TD4: Simplification de maillage par *Vertex Clustering*

Option Majeure 2, Images de Synthèse Animées

1 Introduction

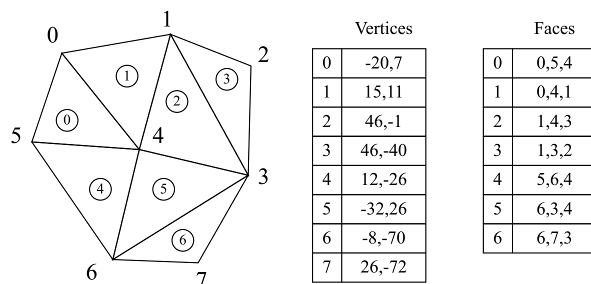
L'objectif de ce TD est d'implémenter l'algorithme de Rossignac et Borrel [RB93] pour la simplification de maillage. Le principe de cet algorithme est de grouper les sommets d'un modèle polygonal en *clusters*, puis de remplacer tout les points d'un cluster par un unique représentant, diminuant ainsi le nombre de sommets du modèle.

1.1 Squelette de programme

Pour ce TD, on vous fournit un squelette de programme capable de charger et afficher un modèle, ainsi qu'un ensemble de classes utilitaires (`Vec` pour représenter les vecteurs 3D, `AABBox` pour représenter une boîte englobante,...). Vous devrez compléter le fichier `viewer.cpp` pour effectuer les différents calculs.

1.2 Représentation du modèle 3D

On utilise une représentation par *Indexed Face Set* (IFS). Les sommets du modèle sont stockées dans une table de coordonnées, chaque sommet est référencé par son index dans la table. Chaque face du modèle est ensuite décrite par la liste des index des sommets qui la composent. Comme on ne considérera que des faces triangulaires, N faces peuvent être stockée dans une table d'entiers de taille $3N$. La figure 1 montre un exemple.



Vertices

0	-20,7
1	15,11
2	46,-1
3	46,-40
4	12,-26
5	-32,26
6	-8,-70
7	26,-72

Faces

0	0,5,4
1	0,4,1
2	1,4,3
3	1,3,2
4	5,6,4
5	6,3,4
6	6,7,3

FIG. 1 – Exemple d'*Indexed Face Set*

1.3 Les classes deque et map

La classe `deque` fait partie de la STL[1] et encapsule un tableau dynamique. Vous pouvez l'utiliser exactement comme un tableau C avec la notation `[]` mais vous n'avez pas à gérer les allocations mémoire. La fonction membre `push_back()` permet de rajouter des éléments. La fonction membre `size()` renvoie le nombre d'élements dans le tableau. Le paramètre *template* `<Vec>` indique que c'est un tableau d'objets de type `Vec`. Voici un exemple d'utilisation :

```
Cell c;

c.push_back(Vec(1,0,0));
c.push_back(Vec(0,1,0));
c.push_back(Vec(0,0,1));

Vec barycenter = (c[0]+c[1]+c[2])/c.size();
```

La classe `map` est une autre classe de la STL et encapsule une table d'association. Sa syntaxe avec l'opérateur `[]` la rend très proche d'un tableau "creux". Ci-dessous un exemple qui utilise une table d'association entier/bool qui indique si un entier est présent dans un tableau :

```
deque<int> entiers;
entiers.push_back(1);
entiers.push_back(3);
entiers.push_back(5);
entiers.push_back(7);
entiers.push_back(9);
entiers.push_back(11);

map<int,bool> estPresent;
for (int i=0;i<entiers.size();++i)
{
   estPresent[entiers[i]] = true;
}
```

[1] *Standard Template Library* du C++. Voir `http://artis.imag.fr/Membres/Xavier.Decoret/resources/stl_tutorial/` pour une introduction.

```
cout<<estPresent[2]<<endl;  // false
cout<<estPresent[9]<<endl;  // true
```

1.4 Compilation et exemples

Récupérez le source du TP avec :

```
tar zxvf ~holzschu/td4/td4.tar.gz
cd td4
```

Pour compiler, il faut d'abord générer le `Makefile` à l'aide de l'utilitaire `qmake` de Qt :

```
qmake
```

puis compiler avec `make`. Le programme produit s'appelle `td4`. Lancez

```
./td4 -h
```

pour voir quelles sont les données à lui passer sur la ligne de commande. Des fichiers d'exemples de modèles 3D au format `.smf`sont disponibles dans le répertoire `holzschuh/usr/share/data`.

