# Intégration d'OpenGL dans une interface Qt

par Florentin Halgand (La page de Architekth)

Date de publication : 12/09/2010

Dernière mise à jour :

À travers ce tutoriel vous allez apprendre à intégrer simplement un widget OpenGL dans une application Qt.

N'hésitez pas à commenter cet article !

# I - Introduction

Qt propose un module OpenGL qui permet d'afficher des rendus OpenGL dans une fenêtre Qt. Toutes les versions de Qt disposent de ce module. Qt wrappe OpenGL c'est à dire que l'utilisation du module OpenGL ne varie pas selon les versions d'OpenGL. Vous pouvez ainsi mettre à jour vos en-têtes OpenGL sans souci de compatibilité avec Qt.

## I-A - De quelle version d'OpenGL Qt dispose-t-il ?

La version d'OpenGL varie selon les compilateurs.

## I-B - Qu'y a-t-il à apprendre dans ce tutoriel ?

Eh bien, il est évident que nous n'allons pas apprendre à utiliser OpenGL de façon poussée puisque le tutoriel porte essentiellement sur l'interaction entre Qt et OpenGL. OpenGL ne disposant pas d'interface graphique (fenêtres, boites de dialogue) en soi, cette bibliothèque permet seulement d'afficher des formes soumises à des transformations dans l'espace. Qt nous permettra donc de créer la fenêtre qui contiendra la zone d'affichage OpenGL. Je vous apprendrai à fermer la fenêtre en appuyant sur une touche, afficher la fenêtre en pleine écran, charger des textures, etc.

## I-C - En quoi utiliser OpenGL avec Qt peut-il être intéressant ?

Eh bien, Qt à l'avantage de conserver le côté multiplateforme d'OpenGL mais aussi simplifie certaines parties assez fastidieuses telles que le chargement d'une image pour l'appliquer en tant que texture ou même l'intégration des shaders. La (interface graphique) de Qt est très complète comparée à d'autres bibliothèques comme SDL et GLUT très utilisées avec OpenGL en général, ainsi elle permet la réalisation de logiciels tels qu'un éditeur de cartes pour votre jeu, modélisateur 3D, etc. Tous ces exemples sont autant de points forts qui justifient l'utilisation d'OpenGL avec Qt. Maintenant que vous savez de quoi le tutoriel traite, nous allons pouvoir nous plonger dans un nouveau projet Qt. L'IDE que j'utilise est QtCreator 2.0, si vous en utilisez un autre cela reste très similaire.

# II - Pré requis

Comme il a déjà été dit dans l'introduction, ce n'est pas un tutoriel sur OpenGL mais plutôt un tutoriel sur l'interaction entre Qt et OpenGL. Il est donc nécessaire que vous ayez des connaissances en OpenGL. Ce n'est pas non plus un tutoriel sur l'apprentissage de Qt, vous devez avoir déjà des connaissances sur Qt, comme créer les fenêtres, les signaux et les slots. Tout d'abord, commençez par créer un nouveau projet vide, vous y ajoutez un fichier source main.cpp comme pour tous vos autres projets Qt habituels avec le code minimal :

```
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    return app.exec();
}
```

Nous allons maintenant configurer le fichier .pro de Qt pour permettre au compilateur d'ajouter le module OpenGL en ajoutant cette ligne :

```
QT += opengl
```

Voilà, notre projet est configuré pour pouvoir traiter du code OpenGL avec Qt. Il est inutile de lancer le programme puisqu'il n'exécute pas de fenêtre.

# III - Première approche de QGLWidget

La classe **QGLWidget** est le widget de Qt permettant d'afficher des rendus OpenGL. L'utilisation reste simple et il suffira de créer une classe héritant de celle-ci. Cette classe est constituée notamment de trois fonctions très importantes que nous allons utiliser dans ce tutoriel.

- **paintGL()** : méthode appelée à chaque fois que le widget doit être mis à jour. Elle redessine la scène OpenGL.
- **resizeGL(int width, int height)** : méthode appelée à chaque fois que le widget est redimensionné. Cette méthode met à jour la scène OpenGL quand la fenêtre est redimensionnée.
- **initializeGL()** : méthode permettant d'initialiser les différents paramètres d'OpenGL. Cette méthode est appelée une fois au lancement de l'application avant même paintGL() et resizeGL(int width, int height).

## III-A - myGLWidget.h

Maintenant que nous avons vue et compris ces trois fonctions, nous allons pouvoir nous concentrer sur le code.

```
#ifndef MYGLWIDGET_H
#define MYGLWIDGET_H

#include <QtOpenGL>
#include <QGLWidget>

class myGLWidget : public QGLWidget
{
    Q_OBJECT
public:
    explicit myGLWidget(int framesPerSecond = 0, QWidget *parent = 0, char *name = 0);
    virtual void initializeGL() = 0;
    virtual void resizeGL(int width, int height) = 0;
    virtual void paintGL() = 0;
    virtual void keyPressEvent( QKeyEvent *keyEvent );

public slots:
    virtual void timeOutSlot();

private:
    QTimer *t_Timer;

};


#endif // MYGLWIDGET_H
```

Nous allons nous concentrer sur les parties du code importantes que nous allons détailler afin de bien les comprendre.

```
#include <QtOpenGL>
#include <QGLWidget>
```

Ces inclusions sont nécessaire pour pouvoir utiliser le module OpenGL de Qt.

```
class myGLWidget : public QGLWidget
```

Notre classe va hériter de **QGLWidget** car elle servira au rendu OpenGL dans une fenêtre Qt.

```
explicit myGLWidget(int framesPerSecond = 0, QWidget *parent = 0, char *name = 0);
```

