Qt Graphics et performance - Le moteur de rendu OpenGL

Qt by Nokia

Par Tom Cooksey - traducteur : Guillaume Belz - Qt Labs - johnlamericain

Date de publication : 24 octobre 2010

Dernière mise à jour : 16 mars 2011

OpenGL est une API qui permet de bénéficier l'accélération matérielle (puce graphique) pour réaliser les rendus 2D et 3D. Le moteur de rendu OpenGL utilise cette API pour améliorer les performances des rendus dans Qt tout en offrant les mêmes fonctionnalités que le moteur Raster.

Cet article décrit en détail le fonctionnement de ce moteur de rendu, quelles sont les fonctionnalités qui sont coûteuses et comment optimiser son utilisation.

Cet article est une traduction autorisée de **Qt Graphics and Performance # OpenGL** de **TomCooksey**.

N'hésitez pas à commenter cet article !
Commentez

I - L'article original..... 3
II - Introduction..... 3
II-a - Pourquoi OpenGL ?..... 3
II-b - Histoire..... 3
II-c - Qu'est-ce que OpenGL (ES) 2 peut fournir ?..... 4
II-d - De quoi a besoin QPainter ?..... 4
III - Fonctionnement du moteur..... 5
III-a - Rendu de primitives..... 5
III-b - Rendu à l'aide de la technique du stencil..... 6
III-c - Remplissage de primitives..... 8
III-d - Gestion des textures..... 9
III-e - Anti-aliasing..... 9
III-f - Découpage..... 9
IV - Recommandations..... 10
IV-a - Rendus entrecroisés..... 10
IV-b - Mélanger QPainter avec de l'OpenGL native..... 10
IV-c - QGLWidget vs le système graphique OpenGL..... 10
IV-d - Conseils pour les performances (fps)..... 11
IX - Conclusion..... 11

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

I - L'article original

Le site **Qt Labs** permet aux développeurs de Qt de présenter les projets, plus ou moins avancés, sur lesquels ils travaillent.

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 Graphics and Performance # OpenGL** de **TomCooksey** paru dans Qt Labs.

II - Introduction

Voici l'épisode suivant de la série de blogs sur les performances graphiques. Nous allons commencer par examiner certains détails sur la façon dont OpenGL et QPainter travaillent. Nous allons ensuite aborder la façon dont les deux sont associés dans le moteur de rendu OpenGL 2 et nous finirons par quelques conseils sur la façon de tirer le meilleur parti de ce moteur. Amusez-vous bien !

II-a - Pourquoi OpenGL ?

Avant de plonger dans le moteur de rendu OpenGL, je veux m'assurer que nous nous comprenons bien sûr les motivations d'avoir un moteur de rendu OpenGL 2.0. J'en ai déjà parlé dans mon article sur l'accélération matérielle, mais nous avons encore souvent des questions du genre "Pourquoi ne pas mettre en place un moteur de rendu Direct2D ?".

Tout le monde sait que OpenGL signifie des graphiques rapides, vrai ? Eh bien, c'est en réalité une idée fausse. Ce qui rend les graphismes rapides, c'est d'avoir du matériel dédié au rendu d'images, appelé GPU (Graphics Processing Unit). OpenGL 2.x est une bibliothèque logicielle qui utilise souvent (mais pas toujours) une classe particulière de GPU pour aider à réaliser les opérations de dessin.

Note : OpenGL 1.x utilisait une autre classe de GPU.

Un GPU moderne programmable (par exemple une carte nVidia GTX 295) peut généralement être programmés via les API OpenGL, Direct3D et OpenCL. La seule différence est alors que Direct3D est disponible uniquement sur la plate-forme Windows et OpenCL n'est pas pris en charge universellement.

Donc, la raison pour laquelle nous investissons notre temps et nos efforts dans OpenGL plutôt que dans Direct3D ou OpenCL, c'est que OpenGL 2.0 est suffisant pour nous donner accès à toutes les fonctionnalités des GPU que nous avons souhaité utiliser actuellement. OpenGL est également disponible sur plusieurs plates-formes, surtout si vous vous limitez au sous-ensemble de fonctionnalités d'OpenGL ES. Nous recherchons également à nous restreindre encore plus en n'utilisant que les API de base d'OpenGL 3.2.

Cela pourrait changer à l'avenir si nous voyons une nouvelle classe de GPU, comme ceux conçus pour les graphismes vectoriels 2D qui ne peuvent pas être correctement utilisés avec OpenGL 2.0 (voir OpenVG) ou si nous voulons commencer à utiliser les fonctionnalités des GPU qui ne sont pas disponibles avec OpenGL (ES) 2.0. Cela dit, OpenGL est très efficace pour prendre en charge les nouvelles caractéristiques des GPU grâce aux extensions.

