Author
Guillaume Damiand

1 Introduction
2 Data Structure Presentation
2.1 Generalized Map and Darts
2.2 Cells as Sets of Darts
2.3 How to Associate Information to Cells
2.4 Generalized Map Properties
2.5 Comparison with Combinatorial Maps
3 Software Design
3.1 Generalized Maps
3.2 Generalized Map Items
3.3 Cell Attributes
3.4 Example of Generalized Map Definition
4 Iteration and Creation Operations
4.1 Iterating over Orbits, Cells, and Attributes
4.2 Construction Operations
4.3 Boolean Marks
5 Modification Operations
5.1 Sewing Orbits and Linking Darts
5.2 Removal and Insertion Operations
6 Examples
6.1 A 3D Generalized Map
6.2 A non orientable 2D Generalized Map
6.3 High Level Operations
6.4 A 4D Generalized Map
6.5 Generalized Map With Attributes
6.6 Use of Dynamic Onmerge and Onsplit Functors
7 Mathematical Definitions
8 Design and Implementation History

1 Introduction

A *d*-dimensional generalized map is a data structure representing an orientable or non-orientable subdivided *d*-dimensional object obtained by taking *d*D cells, and allowing to glue *d*D cells along *(d-1)*D cells. It provides a description of all the cells of the subdivision (for example vertices and edges), together with incidence and adjacency relationships.

This package is a generalization of the combinatorial maps data structure (which allows to describe only orientable objects) in order to be able to describe also non-orientable objects such as a Möbius strip (Figure 29.1 Left) or a Klein bottle (Figure 29.1 Right).

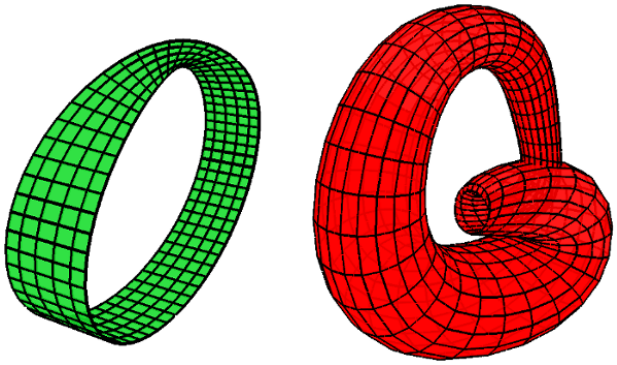


Figure 29.1 Example of two non-orientable objects. Left: A Möbius strip. Right: A Klein bottle.

We denote *i*-cell for an *i*-dimensional cell (for example in 3D, 0-cells are *vertices*, 1-cells are *edges*, 2-cells are *facets*, and 3-cells are *volumes*). A *boundary relation* is defined on these cells, giving for each *i*-cell *c* the set of *(i-1)*-cells contained in the boundary of *c*. Two cells *c1* and *c2* are *incident* if there is a path of cells, starting from the cell of highest dimension to the other cell, such that each cell of the path (except the first one) belongs to the boundary of the previous cell in the path. Two *i*-cells *c3* and *c4* are *adjacent* if there is an *(i-1)*-cell incident to both *c3* and *c4*. You can see an example of a 2D object and a 3D object in Figure 29.2 showing some cells of the subdivision and some adjacency and incidence relations.

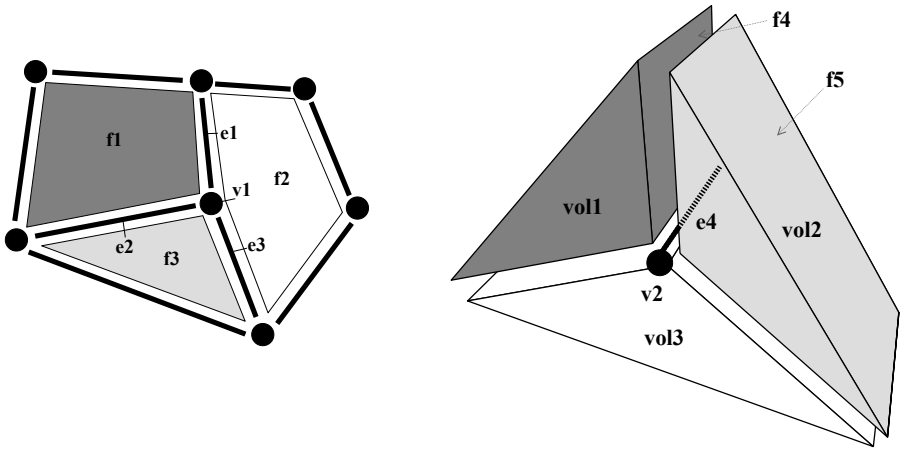


Figure 29.2 Example of subdivided objects that can be described by generalized maps. Left: A 2D object composed of three facets (2-cells), named *f1*, *f2* and *f3*, nine edges (1-cells) and seven vertices (0-cells). *f1* and *f2* are adjacent along edge *e1*, thus *e1* is incident both to *f1* and *f2*. Vertex *v1* is incident to edge *e1*, thus *v1* is incident to *f1* and *f2* by transitivity. Right: A 3D object (only partially represented for vertices and edges) composed of three volumes (3-cells), named *vol1*, *vol2* and *vol3*, twelve facets (2-cells) (there is one facet *f4* between *vol1* and *vol2*, and similarly between *vol1* and *vol3* and *vol2* and *vol3*), sixteen edges (1-cells), and eight vertices (0-cells). *vol1* and *vol2* are adjacent along facet *f4*, thus *f4* is incident both to *vol1* and *vol2*. Edge *e4* is incident to the three facets between *vol1* and *vol2*, *vol1* and *vol3*, and *vol2* and *vol3*. *e4* is also incident to the three volumes by transitivity.

A generalized map is an edge-centered data structure, describing the cells and the incidence and adjacency relations. It uses only one basic element called *dart*, and a set of *pointers* between these darts. A dart can be thought as a part of an edge (1-cell), together with a part of incident cells of dimensions 0, 2, 3, ..., *d*. When a dart *d0* describes a part of an *i*-cell *c*, we say that *d0 belongs* to *c*, and that *c contains d0*. Let us look at the example in Figure 29.3 showing the 2D and 3D generalized maps describing the two objects of Figure 29.2.

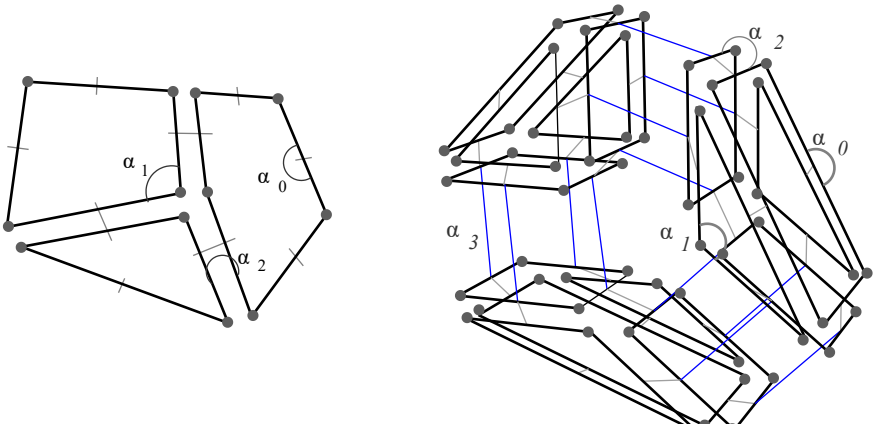


Figure 29.3 Generalized maps representing the objects given in Figure 29.2. Left: The 2D generalized map which contains 24 darts. Right: The 3D generalized map which contains 108 darts (36 for each volume).

First let us start in 2D (Figure 29.3 (Left)). Edge *e1* contains four darts. These darts are linked together with pointers called α_0 and α_2. Starting from a dart and following an α_0 pointer, we get to a dart which belongs to the same edge, to the same facet but to the other vertex (0-cell, which explains the index 0 of α_0). Starting from a dart and following an α_2 pointer, we get to a dart which belongs to the same vertex, to the same edge but to the other facet (2-cell, which explains the index 2).

Facet *f1* is represented by four edges, and thus contains eight darts. The edges are linked together with pointers called α_0 and α_1. Starting from a dart and following an α_1 pointer, we get to a dart which belongs to the same vertex, the same facet but to the other edge (1-cell, which explains the index 1 of α_1).

Similarly, vertex *v1* contains six darts, linked together with pointers α_1 and α_2.

The main interest of generalized map definition based on darts and α_i pointers is to be able to increase the dimension *only* by adding new pointers. This is illustrated thanks to the 3D example given in Figure 29.3 (Right). In addition to α_0, α_1 and α_2 of the 2D case, there is a new pointer α_3.

If we take a closer look at the central edge *e4* shown in Figure 29.4 (Left), we can see that it contains twelve darts linked together. Starting from a dart and following an α_3 pointer, we get to a dart which belongs to the same vertex, to the same edge, to the same facet, but to the neighboring volume (a 3-cell, which explains the index 3 in α_3). Similarly, starting from a dart and following an α_2 pointer, we get to a dart which belongs to the same vertex, to the same edge, to the same volume, but to the neighboring facet (2-cell). And starting from a dart and following an α_0 pointer, we get to a dart which belongs to the same edge, to the same facet, to the same volume, but to the neighboring vertex (0-cell). Starting from any of these twelve darts and following α_0, α_2 and α_3 pointers, we can reach exactly the twelve darts that belong to edge *e4*.

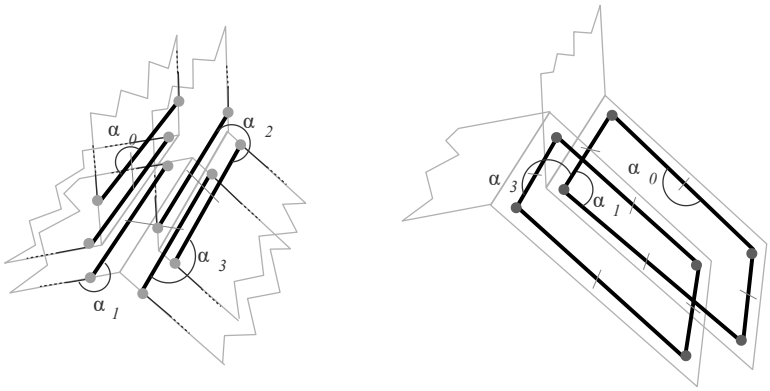


Figure 29.4 Two zooms on the 3D generalized map given in Figure 29.3 (Right). Left: Zoom around the central edge *e4* which details the twelve darts belonging to the edge. Right: Zoom around the facet between volumes *vol2* and *vol3* which details the sixteen darts belonging to the facet.

For facets, by following an α_1 pointer, we get to a dart which belongs to the same vertex, to the same facet, to the same volume, but to the next edge (1-cell, which explains the index 1 of α_1). Starting from any dart and following α_0, α_1 and α_3 pointers, we can reach exactly all the darts belonging to the facet (see Figure 29.4 (Right)). For volumes, starting from any dart and following α_0, α_1 and α_2 pointers, we can reach exactly all the darts belonging to the volume. For vertices, we have to follow α_1, α_2 and α_3 pointers to reach exactly the darts belonging to the vertex *v*.

In some cases, the general rule that by following an α_i we get a dart which belongs to the neighboring *i*-cell is not true, as for example for darts belonging to the boundary of the represented object. For example, in Figure 29.2 (Left), any dart *d0* that does not belong to edge *e1*, *e2* and *e3* belongs to a 2-cell, and there is no neighboring facet along the edge containing *d0*. Another example is in Figure 29.2 (Right), for any dart *d0* that belongs to facet *f5*. *d0* belongs to volume *vol2*, but there is no neighboring volume along this facet. The general rule is also not true for unbounded cells. For example if we remove a dart in Figure 29.3 (Left), we obtain an unbounded facet having one dart without next dart for α_0, and one dart without next dart for α_1, and if we remove a facet in Figure 29.3 (Right), we obtain an unbounded volume having some darts without neighboring facet for α_2. In such a case, the darts are linked with themselves for α_i to describe that a dart *d0* is not linked to another dart in dimension *i*.

