POLYTECH®
CLERMONT-FERRAND

Rapport de projet

3ème année d'école d'ingénieur

Création d'une IHM en QT

Auteurs :
Thibault GLOAGUEN

Tuteurs ISIMA :
M. Jacques Laffont
M. Romuald Aufrère

3 Mars 2015

Remerciements

Je tiens à remercier mon tuteur M. Laffont pour l'aide qu'il m’a apportée tout au long de ce projet et pour le matériel prêté. Je voudrais aussi remercier M. Yannick Amblart et M. Nicolas Mendes pour leur aide dans la réalisation de ce projet.

Résumé

Aujourd'hui, la société ERDF tient à assurer la sécurité des ses ouvriers sur les chantiers et, pour ce faire, propose différentes solutions. Cette société a donc fait appel à l'école d'ingénieur Polytech Clermont Ferrand et à l'ISIMA afin de développer un programme permettant d'alerter les ouvriers lorsque quelqu'un pénètre dans une zone dangereuse. Cette zone dangereuse, délimitée au préalable, est filmée par une camera et le flux d'image est directement retransmis sur le moniteur de l'utilisateur.

L'utilisateur doit donc être capable, via cette application, de délimiter un contour manuellement sur la zone dangereuse, puis, être averti en temps réel quand quelqu'un franchit ce contour.

La partie algorithme de détection et traitement d'images étant le projet de deux élèves de Polytech Clermont Ferrand. Mon travail est de réaliser l'interface homme machine et donc, d'établir la connexion entre le moniteur de l'utilisateur (PC, tablette) et l'algorithme tournant sur le Raspberry Pi.

Pour mener à bien ma mission, j'ai choisi d'utiliser l'environnement QT car il est portable sur de nombreuses cibles, notamment les tablettes Android.

En me basant sur des exemples fournis avec la documentation, j'ai tout d'abord implémenté une zone de dessin afin que l'utilisateur puisse définir à la main les contours. J'ai ensuite créé un modèle Client Serveur afin de pouvoir transférer un flux d'images côté serveur et recevoir ce flux côté serveur tout en renvoyant en temps réel la position en 2D de la zone dangereuse délimitée. Enfin, il m'a fallu adapter ce modèle en utilisant le Serveur implémenté par les élèves de Polytech sur le Rasbperry Pi pour pouvoir dialoguer avec le programme sur tablette et le programme sur le Raspberry Pi. Le programme final doit me permettre de recevoir le flux vidéo envoyé par la Raspberry Pi sur ma tablette et, inversement, les positions de la zone délimitée doivent être renvoyées au Raspberry Pi pour être traitées.

Abstract

Creation of a graphic interface to detect and signal people in dangerous zone on construction sites

Key words : Interface, QT, graphic library, C++, network application

Today, the EDF Company wants to assure security for all its workers on construction sites. For that purpose, this company proposes further solutions. The project's goal is to alert workers when they enter a dangerous zone under video surveillance in the construction site. Two teams worked on this project. The first team from Polytech Clermont Ferrand had to develop an algorithm of detection which can detect if people enter in a defined area. *My work was to implement the interface between the user's monitor which can be a PC or a tablet and the algorithm which is running on the Raspberry Pi. The interface is able to transmit the video's flux from the Raspberry Pi into the user's monitor, then the user can manually define a dangerous area and finally the program sends back the coordinates of the dangerous zone information to the Raspberry Pi where the detection algorithm can treat it and detect if people enter that zone. To carry my goals, i used the QT environment because it allows you to develop graphic interface on devices like computer or android tablet. Basing on example from the QT documentation, I have created a drawing zone which record the 2 dimension position where the user click. Then, i had to create a client Server model to exchange data between the two devices.* The program is able now to watch the webcam content from the Raspberry Pi, draw the area on the tablet and send back the useful information to the algorithm running on the Raspberry Pi. This project allowed me to create a concrete application using C++ language and graphic library QT. I have now a better comprehension of the language and i know how to use a complex IDE like QT Creator to manage my project.

Table des matières

Table des figures

Introduction

Mon projet de troisième année consistait à réaliser une interface homme machine afin de pouvoir prévenir l'utilisateur que quelqu'un rentrait dans une zone dangereuse sous vidéo surveillance sur un chantier. Il s'est déroulé au sein de l'ISIMA (Institut Supérieur d'Informatique, de Modélisation et de leurs Applications) en partenariat avec l'école d'ingénieur Polytech Clermont-Ferrand et la société EDF. Ce projet, à terme, doit permettre d'afficher un flux vidéo provenant d'une webcam branchée sur un Raspberry Pi, de dessiner une zone de contrôle pour enfin renvoyer les coordonnées vers le Raspberry Pi afin qu'un algorithme de détection traite ces données. Il est à noter que la partie algorithme de détection n'est pas encore portable sur Raspberry Pi, mon projet n'incorporera donc pas l'algorithme de détection. Mon projet se déroule en quatre grandes phases. La première phase est l'installation et la prise en main de l'environnement de développement ainsi que les exemples fournis avec cet environnement. La deuxième phase est l'implémentation d'une fenêtre graphique de dessin. La troisième phase consiste à créer une application client server en local qui respecte le cahier des charges, à savoir, l'échange d'images dans un sens et les informations de coordonnées dans l'autre. Enfin, la dernière phase est l'adaptation de ce modèle client server avec le server sur le Raspberry Pi et porter l'application sur tablette Android.

Chapitre 1
Contexte du projet

1. Présentation du projet

De par l'importance d'ERDF, cette entreprise possède de nombreux sites. Certains d'entre eux ont parfois besoin de maintenance qui peut durer de deux mois à un an. Le personnel réalisant cette maintenance peut être exposé à des dangers dus aux installations électriques à proximité des chantiers. ERDF souhaite par conséquent protéger ce personnel d'un danger de mort. Le système à concevoir évoluera sur différents chantiers qui auront une surface moyenne de 5000 m². Les zones à protéger peuvent être multiples. Un balisage est déjà présent sur les chantiers, cependant ERDF souhaite renforcer un peu plus l'information de danger déjà présente, le système sera donc un complément du balisage.

<u>Le Balisage</u> :

Ce balisage définit les zones dangereuses à ne pas franchir sur un chantier.
Le personnel à tendance à franchir ce balisage afin de gagner du temps sur le chantier. Ce franchissement peut avoir de graves conséquences sur le personnel car il peut entraîner des blessures voir la mort des personnes étant donné qu'il y a présence de matériel sous tension. Cependant, une zone dangereuse peut aussi être une zone sans équipement sous tension mais qui comporte tout de même un risque de blessure voir de mort.

