A Triangulation Template Library (TTL): Generic Design of Triangulation Software

Øyvind Hjelle

November 21, 2000

Abstract

Triangulations representing surfaces constructed from scattered measurement data can be dealt with algebraically using principles of generalized maps, or G-maps. High level abstraction of functions operating on triangulations is achieved using G-maps, which are developed for general boundary based topological models. Generic design of a Delaunay triangulation algorithm and of functions operating on triangulations are presented, and principles of implementation using function templates in the C++ programming language are shown. Since the operations can be algebraically defined and implemented generically, a triangulation library based on this concept can operate on arbitrary data structures.

Contents

1 Introduction

Triangulations are used in many application domains for representing two-manifold polygonal surface structures on computers. Flight simulators use triangle-based surfaces for landscape visualization [21], digital terrain models are represented as triangulations in cartography [4], and finite element methods (FEM) use them as a basis for numerical calculations [23, 5]. Triangulations are otherwise widely used in visualization and computer graphics [9, 16].

A triangulation needs an underlying data structure for representing its topological entities: nodes, edges and triangles, their topological adjacency information, and their associated geometric embedding information. Different applications need different data structures depending on the needs in the actual application.

Applications based on triangulations need a number of topological and geometric operations using the topological and geometric entities represented by the data structure of the triangulation. These operations can briefly be divided into two groups: *modifiers* that change the topology or the geometry, and *queries* that extract information from the triangulation. Examples of modifiers are edge-swapping operations and refinement operations based on point insertion. Examples of queries are simple operations like examining neighbour relationships between nodes, edges and triangles, and operations extracting triangle strips for visualization.

It is customary to implement such functions in a static manner in the sence that they are adapted to one specific data structure only. The consequences are that the functions must be totally reimplemented if the underlying data structure is changed. Moreover, in many applications more than one data structure is needed, for example to speed up certain topological operations, and thus, different adaptions to the application data structures are needed.

The reason for this rigid approach has probably been the lack of an abstract description of the topology structure and of functions operating on the topology. Generalized maps, or G-maps, which are introduced later, provide a formal algebraic approach to model general boundary based topological models (B-reps). Its algebraic definition is generic in the sense that it is not dependent on the underlying data structure. This suggests that algorithms based on G-maps can be separated from the data structure and work only with the algebraic concepts that G-maps are based on.

The aim of this report is first to show how G-maps can be used as robust and rigorous algebraic tools to model the topology of triangulations. Having defined the topology of triangulations and functions operating on them algebraically, the concept of *function templates* in the C++ programming language is employed to implement a generic library for triangulations using the principles of G-maps. The generic library will hereafter be called TTL (Triangulation Template Library). The TTL will comprise a set of queries and modifiers, and moreover, a Delaunay triangulation algorithm that can operate on arbitrary application data structures.

In the next section, the concepts of G-maps are introduced and applied to

triangulations. In Section 3, design of the TTL and its interplay with applications are outlined. Queries and modifiers implemented as function templates in C++ are exemplified using the half-edge data structure which is a common data structure for boundary based topological models. Finally, Section 4 is devoted to a generic incremental Delaunay triangulation algorithm.

2 Generalized Maps and Data Structures

The Triangulation Template Library (TTL) to be introduced in Section 3 is based on so-called *Generalized maps*, or G-maps [14, 2]. G-maps are general tools for modelling the topology of boundary based geometric models and have been applied in application areas such as geological modelling [7, 24]. It is algebraically defined based on a few numbers of clear concepts. The topology is described using a single topological element, the *dart*, and a set of functions operating on the set of darts in the topology structure. In this section G-maps are applied to obtain an algebraic description of the topology of triangulations, and thus achieving an abstract level on which generic functions in the TTL can operate. A common data structure for triangulations, the half-edge data structure, is also presented for later use when demonstrating generic functionality in the TTL.

2.1 G-maps Applied to Trianguations

A dart in a triangulation structure can be considered as a unique triple $d = (V_i, E_j, T_k)$, where V_i is a node (or a vertex) of the edge E_j, and V_i and E_j is a node and an edge of the triangle T_k. Thus, for each triangle there are six possible combinations of the triple that can define a dart. In Figure 2.1(a) a dart d is indicated both as an arrow and as a (node, edge, triangle)-triple.

Referring to Figure 2.1(b), we define three unique functions, α_0, α_1 and α_2 operating on the set of triples D in a triangulation as one-to-one mappings from D onto itself,

$$\alpha_i : D \to D,\ i = 0, 1, 2.$$

In the theory of generalized maps these functions are called involutions as they are bijections with the property $\alpha_i(\alpha_i(d)) = d$. We will simply call them α-*iterators* and define them as follows when applied to triangulations:

- $\alpha_0(d)$ maps d to a triple with a different node, but keeps the edge and the triangle fixed,
- $\alpha_1(d)$ maps d to a triple with a different edge, but keeps the node and the triangle fixed,
- $\alpha_2(d)$ maps d to a triple with a different triangle, but keeps the node and the edge fixed.

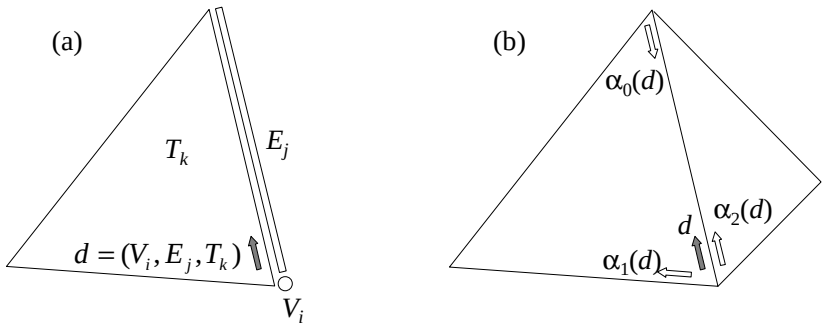

Figure 2.1: (a): A dart considered as a unique triple $d = (V_i, E_j, T_k)$. (b): Shows how d is mapped under the functions $\alpha_i,\ i = 0, 1, 2$.

That is, $\alpha_0(d)$, $\alpha_1(d)$ and $\alpha_2(d)$ change the node, edge and triangle of the triple $d = (V_i, E_j, T_k)$ respectively, while the other topological elements are kept fixed. If an edge E_j in a triple $d = (V_i, E_j, T_k)$ is at the boundary of the triangulation, then $\alpha_2(d) = d$, hence the node, the edge and the triangle are all kept fixed. This is called a *fixed point* of the α_2-iterator. With the definitions above a triangulation can be considered as a *G-map*,

$$G(D, \alpha_0, \alpha_1, \alpha_2),$$

which defines a combinatorial structure, or an algebra, on the set of darts D.

More informally, one can think of a dart $d = (V_i, E_j, T_k)$ as a an element positioned inside the triangle T_k at the node V_i and at the edge E_j as depicted in Figure 2.1(a). The dart changes its content and position in the triangulation structure according to the definition of $\alpha_i,\ i = 0, 1, 2$ operating on d. The α_0 and α_1 iterators "reposition" the dart inside the same triangle while α_2 "moves" the dart over an edge to the neighbour triangle of T_k.

We will also say that a dart has a clockwise or counterclockwise direction with respect to a triangle as seen from one side of the triangulation. Thus, the dart d in Figure 2.1(a) is oriented counterclockwise in the triangle as determined by the direction of the arrow symbolizing the dart. Note that $\alpha_i(d),\ i = 1, 2, 3$ changes the direction of d from clockwise to counterclockwise, or vice versa, except when α_2 is applied to a dart positioned at an edge at the boundary of the triangulation. We also define the regions "*left of d*" and "*right of d*" in the plane as seen from one side of the triangle and in the direction of the arrow. Lastly, a dart $d = (V_i, E_j, T_k)$ can also be interpreted as a vector with direction from V_i to the opposite node of V_i, that is, to the node V_i' associated with the triple of the dart $d' = (V_i', E_j, T_k) = \alpha_0(d)$.

A G-map, and thus a triangulation, has a corresponding labelled graph that can be considered as the dual of the G-map [15]. The triangulation in Figure 2.2 consists of three triangles drawn with dashed edges, and the labelled graph is

drawn with bold edges. Each node of the graph corresponds to a dart drawn as an arrow close to the node and pointing in the direction of the node. The labelled edges incident with[1] each node correspond to the α-iterators applied to the dart that correspond to that node. For example, an edge labelled "1" links two nodes of the graph corresponding to two darts d_i and d_j that are linked through the α_1-iterator; $\alpha_1(d_i) = d_j$ and $\alpha_1(d_j) = d_i$. The dashed edges labelled "2" correspond to fixed points of the α_2-iterator at the boundary of the triangulation. The graph is *regular of degree three* in the sense that every node has exactly three edges incident with it.

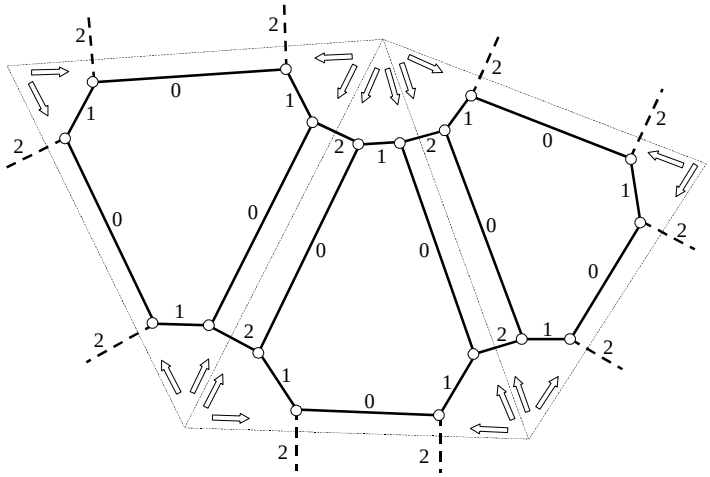

Figure 2.2: A triangulation with three triangles, and its corresponding labelled graph defined as the dual of the G-map.