2 Travaux à réaliser

2.1 Prise en main et OpenGL

Le squelette de programme fourni s'occupe de :

- *parser* la ligne de commandes pour récupérer des options (nom du fichier à charger, résolution de la grille) ;
- charger un modèle 3D en récupérant les coordonnées 3D des sommets (`vertices`) et les index des sommets composants les faces (`faces`) et en calculant une normale pour chaque face (`faceNormals`) ;
- calculer la boîte englobante `bbox` du modèle ;
- fournir une interface souris pour visualiser les résultats (grâce au QGLViewer).

Il vous reste à écrire le code pour l'affichage et le code pour calculer le niveau de détail. Toutes ces fonctionnalités seront dévelopées dans les méthodes `draw()` et `computeLOD()` de la classe `Viewer`. Cette classe est définie dans les fichiers `viewer.h` et `viewer.cpp`. Ce sont les seuls fichiers que vous avez à manipuler.

La fonction `draw()` est la fonction appellée pour dessiner le contenu de la fenêtre. Quand cette fonction est appellée, les matrices de transformation OpenGL ont été correctement mises pour correspondre aux interactions à la souris.

La fonction `keyPressEvent()` gère les interactions au clavier. Un certain nombre de touches sont prédéfinies. Par exemple, la touche ESC permet de quitter le programme et la touche F d'afficher le *framerate*.

Les fonctions `loadModel(const char*)` et `computeLOD(float)` sont appellées par le programme principale pour charger le modèle et calculer le niveau de détail. Vous allez devoir remplir ces deux fonctions.

À faire (a) : Écrire le code OpenGL permettant d'afficher le modèle dans la fonction `draw()`. On n'oubliera pas de spécifier la normale de chaque face.

À faire (b) : Modifier les fonctions `draw()` et `keyPressEvent()` pour que l'appui sur la touche `'w'` permette de basculer entre un affichage du modèle en fil de fer et un affichage "plein". On utilisera la variable membre booléenne `wireFrame`. Même chose pour afficher la boîte englobante `bbox` quand l'utilisateur presse la touche `'b'`. On utilisera la variable membre booléenne `drawBox` et la fonction `drawUnlit()` de la classe `AABBox`.

2.2 Fabrication de la grille

On va maintenant écrire le code de la fonction `computeLOD(float)`. On veut plonger le modèle 3D dans une grille régulière dont chaque cellule est un cube de coté `size`. Le paramète `percentage` passée à `computeLOD(float)` spécifie la valeur de cette variable comme un pourcentage (indiqué sur la ligne de commande) de la plus grande dimension de la boîte englobante :

```
size = percentage*bbox.sizeMax();
```

À faire (a) : Calculer le nombre de cellules `nbI`,`nbJ` et `nbK` que doit avoir la grille le long des 3 axes. Utilisez `bbox.min()` pour obtenir le coin inférieur de la boîte englobante, et `bbox.size(i)` pour obtenir ses dimensions le long des axes $i = 0, 1, 2$.

La classe `Viewer` définit le type `cell` par :

```
typedef deque<Vec> Cell;
```

À faire (b) : Allouer le tableau `cells` de taille `nbI*nbJ*nbK` de `Cell`. Ce tableau représente la grille. Complétez la fonction `int idCellContaining(Vec)` qui prend un point 3D et retourne l'index de la cellule du tableau `cells` contenant ce point. Écrire le code pour afficher la grille quand l'utilisateur presse la touche `'g'`. On utilisera la variable membre booléenne `drawGrid`.

