Triangulations in CGAL

Monique Teillaud

INRIA SOPHIA-ANTIPOLIS

Overview

Specifications

- Definition
- Various triangulations
- Functionalities

Geometry vs. combinatorics

Representation

Software design

- The traits class
- The triangulation data structure

Using the Triangulation packages

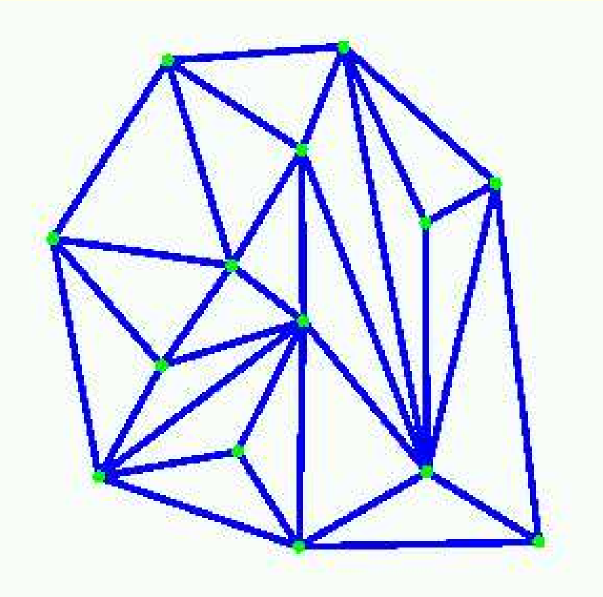
- User Manual
- Reference Manual
- Examples

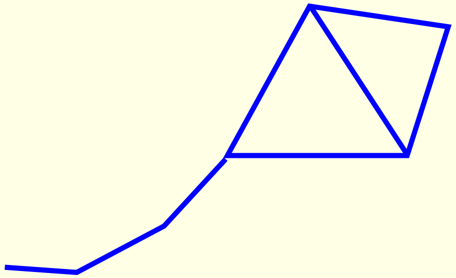
More flexibility

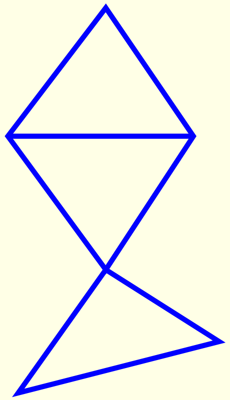
Specifications

Definition

A 2d- (3d-) triangulation is a set of triangles (tetrahedra) such that:
- the set is edge- (facet-) connected
- two triangles (tetrahedra) are either disjoint or share (a facet or) an edge or a vertex.

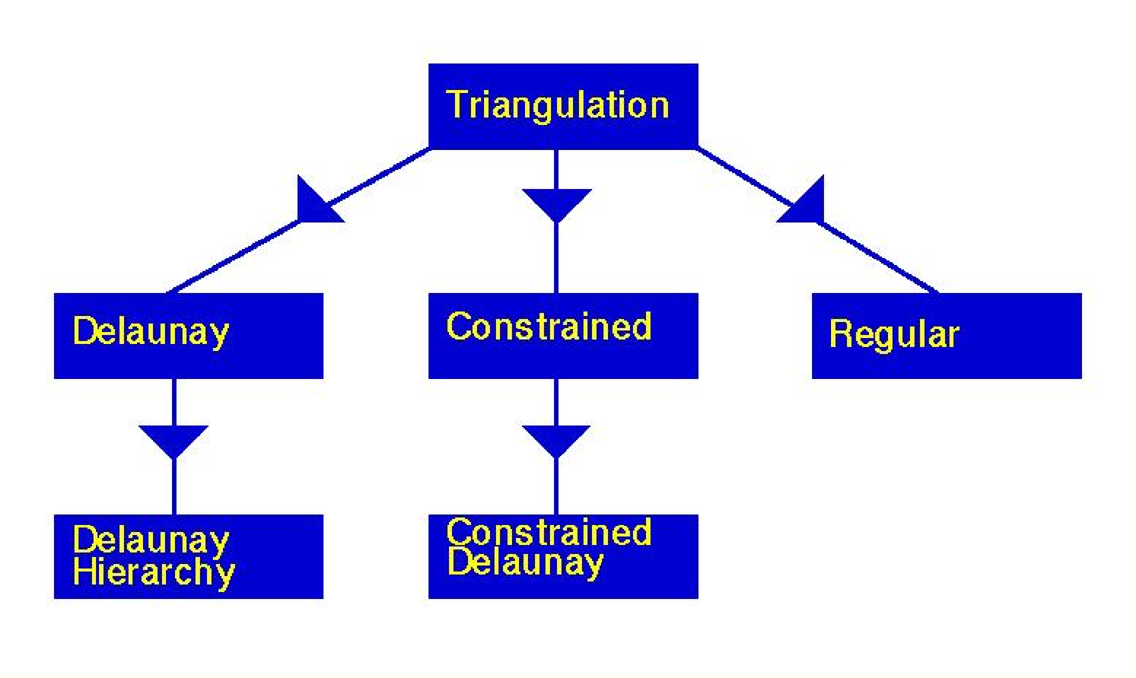
yes : 



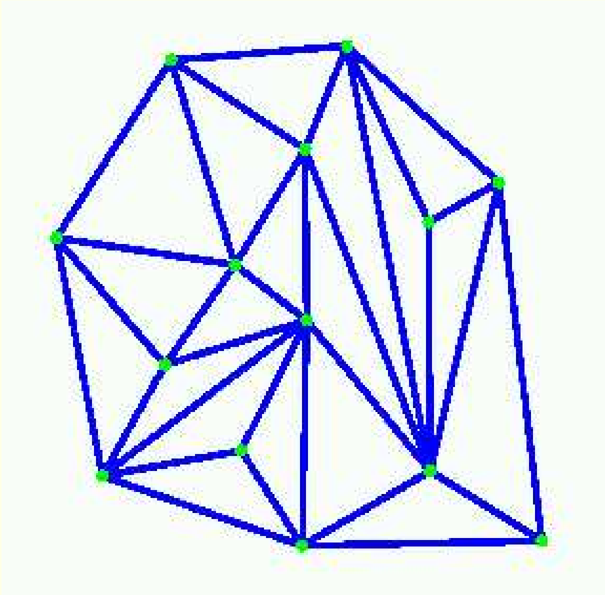
no : 

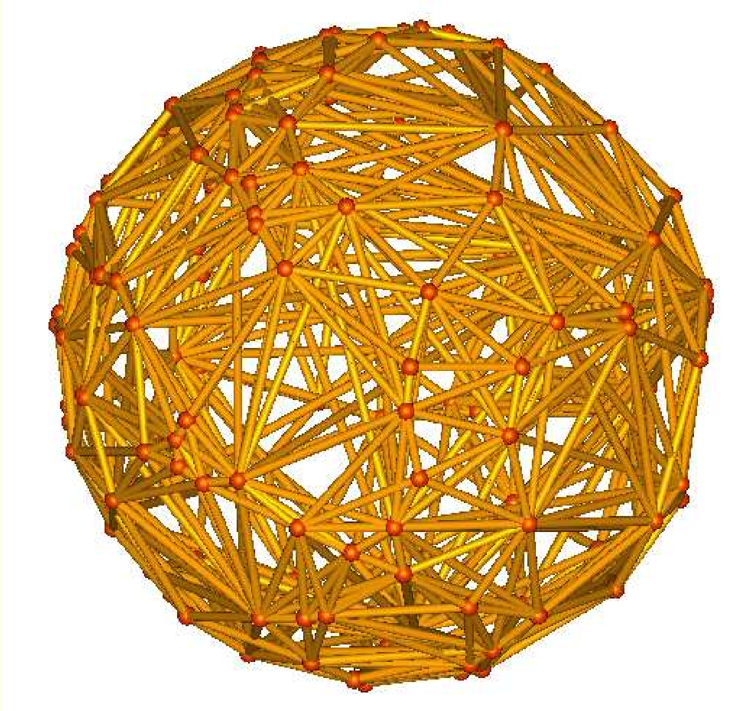
Various triangulations

2D, 3D Basic triangulations
2D, 3D Delaunay triangulations
2D, 3D Regular triangulations
2D Constrained triangulations
2D Constrained Delaunay triangulations

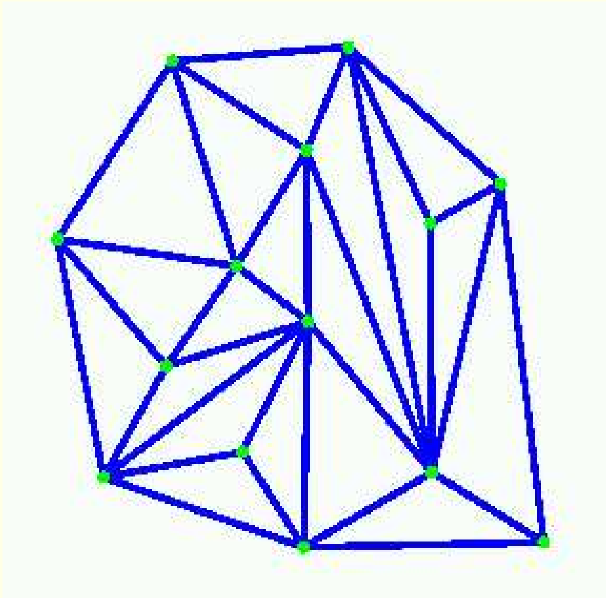


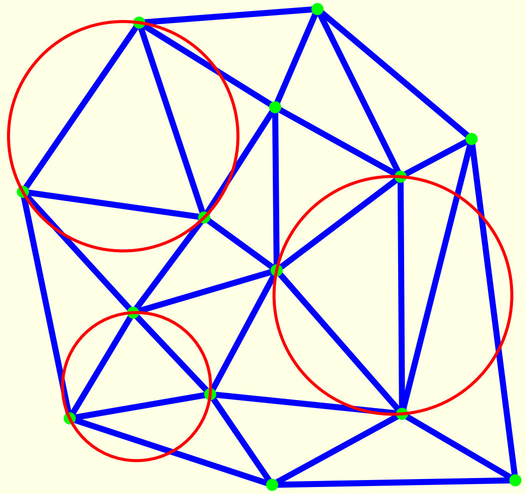
2d - 3d





Basic and Delaunay triangulations





Basic triangulations : lazy incremental construction

Delaunay triangulations: empty circle property

Regular triangulations

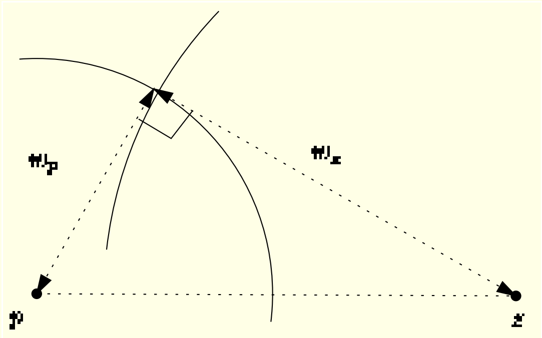
weighted point $p^{(w)} = (p, w_p), p \in \mathbb{R}^3, w_p \in \mathbb{R}$

