Université Libre de Bruxelles
Faculté des Sciences Appliquées
Département CoDE – Computer and Decision Engineering
Service IRIDIA – Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle

Année académique 2006 – 2007

Conception et implémentation en C++ d'un simulateur pour les robots e-puck et réalisation de tests de validation pour la cinématique de base

Mémoire de fin d'études présenté par Laurent Bury
en vue de l'obtention du grade
d'Ingénieur Civil en Informatique

Directeur de Mémoire : M. Marco Dorigo
Co-Promoteurs : M. Mauro Birattari
M. Alexandre Campo
M. Shervin Nouyan

Table des Matières

Remerciements

Je voudrais tout d'abord remercier Monsieur le Professeur Marco Dorigo d'avoir accepté de diriger ce mémoire et d'avoir accepté que je le réalise dans son service.

Je voudrais remercier Monsieur le Docteur Mauro Birattari pour le suivi et l'encadrement constants qui ont été les siens tout au long de ce travail.

J'adresse toute ma gratitude et ma reconnaissance à MM. Alexandre Campo et Shervin Nouyan pour l'implication et l'attention sans failles qu'ils ont portés aux travaux du groupe "e-puck". En plus d'être très efficaces et motivants, vous êtes très très sympathiques ! Merci d'avoir pris le temps de relire le présent travail.

Je remercie également M. Anders Christensen pour ses explications à propos de Twodee. Cela m'a été très précieux.

Je tiens également à remercier tous les étudiants membres du groupe "e-puck" grâce à qui j'ai passé d'excellents moments. J'ai été enchanté par notre cohabitation et par le travail produit pendant ces 8 mois de travail, le tout dans une excellente ambiance.

I would like to thank all the students from the "e-puck" project. I really had a lot of great moments that I will not forget. I am very happy about the way we all worked during those 8 months. Maybe we will meet again in the future. Maybe… Maybe not… We'll see ;-)

Merci de façon générale à tous les membres du service IRIDIA pour leur accueil.

Merci à l'ange étrange qui veille sans cesse sur moi. Merci à elle de m'avoir soutenu dans les quelques moments difficiles qui sont survenus ces derniers mois. Et merci pour toute l'attention qu'elle me porte à chaque instant.

Enfin, merci à ma famille qui a su me supporter depuis autant d'années et me permettre d'en arriver où j'en suis aujourd'hui. Je vous dois énormément !

1. Introduction

Ce travail va vous présenter une partie d'un vaste projet mené par six étudiants à l'IRIDIA. Ce projet global consistait à développer une plate-forme libre d'utilisation pour les robots "e-puck". Plusieurs aspects de la robotique ont été considérés. Parmi ceux-ci, citons la réalisation de tâches multi-robots telles que le tri, le choix collectif, mais aussi l'évolution artificielle et finalement le développement d'un simulateur.

Ce mémoire est consacré à ce dernier aspect : la conception, le développement et la validation d'un simulateur pour les robots e-pucks.

1.1. La robotique en essaim

L'utilisation des e-pucks s'est focalisée sur le concept de robotique en essaim, appelée également "swarm-robotics" dans la littérature anglophone. Il s'agit d'une approche spécifique dans l'étude des systèmes à plusieurs robots.

Ses caractéristiques essentielles sont les suivantes. Elle concerne des systèmes avec de nombreux robots de conception relativement simple. Au sein de ces groupes, aucun robot n'a de rôle particulier a priori, pas de leader donc. Les interactions entre robots sont locales et concernent aussi bien les relations entre robots qu'entre un robot et son environnement. Les systèmes ainsi formés ont pour caractéristiques d'être robustes et adaptatifs.

Les comportements des robots sont, entre autres, inspirés par l'étude des comportements animaux tels que les fourmis, les blattes, les abeilles, les guêpes, les termites ou les poissons. Ces animaux ont pour spécificité d'évoluer dans des sociétés où la notion de groupe est importante. Ils ont la capacité de toujours aboutir à une solution optimale, quitte à être passés par des états contre-productif, pour accomplir leur tâche.

Etant donné ces caractéristiques, les groupes de robot sont à même d'accomplir ensemble et de façon presque optimale des tâches complexes. Citons comme exemple le *patch sorting* qui consiste à grouper des classes d'objets en des endroits distincts dont la taille est très réduite par rapport à l'environnement.

Pour plus de renseignements à propos des comportements collectifs animaliers et de leurs applications en robotique, le lecteur trouvera une liste d'ouvrage dans la section bibliographique à la fin de ce mémoire (références [1] à [10]).

1.2. Intérêt d'un simulateur

Afin de travailler efficacement en robotique, un simulateur est un outil incontournable. En effet, le simulateur apporte une série non négligeable d'avantages dans ce domaine.

Premièrement, il y a l'indépendance par rapport à la qualité du matériel. En effet, un robot est extrêmement dépendant du matériel qui le compose. Cela le rend sujet à des pannes, sensible aux vibrations ou encore cassable. Un simulateur permet de s'affranchir de cet aspect désagréable des choses.

Ensuite, un robot tel que l'e-puck puise son énergie dans une batterie. Celle-ci n'a pas une autonomie infinie. Non seulement cette énergie n'est disponible que durant un temps limité mais, en plus, nos observations nous ont appris qu'un faible niveau de batterie pouvait encore faire fonctionner certaines fonctionnalités mais pas certains autres. Par exemple, il est possible que l'efficacité des capteurs infrarouge chute alors que les moteurs

fonctionnent encore. Le simulateur permet de se rendre indépendant par rapport à cet aspect de la robotique. Toutefois, si la simulation l'exige, il est toujours possible de modéliser cette dépendance à l'énergie.

Enfin, un simulateur permet la répétitivité très rapide des expériences. Il est en effet possible de se faire suivre un grand nombre de répétitions d'une même expérience. L'opération est encore d'avantage facilitée par la capacité de stocker diverses informations utiles afin d'analyser en détail le déroulement des expériences.

Néanmoins, le développement d'un tel outil n'est pas si évident qu'il ne pourrait y paraître. Etant donné qu'il s'agit de modéliser la réalité, il convient de réfléchir aux différentes approches envisageables afin de réaliser cet objectif au mieux. Dans le cas de la modélisation des capteurs du robot, par exemple, il y a le choix entre deux options : bâtir un modèle de la réalité ou se servir de données obtenues à partir d'un vrai robot. Elaborer un modèle fidèle à la réalité nécessite de trouver les différentes lois qui lient la situation d'un robot (sa position, son orientation, ...) avec les valeurs enregistrées par ses capteurs. Cette méthode, très générale, est d'autant plus difficile à mettre en œuvre que le modèle mathématique est complexe. Une autre méthode consiste à se servir de données extraites des robots dans un grand nombre de situations. L'extraction de ces informations s'appelle le sampling. Typiquement, cela consiste à réceptionner les valeurs renvoyées par les capteurs dans une série de situations récurrentes et représentatives. Cette technique permet de s'assurer que le simulateur respectera la réalité pour peu qu'il y ait assez de données issues d'observations empiriques.

2. L'origine de Twodeepuck : Twodee

2.1. Présentation de TwoDee

Twodeepuck, le simulateur que nous avons conçu, est basé sur un autre simulateur : TwoDee.

TwoDee est un simulateur développé à l'IRIDIA par Anders Christensen [11]. Son but est de simuler les robots *S-bots* et leur environnement. Il a été écrit en C++ et a été développé de façon à fonctionner directement sur des plateformes certifiées POSIX.1 afin de pouvoir être utilisé sur une grande variété de systèmes d'exploitation et de pouvoir également fonctionner sur des clusters et des architectures de type grid-computing. De façon plus précise, il a été développé sous *GNU Linux Debian* à l'aide des outils suivants : *GCC version 3.3* et *GNU AutoTools*.

La Figure 1 est une capture d'écran de TwoDee lorsqu'il est exécuté avec tous les paramètres par défaut. On notera la présence deux *S-bots* (les sphères argentées) et d'une source lumineuse (point jaune au centre). L'arène est un carré de 3 blocs de côté dont le bloc supérieur droit est un obstacle et le bloc inférieur gauche est un trou. Les textures ont été activées afin d'offrir un rendu plus agréable pour l'utilisateur.

Figure 1 - TwoDee, le simulateur des S-bots. On aperçoit ici 2 S-bots et une source lumineuse (au centre). Le bloc supérieur droit est un obstacle, le bloc inférieur gauche est un trou. Des textures ont été activées afin d'obtenir un meilleur rendu visuel.

2.1.1. Les S-bots

Les *S-bots* sont des robots d'environ 0,5 kilos. Ils se déplacent grâce à des moteurs qui peuvent entraîner des roues et des chenilles. Ils sont pourvus de capteurs "classiques" tels que des capteurs infrarouges de proximité, une caméra, des microphones, des capteurs de lumière, etc... Ils possèdent également d'autres capteurs moins basiques comme un accéléromètre ou encore des capteurs de température et d'humidité. L'ensemble des capteurs et des activateurs d'un *S-bot* se trouve listé dans le tableau 1.

Activateurs	Capteurs
Chenilles	Camera
Rotation de la tourelle	4 microphones
Pince	Capteurs de traction
2 haut-parleurs	Capteurs d'humidité
8x3 leds de couleur	Capteur de température
Bras flexible (optionnel)	Capteurs de proximité (infrarouge)
	Détecteur de sol (infrarouge)
	Accéléromètre
	Capteur de couple à la pince
	Capteur de couple de rotation
	Capteur de couple aux chenilles
	Capteurs de lumière

Tableau 1 - Liste des activateurs et des capteurs du robot S-bot

Cette quantité de capteur et d'activateurs fait du *S-bot* un robot très polyvalent et est particulièrement indiqué dans un cadre de recherches, et plus particulièrement si celles-ci nécessitent de devoir résoudre des tâches multi-robots variées et complexes. La Figure 2 montre un *S-bot* pleinement équipé.

Figure 2 - Un robot S-bot. On aperçoit, entre autres, la caméra (en bas du tube transparent), la pince, le bras amovible et les chenilles du robot.

Un des atouts majeurs du *S-bot* est d'être muni d'une pince lui permettant de s'accrocher à d'autres *S-bots* et de former alors un swarm-bot. Typiquement, ce type de configuration permet à plusieurs robots de résoudre ensemble une tâche qu'ils ne seraient pas parvenue à mener à bien individuellement, comme déplacer un objet lourd ou franchir un trou comme on peut l'apercevoir sur la Figure 3.

Figure 3 - Un swarm-bot est un ensemble de S-bots attachés les uns aux autres par leur pince. Cette configuration est, par exemple, utilisée pour traverser ensemble un trou qu'ils ne seraient pas parvenus à franchir individuellement.

2.2. Structure de TwoDee

Le simulateur TwoDee modélise les expériences voulues en faisant une approximation de la continuité du temps au moyen de cycles discrets. En effet, après une phase d'initialisation des conditions de l'expérience (phase de setup), le simulateur entre dans une boucle dont les états sont les suivants : perception et action des robots, mise à jour des éléments du monde virtuel, représentation graphique du monde virtuel. La boucle commence par une étape où le robot sent son environnement à l'aide de ses capteurs et où il agit en conséquence à l'aide de la logique décrite dans son

contrôleur (voir la Section 4.2.1.2.). Ensuite, le temps de la simulation est mis à jour et les éléments (robots exclus) sont déplacés en fonction des forces agissant sur eux. Enfin, l'interface graphique est mise à jour et la boucle recommence. La figure 4 représente ce cycle.

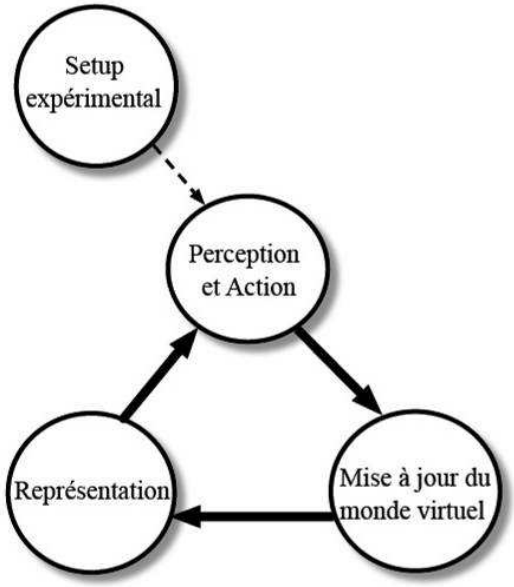

Figure 4 - Le cycle de fonctionnement de TwoDee. Après une étape d'initialisation du setup expérimental, le simulateur entre dans une boucle : perception/action - mise à jour - représentation graphique.

Etant écrit en C++, le programme utilise les concepts de l'orienté objet. Les objets manipulés représentent les différents éléments d'une simulation : *S-bots*, swarm-bots, capteurs, activateurs, arène, contrôleurs, rendus graphiques et le simulateur lui-même. Si on fait exception des rendus graphiques (ci-après dénommés "renders"), tous ces éléments héritent de la super-classe *CSimObject* ce qui se traduit par le fait de pouvoir considérer les objets selon une relation parent-enfant. En effet, tous les objets de *CSimObject* (et donc également des classes qui en héritent) peuvent se voir assigner des enfants. Cette notion de parentalité entre objets est totalement découplée de la notion d'héritage entre classes. Il s'agit de liens logiques instaurés afin de rendre relativement simples les cascades d'appels à chaque pas de simulation. En effet, la classe *CSimObject* implémente deux fonctions cruciales : *AddChild* et *SimulationStep*.

AddChild permet d'ajouter un enfant à un objet donné. *SimulationStep* permet de définir ce que l'objet doit faire à chaque pas de simulation. Par défaut, si un objet héritant de *CSimObject* ne redéfinit pas *SimulationStep*, cette fonction se contente d'appeler la fonction *SimulationStep* sur chacun des enfants de l'objet considéré. La Figure 5 montre la façon dont les différents objets de la simulation sont liés les uns aux autres par cette relation parent-enfant.

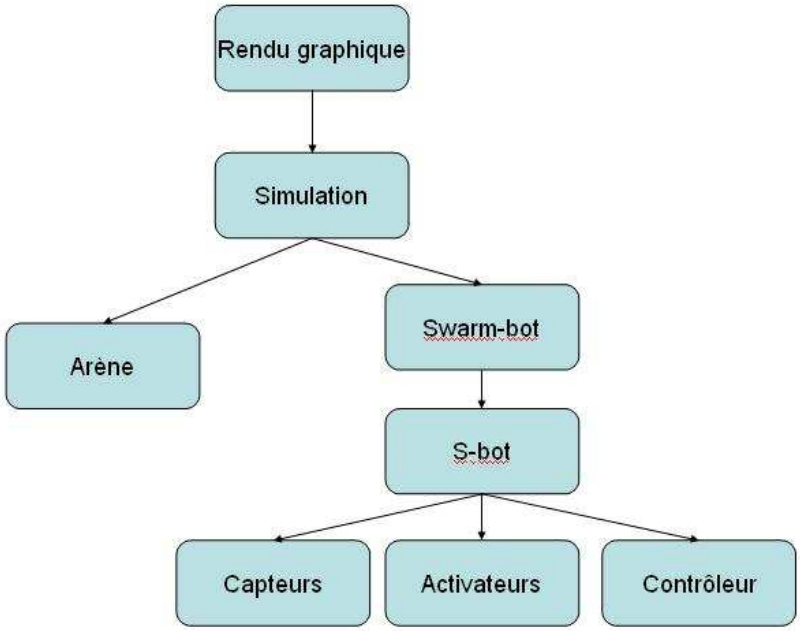

Figure 5 - Diagramme parent/enfant de TwoDee. Une flèche de A vers B signifie que A est le parent de B et est, à ce titre, responsable de provoquer la mise à jour de celui-ci à chaque pas de simulation. Ceci n'a aucun lien avec la notion d'héritage de classes, il s'agit d'un mécanisme d'appels en cascade sur des instances de classes.

La classe *CSimulator* possèdes 2 types d'enfants : l'arène et les robots.

La classe *CArena* est la super-classe de toutes les arènes du simulateur. Une arène est représentée comme une grille à 2 dimensions où chaque élément de la grille est un point de l'arène. Chaque point de l'arène peut avoir 3 hauteurs différentes. Une arène possède une taille et une résolution qui correspondent, respectivement, à des dimensions dans le monde réel et dans le monde virtuel du simulateur. Afin de rendre la définition de l'arène plus facile, il est possible de la définir par un tableau de caractères bidimensionnel. Les caractères indiquent la nature de l'arène dans une

subdivision spatiale donnée. Par exemple, il peut être convenu de créer une arène à l'aide d'une chaîne de caractères dans laquelle un espace, un X et un dièse (#) seront respectivement associés à un trou, un obstacle et un élément du sol.

