Tuto 2

Site: lms.univ-cotedazur.fr
Cours: Realite virtuelle - EIMAD919
Livre: Tuto 2

Imprimé par: Theo bonnet
Date: vendredi 28 février 2020, 14:52

Table des matières

1. Objectifs

2. Classe RVCamera

2.1. Projections

2.2. Projection perspective

3. Classe RVBody

3.1. Code de RVBody

3.2. Position et Orientation

3.3. Classe RVPyramid

4. Nettoyage de RVWidget

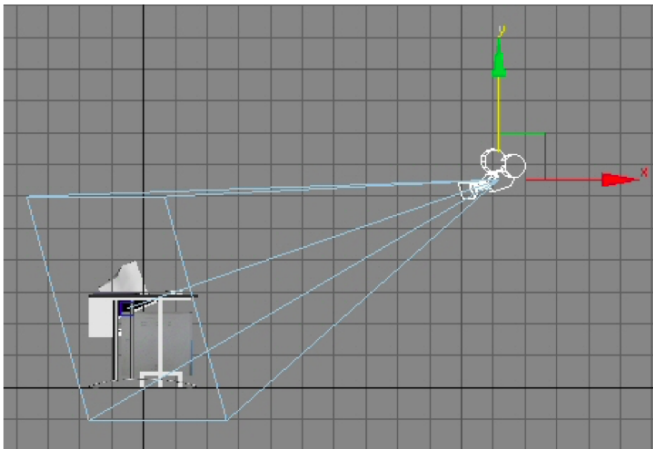
Tutoriel n°2

Le but de ce tutoriel et du TP qui en prend la suite est d'ajouter un peu de *conception objet* à ce qui a été fait dans le premier TP afin d'avoir un code plus clair, plus facile à faire évoluer au fur et à mesure de la complexité des objets 3D que l'on va utiliser.
Objectifs :

- Cantonner `RVWidget` au rôle de classe *Vue-Controlleur* qui gère les interactions avec l'utilisateur **plus** le contexte de rendu OpenGL mais **sans** le détail des objets et des caméras (et des lumières ensuite) qui composent la scène.
- Définir une classe `RVBody` *abstraite* qui soit la **classe de base** pour tous les objets 3D que l'on va voir dans le cours. Cette classe, et les classes qui en hériteront, encapsulent le code OpenGL propre à chaque objet :
  - les buffers qui définissent l'objet : le *vertex buffer object* (vbo) qui contient les sommets, l'*index buffer object* (ibo) qui éventuellement contient les indices des sommets (lorsqu'on utilise un rendu indexé) et le *vertex array object* (vao) qui enregistre le lien entre vbo et vertex shader ;
  - les programmes de shader : le *vertex shader* (vs) qui traite les sommets du vbo et le fragment shader (fs) qui travaille sur chaque pixel de l'image ;
  - la matrice `model` qui sert à placer l'objet sur la scène ;
  - les commandes de rendu.
- Définir une autre classe `RVCamera` qui sera la classe de base qui représente l'observateur sous forme de *caméra virtuelle*. Cette classe encapsule les deux matrices `view` et `projection` qui sont, avec la matrice `model` (propre à chaque objet), nécessaires au rendu (au vertex buffer en fait). Nous allons progressivement implémenter plusieurs classes filles de cette classe en fonction du type de camera que l'on voudra modéliser (caméra sphérique, caméra subjective, caméra stéréo, etc...).

Ainsi on pourra dans la même scène avoir plusieurs objets, plusieurs caméras.

Classe RVCamera



La classe RVCamera et ses classes filles seront utilisées pour représenter l'observateur qui regarde la scène et *prend une photo* de cette scène.
Donc l'observateur est défini par :

