How to add facet attributes to CGAL's 3D geometric triangulations *

Joachim Giesen Matthias John
ETH Zürich, CH-8092 Zürich, Switzerland
{giesen,john}@inf.ethz.ch

Abstract

The 3D triangulation package of CGAL is well suited for many applications, in particular for surface reconstruction. Our experience in using CGAL in the implementation of several surface reconstruction algorithms revealed that it is inconvenient that one cannot add attributes to facets directly. This paper presents three approaches how to deal with this problem.

Keywords. CGAL, Delaunay triangulation, surface reconstruction, facet attributes

1 Introduction

CGAL is a C++ library for geometric algorithms. The goal of CGAL is to provide robust, efficient, flexible, and easy to use implementations of geometric algorithms and data structures. One of its powerful features is the triangulation package that includes Delaunay– and regular triangulations. We were involved in the development and implementation of three different surface reconstruction algorithms and their extensions [1, 5, 7, 9]. Additionally,we implemented prototypes of several other surface reconstruction algorithms [2, 3, 10]. The task in the surface reconstruction problem is to compute a manifold triangular mesh from a set of unorganized points that are sampled from the surface of some solid in $\mathbb{R}^3$. Figure 1 shows an example. More examples and the runtimes can be found in the papers.
It turned out that CGAL is well suited for the implementation of surface reconstruction algorithms that are based on the 3D Delaunay triangulation. This triangulation is part of CGAL's triangulation package. Our implementations have benefited mainly from the following features of the triangulations provided by CGAL,

- the choice of several geometric representations for fast and/or robust computations, see also [6].
- versatile access functions to the faces of the triangulation via iterators and circulators.
- flexible mechanisms to add attributes to vertices and cells

*Partly supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces).

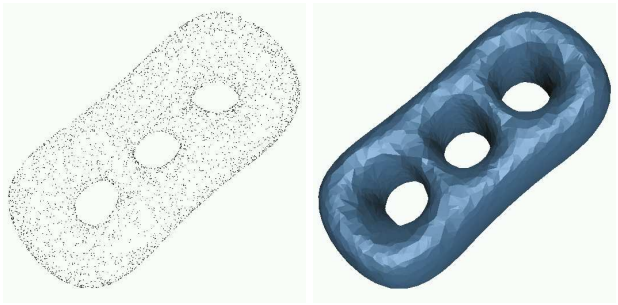

Figure 1: Left: A data set of sample points. Right: The reconstructed surface.

- the derivation mechanism of C++ to adapt the Delaunay triangulation to the requirements of the various algorithms.

The feature we missed at most is the lack of a mechanism to add facet attributes to the triangulation. In implementations of surface reconstruction algorithms many facet attributes are needed. Such an attribute could be the facet circumcenter or a boolean variable that indicates if a facet has to be chosen for the reconstruction. Based on our implementation experience we developed several solutions to this problem which we present in this paper.

This paper is organized as follows. The second chapter gives a short introduction to the design of CGAL's triangulation package. In the third chapter we present three proposals how to add facet attributes.

2 The design of the 3D triangulations

The design of CGAL is described in [8]. In particular the design of the triangulations is described in [4]. Here we give a short overview following these publications.
Each 3D geometric triangulation is parametrized by two template parameters.

```
template < class Traits,
           class Triangulation_data_structure >
Triangulation;
```

The first parameter is a geometric traits class that provides geometric objects and geometric predicates on those objects.