Generalized maps are defined in any dimension. A -1D generalized map is a set of isolated darts describing isolated vertices. A 0D generalized map is a set of darts paired by α_0 describing isolated edges. A 1D generalized map describes paths or cycles of darts corresponding to paths or cycles of edges. The most useful cases are 2D and 3D generalized maps. In 2D, a generalized map is a set of surfaces (orientable or not), and in 3D a generalized map is a set of connected volumes. In the following, notions are mainly illustrated in 3D. But it is important to keep in mind that one main interest of generalized maps is their generic definition in any dimension, and that everything presented in this manual is valid in any dimension.

A *d*D generalized map is useful when you want to describe *d*D objects and the adjacency relations between these objects, and you want to be able to efficiency traverse these objects by using the different relations. For example, we can use a 3D generalized map to describe a 3D segmented image: each 3-cell corresponds to a region in the image and each 2-cell corresponds to a contact area between two regions.

A generalized map does not contain any geometric information. However, this package allows to associate any information to the cells of the generalized map. A specific information, which is often used in practice, consists in adding linear geometry to a generalized map by associating a point to each vertex of the map: this is the object of the Linear cell complex package (when an object has a point associated to each vertex, each edge is thus a straight line segment, which explains the name *linear geometry*). The Linear cell complex package can for example be useful to describe 3D buildings as set of walls, rooms, doors and windows (both combinatorial and geometric descriptions) and all the adjacency relations between these elements allowing for example to move a camera in a given building from rooms to rooms by traversing doors.

2 Data Structure Presentation

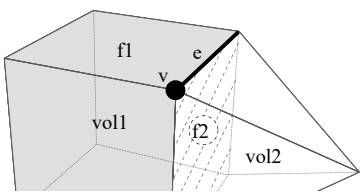
In this section, we describe *d*D generalized maps in terms of data structure and operations. Mathematical definitions are provided in Section Mathematical Definitions, and a package description is given in Section Software Design.

2.1 Generalized Map and Darts

A *d*D generalized map is a set of darts *D*. A dart *d0* is an element that can be *linked* with *d*+1 darts by pointers called α_i, with $0 \leq i \leq d$. Dart *d0* is said *i-free* when α_i(*d0*)=*d0*. Each α_i is its own inverse, i.e. $\alpha_i(\alpha_i(d0))=d0$.

A generalized map is *without i-boundary* if there is no *i*-free dart, and it is *without boundary* if it is without *i*-boundary for all dimensions $1 \leq i \leq d$.

We show in Figure 29.5 a 3D object and the corresponding 3D generalized map. This map has 80 darts, some darts being numbered. In this generalized map, we have for example $\alpha_0(1)=2$, $\alpha_1(1)=8$, $\alpha_2(1)=24$, and $\alpha_3(1)=9$. This generalized map is without 0-boundary, without 1-boundary and 2-boundary, but has some 3-boundary, because some darts are 3-free, for example $\alpha_3(17)=17$.



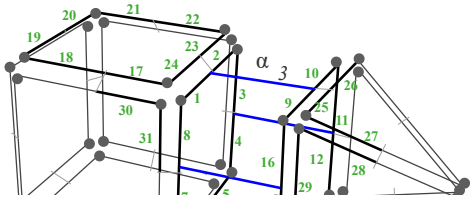


Figure 29.5 Example of a 3D generalized map. Left: A 3D object made of two volumes adjacent along facet *f2*. Right: The corresponding 3D generalized map. Darts are drawn with dark grey and black segments, black darts being numbered. Two darts linked by α_0 are drawn aligned and separated by a small gray orthogonal segment (for example $\alpha_0(1)=2$), two darts linked by α_1 are drawn consecutively and separated by a small gray disk (for example $\alpha_1(1)=8$), and two darts linked by α_2 are drawn parallel, in reverse orientations, with the little gray segment joining them (for example $\alpha_2(1)=24$). α_3 pointers are represented by blue segments in this figure (for example $\alpha_3(1)=9$).

2.2 Cells as Sets of Darts

A cell in a *d*D generalized map is implicitly represented by a subset of darts. In this section, we will see how to retrieve all cells containing a given dart, how to retrieve all darts belonging to a cell containing a given dart, and how incidence and adjacency relations are defined in terms of darts.

The first important property of a generalized map is that each dart belongs to an *i*-cell, $\forall\, i,\ 0 \leq i \leq d$. For example in 3D, a dart belongs to a vertex, an edge, a facet, and a volume. This means that a 3D generalized map containing an isolated dart contains exactly one vertex, one edge, one facet and one volume.

The second important property is that cells of a generalized map correspond to specific *orbits*. Given a set $S \subseteq \{\alpha_1,..., \alpha_d\}$ and a dart *d0*, the *orbit* $\langle\ S\ \rangle(d0)$ is the set of darts that can be reached from *d0* by following any combination of any α_i's in *S* (to simplify notations, we can use for example $\langle\ \alpha_1, \alpha_4\ \rangle(d0)$ to denote $\langle\ S\ \rangle(d0)$ with $S=\{\alpha_1, \alpha_4\}$).

Given a dart *d0*, in general, $\alpha_i(d0)$ (with $0 \leq i \leq d$) belongs to the same cells as *d0*, only the *i*-cell is different. There are two exceptions:

1. if *d0* is *i*-free, then $\alpha_i(d0)=d0$, the *i*-cell is not different;
2. if $\alpha_i(d0)$ belongs to the same *i*-cell as *d0* (case of multi-incidence). For example if an edge is an isolated loop, it is incident twice to the same vertex, then given a dart *d0* belonging to this edge, $\alpha_1(d0)$ goes to the next edge, which is in fact the same edge.

Since $\alpha_i(d0)$ (with $0 \leq i \leq d$) allows to change the current *i*-cell, all the darts that can be reached from *d0* by using any combination of α_j's, $\forall\, j,\ 0 \leq j \leq d$ and $j \neq i$ are contained in the same *i*-cell as *d0*. The *i*-cell containing *d0* is defined in terms of orbit by $\langle\ \alpha_0,..., \alpha_{i-1}, \alpha_{i+1},..., \alpha_d\ \rangle(d0)$.

Orbit $\langle\ \alpha_0,..., \alpha_d\ \rangle(d0)$ is the *connected component* containing dart *d0*. A generalized map is *connected* if this set is equal to the set of all the darts of the generalized map.

A last important property of cells is that for all dimensions *i* the set of *i*-cells forms a partition of the set of darts *D*, i.e. for any *i*, the union of the sets of darts of all the *i*-cells is equal to *D*, and the sets of darts of two different *i*-cells are disjoint.

Let us give some examples of cells in 3D, for the 3D generalized map of Figure 29.5 :

- All the darts belonging to the same vertex can be obtained by any combination of α_1, α_2 and α_3: for example vertex *v* of the object corresponds in the generalized map to the set of darts {1,8,9,16,17,24,25,27,28,29,30,31}. Given any dart belonging to this vertex, we retrieve all the other darts by, for example, a breadth-first traversal. In terms of orbits, this 0-cell corresponds to $\langle\ \alpha_1, \alpha_2, \alpha_3\ \rangle(1)$.
- All the darts belonging to the same edge can be obtained by any combination of α_0, α_2 and α_3: for example edge *e* of the object corresponds in the generalized map to the set of darts {1,2,9,10,23,24,25,26}. In terms of orbits, this 1-cell corresponds to $\langle\ \alpha_0, \alpha_2, \alpha_3\ \rangle(1)$.
- All the darts belonging to the same facet can be obtained by any combination of α_0, α_1 and α_3: for example facet *f2* corresponds in the generalized map to the set of darts {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}. Facet *f1* corresponds to the set of darts {17,18,19,20,21,22,23,24}. Note that these last darts are 3-free since there is no other volume sharing this facet. In terms of orbits, *f2* corresponds to $\langle\ \alpha_0, \alpha_1, \alpha_3\ \rangle(1)$ and *f1* corresponds to $\langle\ \alpha_0, \alpha_1, \alpha_3\ \rangle(17)$.
- All the darts belonging to the same volume can be obtained by any combination of α_0, α_1 and α_2: for example volume *vol1* corresponds in the generalized map to the set of the 48 darts belonging to the cube. In terms of orbits, *vol1* corresponds to $\langle\ \alpha_0, \alpha_1, \alpha_2\ \rangle(1)$.

Using this definition of cells as sets of darts, we can retrieve all the incidence and adjacency relations between the cells of the subdivision in a generalized map. Two cells are *incident* if the intersection of their two sets of darts is non empty (whatever the dimension of the two cells). Two *i*-cells *c1* and *c2*, $1 \leq i \leq d$, are *adjacent* if there is *d1* $\in$ *c1* and *d2* $\in$ *c2* such that *d1* and *d2* belong to the same *(i-1)*-cell.

In the example of Figure 29.5, vertex *v* and edge *e* are incident since the intersection of the two corresponding sets of darts is $\{1,9,24,25\} \neq \emptyset$. Vertex *v* is incident to facet *f2* since the intersection of the two corresponding sets of darts is $\{1,8,9,16\} \neq \emptyset$. Edge *e* and facet *f1* are incident since the intersection of the two corresponding sets of darts is $\{23,24\} \neq \emptyset$. Finally, facets *f1* and *f2* are adjacent because 1 $\in$ *f1*, 24 $\in$ *f2* and 1 and 24 belong to the same edge.

We can consider *i*-cells in a dimension *d'* with $i \leq d' \leq d$. The idea is to consider the *i*-cells as if the generalized map was in *d'* dimension. For that, we only take into account the α_js for $j \leq d'$. The *i*-cell containing *d0* in dimension *d'* is the orbit $\langle\ \alpha_0,..., \alpha_{i-1}, \alpha_{i+1},..., \alpha_{d'}\ \rangle(d0)$. By default, *i*-cells are considered in dimension *d*, the dimension of the generalized map.

In the example of Figure 29.5, the 2-cell containing dart 1 is facet *f2* which is the set of darts {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}. If we consider the same 2-cell in dimension 2, we obtain the set of darts {1,2,3,4,5,6,7,8}. Intuitively we *forget* α_3 and we obtain the set of darts of the facet containing dart 1 restricted to the volume containing this dart.

2.3 How to Associate Information to Cells

Generalized maps only describe the cells of the subdvision, and all the incidence and adjacency relations between these cells. This is not enough for many applications which need to associate *information* to cells. This can be geometric or non-geometric information, such as 3D points associated to vertices, the edge length associated to edges, or a color or normal to a facet.

To answer this need, a generalized map allows to create *attributes* which are able to store any information, and to associate attributes to cells of the generalized map. We denote *i*-attributes for the attributes associated with *i*-cells. Attributes may exist for only some of the dimensions, and if they exist for dimension *i*, they do not necessarily exist for each of the *i*-cells. More precisely, *i*-attributes are associated to *i*-cells by an injection:

- two different *i*-cells are associated to two different *i*-attributes;
- an *i*-cell may have no associated *i*-attribute.

Since *i*-cells are not explicitely represented in generalized maps, the association between *i*-cells and *i*-attributes is transferred to darts: if attribute *a* is associated to *i*-cell *c*, all the darts belonging to *c* are associated to *a*.

We can see two examples of generalized maps having some attributes in Figure 29.6. In the first example (Left), a 2D generalized map has 1-attributes containing a float, for example corresponding to the length of the associated 1-cell, and 2-attributes containing a color in RGB format. In the second example (Right), a 3D generalized map has 2-attributes containing a color in RGB format.