Quelques explications s'imposent, tout d'abord framesPersecond représente le nombre d'images par seconde que l'utilisateur souhaite afficher, cela permettra de définir le temps, entre chaque image, rendu par OpenGL pour éviter de surcharger votre processeur et votre carte graphique inutilement. Ensuite parent est un pointeur sur le widget

qui contiendra notre objet myGLWidget. Le dernier paramètre name concerne tout simplement le titre de la fenêtre affiché en haut de celle-ci.

```
virtual void initializeGL() = 0;
virtual void resizeGL(int width, int height) = 0;
virtual void paintGL() = 0;
```

Remarquons que ces fonctions sont virtuelles pures. Viendra plus tard une classe dérivant de myQGLWidget pour afficher notre scène OpenGL, mais nous y reviendrons dans le chapitre suivant.

```
virtual void keyPressEvent( QKeyEvent *keyEvent );
```

Voici une fonction importante. Son rôle ? Recevoir les informations sur les touches du clavier pressées par l'utilisateur. Il y a encore beaucoup plus à dire sur elle, nous y viendrons avec son implémentation dans le fichier myGLWidget.cpp.

```
public slots:
    virtual void timeOutSlot();

private:
    QTimer *t_Timer;

};
```

Le timer est nécessaire à notre application pour éviter de surcharger notre processeur et notre carte graphique. L'utilisation d'un **QTimer** facilite la tâche. Notre processeur travaille tant qu'on lui donne des informations à calculer, donc sans timer, notre scène OpenGL pourra avoir un nombre d'images par seconde très élevé inutilement. Notre il ne perçoit pas la différence entre 60 images par seconde (ips) et 999 images par seconde. Ainsi il est inutile de demander à notre processeur et notre carte graphique d'afficher 999 ips voir plus (ce n'est qu'un exemple) alors que 60 ips suffiront largement. Nous choisirons 60 ips puisqu'en dessous on perçoit une légère latence qui fatigue les yeux à la longue mais rien ne nous empêche de définir une valeur différente. Pour le moment ces informations ne sont données qu'à titre indicatif puisque nous allons définir cette valeur dans le chapitre suivant.

## III-B - myGLWidget.cpp

Le code complet est le suivant :

```
#include "myGLWidget.h"

myGLWidget::myGLWidget(int framesPerSecond, QWidget *parent, char *name)
    : QGLWidget(parent)
{
    setWindowTitle(QString::fromUtf8(name));
    if(framesPerSecond == 0)
        t_Timer = NULL;
    else
    {
        int seconde = 1000; // 1 seconde = 1000 ms
        int timerInterval = seconde / framesPerSecond;
        t_Timer = new QTimer(this);
        connect(t_Timer, SIGNAL(timeout()), this, SLOT(timeOutSlot()));
        t_Timer->start( timerInterval );
    }
}

void myGLWidget::keyPressEvent(QKeyEvent *keyEvent)
{
    switch(keyEvent->key())
    {
        case Qt::Key_Escape:
            close();
            break;
    }
}
```

```
void myGLWidget::timeOutSlot()
{
}
```

Comme dans la partie précédente, détaillons le code pour mieux le comprendre.

```
myGLWidget::myGLWidget(int framesPerSecond, QWidget *parent, char *name)
    : QGLWidget(parent)
{
    setWindowTitle(QString::fromUtf8(name));
```

La fonction **setWindowTitle()** permet de définir le nom de la fenêtre qui sera le paramètre name.

```
    if(framesPerSecond == 0)
        t_Timer = NULL;
    else
    {
        int seconde = 1000; // 1 seconde = 1000 ms
        int timerInterval = seconde / framesPerSecond;
        t_Timer = new QTimer(this);
        connect(t_Timer, SIGNAL(timeout()), this, SLOT(timeOutSlot()));
        t_Timer->start( timerInterval );
    }

}
```

À cet endroit, un test est effectué sur la valeur de framesPerSecond : s'il vaut 0, ce qui est la valeur par défaut, alors notre **QTimer** t_Timer aura null pour valeur. Sinon, nous calculons l'intervalle entre deux images puis nous enregistrons cette valeur dans timerInterval, ensuite nous créons notre timer puis connectons le signal **timeout()** au slot personnalisé timeOutSlot(). Pour finir nous démarrons le timer avec la valeur de timerInterval. Le timer se met ainsi en marche, contrôlant le temps entre deux images rendu par OpenGL.

```
void myGLWidget::keyPressEvent(QKeyEvent *keyEvent)
{
    switch(keyEvent->key())
    {
        case Qt::Key_Escape:
            close();
            break;
    }
}
```

La fonction **keyPressEvent()** va nous permettre comme dit précédemment de détecter l'appui sur une touche du clavier. Ainsi nous allons pouvoir traiter les événements relatifs au clavier.

Cette fonction reçoit en paramètre un **QKeyEvent** qui correspond à l'événement du clavier. Nous effectuons un switch qui nous permettra de traiter les événements selon leur nature, ici nous nous contenterons de fermer la fenêtre lors d'un appui sur la touche  Echap .

```
void myGLWidget::timeOutSlot()
{
}
```

Ici notre fonction timeOutSlot() qui est appelée par le timer est vide pour le moment. On y ajoutera du code dans le chapitre suivant.

Jusqu'ici, rien de fonctionnel n'a été mis en place. Cependant, toute cette partie est requise, elle pose les bases sur lesquelles tout l'affichage s'appuie. Après cette longue partie théorique, passons à du plus ludique avec l'affichage de votre premier polygone avec OpenGL et Qt !