- Sa **position** dans la scène. C'est un point dont les coordonnées sont exprimées *dans le repère de la scène*.
  Donc une variable membre `QVector3D` appelée `m_position`.
- Le point qu'il regarde ou la direction dans laquelle il regarde ; plutôt que de définir un vecteur qui représenterait la direction de son regard, nous choisissons de spécifier un point, appelé la **cible** de la caméra, qui est le point ciblé par le regard (c'est à dire le point qui apparaitra au centre de la *photo* prise par la caméra virtuelle). Ce point est aussi exprimé dans le repère de la scène.
  Donc une variable membre `QVector3D` appelée `m_target`.
- Pour terminer de spécifier parfaitement le placement de la caméra, il reste à définir son orientation dans l'espace par rapport à l'axe position-cible. Car comme dans un appareil photo *réel*, on peut faire tourner l'appareil de prise de vue pour passer d'une image en mode *portrait* en une image en mode *paysage*, ou même choisir de donner une autre inclinaison... Traditionnellement on spécifie cette orientation en définissant un vecteur (unitaire) appelé `up` qui donne la direction du *haut* (c'est à dire la direction verticale) dans l'image produite. Donc pour une photo en mode paysage on choisira le vecteur (0, 1, 0).
  C'est une troisième variable membre `QVector3D` appelée `m_up`.

```
#include <QMatrix4x4>
#include <QVector3D>

class RVCamera
{
public:
    RVCamera();
    virtual ~RVCamera();

    QMatrix4x4 viewMatrix();
    QMatrix4x4 projectionMatrix();

    QVector3D position() const;
    void setPosition(const QVector3D &position);

    QVector3D target() const;
    void setTarget(const QVector3D &target);

    QVector3D up() const;
    void setUp(const QVector3D &up);

protected:
    QVector3D m_position;
    QVector3D m_target;
    QVector3D m_up;
};
```

Cette classe aura donc (en plus des accesseurs et mutateurs) :

- Un constructeur qui donne des valeurs par défaut aux trois variables membres :

```
RVCamera::RVCamera()
{
    m_position = QVector3D(0, 0, 0);
    m_target = QVector3D(0, 0, -1);
    m_up = QVector3D(0, 1, 0);
}
```

- Une méthode `viewMatrix` qui renvoie la matrice de vue associée à la caméra, c'est à dire la matrice 4x4 associée au changement de repère depuis le repère de la scène vers le repère de la caméra où la position de la caméra est l'origine, elle pointe dans la direction des z négatifs (0, 0, -1) et son "haut" est (0, 1, 0). On utilise la méthode `LookAt()` de la classe `QMatrix4x4` qui fait exactement ça.

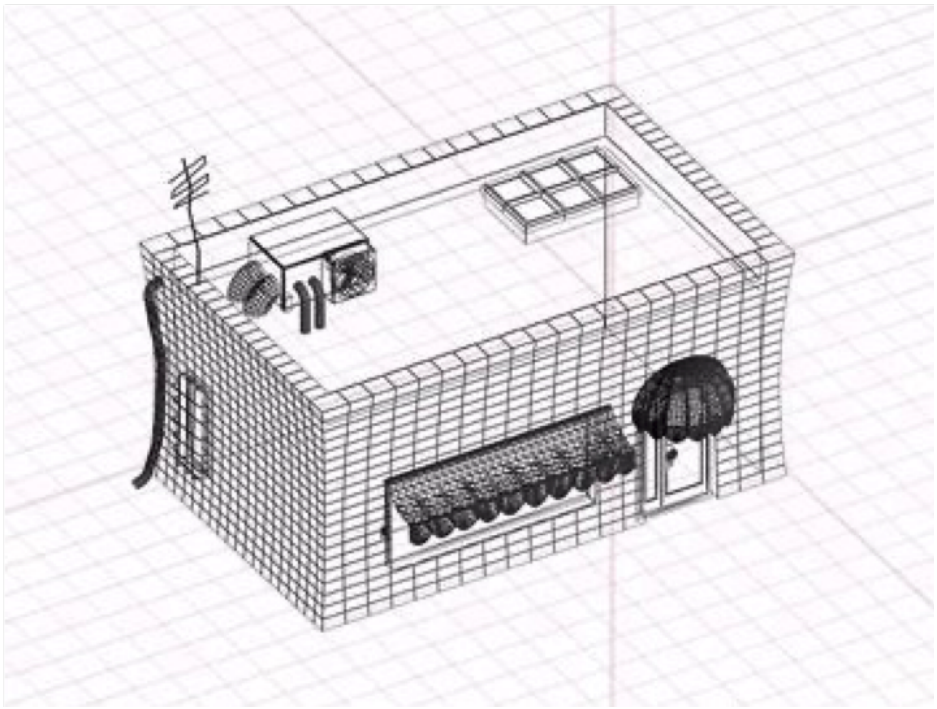
```
QMatrix4x4 RVCamera::viewMatrix()
{
    QMatrix4x4 view;
    view.lookAt(m_position, m_target, m_up);
    return view;
}
```

- Une méthode `projectionMatrix` qui renvoie la matrice de projection utilisée par la caméra pour passer du repère de la caméra au coordonnées normalisées de projection. Les détails de cette matrice font l'objet des chapitres suivants.