Dans TwoDee, les robots sont avant tout des swarm-bots. Ces swarm-bots ont des enfants qui sont les *S-bots*. Et à leur tour, les *S-bots* ont des enfants qui sont leurs capteurs, activateurs et contrôleurs. Et donné cette configuration, il est relativement aisé de créer des *S-bots* avec des caractéristiques différentes au niveau de la nature et du nombre des capteurs, activateurs et contrôleur (un seul contrôleur par *S-bot* !). En effet, il n'est pas nécessaire d'avoir autant de classes pour les *S-bots* qu'il y a de combinaisons possibles des capteurs, activateurs et contrôleurs mais il suffit de créer un *S-bot*, identique à n'importe quel autre *S-bot*, et de lui assigner ensuite ses caractéristiques spécifiques. La classe *CSbot* (qui représente, fort logiquement, les *S-bots*) ne s'occupe dès lors pas de ces éléments et peut se concentrer sur la géométrie et les déplacements du robot durant la simulation.

2.3. Les étapes d'une simulation typique

Jusqu'ici, un point essentiel a été mis de côté : le rôle des renders (rendus graphiques). En plus de dessiner les différents éléments de la simulation (arène, robots et autres éléments éventuels), le render est en charge de la gestion de la simulation. En effet, c'est par sa méthode *Start* que la simulation démarre et ensuite c'est lui qui contrôle les pas de simulation et qui provoque les appels à *TakeSimulationStep* de l'objet *CSimulator* qui représente la simulation en cours.

Avant que la simulation ne démarre, il convient de tenir compte des paramètres de la simulation à exécuter. Ces paramètres sont soit ceux par défaut, soit ceux spécifiés par l'utilisateur à l'aide d'arguments dans la ligne de commande. Les principaux arguments à préciser sont le type d'expérience à lancer (le setup expérimental), le nombre de robots et le contrôleur à leur associer, mais il en existe encore une grande quantité.

Une fois la gestion des paramètres terminée, la simulation démarre par l'appel de *Start* du render activé (par défaut ou par les paramètres). Cela va provoquer une boucle d'appels vers *TakeSimulationStep* du simulator ("simulator" désigne l'objet de CSimulator, à ne pas confondre avec le simulateur en tant que programme). Pour rappel, la Figure 5 donne un aperçu des relations de parentalité qui existent entre les divers éléments d'une simulation. Le simulator commence par faire en sorte que les swarm-bots perçoivent leur environnement. Ensuite commence la cascade d'appels de la fonction *SimulationStep* à travers tous les objets ayant le simulateur à un quelconque degré de parentalité. Etant donné que le simulator ne redéfinit pas cette méthode, c'est celle par défaut, définie dans *CSimObject*, qui est appelée. Celle-ci provoque l'appel récursif à *SimulationStep* sur les enfants de l'objet actif. Les enfants du simulator sont l'arène et les swarm-bots ; la fonction *SimulationStep* de l'arène est donc appelée et comme l'arène n'a pas d'enfant, on remonte dans la hiérarchie et on passe aux autres enfants du simulator : les swarm-bots. Le swarm-bot redéfinit la fonction et dans celle-ci il commence par forcer l'appel de *SimulationStep* sur ses enfants : les *S-bots*. Cette fonction est également redéfinie pour les *S-bots* et elle se décompose en trois parties : gestion de la vitesse souhaitée par rapport à la vitesse courante, gestion de la manipulation de la pince du robot et enfin appel à *SimulationStep* de ses enfants. Ses enfants sont le contrôleur, les capteurs et les activateurs. Ces deux derniers n'ont pas redéfini la fonction *SimulationStep* et n'ont pas d'enfants : la cascade d'appels se termine ici pour eux. Le contrôleur n'a pas d'enfant mais il redéfinit la fonction : elle définit le comportement à adopter pour le robot. On remonte maintenant la cascade d'appels jusqu'à revenir au swarm-bot : après avoir provoqué les appels de *SimulationStep* sur ses enfants, il met à jour les positions et rotations de tous les *S-bots* qui le composent.

La cascade d'appels se termine et on revient au simulator à qui il ne reste plus qu'à mettre à jour le temps virtuel de la simulation et vérifier que celui-ci ne dépasse pas la limite éventuellement définie par l'utilisateur. Après cela, le render provoque à nouveau l'appel à *TakeSimulationStep* du simulator et c'est reparti pour un tour de simulation.

3. Twodeepuck

3.1. Le robot : e-puck

3.1.1. Introduction

Les robots e-pucks ont été développés à l'Ecole Polytechnique Fédérale de Lausanne (EPFL) dans un but pédagogique. Les développeurs ont choisi de le réaliser en "open hardware", c'est-à-dire en publiant toutes les données et tous les plans du matériel. Ceci est très pratique car cela nous a considérablement facilité les recherches lorsque nous avions besoin de l'un ou l'autre renseignement technique a propos de la conception du robot.

3.1.2. Caractéristiques mécaniques

L'e-puck peut être vu comme un cylindre, celui-ci faisant 7 cm de diamètre et 6 cm de haut. Son châssis en plastique réalisé en une seule pièce permet d'englober et de supporter les divers composants mécaniques et électroniques, ce qui en fait un robot relativement robuste (voir Figure 6).

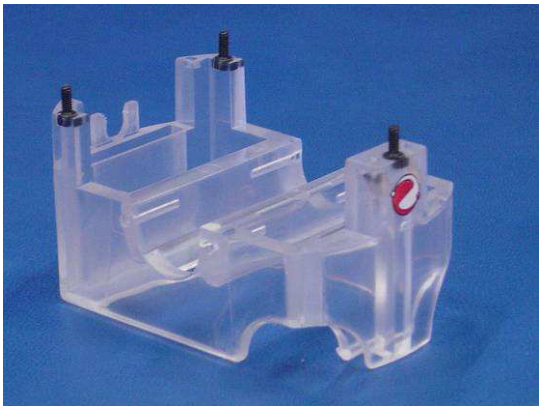

Figure 6 - Le châssis en plastique du robot e-puck, réalisé en une seule pièce, permet d'abriter l'ensemble des composants de l'e-puck.

L'e-puck se déplace au moyen de 2 roues de 41mm de diamètre disposées à 53mm l'une de l'autre. Il dispose de deux moteurs indépendants permettant de faire tourner les roues à une vitesse maximale d'un tour par seconde, soit un déplacement linéaire du robot d'un peu moins de 13cm/s. Il possède une batterie LiION de 3,6V d'une autonomie de 2 à 3 heures en usage normal. Celle-ci est rechargeable en dehors de l'e-puck.

Figure 7 - Le robot e-puck. On aperçoit aisément le cercle de diodes rouges, le haut-parleur (pièce circulaire noire) et la caméra (petit élément noir à l'avant du robot, sous la diode frontale). Les capteurs infrarouges sont logés juste sous l'anneau de leds rouges.

Afin de faire en sorte que nos e-pucks se détectent convenablement et de façon plus uniforme grâce à leurs capteurs infrarouge, nous les avons dotés de morceaux de bande réflective. Comme le montre la Figure 8, la bande réflective a été placée tout autour du robot sans oublier ses roues.

Figure 8 - Les e-pucks ont été munis de bande réflective afin d'améliorer et d'uniformiser leur détection mutuelle à l'aide de leurs capteurs infrarouge. Cette bande réflective a été placée tout autour de l'e-puck, y compris sur ses roues.

3.1.3. Caractéristiques électroniques

Au niveau de l'électronique, l'e-puck a été conçu de façon à pouvoir embarquer un grand nombre d'éléments malgré sa taille relativement réduite. Ce souhait vient principalement du fait qu'il est utilisé d'un point de vue pédagogique et, à ce titre, ses possibilités d'utilisation doivent être variées.

Le contrôleur du robot est un dsPIC 30F6014A. Tous les éléments électroniques fonctionnent à 3,3V à l'exception de la caméra qui a sa propre alimentation à 1,8V. L'e-puck peut communiquer avec le monde extérieur grâce à une connexion série RS232 et une interface bluetooth. Il dispose également d'un sélecteur à 16 positions permettant à l'utilisateur de sélectionner rapidement l'un ou l'autre programme en mémoire, ainsi que d'un bouton de reset afin de faire redémarrer un programme en cours d'exécution. On trouve également 8 capteurs infrarouges tout autour du robot. Comme le montre la Figure 9, ceux-ci ne sont pas disposés de façon régulière, il y en a plus vers l'avant du robot que sur les côtés et à l'arrière.

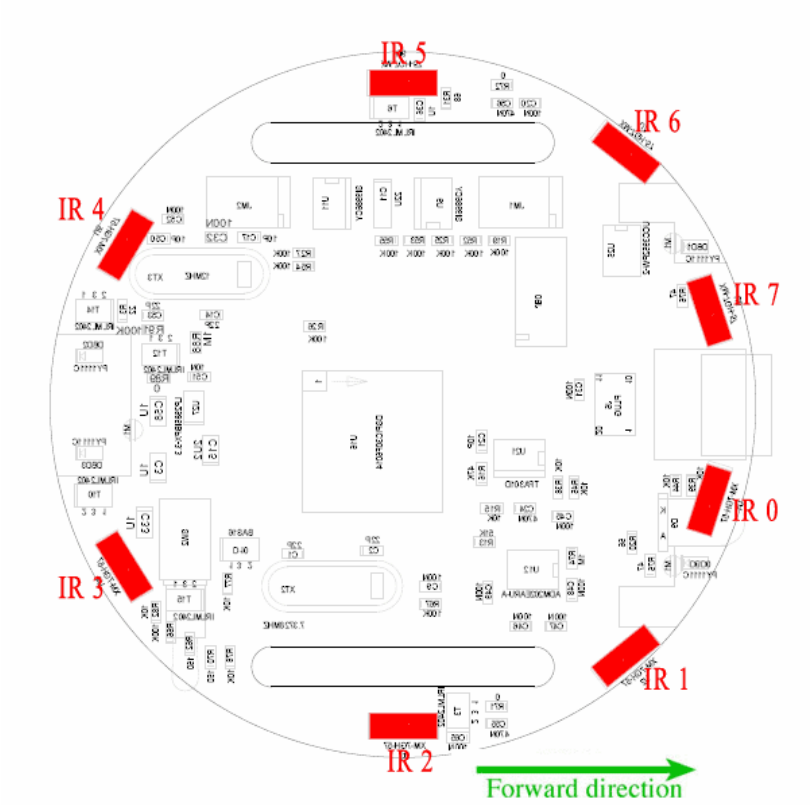

Figure 9 - La disposition des capteurs infrarouges autour de l'e-puck.[1] On notera la dissymétrie de leur placement, de façon à en avoir plus à l'avant que sur les côtés et à l'arrière. La raison est que ces capteurs servent principalement de capteurs de proximité et qu'il est en général plus important d'avoir une plus grande précision à l'avant qu'à l'arrière.

Le robot possède encore un accéléromètre 3D, trois microphones et un haut-parleur. Une caméra avec une résolution de 640x480 est disposée à l'avant du robot. Enfin, l'e-puck est muni de 8 leds rouges tout autour de son châssis ainsi que d'une led verte, appelée body led, à l'avant. Ces différents éléments sont résumés dans le tableau 2.

Activateurs	**Capteurs**
2 moteurs	8 capteurs de proximité (infrarouge)
Haut-parleur	Accéléromètre
8 leds rouges	3 microphones
1 led verte	Caméra

Tableau 2 - Liste des activateurs et des capteurs du robot e-puck.

[1] Ce schéma provient du site officiel du robot e-puck de l'EPFL : http://www.e-puck.org/

Les différents éléments décrits précédemment sont repris dans le schéma électronique de l'e-puck, à la Figure 10.

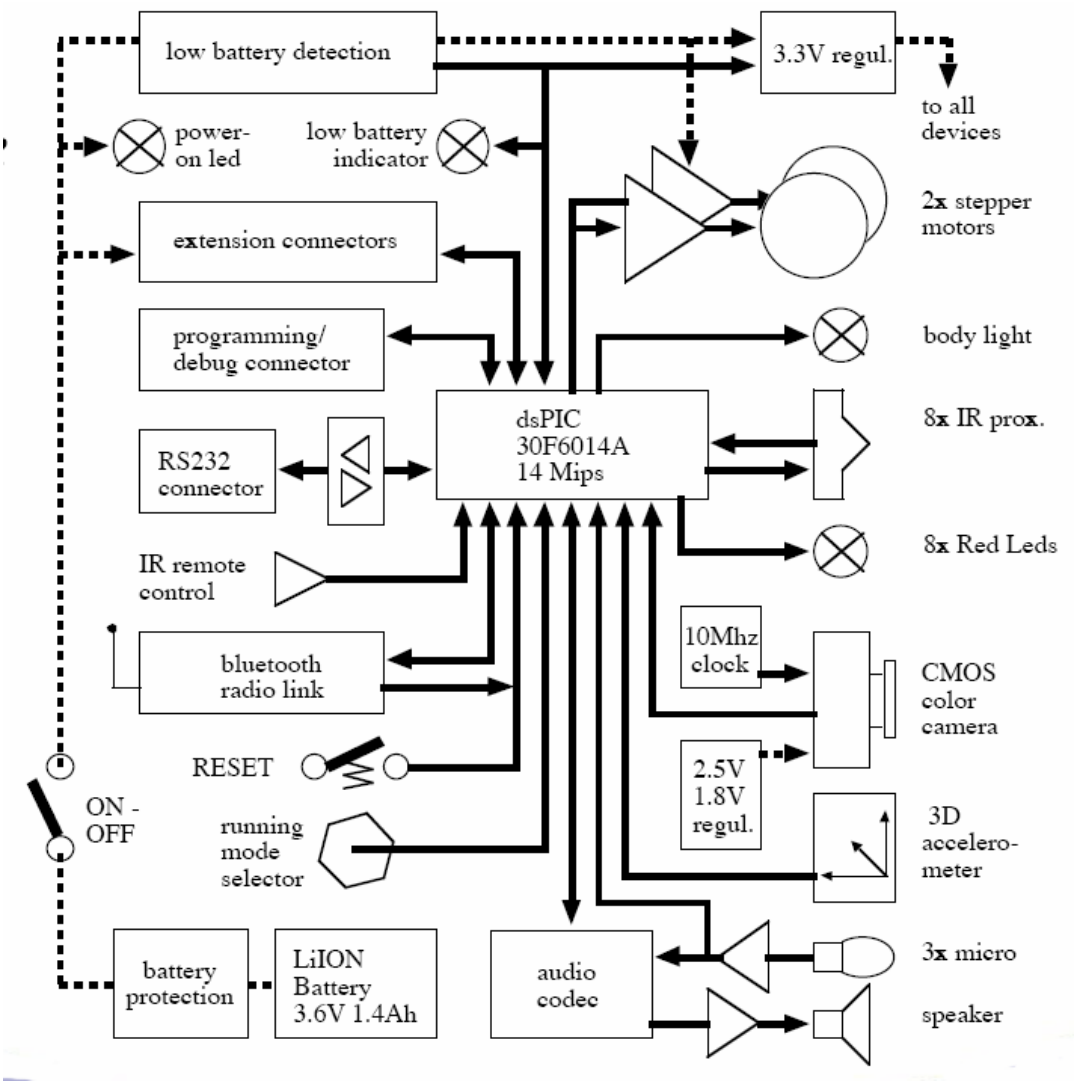

Figure 10 - Schéma électronique de l'e-puck.[2] On y retrouve tous les activateurs et capteurs décrits auparavant ainsi que les fonctionnalités d'entrée-sortie telles que le bouton reset, l'interface de communication bluetooth ou encore le sélecteur de programme (appelé "running mode selector")

[2] Ce schéma provient du site officiel du robot e-puck de l'EPFL : http://www.e-puck.org/

3.2. Le projet e-puck

3.2.1. Le groupe et la méthodologie

Ce mémoire a été réalisé dans le cadre d'un projet de développement d'une plate-forme d'utilisation des e-pucks. Un groupe de 6 étudiants de dernière année a été constitué afin de mener à bien ce projet à travers des travaux de fin d'études. Etant donné que, pour nous, tout était à découvrir à propos des e-pucks, la première partie de nos travaux fut un travail d'exploration que nous avons réalisé ensemble. En effet, la perte de temps et d'énergie aurait été grande si nous avions voulu faire face individuellement aux mêmes problèmes. Alors qu'en mettant nos forces en commun au début et en se répartissant les tâches, nous pouvions avancer plus vite en mettant immédiatement à disposition des autres les résultats de nos découvertes.

