CGoGN: n-dimensional Meshes with Combinatorial Maps

Pierre Kraemer, Lionel Untereiner, Thomas Jund,
Sylvain Thery, and David Cazier

ICube, Université de Strasbourg, CNRS
kraemer@unistra.fr

Summary. Many data structures are available for the representation and manipulation of meshes. In the context of algorithms that need to traverse local neighborhoods, topological structures are of particular interest. Many such existing structures are specialized for the representation of objects of a given dimension like surface or volume meshes. Many of them find their roots in combinatorial maps, a mathematical model for the representation of the topology of the subdivision of objects, which is consistently defined in any dimension. We present a practical implementation of combinatorial maps that competes with modern state-of-the-art data structures in terms of efficiency, memory footprint and usability. Among other benefits, developers can use a single consistent library to manipulate objects of various dimensions.

1 Introduction

Mesh data structures are of fundamental importance in many fields such as surface modeling, mesh generation, finite element analysis, geometry processing, visualization or computational geometry. A mesh is the cellular decomposition of geometric objects such as curves, surfaces or volumes. We are interested here in *topological models* that provide neighborhood relations between the cells of the decomposition (vertices, edges, faces, volumes) as this information is mandatory for many algorithms in the mentioned application fields. Our goal is to propose a library that is able to represent consistently objects of different dimensions composed of arbitrary cells (polygonal faces, polyhedral volumes). It should also: provide an efficient way to traverse the cells and their neighborhood; allow to store data with the cells (preferentially defined at execution time and not at compilation time); allow to efficiently modify the connectivity of the mesh even in highly dynamic settings. We do not aim to provide a generic framework in which the internal structure can be tailored for an application specific needs by avoiding for example unnecessary neighborhood relations or restricting representable cells to simplices.

A classical representation of cells and their incidence relations is a graph known as Incidence Graph [8]. It is an oriented graph whose nodes correspond to the cells and where each oriented arc connects a k-cell to a (k-1)-cell to which it is incident. A very wide variety of objects can be represented by such a graph: non-manifolds or manifolds, orientable or not, with or without boundary. Unfortunately, some neighborhood queries such as finding all the (k+1)-cells incident to a given k-cell are resolved at a prohibitive cost and require to traverse the whole structure. To obtain better performances, arcs of opposite direction connecting each k-cells to its incident (k+1)-cells are usually also stored.

However, many application fields only consider manifolds. The specialization to this more restricted domain allows to derive simpler and more efficient structures. Among those, some structures such as half-edges [19] for surfaces or facet-edges [6] for volumes have gained a huge popularity. Many publicly available C++ libraries like OpenMesh [1], `Surface_mesh` [17], OpenVolumeMesh [12] and the Polyhedron object in CGAL [4] are based on these structures. Even if they are efficient, they are all designed as separated libraries which causes difficulties to handle objects of different dimensions in a consistent way. From the user point of view, this separation implies a new learning process for each structure.

All these structures are actually equivalent and find their roots in the notion of **combinatorial map** described in 1960 by Edmonds [9]. It is a mathematically defined structure that represents the subdivision of surfaces. It has later been extended by Lienhardt to represent the subdivision of volumes [13]. At the same time, Brisson mathematically studied in [2] the ordering of cells around lower dimension cells, thus defining the *cell-tuple* algebraic structure. This structure is equivalent to the generalized combinatorial maps (G-maps) proposed by Lienhardt [14] and allows to represent the subdivision of n-dimensional manifolds.

These models are **dimension-independent** and rely on a single element along with a simple set of relations. All the information about the cells and their incidence and adjacency relations is modelized within this simple model. All **neighborhood traversals** are resolved in **optimal time** (linear in the number of traversed cells) without having to maintain any additional information. As in other topological models, there is a **total separation** between the **topological structure** of the subdivision and the **attributes** that can be associated to the cells.

The mathematical foundation of combinatorial maps mainly gave rise to its usage in theoretical fields such as formal specification [7] and geometrical theorem proving [3]. Recently, a CGAL [4] package implementing combinatorial maps has been released. However, as presented in Sect. 4, its performances are way beyond those of existing libraries. In order to promote the usage of combinatorial maps in a wider variety of communities, we present

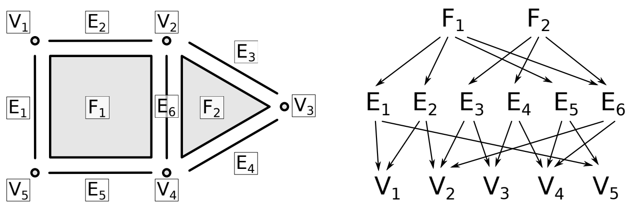


Fig. 1 A cellular decomposition of a 2-dimensional object and its incidence graph

the CGoGN /sigɔɲ/ library. It is a practical implementation of combinatorial maps that proposes an efficient index-based implementation that competes with modern existing libraries. Furthermore, extensions like multiresolution maps presented in [11, 18] are also available.

2 Combinatorial Maps

In this section we give a presentation of combinatorial maps and G-maps. Their definition is intuitively derived from the incidence graph structure. Fig. 1 shows a simple cellular decomposition of a 2-dimensional object and its incidence graph that we use as a running example in our definition.

2.1 From Incidence Graph to Cell-Tuples

In a n-dimensional cellular decomposition, a cell-tuple is defined as an ordered sequence of cells $(C_n, C_{n-1}, ..., C_1, C_0)$ of decreasing dimensions such that $\forall i, 0 < i \leq n$, C_i is incident to C_{i-1}. In other words, a cell-tuple corresponds to a path in the incidence graph from a n-cell to a vertex. Fig. 2 shows the iterative construction of all the cell-tuples generated by the cellular decomposition of Fig. 1.