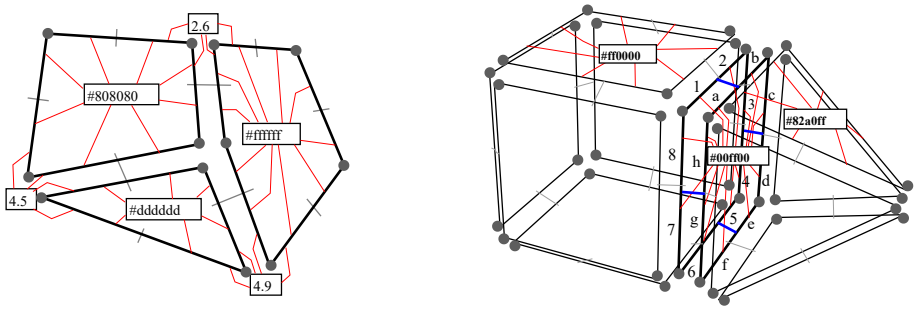


Figure 29.6 Example of generalized maps with attributes. Attributes are represented by black rectangles containing an information, and association between darts and attributes are represented by small red lines. Left: A 2D generalized map with 1-attributes containing a double, for example corresponding to the length of the 1-cell, and 2-attributes containing a color in RGB format. Only three edges of the generalized map, among the nine, are associated to a 1-attribute. All the 2-cells are associated to a 2-attribute. Right: A 3D generalized map with 2-attributes containing a color in RGB format. Only three 2-cells of the generalized map, among the ten, are associated to a 2-attribute.

2.4 Generalized Map Properties

There are some conditions that a generalized map must satisfy to be valid. Some of them have already been given about the α pointers (see Section Generalized Map and Darts) and about the association between darts and attributes (see Section How to Associate Information to Cells).

There is an additional condition related to the type of represented objects, which are *quasi-manifold* *d*D objects. A *d*D quasi-manifold is an object obtained by taking some isolated *d*-cells, and allowing to glue *d*-cells along *(d-1)*-cells. In 2D, quasi-manifolds are manifolds, but this is no longer true in higher dimension as we can see in the example presented in Figure 29.7. In this example, the object to the right is not a manifold since the neighborhood of the point *p* in the object is not homeomorphic to a 3D ball (intuitively, two objects are homeomorphic if each object can be continuously deformed into the second one; in such a case, the two objects have exactly the same topological properties).

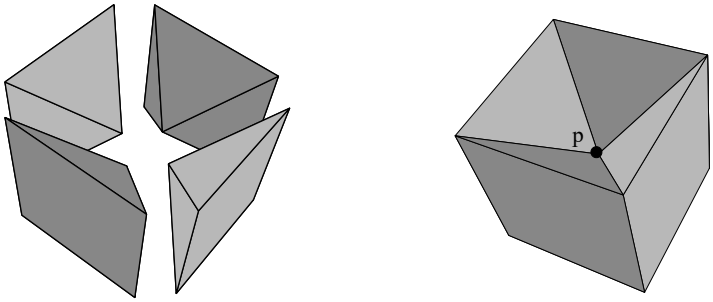


Figure 29.7 Example of a 3D quasi-manifold which is not a manifold. The object to the right is made of the four pyramids (shown to the left) glued together along facets, thus it is a quasi-manifold.

Generalized maps can only represent quasi-manifolds due to the definition of α pointers. As we have seen in Section Cells as Sets of Darts, α_i(*d0*) (with $0 \leq i \leq d$) belongs to the same cells as *d0*, only the *i*-cell is different. In other words, α_i links two *i*-cells that share a common *(i-1)*-cell: it is not possible to link more than two *i*-cells along a same *(i-1)*-cell. For this reason, it is not possible to describe non quasi-manifold objects as those shown in Figure 29.8 by generalized maps.

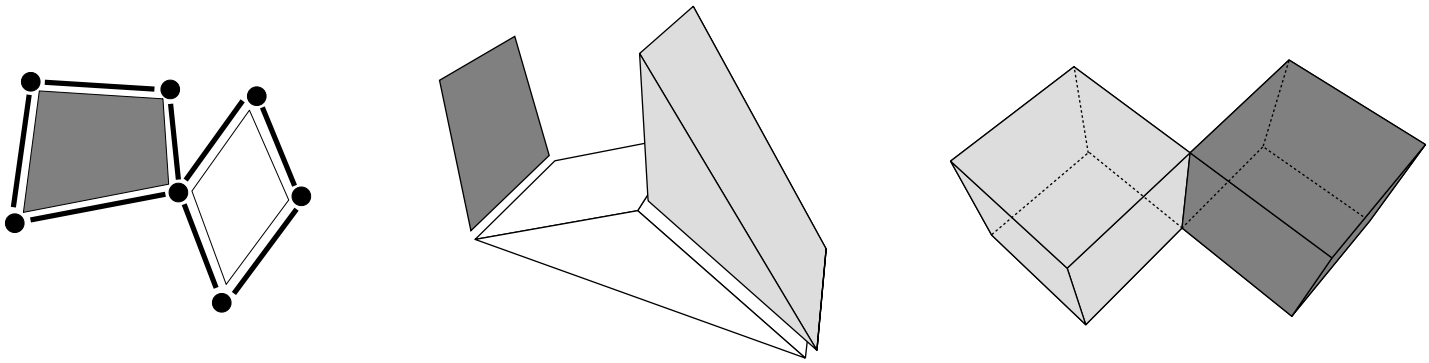


Figure 29.8 Three examples of non quasi-manifold objects. Left: A 2D object which is not a quasi-manifold since the two 2-cells share a common vertex but no common 1-cell. Middle: A 3D object which is not a quasi-manifold since is it not only composed by 3D cells glued together (there is an isolated 2-cell in dark gray). Right: A 3D object which is not a quasi-manifold since the two 3-cells share a common edge but no common 2-cell.

Due to this additional condition, any objects can not be represented by a generalized map but only quasi-manifolds. We need to study now the inverse relation. Does any set of darts linked together by α_i's, with $0 \leq i \leq d$ correspond to a quasi-manifold? As we can see in Figure 29.9, the answer is no.

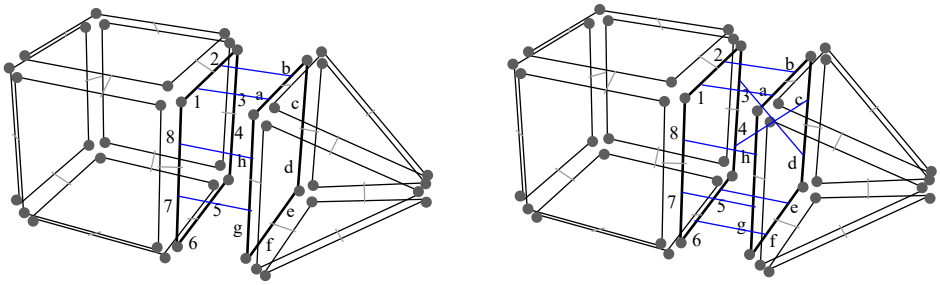


Figure 29.9 Two examples of darts linked together by some α_0, α_1, α_2 and α_3 which does not represent a 3D quasi-manifold, and thus which are not 3D generalized map. Left: In this example, all the darts are 3-free except α_3(1)=a, α_3(2)=b, α_3(7)=g and α_3(8)=h (and vice-versa). Right: In this example, darts 1 to 8 and a to h linked by α_3 are not in the same order in both 3-cells.

In the first example (Left), there are two 3-cells (one to the left for the cube, a second to the right for the pyramid) which are *partially adjacent* along one 2-cell. Indeed, only four darts of the 2-cell are linked by α_3. We have α_3(1)=a, α_3(2)=b, α_3(7)=g and α_3(8)=h (and vice-versa). This configuration is not possible in a quasi-manifold:

two *d*-cells are always glue along an *entire (d-1)*-cells.

But as we can see in the second example (Right), the condition that all the darts of the cell are linked in not sufficient. Indeed, in this example, all the darts of the 2-cell between the cube and the pyramid are linked together by α_3. However, this configuration does not correspond to a 3D quasi-manifold. Indeed, the operation of gluing two *d*-cells along one *(d-1)*-cell must preserve the structure of the initial *(d-1)*-cell.

To avoid these two kinds of configurations, conditions are added on α pointers compositions (see Section Mathematical Definitions, condition (3) of the definition of generalized maps). Intuitively these conditions say that if two darts are linked by α_i, then all the required darts are linked by α_i two by two in such a way that neighborhood relations are preserved.

We say that a generalized map is *valid* if it satisfies all the conditions on α pointers and on association between darts and attributes. High level operations provided on generalized maps ensure that these conditions are always satisfied. Sometimes, it can be useful to use low level operations in a specific algorithm, for example to modify locally a generalized map in a really fast way. In such a case, additional operations may be needed to restore these validity conditions.

2.5 Comparison with Combinatorial Maps

Generalized maps and combinatorial maps are very similar: they are both based on darts and functions, and they both allow to represent quasi-manifold *n*D objects. This explains that they share their main concepts.

However, they have three main differences. Firstly, generalized maps allow to represent orientable and non-orientable objects while combinatorial maps allow only to represent orientable objects. Secondly, generalized maps are homogeneous in each dimension since all functions are involutions, while combinatorial maps are not homogeneous since one function is a permutation while other ones are involutions. This homogeneity simplifies algorithms for generalized maps since it allows to avoid a specific case for the first dimension. Thirdly, a generalized map requires twice the number of darts of a combinatorial map in order to represent an orientable object.

Considering these different advantages and drawbacks, you can choose to use generalized maps or combinatorial maps depending on the needs of your application.

3 Software Design

The diagram in Figure 29.10 shows the different classes of the package. `Generalized_map` is the main class (see Section Generalized Maps). It allows to manage darts and attributes (see Section Cell Attributes). Users can customize a generalized map thanks to an items class (see Section Generalized Map Items), which defines the information associated with darts and the attribute types. These types may be different for different dimensions, and they may also be void (note that the main concepts of `GenericMap`, `GenericMapItems` and `CellAttribute` are shared between combinatorial maps and generalized maps).

The darts and attributes are accessed through *handles*. A handle is a model of the `Handle` concept, thus supporting the two dereference operators `operator*` and `operator->`. All handles are model of `LessThanComparable` and `Hashable`, that is they can be used as keys in containers such as `std::map` and `boost::unordered_map`.

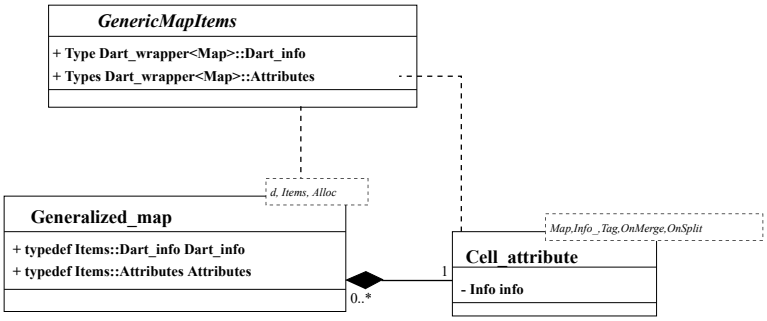


Figure 29.10 UML diagram of the main classes of the package. k is the number of non void attributes.

3.1 Generalized Maps

The class `Generalized_map<d,Items,Alloc>` is a model of the `GeneralizedMap` concept which refines the generic concept of `GenericMap`. It has three template parameters standing for the dimension of the generalized map (an `unsigned int`), an items class (a model of the `GenericMapItems` concept), and an allocator which must be a model of the allocator concept of the STL. Default classes are provided for the items and the allocator classes.

