TP3

Site: lms.univ-cotedazur.fr
Cours: Realite virtuelle - EIMAD919
Livre: TP3

Imprimé par: Theo bonnet
Date: vendredi 28 février 2020, 14:54

Table des matières

1. Cube texturé

RVTexCube

Objectif

Le point de départ du projet est l'état final du Tuto3 qui est ici.
La première étape du TP3 consiste à transformer le cube coloré en un cube coloré et texturé. Comme texture je vous propose les images suivantes :

que l'on va copier sur les 6 faces.

Préparation de la classe

1. copier le fichier d'en-tête `rvcube.h` en `rvtexcube.h` avec `Duplicate File...` ;
2. faire la même chose pour `rvcube.cpp` ;
3. changer le nom dans le `#ifndef` dans `rvtexcube.h` ;
4. modifier l'include de `rvtexcube.h` au début de `rvtexcube.cpp` ;
5. enfin on utilise `Refactor-Renommer le symbole sous le curseur` pour remplacer dans ces deux fichiers (Attention : seulement dans ces deux fichiers pas dans les autres!) toute occurence de `RVCube` en `RVTexCube`.

Modification de `RVTexCube::initializeBuffer()`

Le vertex buffer object de `RVCube` ne suivait pas la règle utilisée dans le tuto 3 basé sur le struct `RVVertex` qui regroupait (dans un même objet position et coordonnées texture). Plutôt on a placé *toutes* les position s au début du VBO, puis, après toutes les couleurs. Nous allons continuer dans cette optique en plaçant maintenant, *après les couleurs* les coordonnées textures. Cela nous oblige à utiliser des QVector3D pour les coorodnnées textures au lieu des QVector2D mais cela nous sera utilie dans la suite.
Donc dans `RVTexCube::initializeBuffer()` :

1. On ajoute 4 `RVVector3D` appelés `SW, SE, NE, NW` (pour Sud-Ouest, Sud-Est, etc..) qui seront les coordonnées texture des 4 coins de chaque face ;
2. A la fin de `vertexData`, pour chacune des 6 faces, on met (dans le bon ordre), ces 4 `RVVector3D` ;

Modification de `RVTexCube::initializeVAO()`

Dans le VAO, il faut faire le lien entre les nouvelles données du VBO avec des nouveaux attributs du *vertex shader* (que nous devons encore écrire) :

1. le nouvel attribut se nommera `"rv_TexCoord"` et sera toujours composé de 3 `GL_Float`;
2. bien indiquer à quel emplacement (dans le VBO) commence les données correspondant à cet attribut.

Modification de `RVTexCube::draw()`

Pour modifier la méthode `RVTexCube::draw()` on s'inspire largement de ce qui est fait dans `RVPlane` .

