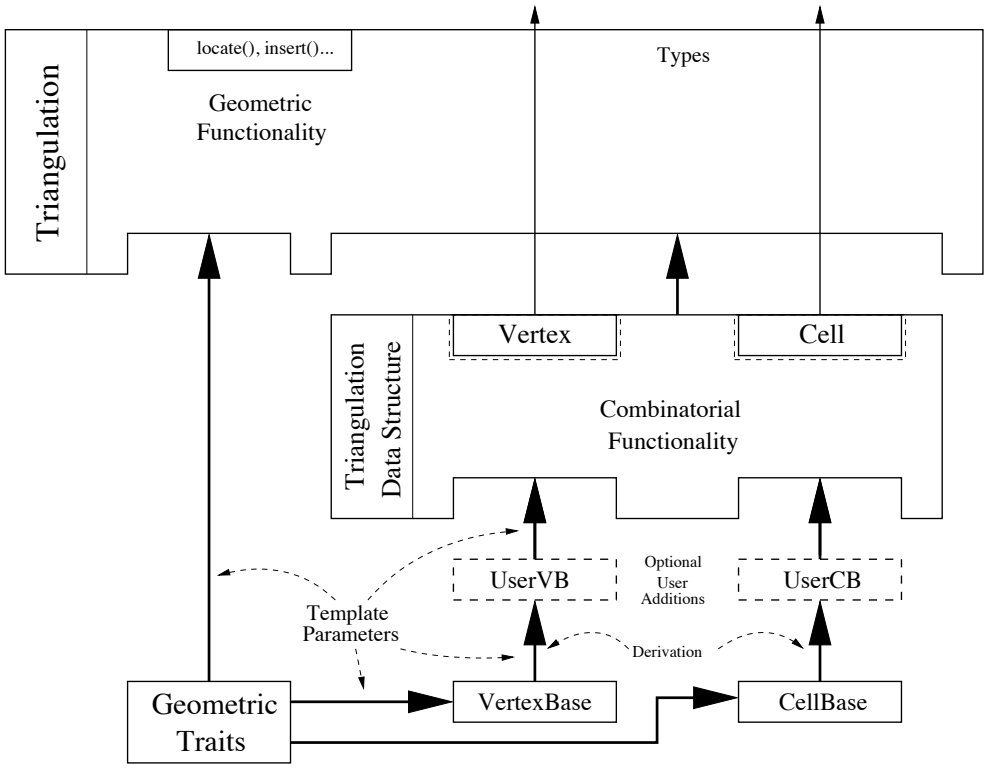

Figure 1.5: Triangulation software design.

```
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) {}
...
};
... // The rest has not changed
```

The situation is exactly similar for cell base classes. Section **??** provides more detailed information.

1.5 Complexity and Performance

In 3D, the worst case complexity of a triangulation is quadratic in the number of points. For Delaunay triangulations, this bound is reached in cases such as points equally distributed on two non-coplanar lines. However,

the good news is that, in many cases, the complexity of a Delaunay triangulation is linear or close to linear in the number of points. Several articles have proven such good complexity bounds for specific point distributions, such as points distributed on surfaces under some conditions.

1.5.1 Running Time

There are several algorithms provided in this package. We will focus here on the following ones and give practical numbers on their efficiency :

- construction of a triangulation from a range of points,
- location of a point (using the *locate* function),
- removal of a vertex (using the *remove* function).

We will use the following types of triangulations, using *Exact_predicates_inexact_constructions_kernel* as geometric traits (combined with *Regular_triangulation_euclidean_traits_3* in the weighted case) :

- **Delaunay** : *Delaunay_triangulation_3*
- **Delaunay - Fast location** : *Delaunay_triangulation_3* with *Fast_location*
- **Regular** : *Regular_triangulation_3* (default setting : memorize hidden points)
- **Regular - No hidden points** : *Regular_triangulation_3* with hidden points discarded (using *Triangulation_cell_base_3* instead of *Regular_triangulation_cell_base_3*).

Figure 1.6 shows, for all these types of triangulations, the times in seconds taken to build a triangulation from a given number of points, then the average time to perform one point location in triangulations of various sizes, and the average time to perform one vertex removal (which is largely independant on the size of the triangulation).

The data sets used here are points randomly distributed in the unit cube (the coordinates are generated using the `drand48` C function). In the weighted case, the weights are all zero, which means that there are actually no hidden points during execution.