La méthodologie employée a été axée immédiatement vers la notion de travail en groupe.

La première chose qui fut mise en place a été de décider de faire des réunions toutes les semaines, à jour et heure fixes, afin que chacun puisse présenter son travail de la semaine écoulée, que nous puissions éclaircir en compagnie de nos co-promoteurs certains points de nos recherches et enfin que nous puissions décider de notre liste de tâches à réaliser pour la semaine suivante. Ces réunions hebdomadaires avaient également l'immense avantage de nous tenir au courant des recherches et avancements des uns et des autres et de pouvoir chercher les points de collaboration entre nos travaux, s'il était pertinent d'en définir.

Un autre point important de notre méthodologie a été l'usage d'un ensemble d'outils de travail collaboratif, tels qu'un wiki, un dépôt SVN, une mailing list et un dépôt SFTP. *Un wiki est un système de gestion de contenu de site Web qui rend les pages Web librement et également modifiables par tous les visiteurs autorisés.*[3] Nous y avons mis les points essentiels de nos recherches au fur et à mesure de leur découverte. Nous y avons également

[3] http://fr.wikipedia.org/wiki/Wiki

stocké l'ensemble des comptes-rendus de nos réunions hebdomadaires. L'adresse de notre wiki est : http://iridia.ulb.ac.be/~e-puck/wiki/tiki-index.php.

Enfin, un dernier point concernant notre façon de travailler : le SVN. Un SVN est un espace de stockage de fichiers particulièrement adapté pour le suivi de projets, informatique ou non. Il inclut en effet une gestion des différentes révisions (un révision étant une version du projet à un instant donné). Cet outil, bien qu'un peu compliqué à prendre en main au début, s'est avéré précieux par la suite lorsque le volume de nos données et la complexité de celles-ci ont augmenté.

Passé cette étape de travail massif en commun, des spécialisations plus claires sont apparues dans le projet et chacun s'est petit à petit concentré sur sa spécialisation. Dans mon cas, ce fut la conception et la réalisation d'un simulateur pour les e-pucks. Durant environ les trois premiers mois, j'ai réalisé ce travail en compagnie d'un autre étudiant, Giovanni Pini du Politecnico di Milano. Après ces trois mois, il s'est spécialisé dans l'étude de l'évolution artificielle à l'aide du simulateur pendant que je continuais le travail de développement.

3.2.2. La découverte des capacités de l'e-puck

Quand nous avons pris en main les e-puck pour la première fois, nous nous sommes immédiatement heurtés à la première difficulté : comment communiquer avec eux ? En effet, nous voulions pouvoir leur charger des programmes en mémoire afin d'observer ses possibilités mais… comment faire ? La façon de charger un programme dans sa mémoire est d'utiliser la connectivité bluetooth. Une fois le bluetooth configuré, nous pouvions enfin voir les effets de quelques programmes de base fournis par l'EPFL. Pour pouvoir aller plus loin, nous avons cherché comment écrire nos propres programmes. Le principe de programmation est le suivant : un programme doit être écrit en langage C, ensuite celui-ci est compilé par un cross-compiler[4] qui fournit enfin le fichier .hex à uploader dans la mémoire de l'e-puck via le canal de transmission approprié (le bluetooth).

[4] Il s'agit d'une modification de GCC permettant de supporter l'architecture dsPic des chips MPLabs.

3.2.3. Construction d'un environnement contrôlé

Durant les premières semaines du projet e-puck, nous avons construit une arène pour les e-pucks. Cette construction s'est révélée importante pour une raison assez simple : les capteurs infrarouges de l'e-puck sont très sensibles aux variations de luminosité. Le robot utilise ses capteurs infrarouges afin d'estimer sa proximité par rapport à des obstacles et également pour estimer la quantité de lumière ambiante à l'endroit où il se trouve. Il est vite apparu que ces deux fonctionnalités seraient parmi les plus prisées durant nos travaux. Quand on parle de sensibilité à la lumière, il s'agit de bien plus que la simple différence en la nuit et le jour. En effet, nous nous sommes aperçu que les capteurs étaient sensibles au point de faire une différence sensible entre la lumière à 10 heures du matin et celle à 15 heures de l'après-midi. Etant donné que nous souhaitions pouvoir répéter plusieurs fois nos expériences dans des conditions aussi identiques que possibles, il nous a semblé nécessaire d'envisager la construction d'un environnement contrôlé : l'arène (voir Figure 11).

Figure 11 – Afin de s'affranchir des fluctuations de la lumière ambiante, nous avons construit un environnement contrôlé. L'arène a été entièrement construite en bois et carton.

Cette arène devait avoir des dimensions permettant de faire circuler sans difficultés une quinzaine d'e-puck. Elle devait également pouvoir accueillir 2 abris ombragés sous lesquels une petite dizaine de robots devait pouvoir s'abriter (cette expérience fait partie du travail de fin d'étude d'Olivier Dédriche). Pour éviter que les e-puck ne rencontrent des problèmes de déplacement dans les coins, nous avons opté pour une forme ronde. Il fallait également pouvoir illuminer l'arène au moyen d'une ou deux ampoules et aussi y installer une webcam afin de pouvoir se rendre compte de ce qui s'y passait une fois l'environnement clôt. Ceci requérait donc d'avoir une structure à trois dimensions sur le dessus de laquelle il serait possible d'accrocher ces éléments précédemment cités. Après avoir imaginé diverses solutions et élaborés quelques croquis, nous nous sommes lancés dans la construction d'une arène de forme cubique, à charpente en bois. Elle comporte trois cadres en bois dont un central pour y attacher les ampoules et la webcam et deux proches des bords afin de soutenir les murs et le toit, faits de cartons. Des câbles ont été tendus afin d'assurer la stabilité de l'ensemble. L'espace où évoluent les e-pucks est délimité par des fines plaques de bois souple qui ont été pliées et qui s'appuient sur les bords de la structure en bois afin d'obtenir la forme ronde souhaitée. Le choix des matériaux a été dicté par le faible budget disponible pour la réalisation du projet.

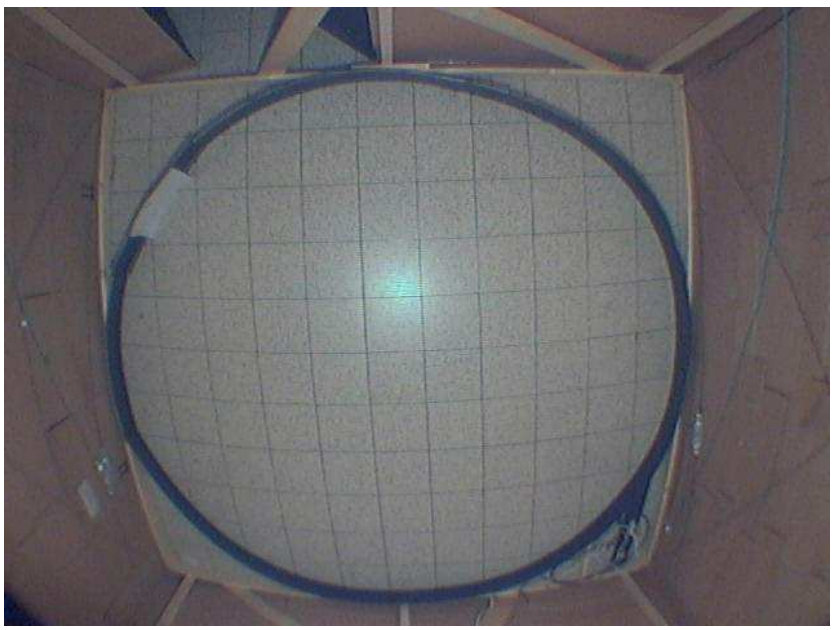

Figure 12 - L'intérieur de l'arène où évoluent les e-pucks. La prise de vue est assurée par une webcam installée dans une boîte accrochée à la structure en bois de l'arène.

3.2.4. La place du simulateur dans le projet

Parmi les avantages d'un simulateur, on peut pointer le fait qu'il ne tombe jamais en panne, contrairement aux robots qui ont, malheureusement, de nombreuses raisons de tomber en panne : moteur défectueux, fil déconnecté, diode grillée, etc… Dans la même gamme, mentionnons le fait qu'un vrai robot est soumis à des chocs, des vibrations, des variations de température ou d'environnement, tous des éléments qui le rendent fragile. Un simulateur permet de s'affranchir de ce genre d'aléa.

Un autre point important concerne l'autonomie des batteries. En effet, un robot fonctionnant sur batterie est toujours extrêmement dépendant de l'état de celle-ci. Et il ne faut pas penser uniquement à la nuance entre une batterie pleine et une batterie vide, nous avons pu observer avec les e-pucks que les robots pouvaient sembler fonctionner "à peu près convenablement" à cause de sa batterie. C'est-à-dire que la batterie était déchargée à un niveau auquel certaines fonctionnalités étaient en parfait état de marche alors que d'autres ne fonctionnaient plus. Un simulateur garantit toujours de fonctionner entièrement et permet de mettre de côté ces soucis d'autonomie et de dépendance énergétique.

Enfin, un autre avantage du simulateur est sa vitesse de fonctionnement. Rien n'empêche l'utilisateur d'accélérer les simulations selon son envie, chose impossible à faire avec de vrais robots. Cela permet, entre autres, de produire des résultats exploitables dans le domaine de l'inférence statistique où de gros volumes de données sont nécessaires afin d'accorder une certaine valeur aux résultats.

La plupart des chercheurs en robotique ne réalisent des expérimentations avec les robots que très rarement et ils travaillent sur un simulateur durant l'énorme majorité de leur temps. On le voit, l'apport d'un simulateur en robotique est essentiel.

3.3. Le choix du simulateur

La partie du projet consacrée au développement du simulateur a été prise en charge par deux étudiants, Giovanni Pini et moi-même. L'approche suivie a été de commencer à développer un simulateur à partir de rien, afin de mieux appréhender et de découvrir par nous-même les différents aspects devant entrer en ligne de compte dans ce type de projet. Ce simulateur, appelé basicSim, est décrit ci-après. Après deux semaines de développement, nous avons entrepris, en parallèle, de prendre en main TwoDee, le simulateur développé à l'IRIDIA pour les robots *S-bots*. Notre tâche avec TwoDee a été de le découvrir en essayant d'implémenter un contrôleur simple, l'évitement d'obstacles.

Après une semaine, nous avons fait le point sur la situation. Nous avions commencé à écrire par nous-même le simulateur basicSim pour nous rendre compte de certaines spécificités propres à ce genre de programme, et en même temps nous avions commencé à comprendre les mécanismes de TwoDee. Nous avions donc un choix définitif à faire : continuer à développer basicSim ou adapter TwoDee aux e-pucks. Il nous a semblé plus opportun de reprendre TwoDee et d'en faire une version qui simulerait les e-pucks. Cette décision a été motivée essentiellement par des contraintes de temps. En effet, nous sommes arrivés à la conclusion que seule l' "option TwoDee" nous permettrait d'avoir un produit en état de marche après 6 mois.

De plus, cette option permet d'avoir deux simulateurs (TwoDee et Twodeepuck) ayant des outils et des interfaces communs, ce qui posera beaucoup moins de problèmes de migration pour les futurs utilisateurs connaissant déjà TwoDee.

3.4. L'adaptation de TwoDee vers Twodeepuck

Afin de réaliser la mutation de TwoDee en Twodeepuck, il importait de comprendre dans le détail les mécanismes de base de TwoDee.

Nous avons décidé de dépouiller le programme de tout ce qui n'était pas essentiel, ensuite d'adapter le squelette ainsi obtenu pour que cela corresponde aux robots e-pucks et enfin de rajouter des classes pour obtenir un simulateur fonctionnel.

La première étape a donc été de supprimer toutes les classes qui ne représentaient pas une fonctionnalité essentielle des robots. Nous avons commencé par tout ce qui concernait la "common interface". Cette classe permet à l'utilisateur de faire facilement correspondre le code écrit pour les simulations et le code qu'il faut écrire pour programmer les vrais robots.

Ensuite, nous avons supprimé bon nombre de sous-classes des classes génériques. Par exemple, nous avons supprimé toutes les sous-classes représentant les différents capteurs (proximité, son, lumière, caméra, …), ces sous-classes héritant de la classe générique de tous les capteurs (appelée *CSensor*). Ce travail a été réalisé pour les capteurs, les activateurs, les contrôleurs, les setup expérimentaux et les émetteurs.

Enfin, pour clore cette étape de "squelettisation", nous avons enlevé toutes les classes excédentaires restantes. Cela comprenait tout ce qui concernait l'évolution artificielle (populations, individus, fitness functions) ainsi que la gestion des arguments et de la communication entre robots.

La deuxième étape a consisté en la mutation proprement dite vers le rôle de simulateur de robots e-pucks et non plus de robots *S-bots*. Dans cette optique, le premier travail a été de retirer la notion de swarm-bot. En effet, dans TwoDee, les objets de type *S-bot* n'étaient pas utilisés directement mais ils étaient groupés par swarm-bot. Un swarm-bot est une structure qui représente un ensemble de robots. Cela signifie que même un seul *S-bot* devait être créé au sein d'un swarm-bot. Il est aisé de concevoir les impacts que cela pouvait avoir au sein du simulateur : tous les objets qui devaient communiquer avec des *S-bots* devaient d'abord en récupérer le swarm-bot associé avant de pouvoir échanger des messages avec le robot proprement dit. Cette notion de groupe de robot a dû être retirée dans l'implémentation de Twodeepuck en raison de l'absence de pince et donc de l'absence de possibilité de créer des chaînes ou des groupes d'e-pucks.

Ensuite il a fallu adapter les classes existantes afin de rendre compte des caractéristiques de l'e-puck. Ainsi, il a fallu réécrire la cinématique de l'e-puck. Pour plus de détails à ce propos, voir la Section 3.5.

Enfin, la dernière étape dans la création de Twodeepuck a été de créer toute une série de classes annexes qui permettent de simuler les différentes fonctionnalités de l'e-puck et de son environnement. Il a fallu recréer des contrôleurs, des setup expérimentaux mais aussi des activateurs, des capteurs (se référer respectivement aux Sections 4.2.1.2, 4.2.1.1, 3.5.2 et 3.6 pour plus de détails), ainsi que créer la classe représentant des arènes rondes.

3.5. La cinématique

Les caractéristiques entrant en ligne de compte pour la cinématique de l'e-puck sont sa position en deux dimensions et son angle de rotation dans le référentiel choisi. Ce référentiel stipule que la position est déterminée selon 2 axes orthogonaux, x et y. L'axe x est horizontal et son sens positif est orienté vers la droite, l'axe y est vertical et son sens positif est orienté vers le haut. Les angles sont comptés positivement dans le sens trigonométrique (anti-horaire) par rapport à l'horizontale.

A chaque pas de simulation, l'objet de la classe *CEpuck* représentant le robot est chargé de mettre à jour sa position et sa rotation en fonctionne des vitesses de ses roues. La vitesse des roues a pour unité le mètre par seconde (m/s).

3.5.1. Equations du mouvement

Il faut distinguer deux cas de figure possibles lorsque le robot bouge : soit son mouvement est rectiligne, soit il est en train de tourner et donc, de suivre une courbe de rayon fini.

Dans la suite de ce travail, les notation et convention suivantes seront adoptées :

Les distances sont exprimées en mètres, les temps en seconde, les angles en radians, les vitesses en mètre par seconde et les vitesses angulaires en radians par seconde.

- *x* est la position en x de l'e-puck, *y* est sa position en y, *rotation* est sa rotation.
- *newX* est sa future position en x, *newY* est sa future position en y, *newRotation* est sa future rotation
- *leftSpeed* est la vitesse de sa roue gauche, *rightSpeed* la vitesse de sa roue droite, Ω (*omega*) sa vitesse angulaire.
- *step_time* est la durée d'un pas de simulation.

3.5.1.1. Déplacement rectiligne

Si l'e-puck est en train d'avancer ou de reculer en ligne droite (*leftSpeed=rightSpeed*), les équations du mouvement sont tout simplement les suivantes :

$$newX = x + leftSpeed \cdot step\_time \times \cos(rotation)$$
$$newY = y + leftSpeed \cdot step\_time \times \sin(rotation)$$

3.5.1.2. Déplacement circulaire

Si, au contraire, le robot doit tourner (*leftSpeed!=rightSpeed*), les équations du mouvement sont moins triviales. Dans ce cas de figure, le mouvement est un mouvement circulaire. Il convient de calculer la vitesse angulaire de rotation du robot, le rayon du cercle sur lequel le mouvement s'effectue, les coordonnées de ce centre du mouvement et enfin les position et rotation finales du mouvement.

