A Generic Implementation of dD Combinatorial Maps in CGAL
Guillaume Damiand, Monique Teillaud

To cite this version:


HAL Id: hal-01090011
https://inria.hal.science/hal-01090011
Submitted on 8 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A generic implementation of $dD$ combinatorial maps in CGAL

Guillaume Damiand$^{a,\ast}$, Monique Teillaud$^b$

$^a$Université de Lyon, LIRIS, UMR 5205 CNRS, Villeurbanne, France
$^b$INRIA Sophia Antipolis - Méditerranée, France

Abstract

We present a generic implementation of $dD$ combinatorial maps and linear cell complexes in CGAL, the Computational Geometry Algorithms Library. A combinatorial map describes an object subdivided into cells; a linear cell complex describes the linear geometry embedding of such a subdivision. In this paper, we show how generic programming and new techniques recently introduced in the C++11 standard allow a fully generic and customizable implementation of these two data structures, while maintaining optimal memory footprint and direct access to all information. We compare our implementation with existing 2D and 3D libraries implementing cellular structures, and illustrate its usage by two applications. To the best of our knowledge, the CGAL software package presented here offers the only available generic implementation of combinatorial maps in any dimension.

Keywords: Combinatorial map; Linear cell complex; CGAL; Generic programming; C++11

1. Introduction

Data structures describing subdivisions of objects have been extensively studied. They can be classified into two types: (1) Regular subdivisions, for example using triangles in 2D, tetrahedra in 3D, or quadrangles in 2D; (2) Irregular subdivisions, where cells can be of different types. For regular subdivisions, many data structures exist, as it is often enough to store cells of maximal dimension and to represent adjacency relations between cells by pointers. Such data structures are easy to implement even in an arbitrary dimension.

Irregular subdivisions are more complex. In 2D, the well known winged edge data structure [2,26] is often used in computer graphics; there are several variants of winged edges, among which the halfedge data structure [14,15] has an implementation in the CGAL library [10,25]. However, there are few data structures allowing to describe irregular subdivisions in arbitrary dimension. The incidence graph [8] represents each cell as a node, with an arc between each pair of nodes corresponding to two incident cells whose dimensions differ by one. This data structure is simple, but it cannot represent multi-incidence relations, which often occur in real applications. Moreover the incidence graph is not ordered, e.g., it does not allow to iterate through all the vertices of a face in a given order, which leads to complex and inefficient operations.

$\ast$ Corresponding author.

E-mail address: guillaume.damiand@liris.cnrs.fr; monique.teillaud@inria.fr

1877-7058 © 2014 The Authors. Published by Elsevier Ltd.
Peer-review under responsibility of organizing committee of the 23rd International Meshing Roundtable (IMR23).
Combinatorial maps [18] address these issues by storing all cells of the subdivision and all incidence relations. They are a generalization of the halfedge data structure. They are based on a unique basic type of element called dart, together with relations between these darts. Combinatorial maps have many advantages: they allow local modifications, they are fully ordered, they enable the representation of multi-incidence relations, they allow many operations, they are independent of the geometry, i.e., they can represent curved objects as well as linear objects.

All these properties make combinatorial maps an ideal tool in many applications. Several software packages were developed to describe 2D and 3D objects as combinatorial maps [11,16,17,21]. However, to the best of our knowledge, the Combinatorial Maps package, first release as part of CGAL in 2011 [5], was the first software capable of describing and handling subdivided objects in an arbitrary dimension thanks to a generic implementation of combinatorial maps in any dimension. The “Linear cell complexes” package, released a few months later [6], provides users with a geometric embedding. Some of the initial ideas used in the implementation were inspired by the design of the CGAL halfedge data structure package [14], which dates back to 1998.

The goal of this paper is to show how new techniques introduced in the C++11 standard [24] lead to a fully generic implementation, in any dimension, allowing customization by users, while using optimal memory, i.e., using the minimal number of pointers required to efficiently associate some information to some specific cells. Note that of course other programming languages propose similar techniques and could be used instead of C++11.

In the next two sections, we recall definitions of combinatorial maps and linear cell complexes and we introduce basic tools used in our implementation. In Section 4 we present the implementation of these two mathematical models. Then we compare our solution with other software in Section 5 and we show two use cases in different applications in Section 6. We conclude and give some perspectives in Section 7.