II-b - Histoire

Qt a eu un moteur de rendu OpenGL depuis les premiers jours de Qt 4.0. Ce moteur a été conçu pour le matériel à pipeline fixe disponibles à l'époque. Comme le temps passait et que les fabricants ont ajouté de nouveaux éléments de traitement à leur GPU, le moteur de rendu OpenGL a été adapté pour utiliser ces fonctionnalités grâce à des

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

extensions OpenGL. Au cours des quatre dernières années, beaucoup de gens ont développé sur le moteur et ont ajouté la prise en charge de fonctionnalités comme les ARB fragment program et ont adapté le moteur pour fonctionner avec OpenGL ES 1.1. Le moteur est assez stable et a beaucoup des solutions de repli (ou de "original code-paths", selon la façon dont vous les regardez) pour le vieux matériel qui n'ont pas les extensions OpenGL que le moteur utilise. Mais, fondamentalement, il s'agit d'un moteur OpenGL 1.x.

Au début de 2008, à l'époque du projet Falcon (le projet Falcon était un projet interne qui a commencé avec Qt 4.5 et qui s'est concentré sur la performance du rendu et l'architecture), il est devenu de plus en plus clair que Qt devait supporter l'accélération matérielle en utilisant l'API OpenGL ES 2.0 qui commençait à apparaître sur systèmes embarqués sur des puces telles que l'OMAP3. Il y avait deux options disponibles : développer encore plus le moteur de rendu OpenGL existant ou développer un nouveau moteur de rendu à partir de zéro. Lorsque l'on regarde le moteur existant, il y avait un problème majeur - bien qu'il supporte les programmes de fragments, il était fortement dépendant des traitements de vertex à fonctions fixes. Une autre considération était que le projet Falcon venait tout juste d'être abandonné et l'avenir de l'API QPaintEngine était incertain. Ces deux facteurs ont abouti à l'écriture d'un nouveau moteur de rendu à partir de zéro basé sur OpenGL ES 2.0. Ce nouveau moteur a un net avantage sur le moteur existant : tout ce que je voulais utiliser dans OpenGL était intégré dans l'API de base d'OpenGL ES 2.0. Cela signifiait que je n'avais pas besoin d'ajouter des solutions de repli en cas d'absence de fonctionnalité, conduisant à code beaucoup plus propre et plus léger.

Un autre point sur OpenGL ES 2.0 est qu'il n'a pas beaucoup de fonctionnalités basées sur les fonctions fixes, forçant à écrire des programmes de shader. Bien que gênant à l'époque, c'est apparemment la meilleure façon de faire les choses, même sur les GPU pour ordinateur de bureau. Ce point est important, car il est vite devenu évident que, même si le moteur a été conçu pour OpenGL ES 2, il peut non seulement travailler sur la version Desktop d'OpenGL 2.0, mais il peut aussi utiliser cette API de façon plus adaptée aux GPU programmables modernes. Ainsi, dans Qt 4.6, le nouveau moteur est utilisé par défaut sur les deux systèmes, OpenGL ES 2 et les versions Desktop qui prennent en charge OpenGL 2.0.

II-c - Qu'est-ce que OpenGL (ES) 2 peut fournir ?

Comme je l'ai déjà mentionné, OpenGL ES 2.0 est une API très légère pour les GPU programmables. Le terme "programmable" est fondamental pour cet API. Cela signifie que vous pouvez écrire de petits programmes, appelés shaders, puis de demander à OpenGL de les compiler et de les exécuter sur le GPU pour traiter les données que vous lui donnez. Il existe deux types de shaders : le premier travaille avec les positions (sommets) et l'autre avec les pixels (fragments). Ils sont appelés respectivement vertex shader et fragment shader. L'idée est de dire à OpenGL que vous voulez dessiner des triangles et le vertex shader est exécuté pour déterminer la position de chaque triangle. Ensuite, le GPU convertit chaque triangle en un ensemble de pixels et le fragment shader est exécuté pour déterminer la couleur de chacun de ces pixels. L'API offre différentes façons de transmettre les données depuis le CPU vers le GPU (sous forme de textures, de listes de triangles ou encore individuellement sous forme de nombres réels) et les moyens de transmettre des données entre le vertex shader et le fragment shader. Fondamentalement, il n'y a rien de plus. Toute la complexité réside dans les shaders que vous donnez à exécuter au GPU.

II-d - De quoi a besoin QPainter ?

Le reste de cet article suppose que vous êtes familier avec l'API QPainter (sinon, allez relire la documentation de QPainter). De plus, ça pourrait être une bonne idée de lire le post de Gunnar sur le fonctionnement du moteur de rendu Raster.