Sur la photo est représenté le balisage d'un chantier pour une zone dangereuse sous tension :

Figure 1 - Balisage d'une zone de danger

L'objectif de ce projet est donc d'avertir une personne qu'elle sort du balisage mais aussi d'avertir les autres membres du personnel présent sur le même chantier qu'une personne est potentiellement en danger.
Ce projet, en collaboration avec l'école d'ingénieur Polytech Clermont-Ferrand et l'ISIMA est scindé en deux parties :

- La première partie, gérée par Yannick Amblart et Nicolas Mendes, consiste à créer un algorithme permettant la détection d'une personne qui franchit cette zone
- La deuxième partie de ce projet, étant mon projet actuel, consiste donc à créer une interface Homme / Machine permettant le dialogue entre l'utilisateur et son moniteur et l'algorithme des deux élèves de Polytech embarqué sur une carte de type Raspberry Pi

2. Analyse des besoins

Objectif: Trouver un langage adapté à mon besoin et développer une interface simple d'utilisation

Pour la réalisation de ce projet, il m'a fallu choisir une bibliothèque graphique adaptée aux attentes de mon projet. La première problématique est la deployabilitée de ce langage. En effet, étant donné que ce programme doit être portable sur tablette Android, il est important de trouver une bibliothèque qui se porte facilement sur support Android. La deuxième problématique est la prise en main de cette bibliothèque. Elle doit être assez facile d'accès et la documentation doit être bien faite pour rapidement pouvoir développer mon application.
Après deux semaines de veille technologique et de comparaison sur les différentes bibliothèques graphiques, j'en suis venu à la conclusion que l'environnement de développement QT était le plus adapté à ma problématique. Il s'avère que l'environnement de développement permet d'intégrer de SDK Android et que l'application est portable sur tous systèmes Windows, Linux et Android. La documentation et les exemples étant conséquents et bien expliqués, j'ai décidé de choisir cette technologie pour commencer mon projet.

Cahier des charges de l'application :

- Fonction principale :

Retransmettre le flux vidéo de la zone de balisage et pouvoir dessiner les contours de cette zone manuellement

Envoyer les coordonnées du balisage tracé par l'utilisateur à l'algorithme de traitement

- Fonction secondaire

Affichage de la date

Création d'un bouton « reset » afin de pouvoir retracer le balisage en cas d'erreur.

3. Présentation de l’environnement QT

Qt est une bibliothèque multiplateforme pour créer des GUI (programme utilisant des fenêtres). Qt est écrite en C++ et elle est, à la base, conçue pour être utilisée en C++. Toutefois, il est aujourd'hui possible de l'utiliser avec d'autres langages comme Java, Python, etc.

QT comprend un grand nombre d'outils afin de pouvoir développer efficacement, le tout est de savoir choisir les outils qui correspondent aux besoins du projet.

Figure 2 - Le framework Qt

L’IDE QTCreator

Cet IDE comprend un débuggeur intégré et une documentation accessible en mode non connecté (ce qui est très pratique). Il est possible d'installer des Toolchains différentes suivant les architectures cible sur lesquelles on veut envoyer son programme. Utilisant un système d’exploitation Windows 8, je n'ai eu besoin d'installer que les outils pour développer sur Android à partir de

QTCreator. Dès lors, je pouvais développer et compiler mon application sur mon ordinateur pour tester, et en un clic, porter l'application sur ma tablette Android sans se soucier de la Cross Compilation.

Qt contient un outil vraiment très efficace pour la création de fenêtre graphique : QtDesign. En effet, cet outil permet de venir créer des fenêtres avec une interface graphique assez simple d’utilisation. J’ai d’ailleurs utilisé cet outil pour le design de mon application au départ.

Chapitre 2
La fenêtre de dessin

1. Architecture de l 'application

Cette application se décompose en plusieurs widgets qui se superposent. Il y a donc une hiérarchie entre les fenêtres.

Nous avons tout d'abord :

- La fenêtre principale *Receiver*

La classe *Receiver* est la classe principale de l'application, elle hérite de la classe *QWidget*.

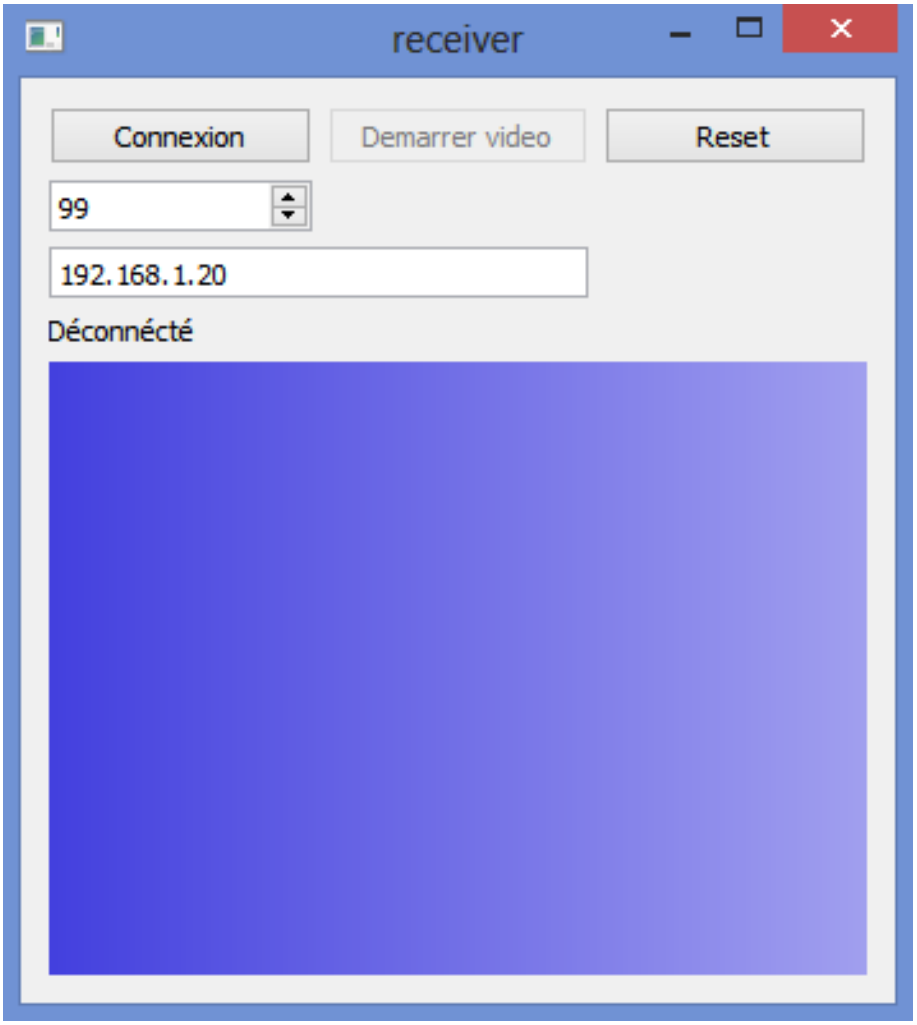


Figure 3 - La fenêtre Receiver

Cette fenêtre se compose de :

- D'un *QPushButton* Connexion qui permet de se connecter au serveur distant qui envoie les informations
- un *QPushButton* Démarrer video qui permet d'afficher le flux video dans la fenêtre prévue à cet effet
- un *QPushButton* reset qui permet d'effacer le dessin sur la fenêtre de dessin et de recommencer
- un *QSpinBox* qui permet de renseigner le port sur lequel on va transférer les données
- Un *QLineEdit* qui permet de renseigner l'adresse IP du serveur sur lequel on veut se connecter
- Un objet de type *MaFenetre*, cet objet est le carré bleu représenté en figure 3, c'est cet objet qui va afficher le flux vidéo et me permettre de dessiner par-dessus
- Enfin, un *QLabel* qui affiche les états du logiciel (déconnecté, connecté, réception d'image, etc …)

Cette classe est aussi en charge de la gestion de connexion avec le serveur, cet aspect sera plus détaillé dans le chapitre 4.