The main role of the class `Generalized_map` is the storage and the management of darts. It allows to create or remove an isolated dart from the generalized map. The `Dart_handle` type defines a handle to the type of used darts (given in the items class). `Generalized_map` provides several *ranges* which allow to iterate over specific subsets of darts of the generalized map (see Section Iterating over Orbits, Cells, and Attributes). It also defines several methods to link and to unlink darts by α_is (see Section Sewing Orbits and Linking Darts). We said that a dart *d0* is *i*-free if $\alpha_i(d0)=d0$. Finally, some high level operations are defined to update the generalized map (see Section Removal and Insertion Operations)

The second role of the class `Generalized_map` is the storage and the management of attributes. It allows to create or remove an attribute, and provides methods to associate attributes and cells. A range is defined for each *i*-attribute allowing to iterate over all the *i*-attributes of the generalized map. Finally, `Generalized_map` defines several types allowing to manage the attributes. We can use `Generalized_map::Attribute_handle<i>::type` for a handle to the *i*-attributes (and the const version `Generalized_map::Attribute_const_handle<i>::type`) and `Generalized_map::Attribute_type<i>::type` for the type of the *i*-attributes.

All information associated to darts (α links and attributes) are accessed through member functions in `GeneralizedMap`.

3.2 Generalized Map Items

The `GenericMapItems` concept defines information associated with darts and attribute types of a generalized map. It contains one inner class named `Dart_wrapper`, having one template parameter, `Map`, a model of `GenericMap` concept. The `Dart_wrapper<Map>` class can provide two local types: `Dart_info` for the information associated with darts, and `Attributes` which defines the attributes and their types.

If `Dart_info` is not defined or if it is equal to `void`, no information is associated with darts.

The `Attributes` tuple must contain at most *d*+1 types (one for each possible cell dimension of the generalized map). Each type of the tuple must be either a model of the `CellAttribute` concept or `void`. The first type corresponds to 0-attributes, the second to 1-attributes and so on. If the i^{th} type in the tuple is `void`, *(i-1)*-attributes are disabled: we say that *(i-1)*-attributes are *void*. Otherwise, *(i-1)*-attributes are enabled and have the given type: we say *(i-1)*-attributes are *non void*. If the size of the tuple is *k*, with $k <$ `dimension` +1, $\forall\ i$: $k \leq i \leq$ `dimension`, *i*-attributes are void. If this type is not defined, all attributes are disabled.

The class `Generic_map_min_items` is a model of the `GenericMapItems` concept which can be used for default behaviors. It defines `void` as type of information associated with darts, and `Attributes` as empty tuple.

3.3 Cell Attributes

The class `Cell_attribute<Map,Info_,Tag,OnMerge,OnSplit>`, a model of the `CellAttribute` concept, represents an attribute associated with a cell of a generalized map. The template parameter `Map` must be a model of the `GenericMap` concept. The attribute stores a handle to one dart of its associated cell when the template parameter `Tag` is `Tag_true`. `Info_` is the type of information stored in the attribute. It may be `void`. `OnMerge` and `OnSplit` must be either `Null_functor`, or models of the `Binary Function` concept having two references to a model of `CellAttribute` as type of both parameters and `void` as return type. There are two default parameters for `OnMerge` and `OnSplit`, which are `Null_functor`, a default parameter for `Tag` which is `Tag_true`, and a default parameter for `Info_` which is `void`.

If `Info_` is different from `void`, the class `Cell_attribute` contains two methods `info()` returning the information contained in the attribute (const and non const version). The information is returned by reference, thus the non const version allows the modification of the information.

Two attributes are merged when their corresponding cells are merged into one cell during some operation. In this case, the functor `OnMerge` is called, unless it is equal to `Null_functor`. This functor allows the user to define its own custom behavior when two attributes are merged (for example if the information is a color, we can compute the average color of the two initial attributes, and affect this value to the first attribute, see example in Section Generalized Map With Attributes). Similarly, the functor `OnSplit` is called when one attribute is split in two, because its corresponding cell is split in two during some operation, unless it is equal to `Null_functor`. In any high level operation, `OnMerge` is called before to start the operation (i.e. before modifying the generalized map), and `OnSplit` is called when the operation is finished (i.e. after all the modifications were made).

In addition, there are dynamic onmerge and onsplit functions that can be associated to i-attributes, and modified, thanks to the `onmerge_function()` and `onsplit_function()`. When these functions are set, they are also called in addition to the previous mechanism when two attributes are merged or one attribute is split into two (see example in Section Use of Dynamic Onmerge and Onsplit Functors).

What we said for the dart also holds for the cell attribute. The generalized map can be used with any user defined model of the `CellAttribute` concept

What we said for the dart also holds for the cell attribute. The generalized map can be used with any user defined model of the `CellAttribute` concept.

3.4 Example of Generalized Map Definition

Here comes an example of two generalized map definitions. The first case `Example_gmap4` defines a 4D generalized map which uses all the default values (`Generic_map_min_items`). The second example `Example_custom_gmap3` uses its own model of the `GenericMapItems` concept. In this model, a `double` is associated as information on darts, and an attribute containing an integer is associated to edges.

```
typedef CGAL::Generalized_map<4> Example_gmap4;

struct Example_items_3
{
  template <class GMap>
  struct Dart_wrapper
  {
    typedef double Dart_info;
    typedef CGAL::Cell_attribute<GMap, int> Edge_attrib;
    typedef std::tuple<void,Edge_attrib> Attributes;
  };
};
typedef CGAL::Generalized_map<3, Example_items_3> Example_custom_gmap3;
```

4 Iteration and Creation Operations

An important operation in generalized maps consists in iterating over specific subsets of darts or over attributes. For that, several *ranges* are offered (see Section Iterating over Orbits, Cells, and Attributes). A range is a model of the `Range` concept, thus supporting the two methods `begin()` and `end()` allowing to iterate over all the elements in the range. Several functions allow to create specific configurations of darts into a generalized map (see Section Construction Operations). Darts can be marked during operations, for example when performing a breadth-first search traversal, thanks to Boolean marks (see Sections Boolean Marks). In the following, we denote by `dh0`, `dh1`, `dh2` the dart handles for the darts `d0`, `d1`, `d2`, respectively. That is `d0 == *dh0`.

4.1 Iterating over Orbits, Cells, and Attributes

The generalized map offers iterators to traverse the darts of a specific orbit, to traverse all darts of one cell, or one dart per cell, and to traverse all *i*-attributes.

Instead of the `begin()/end()` member function pair as we know it from STL containers, and from most CGAL data structures, the generalized map defines range classes which are all models of the `Range` concept.

There are three different categories of dart range classes:

- `Dart_range`: range of all the darts of a generalized map;
- `Dart_of_orbit_range<Alpha...>`: range of all the darts of the orbit $\langle$`Alpha...`$\rangle$(*d0*) for a given *d0*. `Alpha...` is a sequence of integers i_1,..., i_k, each $i_j \in$ {0, ..., *d*}. These integers must satisfy: $i_1 < i_2 <$... $< i_k$ (for example `Dart_of_orbit_range<1,2>` for the orbit $\langle\ \alpha_1, \alpha_2\ \rangle$(*d0*));
- `Dart_of_cell_range<i,dim>`: range of all the darts of the *i*-cell containing a given dart. The *i*-cell is considered in dimension `dim` (with $0 \leq dim \leq d$, *dim=d* by default), with $0 \leq i \leq dim+1$. If *i=dim+1*, `Dart_of_cell_range<i,dim>` is the range of all the darts of the connected component containing a given dart.

There are also two different classes of ranges containing one dart per *i*-cell. Note that in these classes, the dart of each *i*-cell can be any dart of the cell. Moreover, each *i*-cell (and *j*-cell in the second case) is considered in dimension `dim` (with $0 \leq dim \leq d$, *dim=d* by default).

- `One_dart_per_cell_range<i,dim>`: range containing one dart of each *i*-cell of the generalized map, $0 \leq i \leq dim+1$ (for example `One_dart_per_cell_range<2>` for the range of one dart per 2-cell of the generalized map);
- `One_dart_per_incident_cell_range<i,j,dim>`: range containing one dart of each *i*-cell incident to the *j*-cell containing a given dart, with $0 \leq i \leq dim+1$ and $0 \leq j \leq dim+1$ (for example `One_dart_per_incident_cell_range<0,3>` for the range of one dart per vertex of the volume incident to the starting dart). If *i=j*, the range contains only the given dart.

The iterators of the `Dart_range` are bidirectional iterators, while the iterators of the other four ranges are forward iterators. The value type of all these iterators is `Dart` thus all these iterators can be directly used as `Dart_handle`.

Additionally, there is a range over non void *i*-attributes: `Attribute_range<i>::type`, having a bidirectional iterator with value type Attribute_type<i>::type`.

For each range, there is an associated const range, a model of the `ConstRange` concept. You can find some examples of ranges in Section A 3D Generalized Map.

4.2 Construction Operations

Several functions allow to create specific configurations of darts into a generalized map. Existing darts in the generalized map are not modified. Note that the dimension of the generalized map must be large enough: darts must contain all the α pointers used by the operation. All these functions return a `Dart_handle` to a new dart created during the operation.

- `gm.make_edge()`: creates an isolated edge (two darts linked by α_0); dimension must be greater or equal than zero;
- `gm.make_combinatorial_polygon(lg)`: creates an isolated combinatorial polygon of length `lg` (`lg` edges linked by α_1), for `lg>0`; dimension must be greater or equal than one;
- `gm.make_combinatorial_tetrahedron()`: creates an isolated combinatorial tetrahedron (four combinatorial triangles linked together by α_2); dimension must be greater or equal than two;
- `gm.make_combinatorial_hexahedron()`: creates an isolated combinatorial hexahedron (six combinatorial quadrangles linked together by α_2); dimension must be greater or equal than two.

4.3 Boolean Marks

It is often necessary to mark darts, for example to retrieve in *O(1)* if a given dart was already processed during a specific algorithm, for example, iteration over a given range. Users can also mark specific parts of a generalized map (for example mark all the darts belonging to objects having specific semantics). To answer these needs, a `GeneralizedMap` has a certain number of Boolean marks (fixed by the constant `NB_MARKS`). When one wants to use a Boolean mark, the following methods are available (with `gm` an instance of a generalized map):

- get a new free mark: `size_type m = gm.get_new_mark()` (throws the exception Exception_no_more_available_mark if no mark is available);
- set mark `m` for a given dart `d0`: `gm.mark(dh0,m)`;
- unset mark `m` for a given dart `d0`: `gm.unmark(dh0,m)`;
- test if a given dart `d0` is marked for `m`: `gm.is_marked(dh0,m)`;
- unmark all the darts of `gm` for `m`: `gm.unmark_all(m)`;
- negate mark `m` of all the darts of `gm`: `gm.negate_mark(m)`. All the marked darts become unmarked and all the unmarked darts become marked;
- free mark `m`: `gm.free_mark(m)`. This method unmarks all the darts of `gm` for `m` before freeing it.

It is important to free a mark when it is no longer needed, otherwise you may at some point run out of marks.

The following example illustrates how to use marks. Two combinatorial tetrahedra are created and 3-sewn (see Section Sewing Orbits and Linking Darts for a detailed description of the sew operation). Then a mark is reserved and used to mark all the darts belonging to the first combinatorial tetrahedron. Finally, these tetrahedron are merged. The marks allow us to know which darts come from the first and second tetrahedron.

