Apprendre OpenGL moderne

Troisième partie : chargement de modèle 3D

Par Joey de Vries - Jean-Michel Fray (traducteur)

Date de publication : 18 juin 2018

Developpez.com a la joie d'accueillir une traduction en français du célèbre cours de grande qualité **Learn OpenGL**. Au cours de celui-ci, vous apprendrez à programmer des applications graphiques 3D grâce à la bibliothèque OpenGL.

Ces tutoriels sont accessibles aux débutants, mais les connaisseurs ne s'ennuieront pas non plus grâce aux chapitres avancés.

Cette page vous amène à la troisième partie du tutoriel, c'est-à-dire : le chargement de modèles 3D avec Assimp.

Vous pouvez retrouver les autres parties ci-dessous :

- **introduction** ;
- **éclairage** ;
- **chargement de modèle** ;
- **OpenGL avancé** ;
- éclairage avancé ;
- PBR ;
- **mise en pratique**.

Commentez

I - Assimp

Dans les scènes vues jusqu'ici, nous avons utilisé notre petit conteneur de nombreuses façons. Mais avec le temps, nos meilleurs amis peuvent devenir un peu ennuyeux. Dans les applications graphiques, on trouvera nombre de modèles complexes et intéressants, bien plus agréables à regarder qu'un simple conteneur. Cependant, contrairement à un conteneur, on ne peut pas définir manuellement tous les sommets, les normales et les coordonnées de textures de formes complexes comme des maisons, des véhicules ou des formes humaines. En réalité, on *importera* ces modèles dans l'application, lesquels seront conçus par des artistes 3D avec des outils comme **Blender**, **3DS Max** ou **Maya**.

Ces outils, appelés « logiciels de modélisation 3D » permettent la création de formes complexes et l'application de textures, procédé nommé *uv-mapping*. Ces logiciels génèrent automatiquement toutes les coordonnées des sommets, les normales et les coordonnées de textures lors de leur exportation dans un fichier créé selon un format spécifié. De cette façon, les modeleurs disposent de nombreux outils pour créer des modèles de grande qualité, sans avoir trop à se soucier des détails techniques, qui sont cachés dans le fichier exporté du modèle. Nous, en qualité de développeurs, **nous devons** nous soucier de ces aspects techniques.

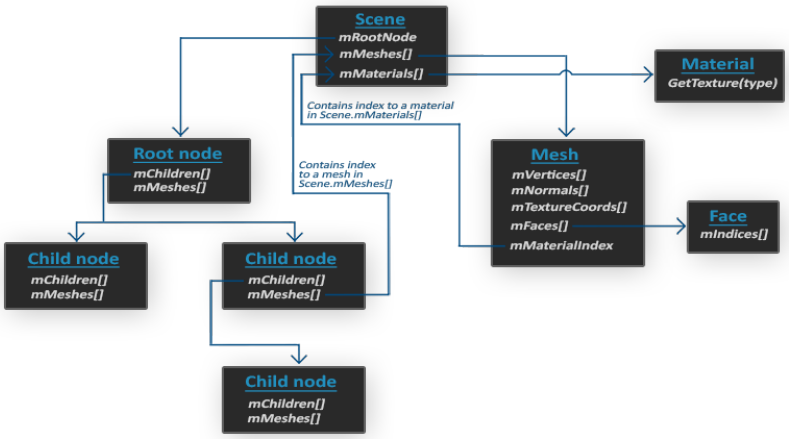
Notre travail consistera à analyser ces modèles exportés pour en extraire les informations utiles, de façon à les présenter sous une forme compréhensible par OpenGL. Un problème courant est lié à l'existence de dizaines de formats différents pour exporter les données d'un modèle. Des formats de modèles comme le **.obj de Wavefront** ne contiennent que les données, avec seulement quelques informations comme les couleurs et les textures (diffuses ou spéculaires), tandis que d'autres formats comme le format **Collada** sont très détaillés, contenant les modèles, les éclairages, beaucoup de types de matériaux, des données d'animation, des caméras, des informations sur la scène entière et bien plus encore. Le format .obj est considéré comme un format de modèle facile à analyser. On vous recommande de visiter la page wiki de Wavefront, au moins pour voir comment un tel format de fichier est structuré. Cela vous donnera un aperçu sur le sujet.

En tout état de cause, il existe tant de formats de fichiers différents que trouver une structure commune et générale se révèle impossible. Ainsi, si nous voulions importer un modèle à partir de ces formats, nous aurions à développer un module d'importation pour chacun des formats envisagés. Heureusement, une bibliothèque existe pour réaliser ce travail.

I-A - Une bibliothèque de chargement de modèle

Une bibliothèque très populaire pour importer des modèles s'appelle **Assimp**, pour **Open Asset Import Library**. Assimp peut importer des dizaines de formats de fichiers modèles (et peut aussi les exporter), en archivant les données du modèle dans des structures de données générales. Dès qu'Assimp charge le modèle, on peut retrouver toutes les données du modèle dans des structures de données propres à Assimp. Ces structures de données restant toujours les mêmes, quel que soit le format du fichier importé, cela nous permet de s'affranchir des formats utilisés pour les fichiers importés.

