Qt sur Mac OS X avec Cocoa

Qt by Nokia

par Trenton Schulz traducteur : Thibaut Cuvelier Qt Quarterly

Date de publication : 20/03/2009

Dernière mise à jour : 01/08/2011

Depuis Qt 4.5, il est désormais possible de demander à Qt d'utiliser Cocoa sur Mac OS X. Les principaux avantages sont la capacité de lancer des applications 64bits et l'accès à des fonctionnalités uniques à Cocoa. Dans cet article, nous allons examiner le fond du changement vers Cocoa, montrer comment compiler Qt pour des besoins si spécifiques, et regarder quelles sont les nouvelles possibilités offertes pas ce portage.

Cet article est une traduction autorisée de **Qt on Mac OS X using Cocoa**, par Trenton Schulz.

I - L'article original..... 3
II - Au début fut Carbon 3
III - Compiler Qt avec Cocoa..... 3
IV - Une meilleure intégration.....4
V - Nouvelles classes.....4
V-A - QMacCocoaViewContainer.....5
V-B - QMacNativeWidget.....6
VI - Conclusion..... 6
VII - Divers.....6

Copyright © 2009 - Trenton Schulz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I - L'article original

Qt Quarterly est une revue trimestrielle électronique proposée par Nokia à destination des développeurs et utilisateurs de Qt. Vous pouvez trouver les **versions originales**.

Nokia, Qt, Qt Quarterly et leurs logos sont des marques déposées de *Nokia Corporation* en Finlande et/ou dans les autres pays. Les autres marques déposées sont détenues par leurs propriétaires respectifs.

Cet article est la traduction de l'article **Qt on Mac OS X using Cocoa** de Trenton Schulz paru dans la Qt Quarterly Issue 28.

Cet article est une traduction d'un des tutoriels écrits par **Nokia Corporation and/or its subsidiary(-ies)** inclus dans la documentation de Qt, en anglais. Les éventuels problèmes résultant d'une mauvaise traduction ne sont pas imputables à Nokia.

II - Au début fut Carbon ...

Quand Qt fut porté pour la première fois sur Macintosh, il y avait deux possibilités de librairies : Carbon, basé sur le C, avec quelques origines de l'original Macintosh Toolbox. Il implémentait les quelques nouveautés apportées par Mac OX X 10.2. Un autre avantage est qu'il était prévu pour être wrappé par une autre librairie. C'est ce que Trolltech a fait, et ils n'étaient pas les seuls : pour porter leurs applications sur Mac OS X, Adobe et Microsoft ont fait de même.

L'autre **framework** était Cocoa, une extension de l'API OPENSTEP, en directe provenance de NeXT, OS acheté par Apple. Il utilisait l'Objective-C, une extension orientée objet du C, très inspirée du SmallTalk. Alors, Carbon devait être l'outil wrappé, et Cocoa, le package complet. Il était possible de créer une application complète avec Cocoa, comme avec Qt. Comme il s'agissait principalement d'un framework prévu pour écrire des programmes, la plupart des développeurs utilisaient Cocoa directement, sans s'occuper réellement de wrapper l'API dans du C++.

Pour la majorité, peu importait le choix entre Cocoa et Carbon. Même si, comme le temps passait, les dernières technologies étaient disponibles uniquement avec l'interface Objective-C, mais elles pouvaient être intégrées dans une application basée sur Carbon. Un exemple est l'implémentation de **QSystemTrayIcon** sur Mac OS X : elle utilise NSStatusItem de Cocoa, c'était la première utilisation d'Objective-C dans Qt.

Au WWDC 2006, Apple a promis d'inclure toutes les versions 64bits de tous leurs frameworks dans Leopard (10.5). Nous avons travaillé à ce que Qt utilise les bonnes API en 64bits. Puis, il a été annoncé que Carbon ne sortira jamais en 64bits : cette plateforme est réservée à Cocoa.

Après quelques temps de recherche, nous avons commencé à créer une version de Qt qui utilise Cocoa. Après une heure de labeur intensif sur le projet et sur l'Objective-C, et sur la manière d'obliger Cocoa à travailler comme nous le désirions, le portage pour Cocoa est sorti avec Qt 4.5.

III - Compiler Qt avec Cocoa

À cause de multiples raisons de compatibilité, Qt n'utilise pas Cocoa par défaut, et ne se compile pas en 64bits. Vous pouvez facilement le contraindre à cela avec le script de configuration.

Vous pouvez configurer Qt pour la compilation en 64bits en lui passant quelques paramètres en ligne de commande. Par exemple, cette commande va compiler Qt pour toutes les plateformes supportées par Mac OS X.

```
configure -arch x86 -arch ppc -arch x86_64 -arch ppc64
```

Ainsi, Carbon sera utilisé en 32bits (i386 et PowerPC), et Cocoa en 64bits (x86_64 et PowerPC64).

Copyright © 2009 - Trenton Schulz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Bien sûr, il serait plus facile de baser tout sur Cocoa. Vous pouvez imposer ce comportement avec le paramètre -cocoa en ligne de commande. Cependant, si vous utilisez Cocoa sur une plateforme 32bits, vous devrez respecter ces quelques conditions.

