PROJET GÉNIE LOGICIEL

2023/2024

Documentation de l'Extension GameBoy du Compilateur Deca

Antoine Borget
Pierre Lin
Anass Haydar
Yassine Safraoui
Jihane Zemmoura

Groupe GL-47

Table des matières

1 Introduction

La série de consoles Nintendo GameBoy a certainement été l'une des séries de consoles de jeux les plus réussies et a marqué le début de ce qui est aujourd'hui connu sous le nom de jeux vidéo. Bien que cela fasse 35 ans depuis sa sortie, des jeux continuent d'être développés pour elle, même si leur taille n'est pas aussi importante que le développement d'autres jeux. L'une des principales façons de développer des jeux pour la GameBoy est d'utiliser le langage C et la bibliothèque GBDK écrite en C. Cependant, cela n'est pas une expérience conviviale pour les développeurs, car le développement en C tend à être difficile et fastidieux en raison de sa nature bas niveau. Une autre option est d'écrire du code en langage assembleur qui est compatible avec le langage assembleur du GameBoy. RGBDS est un compilateur qui a rendu cela possible et a essayé d'ajouter des fonctionnalités syntaxiques à son langage assembleur pour aider les développeurs à écrire le code assembleur plus facilement. Néanmoins, écrire en assembleur RGBDS n'est toujours pas l'expérience la plus conviviale pour les développeurs, surtout lorsqu'il s'agit de la création des jeux de taille relativement importante. Le but de cette extension est d'utiliser la puissance et la nature pratique de la programmation orientée objet du langage de haut niveau Deca afin de faciliter la création des jeux GameBoy.

2 Analyse bibliographique

Le premier défi que nous avons dû affronter avec le développement de l'extension était de comprendre son architecture et aussi l'assembleur RGBASM qu'on vise a générer avec notre compilateur. Nous avons donc essayé de tout d'abord comprendre comment se développent les jeux GameBoy en RGBASM. Ce site [1] nous a beaucoup servi pour comprendre cela, c'est un guide très compréhensif qui nous a beaucoup inspiré pour savoir ce qu'il faut faire pour l'extension. Il y a aussi ces ressources supplémentaires :

- Il y a ce site pour comprendre comment la mémoire est organisé et à quoi sert chaque zone de la mémoire [2], même si le site précèdent était déjà assez suffisant.
- Il y a ce site [3] pour comprendre le fonctionnement du mécanisme d'affichage des tiles et des sprites.
- Il y a ce site [4] pour comprendre comment la GameBoy mets à jour l'écran puisque c'est ce qui détermine comment on doit changer les tiles de l'écran.
- Il y a aussi cette playlist [5] qui donne des explications plus visuelles et intuitives.

1. gbdev.io
2. GameBoy CPU Manual
3. Fonctionnement des tilemaps et des sprites
4. Draw and blank Periods
5. Playlist GameBoy Graphics explained

3 Choix de conception et d'algorithmes

3.1 Génération de code pour la GameBoy

Comme la GameBoy est une machine très ancienne (1989), la génération de code pour la GameBoy a été une partie très difficile de cette extension.

Au début, nous pensions que ce serait de simples traductions d'instructions assembleurs, donc, nous avons commencé à modifier les fonctions **codeGenX** en changeant quelques lignes pour les adapter à la GameBoy. Cependant, en plus de créer des bugs sur la génération du code pour IMA, nous nous sommes rendu compte des grosses différences entre l'architecture de la GameBoy et l'architecture de la machine IMA, et que cela n'allait donc pas être aussi simple.

Nous avons alors simplement décidé de recommencer toute cette étape de génération de code pour la GameBoy en créant à chaque fois une fonction **codeGenXGb** à coté des anciennes fonctions **codeGenX**. Nous avons procédé de la même manière que pour la génération de code pour IMA, c'est-à-dire de manière incrémentale.

3.1.1 Langage "Hello World"

Pour tester le code généré, il est très important de pouvoir afficher quelque chose.

Cependant, la GameBoy n'ayant pas de terminal où afficher du texte, nous avons décidé que cette étape consisterait en l'affichage de quelque chose sur l'écran. En suivant ce tutoriel très simple, nous avons pu afficher "Hello World" sur notre émulateur GameBoy très rapidement.

Nous avons alors décidé que la fonction *println* afficherait cette image "Hello World" quels que soit ses arguments. Le code généré pour afficher ce "Hello World" est disponible dans le fichier *Println.java*.

Comme décrit dans la partie 4.1.1, cela nous a suffit pour pouvoir tester le code généré. Nous n'avons donc pas implémenté les autres fonctions d'affichages (*print*, *printx*, et *printlnx*) à ce stade. Cependant, nous allons voir plus tard que ces autres fonctions d'affichages vont être utiles pour notre librairie GameBoy.decah.

La fonction *println* est surtout destinée aux programmeurs du compilateur, pour pouvoir tester et déboguer le code généré. Bien que utilisable par un utilisateur du compilateur, cette fonction ne leur est pas destiné.

3.1.2 Langage sans Objet

Introduction à l'Assembleur GameBoy

Toutes les instructions assembleurs GameBoy sont disponibles sur ce site.

Pour bien comprendre cette partie et les suivantes, il faut savoir que la GameBoy possède en tout 7 registres de 8-bits utilisables : A, B, C, D, E, H, et L.