Assimp charge le modèle entier dans un objet **Scene** qui contient toutes les données du modèle importé. Assimp contient alors un ensemble de nœuds, chaque nœud contenant des indices pointant vers les données de l'objet Scene, un nœud pouvant avoir un nombre quelconque de nœuds fils. Un modèle (simplifié) de la structure des données d'Assimp est figuré ci-dessous :



- Toutes les données du modèle, comme les matériaux et les mailles, sont contenues dans l'objet **Scene**. Cet objet détient aussi une référence vers le nœud **Root** de la scène.
- Le nœud **Root** de la scène peut comprendre des nœuds fils (comme tout autre nœud) et peut contenir un ensemble d'indices qui pointent vers les données de maillage dans le tableau mMeshes de l'objet Scene. Le tableau mMeshes du nœud Root contient les objets Mesh réels, alors que les valeurs du tableau mMeshes d'un nœud ne sont que des indices pour le tableau des mailles de la scène.
- Un objet **Mesh** contient toutes les données nécessaires pour le rendu : positions des sommets, normales, coordonnées de textures, faces, ainsi que les matériaux de l'objet.
- Une maille contient plusieurs faces. Une **face** représente une primitive de rendu de l'objet (triangles, carrés, points). Une face contient les indices des sommets qui forment une primitive. Les sommets et les indices étant séparés, cela nous facilite le rendu si l'on utilise un tampon d'indices (voir le chapitre **Hello Triangle**).
- Et enfin, une maille contient aussi un objet **Material** qui contient plusieurs fonctions pour retrouver les propriétés de matériau d'un objet. On pense aux couleurs, aux textures (comme les textures diffuses ou spéculaires).

Pour commencer, nous voulons charger un modèle dans un objet Scene, retrouver récursivement les objets Mesh correspondants à partir de chacun des nœuds (nous chercherons récursivement chaque fils de chaque nœud) puis traiter chaque objet Mesh pour récupérer les données des sommets, ses indices et ses propriétés de matériau. Le résultat est ainsi un ensemble de données de maillage qui seront contenues dans un seul objet Model.

Maille (Mesh)
Pour modéliser des objets, les créateurs ne conçoivent pas une forme unique pour tout le modèle. Généralement, chaque modèle est constitué de plusieurs sous-ensembles. Chacune de ces composantes est appelée une maille. Pensez par exemple à un modèle d'humain : l'artiste modélisera la tête, les membres, les vêtements, les armes, tout cela séparément, et la combinaison de ces mailles formera le modèle complet. Une maille est la représentation minimale de ce qui est nécessaire pour afficher un objet avec OpenGL (sommets, indices, matériaux). Un modèle contient généralement plusieurs mailles.

Dans le chapitre suivant, nous créerons nos propres classes Model et Mesh pour charger et mémoriser les modèles importés en utilisant la structure que nous venons de décrire. Si nous voulons ensuite afficher un modèle, nous afficherons ce modèle maille par maille. Cependant, avant de commencer à importer des modèles, nous devons intégrer Assimp à nos projets.

I-B - Installer Assimp

Vous pouvez télécharger Assimp à partir de cette **page** et choisir la version qui vous convient. Lors de l'écriture de ces pages, la dernière version était 3.1.1. Nous vous conseillons de compiler ces bibliothèques vous-mêmes, car,

en général, les versions précompilées ne fonctionnent pas. Consultez le **troisième chapitre** si vous avez oublié comment compiler une bibliothèque avec CMake.

Quelques questions peuvent se poser lors de l'installation d'Assimp, je les mentionne ci-dessous avec la solution si vous les rencontrez :

- CMake renvoie continuellement des erreurs à propos de bibliothèques DirectX manquantes, des messages comme celui-ci :
- La solution consiste à installer le SDK de DirectX si ce n'est pas déjà fait, que vous pouvez télécharger **ici**.
- Lors de l'installation du SDK de DirectX, le code d'erreur s1023 peut apparaître. Dans ce cas, il faudra désinstaller le(s) paquet(s) C++ Redistributable avant d'installer le SDK, comme indiqué **ici**.
- Lorsque la configuration est complète, vous pouvez générer un fichier solution, l'ouvrir et compiler les bibliothèques (soit une version release, soit une version debug, comme vous voudrez).
- La configuration par défaut installe Assimp comme une bibliothèque dynamique, ainsi il faudra ajouter le fichier assimp.dll à côté des binaires de votre application. Il vous suffira de copier cette DLL dans le même répertoire que votre exécutable.
- Après la compilation d'Assimp, les bibliothèques construites et le fichier DLL se trouveront dans le répertoire *code/Debug* ou *code/Release*.
- Vous pouvez simplement copier ces fichiers dans le répertoire de votre choix, et les intégrer à votre solution. N'oubliez pas d'inclure les fichiers d'en-tête, que vous trouverez dans le répertoire *include* du module Assimp.