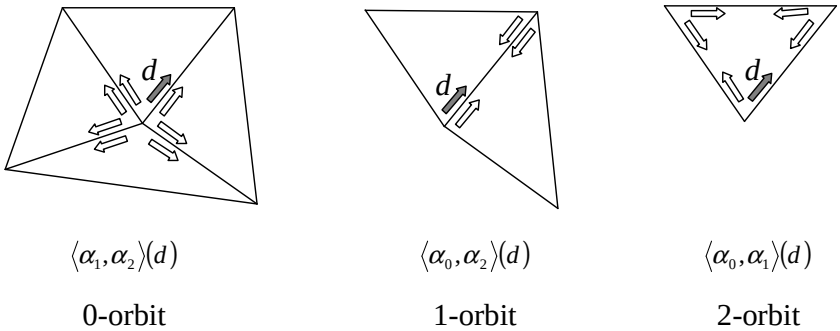

Figure 2.3: The k-orbits, $k = 0, 1, 2$, of a dart d in a G-map

[1] We say that an edge with nodes V_i and V_j as endpoints is *incident with* V_i and V_j.

In the following we denote a *composition* $\alpha_i(\alpha_j(\cdots\alpha_k(d)\cdots))$ by $\alpha_i \circ \alpha_j \circ \cdots \circ \alpha_k(d)$. Note that a composition $\alpha_i \circ \alpha_j(d)$, with $i \neq j$, does not change the orientation of d inside a triangle from clockwise to counterclockwise, or vice versa, unless the composition involves $\alpha_2(d)$ where d is at the boundary of the triangulation. Note also the following interesting properties that we will use later when composing iterators operating on triangulations.

- Applying the composition $\alpha_0 \circ \alpha_1$ repeatedly iterates through the nodes and the edges of a triangle.
- Applying the composition $\alpha_1 \circ \alpha_2$ repeatedly iterates through all edges and triangles sharing a common node.
- Applying the composition $\alpha_0 \circ \alpha_2$ repeatedly iterates around an edge by visiting the two nodes of the edge and the two triangles sharing the edge.
- If the order of a composition $\alpha_i \circ \alpha_j$ is swapped to $\alpha_j \circ \alpha_i$, the iteration goes in the opposite direction.

In addition we have,

$$\begin{aligned} \alpha_i \circ \alpha_i(d) &= d,\ i = 0, 1, 2, \text{ and} \\ (\alpha_0 \circ \alpha_2(d))^2 &= \alpha_0 \circ \alpha_2 \circ \alpha_0 \circ \alpha_2(d) = d. \end{aligned}$$

Definition 2.1 (Orbit and k-orbit, $k = 0, 1, 2$) *Let $\{\alpha_i\}$ be one, two or all three α-iterators of a G-map $G(D, \alpha_0, \alpha_1, \alpha_2)$ and let $d \in D$. An orbit $\langle\{\alpha_i\}\rangle(d)$ of a dart d is the set of all darts in D that can be reached by successively applying compositions of α-iterators in $\{\alpha_i\}$ in any order starting from d. The k-orbit of a dart d in a G-map is defined as the orbit $\langle\alpha_i, \alpha_j,\ i, j \neq k,\ i \neq j\rangle(d)$.*

Thus, the 0-orbit, $\langle\alpha_1, \alpha_2\rangle(d)$, and the 1-orbit, $\langle\alpha_0, \alpha_2\rangle(d)$, is the set of all darts around a node and around an edge respectively. The 2-orbit, $\langle\alpha_0, \alpha_1\rangle(d)$, is the set of all darts inside a triangle; see Figure 2.3. We observe that the orbit $\langle\alpha_0, \alpha_1, \alpha_2\rangle(d)$, d being any dart of D, is the set of all darts in D provided that all triangles in the triangulation associated with G are connected edge by edge. Thus, all the topological elements, nodes, edges and triangles, of a triangulation can be reached by applying the orbit $\langle\alpha_0, \alpha_1, \alpha_2\rangle(d)$ starting from an arbitrary dart $d = (V_i, E_j, T_k)$.

The concept of darts and α-iterators can be generalized to arbitrary dimensions n such that for an *n-G-map* we have iterators $\alpha_i,\ i = 0, \ldots, n$. For example, for $n = 3$ the topology of tetrahedrizations can be modelled by associating a dart with a quadruple, $(node, edge, triangle, tetrahedron)$, and iterators $\alpha_i,\ i = 0, 1, 2, 3$, and compositions of them can be used for navigating in the topology structure as similarly described above for triangulations. Also, arbitrary boundary based (B-rep) models can be modelled by G-maps; that is, faces need not be triangles and volumes need not be tetrahedra.

The α-iterators are basically the only traversal operators needed on a triangulation. Thus they can be used as an interface to arbitrary data structures, and generic algorithms for navigating in the topology can be based solely on these iterators. By compositions $\alpha_i \circ \alpha_j \circ \cdots \circ \alpha_k(d)$ it is then possible to build any traversal operator needed by an application. In Section 3 we specify in detail how this can be implemented generically, using function templates in the C++ programming language.

2.2 The Half-Edge Data Structure

There are many possible topological structures, or data structures, for representing triangulations on computers. A data structure must be chosen in view of the needs and requirements in the actual application. When analyzing different data structures one always faces a trade-off between storage requirements and efficiency of carrying out topological and geometric operations. For example, for visualization purposes one needs a data structure with fast access to data and sufficient topological information for traversing the topology when extracting sequences of triangles for the visualization system. This will normally require more storage than a data structure used only for storing a triangulation in a database. In a real application one might need more than one data structure and tools for mapping one to the other. A detailed analysis of different data structures for triangulations is given in [8].

The notion of *half-edge* as the basic topological entity for boundary based topological representations was introduced by Weiler [27]. The principle is to split each edge of a triangle into two directed half-edges, each of which are oriented opposite to the other. Hence, we can think of a half-edge as belonging to exactly one triangle, and the three half-edges of a triangle can be oriented counterclockwise around the triangle as seen from one side of the triangulation. A half-edge has its *source node* where it starts from, and its *target node* where it points to. In Figure 2.4(a) is shown a half-edge representation associated with a triangulation with six triangles. By graph terms this data structure reflects a planar directed multigraph [17].

A minimal pointer structure, minimal in the sense of storage requirement while maintaining "sufficient" information for topological operators, is shown in Figure 2.4(b) for two of the triangles. Each half-edge has a pointer to the node it starts from, a pointer to the next half-edge in the triangle, in counterclockwise direction, and a pointer to its "twin-edge" belonging to a neighbour triangle.

When using object-oriented programming languages as C++ and Java for implementing data structures for triangulations, it is natural to think of the topological entities nodes, edges and triangles as *classes*. For example, the half-edge data structure would be represented as a node class, a half-edge class and possibly a triangle class. Figure 2.5 depicts a class diagram for a half-edge data structure.

The triangle class consists of a pointer to one of its half-edges, a half-edge class has a pointer to its source node, and pointers to the next half-edge in the triangle and to its twin-edge in the adjacent triangle. The node class may

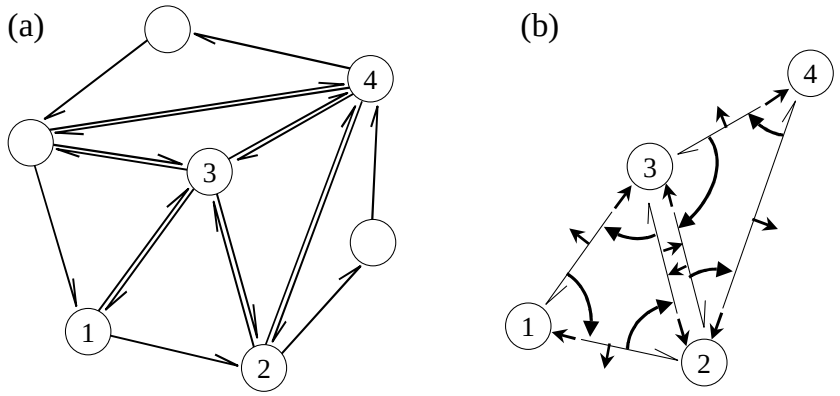

Figure 2.4: Half-edge data structure

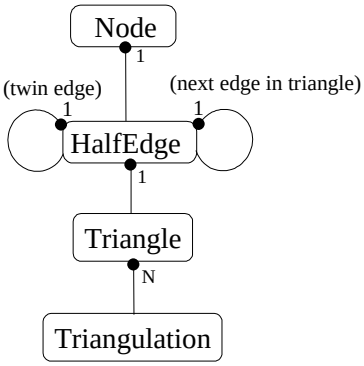

Figure 2.5: Class diagram for the half-edge data structure

carry its geometric positional information in the Euclidean space. In addition, we may have a triangulation class which contains a list of pointers to triangles. We also need member functions in the classes such that their information is accessible from "outside" the classes. For the half-edge class this would typically be `getSourceNode`, `getNextHalfEdge`, and `getTwinEdge`.

Note that a half-edge has its counterpart in a G-map as a dart, say d_i. Thus, the function `GetNextHalfEdge` has its counterpart in the G-map as the composition $\alpha_1 \circ \alpha_0(d_i)$, and likewise, the function `getTwinEdge` has its counterpart in the composition $\alpha_0 \circ \alpha_2(d_i)$. We will elaborate further on this when designing generic functions for triangulations in the sequel.

