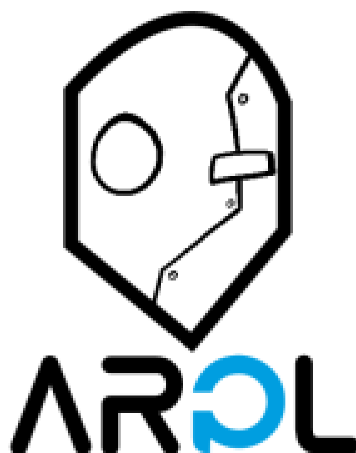
Rapport de Projet

P20 : Création d'un environnement de test virtuel



Marianne Butaye & François Duport

Encadrants : Vincent Coelen
Valentin Vergez
Rochdi Merzouki

Contexte 3
La compétition 3
Les phases du match 3
Cahier des charges 5
Matériel 5
Objectifs 5
Résumé ambitieux des tâches 6
Evolution du projet 8
Organisation du projet 8
Réunions 8
Documentation du code 8
Installation et prise en main de l’environnement 8
Installations logicielles 8
Structure d’un projet ROS 9
Fonctionnement de ROS 10
Structure du projet 10
Phases de jeu 10
Temps 11
Entités 11
Entite, Robot et Machine 11
Déplacement des robots 12
Formes et affichage sur RViz 12
Formes de base 12
Affichage des formes avec Rviz 13
Affichage des entités 13
Affichage du terrain 14
Affichage des feux 15
Gestion de l’orientation 16
Gestion des équipes 17
Gestion des collisions 17
Communication avec la Referee Box 19
Conclusion 21

Contexte

La compétition

Le projet consiste à concevoir et développer un environnement virtuel de test pour l'Association de Robotique de Polytech Lille (ARPL). Il va permettre de fournir à l'équipe de l'ARPL un simulateur léger et complet, leur permettant de simuler en un court laps de temps un match entier ou partiel et de tester leur code et leur stratégie. Cela leur permettra de développer plus rapidement et efficacement.

Ce simulateur est un outil développé dans le cadre de la RoboCup et plus particulièrement de la Logistic League. La RoboCup est une compétition internationale de robotique dans laquelle on voit beaucoup d'entreprises et de robots connus tel que NAO. Cette compétition a pour but de mettre en avant les évolutions en matière de robotique ainsi que les qualités des organismes participants. La Logistic League est une des différentes épreuves proposées, et simule un environnement d'usine avec des machines de production et des robots travailleurs.

La compétition se déroule sur une piste de la taille d'une salle moyenne. Chaque équipe peut utiliser jusqu'à trois robots Robotino 3 pour réaliser les tâches demandées. Sur la piste, à part les robots, se trouvent seulement des machines dites de production ou de livraison. Elles vont servir soit à se procurer un élément nécessaire à la production de l'objet final, soit à livrer celui-ci lorsqu'il est terminé. L'objectif est dans un premier temps de repérer le terrain et l'emplacement des machines, puis dans un second temps de produire des objets avec différents éléments fournis par des machines et à les livrer dans des plages de temps données.

Une Referee Box ou boîte arbitre sert à la coordination ainsi qu'à l'arbitrage. C'est cette boite qui va informer les robots des différents objets à construire avec la date de livraison demandée. Elle va également donner leur rôles aux machines présentes sur le terrain et assure la communication entre les différents acteurs (machines et robots). L'arbitre va enregistrer les horaires de dépôt des objets et la position de chacune des machines lors de la compétition.

Les phases du match

Le match oppose deux équipes qui s'affrontent lors de deux phases.

La première est la phase d'exploration, durant laquelle les robots doivent trouver les machines de leur équipe en explorant le terrain. Ils doivent annoncer la position des machines et détecter la combinaison des signaux des feux de celles-ci : chaque machine, peu importe son type, possède un feu tricolore (rouge, jaune, vert) qui durant la phase

d'exploration aura une combinaison de lumières allumées aléatoirement (au moins une lumière doit l'être).

La deuxième phase est la phase de production, au cours de laquelle les robots réalisent un certain nombre de commandes en utilisant les machines présentes sur le terrain. Chaque machine a un état courant et elles ne peuvent être utilisées que lorsqu'elles sont libres et fonctionnent normalement.

Les robots ordonnent aux machines de produire des types de pièces particuliers. Ils commencent par envoyer un message de setup qui va indiquer à la machine quel type de pièce préparer. À ce moment-là, la machine change d'état (état occupé) et ceci est reflété dans la couleur de son feu. Lorsque la pièce est disponible, la machine rechange d'état pour être de nouveau disponible. Il existe également d'autres types d'états, comme lorsque la machine est down lors d'une maintenance programmée décidée par la Referee Box au début du match, lorsque la machine vient de recevoir un ordre ou encore lorsqu'un robot tente de l'utiliser sans lui envoyer de message de setup, elle est alors hors-service pendant un certain moment.
À chaque utilisation d'une machine, il faut aussi que le robot attende un certain temps avant de renvoyer un ordre, car l'ordre met un certain temps à être réalisé.

