2011-2012

SCOR : Sport Collectif pour Robots

Maxime SWINNEN
Loïc THOUVENIN-JENNER, CdP
Kevin VANDEL
Basile WOLFROM

10/03/2012

TABLE DES MATIERES

I. INTRODUCTION

A. OBJET DU DOCUMENT

Ce rapport décrit dans les grandes lignes le fonctionnement de l'application créée à l'occasion du projet SCOR. Dans un premier temps, on présentera le contexte du projet et les objectifs que nous nous sommes fixés. Ensuite, on présentera la mise en place de l'environnement de développement (configuration, dépendances…) et le manuel d'utilisation de notre application. Dans la suite du document, nous développerons nos choix de conception et d'implémentation. On commencera par une présentation de l'architecture du programme, ce qui permettra au lecteur d'identifier les différents modules fonctionnels du programme. Ensuite, on détaillera les points clés de la mise en œuvre de chacun de ces modules et on dressera un bilan de développement (problèmes rencontrés, améliorations possibles).

B. CONTEXTE DU PROJET

Le projet **SCOR** (**S**port **CO**llectif pour **R**obots) est un projet spécifique de cinquième année du département informatique de l'INSA de Lyon. Il vise à permettre aux étudiants de s'adonner à un projet technique pluridisciplinaire. Concrètement, il s'agit de réaliser un programme permettant de contrôler une équipe de deux robots de type Khepera II ; le but étant de se mesurer au programme réalisé par un groupe étudiant adverse. Pour ce faire, chaque équipe à accès à un PC sous Linux pour exécuter son programme, une caméra permettant de suivre l'activité sur le terrain et deux robots Khepera II.

Les principaux domaines de compétences mis en jeu dans le cadre de ce projet sont l'analyse d'image (localisation des robots et de la balle), l'informatique embarquée (contrôle des robots) et enfin l'intelligence artificielle (stratégies de jeu). Ce dernier point est toutefois à nuancer puisqu'à l'heure actuelle, aucun groupe n'a réellement implémenté avec succès un algorithme de ce type.

C. OBJECTIFS

Après une étude rapide des travaux réalisés par les équipes des années précédentes, nous avons constaté que la plupart des domaines fonctionnels étaient plutôt bien maîtrisés. Par contre, nous avons été désagréablement surpris par la mauvaise qualité et les mauvaises performances globales des applications. Il a donc été décidé que notre objectif principal serait « d'assainir » l'existant, ce qui a nécessité une réécriture complète de l'application. Toutefois, nous avons tâché de capitaliser au mieux sur l'existant. Il est également à noter que cette recherche de qualité nous a conduits à rationaliser les dépendances de notre programme (limitation des librairies utilisées).

D'autre part, à l'issue des réunions avec les autres concurrents, il a été décidé d'implémenter un nouveau mode de tir en rotation pour les robots ; dans le but de permettre d'augmenter la puissance des tirs et donc la vitesse de la balle.

II. MISE EN PLACE DE L'ENVIRONNEMENT

A. DEPLOIEMENT DU PROJET

L'archive fournie contient les sources du projet SCOR et les fichiers de projet Eclipse. Il est à noter que le projet Eclipse a été configuré (répertoires include, lib) en considérant que les dépendances ont été installées dans le répertoire /usr et non pas /usr/local comme c'est le cas par défaut. Installer dans ce répertoire évite toute configuration supplémentaire relative à l'édition des liens dynamique (i.e. à l'exécution).

Les deux PC de la salle sont sous Ubuntu Linux et ont été configurés correctement cette année. Normalement, il vous suffit d'importer le projet et de compiler pour obtenir un exécutable. Pour exécuter ce dernier depuis Eclipse, il vous faudra créer une configuration de lancement personnalisée (« Run configuration »). Dans l'onglet « Main », il faut renseigner le chemin vers l'exécutable dans le champ « C/C++ application ». Puis, dans l'onglet « Arguments », copier par exemple la ligne suivante :

« --comPort1="/dev/ttyUSB0" --comPort2="/dev/ttyUSB1" --LOGLVL=DEBUG », les deux premiers paramètres étant obligatoires (cf. manuel).

Dans le cas où les postes ont été réinstallés entre temps, il vous faudra procéder à la réinstallation des dépendances (librairies). Pour cela, vous pouvez vous référer au paragraphe « Dépendances du projet » si nécessaire.