Adjacency relations are defined on the cell-tuples: two cell-tuples are said to be i-adjacent if their path in the incidence graph share all but the i-dimensional cell. For example, (F_1, E_2, V_1) and (F_1, E_2, V_2) are 0-adjacent. In the context of the cellular decomposition of a quasi-manifold, it has been

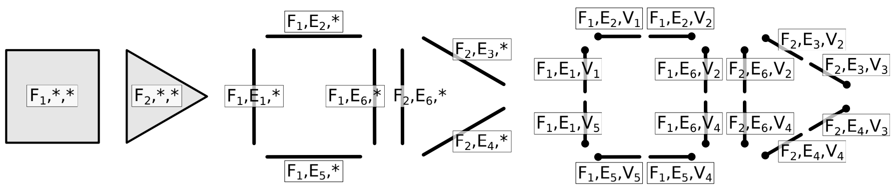


Fig. 2 Iterative construction of the cell-tuples corresponding to the cellular decomposition of Fig. 1

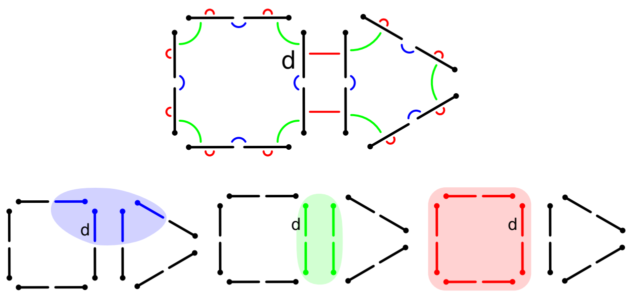


Fig. 3 The top figure shows the G-map corresponding to the cellular decomposition of Fig. 1. Darts are represented like the cell-tuples in Fig. 2; α_0, α_1 and α_2 functions are represented respectively by blue, green and red links. The three bottom figures illustrate the sets of darts representing the vertex, edge and face of dart d.

shown [2, 15] that these n+1 adjacency relations put the cell-tuples in a one-to-one relation (except for the n-adjacency at the boundary of the object where cell-tuples do not have any mate). Based on these definitions, G-maps have been proposed as a model for the representation of the cellular decomposition of n-dimensional quasi-manifolds.

2.2 Generalized Maps

Generalized maps encode a cellular decomposition with a set D of abstract elements called *darts* that are in one-to-one correspondance with the cell-tuples. A set of n+1 functions $\alpha_i : D \to D, 0 \leq i \leq n$ are defined based on the i-adjacency relations of the cell-tuples. Fig. 3 shows the G-map corresponding to the cellular decomposition of Fig. 1. Following the previously mentioned properties of i-adjacencies on cell-tuples, α_i functions are involutions, *i.e.* functions such that $\forall d \in D, \alpha_i(\alpha_i(d)) = d$. Combinatorial constraints express the correct assembly of cells along their boundary. For α_i functions, these constraints are expressed as follows: $\forall i, j, \quad 0 \leq i < i+2 \leq j \leq n, \quad \alpha_i \circ \alpha_j$ is an involution. If the G-map is constructed from the incidence graph of the decomposition of a quasi-manifold – a set of d-cells glued along (d-1)-cells – then those constraints are automatically satisfied.

The original cells are thus decomposed with their incidence relations into a *dimension-independent* set of a unique abstract entity. In order to bring back the notion of cell, the following two observations are a good starting point:

- each dart identifies a set of n cells of different dimension, *i.e.* those contained in the corresponding cell-tuple.
- each k-cell is represented by a set of darts, *i.e.* all the darts whose corresponding cell-tuple contains this cell.

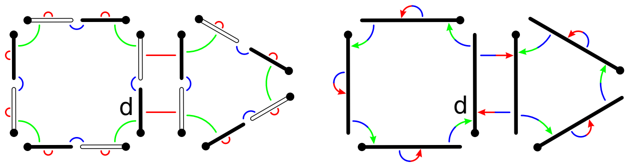


Fig. 4 Left: the darts of the G-map have been partitionned in two sets, each corresponding to one of the two orientations of the object. Right: the oriented combinatorial map yielded by dart d; $\phi_1 = \alpha_1 \circ \alpha_0$ and $\phi_2 = \alpha_2 \circ \alpha_0$ relations are represented respectively by blue-green and blue-red links.

Fig. 3 illustrates these notions. Starting from the same dart d, depending on how it is interpreted, one can build the set of darts that represent its vertex, edge or face. Within each of these sets, any dart equally represents the corresponding vertex, edge or face.

Now we only need a way to construct these sets of darts. Following the previous definitions, $\alpha_i(d)$ is the dart that represents the same cells as d except from the i-dimensional cell. All the other $\alpha_j, j \neq i$ functions will lead to darts that share the same i-cell as d. It follows that starting from a dart, the set of darts representing the same i-cell can be obtained by applying successively all the functions that maintain the i-dimensional cell unchanged, *i.e.* $\{\alpha_j, j \in \{0, 1, ..., i-1, i+1, ..., n\}\}$. Such sets of darts are formally defined as orbits, noted: $< \alpha_0, ..., \alpha_{i-1}, \alpha_{i+1}, ..., \alpha_n >$. For example in Fig. 3, the vertex, edge and face of d are defined respectively by the following orbits: $< \alpha_1, \alpha_2 > (d)$, $< \alpha_0, \alpha_2 > (d)$ and $< \alpha_0, \alpha_1 > (d)$.

