Current design

Overview

```
template < class Gt, Class Tds >
Triangulation_3
```

```
template
< class GT, Class Tds >
class
Triangulation_vertex_3
  : public Tds::Vertex
```

```
template
< class Gt, class Tds >
class
Triangulation_cell_3
  : public Tds::Cell
```

```
template < class Vb, Class Cb >
class Triangulation_data_structure_3
```

```
template <class Vb, class Cb >
class
Triangulation_ds_vertex_3
  : public Vb
```

```
template < class Vb, class Cb >
class
Triangulation_ds_cell_3
  : public Cb
```

```
template < class Gt >
class
Triangulation_vertex_base_3
```

```
template < class Gt >
class
Triangulation_cell_base_3
```

```
template < class R >
class Triangulation_geom_traits_3
```

Current design


```
template < class GT, Class Tds >
class CGAL_Triangulation_3
{
public:
  typedef typename GT::Point Point;
  typedef typename GT::Segment Segment;
  typedef typename GT::Triangle Triangle;
  typedef typename GT::Tetrahedron Tetrahedron;

  typedef CGAL_Triangulation_cell_handle_3<GT,Tds> Cell_handle;
  typedef CGAL_Triangulation_vertex_handle_3<GT,Tds> Vertex_handle;

  typedef CGAL_Triangulation_cell_3<GT,Tds> Cell;
  typedef CGAL_Triangulation_vertex_3<GT,Tds> Vertex;
  typedef pair<Cell_handle, int> Facet;
  typedef CGAL_triple<Cell_handle, int, int> Edge;

private:
  Tds _tds;
  GT  _gt;

/* ... */
};
```

Current design


```
template<class Gt, class Tds >
class CGAL_Triangulation_vertex_3
  : public Tds::Vertex
{
public:
  typedef typename Gt::Point Point;

  typedef CGAL_Triangulation_cell_3<Gt,Tds> Cell;
  typedef CGAL_Triangulation_vertex_3<Gt,Tds> Vertex;

  typedef CGAL_Triangulation_vertex_handle_3<Gt,Tds> Vertex_handle;
  typedef CGAL_Triangulation_cell_handle_3<Gt,Tds> Cell_handle;

/* ... */
};
```

Current design


```
template < class Gt, class Tds >
class CGAL_Triangulation_cell_3
  : public Tds::Cell
{
public:
  typedef typename Gt::Point Point;

  typedef CGAL_Triangulation_vertex_3<Gt,Tds> Vertex;
  typedef CGAL_Triangulation_cell_3<Gt,Tds> Cell;

  typedef CGAL_Triangulation_vertex_handle_3<Gt,Tds> Vertex_handle;
  typedef CGAL_Triangulation_cell_handle_3<Gt,Tds> Cell_handle;

/* ... */
};
```

Current design

Data structure

```
template < class Vb, class Cb >
class CGAL_Triangulation_data_structure_3
{
public:
  typedef CGAL_Triangulation_ds_vertex_3<Vb,Cb> Vertex;
  typedef CGAL_Triangulation_ds_cell_3<Vb,Cb> Cell;
  typedef pair<Cell*, int>  Facet;
  typedef CGAL_triple<Cell*, int, int> Edge;

/* ... */
};
```

Current design

Data structure

```
template <class Vb, class Cb >
class CGAL_Triangulation_ds_vertex_3
  : public Vb
{
public:
  typedef typename Vb::Point Point;
  typedef CGAL_Triangulation_ds_vertex_3<Vb,Cb> Vertex;
  typedef CGAL_Triangulation_ds_cell_3<Vb,Cb> Cell;
/* ... */
};

template < class Vb, class Cb >
class CGAL_Triangulation_ds_cell_3
  : public Cb
{
public:
  typedef CGAL_Triangulation_ds_vertex_3<Vb,Cb> Vertex;
  typedef CGAL_Triangulation_ds_cell_3<Vb,Cb> Cell;
/* ... */
};
```

Current design

Traits class

```
template < class R >
class Triangulation_geom_traits_3
{
public:
  typedef CGAL_Point_3<R>  Point;
  typedef CGAL_Segment_3<R> Segment;
  typedef CGAL_Triangle_3<R> Triangle;
  typedef CGAL_Tetrahedron_3<R> Tetrahedron;

/* ... */
};
```

Current design

Base

```
template < class GT >
class CGAL_Triangulation_vertex_base_3
{
public:
  typedef typename GT::Point Point;

private:
  Point _p;
  void * _c;

/* ... */
};
```

```
template < class Gt >
class CGAL_Triangulation_cell_base_3
{
public:

private:
  void* V[4];
  void* N[4];

/* .. */
};
```