La vitesse angulaire du robot et, par conséquent, sa rotation seront fonction de la différence des vitesses des roues. Etant donné le référentiel choisi, il n'est pas nécessaire de considérer deux cas différents en fonction du signe de cette différence des vitesses des roues. En effet, si le robot tourne vers la gauche, il ira dans le sens de rotation positif et la rotation

finale aura alors une plus grande valeur qu'avant. Le fait de tourner vers la gauche signifiant que la roue droite tourne plus vite que la roue gauche, on a les équations suivantes pour la rotation de l'e-puck (où *WHEELS_DISTANCE* est la distance entre les roues, 53mm) :

$$\Omega = \frac{rightSpeed - leftSpeed}{WHEELS\_DISTANCE}$$
$$\theta = \Omega \cdot step\_time$$
$$newRotation = rotation + \theta$$

Le rayon *R* du cercle décrivant le mouvement de l'e-puck se calcule par la formule suivante :

$$R = \frac{WHEELS\_DISTANCE \cdot (rightSpeed + leftSpeed)}{2 \cdot (rightSpeed - leftSpeed)}$$

Les coordonnées du centre de ce cercle sont dès lors :

$$ICC\_x = x - R \cdot \sin(rotation)$$
$$ICC\_y = y + R \cdot \cos(rotation)$$

A partir de ces éléments, on est en mesure de calculer les nouvelles coordonnées de l'e-puck à l'issue du pas de simulation en cours :

$$newX = (x - ICC\_x) \cdot \cos(\theta) - (y - ICC\_y) \cdot \sin(\theta) + ICC\_x$$
$$newY = (x - ICC\_x) \cdot \sin(\theta) + (y - ICC\_y) \cdot \cos(\theta) + ICC\_y$$

3.5.2. Le sampling des moteurs

Le vrai e-puck ne compte pas sa vitesse en m/s. Lorsqu'on programme le comportement à uploader sur de vrais e-pucks, il faut leur indiquer la vitesse en step/s. Le step est l'unité de référence du moteur. Pour donner un ordre de grandeur, la vitesse maximale théorique d'un moteur d'e-puck est de 1000 steps/s, cela correspond à un tour de roue complet et cela revient à 0,1288 m/s (= 12,88 cm/s).

Afin de pouvoir faire ce lien entre la vitesse dans le simulateur (m/s) et la vitesse dans le robot (step/s), un activateur a été créé : le "wheels actuator" implémenté au sein de la classe *CWheelsActuator*. La principale utilité de cette classe est de mettre à disposition la méthode *SetSpeed*. Cette méthode prend 2 paramètres : les vitesses respectives des roues gauche et droite, exprimées en step/s. *SetSpeed* opère alors une conversion en m/s et applique cette vitesse à l'e-puck.

La conversion théorique revient à multiplier les vitesses passées en paramètres par 0,0001288 mais un travail de sampling a été réalisé afin d'obtenir une valeur empirique. Le sampling avait également pour objectif de montrer si tous les robots avaient la même constante de conversion ou si, au contraire, les moteurs étaient sensiblement différents d'un e-puck à l'autre.

Le sampling a été réalisé en collaboration avec un autre membre du projet e-puck, Antoine Dubois. Pour réaliser ce travail, un premier e-puck a été choisi afin de connaître le nombre approximatif de steps nécessaires pour parcourir une distance de 1 mètre : 7715 steps. Sachant cela, le but était de faire tourner 7715 steps de moteur à différentes vitesses et de mesurer la distance parcourue ainsi que l'angle de déviation. Les vitesses retenues allaient de -1000 steps/s à 1000 steps/s par pas de 200 (sauf 0 step/s, bien sûr).

Les résultats de ce sampling, visibles sur la Figure 13, permettent de tirer l'enseignement suivant :

La vitesse observée varie linéairement avec le nombre de steps/s demandés. La valeur de conversion est donc une constante.

La valeur de la constante de conversion réelle est de 0,0001298 contre 0,0001288 pour la valeur théorique.

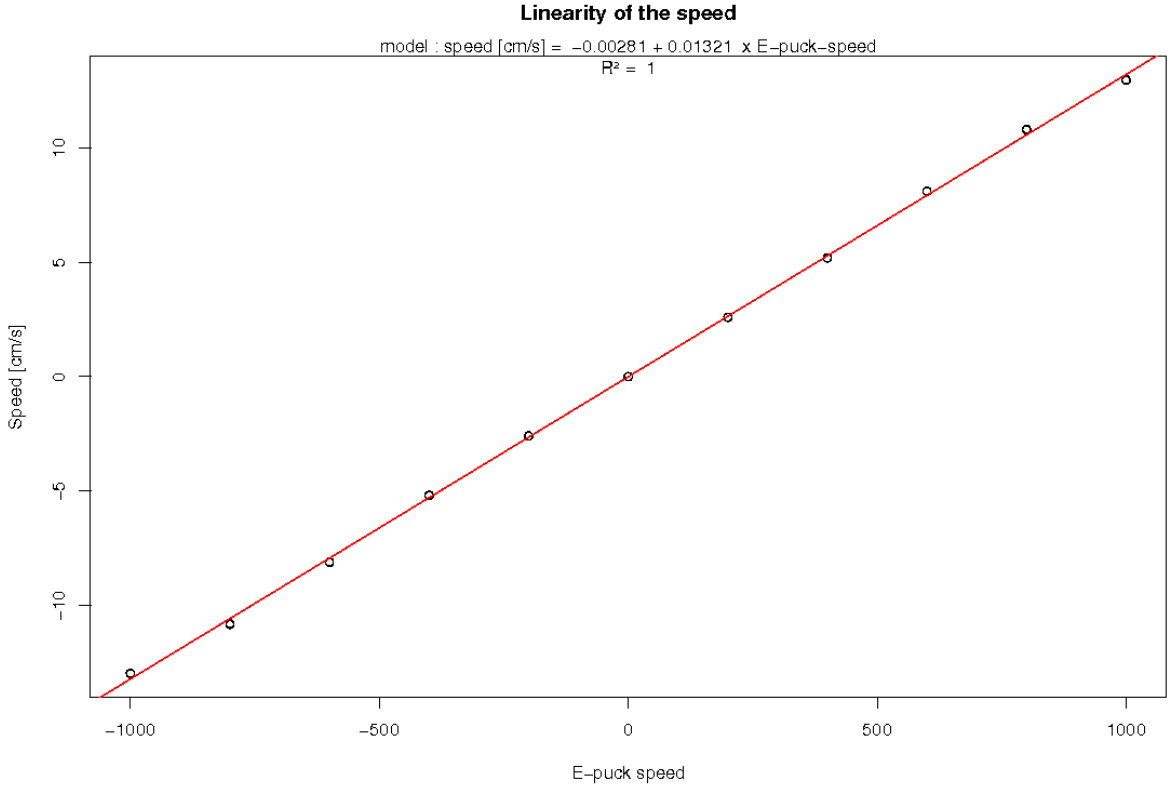

Figure 13 - Graphique montrant la linéarité entre la vitesse de l'e-puck en mètre par seconde et sa vitesse du point de vue du moteur, en step par seconde.

3.6. Les capteurs

3.6.1. Les capteurs de proximité

Les capteurs de proximité ont été implémentés dans Twodeepuck grâce à la classe *CEpuckProximitySensor*. Il s'agit d'une classe permettant de calculer les valeurs des 8 capteurs de proximité et d'en renvoyer les valeurs.

3.6.1.1. Le principe de fonctionnement

Quelques tests préliminaires ont rapidement montré la difficulté qu'il y aurait à essayer de trouver des relations mathématiques liant la nature d'un

obstacle, sa distance et son orientation par rapport à l'e-puck avec la valeur renvoyée par chacun des 8 capteurs. L'option choisie a donc été de travailler à l'aide de fichiers de sampling. Ces fichiers permettent de connaître les valeurs des capteurs qu'un robot réel a renvoyé dans certaines situations connues de distance et d'orientation par rapport à un obstacle.

Dans Twodeepuck, les capteurs mettent leur situation à jour à chaque pas de simulation. Afin de réaliser cela, il s'agit de récupérer tous les éléments de la simulation qui peuvent être des obstacles potentiels : l'arène à cause de son bord et les autres e-pucks. S'en suit alors l'analyse de la situation des obstacles potentiels afin de déterminer s'il sont perçus ou non par les capteurs de proximité. Pour ceux qui le sont, l'étape suivante est d'aller consulter le fichier de sampling afin d'en extraire les données relatives à la situation rencontrée (couple distance-orientation). Les capteurs renvoient des valeurs d'autant plus grandes que l'objet est proche, par conséquent, on stocke dans le vecteur des résultats les plus grandes valeurs rencontrées lors du pas de simulation en cours. Cela permet de gérer avec simplicité les situations dans lesquelles un obstacle serait caché par un autre. En effet, si l'obstacle caché est analysé en premier, les valeurs qu'il aura provoquées dans le vecteur de résultats seront inférieures à celles provoquées par l'obstacle qui le cache, vu que celui-ci est plus proche, et seront donc supplantées. Si, en revanche, il est analysé en deuxième lieu, ses valeurs seront inférieures à celles de l'obstacle le cachant et ne seront donc pas écrites dans le vecteur de résultats.

Le sampling des capteurs de proximité (décrit ci-après) a montré qu'il y avait des différences significatives entre les e-pucks. Les valeurs renvoyées par les capteurs de différents e-pucks dans les mêmes situations sont assez différentes pour justifier le fait d'avoir fait autant de fois le sampling qu'il y a d'e-pucks.

Enfin, la question s'est posée de savoir si il convenait ou non d'intégrer les fichiers de sampling dans l'exécutable du programme ou si ceux-ci restaient des fichiers extérieurs. Chaque situation a ses avantages et ses inconvénients. Ainsi, intégrer les fichiers de sampling dans l'exécutable permet de ne pas avoir à déplacer les fichiers de sampling si on souhaite bouger l'exécutable. En revanche, cela ralentit considérablement la

compilation des fichiers incluant les fichiers de sampling. Et inversement, si on laisse les fichiers de sampling à l'extérieur de l'exécutable, on y gagne en temps de compilation mais pas en portabilité de l'application.

La solution retenue a été d'inclure ces fichiers afin de n'avoir que l'exécutable à prendre en compte pour faire fonctionner le simulateur à n'importe quel endroit d'une arborescence de fichiers. Afin d'intégrer ces fichiers, un programme extérieur a été écrit dont le rôle est de produire un fichier d'en-tête (fichier .h) sur base d'un fichier de données brutes. Ce fichier d'en-tête contient les déclarations et valeurs de variables propres au sampling considéré : nombre de distances et d'orientations différentes, numéro de l'e-puck, etc… Il ne reste alors plus qu'à inclure ce fichier .h dans la définition de la classe *CEpuckProximitySensor* afin que celle-ci puisse utiliser les données du sampling afin de déterminer les valeurs des capteurs à chaque pas de simulation dans Twodeepuck.

3.6.1.2. Le sampling des capteurs de proximité

Le sampling des capteurs de proximité a été réalisé en collaboration avec Olivier Dédriche et Mouhcine Zekkri. Afin de refléter de façon réaliste les variations des valeurs renvoyées par les capteurs d'un vrai robot, le sampling a été réalisé en tenant compte des paramètres suivants : la nature de l'obstacle, la distance et l'orientation par rapport à celui-ci.

L'obstacle ne peut être que de deux sortes : un mur ou un autre e-puck. Des tests rapides ont montré qu'à l'évidence les capteurs de proximité réagissent très différemment selon qu'ils se trouvent confrontés à un mur ou un e-puck, le mur provoquant des valeurs sensiblement plus élevées. Il y aura donc 2 fichiers de sampling distincts : un pour les situations vis-à-vis des murs et un pour les situations vis-à-vis des e-pucks.

A propos des distances et des orientations auxquelles il est pertinent de faire du sampling, plusieurs tests ont dû être menés avant d'arrêter une décision. Finalement, il a été décidé qu'il fallait prendre des mesures tous les centimètres de 0 cm à 20 cm (inclus) et tous les 10° d'angle pour les orientations. Cela représente donc 21 x 36 = 756 mesures.

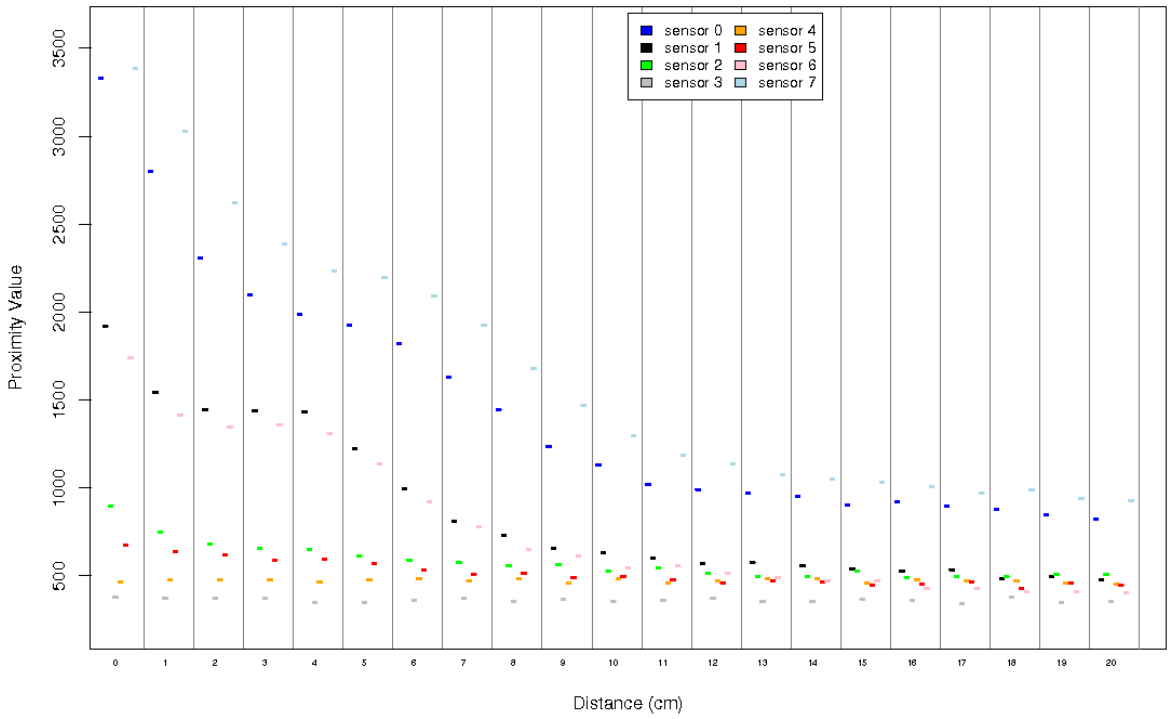

Figure 14 - Graphique donnant la prise de mesure des 8 capteurs de proximité (un par couleur) à différentes distances d'un obstacle. L'e-puck fait face à l'obstacle.

3.6.2. La caméra

La caméra a été implémentée dans Twodeepuck grâce à la classe CCameraSensor. Il s'agit d'une classe permettant de calculer la quantité de pixels d'une couleur déterminée vue par l'e-puck.

3.6.2.1. Le principe de fonctionnement

Le principe de fonctionnement de la caméra dans le simulateur est semblable à celui des capteurs de proximité. Il s'agit de récupérer tous les éléments pouvant être perçus par la caméra et de déterminer la quantité de pixels de la couleur voulue qu'ils renvoient. La caméra peut être paramétrée pour être sensible aux trois couleurs suivantes : rouge, vert, bleu. La corrélation entre les données de l'objet visé (sa nature et sa distance) et la quantité de pixels est établie grâce à un fichier de sampling. La nature des objets visés peut être de deux sortes : un autre e-puck ou un puck (un puck est un palet utilisé dans des expériences de tri collectif).

Si un objet en cache un autre, le problème est plus complexe que pour les capteurs de proximité. En effet, avec la caméra il s'agit maintenant de tenir également compte de l'objet en partie caché car il est lui aussi susceptible d'être vu par la caméra dans la couleur voulue. Une solution a été élaborée et testée afin de régler ce cas de figure. Elle consiste à déterminer la proportion de l'objet caché que l'on perçoit quand même, à l'aide d'un peu de géométrie et de trigonométrie.

