Tuto 6

Site: lms.univ-cotedazur.fr
Cours: Realite virtuelle - EIMAD919
Livre: Tuto 6

Imprimé par: Theo bonnet
Date: vendredi 28 février 2020, 14:56

Table des matières

1. Installation de assimp

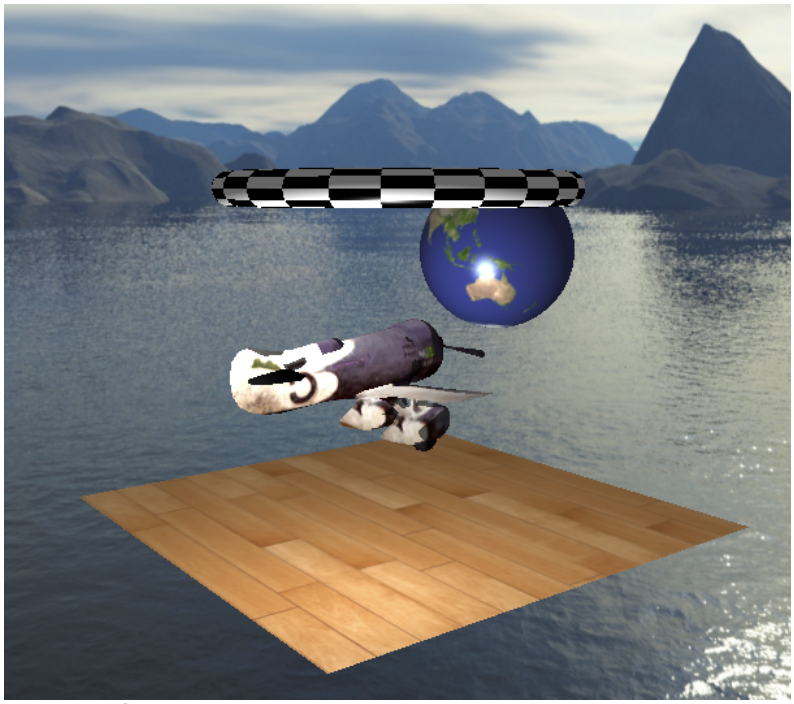
2. Utilisation de assimp

3. Classe RVMesh

4. Classe RVModel

5. Intégration

1. Installation de assimp



Bibliothèque assimp

Présentation

`Assimp` pour *Open Asset Importer* est une bibliothèque portable opensource pour importer des modèles 3D stockés dans des formats variés liés aux différents types de modeleurs 3D (3dsMax, Maya, Blender...).

Site : //assimp.org
Sources : https://github.com/assimp/assimp/releases/tag/v5.0.0 Dernière version : 5.0.0

Installation

CMake

Pour utiliser la libraire `assimp` sur nos différentes plateformes, il va falloir la compiler à partir des sources.
L'outil qui permet de faire ça est `CMake` : c'est un système de construction logicielle multiplateforme.

> Il permet de vérifier les prérequis nécessaires à la construction, de déterminer les dépendances entre les différents composants d'un projet, afin de planifier une construction ordonnée et adaptée à la plateforme. La construction du projet est ensuite déléguée à un logiciel spécialisé dans l'ordonnancement de taches et spécifique à la plateforme, Make, Ninja ou Microsoft Visual Studio. (Wikipedia)

Site : //cmake.org
Dernière version stable : 3.16.4

Sous Linux

- Installer CMake :

```
$ sudo apt-get install cmake
```

- Télécharger les fichiers sources de la bibliothèque assimp-master ;
- Décompresser dans le répertoire de travail (par exemple /home/donati/assimp-master) ;

```
$ cd home/donati/assimp-master
$ cmake CMakeLists.txt -G 'Unix Makefiles'
$ make
```

- Pour ajouter la bibliothèque dans Qt : ouvrir le ficheir .pro puis clic-droit/Ajouter une bibliothèque, puis choisir bibliothèque

