# Rapport: Projet GBA

**Table des matières**

Les sigles:

(?): Informations (éléments à savoir, généralités)
(!): Explication (travail ou raisonnement effectué)
(>): Sous-chapitre.

# 01) Introduction

Cette partie du rapport relate du programme principal pour la Game Boy Advance. Il se présente sous la forme d'une mini-documentation sur la Game Boy Advance décrivant les points utilisés dans notre projet.
Ceci peut également servir de guide dans la programmation sur cette console puisqu'il couvre tous les éléments principaux nécessaires pour faire une utilisation optimale du matériel mis à disposition, en supposant que vous connaissiez bien le C ou que avez des bases dans la programmation de ce genre de systèmes. Si ce n'est pas le cas, lisez déjà le document de cours inclus au projet.

# 02) Tour d'horizon: les modes graphiques

Je pense qu'avant de commencer, il convient de faire un petit aperçu de ce que propose le matériel graphique de la Game Boy Advance (que l'on abrégera GBA pour simplifier un peu) :

Il y a six modes graphiques... Passons-les rapidement en revue.
- Les modes 0 à 2 sont des modes dits *de mosaïques*, qui disposent de 64 ko de mémoire vidéo pour les mosaïques, et 32 ko pour les sprites. Chaque BG peut utiliser la transparence, et la rotation uniquement si c'est proposé par le mode (modes 1/2).
- Les modes 3 à 5 sont des modes dits *bitmap*, qui disposent de 80 ko pour l'écran et 16 ko pour les sprites. Ces modes n'offrent pas de capacités avancées.

Si vous ne connaissez pas les sprites, ne vous en souciez pas, on les verra plus tard. Pour les mosaïques, regardez juste en-dessous.

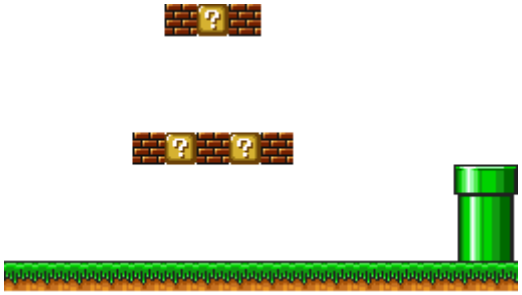
# 03) Les modes de mosaïques

## (?) Principe

Les modes dit « de mosaïques » utilisent le principe des tilemaps. Ce qui signifie qu'on dispose de petits motifs qu'on dessine soi-même (ici ils font 8x8 pixels) et qu'on assemble pour créer une image. La map, c'est en fait un tableau qui indique pour chaque bloc le numéro de motif à utiliser.
Dans la mesure où les modes de mosaïques sont directement implémentées dans le matériel de la console, il est dès lors avantageux de savoir les utiliser au mieux, car vous n'obtiendrez jamais une vitesse comparable en rendu logiciel.

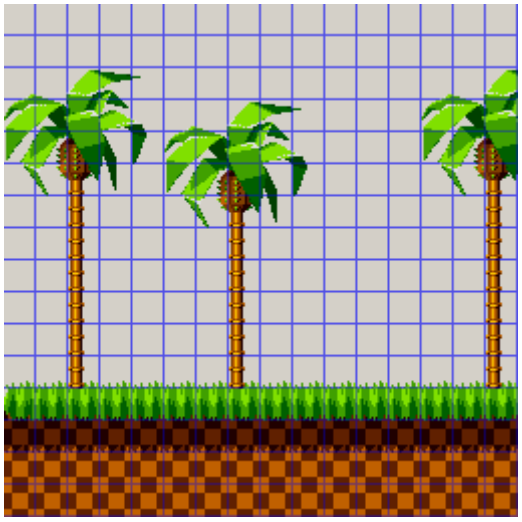
A présent, si je vous montre ceci:

Que pouvez-vous noter, en comparaison de ce qui se trouve dans l'image ci-dessous?

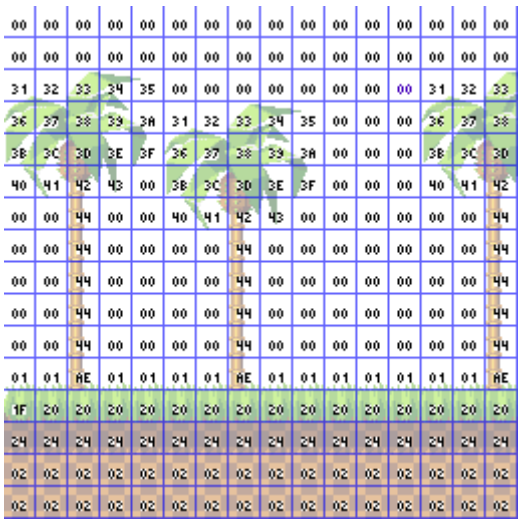


La première chose qui frappe, c'est que chacune des parties de l'image de dessous se retrouve dans celle de dessus. Un peu comme un puzzle dont on utiliserait plusieurs fois les mêmes pièces, on a assemblé les motifs de dessus (peu nombreux) pour créer un tableau bien plus grand.
La première (petite) image est appelée "jeu de mosaïques" (tileset), et la deuxième est appelée "carte" (map).

Maintenant, voyez cette map à laquelle j'ai ajouté une grille. Elle est décomposée en petits motifs. Notez par exemple que le tronc de l'arbre est utilisé de multiples fois, mais il n'existe en fait qu'à un seul exemplaire :



Et la map contient les valeurs correspondantes aux motifs (dans la photo ci-dessous, elles sont en hexadécimal). Par exemple, le motif du tronc est le 44, et vous le remarquerez à plusieurs endroits :



Ceci est très avantageux car on peut dès lors utiliser plusieurs fois les mêmes motifs (dessinés une seule fois), ceci offrant un gain de temps et de place non négligeable.

### (!) Choix du mode graphique

L'intérêt de la console étant le mode de mosaïques (et ce n'est pas pour rien!), nous allons l'utiliser car il est plus adapté à nos besoins.
Voici par exemple la place demandée par la map de l'EMVS que nous avons utilisée pour une bitmap ou pour une tilemap :
Bitmap : *388000 octets*
Tilemap : 52672 + 12200 = *64872 octets*
Et pourtant la carte de l'EMVs n'a pas du tout été optimisée pour le tilemap!
Lorsqu'on effectue des niveaux de jeu, en général, on utilise de très grosses maps, et dès lors les rapports de tailles peuvent dépasser les 10'000% !

Bien sûr, il faut encore savoir que programmer en mode bitmap n'a aucun intérêt. Autant utiliser un PC, une GP32 ou un Palm, dont les microprocesseurs sont bien plus puissants! La force de la GBA est bien son mode de mosaïques extrêmement performant.
Après, pour notre projet, le choix entre les modes 0 à 2 est assez vite fait : comme nous n'allons pas utiliser la rotation des arrière-plans, on prendra le mode 0.

### (>) Fonctionnement sur GBA

Tout d'abord, il est nécessaire de voir un peu de vocabulaire.
- Tile: Mosaïque. Elles font toujours 8x8, mais rien n'empêche de les assembler pour créer des motifs plus grands. Un tileset (jeu de mosaïques) désigne une liste de mosaïques à utiliser.
- Map: Carte. Sur la console, chacun des éléments du tableau est codé sur 16 bits, dont 10 sont utilisés pour coder le numéro de *tile* à utiliser.
- CharBlock: Ce sont des emplacements en mémoire vidéo pouvant contenir des mosaïques. Les *tiles* doivent êtres entassées dans les divers charblocks avant de pouvoir les utiliser. Un charblock correspond à un *tileset* (jeu de mosaïques) de taille définie (16 ko).
- ScreenBlock: Ce sont des emplacements en mémoire vidéo prévus pour accueillir des *maps* de taille définie (2 ko).
- Palette: Comme sur PC, les palettes sont de simples tableaux qui contiennent toutes les couleurs que l'on va utiliser. Au lieu d'indiquer directement qu'un pixel est (par exemple) rouge, vous diriez qu'il est (par exemple) de couleur n° 4, et vous mettriez du rouge au quatrième emplacement de la palette.
- BG: Les BGs (arrière-plans) ne sont qu'une application des tilemaps. On peut considérer un BG comme un écran entier. Le fait d'en utiliser plusieurs permet d'afficher plusieurs couches sur l'écran et donner un effet de profondeur. Un BG n'est que le résultat d'une utilisation simultanée d'une *map* et d'un *tileset*.

Sur notre console, la mémoire vidéo (VRAM) de 64 kilooctets est divisée en 4 Charblocks de 16 ko chacun. Cela permet de répertorier ses mosaïques au mieux. Comme on dispose de 4 BGs, il semblerait logique d'associer un Charblock par BG, mais rien ne vous oblige à le faire.
Ce même espace de VRAM (64 ko) est également divisé en 32 Screenblocks, de 2 ko chacun.
Mais chacun des BGs doit avoir ses propre paramètres. Cela se fait par l'intermédiaire du registre REG_BGxCNT (où x peut valoir de 0 à 3 selon le BG à paramétrer).

REG_BGxCNT BG x control

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | Taille | | Wrap | Screenblock | | | | | Palette | Mosaic | x | x | Charblock | | Priorité | |

0-1; Priorité: Détermine la priorité du plan par rapport aux autres. Le plan qui a la priorité 0 est celui qui passera par-dessus les autres, alors que celui qui a la priorité 3 passera derrière tous.
**Note:** Les sprites de même priorité que les BGs passent toujours par-dessus ceux-ci.
2-3; Charblock: Détermine quel charblock le BG utilise. Vous avez le choix entre 0 et 3. N'oubliez pas que rien ne vous empêche d'utiliser le même charblock pour plusieurs plans.
6; Mosaïque: Active ou non l'effet de mosaïque. Si vous l'activez, vous devrez paramétrer le registre correspondant (REG_MOSAIC). Cet effet divise le contenu de l'écran en gros carrés. Agréablement couplé au "fade" lors des effets de transition:

7; Mode palette: Sélectionne le mode de palette à utiliser. Si le bit vaut 1, les pixels sont codés sur 8 bits (256 couleurs) et les 4 bits "palette" des mosaïques sont ignorés. Si le bit vaut zéro, la palette (256 couleurs) est divisée en 16 palettes de 16 couleurs seulement. Les pixels sont alors codés sur 4 bits (16 valeurs) et il faut sélectionner pour chaque mosaïque quelle palette utiliser (parmi les 16). Voir plus bas dans la définition des mosaïques.
8-C; Screenblock: Détermine le ScreenBlock dans lequel se trouve la map. Par convention, on stocke les tiles dans les premiers charblocks (au "début" de la mémoire vidéo), et les maps dans les derniers screenblocks (à la "fin" de la mémoire vidéo) afin d'éviter qu'ils ne se marchent dessus. Généralement, un plan sera configuré avec le Charblock 0 ainsi que le ScreenBlock 31 (le dernier).
D; Wrap: En mode rotation, détermine si le BG est répété indéfiniment (créant une mosaïque). Dans le cas contraire, les pixels hors du BG seront dessinés comme transparents.
E-F; Taille: Paramètre la taille du BG. La taille réelle dépend du mode en question.

En mode *texte* (16 bits par *tile*):
- 00 : 256x256 (32x32 tiles, 1 screenblock)
- 01 : 512x256 (64x32 tiles, 2 screenblocks)
- 10 : 256x512 (32x64 tiles, 2 screenblocks)
- 11 : 512x512 (64x64 tiles, 4 screenblocks)

En mode *rotation* (8 bits par *tile*):
- 00 : 128x128 (16x16 tiles, 1 screenblock)
- 01 : 256x256 (32x32 tiles, 1 screenblock)
- 10 : 512x512 (64x64 tiles, 2 screenblocks)
- 11 : 1024x1024 (128x128 tiles, 8 screenblocks, soit l'équivalent d'un charblock entier)

En mode texte, le format des mosaïques est le suivant:

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | N° de palette | | | | MV | MH | Numéro de mosaïque (dans le tileset) | | | | | | | | | |

0-9; Numéro de tile: Ceci indique la mosaïque à utiliser. Comme ce nombre est codé sur 10 bits (soit 1024 possibilités) et qu'on peut mettre jusqu'à 2048 mosaïques (16 couleurs) en VRAM, on ne peut accéder à toutes les mosaïques sans changer de charblock (représentant 512 tiles). Le numéro de tile véritablement affiché est donc finalement le suivant: *tile*+512**charblock*.
A; Miroir H: Retourne horizontalement (de droite à gauche) la mosaïque.
B; Miroir V: Retourne verticalement (de bas en haut) la mosaïque.
C-F; Palette: En mode 16 couleurs, définit la palette à utiliser parmi les 16. Ces bits peuvent par exemple être utilisés pour mettre une option de menu en surbrillance. On ne change pas de tileset mais on la distingue des autres puisque les couleurs réellement désignées ne sont plus les mêmes.

Pour information générale, la GBA ne dispose pas de bit de profondeur (permettant d'indiquer si la mosaïque doit passer devant ou derrière les personnages), ce qui est parfois gênant. Cependant, comme elle dispose de 4 plans, il suffit d'en utiliser deux définis avec une priorité différente puis d'écrire les mosaïques soit sur un plan soit sur un autre selon qu'on désire qu'elle passe devant ou derrière le personnage. Cette méthode demande toutefois plus de ressources, mais donne plus de possibilités.

En mode rotation, le format des tiles est très simple; chaque tile est codée sur 8 bits, aucun effet spécial n'est applicable. Aussi, on est toujours en mode 256 couleurs puisqu'il n'y a pas la place pour insérer le numéro de palette.

## (>) Documents supplémentaires

Personnellement, je me suis servi de la documentation GBATek, que vous pouvez trouver ici:
http://www.work.de/nocash/gbatek.htm
Mais j'ai découvert un peu avant la fin du projet la documentation suivante, qui me semble plus adaptée pour débuter, car elle contient plus d'explications et utilise un langage moins technique. Evidemment, ces deux documentations sont en anglais, mais elles ne devraient pas être trop difficiles à comprendre.
http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm

## (>) Exemple

Avant de continuer, il me semble extrêmement important que vous compreniez au moins les bases de la programmation sur cette console, car le projet est tout de même construit sur ces fondations. Vous pouvez trouver ce projet d'exemple dans le dossier du projet */Exemples/Mosaique/*. Il serait bien que vous ayez ce projet sous les yeux avant de continuer.

Mais avant de commencer à lire ces explications, je pense que vous devriez lire le petit cours GBA fourni au projet *(/Documents/Cours GBA.doc*). Il explique les bases (comme l'utilisation de HAM) ainsi que la création de votre premier programme. N'hésitez pas non plus à modifier dans le code certaines choses que vous comprenez difficilement. Cela vous permettra de mieux les comprendre.

De plus, je vous conseille vivement de remplacer la version SDL de VisualBoyAdvance (fournie dans HAM) par la version conventionnelle. Elle inclut plus de fonctionnalités, dont des outils accessibles depuis le menu "Tools…" qui vous permettront de mieux visualiser ce qui se passe.
Pour cela, remplacez le fichier /%HAMDIR%/TOOLS/WIN32/VBA.EXE (où %HAMDIR% est le répertoire d'installation de HAM, pas défini en tant que variable d'environnement MS-DOS) par */tools/VisualBoyAdvance/VisualBoyAdvance.exe* (dossier du projet) et n'oubliez pas de le renommer en vba.exe!

Ouvrez le programme GBA Graphics, que vous trouverez dans le dossier */Exemples/Mosaique/Res/* du projet. A partir de ce programme, familiarisez-vous avec les types de fichiers utilisés (un jeu de mosaïques, une palette et une map). Au mieux, lisez la documentation associée (depuis l'autorun du CD -> Outils… -> Gba Graphics -> Option "Lire la doc").

Sélection du mode vidéo

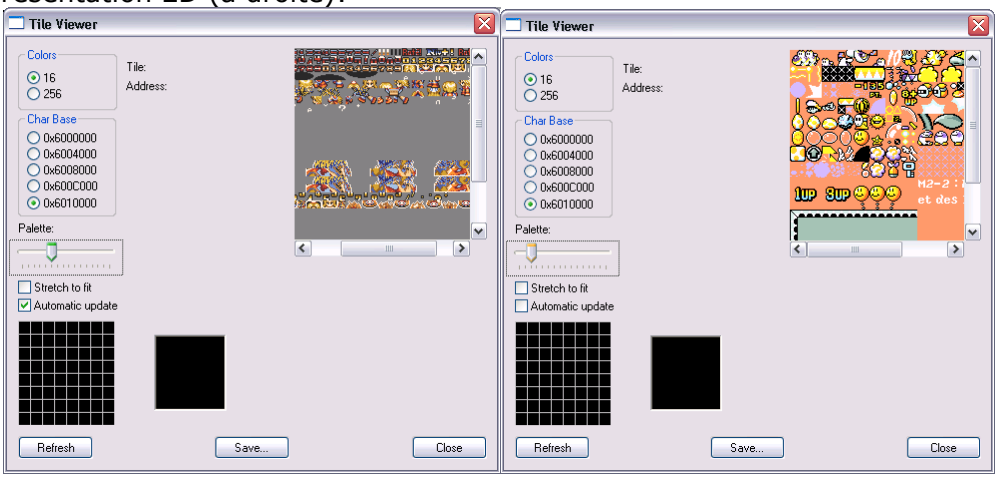
Pour cette petite démo, j'ai utilisé le mode 0, qui est le mode de mosaïques standard. J'ai donc paramétré REG_DISPCNT pour qu'il utilise le mode 0 et qu'il active le BG0. Voici la définition de REG_DISPCNT:

REG_DISPCNT Display Control

| **Bit** | F | E | D | C | B | A | 9 | 8 |
|---|---|---|---|---|---|---|---|---|
| **Effet** | Obj win | Win 1 | Win 0 | OAM | BG3 | BG2 | BG1 | BG0 |

| **Bit** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **Effet** | Blank | 1D/2D | H-libre | Buffer | GBC | Mode graphique | | |

0-2; Mode: Choisit le mode graphique. Vous trouverez une description des modes dans fichier materiel.h du projet Map.
3; GBC: Active le mode Game Boy Color. Ce bit est réglé par le BIOS au démarrage et n'est pas modifiable.
4; Buffer: Dans un mode permettant le double buffering (4 ou 5), détermine l'adresse du buffer affiché. S'il est nul, ce sera *06000000*, sinon *0600A000*.
5; HBlank interval libre: Permet de laisser au contrôleur graphique le temps de la HBLANK entre chaque ligne pour traiter les objets à afficher sur la prochaine ligne. Cela peut éviter les clignotements lorsqu'il y a trop de sprites sur une ligne.
6; Mode sprites: Change le mode de représentation des mosaïques des objets. Si le bit vaut 1, ce sera le mode 1D, sinon le mode 2D. En mode 1D, les mosaïques des objets sont stockées séquentiellement, et dans le cas contraire, elles sont placées comme dans un tableau 2D dont chaque ligne est composée de 64 mosaïques. Voici un exemple de jeu[*1] utilisant la représentation 1D (à gauche) et un autre[*2] utilisant la représentation 2D (à droite):




Le moins qu'on puisse dire est que le mode 2D (celui de droite) est plus facile à se représenter, mais plus compliqué à utiliser que le mode 1D, où toutes les tiles sont contiguës. De plus, en mode 2D, il peut facilement y avoir du gaspillage (comme si vous essayez de placer les formes de manière à pouvoir utiliser toute la pâte en une fois pour vos biscuits de noël).
7; Blank forcé: Eteint l'écran ainsi que le contrôleur graphique. Permet d'économiser de l'énergie lorsque l'affichage n'est pas requis. De plus, comme les accès à la VRAM

sont ralentis (d'un cycle) lorsque la GBA y accède en même temps, les taux de transfert seront légèrement plus élevés si vous activez ce bit.
8-B; BGx: Active les différents BGs disponibles. N'activez pas les BGs dont vous ne vous servez pas, sinon vous verrez des déchets à l'écran.
C; OAM: Active les sprites (objets).
D-F; Fenêtres: Je n'ai pas eu le temps d'expérimenter ces fonctionnalités.