$p^{(w)} = (p, w_p) \simeq$ sphere of center p and radius w_p.

power product between $p^{(w)}$ and $z^{(w)}$

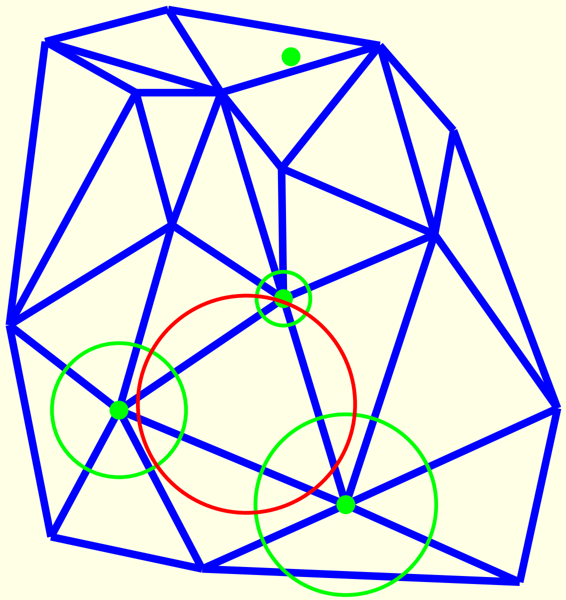
$$\Pi(p^{(w)}, z^{(w)}) = \|p - z\|^2 - w_p - w_z$$

$p^{(w)}$ and $z^{(w)}$ **orthogonal** iff $\Pi(p^{(w)}, z^{(w)}) = 0$



Power sphere of 4 weighted points in $\mathbb{R}^3$ = unique common orthogonal weighted point.

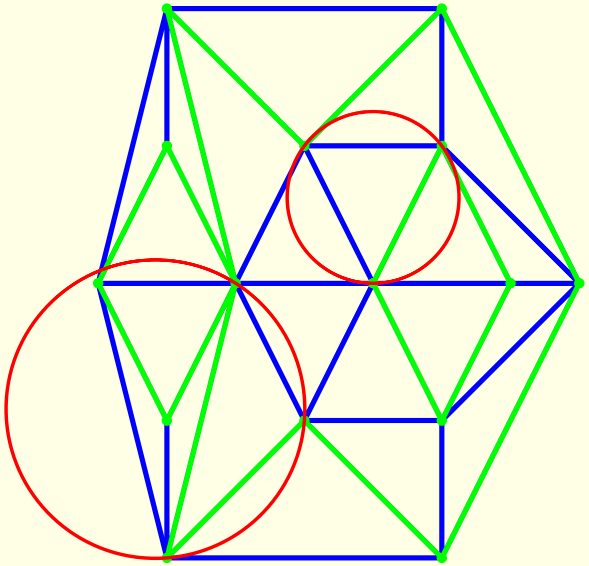
$z^{(w)}$ is **regular** iff $\forall p^{(w)}, \Pi(p^{(w)}, z^{(w)}) \geq 0$



Regular triangulations: generalization of Delaunay triangulations to weighted points. Dual of the **power diagram**.

The power sphere of all simplices is regular.

Constrained [Delaunay] triangulations



Constrained Delaunay triangulations

Constrained empty circle property : the circumscribing circle encloses no vertex $visible$ from the interior of the triangle.

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

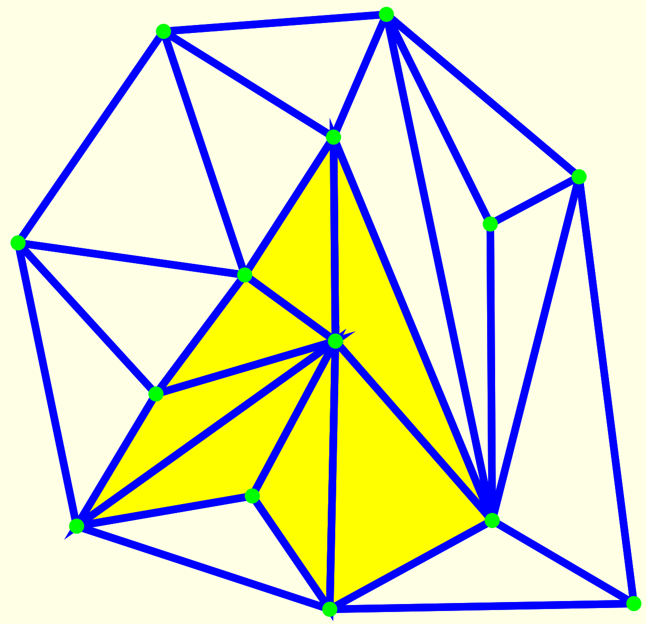
Traversal of a triangulation

Iterators

All_faces_iterator
All_vertices_iterator
All_edges_iterator

Circulators

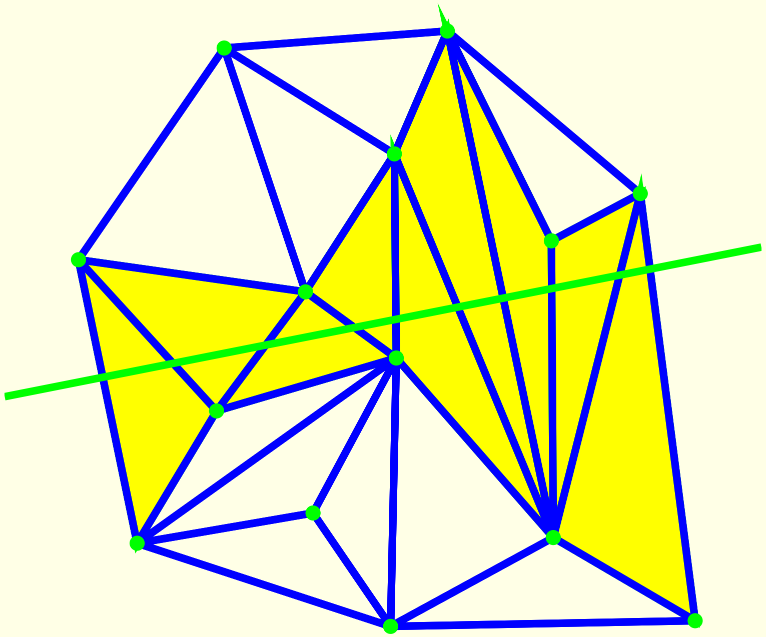
Face_circulator : faces incident to a vertex
Edge_circulator : edges incident to a vertex
Vertex_circulator : incident to a vertex



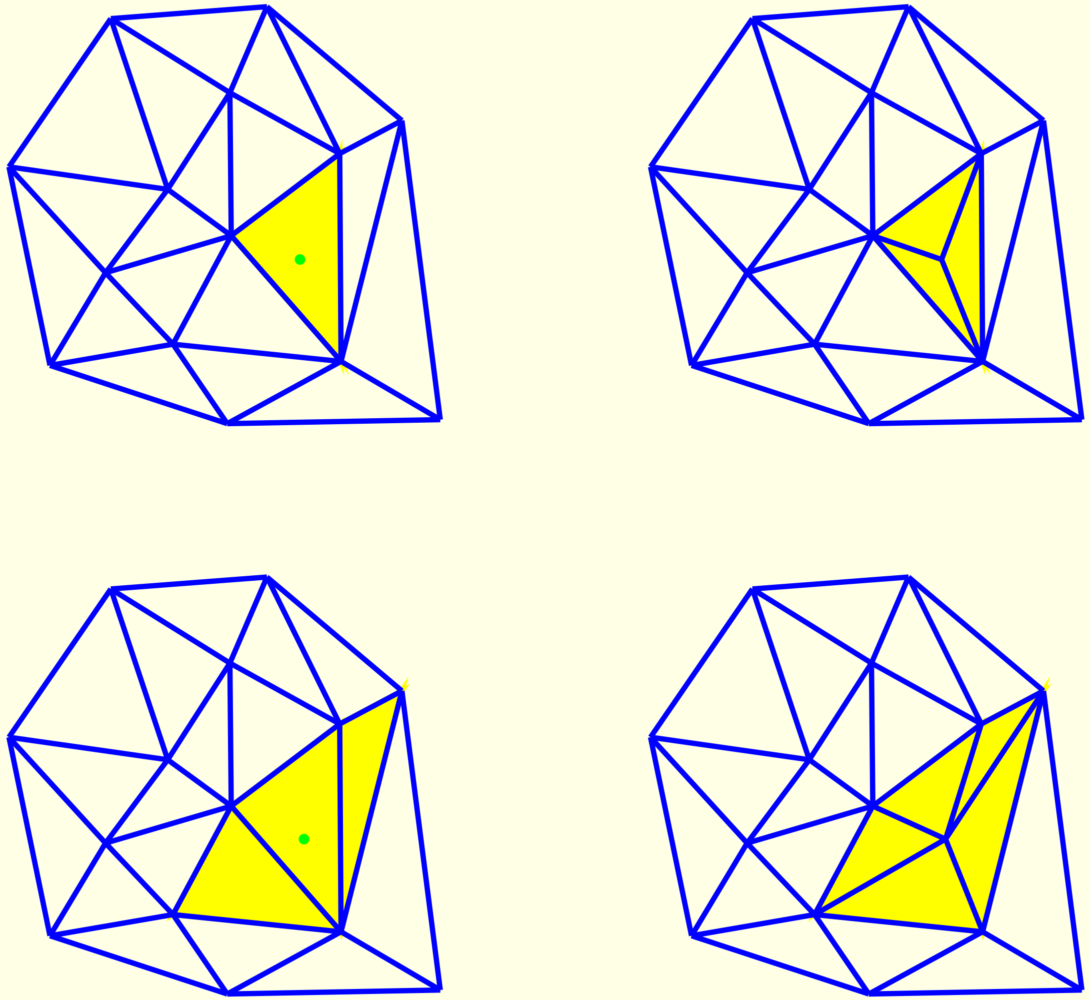
```
All_vertices_iterator vit;
for (vit = T.finite_vertices_begin();
     vit != T.finite_vertices_end(); ++vit)
{ ... }
```

