


Intégrer Ogre à Qt - Partie 2

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

Date de publication : 24/03/2009

Dernière mise à jour : 29/05/2009

Ce tutoriel fait suite à  **"Intégrer Ogre à Qt"**. A la fin de ce dernier, nous avons obtenu un embryon d'outil, permettant assez peu de choses tel quel, mais ayant fourni les bases principales requises. Faisant suite à ce tutoriel, je vais présenter ici 2 améliorations. La première consiste à simplifier le travail à fournir par le développeur quant aux actions à implémenter, tandis que la seconde se penche sur une amélioration critique pour tout utilisateur: la possibilité d'annuler ses actions.
Nous repartons ici du code final du tutoriel précédent dont voici le [lien](#) ([mirroir http](#)).


I - Une autre gestion des événements.....	3
I-1 - Inconvénients de l'approche précédente et motif pour changer.....	3
I-2 - Implémentation.....	3
I-2-1 - Définition de l'interface nécessaire et modification au code actuel.....	3
I-2-2 - Implémentation du déplacement de la caméra.....	4
I-2-3 - Implémentation du déplacement de l'objet sélectionné.....	7
II - Support de l'annulation.....	9
II-1 - Présentation du framework Undo fourni par Qt.....	9
II-2 - Implémentation.....	10
II-2-1 - Préparatif.....	10
II-2-2 - Annulation du changement de couleur de fond.....	11
II-2-3 - Annulation du déplacement d'un objet.....	13
III - Pour les utilisateurs Qt 4.5.....	17
IV - Conclusion.....	17

I - Une autre gestion des événements

I-1 - Inconvénients de l'approche précédente et motif pour changer

A la fin de l'article précédent, nous pouvions déplacer la caméra par 3 méthodes différentes. Cependant, l'approche qui a été choisie n'est pas viable à long terme dû à une conception trop monolithique. En effet, simplement ajouter la possibilité de déplacer l'entité sélectionnée demanderait de complexifier les implémentations d'événements existants. Le résultat serait des fonctions particulièrement "bloatée", et au mieux un enchaînement de if ou un switch pour basculer entre divers gestionnaires d'événements selon le but à atteindre.

Nous allons changer cette approche pour une méthode plus souple. Cette méthode mettra en oeuvre le design pattern Stratégie afin de découpler au maximum les réponses aux événements du widget en lui même. Cette technique va nous permettre de déléguer la gestion des événements du widget. Chaque manipulation du widget se verra isolée dans sa propre classe, et les interactions possibles en hériteront. Du point de vue de la classe OgreWidget, cela nécessite de garder à jour un pointeur sur la classe de base afin de lui transférer les événements (si délégué actif il y a; nous nous laissons la possibilité de ne pas avoir de gestion d'événements en cas de pointeur nul).

Pour plus d'infos sur le pattern Stratégie ainsi que des exemples en C++, je vous recommande de lire  [cette section d'un tutoriel de David Come](#).

I-2 - Implémentation

I-2-1 - Définition de l'interface nécessaire et modification au code actuel

Nous allons donc définir une classe indiquant toutes les fonctions requises à un gestionnaire d'entrées. Il s'agira dans notre cas des méthodes interagissant sur la scène, à l'exception du double clic (ce dernier est "réservé" à la (dé)sélection d'entité). La classe en question est déclarée ainsi:

```
class EventHandler
{
public:
    virtual bool keyPressEvent(QKeyEvent *e);
    virtual bool mouseMoveEvent(QMouseEvent *e);
    virtual bool mousePressEvent(QMouseEvent *e);
    virtual bool mouseReleaseEvent(QMouseEvent *e);
    virtual bool wheelEvent(QWheelEvent *e);
};
```

Ces méthodes ne sont pas sensées spécifier si l'événement est accepté ou non. Cette réponse est laissée au soin de notre widget qui indiquera l'acceptation de l'événement selon que la fonction renvoie **true** ou **false**. Vous l'aurez peut-être remarqué, EventHandler n'est pas une classe abstraite, mais une classe no-op. En effet, chacune des méthodes est implémentée ici pour retourner false. Ça permet de ne réimplémenter qu'une partie des fonctions dans les classes dérivées. Je me passerais donc ici de montrer l'implémentation (une série de return false n'étant pas des plus intéressante ;)).

Il va nous falloir maintenant intégrer ce type à **OgreWidget**. Pour ce faire, il faut commencer par ajouter une variable à cette classe sous la forme d'un pointeur; ceci nous permettra de changer le gestionnaire d'événements selon les besoins. Le corps des méthodes gérant les événements listés dans EventHandler est donc tous remplacé par un code similaire à celui-ci (aucun intérêt à tous les lister):

```
void OgreWidget::keyPressEvent(QKeyEvent *e)
{
    if(eventHandler && eventHandler->keyPressEvent(e))
    {
        e->accept();
    }
    else
```

```
{  
    e->ignore();  
}
```

La vérification sur l'existence d'un eventHandler (en supposant que vous assigniez bien vos pointeurs à 0 une fois supprimés) permet de facilement assigner à OgreWidget un EventHandler (ou classe dérivée) qui ne fait rien, sans pour autant assigner un EventHandler. Il nous faut aussi une méthode pour changer dynamiquement ce comportement, et c'est réalisé par la fonction *OgreWidget::setEventHandler*.

```
void OgreWidget::setEventHandler(EventHandler *newEventHandler)  
{  
    delete eventHandler;  
    eventHandler = newEventHandler;  
}
```