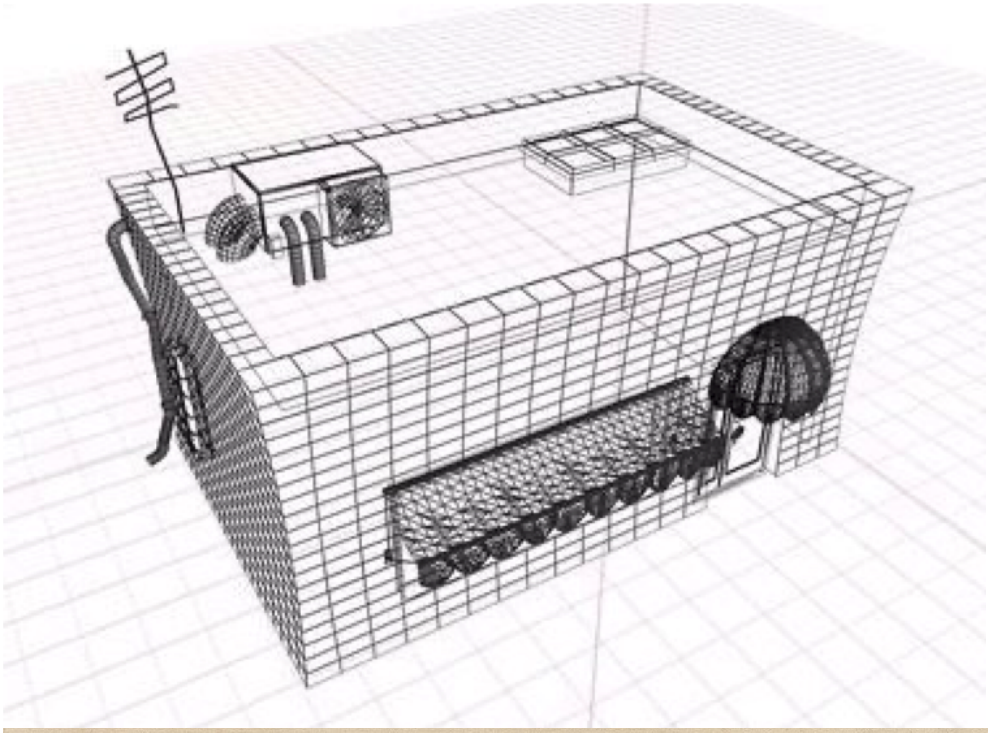
Projections

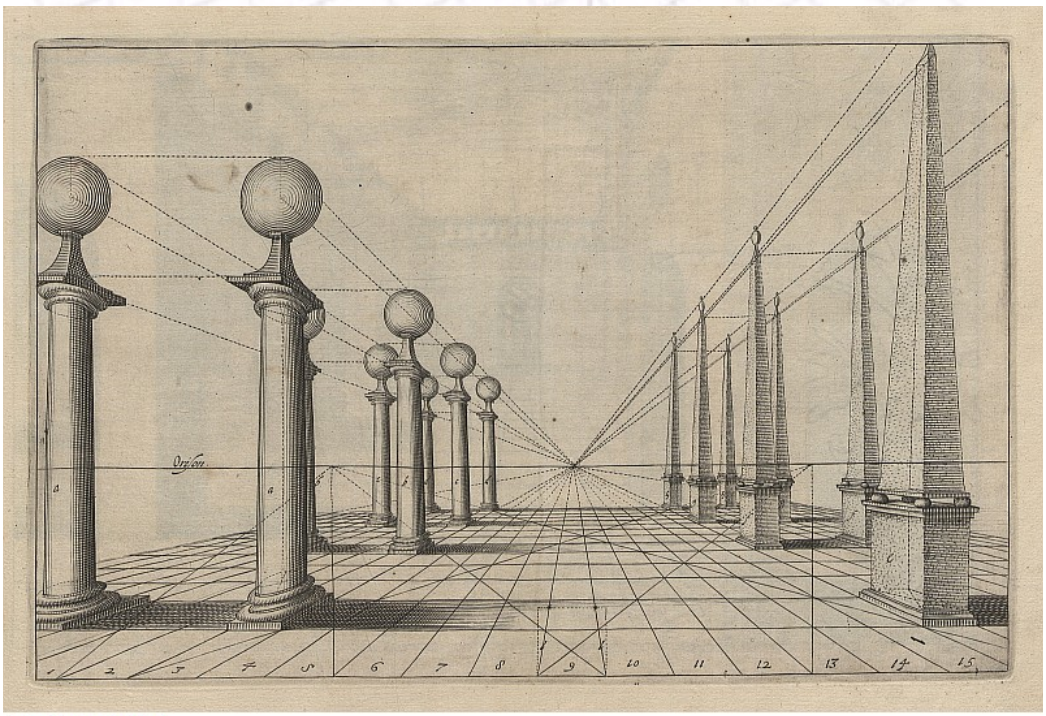
On divise les projections dans le plan en deux familles selon la façon dont se fait la projection :