Le sampling a montré que la quantité de pixels de la couleur voulue était directement proportionnelle au nombre d'e-pucks ou de pucks visés (voir Figure 15). Par conséquent, le fichier de sampling est extrêmement court et il ne provoque pas de rallongement significatif du temps de compilation, contrairement aux fichiers de sampling des capteurs de proximité. Au niveau de la programmation de la caméra, cela a considérablement simplifié la tâche. En effet, il s'agissait simplement de multiplier la quantité de pixels de couleur par le nombre d'objets de même nature se trouvant à des distances égales.

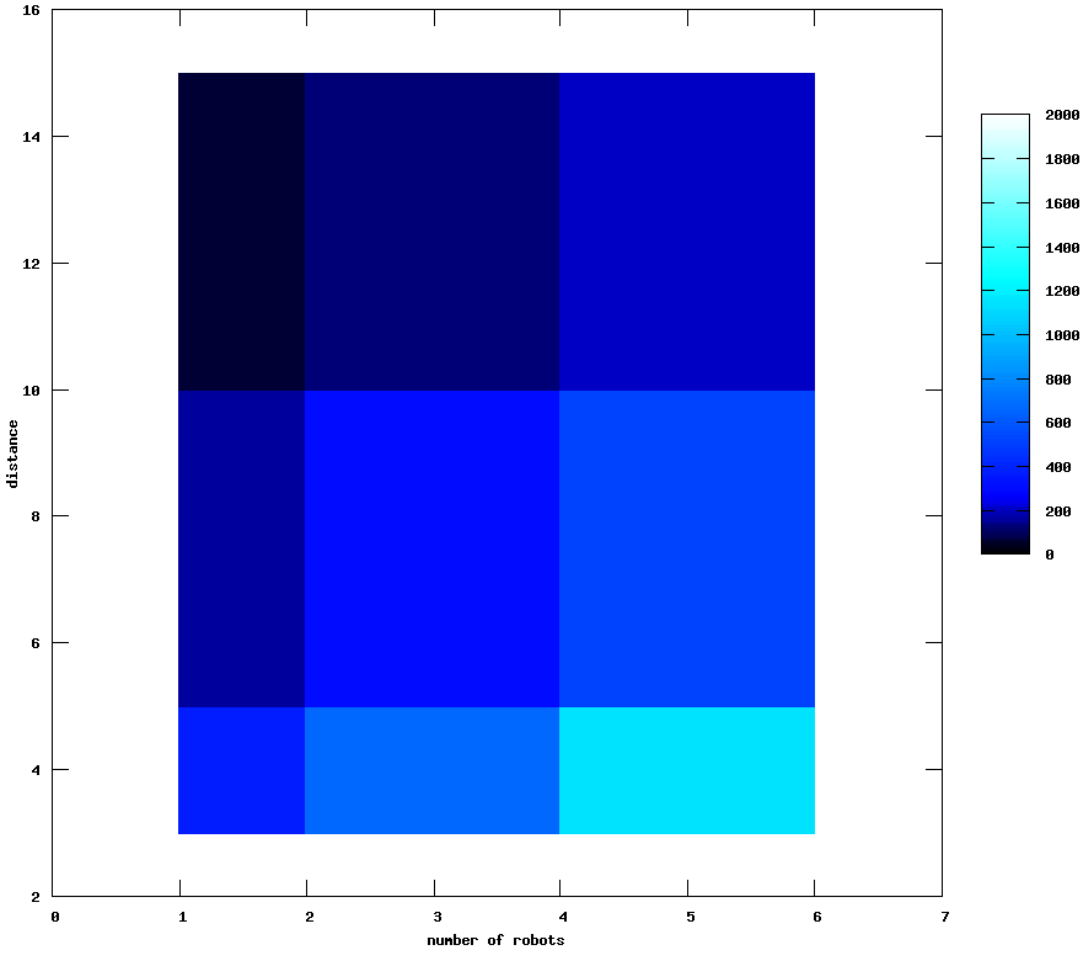

Figure 15 - Graphique montrant la linéarité des mesures du nombre de pixels colorés de la caméra par rapport au nombre d'e-pucks dans son champ de vision.

3.6.2.2. Le sampling de la caméra

Dans le cas du sampling de la caméra vis-à-vis des e-pucks, le sampling, mené à bien par Mouhcine Zekkri, a été réalisé en enregistrant la quantité de pixels rouges de la caméra lorsque qu'elle avait en face d'elle un e-puck dont les 8 diodes led étaient allumées. Quatre distances ont été considérées : 3 cm, 5 cm, 10 cm et 15 cm.

Dans le cas des pucks, sept distances ont été prises en compte : 0 cm, 3 cm, 5 cm, 10 cm, 15 cm, 20 cm et 25 cm.

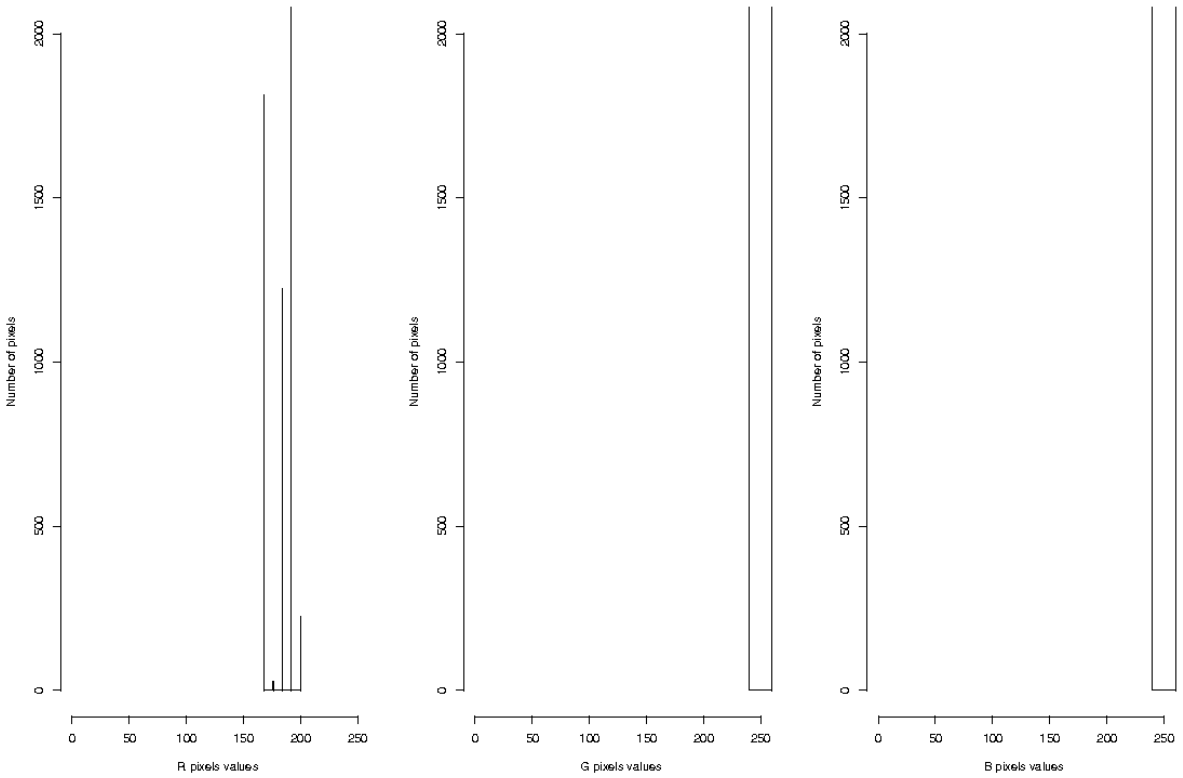

Figure 16 - Graphiques indiquant la quantité de pixels de chaque couleur parmi le rouge, le vert et bleu perçue par la caméra. L'e-puck qui prend cette photo a en face de lui un autre e-puck dont les leds rouges sont allumées et situé à 5 cm.

3.7. La gestion des collisions

La gestion des collisions a été développée en collaboration avec Giovanni Pini. Jusqu'à présent, les e-pucks n'ont été présenté que comme des entités logiques au sein du simulateur. Ils n'ont pas de corps, pas de consistance. La gestion des collisions permet de leur fournir cette consistance. Pour ce faire, plusieurs classes ont été crées pour décrire les éléments de base d'une collision. On trouve la super-classe *CCollisionObject* et plusieurs classes en héritant, chacune associée à une forme géométrique basique : *CCircleCollisionObject*, *CRectangleCollisionObject* et *CRingCollisionObject*.

Les e-pucks devant avoir une gestion des collisions sont représentés par la classe *CCollisionEpuckSimple* qui hérite de *CCollisionEpuck* qui elle-même

hérite bien sûr de *CEpuck*, dans la mesure où l'objet créé est et reste un e-puck. Cet e-puck "solide" est constitué d'un unique objet *CCircleCollisionObject* étant donné la forme du robot.

Dans le même esprit, les murs d'une arène rectangulaire seront représentés comme étant des *CRectangleCollisionObject* et le mur d'une arène ronde comme un *CRingCollisionObject*.

Lorsque des collisions se produisent, le principe de base est de distinguer deux concepts : le repérage des collisions et le comportement à adopter en cas de collision.

Le repérage des collisions est exécuté par un objet de la classe CCollisionManager. Celui-ci identifie tous les objets de la simulation pouvant être sources de collisions et de vérifier pour chacun d'eux si une collision est survenue durant le pas de simulation en cours. Ses principales fonctions consistent, d'une part, à repérer les collisions avec les murs, et d'autre part, à repérer les collisions entre objets héritant de CCollisionObject.

Une fois les collisions identifiées, le simulateur fait appel au "collision handler". Ce nom ne désigne pas un objet ou une classe mais une bibliothèque contenant les fonctions à appliquer pour gérer les collisions qui se sont produites. Dans l'état actuel du simulateur, deux logiques de repositionnement après collision ont été implémentées dans le collision handler.

La première logique de repositionnement consiste à remettre l'e-puck à l'endroit qu'il occupait à l'instant précédent. Cette approche ne conduit pas nécessairement à reproduire sans cesse la même situation pour peu que les déplacements de l'e-puck aient une composante aléatoire, que ce soit directement dans la cinématique ou indirectement via des valeurs de capteurs qui induisent des déplacements différents.

La seconde logique de repositionnement est une logique de rebond sur l'obstacle. Quand un e-puck rencontre un mur, il va rebondir dessus, c'est-à-dire qu'il va reculer d'une courte distance dans une direction qui n'est pas nécessairement celle avec laquelle il était arrivé contre le mur. Il ne reculera

dans la même direction que si son déplacement était perpendiculaire à la face du mur ou à la tangente de celui-ci si l'arène est ronde.

3.8. Les rendus graphiques

Afin de rendre compte de l'état de la simulation en cours, différents rendus graphiques ont été implémentés. A l'exception du rendu "null", tous utilisent les librairies et les fonctions d'OpenGL afin de créer et de visualiser les éléments de l'environnement.

Une particularité de Twodeepuck, héritée de Twodee, est que les rendus sont responsables de la gestion du temps de la simulation et qu'ils gèrent la boucle qui égrène les pas de simulation.

Les différents rendus créés héritent de la super-classe *CRender* et, à ce titre, doivent redéfinir obligatoirement les méthodes de lancement de la simulation et d'affichage de l'arène, de l'e-puck, du puck.

Un premier "rendu graphique", s'il est permis de l'appeler comme cela, est le rendu modélisé par la classe *CNullRender*. Comme son nom l'indique, ce rendu n'affiche rien du tout. Le corps des méthodes à redéfinir est vide.

Le rendu de base est modélisé par la classe *CSimpleDrawStuffRender* dont une capture d'écran est visible à la Figure 17. L'e-puck et le puck y sont représentés sous la forme d'un cylindre noir. L'arène, obligatoirement de type rectangulaire avec ce rendu, y est représentée comme un ensemble de boîtes rectangulaires. On affiche également les abris et les sources de lumières au sein de la méthode d'affichage de l'arène.

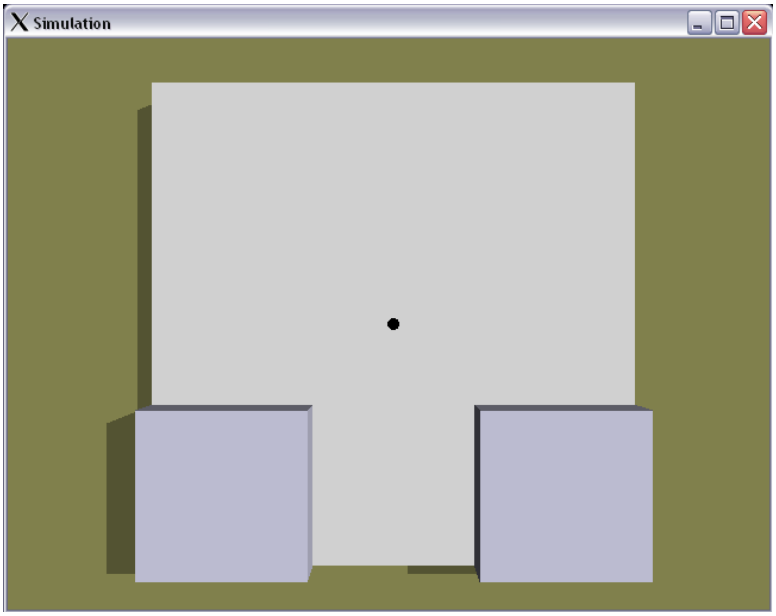

Figure 17 - Rendu graphique de base de Twodeepuck. L'arène représentée ici est une arène de type rectangulaire ayant 2 blocs obstacles (inférieurs droit et gauche). L'e-puck est le point noir au centre (il s'agit d'un cylindre, vu du haut).

Ce rendu permet d'afficher certaines fonctionnalités supplémentaires en ce qui concerne l'e-puck. Ainsi, il est possible d'afficher des lignes droites représentant les directions des capteurs de proximité. Il est également possible de représenter le champ de vision de la caméra et le capteur de lumière ambiante. Enfin, si l'e-puck est muni d'une pince, celle-ci sera également dessinée.

Pour les simulations dont l'arène est ronde, la classe *CRoundArenaRender* a été créée (voir Figure 18). Elle hérite de *CSimpleDrawStuffRender* et se contente de redéfinir la méthode d'affichage de l'arène.

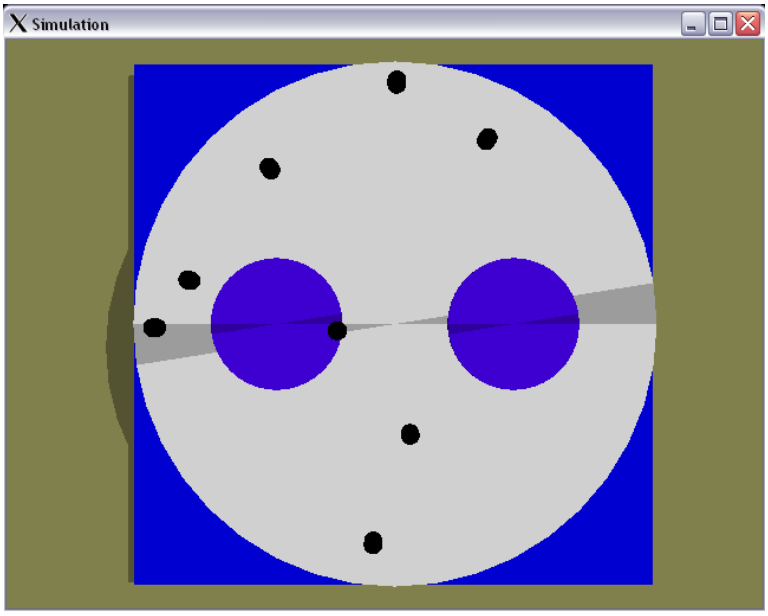

Figure 18 - Rendu graphique de Twodeepuck lors d'une expérience de choix collectif. On peut y voir 8 e-pucks et 2 abris.

Une dernière classe, CEpuckArrowRender, héritant de CRoundArenaRender, permet d'afficher les capteurs de proximité de l'e-puck et d'en faire varier la couleur selon qu'ils ont détecté un obstacle ou pas.

