L'utilisation des QGraphicsScene avec Qt

par Julien Chichignoud

Date de publication : 08 mai 2011

Dernière mise à jour :

I - Introduction

Dans ce tutoriel nous allons voir comment utiliser la classe QGraphicsItem, qui permet de façon générale d'intégrer des objets dans une scène 2D mais aussi de leur donner un aspect 3D, moyennant quelques astuces. Attention toutefois, ces outils de Qt ne sont pas non plus conçus pour afficher des scènes en 3D, il a simplement une bonne souplesse qui permet de réaliser de belles interfaces assez évoluées.

J'aborderai donc l'intégration d'objets dans une scène en deux dimensions grâce à Qt, avant de vous montrer comment simuler la profondeur afin de donner une impression de 3D à ces objets.

II - Insérer des widgets dans une scène

II-A - Quelques classes utiles

Afin de visualiser des widgets (et plus généralement des objets graphiques) dans une scène, il faut commencer par en créer une qui va contenir tout ce beau monde. La classe fournie par Qt s'appelle sobrement QGraphicsScene et permet d'insérer des QGraphicsItem. La classe abstraite QGraphicsItem représente tout ce qui est affichable dans une QGraphicsScene. On trouve notamment un nombre important de classes permettant de gérer des polygones, des textes, des QPixmap, des widgets

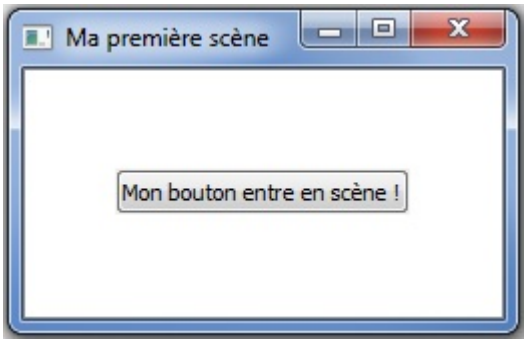
La classe QGraphicsProxyWidget, héritant de QGraphicsItem, est particulièrement adaptée à l'insertion des widgets dans une QGraphicsScene. Il suffit de créer votre QGraphicsProxyWidget, de lui attacher le widget que vous désirez intégrer à la scène puis d'ajouter le proxy en tant qu'objet. Vous manipulez donc votre widget à travers le proxy, tout ceci étant transparent une fois que le proxy et le widget sont attachés. Il est de plus tout à fait possible d'intégrer des widgets complexes, comme une QTabWidget, qui contient lui-même d'autres widgets.

Dans les exemples, j'appliquerai les notions de ce tutoriel à des widgets car c'est leur intégration qui m'a poussé à utiliser ces classes. Cependant, les méthodes utilisées sont pour la plupart héritées de QGraphicsItem, vous pourrez donc les adapter à d'autres objets.

Enfin, comme la scène n'est pas un widget mais un simple conteneur, il faut aussi créer une vue avec la classe QGraphicsView, qui permet d'afficher le contenu de notre scène. Sans plus attendre, commençons par créer notre première scène, avec un bouton au milieu.

main.cpp

```
#include <QApplication>
#include <QtGui>

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

    QPushButton *bouton = new QPushButton("Mon bouton entre en scène !");
    QGraphicsScene scene;
    QGraphicsProxyWidget *proxy = new QGraphicsProxyWidget();
    proxy->setWidget(bouton);
    scene.addItem(proxy);

    QGraphicsView view(&scene);
    view.show();

    return app.exec();
}
```



Ce code devrait vous afficher un bouton au milieu d'un écran blanc. Vous venez d'intégrer un premier widget dans une QGraphicsScene.

II-B - Déplacer les objets dans la scène

Une fois le widget intégré à la scène à travers notre proxy, Qt nous offre un grand nombre de méthodes pour modifier son positionnement.

II-B-1 - Les coordonnées

Si les systèmes de coordonnées 2D ne vous sont pas familiers, sachez que l'axe X est l'axe horizontal orienté vers la droite et que l'axe Y est l'axe vertical orienté vers le bas. *A priori*, si vous avez déjà un peu manipulé Qt ou la SDL par exemple, vous devriez vous y retrouver.

Il faut de plus savoir que la vue, la scène et les items ont chacun leur propre système de coordonnées. Ainsi, le point (0, 0) qui est le point en haut à gauche de la vue dans son propre système aura des coordonnées différentes dans le système de coordonnées de la scène. Ceci se modélise par le schéma suivant issu de la documentation, dans lequel le repère de la vue est en vert, la scène est en bleu et l'item en rouge.