Traversal of a triangulations cont'd

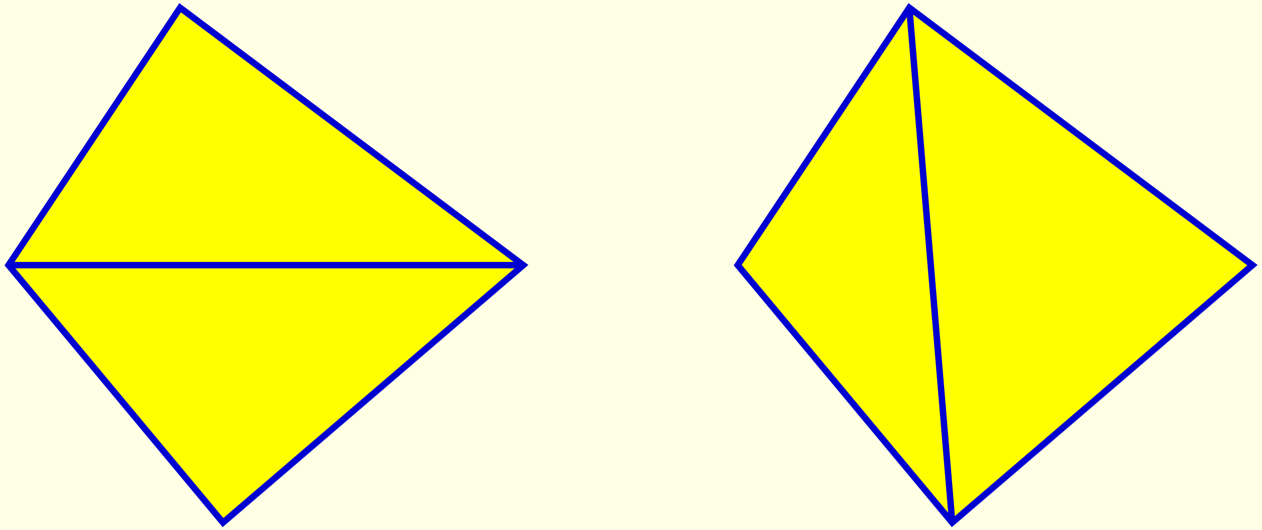
Line_face_circulator



Point location, insertion, removal



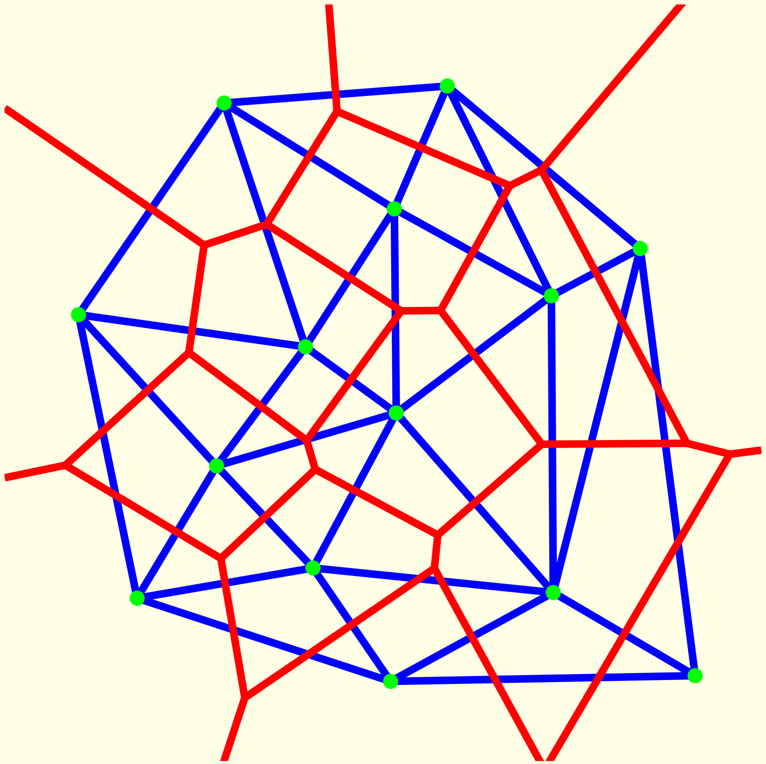
Flip



Additional functionalities for Delaunay triangulations

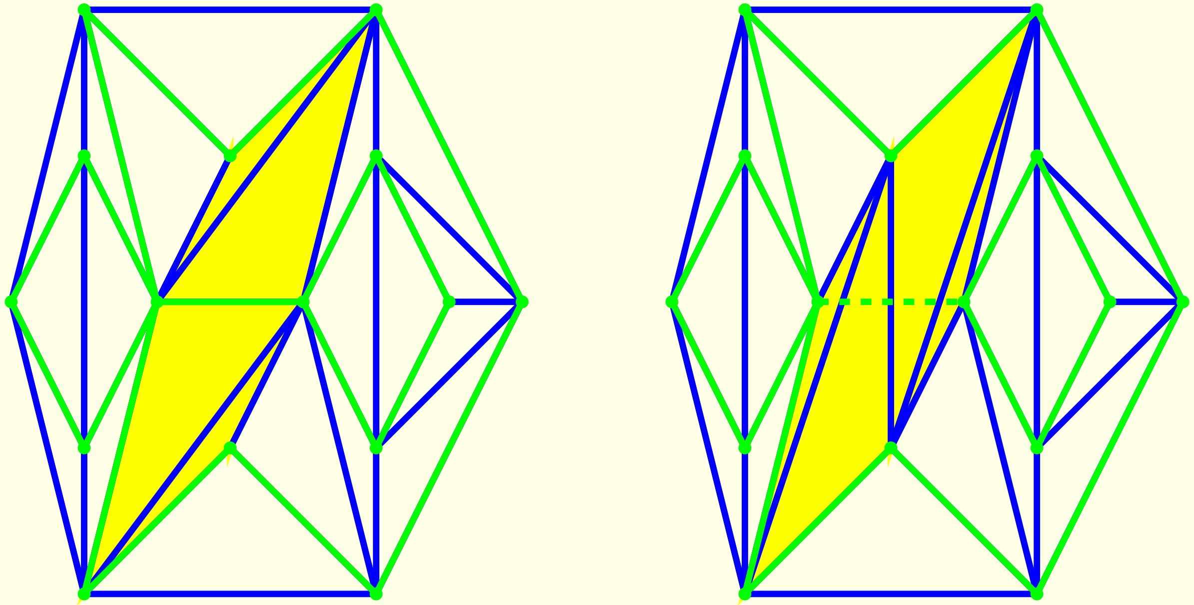
Nearest neighbor queries

Voronoi diagram



Additional functionalities for [Delaunay] constrained triangulations

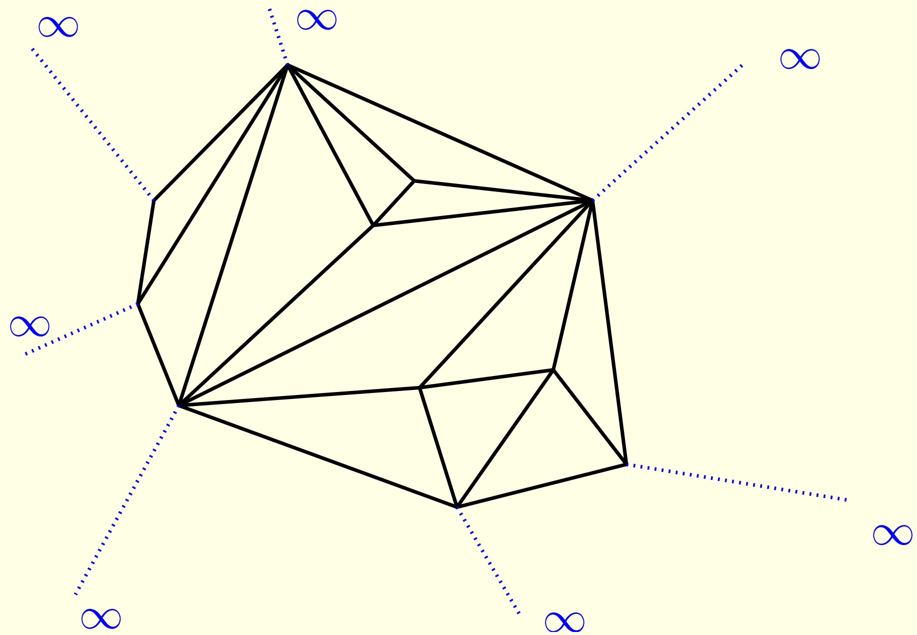
Insertion and deletion of constraints



Geometry vs. Combinatorics

Infinite vertex

Triangulation of a set of points =
partition of the **convex hull** into simplices.



Addition of an **infinite vertex**
$\longrightarrow$ “triangulation” of the outside
of the convex hull.

2D:
- Any face is a triangle.
- Any edge is incident to two faces.

Triangulation of $\mathbb{R}^d$
$\simeq$
Triangulation of the topological **sphere** $\mathcal{S}^d$.

