Université du Québec
École de technologie supérieure

RAPPORT TECHNIQUE
PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
DANS LE CADRE DU COURS GTI792
PROJET DE FIN D'ÉTUDES EN GÉNIE DES TI

MODULE DE CONTRÔLE MANUEL D'UNE TOURELLE-SENTINELLE DE PAINTBALL

DOMINIQUE JACQUES-BRISSETTE
JACD16069105
DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI

Professeur-superviseur
ALAIN APRIL

MONTRÉAL, 23 AVRIL 2015
HIVER 2015

Remerciements

Je tiens à remercier Stéphane Franiatte (agissant comme chef d'équipe en logiciel) pour m'avoir montré et introduit la partie logicielle du projet de tourelle de paintball et pour m'avoir guidé, testé et intégré mon module au projet existant ainsi que pour avoir répondu à mes questions lorsque j'en avais besoin.

J'aimerais finalement remercier Alain April pour avoir été un excellent professeur superviseur, ainsi que pour avoir été très compréhensif et accommodant.

CONCEPTION ET DÉVELOPPEMENT D'UN MODULE DE CONTRÔLE MANUEL D'UNE TOURELLE DE PAINTBALL

DOMINIQUE JACQUES-BRISSETTE
JACD16069105

Résumé

Le projet de tourelle-sentinelle de paintball est présentement développé par le professeur Alain April et son équipe. Le but est de pouvoir éventuellement créer un club étudiant à l'ÉTS qui pourra participer à des compétitions internationales d'étudiants en ingénierie, comme le font les autres clubs de l'université.

Ce projet cible spécifiquement le développement d'un module qui devra être intégré au reste du projet par la suite. Ce module doit permettre le contrôle manuel de la tourelle : contrôle des 2 servomoteurs qui orientent le fusil (axe horizontal et axe vertical), ainsi que le contrôle du servomoteur qui permet de tirer les projectiles (activer la gâchette), en permettant un mode de tir en rafale.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

Sigle/abréviation	Définition
OOP	Programmation orientée-objet
Qt	Cadre de développement logiciel multiplateforme développé pour le langage C++
Qt Creator	IDE bâti autour de la technologie Qt
IDE	Environnement de développement intégré, permettant le développement, débogage et les tests d'une application logicielle
GUI	Interface Utilisateur Graphique
MVC	Modèle-Vue-Contrôleur, idéologie de conception logicielle visant à séparer les couches de présentation (GUI), de logique d'affaire et de modélisation des données
OS	Système d'exploitation (ex: Windows, Linux, Mac OS X)
PFE	Projet de fin d'études
SDK	Plateforme de développement logiciel
SRS	Document de spécification des exigences logicielles
S&S	Signaux et "slots", implémentation du patron observateur dans Qt
Cadriciel	"Cadre logiciel", vient de "framework" en anglais

INTRODUCTION

Le professeur superviseur Alain April et son équipe développent présentement le projet de tourelle-sentinelle de paintball dans l'optique d'avoir un tel club étudiant à l'ÉTS éventuellement. Ce club étudiant pourrait alors participer dans des compétitions internationales pour étudiants, comme le font actuellement la plupart des clubs de l'ÉTS.

Ce projet représente un défi dans de multiples domaines. En effet, le génie mécanique est nécessaire afin de développer et mettre en place les pièces physiques requises pour avoir une tourelle stable (trépied et supports) et qui supporte certains types spécifiques de mouvements (moteurs et engrenages). Le génie logiciel quant à lui est nécessaire pour plusieurs éléments spécifiques. Premièrement, une partie de l'application doit permettre à la tourelle d'utiliser sa vision (via une webcam) pour identifier les cibles en mouvement et faire la prise de décision grâce à une machine à états et à un réseau de neurones. Deuxièmement, une partie doit s'occuper de l'asservissement des servomoteurs, c'est agir à titre d'interface entre les commandes envoyées depuis l'ordinateur (position, tir, etc.) et les microcontrolleurs (qui commandent directement les moteurs) ainsi que leur calibration. Cette partie relève plus du logiciel embarqué et pourrait également chevaucher les génies de production automatisée et électriques, cependant ça peut se faire par des étudiants en génie de l'informatique. Troisièmement, une partie doit s'occuper de la gestion de la tourelle, il s'agit du morçeau central applicatif qui relie tout ensemble, cette partie interface plus particulièrement avec le module de gestion des moteurs, ainsi qu'avec l'interface utilisateur pour configurer certaines actions. Le principal objectif de cette partie est de gérer la zone de tir, la calibration, les positions de tir, les types de tir (une balle à la fois ou mode rafale configurable, soit "burst fire" en anglais) ainsi que le tir lui-même.

Ce projet porte sur la troisième partie, mais spécifiquement dans une optique de contrôle manuel de la tour. Pour l'instant, aucune partie du logiciel ne permet à un humain de contrôler la tourelle, cela limite les capacités à tester et développer différentes composantes et fonctionnalités du projet dans son ensemble. Ce mode manuel permettra à un humain de contrôler lui-même la tour (donc sans intelligence artificielle) pour viser des cibles et tirer, ainsi que déterminer le mode de tir via une interface utilisateur graphique (GUI). Le contrôle

manuel sera en effet très utile pour des fins de test, pour identifier et régler certains bogues potentiels, ainsi que faire la calibration des métriques conjointement au contrôle des moteurs.

Comme cette partie sera développée comme module indépendant (“standalone”) qui sera ensuite intégré au reste de l’application, les qualités logicielles visées sont la réutilisabilité, la fiabilité ainsi que la performance. Également, une bonne documentation du code en plus du rapport est importante afin d’assurer la facilité de compréhension du module et le transfert de connaissances dans l’optique de l’objectif de réutilisation.

