Les bases du développement d'un jeu sur NES

Ronan Dhersignerie - https://ronan-dhersignerie.fr/
Clément Clerc - https://clementclerc.carbonmade.com/
Antoine Pavy - https://antoine-pavy.fr/
Quentin Pamart

Les bases du développement d'un jeu sur NES

Introduction

Bonjour et bienvenue pour ce cours sur les bases du développement de jeux vidéo sur NES !

Nous sommes Antoine, Clément, Ronan et Quentin, 4 étudiants en 2ème année de Master 3D&JV à l'ESGI et nous nous sommes réunis tous ensemble pour vous proposer ce cours sur la NES !

Dans ce cours, vous développerez de nombreuses compétences, toutes liées autour d'un sujet commun : le développement de jeux rétro.

En effet, du fait des limitations techniques, développer un jeu pour une console rétro est complètement différent d'un développement moderne. C'est pourquoi nous allons vous présenter de nombreux outils, afin de vous aider au mieux à développer votre propre jeu.

Nous allons commencer par vous présenter la Nintendo Entertainment System ainsi que son impact dans le monde du jeu vidéo, et ce afin de se resituer où se positionnait la NES par rapport à la concurrence de l'époque. En plus de cela, nous vous présenterons son architecture, pour mieux comprendre son fonctionnement.

Ensuite, nous vous introduirons à la programmation avec l'assembleur ASM 6502 ainsi que ses contraintes. Nous nous pencherons notamment sur les spécificités liées à la NES, comme par exemple les plages d'adresses mémoires dédiés à différentes tâches.

Nous allons ensuite vous présenter la partie graphique de la NES, de la réalisation d'assets graphiques, jusqu'à l'affichage et l'animation de sprites.

Le cours continuera en développant des notions plus complexes telles que les contrôles et déplacements d'un sprite, ou encore les collisions.

Pour finir, nous vous présenterons comment réaliser de la musique pour NES avec FamiTracker, et nous verrons comment intégrer celle-ci à un programme ASM 6502.

D'ici la fin de ce cours, vous aurez toutes les bases requises pour développer votre propre petit programme sur NES de A à Z.

Allez, plongez avec nous dans le monde du retro gaming ! C'est parti !

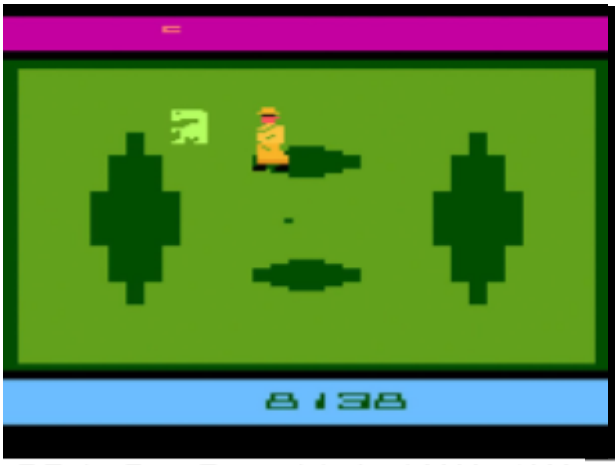
Vidéos : Présentation du groupe et introduction

Présentation de la NES et historique

Présentation de la NES

La NES ou Nintendo Entertainment System, connue au Japon sous le nom de FAMICOM, est une console de jeux vidéo construite et distribuée par Nintendo de 1983 à 2003. A sa sortie, elle connut un grand succès en devenant la console la plus vendue au Japon dès sa deuxième année d'existence. Aujourd'hui ses ventes totales sont estimées à 61,91 millions.

La NES a transcendé cette ère du jeu vidéo en cassant complètement les codes. Tout d'abord, bien que la NES était tout de même un investissement pour l'époque (14 800 yens soit environ 110€), elle proposait un catalogue de jeux inégalé. La seule concurrence notable à sa sortie était l'Atari 2600 sortie en 1977. Étant sortie 6 ans plus tard au Japon, la NES a pu profiter de meilleurs composants, une meilleure architecture, et donc des performances bien meilleures en comparaison.



