Triangulation

Separate Build

September 26, 2012
## Contents

1 Triangulations 1

1.1 Introduction .................................................. 1
1.2 Triangulation Data Structure ................................. 3
1.3 Triangulations ................................................... 10
1.4 Delaunay Triangulations ..................................... 13
1.5 Complexity and Performances ............................... 15
1.6 Design and Implementation History .......................... 15
   Reference Manual .................................................. 17
1.7 Reference Pages Sorted by Type .............................. 17
1.8 Alphabetical List of Reference Pages ....................... 18

Index 77
Chapter 1

Triangulations

Samuel Hornus and Olivier Devillers

Contents

1.1 Introduction ................................................................. 1
1.2 Triangulation Data Structure ............................................ 3
  1.2.1 The class Triangulation_data_structure .......................... 3
  1.2.2 Examples ................................................................. 5
1.3 Triangulations ............................................................... 10
  1.3.1 Implementation ......................................................... 10
  1.3.2 Examples ................................................................. 10
1.4 Delaunay Triangulations .................................................. 13
  1.4.1 Implementation ......................................................... 13
  1.4.2 Examples ................................................................. 13
1.5 Complexity and Performances .......................................... 15
1.6 Design and Implementation History ................................... 15
  Reference Manual ............................................................ 17
1.7 Reference Pages Sorted by Type ....................................... 17
  1.7.1 Concepts ................................................................. 17
  1.7.2 Classes ................................................................. 18
  1.7.3 Enums ................................................................. 18
1.8 Alphabatical List of Reference Pages .................................. 18

This package proposes data structure and algorithms to compute triangulations of points in any dimensions. The Triangulation_data_structure allows to store and manipulate the combinatorial part of a triangulation while the geometric classes Triangulation and Delaunay_triangulation allows to compute a (Delaunay) triangulation of a set of points and to maintain it under insertions (and deletions in the Delaunay case).

1.1 Introduction

Some definitions

A finite abstract simplicial complex is built on a finite set of vertices \( V \) and consists of a collection \( S \) of subsets of \( V \) such that
if $s$ is a set of vertices in $S$, then all the subsets of $s$ are also in $S$.

The sets in $S$ (which are subsets of $V$) are called faces or simplices (the singular of which is simplex). A simplex $s \in S$ is maximal if it is not a proper subset of some other set in $S$. The simplicial complex is pure if all the maximal simplices have the same cardinality, i.e., they have the same number of vertices. In the sequel, we will call these maximal simplices full cells. A face of a simplex is a subset of it. A proper face of a simplex is a strict subset of it.

If the vertices are embedded into Euclidean space $\mathbb{R}^d$, we deal with finite simplicial complexes which have slightly different simplices and additional requirements:

- vertices corresponds to points in space.
- a simplex $s \in S$ is the convex hull of its vertices.
- the vertices of a simplex $s \in S$ are affinely independent.
- the intersection of any two simplices of $S$ is a proper face of both simplices (the empty set counts).

See the wikipedia entry for more about simplicial complexes.

What’s in this package?

This CGAL package deals with pure finite simplicial complexes without boundary, which we will simply call in the sequel triangulations. It provides three main classes for creating and manipulating triangulations.

The class `CGAL::Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell>` models an abstract triangulation: vertices in this class are not embedded in Euclidean space but are only of combinatorial nature.

The class `CGAL::Triangulation<TriangulationTraits, TriangulationDataStructure>` embeds an abstract triangulation in Euclidean space, thus forming a geometric triangulation. Methods are provided for the insertion of points in the triangulation, the traversal of various elements of the triangulation, as well as the localization of a query point inside the triangulation. The convex hull of the points is part of the triangulation, the fact that there is no boundary is ensured by adding an infinite vertex and infinite full cells to triangulate the outside of the convex hull.

The class `CGAL::Delaunay_triangulation<DelaunayTriangulationTraits, TriangulationDataStructure>` adds further constraints to a triangulation, in that all its simplices must have the so-called Delaunay or empty-ball property: the interior of a ball circumscribing any simplex (or full cell) must be free from any vertex of the triangulation. The `CGAL::Delaunay_triangulation` class supports deletion of vertices.

Further definitions

An $i$-face denotes an $i$-dimensional simplex, or a simplex with $i + 1$ vertices. When these vertices are embedded in Euclidean space, they must be affinely independent.

If the maximal dimension of a simplex in the triangulation is $d$, we call:

- an $i$-face for some $i \in [0, d]$ a face;  
- a 0-face a vertex;
• a 1-face an edge;
• a \((d - 2)\)-face a ridge;
• a \((d - 1)\)-face a facet; and
• a \(d\)-face a full cell.

Two faces \(\sigma\) and \(\sigma'\) are incident if and only if \(\sigma'\) is a proper sub-face of \(\sigma\) or vice versa.

1.2 Triangulation Data Structure

In this section, we describe the concept TriangulationDataStructure for which CGAL provides one model class:

\[ \text{CGAL::Triangulation}\_\text{data}\_\text{structure\<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell\>}. \]

A TriangulationDataStructure can represent an abstract pure complex such that any facet is incident to exactly two full cells.

A TriangulationDataStructure has a property called the maximal dimension which is a positive integer equal to the maximum dimension a full cell can have. This maximal dimension can be chosen by the user at the creation of a TriangulationDataStructure and can then be queried using the method \(\text{tds.maximal}\_\text{dimension()}\). A TriangulationDataStructure also knows the current dimension of its full cells, which can be queried with \(\text{tds.current}\_\text{dimension()}\). In the sequel, let us denote the maximal dimension with \(D\) and the current dimension with \(d\). The inequalities \(-2 \leq d \leq D\) and \(0 < D\) always hold. The special meaning of negative values for \(d\) is explained below.

The data structure triangulates \(S^d\)

A TriangulationDataStructure can be viewed as a triangulation of the topological sphere \(S^d\), i.e., its faces can be embedded to form a partition of \(S^d\) into \(d\)-simplices.

One nice consequence of the above important fact is that a full cell has always exactly \(d + 1\) neighbors. Two full cells \(\sigma\) and \(\sigma'\) sharing a facet are called neighbors.

Possible values of \(d\) (the current dimension of the triangulation) include

- \(d = -2\) This corresponds to the non-existence of any object in the TriangulationDataStructure.
- \(d = -1\) This corresponds to a single vertex and a single full cell. In a geometric triangulation, this vertex corresponds to the vertex at infinity.
- \(d = 0\) This corresponds to two vertices (geometrically, the finite vertex and the infinite vertex), each corresponding to a full cell; the two full cells being neighbors of each other. This is the unique triangulation of the 0-sphere.
- \(0 < d \leq D\) This corresponds to a standard triangulation of the sphere \(S^d\).

1.2.1 The class Triangulation_data_structure

We give here some details about the class Triangulation\_data\_structure\<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell\> implementing the concept TriangulationDataStructure.
**Storage**

A *TriangulationDataStructure* explicitly stores its vertices and full cells.

Each vertex stores a reference (a *handle*) to one of its incident full cells.

Each full cell stores references to its $d + 1$ vertices and neighbors. Its vertices and neighbors are indexed from 0 to $d$. The indices of its neighbors have the following meaning: the $i$-th neighbor of $\sigma$ is the unique neighbor of $\sigma$ that does not contain the $i$-th vertex of $\sigma$; in other words, it is the neighbor of $\sigma$ opposite to the $i$-th vertex of $\sigma$ (Figure 1.1).

![Indexing the vertices and neighbors of a full cell](image)

**Figure 1.1:** Indexing the vertices and neighbors of a full cell $c$ in dimension $d = 2$.

The index of a full cell $c$ in the $i$-th neighbor of $c$ is called the $i$-th mirror-index of $c$ (Figure 1.1). Mirror indices are often needed for maintaining the triangulation data structure. Thus, it might be desirable, for performance reasons, to store the mirror indices alongside the references to the vertices and neighbors in a full cell. This improves speed a little, but requires more memory.

CGAL provides the class template *Triangulation_ds_full_cell* for representing full cells in a triangulation. Its second template parameter is used to specify whether or not the mirror indices should be kept in memory or computed on-the-fly, which is the default case. Please refer to the documentation of that class template for specific details.

**Instantiating the class template**

The *Triangulation_data_structure* class template is designed in such a way that its user can choose

- the maximal dimension of the triangulation data structure by specifying the *Dimensionality* template parameter,
- the type used to represent vertices by specifying the *TriangulationDSVertex* template parameter and
- the type used to represent full cells by specifying the *TriangulationDSFullCell* template parameter.

The last two parameters have default values and are thus not necessary, unless the user needs custom types (see the reference manual page for this class template). The first template parameter, *Dimensionality*, must be one of the following:
- CGAL::Dimension_tag\(<D>\) for some integer \(D\). This indicates that the triangulation can store full cells of dimension at most \(D\). The maximum dimension \(D\) is known by the compiler, which triggers some optimizations.

- CGAL::Dynamic_dimension_tag. In this case, the maximum dimension of the full cells must be passed as an integer argument to an instance constructor (see TriangulationDataStructure).

The TriangulationDSVertex and TriangulationDSFullCell parameters to the class template must be models of the concepts TriangulationDSVertex and TriangulationDSFullCell respectively. CGAL provides models for these concepts: Triangulation_ds_vertex<TriangulationDataStructure> and Triangulation_ds_full_cell<TriangulationDataStructure, TDSFullCellStoragePolicy>, which, as one can see, take the TriangulationDataStructure as a template parameter in order to get access to some nested types in TriangulationDataStructure.

This creates a circular dependency, which we resolve in the same way as in the CGAL Triangulation_2 and Triangulation_3 packages (see Chapters ??, ??, ??, and ??). In particular, models of the concepts TriangulationDSVertex and TriangulationDSFullCell must provide a nested template Rebind_TDS which is documented in those two concept’s reference manual pages.

The user that is in need of a custom vertex or full cell class, is encouraged to read the documentation of the CGAL Triangulation_2 or Triangulation_3 package.

### 1.2.2 Examples

#### Incremental Construction

The following examples shows how to construct a triangulation data structure by inserting vertices. Its main interest is that it demonstrates most of the API to insert new vertices into the triangulation. Therefore, the reader will make the best use of this example by reading it slowly, together with the reference manual documentation of the methods that are called (see here: TriangulationDataStructure) and by trying to understand the various assert(...) statements.

```cpp
#include <CGAL/Triangulation_data_structure.h>
#include <iostream>
#include <vector>

int main()
{
    typedef CGAL::Triangulation_data_structure<CGAL::Dimension_tag<7> > TDS;

    TDS S;
    assert( 7 == S.maximal_dimension() );
    assert( -2 == S.current_dimension() );
    assert( S.is_valid() );

    std::vector<TDS::Vertex_handle> V(10);
    V[0] = S.insert_increase_dimension(); //insert first vertex
    assert( -1 == S.current_dimension() );
```
for( int i = 1; i <= 5; ++i )
    V[i] = S.insert_increase_dimension(V[0]);
// the first 6 vertices have created a triangulation
// of the 4-dimensional sphere
assert( 4 == S.current_dimension() );
assert( 6 == S.number_of_vertices() );
assert( 6 == S.number_of_full_cells() );

TDS::Full_cell_handle c = V[5]->full_cell();
V[6] = S.insert_in_full_cell(c);
// full cell c is split in 5
assert( 7 == S.number_of_vertices() );
assert( 10 == S.number_of_full_cells() );

c = V[3]->full_cell();
TDS::Facet ft(c, 2); // the Facet opposite to vertex 2 in c
V[7] = S.insert_in_facet(ft);
// facet ft is split in 4 and the two incident cells are split accordingly
assert( 8 == S.number_of_vertices() );
assert( 16 == S.number_of_full_cells() );

c = V[3]->full_cell();
TDS::Face face(c);
// meant to contain the edge joining vertices 2 and 4 of full_cell c
face.set_index(0, 2); // namely vertex 2
face.set_index(1, 4); // and vertex 4
V[8] = S.insert_in_face(face);
// face is split in 2, and all incident full cells also
assert( S.is_valid() );

TDS::Full_cell_handle hole[2];
hole[0] = V[8]->full_cell();
hole[1] = hole[0]->neighbor(0);
// the hole is made of two adjacent full cells
ft = TDS::Facet(hole[0], 1); // a face on the boundary of hole[0]
V[9] = S.insert_in_hole(hole[0], hole[0]+2, ft);
// the hole is triangulated by linking a new vertex to its boundary
assert( S.is_valid() );
return 0;
}

File: examples/Triangulation/triangulation_data_structure_static.cpp

In previous example, the maximal dimension is fixed at compile time. It is also possible to fix it at run time, as in the next example.

#include <CGAL/Triangulation_data_structure.h>
#include <iostream>
#include <vector>

int main()
{

const int ddim = 5; // dimension of TDS with dynamic dimension
typedef CGAL::Triangulation_data_structure<CGAL::Dynamic_dimension_tag>
TDS;
typedef TDS::Vertex_handle Vertex_handle;
TDS D(ddim); // the argument is taken into account.

assert( ddim == D.maximal_dimension() );
assert( -2 == D.current_dimension() );
assert( D.is_valid() );
std::vector<Vertex_handle> V(5);
V[0] = D.insert_increase_dimension();
V[1] = D.insert_increase_dimension(V[0]);
V[2] = D.insert_increase_dimension(V[0]);
V[3] = D.insert_increase_dimension(V[0]);
assert( 6 == D.number_of_full_cells() );
assert( 2 == D.current_dimension() );
assert( D.is_valid() );
return 0;
}

File: examples/Triangulation/triangulation_data_structure_dynamic.cpp

Barycentric subdivision

This example provides a function for computing the barycentric subdivision of a single full cell \( c \) in a triangulation data structure. The other full cells adjacent to \( c \) are automatically subdivided to match the subdivision of the full cell \( c \). The barycentric subdivision of \( c \) is obtained by enumerating all the faces of \( c \) in order of decreasing dimension, from the dimension of \( c \) to dimension 1, and inserting a new vertex in each face. For the enumeration, we use a combination enumerator, which is not documented, but provided in CGAL.

![Figure 1.2: Barycentric subdivision in dimension \( d = 2 \).](image)

```cpp
#include <CGAL/Triangulation_data_structure.h>
#include <CGAL/internal/Combination_enumerator.h>
#include <iostream>
#include <vector>

template<typename TDS>
void find_face_from_vertices(const TDS & tds, const std::vector<typename TDS::Vertex_handle> & face_vertices, typename TDS::Face & face);
```
template< typename TDS >
void barycentric_subdivide(TDS & tds, typename TDS::Full_cell_handle fc)
{ /* This function builds the barycentric subdivision of a single
    full cell |fc| from a triangulation data structure |tds|. */
typedef typename TDS::Full_cell_handle Full_cell_handle;
typedef typename TDS::Vertex_handle Vertex_handle;
typedef typename TDS::Face Face;
const int dim = tds.current_dimension();

    // First, read handles to the cell’s vertices
std::vector<Vertex_handle> vertices;
std::vector<Vertex_handle> face_vertices;
for( int i = 0; i <= dim; ++i ) vertices.push_back(fc->vertex(i));

    // Then, subdivide the cell |fc| once by inserting one vertex
tds.insert_in_full_cell(fc);
    // From now on, we can’t use the variable |fc|...

