Projet *SCOR* — Bilan

Alexis Fouilhe, Quentin Legrand et Charlotte Simonnet

19 février 2012

Table des matières

Nous présentons ici notre retour sur expérience sur le projet *SCOR*. Pour rappel, cela consiste en faire s'affronter deux équipes de deux robots, pilotés de façon automatique à l'aide d'informations fournies par une Webcam. Le jeu ressemble sensiblement à *Pong* et se joue avec une balle. Une description plus détaillée est disponible sur le site Web du projet.

Ce rapport a été écrit après la fin du projet et mériterait d'être bien davantage illustré.

1 Démarche

Nous avons accès aux sources des solutions des années précédentes dès le début du projet, ce qui constitue une source d'inspiration non négligeable. Néanmoins, l'état du code ne nous a pas permis de le réutiliser en l'état. Assez rapidement, il a été décidé de tout ré-implémenter. Bien que cela soit confortable de travailler avec des composants

que l'on a écrit soi-même, les écrire prend du temps, qui ne peut donc être investi sur autre chose. C'est sans doute regrettable, si on considère que le projet est proposé depuis plusieurs années. Un des objectiifs que nous nous sommes fixés, par conséquent, a été de produire des modules réutilisables et documentés. Malheureusement, si nous avons atteint en partie cet objectif, nous ne sommes pas parvenus à atteindre l'objectif initial : faire jouer les robots à la balle.

Cela étant dit, nous sommes partis sur une démarche de développement itérative afin de se prémunir des problèmes d'intégration de fin de projet. L'idée a donc été d'écrire les composants un à un, en partant des interfaces avec le matériel (robots et Webcam) pour finir par l'intelligence artificielle. Nous avons ainsi pu réaliser des tests sur de vraies données et ne pas avoir à figer des interfaces avant de pouvoir évaluer les besoins de communication entre les modules.

L'approche fonctionne plutôt bien : nous nous sommes retrouvés rapidement avec des modules qui n'ont pas eu besoin d'être retouchés ensuite. C'est le cas, par exemple, des classes *CameraHandler* et *Khep*. L'implémentation de certaines fonctions a été validée à l'aide d'ébauches de tests unitaires. Malheureusement, ceux-ci n'ont parfois pas suivis l'évolution de dernière minute desdites fonctions, mais ils sont encore disponibles dans les sources sous forme de fichiers dont le nom commence par *test*.

2 État des Lieux en Fin de Projet

Comme il a dit été dit plus haut, nous ne sommes pas parvenu à aboutir à une solution complète. L'intelligence artificielle est basique, incomplète et mal testée, de même que l'asservissement en position du robot qui, lui, ne fonctionne pas du tout. Cela a été le fruit d'un développement en urgence quelques jours avant la démonstration.

Le reste, par contre, fonctionne correctement, et a été testé de manière raisonnable. Il n'est pas parfait, pour sûr, certains commentaires dans le code et les remarques ci-dessous donnent les axes que nous pensons être à améliorer.

Chaque fois que nous avons utilisé des algorithmes ou des configurations non triviaux, nous avons fait de notre mieux pour les commenter de façon à les rendre rapidement compréhensibles.

Pour finir, nous n'avons jamais eu de problèmes de performance. S'il devait en survenir, il y a suffisamment de parallélisme dans la plupart des actions pour pouvoir accélérer grandement les traitements.

3 Environnement et Mode d'Emploi

Notre solution est développée en *C++*, et utilise *OpenCV* et *Video4Linux*. Nous l'avons testée sur les deux machines de la salle où se trouvent les robots, c'est-à-dire sur *Ubuntu* 10.10 et 11.04. La version 2 de *Video4Linux* a été utilisée.

3.1 *OpenCV*

Nous avons utilisé exclusivement l'interface *C++* de *OpenCV*. Certains morceaux (*KheperaFinder* et *BallFinder*) ont été testés avec *OpenCV* 2.2.0 et l'intégralité a été testée avec *OpenCV* 2.3.1a.

Nous avions peu de connaissances en matière d'analyse d'image avant de démarrer le projet et *OpenCV* ne nous a pas aidé de ce point de vue. On ne trouve pas dans sa

documentation des méthodes pour accomplir un but, seulement la description de briques de base qu'il faut ensuite assembler.

Une fois ceci compris, la bibliothèque est d'utilisation aisée. Il a été nécessaire, par contre, d'utiliser des tutoriaux ou de plonger dans des livres sur l'analyse d'image et la vision par ordinateur.

