

Eurobot06 : Couche Haute



Titre du projet

Eurobot06 : Couche Haute

École

HE-ARC ingénierie informatique

Chef de projet

Matthieu Amiguet

Date

20 janvier 2006

Candidat

Batiste Bieler
Julien Senn

Numéro du projet

i05d19

Abstract

This project relates to the creation and the simulation of the artificial intelligence of a robot for the Eurobot06 competition. The challenges of this project were the real time systems, industrial vision, and team work. We have created an AI, an environnement simulator and an 3D display program of the environnement.

Résumé

Ce projet porte sur la création et la simulation de l'intelligence artificielle d'un robot devant participer à la compétition Eurobot06. Les défis de ce projet furent les systèmes temps réel, la vision industrielle, et le travail en équipe. Nous avons donc créé une IA, un simulateur de l'environnement du robot et un afficheur 3D pour de cette environnement de simulation.



Table des matières

1. Introduction.....	5
1.1. Définition du projet.....	5
1.2. Organisation du projet.....	5
1.3. Solutions développées.....	5
1.4. Résultats obtenus.....	6
2. Analyse.....	7
2.1. Cahier des charges.....	7
2.2. Temps réel.....	7
2.2.1. Concurrence.....	7
2.2.2. Threads.....	7
2.3. Communication entre les couches.....	8
2.3.1. Communication interprocessus.....	8
2.3.2. Communication via des objets partagés.....	9
2.4. Choix du langage et de l'architecture.....	9
2.4.1. Linux.....	9
2.4.2. Python contre Java.....	9
2.5. Construction du robot.....	10
2.6. Besoin d'une architecture souple.....	10
2.7. Retour sur expérience.....	10
3. Conception.....	12
3.1. Outils utilisés.....	12
3.1.1. Codage.....	12
3.1.2. UML.....	12
3.1.3. Graphique.....	12
3.1.4. Présentation/PV.....	12
3.1.5. Vidéo.....	12
3.1.6. Rapport.....	12
3.2. Décomposition en diagramme de classe.....	12
3.2.1. Version 1.....	12
3.2.2. Version 2.....	13
3.2.3. Version 3.....	13
3.3. Interface de mémoire partagée.....	14
3.3.1. Objet de position.....	14
3.3.2. Interface de communication.....	15
3.4. Décomposition des classes.....	16
3.4.1. Manager.....	16
3.4.2. Strategy.....	18
3.4.3. Command.....	20
3.5. Mécanisme d'arbitrage.....	20
3.6. Stratégie.....	21
3.6.1. Stratégie de catastrophe.....	21
3.6.2. Stratégie de bas niveau.....	21
3.6.3. Stratégie de haut niveau.....	21
3.6.4. Stratégie d'évitement.....	21
3.7. Simulateur.....	21
3.7.1. Simulateur Java.....	21
3.7.2. Afficheur OpenGL/C.....	22
3.8. Filtre et mémoire.....	23



3.9. Autres concepts étudiés.....	23
3.9.1. Reconnaissance du monde par nuage de points.....	24
3.9.2. L'intersection entre deux cercles.....	24
3.10. Retour sur expérience.....	24
4. Réalisation.....	26
4.1. Problèmes rencontrés.....	26
4.1.1. Positionnement.....	26
4.1.2. Intelligence du logiciel.....	26
4.1.3. Mécanisme d'arbitrage.....	27
4.1.4. Boucles et récurrences.....	30
4.2. Retour sur expérience.....	30
5. Conclusion.....	31
5.1. Améliorations.....	31
5.2. Apports personnels.....	31
6. Bibliographie.....	32
7. Table des illustrations.....	

1. Introduction

1.1. Définition du projet

Le but du projet est de concevoir et d'implémenter l'intelligence artificielle d'un robot qui va défendre les couleurs de notre école lors du concours Eurobot 06

Le but du concours sera de faire jouer les robots au golf. Notre robot doit être capable de prendre des balles et les mettre dans les trous correspondant à sa couleur (bleu ou rouge)

Cette intelligence représente la couche haute de l'architecture informatique qui est divisée en trois parties distinctes

- Couche Haute : Stratégie, décisions, IA, représentation du monde ;
- Couche Intermédiaire : Intégration des fonctions de fonctionnement du robot, interfaçage entre les couches ;
- Couche Basse : Interfaçage avec le matériel (moteurs, capteurs).

1.2. Organisation du projet

Pour un projet de cette importance, avec autant d'intervenant et de corps de métier différent, il était nécessaire d'avoir des séances d'information afin de constater l'avancement de chacune des parties. Pour qu'elles soient utiles ces séances ont pris trois formes :

- Les séances Couche Haute : Avec Matthieu Amiguet, Julien Senn, Batiste Bieler ;
- Les séances Informatique : Avec tout les informaticiens du projet ;
- Les séances de Projet : Avec tout les intervenants.

Les informaticiens de ce projet ont utilisé un système de communication *Wikia* afin de centraliser les informations concernant le développement.

1.3. Solutions développées

La solution a été de créer une architecture de décision dynamique grâce à un système d'arbitrage. Cet arbitre choisi le meilleur objectif en fonction de différentes variables.

Cette architecture nous permet d'ajouter ou de supprimer un comportement stratégique au robot. Un tel comportement doit fournir un lot de commande (un objectif) à envoyer au robot et une série de variables qui indiquent les avantages à suivre cet objectif. L'arbitre choisi ensuite dans la liste des objectifs envoyés par les différentes stratégies et choisi le meilleur selon ce qui est important à ce moment précis.

Deuxièmement, afin de pouvoir tester l'effet de nos commandes, il était indispensable de créer un simulateur des couches inférieures. C'est pourquoi nous avons développé un simulateur complet et un afficheur qui permet de d'avoir un retour visuel très précis de l'environnement.

Finalement nous avons améliorés le comportement général du robot en lui ajoutant une mémoire du monde alentour. Cette mémoire est importante car la fréquence à laquelle les

informations nous parviennent est relativement lente. Elle nous permet de savoir où se trouvent les objets du monde qui ne sont pas vus et cela même lors du déplacement de la machine.

1.4. Résultats obtenus

Nous avons créé un système de prise de décision flexible et autonome qui se base sur une représentation du monde. Pour arriver à ce résultat, nous avons créé les outils de débogage nécessaires : Un simulateur des couches inférieurs et un afficheur 3D.

L'intégration avec la couche intermédiaire à l'aide de l'interface définie en début de projet s'est faite sans problèmes. Les délais ont été respectés et les tests concluants.

Il reste du travail dans l'intégration avec le robot final. Nous avons commenté notre code au mieux et réalisé ce rapport dans le but de faciliter la compréhension de notre système aux futurs ingénieurs qui poursuivront notre travail.

2. Analyse

L'analyse du projet a consisté à prendre plusieurs décisions importantes. Nous avons pris le temps d'envisager les alternatives et de d'agir en connaissances de cause. La plupart de nos choix nous ont été dictés par les limitations logicielles rencontrées et par les connaissances et les informations que nous avons en début de projet.

2.1. Cahier des charges

Le but du projet est de concevoir et d'implanter la couche haute (CH), partie intelligente de l'architecture du robot.

Le langage, l'environnement et la plateforme sont à choisir dans l'optique d'une architecture robuste et adaptative :

- Robuste signifie que même en cas de panne de certains éléments, le robot continue ;
- Adaptative signifie qu'il doit être possible de modifier le comportement du robot en fonction de l'adversaire.

