Université Grenoble Alpes

Planification Automatique et Techniques d'Intelligence Artificielle

Master 1 - Informatique

Rapport De Projet

Auteurs :
Paul Carretero
Florent Chastagner

Responsable :
Humbert Fiorino

28 avril 2017

Table des matières

1 Introduction

1.1 Introduction

Notre objectif était ici de proposer un système de navigation et de décision pour un robot Mindstorm afin que celui ci recherche et dépose des palets sur un terrain spécifique.

Nous avons développé un programme pour notre robot (Marvin) s'approchant d'une solution complète à ce problème. D'après nos tests notre solution permet en moyenne de récupérer entre 6 et 9 palets, même s'il arrive que certains essais soit moins fructueux que d'autre.

1.2 Lexique

Afin de décrire au mieux les particularité de ce projet, nous employons les terme suivant dans ce rapport et dans notre code :

Pose Position (x,y) et orientation(h) du robot, tel que précisé dans la bibliothèque LeJos.

Item Un objet sur le terrain. Cet objet peut être un palet ou un robot et a des données temporelles.

Map Ensemble des Items.

Odomètre Enregistre les mouvements des roues du robot et estime une Pose en disposant uniquement de ces données.

Objectif Représente ce que le robot souhaite accomplir.

radar Encapsule les données du capteur ultrason.

grab Action de fermeture sur les pinces du robot

2 Utilisation de notre Robot

2.1 Phases d'initialisations

2.1.1 Sélection du mode de lancement

Le programme *main.jar* est le programme de notre projet. Il est configuré pour être lancé par défaut sur notre robot.

FIGURE 1 – Premier menu du robot, sélection du programme

Le premier menu du robot demande à l'utilisateur l'action qu'il souhaite réaliser, les choix sont :

UP ([DEFAULT START]) Lance le programme principal en utilisant le fichier de calibration des couleurs existant

ENTER ([CALIBRATE COLOR]) Lance le programme de calibration des couleurs puis se termine. Cette calibration sera stocker pour être utilisable ultérieurement.

DOWN ([CALIBRATE AND START]) Lance le programme de calibration des couleurs puis lance le programme principal.

2.1.2 Sélection de la position initiale

FIGURE 2 – Second menu du robot, sélection de la position initiale

Ce menu permet de choisir la position initiale du robot. Le 0 en bas à gauche représente le point origine physique sur le terrain. Le robot est symbolisé par un M, déplacable avec les touche UP, DOWN, LEFT et RIGHT. On valide une position avec la touche ENTER.

2.1.3 Lancement : sélection de la configuration souhaité

```
  [SAFE HAND OPEN]

[UNSAFE]  [UNSAFE]
[OPEN]      [CLOSE]

[SAFE HAND CLOSE]
_____________________
Marvin Start Menu
```

FIGURE 3 – Troisième menu du robot, sélection du mode du programme

Le programme demande enfin à l'utilisateur quelle est la configuration initiale des pinces du robot et quel mod il souhaite utiliser. Le mode *UNSAFE* propose des fonctionnalités supplémentaires susceptible d'entraîner des erreurs (les couleurs bleus, vertes et noirs sont fréquemment confondues, donc peu fiable) ou simplement plus de préjudice que de bénéfice (tentative d'interception). Les options possibles sont :

UP ([SAFE HAND OPEN]) Lance le programme principal de manière standard et considère que ses pinces sont en position ouverte.

DOWN ([SAFE HAND CLOSE]) Lance le programme principal de manière standard et considère que ses pinces sont en position fermée.

LEFT ([UNSAFE OPEN]) Lance le programme principal de manière *unsafe* et considère que ses pinces sont en position ouverte.

RIGHT ([UNSAFE CLOSE]) Lance le programme principal de manière *unsafe* et considère que ses pinces sont en position fermée.

2.2 Phase autonome et fin

Une fois lancé le robot fonctionnera de manière autonome. Il tentera de récupérer et déposer des palets durant 5 min. Une fois ce délai écoulé ou s'il n'y a plus de palet détecté/détectable alors il se terminera proprement.

2.3 *TrillianServer*

Notre robot n'utilise pas la sortie console pour écrire les logs et informations destinées à l'utilisateur *une fois lancé uniquement*. Il envoie les principaux messages de log vers une IP définie dans la variable Main.IP.

```
public static final String IP = "192.168.0.22";
```

FIGURE 4 – exemple de configuration de l'adresse IP du serveur

Le programme autonome *TrillianServer.java* dans le package *TrillianServer* propose une interface client afin de visualiser ces messages. Il suffit de le lancer, il se terminera automatiquement une fois que le robot se termine normalement. Il utilise des codes ANSI afin de proposer une couleur différente par service.