2.3 Clustering

Il s'agit maintenant de "rattacher" chaque sommet du modèle à la cellule de la grille le contenant.

À faire (a) : Écrire le code qui parcourt les sommets du modèles et les rajoute dans la cellule correspondant.

À faire (b) : Parcourir chaque cellule et, si elle est non vide, calculer un représentant pour la cellule. On prendra pour l'instant le premier point de la cellule (voir questions 2.4 (c)). On les stockera `lodVertices` dans un `deque<Vec>` qui aura donc la même taille que `cells`.

2.4 Simplification

On va maintenant construire un nouvel ensemble de faces où chaque sommet appartient à la liste `lodVertices`. On utilisera une représentation par IFS avec un `lodTriangles` qui référencera le tableau `lodVertices`.

À faire (a) : Écrire le code qui parcourt les faces initiales du modèle et construit le tableau `lodTriangles`. On fera attention à supprimer les triangles dégénérés en un point ou une ligne. On fera également attention de ne pas générer plusieurs fois un même triangle. On utilisera une `map<Triplet,bool>` pour savoir si un triangle existe déjà, où `Triplet` est une classe fournie qui représente un triplet d'entier. Affichez le nombre de triangles obtenus. On n 'oubliera pas enfin de construire aussi un tableau `lodTriangleNormals`.

À faire (b) : Écrire le code pour afficher la version simplifiée du modèle quand l'utilisateur presse la touche `'l'`.

À faire (c) : Expérimenter d'autre stratégies pour le calcul du représentant (un point parmi ceux dans la cellule, le barycentre des points pondérés par l'aire des faces). Utilisez un paramètre de ligne de commande pour que l'utilisateur puisse choisir entre les différentes stratégies.
Dans leur papier original [RB93], Rossignac et Borrel "notent" les sommets en fonction de la courbure locale, de la taille des arrêtes adjacentes et de la probabilité d'appartenir à une silhouette. Le représentant d'une cellule est alors le sommet avec le plus haut score. On implémentera pas cette stratégie qui nécessite de manipuler les adjacences entre faces, arrêtes et sommets.

3 Avancées

Si on teste notre algorithme par exemple sur la vache, on constate que les pattes "disparaissent". À quoi cela est il dû à votre avis ?

3.1 Ajout de lignes

On décide de gérer explicitement les lignes dans la version simplifiée au lieu de les ignorer. Une ligne doit être fabriquée quand un triangle à exactement deux de ses sommets dans la même cellule. Modifier votre programme pour fabriquer ces lignes et les afficher. On stockera pour chaque ligne une épaisseur qui pourra être passée à OpenGL par la fonction :

```
glLineWidth(un nombre réel);
```

avant un `glBegin(GL_LINES)`.
Pour ce travail, on fera attention de ne pas générer de ligne entre deux points qui appartiennent déjà à un triangle simplifié.

3.2 Gestion des normales

On cherche à faire une meilleure gestion des normales pour le modèle simplifié. Pour cela, on commence par calculer une normale pour chaque sommet du modèle original en moyennant les normales des faces adjacentes. Puis on calculera une normale pour chaque représentant en moyennant les normales des sommets qu'il représente. Enfin. on fabriquera une normale pour chaque triangle généré en moyennant les normales de ses sommets.
Peut-on faire quelque chose de similaire pour les lignes ?

4 Pour aller plus loin

Une des faiblesses de l'algorithme proposé est sa sensibilité au placement de la grille. Pour ceux que ça intéresse, Tan et Low [TL97] proposent un algorithme qui s'affranchit de ce problème.

Références

[RB93] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. In *Geometric Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, 1993.

[TL97] Tiow-Seng Tan and Kok-Lim Low. Model simplification using vertex-clustering. In *Proceedings of the 1997 Symposium on Interactive 3D graphics*, pages 75–81. ACM Press, 1997.