En cas de souci, n'hésitez pas à poser une question sur le forum.

*Si vous souhaitez qu'Assimp fonctionne en multithreading pour améliorer les performances, vous pouvez le compiler avec Boost, dont vous trouverez les instructions pour l'installation sur cette **page**.*

Maintenant, vous devriez avoir compilé Assimp et l'avoir lié à votre application. Étape suivante : importer un truc sympa !

I-C - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

II - Maillage

En utilisant Assimp, on peut charger beaucoup de modèles différents, et une fois chargés, ils sont accessibles dans les structures de données d'Assimp. Nous souhaitons ensuite transformer ces données dans un format utilisable par OpenGL pour afficher ces objets. Nous avons vu dans le chapitre précédent qu'une maille représente une entité affichable, commençons donc par définir une classe pour représenter une maille.

Revenons sur ce que nous avons vu jusqu'ici et réfléchissons à la structure de données minimale pour une maille. Une maille doit comprendre au moins un ensemble de sommets, chaque sommet étant composé d'un vecteur de position, un vecteur normal, ainsi qu'un vecteur de coordonnées de texture. Une maille devrait aussi comprendre les indices pour un rendu indexé et des données de matériau sous la forme de textures spéculaires ou diffuses.

Maintenant que nous avons défini un contenu minimal pour la classe de mailles, nous pouvons définir un sommet dans OpenGL :

```
struct Vertex {
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
```

```
};
```

Nous mémorisons chacun des sommets dans une structure nommée Vertex, que nous pourrons utiliser pour indexer chacun de nos attributs de sommet. En plus de cette structure Vertex, nous utiliserons aussi une structure Texture pour les textures :

```
struct Texture {
    unsigned int id;
    std::string type;
};
```

Nous mémorisons l'identifiant de la texture et son type (diffuse ou spéculaire).

Nous pouvons maintenant commencer à définir une structure pour la classe Mesh :

```
class Mesh {
       public:
        /*  Données du modèle  */
        std::vector<Vertex> vertices;
        std::vector<unsigned int> indices;
        std::vector<Texture> textures;
        /*  Fonctions  */
        Mesh(std::vector<Vertex> vertices, std::vector<unsigned int>
 indices, std::vector<Texture> textures);
        void Draw(Shader shader);
    private:
        /*  Render data  */
        unsigned int VAO, VBO, EBO;
        /*  Functions    */
        void setupMesh();
};
```

Cette classe de mailles n'est pas très complexe. Nous placerons dans le constructeur les données utiles, nous initialiserons les tampons dans la fonction setupMesh() et finalement nous afficherons la maille avec la fonction Draw(). Noter que nous passons un shader à la fonction Draw() ; en passant le shader à la maille, on peut affecter différentes variables uniformes avant d'effectuer l'affichage (par exemple pour lier les échantillonneurs aux unités de textures).

Le corps du constructeur est assez évident. On initialise les variables publiques de la classe avec les arguments du constructeur. On appelle également la fonction setupMesh() dans le constructeur :

```
Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices, std::vector<Texture>
 textures)
{
    this->vertices = vertices;
    this->indices = indices;
    this->textures = textures;
    setupMesh();
}
```

Rien de particulier à signaler, détaillons la fonction setupMesh().

II-A - Initialisation

Grâce au constructeur, nous disposons d'une longue liste de données de maillage que nous pouvons utiliser pour l'affichage. Nous devons encore initialiser les tampons et préciser la disposition des données pour le vertex shader au moyen des pointeurs d'attributs de sommets. Cela ne devrait pas vous poser de problèmes, mais nous allons corser un peu les choses en introduisant les données des sommets dans une structure :

```
void setupMesh()
{
```

```
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0],
 GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &indices[0],
 GL_STATIC_DRAW);
    // position des sommets
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
    // normales
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
 Normal));
    // coordonnées de texture
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex,
 TexCoords));
    glBindVertexArray(0);
}
```

Le code n'est pas très différent de ce qu'on attendait, mais il contient quelques astuces liées à la structure Vertex.

Les structures en C++ ont une propriété intéressante : les données sont organisées séquentiellement. Si nous représentions les données d'une structure comme un tableau de données, il ne contiendrait que les variables de la structure en ordre séquentiel, ce qui se transpose directement en un tableau de réels comme nous le souhaitons pour les tampons. Par exemple, si nous avions une structure complète initialisée ainsi :

```
Vertex vertex;
vertex.Position  = glm::vec3(0.2f, 0.4f, 0.6f);
vertex.Normal    = glm::vec3(0.0f, 1.0f, 0.0f);
vertex.TexCoords = glm::vec2(1.0f, 0.0f);
```

le contenu mémoire serait celui-ci :

```
[0.2f, 0.4f, 0.6f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f];
```

Grâce à cette propriété, on peut passer directement l'adresse d'une grande liste de structures pour initialiser les données de tampon, et ces données seront transmises sans problème :

```
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), vertices[0], GL_STATIC_DRAW);
```

Naturellement, l'opérateur sizeof peut aussi être utilisé sur la structure pour obtenir la taille correcte en octets. Ici, cela donne 32 octets (8 réels de 4 octets chacun).