2.3 Oriented Combinatorial Maps

A G-map is able to represent orientable or non-orientable quasi-manifolds. However, a representation domain restricted to orientable objects is sufficient in many application contexts. In this case, a more compact model can be used. The orientability of a given G-map can be determined with a binary coloring process of its darts following this rule: a dart of a given color can only be linked to darts of the other color. Starting with an arbitrary dart, if the whole object can be colored this way, then the object is orientable. In this case, the darts of the G-map are partitionned in two sets D^{black} and D^{white} of equal cardinality, each one representing one of the two orientations of the object. For any dart $d \in D, < \phi_1, ..., \phi_n > (d)$ with $\phi_i = \alpha_i \circ \alpha_0$ is the set of darts corresponding to the orientation yielded by d. For example, if $d \in D^{black}$, then $< \phi_1, ..., \phi_n > (d) = D^{black}$. Fig. 4 shows the application of this process.

As they represent the exact same object, keeping the two orientations of an orientable G-map is not necessary and one of these sets can be dropped, leading to a twice more compact representation. One orientation of an orientable

G-map is actually a combinatorial map, defined as a set of darts D along with n functions $\phi_i : D \to D, 1 \leq i \leq n$, with $\phi_i = \alpha_i \circ \alpha_0$. The ϕ_1 function is a permutation that links the ordered vertices around oriented faces. The $\phi_i, i \leq 2 \leq n$ functions are involutions, as stated by the constraint expressed above on the α_i involutions. From a constructive point of view, each of these involutions allows to glue pairs of i-dimensional cells along their common (i-1)-dimensional boundary cell.

The orbits that define the cells are the following. For cells of dimension $i \geq 1$, the sets of darts that represent the cells are defined by the orbit $< \phi_1, ..., \phi_{i-1}, \phi_{i+1}, ..., \phi_n >$. Like for G-maps, starting from any dart, all the functions that maintain the i-dimensional cell unchanged are applied. For vertices, the orbit is $< \phi_1 \circ \phi_2, ..., \phi_1 \circ \phi_n >$.

2.4 Embedded Maps

Maps and G-maps only define the *topology* of the cellular decomposition of quasi-manifolds, *i.e.* the cells – represented implicitly by sets of darts – and the neighborhood relations between them. In order to consistently attach attributes to cells, any data attached to a cell has to be linked to all the darts of this cell. The most flexible solution is to associate an index to each cell. Any data associated to this index is then associated to the corresponding cell.

To model this idea, additional functions can be defined on maps. For each $i, 0 \leq i \leq n, emb_i : D \to \mathbb{N}$, is the function that associates to each dart the index of its i-dimensional cell. In order for a map to be well embedded, the following constraint must be satisfied: $\forall d, d' \in D, d' \in orbit_i(d) \implies emb_i(d') = emb_i(d)$. In other words, for each dimension, all the darts of the same orbit are associated to the same index.

2.5 About Existing Data Structures

In the context of surface meshes, the half-edge data structure [10] is certainly the most widespread. Each half-edge stores a link to the next and previous half-edge within its oriented face, a link to the opposite half-edge of the adjacent face, a link to the incident vertex and a link to the incident face. Edges are usually not explicitly represented. Considering volume meshes, several data structures have been proposed such as facet-edge [6] or handle-face [16]. Basically, the incident face of the half-edge data structure is called a half-face and stores a link to an opposite half-face of the adjacent volume.

The half-edge data structure can actually be expressed as a 2-dimensional oriented map. Each half-edge is represented by a dart; the next half-edge link is encoded by ϕ_1; the previous half-edge link by ϕ_1^{-1} ; the opposite half-edge link by ϕ_2; the target (or source) incident vertex link by emb_0; the incident face link by emb_2. Data structures for volume meshes can be expressed as

3-dimensional maps. The latter can be seen as 2-dimensional maps glued along common faces by ϕ_3 involution. It encodes the opposite half-face link but without needing an extra entity (*i.e.* the half-face) to store the relation.

The *fundamental difference* of combinatorial maps is that **a dart is not the half of an edge**. Following our presentation of combinatorial maps, a dart is to be considered as a cell-tuple. This implies that the model is able to represent consistently objects of any dimension and that each dart represents at the same time a vertex, an edge, a face, a volume, ... of the mesh. Actual cells are explicited only through their indexing and associated attributes. The indexing of the cells of any dimension is completely optional. If the cells of one or even all dimensions are not indexed, the cellular decomposition and its topology are still completely defined. Indeed, cell enumeration and neighborhood traversals are performed using exclusively the darts and their relations.

3 Implementation

CGoGN (**C**ombinatorial and **G**eometric m**o**deling with **G**eneric **N**-dimensional Maps) [5] is a C++ library that provides an efficient index-based implementation of n-dimensional combinatorial maps and G-maps.

3.1 Basic Containers

Maps and G-maps both rely on darts as their basic entity. According to the model and dimension, each dart has to store a variable number of topological relations (links to other darts) and indices of the embedded cells. For each embedded dimension, a variable number of custom attributes have to be associated to the indices of the cells.

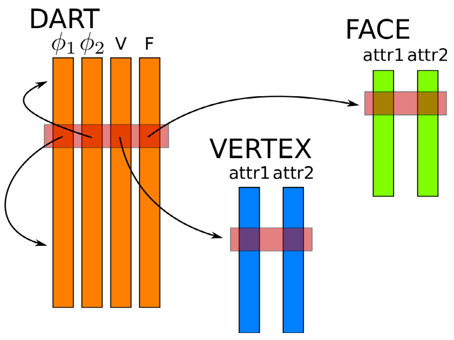


Fig. 5 Containers of a 2-dimensional combinatorial map with vertex and face attributes. The dart container stores ϕ_1 and ϕ_2 links to other darts and V and F links to the embedded vertices and faces.