3 Triangulation Template Library (TTL)

The concept of G-maps introduced in the previous section provides a simple but vigorous algebraic tool for modelling the topology of triangulations. It is well founded on a few clear concepts: one single topological element, the dart, and iterators α_0, α_1 and α_2 operating on the set of darts for navigating in the topology structure. G-maps' algebraic definition enables generic implementation such that algorithms can be clearly separated from the underlying data structure.

An object-oriented design based on G-maps was presented in [7] for topological kernels in 3D geological modelling. The dart was implemented as an abstract base class in C++ with α-iterators as pure virtual member functions. The topological kernel was adapted to the application data structure by deriving a class from the abstract dart class and implementing the α-iterators with access to the actual data structure. The drawbacks with this solution are that the application must rely on a fixed limited interface of the abstract base class, and that dynamic type checking when using class inheritance slows down topological operations [25].

Another approach, without class inheritance, was discussed by Arge [1] with the aim of making a clear distinction between topology and geometric embedding information of triangulations. A scheme was provided for the application programmer to adapt algorithms based on G-maps to arbitrary data structures for triangulations.

In the following, we extend these ideas and suggest a generic library for triangulations implemented using the concept of *function templates* in C++ [25]. The generic library, which we denote TTL (Triangulation Template Library), communicates with the application data structure by means of the dart algebra of G-maps outlined in Section 2.1 through an interface which is provided by the application programmer; see Figure 3.1. Thus, the TTL is totally independent of the underlying data structure. This is in contrast to the traditional rigid design of triangulation software with algorithms working directly on a specific data structure as depicted in Figure 3.2. Implementation of the interface between the TTL and the application data structure is exemplified using the half-edge data structure described in Section 2.2.

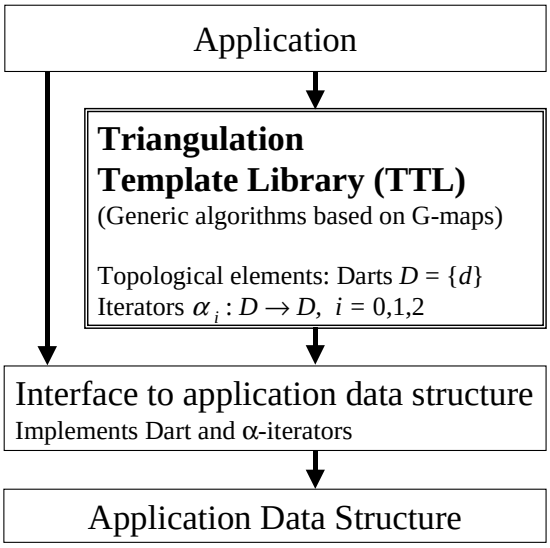

Figure 3.1: Generic design of triangulation software using G-maps

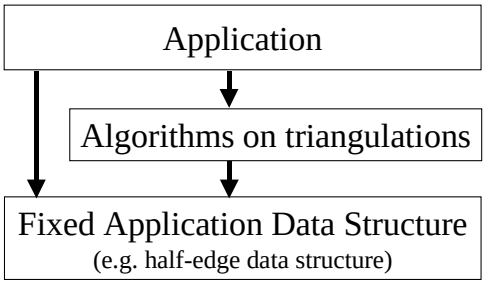

Figure 3.2: Traditional rigid design of triangulation software

3.1 Topological Queries

One class of topological operations on a triangulation consists of functions that are pure queries on the topology structure. A number of such functions are needed in commercial triangulation software, such as accessing edges incident with a given node, finding triangles adjacent to a given triangle, traversing the boundary of the triangulation, etc. These queries can easily be implemented generically in the TTL using the concepts of darts and α-iterators in G-maps.

We start with a very simple function, that of deciding if an edge of a triangle is at the boundary of the triangulation. Recall from the algebraic description of G-maps in Section 2.1 that we allowed for fixed points for the α_2-iterator. Thus, if an edge E_j of a dart $d = (V_i, E_j, T_k)$ is at the boundary of the triangulation, then $\alpha_2(d) = d$. A pseudo code for a function `isBoundaryEdge` that finds if E_j is a boundary edge can be written simply as:

if $\alpha_2(d)$ $==$ d
    return TRUE
else
    return FALSE

Note that the behaviour of this function is not affected by how the incoming dart is implemented, that is, how nodes, edges and triangles of the triangulation are represented in the actual data structure. This suggests that a C++ function template defines `isBoundaryEdge` [18]. If, for a certain data structure, the dart is defined as a class in C++ and the α_2-iterator is implemented as a member function `alpha2` of the dart class, then the algorithm above can be implemented generically using a function template parametrized on a dart type.

```
template <class DartType>
bool isBoundaryEdge(const DartType& dart) {
  DartType dart_iter = dart;
  if (dart_iter.alpha2() == dart)
    return true;
  else
    return false;
}
```

This generic function can be used by applications based on any data structure for triangulations if a proper implementation of a dart class is provided by the application as an interface to the actual data structure. The function template expects that a member function `alpha2` is present in the dart class and should return a dart as defined by the α_2-iterator. In addition, the function template expects a copy constructor and a boolean `operator==` for comparing dart objects. According to the definition of a fixed point, `alpha2` should leave the dart unchanged if the edge associated with the dart is at the boundary of the triangulation.

Similarly, a family of other topological queries can be implemented as function templates. Different queries will require that different member functions are implemented in the dart class. But in general, for topological queries where the topology of the triangulation structure is not changed, only the α-iterators need to be present, in addition to standard class member functions such as constructors, assignment operators and the like. The code listing in Figure 3.3 shows an example of a dart class that can serve as an interface between the TTL and the half-edge data structure. The member functions `alpha0`, `alpha1` and `alpha2` correspond to the iterators α_0, α_1 and α_2 respectively. It is assumed that a class `HalfEdge` is implemented with member functions `getNextHalfEdge` and `getTwinEdge` as explained in Section 2.2.

The data members of the dart class consist of a pointer to a half-edge and a boolean variable indicating whether the dart is positioned at the source node of the half-edge or at the target node of the half-edge. The α-iterators change the content of the dart and return a reference to the dart itself. This is for convenience, as the function templates become more compact and are easier to write; see `isBoundaryEdge` above. Thus, the dart is implemented as a dynamic element which changes its content and position in the triangulation through the α-iterators. The member function `alpha0`, which corresponds to the α_0-iterator, just switches the boolean variable `dir_` such that the dart is repositioned to the other node of the edge. The member functions `alpha1` and `alpha2` need some more operations to carry out the α_1 and α_2 operations. Note that `alpha2` assumes that `HalfEdge::getTwinEdge` returns `NULL` if the edge is at the boundary of the triangulation. This corresponds to a fixed point of the α_2-iterator.

The simple exercise outlined above also applies for other boundary based models where faces are not necessarily triangles. The only assumption above about faces being triangles is the member function `alpha1` of the dart class in Figure 3.3, which must be modified slightly to handle faces with arbitrary number of edges.

In addition to `isBoundaryEdge` a number of other topological queries are needed in an application. The following simple functions are parametrized on a dart type and will work properly if the dart is implemented as shown in Figure 3.3.

- `bool isBoundaryTriangle(const DartType& d);`
  The given dart $d = (V_i, E_j, T_k)$ is an arbitrary one positioned in the triangle T_k to be examined. Edges are checked as in `isBoundaryEdge` above and the composition $\alpha_1 \circ \alpha_0(d)$ is used to reposition the dart in the 2-orbit to the next edge in the triangle.

- `bool isBoundaryNode(const DartType& d);`
  The composition $\alpha_1 \circ \alpha_2(d)$ repositions the dart in the 0-orbit to an edge of the next triangle that has the node of the given dart as a member. The node is at the boundary of the triangulation if one of the edges is. Each edge is checked as explained previously.

```
class Dart {
private:
  HalfEdge* edge_;
  bool  dir_;

public:

  // Constructors and destructors
  ...
  Dart& operator= (const Dart& dart) {. . .} // assignment

  bool operator==(const Dart& dart) const {
    if (dart.edge_ == edge_ && dart.dir_ == dir_)
      return true;
    return false;
  }

  bool operator!=(const Dart& dart) const {
    return !(dart==*this);
  }

  Dart& alpha0() {dir_ = !dir_; return *this;}

  Dart& alpha1() {
    if (dir_ == true) {
      edge_ = edge_->getNextEdgeInFace()->getNextEdgeInFace();
    else
      edge_ = edge_->getNextEdgeInFace();
    dir_ = !dir_;
    return *this;
  }

  Dart& alpha2() {
    // Check if the dart is on the boundary. If yes, the dart
    // will not be changed.
    if (edge_->getTwinEdge()) { // Check if on boundary
      edge_ = edge_->getTwindEdge();
      dir_ = !dir_;
    }
    return *this;
  }
};
```

Figure 3.3: Example of a dart class for the half-edge data structure.

- `int getDegreeOfNode(const DartType& d);`
  The degree (or valency) of a node V_i in a triangulation, is defined as the number of edges incident with V_i, that is, the number of edges joining V_i with another node. The edges are counted by counting the number of $\alpha_2 \circ \alpha_1$ compositions that can be done in the 0-orbit until the given dart is reached again.