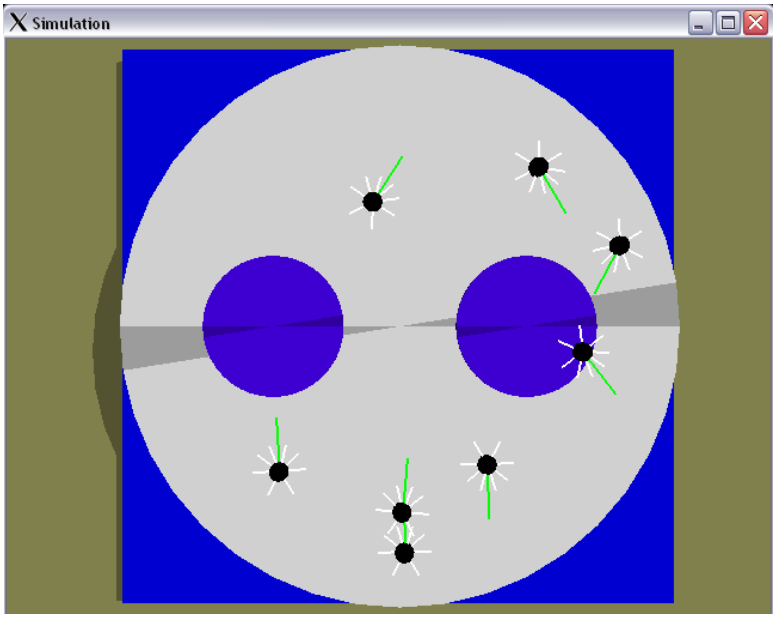

Figure 19 - La même expérience que sur la figure 16 mais cette fois avec le rendu graphique montrant les capteurs de proximité et la direction frontale des e-pucks.

3.9. Diagramme de classe

On le voit, Twodeepuck reprend assez fidèlement les concepts de base de TwoDee et y ajoute ses propres spécificités. Le diagramme de classes simplifié du programme tel qu'il existe à l'heure actuelle est présenté à la Figure 20.

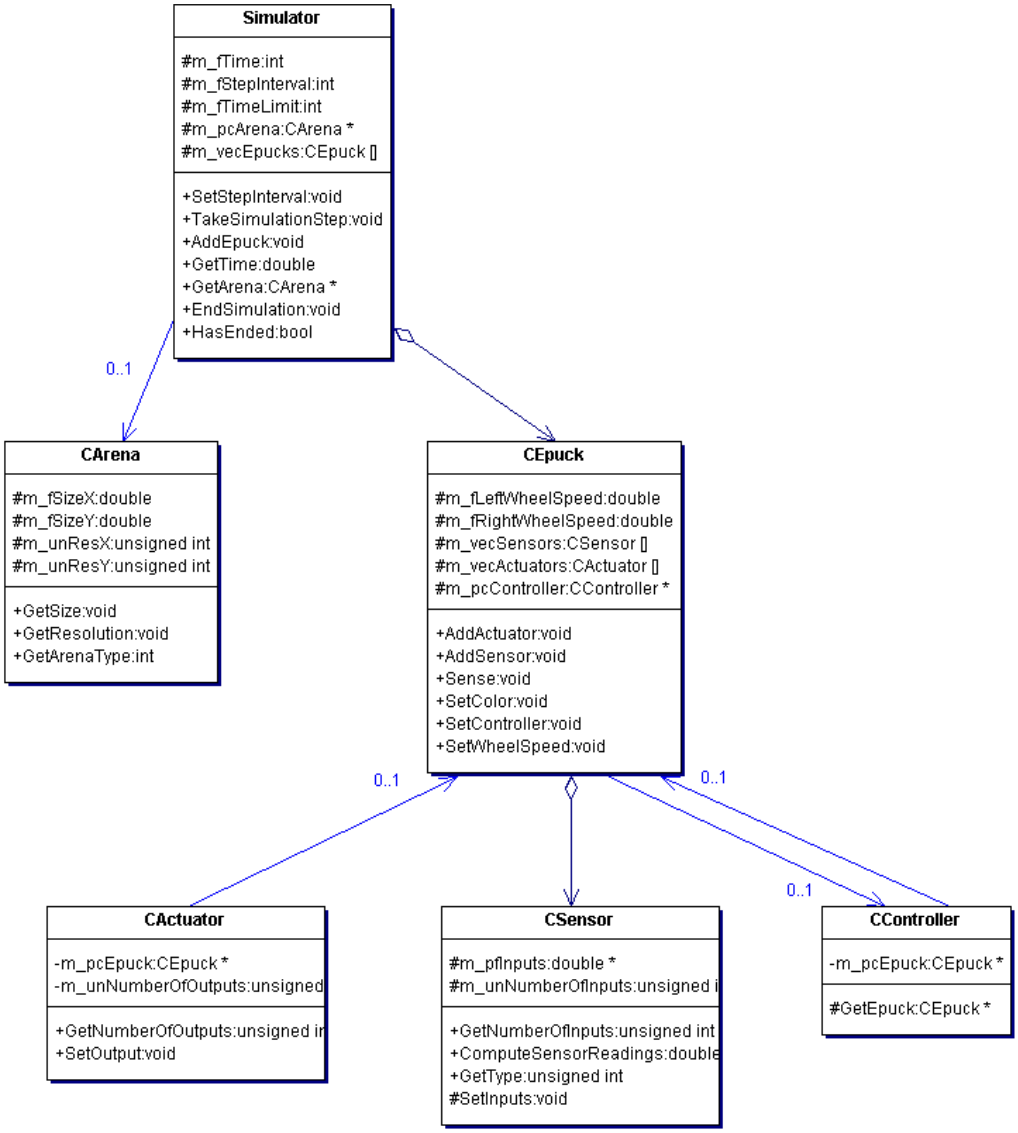

Figure 20 - Diagramme de classes simplifié de Twodeepuck.

4. Utilisation et validation

4.1. Qui seront les utilisateurs ?

L'intérêt d'un simulateur de robots est d'être mis à la disposition de chercheurs en robotique. C'est précisément dans cette optique que Twodeepuck a été conçu. La démarche avec les autres étudiants du projet e-puck a toujours été la même. Ceux qui utilisaient les robots afin de leur faire adopter des comportements précis dictaient au fur et à mesure les besoins qu'ils auraient dans le futur vis-à-vis de Twodeepuck. Un simulateur doit être adapté à l'environnement réel qu'il simule et non l'inverse.

Etant donné que le projet e-puck a été mis en place au début de l'année, Twodeepuck a d'abord servi le petit groupe d'étudiants qui travaillait sur ce projet. Toutefois, il est évident qu'il reste encore beaucoup à faire pour ce projet, tant au niveau du simulateur qu'au niveau des recherches sur les comportements des vrais robots. Par conséquent, Twodeepuck est destiné à être utilisé par les chercheurs et les étudiants de l'IRIDIA. Le fait de s'être inspiré de TwoDee entre tout à fait dans cette optique vu que TwoDee a été développé à l'IRIDIA et y est couramment utilisé.

Dans une vue à plus long terme, Twodeepuck est destiné à être utilisé par n'importe quelle communauté de chercheurs s'intéressant aux e-pucks. Cet objectif est d'autant plus réaliste que Twodeepuck a été développé dans un esprit opensource. On peut même dire que l'ensemble du projet est orienté opensource car il n'y a pas que le simulateur qui entre dans cette logique. En effet, on trouve en libre accès sur internet tous les

renseignements que nous possédons à propos de l'e-puck, et ce à travers le wiki mis en place à l'IRIDIA.

On le voit, Twodeepuck s'inscrit clairement dans une logique de partage. A ce titre, il convient de préciser la manière de l'utiliser. Pour les renseignements techniques à propos de la façon dont il a été développé, il suffit de se référer au Chapitre 3 du présent travail. Nous n'allons pas revenir là-dessus, passons plutôt à la façon dont un utilisateur doit manipuler Twodeepuck.

4.2. Utilisation de Twodeepuck

Pour pouvoir utiliser Twodeepuck, il convient de comprendre la logique employée dans l'élaboration des fichiers du programme. Ces fichiers se répartissent en trois grandes catégories : les fichiers de configuration, les fichiers des classes principales et les fichiers des sous-classes. Il existe également quelques fichiers fournissant des fonctions à caractère très général, comme la gestion des vecteurs ou des fonctions mathématiques. Ceux-ci ne seront que très brièvement mentionnés.

Les fichiers de configuration servent à permettre au compilateur de faire son travail convenablement. Les exécutables *bootstrap.sh* et *configure* servent à la configuration des outils de compilation autoheader, automake et autoconf. L'exécutable *configure* va chercher certains de ses renseignements dans le fichier *configure.ac* dont l'usage sera expliqué ci-après. Enfin, le fichier *Makefile.am* sert à donner les instructions de compilation au programme *make*.

Les fichiers des classes principales sont les fichiers contenant la définition de classes symbolisant des aspects généraux du simulateur. La plupart d'entre-elles sont destinées à être dérivées par des sous-classes afin de rendre compte des spécificités du robot à simuler. Ces classes sont définies dans les fichiers suivants : *actuator.h*, *arena.h*, *controller.h*, *epuck.h*, *experiment.h*, *fitnessfunction.h*, *geometry.h*, population*.h*, *puck.h*, *render.h* et *sensor.h*. D'autres fichiers définissent des classes n'étant pas destinées à être dérivées : *individual.h*, *simobject.h* et *simulator.h*.

Les fichiers des sous-classes définissent des sous-classes des classes principales pouvant être dérivées. On y trouve donc les définitions de différentes arènes héritant de CArena défini dans arena.h, les définitions de plusieurs contrôleurs héritant de CController défini dans controller.h, et ainsi de suite…

Enfin, il existe quelques fichiers fournissant des fonctions d'utilité générale tels que *simmath.h* définissant la notion de vecteur et fournissant des fonctions de manipulation pour ces vecteurs, *random.h* fournissant la gestion des nombres aléatoires et *misc.h* mettant à disposition des fonctions qui ne touchent ni aux mathématiques ni aux nombres aléatoires.

Dans un souci évident de clarté, ces fichiers ont été classés et rangés selon un certain ordre. Ainsi, on trouvera tous les fichiers des sous-classes dans des répertoires séparés du reste des fichiers (configuration, classes principales et fonctions d'utilité générale), placés à la racine de l'arborescence du programme. Les fichiers des sous-classes sont rangés dans des répertoires séparés suivant la super-classe dont ils héritent. On trouve donc le dossier *actuators* pour les différents activateurs, *arenas* pour les arènes, *controllers* pour les contrôleurs, et ainsi de suite… Cette séparation entre les sous-classes et le reste des fichiers s'avère très pratique dans la mesure où on instancie rarement une autre classe que l'une des sous-classes. Par conséquent, si un utilisateur veut des renseignements sur une classe qu'il désire employer, il lui suffit de se rendre dans le dossier portant le nom de l'élément générique et d'y chercher le fichier ayant le nom de l'élément spécifique qu'il instancie. La seule exception est l'instanciation d'un unique objet de la classe CSimulator définie dans *simulator.h* afin de créer la simulation proprement dite.

4.2.1. Les manipulations d'un utilisateur

Cette section décrit les manipulations à exécuter afin de se servir de Twodeepuck en tant que simple utilisateur. C'est-à-dire pour quelqu'un désirant mettre en place son propre setup expérimental et son propre contrôleur pour les robots.

Les manipulations à mener sont très brèves et très simples à comprendre en raison du peu de code à écrire afin de simuler l'expérience voulue. Comme dit ci-dessus, cela ne concerne que deux éléments du simulateur, le setup expérimental et le contrôleur, qui sont décrites ci-dessous.

4.2.1.1. Le setup expérimental

Aussi dénommé "experiment", le setup expérimental est l'ensemble des paramètres définissant la simulation à mener. Il est possible d'y régler le nombre de robots, le nombre de pucks et pour chacun d'eux le comportement à adopter en cas de collision. Cela permet également de redéfinir les méthodes par défaut, définies dans la super-classe CExperiment (dans le fichier experiment.h à la racine de l'arborescence du programme, voir ci-dessus). De la sorte, il est possible de définir la liste des activateurs et des capteurs dont seront munis les e-pucks, leur contrôleur, l'arène à utiliser (rectangulaire ou ronde) et ses dimensions, et enfin la méthode qui donne les positions et orientations initiales des e-pucks et des pucks.

En l'absence de redéfinition, les méthodes de la super-classe CExperiment seront appliquées. Celles-ci mettent en place le setup expérimental suivant. Aucun puck n'est créé. Un seul e-puck est créé, à la position (0,0) avec une orientation nulle, sans gestion des collisions et doté de l'activateur des roues (CWheelActuator), de capteurs de proximité (CEpuckProximitySensor) et du contrôleur d'évitement d'obstacle (CObstacleAvoidanceController). L'arène créée est une arène rectangulaire de 3 blocs de 1 mètre de côté avec le bloc supérieur droit comme obstacle.

4.2.1.2. Le contrôleur

Le contrôleur est la logique permettant de définir les instructions de déplacement de l'e-puck à chaque pas de simulation.

Par défaut, le contrôleur se référera à sa super-classe, CController. Celle-ci contient extrêmement peu de fonctions : un constructeur, un destructeur et une fonction permettant de récupérer l'e-puck dont le

contrôleur définit les mouvements. On le voit, le contrôleur par défaut ne définit aucune logique pour les déplacements du robot.

L'utilisateur qui crée son propre contrôleur a évidemment envie de pouvoir écrire cette logique. Cela se fait de manière fort simple. Comme expliqué à la Section 2.2, le contrôleur d'un e-puck est lié à celui-ci par une relation de parent à enfant, le contrôleur étant un enfant de l'e-puck. Cela implique que la fonction *SimulationStep* du contrôleur sera appelée à chaque pas de simulation car l'e-puck provoque l'appel à cette fonction chez tous ses enfants. Si celle-ci n'est pas redéfinie, ce qui est le cas dans la super-classe CController, c'est la fonction par défaut définie dans la classe CSimObject qui est appliquée et celle-ci provoque simplement l'appel elle-même chez tous les enfants de l'objet. Ceci est inintéressant pour un contrôleur car celui-ci n'a, en principe, pas d'enfant. Donc, pour définir une logique de déplacement à l'e-puck, il suffit de redéfinir la fonction *SimulationStep* dans le contrôleur, fonction qui sera appelée par l'e-puck lui-même à chaque pas de simulation. Il est alors possible de faire réagir l'e-puck en fonction des valeurs de ses capteurs ou en fonction du temps dans la simulation, celui-ci étant un des paramètres de *SimulationStep*.

Un exemple simple de contrôleur est celui qui implémente l'évitement d'obstacle. Son fonctionnement est ici décrit dans le cas d'une expérience à un seul e-puck dans une arène ronde.

A chaque pas de simulation, le simulateur met à jour les valeurs des capteurs de l'e-puck. Une fois ceci fait, tous les enfants du simulateur réagissent grâce à la logique implémentée au sein de leur fonction SimulationStep ou au sein de celle définie par défaut qui se contente de s'auto-appeler sur les enfants de l'objet courant. L'e-puck va appeler la fonction SimualtionStep de ses enfants et en particulier de son contrôleur. Celui-ci, afin de réaliser l'évitement d'obstacle, récupère les valeurs des huit capteurs de proximité de l'e-puck. Sur base de ces valeurs et de la position des capteurs, le contrôleur appelle la fonction *SetSpeed* de l'activateur modélisant les roues du robot en provoquant un virage de l'e-puck si besoin est. Quand l'objet e-puck reprend la main après cet appel à *SimulationStep* du contrôleur, il met à jour sa position en fonction des vitesses de ses roues et de son orientation.

4.2.1.3. Les *Makefile*

Le dernier point important pour un utilisateur désireux de créer sa propre expérience dans Twodeepuck est la gestion des *Makefile*. Ceux-ci définissent la façon dont l'exécutable *make* va compiler le programme.

Lorsque l'utilisateur aura créé ses propres fichiers décrivant son setup expérimental et son contrôleur, il doit signaler au compilateur qu'il faut en tenir compte. Si ce n'est pas fait, le compilateur échoue dans sa compilation en annonçant qu'il ne trouve pas les fichiers voulus.

Il convient alors de renseigner les fichiers source dans le fichier *Makefile.am* du répertoire où ils ont été créés et de renseigner les fichiers d'en-tête dans le *Makefile.am* à la racine de l'arborescence du programme.

4.3. Tâches réalisées avec Twodeepuck

Jusqu'à présent, le simulateur a déjà servi à modéliser plusieurs expériences de robotique.

L'évitement d'obstacles : cette expérience, très simple, consiste à faire évoluer plusieurs robots en leur implémentant une logique visant à ne pas provoquer de collisions. Ceci est concrètement réalisé à l'aide d'un contrôleur qui définit les mouvements des robots comme suit. Si il n'y a pas d'obstacle détecté, on applique une vitesse prédéfinie identique aux deux roues. Si un ou plusieurs obstacles sont détectés, on détermine de quel côté il y a le plus de danger et on fait tourner le robot de l'autre côté. Afin d'affiner ce comportement, il est possible de jouer sur la façon dont l'évitement doit se faire : tourner sur place (vitesses des roues égales en valeur absolue mais opposées en signe) ou décrire une courbe. Le choix de la solution est fonction des valeurs des capteurs : plus ceux-ci renseignent un danger proche, plus le virage sera serré.