Notre classe myGLWidget va être elle même dérivée dans la partie suivante. Pour quelle raison ? Eh bien cette classe sera la classe mère pour tous les autres chapitres. Ainsi on aura par défaut (dans cette classe) des fonctionnalités importantes qu'il n'est pas nécessaire de redéfinir à chaque nouvelle partie du tutoriel telles que la possibilité de changer de mode d'affichage (plein écran/fenêtré), ou de fermer l'application en appuyant sur la touche  Echap , etc.

## IV - Dessiner des objets OpenGL dans ma fenêtre

Ce chapitre va enfin nous amener vers la partie intéressante du tutoriel avec du résultat. Ajoutons deux fichiers à notre projet : myWindow.h et myWindow.cpp. Ces fichiers comporteront la classe myWindow dérivée de myGLWidget qui aura pour but d'afficher le rendu. Nous redéfinirons les fonctions virtuelles pures **initializeGL()**, **resizeGL()** et **paintGL()**. Dans ce chapitre, vous constaterez que l'utilisation d'OpenGL avec Qt reste identique pour ce qui est de la syntaxe, ainsi tout ce que vous avez appris sur le code strictement OpenGL vous sera indispensable pour comprendre certaines lignes non expliquées dans cette partie.

### IV-A - myWindow.h

```
#ifndef MYWINDOW_H
#define MYWINDOW_H

#include "myGLWidget.h"

class myWindow : public myGLWidget
{
    Q_OBJECT
public:
    explicit myWindow(QWidget *parent = 0);
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
};

#endif // MYWINDOW_H
```

### IV-B - myWindow.cpp

```
#include "myWindow.h"

myWindow::myWindow(QWidget *parent)
    : myGLWidget(60, parent, "Premier Polygone avec OpenGL et Qt")
{
}

void myWindow::initializeGL()
{
    glShadeModel(GL_SMOOTH);
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}

void myWindow::resizeGL(int width, int height)
{
    if(height == 0)
        height = 1;
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, 0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
void myWindow::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-1.5f, 0.0f, -6.0f);

    glBegin(GL_TRIANGLES);
        glVertex3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glVertex3f(1.0f, -1.0f, 0.0f);
    glEnd();

    glTranslatef(3.0f, 0.0f, -6.0f);

    glBegin(GL_QUADS);
        glVertex3f(-1.0f, 1.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glVertex3f(1.0f, -1.0f, 0.0f);
        glVertex3d(1.0f, 1.0f, 0.0f);
    glEnd();
}
```

Le code n'est pas vraiment compliqué ; certains points sont cependant assez importants et seront donc détaillés.

```
myWindow::myWindow(QWidget *parent)
    : myGLWidget(60, parent, "Premier Polygone avec OpenGL et Qt")
{
}
```

Nous avons choisi de Â« brider Â» le rendu OpenGL à 60 ips pour éviter de surcharger le processeur et la carte graphique pour rien. Ainsi on attribue la valeur 60. Cette valeur est le nombre d'images par seconde. Le reste des arguments se passe d'explication.

```
void myWindow::initializeGL()
{
    glShadeModel(GL_SMOOTH);
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}
```

Comme vous pouvez le constater, la fonction **initializeGL()** est tout à fait similaire aux fonctions d'initialisation OpenGL avec les autres bibliothèques telles que GLUT ou SDL. Rien de particulier donc à signaler.

```
void myWindow::resizeGL(int width, int height)
{
    if(height == 0)
        height = 1;
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, 0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

Comme dans la plupart des programmes avec OpenGL, il faut gérer le redimensionnement de telle sorte que ça ne déforme pas la scène lors d'un changement de résolution de l'écran ou d'un redimensionnement de la fenêtre. Cette fonction a ce rôle et vous avez déjà d'utiliser une façon similaire pour parer à toute déformation non souhaitée de la scène OpenGL.

```
void myWindow::paintGL()
{
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-1.5f, 0.0f, -6.0f);

    glBegin(GL_TRIANGLES);
        glVertex3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glVertex3f(1.0f, -1.0f, 0.0f);
    glEnd();

    glTranslatef(3.0f, 0.0f, -6.0f);

