Tuto 1

Site: lms.univ-cotedazur.fr
Cours: Realite virtuelle - EIMAD919
Livre: Tuto 1

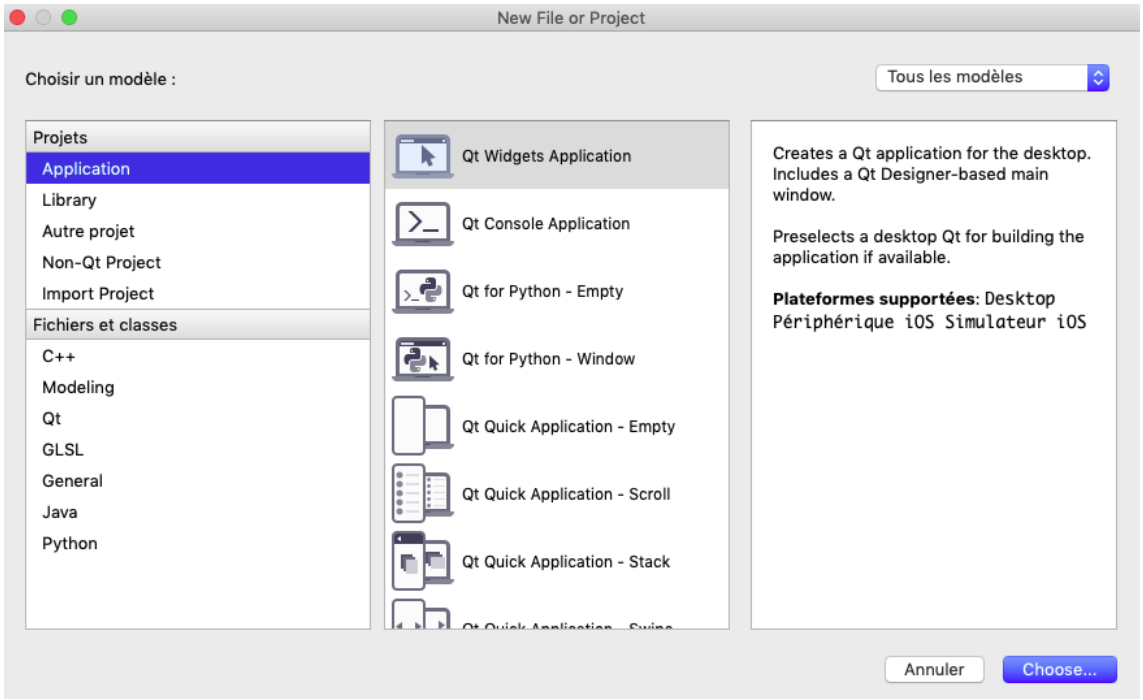
Imprimé par: Theo bonnet
Date: vendredi 28 février 2020, 14:50

Table des matières

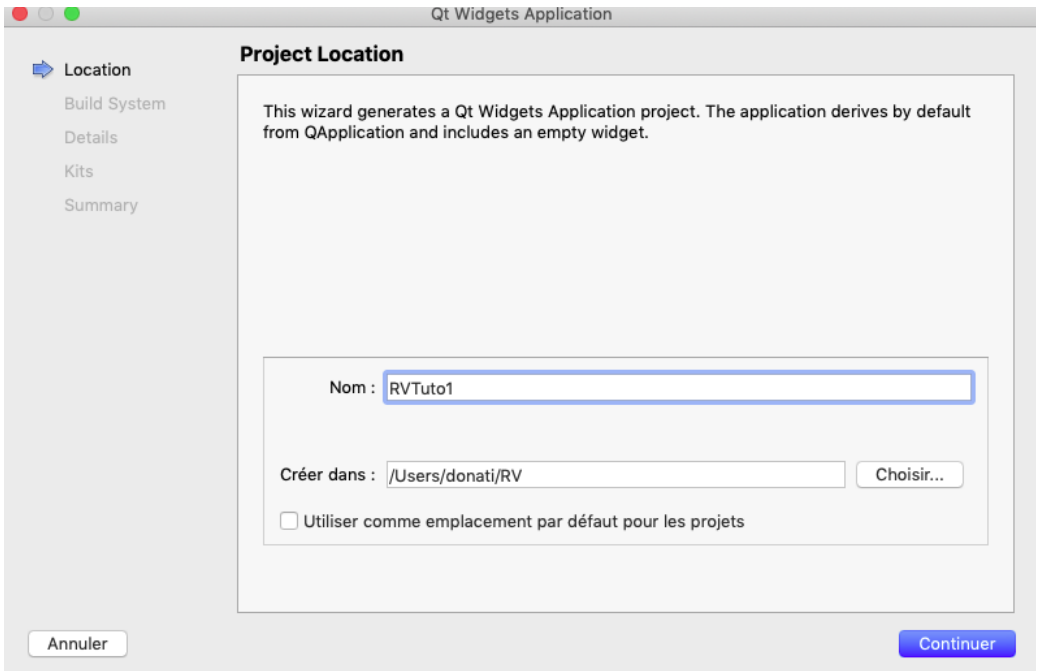
Création du projet

Dans Qt Creator, dans le menu **Fichier** choisir **Nouveau fichier ou projet..**.

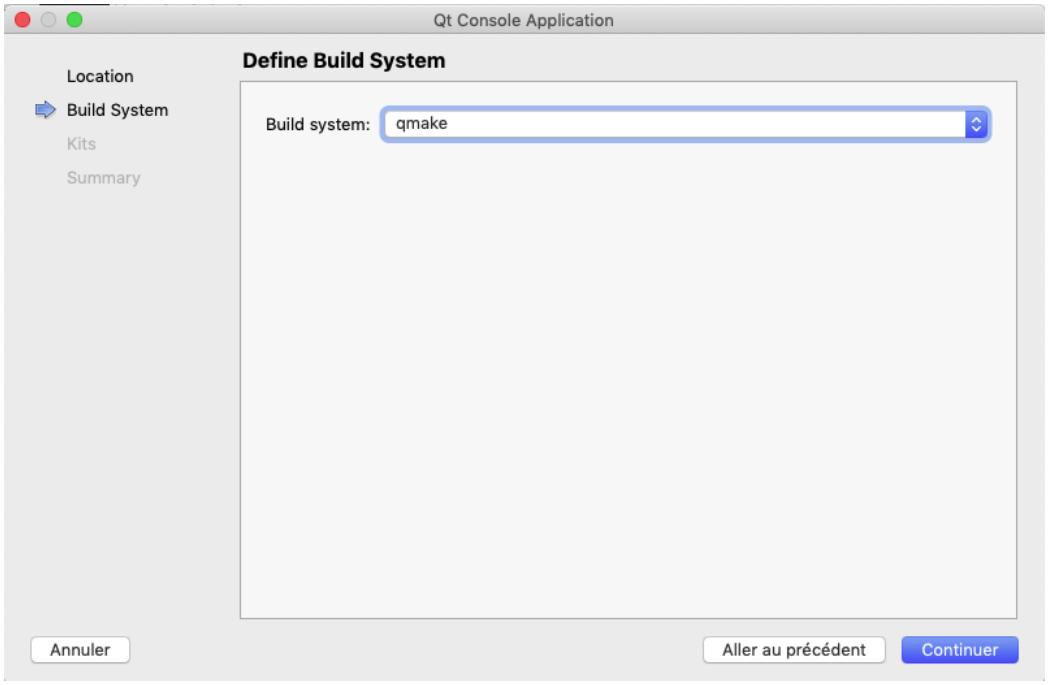
Dans la boite de dialogue qui apparaît, choisir le modèle **Qt Widgets Application**.



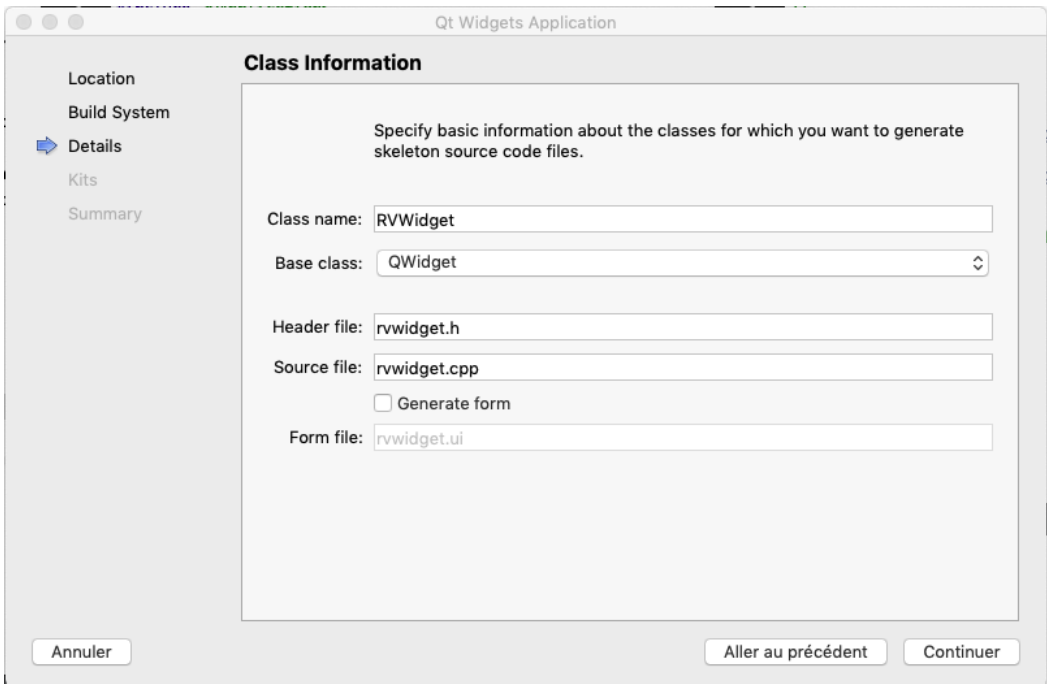
Dans la page **Location** choisissez le répertoire de création et comme nom de projet saisissez `RVTuto1`.