- Leopard. Il y a des objets et des méthodes qui ne sont disponibles que sous cet OS.
- Pas de Qt3Support. Pour un portage Cocoa efficace, nous avons décidé qu'il fallait laisser tomber ce support. Votre application devra donc directement utiliser les API Qt 4.
- Qt en tant que framework. Cocoa requiert le chargement de fichiers NIB (l'équivalent des fichiers .ui) pour un support correct de la barre de menu. La meilleure solution est de mettre le fichier NIB dans le framework et de ne pas le copier dans chaque application Qt.

Historiquement, la principale raison de lier statiquement à Qt était de rendre le déploiement plus facile. Nous avons répondu à cet objectif dans Qt 4.5 via l'outil macdeployqt.

IV - Une meilleure intégration

Puisque nous portions vers un nouveau framework, nous avons pris le temps de résoudre quelques problèmes lors de l'utilisation de Qt sous Mac OS X. Entre autres, cela inclut une meilleure intégration avec les boucles natives et une utilisation plus simple des feuilles dans les applications. Ces changements restent valables pour Carbon, au même titre que Cocoa.

La raison principale pour une meilleure intégration dans les boucles natives de fonctionnement était de rendre plus facile l'emploi de plugins, comme les Audio Units. Qt a son propre jeu d'événements internes qu'il envoie aux widgets pour leur signifier des changements d'état. Ces événements sont mis dans une queue, puis expédiés par un appel à **QCoreApplication::sendPostedEvents**. Normalement, ceci arrive en tant que membre de l'exécution de l'expéditeur d'événements quand il s'en occupe. Cependant, quand Qt n'est pas le contrôleur principal d'une application (dans le cas du développement de plugins), parfois, l'expéditeur n'est pas lancé, et les événements ne sont pas envoyés. Ceci n'est pas forcément un problème, mais, plus tard, il peut rugir et montrer sa tête lors d'une mise à jour d'un écran, qui n'arrive jamais.

Dans Qt 4.5, nous avons inclus ceci dans la boucle principale. Ainsi, les événements Qt sont envoyés en même temps que les autres (comme les sockets ou les mouvements de la souris). Ainsi, peu importe le principal contrôleur : Qt, Carbon, Cocoa, les trois permettent l'envoi des messages de Qt, promptement.

Il a toujours été possible d'utiliser les feuilles avec Qt. Cependant, ils requéraient une maîtrise pour un usage correct. Même en interne, Qt ne les utilise pas pour les dialogues. Même s'il est rapide de les utiliser dès que l'on les maîtrise, le processus entier pourrait être simplifié.

La solution adoptée a été l'ajout d'un slot à **QDialog**, **open()**. Quand on l'appelle, le dialogue sera lancé comme une fenêtre modale sur le parent. Sur Mac OS X, cela résulte en une feuille. Nous avons aussi étendu les dialogues internes de Qt, pour les adapter à ce mécanisme. Vous pouvez désormais instancier un **QFileDialog** et avoir un dialogue natif, qui fonctionne aussi bien avec une feuille qu'avec un dialogue normal, dépendant de quelle méthode vous appelez.

La nouvelle version en entier est intéressante, et mérite probablement son propre article. Un effet de côté du nouveau dialogue et de la meilleure intégration avec la boucle des événements est qu'il est possible d'avoir des polices natives sous Mac OS X.

V - Nouvelles classes

À côté de cela, nous avons aussi décidé de rendre plus facile d'utiliser des vues basées sur Cocoa dans la hiérarchie Qt, et de mettre des widgets basés sur Qt dans des hiérarchies sans Qt. Ceci est accompli avec les classes **QMacCocoaViewContainer** et **QMacNativeWidget**. Ces classes ne sont pas idéales pour une application multi-plateformes, mais elles sont recommandées par le docteur pour les projets-plugins d'autres applications, ou lors de l'ajout de technologies spécifiques aux Mac dans une application prévue pour Mac OS X.

Copyright © 2009 - Trenton Schulz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

V-A - QMacCocoaViewContainer

La première classe fait la même chose que dans le HIViews du Qt Quarterly 20, avec une seule exception : cela fonctionne avec Cocoa. Il profite aussi de la compatibilité avec Carbon, même si ce support requiert Leopard.

Même si vous pouvez certainement utiliser cette classe toute seule, la manière la plus répandue est de la dériver. Ainsi, vous pouvez plus facilement translater les événements entre Qt et l'Objective-C de votre code.

