TP1 : Ghostbusters

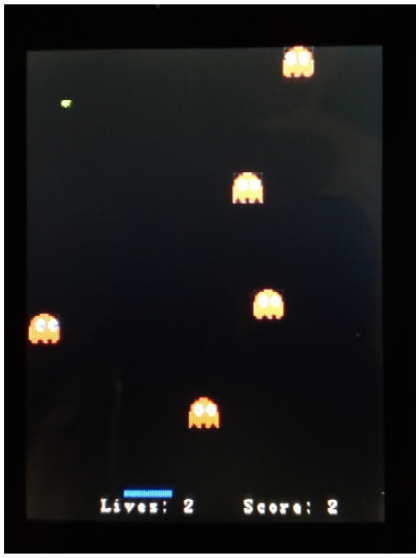
Programmation temps-réel

V. Pilloux

1. Introduction

Le but de ce travail est de réaliser un jeu qui est un mélange entre deux anciens jeux vidéo : le Pacman et le Casse-brique. Il s'agit ici de faire évoluer des « fantômes » sur une partie de l'écran de façon aléatoire et de tenter d'en toucher un maximum avec une balle qui rebondit sur tout ce qu'elle touche. Lorsque la balle touche un fantôme, le score du joueur augmente et le fantôme touché disparaît. Les fantômes rebondissent aussi entre eux lorsqu'ils se touchent (mais sans disparaître). En bas de l'écran, une raquette dirigée par l'utilisateur permet de faire rebondir la balle pour lui éviter de « tomber ». Dans ce dernier cas, celle-ci est perdue. 3 essais, c'est-à-dire 3 pertes de balles sont admises. Au-delà, le jeu s'arrête et affiche un score équivalent au nombre de fantômes touchés pendant la partie.

Le terrain de jeu sera l'écran de votre carte Mylab2 et son apparence devra ressembler à l'image ci-dessous.



2. Objectif pédagogique

Apprendre à utiliser un RTOS dans le cadre de tâches asynchrones et se familiariser avec la gestion du temps en utilisant les primitives du RTOS à cet effet.

3. Préparation

- A l'aide de LPCXpresso, décompressez l'archive ***ghostbuster.zip*** fournie. Cette archive contient un projet qui inclut:
    - FreeRTOS (code source avec adjonction de code pour faciliter la création des traces lors du changement de contexte des tâches)
    - une librairie d'utilitaires (***Mylab_lib.a***) pour la gestion :
        - de l'écran LCD (***lcd.h***)
        - des traces par défaut (***traces_ref.h***)
        - des nombres aléatoires (***custom_rand.h***)
    - un main() affichant une icône et du texte sur l'écran ainsi qu'une fonction de détection de collision utile pour le développement du projet
- **RENOMMEZ TOUT DE SUITE LE PROJET AVEC LE NOM SUIVANT:**

ghostbuster _<*votre N° de groupe*>

4. Spécification du jeu

Le codage du jeu se fera en utilisant FreeRTOS en mode **coopératif**. Les tâches à créer auront toutes la même priorité (mais > `tskIDLE_PRIORITY`), sauf la tâche de gestion de trace qui aura la priorité la plus faible (=`tskIDLE_PRIORITY`). Les tâches seront les suivantes :

- Une tâche par fantôme
- Une tâche pour la gestion de la balle
- Une tâche pour la gestion de la raquette
- Une tâche pour la gestion des traces (à implémenter en dernier, voir §4.3.4)

4.2 Gestion des objets et des collisions

Nous appellerons la balle et les fantômes des « objets ». La raquette sera gérée séparément. Tous les objets se déplaceront au minimum de 2 pixels chaque fois qu'un déplacement est ordonné. Une structure ***object_t*** (fournie dans le code de base) défini les caractéristiques utiles et applicables à tous :

```
typedef struct {
	int x;			// horizontal coordinate of the object center
```

```
        int y;              // vertical coordinate of the object center
        int radius;         /* the radius is the number of pixels to reach the
                               object bound from the center */
        dir_t dir;          // current direction of the object (see dir_t)
        bool active;        // if true, the object must be displayed, otherwise not
} object_t;
```