Dans la page **Build System** choisir `qmake`.

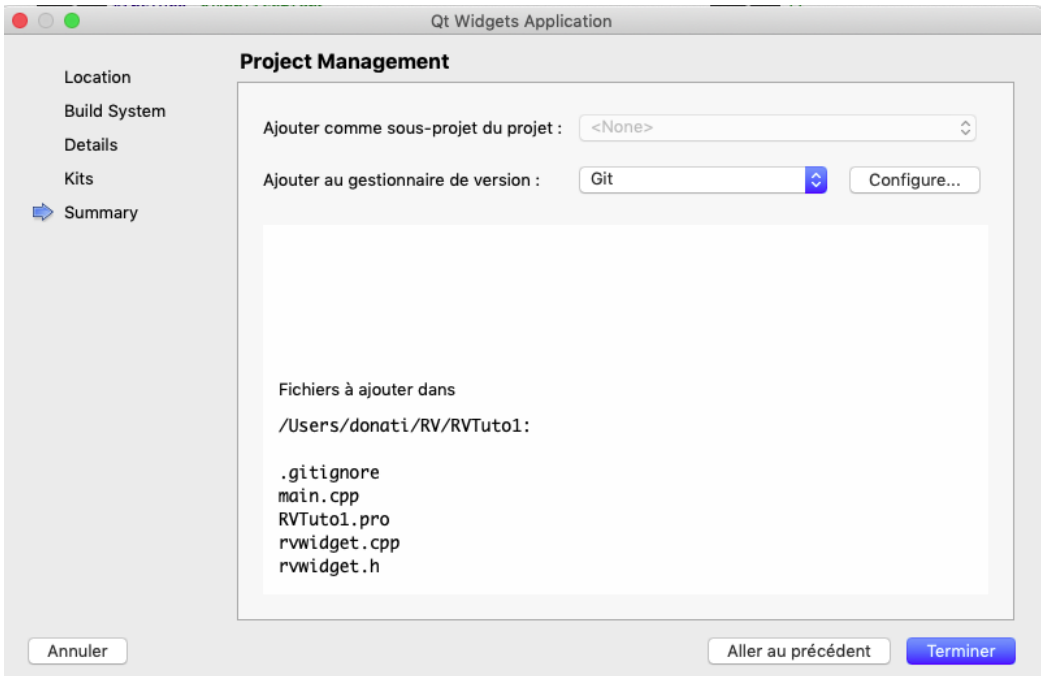


Dans la page **Details**, le nom de la classe du widget sera `RVWidget`, la classe de base est `QWidget` et ne pas cocher la case **Generate form**.



Dans la page **Kits**, sélectionnez le kit le plus récent pour la plateforme sur laquelle vous êtes.

Dans la page **Summary** vous pouvez sélectionner un outil de gestion de version et puis **Terminer**.



Résultat

L'assistant de création de projet a généré 4 fichiers :

- `RVTuto1.pro` qui est le fichier de projet qui indique quels modules de Qt sont utilisés (core et gui), quel type de version de langage à utiliser (C++11), quels fichiers sources et quels fichiers d'en-tête ajouter au projet. Ce fichier sera utilisé par le métacompilateur de Qt (qmake) pour générer un makefile nécessaire à la compilation C++ proprement dite (avec le compilateur choisi dans le Kit).

  Sous **Windows**, pour pouvoir utiliser OpenGL il faut ajouter à la fin de ce fichier l'instruction

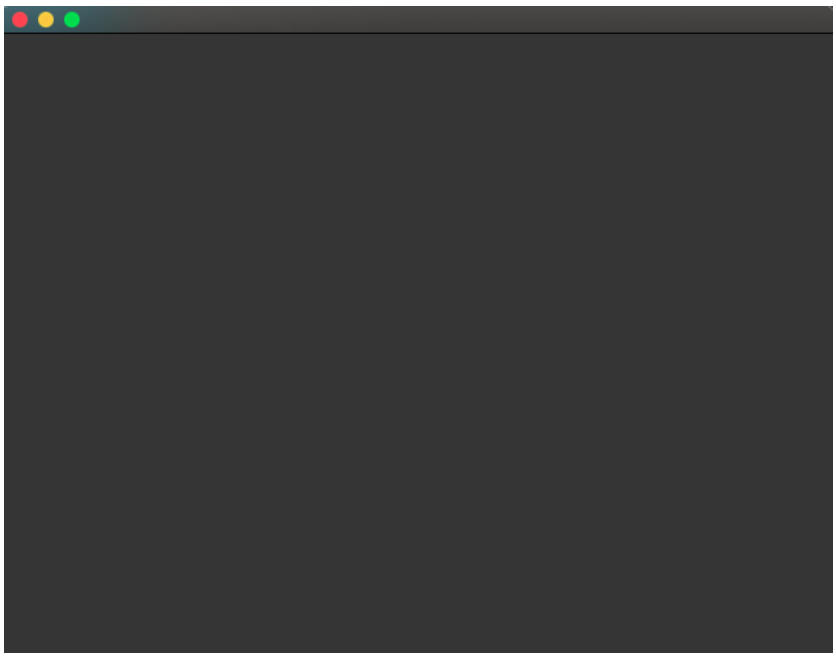
  ```
  windows:LIBS += -lopengl32
  ```

  pour indiquer d'inclure la bibliothèque opengl32.lib.

- `main.cpp` est le point d'entrée du programme : on crée une instance de la classe `QApplication` et une instance de `RVWidget` .

  Puis on active l'affichage du widget puis on lance l'exécution du programme.
- La classe `RVWidget` est déclarée dans le fichier d'en-tête `rvwidget.h` et implémentée dans `rvwidget.cpp`.

Si l'on compile et exécute ce programme (triangle vert) on obtient une fenêtre vide et noire. Ce n'est pas du tout une application OpenGL mais juste un widget quelconque de Qt.



QOpenGLWidget

Dans `rvwidget.h` changer la classe mère dont hérite `RVWidget` de QWidget en `QOpenGLWidget` . Modifier aussi l'*include* et dans `rvwidget.cpp` changer le constructeur de `RVWidget` de façon à ce qu'il appelle le constructeur de la nouvelle classe mère.

Héritage double : notre widget va aussi hériter (de façon `protected`) de la classe `QOpenGLFunctions` . De cette façon à l'intérieur de la classe on va automatiquement avoir accès à l'ensemble de toutes les fonctions les plus récentes de OpenGL. Du coup dans le constructeur il faut aussi appeler le constructeur de la classe parent.

```
#ifndef RVWIDGET_H
#define RVWIDGET_H

#include <QOpenGLWidget>
#include <QOpenGLFunctions>

class RVWidget : public QOpenGLWidget, protected QOpenGLFunctions
{
    Q_OBJECT

public:
    RVWidget(QWidget *parent = nullptr);
    ~RVWidget();
};
#endif
```

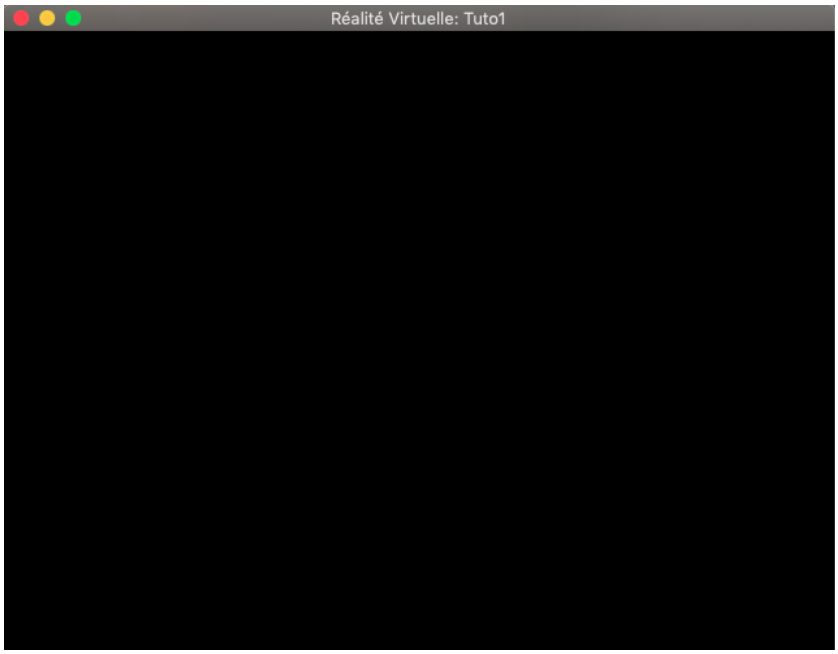
`QOpenGLWidget` est un composant graphique de Qt (un widget) qui met en place automatiquement le contexte de rendu (*rendering context*) nécessaire à OpenGL pour fonctionner. Ce qui fait que cette fois, si l'on compile et lance le programme, on aura toujours un écran vide et noir mais cette fois il y a un moteur de rendu OpenGL prêt à fonctionner.