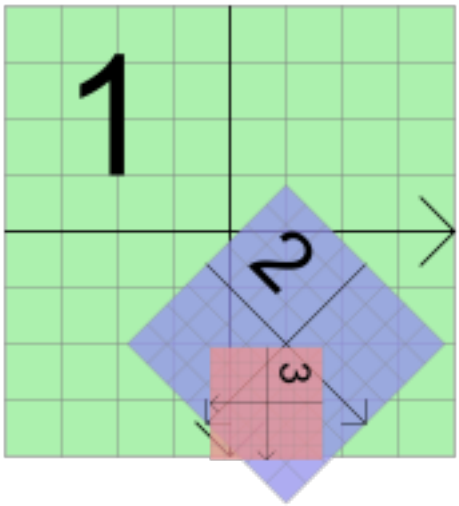


Image issue de la documentation officielle Qt (http://doc.qt.nokia.com/4.7/graphicsview.html)

Afin de simplifier les conversions d'un système de coordonnées à un autre, Qt fournit des méthodes mapToScene() et mapFromScene() dans les classes QGraphicsView et QGraphicsItem qui permettent de retrouver les coordonnées d'un point dans la scène à partir des coordonnées dans la vue ou l'item et inversement.

II-B-2 - Translation

La fonction setPos() de QGraphicsItem prend deux paramètres, dx et dy, qui sont les déplacements relatifs respectivement sur l'axe X et l'axe Y de la scène, paramètres qui deviendront alors les nouvelles coordonnées de notre objet. À savoir qu'une méthode surchargée existe, prenant en paramètre un QPointF. À vous de voir laquelle vous préférez. Il est aussi possible d'appeler la fonction move(), qui ajoute le déplacement sur les deux axes à la position actuelle de l'objet. Essayons tout de suite de déplacer notre objet dans la scène :

main.cpp

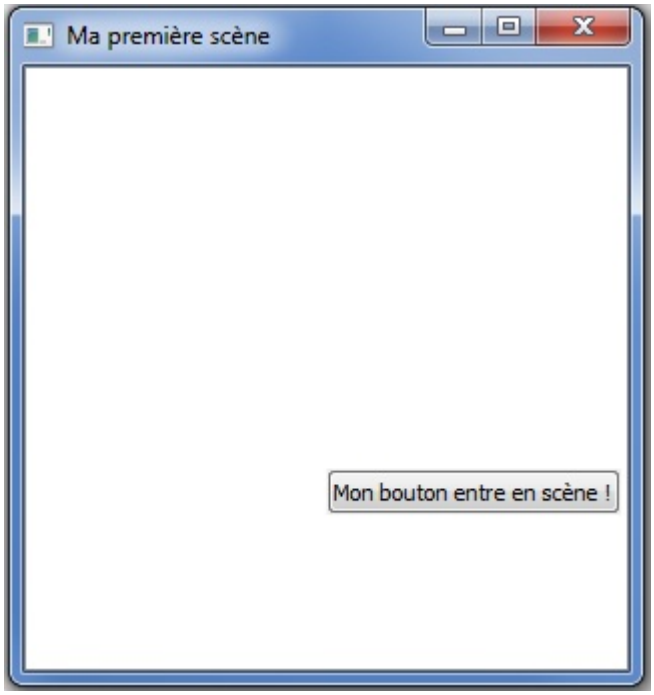
```
#include <QApplication>
#include <QtGui>

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

 QPushButton *bouton = new QPushButton("Mon bouton entre en scène !");
 QGraphicsScene scene;
 QGraphicsProxyWidget *proxy = new QGraphicsProxyWidget();
 proxy->setWidget(bouton);
 scene.addItem(proxy);

 proxy->setPos(150, 200);

 QGraphicsView view(&scene);
 view.show();

 return app.exec();
}
```

Si vous exécutez ce code, vous vous rendrez compte que votre widget ne s'est pas déplacé. En effet, c'est une des particularités de la QGraphicsView qui s'arrange par défaut pour placer vos objets au milieu de la vue.

Heureusement, il est possible de définir la zone de la scène que l'on désire voir à l'écran. Ajoutez cette ligne après la création de la scène.

```
scene.setSceneRect(-150, -150, 300, 300);
```

Les deux premiers nombres sont les coordonnées du point qui doit être placé dans le coin supérieur gauche de la vue (ici, (-150, -150)). Les deux nombres suivants sont la largeur et la hauteur de la scène que je désire afficher. Ainsi, comme le point (-150, -150) est le coin supérieur gauche de ma fenêtre, je vais retrouver l'origine de la scène au centre de l'écran. Si vous relancez votre code avec et sans le déplacement du bouton, vous devriez voir cette fois-ci une différence de position.

