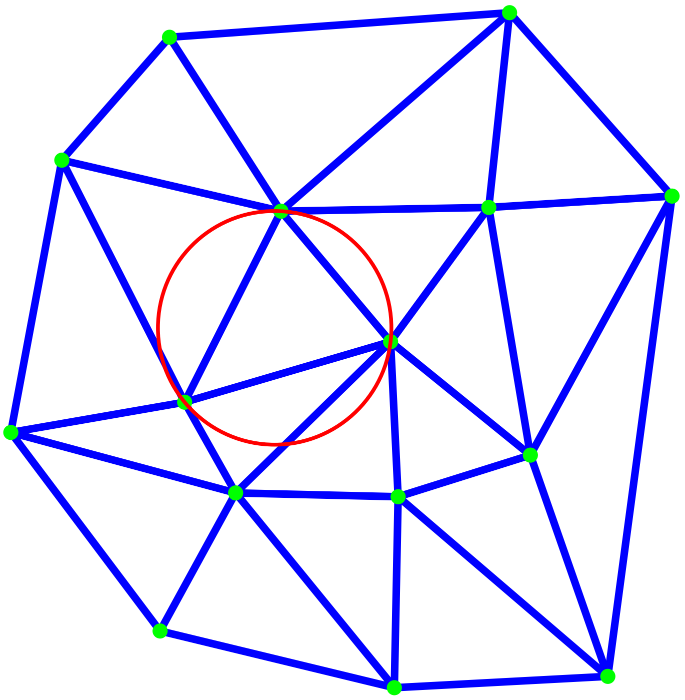
Triangulations

in

CGAL



Mariette Yvinec

INRIA
SOPHIA ANTIPOLIS

Overview

Specifications

- Definition
- Various triangulations
- Functionalities

Representation

- The set of faces
- Representation based on vertices and cells

Software design

- The traits class
- The triangulation data structure

Algorithms

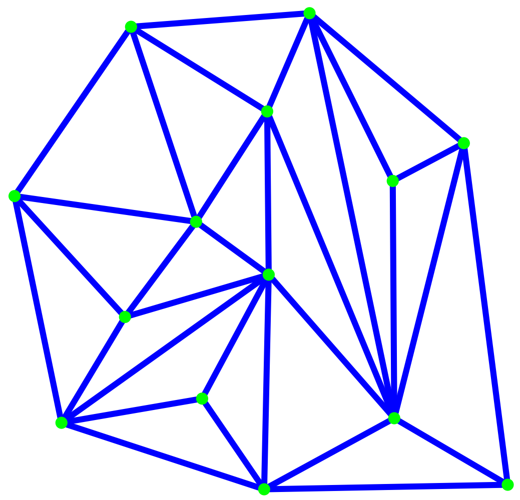
- Point location

Examples

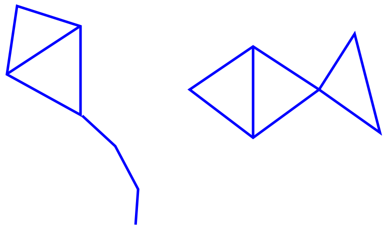
Exercises

What is a triangulation

A 2d-triangulation is a set of triangles such that:
- the set is edge-connected
- two triangles are either disjoint or share an edge or a vertex.



These are not triangulations



Various triangulations

Cgal offers various kind of 2d and 3d triangulations

Basic triangulations

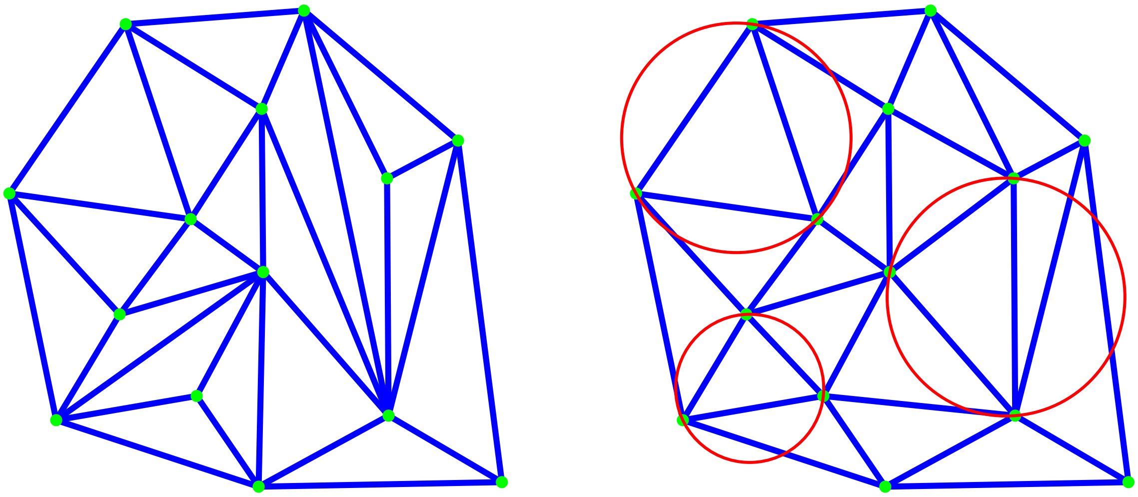
Delaunay triangulations

Regular triangulations

Constrained triangulations

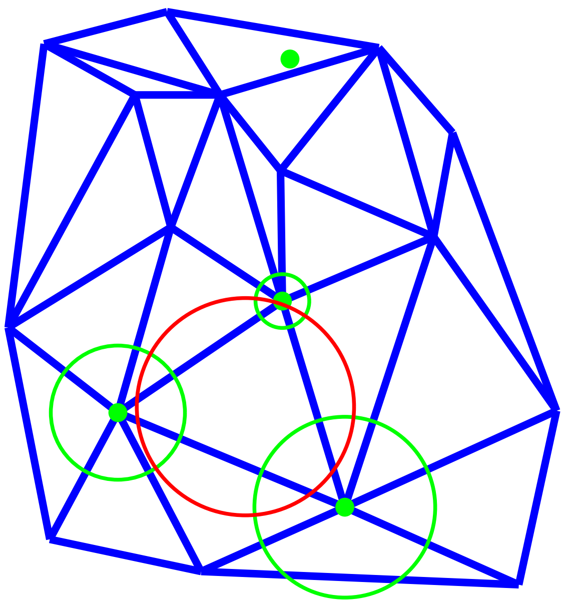
Constrained Delaunay triangulations

Basic and Delaunay triangulations



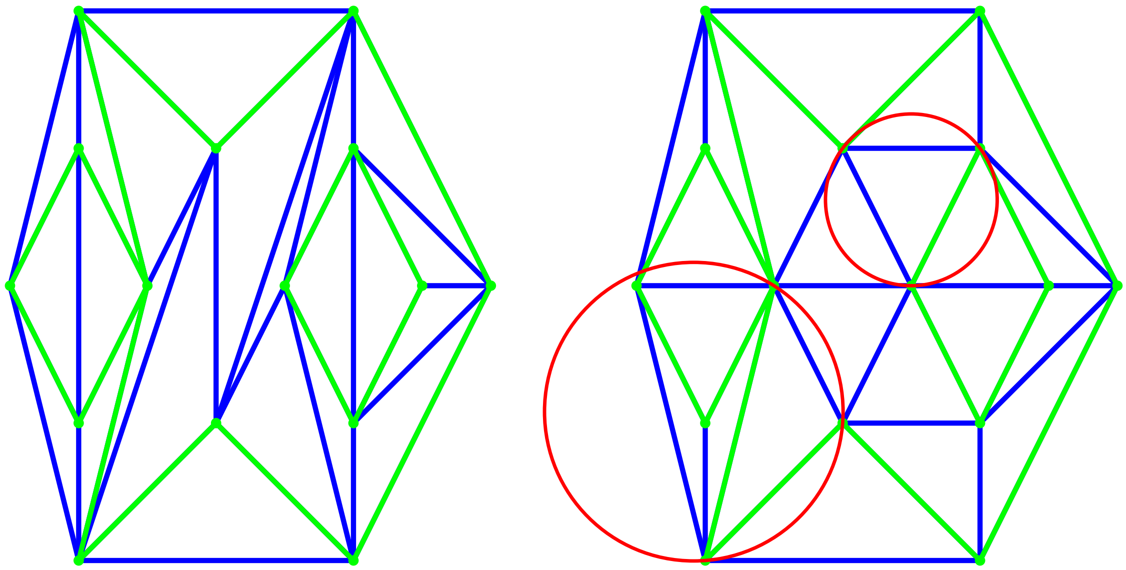
Basic triangulations : lazy incremental construction, no control on the shapes of triangles
Delaunay triangulations: empty circle property

Regular triangulations



Regular triangulations: generalisation of Delaunay triangulations.
A regular triangulation is defined for a set of weighted points. Each weighted point can be considered as a sphere whose square radius is equal to the weight. The regular triangulation is the dual of the power diagram.

Constrained triangulations
Constrained Delaunay triangulations

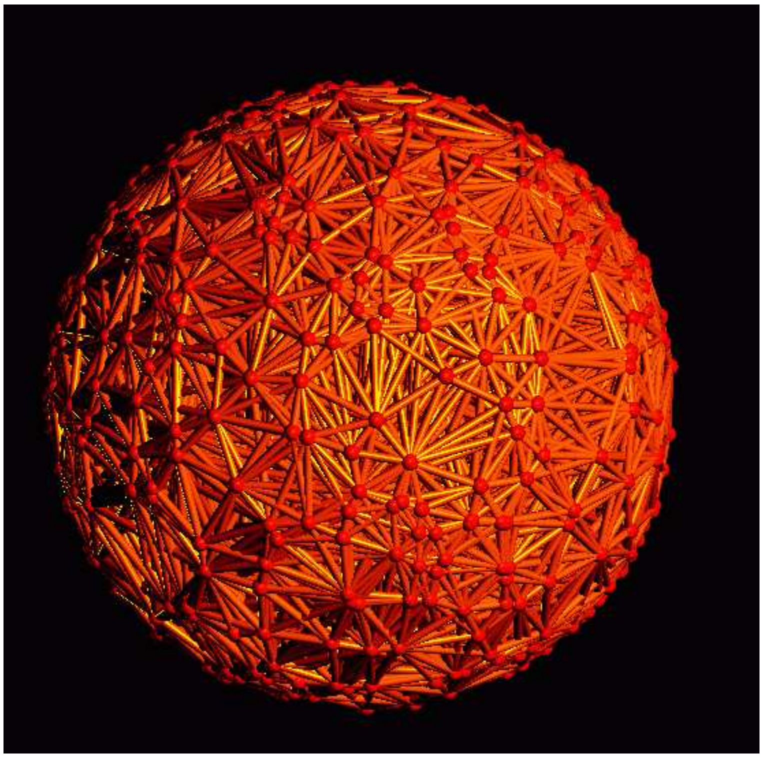


Constrained triangulations : allow to enforce edges in the triangulation.