I-2-2 - Implémentation du déplacement de la caméra

Après avoir apporté les modifications ci-dessus, nous ne pouvons plus que sélectionner et désélectionner le robot. Le but de cette section est donc de rétablir le comportement précédent. Cependant, en ayant délocalisé la gestion des entrées, nous n'avons plus la main sur l'ensemble des chemins possibles déclenchant une modification de la caméra. Il en résulte une impossibilité de garder la position synchrone avec les spinbox. Nous allons donc découpler un peu tout ça, et commencer par créer un wrapper pour la caméra. Nous n'allons redéfinir que les fonctions que nous allons utiliser ici, et en profiter pour implémenter des slots et signaux:

```
class MyCamera : public QObject  
{  
    Q_OBJECT  
  
public:  
    MyCamera(Ogre::Camera &camera, QObject *parent = 0);  
  
    Ogre::Camera* getOgreCamera() const;  
    const Ogre::Vector3& getPosition() const;  
    Ogre::Ray getCameraToViewportRay(Ogre::Real screenx, Ogre::Real screeny) const;  
  
public slots:  
    void setPosition(const Ogre::Vector3 &pos);  
    void lookAt(const Ogre::Vector3 &targetPoint);  
    void setAspectRatio(Ogre::Real ratio);  
  
signals:  
    void positionChanged(const Ogre::Vector3 &pos);  
    void visiblePropertyChanged();  
  
private:  
    Ogre::Camera *ogreCamera;  
};
```

Quelques explications sur cette classe restent à faire. Le constructeur prend une référence afin de rendre impossible le passage d'un pointeur null. Les setters sont implémentés comme des slots afin de faciliter le réglage par des widgets externes afin de ne pas encombrer la classe principale (ici OgreWidget) de code qui ne lui est pas spécifique. Le signal visiblePropertyChanged() est là pour signaler qu'un changement a eu lieu sur une propriété qui modifie potentiellement le rendu (changement de position, d'orientation, etc.. etc...). Les getters ne sont pas vraiment intéressants, ils se contentent de transmettre l'appel à la caméra Ogre. L'implémentation des setters par contre se présente ainsi:

```
void MyCamera::setPosition(const Ogre::Vector3 &pos)  
{  
    ogreCamera->setPosition(pos);  
}
```

```

    lookAt(Ogre::Vector3(0,50,0));

    emit positionChanged(pos);
    emit visiblePropertyChanged();
}

void MyCamera::lookAt(const Ogre::Vector3 &targetPoint)
{
    ogreCamera->lookAt(targetPoint);
    emit visiblePropertyChanged();
}

void MyCamera::setAspectRatio(Ogre::Real ratio)
{
    ogreCamera->setAspectRatio(ratio);
    emit visiblePropertyChanged();
}

```

Après ces modifications, il va falloir nettoyer un peu la classe OgreWidget afin de ne pas laisser de code inutile. Les modifications consistent à:

- Remplacer le pointeur `Ogre::Camera*` par un `MyCamera*`
- Ajouter une méthode permettant d'obtenir le pointeur vers `MyCamera` associé à la vue
- Supprimer le slot `setCameraPosition` ainsi que le signal `cameraPositionChanged`
- Supprimer les variables liées à la gestion du déplacement de la caméra (`oldPos`, `turboModifier` et `invalidMousePoint`)
- Et ajouter une méthode `setupCamera()` afin de séparer un peu la création de la caméra de l'initialisation dans un souci de propreté. Voici le code de cette méthode:

```

void OgreWidget::setupCamera()
{
    delete camera;
    camera = new MyCamera(*ogreSceneManager->createCamera("myCamera"), this);

    connect(camera, SIGNAL(visiblePropertyChanged()), this, SLOT(update()));

    camera->setPosition(Ogre::Vector3(0, 50,150));
    camera->lookAt(Ogre::Vector3(0,50,0));
    camera->setAspectRatio(Ogre::Real(width()) / Ogre::Real(height()));
}

```

Et la déclaration actuelle de `OgreWidget` est donc:

```

class OgreWidget : public QWidget
{
[...]
    MyCamera *getCamera();
    void setEventHandler(EventHandler *newEventHandler);

[...]

private:
[...]
    void setupCamera();

private:
    Ogre::Root          *ogreRoot;
    Ogre::SceneManager  *ogreSceneManager;
    Ogre::RenderWindow  *ogreRenderWindow;
    Ogre::Viewport       *ogreViewport;

    EventHandler        *eventHandler;
    MyCamera             *camera;

    Ogre::SceneNode      *selectedNode;

```

```
};
```

Il nous reste maintenant 2 étapes à franchir avant de pouvoir à nouveau déplacer la caméra: créer un event handler permettant d'agir sur la caméra, et modifier la classe *MainWindow* afin de prendre tout ceci en compte.

Nous allons commencer par créer la classe *CameraEventHandler*, qui va permettre de déplacer la caméra par les méthodes déjà présentées. Cette classe reprend tout simplement l'ancien corps des méthodes d'événements, à l'exception des `event->accept()` et `event->ignore()`. Je ne montre ici que la déclaration puisque l'implémentation est déjà connue. En cas de doute, n'hésitez pas à vous référer à l'archive indiquée en fin de paragraphe.

```
class CameraEventHandler : public EventHandler
{
public:
    CameraEventHandler(MyCamera *targetCamera);

    virtual bool keyPressEvent(QKeyEvent *e);
    virtual bool mouseMoveEvent(QMouseEvent *e);
    virtual bool mousePressEvent(QMouseEvent *e);
    virtual bool mouseReleaseEvent(QMouseEvent *e);
    virtual bool wheelEvent(QWheelEvent *e);

private:
    static const Ogre::Real turboModifier;
    static const QPoint invalidMousePoint;

private:
    QPoint oldPos;
    MyCamera *camera;
};
```