La ligne Q_OBJECT est une macro définie par Qt pout faire fonctionner le mécanisme des *signaux* et des *slots* de Qt. Cela fait partie des actions du métacompilateur `qmake` pour produire du vrai code C++ à la fin. A ne pas toucher !

```
#include "rvwidget.h"

RVWidget::RVWidget(QWidget *parent)
    : QOpenGLWidget(parent), QOpenGLFunctions()
{
}

RVWidget::~RVWidget()
{
}
```

Une petite ligne suffit pour donner un titre à la fenêtre de l'application : dans `main.cpp` après avoir créé `w` (instance de `RVWidget`) on peut appeler sa méthode `setWindowTitle()` en lui passant une chaîne de caractère.

```
#include "rvwidget.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    RVWidget w;
    w.setWindowTitle("Réalité Virtuelle: Tuto1");
    w.show();
    return a.exec();
}
```



La classe `QOpenGLWidget` de Qt non seulement prépare pour nous le contexte de rendu mais contient 3 méthodes virtuelles à surcharger qui assurent l'essentiel de la boucle de rendu.

- `void initializeGL()` dans laquelle on met tout le code responsable de l'initialisation : initialisation de la scène tridimensionnelle, réglages du moteur de rendu OpenGL, initialisation du microcode à envoyer à la carte graphique pour produire le rendu.
- `void paintGL()` dans laquelle, avant de lancer la commande de rendu qui fera démarrer le calcul du rendu, il faut passer des paramètres au micro-code qui peuvent varier à chaque appel de paintGL pour ainsi produire une animation. Cette méthode est automatique ment appelée chaque fois que le système considère que le contenu du widget n'est pas à jour et qu'il faut donc repeindre son contenu.
- `void resizeGL()` est la méthode automatiquement appelée lorsque le widget change de taille ou lorsqu'il passe d'un état iconifié à un état actif.

Instructions : ajouter ces trois déclarations de méthodes publiques (en les marquant *override*) à la classe dans le fichier `rvwidget.h`, puis après un `clic-droit` et `Refactor` on peut demander à QtCreator de produire leur définition dans `rvwidget.cpp`

Ajouter un *override* aussi après le *destructeur* pour ne plus avoir de message de Warning.

Pour vérifier que OpenGL fonctionne bien, nous allons écrire nos premières commandes openGL

- dans `initializeGL` appeler `initializeOpenGLFunctions()` pour activer les fonctions OpenGL de la classe mère `QOpenGLFunctions`, puis appeler `glClearColor(0.8f, 0.8f, 0.8f, 1.0f)` pour définir que la couleur de fond est le gris clair (les couleurs en OpenGL sont des *floats* entre 0 et 1 et les quatre composantes sont rouge, vert, bleu et alpha pour la transparence).
- dans `paintGL` on utilise la commande `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);` pour demander que le buffer de couleur (c'est à dire l'image contenue dans le widget) soit effacée (et du coup remplie avec la couleur définie dans `glClearColor` ci-dessus).

Le résultat c'est que l'écran est toujours vide mais cette fois il est gris!.

Remarque : toutes les fonctions openGL *pures* commencent par **gl** !

Vertex Buffer Object

Pour créer la scène tridimensionnelle, il faut beaucoup de points : ces points sont les sommets des triangles (et autres objets géométriques qu'OpenGL appelle des *primitives*) qui définissent la surface (l'exterieur) des objets à rendre sous la forme d'un maillage. Ces sommets (*vertex* pluriel *vertices* en anglais) ont des coordonnées (x, y, z) dans un repère local (repère du modèle) mais *portent* aussi d'autres données qui seront utilisées par les algorithmes de rendu. Par exemple (c'est ce qu'on va faire aujourd'hui) on peut associer à chaque sommet une *couleur* ; mais cela aurait pu être une information d'opacité, de texture, des vecteurs nécessaires au calcul d'éclairage, etc..

Ces données vont être utilisées par le GPU de façon intensive. Ils faut donc que ces données soient stockées dans la mémoire vidéo de la carte graphique. OpenGL fournit un objet appelé *Vertex Buffer Object* abrégé en VBO qui permet de gérer facilement ces données.

Qt offre une classe `QOpenGLBuffer` qui permet encore plus facilement de travailler avec le VBO.

On va définir une scène toute simple à partir de 4 sommets colorés non coplanaires (pour donner une première sensation de 3D) au lieu du triangle classique (qui le Hello World de OpenGL en quelque sorte). ICI UN DESSIN

- On ajoute à la classe `RVWidget` une variable membre `m_vbo` de type `QOpenGLBuffer` (ne pas oublier les include dans le fichier d'en-tête).
- Ajouter à la classe `RVWidget` une méthode `void initializeBuffer()`.
- Dans cette méthode on définit 4 points A(-1, 0, 0), B(1, 0, 0), C(1, 1, 0), D(-1, 1, -1) sous la forme d'instance de `QVector3D` .
- On définit aussi 4 couleurs rouge, bleu, blanc et noir (toujours comme des `QVector3D`)
- On met ces 8 objets dans un tableau `vertexData`
- On initialise la variable `m_vbo`
- On crée le vertex buffer
- On lie le vertex buffer au contexte
- On copie `vertexData` dans le vertex buffer
- On libère le vertex buffer.

```
void RVWidget::initializeBuffer()
{
    //Définition de 4 points
    QVector3D A(-1, 0, 0);
    QVector3D B(+1, 0, 0);
    QVector3D C(+1, 1, 0);
    QVector3D D(-1, 1, -1);

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

    //On prépare le tableau des données
    QVector3D vertexData[] = {
        A, B, C, D,
        rouge, noir, bleu, blanc
    };

    //Initialisation du Vertex Buffer Object
    m_vbo = QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
    //Création du VBO
    m_vbo.create();
    //Lien du VBO avec le contexte de rendu OpenGL
    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();
}
```

La méthode `intializeBuffer` est appelée dans `initializeGL` qui est la méthode de `QOpenGLWidget` qui est appelée automatiquement lors de l'affichage du widget à l'écran, après avoir préparé le contexte de rendu de OpenGL.

Remarquez que les variables `A, B, C, D, rouge, noir, ..., vertexData` sont locales à la méthode et sont donc détruites aussitôt que l'on sort de cette fonction dont le seul but a été de préparer et de remplir le VBO (géré par `m_vbo`).

Shaders

Les shaders sont des petits programmes écrits pour le processeur graphique (GPU) qui permettent de spécifier finement certaines des étapes mises en place par OpenGL pour produire une image à partir des données contenues dans le vertex buffer object (VBO). Toutes les étapes du *pipeline de rendu* ne sont pas programmables mais celles pour lesquelles il faut obligatoirement écrire du code sont :

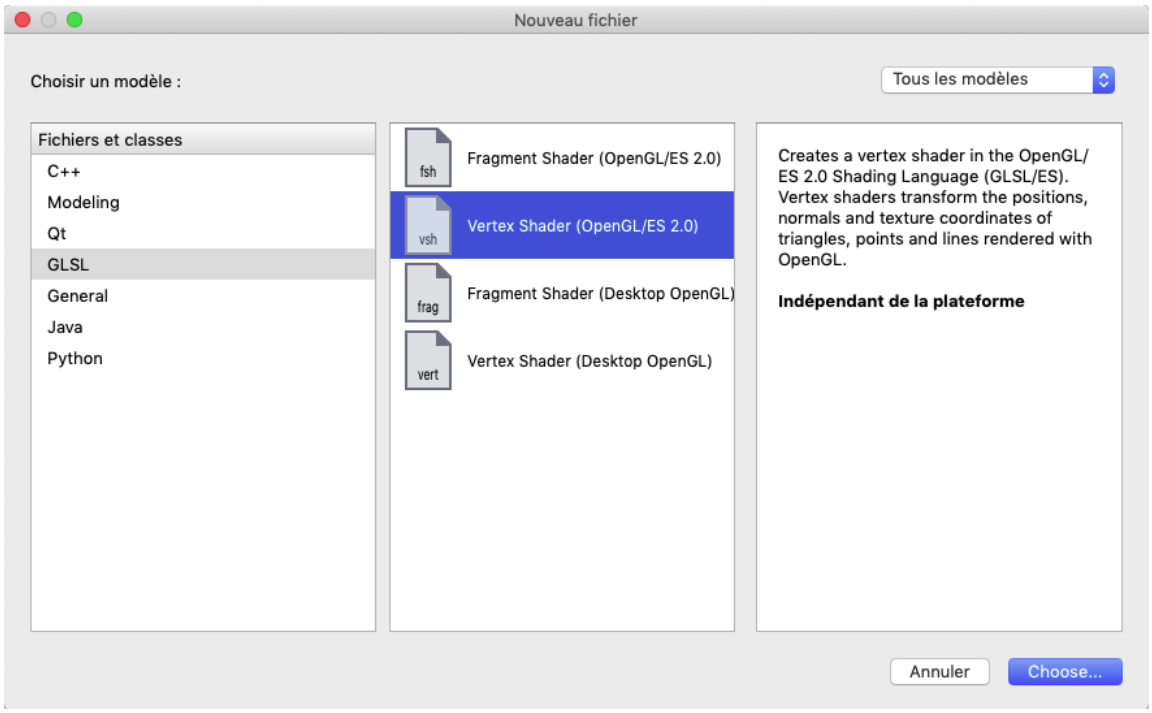
- le vertex shader : c'est un programme qui reçoit en entrée (depuis le vertex buffer) un sommet à la fois (ses coordonnées **plus** les autres informations qui ont été ajoutées comme la couleur). Il est responsable *a minima* de produire un nouveau point `gl_Position` obtenu après projection dans le plan. D'autres données peuvent être calculées et passées à l'étape suivante du pipeline de rendu.
- le fragment shader : c'est presque la dernière étape du pipeline. Un *fragment* est une espèce de pixel tri-dimensionnel qui a été obtenu en discrétisant la primitive construite à partir des sommets (par exemple un triangle se construit avec 3 sommets ; après discrétisation (*rasterization)* ce triangle, selon sa taille, peut donner lieu à des centaines ou des milliers de fragments. Le fragment shader traite chaque fragment individuellement (en réalité cette étape est hautement vectorialisée au niveau du GPU) afin de produire une couleur `gl_FragColor`.

