Implémentation d'un scrolling et importation d'image bitmap, Détection de collision

Objectifs du TP

A l'issue de ce TP vous devez pouvoir montrer un paysage qui défile dans la partie basse de l'écran. Une image est fournie à cet effet mais vous pourrez en utiliser une autre dans votre projet avec un utilitaire d'importation d'image. On conservera le travail du dernier TP : le vaisseau spatial dessiné en assembleur continuera à se déplacer à l'aide des touches de direction, dans une zone d'évolution au dessus du paysage. Enfin le programme d'affichage du vaisseau sera modifié pour retourner une valeur non nulle si l'un des pixels est déjà occupé (non noir) afin de détecter les collisions avec d'autres objets graphiques dessinés dans la même zone.

Notes sur le programme final

Le programme final obtenu à l'issue du prochain TP (TP 4) ajoutera des ennemis sous la formes de soucoupes volantes (UFOs) qui se déplaceront aléatoirement dans la même zone d'évolution que le vaisseau. Le but du jeu sera simplement d'éviter de les toucher, le plus longtemps possible.

L'objectif de cette série de séances n'est pas de créer un jeu d'arcade complet, d'une part il ne s'agit pas d'un rendu de projet pour lequel vous disposez du temps nécessaire, d'autre part l'enjeu est de s'intéresser aux détails d'implémentation et de fonctionnement du programme (mémoire, ressources processeur...). Le résultat sera donc modeste par rapport à ce que vous savez faire avec une librairie plus évoluée comme Allegro : il n'y aura a priori pas de gestions d'armes (tirs, explosions d'ennemis...), pas de score chiffré, pas de menu, pas de travail important sur des graphismes.
Vous pouvez essayer d'apporter ces améliorations si vous le souhaitez et que vous en avez le temps mais ce n'est pas demandé.
Par contre nous essayerons lors du dernier TP d'évaluer précisément les performances du code et de l'optimiser au niveau assembleur.

Copie de projet

Commencez par copier le projet du TP2 dans un nouveau répertoire de travail : copiez simplement l'ensemble du répertoire de départ et donnez lui un nouveau nom. Vous pouvez aussi modifier les noms des fichiers de projet VHAM avec les extensions .vho et .vhw (dans le répertoire).
Puis lancez VHAM et ouvrez ce projet (menu -> File -> Open Workspace). Vérifiez que l'ensemble recompile correctement (F10 puis F5).

Mise en place du scrolling

Un scrolling est un effet de défilement obtenu en répétant une translation du contenu graphique à intervalles réguliers, en général une fois par itération dans la boucle principale d'animation. Ici le but est d'obtenir à chaque itération un déplacement de 2 pixels vers la gauche dans les 40 lignes en bas de l'écran. Il faudra ajouter au projet et appeler la procédure `asmRectcpy` vue au TD3 : en copiant le rectangle source de coordonnées (2,120) vers le rectangle destination de coordonnées (0,120) avec w=238 et h=40 .

- **Ajout du sous programme assembleur `asmRectcpy` au projet**

  Dans le fichier source assembleur `asm.s` ajouter à la fin (en dehors d'un autre sous-programme) le bloc de déclaration "technique"

```
.ARM
.SECTION .iwram,"ax",%progbits
.ALIGN
.GLOBAL  asmRectcpy
.TYPE    asmRectcpy, function
```

  Puis le sous programme proprement dit :

```
asmRectcpy:
    push   {r4,r5}

    rcBoucleV:          @ Boucle verticale : r3 servira de compteur
        mov     r4,r2            @ Init boucle horizontale : r4 sert de compteur
        rcBoucleH:
              ldrh    r5,[r1]            @ transfert depuis adresse source r1
              strh    r5,[r0]            @ transfert vers adresse destination r0
              add     r1,r1,#2           @ src : pixel horizontal suivant (+2 octets)
              add     r0,r0,#2           @ dest : pixel horizontal suivant (+2 octets)
              subs    r4,r4,#1           @ décrémenter compteur boucle horizontale
              bne     rcBoucleH          @ boucler pixel horizontal suivant
        sub     r1,r1,r2,lsl #1   @ source : Retour en début de segment horizontal
        add     r1,r1,#SCREENLINE @          Puis ligne en dessous (SCREENLINE=480 voir incarmgba.s)
        sub     r0,r0,r2,lsl #1   @ dest. : Retour en début de segment horizontal
        add     r0,r0,#SCREENLINE @          Puis ligne en dessous
        subs    r3,r3,#1          @ décrémenter compteur boucle verticale
        bne     rcBoucleV         @ boucler ligne suivante

    pop    {r4,r5}
    bx     lr
```