E.T. the Extra Terrestrial - Atari 2600 - 1982

Duck Hunt - NES - 1984

L'arrivée de la NES fut la sortie de console salvatrice pour l'industrie du jeu vidéo. En effet, Atari ne faisant pas de contrôle de qualité quand aux jeux qui sortaient sur leur plateforme, tout et n'importe quoi pouvait être vendu, ce qui a pollué le marché très rapidement de jeux de très mauvaise qualité. Nintendo y a vu là un créneau à prendre, c'est pourquoi ils ont décidé d'instaurer un petit logo doré qui marquera des générations : le *Nintendo's seal of approval*. Celui-ci était en réalité une preuve destinée aux acheteurs, que chaque jeu terminant sur la NES devrait avoir été validé auprès de Nintendo eux mêmes, ce que Atari ne faisait pas.

Globalement, cette augmentation de la qualité des jeux vidéo apportée par la NES aura été le levier qui a sauvé l'industrie du jeu vidéo toute entière.

Mais la NES, c'est quoi concrètement ?

La Nintendo Entertainment System est, pour faire un euphémisme, assez loin des performances actuelles de nos consoles dites “next-gen”.

Son CPU, le Ricoh 2A03, est basé sur la technologie 8-bit 6502. celui-ci tourne à 1.79MHz (ou 1.66 sur les systèmes PAL) et fournit un total astronomique de 2Ko de RAM.... Pour vous donner une idée, une capture d’écran sur un téléphone en 1080p pèse en moyenne 350Ko, ce qui est 175 fois plus gros que l’entièreté de la RAM de la NES.

La NES s’est vue accompagnée de plusieurs accessoires, à commencer par la manette. Celle-ci était pourvue de 8 boutons, ce qui était un confort important, quand on compare par exemple à l’Atari 2600 qui n’avait qu’un joystick et un bouton unique.

Une deuxième manette plus originale est arrivée avec la NES : le zapper. Ce pistolet en plastique permettait de se servir de sa manette comme d’un vrai pistolet et de pouvoir viser directement sur un écran comme si le pistolet était réel. Cette prouesse technologique a fait partie des innovations de la NES qui l’ont rendue si populaire.

Vidéo : Démontage NES et cartouche

Vidéo : Utilisation de l'émulateur FCEUX

Assembleur 6502

Programmer en assembleur

Présentation du langage d'assemblage

Avant de commencer à créer un jeu pour la NES, nous devons apprendre à coder avec le langage utilisé sur cette machine : le langage d'assemblage. Il s'agit d'un langage de programmation bas niveau. Par conséquent, il est plus performant et permet un contrôle de la mémoire plus précis qu'un langage de plus haut niveau comme du Python ou du C#. Cela signifie également que nous n'avons pas accès aux structures et aux objets auxquels nous sommes habitués. Il n'y a par exemple pas de boucle for, pas de while ou encore pas de liste. Tout s'effectue en manipulant les données stockées en mémoire.

Le code ainsi produit est ensuite lu par un programme appelé assembleur qui va le traduire en code machine compréhensible par le processeur. Cependant le code machine peut largement changer d'un processeur à un autre. Chaque processeur possède en effet un set d'instruction spécifique défini par son constructeur. Par conséquent, il existe un langage d'assemblage spécifique à chaque architecture processeur qui ne sera comprise que par cette dernière. Dans le cas de la NES, le CPU se base sur un microprocesseur 6502. Le langage d'assemblage associé est donc désigné sous le nom Assembleur 6502.

Exemple de code

On trouve en assembleur 5 types de commandes.

- **Les directives** sont des commandes qui fournissent des informations au programme. Elles permettent entre autres d'initialiser de la mémoire, de réserver de l'espace mémoire pour des variables non initialisées ou encore de déclarer des variables globales. Ces commandes sont reconnaissables dans le code car elles commencent par un point.
- **Les Opérations** Codes, abrégées en Opcodes, sont les opérations qui indiquent au processeur quelles actions il doit effectuer. Il s'agit de la forme de programmation de plus bas niveau compréhensible par l'humain. En assembleur ces opérations sont désignées par des instructions Elles sont le plus souvent accompagnées d'un opérande, une donnée ou une adresse mémoire sur laquelle effectuer l'action.
- **Les labels** servent à sauvegarder l'adresse mémoire de la prochaine instruction. On peut donc leur donner un nom arbitraire pour désigner leur utilité afin de ne pas avoir à manipuler des valeurs peu intuitives. Elles sont représentées par un nom suivi de : et peuvent ensuite être utilisées comme des opérandes

- Les commentaires, des indications à destination des utilisateurs et qui ne seront pas lus par l'assembleur. Ces commentaires sont indiqués par un ; au début de la ligne.

