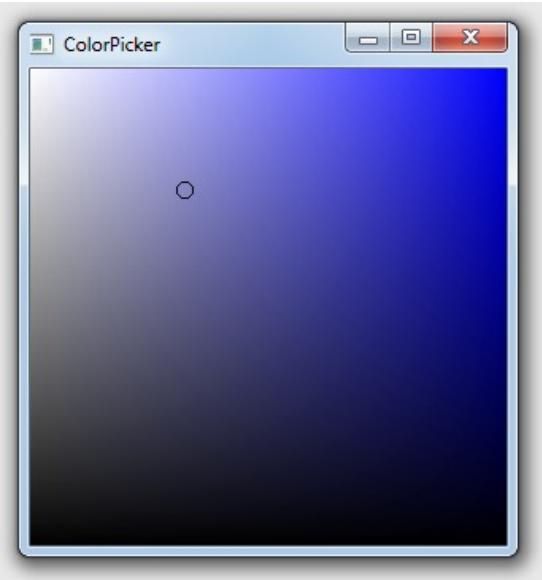
Benchmark et techniques d'optimisation du rendu

Le sujet du premier exercice proposé par la rubrique Qt de Developpez.com consistait à créer un widget permettant de sélectionner les différentes nuances de gris à partir d'une teinte définie.



Deux méthodes ont été proposer pour générer ces nuances de gris :

- utiliser deux boucles *for* imbriquées pour parcourir chaque pixel du widget et calculer la couleur correspondante, en faisant varier les paramètres S et V dans l'espace colorimétrique HSV ;
- Utiliser deux gradient (à l'aide de la classe *QLinearGradient*), le premier, horizontal, allant du blanc à gauche à la teinte choisie à droite, et le second, vertical, allant du transparent en haut au noir en bas.

Je vais présenter ici comment implémenter un benchmark avec Qt pour tester la rapidité de ces différentes méthodes et présenter deux autres méthodes pour accélérer le rendu : l'accès direct au tampon mémoire contenant l'image finale et l'accélération matériel sur carte graphique.

Plan de l'article

- Création du programme Benchmark
  - Benchmark.pro
  - benchmark.h
  - benchmark.cpp
  - Méthode des deux boucles for
  - Méthode des deux gradients linéaires
  - Résultat du benchmark
- Rendu dans un tampon mémoire
  - Résultat du benchmark
- Optimisation de l'algorithme
  - Résultat du benchmark
- Accélération matériel sur carte graphique
  - Performances

Création du programme *Benchmark*

Qt offre différents outils facilitant la création de tests unitaires et de benchmarks, rassemblés dans le module QtTest (http://qt.developpez.com/doc/4.6/QtTest/). Différents tutoriels permettent de se familiariser progressivement avec ce module (http://qt.developpez.com/doc/4.6/qtestlib-tutorial/).

Benchmark.pro

Pour créer notre programme *Benchmark*, commençons par créer un fichier de projet *Benchmark.pro*. Ce fichier de projet contient les déclarations de base suivantes : la liste des modules utilisés (*core*, *gui* puisse que l'on teste les fonctions de dessin et *testlib*) et le nom et le type de binaire généré (une application *Benchmark* dans notre cas) :

```
QT += core gui testlib
TARGET = Benchmark
TEMPLATE = app
```

On ajoute ensuite la liste des fichiers à inclure dans notre projet. On va simplement créer une classe Benchmark : on ajoute donc les fichiers d'en-tête et d'implémentation :

```
SOURCES += benchmark.cpp
HEADERS  += benchmark.h
```

benchmark.h

La classe *Benchmark* hérite de QObject (pour pourvoir utiliser le système de signaux et slots) et contient deux slots privés, un pour chaque méthode que l'on souhaite tester. Ces slots seront automatiquement appelés à l'exécution du programme. A part cela, le contenu de la classe est minimaliste :

```
#ifndef BENCHMARK_H
#define BENCHMARK_H

#include <QtTest/QtTest>

class Benchmark : public QObject
{
    Q_OBJECT

private slots:
    void hsvGradient();
    void doubleLinearGradient();
};

#endif // BENCHMARK_H
```

benchmark.cpp

Dans le fichier d'implémentation, on ajoute chaque méthode dans les slots correspondants :

```
#include "benchmark.h"

void Benchmark::hsvGradient()
{
}

void Benchmark::doubleLinearGradient()
{
}
```

Il faut également ajouter la macro QTEST_MAIN, qui crée une fonction *main*, qui aura pour fonction de créer une instance de la classe *Benchmark* et d'exécuter les slots privés :

```
QTEST_MAIN(Benchmark)
```