    // Then, subdivide faces of |fc| in order of decreasing dimension
for( int d = dim-1; d > 0; --d )
{
    face_vertices.resize(d+1);
    // The following class
    // enumerates all (d+1)-tuple of the set {0, 1, ..., dim}
    CGAL::internal::Combination_enumerator combi(d+1, 0, dim);
    while( ! combi.end() )
    {
        for( int i = 0; i <= d; ++i )
            face_vertices[i] = vertices[combi[i]];
        // we need to find a face with face_vertices
        Face face(dim);
        find_face_from_vertices(tds, face_vertices, face);
        tds.insert_in_face(face);
        ++combi;
    }
}

template< typename TDS >
void find_face_from_vertices( const TDS & tds,
const std::vector<typename TDS::Vertex_handle> & face_vertices,
typename TDS::Face & face)
{ /* The main goal of this function is to find a full cell that
    contains a given set of vertices |face_vertices|. Then, it
    builds a corresponding |face|. */
typedef typename TDS::Face Face;
typedef typename TDS::Vertex_handle Vertex_handle;
typedef typename TDS::Full_cell_handle Full_cell_handle;
typedef typename TDS::Full_cell::Vertex_handle_iterator Vertex_h_iterator;

    // get the dimension of the face we want to build
int fdim(face_vertices.size() - 1);
```cpp
if( fdim <= 0) exit(-1);

// find all full cells incident to the first vertex of |face|
typedef std::vector<Full_cell_handle> Cells;
Cells cells;
std::back_insert_iterator<Cells> out(cells);
tds.incident_full_cells(face_vertices[0], out);

// Iterate over the cells to find one which contains the face_vertices
for( typename Cells::iterator cit = cells.begin(); cit != cells.end(); ++cit){
    // find if the cell *cit contains the Face |face|
    int i(0);
    for( ; i <= fdim; ++i ) {
        Vertex_handle face_v(face_vertices[i]);
        bool found(false);
        Vertex_h_iterator vit = (*cit)->vertices_begin();
        for( ; vit != (*cit)->vertices_end(); ++vit ) {
            if( *vit == face_v ) {
                found = true;
                break;
            }
        }
        if( ! found )
            break;
    }
    if( i > fdim )
        // the full cell *cit contains |face|
        face.set_full_cell(*cit);
        for( int i = 0; i <= fdim; ++i )
            face.set_index(i, (*cit)->index(face_vertices[i]));
        return;
}
std::cerr << "Could not build a face from vertices"<<std::endl;
assert(false);
}

int main()
{
    const int sdim = 5; // dimension of TDS with compile-time dimension
typedef CGAL::Triangulation_data_structure<CGAL::Dimension_tag<sdim>> TDS;
    TDS tds(sdim);
    TDS::Vertex_handle one_vertex = tds.insert_increase_dimension();
    for( int i = 1; i < sdim+2; ++i )
        tds.insert_increase_dimension(one_vertex);
    // we get a triangulation of space of dim sdim homeomorphic to
    // a triangulation of the sphere in dim sdim+1 by a simplex
    assert( sdim == tds.current_dimension() );
    assert( 2+sdim == tds.number_of_vertices() );
    assert( 2+sdim == tds.number_of_full_cells() );
```
barycentric_subdivide(tds, tds.full_cells_begin());

// The number of full cells should be the twice the factorial of
// |tds.current_dimension()+1|. Eg, 1440 for dimension 5.
std::cout << "Triangulation has "
<< tds.number_of_full_cells() << " full cells";
assert( tds.is_valid() );
std::cout << " and is valid!"<<std::endl;
return 0;
}

File: examples/Triangulation/barycentric_subdivision.cpp

1.3 Triangulations

The class `CGAL::Triangulation<TriangulationTraits, TriangulationDataStructure>` embeds an abstract triangulation into Euclidean space. More precisely, it maintains a triangulation (a partition into pairwise interior-disjoint full cells) of the convex hull of the points (the embedded vertices) of the triangulation, as well as a triangulation of the complement of the convex hull in the affine subspace spanned by the triangulation’s points using a special vertex at infinity.

Methods are provided for the insertion of points in the triangulation, the contraction of faces, the traversal of various elements of the triangulation as well as the localization of a query point inside the triangulation.

Infinite full cells outside the convex hull are each incident to a finite facet on the convex hull of the triangulation and to a unique vertex at infinity.

As long as no advanced class method is called, it is guaranteed that all finite full cells have positive orientation. The infinite full cells are oriented so that the finite vertices of the triangulation lies on the negative side of the oriented hyperplane defined by the full cell’s finite facet.

1.3.1 Implementation

The class `CGAL::Triangulation<TriangulationTraits, TriangulationDataStructure>` stores a model of the concept `TriangulationDataStructure` which is instantiated with a vertex type that stores a point, and a full cell type that allows the retrieval of the point of its vertices.

The template parameter `TriangulationTraits` must be a model of the concept `TriangulationTraits` which provides the geometric `Point` type as well as various geometric predicates used by the `Triangulation` class.

1.3.2 Examples

Incremental Construction

The following example shows how to construct a triangulation in which we insert random points. In STEP 1, we generate one hundred random points in $\mathbb{R}^5$, which we then insert into a triangulation. In STEP 2, we ask the
triangulation to construct the set of edges (1 dimensional faces) incident to the vertex at infinity. It is easy to see that these edges are in bijection with the vertices on the convex hull of the points. This gives us a handy way to count the convex hull vertices.

```cpp
#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Triangulation.h>
#include <CGAL/algorithm.h>
#include <CGAL/Random.h>
#include <iterator>
#include <iostream>
#include <vector>

typedef CGAL::Cartesian_d<double> K;
typedef CGAL::Triangulation<K> Triangulation;

int main()
{
    const int D = 5; // we work in Euclidean 5-space
    const int N = 100; // we will insert 100 points
    // - - - - - - - - - - - - - - - - - - - - - - - - STEP 1
    CGAL::Random_points_in_cube_d<Triangulation::Point> rand_it(D, 1.0);
    std::vector<Triangulation::Point> points;
    CGAL::copy_n(rand_it, N, std::back_inserter(points));
    Triangulation t(D); // create triangulation
    assert(t.empty());
    t.insert(points.begin(), points.end()); // compute triangulation
    assert( t.is_valid() );

    // - - - - - - - - - - - - - - - - - - - - - - - - STEP 2
    typedef Triangulation::Face Face;
typedef std::vector<Face> Faces;
    Faces edges;
    std::back_insert_iterator<Faces> out(edges);
    t.tds().incident_faces(t.infinite_vertex(), 1, out); // collect faces of dimension 1 (edges) incident to the infinite vertex
    std::cout << "There are " << edges.size() << " vertices on the convex hull." << std::endl;
    return 0;
}
```

File: examples/Triangulation/triangulation.cpp

Traversing the facets of the convex hull

Remember that a triangulation triangulates the convex hull of its vertices. In general position, each facet of the convex hull is incident to one finite full cell and one infinite full cell. In fact there is a bijection between the infinite full cells and the facets of the convex hull. If vertices are not in general position, convex hull faces that are not simplices are triangulated. So, in order to traverse the convex hull facets, there are (at least) two possibilities:
The first is to iterate over the full cells of the triangulation and check if they are infinite or not:

```cpp
{ int i=0;
typedef Triangulation::Full_cell_iterator Full_cell_iterator;
typedef Triangulation::Facet Facet;

for( Full_cell_iterator cit = t.full_cells_begin();
    cit != t.full_cells_end(); ++cit )
{
    if( ! t.is_infinite(cit) )
        continue;
    Facet ft(cit, cit->index(t.infinite_vertex()));
    ++i; // |ft| is a facet of the convex hull
}
std::cout << "There are " << i << " facets on the convex hull."<<
std::endl;
}
```

**Remark:** the code example above is not self contained, it can be cut and paste at STEP 2 of triangulation.cpp program above.

A second possibility is to ask the triangulation to gather all the full cells incident to the infinite vertex: they form precisely the set of infinite full cells:

```cpp
{ int i=0;
typedef Triangulation::Full_cell_handle Full_cell_handle;
typedef Triangulation::Facet Facet;
typedef std::vector<Full_cell_handle> Full_cells;

Full_cells infinite_full_cells;
std::back_insert_iterator<Full_cells> out(infinite_full_cells);

t.incident_full_cells(t.infinite_vertex(), out);

for( Full_cells::iterator sit = infinite_full_cells.begin();
    sit != infinite_full_cells.end(); ++sit )
{
    Facet ft(*sit, (*sit)->index(t.infinite_vertex()));
    ++i; // |ft| is a facet of the convex hull
}
std::cout << "There are " << i << " facets on the convex hull."<<
std::endl;
}
```

**Remark:** the code example above is not self contained, it can be cut and paste at STEP 2 of triangulation.cpp program above.

One important difference between the two examples above is that the first uses little memory but traverses all the full cells, while the second visits only the infinite full cells but stores handles to them into a potentially big array.
1.4 Delaunay Triangulations

The class `CGAL::Delaunay_triangulation<DelauayTriangulationTraits, TriangulationDataStructure>` derives from `CGAL::Triangulation<DelauayTriangulationTraits, TriangulationDataStructure>` and adds further constraints to a triangulation, in that all its full cells must have the so-called Delaunay or empty-ball property: the interior of the ball circumscribing any full cell must be free from any vertex of the triangulation.

The *circumscribing ball* of a full cell is the ball having all vertices of the full cell on its boundary. In case of degeneracies (co-spherical points) the triangulation is not uniquely defined; Note however that the CGAL implementation computes a unique triangulation even in these cases.

When a new point \( p \) is inserted into a Delaunay triangulation, the finite full cells whose circumscribing sphere contain \( p \) are said to be in conflict with point \( p \). The set of full cells that are in conflict with \( p \) form the conflict zone. That conflict zone is augmented with the infinite full cells whose finite facet does not lie anymore on the convex hull of the triangulation (with \( p \) added). The full cells in the conflict zone are removed, leaving a hole that contains \( p \). That hole is “star shaped” around \( p \) and thus is easily re-triangulated using \( p \) as a center vertex.

Delaunay triangulations also support vertex removal.

1.4.1 Implementation

The class `CGAL::Delaunay_triangulation<DelauayTriangulationTraits, TriangulationDataStructure>` derives from `CGAL::Triangulation<DelauayTriangulationTraits, TriangulationDataStructure>`. It thus stores a model of the concept `TriangulationDataStructure` which is instantiated with a vertex type that stores a geometric point and allows its retrieval.

The template parameter `DelaunayTriangulationTraits` must be a model of the concept `DelaunayTriangulationTraits` which provides the geometric `Point` type as well as various geometric predicates used by the `Delaunay_triangulation` class. The concept `DelaunayTriangulationTraits` refines the concept `TriangulationTraits` by requiring a few other geometric predicates, necessary for the computation of Delaunay triangulations.

1.4.2 Examples

Access to the conflict zone and created full cells during point insertion

When using a full cell type containing additional custom information, it may be useful to get an efficient access to the full cells that are going to be erased upon the insertion of a new point in the Delaunay triangulation, and to the newly created full cells. The second part of code example below shows how one can have efficient access to both the conflict zone and the created full cells, while still retaining an efficient update of the Delaunay triangulation.

```cpp
#include <CGAL/Cartesian_d.h>
//#include <CGAL/Filtered_kernel_d.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Delaunay_triangulation.h>
#include <CGAL/algorithm.h>
#include <CGAL/Random.h>
#include <CGAL/Timer.h>
#include <iterator>
#include <iostream>
```
```cpp
#include <vector>
const int D=5;

typedef CGAL::Cartesian_d<double> K; //D;
//typedef CGAL::Filtered_kernel_d<KD> K;
typedef CGAL::Triangulation_ds_vertex< void > TDS_vertex;
typedef CGAL::Triangulation_vertex< K, int, TDS_vertex > Vertex;
typedef CGAL::Triangulation_ds_full_cell< void, CGAL::TDS_full_cell_default_storage_policy > TDS_cell;
typedef CGAL::Triangulation_full_cell< K, int, TDS_cell > Cell;
typedef CGAL::Triangulation_data_structure< CGAL::Dimension_tag<D> , Vertex, Cell > TDS;
typedef CGAL::Delaunay_triangulation<K,TDS> T;

int main(int argc, char **argv)
{
    int N = 100; if( argc > 2 ) N = atoi(argv[1]); // number of points
    CGAL::Timer cost; // timer

    // Instantiate a random point generator
    CGAL::Random rng(0);
typedef CGAL::Random_points_in_cube_d<T::Point> Random_points_iterator;
    Random_points_iterator rand_it(D, 1.0, rng);
    // Generate N random points
    std::vector<T::Point> points;
    CGAL::copy_n(rand_it, N, std::back_inserter(points));

    T t(D);
    assert(t.empty());

    // insert the points in the triangulation
    cost.reset(); cost.start();
    std::cout << " Delaunay triangulation of " << N << " points in dim " << D << std::endl;
    t.insert(points.begin(), points.end());
    std::cout << " done in " << cost.time() << " seconds. " << std::endl;
    assert( t.is_valid() );

    // insert with special operations in conflict zone and new created cells
    cost.reset();
    std::cout << " adding " << N << " other points " << std::endl;
    for(int i=0; i<N; ++i){
        T::Vertex_handle v;
        T::Face f(t.maximal_dimension());
        T::Facet ft;
        T::Full_cell_handle c;
        T::Locate_type lt;
typedef std::vector<T::Full_cell_handle> Full_cells;
    Full_cells zone, new_full_cells;
    std::back_insert_iterator<Full_cells> out(zone);
    c = t.locate(++rand_it, lt, f, ft, v);
```
1.5 Complexity and Performances

The current implementation locate points by walking in the triangulation, and sort the points with spatial sort to insert a set of points. Thus the theoretical complexity are $O(n \log n)$ for inserting $n$ random points and $O(n^{3/2})$ for inserting one point in a triangulation of $n$ random points.

The actual timing are the following:

⟨⟨todo⟩⟩

This section will be completed, when the code will be fully ready (and preferably with the new Kernel).

1.6 Design and Implementation History

This package is heavily inspired by the works of Monique Teillaud and Sylvain Pion (Triangulation_3) and Mariette Yvinec (Triangulation_2). The first version was written by Samuel Hornus and then pursued by Samuel Hornus and Olivier Devillers.
Triangulations
Reference Manual

Samuel Hornus and Olivier Devillers

A triangulation is a pure simplicial complex, connected and without singularities. Its faces are such that two of them either do not intersect or share a common face.

The basic triangulation class of CGAL is primarily designed to represent the triangulations of a set of points \( A \) in \( \mathbb{R}^d \). It can be viewed as a partition of the convex hull of \( A \) into simplices whose vertices are the points of \( A \). Together with the unbounded full cells having the convex hull boundary as its frontier, the triangulation forms a partition of \( \mathbb{R}^d \).

In order to deal only with full dimensional simplices (full cells), which is convenient for many applications, the space outside the convex hull is subdivided into full cells by considering that each convex hull facet is incident to an infinite full cell having as vertex an auxiliary vertex called the infinite vertex. In that way, each facet is incident to exactly two full cells and special cases at the boundary of the convex hull are simple to deal with.

A triangulation is a collection of vertices and full cells that are linked together through incidence and adjacency relations. Each full cell gives access to its incident vertices and to its adjacent full cells. Each vertex gives access to one of its incident full cells.

The vertices of a full cell are indexed in positive orientation, the positive orientation being defined by the orientation of the underlying Euclidean space \( \mathbb{R}^d \). The neighbors of a full cell are also indexed in such a way that the neighbor indexed by \( i \) is opposite to the vertex with the same index.