Constrained Delaunay triangulations : constrained triangulations which are as much Delaunay as possible.

Each triangle satisfies the *constrained empty circle property* : its circumscribing circle encloses no vertex visible from the interior of the triangle, where enforced edges are considered as visibility obstacles.

3D Triangulations



Functionalities of Cgal triangulations

General functionalities

Traversal of a triangulation

- passing from a face to its neighbors
- iterators to visit all or faces of a triangulation
- circulators to visit all faces around a vertex or all faces intersected by a line.

Point location query

Insertion, removal, flips

These functionalities are adapted to each kind of triangulations. For instance, the insertions and deletions in a Delaunay triangulation maintain the empty circle property.

Additionnal functionalities

for particular triangulations

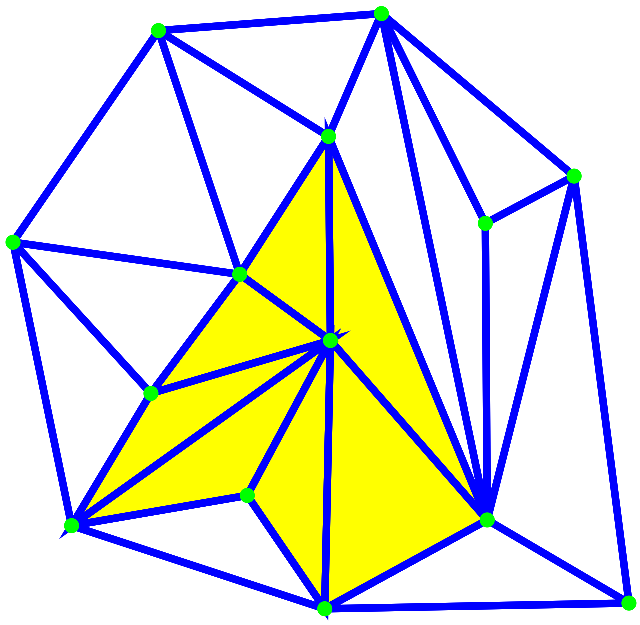
Traversal of a triangulations

Iterators

All_faces_iterator
All_vertices_iterator
All_edges_iterator

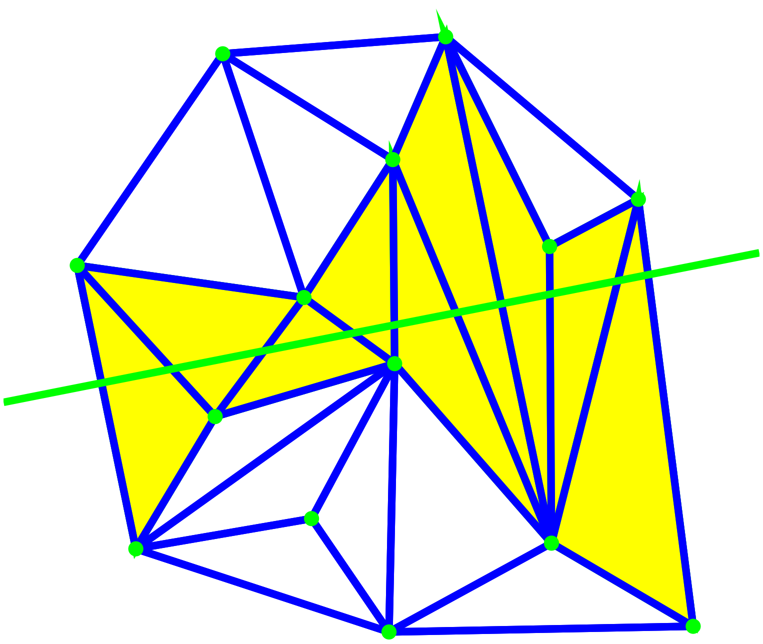
Circulators

Face_circulator : faces incident to a vertex
Edge_circulator : edges incident to a vertex
Vertex_circulator : incident to a vertex