Méthode des deux boucles for

Pour mesurer le temps d'exécution d'une méthode, on utilise la macro BENCHMARK. Le temps du reste du code contenu dans le slot mais en dehors du bloc précédent la macro n'est pas évalué. Pour vérifier que le code généré correctement les nuances de gris, on enregistre les images produites :

```
#include <QtGui/QPainter>
const int width = 500;
const int height = 500;
const QColor m_main_color = Qt::red;

void Benchmark::hsvGradient()
{
    QImage image = QImage(QSize(width, height), QImage::Format_ARGB32);

    QBENCHMARK
    {
        float h = m_main_color.hsvHueF();
        for (int s=0; s<width; ++s)
        {
            for (int v=0; v<height; ++v)
            {
                QColor color = QColor::fromHsvF(
                        h,
                        1.0 * s / (width-1),
                        1.0 - (1.0 * v / (height-1)));
                image.setPixel(s, v, color.rgb());
            }
        }
    }

    image.save("hsvGradient.png");
}
```

Méthode des deux gradients linéaires

```
void Benchmark::doubleLinearGradient()
{
    QImage image;

    QPixmap pixmap(QSize(width, height));
    QPainter painter(&pixmap);
    painter.setPen(QPen(Qt::NoPen));

    QBENCHMARK
    {
        QLinearGradient h_gradient(QPointF(1.0, 0.0), QPointF(width, 0.0));
        h_gradient.setColorAt(0, Qt::white);
        h_gradient.setColorAt(1, m_main_color);
        painter.setBrush(QBrush(h_gradient));
        painter.drawRect(QRect(0, 0, width, height));

        QLinearGradient v_gradient(QPointF(1.0, 0.0), QPointF(0.0, height));
        v_gradient.setColorAt(0, Qt::transparent);
        v_gradient.setColorAt(1, Qt::black);
        painter.setBrush(QBrush(v_gradient));
        painter.drawRect(QRect(0, 0, width, height));

    }

    image = pixmap.toImage();
    image.save("doubleLinearGradient.png");
}
```

Le code étant expliqué dans la solution de l'exercice, je n'entre pas dans le détail ici.

Résultat du benchmark

Lorsque le programme est exécuté, des messages sont générés automatiquement à chaque étape du processus. L'en-tête décrit simplement le nom du programme et les versions de *QtTest* et de *Qt* utilisées :

```
********* Start testing of Benchmark *********
Config: Using QTest library 4.7.0, Qt 4.7.0
```

Le premier slot que lance le programme est un slot spécifique, *initTestCase*, généré par défaut et ne faisant rien. Ce slot permet, si on le souhaite, d'initialiser des objets pour l'ensemble du programme :

```
PASS : Benchmark::initTestCase()
```

Vient ensuite les différents slots que l'on a créé. Le premier, *hsvGradient*, . Lorsque le temps d'exécution du slots est trop petit, la mesure risque de ne pas être précise. Pour palier à cet inconvénient, le code contenu dans le bloc de code suivant la macro *BENCHMARK* peut être exécuté plusieurs fois, le temps indiqué correspondant au temps moyen mesuré. Ici, le slot *hsvGradient* a été exécuté deux fois et le temps moyen est de 47 millisecondes :

```
RESULT : Benchmark::hsvGradient():
47 msecs per iteration (total: 94, iterations: 2)
PASS : Benchmark::hsvGradient()
```

Pour le slot *doubleLinearGradient*, le code a été exécuté une seule fois et le temps mesuré est de 57 millisecondes. On voit donc ici que cette méthode est plus lente que la première version (sur le système testé ! C'est à dire Linux dans mon cas. Sous Windows ou Max OS X, il est possible que les résultats soient différents). Cette différence peut s'expliquer par le fait que la génération de deux gradients et la gestion de la transparence est plus lente que l'accès directe aux pixels, sur le système de rendu par défaut (le système Raster) :

```
RESULT : Benchmark::doubleLinearGradient():
57 msecs per iteration (total: 57, iterations: 1)
PASS : Benchmark::doubleLinearGradient()
```

Le programme exécute ensuite un slot pour libérer les ressources initialisées dans la fonction *initTestCase* :

```
PASS : Benchmark::cleanupTestCase()
```

Pour terminer, le programme indique le nombre de slot exécutés sans incident (*passed*), ceux qui ont échoué (*failed*) et ceux qui n'ont pas été exécutés (*skipped*) :

```
Totals: 4 passed, 0 failed, 0 skipped
********* Finished testing of Benchmark *********
```