Current design

User program

```
#include < ... >

typedef double coord_type;
typedef CGAL_Cartesian<coord_type>  Rep;
typedef CGAL_Triangulation_geom_traits_3<Rep> Gt;
typedef Gt::Point Point;

typedef CGAL_Triangulation_vertex_base_3<Gt> Vb;
typedef CGAL_Triangulation_cell_base_3<Gt>  Cb;

typedef CGAL_Triangulation_data_structure_3<Vb,Cb> TDS;

typedef CGAL_Triangulation_3<Gt,TDS> Triangulation;

int main()
{
  Triangulation_3 T;

  Point p;

  while ( cin > > p ) {
    T.insert(p);
  }

  return 1;
}
```

Application

Colored cells

```
#include <CGAL/basic.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Triangulation_euclidean_traits_3.h>
#include <CGAL/Triangulation_data_structure_3.h>
#include <CGAL/Triangulation_3.h>

/* A facet with a color member variable. */
```

```
template < class Gt >
My_cell_base<Gt>
 : public CGAL_Triangulation_cell_base_3<Gt>
{
public:
   CGAL_Color color;
};
```

Application

Colored cells

```
typedef CGAL_Cartesian<CGAL_Rational> Rep;
typedef CGAL_Triangulation_euclidean_traits_3<Rep> Gt;
typedef CGAL_Triangulation_vertex_base_3<Gt> Vb;
typedef CGAL_My_cell_base_3<Gt> Cb;
typedef CGAL_Triangulation_default_data_structure_3<Gt,Vb,Cb> Tds;
typedef CGAL_Triangulation_3<Gt,Tds> Triangulation;
typedef Triangulation::Cell_iterator Cell_iterator;
typedef Triangulation::Cell_handle Cell_handle;

int main()
{
    Triangulation t;
    Point p;
     while (cin >> p){
        t.insert(p);
    }
    Cell_iterator cit = t.cells_begin();
    while (cit != t.cells_end())
        {cit->color = CGAL_BLUE; }
    cin >> p;
    Cell_handle ch = t.locate(p);
    ch->color = CGAL_RED;
}
```

Degenerate dimensions

```
void insert_outside_affine_hull(Vertex* v,
                                Vertex* star = NULL,
                                bool reorient = false)
/* star = vertex from which we triangulate the facet */
{ /* ... */
while( e != d ){
  enew = new Cell( *this );
  enew->set_vertex(i,e->vertex(j));
  enew->set_vertex(j,e->vertex(i));
  enew->set_vertex(2,star);

  enew->set_neighbor(i,cnew);
  cnew->set_neighbor(j,enew);

  enew->set_neighbor(2,e);
  /* neighbor j will be set during next iteration of the loop */

  e->set_vertex(2,v);
  e->set_neighbor(2,enew);

  c = e;
  e = e->neighbor(i);
  cnew = enew;
}
  /* ... */
}
```

Default geometric traits

Objects

```
template < class R >
class CGAL_Triangulation_geom_traits_3
{
public:

  typedef CGAL_Point_3<R>  Point;
  typedef CGAL_Point_2< CGAL_Cartesian<
                        CGAL_Quotient<typename R::RT> > >  Point2;
  typedef CGAL_Segment_3<R> Segment;
  typedef CGAL_Triangle_3<R> Triangle;
  typedef CGAL_Tetrahedron_3<R> Tetrahedron;

  CGAL_Triangulation_geom_traits_3()
    {}

  CGAL_Triangulation_geom_traits_3
         (const CGAL_Triangulation_geom_traits_3 & gt)
     {}

  CGAL_Triangulation_geom_traits_3 &
  operator=(const CGAL_Triangulation_geom_traits_3 & gt)
    {return *this;}
```

Default geometric traits

Comparisons

```
bool equal(const Point & p, const Point & q) const
{
  return (p == q);
}


CGAL_Comparison_result compare_x
       (const Point & p, const Point & q) const
  {
    return CGAL_compare_x(p, q);
  }


CGAL_Comparison_result compare_y
       (const Point & p, const Point & q) const
  {
    return CGAL_compare_y(p, q);
  }


CGAL_Comparison_result compare_z
       (const Point & p, const Point & q) const
  {
    return CGAL_compare_z(p, q);
  }
```

Default geometric traits — 3D Predicates

```
/* ORIENTATION PREDICATES */
CGAL_Orientation orientation(const Point & p,
                             const Point & q,
                             const Point & r,
                             const Point & s) const
{ return CGAL_orientation(p, q, r, s); }
bool collinear(const Point & p,
               const Point & q,
               const Point & r) const
{ return CGAL_collinear(p,q,r); }

/* for DELAUNAY */


CGAL_Oriented_side
side_of_oriented_sphere(const Point & p,
                        const Point & q,
                        const Point & r,
                        const Point & s,
                        const Point & test) const
{ return CGAL_side_of_oriented_sphere(p, q, r, s, test); }
```