Modification des shaders

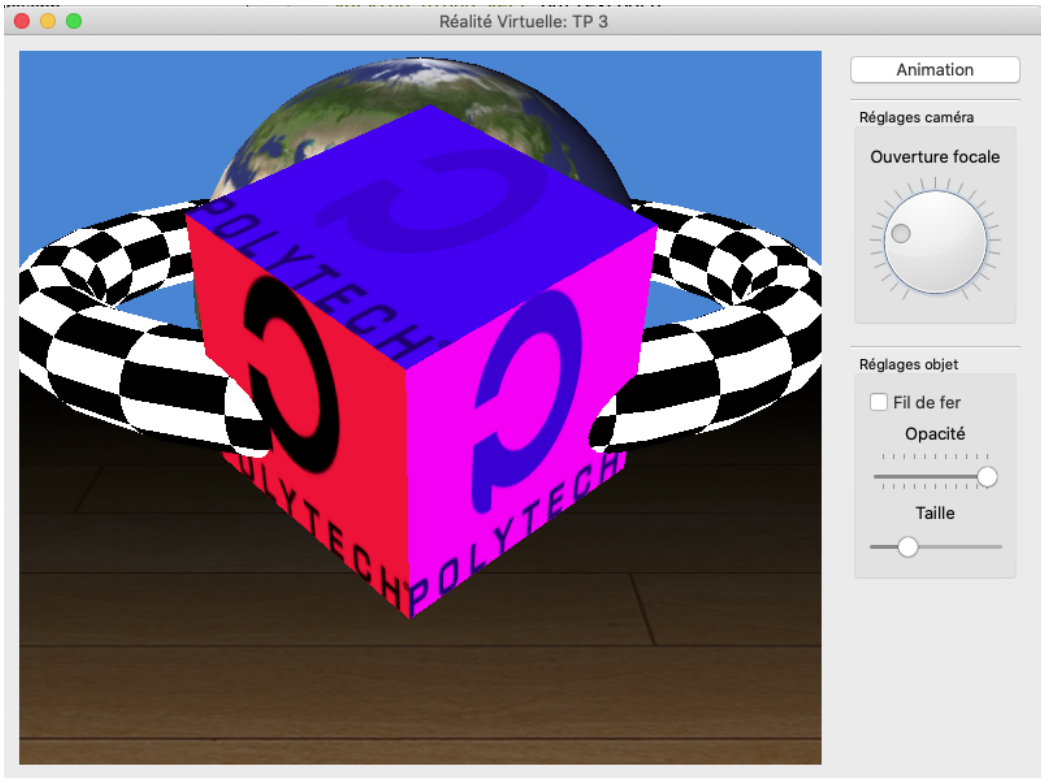
1. Pour le **vertex shader**, on se base sur `VS_simple.vsh` que l'on duplique en `VS_cube_texture.vsh` avec en plus
   - un attribut `rv_TexCoord` (vec3) ;
   - un *varying* `outTexCoord` (vec2) ;
   - `outTexCoord` se définit à partir de l'attribut en prenant ses deux premières cooronnées (utilise `.st`
2. Pour le **fragment shader**, on se base sur `FS_simple_texture.fsh` que l'on duplique en `FS_cube_texture.fsh` ; comme pour le damier (ou la planète terre) on a deux couleurs à mélanger : celle issue du vertex shader dans `outColor` (ici ce sont les 6

couleurs des 6 faces) et celle issue de la texture. Aulieu de les mélanger avec `mix` on va les **multiplier** (dans GLSL le produit de deux vecteurs se fait terme à terme) pour obtenir `gl_FragColor`. Pour la transparence, en revanche, on prend seulement celle issue du vertex shader et qui peut être modifiée via `m_opacity`.

3. Dans le constructeur de `RVTexCube`, il faut déclarer les nouveaux shaders.

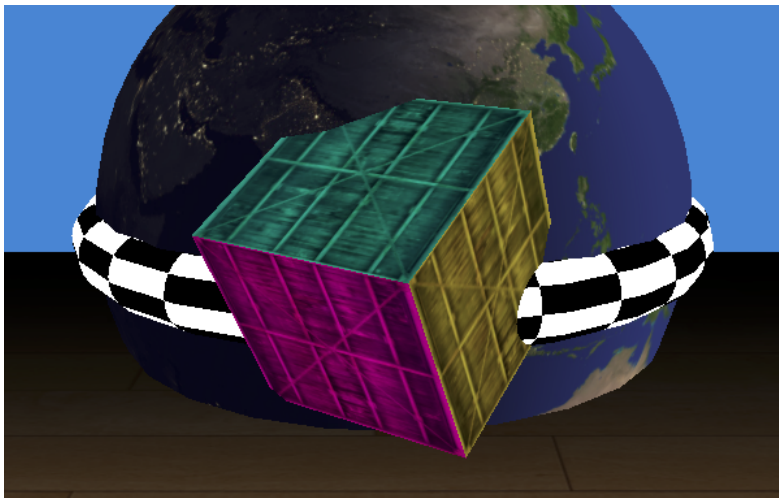
Mise en place dans RVWidget

1. On ajoute l'include de `rvtexcube.h` dans `rvwidget.h`
2. On change le type de `m_body` à `RVTexCube` dans `RVWidget::initializeGL()`.
3. On déplace le cube pour qu'il ne soit plus sous la sphère (ou on déplace la sphère et le tore). On peut aussi agrandir le cube avec `setScale()`
4. On ajoute la texture en ressource et on la passe au cube. La texture du conatiner est plus jolie mais la texture avec le logo Polytech permet de voir s'il y a des problèmes de collage à l'envers de la texture (et dans ce cas il faut modifier soit l'ordre des sommets dans la face du VBO, soit l'ordre des coordonnées textures).



