Rapport du mini-projet

Systèmes embarqués et robotique

Groupe 11 - Ethvignot Marie/Ghez Fanny

1 Introduction

Ce rapport a pour but de résumer la conception, l'implémentation et l'analyse du mini-projet réalisé par notre groupe sur le robot e-puck, dans le cadre du cours de Systèmes embarqués et robotique donné par le Professeur Mondada. Le but de ce projet était de trouver une application à implémenter sur le robot en utilisant plusieurs de ses différents capteurs avec la librairie utilisée lors des travaux pratiques et les ressources étudiées en cours. Pour notre part, nous avons eu l'idée d'améliorer un jeu de notre enfance : le "Tamagotchi". Ce dernier est une mini "console" qui représente un animal de compagnie virtuel japonais, dont il faut s'occuper. En s'inspirant de ce jeu, nous avons décidé de transformer le robot e-puck en chien de compagnie !

2 Conception du mini-projet

2.1 Méthode de travail

Afin de réaliser ce projet de manière optimale dans le temps à notre disposition nous avons suivi plusieurs étapes de travail et nous sommes répartis les tâches. Tout d'abord, nous nous sommes focalisées sur la définition du projet en identifiant les fonctionnalités que nous voulions donner à notre chien et les capteurs du e-puck à utiliser pour cela. Ensuite il s'agissait de réfléchir à l'architecture de notre code et à comment nous allions utiliser de manière optimale des threads avec leur priorité et/ou temps de répétition. Enfin, nous nous sommes répartis le travail par capteur utilisé ou par fichier implémenté. En effet, l'une d'entre nous s'est occupée de gérer la détection des couleurs et les "émotions" du robot, tandis que l'autre s'est occupée de la gestion du mouvement (moteurs) et des capteurs de proximité infrarouges. Finalement, après avoir rejoint nos différents fichiers et nos différentes fonctionnalités nécessaires à la base du projet, nous nous sommes occupées ensemble d'essayer d'implémenter de nouvelles fonctionnalités, et de trouver des solutions aux problèmes rencontrés.

2.2 Présentation des tâches effectuées

Finalement, les tâches que nous avons réussies à implémenter sur le robot sont les suivantes : le "chien" se promène de manière totalement aléatoire dans un enclos ou dans un autre espace en changeant de trajectoire lorsqu'il rencontre des obstacles. De temps en temps, celui-ci aboie et de temps en temps il a faim. Si c'est le cas, le chien s'arrête d'avancer et d'aboyer, allume deux LEDs en rouge et attend qu'on le nourisse. Pour le nourrir, il faut placer devant lui sa gamelle de croquettes rouge. Une fois celle-ci détectée, le chien, satisfait, n'a plus faim, éteint les LEDS rouges et continue de se promener et d'aboyer. Dans notre mini-projet, nous utilisons alors les deux moteurs pas-à-pas, quatre capteurs de proximité infrarouges, la caméra et les hauts-parleurs.

3 Implémentation

Notre projet est constitué des cinq modules suivants :

- main
- motions
- emotions
- camera

- proximity_sensors

Le **main** contient tout d'abord de nombreuses fonctions d'initialisation, générales puis propres aux capteurs (IR, caméra...) ainsi qu'aux moteurs et aux speakers. Puis les fonctions dynamiques sont appelées, fonctions qui appelent elles-mêmes les threads qui implémentent les mouvements, la faim, la détection de couleur et les sons émis par le robot. Enfin, le main finit dans une boucle inifinie, qui est une bonne pratique dans ce genre d'applications où le programme n'est pas sensé s'arrêter.