B. DEPENDANCES DU PROJET

Liste des dépendances du projet et conseils d'installation :

- OpenCV 2.3.1 : Open Computer Vision est une bibliothèque graphique libre, initialement développée par Intel, spécialisée dans le traitement d'images en temps réel.
  Installation : télécharger l'archive Linux et suivre les instructions d'installation sur le site (http://opencv.willowgarage.com/wiki/). Il est à noter que la phase de configuration peut être réalisée en mode graphique avec l'outil cmake-gui. Il est primordial d'activer le support de GTK+ 2.x pour pouvoir utiliser le module highgui (fenêtres OpenCV). D'autre part, bien penser à cette étape à changer le répertoire d'installation (/usr au lieu de /usr/local).
- log4cplus 1.0.4 : bibliothèque de log.
  Installation : télécharger l'archive Linux sur Sourceforge (http://sourceforge.net/projects/log4cplus/) et configurer avec CMake (via cmake en ligne de commande ou cmake-gui en interface graphique). Encore une fois, bien penser à changer le répertoire d'installation (/usr au lieu de /usr/local).
- Autres librairies systèmes utilisées (intégrées par défaut à l'OS) :
  - pthread : gestion des threads à la norme POSIX
  - getopt : parser les paramètres en ligne de commande

Complément : CMake permet de générer des makefiles. Pour cela, sous l'interface graphique (cmake-gui), il faut tout d'abord sélectionner le dossier des sources de la librairie et un dossier de destination pour les makefiles. Ensuite, il s'agit de faire la configuration proprement dite. Enfin cliquer sur « Configurer » puis « Générer ». Il ne reste plus qu'à se placer dans le dossier de destination et appeler make pour compiler puis make install pour copier les binaires dans le dossier d'installation.

C. PARAMETRAGE DU PROGRAMME

De manière à simplifier l'accès aux paramètres principaux de l'application, nous avons décidé de les regrouper au sein d'un header de configuration (« config.h »). Il peut être intéressant de parcourir rapidement ce fichier après avoir pris connaissance de ce document de manière à avoir une idée générale des options de

configurations offertes par le programme. Les définitions étant des macros, leur modification ne prendra effet qu'après recompilation.

III. MANUEL UTILISATEUR

LIGNE DE COMMANDE

SCOR --comPort1=COMPORT1 --comPort2=COMPORT2 [option]...

OPTIONS

(--help | -h) : affiche ce message
--comPort1=COMPORT1 : COMPORT1 fichier device du port COM robot 1
--comPort1=COMPORT2 : COMPORT2 fichier device du port COM robot 2

Gestion des niveaux de log :
arg=(TRACE | DEBUG | INFO | WARN | ERROR | FATAL)

(--LOGLVL | -l)=arg : spécifie le niveau de log global
--LOGLVL_VISION=arg : niveau de log du module "Vision"
--LOGLVL_IA=arg : niveau de log du module "IA"
--LOGLVL_CONTROL=arg : niveau de log du module "Contrôle Robot"
--LOGLVL_COM=arg : niveau de log du module "Communication"
--LOGLVL_MAIN=arg : niveau de log du module principal

Remarque : la valeur COMPORT[1|2] est de la forme "/dev/ttyUSBX" (fichier device correspondant au robot X). Pour déterminer les valeurs de X, il faut lister les fichiers device existants (s'assurer d'abord que les robots sont connectés au PC et que leur alimentation est sous-tension). On peut faire cela simplement en exécutant la commande suivante (sans les guillemets) dans un terminal : « ls /dev/ttyUSB* ».

IV. ARCHITECTURE DE L'APPLICATION

Comme les équipes de l'année précédentes, nous avons fait le choix de réaliser notre application à l'aide d'un langage orienté objet, en l'occurrence ici C++.

A. COMMUNICATION INTER-MODULES

La communication entre ces modules est réalisée par file de messages. Par soucis de flexibilité (et même de portabilité), cette file a été implémentée par nos soins (via un Template C++) et n'a donc aucun lien direct avec un quelconque objet système. Les types de messages échangés sont paramétrables.

Il est à noter que les files de messages sont également souvent utilisées comme moyen de synchronisation. Elles sont par ailleurs créées dans le main de l'application (phase d'initialisation).

