Modélisation et Synthèse d'Image
TP3 : VBO, Maillages et indexation

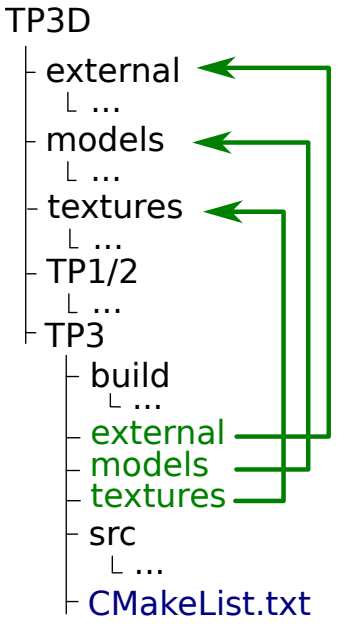
Introduction

L'objectif du TP est de s'initier à la génération et à l'importation de modèle 3D.

La base de code qui vous est fournie reprend les éléments de la dernière séance : création de fenêtre GLFW et initialisation de contexte OpenGL, affichage de géométrie, et utilisation des matrices de transformation pour manipuler la scène. Pour la récupérer, allez à l'adresse suivante : http://romain.vergne.free.fr/blog/?page_id=228.

Pour l'utiliser, faites comme la dernière fois :

- extrayez l'archive du TP dans le dossier de votre répertoire personnel dédié aux TP de 3D (`unzip TP3.zip -d ~/TP3D/ && rm TP3.zip`)
- accédez au dossier du TP (`cd ~/TP3D/TP3/`)
- créez un lien symbolique vers le dossier `external` (`ln -rs ../external/`)
- créez un lien symbolique vers le dossier `models` (`ln -rs ../models/`)
- créez un dossier pour la compilation (`mkdir build`)
- accédez à ce dossier (`cd build`)
- lancez `cmake` (`cmake ..`)
- lancez la compilation (`make`)
- exécutez (`./polytech_ricm4_tp3`)



Voici la structure que devrait avoir votre arborescence de fichiers.

Buffers

Pour rappel, un buffer ("tampon" en français) est une zone en mémoire sur la carte graphique qui contient des données utile pour l'affichage de l'image. Pour l'instant, nous avons seulement utilisé un buffer pour stocker les positions des sommets de nos objets. Nous allons maintenant voir comment créer des buffers pour gérer d'autres types de données : couleurs, normales, indices.

Comme pour le tableau de position, nous allons utiliser les `std::vector` pour représenter nos données sur le CPU. Cette structure a l'avantage de stocker l'information de manière contigue en mémoire. Pour ajouter un élément à un `std::vector`, utilisez la fonction `push_back()`. Pour connaître le nombre d'éléments déjà stockés, utilisez `size()`.

Créez un tableau de stockant une couleur par sommet du cube, que vous nommerez `colors`. Il doit y avoir autant de couleurs que de sommets. Choisissez les couleurs que vous voulez pour chacun des sommets de chacune des faces.

Ensuite, il faut créer un buffer à partir de `colors`. Pour ceci, il suffit de faire exactement pareil que pour les positions des sommets :

```
GLuint colorBufferID;
glGenBuffers(1, &colorBufferID);

// Definition de vertexBufferID comme le buffer courant
glBindBuffer(GL_ARRAY_BUFFER, colorBufferID);

// Copie des donnees sur la carte graphique (dans vertexBufferID)
glBufferData(GL_ARRAY_BUFFER, colors.size() * sizeof(vec3), colors.data(), GL_STATIC_DRAW);

// Obtention de l'ID de l'attribut "in_position" dans programID
GLuint vertexColorID = glGetAttribLocation(programID, "in_color");

// On autorise et indique a OpenGL comment lire les donnees
glVertexAttribPointer(vertexColorID,3,GL_FLOAT,GL_FALSE,0,(void*)0);
glEnableVertexAttribArray(vertexColorID);
```

Notez que ce VBO est automatiquement associé au VAO actif (vaoID). Il faut modifier le vertex shader pour que celui-ci réceptionne ce nouvel attribut :

```
#version 150

// Donnees d'entree
in vec3 in_position;
in vec3 in_color;

// Donnees de sortie
out vec4 my_color;

// Parametres
uniform mat4 ModelMatrix;
uniform mat4 ViewMatrix;
uniform mat4 ProjectionMatrix;

// Fonction appellee pour chaque sommet
void main() {
  // Affectation de la position du sommet
  // gl_Position est definit par defaut dan GLSL
  gl_Position = ProjectionMatrix * ViewMatrix * ModelMatrix * vec4(in_position, 1.0);

  my_color = vec4(in_color,1.0);
}
```

Ainsi que le fragment shader, pour que celui-ci affecte la valeur de couleur des sommets au fragment :

```
#version 150

in vec4 my_color;

out vec4 frag_color;

// Fonction appellee pour chaque fragment
void main() {
  // Affectation de la couleur du fragment
  frag_color = vec4(my_color.rgb ,1.0);
}
```

N'oubliez pas de détruire le buffer à la fin du programme :

```
glDeleteBuffers(1, &colorBufferID);
```

Buffer d'indices

Les sommets que nous avons utilisés pour créer le cube sont redondants. Logiquement, un cube devrait contenir 8 sommets. Or, ils sont tous stockés 3/4 fois. Pourquoi ?