    glBegin(GL_QUADS);
        glVertex3f(-1.0f, 1.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glVertex3f(1.0f, -1.0f, 0.0f);
        glVertex3d(1.0f, 1.0f, 0.0f);
    glEnd();
}
```

La fonction **paintGL()** permet d'afficher la scène OpenGL. Ici notre scène est constituée d'un triangle et d'un carré.

## IV-C - Modification du main.cpp

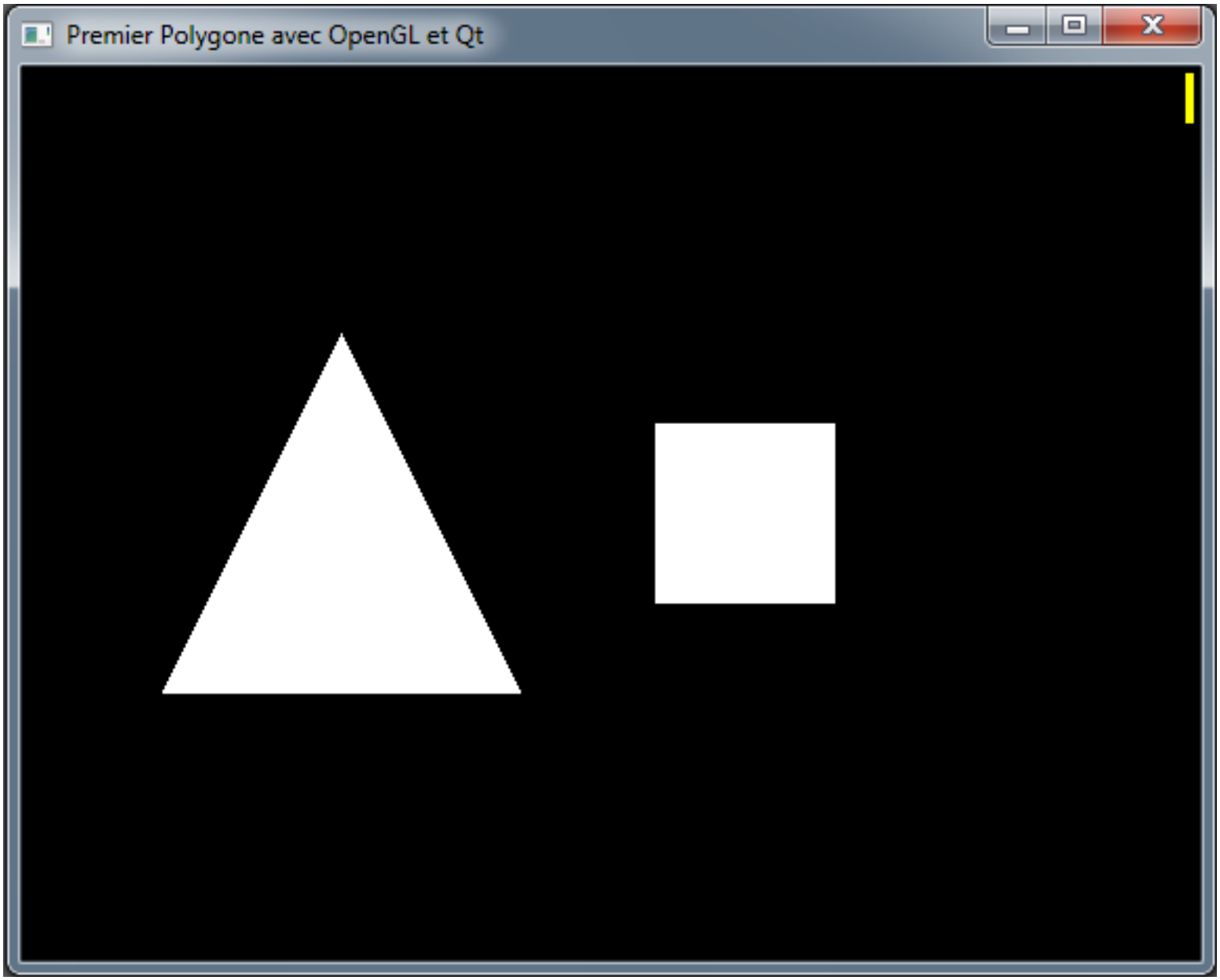
Pour terminer, afin que le programme soit fonctionnel, il nous suffit de retourner dans notre main.cpp et d'y ajouter quelques lignes de code.

```
#include <QApplication>
 #include "myWindow.h"