Le rapport technique est présenté de la façon suivante : premièrement, dans la première section le contexte du projet est expliqué, dont les technologies utilisées, les requis du projet, ainsi que la proposition initiale de maquette. Deuxièmement, l’implémentation technique réalisée est présentée plus en détails. Cette section explique l’interface finale, ainsi que chacune des classes logicielles, notamment leur rôle et leur fonctionnement. Troisièmement, il y a une discussion de plus haut niveau concernant l’application. Effectivement, on y discute de l’application réalisée en termes de conception logicielle en expliquant les philosophies de design et de programmation sous-jacentes. En bref, les raisonnements et décisions derrière le projet tel qu’il a été réalisé sont expliqués, en mentionnant les alternatives identifiés. Quatrièmement, des recommandations pour le futures sont faites, quant aux fonctionnalités qui pourraient être intéressantes pour le projet, mais qui se situent en dehors des requis du projet actuel. Finalement, la conclusion permettra de remettre l’ensemble du projet en perspective et discuter du déroulement du projet, ainsi que de son futur.

Section 1 - Explication du contexte

1.1 - Présentation des technologies utilisées

Comme l'application existante du projet est déjà codée en C++ avec le cadre logiciel Qt, le document de spécifications (SRS) requière donc que le module indépendant de contrôle manuel de la tourelle soit programmé avec cette même technologie pour des raisons évidentes (notamment la réutilisabilité). Cependant, le projet actuel utilise la version 4.8 de Qt, mais Stéphane le chef d'équipe a mentionné que le projet migrerait éventuellement à la version 5. Comme le module est développé de façon indépendante, il a été programmé dès le départ avec la version 5 la plus récente.

Un des avantages de Qt est le support multiplateformes. En effet, il est possible de programmer une application avec le cadre logiciel Qt sur de multiples OS (technologie dite agnostique des OS). Par exemple, je ferai le développement sur une machine roulant sur OS X, et le programme pourra fonctionner sur une machine Windows ou Linux, pourvu que le code soit compilé sur la plateforme concernée. Comme il s'agit d'un projet Open Source, cela est d'une grande utilité, car cela permet à la grande communauté d'y participer peu importe l'OS qu'ils utilisent personnellement. De plus, l'IDE Qt Creator est disponible sur les 3 principaux OS (Windows, Linux, OS X), celui-ci facilite grandement le développement sous Qt et offre une excellente intégration et un bon support des diverses fonctionnalités offertes contrairement à un simple éditeur de texte. Il est bien de noter que l'utilisation de Qt Creator est facultative, mais elle est grandement encouragée dû à sa facilité d'utilisation, ses fonctionnalités et sa superbe commodité.

Un autre avantage est l'amélioration du langage C++ de base via les multiples librairies, cadres de travail, macros, fonctionnalités et diverses implémentations de patrons de conception. De cette manière, Qt est similaire à la librairie BOOST (http://www.boost.org/) du fait que tous les deux rajoutent des fonctionnalités et améliorent l'utilisation du langage C++. Un exemple concret utilisé dans ce projet est l'implémentation du patron "Observateur" sous forme de Signaux et de "slots", dont nous allons expliquer plus en détails dans les prochaines sections.

1.2 - Présentation du projet initial

Tel que défini dans le document des spécifications des requis logiciels (SRS), le module doit pouvoir gérer les deux moteurs pour l'orientation (axe horizontal et axe vertical), ainsi que le moteur permettant de tirer le fusil. Voici un extrait des exigences fonctionnelles du SRS :

> "Le logiciel de gestion du projet de tourelle de paintball automatisée devrait présenter une interface permettant à l'utilisateur de manœuvrer manuellement la tourelle, en agissant sur les effecteurs suivants :
>
> - Moteur A, qui ajuste l'inclinaison du fusil.
> - Moteur B, qui ajuste l'orientation du fusil.
> - Gâchette; une action sur la gâchette provoque un tir du fusil. Un mode rafale est également disponible."

Pour ce qui est des exigences non-fonctionnelles, ou qualités logicielles, on mentionne les suivantes :

- bonnes pratiques générales en termes de qualité du logiciel;
- réutilisabilité;
- fiabilité;
- performance.

Ces objectifs sont qualitatifs plutôt que quantitatifs, alors ils restent assez subjectifs. Cependant, il existe des moyens de les quantifier en précisant les objectifs, faute de définition précise, je me réserverai la discrétion de les déterminer.

Il a été précisé qu'il serait intéressant de s'inspirer d'autres projets similaires tels que http://www.realsentrygun.com pour le développement de ce module. Voici ci-dessous une capture d'écran du logiciel de gestion de la tourelle de ce dernier.

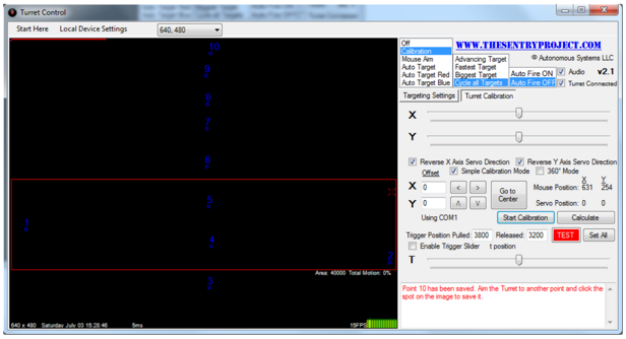

Figure 1.1 - Capture d'écran du projet RealSentryGun.com

1.3 - Projet proposé

Afin de répondre aux exigences mentionnées en 1.2, une maquette de base de l'interface a été proposée. Celle-ci démontre comment l'interface de base est constituée des éléments Qt qui satisfont les exigences fonctionnelles.