Cahier des charges

Matériel

Nous n'avons pas besoin de matériel hormis d'ordinateurs afin de programmer.
Dû à la compétition et aux choix de développement de l'équipe de l'ARPL, nous allons programmer en C++ avec les bibliothèques QT ainsi que le middleware ROS, une surcouche de programmation destinée à la programmation de robots.

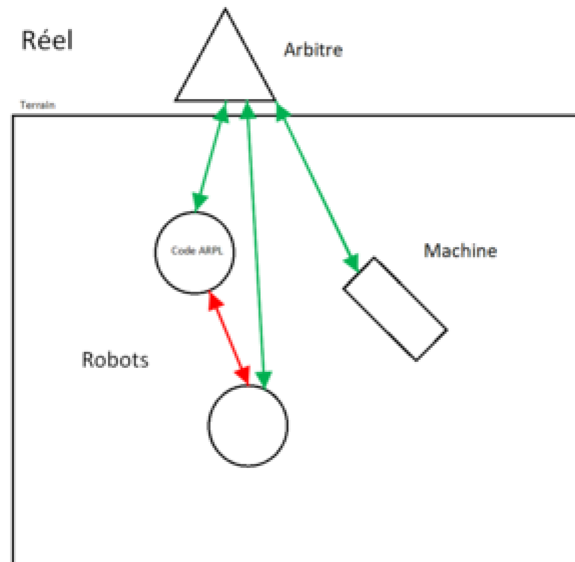
Objectifs

Le simulateur devra répondre à un certain nombre de critères. Tout d'abord, il lui faudra pouvoir simuler toutes les entités présentes sur la piste lors d'un match. Nous parlons des robots et des machines, mais également des conditions de jeu.

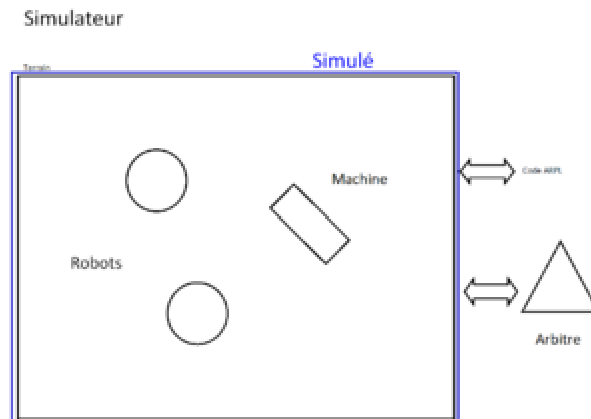
Nous devrons faire en sorte que le simulateur puisse facilement intégrer le code de contrôle des robots afin de le tester. De plus il faut permettre à l'utilisateur de gérer les parties du code à tester ainsi que de changer les conditions et paramètres de simulation afin de ne tester que certaines parties du code (par exemple en excluant la gestion des collisions). Nous inclurons une gestion des phases de jeu (Découverte et Production) dans ce but.
En sus, le simulateur gérant les robots de l'équipe adverse, il aura une fonctionnalité de sauvegarde des configurations que ce soit pour le code ou les stratégies afin de comparer l'évolution des performances.

Le simulateur a certaines contraintes afin d'être utilisable et utile. Il doit opérer une compression du temps c'est à dire réaliser la simulation d'un match en moins de temps qu'une durée normale de match. Il s'agit d'obtenir des données représentant la réalité en un cours laps de temps. La communication avec la Referee Box est indispensable. Celle-ci étant fourni par les organisateurs de la compétition nous n'avons pas à la simuler mais nous devons communiquer avec elle pour rendre la simulation plus réaliste.

Nous avons fait deux schémas simples pour expliquer ce que fait le simulateur par rapport à la réalité.



En Rouge les communications propres à l'équipe (entre les robots), En vert les communications avec l'arbitre



Tout est sur un ordinateur. On remarque que le code de l'ARPL n'est plus dans les robots mais à l'extérieur et se connecte avec l'environnement simulé

On observe que ce qui était physique devient simulé. Le code de l'ARPL pouvant changer nous ne le simulons pas directement il fonctionnera sur le même ordinateur et communiquera avec notre simulateur. L'arbitre fonctionne de même.

Résumé ambitieux des tâches

- Simuler les différentes entités
  - les Robotinos
  - les machines
  - l'environnement
- Gérer l'interfaçage entre le simulateur et le code des Robots
  - lire et lancer le méta-paquet
  - identifier les différentes parties du code

    - proposer l'activation/désactivation de ces parties et créer un fichier de lancement en conséquence
    - créer un système de sauvegarde de tout code interfacé permettant sa réutilisation.
- Simuler un match
    - communiquer avec la Referee Box et recevoir les différents ordres de celle-ci
    - gérer les machines et la communication avec la Referee Box
    - gérer l'équipe de l'utilisateur et son interaction avec le reste de la simulation
    - gérer l'équipe adverse et son interaction avec le reste de la simulation
- Paramétrage avancé de la simulation
    - proposer différents modes de simulations
    - activation/désactivation des paramètres de la simulation
    - gestion des phases
- Interface utilisateur
    - fournir des données utilisables et intéressantes sur la simulation
    - créer une GUI ergonomique
    - importer les données de la simulation sur la GUI pour une meilleure analyse de celle-ci

Evolution du projet

Organisation du projet

Réunions

Nous avons travaillé sur ce projet en binôme et de manière autonome, mais nous avons également programmé quelques réunions avec nos tuteurs de l'ARPL pour discuter de l'avancement du projet et de la direction à prendre.