- Nous avons ensuite la fenêtre de type MaFenetre

Cette fenêtre est issue de la classe MaFenetre qui hérite de la classe QLabel. Cette classe est un attribut de la classe principale Receiver. Elle a pour but d'afficher les QImages reçues de la classe Receiver et de permettre de dessiner sur ces images.

Cette fenêtre se compose d'un :

- D'un QLabel qui permet d'afficher la vidéo
- D'un QPainter, le QPainter est un objet de la librairie QT qui permet de rendre une zone déssinable, je détaillerai ce passage dans le paragraphe suivant

2. Définition de la zone de dessin

Afin de remplir le cahier des charges de mon application, j'ai du créer une zone de dessin afin que l'utilisateur du programme puisse définir sa zone de balisage. Pour cela, j'ai donc créé une classe héritant de la classe QLabel. Cet héritage m'a permit d'utiliser des fonctionnalités de la classe QLabel comme le fait de pouvoir dimensionner la taille et colorer le fond de mon objet avec les instructions :

```
setFixedSize(320,240);
 setStyleSheet("QLabel { background-color: yellow }");
```

➢ Affichage d'une image en fond

Afin de pouvoir afficher une image en fond de cette fenêtre et de pouvoir dessiner par-dessus j'ai utilisé un QPainter. En effet, cet objet Qt me permet d'affecter une image et de la rendre déssinable.

```
QPainter p (this);
QRect dirtyRect = event->rect();
 p.drawImage(dirtyRect,*img, dirtyRect);
```

On peut voir ici dans la création d'une QPainter on lui affecte en parent this qui correspond au pointeur sur la fenêtre courante MaFenetre.
La méthode drawImage() du QPainter prend en paramètre une QImage et l'affiche.

➢ La zone de dessin

Pour pouvoir dessiner sur mon image j'ai utilisé le tracking de la souris et les événements des clicks gauche et droit.

Pour simplifier la zone de dessin, j'ai décidé qu'une zone de balisage pouvait être définie avec 4 points et donc, géométriquement, un quadrilatère.

L'algorithme de principe est le suivant :

Des que l'utilisateur click, on commence à tracer une ligne. Avec le click suivant, on peut finir la ligne et commencer à tracer une nouvelle ligne

directement. On finit le tracé au bout du troisième click et le quadrilatère se complète automatiquement.

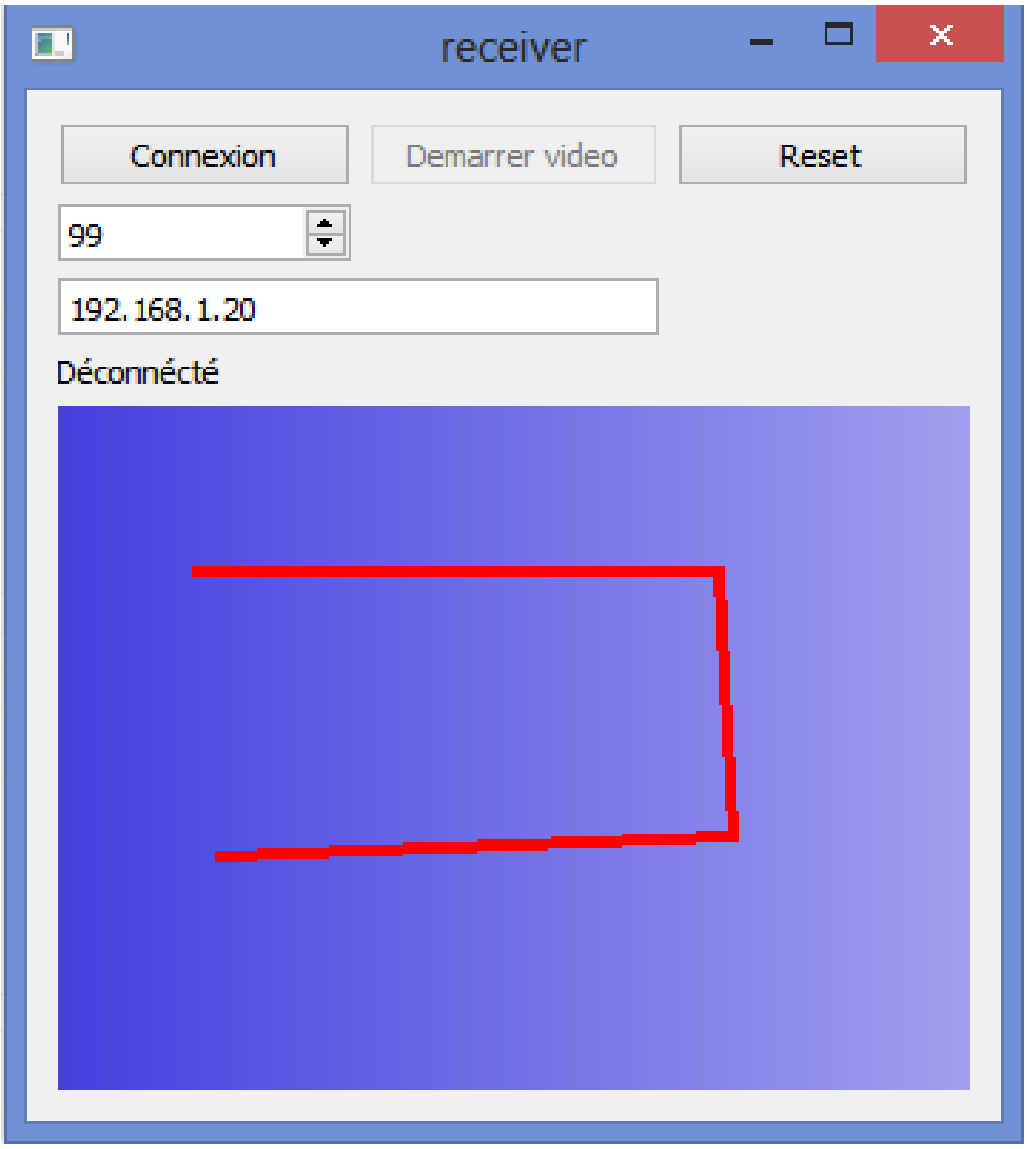


Figure 4 - dessin sur la fenêtre

Pour la récupération des coordonnées du tracé, j'utilise des QPoint. Ces objets me permette d'accéder directement aux coordonnées en X et en Y du point clické. Cela me permet aussi de pouvoir tracer une Qline. J'ai donc deux Qpoint, à savoir un startPos et un endPos afin de pouvoir tracer une ligne entre les deux

```
QLine line = QLine(startPos,endPos);
```