B. FLOT DE DONNEES

Le diagramme ci-dessous présente l'architecture de notre programme du point de vue des échanges de données entre les différents modules fonctionnels.

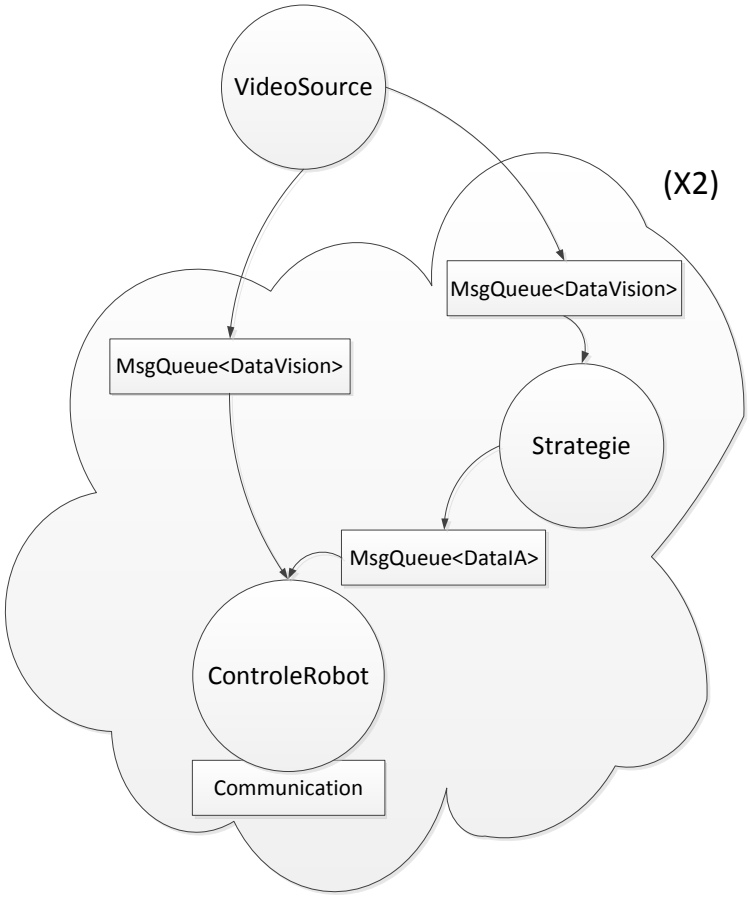

Figure 1 Diagramme de flot de données

Les trois modules principaux de l'application sont « VideoSource », « Strategie » et « ControleRobot » (qui sont des objets C++). Il est à noter que ces modules sont répartis au sein de paquetages distincts dans les sources de manière à les rendre aisément identifiables.

VIDEOSOURCE

Ce module est dédié à l'analyse d'image (basée sur la librairie OpenCV2). Il traite l'information issue de la caméra (à une fréquence paramétrable) de manière à identifier les positions et orientations des robots et la position de la balle. La communication des informations est réalisée par l'envoi de messages de type « DataVision ». Il est à noter que l'ouverture du flux vidéo est gérée en interne.

Chacun des modules suivants assure la prise en charge d'un unique robot. Par conséquent, à l'exécution, ces objets (et les files associées) sont instanciés en double. On rappelle que la cohérence du routage des informations entre les modules est assurée par file de message.

STRATEGIE

Ce module implémente les stratégies préalablement définies par nos soins. La période d'évaluation des stratégies est liée à la cadence d'arrivée des messages du module de vision. A l'issue de l'évaluation, si un nouvel ordre a été échu, le module envoie un message de type « DataIA ».

CONTROLEROBOT

Ce module est synchronisé sur la réception des ordres du module de stratégie. Il assure le contrôle des robots (envoi d'ordre, éventuellement asservissement..). Il est associé à un module passif [1], nommé « Communication », chargé d'abstraire la communication de bas niveau (liaison série).

A. GESTION DU PARALLELISME ET DE L'ASYNCHRONISME

Nous avons décidé d'encapsuler la gestion des fils d'exécution et de l'asynchronisme au sein même de nos objets. On parle ainsi d'objets « actifs ». Cela permet de limiter les dépendances et autorise ainsi le développeur en charge du module à opter pour l'implémentation de son choix. D'autre part, cela permet d'obtenir un code plus clair, du moins « à l'extérieur » des modules.

