TP1 - Maillages

Alexandra Bac

Modélisation géométrique
Polytech Marseille - IRM 4A

Ce TP porte sur les maillages et on utilisera la bibliothèque C++ (très connue) CGAL. Le tuto CGAL vous donnera tout ce dont vous avez besoin pour l'aborder. Par ailleurs, vous téléchargerez le matériel de TP comportant un “starting code” qui vous aidera à démarrer.

Démarrage à partir du “starting code”

- Avant toute chose, éditez votre fichier `.bashrc` à la racine de votre répertoire et ajoutez :

  ```
  export PATH=/amuhome/bruasse.a/Public/Libs/cmake-3.22.2/bin:${PATH}
  export PATH=/amuhome/bruasse.a/Public:${PATH}
  ```

  Fermez votre terminal puis rouvrez-en un (ou exécutez `source ~/.bashrc`).

- Téléchargez l'archive et décompressez-la.
- Allez dans le répertoire `build`
- `cmake ..`
- `make`
- `./poly_starter ../data/bunnyLowPoly.obj`

Vous pouvez maintenant visualiser le fichier généré (`test.off`) avec meshlab (comme vous avez inclus le chemin vers mon répertoire public, il suffit de taper `meshlab test.off` dans un terminal).

Organisez correctement votre code en créant vos fonctions liées aux maillages dans `mesh_functions.hpp` et `mesh_functions.cpp`.

Maintenant, à vous !

1 Parcourir le maillage : degrés, courbures et cartes associées

Exercice 1.

1. Commencez par écrire une fonction :

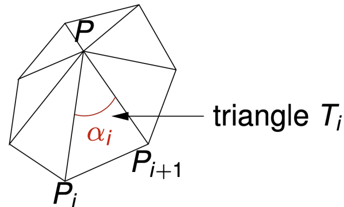
   ```
   vector<Mesh::Vertex_index> vneigh (Mesh::Vertex_index v, Mesh &m)
   ```

   renvoyant (en utilisant la structure de demi-arêtes et non les circulateurs) les indices des voisins du sommet `v`.

2. Définir une propriété entière `deg` attachée aux sommets et y calculer le degré de chaque sommet.

3. Créer une carte de couleurs par degrés (on associera une couleur à chaque degré)

Exercice 2 (Courbures discrètes : prototype simple)**.**
Dans cet exercice, on souhaite estimer la courbure en chaque sommet du maillage. Pour cela, on utilisera un estimateur très simple de la courbure gaussienne au sommet P :



$$\mathrm{K}_P = \frac{2\pi - \sum_i \alpha_i}{\sum_i \mathrm{Aire}(T_i)}$$

avec $\mathrm{Aire}(T_i) = \frac{1}{2}\left\|\overrightarrow{PP_i} \wedge \overrightarrow{PP_{i+1}}\right\|$

1. En partant des fonctions précédentes, écrire une fonction :

   ```
   void gaussian_curv(Mesh &m, Mesh::Property_map<vertex_descriptor,double> &Kcourb) ;
   ```

   prenant un maillage et une propriété permettant d'enregistrer les courbures gaussiennes et calculant et stockant la courbure en chaque sommet.

2. Complétez la classe `distrib` permettant de réaliser des statistiques de base sur un ensemble de valeurs :

   ```
   #ifndef distrib_hpp
   #define distrib_hpp

   #include <iostream>
   using namespace std ;

   template <typename T>
   class distrib {
   vector<T> _data ;
   T _mean_sum, _min, _max, _stdev ;
   public:
   distrib() : _mean_sum(0.), _min(0.), _max(0.), _stdev(0.)  { } ;
   distrib(vector<T> &vec) : _data(vec), _mean_sum(0.), _min(0.), _max(0.), _stdev(0.)
   ```

```
{
if (_data.size()>0)
{
_mean_sum = _data.at(0) ;
_min = _data.at(0) ;
_max = _data.at(0) ;
for (int i=1; i<_data.size(); ++i)
{
                        // TODO
}
}
update_stdev() ;
} ;

void update_stdev ()
{
T mean = _mean_sum / _data.size(), tmp ;
for (int i=0; i<_data.size(); ++i)
{
tmp = _data.at(i) - mean ;
_stdev += tmp*tmp ;
}
_stdev = sqrt(_stdev / _data.size()) ;
} ;
// Accesseurs
T get_mean () { return _mean_sum/_data.size() ; } ;
T get_stdev () { update_stdev() ; return _stdev ; } ;
T get_min () { return _min ; } ;
T get_max () { return _max ; } ;
void add_data (T dat)
{
if (_data.size() == 0)
{
_min = dat ;
_max = dat ;
}
else
{
// TODO
}
_data.push_back(dat) ; _mean_sum += dat ;
}
void clear ()
{ _data.clear() ; _min = 0. ; _max = 0. ; _mean_sum = 0. ; _stdev = 0. ;}
};

#endif /* distrib_hpp */
```

