Tutorial CGAL

Alexandra Bac

Polytech Informatique 4
Modélisation géométrique

Les TPs liés aux maillages et surfaces implicites seront implémentés via la bibliothèque CGAL et ce tutorial (ainsi que le matériel associé) fournit une rapide introduction.

CGAL est une bibliothèque dédiée à la géométrie algorithmique très connue dans le monde de l'informatique graphique.

1 Avant de commencer

CGAL fournit une implémentation très générique de la structure de données de demi-arêtes dont nous utiliserons la classe dérivée des maillages surfaciques. On y trouvera la structure de données elle-même, bien sûr, et tous les outils permettant de la manipuler, de la parcourir, modifier ... Les maillages ainsi codés sont des **maillages polyédriques** (donc pas seulement triangulaires).

Dans ce TP, un matériel "starter" vous est fourni, il vous mettra le pied à l'étrier ... La visualisation se fera en externe via Meshlab (modeleur géométrique).

2 Où est mon maillage ?

Le fichier `starter_defs.h` définit les types de base des maillages que nous utiliserons au cours de ces TP. Votre maillage sera donc un objet de classe `Mesh`.

La classe `Mesh` définit quatre classes imbriquées constituant les brique de base de la représentation en demi-arêtes des maillages :

(i) `Mesh::Vertex_index`

(ii) `Mesh::Halfedge_index`

(iii) `Mesh::Face_index`

(iv) `Mesh::Edge_index`

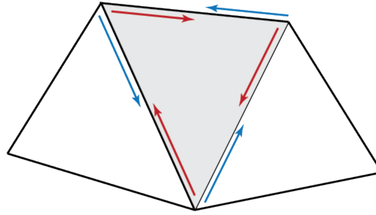
qui correspondent en fait à des indices entiers.

A noter que toutes ces classes sont génériques, et que nous manipulons en fait une instance de maillage surfaciques (appelée `Surface_mesh`), ce sera notre point de départ.

3 Manipuler des maillages triangulaires

3.1 Demi-arêtes - quelques rappels

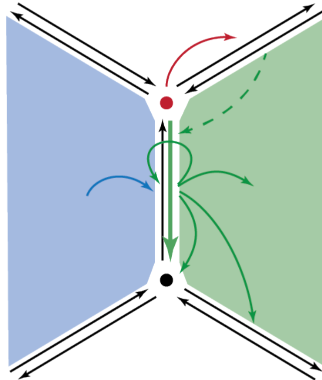
L'implémentation efficace des maillages repose sur la notion de demi-arête :



— Chaque arête du maillage est représentée par deux demi-arêtes.
— Chaque demi-arête est par conséquent attachée à un unique triplet (sommet, arête, face)

Ainsi, en parcourant les demi-arêtes voisines partageant sommet, arête ou face, on peut parcourir le maillage.

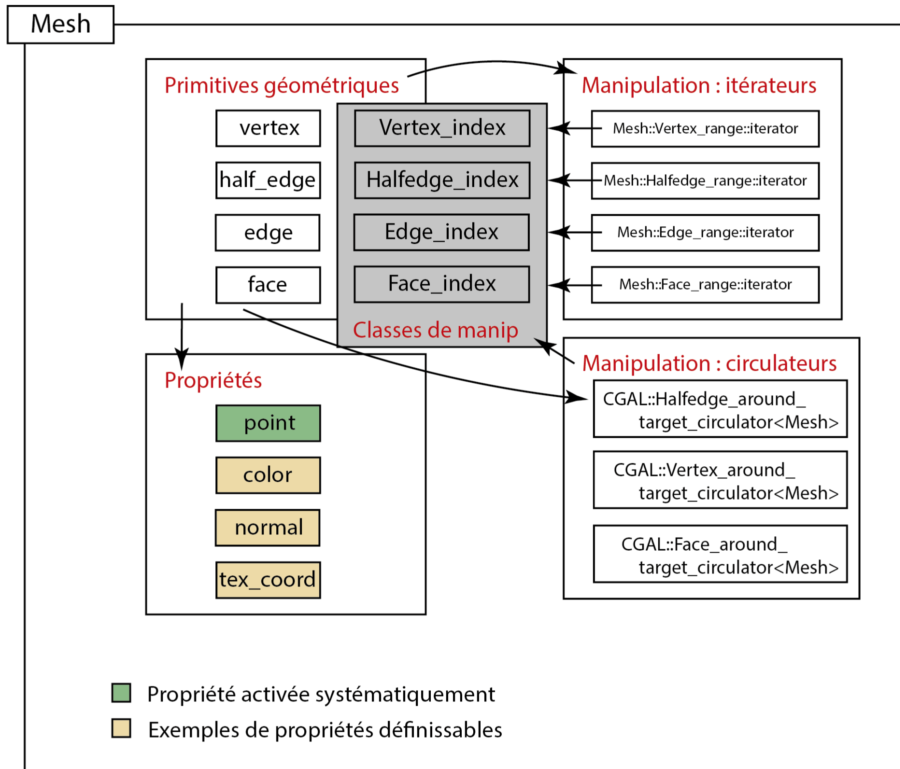
Plus précisement (ce qui ressort, c'est que le chaînage est vraiment réalisé à travers la structure de demi-arêtes) :