Remarquez que avec `FS_simple_texture.fsh` on aurait plutôt ceci :

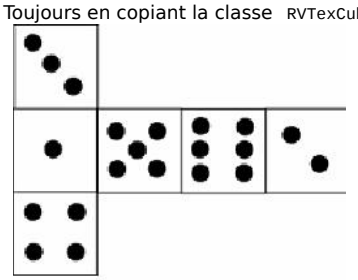
Et avec le container (et les couleurs) :



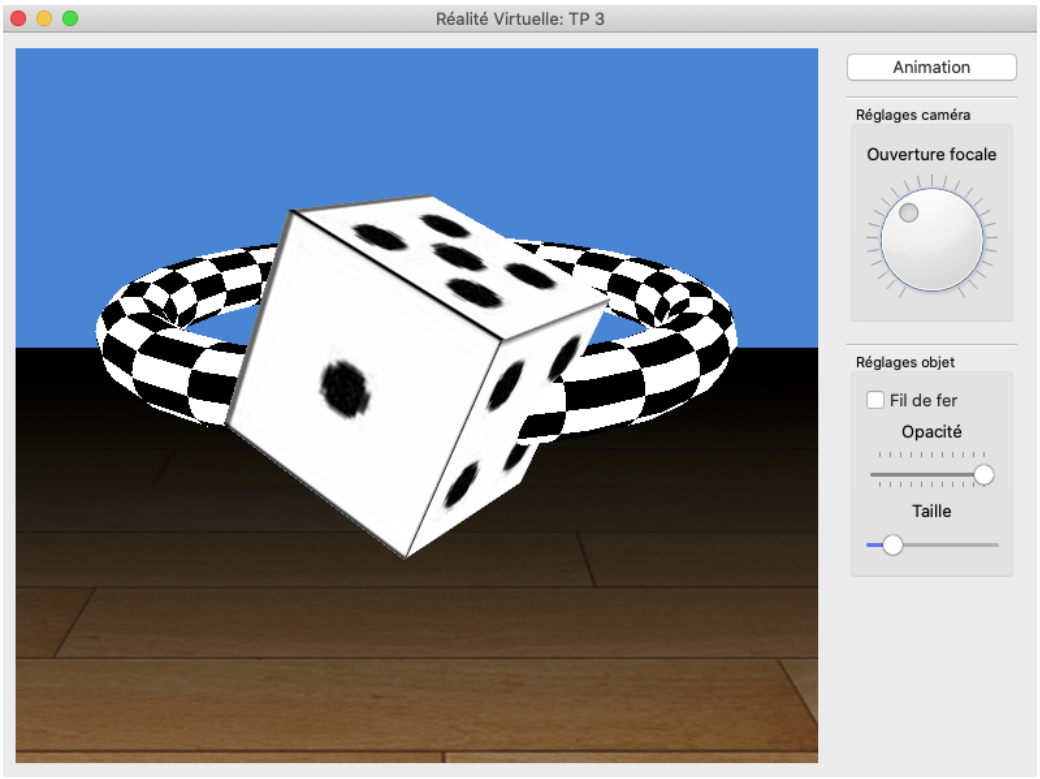
2. Dé à jouer

RVDice

Toujours en copiant la classe `RVTexCube` en `RVDice` , on veut maintenant appliquer la texture `dice_texture.jpg`



qui représente les six faces d'un dé à jouer. Le code est exactement le même que dans la question précédentes sauf que les coordonnées textures des points du dé doivent être modifiées (par exemple pour avoir sur la face avant le 3 les coordonnées textures seront entre 0.75 et 1 pour s et entre 0 et 0.25 pour t). Vous pouvez utiliser `FS_simple_texture.fsh` pour avoir un joli dé blanc comme ci-dessous :