Ces réunions occasionnelles nous ont permis d'obtenir plusieurs retours sur notre travail et l'avancement du projet. Nous avons ainsi pu effectuer immédiatement les modifications nécessaires suite aux commentaires de nos tuteurs.

Nous avons également discuté ensemble de l'avancement du simulateur et de l'ordre dans lequel effectuer les tâches. Ils nous ainsi demandé de créer un planning de nos tâches avec la date prévue pour chacune; ils auront en effet besoin de modifier leur code à certains endroits pour pouvoir utiliser le simulateur et souhaiteraient savoir lorsque nous seront prêt à tester de nouvelles fonctionnalités.

Documentation du code

Nous avons décrit nos classes par des schémas UML avec l'outil gratuit Modélio pour créer une documentation facilement accessible. Les commentaires de la plupart des fonctions, classes et fichiers au format Doxygen nous permettront également de générer une documentation du projet sous format HTML notamment.

Nous suivons aussi la normalisation du code donnée par l'ARPL pour avoir les mêmes conventions de nommage partout.

Installation et prise en main de l'environnement

Installations logicielles

Ce projet ne nécessite aucun matériel physique mais beaucoup d'installations logicielles sont nécessaires. Nous avons ainsi commencé par installer ROS, dont nous allons nous servir tout au long du développement. Nous avons également suivi des tutoriels pour apprendre à utiliser ROS, Rviz (affichage graphique de ROS) et avons révisé notre C++.

Pour pouvoir travailler en binôme sans problèmes et pour que nos encadrants puissent suivre notre avancement et tester le logiciel, nous avons utilisé le logiciel de gestion

de versions Git, sur lequel on publie notre code au fur et à mesure. Nous pouvions commencer la programmation une fois le dépôt Git du projet (nommé logistic-sim) récupéré auprès de nos encadrants.

Au cours du développement, nous avons eu besoin d'utiliser et installer le projet développé par l'ARPL pour réutiliser des messages qu'ils avaient créé et ainsi assurer une certaine compatibilité entre les deux projets. Nous avons aussi installé la Referee Box, dans la prévision du travail sur la communication entre simulation et l'arbitre, mais aussi pour pouvoir utiliser des types de messages particuliers inclus avec la Referee Box.

Structure d'un projet ROS

Un projet ROS est composé d'un environnement de travail dans lequel on peut mettre autant de projets que l'on souhaite, mais ils seront tous compilé en même temps par la même commande, *catkin_make*.

Nous obtenons ainsi la structure de projet suivante :

- Environnement de travail ROS
  - build
  - devel
  - src
    - CMakeLists.txt
    - logistic-sim

Les dossiers **build**, **devel, src** et **CMakeList.txt** sont des fichiers et dossiers générés par Catkin, un outil permettant de créer des paquets ROS facilement.

Nous créons ensuite des packages pour chaque regroupement de fonctionnalités que nous souhaitons ajouter au projet. Là aussi, une commande *catkin_create_pkg* permet facilement de créer leur structure interne de façon automatique.

Nous plaçons ces packages dans différents dossiers pour mieux s'y retrouver dans la structure du projet.

Chaque package a la structure suivante :

- CMakeLists.txt
- include pour les fichiers .h
- package.xml
- README.md
- src pour les classes et nodes .cpp
- launch pour les launchfiles

Fonctionnement de ROS

Un projet ROS est composé de *nodes*, équivalents à des petits programmes qui peuvent être exécutés en même temps ou à des moments différents. Ceux-ci peuvent communiquer ensemble par l'utilisation de *messages* transmis sur des *topics*. Dans chaque *node*, il est ainsi possible de recevoir et/ou transmettre certains types de messages.

Un *launchfile* est un fichier qui permet de démarrer une ou plusieurs *nodes* en leur donnant optionnellement des paramètres. Nous nous en servirons à plusieurs reprises pour paramétrer nos propres noeuds et le simulateur.

Structure du projet

Ceci étant notre premier projet mêlant ROS et Git, nous avons tenté de créer la structure du projet mais avons vite rencontré quelques difficultés. Nous avons ensuite compris qu'il était nécessaire de d'abord créer un environnement de travail ROS pour le projet, dans lequel on clone ensuite le dépôt Git localement.

La création d'un simulateur passe par la réflexion sur les besoins en conception du simulateur (décrits dans la partie précédente). Nous avons donc passé un certain temps à schématiser la structure de notre simulateur et à en assimiler les besoins et l'ordre de création.

Nous avons décidé de séparer le code du projet en différents dossiers en fonction des tâches à effectuer :

- **Communication** pour la communication avec la Referee Box
- **Entites** pour simuler et afficher les différentes entités (Robotinos, machines, environnement...)
- **Match** pour simuler le déroulement d'un match et la gestion des phases
- **Messages** pour stocker les messages encore inexistants dont nous aurions besoin

Dans chacun de ces dossiers, nous créerons des packages contenant les différentes fonctionnalités du projet.