2. Definitions

A subdivided object\(^1\) in dimension \(d\) is described as a set of cells from dimension 0 (vertices) to dimension \(d\), plus relations between these cells (a cell of dimension \(i\) is denoted \(i\)-cell). Two cells \(c_1\) and \(c_2\) are said to be incident if one is part of the boundary of the other. Two \(i\)-cells are adjacent when their respective boundaries share a common \((i-1)\)-cell.

Examples of 2D and 3D subdivided objects are given in Fig. 1. The object depicted in Fig. 1(a) is composed of three faces (2-cells), nine edges (1-cells) and seven vertices (0-cells). For example, vertex \(v1\) is incident to edges \(e1\) and to face \(f1\), edge \(e1\) is incident to face \(f1\). Edges \(e1\) and \(e2\) are adjacent (along vertex \(v1\)) and faces \(f1\) and \(f2\) are adjacent (along edge \(e1\)). Fig. 1(b) illustrates a three-dimensional object with three volumes (3-cells) \(vol1\), \(vol2\) and \(vol3\), twelve faces (2-cells), sixteen edges (1-cells), and eight vertices (0-cells). Only one face, \(f4\), separates the two

---

\(^1\) More precisely, we consider quasi-manifolds which are assemblies of \(n\)-cells along \((n-1)\)-cells. See [19] for a formal definition.
two volumes \( vol 1 \) and \( vol 2 \). For example, vertex \( v2 \) is incident to edge \( e4 \), to face \( f4 \) and to volume \( vol 1 \) and \( vol 2 \). The two volumes \( vol 1 \) and \( vol 2 \) are adjacent along face \( f4 \).

Definition 1 below formally defines a \( d \)-dimensional combinatorial map, but let us start with an intuitive introduction. Roughly speaking, a combinatorial map is an edge-based data structure: A dart is a ‘part’ of an edge (similarly to a halfedge, which is a ‘part’ of an edge in a halfedge data structure), plus a ‘part’ of all its incident \( i \)-cells, \( i \in \{0, 2, \ldots, d\} \).

Let us now give the precise definition. See [4,18] for a complete presentation.

**Definition 1 \((d\text{-map}[4])\).** A \( d \)-dimensional combinatorial map, or \( d \)-map, with \( 0 \leq d \), is a \((d + 1)\)-tuple \( M = (D, \beta_1, \ldots, \beta_d) \) where:

1. \( D \) is a finite set of darts;
2. \( \beta_1 \) is a partial permutation\(^2\) on \( D \), and we denote \( \beta_0 = \beta_1^{-1} \);
3. \( \forall i, 2 \leq i \leq d: \beta_i \) is a partial involution\(^1\) on \( D \);
4. \( \forall i, 0 \leq i \leq d - 2, \forall j, 3 \leq j \leq d, i + 2 \leq j; \beta_i \circ \beta_j \) is\(^4\) a partial involution.

The last line of the definition expresses the conditions to ensure the validity of the combinatorial map. Intuitively it ensures that two \( i \)-cells are adjacent along an entire \((i - 1)\)-cell. This condition is the combinatorial analog of the manifold property defined for topological spaces.

Fig. 2 depicts the combinatorial maps representing the two objects of Fig. 1. In 2D (Fig. 2(a)), an edge is composed of two darts (for example edge \( e1 \) has two darts \{7, 8\}) except for edges that belong to the boundary of the object and are incident to only one face (for example edge \( e4 \) has one dart \{1\}). Each dart belongs to a 0-cell, a 1-cell and a 2-cell. For example dart 3 belongs to vertex \( v1 \), edge \( e2 \) and face \( f3 \). In 3D (Fig. 2(b)), an edge has as many darts as the number of its incident volumes, and each dart belongs to a 0-cell, a 1-cell, a 2-cell and a 3-cell. When a dart \( \delta \) belongs to an \( i \)-cell \( c \) having no adjacent \( i \)-cell along \( \delta \), then \( \delta \) is said to belong to an \( i \)-boundary and in such a case, \( \beta_i(\delta) = \emptyset \).

For example in Fig. 2(a), we have \( \beta_2(1) = \emptyset \).

An \( i \)-cell is implicitly represented by the set of all darts describing a part of this cell. We can retrieve cells using the \( \beta \) maps: indeed, as \( \beta_i(\delta) \) gives a dart that belongs to the same cells except for vertices and \( i \)-cells, if we use all the

---

\(^2\) A partial permutation \( f \) on \( D \) is a map from \( D \cup \{\emptyset\} \) to \( D \cup \{\emptyset\} \) such that \( \forall e_1 \in D, \forall e_2 \neq e_1 \in D, f(e_1) \neq \emptyset \) and \( f(e_2) \neq \emptyset \Rightarrow f(e_1) \neq f(e_2) \).