La structure dir_t défini une direction de déplacement pour un objet. Il est possible que cette direction soit NORTH |EAST par exemple, pour indiquer un déplacement diagonal.

```
// Direction vector. Note that only 8 directions are possible,
// since NORTH|SOUTH is nonsense for example.
typedef enum {
        NORTH=1,
        EAST=2,
        SOUTH=4,
        WEST=8
} dir_t;
```

Le rayon (*radius*) est nécessaire pour la détection de collision. Mais il n'est utile que pour des objets « carrés » ou ronds. La raquette ne rentre pas dans cette catégorie. Une collision a lieu lorsque :

$$|x_2 - x_1| < r_2 + r_1 + m \wedge |y_2 - y_1| < r_2 + r_1 + m$$

x_i et y_i sont les coordonnées horizontales et verticales des objets i et r_i leurs rayons respectifs. m est la marge et dépend de l'unité de déplacement minimum des objets. m=2 dans le cadre de ce projet.

La détection de collision se fait donc sur la base de la mesure de distance entre 2 carrés dont la taille du côté représente « le rayon ».

Afin de pouvoir correctement détecter les collisions, tous les objets susceptibles de se toucher doivent être placés dans un tableau d'objets. Ainsi la déclaration de la balle et des fantômes est faite comme suit:

```
object_t object[GHOST_NB+1];    // +1 for the ball (object[0])
```

La fonction `int test_collision(int object_id, object_t *obj_array, int min_idx, int max_idx)` va regarder si un autre objet se trouve à proximité de l'objet considéré (repéré par l'index object_id), mais **uniquement dans la direction de déplacement** de object_id. Si c'est le cas, une collision est détectée est la fonction retourne l'index d'un des objets proche. Sinon elle retourne NO _COLLISION. La fonction **`test_collision()`** est fournie dans le code original du projet.

Lorsqu'une collision a lieu, l'objet qui la détecte doit changer **l'une** de ses directions pour repartir dans le sens opposé. Exemples :
- un objet qui se déplace vers l'est repartira vers l'ouest.
- un objet qui va vers le nord-est repartira **aléatoirement** soit vers le sud-est, soit vers le nord-ouest.

4.3 Description des tâches

4.3.1 Déplacement des fantômes (1 tâche par fantôme)

Cinq fantômes seront présents en début de partie et se déplacent même avant que le joueur clique sur le joystick pour démarrer le jeu. Leur image de départ est celle du fichier ***ghost_c1.bmp*** et peut être chargée en mémoire avec la fonction ***read_bmp_file()*** (voir ***lcd.h***).

L'espace de l'écran réservé aux fantômes est la partie supérieure, jusqu'à la ligne 270 (l'écran possède une résolution de 240x320 pixels). Les fantômes se déplaceront **orthogonalement** à la vitesse de **2 pixels/(20+*ghost_id**2) ms**, où *ghost_id* est l'index du fantôme considéré. Ainsi le premier fantôme se déplacera à 2 pixels/22 ms et le dernier à 2 pixels/30 ms.

Leur position de départ sur l'écran sera (x, y) = (10+ghost_id*30, 100). Leur direction change aléatoirement, une fois toute les 2000+*ghost_id**200 ms. Ils rebondissent entre eux et sur les bords de l'écran (ou la ligne 270). Les fantômes continuent à évoluer tant qu'ils ne sont pas touchés par la balle. Lorsque c'est le cas, chaque fantôme touché disparaît. Dans la première version de votre développement, ils ne réapparaitront que lorsqu'une balle est perdue. Ensuite, faites réapparaître les fantômes disparus avec une probabilité de 1% toutes les 20 ms, et ce, quel que soit l'état de la partie (même une fois celle-ci terminée).

Facultatif (bonus):

- en fonction de la direction que prend le fantôme, affichez la matrice correspondante :
  - ghost_l***x***.bmp si le fantôme se dirige à gauche
  - ghost_r***x***.bmp si le fantôme se dirige à droite
  - ghost_c***x***.bmp si le fantôme dans les autres cas
  - ***x*** peut valoir 1 ou 2. Il s'agit d'afficher la matrice dont x=1 pendant 100+*ghost_id**10 ms, puis celle dont x=2 pendant le même temps et ainsi de suite. Cela permet de donner une impression de mouvement aux fantômes.