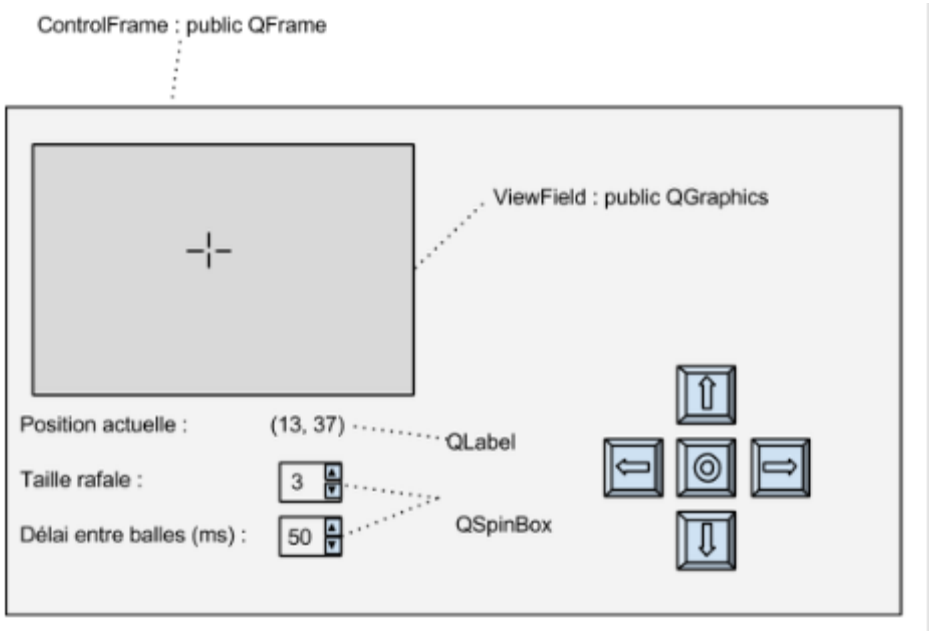

Figure 1.2 - Maquette d'interface proposée

Puisque le projet est développé sous forme de module indépendant, on veut pouvoir l'intégrer facilement. Pour y arriver, tous les éléments nécessaires sont contenus dans le `ControlFrame` (classe enfant de `QFrame`). Un `QFrame` représente seulement un conteneur dans lequel on y ajoute des objets, il est donc très facile de le prendre et l'ajouter à n'importe quel interface graphique, pourvu qu'on y ait prévu l'espace requise (pourrait facilement être ajouté dans une section d'un onglet). De plus, on peut le faire facilement grâce à l'éditeur graphique d'interfaces de Qt Creator. En effet, on n'a qu'à glisser-déposer un `QFrame` ordinaire à l'endroit souhaité, puis utiliser l'option "Promote" (promouvoir) et de sélectionner la classe enfant `ControlFrame`.

Les autres objets de l'interface parlent d'eux-mêmes et ne devraient pas nécessiter d'explications, mais ils seront tous décrits en détails dans la section 2 où on discute de l'implémentation finale (ceci n'est que la maquette initiale préalable).

Section 2 - Implémentation développée

2.1 - Présentation de l'interface développée

Le module développé et fonctionnel comporte une interface très similaire à la maquette qui a été présentée initialement, à quelques détails près. En effet, certains éléments ont été ajoutés ou modifiés afin de rendre l'expérience utilisateur plus agréable et plus efficace. Voici une capture d'écran récente du module indépendant.