File Generalized_map/gmap_3_marks.cpp

```
#include <CGAL/Generalized_map.h>
#include <iostream>
#include <cstdlib>
```

```
typedef CGAL::Generalized_map<3> GMap_3;
typedef GMap_3::Dart_handle Dart_handle;
typedef GMap_3::size_type size_type;

int main()
{
  GMap_3 gm;

  // 1) Reserve a mark.
  size_type amark;
  try
  {
    amark = gm.get_new_mark();
  }
  catch (GMap_3::Exception_no_more_available_mark)
  {
    std::cerr<<"No more free mark, exit."<<std::endl;
    exit(-1);
  }

  // 2) Create two tetrahedra.
  Dart_handle dh1 = gm.make_combinatorial_tetrahedron();
  Dart_handle dh2 = gm.make_combinatorial_tetrahedron();

  CGAL_assertion( gm.is_valid() );
  CGAL_assertion( gm.is_volume_combinatorial_tetrahedron(dh1) );
  CGAL_assertion( gm.is_volume_combinatorial_tetrahedron(dh2) );

  // 3) 3-sew them.
  gm.sew<3>(dh1, dh2);

  // 4) Mark the darts belonging to the first tetrahedron.

  // Mark the darts belonging to the first tetrahedron.
  for (GMap_3::Dart_of_cell_range<3>::iterator
       it(gm.darts_of_cell<3>(dh1).begin()),
       itend(gm.darts_of_cell<3>(dh1).end()); it!=itend; ++it)
    gm.mark(it, amark);

  // 4) Remove the common 2-cell between the two cubes:
  //     the two tetrahedra are merged.
  gm.remove_cell<2>(dh1);

  // 5) Thanks to the mark, we know which darts come from the first tetrahedron.
  unsigned int res=0;
  for (GMap_3::Dart_range::iterator it(gm.darts().begin()),
         itend(gm.darts().end()); it!=itend; ++it)
  {
    if ( gm.is_marked(it, amark) )
      ++res;
  }

  std::cout<<"Number of darts from the first tetrahedron: "<<res<<std::endl;
  gm.free_mark(amark);

  return EXIT_SUCCESS;
}
```

5 Modification Operations

Several operations allow to modify a given generalized map. There are two main categories of modification operations:

- Sew, link_alpha, unsew and unlink_alpha which modify some existing α pointers, without creating or removing darts (see Section Sewing Orbits and Linking Darts);
- Removal and insertion of cells which modify both darts and α pointers (see Section Removal and Insertion Operations).

5.1 Sewing Orbits and Linking Darts

The `GeneralizedMap` defines two groups of methods to modify the α pointers of existing darts.

The sew and unsew methods iterate over two orbits in order to link or unlink specific darts two by two. Intuitively, a `sew<i>` operation glues two *i*-cells by identifying two of their *(i-1)*-cells (see example in Figure 29.11 where `sew<3>` is used to glue two 3-cells along one 2-cell). Reciprocally, a `unsew<i>` operation un-glues two *i*-cells which were glued along one of their *(i-1)*-cells. These methods guarantee that given a valid generalized map and a possible operation we obtain a valid generalized map as result of the operation.

> ***Advanced***
>
> The `link_alpha` and `unlink_alpha` methods only modify the pointer of two darts: the obtained generalized maps may be not valid. These operations can be useful to use low level operations in a specific algorithm, for example to modify locally a generalized map in a really fast way. In such a case, additional operations may be needed to restore the validity conditions.

Linking two darts *d1* and *d2* by α_i, with $0 \leq i \leq d$ and $d1 \neq d2$, consists in modifying two α_i pointers such that $\alpha_i(d1)=d2$ and $\alpha_i(d2)=d1$.

Reciprocally, unlinking a given dart *d0* by α_i, with $0 \leq i \leq d$, consists in modifying two α_i pointers such that $\alpha_i(\alpha_i(d0))=\alpha_i(d0)$ and $\alpha_i(d0)=d0$. Note that is it possible to unlink a given dart for α_i only if it is not *i*-free.

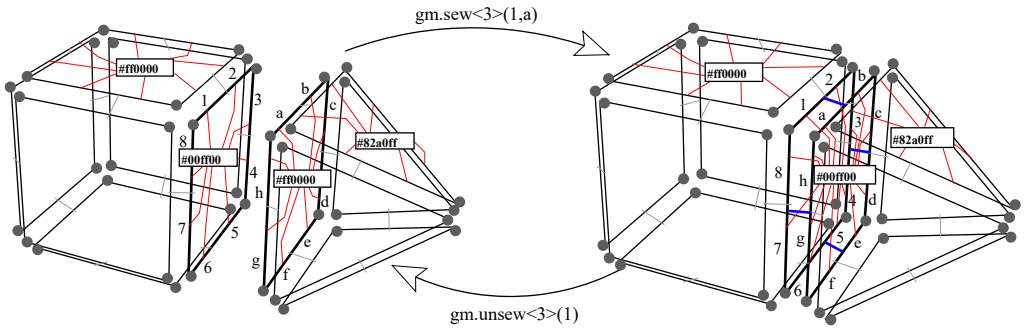


Figure 29.11 Example of 3-sew operation. Left: A 3D generalized map containing two volumes that are not connected, with 2-attributes. Each attribute contains a color in RGB format, and there are four 2-cells associated with attributes. Associations between darts and attributes are drawn with red segments. Right: The 3D generalized map obtained as result of `sew<3>(1,a)` (or `sew<3>(2,b)` ... or `sew<3>(8,h)`). Darts (1,a), ..., (8,h) are linked together by α_3. The two 2-cells *c1*={1,...,8} and *c2*={a,...,h} are merged after the sew into the 2-cell {1,...,8,a,...,h}. We are in the case where the two attributes are non `nullptr`, thus the first one is kept, and all the darts of *c2* are associated with the first attribute.

The `sew<i>(dh1,dh2)` method consists mainly to link two by two several darts by α_i. This operation is possible only if there is a bijection *f* between all the darts of the orbit *D1*= $\langle \alpha_1,..., \alpha_{i-2}, \alpha_{i+2},..., \alpha_d \rangle$(*d1*) and *D2*= $\langle \alpha_1,..., \alpha_{i-2}, \alpha_{i+2},..., \alpha_d \rangle$(*d2*) satisfying: *f*(*d1*)=*d2*, and for all *e* $\in$ *D1*, for all $j \in \{1,..., i-2,i+2,...,d\}$, $f(\alpha_j(e))=\alpha_j^{-1}(f(e))$. Intuitively, this condition ensures the validity of the generalized map by verifying that condition discussed in Section Generalized Map Properties will be satisfied after the operation. This condition can be tested by using the method `is_sewable<i>(dh1,dh2)`. For example, the function `is_sewable<3>` would return `false` if we tried to 3-sew a triangular facet with a quad facet. Note that given two darts *d1* and *d2*, if there is such a bijection, it is uniquely defined. So giving the two darts as arguments of the `sew<i>` is enough to retrieve all the pairs of darts to link. If such a bijection exists, the `sew<i>(dh1,dh2)` operation consists only in linking by α_i each couple of darts *d3* and *d4* such that *d3*=*f*(*d4*).

In addition, the sew operation updates the associations between darts and non void attributes in order to guarantee that all the darts belonging to a given cell are associated with the same attribute (which is a condition of generalized map validity). For each couple of *j*-cells *c1* and *c2* that are merged into one *j*-cell during the sew, we have to update the two associated attributes *attr1* and *attr2*. If both are `nullptr`, there is nothing to do. If one is `nullptr` and the other not, we only associate the non

`nullptr` attribute to all the darts of the resulting cell. When the two attributes are non `nullptr`, we first apply functor `On_merge` on the two attributes *attr1* and *attr2* (see Section Cell Attributes). Then, we associate the attribute *attr1* to all darts of the resulting *j*-cell. Finally, attribute *attr2* is removed from the generalized map.

Note that when the two attributes are non `nullptr`, the first one is kept. But user can customize this behavior in order to update the information contained in the attributes according to its needs. For that, we can define a specific functor, and use it as template argument for `On_merge` parameter of the `Cell_attribute` definition. This functor can for example copy the information of the second attribute in the information of the first one to make as if the second attribute is kept.

For example, in Figure 29.11, we want to 3-sew the two initial volumes. `sew<3>(1,a)` links by α_3 the pairs of darts (1,a), ..., (8,g), thus the generalized map obtained is valid. 2-attributes are updated so that all the darts belonging to the 2-cell containing dart 1 become associated to the same 2-attribute after the operation.

Similarly, `unsew<i>(dh0)` operation unlinks α_i for all the darts in the orbit $\langle \alpha_1,..., \alpha_{i-2}, \alpha_{i+2},..., \alpha_d \rangle$(*d0*), and thus guarantees to obtain a valid generalized map. This operation is possible for any non *i*-free dart.

As for the sew operations, attributes are updated to guarantee that two darts belonging to two different *j*-cells are associated to two different *j*-attributes. If the unsew operation splits a *j*-cell *c* in two *j*-cells *c1* and *c2*, and if *c* is associated to a *j*-attribute *attr1*, then this attribute is duplicated into *attr2*, and all the darts belonging to *c2* are associated with this new attribute. Finally, we call the functor `On_split` on the two attributes *attr1* and *attr2* (see Section Cell Attributes).

Let us consider the generalized map given in Figure 29.11 (Right). If we call `unsew<3>(1)`, we obtain the generalized map in Figure 29.11 (Left) (except for the color of the attribute associated to the 2-cell {a,...,g} which would be `#00ff00`). The `unsew<3>` operation has duplicated the 2-attribute associated to the initial 2-cell {1,...,8,a,...,g} since this 2-cell is split in two after the unsew operation.

Advanced

If one wants to modify a generalized map *manually*, it is possible to switch off the updating between darts and attributes by calling `set_automatic_attributes_management(false)` before to call `sew<i>(dh1,dh2)` and `unsew<i>(dh0)`. In these cases, the generalized map obtained may be no longer valid due to incorrect associations between darts and attributes. A call later to `set_automatic_attributes_management(true)` will correct the invalid non void attributes.

In Figure 29.11 (Left), if we call `sew<3>(1,5)`, the resulting generalized map is similar to the generalized map of Figure 29.11 (Right) (we have linked by α_3 the pairs of darts (1,a), ..., (8,g), but associations between darts and attributes are not valid. Indeed, we have kept the four initial attributes and all the associations between darts and attributes, thus two darts belonging to the same 2-cell (for example darts 1 and a) are associated with two different attributes.

We can also use the `link_alpha<i>(dh1,dh2)` which links `d1` and `d2` by α_i without modifying the other links. Association between darts and attributes are only modified for darts `d1` and `d2`, and similarly as for `sew<i>`, this updating can be avoided by calling `set_automatic_attributes_management(false)` before to call `link_alpha<i>(dh1,dh2)`. Lastly, we can use `unlink_alpha<i>(dh0)` to unlink `d0` for α_i. In this last case, there is no modification of association between darts and attributes.

In Figure 29.11 (Left), if we call `link_alpha<3>(1,a)`, in the resulting generalized map we have now α_3(1)=a and α_3(a)=1. This generalized map is no longer valid (for example dart 2 is 3-free and we should have α_3(2)=b).

Sewing operations can be used in order to build a non-orientable generalized map. Let us consider the 2D generalized map representing a square given in Figure 29.12 (Left). Two opposite edges of the square can be identified by using the `sew<2>` operation. But there are two possibilities to make this identification. The first one, shown in Figure 29.12 (Middle), creates an annulus which is thus orientable. The second one, shown in Figure 29.12 (Right), creates a Möbius strip which is thus non-orientable. The choice of the two darts for the sew operation is thus crucial. See the example A non orientable 2D Generalized Map.

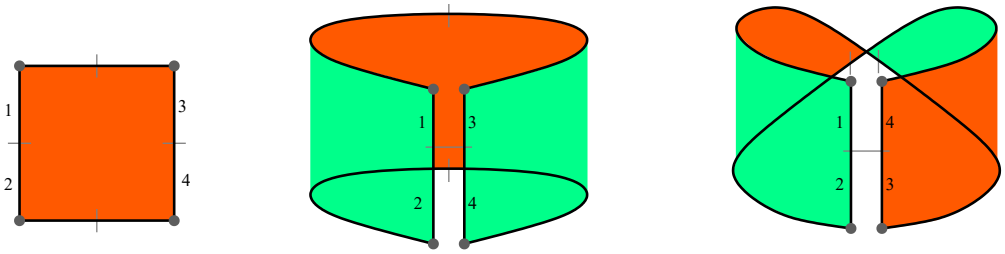


Figure 29.12 Illustration of the use of the 2-sew operation to construct a non-orientable generalized map. Left: A 2D generalized map *M* representing a square (one side colored in orange, the other side in green). Middle: the generalized map obtained from *M* after `sew<2>(1,3)`. This map is orientable and corresponds to an annulus. Right: the generalized map obtained from *M* after `sew<2>(1,4)`. This map is non-orientable and corresponds to a Möbius strip.