The last function is slightly more involved since the node may be at the boundary of the triangulation, in which case a fixed point for the α_2-iteration is reached. Assuming that the actual node is not at the boundary, the function can be implemented as:

```
template <class DartType>
int getDegreeOfNode(const DartType& dart) {
  DartType dart_iter = dart;
  int degree = 0;
  do {
    dart_iter.alpha1().alpha2();
    ++degree;
  } while (dart_iter != dart);
  return degree;
}
```

Other queries commonly present in triangulation software are functions that return topological elements from the triangulation structure. If a fixed number of topological elements should be returned, for example three triangles T_1, T_2 and T_3 adjacent to a given triangle, then the triangles can be returned by a function template as three darts d_{t1}, d_{t2} and d_{t3} that have T_1, T_2 and T_3 as members of their respective dart triples. The declaration may read as follows:

- `void getAdjacentTriangles(constDartType&d,`
  `DartType&dt1, DartType& dt2, DartType& dt3);`
  If the given triangle is at the boundary of the triangulation, one or more of the returned darts will indicate a "NULL object". A default constructor may prepare such an object, and a boolean function `Dart::isNull()` may indicate if the dart is a NULL object.

The class template `list` in STL, The Standard Template Library which is part of the C++ standard, can be used if an unknown number of topological elements should be given or returned from a function. Still, the function template can be parametrized on a dart type only. A function for finding the boundary of a triangulation may be declared thus:

- `void getBoundary(const DartType& d,`
  `list<DartType>& boundary);`
  This function assumes that a dart at the boundary of the triangulation is given as input. The 0-orbit at each boundary node is iterated with $\alpha_2 \circ \alpha_1$

compositions and the function returns an STL list of darts representing all the edges at the boundary. (Alternatively, the list type can also be parametrized by the function template.)

We have parametrized the function templates on a dart type only to keep the interfaces clean and to limit the functionality needed on the application side. It may well turn out that other solutions are more efficient. An alternative to returning topological elements from the functions as darts is to parametrize the function templates on `NodeType`, `EdgeType` and `TriangleType`. The actual data structure needs not contain these types, but there must be a mechanism, for example in the dart class, for retrieving a reference (or a pointer) to a node, an edge or a triangle from the triple represented by a dart. Thus, `getBoundary` above may return `list<EdgeType*>` representing the boundary as a list of pointers to edges. This would save memory and avoid copying of dart objects, but it would require some more implementation by the application programmer.

The k-orbits in Definition 2.1 represent circular sequences in the sense that when traversing the darts of an orbit in one direction the same dart is reached again (though this is not the case for a 0-orbit at a boundary node). Thus, they are different from linear sequences supported by iterators and container classes in STL. The simple concept of *circulators* [10] can be used to integrate such circular sequences in the framework of STL. Traversal of darts in a k-orbit can then be done using the same syntax as when traversing a list or a vector in STL, and moreover, generic algorithms in STL can operate on the k-orbits.

3.2 Geometric Queries

In the previous section, only topological information was used by the function templates, and the topological operations required in the dart class was limited to the α-iterators only. In this section the TTL is extended to handle geometric positional embedding information of nodes in the triangulation structure. The geometric operations will still be at a query level in the sense that no changes of the triangulation take place through the operations. Topological and geometric operations that modify the triangulation will be discussed in Section 3.3.

A simple example involving geometric calculations is that of finding if a given point p is located inside a given triangle. This operation is needed in many contexts, for example when evaluating surface triangulations, and when inserting a new point into an existing triangulation. The latter is dealt with in Section 4 in connection with an incremental Delaunay triangulation algorithm. Assume that a dart $d = (V_i, E_j, T_k)$ is oriented counterclockwise in the triangle T_k. Let $H(d)$ denote the half-plane to the left of d, and containing V_i and the opposite node to V_i, that is, the node V_i' associated with the dart $d' = \alpha_0(d)$. The region defined by the intersection

$$H(d) \cap H\left(\alpha_1 \circ \alpha_0(d)\right) \cap H\left(\alpha_0 \circ \alpha_1(d)\right) \tag{3.1}$$

is the triangle T_i where the dart d is positioned. Thus, p is located in T_i if and only if p lies in the three half-planes of expression (3.1). The query $p \in H(d)$

can be implemented by evaluating the sign of a determinant. Let (x_1, y_1) and (x_2, y_2) be the positions in the plane of V_i and V_i', and let $p = (x_3, y_3)$. The determinant,

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 - x_1y_3 - x_2y_1 + x_2y_3 + x_3y_1 - x_3y_2,$$

evaluates to zero if p is on the line between V_i and V_i', and it is greater than zero if p is to the left of the line between V_i and V_i'. There are many possible ways of solving the point location problem, and similar problems, using function templates. In principle there are two roads ahead. One leads to implementing functionality like the query $p \in H(d)$ inside the TTL. Using the same approach as for topological query operations in Section 3.1, the geometric operations required in the dart class could simply be the access functions, `double Dart::x()`, and likewise for y and z, which return the position in space of a node associated with a dart. The TTL would then implement standard computational geometry functionality necessary for executing the library functions. The problem with this approach is that the application programmer must rely on a fixed computational geometry library hidden inside the TTL.

The other approach leads to a higher level of abstraction by requiring that (low level) computational geometry functionality be provided from outside the TTL. It is then the application's task to implement this functionality. This approach suggests that the query $p \in H(d)$ be implemented as a boolean function `inLeftHalfPlane(Point2d&)` in the dart class[2]. Although some more implementation work is required by the application programmer, the latter approach might be preferable for several reasons. The most important is that the application programmer can choose the level of accuracy. Tests like $p \in H(d)$ can be implemented simply as above by evaluating the sign of a determinant. But the test involves floating point arithmetic which may lead to an incorrect result due to round-off errors when the determinant is near zero. The problem can be solved using exact arithmetic, but then the speed would probably be reduced by orders of magnitude. Shewchuk [22] describes different techniques for solving the problem above and related problems involving point and vector algebra, with different levels of accuracy. A function `inTriangle` can now be implemented in C++ thus:

```
template <class Point2dType, class DartType>
bool inTriangle(const Point2dType& point, DartType& dart) {
  for (int i = 0; i < 3; i++) {
    if (!dart.inLeftHalfPlane(point))
      return false;
    dart.alpha0().alpha1();
```

[2]The function `inLeftHalfPlane(...)` could also be a free function, a function in a name space, or in a traits class.

```
  }
  return true;
}
```

The function template is parametrized on a point type in addition to a dart type. The triangle in question is given as a dart that has the triangle as a member of its triple. A boolean function `inLeftHalfPlane` must be implemented in the dart class and should return `false` if the given point is not positioned to the left of the dart. The function also assumes that the given dart is oriented counterclockwise in the triangle.

A function common in triangulation software is that of locating the triangle in a triangulation containing a given point. It can be implemented based on the same functionality in the dart class as required by `inTriangle` above. Note first how fixed points for the α_2-iterator at the boundary of the triangulation can be handled. If for some dart d_b we have $\alpha_2(d_b) = d_b$, then d_b is positioned at a boundary edge. If d_b is oriented counterclockwise in the triangle and the boundary of the triangulation is convex, then p lies outside the triangulation if $p \notin H(d_b)$. The following takes as input a point p in the plane and an arbitrary dart $d_i = (V_i, E_j, T_k)$ oriented counterclockwise in T_k. It is assumed that a half-plane $H(d_i)$ is defined as above.

Algorithm 3.1 (`Dart locateTriangle(Point` p`, Dart` d_i`, bool` $found$`)`)

1. $d_{start} = d_i$
2. if $p \in H(d_i)$ // is p in the half-plane $H(d_i)$?
3. $d_i = \alpha_1 \circ \alpha_0(d_i)$ // next edge counterclockwise (ccw.)
4. if $d_i == d_{start}$
5. $found$ = TRUE, RETURN d_i // inside triangle of d_i
6. else // try to move to the adjacent triangle
7. if $\alpha_2(d_i) == d_i$ // check if on boundary
8. $found$ = FALSE, RETURN d_i // outside triangulation
9. $d_{start} = \alpha_0 \circ \alpha_2(d_i)$
10. $d_i = \alpha_1 \circ \alpha_2(d_i)$ // next edge ccw. in adjacent triangle
11. GOTO Step 2

Figure 3.4 shows how the given dart d_i is successively repositioned until it reaches the triangle where p is located. The composition $\alpha_1 \circ \alpha_0(d_i)$ (Step 3) moves the dart inside a triangle, and with the composition $\alpha_1 \circ \alpha_2(d_i)$ (Step 10), the dart is moved across an edge of two adjacent triangles and positioned at the next edge. Since a composition of two α-iterators is used to move the dart, it is

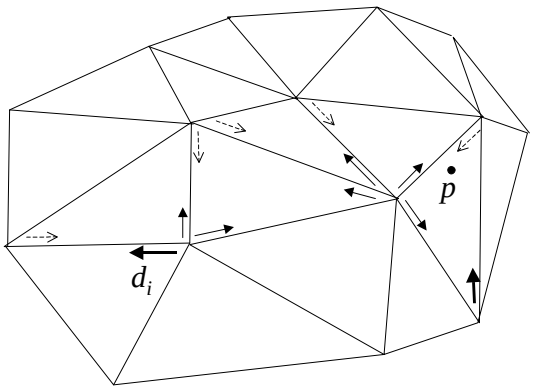

Figure 3.4: Localizing a point p starting from a dart d_i.

always kept in counterclockwise direction inside a triangle. The dashed darts in the figure represent d_{start} in Step 9. No triangle or half-plane associated with an edge is ever considered twice. The algorithm terminates with d_i in the located triangle, or it terminates at Step 8 with d_i at the boundary if p is outside the triangulation. In the latter case of Step 8, we have a fixed point, $\alpha_2(d_i) = d_i$, and $p \notin H(d_i)$. It is assumed that the boundary of the triangulation is convex and that there are no holes in the triangulation. A triangle is also located if p is on an edge or on a node. The algorithm is fast, but it is not robust with respect to triangles that are degenerate or almost degenerate.

Similarly, a range of methods commonly present in triangulation software can be implemented generically as function templates based on a limited number of (primitive) geometric operations required in the interface to the actual data structure. In addition to `inLeftHalfPlane`, scalar product and cross product between vectors (as defined by darts) will suffice to cover a variety of geometric functions.