Les interactions de la couche haute avec les autres éléments doivent être clairement définies et testées de manière approfondie. L'utilisation des outils suivants, que nous ne connaissions pas, sont venus s'ajouter à notre cahier des charges :

- Utiliser l'environnement de développement Eclipse ;
- Utiliser l'outil de gestion de version Subversion.

2.2. Temps réel

Au début du projet, un choix important est survenu. Fallait-il utiliser un système en temps réel ou pas ?

L'expérience en ce domaine des collaborateurs de Saint-Imier, qui ont participé à Eurobot 05, nous a influencé. En effet, l'année passée, le club a opté pour un système en temps réel sous Linux. Ayant perdu du temps à cause de ce choix, ils nous ont conseillé d'éviter la véritable programmation temps réel.

L'expérience a montré que le temps réel n'est pas une priorité pour la conduite de l'intelligence artificiel du robot.

2.2.1. Concurrence

Les informaticiens de ce projet avaient des connaissances théoriques sur les problèmes de la concurrence. Nous avons du les mettre quelque peu à jour pour les utiliser dans le cadre du langage Java. Nous avons surtout utilisé le modificateur `synchronized`. Nous avons donc besoin de mettre à jour nos connaissances dans ce domaine car elles étaient insuffisantes.

2.2.2. Threads

En ce qui concerne notre couche, nous avons limité l'utilisation des *threadcar* nous n'en avons besoin que d'une seule.

Par contre, nous en avons utilisé plusieurs dans le simulateur pour les déplacements du robot.

L'utilisation des *thread* ne nous a pas posé de problèmes particuliers.

2.3. Communication entre les couches

Afin de pouvoir réunir la couche haute et la couche basse à la fin du projet, il était nécessaire de définir une interface de communication. Plusieurs solutions étaient possibles. Elles étaient liées au type d'architecture logiciel que nous allions choisir :

- Architecture multi-processus ;
- Architecture monolithique.

2.3.1. Communication interprocessus

La première direction du projet était de créer deux processus indépendants. Il a donc été nécessaire d'étudier cette communication. On distingue plusieurs types de communication interprocessus :

- les outils permettant aux processus de s'échanger des données ;
- les outils permettant de synchroniser les processus, notamment pour gérer les sections critiques ;
- les outils offrant directement les caractéristiques des deux premiers (ie : permettant d'échanger des données et de synchroniser des processus).

L'intérêt d'avoir plusieurs processus est de pouvoir utiliser des langages différents dans chaque processus sans aucun problème particulier. De plus les processus étant autonomes, ils peuvent être développés facilement dans des bases de code séparées.

Tubes nommés

Il existe plusieurs solutions standard de communication interprocessus. La plus simple et la plus efficace est celle des tubes nommés. Un tube est un fichier mémoire qui simule une pile FIFO (First In First Out).

Un tube est aussi un canal de communication unidirectionnel. D'un côté du tube on écrit les informations et de l'autre on attend de lire une certaine quantité d'informations. Pour créer une communication bidirectionnel, il faut donc deux tubes nommés.

La lecture dans un tube est bloquante. Le processus qui désire lire une information dans le tube doit donc attendre que l'information soit réellement écrite. Ce mécanisme d'attente permet de facilement réaliser des mécanismes de synchronisation en plus de l'échange des données.

Selon nous, cette solution aurait été viable et efficace si le choix avait été de réaliser un système multi-processus.

Sockets

La principale limitation des tubes c'est que les processus doivent tourner sur la même machine. Les sockets offrent une abstraction pour la communication interprocessus via le réseau. Cette communication est bidirectionnel et synchrone. Les sockets ont tout de même un temps de latence plus élevé.

2.3.2. Communication via des objets partagés

La solution finalement retenue dans ce projet fut de créer un programme monolithique en Java utilisant les *thread*. La communication se faisant via une interface implantant les mécanismes de synchronisation nécessaires. Cette solution est similaire à celle des tubes nommés. Avec le modificateur `synchronized` de Java la plupart des problèmes de synchronisation sont très simples à résoudre.

2.4. Choix du langage et de l'architecture

À l'origine, nous pensions faire un système sur Linux avec plusieurs processus, mais finalement c'est le système Microsoft Windows et le langage Java qui a été retenu. Le logiciel devait être programmé d'un seul bloc en Java avec des *thread* pour gérer les problèmes de concurrence.

2.4.1. Linux

Linux a été rejeté car les bibliothèques de vision créées par LabView ne semblent pas compatibles.

De plus avec certains périphériques (bus CAN, webcam), il était délicat de savoir sans expérience si le matériel est véritablement fonctionnel avec les pilotes de périphériques Linux. Une plus grande expérience en amont du projet serait utile pour savoir si les pilotes Linux sont viables sur ce point.

Linux n'est toutefois pas à exclure des prochains projets si une bibliothèque de vision portable et efficace pouvait être utilisée.

2.4.2. Python contre Java

Python a été proposé comme langage de programmation du projet. Python est un langage qui amène des avantages syntaxiques par rapport au langage Java et des structures de données plus puissantes. Il est prouvé que le langage Python permet de produire un logiciel avec moins de lignes de codes que le langage Java. Ceci permet de gagner du temps et de la lisibilité dans le code.

Python aurait demandé un apprentissage pour certains intervenants du projet. Java étant déjà bien maîtrisé par la plupart, nous avons décidé que cela était un avantage suffisant. Les questions de performances sont également en faveur de Java (Python serait une solution). Rétrospectivement, nous avons constaté que les considérations de performances sont assez peu importante par rapport à ce que demande la partie d'analyse d'image du logiciel. En effet cette partie d'analyse consomme 80% du temps processeur alors que notre IA n'en consomme que 5% maximum.

Java a été également jugé plus utile pour les années futures, car il n'est pas prévu pour l'instant d'enseigner le langage Python dans l'école. Comme Java est plus connu dans le monde du

travail que Python, il serait donc plus vendeur pour un futur poste.

2.5. Construction du robot

À ce stade de l'analyse nous ne savons pas encore quelle serait la forme finale du robot. Certaines fonctionnalités ne seront peut être pas disponible dans le robot final. Voici une petite liste des fonctionnalités évoquées :

- Système de positionnement absolu sur le terrain : Pas sur la version finale ;
- Système de reconnaissance de l'environnement par analyse d'image ;
- Commandes des moteurs en vitesse : Abandonné car dépendant du système de positionnement absolu ;
- De multiples capteurs infra-rouge : Finalement il y en aura assez peu ;
- Détection des contacts : Prévu sur la version finale ;
- Outil de contact avec les totems : Prévu sur la version finale.

Dans cette optique, nous avons du nous concentrer sur les composants dont nous étions sûr de disposer. Nous avons choisi de nous concentrer sur les caméras car c'est avec elles que nous pouvions réaliser un travail intéressant au niveau de l'IA.

2.6. Besoin d'une architecture souple

Nous avons identifié plusieurs points essentiels que nous devons respecter afin de pouvoir fournir une IA fiable :

- Une représentation du terrain de jeu qui prenne en compte les incertitudes ;
- Fiabilité des informations décroissante au cours du temps ;
- Position présumée du robot en utilisant le maximum d'informations (vue, moteurs) ;
- Gérer des informations peu précises et peu fiables ;
- Créer un moyen de simuler un terrain de jeu, le robot et ses différents capteurs ;
- Gérer les cas de blocage (murs, robot adverse, totems) ;
- Éviter les boucles de comportement bloquantes.