Phases de jeu

Nous avions décidé au départ du projet de créer un package *match* qui contiendrait les nœuds correspondants aux différentes phases de jeu, en commençant par la phase d'exploration. Il s'est avéré par la suite que nous n'aurions pas le temps de tout faire, et il nous a été demandé de nous concentrer plutôt sur la phase de production, plus utile à l'ARPL. La phase d'exploration a donc été abandonnée en cours de route. Plutôt que de recréer un fichier pour la phase de production, nous nous sommes concentré sur l'ajout et l'implémentation de ses fonctionnalités principales.

Temps

La problématique rendant nécessaire une gestion du temps vient du besoin de l'ARPL de pouvoir contrôler la vitesse de simulation et donc de pouvoir jouer un match très rapidement afin d'observer le comportement de leur code. En effet, il est peu efficace de devoir attendre que le match se joue en entier et en temps réel pour connaître le résultat du code.

ROS intègre une interface de temps qui se cale sur la machine qui lance ROS. Pour que plusieurs machines suivent le même temps, il faut opérer une synchronisation des temps. Cependant, lors de la simulation, tous les robots auront leur référentiel temporel sur la même machine, ce qui rend une synchronisation inutile.

Il existe un topic clock qui sert pour les simulations. En forçant nos nœuds à utiliser le temps de simulation, nous avons une échelle temporelle utilisable et synchronisée. Nous avons donc un nœud qui va publier sur ce topic le temps qui sera initialisé à 0 en début de match.

Un paramètre d'environnement ROS sert à régler le facteur d'accélération de la simulation par rapport au temps réel. Il est modifiable à tout instant et est initialisé en premier dans le *launchfile*.

Le projet étant lié à la Referee Box, nous avons dû chercher à synchroniser le temps de celle-ci avec le simulateur. Cependant, en observant le code du simulateur Gazebo (actuellement non fonctionnel) et la manière de communiquer avec l'arbitre, nous avons conclu que nous n'avions pas le temps de le faire.

Entités

Entité, Robot et Machine

Nous avons créé une classe parente Entité dont hériteront les robots et les machines pour gérer leurs caractéristiques communes, comme l’affichage ou le nom. Dans différents packages machine et robot, nous avons créé les classes filles Machine et Robot.

Nous avons aussi complété le package machine pour lui ajouter les sous-classes de Machine représentant les différentes machines physiques (MPS) du jeu (BaseStation, CapStation, RingStation, DeliveryStation). La BaseStation permettra au robot de récupérer un objet de type base, la RingStation d'y empiler des anneaux de couleurs différentes en fonction des commandes reçues, la CapStation de déposer un chapeau au sommet de l'objet et enfin on l'apporte à la DeliveryStation.

Nous avons également ajouté dans chaque sous-classe du package machine une fonction setup qui permet d'initialiser la machine, la plupart du temps avec une couleur d'anneau ou de base, et pour la DeliveryStation avec un numéro de porte.

Déplacement des robots

Nous avons implémenté une fonction de gestion de la vitesse. Un seul nœud gère le déplacement de chaque robot. Les robots envoient un vecteur "twist" pour donner leur commande vitesse, soit deux vecteurs vitesse 3D, l'un pour la vitesse linéaire et l'autre pour la vitesse angulaire. Cependant ce vecteur vitesse est paramétré dans le repère du robot, c'est-à-dire avec le vecteur x dirigé dans la direction de l'orientation du robot.

Pour nos calculs nous nous servons du repère originel, que ce soit pour placer les entités ou les déplacer. Cela facilite les calculs dans la simulation. Ainsi nous devons transposer les vecteurs vitesse reçus pour les appliquer à notre repère.

La simulation fonctionnant en 2D nous simplifions les données en utilisant un seul vecteur 3D (x et y pour le déplacement linéaire et z pour le changement d'orientation soit la vitesse angulaire autour de l'axe z).

La transposition n'est nécessaire que pour les déplacements linéaires, donc nous utilisons de la trigonométrie simple et nous ajoutons le déplacement ainsi obtenu à la position actuelle, ce qui donne :

```
v_x = v_x1*cos(orientation_radian) - v_y1*sin(orientation_radian)
v_y = v_x1*sin(orientation_radian) + v_y1*sin(orientation_radian)
x_new = v_x *delta_t + x
y_new = v_y *delta_t + y
orientation_new = (v_angulaire*delta_t + orientation) [360]

On note delta_t le temps passé depuis la dernière actualisation.
```

Formes et affichage sur RViz

Formes de base

Nous avons créé un package *formes* chargé de la modélisation et par la suite l’affichage de formes géométriques basiques. Nous l’avons ainsi au début peuplé des classes **Point**, **Forme**, **Rectangle** et **Cercle** (héritant tous deux de Forme) mais aussi plus tard de formes plus évoluées pour l’affichage, comme **Fleche** et **Texte** et pour la détection de collision, **Distance**.

Affichage des formes avec Rviz

Nous nous sommes lancés dans l'affichage de ces formes avec l'outil *Rviz*, qui permet de visualiser en 3D les messages de type *visualisation_msgs* qui lui sont envoyés sur le topic associé. Ces messages sont disponibles en différents types. Pour afficher les formes nous utiliseront le format **Marker** qui permet d'afficher un objet en 3D (cubes, sphères, flèches, cylindres) ou en 2D en lui donnant un z de 0.

