). *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* $l(\epsilon) = 0$.

Such a labeling can be computed via any traversal algorithm as long as the neighbors of a dart are always considered in the same order. We chose a breadth-first traversal, considering dart neighbors ordered by the permutations (first β_1, then β_2, and finally β_3), and labels are given to the darts in prefix order. In the sequel, labeling always means a breadth-first traversal labeling by increasing dimensions. This process is illustrated in 2D on fig. 2.

Given a labeling, the full combinatorics of the map can be expressed as a word built from the labeling by listing the labels of each neighbor.

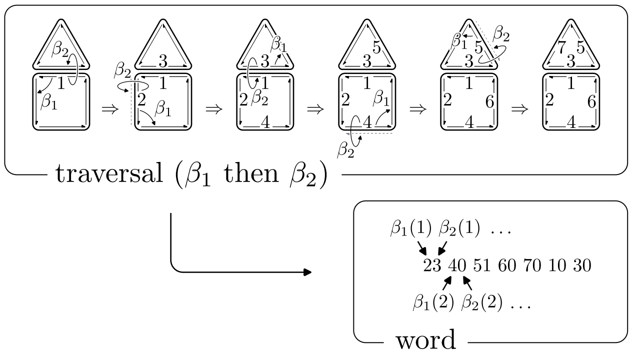

traversal (β_1 then β_2)
β_1(1) β_2(1) ...
23 40 51 60 70 10 30
β_1(2) β_2(2) ...
word

Figure 2: Encoding the combinatorics of a map as a word. In this case, a breadth-first labeling is built: starting from a chosen dart labeled 1, the darts are traversed using β_1 then β_2, and labels are given in prefix order. The associated word encodes, for each dart, the labels of its neighbors, with 0 used to handle the special ϵ dart.

Definition 4 (Word [17]). *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*

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

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 labelled with k, i.e.,* $d_k = l^{-1}(k)$.

Such a word fully encodes the topology of the map. Words can, therefore, be used to assert that maps are isomorphic: two maps are isomorphic if and only if they admit labelings that yield the exact same word [17]. An example of a word is provided in fig. 2. This figure uses some special symbols to deal with boundary conditions that will be explained in section 5.1.2.

3.4. Query-replace

The query-replace construction of [12] only handles the subdivision of topological cells (single volumes for 3-map). In their approach, each rule is encoded as a single 3-map, where the boundary describes the query and the interior defines the replacement. Their matching algorithm is based on *signatures*, which deterministically encode the topology of each volume into a word. Then, matching is performed efficiently by computing the signature of each input volume and checking it against a dictionary of query signatures.

In this paper, we generalize this framework in several directions. First, we redefine the query and replace operations to support queries spanning multiple volumes. This extension requires a more general notion of matching, and places constraints on the replace operation, notably that it must preserve the boundary of the query.

We also introduce two new operations – extend and delete – which respectively add or remove substructures from the mesh without requiring boundary preservation. Finally, we define a fourth operation, apply, which allows users to associate custom code with a query, enabling fully generic transformations. These four operations (redefined replace, and new extend, delete, and apply) form the basis of our extended query-modify framework.

4. Overview of our method

We provide a global overview of our method before detailing it in the following two sections.

Our method modifies an initial 3D mesh M, the *input*, in two steps. First, we identify occurrences of a second mesh Q, called a *query*, in M, extending the query-replace approach of [12] to queries on connected volumes rather than a single one. The second step modifies the match of Q in M. We introduce four possible modifications: *extend*, that adds and glues new volumes around Q; *delete*, that removes some volumes of Q; *replace*, that substitutes the interior of Q; and *apply*, which executes a user-defined function on Q. The first three modifications use a third mesh T, called the *target*, while the fourth is defined by the applied function.

The following two sections present new contributions compared to the previous work of [12]. First, we redefine both the query and replace operations since a new formalization is needed to handle multi-volume patterns. This redefinition leads to two theoretical contributions of this paper: the definition of the query word (see def. 5) via special symbols encoding boundary conditions and the annotation algorithm (see alg. 1) to find a match from a query word. Second, we introduce three new modification operations: extend, delete, and apply. Together with the replace operation, we contribute three algorithms to modify a matched topological query based on conditions about the preserved parts and a fallback approach when these conditions cannot be met.

5. Query

The query operation first finds a submap of the input isomorphic to the query map by encoding a traversal of the query as a word and reproducing it on the input. We require the query pattern to be connected for efficient traversal, as assumed in def. 4. We allow boundary on the query map for the dimension 3 and require all darts to be 1- and 2-sewn.

5.1. Query for a single pattern

We start by explaining how to process a single query, first by creating an annotation from a word before extending the notion of a word to a query word, encoding boundary condition.

5.1.1. Basic case

Even if a query is formally given as a 3-map, we encode its combinatorics as a word (Definition 4) for efficient queries. This word explicitly describes how to check occurrences of the query map within the input map. A match is found by identifying a partial input labeling, which we call an *annotation* such that the induced submap has the same word as the query. More precisely, an annotation maps darts to either a unique integer or a special symbol used to deal with boundaries (as explained in section 5.1.2). The mapping is partial, thus describing a sub-isomorphism, but requires that different darts get mapped to different integers.

Since the query is fixed, we use the word associated with a (BFS) labeling of the query to mimic the traversal on the input map. Thus, labels are propagated according to the word to retrieve the associated annotation, as detailed in alg. 1, starting with label 1 on an input dart. In this basic approach, boundary darts of the query can be mapped to any dart (including ϵ) in the input. The following section refines this behavior by introducing boundary conditions.

Algorithm 1: Annotate a 3-map with a query word.

```
Input: $M = (D, \beta_1, \beta_2, \beta_3)$: a connected 3-map
       $W$: a query word describing the searched pattern
       $d_s \in D$: a starting dart of $M$
Output: An annotation of $M$ for the query word $W$ or fail
1  let annotation be an empty map with darts as keys
2  let dart_annotated be an array of size $|W|/3 + 1$ filled with $\varnothing$
3  annotation[$d_s$] ← 1; dart_annotated[1] ← $d_s$
4  let $Q$ be an empty queue
5  add $d_s$ to $Q$
6  while $Q$ is not empty do
7      pop $d$ from the head of $Q$
8      for $i \leftarrow 1$ to 3 do
           // linked dart and constraint
9          $d_i \leftarrow \beta_i(d)$
10         $c \leftarrow W[(\text{annotation}[d] - 1) \times 3 + i]$
           // check word constraint
11         if $c = 0$ then continue        // $\epsilon$ in query
12         if $d_i$ is annotated then
13             assert $c$ = annotation[$d_i$]
14         else
15             assert $d_i \neq \epsilon$ and dart_annotated[$c$] = $\varnothing$
16             annotation[$d_i$] ← $c$
17             dart_annotated[$c$] ← $d_i$
18             add $d_i$ to $Q$
19 return annotation
```

5.1.2. Boundary conditions

From defs. 3 and 4, $w_{3 \cdot k} = 0$ if the dart labeled k belongs to the query boundary. In [12], queries were extracted as the boundary of the replace pattern, ensuring fully automatic boundary detection and word construction. Our extension to multi-cell patterns aims to provide more control over the query pattern, and we define four constraints on how a query boundary dart can be matched to an input dart d:

wildcard ($*$) match any input dart, as in alg. 1;

strict boundary (ϵ) only match a boundary dart: $\beta_3(d) = \epsilon$;

strict interior ($\bar{\epsilon}$) do not match a boundary dart: $\beta_3(d) \neq \epsilon$;

no connection ($\nleftrightarrow$) do not 3-sew to another matched input dart. This constraint prevents new β_3 connections between darts of the matched submap that were not originally sewn together in the query.