La marche aléatoire avec arrêt sous abri : dans cette expérience, un robot a pour mission de se mouvoir dans une arène ronde, de façon aléatoire, et de s'arrêter lorsqu'il se trouve sous l'abri situé au centre de l'arène. L'abri mis en place consiste, dans l'expérience réelle, en une plaque de plexiglas munie d'un filtre coloré et d'étroites bandelettes de carton noir. Cela a pour effet de provoquer une zone d'ombre sous cet abri. Dans le simulateur, les abris ont été implémentés au sein de la classe CArena. Les conditions détaillées et les résultats de l'expérience sont décrits dans la Section 4.4 consacrée à la validation du simulateur.

L'aggrégation sous un abri parmi deux disponibles : cette expérience, détaillée dans le travail de fin d'études d'Olivier Dédriche, consiste à faire en sorte que tous les e-pucks de l'expérience se rassemblent sous un seul abri alors que deux abris sont disponibles, sans leur dire sous lequel ils doivent aller, bien entendu.

Les conditions sont les suivantes. Il y a 9 e-pucks. Les deux abris ont la même taille et peuvent abriter l'ensemble des e-pucks de l'expérience. Pour parvenir à obtenir le comportement voulu, les e-pucks exécutent une marche aléatoire lorsqu'ils sont en-dehors des abris. Quand ils entrent sous un abri, ils continuent leur marche aléatoire durant 20 pas de simulation, afin qu'ils pénètrent plus à l'intérieur de la zone abritée. Un fois à l'intérieur depuis plus de 20 pas de simulation, ils s'arrêtent, allument leurs 8 diodes leds rouges et, à intervalles réguliers, tournent sur eux-mêmes afin de prendre des photos de ce qui les entoure. Sur base de ces photos et grâce au fait que les robots sous l'abri ont leurs leds rouges allumées, ils peuvent calculer une approximation du nombre d'e-pucks déjà sous l'abri. Ce nombre de camarades les entourant leur permet de calculer une probabilité de sortie. Plus il y a d'e-pucks sous le même abri, moins la probabilité de quitter cet abri est élevée.

La plupart du temps, l'expérience aboutit bien à un état stable dans lequel tous les e-pucks sont rassemblés sous un seul des deux abris.

4.4. Validation

La validation d'un simulateur est une étape importante dans le développement d'un tel logiciel. En effet, le but d'un simulateur étant, par définition, de modéliser un environnement, il convient de s'assurer qu'il le fait convenablement.

Il y a plusieurs aspects qui méritent une validation pour un simulateur tel que Twodeepuck est conçu. Premièrement, on peut s'intéresser à la gestion du temps. Cet aspect est très important étant donné que le temps est au cœur de la dynamique des simulations à travers la vitesse et donc la position des robots à simuler. Ensuite, il est envisageable de faire une validation des aspects précis, non-aléatoires du simulateur. Cela revient à analyser des résultats dont la concordance avec le monde réel doit être sans faille. Enfin, pour les expériences nécessitant l'intervention d'un aspect aléatoire, un dernier type de validation peut être mené. En effet, cette validation nécessite une procédure particulière dans le sens où il faut faire ressortir le caractère statistique des résultats. Cette dernière catégorie de validation est celle qui a été menée avec Twodeepuck.

4.4.1. Que valider ?

La première réflexion à mener est de déterminer les aspects du simulateur qui devront faire l'objet d'une validation.

La validation de la gestion du temps a pu être faite de façon relativement simple en ralentissant la durée du pas de simulation afin de vérifier que le compteur du temps virtuel s'écoulait à une vitesse régulière et que les vitesses et positions des e-pucks concordaient parfaitement avec les prévisions.

La réalisation de la validation des comportements précis et prévisibles n'a pas été jugée utile. En effet, cela revenait à valider les mesures des capteurs. Or, ces capteurs prennent leurs données dans des fichiers de sampling, c'est-à-dire dans des fichiers dont les données sont directement extraites du monde réel. A partir du moment où le simulateur va bien

chercher les bonnes valeurs à la bonne place dans le fichier de sampling, il n'y a plus aucun risque de voir apparaître une erreur.

Il reste donc à valider le simulateur par une expérience faisant intervenir de l'aléatoire. Pour faire intervenir cet aspect, la méthode employée a été de faire plusieurs milliers d'expériences sur simulateur. Malheureusement, il est inconcevable de réaliser cela avec les vrais robots en raison des contraintes techniques et temporelles. Il faut donc choisir un nombre d'expériences, à mener dans la réalité, qui soit suffisamment petit pour être réalisable et suffisamment grand pour refléter le côté statistique de l'expérience.

4.4.2. Comment valider ?

L'expérience choisie est la marche aléatoire avec arrêt sous abri. L'arène est circulaire de rayon 1 mètre et a en son centre un abri circulaire de rayon 0,25 mètre.

La marche aléatoire a été définie de la façon suivante. Le robot avance en ligne droite durant un temps aléatoire, ensuite il exécute une rotation d'un angle aléatoire et il recommence le processus. Durant l'étape qui consiste à avancer en ligne droite, une logique d'évitement d'obstacle a été implémentée au cas où le robot rencontrerait le bord de l'arène. Afin de s'assurer que ce comportement était bien implémenté dans le contrôleur à la fois en simulation et sur les vrais robots, une machine à états finis a été élaborée. La Figure 21 présente cette machine à machine à états finis.

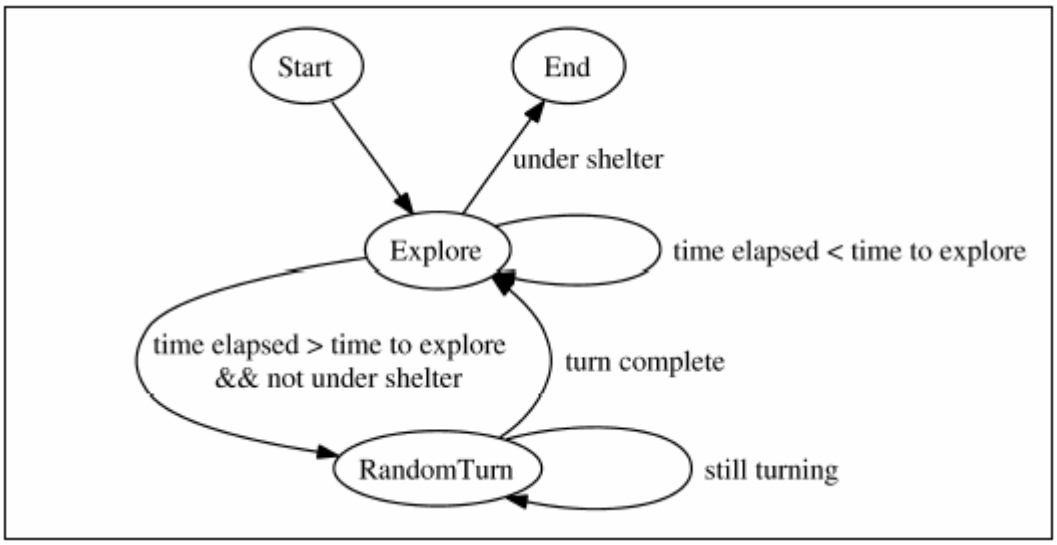

Figure 21 - Automate à états finis utilisé pour modéliser la marche aléatoire aux fins de validation du simulateur.

L'abri est fabriqué en plexiglas recouvert d'un filtre de couleur rouge et sur lequel ont été collées des bandelettes de couleur noires afin de créer une ombre suffisante sur le sol de l'arène. Vu que l'e-puck n'a de notion d'éclairement que grâce à ses capteurs infrarouges, l'ombre doit donc être une variation de la quantité d'infrarouges, ce qui a guidé notre choix vers le dispositif plexiglas-filtre-bandelettes.

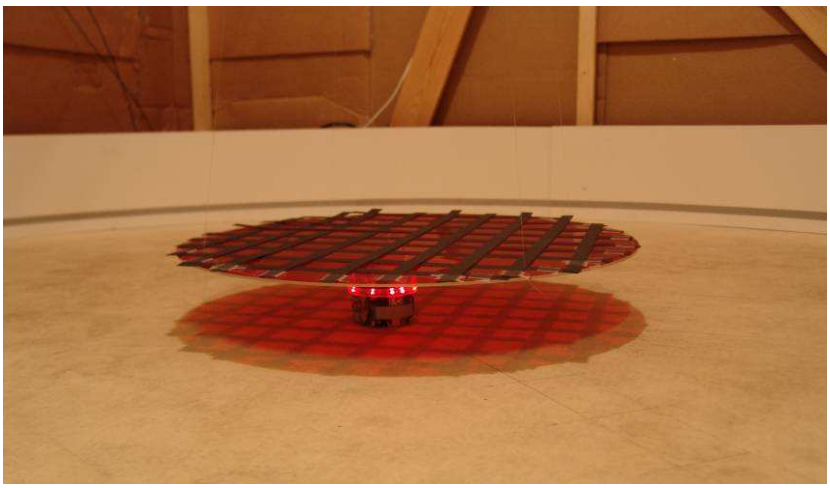

Figure 22 - L'abri pour les e-pucks. Il est constitué d'une plaque circulaire en plexiglas recouverte d'un filtre plastique rouge. Le tout est également recouvert par des bandelettes de carton noir afin d'augmenter l'ombre sous l'abri.

Un e-puck évolue dans l'arène de façon aléatoire tant qu'il n'est pas sous l'abri. Lorsqu'il se trouve sous cet abri, il s'arrête et l'expérience prend fin. Le principe est de mesurer le temps que met un e-puck avant de se trouver sous l'abri.

Etant donné que l'expérience nécessite l'utilisation des capteurs de proximité pour l'évitement d'obstacle et la détection de l'ombre, l'arène doit être fermée afin de ne pas faire entrer de lumière naturelle. La seule lumière pour cette expérience vient de deux ampoules de 100 watts. L'expérience est visualisée de l'extérieur grâce à une webcam installée dans l'arène.

Du côté du simulateur, l'expérience a été réalisée avec le setup expérimental tel que décrit ci-dessus. Elle a été répétée 10.000 fois afin d'obtenir des résultats statistiques fiables. On trouve sur la Figure 23 les résultats de cette expérience. L'abscisse représente le temps de survie et l'ordonnée représente le logarithme de la proportion de simulations encore en cours, c'est-à-dire dont l'e-puck n'est toujours pas passé sous l'abri.

Le coefficient de corrélation r^2 de la régression linéaire, donné sur la Figure 23, est très proche de l'unité ce qui prouve le caractère exponentiel attendu de la part d'une marche aléatoire. On peut donc dire que l'algorithme, tel qu'il a été implémenté sous la forme de la machine à états finis de la Figure 21, donne bien le comportement attendu en simulation.

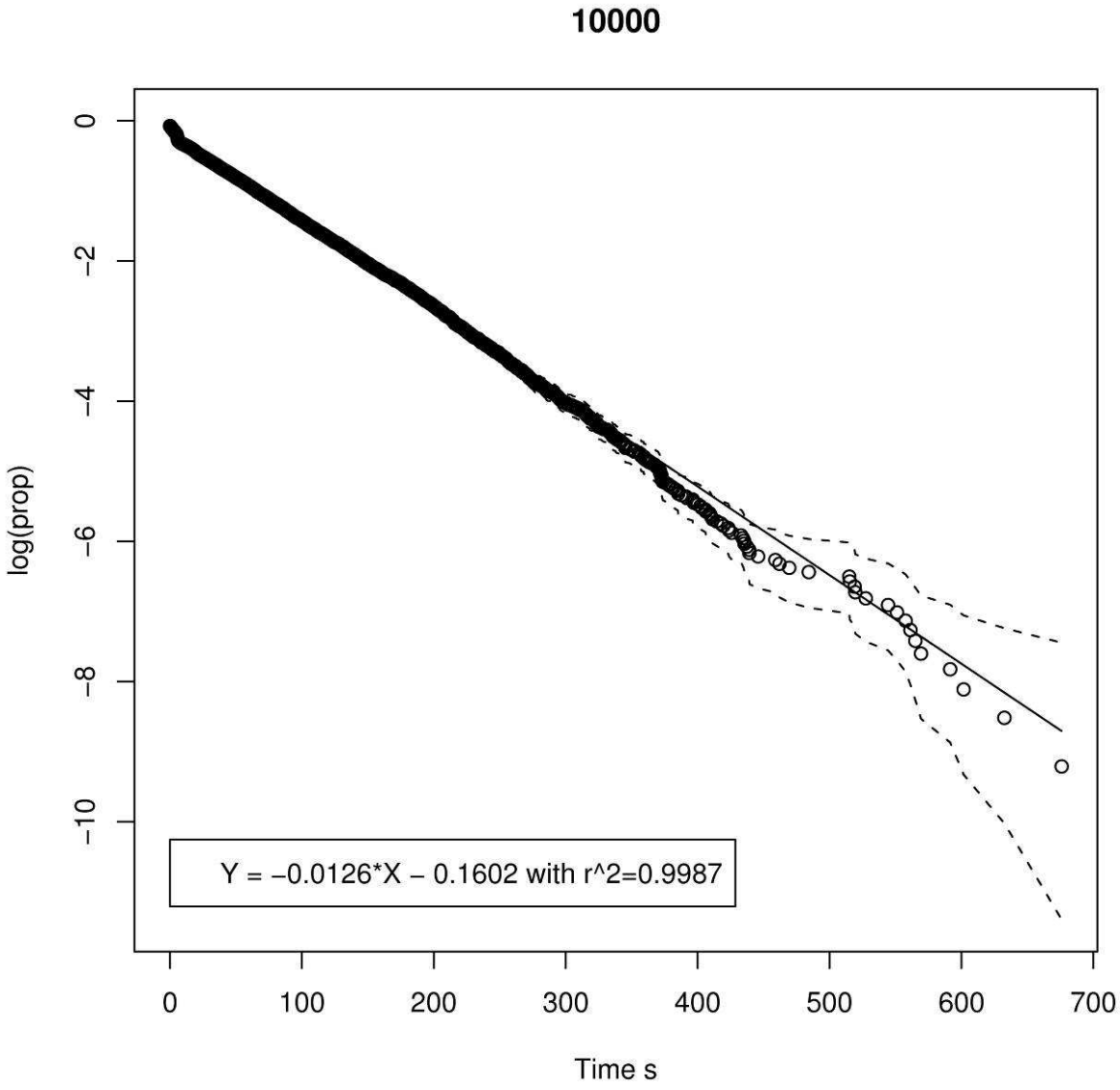

Figure 23 - Courbe de survie pour l'expérience de la marche aléatoire réalisée avec Twodeepuck, répliquée 10000 fois. La droite, dont l'équation est encadrée, est la régression linéaire de cette courbe de survie. L'abscisse représente le temps de survie (en seconde) et l'ordonnée représente le logarithme de la proportion de simulations encore en cours au temps correspondant. Les courbes en pointillés représentent l'intervalle de confiance à 95%.

La deuxième étape de la validation est de réaliser la même expérience avec les vrais robots. Afin d'obtenir des résultats fiables et, en même temps, réalisables, il a été choisi de répéter l'expérience 30 fois. Le contrôleur implémenté dans le vrai e-puck est identique à celui implémenté dans le simulateur (voir Figure 21).

Afin de déterminer les positions et orientations initiales des 30 expériences, il a été employé la même méthode que dans le simulateur.

On peut observer les résultats sur la Figure 24. A nouveau, comme sur la Figure 23, l'abscisse représente le temps et l'ordonnée représente le logarithme de la proportion de simulations encore en cours.

