TP1

Site: lms.univ-cotedazur.fr
Cours: Realite virtuelle - EIMAD919
Livre: TP1

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

Description

Premier TP noté à rendre.

Basé sur le tuto n°1

Table des matières

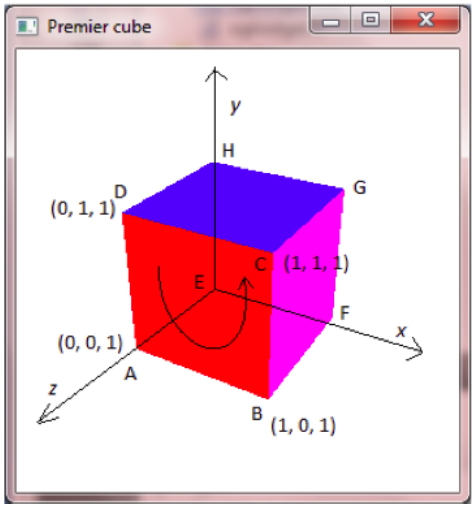
Cube coloré

Préparation du TP

Le point de départ est le projet du tut n°1 : copier et renommer le répertoire du projet et renommez le fichier `RVTuto1.pro` en `RVTP1.pro`. Puis lancez `QtCreator` sur ce projet puis modifier dans `main.cpp` le titre de la fenêtre. Compilez et vérifiez que tout fonctionne.

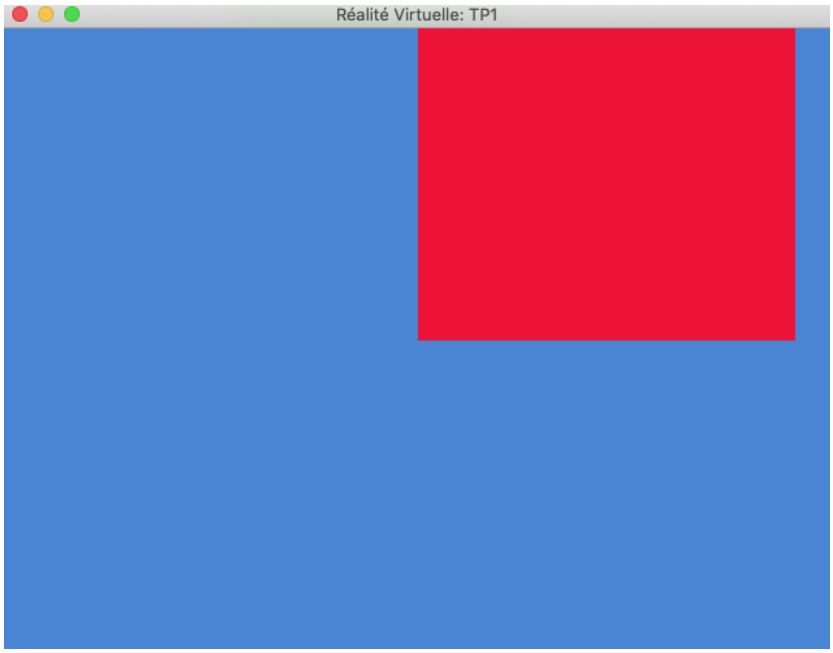
Constrution d'un cube

On veut remplacer les quatre sommets du tutoriel par un cube de coté 1 avec chaque face colorée d'une couleur différente. Alors qu'un cube a 8 sommets, nous allons devoir metre dans le vertex buffer 24 sommets car chaque sommet doit apparaître 3 fois, chaque fois porteur d'une couleur différente. Par exemple sur le dessin ci-dessous, on voit que le sommet C est rouge lorsqu'il appartient à la face ABCD, il est bleu lorsqu'il appartient à la face CGHD et magenta sur la face BFGC.



1. Dans la méthode `initializeGL()` de la classe `RVWidget` changer la couleur de fond de `glClearColor` en (0.0f, 0.566f, 0.867f) pour avoir un joli bleu Polytech !
2. Dans la méthode `initializeBuffer()` :
   - On définit 8 `QVector3D` appelés A, B, C, D, E, F, G avec les coordonnées données par le dessin ci-dessus;
   - On définit 6 `QVector3D` pour les 6 couleurs primaires;
   - Dans le tableau `vertexData` on met 6 fois 4 sommets pour chacune des 6 faces et puis 6 fois la même couleur répétée 4 fois pour que les 4 sommets d'une face aient la même couleur.
3. Dans la méthode `initializeShader()` la seule chose à modifier est lors de la définition de l'attribut `rv_Color` (définition du VAO). Le troisième argument de `setAttributeBuffer` représente la position où commence l'information de couleur dans le vertex buffer. Maintenant ce n'est plus `sizeof(QVector3D)*4` mais bien plus...
4. Dans la méthode `paintGL()` on doit maintenant dessiner 6 carrés, c'est à dire 6 éventails de deux triangles. Il faut donc dans une boucle utiliser 6 fois la commande `glDrawArray` avec la primitive `GL_TRIANGLE_FAN` mais en indiquant proprement pour chaque éventail quel est l'indice du premier vertex.

Si vous compilez le programme, au mieux vous verrez une seule face (rouge) mal centrée dans l'écran. Comme ceci :