— Chaque **sommet** pointe sur une demi-arête sortante
— Chaque **face** pointe sur une des demi-arêtes la bordant
— Chaque **demi-arête** pointe sur :
  — Un sommet vers lequel elle pointe
  — La face à laquelle elle appartient
  — La demi-arête suivante de la face (sens horaire)
  — La demi-arête opposée
  — (éventuellement : la demi-arête précédente de la face)

3.2 Les ingrédients du maillage

Les schéma suivant reprend les ingrédients des maillages :



Comme on peut s'y attendre, le maillage repose sur des structures de points (`vertex`), arêtes (`edge`), demi-arêtes (`half_edge`) et faces (`face`).

Ces composants sont stockés comme des vecteurs et on y accède donc via leur *indices* (en fait des entiers).

(i) le parcours de l'ensemble des sommets (ou arêtes ...) se fait par des *itérateurs*

(ii) en déréférençant un itérateur on obtient l'indice de l'objet (`Mesh::Vertex_index`, `Mesh::Halfedge_index` ...) **c'est à travers ces indexes que se font toutes les manipulations.**

(iii) par ailleurs, des méthodes permettent de manipuler la structure de demi-arêtes pour atteindre ses voisins directes et les *circulateurs* fournissent un moyen de visiter les voisinages.

Parcourir avec les itérateurs sur les sommets, arêtes, faces

Un exemple de parcours de l'ensemble des sommets en utilisant un itérateur (il suffit de changer *Vertex* en *Face* ou *Halfedge* pour passer à l'itérateur correspondant) :

```
Mesh::Vertex_range r = m.vertices() ;
for(Mesh::Vertex_range::iterator v_it = r.begin() ; v_it != r.end() ; ++v_it)
{
    ...
}
```

On notera que pour récupérer l'indice du sommet pointé par un itérateur (parfois aussi appelé descripteur) :

```
Mesh::Vertex_index vh = *v_it ;
ou (cf typedef dans starter_defs.h)
vertex_descriptor vh = *v_it ;
```

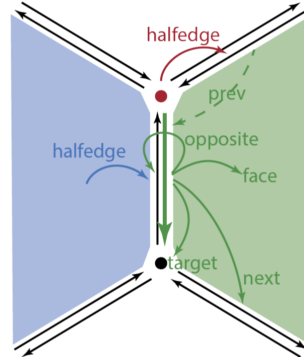
Et pour récupérer l'index d'une demi-arête touchant le sommet :

```
Mesh::Halfedge_index hei = m.halfedge(*v_it) ;
```

Récupérer les informations stockées dans la structure de demi-arêtes ...

Comme indiqué initialement dans la structure de demi-arête, sommets, faces et demi-arêtes stockent des pointeurs sur des structures voisines. D'une manière générale, on trouvera la référence des maillages à :

Pour les récupérer :



— Chaque **sommet** pointe sur une demi-arête **entrante**

```
Mesh::Vertex_index vi ...
Mesh:: Halfedge_index hei = m.halfedge(vi);
```

— Chaque **face** pointe sur une des demi-arêtes la bordant

```
Mesh::Face_index fi ...
Mesh:: Halfedge_index hei = m.halfedge(fi);
```

— Chaque **demi-arête** (`Mesh::Halfedge_index hei ... ;`) permet de récupérer :

  — Un sommet vers lequel elle pointe

```
Mesh:: Vertex_index vi = m.target(hei);
```

