Qt Graphics et performance - La folie est de mettre en forme le même texte encore et espérer un résultat différent

Qt by Nokia

Par Eskil Abrahamsen Blomfeldt - traducteur : Guillaume Belz - Qt Labs

Date de publication : 11 décembre 2010

Dernière mise à jour : 16 mars 2011

On attribue à Albert Einstein la phrase : « la folie est de faire la même chose encore et encore en espérant un résultat différent ». Apparemment, la citation est mauvaise et il faut en fait l'attribuer à Rita Mae Brown mais ce n'est pas important pour le moment. Ce qui est important est que la plupart des applications Qt sont folles.

Cet article est une traduction autorisée de **Insanity is shaping the same text again and expecting a different result** de **Eskil Abrahamsen Blomfeldt**.

N'hésitez pas à commenter cet article !
Commentez

I - L'article original

Qt Labs est un site géré par les développeurs de Qt. Ils y publient des projets, des idées propres et des composants afin d'obtenir les retours d'informations sur les API, le code et les fonctionnalités ou simplement pour partager avec nous ce qui les intéressent. Le code que vous y trouverez peut fonctionner tel quel mais c'est sans aucune garantie ni support. Voir les **conditions d'utilisation** pour plus d'informations.

Nokia, Qt, Qt Labs 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 **Insanity is shaping the same text again and expecting a different result**, par **Eskil Abrahamsen Blomfeldt** paru dans Qt Labs.

II - Introduction

On attribue à Albert Einstein la phrase : « la folie est de faire la même chose encore et encore en espérant un résultat différent ». Apparemment, la citation est mauvaise et il faut en fait l'attribuer à **Rita Mae Brown** mais ce n'est pas important pour le moment. Ce qui est important est que la plupart des applications Qt sont folles.

III - Contexte

Je vais vous expliquer. Certains lecteurs se souviennent de la série d'excellents blogs de Gunnar concernant **les performances graphiques** sur la façon d'obtenir le meilleur résultat dans Qt. À quelques reprises, il a mentionné le fait que le rendu des textes en Qt est plus lent que nous le souhaiterions.

Pour comprendre pourquoi le rendu du texte est si lent, il est nécessaire de regarder ce qui se passe lorsque vous passez une **QString** à **QPainter::drawText()** et que vous demandez de l'afficher à l'écran. Un **QString** est juste un tableau de valeurs entières qui sont définies pour correspondre à des symboles spécifiques dans certains systèmes d'écriture. Comment ces symboles doivent effectivement être affichés sur l'écran est défini en fonction de la police que vous avez sélectionnée dans votre moteur de rendu.

Donc la première étape de **drawText()** est de prendre les valeurs entières et de les transformer en valeurs d'index, lesquelles servent de références dans la table interne de la police. Les indices sont spécifiques à chaque police et n'ont aucune signification en dehors du contexte de la police courante.

La deuxième étape de **QPainter::drawText()** est de recueillir des informations de la police qui décrivent la façon dont le symbole doit être positionné par rapport aux symboles qui se trouvent à proximité. Cette étape, le positionnement de chaque symbole, est potentiellement très complexe. Plusieurs tables différentes dans le fichier de police doivent être consultées, avec des programmes et des instructions qui peuvent faire des choses comme par exemple le crénage (permettant aux parties de certains glyphes **to "hang over" og "stretch underneath" other glyphs**) et placer un ou plusieurs signes diacritiques sur le même caractère. Certains systèmes d'écriture permettent également la réorganisation complexe des symboles basés sur le contexte entourant les caractères, comme l'a expliqué Simon dans **son blog** depuis 2007. Cette mise en forme complexe du texte est actuellement gérée par la bibliothèque **Harfbuzz** dans Qt.