Réalité Virtuelle: TP1

Placer le cube

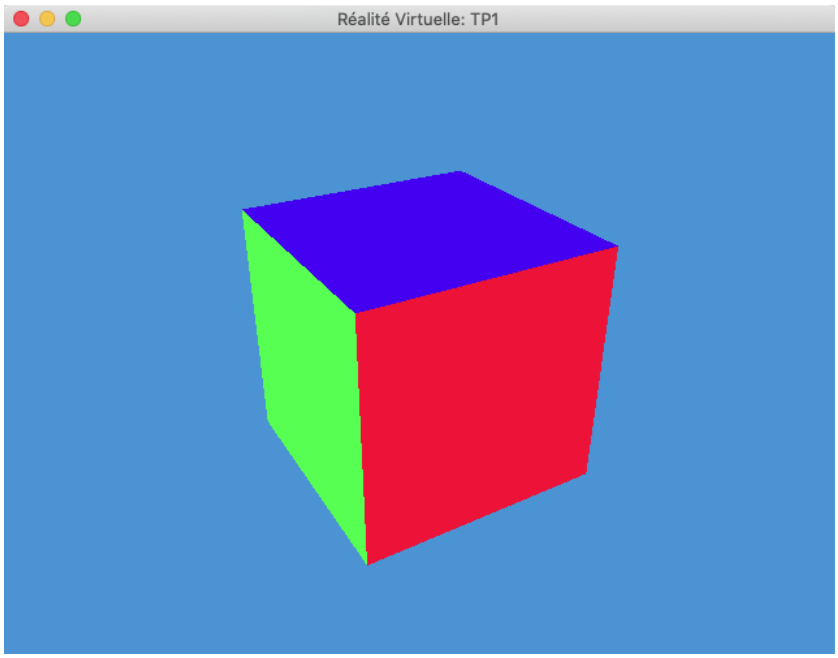
Pour avoir la sensation de voir un cube on va devoir modifier sa matrice de placement, la matrice `model`. Actuellement l'observateur est à l'origine (0, 0, 0), regarde dans la direction des z négatifs et le cube a subi une translation de vecteur (0, 0, -3). C'est donc normal qu'au centre du widget il y a le sommet A rouge.
Pour avoir une vue de 3/4 du cube on va faire une succession de tranformations :

1. en premier une translation T1 de vecteur (-0.5, -0.5 - 0.5) qui amènera le point E (qui était à l'origine) en (-0.5, -0.5 - 0.5) et donc le centre du cube sera maintenant à l'origine ;
2. ensuite une rotation R1 de 30° autour de l'axe vertical (qui est l'axe y) ;
3. ensuite une rotation R2 de 30° autour de l'axe horizontal (qui est l'axe x) ;
4. enfin la translation T2 de vecteur (0, 0, -2) qui éloigne le cube de l'observateur (pour qu'il puisse être vu).

Cette composition d'isométries affines correspond au produit de matrice 4x4 (voir le cours sur les coordonnées homogènes) qui est un produit non commutatif (c'est à dire que l'ordre compte !). Donc la transformation finale (la matrice model) sera (dans cet ordre)

$$model = T2 * R2 * R1 * T1$$

c'est à dire que la transformation qui arrive en dernier dans model est la première à être appliquée (et viceversa).



Animer le cube

Le framework Qt offre une classe `QTimer` qui représente un *timer* que l'on peut lancer et arrêter et à qui on peut demander d'émettre tous les n millièmes de seconde un *signal* qui s'appele `timeout()`. Qt met en place un mécanisme de *couplage faible* entre classes (souvent utilisé pour les composants graphiques) appelé `signal` et `slot` : on associe (on connecte) le signal d'une classe à une méthode d'une autre classe qui doit être identifiée comme étant un *slot* c'est à dire un connecteur (ou une prise). C'est sur ce principe que Qt met en place la programmation événementielle.
Ajouter à la classe `RVWidget` :

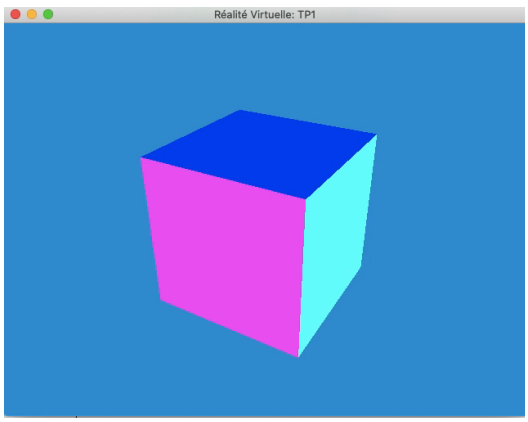
1. une variable membre de type `float` appelée `m_angleY` (initialisée à 0 dans le constructeur) qui représentera l'angle (variable) de la rotation autour de l'axe y dans la matrice `model`
2. une variable membre `m_timer` de type `QTimer*` (pointeur sur timer). Elle doit être initialisée (avec `new` car c'est un pointeur) dans le constructeur.
3. une méthode de type `private slots:` appelée `update()`. Dans le code de cette méthode on se contente d'incrémenter l'angle `m_angleY` de 5° puis d'appeler la méthode `update()` de la classe parent avec

   ```
   QOpenGLWidget::update();
   ```

   qui force le réaffichage du contenu dans la fenêtre.
4. on doit connecter le timer au slot. On le fait à la fin de la méthode `initializeGL()` avec la commande

   ```
   connect(m_timer, SIGNAL(timeout()), this, SLOT(update()));
   ```

5. toujours dans `initializeGL()` on peut lancer le timer avec la méthode `start` auquel on passe l'intervalle de temps (en millisecondes) entre deux `timeout` (50 ms est une bonne valeur).



Déplacer le cube à la souris

On aimerait déplacer le cube manuellement à la souris. Donc on va surcharger (override) dans la classe `RVWidget` deux méthodes de tous les widget qui réagisent aux actions claviers :

- `void mousePressEvent(QMouseEvent* event)` qui est appelée dès que l'on appuie sur le bouton de la souris ;
- `void mouseMoveEvent(QMouseEvent* event)` qui est appelée durant le mouvement de la souris (avec un bouton appuyé) ;
- `QMouseEvent` est une classe Qt (dont il faut rajouter l'include) qui contient des informations sur la souris : par exemple `pos()` donne accès à un `QPoint` qui contient l'abscisse et l'ordonnée du curseur de la souris.