Nous allons finir avec les modifications à apporter à la classe *MainWindow*. Nous allons activer/désactiver le déplacement de la caméra par le biais d'un menu. Il nous faut donc créer une nouvelle action et l'ajouter au menu Divers. Lorsque l'état de cette action est basculé, ce sera le (nouveau) slot *moveCamModeToggled(bool)* qui sera exécuté. Selon l'état de l'action nous allons activer ou désactiver le déplacement de la caméra, afficher ou masquer le dock contenant notre widget modifiant des coordonnées et connecter les changements de coordonnées entre la caméra et le widget *Coordinate3DModifier*. Voici les méthodes qui ont changées:

```
MainWindow()
:OgreWidget(0)
{
    OgreWidget = new OgreWidget;
    camPosModifier = new Coordinate3DModifier;

    createActionMenus();
    createDockWidget();

    setCentralWidget(OgreWidget);
}

[...]
```

```
void createActionMenus()
{
    QAction *changeColorAct = new QAction("Changer la couleur de fond", this);
    connect(changeColorAct, SIGNAL(triggered()), this, SLOT(chooseBgColor()));

    QAction *moveCamModeAct = new QAction("Déplacement de la camera", this);
    moveCamModeAct->setCheckable(true);
    moveCamModeAct->setChecked(false);
    connect(moveCamModeAct, SIGNAL(toggled(bool)), this, SLOT(moveCamModeToggled(bool)));

    QAction *closeAct = new QAction("Quitter", this);
    connect(closeAct, SIGNAL(triggered()), this, SLOT(close()));

    QMenu *menu = menuBar()->addMenu("Divers");
    menu->addAction(changeColorAct);
```

```

        menu->addAction(moveCamModeAct);
        menu->addAction(closeAct);
    }

[...]
```

private slots:

```

[...]
```

void moveCamModeToggled(bool on)

```

{
    if(on)
    {
        MyCamera *cam = ogreWidget->getCamera();
        CameraEventHandler *camEventHandler = new CameraEventHandler(cam);
        ogreWidget->setEventHandler(camEventHandler);

        coordModifier->setNewCoordinate(cam->getPosition());
        coordModifierDock->setVisible(true);

        connect(coordModifier, SIGNAL(coordinateChanged(const Ogre::Vector3&)),
                cam, SLOT(setPosition(const Ogre::Vector3&)));
        connect(cam, SIGNAL(positionChanged(const Ogre::Vector3&)),
                coordModifier, SLOT(setNewCoordinate(const Ogre::Vector3&)));
    }
    else
    {
        ogreWidget->setEventHandler(0);

        coordModifierDock->setVisible(false);
        coordModifier->disconnect();
    }
}

```

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

I-2-3 - Implémentation du déplacement de l'objet sélectionné

Le processus à suivre suit de très près celui utilisé pour la caméra ; je plongerais donc un peu moins dans les détails. A l'image de la caméra pour laquelle nous avons créé un wrapper, nous allons en créer un pour la classe `Ogre::SceneNode`. Il s'agit simplement d'une version "allégée" du wrapper de la caméra dont voici la déclaration :

```

class MySceneNode : public QObject
{
    Q_OBJECT

public:
    MySceneNode(Ogre::SceneNode &node, QObject *parent = 0);

    Ogre::SceneNode* getOgreSceneNode() const;
    const Ogre::Vector3& getPosition() const;

public slots:
    void setPosition(const Ogre::Vector3 &pos);

signals:
    void positionChanged(const Ogre::Vector3 &pos);

private:
    Ogre::SceneNode *ogreSceneNode;
};

```

L'implémentation de cette classe est particulièrement triviale (il s'agit simplement de transférer les appels au `SceneNode` wrappé).


C'est cette classe qui va être utilisée pour stocker l'objet sélectionné. Nous allons commencer par mettre à jour le code de `OgreWidget::mouseDoubleClickEvent()` afin d'utiliser cette nouvelle classe. Voici le snippet concerné:

```
if(queryResultIterator != queryResult.end())
{
    if(queryResultIterator->movable)
    {
        Ogre::SceneNode *node = queryResultIterator->movable->getParentSceneNode();
        node->showBoundingBox(true);
        delete selectedNode;
        selectedNode = new MySceneNode(*node, this);
    }
}
else
{
    if (selectedNode)
    {
        selectedNode->getOgreSceneNode()->showBoundingBox(false);
        delete selectedNode;
        selectedNode = 0;
    }
}
```

Maintenant, il nous faut une dernière modification à `OgreWidget` afin de permettre à l'interface de déplacer l'objet. Nous ajoutons une méthode (`OgreWidget::getSelectedSceneNode()`) nous permettant d'obtenir l'objet sélectionné. Passons maintenant à l'event handler à qui l'on délègue la gestion des événements pour déplacer le noeud concerné. Le comportement est semblable à l'event handler, à 3 exceptions près:

- il n'y a pas de modificateur "turbo", et le déplacement se fait par incrément d'une unité,
- le déplacement à la souris se fait sur le plan XZ, contrairement à la caméra où la translation se fait en XY,
- les événements de la molette sont ignorés (`SelectedNodeEventHandler::wheelEvent()` renvoie false).

Il ne nous reste donc plus qu'à créer une entrée dans le menu afin de connecter tout ça. Cette action affichera le widget `Coordinate3DModifier`, mais cette fois, il affichera et permettra de mettre à jour la position du noeud sélectionné. Cette entrée de menu est mutuellement exclusive avec celle permettant de déplacer la caméra.

Pour implémenter cette exclusivité, nous allons utiliser un  **QActionGroup**. Cette classe nous simplifiera la vie en nous permettant de ne pas nous soucier de désactiver l'event handler que nous aurions pu activer précédemment, ni de décocher l'entrée du menu correspondante. Voici le code qui fait ceci:

```
QActionGroup *actionGroup = new QActionGroup(this);
actionGroup->addAction(moveCamModeAct);
actionGroup->addAction(moveSelNodeModeAct);
```