5.2 Removal and Insertion Operations

The following high level operations are defined. All these methods ensure that given a valid generalized map and a possible operation, the modified generalized map is also valid.

The first one is `gm.remove_cell<i>(dh0)` which modifies the generalized map to remove the *i*-cell containing dart `d0`, with $0 \leq i \leq d$. This operation is possible if *i=d* or if the given *i*-cell is incident to at most two *(i+1)*-cells which can be tested thanks to `gm.is_removable<i>(dh0)`. If the removed *i*-cell was incident to two different *(i+1)*-cells, these two cells are merged into one *(i+1)*-cell. In this case, the `On_merge` functor is called if two *(i+1)*-attributes are associated to the two *(i+1)*-cells. If the *i*-cell is associated with a non void attribute, it is removed from the generalized map (see three examples on Figure 29.13, Figure 29.15 and Figure 29.16).

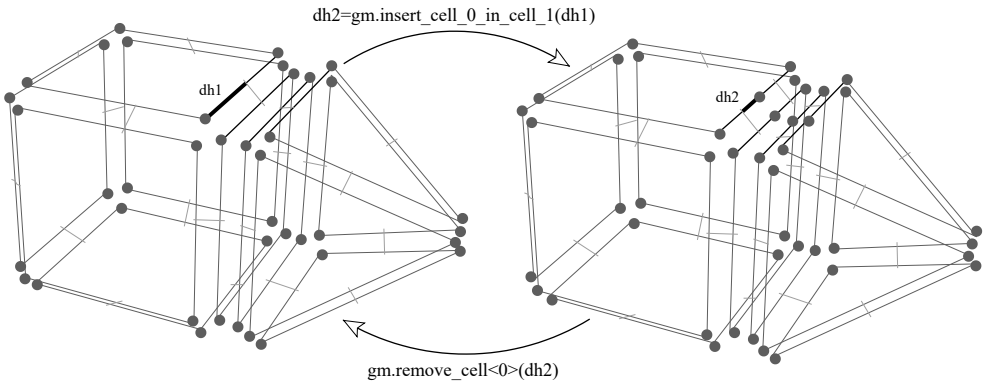


Figure 29.13 Example of `insert_cell_0_in_cell_1` and `remove_cell<0>` operations. Left: Initial generalized map. Right: After the insertion of a 0-cell in the 1-cell containing dart `d1`. Now if we remove the 0-cell containing dart `d2`, we obtain the initial generalized map.

The inverse operation of the removal is the insertion operation. Several versions exist, sharing a common principle. They consist in adding a new *i*-cell *inside* an existing *j*-cell, $i < j$, by splitting the *j*-cell into several *j*-cells. Contrary to `remove_cell<i>`, is it not possible to define a unique `insert_cell_i_in_cell_j<i,j>` function because parameters are different depending on `i` and `j`.

`gm.insert_cell_0_in_cell_1(dh0)` adds a 0-cell in the 1-cell containing dart `d0`. The 1-cell is split in two. This operation is possible if `d0` $\in$ `gm.darts()` (see example on Figure 29.13).

`gm.insert_cell_0_in_cell_2(dh0)` adds a 0-cell in the 2-cell containing dart `d0`. The 2-cell is split in triangles, one for each initial edge of the facet. This operation is possible if `d0` ∈ `gm.darts()` (see example on Figure 29.14).

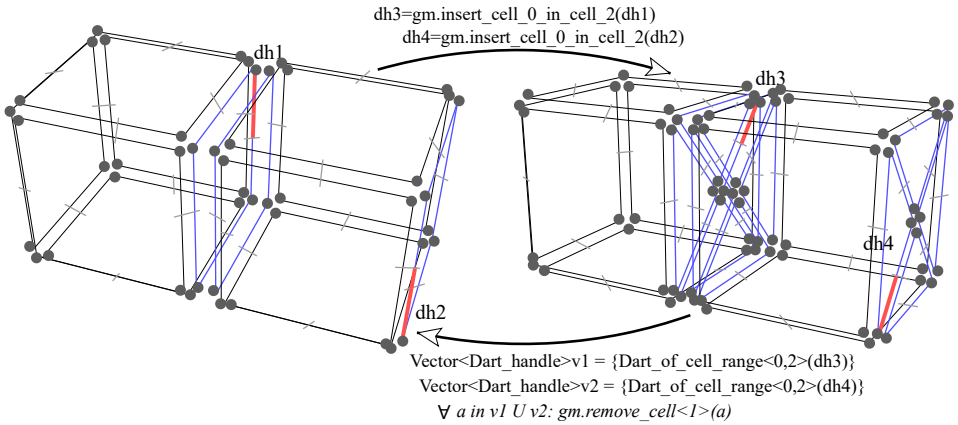


Figure 29.14 Example of `insert_cell_0_in_cell_2` operation.

`gm.insert_cell_1_in_cell_2(dh1,dh2)` adds a 1-cell in the 2-cell containing darts `d1` and `d2`, between the two 0-cells containing darts `d1` and `d2`. The 2-cell is split in two. This operation is possible if *d1* ∈ $\langle \alpha_0, \alpha_1 \rangle$(*d2*) which can be tested thanks to `gm.is_insertable_cell_1_in_cell_2(dh1,dh2)`. In the example on Figure 29.15, it is possible to insert an edge between darts *d2* and *d3*, but it is not possible to insert an edge between *d1* and *d3*.

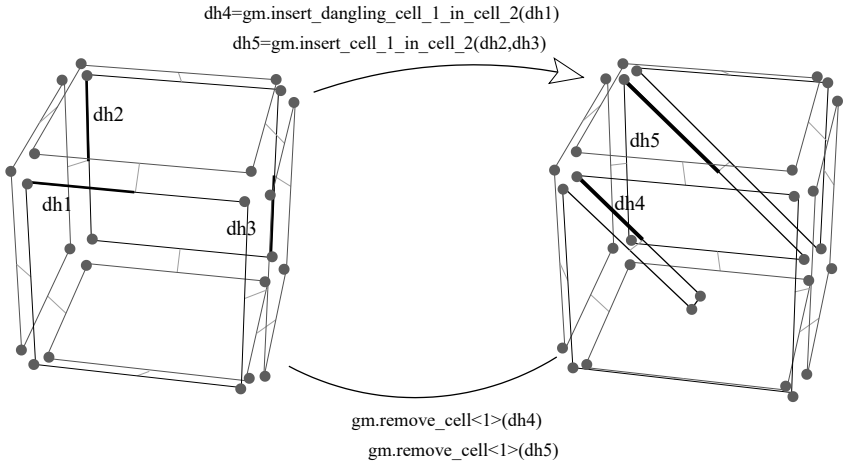
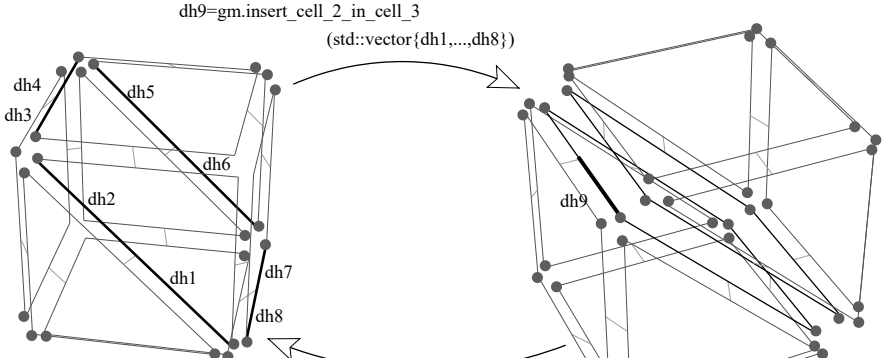


Figure 29.15 Example of `insert_cell_1_in_cell_2` and `remove_cell<1>` operations. Left: Initial generalized map. Right: After the insertion of two 1-cells: a first one between the two 0-cells containing darts `d2` and `d3`, and a second one incident to the 0-cell containing dart `d1`. Now if we remove the two 1-cells containing darts `d4` and `d5`, we obtain the initial generalized map.

`gm.insert_dangling_cell_1_in_cell_2(dh0)` adds a 1-cell in the 2-cell containing dart `d0`, the 1-cell being attached by only one of its vertex to the 0-cell containing dart `d0`. This operation is possible if `d0` ∈ `gm.darts()`.

`gm.insert_cell_2_in_cell_3(itbegin,itend)` adds a 2-cell in the 3-cell containing all the darts between `itbegin` and `itend`, along the path of 1-cells containing darts in [`itbegin`,`itend`). The 3-cell is split in two. This operation is possible if all the darts in [`itbegin`,`itend`) form a closed path inside a same 3-cell which can be tested thanks to `gm.is_insertable_cell_2_in_cell_3(itbegin,itend)` (see example on Figure 29.16).



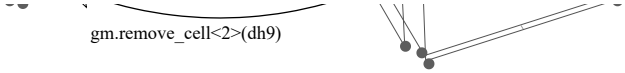


Figure 29.16 Example of `insert_cell_2_in_cell_3` and `remove_cell<2>` operations. Left: Initial generalized map. Right: After the insertion of a 2-cell along the path of 1-cells containing respectively `d1,d2,d3,d4`. Now if we remove the 2-cell containing dart `d5`, we obtain the initial generalized map.

As the sew operation, insertion operations could create non-orientable generalized map. This is for example the case if we start from the 3D generalized map given in Figure 29.15 (Left) and insert an edge not between darts `d2` and `d3` but between darts `d2` and α_1(`d3`).

Some examples of use of these operations are given in Section High Level Operations.

Advanced

If `set_automatic_attributes_management(false)` is called, all the future insertion or removal operations will not update non void attributes. These attributes will be updated later by the call to `set_automatic_attributes_management(true)`. This can be useful to speed up an algorithm which uses several successive insertion and removal operations. See example Automatic attributes management.

6 Examples

6.1 A 3D Generalized Map

In this example, a 3-dimensional generalized map is constructed. Two combinatorial tetrahedra are created, then the numbers of cells of the generalized map are displayed, and the validity of the generalized map is checked. Then, we illustrate the use of ranges to iterate over specific darts. The first loop enumerates all the darts of the first tetrahedron by using the range `Dart_of_orbit_range<0,1,2>`, and the second loop enumerates all the darts of the facet containing dart `dh2` by using the range `Dart_of_orbit_range<0,1>`.

File Generalized_map/gmap_3_simple_example.cpp

```
#include <CGAL/Generalized_map.h>
#include <iostream>
#include <cstdlib>

typedef CGAL::Generalized_map<3> GMap_3;
typedef GMap_3::Dart_handle Dart_const_handle;

int main()
{
  GMap_3 gm;

  // Create two tetrahedra.
  Dart_const_handle dh1 = gm.make_combinatorial_tetrahedron();
  Dart_const_handle dh2 = gm.make_combinatorial_tetrahedron();

  // Display the generalized map characteristics.
  gm.display_characteristics(std::cout);
  std::cout<<", valid="<<gm.is_valid()<<std::endl;

  unsigned int res = 0;
  // Iterate through all the darts of the first tetrahedron.
  // Note that GMap_3::Dart_of_orbit_range<0,1,2> in 3D is equivalent to
  // GMap_3::Dart_of_cell_range<3>.
  for (GMap_3::Dart_of_orbit_range<0,1,2>::const_iterator
       it(gm.darts_of_orbit<0,1,2>(dh1).begin()),
       itend(gm.darts_of_orbit<0,1,2>(dh1).end()); it!=itend; ++it)
    ++res;

  std::cout<<"Number of darts of the first tetrahedron: "<<res<<std::endl;

  res = 0;
  // Iterate through all the darts of the face incident to dh2.
  for (GMap_3::Dart_of_orbit_range<0,1>::const_iterator
       it(gm.darts_of_orbit<0,1>(dh2).begin()),
       itend(gm.darts_of_orbit<0,1>(dh2).end()); it!=itend; ++it)
    ++res;

  std::cout<<"Number of darts of the face incident to dh2: "<<res<<std::endl;

  return EXIT_SUCCESS;
}
```

The output is:

```
#Darts=48, #0-cells=8, #1-cells=12, #2-cells=8, #3-cells=2, #ccs=2, orientable=true, valid=1
Number of darts of the first tetrahedron: 24
Number of darts of the face incident to d1: 6
```

which gives the number of darts of the generalized map, the numbers of different cells, the number of connected components, a Boolean showing if the generalized map is orientable or not and finally a Boolean showing the validity of the generalized map (a tetrahedron is made up of 48 darts because there are 12 darts per facet and there are 4 facets).