IMPORTANT : lorsque vous ajoutez un nouveau sous-programme assembleur au projet il est possible de copier/coller le bloc de déclaration (.ARM .SECTION ...) depuis un autre sous-programme mais **il faut adapter le nom dans le .GLOBAL et le .TYPE**, en plus de préciser ce même nom comme étiquette de début (en rouge dans le code ci-dessus)

- Ajout du prototype dans **`projet.h`** : ici le prototype à ajouter est **`void asmRectcpy(u16 * dest, u16 * src, int w, int h);`**
- Pour tester : dessiner un ou plusieurs carrés de couleur à cheval sur la zone à déplacer, avant la boucle d'animation.
  Par exemple **`asmDrawBlock(100, 100, 40, 0x03E0);`**
- Appeler la fonction de copie, toujours avant la boucle d'animation, avec les paramètres correspondant au déplacement attendu :
  `asmRectcpy((u16 *)(VRAM+2*0+480*120), (u16 *)(VRAM+2*2+480*120), 238, 40);`
  Note : VRAM est une constante (#define) qui vaut 0x06000000 (voir armgba.h)
- Lancez le programme : en principe on doit observer un décalage dans la partie basse du carré dessiné, au niveau de la zone déplacée
- Basculer l'appel d'asmRectcpy à l'intérieur de la boucle d'animation (après l'affichage du vaisseau)
  Relancez le programme, qu'observez vous ?

Facultatif :
Pourquoi est-il nécessaire d'encadrer le calcul des adresses source et destination par des parenthèses avant de "caster" en pointeur sur u16 ?
Que se passe-t-il si on écrit **`(u16 *)VRAM+2*2+480*120`** *? Comment modifier cette formule sans parenthèse pour qu'elle devienne correcte ?*

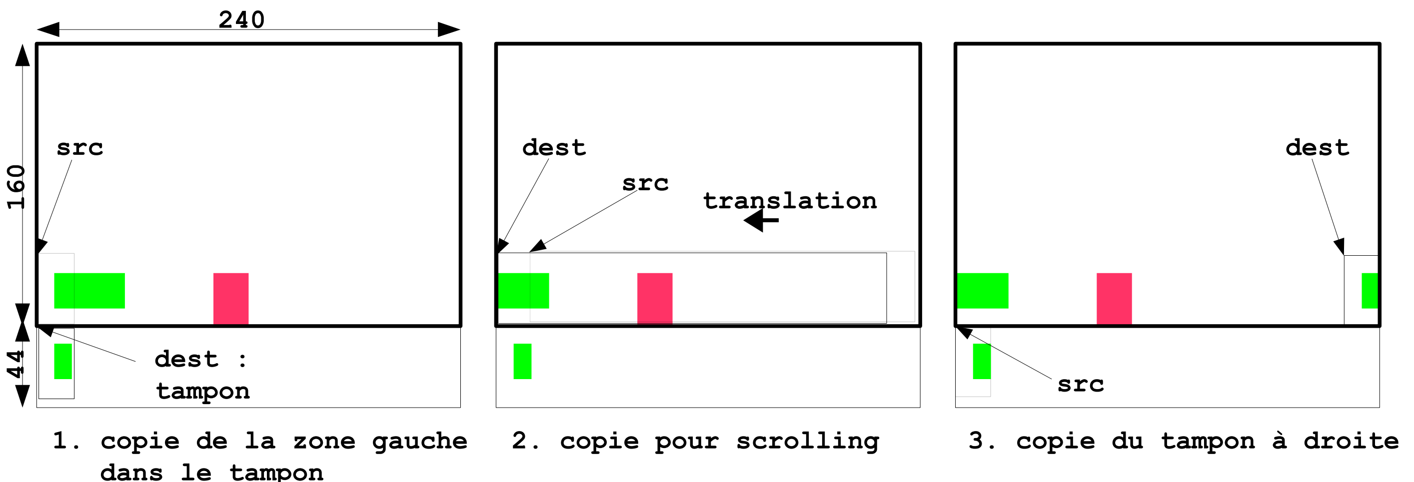
Amélioration pour la lisibilité : comme beaucoup de nos sous-programmes vont demander des adresses de pixels plutôt que des coordonnées, nous avons intérêt à ajouter au projet une fonction C qui prend en paramètre les coordonnées et retourne l'adresse :

```
u16 * pixelAddress(int x,int y){
   return (u16 *)(VRAM+2*x+480*y);
}
```