1.7 Reference Pages Sorted by Type

1.7.1 Concepts

Triangulation data structure

TriangulationDataStructure ................................................................. page 20

The above concept defines three types, *Vertex, Full_cell* and *Face*, that must respectively fulfill the following concepts:

TriangulationDSVertex ................................................................. page 34
TriangulationDSFullCell ............................................................... page 36
TriangulationDSFace ................................................................. page 40
1.7.2 Classes

Triangulation data structure

CGAL::Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell> ............................................ page 50
CGAL::Triangulation_ds_vertex<TriangulationDataStructure> ............................................. page 53
CGAL::Triangulation_ds_full_cell<TriangulationDataStructure, TDSFullCellStoragePolicy> ........ page 55
CGAL::Triangulation_face<TriangulationDataStructure> .............................................. page 57

(Geometric) triangulations

CGAL::Triangulation<TriangulationTraits, TriangulationDataStructure> ............................... page 58
CGAL::Delaunay_triangulation<DelaunayTriangulationTraits, TriangulationDataStructure> ....... page 68
CGAL::Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex> ................. page 72
CGAL::Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell> .......... page 74

1.7.3 Enums

CGAL::Triangulation::Locate_type ................................................................. page 76

1.8 Alphabetical List of Reference Pages

Cell .................................................................................................................. page 32
DelaunayTriangulationTraits ........................................................................ page 45
Delaunay_triangulation<DelaunayTriangulationTraits, TriangulationDataStructure> ............ page 68
Locate_type .................................................................................................... page 76
Triangulation<TriangulationTraits, TriangulationDataStructure> ................................. page 58
TriangulationDataStructure ............................................................................ page 20
TriangulationDSFace ...................................................................................... page 40
TriangulationDSFullCell ............................................................................... page 36
TriangulationDSTraverse .............................................................................. page 34
TriangulationFullCell .................................................................................... page 49
TriangulationTraits ......................................................... page 42
TriangulationVertex ........................................................ page 47
Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell> ... page 50
Triangulation_ds_full_cell<TriangulationDataStructure, TDSFullCellStoragePolicy> ................ page 55
Triangulation_ds_vertex<TriangulationDataStructure> ................................................. page 53
Triangulation_face<TriangulationDataStructure> ......................................................... page 57
Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell> ..................... page 74
Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex> ........................ page 72
Vertex ........................................................................ page 30
TriangulationDataStructure

Definition

The TriangulationDataStructure concept describes objects responsible for storing and maintaining the combinatorial part of a $d$-dimensional pure simplicial complex (all simplices that are not sub-faces of another have the same dimension $d$). Its topology is the topology of the sphere $S^d$ with $d \in [-2, D]$. In a pure (or homogeneous) simplicial $d$-complex, all faces are sub-faces of some $d$-simplex. (A simplex is also a face of itself.) In particular, it does not contain any $d+1$-face, and any $d-1$-face belongs to exactly two $d$-dimensional full cells.

Values of $d$ (the current dimension of the complex) include

-2 This corresponds to the non-existence of any object in the triangulation.
-1 This corresponds to a single vertex and a single full cell, which is also the unique vertex and the unique full cell in the TriangulationDataStructure. In a geometric realization of the TriangulationDataStructure (e.g., in a Triangulation<TriangulationTraits, TriangulationDataStructure> or a Delaunay_triangulation<DelaunayTriangulationTraits, TriangulationDataStructure>), this vertex corresponds to the vertex at infinity.

0 This corresponds to two vertices, each incident to one 0-face; the two full cells being neighbor of each other. This is the unique triangulation of the 0-sphere.

$d > 0$ This corresponds to a standard triangulation of the sphere $S^d$.

An $i$-simplex is a simplex with $i+1$ vertices. An $i$-simplex $\sigma$ is incident to a $j$-simplex $\sigma'$, $j < i$, if and only if $\sigma'$ is a proper face of $\sigma$.

We call a 0-simplex a vertex, a $(d-1)$-simplex a facet and a $d$-simplex a full cell. A face can have any dimension. Two full cells are adjacent if they share a facet. Two faces are incident if one is included un the other.

Has Models

CGAL::Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell>

TriangulationDataStructure:: Vertex
Vertex type.

TriangulationDataStructure:: Full_cell
Full cell type.

The concept TriangulationDataStructure also defines a type for describing facets of the triangulation with codimension 1:

TriangulationDataStructure:: Facet
The constructor Facet(c,i) constructs a Facet representing the facet of full cell $c$ opposite to its $i$-th vertex. Its dimension is current_dimension()-1.

TriangulationDataStructure:: Face
A model of the concept TriangulationDSFace.

Vertices and full cells are manipulated via handles. Handles support the usual two dereference operators operator* and operator->.
TriangulationDataStructure:: Vertex_handle
Handle to a Vertex.

TriangulationDataStructure:: Full_cell_handle
Handle to a Full_cell.

Requirements for Vertex and Full_cell are described in concepts TriangulationDataStructure::Vertex and TriangulationDataStructure::FullCell (page 30 and page ??).

advanced

TriangulationDataStructure:: template <typename Vb2> struct Rebind_vertex
This nested template class allows to get the type of a triangulation data structure that only changes the vertex type. It has to define a type Other which is a rebound triangulation data structure, that is, the one whose TriangulationDSVertexBase will be Vb2.

TriangulationDataStructure:: template <typename Fcb2> struct Rebind_full_cell
This nested template class allows to get the type of a triangulation data structure that only changes the full cell type. It has to define a type Other which is a rebound triangulation data structure, that is, the one whose TriangulationDSFullCellBase will be Fcb2.

advanced

Vertices, facets and full cells can be iterated over using iterators. Iterators support the usual two dereference operators operator* and operator->.

TriangulationDataStructure:: Vertex_iterator
Iterator over the list of vertices.

TriangulationDataStructure:: Full_cell_iterator
Iterator over the list of full cells.

TriangulationDataStructure:: Facet_iterator
Iterator over the facets of the complex.

TriangulationDataStructure:: size_type
Size type (an unsigned integral type)

TriangulationDataStructure:: difference_type
Difference type (a signed integral type)

Creation

TriangulationDataStructure tds( int dim = 0);
Creates an instance tds of type TriangulationDataStructure. The maximal dimension of its full cells is dim and tds is initialized to the empty triangulation. Thus, tds.current_dimension() equals -2. The parameter dim can be ignored by the constructor if it is already known at compile-time. Otherwise, the following precondition holds:
Precondition: dim>0.
Queries

**int tds.maximal_dimension() const**  
Returns the maximal dimension of the full dimensional cells that can be stored in the triangulation `tds`.  
*Postcondition:* the returned value is positive.

**int tds.current_dimension() const**  
Returns the dimension of the full dimensional cells stored in the triangulation. It holds that `tds.current_dimension() = -2` if and only if `tds.empty()` is true.  
*Postcondition:* the returned value `d` satisfies $-2 \leq d \leq tds.maximal_dimension()`.

**bool tds.empty() const**  
Returns `true` if the triangulation contains nothing. Returns `false` otherwise.

**size_type tds.number_of_vertices() const**  
Returns the number of vertices in the triangulation.

**size_type tds.number_of_full_cells() const**  
Returns the number of full cells in the triangulation.

**bool tds.is_vertex( Vertex_handle v) const**  
Tests whether `v` is a vertex of the triangulation.

**bool tds.is_full_cell( Full_cell_handle c) const**  
Tests whether `c` is a full cell of the triangulation.

**template<typename TraversalPredicate, typename OutputIterator> void tds.full_cells( Full_cell_handle c, TraversalPredicate & tp, OutputIterator & out) const**  
This function computes (gathers) a connected set of full cells satisfying a common criterion. Call them good full cells. It is assumed that the argument `c` is a good full cell. The full cells are then recursively explored by examining if, from a given good full cell, its adjacent full cells are also good. The argument `tp` is a predicate that takes as argument a Facet whose defining Full_cell is good. The predicate must return `true` if the traversal of that Facet leads to a good full cell. All the good full cells are output into the last argument `out`.  
*Precondition:* `c!=Full_cell_handle()` and `tp(c)==true`.

**template<typename OutputIterator> OutputIterator tds.incident_full_cells( Vertex_handle v, OutputIterator out) const**  
Insert in `out` all the full cells that are incident to the vertex `v`, i.e., the full cells that have the Vertex `v` as a vertex. Returns the output iterator.  
*Precondition:* `v!=Vertex_handle()`.
template< typename OutputIterator >
OutputIterator tds.incident_full_cells( Face f, OutputIterator out) const

Insert in out all the full cells that are incident to the face f, i.e., the full cells that have the Face f as a subface. Returns the output iterator.
Precondition: f.full_cell()!=Full_cell_handle().

template< typename OutputIterator >
OutputIterator tds.star( Face f, OutputIterator out) const

Insert in out all the full cells that share at least one vertex with the Face f. Returns the output iterator.

template< typename OutputIterator >
OutputIterator tds.incident_faces( Vertex(handle v, const int d, OutputIterator out)

Constructs all the Faces of dimension d incident to Vertex v and inserts them in the OutputIterator out. If d >= tds.current_dimension(), then no Face is constructed.
Precondition: 0 < d and v!=Vertex_handle().

Accessing the vertices

Vertex(handle) tds.vertex( Full_cell_handle c, const int i) const

Returns a handle to the i-th Vertex of the Full_cell c.
Precondition: 0 ≤ i ≤ tds.current_dimension() and c!=Full_cell_handle().

int tds.mirror_index( Full_cell_handle c, int i) const

Returns the index of the vertex mirror of the i-th vertex of c. Equivalently, returns the index of c as a neighbor of its i-th neighbor.
Precondition: 0 ≤ i ≤ tds.current_dimension() and c!=Full_cell_handle().

Vertex(handle) tds.mirror_vertex( Full_cell_handle c, int i) const

Returns the vertex mirror of the i-th vertex of c. Equivalently, returns the vertex of the i-th neighbor of c that is not vertex of c.
Precondition: 0 ≤ i ≤ tds.current_dimension() and c!=Full_cell_handle().

Vertex_iterator tds.vertices_begin()
The first vertex of tds. User has no control on the order.

Vertex_iterator tds.vertices_end()
The beyond vertex of tds.
Accessing the full cells

```cpp
Full_cell_handle tds.full_cell( Vertex_handle v) const
```

Returns a full cell incident to `Vertex v`. Note that this full cell is not unique (`v` is typically incident to more than one full cell).

*Precondition:* `v` is not the default constructed `Vertex_handle`

```cpp
Full_cell_handle tds.neighbor( Full_cell_handle c, int i) const
```

Returns a `Full_cell_handle` pointing to the `Full_cell` opposite to the `i`-th vertex of `c`.

*Precondition:* `0 ≤ i ≤ tds.current_dimension()` and `c` is not the default constructed `Full_cell_handle`

```cpp
Full_cell_iterator tds.full_cells_begin()
```

The first full cell of `tds`. User has no control on the order.

```cpp
Full_cell_iterator tds.full_cells_end()
```

The beyond full cell of `tds`.

Faces and Facets

```cpp
Facet_iterator tds.facets_begin()
```

Iterator to the first facet of the triangulation.

```cpp
Facet_iterator tds.facets_end()
```

Iterator to the beyond facet of the triangulation.

```cpp
Full_cell_handle tds.full_cell( Facet f) const
```

Returns a full cell containing the facet `f`

```cpp
int tds.index_of_covertex( Facet f) const
```

Returns the index of vertex of the full cell `c=tds.full_cell(f)` which does not belong to `c`.

Vertex insertion

```cpp
Vertex_handle tds.insert_in_full_cell( Full_cell_handle c)
```

Inserts a new vertex `v` in the full cell `c` and returns a handle to it. The full cell `c` is subdivided into `tds.current_dimension()+1` full cells which share the vertex `v` (see Figure 1.3).

*Precondition:* Current dimension is positive and `c` is a full cell of `tds`.

```cpp
Vertex_handle tds.insert_in_face( Face f)
```

Inserts a vertex in the triangulation data structure by subdividing the `Face f`. Returns a handle to the newly created `Vertex` (see Figure below 1.4).
Figure 1.3: Insertion in a full cell, \( d = 2 \)

Figure 1.4: Insertion in face, \( d = 3 \)

```
Vertex_handle tds.insert_in_facet(Facet ft)  
Inserts a vertex in the triangulation data structure by subdividing the Facet ft. Returns a handle to the newly created Vertex.
```

```
template< class ForwardIterator >
Vertex_handle tds.insert_in_hole(ForwardIterator start, ForwardIterator end, Facet f)  
The full cells in the range \( C = [start, end) \) are removed, thus forming a hole \( H \). A Vertex is inserted and connected to the boundary of the hole in order to “fill it”. A Vertex_handle to the new Vertex is returned (see Figure 1.5). 
Precondition: \( c \) belongs to \( C \) and \( c \rightarrow \text{neighbor}(i) \) does not, with \( f = (c, i) \). \( H \) the union of full cells in \( C \) is simply connected and its boundary \( \partial H \) is a combinatorial triangulation of the sphere \( S^{d-1} \). All vertices of the triangulation are on \( \partial H \).
```

```
template< class ForwardIterator, class OutputIterator >
Vertex_handle tds.insert_in_hole(ForwardIterator start, ForwardIterator end, Facet f, OutputIterator out)  
Same as above, but handles to the new full cells are appended to the out output iterator.
```
Figure 1.5: Insertion in a hole, $d = 2$

Vertex\_handle \quad tds.insert\_increase\_dimension(\; Vertex\_handle \; star) \\

Transfers a triangulation of the sphere $S^d$ into the triangulation of the sphere $S^{d+1}$ by adding a new vertex $v$. $v$ is used to triangulate one of the two half-spheres of $S^{d+1}$ ($v$ is added as $(d+2)^{th}$ vertex to all full cells) and $star$ is used to triangulate the other half-sphere (all full cells that do not already have star as vertex are duplicated, and $star$ replaces $v$ in these full cells). The indexing of the vertices in the full cell is such that, if $f$ was a full cell of maximal dimension in the initial complex, then $(f,v)$, in this order, is the corresponding full cell in the updated triangulation. A handle to $v$ is returned (see Figure 1.6).

Precondition: $tds$. If the current dimension is -2 (empty triangulation), then $star$ has to be omitted, otherwise the current dimension must be strictly less than the maximal dimension and $star$ must be a vertex of $tds$.

Figure 1.6: Insertion, increasing the dimension from $d = 1$ to $d = 2$

advanced

Full\_cell\_handle \quad tds.new\_full\_cell() \\

Adds a new full cell to $tds$ and returns a handle to it. The new full cell has no vertex and no neighbor yet.

Vertex\_handle \quad tds.new\_vertex() \\

Adds a new vertex to $tds$ and returns a handle to it. The new vertex has no associated full cell nor index yet.
void tds.associate_vertex_with_full_cell( Full_cell_handle c, int i, Vertex_handle v)

Sets the i-th vertex of c to v and, if v is non-NULL, sets c as the incident full cell of v.

void tds.set_neighbors( Full_cell_handle ci, int i, Full_cell_handle cj, int j)

Sets the neighbor opposite to vertex i of Full_cell ci to cj. Sets the neighbor opposite to vertex j of Full_cell cj to ci.

void tds.set_current_dimension( int d)

Forces the current dimension of the complex to d.
Precondition: \(-1 \leq d \leq \text{maximal_dimension}()\).

---

**Vertex removal**

void tds.clear()

Reinitializes tds to the empty complex.

Vertex_handle tds.collapse_face( Face f)

Contracts the Face f to a single vertex. Returns a handle to that vertex (see Figure 1.7).
Precondition: The boundary of the full cells incident to f is a topological sphere of dimension tds.current_dimension()-1).

Figure 1.7: Collapsing an edge in dimension \(d = 3\), v is returned
void tds.remove_decrease_dimension( Vertex_handle v, Vertex_handle star)

This method does exactly the opposite of insert_increase_dimension(): \(v\) is removed, full cells not containing \(\text{star}\) are removed full cells containing \(\text{star}\) but not \(v\) loose vertex \(\text{star}\) full cells containing \(\text{star}\) and \(v\) loose vertex \(v\) (see Figure 1.6).

Precondition: All cells contains either \(\text{star}\) or \(v\). Edge \(\text{star-}v\) exists in the triangulation and current_dimension()!\(=2\).

advanced

void tds.delete_vertex( Vertex_handle v)

Remove the vertex \(v\) from the triangulation.

void tds.delete_full_cell( Full_cell_handle c)

Remove the full cell \(c\) from the triangulation.

template< typename ForwardIterator >
void tds.delete_full_cells( ForwardIterator start, ForwardIterator end)

Remove the full cells in the range \([\text{start},\text{end})\) from the triangulation.

debugging support

Validity check

bool tds.is_valid( bool verbose=false) const

Partially checks whether \(tds\) is indeed a triangulation. It must at least

- check the validity of the vertices and full cells of \(tds\) by calling their respective is_valid method.