The traits classes are of no further interest in the context of this paper. The second parameter is a geometric data structure that provides the combinatorial structure of the triangulation, i.e. the necessary information about incidences between the faces of the triangulation.
To describe the combinatorial structure it is sufficient to describe the incidences between the vertices and cells. The current implementation of CGAL makes widely use of this fact, i.e. only vertices and cells are represented explicitly. The currently implemented triangulation data structure is parametrized by two base classes for vertices and cells.

```
template < class Vertex_base, class Cell_base >
Triangulation_data_structure;
```

They contain geometric information and can be used to store additional attributes of vertices and cells (e.g. color information).
In the triangulation vertices and cells are usually represented as handles. A handle is a class that behaves like a pointer. The following code fragment shows how to access a vertex that is incident to a cell.

```
Triangulation::Cell_handle ch;
...
Triangulation::Vertex_handle vh = ch->vertex(0);
```

In the current CGAL implementation vertex handles and cell handles are defined in the triangulations, not in the triangulation data structure.
Facets and edges are represented implicitly. A facet is given by a cell and an index. An edge is given by a cell and two indices.

```
typedef pair < Cell_handle, int >         Facet;
typedef triple < Cell_handle, int, int >  Edge;
```

Therefore it is not possible to store facet/edge attributes directly in a facet/edge and there are no handles for facets and edges. One should be aware of the fact that both implicit representations are not unique, not even in CGAL's iterators and circulators.

3 Adding facet attributes

Because facets are represented implicitly one has to find another way to store facet attributes. In this chapter we present three solutions.
One of our design goals was to change the source code of the current CGAL implementation as little as possible. Instead we used the derivation and template mechanisms of C++ to reuse almost the whole code.

3.1 Storing facet attributes in the cell base

The implicit declaration of facets suggests a direct way to store facet attributes. Four attribute instances are stored in each cell, one for every facet that is incident to that cell. Thereby an attribute for one particular facet is stored in both of its adjacent cells, i.e. it is stored twice. This can be implemented by adding attributes and access functions to the cell base.

```
template < class Traits >
class Cell_base_with_attribute
  : public Cell_base < Traits > {
 public:
  int facet_color[4];
}
```

This approach gets by with a small overhead to the current CGAL implementation. On the other hand storing an attribute twice yields two disadvantages. First one needs twice the memory. Secondly, due to the not necessarily unique implicit representation of facets the access to an attribute is not consistent. Therefore every update of a facet attribute has to be made in both adjacent cells. One way to guarantee this is to implement attribute update functions parametrized by a cell and an index. These functions automatically update the corresponding attribute in the neighboring cell. They cannot be member functions of the cell base without rewriting existing CGAL classes. Therefore they have to be member functions of a derived triangulation class.

```
template < class Triangulation >
class Triangulation_with_facet_colors
  : public Triangulation {
  ...
 public:
  void set_facet_color(Cell_handle ch, int ind_ch,
                       int color) {
    Cell_handle ne = ch->neighbor(ind_ch);
    int ind_ne = ne->index(ch);
    ch->facet_color[ind_ch] = color;
    ne->facet_color[ind_ne] = color;
  }
}
```

Unfortunately it is too expensive to implement these functions for more than a few attributes. In addition it makes the source code difficult to read since the update functions are part of the triangulation and not part of the cell.

3.2 Collecting facet attributes in a separate class

A more sophisticated approach is to collect all facet attributes in a class.

```
class Facet_attributes {
 public:
  int color;
}
```

Each cell aggregates four pointers to instances of such a class. Instead of pointers one can use handles which results in a class `Facet_attributes_handle`. The new cell base can be parametrized by the facet attributes–class via a template argument.

```
template < class Traits, class Facet_attributes >
class Cell_base_with_facets
  : Cell_base < Traits > {
  ...
 private:
  Facet_attributes_handle fa[4];
 public:
  Facet_attributes_handle facet_attributes(int i) {
    return fa[i];
  }
  void set_facet_attributes(int i,
         Facet_attributes_handle fah) {
    fa[i] = fah;
  }
}
```