Une autre propriété des structures est la directive du préprocesseur appelée offsetof(s,m), qui prend en argument une structure et le nom d'un champ de cette structure. La macro retourne le décalage en octets de ce champ à compter du début de la structure. Cela est parfait pour définir le paramètre de décalage de la fonction glVertexAttribPointer() :

```
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
```

Le décalage est ainsi défini par la macro offsetof qui dans ce cas initialise le décalage du vecteur normal dans le tampon avec le décalage du vecteur normal dans la structure, qui vaut 3 réels, donc 12 octets. Noter que nous donnons au paramètre stride la taille de la structure Vertex.

Utiliser une structure comme celle-ci permet non seulement de produire un code plus lisible, mais nous permet aussi de modifier la structure. Si nous voulons ajouter un autre attribut, il suffit de l'ajouter à la structure sans avoir besoin de modifier le code.

II-B - Affichage

La dernière fonction à définir pour compléter la classe Mesh est la fonction Draw(). Mais pour afficher notre maille, il nous faut lier les textures avant d'appeler glDrawElements(). Cependant, cela n'est pas très facile, car nous ne savons pas combien de textures la maille utilise et de quel type elles sont. Comment initialiser les unités de textures et les échantillonneurs dans les shaders ?

Pour résoudre ce problème, nous allons choisir une convention pour les noms des textures : chaque texture diffuse sera nommée texture_diffuseN et chaque texture spéculaire texture_specularN, N étant un nombre variant de 1 jusqu'au nombre maximum de textures admises. Supposons que nous ayons trois textures diffuses et deux textures spéculaires, les échantillonneurs de textures s'appelleraient ainsi :

```
uniform sampler2D texture_diffuse1;
uniform sampler2D texture_diffuse2;
uniform sampler2D texture_diffuse3;
uniform sampler2D texture_specular1;
uniform sampler2D texture_specular2;
```

Avec cette convention, nous pouvons définir dans les shaders autant d'échantillonneurs de textures que nous voulons, et si une maille contient beaucoup de textures, nous saurons comment elles s'appellent. Le développeur pourra en utiliser autant qu'il le souhaite en définissant ses propres échantillonneurs (en définir moins conduirait à perdre un peu de temps avec les appels pour les lier).

Il existe beaucoup de solutions pour ce genre de problèmes et si vous n'aimez pas celle-ci, vous pouvez très bien en imaginer une autre.

Le code pour l'affichage devient donc :

```
void Draw(Shader shader)
{
    unsigned int diffuseNr = 1;
    unsigned int specularNr = 1;
    for(unsigned int i = 0; i < textures.size(); i++)
    {
        glActiveTexture(GL_TEXTURE0 + i); // Activation de l'unité de texture adéquate avant
 liaison
        // récupère le numéro de la texture (le N dans diffuse_textureN)
        std::string number;
        std::string name = textures[i].type;
        if(name == "texture_diffuse")
            number = std::to_string(diffuseNr++);
        else if(name == "texture_specular")
            number = std::to_string(specularNr++);
        shader.setFloat(("material." + name + number).c_str(), i);
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
    }
    glActiveTexture(GL_TEXTURE0);
    // affiche le mesh
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}
```

On commence par calculer la valeur de N par type de texture et on la concatène à la chaîne du type de texture pour obtenir le nom correct de la variable uniforme. On retrouve l'échantillonneur correspondant, on lui passe l'emplacement de l'unité de texture active et on lie cette texture.

On a aussi ajouté « **material** » au nom de la variable uniforme, car nous mémorisons en général les textures dans une structure **material** (cela peut changer selon les implémentations).

Notons que nous incrémentons les compteurs lors de leur conversion en chaîne de caractères. En C++, l'instruction i++ retourne la valeur de i, puis incrémente i, tandis que ++i incrémente d'abord i, puis retourne la valeur incrémentée. Dans notre cas, la valeur passée est la valeur avant incrémentation.

Vous trouverez le code source complet de la classe Mesh **ici**.

La classe Mesh que nous avons définie est une encapsulation de beaucoup de concepts discutés dans les premiers chapitres. Dans le prochain chapitre, nous créerons un modèle de conteneur pour plusieurs mailles et qui implémente complètement l'interface de chargement d'Assimp.