\(^1\) A partial involution \( f \) on \( D \) is a partial permutation on \( D \) such that \( \forall e \in D, f(e) \neq \emptyset \Rightarrow f(f(e)) = e \).

\(^4\) \( \beta_i \circ \beta_j \) is the composition of \( \beta_j \) and \( \beta_i \), i.e. \( \forall \delta \in D, \beta_i \circ \beta_j(\delta) = \beta_i(\beta_j(\delta)) \).
\(\beta\) maps except \(\beta_i\) we obtain all the darts that belong to the same \(i\)-cell as \(\delta\). More formally, cells correspond to orbits,\(^5\) as described in definition 2.

**Definition 2** (\(i\)-cell [4]). Let \(M = (D, \beta_1, \ldots, \beta_d)\) be a \(d\)-map, and \(\delta \in D\) be a dart. For any \(i, 0 \leq i \leq d\), the set of darts \(c_i(\delta)\) representing the \(i\)-cell containing \(\delta\) is:

\[
c_i(\delta) = \begin{cases} 
\{\beta_j \circ \beta_k \mid \forall j, k: 1 \leq j < k \leq d\}\langle \delta \rangle & \text{if } i = 0, \\
\{\beta_1, \ldots, \beta_{i-1}, \beta_{i+1}, \ldots, \beta_d\}\langle \delta \rangle & \text{otherwise.}
\end{cases}
\]

Note that there is a special case for vertices as each \(\beta_i\) changes not only the \(i\)-cell but also the vertex. Thus by combining two \(\beta\) maps we obtain a dart that belongs to the same vertex as the initial dart.

As its name suggests, a combinatorial map describes only the combinatorics of a given object, i.e. its subdivision into cells and all the incidence and adjacency relations between these cells. It is often necessary to also store additional information, which can be done using *attributes*. An \(i\)-attribute is an information associated with an \(i\)-cell. As cells are implicit, the link between an \(i\)-cell \(c\) and an \(i\)-attribute \(a\) is done through darts: all the darts in \(c\) are linked with \(a\). \(i\)-attributes are said *enabled* when they are some information associated with \(i\)-cells and they are said *disabled* otherwise.

It is common in applications to associate a point with each vertex, and no attribute with cells of other dimensions; this induces a linear geometry for the object. Such an embedded combinatorial map is called a *linear cell complex*. Another example of additional information is color or texture associated with faces of a 3D object.

### 3. Tools

The new standard C++11, approved in August 2011, provides many new functionalities [24]. We use two new features that are particularly interesting for an efficient and generic implementation.

The first one is the *variadic template*, which is a template having a variable number of arguments. In the following declaration: template<typename... Args> class Test; class template Test can have any number of template arguments. Test<int, bool, double> t1; and Test<vector<int>, char*, string, bool, double> t2; are two valid instantiations of the Test class template with different types and numbers of template arguments.

The second feature is the *std::tuple* object, which allows to hold different elements, possibly with different types. A tuple uses variadic templates so that it can contain any number of elements. For example, typedef tuple<int, bool, double> tuple1; and typedef tuple<vector<int>, char*, string, bool, double> tuple2; define two new types based on tuples. We can get the \(j^{th}\) type in a tuple using std::tuple_element<tuple_element<1, tuple1>::::type is bool and tuple_element<0, tuple2>::::type is vector<int>>. A tuple can be instantiated: e.g. tuple1 t1; tuple2 t2. Each element of the tuple can be accessed by using the get function: get<1>(t1) returns a reference to the bool in tuple t1 and get<0>(t2) returns a reference to the vector<int> in tuple t2.

Note that variadic template and tuples are instantiated at compile time and thus there is no overhead at execution time.

We also use a more classic tool to allocate and store elements of type \(T\): a container CGAL::Compact_container<T> [12]. This container has high memory efficiency (it is much more compact than a linked list) while allowing to insert an element in amortized constant time and to erase any element in constant time. An element in a Compact_container is accessed through a handle, which is a kind of pointer, defined as inner type of Compact_container.