3.2 Compilation

Pour compiler notre projet, il suffit de lancer `make` à la racine de son arborescence. Attention toutefois aux conflits entre versions d'*OpenCV*. Plusieurs versions sont installées sur les postes et ce, à plus endroits. Nous avons eu besoin de compiler notre propre *OpenCV*, d'où le chemin vers *OpenCV* dans *mk/rules.mk*. Par facilité, nous avons néanmoins mis les bibliothèques binaires dans le chemin standard, mais il est possible de jouer avec le drapeau *-rpath* de `ld`, ou la variable d'environnement *LD_LIBRARY_PATH* pour utiliser des bibliothèques installées à des endroits arbitraires.

3.3 Lancer l'Exécutable

Le programme est, dans une certaine mesure, auto-documenté : nous renvoyons à `./scor -h` pour le synopsis de la commande. Le programme se présente sous la forme d'un *shell*, dans lequel la commande `help` fonctionne.

Il faut créer les tâches qui gèrent les robots avant de lancer l'intelligence artificielle. Lancer cette dernière, toutefois, lancera la tâche acquisition d'images. La séquence suivante fonctionne :

1. `newKhep /dev/ttyUSB0`
2. `newKhep /dev/ttyUSB1`
3. `startIA`

4 Architecture

L'architecture à laquelle nous avons abouti est répartie sur plusieurs processus. Les machines à notre disposition gérant plusieurs fils d'exécution en parallèle, cela nous permet de *pipeliner* les traitements et de répartir les attentes bloquantes, rendant les algorithmes plus simples à écrire. Plus précisément, nous avons le découpage suivant :

Le *shell*. C'est l'interface avec l'utilisateur. Il permet de contrôler l'ensemble des autres tâches (les démarrer et les arrêter) et permet également de piloter les robots en leur envoyant des commandes brutes (celles qui sont envoyées sur le ligne série).

L'acquisition vidéo. Comme son nom l'indique, il s'agit de l'interface avec la Webcam, la tâche est responsable de lire l'image proprement dit et de l'analyser, c'est-à-dire en dégager les positions et l'orientation des robots et la position, la vitesse et la direction de la balle.

Le contrôle des robots. Il existe une tâche par robot, qui le pilote. Au plus bas niveau, elle lui envoie des commandes qu'il est capable de comprendre. Nous avions également en projet qu'elle réalise un asservissement en position du robot.

L'intelligence artificielle. Il s'agit de la composante stratégique qui, étant donnés les résultats de l'analyse d'image, produit des ordres pour les robots.

Graphiquement, cela donne la figure 1. Si abstraction est faite du *shell*, l'enchaînement naturel acquisition → intelligence artificielle → robot apparaît.

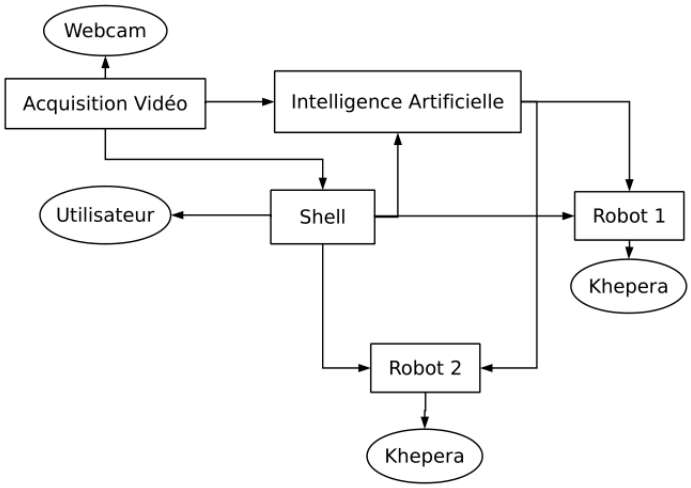

FIGURE 1 – L'architecture en tâches de notre solution.

4.1 Communication entre Tâches

Les communications entre le *shell* et les autres tâches sont réalisées à l'aide de boîtes à lettres, puisqu'elles constituent un canal avec historique. La communication entre l'intelligence artificielle et le contrôleur de robot se fait également par boîte à lettres. La raison initiale était que le destinataire était supposé consommer les messages bien plus vite qu'il ne les recevait. Notre essai d'asservissement nous a montré l'inverse. Nous y revenons plus bas.

Enfin, nous n'avons, cette fois-ci, pas voulu faire d'hypothèse sur la communication entre l'acquisition et l'intelligence artificielle. Puisqu'on ne traite que la dernière image reçue, la communication est réalisée par mémoire partagée dont l'accès est protégé par un sémaphore.