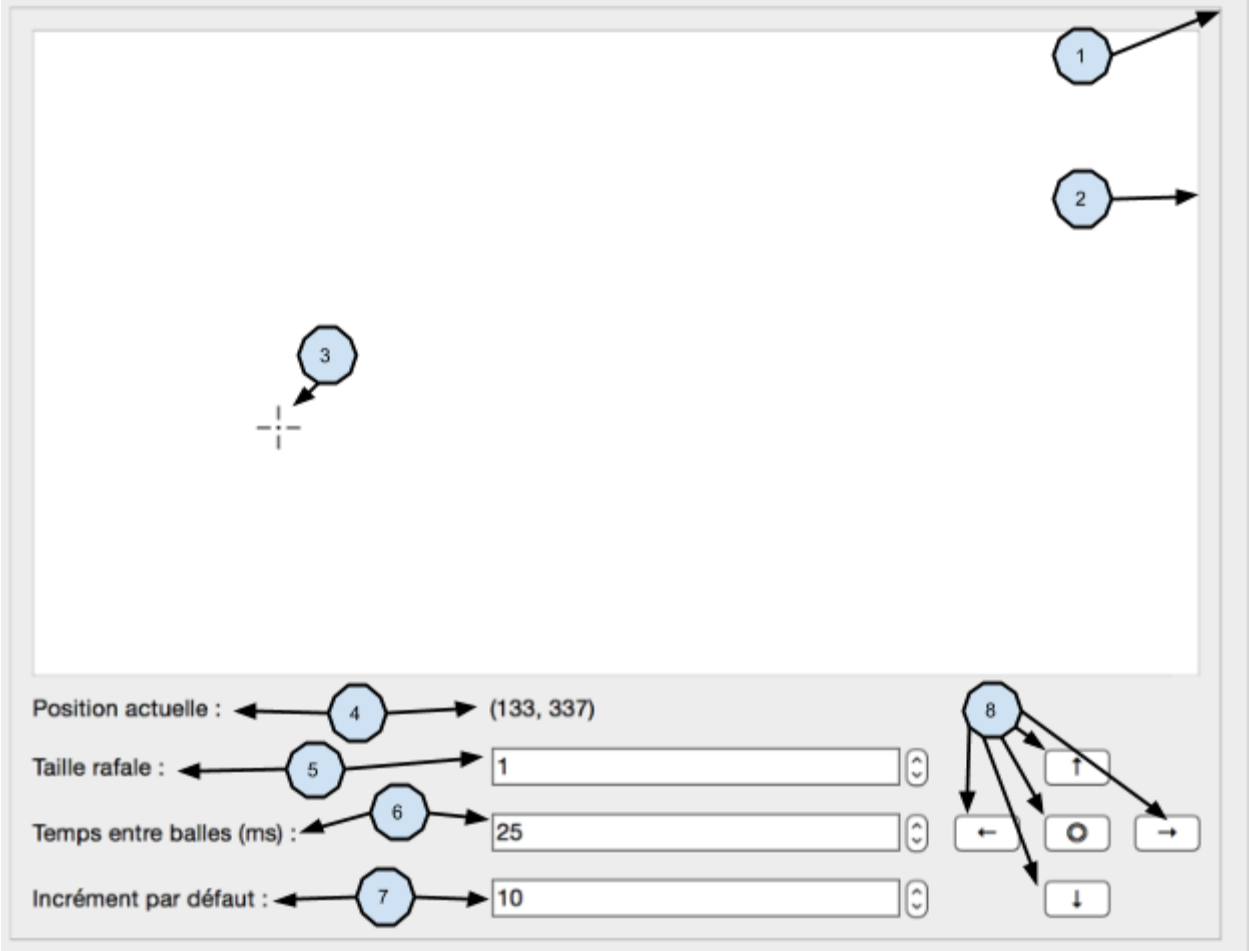

Figure 2.1 - Capture d'écran annotée du module

1. `TurretControlFrame` : classe conteneur qui représente et contient tout le module. Classe qui hérite de `QFrame`, sur l'image on y pointe la bordure. Utilise une disposition des éléments de type `QGridLayout`.

2. `DisplayArea` : classe héritant de `QGraphicsView` qui permet d'afficher la position actuelle de la direction visée de la tourelle. Permettra plus tard d'y ajouter la vision de la caméra en fond d'écran.
3. Réticule ("crosshair") : affiche la position actuelle de la direction de la tourelle. Dessinée par la classe `DisplayArea` via sa classe sous-jacente `QGraphicsScene`.
4. Libellés (`QLabel`) qui décrivent la position actuelle en coordonnées X et Y.
5. Libellé (`QLabel`) et contrôle numérique (`QSpinBox`) qui indique et permet de modifier le nombre de balles tirées pour chaque rafale (1 revient à un mode semi-automatique, relié au bouton de tir).
6. Libellé (`QLabel`) et contrôle numérique (`QSpinBox`) qui indique et permet de modifier le nombre de millisecondes de délai entre chaque balle tirée dans une rafale (permet donc de configurer la rapidité des rafales, également relié au bouton de tir).
7. Libellé (`QLabel`) et contrôle numérique (`QSpinBox`) qui indique et permet de modifier l'incrément par défaut pour modifier les coordonnées X et Y de la direction (relié aux boutons). Cela permet un contrôle plus fin ou plus grossier de la direction.
8. Boutons (`QPushButton`) permettant de contrôler les coordonnées X et Y de la direction, ainsi qu'un bouton pour effectuer le tir de rafale.

La disposition et l'initialisation des éléments graphiques est faite par programmation et non par l'éditeur graphique de Qt Creator. Pour faciliter la tâche d'altération de l'interface graphique, le code nécessaire est entièrement inclus dans une seule et même classe logicielle : `TurretControlFrame`. Cette décision diminuera grandement le temps requis pour un nouveau développeur voulant contribuer au projet de faire un ajout ou une modification au GUI.

2.2 - Présentation des classes objet

Pour de plus ample détails sur le fonctionnement de ces classes, se référer à la documentation fournie dans les fichiers d'en-têtes (`.h` "header files"). Chaque variable membre et méthode y est documenté de façon à ce qu'on ne devrait pas avoir besoin de lire le code C++ d'implémentation pour comprendre la classe.

Les méthodes dont on ne spécifie pas le type de retour ne retournent simplement rien (`void`). Les méthodes qui ne représentent que des simples accesseurs et mutateurs ("getters and

setters”) ne seront pas décrites, puisqu’elles sont banales (surtout pour les classes modèles), de même que les constructeurs et destructeurs.

2.2.1 - AimPosition

Cette classe représente le modèle de données de la position de tir (direction visée par le fusil). Elle store la position sous formes de coordonnées X et Y entières. On lui accorde des limites pour borner les positions possibles, ayant de 0 jusqu’au maximum établi. Toutes les données membres sont encapsulées de façon standard OOP avec les méthodes accesseurs et mutateurs (“setters” & “getters”, il s’agit des seules méthodes de cette classe). Elle hérite de `QObject` comme toute bonne classe modèle en Qt et permet d’utiliser les S&S.

Données membres :

- `m_positionX : int`
  - position horizontale actuelle, 0 étant à gauche
- `m_positionY : int`
  - position verticale actuelle, 0 étant au plus haut
- `m_maximumX : int`
  - Valeur maximale possible de X (défaut de 800)
- `m_maximumY : int`
  - Valeur maximale possible de Y (défaut de 600)

Signaux :

- `void valueChanged()`
  - Ce signal est émis dès qu’une des valeurs X ou Y change.

