

### Introduction à OpenGL un cube qui tourne.

Projet de base : exemple4

1. **Mise en place** : dans **Qt Creator**, créez un nouveau projet de type **Application Qt avec widgets** appelé **TP1** avec une classe de base **MainWindow** qui hérite de **QMainWindow** avec interface graphique.
  - a. Copier les fichiers *widgetopengl.h* et *widgetopengl.cpp* qui définissent la classe **WidgetOpenGL** issus de l'exemple4 du cours dans le répertoire du projet puis ajouter ces fichiers au projet avec **Clic-droit/Ajouter des fichiers existants...**
  - b. Copier aussi les programmes des shaders *simple.vert* et *simple.frag* depuis l'exemple4 dans le répertoire du projet. On va ajouter ces deux fichiers en tant que ressources : **Clic-droit/Ajouter Nouveau...** puis **Qt/QtRessource File** nommé **Ressources**. Dans le nouveau fichier de ressource *ressources.qrc* un **Clic-droit/Ajouter des fichiers existants** puis choisir *simple.vert* et *simple.frag*. Modifier enfin le **préfixe** des deux fichiers en *shaders*. De cette façon (comme dans exemple4) ces deux fichiers sont accessibles par le chemin «:/shaders/simple.frag» mais en fait seront intégrés à l'exécutable.
  - c. Librairie mathématique : télécharger sur [glm.g-truc.net](http://glm.g-truc.net) la librairie glm-0.9.8.3.zip et décompressez-la à l'endroit où vous aller placer les projets de ce cours. Dans TP1.pro ajouter le bon répertoire d'*include* avec (par exemple) :

```
INCLUDEPATH += $$PWD/./glm
```

- d. Sur Windows il faut aussi ajouter dans le fichier TP1.pro

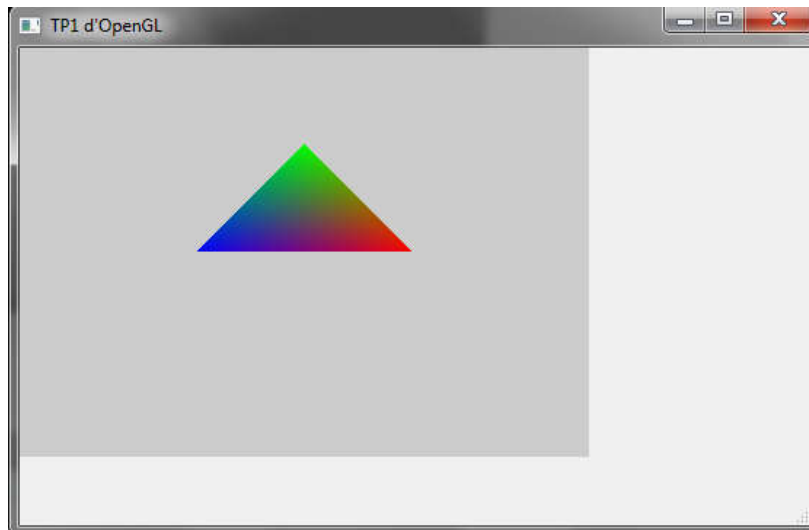
```
windows:LIBS += -lopengl32
```

- e. Modification du main : il faut définir la surface de rendu comme une surface qui fonctionne avec OpenGL 3.3+ : ajouter dans main.cpp le code suivant entre la définition de l'instance de **QApplication** et celle de **QWindow**

```
QSurfaceFormat format;  
format.setRenderableType(QSurfaceFormat::OpenGL);  
format.setDepthBufferSize(24);  
format.setVersion(3, 3);  
format.setProfile(QSurfaceFormat::CoreProfile);  
QSurfaceFormat::setDefaultFormat(format);
```

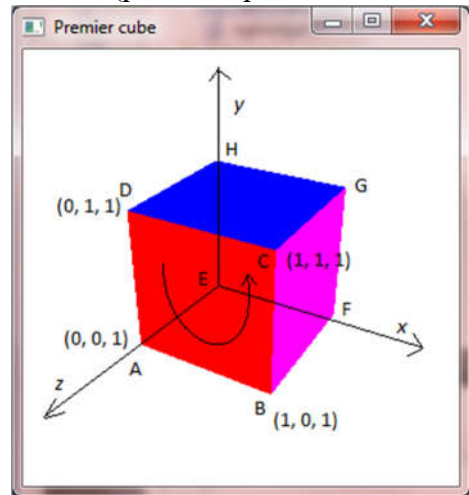
Il faut aussi ajouter un include de **QSurfaceFormat**