5 Détail des Composants

Dans la partie qui suit, nous allons détailler les différentes parties de ce que nous avons fait, en montrant ce que nous avons essayé d'améliorer par rapport aux années précédentes. Chaque partie dresse également un tableau de ce que nous pensons qui peut être réutilisé. Les références aux fichiers sources sont relatives au dossier *src*, à la racine de l'arborescence du projet.

5.1 L'Acquisition Vidéo

L'acquisition vidéo a, tous comptes faits, représenté la majeure partie de notre travail sur le projet. La problématique à laquelle nous avons essayé d'apporter une solution est la dépendance des algorithmes utilisés jusqu'à maintenant aux couleurs choisies pour identifier tel ou tel élément. En d'autres termes, il est absolumment indispensable que la balle, le terrain et les robots sont de trois couleurs qui soient "primaires" (au sens RGB du terme) et différentes. Nous ne trouvions pas cette contrainte satisfaisante et avons tâché,

sans beaucoup de succès, d'améliorer les choses.

Références aux sources : *Acquisition/video.h* et *Acquisition/video.cpp*. C'est là qu'a lieu l'aggrégation de tous les résultats.

5.1.1 Utilisation de la Webcam

Il a été mentionné plusieurs fois dans les rapports des années précédentes que la faible résolution de la capture vidéo empêchait peut-être d'effectuer des analyses fines de l'image. Nous avons expérimenté cette voie pour réaliser une capture de plus grande taille que les 640 par 480 fournis par défaut. Il se trouve que cette résolution est fixée en dur, dans le fichier *modules/highgui/src/cap_v4l.cpp* des sources de *OpenCV*. Comme il a été mentionné plus haut, l'interface à la Webcam que fournit GNU/Linux est *Video4Linux*. *OpenCV* possède deux interfaces pour utiliser *Video4Linux*, celle qui fixe la résolution (*cap_v4l.cpp*) et une autre (*cap_libv4l.cpp*, dans le même répertoire, qui utilise la bibliothèque *libv4l*. Nous avons tenté d'utiliser *OpenCV* avec cette dernière, sans grand succès, même s'il faut admettre que nous n'avons pas été très tenaces. *OpenCV* supporte d'être utilisé avec des données importées par ailleurs, nous avons donc utiliser l'interface *Video4Linux*, dans sa version 2, directement.

Cette interface est très bien documentée, ce qui nous a permis d'obtenir de bons résultats très rapidement. Entre autres choses, elle permet d'interroger la caméra sur ses capacités, ce qui nous a rendu la vie facile. Le code d'interrogration est encore disponible dans le fichier *Acquisition/CameraHanlder/v4ltest.cpp* des sources de notre projet. La fonction *main()* demande un peu de remaniement, mais les fonctions sont opérationnelles.

Réaliser cette interface nous a permis d'obtenir des images de 1600 par 1200 pixels, même si nous n'en avons sûrement pas encore tiré le meilleur parti.

Utilisation de fichiers. Pour réaliser des tests, il peut être fastidieux d'avoir à utiliser la Webcam, puisque cela implique de se rendre dans la salle, alors qu'il est simple de se constituer une bibliothèque d'images sur lesquelles faire tourner ses algorithmes. Pour répondre à ce besoin, nous avons créé une interface (*ImageProvider*) qui fournit des captures au reste de l'application. Cette interface est instanciée par *CameraHandler* qui utilise la Webcam et par *VirtualCamera* qui utilise des fichiers.

Référence aux sources : *Acquisition/CameraHandler/*. Le code est réutilisable en l'état. Attention toutefois, pour éviter toute surprise, la configuration de la Webcam fixe tous les paramètres dont dépend le reste du programme et lève une exception si l'une d'entre elles n'est pas disponible. En pratique, il y a toutes les chances qu'il soit impossible de construire avec succès une instance de *CameraHandler* sans la Webcam qui équipe les postes de la salle machine.

Pistes d'amélioration. La Webcam est capable de transmettre les données encodées en deux formats : $UYUV$, qui est celui que nous avons utilisé, et $MJPEG$, qui ne tolère que des résolutions inférieures mais que nous n'avons pas testé. *VirtualCamera* pourrait être encore plus utile.

5.1.2 Détection Automatique du Terrain

Cette partie est particulièrement complexe et sujette à l'éclairage de la salle. En effet, la couleur du terrain n'est pas uniforme. Elle est très claire au centre et foncée près des bords. Vous pouvez voir l'image transmise par la caméra sur la figure 2.

