

Intégrer Ogre à Qt

par **Denys Bulant** ([Tutoriels Qt](#))

Date de publication : 11 juillet 2008

Dernière mise à jour : 29/05/2009


Ce tutoriel est destiné à illustrer l'intégration du moteur 3D Ogre à Qt. Les concepts utilisés ici sont tout à fait utilisables pour d'autres moteurs (comme IrrLicht) et/ou pour d'autres toolkit graphique (comme WxWidget ou gtkmm).

I - Introduction.....	3
I-1 - Pourquoi intégrer Ogre à Qt?.....	3
I-2 - Pré-requis.....	3
II - Initialiser le système.....	3
II-1 - Objectif: écran noir (mais grâce à Ogre!).....	4
II-2 - Première interaction: modification de la couleur de fond.....	6
III - Affichage d'un objet et gestion de la caméra.....	7
III-1 - Ajouter de la lumière et un objet.....	7
III-2 - Gérer la caméra (gestion des événements et utilisation d'un dockwidget).....	8
III-2-a - Contrôle du positionnement de la caméra par des spinbox.....	8
III-2-b - Contrôle du positionnement de la caméra par le clavier.....	10
III-2-c - Contrôle du positionnement de la caméra par la souris.....	12
IV - Sélection d'entité.....	13
V - Conclusion.....	14
VI - Pour les utilisateurs Qt 4.5.....	14
VII - Remerciements.....	15

I - Introduction

I-1 - Pourquoi intégrer Ogre à Qt?

Interagir avec un jeu ou une scène 3D se fait couramment par le biais d'interfaces. L'utilisation majoritaire de ces interfaces est plutôt à chercher du côté de la conception d'interface in-game (tels les systèmes d'inventaires ou les menus que vous pouvez voir dans n'importe quel jeu). A ces interfaces, il faut ajouter ce que les concepteurs vont utiliser, c'est-à-dire des outils qui vont leur faciliter la création de scènes ou de niveaux, de matériaux, etc... Vous pouvez penser à l'éditeur Aurora de Neverwinter Nights, ou encore à l'Unreal Editor.

Ces deux besoins d'interfaces appellent deux réponses différentes. Pour le premier cas d'utilisation (interfaces in-game), on peut trouver des frameworks très performants, tel  **CEGUI**, qui remplissent parfaitement leur rôle de framework d'interfaces dédiées aux menus de jeux. Cependant, pour des besoins plus lourds tels les éditeurs, les fonctionnalités fournies ne répondent plus vraiment aux besoins des concepteurs, et plus particulièrement des concepteurs d'outils. C'est pourquoi, afin de gagner en concision, en souplesse et afin de ne pas réinventer constamment la roue, il est indispensable d'intégrer le moteur graphique dans un framework de conception d'applications (tels Qt, WxWidgets, gtkmm, .Net etc...).

Ce tutoriel s'appliquera, vous l'aurez deviné, à l'intégration du moteur 3D Ogre dans Qt. Plus que l'intégration, nous verrons par la suite quelques cas standards (bien que basiques) d'utilisation.

I-2 - Pré-requis

- Une installation de Qt4 fonctionnelle
- une installation d'Ogre3D fonctionnelle (tuto réalisé avec la 1.4.5 et 1.4.8)
- En plus des bases en Qt (signaux/slots, création de widget), il est utile de connaître un minimum Ogre.

Avant d'entrer plus loin dans ce sujet, voici les tutoriaux qu'il est bon d'avoir suivi; les mécanismes présentés ici s'appuient sur des concepts basiques, mais les mécanismes spécifiques à Qt et Ogre seront assez peu abordés

-  **Tutoriaux Ogre sur DVP**
-  **Tutoriel avancé n°2** et  **n°3 sur le Wiki Ogre** (utilisation de RaySceneQuery utilisé dans la partie IV de ce tuto)

II - Initialiser le système

L'objectif de cette section sera d'obtenir un widget affichant effectivement ce que Ogre dessine. A la fin nous aurons donc un écran noir ainsi que la possibilité de changer la couleur de fond de notre viewport.