On a cette fois besoin de deux angles `m_angleX` et `m_angleY` (que l'on a dèjà) pour les rotations autour de l'axe x (horizontal) et y (vertical) dans la matrice `model`. On désactive la rotation automatique en ne faisant pas démarrer le timer.
Le principe est le suivant :

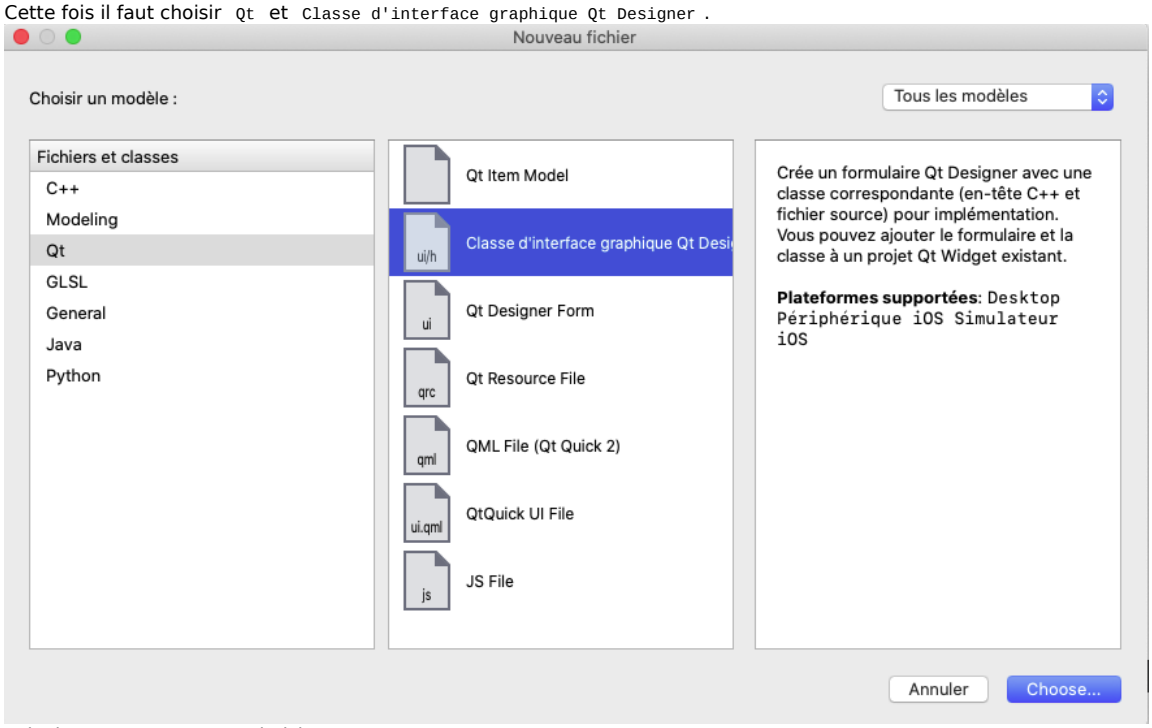
1. dans `mousePressEvent` on enregistre la position de la souris dans un variable membre `m_oldPos` de type `QPoint`
2. dans `mouseMoveEvent`
   - on calcule deux `float dx` et `dy` qui représentent le déplacement du curseur de la souris en x et en y (la nouvelle position par rapport à l'ancienne) relatif à la taille de la fenêtre (c'est-à-dire que l'on divise par `width()` et `height()`) ;
   - ensuite on incrémente `m_angleX` et `m_angleY` respectivement de `dy*180` et `dx*180` ;
   - enfin on met à jour la valeur de `m_oldPos` puis on appelle `QOpenGLWidget::update` pour forcer le réaffichage.

Intégrer RVWidget dans une IHM complexe

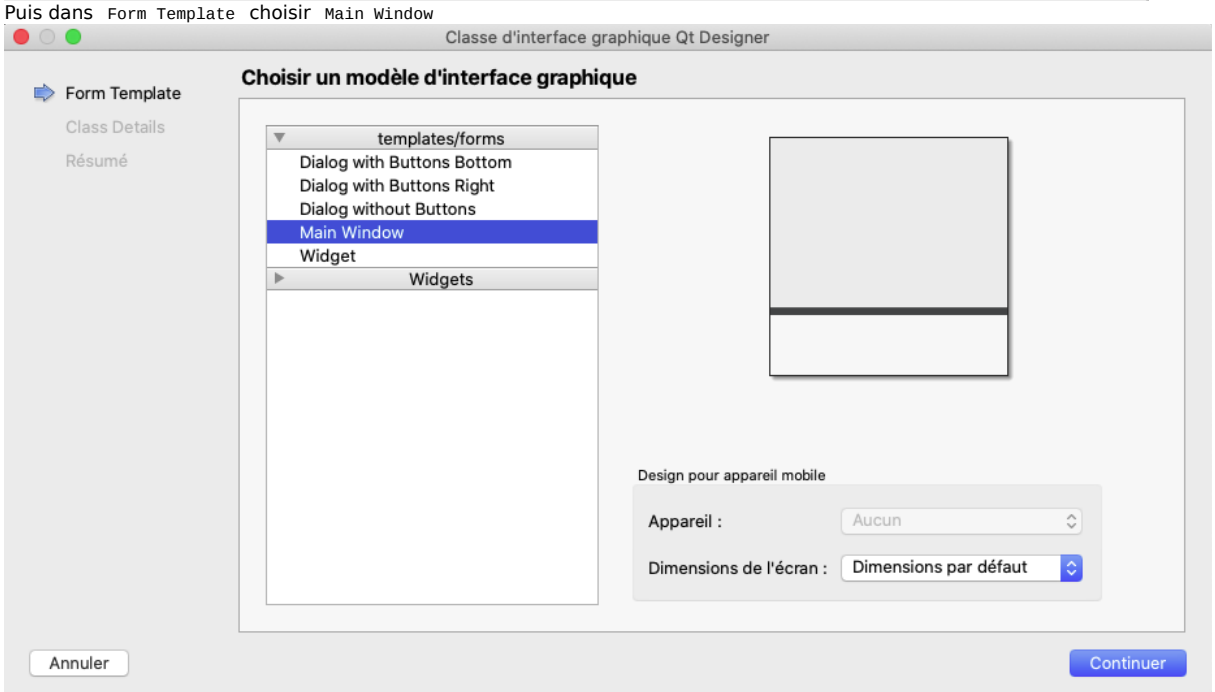
L'avantage d'avoir utilisé un widget pour encapsuler notre contexte de rendu OpenGL c'est que l'on peut mélanger dans une même interface des widgets classiques (boutons, sliders, etc..) avec notre widget 3D. Et Qt permet grâce aux signaux et aux slots de faire en sorte que la communication entre les différents éléments soit très simple.