De plus, il y a un autre registre F, qui est utilisé pour stocker les flags (comme en IMA). Ici, les deux flags les plus importants sont Z et C (à ne pas confondre avec le registre C qui a le même nom...). Z est le flag d'égalité équivalent au EQ en IMA, et, C est le flag d'infériorité ou égalité équivalent au flag LE en IMA. On peut avoir l'inverse de ces flags en écrivant !Z et !C.

Enfin, on peut utiliser les registres cités comme des registres 16-bits en les fusionnant, c'est-à-dire en écrivant : BC, DE, ou HL. Ceci est utile pour les opérations manipulant des adresses sur GameBoy, car ces adresses sont codées sur 16-bits.

Spécificités de l'Assembleur GameBoy

Le registre A est spécial, c'est le seul avec qui on va pouvoir faire le plupart des opérations arithmétiques sur 8-bits.

De même, le registre 16-bits HL est spécial, c'est le seul avec qui on va pouvoir faire la plupart des opérations manipulant des adresses.

Comme A et HL sont spéciaux, ils seront utilisés comme intermédiaire dans beaucoup d'opérations, ce seront donc des registres scratchs. Il nous reste donc 4 registres avec lesquels travailler : B, C, D, et E. Cependant, par souci de simplicité, une adresse étant codée sur 16-bits, nous avons décidé de travailler directement avec BC et DE (vu qu'une instance d'un objet est une adresse) même si nous étions encore sur le langage sans objet.

En GameBoy, il n'y a qu'un seul registre pointant sur la pile, c'est le registre SP.

En conclusion, pour faire une sorte de comparaison à IMA, on peut dire que :
- Les registres A et HL sont équivalents à R0 et R1.
- BC et DE sont équivalents aux registres R2, ..., R15 de IMA.
- SP fera office de ce que représente SP, GB, et LB en IMA, tout en même temps...

Comparaisons & Structures Conditionnelles

Comme décrit dans la partie 4.1.1, il était primordial d'avoir les comparaisons et le *if/else* fonctionnels en premier. Heureusement, cela n'a pas été trop difficile. En effet, nous avons pu réutiliser la majorité du code faisant l'évaluation paresseuses des expressions booléennes que nous avons écrit pour IMA.

En effet, pour faire une comparaison, il faut utiliser l'instruction **Cp A, r8** qui place les flags Z et C, où *r8* est un registre 8-bits. Z est placé si *r8* $=$ A, et C est placé si *r8* $>$ A. Comme il faut absolument passer par le registre A pour faire la comparaison, on doit mettre la "valeur basse" de l'autre registre dans A via **Ld A, r8** avant de faire la comparaison. La "valeur basse" des registres 16-bits avec lesquels on travaille est le sous-registre low 8-bits associé, par exemple, pour le registre DE, c'est le registre E. Il y a la même chose pour le sous-registre high.

Enfin, la gestion des branchements se fait grâce à l'instruction **Jp cc, Label** où *cc* est un flag. Le test d'égalité et de non-égalité se fait de la même manière qu'en IMA en utilisant Z et !Z. De même pour l'inférieur ou égal et la supériorité strict en utilisant C et !C. Pour le test de supérieur ou égal et d'inférieur strict, il suffit d'ajouter 1 à *r8* et d'utiliser C et !C.

Le codage des évaluations paresseuses des expressions booléennes, des *if/else*, et des *while* est exactement la même qu'en IMA.

Déclaration de Variables & Assignations

Comme il n'y a pas de valeur flottante dans l'assembleur GameBoy, on ne parlera que de *int* et *boolean* pour les types de base. Néanmoins, les *float* devrait fonctionner de la même manière que les *int* en Deca, même si ça n'a pas été testé. De plus, comme toutes les opérations arithmétiques de la GameBoy se font sur des registres 8-bits, les *int* et *boolean* seront tous codés sur 8-bits allant de 0 à 255 (pas de valeur négative en GameBoy).

Les variables globales sont stockés à partir du sommet de la pile (à l'adresse 0xDFFF) vers le bas jusqu'à la tête (0xC000). En effet, on part dans le sens inversé, car en GameBoy, une instruction **Push r16** décrémente SP. De plus, comme vous pouvez le voir on ne peut que *push* des registres 16-bits, raison de plus qui nous à pousser à travailler directement avec les registres 16-bits. Cette instruction **Push r16** va :
- Décrémenter SP de 1.
- Mettre la valeur du sous-registre high à l'adresse pointée par SP.
- Encore décrémenter SP de 1.
- Mettre la valeur du sous-registre low à l'adresse SP.

Pour chaque déclaration de variable globale, on va donc mettre la valeur de l'initialisation dans un registre 16-bits disponible :
- Si c'est un int ou $boolean$, on choisi de toujours mettre ces valeurs dans le sous-registre low.
- Si c'est un objet, on utilise tout le registre pour y mettre son adresse. Ensuite, on fait un simple *push* de ce registre dans la pile.

Enfin, dans notre compilateur, pour chaque déclaration globale, nous stockons une *HashMap* qui associe le nom de la variable à son adresse. Il est alors très facile de retrouver l'emplacement mémoire d'une variable pour faire une assignation, ou encore d'accéder à sa valeur.

Opérations Arithmétiques

Les seules opérations arithmétiques disponibles sont **Add A, r8** et **Sub A, r8**.

Pour faire une addition ou une soustraction, il faut encore une fois mettre la valeur basse de l'autre registre dans A, faire l'opération, et stocker A dans la partie basse de l'autre registre. Par exemple, pour l'addition de BC et DE :
- Ld A, C
- Add A, E
- Ld C, A

Pour la multiplication, la division, et le modulo c'est plus difficile. Comme il n'y a pas d'instruction native pour ces opérations, nous avons dû faire des boucles assez naïves pour calculer le résultat. Par exemple, pour la multiplication, à chaque tour de boucle, tant que l'opérande de droite est supérieur à 0, on ajoute à l'opérande de gauche sa valeur initiale, puis on retire 1 à l'opérande de droite.