L'approche que je vais aborder ici est extrêmement simple, dans un souci de réduire le code montrant l'essentiel. Nous allons créer un widget stockant l'intégralité du système Ogre (un objet Root, scene manager etc...). Il est évident que dans un vrai code il est de loin préférable de découpler tout ceci, mais ce ne devrait pas être un problème une fois que vous aurez compris les principes. Il y a toutefois une précaution à prendre avant d'entrer dans le sujet. N'importez jamais le namespace Ogre dans le namespace global. Faire ceci vous mènera droit à des conflits de types définis en plusieurs endroits (par Ogre et Qt donc). Après vous avoir montré l'en-tête de la classe, je détaillerai les méthodes étape par étape.

En tête utilisé pour l'initialisation du système

```
class OgreWidget : public QWidget
{
    Q_OBJECT

public:
    OgreWidget(QWidget *parent = 0);
    ~OgreWidget();

public slots:
```

En tête utilisé pour l'initialisation du système


```
void setBackgroundColour(QColor c);

protected:
    virtual void paintEvent(QPaintEvent *e);
    virtual void resizeEvent(QResizeEvent *e);
    virtual void showEvent(QShowEvent *e);

private:
    void initOgreSystem();

private:
    Ogre::Root      *ogreRoot;
    Ogre::SceneManager *ogreSceneManager;
    Ogre::RenderWindow *ogreRenderWindow;
    Ogre::Viewport   *ogreViewport;
    Ogre::Camera      *ogreCamera;
};
```

II-1 - Objectif: écran noir (mais grâce à Ogre!)

Je vais détailler dans un premier temps la création du widget qui servira de réceptacle à l'affichage. Ce widget doit dériver de  **QWidget** (surpris? :)), et réimplémenter quelques méthodes; le minimum étant *showEvent*, *paintEvent*, *moveEvent* et *resizeEvent* (toutes 4 sont des méthodes virtuelles protégées). Ce widget devra avoir la propriété permettant de ne pas laisser le système dessiner un fond automatiquement: *Qt::WA_OpaquePaintEvent* et de ne pas laisser Qt gérer un double buffer: *Qt::WA_PaintOnScreen*.

```
OgreWidget::OgreWidget(QWidget *parent)
:QWidget(parent),
ogreRoot(0), ogreSceneManager(0), ogreRenderWindow(0), ogreViewport(0),
ogreCamera(0)
{
    setAttribute(Qt::WA_OpaquePaintEvent);
    setAttribute(Qt::WA_PaintOnScreen);
    setMinimumSize(240, 240);
}
```

Nous allons réimplémenter *showEvent* afin d'initialiser le système Ogre. En effet, son initialisation dépend du fait que nous ayons un handle système pour la fenêtre. Or cet handle n'est récupérable que lorsque la fenêtre est créée au niveau du système, et le moment le plus sûr est sa première apparition.

```
void OgreWidget::showEvent(QShowEvent *e)
{
    if(!ogreRoot)
    {
        initOgreSystem(); // l'initialisation d'ogre est détaillée plus bas
    }

    QWidget::showEvent(e);
}
```

paintEvent sera réimplémenté afin de demander un rafraîchissement du viewport d'Ogre. En effet, si Ogre n'est pas maître de la boucle d'événement (ce qui va être le cas), il faut demander explicitement la mise à jour des viewports utilisés. De plus, on informe le reste du moteur qu'une nouvelle frame est dessinée.

```
void OgreWidget::paintEvent(QPaintEvent *e)
{
    ogreRoot->_fireFrameStarted();
    ogreRenderWindow->update();
    ogreRoot->_fireFrameEnded();
}
```

```
e->accept();
}
```

Il est aussi nécessaire de réimplémenter *resizeEvent* et *moveEvent* afin de signifier à Ogre un changement au niveau de la résolution, et prendre en compte le nouvel aspect pour la caméra.

```
void OgreWidget::moveEvent(QMoveEvent *e)
{
    QWidget::moveEvent(e);

    if(e->isAccepted() && ogreRenderWindow)
    {
        ogreRenderWindow->windowMovedOrResized();
        update();
    }
}


void OgreWidget::resizeEvent(QResizeEvent *e)
{
    QWidget::resizeEvent(e);


    if(e->isAccepted())
    {
        const QSize &newSize = e->size();
        if(ogreRenderWindow)
        {
            ogreRenderWindow->resize(newSize.width(), newSize.height());
            ogreRenderWindow->windowMovedOrResized();
        }
        if(ogreCamera)
        {
            Ogre::Real aspectRatio = Ogre::Real(newSize.width()) / Ogre::Real(newSize.height());
            ogreCamera->setAspectRatio(aspectRatio);
        }
    }
}
```

Passons maintenant à l'initialisation d'Ogre. Si vous avez déjà programmé avec Ogre sans utiliser le framework d'exemple, vous vous y retrouverez très certainement. Dans le cas contraire, il y aura quelques différences avec un développement se basant sur ledit framework. Nous allons nous même créer un objet *Ogre::Root*, ainsi que le *Ogre::RenderingSystem* et le *Ogre::SceneManager*. L'ensemble du code concernant l'initialisation d'Ogre est celui de la fonction **OgreWidget::initOgreSystem()**.