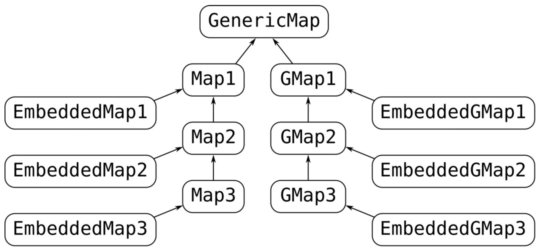


Fig. 6 Map and G-map classes inherit from a common generic class that manages basic data structures. Embedded versions of these classes ensure the proper embedding of the maps during topological operations.

Following these requirements, a map can be encoded as a set of attribute containers: one for the darts and one for each embedded dimension. Each container stores a set of attributes of any type that can be dynamically added or removed. All the attributes of a given container have the same number of records, each record being identified by an index. In particular, darts are identified by indices. Fig. 5 illustrates the containers of a 2-dimensional map with embedded vertices and faces. The dart container has a ϕ_1 and a ϕ_2 attribute that store for each dart d the indices of $\phi_1(d)$ and $\phi_2(d)$ within the dart container. It also has `V` and `F` attributes that store for each dart the indices of its associated vertex and face within the vertex and face attributes container.

We have chosen to store attributes in chunk arrays which are sets of chained arrays. It allows to allocate additional memory while leaving all existing elements in place. Entries that are marked as deleted are automatically skipped and used in priority when a free index is requested. The size of a chunk can be parameterized to balance between memory fragmentation and early memory allocation. Access to the n^{th} cell of the array is achieved by division and modulo operations to retrieve the good chunk and the good index within this chunk (the chunk size is a power of 2 to optimize those arithmetic operations). As we discuss in Sect. 4, this memory management policy has several advantages, in particular when representing meshes with dynamic connectivity.

3.2 Generic Map and Maps Hierarchy

The containers are managed by an object called `GenericMap` that is the base class of all map objects. Fig. 6 shows the hierarchy of map classes. `[G]Map1`, `[G]Map2` and `[G]Map3` represent respectively 1, 2 and 3-dimensional [G-]maps. The constructor of each class creates the dart attributes corresponding to the topological relations of its model. For example, `Map1` creates the ϕ_1 attribute, `Map2` adds the ϕ_2 attribute, and so on. They provide accessors to these relations, and a set of classical operators. For example, `Map2` provides operators such as `newFace`, `cutEdge`, `flipEdge` or `splitFace`. To allow the development of generic algorithms, `Map` and `GMap` provide the same set of operators. The `Embedded[G]Map{1,2,3}` classes overload the topological operators and ensure that the maps remain properly embedded (see Sect. 2.4). Listing 1 shows how our running example is created in an embedded 2-dimensional map.

Listing 1. Initialization of our running example in a 2-dimensional map.

```
EmbeddedMap2 map;
Dart square = map.newFace(4);
Dart triangle = map.newFace(3);
map.sewFaces(square, triangle);
```

3.3 Attributes Management

Attributes are accessed through `AttributeHandler` objects. Such objects are defined by the data type of the attribute and the dimension of the cell. Specialized classes (like `VertexAttribute`) parameterized only by the data type are available for a more concise syntax. Upon creation of the first attribute of a given cell, the corresponding container is initialized and a new attribute is added in the dart container to store the indices.

The `GenericMap` provides functions to add, remove and get attributes as illustrated in listing 2. Read and write access to the values of an attribute for a given cell is achieved using the bracket operator. The parameter of this bracket operator can be either an index or a dart: an index provides direct access to the corresponding entry; a dart gives access to the entry pointed by the dart's embedding for this dimension. The end of the listing shows a loop over the darts of the triangle face in order to set the position and normal of all its vertices. Of course, as we detail in the following, a more convenient way to traverse local neighborhoods is provided.

Listing 2. Creation, removal, access to attributes with AttributeHandler objects.

```
VertexAttribute<Vec3> pos = map.addAttribute<Vec3, VERTEX>("position") ;
VertexAttribute<Vec3> normal = map.addAttribute<Vec3, VERTEX>("normal") ;
FaceAttribute<float> area = map.addAttribute<float, FACE>("area") ;
FaceAttribute<Vec4> color = map.addAttribute<Vec4, FACE>("color") ;

map.removeAttribute(color) ;

VertexAttribute<float> kmax = map.getAttribute<float, VERTEX>("kmax") ;
if(!kmax.isValid())
    // the attribute does not exist

area[triangle] = 2.0f ;

Dart d = triangle ;
do
{
    pos[d] = Vec3(...) ;
    normal[d] = Vec3(...) ;
    d = map.phi1(d) ;
} while(d != triangle) ;
```

3.4 Markers

The CGoGN library provides a marking mechanism that is needed by many algorithms. Any entry of any container can be marked. A `CellMarker` can be declared for any cell dimension by giving the appropriate template parameter. When a `CellMarker` is instanciated, a free marker is reserved. It is automatically released at the destruction of the object. As shown in listing 3, a `CellMarker` can `mark`, `unmark` and check if a cell is marked. Similarly to the bracket operator of `AttributeHandler`, these functions take as parameter either an index or a dart.

The `DartMarker` class is dedicated to the marking of darts. A `DartMarker` also disposes of the `mark`, `unmark` and `isMarked` functions. The difference lies in the possibility to mark or unmark any orbit of a given dart at once using the `markOrbit` function. As the traversal of an orbit is different according to the dimension of the map and the underlying model (map or G-map), this function uses a generic orbit traversal. This is achieved using C++ polymorphism: each map or G-map class provides a set of `foreach_dart_of_XXX(d,f)` functions that apply a functor `f` on all the darts of the `XXX` orbit of `d`. The appropriate function is used by the `DartMarker` to mark the orbit according to the actual type of the map.