Un des grands besoins de l'intelligence artificielle est qu'elle puisse s'adapter aux différents cas de figures et éventuellement aux pannes qui pourraient survenir durant le jeu. Pour ce faire, nous avons identifié des stratégies de déplacement qui utilisent différentes parties du robot :

- Les informations de la mémoire ;
- Les capteurs infra-rouge ;
- Les informations de la caméra de devant et ;
- Les informations de la caméra de devant du dessous.

Ainsi, si l'une des parties matérielle tombe en panne, il reste d'autres stratégies capables de fournir un objectif. La plupart de ces stratégies utilise la représentation du monde. Pour qu'elle soit efficace, il faut que l'odométrie fonctionne. Les moteurs et l'odométrie sont des composants indispensables.

2.7. Retour sur expérience

Le processus d'analyse s'est déroulé bien en amont de la première ligne de code. Nous avons



eu le temps de discuter les différents choix avec les différents intervenants informatique.

En ce qui concerne la couche haute, le choix du langage Java ne nous a posé aucun véritable problème. l'IDE Eclipse nous a certainement permit de gagner du temps. Subversion est un excellent outil pour le travail en équipe.

3. Conception

3.1. Outils utilisés

3.1.1. Codage

- Eclipse SDK 3.1.1
- Borland C++ 5.02
- DevCpp 4.9.9.2
- Subversion 1.2.3
- Subclipse 0.9.102

3.1.2. UML

- Poseidon For UML 3.2
- Umbrello UML 1.5
- Jude 2.4.4

3.1.3. Graphique

- Karbon14 0.2

3.1.4. Présentation/PV

- Microsoft Office 2003
- OpenOffice 2

3.1.5. Vidéo

- CamStudio 2.00

3.1.6. Rapport

- MediaWiki
- FOP (Formating Object Processor) 0.20.5

3.2. Décomposition en diagramme de classe

3.2.1. Version 1

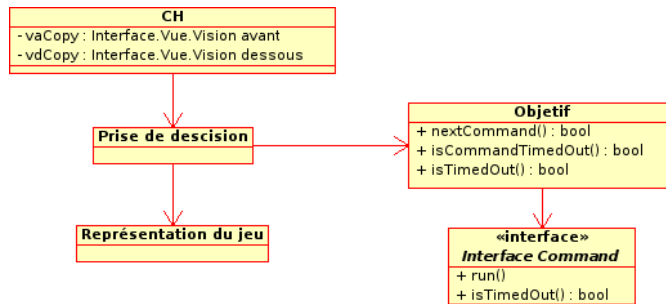


Figure 1 : Première version du diagramme UML

Dans un premier temps, nous avons pensé a un objet de prise de décision qui créait un objectif (suite de commande) en se basant sur une représentation du monde

3.2.2. Version 2

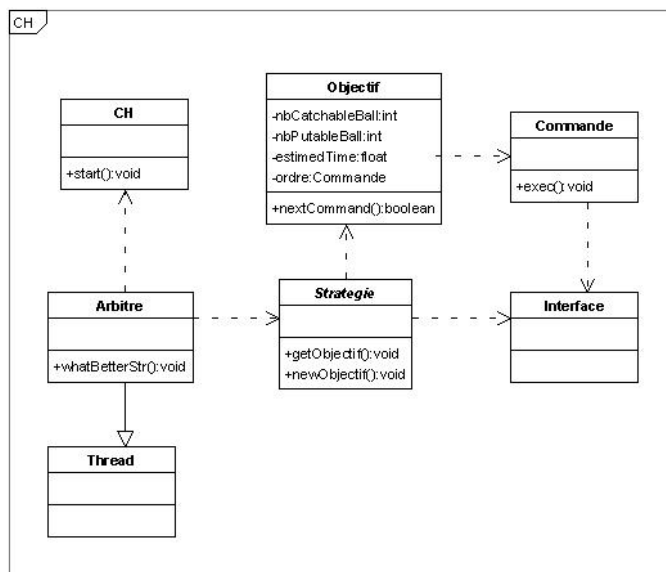


Figure 2 : Deuxième version du diagramme UML

Dans un second temps, l'arbitre choisissait une stratégie (action à exécuter) et cette stratégie créait un objectif avec un lot de commande.

3.2.3. Version 3

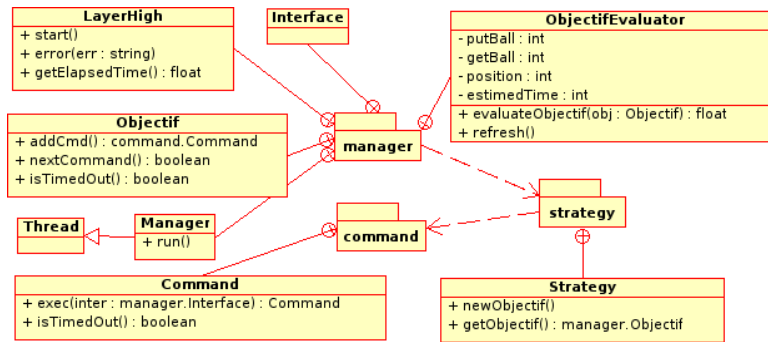


Figure 3 : Troisième version du diagramme UML

Nous avons commencé l'implémentation à partir de ce dernier diagramme. La forme finale du logiciel est relativement semblable. Il y a déjà les trois packages principaux, l'idée des objectifs et des stratégies. Par contre la représentation du monde avait été momentanément abandonnée car jugée trop complexe. Comme il nous restait suffisamment de temps nous avons eu le loisir d'implémenter cette partie pour créer des stratégies de haut niveau.

3.3. Interface de mémoire partagée

3.3.1. Objet de position

Afin d'unifier les traitements entre les couches, nous avons créé une structure de données nommée `ObjectPosition` qui permet de considérer les objets du terrain de jeu comme des spécialisations de cet objet (trous, balles, totems, adversaire). Cet objet contient tous les attributs nécessaires à chaque type d'objet. Il permet également au deux couches de travailler ensemble sans se soucier de la valeur des différents paramètres car ceux-ci sont définis une fois pour toute dans la classe.

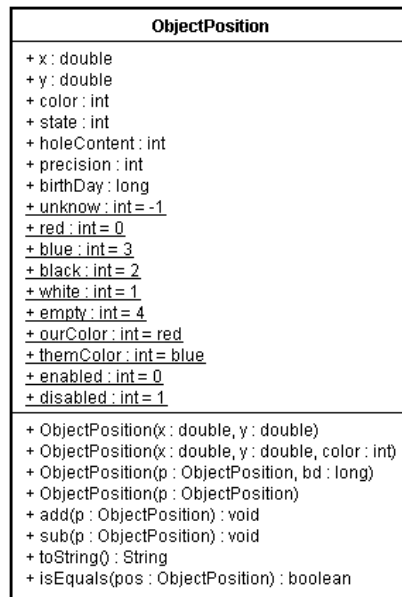


Figure 4 : Fonctions de la classe ObjectPosition

3.3.2. Interface de communication

Pour pouvoir communiquer entre la couche haute et la couche intermédiaire, une interface a été implémentée. Cette interface représente les éléments du robot : les commandes déplacement, les caméras (lointaine et proche), les IRs, le magasin, le retour du déplacement...