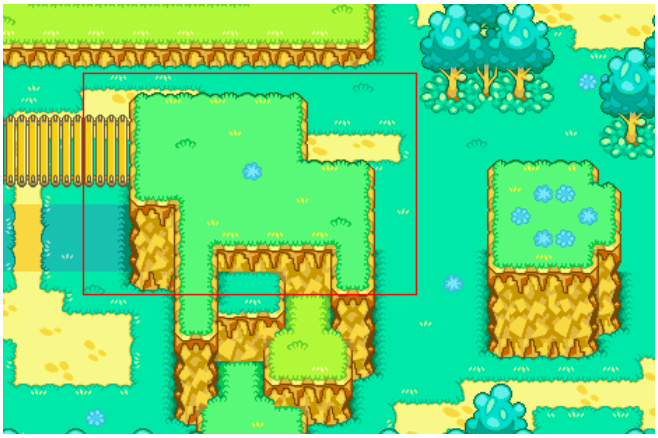
Copie des éléments graphiques

Avant de pouvoir les utiliser, il est obligatoire de copier tous ces éléments en VRAM car le contrôleur graphique ne peut pas accéder à la mémoire standard. Chacun de ces éléments est dépendant d'un autre (la map est dépendante du tileset qu'elle utilise. Le tileset, lui, est dépendant de sa palette pour permettre de décoder les couleurs).
Dans les jeux, il est courant d'intervertir ces éléments entre eux (d'utiliser une autre palette pour le même tileset, ou d'utiliser un autre tileset pour la même map). Cela permet de mettre un peu d'animation sans avoir à tout redessiner.

Ces copies sont réalisées avec l'aide du contrôleur DMA, qu'on abordera dans un chapitre plus loin dans ce document. Pour l'instant, ne vous en souciez pas, et sachez juste que les fonctions CopieObjet et Copie effectuent des copies d'une partie de la mémoire vers une autre. La différence est que CopieObjet copie un élément entier, mais que Copie n'en copie qu'une partie (spécifiée par la taille).

Copie de la map

Dans cet exemple, j'ai tenu à utiliser une méthode permettant de copier à l'écran des maps plus grosses que celles par défaut sur le BG actif (256x256).
Le principe consiste en fait à sélectionner une partie de la map à copier, comme ici si on prend cette grande map:

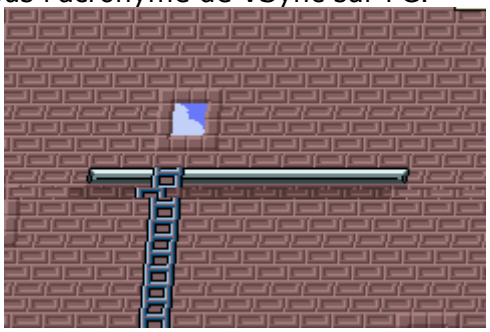


La zone encadrée en rouge contient la seule partie dont on aurait besoin sur cet écran-ci (les images sont extraites du jeu Mario & Luigi sur *GBA* afin d'apporter à ce rapport une touche subtile de variété):

Ce procédé de copie consiste donc à déterminer quelle partie de la map on a besoin et la copier directement. Comme la copie est effectuée au bloc (mosaïque) près, elle a une précision de 8 pixels pour le défilement. Le défilement restant (modulo 8) est effectué par le matériel à partir des registres REG_BG0HOFS et REG_BG0VOFS. Il s'agit de quelque chose qu'il aurait été dur de faire en software, car il aurait fallu vérifier que chaque pixel à dessiner ne dépasse pas de l'écran, et le dessin de la map aurait pris un temps fou.

### (?) Synchronisation, VSync et VBLANK

Lorsqu'on travaille avec l'affichage (que ce soient les BGs, les sprites ou tout autre paramètre graphique), on utilise toujours un tableau en cache (RAM) plutôt que directement la mémoire vidéo. Cela nous permet de travailler sur l'écran sans causer de clignotement ou de cisaillement dû au fait que les informations sont modifiées durant le balayage de celui-ci.
En fait, on ne peut pas directement accéder à l'écran car s'il est en train d'être redessiné à ce moment, le résultat ne sera pas très beau étant donné qu'une partie de l'affichage utilisera les premiers paramètres et une autre les nouveaux. C'est pourquoi on effectue toutes les copies durant une période appelée VBLANK, où le contrôleur graphique "remonte" de la dernière ligne de l'affichage jusqu'à la première. La VBLANK fait dure environ 67 lignes de LCD. Lorsqu'on effectue des opérations durant les 160 lignes (hauteur du LCD) restantes, on est donc obligé de les mettre en cache d'abord, avant de copier le résultat à la fin pendant la VBLANK. Le fait d'effectuer ces opérations durant la VBLANK est appelée synchronisation verticale, plus connue sous l'acronyme de VSync sur PC.



Ci-dessus: le projet /Exemples/Mosaique/ si on oublie d'attendre la Vsync. On remarque d'une part qu'il est trop rapide (l'écran défile plusieurs fois par image), et que cela cause des cisaillements car les informations sont modifiées alors que le contrôleur graphique est en train de bosser.

## 04) Gestion des touches

Bien, maintenant on va attaquer les choses sérieuses. Mais la gestion des touches est un élément encore assez simple à comprendre.

## (!) Registres matériels

L'état des touches nous est renvoyé de par le registre REG_KEYS. Il n'est pas important d'en connaître la définition, étant donné que vous n'allez jamais vous en servir directement. Dans ce registre, seuls les 10 premiers bits contiennent quelque chose d'intéressant. Les bits prennent la valeur un pour indiquer que la touche avec laquelle il est associée est relâchée, tandis qu'ils prendront la valeur zéro si la touche en question est pressée.
Pour un plus grand confort d'utilisation, on masque les bits autre que ceux qui nous intéressent (c à d tout ceux qui sont hors des dix premiers) avant d'inverser l'état du registre. En effet, il semble plus logique que si une touche est appuyée, elle prenne la valeur '1', n'est-ce pas?
C'est ce qui donne REG_TOUCHES, qui n'est pas un registre officiel, mais seulement REG_KEYS inversé et épuré des bits inutiles (avec un AND précisément).
Ensuite, on va pouvoir en extraire les touches dont on a besoin, toujours avec un AND. Petit rappel:
1 ET 1 = 1
1 ET 0 = 0
0 ET 1 = 0
0 ET 0 = 0
En modifiant le premier bit, on remarque que s'il vaut un, le résultat de l'opération est exactement la valeur du deuxième bit (en rouge), alors que s'il vaut zéro, le résultat de l'opération est toujours nul. On peut en conclure:
1 ET X = X
0 ET X = 0
Ainsi, si on veut savoir la valeur d'un bit en particulier, il suffit de mettre tous les bits autres que celui qui nous intéresse à zéro. Le résultat sera la seule valeur du bit extrait; ainsi: pqrs ET 0010 = 00r0. Cette opération permet de déterminer de manière sûre la valeur du bit r!

Il reste maintenant à définir une série de masques permettant d'extraire les touches dont on a besoin. Comme cette recherche n'a aucun intérêt réel, vous pouvez vous servir des #define déjà écrits dans le fichier /Exemples/Mosaique/touches.h inclus au projet.

Maintenant, pour tester une touche, vous l'aurez deviné, ce n'est vraiment pas compliqué:

```
if (REG_TOUCHES&TOUCHE_A)           {
    // La touche A est appuyée
}
```

## (!) Menus et auto-répétition

C'est bien joli, mais cela ne nous donne que l'état courant de la touche. Dans les contrôles des jeux, c'est ce que l'on souhaite, mais dans le cas d'un menu dans lequel on attend par exemple sur la touche flèche haut pour faire monter le curseur, celui-ci montera inévitablement en boucle tant que l'utilisateur laisse appuyée la touche (et ce, environ 60 fois par seconde). Alors qu'un bon menu ne fait monter le curseur qu'au premier appui, nous allons tenter de trouver une astuce permettant de palier à ce problème.

Et cette astuce, là voici. Imaginons que l'utilisateur vienne d'appuyer la touche haut (bit 6), mais qu'il avait déjà appuyé celle de droite (bit 4) avant, cela donnerait:
00 0010 1000
Et avant qu'il appuie cette touche, l'état des touches était donc:
00 0000 1000

Encore une fois, le AND vient à notre aide. Avez-vous remarqué qu'on n'a pas besoin du bit 4 puisqu'il est déjà appuyé depuis un moment? Le but est de trouver un masque qui nous enlève (en l'occurrence met à zéro) tous les bits qui étaient déjà pressés avant. En fait c'est simple, il faut inverser le masque.

| | |
|---|---|
| Valeur actuelle: | 00 0010 1000 |
| Ancienne valeur inversée: | 11 1111 0111 |
| Résultat de l'opération: | 00 0010 0000 |

Le bit 4, qui était déjà allumé la dernière fois sera masqué et la touche ne sera renvoyée qu'à l'appui même (c à d la première fois uniquement). A ce moment, on peut tester ce nouveau "registre" improvisé de la même manière que celui de base. Et il nous signale justement un appui de la touche du haut (bit 6).
Une fois cela terminé, il faut copier la valeur actuelle dans ce qu'on traitera comme l'ancienne valeur pour la comparaison (en fait c'est la "valeur actuelle de la dernière fois", donc avec un temps de retard permettant de détecter les changements entre deux).

Le code est très simple:

```
int etat_clavier, derniere_fois = REG_TOUCHES;

//Boucle principale
while (1)               {
   etat_clavier = REG_TOUCHES & ~derniere_fois;
   derniere_fois = REG_TOUCHES;                    //Pour la prochaine fois
}
```

Le ~ permet d'inverser bit à bit le contenu d'une variable.

Autorépétition

Il ne reste plus que cela. Le principe est lui aussi très simple. Je vérifie si l'état du registre *etat_clavier* ne change pas pendant un certain moment (appelé *délaiInit*), ce qui signifie que l'utilisateur ne touche pas au joystick durant cette période. Ensuite, chaque fois qu'une seconde période (appelée *délaiInter*) est écoulée, je stocke dans etat_clavier l'état réel des touches. En imaginant que l'utilisateur laisse appuyé la touche haut, elle ne sera reportée que la première fois, mais si le délai est écoulé, c'est l'état du registre qui sera directement renvoyé, et la touche haut sera bien appuyée (une fois de plus) à ce moment-là.
La répétition est également active si l'utilisateur n'appuie sur aucune touche (car rien ne change malgré tout). Mais comme l'état de REG_TOUCHES est nul à ce moment-là, aucune touche ne sera de toutes manières reportée.
Ce qui donne le code suivant:

```
#define delaiInit 40
#define delaiInter 8

int etat_clavier, derniere_fois = REG_TOUCHES, nFois=0;

//Boucle principale
while (1)               {
   if (REG_TOUCHES == derniere_fois)               //N'a pas changé?
      nFois++;
   else                                            //Modifié -> retour à zéro
      nFois=0;

   if (nFois>delaiInit && nFois%delaiInter==0)
      etat_clavier = REG_TOUCHES;                  //Directement
   else
      etat_clavier = REG_TOUCHES & ~derniere_fois;      //Filtré


   derniere_fois = REG_TOUCHES;                    //Pour la prochaine fois
}
```

Le code réellement utilisé dans le projet diffère légèrement, mais cette implémentation est plus simple à comprendre. Et il vous reste toujours les commentaires abondants du code.

## *05) Mon « Tilesystem » : Dépasser un peu les limites*

Bien sûr, l'idéalisation du mode de mosaïques est plutôt éloignée de la réalité. Malheureusement, on se retrouve finalement assez limité de tous les côtés suivant ce que l'on désire faire. Pour une grande majorité des jeux, les capacités et la mémoire fournie est suffisante. Mais pour d'autres (notamment en images de synthèse), la mémoire vidéo pouvant contenir jusqu'à 2000 mosaïques différentes ne suffit largement pas pour afficher un niveau entier. On a pour cela recours à une bidouille (ne le prenez pas mal mais ce ne sera certainement pas la dernière!) qui sera expliquée plus bas.
Mais ne vous inquiétez pas, malgré toutes les bidouilles pour dépasser les capacités graphiques de base, on s'en sort toujours, et le mode bitmap reste quand même à des lieues du mode de mosaïques en ce qui concerne les performances.

Voici le principe de fonctionnement de mon "tilesystem" (extrait du fichier tilesystem.h du projet).

### (!) Moteur de gestion de mosaïques

Le problème est la limitation de la mémoire vidéo. Normalement on est censé charger toutes les mosaïques qu'on serait susceptible d'utiliser d'abord en VRAM (mémoire vidéo) et ensuite les utiliser directement.
Mais faisons le calcul. Notre map (256 couleurs, 8 bits), pour chaque mosaïque (8x8 pixels), utilise 64 octets. Or, il y a 64 ko de VRAM pour les arrière-plans, donc 1024 mosaïques si la mémoire vidéo n'était utilisée que pour cela. Mais il faut encore compter qu'une partie sera utilisée pour stocker les maps.
Et la carte de l'EMVs, aussi petite qu'elle soit, utilise déjà 823 tiles (mosaïques). En imaginant une application utilisant une carte un peu plus compliquée, on remarque que la limite serait vite atteinte.

Ce moteur de tiles évolué permet d'afficher sur une même carte un nombre "indéfini" de tiles dans une map, et sur un nombre "indéfini" de plans différents. Forcément, la première idée qui vient à l'esprit serait de charger dynamiquement les tiles dont on a besoin. Cela impliquerait qu'on associe à chaque tile sur l'écran un emplacement différent en VRAM. Malheureusement, ainsi les mosaïques ne sont jamais réutilisées, réduisant l'efficacité de la tilemap.

Il s'agit bien de la méthode utilisée par les programmeurs commerciaux mais cela consomme énormément de mémoire, et si on veut couvrir un seul plan entier en mode 256 couleurs (comme dans le cas du mien), il faudrait 31x21 tiles = 651 tiles, soit 65% de la mémoire.
Ma méthode consiste à associer les tiles virtuelles (pas en VRAM et pas limitées en nombre) aux tiles réelles (qui se trouvent en VRAM). Chaque tile est analysée, puis on regarde si elle se trouve déjà en VRAM. Si ce n'est pas le cas on la charge, sinon on l'utilise directement. Ainsi les tiles identiques peuvent tout de même être réutilisées, économisant de la mémoire vidéo. Les tiles sont entassées à la demande jusqu'à saturation de la VRAM (nombre de tiles en VRAM > MAX_TILES décidé).
A ce moment-là, il faut rechercher les tiles qui ont été chargées mais qui ont disparues entre-temps pour pouvoir les réutiliser. C'est assez simple, un parcours de

la map et une re-validation des tiles qui s'y trouvent encore permet de libérer par la suite toutes celles qui n'ont pas été signalées "présentes".

Ma méthode utilise moins de mémoire, et dépend de MAX_TILES qui est arbitraire (enfin réglé de manière à être sûr qu'il n'y aura jamais plus de tiles que ça sur le même écran). Avec 512 tiles, cela représente un peu plus de 50%, mais on pourrait le réduire à 256 par exemple, 512 c'était juste pour prévoir large. De plus, elle fonctionne sur un nombre indéfini de plans. En contrepartie, elle consomme pas mal de temps CPU supplémentaire (nécessaire pour l'analyse des tiles).
Mais bon, si la vitesse est vraiment un problème, ma méthode est optimisable pour la rendre à peu près aussi rapide que la méthode standard.

Si vous ne comprenez toujours pas le principe de fonctionnement, vous pouvez faire deux choses (en plus de relire ce chapitre):
- Voir le fichier tilesystem.h et tenter de comprendre les routines qui y sont commentées.
- Ouvrir VisualBoyAdvance (inclus au projet) avec la ROM du projet (Map/map_final.gba) et ouvrir le menu Tools… puis Tile viewer. Sélectionnez l'option 256 du groupe "Colors". Cochez la case "Auto update" et réactivez la fenêtre principale (sans fermer celle que vous venez d'ouvrir) et déplacez-vous dans la map de manière à faire défiler l'écran. Vous verrez s'ajouter les tiles en fonction de la demande à l'endroit où vous vous trouvez. Ceci vous permettra d'acquérir une vue d'ensemble qui vous aidera à comprendre le principe si celui-ci reste obscur.

# *06) Gestion des objets*

### (?) Qu'est-ce qu'un sprite ?

Les sprites, aussi appelés parfois objets, sont de petites images qu'on utilise pour tout ce qui est censé se déplacer activement sur l'écran.
Bien que l'arrière-plan soit en général fixe, les objets eux sont très animés et peuvent se déplacer n'importe où. La GBA offre un support matériel pour 128 sprites. Ce qui signifie qu'elle dessinera directement ces petits lutins en utilisant les effets avec lesquels on les paramétrés. Mais attention, le support matériel ne simplifie pas la gestion des objets, il permet juste d'économiser un peu de temps CPU.
Sur GBA, il y a une OAM (Object Attribute Memory) de 1 ko qui contient les attributs nécessaires à la console pour dessiner les objets. Il s'agit d'un tableau (mappé sur le matériel graphique) permettant d'accueillir les paramètres de 128 sprites au maximum (8 octets par sprite, 4 attributs 16 bits).
Ensuite, il y a un emplacement en mémoire vidéo (32 ko) pour stocker les mosaïques des sprites. Comme les sprites ne disposent pas d'une map bien à eux, la place demandée est plus grande (mais ces objets sont bien plus petits que les BG). Les BG ont 64 ko et les sprites 32 ko en mode de mosaïques. En mode bitmap, la bitmap de l'écran utilise 80 ko et seuls 16 ko sont utilisables pour les sprites. En mode bitmap, la première mosaïque utilisable pour les sprites est donc la 512$^{ème}$ puisque chaque mosaïque prend 32 octets. En mode de mosaïques, 1024 sont utilisables.