II-C - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

III - Le Modèle

Il est temps de se plonger dans le cœur d'Assimp et de commencer à créer le code de chargement d'un modèle. Le but de ce tutoriel est de créer une autre classe pour représenter un modèle dans son intégralité : un modèle ayant plusieurs mailles et éventuellement plusieurs objets. Une maison, qui comprend un balcon en bois, une tour et même une piscine pourra être ainsi chargée dans un seul modèle. Nous chargerons le modèle grâce à Assimp et le traduirons en plusieurs objets Mesh selon le schéma créé dans le tutoriel précédent.

Sans plus attendre, je vous présente la structure de la classe Model :

```
Class Model
{
    public:
        /*  Fonctions   */
        Model(char *path)
        {
            loadModel(path);
        }
        void Draw(Shader shader);
    private:
        /*  Données du modèle  */
        std::vector<Mesh> meshes;
        std::string directory;
        /*  Fonctions   */
        void loadModel(std::string path);
        void processNode(aiNode *node, const aiScene *scene);
        Mesh processMesh(aiMesh *mesh, const aiScene *scene);
        std::vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType
 type, std::string typeName);
};
```

La classe Model contient un vecteur d'objets Mesh et nous oblige à passer, à son constructeur, un chemin pointant sur le fichier à charger. On peut ainsi charger le fichier directement par la fonction loadModel(), appelée dans le constructeur. Les fonctions privées sont destinées à traiter une partie de la tâche d'importation d'Assimp et nous les examinerons brièvement. Nous mémorisons aussi le répertoire qui nous servira plus tard pour le chargement des textures.

La fonction Draw() n'a rien de particulier et ne fait qu'afficher chacune des mailles au moyen de leur fonction Draw() respective :

```
void Draw(Shader shader)
```

```
{
    for(unsigned int i = 0; i < meshes.size(); i++)
        meshes[i].Draw(shader);
}
```

III-A - Importation d'un modèle 3D dans OpenGL

Pour importer un modèle et le transposer dans nos propres structures, nous devons d'abord inclure les fichiers d'en-tête d'Assimp :

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
```

La première fonction que nous appelons est loadModel(), cela est effectué dans le constructeur. Avec cette fonction loadModel(), nous utilisons Assimp pour charger le modèle dans une structure de données qu'Assimp appelle l'objet *scene*. Souvenez-vous du chapitre 19 (Assimp), cet objet est à la racine de l'interface de données d'Assimp. À partir de cet objet *scene*, nous pouvons accéder à toutes les données du modèle importé.

L'atout majeur d'Assimp est de pouvoir s'abstraire de tous les détails techniques des différents formats de fichiers, et ceci grâce à une seule ligne de code :

```
Assimp::Importer importer;
const aiScene *scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs);
```

On commence par déclarer un objet Importer provenant de l'espace de données d'Assimp puis on appelle la méthode ReadFile() de cet objet. Cette méthode requiert un nom de fichier et des options en second argument. Non seulement cette méthode charge le fichier, mais elle permet de spécifier différentes options pour qu'Assimp effectue des opérations supplémentaires sur les données importées. L'option aiProcess_Triangulate a pour effet de transformer toutes les primitives du modèle en triangles si ce n'est pas déjà le cas. L'option aiProcess_FlipUVs inverse les coordonnées de textures sur l'axe y si nécessaire (on se rappelle, comme expliqué dans le chapitre sur les textures, que la plupart des images dans OpenGL sont inversées par rapport à l'axe y). D'autres options sont disponibles :

- aiProcess_GenNormals : crée les normales pour chaque sommet si le modèle n'en contient pas.
- aiProcess_SplitLargeMeshes : découpe les grandes mailles en mailles plus petites, ce qui est utile si votre système de rendu est limité en nombre de sommets et ne peut afficher que des petites mailles.
- aiProcess_OptimizeMeshes : réunit plusieurs mailles si possible, réduisant les appels d'affichage pour optimisation.

Assimp propose un large ensemble d'instructions de post-traitements, vous les trouverez **ici**. Charger un modèle avec Assimp est donc vraiment facile. La partie délicate est l'utilisation de l'objet *scene* obtenu pour traduire les données chargées en un tableau de Mesh.

La fonction loadModel() complète est donnée ci-après :

```
void loadModel(std::string path)
{
    Assimp::Importer import;
    const aiScene *scene = import.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs);
       if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
    {
        cout << "ERROR::ASSIMP::" << import.GetErrorString() << endl;
        return;
    }
    directory = path.substr(0, path.find_last_of('/'));
    processNode(scene->mRootNode, scene);
}
```

Après avoir chargé le modèle, on vérifie que la scène et le nœud racine de la scène ne sont pas nuls et l'on teste un des indicateurs pour voir si les données sont complètes. Si l'une de ces erreurs se produit, on affiche un message d'erreur au moyen de la méthode GetErrorString() et on termine la fonction. On récupère ensuite le nom du répertoire contenant le fichier.

Si tout s'est bien passé, on traite tous les nœuds de la scène en passant le nœud racine à la fonction récursive processNode(). Chaque nœud pouvant contenir des nœuds fils, nous traitons d'abord le nœud en question, et nous continuons en traitant les nœuds fils. Cette structure récursive se traite naturellement par une fonction récursive, une fonction qui effectue un certain traitement puis s'appelle elle-même avec des paramètres différents si une certaine condition est satisfaite. Dans notre cas, la condition d'arrêt sera le traitement de tous les nœuds fils.

On se rappelle que dans la structure d'Assimp, chaque nœud contient un ensemble d'indices de mailles, chaque indice pointant vers une maille particulière de l'objet scène. Nous voulons donc retrouver ces indices de mailles, puis chaque maille, et enfin traiter chacune de ces mailles, et cela pour chacun des nœuds fils du nœud principal. Le contenu de cette fonction processNode() est exposé ci-après :