Ainsi, l'API de QPainter fournit plus que de simples triangles. C'est donc le travail du moteur de rendu OpenGL de convertir l'ensemble de l'API de QPainter en "un ensemble de triangles". Pour comprendre un peu mieux ce qu'il doit faire, vous devez diviser QPainter en différents morceaux, chacun devant s'accorder mieux avec OpenGL. Un bon exemple de ceci est drawRect(). En ce qui concerne QPainter, il s'agit d'une primitive unique, mais, en termes de moteur OpenGL, c'est en réalité deux primitives : un rectangle (l'intérieur à remplir) et la ligne (peut-être assez complexe) qui fait le tour à l'extérieur (le contour). Le moteur de rendu OpenGL tente de maintenir une séparation assez nette entre la forme de quelque chose qui est dessiné et son remplissage. Donc, voici la liste des primitives (formes) que QPainter demande au moteur de rendu de dessiner :

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

- les primitives simples (rectangles, polygones convexes, ellipses, etc.) ;
- les textes ;
- les pixmaps ;
- les contours ;
- les tracés vectoriels complexes (QPainterPath).

En plus de cela, il y a différents mode de remplissage fournies par QBrush que nous pouvons utiliser dans nos primitives :

- une couleur unie ;
- des dégradés linéaires ;
- des dégradés radiaux ;
- des dégradés coniques ;
- des images de motifs ;
- des textures.

Non seulement nous avons différents types de remplissage, mais nous devons également supporter complètement des matrices de transformation 3x3 pour les pinceaux. Cela vous permet de dessiner un rectangle et de l'utiliser comme une sorte de gabarit (par exemple) pour appliquer une texture transformée en perspective.

Enfin, QPainter exige également que le moteur de rendu permette le découpage, les modes de composition différents et le support de la pile d'états (QPainter::save() et QPainter::restore()).

III - Fonctionnement du moteur

III-a - Rendu de primitives

Les primitives simples : pour dessiner des primitives convexes telles que des rectangles arrondis, nous devons simplement créer des triangles OpenGL de type GL_TRIANGLE_FAN et les dessiner à l'aide de glDrawArrays.

Les textes : pour un texte avec une grande taille de police, nous le convertissons en un chemin complexe et nous le dessinons comme telle. Toutefois, pour des tailles de police plus petites, nous pixellisons les glyphes de polices individuellement et les envoyons comme une texture (texture 8 bits pour les bitmaps et les caractères avec anti-aliasing et 24 bits RGB pour les caractères avec anti-aliasing et des sous-pixels). Cette texture de glyphe est utilisée comme un masque pour le moteur de pixels du pipeline (voir ci-dessous). Donc, en termes de primitives, le texte est en fait rendu comme un ensemble de rectangles : un rectangle pour chaque glyphe. Lors du rendu avec des caractères avec sous-pixels et anti-aliasing, il est possible que le moteur ait besoin de faire deux passes (si le pinceau n'est pas une couleur unie). C'est parce que le moteur utilise une astuce et met la couleur de la brosse dans le glBlendColor et le masque RVB dans le pixel shader. Il est alors possible de définir une fonction dans glBlendFunc qui combine les deux et réalise le mélange par sous-pixel. Si vous définissez une brosse plus complexe, le moteur doit faire deux passes : la première applique le masque à la destination, puis la seconde passe applique le pinceau, avec la fonction glBlendFunc pour donner le résultat correct.

Les pixmaps : une pixmap est en fait seulement un rectangle.

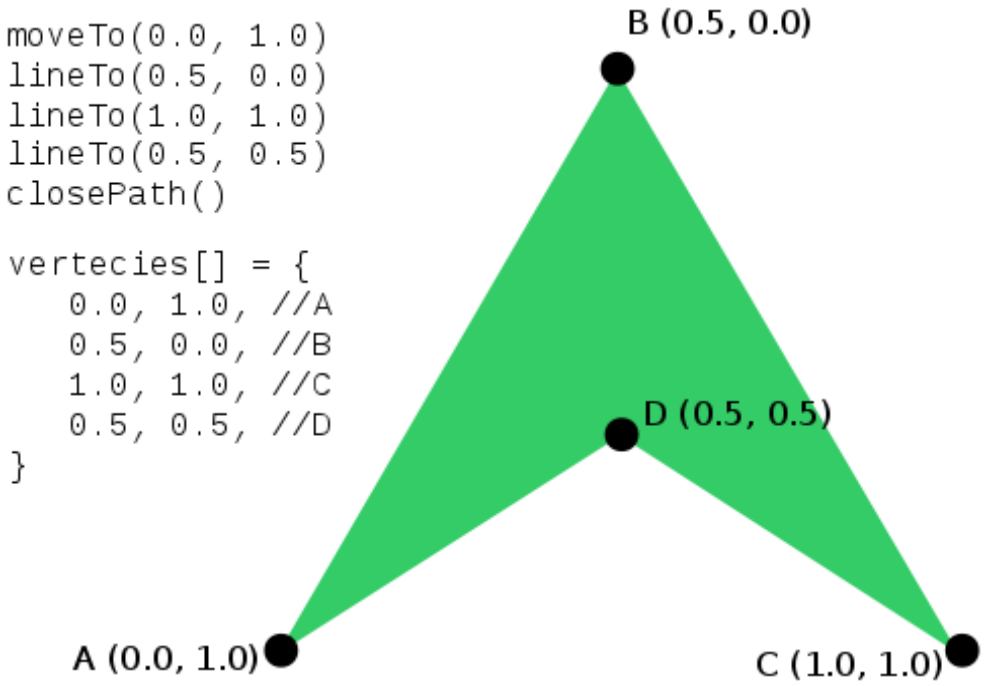
Les contours : les contours peuvent être très complexes - il suffit de jeter un œil à la démo **pathstoke** ! Cependant, même le motif en pointillés le plus complexe **avec des angles** et des **extrémités arrondis** peuvent être transformés en une série de triangle OpenGL relativement facilement. Cela est réalisé par la classe QTriangulatingStroker.

