Rapport TP segmentation de maillages

I- Introduction

Ce TP a pour but d'apprendre à parcourir les facettes d'un maillage et d'en extraire des informations. On peut récupérer la surface, le périmètre, le nombre de côtés et d'autres informations sur la face. Nous utiliserons principalement le calcul du périmètre. De plus, ce TP a aussi pour but de travailler sur les classes pour ranger certaines faces dans certaines familles. Ceci entraînera quelques notions liées à la topologie comme celle de la composante connexe.

II- Introduction au seuillage

Le but de cette première partie est de séparer les faces en différentes catégories.
Le plus simple pour commencer est de faire **deux catégories** en fonction de la valeur du **périmètre** de chaque face.

Pour faire cela, nous nous sommes inspirés de la fonction *normalizeMap* de measures.cpp. Si la face a un périmètre inférieur à un seuil, elle se trouve dans la première classe, sinon elle se trouve dans la deuxième classe. La définition du seuil est faite grâce à la moyenne de tous les périmètres des faces du solide.
Ensuite, dans *writeOFFfromValueMap* on change la répartition des couleurs en fonction de la classe. On a donc deux couleurs distinctes vu que pour le moment on s'occupe d'échantillonner les faces en seulement deux groupes distincts.

Le code est écrit dans le fichier color.cpp. On calcule en premier temps la moyenne pour ensuite répartir dans les deux classes. On retourne un Facet_int_map dont le int correspond à la classe à laquelle appartient la "facet".

```
Facet_int_map simpleThershold(Facet_double_map &facetMap){
    double avgValue = 0;
    int nb = 0, em=0;
    Facet_iterator i = 0;
    Facet_int_map PerimInt;

    // look for min and max value in the map
    for (const auto &elem : facetMap)
    {
        avgValue += elem.second;
        nb++;
    }
    avgValue /= nb;

    for (const auto &elem : facetMap)
    {


        if (elem.second > avgValue)
        {

            PerimInt[elem.first] = 1;

        } else {
            PerimInt[elem.first] = 0;
        }
    }
    return PerimInt;
}
```

III- Génération de maillages colorés

Maintenant que nous avons réussi avec une couleur pour chacune des deux classes, le but est de généraliser à N couleurs pour N classes. Nous avons donc choisi de créer une structure RGB et un tableau de structures. Ainsi nous avons :

```
#define N_CLASS 10 //nombre de classes

struct color
{
	float R;
	float G;
	float B;
};

struct color list_class[N_CLASS ];
```

Une fois la structure de couleurs faite, nous avons initialisé quelques couleurs dans un tableau de structure color nommé colorPalette[]. La taille de ce tableau est supérieure au nombre de classe

```
// Define the color palette
color colorPalette[Nb_Color] = {
    {1.0, 0.0, 0.0},   // red
    {1.0, 0.5, 0.0},   // orange
    {1.0, 1.0, 0.0},   // yellow
    {0.0, 1.0, 0.0},   // green
    {0.0, 0.5, 1.0},   // blue-green
    {0.0, 0.0, 1.0},   // blue
    {0.5, 0.0, 1.0},   // purple
    {1.0, 0.0, 1.0},   // magenta
    {0.5, 0.5, 0.5},   // gray
    {0.0, 0.0, 0.0}    // black
};
```

Pour cette partie, au lieu d'avoir 0 et 1 pour le numéro de classe, on définit l'intervalle entre chaque classe puis on trouve à quelle classe appartient l'élément pour ensuite lui attribuer le numéro de classe.

```
float intervalSize = (maxValue - minValue) / N_Class;
int intervalIndex = (elem.second - minValue) / intervalSize;
ClassInt[elem.first] = intervalIndex;
```

La définition des couleurs pour chaque face est réalisée dans la fonction *writeOFFfromValueMap()* en utilisant simplement le numéro de classe comme indice pour le tableau de couleurs.

IV- Intégration des composantes connexes

Cette partie met en œuvre la connexité. Il ne faut plus qu'une couleur ait les faces de sa classe à différents endroits du solide. En soit, si une couleur est présente sur une face, seules ses faces voisines pourront avoir cette couleur. Cette contrainte est un ajout à la contrainte de classe liée à la taille du périmètre de la face vu dans la partie précédente.

Pour faire ceci, nous avons quelques changements à faire. Pour commencer, il faut un grand nombre de couleur. Ce nombre peut être égal au nombre de face. On va donc utiliser des couleurs aléatoires en jouant sur le triple aléatoire du RGB. On va donc utiliser le tableau de couleurs mais on va l'initialiser dans une fonction afin d'avoir autant de couleurs que de classes. Le Nb_Color est une variable globale qui donne le nombre de classes final.

```
void initColor(color *colorPalette){
    for(int i = 0; i < Nb_Color; i++){
            colorPalette[i].R = ( rand()*i % 1000 ) / 1000.;
            colorPalette[i].V = ( rand()*(50-i) % 1000 ) / 1000.;
            colorPalette[i].B= ( rand()*(100-i) % 1000 ) / 1000.;
    }
}
```

Le rand() donne un int aléatoire. Le modulo est utilisé pour avoir 1000 possibilités pour chaque couleur. Puis la division par 1000. permet de transformer ce chiffre en un float compris entre 0 et 1.