La création de *Ogre::Root* est simplissime. Nous utiliserons ici le constructeur par défaut. Il faudra donc que vous fournissiez les fichiers "ogre.cfg" (bien que le contenu de celui-ci ne soit pas utilisé et n'est donc pas indispensable physiquement) et "plugins.cfg" (celui-ci reste par contre indispensable).

```
void OgreWidget::initOgreSystem()
{
    ogreRoot = new Ogre::Root();
```


Le *RenderSystem* se crée en demandant à root une instance d'un rendering manager correspondant à un nom donné. Ici, j'utiliserai directement le nom du rendersystem opengl pour rester cross platform et que nous n'ayons pas à proposer une liste de rendering system. Lors de l'appel à  **Root::initialise**, notez que nous spécifions que nous ne voulons pas qu'Ogre crée une fenêtre automatiquement.

Si vous voulez le faire dans votre propre outil, il vous faudra en récupérer la liste grâce à  **Ogre::Root::getAvailableRenderers**.

```
Ogre::RenderSystem *renderSystem = ogreRoot->getRenderSystemByName("OpenGL Rendering Subsystem");
ogreRoot->setRenderSystem(renderSystem);
ogreRoot->initialise(false);
```

Il n'y a pas de difficulté particulière au niveau de la création du scene manager. Ici encore, je code en dur le nom d'un SM standard afin de simplifier le code d'exemple. Pour proposer à l'utilisateur tout les SM disponibles, il vous faudra utiliser  **Ogre::SceneManagerEnumerator**. Contrairement aux rendersystems, il vous faudra obtenir un itérateur sur les méta-données des SceneManager, grâce à  **Ogre::SceneManagerEnumerator::getMetaDatalterator**. Une fois déréférencé, cet itérateur vous permettra d'obtenir un  **Ogre::SceneManagerMetaData** qui contiendra tout ce qui permet de décrire un SceneManager.

```
ogreSceneManager = ogreRoot->createSceneManager(Ogre::ST_GENERIC);
```

Vient maintenant le moment d'embarquer une vue Ogre au sein d'une fenêtre Qt : la création d'une **Ogre::RenderWindow**. Comme pour tout les objets vus précédemment, c'est sur demande à l'objet Root qu'une renderwindow est instanciée. Je vous encourage à jeter un oeil à  **la doc de cette méthode**. Nous allons donc devoir remplir une NameValuePairList dans laquelle nous spécifierons l'attribut *externalWindowHandle*, et bien sûr nous ne demanderons pas un affichage fullscreen.

```
Ogre::NameValuePairList viewConfig;
size_t widgetHandle;
#ifdef Q_WS_WIN
    widgetHandle = (size_t)((HWND)winId());
#else
    QWidget *q_parent = dynamic_cast<QWidget*>(parent());
    QX11Info xInfo = x11Info();


    widgetHandle = Ogre::StringConverter::toString((unsigned long)xInfo.display()) +
        ":" + Ogre::StringConverter::toString((unsigned int)xInfo.screen()) +
        ":" + Ogre::StringConverter::toString((unsigned long)q_parent->winId());
#endif
viewConfig["externalWindowHandle"] = Ogre::StringConverter::toString(widgetHandle);
ogreRenderWindow = ogreRoot->createRenderWindow("Ogre rendering window",
width(), height(), false, &viewConfig);
```

Il ne reste plus après qu'à créer une caméra (sur demande au SceneManager) et un viewport (sur demande auprès de l'objet RenderWindow nouvellement créé).

```
ogreCamera = ogreSceneManager->createCamera("myCamera");

ogreViewport = ogreRenderWindow->addViewport(ogreCamera);
ogreViewport->setBackgroundColour(Ogre::ColourValue(0,0,0));
ogreCamera->setAspectRatio(Ogre::Real(width()) / Ogre::Real(height()));
}
```