Un pseudo-code pour faire $x \times y$ pour cela serait :

```
result = 0;
while (y > 0) {
    result = result + x;
    y = y - 1;
}
```

Encore une fois, il faut passer par A pour faire l'addition et la soustraction, comme A ne peut pas stocker les deux valeurs courantes à la fois, on utilise les registres H et L comme registres temporaires.

Pour la division et le modulo, c'est le même type d'idée de boucle.

Nous sommes conscient que cela n'est le plus efficace, et, nous avons quelques pistes pour optimiser ces opérations, par exemple, la décomposition d'une opérande en puissance de 2, puis, utiliser les shifts. Cependant, par manque de temps, cela n'a pas été possible à implémenter.

3.1.3 Langage Essentiel

Construction d'une VTable

Pour chacune des classes, la construction de sa VTable se fait de la même manière qu'en IMA, c'est-à-dire que :
- Pour chaque attribut, on va y stocker sa position.
- Pour chaque méthode, on va y stocker l'information sur ses paramètres.
Cependant, il n'est pas utile de stocker l'index de la méthode car nous avons décidé qu'il n'y aurai pas de polymorphisme en Deca pour la GameBoy. Nous verrons pourquoi lorsque nous parlerons des appels de méthodes.

Initialisation des Attributs

Pour faire l'initialisation des attributs, nous avons tout d'abord eu à décider où stocker l'instance courante de la classe avant d'appeler la fonction *init* de la classe. Pour rester en accord avec ce que nous avons fait pour IMA, nous avons décidé de stocker l'instance courante dans les adresses SP+2 (valeur basse) et SP+3 (valeur haute). L'adresse de retour est quant à elle, stocké dans SP+0 et SP+1 (un schéma est disponible dans la prochaine partie).

En effet, cela correspond bien à ce qui se fait en IMA où l'instance est stocké en -2(LB), et, l'adresse de retour en -1(LB), car :
- Une case de pile en IMA est équivalent à deux cases en GameBoy.
- La pile en GameBoy va dans le sens inverse, d'où les + au lieu des -.

Grâce à cela, nous avons pu accéder aux adresses des attributs de la classe à partir de l'instance courante pour les initialiser à 0, puis, à la valeur voulue s'il y en a une. Nous allons détailler dans la partie suivante, comment accéder à ces attributs.

Le New

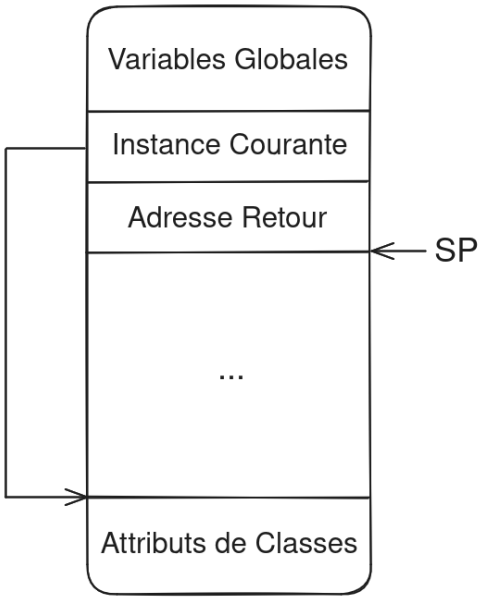
La GameBoy n'ayant pas de tas, nous avons dû chercher une autre méthode pour stocker les attributs d'une classe dynamiquement. Après avoir testé nombreuses méthodes d'allocation de place mémoire pour le *new* sans succès, nous sommes finalement arrivé à faire quelque chose qui marche comme en IMA, juste avec la pile de la GameBoy.

Le principe est simple, il faut :
- Stocker un compteur contenant l'adresse du prochain *new* dans une case prédéfinie.
- Allouer les attributs en partant de la tête de la pile (0xC000).

Ici, nous avons décidé de stocker ce compteur tout en haut de la pile, aux adresses 0xDFFF (valeur haute) et 0xDFFE (valeur basse).

Pour chaque *new*, il suffit donc de :
- Récupérer l'adresse du prochain *new* dans le compteur.
- Incrémenter le compteur de $2\times$ (Nombre d'attributs de la classe à instancier).
- *Push* l'instance de la classe dans la pile.
- Appeler la fonction *init* de la classe à instancier.

Voici un schéma de la pile pendant un *new* :



Selection d'Attributs & This

Pour le *this*, il suffit de mettre l'instance courante de la classe stocké dans SP+3 et SP+2 dans un registre libre.

Ensuite, pour sélectionner un attribut, il faut ajouter la position de l'attribut à l'adresse vers laquelle l'instance courante pointe. Par exemple, pour accéder à un attribut de position N dans la classe, il faut donc ajouter $2 \times N$ dans l'adresse pointée par l'instance courante.