Examples of these boundary conditions are given in fig. 3. To handle these additional behaviors, we extend the definition of a word.

Definition 5 (Query word). *Given a connected input* 3*-map* $M = (D, \beta_1, \beta_2, \beta_3)$ *and a labeling* $l : D \cup \{\epsilon\} \to \{0, \dots, |D|\}$, *the* query word *associated with* (M, l) *is the finite sequence*

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

in $\{1, \dots, |D|\} \cup \{\epsilon, \bar{\epsilon}, *, \nleftrightarrow\}$ *such that for all* $i \in \{1, 2, 3\}$ *and* $k \in \{1, \dots, |D|\}$, *if* $\beta_i(d_k) \neq \epsilon$, *then* $w_{i\cdot k} = l(\beta_i(d_k))$ *(where* $d_k = l^{-1}(k)$*) and otherwise (i.e.,* $\beta_i(d_k) = \epsilon$*)* $w_{i\cdot k} \in \{\epsilon, \bar{\epsilon}, *, \nleftrightarrow\}$.

The query word is built from a valid query map, where all darts are 1- and 2-sewn. The additional symbols $\epsilon, \bar{\epsilon}, *, \nleftrightarrow$ apply only to 3-free darts. In alg. 1, we refine the constraint checks (line 11) between c and d_i as follows:

$c \rightarrow$ / $d_i \downarrow$	$\{1, \dots, \lvert D \rvert\}$	$*$	ϵ	$\bar{\epsilon}$	$\nleftrightarrow$
$d_i = \epsilon$	×	✓	✓	×	✓
d_i is not annotated	(a)	✓	×	✓	(c)
annotation$[d_i] \in \{1, \dots, \lvert D \rvert\}$	(b)	✓	×	✓	×
annotation$[d_i] = \emptyset$	×	✓	×	✓	✓

where ✓ means that the dart fulfills the constraint and × that it violates it. Cases (a) and (b) are handled as in alg. 1 while case (c) accepts and annotates the dart with $\emptyset$ to prevent it from receiving another label later. This case is illustrated on fig. 3 where the $\nleftrightarrow$ and ϵ boundary conditions prevent some matches. In particular, the ϵ condition requires the matched dart to be on the boundary, and the $\nleftrightarrow$ condition prevents two quads from being connected.

5.2. Multiple queries

Many modeling operations are described by a table of local topological configurations to be processed. Implementing the case analysis on an input mesh can be tedious, depending on the number and complexity of the configurations. For example, in section 7.1, we implement a tetrahedra recombination method with 171 configurations of up to 15 tetrahedra [35]. A naive approach would be to iteratively test each input dart against every query word using alg. 1. Here, we show that the search can be improved by handling multiple queries on the same dart and properly managing overlapping matches.

5.2.1. Multiple query words on the same dart

Given a set of query words (def. 5), we propose using a prefix tree (or trie [26, Section 6.3]). In such a tree, each node stores a symbol from the alphabet. A path from the root

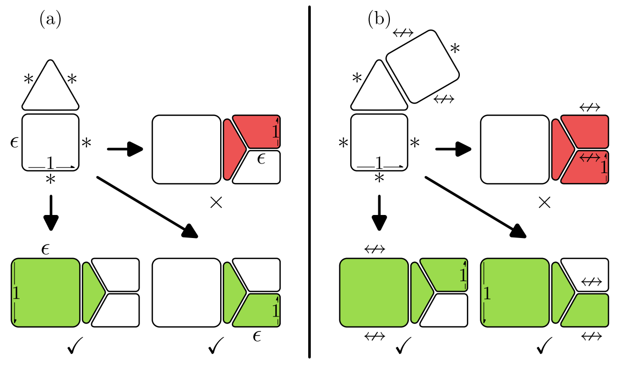


Figure 3: 2D cases for the boundary conditions. Valid matches are marked with a ✓ and invalid ones with a ×. Query (a) uses an ϵ constraint enforcing the match onto a boundary dart. In query (b), the $\nleftrightarrow$ boundary conditions allow matching anything except another matched dart, thus preventing unintended connections between parts of the matched pattern that were unconnected in the query.

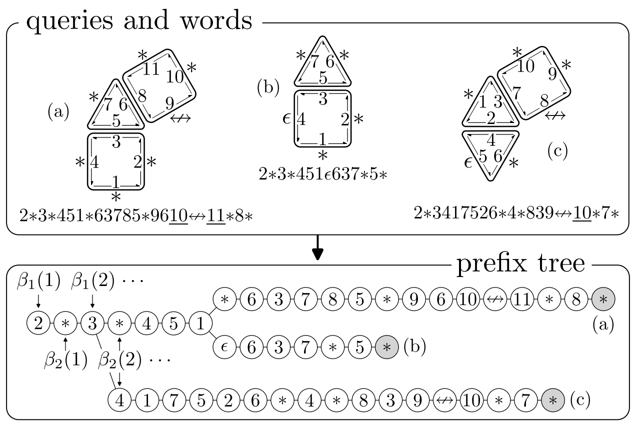


Figure 4: A prefix tree encoding multiple 2D queries simultaneously. Queries (a), (b), and (c) are first encoded into words via a labeling. These words are then merged into a prefix tree. With such a tree, trying to build a match on a given starting dart means extending valid prefixes until a query is indeed found. Thus, queries with a common prefix are checked simultaneously, and branches are pruned when incompatible with the input.

to a node represents a prefix of a query word. An example is given on fig. 4. A node is *terminal* when the path corresponds to a complete query word. In particular, all leaves are terminal, while internal nodes are terminal when the corresponding query word is a prefix of another. Algorithm 1 can be extended to handle prefix trees and test multiple query words by traversing the input 3-map and the tree simultaneously. The path from the root to the current node tracks the prefix accepted so far. A match is found when a terminal node is reached. The associated annotation is then recorded. The search continues recursively for child nodes that satisfy the constraints.