2.2.2 - ShootingMode

Cette classe représente le modèle de données du mode de tir (permet le mode rafale). Elle store le nombre de balles par rafale (1 étant un mode de tir semi-automatique) ainsi que les délais entre les balles (permet d’établir la vitesse d’exécution de rafale). Toutes les données membres sont encapsulées de façon standard OOP avec les méthodes accesseurs et mutateurs (“setters” & “getters”, il s’agit des seules méthodes de cette classe). Elle hérite de `QObject` comme toute bonne classe modèle en Qt et permet d’utiliser les S&S.

Données membres :

- `m_burstSize : unsigned int`
    - nombre de balles tirées pour chaque rafale, impossible d'être négatif, mais zéro représente le mode de sûreté.
- `m_timeSpread : unsigned int`
    - Nombre de millisecondes entre chaque balle tirée dans une rafale, permet de configurer la vitesse de rafale.

2.2.3 - DisplayArea

Cette classe de catégorie graphique représente la zone d'affichage où on montre la réticule de tir et éventuellement le "live-feed" de la caméra de type "Webcam". Elle hérite de `QGraphicsView` qui est une classe flexible permettant d'afficher des objets de fond ("background") ainsi que de dessiner et afficher des objets par dessus ("foreground"). En lui fournissant les coordonnées voulues, elle peut afficher la réticule au bon endroit.

Variables membres :

- `m_scene : QGraphicsScene*`
    - Pointeur vers la scène graphique où on y dessine les objets (réticule par exemple).

Méthodes publiques :

- `drawCrosshair(x : int, y : int)`
    - Rafraîchit la position de la réticule à l'écran avec les coordonnées X et Y passées en paramètre.

2.2.4 - TurretInterface

Cette classe représente la frontière de notre module qui va communiquer avec le reste de l'application, c'est-à-dire qu'elle devra utiliser le module d'asservissement des servomoteurs pour envoyer les commandes de position aux servomoteurs A et B et de tir de la gâchette. Pour des raisons de fiabilité, cette classe implémente le patron "Singleton" en s'assurant qu'il n'y ait une seule interface (pour ce module) qui envoie des commandes aux moteurs à la fois. Lors d'envoi des commandes, cette classe utilise directement les valeurs des objets modèles, comme elle possède leurs références (pointeurs). Comme la communication avec le reste de l'application réside à l'intérieur de cette classe, un développeur aura une grande facilité à

intégrer ce module. Ceci a été conçu en fonction du fait qu'il s'agit d'un logiciel libre, alors il faut garder en tête de faciliter la tâche aux développeurs s'intéressant nouvellement au projet.
La classe logicielle hérite de `QObject` ce qui permet d'utiliser les S&S.
Cette classe est une partie importante du module puisque c'est avec celle-ci que va interagir l'application qui intègre ce module.

Variables membres :

- `m_instance : static TurretInterface*`
    - Unique instance du singleton (statique)
- `m_aimPosition : AimPosition*`
    - Référence vers l'objet modèle de position de tir, utilisé pour obtenir les valeurs X et Y à envoyer
- `m_shootingMode : ShootingMode*`
    - Référence vers l'objet modèle de mode de tir, utilisé pour obtenir les valeurs de taille et de délai de rafale

Méthodes publiques :

- `setAimPosition(aimPosition : AimPosition*)`
    - Assigne le pointeur vers l'instance de l'objet modèle de position de tir
- `setShootingMode(shootingMode : ShootingMode*)`
    - Assigne le pointeur vers l'instance de l'objet modèle de mode de tir (rafale)
- `shootSingle()`
    - Envoie les commandes nécessaires pour effectuer le tir d'une seule balle (utilise les méthodes `sendTriggerOn()` et `sendTriggerOff()`)
- `shootBurst()`
    - Envoie les commandes nécessaires pour effectuer un tir en mode rafale (selon les valeurs du modèle `ShootingMode)`, utilise la méthode `shootSingle()`.
- `sendTriggerOn()`
    - Envoie la commande au moteur pour tirer sur la gâchette
- `sendTriggerOff()`
    - Envoie la commande au moteur pour relâcher la gâchette
- `sendPositionAxisX()`
    - Envoie la coordonnée X de position au servomoteur horizontal

- `sendPositionAxisY()`
    - Envoie la coordonnée Y de position au servomoteur vertical

2.2.5 - TurretControlFrame

Cette classe représente le coeur du module, car elle hérite de la classe conteneur `QFrame` et contient donc tous les éléments graphiques et elle gère les objets modèles et connecte aussi les signaux et “slots” ensemble. Elle s’occupe aussi d’initialiser les éléments GUI et de les placer dans une disposition en grille (`QGridLayout`). En ajoutant ce conteneur graphique dans une application existante, on intègre effectivement le module de contrôle manuel dans son ensemble, il s’agit donc du point central et crucial du module. C’est aussi cette classe qui interagit avec l’instance unique (Singleton) de l’interface `TurretInterface` pour “parler” aux moteurs.

