TP1 : Ghostbusters
Programmation temps-réel

Groupe 2
Orphée Antoniadis Raed Abdennadher Steven Liatti

22 avril 2017

1 Introduction

Le but de ce travail était de réaliser un jeu qui est un mélange entre deux anciens jeux vidéo : le Pacman et le Casse-brique. L'objectif était d'apprendre à utiliser un RTOS et de comprendre son fonctionnement en mode coopératif. Pour ce faire, nous avions à disposition la carte Mylab2 accompagnée de la librairie FreeRTOS contenant les primitives du RTOS ainsi que la librairie myLab_lib contenant des fonctions utilitaires pour communiquer avec les périphériques de la carte.

2 Intêret de l'utilisation d'un RTOS

L'utilisation d'un RTOS nous facilite grandement la vie pour la gestion des tâches, l'OS gère "tout seul" leur ordonnancement : nous disposons ainsi d'un système multi-tâches clé en main. De plus, il nous fournit des mécanismes de synchronisation tels que les sémaphores. Avec tout ceci, nous avons pu nous concentrer sur le développement de la logique et mécanique du jeu uniquement, tout en sachant que cette partie temps réel/synchronisation était gérée par l'OS.

3 État du développement au moment du rendu

Toutes les spécifications du jeu demandées sont fonctionnelles, avec l'animation des fantômes en bonus. Il n'y a pas de problèmes connus également.

4 Description des tâches

4.1 Gestion du jeu

```
void game_task(void *arg) {
  while(1) {
    while (!ball->active) {
      //lcd_print(65, 160, SMALLFONT, FONT_COLOR, BACKGROUND_COLOR, "Press joystick");
      joystick_handler(check_start, TRIGGER);
      SLEEP(10);
    }
    lcd_print(65, 160, SMALLFONT, BACKGROUND_COLOR, BACKGROUND_COLOR, "Press joystick");
    xSemaphoreGive(sem_ball);
    xSemaphoreTake(sem_game, portMAX_DELAY);
  }
}
```

Lorsque la partie commence, le joueur doit appuyer sur le joystick pour commencer le jeu et lancer la balle. C'est cette tâche qui s'occupe de la gestion de cette fonctionnalité. Pour détecter la pression du joystick, nous avons été obligés de passer par une attente "semi-passive". La tâche va vérifier toutes les 10ms l'état du joystick puis se mettre en attente passive et laisser la main aux autres tâches. Lorsque le joystick est appuyé, la tâche initialise le nombre de vies ainsi que le score du joueur, sort de sa boucle, débloque le sémaphore sur lequel attendait la tâche de la balle puis attend lui-même sur un autre sémaphore qui sera débloqué lorsque le joueur aura perdu la balle 3 fois.

4.2 Gestion de la balle

```
void ball_task(void *arg) {
  while(1) {
    xSemaphoreTake(sem_ball, portMAX_DELAY);
    while(ball->active) {
      int x = ball->x;
      int y = ball->y;
      lcd_filled_circle(ball->x, ball->y, ball->radius, BALL_COLOR);
      collision_ball_wall();
      collision_ball_ghost();
      move_object(ball);
      SLEEP(10);
      lcd_filled_circle(x, y, ball->radius, BACKGROUND_COLOR);
      if (down_collision(ball)) lost_ball();
    }
    xSemaphoreGive(sem_game);
  }
}
```

Comme expliqué précedement, une fois que le joueur appuie sur le joystick, le sémaphore bloquant la tâche de la balle est débloqué et entre dans une boucle tant que la balle est ”active” (c'est-à-dire que le joueur a encore des vies). Dans cette boucle la tâche va d'abord dessiner la balle (un cercle de rayon 3 ici). Elle va ensuite vérifier s'il y a une collision avec les bords de l'écran, la raquette, ou un fantôme et si oui changer la direction de la balle en fonction. Les coordonées de la balle sont finalement incrémentées de STEP (ici STEP = 2) et la tâche se met en attente passive pendant 10ms afin de donner la main.
Lorsque la tâche reprend la main, la balle est effacée en redessinant un cercle de la couleur de fond là où était la balle. Nous vérifions ensuite s'il y a collision avec le bas de l'écran (donc si la balle est perdue). Si oui, le score est incrémenté, une vie est perdue et si le joueur est à 0 vies la balle devient ”inactive”. Après, soit la balle refait la boucle, soit elle libère la tâche de gestion du jeu et se rebloque elle-même dépendamment de si le joueur a perdu ou non.