L'appel précédent à **`asmRectcpy`** devient : **`asmRectcpy(pixelAddress(0,120), pixelAddress(2,120), 238, 40);`**

Scrolling circulaire

L'effet de scrolling précédent fait sortir les graphismes à gauche de l'écran. Pour continuer à faire défiler des graphismes il faudrait redessiner de nouvelles images à droite (apparition de graphisme au niveau du bord droit). Pour ce projet on va se contenter de faire tourner le graphisme en boucle : ce qui disparaît à gauche réapparaît à droite. Comme la procédure de copie écrase le contenu précédent au bord gauche, il va d'abord falloir le sauver dans un tampon, puis faire la copie de scrolling vers la gauche, et enfin recopier le tampon sur le bord droit.
Sur GBA le mode graphique 3 utilise 2 octets par pixels pour 240x160 pixels, soit un total de 76800 octets. La mémoire vidéo disponible est de 96ko (voir cours, découpage de l'espace mémoire en plages) donc 96*1024=98304 octets. Il reste 21504 octets qui sont "en dehors" de l'écran, mais qu'on peut exploiter pour stocker des données graphiques temporaires comme le tampon du scrolling circulaire; il y a 44 lignes disponibles "en dessous" de la dernière ligne en bas de l'écran (21504/480=44,8)



Déterminez les coordonnées nécessaires et remplacez le seul appel à `asmRectcpy` pour le scrolling simple par ces 3 appels successifs pour réaliser le scrolling circulaire. La zone de scrolling correspond aux 40 dernières lignes visibles de l'écran, ce sont donc les ordonnées 120 à 159.

Importation d'une image bitmap

Sur le site, en dessous de cet énoncé, vous trouverez un fichier **`TP3.zip`** qui contient un programme source Allegro **`img2tab.c`** qui convertit des fichiers images en programmes sources prêts à être ajoutés au projet. Le principe est d'écrire les données images sous forme d'une déclaration de tableau C initialisé. Vous pouvez utiliser cet utilitaire pour importer vos propres images. Attention **`img2tab.c`** ne se compile pas dans l'environnement VHAM, si vous voulez le compiler (facultatif) ouvrez un projet Allegro par exemple avec CodeBlocks !

Pour l'instant nous allons partir d'un fichier déjà converti, également dans **`TP3.zip`** : **`panorama.c`**
L'image d'origine (panorama.bmp) mesure 240 pixels de large et 40 pixels de haut : exactement la taille de la zone de scrolling circulaire.
Le but est d'afficher cette image au niveau de cette zone.