Création d'une fenêtre principale

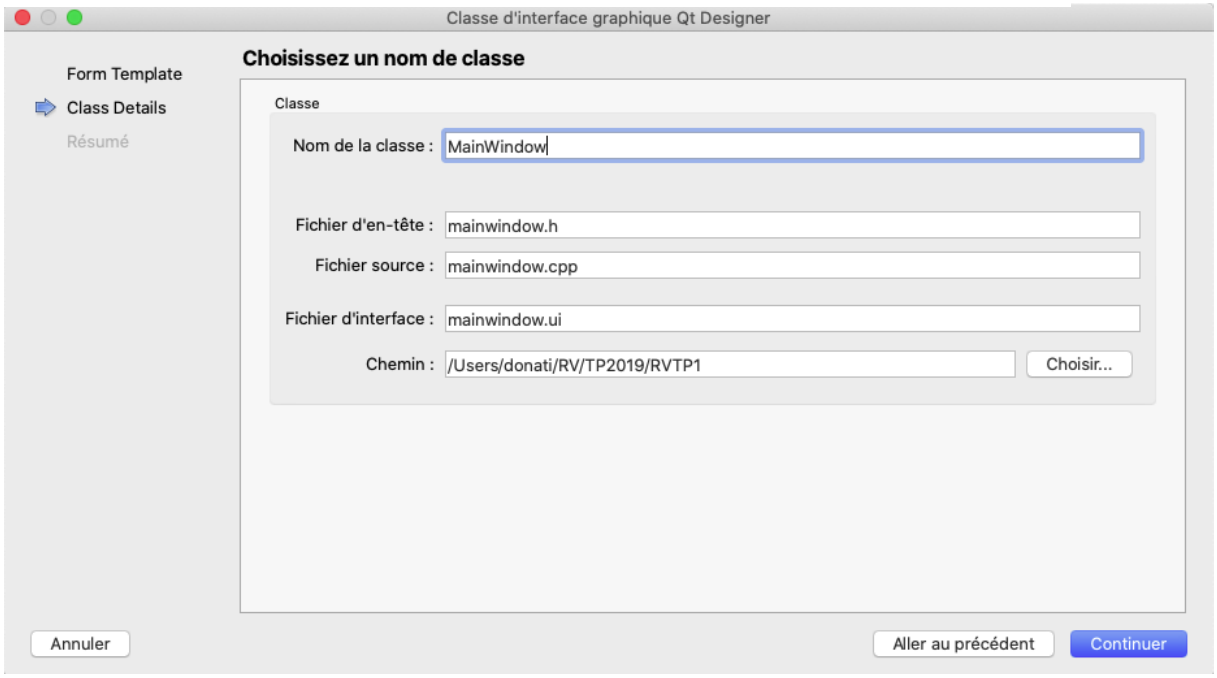
Avec un `Clic-droit` sur le projet, puis `Add New...` on retombe sur une interface que nous avons vu plusieurs fois. Cette fois il faut choisir `Qt` et `Classe d'interface graphique Qt Designer` .



Puis dans `Form Template` choisir `Main Window`



Puis dans `Class Details` conserver `MainWindow` comme nom de classe, puis validez.