Voici un exemple de code pour illustrer tous ces concepts :

```
$8000
ExempleFunction: ;enregistre une valeur dans l'accumulateur
LDA #$01
JMP ExempleFunction
```

La première ligne est une directive indiquant que le code commence à utiliser la mémoire à l'adresse $8000.

La deuxième ligne est un label qui enregistre l'adresse à laquelle la ligne suivante est située. Avec la directive utilisée précédemment, cette adresse est $8000. La phrase après le point virgule est un commentaire.

La 3ème ligne est un Opcode qui consiste à charger l'opérande $01 dans l'accumulateur (nous verrons plus tard ce qu'est l'accumulateur).

La dernière ligne est un autre Opcode qui indique au code de se positionner à l'adresse enregistrée par le label.

Par conséquent cet exemple va continuellement boucler sur lui-même et exécuter la même opération

Focus sur l'assembleur 6502

Les registres

En assembleur, un registre désigne une cellule mémoire située directement dans le microprocesseur qui peut stocker des données. L'avantage d'un registre par rapport à la mémoire vive est la rapidité d'accès dû au fait qu'il est implanté directement dans le microprocesseur. Le microprocesseur 6502 propose 6 registres différents.

- **L'accumulateur**, désigné par le nom "A", est un registre de 8 bits. Il est principalement utilisé pour toutes les opérations arithmétiques et logiques.
- Les **registres d'index X et Y** sont deux registres de 8 bits très similaires. Le plus souvent, ils sont utilisés pour gérer des compteurs, dans une boucle par exemple.
- Le **registre Program Counter** ou PC est un registre de 16-bits. Il pointe vers l'adresse de la prochaine instruction à exécuter. Sa valeur est donc le plus

souvent modifiée automatiquement avec l'exécution du programme, mais elle peut également être modifiée manuellement avec certaines instructions.

- Le **Stack Pointer** ou SP est un registre 8 bit qui pointe vers le prochain emplacement libre de la pile du microprocesseur. Cette pile est un espace mémoire fixe de 256 octets situé entre $0100 et $01FF.
- Le dernier registre est le **Status Register** ou Processor Status. Il s'agit également d'un registre 8 bit. Comme son nom l'indique, il stocke des informations sur l'état du processeur au fur et à mesure de l'exécution et des opérations. Chaque bit de ce registre est appelé "drapeau" ou "flag".
    - Le second bit est le "**Zero Flag**" : il indique si le résultat de l'opération précédente vaut 0 ou non.
    - Le second bit est le "**Zero Flag**" : il indique si le résultat de l'opération précédente vaut 0 ou non.
    - *Le 3ème bit est le "***Interrupt Disable Flag***" :* tant que ce drapeau est activé, le processeur ignorera les interruptions liées à des périphériques (par exemple un input).
    - Le 4ème bit est le "**Decimale Mode Flag**" : lorsqu'il est activé, les calculs arithmétiques suivent la logique BCD (Binary Coded Decimal).
    - Le 5ème bit est le "**Break Flag**" : il est activé lorsqu'une requête d'interruption est envoyée.
    - Le 6ème bit est inutilisé.
    - Le 7ème bit est le "**Overflow Flag**" : il fonctionne de façon similaire au "Carry Flag" mais pour des entiers signés qui vont de -128 à 127.
    - Le 8ème bit est le "**Negative Flag**" : il est activé lorsque le résultat d'un opération mathématique est négatif.

Le jeu d'instruction

L'assembleur 6502 utilise 56 instructions différentes que nous allons détailler dans cette partie.

Pour commencer nous avons 12 instructions dédiés à la copie de données :

- Les instructions **TAX, TAY, TSX, TXA, TXS, TYA** qui permettent de copier des données de registre à registre. Le T signifie "Transfert" et les deux lettres suivantes désignent respectivement le registre duquel on récupère la donnée et le registre dans laquelle la donnée sera copiée. A pour l'Accumulateur, S pour le registre Stack Pointer, X ou Y pour les registre X et Y. L'instruction TAX par exemple consiste donc à copier la donnée contenue dans l'Accumulateur vers le registre X.
- Les instructions **STA, STX et STY** sont utilisées pour copier des données d'un registre vers un emplacement mémoire. La dernière lettre de la commande indique le registre de départ tandis que l'emplacement d'arrivée dans la mémoire doit être passé en opérande.

- Les instructions **LDA, LDX et LDY** permettent de charger une donnée passée en opérande dans les différents registres, respectivement dans l'accumulateur, le registre X et le registre Y.