Par ailleurs, tous les modules du projet se basent sur la librairie POSIX de bas niveau « pthread ». Il est à noter que cette dernière est portable sous Windows (du moins en grande partie).

V. DESCRIPTION DETAILLEE DES MODULES

Dans ce paragraphe, il s'agira de présenter nos choix de conception et d'implémentation pour chacun des modules fonctionnels (points clés de mise en œuvre). On dressera également un bilan de développement dans lequel on exposera les problèmes rencontrés et les axes d'améliorations qui nous semblent pertinents.

A. MODULE VISION

1. OBJECTIFS

Ce module a pour objectif de détecter la position des robots et de la balle, informations qu'il transmet ensuite aux modules de Stratégie et de Contrôle.

2. FONCTIONNEMENT

Ce module maintient son propre thread. Lors de l'initialisation il lance une fenêtre affichant l'image obtenue par la Webcam permettant à l'utilisateur de cliquer sur les 4 coins du terrain, puis détecte le robot gauche et le robot droite.

Les opérations effectuées au sein du thread peuvent se résumer ainsi :

- Récupération de l'image provenant de la webcam
- Traitement de l'image pour ne garder que le terrain
- Détection de la balle sur le terrain

[1] Objet dont les méthodes sont toutes synchrones

- Détection des robots sur le terrain
- Envoi des coordonnées des robots et de la balle via les files de messages
- Affichage de l'image du terrain avec les positions des robots et de la balle, ainsi que les positions objectifs des robots (si le booléen FluxVideo est vrai)

Toutes ses étapes sont réalisées à une fréquence temporelle réglable moyennant une recompilation (cf. fichier config.h). Il est conseillé de le faire au moins 10 fois par seconde.

Les coordonnées sont transmises en centimètre, la conversion se fait grâce aux mesures de la taille réelle du terrain (LONGUEUR_TERRAIN_Y cm de long par LARGEUR_TERRAIN_X cm de large).

POINTS CLES

Le point clé de la vision est la détection des robots et de la balle. Elle se fait grâce aux couleurs.

Prenons l'exemple de la balle. Pour être détectée, cette dernière doit être rouge. La première étape consiste à filtrer l'image de façon à ne garder que les pixels rouges. La différentiation des pixels rouges se fait à partir des valeurs des couleurs en RGB. Un pixel est considéré comme rouge si sa composante rouge est supérieure à la fois à sa composante bleu et à sa composante verte auxquelles on ajoute une constante (appelée TOLERANCE). Une fois l'image filtrée récupérée, on cherche le barycentre des points conservés. Ce barycentre correspond à la position de la balle.

Les coordonnées en pixels ainsi obtenues doivent ensuite être transformées en centimètre pour les autres modules (à l'aide d'un produit en croix).

3. PROBLEMES RENCONTRES

Le principal problème rencontré dans le module de vision est la difficulté à réaliser une bonne distinction des couleurs en fonction de l'éclairage de la pièce. Il est en effet nécessaire de faire varier les tolérances, en particulier celle du bleue, suivant la nature de l'éclairage (lumière du jour, néons) et même parfois de l'heure de la journée.

4. AMELIORATIONS POSSIBLES

Ce module, repris en grande partie à partir du code de l'année de précédente, fonctionne tout à fait convenablement, tout en ne consommant pas trop de ressources. En dehors de quelques petites optimisations, il ne nécessite pas de réelles améliorations.

Une piste d'amélioration serait peut-être de reprendre entièrement le principe de détection et d'exploiter les possibilités de la librairie OpenCV en matière de recherche de motif dans une image (méthode d'apprentissage). Cependant le gain envisageable risque de ne pas valoir le temps de développement.

B. MODULE STRATEGIE

1. OBJECTIFS

Ce module a pour rôle d'étudier les conditions de jeux qui lui sont fournies par le module de Vision (positionnement de la balle, positionnement des robots) et d'en déduire la stratégie de jeu à adopter, ainsi que le robot devant intervenir.