La troisième étape ne s'applique que si on attribue une mise en page sur le texte. La mise en page permet de séparer le texte en lignes correctement formatées. Dans Qt, cela peut être réalisé avec du code HTML, en utilisant **QTextDocument** ou **QtWebKit**, ou cela peut être une simple mise en page, pour laquelle il suffit d'inclure et d'aligner le texte dans un rectangle. Le premier n'est pas pris en charge par **QPainter::drawText()** donc je vais mettre l'accent sur ce dernier. En utilisant les informations fournies lors de l'étape de la mise en forme, l'étape de mise en page du texte calcule la largeur de portions insécables du texte et tente de formater le texte d'une manière qui semble correcte à l'écran mais qui ne s'étendent pas au-delà des limites fixées par l'utilisateur.

Dans la quatrième et dernière étape, le moteur de rendu prend le relais. Son travail consiste à dessiner les symboles trouvés dans la première étape à la position calculée dans les deuxième et troisième étapes. Dans la plupart des moteurs de rendu de Qt sensibles aux performances, cela se fait en mettant en cache une pixmap de la représentation d'un symbole la première fois que celui-ci est dessiné puis de simplement redessiner cette pixmap pour chaque appel. C'est potentiellement très rapide.

Bien que ces quatre étapes puissent être légèrement imbriquées dans Qt aujourd'hui, c'est en principe ce qui se passe chaque fois que vous appelez **drawText()** avec un **QString** et un **QRect** enveloppant. Pourtant, dans de très nombreux cas, à la fois le texte, la police et le rectangle restent totalement statiques pendant toute la durée de votre application, ou tout au moins pendant une grande partie de celle-ci. Et c'est la partie folle : beaucoup de temps est gaspillé ici. Qt fournit déjà **QTextLayout** comme un moyen de mettre en cache les résultats des trois premières étapes et de les envoyer directement dans le moteur de rendu. Toutefois, **QTextLayout** est un peu compliqué à utiliser, il a des fonctionnalités supplémentaires liées à ses autres possibilités d'utilisation, et il stocke beaucoup plus d'informations que ce qui est nécessaire pour afficher en particulier des symboles à l'écran, ce qui est très insatisfaisante dans des configurations où la mémoire est très importante.

IV - QStaticText !

Nous avons décidé qu'il fallait une classe spécialisée pour résoudre ce problème. Nous l'avons nommée **QStaticText** et elle sera disponible dans Qt 4.7. **QStaticText** a été optimisée spécifiquement pour redessiner du texte qui ne change pas entre deux **paintEvent**. Nous avons essayé de minimiser l'occupation mémoire et actuellement, il y a environ 14 octets supplémentaires par symbole (y compris les 2 octets par caractère Unicode dans la chaîne, lesquels font déjà probablement partie de l'application) plus environ 200 octets constants supplémentaires.

Dans le reste de ce blog, je vais vous montrer quelques graphiques pour illustrer les avantages de l'utilisation **QStaticText** pour dessiner du texte. **QStaticText** est pris en charge par le moteur Raster (le moteur de rendu logiciel utilisé par défaut sur Windows), le moteur OpenGL et le moteur OpenVG. Pour l'instant, je vais me focaliser dans ce blog sur le moteur Raster et le moteur OpenGL. Je vais aussi me focaliser sur les plateformes suivantes : Windows/desktop, Linux/desktop et le N900 (également sous Linux, bien sûr). Notez que le matériel sur les machines Windows et Linux est différent, donc les résultats ne seront pas comparables d'une plateforme à l'autre.