- s'il existe une direction telle que toutes les projections se font parallèlement à cette direction, on parle de **projection parallèle** ; selon l'angle entre les directions de projection et le plan de projection, ce sera une projection parallèle orthogonale (comme la projection isométrique où la direction de projection est (1, 1, 1)) ou projection parallèle oblique (projection cavalière). Mathématiquement il s'agit d'une application affine.



- si, en revanche, la projection se fait selon des droites concourrantes vers un centre de projection, on dit qu'il s'agit d'une **projection perspective**. Cette transformation n'est pas une application affine car elle ne réspecte pas le parallélisme. Elle modélise (plus fidèlement que la projection orthogonale) la façon dont fonctionne l'appareil photo où une lentille fait converger les rayons lumineux vers le centre focal avant d'impressionner la péllicule (où un capter ccd). C'est un peu moins vrai pour l'oeil humain à cause de la sphéricité de la rétine mais c'est quand même le modèle qui est couramment utilisé. En tout cas, ces projections produisent l'effet de *réduction perspective* qui fait apparaître plus petits les objets éloignés (ce qui n'est pas le cas pour les projections parallèles orthogonales) et fait converger les droites parallèles vers des points de fuite à l'horizon.

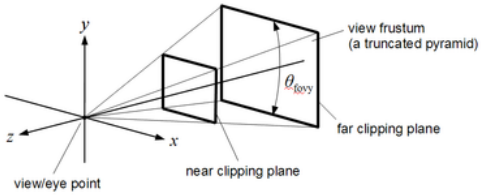




Quelle: Deutsche Fotothek

Projection perspective

Nous allons faire en sorte que la classe `RVCamera` soit capable d'utiliser aussi bien une projection perspective qu'une projection orthogonale.
La projection perspective est ce que nous avons utilisé dans le TP1.
Dans le repère de la caméra, l'origine est le centre de projection et la demi-droite des z négatif est l'axe de la caméra. Le **volume de vue** (*view frustum* en anglais) est une pyramide tronquée qui délimite la zone de l'espace 3D qui va apparaître dans l'image (sur la pellicule).



Ce sont les propriétés de ce volume de vue qui permettent de définir la matrice de projection : il y a plusieurs façons de la définir, mais on va faire le même choix que dans le tuto n° 1.

Les variables membres à rajouter à `RVCamera` sont donc :

- `m_fov` qui est un *float* qui représente l'angle (en degrés) qui donne l'ouverture verticale du volume de vue ;
- `m_aspect` qui est un *float* qui donne le format du rectangle obtenu par l'intersection du volume de vue avec le plan de projection $z = -1$
- `m_zMin` et `m_zMax` correspondent aux plans $z = z_{Min}$ (plan de fenêtrage avant : *near clipping plane*) et $z = z_{Max}$(plan de fenêtrage arrière : *far clipping plane*) qui délimitent en avant et en arrière la pyramide de vue.

```
class RVCamera{

    ...

    float fov() const;
    void setFov(float fov);
    float aspect() const;
    void setAspect(float aspect);
    float zMin() const;
    void setZMin(float zMin);
    float zMax() const;
    void setZMax(float zMax);

protected:
    float m_fov;
    float m_aspect;
    float m_zMin;
    float m_zMax;
};
```

On doit initialiser les nouvelles variables membres dans le constructeur :

```
RVCamera::RVCamera()
{
    ....

    m_fov  = 40;
    m_aspect = 1.33f;
    m_zMax = 100;
    m_zMin  = 0.1f;
}
```

Enfin on peut passer à la définition de la méthode `projectionMatrix`.

```
QMatrix4x4 RVCamera::projectionMatrix()
{
    QMatrix4x4 proj;
    proj.perspective(m_fov, m_aspect, m_zMin, m_zMax);
    return proj;
}
```

Dans le cadre du *pipeline de rendu* d'OpenGL, la matrice de projection doit transformer la pyramide tronquée du volume de vue en un cube de coté 2 où l'on a normalisé les coordonnées des points de façon à ce que x, y et z soient entre -1 et 1. Tout le reste du processus de OpenGL (discrétisation des pyramides et création des fragments) se fait avec ces coordonnées normalisées. Les coordonnées x et y vont devenir les coordonnées d'un pixel du buffer d'image et z est utilisé pour calculer le buffer de profondeur (z-buffer) qui est l'algorithme utilisé par OpenGL pour l'élimination des parties cachées.
Dans notre classe `RVCamera` nous utilisons la méthode `perspective` de `QMatrix4x4` comme dans le tuto1, mais on aurait pu utiliser une méthode plus sophistiquée, `frustum`, qui donne plus de liberté dans la conception du volume de vue (et c'est ce que nous

utiliserons dans le cadre de la caméra stéréo).