Next we discuss how one can ensure that two neighboring cells point to the same attribute instance for the same facet. This solves the problems of the first solution. That is, no

attribute of a particular facet is stored twice and all accesses are consistent. One way to achieve this is to allow point insertions to the triangulation only at once. This can be guaranteed by providing only one constructor for the triangulation. This constructor takes a set of points as an argument and builds the facet- and attribute incidences after inserting all points. No later point insertion is allowed.

```
template < class Triangulation >
class Triangulation_with_facet_attributes
  : public Triangulation {
  ...
 public:
  template < class InputIterator >
  Triangulation_with_facet_attributes
    (Input_iterator first, Input_iterator last) {
    ...
  }
  ...
}
```

To simplify the access to the facet attributes it is helpful to provide access functions in a derived triangulation. This functions should be available for every facet representation that is provided by CGAL: a pair of `Cell_handle` and `int`, `Facet`, `Facet_iterator`, and `Facet_circulator`.

```
template < class Triangulation >
class Triangulation_with_facet_attributes
  : public Triangulation {
  ...
 public:
  Facet_attributes_handle
    attributes(Cell_handle ch, int ind) {
      return ch->facet_attributes(ind);
    }
  Facet_attributes_handle
    attributes(Facet_iterator fit) {
      return attributes((*fit).first, (*fit).second);
    }
  // same for arguments Facet_circulator and Facet ...
}
```

With the above class definitions an implementation that uses facet attributes looks like the following. It shows how the classes are combined to define a Delaunay triangulation and how to access the facet attribute `color`.

```
typedef Traits                                  Gt;
typedef Vertex_base
        < Gt >                                  Vb;
typedef Cell_base_with_facet_attributes
        < Gt, Facet_attributes >                Cb;
typedef Triangulation_data_structure
        < Vb, Cb >                              Tds;
typedef Delaunay_triangulation
        < Gt, Tds >                             Dt;
typedef Triangulation_with_facet_attributes
        < Dt >                                  My_tr;
...
std::list < Point > l;
...
My_tr mt(l.begin(), l.end());
My_tr::Facet_iterator fit = ...
while (...) {
  mt.attributes(fit)->color = i;
  ...
}
```

This second approach has a small overhead to the current CGAL implementation. But all new classes are defined in a way such that they are very flexible. That is, they can be used with every predefined CGAL–3D–triangulation and an arbitrary set of facet attributes.

3.3 Representing facets explicitly

An improvement to the previous approach would be a direct access to facet attributes via facets. Instead of a function call

```
attributes(facet_iterator)->color
```

one would like to have

```
facet_iterator->color.
```

With an explicit representation of facets this can be achieved. Therefore we provide a facet class that is parametrized by a facet base class that includes all facet attributes.
To link the facets with the triangulation data structure we add an attribute to the facet class for an incident cell handle. Additionally we equip every cell with four instances of facet handles.

```
template < class Facet_base >
class Facet_with_cell_base : public Facet_base {
  Cell_handle ch;
  ...
}
template < class Cell_base >
class Cell_with_facets_base : public Cell_base {
  Facet_handle[4] fh;
  ...
}
```

Unfortunately this yields cyclic dependencies that are shown in Figure 2. They can be resolved using a technique described in [11] which results in the following class signature for a triangulation data structure inlcuding facets.

```
template < class Traits,
           class Vertex_base,
           class Cell_base,
           class Facet_base>
Triangulation_with_facets_data_structure;
```

Furthermore we provide a new triangulation class that is derived from a CGAL triangulation class and overwrites all member functions that are related to facets. Additionally all types that are related to facets like facet iterators and facet circulators are redefined. Like in the previous solution points can only be inserted via a constructor.