Nous avons donc commencé par ajouter le code nécessaire à l'affichage des formes. Nous avons ainsi décidé d'ajouter la variable du message de type Marker dans la classe Forme; celle-ci sera accessible par toutes les classes filles de Forme, notamment Cercle et Rectangle. Nous avons également créé une méthode *display* qui sera chargée de mettre à jour la position de la forme et de publier le message. Les différents setters des classes filles modifieront également le message pour qu'il soit à jour lors de l'envoi.

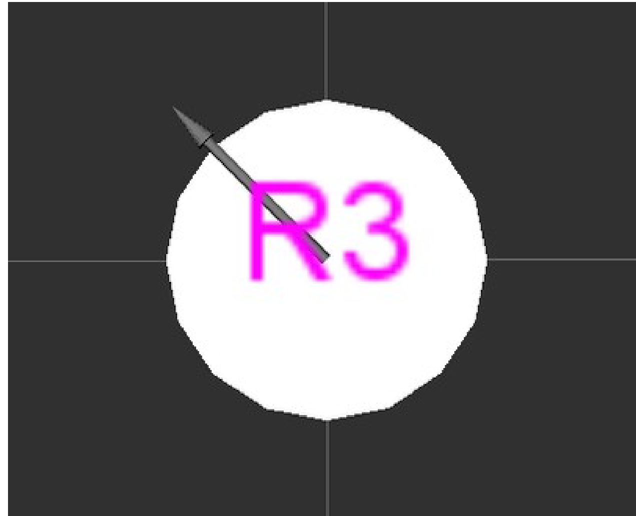
Nous avons également ajouté une forme Texte pour afficher du texte informatif sur les formes (par exemple, une lettre et un id pour une entité, "R1" dans le cas du premier robot de chaque équipe). Ce texte est affiché en cyan ou en magenta, en fonction de la couleur de l'équipe. Les noms des markers de la forme et du texte eux aussi dépendent du type de l'entité.

En cours de route, nous avons été confrontés à un problème au niveau de l'affichage en 2D. Il semblait logique que la hauteur des objets (en z) soit égale à 0 pour un affichage 2D, mais il s'est avéré que la couleur de l'objet était alors imprévisible (plus souvent noir que de la bonne couleur). Il en fait fallu laisser une légère épaisseur (0.001 mètres dans notre cas) pour que la couleur s'affiche correctement.

Affichage des entités

La classe **Robot** (qui possède une forme, étant une Entité) nécessite aussi un affichage pour pouvoir visualiser le robot avec rviz. Nous avons donc ajouté une fonction display à Entite, faisant elle-même appel à la fonction display de sa forme; ce qui permettra un affichage plus facile de l'objet.

Nous avons aussi permis aux entités de type **Machine** de gérer leur affichage. Pour cela, nous avons dû créer une classe fille de cette entité pour pouvoir tester son affichage. Le choix s'est porté sur les machines de type BS (Base Station).



Affichage d'un robot dans Rviz

Affichage du terrain

Pour afficher le terrain avec RViz, nous pouvions nous inspirer de code écrit par l'ARPL permettant de charger une carte depuis un fichier *.yaml*. Certains de leurs packages nous ont été utiles, mais d'autres se sont avérés inutilisables tels quels à cause d'erreurs tantôt de "référence indéfinie" à un fichier, tantôt de "redéfinition" du même fichier (la cause de ces erreurs reste pour le moment toujours un mystère).

Nous avons donc opté, pour pouvoir avancer rapidement sur le projet, d'utiliser tel quel ce qu'il était possible d'utiliser, et sinon de recopier les fonctions nécessaires dans notre package. Certaines fonctions nécessitaient d'ailleurs de toute façon des modifications, car dans notre simulateur nous gérons aussi l'affichage de la carte alors qu'eux ne faisaient que la charger.

Un fichier *.yaml* a la syntaxe suivante :