Classe RVBody

Le but de la classe `RVBody` est de servir de case de base à **tous** les objets 3D que nous allons définir dans la suite du cours. Elle aura donc tendance à évoluer mais pour l'instant nous allons définir la version v.1.0 qui est basée sur ce qui a déjà été fait dans le TP n°1. Cette classe est une **classe abstraite** car elle va contenir **trois méthodes virtuelles pures** qui devront être obligatoirement implémentées dans les classes filles :

- la méthode `initializeBuffer()` qui prépare et remplit le vertex buffer (et aussi éventuellement l'index buffer) qui contient les sommets des faces du maillage qui compose l'objet,
- la méthode `initializeVAO()` qui prépare le *Vertex Array Object*, c'est à dire qu'elle se charge de définir le lien entre les données stockées dans le vertex buffer et les attributs du programme du vertex shader,
- la méthode `draw()` qui lance le rendu.

Donc même si cette classe est *vide* car elle ne représente aucun objet concret, nous allons y mettre les informations communes à tout objet 3D.
Déja, concernant la partie OpenGL, la classe hérite de `QOpenGLFunctions` et définit les variables membres suivantes :

- un vertex buffer object `m_vbo`,
- un vertex array object `m_vao`,
- un index buffer object `m_ibo`,
- un programme de shader `m_program`,
- trois entiers `m_numVertices, m_numTriangles, m_numIndices` qui donnent des informations sur la taille du maillage utilisé pour dessiner l'objet,
- deux `QString` pour stocker les noms des fichiers qui contiennent le programme du vertex shader et celui du fragment shader,

Et contient, en plus des méthodes virtuelles pures citées plus haut :

- un constructeur qui appelle le constructeur de la classe parent, initialise les variables membres et crée (via leur méthode create()) les différents objets OpenGL;
- une méthode `initialize()` qui a la responsabilité d'appeler `initializeShader()`, `initializeBuffer()` et `initializeVAO()` (dans cet ordre).
- une méthode `initializeShader()` copiée à partir de la même méthode de `RVWidget` du Tuto n°1 qui initialise `m_program` avec les programmes de shader. Cette fonction ne bougera plus, c'est pourquoi elle n'est pas virtuelle.

La classe contient aussi comme variable membre un pointeur sur un objet `RVCamera` qui représente la camera active que `draw()` doit utiliser lors du rendu.
Enfin, tout objet 3D doit être positionné dans l'espace : non seulement il faut donner les 3 coordonnées d'un point de référence (centre de gravité par exemple) dans le repère de la scène 3D, `m_position`, mais aussi il faut définir son orientation dans l'espace, c'est à dire son *attitude*, `m_orientation`.

```
class RVBody : public QOpenGLFunctions
{
public:
    RVBody();
    virtual ~RVBody();

    virtual void draw() = 0;
    virtual QMatrix4x4 modelMatrix();

    void initialize();
    void initializeShader();
    virtual void initializeBuffer() = 0;
    virtual void initializeVAO() = 0;

    // accesseurs et mutateurs .....

protected:
    QString m_VSFileName;               //! : nom du fichier qui contient le vertex shader
    QString m_FSFileName;               //! : nom du fichier qui contient le fragment shader
    QOpenGLBuffer m_vbo;                //! : vertex buffer qui stocke les sommets du maillage
    QOpenGLBuffer m_ibo;                //! : index buffer (optionnel) qui stocke les index à utiliser pour le rendu
    QOpenGLVertexArrayObject m_vao;     //! : vertex array object (obligatoire)
    QOpenGLShaderProgram m_program;     //! : shader program (obligatoire)

    QVector3D m_position;               //! : position de l'objet dans la scène
    QQuaternion m_orientation;          //! : attitude de l'objet dans la scène

    int m_numVertices;                  //! : nombre de sommets de l'objet (lecture seule)
    int m_numTriangles;                 //! : nombre de triangles de l'objet (lecture seule)
    int m_numIndices;                   //! : nombre d'index (0 si on n'utilise pas l'ibo)

    RVCamera *m_camera;                 //! : pointeur sur la caméra utilisée pour le rendu.
};
```