Oui oui, c'est tout :)

moveSelNodeModeAct est une action qui est créée de la même façon que *moveCamNodeAct*. Seul son texte ainsi que son slot change. Slot dont voici d'ailleurs le code:

```
void MainWindow::moveSelectedNodeModeToggled(bool on)
{
    if(on)
    {
        MySceneNode *selNode = ogreWidget->getSelectedSceneNode();
        if (!selNode)
        {
            moveSelNodeModeAct->setChecked(false);

            // On récurse afin de nettoyer correctement les signaux/slots et event handlers
            moveSelectedNodeModeToggled(false);
            return;
        }
        SelectedNodeEventHandler *selNodeEventHandler = new SelectedNodeEventHandler(selNode);
```



```

ogreWidget->setEventHandler(selNodeEventHandler);

coordModifier->disconnect();
coordModifier->setNewCoordinate(selNode->getPosition());
coordModifierDock->setVisible(true);
coordModifierDock->setWindowTitle("Selected node position");

connect(coordModifier, SIGNAL(coordinateChanged(const Ogre::Vector3&)),
        selNode, SLOT(setPosition(const Ogre::Vector3&)));
connect(selNode, SIGNAL(positionChanged(const Ogre::Vector3&)),
        coordModifier, SLOT(setNewCoordinate(const Ogre::Vector3&)));
connect(selNode, SIGNAL(positionChanged(const Ogre::Vector3&)),
        ogreWidget, SLOT(update()));
}
else
{
    ogreWidget->setEventHandler(0);

    coordModifierDock->setVisible(false);
    coordModifier->disconnect();
}
}

```

Nous commençons bien sûr par nous assurer qu'un objet est bel et bien sélectionné; dans le cas contraire nous décochons le menu pour signifier que la demande n'est pas acceptée. En l'état, vous pouvez commencer à tester, mais il nous reste un dernier problème à résoudre.

En effet, si vous sélectionnez un objet, que vous entrez en mode de déplacement d'objet, et que vous le désélectionnez, tout événement que nous traitons entraînera un crash puisque le sceneNode piloté est détruit par OgreWidget. Nous allons donc ajouter un signal à cette classe afin de notifier d'un changement de sélection. Voici le signal ajouté:

```

signals:
    void selectionChanged(MySceneNode *newSelectedNode);

```

Nous passons en paramètre le nouveau noeud par commodité bien qu'il n'y en ait pas d'utilité dans le cas présent. Il nous faut maintenant écrire un slot dans MainWindow afin de mettre à jour l'interface selon le besoin:

```

void MainWindow::selectionChanged(MySceneNode * /*newSelectedNode*/)
{
    if (moveSelNodeModeAct->isChecked())
        moveSelectedNodeModeToggled(true);
}



```




Ce slot se contente de mettre à jour l'event handler selon la sélection si nous sommes dans le mode de déplacement d'objet. Maintenant, nous émettons ce signal à tout changement de sélection (une petite modification de OgreWidget::mouseDoubleClickEvent() est nécessaire). Enfin, après avoir connecté le signal fourni par OgreWidget à MainWindow::selectionChanged(MySceneNode *), il ne nous reste plus qu'à compiler et exécuter notre petite application afin de constater que nous pouvons tour à tour modifier la position de la caméra et de l'objet sélectionné. Le code illustrant ce paragraphe est **disponible** ([miroir http](http://miroir)).


II - Support de l'annulation

II-1 - Présentation du framework Undo fourni par Qt

La possibilité d'annuler et réappliquer des modifications dans un document est devenue extrêmement courant. C'est en effet une fonctionnalité essentielle pour faciliter l'utilisation de votre logiciel. Les éditeurs 3D ou encore de niveaux n'échappent pas à la règle, nous allons donc voir comment implémenter ce système avec Qt.

Le framework Undo de Qt est basé sur le design pattern "Command". Je vous encourage à lire la définition écrite par Baptiste Wicht sur  [ce même site](#). Pour une description plus détaillée avec un exemple complet, vous pouvez vous référer à  [cette page](#).

L'idée est donc de wrapper nos actions sur la scène dans des objets, des instances d'une classe dérivant de  **QUndoCommand**. Lors de toute modification de la scène, un tel objet sera donc créé de façon à définir comment annuler la modification, ainsi que la façon de l'appliquer. Chacun de ces objets sera poussé dans une  **QUndoStack**. Cette dernière est en générale propre à chaque document. La dernière classe que nous utiliserons ici sera  **QUndoView**. Elle permet d'afficher le contenu d'une QUndoStack, ainsi que d'y naviguer (c'est-à-dire cliquer sur un état pour voir le document tel qu'il l'était lors de ce snapshot).

Dans le cas d'une application permettant de gérer plusieurs documents, l'utilisation de  **QUndoGroup** est recommandée. L'expression "plusieurs documents" est à prendre au sens général du terme. Prenons un éditeur 3D: vous pouvez vouloir gérer une pile d'undo pour les modifications de la scène, ainsi qu'une autre indépendante pour gérer un panel de matériau. Cette classe ne sera pas utilisée ici.

II-2 - Implémentation

II-2-1 - Préparatif

Avant d'entrer dans les détails de chaque modification, nous allons mettre en place le nécessaire pour supporter l'undo/redo.

Voici les éléments que nous voulons présenter:

- ajout au menu "Divers" d'une action "Annuler" suivi d'une description de l'action,
- ajout au menu "Divers" d'une action "Refaire" suivi d'une description de l'action,
- annulation du changement de couleur de fond ainsi que du déplacement d'une entité,
- possibilité de fusionner des déplacements de l'entité si le temps écoulé entre 2 commandes est de moins de 5 dixièmes de seconde (afin de prendre en compte le déplacement constant à la souris, au clavier ou aux spinners) et que l'entité est la même que la précédente,
- possibilité d'afficher la pile des modifications.