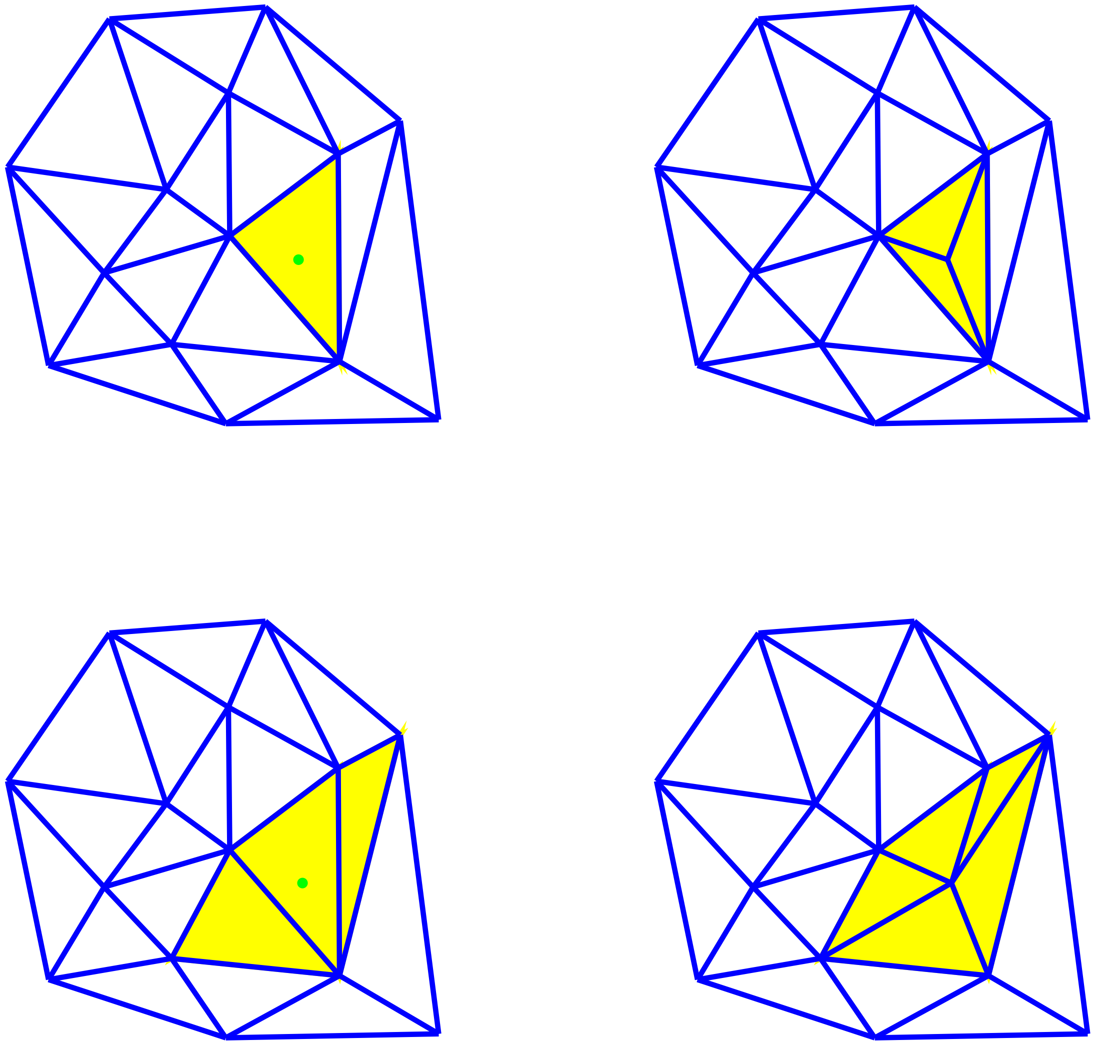


Traversal of a triangulations cont'd

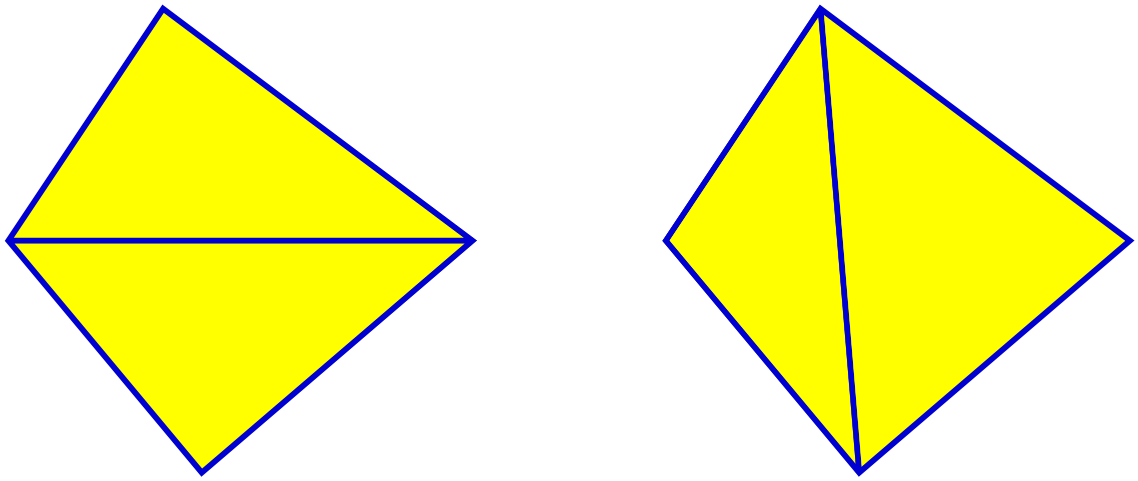
Line_face_circulator



Point location, insertion, removal



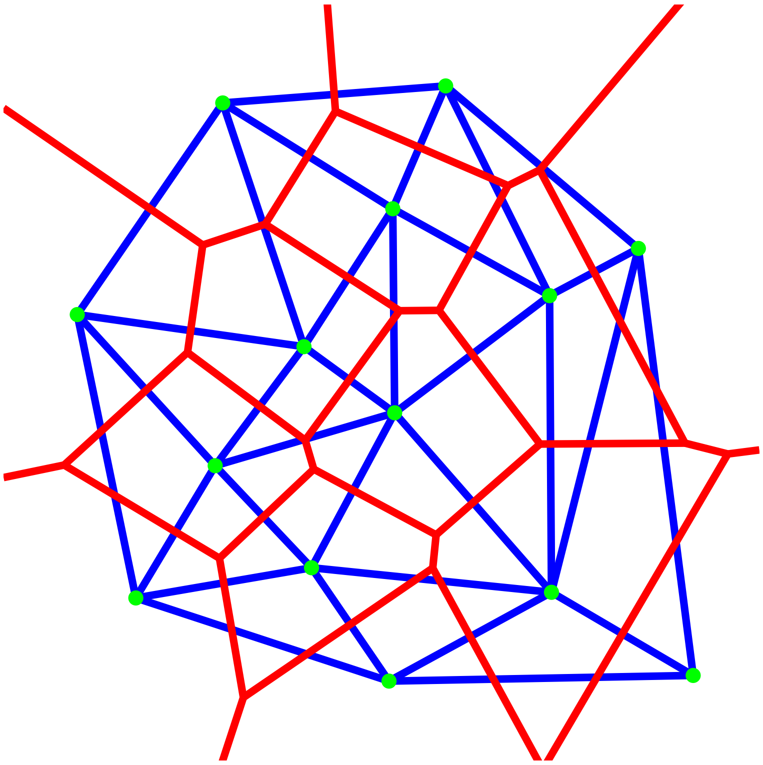
Flip



Additional functionalities for Delaunay triangulations

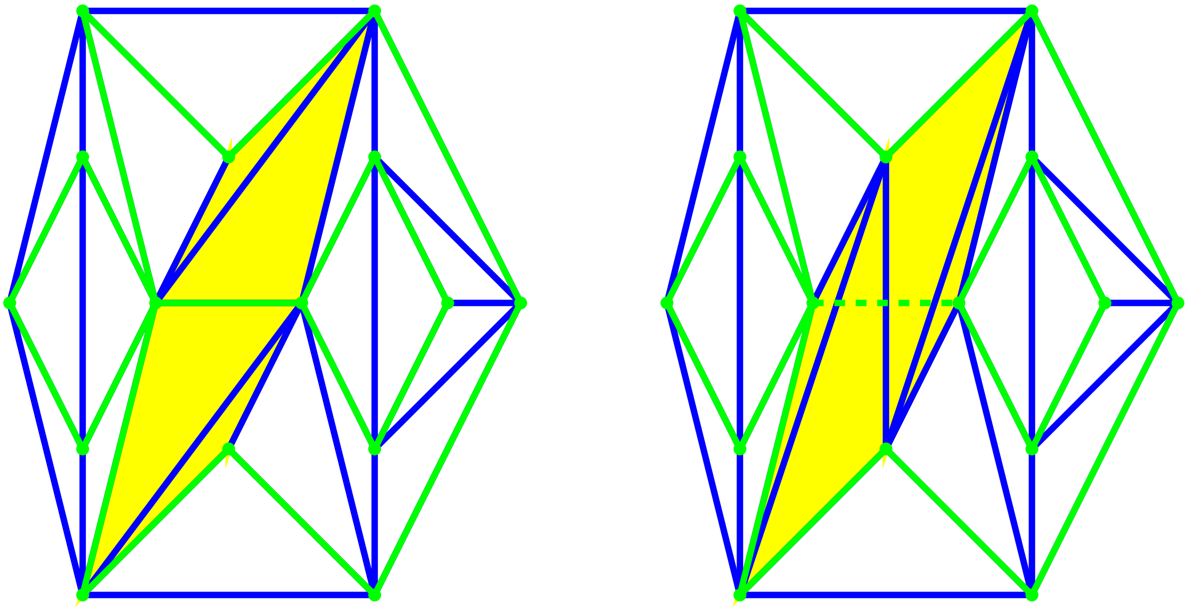
Nearest neighbor queries

Voronoi diagram

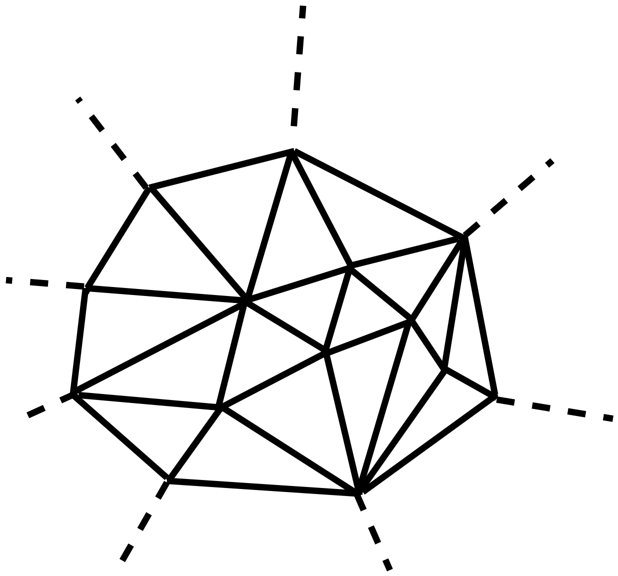


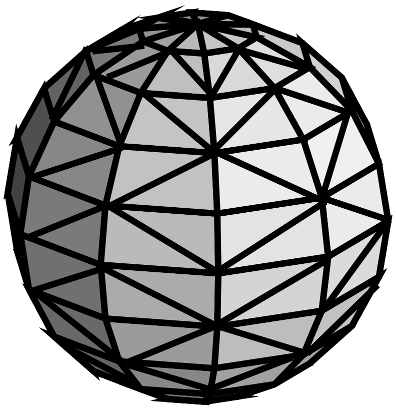
Additional functionalities for constrained triangulations, and Delaunay constrained triangulations

Insertion and suppression of constraints



Triangulations **Set of Faces**





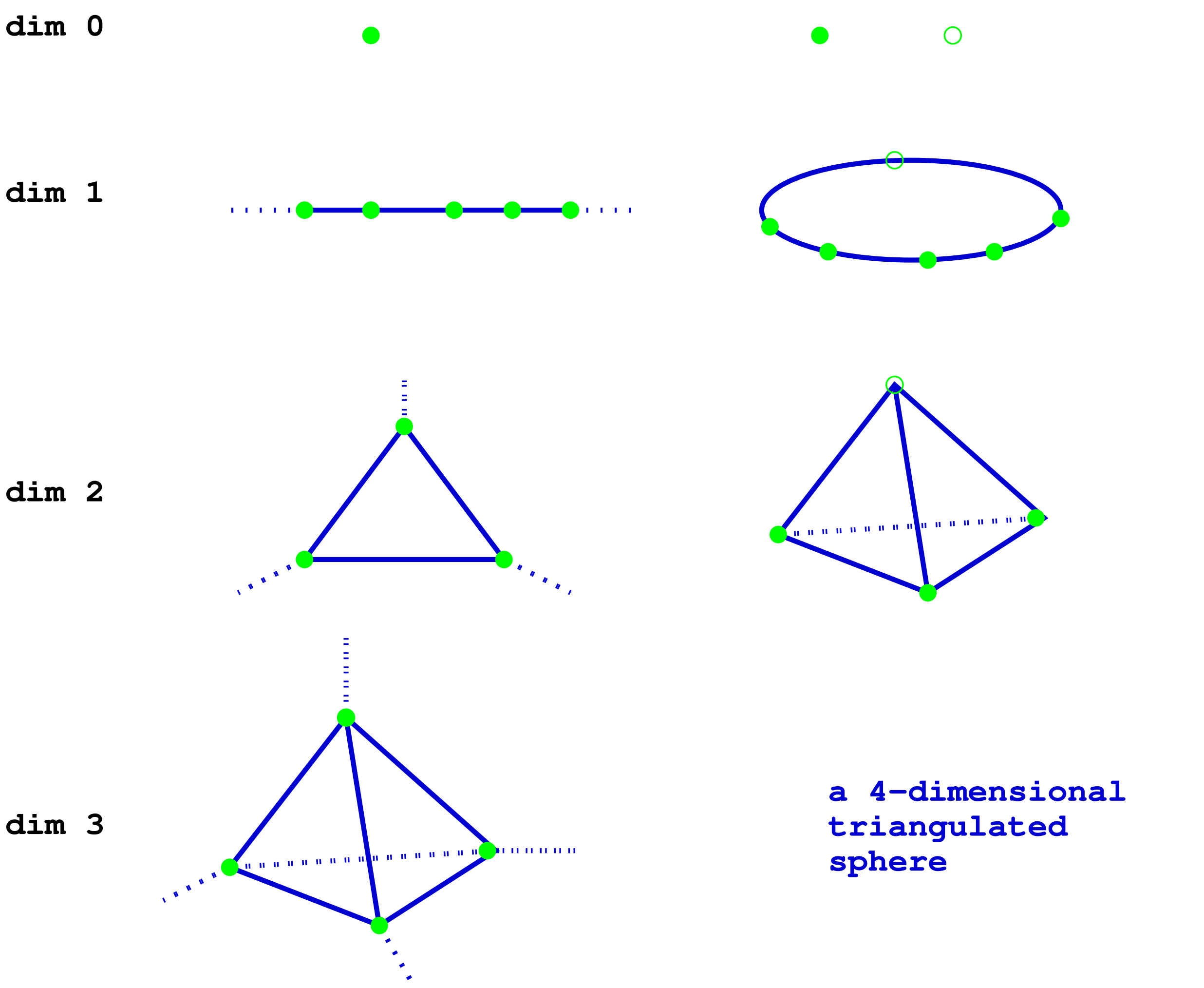
Triangulations in Cgal always covers the whole convex hull of their vertices.
Triangulated polygonal regions can be obtained through constrained triangulations.