Et ensuite pour tracer la ligne on utilise un QPaintEvent. Un QPaintEvent est une méthode qui est appelée à chaque fois que l'on doit dessiner quelque chose. Il est donc important de redéfinir cette méthode ainsi

```
void MaFenetre::paintEvent(QPaintEvent *event)
{
    QPainter p (this);
    QRect dirtyRect = event->rect();
    p.drawImage(dirtyRect,*img, dirtyRect);
    QPen pen;
    pen.setColor(Qt::red);
    pen.setWidth(7);
    p.setPen(pen);
    drawLines(&p);
}
```

A l'intérieur de cette méthode, on déclare donc un QPainter qu'on affecte à notre fenêtre de dessin. On définit la zone et l'image de fond avec les lignes

```
    QRect dirtyRect = event->rect();
    p.drawImage(dirtyRect,*img, dirtyRect);
```

On définit l'outil pour dessiner (sa taille, sa couleur) et enfin on dessine la ligne avec :

```
    drawLines(&p);
```

Afin de pouvoir utiliser la souris comme périphérique pour dessiner, il est important t'utiliser la gestion d'événement de la souris que Qt propose.

Pour utiliser les mouvements de la souris, il faut implémenter la méthode

```
void MaFenetre::mouseMoveEvent(QMouseEvent
*event)
{

    if (inDrawing )
    {
        endPos = event->pos();
        update();
    }
}
```

Cette méthode permet de venir renseigner la position du curseur en temps réel tant que l'on dessine.

```
void MaFenetre::mousePressEvent(QMouseEvent *event)
```

La méthode mousePressEvent doit être implémentée aussi si on désire utiliser le click dans notre programme en utilisant la condition :

```
if (event->buttons())
{
   //traitement si on click avec la souris
}
```

➢ L'enregistrement des coordonnées

Il était important, pour suivre le cahier des charges, de trouver un moyen de stocker les coordonnées de mon quadrilatère tracé pour les envoyer plus tard à l'algorithme de traitement. Pour cela, j'ai utilisé un Qvector de Qpoint. C'est donc un vecteur des coordonnées. Le principe est donc qu'à chaque fois que je click (droit ou gauche)

```
endPos = event->pos();
points.append(endPos);
emit pointPret();
```

On vient donc récupérer la position du point courant dans endPos puis avec la méthode *append()* on vient enregistrer ce point dans le vecteur de Qpoint. La dernière ligne permet d'envoyer le signal pointPret() à la méthode d'envoi de données comme ceci :

```
connect(drawFenetre, SIGNAL(pointPret()), this,
SLOT(envoyerMessage()));
```

C'est une particularité de Qt, il est possible d'envoyer des signaux entre les méthodes comme ici. A chaque fois qu'un point sera enregistré, un signal sera envoyé à la méthode envoyerMessage() lui disant qu'il peut lire dans le vecteur de Qpoint car il vient d'être rempli et donc il peut envoyer la donnée sur le réseau.

Chapitre 3
Communication IHM/ Carte d'acquisition

1. Mise en place d'un réseau

Afin de simuler un cas d'utilisation de mon application, j'ai du mettre en place un réseau pour pouvoir établir une connexion entre mon ordinateur ou tourne l'application client et le Raspberry Pi ou tourne le serveur d'envoi de données vidéo

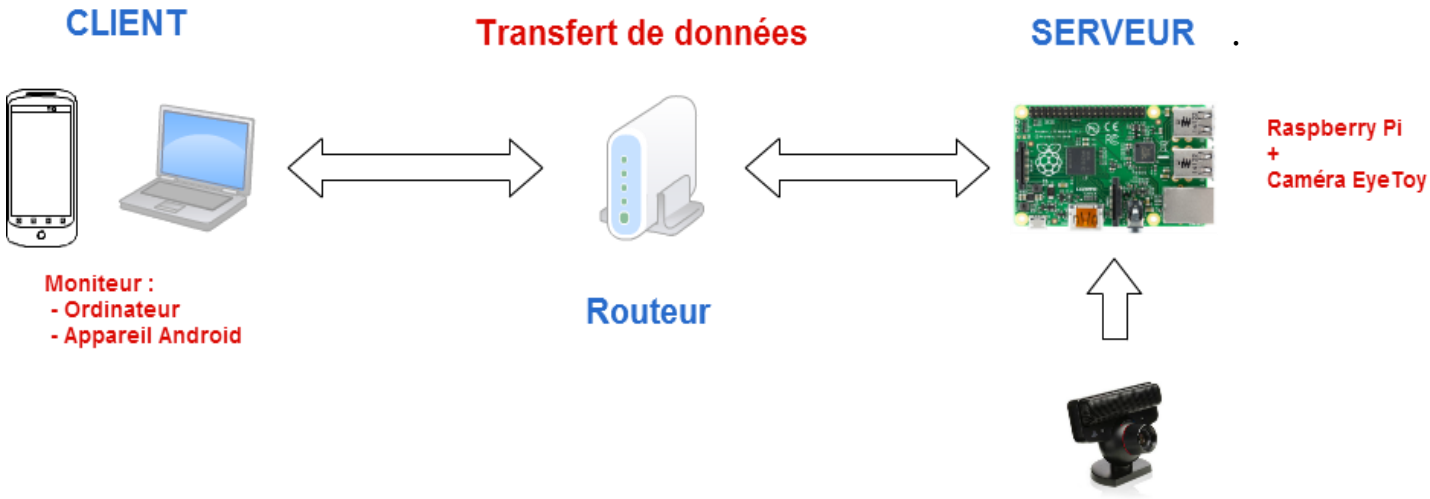


Figure 5 - Schéma du réseau

Le Raspberry Pi étant équipé d'un serveur DHCP l'établissement du réseau c'est fait assez facilement.

Le Raspberry Pi en local obtient l'adresse IP : 192.168.1.20 et le transfert de données s'effectue sur le port 55555

2. Création d'un serveur

1. Principe de l'application

Cette application sur le Raspberry Pi à pour but de faire l'acquisition du flux video issue de la camera image par image, et d'ensuite envoyer image par image les données à travers le réseau.

Pour ce faire, j'utilise l'algorithme de principe suivant :

Tout d'abord on crée l'objet qui va contenir l'image

On fait l'acquisition de la première image. On renseigne l'objet sur toute les caractéristiques de l'image (taille, largeur, hauteur) une seule fois, car elles seront constantes durant l'acquisition.

On prépare le serveur et on vient écouter sur le port pré définis pour identifier un client qui se connecte.

Un client se connecte, on envoie une première trame contenant les données caractéristique de l'image (taille, largeur, hauteur)

On se met en attente que le programme client acquitte de la bonne réception de ces données

Après acquittement on fait l'acquisition de l'image suivante et on envoie uniquement les données pixels de l'image en boucle tant qu'on ne demande pas au serveur d'arrêter.