La première étape consiste à créer une QUndoStack. Notre petit "éditeur" étant mono-document, nous allons la gérer à partir de la classe MainWindow. Une fois doté d'une instance de cette classe, nous allons ajouter les entrées de menus permettant de faire et défaire des actions:

```
QAction *undoAct = undoStack.createUndoAction(this, "Annuler : ");
undoAct->setShortcut(QKeySequence("Ctrl+Z"));

QAction *redoAct = undoStack.createRedoAction(this, "Refaire : ");
redoAct->setShortcut(QKeySequence("Ctrl+Shift+Z"));
```

Ces actions seront automatiquement mises à jour par QUndoStack lors de l'insertion ou du retrait d'éléments. Nous allons maintenant ajouter "l'inspecteur", ce petit widget permettra d'avoir une vue sur l'évolution de la pile d'annulation. Il sera présenté dans un QDockWidget dont l'affichage sera pilotable par une entrée dans le menu "Divers". Voici le code nécessaire à la création de ce dock:

```
// Constructeur de MainWindow:
undoView = new QUndoView(&undoStack, this);
undoView->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
undoView->setEmptyLabel("<Scene initiale>");
undoView->setMinimumWidth(150);
undoView->setMaximumWidth(150);

// [...]
```

```
// Ajout à MainWindow::createActionMenus():
QAction *undoHistoryAct = undoViewDock->toggleViewAction();
menu->addAction(undoHistoryAct);

// [...]

// Ajout à MainWindow::createDockWidget():
undoViewDock = new QDockWidget(this);
undoViewDock->setAllowedAreas(Qt::LeftDockWidgetArea | Qt::RightDockWidgetArea);

undoViewDock->setFeatures(QDockWidget::DockWidgetMovable | QDockWidget::DockWidgetFloatable | QDockWidget::DockW
undoViewDock->setWidget(undoView);
undoViewDock->setWindowTitle("Historique");
undoViewDock->setVisible(true);
addDockWidget(Qt::LeftDockWidgetArea, undoViewDock);
```

Prenez bien garde à appeler `createDockWidget` **avant** `createActionMenus` si vous vous servez du `QAction` fourni par `QDockWidget` pour piloter son affichage.

Il nous reste une dernière chose à faire avant de passer à l'implémentation des commandes que nous avons définies précédemment. Afin de pouvoir fusionner des commandes, nous allons avoir besoin d'un identifiant. Pour ce faire, j'ai choisi ici de les (enfin, "le" dans le cadre de ce tuto) stocker dans un en-tête sous la forme d'un enum:

```
#ifndef COMMANDS_H
#define COMMANDS_H

enum CommandID
{
    CID_MoveEntity
};

#endif
```

Et nous sommes enfin prêt à implémenter nos commandes !

II-2-2 - Annulation du changement de couleur de fond

Les classes dérivant de `QUndoCommand` sont en général très simples puisqu'elles ne stockent que le nécessaire pour résoudre un changement d'état, voire des informations liées à la fusion des commandes de même type. Voici l'interface utilisée pour la commande de changement de couleur de fond:

```
class ChangeBackgroundColor : public QUndoCommand
{
public:
    ChangeBackgroundColor(OgreWidget *ogreWidget, const QColor &previousColor, const
    QColor &nextColor);

    virtual void undo();
    virtual void redo();

private:
    OgreWidget * target;
    QColor prev;
    QColor next;
};
```

La méthode `id()` nous servira afin de déterminer si 2 commandes peuvent être fusionnées. Dans ce contexte, elle renvoie simplement `CID_ChangeBGColor`. Les méthodes `undo()` et `redo()` sont celles permettant à la magie d'opérer. Voici le corps d'`undo()`, à titre d'exemple (`redo()` étant similaire à l'exception de la couleur appliquée):

```
void ChangeBackgroundColor::undo()
{
```

```
target->setBackgroundColor(prev);  
target->update();  
}
```

Afin de supporter l'annulation du changement de couleur, il nous faut une méthode pour obtenir la couleur actuellement utilisée. Nous ajoutons donc une méthode `getBackgroundColor()` à `OgreWidget` qui se chargera de nous retourner un `QColor`:

```
QColor OgreWidget::getBackgroundColor() const  
{  
    if(ogreViewport)  
    {  
        Ogre::ColourValue bgColour = ogreViewport->getBackgroundColour();  
        return QColor::fromRgbF(bgColour.r, bgColour.g, bgColour.b);  
    }  
  
    return QColor();  
}
```

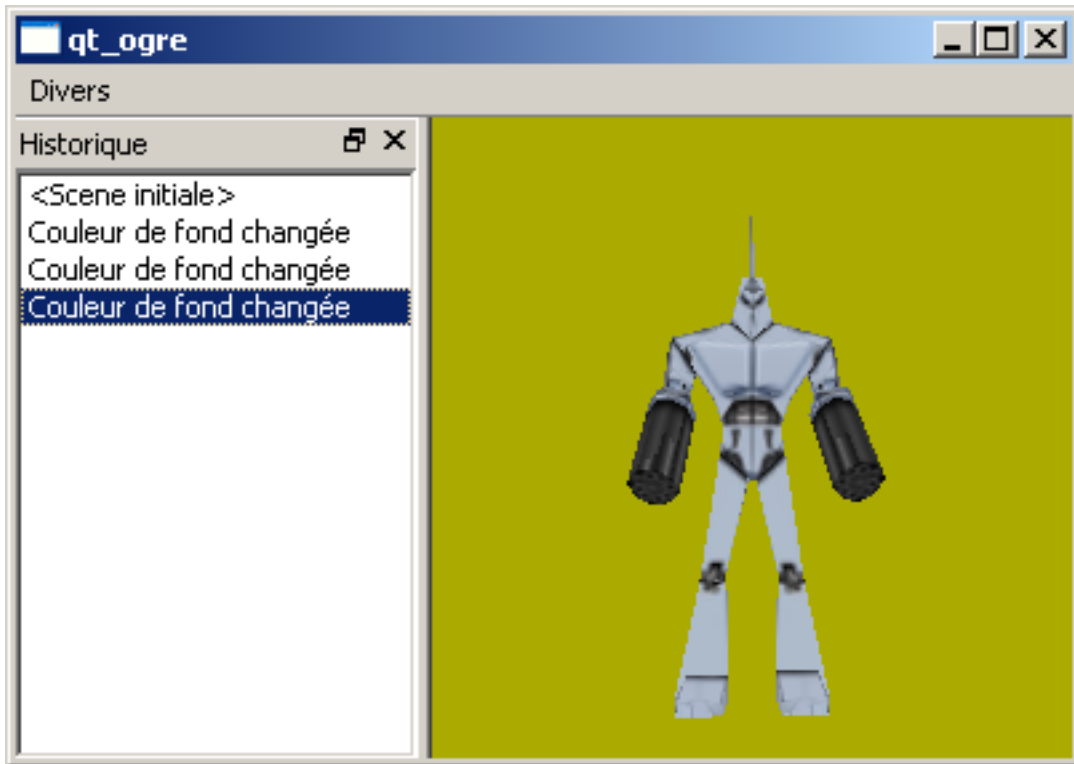
La dernière étape avant de pouvoir naviguer à travers l'historique de la scène est de créer la commande puis l'ajouter à notre `QUndoStack`. Le meilleur endroit pour le faire est `MainWindow::chooseBgColor()`, qui devient donc:

```
void MainWindow::chooseBgColor()  
{  
    QColor nextCol = QColorDialog::getColor();  
    QColor prevCol = ogreWidget->getBackgroundColor();  
    if (!nextCol.isValid())  
        return;  
  
    ChangeBackgroundColor *changeBgCommand = new ChangeBackgroundColor(ogreWidget, prevCol, nextCol);  
    undoStack.push(changeBgCommand);  
}
```