Ensuite on initialise le tableau dans la fonction *writeOFFfromValueMap()* :

```
    color colorPalette[Nb_Color] ;
    initColor(&colorPalette);
```

Pour finir, on utilise une boucle for pour attribuer la couleur de la face en fonction de sa classe.

```
        float redValue = colorPalette[facetIntM.at(i)].R;
        float greenValue = colorPalette[facetIntM.at(i)].V;
        float blueValue = colorPalette[facetIntM.at(i)].B;
```

La partie la plus importante de cette fin de TP est la distinction des classes. Le but étant de séparer les classes en plusieurs si jamais les faces d'une même classe ne sont pas toutes voisines. *visit_facet()* est la fonction utilisée pour séparer les classes entre elles. C'est ici que l'on regarde si une face a été visitée et que l'on regarde si ses voisins sont de la même classe ou non. Pour cela les demies-arêtes sont utilisées.

La visite se fait grâce à la fonction *segmentationParCC().* Pour chaque face, la fonction va visiter ses voisins et changer son numéro de classe en fonction de ce que retourne la visite.

```
for (Facet_iterator f = mesh.facets_begin(); f != mesh.facets_end();
++f)
{
      if (!visit[f])
      {
      visit_facet(mesh, f, visit, NewSeg, segmentation, nClass);
                  nClass++;
      }
}
```

V- Segmentation complexe

L'idée de cette partie est de regrouper les facets qui sont connexes et qui ont un paramètre proche. Pour cela, on se base sur le parcours précédent et on compare le tableau de données plutôt que celui d'une segmentation. Le but est de regarder la première face, regarder pour chaque voisin s'ils ont une caractéristique commune. En effet on va utiliser un seuil. Ce dernier est calculé grâce à une division de l'intervalle par le nombre de classes voulues.

L'explication n'étant pas simple, un exemple sera mieux pour illustrer : on prend les périmètre maximum et minimum des faces. On obtient un intervalle, on divise cet intervalle par le nombre de classes voulues. On obtient donc le seuil. Ensuite on parcourt les faces. Pour chaque face on regarde son périmètre et pour chaque voisin, on regarde si le périmètre est égale à celui de la face principale avec plus ou moins le seuil.

Avec cette méthode, il est possible d'avoir moins de classes finales que de classes initiales.

VI- Expérimentations

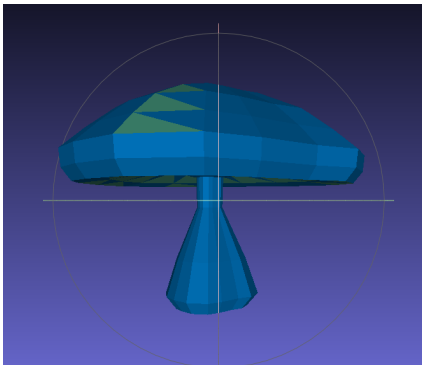
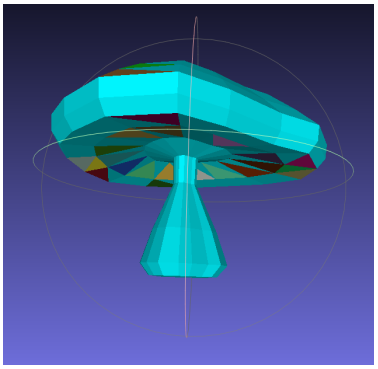
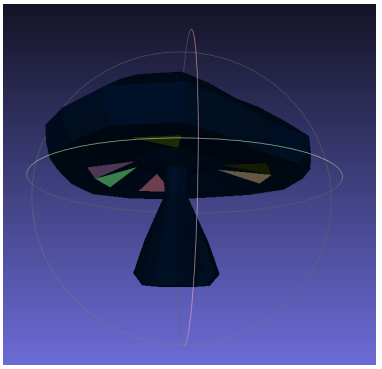
Nous allons montrer quelques exemples. Pour chacun d'eux, il y aura le screen de la segmentation vu en partie II, puis celle avec les composantes connexes et pour finir la segmentation complexe. Nous allons faire ceci pour 2 et 6 classes initiales et pour les paramètres périmètre et aire. Ceux-ci seront fait sur le mesh du champignon.