D'autres permettent de manipuler le contenu des registres :

- On a par exemple les instructions **DEX, DEY** qui permettent de décrémenter respectivement les registres X ou Y, et leur opposé **INX** et **INY** qui permettent d'incrémenter ces mêmes registres. Ces commandes existent spécifiquement pour les registres X et Y car ils sont souvent utilisés comme indice. On peut également incrémenter ou décrémenter la valeur à une adresse mémoire avec les instructions **INC** et **DEC**, suivi de l'adresse mémoire voulue en opérande.
- Le registre Accumulateur étant utilisé pour les opérations arithmétiques, il existe deux instructions afin d'effectuer des calculs : une addition avec **ADC** et une soustraction avec **SBC**. Les deux doivent être suivis d'un opérande indiquant la valeur à manipuler. Celle-ci sera donc additionnée ou soustraite à la valeur contenue dans l'Accumulateur et le résultat sera à nouveau stocké dans l'Accumulateur. Ces deux opérations prennent en compte le Carry Flag dans le calcul. L'addition prend donc la forme A = A + opérande + C tandis que la soustraction peut s'écrire A = A - opérande - (1 - C). S'il s'agit d'un nouveau calcul, il faut donc penser à réinitialiser le Carry Flag avec la commande CLC avant une addition, ou au contraire à l'initialiser avec SEC avant une soustraction.
- Il est également possible de décaler d'un bit vers la gauche ou la droite le contenu de l'Accumulateur avec les commandes **ASL** et **LSR**. Cela équivaut en fait à une multiplication ou une division par 2. Dans les 2 cas, le bit sortant est stocké dans le Carry Flag, et le bit d'entrée est un 0. Les commandes ROL et **ROR** effectuent un traitement similaire, à la différence que le bit d'entrée sera alors la valeur stockée dans le Carry Flag.

Nous avons ensuite accès aux 3 opérations logiques suivantes :

- L'instruction **AND** correspond à un ET binaire. Elle permet de comparer bit à bit le contenu de l'accumulateur avec la donnée passée en opérande. Le résultat de cette opération est un octet donc chaque bit vaut 1 si les bits respectifs de l'accumulateur et de la variable en opérande valent 1. Dans le cas contraire, le bit du résultat vaut 0. Ce résultat est ensuite stocké dans l'accumulateur.
- L'instruction **ORA** fonctionne de manière similaire, mais cette fois-ci l'opération correspond à un OR binaire. Cela signifie que le bit de retour vaut 1 tant qu'au moins un des bits respectifs de l'accumulateur et de l'opérande vaut 1.
- L'instruction **EOR** correspond à un XOR binaire. Cette fois ci le bit de résultat vaut 1 si strictement un seul des bits respectifs que l'on teste vaut 1.

Viennent ensuite les opérateurs de comparaison : **CMP**, **CPX** et **CPY** qui comparent respectivement le contenu de l'Accumulateur, du registre X et du registre Y avec un opérande passé en paramètre. Cette comparaison prend la forme d'une soustraction entre la valeur du registre et celle de l'opérande. Si les valeurs sont identiques, le résultat de la soustraction est 0, le Zero Flag et le Carry Flag sont passés à 1. Si la valeur du registre est supérieure, seul le Carry Flag est passé à 1.

En exploitant les résultats des comparaisons, on peut effectuer des branchements. Cela signifie que les instructions vont modifier l'ordre d'exécution du code selon diverses conditions. On précise pour cela en opérande la nouvelle adresse mémoire à laquelle le code doit s'exécuter. Souvent cette adresse est stockée grâce à un label, comme expliqué plus haut. Les instructions branchements sont les suivantes :

- **BEQ** et **BNE** vont vérifier l'état du Zero Flag et s'exécuter respectivement s'il est activé ou non. Comme on l'a vu avec les comparaisons, l'état de ce Flag peut dépendre d'une égalité, ce qui explique que ce soit la condition vérifiée par ces commandes.
- **BCS** et **BCC** vont vérifier l'état du Carry Flag. Comme pour les instructions précédentes, la première commande s'exécute si le Carry Flag est activé, tandis que la seconde s'exécute dans le cas contraire.
- **BMI** et **BPL** ont le même comportement avec le Negative Flag
- Enfin **BVS** et **BVC** vérifient l'état del'Overflow Flag.