Pour éviter cela, nous allons :

- modifier les tableaux de sommets et de couleurs pour contenir seulement 8 vecteurs chacun. Ces tableau représenteront donc seulement les attributs des sommets du maillage.
- créer un tableau d'indice qui contiendra la topologie du maillage (les indices reliant les sommets entre eux pour former des triangles).

Par exemple, le tableau d'indice suivant crée 1 triangle avec les sommets d'indices 0, 1 et 2 :

```
vector<uint> indices;
indices.push_back(0);
indices.push_back(1);
indices.push_back(2);
```

Pour l'envoie des données au GPU, c'est pratiquement pareil qu'avant, excepté qu'il faut préciser qu'il s'agit la d'un tableau d'indices (avec le mot clé `GL_ELEMENT_ARRAY_BUFFER`) et qu'on n'a pas besoins de faire le lien avec les shaders (puisqu'il ne s'agit pas d'attributs de sommets mais de topologie) :

```
GLuint indiceBufferID;
glGenBuffers(1, &indiceBufferID);

// Definition de vertexBufferID comme le buffer courant
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indiceBufferID);

// Copie des donnees sur la carte graphique (dans vertexBufferID)
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(uint),
             indices.data(), GL_STATIC_DRAW);
```

N'oubliez pas non plus de détruire ce tableau à la fin. Enfin, il faut aussi changer la manière de dessiner dans la boucle de rendu, avec `glDrawElements(GL_TRIANGLES,indices.size(),GL_UNSIGNED_INT,0);`

Faites toutes ces modifications nécessaires pour obtenir un rendu à base de tableaux indicés. Comment peut-on obtenir une seule couleur par face dans ce cas ?

Maillage

Nous allons maintenant nous intéresser à un mode de représentation des objets en 3D : les maillages. Un maillage est une surface discrétisée sous la forme de polygones. Nous nous en servons en informatique graphique pour représenter les objets.

Un maillage est constitué :

- d'un ensemble de points, ils représentent les sommets des polygones formant notre maillage
- d'un ensemble d'indices, ils représentent l'appariement des sommets pour former les triangles

Pour stocker un maillage dans un fichier, il existe différents formats. Le plus simple d'entre eux est le format OFF. Pour plus de détail, voir figure 1.

Les fichiers "Mesh.h" et "Mesh.cpp" implémentent une classe permettant de représenter un maillage : cette classe contient des attributs pour les tableaux de positions, de normales, de couleurs, et d'indices. Par ailleurs, cette classe propose de charger automatiquement ces informations à partir d'un fichier OFF.

Pour l'utiliser, faites comme ceci :

```
OFF

#vertex_count face_count  edge_count
3     1     0

#One line for each vertex
0.0   0.5   0.0
0.5   -0.5  0.0
-0.5  -0.5  0.0

#One line for each polygonal face
#vertex_number vertex_indices
3 0 1 2
```

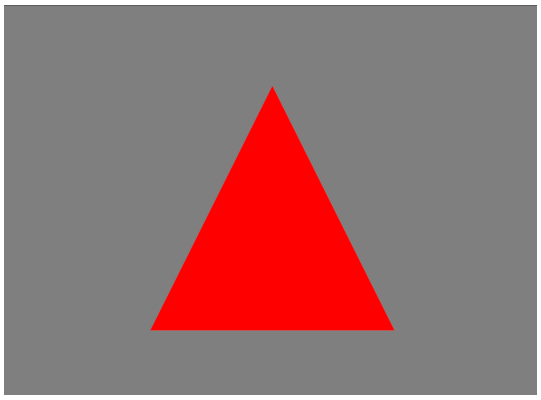


Figure 1: Exemple de fichier OFF pour un triangle

```
Mesh m("../models/armadillo.off");   // chargement du maillage
m.vertices.data();                   // acces au tableau de positions
m.normals.data();                    // acces au tableau de normales
m.faces.data();                      // acces au tableau d'indices
```

Travail à faire : chargez un maillage du répertoire "models/" et affichez le dans la fenêtre.

Organisation

Rendre une archive *nom1_nom2.zip* contenant :

- Le code commenté prêt à être compilé/executé et générant les sorties demandées.
- Un rapport contenant vos résultats (réponses, images, commentaires, ...).

À envoyer par mail à `romain.vergne@inria.fr` avec "[RICM4-TP3] Nom1 Nom2" en objet.