Les tracés vectoriels complexes : c'est là que les choses se compliquent. Les **QPainterPath** peuvent avoir beaucoup de choses qui brisent l'algorithme "convertir les lineTo, les moveTo et les curveTo en vertices et les dessiner sous forme de triangles"...

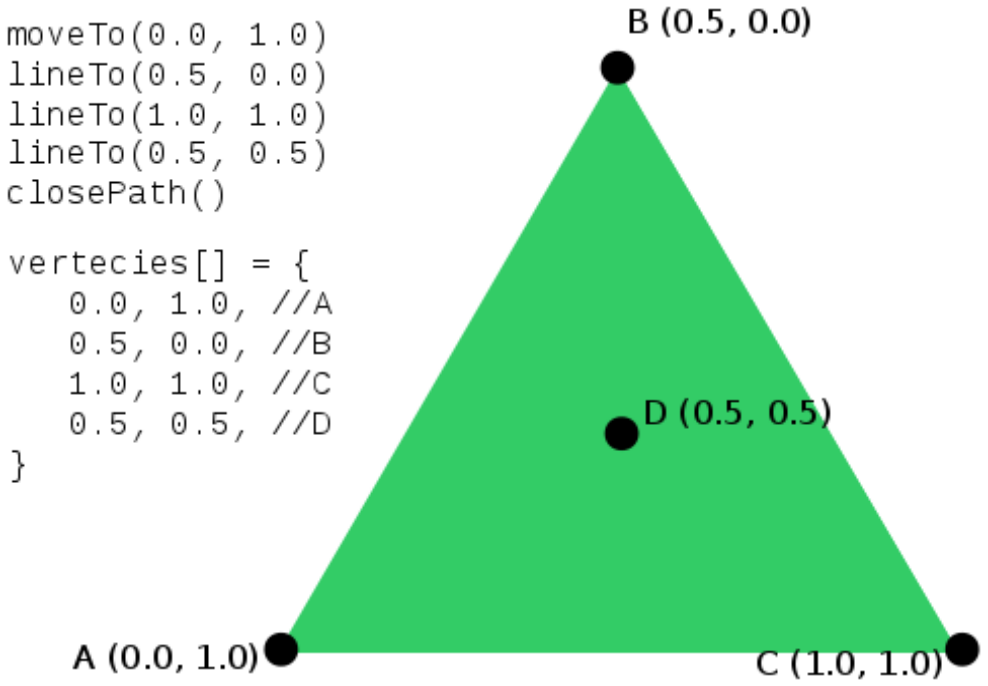
Copyright ® 2010 Gunnar. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

III-b - Rendu à l'aide de la technique du stencil

Prenez le chemin suivant à titre d'exemple :



Ici, nous avons un chemin apparemment trivial avec seulement quatre points. Pour le dessiner avec OpenGL, vous pourriez convertir les points du chemin en vertices et le dessiner en tant que suite de triangles (GL_TRIANGLE_FAN), ce qui donne deux triangles : le triangle 1 (ABC) et le triangle 2 (ACD). Le problème est que le résultat correspond à un triangle plein, pas au chemin que nous voulions :