En utilisant ces opérations, on peut recréer le comportement d'un if ou d'un for. Si besoin, on peut aussi modifier l'ordre d'exécution sans condition préalable avec la commande JMP suivi de l'adresse mémoire en opérande. On parle de saut pour désigner ces changements dans la logique d'exécution. Il y a une subtilité à prendre en compte : les sauts effectués avec les opérandes de branchement sont assez limités. On ne peut se déplacer que de 128 octets en avant ou en arrière.

Avec les instructions **JSR** et **RTS**, on peut simuler le comportement d'une fonction. La première sauvegarde dans la pile l'adresse mémoire de la ligne suivante à exécuter puis effectue un saut vers une adresse passée en opérande. La seconde permet au contraire de revenir à l'adresse mémoire stockée sur la pile. Grâce à la logique de la pile on peut appeler une fonction au sein d'une autre fonction. Il y a une commande spéciale pour retrouver l'adresse mémoire à laquelle l'exécution s'est arrêtée après une interruption : **RTI**. De plus, cette commande permet également de récupérer l'état des Flags au moment de l'interruption.

Enfin, nous avons 4 instructions qui permettent de manipuler la pile :

- Avec **PHA** on récupère la donnée de A et on l'enregistre au sommet de la pile.
- Avec **PLA** on récupère le sommet de la pile et on l'enregistre dans A.
- Les commandes **PHP** et **PLP** fonctionnent de façon similaire avec le registre P.

Il existe encore 3 instructions au comportement particulier et dont l'utilisation est beaucoup moins fréquente. La première est l'instruction **BIT** qui s'accompagne obligatoirement d'une adresse en opérande. L'instruction va récupérer les bits 7 et 6 de l'opérande et les copier respectivement dans le Negative Flag et l'Overvflow Flag. Le Zero Flag quant à lui est modifié en fonction du résultat d'un **AND** entre l'accumulateur et l'opérande. Ensuite, nous avons l'instruction **BRK** qui déclenche une interruption. Enfin, la commande **NOP** qui n'effectue aucune opération. Son utilité la plus fréquente consiste à créer des boucles d'attente.

Syntaxe des opérandes et adressage

Les opérandes qu'on envoie à nos instructions peuvent être vues comme des paramètres que l'on passe à une fonction. Seulement ici nous sommes limités à des valeurs numériques simples. On peut écrire ces valeurs dans différentes bases arithmétiques. On ajoute pour cela un caractère devant la valeur pour préciser la base :

- Si on n'écrit pas de caractère, la base par défaut est la base décimale.
- Si on ajoute un %, on indique que la valeur est en base binaire.
- Avec un $ devant le chiffre, cela signifie qu'on est en base hexadécimale.

En plus de la base à utiliser, l'opérande peut prendre 13 formes différentes. On parle alors de mode d'adressages.
Le premier mode, et le plus simple, est le mode implicite : on n'a pas besoin d'indiquer d'opérandes car il est implicitement connu par l'instruction. Par exemple on peut l'instruction CLC, qui met la valeur du Carry Flag à 0. L'instruction connaît l'adresse du Carry Flag à modifier ainsi que la valeur à lui attribuer, on n'a donc aucune information supplémentaire à lui partager
Ensuite on a le mode immédiat, qu'on peut utiliser en ajoutant le caractère # devant l'opérande. Cela indique qu'on utilise directement la valeur donnée en paramètre. Par exemple, si on écrit LDA #12, cela signifie tout simplement qu'on sauvegarde la valeur 12 dans l'accumulateur.
Vient ensuite l'adressage absolu : c'est le mode utilisé par défaut si on n'ajoute aucun caractère. Cela signifie que la valeur qu'on passe en opérande est une adresse et que c'est la valeur stockée à cette adresse qu'on veut utiliser. Avec la commande LDA 12, on sauvegarde la valeur à l'adresse 12 dans l'accumulateur.
En temps normal, une adresse s'écrit sur 16 bits. Cependant, les adresses qui permettent d'accéder aux 256 premiers octets de la mémoire peuvent s'écrire en utilisant uniquement les 8 bits de poids faibles car les 8 bits de poids lourds sont à 0. Ainsi, l'adresse $0043 peut s'écrire $43. On parle alors de mode d'adressage Zero Page. Comme il n'y a qu'un seul octet à spécifier, le traitement est plus rapide. On désigne d'ailleurs souvent ces 256 octets comme la "mémoire rapide".