II-2 - Première interaction: modification de la couleur de fond

Les lecteurs attentifs auront remarqué dans l'en-tête la présence d'un slot **setBackgroundColour(QColor c)**. J'ai choisi QColor comme type afin de faciliter l'intégration avec le reste de Qt, et tout particulièrement  **QColorDialog**. Le seul problème de cette boîte de dialogue est qu'elle ne nous permettra pas de mettre à jour la vue en temps réel (elle est exécutée de façon modale, et n'émet pas de signaux correspondant à un changement de couleurs). Mais à titre d'exemple, cela suffira amplement. L'intérêt du slot est nul dans ce cas de figure, mais si vous faites votre propre boîte de dialogue non modale, simplement avoir un slot à connecter à un signal sera un plaisir! Il y a une chose utile à connaître ici : QColor stocke et permet d'obtenir la couleur grâce à un seul appel sous le format AARRGGBB. Ogre nous permet de spécifier une couleur dans ce format, cela nous évite donc de récupérer et spécifier chaque canal.

```
void OgreWidget::setBackgroundColour(QColor c)
{
```

```
if(ogreViewport)
{
    Ogre::ColourValue ogreColour;
    ogreColour.setAsARGB(c.rgb() );
    ogreViewport->setBackgroundColour(ogreColour);
}
```

Pour profiter de tout ça, il ne vous reste plus qu'à construire une QMainWindow avec notre OgreWidget pour widget principal. Pour spécifier la couleur, vous pouvez créer une action dont l'activation affichera une QColorDialog, et vous permettra d'en récupérer la valeur pour la spécifier à l'instance de OgreWidget.

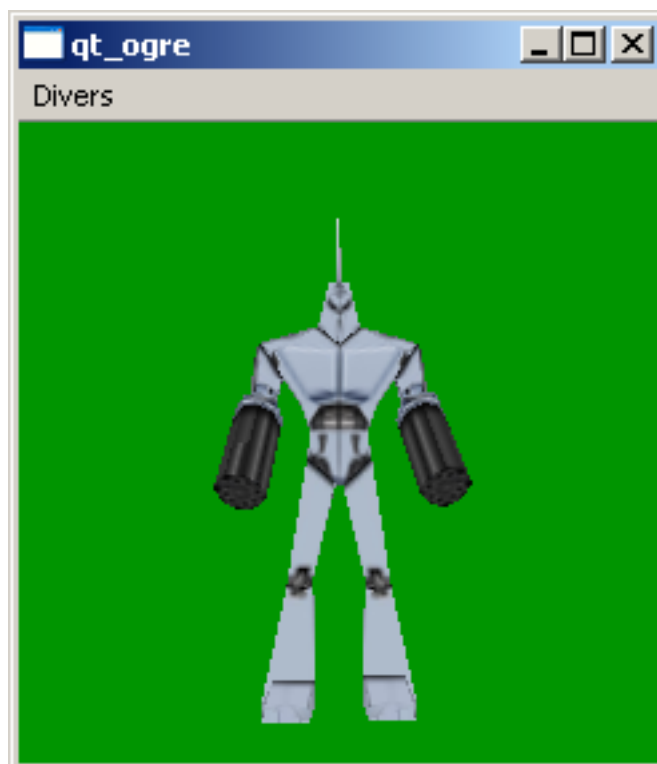
Une archive contenant tout le code nécessaire à cette partie est [disponible \(mirroir http\)](#). Prenez simplement garde à configurer les fichiers qt_ogre.pro et plugins.cfg selon vos besoins, et n'oubliez pas de le mettre dans le répertoire d'exécution de votre application.

III - Affichage d'un objet et gestion de la caméra

Le but à la fin de cette section sera d'être capable de gérer la position et l'orientation de la caméra à la souris, au clavier ou par des spinbox placées dans un dock widget. La première partie est dédiée à avoir un objet qui s'affiche. Le gros du travail sera fait dans la partie 2 où j'aborderai chacune des méthodes d'entrées.