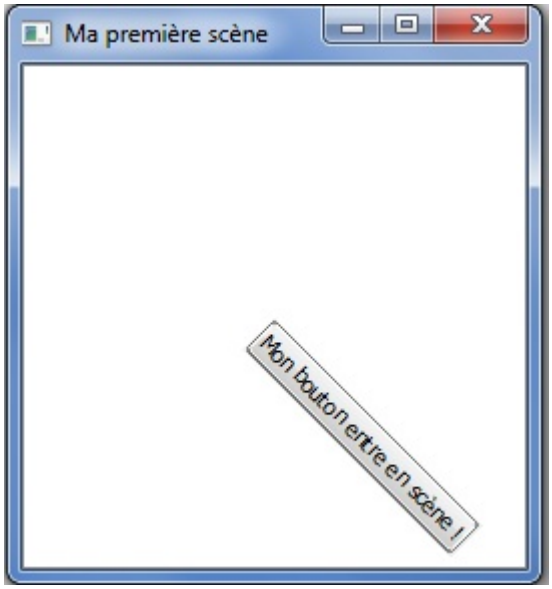


II-B-3 -

Afin d'appliquer une rotation à l'objet, il suffit d'utiliser la méthode setRotation(), pour laquelle vous devrez fournir l'angle de rotation en degrés. Un angle positif tourne l'objet dans le sens des aiguilles d'une montre, un angle négatif le tourne dans le sens inverse.

Par exemple :

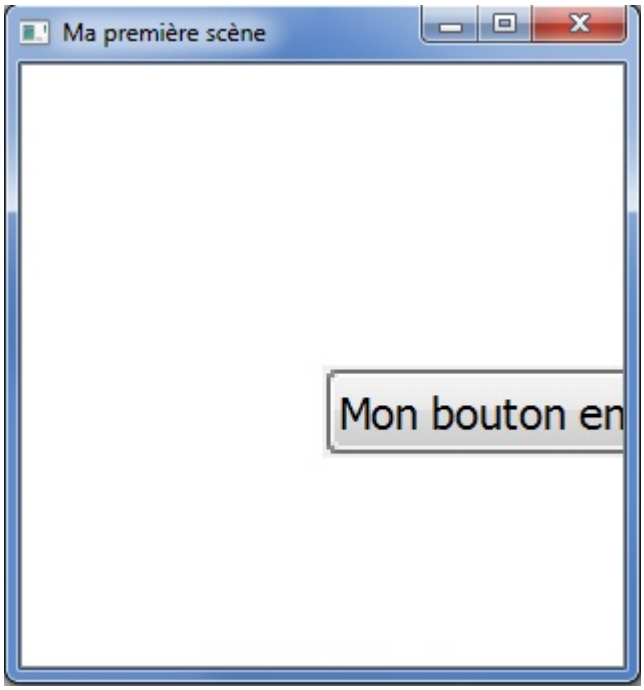
```
proxy->setRotation(45);
```



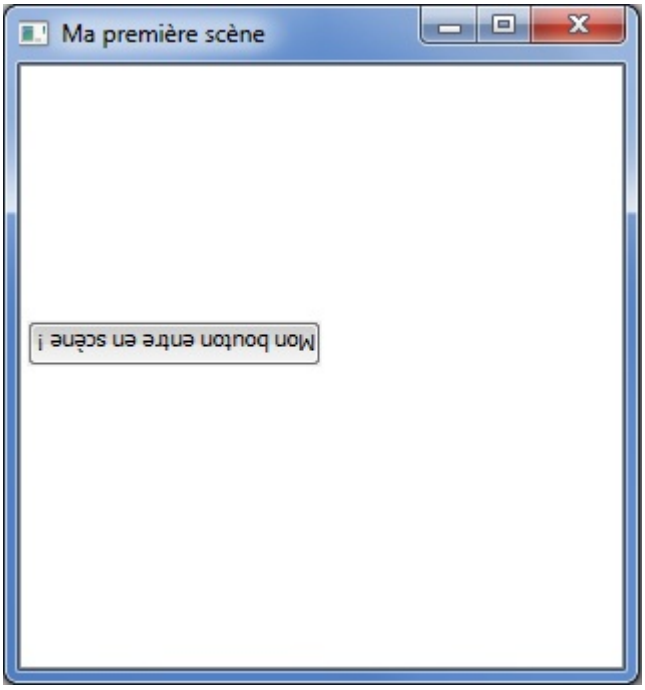
II-B-4 - Mise à l'échelle

Enfin, la méthode setScale() vous permet d'augmenter ou de diminuer la taille prise par l'objet dans la scène. Un nombre supérieur à 1 agrandit l'image, tandis qu'un nombre compris entre 0 et 1 la réduit. Ci-dessous, j'ai pris un facteur de 2 pour agrandir le bouton, dont une partie sort maintenant de l'écran.

```
proxy->setScale(2);
```