3. RVScene

Description de la classe RVScene

`RVScene` est un conteneur, hérite de `QList<RVBody*>`, c'est à dire que c'est une liste de pointeurs sur des objets 3D. Il hérite du coup des méthodes de `QList` que vous trouverez ici.
Il possède en outre une variable membre *protected* `m_camera` qui est le pointeur sur l'instance de `RVCamera` utilisée dans la scène.
Ses méthodes sont :

- `void setCamera(RVCamera* camera)` qui change la valeur de `m_camera` et change la caméra de tous les objets 3D de la liste (on peut itérer dans une `QList` avec la syntaxe

```
foreach (RVBody* body, *this) {
   ...
}
```

- `void translate(QVector3D vec)` pour appliquer la translation à tous les objets de la liste
- `void rotate(float, QVector3D)` pour appliquer la rotation à tous les objets de la liste
- `void draw()` pour appeler la méthode `draw()` de tous les objets de la list.

Utilisation de la classe RVScene

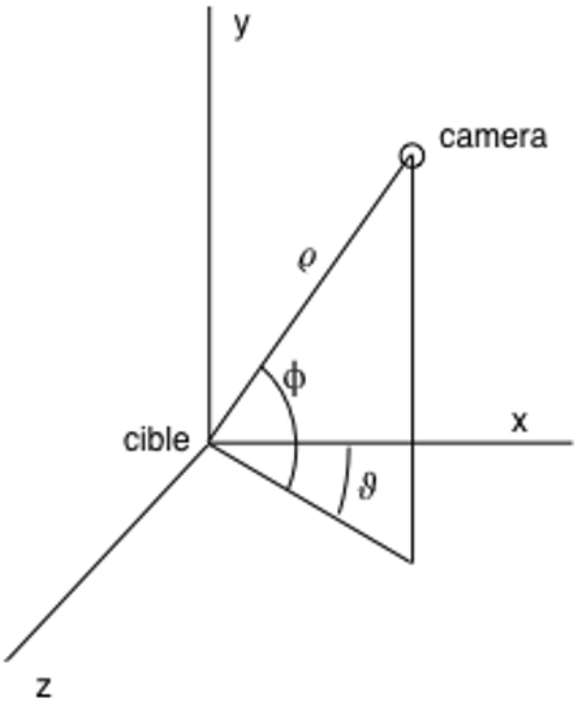
- Ajouter `m_scene` à `RVWidget`
- Ajouter les objets crées à `m_scene`
- Lui passer la caméra
- Utiliser `m_scene` dans paintGL et dans update

4. Caméra Sphérique

RVSphericalCamera

Acteullement notre caméra n'est pas très satisfaisante car elle ne permet pas vraiment de tourner autour des objets. Pour améliorer cela, vous aller créer une sous-classe de `RVCamera` qui implémente une caméra perfectionnée : ce sera une *caméra sphérique*, c'est -à-dire une caméra dont la position évolue sur une sphère centrée sur sa cible. C'est le type de caméra utilisé souvent dans le jeux vidéos où l'on suit un personnage dans son monde en gravitant autour de lui (par exemple WoW ou Zelda).

Votre classe s'apellera `RVSphericalCamera` qui hérite de `RVCamera` et contient trois variables membres de type *float* `m_phi, m_theta, m_rho` qui donnent les *coordonnées sphériques* de la caméra sur la sphère centrée sur la cible (voir dessin ci-dessous):



- Ajouter une méthode privée void `update_position()` qui met à jour la variable membre `m_position` de `RVCamera` en fonction de la position de la cible et des trois coordonnées sphériques (utiliser les fonctions trigonométriques qCos et qSin de QtMath qui calculent avec des floats et pas des doubles) :

$$\begin{cases} \text{cible} & = & x_{\text{cible}} + \rho\cos(\phi)\cos(\theta) \\ y_{\text{cible}} & = & y_{\text{cible}} + \rho\sin(\phi) \\ z_{\text{cible}} & = & z_{\text{cible}} + \rho\cos(\phi)\sin(\theta) \end{cases}$$

  Rappel : contrairement à l'angle utilisé pour l'argument `m_fov` (qui est en degrés), ici les deux angles sont en radians.
- Dans le constructeur donner comme les valeurs par défaut (m_rho à 10 et les deux angles à 0) et appeler `update_position()`.