III-1 - Ajouter de la lumière et un objet

Au code présenté ci-dessus, il va nous falloir ajouter une fonction qui va initialiser les ressources (pour l'objet et son material) ainsi qu'une fonction qui va créer la scène. La fonction initialisant les ressources n'est pas très intéressante en elle-même, il vous suffit de reprendre le code du framework d'exemple (ExampleApplication::setupResources dans le fichier ExampleApplication.h). Nous pourrions utiliser QSettings pour lire le fichier, mais il n'y a pas grand chose à y gagner ici. La scène créée sera très simple; il s'agit juste d'une lumière ambiante et d'un objet non animé. Le chargement des ressources et la création de la scène se font à la fin de notre fonction d'initialisation. Ce sont des manipulations purement liées à Ogre et l'intégration dans un toolkit graphique n'a pas la moindre importance. Vous devriez obtenir l'image suivante en compilant et en exécutant l'archive fournie [ici \(mirroir http\)](#). Dans cette dernière, j'ai ajouté le besoin du plugin Cg ainsi que les matériaux et objets nécessaires.



III-2 - Gérer la caméra (gestion des événements et utilisation d'un dockwidget)

Cette partie va vous permettre de voir plusieurs façons de manipuler un objet. Ici, nous allons voir comment déplacer la caméra de trois façons différentes:

- Utilisation de spinbox
- Utilisation du clavier
- Utilisation de la souris

Une fois que nous aurons vu le positionnement, vous ne devriez pas avoir de problème pour implémenter l'orientation. La première modification à apporter concerne la classe `OgreWidget`. En effet, quelle que soit la méthode, nous allons devoir communiquer par le biais d'un signal la nouvelle position de la caméra. Le but ici est de conserver les spinbox synchronisées avec la position de la caméra, quelque soit la méthode qui sert au déplacement. Voici la signature dudit signal:

```
signals:
    void cameraPositionChanged(const Ogre::Vector3 &pos);
```

III-2-a - Contrôle du positionnement de la caméra par des spinbox

Le but ici est d'avoir 3 `QDoubleSpinBox` qui nous permettront de spécifier la position sur chacun des 3 axes de la caméra.

Afin de simplifier ce contrôle, mais aussi dans le but de pouvoir le réutiliser pour autre chose que la caméra, vous pouvez créer un widget qui regroupera 3 de ces spinbox (la classe `Coordinate3DModifier` dans l'exemple). Ce widget s'est vu doté d'un signal `coordinateChanged(const Ogre::Vector3&)` qui est émis à chaque modification de la valeur de l'une des spinbox. La fonction suivante est donc ce slot appelé à chaque modification de la valeur de l'une des spinbox.

```
void Coordinate3DModifier::onCoordinateChanged()
{
    Ogre::Vector3 newCoord(sbx->value(),
                           sby->value(),
                           sbz->value());
    emit coordinateChanged(newCoord);
}
```

Ce signal est connecté à un slot qui a été créé dans notre `OgreWidget` dont voici le code:

```
void OgreWidget::setCameraPosition(const Ogre::Vector3 &pos)
{
    ogreCamera->setPosition(pos);
    ogreCamera->lookAt(0, 50, 0);
    update();
    emit cameraPositionChanged(pos);
}
```

Dans ce slot, nous repositionnons la caméra à l'endroit désiré, tout en lockant l'orientation vers le centre (approximativement) du robot (le modèle utilisé dans les exemples fournis). Nous demandons ensuite à mettre à jour l'affichage de notre widget, ce qui entraîne le rendu d'une frame par Ogre. Nous finissons par émettre la nouvelle position de la caméra à qui sera intéressé.

J'ai précisé plus haut que quelque soit la méthode de déplacement, il faut que les spinbox indiquant les coordonnées de la caméra restent synchrones avec sa position réelle. Nous allons donc connecter le signal `OgreWidget::cameraPositionChanged` à un slot qui s'occupera de mettre à jour les dites spinbox. Exemple tiré du code fourni plus bas:


```
void Coordinate3DModifier::setNewCoordinate(const Ogre::Vector3 &coordinate)
{
    blockSignals(true);
    sbx->setValue(coordinate.x);
    sby->setValue(coordinate.y);
    sbz->setValue(coordinate.z);
    blockSignals(false);
}
```

Je tiens à souligner l'utilisation des méthodes `QObject::blockSignals`. En effet, un appel à `QDoubleSpinBox::setValue` entraîne l'émission d'un signal `QDoubleSpinBox::valueChanged`. Ce signal entraîne l'exécution de notre slot `Coordinate3DModifier::onCoordinateChanged`. Ce dernier entraîne l'émission du signal `Coordinate3DModifier::coordinateChanged` et donc l'exécution du slot `OgreWidget::setCameraPosition`. Ce dernier va émettre le signal `OgreWidget::cameraPositionChanged`, lequel entraîne l'exécution du slot `Coordinate3DModifier::setNewCoordinate`. Si nous ne bloquons pas immédiatement tout déclenchement de signal, je vous laisse le soin d'imaginer la tête qu'aura la pile d'appel (enfin, le peu de temps qu'elle et le programme vont exister).