- f. Allez en mode de création d'interface graphique en cliquant sur le formulaire *mainwindow.ui*. Placez dans la fenêtre principale un **OpenGL Widget** (dans les display widgets), ajustez sa taille puis en faisant un clic-droit dessus choisir **Promouvoir en...** puis **WidgetOpenGL**. Compilez le projet : vous devriez voir le triangle de l'exemple 4 dans un widget dans une fenêtre.
  - Vous pouvez modifier le nom de la fenêtre via la propriété **WindowTitle** de **MainWindow**
  - Et faire disparaître la barre d'outils en l'enlevant de l'**Object Inspector**



2. **Cube** : on veut remplacer le triangle par un cube de côté 1 (dont un sommet est à l'origine) avec chaque face d'une couleur primaire différente (par exemple la face avant ici est rouge et la face droite est magenta) à l'aide de VBO et VAO (exemple4 sert de base)

- Dans la classe **WidgetOpenGL** on modifie la méthode *initializeVAO* qui va se charger de créer et remplir le VBO. Dans ce vertex buffer il faudra mettre 24 sommets (4 par face) et 24 couleurs : on définit donc 8 **glm::vec3** appelés A, B, C, D, E, F, G, H correspondant aux 8 sommets du cube (utilisez le dessin) et 6 autres **glm::vec3** appelés rouge, vert, bleu, cyan, magenta, jaune pour les 6 couleurs primitives.
- Ensuite **vertex\_data** devient maintenant un tableau de **glm::vec3** dans lequel vous mettrez les 4 sommets pour chacune des six faces suivis par les couleurs de chacun des 24 sommets. Attention : pour chaque face, il faut donner les sommets en tournant dans le sens contraire des aiguilles d'une montre, **vu de l'extérieur du cube** ! Sinon c'est la face interne du cube qui sera rendue et pas celle extérieure.



```
glm::vec3 vertex_data[] = {
    A, B, C, D, //face avant
    ....
};
```

- Toujours dans *initializeVAO*, la seule chose à modifier c'est la commande **setAttributeArray** correspondant à la couleur (index 1) pour qui il faut donner la position (en octet) où commence l'information de couleur dans le VBO !
- Dans **paintGL()** on fait le rendu du VBO comme dans l'exemple 4 mais cette fois on a 24 sommets et on veut dessiner 6 carrés ; deux solutions sont possibles
  - On utilise la primitive **GL\_QUADS** dans *glDrawArrays* : OpenGL va lire les 24 sommets en les regroupant 4 par 4 pour construire des quadrilatères ; c'est simple mais la primitive **GL\_QUADS** est **dépréciée** et donc certaines cartes graphiques ne l'accepteront pas en mode *Core Profile*.
  - Dans une boucle, on utilise 6 fois la primitive **GL\_TRIANGLE\_FAN** qui



va lire 4 sommets et construire deux triangles qui forment le carré.

Il faut aussi modifier le calcul de la matrice `m_matrix` qui transforme les sommets :

- **model** est une composition d'une translation de vecteur  $(-0.5, -0.5, -0.5)$ , qui met le centre du cube à l'origine du repère, avec une rotation de  $30^\circ$  autour de l'axe y et une rotation de  $30^\circ$  autour de l'axe des x. Ceci est l'ordre logique dans lequel les transformations vont être appliquées, donc dans la définition de la matrice **model** il faut les définir dans l'ordre **inverse**.
- **vue** est une translation de -4 le long de l'axe des z,
- **proj** ne change pas

3. **Animation** : pour faire tourner le cube autour de l'axe des y on va créer un **Timer Qt** qui va envoyer un signal tous les 100èmes de seconde à un slot qui va modifier l'angle de rotation.