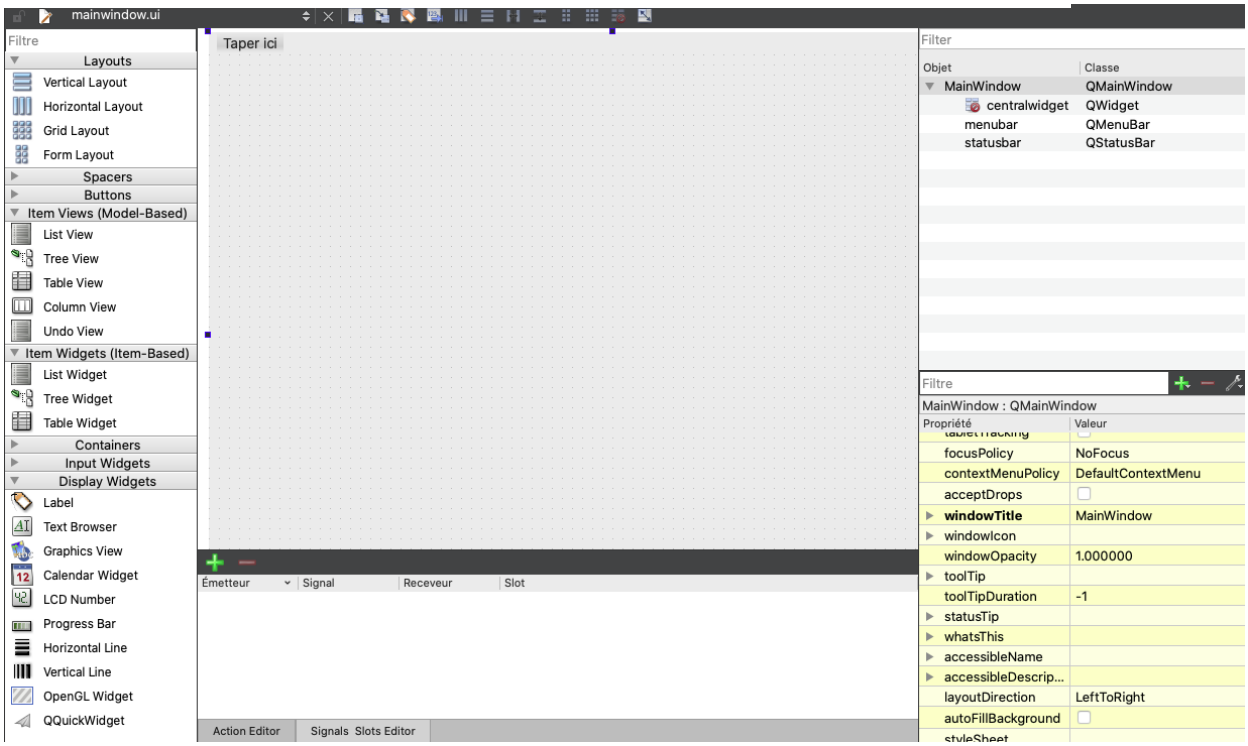


L'assistant de QtCreator a créé 3 fichiers

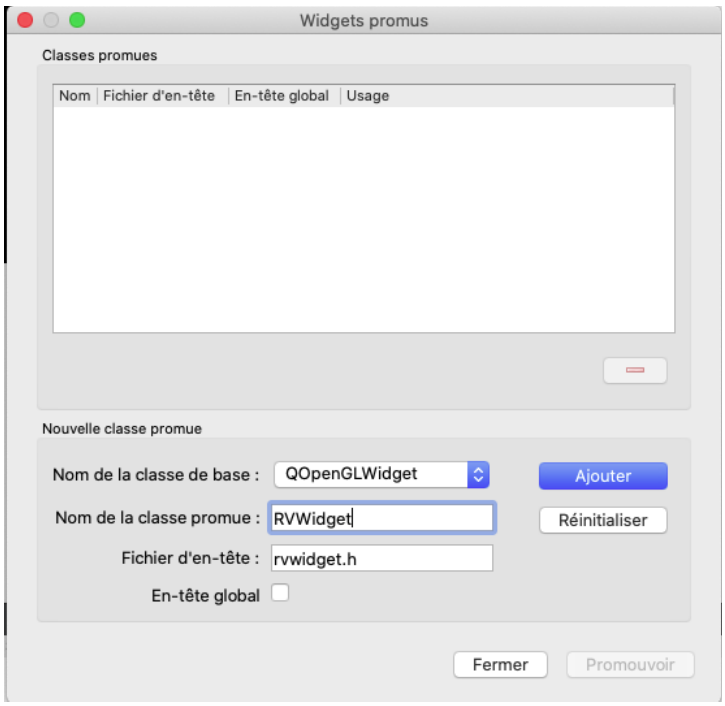
- `mainwindow.h` et `mainwindow.cpp` qui implémentent la classe `MainWindow` . On voit que cette classe hérite de `QMainWindow` et qu'elle contient une seule variable membre `ui` qui représente l'interface graphique (*user interface*) de cette fenêtre. Grâce à `ui` la fenêtre va avoir accès (par leur nom) à tous les composants de l'interface graphique.
- `mainwindow.ui` qui est un fichier `xml` qui va contenir la description de l'interface graphique qui sera créée *graphiquement* grâce à **Qt Designer**. Ce fichier peut donc soit être affiché dans QtDesigner soit affiché en tant que fichier xml. Il est fortement déconséillé d'éditer directement le fichier xml ! C'est le constructeur de `MainWindow` qui via la méthode `setupUI` va lire ce fichier xml et construire l'interface graphique de la fenêtre.

En sélectionnant le fichier `mainwindow.ui`, on active automatiquement l'éditeur d'IHM de Qt qui s'appelle `QtDesigner` . Comme d'autres concepteurs d'interface WYSIWYG, il se compose de

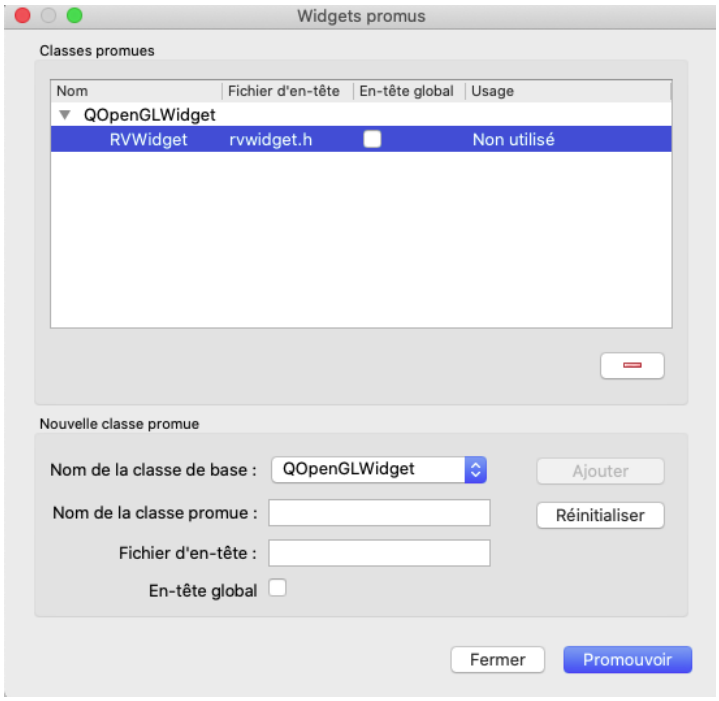
- au centre la fenêtre (ou le widget) que l'on consrtuit ;
- à gauche le `Widget Box` qui présente (organisés en catégories) tous les widget de Qt ;
- en haut à droite : le `Object Inspector` où l'on voit évoluer la hiérarchie des composants que l'on dépose et on place dans la fenêtre principale ;
- en bas à droite : le `Property Editor` qui permet d'accéder et de modifier les propriétés du composant qui est sélectionné ;
- au centre, en bas l'éditeur de Signaux et de Slots (dans un onglet) et l'éditeur d'actions dans un autre.
- au centre en haut, la barre des outils, permet de changer de mode d'édition (édition des widgets, édition des signaux et des slots,...) ou bien pour modifier la mise en page des widgets.



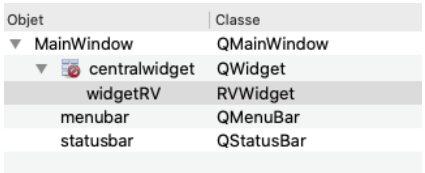
Ajoutez un widget `OpenGL Widget` à la fenêtre principale (on le trouve dans la zone `Display Widgets` de la Widget Box). En sélectionnant le widget à peine créé vous pouvez changer son nom grâce à la propriété `objectName` (dans l'éditeur de propriétés) en `widgetRV`. On voit dans l'inspecteur d'objets que `widgetRV` est une instance de la classe `QOpenGLWidget`. On veut changer cela pour que `widgetRV` soit une instance de notre propre classe `RVWidget`. Pour faire cela, faire un clic-droit sur le widget et choisir `Promouvoir en ...`