Ce code d'exemple montre ce design pattern. Il commence par prendre la vue sur l'animation principale de l'exemple CocoaSlides **disponible chez Apple**, en le mettant dans la hiérarchie de Qt, et en connectant les signaux des contrôles Qt aux slots de QCocoaViewContainer. Voici le prototype de la classe qui fait la majorité du travail, AssetController.

```
class AssetController
 : public QMacCocoaViewContainer
{
 Q_OBJECT
public:
 enum AnimationPath { Circular, Loop,
       Scatter, Wrapped };
 AssetController(QWidget *parent);
 ~AssetController();
public slots:
 void browseForDefaultSlides();
 void browseForSlides(const QString &patch);
 void performAnimation(AnimationPath path);
 void setDuration(qreal newDuration);
 void setCycleInterval(qreal interval);
 void performDefaultAnimation();
 void setSlidesHaveShadows(bool shadow);
 void setUsesQuartzCompositionBackground(bool);
private:
 AssetCollection *assetCollection;
 AssetCollectionView *assetCollectionView;
};
```

Cette classe est une simple dérivée de **QMacCocoaViewContainer** qui contient des pointeurs, et vers le modèle (AssetCollection), et vers la vue (AssetCollectionView). Ceci est standard, vu que Cocoa est basé sur le pattern **MVC**. Les slots déclarés sont simplement des trous vers les méthodes de la vue ou du contrôleur. Cependant, vu que ce sont des slots Qt, ils peuvent être connectés à des signaux Qt.

```
AssetController::AssetController(QWidget *parent)
 : QMacCocoaViewContainer(0, parent)
{
 assetCollectionView = [[AssetCollectionView alloc]
 initWithFrame:NSMakeRect(0, 0, 100, 100)];
 setCocoaView(assetCollectionView);
 /*
 ...
 */
}
```

Le constructeur va aussi vite. Nous créons notre propre vue Cocoa sur le contenu, nous ne passons donc pas initialement de vue à notre classe de base, mais, à la place, nous allouons et initialisons notre vue et la passons à setCocoaView() pour installer le conteneur. Il vaut la peine de remarquer que setCocoaView() va retenir une référence à la vue qui lui est passée et sortir si une autre vue est définie ou s'il est détruit. Depuis que la dérivée AssetController doit aussi avoir accès au pointeur, tout juste alloué, il n'y donne pas accès.

```
void AssetController::browseForSlides(const QString &qpath)
{
 [assetCollectionView setWantsLayer:YES];
 NSString *path = [NSString stringWithCharacters:(
  const unichar *)qpath.data() length:qpath.length()];
```


Copyright © 2009 - Trenton Schulz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
 assetCollection = [[AssetCollection alloc]
  initWithRootURL:[NSURL fileURLWithPath:path]];
 [assetCollectionView setAssetCollection:assetCollection];
 [assetCollection startRefresh];
}
void AssetController::performAnimation(AnimationPath path)
{
 [assetCollectionView setSubviewsLayoutType:path];
}
void AssetController::setSlidesHaveShadows(bool shadow)
{
 [assetCollectionView setSlidesHaveShadows:shadow];
}
```

Les méthodes suivantes montrent à quel point il est simple de juste passer l'information aux classes de Cocoa. Par exemple, lors de la navigation, nous savons que **QString** et NSString contiennent des caractères Unicode sur 16bits. Il est donc facile de passer de l'un à l'autre.

Bien sûr, il est possible d'utiliser Core Animation directement sur nos widgets. Il suffit de caster le retour de winId() sur un pointeur sur NSView, et vous pourrez utiliser toutes les API Cocoa sur le widget, dont les transitions complexes à partir des filtres Core Animation ou Core Image.

V-B - QMacNativeWidget

Cette classe fonctionne sur une différence de conception entre Qt et les frameworks de Mac OS X.Qt ne fait pas de distinction entre les widgets de niveau maximal (comme les fenêtres), et les vues et contrôles dans cette fenêtre, alors qu'Apple la fait. Tant que l'on n'utilise que Qt, il n'y a pas de problème. La situation devient réellement problématique lorsqu'il faut intégrer des widgets Qt dans des applications non-Qt.

Cette classe fonctionne en prétendant qu'il s'agit d'une fenêtre pour tous les widgets Qt, en même temps que prétendre être une vue normale à toutes les méthodes Carbon et Cocoa. Ceci signifie que Qt va fonctionner correctement, et qu'il n'y a plus besoin de wrapper toute la hiérarchie des fenêtres Qt.

Après avoir instancié cette classe, on l'utilise comme n'importe quel **QWidget**, en créant des widgets enfants. Ceci requiert du développeur un peu de connaissance de la manière dont Carbon ou Cocoa fonctionnent, vu qu'il va travailler directement avec eux.

VI - Conclusion

Nous avons vu les raisons de la création d'un portage Cocoa sur Mac OS X et les avantages qui sont parvenus via ces changements. Nous avons aussi vu de nouveaux moyens d'utiliser les dialogues et quelques classes spécifiques qui peuvent se révéler utiles lors de la combinaison avec d'autres applications et librairies.

VII - Divers

Au nom de toute l'équipe Qt, j'aimerais adresser le plus grand remerciement à Nokia pour nous avoir autorisé la traduction de cet article !

J'aimerais adresser ici un remerciement tout particulier à **ram-0000** pour sa relecture attentive !

Copyright © 2009 - Trenton Schulz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.