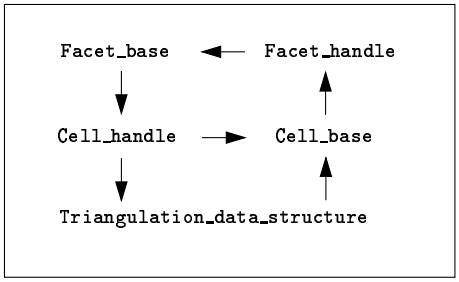

Figure 2: Cyclic dependencies from including an expicit facet representation in the exisitng CGAL data structure. An arrow indicates that a class depends on another class.

```
template < class Triangulation >
class Triangulation_with_facets
: public Triangulation {
  ...
}
```

Our implementation reverts to the functions and classes that are provided by the underlying triangulation.
An example that uses the explicit facet representation looks like the following.

```
typedef Traits              Gt;
typedef Vertex_base < Gt >  Vb;
typedef Cell_base < Gt >    Cb;
typedef Facet_attributes    Fb;

typedef Triangulation_with_facets_data_structure
        < Gt, Vb, Cb, Fb > :: Type              Tds;
typedef Delaunay_triangulation < Gt, Tds >      Dt;
typedef Triangulation_with_facets < Dt >        My_tr;
...
std::list < Point > l;
...
My_tr mt(l.begin(), l.end());
My_tr::Facet_iterator fit = ...
while (...) {
  fit->color = i;
  ...
}
```

We should remark that this solution has some restrictions. Usually the member function `index` returns the index of an incident vertex or cell. One would like to overload this function such that it works for facets.

```
int i = cell_handle->index(facet_handle);
```

If we add this function to our cell base it will be overwritten by a different function with the same name. So we decided to add an extra member function for facet indices with a different name.

```
int i = cell_handle->index_facet(facet_handle);
```

4 Conclusion

In this paper we motivated the necessity to add facet attributes to CGAL's triangulations. While implementing several surface reconstruction algorithms we developed some methods to add attributes to facets. They were refined for this paper.
The first (ad hoc) solution is well known and is widely used, e.g. in the alpha shape implementation of CGAL. It can be used for applications with only a few facet attributes.
In our implementations we mainly used the second approach. The implementation of the more sophisticated third solution is quite new. We hope to benefit in future projects from an explicit facet representation and the direct access to facet attributes via handles, iterators, and circulators. it would be really nice if future releases of CGAL would provide support for facet attributes directly.

Acknowledgment The authors would like to thank Michael Hoffmann for answering many questions concerning CGAL and C++ issues.

References

[1] U. Adamy, J. Giesen, M. John. New Techniques for Topologically Correct Surface Reconstruction. In *Proc. IEEE Visualization 2000*, pp. 373–380, (2000)

[2] N. Amenta, M. Bern, and M. Kamvysselis. A new Voronoi-based surface reconstruction algorithm. *Proc. SIGGRAPH 98*, pp. 415-421, (1998)

[3] N. Amenta, S. Choi, T.K. Dey, and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. In *Proc. 16th. ACM Sympos. Comput. Geom.*, pp. 213–222, (2000)

[4] J.–D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pp. 11–18, (2000)

[5] T.K. Dey, J. Giesen. Detecting Undersampling in Surface Reconstruction, In *Proc. 17th ACM Symp. Comp. Geom.*, pp. 257–263, (2001)

[6] T.K. Dey, J. Giesen, W. Zhao. Robustness Issues in Surface Reconstruction, In *Proc. Int. Conf. Comp. Sciences*, LNCS 2073, (2001)

[7] T.K. Dey, J. Giesen, J. Hudson. Delaunay Based Shape Reconstruction from Large Data, In *IEEE Symposium in Parallel and Large Data Visualization and Graphics*, pp. 19–27, (2001)

[8] A. Fabri, G.–J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Softw. - Pract. Exp.*, **30**(11), pp. 1167–1202, (2000)

[9] J. Giesen, M. John. Surface reconstruction based on a dynamical system. Manuscript, (2002)

[10] M. Gopi, S. Krishnan, C.T. Silva. Surface Reconstruction based on Lower Dimensional Localized Delaunay Triangulation. *Computer Graphics Forum, Proceedings of Eurographics*, **19**(3), pp. 467–478, (2000)

[11] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An Adaptable and Extensible Geometry Kernel In *Proc. 5th Workshop on Algorithm Engineering*, LNCS 2141, pp. 79–90, (2001)