2. FONCTIONNEMENT

ARCHITECTURE

Ce module met en jeu son propre thread dédié à la détermination de la stratégie à adopter. Le module communique à la fois avec le module de Vision, en recevant des messages qui contiennent la position des robots et la position de la balle à un instant t, et avec le module de Contrôle en lui envoyant des messages, pour chaque robot, qui contiennent : le type de message (Attaque, Defense, Replacement), la position à atteindre pour réaliser l'objectif et accessoirement un vecteur d'orientation qui n'a pas été utilisé mais qui pourrait utile dans le futur pour indiquer la position dans laquelle le robot doit se trouver lorsqu'il atteint son objectif.

INITIALISATION DE L'ANALYSE

Le module de stratégie est cadencé par la réception des messages du module de Vision. La première action réalisée est le calcul de la vitesse de la balle. Ensuite, le module teste si les informations reçues sont valides ou correspondent à des aberrations dues à l'incertitude de détection du module de Vision. Ces incertitudes peuvent concerner la position de la balle à un instant t (mauvaise détection) ou à des incohérences de trajectoire. Pour détecter une nouvelle trajectoire, il faudra donc que plusieurs messages consécutifs traduisent ce changement de trajectoire. (Ce nombre de messages consécutifs est paramétrable).

EVALUATION DE LA STRATEGIE

Il existe actuellement 3 stratégies :

- Attaque

La stratégie d'attaque est déclenchée lorsque la balle se trouve dans notre moitié de terrain et qu'elle ne se dirige pas vers nos buts. Le robot situé dans le même quart de terrain que la balle reçoit un ordre d'attaque avec une position proche de celle-ci lui permettant de frapper par rotation sur lui-même et l'autre robot reçoit un ordre de replacement défensif avec la position correspondante.

- Défense

Cette stratégie est lancée lorsque la balle se dirige vers nos buts avec une vitesse supérieure à un certain seuil paramétrable. Les 2 robots reçoivent un ordre de défense avec la position à laquelle ils doivent se rendre. Cette position correspond à la projection orthogonale de leur position sur la trajectoire de la balle.

- Replacement

Cette stratégie intervient lorsque les robots ne sont ni en phase offensive, ni en phase défensive. Chaque robot reçoit une position prédéfinie. Ces positions sont reculées dans notre moitié de terrain et les 2 robots sont toujours positionnés en diagonale pour couper la trajectoire d'une éventuelle attaque. Le robot qui est légèrement avancé est celui qui se trouve dans la même moitié de terrain (dans la largeur) que la balle.

ENVOI DES MESSAGES

Pour éviter d'envoyer des messages incohérents ou trop fréquemment aux robots (sujet à plantage), plusieurs vérifications sont effectuées. Dans un premier temps, on vérifie que l'objectif envoyé ne se situe pas dans l'autre moitié ou l'autre quart de terrain ou encore en dehors des limites. Dans ce cas le robot concerné ne fait rien et attend le prochain ordre. Il faut remarquer que les limites effectives de déplacement des robots dans le terrain sont plus faibles que les dimensions réelles de celui-ci pour éviter des blocages dans les bordures.

De plus, on vérifie pour chaque robot qu'un nouvel ordre envisagé se distingue sensiblement de l'ordre envoyé précédemment, de manière à éviter d'interrompre inutilement un déplacement en cours. (Le seuil de tolérance de cette vérification est paramétrable).

Enfin, une limite temporelle de fréquence d'envoi des messages a également été fixée.

3. PROBLEMES RENCONTRES

Le problème principal que nous avons pu rencontrer pour le module de stratégie a été la difficulté de test du module, due aux problèmes techniques et matériels qui ont provoqué des retards. Ces retards ont largement freiné notre avancement dans ce module.

4. AMELIORATIONS POSSIBLES

Une première modification à apporter serait d'améliorer la réactivité des robots, en particulier en défense. Cette modification est probablement aussi liée au contrôle des robots mais peut-être que la stratégie en elle-même n'est pas la plus efficace possible.

Il peut être intéressant de se pencher sur des stratégies plus complexes, comme par exemple, déplacer dans un premier temps une balle difficile à jouer pour la placer à un endroit plus intéressant pour attaquer efficacement.

Enfin, il serait intéressant d'adapter la stratégie en fonction des positions (voire des vecteurs vitesses) des robots adverses. En attaque, par exemple, il serait alors possible d'essayer de tirer dans une direction stratégique pour marquer plus facilement des buts.

