Note d'application :

Lien entre le QML et le C++ pour réaliser le traitement d'image

I- Introduction :

Pour mon projet : «Automatisation du contrôle de serrage au couple sur un bloc de traction TGV » qui a été proposé par l'entreprise ALSTOM Tarbes et qui a pour but l'amélioration et le gain de temps lors de la vérification de leur bloc de traction TGV.

ALSTOM souhaite un système de vérification rapide, simple et efficace. Il a donc été proposé au client de développer une application pour appareil mobile (smartphone, tablette) qui serait utilisée par le contrôleur. Cette application permettra aussi d'archiver toutes les photos des vérifications effectuées sur un poste, ceci afin d'améliorer le suivi des contrôles des blocs de traction.

Pour réaliser cette application, nous avons décidé d'utiliser le logiciel QT qui, en plus d'être un logiciel gratuit, permet de réaliser des applications multiplateforme. Cette caractéristique a été imposée par notre client car dans l'usine de Tarbes, les tablettes utilisées sont sous IOS et les téléphones sont sous Android.

Au début de notre projet, nous avons été informés par le retour d'expérience de nos prédécesseurs, qui ont eu le même projet, que des problèmes pour l'utilisation de la camera en C++ (via la class QCamera) étaient reconnus.

Pour contourner ces problèmes, nous avons choisi de réaliser notre application via le langage QML via l'application QT Quick disponible sous le logiciel QT. Ce langage est un langage déclaratif permettant de réaliser les interfaces graphiques.

Avec ce langage de programmation, nous avons pu réaliser une grande partie de notre application, mais nous avons été confrontés à la limite de ce langage lorsque nous avons voulu réaliser le traitement d'image. Aucune fonction n'est définie pour le traitement des images en QML. Nous avons donc dû réaliser un lien entre le QML (l'interface graphique) et le C++, où nous allons implémenter la fonction pour réaliser le traitement d'image.

Je vais donc détailler dans ce document comment réaliser cette partie technique qu'est le lien entre le QML et le C++. Je vais par la suite détailler le programme que j'ai réalisé en C++ pour assurer la détection de la couleur prédéfinie des marquages qui doit être sur les boulons.

II- Réalisation d'un lien entre le QML et le C++

La première étape pour réaliser ce lien est d'ajouter au projet QT Quick (avec un main en QML) un fichier source C++, ainsi que le fichier header C++ associé.

Ces fichiers ont le même format que dans d'autres environnements de développement :

- Le header contient la déclaration des fonctions de la class
- Le code source contient l'implémentation des fonctions de la class

Cependant, il y a quelques changements sous QT, comme par exemple la déclaration d'une chaine de caractères en C++ qui peut se faire via la déclaration string. Ici cette déclaration doit être QString.

Le code de ces deux fichiers sera détaillé dans la partie III qui concerne le traitement d'image.

Une fois ces deux fichiers créés, il faut ensuite les ajouter à la configuration du projet pour qu'ils puissent être compilés. Cette étape est réalisée en modifiant le .pro. Dans ce fichier, un item SOURCES doit contenir uniquement le main.cpp, il faut alors lui ajouter votre source.cpp. Il en va de même pour l'item HEADERS en lui ajoutant votre header.h.

Squirrel.pro — Line: 1, Col: 1

```
TEMPLATE = app

QT += qml quick
CONFIG += c++11

SOURCES += main.cpp \
    detection.cpp \
    fileio.cpp

RESOURCES += qml.qrc

# Additional import path used to resolve QML modules in Qt Creator's code model
QML_IMPORT_PATH =

# Additional import path used to resolve QML modules just for Qt Quick Designer
QML_DESIGNER_IMPORT_PATH =

# The following define makes your compiler emit warnings if you use
# any feature of Qt which as been marked deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

HEADERS += \
    detection.h \
    fileio.h
```

Voici le .pro de notre projet. Comme vous pouvez le constater, en plus de la détection, nous avons également utilisé le lien QML C++ pour réaliser la lecture et l'écriture dans des fichiers tels que la base de donnés.

Il faut maintenant inclure le header dans le programme principal, i.e. le programme main.cpp. Pour cela, comme dans tout autre programme C++, on ajoute la ligne #include "detection.h".

Il faut également, dans ce fichier main.cpp, identifier les modules pour que nous puissions les utiliser dans le QML. Cette déclaration est réalisée via la fonction qmlRegisterType <nomDeL'Item, versionDeL'Item> ("nomDeDeclarationDeL'ImportEnQML", versionHaute, versionBasse, "nomDeDeclarationDuComposentEnQML").

Pour se simplifier la tâche, nous pouvons assigner le même nom à tous les éléments, comme vous le montre l'exemple présent dans notre projet.

main.cpp <Selectionner un symbole> Lines: 14, Col: 50

```cpp
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "fileio.h"
#include "detection.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    qmlRegisterType<FileIO, 1>("FileIO", 1, 0, "FileIO");
    qmlRegisterType<Detection, 1>("Detection", 1, 0, "Detection");

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;

    return app.exec();
}
```

Il reste maintenant à importer cet élément dans le fichier QML et utiliser les différentes fonctions que vous avez implémentées dans votre fichier C++. Pour ce faire, ajouter la ligne import nomDeDeclarationDeL'ImportEnQML versionHaute.versionBasse.