Variables membres (excluant la plupart des éléments graphiques) :

- `m_aimPosition : AimPosition*`
    - Objet modèle de la position de tir
- `m_shootingMode : ShootingMode*`
    - Objet modèle du mode de tir (rafale)
- `m_displayArea : DisplayArea*`
    - Objet graphique de la zone d’affichage
- `m_layout : QGridLayout*`
    - Disposition en grille, on y place tous les éléments GUI par leur position dans la grille

Slots publiques :

- `handleUpButton()`
    - Connecté au signal émit par le bouton “↑” lorsqu’il est activé. Modifie la position Y dans l’objet modèle de position selon l’incrément par défaut.
- `handleDownButton()`
    - Connecté au signal émit par le bouton “↓” lorsqu’il est activé. Modifie la position Y dans l’objet modèle de position selon l’incrément par défaut.
- `handleLeftButton()`

    - Connecté au signal émit par le bouton “←” lorsqu’il est activé. Modifie la position X dans l’objet modèle de position selon l’incrément par défaut.
- `handleRightButton()`
    - Connecté au signal émit par le bouton “→” lorsqu’il est activé. Modifie la position X dans l’objet modèle de position selon l’incrément par défaut.
- `handleShootButton()`
    - Connecté au signal émit par le bouton de tir lorsqu’il est activé. Initie la séquence de tir par la classe `TurretInterface`.
- `handlePositionChanged()`
    - Connecté au signal émit par le changement de valeur du modèle `AimPosition`. Notifie `TurretInterface` que la position a changé (qui doit donc relayer l’information aux servomoteurs). Utilise la méthode publique `showCurrentPosition()`
- `handleBurstSizeChanged()`
    - Connecté au signal émit par le changement de valeur du sélecteur graphique de taille de rafale. Modifie la valeur interne du modèle `ShootingMode`.
- `handleTimeSpreadChanged()`
    - Connecté au signal émit par le changement de valeur du sélecteur graphique de délai de rafale. Modifie la valeur interne du modèle `ShootingMode`.

Méthodes publiques :

- `showCurrentPosition()`
    - S’assure d’afficher correctement la bonne valeur présente de la position de tir (libellés et zone d’affichage : utilise la méthode `drawCrossair(x, y)` de `DisplayArea`).

Section 3 - Conception et discussion

Dans cette section, nous allons discuter des différents choix de conceptions ayant été faits durant l'élaboration de ce module, ainsi que les raisonnements derrière ces choix. Nous devons se rappeler que ce module a plusieurs objectifs en tête :

- réutilisabilité;
- fiabilité;
- performance.

3.1 - Paradigme MVC

Dans l'optique de réutilisabilité, j'ai décidé de prendre une approche MVC afin de séparer les différentes couches de l'application, sans toutefois utiliser l'approche strictement pure du terme, afin de s'adapter à l'idéologie Qt (notamment l'approche S&S dont nous allons discuter à la section 3.2). En effet, la classe de contrôleur principale hérite d'une classe d'élément graphique, mais celle-ci ne représente qu'un conteneur et non pas un élément proprement actif de l'interface. L'interface logicielle, c'est-à-dire la partie qui devra être connecté à l'application lors de l'intégration, réside à la frontière du module et est contenue dans une seule classe. Cette décision facilite grandement l'intégration en n'ayant qu'à ce soucier de cette partie et comment elle communique les informations.