Avec les instructions de branchement, on utilise des adresses relatives. La valeur qu'on passe en opérande est un offset qu'on ajoute à l'adresse à laquelle le programme est en train de s'exécuter. L'opérande ne pouvant dépasser 8 bits, cela explique la limite de déplacement dont nous parlions plus tôt. Si on passe une adresse absolue, avec un label par exemple, la logique reste la même car l'assembleur s'occupe de la conversion de l'adresse absolue en adresse relative.
L'instruction JMP possède un mode d'adressage qui lui est unique : l'absolu indirect, qu'on spécifie en écrivant entre parenthèses l'opérande passé à JMP. Cette instruction va récupérer la valeur stockée à l'adresse spécifiée ainsi que celle de l'adresse suivante. Ensuite ces deux valeurs vont être utilisées pour créer une adresse avec la première valeur en octet de poid faible, et la seconde valeur en octet de poid lourd. L'exécution du programme sera alors déplacée à cette adresse. Voici un exemple pour mieux visualiser. Imaginons que l'adresse $0312 contiennent la valeur $21 et que l'adresse suivante, $0313 contient la valeur $45. L'instruction JMP ($0312) va récupérer les valeurs $21 et $45 pour créer l'adresse $4521 et déplacer le Program Counter à cette adresse.
Nous avons ensuite 4 modes très similaires : l'adressage absolu indexé avec X et avec Y, qui fonctionne également avec un adressage Zero Page. Pour l'utiliser, on ajoute le nom du registre souhaité (X ou Y) avec une virgule derrière l'opérande. Cela permet d'additionner le contenu du registre avec l'opérande. Ainsi, si le registre X contient la valeur $03, l'instruction ADC $F012, X va ajouter $03 à $F012 pour donner $F015. C'est donc la variable à cette adresse qui sera utilisée.

Vidéo : Compiler un programme

PPU (Picture Processing Unit)

C'est l'une des puces centrales de la NES et qui définit la NES avec l'APU (Audio Processing Unit). N'importe qui peut aller acheter un CPU 6502 sur internet, mais il est compliqué de trouver le PPU / APU de la NES neuf (il est possible de s'en procurer en pièces détachées d'une autre NES)
Le PPU s'occupe de créer les graphismes 2D appelés Tiles et Background en les convertissant en signal vidéo. Pour créer le signal vidéo, le PPU a besoin de savoir quel type de graphismes dessinés, où le dessiner en screen space, et comment (Quelle palette utiliser).
Pour savoir cela le PPU possède quelques adresses mémoire préprogrammées :

Début	Fin	Type	Utilisation
$0000	$0FFF	ROM (Cartouche)	Première banque de 265 tuiles
$1000	$1FFF	ROM (Cartouche)	Deuxième banque de 265 tuiles
$2000	$23FF	RAM (NES)	Premier écran
$2400	$27FF	RAM (NES)	Deuxième écran
$2800	$2BFF	RAM (NES)	Troisième écran
$2C00	$2FFF	RAM (NES)	Quatrième écran
$3F00	$3F1F	RAM interne PPU	Palette de couleurs

Adresses mémoires utiles du PPU de la NES

Scanlines (Ligne de balayage)

La scanline est la technologie utilisée dans les écrans cathodiques pour afficher une image.

Démonstration de scanline en slow motion

Vidéo : Explication Scanline

Tilesets

Littéralement tuile en français, une tile est une image généralement carrée, (rarement rectangulaire, encore plus rarement hexagonale). Lorsque l'on dispose des tiles en grille cela crée un tileset (qu'on peut traduire par "jeu de tuiles" en français).

Sur NES

Pour la suite, nous partons du principe que nous avons les mêmes ressources qu'une cartouche de Super Mario Bros. (8KB pour les assets graphiques). Dans ces 8KB on peut stocker 2 tilesets de 16x16 tiles soit 256 tiles qui mesure chacune 8x8 pixels stockée en CHR Format. L'une de ces tilesets contient tous les assets du background c'est-à-dire tous les éléments visuels qui seront statiques. L'autre tileset est dédiée aux sprites, donc à tous les éléments qui seront animés.

CHR Format

Le CHR format est une façon de stocker les informations d'un pixel sur 2 bits (on peut donc stocker les informations sous la forme suivante : 00, 01, 10, ou 11). Si la NES utilisait le RGB une seule tile occuperait 192 bits. La NES aurait eu besoin de 96 Ko pour contenir les deux tilesets. Mais comment avoir des couleurs du coup ? On associe la valeur du bit du pixel à une adresse de couleur contenue dans une palette.