Listing 3. Markers usage.

```
CellMarker<FACE> cm(map) ;
cm.mark(square) ;
if(cm.isMarked(square))
    cm.unmark(square) ;

DartMarker dm(map) ;
dm.markOrbit<VERTEX>(triangle) ;
```

3.5 Boundary

Maps and G-maps can handle boundaries by the means of fixed point topological relations. For example, in a 2-dimensional map, the ϕ_2 relation of the darts of boundary edges can point to themselves like illustrated in Fig. 4. The same idea holds in any dimension: in a volume mesh, darts of the boundary faces are fixed points of the ϕ_3 relation.

However, introducing such fixed points alter the integrity of some basic traversals. Let's take the example of the traversal of all edges incident to a vertex in a 2-dimensional map. In Fig. 7, the darts of the vertex of `d` are highlighted in bold. The standard way of traversing this orbit is to compose the ϕ_1 and ϕ_2 relations to go from one dart to the next around the vertex in an ordered way. This simple and efficient process is not applicable in the presence of fixed points. Even if the traversal is tweaked to take the fixed points into account, it misses one of the incident edges (the valence of the vertex is 3 but it is composed of only 2 darts).

In CGoGN, we have chosen to represent only closed maps (*i.e.* without any fixed point). Objects boundaries are represented using a special boundary marker. 2-dimensional maps can feature boundary faces, and 3-dimensional maps can feature boundary volumes. The right image of Fig. 7 shows how the same example mesh is represented with an additional boundary face in orange. The vertex of `d` is now composed of 3 darts and the traversal of incident edges can be done efficiently without missing any edge.

3.6 Global Traversals

The CGoGN library provides a convenient way to traverse all the elements of the represented maps. The simplest global traversal consists in traversing all the darts of the map, as illustrated in listing 4.

Listing 4. Darts traversal.

```
for (Dart d = map.begin(); d != map.end(); map.next(d))
{
    // do something with d
}
```

Any container can be traversed in the same way. For example, going through all valid indices of the vertex container and accessing the data of some of its attributes is achieved like shown in listing 5. A single attribute can also be traversed by using directly the `begin`, `end` and `next` functions that are available on the `AttributeHandler`.

Listing 5. Container traversal.

```
AttributeContainer& cont = map.getAttributeContainer<VERTEX>() ;
for (unsigned int i = cont.begin(); i != cont.end(); cont.next(i))
{
    position[i] = ... ;
    normal[i] = ... ;
}
```

A fundamental global traversal consists in passing through each topological cell exactly once. This can be performed using a global traversal on darts combined with an appropriate marking. The CGoGN library provides a class called `TraversorCell` to perform this kind of traversals. During the traversal, each topological cell is visited through one of its darts. No assumption can be

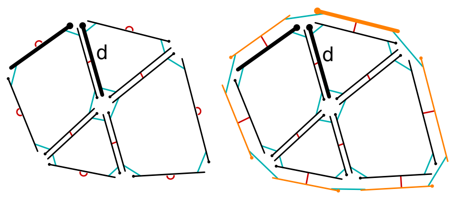


Fig. 7 The CGoGN library only represents closed meshes. Boundaries are managed through the marking of boundary cells.

done on which dart of the concerned orbit is returned. As they are not part of the actual mesh, boundary cells are not returned by these objects. In listing 6, all the vertices of the map are traversed. The `MAP` template parameter defines the type of the `map` object.

Listing 6. Cells traversal.

```
TraversorCell<MAP, VERTEX> trav(map) ;
for (Dart d = trav.begin(); d != trav.end(); d = trav.next())
{
    // do something with d
}
```

`TraversorCell` objects make an internal use of markers. If the traversed cell is currently embedded, a `CellMarker` is used. Otherwise a `DartMarker` is used. This choice is done for performance reasons as there are much less indiced cells to mark and unmark than darts (for example 6 darts for 1 vertex in a typical 2-dimensional map).

In the context of meshes with fixed connectivity, a much faster way to traverse the mesh is provided. The `enableQuickTraversal<CELL>` function creates an attribute in the `CELL` container that stores one dart for each cell of the mesh. If it is available, a corresponding `TraversorCell` will automatically iterate on this attribute and return successively a dart of each cell without having to traverse all the darts of the map or to mark/unmark anything, leading to a huge speed-up of the traversal. Topological operators that modify the connectivity of the mesh do not update this attribute. However, it can be manually updated by a call to `updateQuickTraversal<CELL>`, for example after a subdivision or simplification process. If this feature is not desired anymore, it can be disabled with a call to `disableQuickTraversal<CELL>` which removes the corresponding attribute.

3.7 Neighborhood Traversals

The CGoGN library provides a set of objects to handle neighborhood traversals. Starting from a given cell, there are two kinds of neighborhood traversals: incident cells and adjacent cells. In the following, the letters V, E, F and W respectively stand for vertex, edge, face and volume.

Incident Cell Traversals

A cell b is said to be incident to a cell a if b belongs to the boundary of a or if a belongs to the boundary of b. `TraversorDXY` objects are provided with D the dimension of the traversed map, $X \in \{V, E, F, W\}$ the type of starting cell and $Y \in \{V, E, F, W\} \setminus \{X\}$ the type of the aimed incident cells. Listing 7 illustrates the traversal of all the faces incident to the vertex of a given dart `d` in a 2-dimensional map.

