Modélisation et géométrie discrète

Compte-rendu TP9

Cartes combinatoires généralisées

Louis Jean
Master 1 IMAGINE
Université de Montpellier

28 novembre 2023

Table des matières

1 Introduction

Ce travail pratique en **C++** et **OpenGL** (qui repose sur une base de code fournie) nous plonge dans l'étude des cartes combinatoires, une approche de la topologie algorithmique. Au cœur de ce TP se trouve l'exploration des gmaps, une structure de données puissante pour représenter et manipuler des formes géométriques en 3D.

2 La structure de 2-G carte

2.1 Question 1.a)

La fonction **add_dart** a été implémentée pour créer un nouveau brin et lui attribuer un identifiant unique (**id_t**). Chaque brin est initialisé de sorte que ses relations alpha (**alpha_0**, **alpha_1**, **alpha_2**) pointent sur lui-même, représentant ainsi des points fixes.

```
1 GMap::id_t GMap::add_dart() {
2     id_t dart = maxid;
3     maxid++;
4     alphas[dart] = alpha_container_t(dart,dart,dart);
5     return dart;
6 }
```

2.2 Question 1.b)

Deux versions de la fonction **alpha** ont été écrites. La première applique une relation alpha (**alpha_0**, **alpha_1** ou **alpha_2**) à un brin spécifié. La seconde fonction applique une composition de relations alpha dans l'ordre inverse à un brin donné. Ces fonctions permettent de naviguer dans la structure de la carte combinatoire.
La fonction **is_free** vérifie si un brin est libre pour un degré alpha spécifié, c'est-à-dire si l'application de l'alpha sur ce brin le renvoie sur lui-même, indiquant qu'il n'est pas lié à un autre brin pour cette relation.

```
1  GMap::id_t GMap::alpha(degree_t degree, id_t dart) const
       ↪ {
2      assert(degree < 3);
3      assert(dart < maxid);
4      return alphas.at(dart)[degree];
5  }
6
7  GMap::id_t GMap::alpha(degreelist_t degrees, id_t dart)
       ↪ const {
8      std::reverse(degrees.begin(), degrees.end());
9      for(degree_t degree : degrees){
10         dart = alpha(degree,dart);
11     }
12     return dart;
13 }
14
15 bool GMap::is_free(degree_t degree, id_t dart) const {
16     assert(degree < 3);
17     assert(dart < maxid);
18     return alpha(degree,dart) == dart;
19 }
```

2.3 Question 1.c)

La fonction **link_darts** lie deux brins pour un degré alpha spécifié, à condition que les deux brins soient libres pour ce degré. Cette fonction modifie les relations alpha des brins pour les relier entre eux.

```
bool GMap::link_darts(degree_t degree, id_t dart1, id_t
    ↪ dart2) {
    if (!is_free(degree,dart1)) return false;
    if (!is_free(degree,dart2)) return false;

    alphas.at(dart1)[degree] = dart2;
    alphas.at(dart2)[degree] = dart1;

    return true;
}
```

2.4 Question 1.d)

Enfin, la fonction **is_valid** teste la validité de la structure de la carte combinatoire en vérifiant que les relations **alpha_0** et **alpha_1** sont des involutions sans points fixes, que **alpha_2** est une involution, et que la composition **alpha_0** o **alpha_2** est également une involution.

```
bool GMap::is_valid() const {
    for(id_t dart : darts()) {
        if(alpha({0,0},dart) != dart) return false;
        if(alpha(0,dart) == dart) return false;
        if(alpha({1,1},dart) != dart) return false;
        if(alpha(1,dart) == dart) return false;
        if(alpha({2,2},dart) != dart) return false;
        if(alpha({0,2,0,2},dart) != dart) return false;
    }
    return true;
}
```

3 Les parcours de la structure

Cette fonction calcule l'orbite d'un brin donné en utilisant une liste de relations alpha. Elle commence avec un ensemble de brins à traiter, initialement contenant le brin spécifié. Pour chaque brin dans cet ensemble, si ce n'est pas déjà marqué, il est ajouté à l'ensemble des résultats et marqué comme traité. Ensuite, pour chaque relation alpha dans la liste, la fonction calcule le brin correspondant et l'ajoute à l'ensemble des brins à traiter. Ce processus se poursuit jusqu'à ce que tous les brins soient traités, fournissant ainsi l'ensemble complet des brins dans l'orbite.

```
1  GMap::idlist_t GMap::orbit(const degreelist_t& alphas,
       ↪ id_t dart) const
2  {
3      idlist_t result;
4      idset_t marked;
5      idlist_t toprocess = {dart};
6
7      while (!toprocess.empty()){
8          id_t d = toprocess.front();
9          toprocess.erase(toprocess.begin());
10         if (marked.count(d) == 0){
11             result.push_back(d);
12             marked.insert(d);
13             for (degree_t degree : alphas) {
14                 toprocess.push_back(alpha(degree, d));
15             }
16         }
17
18     }
19     return result;
20 }
```