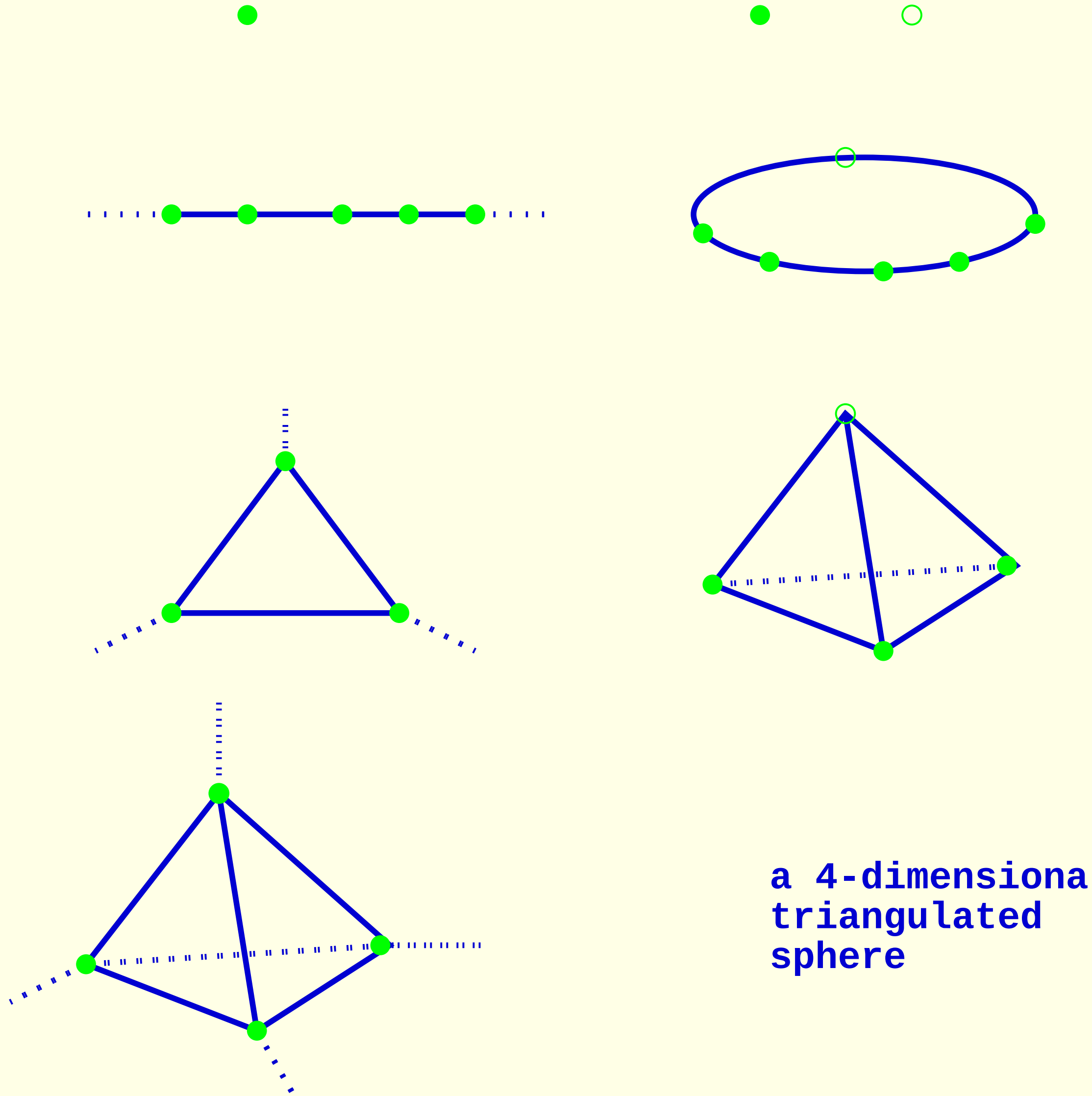
Dimensions

dim 0

dim 1

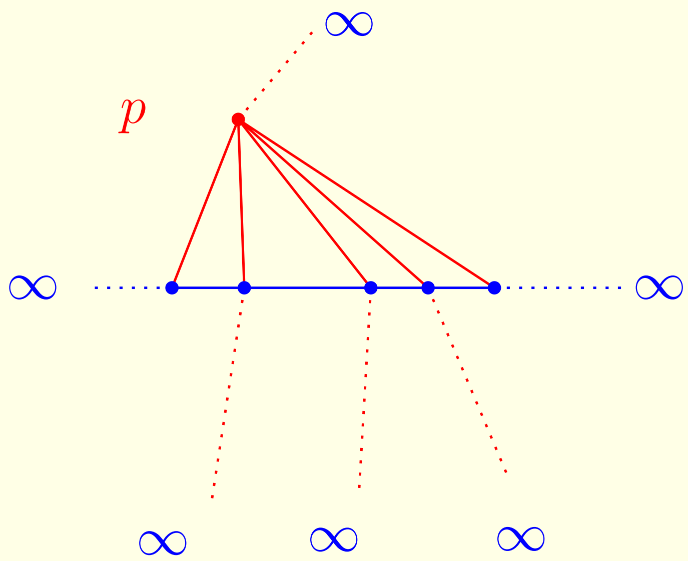
dim 2

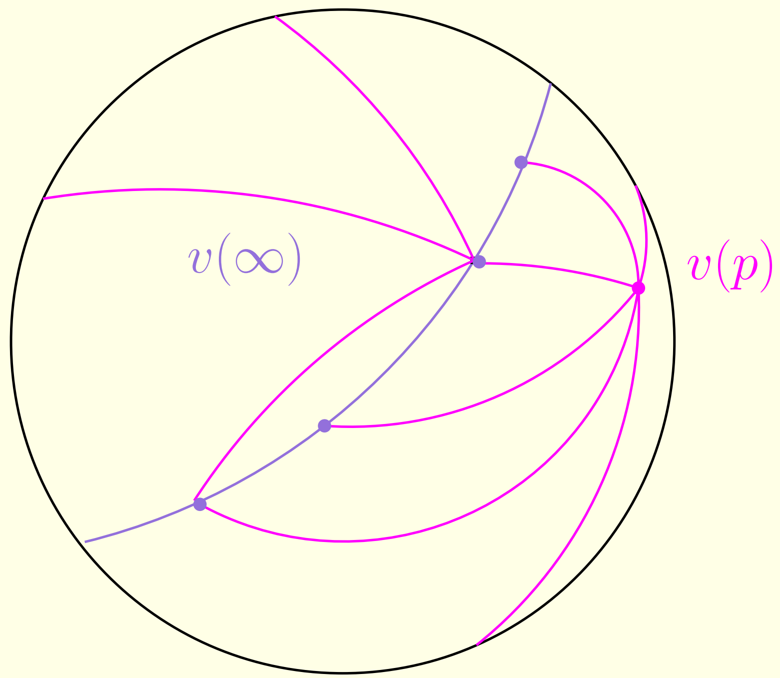
dim 3



Adding a point outside the current affine hull:

From $d = 1$ to $d = 2$

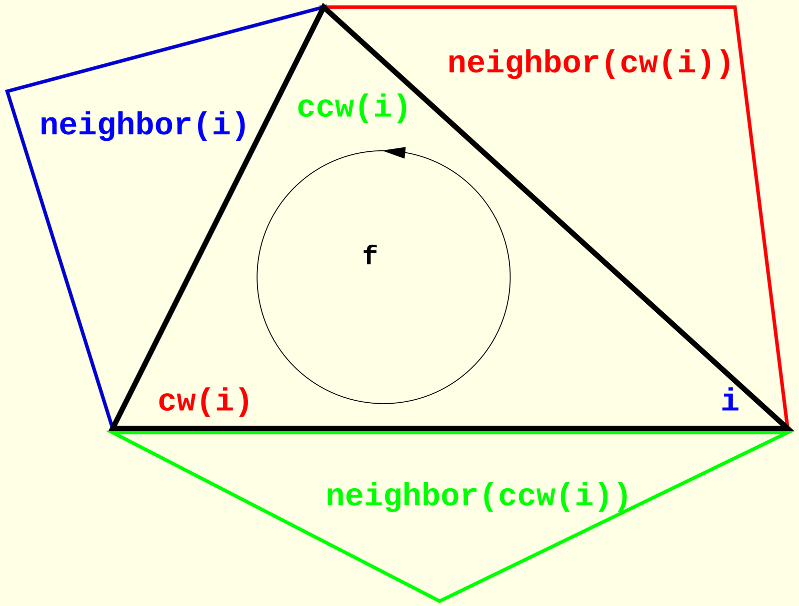




Representation

2D

Based on faces and vertices.



Vertex

Face_handle v_face

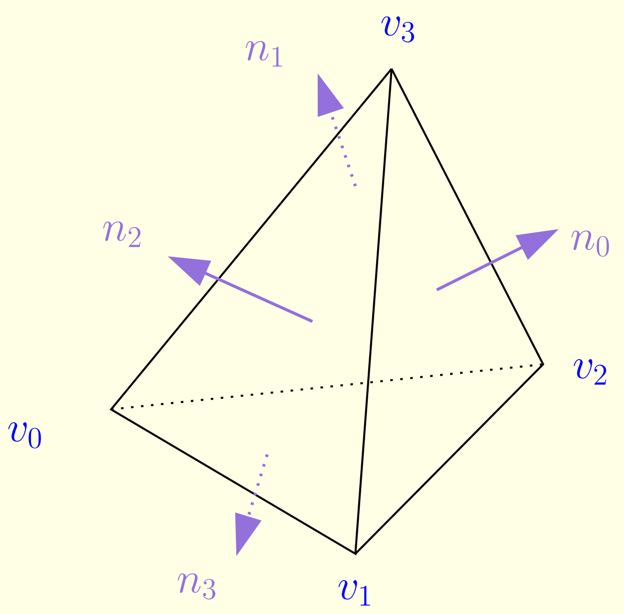
Face

Vertex_handle $vertex[3]$

Face_handle $neighbor[3]$

Edges are implicit: std::pair< f, i >
where f = one of the two incident faces.