V - Test de performance pour cinquante caractères de texte sur une seule ligne

Le test de performance que j'ai réalisé est le suivant : dessiner la même chaîne de 50 caractères encore et encore, dans chaque **paintEvent** et mesurer combien de "symboles par seconde" pouvons-nous atteindre en utilisant différentes techniques pour dessiner le texte. J'ai testé les méthodes de dessin de texte suivantes :

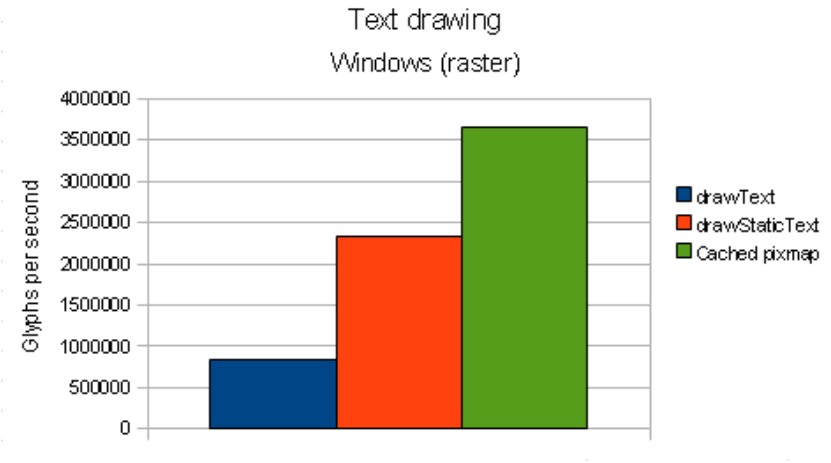
- appeler **QPainter::drawText()** sans rectangle enveloppant ;
- appeler **QPainter::drawStaticText()** sans rectangle enveloppant ;
- mettre toute la chaîne dans une pixmap dans un premier temps et la redessiner dans chaque **paintEvent** en utilisant **QPainter::drawPixmap()**.

Lors des tests sur le moteur de rendu OpenGL, le graphique contient également les résultats pour **QStaticText** avec l'option de performance **QStaticText::AggressiveCaching**. Il s'agit d'une option qui permet au moteur de rendu de mettre en cache ses propres données, échangeant ainsi de la mémoire contre de la vitesse. Il est actuellement utilisé par le moteur OpenGL pour mettre en cache les tableaux de coordonnées contenant les sommets et les textures qui sont envoyés au processeur graphique au moment du rendu des symboles.

VI - Sur Windows

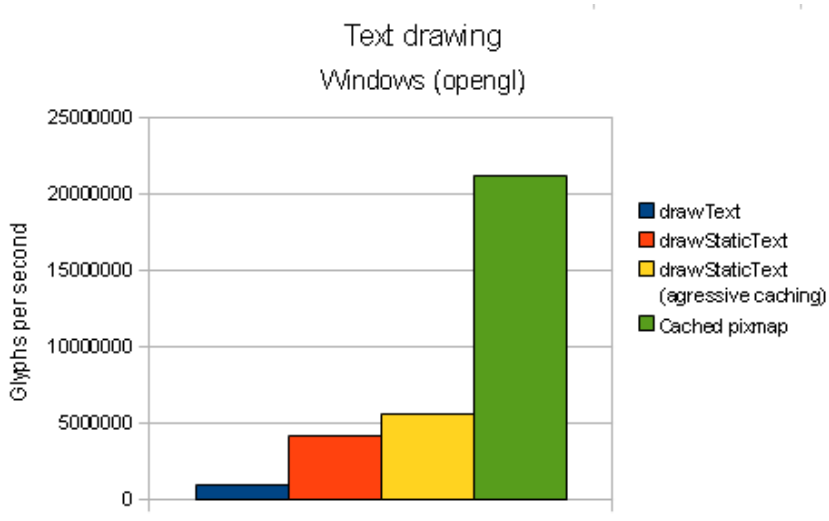
Commençons avec les résultats du moteur Raster sur Windows. Comme je l'ai dit, la mesure est en "symboles par seconde", c'est-à-dire le nombre de symboles que nous pouvons afficher à l'écran pendant une seconde d'exécution du test. La mesure est basée sur le taux de rafraîchissement du test, qui est pris comme étant la moyenne de neuf secondes d'exécution par cas de test. Notez que le rendu cleartype est désactivé dans le système d'exploitation

lors du test. La différence entre les résultats de **drawPixmap()** et de **drawStaticText()** serait plus grande encore avec cleartype activé mais cleartype n'est généralement pas pris en charge lorsque l'on met en cache du texte dans une pixmap, puisque la pixmap a besoin inévitablement d'avoir un fond transparent, et vous ne pouvez pas faire d'antialiasing sur un fond transparent. Par conséquent, tous les tests de performance sont réalisés sans antialiasing pour obtenir une meilleure comparaison.