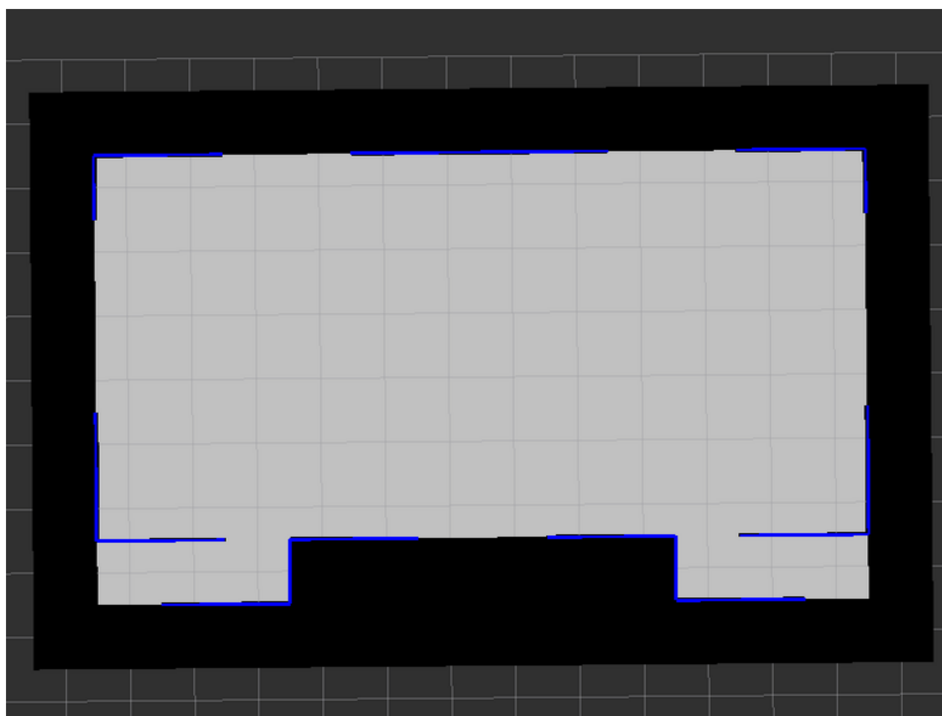
```
field:
   name: robocup_simulator
   size: [14, 9, 0.0]
   gradient:
       distance: 0.35
       minValue: 0
   walls:
       bottom1:
            start: [-6, 0, 0.0]
            end: [-4, 0, 0.0]
```

Nous avons donc commencé par charger le fichier en mémoire en le passant en paramètre à un *launchfile* lançant le nœud chargé de lire et d'afficher la carte. Ensuite, en faisant appel à des *param* (paramètres) du nœud actuel et en leur donnant le nom du champ (*name* par exemple), on peut récupérer les valeurs du fichiers voulues.

Nous avons ainsi créé une classe *Carte* pour gérer le chargement des données depuis le fichier et toutes les fonctions utilitaires associées, mais permettant aussi d'afficher la carte grâce à une fonction *display* de même fonctionnement que celles créées précédemment pour les entités.

Pour l'affichage sur Rviz, nous nous sommes servis du type d'objet *Map* pour visualiser le sol du terrain (en blanc ci-dessous). Nous avons également créé un vecteur de Rectangles pour afficher les murs (en bleu foncé ci-dessous) et un vecteur de Rectangles pour les zones interdites au robot (en noir).

Nous aurions aimé mettre à jour la taille de la grille affichée en fonction de la taille de la carte, mais il s'avère que c'est impossible, la grille ne provenant pas d'un message publié.

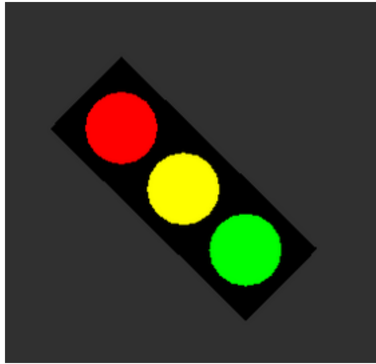


Affichage du terrain dans Rviz

Affichage des feux

Nous avons ajouté l'affichage des feux (composés d'un rectangle et de trois cercles rouge, jaune, vert). En fonction de l'état de la machine à laquelle ils appartiennent, les couleurs affichés changent. D'après le règlement, si tout va bien le feu sera vert fixe, si la machine est occupée il sera jaune fixe et si la machine est en maintenance programmée il sera rouge fixe. Si la machine vient de recevoir un ordre, il sera vert clignotant et si elle est hors-service il sera clignotant jaune et rouge.

C'est pourquoi nous avons aussi implémenté le clignotement de celui-ci qu'il est possible de voir dans la vidéo du wiki.



Affichage d'un feu en rotation sur Rviz

Gestion de l'orientation

Un de nos objectifs était aussi de gérer l'affichage de l'orientation de nos entités. En effet, cela semble essentiel pour les robots par exemple, représentés par des ronds et pouvant se déplacer.

Nous avons ainsi créé un nouveau marker *Fleche*; il était également possible de changer le type de marker en *interactive marker* ce qui aurait permis l'affichage de flèches de direction, mais celles-ci permettaient de modifier la position de l'objet et n'étaient disponibles qu'en multiple de 2 (déplacement sur un axe). Le choix s'est donc porté sur l'utilisation d'un marker classique dont on change l'orientation en même temps que celle de l'entité.

Nous avons rencontré quelques soucis quant à la gestion de l'orientation du marker, celle-ci étant sous forme d'un *quaternion* (x, y, z, w) alors que nous souhaitions donner l'angle en degrés (ou en radians éventuellement); la documentation n'est pas forcément facile à trouver pour ROS.

Les machines ayant chacune un feu, ceux-ci aussi doivent effectuer une rotation du même angle que leurs machines. Nous avons ainsi commencé par permettre aux cercles du haut et du bas d'effectuer une rotation autour du cercle central. Une fois la chose faite, nous avons modifié la rotation d'une machine.

En effet, jusque-là elle était implémentée de la même manière que pour les robots, or ceux-ci se déplacent et pas notre machine. Il n'est donc pas nécessaire que la flèche soit en rotation mais plutôt qu'elle soit positionnée de manière fixe (mais cela reste modifiable dans le code) et que ce soit le rectangle de la machine qui effectue une rotation sur lui-même. Il en va de même pour le rectangle du feu.