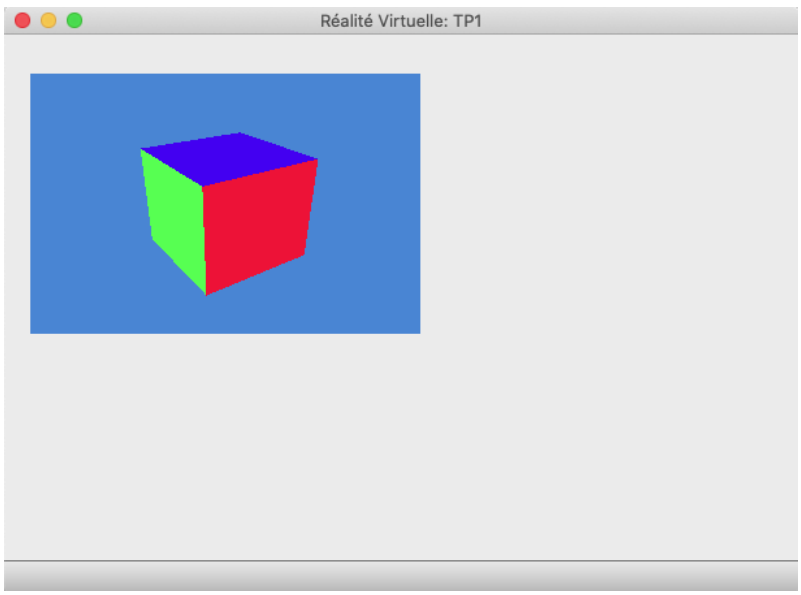
Saisir le nom de la classe `RVWidget` et vérifier que le nom du fichier d'en-tête où est définie cette classe est correct. Puis cliquer sur `Ajouter`.



Puis cliquer sur `Promouvoir` .
Au final, dans l'inspecteur d'objet on doit bien voir apparaître le widget `widgetRV` instance de `RVWidget` .



Il reste à changer le code du `main` pour que l'application principale utilise la fenêtre `MainWindow` au lieu du widget `RVWidget` (puisque maintenant le RVWidget est inclus en tant que widget dans MainWindow).
Donc dans `main.cpp`, il faut juste remplacer la ligne qui dit que `w` est une instance de `RVWidget` , en la déclaration de `w` en tant que instance de `MainWindow` . Il faut aussi mettre le bon `#include`.
Et c'est tout.
En compilant on obtient bien une fenêtre qui contient notre widget OpenGL avec le cube coloré que l'on peut manipuler à la souris !



Mise en place de l'IHM

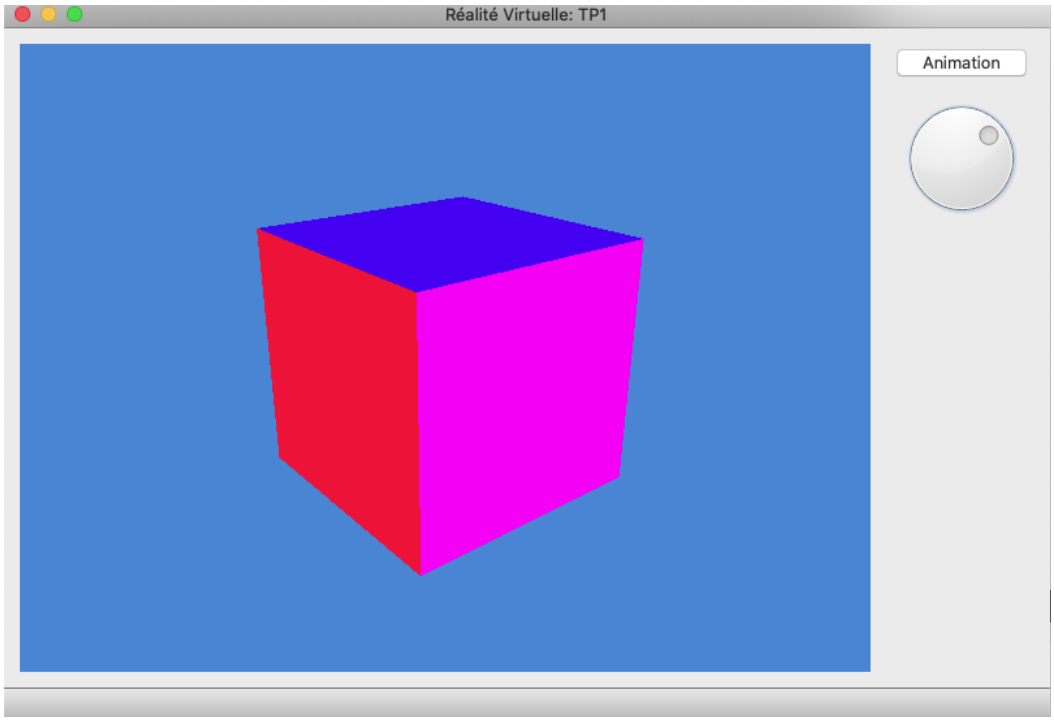
Dans `QtDesigner` ajouter à droite de `widgetRV`

- un `PushButton` nommé `animationButton` (propriété Text),
- un `Dial` (bouton de volume) appelé `fov`
- un `Vertical Spacer` (ressort vertical).

Sélectionnez ces 3 composants et avec un clic droit choisissez `Mettre en place` puis `Mettre en place verticalement` (ou bien un clic sur l'icône ou `Ctrl-L`). Ceci crée un `Vertical Layout` qui organise les trois composants en une colonne verticale. De même sélectionnez le `widgetRV` et le `Vertical Layout` et organisez-le avec `Mettre en place horizontalement` dans un `Horizontal Layout` . Il faut aussi donner une dimension `minimumSize` à `widgetRV` dans l'éditeur de propriétés - par exemple 400x300.
Voià la façon dont apparaissent les différents composants de la fenêtre.

Objet	Classe
MainWindow	QMainWindow
centralwidget	QWidget
horizontalLayout	QHBoxLayout
verticalLayout	QVBoxLayout
animationButton	QPushButton
fov	QDial
verticalSpacer	Spacer
widgetRV	RVWidget
menubar	QMenuBar
statusbar	QStatusBar

On voit que le `centralWidget` n'a pas de règle de mise en place ; choisissez `Mettre en place horizontalement` . Enfin, pour faire en sorte que lors d'un changement de taille la fenête OpenGL soit toujours maximale (par rapport aux autres widgets), il faut changer la propriété `sizePolicy` (politique de taille) en mettant `Expanding` à la fois en horizontal et en vertical.
Lors de la compilation on doit avoir l'aspect suivant.