FIGURE 2 – L'image de base sur laquelle nous travaillons.

Pour détecter le terrain, nous cherchons à obtenir un masque binaire de l'image où le blanc représente la couleur verte (le terrain) et le noir le reste. Pour ce faire, nous réalisons un seuil sur la couleur verte en HSV (Hue Saturation Value) qui est un espace de couleur équivalent au RGB. C'est ici que les constantes de couleur sont très arbitraires et surtout ne sont pas identiques pour les deux caméras. Il serait bon d'envisager leur détermination de façon automatisée.

Une fois le masque binaire obtenu, une dilatation puis une érosion sont appliquées avec un rayon important pour homogénéiser le terrain car nous avons besoin qu'il ne reste aucun bruit sur celui-ci. Graphiquement, cela donne la figure 3.

Enssuite, nous cherchons le centre approximatif du terrain et nous remontons vers le haut jusqu'à trouver du noir. À ce point précis, nous sommes normalement sur le bord supérieur du terrain (frontière blanc $\rightarrow$ noir). En prenant plusieurs points par bord, on peut trouver un ligne qui correspond expérimentalement assez bien au bord réel tant que le masque du terrain est correct (c'est-à-dire sans trou).

Avec cinq points par bord, nous calculons les équations des quatre droites que suivent les quatre bords grâce à une régression linéaire (méthode des moindres carrés). Nous avons parfois constaté des points aberrants, c'est-à-dire des points qui ne se trouvent pas sur bord du terrain. Cela peut se produire en cas de discontinuité dans ce dernier. Pour rendre notre algorithme plus robuste à des erreurs de ce type, nous calculons en fait l'équation de la droite que suit le bord du terrain pour chaque sous-ensemble de quatre points des cinq points que nous avons positionnés précédemment. Nous sélectionnons l'équation qui minimise la somme des distances ces quatre points à la droite. Le point qui ne fait pas partie de sous-ensemble est donc écarté.

À partir de ces quatre droites, on obtient facilement [1] la position des coins du terrain.

1. enfin presque : les bords du terrain étant presque parallèles aux axes du repère, il faut effectuer une

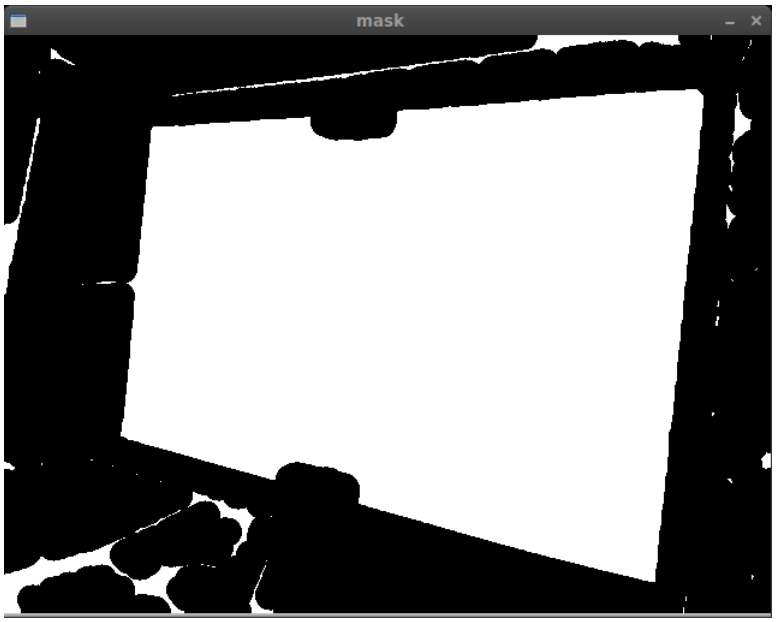

FIGURE 3 – Le masque obtenu par seuillage de la couleur verte.

Les fonctions de *OpenCV* *getPerspectiveTransform()* et *warpPerspective()* permettent ensuite de changer la perspective de l'image pour travailler sur un terrain rectangulaire.

La figure 4 montre les différents traitements appliqués successivement.

5.1.3 Position de la Balle

Pour plusieurs raisons, la détection de la balle ne s'inscrit pas à notre effort de rendre les algorithmes moins dépendants des couleurs. Nous avons repris exactement la même idée que les années précédentes pour détecter la balle : elle est considérée comme se trouvant au barycentre des pixels rouges du terrain. Si, pour un pixel p représenté en RGB, les fonctions $R(p)$, $G(p)$ et $B(p)$ représentent chacune de ses composantes, un pixel rouge est un pixel p qui vérifie :

$$R(p) > G(p) + \text{seuil et } R(p) > B(p) + \text{seuil}$$

En pratique, nous avons pris une valeur de seuil à 100.