Cette fonction prend aussi en compte les nombres négatifs, ce qui permet d'effectuer une symétrie de l'image en plus de modifier sa taille. Avec un facteur -1 on obtient ainsi un bouton renversé.



II-B-5 - Conclusion

Ces opérations de base sont en gros le minimum pour pouvoir gérer vos objets dans la scène 2D.

Je parle ici de scène 2D puisque, si vous avez bien suivi mes explications, je n'ai parlé que de deux axes pour la translation, et uniquement d'une rotation autour de l'axe perpendiculaire à votre écran. En effet, un QGraphicsItem est bien affiché en deux dimensions dans la scène et ne prend pas directement la profondeur en paramètre.

Avant de donner pour de bon de la profondeur à notre scène et de faire des manipulations 3D, il va falloir passer un peu de temps sur les matrices, un outil mathématique qui permet d'effectuer les manipulations de base en 3D, et pour lequel Qt nous donne tous les outils nécessaires pour limiter les calculs à faire.

III - Les transformations avec QTransform

On a vu que les QGraphicsItem proposaient diverses méthodes pour déplacer, tourner ou agrandir les widgets insérés. Seulement, tout cela n'a pas vraiment l'air d'avoir la profondeur qu'on attend d'une scène 3D. Pour cela, on va utiliser une nouvelle classe de Qt, la classe QTransform.

Si vous avez travaillé un peu les mathématiques dans vos études supérieures, ce qui suit devrait vous être familier, dans le cas contraire, lisez attentivement la suite.

III-A - Utiliser un QTransform avec un QGraphicsScene

Un QTransform représente un objet mathématique appelé matrice, que l'on peut représenter dans notre cas comme un tableau comportant trois lignes et trois colonnes et contenant toutes les informations nécessaires pour effectuer des modifications sur les coordonnées d'un objet dans l'espace.

Voici comment on représente généralement une matrice :

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

L'édition d'un QTransform est très simple dans le code puisque Qt s'occupe de toute la partie mathématique en proposant des fonctions vous permettant de générer et modifier ces matrices.
Comme un exemple vaut mieux qu'un long discours, voici tout de suite comment réaliser une translation avec un QTransform.

main.cpp

```
#include <QApplication>
#include <QtGui>

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

    QPushButton *bouton = new QPushButton("Mon bouton entre en scène !");

    QGraphicsScene scene;
    scene.setSceneRect(-150 , -150, 300, 300);

    QGraphicsProxyWidget *proxy = new QGraphicsProxyWidget();
    proxy->setWidget(bouton);
    scene.addItem(proxy);

    QTransform matrix;
    matrix.translate(150, 200)
    proxy->setTransform(matrix);


    QGraphicsView view(&scene);
    view.show();

    return app.exec();
}
```

Comme précédemment, il a suffi de renseigner les valeurs de mon vecteur déplacement et Qt a généré tout seul la matrice correspondante.

Exécutez ce code qui doit vous afficher la même chose que tout à l'heure. Oui, en l'occurrence, le résultat est le même lorsque l'on regarde notre fenêtre. Le widget s'est bien déplacé de 150 pixels vers la droite et de 200 pixels vers le bas.

De même, vous pouvez générer des rotations et des mises à l'échelle à l'aide des méthodes rotate() et scale() de la classe QTransform.

L'avantage des QTransform par rapport à l'utilisation des méthodes précédentes est la possibilité de faire des transformations composées en une seule opération, plutôt que d'appliquer une méthode à la fois. En revanche, il n'est pas souhaitable de combiner l'utilisation des QTransform avec la méthode move() du QGraphicsItem par exemple, puisque les modifications effectuées par un QTransform ne sont récupérables que par la méthode transform() du QGraphicsItem modifié. Cela veut dire que les coordonnées de l'objet, auxquelles on pouvait accéder par QGraphicsItem::x() et QGraphicsItem::y(), n'ont pas été modifiées et valent toujours 0 !

```
QTransform matrix;
matrix.translate(150, 200);
proxy->setTransform(matrix);

matrix = QTransfrom().translate(proxy->x(), proxy->y());
proxy->setTransform(matrix);
```

Si vous remplacez le code précédent par ces lignes, vous vous rendrez compte que votre widget n'a pas bougé de l'origine de la scène. En effet, la dernière transformation étant celle qui prend en compte les paramètres x() et y() de notre proxy, il est revenu à sa position (0, 0) car l'application du QTransform ne les a pas modifiées. Evitez donc de mélanger les outils et choisissez celui-ci qui vous semble convenir à votre besoin.