- Ajouter les accesseurs et mutateurs pour les attributs de `RVSphericalCamera` . Dans le code de chaque mutateur ajouter un appel à update_position() après la modification de l'attribut. De plus il faut empêcher ρ d'être nul ou négatif et la variable m_phi doit être $-\pi/2 < \phi < \pi/2$.

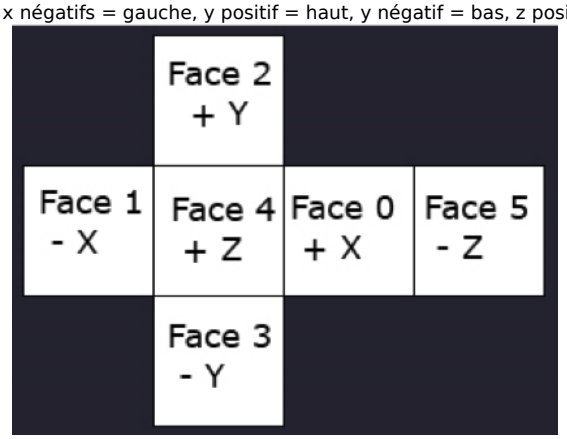
Pour tester le fonctionnement de la nouvelle caméra :

- Dans `rvwidget.h` ajouter l'include de la nouvelle classe
- Dans `RVWidget::initializeGL()`, faire en sorte que `m_camera` soit cette fois une instance de `RVSphericalCamera` . Changer ensuite ses paramètres pour que sa cible soit l'origine de la scèbne et placez-y un cibe (ou un dé) pour fixer la position.
- Dans `RVWidget::mouseMoveEvent`, on veut maintenant que les mouvements de la souris changenet les paramètres ϕ et $theta$ de la caméra sphérique ; pour faire cela, il faut tenir compte d'une part que `angleX, angleY` doivent être petits (puisque ce sont des angles en radian) et qu'il doivent servir à incrémenter les valeurs existantes de (\phi\) et $theta$ respectivement.

5. SkyBox

RVSkyBox

Une *skybox* est une boite (un cube) englobant la scène sur laquelle est plaquée une *texture cubique* (*cube map* en anglais) ; c'est à dire 6 images (des photos ou des images synthétiques) représentant un décor dans chacune des 6 directions (x positifs = droite, x négatifs = gauche, y positif = haut, y négatif = bas, z positif = devant, z négatif = derrière).



Sites où l'on trouve de beaux bitmaps pour construire des skyboxes :

- A partir de photos : http://www.humus.name/index.php?page=Textures
- A partir d'images de synthèse : http://www.custommapmakers.org/skyboxes.php

OpenGL permet de réunir les 6 images en une seule texture (avec le flag GL_TEXTURE_CUBE_MAP au lieu de GL_TEXTURE_2D) qui est passée au fragment shader et GLSL utilise la fonction `textureCube` qui lit dans cette texture cubique avec des coordonnées textures (un vec3 cette fois) qui est un vecteur qui donne la direction issue du centre du cube. Ce vecteur n'a pas besoin d'être normalisé.