3.3 Geometric and Topological Modifiers

The scope so far has been on generic function templates that are queries, in the sense that the topology and the geometry of the triangulation is not changed through the operations. Below we discuss how the TTL can be extended to include operations that *modify* the topology and the geometric embedding information of triangulations.

In many applications, for example terrain modelling, the only geometric embedding information in a tringulation is the 3D positions of nodes. Thus, a natural extension of the concepts above is to require elementary geometric modifiers in the dart class, for example, `Dart::setNodeX(realType x)`, and likewise for y and z, that modify the position in space of a node associated with a dart. Together with corresponding access functions and topological query

functions based on α-iterators, this would suffice for functions that modify the geometry of a triangulation.

Topological modifiers are more complex than geometric modifiers. They can briefly be subdivided into three main categories.

1. Modifiers that preserve all nodes and the number of edges and triangles. This can be done by swapping edges that are diagonals of strictly convex quadrilaterals. It is well known that all possible triangulations of a set of points can be reached by a sequence of edge-swaps starting from an initial triangulation of the points [11].

2. Modifiers that remove nodes edges and triangles. Removing one of these elements may also imply removing some of the others. For example, removing an internal edge also implies removing two triangles.

3. Modifiers that add nodes, edges and triangles in a triangulation.

Several extensions of the TTL described previously are necessary to incorporate topological modifiers as generic functions, and more effort will be put on the application programmer to implement counterparts interfacing the actual data structure.

Topological modifiers can be described algebraically using topological operators based on the concept of *sewing* in G-maps. These operators establish the involutions in a G-map, that is, the relationships between darts as defined through α-iterators. Two darts d_i and d_j are said to be *k-sewed,* or α_k*-sewed*, in a G-map if $\alpha_k(d_i) = d_j$ (and $\alpha_k(d_j) = d_i$). This may suggest that topological modifiers in the TTL can be based on sewing-operators if it is possible to implement associated counterparts in the interface to the application data structure.

Let us analyze the edge-swapping operation mentioned above algebraically through sewing-operators. Figure 3.5 shows two triangles forming a convex quadrilateral in a triangulation. The edge E_i with nodes V_1 and V_2 can be swapped to become a new edge E_i' with nodes V_3 and V_4. Apart from assigning new nodes to E_i', a total of six sewing-operations are required to perform the edge-swapping: each unprimed dart d_i in the figure must be α_1-sewed with the primed dart d_i', $i = 1, \ldots, 6$ for establishing a correct topology. This involves all darts of the two triangles. Only α_1-sewings are involved in the swapping, α_0-sewings and α_2-sewings are all maintained.

If we try to implement these lower level sewing-operators in an interface to the half-edge data structure, we may face severe problems. An attempt to implement a function like `sew1(`d_1, d_2`)`, that establishes the relationship $\alpha_1(d_1) = d_2$ and $\alpha_1(d_2) = d_1$, would destroy the pointer structure and create an intermediate topological representation that is not legal for a triangulation.

Another approach is to require that a higher level function `swapEdge(Dart d)`, that swaps an edge associated with a dart, be implemented in the interface to the data structure (and not as part of the TTL). The responsibility for lower level operations carrying out the edge-swap on the actual data structure would

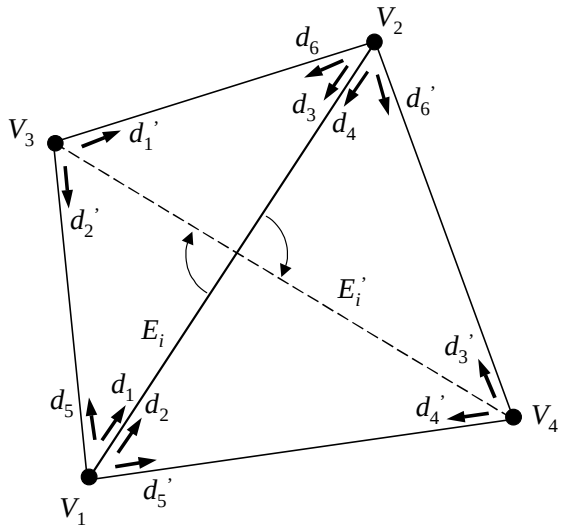

Figure 3.5: Swapping an edge in a triangulation

then be left to the application programmer. Accompanied by a boolean function `swappableEdge(Dart d)` that checks if the edge is a diagonal in a strictly convex quadrilateral, this would be sufficient for algorithms in the TTL based on edge-swapping. We will use this approach for the incremental Delaunay triangulation algorithm in the next section.

Modifiers that remove nodes, edges and triangles from a triangulation can also be handled the TTL, as well as modifiers that create and add topological elements. But there are many possible ways of defining proper syntax and semantics for the interplay between the TTL on the one side, and the application with the actual data structure on the other. Existing nodes, edges and triangles can be passed as dart objects when remove-operations are required on the application side, and function templates in the TTL can be parametrized on `NodeType`, `EdgeType`, and `TriangleType` when create-operations are required (though these topological elements need not be part of the actual data structure). Thus, the application programmer will control the content of these topological types, as well as creating and deleting them. A new interface channel between the TTL and the application would also be needed in addition to the dart class for creating and deleting nodes, edges and triangles, and to carry out basic operations based on these types that can be used by higher level operations in the TTL. This can, for example, be supported by a *traits* class in C++ which is passed as a template argument to the generic functions [19]. The traits class would contain type definitions and static functions like `createNode` and `deleteNode`, and likewise for edge and triangle. Another function would be `splitTriangle(DartType& d, NodeType& n)` for inserting a new node by

splitting a given triangle associated with a dart into three new triangles. The latter will be used by the generic Delaunay triangulation algorithm that follows.

4 Generic Delaunay Triangulation

Delaunay triangulations are the most frequently used constructions for making triangulations from a set of points in the plane. They have a nice equiangular property in the sense that they aim at avoiding long and thin triangles, which may cause numerical unstability in many algorithms. In this section we briefly summarize some of the classical theory of Delaunay triangulations and use these results directly to obtain a generic algorithm for their construction.

4.1 Theoretical Foundation

A Delaunay triangulation Δ of a set of points P in the plane maximizes the minimum interior angle of the triangles which have their nodes in P. Moreover, it is the triangulation of P that maximizes the lexicographical measure of an *indicator vector* $I(\Delta)$, sorted non-decreasing, where each entry α_i of the vector represents the smallest interior angle of each triangle in the triangulation,

$$I(\Delta) = (\alpha_1, \alpha_2, \ldots, \alpha_{|T|}),\ \alpha_i \leq \alpha_j,\ i < j,$$

where $|T|$ is the number of triangles in Δ. We say that a vector I is lexicographically larger than a vector I' if for some integer m, we have $\alpha_i = \alpha'_i$ for $i = 1, \ldots, m-1$ while $\alpha_m > \alpha'_m$. So, among all possible triangulations of a point set P a Delaunay triangulation is one that has the largest indicator vector measured lexicographically. It is assumed that the boundary edges of the triangulations form the convex hull[3] of P. The edges and triangles of a Delaunay triangulation are called *Delaunay edges* and *Delaunay triangles*.

An equivalent characterization is that the Delaunay triangulation of a point set P is the straight line dual of the Voronoi diagram of P; see for example Lawson [12] or Preparata & Shamos [20]. Yet another equivalent characterization of a Delaunay triangulation is that the circumcircle of each triangle does not contain any points from P in its interior.

Figure 4.1(a) shows two triangles of a triangulation that is not Delaunay since the point p_1 is interior to the circumcircle of one of the triangles. This is called the *circumcircle test* when applied to two triangles forming a quadrilateral as in the figure. It is easy to show that if the edge E_i is swapped to become a new edge E'_i with nodes p_1 and p_2, the circumcircle test would hold for the two triangles; see Figure 4.1(b). An edge is called *locally optimal* if the decision is *not* to swap it according to the circumcircle test. An edge is also called locally optimal if it cannot be swapped, that is, if the edge is at the boundary of the triangulation or if it is a diagonal in a quadrilateral that is not strictly convex.

An algorithmic approach to determine if and edge E_i is locally optimal is to examine the interior angles α and β opposite to E_i in the quadrilateral. If

[3] In this context the convex hull of P is the *boundary* of the smallest convex set of P.

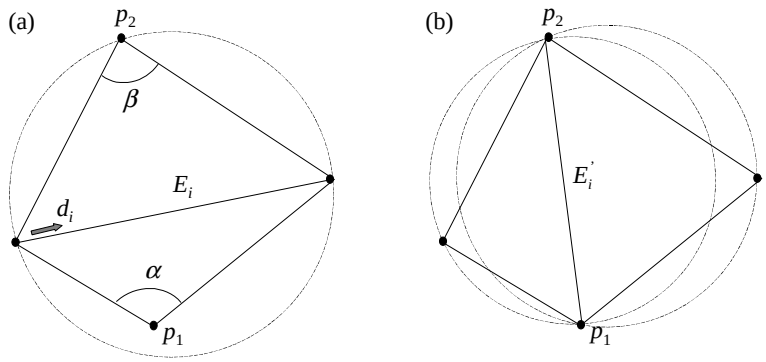

Figure 4.1: Circumcircle test. The edge E_i in (a) is swapped to E_i' in (b) and becomes locally optimal.

$\alpha + \beta > \pi$ then E_i is not locally optimal if it is a diagonal in a strictly convex quadrilateral. If the four points forming the quadrilateral lie on a common circle, then $\alpha + \beta = \pi$ and the choice of diagonal is arbitrary. This is called a *neutral case* for the circumcircle test.