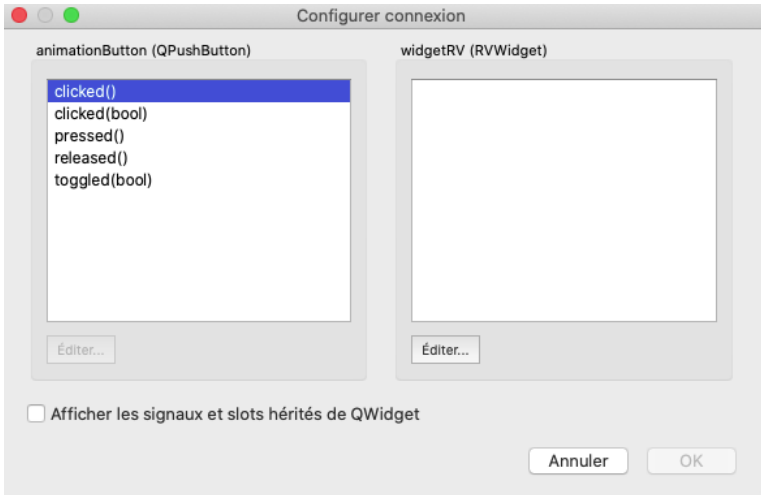

Réalité Virtuelle: TP1
Animation

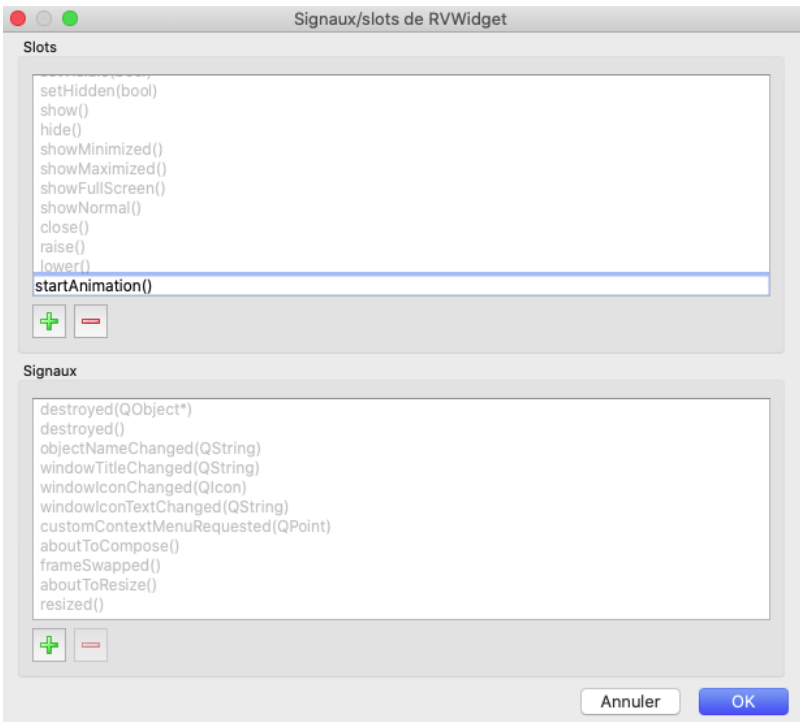
Intéraction

Pour l'instant les deux widget que nous avons ajoutés à la fenêtre principale de notre application ne font rien ! C'est encore grâce au mécanisme `signal/slot` que l'on va les rendre actifs. Mais au lieu de faire les connections par programme (avec la commande `connect`) comme on a fait pour le timer, on va créer ces connections *graphiquement* dans QtDesigner.

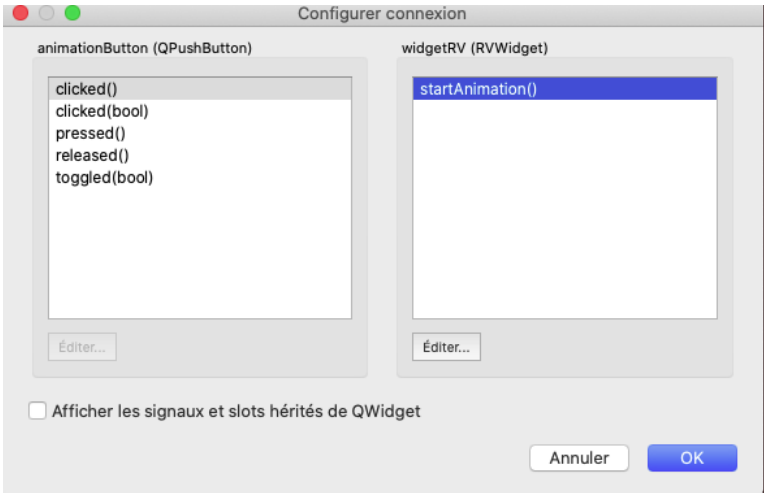
Le bouton

Toujours dans QtDesigner

1. passer en mode `Edit Signal/slots (F4)` 
2. à la souris, faire un *drag* depuis le bouton jusqu'au `widgetRV`
3. la boite de dialogue qui apparaît vous demande de choisir à gauche quel signal de QButton, doit être associé à quel slot de RVWidget (à droite). 

choisir `clicked()` comme signal puis à droite choisir `Editer...` et dans la nouvelle boite de dialogue, ajoutez un slot avec + et donnez-lui le nom `startAnimation` 

4. valider avec `Ok` et ensuite dans le dialogue de départ connectez le signal avec le nouveau slot

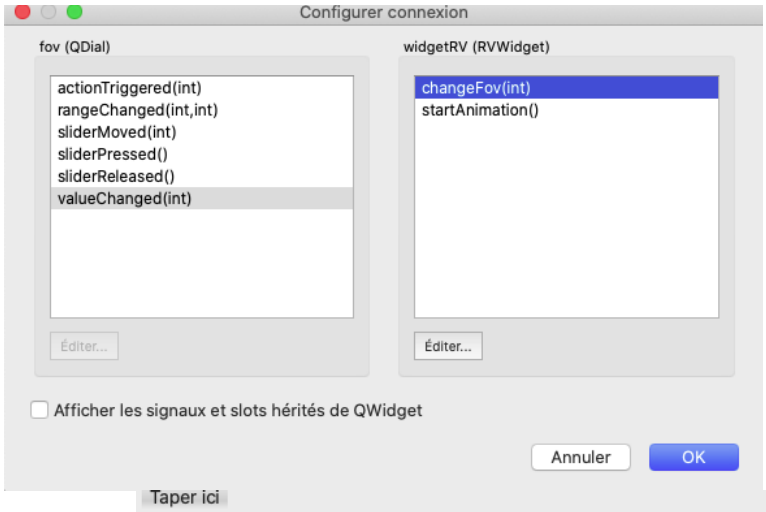


Le résultat apparait dans `QtDesigner` comme un lien qui part de l'émetteur du signal et qui va vers le receveur.