Cette fonction calcule l'orbite ordonnée d'un brin en appliquant de manière répétée les relations alpha de la liste sur ce dernier. Elle maintient un index pour parcourir les relations alpha et applique ces relations à partir du brin spécifié. À chaque étape, le brin résultant est ajouté à la liste des résultats. La fonction continue jusqu'à ce que le brin initial soit revisité, complétant ainsi l'orbite.

```
1 GMap::idlist_t GMap::orderedorbit(const degreelist_t&
    ↪ list_of_alpha_value, id_t dart) const {
2     idlist_t result;
3     id_t current_dart = dart;
4     unsigned char current_alpha_index = 0;
5     size_t n_alpha = list_of_alpha_value.size();
6     do {
7         result.push_back(current_dart);
8         degree_t current_alpha = list_of_alpha_value[
              ↪ current_alpha_index];
9         current_dart = alpha(current_alpha, current_dart)
              ↪ ;
10        current_alpha_index = (current_alpha_index+1) %
              ↪ n_alpha;
11    } while (current_dart != dart);
12    return result;
13 }
```

4 Le plongement géométrique

La fonction **get_embedding_dart** est essentielle pour identifier le brin porteur de l'information de plongement d'une i-cellule dans une carte combinatoire 3D. Elle opère en parcourant l'orbite d'un brin initial, constituée par les relations **alpha_1** et **alpha_2**, qui ensemble représentent l'ensemble des brins d'une i-cellule, particulièrement un sommet dans le cas d'une 0-cellule. La fonction vérifie si l'un de ces brins a déjà une propriété de plongement (comme des coordonnées spatiales) associée dans le dictionnaire **properties**. Si un tel brin est trouvé, il est retourné comme détenteur de l'information de plongement pour cette i-cellule. Dans le cas où aucun brin de l'orbite ne possède de propriété, le brin initial est retourné, indiquant qu'aucune donnée de plongement n'est encore attribuée à la i-cellule concernée. Cette méthode assure une gestion efficace et une visualisation claire des informations géométriques dans la structure de la carte combinatoire 3D.

```
template<class T>
GMap::id_t EmbeddedGMap<T>::get_embedding_dart(id_t dart)
    ↪ const
{
    for (id_t d : orbit({1,2}, dart)){
        if (properties.count(d) == 1){
                return d;
        }
    }
    return dart;
}
```

5 La couture liant deux éléments

La fonction **sew_dart** est conçue pour lier deux éléments de degré spécifié, avec une logique adaptée selon le degré. Pour un degré 1, elle effectue une liaison simple entre les brins en utilisant **alpha_1**. Pour les degrés 0 et 2, elle calcule d'abord les orbites correspondantes (orbite par **alpha_2** pour le degré 0 et orbite par **alpha_0** pour le degré 2) des brins donnés. La fonction vérifie ensuite si ces orbites ont la même taille, un prérequis pour une liaison cohérente. Si elles sont compatibles, la fonction procède à la liaison des brins correspondants dans ces orbites avec la relation alpha appropriée.

```
1  bool GMap::sew_dart(degree_t degree, id_t dart1, id_t
       ↪ dart2) {
2      if (degree == 1) link_darts(1, dart1, dart2);
3      else {
4          degreelist_t alpha_list = {0};
5          if (degree == 0) alpha_list = {2};
6
7          idlist_t orbit1 = orbit(alpha_list, dart1);
8          idlist_t orbit2 = orbit(alpha_list, dart2);
9
10         if (orbit1.size() != orbit2.size()) return false;
11
12         idlist_t::const_iterator it1 =   orbit1.begin();
13         idlist_t::const_iterator it2 =   orbit2.begin();
14         for (; it1 != orbit1.end() ; ++it1, ++it2)
15             link_darts(degree, *it1, *it2);
16         }
17     return true;
18 }
```