C. MODULE CONTROLE ROBOT

1. OBJECTIFS

L'objectif de ce module est, comme son nom l'indique, de contrôler les robots, c'est-à-dire de faire en sorte que les robots exécutent correctement les ordres qu'on leur donne. Ce module est en quelques sortes une interface entre le module de Stratégie et les robots.

2. FONCTIONNEMENT

ARCHITECTURE

Le module de contrôle se divise en deux classes : la classe « Communication » et la classe « ControleRobot ».

La classe « ControleRobot » possède en attribut un objet « Communication », et un objet « ControleRobot » est instancié pour chaque robot (cf. le main). Cette classe implémente 3 threads différents : le thread principal de

contrôle (qui boucle sur la file de message des ordres issus du module de stratégie), le thread mettant à jour la position courante du robot (qui boucle sur la file de message des positions alimentée par le module de vision), et le thread d'asservissement (qui n'est plus utilisé, mais qui peut être *ressuscité* facilement).

La classe « Communication » implémente les méthodes de bas niveau permettant l'initialisation d'un port de communication ainsi que l'envoi et la réception de messages du PC vers les robots. Elle implémente aussi les méthodes de plus haut niveau correspondant aux commandes de base du robot (définir sa vitesse, remettre à zéro son compteur de position des roues, etc.).

La classe « ControleRobot » s'occupe d'exécuter au mieux les ordres donnés par le module de Stratégie, via des messages envoyé dans une file de message. Il existe trois types d'ordres : replacement, défense et attaque. Pour exécuter ces ordres, nous avons défini trois types de mouvement (implémenté dans trois méthodes distinctes dédiées) : tourner, avancer et tirer.

MOUVEMENTS DU ROBOT

Il existe deux types de déplacements pour les robots Khepera II : le déplacement par vitesse et le déplacement par position. Le déplacement par vitesse consiste simplement à affecter une vitesse définie aux deux roues du robot (on constate que le robot dépasse difficilement la barre des 30). Le déplacement par position consiste à affecter aux roues une position cible. Cette position est donnée en « pulses » : si l'on affecte 1 pulse à une roue, elle tournera de manière à parcourir 0,08 mm. Il faut donc voir cette position comme un nombre de tours de roues. Il est aussi important de savoir que le robot garde en mémoire un compteur du nombre de pulses parcourus par les roues. Pour plus de détails, vous pouvez vous référer au « User manual » : *3.1.5 Motors and motor control* (p. 10) ; *List of Available Commands* (p. 47 - commandes C et D).

Tourner

Pour tourner, nous utilisons le déplacement par position. Grâce à de multiples mesures, nous avons pu calculer un coefficient permettant de convertir l'angle de rotation souhaité en pulses. Cela nous permet de faire tourner le robot très précisément (les erreurs sur la rotation venant principalement du module de vision), en remettant à zéro son compteur puis en lui commandant d'atteindre la position de roue définie (les valeurs des positions des deux roues étant évidemment opposées pour que le robot tourne sur place).

Avancer

Ici encore, nous utilisons le déplacement par position. En fait, nous commandons au robot d'avancer (en marche avant ou marche arrière) sur une distance définie. Ainsi, nous avons juste à remettre le compteur du robot à zéro et à convertir la distance souhaitée en pulses (les valeurs des positions des deux roues sont dans ce cas identique pour que le robot aille tout droit).

Tirer

Cette année, nous nous sommes mis d'accord pour ajouter aux robots un bras en plastique (serre-fil), leur permettant de tirer en tournant sur place. Ainsi, la puissance du tir est décuplée par rapport à une simple charge en marche avant. Pour le tir, nous utilisons le déplacement par vitesse. On commande au robot de tourner sur place à grande vitesse pendant une durée prédéfinie. Passé ce délai, on lui commande de s'arrêter.

STRATEGIES DE DEPLACEMENT

Défense et replacement

Les traitements des ordres de replacement et de défense sont similaires, il s'agit de déplacer le robot jusqu'à un point de destination (dont les coordonnées sont spécifiées dans le message reçu du module de stratégie). Le déplacement du robot se fait en deux temps. Tout d'abord, on ordonne au robot de tourner pour qu'il se positionne en direction de l'objectif. Pour gagner du temps, on permet au robot de se retrouver soit face à

l'objectif, soit dos à l'objectif (l'angle maximal à parcourir est alors de 90° au lieu de 180°). Ensuite, on calcule la distance qui sépare le robot de son objectif, et on commande au robot d'avancer (ou de reculer s'il fait dos à l'objectif) de cette distance.