Note the creation in the for loops of the two instances of `Dart_of_orbit_range`::`const_iterator`: `it` is the current iterator, and `itend` an iterator to the end of the range. Having `itend` avoids calling `gm.darts_of_orbit<0,1,2>(dh1).end()` again and again as in the following example (which is a bad solution):

```
for (GMap_3::Dart_of_orbit_range<0,1,2>::const_iterator
     it(gm.darts_of_orbit<0,1,2>(dh1).begin());
     it!=gm.darts_of_orbit<0,1,2>(dh1).end(); ++it)
{...}
```

6.2 A non orientable 2D Generalized Map

In this example, a square is constructed in a 2-dimensional generalized map. Then two darts belonging to two opposite edges of the square are 2-sewn. Since they darts do not belong to the same orientation of the initial square, this creates a torsion and thus leads to a non orientable generalized map which represents a Möbius strip (cf. Figure 29.12 (Right)).

The output is:

```
#Darts=8, #0-cells=2, #1-cells=3, #2-cells=1, #ccs=1, orientable=false, valid=1
```

showing that the generalized map is non orientable.

File Generalized_map/gmap_2_moebius.cpp

```
#include <CGAL/Generalized_map.h>
#include <iostream>
#include <cstdlib>

typedef CGAL::Generalized_map<2> GMap_2;
typedef GMap_2::Dart_handle Dart_handle;

int main()
{
  GMap_2 gm;
  Dart_handle dh=gm.make_combinatorial_polygon(4);
  gm.sew<2>(dh, gm.alpha<1,0,1,0>(dh));

  gm.display_characteristics(std::cout);
  std::cout<<", valid="<<gm.is_valid()<<std::endl;
```

```
  return EXIT_SUCCESS;
}
```

6.3 High Level Operations

This example shows some uses of high level operations. First we create a combinatorial hexahedron, the generalized map obtained is shown in Figure 29.17 (Left). Then we insert two 1-cells along two opposite 2-cells of the hexahedron. The generalized map obtained is shown in Figure 29.17 (Middle). Finally, we insert a 2-cell in the diagonal of the hexahedron in order to split it into two parts. We obtain the generalized map shown in Figure 29.17 (Right). We display the characteristics of the generalized map and check its validity.

The second part of this example removes the inserted elements. First we remove the inserted 2-cell, then the two inserted 1-cells. We get back the initial combinatorial hexahedron, which is verified by displaying once again the characteristics of the generalized map.

File Generalized_map/gmap_3_operations.cpp

```
#include <CGAL/Generalized_map.h>
#include <iostream>
#include <cstdlib>

typedef CGAL::Generalized_map<3> GMap_3;
typedef GMap_3::Dart_handle      Dart_handle;

int main()
{
  GMap_3 gm;

  // Create one combinatorial hexahedron.
  Dart_handle d1 = gm.make_combinatorial_hexahedron();

  // Add two edges along two opposite facets.
  gm.insert_cell_1_in_cell_2(d1,gm.alpha<0,1,0>(d1));
  CGAL_assertion( gm.is_valid() );

  Dart_handle d2=gm.alpha<2,1,0,1,2>(d1);
  gm.insert_cell_1_in_cell_2(d2,gm.alpha<0,1,0>(d2));
  CGAL_assertion( gm.is_valid() );

  // Insert a facet along these two new edges plus two initial edges
  // of the hexahedron.
  std::vector<Dart_handle> path;
  path.push_back(gm.alpha<1>(d1));
  path.push_back(gm.alpha<1,0,1,2,1>(d1));
  path.push_back(gm.alpha<1,0>(d2));
  path.push_back(gm.alpha<2,1>(d2));

  Dart_handle d3=gm.insert_cell_2_in_cell_3(path.begin(),path.end());
  CGAL_assertion( gm.is_valid() );

  // Display the generalized map characteristics.
  gm.display_characteristics(std::cout) << ", valid=" <<
                                       gm.is_valid() << std::endl;

   // We use the removal operations to get back to the initial hexahedron.
  gm.remove_cell<2>(d3);
  CGAL_assertion( gm.is_valid() );

  gm.remove_cell<1>(gm.alpha<1>(d1));
  CGAL_assertion( gm.is_valid() );

  gm.remove_cell<1>(gm.alpha<1>(d2));
  CGAL_assertion( gm.is_valid() );
  CGAL_assertion( gm.is_volume_combinatorial_hexahedron(d1) );

  // Display the generalized map characteristics.
  gm.display_characteristics(std::cout) << ", valid="
                                        << gm.is_valid() << std::endl;

  return EXIT_SUCCESS;
}
```

The output is:

```
#Darts=72, #0-cells=8, #1-cells=14, #2-cells=9, #3-cells=2, #ccs=1, orientable=true, valid=1
#Darts=48, #0-cells=8, #1-cells=12, #2-cells=6, #3-cells=1, #ccs=1, orientable=true, valid=1
```

The first line gives the characteristics of the generalized map after all the insertions (the generalized map drawn in Figure 29.17 (Right)). There are two 3-cells (since the combinatorial hexahedron was split in two by the 2-cell insertion), nine 2-cells (since two 2-cells of the original hexahedron were split in two by the two 1-cell insertions, and a new 2-cell was created during the 2-cell insertion), fourteen 1-cells (since there are two new 1-cells created by the 1-cell insertion) while the number of 0-cells remains unchanged.

The second line is the result after the removal operations. We retrieve the original combinatorial hexahedron since we have removed all the inserted elements.

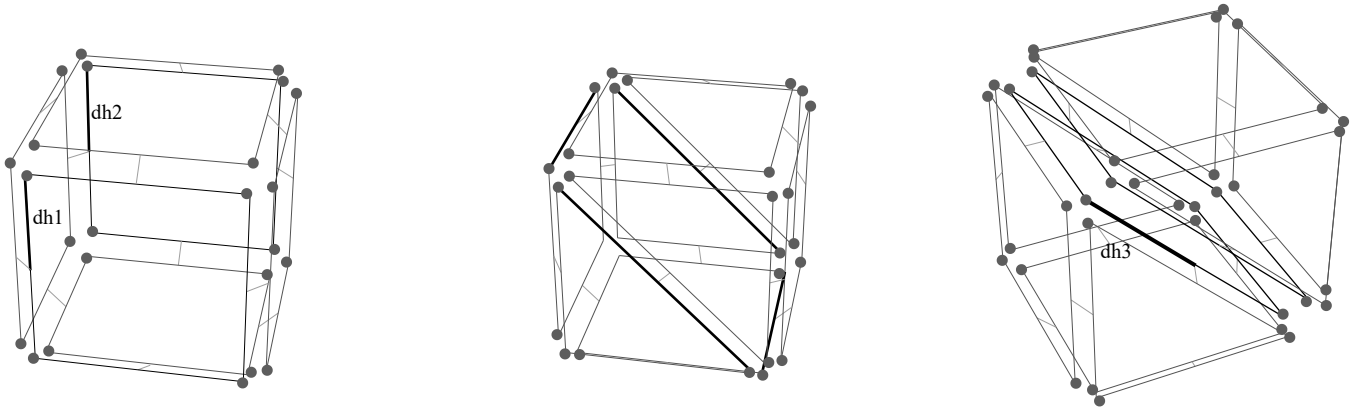


Figure 29.17 Example of high level operations. Left: Initial 3D generalized map after the creation of the generalized hexahedron. Middle: Generalized map obtained after

the two 1-cell insertions. The two 2-cells were split in two. Right: Generalized map obtained after the 2-cell insertion. The 3-cell was split in two.

6.4 A 4D Generalized Map

In this example, a 4-dimensional generalized map is used. Two tetrahedral cells are created and sewn by α_4. Then the numbers of cells of the generalized map are displayed, and its validity is checked.

By looking at these numbers of cells, we can see that the 4D generalized map contains only one 3-cell. Indeed, the `sew<4>` operation has identified by pairs all the darts of the two 3-cells by definition of the sew operation (see Section Sewing Orbits and Linking Darts) which, in 4D, links by α_3 all the darts in $\langle \alpha_1, \alpha_2 \rangle(d1)$ and in $\langle \alpha_1, \alpha_2 \rangle(d2)$. The situation is similar (but in higher dimension) to the configuration where we have two triangles in a 3D generalized map, and you use `sew<3>` between these two triangles. The two triangles are identified since all their darts are linked by α_3, thus we obtain a 3D generalized map containing only one 3-cell. Note that this 3-cell is unbounded since the darts of the two triangles are all 2-free. In the 4D case, the 4-cell is unbounded since all its darts are 3-free.

In this example, we also illustrate how to use the basic methods to build *by hand* some specific configuration in a generalized map. In fact, these functions are already present in the package: function `make_triangle(gm)` is equivalent to `gm.make_combinatorial_polygon(3)` and `make_tetrahedron(gm)` is equivalent to `gm.make_combinatorial_tetrahedron()`. If we want to create a 4D simplex, we must create five 3D simplexes, and sew them correctly two by two by α_3 (and so on if you want to create higher dimensional generalized map).

File Generalized_map/gmap_4_simple_example.cpp

```
#include <CGAL/Generalized_map.h>
#include <iostream>
#include <cstdlib>

typedef CGAL::Generalized_map<4> GMap_4;
typedef GMap_4::Dart_handle Dart_handle;

int main()
{
  GMap_4 gm;
  Dart_handle d1 = gm.make_combinatorial_tetrahedron();
  Dart_handle d2 = gm.make_combinatorial_tetrahedron();

  CGAL_assertion(gm.is_valid());

  gm.sew<4>(d1,d2);

  gm.display_characteristics(std::cout);
  std::cout<<", valid="<<gm.is_valid()<<std::endl;

  return EXIT_SUCCESS;
}
```

The output is:

```
#Darts=48, #0-cells=4, #1-cells=6, #2-cells=4, #3-cells=1, #4-cells=2, #ccs=1, orientable=true, valid=1
```

6.5 Generalized Map With Attributes

In the following example, we illustrate how to specify the 2-attributes in a 3D generalized map. For that, we define our own item class using `Cell_attribute<GMap,int,Tag_true,Sum_functor,Divide_by_two_functor>` as attributes which contain an `int` and which are associated to 2-cells of the generalized map.

Functors `Sum_functor` and `Divide_by_two_functor` define a custom behavior: when two attributes `ca1` and `ca2` are merged into `ca1`, the value of `ca1` is the sum of the two initial values. When an attribute `ca1` is split in the two attributes `ca1` and `ca2`, the value of each attribute is half of the first value.