Le constructeur de `RVBody` appelle le constructeur de la classe parent, crée le vertex buffer et l'index buffer et initialise toutes les variables membres :

```
RVBody::RVBody()
    :QOpenGLFunctions(),
      m_vao(), m_program(),
      m_position(), m_orientation()
{
    m_vbo = QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
    m_ibo = QOpenGLBuffer(QOpenGLBuffer::IndexBuffer);
    m_vbo.create();
    m_ibo.create();
    m_vao.create();
    m_program.create();

    m_numVertices = 0;
    m_numTriangles = 0;
    m_numIndices = 0;

    m_VSFileName = "NO VS FILE";
    m_FSFileName = "NO FS FILE";
}
```

Destructeur

```
RVBody::~RVBody()
{
    m_program.removeAllShaders();
}
```

Méthode `initialize()` doit être appelée dans le constructeur de la classe fille après avoir renseigné les noms des fichiers des programmes de shader :

```
void RVBody::initialize()
{
    initializeShader();
    initializeBuffer();         //à écrire dans la classe fille
    initializeVAO();            //à écrire dans la classe fille
}
```

Méthode `initializeShader()`

```
void RVBody::initializeShader()
{
    initializeOpenGLFunctions();

    bool resultat;

    m_program.bind();
    //Vertex Shader
    resultat = m_program.addShaderFromSourceFile(QOpenGLShader::Vertex, m_VSFileName);
    if (!resultat)     {
        QMessageBox msg;
        msg.setWindowTitle("Vertex shader");
        msg.setText(m_program.log());
        msg.exec();
        exit(EXIT_FAILURE);
    }

    //Fragment Shader
    resultat = m_program.addShaderFromSourceFile(QOpenGLShader::Fragment, m_FSFileName);
    if (!resultat)     {
        QMessageBox msg;
        msg.setWindowTitle("Fragment shader");
        msg.setText(m_program.log());
        msg.exec();
        exit(EXIT_FAILURE);
    }

    //Link
    resultat = m_program.link();
    if (!resultat)     {
        QMessageBox msg;
        msg.setWindowTitle("Link du shader program");
        msg.setText(m_program.log());
        msg.exec();
        exit(EXIT_FAILURE);
    }

    //Libération
    m_program.release();
}
```

Méthodes pour déplacer l'objet 3D

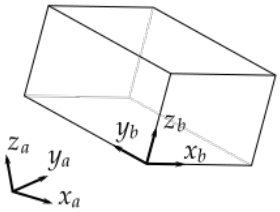
```
void RVBody::rotate(float angle, QVector3D axis)
{
    m_orientation =  QQuaternion::fromAxisAndAngle(axis, angle) * m_orientation ;
}

void RVBody::translate(QVector3D position)
{
    m_position += position;
}

void RVBody::setOrientation(float yaw, float pitch, float roll)
{
    m_orientation = QQuaternion::fromEulerAngles(pitch, yaw, roll);
}
```

Position et Orientation

Un corps rigide, c'est à dire un corps qui ne se déforme pas et dont ses différents composants ont toujours la même distance entre eux, est défini par 6 grandeurs réelles : on dit qu'il a 6 degrés de liberté. Ces 6 degrés de liberté sont d'une part les 3 coordonnées du centre de gravité du solide, et puis 3 réels qui définissent l'orientation de l'objet dans l'espace.
De même que spécifier la position du centre de gravité du solide correspond à spécifier la *translation* qui amène le centre de gravité de sa position originale (au repos) à la position voulue, spécifier une orientation revient à définir la rotation qui réoriente le solide de son orientation de départ à son orientation souhaitée. Le couple translation plus rotation compose la matrice 4x4 que nous appellons `modelMatrix`. Je vous invite à lire le cours sur les coordonnées homogènes pour comprendre le détail de cette opération.



Comme on peut le voir dans le cours sur les rotations dans l'espace, il y a plusieurs façons de définir une rotation :

- par une matrice 3x3 orthogonale de déterminant +1. Autrement dit un élément de SO(3), le groupe spécial orthogonal.
- par son axe et son angle. Cela fait un total de 4 réels mais comme en fait l'axe est un vecteur unitaire, l'une des coordonnées peut être déduite des 2 autres.
- par 3 angles d'Euler correspondant à l'écriture de cette rotation comme une composition de trois rotations autour des axes coordonnées. Par exemple les angles de roulis, tangage et lacet.
- par un quaternion unitaire.