2.4 Exceptions

Malgré les vérifications réalisées, il est possible que le programme ne s'exécute pas normalement. Dans le cas ou le robot se *perd* sur le terrain il est possible de l'interrompre a tout moment en appuyant sur la touche *échap*.

Il existe une erreur relativement fréquente au niveau de l'initialisation du capteur ultrason. Nous pensons qu'il s'agit d'un défaut matériel ou de la bibliothèque. Dans le cas ou une Exception de type *InvalidSensorMode* est soulevée par la bibliothèque LeJos, nous terminons immédiatement le programme.

Dans le cas également relativement fréquent ou les données lues par le capteur ultra-son seraient invalide dès le début, nous le désactivons et nous ne l'utiliserons plus durant l'ensemble de la partie. Dans ce cas la précision générale du robot sera dégradée, pour cette raison il est recommandé d'interrompre (*échap*) l'exécution. Lorsqu'un tel problème survient, le robot informe l'utilisateur par une série de 3 bips durant la phase d'initialisation et par un message de log.

3 Choix d'implémentation

Nous décrirons ici les principaux choix d'implémentation de notre solution. Notre implémentation se base sur plusieurs Threads afin de gérer de manière indépendante plusieurs capteurs ou plusieurs services. Des outils d'analyse statique du code [1] ont été utilisés pour réduire le risque de deadlock, problème de lecture/écriture sur une variable partagée etc.

3.1 Gestion des décisions : les Objectifs

3.1.1 Idée et buts des Objectifs

Notre programme gère actuellement 9 objectifs distincts (par exemple *Drop* ou *GoToPosition*). Ces objectifs héritent tous d'une même classe abstraite (*Goal*). Les objectifs représentent à la fois un but précis et des actions *atomiques* (généralement non interruptible) à réaliser par le robot pour réaliser ce but. Les Objectifs incluent le script de leur exécution et, éventuellement, des préconditions a vérifier pour s'exécuter. Les objectifs peuvent modifier l'état du système.

3.1.2 Principe de fonctionnement

De manière générale si un objectif a besoin qu'une action soit réalisées et qu'un autre objectif permet de réaliser cette action alors il ajoutera cet objectif à réaliser. Un objectif peut également demander à être réalisé à nouveau, par exemple pour poursuivre une fois que les objectifs ajoutés sont terminés. Ceci est particulièrement pratique pour décomposer une action complexe. Des objectifs peuvent également être ajouter par un gestionnaire

1. Threadsafe (www.contemplateltd.com/threadsafe) notamment

de service de manière asynchrone (actuellement seulement l'objectif de recalibration).

3.1.3 Gestionnaire d'objectif

Le gestionnaire d'objectif gère une pile (First In Last Out) d'objectifs et autorise (ou rejette dans certains cas) l'ajout de nouveau dans celle-ci. Une fois qu'un objectif est réalisé alors le gestionnaire d'objectif effectuera un pull sur cette pile pour récupérer l'objectif suivant à réaliser. L'intérêt d'utiliser une pile est de donner la priorité à l'objectif en train d'être exécuté pour en ajouter de nouveaux.
Le gestionnaire d'objectif n'exécutera pas les objectifs ne remplissant pas leurs préconditions même après avoir tenté de les satisfaire (il y en a peu dans la version actuelle de notre programme).

3.2 Gestion des décisions : *GrabAndDrop*

L'objectif *GrabAndDrop* recherche le palet le plus proche, sans être trop proche (approche souvent peu précise dans ce cas) puis se divise en deux objectif, *grab* et *drop*.

3.2.1 Approche Optimiste

De manière générale l'objectif de grab effectue ces actions :

1. On vérifie que le palet est toujours sur la map, on abandonne sinon.
2. On se déplace jusqu'à être a porté radar du palet où l'on est *sûr* de le détecter.
3. On vérifie que les coordonnées radar et de la map sont cohérente, on abandonne sinon.
4. On se déplace a vitesse modérée jusqu'à être interrompu par lorsque l'on détecte une pression.
5. Si l'on détecte une pression, alors on dispose du palet, on a réussi l'objectif.
6. Si l'on ne détecte pas de pression alors on tentera de corriger l'angle d'approche en :
   (a) Reculant de manière modéré.