5.2.2. *Overlapping matches*

The prefix tree efficiently checks multiple queries from a single dart in the input mesh. However, since matches may overlap, we need a strategy to determine a valid set of modifications to apply. The intended behavior is similar to L-systems, where all occurrences of a given sequence of symbols are modified simultaneously. We propose two solutions: (1) a local greedy search over the input map and (2) a global search that ranks valid matches.

The greedy approach selects matches in the order they are found. Once a match is selected, no other match may overlap its labeled darts. Using the annotations from section 5.1.2, we label every dart involved in a selected match with $\varnothing$ before re-running alg. 1 from another dart. This process prevents the algorithm from assigning a label to an already matched dart, efficiently filtering out overlapping matches.

Beyond the mesh's combinatorics, additional criteria can help rank query results. To this end, we propose a global approach that first collects all possible matches, including overlapping ones, and then ranks them based on an external metric. Inspired by another work of Pellerin *et al.* [34], we construct a graph where each node represents a match, and edges link overlapping matches. Ideally, we would weight each match and find a maximally weighted independent set in this graph. However, this is NP-complete, and its complexity grows exponentially with the number of matches. Instead, we sort matches by the desired criterion and select them in order, discarding conflicting matches.

6. Modify

Our query-modify framework identifies and modifies a pattern in an input mesh. The pattern is precisely defined by its encoding as a 3-map. Once a match is found, we perform the modification using four operations: *extend*, *delete*, *replace*, and *apply*. The first three operations are no-code solutions, requiring only the query and target meshes from the user. While they follow the same abstract behavior – removing the query and inserting the target – they differ in their intended behavior and, therefore, their underlying assumptions. The fourth operation, *apply*, generalizes the first three operations by applying an arbitrary function to the match. It removes the previous constraints but requires manual implementation by the user and special care to preserve a valid topology.

In this section, we write Q for the query, T for the target, and M for the input, each given as a 3-map. From them, we compute:

1. m_{QT}, a mapping from Q to T describing the links between the query and the target;

2. m_{QM}, a mapping from Q to M, obtained from alg. 1, describing the links between the query and the input.

We now detail the three structured modify operations, their modifications, and the constraints that guarantee their soundness. We recall that we assume Q and T to be connected, without boundary for dimensions 1 and 2, but possible boundaries for dimension 3.

6.1. Extend

The *extend* operation preserves the match of the query in the input mesh while attaching new volumes at its border, effectively extending the query. Thus, Q must be a submap of T, meaning that m_{QT} is an isomorphism from Q to a subset of T (i.e., some darts in T have no pre-image in Q). We write $D_{T\backslash Q}$ for the set of darts in T without a pre-image in Q. This set corresponds to the new elements added by the extend operation. After adding these darts to M, they are reconnected at the border of the query match with alg. 2.

Algorithm 2: Extend a 3-map M, given a query Q, a target T, and mappings m_{QT} and m_{QM}.

```
1 Copy all darts from $D_{T\backslash Q}$ to $M$, building a
    mapping $m_{TM}$ from each dart in $D_{T\backslash Q}$ to its copy
  // Copy all links $\beta_i$
2 for any two darts $d$ and $d'$ in $D_{T\backslash Q}$ such that
    $d = \beta_i(d')$ do
3   $i$-sew their copies by sewing $m_{TM}(d)$ and
      $m_{TM}(d')$
  // Connect the new volumes
4 for each dart $d$ in $D_{T\backslash Q}$ do
5   if $d$ is 3-sewn to $d' = \beta_3(d)$ that is not in $D_{T\backslash Q}$
      then
6     3-sew $m_{TM}(d)$ and $m_{QM}(m_{QT}^{-1}(d'))$
```

For the operation to be valid, all darts in M attached to a new volume must be 3-free before the operation. The query automatically annotates these darts with ϵ (section 5.1.2).

6.2. Delete

The *delete* operation, summarized in alg. 3, is essentially the inverse of extend: it preserves only a subpart of the query in the input, deleting the rest. Thus, T must be a submap of Q, meaning that m_{QT} is an isomorphism from a subset of Q to T (i.e., some darts in Q have no image in T). We denote by $D_{Q\backslash T}$ the darts in Q without an image in T, corresponding to the elements removed by the operation. To maintain the topological consistency, we first disconnect these darts from the rest of the mesh before deleting them. Since Q and T are restricted to having only 3-boundaries, only β_3 links may exist between $D_{Q\backslash T}$ and the rest of the query mesh, ensuring that only volumes are disconnected.

6.3. Replace

The *replace* operation performs a rewriting step, replacing the query with the target while preserving the border of the match. The operation first merges all matched volumes into one using the removal operation [10], then replaces the volume's interior with the target structure as in [12], as explained in alg. 4

Algorithm 3: Delete volumes from a 3-map M, given a query Q, a target T, and mappings m_{QT} and m_{QM}.

```
   // Disconnect the volumes to delete
1  for any darts $d \in D_{Q\setminus T}$ and $d' \notin D_{Q\setminus T}$ such that
     $d = \beta_3(d')$ do
2      3-unsew the darts $m_{QM}(d)$ and $m_{QM}(d')$
   // Delete the disconnected volumes
3  Delete all darts in $m_{QM}(D_{Q\setminus T})$
```

Algorithm 4: Replace interior of volumes in a 3-map M, given a query Q, a target T, and mappings m_{QT} and m_{QM}.

```
   // Merge all volumes of the match into one
1  for each 3-sewn dart $d \in Q$ do
2      Remove the face containing $m_{QM}(d)$ in $M$,
         merging the two incident volumes.
   // Replace the interior of the volume
3  Replace the interior using $T$, $m_{QT}$, and $m_{QM}$.
```

Since the replace operation requires that the boundary of the query is preserved, it cannot simulate either the extend operation or the delete one. In all generality, a query-replace can also not be simulated by a sequence of a query-delete followed by a query-extend as the replace operation merges the volumes.

6.4. Apply

The previous three operations describe modifications through a query and target map, expanding, removing, or substituting mesh parts while maintaining topological consistency. We introduce the *apply* operation for more complex transformations, applying an arbitrary function to the match. The implementation is simplified by using the result of alg. 1 and only asking the user for an implementation of the function. As a result, some parts of the apply operation still need manual implementation, but it allows for modifications beyond the predefined transformations. In particular, the extend, delete, and replace operations can be seen as specific apply operations.

6.5. Dealing with the geometry

The explanations so far have focused on the modification of mesh topology. Geometry must also be considered to describe the shape of the mesh properly. We now describe how vertex positions are handled during the extend, delete, and replace operations. For the apply operation, geometry management is delegated to the user-provided function.