Listing 7. Traversal of all faces incident to a vertex.

```
Traversor2VF<MAP> trav(map, d) ;
for(Dart it = trav.begin(); it != trav.end(); it = trav.next())
{
    // do something with it
}
```

In dimension 2, all these traversals go through a set of incident cells that are ordered around the central cell. They are clearly executed in a time linear in the number of traversed cells. Fig. 8 shows an example mesh for some 2-dimensional incident cell traversals. In dimension 3, incident cell traversors centered on an edge or a face are ordered, while those centered on a vertex or a volume are not. Non-ordered traversals are resolved using a local flooding algorithm with marking that is also linear in the number of reached cells. In this case, no assumption can be made on the order in which the incident cells are returned. Fig. 9 shows some of the 3-dimensional neighborhood traversals on a regular hexaedral grid. In both cases, boundary cells are not returned by `TraversorDXY` objects.

Adjacent Cell Traversals

A cell b is said to be adjacent to a cell a by a cell c if $dim(a) = dim(b) \neq dim(c)$ and $\exists c$ such that a is incident to c and c is incident to b. The CGoGN library provides `TraversorDXXaY` objects where D is the dimension of the traversed map, $X \in \{V, E, F, W\}$ is the type of starting and aimed cells and $Y \in \{V, E, F, W\} \setminus \{X\}$ is the type of the cell through which the adjacency is defined.

Listing 8. Traversal of all vertices adjacent to a vertex through an edge.

```
Traversor2VVaE<MAP> trav(map, d) ;
for(Dart it = trav.begin(); it != trav.end(); it = trav.next())
{
    // do something with it
}
```

Listing 8 illustrates the traversal of all the vertices adjacent to the vertex of a given dart `d` through a common edge in a 2-dimensional map. As for inci-

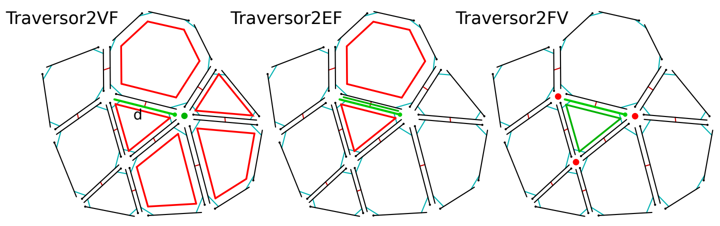


Fig. 8 Some 2-dimensional incident cell traversals. All cases are initialized with the same dart `d` (in green) interpreted as a vertex (left), an edge (middle) and a face (right). A dart of each reached cell (in red) is successively returned by the `Traversor2XY`.

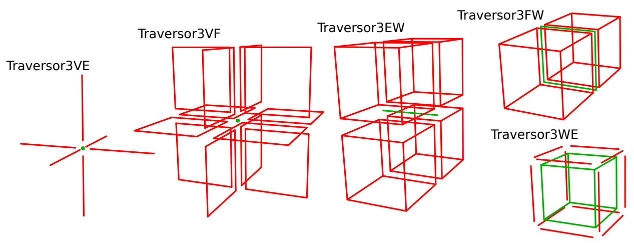


Fig. 9 Some 3-dimensional incident cell traversals

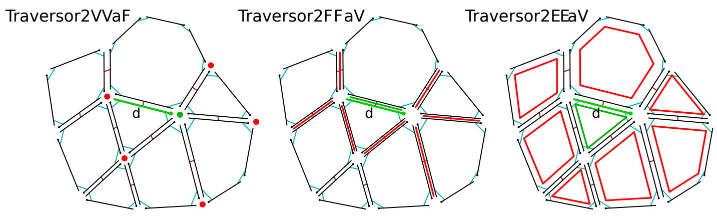


Fig. 10 2-dimensional adjacent cell traversals. All cases are initialized with the same dart `d` (in green) interpreted as a vertex (left), an edge (middle) and a face (right). A dart of each reached cell (in red) is successively returned by the `Traversor2XXaY`.

dence queries, adjacency queries are resolved in a time linear in the number of traversed cells. Fig. 10 shows an example mesh for some 2-dimensional adjacent cell traversals. Boundary cells are also not returned by `TraversorDXaYY` objects.

Quick Traversals

Following the same approach than for global traversals, quick traversals can be enabled for any of the incidence or adjacency neighborhood traversals. For example, a call to `enableQuickIncidenceTraversal<MAP,C1,C2>` creates an attribute in the C1 container that stores for each cell a set of darts containing one dart per incident C2 cell. If it is available, this attribute is automatically used by the corresponding `TraversorDXY`, leading to a huge improvement of traversal performances. This feature is of course more adapted to the case of a static connectivity. The attribute can be updated or removed when desired by a call to the appropriate `update` or `disable` function.

4 Comparison

In this section, we compare the CGoGN library with other widely used data structures: CGAL (Polyhedron and Combinatorial Maps), OpenMesh and `Surface_mesh` for surface meshes representation; CGAL (Combinatorial