Appel d'une Méthode

L'équivalent de l'instruction assembleur **Bsr** en IMA, est l'instruction **Call** en GameBoy. Cependant lorsque l'on regarde cette instruction plus en détail, on voit qu'elle ne peut que prendre un label en argument : **Call Label**. Ceci pose problème pour le polymorphsime, en effet, on ne pourrait pas faire quelque chose comme **Call N(Registre)** comme en IMA. Nous avons alors décidé de ne pas faire de polymorphisme pour Deca en GameBoy. Les appels de méthode se feront tous donc sur le type statique de la variable d'instance, par exemple, si on a :

```
class A { int f() { return 1; } }
class B { int f() { return 2; } }
{ A a = new B();
int x = a.f(); }
```

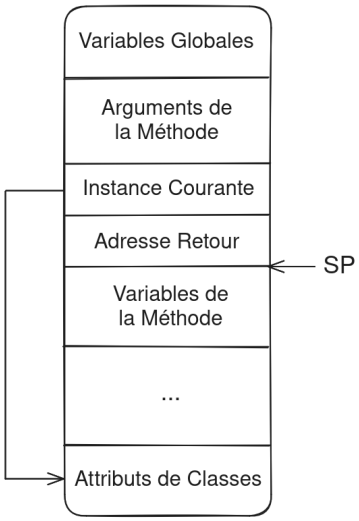
Alors, la méthode appelée sera celle de A et non de celle B, contrairement à ce qu'il se passerait sur IMA. La variable x contiendra donc 1 et non 2.

Arguments d'une Méthode & Variables d'une Méthode

Pour rester en accord avec IMA, les arguments lors d'un appel de méthode sont stockés juste après l'instance de la classe courante. C'est-à-dire que le 1^{er} argument est stocké à l'adresse SP+5 (valeur haute) et SP+4 (valeur basse), le $2^{ème}$ argument est stocké à l'adresse SP+7 et SP+6, etc... Pour cela, nous faisons simplement un *push* des arguments dans le sens inverse pour chaque appel de méthode.

Les variables d'une méthode sont quant à elles stockées juste après l'adresse de retour (plus les éventuelles sauvegardes de registres au début de la méthode). Par exemple, la $1^{ère}$ variable est stockée à l'adresse SP-1 (valeur haute) et SP-2 (valeur basse), la $2^{ème}$ variable est stockée à l'adresse SP-3 et SP-4, etc...

Voici un schéma de la pile pendant un appel de méthode :



Return d'une Méthode

La valeur de retour d'une méthode est stockée dans le registre scratch HL. En effet, on ne peut pas utiliser le registre A, car, comme on peut renvoyer une instance de classe, et qu'une adresse est codée sur 16-bits, il faut absolument utiliser un registre scratch 16-bits.

Gestion du SP dans une Méthode

La partie la plus difficile était la gestion du pointeur SP pendant un appel de méthode. En effet, en IMA, nous avions les 3 pointeurs SP, GB et LB pour nous aider, alors qu'en GameBoy, SP doit s'occuper seul de tout, afin de respecter les emplacements des arguments, variables de la méthode etc... cités dans les précédentes parties.

Nous avions eu une multitude de problème par rapport à la gestion du pointeur SP dans une méthode, voici une liste non exhaustive de choses que nous avions eu à considérer :

— Quand on fait un appel de méthode dans une méthode courante, la méthode courante possède des variables locales, donc il faut d'abord déplacer SP vers le bas, après ces variables. Sauf que dans ce cas, il y a par exemple les arguments de la méthode courante qui ne sont plus au bon endroit (SP+5 et SP+4, etc...), ils sont beaucoup plus loin. Ensuite, il faut *push* les arguments de la méthode qu'on veut appeler, et cela décrémente encore une fois SP de 2 pour chaque argument. Par exemple :

```
void f(int x) {
    int a = 0;
    int b = 1;
    int c = 2;
    g(x);
}
```

Pour aller chercher la valeur de x pour le mettre en argument dans l'appel de la méthode g, on va aller chercher x non pas à l'adresse SP+5 et SP+4, mais à SP+11 et SP+10. En effet, il faut ajouter 6 aux deux adresses car il y a 3 variables locales.
En plus des variables locales, il fallait aussi considérer les valeurs sauvegardées temporairement qu'il ne fallait pas écraser aussi.

— Quand on fait un appel de méthode dans un appel de méthode dans une méthode courante, le premier appel de méthode aura déjà déplacé SP vers le bas, donc le deuxième n'a plus besoin de le déplacer. C'est pareil si on fait un appel dans un appel dans un appel...

— Dans une méthode, quand on fait une opération arithmétique compliquée et qu'on doit sauvegarder des registres, il ne faut absolument pas *push* le registre comme ça, sinon on va écraser les valeurs des variables locales de la méthode. Il faut mettre SP à l'adresse de la dernière variable locale, puis *push* la valeur temporaire. Puis, pour reprendre la variable avec un *pop*, il faut aller à l'adresse + 2 où l'on a *push*, et faire le *pop*.

Les Méthodes Asm

Les méthodes *asm* sont gérées de la même manière qu'en IMA, la sauvegarde des registres, le retour d'une fonction, ainsi que la bonne gestion du pointeur SP, sont à la charge du programmeur. Le programmeur doit néanmoins savoir que :
- L'instance de la classe courante est stockée dans SP+3 et SP+2.
- Les arguments sont stockés dans SP+5 et SP+4, puis, SP+7 et SP+6 etc...
- La valeur de retour doit être stockée dans le registre HL, où H sera la partie haute, et L la partie basse de la valeur de retour.

La Méthode Equals de Object

La méthode *equals* fonctionne de la même manière qu'en Deca pour IMA. Pour la coder, il a fallu faire deux comparaisons car une adresse est codée sur 16-bits. On compare d'abord la partie haute des deux adresses, puis la partie basse.