Comme vous pouvez le voir, le moyen le plus rapide pour dessiner le texte est de le mettre en cache dans une pixmap et de dessiner celle-ci, redessiner une pixmap étant extrêmement rapide sur du matériel moderne. Cependant, dans bien des cas, vous n'avez pas la mémoire à perdre pour ce genre d'extravagance et **drawStaticText()** permet d'atteindre plus de la moitié du nombre de symboles par seconde que pour l'équivalent avec **drawStaticText()**. Il est aussi trois fois plus rapide qu'un appel standard à **drawText()**.

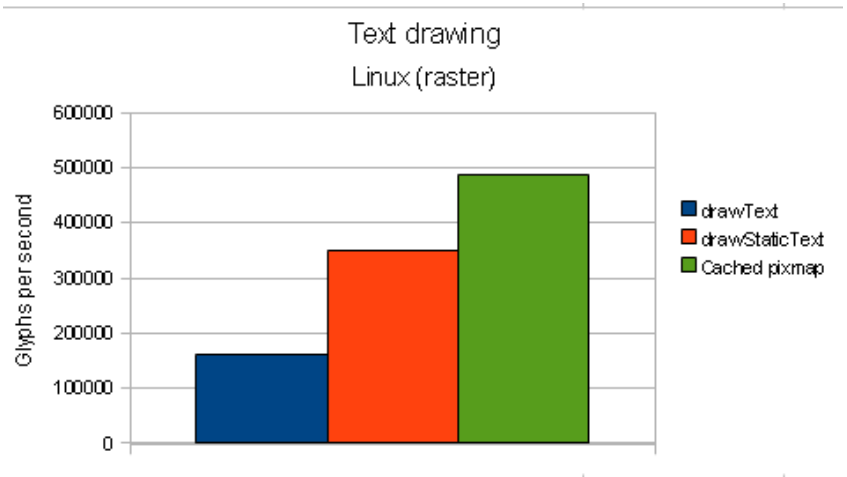
En utilisant le moteur de rendu OpenGL à la place, les performances de **drawPixmap()** explosent le plafond :



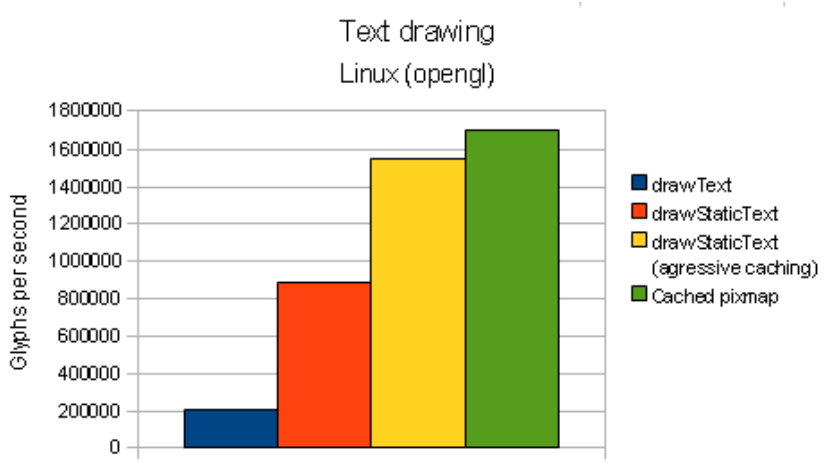
Les autres barres semblent petites en comparaison mais **drawStaticText()**, en utilisant l'option de mise en cache agressive des performances, dépasse en fait les 5,6 millions de symboles par seconde dans ce test, tandis qu'un appel standard à **drawText()** atteint un misérable 20 % de cela.