Le QTransform a donc plus pour but de modifier la façon dont est affiché l'objet plutôt que de modifier l'objet lui-même.

III-B - Composer les matrices

III-B-1 - Un outil pratique

Maintenant qu'on a vu comment utiliser les matrices avec les objets de la scène, voyons pourquoi j'ai introduit cette nouvelle notion : c'est un outil très puissant, souvent utilisé lorsque l'on travaille en trois dimensions.

Une matrice (et donc ici un QTransform) ne se limite pas à effectuer une seule opération à la fois. Elle peut en effet contenir les informations nécessaires pour effectuer à la fois une rotation, une translation et une mise à l'échelle ! Il ne suffira plus que d'appliquer la matrice à l'objet avec la méthode QGraphicsItem::setTransform() et le tour est joué.

Mathématiquement, la composition de matrices s'obtient en les multipliant entre elles, pour donner une nouvelle matrice qui contient les deux transformations. Qt permet de faire ces transformations directement via des méthodes de la classe QTransform.

Par exemple, si je veux créer une matrice de rotation puis que je veux rajouter une translation, je vais écrire ceci :

```
QTransform matrix;
matrix.rotate(30, Qt::YAxis);
matrix.translate(20, 15);
```

Il est aussi possible d'utiliser directement la multiplication pour composer vos matrices. Cependant il faut faire attention à l'ordre des calculs. La matrice la plus à gauche dans la multiplication correspond à la transformation effectuée en dernier. Les QTransform tiennent compte de cela, l'utilisation des méthodes rotate() et translate() place la matrice paramètre au début du calcul et non à la fin. Par exemple, voici deux lignes de code qui effectuent le même calcul.

```
matrix.rotate(30, Qt::YAxis);
```

```
matrix = QTransform::rotate(30, Qt::Axis) * matrix;
```

Par conséquent, soyez encore plus prudent dans le choix de l'ordre de vos lignes de code.
Si vous utilisez les multiplications directement, placez vos matrices dans l'ordre inverse dans lequel vous devez faire vos transformations. Si vous utilisez les méthodes fournies par Qt, vous devrez les écrire dans l'ordre normal, ce qui est plus intuitif.

Je n'ai plus qu'à appliquer la matrice à mon objet situé dans la scène pour que ces deux transformations soient prises en compte.

III-B-2 - Une question d'ordre

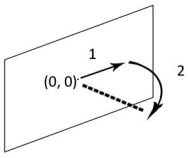
Voyons maintenant pour quelle raison l'ordre des transformations est important et doit être bien choisi. Par exemple, les deux codes suivants donneront deux matrices (et donc deux transformations) différentes.

```
QTransform matrix;
matrix.translate(10, 10);
matrix.rotate(45, Qt::YAxis);
```

```
QTransform matrix;
matrix.rotate(45, Qt::YAxis);
matrix.translate(10, 10);
```

Avec mon premier code, j'ai déplacé mon objet avant de le faire tourner autour de l'origine ; dans le second, je l'ai déplacé après l'avoir fait tourner autour de l'origine.

Dans le premier cas, la transformation finale pourrait se schématiser ainsi :



L'objet a parcouru le chemin en pointillés et ne se trouve ainsi plus dans le plan (x, y) ! L'autre cas en revanche correspond bien à ce qu'on attend intuitivement, c'est-à-dire un objet situé dans le plan (x, y) mais qui a tourné sur lui-même.

Ainsi, pour orienter un objet dans la scène puis le placer, il faut effectuer d'abord la rotation suivant l'angle voulu, puis le déplacement. La seconde solution est donc celle vous devriez utiliser.

IV - Des widgets en 3D

IV-A - Rotation autour de l'axe Y

Nous allons tout d'abord voir comment faire faire à nos widgets des rotations autour de l'axe Y. Un petit tour dans la doc nous apprend que la méthode QTransform::rotate(), qui prend en paramètre un angle et un axe, correspond tout à fait à ce besoin.