We again consider a query Q, its match in an input mesh M, and a target mesh T. In a 3-map, each vertex corresponds to an orbit in the combinatorial structure, i.e., a set of darts that define a unique vertex together. Each such vertex is then associated with a single 3D point. This structure has two important implications. First, we can consider a matched vertex as preserved if at least one of its darts is preserved, i.e., if it appears in the mapping m_{QT}. On the contrary, a vertex is considered deleted, respectively created, if all its darts are. Second, a unique position must be assigned to each vertex. However, vertex merging from the input mesh M never occurs in the proposed modify operations (only volumes are merged in the replace operation). Thus, no merging conflict needs to be solved.

We propose minimally modifying the geometry, meaning that vertices unaffected by the transformation retain their original position. This includes preserved vertices and those not involved in the operation, conveying the motivation that our Q&M framework encodes local edits. If a vertex is deleted, i.e., all its darts are in $D_{Q\setminus T}$, the associated 3D point is discarded. If a vertex is created, i.e., all its darts are in $D_{T\setminus Q}$, then a new 3D point must be computed. We propose to use the generalized barycentric coordinates of each new vertex in the target mesh expressed on the preserved vertices of T, following the method of [23]. More precisely, positions for the new vertices are interpolated using these barycentric coordinates with the positions of the preserved vertices in the input mesh M. This computation ensures the new geometry is consistent with the shape of the original pattern in the target mesh T while adapting to the input mesh M(see examples in section 7).

6.6. Implementation details

A 3D combinatorial map can be encoded as an array of darts. Each dart d stores four indices, one for each dart linked to d by β_i, for $i \in \{0, 1, 2, 3\}$. The query mechanism directly implements alg. 1. The three modification operations (extend, delete, and replace) are also direct implementations of their respective algorithms (algs. 2 to 4). The *sew* (resp. *unsew*) operation simply updates the relevant indices of the involved darts. Replacing the interior of a volume follows the approach described in [12]. The apply operation simply encapsulates the user-provided function to be executed on the matched darts.

7. Experiments

We implemented the four query-modify operations (query-extend, query-delete, query-replace, and query-apply) in C++, using CGAL's combinatorial maps [7] and linear cell complexes [8], which provides an additional layer for handling the geometry. All experiments were conducted on an Intel® i9-10900K GPU @ 3.70 GHz with 64 GB of RAM. Our code will be publicly available upon acceptance of the paper.

We designed four experiments to show the versatility of our query-modify operations. The first experiment (section 7.1) implements a tetrahedral recombination method, converting a tetrahedral mesh into a hexahedral one using all 171 topological configurations of tetrahedra from [35]. Since they made these configurations available, we directly used them as query-replace operations, achieving, to our knowledge, the first exhaustive implementation of tetrahedral recombination. The second experiment (section 7.2) shows how to combine query-delete and query-extend to modify features of an existing model as a way to ease design processes. The third experiment (section 7.3) uses our method to modify production meshes extracted from Blender open movie projects. The last experiment (section 7.4) is a toy example to illustrate the query-apply operation via the generation of a terrain with small houses.

7.1. Tetrahedra recombination

In our first and main experiment, we apply our query-modify operation to implement a tetrahedra recombination method (introduced in section 2.3). From a tetrahedral mesh as input, sets of tetrahedra are combined into hexahedra whenever possible. This first experiment is a core contribution of our framework since, to our knowledge, a method of identifying all 171 different cases from [35] was never implemented. Our solution is the first successful implementation, made feasible by our query-replace operation. It follows these three main steps:

1. load the 171 VTK files from [35] and create 171 corresponding query configurations;
2. for each query mesh, generate a corresponding target mesh by merging all its tetrahedra into a single triangulated hexahedron;
3. apply as many query-replace operations as possible to the input tetrahedral mesh.

Different strategies can be considered to find all possible matches, as discussed in section 5.2.2. This experiment adopts a global approach based on the scaled Jacobian [25] as a geometric quality metric. The scaled Jacobian quantifies how much a hexahedron deviates from a perfect cube, ranging from 1 (a cube) to negative values (invalid elements, e.g., with self-intersections). This strategy follows four steps:

1. compute all possible matches of the 171 queries in the given mesh, allowing overlaps;
2. compute the scaled Jacobian of each match, considering the hexahedron formed by the union of the matched tetrahedra;
3. sort the matches in decreasing order of their scaled Jacobian;
4. apply query-replace operations to matches from highest to lowest Jacobian values, discarding conflicting matches after each replacement to avoid overlaps.

We automatically annotate all the darts on the border of the queries with $\nleftrightarrow$ to avoid degenerate matches, e.g., folded ones with less than eight vertices.

We compare our results with the method of [34], which identifies and assembles groups of tetrahedra into higher-order elements like hexahedra, prisms, and pyramids. Their approach relies on a vertex-based strategy. For hexahedron recombination, the algorithm examines combinations of 8 vertices corresponding to potential hexahedron configurations with 8 nested `for`-loops. Poor-quality candidates are filtered out using the scaled Jacobian as a geometric quality measure. For comparison, we used the publicly available code from `https://www.hextreme.eu/download/`, written HXT in our results. We additionally used the HXT's implementation of the scaled Jacobian as our geometric quality metric, meaning that the only difference is the combinatorial matching.

It is important to note that the HXT method does not cover the full set of 171 cases of [35], and to our knowledge, no method has provided such an implementation yet. Manually handling so many configurations would require significant effort, an issue our Q&M framework circumvents by design.

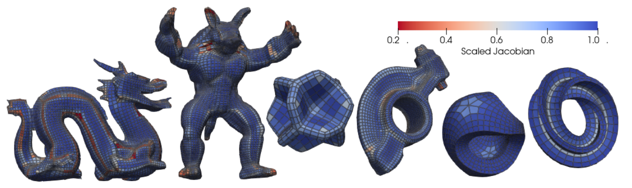


Figure 5: The six hexahedral meshes used in our tetrahedra recombination experiment. From left to right: Armadillo, Dragon, Dualocta, Rockerarm, Sculpture, and Torus-twist. The colormap represents the scaled Jacobian of the hexahedra.

7.1.1. Results on a synthetic dataset

This experiment aims to evaluate our implementation of the tetrahedra recombination method from [34]. To establish a ground truth, we constructed optimal scenarios by transforming hexahedral meshes into tetrahedral meshes without adding new points. Each hexahedron was randomly triangulated by splitting quads into triangles, then decomposed into a compatible set of tetrahedra. This construction guarantees that an optimal recombination exists, i.e., without remaining tetrahedra. Thus, we can quantify how close our results are to this optimal. In practice, we used the six hexahedral meshes shown in fig. 5, obtained from HexLab.net [5].

The results of our method and those of Pellerin *et al.* [34] are shown in table 1. We provide metrics for the initial hexahedral meshes, their tetrahedralized versions, and the results of both methods. For each case, we indicate the number of recombined hexahedra and the number of remaining tetrahedra. Our method consistently achieves perfect recombination, recovering 100% of the original hexahedra. While producing very good results, HXT does not always recover all hexahedra.