### 4. Generic, compact, and efficient implementation of combinatorial maps

Implementing a combinatorial map mostly consists in encoding the darts, the \(\beta\) maps, and the associations between darts and enabled attributes. We aim at an implementation that is: (1) *Generic*: Users must be able to customize the

---

\(^5\) Let \(\Phi = \{f_1, \ldots, f_k\}\) be a finite set of permutations on some domain and \(\langle \Phi \rangle\) be the group generated by \(\Phi\). The *orbit* of an element \(\delta\) with respect to \(\Phi\) is \(\langle \Phi \rangle(\delta) = \{\phi(\delta) \mid \phi \in \langle \Phi \rangle\}\).
types of darts and the enabled attributes; (2) Compact: Represents associations only for enabled attributes; (3) Efficient: Allows direct access to every piece of information ($\beta$ maps and associations to attributes) associated with darts.

To reach these objectives, the key points of our implementation are: (1) An items class where users can choose the dimension of the combinatorial map, the type of darts, and the type of enabled attributes; (2) A tuple of handles through the enabled attributes; disabled attributes have no associated handle; (3) Direct access provided through handles for $\beta$ maps and tuple of handles for associations to enabled attributes.

4.1. Main classes

As mentioned above, the items class defines all types used in the combinatorial map. We present in listing 1 an example that uses 3D darts with attributes associated with faces and containing an int. Other attributes are disabled.

The items class must define an inner class called Dart_wrapper having a combinatorial map as a template parameter. This specific construction is required as the Dart class has a combinatorial map as template parameter (see listing 3 below), so that we can use handles through darts (these handles are defined in the Combinatorial_map class, see listing 2). In the class template Dart_wrapper, users can define their own type of darts using the Dart type, and their own type of attributes using the Attributes type. This last type is an instance of the tuple template having $d+1$ types. The $i^{th}$ element of the tuple gives the type of the $(i-1)$-attributes associated with $(i-1)$-cells, and must be either an instantiation of the class Cell_attribute or void; if it is void, $(i-1)$-attributes are disabled, i.e., there is no information associated with $(i-1)$-cells.

```
class Combinatorial_map_items_example
{
  template <class CMap>
  class Dart_wrapper
  {
    typedef Dart<3, CMap> Dart;
    typedef Cell_attribute<CMap, int> Face_attribute;
    typedef std::tuple<void, void, Face_attribute> Attributes;
  };
};
```

Listing 1. Example of items class for a 3-map

The main component of our implementation is the Combinatorial_map class template given in listing 2. This class has two template arguments: first the dimension $d$ of the map, and second an Items class that defines the types used in the combinatorial map.

```
template<unsigned int d, typename Items>
class Combinatorial_map
{
  typedef Combinatorial_map<d, Items> Self;
  typedef Items::Dart_wrapper<Self>::Dart Dart;
  typedef Compact_container<Dart> Dart_container;
  typedef Dart_container::iterator Dart_handle;
  typedef Transform_to_handles<Items::Attributes>::type Attribute_handles;

  Dart_container darts;
  Transform_to_containers<Items::Attributes>::types attributes;
};
```

Listing 2. $d$-map class

This class has two data members. (1) darts is the container of darts, implemented as Compact_container<Dart>, where Dart is the type of darts defined in the Items class. (2) attributes is the tuple of containers for all the enabled attributes. This tuple is defined by the Transform_to_containers tool class (given in appendix), which transforms the
tuple of attributes \texttt{Items::Attributes} into a tuple of compact containers, only for non-void types. For the attributes defined in listing 1, \texttt{Transform_to_containers} defines \texttt{std::tuple<Compact_container<Face_attribute> >}.

We use a similar technique to transform the \texttt{Attributes} tuple into \texttt{Attribute_handles}, a tuple of handle through all the non-void attributes, by using the \texttt{Transform_to_handles} tool class. For the attributes defined in listing 1, \texttt{Transform_to_handles} defines \texttt{std::tuple<Face_attribute_handle>}

Note that the number of handles in the \texttt{attributes} tuple is exactly the number of non-void attributes allowing for compact memory footprint. Moreover all the transformations are done without overhead for execution time since they are all done at compiling time. This guaranty also a direct access to each handle. This illustrates one important advantage of using variadic templates and tuples.