(b) Re-tentant d'obtenir une pression d'un palet en s'orientant légèrement à gauche de l'approche initiale.

(c) Re-tentant d'obtenir une pression d'un palet en s'orientant légèrement à droite de l'approche initiale en cas d'échec.

7. Si l'on a réussi à attraper un palet lors de ces deux tentatives, alors l'objectif est réussi mais on tentera une approche pessimiste la prochaine fois (car la recherche droite/gauche est coûteuse).

3.2.2 Approche Pessimiste

L'approche Pessimiste est réalisée si la dernière tentative de *grab* a échoué à attraper le palet à la première approche. Elle diffère de l'approche optimiste dans le sens ou une fois à portée radar du palet, elle recherchera le meilleur angle d'approche (parmi 0, -20° et +20°). On recherche le meilleur angle d'approche en récupérant les données radar à 0, -20° et +20° depuis la Pose considérée comme *face au palet* et en s'orientant vers celle affichant la plus petite distance.

3.2.3 *Drop*

L'objectif de Drop repose entre autre sur l'objectif *goToPosition* pour se déplacer jusqu'à la zone d'en-but adverse. Si l'on n'a pas détecté la ligne blanche alors on tentera d'avancer un peu plus. Au début, nous prenons soin de nous déplacer afin de ne pas amener les 3 palets face à nous dans la zone adverse sans marquer de points.

3.3 Gestion de la position

Afin de garder une position la plus précise possible, nous effectuons une moyenne pondérée des positions retournées par 3 services fournisseurs de positions. Si la distance parcourue est assez grande, alors nous mettons également à jour, si nécessaire, l'angle du robot avec la position calculée initiale et la position calculée finale.

3.3.1 Odomètre

L'Odomètre fournit par la librairie LeJos nous permet de disposer d'une base pour déterminer la *Pose* du robot après chaque déplacement. Les don-

nées fournies par celui-ci sont précise pour quelques opérations mais deviennent rapidement invalide sans utiliser d'autre moyen pour calculer et mettre à jour la *Pose* du robot.

3.3.2 Areas

Notre programme gère 2 Areas en mode *safe* et 4 en mode *unsafe*. les Areas sont définies par les lignes de couleurs ayant une couleur unique (donc les lignes rouge, jaune, verte et bleu). Toutefois, nous avons remarqué un grand nombre d'erreurs où le capteur de couleur confondait les couleurs bleu, noire et verte, pour cette raison nous ne considérons pas les Areas associées en mode *safe*.

Une Area est mise à jour à chaque fois où l'on rencontre la ligne associée et garde en mémoire si l'on est *avant* ou *après* celle ci. Ceci permet de mettre à jour la pose du robot en cas de données contradictoires. Des vérifications sont toujours réalisées pour confirmer l'information de l'Area, le capteur de couleur pouvant se tromper et certains mouvement du robot sont susceptible d'invalider cette information.

3.3.3 Caméra et *Map*

Calibration Lors du lancement du robot, celui-ci effectue une calibration en fonction des positions connues des palets initiaux les données de la caméra. Ceci est réalisé en faisant une moyenne et permet généralement d'augmenter la précision des données reçues et leurs cohérences avec les positions fixes des lignes de couleurs par exemple.

Récupération de la position L'odomètre fourni une première estimation de position, nous pouvons donc récupérer la position de l'item le plus proche sur la map. Cet item représente très certainement le robot. Nous mettons ensuite la Pose de l'odomètre avec cette donnée (pondérée).

3.3.4 Stratégie

De manière générale, plus un trajet est long, plus la probabilité de se perdre est importante. Le risque de se perdre augmente également légèrement avec la

vitesse. Pour palier à ce problème, nous divisons les trajets *trop* long en deux sous-trajets avec un calcul et mise à jour de la Pose du robot intermédiaire.

On tient également compte dans les calculs de rotation la présence ou non de palets actuellement attrapé par le robot (on tourne un peu moins avec).

3.4 Gestion des erreurs

Malgré la recherche de la précision maximum lors du calcul de position, le robot peut *se perdre*. Nous détectons plusieurs types d'erreur, parmi les erreurs détectées, il en existe deux types :

3.4.1 Détection des erreurs de positions

Lors du démarrage du robot, nous estimons sa *Pose* d'arrivée. Dans le cas ou celle ci serait trop différente de la Pose d'arrivé calculée (supposé incohérente) alors nous considérons que le robot est en situation de perte et doit se re-calibrer.

Si un trop grand nombre d'objectif de *Grab* d'un palet échouent, alors on considère également que le robot est en situation de perte.