Même si ce ne serait pas nécessaire dans notre cas, j'ai utilisé une méthode bidouille pour permettre un nombre d'objets (sprites) quasiment infini.

Chaque sprite utilise 4 paramètres (attributs), donc. Parmi les attributs des sprites, on trouve plein de paramètres très intéressants. Chaque paramètre étant sur 16 bits, voici la description de l'utilité des bits :

**Attribut 0**

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | Forme | | Palette | Mosaic | Transparence | | Dblsize | Rotation | Coordonnée Y | | | | | | | |

0-7; Coordonnée Y: … du sprite en pixels. Pour les sprites normaux, il s'agit de la coordonnée du coin haut-gauche, pour les sprites en rotation, il s'agit du centre de celui-ci.
**Note :** Pour utiliser des coordonnées négatives, il faut afficher le sprite aux positions 256+y si y<0. Par exemple, pour afficher le sprite à la coordonnée –1, il faudra utiliser la valeur 255.

8; Rotation: Active ou non la Rotation ou le Scaling
9; Double size: Utilise un buffer deux fois plus grand que la taille du sprite pour la rotation. Cela lui permet de déborder du carré de base qui lui est alloué. En imaginant un sprite de 16x16 qui est roté à 45°, on remarque ses dimensions deviennent selon pythagore : $(16^2+16^2)^{1/2}$ = 22.62 pixels. Un carré plus grand que 16x16 lui est donc nécessaire.

A-B; Transparence: (alpha-blend).
- 00 – Normal (opaque)
- 01 - semi-transparent (utilise COLEV)
- 10 – Dans la fenêtre OBJ WINDOW
- 11 – Illégal. Pas essayé ;)

**Note :** si un sprite se trouve derrière un autre, on ne verra que celui qui est tout au-dessus, même s'il est transparent. Par exemple, on peut remarquer dans le projet que les points qui se trouvent derrière la boîte de dialogue transparente n'apparaissent pas en dessous.

C; Mosaïque: Utilise l'effet mosaïque. Je ne l'ai pas testé.
D; Palette: Change le mode de palette (1=256 couleurs 8 bits, 0=16 couleurs 4 bits -> utilise un numéro de palette)

E-F; Forme: Forme du sprite. Voir plus bas

**Attribut 1**

Pour les sprites standard:

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | Taille | | MV | MH | x | x | x | Coordonnée X | | | | | | | | |

Si le mode rotation/scaling est activé

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | Taille | | Rotset | | | | | Coordonnée X | | | | | | | | |

0-8; Coordonnée X: … du sprite. Mêmes remarques que pour la coordonnée Y, excepté qu'en cas de coordonnées négatives, il faut appliquer 512+X.

C; Miroir horizontal: Active le retournement horizontal
D; Miroir vertical: Active le retournement vertical

9-D; Rotset: Index du numéro de "rotset" à utiliser pour le sprite. Utilisé si la rotation est activée, et peut aller de 0 à 31. Les rotsets sont définis à partir de la

place gagnée par les ''attribut 3'' non-utilisés accumulée sur 4 sprites, permettant 32 rotsets (128/4).

E-F; Taille: Dimensions du sprite. Utilise les deux bits (E-F) de l'attribut 0 pour obtenir la taille entière du sprite. En violet, le bit S (size) de l'attribut 0 et en vert celui-ci :

| | |
|---|---|
| 0000: 8 x 8 | 1000: 8 x 16 |
| 0001: 16 x 16 | 1001: 8 x 32 |
| 0010: 32 x 32 | 1010: 16 x 32 |
| 0011: 64 x 64 | 1011: 32 x 64 |
| 0100: 16 x 8 | 1100: Inutilisé |
| 0101: 32 x 8 | 1101: Inutilisé |
| 0110: 32 x 16 | 1110: Inutilisé |
| 0111: 64 x 32 | 1111: Inutilisé |

**Attribut 2**

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | N° Palette | | | | Priorité | | Mosaïque de référence | | | | | | | | | |

0-9; Mosaïque: Numéro de mosaïque de référence pour le sprite. Les sprites utilisent eux aussi des mosaïques de 8x8. Par exemple, un sprite de 16x16 utilisant la mosaïque de référence 2 ressemblera à cela (index des mosaïques utilisées) :

2 3
4 5

A-B; Priorité: Les sprites passent toujours par-dessus les BG de même priorité.

C-F; N° Palette: … à utiliser, uniquement en mode 16 couleurs.

**Attribut 3**

Ce paramètre n'est pas utilisé pour les sprites. Il sert à définir les rotsets uniquement!

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | Signe | Partie entière | | | | | | | Partie fractionnaire | | | | | | | |

La GBA utilise une arithmétique avec des nombres à virgule fixe, qui sont définis comme suit:
0-7; Fraction: (1/256$^{ème}$ d'unité).
8-E; Partie entière.
F; Bit de signe.

J'ai bien sûr écrit quelques macros pour simplifier la manipulation des sprites (fichier sprites.h).

**(!) Moteur de gestion des objets**

Ma méthode est loin d'être la plus simple. En fait, je dirai même qu'il s'agit d'une méthode avancée, complètement inutile pour afficher simultanément trois ou quatre sprites au maximum à l'écran… mais bon, la plupart des jeux utilisent des méthodes semblables.

En fait, à chaque image, on vide tous les sprites présents en mémoire vidéo et on place les nouveaux à l'aide de la fonction CreeObjetStandard qui crée un objet avec les paramètres standard. Il suffit de modifier par la suite les paramètres spécifiques à chaque sprite (comme sa position) et voilà le travail.
Ensuite, les tiles correspondantes aux sprites utilisés sont copiées en mémoire vidéo. Chaque sprite n'a qu'à indiquer les tiles qu'il veut utiliser ainsi que certains paramètres comme sa taille et sa palette. Le système vérifiera si les tiles du sprite existent déjà et les copiera en mémoire vidéo sinon.

## *07) Fonctions graphiques d'accès au pixel*

Pour le texte, ce fût assez compliqué. Bien sûr déjà le premier problème étant de réaliser quelque chose de suffisamment rapide pour pouvoir être utilisable. Eh oui, la GBA s'essouffle assez rapidement lorsqu'on lui demande des tonnes d'accès à la VRAM (le timing d'accès de la VRAM sous 16 bits étant de 3 ou 4 cycles selon que la GBA est en train d'y accéder au moment même ou non, et le double en 32 bits).

**(!) Le mode 0 est utilisé. Qu'est-ce que cela implique?**

L'utilisation du mode de mosaïques impose tout de même quelques complications pour notre projet, dont celle de ne pas permettre un accès au pixel près (forcément, puisqu'on travaille avec des mosaïques complètes).

Le mode de mosaïques permet d'afficher facilement du texte, pour autant que la taille de chaque caractère soit toujours identique (normalement en fait chaque caractère devrait faire 8x8 pixels, ce qui rendrait les caractères vraiment "carrés"), ce qui n'est pas notre cas.

Pour la plupart des jeux, ce n'est pas vraiment gênant, mais les jeux de rôle (RPG) ont meilleur temps d'implémenter un système permettant de contourner ce problème, sous peine d'avoir un système de dialogues limité et graphiquement infect. Et c'est ce que nous allons faire.


Exemple de jeu[*3] utilisant un système de dialogue limité et graphiquement infect.
Notez les caractères applatis pour tenir dans des mosaïques de 8x8!

Mais comme on aimerait dessiner du texte avec une police non proportionnelle, on est obligé de simuler un mode bitmap sur un BG mosaïque. En fait, le principe est assez simple: on associe une mosaïque différente à chaque bloc de la map de l'écran. En modifiant directement les bonnes mosaïques en mémoire vidéo, on affecte indirectement l'écran. Les calculs sont plus complexes, mais cela reste malgré tout avantageux par rapport à un mode bitmap pur, car les pixels sont codés sur 4 bits (au lieu de 8 ou 16 en mode bitmap).

## (!) Principe de mes fonctions de dessin

Le principe des fonctions à accès aux pixels a toujours été un peu du bidouillage. Et bien sûr, la GBA ne fait pas exception à la règle, car dessiner les pixels un à un serait impensable avec un processeur à 16 MHz. On va donc se débrouiller pour écrire plusieurs pixels à la fois. Voici la méthode que j'ai utilisé:

**Note:** Avant de lire ceci, je vous conseille vivement de lire le petit document de cours inclus à ce projet. Vous comprendrez mieux comment l'écran est formé.

Tout d'abord, il faut trouver l'adresse du motif à modifier. Etant donné que l'écran fait 240x160 pixels, il y a 30x20 mosaïques de 8x8 sur l'écran. La map que l'on va utiliser est donc définie ainsi :

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 | | 54 | 55 | 56 | 57 | 58 | 59 |
| 60 | 61 | 62 | 63 | 64 | 65 | | 84 | 85 | 86 | 87 | 88 | 89 |
| 90 | 91 | 92 | 93 | 94 | 95 | | 114 | 115 | 116 | 117 | 118 | 119 |
| 120 | 121 | 122 | 123 | 124 | 125 | | 144 | 145 | 146 | 147 | 148 | 149 |
| 150 | 151 | 152 | 153 | 154 | 155 | | 174 | 175 | 176 | 177 | 178 | 179 |
| | | | | | | […] | | | | | | |
| 450 | 451 | 452 | 453 | 454 | 455 | | 474 | 475 | 476 | 477 | 478 | 479 |
| 480 | 481 | 482 | 483 | 484 | 485 | | 504 | 505 | 506 | 507 | 508 | 509 |
| 510 | 511 | 512 | 513 | 514 | 515 | | 534 | 535 | 536 | 537 | 538 | 539 |
| 540 | 541 | 542 | 543 | 544 | 545 | | 564 | 565 | 566 | 567 | 568 | 569 |
| 570 | 571 | 572 | 573 | 574 | 575 | | 594 | 595 | 596 | 597 | 598 | 599 |

On peut tout d'abord calculer assez simplement le numéro de mosaïque à modifier en divisant les coordonnées (x,y) par huit (la taille d'une mosaïque).
Les pixels entre 0 et 7 se trouve donc dans la première mosaïque, ceux entre 8 et 15 dans la deuxième, ceux entre 16 et 23 dans la troisième, et ainsi de suite. Ceci est valable en x (horizontalement) comme en y (verticalement).
Comme chaque ligne comporte 30 mosaïques (240/8), on peut élaborer la formule qui nous permet de calculer le numéro de mosaïque dans laquelle est contenu un pixel *x,y* de la manière suivante (où *largeur* est la largeur d'une ligne, ici 30) :
largeur*partieEntière(y/8)+partieEntière(x/8)

Mais il reste encore à effectuer la liaison avec le code. Comme nous sommes en mode 4 bits par pixel, chaque mosaïque de 8x8 (=64) pixels prend 64x4=256 bits, soit 32 octets.
**Note :** J'adresse la mosaïque sur 32 bits, ce qui me permet de modifier directement 8 pixels à la suite, soit une ligne de mosaïque entière). Cela signifie que *adresse+1* pointe en fait 4 octets plus loin que *adresse*.
C'est pourquoi, pour chaque mosaïque, le saut nécessaire sera de huit (32/4). Et par chance, cela simplifie largement notre calcul, comme nous allons le voir plus bas.

## (>) Calcul de l'adresse

En tenant compte des paramètres ci-dessus, le calcul pour trouver l'adresse en mémoire de la mosaïque de destination est donc le suivant:
Adresse=premMosaïque+((x/8)+(y/8)*largeur)*8;
En connaissant l'adresse de la première mosaïque *premMosaïque*, la *largeur* en mosaïques d'une ligne, et les positions *x,y*, tout ce dont on a besoin est là.
Maintenant, ceci est un peu simplifiable et optimisable, car les divisions sont extrêmement lentes avec le processeur de la GBA, qui ne dispose pas de support matériel pour celles-ci.
Comme une mosaïque représente 8 *mots* 32 bits, et que le seul but de la division par huit est d'arrondir à la mosaïque près, on peut simplifier facilement, car :

Si x vaut 27, le but de la manipulation est simplement d'arrondir x au multiple de 8 inférieur :
(x/8)*8 -> 24

Pour cela, on peut effectuer une opération plus simple et plus rapide pour le processeur, il s'agit de la suivante :
x&~7 -> 24 aussi
Un AND de l'inverse de 7 permet de masquer (mettre à zéro) les trois derniers bits, et de laisser intact les autres. Forcément, avec les trois derniers bits à zéro, un nombre sera toujours multiple de huit :
11011 -> 27
Après le AND de la valeur 11000, cela donne bien :
11000 -> 24

En appliquant la même méthode en y, cela donne une formule de calcul de l'adresse bien plus efficace pour le microprocesseur :
adresse=premMosaïque+(x&~7)+((y&~7)*largeur);

Maintenant qu'on a trouvé l'adresse de la mosaïque, il reste encore à trouver l'adresse de la ligne qu'on veut modifier… mais ce n'est pas compliqué. Il n'y a pas besoin de calculer d'offset en x, puisqu'un mot 32 bits couvre une ligne entière! Il suffit donc d'ajouter le numéro de ligne à l'intérieur de la mosaïque, donc modulo 8. Pour effectuer un modulo 8, on peut utiliser un AND 7, ce qui masque tous les bits sauf les trois derniers, qui ne peuvent contenir que des valeurs entre 0 et 7.
```
adr+=(y&7);
```
11011 -> 27
Après le AND de la valeur 00111, cela donne bien :
00011 -> 3 (27 modulo 8 = 3)

**(>) Stocker le motif (8 pixels d'un coup)**

Maintenant que l'adresse du motif à modifier est trouvée, il reste à calculer ce qu'on va y stocker !

Au départ, comme sur à peu près toutes les machines, il faut passer un masque (AND) sur les données de la mosaïque. Les valeurs à zéro effaceront les pixels qui devront être modifiés, et les valeurs à 1 les laisseront intacts. Ensuite, on pourra écrire la nouvelle valeur souhaitée à l'aide d'un OR sans modifier les pixels qui ne font pas partie du motif. C'est commun comme méthode et cela permet de modifier plusieurs pixels en un accès.

Prenons le cas où l'on veut dessiner un motif de 8 pixels (4 bits chacun) à la position x de 1. L'offset en y ne change absolument rien (il a déjà été calculé plus haut).
D'abord: l'application du masque. Les valeurs à 0xf représentent là où on écrira un pixel. Ce sera inversé par la suite, comme on le verra. On le fait d'abord décaler de 1 (x modulo 8). Il faut savoir que par rapport à notre représentation d'un nombre hexa, 0x12345678 par exemple, les pixels affichés à l'écran seront en fait 87654321. Donc pour faire décaler à droite, on fera en fait décaler le nombre vers la gauche.
Masque: 0xffffffff, avec décalage (4 bits car 1 pixel d'offset): 0xfffffff0.
Après inversion, on obtient 0x0000000f. Seul les 4 premiers bits ne seront pas modifiés, soit le premier pixel, comme voulu (car on dessine à la position 1, donc pas de modification du pixel 0!).

Effectuer l'opération ci-dessus avec un masque standard (non inversé) n'eût pas été possible, car si on se retrouve avec 0x00000000 (=effacer tous les pixels), un

décalage dans quelle direction que ce soit ne donnera pas 0x0000000f puisque les bits décalés sont remplis avec des zéros et non des 'f'.

Maintenant l'écriture se fait de la même manière avec un OR. Un motif de 0x12345678 sera, après décalage, écrit 0x23456780, (les bits à l'extrémité ne seront pas écrits. Le pixel 1 n'est définitivement pas modifié (ouf!) car x OR 0 = x. Ensuite, il faut écrire la fin du motif (le pixel restant). On modifie le pointeur pour le positionner sur la prochaine mosaïque (qui pointera donc 32 octets plus loin, soit 8 mots 32 bits). Pour écrire le reste du motif, on fait la même chose qu'avant mais avec le comble du décalage (8-décalage courant), donc un décalage de 7 dans ce cas-là. Je vous laisse refaire le raisonnement avec cette valeur, on obtient bien ce qu'on désire...

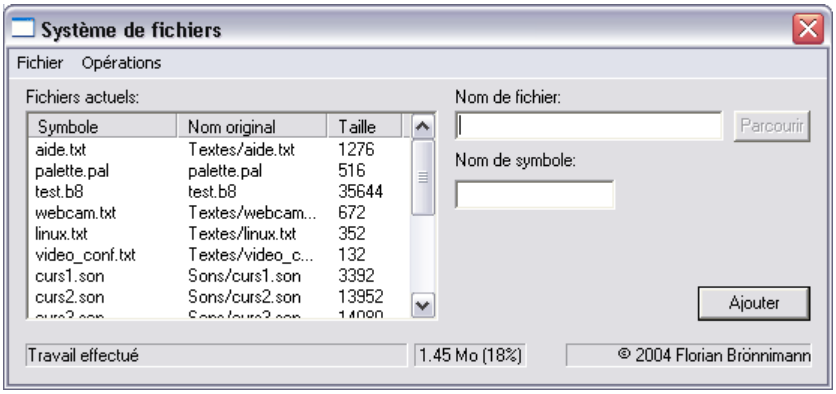
Jetez un coup d'œil au fichier texte.c du projet. Vous y trouverez également une petite partie de l'explication ci-dessus ainsi qu'une réelle implémentation de ces routines. Même si certaines ne sont pas utilisées, je les ai implémentées car elles pourraient être utiles pour la suite du projet.

## *08) Système de fichiers*

La Game Boy Advance ne peut pas (et n'est pas prévue pour) lire des fichiers comme sur PC. Il faut pour cela se débrouiller comme on peut. Mon idée a été de simplement accoler un fichier binaire au fichier de sortie (exécutable). Le système de fichiers rechercherait le début du supplément binaire (là où se trouvent les données, ainsi qu'une table de localisation de celles-ci) et pourrait y rechercher des fichiers. J'ai pour l'occasion implémenté un programme : « filesystem.exe ». Comme j'étais très pressé, j'y ai juste apporté le soin nécessaire pour fournir quelque chose de fonctionnel et fiable mais l'ergonomie et le graphisme restent à réviser sérieusement (je n'avais pas que ça à faire durant ce projet) ; le but étant simplement qu'il fasse ce que je lui demande : permettre d'ajouter des fichiers et créer une archive de données binaires contenant une table de localisation. Accessoirement, il permet, en un appel en ligne de commande (filesystem /updateall) de mettre à jour tous les fichiers ajoutés au projet. Tous seront relus et ajoutés à l'archive binaire. Ceci est exécuté par le makefile à chaque compilation pour s'assurer que le système de fichiers contient toujours les dernières versions des fichiers de données.
Ce programme peut également lire les fichiers source C (.c). Il recherchera un tableau précédé de l'attribut «const» et « pseudo compilera » les données avec un interpréteur interne (acceptant les nombres décimaux et hexadécimaux) et ajoutera les données au fichier binaire. Tous ces procédés sont extrêmement rapides, ce qui permet de mettre à jour plusieurs Mégaoctets dans l'archive de données en moins d'une demi-seconde. De plus, on peut grâce à ça travailler avec les données générées par GBA Graphics (qui sont au format C) et s'économiser du temps de compilation (GCC étant extrêmement lent à compiler de gros tableaux).