The dart class, given in listing 3, is essentially composed of an array of \(d + 1\) dart handles that encode the \(\beta\) maps (from \(\beta_0\) to \(\beta_d\)), and a tuple of attribute handles, one for each non-void attribute. The two types \texttt{Dart_handle} and \texttt{Attribute_handles} are defined as inner types in the \texttt{CMap} class.

```cpp
template<unsigned int d, typename CMap>
class Dart {
  CMap::Dart_handle betas[d+1];
  CMap::Attribute_handles attribute_handles;
};
```

Listing 3. Dart class

4.2. Linear cell complexes

A linear cell complex is a linear geometric embedding of a combinatorial map. The class \texttt{Linear_cell_complex<\delta,d2,Traits,Items>} inherits from the class \texttt{Combinatorial_map<\delta,Items>} and adds the constraint that each vertex of the combinatorial map must be associated with a point. \(d\) is the dimension of the combinatorial map and \(d_2\) is the dimension of the geometric ambient space (generally \(d_2 \geq d\)). For example, \(d = d_2 = 2\) for a planar graph embedded in a plane, \(d = 2\) and \(d_2 = 3\) for a surface embedded in \(\mathbb{R}^3\).

The \texttt{Traits} template parameter is the geometric traits class that defines the types for geometric objects such as \texttt{Point} and \texttt{Vector}, and the functors for geometric operations (for example \texttt{Construct_translated_point}, \texttt{Construct_sum_of_vectors}, or \texttt{Construct_midpoint}). The type \texttt{Point} is a \texttt{Cgal Point} type depending on the dimension of the ambient space. \texttt{Cgal} provides us with different so-called \texttt{kernels}, allowing us to choose between exact or inexact construction methods. The relation between vertices and points is encoded through attributes. The class \texttt{Cell_attribute_with_point} contains a \texttt{Point}, an object representing a point in the ambient space. This attribute may also optionally contain additional information associated with vertices, for example a color or a normal.

4.3. Iterators

The main basic features used in operations on combinatorial maps are iterators: they are used each time we need to process cells, as cells are orbits defined by \(\beta\) maps, and orbits are retrieved by iterating through all their corresponding darts. Since combinatorial maps are defined for any dimension, orbits can be used with any number of permutations. The C++11 variadic template mechanism allows us to define a generic iterator taking an arbitrary number of integers as template arguments. These integers give the indices of the \(\beta\) maps that define the orbit.

Iterators are defined as inner classes of \texttt{Combinatorial_map}, and are grouped into \texttt{ranges}, a range being simply a pair of iterators \texttt{begin} and \texttt{end}. For example, we have the following ranges:

\texttt{Dart_of_orbit_range<\text{\dots} Beta>\text{\dots}} ranges through all the darts belonging to \((\text{Beta}\ldots)(\delta)\) for a given dart \(\delta\). For instance, \texttt{Dart_of_orbit_range<\text{\texttt{1,2}}>(\delta)} ranges through all the darts of orbit \((\beta_1, \beta_2)(\delta)\), or \texttt{Dart_of_orbit_range<\text{\texttt{1,2,4,5}}>(\delta)} ranges through all the darts of orbit \((\beta_1, \beta_2, \beta_4, \beta_5)(\delta)\);

\texttt{Dart_of_cell_range<\text{\texttt{i}}>(\delta)} ranges through all the darts of the \texttt{i-cell} containing a given dart \(\delta\). For example \texttt{Dart_of_cell_range<\texttt{2}>(\delta)} ranges through all the darts of the face containing dart \(\delta\);

\texttt{One_dart_per_cell_range<\text{\texttt{i}}>(\delta)} ranges through one dart of each \texttt{i-cell} of the combinatorial map. For example \texttt{One_dart_per_cell_range<\texttt{0}>(\delta)} ranges through one dart of each vertex of the combinatorial map.
All the ranges are defined in a generic way allowing their use for any dimension of the combinatorial map. Moreover, *template specialization* allows us to propose optimized versions that are automatically used instead of the generic version in specific cases. For example, the generic version of `Dart_of_orbit_range` uses a stack storing the darts linked with the current darts by the considered β maps. These darts will be visited later (in a similar way as traversal algorithms for graphs). However versions with only one β map and some versions with two β maps can be implemented without this stack by using an order of the darts in the considered orbit, where each dart can be obtained from the previous darts in the order.

### 4.4. Operations

There are three types of operations defined on combinatorial maps and linear cell complexes.

*Computation operations* allow to compute some properties of a given combinatorial map. We can for example compute the number of cells using `count_cells`, which fills an `std::vector` with the number of cells of the combinatorial maps (for all the i-cells between 0 and d). We can also compute the normal vector of a face, given one of its darts, using `compute_normal_of_cell_2`.