  — La face à laquelle elle appartient

```
Mesh:: Face_index fi = m.face(hei);
```

  — La demi-arête suivante de la face (sans anti-horaire)

```
Mesh:: Halfedge_index hei2 = m.next(hei);
```

  — La demi-arête opposée

```
Mesh::Halfedge_index hei2 = m.opposite(hei);
```

  — La demi-arête précédente de la face :

```
Mesh::Halfedge_index hei2 = m.prev(hei);
```

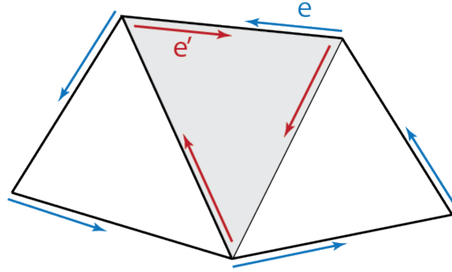
Parcourir les demi-arêtes voisines

Etant donnée une demi-arête, précisons que CGAL aura un comportement différent pour le parcours du voisinage selon que l'arête est sur le bord (voisine d'une face vide) ou non du maillage (cette propriété dérive de la structure de demi-arête).

Pour tester si une demi-arête est sur le bord :

```
Mesh:: Halfedge_index hei ;
if(m.is_border(hei)) ; // true/false
```

Cette fonction peut également s'appliquer aux sommets (cf. manuel).



— L'arête `e` se trouve sur le bord du maillage (donc par adjacente à une face). Dans ce cas, récupérer les 1/2 arêtes voisines permettra de parcourir le bord (c'est-à-dire les arêtes bleues)
— L'arête `e'` n'est pas une arête de bord (donc elle touche une face), les 1/2 arêtes voisines seront donc celles de la face

Etant donné l'*index* de demi-arête précédent, on passe alors à la demi-arête suivante (sur le bord) :

```
hei2 = m.next(hei);
```

Et à la demi-arête précédente avec `prev`.

Se déplacer dans le cercle des premiers voisins : les circulateurs
Plus généralement, OpenMesh offre de nombreux itérateurs sur le cercle des premiers voisins ; on les appelle circulateurs.
A titre d'exemple, voici comment parcourir les sommets du voisinage d'un sommet (on suppose que `v` est un index de sommet dont on veut parcourir le voisinage) :

```
CGAL::Halfedge_around_target_circulator<Mesh> hebegin(m.halfedge(v),m), hedone(hebegin) ;
do {
    // .../...
    ++hebegin ;
} while (hebegin != hedone) ;
```

On peut donc circuler autour d'un sommet (et trouver toutes les primitives géométriques partageant le même *target*, ie. sommet destination). Dans l'exemple ci-dessus, le *target* est le sommet `v`. Quatre circulateurs existent pour itérer sur les sommets/demi-arêtes/arêtes/faces autour du *target* :

`CGAL::Vertex_around_target_circulator<Mesh>` : itérer sur les sommets voisins du *target*
`CGAL::Halfedge_around_target_circulator<Mesh>` : itérer sur les demi-arêtes voisines du *target*
`CGAL::Edge_around_target_circulator<Mesh>` : itérer sur les arêtes voisines du *target*
`CGAL::Face_around_target_circulator<Mesh>` : itérer sur les faces voisines du *target*

Tous ces circulateurs fournissent des pointeurs sur des indexes.

Il existe une large variété de circulateurs : `CGAL::Face_around_face_iterator` ... Tous fonctionnement sur le même principe ...

3.3 Modifier le maillage (ajout/suppression)

L'ajout de primitive se fait par les méthodes propres du maillage :
— `add_vertex`
— `add_edge`
— `add_face`

Un exemple :

```
vertex_descriptor u = m.add_vertex(K::Point_3(0,1,0));
vertex_descriptor v = m.add_vertex(K::Point_3(0,0,0));
vertex_descriptor w = m.add_vertex(K::Point_3(1,0,0));
vertex_descriptor x = m.add_vertex(K::Point_3(1,1,0));
face_descriptor f = m.add_face(u,v,w,x);
```

La suppression d'un sommet et d'une arête est réalisée par des méthodes propres du maillage (attention, si l'élément appartient à une arête ou face, il faut les avoir supprimées au préalable) :

- `remove_vertex`
- `remove_edge`

Par souci d'efficacité, ces opérations ne réalisent pas la suppression (modification des listes de sommets/arêtes/faces, modification des demi-arêtes), mais marquent les éléments avec une propriété `removed`.