- a. Ajouter à la classe **WidgetOpenGL** un *slot* appelé **update()**, un pointeur sur un **QTimer** appelé **m\_timer** et un **GLfloat** **m\_angleY** (variables membres privées).

```
private slots:
    void update();

private:
    QTimer* m_timer;
    GLfloat m_angleY;
```

- b. Dans le constructeur on met **m\_angleY** à zéro et on initialise **m\_timer**
- c. Dans **initializeGL** on crée la connexion entre le signal **timeout** de **m\_timer** et le slot **update** de la classe (**this**) ; on lance ensuite le **timer** en appelant sa méthode **start** avec en argument le nombre de millisecondes entre deux **timeout**.

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

- d. Dans la méthode **update** on incrémente l'angle (de 4 degrés par exemple) et on appelle **QOpenGLWidget::update** qui force le réaffichage du contenu de la fenêtre.
- e. Enfin dans **paintGL** il faut modifier la matrice **view** pour qu'elle utilise **m\_angleY** comme angle de rotation par rapport à l'axe des y.

4. **Manipulation à la souris** : au lieu de faire tourner le cube tout seul, on aimerait pouvoir faire tourner le cube à la souris. Pour cela il faut ajouter dans la classe principale les méthodes qui sont appelées quand le bouton de la souris est pressé (**mousePressEventQMouseEvent\***) et quand la souris bouge (**mouseMoveEventQMouseEvent\***). Ensuite :

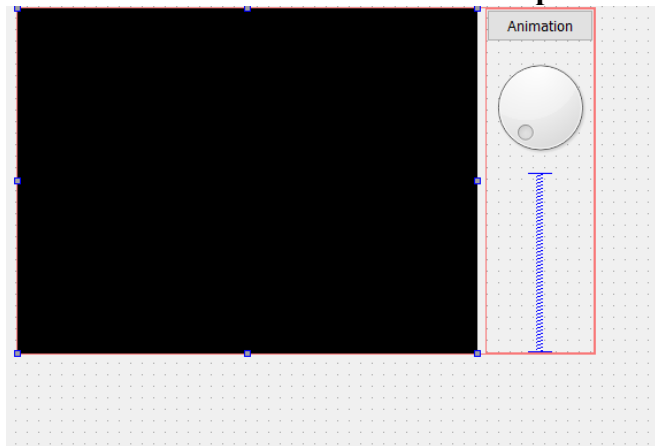
- a. on va compléter la variable membre **m\_angleY** par 2 variables membres **m\_angleX**, **m\_angleZ** (initialisées à 30, 30, 0) et dans **paintGL** pour définir la matrice **view** on met les trois rotations (dans l'ordre X, Y, Z) ;
- b. on ajoute une variable membre de type **QPoint** **lastPos** dans laquelle la méthode **mousePressEvent** stocke la position du curseur de la souris (accessible par **event→pos()**) ;
- c. dans **mouseMoveEvent** :
- on calcule **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 si la bouton droit de la souris (**event→buttons()**) contient cette

info) est pressé on *incrémente* **m\_angleX** et **m\_angleY** *respectivement* de **dy\*180** et **dx\*180** ;

- si par contre c'est le bouton gauche de la souris qui est pressé on *incrémente* **m\_angleX** et **m\_angleZ** *respectivement* de **dy\*180** et **dx\*180** ;
- ensuite on appelle **update** et on stocke la nouvelle valeur de **lastPos**.


5. **Interaction avec le widget OpenGL** : Qt permet, via le mécanisme des signaux et des slots de relier entre eux les composants graphiques qui composent l'interface graphique. Par exemple on peut ajouter un bouton pour activer/arrêter la rotation automatique et un autre pour changer le paramètre « field of view » de la matrice de projection:

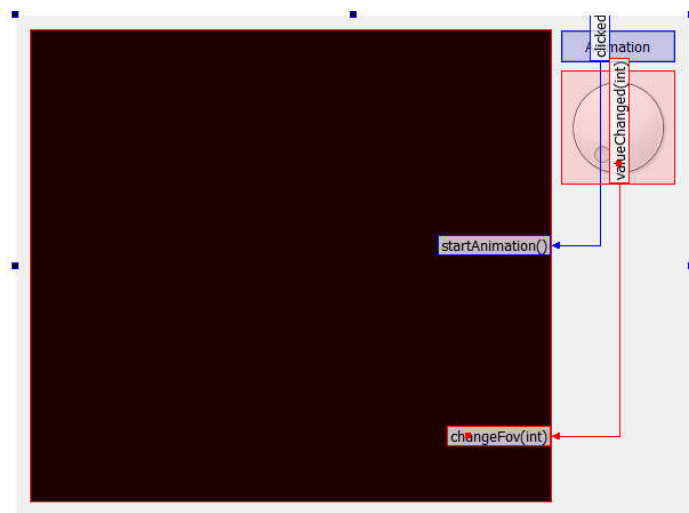
- Dans le concepteur graphique ajouter à droite du **WidgetOpenGL** un **PushButton** nommé **Animation**, un **QDial** appelé **fov** et un **Vertical Spacer**. Sélectionner les 3 composants et choisissez le bouton « **Mettre en place verticalement** ». Ceci crée un **Vertical Layout** qui contient les trois composants. De même sélectionnez le **WidgetOpenGL** et le **Vertical Layout** et organisez-le avec « **Mettre en place horizontalement** » (il faut peut-être donner une dimension **minimumSize** à **OpenGLWidget** – par exemple 400x300)



Objet	Classe
MainWindow	QMainWindow
centralWidget	QWidget
horizontalLayout	QHBoxLayout
verticalLayout	QVBoxLayout
Animation	QPushButton
fov	QDial
verticalSpacer	Spacer
openGLWidget	WidgetOpenGL

Enfin pour faire en sorte que le **centralWidget** suive la taille de la fenêtre principale il faut le sélectionner et appuyer sur « **Mettre en place horizontalement** ». Tester ! Pour que lors du **resize** la fenêtre **OpenGL** soit toujours maximale (et pas les boutons) il faut que la **sizePolicy** de **WidgetOpenGL** soit **Expanding, Expanding**

- Passer en mode **Edit Signals/Slots** (F4)  et à la souris faire un drag depuis le bouton **Animation** vers le **WidgetOGL** : dans la boîte de dialogue qui apparaît choisir à gauche le signal **clicked()** du **QPushButton** et à droite cliquer sur **Editer..** et ajoutez un slot appelé **startAnimation()**. Validez et sélectionnez ce nouveau slot pour terminer l'édition de ce premier lien. De même associer à l'action **valueChanged(int)** du **QDial**, un slot **changeFov(int)** de



## WidgetOpenGL

- c. Revenir dans le code et ajouter les deux slots *protected* dans WidgetOpenGL.h. Dans WidgetOGL.cpp définir ces deux méthodes pour avoir le comportement désiré (avec l'ajout des variables membres nécessaires).
  - d. On peut affiner le comportement du **QDial** en imposant que les valeurs soient comprises entre 10° et 120° et que la valeur de départ soit 30°. Et on peut activer l'option **checkable** du bouton pour qu'il reste appuyé tant que l'on ne re-appuie pas dessus.
6. **Modifier - un peu - le vertex shader :**
- a. Transparence : actuellement dans le vertex shader la composante alpha de la couleur `out_color` est égale à 1 (opacité totale). On peut la remplacer par une nouvelle variable uniforme **u\_opacity** (de type float) qui va être modifiée par un slider. Pour que OpenGL utilise effectivement la composante alpha pour faire le mélange de couleur il faut rajouter dans *initializeGL* deux nouveaux réglages :

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

- b. Modifier la couleur : au lieu de recopier la couleur présente sur le vertex (`col`) directement sur la sortie (`out_color`) en lui ajoutant une opacité, on peut utiliser une fonction qui calcule la couleur en sortie à partir de la couleur en entrée. Par exemple pour avoir une image en niveau de gris, la C.I.E préconise pour les écrans la fonction suivante :

$$\text{gris} = 0.299 \text{ rouge} + 0.587 \text{ vert} + 0.114 \text{ bleu}$$

Et cette valeur gris doit être recopiée dans les 3 composantes `rgb` de `out_color` ; cela peut être obtenu simplement en multipliant via une matrice 3x3 qui sera une nouvelle variable uniforme du *shader*. D'autres matrices font apparaître les couches R, V, B, etc... Ces options pourraient être activées via des radio-boutons comme ci-dessous.