File Generalized_map/gmap_3_with_colored_facets.cpp

```
#include <CGAL/Generalized_map.h>
#include <CGAL/Cell_attribute.h>
#include <iostream>
#include <algorithm>
#include <cstdlib>

struct Sum_functor
{
  template<class Cell_attribute>
  void operator()(Cell_attribute& ca1,Cell_attribute& ca2)
  { ca1.info()=ca1.info()+ca2.info(); }
};
struct Divide_by_two_functor
{
  template<class Cell_attribute>
  void operator()(Cell_attribute& ca1,Cell_attribute& ca2)
  {
    ca1.info()=(ca1.info()/2);
    ca2.info()=(ca1.info());
  }
};

struct Myitem
{
  template<class GMap>
  struct Dart_wrapper
  {
    typedef CGAL::Cell_attribute<GMap, int, CGAL::Tag_true,
     Sum_functor, Divide_by_two_functor> Facet_attribute;
    typedef std::tuple<void,void,Facet_attribute> Attributes;
  };
};

typedef CGAL::Generalized_map<3,Myitem> GMap_3;
typedef GMap_3::Dart_handle             Dart_handle;

int main()
{
  GMap_3 gm;

  // Create 2 hexahedra.
  Dart_handle dh1 = gm.make_combinatorial_hexahedron();
  Dart_handle dh2 = gm.make_combinatorial_hexahedron();

  // 1) Create all 2-attributes and associated them to darts.
  for (GMap_3::Dart_range::iterator
         it=gm.darts().begin(), itend=gm.darts().end();
       it!=itend; ++it)
  {
    if ( gm.attribute<2>(it)==nullptr )
      gm.set_attribute<2>(it, gm.create_attribute<2>());
  }

  // 2) Set the color of all facets of the first hexahedron to 7.
  for (GMap_3::One_dart_per_incident_cell_range<2, 3>::iterator
         it=gm.one_dart_per_incident_cell<2,3>(dh1).begin(),
         itend=gm.one_dart_per_incident_cell<2,3>(dh1).end(); it!=itend; ++it)
  { gm.info<2>(it)=7; }

  // 3) Set the color of all facets of the second hexahedron to 13.
  for (GMap_3::One_dart_per_incident_cell_range<2, 3>::iterator it=
         gm.one_dart_per_incident_cell<2,3>(dh2).begin(),
         itend=gm.one_dart_per_incident_cell<2,3>(dh2).end(); it!=itend; ++it)
  { gm.info<2>(it)=13; }

  // 4) 3-Sew the two hexahedra along one facet.
  gm.sew<3>(dh1, dh2);

  // 5) Display all the values of 2-attributes.
  for (GMap_3::Attribute_range<2>::type::iterator
       it=gm.attributes<2>().begin(), itend=gm.attributes<2>().end();
       it!=itend; ++it)
  {
    std::cout<<gm.info_of_attribute<2>(it)<<" ";
```

```
    std::cout<<gm.info_of_attribute<2>(it)<<"; ";
  }
  std::cout<<std::endl;

  // 6) Insert a vertex in the facet between the two hexahedra.
  gm.insert_cell_0_in_cell_2(dh2);

  // 7) Display all the values of 2-attributes.
  for (GMap_3::Attribute_range<2>::type::iterator
         it=gm.attributes<2>().begin(), itend=gm.attributes<2>().end();
       it!=itend; ++it)
  {
    std::cout<<gm.info_of_attribute<2>(it)<<"; ";
  }
  std::cout<<std::endl;
  gm.display_characteristics(std::cout);
  std::cout<<", valid="<<gm.is_valid()<<std::endl;

  return EXIT_SUCCESS;
}
```

The output is:

```
20; 7; 7; 7; 7; 7; 13; 13; 13; 13; 13;
2; 7; 7; 7; 7; 7; 10; 13; 13; 13; 13; 5; 2;
#Darts=128, #0-cells=13, #1-cells=24, #2-cells=14, #3-cells=2, #ccs=1, orientable=true, valid=1
```

Before the `gm.sew<3>`, each 2-cell of the first cube is associated with an attribute having 7 as value, and each 2-cell of the second cube with an attribute having 13 as value. During the `gm.sew<3>`, two 2-cells are merged, thus the functor `Sum_functor` is called on the two associated 2-attributes, and the value of the new 2-cell is the sum of the two previous one: 20.

Then we call `insert_cell_0_in_cell_2` on a dart which belong to this 2-cell. This method splits the existing 2-cell in *k* 2-cells, *k* being the number of 1-cells of the initial 2-cell (4 in this example). These splits are made consecutively, thus for the first split, we create a new attribute as copy of the initial one and call functor `Divide_by_two_functor` on these two attributes: the value of each attribute is thus 20/2=10. For the second split, the value of each attribute is thus 10/2=5, and for the last split the value of each attribute is thus 5/2=2 (remember that information contained in 2-attributes in an `int`). At the end, we obtain five 2-attributes with 7 as value, five 2-attributes with 13 as value, and four 2-attributes having respectively 2, 2, 5 and 10 as values.

6.6 Use of Dynamic Onmerge and Onsplit Functors

In the following example, we show an example of use of dynamic onmerge and onsplit functor. We define our 3D generalized map with 2-attributes. Then we create two hexahedra and create all the 2-attributes, having their info initialized to 1.

Step 2 defines the onsplit and onmerge dynamic functors. We can see here that with this mechanism, functors can store data member. This is the case in the example for `Split_functor` which stores a reference to the generalized map.

The next operations will call these functors when 2-cells are split or merged. The `sew<3>` operation calls 1 onmerge as two faces are identified; the `insert_cell_0_in_cell_2` operation calls 3 onsplit as one face is split in 4.

Lastly we remove the dynamic onmerge functor (step 7). This is done by initializing the fonctor to a default boost::function. After this initialization, no dynamic merge functor is called when two faces are merged.

File Generalized_map/gmap_3_dynamic_onmerge.cpp

```
#include <CGAL/Generalized_map.h>
#include <CGAL/Cell_attribute.h>
#include <iostream>
#include <cstdlib>

// My item class: no static functor is associated with Face_attribute.
struct Myitem
{
  template<class GMap>
  struct Dart_wrapper
  {
    typedef CGAL::Cell_attribute<GMap, double> Face_attribute; // A weight
    typedef std::tuple<void,void,Face_attribute> Attributes;
  };
};

// Definition of my generalized map.
typedef CGAL::Generalized_map<3,Myitem> GMap_3;
typedef GMap_3::Dart_handle             Dart_handle;
typedef GMap_3::Attribute_type<2>::type Face_attribute;

// Functor called when two faces are merged.
struct Merge_functor
{
  // operator() automatically called before a merge.
  void operator()(Face_attribute& ca1, Face_attribute& ca2)
  {
    ca1.info()=ca1.info()+ca2.info(); // Update can be done incrementally.
    std::cout<<"After on merge faces: info of face1="<<ca1.info()
             <<", info of face2="<<ca2.info()<<std::endl;
  }
};

// Functor called when one face is split in two.
struct Split_functor
{
  Split_functor(GMap_3& amap) : mmap(amap)
  {}

  // operator() automatically called after a split.
  void operator()(Face_attribute& ca1, Face_attribute& ca2)
  {
    // We need to reinitalize the weight of the two faces
    GMap_3::size_type nb1=mmap.darts_of_cell<2>(ca1.dart()).size();
    GMap_3::size_type nb2=mmap.darts_of_cell<2>(ca2.dart()).size();
    mmap.info<2>(ca1.dart())*=(double(nb1)/(nb1+nb2));
    mmap.info<2>(ca2.dart())*=(double(nb2)/(nb1+nb2));
    std::cout<<"After on split faces: info of face1="<<ca1.info()
             <<", info of face2="<<ca2.info()<<std::endl;
  }

private:
  GMap_3& mmap;
};

// Function allowing to display all the 2-attributes, and the characteristics
// of a given combinatorial map.
void display_map_and_2attributes(GMap_3& gm)
{
  for (GMap_3::Attribute_range<2>::type::iterator
       it=gm.attributes<2>().begin(), itend=gm.attributes<2>().end();
       it!=itend; ++it)
  { std::cout<<gm.info_of_attribute<2>(it)<<"; "; }
  std::cout<<std::endl;
  gm.display_characteristics(std::cout);
  std::cout<<", valid="<<gm.is_valid()<<std::endl;
}

int main()
{
  GMap_3 gm;

  // 0) Create 2 hexahedra.
  Dart_handle dh1 = gm.make_combinatorial_hexahedron();
  Dart_handle dh2 = gm.make_combinatorial_hexahedron();

  // 1) Create and initialize 2-attributes.
  for (GMap_3::One_dart_per_cell_range<2>::iterator
       it=gm.one_dart_per_cell<2>().begin(),
       itend=gm.one_dart_per_cell<2>().end(); it!=itend; ++it)
  {
    gm.set_attribute<2>(it, gm.create_attribute<2>(1));
  }

  // 2) Set the onsplit and onmerge functors
```

```
  // 2) Set the onsplit and onmerge functors.
  gm.onsplit_functor<2>()=Split_functor(gm);
  gm.onmerge_functor<2>()=Merge_functor();

  // 3) 3-Sew the two hexahedra along one face. This calls 1 onmerge.
  gm.sew<3>(dh1, dh2);

  // 4) Display all the values of 2-attributes.
  display_map_and_2attributes(gm);

  // 5) Insert a vertex in the face between the two hexahedra.
  //    This calls 3 onsplit.
  Dart_handle resdart=gm.insert_cell_0_in_cell_2(dh2);

  // 6) Display all the values of 2-attributes.
  display_map_and_2attributes(gm);

  // 7) "Remove" the dynamic onmerge functor.
  gm.onmerge_functor<2>()=boost::function<void(Face_attribute&,
                                               Face_attribute&)>();

  // 8) Remove one edge: this merges two faces, however no dynamic
  //    functor is called (because it was removed).
  gm.remove_cell<1>(resdart);

  // 9) Display all the values of 2-attributes.
  display_map_and_2attributes(gm);

  return EXIT_SUCCESS;
}
```

7 Mathematical Definitions

The definition of generalized map in any dimension is given in [2], [3]. See also the book [1] which regroups many definitions, operations and algorithms about combinatorial and generalized maps.

An *involution* on a finite set *E* is a mapping *f* from *E* to *E* which is bijective and equal to its inverse. Thus $\forall$ *e* $\in$ *E*, we have *f*(*e*) = f^{-1}(*e*) and *f*(*f*(*e*))=*e*.

Let *d* $\geq$ 0. A *d*-dimensional generalized map (or *d*-Gmap) is a (d+1)-tuple *G*=(*D*, α_0,..., α_d) where:

1. *D* is a finite set of darts;
2. $\forall$ *i*, 0 $\leq$ *i* $\leq$ *d*, α_i is an involution on *D*;
3. $\forall$ *i*: 0 $\leq$ *i* $\leq$ *d*-2, $\forall$ *j*: 2 $\leq$ *j* $\leq$ *d*, *i*+2 $\leq$ *j*, $\alpha_i \circ \alpha_j$ is an involution.

A *d*-dimensional generalized map represents a subdivision of an orientable or non-orientable *d*-dimensional quasi-manifold. A dart is an abstract element which is only required to define involutions. The last line of the definition fixes constraints which guarantee the topological validity of the represented object, i.e., the fact that it is a quasi-manifold. This definition allows us to verify the validity of a given generalized map by checking if each item of the definition is satisfied.

Given a set of involutions *S*= $\{f_1,..., f_k\}$, we denote by $\langle$ *S* $\rangle$ the *permutation group* generated by $\{f_1,..., f_k\}$ and whose group operation is the composition of involutions. The orbit $\langle$ $f_1,..., f_k$ $\rangle$(*a*) is the set of darts which can be obtained from *a* by elements of $\langle$ *S* $\rangle$: $\langle$ $f_1,..., f_k$ $\rangle$(*a*)= $\{\phi(a) \mid \phi \in \langle S \rangle\}$.

Let *d0* $\in$ *D* be a dart. Given *i*, 0 $\leq$ *i* $\leq$ *d*, the *i*-cell containing *d0* is $\langle$ $\alpha_0,..., \alpha_{i-1}, \alpha_{i+1},..., \alpha_d$ $\rangle$(*d0*).

8 Design and Implementation History

The code of this package followed the code of Combinatorial maps and was inspired by Moka, a 3D topological modeler that uses 3D generalized maps (http://moka-modeller.sourceforge.net/).