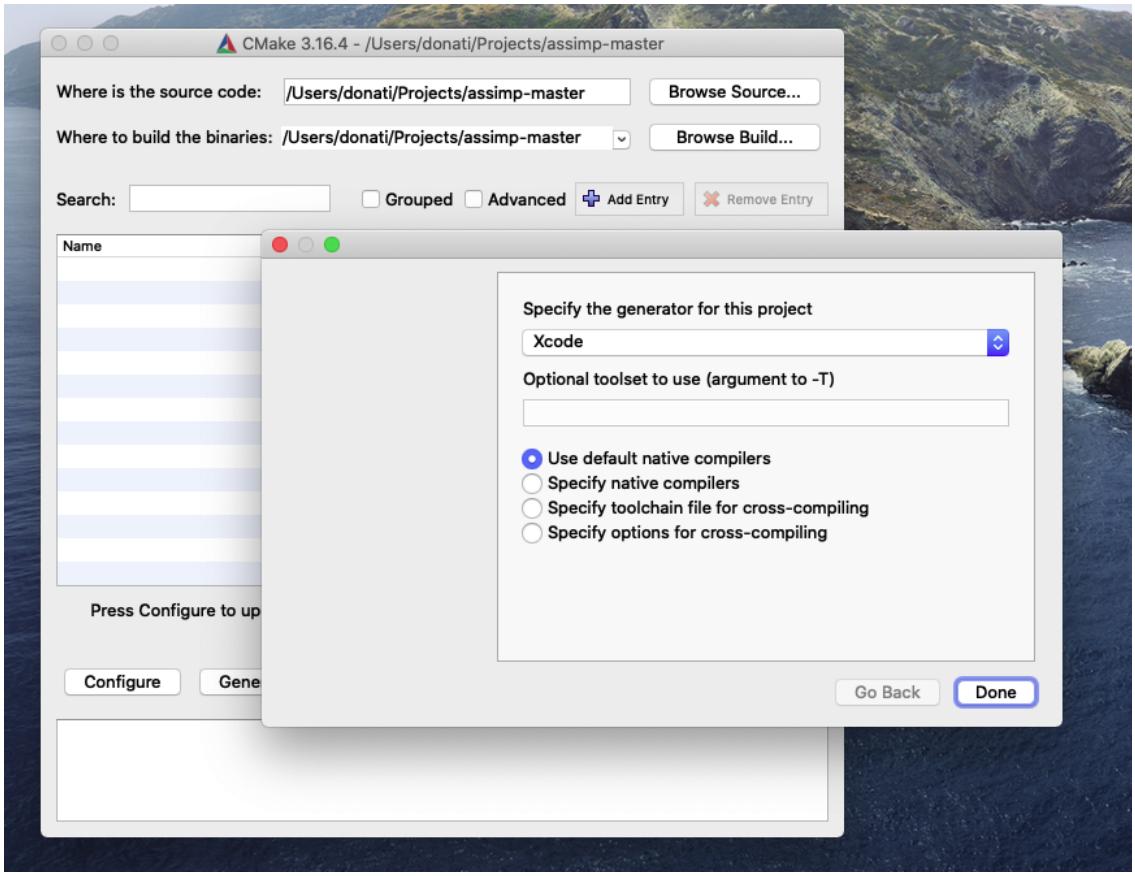
externe ; cocher linux seulement puis sélectionner le fichier dans /home/donati/assimp-master/lib/assimp.so. Normalement ça ajoute aussi le répertoire des include correspondant.

Sous Windows

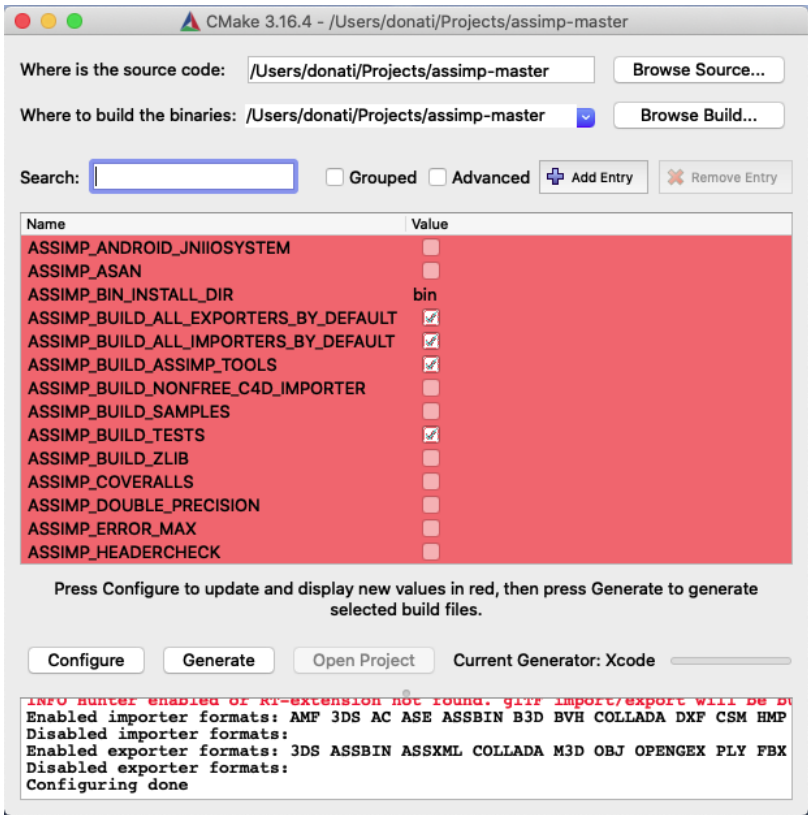
- Télécharger et installer CMake pour Windows sur http://cmake.org ;
- Télécharger les fichiers sources de la bibliothèque assimp-master.zip à partir de http://assimp.org ;
- Décompresser dans le répertoire de travail (par exemple C:\assimp-master) ;
- Lancer CMake ; choisir le répertoire contenant le code source ; choisir le répertoire où mettre les fichiers binaires (utiliser le même répertoire de préférence) ; appuyer sur le bouton Configure et choisir comme générateur le même compilateur que celui que vous utilisez pour Qt (donc soit une certaine version de VisualStudio, soit MinGW Makefiles – attention à choisir la bonne version de l'architecture 32 ou 64 bits en conformité avec le profil utilisé dans Qt) ... attendre un peu. A ce moment il y aura des tas de réglages en rouge dans la boite de dialogue. Ne rien modifier a priori et appuyer à nouveau sur Configure ; à ce moment les réglages ne sont plus rouges et vous pouvez appuyer sur le bouton Generate (image ci-dessous) :
- Appuyer sur le bouton Open Project pour lancer Visual Studio (par exemple) ; lancer la compilation avec Générer/Générer ALL_BUILD en mode Debug (puis éventuellement en mode Release) ; quand la compilation est terminée vous pouvez fermer Visual Studio. Dans le répertoire choisi pour le binaire vous trouverez les include dans C:\assimp-4.1.0\include et la librairie dans C:\assimp-master\lib\Debug ;
- Pour ajouter la bibliothèque dans Qt : ouvrir le fichier .pro puis clic-droit/Ajouter une bibliothèque, puis choisir bibliothèque externe ; cocher windows seulement puis sélectionner le fichier dans C:\assimp-master\lib\Debug\ Pour les include rajouter le répertoire C:\assimp-master\include ;
- Pour que le programme puisse trouver la bibliothèque dynamique lors de l'exécution il faut ajouter à la variable d'environnement Path le chemin vers C:\assimp-master\bin\Debug.