Performance-wise, our method is significantly slower than HXT, as expected, since HXT uses a much simpler matching algorithm. Our implementation prioritizes generality over performance, but several optimizations could significantly improve runtime. A minor gain would come from replacing the recursive query traversal with an iterative version. More substantial improvements include parallelizing the matching process. At the tree level, we can start

	Hex mesh		Tet mesh		Q&M Jacobian strategy					HXT				
	#v	#d	#v	#d	#Q	#h	#t	Memory	Time	#Q	#h	#t	Memory	Time
Armadillo	29,935	718,440	149,675	1,796,100	827,729	29,935	0	130 MB	205.8s	392,572	29,667	1,146	163.7MB	4.26s
Dragon	14,009	336,216	70,045	840,540	358,737	14,009	0	70.8MB	89.5s	175,128	13,997	55	84.3MB	2.36s
Dualocta	1,344	32,256	6,720	80,640	30,072	1,344	0	26.1MB	7.8s	12,999	1,344	0	11.7MB	0.19s
Rockerarm	10,600	254,400	53,000	636,000	280,360	10,600	0	59.7MB	69.4s	130,372	10,282	1,370	61.5MB	2.08s
Sculpture	2,240	53,760	11,200	134,400	52,040	2,240	0	29.2MB	12.9s	24,306	2,240	0	17.6MB	0.35s
Torus-twist	768	18,432	3,840	46,080	12,800	768	0	24.1MB	3.3s	6,007	768	0	9.4MB	0.10s

Table 1
Mesh statistics about the initial hexahedral meshes (Hex mesh), the generated tetrahedral ones (Tet mesh), and the result of Q&M and HXT. For Hex mesh and Tet mesh, #v and #d respectively denote the number of volumes and darts. For the final recombined meshes: #Q is the total number of query found, #h and #t are the number of hexahedra and remaining tetrahedra, Memory is the maximal resident memory used (in Mega-Bytes) and Time is the computation time of the method (in seconds).

with a single thread at the root, and spawn a new thread for each branch, such that branches can be explored independently without requiring synchronization. Matching could also be parallelized at the mesh level, as candidate starting points can be evaluated concurrently using read-only access. Additional speedups could be achieved through lightweight geometric filtering. For example, restricting searches to local neighborhoods using bounding spheres, or aborting early when the matched geometry becomes overly distorted. These optimizations remain compatible with the generality of the framework and are left for future work.

Our method also identifies over twice as many valid queries on average, confirming that HXT does not find all possible recombinations. While this does not lead to important differences in these synthetic scenarios, the difference becomes more significant for real-world meshes, as demonstrated in the following experiment.

Table 1 suggests that the runtime scales linearly with mesh size. The total runtime is the sum of the costs of individual queries and thus depends on both the complexity of the mesh and the structure of the prefix tree encoding the query patterns. In this experiment, the query set is fixed (the 171 cases), and each pattern is small relative to the mesh. As a result, the dominant factor becomes the number of starting points in the mesh, leading to an approximately linear runtime. A detailed performance breakdown is beyond the scope of this paper, which focuses on demonstrating the general applicability and extensibility of our query-modify framework.

To grasp the memory impact of our method, we compared the maximum Resident Set Size (RSS) of both methods and found no significant difference. On average, our method used 56.7 MB, while HXT used 58 MB. While RSS is not a perfect measure of memory usage for each data structure, it provides a good approximation for comparing the two methods.

7.1.2. Results on real cases

In the previous experiment, we compared our method and HXT on specific meshes designed to ensure optimal recombination. While this was useful for evaluating the methods, these meshes are somewhat artificial. In this second experiment, we apply our method to real tetrahedral meshes from the `HexMe`[1] database [2], which contains 189 tetrahedral meshes generated by their authors to *"enable consistent and practically meaningful evaluation of hexahedral meshing algorithms and related techniques"*. Three examples are shown in fig. 6.

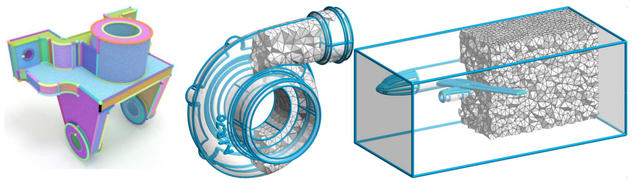


Figure 6: Three tetrahedral meshes from `HexMe` database (images from [2]).

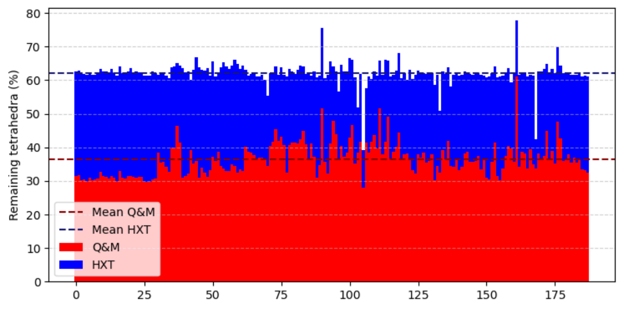


Figure 7: Percentage of remaining tetrahedra (relative to the initial meshes) for our method (Q&M) and the HXT method. The x-axis lists the 189 input meshes used in the evaluation.

We ran the same experiment on 189 meshes using our Q&M method (based on the 171 configurations and the scaled Jacobian) and HXT. Both methods use the implementation of the same geometric criterion, but our method consistently outperforms HXT, with only 36% remaining tetrahedra on average versus 62% for HXT (see fig. 7). This result suggests that HXT misses many matches that our method captures. The trade-off is the computation time: our method takes 814 seconds on average versus 36 seconds for HXT. As mentioned earlier, our implementation is not optimized for this specific task, but it can be improved.

These results demonstrate the potential of our method for transforming tetrahedral meshes into hexahedral ones. While

[1] HexMe: `https://www.algohex.eu/publications/hex-me-if-you-can/`.

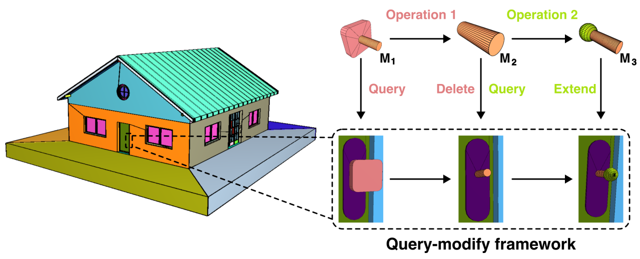


Figure 8: Example of use of our new query-modify operation to change all the handles of a given 3D model of a house (volumes are rendered in a randomly assigned color).

our simple Jacobian-based strategy is insufficient for high-quality results (and is not the goal of our work), more advanced methods can build on our query-modify framework as a core tool.