- Copier **`panorama.c`** dans le répertoire de projet
- Dans VHAM, ajouter ce fichier source au projet (panneau de gauche) en faisant click droit sur **`Source Files`** puis **`Add file(s)`**
- Indiquer au compilateur ce nouveau fichier source en éditant le fichier **`makefile`** (panneau de gauche) :
  Ajouter un fichier objet en plus à la 9ème ligne : OBJS := main.o armgba.o panorama.o
  Notez que pour compiler un fichier source **`toto.c`** on spécifie l'objectif de compilation (target) donc un fichier objet **`toto.o`**

- La variable **panorama** initialisée dans **panorama.c** est de type **const t_image** , il faudra ajouter dans **projet.h** :
  - La définition de ce nouveau type de structure (si ce n'est pas déjà fait)

```
typedef struct{
    int w,h;
    u16 img[];
} t_image;
```

  - ajouter aussi dans **projet.h** la déclaration de cette variable pour qu'elle soit visible depuis les autres fichiers sources (mot clé **extern**)

```
extern const t_image panorama;
```

  - A noter que la déclaration en tant que variable non modifiable (const) place automatiquement les données d'initialisation de la variable dans la RAM externe EWRAM qui est plus spacieuse (256ko) que la RAM interne IWRAM (32ko) destinée aux variables usuelles.

- Ajouter au projet cette procédure **drawImage** qui prend les données couleurs dans la structure et les affiche à l'écran aux coordonnées x,y :

```
void drawImage(const t_image * bmp, int x, int y){
   int xi,yi;
   for (yi=0;yi<bmp->h;yi++)
      for (xi=0;xi<bmp->w;xi++)
         *pixelAddress(x+xi,y+yi) = bmp->img[xi+bmp->w*yi];
}
```

- Appeler cette procédure pour afficher **panorama** en x=0 y=120 (avant d'entrer dans la boucle d'animation)

- Le panorama doit maintenant défiler de manière circulaire en bas de l'écran

Détection de collisions

Renommez proprement le sous programme assembleur de dessin du vaisseau en **asmDrawShip**
Spécifiez le prototype de ce sous programme en **int asmDrawShip(int x,int y);**
Le but est de modifier le sous-programme pour retourner une valeur non nulle si l'un des pixels est déjà occupé (non noir) afin de détecter les collisions avec d'autres objets graphiques dessinés dans la zone d'évolution du vaisseau.

Modifier le code assembleur pour vérifier avant chaque écriture de pixel (**strh**) si la couleur du pixel en cours est non nulle. Rendre un registre non nul dans ce cas. Une possibilité est d'utiliser l'opérateur logique OU (orr en assembleur)

```
asmDrawShip:
   push  {r4,r5}

   @ au début du sous programme : initialiser r5 à 0
   mov   r5,#0

   ...

   @ à chaque pixel à écrire :
   ldrh  r4,[r2]   @ On commence par rapatrier la valeur actuelle du pixel dans r4
   orr   r5,r5,r4  @ r5 <- r5 OU r4    (si r4!=0 alors r5 devient non nul)
   strh  r3,[r2]   @ On colorie le pixel de toute façon (on traitera la collision plus tard..)

   ...

   @ fin du sous programme : la valeur de retour doit être dans r0
   mov   r0,r5

   pop  {r4,r5}
   bx   lr
```

Si ce travail vous semble trop difficile en assembleur, essayez d'écrire une fonction en C qui réalise les mêmes opérations : vérifier couleur actuelle du pixel à afficher, si non nulle marquer une variable **collision**, afficher couleur vaisseau sur pixel, ceci pour chaque pixel du vaisseau. Retourner valeur de la variable **collision**.

Pour tester le fonctionnement de la détection de collision il faut mettre un obstacle sur l'écran, par exemple un carré (en attendant un UFO).
Si le vaisseau est effacé de son ancienne position par un carré (asmDrawBlock) cette zone plus grosse que le vaisseau risque d'effacer l'obstacle à mesure que le vaisseau s'en approche : il faut alors réafficher l'obstacle à chaque itération :

```
boucle infinie :
   appel vSync
   effacer ancienne position vaisseau
   interaction vaisseau (changer position)
   afficher obstacle
   appeler afficher nouvelle position vaisseau et récupérer valeur en retour
   si valeur retour non nulle indiquer collision (par exemple dessiner un gros carré rouge)
   séquence de copies qui réalise le scrolling circulaire
fin de boucle infinie
```