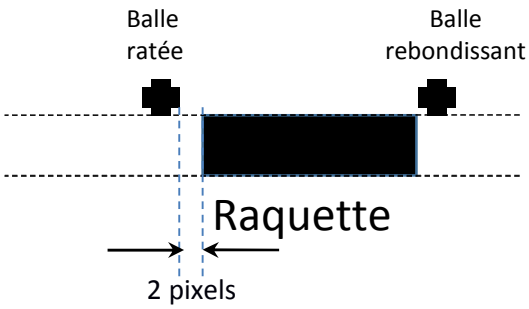
4.3.2 Gestion de la balle (1 tâche)

La position et la direction de la balle est imposée au début de chaque partie :

- Position du centre de la balle : (x, y) = (120, 299). Rayon de la balle : 3 pixels
- Direction de la balle : nord-est

La balle se déplace toujours en diagonale à **2 pixels/10 ms**. Elle rebondit sur tout ce qu'elle touche (voir gestion des collisions), ainsi que sur les bords de l'écran ou la raquette. Lorsqu'un fantôme est touché, celui-ci doit disparaître. Si la raquette manque la balle, le joueur perd une des 3 balles à disposition et son nombre de « vies » est décrémenté. Le programme attendra alors 1 seconde avant de réafficher une nouvelle balle à sa position originale. Lorsqu'il n'y a plus de balles disponibles, on atteint la fin du jeu (voir §4.4 ci-dessous).

On considère que la raquette rate la balle lorsque la balle peut entièrement passer à côté de la raquette verticalement avec 2 pixels de marge comme illustré dans la figure suivante (tenir compte de la taille de la balle).



Cette tâche s'occupe également de calculer le nombre de balles encore à disposition ainsi que le score du joueur. Elle gère aussi la fin du jeu (voir §4.4).

4.3.3 Gestion de la raquette (1 tâche)

Au début du jeu, la raquette est positionnée en (x, y) = (110, 299) (bord haut gauche). La raquette est un rectangle de 30x4 pixels de la couleur de votre choix (autre que noire!). Le déplacement de la raquette s'effectue uniquement horizontalement, en actionnant le joystick : basculer à gauche pour un déplacement à gauche et basculer à droite pour un déplacement à droite. La vitesse de déplacement est de **2 pixels/8 ms**. Bien sûr, la raquette ne doit pas pouvoir dépasser les bords de l'écran.

La raquette doit toujours pouvoir se déplacer, y compris lorsque la partie est perdue.

4.3.4 Traces des timings des tâches

La création des traces a été intégrée au RTOS et il suffit de l'activer en initialisant la constante :

`configHEPIA_TRACING à 1 dans` ***`FreeRTOSConfig.h`***`,`

et en appelant ensuite la fonction du header ***trace_ref.h*** **: init_traces(``115200, 1, true`**`)` avant le démarrage du RTOS. Ainsi, lorsque le séquenceur de FreeRTOS est lancé, il appellera la fonction **write_trace()** chaque fois qu'une tâche prend ou rend la main. **write_trace()** appelle par défaut **write_trace_ref()** qui est une fonction interne à la libraire Mylab_lib.

Dès lors, lorsque FreeRTOS est lancé, tout changement de tâche enverra une trace sur l'UART, que vous pourrez collecter et visualiser avec un PC, grâce aux utilitaires ***uart_to_file, bin2vcd*** et ***GTKWave***, comme d'habitude.

Lorsque vous aurez implémenté le jeu, intégrez finalement votre version de **write_trace()** et placez dans **`vApplicationIdleHook()`** l'envoi des traces sur l'UART. **`vApplicationIdleHook()`** est une

tâche de la plus basse priorité que FreeRTOS déclare lorsque `configUSE_IDLE_HOOK` est à 1 dans ***FreeRTOSConfig.h***. Ôtez aussi l'appel à **init_traces()** donné dans l'archive initiale.