- check that each full cell has no duplicate vertices and has as many neighbors as its number of facets (current_dimension()+1).

- check that each full cell share exactly \(tds\).current_dimension() vertices with each of its neighbor.

Returns true if all the tests pass, false if any test fails. See the documentation for the models of this concept to see the additionnal (if any) validity checks that they implement.
**Input/Output**

\[
\text{istream} \ & \ \text{istream} \ & \ \text{istream} \ & \ \text{istream} \  \triangleright \  \\text{tds}
\]

Reads a combinatorial triangulation from \textit{is} and assigns it to \textit{tds}.

\textbf{Precondition:} The dimension of the input complex must be less than or equal to \textit{tds.maximal_dimension()}. 

\[
\text{ostream} \ & \ \text{ostream} \ & \ \text{ostream} \ & \ \text{ostream} \  \leftarrow \  \\text{tds}
\]

Writes \textit{tds} into the output stream \textit{os}.

The information stored in the \textit{iostream is}:

- the current dimension (which must be \textless\textless=\textit{tds.maximal_dimension()},
- the number of vertices,
- for each vertex the information of that vertex,
- the number of full cells,
- for each full cell the indices of its vertices and extra information for that full cell,
- for each full cell the indices of its neighbors.

The indices of vertices and full cells correspond to the order in the file, the user cannot control it. The classes \textit{Vertex} and \textit{Full_cell} have to provide the relevant I/O operators (possibly empty).

\textbf{See Also}

\textit{TriangulationDSVertex}
\textit{TriangulationDSFullCell}
\textit{TriangulationDSFace}
\textit{Triangulation}
TriangulationDataStructure::Vertex

Definition

The concept Vertex represents the vertex class of a triangulation data structure. It must define the types and operations listed in this section. Some of these requirements are of geometric nature, they are optional when using the triangulation data structure class alone. They become compulsory when the triangulation data structure is used as a layer for the geometric triangulation class.

Types

**Vertex:: Point**  
Optional for the triangulation data structure alone.

The class Vertex defines types that are the same as some of the types defined by the triangulation data structure class TriangulationDataStructure.

```cpp
typedef TriangulationDataStructure Triangulation data structure;
typedef TriangulationDataStructure::Vertex handle Vertex handle;
typedef TriangulationDataStructure::Full_cell handle Full_cell handle;
```

Creation

In order to obtain new vertices or destruct unused vertices, the user must call the `new_vertex()` or `delete_vertex()` method of the triangulation data structure.

Operations

Access Functions

```cpp
Full_cell_handle v.full_cell() const  
Returns a full cell of the triangulation having v as vertex.

Point v.point() const  
Returns the point stored in the vertex. Optional for the triangulation data structure alone.
```

Setting

```cpp
void v.set_full_cell( Full_cell_handle c)  
Sets the incident cell to c.

void v.set_point( Point p)  
Sets the point to p. Optional for the triangulation data structure alone.
```

debugging support
Checking

bool v.is_valid( bool verbose = false) const

Checks the validity of the vertex. Must check that its incident cell has this vertex. The validity of the base vertex is also checked.
When verbose is set to true, messages are printed to give a precise indication on the kind of invalidity encountered.

See Also

TriangulationDataStructure::FullCell
TriangulationDataStructure.
TriangulationDataStructure::Cell

Definition

The concept Cell stores Vertex handles to its vertices and Full cell handles to its neighbors. The vertices are indexed 0, 1, ..., d in consistent order. The neighbor indexed i lies opposite to vertex i.

Types

The class Cell defines the following types.

```cpp
typedef TriangulationDataStructure Triangulation_data_structure;
typedef TriangulationDataStructure::Vertex handle Vertex_handle;
typedef TriangulationDataStructure::Full cell handle Full_cell_handle;
```

Creation

In order to obtain new cells or destruct unused cells, the user must call the `new_full_cell()` and `delete_full_cell()` methods of the triangulation data structure.

Operations

Access Functions

```
Vertex_handle c.vertex( int i) const
int c.index( Vertex_handle v) const
bool c.has_vertex( Vertex_handle v) const
bool c.has_vertex( Vertex_handle v, int &i) const
Full_cell_handle c.neighbor( const int i) const
int c.index( Full_cell_handle n) const
bool c.has_neighbor( Full_cell_handle n) const
bool c.has_neighbor( Full_cell_handle n, int &i) const
```

 setting

```
void c.set_vertex( int i, Vertex_handle v)
```
void c.set_neighbor( int i, Full_cell_handle n) 
Sets neighbor \(i\) to \(n\).

Precondition: \(i \in [0, D]\).
**TriangulationDSVertex**

**Definition**

The concept TriangulationDSVertex describes what a vertex is in a model of the concept TriangulationDataStructure. It sets requirements of combinatorial nature only, as geometry is not concerned here. In particular, we only require that the vertex holds a handle to a full cell incident to it in the triangulation.

**Has Models**

CGAL::Triangulation_ds_vertex<TriangulationDataStructure>

CGAL::Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>

**Types**

TriangulationDSVertex:: Full_cell_handle  
A handle to a cell. It must be the same as the nested type TriangulationDataStructure::Full_cell_handle of the TriangulationDataStructure in which the TriangulationDSVertex is defined/used.

**Rebind TDS**

TriangulationDSVertex:: template <typename TDS2> Rebind_TDS  
This nested template class has to define a type Other which is the rebound vertex, that is, the one whose Triangulation_data_structure will be the actually used one. The Other type will be the real base class of Triangulation_data_structure::Vertex.

**Creation**

TriangulationDSVertex v;  
The default constructor (no incident full cell is set).

TriangulationDSVertex v( Full_cell_handle c);  
Sets the incident full cell to c.  
Precondition: c must not be the default-constructed Full_cell_handle.

**Operations**

`void v.set_full_cell( Full_cell_handle c)`  
Set c as the vertex’s incident full cell.  
Precondition: c must not be the default-constructed Full_cell_handle.

`Full_cell_handle v.full_cell() const`  
Returns a handle to a full cell incident to the vertex.

---

debugging support
Validity check

```cpp
bool v.is_valid( bool verbose=false) const
```

Performs some validity checks on the vertex `v`. It must at least check that `v` has an incident full cell, which in turn must contain `v` as one of its vertices. Returns `true` if all the tests pass, `false` if any test fails. See the documentation for the models of this concept to see the additional (if any) validity checks that they implement.

---

Memory management

```cpp
void* v.for_compact_container() const
void*& v.for_compact_container()
```

These member functions are required by `Triangulation_data_structure` because it uses `Compact_container` to store its cells. See the documentation of `Compact_container` for the exact requirements.

---

Input/Output

These operators can be used directly and are called by the I/O operator of class `TriangulationDataStructure`.

```cpp
template<class TriangulationDataStructure>
istream& istream & is >> Triangulation_ds_vertex<TriangulationDataStructure> & v
```

Reads (possibly) non-combinatorial information about a vertex from the stream `is` into `v`.

```cpp
template<class TriangulationDataStructure>
ostream& ostream & os << Triangulation_ds_vertex<TriangulationDataStructure> v
```

Writes (possibly) non-combinatorial information about vertex `v` to the stream `os`.

See Also

- `TriangulationDSFullCell`
- `TriangulationDSFace`
- `TriangulationDataStructure`
- `Triangulation`
TriangulationDSFullCell

Definition

The concept TriangulationDSFullCell describes what a full cell is in a model of the concept Triangulation-DataStructure. It sets requirements of combinatorial nature only, as geometry is not concerned here. In the context of triangulation, the term full cell refers to a face of maximal dimension. This maximality characteristic is emphasized by using the adjective full.

A TriangulationDSFullCell is responsible for storing handles to the vertices of a full cell as well as handles to its neighbors.

Has Models

CGAL::Triangulation_ds_full_cell<TriangulationDataStructure,DSFullCellStoragePolicy>
CGAL::Triangulation_full_cell<TriangulationTraits,Data,TriangulationDSFullCell>

Types

* TriangulationDSFullCell::Vertex_handle
  A handle to a vertex. It must be the same as the nested type TriangulationDataStructure::Vertex_handle of the TriangulationDataStructure in which the TriangulationDSFullCell is defined/used.

* TriangulationDSFullCell::Vertex_handle_iterator
  An iterator over the handles to the vertices of the full cell.

* TriangulationDSFullCell::Full_cell_handle
  A handle to a full cell. It must be the same as the nested type TriangulationDataStructure::Full_cell_handle of the TriangulationDataStructure in which the TriangulationDSFullCell is defined/used.

typedef TriangulationDataStructure::Full_cell_data

  TDS_data;

  A data member of this type has to be stored and accessible through access function below.

* TriangulationDSFullCell::template<typename TDS2> Rebind_TDS
  This nested template class has to define a type Other which is the rebound vertex, that is, the one whose Triangulation_data_structure will be the actually used one. The Other type will be the real base class of Triangulation_data_structure::Full_cell.
Creation

`TriangulationDSFullCell c( int dmax);` Sets the maximum possible dimension of the full cell.

`TriangulationDSFullCell c(fc);` Copy constructor.

If you want to create a full cell as part of a `TriangulationDataStructure`, you would rather want to call the `new_full_cell()` from the latter concept, as it is not possible to incorporate an existing external full cell into a triangulation.

Access functions

`int c.maximal_dimension() const` Returns one less than the maximum number of vertices that the full cell can store. This does not return the dimension of the actual full cell stored in `c`.

`Vertex_handle_iterator c.vertices_begin() const` Returns an iterator to the first `Vertex_handle` stored in the full cell.

`Vertex_handle_iterator c.vertices_end() const` Returns an iterator pointing beyond the last `Vertex_handle` stored in the full cell.

`Vertex_handle c.vertex( const int i) const` Returns the `i`-th vertex of the full cell. 
*Precondition:* `0 \leq i \leq \text{maximal\_dimension}()`.

`Full_cell_handle c.neighbor( const int i) const` Returns the full cell opposite to the `i`-th vertex of the full cell `c`. 
*Precondition:* `0 \leq i \leq \text{maximal\_dimension}()`.

`int c.mirror_index( const int i) const` Returns the index `j` of the full cell `c` as a neighbor in the full cell `c.neighbor(i)`. If the returned integer is not negative, it holds that `c.neighbor(i)->neighbor(j) == c`. Returns `-1` if `c` has no adjacent full cell of index `i`. 
*Precondition:* `0 \leq i \leq \text{maximal\_dimension}()`.

`int c.index( Full_cell_handle n) const` Returns the index `i` such that `c.neighbor(i)==n`. 
*Precondition:* `n` must be a neighbor of `c`.

`int c.index( Vertex_handle v) const` Returns the index `i` of the vertex `v` such that `c.vertex(i)==v`. 
*Precondition:* `v` must be a vertex of the `c`.

`TDS\_data c.get_tds_data() const` Returns the data member of type `TDS_data`. It is typically used to mark the full cell as visited during operations on a `TriangulationDataStructure`. 

`TDS\_data& c.get_tds_data()` Same as above, but returns a reference to a non-const object.
**advanced**

```cpp
Vertex_handle c.mirror_vertex( const int i, const int cur_dim) const
```

Returns a handle to the mirror vertex of the \( i \)-th vertex of full cell \( c \). This function works even if the adjacency information stored in the neighbor full cell \(*c\.neighbor(i)\) is corrupted. This is useful when temporary corruption is necessary during surgical operation on a triangulation.

**Precondition**: \( 0 \leq i, cur\_dim \leq \text{maximal\_dimension}() \).

---

**Update functions**

```cpp
void c.set_vertex( const int i, Vertex_handle v) 

void c.set_neighbor( const int i, Full_cell_handle n)

void c.set_mirror_index( const int i, const int index)

void c.swap_vertices( int d1, int d2)
```

- **Sets the** \( i \)-th vertex of the full cell. **Precondition**: \( 0 \leq i \leq \text{maximal\_dimension}() \).
- **Sets the** \( i \)-th neighbor of \( c \) to \( n \). Full cell \( n \) is opposite to the \( i \)-th vertex of \( c \). **Precondition**: \( 0 \leq i \leq \text{maximal\_dimension}() \).
- **Sets the mirror index of the** \( i \)-th vertex of \( c \) to \( index \). This corresponds to the index, in \( c\rightarrow neighbor(i) \), of the full cell \( c \). Note: an implementation of the concept \( c \) may choose not to store mirror indices, in which case this function should do nothing. **Precondition**: \( 0 \leq i \leq \text{maximal\_dimension}() \).
- **Switches the orientation of the full cell** \( c \) by swapping its vertices with index \( d1 \) and \( d2 \). **Precondition**: \( 0 \leq d1, d2 \leq \text{maximal\_dimension}() \).

---

**Queries**

```cpp
bool c.has_vertex( Vertex_handle v) const

bool c.has_vertex( Vertex_handle v, int & ret) const

bool c.has_neighbor( Full_cell_handle n) const

bool c.has_neighbor( Full_cell_handle n, int & ret) const
```

- **Returns true** if the vertex \( v \) is a vertex of the full cell \( c \). **Returns false** otherwise.
- **Returns true** and sets the value of \( ret \) to the index of \( v \) in \( c \) if the vertex \( v \) is a vertex of the full cell \( c \). **Returns false** otherwise.
- **Returns true** if the full cell \( n \) is a neighbor of the full cell \( c \). **Returns false** otherwise.
- **Returns true** and sets the value of \( ret \) to the index of \( n \) as a neighbor of \( c \) if the full cell \( n \) is a neighbor of the full cell \( c \). **Returns false** otherwise.
Validity check

```cpp
bool c.is_valid( bool verbose=false) const
```

Performs some validity checks on the full cell `c`. It must at least check that for each existing neighbor `n`, `c` is also a neighbor of `n`. Returns `true` if all the tests pass, `false` if any test fails. See the documentation for the models of this concept to see the additional (if any) validity checks that they implement.

Memory management

```cpp
void* c.for_compact_container() const
void*& c.for_compact_container()
```

These member functions are required by `Triangulation_data_structure` because it uses `Compact_container` to store its cells. See the documentation of `Compact_container` for the exact requirements.

Input/Output

These operators can be used directly and are called by the I/O operator of class `TriangulationDataStructure`.

```cpp
template<class TriangulationDataStructure>
istream& is >> Triangulation_ds_full_cell<TriangulationDataStructure> & c
```

Reads (possibly) non-combinatorial information about a full cell from the stream `is` into `c`.

```cpp
template<class TriangulationDataStructure>
ostream& os << Triangulation_ds_full_cell<TriangulationDataStructure> c
```

Writes (possibly) non-combinatorial information about full cell `c` to the stream `os`.

See Also

`TriangulationDSVertex`
`TriangulationDSFace`
`TriangulationDataStructure`
`Triangulation`
**TriangulationDSFace**

**Definition**

A TriangulationDSFace describes a $k$-face $f$ in a triangulation. It gives access to a handle to a full cell $c$ containing the face $f$ in its boundary, as well as the indices of the vertices of $f$ in $c$. It must hold that $f$ is a *proper* face of full cell $c$, i.e., the dimension of $f$ is strictly less than the dimension of $c$.

**Types**

- **TriangulationDSFace:: Full_cell_handle**
  
  Must be the same as the nested type `TriangulationDataStructure::Full_cell_handle` of the `TriangulationDataStructure` in which the `TriangulationDSFace` is defined/used.

- **TriangulationDSFace:: Vertex_handle**
  
  Must be the same as the nested type `TriangulationDataStructure::Vertex_handle` of the `TriangulationDataStructure` in which the `TriangulationDSFace` is defined/used.

**Has Models**

- `CGAL::Triangulation_face<TriangulationDataStructure>`.

**Creation**

There is no default constructor, since the maximal dimension (of the full cells) must be known by the constructors of a TriangulationDSFace.

- `TriangulationDSFace f(Triangulation_face g);`
  
  Copy constructor.