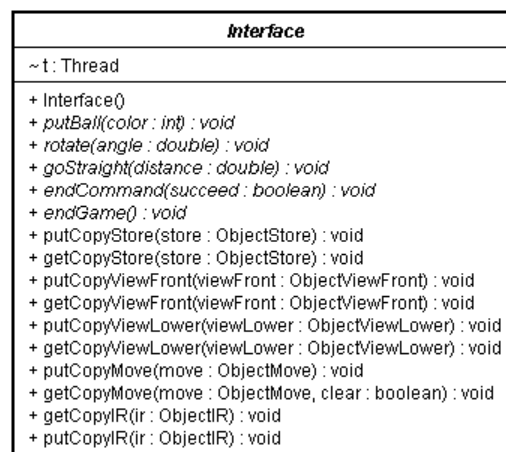


Figure 5 : Fonction de la classe Interface

Deux classes d'interface ont été implémentées, une pour les interactions avec la CI et une pour le simulateur des couches inférieures.

D'autres classes apparaissent dans cette interface, ces classes servent de structure d'échange de données entre la CH et la CI. Ces classes s'appellent `ObjectIR` pour les informations des infrarouges, `ObjectMove` pour les informations de déplacement, `ObjectStore` pour les informations du magasin, `ObjectViewFront` pour les informations de la caméra champ lointain et `ObjectViewLower` pour les informations de la caméra champ proche. Ces classes possèdent toutes une fonction statique de copie synchronisée. Quand cette fonction est appelée dans une instance de classe, les autres instances doivent attendre la fin de la copie pour pouvoir entrer dans la fonction de copie. Cela garanti qu'une copie sera toujours cohérente car la copie d'un objet de données est atomique.

Concurrence dans l'interface

Nous avons utilisé le mots clé `synchronized` pour résoudre nos problèmes. Ce mot clé permet de contrôler l'accès aux fonctions d'un objet ou à l'objet dans sa globalité.

Par exemple, lorsqu'une *thread* veut exécuter une méthode d'instance portant le modificateur `synchronized`, Java verrouille l'instance de classe. Si une autre *Thread* invoque une méthode synchronisée sur ce même objet, elle sera bloquée tant que le verrou ne sera pas levé, c'est à dire quand la première fonction aura terminé son travail.

De la même façon, si une *Thread* veut exécuter une méthode de classe `static` portant le modificateur `synchronized`, Java pose un verrou empêchant un autre *Thread* de solliciter une méthode synchronisée de la même classe. C'est cette méthode que nous avons utilisée pour réaliser la copie de nos objets de données entre les deux couches.

3.4. Décomposition des classes

La décomposition des classes ce fait entre trois différent *packages*:

- `Manager` : Prise de décision ;
- `Strategy` : Les stratégies ;
- `Command` : Les commandes.

3.4.1. Manager

Le *packages* `Manager` regroupe les classes d'arbitrages et la représentation du monde.

LayerHigh

`LayerHigh` est la classe de lancement de la couche haute, c'est elle qui lance le début du jeux et le mécanisme d'arbitrage. Elle la classe racine du projet, elle contient toute les autres classes.

Manager

C'est cette classe qui contient le mécanisme d'arbitrage. Cette classe est une *Thread* qui va devoir tourner pendant toute la durée du jeu. Elle va déterminer si il est nécessaire de changer

de stratégie, de lancer une nouvelle commande, ou d'attendre la fin de l'exécution d'une tâche.

ObjectMemory

Dans cette classe se trouve une copie de tous les objets présent dans l'interface, elle permet d'économiser des copies inutiles et de garantir la cohérence des informations pendant tout un cycle de l'IA.

Orders

`Orders` est la classe de gestion des objectifs, les objectifs sont des suites de commandes qui doivent s'exécuter consécutivement. L'exécution d'une commande doit attendre la fin de la précédente avant de pouvoir être exécutée. C'est le rôle de la classe `Manager` de contrôler le bon fonctionnement de ce processus.

PositionEvaluator

Dans cette classe se trouve des outils de géométries simple comme le théorème de pythagore et les matrices de rotations. Nous utilisons principalement ces fonctions dans la classe `WorldMemory`. La plupart des fonctions fonctionne directement avec des `ObjectPosition` en entrée.

StrategyEvaluator

C'est cette classe permet d'évaluer un objectif en fonction de différents critères (nombre de balles mises, avantage positionnel, etc). `Manager` l'utilise afin de différencier les différents objectif possibles et de choisir le meilleur.

WorldMemory

`WorldMemory` contient la représentation du monde : la positions des trous, des balles, des totems, etc.

Elle offre également des filtres qui permettent de sélectionner uniquement des éléments valides ou qui se trouve dans une certaine trajectoire. Elle mémorise également l'emplacement des totems, qui sont considérés comme des obstacles. Cette classe est en fait le point névralgique de l'intelligence du robot. Sans elle le robot se bornerait à utiliser ce qu'il voit dans l'immédiat, c'est à dire pas grand chose.

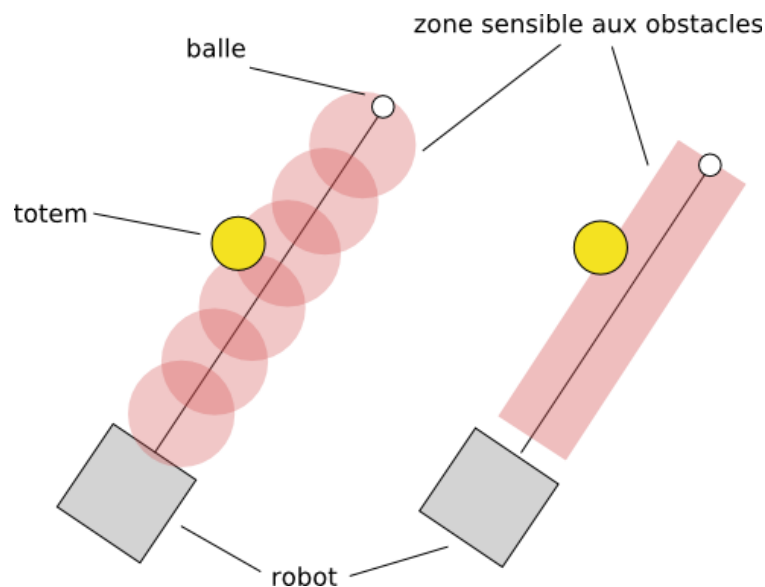


Figure 6 : Illustration du mécanisme d'évitement des obstacles

Si un objet considéré comme un obstacle (dans cet exemple un totem), est présent sur la trajectoire du robot, on ne peut pas se diriger vers la cible. Cette cible est donc considérée comme non atteignable.

La zone sensible dans la mémoire à été implémentée selon la première méthode. les obstacles sont détectés par une suite de cercle. Cette méthode approximative fonctionnant très bien dans nos tests nous avons jugés qu'il n'était pas nécessaire d'implémenter la deuxième méthode qui est plus correcte et qui, à notre avis, consommerait moins de ressources processeurs.