A fictitious vertex called the *infinite vertex* is added.
Any face is a triangle.
Any edge is incident to two faces.
The set of faces $\equiv$ a 2d topological sphere

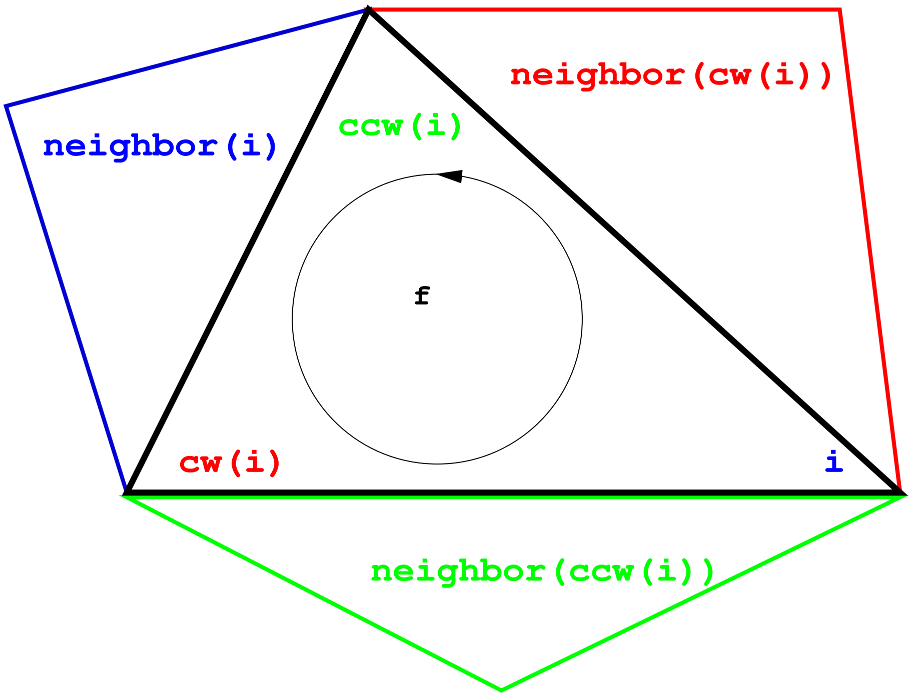
Triangulations

Set of Faces

In any dimension, the set of faces of a Cgal Triangulation is combinatorically equivalent to a triangulated sphere.



The representation is based on faces and vertices.
Edges are only implicitely represented



Vertex

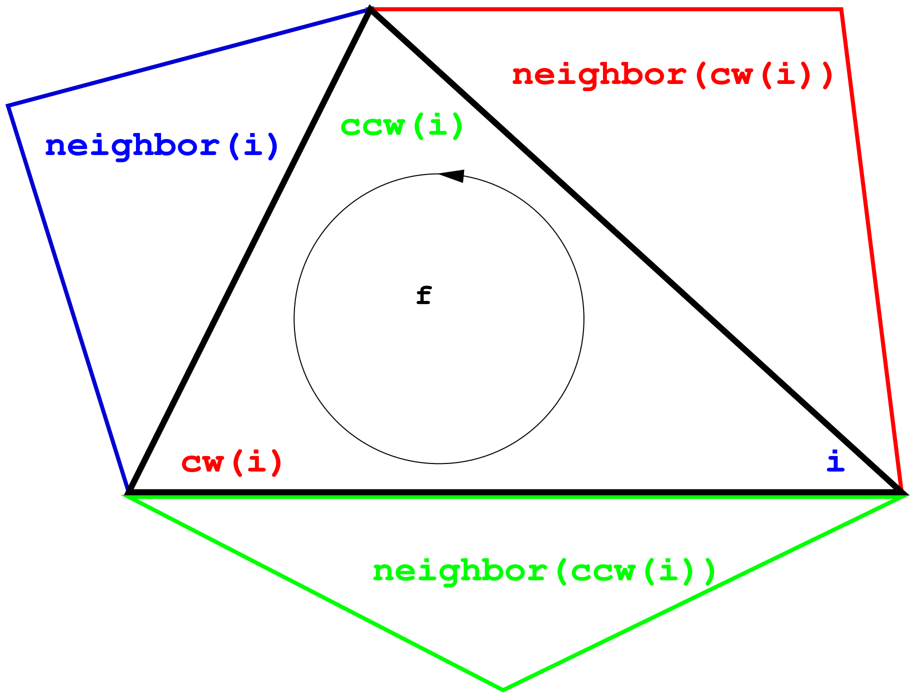
Face* v_face

Face

Vertex* $vertex$[3]
Face* $neighbor$[3]

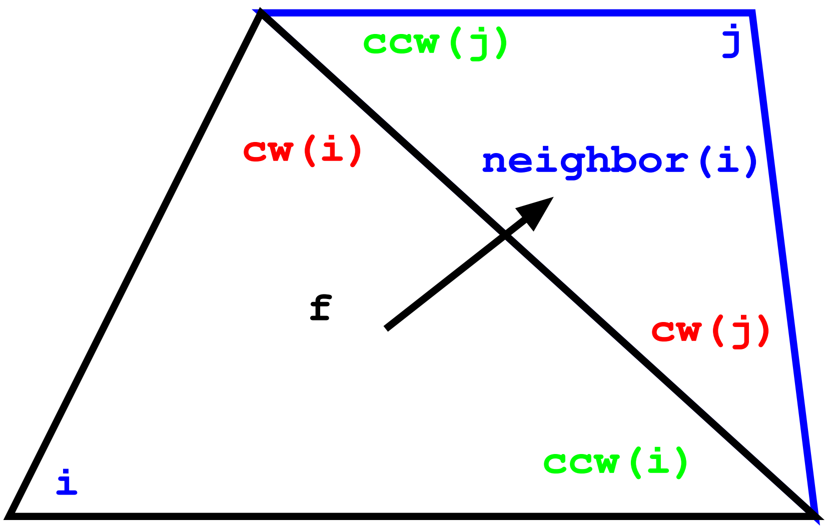
Triangulations Representation cont'd

functions $cw(i)$ et $ccw(i)$



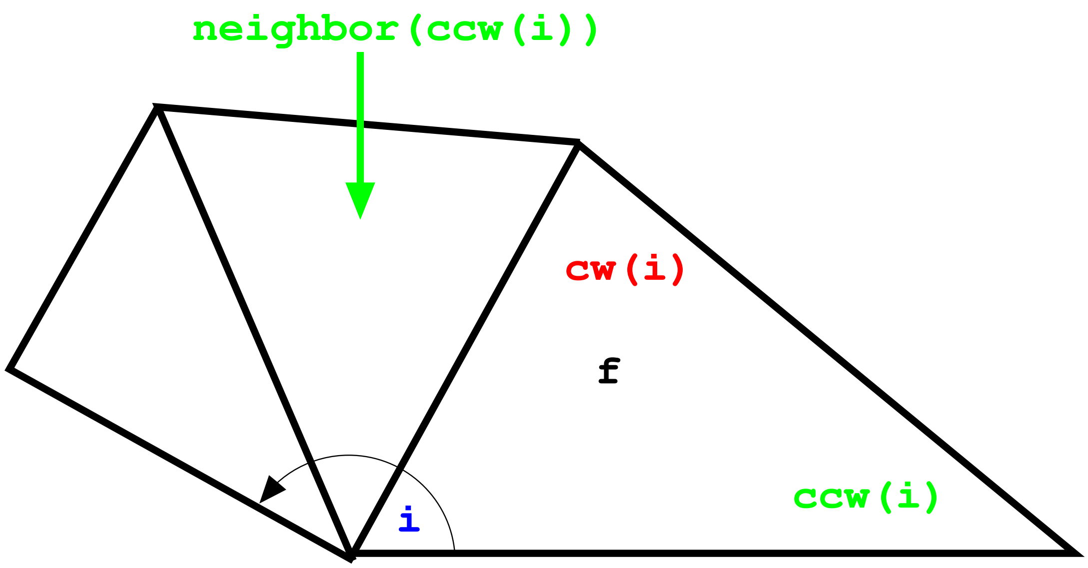
Triangulations   Representation cont'd

From one face to a another

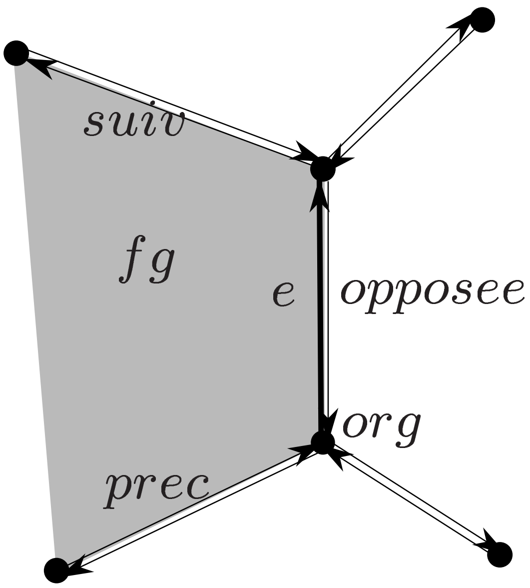



```
n = f->neighbor(i)
j = n->index(f)
```

Around a vertex



DCEL
Doubly Connected Edge List



Sommet

Halfedge* v_edge

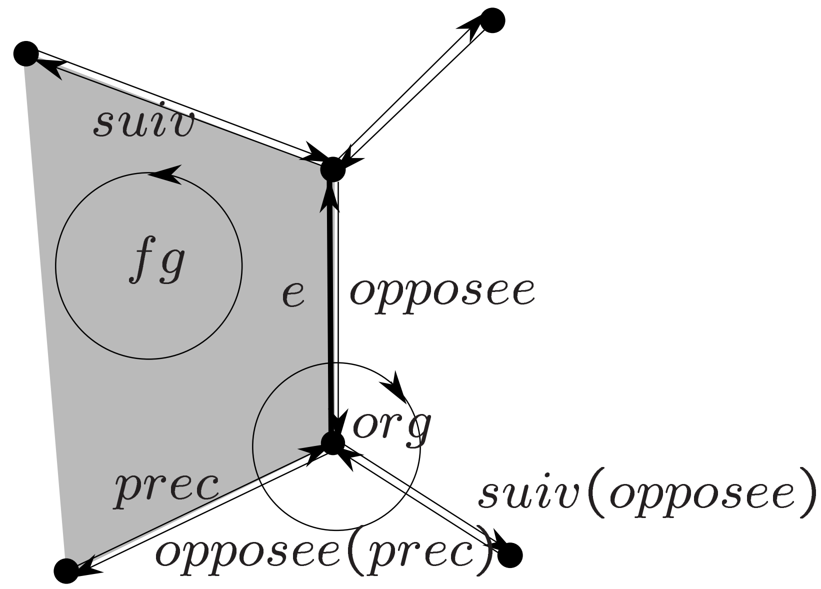
Face

Halfedge* f_edge

Demi-arte

Face *	fg
Vertex*	org
Halfedge*	$opposee$
Halfedge*	$suiv$
Halfedge*	$prec$

DCEL
Doubly Connected Edge List



Demi-artes incidentes à une face f:

$e = f\_edge(f)$

counterclockwise $suiv(e)$

clockwise $prec(e)$

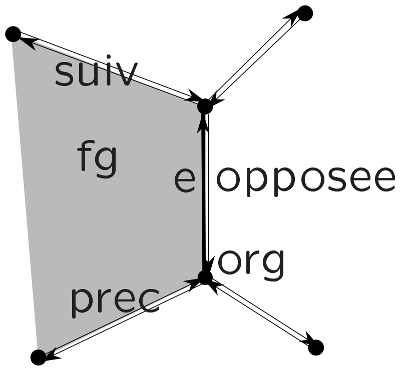
Demi-artes issues d'un sommet v:

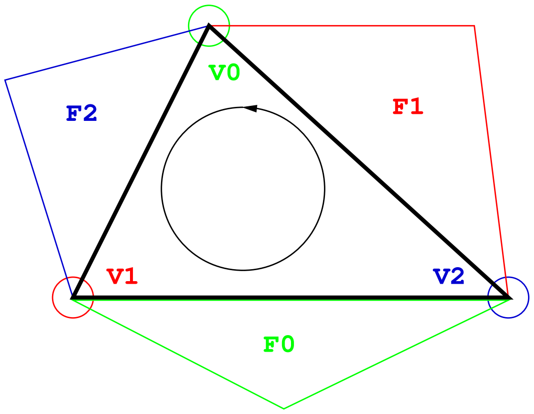
$e = v\_edge(v)$

counterclockwise $opposee(prec(e))$

clockwise $suiv(opposee(e))$

Triangulations Representation cont'd





n	sommets
$3n - 6$	artes
$2n - 4$	faces

	DCEL	CGAL Tds
Vertex **Edges** **Faces**	n $4 \times 2 \times (3n - 6)$ $(2n - 4)$	n $6 \times (2n - 4)$
Total	$27n$	$13n$

Software design — Traits class

Like many other Cgal classes, the triangulation classes are parametrised by two template parameters.

Polygon_2<Traits, Container>
Polyhedron_3<Traits, HDS>
Planar_map_2<Dcel,Traits>
Arrangement_2<Dcel,Traits,Base_node>
Min_circle_2<Traits>
Point_set_2<Traits>
Range_tree_k<Traits>

Triangulation_2<Traits, TDS>
Triangulation_3<Traits, TDS>

Traits Geomeric traits
TDS Triangulation data structure

Geometric traits

Geometric traits classes provide :

- Basic geometric objects
- Predicates and Constructors needed

Requirements for traits are documented

$\Longrightarrow$ basic library data structure and algorithms

can be used with user defined objects

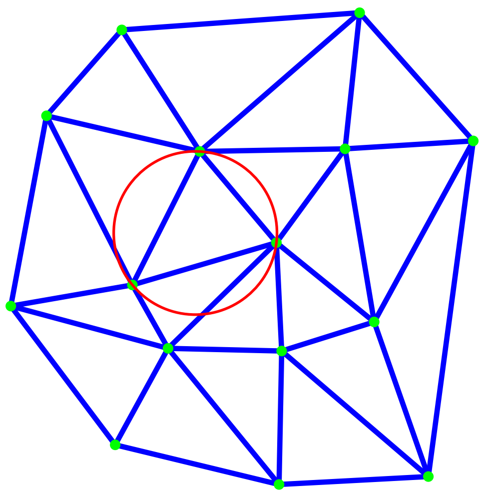
Default traits classes provided

Delaunay triangulation traits class

Requirements

Point,
Segment, Triangle
orientation test
in_circle test

Line, Ray
circumcenter
bisector



Default traits class

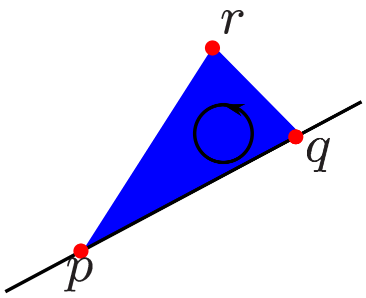
Triangulation_euclidean_traits_2<Rep>

Delaunay triangulation of 2d points

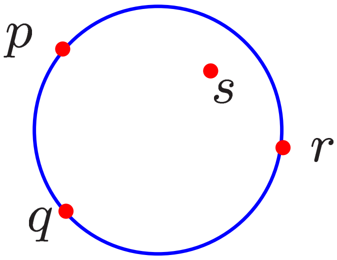
```
typedef Cartesian<double> Rep;
typedef Triangulation_euclidean_traits_2<Rep> Traits;
typedef Delaunay_triangulation_2<Traits> Triangulation;
```