1- Périmètre / 2 classes

On peut voir ici qu'il y a beaucoup de faces de d'une même classe mais dispersées. C'est pourquoi on passe de 2 à 31 classes. Par contre on comprend que les périmètres sont assez proches car on redescend à 7 classes par la suite.

```
2
Le résultat a été exporté dans result.off !
31
Le résultat a été exporté dans resultCC.off !
7
Le résultat a été exporté dans resultcompl.off !
```

Seuillage | Composante Connexe | Complexe

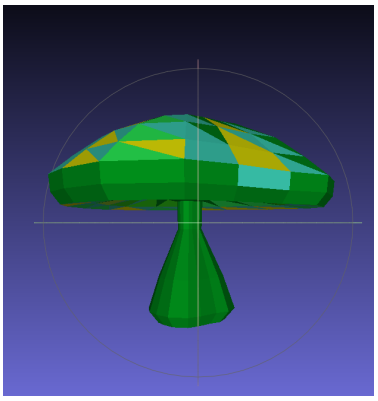
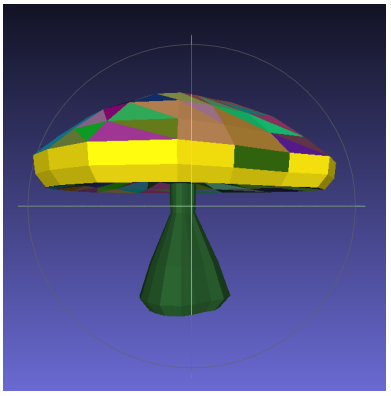
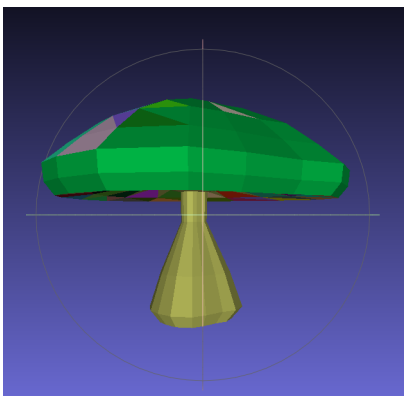




2- Périmètre / 6 classes

Pour 6 classes initiales, on peut faire les mêmes analyses en prenant en compte qu'il y a plus de classes à chaque fois.

```
6
Le résultat a été exporté dans result.off !
126
Le résultat a été exporté dans resultCC.off !
58
Le résultat a été exporté dans resultcompl.off !
```

Seuillage | Composante Connexe | Complexe

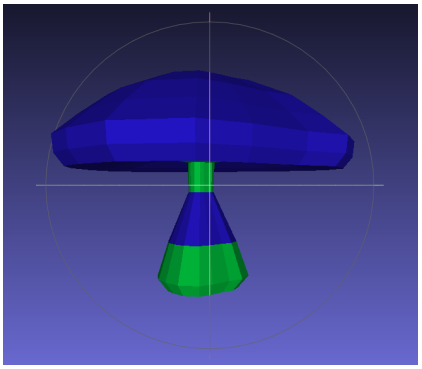
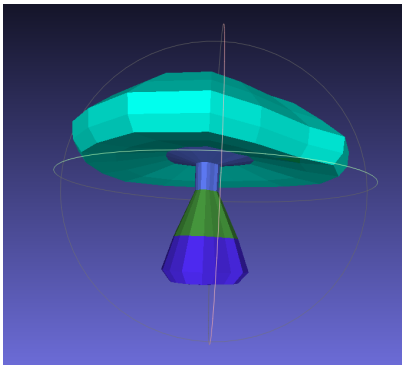
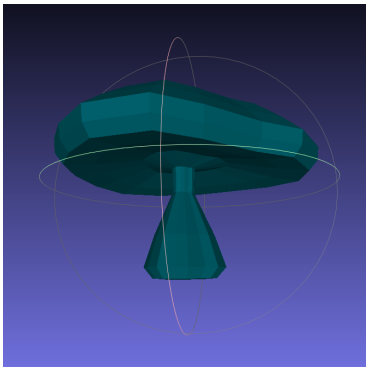




3- Aire / 2 classes

Ce qui est intéressant à noter dans cette configuration est le manque de classe pour la partie complexe de la segmentation. En effet, on passe de deux à une seule classe.

```
2
Le résultat a été exporté dans result.off !
5
Le résultat a été exporté dans resultCC.off !
1
Le résultat a été exporté dans resultcompl.off !
```

Seuillage | Composante Connexe | Complexe

4- Aire / 6 classes

Pour cette dernière expérimentation, on se rend encore compte que le nombre de classe diminue avec les aires.

```
6
Le résultat a été exporté dans result.off !
94
Le résultat a été exporté dans resultCC.off !
13
Le résultat a été exporté dans resultcompl.off !
```

Seuillage | Composante Connexe | Complexe