Les shaders sont écrits dans un langage de programmation appelé *GLSL*.

Ajouter un vertex shader

Création

Avec un clic-droit sur le projet, choisir `Add New...` puis dans la boite de dialogue choisir `GLSL` dans la première colonne, puis `Vertex Shader (OpenGL/ES 2.0)` dans la seconde (pour avoir un shader qui soit compatible à la fois aux cartes graphiques des Desktop et avec celles des dispositifs mobiles comme les smartphone).



Donner à votre vertex shader le nom `VS_simple`.

Résultat

En cliquant sur le fichier nouvellement créé `VS_simple.vsh` dans l'explorateur de projet de QtCreator sous la rubrique `Other files` on voit le code suivant :

```
attribute highp vec4 qt_Vertex;
attribute highp vec4 qt_MultiTexCoord0;
uniform highp mat4 qt_ModelViewProjectionMatrix;
varying highp vec4 qt_TexCoord0;

void main(void)
{
    gl_Position = qt_ModelViewProjectionMatrix * qt_Vertex;
    qt_TexCoord0 = qt_MultiTexCoord0;
}
```

Sans entrer dans le détail dans un premier temps (car on va le remplacer par un code encore plus simple) ce qu'il faut retenir est :

- le code principal du vertex shader est contenu dans le `main` et ne fait que *deux lignes !*
- quatre variables globales sont définies dans le préambule `qt_Vertex, qt_MultiTexCoord0, qt_ModelViewProjectionMatrix, qt_TexCoord0`
  - leur *type* est `vec4` (un vecteur à 4 coordonnées) et `mat4` (une matrice carrée 4x4)
  - ces variables ont toutes des spécificateurs qui indiquent leur statut dans le pipeline de rendu OpenGL

  `attribute`

  indique que ces variables viennent directement des données présentes dans le vertex buffer.

  `uniform`

  est une variable qui est constante durant cette phase de rendu, la même quelle que soit la donnée provenant du vertex buffer; ces variables sont spécifiées par le programme C++ via des commandes OpenGL et font partie du processus préliminaire de préparation du pipeline de rendu (voir plus bas).

  `varying`

  est une donnée en sortie qui va donc être attachée au vertex actuel en plus de sa position qui est dans `gl_Position`.

  - le mot-clé `highp` est une spécificité de OpenGL ES qui indique le type de précision utilisé pour ces variables et est totalement ignoré dans le OpenGL sur desktop.

Modification du vertex shader

Le vertex shader par défaut est assez simple mais il correspond à une situation où la couleur est donnée par une texture qui est appliquée sur les faces des triangles (voir dans quelques semaines). Voilà pourquoi chaque sommet doit porter (et transmettre) des *coordonnées texture*.

1. Dans notre cas le vertex shader contient pour chaque vertex une position (3 coordonnées) et une couleur (3 coordonnées) donc nos deux attributs vont être deux `vec3` et on les appellera `rv_Position` et `rv_Color`. Qui vont remplacer les anciens attributs.
2. Pour la variable uniforme on va laisser comme ça, juste en mettant le préfixe `u` à la place de `qt` (pour montrer qui c'est qui commande, et parce que c'est une variable uniforme).
3. la variable varying sera la couleur en sortie du VS, que l'on va appeler `outColor` et qui sera de type `vec4` pour intégrer la transparence.
4. dans la première ligne du `main`, il faut évidemment insérer le nouveau nom de la variable `rv_Position` au lieu de `qt_Vertex`. Mais attention ! comme on ne peut pas multiplier une matrice 4x4 par un vecteur 3x1, il faut construire un `vec4` à partir de `rv_Position` en écrivant `vec4(rv_Position, 1)` ce qui permet de mettre la dernière coordonnée à 1 et pas à 0 (l'explication viendra lorsque nous verrons les coordonnées homogènes).
5. enfin on remplace la seconde ligne du `main` par `outColor = vec4(rv_Color, 1);` La aussi on amis la quatrième coordonnée de la couleur à 1, ce qui correspond à une opacité totale.

Après modification la nouvelle version du vertex shader est :

```
attribute highp vec3 rv_Position;
attribute highp vec3 rv_Color;
uniform   highp mat4 u_ModelViewProjectionMatrix;
varying   highp vec4 outColor;

void main(void)
{
    gl_Position = u_ModelViewProjectionMatrix * vec4(rv_Position, 1);
    outColor = vec4(rv_Color, 1);
}
```

Ajouter un fragment shader

Pour créer le fragment shader les étapes sont les mêmes que ci-dessus : à la fin on doit obtenir le fichier `FS_simple.fsh` dont le contenu (encore plus simple que le VS) est affiché ci-dessous :

```
uniform sampler2D qt_Texture0;
varying highp vec4 qt_TexCoord0;

void main(void)
{
    gl_FragColor = texture2D(qt_Texture0, qt_TexCoord0.st);
}
```

Ici encore il y a une variable `uniform` (qui représente la texture à utiliser) et une variable `varying` en entrée qui n'est rien d'autre que la varying qui sortait du vertex shader.

Modification du fragment shader

1. on enlève les deux variables gloables et on ajoute la varriable varying `outColor` de notre vertex shader,
2. on enlève tout le code du `main` et on remplace par `gl_FragColor = outColor`.

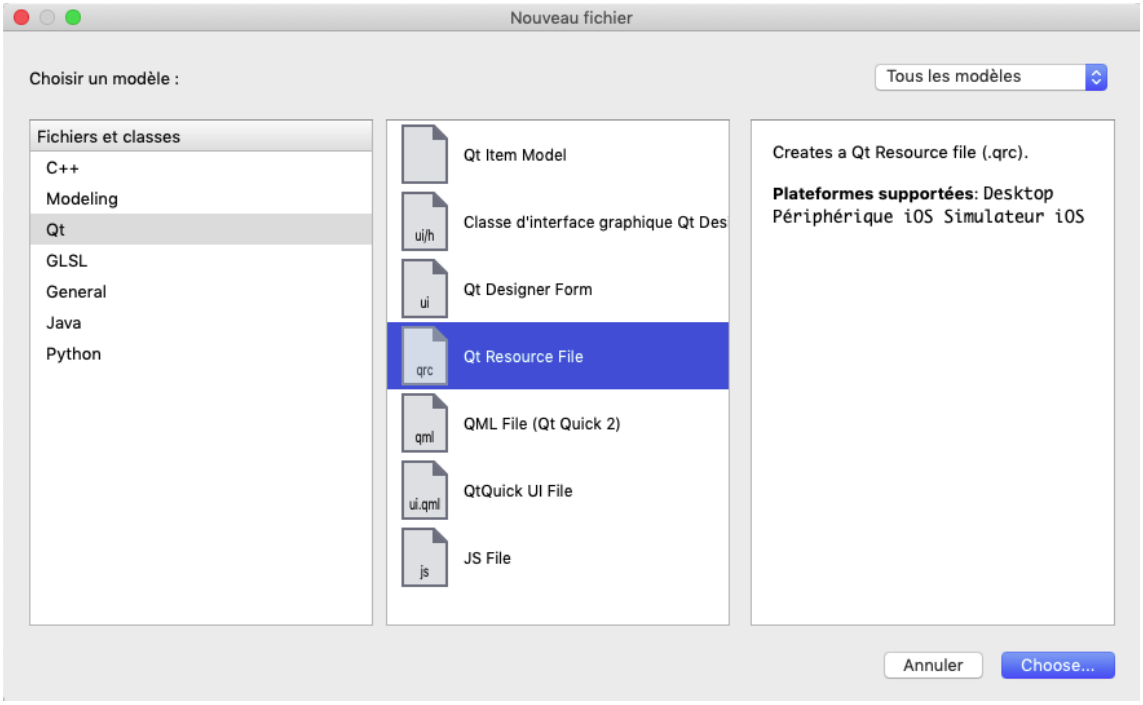
Autrement dit le fragment shader ne fait rien d'autre que de donner au fragment la couleur qui arrive en entrée (sans aucun traitement).
Voilà donc le code final du fragment shader

```
varying highp vec4 outColor;

void main(void)
{
    gl_FragColor = outColor;
}
```

Utilisation des shaders

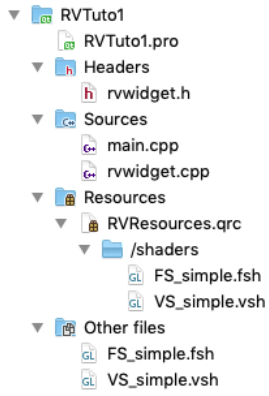
On revient au C++ et à notre classe `RVWidget` . Les deux programmes de shader doivent être lus à partir du fichier, compilés et envoyés au GPU. La compilation des shaders se fait durant *run-time* et pas quand le programme C++ est compilé. En effet c'est seulement durant le run time que l'on connait le type de carte graphique de la machine sur laquelle est en train de tourner le programme et donc la compilation du shader doit d'adapter au matériel présent à cet instant là. Heureusement c'est le driver de la carte graphique qui sert d'interface entre les deux.
Pour toutes ces étapes, Qt propose la classe `QOpenGLShaderProgram` qui encapsule les deux programmes de shader et qui a des méthodes pour les compiler.