### (?) Utilisation du logiciel

Lancez-le. Tapez dans la case à droite le nom du fichier (accessoirement suivi du répertoire). Dans la mesure du possible, utilisez toujours des chemins relatifs (par exemple Textes/aide.txt), le projet en sera d'autant plus portable (C:\Documents and settings\Florian\Desktop\My documents\Pro_gba\Map\Fichiers\Textes\aide.txt n'existe peut-être pas chez tout le monde).




Vous pouvez mettre à jour des fichiers individuellement en en sélectionnant un et en cliquant sur Mettre à jour. Accessoirement, vous pouvez modifier leur nom de cette manière.
Vous pouvez mettre à jour tous les fichiers. En général, le temps demandé à l'opération est très petit. Cependant, un message de statut apparaîtra dans la barre (au fond à gauche de la boîte) pour indiquer que l'opération est en cours.
**Attention :** Ce programme n'affichera pas une boîte de dialogue typique en cas d'erreur, cependant celle-ci sera reportée dans la barre de statut.

Vous pouvez glisser-déposer les fichiers dans le programme à partir de l'explorateur. Mais faites tout de même attention à utiliser des noms relatifs dans la mesure du possible, sinon votre projet ne sera pas utilisable sur un ordinateur ayant une structure de fichiers différente de la votre (en gros, tout autre ordi que le votre).

Je crois que ce programme est bien assez simple à utiliser comme ça. Faites tout de même attention lorsque vous exécutez le programme à partir d'un fichier batch, le répertoire d'exécution. Un filesystem.exe /updateall n'aura aucun effet si vous ne vous trouvez pas dans le même répertoire que votre projet! (projet.pfs)

## (!) Recherche du fichier binaire dans la ROM

Tout d'abord, mon makefile appelle un utilitaire nommé gbafix.exe (inclus dans HAM) qui aligne les données au kilooctet près. Lorsqu'on parcourra la ROM à la recherche du système de fichiers, on pourra le faire par incrémentation de 1024 octets, ce qui est plus rapide que de tester octet par octet.
La fonction InitFS effectue cette recherche. Elle doit évidemment être appelée avant de tenter d'utiliser le système de fichiers.
Cette fonction parcourt la ROM entière depuis son adresse jusqu'à une limite supérieure (elle s'arrêtera si elle ne trouve définitivement rien). Le fichier binaire du système de fichiers est précédé du texte suivant (cela permet de le reconnaître).
**** Game boy advance filesystem by Brünni ****
Inclure la chaîne entière au code C causerait un problème! En effet, il risquerait de se prendre lui-même pour le système de fichiers en recherchant dans le code. C'est pourquoi les quatre premiers octets (****, valeur ASCII assemblée 0x2a2a2a2a) sont d'abord recherchés, puis, s'ils sont trouvés, on analyse la suite de la chaîne (qui devrait se trouver immédiatement après).

La fonction SymFind recherche un fichier ayant le nom désiré. Comme elle n'est vraiment pas difficile à comprendre, elle ne sera pas commentée ici.
Pour plus d'informations, regardez le fichier filesystem.c.

# 09) Convertisseur de graphismes

Pour convertir vos graphismes afin de pouvoir les utiliser sur votre GBA, vous pouvez utiliser le programme que j'ai écrit et qui est fourni au projet: GBA Graphics.
Il s'agit d'un convertisseur de graphismes ainsi qu'un éditeur de maps. Pas besoin de Photoshop ou de tout autre lourd programme de retouche graphique, car GBA Graphics effectue lui-mêmes les tâches communes plus avancées, de manière à ce que MS Paint suffise pour effectuer tout ce dont vous auriez besoin pour bien démarrer dans la programmation de cette console.

GBA Graphics convertit les graphismes (bitmap 24 bits) au format C, compilables et utilisables directement dans vos programmes, enfin pour peu que vous savez vous en servir. Il inclut également un éditeur de maps universel. Je n'ai malheureusement pas eu le temps de terminer l'implémentation du support multi-format (le temps disponible pour le projet m'a coupé l'herbe sous le pied). Cependant, un support basique est tout de même implémenté, et il permet de créer des bitmaps 8 bits au format binaire directement, à utiliser avec le lecteur de textes.

Pour ce programme, j'ai rédigé une documentation complète que j'ai incluse au programme. Pour la lire, rendez-vous dans le dossier du projet :
`/Tools/GBA Graphics/Français/`
Vous pouvez la lire en démarrant le programme et en appuyant sur la touche F1 ou en ouvrant le fichier Aide.htm.

# 10) Système de menus

Revenons-en à la chose principale : le programme « Map » pour GBA.
Le système de menus intégré permet d'afficher et de permettre la navigation dans un menu quelconque (dont le texte a été écrit avec le prog à Zanella).

**(!) Explication du fonctionnement**

Il s'agit d'une partie assez complexe du programme. Le fichier qui contient les fonctions définies pour le menu est infos.h. Il est assez abondamment commenté, c'est pourquoi je ne traiterai ici que des points les plus sensibles.

Types de données définis

**Structure INFO:**
```
typedef struct      {
    u32 x0,y0,x1,y1,event;
} INFO;
```
**Note:** le type u32 est un entier 32 bits non-signé, et cela correspond au type standard unsigned long.
**Utilité:** Cette structure sert à stocker les données qui déclenchent un événement sur la map principale. Elle contient les coordonnées du rectangle représentant la zone de collision (pour le curseur) dans laquelle l'événement doit être déclenché. Est également défini un lien vers le numéro d'événement (à l'intérieur du tableau d'événements nommé *evenements* et défini dans le fichier zones.h) qui contient les données additionnelles, comme le texte de la boîte à afficher.

**Structure EVENEMENT:**
```
typedef struct      {
    char *titre;
    char *texte;
```

```
    unsigned char *(*fonction)();
} EVENEMENT;
```

**Utilité:** Cette structure contient les données nécessaires pour dessiner une boîte d'informations ainsi que l'éventuel menu qu'elle peut contenir.
Elle comprend le titre de la boîte, le texte qui se trouve à l'intérieur et une fonction gestionnaire de menu (qu'on abordera un peu plus loin).

**- Titre :** Contient une chaîne de caractères simple indiquant le titre de la boîte. Il est dessiné en fonte moyenne et en gras.
- **Texte :** Contient une chaîne formatée avec des liens.
**- Fonction :** Contient une fonction de gestion du menu.

Pour le texte du menu, il s'agit d'un texte simple, avec des \n pour indiquer la fin d'une ligne. Il est possible de définir des liens. Ces liens peuvent bien entendu être découpés sur plusieurs lignes, cependant un \n signifie obligatoirement la fin de celui-ci.
De plus, s'il y a plus de texte que la taille de la boîte, le menu va s'occuper de gérer cela au mieux pour qu'on voie toujours les informations les plus importantes.
Les liens débutent par un caractère # et finissent par un #. La syntaxe à l'intérieur de cette balise est la suivante :
#lien.txt|Message#Lien\n
Tout le reste de la ligne (jusqu'au \n) est considéré comme le texte associé au lien. C'est celui qui apparaîtra en surbrillance lorsqu'il sera sélectionné. Le lien (fichier texte) est le nom du fichier à afficher dans le lecteur lorsque le lien sera sélectionné. Le message est affiché au fond de l'écran lorsque le pointeur se trouve sur cette option.

Affichage des menus

C'est la fonction DessTexteSpecial du fichier texte.c qui s'occupe de dessiner l'intérieur de la boîte (pouvant contenir un menu). Le prototype de cette fonction, est le suivant:

```
void IN_IWRAM DessTexteSpecial(u32 x, u32 y, char *chaine, u32 couleur,
u32 selection, u32 offset, unsigned char *(*handler)());
```

Ne vous souciez pas pour le moment du mot clé IN_IWRAM, car il n'interfère en rien avec le déroulement de la fonction, ni avec son appel.

Arguments:

x, y: Coordonnées du sommet où sera dessiné le texte.
chaine: Chaîne de caractères représentant le texte (pouvant contenir des balises) à analyser et afficher.
couleur: Couleur de base des options de menu inactives. Les options sélectionnées utiliseront la couleur+1. Chez moi, les options utilisent la couleur 3, qui est blanche, et l'option sélectionnée utilise la couleur 4, qui nuance entre le rouge et le jaune.
selection: Indique le numéro de l'option sélectionnée. Cette option sera affichée avec une couleur différente, conformément à ce qui est indiqué ci-dessus.
offset: Ajoute un offset (défilement) vertical au texte. L'unité est en pixels. Cela permet de n'afficher le texte par exemple qu'à partir de la troisième ligne, donnant l'impression que le texte a défilé de deux lignes.
handler: Fonction gestionnaire de menu. Si le paramètre n'est pas *NULL*, le menu questionnera la fonction lorsqu'il lui manque des informations pour former son menu. Pour plus d'informations, voyez plus bas dans ce chapitre.

**(!) Gestionnaire de menu?**

Avez-vous remarqué le menu pause? Eh bien il n'est pas lui non-plus codé en dur, mais il utilise cependant un gestionnaire d'actions.

Le principe est simple: le menu "questionne" le *handler* à propos de ce qu'il a besoin pour composer son menu. Bien entendu, on pourrait étendre les possibilités au gré du nécessaire. Ici, le *handler* est questionné pour les options de menu inconnues (non-spécifiées en fait) et pour exécuter une action lorsqu'une option est sélectionnée. A ce moment-là, il renverra s'il faut fermer le menu ou non.
Par exemple, l'option "Reprendre" ferme le menu, mais la (dés)activation des bruitages par exemple ne le fait pas...
Pour insérer un lien dynamique, il faut que le lien soit vide. Par exemple :
##Reprendre\n
Il est possible de spécifier un message :
#|Appuyez sur le bouton A#Reprendre\n
Si le lien est nul, le handler sera questionné sur l'action à entreprendre lorsque l'utilisateur appuiera sur la touche de validation (A).
Mais il est également possible de ne pas spécifier le texte associé au lien, comme dans le cas suivant:
#test|Test#\n
A ce moment-là, le handler (si défini) sera questionné sur le texte à afficher pour le lien. Il devra alors retourner une chaîne de caractères contenant le texte à écrire.
Bien sûr, vous pouvez ne définir ni message, ni lien, ni texte pour celui-ci. A ce moment-là, le handler effectuera intégralement le travail. Voici le prototype des handlers:

```
unsigned char *NomHandler(int demande, int param, int touches);
```

Ce handler retournera éventuellement une chaîne de caractères (ou *NULL* s'il n'y en a pas besoin).
Sont fourni les paramètres suivants:
demande: La demande effectuée par le système de menus. Les valeurs peuvent être les suivantes:

- DEM_OPTION: Demande de renvoyer la chaîne correspondante au numéro de l'option contenu dans *param*.
- DEM_ACTION: Demande d'effectuer une action lorsque l'option contenue dans la variable *param* a été sélectionnée par l'utilisateur. Il faut renvoyer une valeur pour indiquer ce qu'il faut faire à partir de ce moment:
  - 0: Le menu doit être mis à jour (le texte a été modifié). Le menu reste ouvert.
  - 1: Le menu ne doit pas être mis à jour, et doit rester ouvert (rien ne se passe).
  - 2: Le menu doit être fermé.
- DEM_GESTION: Cette demande est envoyée périodiquement, principalement pour demander de gérer les touches pressées par l'utilisateur. Les touches ne sont envoyées qu'au flanc montant (lorsque ces touches viennent d'être appuyées et non si elles sont appuyées à ce moment-là).

param: Numéro d'option sélectionnée. Pourrait être autre chose suivant le contexte des messages envoyés, mais pas dans le cas de notre projet.
touches: Etat courant des touches. Les bits à 1 indiquent les touches qui viennent d'être pressées (et qui ne l'étaient pas la dernière fois).

Pour un exemple concret de l'utilisation d'un handler, consultez le fichier menupause.h.

# 11) Adaptation à vos besoins

Modification de la carte utilisée:

Il n'est pas trop compliqué de changer l'image de la carte. Voici la marche à suivre : Remplacez fond.bmp par votre image (bitmap 24 bits obligatoire) de taille quelconque dans le dossier /Map/Res/ du projet. Ensuite, démarrez GBA Graphics et sélectionnez la page « Mosaïques » dans l'arbre. Faites « Parcourir… » et choisissez « fond.bmp ». Changez le nombre de couleurs à 192 et appuyez sur Entrée, acceptez, après l'avoir lu, le message d'avertissement éventuel.
Quittez le programme et exécutez all.exe dans le même dossier. Vous pouvez maintenant recompiler le programme avec la nouvelle carte.

Modification des sons:

Pour les fichiers .son et .8ad, il suffit de modifier un son dans le dossier /Fichiers/Sons/ (si vous en ajoutez un nouveau, n'oubliez pas de l'ajouter au dossier filesystem après l'avoir converti au bon format à l'aide des outils "wav2son" et "wav28ad") et de l'enregistrer, et finalement d'exécuter FaitSons.bat. Vous devrez modifier le fichier batch pour convertir d'autres musiques, cela va de soi, mais les exemples sont déjà là, donc ça ne devrait pas être trop compliqué...

Modification des menus et événements:

Utilisez pour cela le programme à David Zanella; C'est quand même son utilité principale!

Pour modifier des éléments relatifs au lecteur de texte, utilisez le programme convert.exe, dont l'utilisation est décrite dans le chapitre ci-dessous.

**(!) Important!**

Lorsque vous sauvez le fichier depuis le programme à mon collègue David, nommez-le zones.h et enregistrez-le dans le dossier du projet, de manière à remplacer le fichier existant. Depuis le système de menus, ce fichier est inclus et compilé avant de pouvoir être utilisé en tant que tel, ce qui signifie qu'il vous faudra recompiler le programme pour voir les changements. Lorsque mon collègue a commencé son travail, je n'avais pas encore écrit le système de fichiers, et nous nous sommes donc mis d'accord sur le format C pour l'import/export des données…

# 12) Le lecteur de textes

**(>) Description générale**

Le lecteur de textes utilise le mode 4 (bitmap 8 bits). Il affiche des textes formatés avec des balises simples. Ces balises permettent de modifier la couleur du texte, le mettre en gras, en italique, etc. Pour une description de ces formats, voyez le programme de mon collègue David Zanella. Vous pouvez éditer les fichiers texte avec le bloc-notes Windows ou avec le programme à Zanella, et finalement les insérer dans le projet du système système de fichiers (filesystem).

**(!) Formatage**

Pour formater votre texte, vous devez utiliser des codes de format. Ceci indique au lecteur de texte ce qu'il doit faire. Les codes de formats sont toujours sur 3 caractères, un qui permet au lecteur de détecter que ce qui va suivre est un code de format, un autre donnant la propriété du format à modifier, et un dernier donnant la nouvelle valeur à assigner à la propriété.
D'une manière générale, la valeur zéro signifie *désactiver* un effet, alors qu'un autre nombre permet de l'activer et, par la même occasion, de choisir entre les différentes possibilités. Voici quelques exemples: ©C1, ©G1, ©G0.
Voici la liste des propriétés:
©Cx: x entre 0 et f (en hexadécimal), change la couleur du texte. N'utilisez pas la couleur zéro ou vous ne verrez plus rien.
©Gx: x entre 0 et 3, applique un effet au texte (0=normal, 1=gras, 2=coulé, 3=ombré).
©Ix: x entre 0 et 1, active ou non l'italique.
©S1: x entre 0 et 1, active ou non le soulignement.
©Fx: x entre 0 et *nFontes*-1, choisit entre les fontes chargées. Au départ, seule la fonte par défaut est chargée. Elle porte le numéro 0, tandis que celles que vous rajouterez par le biais de la commande #F prendront les numéros suivants.
©C+xx: Permet de choisir parmi une couleur de la palette physique. Pour visualiser toutes les couleurs, ouvrez GBA Graphics (depuis le dossier /Map/Fichiers/ du projet), choisissez le format binaire (Affichage -> formats de fichier… -> Binaire), ouvrez l'élément *Palettes* de l'arbre du projet et choisissez l'élément *palette*. A droite s'affichera la palette utilisée dans le lecteur de textes. Pointez une couleur avec la souris et recopiez le nombre hexadécimal inscrit sur la partie droite de la fenêtre du programme ("Idx: FE hex" par exemple). Dans ce cas, cela ferait ©+fe.

**(!) Commandes**

Le lecteur de texte vous permet d'intégrer quelques commandes simples au texte. En général, celles-ci sont utilisées pour paramétrer le texte ou pour dessiner divers objets graphiques.

Syntaxe

La syntaxe de base est assez simple. On doit tout d'abord ouvrir une balise de commande à l'aide du caractère #, puis la refermer une fois terminé avec ce même caractère. A l'intérieur de la balise, on trouve tout d'abord un premier caractère indiquant quelle est la commande à exécuter. Ensuite, comme dans n'importe quel langage, il faut spécifier des *arguments* utilisés par la commande. Ceux-ci, pouvant être de type numérique ou texte, seront automatiquement analysés, et remis au lecteur de textes sous une forme plus adaptée à son usage. Voici le squelette de chaque commande:
`#C1,2,3,texte#`
Le C est le caractère de commande (à choisir parmi l'un de ceux décrits ci-dessous), les nombres sont des arguments numériques (qui viennent toujours au début), et, toujours à la fin, le texte.

Système de coordonnées

Ce système est doté d'un analyseur d'arguments numériques qui permet de spécifier des coordonnées relatives ou absolues à partir de la position actuelle du curseur. Pour spécifier des positions relatives, vous devez insérer un + ou un – juste avant la valeur numérique. Voici quelques exemples, en utilisant la commande #C, qui déplace le curseur:
#C20# -> Bouge le curseur à la position 20

#C+20# -> Déplace le curseur de 20 pixels vers la droite
#C-20# -> Déplace le curseur de 20 pixels vers la gauche
Lorsque des paramètres sont ignorés, la position par défaut est prise; il s'agit de celle du curseur. Par exemple, #C,2# ne modifiera pas le premier paramètre (x), mais juste le deuxième (y).