3D



Vertex

Cell_handle v_cell

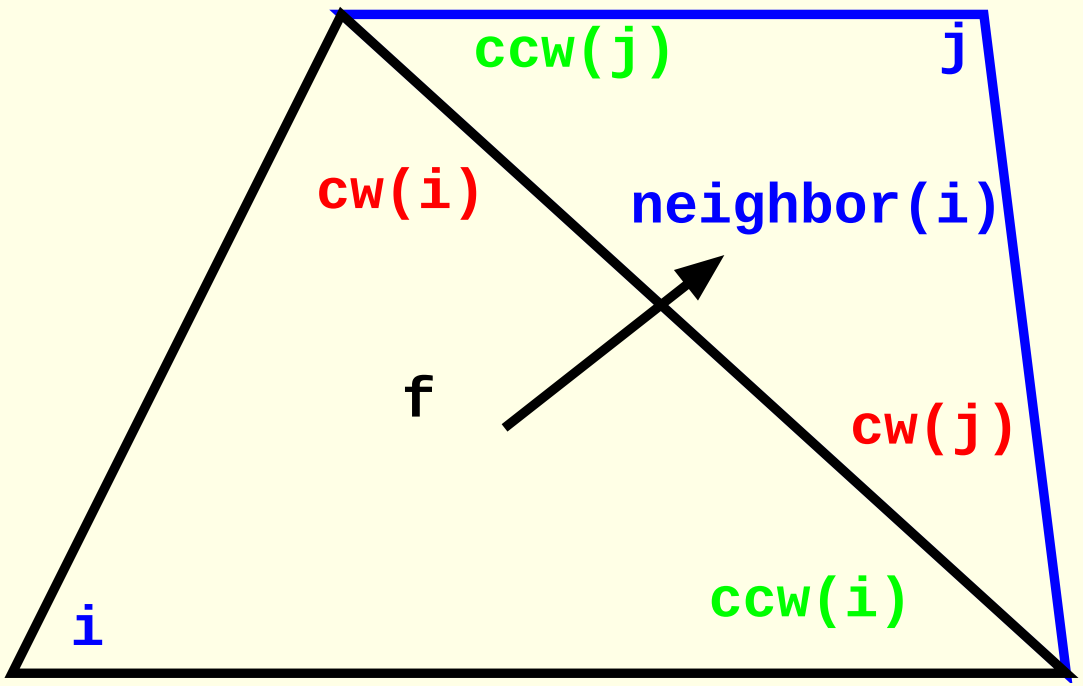
Cell

Vertex_handle $vertex[4]$
Cell_handle $neighbor[4]$

Faces are implicit: `std::pair<` c, i `>`
where c = one of the two incident cells.

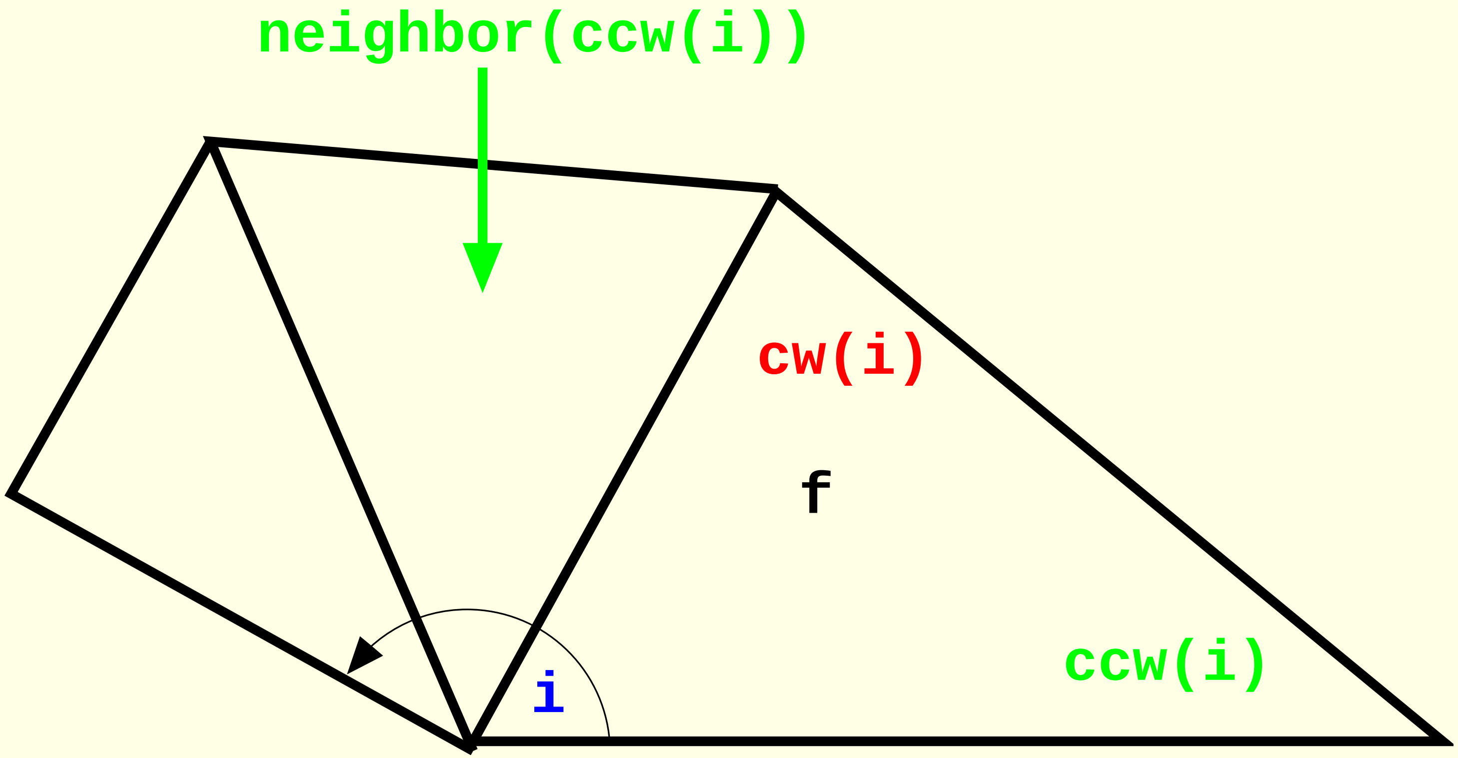
Edges are implicit: `std::pair<` u, v `>`
where u, v = vertices.

From one face to a another

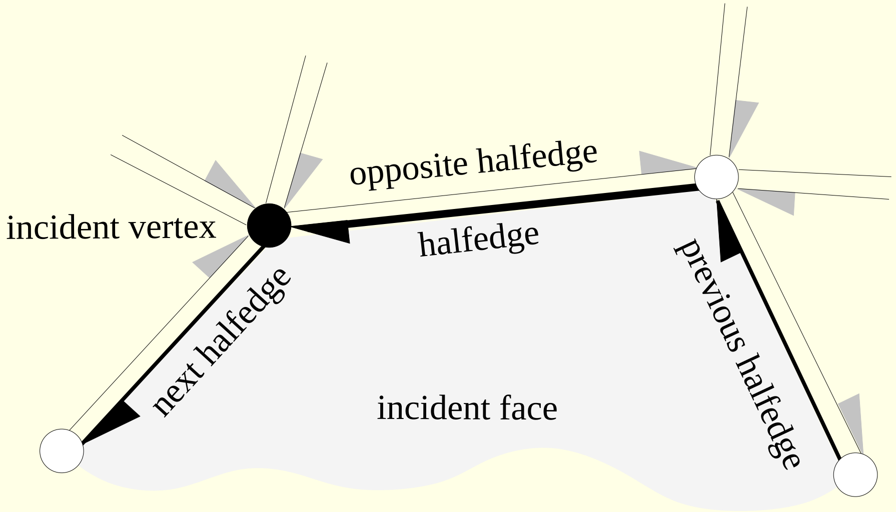



```
n = f->neighbor(i)
j = n->index(f)
```

2D - Around a vertex



Doubly Connected Edge List



Vertex

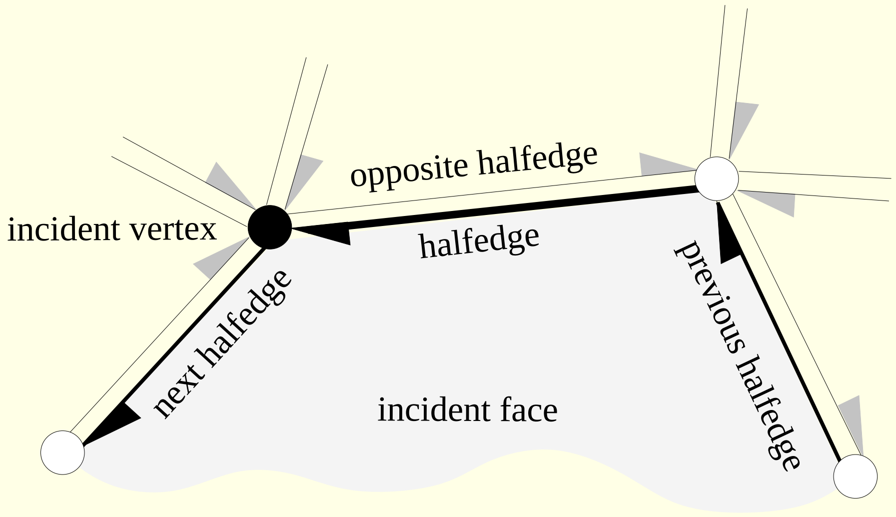
Halfedge* vhe

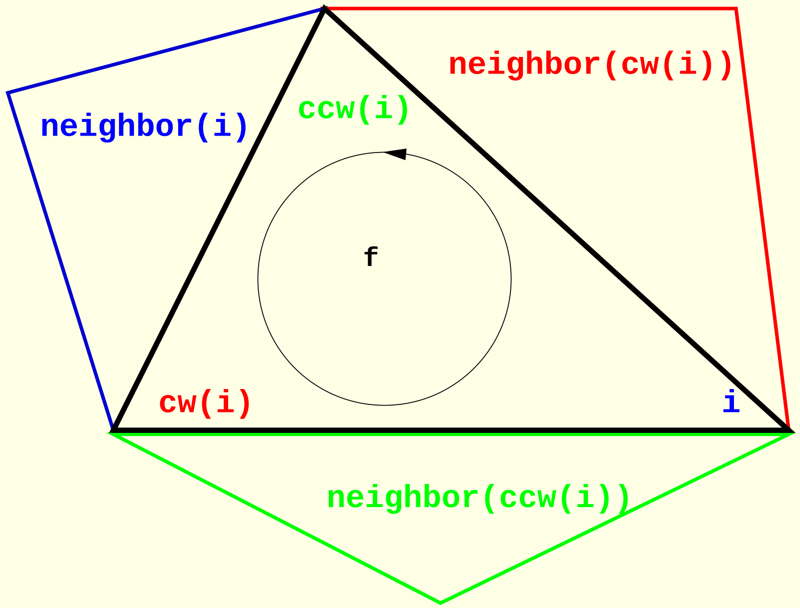
Face

Halfedge* fhe

Halfedge

Face * $left$
Vertex* $source$
Halfedge* $opposite$
Halfedge* $next$
Halfedge* $prev$





n	vertices
$3n - 6$	edges
$2n - 4$	faces

	DCEL	CGAL TDS
Vertices	n	n
Edges	$4 \times 2 \times (3n - 6)$	
Faces	$(2n - 4)$	$6 \times (2n - 4)$
Total	$27n$	$13n$

Software Design

“Traits” classes

convex_hull_2<InputIterator, OutputIterator, **Traits**>
Polygon_2<**Traits**, Container>
Polyhedron_3<**Traits**, HDS>
Triangulation_2<**Traits**, TDS>
Triangulation_3<**Traits**, TDS>
Min_circle_2<**Traits**>
Range_tree_k<**Traits**>
. . .