*Construction operations* allow to create objects in a combinatorial map. We can create an object from scratch, for example `make_combinatorial_hexahedron` creates an isolated combinatorial hexahedron. It is also possible to convert an existing object from another format, for example `import_from_triangulation_3` converts a given CGAL 3D triangulation into the combinatorial map.

*Modification operations* allow to modify the combinatorics of a given map. We can create isolated darts, identify some cells to glue objects using `new` operations, or reciprocally split some cells to detach two glued objects using `unnew` operations. We can also modify the structure of an object by merging two (i + 1)-cells incident to an i-cell containing a given dart using the `remove<i>` operation, or similarly by merging two (i − 1)-cells incident to an i-cell containing a given dart using the `contract<i>` operation. Reciprocally, we can add an i-cell inside a j-cell using the `insertion` operations. These operations are a generalization in any dimension of the Euler operators [20] defined for polygonal meshes.

### 5. Comparison with other software

To the best of our knowledge, our two CGAL packages are the only available software that can describe dD irregular subdivided objects for any d. So, we have made two sets of experiments: a first one on 2D objects (i.e. surfaces), and a second one on 3D objects (i.e. volumes). Note that among all other packages tested, CGoGN is the only one that is able to describe both 2D and 3D objects. The goal of these experiments is not to show that our generic implementation is the best one but only to show that we are competitive, i.e. not so far from specialized implementations, while being fully generic (which is our main goal).

In 2D, we compare with OpenMesh [3], SurfaceMesh [23] and CGAL Polyhedron [7] [13], which are all based on halfedge data structures, and with CGoGN [16], which implements combinatorial maps.

We refer to [16,23] for a more precise description of the protocol that we follow. In a few words: (1) `circulator` iterates on all vertices incident to all faces; (2) `barycenter` computes the barycenter and recenters the mesh at the origin; (3) `normal` computes (and stores) all normals to faces; (4) `smoothing` performs a Laplacian smoothing; (5) `subdivision` does one step of √3-subdivision; (6) `collapse` splits all faces and then collapses each new edge.

We made these tests on five classical meshes (number of vertices, number of edges, number of faces): armadillo (26k, 78k, 52k), bunny (26k, 78k, 52k), horse (20k, 59k, 39k), octopus (16k, 49k, 33k) and vasselion (38k, 116k, 77k).

The results are shown in Fig. 3 (times are given relatively to the times obtained by our method to make interpretation easier). SurfaceMesh and OpenMesh are the best for access operations because they store elements in vectors, which yield very efficient iterators; the drawback of such representations is slower modification operations, as can be seen

---

6 The code of these benchmarks and all the detailed results are available at this url http://liris.cnrs.fr/gdamiand/download/linear_cell_complex_benchmarks.tgz.

7 We tested the version using a list, since the version using a vector does not allow to remove elements, thus many operations of modification are not supported.
for subdivision and collapse. Our solution is always better than Polyhedron, which uses doubled linked lists as containers. We obtain very similar results as CGoGN, which is also based on combinatorial maps; CGoGN is better for normals and smoothing, due to the use of a cache during the first traversal of iterators, to optimize the future uses. Such a technique could also be integrated in our solution to speed up circulators.

In 3D, we compare with OpenVolumeMesh [17], which is based on an extension of the halfedge data structure, and with CGoGN [16].

We again follow the same protocol as in [16,17]: (1) circulator iterates along all vertices incident to all volumes; (2) circulator2 iterates along all vertices adjacent along a common volume; (3) barycenter computes and stores the barycenters of all volumes; (4) smoothing performs a Laplacian smoothing; (5) subdivision does one step of $(1 - 4)$-subdivision of each tetrahedron; (6) collapse does a series of edge collapses of the shortest edge.

We reuse the five previous surfacic meshes and use TetGen [22] to build corresponding tetrahedral meshes. The sizes of the five volumic meshes obtained are (number of vertices, number of tetrahedra): $(26k, 174k)$, $(26k, 176k)$, $(20k, 136k)$, $(16k, 127k)$ and $(38k, 262k)$.

The results are shown in Fig. 4. OpenVolumeMesh is faster than our implementation for circulator and smoothing, because it explicitly stores cells and incidence relations between cells; as a counterpart, this makes it definitely inefficient for modification operations. CGoGN is also better for circulator and smoothing, due to the computation and the storage of these incidence relations, which are integrated inside the software itself. We use the same techniques for smoothing, which allow for fast circulators, but only in the benchmarking code. In high level operations, time is
generally not spent by circulators, but by the operation itself; this is confirmed in subdivision and collapse, where the results obtained by Linear Cell Complexes are the best or close to the best.