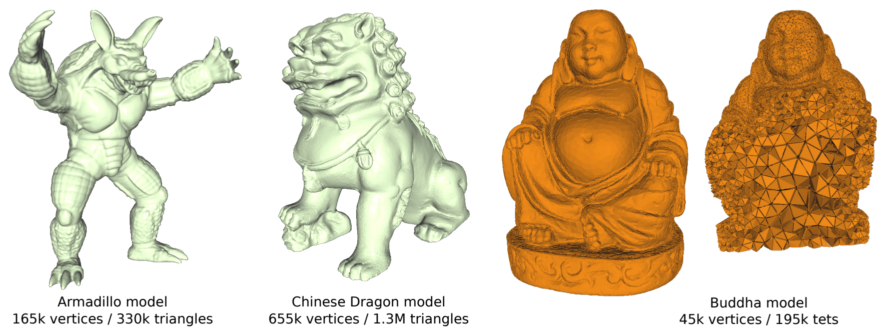


Fig. 11 Surface and volume meshes used in the evaluation

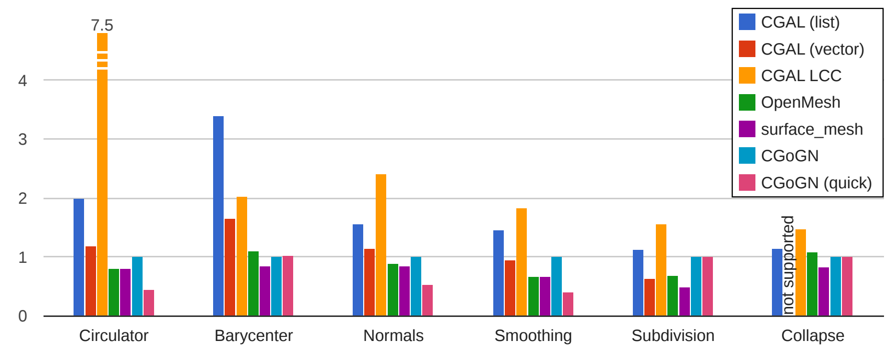
Maps) and OpenVolumeMesh for volume meshes representation. To evaluate and compare the performances of CGoGN to other structures, we use the benchmarks proposed in [17] and [12]. They consist in algorithms that evaluate fundamental features of these libraries such as iterating through elements of the mesh, neighborhood queries or connectivity modifications.

In the context of surface meshes, we use the following tests:

- *Circulator*: for each vertex, enumerate its incident faces then for each face, enumerate its vertices.
- *Barycenter*: center the mesh at the origin by first computing its barycenter and then substracting it from each vertex.
- *Normal*: compute and store face normals, then compute and store vertex normals as an average of incident faces normals.
- *Smoothing*: move each non-boundary vertex to the barycenter of its adjacent vertices.
- *Subdivision*: perform a $\sqrt{3}$ subdivision step by splitting faces at their center, computing old vertices positions and flipping old edges.
- *Collapse*: split all faces at their center and then restore the original connectivity by collapsing each new vertex into one of its neighbours.

In the context of volume meshes, we use the following tests:

- *Circulator*: for each vertex, enumerate its incident volumes then for each volume, enumerate its vertices.
- *Circulator2*: for each vertex, enumerate its vertices adjacent through a common volume.
- *Barycenter*: compute and store the barycenter of each volume.
- *Smoothing*: move each vertex to the barycenter of its adjacent vertices.
- *Subdivision*: perform a 1-to-4 split of each volume of a tetrahedral mesh by inserting a central vertex.
- *Collapse*: perform a series of edge collapses by selecting each time the shortest edge of the mesh.



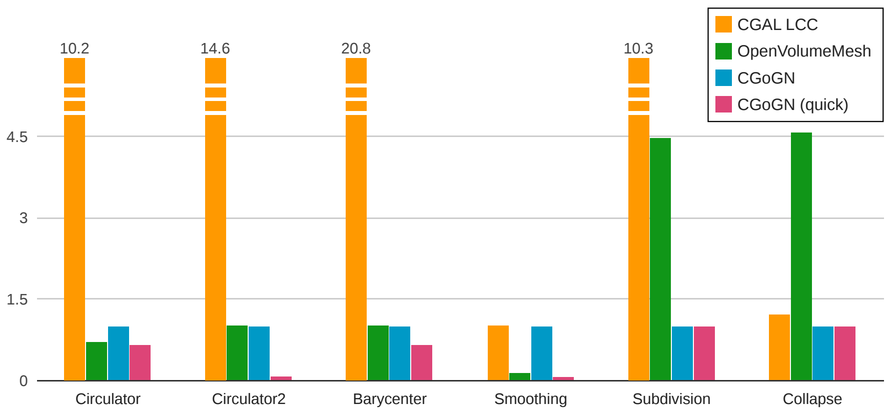


Fig. 12 Time performance for the surface (top) and volume (bottom) benchmarks. Values are expressed relatively to the performance of CGoGN. CGAL (list) and CGAL (vector) are the list-based and array-based versions of the Polyhedron object in CGAL. CGAL LCC is the Linear Cell Complex structure of CGAL, *i.e.* a CGAL Combinatorial Map with a point associated to each vertex. CGoGN (quick) is CGoGN library with appropriate incidence or adjacency quick traversals enabled.

Fig. 11 shows the surface and volume meshes used in our benchmarking process. The obtained performances are presented in Fig. 12.

In the surface benchmarks, performances of CGoGN library are generally close to the most efficient. The CGAL (list) and CGAL LCC data structures, the only ones that are not index-based, obtain the worst performances. When it is possible, activating the appropriate incidence or adjacency quick traversals improves CGoGN performance beyond that of other libraries.

In the volume benchmarks, CGAL LCC data structure still obtains globally poor performances. The *Circulator* test uses incidence relations that are stored in the OpenVolumeMesh (OVM) data structure and traversed dynamically in CGoGN library. When activating quick traversals, the performances of CGoGN library are equivalent or even better than that of the OVM data structure. The same observation can be made on the *Smoothing* test. The

Circulator2 test uses adjacency relations that are not directly available in the OVM data structure. In this case, the OVM data structure and CGoGN library obtain similar performances. When activating quick traversals, the CGoGN library largely outperforms the OVM data structure. In algorithms that modify the connectivity of the mesh, the CGoGN library performs better than other tested libraries.