3.4.2. Strategy

Le `packagesStrategy` contient toutes les stratégies que nous avons implémenté, une stratégie sert à créer un lot de commandes en fonction de l'état du jeu. Les stratégies utilisent le plus souvent les filtres et les informations disponibles dans `WorldMemory`.

Strategy

`Strategy` est une classe abstraite de stratégie. Toutes les stratégies dérive de cette classe.

StEscBehindIR

Stratégie d'évitement utilisant le capteur infrarouge arrière. Quand le capteur atteint une limite inférieure à une certaine valeur, un évitement est commandé. Il s'agit d'un avancement de 10 cm.

StEscFrontIR

Stratégie d'évitement utilisant le capteur infrarouge arrière. Quand le capteur atteint une limite

inférieur à une certaine valeur, un évitement est commandé. Il s'agit d'un recul de 15 cm.

StEscSideIR

Stratégie d'évitement qui provoque une rotation de côté en fonction des capteurs latéraux. Il s'agit d'une rotation de + ou - 60°.

StGoAdvHoleMemory

Cette stratégie d'approche se rend vers le premier trou adversaire en mémoire.

StGoAdvHoleView

Cette stratégie d'approche se rend vers le premier trou adversaire visible.

StGoBallMemory

Cette stratégie ramasse la première balle en mémoire.

StGoBallOptimizedMemory

Cette stratégie ramasse de manière optimisée les premières balles. Elle prend toutes les balles qui sont sur la même trajectoire.

StGoBallView

Cette stratégie ramasse la première balle visible.

StGoBetterPosition

`StGoBetterPosition` se déplace vers un des points stratégiques définis, mais ces points sont enregistrés en absolu sur le terrain donc en cas de dérive grave du robot ces points ne seront pas atteints.

StGoHoleMemory

Cette stratégie d'approche se rend vers le premier trou ami en mémoire.

StGoHoleView

Cette stratégie d'approche se rend vers le premier trou ami visible.

StGoStraight

`StGoStraight` permet au robot de se placer jusqu'au milieu du jeu. Cette stratégie ne doit être utilisée qu'en cas de problème grave pour passer la première ligne.

StLookAround

Stratégie de recherche d'informations. Le robot fait un tour sur soi pour voir des balles ou des

trous.

StPutBlackBall

Cette stratégie permet de mettre une balle noire dans un trou ennemi si il est visible dans la caméra champ proche.

StPutWhiteBall

Cette stratégie permet de mettre une balle blanche dans un trou ami s'il est visible dans la caméra champ proche.

3.4.3. Command

Le *package* `Command` rassemble toutes les commandes à donner au robot(déplacement, mettre une balle, etc).

Command

Classe abstraite de commande. Toutes les commandes dérive de cette classe.

CmdClearMemory

Cette commande nettoie complètement la mémoire. Elle utilise pour cela une méthode de `WorldMemory`. Elle pourrait être utilisée après un contact par exemple.

CmdPutBall

Commande de mise d'une balle dans un trou.

CmdRotate

Commande de rotation selon un angle.

CmdStraight

Commande de déplacement en ligne droite sur une distance donnée.

3.5. Mécanisme d'arbitrage

Le mécanisme d'arbitrage est implémenté dans les classes `Arbitre` et `StrategyEvaluator`. La classe `Arbitre` va parcourir toutes les stratégies à la recherche de celle qui a le plus de poids et parallèlement `StrategyEvaluator` va attribuer un poids a chaque stratégie.

Toutes les stratégies ont des paramètres qui vont influencer sur l'intérêt à suivre cet objectif : nombre de balles mettables dans des trous, nombre de balles attrapables, temps d'exécution, déplacement avec avantage de positionnement.

De son côté `StrategyEvaluator` va multiplier ces paramètre en fonction critère : nombre de

balles déjà attrapées, temps de jeu restant,... pour avoir une valeur de poids la plus plausible possible.

Ensuite l'arbitre va simplement sélectionner la stratégie avec le plus d'intérêt sauf en cas de contact où là seules les stratégies d'évitement seront considérées. Dans ce cas le mécanisme de sélection sera utilisé.

3.6. Stratégie

Les stratégies ont été pensées par couche. Les stratégies les plus simples (ne nécessitant ni caméra, ni capteur) ont été implémentées au début. Ceci afin de ne pas oublier que de nombreux imprévus peuvent survenir pendant le concours et qu'il est important que le robot réagisse si l'un des composants matériel tombait en panne.

3.6.1. Stratégie de catastrophe

- `StGoStraight` (ne nécessite aucun périphérique)(peu d'intérêt).

3.6.2. Stratégie de bas niveau

- `StLookAround` (nécessite les IR mais fonctionne sans) ;
- `StGoBetterPosition` (nécessite l'odométrie).

3.6.3. Stratégie de haut niveau

- `StGoBallMemory` (nécessite une représentation du monde évolué) ;
- `StGoHoleMemory` (nécessite une représentation du monde évolué) ;
- `StGoAdvHoleMemory` (nécessite une représentation du monde évolué) ;
- `StGoBallOptimizedMemory` (nécessite une représentation du monde évolué).

- `StGoBallView` (nécessite la webcam champ lointain) ;
- `StGoHoleView` (nécessite la webcam champ lointain) ;
- `StGoAdvHoleView` (nécessite la webcam champ lointain) ;
- `StPutBlackBall` (nécessite la webcam champ proche) ;
- `StPutWhiteBall` (nécessite la webcam champ proche).

3.6.4. Stratégie d'évitement

- `StEscBehindIR` (nécessite les IR) ;
- `StEscFrontIR` (nécessite les IR) ;
- `StEscSideIR` (nécessite les IR).

3.7. Simulateur

La simulation de la couche haute c'est découper en deux parties distinctes, la simulation de la couche intermédiaire en Java et un afficheur en OpenGL.

3.7.1. Simulateur Java

Créer un simulateur s'est révélé un choix très judicieux car comprendre le comportement du robot sans retour visuel est laborieux voir impossible. Le simulateur nous a permis de détecter une multitude de comportement non désiré qu'il aurait été très difficile à prévoir. Notamment les problèmes de boucle où le robot répète inlassablement la même action annulée par un contact.

De plus, il a été nécessaire de créer une autre couche intermédiaire pour le simulateur. En effet, le but du simulateur étant de voir les actions de la couche haute, il était nécessaire de simuler complètement une couche intermédiaire.

La simulation de la couche intermédiaire pour l'afficheur graphique se trouve dans le package simulateur. Dans ce package se trouvent plusieurs classes :

- Les classes de gestion du simulateur (`Simu` et `RobotDisplayThread`) ;
- La classe de lancement de l'OpenGL (`InitGl`) ;
- Les classes de simulation des actions mécaniques (`LayerIntermediaryMove`, `LayerIntermediaryRotate` et `LayerIntermediaryPutBall`) ;
- Les classes de pont JNI (`tableDisplay` et `tableDisplayJNI`).

3.7.2. Afficheur OpenGL/C

Après la réalisation du simulateur, il nous fallait pouvoir afficher l'état du monde tel qu'il était dans le simulateur. Pour cela nous avons créé un petit logiciel qui nous permet de visionner et de guider le robot adverse sur la table de jeu.

L'afficheur a été réalisé avec la technologie graphique OpenGL et a été lié au simulateur Java grâce à un connecteur JNI (Java Native Interface). SWIG (Simplified Wrapper and Interface Generator) a été utilisé dans cette optique.