7.2. Modification of a 3D model

We applied our query-modify framework to update an existing 3D model in a second experiment. This experiment addresses a practical use case: designers often create 3D models using object databases, meaning that model parts inherently share the same topology. Queries via words encoding this topology enable simultaneous modification of all occurrences of a given part, making our framework highly useful for real-world applications. Starting from a 3D mesh of a house (fig. 8 left), loaded from an IFC (Industry Foundation Classes) file containing 895 volumes, we modified its door handles. While the doors are different, they share the same handle, meaning that an architect could easily replace them using our query-modify operations:

1. remove some part of the handle while preserving its support via a query-delete operation using the mesh labeled M_1 as the query (in fig. 8) and labeled M_2 as the target;
2. add a new handle onto the support via a query-extend operation with M_2 as the query and M_3 as the target (cf. fig. 8).

Applying these two query/modify operations transforms both handles, as illustrated for one door in fig. 8 bottom/right. The computation is fast: 0.04s for the first query, 0.0001s for the delete, 0.17s for the second query, and 0.02s for the extend. Thus, the total operation takes 0.23s. Beyond efficiency, one main advantage of this method is its simplicity. No additional code is needed—only three meshes are required: the initial mesh of the door handle extracted from the house model, a second mesh with one volume removed, and a third with a new handle added using a 3D modeler. The example illustrates that parts of the initial mesh, here the door handles, can be detected only from the topology without requiring additional metadata.

Flexibility is another key benefit: we can edit the third mesh in a 3D modeler to customize the result. Intuitively, our method allows editing all handles of the house by only modifying one.

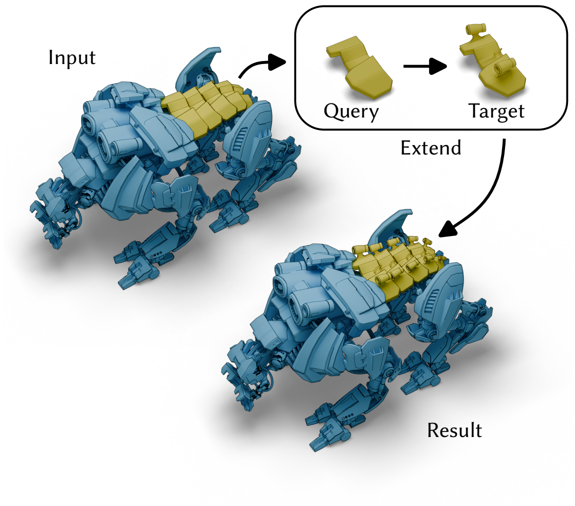


Figure 9: On this production model, many mechanical parts are duplicated across the model. In this context, our method provides a means for coherently modifying all the replicas. The Quadbot model is courtesy of Francesco Siddi from the Blender Tears of Steel open movie project.

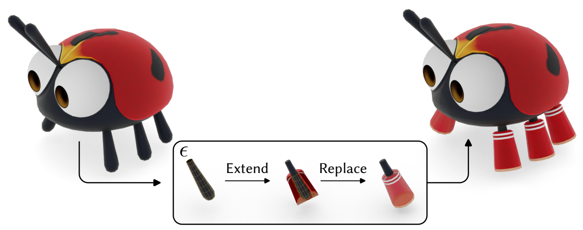


Figure 10: On this model, we first apply an extend operation, leading to a non-manifold intermediate state, and follow with a replace operation to restore the manifoldness of the model. On the replace operation, the query had to be marked with an ϵ boundary condition to prevent matches on the antennas. Ladybug model courtesy of Simon Thommes, Blender Sprite Fright open movie project.

7.3. Modification of production surface meshes

We demonstrate here the usability of our method on production meshes extracted from Blender open movie projects. These examples demonstrate that production meshes have self-similar portions that can benefit from our method for batch modifications, as in fig. 9. Blender uses a *bmesh* data structure, extending half-edges to represent non-manifold meshes.

This shows an interest in using combinatorial maps and volumes, even when dealing with surface meshes. Indeed, combinatorial maps deterministically represent non-manifold structures that can be present in the input or as intermediate steps as in fig. 10.

This final experiment shows the value of our method in real-case scenarios. Indeed, surface meshes used in production often contain repeated elements, which can all be edited simultaneously with our query-modify framework. Our solution particularly appeals to designers as it enables coherent

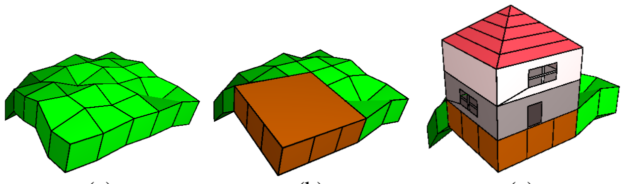


Figure 11: Example of terrain generation with houses. (a) Generation of a grid of voxels of size 5×5. (b) Result after one query-apply operation. (c) Final result after one query-extend operation.

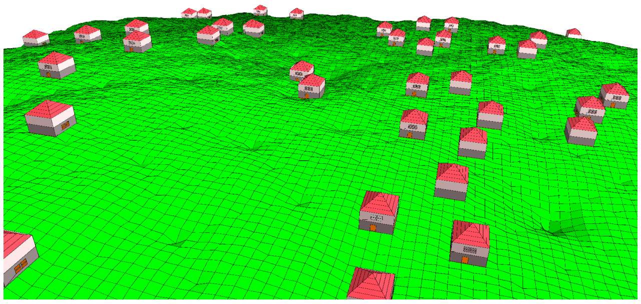


Figure 12: Result of terrain generation with houses on a 128×128 grid.

Step	Volumes	Darts	Computation Time (s)
(1)	16,384	425,984	0.028
(2)	15,984	419,984	0.012 (0.008 query, 0.004 apply)
(3)	16,134	431,684	0.68 (0.62 query, 0.06 extend)

Table 2
Evolution of size metrics and computation times for the terrain generation.

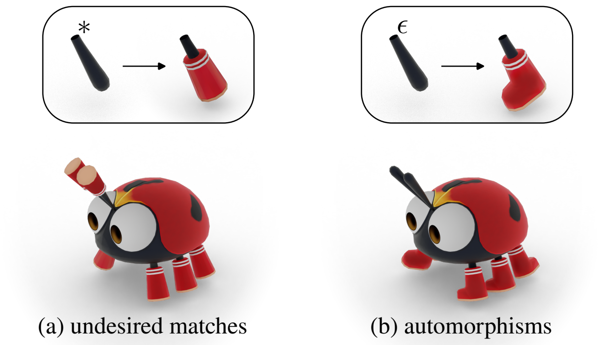


Figure 13: Limitations of topological matching. Some undesired portions of the mesh may exhibit the same topology (a) thus leading to undesirable modifications. Queries can also be automorphic (b) which results on unpredictable modifications, here with random feet orientation.

modifications of all replicas without coding or interactive manipulations.