Là où le code d'origine réalisait un masque pour obtenir une image binaire, puis réalisait le barycentre sur cette image binaire, nous ne faisons qu'une seule passe en réalisant le barycentre sur les pixels rouges directement.

Référence aux sources : *Acquisition/ballFinder.h* et *Acquisition/ballFinder.cpp*. La classe est tout à fait réutilisable en l'état.

5.1.4 Déplacement de la Balle

La détermination de la vitesse et de la direction de la balle requiert de conserver un historique de points. Nous n'avions aucune idée du nombre d'images que nous pourrions traiter par seconde lorsque nous nous sommes penchés sur cette partie. L'algorithme devait considérer des positions de la balle suffisamment éloignées pour que les calculs restent

rotation sur ce dernier (nous avons choisi $\frac{\pi}{4}$) pour avoir des équations de droite dont les coefficients ne sont ni trop grands, ni trop proches de zéro.

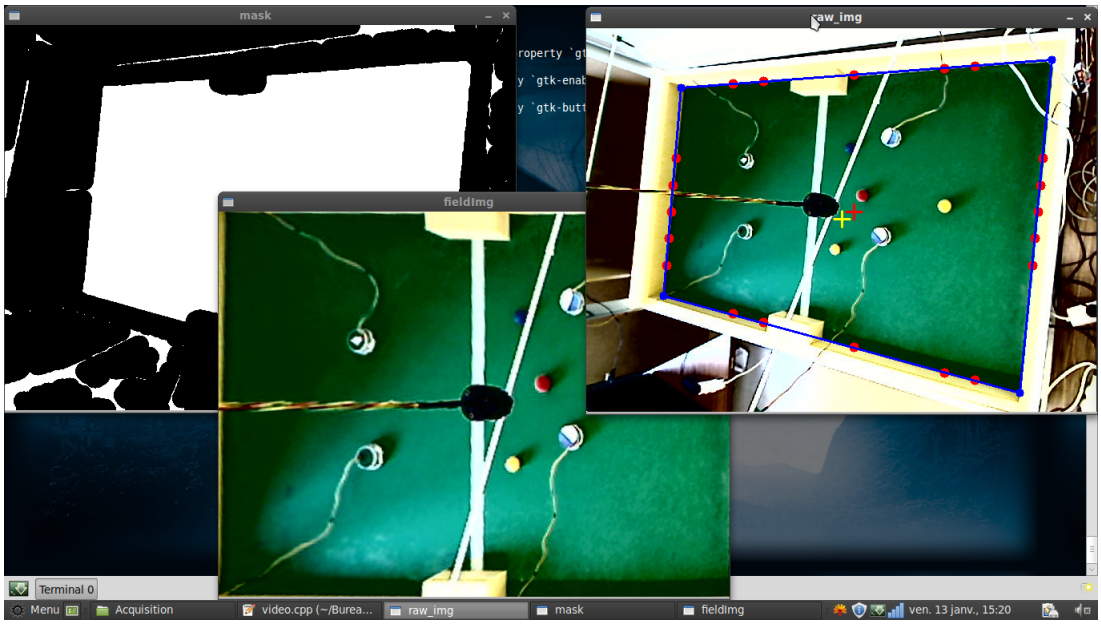

FIGURE 4 – Les différentes étapes de la détection du terrain.

pertinents. Remonter dans le temps pose néanmoins des problèmes dans la mesure où considérer deux positions pour calculer une vitesse, l'une avant un impact et un rebond et l'autre après, va donner des résultats aberrants.

Nous avons utilisé une méthode qui consiste à utiliser deux points pour calculer la direction et la vitesse. Le premier est le position de la balle dans l'image que nous sommes en train d'analyser. Le second est la position de la balle à un instant t dans le passé, distant de moins d'une demie-seconde. En plus de cette condition, ce point p_n doit être tel que, si on nomme p_0 le point dans l'image que nous sommes en train d'analyser, p_1 le point de l'image immédiatement précédente et ainsi de suite jusqu'à p_n, la relation suivante est vérifiée :

$$\sum_{0}^{n-1} p_i\vec{p_{i+1}} = (1+\epsilon)p_0\vec{p}_n$$

L'idée sous-jacente est qu'on s'autorise à remonter dans le temps tant que la balle conserve la même direction. En pratique, nous avons pris $\epsilon = 0.1$.