```
void processNode(aiNode *node, const aiScene *scene)
{
    // traitement de toutes les mailles du nœud
    for(unsigned int i = 0; i < node->mNumMeshes; i++)
    {
        aiMesh *mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh, scene));
     }
    // effectuer la même opération pour chaque nœud fils
    for(unsigned int i = 0; i < node->mNumChildren; i++)
    {
        processNode(node->mChildren[i], scene);
    }
}
```

D'abord, pour chacun des indices de maille du nœud, on retrouve la maille correspondante en indexant le tableau mMeshes de la scène. La maille obtenue est ensuite passée à la fonction processMesh() qui retourne un objet Mesh que l'on mémorise dans le vecteur meshes.

Une fois que toutes les mailles ont été traitées, on itère sur tous les nœuds fils en appelant la même fonction pour chacun d'entre eux. Lorsqu'un nœud n'a plus de nœud fils, la fonction s'arrête.

Un lecteur attentif aura noté que l'on pourrait oublier les nœuds et simplement itérer sur chacune des mailles de la scène directement sans compliquer les choses avec les indices. La raison pour utiliser notre méthode tient en la définition d'une relation parent-fils entre les mailles. En traitant récursivement ces relations, on peut définir certaines mailles comme parentes d'autres mailles (par exemple, une maille machine sera parente d'une maille roue et de son fils pneu). Cette organisation des données est naturellement récursive.
Cependant, pour l'instant, nous n'utiliserons pas un tel système, mais cette approche est recommandée si vous souhaitez un contrôle important sur les mailles. Après tout, ces relations parent-fils sont définies par les concepteurs de modèles.

L'étape suivante est le traitement effectif des données Assimp dans la classe Mesh créée dans le chapitre précédent.

III-A-1 - D'Assimp à la classe Mesh

Transcrire un objet aiMesh en un objet maille de notre cru n'est pas trop compliqué. Il suffit d'accéder à chacune des propriétés de la maille et de la mémoriser dans notre objet. La structure générale de la fonction processMesh() devient donc :

```
Mesh processMesh(aiMesh *mesh, const aiScene *scene)
```

```
{
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    std::vector<Texture> textures;
    for(unsigned int i = 0; i < mesh->mNumVertices; i++)
    {
        Vertex vertex;
        // traitement des positions, normales et coordonnées de textures des sommets
        …
        vertices.push_back(vertex);
    }
    // Traitement des indices
    …
    // Traitement des matériaux
    if(mesh->mMaterialIndex >= 0)
    {
        …
    }
    return Mesh(vertices, indices, textures);
}
```

Traiter une maille consiste en trois parties : retrouver tous les sommets, retrouver les indices de la maille et finalement retrouver le matériau correspondant. Les données sont mémorisées dans l'un des trois vecteurs et ainsi un objet Mesh est créé et renvoyé par la fonction.

Retrouver les données de sommets est assez simple : on définit une structure Vertex que l'on ajoute au tableau vertices après chaque itération. On boucle sur les sommets de la maille (leur nombre est donné par mesh->mNumVertices). Dans la boucle, on remplit cette structure avec les données correspondantes. Pour la position des sommets, cela est fait ainsi :

```
glm::vec3 vector;
vector.x = mesh->mVertices[i].x;
vector.y = mesh->mVertices[i].y;
vector.z = mesh->mVertices[i].z;
vertex.Position = vector;
```

Nous définissons un vecteur vec3 pour y transférer les données d'Assimp. Ce vecteur est nécessaire, car Assimp possède son propre type de données pour les vecteurs, les matrices, les chaînes de caractères, etc., et leur conversion ne se fait pas facilement dans les types de données de glm.

Assimp nomme les positions de sommets mVertices, ce qui n'est pas très intuitif.

Le traitement des normales est sans surprise :

```
vector.x = mesh->mNormals[i].x;
vector.y = mesh->mNormals[i].y;
vector.z = mesh->mNormals[i].z;
vertex.Normal = vector;
```

Les coordonnées de textures sont traitées pareillement, mais Assimp autorise jusqu'à huit coordonnées de textures différentes pour un sommet, ce que nous n'utiliserons pas, en se limitant au premier ensemble de ces coordonnées. Nous vérifierons aussi que la maille contient des coordonnées de textures (ce qui n'est pas toujours le cas) :