Sous MacOS

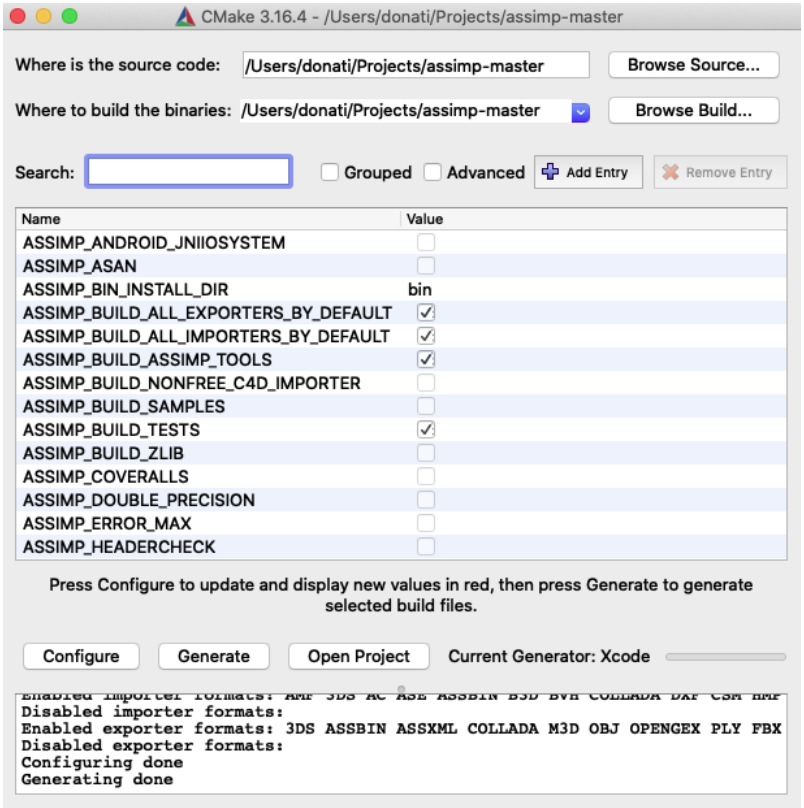
- Télécharger et installer CMake pour Mac sur http://cmake.org ;
- Télécharger les fichiers sources de la bibliothèque assimp-master.zip à partir de git ;
- Décompresser dans le répertoire de travail (par exemple /Users/donati/Projects/assimp-master) ;
- Lancer CMake ; choisir le répertoire contenant le code source ; choisir le répertoire où mettre les fichiers binaires (utiliser le même répertoire de préférence) ; appuyer sur le bouton Configure et choisir Xcode comme générateur



attendre un peu. A ce moment il y aura des tas de réglages en rouge dans la boite de dialogue.



Ne rien modifier a priori et appuyer à nouveau sur `Configure` ; à ce moment les réglages ne sont plus rouges et vous pouvez appuyer sur le bouton `Generate` (image ci-dessous) :