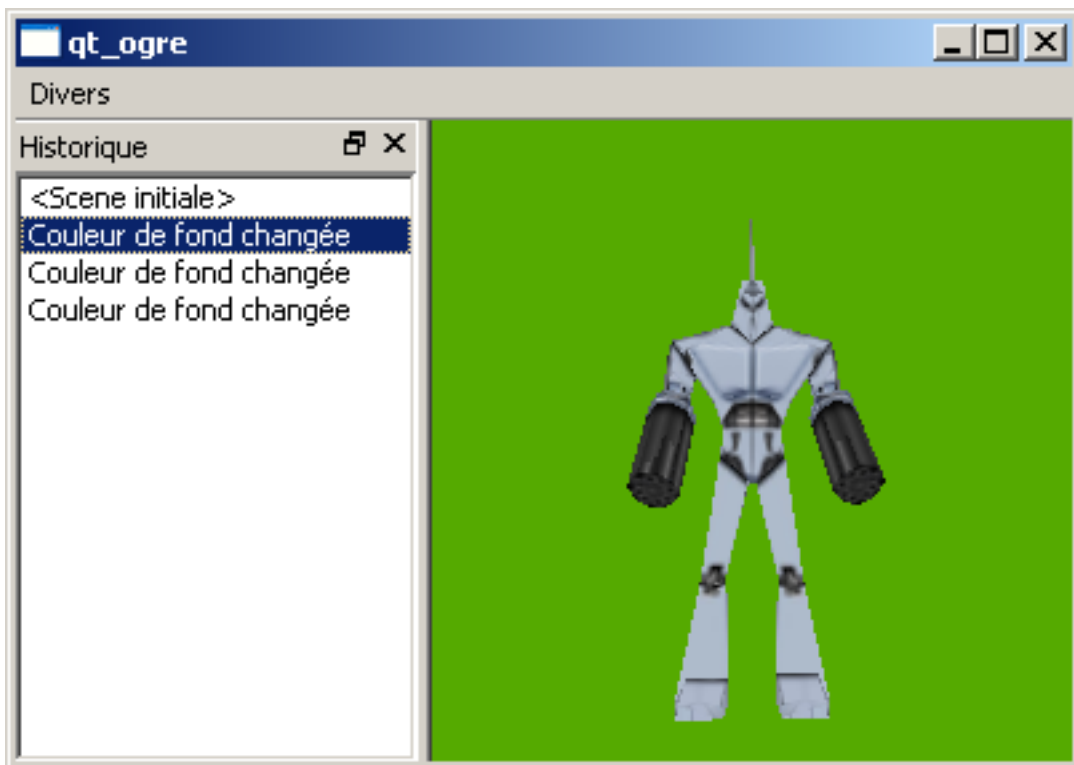
Il est très important de noter que `QUndoStack::push(QUndoCommand*)` va **exécuter l'action dès le push**. Il est dès lors inutile d'appeler `OgreWidget::setBackgroundColor()` par nous même. De même, attention aux récursions infinies si la méthode qui instancie la commande est celle la réalisant ;)

Vous pourrez constater qu'après avoir annulé une action, les actions suivantes sont supprimées si la couleur de fond est à nouveau changée. Plus que le fait de changer la couleur de fond, c'est l'acte de changer l'état de l'historique qui entraîne cet effet.

Il ne vous reste plus qu'à compiler et constater que vous pouvez annuler/refaire cette modification, et ce par notre `QUndoView` ou les actions ajoutées au menu:



Couleur de fond la plus récente (pardonnez le mauvais goût :))



Couleur de fond originellement modifiée

II-2-3 - Annulation du déplacement d'un objet

L'approche à adopter pour historiser les positions successives d'une entité est très similaire. La grosse différence est le point d'instanciation de l'action, ainsi que la fusion des commandes. Rien dans MainWindow ne nous permet de savoir que la modification de la position est effective. MySceneNode::setPosition est le point central de modification

d'un noeud de la scène (ici l'entité), c'est donc dans cette méthode que l'instanciation de la commande d'annulation/application se fera.

Vous pourriez être tenté de simplement créer et pousser la commande dans la stack à partir de cette méthode. Je dois insister sur un point si vous n'avez pas vraiment lu la doc de QUndoStack: **la méthode QUndoStack::push() applique la commande en l'ajoutant à la pile!** Nous allons devoir créer une méthode supplémentaire afin de régler la position d'un noeud sans le faire directement afin d'éviter une récursion infinie. Si c'est encore un peu flou, tout devrait s'éclaircir avec le code peu à peu.