4.3 Déplacement des fantômes

```
void ghost_task(void *arg) {
  ghost_t *ghost = (ghost_t*)arg;
  uint8_t change_dir = 0, change_img = 0, random;
  uint16_t x, y;
  while(1) {
    while(ghost->obj->active) {
      if (change_dir == 100) {
        ghost->obj->dir = direction_map[rand_direction()];
        change_dir = 0;
      }
      if (change_img == 5) {
        animate(ghost);
        change_img = 0;
      }
      x = ghost->obj->x;
      y = ghost->obj->y;
      collision_ghost_ghost(ghost->id);
      collision_ghost_wall(ghost);
      display_ghost(ghost);
      move_object(ghost->obj);
      SLEEP(ghost->speed);
      if (ghost->obj->active) update_ghost(ghost, x, y);
      else clear_ghost(x, y);
      change_dir++;
      change_img++;
    }
    SLEEP(20);
```

```
    random = rnd_32() % 100;
    if (random < 1) ghost->obj->active = true;
  }
}
```

Chaque fantôme est défini par une structure `ghost_t` ayant un pointeur sur la case du tableau `object` ainsi qu'un pointeur sur l'image bmp, l'index de de l'image à afficher (voir plus loin) et sa vitesse. Tant qu'il n'est pas touché par la balle, il est actif, il teste s'il n'entre pas en collision avec un mur ou avec un autre fantôme et sa nouvelle position est calculée (de la même manière que pour la balle). Une fois sa position mise à jour, il fait une attente passive correspondant à sa vitesse propre. Dans le cas où il n'y a pas de collision avec la balle, il est dessiné à sa nouvelle position, sinon il est "effacé" (`active` mis à `false`).
Dans le cas ou le fantôme est inactif, toutes les 20 ms il a 1% de chances de revenir dans le jeu. Tous les 100 cycles de vie d'un fantôme (relatif à sa vitesse), il change aléatoirement de direction entre le nord, le sud, l'est et l'ouest. Pour "animer" le fantôme, tous les 5 cycles (toujours relatif à sa vitesse), on alterne entre les deux images correspondantes à la direction actuelle du fantôme. Nous avons rajouté un jeu de deux images supplémentaire, simulant un fantôme de dos lorsqu'il va vers le nord. Lorsqu'il y a un changement de direction, l'accès au bon tableau des images est mis à jour en conséquences.

4.4 Gestion de la raquette

```
void racket_task(void *arg) {
  uint16_t last_x = racket.x;
  uint16_t last_y = racket.y;
  lcd_filled_rectangle(racket.x, racket.y, racket.x + racket.width, racket.y +
racket.height, RACKET_COLOR);
  while(1) {
    if (joystick_left_pressed(last_x, last_y) || joystick_right_pressed(last_x, last_y)) {
      last_x = racket.x;
      last_y = racket.y;
      SLEEP(8);
    } else {
      SLEEP(10);
    }
  }
}
```

La gestion de déplacament de la raquette est similaire à la gestion de déplacement de la balle. Par contre, elle est toujours active (d'où la boucle `while(1)`) avec une attente passive de 10 ms pour tester si le joystick est appuyé (vers la droite ou la gauche). Si c'est le cas, la raquette va être déplacée de STEP pixels tous les 8 ms tant qu'on maintient l'appui. On a rajouté une amélioration qui permet de ne pas dessiner la raquette à nouveau si l'utilisateur essaie de la bouger vers la droite et qu'elle arrive sur le coté droit (`x = LCD_MAX_WIDTH`), et respectivement à gauche si elle arrive sur le coté gauche (`x = 0`).
En ce qui concerne la gestion de collision avec la balle, cette dernière rebondit si elle arrive sur la raquette en haut, à gauche ou à droite.

5 Analyse des traces

6 Limite du nombre de fantômes

Ralentissement du jeu avec un nombre de traces élevées. La fonction de récupération des traces n'est pas assez performante et le buffer prend un certain temps pour se vider. Le jeu reste bloqué lorsqu'il y a plus que 21 tâches en parallèle. Nous pensons que cette limite vient de la mémoire allouée à FreeRTOS. Au-delà de 21 tâches, on dépasse cette mémoire définie par la macro configTOTAL_HEAP_SIZE définie ici à 20 * 1024. En augmentant cette variable nous pouvons augmenter le nombre de fantômes jusqu'à la limite de la mémoire de la carte. La limite que nous avons atteint est de configTOTAL_HEAP_SIZE = 27 * 1024 avec un nombre de tâches de 29 (sans compter VApplicationIdleHook). Au-dessus nous avons un hardfault que nous pensons être du à un dépassement de mémoire de la carte.