1. Ajouter à la classe `RVWidget` une variable membre `m_program` de type `QOpenGLShaderProgram` . Ajouter aussi l'include correspondant.
2. Ajouter à la classe `RVWidget` une méthode `protected` appelée `initializeShader()`. C'est cette méthode qui sera chargée de lire les shaders à partir des deux fichiers, de les compiler et de les *linker*.
3. Ajouter les shaders en tant que ressources au projet (pour qu'ils soient incorporés dans le binaire produit lors de la compilation) : avec un `Clic-droit` sur le projet choisir `Add New...` et dans la boite de dialogue qui apparaît choisir `Qt` et `Qt Resource File` .



Appelez ce fichier de ressources `RVResources`.

4. Dans le navigateur de fichier, il y a un nouveau groupe appelé `Resources` dans le quel se trouve le fichier `RVResources.qrc`. Avec un clic-droit sur le fichier de ressources, choisissez `Add Prefix...` et saissez `shaders`. Puis dans l'onglet `shaders` qui vient d'être créé, choisissez (toujours par clic-droit) `Ajouter des fichiers existants` et sélectionnez `VS_simple` et `FS_simple`. Ainsi ces deux fichiers font partie du *bunlde* du projet et on peut donc y faie référence en utilisant un chemin logique `":/shaders/VS_simple.vsh"`.



1. Dans la méthode `initializeShader()`:
    1. On appelle la méthode `create()` de `m_program`,
    2. On appelle sa méthode `bind()` pour lier le programme de shader au contexte de rendu,
    3. On charge le fichier contenant le vertex shader et on le compile avec

       ```
       m_program.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/shaders/VS_simple.vsh");
       ```

       Le premier argument indiue qu'il s'agit bien du vertex shader et le second est le chemin vers le fichier. Remarquer que cette fonction renvoie un cooléen que l'on peut tester pour savoir si le chargement a bien fonctionné. Dans le code fourni en annexe, c'est ce que je fais (voir).
    4. On fait la même chose avec le fragment shader :

       ```
       m_program.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/shaders/FS_simple.fsh");
       ```
    5. Avec la méthode `link()` on *lie* les deux programmes ensembles et on les associe au contexte de rendu. C'est cette étape qui transfère le code binaire des shaders à la carte graphique.
    6. On libère le programme de shader avec `release()`

Cette méthode `initializeShader()` doit être appelée dans `initializeGL()`.

Voici le détail du code de la méthode `initializeShader()`:

```
void RVWidget::initializeShaders()
{
    bool resultat;
    m_program.create();
    m_program.bind();

    //Vertex Shader
    resultat = m_program.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/shaders/VS_simple.vsh");
    if (!resultat)     {
        ...
    }

    //Fragment Shader
    resultat = m_program.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/shaders/FS_simple.fsh");
    if (!resultat)     {
        ...
    }

    //Link
    resultat = m_program.link();
    if (!resultat)     {
        ...
    }

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

Préparation et rendu

Tous les éléments sont prêts et il ne reste qu'à mettre ensemble les différents éléments :

- connecter les données du *vertex buffer object* (position et couleur) aux entrées correspondantes du *vertex shader*,
- définir la mystérieuse matrice `ModelViewProjection` dont le vertex shader a besoin en tant que variable *uniforme*
- lancer la commande de rendu en expliquant quoi faire, quoi construire avec les 4 sommets que l'on a mis dans le VBO
- définir la façon dont on veut que le buffer de rendu soit affiché dans le widget.

Pour l'instant nous allons tout faire dans la méthode `paintGL()` de la classe `RVWidget` même si nous verrons ensuite, avec les *vertex array objects* (VAO) que la première de ces actions peut être faite une seule fois dans `initializeBuffer()`. Donc après l'appel de `glClear` :

1. Lier `m_program` et `m_vbo` au contexte de rendu.
2. Indiquer que l'attribut `rv_Position` du vertex shader
   - utilise comme type de base le `float`
   - se compose de `3` floats
   - commence à la position 0 du VBO

   ```
   m_program.setAttributeBuffer("rv_Position", GL_FLOAT, 0, 3);
   ```
3. Activer l'attribut `rv_Position`

```
m_program.enableAttributeArray("rv_Position");
```

1. Faire la même chose pour l'attribut `rv_color` du VS. Mais cette fois la couleur commence dans le VBO après les positions des 4 points, donc `m_program.setAttributeBuffer("rv_Color", GL_FLOAT, sizeof(QVector3D)*4, 3);`
2. Activer l'attribut `rv_color`.

```
m_program.enableAttributeArray("rv_Color");
```

1. Il faut ensuite passer la variable `u_ModelViewProjectionMatrix` qui est le produit de 3 matrices 4x4 :
   - la matrice `QMatrix4x4 model` qui place l'objet dans la scène. Dans notre cas nous allons appliquer à nos 4 points une translation de vecteur (0, 0, -3). Donc définie par `model.translate(0, 0, -3);`
   - la matrice `QMatric4x4 view` est celle qui passe du repère de la scène au repère de la vue, c'est à dire le repère où l'observateur est à l'origine et regarde dans la direction des z négatifs. Pour nous cette matrice sera la matrice Identité. Donc il n'y a rien à faire car le constructeur par défaut crée une matrice identité.
   - la matrice `QMatrix4x4 proj` est celle qui calcule la projection perspective de la 3D vers 2D. Elle a besoin pour être définie de :
     - l'*ouverture angulaire verticale* de l'objectif en degré (appelé *field of view* en anglais, abrégé en fov) : choisir 45°
     - le *format* de l'image, c'est à dire le rapport largeur/hauteur : on choisit 1.33f c'est à dire 4/3
     - les deux arguments suivants définissent la *profondeur de vue* : c'est à dire, le long de l'axe z, les distances minimales et maximales des objets qui vont être pris en compte pour le rendu. Tous les objets avant zmin et après zmax seront donc ignorés (on dit que c'est deux valeurs définissent les *plans de fenétrage* avant et arrière) : on prendra zmin à 0.1f et zmax à 100.0f.
       En définitive cela fait :

     ```
     proj.perspective(45, 1.33f, 0.1f, 100.0f);
     ```

   C'est la méthode `setUniformeValue` de `QtOpenGLShaderProgram` qui affecte la matrice résultante à la variable uniforme `u_ModelViewProjectionMatrix` avec

   ```
   m_program.setUniformValue("u_ModelViewProjectionMatrix", matrix);
   ```
2. On est prêts à lancer la commande de rendu, maintenant que le VBO et les shaders ont été définis, que l'on a fait le lien entre VBO et VS et qu'on a initialisé les variables uniformes du es shaders. La commande openGL pour lancer le rendu est `glDrawArrays`. Ses 4 arguments sont :
   - le type de primitive à construire avec les vertices,
   - à quel indice il faut commencer à lire les vertices dans le VBO,
   - combien de vertex il faut lire dans le VBO.

   ce qui donne dans notre cas `glDrawArrays(GL_TRIANGLES, 0, 4);`

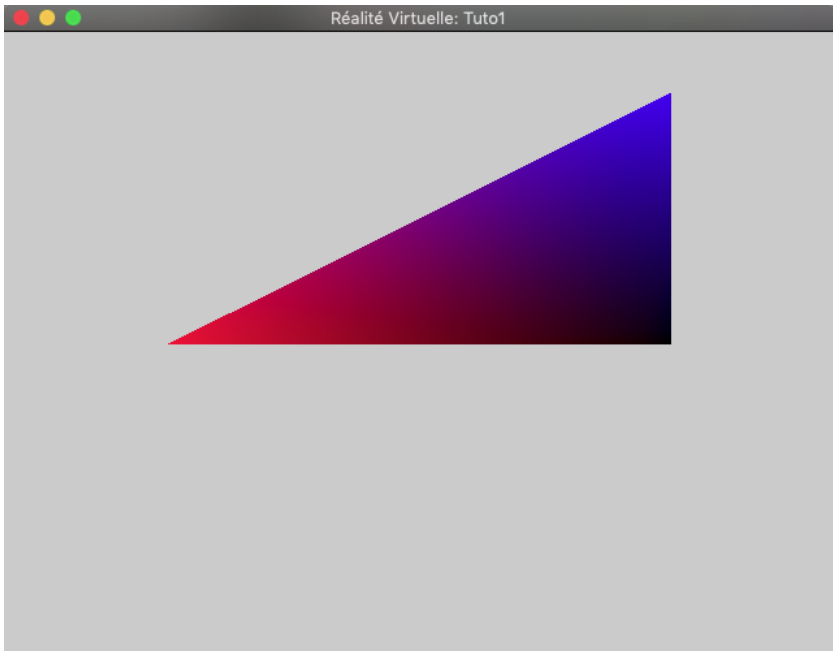
3. A la fin de `paintGL()` il faut libérer shader et VBO.
4. Il reste à définir la transformation qui amène le buffer d'image sur le widget, qui s'appelle le `viewport`. On met ce code dans la méthode `resizeGL()` car on doit tenir compte dans le viewport de la taille du widget : donc on doit être informés dans le cas où cette taille change. Donc il faut ajouter