- `TriangulationDSFace f(Full_cell_handle c);`
  
  Sets the Face’s full cell to $c$ and the maximal dimension to $c$.maximal_dimension().
  
  **Precondition:** $c != Full_cell_handle()$

- `TriangulationDSFace f(const int ad);`
  
  Setup the Face knowing the maximal dimension $ad$. Sets the Face’s full cell to the default-constructed one.

**Access functions**

- `Full_cell_handle f.full_cell() const`
  
  Returns a handle to a cell that has the face in its boundary.

- `int f.face_dimension() const`
  
  Returns the dimension of the face (one less than the number of vertices).
int     f.index( int i) const     Returns the index of the i-th vertex of the face in the cell f.full_cell().
          Precondition: 0 ≤ i ≤ f.face_dimension().

Vertex_handle     f.vertex( int i) const     Returns a handle to the i-th Vertex of the face in the cell f.full_cell().
          Precondition: 0 ≤ i ≤ f.face_dimension().

Update functions

void     f.clear()     Sets the facet to the empty set. Maximal dimension remains unchanged.

void     f.set_full_cell( Full_cell_handle c)     Sets the cell of the face to c.
          Precondition: c!=Full_cell_handle()

void     f.set_index( int i, int j)     Sets the index of the i-th vertex of the face to be the j-th vertex of the full cell.
          Precondition: 0 ≤ i ≤ f.full_cell()->face_dimension().
          Precondition: 0 ≤ j ≤ f.full_cell()->maximal_dimension().

See Also

TriangulationDataStructure::FullCell
TriangulationDataStructure::Vertex
TriangulationDataStructure
Triangulation
TriangulationTraits

Definition

The concept TriangulationTraits describes the various types and functions that a class must provide as the first parameter (TriangulationTraits) to the class template Triangulation<TriangulationTraits, TriangulationDataStructure>. It brings the geometric ingredient to the definition of a triangulation, while the combinatorial ingredient is brought by the second template parameter, TriangulationDataStructure.

Inserting a range of points in a triangulation is optimized using spatial sorting, thus besides the requirements below, a class provided as TriangulationTraits should also satisfy the concept SpatialSortingTraits_d.

Refines

SpatialSortingTraits_d

If a range of points is inserted, the traits must refine SpatialSortingTraits_d. This is not needed if the points are inserted one by one.

Types

**TriangulationTraits:: Dimension**

A type representing the dimension of the underlying space. It can be static (Maximal_dimension=CGAL::Dimension_tag<int dim>) or dynamic (Maximal_dimension=CGAL::Dynamic_dimension_tag). This dimension must match the dimension of the predicate Orientation_d but not necessarily the one of Point_d.

**TriangulationTraits:: Point_d**

A type representing a point in Euclidean space.

**TriangulationTraits:: Point_dimension_d**

Functor returning the dimension of a Point_d. Must provide int operator()(Point_d p) returning the dimension of p.

**TriangulationTraits:: Orientation_d**

A predicate object that must provide the templated operator template<typename ForwardIterator> Orientation operator()(ForwardIterator start, ForwardIterator end).

The operator returns CGAL::POSITIVE, CGAL::NEGATIVE or CGAL::COPLANAR depending on the orientation of the simplex defined by the points in the range [start, end).

Precondition: std::distance(start,end)=D+1, where Point_dimension_d(*it) is D for all it in [start,end).
**TriangulationTraits:: Contained in affine hull**

A predicate object that must provide the templated operator

```
template<typename ForwardIterator> bool operator()(ForwardIterator start, ForwardIterator end, const Point_d & p).
```

The operator returns `true` if and only if point `p` is contained in the affine space spanned by the points in the range `[start, end)`. That affine space is also called the **affine hull** of the points in the range.

*Precondition:* The `k` points in the range must be affinely independent. `Point_dimension_d(*it)` is `D` for all `it` in `[start,end)`, for some `D`. `2 ≤ k ≤ D`.

In the `D`-dimensional oriented space, a `k − 1` dimensional subspace (flat) define by `k` points can be oriented in two different ways. Choosing the orientation of any simplex defined by `k` points fix the orientation of all other simplices. To be able to orient lower dimensional flats, we use the following classes:

**TriangulationTraits:: Flat_orientation_d**

A type representing an orientation of an affine subspace of dimension `k` strictly smaller than the maximal dimension.

**TriangulationTraits:: Construct_flat_orientation_d**

A construction object that must provide the templated operator

```
template<typename ForwardIterator> Flat_orientation_d operator()(ForwardIterator start, ForwardIterator end).
```

The flat spanned by the points in the range `R=[start, end)` can be oriented in two different ways, the operator returns an object that allow to orient that flat so that `R=[start, end)` defines a positive simplex.

*Precondition:* The `k` points in the range must be affinely independent. `Point_dimension_d(*it)` is `D` for all `it` in `R` for some `D`. `2 ≤ k ≤ D`.

**TriangulationTraits:: In_flat_orientation_d**

A predicate object that must provide the templated operator

```
template<typename ForwardIterator> Orientation operator()(Flat_orientation_d orient,ForwardIterator start, ForwardIterator end).
```

The operator returns `CGAL::POSITIVE`, `CGAL::NEGATIVE` or `CGAL::COPLANAR` depending on the orientation of the simplex defined by the points in the range `[start, end)`. The points are supposed to belong to the lower dimensional flat whose orientation is given by `orient`.

*Precondition:* `std::distance(start,end)=k` where `k` is the number of points used to construct `orient`. `Point_dimension_d(*it)` is `D` for all `it` in `[start,end)` where `D` is the dimension of the points used to construct `orient`. `2 ≤ k ≤ D`. 
A predicate object that must provide the operator
\[ \text{Comparison result} \text{ operator}(\text{const Point}_d & p, \text{const Point}_d & q). \]
The operator returns \textit{SMALLER} if \( p \) is lexicographically smaller than point \( q \), \textit{EQUAL} if both points are the same and \textit{LARGER} otherwise.

Creation

\texttt{TriangulationTraits traits;}
The default constructor.

Operations

The following methods permit access to the traits class’s predicates:

- \texttt{Orientation
d traits.orientation
d object() const}
- \texttt{Contained_in_affine_hull
d traits.contained_in_affine
hull
d object() const}
- \texttt{Construct_flat_orientation
d traits.construct_flat
orientation
d object() const}
- \texttt{In_flat_orientation
d traits.in_flat
orientation
d object() const}
- \texttt{Compare_lexicographically
d traits.compare_lexicographically
d object() const}

Has Models

- \texttt{CGAL::Cartesian
d<FT, Dim, LA>},
- \texttt{CGAL::????<K> (recommended). (The new kernel is currently under development)}

See Also

\texttt{DelaunayTriangulationTraits Triangulation}
DelaunayTriangulationTraits

Definition

The concept DelaunayTriangulationTraits describes the various types and functions that a class has to provide as the first parameter (DCTraits) to the class template `Delaunay_triangulation<DelaunayTriangulationTraits, TriangulationDataStructure>`. It brings the geometric ingredients to the definition of a Delaunay complex, while the combinatorial ingredients are brought by the second template parameter, `TriangulationDataStructure`.

Refines

`TriangulationTraits`.

Types

DelaunayTriangulationTraits:: `Side_of_oriented_sphere_d`

A predicate object that must provide the templated operator

```cpp
template<typename ForwardIterator> Oriented_side operator()(ForwardIterator start, ForwardIterator end, const Point_d & p).
```

The operator returns `ON_POSITIVE_SIDE`, `ON_NEGATIVE_SIDE` or `ON_ORIENTED_BOUNDARY` depending of the side of the query point `p` with respect to the sphere circumscribing the simplex defined by the points in range `{start,end}`. If the simplex is positively oriented, then the positive side of sphere corresponds geometrically to its bounded side.

**Precondition:** `std::distance(start,end)=D+1`, where `Point_dimension_d(*it)` is `D` for all `it` in `{start,end}`. `Point_dimension_d(p)` is also `D`. The points in range `{start,end}` must be affinely independent, i.e., the simplex must not be flat.

DelaunayTriangulationTraits:: `In_flat_side_of_oriented_sphere_d`

A predicate object that must provide the templated operator

```cpp
template<typename ForwardIterator> Oriented_side operator()(Flat_orientation_d orient, ForwardIterator start, ForwardIterator end, const Point_d & p).
```

The operator returns `ON_POSITIVE_SIDE`, `ON_NEGATIVE_SIDE` or `ON_ORIENTED_BOUNDARY` depending of the side of the query point `p` with respect to the sphere circumscribing the simplex defined by the points in range `{start,end}`. If the simplex is positively oriented according to `orient`, then the positive side of sphere corresponds geometrically to its bounded side. The points in range `{start,end}` and `p` are supposed to belong to the lower dimensional flat whose orientation is given by `orient`.

**Precondition:** `std::distance(start,end)=k+1` where `k` is the number of points used to construct `orient`. `Point_dimension_d(*it)` is `D` for all `it` in `{start,end}`. `Point_dimension_d(p)` is also `D`. The points in range `{start,end}` must be affinely independent, i.e., the simplex must not be flat.
Creation

DelaunayTriangulationTraits traits;

The default constructor.

Operations

The following methods permit access to the traits class’s predicates:

\[ Side\_of\_oriented\_sphere\_d \quad traits.side\_of\_oriented\_sphere\_d\_object()\text{ const} \]
\[ In\_flat\_side\_of\_oriented\_sphere\_d \quad traits.in\_flat\_side\_of\_oriented\_sphere\_d\_object()\text{ const} \]

Has Models

\[ \text{CGAL::Cartesian}_d\lt FT, \text{Dim, LA}\gt , \]
\[ \text{CGAL::Simple cartesian}_d\lt FT, \text{Dim, LA}\gt , \]
\[ \text{CGAL::New kernel}_d \text{ (recommended when available)} \]

See Also

TriangulationTraits
DelaunayTriangulation
TriangulationVertex

Definition

The concept TriangulationVertex describes the requirements on the type used by the class Triangulation\(<\text{TriangulationTraits},\text{TriangulationDataStructure}\>, and its derived classes, to represent a vertex.

Refines

TriangulationDSVertex

We only list below the additional specific requirements of TriangulationVertex.

Compared to TriangulationDSVertex, the main difference is the addition of an association of the vertex into a geometric point.

Has Models

\text{CGAL::Triangulation\_vertex}\langle\text{TriangulationTraits},\text{Data},\text{TriangulationDSVertex}\rangle

Types

\text{TriangulationVertex::Point}

The type of the point stored in the vertex. It must be the same as the point type \text{TriangulationTraits::Point} (or its refined concepts) when the \text{TriangulationVertex} is used in the class \text{Triangulation\langle\text{TriangulationTraits}, \text{TriangulationDataStructure}\rangle} (or its derived classes).

Creation

\text{TriangulationVertex v( Full\_cell\_handle c, Point p);} \quad \text{Constructs a vertex with incident full cell c. The vertex is embedded at point p.}

\text{TriangulationVertex v( Point p);} \quad \text{Same as above, but without incident full cell.}

\text{TriangulationVertex v;} \quad \text{Same as above, but with a default-constructed Point.}

Operations

\text{void v.set\_point( Point p)} \quad \text{The parameter p becomes the new geometrical position of the vertex.}

\text{Point v.point() const} \quad \text{Returns the vertex’s position.}
**Input/Output**

These operators can be used directly and are called by the I/O operator of class *Triangulation*.

\[ \text{istream } \& \quad \text{istream } \& \text{ is } >> \quad \& \ v \]

Inputs the non-combinatorial information given by the vertex, i.e., the point and other possible information.

\[ \text{ostream } \& \quad \text{ostream } \& \text{ os } << \quad v \]

Outputs the non-combinatorial information given by the vertex, i.e., the point and other possible information.

**See Also**

*Triangulation\_\_vertex\_<Triangulation\_\_Traits, Data, Triangulation\_\_DS\_\_Vertex>*

*Triangulation\_\_Full\_\_Cell*

*Triangulation\_\_<Triangulation\_\_Traits, Triangulation\_\_Data\_\_Structure>*
TriangulationFullCell

Definition

The concept TriangulationFullCell describes the requirements on the type used by the class Triangulation<
TriangulationTraits, TriangulationDataStructure>, and its derived classes, to represent a full cell.

Refines

TriangulationDSFullCell

We only list below the additional specific requirements of TriangulationFullCell.

Has Models

CGAL::Triangulation_full_cell<TriangulationTraits, TriangulationDSFullCell>

Input/Output

These operators can be used directly and are called by the I/O operator of class Triangulation.

\[
\begin{align*}
\text{istream } & \triangleleft \\ & \text{is } >> \triangleleft c \\
\text{ostream } & \triangleleft \\ & \text{os } << c
\end{align*}
\]

Inputs additional information stored in the full cell.

Outputs additional information stored in the full cell.

See Also

Triangulation_full_cell<TriangulationTraits, TriangulationDSFullCell>
TriangulationVertex
Triangulation<TriangulationTraits, TriangulationDataStructure>
CGAL::Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell>

Definition

This class is used for storing the combinatorial information of a triangulation of dimension \( k \leq d \).

#include <CGAL/Triangulation_data_structure.h>

Parameters

Dimensionality can be either:

- CGAL::Dimension_tag\(<d>\) for some integer \( d \). This indicates that the triangulation data structure can store simplices (full cells) of dimension at most \( d \). The maximum dimension \( d \) is known by the compiler, which triggers some optimizations. Or

- CGAL::Dynamic_dimension_tag. In this case, the maximum dimension of the simplices (full cells) is passed as an integer argument to an instance constructor (see TriangulationDataStructure).

TriangulationDSVertex is the class to be used as the base Vertex type in the triangulation data structure. It must be a model of the concept TriangulationDSVertex. The class template Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell> can be defined by specifying only the first parameter. It also accepts the tag CGAL::Default as second parameter. In both cases, TriangulationDSVertex defaults to CGAL::Triangulation_ds_vertex<>.

TriangulationDSFullCell is the class to be used as the base Full_cell type in the triangulation data structure. It must be a model of the concept TriangulationDSFullCell. The class template Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell> accepts that no third parameter be specified. It also accepts the tag CGAL::Default as third parameter. In both cases, TriangulationDSFullCell defaults to CGAL::Triangulation_ds_full_cell<>.

Is Model for the Concepts

TriangulationDataStructure.

In addition, the class Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell> provides the following types and methods:

Creation

Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell> t(Triangulation_data_structure t2)

The copy constructor. Creates a copy of the Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell> t2 passed as argument. All vertices and full cells are duplicated.
Validity check

The \texttt{is\_valid} method is only minimally defined in the \texttt{TriangulationDataStructure} concept, so that we document it more precisely here, for the model \texttt{Triangulation\_data\_structure\langle Dimensionality, TriangulationDSVertex, TriangulationDSFullCell\rangle}:

\begin{verbatim}
bool t.is_valid( bool verbose = true) const

    Implements the validity checks required by the concept TriangulationDataStructure.
    Note that passing all these tests does not guaranty that we have a triangulation (abstract pure simplicial complex).
\end{verbatim}

Types

\texttt{Triangulation\_data\_structure\langle Dimensionality, TriangulationDSVertex, TriangulationDSFullCell\rangle:: Full\_cell\_data}

A data member of type \texttt{Full\_cell\_data} is stored in every full cell (models of the concept \texttt{TriangulationDSFullCell}). It is used to mark some full cells, during modifications of the triangulation data structure.