Référence aux sources : *Acquisition/ballFinder.h* et *Acquisition/ballFinder.cpp*. La classe est tout à fait réutilisable en l'état.

Pistes d'amélioration. L'analyse, pour l'instant, est réalisée sur une représentation *RGB* de l'image. Le fait que la balle soit rouge rend la chose facile. Une méthode plus générique utiliserait une représentation *HSV*, cela fonctionne d'après nos tests avec un algorithme préliminaire. L'approche originale par masque n'apporte pas grand-chose en l'état, mais on pourrait envisager y appliquer une transformée de Hough [2]. La chose aurait l'avantage d'utiliser *OpenCV* et son implémentation a priori efficace des algorithmes. Rechercher explicitement des cercles permet également s'affranchir du bruit éventuel : connecteur rouge, fil rouge dans le câble série, etc..

2. La transformée de Hough permet, dans une de ses variantes, de repérer des disques dans une image.

5.1.5 Position des Robots

Dans le principe, une fois encore, notre solution de repérage des robots est similaire à celle proposée les années précédentes. Elle s'en éloigne toutefois davantage que dans le cas de la balle. Comme les années précédentes, un disque était disposé sur le dessus des robots. Le disque est coloré pour moitié de bleu, et pour moitié de blanc.

Pour commencer, l'analyse est effectuée sur une représentation HSV de l'image. Par visualisation séparée des trois canaux — trois images en niveau de gris — nous avons remarqué que la valeur de saturation permettait de distinguer de façon marquée la moitié blanche des disques. Ce choix s'est imposé de lui-même en raison de la faible distance, suivant l'éclairage, entre les pixels bleus et le pixels verts du terrain.

Une fois le centre des demi-cercles blancs trouvé par *clustering*, on recherche, dans une zone réduite, les demi-cercles bleus à l'aide d'un critère portant sur la teinte et la saturation (supérieure à celle des pixels du terrain). Un barycentre nous donne la position de son centre. Les centres des deux demi-cercles nous donnent le centre du robot, qui pour nous définit sa position, et sa direction, qui est celle du vecteur qui relie les deux points une fois l'avant (le blanc dans notre cas) décidée de l'arrière.

Tout n'est pas si simple. Ca aurait été trop beau. Plusieurs problèmes se posent :

- Dans la version initiale du terrain, les robots étaient reliés à l'ordinateur par un fléau, visible depuis la Webcam et couleur aluminium. La valeur de saturation du gris aluminium est très comparable à celle de la moitié blanche du dessus des robots. Elle apparaît donc sur l'image après le premier filtre. La transformée de Hough nous permet de repérer les longues lignes sur l'image, ce qui inclut le fléau, et nous donne les coordonnées de leurs extrêmités. Il ne reste plus qu'à les retirer en dessinant dessus.
- La bande d'adhésif au milieu du terrain ne colle plus très bien et, lorsqu'elle se décolle, elle apparaît blanche à la Webcam, avec l'effet que l'on devine. La technique précédente à base de transformée de Hough ne fonctionne pas puisque seules les extrêmités de l'adhésif se détache et ne forment pas une ligne. La solution que nous avons retenu est de réduire la zone de recherche à notre moitié du terrain en commençant juste après la bande d'adhésif. Cette solution, toutefois, n'est pas satisfaisante puisqu'elle ne permet pas de repérer les robots ennemis. Puisque la bande est fixe, on pourrait envisager dessiner dessus de manière inconditionnelle.
- Peu avant la fin du projet, le terrain a été modifié et les fléaux mentionnés ci-dessus ont été supprimés. Les ont remplacés de petits dérouleurs fixés avec le même adhésif qui apparaît blanc. De nouveaux points indésirables apparaissent sur le premier filtre. Nous n'avons pas eu le temps d'envisager des corrections au problème.

Autres essais. Avant d'aboutir à la solution décrite ici, nous avons eu l'occasion d'essayer plusieurs autres algorithmes. Voici un résumé de nos tentatives infructueuses :

Détection d'arêtes. Nous avons essayé d'utiliser l'algorithme de détection d'arêtes de Canny (*Canny edge detector*) dans l'espoir de voir se dessiner les robots sous la forme d'un cercle traversé par un diamètre. Nous sommes parvenus à cela, mais avec beaucoup de bruit autour (le fléau, la balle, parfois des reliefs du terrain). La transformée de Hough semble nécessité des disques plutôt que des cercles, nous n'avons pas pu l'utiliser. De là, aucune piste ne s'est dégagée.