Il est donc assez important de documenter le fait que la modification des coordonnées par le slot `Coordinate3DModifier::setNewCoordinate` n'entraîne **pas** l'émission du signal `Coordinate3DModifier::coordinateChanged`. Mais que pour rester à l'écoute de la position de la caméra, il vaut mieux se connecter directement au signal fourni par `OgreWidget`.

Voici le résultat obtenu à la fin de cette partie:



Une archive contenant tout le code nécessaire à cette partie est [disponible \(mirroir http\)](#).

III-2-b - Contrôle du positionnement de la caméra par le clavier


Ici, nous allons voir un autre moyen de contrôler la position de la caméra. Le but est que si la touche reste enfoncée, le mouvement doit être continuellement répété. Les touches utilisées seront standard (désolé pour ceux qui utilisent un clavier QWERTY par contre, il vous faudra adapter):

- Z et S serviront à se déplacer en -Z et +Z respectivement

- Q et D pour -X et +X respectivement
- Pg Down et Pg Up pour -Y et +Y respectivement

L'implémentation de cette méthode demande 2 choses:

- définir une police de focus pour OgreWidget de façon à ce que Qt transmette les événements
- réimplémenter keyPressedEvent

La spécification de la police de focus est triviale; il s'agit juste d'un appel dans le constructeur de OgreWidget. Les flags à passer dépendent de ce que vous voulez assigner comme focus. Dans cet exemple, j'utiliserais simplement la prise de focus lors d'un clic de la souris dans le widget (pour les autres types de prises de focus possibles, je vous renvoie à la liste des  **Qt::FocusPolicy**). Le constructeur d'OgreWidget se présente ainsi à cette étape:

```
OgreWidget::OgreWidget(QWidget *parent)
:QWidget(parent),
ogreRoot(0), ogreSceneManager(0), ogreRenderWindow(0), ogreViewport(0),
ogreCamera(0)
{
    setAttribute(Qt::WA_OpaquePaintEvent);
    setAttribute(Qt::WA_PaintOnScreen);
    setMinimumSize(240, 240);

    setFocusPolicy(Qt::ClickFocus);
}
```

Quant au déplacement proprement dit, tous les déplacements sont codés en dur, toujours à titre d'exemple. Le principe est assez facile. Trop pour une utilisation sérieuse : nous stockons sous forme de map une correspondance entre une touche et un vecteur, et nous ne nous déplaçons que sur un seul axe à la fois (celui correspondant à la dernière touche pressée). La map en question contient donc un keycode allié à un vecteur de déplacement qui est relatif à la position actuelle de la caméra. Ce vecteur est ensuite ajouté à la position actuelle de la caméra et est ensuite transmis à la fonction *OgreWidget::setCameraPosition* afin de continuellement verrouiller l'orientation de la caméra vers le point qui a été spécifié, demander un rafraichissement, ainsi qu'émettre la nouvelle position de la caméra (ceci dans le but de garder les spinbox indiquant la position synchrone avec la position).

Voici le code de la fonction permettant ceci:

```
void OgreWidget::keyPressEvent(QKeyEvent *e)
{
    static QMap<int, Ogre::Vector3> keyCoordModificationMapping;
    static bool mappingInitialised = false;

    if(!mappingInitialised)
    {
        keyCoordModificationMapping[Qt::Key_Z] = Ogre::Vector3( 0, 0, -5);
        keyCoordModificationMapping[Qt::Key_S] = Ogre::Vector3( 0, 0, 5);
        keyCoordModificationMapping[Qt::Key_Q] = Ogre::Vector3(-5, 0, 0);
        keyCoordModificationMapping[Qt::Key_D] = Ogre::Vector3( 5, 0, 0);
        keyCoordModificationMapping[Qt::Key_PageUp] = Ogre::Vector3( 0, 5, 0);
        keyCoordModificationMapping[Qt::Key_PageDown] = Ogre::Vector3( 0, -5, 0);

        mappingInitialised = true;
    }

    QMap<int, Ogre::Vector3>::iterator keyPressed =
        keyCoordModificationMapping.find(e->key());
    if(keyPressed != keyCoordModificationMapping.end() && ogreCamera)
    {
        const Ogre::Vector3 &actualCamPos = ogreCamera->getPosition();
        setCameraPosition(actualCamPos + keyPressed.value());

        e->accept();
    }
    else
```

```

{
    e->ignore();
}

```

Une archive contenant tout le code nécessaire à cette partie est **disponible** ([mirroir http](#)).