Geometric traits classes provide:
Geometric objects + predicates + constructors

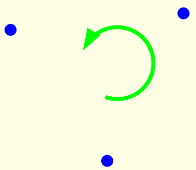
Flexibility:

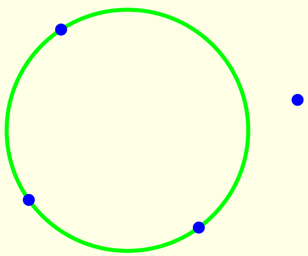
- The **Kernel** can be used as a traits class for several algorithms
- Otherwise: **Default traits classes** provided
- The **user** can plug his own traits class

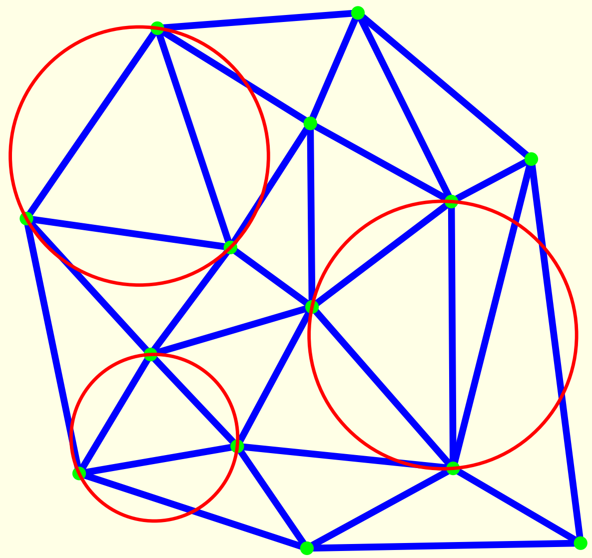
2D Delaunay Triangulation

Requirements for a traits class:

- 2D point
- orientation test, in_circle test





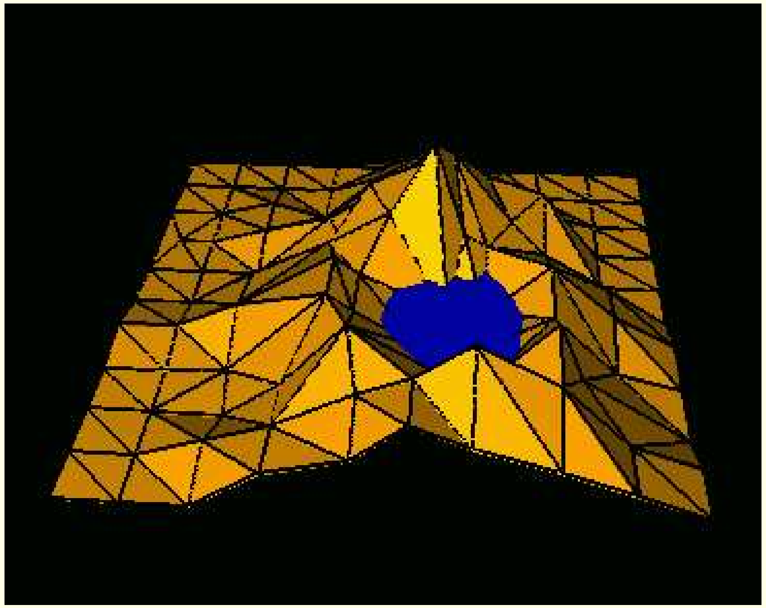


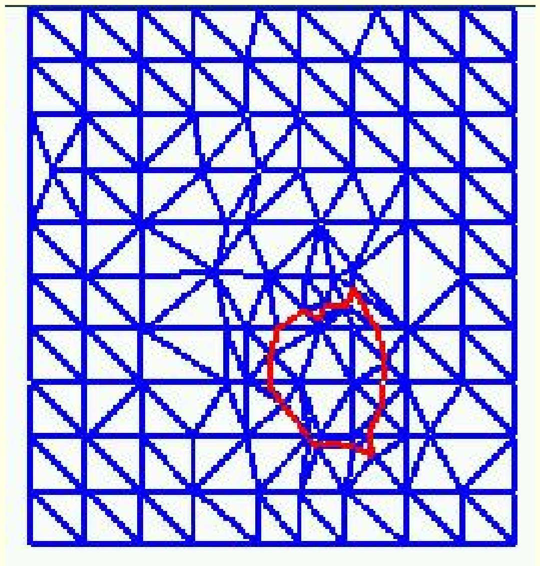
```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2< K > Delaunay;
```

Playing with traits classes

- 3D points: coordinates (x, y, z)
- orientation, in_circle: on x and y coordinates

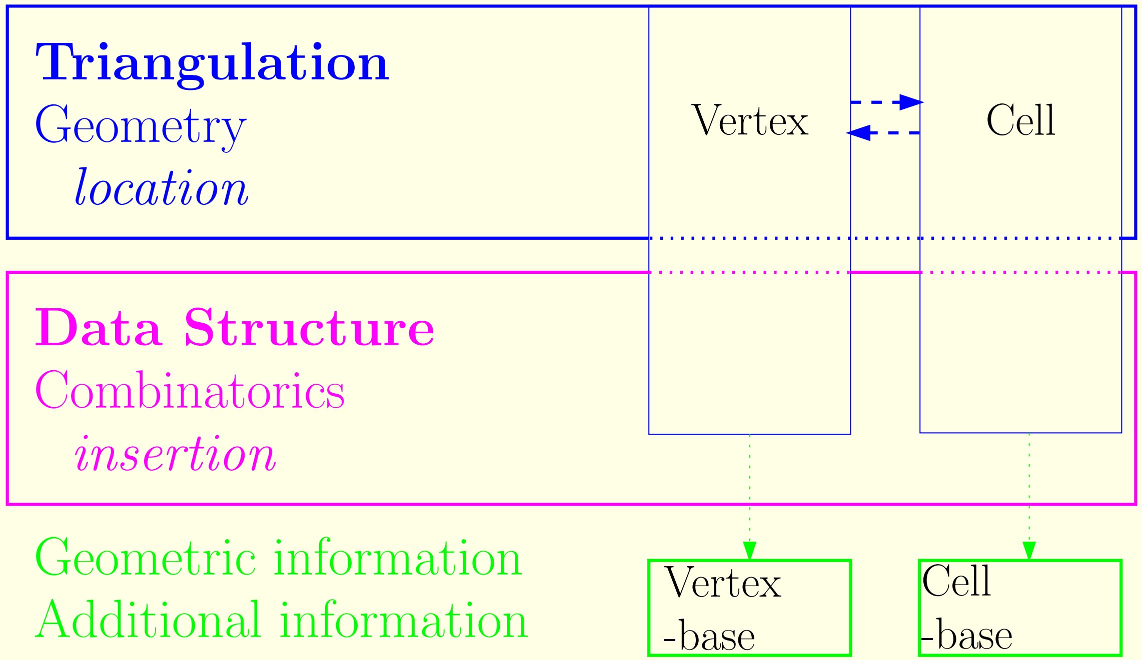
typedef CGAL::Exact_predicates_inexact_constructions_kernel **K**;
typedef CGAL::**Triangulation_euclidean_traits_xy_3**< **K** > **Traits**;
typedef CGAL::**Delaunay_triangulation_2**< **Traits** > Terrain;





Layers

Triangulation_3< Traits, **TDS** >



Triangulation_data_structure_2< Vb, Cb> ;

Vb and Cb have default values.

The base level

Concepts **VertexBase** and **CellBase**.

Provide

- Point + access function + setting

- incidence and adjacency relations (access and setting)

Several models, parameterised by the **traits** class.

Using the Triangulation packages

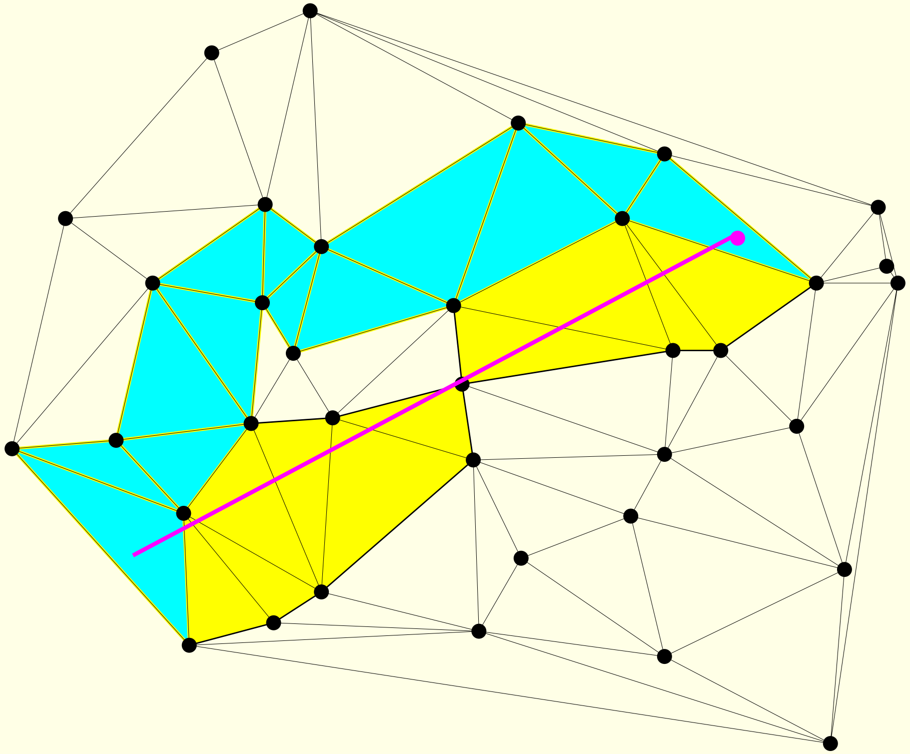
A look at the User Manual

Representation, classes,. . .