Plusieurs commandes à la suite

Il est courant de vouloir spécifier plusieurs commandes à la suite. Pour alléger un peu l'allure du document brut, vous pouvez utiliser une barre verticale | pour séparer les diverses commandes, avant de refermer le # une fois pour toutes. Par exemple:
#C0|Pimage.b8# -> Déplace le curseur **puis** dessine une image

Commandes disponibles

Voici une liste des commandes disponibles, ainsi que les arguments qu'elles demandent. Lorsque vous trouvez des crochets une définition de commande, cela signifie que cette partie est facultative. Prenons par exemple ce cas:
#C[x][,y]#
Vous pourriez alors l'écrire de multiples façons différentes; comme par exemple: #C# (tous les paramètres ont été ignorés; ne sert à rien dans ce cas), #C10# (seul le paramètre y a été ignoré), #C10,2# (aucun paramètre ignoré), ou encore comme ceci: #C,2# (seul le paramètre x a été ignoré).

#A*align*#: Modifie l'alignement du texte. Les valeurs possibles sont les suivantes: 0=gauche, 1=centré, 2=droite.
#C[*x*][,*y*]#: Déplace le curseur. Même si vous modifiez la position y, vous ne pouvez pas sortir de la ligne en cours (en clair, deux lignes ne peuvent pas se marcher dessus). Déplacer le curseur plus bas que la ligne l'agrandira simplement.
#F*fonte*#: Charge une fonte (*fonte* en est le nom de fichier lié), que vous pourrez ensuite l'utiliser à l'aide de la spécification de format ©F*x*, où x est le numéro de la fonte à utiliser.
#H*x*#: Rajoute un espacement vertical en-dessous du paragraphe. Cet espace sera rajouté au prochain retour à la ligne, et réinitialisé à zéro ensuite.
#P[*x*][,*y*]*image*#: Affiche une image (dont le nom de fichier est spécifié par le paramètre *image*). Vous pouvez spécifier les positions où vous voulez la placer, mais la position y sera toujours relative. Cependant, sachez que si vous laissez les positions par défaut, l'image sera intégrée au texte (de même que Habillage -> Aligné sur le texte sous Word), mais dans le cas contraire, l'image sera placée indépendamment du texte (de même que Habillage -> Derrière le texte sous Word).

## (>) Partie programmation

### (!) Fonctions graphiques

Là, c'est un peu plus compliqué. Déjà, la différence principale est que j'utilise le mode 4. Ce qui signifie qu'au lieu de travailler avec 4 bits par pixel comme précédemment, ici chaque pixel est codé sur 8 bits. C'est pourquoi les routines utilisées dans ce mode ont été revues, même si elles sont calquées et ressemblent fortement à celles utilisées pour le mode 0 (que nous avons vues précédemment).

Tout d'abord, dans ce mode, il n'est plus question de mosaïques. Les pixels sont alignés les uns à la suite des autres. Pour plus d'informations sur la représentation de l'écran, voyez le cours inclut au projet. La formule de calcul de l'adresse du motif

à modifier change donc, mais le mode d'accès reste sur 32 bits, pour les mêmes raisons qu'avant: la vitesse. Grâce à cela, on peut écrire 4 pixels d'un coup, ce qui n'est pas négligeable comparé à l'écriture pixel par pixel.

En mode 4, on sait que l'écran fait 240 pixels de large. Comme chaque pixel est codé sur 8 bits et que nous utilisons un pointeur 32 bits, il faut diviser la taille à ajouter par quatre car GCC la multiplie automatiquement de façon interne. Comme une valeur 32 bits couvre 4 pixels, il faut diviser la position en x par 4 pour obtenir l'offset réel horizontalement. La formule devient alors la suivante:
`Adresse=memVidéo+60*y+x/4;`
Ce qui donne, en remplaçant la division par un décalage:
`Adresse=memVidéo+60*y+(x>>2);`
GCC fera lui-même une optimisation et remplacera ce code par:
`Adresse=memVidéo+64*y-4*y+x/4`, ce qui donne le code `memVidéo+(y<<6)-(y<<2)+(x<<2)`, bien plus efficace pour le processeur, car les multiplications sont très lentes.

La dernière chose qui change est le calcul du décalage à appliquer. En mode 0, celui-ci était calculé sur 8 pixels (4 bits), alors que là, il n'est que sur 4 pixels (8 bits). La différence réside dans le fait qu'au lieu d'utiliser la formule suivante:
`Décalage=(x modulo 8) * 4`
Ce sera simplement:
`Décalage=(x modulo 4) * 8`
Donc `(x&3)<<3` avec les optimisations.

Ensuite, la copie des pixels se fait de la même manière qu'en mode 0, excepté qu'il faut deux fois plus d'opérations pour effectuer la même chose, et que le prochain motif à modifier est toujours contigu. Alors que dans le cas du mode 0, il fallait calculer de manière à arriver sur la prochaine mosaïque (à droite). L'addition par 8 (32 octets = une mosaïque) est remplacée par une addition par 1 (4 octets = 4 pixels).

## (!) Analyse du texte

Tout d'abord, avant d'afficher quoi que ce soit, on passe par une étape d'analyse du texte. Elle permet de déterminer toutes les informations nécessaires ensuite à l'affichage (positionnement des images, largeur de chaque ligne, obligatoire pour le centrage, etc.).
C'est la fonction AnalyseTexte qui s'occupe de cela. Elle alloue tout d'abord une table contenant des structures de type LIGNE. Ces structures sont utilisées pour stocker les informations relatives à chaque ligne à afficher. Voyons ci-dessous les éléments qui la composent:

```
typedef struct      {
    unsigned char *donnees;                 //Pointeur sur le texte
    int x;                                  //Marge en x
    int y;                                  //Position absolue y
    int l;                                  //Largeur
    char alignement;                        //Alignement
    FORMAT_TEXTE fmt;                       //Format
} LIGNE_TEXTE;
```

donnees: Pointeur sur le début du texte de la ligne.
x: Marge horizontale. Détermine en fait où le texte commence. Dans notre cas, la marge est toujours nulle car je n'ai pas intégré de commande permettant de la modifier.
y: Position verticale absolue. Indique l'endroit où il faudra dessiner le texte. Le calcul à effectuer pour connaître où afficher la ligne est assez simple: y-défilement. Les

lignes qui sortent complètement de l'écran ne sont pas affichées, et les autres sont "clippées", ce qui signifie que les données graphiques qui sortent de la plage de l'écran ne sont pas écrites.
l: Largeur de la ligne (en pixels, de même que tout le reste).
alignement: Indique l'alignement horizontal de la ligne. Cela peut valoir zéro (à gauche), un (centré) ou deux (à droite). Connaissant la largeur ainsi que la marge x (qui n'est pas utilisée), on peut trouver facilement l'endroit où l'on devra réellement commencer à dessiner le texte.
Pour l'alignement à gauche: 0.
Pour le centrage: milieu_écran-largeur/2
Pour l'alignement à droite: taille_écran-largeur.
fmt: Structure de type *FORMAT_TEXTE* contenant le format au début de la ligne. Celui-ci peut bien entendu être modifié en milieu de ligne.

Au début de chaque ligne, le format courant est stocké, pour permettre au lecteur de s'y retrouver lorsqu'il voudra afficher la ligne. Toutes les propriétés à stocker dans la ligne sont réinitialisées (la hauteur de celle-ci, la largeur, la position verticale), mais pas le format.

On entre ensuite dans la boucle principale, où chaque caractère est analysé.
Si des caractères spéciaux sont trouvés, alors ceux-ci sont analysés individuellement, avant que le cours des choses reprenne tranquillement, comme s'il ne s'était rien passé.

- S'il s'agit d'un caractère ©, on fait appel à GetFormat, qui permet de modifier le format courant en fonction des paramètres spécifiés dans le texte.
- S'il s'agit d'un caractère de fin de ligne, on y met fin prématurément, par un *goto stocke_ligne* (sans vouloir relancer la polémique sur les gotos, je trouve celui-ci parfaitement justifié, et bien plus clair qu'une triple boucle while imbriquée avec des variables de conditions douteuses à remplir).
- S'il s'agit d'un caractère de commande (#), celle-ci est gérée par un appel de AnalyseCommande. Ensuite, selon les paramètres retournés, il est aisé d'exécuter la commande. a*rgs[]* est un tableau qui contient tous les arguments de type entier fournis (utilisables directement), et *str* contient la chaîne fournie.
- Si c'est un caractère imprimable, on se contente de vérifier si celui-ci ne fait pas déborder la ligne en largeur. Dans ce cas-là, l'état du texte sera repris tel qu'il était au dernier espace, pour éviter de couper les mots en deux. La hauteur de la ligne est constamment calculée et mise à jour si elle est plus grande que la hauteur actuelle; car le but est de savoir quel est l'endroit où la taille de la ligne est la plus grande, pour pouvoir laisser un saut de ligne suffisant à la fin de celle-ci.

## (?) Affichage des lignes

L'affichage se comporte de manière assez identique à l'analyse, excepté qu'il n'a pas besoin d'effectuer certaines tâches qui ont déjà été faites par l'analyseur.
L'afficheur analyse lui aussi chaque caractère, à la recherche d'un éventuel code de format (oui, car le format peut être modifié en cours de ligne), d'une commande ou de la fin de la ligne.
La seule spécificité est la routine d'affichage des caractères. J'y fais appel par le biais d'une autre routine, qui pointe sur celle à appeler réellement (je sais c'est compliqué). routineDessCar peut donc pointer soit sur DessCar256 (qui dessine un caractère normal), soit sur DessCarItalique256 (qui dessine… devinez quoi… un caractère en italique!).

**(?) La boucle principale (LTMain)**

Cette fonction s'occupe de trouver le fichier texte (elle affiche un message d'erreur sinon), d'effectuer les initialisations, et d'appeler les bonnes fonctions. Il n'y a rien de particulier à cela.
Néanmoins, lorsqu'il faut mettre à jour une partie de l'affichage (lors d'un défilement vertical), c'est déjà une autre histoire. Voici donc les étapes à suivre dans ce cas:

- Calculer les positions de clipping, c'est à dire les coordonnées en dehors desquelles il est inutile d'écrire, soit parce que l'affichage est déjà à jour dans cette partie, soit parce que celle-ci se trouve en-dehors de l'écran.
- Afficher les éventuelles images, comme un fond d'écran. Elles sont dessinées derrière le texte, mais pour les placer à l'avant, il suffirait de le faire juste après avoir dessiné le texte.
- trouver la première ligne qui serait susceptible d'être affichée (histoire d'économiser un peu de boulot à ce pauvre processeur).
- Afficher chaque ligne depuis là, et continuer vers l'aval jusqu'à ce qu'on ait dessiné suffisamment de lignes (si la ligne à dessiner ne se trouve plus du tout sur l'écran, il est inutile de continuer!).
- Entrer dans une boucle gérant les touches, et faisant défiler l'écran à l'aide des fonctions DefileHaut et DefileBas, qui demandent en argument l'amplitude du défilement. Ceci paramètre également la variable *rafraichissement*, qui indique quelle partie de l'écran il faudra mettre à jour. Elle peut accueillir les valeurs suivantes: 0=haut de l'écran, 1=tout l'écran, 2=bas de l'écran.

**(>) Le programme convert.exe**

Pour créer de nouvelles fontes ou de nouvelles images bitmap à insérer dans vos textes, vous pouvez utiliser le programme *convert.exe*, qui se trouve dans le répertoire */Map/Fichiers/* du projet.
Vous n'avez qu'à le lancer et lire les instructions à l'écran.

**(?) Créer une fonte (police)**

Si vous choisissez de créer une fonte, vous aurez besoin des choses suivantes:

- Un fichier bitmap contenant les images de TOUS les caractères ASCII entre 32 (espace) et 255 (ÿ), soit 224 caractères placés verticalement les uns à la suite des autres.
- Une table contenant la taille (largeur) de chaque caractère. Si le fichier table (nom_de_la_bitmap.tbl) n'existe pas, le programme vous en créera une automatiquement à partir de ce qu'il voit. Mais je ne vous garantis pas que cela vous plaise, alors si ce n'est pas le cas, vous pouvez éditer le fichier (avec le bloc-notes) et relancer la conversion par après.

Par exemple, vous pourrez trouver dans le dossier */Map/Res/* du projet un fichier nommé arial.bmp. Vous pouvez dès lors essayer d'utiliser ce fichier pour créer une nouvelle fonte basée sur ce jeu de caractères. Si cette police ne vous plaît pas, il est toujours assez simple de se programmer une petite application juste pour l'occasion qui écrit tous les caractères d'une police Windows sur un fichier bitmap ainsi que la taille de ceux-ci dans un fichier table pour le programme.
**Important:** Bien que chaque pixel prenne un octet, les motifs utilisés sont toujours codés sur 32 bits. Ce qui signifie que la largeur de vos bitmaps sera toujours multiple de 4. Donc, si votre jeu de caractères fait 9 pixels de large (3 motifs 32 bits, puisque comme pour les puissances de deux, on arrondit à l'entier supérieur), il pourrait être judicieux de voir si vous ne pourriez pas le faire rentrer sur 8 pixels de largeur, car cela représenterait un gain de taille de 33%. Il en va de même avec les bitmaps.

Maintenant, pour pouvoir utiliser la fonte, il faut bien sûr la lier avec la map par le biais du système de fichiers. Démarrez pour cela le programme filesystem.exe (dans le même répertoire) et glisser-déposez le fichier depuis l'explorateur dans la fenêtre. Modifiez le nom de fichier de manière à ce qu'il devienne relatif au chemin actuel. Appuyez sur Entrée pour l'ajouter et enfin fermez le programme.
Si vous avez besoin d'un exemple, regardez arial.fnt qui est utilisée dans le fichier d'aide.
Ensuite, pour pouvoir l'utiliser, vous devez la charger (à l'intérieur de votre texte). Il y a pour cela la commande #F. Vous devez juste spécifier le nom de fichier (par exemple #Farial.fnt#). Il ne vous reste plus qu'à faire un appel de ©Fx où x est le numéro de la nouvelle fonte. Les fontes sont numérotées de 0 à 9. La ou les fontes par défaut utilisent les premiers numéros (à partir de zéro) et les fontes chargées dynamiquement occupent les prochains. Dans notre cas, comme il n'y a qu'une seule fonte par défaut (©F0), il faudra écrire ©F1 pour utiliser celle que l'on vient de charger, ©F2 pour la prochaine, et ainsi de suite.

### (?) Créer une bitmap

Vous pouvez aisément créer des images à insérer dans votre texte. A l'invite du programme, choisissez la deuxième option. Vous n'avez qu'à rentrer le nom du fichier (sans l'extension .bmp) et la conversion se fera. Il en résultera un fichier .b8 que vous pourrez lier avec la map de la même manière que décrit dans le chapitre "Créer une fonte".
Pour insérer l'image dans le texte, il faut utiliser la commande #P. Des arguments optionnels sont possibles, mais vous pouvez ne spécifier que le nom de l'image. Le texte sera automatiquement formaté pour intégrer au mieux l'image en question.
Exemple: #Ptest.b8#.

## 13) Les contrôleurs DMA

Attaquons un peu plus en détail le matériel de la Game Boy Advance.
Les contrôleurs DMA (Direct Memory Access) sont des éléments importants dans le système qu'est la GBA. Leur tâche principale est de copier très rapidement des données en mémoire. Du fait qu'il accèdent directement à la mémoire (comme leur nom l'indique), ils effectuent ces copies beaucoup plus rapidement que le microprocesseur, qui lui doit continuellement recevoir des instructions pour ce faire.

Mais ces contrôleurs ont plusieurs utilités, comme nous le verrons ci-dessous, puis également plus tard, en abordant le chapitre sur le son. De plus, si vous êtes déjà familier avec les fonctions de manipulation de mémoire provenant de la librairie standard C (memcpy, memmove, memset), vous ne devriez pas avoir trop de problème à comprendre ce chapitre.

### (!) Utilisation des contrôleurs DMA

Comme pour les autres éléments matériels, l'accès au contrôleur DMA se fait par l'intermédiaire des registres adéquats. Ces registres sont définis dans le fichier matériel.h du projet et sont les suivants:
REG_DMAxSAD: Adresse source de la copie (27 bits)
REG_DMAxDAD: Adresse de destination de la copie (27 bits)
REG_DMAxCNT: Registre de contrôle du DMA, dont les bits signifient ceci:

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Effet | Enable | IRQ | Timing | | ? | Taille | Repeat | Src cnt | | Dest cnt | | x | x | x | x | x |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Effet | x | x | Nombre d'objets à copier | | | | | | | | | | | | | |

0-13: Nombre d'éléments à copier; Le nombre d'octets qui seront réellement copiés dépend du mode de transfert (16 ou 32 bits).
21-22: Contrôle du compteur destination; Les valeurs suivantes sont possibles:
- 00: Incrémente l'adresse après chaque élément copié (par défaut)
- 01: Décrémente l'adresse après chaque élément copié.
- 10: Fixe l'adresse de destination
- 11: Je n'en ai pas trouvé l'utilité. Peut-être illégal.

23-24: Contrôle du compteur source; Les valeurs possibles sont identiques à celles du compteur de destination.
25: Répétition; Lorsque les timings 1 et 2 sont utilisés, le transfert est répété à chaque fois que l'intervalle est terminé. Utile pour copier une nouvelle valeur d'un tableau en mémoire vidéo à chaque ligne d'écran (HBLANK).
26: Taille; Si ce bit vaut 1, chaque élément copié est fait sur 32 bits (donc la taille en octets copiée est *nbreElements*4*) et sinon, chaque élément est copié sur 16 bits (donc la taille réellement copiée est *nbreElements*2*).
28-29: Timing; Permet de spécifier quand le transfert commencera. Les valeurs possibles sont les suivantes:
- 00: Démarre le transfert immédiatement
- 01: Démarre le transfert au prochain VBLANK (chaque *frame*)
- 10: Démarre le transfert au prochain HBLANK (chaque ligne de l'écran)
- 11: Dépend du canal DMA en question.
  - DMA 1 et 2: Copie un mot 32 bits lorsque le registre son (FIFO) est vide et a besoin d'être mis à jour. Pour plus d'informations, voyez le chapitre sur le son.
  - DMA 3: Copie au moment où la prochaine ligne d'écran commence à être dessinée.

30: IRQ; Déclenche une interruption lorsque le transfert est achevé. Personnellement, j'y trouve assez peu d'intérêt étant donné que le processeur est de toutes façons stoppé durant un transfert DMA.
31: Enable; Dès que ce bit passe à un, le transfert démarre en utilisant les paramètres définis. Il va de soi que vous devez vous assurer que ce bit est le dernier à être activé, sinon le transfert pourrait démarrer avec des réglages incomplets qui risqueraient de donner quelque chose de vraiment pas beau à voir (âmes sensibles s'abstenir!).

**Attention:** Notez que ces registres ne sont accessibles qu'en écriture! Cela ne change pas grand chose en temps normal, mais n'oubliez pas que si vous voulez rajouter un paramètre à l'aide d'un OU logique (notamment pour le registre de contrôle), ce ne sera pas possible (puisque cela nécessite une lecture) et votre GBA risquera de planter méchamment. C'est pourquoi il serait bon de songer à utiliser dans un premier temps une variable à laquelle vous ajoutez les paramètres avant de la copier complètement sur le registre DMA en question.