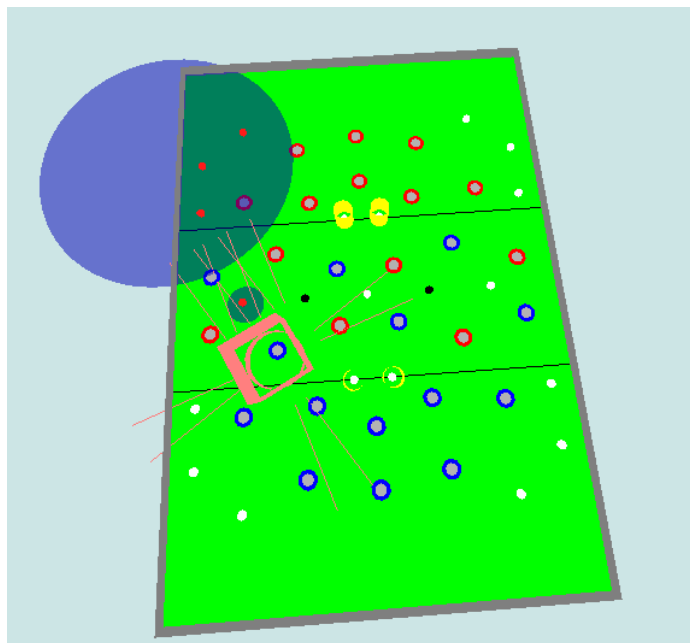


Figure 7 : Capture de l'affichage du simulateur

Le projet OpenGL est intégré au projet Java bien que seul la dll est utilisée, il est situé dans le dossier afficheur_simulateur. Pour créer cet afficheur, nous avons utilisé le canevas standard d'infographie utilisé lors des cours de M.Chatelain en 3e année.

Ce projet comporte les bibliothèques de fonctions suivantes:

- `displayTable.c` qui contient les fonctions d'entrée/sortie de la dll ;
- `displayTable_wrap.c` qui permet d'utiliser les fonctions de `displayTable.c` en Java ;
- `geometrie.c` qui contient les fonctions de dessins de l'afficheur ;
- `callistes.c` qui contient le modèle des trous rouges et bleus ;
- `event.c` qui contient la gestion des événements ;
- `global.c` qui contient les variables globales au projet.

3.8. Filtre et mémoire

Première idée préliminaire pour un filtre au niveau de la représentation du monde :

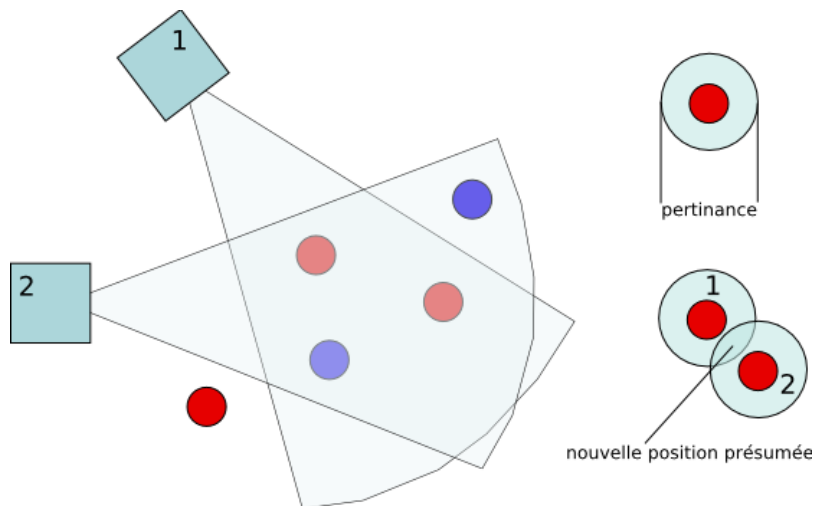


Figure 8 : Schéma d'un filtre mémoire pour la vue

Pour gérer les imprécisions de la vue et la dérive de l'odométrie, nous avons utilisé des filtres qui mettent la mémoire à jour intelligemment. Quand un trou est vu, on compare sa position à ceux de la mémoire. Si l'un d'eux correspond à notre trou, c'est à dire qu'il se trouve dans un périmètre relativement réduit, l'information est mise à jour dans la mémoire. Si ce n'est pas le cas, c'est qu'il s'agit d'un trou jamais observé. Un nouveau trou est alors créé dans la mémoire.

Cette méthode fonctionne bien en simulation. Nous n'avons pas encore testé cet algorithme en situation réelle et notamment avec un taux de rafraîchissement de la vue réaliste. Il serait intéressant de simuler cela pour voir si ça fonctionne toujours.

3.9. Autres concepts étudiés

3.9.1. Reconnaissance du monde par nuage de points

Il existe une méthode intéressante pour découvrir l'environnement matériel du robot, il s'agit de la méthode par nuage de points. On pourrait utiliser les capteurs infra-rouge afin de connaître un maximum de points de contact avec les murs. Ces points devraient être enregistré dans une représentation du monde. Pour que cette méthode soit fiable il faut que le robot sache avec une certaine précision où il se trouve dans l'air de jeu. Si on ajoute un temps de vie aux points de contacts, on peut tolérer une certaine dérive du positionnement ce qui arrive quand on utilise les moteurs et l'odométrie.

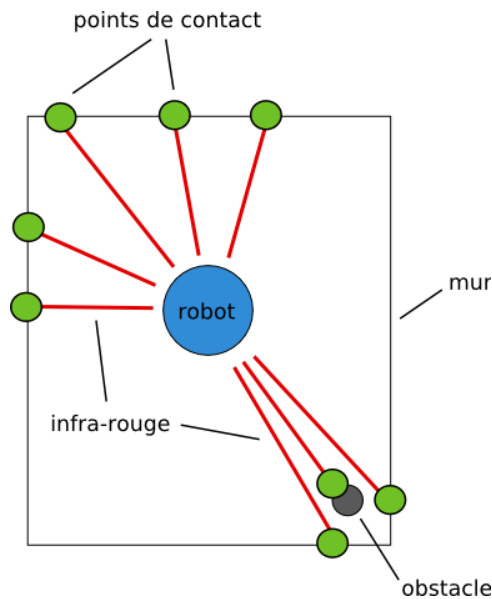


Figure 9 : Schéma de perception par nuage de points

3.9.2. L'intersection entre deux cercles

Un autre algorithme de positionnement a été étudié, il s'agissait de détecter deux totems et en sachant leurs positions, de recalibrer notre position sur le terrain. Le principal problème de ce concept est que la position des totems doit être connue très précisément alors que la caméra champ lointain ne peut pas avoir une précision suffisante. De plus il y a toujours une ambiguïté suivant l'orientation du robot et depuis quelle position le robot voit les totems.

3.10. Retour sur expérience

Le principal problème lors de la conception était le manque d'informations précises sur le robot final :

- Le système de positionnement : Finalement il n'y en aura aucun ;
- L'utilisation d'infrarouge : Nous ne savons toujours à pas quel genre de capteurs nous aurons affaire ni leur nombre ;

- Le nombre de caméra : La deuxième caméra a été ajouté à un stade assez avancé du projet.