Le module **motions** contient un thread *Wandering* qui s'occupe du mouvement du robot. Tout d'abord pour implémenter la promenade du chien nous utilisions une suite de mouvements allant tout droit, à droite et à gauche. Cependant ce mouvement n'était pas assez aléatoire à notre goût. Nous avons donc trouvé et utilisé le Random Number Generator de la librairie ChibiOs pour les microcontroleurs STM34 permettant de générer un nombre totalement aléatoire. Le thread *Wandering* s'effectuant toutes les 0.4 secondes, génère donc un nombre aléatoire (entre 1 et 4) et vérifie toutes les 0.4 secondes si la variable booléenne **enabled_motors** est vraie ou pas. Si c'est le cas, ce thread effectue le mouvement correspondant au nombre généré entre 1 et 4. Les 4 cas sont : avancer tout droit (présent dans deux cas sur quatre pour que le robot ait plus de probabilité d'aller tout droit), tourner à droite et tourner à gauche. Si la variable **enabled_motors** est fausse, les moteurs ne tournent pas.

Le module **emotions** est responsable de la gestion de la faim du chien et du son qu'il émet. La faim est gérée par un tread *Hungry* qui permet de passer une variable **hungry** à true toutes les 20 secondes, en vérifiant qu'elle ne l'est pas déjà. Ce changement d'état entraîne l'arrêt des moteurs (enabled_motors = false), l'allumage de deux LEDs rouges et l'arrêt des sons d'aboiement. Ces sons sont gérés dans un autre thread du module emotions : *Bark*. Il s'agit simplement d'un thread appelé depuis le main toutes les 3 secondes via la fonction **barking_sound**, permettant, si le chien n'a pas faim, d'appeler le thread *PlaySoundFile* de la librairie afin de jouer un son d'aboiement au format .wav présent sur la carte SD insérée dans le robot.

Le module **camera** implémente la capture d'image (thread *CaptureImage*) et la détection de couleur (thread *ProcessImage*) par le biais de la caméra P08030D du robot e-puck, au format 640x480 pxl RGB. Dans ce projet, seule la détection de la couleur rouge nous intéresse. Après que l'image ait été capturée par le premier thread, la sémaphore se libère pour indiquer qu'une nouvelle image est prête. Le second thread permet alors de filtrer la couleur rouge grâce à un masque, et la valeur de rouge de chacun des 640 pixels de la ligne est contenue dans la variable **image_red**. Enfin, une moyenne **moyenne_red** est calculée. Lorsque le robot est à l'état **hungry**, il compare la valeur de **moyenne red** avec une valeur THRESHOLD_RED qui indique un niveau suffisant de rouge lorsque la gamelle rouge est placée devant la caméra.