Vertex insertion

\begin{verbatim}
template\}< OutputIterator >
Full\_cell\_handle t.insert\_in\_tagged\_hole( Vertex\_handle v, Facet f, OutputIterator new\_full\_cells)

    A set \( C \) of full cells satisfying the same condition as in method \texttt{Triangulation\_data\_structure\langle Dimensionality, TriangulationDSVertex, TriangulationDSFullCell\rangle::insert\_in\_hole()} is assumed to be marked. This method creates new full cells from vertex \( v \) to the boundary of \( C \). The boundary is recognized by checking the mark of the full cells. This method is used by \texttt{Triangulation\_data\_structure\langle Dimensionality, TriangulationDSVertex, TriangulationDSFullCell\rangle::insert\_in\_hole()}. 
    Precondition: same as \texttt{Triangulation\_data\_structure\langle Dimensionality, TriangulationDSVertex, TriangulationDSFullCell\rangle::insert\_in\_hole()}
\end{verbatim}
See Also

Triangulation_ds_vertex
Triangulation_ds_full_cell
Triangulation
CGAL::Triangulation_ds_vertex<TriangulationDataStructure>

Definition

The class \texttt{Triangulation_ds_vertex\langle TriangulationDataStructure\rangle} serves as the default vertex template parameter in the class \texttt{Triangulation_data_structure\langle Dimensionality, TriangulationDSVertex, TriangulationDSFullCell\rangle}.

This class does not contain any geometric information but only combinatorial (adjacency) information. Thus, if the \texttt{Triangulation_data_structure} is used as a parameter of a (embedded) \texttt{Triangulation}, then its vertex template parameter has to fulfill additional geometric requirements, i.e., it has to be a model of the refined concept \texttt{TriangulationVertex}.

This class can be used directly or can serve as a base to derive other classes with some additional attributes tuned for a specific application (a color for example).

\#include \texttt{CGAL/Triangulation_ds_vertex.h}

Parameters

The template parameter \texttt{TriangulationDataStructure} must be a model of the \texttt{TriangulationDataStructure} concept.

Is Model for the Concepts

\texttt{TriangulationDSVertex}

\begin{verbatim}
  debugging support
\end{verbatim}

Validity check

The \texttt{is_valid} method is only minimally defined in the \texttt{TriangulationDSVertex} concept, so that we document it more precisely here, for the model \texttt{Triangulation_ds_vertex\langle TriangulationDataStructure\rangle}:

\begin{verbatim}
  bool v.is_valid( bool verbose=false) const

  Implements the validity checks required by the concept \texttt{TriangulationDSVertex}. Does not implement additional checks.
\end{verbatim}

\begin{verbatim}
  debugging support
\end{verbatim}

\begin{verbatim}
  advanced
\end{verbatim}

Rebind mechanism

In case of derivation from that class, the nested class \texttt{Rebind\_TDS} need to be provided in the derived class.

\begin{verbatim}
  advanced
\end{verbatim}
See Also

* Triangulation_ds_full_cell<TriangulationDataStructure, TDSFullCellStoragePolicy>*
* Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell>*>
CGAL::Triangulation_ds_full_cell<TriangulationDataStructure, TDSFullCellStoragePolicy>

Definition

The class `Triangulation_ds_full_cell<TriangulationDataStructure, TDSFullCellStoragePolicy>` serves as the default full cell template parameter in the class `Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell>.

This class does not provide any geometric capabilities but only combinatorial (adjacency) information. Thus, if the `Triangulation_data_structure` is used as a parameter of an (embedded) `Triangulation`, then its full cell template parameter has to fulfill additional geometric requirements, i.e. it has to be a model of the refined concept `TriangulationFullCell`.

This class can be used directly or can serve as a base to derive other classes with some additional attributes tuned for a specific application.

```
#include <CGAL/Triangulation_ds_full_cell.h>
```

Parameters

The first template parameter, `TriangulationDataStructure`, must be a model of the `TriangulationDataStructure` concept.

The second parameter, `TDSFullCellStoragePolicy`, indicates whether or not the full cell should additionally store the mirror indices (the indices of the mirror vertices). This improves speed a little, but takes more space:

The class template `Triangulation_ds_full_cell<TriangulationDataStructure, TDSFullCellStoragePolicy>` accepts that no second parameter be specified. It also accepts the tag `CGAL::Default` as second parameter. Both cases are equivalent to setting `TDSFullCellStoragePolicy` to `CGAL::TDS_full_cell_default_storage_policy`.

When the second parameter is specified, its possible “values” are:

- `CGAL::Default`, which is the default value. In that case, the policy `CGAL::TDS_full_cell_default_storage_policy` is used.
- `CGAL::TDS_full_cell_default_storage_policy`. In that case, the mirror indices are not stored.
- `CGAL::TDS_full_cell_mirror_storage_policy`. In that case, the mirror indices are stored.

See the user manual for how to choose the second option.

Is Model for the Concepts

`TriangulationDSFullCell`
Validity check

The `is_valid` method is only minimally defined in the `TriangulationDSFullCell` concept, so that we document it more precisely here, for the model `Triangulation_ds_full_cell`:

```cpp
bool c.is_valid( bool verbose=false) const
```

Implements the validity checks required by the concept `TriangulationDSFullCell`. In addition, it is checked that there is no NULL handle to vertices in the middle of non-NUL ones, that is, that the internal memory layout is not corrupted.

<table>
<thead>
<tr>
<th>debugging support</th>
</tr>
</thead>
</table>

| advanced |

Rebind mechanism

In case of derivation from that class, the nested class `Rebind_TDS` need to be provided in the derived class.

| advanced |

See Also

- `Triangulation_ds_vertex`<`TriangulationDataStructure`>
- `Triangulation_data_structure`<`Dimensionality, TriangulationDSVertex, TriangulationDSFullCell`>
CGAL::Triangulation_face<TriangulationDataStructure>

**Definition**

A `Triangulation_face<TriangulationDataStructure>` is a model of the concept `TriangulationDSFace`.

**Parameters**

Parameter `TriangulationDataStructure` must be a model of the concept `TriangulationDataStructure`. Actually, `Triangulation_face<TriangulationDataStructure>` needs only that this concept defines the types `Full_cell_handle`, `Vertex_handle`, and `Maximal_dimension`.

**Is Model for the Concepts**

`TriangulationDSFace`

**See Also**

`TriangulationDSFace`
`TriangulationDataStructure`
CGAL::Triangulation<TriangulationTraits, TriangulationDataStructure>

Definition

The class Triangulation<TriangulationTraits, TriangulationDataStructure> is used to store and query the full cells and vertices of a triangulation in dimension \( d \). A special vertex, named an infinite vertex, is used to triangulate the outside of the convex hull of the points in so-called infinite cells.

```cpp
#include <CGAL/Triangulation.h>
```

Parameters

\( \text{TriangulationTraits} \) is the geometric traits class that provides the geometric types and predicates needed by triangulations. \( \text{TriangulationTraits} \) must be a model of the concept \( \text{TriangulationTraits} \).

\( \text{TriangulationDataStructure} \) is the class used to store the underlying triangulation data structure. \( \text{TriangulationDataStructure} \) must be a model of the concept \( \text{TriangulationDataStructure} \). The class template \( \text{Triangulation<TriangulationTraits, TriangulationDataStructure>} \) can be defined by specifying only the first parameter, or by using the tag \( \text{CGAL::Default} \) as the second parameter. In both cases, \( \text{TriangulationDataStructure} \) defaults to \( \text{Triangulation\_data\_structure\_<Maximal\_dimension\_<\_<\_;} \), \( \text{Triangulation\_vertex\_<\_<\_;} \), \( \text{Triangulation\_full\_cell\_<\_<\_;} \).

Types

```cpp
typedef TriangulationTraits Geom_traits; // Type for the model of the TriangulationTraits concept.

typedef TriangulationTraits::Point_d Point; // A point in Euclidean space.

typedef TriangulationTraits::Dimension Maximal_dimension; // This indicates whether the dimension of the underlying space is static (Maximal_dimension=CGAL::Dimension_tag<int dim>) or dynamic (Maximal_dimension=CGAL::Dynamic_dimension_tag). In the latter case, the dim parameter passed to the class's constructor is used.

typedef TriangulationDataStructure Triangulation_ds; // The second template parameter.

typedef TriangulationDataStructure::Vertex Vertex; // A model of the concept TriangulationVertex.

typedef TriangulationDataStructure::Full_cell Full_cell; // A model of the concept TriangulationFullCell.

typedef TriangulationDataStructure::Facet Facet; // The facet class
```
typedef TriangulationDataStructure::Face Face;  
A model of the concept TriangulationDSFace.

The vertices and full cells of triangulations are accessed through handles, iterators and circulators. A handle is a model of the Handle concept, and supports the two dereference operators \texttt{operator\^{*}} and \texttt{operator\texttt{->}}. A circulator is a model of the concept Circulator. Iterators and circulators are bidirectional and non-mutable.

Iterators and circulators are convertible to the corresponding handles, thus the user can pass them directly as arguments to the functions.

typedef TriangulationDataStructure::Vertex_handle Vertex_handle;  
handle to a a vertex

typedef TriangulationDataStructure::Vertex_iterator Vertex_iterator;  
iterator over all vertices

typedef TriangulationDataStructure::Full_cell_handle Full_cell_handle;  
handle to a full cell

typedef TriangulationDataStructure::Full_cell_iterator Full_cell_iterator;  
iterator over all full cells

typedef TriangulationDataStructure::Facet_iterator Facet_iterator;  
iterator over all facets

typedef TriangulationDataStructure::size_type size_type;  
Size type (an unsigned integral type).

typedef TriangulationDataStructure::difference_type difference_type;  
Difference type (a signed integral type).

The \texttt{Triangulation\langle\texttt{TriangulationTraits, TriangulationDataStructure}\rangle} class also defines the following enum type to specify which case occurs when locating a point in the triangulation:

define \texttt{Locate\_type} { \texttt{ON\_VERTEX}, \texttt{IN\_FACE}, \texttt{IN\_FACET}, \texttt{IN\_FULL\_CELL}, \texttt{OUTSIDE\_CONVEX\_HULL}, \texttt{OUTSIDE\_AFFINE\_HULL} }

See \texttt{CGAL::Triangulation::Locate\_type}

\textbf{Creation}

\texttt{Triangulation\langle\texttt{TriangulationTraits, TriangulationDataStructure}\rangle \ tr( \ const \ int \ dim, \newline \quad \quad Geom\_traits \ \textit{gt} = \ Geom\_traits())}

Instantiates a triangulation with one vertex (the vertex at infinity). See the description of the nested type \texttt{Maximal\_dimension} above for an explanation of the use of the parameter \texttt{dim}. The triangulation stores a copy of the geometric traits \textit{gt}.

\texttt{Triangulation\langle\texttt{TriangulationTraits, TriangulationDataStructure}\rangle \ tr( \ \texttt{Triangulation} \ \textit{t2});}

The copy constructor.
**Access functions**

<table>
<thead>
<tr>
<th>Class</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangulation_ds</td>
<td><code>tr.tds()</code> const</td>
<td>Returns a const reference to the underlying triangulation data structure.</td>
</tr>
<tr>
<td></td>
<td><strong><strong><strong>advanced</strong></strong></strong></td>
<td></td>
</tr>
<tr>
<td>Triangulation_ds</td>
<td><code>tr.tds()</code></td>
<td>Returns a non-const reference to the underlying triangulation data structure.</td>
</tr>
<tr>
<td></td>
<td><strong><strong><strong>advanced</strong></strong></strong></td>
<td></td>
</tr>
<tr>
<td>Geom_traits</td>
<td><code>tr.geom_traits()</code> const</td>
<td>Returns a const reference to the geometric traits instance.</td>
</tr>
<tr>
<td>int</td>
<td><code>tr.maximal_dimension()</code> const</td>
<td>Returns the dimension of the embedding Euclidean space.</td>
</tr>
<tr>
<td>int</td>
<td><code>tr.current_dimension()</code> const</td>
<td>Returns the dimension of the triangulation (as an embedded manifold).</td>
</tr>
<tr>
<td>bool</td>
<td><code>tr.empty()</code> const</td>
<td>Returns <code>true</code> if the triangulation has no finite vertex. Returns <code>false</code> otherwise.</td>
</tr>
<tr>
<td>size_type</td>
<td><code>tr.number_of_vertices()</code> const</td>
<td>Returns the number of finite vertices in the triangulation.</td>
</tr>
<tr>
<td>size_type</td>
<td><code>tr.number_of_full_cells()</code> const</td>
<td>Returns the number of full cells of maximal dimension in the triangulation (full cells incident to the vertex at infinity are counted).</td>
</tr>
<tr>
<td>Vertex_handle</td>
<td><code>tr.infinite_vertex()</code> const</td>
<td>Returns a handle to the vertex at infinity.</td>
</tr>
<tr>
<td>Full_cell_handle</td>
<td><code>tr.infinite_full_cell()</code> const</td>
<td>Returns a handle to some full cell incident to the vertex at infinity.</td>
</tr>
</tbody>
</table>

**Non-constant-time access functions**

<table>
<thead>
<tr>
<th>Class</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_type</td>
<td><code>tr.number_of_finite_full_cells()</code> const</td>
<td>Returns the number of full cells of maximal dimension that are not incident to the vertex at infinity.</td>
</tr>
</tbody>
</table>

**Tests for finite and infinite elements**

<table>
<thead>
<tr>
<th>Class</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td><code>tr.is_infinite( const Vertex_handle v) const</code></td>
<td>Returns <code>true</code> if and only if the vertex <code>v</code> is the infinite vertex.</td>
</tr>
</tbody>
</table>
bool tr.is_infinite( const Full_cell_handle c) const
Returns true if and only if c is incident to the infinite vertex.

bool tr.is_infinite( Facet ft) const
Returns true if and only if facet ft is incident to the infinite vertex.

bool tr.is_infinite( Face f) const
Returns true if and only if the face f is incident to the infinite vertex.

Faces and Facets

Full_cell_handle tr.full_cell( Facet f) const
Returns a full cell containing the facet f

int tr.index_of_covertex( Facet f) const
Returns the index of the vertex of the full cell c=tr.full_cell(f) which does not belong to c.

Triangulation traversal

Vertex_iterator tr.vertices_begin() The first vertex of tr.
Vertex_iterator tr.vertices_end() The beyond vertex of tr.

Finite_vertex_iterator tr.finite_vertices_begin() The first finite vertex of tr.
Finite_vertex_iterator tr.finite_vertices_end() The beyond finite vertex of tr.

Full_cell_iterator tr.full_cells_begin() The first full cell of tr.
Full_cell_iterator tr.full_cells_end() The beyond full cell of tr.

Finite_full_cell_iterator tr.finite_full_cells_begin() The first finite full cell of tr.
Finite_full_cell_iterator tr.finite_full_cells_end() The beyond finite full cell of tr.

Facet_iterator tr.facets_begin() Iterator to the first facet of the triangulation.
Facet_iterator tr.facets_end() Iterator to the beyond facet of the triangulation.

Finite_facet_iterator tr.finite_facets_begin() Iterator to the first finite facet of the triangulation.
Finite_facet_iterator tr.finite_facets_end() Iterator to the beyond finite facet of the triangulation.

Point location

The class Triangulation<TriangulationTraits, TriangulationDataStructure> provides methods to locate a query point with respect to the triangulation:
The optional argument \textit{hint} is used as a starting place for the search. If the \textit{query} point lies outside the affine hull of the points (which can happen when \texttt{tr.current\_dimension() < tr.maximal\_dimension()}) or if there is no finite vertex yet in the triangulation, then \texttt{locate} returns a default constructed \texttt{Full\_cell\_handle()}. If the point \textit{query} lies in the interior of a bounded (finite) full cell of \textit{tr}, the latter full cell is returned. If \textit{query} lies on the boundary of some finite full cells, one of them is returned. Let \( d = \texttt{tr.current\_dimension()}. \) If the point \textit{query} lies outside the convex hull of the points, an infinite full cell with vertices \( \{p_1, p_2, \ldots, p_d, \infty\} \) is returned such that the full cell \((p_1, p_2, \ldots, p_d, \textit{query})\) is positively oriented (the rest of the triangulation lies on the other side of facet \((p_1, p_2, \ldots, p_d)\)).

Same as above but \textit{hint} is a vertex and not a full cell.