3.4.2 Détection des erreurs aléatoires

Plusieurs erreurs *aléatoires* peuvent se réaliser. Celles ci sont traiter par un thread séparé. Parmi ces erreurs nous pouvons trouver un mouvement infini du robot pour une raison indéterminée. Nous détectons également les obstacles, mur ou robot ennemi lorsqu'il sont à portée radar et très proche. Si un obstacle est détecté alors on recule légèrement avant de poursuivre.

3.4.3 Correction des erreurs

L'objectif de re-calibration n'affiche malheureusement pas une fiabilité parfaite, notamment en raison des erreurs de détection de couleur de ligne, ou d'obstacle empêchant son bon déroulement. Toutefois, il permet dans beaucoup de cas de re-trouver sa position. Les étapes de re-calibration s'effectuent à vitesse réduite et sont :

1. On recule (ou avance une fois sur deux) jusqu'à tomber sur une ligne de couleur (autre que gris, noir ou blanc qui ne sont pas des lignes uniques).
2. On récupère la liste des Items présents sur la ligne rencontrée.
3. On avance légèrement d'une distance fixée.
4. On récupère la liste des Items qui ne sont plus présents sur la ligne rencontrée précédemment (en faisant une différence avec une nouvelle capture).
5. Si il n'y a pas un unique Item qui a disparu alors la re-calibration a échoué.
6. Sinon on recherche un nouvel Item à la distance fixée de l'Item ayant disparu (le robot donc).
7. Si on en trouve un alors nous connaissons notre position et nous pouvons calculer notre angle entre ces deux points
8. Si on en trouve 0 ou plus de 1 alors la re-calibration a échoué.

Si la re-calibration échoue alors elle redémarrera en choisissant un angle aléatoire puis se déplaçant un peu pour, avec de la chance, trouver une autre ligne de couleur. Nous inversons le sens (avant/arrière) lors de la recherche de couleur afin de limiter les cas de blocage par un obstacle.

4 Limites de notre implémentation

4.1 Limites lors de la détection d'Item

Notre implémentation se base exclusivement sur les données de la caméra pour définir ce qui est un palet (un item qui n'a pas bougé depuis quelque temps) et de ce qui n'en est pas. Cette option permet une assez bonne précision. Toutefois, il arrive que la position d'un item fourni par la caméra ne soit pas stable. Ceci a pour effet de réinitialiser le timer de cet Item fréquemment et donc ne sera jamais considéré comme un palet.
Notre solution limite ce problème en arrondissant les coordonnées (qui ne sont pas précises au centimètre), mais le problème persiste (moins fréquemment). De manière générale, il arrive qu'un palet (pas plus de 1 dans nos tests) soit ignoré durant toute la partie.

4.2 Limites lors de la re-calibration

Une des difficultés de la re-calibration était de trouver quelque chose de connu indépendamment de sa position. Il est donc fréquent qu'une tentative de re-calibration échoue.
Par exemple lorsque le robot tente une marche arrière contre un mur (difficile à détecter). Afin de ne pas échouer en boucle nous tournons de manière aléatoire en cas d'échec et nous alternons recherche en marche avant et recherche en marche arrière. Ceci permet d'éviter des cycles infinis au risque d'entrer en collision avec un mur ou un autre robot.

4.3 Limites lors du dépôt d'un palet

La détection des couleurs n'étant pas toujours fiable, nous n'utilisons pas uniquement la détection d'une ligne blanche pour lâcher un palet. En de rare occasion, un palet est déposé un peu avant la zone d'en-but.

4.4 Limites lors du calcul de la position

La principale limite de notre implémentation concernant la position est que nous ne calculons pas la position lorsque le robot est en mouvement. D'une certaine manière, ceci limite les erreurs (on cherche un Item là où on pense être par exemple), mais cela empêche certaines fonctionnalités s'exécutant sur d'autre thread de bénéficier de données précises en temps réel. Par manque de temps nous n'avons pas implémenté cette fonctionnalité.

5 Conclusion

Malgré les limites imposées par les capacités physiques du robot et le manque de précision des différents capteurs, nous avons pu proposer un programme offrant des performances moyennes satisfaisantes. L'utilisation cumulées de toutes les sources d'informations disponibles nous ont permis de limiter l'impact de ces problèmes.

L'utilisation *d'objectifs* associés a plusieurs *fournisseurs de service* (position etc.) permet une relativement bonne modularité et offre des opportunités pour d'ajouter des fonctionnalités ou en modifier sur le système existant.