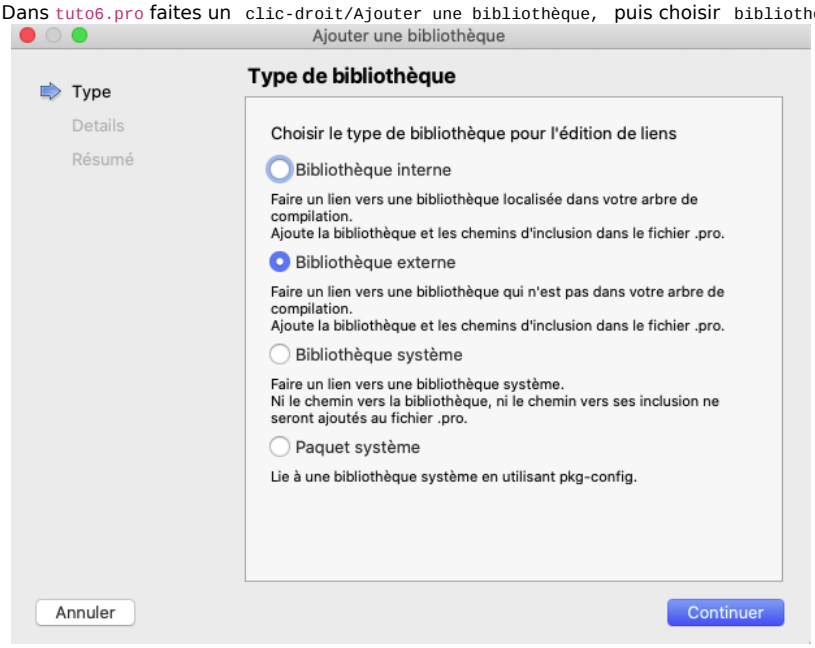


- A ce moment CMake a créé un fichier de projet `Xcode` appelé `Assimp.xcodeproj`. C'était le but de Cmake : préparer un projet destiné à être compilé par xcode et qui compile la librairie.
- Appuyer sur le bouton `Open Project` pour lancer Xcode ; lancer la compilation du target ALL_BUILD (ça prend pas mal de temps 221 fichiers a compiler) ; quand la compilation est terminée vous pouvez fermer Xcode et CMake.
  Dans le répertoire choisi pour le binaire vous trouverez les include dans `/Users/donati/Projects/assimp-master/include` et la librairie dans `/Users/donati/Projects/assimp-master/lib/Debug` : `libassimpd.dylib` (qui est en fait un alias pour la version courante `libassimpd.5.0.1.dylib`). Note que le petit d final indique que c'est la version debug. Si l'on veut la version release de la librairie (plus compacte et plus rapide) il faut recompiler le projet (dans xcode) en version release (avec Product/Scheme/Edit scheme....)

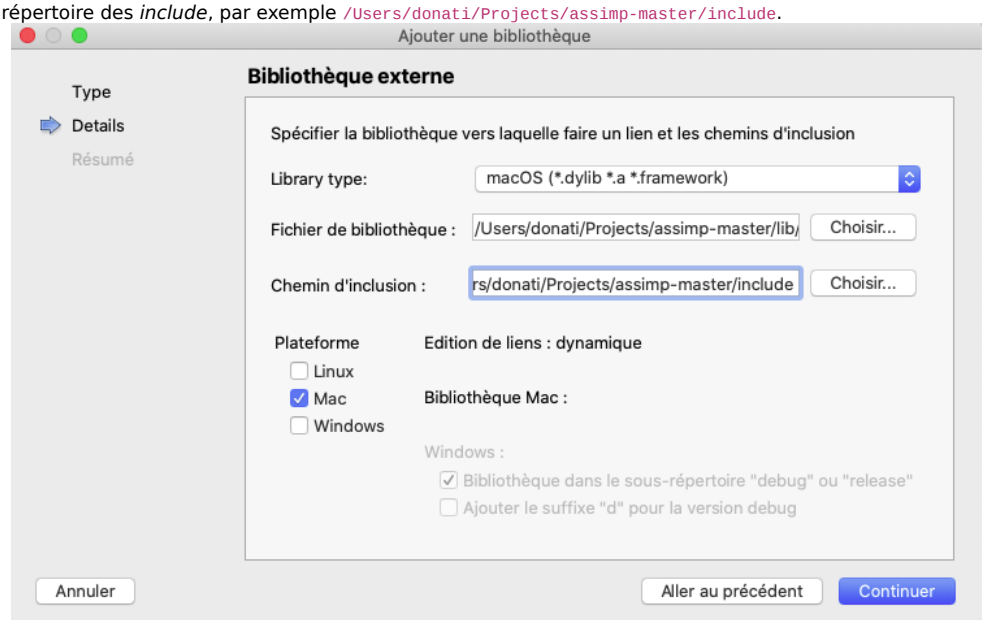
2. Utilisation de assimp

Ajouter assimp au projet Qt

Ouvrir le projet `rvtuto6.pro`. Pour pouvoir utiliser assimp dans la suite, il faut informer le compilateur de l'endroit où se trouve la bibliothèque et les fichiers d'en-tête pour les `include`.
Dans `tuto6.pro` faites un `clic-droit/Ajouter une bibliothèque,` puis choisir `bibliothèque externe ;`



cocher la plateforme que vous utilisez, puis sélectionner le fichier de librairie (le nom va changer selon que vous êtes sous Windows (.dll), Linux (.lib) ou MacOS (.dylib)) qui se trouve par exemple dans `/Users/donati/Projects/assimp-master/lib/debug` pour la version debug et `/Users/donati/Projects/assimp-master/lib/release` pour la version release. Pour les include rajouter le répertoire des *include*, par exemple `/Users/donati/Projects/assimp-master/include`.