 int main(int argc, char *argv[])
 {
     QApplication app(argc, argv);
     myWindow myWin; //Ajout de notre classe myWindow
     myWin.show();   //Exécution de notre fenêtre de rendu OpenGL
     return app.exec();
 }
```

## IV-D - La fonction updateGL

Maintenant vous pouvez exécuter notre programme et constater que la scène OpenGL affiche un carré et un triangle. Qt lui permet l'affichage de la fenêtre, la gestion des événements et la régulation du nombre d'images par seconde rendu par OpenGL grâce à son **QTimer**. Le résultat attendu devrait être le suivant :




Le petit « 1 » jaune est l'affichage du nombre d'images par seconde. J'utilise le logiciel Fraps pour afficher le nombre d'image par seconde que je vous conseille vivement d'installer pour afficher ce nombre. Cela vous permettra de surveiller l'optimisation de vos applications. Vous pouvez le télécharger gratuitement : **Fraps**.

Revenons à notre application, quelque chose devrait vous paraitre bizarre. Pourquoi une image par seconde ? Je vous rassure ce n'est pas Fraps qui à fait une erreur parce qu'il n'y a pas d'erreur. Pourtant nous avons bien demandé d'afficher 60 images par seconde ? Eh bien oui, c'est vrai, mais il manque un petit quelque chose dans notre classe myGLWidget.

Un petit rappel sur les trois fonctions importantes de Qt pour gérer OpenGL s'impose :

- **initializeGL()** ne fait qu'initialiser OpenGL, donc rien d'important à signaler ;
- **resizeGL()** permet de redessiner la scène OpenGL lorsqu'on modifie la taille de la fenêtre ou la résolution ;
- **paintGL()** permet de dessiner la scène OpenGL.

Effectuons un petit test avec Fraps, changez la taille de la fenêtre. Mais, le nombre d'image varie ? Effectivement, et c'est parce que **resizeGL()** appel **paintGL()** qui redessine la scène OpenGL lorsqu'on modifie la taille de la fenêtre.

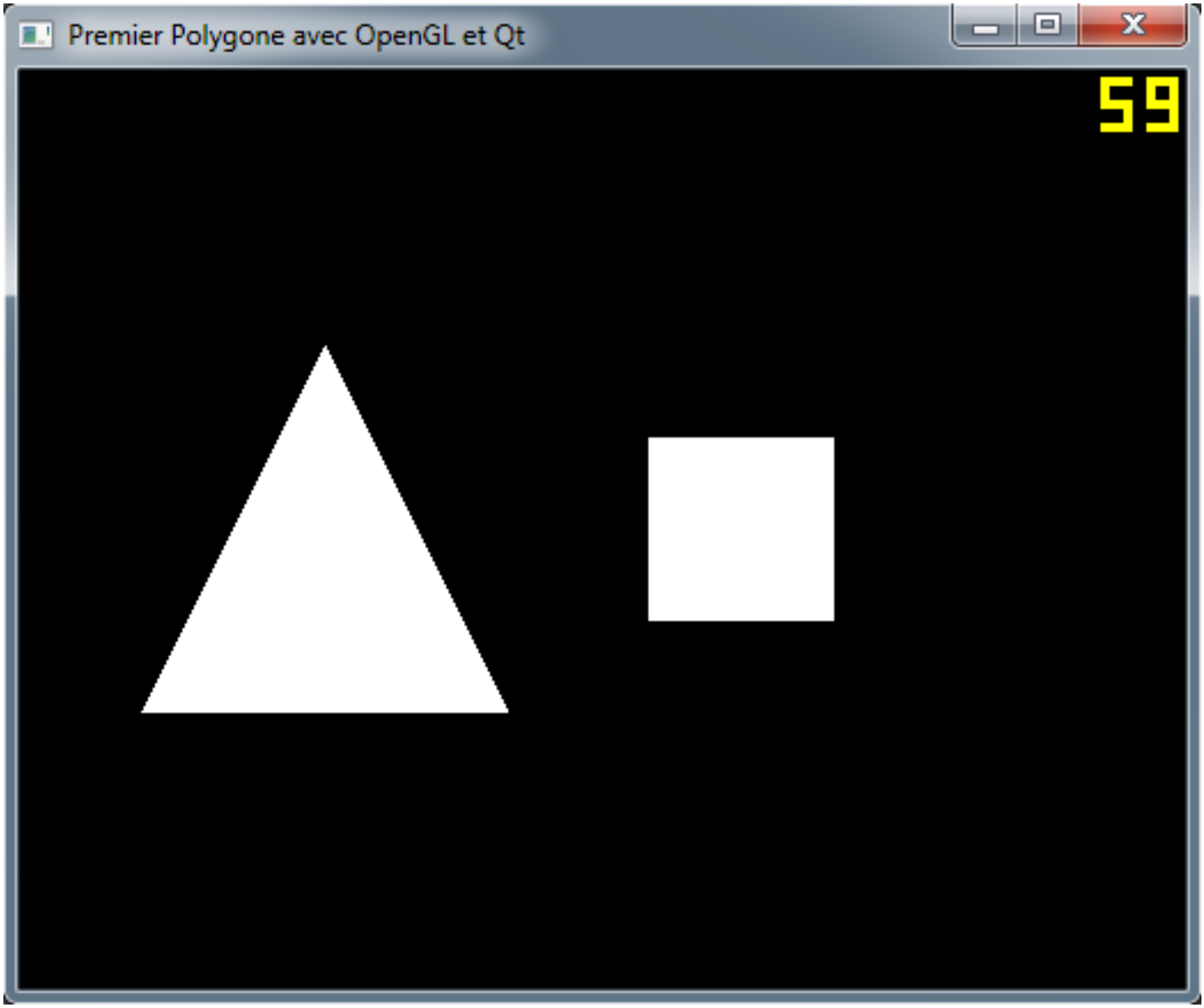
C'est là que ça devient intéressant, **paintGL()** doit être appelé à chaque fois qu'on veut redessiner la scène OpenGL. Seulement, actuellement cette fonction n'est appelée qu'une fois lors de l'exécution du programme et à chaque fois que nous modifions la taille de la fenêtre, ce qui fait varier le nombre d'images par seconde. Pour que **paintGL()** soit appelé même lorsqu'aucune modification de la fenêtre n'est faite, il va falloir modifier la classe myGLWidget.

Ajoutons la ligne suivante dans la fonction timeOutSlot() de la classe myGLWidget :

```
updateGL();
```

**updateGL()** fait un appel à **paintGL()**. Dans notre programme, la fonction timeOutSlot() est appelée 60 fois par seconde comme nous l'avons demandé, ainsi **updateGL()** va appeler **paintGL()** 60 fois par seconde ce qui va nous afficher environ 60 images par seconde.

Vous devez maintenant avoir ceci :




## V - Agrémenter votre widget avec de la couleur et un mode plein écran

Maintenant que nous avons créé notre application OpenGL avec Qt, nous allons lui ajouter une fonctionnalité qui peut-être intéressante dans le cadre d'un jeu, par exemple : la fonction de mode plein écran. L'utilisateur aura le

choix entre un mode plein écran ou un mode fenêtré. Pour réaliser cette fonctionnalité, il nous suffira de rajouter un événement dans notre fonction **keyPressEvent()**. Pour égailler notre application nous allons aussi ajouter de la couleur dans notre programme. Les formes blanches c'est bien gentil mais un peu de couleur ne serait pas de refus.

## V-A - De la couleur

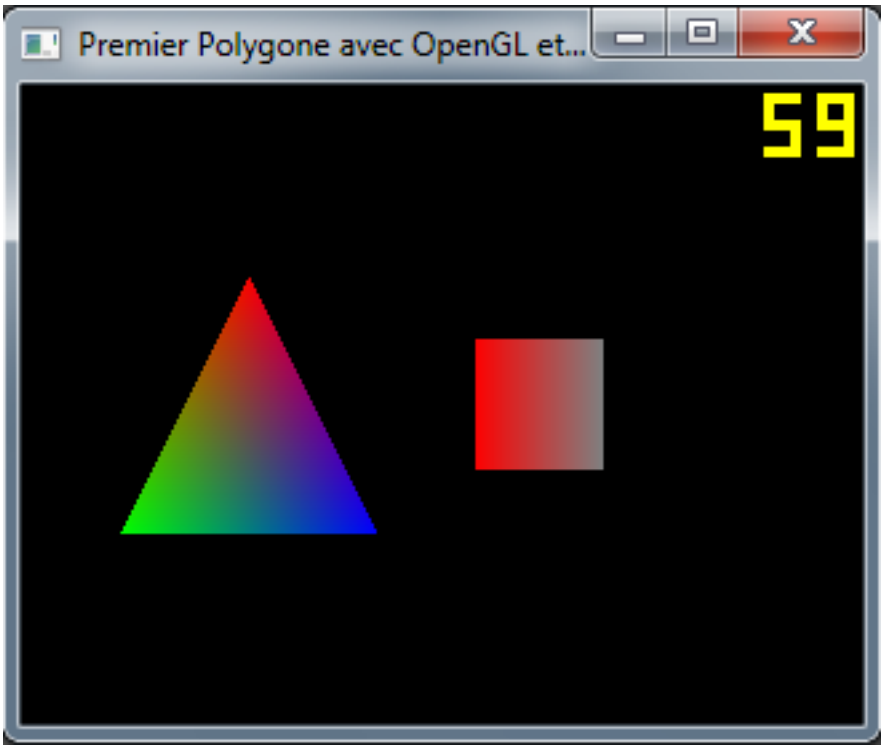
Pour ajouter de la couleur il nous suffit de rajouter les fonctions d'OpenGL **glColor3f()** dans notre fonction **paintGL()**:

```
glColor3f(GLfloat red, GLfloat  green, GLfloat blue);
```

Nous voyons donc que le code ci-dessus est du code OpenGL. Je le rappelle, Qt utilise les en-têtes d'OpenGL donc ce que vous avez appris avec OpenGL reste strictement identique. Certaines personnes ayant codé avec OpenGL et une autre bibliothèque de fenêtrage telle que SDL ou GLUT, pensent qu'en passant sur une autre bibliothèque de fenêtrage, le code d'OpenGL change. C'est strictement faux, OpenGL reste OpenGL, cette petite partie du tutoriel est présente juste pour bien mettre tout ça au clair. Pour revenir à notre projet, nous pouvons donc ajouter le code suivant dans la fonction **paintGL()** :

```
void myWindow::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-1.5f, 0.0f, -6.0f);
    glBegin(GL_TRIANGLES);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(0.0f, 1.0f, 0.0f);
        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glColor3f(0.0f, 0.0f, 1.0f);
        glVertex3f(1.0f, -1.0f, 0.0f);
    glEnd();
    glTranslatef(3.0f, 0.0f, -6.0f);
    glBegin(GL_QUADS);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-1.0f, 1.0f, 0.0f);
        glVertex3f(-1.0f, -1.0f, 0.0f);
        glColor3f(0.5f, 0.5f, 0.5f);
        glVertex3f(1.0f, -1.0f, 0.0f);
        glVertex3d(1.0f, 1.0f, 0.0f);
    glEnd();
}
```

J'ai volontairement recopié tout le code de la fonction pour qu'il n'y ait pas de confusion. Nous avons donc ajouté les fonctions **glColor3f()** dans **paintGL()** ce qui ajoutera donc de jolis dégradés pour nos formes géométriques présentes dans notre scène OpenGL. Le résultat devrait être le suivant :




## V-B - Plein écran ou fenêtré ?

Qt offre la possibilité de créer facilement des fenêtres et c'est pour cette raison que cette bibliothèque est vraiment intéressante. Mais si nous souhaitons créer un jeu, nous n'allons pas obliger nos joueurs à rester en mode fenêtré n'est-ce pas ? Il serait agréable de pouvoir choisir entre le mode fenêtré et le mode plein écran. Et c'est ce que nous allons apprendre dans cette partie et par ailleurs vous allez pouvoir admirer la facilité d'utilisation de Qt.

Deux choix s'offrent à nous, soit on redéfini la fonction **keyPressEvent()** dans la classe myWindow, soit on ajoute la fonctionnalité dans la classe myGLWidget. J'ai choisi d'ajouter la fonctionnalité plein écran/fenêtré dans la classe myGLWidget car c'est une fonctionnalité qui se révèlera utile dans tous nos prochains programmes. Avant de se plonger dans le code de la fonction **keyPressEvent()** et de le modifier, nous allons utiliser une variable qui nous permettra de connaitre l'état de la fenêtre (plein écran ou fenêtré). Ajoutons dans la classe myGLWidget un attribut privé :

```
bool b_Fullscreen;
```

Par défaut, notre booléen vaudra false car la fenêtre est par défaut en mode fenêtré. Il va falloir donc initialiser notre attribut dans le constructeur de la classe myGLWidget :

```
b_Fullscreen = false ;
```

Nous allons ajouter une fonction toggleFullWindow() dans notre classe myGLWidget, partie public.

```
void toggleFullWindow();
```

Cette fonction aura pour rôle de passer du mode fenêtré au mode plein écran. Pour cela nous allons utiliser notre variable b_Fullscreen qui enregistrera l'état de la fenêtre.

```
void myGLWidget::toggleFullWindow()
{
    if(b_Fullscreen)
    {
        showNormal();
```

```
        b_Fullscreen = false;
    }
    else
    {
        showFullScreen();
        b_Fullscreen = true;
    }
}
```

Détaillons le code afin que ça reste bien clair pour tout le monde.

```
if(b_Fullscreen)
```

Nous testons si la fenêtre est en mode plein écran.

```
{
    showNormal();
    b_Fullscreen = false;
}
```

Si la fenêtre est déjà en mode plein écran nous repassons notre fenêtre en mode normal grâce à la fonction **showNormal()**, puis affectons la valeur false à notre variable b_Fullscreen pour enregistrer l'état fenêtré de notre programme.