VII - Sur Linux

Des chiffres similaires sont obtenus sur Linux :



Utiliser **drawStaticText()** vous permet de doubler les performances par rapport à **drawText()** et **drawPixmap()** atteint un peu moins de 1,5 fois la vitesse du **drawStaticText()**. Lorsque vous utilisez le moteur OpenGL, la différence est moindre :

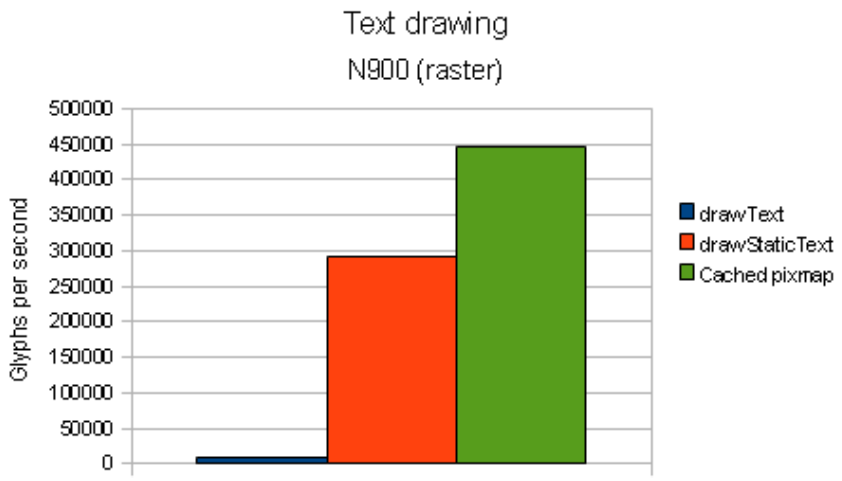


Comme vous pouvez le voir, dessiner une pixmap mis en cache sur Linux est à peine plus rapide que de dessiner du texte statique en utilisant l'option de mise en cache agressive. Le matériel et les pilotes sont importants ici mais de toute façon, nous pouvons voir que les deux donnent des résultats sept ou huit fois meilleurs que **drawText()**.

VIII - Sur N900

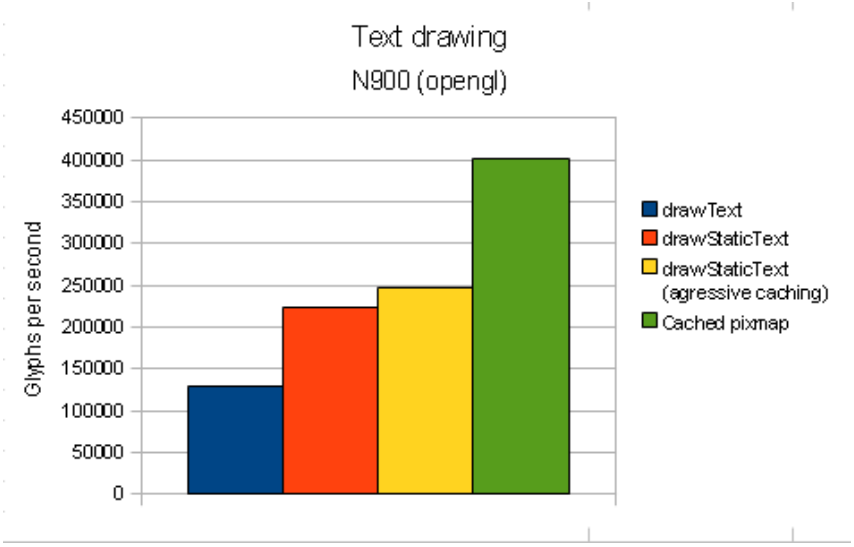
Tous les tests de performance ont été réalisés jusqu'à présent sur un ordinateur de bureau, où la mémoire est bon marché. Mettre en cache plusieurs objets texte dans des pixmaps est une goutte d'eau sur ces plateformes et comme nous l'avons vu, mettre en cache dans une pixmap est potentiellement très rapide. Cependant, sur un appareil embarqué, nous devons faire un peu plus attention quand nous allouons des blocks mémoires importants, donc quelque chose comme QStaticText, qui est à la fois peu gourmand et rapide, peut être un excellent outil sur ces plateformes. Regardons un peu quelques tests de performance sur un N900.