```
glViewport(0, 0, w, h);
```

Voici le code final de ces deux méthodes

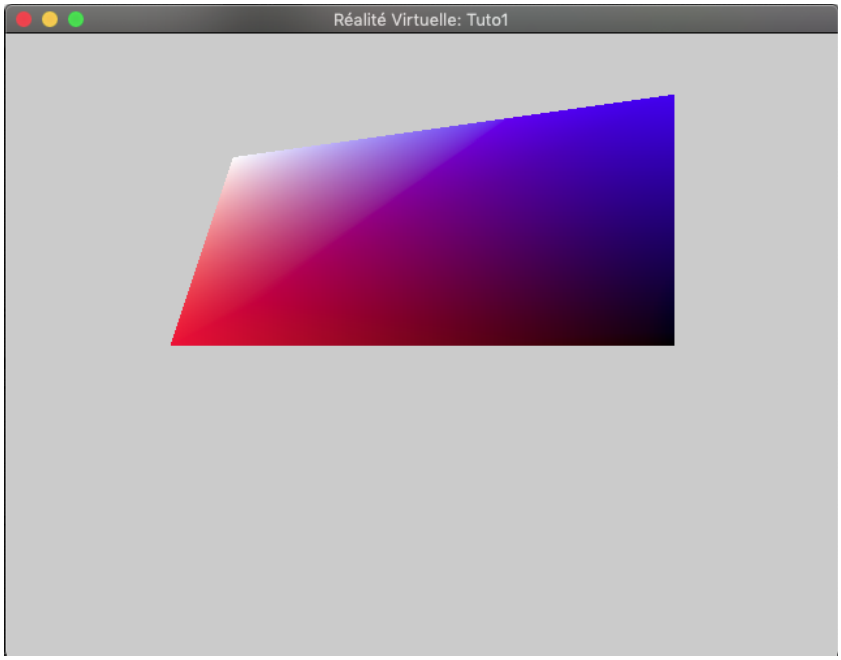
```
void RVWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    m_program.bind();
    m_vbo.bind();

    m_program.setAttributeBuffer("rv_Position", GL_FLOAT, 0, 3);
    m_program.enableAttributeArray("rv_Position");

    m_program.setAttributeBuffer("rv_Color", GL_FLOAT, sizeof(QVector3D)*4, 3);
    m_program.enableAttributeArray("rv_Color");

    QMatrix4x4 model, proj, view, matrix;

    //Définition de la matrice de projetcion
    proj.perspective(45, 1.33f, 0.1f, 100.0f);

    //Définiion de la matrice de vue
    view  = QMatrix4x4();

    //Définition de la matrice de placement
    model.translate(0, 0, -3);

    //Produit des trois matrices (dans ce sens!)
    matrix = proj * view * model;

    m_program.setUniformValue("u_ModelViewProjectionMatrix", matrix);

    glDrawArrays(GL_TRIANGLES, 0, 4);

    m_vbo.release();
    m_program.release();