The measurements have been performed using CGAL 3.6, using the GNU C++ compiler version 4.3.2, under Linux (Fedora 10 distribution), with the compilation options `-O3 -DCGAL_NDEBUG`. The computer used was equiped with a 64bit Intel Xeon 3GHz processor and 32MB of RAM (a recent desktop machine as of 2009).

1.5.2 Memory Usage

We give here some indication about the memory usage of the triangulations. Those structures being intensively based on pointers, the size almost doubles on 64bit platforms compared to 32bit.

The size also depends on the size of the point type which is copied in the vertices (hence on the kernel). Obviously, any user data added to vertices and cells also affect the memory used.

More specifically, the memory space used to store a triangulation is first a function of the size of its *Vertex* and *Cell* types times their numbers (and for volumic distribution, one sees about 6.7 times more cells than vertices). However, these are stored in memory using *Compact_container*, which allocates them in lists of

	Delaunay	Delaunay Fast location	Regular	Regular No hidden points
Construction from 10^2 points	0.00054	0.000576	0.000948	0.000955
Construction from 10^3 points	0.00724	0.00748	0.0114	0.0111
Construction from 10^4 points	0.0785	0.0838	0.122	0.117
Construction from 10^5 points	0.827	0.878	1.25	1.19
Construction from 10^6 points	8.5	9.07	12.6	12.2
Construction from 10^7 points	87.4	92.5	129	125
Point location in 10^2 points	9.93e-07	1.06e-06	7.19e-06	6.99e-06
Point location in 10^3 points	2.25e-06	1.93e-06	1.73e-05	1.76e-05
Point location in 10^4 points	4.79e-06	3.09e-06	3.96e-05	3.76e-05
Point location in 10^5 points	2.98e-05	6.12e-06	1.06e-04	1.06e-04
Point location in 10^6 points	1e-04	9.65e-06	2.7e-04	2.67e-04
Point location in 10^7 points	2.59e-04	1.33e-05	6.25e-04	6.25e-04
Vertex removal	1e-04	1.03e-04	1.42e-04	1.38e-04

Figure 1.6: Running times for algorithms on 3D triangulations.

	Delaunay	Delaunay Fast location	Regular	Regular No hidden points
32bit	274	291	336	282
64bit	519	553	635	527

Figure 1.7: Memory usage in bytes per point for large data sets.

blocks of growing size, and this requires some additional overhead for bookkeeping. Moreover, memory is only released to the system when clearing or destroying the triangulation. This can be important for algorithms like simplifications of data sets which will produce fragmented memory usage (doing fresh copies of the data structures are one way out in such cases). The asymptotic memory overhead of *Compact_container* for its internal bookkeeping is otherwise on the order of $O(\sqrt{n})$.

Figure 1.7 shows the number of bytes used per points, as measured empirically using *Memory_sizer* for large triangulations (10^6 random points).

1.5.3 Variability Depending on the Data Sets and the Kernel

Besides the complexity of the Delaunay triangulation that varies with the distribution of the points, another critical aspect affects the efficiency : the degeneracy of the data sets. These algorithms are quite sensitive to numerical accuracy and it is important to run them using exact predicates.

Using a kernel with no exact predicates will quickly lead to crashes or infinite loops once they are executed on non-random data sets. More precisely, problems appear with data sets which contain (nearly) degenerate cases for the *orientation* and *side_of_oriented_sphere* predicates, namely when there are (nearly) coplanar or (nearly) cospherical points. This unfortunately happens often in practice with data coming from various kinds of scanners or other automatic acquisition devices.

Using an inexact kernel such as *Simple_cartesian<double>* would lead to optimal performance, which is only about 30% better than *Exact_predicates_inexact_constructions_kernel*. The latter is strongly recommended since it takes care about potential robustness issues. The former can be used for benchmarking purposes mostly, or when you really know that your data sets won't exhibit any robustness issue.

Exact predicates take more time to compute when they hit (nearly) degenerate cases. Depending on the data set, this can have a visible impact on the overall performance of the algorithm or not.

Sometimes you need exact constructions as well, so *Exact_predicates_exact_constructions_kernel* is a must. This will slow down the computations by a factor of 4 to 5 at least (it can be much more).

1.6 Examples

1.6.1 Basic Example

This example shows the incremental construction of a 3D triangulation, the location of a point and how to perform elementary operations on indices in a cell. It uses the default parameter of the *Triangulation_3* class.

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

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

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()
{
  // construction from a list of points :
  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();

  // insertion from a vector :
  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());
```