The following theorem establishes the theoretical foundation for the Delaunay triangulation algorithm, which will be outlined in the sequel. The proof can be found in [12].

Theorem 4.1 *A triangulation Δ of a set of points P is a Delaunay triangulation if and only if all edges of Δ are locally optimal.*

It can be shown that when swapping the edge E_i in Figure 4.1, the minimum interior angle of the two triangles in the quadrilateral becomes larger. Thus, the lexicographical measure of the indicator vector of a triangulation also becomes larger each time an edge-swap occurs. Suppose now that edge-swapping is applied repeatedly to edges that are not locally optimal in an arbitrary triangulation. Since the number of possible triangulations of a finite point set is finite, this process converges to a final triangulation Δ^* whose edges are all locally optimal. Then, by Theorem 4.1, Δ^* must be a Delaunay triangulation. Moreover, if there are no neutral cases in a Delaunay triangulation of a point set P, one can also show that a Delaunay triangulation is *unique* and that its indicator vector is lexicographically maximum.

4.2 Incremental Delaunay Triangulation

The generic triangulation algorithm, which will be outlined below, is an *incremental* Delaunay triangulation algorithm [6, 13, 26]. These algorithms start with an initial triangulation and insert all points from a point set P one by one into the triangulation. After each point insertion the triangulation is updated

to be Delaunay, such that all triangles satisfy the circumcircle test. The initial triangulation can, for example, be two (large) triangles enclosing all points of P.

We will use the following scheme to insert a point p into an existing Delaunay triangulation Δ_N with N nodes to obtain a new Delaunay triangulation Δ_{N+1} with $N+1$ nodes (consult Figure 4.5):

1. Locate the triangle T_i in Δ_N that contains p.
2. Split T_i into three triangles by making three new edges between p and the nodes of T_i and thus obtain a new triangulation Δ'_{N+1}.
3. Apply a swapping procedure based on the circumcircle test to swap edges that are not locally optimal in Δ'_{N+1} until all edges are locally optimal. Then by Theorem 4.1 the final triangulation Δ_{N+1} is Delaunay.

Fortunately, the swapping procedure in Step 3 appears to be a local process such that only edges in a local neighbourhood around p need to be examined. The following results lead to a swapping procedure which produces the new Delaunay triangulation Δ_{N+1}. (See for example [6] or [8] for proofs).

Theorem 4.2 *Let Δ_{N+1} be a Delaunay triangulation obtained by inserting a point p into a Delaunay triangulation Δ_N. Then all new edges of Δ_{N+1} will have p as a common node.*

This implies that the region of Δ_N that needs to be modified by the insertion of p is connected and star-shaped as seen from p. Moreover, the following theorem state that the three new edges created in Step 2 are already Delaunay edges.

Theorem 4.3 *Assume p is inserted interior to a triangle T_i in a Delaunay triangulation Δ_N. Then the three edges obtained by connecting p to the nodes of T_i are Delaunay edges of Δ_{N+1}.*

Finally, following from Theorem 4.2 is a theorem which implies that the swapping procedure in Step 3 can be done in linear time.

Theorem 4.4 *An edge will never be swapped more than once as the result of applying the circumcircle test after insertion of a point into a Delaunay triangulation.*

We are now in a position to give the algorithmic details of Step 1, 2 and 3 above. Figure 4.2(b) shows the situation after point p has been inserted into an existing Delaunay triangulation Δ_N (Figure 4.2(a)) in Step 2 above, but before the proceeding step of swapping edges. Note first that the three new edges created in Step 2 are locally optimal since they are diagonals of non-convex quadrilaterals. Also, those three edges are Delaunay edges in Δ_{N+1} by Theorem 4.3, and it follows from the proceeding theorem that they will never

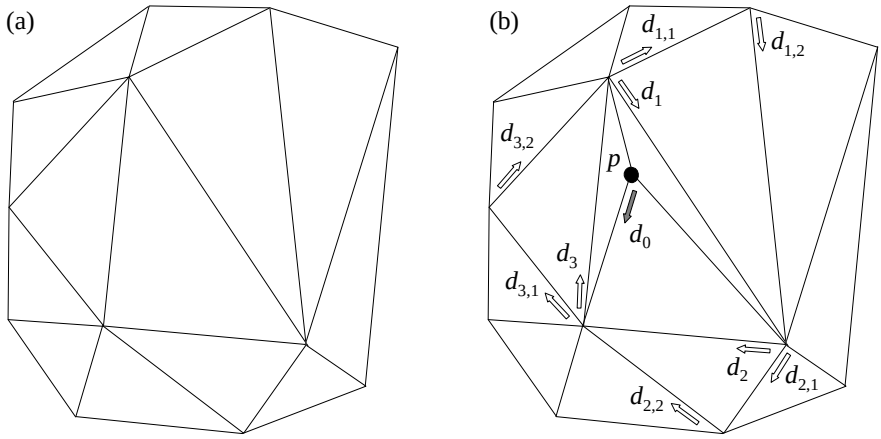

Figure 4.2: Illustration for Algorithm 4.2 when starting the swapping procedure.

be swapped. The only edges that are candidates for swapping when starting the swapping procedure, i.e., edges that might not be locally optimal, are those associated with the darts d_1, d_2 and d_3 in the figure. All the other edges in the triangulation are diagonals of the same quadrilaterals as they were before inserting p into Δ_N, and since Δ_N is a Delaunay triangulation those edges are locally optimal.

Assume that one of the edges associated with d_1, d_2 or d_3 in Figure 4.2(b), say E_i associated with d_i, is swapped as a result of the circumcircle test. Then the two edges opposite to d_i associated with $d_{i,1}$ and $d_{i,2}$ also become candidates for swapping since they become diagonals of new quadrilaterals. The other two edges of the new quadrilateral have already one of their nodes in the insertion point p, and it follows from the theorems above that they are Delaunay edges with respect to Δ_{N+1} and need not be examined again. Thus, when E_i is swapped it generates exactly two new candidate edges, $E_{i,1}$ and $E_{i,2}$ for swapping. This argument can be repeated for $E_{i,1}$ and $E_{i,2}$, and so forth, such that the swapping procedure fits into a binary tree structure where each swapped edge generates exactly two new candidates for swapping.

The pseudocodes that follow make use of the dart algebra of G-maps outlined in Section 2.1. Implementation in C++ is a straightforward exercise that can be done similarly to the examples in Section 3.1 and 3.2. It is assumed that a dart d given as input or output from the generic algorithms is always kept in a counterclockwise direction inside a triangle. Note that some details, for example handling fixed points at the boundary, are omitted. The first algorithm, which is given as input an insertion point p and an arbitrary dart d in the existing Delaunay triangulation Δ_N, implements Step 1, 2 and 3 above.

Algorithm 4.1 (`insertNodeAndSwapDelaunay(Point` p`, Dart` d`)`)

1. d_t = `ttl::locateTriangle(`p, d, $found$`)`
2. d_0 = `app::splitTriangle(`d_t`,` p`)`
3. $d_1 = \alpha_2 \circ \alpha_1 \circ \alpha_0 \circ \alpha_1 \circ \alpha_2 \circ \alpha_1(d_0)$
4. $d_2 = \alpha_2 \circ \alpha_1 \circ \alpha_0 \circ \alpha_1(d_0)$
5. $d_3 = \alpha_2 \circ \alpha_1 \circ \alpha_2 \circ \alpha_0(d_0)$
6. `ttl::recSwapDelaunay(`d_1`)`
7. `ttl::recSwapDelaunay(`d_2`)`
8. `ttl::recSwapDelaunay(`d_3`)`

The prefix "`ttl::`" at a function call indicates that the function is part of the generic TTL library, and "`app::`" indicates that the function must be present on the application side as an interface to the actual data structure; see Figure 3.1. In Step 1, Algorithm 3.1 in Section 3.2 is called to locate the triangle that contains p. The located triangle is represented by the dart d_t. Then the located triangle is split into three new triangles by `splitTriangle`, which also delivers a new dart d_0 oriented counterclockwise and located at the insertion point p; see Figure 4.2. The latter function is required in the interface to the application data structure and is not part of the TTL. Next, the darts d_1, d_2 and d_3 shown in Figure 4.2 are found by compositions of α-iterators. The binary tree structure of the swapping process explained above suggests a recursive scheme for a swapping procedure starting with each of d_1, d_2 and d_3. The function `recSwapDelaunay` takes the darts d_1, d_2 and d_3 as input, one at a time, and swaps recursively edges that are not locally optimal.

Algorithm 4.2 (`recSwapDelaunay(Dart` d_i`)`)

1. if `(ttl::circumcircleTest(`d_i`) == OK)`
2. RETURN
3. $d_{i,1} = \alpha_2 \circ \alpha_1(d_i)$
4. $d_{i,2} = \alpha_2 \circ \alpha_0 \circ \alpha_1 \circ \alpha_0(d_i)$
5. `app::swapEdge(`d_i`)`
6. `ttl::recSwapDelaunay(`$d_{i,1}$`)` // Call this procedure recursively
7. `ttl::recSwapDelaunay(`$d_{i,2}$`)` // Call this procedure recursively

In Step 1 the circumcircle test is applied to the current edge E_i associated with the dart d_i. As indicated by the prefix, it is assumed that the test is part of the TTL; see Algorithm 4.3 below. The recursion is stopped if E_i is locally optimal. Otherwise, candidate edges for swapping in the next recursion, represented by the darts $d_{i,1}$ and $d_{i,2}$ opposite to d_i, are found by compositions of α-iterators. After E_i is swapped in Step 5, the same procedure is called again recursively with $d_{i,1}$ and $d_{i,2}$. The result is that all edges of the new triangulation Δ_{N+1} are locally optimal and it follows from Theorem 4.1 that Δ_{N+1} is a Delaunay triangulation.