Commençons donc par regarder l'interface de la commande qui va nous permettre ceci:

```
class SceneNodeMoved : public QUndoCommand
{
public:

    SceneNodeMoved(MySceneNode *target, const Ogre::Vector3 &previousPosition, const Ogre::Vector3 &nextPosition);

    virtual int    id() const;
    virtual bool   mergeWith(const QUndoCommand *command);
    virtual void   undo();
    virtual void   redo();

private:
    MySceneNode    *   node;
    Ogre::Vector3   prev;
    Ogre::Vector3   next;
    QTime           actionTime;
};
```

La méthode virtuelle id nous permettra de nous assurer qu'une fusion est possible entre 2 commandes. Elle se contente de retourner la valeur *CID_MoveEntity*. La dite fusion s'évalue au sein de la méthode mergeWith(). Si elle renvoie true, alors vous devez fusionner vous même les infos nécessaires; en voici le code:

```
bool SceneNodeMoved::mergeWith(const QUndoCommand *command)
{
    if(command->id() != CID_MoveEntity)
        return false;

    const SceneNodeMoved *otherCommand = dynamic_cast<const SceneNodeMoved*>(command);
    if (!otherCommand || node != otherCommand->node)
        return false;

    int msElapsed = qAbs(actionTime.msecsTo(otherCommand->actionTime));
    if(msElapsed > 500)
        return false;

    next = otherCommand->next;
    actionTime = otherCommand->actionTime;

    return true;
}
```

Comme précisé plus haut, la première chose que nous faisons est de nous assurer que la commande est bien du même type que la notre. Une fois ceci fait, les contraintes précisées en introduction de ce chapitre sont implémentées:

- le temps écoulé entre ces 2 commandes est de moins de 5 dixièmes de seconde,
- l'entité est la même que la précédente.

Si ces 2 conditions sont remplies, nous nous approprions les infos de la commande la plus récente (le paramètre). Les méthodes undo() et redo() s'occupent de positionner l'élément de cette façon:

```
void SceneNodeMoved::undo()
{
```

```
node->setPositionDirect (prev);  
}
```

setPositionDirect() est la méthode dont je parlais plus haut que nous allons ajouter à MySceneNode afin d'éviter une récursion infinie.

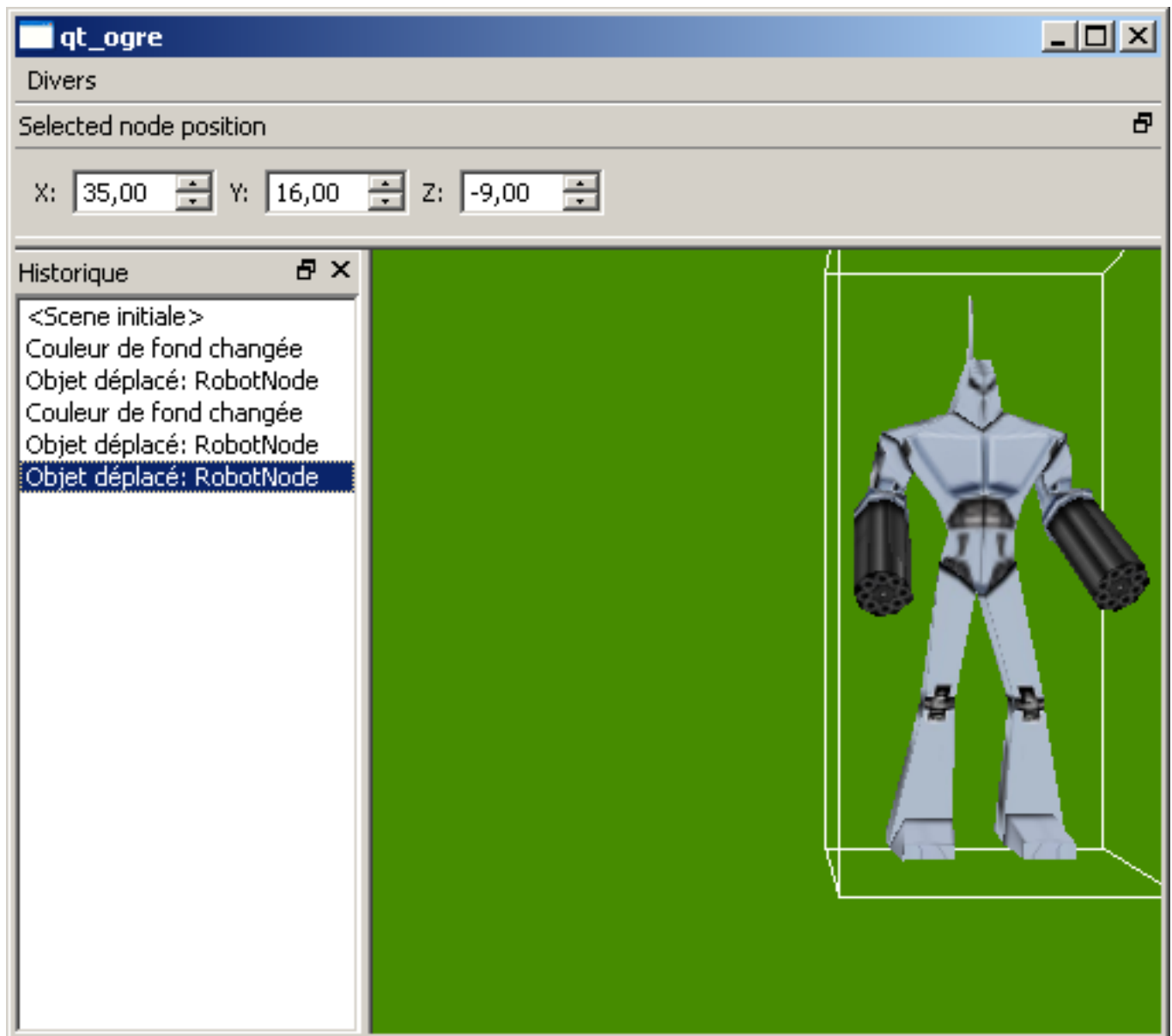
Passons maintenant à la création de cette commande. C'est OgreWidget qui s'occupe actuellement d'instancier les éléments de la scène, nous allons donc lui fournir un pointeur vers *undoStack* à partir de MainWindow, qui elle même le transférera aux objets nécessaires.