Remarquez que cette action a seulement créé la connection : la méthode `startAnimation()` n'existe toujours pas dans la classe `RVWidget` ! C'est à vous de l'ajouter (en tant que `protected slots:` comme `update()`) et d'écrire le code qui active ou désactive la rotation du cube sur lui-même (en ajoutant les variables membres dont vous pourriez avoir besoin).
En rendant le bouton `checkable` vous pouvez faire en sorte que le bouton reste appuyé lorsque l'animation est activée et reste grisé dans le cas contraire...

Le Dial

Le principe est le même sauf que cette fois le signal de QDial que l'on utilise est `valueChanged(int)` qui est un signal qui comporte un argument de type `int` (qui est donc la nouvelle valeur du Dial après le changement). Donc le slot à qui il faut connecter ce signal doit aussi avoir un argument de type `int`. Appelez ce slot `changeFov(int)`



Ainsi on voit donc en mode `QtDesigner` les deux connections :

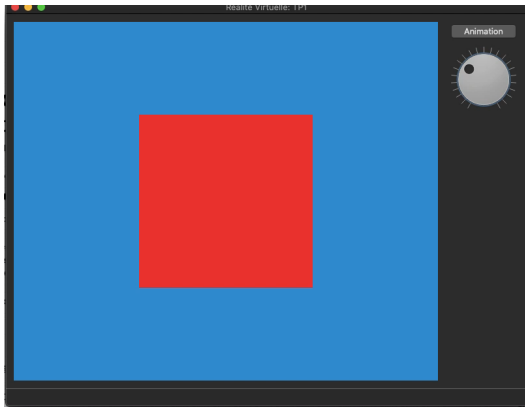
et dans l'éditeur de signaux/slots le détail :

Émetteur	Signal	Receveur	Slot
fov	valueChanged(int)	widgetRV	changeFov(int)
animationButton	clicked()	widgetRV	startAnimation()

La méthode `changeFov(int)` de `RVWidget` doit utiliser la valeur passée en argument pour la transformer en un angle en degrés qui sera utilisé dans la définition de la matrice de projection `proj` à passer au vertex shader.
On peut affiner le comportement de `QDial` dans l'éditeur de propriété en imposant que les valeurs permises soient uniquement entre 10° et 120° et que la valeur de départ soit 45° qui est la valeur par défaut que nous avons donné à cet angle dans le tuto n°1.

Gestion du resize

Quand on déforme la fenêtre de l'application, on voit bien que toute la place en plus est prise par le `RVWidget` alors que les autres composantes gardent leur taille fixe et aussi qu'on ne peut pas retrécir la fenêtre au dessous d'une taille minimale.
Ce qui ne va pas en revanche c'est que dans certains cas le cube apparaît étiré ou aplati : il n'a plus la forme d'un cube !
Ce qu'il s'est passé vient du fait que dans la définition de la matrice de projection `proj` on a fixé le *aspect ratio* du rectangle qui recevait la projection en 1.33 (c'est à dire 4/3). Cela allait bien tant que le widget avait ce ratio 400px x 300px) mais si ce ratio change il y a une déformation qui se produit dans la transformation de viewport lorsqu'OpenGL transforme un buffer d'affichage au ratio 4/3 sur un widget dans la taille n'est pas 4/3 !
La solution conssite à adapter dans la matrice `proj` l'argument `aspetcRatio` au rapport réel largeur/hauteur du widget :
tout widget possède des méthodes `Width()` et `heigth()` qui donnent ces informations sous la forme de `int`. Attention donc lorsque vous calculez le rapport à ne pas faire de division entre deux `int` !


Animation

Transparence dynamique grâce au shader

On veut ajouter un nouvel élément d'IHM : un slider qui permette de régler l'opacité (la transparence) du cube entre 0 (totalement transparent, donc invisible) et 1 (opaque). L'opacité sera une nouvelle variable uniforme du vertex shader que l'on utilisera comme composante *alpha* de la couleur de chaque vertex.

1. Dans Qt Designer, ajoutez sous le QDial, un `Horizontal Slider` , instance de `QSlider` ; donnez-lui le nom `opacitySlider` avec un intervalle de valeurs allant de 0 à 100 et une valeur de départ de 100. Modifier sa politique de taille horizontale à `Preferred`.
2. En mode **Editeur de signaux/slots** associez au signal `valueChanged(int)` du slider le signal `changeOpacity(int)` de `RVWidget` .
3. Dans la classe `RVWidget`, ajoutez :
   - une variable membre `m_opacity` de type `int` initialisée à 100.
   - un slot `void changeOpacity(int newOpacity)` qui met à jour la valeur de l'opacité et appelle la méthode `update()`.
4. Dans le vertex shader ajoutez une variable uniforme `u_Opacity` de type `float` et utilisez cette variable comme 4ème argument de `outColor`.
5. Dans la méthode `glPaint()` de `RVWidget` il faut utiliser `m_opacity` pour renseigner la variable uniforme `u_Opacity`. Attention de bien convertir un entier entre 0 et 100 en un float entre 0 et 1 !
6. Parmi les réglages du contexte de rendu OpenGL, par défaut la transparence n'est pas activée. Si l'on veut que le moteur de rendu utilise la valeur *alpha* de la couleur pour calculer la transparence il faut modifier les réglages par défaut. Donc dans `initializeGL()` il faut rajouter

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

   Remarquez qu'avec la transparence et le *culling* désactivé, on voit apparaître la face *arrière*.