Et le code alors?

Yes sir. Pour lancer une copie DMA, il suffit simplement de remplir les registres avec les adresses source et destination, ainsi que paramétrer le registre de contrôle avec ce dont vous avez besoin avant de lui faire démarrer le transfert.
Voici un petit code qui lance un transfert simple:

```
    REG_DMA0SAD = (unsigned int) source;       //Adresse source
    REG_DMA0DAD = (unsigned int) dest;         //Adresse destination
    REG_DMA0CNT = nbval | DMA_ENABLE;          //Registre de contrôle
```

Dès que l'exécution de la troisième ligne de code est terminée, la copie DMA démarre.

Diverses routines de copie DMA sont définies dans le fichier routines.c. Elles sont nommées CopieDMA, MemMove16bits, MemSet et CopieDMA0. Comme elles sont commentées, je pense que ce ne sera pas trop un problème de les comprendre…

**(!) Memmove – déplacement de mémoire**

Pour copier des données d'un point à un autre de la mémoire, un appel simple au contrôleur DMA suffit. En revanche, lorsque les données peuvent se marcher dessus, il y a un problème. Imaginons le cas suivant:
Demande de l'utilisateur: copier 3 octets depuis l'adresse 2 vers la 3.
Avec un appel au DMA comme vu plus haut (correspondant à un memcpy), cela opérera séquentiellement les opérations suivantes:

| Opération | Adresse 2 | Adresse 3 | Adresse 4 | Adresse 5 |
|---|---|---|---|---|
| Départ | 5 | 18 | 143 | 76 |
| Copie de 2 en 3 | 5 | 5 | 143 | 76 |
| Copie de 3 en 4 | 5 | 5 | 5 | 76 |
| Copie de 4 en 5 | 5 | 5 | 5 | 5 |

Or, il y a manifestement un problème comme vous pouvez le voir. Mais ne désespérons pas; la solution est assez simple! Il suffit de commencer à copier depuis la fin. En effet, regardez plutôt ce que cela donne à l'envers:

| Opération | Adresse 2 | Adresse 3 | Adresse 4 | Adresse 5 |
|---|---|---|---|---|
| Départ | 5 | 18 | 143 | 76 |
| Copie de 4 en 5 | 5 | 18 | 143 | 143 |
| Copie de 3 en 4 | 5 | 18 | 18 | 143 |
| Copie de 2 en 3 | 5 | 5 | 18 | 143 |

Notre déplacement de mémoire d'est donc déroulé avec succès! Il reste donc à faire la même chose avec le contrôleur DMA si (et seulement si) l'adresse destination est supérieure à l'adresse source (essayez, dans le cas contraire, et vous remarquerez que c'est une copie normale qu'il faut réaliser sinon c'est le même problème que vous rencontrerez, mais tourné à l'envers).

Pour une implémentation de cette routine en utilisant le DMA, voyez la fonction MemMove16bits du fichier routines.c.

## 14) Les interruptions

Dans tout système informatique, on utilise les interruptions logicielles pour des tâches diverses et critiques. Parfois, leur utilisation est très fastidieuse, mais heureusement, sur ARM7TDMI, cela ne posera pas trop de problèmes.

On utilise pour cela quatre registres. Voici d'abord une description de ceux-ci:

REG_IME: Interrupt Master Enable

| Bit | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Effet | Non utilisés | | | | | | | | | | | | | | | Enable |

0; Enable: Active les interruptions. Elles sont désactivées par défaut.

Ensuite, dans les registres IE (Interrupt Enable) et IF (Interrupt Flags), les bits ont la même signification. Dans IE, ils déterminent quelles sources d'interruptions peuvent déclencher une IRQ, dans IF, ils déterminent quelle source est la cause de l'IRQ.
La procédure pour paramétrer les interruptions est donc la suivante:
-Ecrire sur IE les interruptions que l'on souhaite déclencher
-Lorsqu'une interruption est déclenchée, vérifier IF pour savoir quelle interruption on doit gérer.

REG_IE/REG_IF: Interrupt Enable/Flags

| **Bit** | F | E | D | C | B | A | 9 | 8 |
|---|---|---|---|---|---|---|---|---|
| **Effet** | Non utilisés | | Game pak | Touches | DMA3 | DMA2 | DMA1 | DMA0 |

| **Bit** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **Effet** | SIO | TM3 | TM2 | TM1 | TM0 | VCOUNT | HBLANK | VBLANK |

0; VBlank: Je pense que vous la connaissez depuis le temps, hm? Cette interruption signifie que le LCD vient d'entrer en période VBLANK. Cette interruption est déclenchée 60 fois par seconde (à chaque fois que le dessin de l'écran est terminé).
1; HBlank: Cette interruption est générée à chaque HBLANK, c'est-à-dire à chaque ligne de LCD.
2; VCount: Cette interruption est déclenchée lorsque le numéro de la ligne de LCD courant correspond à celle définie dans les bits 8-F du registre DISPSTAT.
3-6; Timer 0-3: Ces interruptions sont déclenchées lorsqu'il y a un débordement sur l'un des quatre timers (i.e. la valeur du décompteur est arrivée à zéro). Pour plus d'informations, voir le chapitre correspondant.
7; E/S sérielle: Je n'ai pas testé cette fonctionnalité, alors je ne peux pas la décrire. Je pense qu'elle doit être déclenchée lorsque des données sont arrivées.
8-B; DMA 0-3: Ces interruptions sont lancées lorsqu'un contrôleur DMA a terminé son travail et que le bit 30 (IRQ) était activé lors du paramétrage du transfert.
C; Touches: Je ne l'ai pas utilisé. Est peut-être lancé lorsqu'il y a un changement d'état du clavier? A tester.
D; Game Pak: Source d'interruption externe.

REG_INTERRUPT: Ce registre contient l'adresse du gestionnaire d'interruptions utilisateur (User IRQ Handler). Lorsqu'une interruption est générée, la main est laissée au BIOS qui saute à l'adresse contenue dans ce registre après avoir sauvé les registres nécessaires pour le retour. Cela permet d'utiliser une fonction C standard comme gestionnaire d'interruption.

## (?) Paramétrage des interruptions

Cette opération est assez simple, mais il faut tout de même faire attention à certaines choses, sinon votre GBA risque de vous donner l'impression d'être possédée.

Voici les différentes étapes à effectuer (en respectant l'ordre) ainsi que le code d'exemple associé.

1) Désactiver les interruptions. Si une interruption arrive pendant qu'on est en train de les paramétrer, ça fera mal:
```
REG_IME=0;
```
2) Après, il faut écrire sur IE les interruptions désirées. Ici, on veut VCOUNT (bit 2).
```
REG_IE = INT_VCOUNT;
```

3) Pour certaines interruptions, comme VBLANK, HBLANK, VCOUNT, il faut paramétrer DISPSTAT pour qu'il les génère (voir le chapitre sur le son, qui documente le registre DISPSTAT).

```
REG_DISPSTAT = (1<<5)|(50<<8);
```

4) Reste à définir l'adresse du gestionnaire d'interruptions (Interrupt handler)

```
REG_INTERRUPT=(u32)&InterruptHandler;
```

5) Finalement, on peut réactiver les interruptions!

```
REG_IME=1;
```

# *15) Le son*

**Caractéristiques techniques :**

Le système sonore de ce projet mixe une musique au format ADPCM (taux d'échantillonnage de 10512 Hz, 42 kbps) avec un son au format PCM (21024 Hz, 168 kbps). Rendez-vous dans le fichier son.c. Le code est commenté et de toutes manières n'est pas très difficile à comprendre.
Le fichier playad.c s'occupe de la décompression ADPCM selon l'algorithme IMA.
Wav28ad (par Damian Yerrick) convertit un fichier wav (mono) en ADPCM.
Wav2son (par moi) fait la même chose, mais le convertit en PCM utilisable par le mixeur.
Les fichiers sont convertis à l'aide du batch FaitSons.bat, et ajoutés au système de fichiers. Les fonctions JoueSon et JoueMusique s'occupent de jouer les bruitages et les musiques.

**(?) Comment fonctionne le système sonore?**

La Game Boy Advance dispose d'un chip sonore simple. On y trouve les générateurs de tonalités des anciennes Game Boy (qui furent utilisés dans la plupart des consoles des années 80) au nombre de quatre.
Mais le son sortant de ces générateurs de tonalités pouvant paraître particulièrement énervant, il y a également un contrôleur son appelé DirectSound qui vient remédier au problème. Il permet de jouer n'importe quel son, comme nous allons le voir plus bas.

Principe de fonctionnement

Je ne me suis pas intéressé à l'utilisation des générateurs de tonalités (faute de temps et d'intérêt réel), mais seulement au contrôleur DirectSound, qui, lui, permet de jouer du vrai et beau son.
Son fonctionnement est très proche de celui des autres machines "évoluées", comme le PC. Il ne génère pas de sons provenant d'instruments matériels contrairement aux formats tel le MIDI, mais s'occupe de jouer en sortie un signal PCM (*Pulse Code Modulation*), utilisé notamment dans les fichiers .wav sur PC.
Il y a deux voies DirectSound, qui peuvent être mélangées. Mais vous noterez que l'utilité réelle d'avoir deux voies est de reproduire du son stéréo. Eh oui, bien qu'il n'y ait qu'un haut parleur, la plupart des jeux offrent un support stéréo de haute qualité, mais il faut pour cela utiliser les écouteurs.

Choix d'un format

Puisqu'on va devoir générer du son en software, il va donc falloir s'inventer un format personnel sur mesure qui soit le plus efficace possible ou s'inspirer d'un format existant. Le choix de beaucoup de gens s'est porté sur le format MOD (aussi appelé "SoundTrack"), un format son qui fut d'abord créé pour l'amiga, mais qui est aujourd'hui encore assez sollicité dans la musique électronique amateur.

Ce type de fichier musical était particulièrement utilisé dans les jeux en raison de l'important gain de place obtenu en comparaison avec le son PCM brut.
L'idée est assez simple, et consiste à stocker en mémoire les instruments que l'on va utiliser sous la forme d'un échantillon sonore. Par la suite, on peut jouer ce même échantillon à toute la gamme en modifiant simplement la fréquence d'échantillonnage!

S'il faut retenir une chose, c'est que c'est un peu comme le MIDI, sauf que l'on peut définir ses propres échantillons ("instruments") facilement, ce qui permet d'élargir la banque son de base à moindre frais.

Mais je ne sais pas pourquoi je vous parle de ça puisque pour ce projet, je n'ai pas utilisé ce format, mais simplement joué du PCM pur. Ce n'est pas vraiment une bonne idée, en raison de la taille exorbitante des fichiers son à ce format-ci. Cependant, comme je n'ai ni les connaissances, ni le temps nécessaire pour implémenter un mixeur MOD, je m'en contenterai tout de même…

Implémentation matérielle

La GBA dispose de deux registres FIFO pour le son (un pour chaque voie). Ces registres (32 bits) peuvent contenir 4 échantillons sonores 8 bits. Ce registre devra être remis à jour très souvent avec les quatre prochains échantillons.
Heureusement, la GBA ne nous abandonne pas là! Il y a deux contrôleurs DMA spécialement prévus pour cette tâche. Ils proposent un mode de timing qui copie les données au prochain FIFO (First in first out). Cela couplé au bit de contrôle 'repeat' (qui répète indéfiniment l'opération) suffit à alimenter (indépendamment du processeur) le registre dès qu'il en a besoin. Faisons un résumé du paramétrage nécessaire du contrôleur DMA 1 pour obtenir ce qu'on veut:

DMA 1:
- Source = buffer son
- Destination = registre FIFO
- Compteur source = incrémentation (32 bits)
- Compteur destination = fixe (toujours sur le registre FIFO)
- Répéter à la fin de l'opération = Oui

Ce qui donne le code suivant du fichier son.c (fonction dsound_set_buffer):

```
DMA[1].src = src;                           //Source du son à copier
DMA[1].dst = (void *)0x040000a0;            //Adresse du FIFO A
DMA[1].count = 1;                           //Un mot 32 bits à la fois
DMA[1].control = DMA_DSTUNCH | DMA_SRCINC | DMA_REPEAT | DMA_U32 | DMA_SPECIAL
| DMA_ENABLE;                              //Lance la copie
```

La fonction init_sound du fichier son.c est assez compliquée, mais il n'est pas important que vous la compreniez. Il s'agit de l'initialisation du chip son, et les paramètres sont ceux par défaut.
Passons au paramétrage du timer 0 (dans cette même fonction). Le timer doit être déclenché à chaque fois qu'il faut réalimenter le registre FIFO. Il faut donc calculer une fréquence qui donne un nombre de cycles entier entre chaque intervalle, ainsi qu'une taille de buffer correspondante qui soit elle aussi entière.

**(!) Comment calculer les taux d'échantillonnage possibles**

Les explications ci-dessous sont inspirées de la page web suivante: http://www.pineight.com/gba/samplerates/ (anglais). J'en profite pour remercier l'auteur (Damian Yerrick) qui a également écrit le programme de compression ADPCM (wav28ad) que j'ai utilisé.

Sachant qu'on mettra à jour le buffer son une fois par *frame*, qui dure 280896 cycles de CPU, que la GBA lit 16 bits à la fois et que la fréquence du processeur est de 16777216 Hz, on définit les constantes suivantes qu'on utilisera plus bas:
*masterClock* = 16'777'216 (Hz)
*nEchantillons* = 16 (échantillons)
*tempsFrame* = 280896 (cycles)
**Note:** Bien que la GBA lise 16 bits à la fois, le format de son que l'on doit y placer est de 8 bits par échantillon.

Ensuite, on peut faire une liste des valeurs possibles de la manière suivante (en pseudo-code):

```
Pour période valant de 500 à 1800 (plage d'intérêt entre 10 et 32 kHz)
    TailleLecture = tempsFrame / période / nEchantillons
    Si TailleLecture est entier alors
        La période est déjà définie
        tailleBuffer = tempsFrame / période
        fréqEchantillonnage = masterClock / période
    Fin Si
Prochaine période
```

Reste à inscrire les différentes valeurs trouvées pour les trois variables (*période*, *tailleBuffer* et *fréqEchantillonnage*) dans le tableau ci-dessous:

| Période | Taille du buffer son | Taux d'échantillonnage |
|---|---|---|
| 532 cycles | 528 octets | 31536.12 Hz |
| 627 cycles | 448 octets | 26757.92 Hz |
| 798 cycles | 352 octets | 21024.08 Hz |
| 836 cycles | 336 octets | 20068.44 Hz |
| 924 cycles | 304 octets | 18157.16 Hz |
| 1254 cycles | 224 octets | 13378.96 Hz |
| 1463 cycles | 192 octets | 11467.68 Hz |
| 1596 cycles | 176 octets | 10512.04 Hz |

On dispose dès lors d'une tabelle représentant toutes les valeurs (malheureusement peu nombreuses) utilisables.
Si vous n'avez pas encore compris l'utilité de ce tableau, voici ce que représentent ces variables:
Période: Nombre de cycles CPU après lequel un nouvel échantillon doit être copié sur le FIFO. Evidemment, ce nombre doit être entier sinon on entendra des craquements dus au mauvais alignement causé par l'arrondissement.
Taille du buffer son: Nombre d'octets de son à remplir dans le buffer à chaque *frame* (VBLANK).
Taux d'échantillonnage: Représente la fréquence à laquelle le son sera joué. Lorsque vous convertissez une musique, assurez-vous que sa fréquence d'échantillonnage corresponde avec celle que vous avez choisie, sinon le son sera trop lent ou au contraire trop rapide.

**Note:** Vous pouvez modifier le taux d'échantillonnage d'un fichier son à l'aide du magnétophone Windows par exemple. Ouvrez le son (.wav) et allez dans le menu Fichier puis sous Propriétés. Choisissez "Convertir maintenant" et sélectionnez le taux désiré (tout en restant dans la catégorie PCM, les autres utilisant des codecs spéciaux).

**Remarque:** Plus la qualité du son est haute, plus le buffer son devra contenir de données à la fois, plus la période sera petite et plus la taille des sons sera grande. Ceci demande donc non seulement plus de mémoire, mais plus de données à copier dans une période plus courte, donc un ralentissement général du programme ainsi qu'un travail plus grand à chaque *frame*, lors de la décompression ADPCM par exemple.

## (!) La fonction dsound_vblank

Le buffer doit être mis à jour à chaque *frame* avec les nouveaux sons. En imaginant que vous voudriez jouer un son qui dit "Hello!" (comme dans la pub Fanta), on peut imaginer le découper ainsi:
*Frame 1*: He
*Frame 2*: ee
*Frame 3*: lloo
*Frame 4*: oo
*Frame 5*: oo!
Bien entendu, comme on utilise de la musique au format ADPCM, il serait idiot de décompresser toute la musique puis de commencer à la jouer sans plus s'en occuper, d'autant qu'un morceau décompressé prend bien déjà bien plus de RAM qu'il n'en est disponible au total! C'est pourquoi on le divise en plusieurs petits échantillons qu'on décompresse puis joue en temps voulu uniquement.
De plus, si un son est joué entre deux, il faut qu'il soit mixé avec le buffer assez rapidement. Si le buffer n'est par exemple mis à jour qu'une fois par seconde, il faudrait attendre jusqu'à la prochaine seconde pour entendre le nouveau son par-dessus (forcément, puisque la seconde précédente, on ne pouvait pas deviner que ce son allait être joué durant ce temps).

Une autre notion est celle d'utiliser deux buffer sons. Lorsque le buffer actif est en train d'être joué par la console, on ne doit surtout pas y toucher (sinon les résultats sont vraiment moches… je vous laisserai essayer si vous le souhaitez).
Donc, pour contourner cela, on utilise tout simplement deux buffers qu'on bascule simultanément entre eux. Cela fonctionne exactement de la même manière que pour l'affichage; on a un buffer qui est celui que l'utilisateur entend, et un autre qui est celui sur lequel on est en train d'écrire. Lorsque l'écriture est terminée, on peut basculer les deux buffers et recommencer à écrire sur l'autre (l'utilisateur entendra le contenu du buffer terminé uniquement).

La fonction dsound_vblank s'occupe donc de mettre à jour le buffer son avec les nouvelles données, en faisant appel à la fonction decode_ad (playad.c) qui décompresse une partie de la musique et la mixe avec un autre son. Puis, une fois fini, il échange les buffers et paramètre le DMA pour qu'il copie le nouveau buffer actif.

## (?) Astuce

Avez-vous remarqué le paramétrage un peu bizarre des interruptions effectué dans la fonction ModeSon? En fait, normalement, on est censé jouer le son durant la VBLANK. Mais comme je suis très limité par le temps disponible en VBLANK, j'ai trouvé une petite astuce pour contourner ce problème.