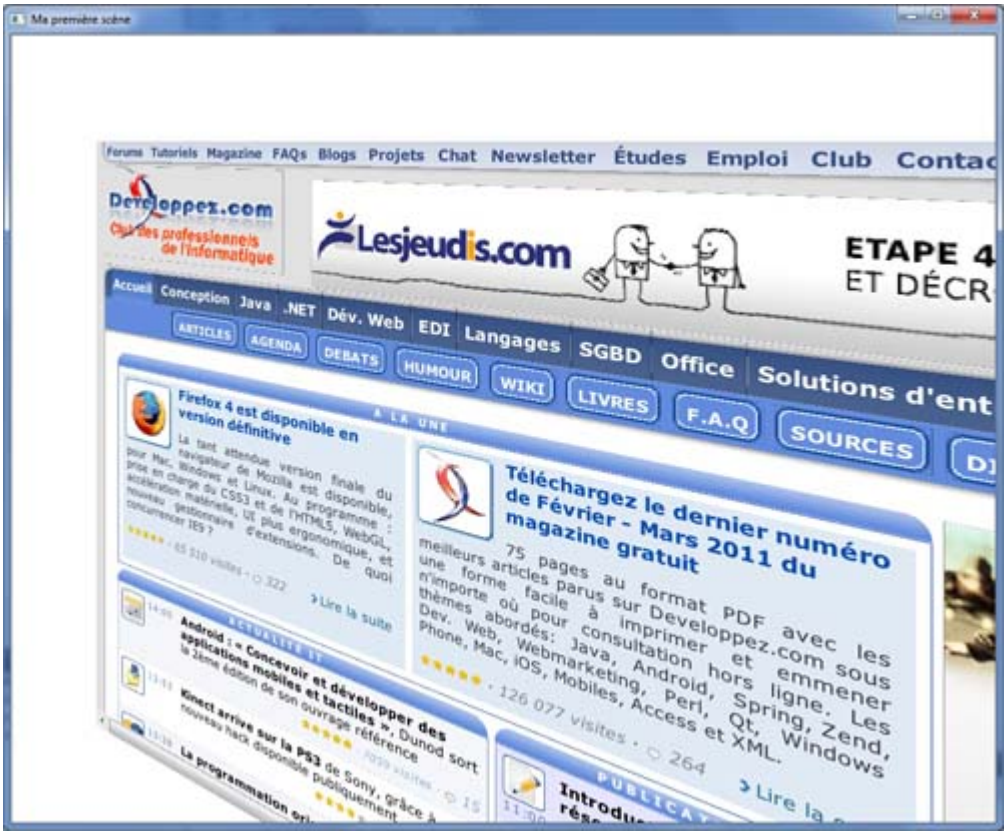
Voyons ce que ça va donner avec un QWebView par exemple (n'oubliez pas d'ajouter la ligne QT += webkit à votre fichier .pro).

main.cpp

main.cpp

```
#include <QApplication>
#include <QtGui>
#include <QWebView>

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

    QWebView *web = new QWebView();
    web->load(QUrl("http://www.developpez.com"));
    web->show();

    QGraphicsScene scene;
    scene.setSceneRect(0 , 0, 1000, 800);

    QGraphicsProxyWidget *proxy = new QGraphicsProxyWidget();
    proxy->setWidget(web);
    scene.addItem(proxy);

    QTransform matrix;
    matrix.rotate(45, Qt::YAxis);
    proxy->setTransform(matrix);

    QGraphicsView view(&scene);
    view.show();

    view.setWindowTitle("Ma première scène");
    return app.exec();
}
```

Compilez ça et normalement vous obtenez une petite perspective sur votre widget, dont les fonctionnalités sont toujours utilisables ! Un QTextEdit continuera à être éditable dans l'espace, une QWebView continuera à charger les pages demandées, etc. Bien sûr, on peut se contenter de formes géométriques plus basiques, mais il est bon de savoir que cela fonctionne avec tous les widgets, il est donc possible d'afficher une page web par exemple.



IV-B - Implémenter le déplacement

Maintenant, nous pouvons ajouter un peu d'interaction à notre scène, en permettant à notre vision de bouger par rapport aux objets. Pour cela je vais créer une classe spéciale, héritée de QGraphicsProxyWidget, qui va contenir les informations dont je souhaite garder la trace : orientation, position, échelle...

Voici le header de la classe que je vous propose.