```
if(mesh->mTextureCoords[0]) // y a-t-il des coordonnées de textures ?
{
    glm::vec2 vec;
    vec.x = mesh->mTextureCoords[0][i].x;
    vec.y = mesh->mTextureCoords[0][i].y;
    vertex.TexCoords = vec;
}
else
    vertex.TexCoords = glm::vec2(0.0f, 0.0f);
```

La structure vertex est maintenant complétée avec les attributs de sommets et l'on peut la placer dans le vecteur vertices à la fin de la boucle. Ce processus est répété pour chacun des sommets de la maille.

III-A-2 - Les indices

L'interface d'Assimp définit chaque maille comme ayant un tableau de faces où chaque face représente une primitive, ce qui dans notre cas est toujours un triangle (du fait de l'option aiProcess_Triangulate). Une face contient les indices qui définissent quels sommets doivent être affichés et dans quel ordre pour chaque primitive, donc nous itérerons sur les faces et nous mémoriserons les indices de faces dans le vecteur *indices* :

```
for(unsigned int i = 0; i < mesh->mNumFaces; i++)
{
    aiFace face = mesh->mFaces[i];
    for(unsigned int j = 0; j < face.mNumIndices; j++)
        indices.push_back(face.mIndices[j]);
}
```

Une fois la boucle extérieure terminée, nous disposons d'un ensemble complet de sommets et des indices pour afficher la maille avec glDrawElements(). Cependant, pour terminer et ajouter certains détails à la maille, il nous faut traiter aussi les matériaux de la maille.

III-A-3 - Les matériaux

De même que les nœuds, une maille ne contient qu'un indice pointant vers un matériau, et pour retrouver le matériau d'une maille, il faut utiliser le tableau mMaterials de la scène. Cet indice de matériau est disponible grâce à la propriété mMaterialIndex que nous pouvons aussi utiliser pour savoir si un matériau a été utilisé ou non :

```
if(mesh->mMaterialIndex >= 0)
{
    aiMaterial *material = scene->mMaterials[mesh->mMaterialIndex];

    std::vector<Texture> diffuseMaps = loadMaterialTextures(material,
    aiTextureType_DIFFUSE, "texture_diffuse");
    textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());

    std::vector<Texture> specularMaps = loadMaterialTextures(material,
 aiTextureType_SPECULAR, "texture_specular");
    textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
}
```

On retrouve d'abord l'objet aiMaterial du tableau mMaterials de la scène. Ensuite, nous voulons charger les textures diffuse et/ou spéculaire. Un objet matériau contient un tableau des emplacements alloués pour chaque type de texture. Les différents types de texture sont préfixés avec aiTextureType_. On utilise la fonction loadMaterialTextures() pour retrouver les textures à partir du matériau. La fonction retourne un vecteur de structures Texture que l'on peut ensuite placer à la fin du vecteur textures du modèle.

La fonction loadMaterialTextures() boucle sur toutes les textures d'un type donné, retrouve l'emplacement des fichiers texture et charge puis génère la texture et place l'information dans une structure Vertex. Le code ressemble à ceci :

```
std::vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type, std::string
 typeName)
{
    std::vector<Texture> textures;
    for(unsigned int i = 0; i < mat->GetTextureCount(type); i++)
    {
        aiString str;
        mat->GetTexture(type, i, &str);
        Texture texture;
        texture.id = TextureFromFile(str.C_Str(), directory);
        texture.type = typeName;
        texture.path = str;
```

```
            textures.push_back(texture);
        }
        return textures;
    }
```

On commence par vérifier le nombre de textures du type spécifié contenues dans le matériau avec la fonction GetTextureCount(). On retrouve ensuite chaque emplacement de texture grâce à la fonction GetTexture() qui mémorise le résultat dans une chaîne aiString. La fonction utilitaire TextureFromFile() est ensuite appelée pour charger une texture (avec la bibliothèque **SOIL**) et retourne l'identifiant de la texture. Vous pouvez étudier le code complet à la fin si vous n'êtes pas sûr de la façon dont cette fonction est écrite.

Noter que nous avons supposé que les chemins d'accès aux fichiers textures sont locaux au modèle de l'objet, donc dans le même répertoire que le modèle lui-même. On peut donc simplement concaténer la chaîne contenant le nom de la texture et le nom du répertoire que nous avons trouvé précédemment (dans la fonction loadModel()) pour obtenir le chemin d'accès complet de la texture.

Certains modèles disponibles sur internet utilisent des chemins d'accès absolus pour leurs textures, ce qui ne fonctionnera pas sur votre machine. Dans ce cas, il vous faudra éditer manuellement le fichier (si cela est possible) pour obtenir des chemins d'accès locaux pour les textures.

Et c'est tout pour importer un modèle en utilisant Assimp.

III-B - Une optimisation importante

Nous n'avons pas tout à fait fini, car il est possible d'optimiser ce travail (ce n'est pas indispensable toutefois). La plupart des scènes utilisent une même texture pour plusieurs mailles ; pensez à une maison qui utilise une texture de granit pour les murs. Cette texture peut aussi être utilisée pour le sol, le plafond, les escaliers, peut être une table… Charger une texture prend du temps et dans notre implémentation, une nouvelle texture est chargée et générée pour chaque maille même si elle a déjà été chargée auparavant. Cela forme un goulot d'étranglement pour notre mise en œuvre du chargement d'un modèle.

Nous allons modifier un peu le code en mémorisant globalement les textures chargées et lorsqu'on aura besoin d'une texture, on vérifiera si elle n'a pas été déjà chargée. Si c'est le cas, nous économiserons du temps de calcul en retrouvant cette texture. Pour comparer les textures, nous utiliserons leur chemin d'accès, qu'il faudra donc mémoriser :