A look at the Reference Manual

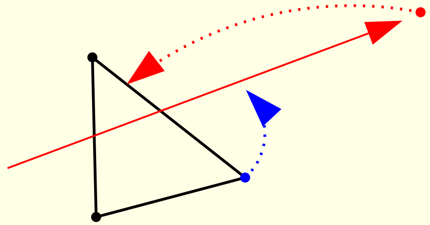
Locate_type

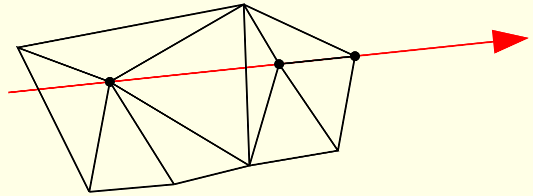
locate



- Along a straight line

2 (/3) orientation tests
per triangle (/tetrahedron)

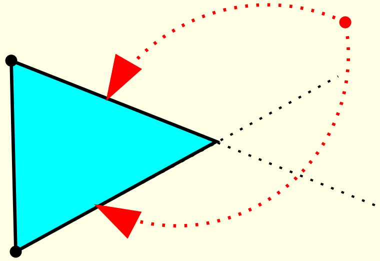


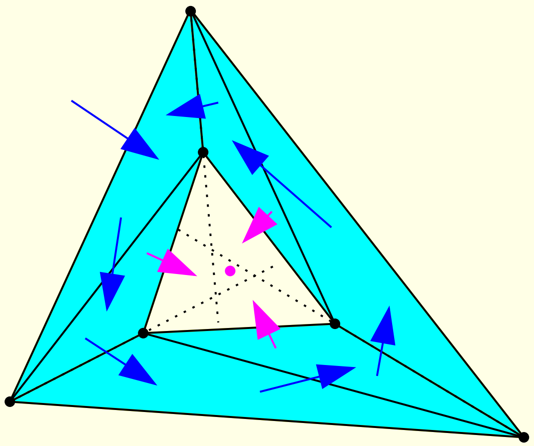


degenerate cases

- By visibility

$<$ 1.5 (/2) tests per triangle (/tetrahedron)





Breaking cycles: random choice of the neighbor

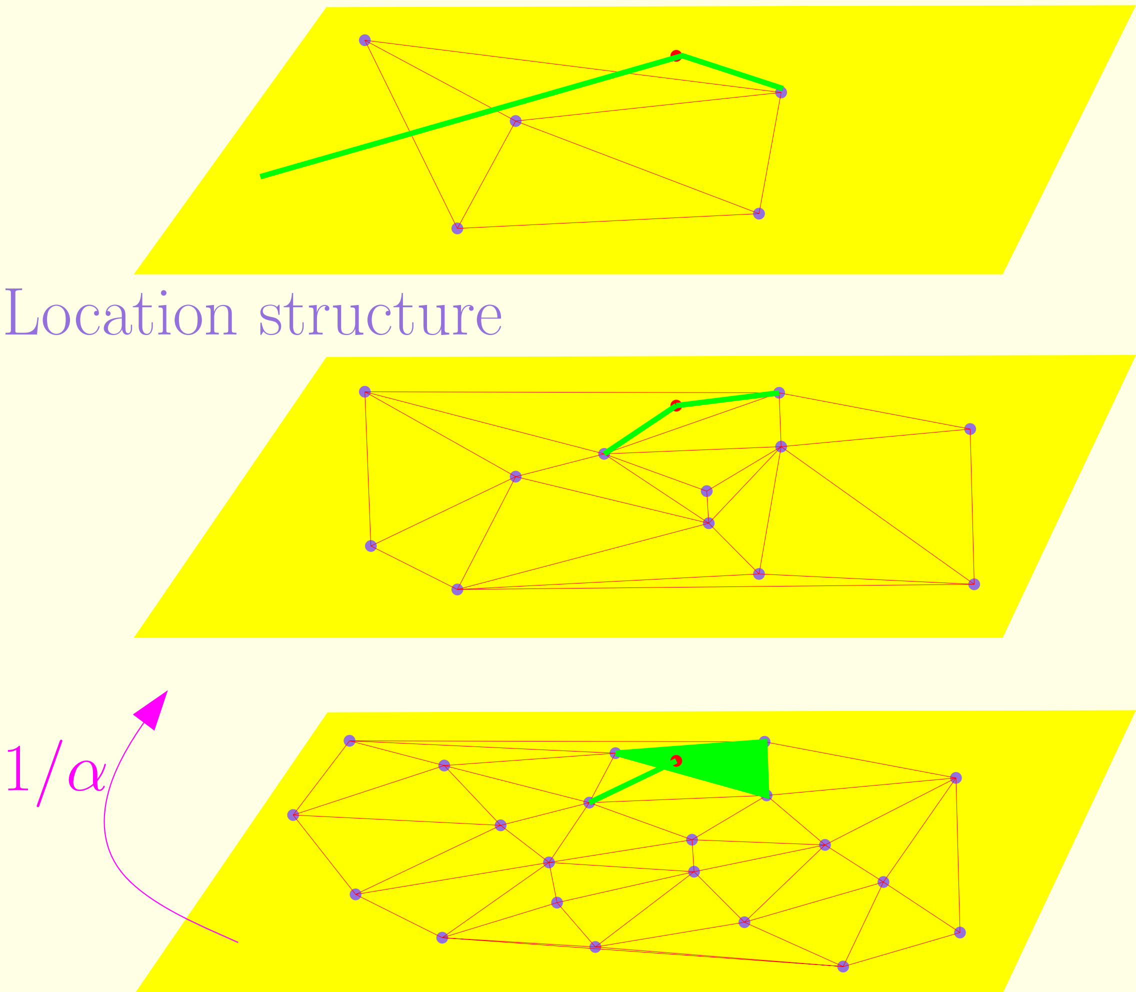
First example

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_3.h>

#include <iostream>
#include <fstream>
#include <cassert>
#include <list>
#include <vector>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_3<K>      Triangulation;

typedef Triangulation::Cell_handle    Cell_handle;
typedef Triangulation::Vertex_handle  Vertex_handle;
typedef Triangulation::Locate_type    Locate_type;
typedef Triangulation::Point          Point;
```

```
int main()
{
  std::list<Point> L;
  L.push_front(Point(0,0,0));
  L.push_front(Point(1,0,0));
  L.push_front(Point(0,1,0));

  Triangulation T(L.begin(), L.end());

  int n = T.number_of_vertices();

  std::vector<Point> V(3);
  V[0] = Point(0,0,1);
  V[1] = Point(1,1,1);
  V[2] = Point(2,2,2);

  n = n + T.insert(V.begin(), V.end());

  assert( n == 6 );
  assert( T.is_valid() );
```

```
  Locate_type lt;
  int li, lj;
  Point p(0,0,0);
  Cell_handle c = T.locate(p, lt, li, lj);
  assert( lt == Triangulation::VERTEX );
  assert( c->vertex(li)->point() == p );

  Vertex_handle v = c->vertex( (li+1)&3 );
  Cell_handle nc = c->neighbor(li);
  int nli;
  assert( nc->has_vertex( v, nli ) );

  std::ofstream oFileT("output",std::ios::out);
  oFileT << T;
  Triangulation T1;
  std::ifstream iFileT("output",std::ios::in);
  iFileT >> T1;
  assert( T1.is_valid() );
  assert( T1.number_of_vertices() == T.number_of_vertices() );
  assert( T1.number_of_cells() == T.number_of_cells() );
  return 0;
}
```

Using the Delaunay Hierarchy



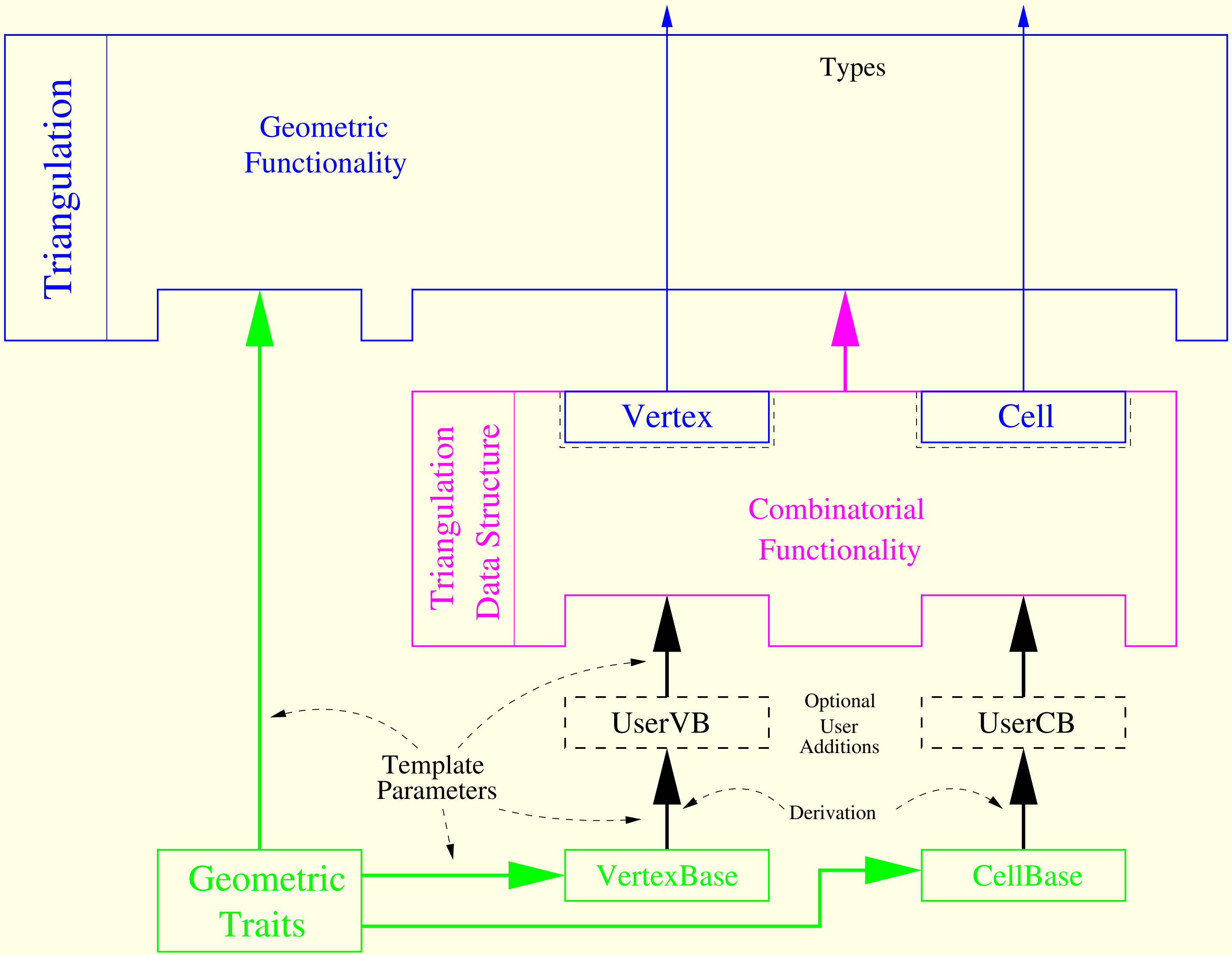
```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_hierarchy_3.h>