Pour le moteur Raster sur la N900, le rendement de **drawText()** est actuellement pour le moins horrible, comme vous pouvez le voir dans le graphique suivant :



C'est bien sûr un problème que nous étudierons de plus près car il n'y a aucune raison qui explique pourquoi c'est si lent d'appeler **drawText()** mais pour le moment, nous recommandons d'utiliser le moteur natif ou une fenêtre **QGLWidget** sur ce système. Au moins, les autres barres semblent vraiment grandes en comparaison. Un résultat plus intéressant, c'est que **drawStaticText()** peut produire les deux tiers du nombre de symboles par seconde que l'on peut obtenir en dessinant simplement une pixmap qui couvre la même superficie, donc nous avons un rapport de performance assez bon sur ce système.

Comme nous le voyons dans le graphique suivant, des chiffres similaires peuvent être obtenus en utilisant le moteur OpenGL :



IX - Conclusion

Les résultats des tests de performance présentés ici correspondent à un texte tenant sur une ligne unique ; il n'y a pas besoin d'accomplir la troisième étape décrite dans le fonctionnement de **drawText()**, présentée au début, où le texte est formaté selon une mise en page. Cela a des implications, à savoir que l'appel à **drawText()** peut sauter la troisième étape comme indiqué dans le début du blog, car elle n'a pas besoin de faire une mise en page du texte de haut niveau. Sur un texte qui a besoin de cette mise en page, le rendement sera encore pire avec **drawText()** mais à peu près le même avec **drawStaticText()** et **drawPixmap()**, puisque l'étape de mise en page a déjà été faite à l'avance. Une autre chose à noter est que le texte est assez long et assez dense. Pour du texte court et/ou avec beaucoup d'espaces (comme une chaîne multi-ligne aurait pu avoir), les performances de **drawStaticText()** peuvent très bien être meilleures que celles pour dessiner une pixmap, puisque le nombre de pixels touchés devient un facteur plus important dans l'équation.

Une mesure intéressante qui n'est pas présentée ici est la charge du processeur pour les différentes fonctions. Nous n'avons pas de tests de performance formels pour le moment, mais puisque le processeur consacre moins de temps à un travail intensif lors de l'utilisation **drawStaticText()** par rapport à **drawText()**, il aura plus de temps libre pour faire d'autres choses, ce qui est une bonne chose. Et une autre découverte agréable que nous avons faite lors des tests de performance de **drawStaticText()** sur le N900, (**is that you have to increase the number of draw-calls made per frame to a pretty high number for it to visibly factor into the time spent in the paint event**). Cela signifie que même avec, par exemple, cinquante chaînes de caractères, les appels à **drawStaticText()** ne devraient pas avoir un impact considérable sur les performances de l'application. Réaliser l'échange entre les tampons en premier plan et en arrière-plan restera le principal goulot d'étranglement, ce qui reste acceptable.

Le mot de la fin sera le suivant : si vous utilisez **drawText()** dans votre application pour afficher du texte qui n'est jamais ou très rarement mis à jour, alors vous pourrez envisager d'utiliser **QStaticText** à la place lorsque vous commencerez à utiliser Qt 4.7 et nous serons ravis d'écouter ce que vous en pensez de l'API et des performances une fois que vous aurez la chance de l'essayer.

Merci à **Mahefasoa** et à **jacques_jean** pour sa relecture et ses conseils.

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 !