```
else
{
    showFullScreen();
    b_Fullscreen = true;
}
```

Au contraire, si notre fenêtre n'est pas en mode plein écran nous exécutons ce mode grâce à la fonction **showFullScreen()** et affectons la valeur true à notre booléen b_Fullscreen pour enregistrer l'état plein écran de notre programme.

Observons la fonction **keyPressEvent()** de notre classe myGLWidget, voici le code actuel :

```
void myGLWidget::keyPressEvent(QKeyEvent *keyEvent)
{
    switch(keyEvent->key())
    {
        case Qt::Key_Escape:
            close();
            break;
    }
}
```

Seule la touche « Echap » (Escape en anglais) effectue une action, fermer la fenêtre. Nous allons donc ajouter une touche qui sera « F1 » pour effectuer une action qui aura pour rôle de changer de mode d'affichage de la fenêtre. Voici le code complet:

```
void myGLWidget::keyPressEvent(QKeyEvent *keyEvent)
{
    switch(keyEvent->key())
    {
        case Qt::Key_Escape:
            close();
            break;
        case Qt::Key_F1:
            toggleFullWindow();
            break;
    }
}
```

L'appuie sur la touche « F1 » va entrainer l'entrée dans la case puis le programme va appeler la fonction toggleFullWindow() qui modifiera le mode d'affichage.

Nous pouvons maintenant tester notre programme et changer son mode d'affichage en appuyant sur la touche « F1 ».

## V-C - Conclusion

Vous avez pu dans ce chapitre vous rendre compte que le code OpenGL reste identique, puisque Qt ne fait que wrapper la version d'OpenGL que vous avez installée avec votre compilateur ou vous même, si vous avez décidé de mettre à jour vos en-têtes d'OpenGL. Ainsi tout ce que vous avez appris sur OpenGL s'applique à l'identique avec Qt. Nous avons pu aussi apprendre à changer le mode d'affichage de notre fenêtre en permettant de choisir entre le mode fenêtré et le mode plein écran. Une seule fonction suffit pour passer d'un mode à un autre.

Le prochain chapitre traitera d'une partie intéressante, les textures.

# VI - Appliquer des textures pour rendre votre rendu réaliste

Dans ce chapitre, nous allons apprendre à charger une image et à l'appliquer à une forme en tant que texture. Nous allons pouvoir observer comment Qt nous simplifie la tâche en matière de chargement d'images. La classe **QImage** nous fera quasiment tout le travail. Pour ce qui est de la texture j'utiliserai celle-ci :

## VI-A - myWindow.h

Tout d'abord, étant donné que nous allons utiliser la classe **QImage**, il va falloir inclure l'en-tête :

```
#include <QImage>
```

OpenGL stock les textures dans un tableau, c'est pourquoi nous allons devoir en créer un en attribut privée de notre classe :

```
GLuint texture[1];
```

Étant donné que nous souhaitons ici ne charger qu'une texture, notre tableau aura pour taille 1. Bien sûr, si vous souhaitez charger deux textures différentes, alors il vous faudra un tableau ayant pour taille 2.

Nous allons maintenant ajouter une fonction qui aura pour rôle de charger la texture comme il se doit. Elle prendra en paramètre un **QString** qui contiendra le chemin d'accès de notre texture. Ainsi vous pourrez réutiliser cette fonction pour charger n'importe quelle texture.

```
void loadTexture(QString textureName);
```

Pour être sûr d'être sur la même longueur d'onde et d'avoir le même code, je vous mets le code complet de notre en-tête :

```
#ifndef MYWINDOW_H
#define MYWINDOW_H

#include <QImage>
#include "myGLWidget.h"

class myWindow : public myGLWidget
{
    Q_OBJECT
public:
    explicit myWindow(QWidget *parent = 0);
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void loadTexture(QString textureName);
private :
    GLuint texture[1];
    float f_x;
};

#endif // MYWINDOW_H
```

## VI-B - myWindow.cpp

Concentrons-nous maintenant sur notre fonction loadTexture(). Cette fonction comme son nom l'indique aura pour rôle de charger les textures. Qt par défaut permet de charger les images de format .png et .bmp sans DLL requis. Toutefois, pour les images de format .gif, .ico, .jpeg, .mng, .svg et .tiff il faudra inclure le dossier « imageformats » situé dans votre dossier d'installation de « Qt/qt/plugins/imageformats » à côté de votre exécutable. Pour nous simplifier la vie, j'ai choisi d'utiliser dans ce tutoriel une image au format .png, donc pas de DLL à fournir.

Passons au vif du sujet, notre fonction loadTexture() :

```
void myWindow::loadTexture(QString textureName)
{
    QImage qim_Texture;
    QImage qim_TempTexture;
    qim_TempTexture.load(textureName);
    qim_Texture = QGLWidget::convertToGLFormat( qim_TempTexture );
    glGenTextures( 1, &texture[0] );
    glBindTexture( GL_TEXTURE_2D, texture[0] );
    glTexImage2D( GL_TEXTURE_2D, 0, 3, qim_Texture.width(), qim_Texture.height(), 0, GL_RGBA,
 GL_UNSIGNED_BYTE, qim_Texture.bits() );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
}
```

Nous allons détailler tout le code :

```
QImage qim_Texture;
QImage qim_TempTexture;
```

Notre variable qim_Texture aura pour but de contenir la texture au format compréhensible pour OpenGL tandis que la variable qim_TempTexture aura pour rôle de contenir la texture dans son format brut. Ce sera plus clair pour nous dans quelques lignes.

```
qim_TempTexture.load(textureName);
```

Ici on charge la texture. N'oublions pas que la variable textureName contient l'adresse de la texture sur votre disque dur.