Mise en place

Création d'une classe `RVSkyBox` qui hérite de `RVBody` dans laquelle on va devoir surcharger les méthodes `initializeBuffer(), initializeVAO() et draw()` plus une nouvelle méthode `setCubeTexture()` qui prend comme argument 6 `QString` correspondant au nom des 6 bitmaps à utiliser pour construire la texture cubique.

```
void setCubeTexture(QString leftImage, QString rightImage,
                    QString frontImage, QString backImage,
                    QString topImage, QString bottomImage);
```

Mise en place de la texture cubique

La méthode `setCubeTexture(..)` est la plus critique ; on va toujours utiliser la variable membre `m_texture` définie dans `RVBody` mais cette fois quand on l'initialise on doit déclarer qu'on va créer une *cubemap*. Donc

```
m_texture = new QOpenGLTexture(QOpenGLTexture::TargetCubeMap);
m_texture.create();
```

1. il faut charger les fichiers des 6 images en mémoire dans 6 instances de `QImage` en utilisant les noms des fichiers passés en argument ; on en profite pour convertir le format de ces images à un format fixe : dans notre cas RGBA888 ce qui signifie que chaque pixel est codé sur 32 bits dont 8 pour le rouge, 8 pour le vert, 8 pour le bleu et 8 pour l'opacité.

   ```
   QImage posX = QImage(rightImage).convertToFormat(QImage::Format_RGBA8888);
   ```

2. on utilise une de ces 6 images pour définir la taille de `m_texture` avec sa méthode `setSize` en lui passant la largeur, la hauteur et la profondeur de `posX`. On suppose que les 5 autres images ont exactement la même taille ce qui est le cas pour les images utilisées pour définir des skyboxes.
3. on définit le format de `m_texture` avec sa méthode `setFormat` en lui passant `QOpenGLTexture::RGBA8_UNorm`
4. on appelle sa méthode `allocateStorage()` (sans arguments) pour que OpenGL (via la classe `QOpenGLTexture` de Qt) puisse allouer en mémoire vidéo l'espace suffisant pour stocker les pixels (c'est pour cela qu'il est nécessaire de définir ***avant*** la taille des 6 images et le format des pixels).
5. C'est la méthode `setData` qui va copier les pixels contenus dans les QImage pour construire la texture cubique OpenGL. Cette méthode devra être appelée 6 fois, une fois pour chaque face. Par exemple pour la face des x positifs la commande est :

   ```
   m_texture->setData(0, 0, QOpenGLTexture::CubeMapPositiveX,
                      QOpenGLTexture::RGBA,
                      QOpenGLTexture::UInt8,
                      posX.constBits(),
                      Q_NULLPTR);
   ```

   Notez que `posX.constBits()` représente l'adresse mémoire des pixels du `QImage` `posX`.
6. Il faut ensuite générer les niveaux de mipmap de `m_texture` avec `generateMipMaps()`.
7. Enfin, ifaut faire quelques réglages sur la façon dont se fait l'échantillonage de la texture et la façon dont elle traite les bords de la texture.

   ```
   m_texture->setWrapMode(QOpenGLTexture::ClampToEdge);
   m_texture->setMinificationFilter(QOpenGLTexture::LinearMipMapLinear);
   m_texture->setMagnificationFilter(QOpenGLTexture::Linear);
   ```

Autres méthodes

- **initializeBuffer()** : le buffer est simple car on n'a besoin que des coordonnées des 8 points (sans couleur et sans coordonnées texture) car on va utiliser les coordonnées des points du cube comme coordonnées textures. Il faut juste modifier A, B, C, D, E, F, G, H pour que leurs coordonnées soient maintenant entre -1 et 1 (au lieu de 0 et 1)
- **initializeVAO()** : légère modification par rapport au code de la classe `RVCube` car il y a un seul attribut dans le VBO qui est `rv_Position`.
- **draw()** : quasiment identique à la version du cube texturé sauf que les réglages de `m_texture` sont différents :

```
if (m_texture) {
    glEnable(GL_TEXTURE_CUBE_MAP);
    glEnable(GL_TEXTURE0);
    m_texture->bind();
}
```

Les shaders

On peut faire des copies des shaders utilisés pour le cube texturé avec les modifications suivantes :