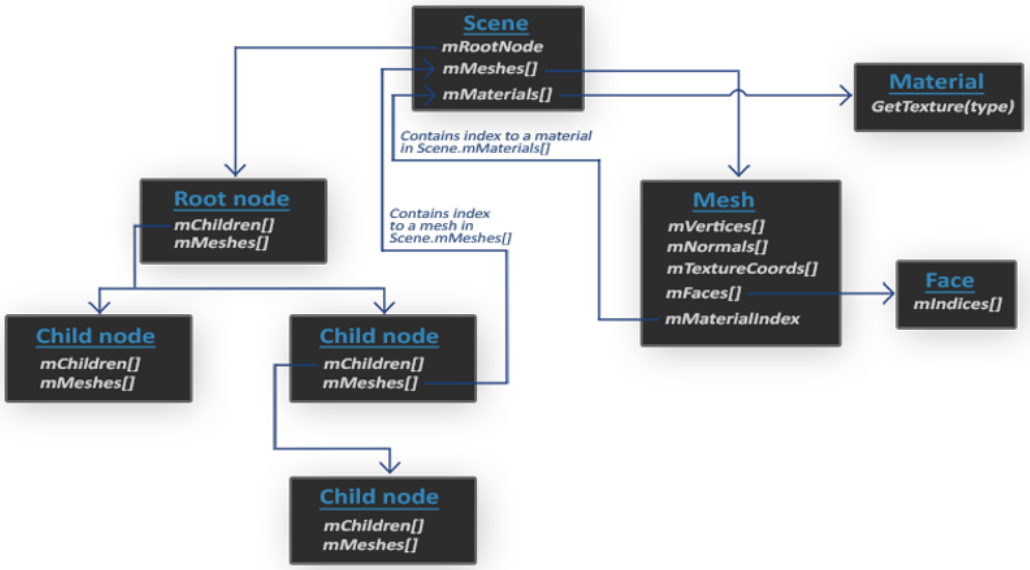
Le résultat est qu'il va y avoir à la fin de votre fichier `rvtuto6.pro`, des lignes additionnelles qui ressembleront à ça :

```
macx: LIBS += -L$$PWD/../../../../Projects/assimp-master/lib/Release/ -lassimp.5.0.1

INCLUDEPATH += $$PWD/../../../../Projects/assimp-master/include
DEPENDPATH += $$PWD/../../../../Projects/assimp-master/include
```

Utilisation de assimp

La bibliothèque `assimp` est capable de lire plusieurs formats de fichier contenant la description d'un modèle 3D et construit sa propre scène 3D avec ses propres classes : `aiScene, aiMesh, aiMaterial`



Nous allons utiliser les classes produites par assimp pour créer nos propres objets compatibles avec notre architecture de classes :

- la classe `RVMesh` qui hérite de `RVBody` est construit à partir de `aiMesh` ;
- la classe `RVModel` qui hérite aussi de `RVBody` contient un tableau de `RVMesh` (différent donc de notre classe `RVScene` qui **est** un tableau de `RVBody` et n'en hérite pas..

3. Classe RVMesh

La classe `RVMesh` hérite de `RVBody` est contient deux variables membres :

- `m_vertices` qui est un pointeur sur vecteur de `RVVertex`
- `m_indices` qui est un pointeur sur un vecteur de `uint`

On y ajoute les mutateurs de ces deux variables et les méthodes :

- surcharge de `initializeBuffer`
- surcharge de `draw()`

```
#include "rvbody.h"

class RVMesh : public RVBody
{
public:
    RVMesh();
    ~RVMesh() override;

    void initializeBuffer() override;
    void draw() override;

    void setVertices(QVector<RVVertex> *vertices);
    void setIndices(QVector<uint> *indices);

protected:
    QVector<RVVertex> *m_vertices;
    QVector<uint> *m_indices;
};
```

Constructeur et destructeur

```
RVMesh::RVMesh()
    :RVBody()
{
    m_VSFileName = ":/shaders/VS_lit_texture.vsh";
    m_FSFileName = ":/shaders/FS_lit_texture.fsh";
}

RVMesh::~RVMesh()
{
    delete m_vertices;
    delete m_indices;
}
```

Mutateurs

```
void RVMesh::setVertices(QVector *vertices)
{
    m_vertices = vertices;
    m_numVertices = m_vertices->length();
}

void RVMesh::setIndices(QVector *indices)
{
    m_indices = indices;
    m_numIndices = m_indices->length();
}
```

initializeBuffer()

Son but c'est de recopier les données sur le maillage qui sont stockées dans les variables membres, dans le VBO et le IBO (c'est à dire dans la mémoire vidéo) :

```
void RVMesh::initializeBuffer()
{
    m_vbo.bind();
    m_vbo.allocate(&m_vertices->at(0), m_numVertices*sizeof (RVVertex));
    m_vbo.setUsagePattern(QOpenGLBuffer::StaticDraw);
    m_vbo.release();

    m_ibo.bind();
    m_ibo.allocate(&m_indices->at(0), m_numIndices*sizeof (uint));
    m_ibo.setUsagePattern(QOpenGLBuffer::StaticDraw);
    m_ibo.release();
}
```

Remarque : on n'a pas à surcharger `initializeVAO()` car la version par défaut de `RVBody` est parfaitement adaptée puisque le `RVVertex` qu'on utilise dans `m_vertices` est le même que celui de `RVBody` , c'est à dire `position, texCoord, normal`.

draw()