2. Envoi et réception de données avec une socket

Tout d'abord l'acquisition des données video se fait grâce à la bibliothèque OpenCV qui à été installée au préalable sur le Raspberry Pi.

```
IplImage* frame;

frame = cvCreateImage ();
```

L'objet IplImage est celui qui va contenir l'image.

```
SOCKADDR_IN to = { 0 };
int tosize = sizeof to;

to.sin_addr.s_addr = htonl(INADDR_ANY);
to.sin_port = htons(PORT);
to.sin_family = AF_INET;

//on met les infos du serveur dans la socket
bind(sock, (struct sockaddr*)&to, sizeof(to));
listen(sock, 10);
```

Ici, on vient renseigner la structure to avec les infos du réseau, à savoir, le port, l'adresse IP et la famille de protocole utilisée. Puis avec la fonction bind() on prépare la socket pour l'envoi. Enfin on attend qu'un client se connecte avec la fonction listen.

```
CvCapture *capture =
cvCreateCameraCapture(CV_CAP_ANY);

if (!capture)    std::cout<<"Erreur Capture
video"<<std::endl;

frame=cvQueryFrame( capture );
```

Ici, on utilise les fonctions d'openCV pour capturer le flux vidéo dans l'objet CvCapture puis on récupère la première image avec la fonction cvQueryFrame

```
int longueur = frame->imageSize+sizeof(IplImage);
wait=accept(sock, (struct sockaddr*)NULL, NULL);
write(wait,&longueur,sizeof(longueur));
write(wait,frame,sizeof(IplImage));
write(wait,frame->imageData,longueur);
wait=accept(sock, (struct sockaddr*)NULL, NULL);
longueur=longueur-sizeof(IplImage);
```

Dans cette partie, tout d'abord on prépare la longueur du message que l'on va envoyer dans la socket. C'est donc la taille des données de l'image + l'entête de l'objet contenant les caractéristiques de l'objet.

La fonction accept est bloquante pour le programme, elle attend que le client ait accepté la connexion avec le serveur. On crée alors une nouvelle socket, la socket wait et on envoie la première trame avec la fonction write.

- Le premier message contient donc la taille du message à venir afin de préparer coté client le buffer pour l'image.
- Le deuxième message contient l'entête de l'objet avec la taille, la longueur et la largeur de l'image qui seront renseignées au client une unique fois
- Le troisième message contient l'imageData, donc les informations pixel de la première image capturée

Ensuite on se remet en mode attente afin que le client acquitte des premières informations puis on redéfinit la longueur du message à envoyer. En effet, on doit soustraite la taille de l'objet IplImage car on n'a plus qu'à envoyer l'imageData en continue, image par image.

```
while(1)
{
    int n;
    frame = cvQueryFrame( capture );
    write(wait,frame->imageData,longueur);
    printf("Attention retour n° image\n");
    read(wait,&n,sizeof(n));
    printf ("  %d |",n);
    l=cvWaitKey(1);
    if (l=='q')
        break;
}
```

Des lors que le client acquitte, on rentre dans le while(1) et on fait l'acquisition de la nouvelle image puis on envoie les datas directement avec la socket wait. Enfin on vient lire l'acquittement du client qui renvoie le numéro de l'image reçue. On fait cette opération en boucle afin d'avoir un flux d'information constant. La touche q permet de sortir de cette boucle pour arrêter l'application.

3. Création d'un client

1. Principe de l'application

Cette application doit donc être en charge de récupérer le flux vidéo image par image et de l'afficher pour reconstituer la video sur le moniteur (soit la tablette soit l'écran d'ordinateur).Il doit notamment être capable d'envoyer les informations de coordonnées du quadrilatère qu'il trace sur la vidéo au serveur. Pour effectuer ces actions il faut suivre un certain protocole. Je vais donc expliquer ici l'algorithme de principe dans un premier temps.

- Réception des données

Pour gérer l'arrivée de paquet réseau, coté Qt on va implémenter tout d'abord une application client qui vient se connecter au serveur distant

Dés lors que l'on renseigne l'adresse du Serveur et le port utlisé, on click sur connexion et la connexion client/serveur s'établie.

Le client reçoit d'abord un premier paquet contenant les caractéristiques de l'image (taille, largeur, longueur) et fait l'allocation nécessaire pour afficher les images à venir.

Le serveur se déconnecte le temps que le client acquitte de la bonne réception des informations et redemande la connexion afin de recevoir le flux video en continu.

Le client click sur le bouton Démarrer video et récupère les données pixel dans un buffer qu'il traduit en QImage et affiche sur le moniteur.

- Envoi des données

L'envoi des données ne se produit que lorsque le quadrilatère à ses 4 côtés.

Dés lors qu'on reçoit la video, on commence à dessiner la zone de balisage.

Tant qu'on n'a pas 4 Qpoints on ne fait rien, Dès lors que les 4 Qpoints sont définis, on envoie un signal à la méthode EnvoyerMessage qui se charge d'envoyer les coordonnées sur le réseau au Raspberry Pi.

2. Réception et envoi des données

Tout d'abord, pour faire l'acquisition des messages sur le réseau, il faut se connecter au serveur qui envoie les messages. On utilise le bouton Connexion pour se connecter au serveur :

```
connect(connexion, SIGNAL(clicked()), this, SLOT(on_boutonConnexion_clicked()));
```

La méthode ici connect s'assure que si on click sur le bouton connexion, on accède à la méthode on_bouton_clicked() .

```
void Receiver::on_boutonConnexion_clicked()
{
 connexion->setEnabled(false);
 socket->abort(); // On désactive les connexions
 précédentes s'il y en a
 socket->connectToHost(ip->text() port->value());
  // On se connecte au serveur demandé
}
```

Cette méthode vient donc rendre inclickable le bouton connexion en le grisant dans un premier temps, pour ne pas essayer de se connecter plusieurs fois au même réseau.

On s'assure que la socket que l'on va utiliser pour le client n'est pas déjà utilisée puis on vient se connecter au serveur avec la méthode connectToHost() avec l'IP et le port renseignés.

Dés lors qu'on est connecté au serveur, on peut s'attendre à recevoir le premier paquet. Celui-ci est annoncé grâce au signal de la socket readyRead()

```
connect(socket,SIGNAL(readyRead()),this,SLOT(donneesRecues()))
```

Ici, le signal readyRead() est émit des qu'un paquet est prêt à être lu. On le connecte donc à la méthode `donneesRecues()` du programme en charge de réceptionner les paquets.

```
void Receiver::donneesRecues()
{
    etat->setText("Reception en cours...");
    QDataStream in(socket);
    // on inverse l'ordre des bytes reçu ici
    in.setByteOrder(QDataStream::LittleEndian);
```

Tout d'abord, j'utilise un QDataStream pour lire les messages reçus. Cet objet qu'on affecte à la socket, permet, par ces méthodes, de lire ou d'envoyer des trames réseaux de manière très simple.