Transformée de Hough. Plutôt que de calculer sur des *clusters* sur une image de laquelle ont été enlevés un certain nombre de parasites (le fléau, notamment), nous avons essayé de réaliser les deux filtrages présentés au début de cette longue partie,

puis d'additionner les deux images binaires résultantes. L'idée ici était de reconstituer les cercles de robots pour les repérer à l'aide de la transformée de Hough, nous affranchissant ainsi d'enlever les parasites longilignes. Cela est peut-être dû à une mauvaise compréhension de l'algorithme, mais nous avons obtenu de nombreux faux positifs. Peut-être est-il possible de s'en débarraser ensuite, par exemple en cherchant des lignes dans les cercles obtenus pour vérifier que la ligne de séparation entre blanc et bleu s'y trouve bien. Nous ne sommes toutefois pas allés si loin et avons abandonné la piste.

Résultat des courses. La motivation de tout ce travail était, à l'origine, de trouver une méthode d'identification des robots plus robuste aux variations d'éclairage, permettant également de s'affranchir de la couleur bleue. Nous ne sommes pas spécialistes de l'analyse d'image, mais nos quelques essais n'ont pas vraiment abouti à quelque chose de fantastique. De façon assez ironique, la méthode utilisée les années précédentes semble avoir été plus robuste aux adaptations du terrain réalisées en fin de projet.

Références aux sources : *Acquisition/kheperaFinder.h* et *Acquisition/kheperaFinder.cpp*. Ce code est réutilisable, à voir si conserver cette méthode d'analyse apporte réellement quelque chose.

Pistes d'amélioration. Dans son état actuel, l'algorithme de détection des robots est responsable la majorité du temps nécessaire à l'analyse d'une image. Bien que nous ne l'ayons pas testé, nous supposons que cela est, en grande partie, dû aux allocations dynamiques de grande taille (plusieurs mégaoctets) que réalise l'algorithme. Dans le détail, à chaque fois qu'il est appelé, des nouvelles matrices, qui représentent, les images sont allouées, traitées, puis libérées. Puisque nous n'analysons qu'une image à la fois, il est tout à fait possible de faire les allocations lors de l'initialisation de la tâche. La raison d'un tel comportement de l'algorithme est le fait qu'il était ainsi plus facile à écrire et à modifier pour réaliser des essais (la gestion de la mémoire est transparente lorsqu'on utilise l'interface *C++* de *OpenCV*) et que nous n'avons pas eu le temps de le reprendre une fois qu'il a été stabilisé. Y a-t-il toutefois besoin d'un gain de performance à ce niveau, la question reste ouverte. Il pourrait être intéressant d'expérimenter les techniques de repérage à base de transformée de Fourier, du type réalisé en 3IF pour repérer des caractères dans une image.

5.2 Le *Shell*

Dans notre approche itérative, nous avons commencé par écrire les composants de plus bas niveau avant de s'atteler aux suivants. Pour ce faire de façon constructive, il a été nécessaire de mettre en place l'infrastructure nécessaire au test. À ceci s'ajoute un constat d'échec lorsque nous avons voulu lancer pour essai le programme des vainqueurs de l'année précédente : nous ne savions pas comment l'exécuter et son exécution sans paramètre a abouti à une erreur de segmentation.

Nous avons donc conçu une interface de type *shell* pour contrôler individuellement les fonctionnalités de notre programme. Ce *shell* se veut en partie auto-documenté, à l'aide du drapeau `-h` que supporte l'exécutable et de la commande `help` de notre *shell*, une fois celui-ci lancé. Le *shell* est la seule interface au programme et, dans notre conception initiale, devait être seul à contrôler l'affichage dans le terminal. En pratique, la convention a subi des exceptions pour les traces de mise au point. Le *shell* est également scriptable.

Références aux sources : *Shell/*. Il est globalement réutilisable en l'état. Attention toutefois, pour ajouter ou supprimer une commande, il est nécessaire de modifier la déclaration (*Shell/Shell.h*) et la définition (*Shell/Shell.cpp*) du tableau *cmds*.