Note that we obtain better results here for Linear Cell Complexes than the results given in the two previous papers [16,17], where operations for testing linear cell complex where not implemented in the best way. We improved the benchmarking code, which lead to a speed up factor between 2 and 20! Such optimized algorithms are actually integrated in CGoGN, while low level functions must currently be used for Linear Cell Complexes, which could be enriched to offer the same optimized functions in the future.

To summarize, we observe that software using vectors to store their elements (OpenMesh, SurfaceMesh, and OpenVolumeMesh) are generally better for iterations, and thus better for static operations, while they obtain slower performances for modification operations.

The most important observation for us is that our generic software obtains results that are competitive when compared with other software, which are all restricted to to 2D and 3D, and it shows performance that are among the best ones for modification operations. This illustrates that generic programming and features introduced in C++11 lead to both genericity and efficiency.

6. Examples of applications

Combinatorial maps and linear cell complexes can be used in various applications manipulating objects that are subdivided into irregular cells.

6.1. Building reconstruction

Computer models of 3D buildings are extensively used, e.g., by architects or city planners, to create virtually new buildings and to visualize them in virtual 3D environments. When the models are used only for visualization purposes, there is no need for advanced data structures; a soup of polygons can be used to represent the buildings. However, such a description does not allow to manipulate the different parts of the scene or to iterate through the different objects by using adjacency relations, therefore it cannot be used for high level treatments such as physical simulations.

To answer this need, the TopoBuilding project [7] is aiming at reconstructing a 3D building from a given geometry into a valid topological description as a 3D linear cell complex. The input is a geometric description of a building, given in a Collada format [1], which is an exchange file format for 3D applications. The resulting 3D linear cell complex describes the topology of the input building. Volumes correspond to the different elements of the building.
(such as rooms, walls and doors) with adjacency relations between these volumes (allowing for example to know if two rooms are connected by a door), and with incidence relations between the different parts of these volumes (for example to know how the different walls of a room are linked). Fig. 5 depicts an example of reconstruction. The description of buildings as 3D linear cell complexes can be used in order to perform energy and acoustic simulations.

6.2. Physical simulation

In this application, the goal is to use 3D linear cell complexes as basic topological framework to implement a physical simulation method based on mass spring systems. Classical implementations often use graphs where vertices are physical particles, and edges are springs. Using a more complete description allows to define more efficient operations that change the subdivision during the simulation (for example cutting the object or locally refining some cells) while guaranteeing the topological validity of the objects.

For this application, the project TopoSim [9] implements a generic topological framework for physical simulation based on 3D linear cell complexes. We can see in Fig. 6 an example of some preliminary results. It is possible to apply a physical simulation on 3D hexahedral or tetrahedral meshes, while allowing the cutting of some volumes during the simulation. Future work could allow to refine some cells during the simulation, and could provide more general types of cuttings (for example cutting by a plane). In both cases, the generic type of cells should simplify the operations and 3D linear cell complexes will provide all basic tools to quickly develop new high level operations.

7. Conclusion

We have presented a generic implementation of dD combinatorial maps and linear cell complexes, which is compact, i.e., it does not use useless data members for disabled attributes, and efficient, i.e., it gives direct access to information associated with each dart. Genericity, compactness and efficiency are achieved by using generic programming techniques and new possibilities introduced in the recent C++11 standard, such as variadic templates and tuples. We have compared our solution with existing 2D and 3D software. These tests show that our solution is competitive compared to dedicated solutions. We have illustrated the practical interest of such data structures by showing two applications under development.

Future work may improve the two CGAL modules by adding new operations and by proposing alternative implementations to describe β maps, for example using indices instead of handles. We will also implement some optimization of our iterators, using ideas that are similar to those used in CGoGN.
Work will be pursued on the different applied projects that use these combinatorial maps. Lastly, it will be interesting to explore applications in higher dimensions; for example, it is possible to use 4D linear cell complexes to describe temporal sequences of 3D MRI images and to use the cells and incidence relations to propose 4D image processing (the fourth dimension being the time).