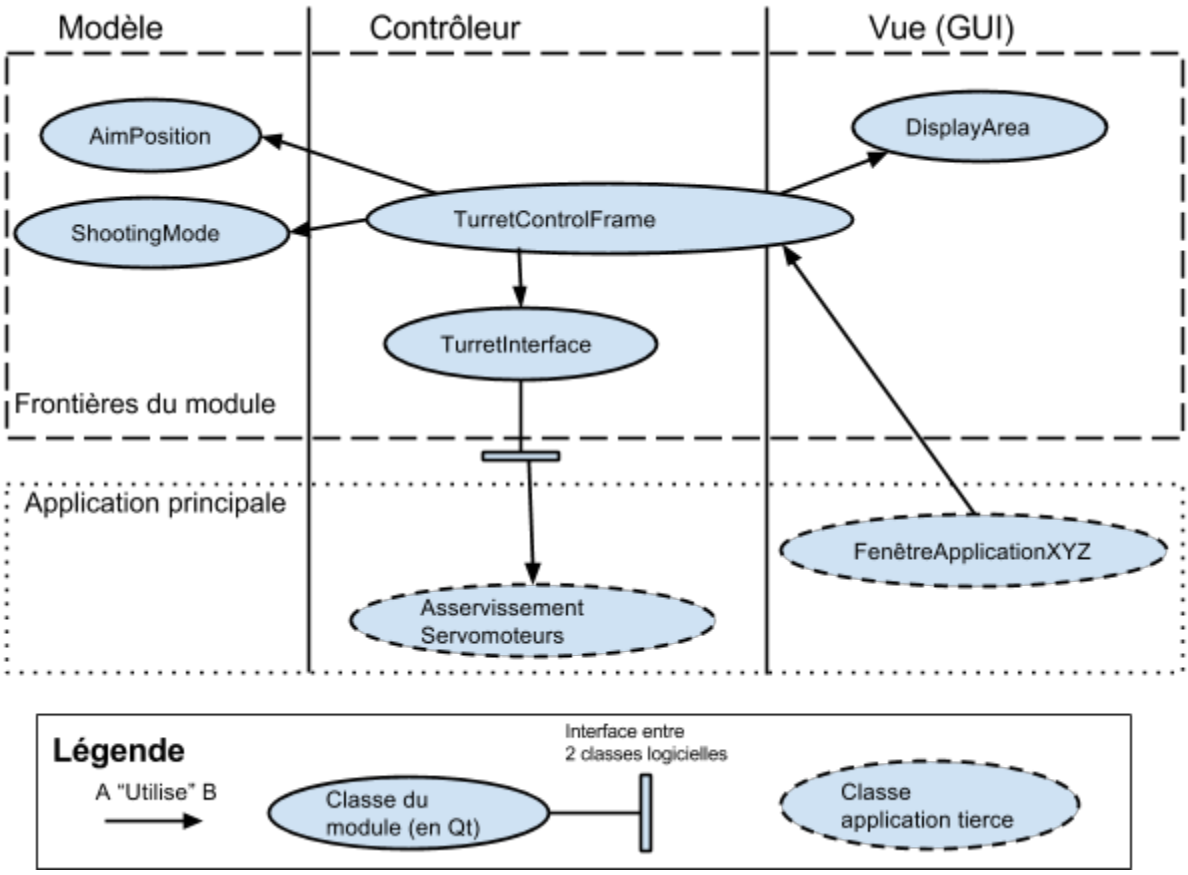

Figure 3.1 - Vue abstraite de l'implantation MVC

La figure 3.1 montre visuellement comment les différentes classes du modules sont connectées au niveau hiérarchique (qui contrôle ou utilise qui), ainsi que comment le module pourra être intégré à l'application générale de la tourelle (on reste abstrait, car le module doit être agnostique de cette application).

Les classes modèles restent très bien découplées du reste et conservent une forte cohésion, dû à leur rôle bien défini.

Pour ce qui est de la partie vue, la classe `DisplayArea` a aussi une forte cohésion du fait qu'elle ne s'occupe que de son affichage (ce qui reste toujours dans l'optique de la couche vue), et son interface (méthode publique `drawCrosshair)` ne résulte pas en un couplage important.

La seule “transgression” de l’idée MVC est contenue dans la classe `TurretControlFrame`, puisqu’il s’agit en fait d’une sous-classe d’un élément graphique (`QFrame`). Cependant, comme il ne s’agit d’un conteneur, le rôle de vue est très limité pour cette classe. En effet, elle ne s’occupe que de placer les éléments graphiques et connecter les événements GUI aux bons endroits. De plus, le fait d’utiliser cette classe d’une telle manière permet l’intégration facile du module en entier (sauf pour `TurretInterface`, mais on y arrive) en n’ayant qu’à ajouter cette extension de `QFrame` à l’interface graphique. Finalement, l’interface `TurretInterface` a été simplifiée à ses exigences fonctionnelles de base, afin de rendre l’intégration et la réutilisation le plus facile possible.

3.2 - Signaux et slots

Le SDK Qt permet d’avoir accès à une fonctionnalité extrêmement pratique qu’est le S&S. L’idée derrière ce principe est que certains éléments (objets héritant directement ou indirectement de QObject) peuvent émettre des signaux quelconques (à définir selon les besoins). Ces signaux peuvent ensuite être connectés à des slots, qui agissent comme des réceptacles, soit des méthodes qui réagissent à ces signaux. Il est possible de connecter un signal à plusieurs slots, et une slot à plusieurs signaux.

À partir de ce principe, on s’assure que lorsqu’une valeur d’un des modèle change, que les réactions voulues arrivent immédiatement après le changement de valeur. Pour donner un exemple concret, lorsque une coordonnée de `AimPosition` change, on le voit immédiatement à l’écran et les moteurs en sont instantanément notifiés. De plus, un modèle peut notifier un contrôleur, sans que le modèle ait à connaître le contrôleur (réduit le couplage à éviter).

Sans être exhaustive, la figure 3.2 ci-dessous montre la séquence d’exécution des S&S dans le cas d’un changement de position ou d’un tir.