Enfin, nous avons dû gérer le positionnement du feu par rapport au rectangle : il faut, lorsque la machine effectue une rotation, calculer la position de son coin inférieur gauche et positionner le feu par rapport à celui-ci.

Gestion des équipes

Un nouveau package *management* a également fait son apparition. Il contiendra les classes **Equipe** et **Manager**. Le Manager sera chargé de créer les équipes et permettra d'y accéder partout dans le code, mais aussi de procéder aux différentes initialisations lors du démarrage d'une nouvelle phase.

La simulation est faite de manière à permettre à une Equipe d'afficher ses six machines et ses trois robots. Le Manager, lui, affiche les deux équipes; nous avons ainsi tous les acteurs d'un match sur le même terrain.

La classe Manager a été faite selon le schéma ***Singleton***. C'est un *pattern* de programmation permettant de n'instancier une classe qu'en un seul exemplaire unique accessible partout dans le programme. Dans notre cas, cela est particulièrement utile car le Manager permet de gérer les équipes et leurs états. On a seulement besoin d'instancier les équipes du Manager une seule fois au début du Match, et pas à chaque fois qu'on souhaite y accéder.

Gestion des collisions

Nous avons décidé d'implémenter la gestion de collisions dans le simulateur. Nous avons dû passer par la théorie avant de mettre ce système en pratique.

Visuellement et physiquement nous reconnaissons une collision de manière très évidente et naturelle. Nous la définissons presque mathématiquement comme le contact en un point de deux formes quelconques. Cependant pour que le programme les reconnaisse, il faut étudier entièrement l'aspect mathématique de la chose.

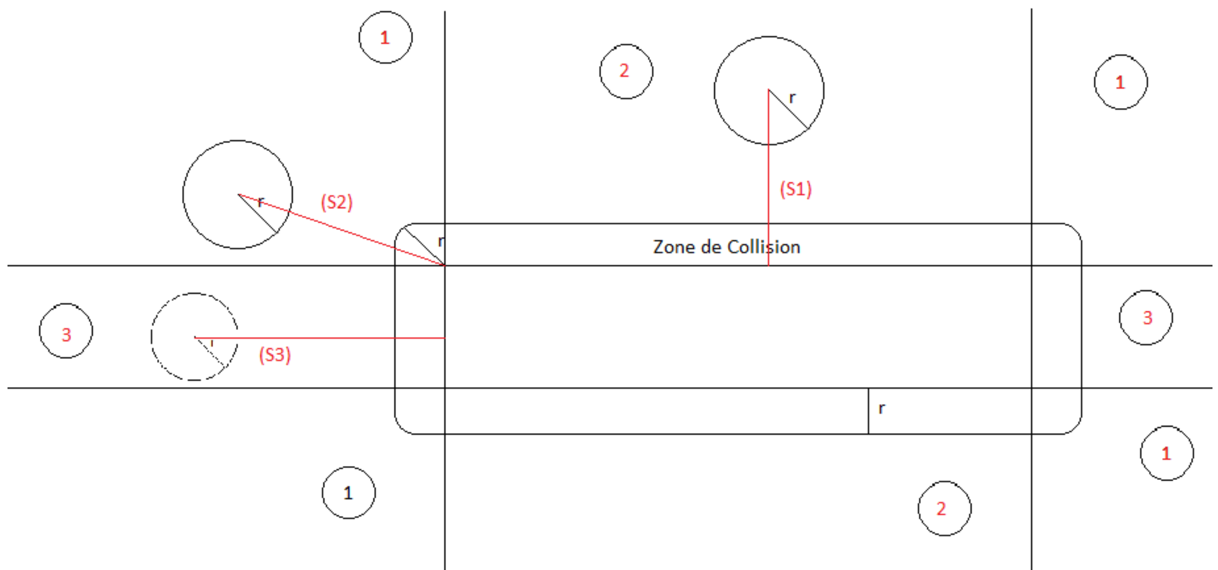
Nous allons donc déterminer la distance la plus courte entre ses deux formes qui sera le point de collision le plus probable. Par ailleurs nous négligeons d'abord le cas où les deux formes sont en collision. Il faut récupérer les points correspondant.

Entre deux cercles, il faut déterminer l'équation de la droite qui passe par le centre des deux cercles puis résoudre le système correspondant à l'intersection de la droite et du cercle qui nous donne une équation du second degré. On choisit alors le point le plus proche de l'autre cercle. Pour savoir si il y a collision, il suffit de calculer la distance entre les deux centres et d'enlever les rayons. Si le résultat est négatif, il y'a déjà collision, sinon on peut prévoir la collision avec le calcul précédent et la distance obtenue.

Lorsqu'il s'agit de calculer la distance la plus courte avec un rectangle, c'est un peu plus compliqué et très rapidement le nombre de calculs devient très grand. Nous avons choisi une stratégie calculatoire réduisant le nombre de calculs avec les conseils de Mme Nathalie Delfosse.

Pour un cercle et un rectangle, c'est plus complexe, on doit tester zone par zone. Nous ne sommes pas obligés de nous placer dans le référentiel du rectangle mais cela rend l'explication plus claire.