Figure 4.4 shows the whole swapping process of Algorithm 4.1 when inserting p. From (b) to the final triangulation in (e), each picture shows the triangulation after one new edge has been swapped. As can be seen in (e), the affected region is connected and all new edges radiate from the insertion point p.

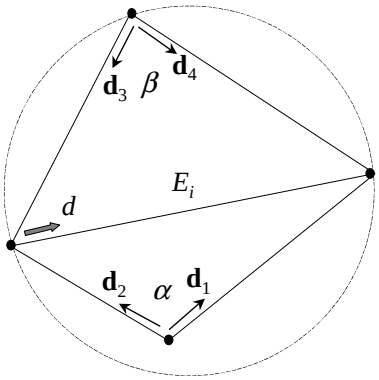

Figure 4.3: Circumcircle test and darts interpreted as vectors.

The circumcircle test in Algorithm 4.2 involves floating point arithmetic and could have been required on the application side to force the application programmer to control the level of accuracy of the numerical calculations. We include the algorithm in the TTL now to show the principle of how darts can be interpreted as vectors in generic algorithms.

Recall that the edge E_i should be swapped if $\alpha + \beta > \pi$. Since $\alpha + \beta < 2\pi$, this is equivalent to $\sin(\alpha + \beta) < 0$ which expands to

$$\sin\alpha\cos\beta + \cos\alpha\sin\beta < 0. \tag{4.1}$$

This also applies for quadrilaterals that are not convex such that testing for convexity is not necessary. Suppose that the darts $\mathbf{d}_i,\ i = 1, 2, 3, 4$ in Figure 4.3 are interpreted as unit vectors. For common notation we assume that the vectors are defined in 3D space such that the z-components are zero. Let $\mathbf{e}_3$ be the unit vector $(0, 0, 1)$. Then, sine and cosine of α and β can be computed by cross products and scalar products,

$$\begin{aligned}
\sin\alpha &= (\mathbf{d}_1 \times \mathbf{d}_2) \cdot \mathbf{e}_3 \\
\sin\beta &= (\mathbf{d}_3 \times \mathbf{d}_4) \cdot \mathbf{e}_3 \\
\cos\alpha &= \mathbf{d}_1 \cdot \mathbf{d}_2 \\
\cos\beta &= \mathbf{d}_3 \cdot \mathbf{d}_4.
\end{aligned}$$

Assume that cross product and scalar product between vectors, represented as darts, are present on the application side. Let a function `crossProduct(`$\mathbf{d}_i, \mathbf{d}_j$`)` deliver the scalar $(\mathbf{d}_i \times \mathbf{d}_j) \cdot \mathbf{e}_3$, that is, the sine of the angle between $\mathbf{d}_i$ and $\mathbf{d}_j$ in accordance with the equations above. The circumcircle test takes a dart d representing a diagonal edge in a quadrilateral as input:

Algorithm 4.3 (`bool circumcircleTest(`d`)`)

1. $\mathbf{d}_1 = \alpha_1 \circ \alpha_0 \circ \alpha_1 \circ \alpha_2(d)$
2. $\mathbf{d}_2 = \alpha_0 \circ \alpha_1 \circ \alpha_2(d)$
3. $\mathbf{d}_3 = \alpha_0 \circ \alpha_1(d)$
4. $\mathbf{d}_4 = \alpha_1 \circ \alpha_0 \circ \alpha_1(d)$
5. $\sin\alpha =$`app::crossProduct(`$\mathbf{d}_1, \mathbf{d}_2$`)`
6. $\sin\beta =$`app::crossProduct(`$\mathbf{d}_3, \mathbf{d}_4$`)`
7. $\cos\alpha =$`app::scalarProduct(`$\mathbf{d}_1, \mathbf{d}_2$`)`
8. $\cos\beta =$`app::scalarProduct(`$\mathbf{d}_3, \mathbf{d}_4$`)`
9. if $(\sin\alpha\cos\beta + \cos\alpha\sin\beta) < 0$
10. return OK
11. else
12. return FALSE

Great care should be taken to ensure numerical stability of the swap test. When there are almost neutral cases, or when points of a quadrilateral are almost coincident or almost collinear, round-off errors may lead to incorrect results. For example, cycling may occur and lead to infinite loops in the algorithms. The test can be improved by using multiple and more robust tests [3], or using exact or "almost exact" arithmetic [22]. Thus, there are some motives for directing the circumcircle test and other functionality involving numerical calculations to the application programmer. A solution may be to direct the tests to the application, but also implement them in the TTL. Then the application programmer can choose whether to implement the functions at desired level of accuracy or call the functions present in the TTL.

4.3 Fixing the Boundary

The algorithms outlined above comprise the necessary functionality for an incremental Delaunay triangulation algorithm. Starting with an initial triangulation created on the application side, Algorithm 4.1 is called repeatedly for all points in a point set P. The initial triangulation can, for example, be two triangles forming a rectangle enclosing all points of P as in Figure 4.5(a). Figure 4.5(b) shows an example of a Delaunay triangulation $\Delta(P \cup B)$ after all points of P have been inserted, where B denotes the four points at the boundary.

In most cases one wants the final result to be the Delaunay triangulation $\Delta(P)$ without nodes at the boundary of the initial triangulation. This implies that some of the edges near the boundary must be swapped such that the convex hull of P, $conv(P)$, is formed by triangle edges in $\Delta(P)$. In addition, triangles with one or two nodes in B must be removed. The triangulation $\Delta(P \cup B)$ in Figure 4.5(b) has three regions with different topological cases where edges must be swapped to form $conv(P)$. The edge E_1 at the left boundary can be swapped immediately to become a line segment of $Conv(P)$. At the right boundary both E_1 and E_2 must be swapped (in any order) such that one of them is contained in $conv(P)$. These are the only topological cases that can occur at the boundary where $conv(P)$ is not formed by triangle edges in $\Delta(P \cup B)$, but they can be "nested" as shown at the lower boundary of $\Delta(P \cup B)$ in Figure 4.5(b). The different topological cases can be identified by a generic algorithm in the TTL based on dart algebra and requires no additional functionality on the application side than required by the algorithms above. A recursive swapping procedure must be called each time an edge is swapped. The swapping procedure is similar to that of Algorithm 4.2, but is somewhat more involved. Swapping an edge now generates *four* new candidate edges for swapping, as opposed to two only in Algorithm 4.2. In addition, some tests are required to avoid infinite loops by swapping edges to and from the boundary.

Figure 4.5(c) shows the result after the swapping procedure has been run, and in (d) the final result is shown after triangles at the boundary have been removed. The latter procedure requires a function `removeBoundaryTriangle(`d`)` on the application side, which removes a triangle T_k associated with a dart $d = (V_i, E_j, T_k)$ at the boundary of the triangulation.

4.4 Some Remarks

It can be shown that incremental Delaunay triangulation is of order $O(N^2)$ in the worst case, where N is the number of input points. This is contrary to $O(N \log N)$ which is the theoretical optimal running time for Delaunay triangulation which can be achieved with divide-and-conquer algorithms [6, 13]. But $O(N^2)$ for the incremental approach only occurs for point configurations that are very rare, and in practical applications, average performance of $O(N \log N)$ or even better is achieved.

For optimal performance the points in P can be sorted in lexicographically ascending order such that $p_i = (x_i, y_i) < (x_j, y_j) = p_j$ if and only if $x_i < x_j$,

or $x_i = x_j$ and $y_i < y_j$. Then the next point to be inserted will be close to the previous one. By returning a dart from Algorithm 4.1 close to the insertion point, this dart can be used as input when the algorithm is called again for the next insertion point. Thus, the average performance of `locateTriangle` in Step 1 would be much faster than $O(N)$, which is the time complexity in general.

In the actual implementation in C++ it was adopted as a general rule that a dart outside a quadrilateral should not be changed as the result of swapping the diagonal of the quadrilateral, while darts inside the quadrilateral could be changed. This was necessary for some of the data structures that were adapted to the generic Delaunay triangulation algorithm during testing. Figure 4.2(b) is in agreement with this rule as the darts $d_{1,1}$ and $d_{1,2}$ are "hidden" outside the quadrilateral where the edge associated with d_1 is swapped.

5 Conclusion

Generic algorithms for triangulations can be founded on sound algebraic concepts as defined by generalized maps, or G-maps. G-maps provide an algebra with all the necessary functions for navigating in the topology of a triangulation at an abstract level independent of a specific underlying data structure. The few and clear basic concepts that G-maps are based on are intuitive and easy to deal with for application programmers. Implementation by means of function templates in C++ separates algorithms from data structures such that applications can adapt arbitrary data structures for triangulations to a generic library, which we have called TTL. Among many useful tools that have been implemented in the TTL is an incremental Dealaunay triangulation algorithm. Algorithms in the TTL are compact and easy to read, and thus, easy to maintain and extend with new functionality. Interfaces between the TTL and application data structures are clean and narrow with only a few parameters in the argument lists. In fact, many algorithms in the TTL pass only one single object in their interfaces: a dart, which represents a (node, edge, triangle)-triple.

The solutions we suggest run without significant loss of efficiency compared to algorithms that work directly on a specific data structure. In particular, the basic topological traversal operators, α_0, α_1 and α_2 of G-maps that dominate the running time of many algorithms, can be implemented as C++ inline functions in a dart class. Compared to abstraction through class inheritance and dynamic binding the algorithms in the TTL are much more efficient.