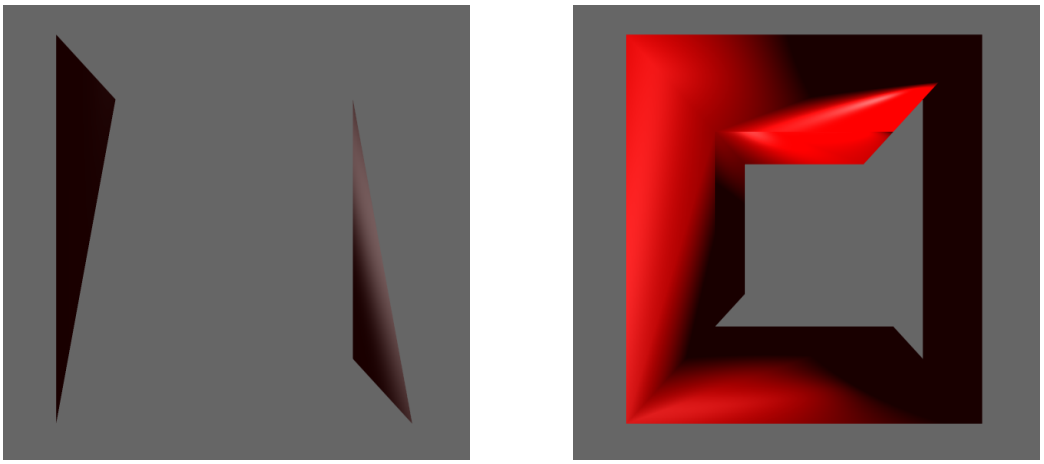
6 Caractéristique d'Euler-Poincaré

La fonction **eulercharacteristic** est conçue pour calculer la caractéristique d'Euler-Poincaré, une mesure topologique importante. La fonction suit une approche itérative pour calculer cette caractéristique. Elle initialise d'abord un signe à 1 et la caractéristique à 0. Ensuite, elle boucle sur les degrés de 0 à 2 (représentant S, A, et F dans la formule S - A + F). Pour chaque degré, elle ajoute à la caractéristique le nombre d'éléments de ce degré, multiplié par le signe actuel. Le signe est alterné à chaque itération (multiplié par -1) pour refléter l'alternance de soustraction et d'addition dans la formule de la caractéristique d'Euler-Poincaré. La fonction retourne finalement la valeur calculée de la caractéristique, qui représente la différence entre le nombre de sommets (S), d'arêtes (A) et de faces (F) dans la structure. Cette méthode permet de déterminer des propriétés topologiques fondamentales de la structure étudiée.

```
1 int GMap::eulercharacteristic() const {
2     int sign = 1;
3     int characteristic = 0;
4     for (unsigned char degree = 0 ; degree < 3 ; ++degree
          ↪ ){
5         characteristic += sign * elements(degree).size();
6         sign *= -1;
7     }
8     return characteristic;
9 }
```

7 Visualisation

Pour la visualisation, je n'ai pas réussi à faire mieux que les affichages ci-dessous.



(a) Cube (b) Cube troué

Figure 1: Visualisation de cartes combinatoires

8 Le dual d'une 2-G carte

La fonction **dual** génère le dual d'une carte combinatoire 3D. Elle commence par créer une nouvelle carte **dual_gmap**, inversant les valeurs des **alpha_2** et **alpha_0** de la carte originale pour transformer la structure topologique, échangeant ainsi les rôles des sommets et des faces. Ensuite, pour chaque brin représentant une face dans la carte originale, elle calcule le barycentre et l'associe au brin correspondant dans le dual, positionnant les sommets du dual aux centres des faces de la carte originale. Ce processus crée une version duale de la carte combinatoire, où les sommets et les faces sont inversés.

```
1
2 GMap3D GMap3D::dual()
3 {
4     GMap3D dual_gmap;
5     for (idalphamap_t::const_iterator it = alphas.begin()
      ↪ ; it != alphas.end(); ++it){
6         dual_gmap.alphas[it->first] = it->second.flip();
7     }
8     dual_gmap.maxid = maxid;
9     for (id_t face_dart : elements(2)){
10        vec3_t pos = element_center(2, face_dart);
11        dual_gmap.set_position(face_dart,pos);
12    }
13    return dual_gmap;
14 }
```

9 Comparaison avec des structures alternatives

Pour encoder une simple 2-Carte :

- **Changements nécessaires :** la transition vers une simple 2-Carte impliquerait une simplification de la structure de données, se concentrant uniquement sur les sommets, les arêtes et les faces, sans les relations complexes des cartes combinatoires.
- **Avantages :** simplification de la structure, la rendant plus intuitive et peut-être plus performante pour certains calculs spécifiques.
- **Inconvénients :** perdrait en flexibilité et en capacité à représenter des structures topologiques complexes.

10 Conclusion

Ce TP sur les cartes combinatoires a été un exercice intéressant, mettant en évidence la possibilité de représenter des maillages par d'autres structures de données. J'ai apprécié y prendre part.

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.