Pour le son, ce qui est important, c'est qu'il soit joué périodiquement, et toujours au même moment à l'intérieur de cette période. Dans mon cas, la période est chaque *frame*. Mais bon, cela ne spécifie pas *à quel moment* de la frame (pouvu que ce *moment* soit toujours le même).
C'est pourquoi j'ai décidé qu'au lieu de mettre le buffer à jour durant la VBLANK comme tout le monde, je le ferais lorsque le circuit graphique arriverait à la ligne 50 (c'est arbitraire). Le temps durant lequel le circuit graphique est en train de balayer l'écran peut être "gaspillé" sans trop de problèmes, étant donné qu'on ne peut de toutes façons pas accéder à l'écran pour les raisons mentionnées dans le chapitre *"Gestion des objets"*. Cependant, le temps de la VBLANK, lui, est précieux et (trop) limité. C'est pourquoi je ne vais pas l'encombrer encore plus avec le son.

Pour déclencher une interruption à la ligne 50, j'utilise le registre de statut de l'affichage, qui se décompose ainsi:

REG_DISPSTAT: Display status

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | Ligne (VCOUNT) | | | | | | | | x | x | VCNT IRQ | HBL IRQ | VBL IRQ | VCNT | HBL | VBL |

0; Flag VBlank: Ce bit est à un si la VBLANK est en cours. Cela signifie que le contrôleur graphique est arrivé au fond de l'écran et qu'il est en train de remonter au sommet. La VDRAW (affichage de l'écran) dure 160 lignes, alors que la VBLANK dure 68 lignes.
1; Flag HBlank: Ce bit est à un si la HBLANK est en cours. La HBLANK, c'est lorsqu'une ligne est finie d'être dessinée. Le contrôleur graphique se trouve tout à droite de la ligne et revient à gauche et se positionne pour dessiner la prochaine ligne. Voir le chapitre sur les effets spéciaux.
2; Flag Vcount: Ce bit est à un si la ligne de LCD en train d'être dessinée est celle indiquée dans les bits 8-F (VCOUNT).
3; IRQ VBlank: Ce bit permet de déclencher une interruption à l'entrée de la VBLANK. Pratiquement tous les jeux l'utilisent et attendent que cette interruption soit arrivée pour commencer à dessiner l'écran.
4; IRQ HBlank: Lance une interruption à chaque HBLANK. L'interruption arrive même durant la VBLANK (alors que rien n'est dessiné). Comme l'HBLANK ne dure que 228 cycles au maximum, le code du gestionnaire d'interruptions se doit d'être extrêmement rapide sinon votre processeur et/ou le contrôleur graphique se retrouveront paralysés et leurs performances chuteront.
5; IRQ VCount: Ceci lance une interruption lorsque le contrôleur graphique est arrivé à la ligne définie par les bits 8-F (VCOUNT). C'est ce que j'ai utilisé pour le son, en mettant une valeur de 50 dans les bits 8-F et en activant ce bit.
8-F; Réglage VCount: Ceci paramètre le numéro de ligne de LCD qui déclenchera une interruption si le bit 5 est activé.

Ce qui donne le code suivant pour activer une interruption à la ligne 50:
```
REG_DISPSTAT = (1<<5)|(50<<8);
```
Mais il faut aussi paramétrer IE (*Interrupt Enable*) pour qu'il accepte cette interruption (INT_VCOUNT):
```
REG_IE = INT_VCOUNT;
```
Et voilà, c'est fait! On peut maintenant faire appel à dsound_vblank (qui s'occupe de tout) à l'intérieur de notre gestionnaire d'interruptions (InterruptHandler).

## (?) Ajouter plusieurs voies sonores

Même si, pour ce projet, seule une voie n'est disponible en même temps pour les bruitages, il n'est vraiment pas dur d'en ajouter d'autres. Il suffit pour cela de les

additionner à l'intérieur du mixeur (c'est cette ligne qui effectue l'addition, fonction decode_ad dans le fichier playad.c) :

```
i=(last_sample>>8)+(*son2++);
```

On pourrait ajouter un nombre infini de sons:

```
i=(last_sample>>8)+(*son2++)+(*son3++)+(*son4++);
```

En veillant bien entendu à les gérer proprement dans la fonction dsound_vblank (i.e. de la même manière que le bruitage fourni de base).

Mais si vous voulez utiliser un système son à plusieurs voies, il faudra définir une certaine "hiérarchie" entre eux pour éviter de casser les oreilles (et les pieds) de l'utilisateur. Voici donc ce que je vous propose dans ce cas.
Il faut que chaque son spécifie de lui-même quelle voie il utilise. Si un son doit être joué sur une voie qui est déjà en train de jouer du son, celui-ci sera coupé est remplacé par le nouveau son. Pour exemple, prenons trois sons ainsi que les voies sur lesquelles on va les jouer:

- Prise d'une pièce Voie 1
- Saut du personnage Voie 2
- Ennemi détruit Voie 1

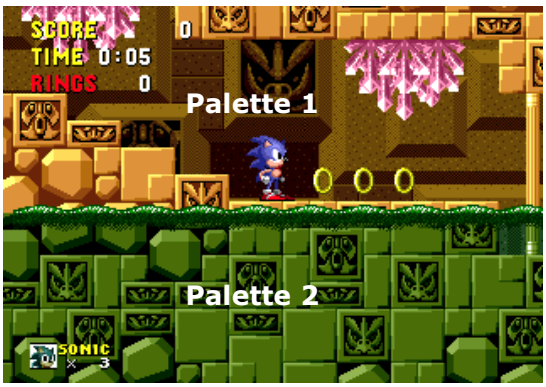
Lorsque deux pièces sont prises simultanément, leurs bruits sont remplacés et non ajoutés. Et c'est tant mieux! Car s'ils étaient ajoutés, il suffirait d'en prendre quatre ou cinq en même temps pour que cela vous explose les tympans.
De plus, comme le nombre de voies n'est pas infini, on peut placer certains sons qui ne devraient normalement pas (ou qu'on ne veut pas) être joués en même temps sur la même voie. Dans cet exemple, la destruction d'un ennemi ne devrait pas arriver en même temps que le saut, sinon on ne pourra entendre que le dernier des deux événements.

## 16) *Effets spéciaux*

Dans les plus anciennes consoles, il était assez fréquent d'utiliser des méthodes spéciales qui permettaient d'étendre les possibilités graphiques. Cela consiste à reprogrammer certaines données utilisées par le contrôleur graphique pendant qu'il balaye verticalement l'écran. Premièrement, ce fut utilisé pour la palette et le défilement. Nous allons passer en revue les effets les plus courants.

Reprogrammation de la palette

Un exemple typique est Sonic The Hedgehog (sur *Mega Drive*):




Vous remarquerez l'eau qui se trouve en-dessous du personnage. Comme la Mega Drive ne dispose pas de véritables effets de transparence, ce jeu utilise une astuce,

qui vise à reprogrammer la palette lorsque le contrôleur graphique arrive à la ligne correspondante au niveau de l'eau. Il la remplace par une autre, bleutée.
Le contrôleur graphique continue alors de dessiner l'écran (sans effacer ce qu'il a déjà fait) mais en utilisant cette fois la nouvelle palette.
Egalement utilisé pour les dégradés dans le ciel par exemple. On n'utilise en fait qu'une seule couleur de palette, mais elle est simplement modifiée à chaque ligne.

Reprogrammation du défilement

On peut également reprogrammer le défilement de l'écran durant le balayage vertical, ce qui permet de donner l'illusion que plusieurs objets situés à différents niveaux sur un plan bougent à une vitesse différente. C'est rarement utile en tant que tel sur GBA puisqu'elle offre un support matériel pour 4 plans qui peuvent défiler indépendamment. Cependant, sur la première Game Boy par exemple, seul un plan (et donc un seul défilement) n'était disponible. Néanmoins, rien n'empêchait de modifier le défilement de ce même plan durant le balayage vertical pour donner de jolis effets (mêmes si les possibilités sont beaucoup plus limitées que si on avait un support direct de plusieurs plans). Ci-dessous, on peut voir une image décomposée en quatre parties verticales (que j'ai encadrées), chacune utilisant un défilement différent (source: Wario Land 3, *Game Boy Color*):


Dans le même style, on peut également modifier le défilement vertical durant le balayage. En contrôlant les lignes qui sont affichées, et en en supprimant quelques-unes ou en affichant la même durant deux lignes ou plus de suite, on peut donner un effet d'agrandissement ou de rétrécissement.
Par exemple, si à la ligne 0, je donne un défilement de 0, le contrôleur graphique affichera la ligne 0 (0-0). Ensuite, à la ligne 1, il serait censé afficher la ligne 1 (1-0). Mais si à cette ligne, je définis un défilement de 1, après un petit calcul, le contrôleur graphique trouvera qu'il doit afficher la ligne 0 (1-1). Il affichera deux fois la même ligne (0), donnant un effet d'étirement de l'image.

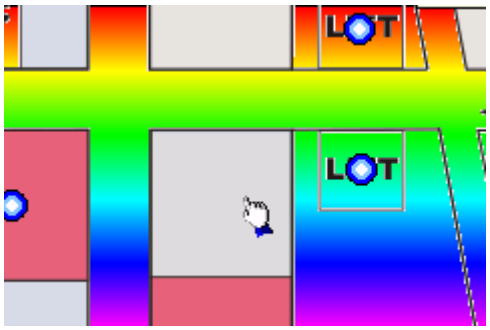
Reprogrammation de la rotation

Puis, sur *Super Nintendo* ainsi que sur *Game Boy Advance*, on peut reprogrammer les registres de rotation/zoom pendant le balayage vertical pour donner une illusion de 3 dimensions. C'était beaucoup utilisé pour les jeux de voiture (source de l'image: Mario Kart super circuit, *GBA*):


(Celui-là je le connais bien puisque c'était mon projet VB de l'année passée!)
Le principe est assez simple. On a un angle (il s'agit de celui de la caméra), et on calcule les valeurs de la rotation à appliquer à chaque nouvelle ligne de l'écran, en utilisant un coefficient de zoom différent. A l'horizon, on trouve donc un zoom infiniment petit, alors que tout près du personnage, il vaut 1 (ou une autre valeur qui détermine en fait l'élévation de la caméra par rapport au sol). Grâce aux nombres à virgule fixe utilisables pour notre *GBA*, ces effets sont bien mieux réussis que sur *Super Nintendo* (qui utilisait exclusivement des entiers).

## Implémentation

La GBA permet bien sûr d'utiliser les effets susmentionnés. Le matériel offre même quelques fonctionnalités supplémentaires qui aident beaucoup. Pour vous expliquer tout ça, je vais m'appuyer sur un exemple qui consistera à modifier la couleur de fond de l'écran à chaque ligne pour donner un dégradé vertical.



**Note:** Comme j'ai implémenté ce que vous voyez ci-dessus par-dessus la map du projet, j'ai créé un projet montrant cela, et épuré de toutes les choses inutiles.
Tout d'abord, comme vu au chapitre sur les interruptions, on va utiliser la période de la HBLANK (où le contrôleur LCD est inactif) pour effectuer notre travail. Pour ce faire, il faut d'abord paramétrer DISPSTAT pour qu'il génère une interruption à chaque HBLANK. Dans cet exemple, je laisserai tomber les paramètres déjà effectués (ceux de la VCOUNT notamment) pour me concentrer sur le sujet:

```
REG_DISPSTAT = 1<<4;
```

Sans oublier de paramétrer IE pour qu'il accepte les interruptions qui seront signalées par le contrôleur graphique.

```
REG_IE = INT_HBLANK;
```

Maintenant, c'en est terminé du paramétrage pur et dur. Il faut encore écrire un gestionnaire d'interruptions qui fasse ce dégradé. Mais je pense qu'il est inutile de vous préciser que si votre fonction gère les interruptions de type HBLANK, elle devra vraiment être rapide, puisqu'elle sera exécutée 228 fois par *frame* (elle est également appelée durant la VBLANK).

```
//Gestionnaire d'interruptions avec rotation de palette
void InterruptHandler(void)
{
    u32 Int_Flag,i;
    //Désactive les interruptions (on ne doit pas être interrompu pendant les
    //interruptions sinon en général, c'est le plantage garanti).
    REG_IME = 0x00;
    //Sauve le contenu de Interrupt Flag pour plus tard.
    Int_Flag = REG_IF;
    //Interruption de type HBLANK déclenchée?
    if (REG_IF & INT_HBLANK)            {
        i=R_VCNT;                         //La ligne en cours de dessin
        if (i<32)                         //Rouge -> jaune
            Palette[0]=RGB(31,i,0);
        else if (i<64)                    //Jaune -> vert
            Palette[0]=RGB(31-(i-32),31,0);
        else if (i<96)                    //Vert -> cyan
            Palette[0]=RGB(0,31,i-96);
        else if (i<128)                   //Cyan -> bleu
            Palette[0]=RGB(0,31-(i-96),31);
        else if (i<160)                   //Bleu -> magenta
            Palette[0]=RGB(i-128,0,31);
        else                              //Retour au rouge
            Palette[0]=RGB(31,0,0);
    }

    //Comme pour le PIC16F777A du cours uP1, on doit écrire sur IF pour
    //acquiescer l'interruption (sinon elle serait déclenchée en boucle).
    REG_IF = Int_Flag;
    //Réactive Interrupt Master Enable.
    REG_IME = 0x01;
}
```

Ainsi se termine ce chapitre. Même si les effets possibles sont toujours verticaux, cela permet déjà de faire pas mal de choses assez impressionnantes.

## 17) Les timers

Comme toute machine qui se respecte, la GBA offre des timers performants et paramétrables. Ces timers sont au nombre de quatre, et ils peuvent fonctionner indépendamment. Ils agissent en fait comme des compteurs; on écrit une valeur (16 bits) et on peut aller voir plus tard ce qu'elle est devenue. Il est également possible de lancer une interruption lorsque le compte dépasse la valeur maximale (0xFFFF) et revient à zéro. Cela permet de lancer des interruptions à des intervalles de temps très précis.

Voici la documentation des registres REG_TM0CNT à REG_TM3CNT:

REG_TMxCNT Timer x control

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | Pas utilisés | | | | | | | | Enable | IRQ | Pas utilisés | | | Up | Prescaler | |

0-1; Prescaler: Divise la fréquence du timer, de telle manière que les fréquences possibles sont les suivantes.

- 00: 16 MHz (non divisé)
- 01: 262 kHz (divisé par 64)
- 10: 65 kHz (divisé par 256)
- 11: 16 kHz (divisé par 1024)

2; Cascade: Place les timers en cascade. Lorsque le timer 0 a terminé sa période, il incrémente le timer 1, et ainsi de suite. Pour une utilisation normale, laissez ce bit à 0. A tester.
6; IRQ: Lance une interruption lorsque le compteur déborde (i.e. il revient à zéro).
7; Enable: Sélectionne l'état (marche/arrêt) du timer.

Pour lire/écrire la valeur actuelle du compteur, il y a le registre REG_TMxD (x entre 0 et 3), qui est sur 16 bits.

Fonctionnement

Lorsque vous activez le timer (bit 7), celui-ci va copier la valeur fournie dans le registre REG_TMxD sur le compteur. Dès lors, le timer commence à incrémenter le compteur à chaque fois que le délai est atteint.
Lorsque cette valeur atteint 0xFFFF, la valeur du registre (que vous avez écrite la dernière fois) est réécrite sur le compteur et le comptage recommence.
En activant l'IRQ, vous pourriez par exemple demander qu'une interruption soit lancée tous les 284 cycles de processeur (bon ok, là ça fait vraiment ristrette); pour cela, il faudrait sélectionner le prescaler 00 (16 MHz), activer le bit 6 (IRQ), mettre la valeur 0xffff-284 sur le compteur, et le lancer. Lorsque le compteur arrivera à 0xffff, il lancera une interruption puis rechargera le compteur avec la valeur 0xffff-284, et ce sera reparti pour un tour!

## (>) Programmation

Dans le projet, j'ai utilisé un timer pour mesurer l'utilisation du CPU. Le principe de base est simple; avant de commencer à travailler, je mets une valeur sur le timer, et une fois que j'ai terminé (avant d'attendre la VSync), je regarde quelle est la nouvelle valeur. Une soustraction et une adaptation à l'échelle du 100% me permet de connaître le temps réellement pris pour faire tout cela. Vous pouvez trouver cela dans le fichier routines.c, il s'agit des fonctions InitTimers(), UtilCPU et NouvelleTrame.

Initialisation des timers

A chaque fois qu'un cycle est terminé (une image affichée), il faut réinitialiser le timer.

Avant de commencer, il faut désactiver le timer, sinon celui-ci fera la sourde oreille et ne prendra pas les paramètres en compte.
```
REG_TM1CNT=0;
```
Puis il faut le remettre à zéro. On l'utilise comme un compteur, donc on pourra plus tard lire la nouvelle valeur.
```
REG_TM1D=0;
```
Maintenant que la valeur est correcte, on peut sélectionner la vitesse correcte. Personnellement, j'ai utilisé un prescaler de 256. La fréquence de 65 kHz générée va bien avec l'afficheur à 60 Hz. On peut dire que à peu près 1000 incréments représentent 100%; il suffit donc de diviser la valeur lue sur le timer par dix pour pouvoir l'afficher directement!
```
REG_TM1CNT=2;
```
On peut maintenant démarrer le timer! (en activant le bit 7 = enable)
```
REG_TM1CNT |= 1<<7;
```

Une fois cela terminé, on peut lire la valeur sur le timer sans restriction, en accédant simplement à `REG_TM1D`.

# 18) La sauvegarde

Eh oui, c'est un point assez important à prévoir, sinon le joueur frustré de perdre ses 10 heures de jeu risque de vous en vouloir à vie.

Tout d'abord, il faut savoir qu'il y a trois systèmes de sauvegarde. Passons-les brièvement en revue:

- SRAM
  - Celui que j'ai utilisé. Très simple, ce système utilise une mémoire RAM statique sauvegardée à l'aide d'une batterie au lithium. Ce système est utilisé depuis les premières Game Boy, et il est étonnant de constater que cela fonctionne encore aujourd'hui!
- Flash ROM
  - Ce système semblerait (encore) plus fiable que la SRAM. Il s'agit d'une mémoire reprogrammable (durée de vie indiquée: plus de 10'000 écritures par secteur). Mais la sauvegarde en devient extrêmement lente.
- EEPROM
  - Cette mémoire fonctionne différemment des autres, et je n'ai pas d'informations là-dessus. La durée de vie indiquée est de 100'000 écritures par secteur.

La gamme des tailles de mémoire de sauvegarde possible est de 8 ko à 128 ko (64 à 1024 kbits), mais dans les cartouches commerciales, elles ne peuvent en général dépasser 64 ko.