Pistes d'amélioration. En l'état actuel, le *shell* est une grosse classe. Son implémentation se répartit sur plusieurs fichiers, mais il est mono-bloc au niveau logique. Il pourrait être intéressant de mettre dans des classes séparées, dont des instances seraient membres de *Shell*, les méthodes de gestion de l'acquisition vidéo, de l'intelligence artificielle et les contrôleurs de robot. Une amélioration utile serait de corriger les brèches de l'encapsulation du contrôle du terminal par le *shell* en mettant au point un mécanisme de journal, dans des fichiers par exemple. Cela rendrait aussi plus exploitables les traces. Enfin, il faut pour l'instant lancé la tâche concernée pour tester une fonctionnalité (par exemple, lancer la tâche acquisition pour analyser des images). Pour tester une fonctionnalité bien précise, une exécution multi-processus n'est sans doute pas nécessaire. S'il existe un moyen simple de pouvoir lancer des méthodes directement depuis le *shell*, il pourrait être intéressant de l'exploiter. Utiliser un vrai outil pour l'analyse syntaxique des commandes est résolumment nécessaire.

5.3 Le Contrôle du Robot

La première chose que nous avons faite sur ce projet a été de faire en sorte de communiquer correctement avec le robot. Nous avons pour ce faire corriger quelques erreurs dans le paramétrage de la ligne série qui sert à dialoguer avec le robot. Cela a abouti à la classe *Khep* (*Khep/Khep.h*) que nous n'avons pas eu besoin de retoucher par la suite.

À la fin du projet, nous avons décidé de mettre un peu d'intelligence dans cette tâche et de la rendre capable d'asservir en position le robot. Par manque de temps, nous n'avons pas trouvé de solution à ce problème. Là se trouve d'ailleurs la raison du non-aboutissement de notre projet. Entre autres choses, nous avons été confrontés au problème du temps : le calcul des consignes au vu des résultats de l'analyse d'image est bien trop rapide par rapport aux déplacement des robots.

Références aux sources : *Khep/*. Dans ce dossier, seuls *Khep.h* et *Khep.cpp* sont réutiliables en l'état.

Pistes d'amélioration. Ce n'est pas réellement une amélioration : il faut, dans nos plans, être capable d'asservir le robot en position.

5.4 Intelligence Artificielle

L'intelligence artificielle est relativement basique et est synchronisée via un sémaphore avec l'acquisition pour effectuer un traitement après chaque acquisition. À chaque capture, elle copie le contenu de la mémoire partagée qui contient la position, la vitesse et la direction de la balle, ainsi que la position et l'orientation des robots, puis elle établit une stratégie et transmet ses ordres aux robots.

La stratégie est établie sur deux critères :

- la position de la balle (est-elle dans notre demi-terrain ou non ?)
- la direction de la balle (va-t-elle vers nous ou vers les buts adverses ?)

Elle consiste en deux actions envisageables :

- se replacer (les robots reprennent leur position d'attente)
- *shooter* (essayer tout du moins) dans le balle

Le *shoot* correspond en fait à intercepter et ensuite tirer dans la balle pour le robot le plus proche (le second allant se replacer).

Pour transmettre ses ordres aux robots, l'intelligence artificielle utilise la classe *RobotIA* pour calculer les points que les robots doivent atteindre (en coordonnées absolues) puis transmet ce point ainsi que la position et l'orientation du robot concerné à la tâche qui le contrôle via une boîte aux lettres.

6 Bilan

La fin du projet a amené son lot de déception, puisque nous ne sommes pas parvenu à aboutir à une solution fonctionnelle. Cela découle de notre idée de vouloir repartir de rien, ou presque, et aboutir à des solutions robustes. Nous avons donc fait beaucoup d'exploration, dont une certaine partie n'a toutefois pas été vaine. Nous espérons également que certains morceaux de notre travail pourront servir de blocs de base pour des projets futurs, c'est en tout cas dans cette optique qu'ils ont été écrits.

La fin de projet que nous avons constatée est le résultat des choix de gestion de projet que nous avons fait. C'est là un résultat inattendu de ce projet qui prétendait ne pas nous charger avec la gestion de projet chère aux projets de conception 5IF. Gérer un projet sur une durée importante (nous avons commencé tôt) est l'occasion de se casser les dents, ce qui est bien plus formateur que de se laisser guider comme c'est d'ordinaire le cas.

Techniquement le projet a été l'occasion de nous initier à la vision par ordinateur, à *Video4Linux* et à l'interface du noyau Linux pour les lignes série. Il a été l'occasion d'architecture un projet de taille relativement importante et de le réaliser.

Notre organisation du travail a été assez libre : chacun a travaillé sur la partie qui l'intéressait, avec un recadrage régulier. La branche principale du projet a toutefois été administrée exclusivement pas le chef de projet, ce qui a permis de garder une certaine cohérence de l'ensemble. Enfin, notre méthode de développement nous a permis de toujours conserver un ensemble fonctionnel pour un sous-ensemble de fonctionnalités : la partie génie logiciel a plutôt bien fonctionné.