Let \( d = \texttt{tr.current\_dimension()}. \) If the point \textit{query} lies outside the convex hull of the points, an infinite full cell with vertices \( \{p_1, p_2, \ldots, p_d, \infty\} \) is returned such that the full cell \((p_1, p_2, \ldots, p_d, \textit{query})\) is positively oriented (the rest of the triangulation lies on the other side of facet \((p_1, p_2, \ldots, p_d)\)).
\textit{Full\_cell\_handle hint = Full\_cell\_handle()}

The optional argument \textit{hint} is used as a starting place for the search. If the \textit{query} point lies outside the affine hull of the points (which can happen when \textit{tr.current\_dimension()} < \textit{tr.maximal\_dimension()}) or if there is no finite vertex yet in the triangulation, then \textit{loc\_type} is set to \textit{OUTSIDE\_AFFINE\_HULL}, and \textit{locate} returns \textit{Full\_cell\_handle()}.

If the \textit{query} point lies inside the affine hull of the points, a \textit{k}-face that contains \textit{query} in its relative interior is returned. (If the \textit{k}-face is finite, it is unique.)

\begin{itemize}
  \item \textbf{\textit{k} = 0} \textit{loc\_type} is set to \textit{ON\_VERTEX}, \textit{f} is set to the vertex \textit{v} the \textit{query} lies on and a full cell having \textit{v} as a vertex is returned.
  \item \textbf{0 < \textit{k} < \textit{c.current\_dimension()}-1} \textit{loc\_type} is set to \textit{IN\_FACE}, \textit{f} is set to the unique finite face containing the \textit{query} point. A full cell having \textit{f} on its boundary is returned.
  \item \textbf{\textit{k} = \textit{c.current\_dimension()}-1} \textit{loc\_type} is set to \textit{IN\_FACET}, \textit{ft} is set to one of the two representation of the finite facet containing the \textit{query} point. The full cell of \textit{ft} is returned.
  \item \textbf{\textit{k} = \textit{c.current\_dimension()}} If the \textit{query} point lies outside the convex hull of the points in the triangulation, then \textit{loc\_type} is set to \textit{OUTSIDE\_CONVEX\_HULL} and a full cell is returned as in the \textit{locate} method above. If the \textit{query} point lies inside the convex hull of the points in the triangulation, then \textit{loc\_type} is set to \textit{IN\_FULL\_CELL} and the unique full cell containing the \textit{query} point is returned.
\end{itemize}

\textit{Full\_cell\_handle} \textit{tr.locate( Point query, Locate\_type & loc\_type, Face & f, Vertex\_handle hint) const}

Same as above but \textit{hint}, the starting place for the search, is a vertex. The parameter \textit{hint} is ignored if it is a default constructed \textit{Vertex\_handle()}.  

\begin{flushleft}
\textbf{Removal}
\end{flushleft}

\begin{center}
\textit{advanced}
\end{center}
**Vertex_handle**

```
tr.collapse_face( Point p, Face f)
```

Contracts the Face f to a single vertex at position p. Returns a handle to that vertex.

*Precondition:* The boundary of the union of full cells incident to f must be a triangulation of a sphere of dimension `tr.current_dimension()`.

---

### advanced

**Point insertion**

The class `Triangulation<TriangulationTraits, TriangulationDataStructure>` provides functions to insert a given point in the triangulation:

```
template< typename ForwardIterator >
size_type   tr.insert( ForwardIterator s, ForwardIterator e)
```

Inserts the points found in range `{s,e}` in the triangulation. Returns the number of vertices actually inserted. (If several vertices share the same position in space, only the first insertion is counted.)

```
Vertex_handle tr.insert( const Point p, Full_cell_handle hint = Full_cell_handle())
```

Inserts point p in the triangulation. Returns a `Vertex_handle` to the vertex of the triangulation with position p. Prior to the actual insertion, p is located in the triangulation; `hint` is used as a starting place for locating p.

```
Vertex_handle tr.insert( const Point p, Vertex_handle hint)
```

Same as above but uses a vertex `hint` as the starting place for the search.

---

**Vertex_handle**

```
tr.insert( const Point p,
          Locate_type loc_type,
          Face & f,
          Facet & ft,
          )
```

...
Full_cell_handle c)

Inserts point \( p \) into the triangulation and returns a handle to the Vertex at that position. The action taken depends on the value of \( \text{loc.type} \):

ON_VERTEX The point of the Vertex described by \( f \) is set to \( p \).

IN_FACE The point \( p \) is inserted in the Face \( f \).

IN_FACET The point \( p \) is inserted in the Facet \( f_t \).

Anything else The point \( p \) is inserted in the triangulation according to the value of \( \text{loc.type} \), using the full cell \( c \).

This method is used internally by the other insert() methods.

\[
\text{template} \ < \ \text{typename ForwardIterator, typename OutputIterator} > \\
\text{Vertex.handle} \quad \text{tr.insert_in_hole( Point p,} \\
\quad \text{ForwardIterator s,} \\
\quad \text{ForwardIterator e,} \\
\quad \text{Facet ft,} \\
\quad \text{OutputIterator out)}
\]

The full cells in the range \( C = [s, e) \) are removed, thus forming a hole. A Vertex is inserted at position \( p \) and connected to the boundary of the hole in order to “fill it”. A Vertex handle to the new Vertex is returned. The facet \( f_t \) must lie on the boundary of \( C \) and its defining full cell, \( \text{tr.full.cell}(f_t) \) must lie inside \( C \). Handles to the newly created full cells are output in the \( \text{out} \) output iterator.

Precondition: \( C \) must be a (geometric) ball, must contain \( p \) in its interior and not contain any vertex all of whose incident full cells are in \( C \). (This implies that \( \text{tr.current_dimension}() \geq 2 \) if \( |C| > 1 \).)

The boundary of \( C \) must be a triangulation of the sphere \( S^{k-1} \).

\[
\text{template} \ < \ \text{typename ForwardIterator} > \\
\text{Vertex.handle} \quad \text{tr.insert_in_hole( Point p, ForwardIterator s, ForwardIterator e, Facet ft)}
\]

Same as above, but the newly created full cells are not retrieved.

\[
\text{Vertex.handle} \quad \text{tr.insert_in_face( Point p, Face f)}
\]

Inserts point \( p \) in the triangulation.

Precondition: \( p \) must lie in the relative interior of \( f \).
**Vertex_handle**

```
tr.insert_in_facet( Point p, Facet ft)
```

Inserts point \( p \) in the triangulation.  
**Precondition:** \( p \) must lie in the relative interior of \( ft \).

**Vertex_handle**

```
tr.insert_in_full_cell( Point p, Full_cell_handle c)
```

Inserts point \( p \) in the triangulation.  
**Precondition:** \( p \) must lie in the interior of \( c \).

**Vertex_handle**

```
tr.insert_outside_convex_hull( Point, Full_cell_handle c)
```

Inserts point \( p \) in the triangulation.  
**Precondition:** \( p \) must lie outside the convex hull of \( tr \). The half-space defined by the infinite full cell \( c \) must contain \( p \).

**Vertex_handle**

```
tr.insert_outside_affine_hull( Point)
```

Inserts point \( p \) in the triangulation.  
**Precondition:** \( p \) must lie outside the affine hull of \( tr \).

---

**advanced debugging support**

---

**Validity check**

**bool**

```
tr.is_valid( bool verbose=false) const
```

Partially checks whether \( tr \) is a triangulation. This function returns \texttt{true} if the combinatorial triangulation data structure’s \texttt{is_valid()} test returns \texttt{true} and if some geometric tests are passed with success: It is checked that the orientation of each finite full cell is positive and that the orientation of each infinite full cell is consistent with their finite adjacent full cells. The \texttt{verbose} parameter is not used.

**bool**

```
tr.are_incident_full_cells_valid( Vertex_const_handle v, bool verbose = false) const
```

Returns \texttt{true} if and only if all finite full cells incident to \( v \) have positive orientation. The \texttt{verbose} parameter is not used.

---

**debugging support**

---

66
Input/Output

`istream & is >> Triangulation & t`

Reads the underlying combinatorial triangulation from `is` by calling the corresponding input operator of the triangulation data structure class (note that the infinite vertex is numbered 0), and the non-combinatorial information by calling the corresponding input operators of the vertex and the full cell classes (such as point coordinates), which are provided by overloading the stream operators of the vertex and full cell types. Assigns the resulting triangulation to `t`.

`ostream & os << Triangulation t`

Writes the triangulation `t` into `os`.

The information in the `istream` is: the current dimension, the number of finite vertices, the non-combinatorial information about vertices (point, etc.), the number of full cells, the indices of the vertices of each full cell, plus the non-combinatorial information about each full cell, then the indices of the neighbors of each full cell, where the index corresponds to the preceding list of full cells.

See Also

`Triangulation_data_structure<Dimensionality, TriangulationDSVertex, TriangulationDSFullCell>, Delaunay_triangulation<DelaunayTriangulationTraits, TriangulationDataStructure>,`
Definition

The class Delaunay triangulation\<DelaunayTriangulationTraits, TriangulationDataStructure> is used to main-
tain the full cells and vertices of a Delaunay triangulation in \(\mathbb{R}^D\). It permits point insertion and removal. The
dimension \(D\) should be kept reasonably small, see the performance section in the user manual for what reason-
able means.

#include \textless CGAL/Delaunay triangulation.h \textgreater

Parameters

\textit{DelaunayTriangulationTraits} is the geometric traits class that provides the geometric types and predicates
needed by Delaunay triangulations. \textit{DelaunayTriangulationTraits} must be a model of the concept \textit{DelaunayTri-
angulationTraits}.

\textit{TriangulationDataStructure} is the class used to store the underlying triangulation data structure. \textit{Triangula-
tionDataStructure} must be a model of the concept \textit{TriangulationDataStructure}. The class template Delaunay_
triangulation\<DelaunayTriangulationTraits, TriangulationDataStructure> can be defined by specifying only
the first parameter, or by using the tag CGAL::Default as the second parameter. In both cases, \textit{TriangulationDataStructure}
defaults to \textit{Triangulation\_data\_structure}\<Maximal\_dimension\<\textit{TriangulationTraits}::\textit{Point}\_d\>::\textit{type}, \textit{Triangulation\_vertex}\<\textit{TriangulationTraits}>, \textit{Triangulation\_full\_cell}\<\textit{TriangulationTraits}>>.

Inherits From

\textit{Triangulation\<DelaunayTriangulationTraits, TriangulationDataStructure>}. The class Delaunay triangulation\<DelaunayTriangulationTraits, TriangulationDataStructure> inherits all the
types defined in the base class \textit{Triangulation\<DelaunayTriangulationTraits, TriangulationDataStructure>}. Ad-
ditionally, it defines or overloads the following methods:

Creation

\textit{Delaunay triangulation\<DelaunayTriangulationTraits, TriangulationDataStructure> \textit{dt( const int dim,}

\begin{verbatim}
const Geom_traits gt
\end{verbatim}

\textit{= Geom\_traits()})

Instantiates a Delaunay triangulation with one vertex (the
vertex at infinity). See the description of the inherited nested
type \textit{Triangulation\<DelaunayTriangulationTraits, TriangulationDataStructure>}::\textit{Maximal}\_\textit{dimension} for an explana-
tion of the use of the parameter \textit{dim}. The complex stores
a copy of the geometric traits \textit{gt}.
Point removal

\[ \text{Full\_cell\_handle } dt.remove( \text{Vertex\_handle } v) \]

Remove the vertex \( v \) from the Delaunay triangulation. If the current dimension of the triangulation has not changed after the removal, then the returned full cell \( c \) geometrically contains the removed vertex \( v \) (\( c \) can be finite or infinite). Otherwise, the default-constructed \text{Full\_cell\_handle} is returned.

Precondition: \( v \) is a vertex of the triangulation, different from the infinite_vertex().

\[ \begin{align*}
&\text{template< typename ForwardIterator >} \\
&\quad \text{void } dt.remove( \text{ForwardIterator } start, \text{ForwardIterator } end)
\end{align*} \]

Remove the points or the vertices (through their \text{Vertex\_handle}) in the range \([start, end)\). *start must be of type \text{Vertex\_handle}.

Point insertion

\[ \begin{align*}
&\text{template< typename ForwardIterator >} \\
&\quad \text{size\_type } dt.insert( \text{ForwardIterator } s, \text{ForwardIterator } e)
\end{align*} \]

Inserts the points found in range \([s,e)\) in the Delaunay triangulation and ensures that the empty-ball property is preserved. Returns the number of vertices actually inserted. (If more than one vertex share the same position in space, only one insertion is counted.)

\[ \text{Vertex\_handle } dt.insert( \text{Point } p, \text{Full\_cell\_handle } hint = \text{Full\_cell\_handle}()) \]

Inserts point \( p \) in the Delaunay triangulation and ensures that the empty-ball property is preserved. Returns a \text{Vertex\_handle} to the vertex of the triangulation with position \( p \). Prior to the actual insertion, \( p \) is located in the triangulation; \( hint \) is used as a starting place for locating \( p \).

\[ \text{Vertex\_handle } dt.insert( \text{Point } p, \text{Vertex\_handle } hint) \]

Same as above but uses a vertex as starting place for the search.
Vertex
handle dt.insert( Point p, const Locate
_type lt, Face f, Facet ft, const Full
_cell_handle c)

Inserts the point \( p \) in the Delaunay triangulation and ensures that the empty-ball property is preserved. Returns a handle to the (possibly newly created) vertex at that position. The behavior depends on the value of \( lt \):

**OUTSIDE_AFFINE_HULL** Point \( p \) is inserted so as to increase the current dimension of the Delaunay triangulation. The method \( dt.insert\_outside\_affine\_hull() \) is called.

**ON_VERTEX** The position of the vertex \( v \) described by \( f \) is set to \( p \). \( v \) is returned.

Anything else The point \( p \) is inserted. the full cell \( c \) is assumed to be in conflict with \( p \). (Roughly speaking, the method \( dt.insert\_in\_conflicting\_cell() \) is called.)

The parameters \( lt, f, ft \) and \( c \) must be consistent with the localization of point \( p \) in the Delaunay triangulation e.g. by a call to \( c = locate(p, lt, f, ft) \).

Vertex
handle dt.insert\_outside\_affine\_hull( Point p)

Inserts the point \( p \) in the Delaunay triangulation. Returns a handle to the (possibly newly created) vertex at that position. **Precondition:** The point \( p \) must lie outside the affine hull of the Delaunay triangulation. This implies that \( dt.current\_dimension() \) must be less that \( dt.maximal\_dimension() \).

Vertex
handle dt.insert\_in\_conflicting\_cell( Point p, const Full\_cell\_handle c)

Inserts the point \( p \) in the Delaunay triangulation. Returns a handle to the (possibly newly created) vertex at that position. **Precondition:** The point \( p \) must be in conflict with the full cell \( c \).

--- advanced

**Queries**

*template< typename OutputIterator >*

bool dt.is_in_conflict( Point p, Full_cell_const_handle c) const

Returns true if and only if the point \( p \) is in (Delaunay) conflict with full cell \( c \) (i.e., the circumscribing ball of \( c \) contains \( p \) in its interior).
Facet \texttt{dt.compute\_conflict\_zone( Point p, const Full\_cell\_handle c, OutputIterator out) const}

Outputs handles to the full cells in conflict with point \( p \) into the \texttt{OutputIterator out}. The full cell \( c \) is used as a starting point for gathering the full cells in conflict with \( p \). A facet \((cc,i)\) on the boundary of the conflict zone with \( cc \) in conflict is returned.

\textit{Precondition}: \( c \) is in conflict with \( p \).
\textit{dt.current\_dimension()} \( \geq 2 \).

\begin{footnotesize}
\begin{enumerate}
\item \textit{See Also}
\end{enumerate}
\end{footnotesize}

\begin{footnotesize}
\begin{enumerate}
\item \textit{Triangulation\_data\_structure<Dimensionality, Triangulation\_DS\_Vertex, Triangulation\_DS\_Full\_Cell>},
\end{enumerate}
\end{footnotesize}
CGAL::Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>

Definition

The class `Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>` is a model of the concept `TriangulationVertex`. It is used by default for representing vertices in the class `Triangulation<TriangulationTraits, TriangulationDataStructure>`.