7.4. Terrain generation with houses

We implemented a last experiment to generate a 3D terrain with houses. This experiment is a toy example intended to illustrate the flexibility of our query-modify operations. The generation is done in three steps.

(1) A voxel grid is generated from a height map using the diamond-square method [16]. Voxels are then created and assembled into a 3-map using methods from CGAL.

(2) Flat spaces are created for placing houses using a query-apply operation. The query consists of 3×3 cubes with triangulated bottom faces. The apply operation merges the volumes and their upper faces, and flattens the resulting face to the average height of the original points.

(3) The final modification adds houses to the flattened spots with a query-extend operation. The query mesh is the target of the previous query-apply. The target mesh, shown in fig. 11b, extends the query with three additional volumes to form a house. The meshes were created using a 3D modeler. Applying the query-extend operation on the modified terrain automatically adds one house to each flattened area.

The mesh shown in fig. 12 has been obtained by generating a 128×128 grid, flattening areas with 50 query-apply operations and adding houses with as many query-extend operations. Size metrics and computation time are given in table 2, showing the method's efficiency. Our method generates a terrain of arbitrary size with minimal implementation effort: a function for the grid creation (~20 lines of code) and a function in the apply operation (~10 lines). The rest of the process relies on two calls to query-modify operations. Refining the houses would only require updating the target mesh in a 3D modeler without code update.

8. Limitations

Our approach solely focuses on topological matching. We are therefore able to locate topologically identical pieces of meshes, even though their geometry differs, and will not match portions with similar shapes, but different topology. This is enough in several cases, as illustrated in the previous results.

However, relying solely on topological matching also introduces risks of false positives when distinct geometric components share the same topology. For instance, in fig. 13a, the leg mesh is topologically equivalent to part of the antenna mesh. In the ladybug experiment (fig. 10), we avoided such mismatches by constraining the query boundary with fixed labels (ϵ) rather than wildcards ($*$). To improve robustness, lightweight geometric filters – such as bounding box comparisons, orientation checks, or proximity measures – can help eliminate irrelevant matches, as already done in our hexahedral recombination experiments using the scaled Jacobian (section 7.1).

Such filters can also assist in handling query automorphisms, where topological symmetry leads to multiple valid matches. Without disambiguation, the current implementation selects one arbitrarily, potentially leading to unpredictable results in batch editing. For example, the boot modification in fig. 13b can be applied in four orientations. A simple geometric heuristic, such as preferring toes that point toward the head, can restore consistent behavior.

While we leave full integration of generic strategies to future work, we believe such extensions could significantly enhance practical robustness.

9. Conclusion

In this paper, we defined a new generic query-modify framework that allows automatic volumetric mesh processing by topological matching. The first part of our method is the query of a pattern into a given 3D mesh. Matching the query is efficient thanks to a topological word representing the pattern's combinatorics. Moreover, several words can be grouped in a search tree, allowing fast simultaneous queries of several patterns. The second part of our method allows us to extend, delete, or replace some volumes of the pattern given a target mesh. A more generic operation allows the application of any code associated with some query, providing a fully generic method.

The strong points of our method are: (1) its genericity – the framework is applicable to a wide range of use cases, as demonstrated in our experiments; (2) its topological validity – grounded in the formal definitions of combinatorial maps, isomorphisms, and our operations, the framework ensures topological correctness of the result; (3) its simplicity – the extend, delete, and replace operations require no additional coding but only query and target meshes. The apply operation, while requiring a small implementation effort, offers full flexibility for custom modifications.

Our query-modify framework could still be improved by relaxing some constraints imposed on the query and target meshes. We could enable non-connected query and target meshes, meshes with 1D or 2D boundaries, or annotated darts outside certain boundaries, raising questions about the representational power of words and their practical use. For instance, allowing non-connected query meshes leads to an exponential complexity without further optimizations. Another promising direction is the composition of multiple modification steps, which requires new constraints to ensure topological soundness. We can also investigate how to process overlapping queries and determine whether their associated modify operations are independent and can be realized simultaneously, in the spirit of critical pair analysis [20, 30]. As a speed-up technique, we could explore how parallelization can enable synchronous operations applications.

More challenging future work remains to improve our framework's simplicity and expressiveness, including automatically generating query-modify operations from query and target meshes to obtain complex operations without human intervention. We could also explore using our query-modify framework for partial matchings or geometric matching. Exact topological matching is helpful for many real-world objects as designers use copy-pasting and databases containing parts to build larger objects. Our method is naturally appealing in these cases as it allows for simultaneous modification of all isomorphic parts. However, extending our framework to consider partial or geometrical matching would enable dealing with slightly modified copies of an original part. Such an extension raises many difficult questions that should be studied in future work. For instance, how do we partially match a given word or define modification over partial queries? How can we encode geometric properties in an equivalent of the topological words that allow efficient matching and generic edits?

References

[1] Arnould, A., Belhaouari, H., Bellet, T., Le Gall, P., Pascual, R., 2022. Preserving consistency in geometric modeling with graph transformations. Mathematical Structures in Computer Science 32, 300–347. doi:10.1017/S0960129522000226.

[2] Beaufort, P.A., Reberol, M., Kalmykov, D., Liu, H., Ledoux, F., Bommes, D., 2022. Hex Me If You Can. Computer Graphics Forum doi:10.1111/cgf.14608.

[3] Belhaouari, H., Arnould, A., Le Gall, P., Bellet, T., 2014. Jerboa: A graph transformation library for topology-based geometric modeling, in: Giese, H., König, B. (Eds.), Graph Transformation, Springer International Publishing. pp. 269–284.

[4] Bonsma, P., 2012. Surface Split Decompositions and Subgraph Isomorphism in Graphs on Surfaces, in: Dürr, C., Wilke, T. (Eds.), 29th International Symposium on Theoretical Aspects of Computer Science (STACS 2012), pp. 531–542. doi:10.4230/LIPIcs.STACS.2012.531.

[5] Bracci, M., Tarini, M., Pietroni, N., Livesu, M., Cignoni, P., 2019. Hexalab.net: An online viewer for hexahedral meshes. Computer-Aided Design 110, 24–36. doi:https://doi.org/10.1016/j.cad.2018.12.003.

[6] Cagan, J., 2001. Engineering Shape Grammars: Where We Have Been and Where We Are Going, in: Antonsson, E.K., Cagan, J. (Eds.), Formal Engineering Design Synthesis. Cambridge University Press, pp. 65–92. doi:10.1017/CBO9780511529627.006.

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

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

[9] Damiand, G., De La Higuera, C., Janodet, J.C., Samuel, E., Solnon, C., 2009. Polynomial algorithm for submap isomorphism: Application to searching patterns in images, in: Proc. of 7th Workshop on Graph-Based Representation in Pattern Recognition (GBR), Springer Berlin/Heidelberg. pp. 102–112. doi:10.1007/978-3-642-02124-4_11.

