

# TP3 bis : Maillages et indexation

## Introduction

Ce TP poursuit le TP précédents. La base de code et le sujet ont été mis à jour pour intégrer la création d'un buffer de couleurs, et reprend à partir de l'indexing.

## Buffers

Cette section rappelle l'utilisation des tampons ou buffers.

Un tampon est une zone mémoire dans laquelle vous allez pouvoir stocker les données nécessaires à l'affichage de votre objet 3D. Ces données sont d'abord générées par le CPU, et représentées dans la mémoire de celui-ci (la mémoire vive). Par la suite, elles seront copiées dans la mémoire du GPU (cache) dans des tampons afin d'être utilisées directement par les shaders. Enfin, les tampons seront détruits, libérant l'espace qu'ils occupaient en mémoire cache.

### Creation.

Pour créer les données il faut :

- Déclarer un conteneur de donnée :  
`std::vector<T> conteneur;`  
Ici, T représente le type de vos données, `glm::vec3` dans le cas de positions 3D, ou `uint` dans le cas d'indices par exemple.
- Remplir le conteneur :  
`conteneur.push_back(glm::vec3(0.0f));`  
`conteneur.push_back(glm::vec3(1.0f, 0.0f, 0.0f));`  
`conteneur.push_back(glm::vec3(0.0f, 1.0f, 0.0f));`  
par exemple.

Pour créer un tampon et y recopier les données précédentes il faut :

- Déclarer un identifiant pour le tampon :  
`GLuint bufferID;`  
C'est grâce à cet identifiant qu'on va référencer le tampon.
- Générer le tampon :  
`glGenBuffers(1, &bufferID);`  
Le tampon existe maintenant sur le GPU.
- Facultativement, afficher l'identifiant :  
`std::cout << "bufferID = " << bufferID << std::endl;`  
La valeur devrait normalement être un nombre entre 1 et le nombre de tampons que vous avez créé jusqu'à maintenant (typiquement moins d'une dizaine dans le cadre de ces TP).
- Définir celui-ci comme le tampon courant :  
`glBindBuffer(GL_ARRAY_BUFFER, bufferID);`  
Si votre tampon contient des indices (pour l'indexing), il faut remplacer `GL_ARRAY_BUFFER` par `GL_ELEMENT_ARRAY_BUFFER`.
- Le remplir de vos données :  
`glBufferData(GL_ARRAY_BUFFER, conteneur.size() * sizeof(T), conteneur.data(), GL_STATIC_DRAW);`  
Encore une fois, T est à remplacer par le type de vos données, et `GL_ARRAY_BUFFER` par `GL_ELEMENT_ARRAY_BUFFER` si votre buffer contient des indices.

## Utilisation.

Les données situées dans les tampons vont être utilisées directement dans les shaders. Pour ceci, il faut préciser le lien entre chaque variable d'entrée du shader et le tampon correspondant. La seule exception est le tampon d'indices, qui est utilisé directement par OpenGL pour ordonner les autres variables.

La variable d'entrée du shader doit être d'un type compatible avec celle stockée dans le tampon : par exemple un `glm::vec3` en C++ est compatible avec le type `vec3` natif de GLSL.

Afin de spécifier le lien entre une variable d'entrée d'un shader et un tampon, il faut :

- Récupérer l'identifiant de la variable :  

```
GLuint varID = glGetAttribLocation(programID, "in_variable");
```

 Ici, on suppose que le shader a été compilé et que le programme résultant est identifié par `programID`.
- Activer la variable d'entrée du shader :  

```
glEnableVertexAttribArray(varID);
```
- Définir le tampon qu'on souhaite lier à cette variable comme le tampon courant :  

```
glBindBuffer(GL_ARRAY_BUFFER, bufferID);
```
- Préciser à OpenGL la manière dont le tampon doit être lu, et dans quelle variable mettre le résultat :  

```
glVertexAttribPointer(varID, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
```

 Ici, `varID` indique la variable d'entrée avec laquelle lier le tampon courant, 3 est le nombre de composants de la variable, `GL_FLOAT` est le type des composantes. Les autres paramètres ne nous intéressent pas pour l'instant.