III-2-c - Contrôle du positionnement de la caméra par la souris

La dernière méthode de positionnement de la caméra est donc l'utilisation de la souris. Voici le contrôle décidé :

- Roulette vers l'avant: déplacement vers -Z
- Roulette vers l'arrière: déplacement vers +Z
- LMB et déplacement vers la gauche: déplacement vers -X
- LMB et déplacement vers la droite: déplacement vers +X
- LMB et déplacement vers le haut: déplacement vers +Y
- LMB et déplacement vers le bas: déplacement vers -Y

Si vous appuyez sur la touche Ctrl en même temps, le déplacement sera multiplié par 10 (défini comme la constante de classe `turboModifier`). Il va nous falloir réimplémenter 4 fonctions pour ce faire :

- `wheelEvent`
- `mouseMoveEvent`
- `mousePressEvent`
- `mouseReleaseEvent`

Dans un premier temps, nous allons nous occuper de la roulette (peu importe laquelle, nous ne tenons pas compte de plusieurs roulettes). Le réglage du code ci-dessous permet de se déplacer de 2 unités à chaque pas de roulette franchi (bien qu'avec certaines souris, il soit possible que ce soit une autre valeur). Pour régler vous même la sensibilité, je vous renvoie à la documentation de la fonction  `QWheelEvent::delta()`. La façon de procéder reste similaire au déplacement par le clavier, à savoir la création d'un vecteur de translation relatif à la position actuelle, puis addition de ce vecteur à la position actuelle de la caméra pour en définir la nouvelle position :

```

void OgreWidget::wheelEvent(QWheelEvent *e)
{
    Ogre::Vector3 zTranslation(0,0, -e->delta() / 60);

    if(e->modifiers().testFlag(Qt::ControlModifier))
    {
        zTranslation.z *= turboModifier;
    }

    const Ogre::Vector3 &actualCamPos = ogreCamera->getPosition();
    setCameraPosition(actualCamPos + zTranslation);

    e->accept();
}

```

Maintenant, nous allons aborder le déplacement sur les axes XY en maintenant le bouton gauche de la souris enfoncé et en déplaçant la souris. Ceci nécessite l'utilisation d'une variable qui va stocker le dernier point connu du mouvement de la souris. Il est égal à `QPoint(-1,-1)` (ce que j'ai défini ici comme point invalide) au début du programme et à chaque fois que l'utilisateur relâche le bouton gauche de la souris (passage dans `OgreWidget::mouseReleaseEvent`). Lors de l'appui sur le bouton gauche (`OgreWidget::mousePressEvent`), nous stockons la position de la souris. Ensuite, à chaque passage dans la fonction `OgreWidget::mouseMoveEvent`, il nous suffit de récupérer le delta entre ces 2 points, d'éventuellement le multiplier par 10 si la touche Ctrl est enfoncée, et de définir le vecteur de translation. Ce dernier est, comme à l'accoutumée, ajouté à la position actuelle de la caméra pour en définir une nouvelle. Le code nécessaire est présenté ici (`oldPos` est la variable stockant l'ancienne position du curseur) :

```

void OgreWidget::mouseMoveEvent(QMouseEvent *e)
{
    if(e->buttons().testFlag(Qt::LeftButton) && oldPos != invalidMousePoint)
    {
        const QPoint &pos = e->pos();
        Ogre::Real deltaX = pos.x() - oldPos.x();
        Ogre::Real deltaY = pos.y() - oldPos.y();

        if(e->modifiers().testFlag(Qt::ControlModifier))
        {
            deltaX *= turboModifier;
            deltaY *= turboModifier;
        }

        Ogre::Vector3 camTranslation(deltaX, deltaY, 0);
        const Ogre::Vector3 &actualCamPos = ogreCamera->getPosition();
        setCameraPosition(actualCamPos + camTranslation);

        oldPos = pos;
        e->accept();
    }
    else
    {
        e->ignore();
    }
}

void OgreWidget::mousePressEvent(QMouseEvent *e)
{
    if(e->buttons().testFlag(Qt::LeftButton))
    {
        oldPos = e->pos();
        e->accept();
    }
    else
    {
        e->ignore();
    }
}

void OgreWidget::mouseReleaseEvent(QMouseEvent *e)
{
    if(!e->buttons().testFlag(Qt::LeftButton))
    {
        oldPos = QPoint(invalidMousePoint);
        e->accept();
    }
    else
    {
        e->ignore();
    }
}

```

Une archive contenant tout le code nécessaire à cette partie est **disponible** ([mirroir http](http://www.irmatden.developpez.com/tutoriels/qt/integration-ogre-qt/)).