Ensuite, vous devez créer un composant comme n'importe quel autre composant en QML ayant le même nom que celui déclaré dans le main.cpp

```
nomDeDeclarationDuComposentEnQML {
    id: myID
}
```

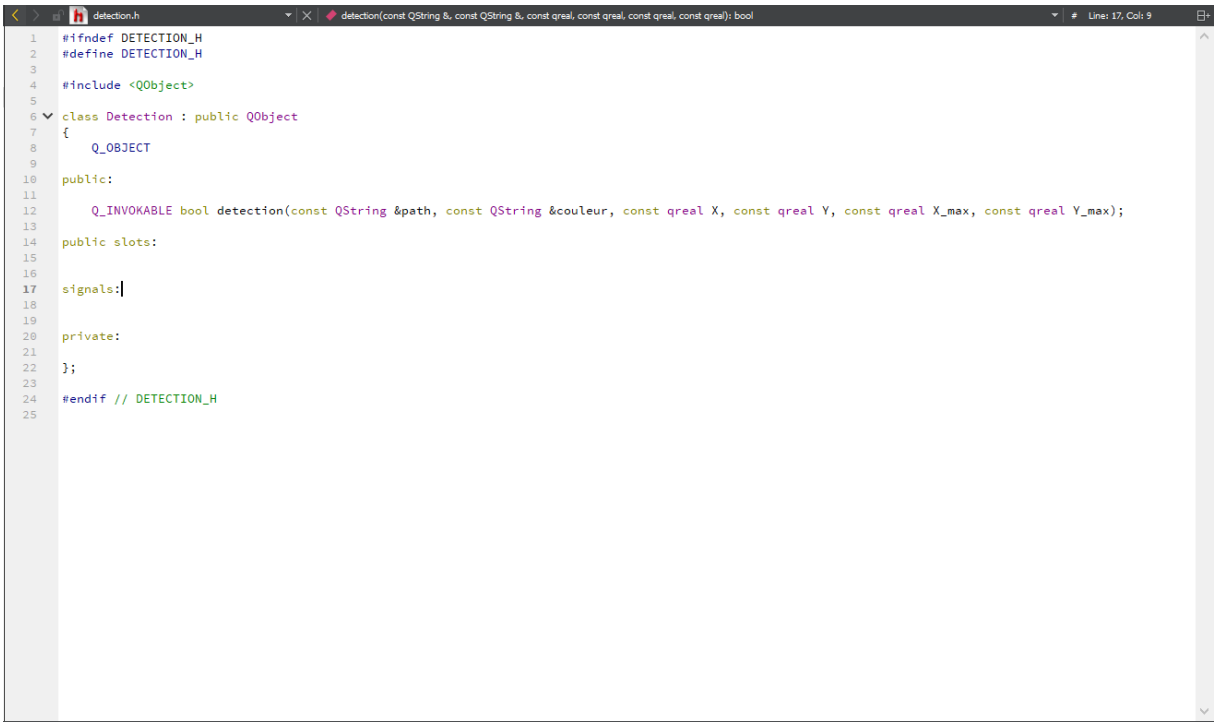
La dernière étape est d'invoquer la fonction avec les mêmes arguments que celle que vous avez implémentée en C++ sous cette forme :

idDuComposentDéclaré.nomDeLaFonctionEnC++(argumentsDeLaFonction)

Les fonctions implémentées en C++ sont maintenant liées et peuvent être exécutées à partir du QML. Je vais donc détailler, dans la troisième partie de cette note d'application, le code du fichier source.cpp et header.h que j'ai réalisé pour le traitement d'image.

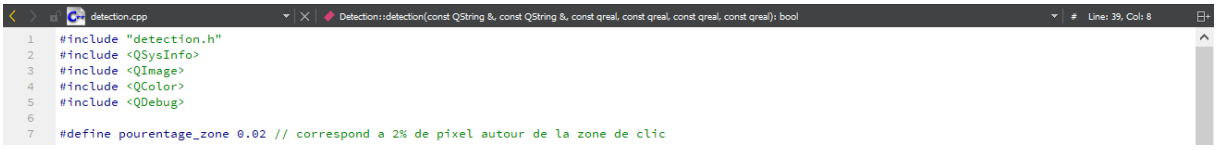
III- Etude du programme C++ pour le traitement d'image

Nous allons dans un premier temps étudier le programme du header. Celui-ci, comme vous pouvez le voir dans notre exemple ci-dessous, est identique à un header réalisé sous un autre IDE en C++, à l'exception du mot clef Q_INVOKABLE mis devant la déclaration de la fonction. Ce mot clef permet à cette fonction de pouvoir être appelée depuis le QML.



Pour ce qui est du fichier source.cpp, je vais détailler ligne par ligne les éléments qui composent ce fichier.

Tout d'abord, comme dans tout fichier C++, nous allons trouver les includes et les defines pour notre fichier.