## (>) Ecriture en SRAM

Personnellement, j'ai utilisé la SRAM, qui est quand même un système rôdé depuis le temps. Cette mémoire est mappée à l'adresse 0E000000, cela signifie qu'on peut y accéder comme de la mémoire RAM standard en écrivant ou lisant à partir de cette adresse. La seule restriction est qu'elle utilise un bus 8 bits pour les données, ce qui signifie qu'on ne devra écrire qu'un octet à la fois. Pas d'accès avec d'autres types qu'un **char** donc.

Mais bon, on peut contourner ce problème en écrivant une fonction qui agit de la même manière que *fwrite* en C. On donne un pointeur sur les données ainsi que la taille à écrire et une boucle pour écrit octet par octet de donnée. C'est la fonction SRAM_Ecrit du fichier sram.c du projet. Et de la même manière, on peut relire les données depuis la SRAM (toujours par paquets de 8 bits), comme le fait la fonction SRAM_Lit. Pour expliquer le principe de ces deux fonctions, imaginons que vous voudriez copier le contenu d'un tableau vers un autre. Cela donnerait ceci:

```
char tab1[100];                    //Deux tableaux à ...
char tab2[100];                    //... copier entre eux
int idx;                           //Variable de boucle

for (idx=0;idx<100;idx++)
   tab2[idx] = tab1[idx];
```

Ce n'est pas trop compliqué. Maintenant, on veut copier une variable de type entier (int) vers une autre. Comme les types int sont codés sur 4 octets, on peut les représenter comme un tableau de 4 char (un char = 1 octet), vous suivez?

```
int var1, var2;                    //Nos deux variables à copier
char *tab1, *tab2;                 //Agissent comme des tableaux

//Maintenant, on va faire croire que var1 est un tableau de char en l'assignant
```

```
//à tab1, et de même pour var2 <-> tab2.
tab1 = &var1;
tab2 = &var2;

//On peut dès lors effectuer une copie de 4 octets de la même manière que si
//tab1 et tab2 étaient de vrais tableaux.
for (idx=0;idx<4;idx++)
   tab2[idx] = tab1[idx];
```

Bien, si vous avez compris ce principe, vous avez compris comment fonctionnent SRAM_Lit et SRAM_Ecrit, ou du moins vous seriez capable d'écrire ces routines vous-même.

## 19) Contrôle système

Je vous présente ici les différences entre les plages de mémoire disponibles sur la Game Boy Advance. Lorsqu'on débute, cela ne change pas grand chose, mais dès qu'on veut réaliser des applications importantes, il devient crucial de savoir paramétrer sa GBA comme il faut, car les paramètres par défaut des temps d'accès à la mémoire sont loin d'êtres optimaux, et vous donneront l'impression que la GBA est encore moins puissante que la TI-89, ce qui n'est pas peu dire.

**REG_WSCNT**

Encore du chinois, devez-vous vous dire. WSCNT signifie *Wait State Control*. C'est le registre qui s'occupe de paramétrer les temps d'attente (wait states) lors des accès au GamePak (ROM) et à la SRAM.

REG_WSCNT Wait State Control

| **Bit** | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Effet** | Type | Prefetch | x | ? | | WS2 2ème | WS2 1er | | WS1 2ème | WS1 1er | | WS0 2ème | WS0 1er | | SRAM | |

0-1; SRAM: Règle le temps d'accès à la SRAM. Les jeux utilisent toujours la valeur 11 (8 cycles), mais je ne sais pas pourquoi. Toujours est-il que nos cartes flash supportent en tous cas le réglage par défaut (00, 4 cycles) et peut-être même plus.
- 00: 4 cycles (par défaut)
- 01: 3 cycles
- 10: 2 cycles
- 11: 8 cycles (jeux commerciaux)

Cela signifie que si vous souhaitez lire ou écrire sur le port de la SRAM, le processeur attendra 4, 3, 2 ou 8 cycles pour être sûr que l'information soit bien lue ou écrite. Il va de soi que si vous n'attendez pas assez, l'information retournée lors d'une lecture sera erronée puisque la RAM n'aura pas terminé son travail que vous lirez déjà l'information. Au mieux, c'est la dernière information que vous recevrez, au pire ce sera n'importe quoi.

2-3; ROM (WS0) premier accès: Règle le temps d'accès au GamePak effectué entre l'adresse 08000000 et 09FFFFFF (par défaut). La ROM est "miroitée" aux adresses 0A000000 et 0C000000 mais les accès dans ces plages utilisent des waitstates différents (WS0 pour 08, WS1 pour 0A et WS2 pour 0C). C'est utile dans le cas de GamePaks composés de plusieurs chips ROM ayant des vitesses d'accès différentes. Les paramètres (cycles d'attente) sont les mêmes que pour la SRAM et la remarque est valable ici aussi, sauf que la vitesse n'est pas adaptée au GamePak, le résultat est en général nettement plus marqué car le code y est entièrement exécuté.
- 00: 4 cycles (par défaut)

01: 3 cycles (jeux commerciaux)
10: 2 cycles
11: 8 cycles

4; WS0 deuxième accès: L'accès à la ROM est plus rapide lors de deux accès 16 bits séquentiels. Vous pouvez paramétrer le temps demandé ici. Mais seules deux vitesses sont utilisables; si vous souhaitez des accès plus lents, utilisez les waitstates 1 et 2.

00: Dépend du waitstate: 2 cycles (WS0), 4 cycles (WS1) ou 8 cycles (WS2) (par défaut)
01: 1 cycle (jeux commerciaux)

5-A; Idem, mais pour les autres bancs de ROM (waitstates 1 et 2).
E; Prefetch: J'en parlerai plus tard.
F; Type de cartouche: 0 s'il s'agit d'un GamePak Game Boy Advance (AGB), 1 s'il s'agit d'un GamePak Game Boy / Game Boy Color (DMG/GBC). En lecture seule.

Le bus des GamePak est accessible par 8 ou 16 bits. Les accès 32 bits sont divisés en deux accès 16 bits, donc lents. Mais comme expliqué plus haut, le deuxième accès 16 bits séquenciel est plus rapide que le premier.

Finalement, le temps réel d'accès est défini par 1 cycle + le nombre de waitstates. Par défaut, cela donne 4+1 cycles (1$^{er}$ accès) et 2+1 cycles (2$^{ème}$ accès), soit 8 cycles pour un accès 32 bits, donc une instruction ARM. Mais que c'est lent! Eh oui, il ne faut pas s'attendre à ce que la ROM soit rapide, mais elle peut quand même faire un peu mieux que cela.
Si on prend le réglage utilisé par les jeux commerciaux, cela donne 3+1 cycles (1$^{er}$ accès) et 1+1 cycles (2$^{ème}$ accès), soit 6 cycles. Avec une carte EZFA, on peut utiliser le réglage 10 (2 cycles) au premier accès, soit 5 cycles par accès 32 bits. Mais comme c'est la seule carte à ma connaissance à supporter ce réglage, j'en déconseille fortement l'utilisation.
Ce qui donne: 6 cycles de lecture d'une instruction + 1/2 cycle(s) pour l'exécuter… soit une puissance d'environ 2.6 MIPS (Million d'Instructions Par Seconde), contre 3 MIPS pour la TI-89… ça n'est toujours pas très glorieux tout de même.

Mais ne désespérez point. Il y a d'une part le prefetch, et d'une autre l'IWRAM qui viennent à la rescousse!

Le prefetch

"Ca y est, il recommence à parler en chinois!", vous direz-vous.
Les GamePak disposent d'un système qui lit les données avant d'en avoir réellement besoin. On appelle cela le prefetch. Si vous accédez à l'adresse x, cet accès sera très lent. Mais le GamePak se préparera déjà et commencera directement à lire l'adresse x+1 sans que le processeur ne le lui demande, histoire de rendre le prochain accès plus rapide s'il est effectué séquentiellement, ce qui est généralement le cas.
Cela améliore déjà pas mal la vitesse d'exécution (de l'ordre d'à peu près 1.5 fois, voire plus, dépendant de votre code).

L'IWRAM

Mais bon la chose qui vous fera vraiment gagner de la vitesse, c'est bien l'IWRAM. Ces initiales signifient en fait Internal Work RAM (mémoire de travail interne). Cette petite mémoire de 32 ko est connectée au processeur par un bus 32 bits, et les accès sont effectués en 1 cycle! Et la bonne nouvelle, c'est qu'on peut aussi y placer du code, pour peu qu'il ne soit pas trop volumineux. Mais cette mémoire est également utilisée pour diverses choses, comme la pile (stack), et toutes les variables et tableaux locaux. Il est donc facile de la remplir avec des éléments dont on n'a pas besoin. Si on soustrait les utilisations "système" de l'IWRAM, il reste

environ 24 ko de libre pour y mettre ce qu'on veut (du code ou des variables). Si je vous mets ceci:

```
int tableau[1000];
```

Vous devriez vous exclamer "Malheureux! Tu *gaspilles* 4000 octets d'IWRAM!". Quatre mille octets d'IWRAM, cela représente mille instructions ARM qui pourraient être exécutées à une vitesse de 16 MIPS*! C'est pourquoi il y a l'EWRAM.
* Utopique. Il est impossible de faire une boucle, même la plus simple, sans dépenser une bonne dizaine de cycles par itération…

L'EWRAM

Parfois, on ne peut se passer de très gros tableaux, et les 32 ko de mémoire fournis ne sont pas suffisants (ou alors on aimerait les utiliser pour autre chose). Pas de souci, l'EWRAM est là. Ces initiales signifient External Work RAM. Comme cette mémoire est externe et connectée sur un bus 16 bits, l'accès y est très lent (à peu près identique à celui de la ROM, voire même un peu plus lent). J'ai fait une tabelle décrivant les temps d'accès aux divers composants (les temps sont ceux par défaut), dans laquelle vous pourrez trouver la vitesse (pitoyable) de l'EWRAM. Elle présente le nom du composant, la taille de la mémoire allouée, le bus (8, 16 ou 32 bits), et les temps d'accès (en cycles) suivant le nombre de données à accéder.

| Composant | Taille | Bus | 8 bits | 16 bits | 32 bits |
|---|---|---|---|---|---|
| BIOS | ? | 32 | 1 | 1 | 1 |
| IWRAM | 32 ko | 32 | 1 | 1 | 1 |
| I/O (paramètres) | ? | 32 | 1 | 1 | 1 |
| OAM (sprites) | 1 ko | 32 | 1 | 1 | 1 |
| EWRAM | 256 ko | 16 | 4 | 4 | 8 |
| Palette RAM | 1 ko | 16 | 1 | 1 | 2 |
| VRAM | 96 ko | 16 | 1 | 1 | 2 |
| ROM | 16 Mo | 16 | 5 | 5 | 8 |
| SRAM | 64 ko | 8 | 5 | - | - |

Le compilateur s'occupera lui-même de stocker les données au bon endroit. Vous n'avez qu'à lui spécifier la section dans laquelle vous voulez que vos données se trouvent et le reste se fera automatiquement. C'est pourquoi j'ai défini ces deux #define pour permettre de spécifier l'endroit où vous voulez stocker vos variables. Par défaut (si rien n'est spécifié), les variables sont stockées en IWRAM, et le code en ROM.

```
//Spécification des sections pour placer le code et les données.
#define IN_IWRAM __attribute__ ((section (".iwram")))
#define IN_EWRAM __attribute__ ((section (".ewram")))
```

Ainsi, il suffit d'écrire ceci:

```
int IN_EWRAM tableau[1000];
```

Et le tableau ne gaspillera plus les 4 ko d'IWRAM. L'accès au tableau sera plus lent, mais cette perte de vitesse sera largement compensée par le code que vous mettrez en IWRAM de par le gain occasionné. En effet, ce qui importe le plus est le temps d'exécution des instructions. L'accès à la mémoire est malgré tout plutôt rare. Il faudra peut-être trois ou quatre instructions ARM avant d'accéder à un élément du tableau ci-dessus. L'accès à cet élément sera ralenti, mais c'est mieux ça que les quatre instructions soient ralenties…

**Premier cas:** Code en ROM (lente), tableau en IWRAM (rapide)
Si chaque instruction demande un accès de 8 cycles et qu'il en faut 1 pour accéder au tableau:

```
@ Ce code effectue tableau[r3]=0x23;
ldr     r0, =tableau        @ r0=tableau          8+8? cycles (pool)
mov     r1, #0x23           @ r1=0x23             8 cycles
```

```
add     r0, r0, r3          @ r0+=r3                    8 cycles
str     r1, [r0]            @ *(tableau+r3)=r1          8+1 cycles
```
Soit au total: 41 cycles.

Deuxième cas: Code en IWRAM (rapide), tableau en EWRAM (lente)
```
@ Ce code effectue tableau[r3]=0x23;
ldr     r0, =tableau        @ r0=tableau                1+1? cycles (pool)
mov     r1, #0x23           @ r1=0x23                   1 cycles
add     r0, r0, r3          @ r0+=r3                    1 cycles
str     r1, [r0]            @ *(tableau+r3)=r1          1+8 cycles
```
Soit au total 13 cycles. Voilà la preuve qu'il est tout de même plus avantageux de mettre son code en IWRAM au détriment des variables.

**Note:** Je ne suis pas sûr des timings (je dis ça de tête), ils ne sont certainement pas exacts. Mais au fond, la différence reste évidente. De plus, ce code n'est peut-être pas optimal, mes connaissances en ASM étant vraiment limitées car je n'en ai fait qu'un petit peu au début de ce projet. Vous pouvez en trouver les résultats dans le répertoire du projet /Exemples/Assembleur/.

## Mise en garde!

Cependant, il reste une chose à laquelle faire très attention. Il n'y a aucun moyen de vérifier un débordement de l'IWRAM (i.e. si vous avez utilisé plus de 32 ko de données). Mais si c'est le cas, votre programme plantera dans des conditions très bizarres, ou des bugs étranges surgiront, laissant penser à une petite faute dans votre code, que vous ne trouverez évidemment pas (à moins que c'en soit réellement une).
Dans ces cas là, pour en avoir le cœur net, je vous suggère de créer un gros tableau qui sera placé en IWRAM.
```
int tableau[1000];
```
A ce moment, si le problème venait de là, votre programme devrait planter complètement. Si, après l'ajout du tableau, son état ne s'aggrave pas, alors c'est que le problème vient de vous.
Pour éviter ce genre de mésaventures, faites attention aux points suivants, et tout ira bien:

- Ne stockez jamais de gros tableaux (disons… plus gros qu'un kilooctet) en IWRAM, mais utilisez l'EWRAM à la place.
- Ne sous-estimez pas la taille du code. Une fonction, même "petite", peut aisément prendre plusieurs kilooctets.
- Ne placez pas n'importe quelle fonction en IWRAM, mais seulement les plus importantes. Si le gain de performances engendré n'est que de 1 ou 2% de CPU, c'est complètement inutile, et vous devriez déterminer quelles sont vraiment les fonctions les plus lentes.

## Exécution de fonctions en IWRAM

Si, après avoir placé une fonction en IWRAM, vous voyez l'erreur suivante lors de l'édition des liens:
xxx.o(.iwram+0x????): In function `yyy:
: relocation truncated to fit: R_ARM_PC24 zzz
C'est que vous avez mixé des fonctions en IWRAM et en ROM dans le même fichier source. Et le linkeur n'aime pas ça du tout. Toutes les fonctions situées en IWRAM devraient se trouver dans un fichier source séparé des autres.
Si vous ne savez pas comment utiliser plusieurs fichiers source simultanément (c'est vrai que c'est plutôt mal fait en C sur ce coup là)., jetez un coup d'œil dans le répertoire du projet /Exemple/IWRAM/.

Il existe cependant une autre possibilité pour mixer les types de fonctions au sein d'un même fichier source. Mais c'est assez sale comme méthode. J'ai écrit les macros suivantes:

```
void (*pRoutineLointaine)();
#define Exec(a,b...) ({ pRoutineLointaine = (void*)(a); pRoutineLointaine(b);
})

u32 (*pFctLointaine)();
#define ExecFct(a,b...) ({ pFctLointaine = (void*)(a); pFctLointaine(b); })
```

En incluant ce code à votre projet, vous pourrez faire appel à des fonctions placées dans une autre section que celle où vous vous trouvez actuellement. Au lieu d'appeler par exemple:
`SetMode(x,y);`
Vous feriez:
`Exec(SetMode,x,y);`
Et si la fonction doit retourner une valeur, utilisez ExecFct. Cependant la valeur retournée sera toujours du type défini par `u32 (*pFctLointaine)();` soit l'équivalent d'un entier non signé.

## REG_WSRAM

C'est un registre non documenté que j'ai nommé ainsi. Son adresse est 04000800. Ce registre règle la vitesse de l'EWRAM. Cela permet de gagner un petit peu de vitesse mais c'est déconseillé car pas officiellement supporté.

Je ne connais pas grand chose à ce registre, si ce n'est qu'il est sur 32 bits et que le bit 5 doit toujours être à 1 sinon la GBA plante. Le nombre de waitstates à appliquer à la RAM est situé entre les bits 24 et 27. Pour modifier cette valeur, cela donne donc le code suivant (20 pour garder le bit 5 activé):
REG_WSRAM = 0x0X000020;
Où X peut valoir entre 0 et 14 (15 fait planter la GBA). Le nombre de waitstates appliqué est 15-X. Ce qui signifie que l'accès le plus rapide serait de 2 cycles (1 waitstate + 1 cycle d'accès), et 4 cycles en mode 32 bits.

Notre GBA supporte le réglage maximum (0x0E000020), mais comme la compatibilité future n'est pas garantie et que de toutes manières je n'utilise que très peu la RAM, je ne l'ai pas appliqué dans le projet.

# *20) Annexe – sources*

Images tirés de jeux non mentionnés
*1: Sword of mana, *GBA*
*2: Super Mario Advance 3: Yoshi's island, *GBA*
*3: Pokemon Crystal, *GBC*

Infos sur les registres
Parfois adapté et traduit (très) librement du document GBATek.
http://www.work.de/nocash/gbatek.htm

Bande son utilisée du programme Map
**Note:** Cette bande son est juste là à titre de divertissement et ne constitue une part du projet que dans la mesure où elle est liée avec le chapitre traitant du son, mais n'engage en aucun cas la nature du projet. Si son utilisation ici (ainsi que celle de tout autre élément du programme final) venait à vous déranger, prière de me le faire savoir et je retirerai la partie concernée.
Musiques "Palais de glace" et "Mines de diamants", tirés du jeu Donkey Kong Country sur *Super Nintendo*, © 1995 Nintendo & RARE.
Musique "Sonic - Star light", tirée du jeu Sonic the hedgehog sur *Mega Drive*. © 1991 Sonic Team, licensé par Sega.
Bruitages partiellement issus de Golden Sun sur *Game Boy Advance*, © 2001 – 2003 Nintendo / Camelot.
Bruits d'erreur et exclamation issus de Microsoft Windows XP, © 2001 Microsoft corporation.

Contact
Si vous souhaitez me contacter pour une raison ou une autre, n'hésitez pas à utiliser l'adresse suivante:
f_bron@hotmail.com