Par ailleurs, la comparaison d'une variable de classe à *null* (ou une autre variable) se fait de la même manière. Comme il n'y a pas d'équivalent du *#null* de IMA en GameBoy, nous avons simplement considéré que la valeur *null* est l'adresse 0. Comme on travaille sur la plage d'adresses 0xC000 - 0xDFFF, la valeur 0 est cohérente pour *null*.

Héritage

Pour l'héritage, nous n'avons rien eu à faire, il a suffit de réutiliser le code que nous avions fait pour IMA.

3.1.4 Langage Complet

Le langage complet consiste à ajouter le *instanceof* et les *cast* au langage essentiel. Cependant, au vu de notre implémentation du langage essentiel, nous n'avons pas jugé utile d'implémenter ces deux parties du langage.

En effet, l'utilisation du *instanceof* est surtout utile pour la gestion du polymorphisme dans les programmes, chose que nous avons décidé de ne pas faire comme décrit précédemment. C'est la même chose pour les *casts* d'objets, ils sont inutiles sans polymorphisme. De plus, comme les *int* et les *float* sont équivalents en Deca pour la GameBoy, faire un *cast* pour passer de l'un à l'autre est inutile.

3.2 Conception de la librarie GameBoy.decah

Cette librairie contient des méthodes et des constantes servant à gérer les *input* et les *output* de la console. **La méthode init :**

Cette méthode défini des label utils pour les autres méthodes (sans les éxecuter car on place un ret avant eux) met en place dans la mémoire un environnement facilitant le développement :

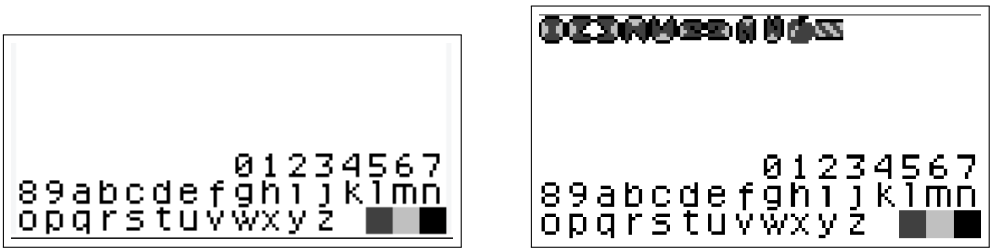


Figure 1 – *Visualisation des "tiles" dans la mémoire à l'aide de l'émulateur (à gauche l'utilisateur à importé ses tiles avec #includeTiles).*

Les *tiles* sont des blocs de 8×8 pixels que l'on peut placer où on veut sur l'écran. Dans la mémoire VRAM, un espace de 2MB est consacré à ces *tiles*. Les données *tile* sont stockées sur 16 octets (64 pixels * 2 bit pour la couleur) donc il peut y en avoir 128. Elles sont référencées par leur indice dans la mémoire, de 0 à 127, nous verrons cela plus tard. On peut les éditer avec des outils en ligne, qui donnent directement les 16 octets en brut, comme sur la Figure 2.

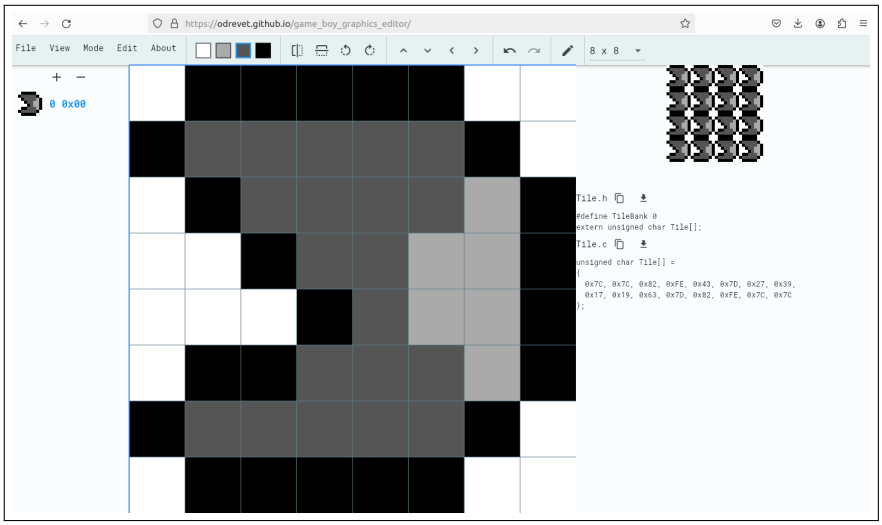


Figure 2 – *Rafraîchissement de l'écran*

Dans la méthode **init**, on place les *tiles* des charactères et des couleurs pleines à la fin, de cette manière, si l'utilisateur à importé ses propres *tiles* avec **#includeTiles**, elles sont placées au début, toujours avec la méthode **init**, voir Figure 1.

Dans cette librairie, les couleurs sont en fait des entiers qui représentent les des *tiles* correspondantes dans la mémoire (ici 124, 125, 126, 127, car ce sont les 4 dernières des 128 *tiles* dans la mémoire. On peut y accéder avec les constantes **BLACK, DARK, LIGHT, WHITE**.

Dans la méthode **init**, on initialise une liste chaînée, la *DrawEventList*, contenant 20 objets de type *DrawEvent*, ceux-ci contiennent l'indice d'une *tile* à dessiner, ainsi que sa position.