- Pour le vertex shader **"VS_skybox_texture.vsh"**:
  - un seul attribut `rv_Position`
  - la seule variable uniforme est la matrice usuelle
  - la seule variable `varying` est `outTexCoord` qui cette fois est un `vec3` et qui est simplement définie dans le main par

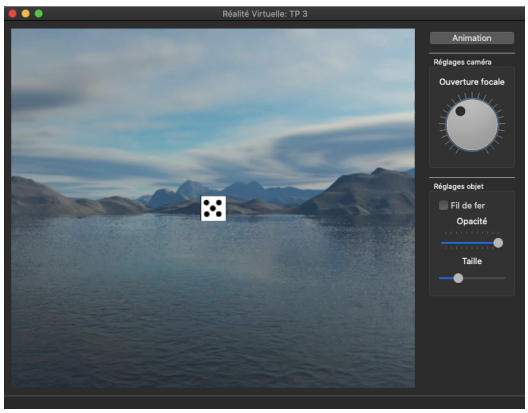
```
        outTexCoord = rv_Position;
```

- Pour le fragment shader **"FS_skybox_texture.fsh"**:
  - la variable `varying outTexCoord` est un `vec3`
  - le type de la variable uniforme `texture0` doit maintenant être `samplerCube`
  - on utilise la fonction `textureCube` au lieu de `texture2D` pour récupérer la couleur du fragment à partir de la texture.
  - et il n'y a plus de mélande de couleurs.

Enfin dans le constructeur de `RVSkyBox` il faut donner les bons noms des fichiers de shaders.

Test

Il faut récupérer un ensemble d'images de skybox sur internet : c'est souvent un fichier zip qui donne un dossier contenant les noms des 6 images avec des titres clairs (genre "posX.jpg" ou "left.jpg"). On place ce dossier dans le répertoire du projet. On les ajoute aux ressources du projet dans l'onglet "textures" avec un Clic-droit puis `Add Existing Directory...` .
Dans `RVWidget` , il faut ajouter une nouvelle variabe membre `m_skybox` pointeur sur `RVSkyBox` , qui sera initialisée dans `initializeGL()` et rendue dans `paintGL()` comme d'habitude : la skybox devra être placée à l'origine du repère de la scène et sa taille doit être très grande (pour englober tous les éléments de la scène). Il faudra peut être augmenter l'attribut zMax de la caméra.