An advantage that can be pointed out is that combinatorial maps do not force the storage of any particular incidence relation. All the topological information is encoded within darts and their relations. Neighborhood caching is proposed optionally by activating the desired quick traversals. As a result, all possible incidence or adjacency queries can always be computed, and any of them can be accelerated on demand.

In the context of interactive applications like physical simulation, view-dependant refinement or modelization tools, the represented meshes often present a highly dynamic connectivity where topological cells are regularly created and deleted. The memory management policy of the CGoGN library, using chunk arrays and reusing deleted entries (see 3.1), makes it particularly adapted to this context. Data structures like OpenMesh or `Surface_mesh` rely on one-block contiguous containers. The holes created by the deletion of elements are not automatically reused and new elements are always created at the end of the containers after an appropriate memory pre-allocation (which causes a recopy of all existing elements). In the case of dynamic meshes, the amount of memory to pre-allocate in these structures is hard to anticipate. As new elements are added at the end of the containers, the available capacity is going to be reached eventually. Furthermore, as holes are created in the containers, the global performance of traversals is reduced along with the contiguity of active elements. Garbage collection algorithms are proposed to compact the elements in the containers and recover good performances. However, this operation also frees unused memory, which is going to be re-allocated again for further cell creations. Moreover, garbage collection is a costly process whose optimal triggering is not obvious.

We have conducted benchmarks consisting in repeatedly applying batches of splits and collapses to point out these behaviors. We observed that unlike other libraries, the performances and memory usage of the CGoGN library in this context remain constant.

5 Conclusion

The CGoGN library provides an efficient and easy to use implementation of combinatorial maps. It allows to represent and manipulate objects of different dimensions within a common framework. Algorithms written for maps of a given dimension can even benefit from the features of maps of lower dimensions. The total separation between the topological structure of the mesh and the cells attributes allows a flexible and optional management of the latter.

Based on [11, 18], the CGoGN library also allows the represention and manipulation of multi-resolution meshes. Given all these features, the proposed index-based underlying structure allows performances that are comparable or better than those of existing libraries. The memory management policy makes it particularly well suited to applications where the connectivity of the mesh is highly variable.

We already developed several high-level applications based on the CGoGN library like surface deformation, remeshing, progressive meshes or volume meshes subdivision. We are also developping a plugin-based application that allows to share CGoGN based developments into one common place.

References

1. Botsch, M., Steinberg, S., Bischoff, S., Kobbelt, L.: Openmesh - a generic and efficient polygon mesh data structure. In: Proceedings of the OpenSG Symposium (2002)
2. Brisson, E.: Representing geometric structures in *d* dimensions: Topology and order, Saarbrücken, Germany, pp. 218–227 (1989)
3. Brun, C., Dufourd, J.-F., Magaud, N.: Designing and proving correct a convex hull algorithm with hypermaps in coq. Computational Geometry 45(8), 436–457 (2012)
4. CGAL. Computational geometry algorithms library, `http://www.cgal.org`
5. CGoGN. Combinatorial and geometric modeling with generic n-dimensional maps, `http://cgogn.unistra.fr`
6. Dobkin, D.P., Laszlo, M.J.: Primitives for the manipulation of three-dimensional subdivisions. In: Proceedings of 3rd ACM Symposium on Computional Geometry, Waterloo, Ontario, Canada, pp. 86–99 (1987)
7. Dufourd, J.-F.: Formal specification of topological subdivisions using hypermaps. Computer-Aided Design 23(2), 99–116 (1991)
8. Edelsbrunner, H.: Algorithms in combinatorial geometry. Springer (1987)
9. Edmonds, J.R.: A combinatorial representation for polyhedral surfaces. Notices of the American Mathematical Society 7 (1960)
10. Kettner, L.: Using generic programming for designing a data structure for polyhedral surfaces. Computational Geometry - Theory and Applications (13), 65–90 (1999)
11. Kraemer, P., Cazier, D., Bechmann, D.: Extension of half-edges for the representation of multiresolution subdivision surfaces. The Visual Computer 25(2), 149–163 (2009)
12. Kremer, M., Bommes, D., Kobbelt, L.: OpenVolumeMesh – A versatile index-based data structure for 3D polytopal complexes. In: Jiao, X., Weill, J.-C. (eds.) Proceedings of the 21st International Meshing Roundtable, vol. 123, pp. 531–548. Springer, Heidelberg (2013)
13. Lienhardt, P.: Extension of the notion of map and subdivisions of a three-dimensional space. In: 5th Symposium on Theoretical Aspects in Computer Science, Bordeaux, France, pp. 301–321 (1988)
14. Lienhardt, P.: Subdivision of n-dimensional spaces and n-dimensional generalized maps. In: 5th ACM Conference on Computational Geometry, Saarbrücken, Germany, pp. 228–236 (1989)

15. Lienhardt, P.: N-dimensional generalized combinatorial maps and cellular quasi-manifolds. Journal on Computational Geometry and Applications 4(3), 275–324 (1994)
16. Lopes, H., Tavares, G.: Structural operators for modeling 3-manifolds. In: Proceedings of the 4th ACM symposium on Solid Modeling and Applications, SMA 1997, pp. 10–18. ACM (1997)
17. Sieger, D., Botsch, M.: Design, implementation, and evaluation of the surface_mesh data structure. In: Quadros, W.R. (ed.) Proceedings of the 20th International Meshing Roundtable, vol. 90, pp. 533–550. Springer, Heidelberg (2011)
18. Untereiner, L., Cazier, D., Bechmann, D.: n-dimensional multiresolution representation of subdivision meshes with arbitrary topology. Graphical Models 75(5), 231–246 (2013)
19. Weiler, K.: Edge-based data structures for modeling in curved-surface environments. IEEE Computer Graphics & Applications 5(1), 21–40 (1985)