Enfin, dans le cas particulier du replacement, on commande au robot de tourner de manière à être orienté dans la direction indiquée par la Stratégie (ainsi le robot sera mieux positionné pour une éventuelle interception).

Attaque

Le traitement des ordres d'attaque se fait en deux étapes principales : le positionnement (jusqu'aux coordonnées indiquées dans le message d'ordre), puis le tir. Pour éviter au maximum les collisions, nous avons décomposé le positionnement du robot en deux lignes droites, selon la longueur puis la largeur du terrain. Aussi, pour éviter que le bras du robot ne touche la balle ou les bords du terrain, nous forçons le robot à s'orienter systématiquement en marche arrière.

Astuce pour gagner en précision

Les déplacements des robots étant relativement peu précis (aussi bien pour tourner que pour avancer), nous avons eu l'idée de vérifier à la fin du replacement défensif et du placement en position d'attaque si le robot est proche ou pas de son objectif. Dans le cas contraire, on réinsère l'ordre courant dans la file de message, ainsi le robot ré-effectuera le déplacement (et étant donné qu'il est normalement plus proche de l'objectif, il l'atteint plus facilement). Cette méthode empirique fonctionne bien, mais il serait quand même préférable d'améliorer la précision des déplacements (cf. *Améliorations possibles*), d'autant plus que cela entraîne un temps de déplacement bien plus long, ce qui peut être préjudiciable, surtout en défense.

3. PROBLEMES RENCONTRES

Les robots : leur comportement n'a pas l'air d'une rationalité parfaite ! En effet, il arrive parfois qu'après un tir le robot devienne incontrôlable. Cela serait dû au fait que la puissance demandée par les moteurs lors du tir serait trop grande, engendrant une sous-alimentation du processeur du Khepera II. Une solution serait d'optimiser le câblage (pertes de puissance importante) ou de diminuer la vitesse demandée lors des tirs (fixée à 40, une vitesse de 30 suffirait peut-être).
Le déplacement par vitesse manque également cruellement de précision.
De façon à profiter au mieux du projet, nous vous conseillons de considérer les problèmes matériels au plus tôt et de trouver moyens d'y remédier.

4. AMELIORATIONS POSSIBLES

- Dans la communication, prendre en compte la valeur retournée par les méthodes d'envoi et réception de messages (gestion des cas d'erreurs).
- Amélioration de la trajectoire des robots : faire en sorte que le robot atteigne de manière la plus précise possible la destination objectif (du premier coup). Cela pourrait se faire par asservissement.
- Gestion des ordres de la Stratégie : simplement prendre en compte les ordres d'attaque donnés par la Stratégie (attaque gauche et attaque droit) afin de déterminer le sens de rotation pour le tir (au lieu le recalculer dans le thread de contrôle...).
- Structure du code : les méthodes "tourner", "tirer", etc. sont censées être privées. Elles ont été mises en publique simplement pour faciliter les tests.

VI. CONCLUSION

En l'état, nous avons partiellement atteint les objectifs qui nous nous étions fixés. Paradoxalement, notre démarche de rationalisation nous a conduits à un code qui manque parfois de lisibilité de par sa densité. Nous y aurions gagné à réaliser un découpage plus fin (par exemple, création d'un module d'interface graphique distinct du module de traitement vidéo). D'autre part, la qualité du code interne aux modules n'est pas toujours conforme à ce que nous nous étions fixés en début de projet. Toutefois, tous nos efforts n'ont pas été vains et notre objectif de performance a été atteint avec brio. L'application est aujourd'hui réactive, et sa consommation en ressources système reste tout à fait maîtrisée.

On ne peut que conseiller aux futures équipes de chercher à concentrer le meilleur de chaque projet. On peut par exemple citer la détection automatique du terrain, innovation intéressante introduite par l'équipe d'Alexis Fouilhe. En effet, l'évolution du projet ne pourra se faire à notre sens que par de la capitalisation sur le long terme ainsi que de l'innovation sur le plan matériel.