#include <cassert>
#include <vector>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_vertex_base_3<K>             Vb;
typedef CGAL::Triangulation_hierarchy_vertex_base_3<Vb>  Vbh;
typedef CGAL::Triangulation_data_structure_3<Vbh>        Tds;
typedef CGAL::Delaunay_triangulation_3<K,Tds>            Dt;
typedef CGAL::Triangulation_hierarchy_3<Dt>              Dh;

typedef Dh::Vertex_iterator Vertex_iterator;
typedef Dh::Vertex_handle   Vertex_handle;
typedef Dh::Point           Point;
```

```
int main()
{
  Dh T;

      // insertion of points on a 3D grid
  std::vector<Vertex_handle> V;

  for (int z=0 ; z<5 ; z++)
    for (int y=0 ; y<5 ; y++)
      for (int x=0 ; x<5 ; x++)
  V.push_back(T.insert(Point(x,y,z)));

  assert( T.is_valid() );
  assert( T.number_of_vertices() == 125 );
  assert( T.dimension() == 3 );

      // removal of the vertices in random order
  std::random_shuffle(V.begin(), V.end());

  for (int i=0; i<125; ++i)
    T.remove(V[i]);
```

```
  assert( T.is_valid() );
  assert( T.number_of_vertices() == 0 );

  return 0;
}
```

More flexibility

Changing the Vertex_base and the Cell_base



First option: Triangulation_vertex_base_with_info_3

When the additional information does not depend on the TDS.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_with_info_3.h>
#include <CGAL/IO/Color.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_vertex_base_with_info_3<CGAL::Color,K> Vb;

typedef CGAL::Triangulation_data_structure_3<Vb>             Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds>               Delaunay;

typedef Delaunay::Point   Point;

int main()
{
  Delaunay T;
```

```
  T.insert(Point(0,0,0));
  T.insert(Point(1,0,0));
  T.insert(Point(0,1,0));
  T.insert(Point(0,0,1));
  T.insert(Point(2,2,2));
  T.insert(Point(-1,0,1));

   // Set the color of finite vertices of degree 6 to red.
  Delaunay::Finite_vertices_iterator vit;

  for (vit = T.finite_vertices_begin();
       vit != T.finite_vertices_end(); ++vit)

    if (T.degree(vit) == 6)
        vit->info() = CGAL::RED;

  return 0;
}
```

Third option: write new models of the concepts

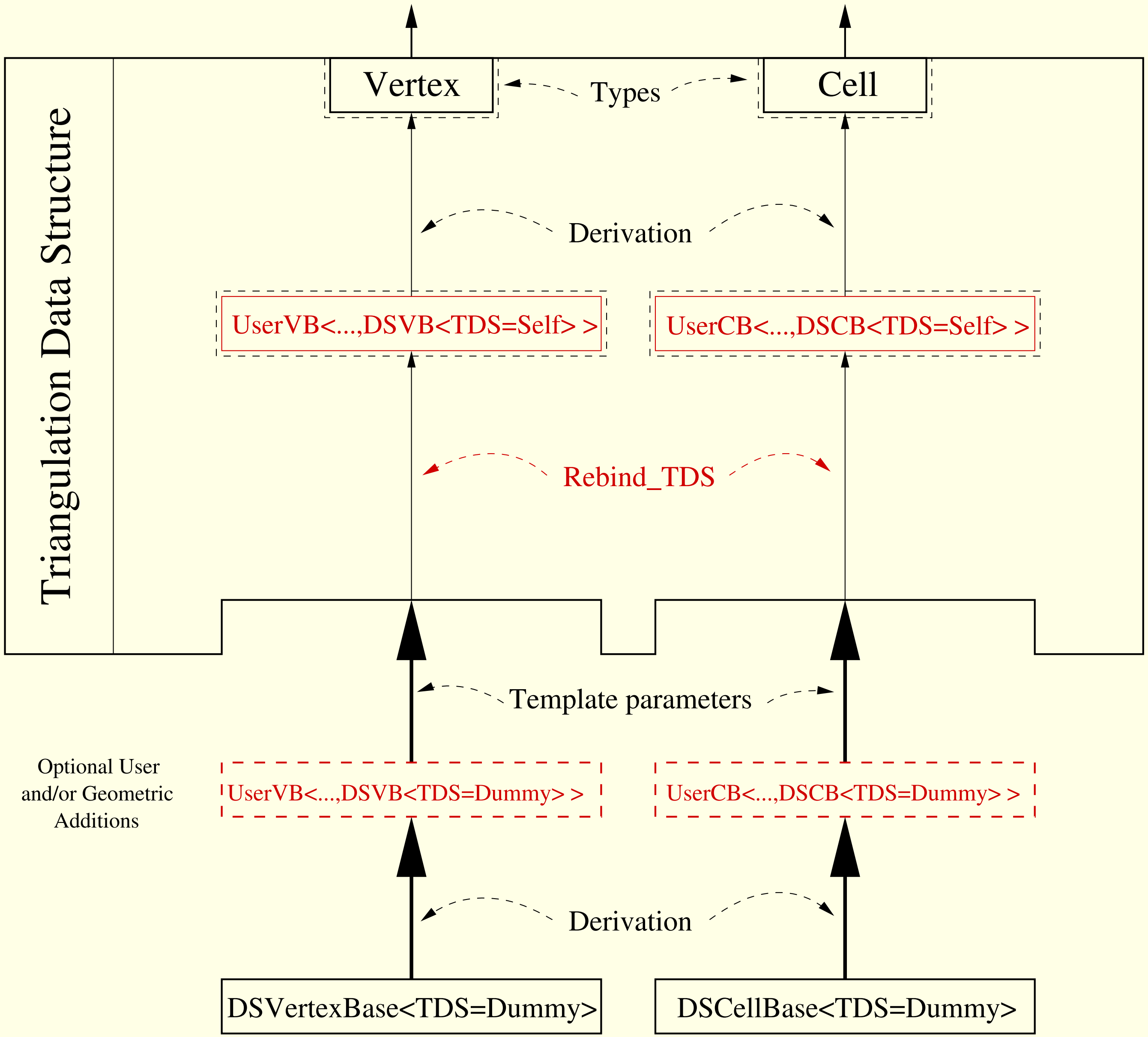
Second option: the "rebind" mechanism

• Vertex and cell base classes: initially given a **dummy TDS** template parameter:

dummy TD provides the types that can be used by the vertex and cell base classes (such as handles).

• inside the TDS itself, vertex and cell base classes are **rebound** to the real TDS type

→ the same vertex and cell base classes are now **parameterized with the real TDS** instead of the dummy one.


Triangulation Data Structure
Vertex
Types
Cell
Derivation
UserVB<...,DSVB<TDS=Self> >
UserCB<...,DSCB<TDS=Self> >
Rebind_TDS
Template parameters
Optional User
and/or Geometric
Additions
UserVB<...,DSVB<TDS=Dummy> >
UserCB<...,DSCB<TDS=Dummy> >
Derivation
DSVertexBase<TDS=Dummy>
DSCellBase<TDS=Dummy>

```
...
template < class GT, class Vb = Triangulation_vertex_base<GT> >
class My_vertex
  : public Vb
{
public:
  typedef typename Vb::Point           Point;
  typedef typename Vb::Cell_handle     Cell_handle;

  template < class TDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other  Vb2;
    typedef My_vertex<GT, Vb2>                             Other;
  };

  My_vertex() {}
  My_vertex(const Point&p)                  : Vb(p) {}
  My_vertex(const Point&p, Cell_handle c) : Vb(p, c) {}
...
}
```

Example

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_3.h>

template < class GT, class Vb=CGAL::Triangulation_vertex_base_3<GT> >
class My_vertex_base
  : public Vb
{
public:
  typedef typename Vb::Vertex_handle  Vertex_handle;
  typedef typename Vb::Cell_handle    Cell_handle;
  typedef typename Vb::Point          Point;

  template < class TDS2 >
  struct Rebind_TDS {
    typedef typename Vb::template Rebind_TDS<TDS2>::Other  Vb2;
    typedef My_vertex_base<GT, Vb2>                        Other;
  };

  My_vertex_base() {}
```

```
  My_vertex_base(const Point& p)
    : Vb(p) {}

  My_vertex_base(const Point& p, Cell_handle c)
    : Vb(p, c) {}

  Vertex_handle   vh;
  Cell_handle     ch;
};

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_data_structure_3<My_vertex_base<K> > Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds>              Delaunay;

typedef Delaunay::Vertex_handle    Vertex_handle;
typedef Delaunay::Point            Point;
```

```
int main()
{
  Delaunay T;

  Vertex_handle v0 = T.insert(Point(0,0,0));
  Vertex_handle v1 = T.insert(Point(1,0,0));
  Vertex_handle v2 = T.insert(Point(0,1,0));
  Vertex_handle v3 = T.insert(Point(0,0,1));
  Vertex_handle v4 = T.insert(Point(2,2,2));
  Vertex_handle v5 = T.insert(Point(-1,0,1));

   // Now we can link the vertices as we like.
  v0->vh = v1;
  v1->vh = v2;
  v2->vh = v3;
  v3->vh = v4;
  v4->vh = v5;
  v5->vh = v0;

  return 0;
}
```