Default geometric traits — 2D in 3D

```
CGAL_Orientation orientation_in_plane(const Point & q,
     const Point & r, const Point & s, const Point & p) const
     /* tests whether p is on the same side of q,r as s */
{
  CGAL_triangulation_precondition( ! collinear(q,r,s) );
  CGAL_triangulation_precondition( orientation(p,q,r,s)
                                   == CGAL_COPLANAR );

  /* projection on the x,y-plane */
  Point2 P(p.hx(), p.hy(), p.hw()); /* Q, R, S */
  CGAL_Orientation oxy_qrs = CGAL_orientation(Q,R,S);

  if ( oxy_qrs != CGAL_COLLINEAR ) /* projection on x,y is ok */
    return CGAL_Orientation( oxy_qrs * CGAL_orientation(Q,R,P) );

  if ( ( Q.x() != R.x() ) || ( Q.x() != S.x() ) ) {
    /* projection on x,z-plane is ok */
    P = Point2(p.hx(), p.hz(), p.hw()); /* Q, R, S */
  }
  else { /* projection on y,z-plane */
    P = Point2(p.hy(), p.hz(), p.hw()); /* Q, R, S */
  }
  return CGAL_Orientation ( CGAL_orientation(Q,R,S)
                          * CGAL_orientation(Q,R,P) );
}
```

Default geometric traits 2D in 3D

```
CGAL_Oriented_side
side_of_oriented_circle(const Point & p,
                        const Point & q,
                        const Point & r,
                        const Point & test) const
 {
   CGAL_triangulation_precondition( orientation(p,q,r,test) ==
    CGAL_COPLANAR );

   /* test belongs to the circle if and only if it belongs to a */
   /* sphere passing through pqr */
   CGAL_Orientation or;
   Point O(0,0,0), A(1,0,0), B(0,1,0), C(0,0,1);

   Point P = ((or = orientation(p,q,r,O)) != CGAL_ZERO) ? O:
             ((or = orientation(p,q,r,A)) != CGAL_ZERO) ? A:
             ((or = orientation(p,q,r,B)) != CGAL_ZERO) ? B:
             ((or = orientation(p,q,r,C)) != CGAL_ZERO) ? C: C;

   return CGAL_Oriented_side( or *
   CGAL_side_of_oriented_sphere(p, q, r, P, test));
  }

}; /* end CGAL_Triangulation_geom_traits_3 */
```

A traits class for terrains

```
#include <CGAL/Point_3.h>
#include <CGAL/predicates_on_ftC2.h>
template < class R >
class CGAL_Triangulation_terrain_traits
{
public:
    typedef CGAL_Point_3<R>  Point;
    static typename R::FT x(const Point &p) { return p.x(); }
    static typename R::FT y(const Point &p) { return p.y(); }

    bool equal(const Point &p, const Point &q) const
    {
      return ( x(p)==x(q)) &&  (y(p)==y(q) );
    }

    CGAL_Oriented_side side_of_oriented_circle
          (const Point &p, const Point &q,
           const Point &r, const Point &s) const
    {
      if ( equal(p,q) || equal(q,s) || equal(r,s) )
              return CGAL_ON_ORIENTED_BOUNDARY;

      return CGAL_side_of_oriented_circleC2
          ( x(p), y(p),  x(q), y(q),  x(r), y(r),  x(s), y(s) );
    }
    /* ... */
};
```

Combinatorial Validity

CGAL_Triangulation_ds_vertex_3

```
bool is_valid() const
{
  if ( ! Vb::is_valid() ) return false;
  return ( cell()->has_vertex(this) );
}
```

CGAL_Triangulation_ds_cell_3

```
bool is_valid(int dim = 3) const /* dimension 3 */
{
  if ( ! Cb::is_valid() ) return false;

  for(int i = 0; i < 4; i++) {
    if ( vertex(i) == NULL ) return false;
    vertex(i)->is_valid(); }

  for(i = 0; i < 4; i++) {
    Cell* n = neighbor(i);
    if ( n == NULL ) return false;
    int in; if ( ! n->has_neighbor(this,in) ) return false;

    int j1n,j2n,j3n;
    if ( ! n->has_vertex(vertex((i+1)&3),j1n) ) return false;
    if ( ! n->has_vertex(vertex((i+2)&3),j2n) ) return false;
    if ( ! n->has_vertex(vertex((i+3)&3),j3n) ) return false;
    if ( in+j1n+j2n+j3n != 6) return false;
```