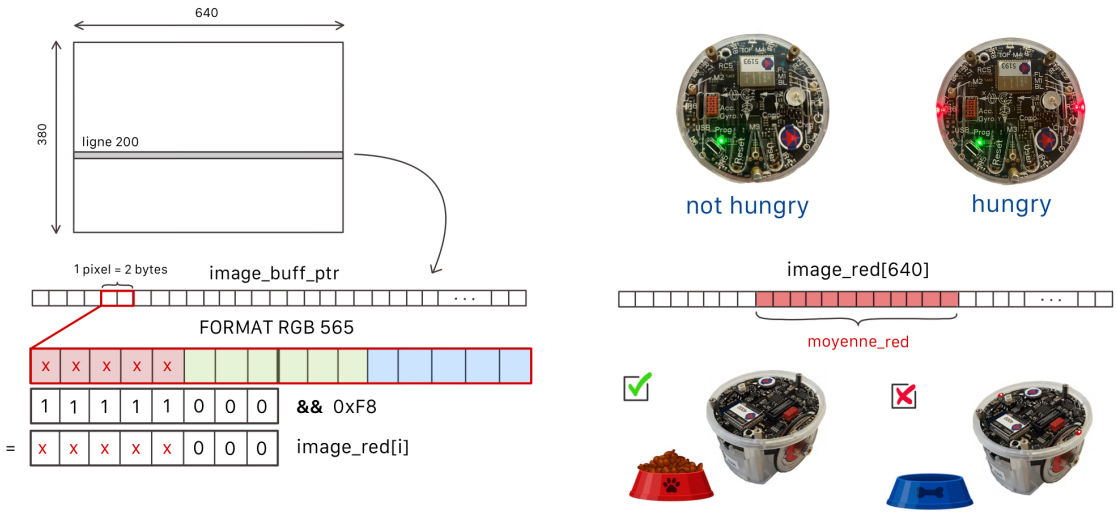

FIGURE 1 – Fonctionnement global de la détection de couleur

Le module **proxmity_sensors** s'occupe d'initialiser et d'utiliser les capteurs infrarouges de proximité afin d'éviter les obstacles. Cela se fait par le thread *CaptDistance* qui vérifie toutes les 100 millisecondes si la variable booléenne **hungry** provenant du module **emotions** (récupérée par la fonction **get_hungry**) est vraie ou pas. Si elle l'est, le robot est déjà à l'arrêt puisque la variable **enabled_motors** est fausse, et n'a donc pas besoin de vérifier la proximité des obstacles. Cependant, si le robot n'a pas faim, il est en mouvement et fait demi-tour si il détecte un obstacle à la distance DISTANCE_MIN. La distance (front_distance) qu'il compare avec DISTANCE_MIN est une moyenne des distances aux obstacles proches calculées par les quatre capteurs de proximité à son avant, à l'aide de la fonction **get_prox(sensor_number)** définie dans le module propre aux capteurs de proximité de l'e-puck et appelée dans ce thread.

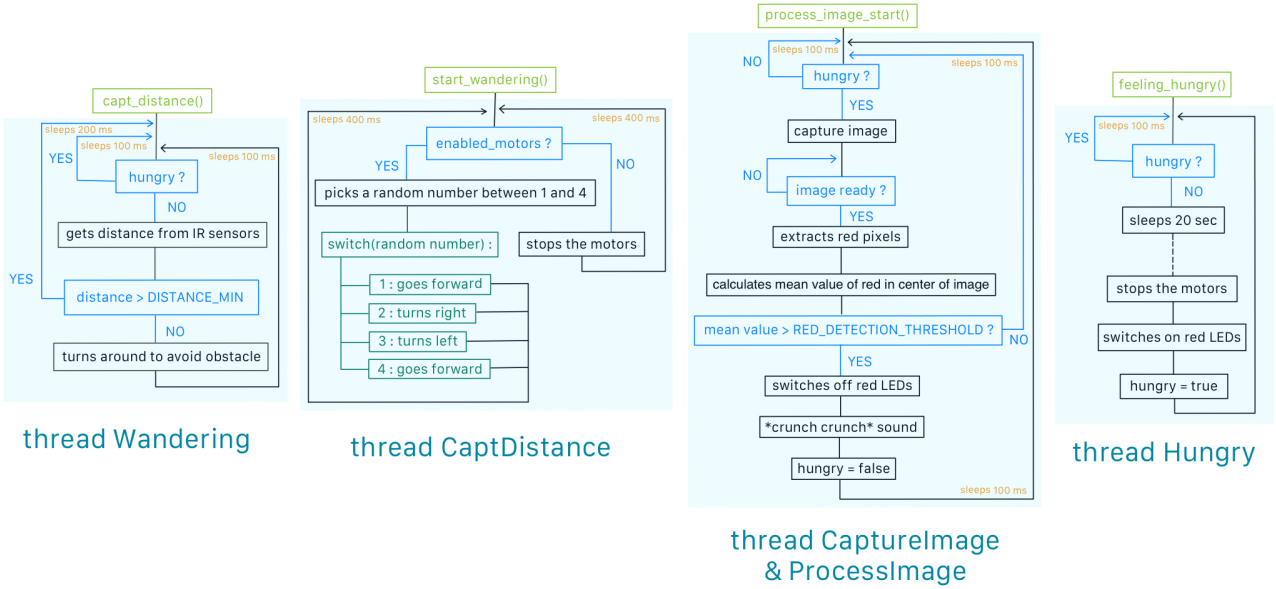

FIGURE 2 – Schémas logiques des différents threads

Tous les threads des modules cités ci-dessus ont la même priorité : NORMALPRIO. En effet, cela est nécessaire pour que les threads de notre projet puissent s'interrompre et se rendre la main. De plus, le thread *Bark* qui permet de jouer un son d'aboiement appelle le thread *PlaySoundFile* dans le module **play_sound_file** (audio) de la librairie, dont on ne pouvait donc pas changer la priorité.