```
struct Texture {
    unsigned int id;
    std::string type;
    std::string path;  // nous mémorisons le chemin d'accès pour comparaison avec d'autres
 textures
};
```

Nous archivons toutes les textures chargées dans un autre vecteur déclaré dans la classe du modèle au moyen d'une variable privée :

```
std::vector<Texture> textures_loaded;
```

Dans la fonction loadMaterialTextures() nous comparons le chemin d'accès avec tous ceux déjà archivés dans le vecteur textures_loaded. Si on le trouve, on saute la partie chargement et génération de la texture et on utilise simplement la texture déjà chargée. La fonction modifiée devient donc :

```
std::vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type, std::string
 typeName)
```

```
{
    std::vector<Texture> textures;
    for(unsigned int i = 0; i < mat->GetTextureCount(type); i++)
    {
        aiString str;
        mat->GetTexture(type, i, &str);
        bool skip = false;
        for(unsigned int j = 0; j < textures_loaded.size(); j++)
        {
            if(std::strcmp(textures_loaded[j].path.data(), str.C_Str()) == 0)
            {
                textures.push_back(textures_loaded[j]);
                skip = true;
                break;
            }
        }
        if(!skip)
        {   // si la texture n'a pas été déjà chargée, on le fait
            Texture texture;
            texture.id = TextureFromFile(str.C_Str(), directory);
            texture.type = typeName;
            texture.path = str.C_Str();
            textures.push_back(texture);
            textures_loaded.push_back(texture); // ajouter aux textures connues
        }
    }
    return textures;
}
```

Dès lors, nous avons non seulement un système universel de chargement de modèle, mais qui de plus est optimisé pour charger les modèles très rapidement.

Certaines versions d'Assimp sont vraiment lentes compilées en version debug, ainsi assurez-vous de compiler une version release si vos temps de chargement sont longs.

Vous trouverez le code complet de la classe optimisée **ici**.

III-C - Fini les conteneurs !

Bien, améliorons notre application en chargeant un modèle créé par des artistes, et pas un truc créé par le génie que je suis (vous admettrez que mes conteneurs sont certainement les plus beaux cubes que vous ayez connus). Ne voulant pas me donner trop d'importance, je fais quelquefois appel à d'autres graphistes et cette fois, nous allons utiliser la nanosuit originale utilisée par Crytek dans Crysis (et téléchargé depuis tf3dm.com pour cet exemple). Le modèle est exporté dans un fichier *.obj* avec un fichier *.mtl* contenant les textures diffuses et spéculaires ainsi que les textures de normales (des précisions plus loin). Vous pouvez télécharger ce modèle (un peu modifié) **ici**. Notez que les textures et le fichier modèle doivent se trouver dans le même répertoire.

La version que vous pouvez télécharger depuis ce site est une version modifiée dans laquelle chaque chemin d'accès aux textures a été modifié pour devenir relatif alors que dans l'original ce sont des chemins absolus.

Dans le code, déclarez un objet Model et passez l'emplacement du fichier modèle. Le modèle devrait se charger automatiquement et (sauf erreur) s'afficher dans le jeu avec la fonction Draw(). Plus besoin d'allouer des tampons mémoire, des pointeurs d'attributs et des commandes de rendu, une simple ligne suffit. Ensuite, si vous créez un simple ensemble de shaders où le fragment shader ne fait que produire la couleur des textures diffuses, le résultat sera à peu près celui-là :

Le code complet se trouve **ici**.

Vous pouvez aussi être plus créatif et introduire deux lampes ponctuelles dans la scène comme nous l'avons vu dans le **chapitre sur les éclairages** et avec les textures spéculaires vous obtiendrez cela :

Même moi, je dois admettre que c'est un peu plus sympa que mes conteneurs. Avec Assimp, vous pourrez charger quantité de modèles trouvés sur internet. Certains sites offrent gratuitement des modèles 3D à télécharger dans différents formats. Certains ne se chargent pas correctement, ont des textures qui ne fonctionnent pas ou encore utilisent des formats inconnus d'Assimp.

III-D - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.