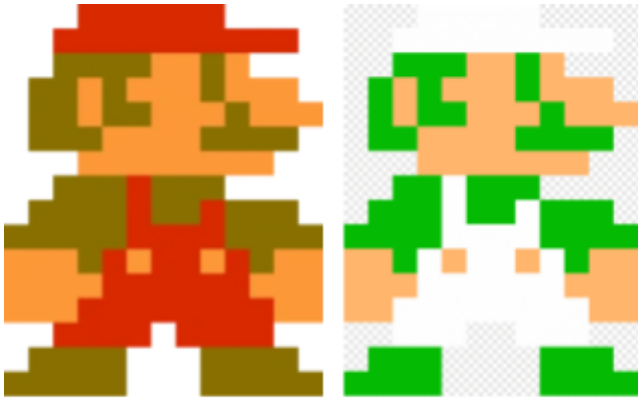
Palettes

La NES nous permet de stocker jusqu'à 4 palettes différentes par tileset. Chacune contient quatres couleurs dont une commune à toutes les palettes pour représenter la transparence. Il est possible d'utiliser plusieurs fois la même palette sur plusieurs tiles différentes. Cependant, une tile ne peut utiliser les couleurs que d'une seule palette.

Plage adresses	Utilisation
$3F00	Couleur de background
$3F01-$3F03	Background palette 0
$3F05-$3F07	Background palette 1
$3F09-$3F0B	Background palette 2
$3F0D-$3F0F	Background palette 3
$3F11-$3F13	Tile palette 0
$3F15-$3F17	Tile palette 1
$3F19-$3F1B	Tile palette 2
$3F1D-$3F1F	Tile palette 3

Les adresses des palettes

Le fait de stocker en CHR permet à la NES d'utiliser des palettes swap (Changement de palette) et d'appliquer des palettes de couleurs différentes à une même tile.



Exemple de palette swap avec Mario et Luigi

Background layer

Le background mesure 512*480 pixels qui contient des tiles statiques. Le **viewtable** (ce qui est affiché à l'écran) est plus petit que ça. Cela facilite la création d'un effet de scrolling. Pour sauvegarder de la mémoire on groupe les tiles en **blocks** de 16*16 pixels, ce qui permet d'avoir une adresse mémoire au lieu de quartes différentes.

Les **Nametables** (stockées en VRAM) spécifient les tiles affichées sur le background. Le **PPU** peut regarder quatre **Nametables** de 1024 bytes à la fois. Cependant, il n'y a que 2KB de VRAM disponible, ce qui veut dire que seulement deux Nametables peuvent être stockées, sans ajout de VRAM dans la cartouche. Il y a un autre type de **Nametable** : les **AttributeTable** qui stockent quelle palette est utilisée pour chaque **Block**.

```
DC.B
$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$
b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5,$b4,$b5
```

Exemple de line d'une nametable

Représentation graphique du code ci-dessus

Pour se faciliter la tâche il est plus simple d'utiliser du binaire pour stocker les AttributeTable.

```
; 00 = palette 1
; 01 = palette 2
; 10 = palette 3
; 11 = palette 4
DC.B %11100100,%11111111,%11111111,%11111111,%11111111,%11111111,%11111111,%11111111
```

Sprite layer

Les sprites sont les tiles qui peuvent bouger sur l'écran. Ils peuvent se superposer, apparaître devant ou derrière le **background layer**. C'est la viewtable qui décide de qui doit s'afficher sur quel plan selon la **priorité**. C'est le même concept que les calques dans Photoshop.

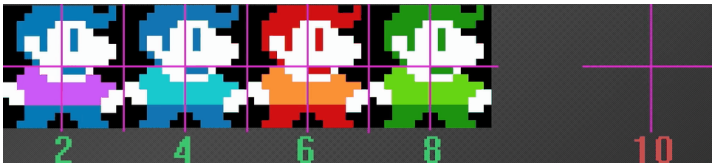
La table Object Attribute Memory (OAM) spécifie quelles tiles seront utilisées comme sprites. En plus de l'index chaque entrée contient une position (x,y) et d'autres attributs (**Palette**, **priorité** et le **Flip flag**). Cette table est stockée en 256 bytes DRAM dans le **PPU**. Le **PPU** est donc capable d'afficher 64 sprites par frame mais seulement 8 sur la même scanline.