Acknowledgments

Thanks to Abdoulaye Abou Diakité and Dirk Van M Aercke for their collaboration in TopoBuilding; to Elsa Fréchon, Fabrice Jallet and Florence Zara for their collaboration in TopoSim. Thanks to Pierre Kraemer, Michael Kremer, Lionel Unterreiner for helpful discussions about their experiments. Thanks to Andreas Fabri, Sébastien Lorioit, and Laurent Rineau for their help during the implementation of the two CGAL packages.

This work has been partially supported by the French National Agency (ANR), projects DIGITALSNOW ANR-11-BS02-009 and SOLSTICE ANR-13-BS02-01.

References

Appendix A. How to transform a tuple of types into a tuple of Compact_container

The code given in this appendix can be found in the public release of Cgal, in file internal/Combinatorial_map_utility.h.

The goal is to transform a tuple of types into a tuple of Compact_container’s on the corresponding types, but only for non-void attributes (see Section 4.1).

We first transform the tuple of types into a tuple where we have removed all the void types using class Keep_non_void_type given in listing 4.

```cpp
template <class Attrs, class Res=tuple<> >
struct Keep_non_void_type;

template <class T, class ...Attrs, class ...Res>
struct Keep_non_void_type<
tuple<T, Attrs...>, tuple<Res...> >
{
    typedef Keep_non_void_type<
tuple<Attrs...>, tuple<Res..., T> >::type type;
};

template <class ...Attrs, class ...Res>
struct Keep_non_void_type<
tuple<void, Attrs...>, tuple<Res...> >
{
    typedef Keep_non_void_type<
tuple<Attrs...>, tuple<Res...> >::type type;
};

template <class ...Res>
struct Keep_non_void_type<
tuple<> , tuple<Res...> >
{
    typedef tuple<Res...> type;
};
```

Listing 4. Remove void types from a tuple

This code uses the variadic templates mechanism (as tuples are also defined thanks to this mechanism).

The principle of this class is to make a recursion at compile time on the tuple of attributes (called Attrs). The general case is the first specialization

```cpp
    Keep_non_void_type<
tuple<T, Attrs...>, tuple<Res...> >
```

where the first type in the tuple Attrs is different from void. Then we have the special case

```cpp
    Keep_non_void_type<
tuple<void, Attrs...>, tuple<Res...> >
```

when the first type is void and the last case

```cpp
    Keep_non_void_type<
tuple<> , tuple<Res...> >
```

stops the recursion when the tuple is empty.

Note that the class Keep_non_void_type has two template parameters, which are two tuples. This explains why even by using two variadic templates arguments, the compiler can retrieve which arguments correspond to the first tuple and which arguments correspond to the second one.

In order to construct the resulting tuple, we use an additional template argument, Res. When the first type of the tuple Attrs is a non-void T, we simply add T at the end of Res; when T is void, then Res is not modified. When the

---

8 Cf. classes Keep_type_different_of and Tuple_converter which are generalizations of Keep_non_void_type, Transform_to_containers and Transform_to_handles.
**Attrs** tuple is void, the **Res** tuple contains the result of the transformation: A tuple of types equal to **Attrs**, from which we have removed all the void types.

Now we present in listing 5 the code allowing to transform a tuple of types info a tuple of **Compact_container** of these types.

```cpp
template <class ...T>
struct Transform_to_containers<tuple<T...> >
{
    typedef tuple<Compact_container<T>... > type;
};
```

Listing 5. Transform a tuple of types into a tuple of **Compact_container** of these types

Here the transformation is direct: C++11 allows to transform a tuple `tuple<T...>` into another tuple by extension: `tuple<Compact_container<T>...>`. As an example, if we are given a tuple

```cpp
typedef tuple<int, void, char, void, Dart> Attrs,
```

we first get

```cpp
typedef Keep_non_void_type<Attrs>::type Attrs_novoid,
```

which is equal to `tuple<int, char, Dart>`. Finally,

```cpp
Transform_to_containers<Attrs_novoid>::type
```

is `tuple<Compact_container<int>, Compact_container<char>, Compact_container<Dart>>`. Remind that all transformations are done at compile time, there is no overhead for running time.

Lastly, we give in listing 6 the code allowing to transform a tuple of types into a tuple of handle to these types (these handles are defined as inner type of the corresponding **Compact_container**).

```cpp
template <class ...T>
struct Transform_to_handles<tuple<T...> >
{
    typedef tuple<Compact_container<T>::iterator... > type;
};
```

Listing 6. Transform a tuple of types into a tuple of handle to these types