4 Résultats obtenus et difficultés rencontrées

Nous avons rencontré de nombreuses difficultés avec la caméra en ce qui concerne la détection de couleurs. Les valeurs renvoyées par le robot et affichées par l'interface de RealTerm lors des tests ne corrélaient pas ou très peu avec les couleurs présentées devant la caméra. Une première modification a permis une certaine amélioration des résultats : désactiver la balance automatique des blancs via l'appel de la fonction **po8030_set_awb(0)**. Après cela, la détection des couleurs vertes et bleues n'était toujours pas très concluante mais celle du rouge correspondait assez bien. Dans notre première idée de conception de ce projet, nous voulions que le "chien" ait faim et soif. C'est-à-dire que lorsqu'il avait faim celui-ci attendait de détecter une gamelle de croquette rouge, et lorsqu'il avait soif une gamelle d'eau bleue. Cependant, commme nous n'avons pas réussi à débloquer le problème de la détection des couleurs pour le bleu et le vert, en partie à cause du manque de temps à notre disposition, nous avons dû abandonner l'idée que le chien ait aussi soif, car la détection du bleu n'était pas fiable.

Un second problème était la détection d'image lors d'objets en mouvement : dans un contexte statique, la condition de RED_DETECTION_THRESHOLD fonctionnait sans problème mais si un objet passait rapidement en mouvement devant la caméra, quelle que soit sa couleur, le robot pouvait détecter du rouge de manière erronée. Une solution au problème a été de créer 4 variables différentes (moyenne_red_0, 1, 2 et 3) pour stocker la valeur

moyenne de rouge d'une ligne pour 4 images successives, accompagnées d'un compteur **counter_image** permettant de ranger les valeurs dans les différentes variables. Puis de modifier la condition de détection du rouge pour que ces 4 moyennes de rouge dépassent la valeur de RED_DETECTION_THRESHOLD. Cette solution a permis de supprimer la sensibilité aux mouvements. Avant cette amélioration, une implémentation avec seulement 2 variables moyennes pour 2 images successives et un compteur avait été mise en place mais n'avait pas résolu le problème.

Un autre problème que nous n'avons pas réussi à résoudre était la détection d'une ligne noire. Dans notre conception de projet de base, nous voulions que le chien, lorsqu'il détectait un "chat" noir (représenté par une ligne noir d'une certaine épaisseur) se mette à le poursuivre, en se basant donc sur le TP4. Cependant, bien que la détection d'une ligne fonctionnait comme effectuée lors du TP4, la détection de **l'absence de ligne** quant à elle ne fonctionnait pas. La caméra percevait de manière trop sensible des lignes dans son environnement, et ce, même placé devant une feuille blanche. C'est à dire que le robot aurait été en état constant de poursuite d'un chat, même absent, ce qui aurait écrasé les autres fonctionnalités. Après plusieurs essais de modification et phases de test de la fonction de détection d'une ligne, nous avons finalement décidé de ne pas ajouter cette fonctionnalité au projet car trop complexe et coûteuse en temps, étant donné que la condition "rentrer en poursuite seulement lorsqu'une ligne est détectée" était trop difficile à satisfaire.

Enfin, nous voulions, au début de ce projet, implémenter plusieurs autres fonctionnalités (réaction à une certaine fréquence de son par exemple) que nous n'avons pas pu implémenter dans le temps imparti. Pour conclure, le manque de temps et le manque de fiabilité de la détection de couleurs ont été nos principales difficultés rencontrées. Néanmoins, nous sommes parvenues à réaliser les fonctionnalités de base que nous avions imaginé pour notre projet, et avons réussi à solutionner bon nombre de problèmes rencontrés au fil du projet. Nous sommes ainsi arrivées à une finalité simple mais totalement fonctionnelle et sans bug, ce qui nous a satisfaites.