Ici, avec la méthode setByteOrder, j'indique dans quel sens la suite d'octets du message doit être lue. Je choisi ici le `LittleEndian` car en testant mon programme, je me suis aperçu que le serveur en C++ utilisant des sockets classiques envoyait des messages en `LittleEndian` tandis que , côté client en Qt, les QSocket lisaient les messages en `BigEndian`.

```
if(!connected)
    {
        connected = true;
      // Premier traitement
   }

Else{

// Deuxième traitement

}
```

Il faut savoir que la méthode donnéesRecue() est utilisée pour deux fonctions différentes :

- la première consiste à récupérer la taille de message et les caractéristiques de l'image
- la deuxième consiste à récupérer le flux de données image uniquement sur le réseau

C'est pourquoi j'utilise le booléen `connected` pour séparer ces deux parties.

```
if (tailleMessage == 0) // Si on ne connaît pas
encore la taille du message, on essaie de la
récupérer
{
 if (socket->bytesAvailable() < (int)sizeof(quint32))
// On n'a pas reçu la taille du message en entier
      return;
 in >> tailleMessage; // Si on a reçu la taille du
message en entier, on la récupère
}
```

Tout d'abord, sachant que le premier message réseau sera la taille de l'image, on test donc si les bytes disponibles sont inferieurs à un quint32. Si ce n'est pas le cas, c'est qu'on à pas reçu la taille de message en entier donc on reboucle, sinon on vient stocker cette taille en utilisant le QDatastream et sa surcharge de l'operateur >> qui permet directement de lire le message et de l'insérer dans la variable tailleMessage.

```
temp = new char[sizeof(IplImage)] ;
temp2 = new char[tailleMessage-sizeof(IplImage)];
in.readRawData(temp,sizeof(IplImage));
in.readRawData(temp2,tailleMessage-sizeof(IplImage));
newImage = (IplImage*) temp;
tailleMessage=tailleMessage-sizeof(IplImage);
etat->setText("Image reçue");
afficher->setEnabled(true);
```

On alloue donc deux buffers, le premier va servir à stocker l'objet IplImage qui est envoyé sur le réseau. Le deuxième, lui, est destiné à recevoir le tableau de pixels de l'image.
Dès qu'on a lu les deux messages de la socket avec la méthode readRawData(), on vient faire pointer le pointeur de newImage (qui est une IplImage) sur le buffer et enfin, on modifie la tailleMessage qui ne doit contenir que la taille du tableau de pixel une fois que les caractéristiques de l'image sont enregistrées dans newImage.

```
else
{
   if (socket->bytesAvailable() < tailleMessage)
   { // Si on n'a pas encore tout reçu, on arrête la
méthode
     return;
   }
   in.readRawData(temp2,tailleMessage);
   newImage->imageData = temp2;
   image = Ipl2QImage(newImage);
   drawFenetre->setImage(image);
   etat->setText("Image reçue");
 }
```

Une fois que les attributs caractéristiques sont passés, on s'occupe du tableau de pixel de taille fixe connu.

Si les bytes disponibles ne correspondent pas encore à la taille du message attendu, on reboucle jusqu'à ce que le message soit complet. Alors, on vient lire sur la socket le tableau entier de pixel qu'on stock dans temp2 et on fait pointer le champ imageData de notre newImage sur ce buffer d'octets.

Il s'agit ici d'une reconstruction de l'objet IplImage du côté client. Le problème est que Qt ne traite que des Qimages qui sont propre à sa bibliothèque et dans un souci de portage sur tablette, je ne pouvais pas me permettre d'importer aussi la librairie OpenCV. J'ai donc implémenté une méthode Ipl2Qimage() qui

demande une IplImage en argument et renvoi un objet Qimage correspondant.

Enfin, avec la méthode setImage sur la fenêtre de dessin, on vient changer l'image de fond.

- Envoi des coordonnés de dessin vers le serveur

Pour l'envoi des coordonnées sur le réseau j'ai du utiliser les signaux Qt pour émettre un signal dès qu'on a les 4 Qpoints préts.

```
if (event->buttons() && points.size() < 3  )
    {
        // si  on à pas encore les points nécessaires
    }
else
{
    emit pointPret();
}
```

Dans la méthode MousePressedEvent, j'ai inséré une condition afin d'émettre le signal pointPret() à l'aide du mot clé emit dès que les QPoints étaient prêts à être envoyés. Alors, avec le mot clé connect,

```
connect(drawFenetre,SIGNAL(pointPret()),this,
SLOT(envoyerMessage()));
```

je viens connecter le signal PointPret() à ma méthode d'envoi envoyerMessage(). La méthode sera donc appelée dès que le signal sera émis.

```
void Receiver::envoyerMessage()
{
    QVector<QPoint> data = drawFenetre->getPoints();
    QString messageAEnvoyer;
    for (int j = 0; j < data.size(); ++j)
    {
         messageAEnvoyer += " " +
        QString::number(data[j].x())
       + " "  + QString::number(data[j].y()) + "|" ;
    }

    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);
    out << messageAEnvoyer;
    socket->write(paquet); // On envoie le paquet
}
```

La méthode d'envoi des coordonnées est assez simple, dans un premier temps on récupère le vecteur de Qpoint de notre fenêtre de dessin, puis on vient concaténer dans une chaîne de caractères les Qpoints. Enfin, on déclare un QByteArray et un QDatastream afin de sérialiser mon message pour l'envoyer sur la socket. De cette manière on envoie bien les coordonnées dans un tableau de caractères vers le serveur.

4. Portage sur Tablette Android

Le portage sur tablette c'est effectué très facilement. En effet, la dernière version de Qt permettant un portage complet sur surface Android, j'ai pu en quelques clicks, porter mon application sans problème sur une tablette.

Pour ce faire, j'ai du installer le SDK Android et le relier à Qt dans les options.