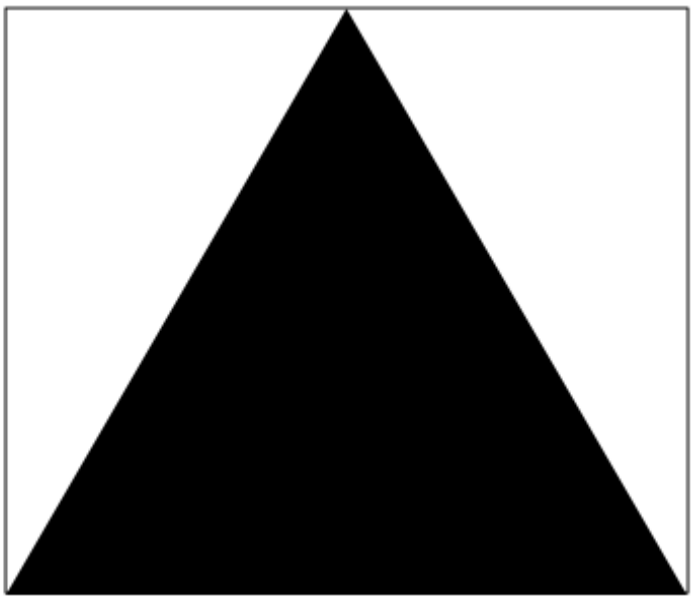
Donc, pour surmonter cette difficulté, nous passons à une méthode de rendu en deux passages, qui utilise le stencil buffer comme un bloc-notes temporaire. Pour commencer, nous effaçons le stencil buffer (représenté en blanc) :

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
vertecies[] = {
   0.0, 1.0, //A
   0.5, 0.0, //B
   1.0, 1.0, //C
   0.5, 0.5, //D
}
```

Après, nous mettons le mode d'opération du stencil en inversion, ce qui signifie que, au lieu de mettre la valeur du stencil à '1' lorsqu'un triangle touche un pixel, on inverse à la place la valeur existante. Donc 0 devient '1' et '1' devient 0. Donc, nous dessinons d'abord le premier triangle (ABC). Comme tous les pixels sont actuellement à 0, chaque pixel touché par le triangle passent à 1 (représenté en noir) :

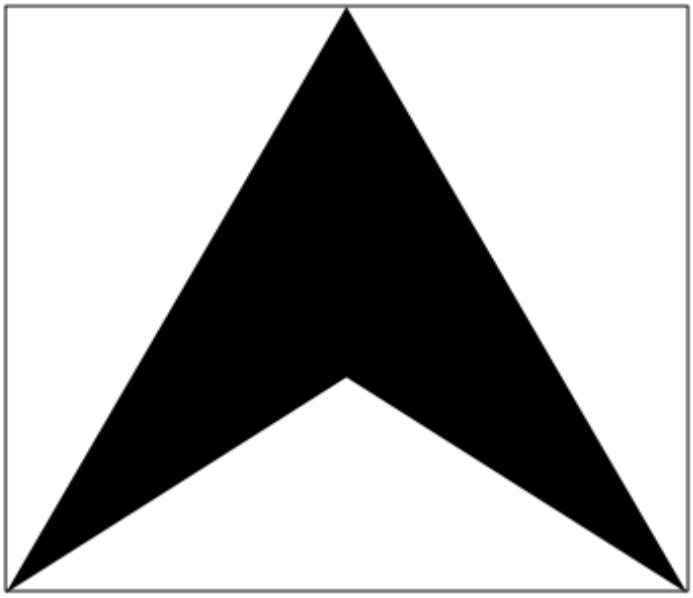
```
vertecies[] = {
   0.0, 1.0, //A
   0.5, 0.0, //B
   1.0, 1.0, //C
   0.5, 0.5, //D
}
```



Ensuite, nous dessinons le second triangle (ACD). Note : nous inversons la valeur du stencil, donc les pixels noirs touchés par le deuxième triangle passent en blancs et les pixels blancs deviennent noirs :

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
vertecies[] = {
    0.0, 1.0, //A
    0.5, 0.0, //B
    1.0, 1.0, //C
    0.5, 0.5, //D
}
```



À ce moment, le stencil buffer contient la silhouette de notre chemin. Tout ce que nous faisons maintenant est de dessiner un rectangle dans la fenêtre de destination, mais en activant le test du stencil.

En plus de la technique du stencil, nous ajoutons également un support expérimental pour la triangulation de QPainterPath et la mise en cache de la triangulation. Bien que ce soit plus lent pour les chemins qui changent souvent ou qui sont réduit ou agrandit, les chemins qui sont relativement statiques peuvent être triangulés une fois et être redessinés à plusieurs reprises sans avoir à re-trianguler.