La méthode `draw` de `RVMesh` est identique à celle écrite dans le Tuto5 pour `RVMesh` (qui utilise un IBO, ce qui n'est pas le cas de `RVPlan`) puisque les maillages utilisés par `RVMesh` sont : `VS_lit_texture.vsh et FS_lit_texture.fsh` qui intègrent texture et éclairage.
La seule modification à faire est de désactiver l'option `ClampToEdge` qui traite des coordonnées textures qui dépassent l'intervalle [0, 1].

```
void RVMesh::draw()
{
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    if (m_wireFrame)
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    else
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

    glDisable(GL_CULL_FACE);

    if (m_texture) {
        glEnable(GL_TEXTURE_2D);
        glEnable(GL_TEXTURE0);
        //m_texture->setWrapMode(QOpenGLTexture::ClampToEdge);
        m_texture->setMinificationFilter(QOpenGLTexture::LinearMipMapLinear);
        m_texture->setMagnificationFilter(QOpenGLTexture::Linear);
        //Liaison de la texture
        m_texture->bind();
    }

    m_program.bind();
    m_vao.bind();

    m_program.setUniformValue("u_ModelMatrix", this->modelMatrix());
    m_program.setUniformValue("u_ViewMatrix", m_camera->viewMatrix());
    m_program.setUniformValue("u_ProjectionMatrix", m_camera->projectionMatrix());
    m_program.setUniformValue("u_opacity", m_opacity);
    m_program.setUniformValue("u_color", m_globalColor);
    m_program.setUniformValue("texture0", 0);

    m_program.setUniformValue("light_ambient_color", m_light->ambient());
    m_program.setUniformValue("light_diffuse_color", m_light->diffuse());
    m_program.setUniformValue("light_specular_color", m_light->specular());
    m_program.setUniformValue("light_position", m_light->position());
    m_program.setUniformValue("light_specular_strength", this->specStrength());

    m_program.setUniformValue("eye_position", m_camera->position());

    glDrawElements(GL_TRIANGLES, m_numIndices, GL_UNSIGNED_INT, nullptr);

    m_vao.release();
    m_program.release();

    if (m_texture) {
        m_texture->release();
        glDisable(GL_TEXTURE0);
        glDisable(GL_TEXTURE_2D);
    }
}
```

4. Classe RVModel

Principe

C'est la classe `RVModel` qui utilise la bibliothèque `assimp` :

- on utilise l'importeur de Assimp`Assimp::Importer` pour lire le modèle à partir d'un fichier (à partir du disque et pas à partir des ressources Qt)
- l'importeur produit une scène `aiScene` qui est composée (entre autre) d'un tableau de `aiMesh`
- on parcourt cette arborescence et on construit des instances de `RVMesh` dans un tableau `m_meshes`

L'utilisateur de `RVModel` devra initialiser le modèle à partir d'un fichier, puis traiter cet objet comme les autres objets que l'on a vu jusqu'à maintenant : lui fournir une caméra, lui fournir une lumière, le placer dans la scène et appeler sa méthode `initialize()`.

Composition de RVModel

`RVModel` hérite de `RVBody` et a les variables membres suivantes :

- `m_meshes` qui est un tableau de pointeurs sur des `RVMesh` ;
- un entier `m_nbMeshes` qui donne le nombre de maillage qui composent le modèle ;
- une instance de `QDir` appelée `m_directory` qui stocke la place (dans l'arborescence des fichiers) où se trouve le fichier du modèle et les fichiers des textures utilisées par le modèle (qui doivent donc forcément se trouver au même endroit).

En plus des fonctions à surcharger habituelles, il y a aussi 3 méthodes privées qui seront utilisées par le constructeur.

```
#include <QString>
#include <QDir>
#include <QVector>

#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
#include <assimp/matrix4x4.h>

#include "rvmesh.h"

class RVModel : public RVBody
{
public:
    RVModel(QString nomFichier);

    void draw() override;
    void initializeBuffer() override;
    void initializeVAO() override {}

protected:
    QVector<RVMesh*> m_meshes;
    QDir m_directory;
    int m_nbMeshes;

private:
    void loadModel(QString nomFichier);
    void processNode(aiNode* node, const aiScene* scene);
    RVMesh* processMesh(aiMesh* mesh, const aiScene* scene);
};
```

Construction du RVModel

Le constructeur utilise le nom de fichier pour en extraire le répertoire dans lequel le fichier est stocké (qui sera utilisé ensuite pour charger les textures), puis appelle `loadModel`

```
RVModel::RVModel(QString fileName)
    :RVBody(), m_meshes(), m_directory(fileName)
{
    m_directory.makeAbsolute();
    QString filePath = m_directory.path();
    m_directory.cdUp();
    loadModel(filePath);
}
```

`loadModel(QString nomFichier)` qui lit le modèle à partir du fichier, construit la scène assimp et appelle `processNode` sur tous les noeuds de la scène :

```
void RVModel::loadModel(QString fileName)
{
    Assimp::Importer import;
    const aiScene* scene = import.ReadFile(fileName.toStdString(), aiProcess_Triangulate | aiProcess_FlipUVs);

    if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
    {
        qDebug() << "ERROR::ASSIMP::" << import.GetErrorString() << endl;
        return;
    }

    // Lance le traitement de tous les noeuds de la scène
    processNode(scene->mRootNode, scene);

    // Stocke le nombre de maillages contenu dans la scène
    m_nbMeshes = m_meshes.size();
}
```