4.4 Début et fin du jeu

Au départ du jeu, les fantômes sont créés et se déplacent selon les règles énoncées au §2.1. La raquette est également mobile (voir §2.3), le score affiche 0 et le nombre de « vies » est aussi à 0. Un texte propose de presser sur le joystick (bouton central) pour démarrer le jeu.

Dès la pression du joystick, le score est remis à 0, le nombre de « vies » à 3, la balle apparaît et le jeu commence. La fin du jeu se produit lorsque le nombre de balles atteint 0. On attend alors une nouvelle pression sur le joystick pour que le jeu recommence.

5. Guide de développement

5.1 Contraintes de développement

- **N'utilisez pas les timers pour gérer le temps, mais uniquement les primitives du RTOS**
- La gigue temporelle des tâches est tolérée, mais **aucune dérive temporelle** ne doit avoir lieu.
- Par souci d'efficacité et de simplicité, l'utilisation de variables globales est **permise**, mais elles doivent être limitées au minimum et commentées.
- Pour les choix aléatoires, utilisez la fonction de la librairie fournie et décrite dans ***custom_rand.h***, car la fonction rand() de la librairie C est très peu équiprobable sous LPCXpresso.
- Le projet doit pouvoir compiler sans erreur, sans warning et exécuter au minimum une partie du TP de manière fiable.

5.2 Démarche de développement conseillée

- Commencez par développer le déplacement d'un fantôme seul (sans FreeRTOS), même chose pour la balle (servez-vous de la structure object_t pour cela).
- Ajoutez la gestion de la collision des fantômes et intégrez-les dans FreeRTOS sous forme de tâches. Vérifiez leurs déplacements et que les les collisions sont correctement gérées.
- Ajoutez les tâches gérant la balle et la raquette.
- Gérez le score et la fin de partie.

- Ajoutez votre version des traces et vérifiez que vos tâches ne génèrent pas de dérive temporelle.
- Ajoutez optionnellement l'animation des fantômes (bonus).
- Ecrivez le mini-rapport.

6 Travail à rendre

- Une archive LPCXPresso avec le nom suivant : **ghostbuster_<***votre N° de groupe***>.zip** contenant votre projet complet. Créez cette archive avec LPCXpresso : *File->export->archive file.*
- Le mini rapport dont le contenu est décrit ci-dessous, au format pdf.
- Si vous n'arrivez pas à faire fonctionner l'ensemble du projet, envoyez les « briques » de code que vous avez construites séparément selon la démarche de développement proposée au §5.2 et indiquez dans le rapport ce qui fonctionne et ce qui ne fonctionne pas.

6.1 Format et contenu du mini-rapport

Ce rapport a pour objet de décrire très brièvement l'état du/des projet(s) rendu(s), de mentionner les anomalies observées (s'il y en a) et de les commenter. **Le rapport ne doit pas dépasser 4 pages A4.**

- Titre : nom du TP, mention du cours, numéro du groupe, noms des participants et date.
- Expliquez brièvement, selon vous, l'intérêt à utiliser un RTOS dans le cadre de **ce projet**.
- Indiquez l'état général de votre développement au moment où le code a été rendu : état général du programme et de chacune des tâches. Si le code n'est pas terminé, expliquez pourquoi et quelle aurait été votre démarche pour pouvoir le finir.
- Expliquez les anomalies que vous avez observées (sur le code rendu uniquement) et donnez une piste pour les expliquer.
- Montrez un extrait des traces des tâches que vous obtenez et expliquez-le. Entre autres, démontrez que vous n'avez pas de dérive temporelle, sinon expliquez pourquoi.
- Si vous pensez qu'un point particulier sera difficile à comprendre en lisant simplement le code et ses commentaires, détaillez-le dans le rapport, si possible avec une figure à l'appui.
- Facultatif (bonus): cherchez quelle est la limite maximum expérimentale du nombre de fantômes affichables et expliquez pourquoi, traces à l'appui.
- **Inutile de répéter les éléments décrits dans cette présente spécification !** Au besoin, donnez simplement une référence aux chapitres concernés.