```
qim_Texture = QGLWidget::convertToGLFormat( qim_TempTexture );
```

Maintenant nous convertissons notre texture en un format compréhensible par OpenGL et l'enregistrons dans notre variable qim_Texture.

```
glGenTextures( 1, &texture[0] );
glBindTexture( GL_TEXTURE_2D, texture[0] );
```

Le code suivant ne nécessite pas d'explication puisque vous êtes déjà censé l'avoir utilisé. Il sert juste à préparer notre tableau texture[] à recevoir une texture.

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, qim_Texture.width(), qim_Texture.height(), 0, GL_RGBA,
 GL_UNSIGNED_BYTE, qim_Texture.bits());
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
}
```

Cette ligne défini les propriétés de l'image, avec sa taille qu'on obtient grâce aux fonctions qim_Texture.width() et qim_Texture.height() ainsi que ses données avec qim_Texture.bits(). Bref, rien de bien spécial à dire dessus, puisque c'est du code purement OpenGL.

Grâce à cette fonction nous pouvons charger n'importe quelle texture. On aura plus qu'à faire appel à elle quand on voudra charger une texture.

Toutefois, notre travail n'est pas encore tout à fait fini. Nous devons faire appel à la fonction loadTexture() et il va falloir indiquer à OpenGL que nous utiliserons des textures. Tout ceci va se réaliser dans la fonction **initializeGL()** en ajoutant ces deux lignes de code :

```
loadTexture("texture/box.png");
glEnable(GL_TEXTURE_2D);
```

Et pour terminer, nous appliquerons la texture sur un cube grâce au code suivant, qui se passe de tout commentaire, vous l'avez très certainement déjà utilisé :

```
glBindTexture(GL_TEXTURE_2D, texture[0]);

glBegin(GL_QUADS);
    // Face Avant
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
    // Face Arrière
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    // Face Haut
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
    // Face Bas
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
```

```
    // Face Droite
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
    // Face Gauche
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
glEnd();
}
```

Pour être sur là aussi qu'il n'y ait pas de confusion, voici le code complet de notre fichier :

```
#include "myWindow.h"

myWindow::myWindow(QWidget *parent)
   : myGLWidget(60, parent, "Premier Polygone avec OpenGL et Qt")
{
}

void myWindow::initializeGL()
{
    f_x = 0.0;
    loadTexture("texture/box.png");

    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_SMOOTH);

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0f);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}

void myWindow::resizeGL(int width, int height)
{
    if(height == 0)
        height = 1;
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, 0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void myWindow::paintGL()
{
    f_x += 0.1;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-1.5f, 0.0f, -6.0f);
    glRotatef(f_x, 1.0, 0.3, 0.1);

    glBindTexture(GL_TEXTURE_2D, texture[0]);

   glBegin(GL_QUADS);
       // Face Avant
       glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
       glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
       glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
       glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
       // Face Arrière
       glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
       glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
       glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
```

```
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        // Face Haut
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
        // Face Bas
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
        // Face Droite
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
        // Face Gauche
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
    glEnd();
}

void myWindow::loadTexture(QString textureName)
{
    QImage qim_Texture;
    QImage qim_TempTexture;
    qim_TempTexture.load(textureName);
    qim_Texture = QGLWidget::convertToGLFormat( qim_TempTexture );
    glGenTextures( 1, &texture[0] );
    glBindTexture( GL_TEXTURE_2D, texture[0] );
    glTexImage2D( GL_TEXTURE_2D, 0, 3, qim_Texture.width(), qim_Texture.height(), 0, GL_RGBA,
 GL_UNSIGNED_BYTE, qim_Texture.bits() );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
}
```

Le résultat attendu est donc le suivant :


## VII - Conclusion

Dans ce chapitre nous avons appris à charger des textures et nous avons pu remarquer que Qt nous simplifie énormément la tâche comparé à la bibliothèque GLUT, par exemple, qui ne contient aucune fonction permettant de charger des images. Qt supporte un grand nombre d'images par défaut ce qui est très pratique.

## VII-A - Conclusion et liens utiles

Durant tout ce tutoriel, nous avons appris à utiliser OpenGL avec Qt. L'utilisation de ce mariage entre ces deux bibliothèques nous offre de grandes possibilités ! Réalisation d'un éditeur de niveau pour votre jeu, modélisateur 3D, simulateur 3D, etc. Ce tutoriel ne couvre pas toutes les possibilités, bien sûr, de Qt avec OpenGL mais vous ouvre la porte vers l'apprentissage de techniques plus poussées. Ce tutoriel nous a permit de :

- comprendre l'utilisation de la classe QGLWidget ;
- créer une fenêtre contenant un rendu OpenGL ;
- utiliser le gestionnaire d'événements pour fermer la fenêtre ou changer de mode d'affichage ;
- réguler le nombre d'images par seconde de notre application grâce au QTimer ;
- charger des images pour pouvoir les appliquer en tant que textures sur des formes de la scène OpenGL ;

## VII-B - Liens utiles

Voici quelques liens utiles pour approfondir vos connaissances sur OpenGL ou Qt :

- **Tutoriel de NeHe** (la référence en matière d'apprentissage d'OpenGL en anglais)
- **Tutoriel OpenGL sur Developpez.com**
- **Tutoriel OpenGL/Qt**

- **Tutoriel Qt sur Developpez.com**

Et les liens indispensables :

- **Site officiel d'OpenGL**
- **Site officiel de Qt**

## VII-C - Remerciements

Je tiens aussi à remercier **dourouc05**, **johnlamericain** et **abdelite** pour leur aide et leur soutien lors de la rédaction de ce tutoriel. Un grand merci aussi à **jacques_jean** pour les diverses corrections d'orthographe et de syntaxe.