Chacune de ces représentations a ses avantages et inconvénients (voir le cours sur les rotations) mais nous allons choisir la représentation par quaternion (ici le cours sur les quaternions) qui donne le plus de souplesse. De plus la classe `QQuaternion` de Qt, offre énormément de méthodes pour passer d'une représentation à une autre.
Ainsi la méthode `modelMatrix()` qui renvoie la matrice de placement s'écrit simplement (où je rappelle que `m_position` est un `QVector3D` et `m_orientation` est un `QQuaternion`) :

```
QMatrix4x4 RVBody::modelMatrix()
{
    QMatrix4x4 model;
    model.translate(m_position);
    model.rotate(m_orientation);
    return model;
}
```

Ainsi si l'on veut ultérieurement modifier la position et l'orientation du solide, qui se rajouteraient à la position et l'orientation active, on a deux méthodes `translate()` et `rotate()` dont le code est :

```
void RVBody::translate(QVector3D position)
{
    m_position += position;
}

void RVBody::rotate(float angle, QVector3D axis)
{
    m_orientation =  QQuaternion::fromAxisAndAngle(axis, angle) * m_orientation ;
}
```

Enfin si l'on veut spécifier l'orienttaion de l'objet avec les angles roulis (*roll*), tangage (*pitch*) et lacet (*yaw*) on utilise la méthode `setOrientation(float yaw, float pitch, float roll)` qui est

```
void RVBody::setOrientation(float yaw, float pitch, float roll)
{
    m_orientation = QQuaternion::fromEulerAngles(pitch, yaw, roll);
}
```

Classe RVPyramid

Pour tester notre classe `RVBody` nous allons faire une classe fille, toute simple, avec 4 points formant une pyramide. Il n'y a qu'à implémenter dans la classe `RVPyramid` , le constructeur et les trois méthodes virtuelles pures de `RVBody` :

```
class RVPyramid : public RVBody
{
public:
    RVPyramid();
    void draw() override;

protected:
    void initializeBuffer() override;
    void initializeVAO() override;
};
```

- Dans le constructeur, après avoir appelé le constructeur de a classe parent, on spécifie les noms des fichiers des programmes de shader. Ce sera les mêmes que ceux utilisés dans le tuto n°1

```
    RVPyramid::RVPyramid()
    :RVBody()
{
    //Je définis ici les shaders à utiliser dans RVPyramid
    m_VSFileName = ":/shaders/VS_simple.vsh";
    m_FSFileName = ":/shaders/FS_simple.fsh";
}
```

- Dans la méthode `initializeBuffer()` on doit :
  - Définir les 4 points A, B, C, D et les 4 couleurs :

```
//Définition de 4 points
QVector3D A(0, 0, 1);
QVector3D B(0.85f, 0, -0.5f);
QVector3D C(-0.85f, 0, -0.5f);
QVector3D D(0, 1, 0);

//Définition de 4 couleurs
QVector3D rouge(1, 0, 0);
QVector3D vert(0, 1, 0);
QVector3D bleu(0, 0, 1);
QVector3D blanc(1, 1, 1);
```

  - créer et remplir le vertex buffer object `m_vbo` avec les points et les couleurs

```
//Initialisation du Vertex Buffer Object
//On prépare le tableau des données
QVector3D vertexData[] = {
    A, B, C, D,
    rouge, vert, bleu, blanc
};
//Lien du VBO avec le contexte de rendu OpenG
m_vbo.bind();
//Allocation des données dans le VBO
m_vbo.allocate(vertexData, sizeof (vertexData));
m_vbo.setUsagePattern(QOpenGLBuffer::StaticDraw);
//Libération du VBO
m_vbo.release();
```

- créer et remplir l'index buffer object `m_ibo` car on va utiliser un rendu indexé pour faire 4 faces triangulaires avec les 4 sommets A, B, C, D

```
//Initialisation de l'Index Buffer Object
uint indexData[] = {
    0, 1, 3,
    0, 3, 2,
    0, 2, 1,
    1, 2, 3
};

//Lien du IBO avec le contexte de rendu OpenG
m_ibo.bind();
//Allocation des données dans le IBO
m_ibo.allocate(indexData, sizeof (indexData));
m_ibo.setUsagePattern(QOpenGLBuffer::StaticDraw);
//Libération du VBO
m_ibo.release();
```

- définir les entiers qui décrivent le maillage utilisé :

```
m_numVertices = 4;
m_numTriangles = 4;
m_numIndices = 12;
```