Acknowledgments *This work was supported by the Research Council of Norway under the research program 117644/223, "DYNAMAP II, Management and Use of Geodata". The author thanks Yvon Halbwachs, Rune Aasgaard and Thomas Engh Sevaldrud for fruitful discussions on generic programming in C++ and for providing many useful hints when implementing the TTL.*

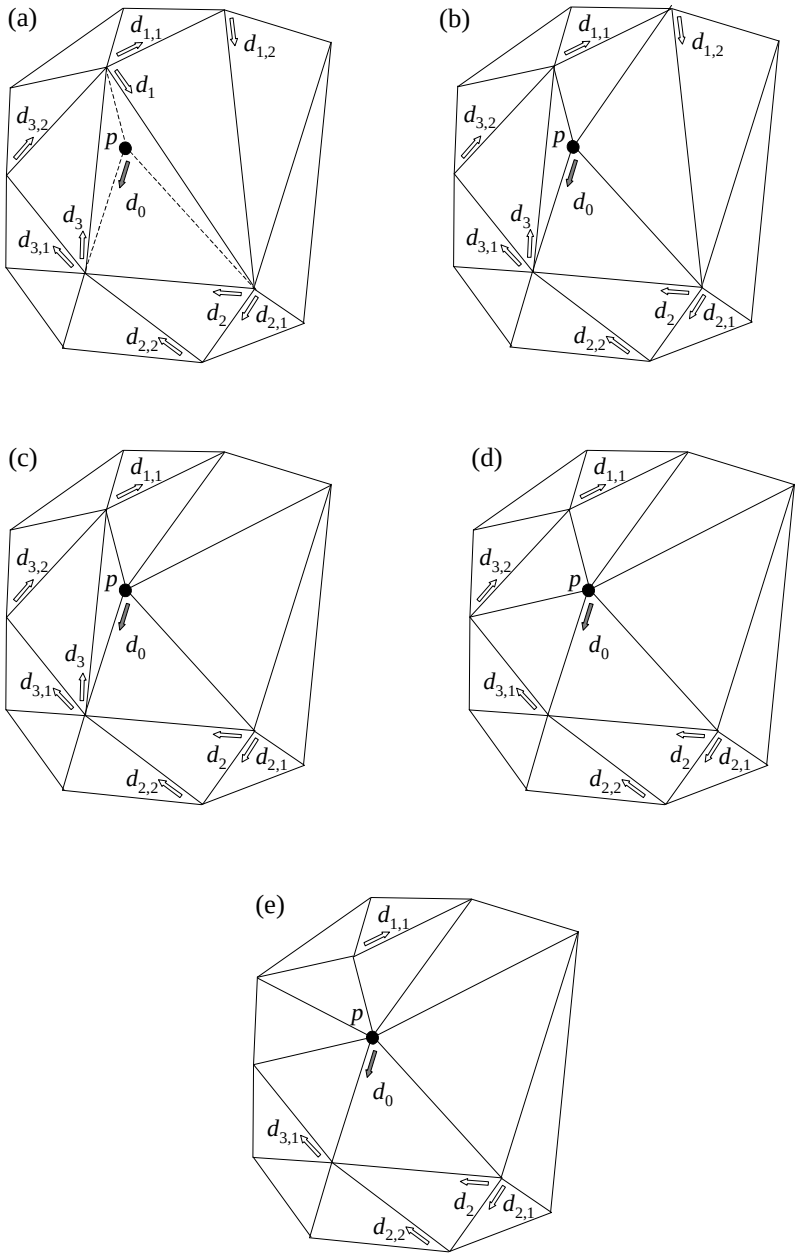

Figure 4.4: Swapping procedure when inserting a point p into a Delaunay triangulation. From (b) to the final triangulation in (e), each picture shows the triangulation after one new edge has been swapped.

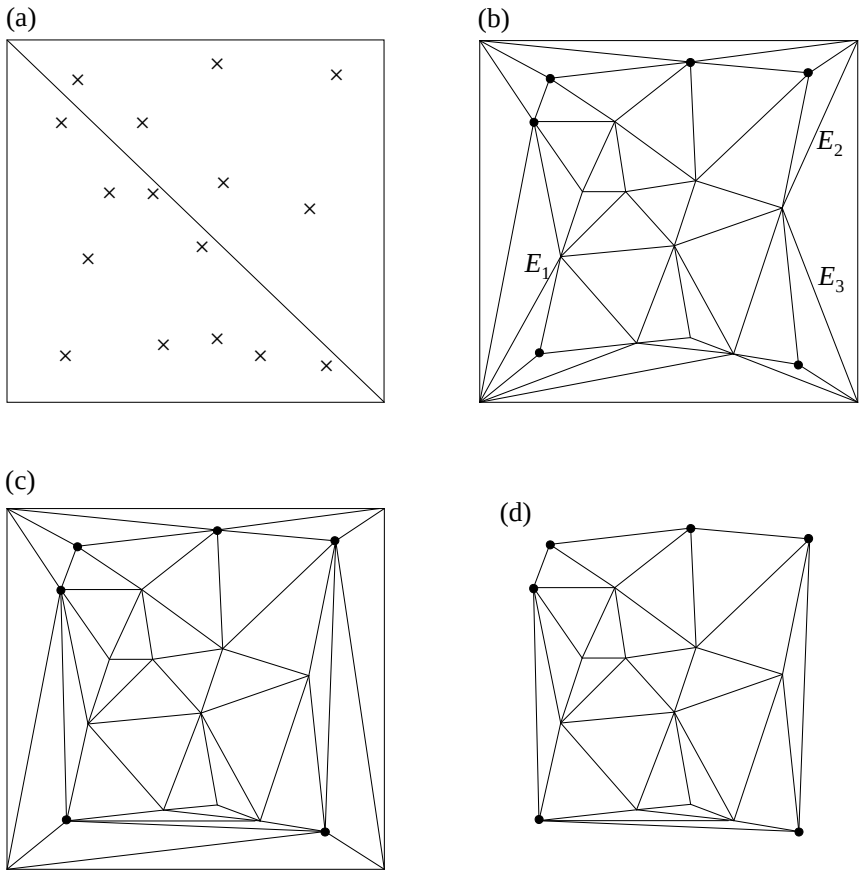

Figure 4.5: Illustration of the incremental Delaunay triangulation algorithm. (a): two triangles enclosing the point set P; (b): all points in P are inserted; (c): forming the convex hull of P by swapping edges away from the boundary; (d): the final Delaunay triangulation. The bullets indicate vertices of the convex hull of P.

References

[1] E. Arge. Siscat triangulation, ideas for new design. Technical report, Numerical Objects AS, 1997.

[2] Y. Bertrand and J.-F. Dufourd. Algebraic specification of a 3d-modeller based on hypermaps. *Graphical Models and Image Processing*, 56(1):29–60, January 1994.

[3] A. K. Cline and R. J. Renka. A storage-efficient method for construction of a Thiessen triangulation. *Rocky Mountain J. Math.*, 14:119–140, 1984.

[4] M. Dæhlen and M. Fimland. Constructing hierarchical terrain models by edge ordering over triangulations. In *Proceeding from the 7th Scandinavian Research Conference on Geographical Information Science (ScanGIS'99)*, pages 25–38, Aalborg, Denmark, June 1999.

[5] P. L. George and H. Borouchaki. *Delaunay Triangulation and Meshing: Application to Finite Elements*. Hermes, Paris, 1998.

[6] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transaction on Graphics*, 4(2):74–123, 1985.

[7] Y. Halbwachs and Ø. Hjelle. Generalized maps in geological modeling: Object-oriented design of topological kernels. In H. P. Langtangen, A. M. Bruaset, and E. Quak, editors, *Advances in Software Tools for Scientific Computing*, pages 339–356. Springer Verlag, December 1999.

[8] Ø. Hjelle. Triangulations and triangle-based surfaces. Lecture notes, University of Oslo, 2000. (In preparation).

[9] H. Hoppe. Smooth view-dependent level-of-detail control and its applications to terrain rendering. In *IEEE Visualization'98*, pages 35–42, 1998.

[10] L. Kettner. Designing a data structure for polyhedral surfaces. In *The 14th ACM Symp. On Computational Geometry*, pages 146–154, Minneapolis, Minnesota, June 1998.

[11] C. L. Lawson. Transforming triangulations. *Discrete Mathematics*, 3:365–372, 1972.

[12] C. L. Lawson. Software for C^1 surface interpolation. In J. Rice, editor, *Mathematical Software III*, New York, 1977. Academic Press.

[13] D. T. Lee and B. J. Schachter. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer and Information Sciences*, 9(3):219–242, 1980.

[14] P. Lienhardt. Subdivision of n-dimensional spaces and n-dimensional generalized maps. In *5th ACM Symposium on Computational Geometry*, pages 228–236, Saarbrucken, Germany, 1989.

[15] P. Lienhardt. Topological models for boundary representation: A survey. Technical report, University of Luis Pasteur, Strasbourg, February 1990.

[16] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *ACM SIGGRAPH 96*, pages 109–118, August 1996.

[17] J. A. Mchugh. *Algorithmic Graph Theory*. Prentice-Hall Inc., 1990.

[18] S. Meyers. *Effective C++, 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Reading, MA, 1992.

[19] N. Myers. A new and useful template technique: Traits. *C++ Report*, 7(5):32–35, June 1995.

[20] F. P. Preparata and M. I. Shamos. *Computational Geometry, an Introduction*. Springer-Verlag, New York, 1985.

[21] T. E. Sevaldrud. Hierarchical terrain models with applications in flight simulation. Master's thesis, University of Oslo, Department of Informatics, May 1999.

[22] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[23] J. R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1997. Available as Technical Report CMU-CS-97-137.

[24] B. L. Stephane Contreaux and J.-L. Mallet. A celluar topological model based on generalized maps: The GOCAD approach. In *GOCAD ENSG Conference. 3D Modelling of Natural Objects: A Challenge for the 2000's*, Nancy, June 1998.

[25] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, third edition, 1997.

[26] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[27] K. Weiler. Edge based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, January 1985.