III-c - Remplissage de primitives

Maintenant, nous savons comment les différentes opérations de QPainter peuvent être transformées en primitives OpenGL, mais nous ne savons toujours pas comment les remplir. Comme déjà mentionné, la couleur d'un pixel est déterminée par le fragment shader. Nous avons donc beaucoup de fragment shader pour les différents types de remplissage. Cependant, nous devons également supporter le rendu de texte avec remplissage arbitraire (QPainter vous permet de remplir du texte avec un dégradé radial transformé en perspective). Dans l'avenir, nous voulons aussi supporter les modes de composition qu'OpenGL ne fournit pas. Nous avons également constaté qu'il existe des situations pour lesquelles nous pouvons simplifier les shaders (et donc d'améliorer les performances). Le résultat est que Qt a besoin de beaucoup de shaders différents. Au dernier décompte, nous aurions besoin de plus de 1000 shaders différents pour couvrir toutes les situations. C'est beaucoup de code GLSL à maintenir et à tester, beaucoup plus que les ressources dont nous disposons. À la place, nous avons réparti les shaders en différentes "étapes" interchangeables. Ceci est réalisé en donnant à chaque étape sa propre fonction GLSL. À titre d'exemple, prenons le rendu d'un texte standard, avec anti-aliasing sans sous-pixels et un gradient radial transformé. Notez que ceci est seulement un exemple pour montrer comment le moteur fonctionne et vous ne devriez pas le faire dans des situations de performances critiques.

Nous dessinons les gradients en pré-calculant une texture de hauteur un pixel (comme une texture 1D) sur le CPU que nous envoyons ensuite dans le fragment shader. De plus, nous calculons les coordonnées de la texture dans le vertex shader et nous les passons dans le fragment shader en tant que variable. Nous utilisons cette approche parce que c'est une bonne idée de faire autant de travail que possible dans le vertex shader, plutôt que dans le fragment shader, celui-ci étant appelé beaucoup moins souvent.

Comme déjà mentionné, nous avons réalisé le rendu du texte avec anti-aliasing (mais sans sous-pixels) en utilisant une texture 8 bits comme masque. Nous multiplions ensuite la couleur du pixel par une valeur issue de ce masque. Donc, si nous sommes sur le bord d'un caractère dont la valeur alpha est inférieure à un, nous ajustons la valeur du canal alpha de la source SrcPixel par cette valeur alpha du masque (en fait, nous devons également ajuster les valeurs RGB du fait que nous utilisons le format de pixel avec canal alpha pré-multiplié en interne).

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

S'il y avait un mode de composition non-standard, nous aimerions passer ensuite le pixel après masquage à une autre étape qui le mélangerait avec le fond (mais ce n'est pas encore implémenté !)

Ainsi, comme vous pouvez le voir dans le fragment shader, il y a trois étapes différentes. La première étape (srcPixel) détermine la couleur du pinceau pour le pixel. L'étape suivante (applyMask) module le pixel par un masque pour réaliser le rendu du texte anti-aliasé. La dernière étape (compose) fusionne les pixels avec l'arrière-plan. Nous avons également une technique similaire, en plusieurs étapes, pour les vertex shader. Toute cette complexité est correctement abstraite par le QGLEngineShaderManager. Le moteur de rendu explique au gestionnaire de shader ce qu'il veut faire et le gestionnaire de shader sélectionne un ensemble approprié de shaders. Un dernier mot sur ce point : tandis que la version Desktop d'OpenGL 2 supporte la liaison de plusieurs fragment shader dans un seul programme, OpenGL ES 2.0 ne le permet pas. Cela signifie que nous utilisons en réalité les différentes étapes en les ajoutant dans une seule chaîne de caractères pour former le code GLSL que nous passons à OpenGL. Cela donne également à l'implémentation d'OpenGL de meilleures chances d'optimiser les différentes étapes (sans cela, la performance serait à craindre).

III-d - Gestion des textures

Le moteur de rendu OpenGL fait un usage intensif des gradients. Par exemple, même s'il est parfaitement possible de calculer des gradients de couleurs dans le pixel shader, nous utilisons toujours une texture comme une table de référence car c'est beaucoup plus rapide. Répéter à plusieurs reprises l'envoi des textures à chaque fois que nous en avons besoin ruine les performances. Ainsi, à la place, nous gardons un cache pour chaque contexte de façon à ce que les QPixmap/QImage soient déjà présents dans la mémoire de texture. Si deux contextes l'utilisent, alors nous pouvons le détecter et ne pas dupliquer les textures. Cette fonctionnalité est disponible publiquement dans QGLContext::bindTexture().

Pour les plateformes Linux/X11 qui le supportent, Qt utilise l'extension glX/EGL texture-depuis-une-pixmap. Cela signifie que si votre QPixmap a une vraie pixmap dans le backend X11, il suffit de lier cette pixmap X11 comme une texture et d'éviter de la copier. Vous utiliserez la pixmap du backend X11 si une pixmap a été créé avec QPixmap::fromX11Pixmap() ou si vous utilisez le système graphique natif. Non seulement ceci évite les frais généraux, mais il vous permet également d'écrire un gestionnaire de composition ou même un widget qui affiche des aperçus de toutes vos fenêtres.

III-e - Anti-aliasing

Le moteur de rendu OpenGL utilise l'échantillonnage multiple d'OpenGL pour fournir l'anti-aliasing. Typiquement, ça sera du 4x/8x FSAA, ce qui signifie 4/8 niveaux de couverture, ce qui est de moins bonne qualité que le moteur Raster, qui utilise toujours 256 niveaux de couverture. Toutefois, comme la résolution des écrans modernes ne fait qu'augmenter, vous pouvez vous contenter d'anti-aliasing de moins bonne qualité.