Nous pouvons facilement tracer la zone de collision, soit la zone qui permet de dire qu'il y a une collision si le centre du robot est dessus. Ensuite nous découpons l'espace en 8 zones en prolongeant les côtés du rectangle pour en faire des droites.



Lorsqu'un robot se situe en zone n°1 alors la distance la plus proche est celle qui sépare le cercle de l'angle le plus proche du rectangle. Donc (S2) est cette distance avec la résolution du système composé de la droite par le centre du cercle et l'angle, et le cercle. Nous obtenons alors le point du cercle le plus proche du rectangle.

(S2)
$(x-xc)^2+(y-yc)^2==R^2$
$y= ax+b$

Dans la zone n°2, la distance la plus courte est une droite verticale, ce qui nous donne facilement le point du cercle et le point du rectangle.

```
(S1)
xpc = xpr= xc
ypc = yc +/- r
ypr = yr +/- (l/2)
```

Dans la zone n°3, on observe le même comportement qu'avec la zone n°2 mais cette fois-ci à l'horizontal, les équations sont donc les mêmes avec une inversion des x en y et inversement.
Nous pouvons donc faire le calcul de distance et déterminer si collision il y a ou la distance.

Comme dit-ci dessus, ces equations ne fonctionne que pour le cas où nous sommes dans le référentiel du rectangle ce qui force plusieurs transformations. Au lieu de cela, nous allons vérifier zone par zone l'appartenance du cercle à celles-ci grâce aux systèmes composés de trois droites (les côtés du rectangles) pour les zones latérales et de deux pour les zones d'angles. Ces systèmes s'excluent les uns les autres. Si le cercle n'est dans aucune zone alors il est à l'intérieur du rectangle. Ce qui est soit une situation impossible, soit une collision. Lorsque l'on a la zone, on résout le système correspondant en adaptant avec l'orientation.

Nous avons découpé cette théorie en plusieurs fonctions pour le programme : Une fonction pour obtenir la distance entre deux points, plusieurs fonctions permettant de déterminer les points les plus proches entre chaque structure, et enfin une fonction déterminant s'il y a collision.

Nous n'avons cependant pas eu le temps de finaliser ce module.

Communication avec la Referee Box

Nous avons également commencé à réfléchir à la façon dont les informations sont communiquées dans le match : il y a un arbitre, la *Referee Box* qui permet de faire communiquer le simulateur et les robots jouant le match. En effet, l'arbitre prend les décisions importantes (placement des machines au début du jeu etc.) et c'est à lui que les robots s'adressent. Nous devons donc aussi communiquer avec lui pour recevoir certaines informations (par exemple les informations de base de la phase d'exploration) et les retransmettre dans le simulateur aux nœuds qui en auront besoin.

Il y a trois modes de communication avec la Referee Box : en tant que “pair” (communication UDP) sur un canal public, idem sur un canal privé ou en tant que “contrôleur” (communication TCP). Dans notre cas, c'est le mode “contrôleur” qui nous intéresse : en effet, nous devons avoir accès à toutes les informations de l'arbitre pour pouvoir réaliser une simulation correcte, et seul la communication en TCP le permet. En UDP sur canal public, seul un nombre très limité d'informations est disponible et sur canal privé les informations sont cryptées pour les cacher à l'autre équipe.

Nous avons donc pour cela créé une classe *ClientTCP* d'après un exemple inclus dans le projet installé. Le client s'inscrit comme devant recevoir certains types de messages, par exemple le message de type *llsf_msgs::MachineInfo*, qui permet d'obtenir les positions des machines et leur état. Ensuite, lorsqu'il reçoit un nouveau message, on vérifie de quel type il s'agit et on agit en conséquence. Dans l'exemple précédent, on appelle ainsi une fonction d'initialisation de la position des machines une unique fois après la réception d'un message de type MachineInfo, lorsque le Manager est initialisé mais que les positions des machines ne le sont pas. Cette fonction prenant en paramètre le message reçu, on n'a plus qu'à changer les différentes valeurs nécessaires.

Pour tester cela, nous avons utilisé la Refbox. Pour la démarrer, il faut se trouver dans le dossier *llsf-refbox/bin* et exécuter deux scripts; d'abord *llsf-refbox* pour la démarrer

puis *llsf-refbox-shell* pour voir une interface graphique de gestion apparaître dans le terminal.

De la façon dont nous avons réalisé le projet, il est nécessaire de créer dans son *bashrc* une variable d'environnement ROBOCUP_DIR qui correspond au répertoire ROS qui contient le projet et la Referee Box. Celle-ci permet d'utiliser les messages du package *llsf_msgs*.

Conclusion

Ce projet étant de grande ampleur, il n'est pas terminé. Nous avons cependant su lui apporter de bonnes bases alors même que nous apprenions plusieurs notions en travaillant dessus. Outre les compétences avec ROS et en C++, cela nous a permis d'apprendre à nous intégrer à la norme d'écriture d'un groupe et à travailler sur des codes et des modèles déjà existant tout en créant quelque chose.

Ce projet doit maintenant être continué pour lier ses différents éléments et communiquer avec le code de l'ARPL.