Rendu dans un tampon mémoire

Dans la première méthode, on utilise la fonction *Qimage::setPixel* pour modifier chaque pixel un par un. Cette méthode permet d'utiliser une interface sécurisée et simple. Malheureusement, les calculs (calcul de l'adresse mémoire correspondant au pixel, conversion de l'espace colorimétrique HSV en RGB) et tests effectués en interne (la position du pixel est bien dans l'image ? Les valeurs de la couleurs sont-elles correctes ?) peuvent ralentir fortement l'exécution.
Il est alors possible d'optimiser le rendu en accédant directement à la mémoire à l'aide de pointeurs : on créer un bloc mémoire de taille suffisante que l'on alloue à une image puis on accède à chaque octet un par un.

ATTENTION : ce type d'approche peut comporter certains risques : en cas d'erreur dans le code, il est possible que l'on écrive dans une zone mémoire qui ne correspond pas à l'image. Aucun test n'est effectué pour vérifier que le pointeur pointe vers une zone valide. L'application peut devenir instable. De plus, il est possible que le code ne soit pas portable et donne des résultats différents en fonction des systèmes d'exploitation utilisés (je l'ai testé uniquement sur Linux, pensez à bien enregistrer tous vos documents avant de lancer le programme sur un autre système:)).

La première chose a faire est de créer une zone mémoire de taille suffisante : un pixel étant codé sur 32 bits (un octet par composante de la couleur et quatre composantes : rouge, vert, bleu et alpha), il faudra donc réserver un bloc mémoire de width * height * 4 pour contenir toute l'image :

```
void Benchmark::drawInBuffer()
{
    uchar buffer[width * height * 4];
    QImage image(buffer, width, height, QImage::Format_ARGB32);
```

Il n'est pas nécessaire d'initialiser chaque octet du bloc mémoire (avec *memset* par exemple) puisque l'algorithme utilisé garanti que chaque octet sera initialisé durant l'exécution de celui-ci.

Pour commencer, on crée un pointeur pointant vers le premier octet du bloc mémoire :

```
    QBENCHMARK
    {
        uchar* pbuffer = &buffer[0];
```

On utilise deux boucle for, comme dans la première méthode, pour parcourir chaque pixel (en fait, on parcourt le bloc mémoire linéairement, en « déplacant » le pointeur octet par octet et non en accédant aux coordonnées (x, y) comme dans le cas d'une image) :

```
        for (int i=0; i<width; ++i)
        {
            //uchar green_blue = red;
            for (int j=0; j<height; ++j)
            {
```

Les valeurs de composantes RGB sont calculés par la méthode présentée dans la version QML de la solution de l'exercice :

```
                float a = 255.0 * (1.0 - (1.0 * i / width));
                float b = a * (1.0 - (1.0 * j / height));
```

Ensuite, pour chaque composante, on affecte la valeur calculée (sous forme de *uchar* pour être sur qu'elle occupe bien à un octet en mémoire) à l'octet pointé par le pointeur (en dé-référençant celui-ci) puis on « avance » le pointeur d'un octet :

```
                *pbuffer = (uchar) b; // blue
                ++pbuffer;
```

On fait de même pour les autres composantes :

```
                *pbuffer = (uchar) b; // green
                ++pbuffer;

                *pbuffer = (uchar) a; // red
                ++pbuffer;

                *pbuffer = (uchar) 255; // alpha
                ++pbuffer;
            }
        }
    }
```

Pour terminer, on enregistre l'image générée :

```
    image.save("buffured.png");
}
```

Résultat du benchmark

Les messages de sortie d'application générés par le slot *drawInBuffer* indique que cette méthode est plus rapide d'un facteur 10 que les deux autres méthodes, avec un temps d'exécution moyen de 3,9 millisecondes :