`processNode()` qui parcourt tous les maillages assimp attachés à ce noeud et appelle `processMesh` dessus pour en faire des `RVMesh` qui sont ensuite ajoutés à `m_meshes` :

```
void RVModel::processNode(aiNode* node, const aiScene* scene)
{
    // On parcourt tous les maillages rattachés à ce noeud
    for(GLuint i = 0; i < node->mNumMeshes; i++)
    {
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        //on convertit mesh qui est une instance de aiMesh
        //en maillage qui est une instance de Mesh
        RVMesh* rvMesh = processMesh(mesh, scene);

        //on récupère l'information de placement de ce noeud
        //pour en faire la matrice world du maillage
        aiMatrix4x4t<float> mat =  node->mTransformation;
        aiVector3t<float> scal;
        aiQuaterniont<float> rot;
        aiVector3t<float> pos;
        mat.Decompose(scal, rot, pos);
        rvMesh->setPosition(QVector3D(pos.x, pos.y, pos.z));

        //on ajoute maillage au tableau des Mesh
        m_meshes.append(rvMesh);
    }
    //On appelle récursivement la fonction sur tous les fils du noeud
    for(GLuint i = 0; i < node->mNumChildren; i++)
    {
        processNode(node->mChildren[i], scene);
    }
}
```

`processMesh` transforme un `aiMesh` en `RVMesh` en parcourant tous les sommets du maillage et en construisant un tableau de `RVVertex` et de `uint` qui sont passés au nouveau `RVMesh`.

```
RVMesh* RVModel::processMesh(aiMesh* mesh, const aiScene* scene)
{
    QVector<RVVertex>* vertices = new QVector<RVVertex>();
    QVector<GLuint>* indices = new QVector<GLuint>();

    //On travaille sur les sommets du aiMesh
    for(GLuint i = 0; i < mesh->mNumVertices; i++)
    {
        //Le RVVertex qui va contenir les infos
        RVVertex vertex;

        //On copie les coordonnées
        QVector3D vector;
        vector.setX(mesh->mVertices[i].x);
        vector.setY(mesh->mVertices[i].y);
        vector.setZ(mesh->mVertices[i].z);
        vertex.position = vector;

        //on copie les normales
        QVector3D normal;
        normal.setX(mesh->mNormals[i].x);
        normal.setX(mesh->mNormals[i].y);
        normal.setX(mesh->mNormals[i].z);
        vertex.normal = normal;

        //on copie les coordonnées textures
        QVector2D tex(0, 0);
        if(mesh->mTextureCoords[0]) // s'il y en a
        {
            tex.setX(mesh->mTextureCoords[0][i].x);
            tex.setY(mesh->mTextureCoords[0][i].y);
        }
        // sinon c'est (0, 0)
        vertex.texCoord = tex;

        //Notre vertex est pret on le met dans le tableau
        vertices->append(vertex);
    }
    //On travaille sur les indices
    for(GLuint i = 0; i  mesh->mNumFaces; i++)
    {
        aiFace face = mesh->mFaces[i];
        for(GLuint j = 0; j < face.mNumIndices; j++)
            indices->append(face.mIndices[j]);
    }

    //On construit le Mesh
    RVMesh* rvMesh = new RVMesh();
    rvMesh->setVertices(vertices);
    rvMesh->setIndices(indices);

    //Récupération de la texture
    if(mesh->mMaterialIndex > 0)
    {
        //On récupère le matériau de l'aiMesh
        aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
        //On en extrait le nom du fihier texture
        aiString str;
        material->GetTexture(aiTextureType_DIFFUSE,0, &str);
        //On l'utilise pour initialiser la texture de maillage
        rvMesh->setTexture(m_directory.filePath(str.C_Str()));
    }
    return rvMesh;
}
```

Méthodes de RVBody à surcharger

Les méthodes `draw()` et `initializeBuffer()` ne font rien d'autre que d'appeler les méthodes respectives sur chacun des `RVmesh` que contient le `RVModel` sauf que `initializeBuffer` va appeler directement `initialize()` pour que chaque `RVMesh` soit correctement initialisé.

```
void RVModel::draw()
{
    foreach (RVMesh* mesh, m_meshes) {
       mesh->draw();
    }
}

void RVModel::initializeBuffer()
{
    foreach (RVMesh* mesh, m_meshes) {
       mesh->initialize();
    }
}
```