IV - Sélection d'entité

Le but de cette partie est d'arriver à sélectionner notre robot en double cliquant dessus avec le bouton gauche, et à le désélectionner en double cliquant ailleurs que dans le volume de sa bounding box. Petit rappel sur le mécanisme : Ogre permet ceci par un RaySceneQuery qui collecte toute géométrie dont la bounding box est traversée par un rayon. Nous allons construire le rayon grâce à la caméra qui attend des coordonnées x et y comprises entre 0 et 1 (ie, ratio entre la position "physique" et la largeur de la fenêtre). Il ne nous reste plus alors qu'à vérifier que le rayon a bien intersecté un MovableObject, auquel cas, ce ne peut être que notre robot. Il ne reste plus qu'à récupérer la SceneNode associée, et à lui demander d'afficher la bounding box qui lui est associée. Si aucune intersection n'a lieu, nous désélectionnons l'objet SceneNode précédemment sélectionné. Un petit rafraîchissement de l'affichage à

la fin de la fonction, et le tour est joué. La construction des coordonnées relatives se fait simplement en prenant la position du clic et en divisant par la largeur puis la hauteur.

Voici le code de cette partie (selectedNode est un *Ogre::SceneNode* membre de la classe qui n'est manipulé qu'ici):

```
void OgreWidget::mouseDoubleClickEvent(QMouseEvent *e)
{
    if(e->buttons().testFlag(Qt::LeftButton))
    {
        Ogre::Real x = e->pos().x() / (float)width();
        Ogre::Real y = e->pos().y() / (float)height();

        Ogre::Ray ray = ogreCamera->getCameraToViewportRay(x, y);
        Ogre::RaySceneQuery *query = ogreSceneManager->createRayQuery(ray);
        Ogre::RaySceneQueryResult &queryResult = query->execute();
        Ogre::RaySceneQueryResult::iterator queryResultIterator = queryResult.begin();

        if(queryResultIterator != queryResult.end())
        {
            if(queryResultIterator->movable)
            {
                selectedNode = queryResultIterator->movable->getParentSceneNode();
                selectedNode->showBoundingBox(true);
            }
        }
        else
        {
            selectedNode->showBoundingBox(false);
            selectedNode = 0;
        }

        ogreSceneManager->destroyQuery(query);

        update();
        e->accept();
    }
    else
    {
        e->ignore();
    }
}
```

Une archive contenant tout le code nécessaire à cette partie est **disponible** ([mirroir http](http://www.irmatden.developpez.com/tutoriels/qt/integration-ogre-qt/)).

V - Conclusion

Bravo si vous avez eu le courage de lire jusqu'ici, ce tuto n'est pas des plus courts! :)

J'espère avoir ici réussi à faire le tour des principales méthodes pour vous permettre de faire cohabiter Qt et Ogre. Comme je l'ai précisé au début, cette approche n'est pas idéale dans le cadre d'un vrai programme. Son aspect monolithique sera très vite paralysant, et les interactions possibles se densifieront avec les features. De plus, il est impossible avec ce code d'avoir plusieurs widgets affichant la même scène.

VI - Pour les utilisateurs Qt 4.5


Avec Qt 4.5, le code de ce tuto entraîne un flickering. Le fix consiste à surcharger la méthode `paintEngine()` dans le widget Ogre de cette façon:

```
QPaintEngine *OgreWidget:: paintEngine() const
{
    return 0;
}
```

VII - Remerciements

Merci à Trolltech et les développeurs Ogre pour leur fantastique toolkit respectif.

Ainsi qu'à Raptor70, Loka, Alp et Mongaulois et Diogene pour leur relectures.

Merci à Caesius pour avoir donné l'info concernant Qt 4.5 (origine:  **forum officiel Ogre**).