A Gestion simplifiée des objectifs

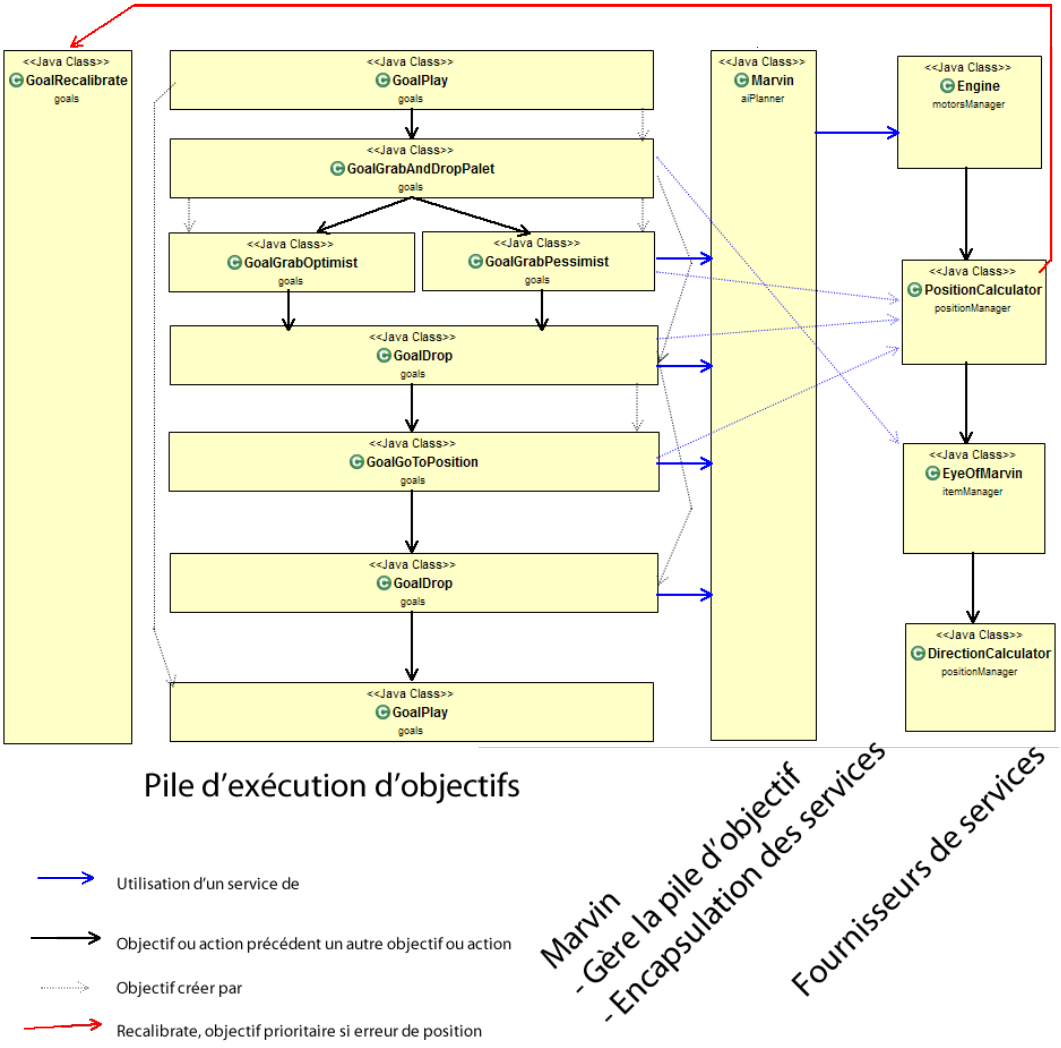

FIGURE 5 – Représentation des actions principales réalisées lors du *grab* d'un palet, des objectifs se créer à partir de l'objectif *play* de départ, appellent des primitives offertes par le gestionnaire d'objectif qui appellera les fonctions des gestionnaires de services pour garantir la cohérence du système