Toutes ces instructions sont à mettre à l'initialisation du programme, avant la boucle de rendu.

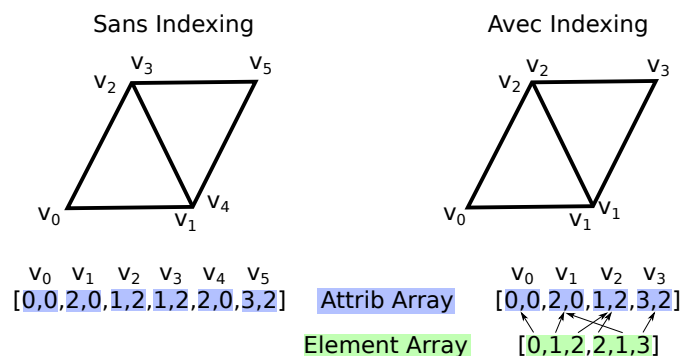
## Destruction.

Une fois que vous avez fini de vous en servir, c'est à dire une fois sortis de la boucle de rendu, il faut détruire tout les buffers que vous avez créé. Pour ceci, il suffit de faire :

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

## Indexing

L'indexing est la méthode consistant à référencer les attributs d'un sommet par leurs indices plutôt que de les re-donner à OpenGL lorsque celui-ci en a besoin pour le rendu. Dans l'exemple suivant, l'indexing est utilisé pour diminuer le nombre de positions nécessaire à la définition d'un parallélogramme :



Reproduisez l'exemple ci-dessus en utilisant l'indexing.

## Travail à faire

Dans le programme qui vous est donné, un carré est dessiné sans indexing. Modifiez ce programme pour le dessiner ce même carré en utilisant l'indexing. Pour ceci, vous devez :

- Modifier les données des tampons de positions et de couleurs pour éliminer les redondances.
- Créer un tampon d'indices.
- Remplir celui-ci des indices adéquats.
- Le définir comme le buffer courant (pour les buffers d'indices) :  
`glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indicesBufferID);`  
Cette commande est à utiliser avant de dessiner.
- Appeler la fonction de dessin par indexing :  
`glDrawElements(GL_TRIANGLES, nb_indices, GL_UNSIGNED_INT, (void*)0);`  
Cette fonction remplace l'appel à `glVertexAttribPointer` pour les attributs associés à des variables d'entrée du shader.
- Détruire le buffer d'indices :  
`glDeleteBuffers(1, &indicesBufferID);`  
Cette commande est à utiliser à la fin du programme.

## 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 la donnée de :

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

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, passez le chemin du fichier off au constructeur : `Mesh m("../models/armadillo.off");` Vous pouvez ensuite accéder aux données du maillage avec `m.vertices`, `m.normals`, `m.faces`.

## Travail à faire

Chargez un maillage du répertoire "models/" et affichez le dans la fenêtre.

## Bonus

- Créez un cube par indexing en utilisant huit positions et huit couleurs.
- Essayez de colorer chaque face uniformément. Qu'observez vous ? Proposez une solution.
- Faites une fonction qui génère un maillage représentant un cylindre, en vous servant de la définition mathématique de cet objet.

OFF

```
#vertex_count face_count edge_count  
3 1 0
```

```
#One line for each vertex
```

```
0.0 0.5 0.0
```

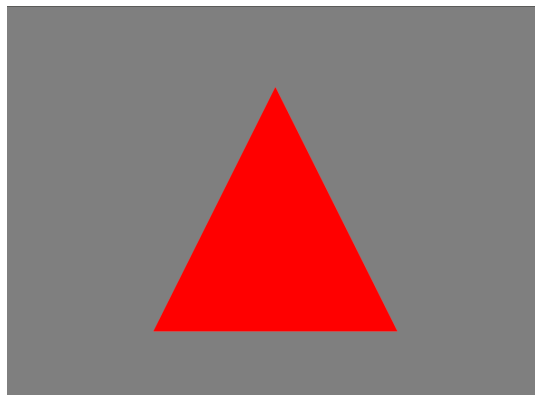
```
0.5 -0.5 0.0
```

```
-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