On inclut donc les différentes class dont nous avons besoin. J'ai également définit une valeur qui représente le pourcentage de pixel à tester pour la détection des couleurs.

```
bool Detection::detection(const QString &path,const QString &couleur, const qreal xMouse, const qreal yMouse, const qreal xScreen, const qreal yScreen)
{
    QColor color;
    QImage image;
    QString my_path;
    unsigned short X;
    unsigned short Y;
    unsigned short DeltaPixel;
    int cpt = 1, cpt_tot = 1;

    if(QSysInfo::productType() == "windows") my_path = path.mid(8);
    else my_path = path.mid(7);
```

On fait ensuite la déclaration de notre fonction, ici pour la détection de la couleur du marquage.

Au début de cette fonction, on déclare tous les éléments qui vont nous servir dans cette fonction. On peut voir que des éléments spécifiques à QT peuvent être déclarés comme le QColor, si l'on a bien inclue la class QColor.

Le test suivant sert à supprimer les premiers éléments de la chaine de caractères du chemin où a été enregistrée la photo. En QML, il est sous forme : file:///path. Or, en C++, la forme est différente suivant le système d'exploitation, sous tablette : /path et sous window : path. C'est pourquoi, on regarde le système d'exploitation pour choisir le nombre de caractères à supprimer.

```
    if(image.load(my_path)){

        X = (image.width()*xMouse)/xScreen; //Conversion du point de clic en fonction de la taille de l'écran et de la photo
        Y = (image.height()*yMouse)/yScreen;

        if(image.width()>image.height())DeltaPixel = image.width()*pourentage_zone; // sélection de la zone à controler autour du point de touche
        else DeltaPixel = image.height()*pourentage_zone;
```

Dans cette partie de la fonction, je converti la valeur de la position du clic sur l'écran par rapport au nombre de pixels de la photo, qui a préalablement été chargée. Cette conversion permet donc de savoir le pixel où l'utilisateur a cliqué. On peut donc par la suite, chercher la couleur dans un pourcentage de pixel autour de cette zone. Ce pourcentage se calcule par rapport à la plus grande valeur entre la hauteur et la largeur.

```
        for(int width = (X-DeltaPixel); width<(X+DeltaPixel); width++){ //pour la taille de la zone en largeur
            for(int height = (Y-DeltaPixel); height < (Y+DeltaPixel); height ++){ // pour la taille de la zone en hauteur
                color = image.pixelColor(width, height);
                cpt_tot ++;
                if(color.value() > 50 && color.hsvSaturation() > 150){ //Pour éviter les couleurs noires/foncées
                    //suivant les couleurs en entrées, on valide la couleur via la teinte
                    if(couleur=="rouge"){
                        if(color.hsvHue()>330 || color.hsvHue() < 30)cpt++;
                    }
                    if(couleur=="orange"){
                        if(color.hsvHue()>0 && color.hsvHue() < 60)cpt++;
                    }
                    if(couleur=="jaune"){
                        if(color.hsvHue()>30 && color.hsvHue() < 90)cpt++;
                    }
                    if(couleur=="vert"){
                        if(color.hsvHue()>90 && color.hsvHue() < 140) cpt++;
                    }
                    if(couleur=="bleu"){
                        if(color.hsvHue()>150 && color.hsvHue() < 270) cpt++;
                    }
                }
            }
        }
```

Donc pour toutes les valeurs (hauteur et largeur), on récupère la valeur du pixel dans la variable couleur. Nous avons choisi d'utiliser la méthode HSV

pour détecter les couleurs. On réalise donc un seuillage sur la saturation, ainsi que sur la value pour éliminer toutes les valeurs noires et foncées. Ces valeurs ont été choisies par test sur différentes photos fournies par notre client.

Nous réalisons ensuite des tests sur la teinte du pixel par rapport à la couleur qui nous est fournie dans les arguments de la fonction. D'après une recherche rapide sur Wikipédia (lien), voici les teintes qui correspondent aux couleurs :

- 0° ou 360° : rouge ;
- 60° : jaune ;
- 120° : vert ;
- 180° : cyan ;
- 240° : bleu ;
- 300° : magenta.

Nous avons considéré qu'une valeur était jaune si elle était comprise entre 30° et 90°. Les valeurs ont été approuvées grâce aux photos des différentes couleurs fournies par notre client. Lorsque toutes les conditions sont remplies, on incrémente un compteur.

```
54          if (cpt > (cpt_tot/10)) return true; // si le nombre de pixel conforme > au nombre de pixel de la zone/10
55          else return false;
```

Il suffit ensuite de faire le ratio entre le compteur de tous les pixels vérifiés sur les pixels qui correspondent à la bonne couleur et si ce ratio est supérieur à 10, on considère que le marquage est conforme à la couleur demandée et la fonction retourne alors vrai.

```
56      }else {
57          qDebug() << "Unable to load the picture";
58       }
59       return false;
60  }
```

Cette dernière partie permet juste d'envoyer un message d'erreur si la fonction n'arrive pas à lire la photo.