3. En utilisant cette classe, calculer la moyenne ($kmean$), min ($kmin$), max ($kmax$), écart type ($kstdev$) de la courbure gaussienne sur l'ensemble des sommets.

4. On veut colorer les sommets selon leur courbure de telle sorte que :

$$
\begin{aligned}
K_P \in [kmin, kmean - kstdev] &\mapsto \text{bleu} \\
K_P \in [kmean - kstdev, kmean + kstdev] &\mapsto \text{couleur entre bleu et rouge} \\
K_P \in [kmean + kstdev, kmax] &\mapsto \text{rouge}
\end{aligned}
$$

   Que fait le code ci-dessous :

```
CGAL::Color color_ramp(double vmin, double vmax, double mean, double stdev, double v, CGAL::Color col1, CGAL::Color col2)
{
stdev = stdev/2 ;
if (vmin < mean-stdev)
vmin = mean-stdev ;
if (vmax > mean+stdev)
vmax = mean+stdev ;

if (v < vmin)
v = vmin ;
else if (v > vmax)
v = vmax ;
double per ;
CGAL::Color c ;
CGAL::Color col_first, col_second ;
if (v <= mean)
{
per = (v-vmin)/(mean-vmin) ;
col_first = col1 ;
```

```
col_second = CGAL::white() ;
}
else
{
per = (v-mean)/(vmax-mean) ;
col_first = CGAL::white() ;
col_second = col2 ;
}

c.red() = floor(col_first.red() + per*(col_second.red()-col_first.red())) ;
c.green() = floor(col_first.green() + per*(col_second.green()-col_first.green())) ;
c.blue() = floor(col_first.blue() + per*(col_second.blue()-col_first.blue())) ;
return c ;
}
```

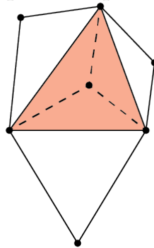
5. Construire la carte de couleur du maillage par la courbure gaussienne en chaque sommet.

6. Tester sur différents modèles et interpréter.

2 Une petite chaîne de traitement de maillages

Le but de cette partie est de pouvoir raffiner le maillage dans les zônes fortement courbées puis de le lisser localement. L'idée est que les fortes courbures provoquent des lignes saillantes que l'on souhaite diminuer (alors que dans les zônes plates, il est possible de conserver de grands triangles sans pour cela avoir des facettes marquées).

Exercice 3 (Subdivision barycentrique d'un triangle)**.** La subdivision barycentrique est l'une des méthodes les plus simples permettant un raffinement local :



Ecrire une fonction :

```
vector<face_descriptor> bary_sub(Mesh::Face_index f, Mesh &m)
```

réalisant cette opération sur la face `f` et renvoyant l'ensemble des faces créées.