L'utilisation de l'échantillonnage multiple n'affecte pas non plus le rendu de texte puisque son anti-aliasing est réalisé en utilisant des masques plutôt que l'échantillonnage multiple (pour les tailles de police les plus petites). Ainsi, le texte rendu avec le moteur OpenGL devrait être presque aussi bon que le texte rendu avec le moteur Raster (qui fait aussi de la correction gamma). Le seul inconvénient de l'utilisation de l'échantillonnage multiple, c'est que certaines implémentations d'OpenGL ne prennent pas en charge la désactivation de l'échantillonnage multiple. En effet, les spécifications d'OpenGL ES 2.0 ne prévoient même pas l'API pour le désactiver. La conséquence est que les rendus sans anti-aliasing (c'est-à-dire avec aliasing) peuvent être brisés (tout est rendu avec anti-aliasing, même lorsque l'option QPainter::Antialiasing n'est pas défini). Il y a peu de choses que nous pouvons faire à ce sujet.

III-f - Découpage

QPainter supporte l'utilisation d'un découpage quelconque, y compris des QPainterPath complexes. Qt utilise le stencil buffer d'OpenGL (ou plus précisément les 7 bits inférieurs du stencil buffer) pour stocker la zone de découpage. La zone à découper est écrite de la même manière que nous réalisons le rendu de toutes les autres primitives, c'est-à-dire en utilisant la technique du stencil pour les chemins complexes. Cependant, au lieu de remplir les couleurs des pixels dans le colour buffer, nous remplissons les valeurs du stencil dans le stencil buffer. La valeur réelle que

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

nous utilisons dépend de la profondeur de la pile de QPainter (combien de fois save() a été appelé moins le nombre de fois que restore() a été appelé). Cela signifie que si vous vous limitez à calculer les intersections des zones de découpage (**Qt::ClipOperation** est défini par **Qt::IntersectClip**), le moteur a seulement besoin d'écrire dans la partie du stencil buffer qui a besoin d'être découpée. De plus, le moteur n'a pas besoin du tout d'écrire dans le stencil buffer lorsque vous appelez restore() : il change simplement la valeur pour laquelle le test de stencil passe.

En plus d'utiliser le stencil buffer pour le découpage, le moteur de rendu OpenGL peut aussi simplement utiliser glScissor. Cela permet seulement à un seul rectangle non transformé d'être utilisé comme une zone de découpage, ce qui peut être assez restrictif. Toutefois, c'est de loin le moyen le plus rapide de faire du découpage. Donc, si le rendement est plus important pour vous que l'utilité, toujours utiliser la zone de découpage rectangulaire non transformée.

IV - Recommandations

IV-a - Rendus entrecroisés

Contrairement à OpenGL, QPainter permet un nombre arbitraire de contextes de rendu (QPainter) à être actifs dans le même thread en même temps. Par exemple, dans la fonction paintEvent de votre widget, vous pouvez commencer un QPainter sur votre widget et commencer un QPainter sur une QPixmap et entrecroiser les rendus entre eux :

```
void Widget::paintEvent(QPaintEvent*)
{
 QPainter widgetPainter(this);
 widgetPainter.fillRect(rect(), Qt::blue);
 QPixmap pixmap(256, 256);
 QPainter pixmapPainter(&pixmap);
 pixmapPainter.drawPath(myPath);
 widgetPainter.drawPixmap(0, 0, &pixmap);
}
```

Bien que cela fonctionne bien avec le système graphique OpenGL, devoir basculer entre ce que vous faites avec un QPainter pour faire autre chose avec un autre QPainter peut être très coûteux et doit être évité autant que possible.

IV-b - Mélanger QPainter avec de l'OpenGL native

Comme montré dans plusieurs exemples, il est possible de mélanger vos propres codes de rendu OpenGL avec du code de rendu QPainter. Cependant, comme OpenGL est une gigantesque machine à états, il est très facile pour vous de mettre à mal accidentellement l'état d'OpenGL dans Qt et vice versa. Pour surmonter cela, nous avons ajouté quelques nouvelles API dans QPainter dans Qt 4.6 : QPainter::beginNativePainting() et QPainter::endNativePainting(). Pour éviter les artefacts, vous devez encadrer votre rendu personnalisé entre beginNativePainting() et endNativePainting(). C'est très important : même si vous ne voyez pas de problème de rendu pour le moment, vous pourrez voir votre code commencer à planter dans une future version de Qt, dans laquelle le moteur de rendu OpenGL pourra fonctionner très différemment. De plus, comme beginNativePainting() et endNativePainting() modifient beaucoup de variables d'état d'OpenGL, ça peut être très couteux et vous devez alors essayer de l'utiliser avec modération. Essayez d'écrire votre propre OpenGL dans un seul bloc.