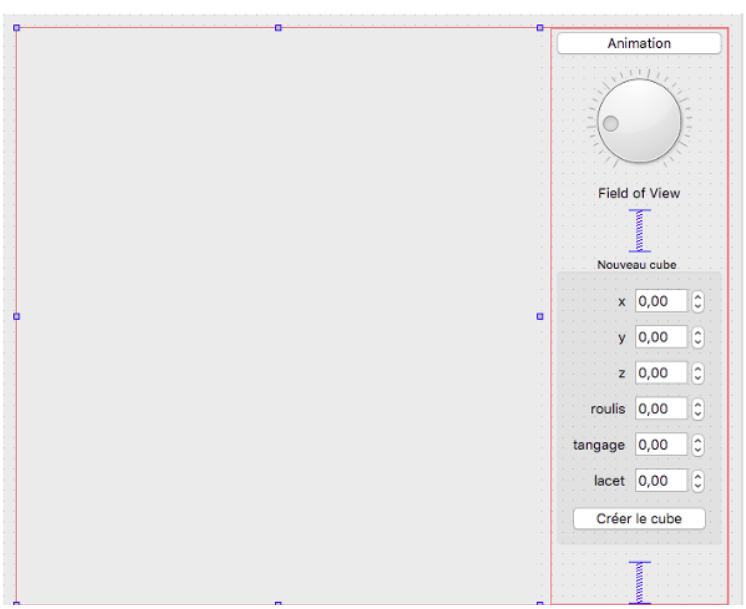


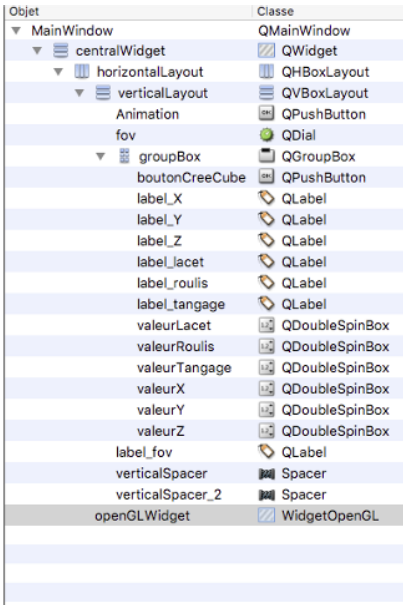
6. IHM

Vous devez modifier l'interface du programme pour pouvoir ajouter dynamiquement des cubes texturés (ou autre) dans la scène.

Instructions

1. On garde le bouton qui lance l'animation et le `QDial` qui modifie le *field of view* mais on enlève le *slider* qui modifie l'opacité.
2. Dans l'interface `mainwindow.ui` (voir ci desssous) :
   - On ajoute un `QGroupBox` sous le `QDial` dont le titre est "nouveau cube"
   - On choisit pour ce `groupBox` un *layout* de formulaire
   - Dans le *groupBox* on met à gauche 6 `QLabel` et à droite `QDoubleSpinBox` avec les intitulés ci-dessous pour choisir la position et l'orientation du cube à créer
   - A la fin on place un bouton `buttonCreateCube`





3. On associe à l'action `clicked()` de `buttonCreateCube` l'appel d'un *slot* `createCube()` de *MainWindow*. Pourquoi de `MainWindow` et pas de `RVWidget` (comme pour les autres contrôles) ? Car pour construire le cube il faut récupérer les informations des `QDoubleSpinBox` qui sont accessibles (via la variable membre `ui`) seulement dans `MainWindow` .
4. Ajouter le slot `createCube()` dans `MainWindow` :
   - Qui construit une instance de `RVTexCube`
   - Qui récupère via `ui` les valeurs des contrôles *valeurX*, *valeurY*, *valeurZ* puis appelle `setPosition`
   - Qui récupère *valeurRoulis*, etc.. et appelle `setOrientation`
   - Enfin appelle la méthode `addBody` de `RVWidget` en lui passant ce cube.
5. Ajouter une méthode publique `addBody(RVBody* obj)` dans `RVWidget`
   - Qui active le contexte de rendu OpenGL avec `this->makeCurrent()`
   - Qui appelle `obj->initialize()`
   - Qui définit `m_camera` comme étant la caméra à utiliser pour le rendu de l'objet 3D `obj`
   - Qui ajoute (append) l'objet `obj` à la pile des objets à rendre de `m_scene`

7. Bonus

Qulques idées (non exhaustives) pour compléter le TP3, certaines étant plus faciles que d'autres :

1. Pouvoir choisir dans l'IHM le type d'objet à ajouter
2. Prévoir un bouton pour un placement aléatoire
3. Pouvoir modifier via l'IHM certains des paramètres utilisés dans le TP : par exemple le nombre de cases du damier sur le tore, ou la taille du tore (petit rayon et/ou grand rayon)
4. Ajouter d'autres surfaces mathématiques à la scène
5. Pouvoir changer d'image de *skybox*
6. Pouvoir répéter la texture sur une face (par exemple le logo Polytech 3x3 sur chaque face du cube)
7. Pouvoir charger une texture en choisissant un bitmap dans un fichier via un menu (par exemple dans la *MenuBar* on peut ajouter un titre *File* comme sous-titre une `QAction loadTexture`. A cette action on associe un slot `triggered` qui crée dans MainWindow la méthode `void MainWindow::on_actionLoadTexture_triggered()`. Ici on utilise une boite de dialogue toute faite `QFileDialog` qui demande à l'utilisateur de choisir un fichier...)
8. Faire glisser une texture sur une face en utilisant une variable uniforme basée sur le temps (comme pour la planète terre) pour modifier une coordonnée texture. Ainsi on pourrait simuler sur le plan l'effet des titres Star Wars (voir ici).