7 Structure du programme

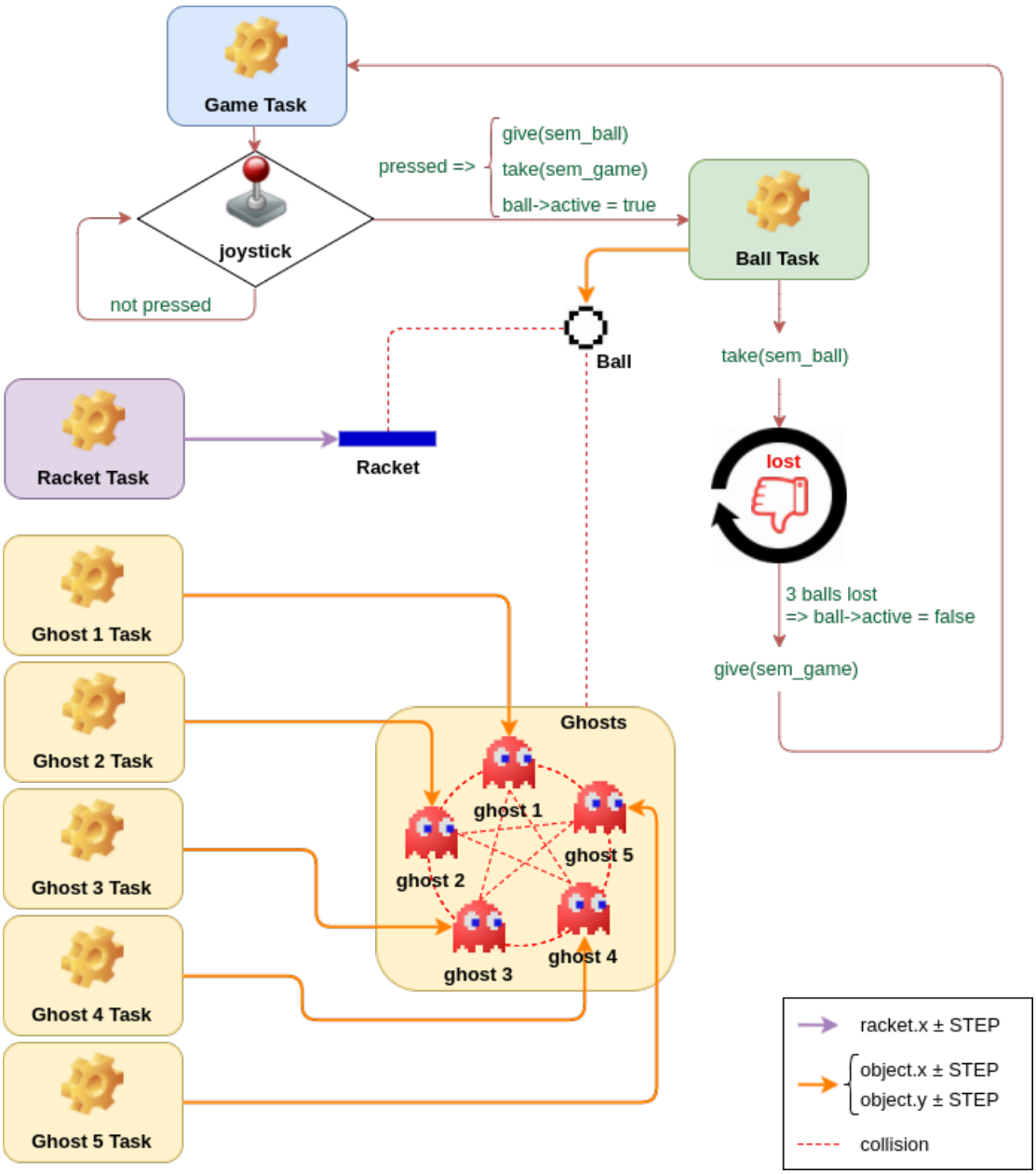

FIGURE 1 – Schéma bloc du programme