IV-c - QGLWidget vs le système graphique OpenGL

Contrairement aux moteurs de rendu Raster et OpenVG, vous n'avez pas besoin d'utiliser un système graphique spécifique pour réaliser des dessins avec le moteur de rendu OpenGL. Le module QtOpenGL propose plusieurs classes, dont WGLWidget, lesquelles permettent le moteur de rendu OpenGL indépendamment du système graphique utilisé. QGLWidget est à la base un widget standard qui a toujours un identifiant de fenêtre native et qui réalise toujours le rendu en utilisant OpenGL. Vous êtes libre de choisir quelle méthode vous souhaitez utiliser pour réaliser les rendus OpenGL (le système graphique ou QGLWidget). Cependant, utiliser le système graphique OpenGL peut souvent être plus lent que d'utiliser QGLWidget, puisque Qt a besoin de mettre le contenu dans un tampon de

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

retour (ou dans une QWindowSurface) pour qu'il soit préservé lors que l'on envoie le rendu dans la fenêtre système. OpenGL ne garantit pas cela et ce n'est pas souvent le cas, Qt devra donc utiliser un FBO (Frame buffer objet) ou un PBuffer (Pixel buffer) comme tampon de retour. Lors que le rendu a besoin d'être envoyé, le FBO ou le PBuffer est appliqué dans une texture puis est dessiné dans une fenêtre et les tampons mémoire de travail d'OpenGL sont échangés. Cette fonctionnalité supplémentaire est disponible avec QGLWidget, mais a comme conséquence qu'il n'est pas possible de redessiner une région spécifique de QGLWidget : à chaque fois qu'un QGLWidget est mis à jour, la totalité du widget doit être redessiné.

Il faut également noter qu'utiliser le moteur de rendu OpenGL n'est pas comme une balle en argent, qui peut rendre tout plus rapide. Par exemple, le moteur OpenGL est vraiment mauvais pour dessiner beaucoup de petites figures avec des changements d'états entre chaque opération de dessin. Tant que nous travaillons pour le moment pour améliorer ce cas d'utilisation, le moteur de rendu Raster restera probablement toujours plus rapide, seulement parce qu'il fait beaucoup moins de frais généraux. Donc QGLWidget peut être un bon moyen d'avoir les meilleurs des deux mondes en le combinant avec le système graphique Raster : utilisez QGLWidget pour les opérations pour lesquelles OpenGL est excellent et le moteur Raster pour tout le reste.

IV-d - Conseils pour les performances (fps)

Comme règle générale : les changements d'états d'OpenGL sont très coûteux. Donc, utiliser les connaissances que vous avez maintenant sur ce qui se passe derrière QPainter et essayez de minimiser le nombre de changement d'états d'OpenGL que le moteur de rendu a besoin de faire. Par exemple, si vous implémentez un clavier virtuel, vous savez maintenant que le moteur utiliser un shader pour le rendu du texte et un shader différent pour les pixmaps. Donc, dessinez l'ensemble des pixmaps pour les touches en premier puis dessinez tout le texte par-dessus. De cette manière, le moteur a seulement besoin de changer les shaders deux fois par affichage.

- N'utilisez jamais autre chose que des découpages qui se croisent ;
- ne changez pas de cible de rendu en plein milieu d'un rendu ;
- essayez d'utiliser les découpages non transformés rectangulaires dans la mesure du possible ;
- réduisez les changements de brosse dans la mesure du possible ;
- réalisez le rendu d'un groupe de primitives de même type ensemble ;
- évitez de dessiner les pixels transparents et le blending (particulièrement important sur les GPU mobiles) ;
- essayez de mettre en cache QPainterPaths et de les réutiliser plutôt que de les créer et de les supprimer dans votre paintEvent ;
- utilisez QPainterPaths même quand il y a une fonction plus pratique dans QPainter (par exemple pour les rectangles arrondis et les ellipses ;
- si vous dessinez beaucoup de petites pixmaps, essayez de les regrouper en une seule pixmap plus grande ;
- préférez utiliser une hauteur et une largeur multiple d'une puissance de deux (2^n) pour QImage et QPixmap (128x256, 256x256, 512x512, etc.)
- si vous utilisez QGLWidget et que vous n'avez pas besoin d'anti-aliasing, n'activez pas les tampons d'échantillons dans le QGLFormat ;
- si vous réalisez le rendu d'un QPainterPath complexe, essayez de n'utiliser que la **règle de remplissage**.

IX - Conclusion

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 !

Merci à **dourouc05** et à **Mahefasoa** pour leur relecture !

Copyright ® 2010 Gunnar. 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 trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.