Cela ne suffit pas pour que `RVModel` soit opérationnel : il faut aussi que lorsqu'on passe la caméra ou la lumière à une instance de `RVModel` , celle-ci la passe à tous les `RVMesh` . Il faut donc **dans RVBody**, rendre virtuelles les méthodes `rotate, translate` et les mutateurs `setCamera, setLight, setPosition, setOrientation, setScale, setScale` pour pouvoir les surcharger dans `RVModel` .
Par exemple (et c'est la même chose pour les autres) :

```
void RVModel::setCamera(RVCamera *camera)
{
    m_camera = camera;
    foreach (RVMesh* mesh, m_meshes) {
       mesh->setCamera(camera);
    }
}

void RVModel::rotate(float angle, QVector3D axis)
{
    RVBody::rotate(angle, axis);
    foreach (RVMesh* mesh, m_meshes) {
       mesh->rotate(angle, axis);
    }
}

void RVModel::translate(QVector3D position)
{
    RVBody::translate(position);
    foreach (RVMesh* mesh, m_meshes) {
       mesh->translate(position);
    }
}

void RVModel::setOrientation(float yaw, float pitch, float roll)
{
    RVBody::setOrientation(yaw, pitch, roll);
    foreach (RVMesh* mesh, m_meshes) {
       mesh->setOrientation(yaw, pitch, roll);
    }
}
```

5. Intégration

Pour tester avec le modèle d'avion fourni (modéle dans `GeeBee2.x` et texture dans `Geebee.jpg` dans un répertoire `Model`) il suffit (après avoir mis les bons *include*) de modifier le type de `m_body` dans `RVWidget::initializeGL()` en écrivant (avec le bon nom de répertoire) :

```
    m_body = new RVModel("/Users/donati/model/GeeBee2.x");
```

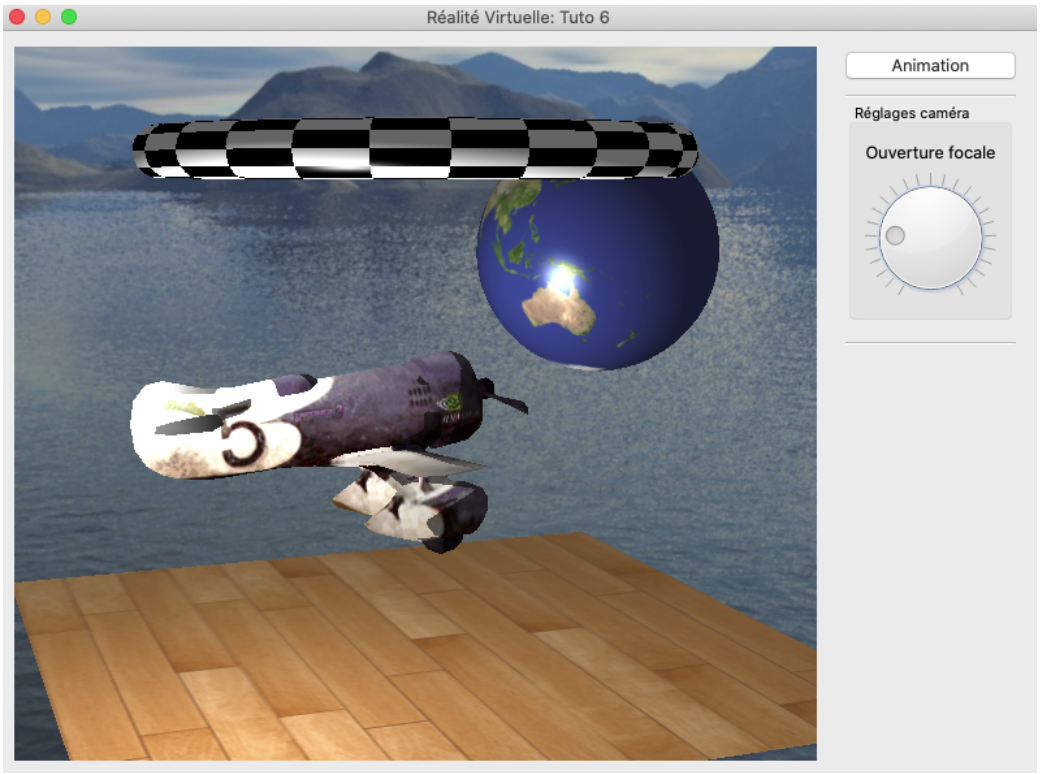
La texture semble mal collée : en fait elle est à l'envers. Et cela ne s'améliore pas si on met à false, le deuxième argument de `setTexture` dans les dernières lignes de `RVModel::processMesh` car la texture a besoin d'un retournement vertical et pas horizontal. Pour résoudre le problème il faut modifier la méthode `setTexture` de `RVBody` pour qu'elle ait trois arguments en tout : le nom de la texture est deux booléens, le premier pour le renversement horizontal, le second pour le renversement vertical (toute les deux à `true` par défaut.

```
void setTexture(QString textureFilename, bool hMirrored = true, bool vMirrorred = true);
```

Dans l'implémentation de la méthode on utilise les arguments de la méthode `QImage::mirrorred` :

```
void RVBody::setTexture(QString textureFilename, bool hMirrored, bool vMirrorred)
{
    if (m_texture) {
        m_texture->destroy();
        delete m_texture;
    }
    m_texture = new QOpenGLTexture(QImage(textureFilename).mirrored(hMirrored,vMirrorred));
}
```

Ainsi, si dans `processMesh` on passe `false, false` à `setTexture`, on a un avion avec une texture enfin droite.


Réalité Virtuelle: Tuto 6
Animation
Réglages caméra
Ouverture focale