Il est évident que nous aurions pu mieux planifier et organiser notre travail si les composants matériels étaient déjà décidé depuis le début, car nous sommes souvent parti sur des pistes que nous avons du abandonner en cours de route.

4. Réalisation

4.1. Problèmes rencontrés

4.1.1. Positionnement

Le positionnement du robot a posé beaucoup de problème. À l'origine, un positionnement absolu était envisagé, plusieurs pistes ont été étudiées :

- Technologie à laser tournant ;
- Balise sonore (sonar) ;
- Boussole et gyromètre ;
- Webcam avec des balises colorés.

Ces recherches n'ont pas abouties, ces technologies auraient coûtés trop chère ou n'étaient pas assez précises.

Finalement la position du robot sera donnée par les moteurs grâce à l'odométrie : Tous les déplacements des moteurs sont enregistrés et permettent de positionner le robot.

Cette méthode peut être efficace mais il faut gérer une certaine dérive du système. En effet les glissements sont inévitables et il est donc impossible de savoir avec précision où se trouve le robot après quelque déplacements. En donnant un temps de vie aux informations de positions des différents objets vus, on peut résoudre ce problème.

4.1.2. Intelligence du logiciel

La partie intelligente de notre logiciel se situe dans le package Manager. C'est là que toutes les décisions sont prises.

- `Manager` va évaluer toutes les stratégies et sélectionner si la stratégie à suivre est une stratégie d'évitement ou pas ;
- `StrategyEvaluator` va donner l'intérêt à suivre cet objectif pour chaque stratégie ;
- `WorldMemory` reçoit et filtre toutes les informations qui parviennent au robot. C'est là qu'est implémentée la représentation du monde ;
- `Orders` est le conteneur des lots de commande.

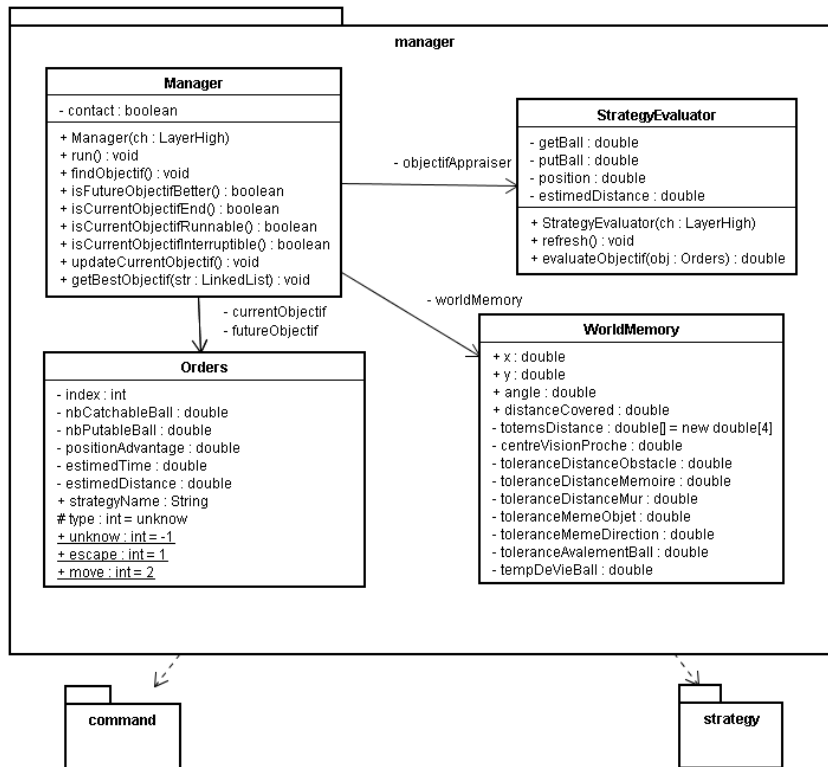


Figure 10 : Contenu du package Manager

4.1.3. Mécanisme d'arbitrage

Le mécanisme d'arbitrage a été une des choses les plus complexes à implémenter. Il était important de faire une architecture souple qui permette de changer d'objectif si il y en avait un autre meilleur ou si un obstacle survenait.

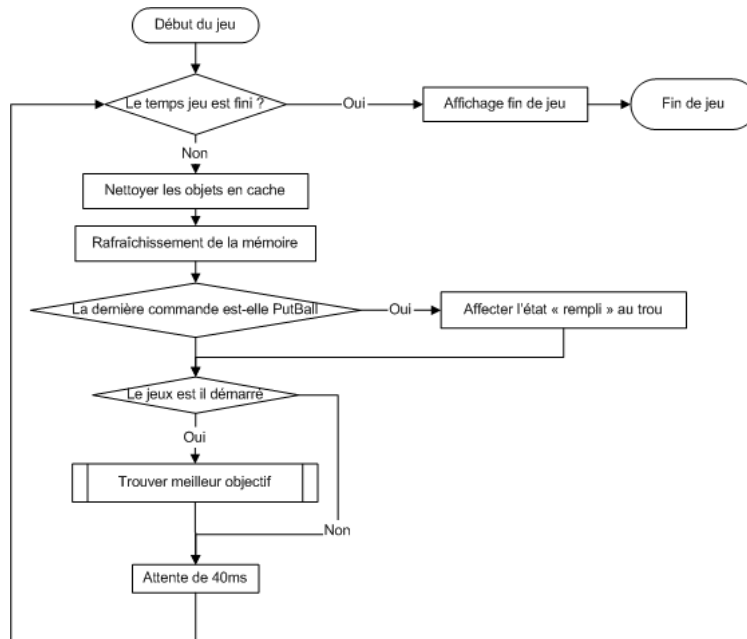


Figure 11 : Diagramme de descision de l'arbitre

Ensuite c'est la méthode d'affectation du meilleur objectif qu'il a fallut implémentée. Le but étant de donner la priorité aux stratégies d'évitement en cas de contact.

Ces problèmes ont été résolus en considérant d'abord toutes les stratégies d'évitement. Si l'une de ces stratégies peut s'appliquer, alors les autres stratégies ne sont pas considérées.