A `Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>` stores a point and an incident full cell.

```cpp
#include <CGAL/Triangulation_vertex.h>
```

Parameters

`TriangulationTraits` must be a model of the concept `TriangulationTraits`. It provides geometric types and predicates for use in the `Triangulation<TriangulationTraits, TriangulationDataStructure>` class. It is of interest here for its declaration of the `Point` type.

`Data` is an optional type of data to be stored in the vertex class. The class template `Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>` accepts that no second parameter be specified. In this case, `Data` defaults to `CGAL::No_vertex_data`. `CGAL::No_vertex_data` can be explicitly specified to allow to access the third parameter.

Parameter `TriangulationDSVertex` must be a model of the concept `TriangulationDSVertex`. The class template `Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>` accepts that no third parameter be specified. It also accepts the tag `CGAL::Default` as third parameter. In both cases, `TriangulationDSVertex` defaults to `CGAL::Triangulation_ds_vertex<>`.

Inherits From

`TriangulationDSVertex` (the third template parameter)

Is Model for the Concepts

`TriangulationVertex`

Additionally, the class `Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>` also provides the following type, constructors and methods:

Types

```cpp
typedef Data Data;
```

The type of the additional data stored in the vertex. If you read a `Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>` from a stream (a file) or write a `Triangulation_vertex<TriangulationTraits, Data, TriangulationDSVertex>` to a stream, then streaming operators `<<` and `>>` must be provided for this type.
Creation

\[
\text{template}\langle\text{typename } T\rangle \\
\text{Triangulation\_vertex}\langle\text{TriangulationTraits, Data, TriangulationDSVertex}\rangle \ v(\text{Full\_cell\_handle } c, \text{Point } p, T t); \\
\]

Constructs a vertex with incident full cell \(c\). The vertex is embedded at point \(p\) and the parameter \(t\) is passed to the \(Data\) constructor.

\[
\text{template}\langle\text{typename } T\rangle \\
\text{Triangulation\_vertex}\langle\text{TriangulationTraits, Data, TriangulationDSVertex}\rangle \ v(\text{Point } p, T t); \\
\]

Same as above, but without incident full cell.

\[
\text{Triangulation\_vertex}\langle\text{TriangulationTraits, Data, TriangulationDSVertex}\rangle \ v; \\
\]

Same as above, but with default-constructed \(Point\) and \(Data\).

Data access

\[
\text{Data} \quad v.\text{data()} \text{ const} \\
\text{Data} & \quad v.\text{data()} \\
\]

Returns a const reference to the stored data.

Returns a non-const reference to the stored data.

Input/Output

\[
\text{istream} & \quad \text{istream} \ & \text{is} \ >> \text{Triangulation\_vertex} & \ v \\
\]

Inputs the non-combinatorial information given by the vertex, i.e., the point and other possible information. The data of type \(Data\) is also read.

\[
\text{ostream} & \quad \text{ostream} \ & \text{os} \ << \text{Triangulation\_vertex} \ v \\
\]

Outputs the non-combinatorial information given by the vertex, i.e., the point and other possible information. The data of type \(Data\) is also written.

See Also

\[
\text{Triangulation\_full\_cell}\langle\text{TriangulationTraits, Data, TriangulationDSFullCell}\rangle \\
\text{Triangulation\_data\_structure}\langle\text{Dimensionality, TriangulationDSVertex, TriangulationDSFullCell}\rangle \\
\text{Triangulation}\langle\text{TriangulationTraits, TriangulationDataStructure}\rangle \\
\text{Delaunay\_triangulation}\langle\text{DelaunayTriangulationTraits, TriangulationDataStructure}\rangle \\
\]

73
The class `CGAL::Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell>` is a model of the concept `TriangulationFullCell`. It is used by default for representing full cells in the class `CGAL::Triangulation<TriangulationTraits, TriangulationDataStructure>`. A `Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell>` stores handles to the vertices of the cell as well as handles to its adjacent cells.

#include `<CGAL/Triangulation_full_cell.h>`

**Parameters**

`TriangulationTraits` must be a model of the concept `TriangulationTraits`. It provides geometric types and predicates for use in the `CGAL::Triangulation<TriangulationTraits, TriangulationDataStructure>` class.

`Data` is an optional type of data to be stored in the full cell class. The class template `CGAL::Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell>` accepts that no second parameter be specified. In this case, `Data` defaults to `CGAL::No_full_cell_data`. `CGAL::No_full_cell_data` can explicitly be specified to access the third parameter.

Parameter `TriangulationDSFullCell` must be a model of the concept `TriangulationDSFullCell`. The class template `CGAL::Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell>` accepts that no third parameter be specified. It also accepts the tag `CGAL::Default` as third parameter. In both cases, `TriangulationDSFullCell` defaults to `CGAL::Triangulation_ds_full_cell<>`.

**Inherits From**

`CGAL::TriangulationDSFullCell` (the third template parameter)

**Is Model for the Concepts**

`CGAL::TriangulationFullCell`

Additionally, the class `CGAL::Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell>` also provides the following type, constructors and methods:

**Types**

typedef Data Data;  

The type of the additional data stored in the cell. If you read a `CGAL::Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell>` from a stream (a file) or write a `CGAL::Triangulation_full_cell<TriangulationTraits, Data, TriangulationDSFullCell>` to a stream, then streaming operators `<<` and `>>` must be provided for this type.
Creation

\texttt{template<\texttt{typename T}>}
\texttt{Triangulation\_full\_cell<\texttt{TriangulationTraits, Data, TriangulationDSFullCell}> c(\texttt{int dmax, T t});}

Sets the maximum possible dimension of the cell to \textit{dmax}. The parameter \textit{t} is passed to the \textit{Data} constructor.

Data access

\begin{itemize}
  \item \texttt{Data} \quad \texttt{c.data()} const \quad \texttt{c.data()}\texttt{ const}\n  \item \texttt{Data} \quad \texttt{c.data()} \quad \texttt{c.data()}\texttt{ }\texttt{&}\texttt{ }
\end{itemize}

\texttt{Data} \texttt{c.data()} const \quad \texttt{c.data()} \quad Returns a const reference to the stored data.

\texttt{Data} \texttt{c.data()} \quad \texttt{c.data()}\texttt{ &} \quad Returns a non-const reference to the stored data.

Input/Output

\begin{itemize}
  \item \texttt{istream} \texttt{is} \quad \texttt{istream} \texttt{is} >> \texttt{Triangulation\_full\_cell} \texttt{& v} \quad Inputs the non-combinatorial information given by the cell, i.e., the point and other possible information. The data of type \textit{Data} is also read.
  \item \texttt{ostream} \texttt{os} \quad \texttt{ostream} \texttt{os} << \texttt{Triangulation\_full\_cell} \texttt{v} \quad Outputs the non-combinatorial information given by the cell, i.e., the point and other possible information. The data of type \textit{Data} is also written.
\end{itemize}

See Also

\texttt{Triangulation\_vertex<\texttt{TriangulationTraits, Data, TriangulationDSVertex}>}
\texttt{Triangulation\_data\_structure<\texttt{Dimensionality, TriangulationDSVertex, TriangulationDSFullCell}>}
\texttt{Triangulation<\texttt{TriangulationTraits,TriangulationDataStructure>}
\texttt{Delaunay\_triangulation<\texttt{DelaunayTriangulationTraits, TriangulationDataStructure>}

75
CGAL::Triangulation::Locate_type

Definition

The enum Locate_type is defined by the class Triangulation to specify in what kind of face a point has been located in a triangulation.

```
enum Locate_type { ON_VERTEX,
    IN_FACE,
    IN_FACET,
    IN_FULL_CELL,
    OUTSIDE_CONVEX_HULL,
    OUTSIDE_AFFINE_HULL}
```

See Also

CGAL::Triangulation
Index

Pages on which definitions are given are presented in boldface.

<<  
Triangulation, 67  
Triangulation_ds_full_cell, 39  
Triangulation_ds_vertex, 35  
Triangulation_full_cell, 75  
Triangulation_vertex, 73  
TriangulationDataStructure, 29  
TriangulationFullCell, 49  
TriangulationVertex, 48  

Triangulation, 67  
Triangulation_ds_full_cell, 39  
Triangulation_ds_vertex, 35  
Triangulation_full_cell, 75  
Triangulation_vertex, 73  
TriangulationDataStructure, 29  
TriangulationFullCell, 49  
TriangulationVertex, 48  

are_incident_full_cells_valid  
Triangulation, 66  
associate_vertex_with_full_cell  
TriangulationDataStructure, 26  

Cell, 32–33  
clear  
TriangulationDataStructure, 27  
TriangulationDSFace, 41  
collapse_face  
Triangulation, 63  
TriangulationDataStructure, 27  
compare_lexicographically_d_object  
TriangulationTraits, 44  
compute_conflict_zone  
Delaunay_triangulation, 70  
construct_flat_orientation_d_object  
TriangulationTraits, 44  
contained_in_affine_hull_d_object  
TriangulationTraits, 44  
current_dimension  
Triangulation, 60  
TriangulationDataStructure, 22  

data  
Triangulation_full_cell, 75  
Triangulation_vertex, 73  
Delaunay_triangulation, 68–71  
DelaunayTriangulationTraits, 45–46  
delete_full_cell  
TriangulationDataStructure, 28  
delete_full_cells  
TriangulationDataStructure, 28  
delete_vertex  
TriangulationDataStructure, 28  
difference_type, 59  
empty  
Triangulation, 60  
TriangulationDataStructure, 22  

Face, 59  
face_dimension  
TriangulationDSFace, 40  
Facet, 58  
Facet_iterator, 59  
facets_begin  
Triangulation, 61  
TriangulationDataStructure, 24  
facets_end  
Triangulation, 61  
TriangulationDataStructure, 24  
finites_begin  
Triangulation, 61  
TriangulationDataStructure, 24  
finites_end  
Triangulation, 61  
finites_begin  
Triangulation, 61  
finites_end  
Triangulation, 61  
finites_begin  
Triangulation, 61  
finites_end  
Triangulation, 61  
finites_begin  
Triangulation, 61  
finites_end  
Triangulation, 61  
for_compact_container  
TriangulationDSFullCell, 39  
TriangulationDSVertex, 35  
Full_cell, 58  
full_cell  
Triangulation, 61  

Data, 72, 74

77
TriangulationDataStructure, 24
TriangulationDSFace, 40
TriangulationDSVertex, 34
Vertex, 30

Full_cell_handle, 30, 32, 59
Full_cell_iterator, 59
full_cells
TriangulationDataStructure, 22
full_cells_begin
Triangulation, 61
TriangulationDataStructure, 24
full_cells_end
Triangulation, 61
TriangulationDataStructure, 24

Geom_traits, 58
geom_traits
Triangulation, 60
get_tds_data
TriangulationDSFullCell, 37

has_neighbor
Cell, 32
TriangulationDSFullCell, 38
has_vertex
Cell, 32
TriangulationDSFullCell, 38

IN_FACE, 59, 76
IN_FACET, 59, 76
in_flat_orientation_d_object
TriangulationTraits, 44
in_flat_side_of_oriented_sphere_d_object
DelaunayTriangulationTraits, 46
IN_FULL_CELL, 59, 76
incident_faces
TriangulationDataStructure, 23
incident_full_cells
TriangulationDataStructure, 22, 23
index
Cell, 32
TriangulationDSFace, 41
TriangulationDSFullCell, 37
index_of_covertex
Triangulation, 61
TriangulationDataStructure, 24
infinite_full_cell
Triangulation, 60
infinite_vertex
Triangulation, 60
insert
Delaunay_triangulation, 69
Triangulation, 64
insert_in_conflicting_cell
Delaunay_triangulation, 70
insert_in_face
Triangulation, 65
TriangulationDataStructure, 24
insert_in_facet
Triangulation, 65
TriangulationDataStructure, 24
insert_in_full_cell
Triangulation, 66
TriangulationDataStructure, 24
insert_in_hole
Triangulation, 65
TriangulationDataStructure, 25
insert_in_tagged_hole
Triangulation_data_structure, 51
insert_increase_dimension
TriangulationDataStructure, 25
insert_outside_affine_hull
Delaunay_triangulation, 70
Triangulation, 66
insert_outside_convex_hull
Triangulation, 66
is_full_cell
TriangulationDataStructure, 22
is_in_conflict
Delaunay_triangulation, 70
is_infinite
Triangulation, 60, 61
is_valid
Cell, 33
Triangulation, 66
Triangulation_data_structure, 51
Triangulation_ds_full_cell, 56
Triangulation_ds_vertex, 53
TriangulationDataStructure, 28
TriangulationDSFullCell, 39
TriangulationDSVertex, 35
Vertex, 31
is_vertex
TriangulationDataStructure, 22
locate
Triangulation, 61–63
Locate_type, 59, 76

Maximal_dimension, 58
maximal_dimension
Triangulation, 60
TriangulationDataStructure, 22
TriangulationDSFullCell, 37
mirror_index
TriangulationDataStructure, 23
TriangulationDSFullCell, 37
mirror_vertex
TriangulationDataStructure, 23
TriangulationDSFullCell, 38
neighbor
    Cell, 32
    TriangulationDataStructure, 24
    TriangulationDSFullCell, 37
new_full_cell
    TriangulationDataStructure, 26
new_vertex
    TriangulationDataStructure, 26
number_of_finite_full_cells
    Triangulation, 60
number_of_full_cells
    Triangulation, 60
    TriangulationDataStructure, 22
number_of_vertices
    Triangulation, 60
    TriangulationDataStructure, 22
ON_VERTEX, 59, 76
orientation_d_object
    TriangulationTraits, 44
OUTSIDE_AFFINE_HULL, 59, 76
OUTSIDE_CONVEX_HULL, 59, 76
Point, 58
point
    TriangulationVertex, 47
    Vertex, 30
remove
    DelaunayTriangulation, 69
remove_decrease_dimension
    TriangulationDataStructure, 27
set_current_dimension
    TriangulationDataStructure, 27
set_full_cell
    TriangulationDSFace, 41
    TriangulationDSVertex, 34
    Vertex, 30
set_index
    TriangulationDSFace, 41
set_mirror_index
    TriangulationDSFullCell, 38
set_neighbor
    Cell, 32
    TriangulationDSFullCell, 38
set_neighbors
    TriangulationDataStructure, 27
set_point
    TriangulationVertex, 47
    Vertex, 30
set_vertex
    Cell, 32
    TriangulationDSFullCell, 38
side_of_oriented_sphere_d_object
    DelaunayTriangulationTraits, 46
size_type, 59
star
    TriangulationDataStructure, 23
swap_vertices
    TriangulationDSFullCell, 38
tds
    Triangulation, 60
TDS_data, 36
Triangulation, 58–67
    Locate_type, 76
Triangulation_data_structure, 30, 32, 50–52
Triangulation_ds, 58
Triangulation_ds_full_cell, 55–56
Triangulation_ds_vertex, 53–54
Triangulation_face, 57
Triangulation_full_cell, 74–75
Triangulation_vertex, 72–73
TriangulationDataStructure, 20–29
TriangulationDSFace, 40–41
TriangulationDSFullCell, 36–39
TriangulationDSVertex, 34–35
TriangulationFullCell, 49
TriangulationTraits, 42–44
TriangulationVertex, 47–48
Vertex, 30–31
Vertex, 58
vertex
    Cell, 32
    TriangulationDataStructure, 23
    TriangulationDSFace, 41
    TriangulationDSFullCell, 37
Vertex_handle, 30, 32, 59
Vertex_iterator, 59
vertices_begin
    Triangulation, 61
    TriangulationDataStructure, 23
    TriangulationDSFullCell, 37
vertices_end
    Triangulation, 61
    TriangulationDataStructure, 23
    TriangulationDSFullCell, 37