Exercice 4 (Subdivision des zônes fortement courbées)**.** Pourquoi le code suivant permet-il de trier les sommets par courbure décroissante ? En particulier, quel est le 3ème argument de la fonction `std::sort` et que signifie-t-il ?

```
vector<Mesh::Vertex_index> sorted_vert ;
Mesh::Vertex_range r = m.vertices() ;

for (Mesh::Vertex_range::iterator it = r.begin(); it != r.end(); ++it)
{
   sorted_vert.push_back(*it) ;
```

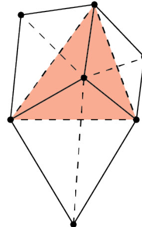
```
}
// Tri des sommets par ordre décroissant de |Kcourb|
std::sort(sorted_vert.begin(), sorted_vert.end(), // Lambda expression begins
          [&Kcourb](vertex_descriptor &v1, vertex_descriptor &v2)
             {
                return abs(Kcourb[v1]) > abs(Kcourb[v2]) ;
             } // end of lambda expression) ;
         );
```

En utilisant ce code, écrire une fonction :

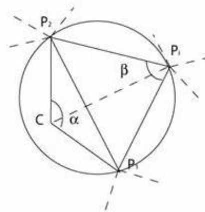
```
vector<Mesh::Halfedge_index> subdiv_curved(Mesh &m, //
                   const Mesh::Property_map<vertex_descriptor,double> &Kcourb, //
                   double percent)
```

Subdivisant les triangles voisins des `percent` pourcents des sommets où la courbure est la plus importante. Votre fonction renverra les faces créées (en fait, on identifiera ces faces via une demi-arête leur appartenant, ce qui est équivalent à l'indexe de la face).

Exercice 5 (Permutation des arêtes (flip))**.** Comme vous le voyez, le résultat n'est pas très élégant ... La subdivision barycentrique crée de nombreux triangles obtus (donc très loin de critères de Delaunay). Mais bonne nouvelle, cela peut généralement être facilement résolu en permuttant les arêtes joignant deux faces (généralement appelé “flip”) :



La condition pour décider d'échanger une arête est inspirée par la condition de Delaunay (2D) :



L'arête est échangée si $\alpha + \beta > \pi$.

1. Ecrire une fonction :

   ```
   vector<Mesh::Halfedge_index> edges_from_faces (vector<Mesh::Halfedge_index> &hef, //
                                                  Mesh &m)
   ```

renvoyant la liste des demi-arêtes bordant les faces qui ont été créées (identifiées par `hef`). Bien entendu, cette liste ne doit pas comporter de redondance. On définira donc une propriété booléenne temporaire permettant de marquer les demi-arêtes au fur et à mesure, puis d'en extraire la liste.

2. Puis écrire la fonction :

```
vector<Mesh::Vertex_index> flip_edges(vector<Mesh::Halfedge_index> &hef, //
                                       Mesh &m)
```

   Testant chacune de ces arêtes et l'échangeant si la condition est vérifiée et renvoyant l'ensemble des sommets voisins d'une arête échangée.

Exercice 6 (Lissage Laplacien)**.** Enfin, puisque nous avons raffiné localement le maillage en insérant de nouveaux sommets et triangles, il est naturel de lisser les zônes ainsi subdivisées.

Le lissage Laplacien s'appuie sur un modèle physique bien connu : celui de la diffusion de la chaleur. La modéliation de la conduction thermique (régularistion d'un champ de température) s'exprime par :

$$\frac{\partial}{\partial \mathrm{t}} x = \lambda \Delta x$$

où Δ désigne l'opérateur Laplacien ($\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$).

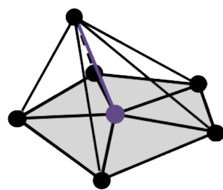
Cette équation est discrétisée pour filtrer la géométrie (points du maillage) :

$$\frac{\partial}{\partial \mathrm{t}} P(v_i) = \lambda \Delta P(v_i)$$

On aboutit à la mise à jour itérative :

$$P(v_i)^{t+1} \leftarrow P(v_i)^t - \lambda \Delta P(v_i)$$

où le Laplacien est approximé par :



$$\Delta P(v_i) = \frac{1}{|\mathcal{V}(v_i)|} \left(\sum_{v_j \in \mathcal{V}(v_i)} P(v_j) \right) - P(v_i)$$

Ecrire une fonction :

```
void laplac_smooth(vector<Mesh::Vertex_index> l, double lambda, Mesh &m)
```

réalisant une itération de filtrage laplacien sur tous les sommets de la liste `l` pour un paramère `lambda` passé en argument.

Si tout se passe bien vous devriez passer du maillage de gauche à celui de droite avec possibilité de “jouer sur le lissage” via le paramètre `lambda`.