Méthodes de modifications de l'écran :

La libraire contient les méhodes **setTile(tileIndex, x, y)** et **setColor(color, x, y)** (qui appelle juste **setTile**), qui "rajoutent" un *DrawEvent* à la *DrawEventList* avec les données passées en paramètre. En faisant cela, on ne crée pas de nouvel objet en mémoire, car si on en créait à chaque modification, comme on ne peut pas libérer celle-ci, on déborderait assez vite. C'est pour cette raison que la liste est de taille fixe : on crée les *DrawEvent* dès le début et après on modifie simplement leurs attributs. On a un compteur interne à la classe *DrawEventList* pour savoir où on en est dans les modifications. On ignore les modifications qui dépassent la capacité. Cela dit, 20 c'est déjà beaucoup de tiles modifiées entre deux rafraîchissements de l'écran.

On dispose également de la méthode **setBackgroundColor(color)**, qui modifie la couleur du champs *map* de type *BackgroundMapMod*. Ce type possède un attribut *changed* pour ne pas faire de changements inutiles.

Avant de continuer, précisons que le rafraîchissement de l'écran se fait ligne par ligne, et ne peut pas être arrêté sauf en éteignant provisoirement l'écran. De plus si on veut faire des modifications dans la mémoire VRAM (qui contient les *tiles* et les *tilemaps*), on doit faire toutes ces modifications pendant le court laps de temps durant lequel la GameBoy "dessine" le "VBlank" (voir la Figure **??**).

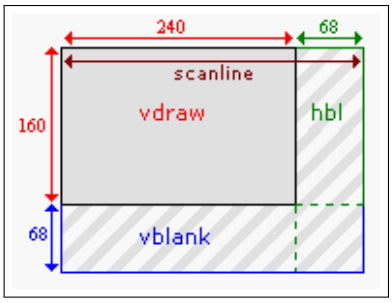


Figure 3 – *Rafraîchissement de l'écran*
(ici les dimensions ne sont pas les bonnes, c'est un exemple illustratif)

De plus, une modification consiste à charger l'indice d'une *tile* à dessiner dans la *tile map* à l'adresse correspondante aux coordonnées voulues, voir Firgure 4

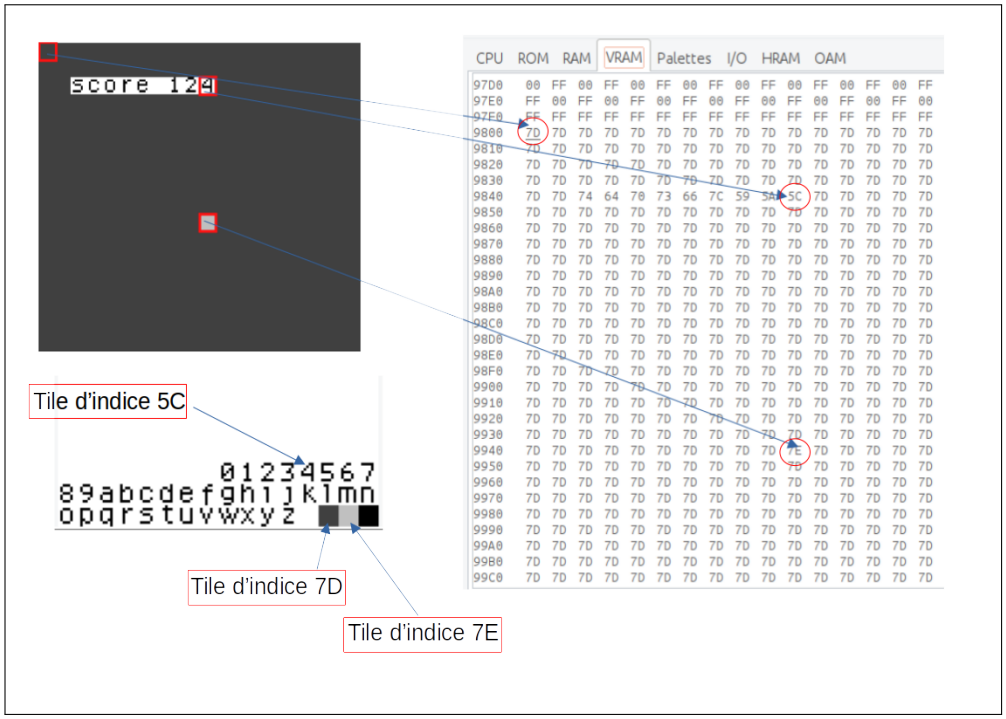


Figure 4 – *Dessin des tiles sur l'écran*

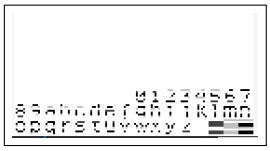


Figure 5 – *Si on charge les tiles dans la VRAM au mauvais moment.*

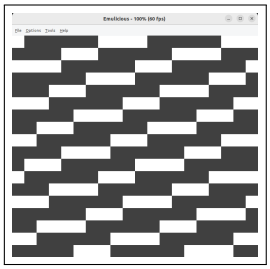


Figure 6 – *Si on écrit les indices des tiles souhaitées dans la map, dans la VRAM, au mauvais moment (ici on voulait un écran tout gris).*

Update Screen :

Pour mettre à jour l'écran, on a créé la méthode **updateScreen**. Dans un premier temps, elle regarde si la *map* a changé de couleur, si oui on éteint l'écran pour mettre en mémoire la nouvelle couleur. On doit éteindre l'écran pour faire autant de changements dans la mémoire car cela met plus de temps que l'intervalle laissé par le *VBlank*.