Une fois ceci fait, la dernière étape consiste à créer l'action et implémenter le déplacement de l'objet proprement dit:

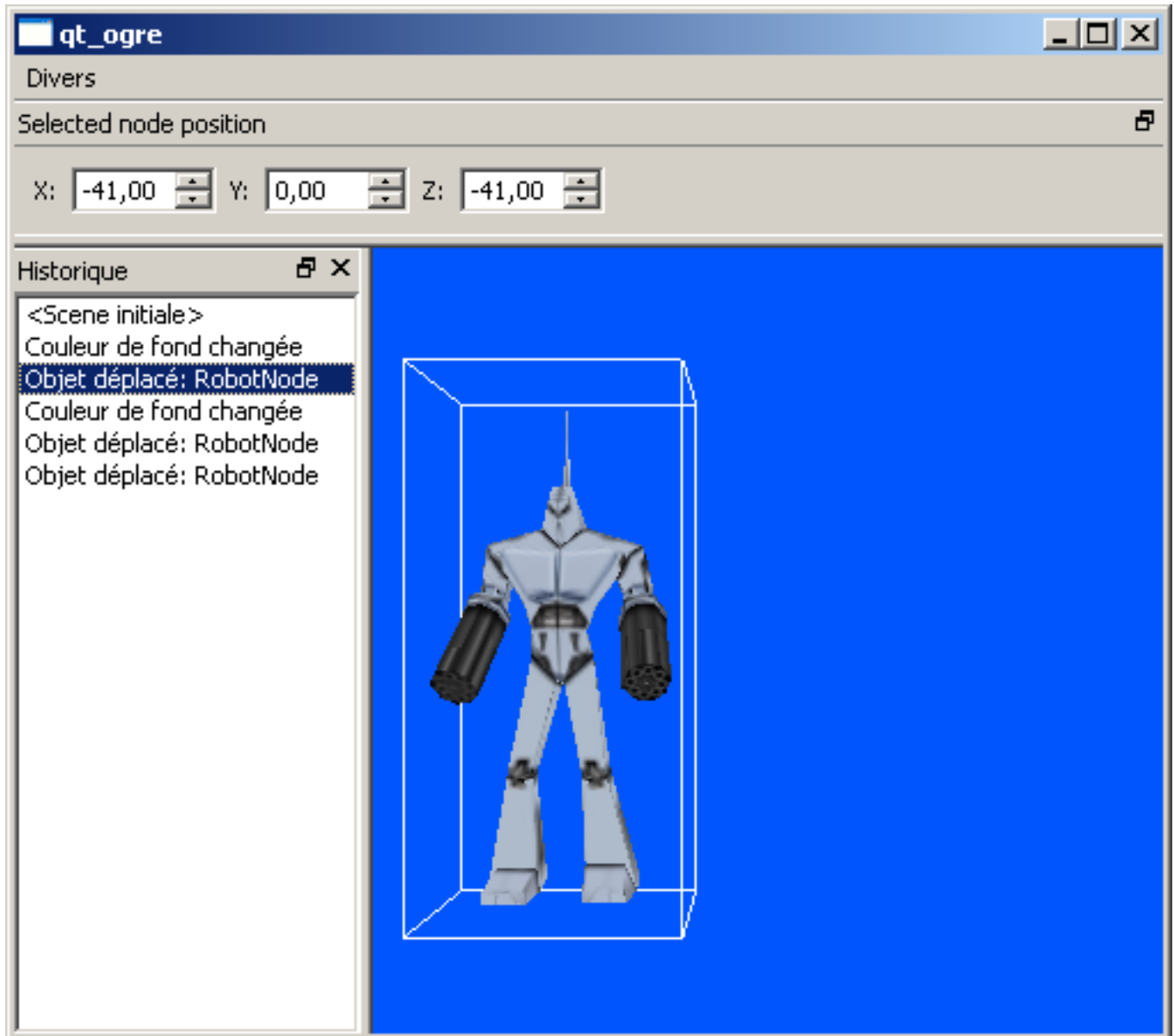
```
void MySceneNode::setPositionDirect(const Ogre::Vector3 &pos)  
{  
    ogreSceneNode->setPosition (pos);  
  
    emit positionChanged (pos);  
}  
  
void MySceneNode::setPosition(const Ogre::Vector3 &pos)  
{  
    if (undoStack)  
    {  
        Ogre::Vector3 currentPos = ogreSceneNode->getPosition();  
        SceneNodeMoved *nodeMovedCommand = new SceneNodeMoved(this, currentPos, pos);  
        undoStack->push (nodeMovedCommand);  
    }  
    else  
        setPositionDirect (pos);  
}
```

MySceneNode::setPosition() s'occupe maintenant de créer la commande et la pousser sur l'historique du document. Pour garantir le bon fonctionnement même si une QUndoStack n'est pas fournie, nous transférons l'appel à la bonne méthode. Notez bien que nous n'avons pas à toucher à la moindre des connections pour que le programme continue de tourner correctement (notre widget CoordModifier3D reste toujours synchronisé).

Si tout va bien, vous devriez obtenir ce genre de résultats:



Etat le plus récent



Quelques modifications auparavant...

Vous trouverez les sources correspondant à ce chapitre à [cette adresse \(miroir http\)](#).

III - 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;
}
```


IV - Conclusion

Et nous voici arrivé à la fin de ce tutoriel dans lequel vous aurez vu:

- une méthode permettant de décentraliser les actions sur un document du document lui-même (pensez à Paint.Net, Photoshop etc... le code affichant l'image n'implémente sans doute pas toutes les manipulations, uniquement celles rendant possible les autres interactions),
- comment utiliser le framework Undo fourni avec Qt.

J'espère que ce tutoriel aura pu répondre à d'éventuelles questions. A bientôt pour un autre voyage au pays des trolls et des ogres!

Remerciements: Merci à raptor70, Alp, dourouc05 et RideKick pour leurs relectures et commentaires :)

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