}

void RVWidget::resizeGL(int w, int h)
{
    //transformation de viewport
    glViewport(0, 0, w, h);
}
```

Et le résultat que l'on obtient si on lance la compilation :



Ce n'est pas ce à quoi on s'attendait puisqu'on a mis 4 sommets on voulait voir 4 sommets : or ici on ne voit que les 3 premiers : le dernier (qui est blanc) n'apparaît pas ! Pourquoi ?

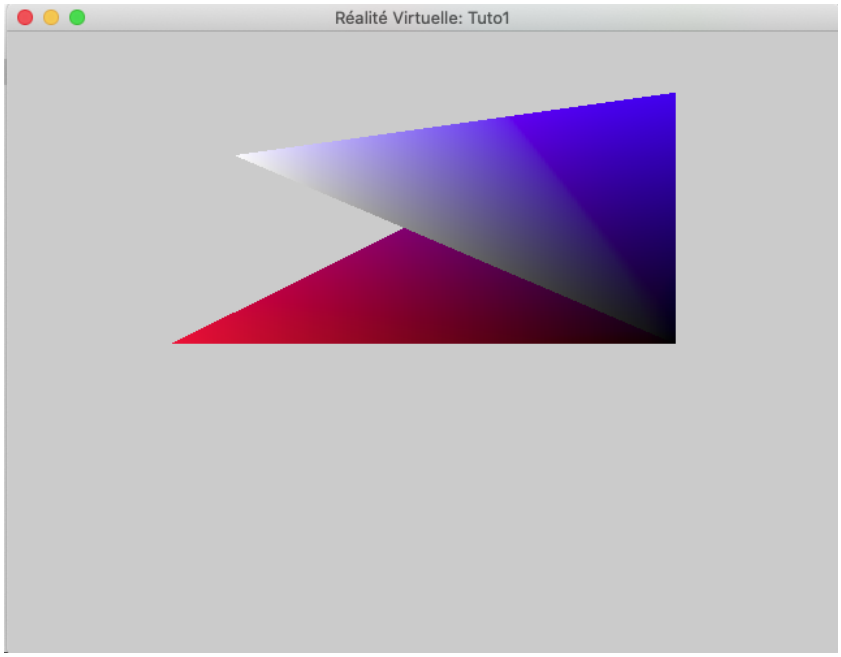
La réponse vient du choix de primitive dans `glDrawArrays` : la primitive `GL_TRIANGLES` construit des triangles en prenant les sommets **3 par 3**. Nous ne lui avons passé que 4 sommets odnc pas assez pour construire le 2ème triangle. IL aurait fallu passer 6 sommets.
La primitive `GL_TRIANGLE_FAN` (éventail de triangle) permet de construire n-2 triangles avec n sommets : le premier triangle avec les 3 premiers, le second avec le sommet 1, 3 et 4, puis 1, 4 et 5... etc.. Le premier sommet est commun à tous les triangles (comme dans un éventail justement). En modifiant la primitive le résultat obtenu est le suivant :



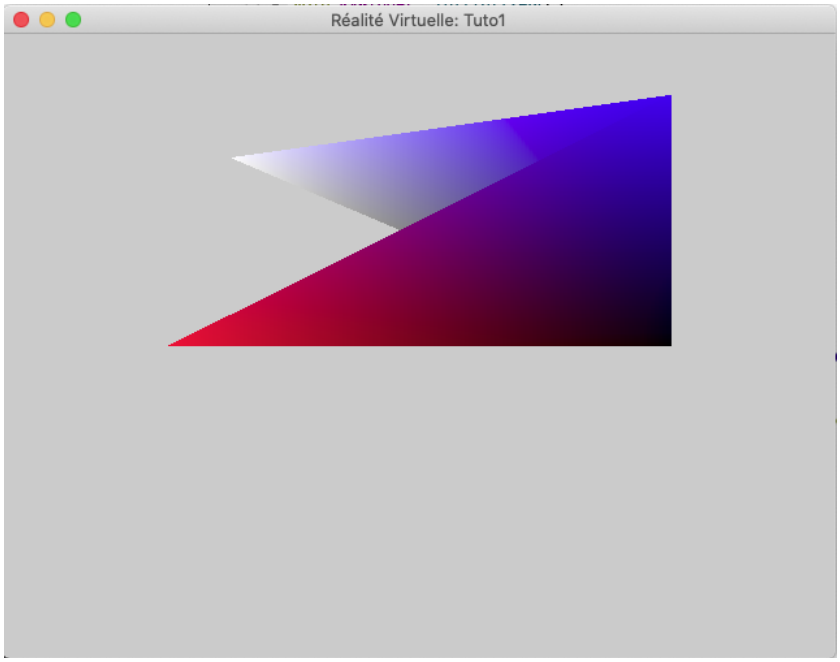
On voit finalement le sommet blanc et on voit aussi qu'il est en quelque sorte "derrière"

Variantes

On aurait pu utiliser aussi la primitive `GL_TRIANGLE_STRIP` qui construit une bande de triangle, en ajoutant à chaque nouveau vertex un nouveau triangle formé par lui et les deux derniers sommets du VBO. Dans notre cas le résultat n'est pas très joli.

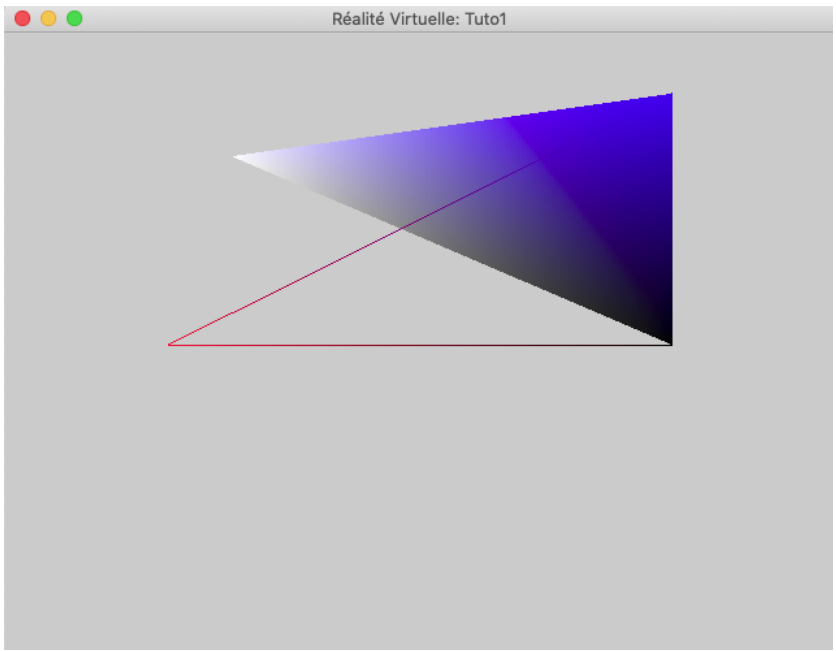


Il y a donc deux triangles : le premier formé par les sommets A, B, C, le second formé par les sommets B, C, D. C'est le principe du ruban de triangles. En revanche ce qui n'est pas logique c'est que le second triangle apparaisse devant le premier alors que le 4ème point (D) est *derrière !*. La raison vient du fait qu'on n'a pas activé parmi les réglages du contexte de rendu, le fait que l'on veut activer le calcul de visibilité. OpenGL met en place l'algorithme du z-buffer (voir ici) ou *tampon de profondeur* seulement si on lui demande. Pour l'avoir il faut le demander : donc au début de `paintGL()` ajouter `glEnable(GL_DEPTH_TEST);`
Le résultat est :

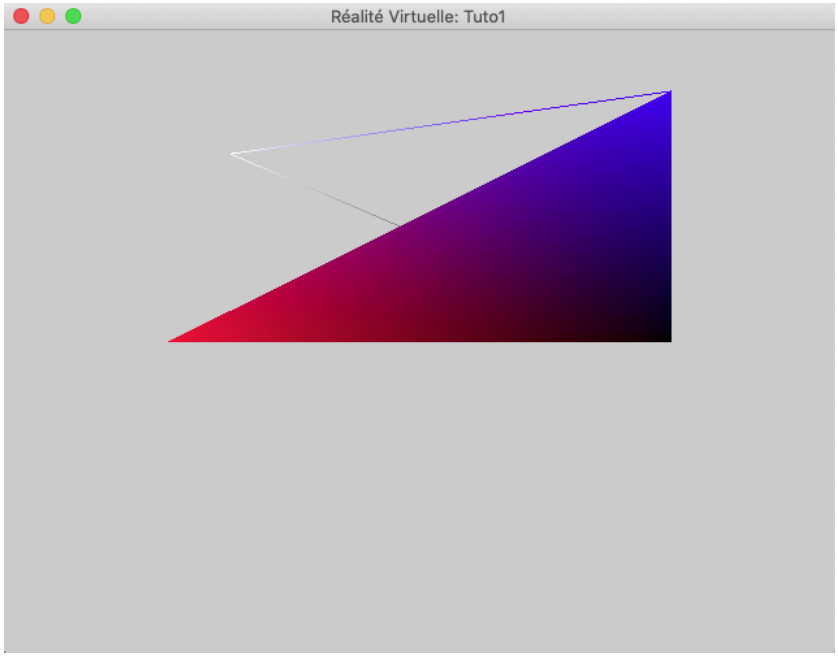


Avant - Arrière ?

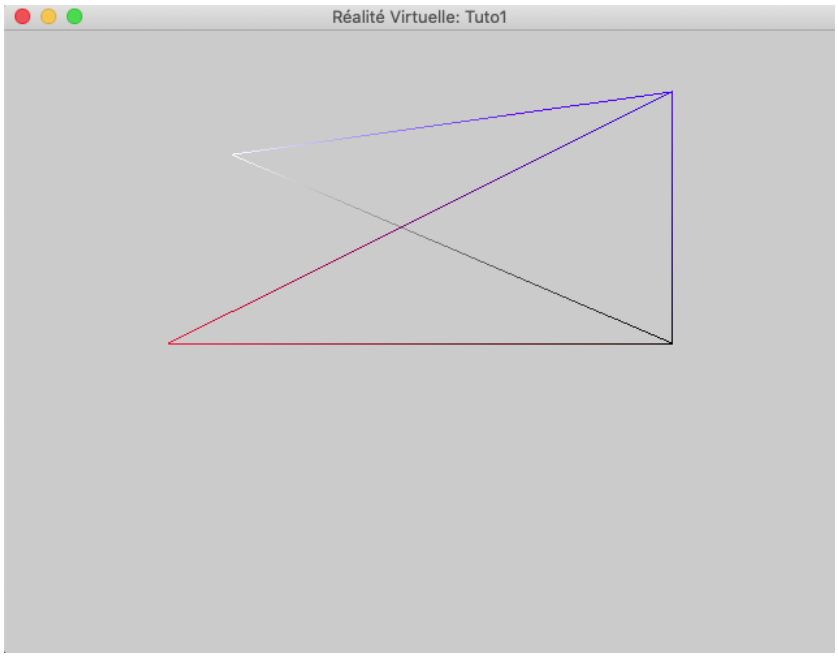
Parmi les différents réglages du contexte de rendu il y en a un qui est utile pour voir les triangles "nus" sans la couleur des fragments. Il suffit de définir comment on veut que les polygones soient rendus avec `glPolygonMode()`. Ses arguments sont le choix de la face à affecter (face avant ou face arrière ou les deux) et le type de rendu (les choix possibles sont `GL_FILL, GL_LINE, GL_POINT`).
Si vous essayez `glPolygonMode(GL_FRONT, GL_LINE);` vous aurez :



Pour comprendre le résultat, il faut bien saisir la notion de **avant** et **arrière** d'un triangle. Car clairement le premier triangle nous présentait sa face **avant** puisque maintenant il est rendu par des lignes (ce que nous avons demandé) alors que le second nous présente sa face **arrière** puisqu'il n'a pas été affecté par la commande qui concerne seulement la face avant. D'ailleurs en mettant `GL_BACK` à la place de `GL_FRONT` on inverse la situation.



Et en mettant `GL_FRONT_AND_BACK` on a finalement ce que l'on veut !



Donc comment est défini le devant et l'arrière d'un triangle ?
Par défaut, le devant d'un triangle est le coté duquel l'ordre des sommets dans le rendu (donc dans le VBO) tourne en sens anti-horaire (le sens positif de la trigonométrie).

- Le premier triangle est ABC : on voit les sommets tourner dans le sens positif donc on voit la face avant.
- le second triangle est BCD : les sommets tournent dans le sens des aiguilles d'une montre donc dans le sens négatif : on voit l'arrière du triangle BCD pas le devant.

Comme je l'ai dit c'est un réglage par défaut, que l'on peut modifier. La commande est `glFrontFace()`. Son argument est `GL_CCW` (pour *CounterClockWise*, par défaut) ou `GL_CW`.
Remarquez que dans le cas de l'éventail de triangles, les deux triangles ABC et ACD nous montraient tous les deux leur face avant.

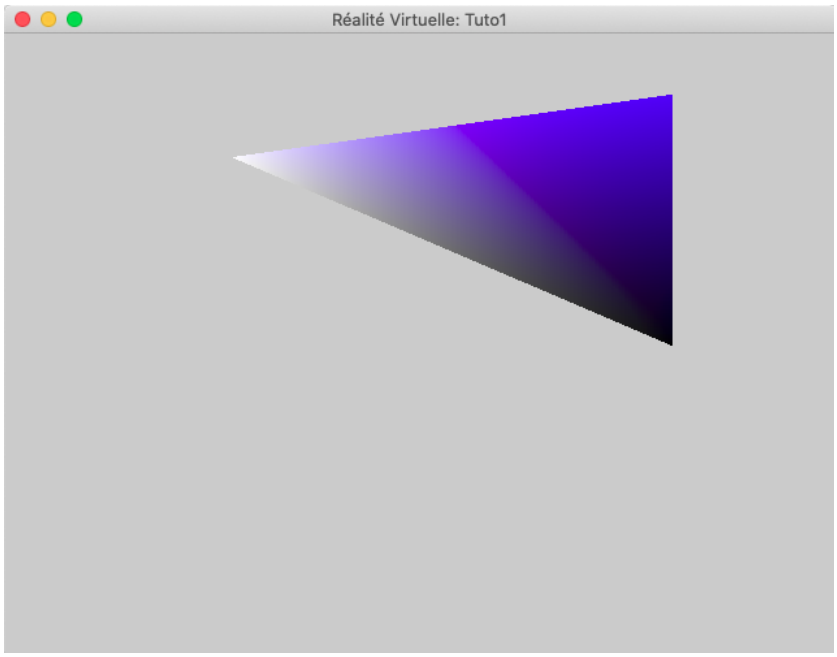
Culling

Cette notion de face avant et face arrière d'un triangle est importante en relation d'un autre concept du rendu 3D qui est le *culling*. Le culling est le fait d'éliminer (d'ignorer) lors du rendu certains triangles selon l'orientation de leur face. C'est ue technique qui permet d'accélérer le rendu en diminuant (souvent drastiquement) le nombre de triangles à traiter. En général, si on a un maillage qui représente une surface fermée (par exemple un personnage 3D) la moitié des triangles n'est pas visible car elle est *derrière*.
Par défaut le culling n'est pas activé. Pour activer le culling la commande est `glEnable(GL_CULL_FACE);` et pour le désactiver `glDisable(GL_CULL_FACE);`.

Ensuite il faut spécifier si le culling (l'élimination) concerne les triangles vus de face ou vus de derrière (par défaut c'est vu de derrière). Par exemple `glCullFace(GL_BACK);` signifie (si le culling est par ailleurs activé) qu'il faut ignorer tous les triangles qui ne sont pas face à l'observateur (et la notion de face et arrière dépend du réglage de `glFrontFace` que l'on a vu plus haut).
Cette situation est typique d'OpenGL : beaucoup de réglages, qui dépendent les uns des autres et surtout qui restent actifs tant qu'ils ne sont pas spécifiquement modifiés à nouveau : c'est le fonctionnement d'une *state machine*.
Par exemple

- si la définition de la face avant est `GL_CCW`,
- si le culling est activé,
- si les faces à ignorer est `GL_FRONT`,
- si le mode de polygone `GL_FILL` pour les faces avant et arrière,
- et si les primitives à rendre est `GL_TRIANGLE_STRIP`,

alors on demande à OpenGL d'ignorer les faces dont les sommets tournent dans le sens positif et le rendu est



Mais on aurait obtenu exactement la même chose en disant que la face avant est `GL_CW` et qu'il faut ignorer la face `GL_BACK`.

Index Buffer

On peut être agacés du fait que le résultat du rendu dépende tant de l'ordre dans lequel les sommets sont stockés dans le VBO. On aimerait utiliser les sommets dans l'orde que l'on veut pour construire les triangles, et même utiliser les mêmes sommets pour plusierus triangles. Par exemple avec nos 4 points A,B,C,D non planaires, on peut bien définir 4 triangles, non ?
La solution offerte par OpenGL consiste à définir un tableau d'indices (c'est le *Index Buffer Object* abrégé en **IBO**) dans lequel on fait référence à l'index de chaque sommet dans le VBO. Par exemple (0, 3, 2), dans notre exemple définirait le triangle ADC.
C'est ce que OpenGL appelle le *rendu indexé* : on doit définir un IBO en plus du VBO et utiliser la commande `glDrawElements` au lieu de `glDrawArrays` pour lancer le rendu.

Utiliser un Index Buffer Object

Pour montrer comment cela fonctionne, je vais dupliquer le projet `tuto1` en `tuto1.2` (les deux projets sont disponibles dans Moodle).

1. on ajoute à `RVWidget` une variable membre `m_ibo` de type `QOpenGLBuffer` ,
2. dans `initializeBuffer()` on initialise `m_ibo` avec `m_ibo = QOpenGLBuffer(QOpenGLBuffer::IndexBuffer);` (on spécifie que l'on demande la création d'un index buffer et pas d'un vertex buffer)
3. on prépare un tableau d'entiers non signés (*unsigned int*) avec les numéros des sommets à utiliser