The predicates for Delaunay triangulations

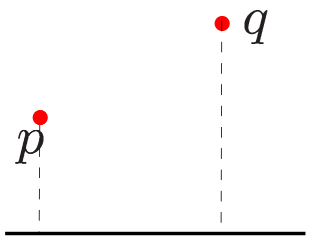
- **Orientation test**



- **Incircle test**

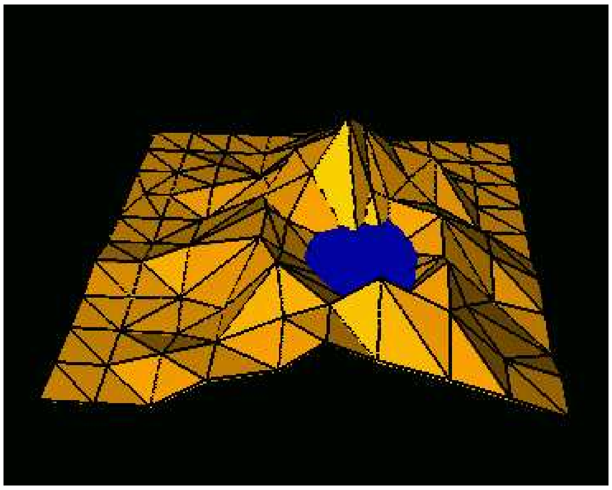


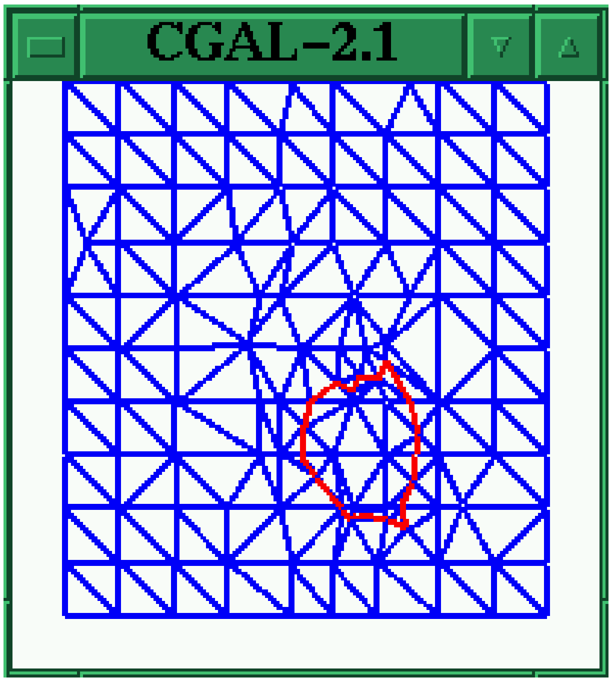
- **Coordinates comparison**



Use of traits class

Terrain





Traits class for a terrain

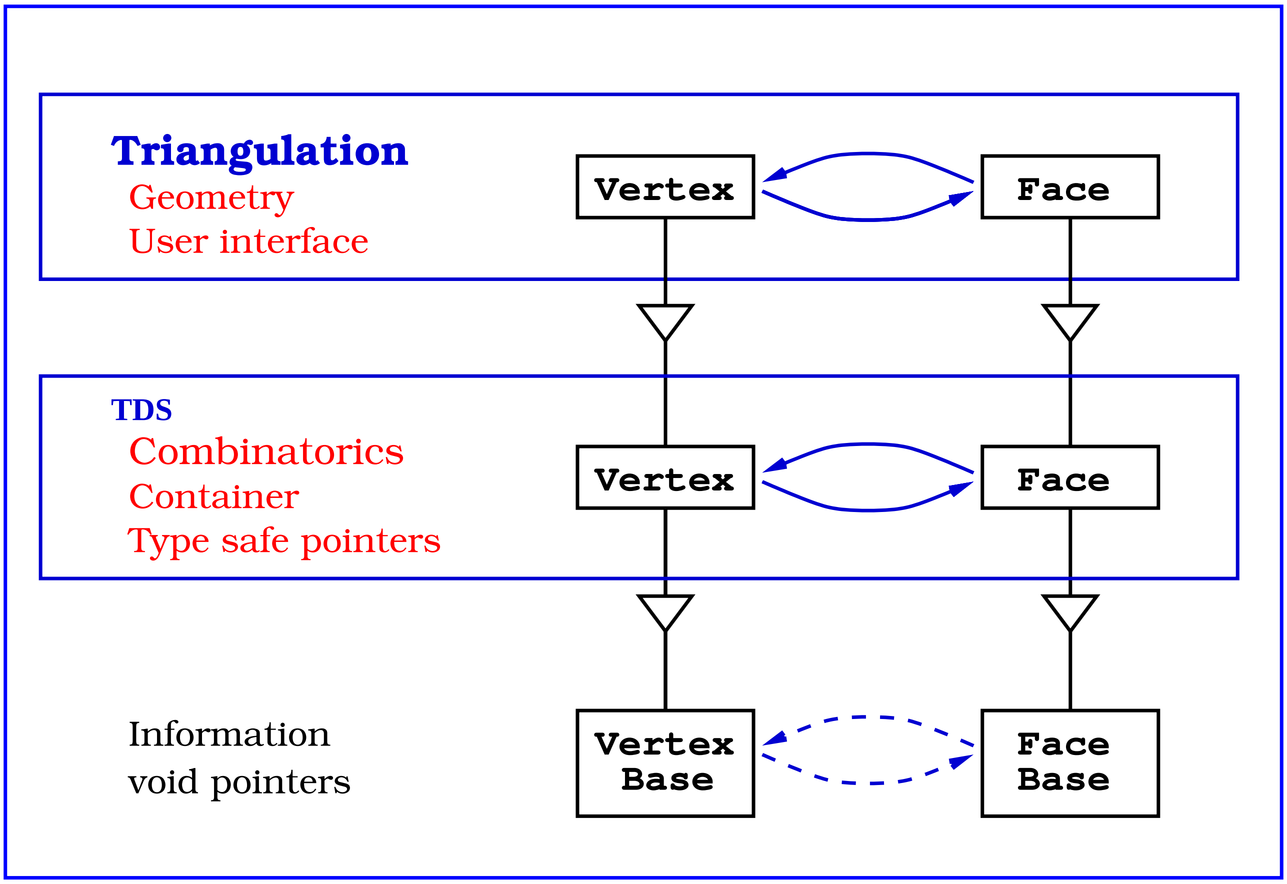
3D points

orientation in_circle	on x and y coordinates

▷ Triangulation_euclidean_traits_xy_3<Rep>

Definition of a Terrain

```
typedef Cartesian<double> Rep;
typedef Triangulation_euclidean_traits_xy_3<Rep> Traits;
typedef Delaunay_triangulation_2<Traits> Triangulation;
```



template< class Gt, class Tds>
Triangulation_2 ;

template<class Vb, class Fb>
Triangulation_data_structure_2;

Software design Triangulations are implemented in Cgal along a three layers design. Analogous designs are used for the planar map and polyhedron packages of Cgal.

The bottom layer in this design consists of two base classes for faces and vertices that store geometrical informations such as the coordinates of vertices, and any other attribute (such as colors or information about constrained edges) needed by the application. Those base classes handle incidence and adjacency relations but remain nevertheless independent of each other types (they exchange void* pointers).

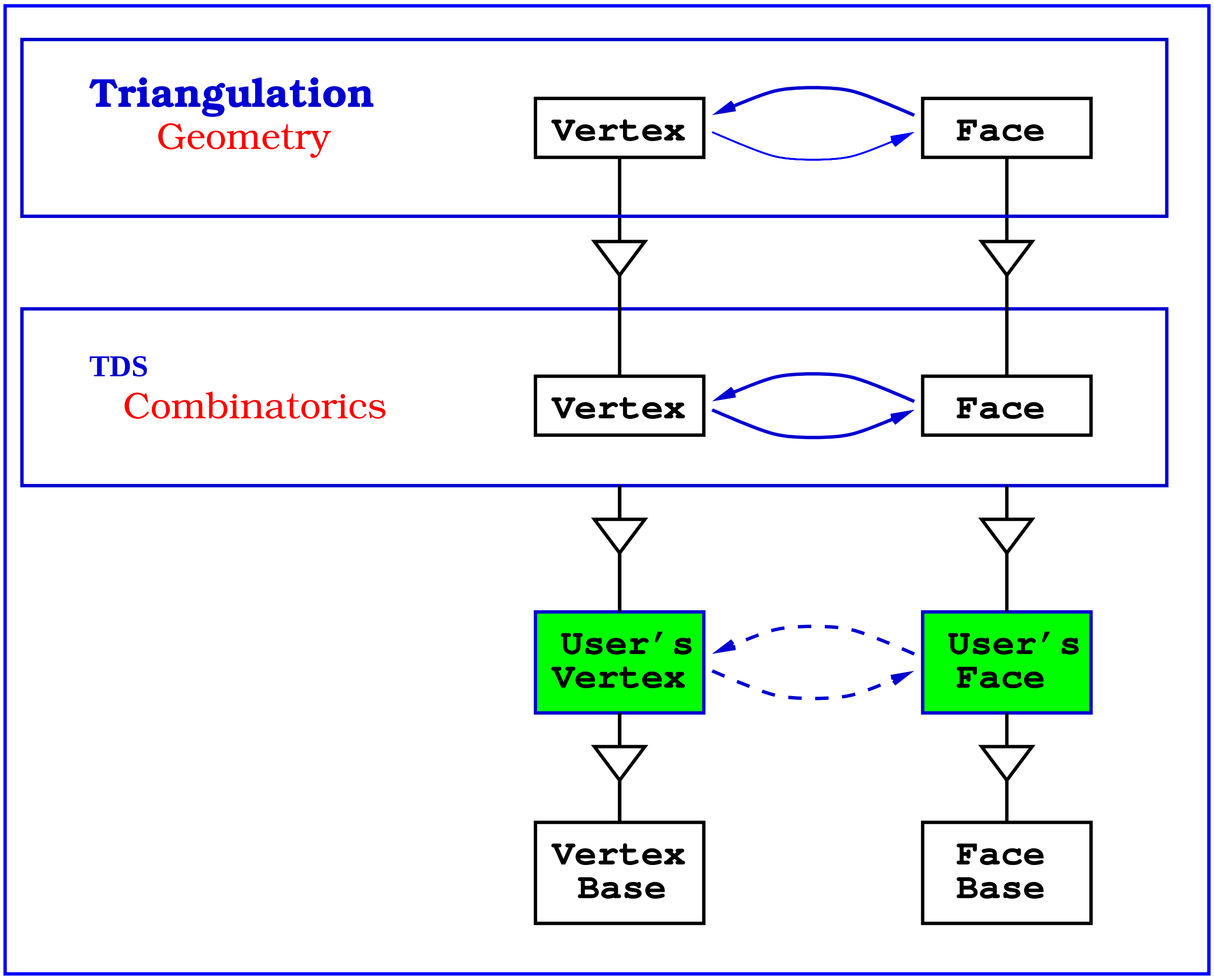
At the middle layer, the triangulation data structure acts as a container for the set of faces and vertices of a triangulation. It also handles the combinatorial operations on triangulations. The triangulation data structure class is templated by the face base class and the vertex base class. It derives its own vertex and face classes from those base classes and restore type safe pointer operations.

At last, at the top level, the triangulation class itself handles the geometric embedding of the triangulation and provides high level functionalities. At this top level, different classes are available to represent the different kinds of triangulations. The triangulation classes offer a user interface sufficient for most applications.

Some additional flexibility arise from this design. First because the base classes for faces and vertices are independent from each other, the user can easily plug in the triangulation data structure his own vertex base class and/or his own face base class instead of the default ones. This allows the user to add in the vertices and/or faces some additional information required by his application. Second, because the triangulation data structure is a concept rather than a class, the user can instantiate the corresponding parameter with any model for this concept.

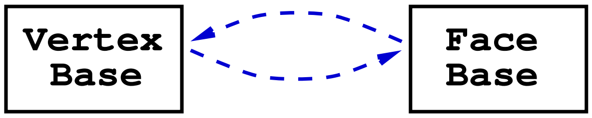
Triangulations

Software design



Triangulations design   The base level

Information
void pointers



Vertex_base

Vertex_base :: Point
Vertex_base(Point p, void* f)

Point point();
void* face();

void* set_point();
void* set_face();

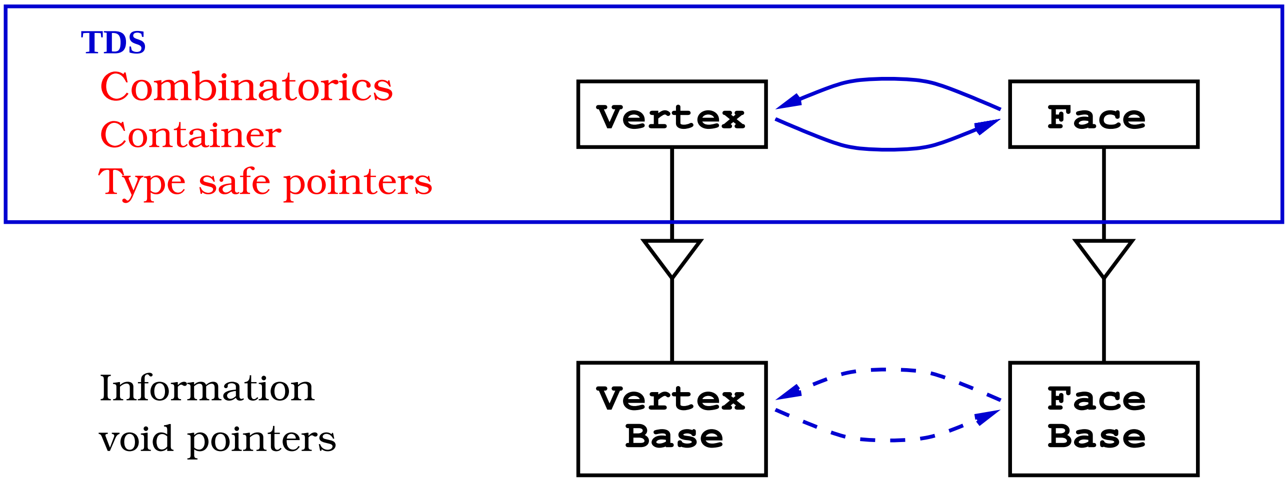
Face_base

Face_base(void* v0, void* v1, void* v2,
          void* n0, void* n1, void* n2)

void* vertex(int i);
void* neighbor(int i);

void* set_vertex(int i, void* v);
void* set_neighbor(int i, void* f);