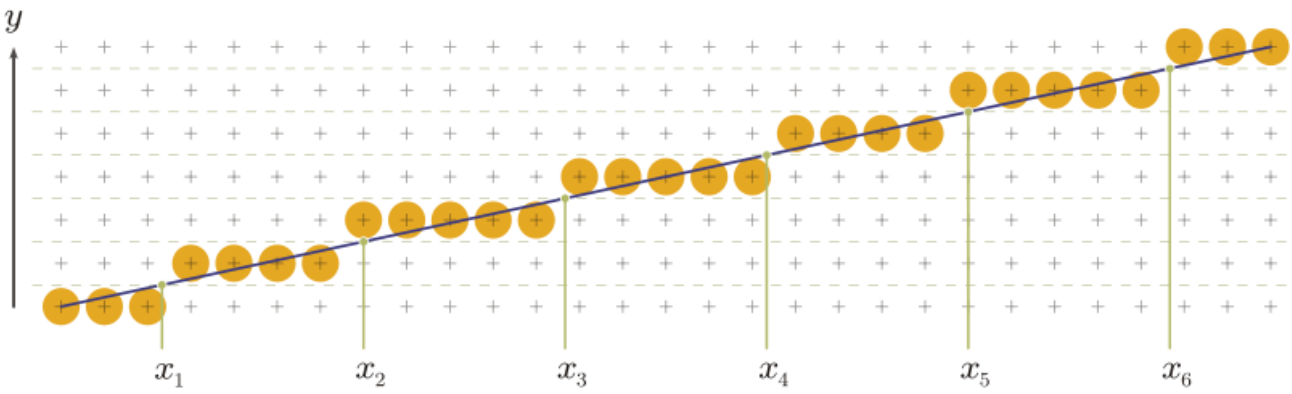
```
RESULT : Benchmark::drawInBuffer():

3.9 msecs per iteration (total: 63, iterations: 16)
PASS : Benchmark::drawInBuffer()
```

Optimisation de l'algorithme

Jusqu'à présent, nous avons simplement travailler sur l'implémentation pour optimiser le rendu des nuances de gris. Une autre approche consiste à travailler sur l'algorithme utilisé et d'essayer de diminuer les calculs effectués.
Quelque soit l'approche utilisée, il sera nécessaire de parcourir l'ensemble des pixels du widget, soit explicitement, comme dans le cas de la première méthode présentée, soit implicitement, comme dans le cas des deux *QLinearGradient*. On ne pourra donc pas optimiser cette partie (on peut au mieux éviter de parcourir plusieurs fois chaque pixel, comme on fait avec la méthode avec les deux *QLinearGradient*).
On peut remarquer que, dans chaque méthode présentée jusqu'ici, on travaille avec des nombres réels, ce qui n'est pas optimal. L'idéal serait de travailler qu'avec des nombres entiers, mieux gérés par les processeurs, et d'effectuer que des opérations élémentaires : addition, soustraction et multiplication.

Regardons en détail ce que l'on souhaite faire lorsque l'on parcourt un ligne pour une composante en particulier. La composante est une valeur entière allant de 0 à 255 (ou de 255 à 0) pour les pixels allant de 0 à N (avec N > 256 pour être sur d'afficher toutes les nuances). Il faut donc parcourir chaque pixel et incrémenter, *quand c'est nécessaire*, la composante. Si on représente cela avec un graphique 2D, on obtient :



avec x, l'index des pixels, et y, la composante de la couleur
(http://commons.wikimedia.org/wiki/File:Bresenham_run-based.svg)

Le lecteur averti aura compris l'idée sous-jacente : calculer la composante de la couleur revient à dessiner une ligne dans un espace (x, y) discret. Or, il existe un algorithme très performant pour réaliser cela : l'algorithme de Bresenham (http://fr.wikipedia.org/wiki/Algorithme_de_trac.%C3%A9_de_segment_de_Bresenham). Il suffit juste de remplacer y par la composante de la couleur. Le lecteur se reportera à l'article de Wikipédia pour l'explication du principe de cet algorithme.

L'implémentation de l'algorithme est assez triviale à partir du pseudo-code. On fera attention aux notations utilisées, qui correspondent au pseudo-code, c'est à dire que *y1* correspondent à la composante de la couleur et non à la position verticale du pixel. La variable *i* est utilisée pour parcourir verticalement l'image dans une boucle *for* :

```
QImage image(width, height, Qimage::Format_ARGB32);

int x1 = 0; // premier pixel
int y1 = 0; // composante dans le premier pixel
const int x2 = width - 1; // dernier pixel
const int y2 = 255; // composante dans le dernier pixel
```

```
int e = x2 - x1;
const int dx = 2 * e;
const int dy = 2 * (y2 - y1);

while (x1 <= x2)
{
    for (int i=0; i<height; ++i)
        image.setPixel(QPoint(x1, i), qRgb(y1, y1, y1));

    ++x1;
    e -= dy;
    if (e <= 0)
    {
        ++y1;
        e += dx;
    }
}

image.save("bresenham.png");
```

L'image générée est un gradient allant du noir au blanc :

Pour générer toutes les nuances, il va falloir imbriquer deux fois algorithme, le premier allant du blanc à la teinte choisie (première ligne de pixels) et le second allant de la teinte (pixel de la première ligne) au noir (pixel de la dernière ligne). Il faut également gérer chaque composante indépendamment.

Voir le code dans le fichier joint.