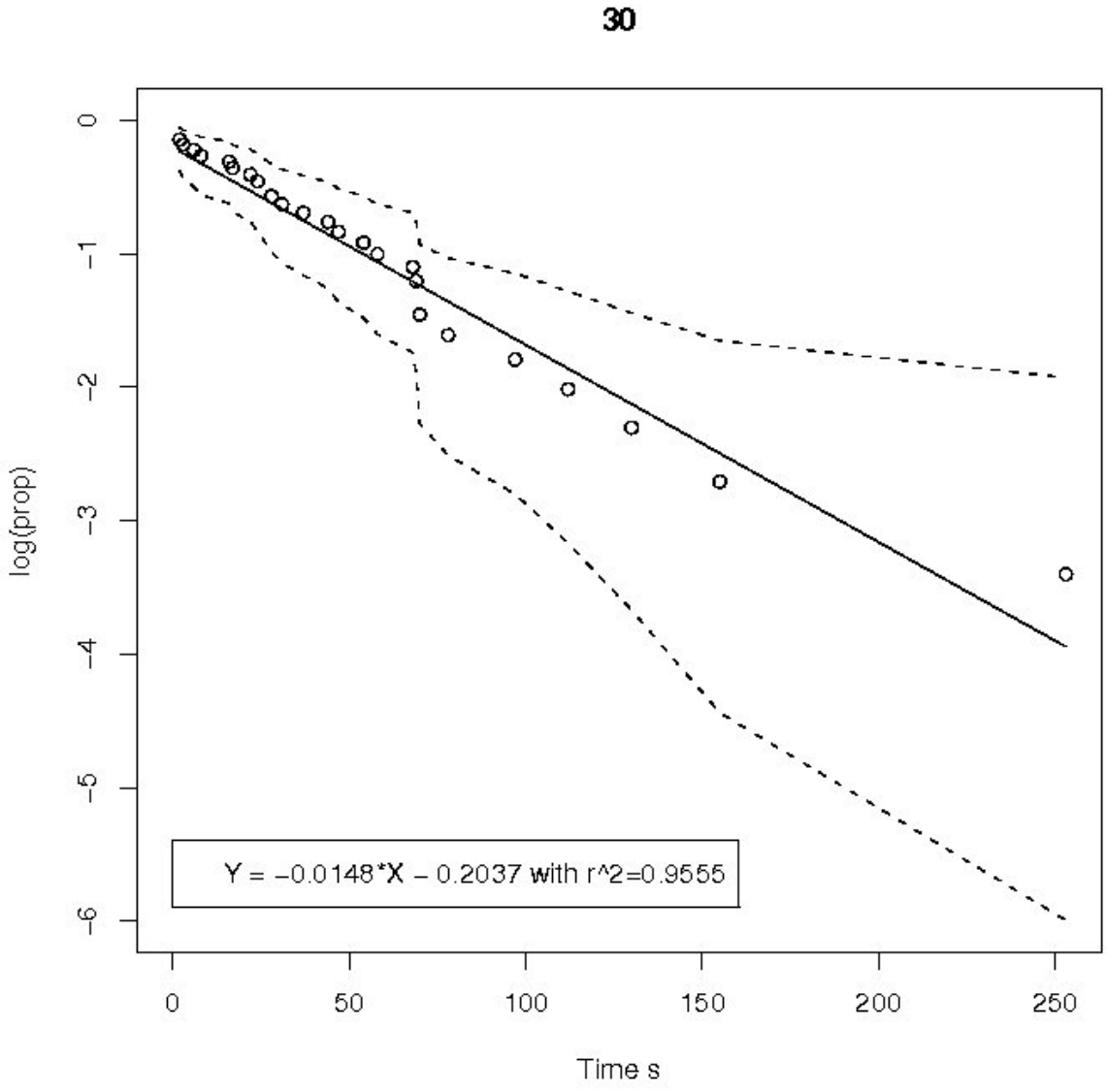

Figure 24 - Courbe de survie pour l'expérience de la marche aléatoire réalisée avec les e-pucks, répliquée 30 fois. La droite, dont l'équation est encadrée, est la régression linéaire de cette courbe de survie. L'abscisse représente le temps de survie (en seconde) et l'ordonnée représente le logarithme de la proportion de simulations encore en cours au temps correspondant. Les courbes en pointillés représentent l'intervalle de confiance à 95%.

On constate que les résultats sont, là encore, très satisfaisants. En effet, le coefficient de corrélation r^2 de la régression linéaire est très proche de l'unité.

En comparant les résultats issus de la réalité (voir Figure 24) avec ceux issus des simulations avec Twodeepuck (voir Figure 23), on constate que tous les deux aboutissent bien à un comportement de type marche aléatoire. Ceci se vérifie en remarquant que le coefficient de corrélation de la régression linéaire est proche de l'unité. Et comme le graphique représente un logarithme en fonction du temps, on a bien mis en évidence le caractère exponentiel que l'on attend de la marche aléatoire.

De plus, les deux implémentations de l'expérience donnent des résultats forts semblables au niveau des coefficients dans l'équation de la droite de régression.

Tout en étant bien conscient que ce travail de validation ne répond pas aux critères extrêmement rigoureux et fastidieux que demande une véritable validation, nous pouvons dire que la validation Twodeepuck est un succès.

5. Conclusions

5.1. Concepts-clé de Twodeepuck

Twodeepuck, basé sur TwoDee, est un simulateur destiné à modéliser le comportement et l'environnement des robots e-pucks. Pour ce faire, il procède en exécutant autant de fois que nécessaire la boucle suivante : perception, action, représentation. En premier lieu, le simulateur fait percevoir au robot son environnement. Ensuite, il le fait agir en concordance avec la logique implémentée dans son contrôleur. Et enfin, il met à jour la représentation graphique de la simulation en cours. Chaque pas de simulation représente un temps, typiquement 1/10ème de seconde.

Afin de faire en sorte que tous les éléments concernés puissent agir à chaque pas de simulation, Twodeepuck a une structure de type parent/enfant entre certaines classes. Par exemple, la simulation a pour enfants l'arène et les e-pucks et les e-pucks ont, à leur tour, pour enfants leurs activateurs, leurs capteurs et leur contrôleur. L'intérêt de cette structure est de faire en sorte que la mise à jour d'un parent, via la fonction *SimulationStep*, entraîne la mise à jour de ses enfants. Cela permet donc au simulateur de mettre à jour en cascade tous les éléments qui ont la possibilité de réagir à chaque pas de simulation.

Twodeepuck a été écrit en C++ sous un environnement Linux/Ubuntu. Sa compilation utilise les outils *autoheader*, *autoconf* et *automake*. Son rendu graphique est assuré par les librairies OpenGL.

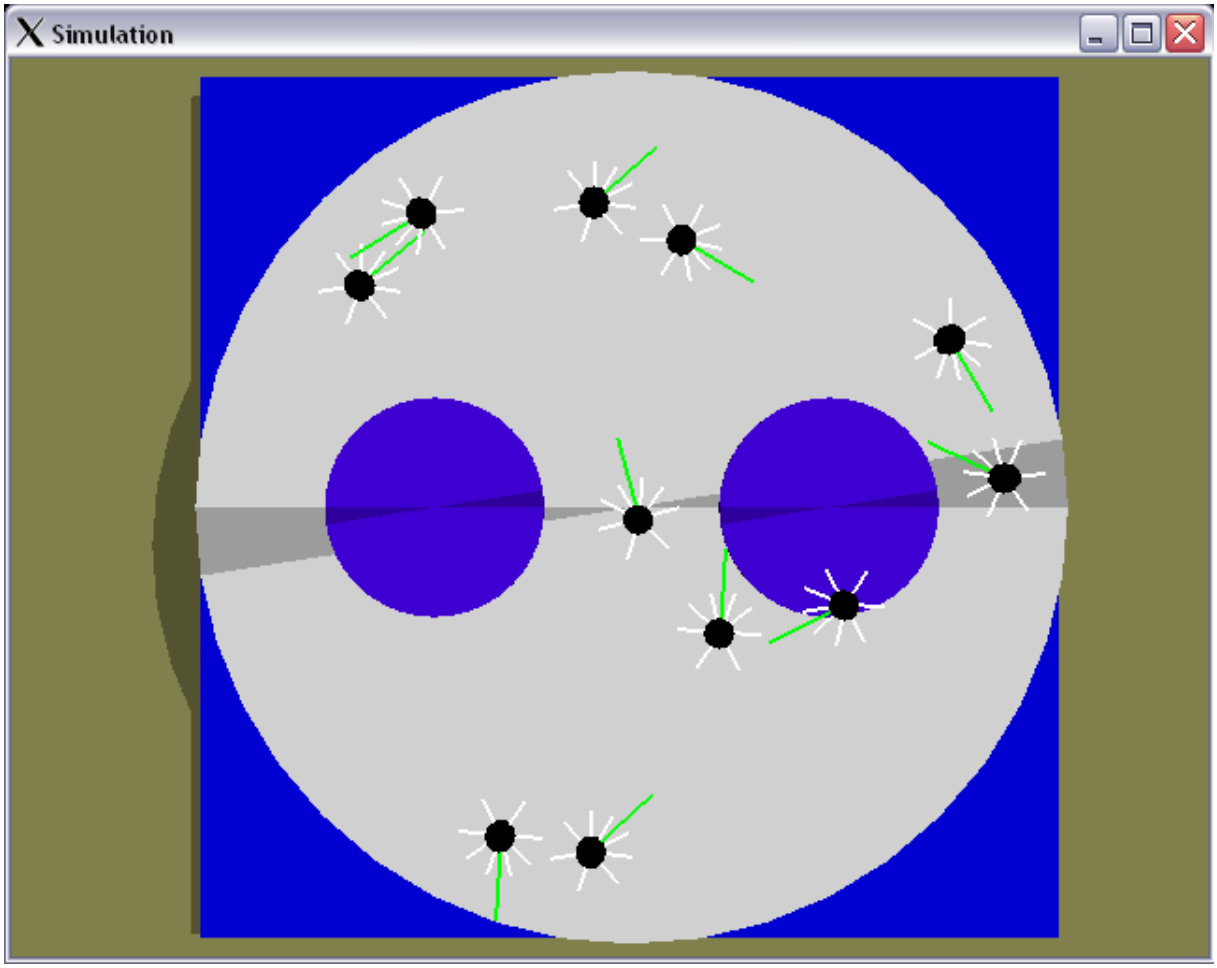

Figure 25 - Le simulateur Twodeepuck. On y voit 8 e-pucks et 2 abris. Le rendu graphique montre les capteurs de proximité et la direction frontale des e-pucks.

5.2. Spécificités de Twodeepuck

5.2.1. La gestion du sampling

Une des principales spécificités de Twodeepuck est d'utiliser des valeurs réelles pour les valeurs des capteurs de l'e-puck. Ces valeurs sont reprises dans des fichiers de sampling et ceux-ci sont intégrés directement dans l'exécutable du programme.

Le fait que les valeurs de sampling soient directement utilisées par le programme est un atout évident pour sa justesse par rapport aux situations

réelles. En effet, quoi de mieux que d'utiliser les valeurs que donnent les vrais robots dans des situations données afin d'en établir une modélisation ?

Concernant l'intégration des ces valeurs dans l'exécutable du programme, c'est un grand avantage en matière de portabilité. En effet, si les fichiers de sampling étaient restés des fichiers extérieurs, rangés dans des sous-répertoires précis, il aurait fallu déplacer ces fichiers en même temps que l'exécutable et veiller à conserver le même chemin à travers les répertoires pour relier l'exécutable et les fichiers de sampling. Avec Twodeepuck, ces données de sampling ont été converties en données dans des fichiers d'en-tête (fichiers .h) et sont donc intégrées dans l'exécutable comme tout autre fichier d'en-tête. Cela ralentit un peu la compilation lorsque le compilateur éprouve le besoin de reconsidérer les fichiers en question mais cela permet une meilleure portabilité de l'application. En effet, avec Twodeepuck, l'exécutable seul suffit à faire fonctionner tout le programme avec toutes ses fonctionnalités.

5.2.2. Un outil dans un projet plus vaste

Un autre aspect intéressant de Twodeepuck est la logique de travail en groupe dans laquelle il a été développé. En effet, le développement du simulateur s'inscrit dans le cadre d'un projet de mise en place d'une plate-forme d'utilisation des robots e-pucks. Cela signifie que le groupe de 6 étudiants que nous formions a dû prendre en main les e-pucks, découvrir leurs possibilités et imaginer des applications pratiques à réaliser avec ceux-ci.

Le simulateur s'est intégré dans ce projet comme étant un support utile, voire nécessaire, à apporter à ceux qui développaient des comportements de robotique en essaim avec les e-pucks. Twodeepuck a donc grandit, en suivant l'avancement du projet de groupe, en se voyant ajouter des fonctionnalités au fur et à mesure que les besoins s'en faisaient sentir auprès de ceux qui travaillaient avec les robots. Par exemple, la caméra et la gestion de celle-ci n'ont été intégré que lorsque d'autres membres du groupe se sont penchés sur les possibilités d'utilisation de cet outil sur les e-pucks pour leurs expériences.

Dans une vision plus large, Twodeepuck est destiné à être utilisé à l'IRIDIA par les chercheurs en robotiques s'intéressant aux robots e-pucks. Cette perspective a été un des éléments qui nous a fait décider de développer Twodeepuck à partir de TwoDee qui est un simulateur pour robots *S-bots* développé et utilisé couramment à l'IRIDIA.

Enfin, à plus long terme, le simulateur pourra être rendu disponible auprès de toutes communautés de chercheurs faisant des recherches avec les e-pucks. Ceci est valable également pour l'ensemble des connaissances acquises au sein du projet e-puck à l'IRIDIA. C'est dans cette optique que nous avons travaillé toute l'année en mettant à disposition nos recherches sur un site internet libre d'accès, hébergé par l'IRIDIA.

5.3. Validation

Le simulateur a été soumis à un test de validation afin de vérifier la concordance entre les simulations et le comportement des vrais robots. Cette validation a été effectuée en faisant une expérience qui consistait à observer le temps que met un e-puck pour arriver sous un abri central dans une arène ronde. Tant qu'il n'était pas sous l'abri, l'e-puck effectuait une marche aléatoire : répétition d'un avancement en ligne droite (avec évitement d'obstacle) pendant un nombre aléatoire de seconde et d'un virage d'un angle également aléatoire.

Les résultats ont été présentés dans les Figures 23 et 24 pour, respectivement, les 10000 simulations avec Twodeepuck et les 30 répétitions avec les vrais e-pucks. La comparaison des deux approches montrent qu'elles ont toutes les deux aboutit à la mise en œuvre d'un comportement que l'on peut qualifier de marche aléatoire. Et par ailleurs, les deux approches donnent des résultats très semblables entre elles.

La validation de Twodeepuck est donc un succès, tout en restant conscient du fait qu'il ne s'agissait pas là d'une véritable validation qui aurait demandé un respect bien plus rigoureux des différents critères entrant en ligne de compte.

Bibliograhie

[1] A. Campo, S. Nouyan, M. Birattari, R. Groß and M. Dorigo, *Negotiation of goal direction for cooperative transport*, ANTS 2006, Brussels, 2006.

[2] J.-M. Ame, C. Riveault and J.-L. Deneubourg, *Cockroach aggregation based on strain odour recognition*, Animal behaviour vol. 68, pp 793-801, Elsevier, 2004.

[3] R. Jeanson, J.-L. Deneubourg, A. Grimal and G. Theraulaz, *Modulation of individual behaviour and collective decision-making during aggregation site selection by the ant Messor Barbarus*, Behavioral Ecology and Sociobiology vol. 55, pp 388-394, February 2004.

[4] A. Colot, G. Caprari and R. Siegwart, *InsBot: design of an autonomous mini mobile robot able to interact with cockroaches*, IEEE Conference on Robotics and Automation 2004 vol. 3, pp 2418-2423, May 2004.

[5] S. Scholes, M. Wilson, A.B. Sendova-Francks and C. Melhuish, *Comparisons in Evolution and Engineering: The Collective Intelligence of Sorting*, Adaptive Behavior vol. 12, pp 147-159, 2004.

[6] M. Wilson, C. Melhuish and A.B. Sevona-Francks, *Multi-object Segregation: Ant-like Brood Sorting Using Minimalism Robots*, 7th International Conference on Simulation Adaptive Behaviour, pp 371-372, 2002.

[7] O. Holland and C. Melhuish, *Stigmergy, self-organisation, and sorting in collective robotics*, Artifical Life, 5, pp 173-202, 1999.

[8] C. Melhuish, O. Holland and S. Hoddell, *Collective sorting and segregation in robots with animal sensing*, 5th International Conference on Simulation Adaptive Behaviour, pp 465-470, 1998.

[9] T.H. Labella, M. Dorigo and J.-L. Deneubourg, *Division of Labour in a Group od Robots Inspired by Ants' Foraging Behaviour*, ACM Transactions on Autonomous and Adaptive Systems 1, pp 4-25, 2006.

[10] V. Triviani, T.H. Labella and M. Dorigo, *Evolution of Direct Communication for a Swarm-bot Performing Hole Avoidance*, Ants 2004, Brussels, pp 130-141, 2004.

[11] A.L. Christensen, *Efficient Neuro-Evolution of Hole-avoidance and Phototaxis for a Swarm-bot*, IRIDIA – Technical Report Series, October 2005.