Combinatorial Validity II

```
    /* tests whether the orientations of this and n are consistent */
    if ( ((i+in)&1) == 0 ) { /* i and in have the same parity */
      if ( j1n == ((in+1)&3) ) {
        if ( ( j2n != ((in+3)&3) ) || ( j3n != ((in+2)&3) ) )
          return false; }
      if ( j1n == ((in+2)&3) ) {
        if ( ( j2n != ((in+1)&3) ) || ( j3n != ((in+3)&3) ) )
          return false; }
      if ( j1n == ((in+3)&3) ) {
        if ( ( j2n != ((in+2)&3) ) || ( j3n != ((in+1)&3) ) )
          return false; }
    }
    else { /* i and in do not have the same parity */
      if ( j1n == ((in+1)&3) ) {
        if ( ( j2n != ((in+2)&3) ) || ( j3n != ((in+3)&3) ) )
          return false; }
      if ( j1n == ((in+2)&3) ) {
        if ( ( j2n != ((in+3)&3) ) || ( j3n != ((in+1)&3) ) )
          return false; }
      if ( j1n == ((in+3)&3) ) {
        if ( ( j2n != ((in+1)&3) ) || ( j3n != ((in+2)&3) ) )
          return false; }
    }
  } /* end looking at neighbors */
  return true;
} /* end is_valid */
```

Delaunay Hierarchy

Vertex

```
template < class Traits >
class CGAL_Delaunay_hierarchy_vertex_base_2
 : public CGAL_Triangulation_vertex_base_2<Traits>
{
/* ... */

private:
  void* _up;    /* same vertex one level above */
  void* _down;  /* same vertex one level below */
};
```

Delaunay Hierarchy

```
/*: parameterization of the Delaunay hierarchy */
const float CGAL_Delaunay_hierarchy_2__ratio    = 30.0;
const int   CGAL_Delaunay_hierarchy_2__minsize  = 20;
const int   CGAL_Delaunay_hierarchy_2__maxlevel = 5;
/ maximal number of points is 30^5 = 24 millions ! */

template < class Traits, class Tds>
class CGAL_Delaunay_hierarchy_2
: public CGAL_Delaunay_triangulation_2<Traits,Tds>
{
private:
  Delaunay* hierarchy[CGAL_hierarchy_2__maxlevel];
  CGAL_Random random; /* random generator */

public:
  typedef CGAL_Delaunay_triangulation_2<Traits,Tds> Delaunay;

  CGAL_Delaunay_hierarchy_2(const Traits & traits)
    : Delaunay(traits),  random((long)0)
    {
      hierarchy[0] = this;
      for(int i=1;i<CGAL_hierarchy_2__maxlevel;++i)
            hierarchy[i] = new Delaunay();
    }

  /* ... */
```

```
void locate(const Point& p,
            Locate_type& lt, int& li,
  {         Face_handle pos[CGAL_hierarchy_2__maxlevel]) const
  Face_handle position;
  Vertex_handle nearest;
  int level  = CGAL_hierarchy_2__maxlevel;

  /* find the highest level with enough vertices */
  while (hierarchy[--level]->number_of_vertices()
          < CGAL_Delaunay_hierarchy_2__minsize){
    if ( ! level) break;  /* do not go below 0 */ }

  for (int i=level+1; i<CGAL_hierarchy_2__maxlevel;++i) pos[i]=0;
  while(level > 0) {
    pos[level]=position=hierarchy[level]->locate(p,position);
    /* usual location at that level */

    if (hierarchy[level]->is_infinite(position)) /* ... */
    else { nearest = closest( p, position->vertex(0),
                         position->vertex(1), position->vertex(2)) }

    nearest = (Vertex*)( nearest->down() ); /* same v, level below */
    position = nearest->face();             /* incident face */
    --level;
  }
  pos[0]=hierarchy[level]->locate(p,lt,li,position);  /* at level 0 */
  }
```

Delaunay Hierarchy

Insertion

```
Vertex_handle
insert(const Point  &p,
       Locate_type& lt)
{
  int vertex_level = random_level();
  int i;

  /* locate using hierarchy */
  Face_handle positions[CGAL_hierarchy_2__maxlevel];
  locate(p,lt,i,positions);

  /* insert at level 0 */
  Vertex_handle vertex=hierarchy[0]->insert(p,lt,positions[0]);
  Vertex_handle previous=vertex;
  Vertex_handle first = vertex;

  int level  = 1;
  while (level <= vertex_level ){
    vertex=hierarchy[level]->insert(p,positions[level]);
    vertex->set_down((void *) &*previous); /* link with level above */
    previous->set_up((void *) &*vertex);
    previous=vertex;
    level++;
 }
  return first;
}
```