Pour terminer, il est possible de combiner cette méthode avec la méthode utilisant un tampon mémoire. Le code est très proche du précédent.

Voir le code dans le fichier joint.

Résultat du benchmark

La méthode utilisant l'algorithme amélioré permet de gagner un facteur 3 à 4 par rapport aux premières méthodes :

```
RESULT : Benchmark::bresenham():
14 msecs per iteration (total: 58, iterations: 4)
PASS : Benchmark::bresenham()
```

La version combinant les deux approches permet un gain de performance d'un facteur 4 par rapport à la méthode utilisant un tampon mémoire et un facteur 50 par rapport au premières méthodes :

```
RESULT : Benchmark::bresenhamInBuffer():
1.1 msecs per iteration (total: 72, iterations: 64)
PASS : Benchmark::bresenhamInBuffer()
```

Accélération matériel sur carte graphique

Pour réaliser le rendu des nuances de gris sur carte graphique, on utilise une classe dérivée de *QGLWidget*. Il suffit de dessiner un rectangle s'adaptant à la taille du widget, le reste du travail de rendu est réalisé dans le *fragment shader* : on calcule le rapport entre la position du pixel et la largeur du widget (passé en paramètre) pour calculer la nuance de gris correspondante :

```
"uniform vec4 main_color;\n"
"uniform vec2 size;\n"
"vec4 pixel_color;\n"

"void main(void)\n"
"{\n"
"    pixel_color.r = (gl_FragCoord.y / size.y) * (1.0 + (gl_FragCoord.x /
size.x) * (main_color.r - 1.0));\n"
"    pixel_color.g = (gl_FragCoord.y / size.y) * (1.0 + (gl_FragCoord.x /
size.x) * (main_color.g - 1.0));\n"
"    pixel_color.b = (gl_FragCoord.y / size.y) * (1.0 + (gl_FragCoord.x /
size.x) * (main_color.b - 1.0));\n"
"    pixel_color.a = main_color.a;\n"
"    gl_FragColor = pixel_color;\n"
"}\n";
```

Pour que le *fragment shader* soit appliqué sur chaque pixel du widget, il suffit de dessiner un quadrilatère sur la totalité de la surface :

```
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_QUADS);
    glVertex2f(-1.0, -1.0);
    glVertex2f(-1.0, 1.0);

    glVertex2f(1.0, 1.0);
    glVertex2f(1.0, -1.0);
glEnd();
```

Les différents paramètres utilisés dans le *fragment shader* sont définis à l'aide des fonctions *uniformLocation* et *setUniformValue*. Par exemple, pour mettre à jour la taille du widget :

```
position_location = program->uniformLocation("position");

QVector2D s = QVector2D(width, height);
program->setUniformValue(size_location, s);
```

Par contre, je n'ai pas réussit à utiliser les fonctions de benchmarks de Qt Test pour tester la rapidité du rendu par cette méthode. Pour pouvoir mesurer le temps, j'ai ajouté un calcul du nombre de FPS (frame par seconde) avec un simple compteur, incrémenté à chaque cycle et remis à zéro toutes les secondes :

```
int delay = m_time.msecsTo(QTime::currentTime());
if ( delay >= 1000 )
{
    if (m_fps > m_fps_max) m_fps_max = m_fps;
    setWindowTitle(QString("fps:%1 - max:%2 - w:%3 - h:%4")
        .arg(m_fps).arg(m_fps_max).arg(width()).arg(height()));

    m_fps = 0;
    m_time = QTime::currentTime();
}
else
{
    ++m_fps;
}
```

Pour mettre à jour la widget, j'utilise un *QTimer* avec un délai de 0.
ATTENTION : Il en faut pas laisser le programme tourner trop longtemps avec cette méthode. Le programme tourne en boucle et utilise la carte graphique au maximum de ses capacités, ce qui augmente progressivement sa température. Personnellement, je n'ai pas pris le risque de tester si cela pouvait endommager la carte graphique.

```
m_timer = new QTimer(this);
connect(m_timer, SIGNAL(timeout()), this, SLOT(timerUpdate()));
m_timer->start(20);

void GradientWidget::timerUpdate()
{
    updateGL();
}
```

Performances

J'ai réalisé le test sur un ordinateur portable équipé d'une carte Nvidia 8600M GT avec 512 Mo et un widget de 256 x 256 pixels. Avec cette configuration, les mesures donnent un FPS moyen de 5000 fps avec de pointes à 5200 fps, c'est à dire des temps de rendu de l'ordre de 0,2 millisecondes (soit 250 fois plus rapide que les premières versions).