Triangulation data structure



Tds<Vb,Fb>

Types

Tds<Vb,Fb>::Vertex inherits from Vb
Tds<Vb,Fb>::Face inherits from Fb

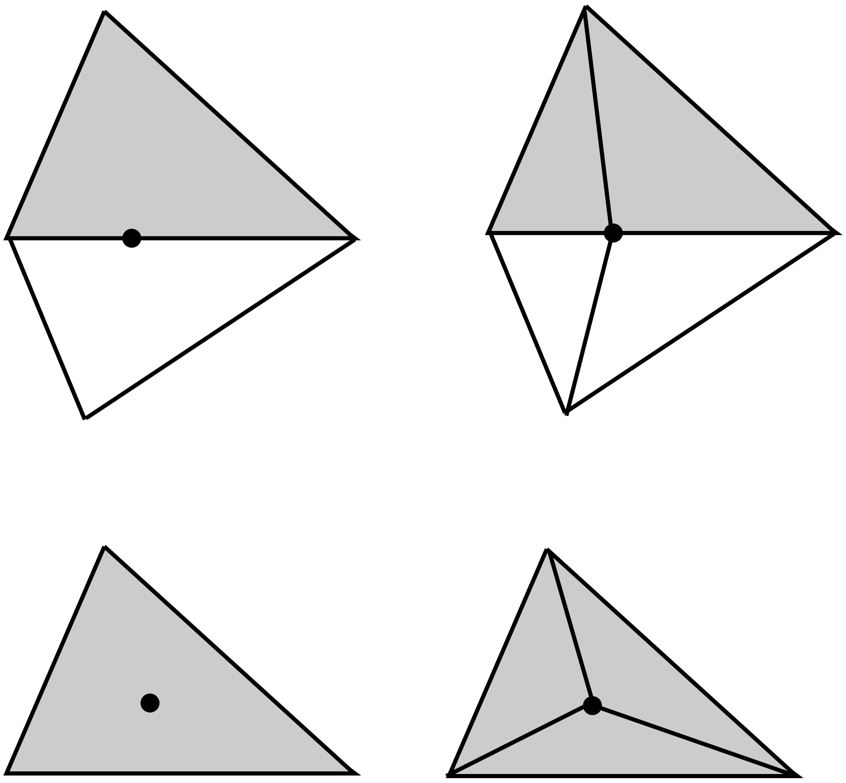
Tds<Vb,Fb>::Face_iterator
Tds<Vb,Fb>::Edge_iterator
Tds<Vb,Fb>::Vertex_iterator

Tds<Vb,Fb>::Face_circulator
Tds<Vb,Fb>::Edge_circulator
Tds<Vb,Fb>::Vertex_circulator

Triangulation data structure

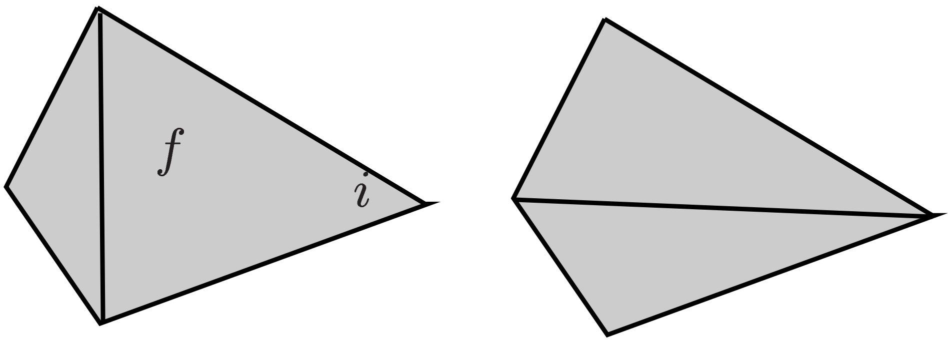
Combinatorial operations

void insert_in_face (Vertex* v,Face* f)
void insert_in_edge (Vertex* v, Face* f, int i)
void remove_degre_3 (Vertex* v);

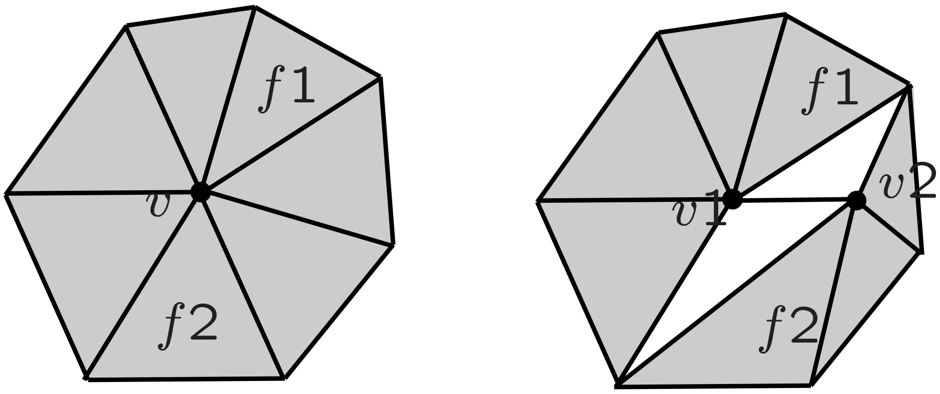


Triangulation data structure
Combinatorial operations

void flip (Face* f, int i);



To be done
void split_vertex (Vertex*, Face* f1, Face* f2)
void join_vertices (Vertex* v1, vertex* v2)

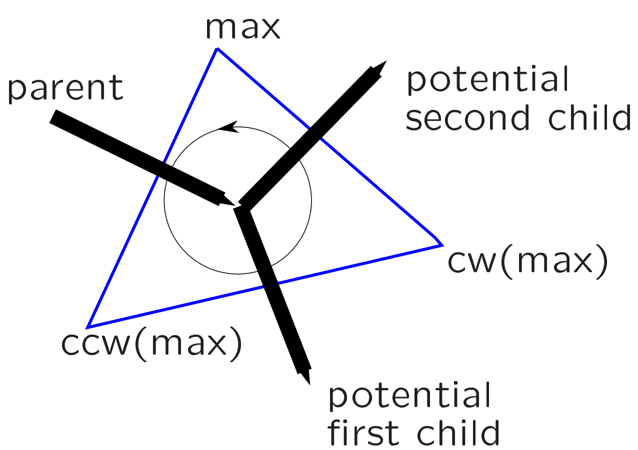


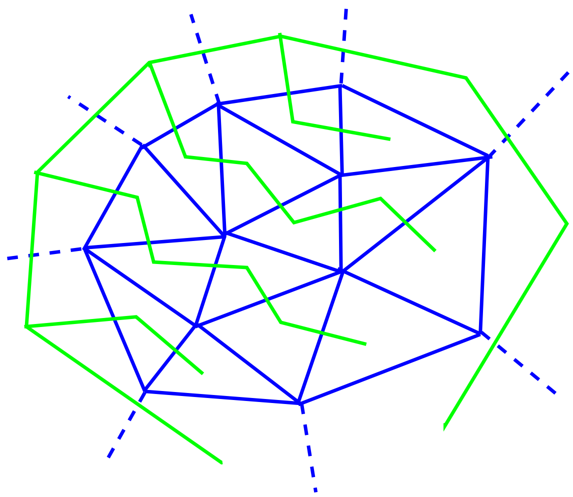
Models of triangulation data structure Currently, Cgal provides two different models for the concept of triangulation data structure. The first one is restricted to two dimensional triangulations embedded in a plane. It does not use any additional pointers to handle the set of vertices and faces of the triangulation. The iterators on faces and vertices of the triangulation are coded using an implicit tree on the faces related to the planar embedding. The second model of triangulation data structure offered in Cgal uses a doubly connected list to handle the set of faces. This model allows to represent as well triangulated surfaces embedded in three space. For three dimensional triangulations, Cgal offers only one model of triangulated data structure which uses a doubly connected list to handle the set of cells.

Triangulation data structure　　Models

Triangulation_default_data_stucture_2 <Gt, Vb, Fb>

for 2-triangulation embedded in the plane





Triangulation_data_stucture_using_list_2 <Vb,Fb>

Any orientable 2-triangulation

The Triangulation Class
CGAL::Triangulation_2<Gt, Tds >

typedef Gt geometric traits
typedef Tds Triangulation data structure
typedef Triangulation_2<Gt, Tds > Triangulation;

Types

Gt::Point_2
Gt::Segment_2
Gt::Triangle_2

Triangulation::Vertex inherits from Tds::Vertex
Triangulation::Face inherits from Tds::Face

Triangulation::Vertex_handle
Triangulation::Face_handle
typedef pair<Face_handle, int> Edge ;

Triangulation::Face_iterator
Triangulation::Edge_iterator
Triangulation::Vertex_iterator

Triangulation::Line_face_circulator

Triangulation::Face_circulator
Triangulation::Edge_circulator
Triangulation::Vertex_circulator

The triangulation class (cont'd)

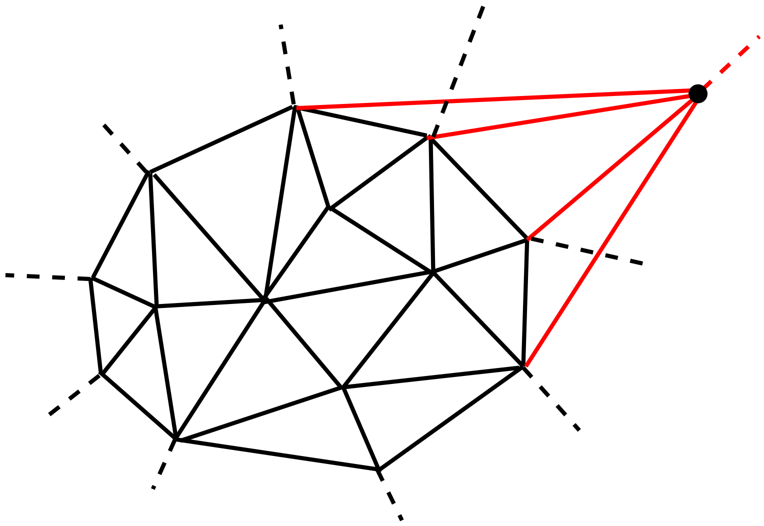
High level functions

```
enum Locate_type { VERTEX=0, EDGE, FACE,
                   OUTSIDE_CONVEX_HULL,
                   OUTSIDE_AFFINE_HULL }

Face_handle
locate(Point query,
       Locate_type& lt,
       int& li,
       Face_handle h =Face_handle() );

Vertex_handle
insert(Point p)

void
remove(Vertex_handle v);
```

The triangulation class (cont'd)

Insertion

```
Vertex_handle insert (Point p)
{
   Locate_type lt; int li;
   Face_handle loc = locate(p, lt, li);
   switch(lt) {
   case VERTEX            : return f->vertex(li);
   case EDGE              :return insert_in_edge( p, loc,li);
   case FACE              :return insert_in_face(v,loc);
   case OUTSIDE_CH        :return insert_outside_ch(p,loc);
   case OUTSIDE_AH        :return insert_outside_ah(p);
   }
}
```