Ensuite, on parcours la *DrawEventList* jusqu'au dernier index modifié, et au pour chaque *DrawEvent*, on appelle *pushInTileMap* qui est une méthode asm "privée" (pas déstinée aux utilisateurs). Elle fait les calculs nécessaire à partir de x et y pour trouver l'adresse correspondante dans la mémoire au niveau de la *tile map*. On ne peut pas passer directement cette adresse en paramètre car les entiers sont codés sur 8 bits et sont positifs, donc entre 0 et 255, et donc on ne peut manipuler que des coordonnées. Elle attend ensuite d'être dans le *VBlank* pour charger l'indice de la *tile* à dessiner dans la VRAM.

Code menant à la Figure 7 :

```
#include "GameBoy.decah"

{
    GameBoy gb = new GameBoy();
    int x;
    gb.init();
    gb.setBackgroundColor(gb.DARK);
    gb.setColor(gb.LIGHT, 10, 10);
    gb.updateScreen();
    x = 124;
    print("score ", 2, 2);
    gb.printNumber(x, 8, 2);
}
```

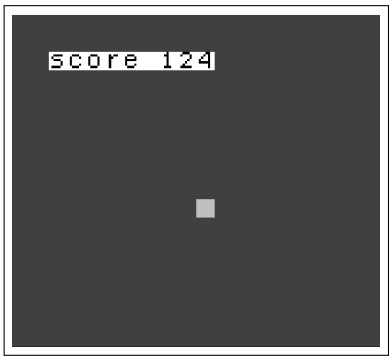


Figure 7 – *Exemple d'utilisation de la librairie.*

On remarque qu'il est donc très simple d'afficher les éléments souhaités, et que le contenu des **print** et **GameBoy.printNumber** s'affiche immédiatement.

Input :

On peut savoir les touches pressées par l'utilisateur à l'aide de la méthode **keyPressed(key)**. La librairie contient les constantes : **UP_KEY**, **DOWN_KEY**, **LEFT_KEY**, **RIGHT_KEY**, que l'on peut passer en paramètre de cette méthode. Celle-ci renvoie vraie si la touche a été pressée.

Cette méthode commence par *call UpdateKeys*, qui est un label défini dans **init**, qui met dans une "variable" le code associé à une touche. Ces "variables sont stockées dans ce qui est appelé la WRAM dans l'assembleur, mais qui correspond en fait à une zone de la pile.

```
SECTION \"Variables\", WRAM0

wVBlankCount: db
wCurKeys: db
wNewKeys: db
randstate: ds 4
```

Ensuite, la méthode **keyPressed** compare le contenu de la variable *wNewKeys* avec le code passé en argument.

Print :

Print est une instruction native, alors pourquoi en parler ici ?

Et bien tout simplement car elle se base sur l'environnement de *tiles* initialisé par la librairie.

Le compilateur contient une classe **GameBoysUtils.java**, qui contient entre autres une *HashMap* qui à chaque caractère associe son indices dans les *tiles*, ou l'indice de la *blanche* s'il ne s'agit pas d'un caractère imprimable.

En suite la méthode **codeGenGb** de la classe **Print** effectue une vérification sur la validité des argument, et lève éventuellement une erreur contextuelle :

- print(String stringLiteral, int x, int y)
- print(String stringLiteral)

On éteint alors l'écran, car il va falloir faire plusieurs modifications, on décompose la chaîne de caractères, on calcul l'adresse dans la tile map à partir des coordonnées passées (en (5, 5) sinon), et on y charge l'indice obtenu aveec la **HashMap**. Pour finir on rallume l'écran.

printNumber :

L'affichage d'entiers, literaux ou non, se fait cependant avec une méthode de la librairie.

Celle-ci applique un modulo 10 puis une division entière plusieurs fois à l'entier passé en paramètre, pour obtenir ses chiffres et les afficher ensuite. L'affichage d'un simple chiffre est quant à lui un problème simple :

Les chiffres sont dans l'ordre dans les *tiles*, il suffit d'ajouter le chiffre à l'indice de la *tile* de 0 pour obtenir l'indice de sa propre *tile*. Finalement il suffit de ne pas oublier de décaler l'affichage, car on commence par le dernier chiffre. pour cela il suffit de vérifier si le nombre contient 1, 2 ou 3 chiffres.

4 Validation de l'extension

4.1 Validation du code généré

4.1.1 Méthode de validation

Avant de pouvoir écrire notre librairie GameBoy en Deca, ou encore faire des jeux GameBoy en Deca, il faut vérifier que le code généré par le compilateur est valide et fonctionne sur la GameBoy.

La GameBoy n'ayant pas de terminal pouvant afficher du texte comme en IMA, notre méthode de test utilisé pour la validation du code généré en IMA n'était pas ré-applicable ici [6].

Nous avons donc choisi d'opter pour une méthode de validation semi-automatisée. Cette méthode consiste à utiliser l'écran de la GameBoy pour afficher un certain motif si un test ne donne pas le résultat attendu. Pour cela, nous avons transformé la fonction *println* pour qu'au lieu d'afficher du texte dans la sortie standard, affiche une image contenant "Hello World" sur l'écran de la GameBoy (quels que soit les paramètres passés dans la fonction *println*).

Par exemple, pour tester l'assignation d'une valeur à une variable, on ferait :

```
int x;
x = 1;
if (x != 1) { println(); }
```

Ici, si l'assignation est valide et que x vaut bien 1, l'écran restera comme il est par défaut (logo "Nintendo"), alors que si x ne vaut pas 1, il s'affiche "Hello World" en gros sur l'écran.