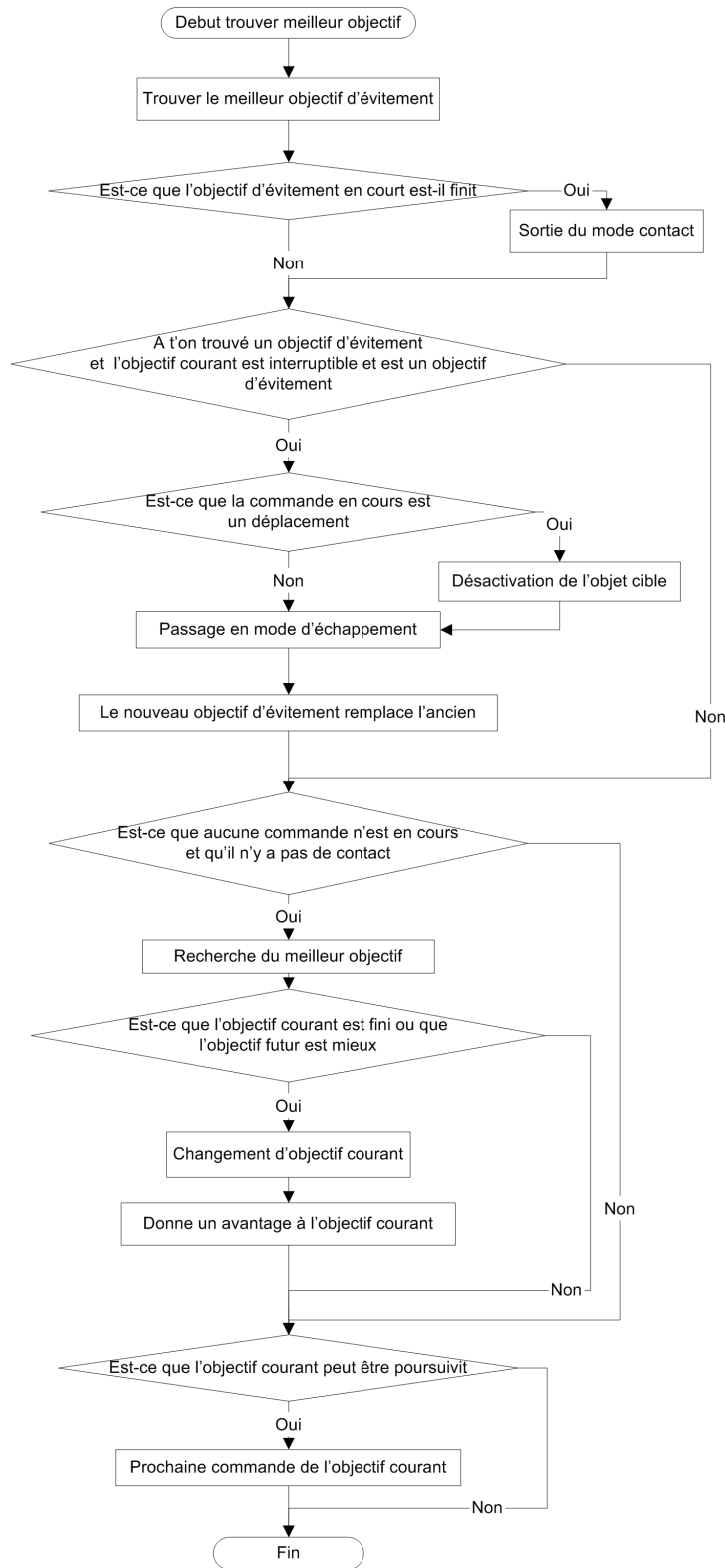


Figure 12 : Diagramme de descision du meilleur objectif

4.1.4. Boucles et récurrences

Nous sommes tombés souvent sur un problème connu en robotique : Les boucles.

Une boucle est une succession de deux ou plusieurs ordres contradictoires, le robot va alors alterner entre ces ordres et se mettre dans un état de boucle récurrente d'où il ne peut plus sortir.

Les causes d'une boucle sont parfois difficiles à déterminer, il est donc difficile de corriger ce genre d'erreur.

Un exemple de boucle est apparu quand il a été question de diviser l'approche d'un trou en deux phases : Un premier déplacement et ensuite un recalibrage avec la camera champ proche pour avoir suffisamment de précision. Le robot étant à proximité du point d'approche, mais pas exactement au bon endroit, il préférerait effectuer une correction minime pour se rapprocher de ce point plutôt que de finir le mouvement jusqu'au trou pour mettre la balle.

Nous avons donc désactivé l'intérêt de réaliser une très petite correction sur les points d'approches car elle ne serait pas utile.

Nous avons trouvé une solution efficace pour éviter toutes boucles : Désactiver la cible dans la mémoire s'il y a une répétition ou s'il y a une annulation par un contact.

4.2. Retour sur expérience

La réalisation s'est déroulée sans problème. Nous conseillons à ceux qui travaillent à niveau de l'IA de ne prêter aucune attention à l'optimisation de leur code. Il faut privilégier la lisibilité à la rapidité. Il est toujours possible d'optimiser par la suite, mais rendre le code plus lisible est autrement plus délicat.

Le goulet d'étranglement en terme de performance ne venait pas notre partie mais de l'analyse d'images de la couche intermédiaire. C'est cette partie qu'il faudrait améliorer car elle consomme plus de 80% des ressources matérielles.

5. Conclusion

5.1. Améliorations

Nous avons identifié certaines parties du logiciel qui pourraient être améliorées facilement :

- Les matrices de rotations peuvent être optimisées pour obtenir de meilleures performances ;
- Il est possible d'utiliser les matrices de rotation dans la détection d'obstacles pour une meilleure précision et une performance accrue ;
- Des boucles bloquantes apparaissent quand on est coincé par le robot adverse. Il faudrait réussir à comprendre pourquoi ces boucles existent encore malgré les mesures de protections prises ;
- Le simulateur peut être encore amélioré pour être plus proche de la réalité. Nous pensons notamment au retard de 300ms du au traitement des images.

5.2. Apports personnels

L'organisation et l'utilisation des outils appropriés au travail en équipe furent des points intéressants du projet. Nous devons faire en sorte que le travail des informaticiens se rejoigne vers la fin du projet. Pour cela de nombreuses sessions de communication ont été organisées. Ce fut l'occasion d'apprendre comment réaliser un compte rendu de l'avancement de notre travail.

Durant ce projet, nous avons appris à utiliser l'environnement de développement Eclipse. Ce logiciel est un logiciel libre avec des fonctions avancées qui facilitent le développement des applications Java. La compilation se fait en tâche de fond, il offre des fonctionnalités qui permettent de contrôler la syntaxe en cours de rédaction et son architecture à base de plugin est efficace. Mention spéciale au plugin Subclipse qui permet à Eclipse de fonctionner merveilleusement avec le gestionnaire de version Subversion.

Nous avons également abordé les problèmes de la programmation concurrente en Java via l'utilisation des *thread*. La gestion de la concurrence apparaît entre les couches, ce qui impliquait une excellente communication entre les intervenants.

6. Bibliographie

- WIKIPEDIA (), *Communication inter-processus*,
http://fr.wikipedia.org/wiki/Communication_inter-processus
- WWW.LALTRUISTE.COM (), *Le modificateur synchronized*,
http://www.laltruiste.com/coursjava/modificateur_synchronized.html
- WWW.LARCHER.COM (), *Les threads Java*,
<http://www.larcher.com/eric/guides/javactivex/VII.htm>
- RIVEILL (Michel), *A la découverte des threads Java*,
<http://rangiroa.essi.fr/cours/systeme1/thread/>
- SWIG (), *Simplified Wrapper and Interface Generator*, <http://www.swig.org/>
- JNI (Sun), *Java Native Interface*, <http://java.sun.com/j2se/1.4.2/docs/guide/jni/>

7. Table des illustrations

Figure 1 : Première version du diagramme UML	13
Figure 2 : Deuxième version du diagramme UML	13
Figure 3 : Troisième version du diagramme UML	14
Figure 4 : Fonctions de la classe ObjectPosition	15
Figure 5 : Fonction de la classe Interface	15
Figure 6 : Illustration du mécanisme d'évitement des obstacles	18
Figure 7 : Capture de l'affichage du simulateur	22
Figure 8 : Schéma d'un filtre mémoire pour la vue	23
Figure 9 : Shéma de perception par nuage de points	24
Figure 10 : Contenu du package Manager	27
Figure 11 : Diagramme de descision de l'arbitre	28
Figure 12 : Diagramme de descision du meilleur objectif	29