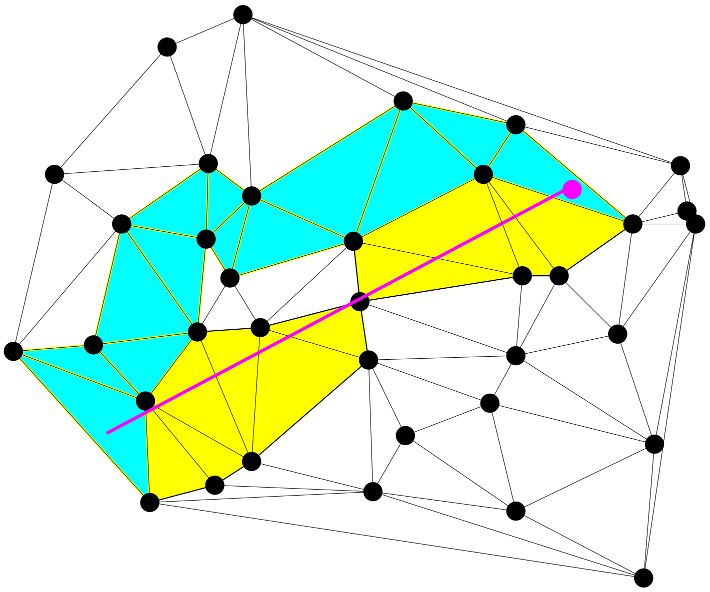


Triangulation algorithmic

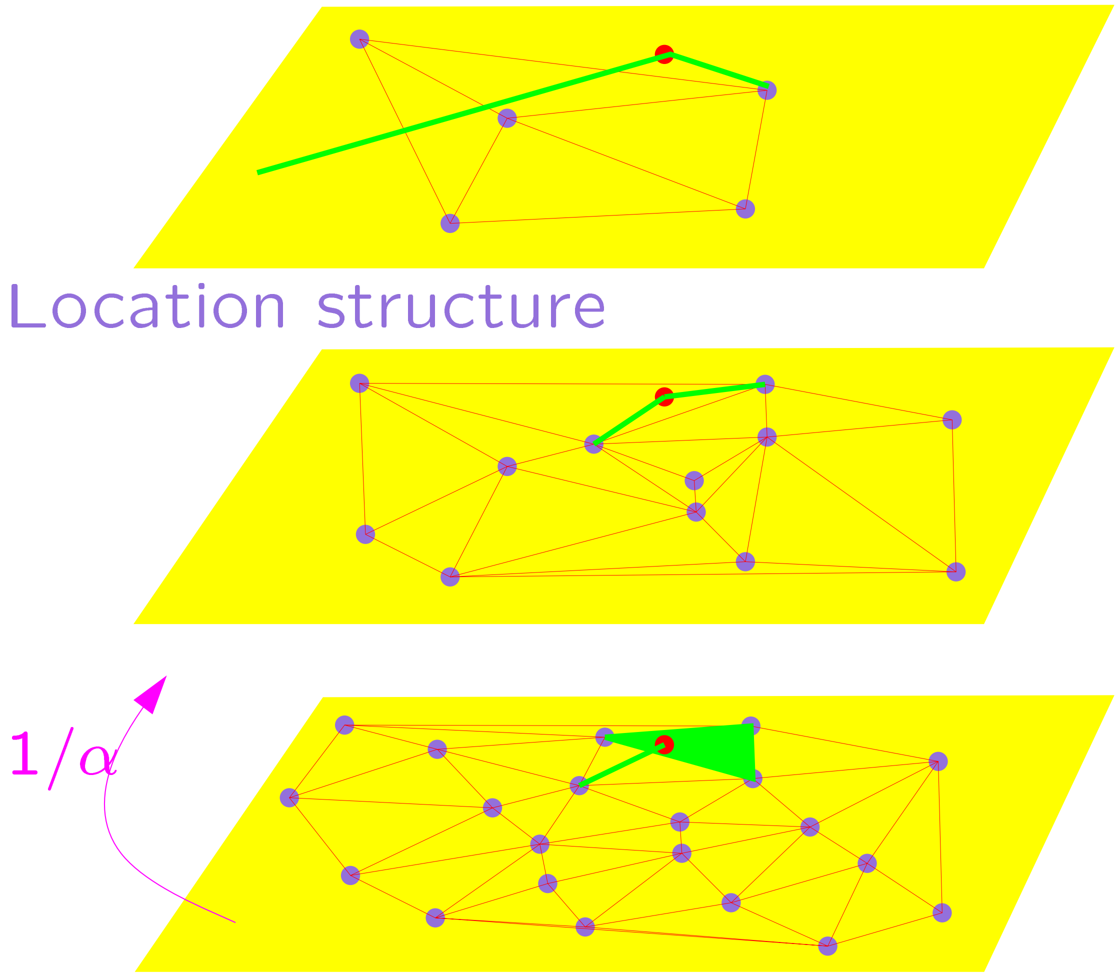
Triangulations in Cgal are built through incremental on-line insertion of vertices. The main algorithmic issue, given this strategy concerns the point location. Cgal uses and offers different algorithms :

- linewalk
- Zigzag walk
- jump and walk strategy
- the Delaunay hierarchy

Linewalk
Zigzag walk

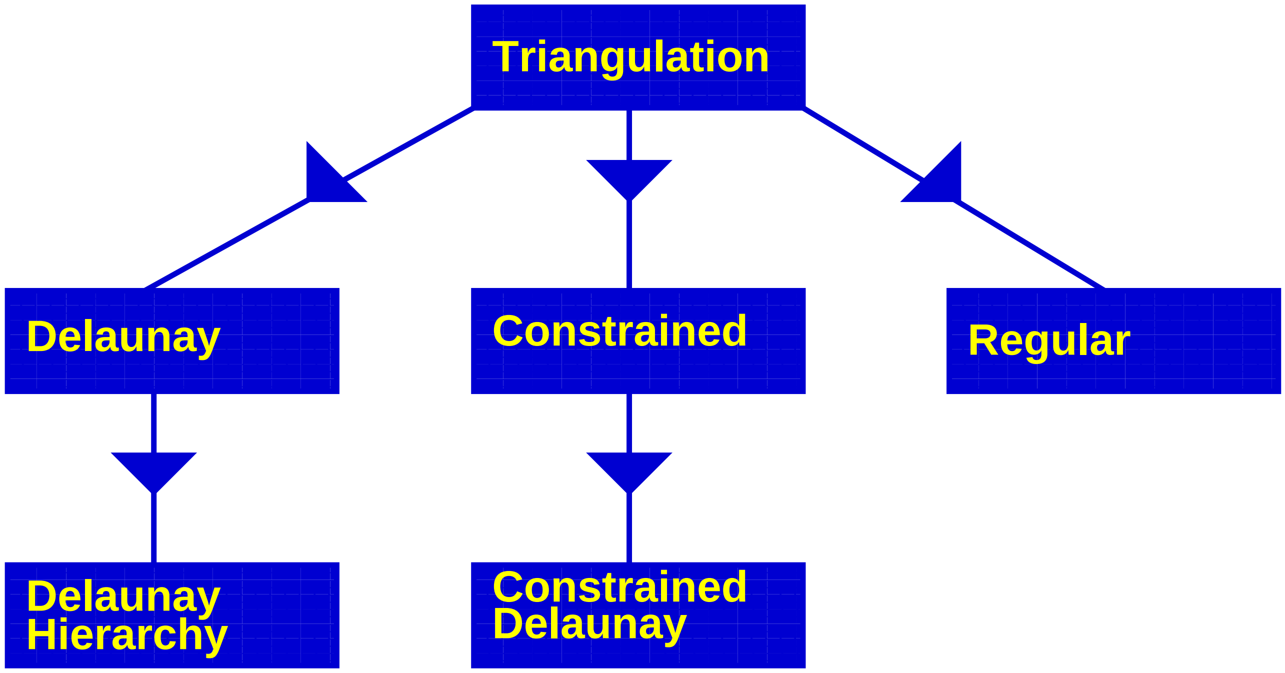


The Delaunay hierarchy



Triangulations in CGAL

The derivation tree



Exemple 1

```
#include <CGAL/Cartesian.h>
#include <CGAL/Triangulation_2.h>

using namespace CGAL;
using namespace std;

typedef Cartesian<double> Ker;
typedef Triangulation_2<Ker> Triangulation;
typedef Triangulation::Vertex_circulator V_cir;
typedef Ker::Point_2 Point;

int main()
{
    Triangulation t();
    Point p;
    while (cin >> p) t.insert(p);

    V_circ vc = t.incident_vertices(t.infinite_vertex());
    V_circ done(vc);
    if (vc != NULL) {
        do cout << vc->point();
        while(++vc != done);
    }
}
```

Example 2 : Triangulation with colored faces

```
#include <CGAL/basic.h>
#include <CGAL/Cartesian.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Triangulation_2.h>


using namespace CGAL;
using namespace std;

/* A facet with a color member variable. */
template < class Gt >
My_face_base<Gt>
      : public Triangulation_face_base_2<Gt>
{
   typedef Triangulation_face_base_2<Gt> Base;
   Color color;
   My_face_base() : Base() {}
   My_face_base(void* v0, void* v1, void* v2)
         : Base(v0,v1,v2) { }
   My_face_base(void* v0, void* v1, void* v2,
                void* n0, void* n1, void* n2)
      : Base(v0,v1,v2,n0,n1,n2) { }
};
```

Example 2 : Triangulation with colored faces (cont'd)

```
typedef Cartesian<double>                              Gt;
typedef Triangulation_vertex_base_2<Gt>                Vb;
typedef My_face_base<Gt>                               Fb;
typedef Triangulation_default_ds_2<Gt,Vb,Fb >Tds;
typedef Triangulation_2<Gt,Tds>                        Triangulation;
typedef Triangulation::Face_handle                     Face_handle;
typedef Triangulation::Face_iterator                   Face_iterator;
typedef Rp::Point_2                                    Point;
int main()
{
    Triangulation t;
    Point p;
    while (std::cin >> p) t.insert(p);

    Face_iterator fc = t.faces_begin();
    for ( ;fc != t.faces_end(); ++fc) {
        fc->color = CGAL::BLUE;
    }
    cin >> p;
    Face_handle fh = t.locate(p);
    fh->color = CGAL::RED;
}
```