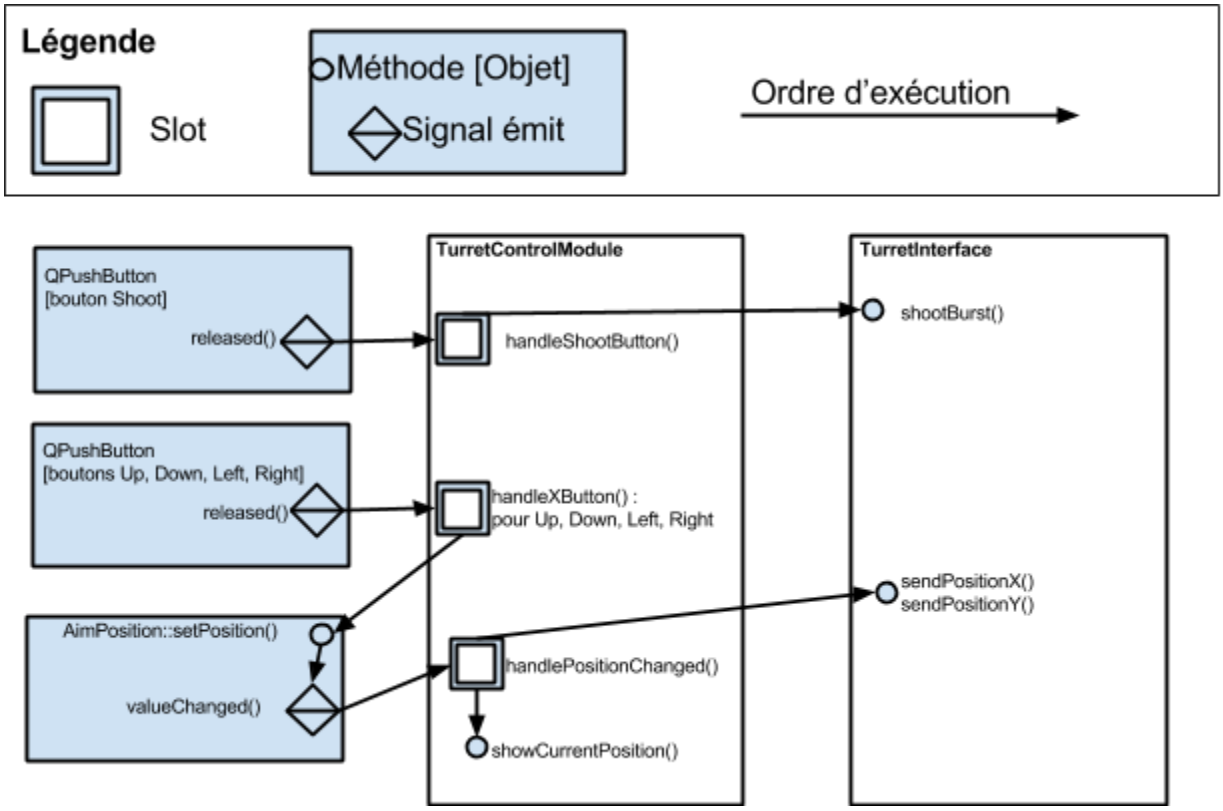

Figure 3.2 - Séquence d'exécution S&S pour un changement de position ou tir

Dû au nombre de connexions de type S&S, seulement la séquence d'exécution pour les changements de coordonnées et des tir ont été illustrés, afin d'éviter un diagramme trop complexe et illisible. En effet, ce paradigme a été utilisé à plusieurs endroits dans le projet, dès qu'on voulait qu'une composante réagisse immédiatement à un événement (surtout ceux reliés à l'interface), car c'est ce mécanisme qui est prôné par la philosophie de développement en Qt, et avec raison.

Section 4 - Recommandations

Dans cette section, nous jetons un regard sur le futur du projet, en donnant des recommandations et suggestions sur ce qui serait éventuellement à rajouter ou à modifier.

4.1 - Fonctionnalités reliées au GUI

Éventuellement, il serait intéressant d'ajouter quelques contrôles supplémentaires à l'application, qui ont été considérés, mais non implémentées faute de temps.

- Contrôle de sélection de la résolution : un sélecteur pourrait permettre de sélectionner une résolution fixe qui correspondrait à la vision donnée par la webcam ;
- Visualisation en temps réel de la zone de tir : le "live feed" de la webcam pourrait être ajouté en arrière-plan du DisplayArea pour voir plus précisément où on vise (et sur quels objets du monde physique) .
- Contrôles supplémentaires pour ajustement de la position : au lieu de n'utiliser que les boutons pour contrôler la position, un support pour d'autres types de contrôles pourraient être ajoutés. Par exemple : cliquer directement à l'endroit souhaité avec la souris, utiliser le mouvement de la souris pour bouger, etc.

4.2 - Fonctionnalités autres

- Unité de conversion de positions et d'unités : cette composante pourrait convertir les coordonnées X et Y en degrés d'inclinaison, ce qui correspond mieux à la réalité d'angles de tir par rapport aux servomoteurs. De plus, elle pourrait faire une conversion des coordonnées à l'écran vers des coordonnées ajustées à la géométrie du monde physique, ce qui faciliterait une calibration d'une meilleure précision;
- Journalisation des événements ("event logger") : une composante pourrait sauvegarder tous les événements importants qui arrivent dans un format qui pourrait facilement être lu par la suite, cela ouvrirait la porte à une fonctionnalité de type "enregistrement-lecture" qui pourrait être intéressant au niveau des tests et de la calibration.

Conclusion

Le projet de tourelle automatisée de paintball dans son état actuel n'est pas en mesure d'être contrôlé de façon manuelle. Cela limite l'équipe du projet dans l'établissement et exécution de tests pour vérifier l'exactitude des résultats, soit l'orientation précise du fusil de paintball et le tir des projectiles aux moments voulus. De plus ça rend encore plus difficile de faire une calibration précise de ces résultats.

L'objectif visé par ce module est de permettre à un humain de contrôler manuellement ces paramètres (tir et relâchement de la gâchette, position horizontale et position verticale de l'orientation) via une interface graphique s'exécutant sur un ordinateur portable. Ce module doit également être conçu de façon à facilement s'intégrer au reste de l'application existante, en restant performant et stable.

Pour ce faire, une interface graphique et le logiciel pour l'opérer ont été conçus à l'aide de la technologie Qt dans l'environnement Qt Creator avec le langage de programmation C++. En ce qui concerne l'interface graphique, elle est très facilement modifiable, puisque tous ses éléments sont définis et initialisés au même endroit dans le code. Cela permet à un développeur de sauver du temps en n'ayant pas à chercher longtemps où ajouter ou modifier du code pour altérer l'interface. De plus, l'établissement de l'interface logicielle facilite grandement l'intégration du module au reste de l'application de gestion de la tourelle en n'ayant qu'à modifier une seule classe. Encore une fois la complexité et le temps requis sont grandement diminués par cette décision.

Cependant, le potentiel de ce module n'a pas atteint sa limite. En effet, il serait possible d'ajouter des fonctionnalités ou possibilités de contrôle au module qui rendraient l'utilisation du logiciel de gestion encore plus puissant et/ou efficace dans son fonctionnement. De telles modifications mentionnées dans la section des recommandations ne sont pas extrêmement complexes à implémenter, mais n'ont pas été effectuées dû à un manque de temps. Cependant, comme il s'agit d'un projet de logiciel libre, celles-ci peuvent être ajoutées par n'importe quel développeur ayant l'intérêt de contribuer au projet.

Annexes

Annexe A - Code

Le code étant Open Source sous la licence Creative Commons, il est hébergé sur GitHub et disponible au public à l'adresse suivante :

https://github.com/dBizzle/PaintballTurretControlModule