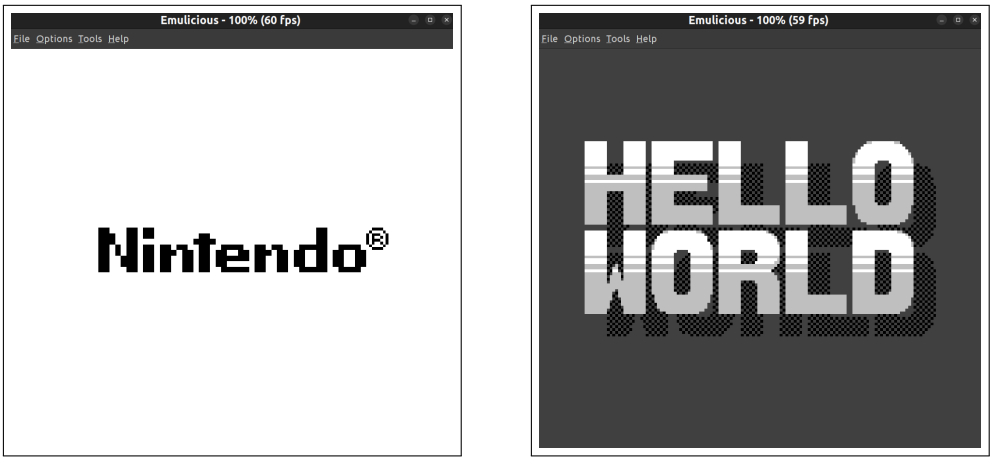


Figure 8 – *Affichage "Nintendo" si le programme est correct, sinon "Hello World".*

6. Suite a quelques recherches, nous avons trouvé que l'architecture GameBoy est compatible avec le protocole de communication UART. Cependant, l'émulateur Emulicious que nous avons utilisé ne fourni pas de moyen pour afficher les messages communiqués à la GameBoy.

Pour cela, il fallait bien-sûr s'assurer que *println*, les comparaisons, et les *if/else* étaient corrects dès le début. Heureusement, pour les comparaisons, et les *if/else*, la façon de générer le code était très similaire à celui pour IMA en utilisant les évaluations paresseuses des expressions booléennes.

Avec cela, nous avons alors pu faire un script Python **tests_gameboy.py**, qui pour chaque tests, génère le code du programme Deca, et lance l'émulateur. Il faut alors voir si "Hello World" est affiché ou non sur chaque écran d'émulateur.

Cependant, il faut aussi gérer le cas où "Hello World" ne s'affiche pas, non pas parce que le programme est valide, mais parce que le programme est bloqué dans un autre endroit, ou réussi la condition de test "par chance". Pour cela, nous nous assurons que le programme fait bien les conditions, simplement en inversant une condition. Dans ce cas, si "Hello World" s'affiche, alors on passe bien dans la condition, et le code généré est correct.
En reprenant l'exemple précédent, on mettrait donc juste :

```
int x;
x = 1;
if (x == 1) { println(); }
```

Dans ce cas, on s'attend à voir un "Hello World".

Comme faire ces inversions de conditions prend quand même pas mal de temps à tester, nous ne les faisons pas à chaque fois qu'on veut tout tester. Par exemple, si on est entrain de travailler sur la génération de code pour l'assignation, on va faire cette inversion de condition pour le test associé, mais pas ceux qui passaient déjà avant.

Finalement, il faut quand même s'assurer que tout fonctionne correctement, donc, lorsque l'on fait une grosse modification sur la génération du code, nous faisons cette inversion de condition pour des anciens tests ciblés, et on vérifie qu'ils fonctionnent toujours correctement.

4.1.2 Tests utilisés pour la validation

Les tests utilisés sont tous stocké dans le dossier **src/test/deca/gameboy/**. Cependant, les sous-dossiers utilisés pour les tests de la génération de code sont **cond**, **base**, et **object**. Le dossier **bin** est utilisé pour stocker les fichiers générés par le compilateur, il y a aussi un fichier **hardware.inc** nécessaire au fonctionnement des tests, qu'on inclut au tout début de chaque fichier assembleur généré. Les autres dossiers sont utilisés pour les tests de la librairie.

Tous les tests utilisés sont dans le fichier **tests_gameboy.py**. Les tests ont une difficulté incrémentale. Les tout derniers tests sont des tests de listes chaînées et d'arbres binaires, ce qui montre que notre compilateur pour la GameBoy est tout de même bien robuste. De plus, l'écriture de la librairie GameBoy.decah en parallèle a permis de trouver quelques bugs sur le code généré, c'était aussi en quelque sorte un test que nous avons utilisé.

4.2 Validation de la Librairie GameBoy.decah

Nous avons écrit un programme deca du jeu Snake avant même d'avoir commencé la librairie, de cette manière, on obligeait en quelque sorte la librairie à être intuitive.

Ensuite, au cours du développement nous avons écrit des tests unitaire pour chaque méthodes de la librairie, puis des petits programmes pour voir s'ils affichaient bien les bonnes choses :

- Un programme appelant simplement la méthode **init**, qui met le fond en blanc par défaut.
- Un programme faisant plusieurs appels à **setColor**.
- Un programme utilisant l'instruction native **print**.
- Un programme utilisant la méthode **printNumber**.

Ces tests manquent de dynamisme, on a donc ensute écrit les tests suivants :

- Balle rebondissante.
- Point qui avance avec les touches pour tester les inputs.
- Finalement, il suffisait de légèrement adapter le jeu Snake écrit au début pour qu'il fonctionne, permettant au passage de tester les **#includeTiles**.