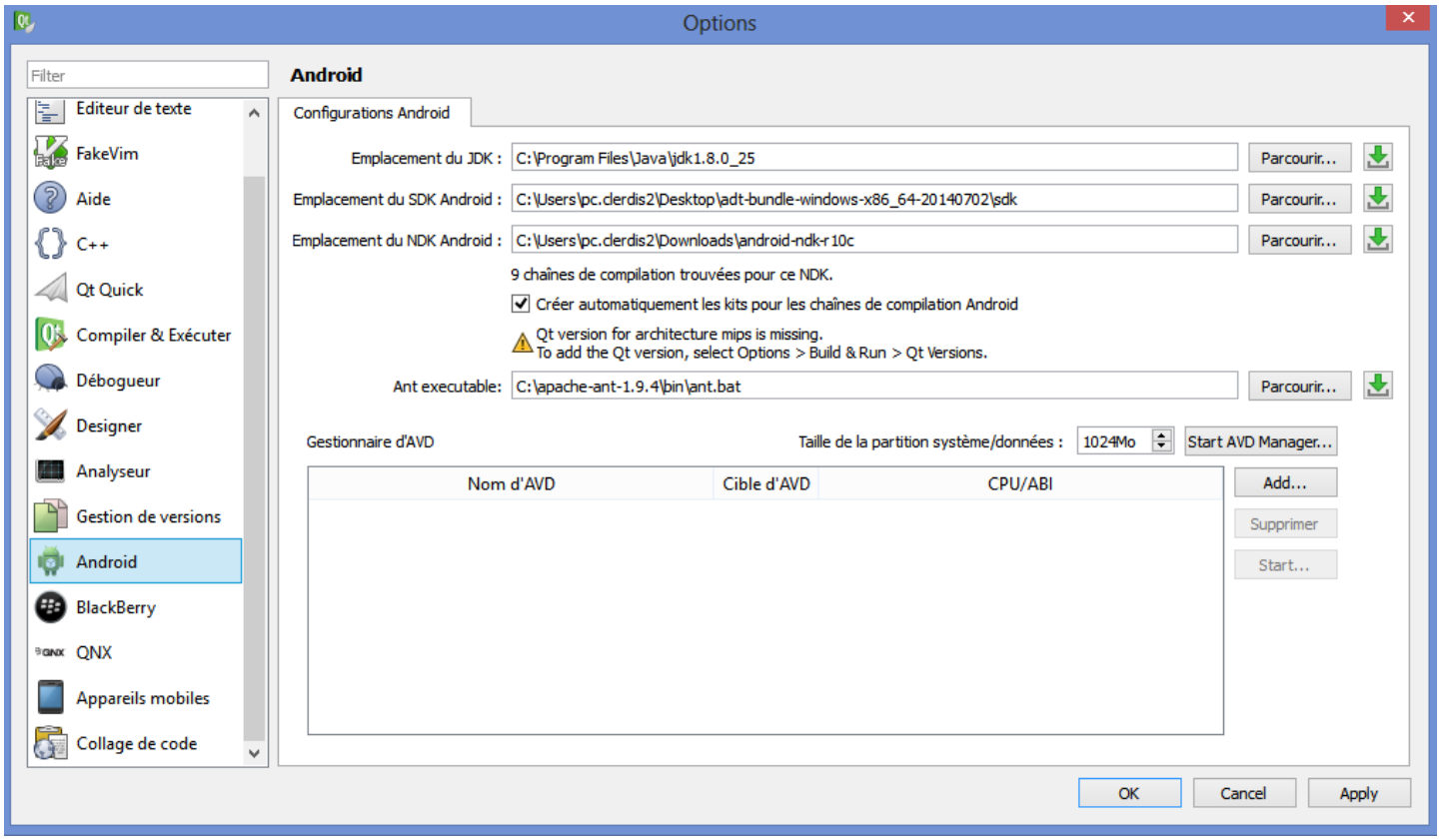


Figure 6 - Android et Qt

Après avoir installé le SDK et le NDK d'Android, on spécifie les chemins d'accès.

Ensuite on précise le bon compilateur pour le projet que l'on veut porter sur la tablette.

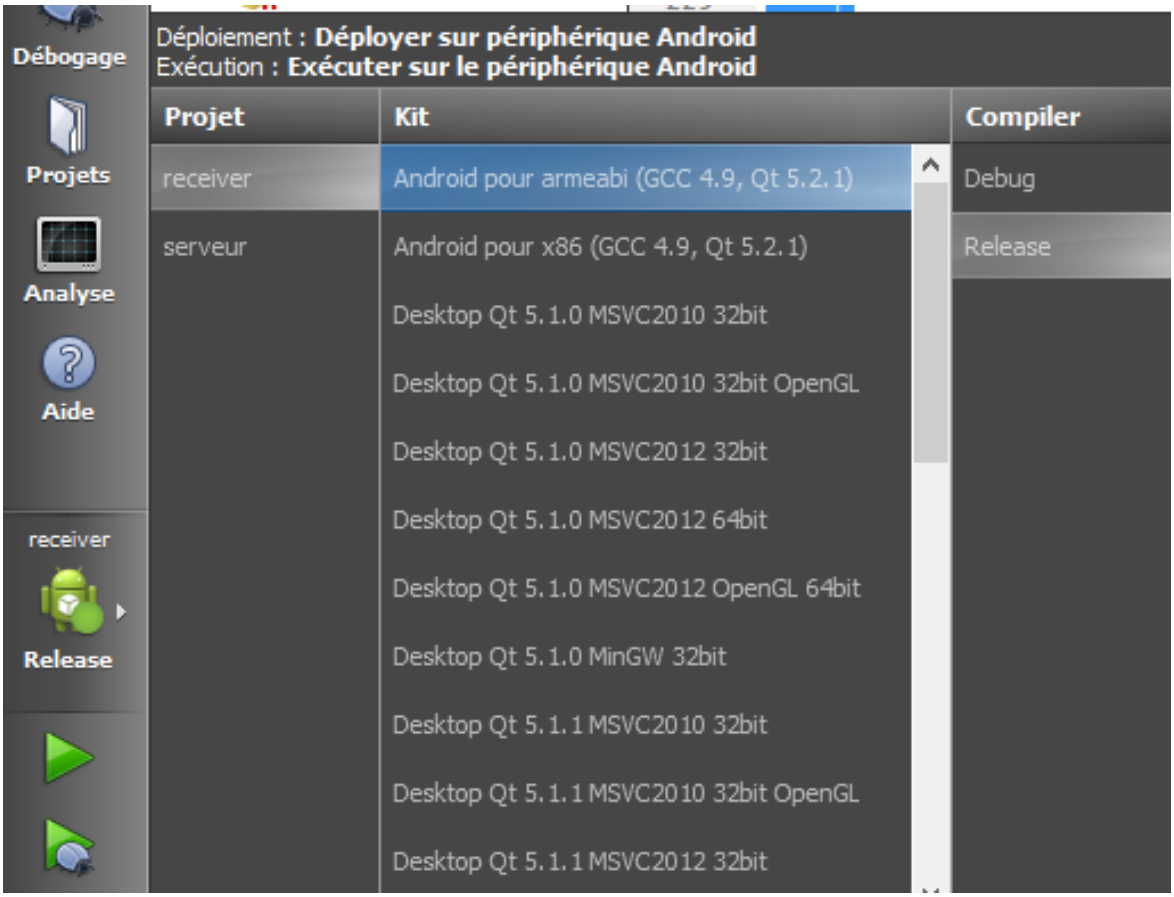


Figure 7 - sélection du compilateur

Le projet est donc déployé sur tablette et QT s'occupe lui-même de la cross compilation.