MyProxy.h

```
#include <QGraphicsProxyWidget>

class MyProxy : public QGraphicsProxyWidget
{
public:
    MyProxy();

    qreal rotationY();
    void setRotationY(qreal rotation);

    QPointF center();

private:
    qreal m_rotationY; // enregistre la rotation autour de l'axe Y

    QPointF m_center; // contient la position du centre du widget
};
```

Puis nous allons créer une classe dérivée de QGraphicsScene pour gérer les événements de la souris. Je vous laisse créer la classe tout seul, il suffit de réimplémenter la méthode mouseMoveEvent(), appelée en cas de déplacement de la souris, et de créer un bouton dans le constructeur que l'on attache à une instance de MyProxy que j'ai placé en attribut de MyScene.

MyScene.cpp

```
MyScene::MyScene() : QGraphicsScene()
{
    QPushButton *bouton = new QPushButton("Mon bouton entre en scène !");
    m_proxy = new MyProxy();
    m_proxy->setWidget(bouton);
    this->addItem(m_proxy);
}
```

Nous allons par exemple demander d'effectuer une rotation autour de l'axe Y lorsque la souris est déplacée latéralement et que le clic gauche est enfoncé.

MyScene.cpp

```
void MyScene::mouseMoveEvent(QGraphicsSceneMouseEvent*e)
{
    if(e->buttons() & Qt::LeftButton)
    {
        QPointF delta(e->scenePos() - e->lastScenePos());
        qreal rotation = delta.x();
        m_proxy->setRotationY(rotation + m_proxy->rotationY());

        QTransform matrix;
        matrix.rotate(m_proxy->rotationY(), Qt::YAxis);
        m_proxy.setTransform(matrix);
    }
}
```

Dans le main il suffit maintenant de créer une instance de MyScene et de lui associer la vue.

main.cpp

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

    MyScene scene;
    scene.setSceneRect(-150, -150, 300, 300);

    QGraphicsView view(&scene);
    view.setWindowTitle("Ma première scène");
    view.show();
    return app.exec();
}
```

Maintenant, lorsque vous déplacez la souris avec le clic gauche enfoncé, votre widget devrait se mettre à tourner verticalement autour de l'origine de la scène. Cependant, cette origine coïncide avec le bord gauche du widget, donc celui-ci ne tourne pas sur de lui-même.

Par conséquent, pour remédier à ce problème, il faut déplacer le widget pour que son centre se situe au point (0, 0) ! Une autre solution consisterai à utiliser la méthode QGraphicsItem::setTransformOriginPoint() et à envoyer le centre de l'item en paramètre.

Reprenons notre scène avec la QWebView. Le déplacement à effectuer est représenté par le schéma ci-dessous : pour que la rotation s'effectue par rapport au centre du widget, il faut ramener ce centre à l'origine de la scène. C'est donc le déplacement représenté par la flèche rouge qui va nous permettre de réaliser cela.

A priori, vous devriez savoir faire cela tout seul. Il suffit de composer notre matrice d'une translation supplémentaire pour déplacer notre widget vers la gauche et en haut, sur une longueur respective de la moitié de la largeur et la moitié de la hauteur (avec un signe négatif, car on va vers la gauche et vers le haut). Vous pouvez récupérer la taille de vos widgets avec les méthodes proxy->widget()->width() et proxy->widget()->height() et les ajouter comme attributs de votre classe MyProxy pour ne plus avoir à les rechercher. Voici donc le code de la rotation avec le recentrage du widget.

```
QTransform matrix;
matrix.rotate(m_proxy->rotationY(), Qt::YAxis);
matrix.translate(-m_proxy->widget()->width()/2, -m_proxy->widget()->height()/2);
m_proxy->setTransform(matrix);
```

Pensez à déplacer l'origine de la scène au centre de la fenêtre avec la méthode QGraphicsScene::setSceneRect() si ce n'est déjà fait, ou bien votre widget sera coupé par le bord de l'écran à cause de ce décalage.

```
scene.setSceneRect(-150, -150, 300, 300);
```

V - Pour aller plus loin : gestion de la caméra

V-A - La caméra : reprenons à l'envers !

Je vous ai déjà dit que l'ordre dans lequel vous composez vos transformations était important. Ça le devient encore plus si vous voulez gérer une caméra dans l'espace !

Prenons un exemple. Considérons la tourelle d'un tank dans la scène. Pour la placer correctement, vous allez devoir lui appliquer sa rotation par rapport au tank, puis la rotation du tank par rapport à la scène, puis enfin la translation qui la positionnera à l'endroit voulu.
Si maintenant vous voulez placer une caméra (votre point de vue) dans la scène, que se passe-t-il ? Dans ce cas, si vous tournez sur vous-même, vous voyez le monde 3D entier qui tourne autour de votre position. La rotation de la caméra doit donc être appliquée après sa translation.

Autant vous prévenir tout de suite, cet exemple ne vous permettra pas de tourner sur vous-même et de vous déplacer dans l'espace, mais plutôt de déplacer tous les objets de la scène d'un coup ou bien de tous les faire tourner devant vous. La construction des matrices utilisées dans les jeux vidéo, par exemple pour afficher correctement la scène devant la caméra, n'est pas évidente à faire à la main et dépasse le cadre de ce tutoriel.

Voyons donc tout de suite concrètement comment on peut donner l'impression que l'on se déplace dans la scène.

V-B - Créer une caméra

Déterminons tout de suite de quelles informations nous allons principalement avoir besoin pour notre caméra : position dans la scène et orientation, tout comme les objets de notre scène.