```
uint indexData[] = {
    0, 1, 2,
    0, 2, 3,
    0, 3, 1,
    2, 1, 3
};
```

1. on crée le ibo, on lui alloue les données exactement de la même façon dont on a fait avec le vertex buffer.

```
//Initialisation de l'Index Buffer Object
m_ibo = QOpenGLBuffer(QOpenGLBuffer::IndexBuffer);
//Création du VBO
m_ibo.create();
//Lien du VBO avec le contexte de rendu OpenG
m_ibo.bind();
//Allocation des données dans le VBO
m_ibo.allocate(indexData, sizeof (indexData));
m_ibo.setUsagePattern(QOpenGLBuffer::StaticDraw);
//Libération du VBO
m_ibo.release();
```

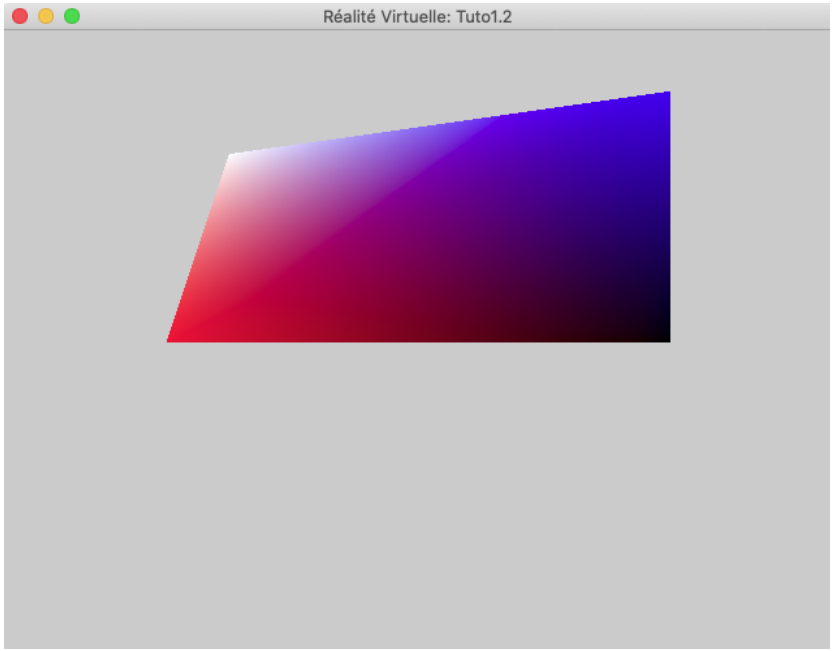
Rendu indexé

On peut maintenant, dans `paintGL()`, remplacer le rendu direct du vertex buffer, par un *rendu indexé*, c'est à dire piloté par l'index buffer :

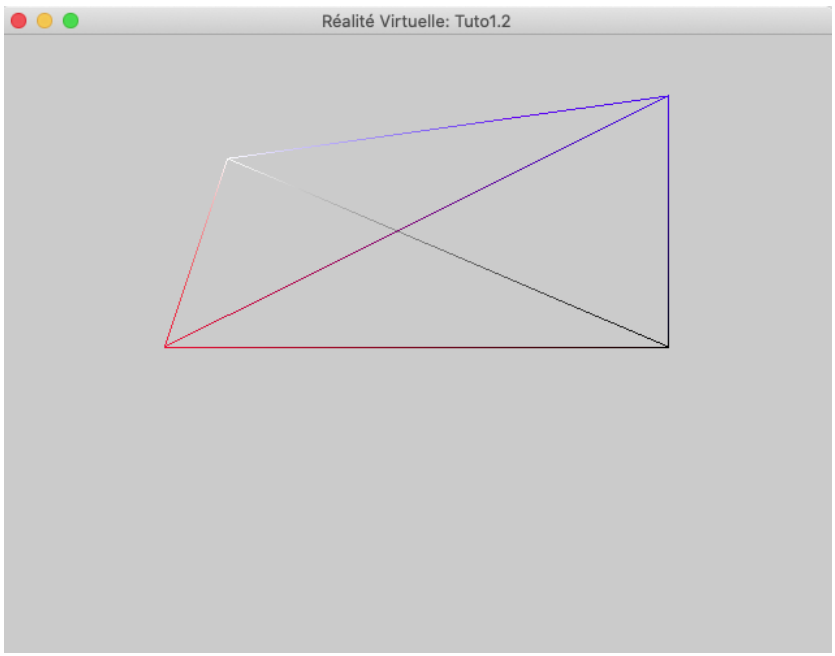
1. après avoir *lié* le vertex buffer au contexte de rendu, il faut maintenant aussi lier l'index buffer `m_ibo`,
2. au lieu d'utiliser la commande `glDrawArrays` pour le rendu, j'utilise maintenant `glDrawElements`. Ses arguments sont :
   - le type de primitive à construire : cette fois `GL_TRIANGLES` convient très bien car nous donnons 4 groupes de 3 sommets,
   - le nombre d'index à lire dans l'IBO : ici 12,
   - le type de variable utilisé pour les index,
   - le dernier est un pointeur nul (utilisé seulement si on passe le tableau lors de l'appel au lieu d'utiliser un tableau qui est stocké dans un index buffer, ce qui est l'option plus moderne et plus rapide)

```
glDrawElements(GL_TRIANGLES, 12 ,GL_UNSIGNED_INT, nullptr);
```

Le résultat est



Pour voir qu'on a bien 4 faces triangulaires, on peut demander un rendu en *fil de fer*, ce qui donne :



Ou bien faire tourner mon objet pour l'observer par derrière :