- dans `initializeVAO()`, il faut créer l'array buffer object `m_vao` qui fasse le lien entre les variables de type attribut du programme de shader et les éléments du vertex buffer.

```
void RVPyramid::initializeVAO()
{
    //Initialisation du Vertex Array Object
    m_program.bind();
    m_vao.bind();
    m_vbo.bind();
    m_ibo.bind();

    //Définition des attributs
    m_program.setAttributeBuffer("rv_Position", GL_FLOAT, 0, 3);
    m_program.enableAttributeArray("rv_Position");

    m_program.setAttributeBuffer("rv_Color", GL_FLOAT, sizeof(QVector3D)*m_numVertices, 3);
    m_program.enableAttributeArray("rv_Color");

    //Libération
    m_vao.release();
    m_program.release();
}
```

- Dans la méthode `draw()` il faut :

- faire des réglages spécifiques
- lier le programme et l'array buffer au contexte de rendu
- spécifier la valeur de la variable uniforme `u_ModelViewProjectionMatrix` du shader en utilisant le produit de la matrice projetcion de la camera, par la matrice vue de la caméra par la matrice model du RVBody lui-même.
- lancer la commande de rendu indexé
- libérer le vao et le programme.

```
void RVPyramid::draw()
{
    glCullFace(GL_BACK);
    glFrontFace(GL_CCW);
    //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

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

    //Définition de la variable uniforme
    QMatrix4x4 matrix;
    matrix = m_camera->projectionMatrix() * m_camera->viewMatrix() * this->modelMatrix();
    m_program.setUniformValue("u_ModelViewProjectionMatrix", matrix);

    //Commande de rendu (indexé)
    glDrawElements(GL_TRIANGLES, m_numIndices, GL_UNSIGNED_INT, nullptr);

    m_vao.release();
    m_program.release();
}
```

Nettoyage de RVWidget

Pour terminer le tutoriel n°2 et pour pouvoir tester nos classes , on doit nettoyer la classe `RVWidget` par rapport au TP n°1.
Ce qu'il faut enlever :

- plus de variables membres pour le vertex buffer, ni vertex array, ni programme de shader car chaque objet 3D aura ses propres ;
- plus de méthode `initializeShader()` et `initializeBuffer()` pour les mêmes raisons.

Ce qu'il faut ajouter :

- un pointeur `m_body` sur un objet `RVBody`
- un pointeur `m_camera` sur un objet `RVCamera`
- des *include* pour ces classes et leurs classes filles
- dans la méthode `initializeGL()`, au lieu d'appeler les deux méthodes qui n'existent plus :
  - on initialise `m_camera` à partir de la classe `RVCamera`
  - on initialise `m_body` à partir de la classe `RVPyramid`
  - on spécifie la variable camera de `m_body` avec `m_camera`
  - on spécifie la position de `m_body`
  - on appelle la méthode initialize() de `m_body`
- Remarquez que ces étapes sont les étapes classiques que l'on répètera pour chaque objet que l'on ajoutera au widget.

```
void RVWidget::initializeGL()
{
    initializeOpenGLFunctions();

    glClearColor(0.0f, 0.566f, 0.867f, 1.0f);

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    //glDisable(GL_CULL_FACE);

    m_camera = new RVCamera();
    m_body = new RVPyramid();
    m_body->setCamera(m_camera);
    m_body->setPosition(QVector3D(0, -0.5f, -4));
    m_body->initialize();
    connect(m_timer, SIGNAL(timeout()), this, SLOT(update()));
    m_timer->start(10);
}
```

- dans la méthode `paintGL()` tout est beaucoup plus simple puisqu'il suffit d'appeler la méthode `draw()` de `m_body`.

```
void RVWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    m_body->draw();
}
```

- Lorsque l'on veut modifier la position de l'objet ou son orientation (par exemple dans `update()` ou dans `mouseMoveEvent()` il faut maintenant appeler une méthode de `m_body`. Par exemple sa méthode `rotate()`.
- De même, lorsque l'on veut modifier une caractéristique de la camera, par exemple dans le slot `changeFov()` il faut appeler la méthode correspondante de `m_camera`.

```
void RVWidget::changeFov(int newFov)
{
    m_camera->setFov(newFov);
    this->update();
}
```

- dans la méthode `resizeGL()` il faut juste passer le nouveau rapport w/h (attention à la division de deux int) à `m_camera` via sa méthode `setAspect()` pour que la projection de la caméra tienne compte du changement de format du widget (sinon il y aura une déformation de l'image).

Ce qui ne change pas :

- les slots et la gestion de la souris qui restent la prérogative de cette classe en tant que classe contrôleur.
- le timer.