La suppression réelle est réalisée par appel à la méthode propre `collect_garbage()`.

La suppression des faces implique de nombreuses modifications sur les demi-arêtes de bord. L'opération haut niveau effectuant ces opérations topologique est située dans un module regroupant ces opérations de haut niveau : celui des **opérations Eulériennes**. En l'occurrence, pour supprimer une face `f` :

- `CGAL::Euler::remove_face(m.halfedge(f), m) ;`

Ce module comporte de nombreuses autres opértions de haut niveau sur la topologie (lien vers le manuel : opérations Eulériennes).

3.4 Couleurs ... propriétés ... et propriétés à la carte

Pour représenter les informations calculées, il est souvent utile de créer des cartes de couleurs (donc de colorer les points / arêtes / faces en fonction de la valeur calculée). Cela fait plus généralement partie des propriétés : attribuer des propriétés aux sommets/arêtes/faces, c'est y stocker des informations supplémentaires (couleur, normale, texture).

La seule propriété définie de manière systématique est le **point** associé à un sommet : Donc :

- Récupérer le point associé à un sommet `v` dans un maillage `m` : `m.point(v)`

Mais il est très simple d'ajouter une propriété d'un type quelconque. Toute propriété est codée comme un tableau associatif (map) associant aux indices la propriété concernée. Ci-dessous un exemple de code définissant une propriété de normale associée aux sommets dans un maillage `m` :

```
Mesh::Property_map<Mesh::Vertex_index,K::Vector_3> normal; // nom de la propriété
// template1 : propriété attachée aux sommets
// template2 : propriété contient un vecteur 3
bool created_normal; // booléen testant si la propriété a bien été créée
boost::tie(normal, created_normal) =
              m.add_property_map<Mesh::Vertex_index,K::Vector_3>("v:normal", K::Vector_3(0.,0.,0.));
// v:normal : nom interne de la propriété CGAL
assert(created_normal);
```

Puis libération de l'espace mémoire :

```
m.remove_property_map(normal) ;
```

La propriété est un tableau associatif, donc l'accès se fait simplement par :

```
normal[v_ind] ; // simple appel par l'indice de vertex
```

Cela peut s'appliquer à toutes les propriétés et toutes les structures (donc on peut associer n'importe quel type d'information à des sommets, arêtes ou demi-arêtes, faces) ...

En particulier, on ajoutera souvent des propriétés booléennes pour étiqueter des sommets/arêtes ... en cours d'algorithme, ou des propriétés `double` pour stocker des informations comme la courbure ...

3.5 Opérations vectorielles et géométriques

Pour récupérer les coordonnées d'un sommet `v` (donc d'un `Mesh::Vertex_index`) :

```
K::Point_3 p = m.point(v) ;
```

Opérations vectorielles

La plupart des opérations vectorielles standard sont disponibles directement et les opérateurs ont été surchargés. Les pages de documentation :
— documentation opérateurs vectoriels
— documentation fonctions géométriques.
listent la plupart des ces opérations et fonctions sur les points et vecteurs.
Attention ! CGAL distingue clairement les points `K::Point_3` (points affines) et les vecteurs `K::Vector_3`. Un vecteur peut être obtenu comme la différence de deux points, un point peut être translaté d'un vecteur, etc. ... Donc attention à la cohérence de ce que vous écrivez : il est impossible de sommet deux points par exemple (ce sont des vecteurs qu'on somme ...).
On peut citer pour les vecteurs :

`operator*`	vecteur * scalaire ou scalaire * vecteur ou vecteur * vecteur (produit scalaire) vecteur * vecteur (composante à composante)
`operator+` `operator-`	vecteur + vecteur vecteur - vecteur
`squared_length()`	norme euclidienne au carré

Opérations géométriques

Quelques opérations géométriques disponibles (noyau *3D linear geometry Kernel* des fonctions géométriques cité ci-dessus) :

`CGAL::approximate_angle`	angle $\widehat{P_1P_2P_3}$
`CGAL::approximate_dihedral_angle`	angle diédral entre deux triangles
`CGAL::cross_product`	produit vectoriel
`CGAL::centroid`	centroïde de points ou triangle
`CGAL::normal`	normale à une face triangulaire
...	...