[10] Damiand, G., Lienhardt, P., 2003. Removal and contraction for n-dimensional generalized maps, in: Proc. of 11th International Conference on Discrete Geometry for Computer Imagery (DGCI), Springer Berlin/Heidelberg. pp. 408–419.

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

[12] Damiand, G., Nivoliers, V., 2022. Query-replace operations for topologically controlled 3d mesh editing. Computers & Graphics (C&G) 106, 187–199.

[13] Danos, V., Heckel, R., Sobocinski, P., 2014. Transformation and Refinement of Rigid Structures, in: Giese, H., König, B. (Eds.), Graph Transformation, Springer International Publishing. pp. 146–160. doi:10.1007/978-3-319-09108-2_10.

[14] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G., 2006. Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series, Springer-Verlag. doi:10.1007/3-540-31188-2.

[15] Fortin, S., 1996. The Graph Isomorphism Problem. Technical Report. University of Alberta Libraries. doi:10.7939/R3SX64C5K.

[16] Fournier, A., Fussell, D., Carpenter, L., 1998. Computer rendering of stochastic models. Association for Computing Machinery. pp. 189–202.

[17] Gosselin, S., Damiand, G., Solnon, C., 2011. Efficient search of combinatorial maps using signatures. Theoretical Computer Science (TCS) 412, 1392–1405. doi:10.1016/j.tcs.2010.10.029.

[18] Grohe, M., Schweitzer, P., 2020. The graph isomorphism problem. Communications of the ACM 63, 128–134. doi:10.1145/3372123.

[19] Haakonsen, S.M., Rønnquist, A., Labonnote, N., 2023. Fifty years of shape grammars: A systematic mapping of its application in engineering and architecture. International Journal of Architectural Computing 21, 5–22. doi:10.1177/14780771221089882.

[20] Heckel, R., Küster, J.M., Taentzer, G., 2002. Confluence of Typed Attributed Graph Transformation Systems, in: Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G. (Eds.), Graph Transformation (ICGT 2002), Springer. pp. 161–176. doi:10.1007/3-540-45832-8_14.

[21] Heckel, R., Taentzer, G., 2020. Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering. Springer International Publishing. doi:10.1007/978-3-030-43916-3.

[22] Hopcroft, J.E., Wong, J.K., 1974. Linear time algorithm for isomorphism of planar graphs, in: STOC, ACM. pp. 172–184.

[23] Ju, T., Schaefer, S., Warren, J., 2005. Mean value coordinates for closed triangular meshes, in: ACM SIGGRAPH 2005 Papers, Association for Computing Machinery. pp. 561–566. doi:10.1145/1186822.1073229.

[24] Kawarabayashi, K.i., Mohar, B., 2008. Graph and map isomorphism and all polyhedral embeddings in linear time, in: Proceedings of the fortieth annual ACM symposium on Theory of computing, pp. 471–480. doi:10.1145/1374376.1374443.

[25] Knupp, P.M., 2000. Achieving finite element mesh quality via optimization of the jacobian matrix norm and associated quantities. part i—a framework for surface mesh optimization. International Journal for Numerical Methods in Engineering 48, 401–420. doi:https://doi.org/10.1002/(SICI)1097-0207(20000530)48:3<401::AID-NME880>3.0.CO;2-D.

[26] Knuth, D.E., 1998. The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd ed., Addison-Wesley.

[27] Lienhardt, P., 1994. N-Dimensional generalized combinatorial maps and cellular quasi-manifolds. Inte. J. of Computational Geometry and Applications 4, 275–324.

[28] Lindenmayer, A., 1968. Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. Journal of Theoretical Biology 18, 300–315. doi:10.1016/0022-5193(68)90080-5.

[29] Marvie, J.E., Perret, J., Bouatouch, K., 2005. Fl-system: A functional l-system for procedural geometric modeling. The Visual Computer 21, 329–339. doi:10.1007/s00371-005-0289-z.

[30] Mens, T., Taentzer, G., Runge, O., 2005. Detecting structural refactoring conflicts using critical pair analysis. Electronic Notes in Theoretical Computer Science 127, 113–128. doi:https://doi.org/10.1016/j.entcs.2004.08.038. proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004).

[31] Meshkat, S., Talmor, D., 2000. Generating a mixed mesh of hexahedra, pentahedra and tetrahedra from an underlying tetrahedral mesh. International Journal for Numerical Methods in Engineering 49, 17–30. doi:https://doi.org/10.1002/1097-0207(20000910/20)49:1/2<17::AID-NME920>3.0.CO;2-U.

[32] Müller, P., Wonka, P., Haegler, S., Ulmer, A., Van Gool, L., 2006. Procedural modeling of buildings. ACM Trans. Graph. 25, 614–623. doi:10.1145/1141911.1141931.

[33] Pascual, R., Le Gall, P., Arnould, A., Belhaouari, H., 2022. Topological consistency preservation with graph transformation schemes. Science of Computer Programming 214. doi:10.1016/j.scico.2021.102728.

[34] Pellerin, J., Johnen, A., Verhetsel, K., Remacle, J.F., 2018a. Identifying combinations of tetrahedra into hexahedra: A vertex based strategy. Computer-Aided Design 105, 1–10. doi:10.1016/j.cad.2018.05.004.

[35] Pellerin, J., Verhetsel, K., Remacle, J.F., 2018b. There are 174 subdivisions of the hexahedron into tetrahedra. ACM Transactions on Graphics 37, 1–9. doi:10.1145/3272127.3275037.

[36] Prusinkiewicz, P., Hammel, M.S., Mjolsness, E., 1993. Animation of plant development, in: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, Association for Computing Machinery. pp. 351–360. doi:10.1145/166117.166161.

[37] Rozenberg, G., Salomaa, A., 1980. The Mathematical Theory of L Systems. Academic press.

[38] Sokolov, D., Ray, N., Untereiner, L., Lévy, B., 2016. Hexahedral-dominant meshing. ACM Trans. Graph. 35. doi:10.1145/2930662.

[39] Solnon, C., Damiand, G., de la Higuera, C., Janodet, J.C., 2015. On the complexity of submap isomorphism and maximum common submap problems. Pattern Recognition (PR) 48, 302–316. doi:10.1016/j.patcog.2014.05.019.

[40] Stiny, G., Gips, J., 1971. Shape Grammars and the Generative Specification of Painting and Sculpture, in: Freiman, C.V., Griffith, J.E., Rosenfeld, J.L. (Eds.), IFIP Congress 1971, pp. 1460–1465.

[41] Weiler, K., 1985. Edge-based data structures for solid modelling in curved-surface environments. Computer Graphics and Applications 5, 21–40.