```
#include <QPointF>

class Camera
{
public:
    Camera();

    QPointF pos();
    qreal zpos();
    qreal rotationY();

    void setPos(QPointF pos);
    void setZPos(qreal pos);
    void setRotationY(qreal angle);

private:
    QPointF m_pos;
    qreal m_zpos;
    qreal m_rotationY;
    MyScene *m_scene;
};
```

Vous remarquerez aussi que j'ai ajouté un attribut m_zpos à la caméra.
C'est en fait une astuce que je voulais vous montrer et qui permettra de donner l'impression que l'on déplace celle-ci vers l'avant où l'arrière. Nous verrons comment faire lors de l'implémentation dans la scène. Sachez simplement que la valeur de cet attribut est modifiée lors de l'appui d'une touche, par exemple des flèches haut et bas ou bien la molette de la souris, au choix.

Ainsi, il n'y a pas grand-chose de plus à implémenter pour avoir une caméra de base. Notez qu'ici je ne considère qu'une rotation par rapport à l'axe Y, mais libre à vous d'ajouter les deux autres axes si vous le voulez.

L'avantage d'utiliser un objet caméra dans cette scène, c'est que vous n'avez que les paramètres de ladite caméra à modifier lorsque vous la tournez ou la déplacez, plutôt que de redéplacer tous les objets un par un.

MyScene.h

```
QList<MyProxy *> m_objets;
```

Pour ajouter un objet à la liste c'est très simple, vous pouvez utiliser l'opérateur de flux <<.

```
MyProxy *proxy;
// ...
m_objets << proxy;
```

Lorsque l'on modifie un paramètre de la caméra comme la rotation, on appelle une méthode updateScene() de la classe MyScene qui va parcourir tous les objets de la scène et appeler, pour chacun d'eux une méthode replacer() se chargeant d'appliquer les transformations dues à la caméra.

MyScene.cpp

```
void MyScene::updateScene()
{
    foreach (MyProxy *objet, m_objets)
    {
        objet->replacer();
    }
}
```

Il suffit maintenant d'ajouter un attribut caméra dans la classe MyProxy, et d'écrire la fonction replacer() comme suit.

```
void MyProxy::replacer()
{
    QTransform m;
    m.translate(this->center().x(), this->center().y());
    m.translate(-this->widget()->width()/2, -this->widget()->height()/2);

    // ajout des transformations liées à la position de la caméra
    m.translate(-m_camera.pos().x(), -m_camera.pos().y());
    m *= QTransform().rotate(m_camera.yaw(), Qt::YAxis);

    // simulation du déplacement suivant Z
    qreal scale = m_camera.zpos();
    m *= QTransform().scale(scale, scale);

    this->setTransform(m);
}
```

Enfin vous remarquez l'utilisation de la fonction QTransform::scale() qui a un paramètre dépendant de l'attribut m_zpos de ma caméra. En effet, si je considère que lorsque ma position en Z augmente je me rapproche de la scène, alors cela revient à voir les objets dans la scène en plus gros, donc à les agrandir ! Ici j'ai fait une implémentation assez basique pour que vous compreniez le principe, on peut cependant affiner le comportement en ajoutant des coefficients multiplicateurs pour gagner en précision.

VI - Conclusion

Voilà ! J'espère que les classes QGraphics**** de Qt vous paraissent plus familières et que vous avez une idée du genre de scènes qu'il est possible de réaliser.

Pour des notions plus exhaustives, vous pouvez de plus visitez la documentation sur ce module de Qt à l'adresse suivante : **http://doc.qt.nokia.com/4.7/graphicsview.html**

J'adresse aussi des remerciements tous particuliers à dourouc05 et gbdivers pour leurs relectures et conseils tout au long de la correction afin de proposer un cours le plus juste possible.