Conclusion

L'objectif de ce projet était la réalisation d'une interface de contrôle pour l'utilisateur afin de pouvoir visualiser la vidéo du chantier sous surveillance sur sa tablette et de délimiter à la main une zone de balisage sur la vidéo. Ce projet s'est réparti en 4 grandes phases :

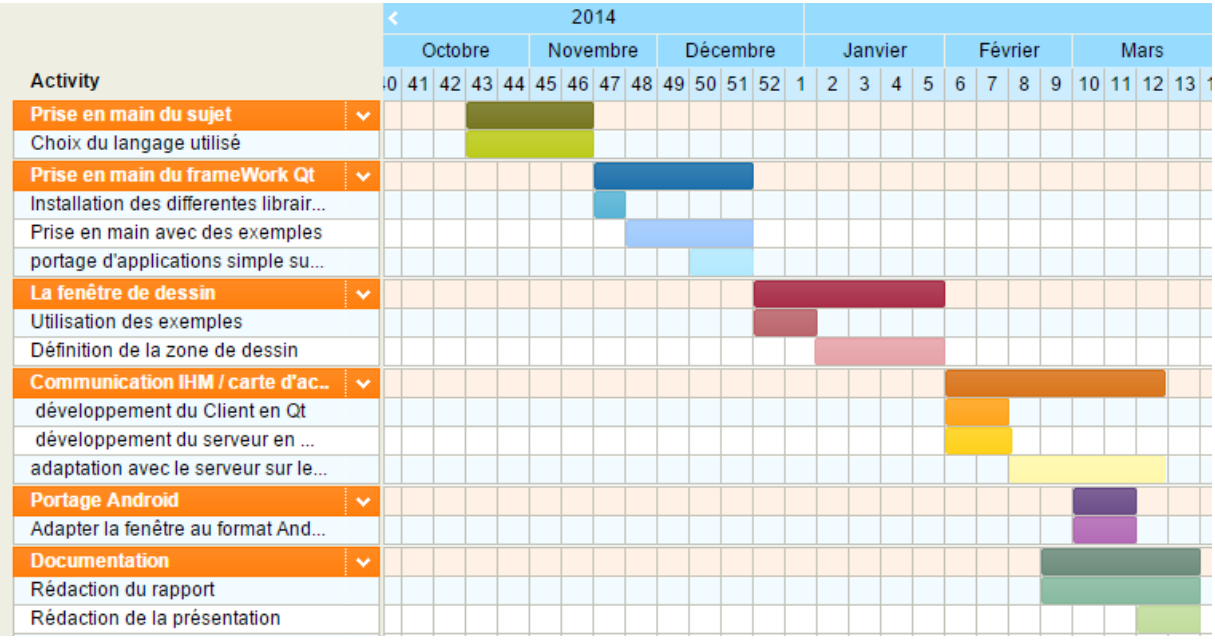
- La recherche du langage adéquat pour développer l'application
- La prise en main de l'environnement Qt
- Le développement de la fenêtre de dessin
- Le développement du client / serveur entre les deux programmes

J'ai principalement rencontré des difficultés dans l'établissement de la connexion entre un programme C++ serveur et l'application Qt client. En effet, la gestion des sockets était sensiblement différente, ce qui m'a valu quelques heures de recherche pour trouver la solution et faire fonctionner l'application. J'ai notamment rencontré des problèmes dans la gestion des paquets sur le réseau et leur taille. En effet, il m'a fallu un certain temps avant de trouver les bonnes tailles pour chaque paquet afin de lire les bonnes informations. Cependant, cela m'a permis de maitriser mon environnement de développement et plus spécifiquement le débuggeur de QtCreator. Je suis maintenant capable de debugger rapidement mon programme à l'aide des différents outils que Qt propose. Il est désormais possible avec l'application de recevoir le flux vidéo provenant du Raspberry Pi, de dessiner sur ce flux un quadrilatère définissant la zone à baliser et de renvoyer ces informations au serveur. Il est à noter que, pour l'instant, le serveur ne fait rien de ces informations car l'algorithme de détection développé par les élèves de Polytech sur ce projet est terminé mais pas encore portable sur Raspberry Pi, en effet, la puissance de calcul demandé pour un tel algorithme est encore trop importante. Cela pourrait d'ailleurs faire le sujet d'un projet pour l'année prochaine : optimiser cet algorithme de traitement d'image pour qu'il puisse être portable sur le Raspberry Pi et connecter l'IHM que j'ai développé avec ce programme. Pour finir, ce projet m'a vraiment beaucoup apporté en connaissance sur le framework Qt ainsi que sur la programmation orientée objet. Ayant réalisé cette interface seul, j'ai eu l'impression d'avoir plus de travail et de difficultés à remplir mes objectifs, mais finalement j'ai su remplir ma mission à temps.

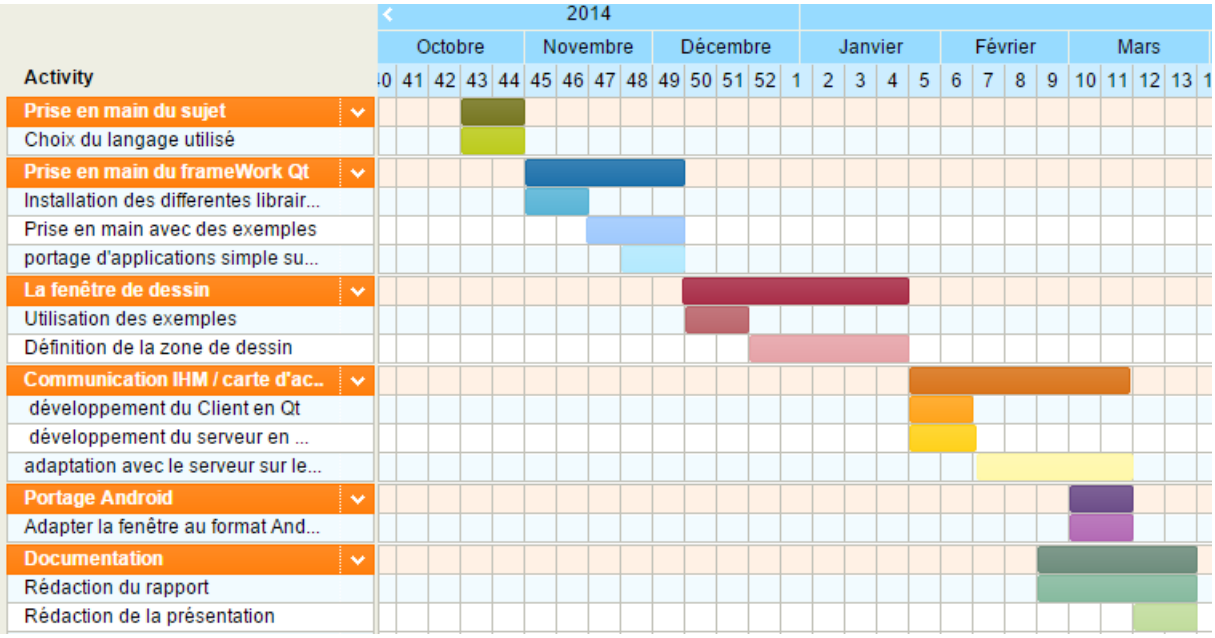
Annexe A

Diagramme de Gantt

- Diagramme prévisionnel



- Diagramme réel



Bibliographie/Webographie

Documentation Qt, http://doc.qt.io/

Documentation Raspberry Pi, http://www.raspberrypi.org/documentation/

Exemples d'IHM qt, http://dirac.epucfe.eu/projets/wakka.php?wiki=P11A09