B Coordonnées utilisées par le robot

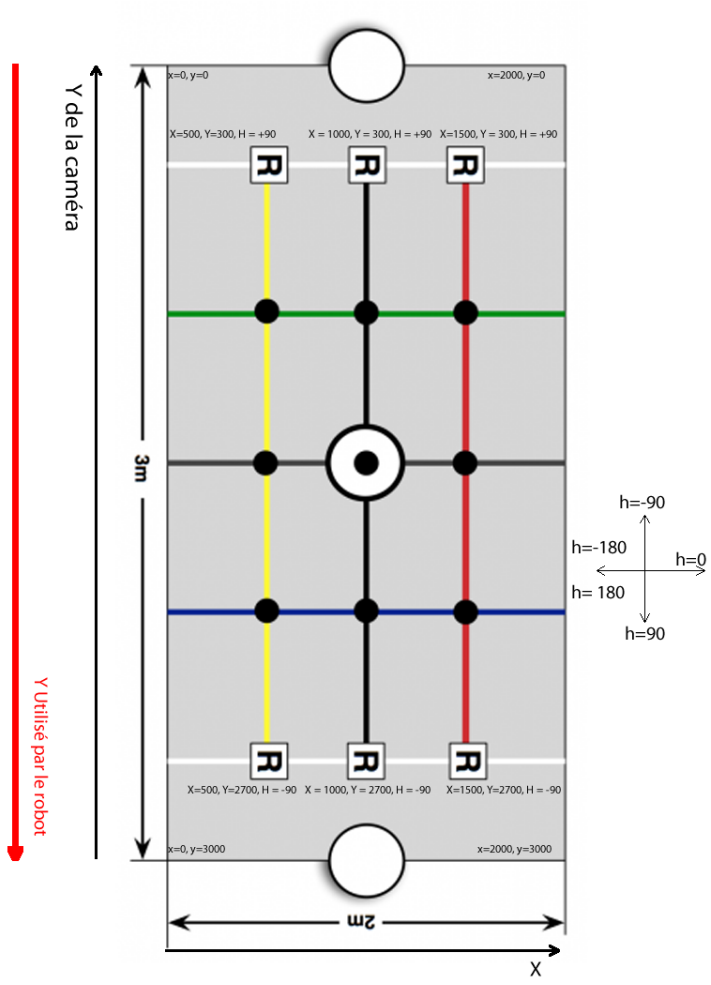

FIGURE 6 – Représentation du terrain avec les coordonnées utilisées par le robot

C *Areas* : Exemple

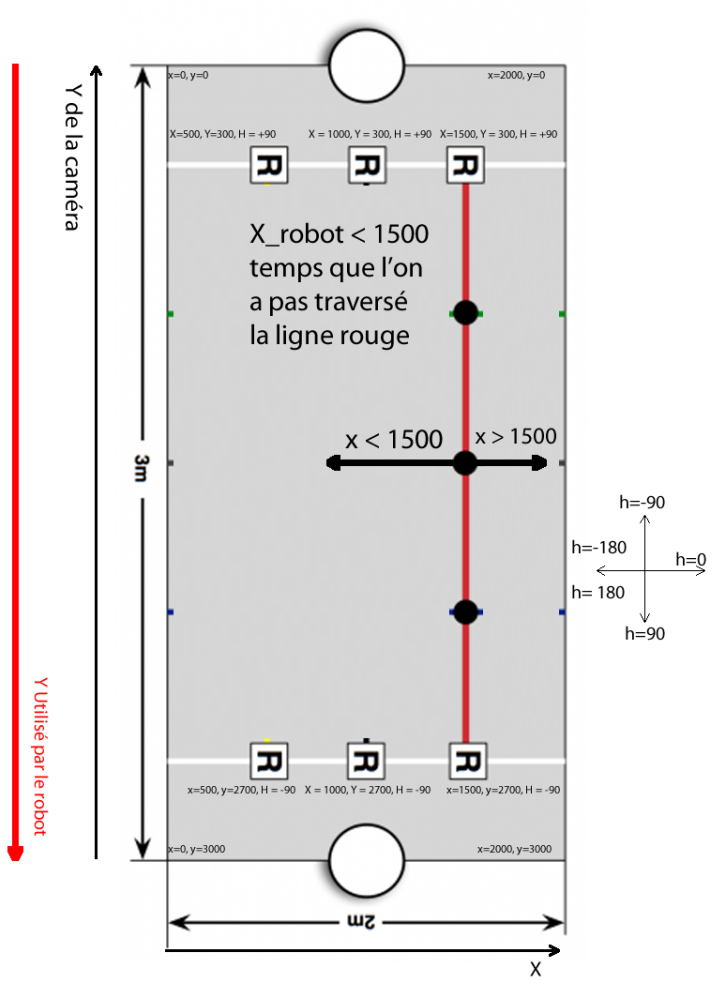

FIGURE 7 – Représentation de l'Area *Rouge*, Cette Area représente la position du robot par rapport à la ligne rouge