Rendu d'un objet simple

malek.bengougam@gmail.com

Partie1. Rendu à partir de données statiques

1.1 Chargement des données

Le fichier DragonData.h défini deux tableaux, un tableau de float-s contenant les attributs de chaque sommet (DragonVertices), et un tableau d'unsigned short (entier court sur 16 bits /2 octets) contenant les indices des triangles (DragonIndices).

Les vertices sont définis comme ceci :

```
// Vertex du Dragon
struct Vertex {
    float position[3];   // x, y, z
    float normal[3];     // nx, ny, ny
    float uv[2];         // u, v
};
```

Note : on utilisera les normales dans un prochain TP concernant l'illumination. A ce stade vous pouvez les interpréter temporairement comme par ex. des couleurs si vous souhaitez les utilisez.

En C et C++, lorsqu'un tableau est défini de manière statique (en dur dans le code) il est possible d'utiliser `sizeof()` afin de connaitre la taille totale en octet.
Si l'on souhaite plutôt connaitre le nombre d'éléments, on utilise de préférence la macro `_countof()` qui est définie sur la majeure partie des compilateurs.

On va utiliser `_countof(DragonIndices)` pour récupérer le nombre total d'indices.
Dans les cas de DragonVertices, comme on souhaite en pratique connaitre le nombre de sommets (et pas le nombre de float) il faut diviser la valeur du `sizeof(DragonVertices)` par `sizeof(Vertex)`.

Pour dessiner notre objet, on doit remplir le VBO avec les données de DragonVertices, mais également créer un IBO pour stocker les données de DragonIndices.
A titre de rappel, un VBO est de type **GL_ARRAY_BUFFER**, tandis qu'un IBO se caractérise par **GL_ELEMENT_ARRAY_BUFFER**.
Pensez à adapter les attributs en tenant compte des propriétés des sommets du Dragon.

Le rendu s'effectue comme précédemment en utilisant le VAO mais cette fois-ci nous allons utiliser la fonction **`glDrawElements()`** à la place de **`glDrawArrays()`** afin de prendre en compte les indices (**ELEMENT**).

Pensez également à créer une texture afin de pouvoir plaquer l'image « dragon.png » sur l'objet.

Note : comme l'origine des textures en OpenGL est en bas à gauche, il est souvent nécessaire d'inverser le sens de la texture. Le procédé le plus simple consiste à inverser la coordonnée t (ou y) des UV dans le vertex shader :

```
v_texcoords = vec2(a_texcoords.s, 1.0 – texcoords.t) ;
```

1.2 Passage à la 3D

Le passage à la 3D entraîne plusieurs changements importants :

Tout d'abord, on remarque que les faces ne sont pas triées correctement. Pour utiliser le tri automatique des faces/fragments, il faut activer l'implémentation matérielle du z-buffer.
On va utiliser les fonctions suivantes :

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS); // optionnelle ici, car valeur par défaut
```

L'algorithme va tester la profondeur des fragments avec la valeur déjà inscrite dans le Depth Buffer.
La fonction de comparaison, **`glDepthFunc()`**, va servir à classifier le tri. Avec GL_LESS on spécifie que l'on ne souhaite conserver que les valeurs strictement inférieures aux valeurs courantes.

Cependant, il reste un problème. Le contenu du depth buffer est persistant d'une image à une autre, ce qui provoque un bug : notre objet disparait.
La solution est très simple : il suffit d'effacer le Depth Buffer comme on efface l'écran (Color Buffer).
Pour ce faire, on va ajouter **`GL_DEPTH_BUFFER_BIT`** à **`glClear()`**, valeur que l'on va combiner à l'aide d'un 'pipe' (la barre verticale ' |' représentant le OU binaire).

Comme dernière optimisation, on peut activer le « face culling » (**GL_CULL_FACE**) afin de supprimer les faces cachées et gagner un peu en performance.

Partie2. Rendu d'un objet au format OBJ

Le format OBJ est un format classique, mais vieillissant -pour ne pas dire désuet- de l'infographie.
Il a pour avantage d'être un format textuel facile à traiter mais reste limitatif pour des applications exigentes (il existe des mises à jour « moderne » du format), mais quoiqu'il en soit d'autres formats sont préférable que cela soit pour une question d'échange de fichiers ou de praticité du fichier.

On pourrait « parser » le fichier OBJ à la main, mais on va plutôt utiliser une bibliothèque pour nous faciliter le travail. TinyObjLoader (https://github.com/tinyobjloader/tinyobjloader) est une bibliothèque dans le même esprit que STB, c'est-à-dire en entête seule.
TinyObjLoader va nous permettre de récupérer les données sous forme de plusieurs tableaux qu'il nous faut interpréter et re-assembler pour former un tableau d'attributs (et/ou d'indices) compatible avec OpenGL.

Note : les fichiers OBJ sont généralement accompagné d'un fichier MTL qui décrit les matériaux de(s) objet(s). Nous n'avons pas d'utilité pour les matériaux dans ce TP, mais ils s'avèreront utiles lors d'un prochain cours sur l'illumination.

La particularité du format OBJ réside dans sa définition des faces.
Le format gère 3 tableaux, 1 tableau pour les positions ('v'), 1 tableau pour les normales ('vn') et un tableau pour les coordonnées de texture ('vt').
Une face est composée pour chaque sommet d'un triplet (*iv, ivn, ivt*) où chaque composante est l'indice d'attribut dans le tableau correspondant.

Ces trois tableaux peuvent être de taille différente car le format optimise les redondances – par exemple si une même normale est utilisée par plusieurs sommets, elle ne sera pas répétée.

Attention ! Le format OBJ supporte les maillages non triangulaires, mais pas OpenGL. Il est fondamental de s'assurer que l'OBJ a bien été exporté avec une triangularisation du maillage !

Cependant, ce fonctionnement en tableaux de tailles différentes n'est pas compatible avec OpenGL.
On a alors deux solutions.

La première consiste à lire les faces une à une, puis lire les indices de chaque attributs (*v, vt, vn*) et les ajouter dans une structure Vertex que l'on va insérer dans un tableau de Vertex. Cette première solution est relativement simple à mettre en œuvre, mais implique beaucoup de redondance. Comme on n'utilise plus de tableau d'indice au final on peut se contenter d'effectuer le rendu avec `glDrawArrays()`.

La seconde, plus poussée, consiste à n'insérer un Vertex dans un tableau que s'il n'est pas déjà présent.
Pour cela, on compare les 3 attributs, et s'ils sont similaires c'est que le vertex a déjà été inséré.
On utilise un second tableau, les indices, que l'on rempli au fur à mesure en fonction.

```
Indices = vide
Vertices = vide
Pour chaque Faces chacun du fichier OBJ
  Pour chaque sommet j des Faces
    Créer le Vertex Vj = { position, normale, uv }
    Rechercher le Vertex Vj dans le tableau Vertices
    Si Vj n'existe pas, Index = size ; insérer Vj dans le tableau Vertices
    Sinon, v existe déjà dans le tableau à l'index, Index = i
    Ajouter Index au tableau « indices »
```