Bit	But
7	Flip verticale (si 1)
6	Flip horizontal (si 1)
5	Priorité (si 1 derrière le background)
4-2	Non utilisé
1-0	Palette

Octet d'attribut

Flickering

Une autre limitation de la NES est qu'il n'est pas possible d'afficher plus de 8 sprites par scanline. c'est à dire que sur la même ligne il n'est pas possible d'afficher plus de 8 sprites, le 9eme n'est simplement jamais affiché. Pour contrer cette limitation certains jeux utilisent le *Flickering* qui consiste à afficher les sprites sur différentes frames. Si on souhaite afficher 9 sprites sur la même scanline un des sprites disparaîtrait toutes les 9 frames. Le taux de rafraîchissement étant compris entre 50 et 60 Hz le *Flickering* n'est quasiment pas visible grâce a la persistance rétinienne, mais plus on augmente le nombre de sprites plus les sprites restent "éteints" longtemps et donc on augmente les chances de voir le trucage. Cette technique est présentée ici car elle est importante dans le développement sur NES mais nous ne la développerons pas car c'est une technique avancée.



Démonstrations du Flickering

Et concrètement ?

- Effaçage des attributs Première chose à vérifier c'est que le type de cartouche soit bien sur 1 dans le header pour les émulateurs.
- Charge les palettes

```
 LDA #$3F    ; On positionne le registre -> ici poids fort
 STA PPUADDR ;   d'adresse du PPU
 LDA #$00    ;   à la valeur $3F00 -> poids faible
 STA PPUADDR

 LDX #3      ; Initialise X à 0
- LDA palette,X  ; On charge la Xième couleur (et pas la xieme palette)
 STA PPUDATA    ; pour l'envoyer au PPU
 INX         ; On passe à la couleur suivante -> ++x
 CPX #32     ; Et ce, 32 fois
 BNE -
```

- Après l'effaçage des attributs remplir le PPU avec les nametables contenues dans le fichier *nametables.asm*

```
 LDA PPUSTATUS  ; Resynchronisation
 LDA #$20     ;   On copie maintenant
 STA PPUADDR    ;     vers l'adresse $2000
 LDA #$00
 STA PPUADDR

 LDA #<nametables ; On initialise notre pointeur
 STA pointer    ;    avec le début des données
 LDA #>nametables ;    nametables + attributs
 STA pointer+1

 LDX #0         ; X = compteur de pages de 256 octes
 LDY #0         ; Y = décalage dans une page
- LDA (pointer),Y   ; On récupère la Yième donnée
 STA PPUDATA      ;   que l'on transmet au PPU
 INY            ; Passage à la donnée suivante
 BNE -          ; Jusqu'à Y = 256 (== 0)
 INC pointer+1    ; Sinon on incrémente le poids fort du pointeur
 INX            ; Et on passe à la page suivante
 CPX #8         ; Pendant 8 pages (8 * 256 = 2 Ko)
 BNE -

 BIT PPUSTATUS ; Resynchronisation
- BIT PPUSTATUS ; On attend une dernière fois
 BPL -
```

- Juste après la mainloop: include le fichier *palette.asm* et *nametable.asm*

```
.include "palette.asm"
.include "nametables.asm"
```

- Include le *.chr* à la toute fin du fichier

```
INCBIN "mario.chr"
```

- Afficher un sprite :

```
 LDA #0
  STA OAMADDR ; On se place dans la mémoire dédié aux sprite du PUU

  LDA #128    ; Position en Y (en pixel)
  STA OAMDATA
  LDA #$32    ; Adresse de la tile (en hex)
  STA OAMDATA
  ; 7 : Flip verticale (si 1)
  ;6 : Flip horizontal (si 1)
  ;5 : Priorité (si 1 derrière le background)
  ;4-2 : Non utilisé
  ;1-0 : Palette
  LDA #%00000000       ; La palette utilisé
  STA OAMDATA
  LDA #128    ; Position en X (en pixel)
  STA OAMDATA

  ;; 128 en x et 128 en y = au centre de l'écran
```

Plus de précision dans les vidéos

Vidéo : Présentation de YY CHR

Vidéo : Présentation de Nes Screen Tool

Animations

Dans la partie précédente, nous sommes parvenus à afficher un background statique sur notre écran puis nous avons ajouté un sprite indépendant. Notre objectif dans ce chapitre sera de réussir à animer ce sprite.

Organiser le tileset

Pour ce cours, nous allons commencer avec une animation basique composée de deux sprites qui s'intervertissent lorsque le personnage se déplace. Dans l'exemple que nous utilisons, les 4 tiles qui composent l'animation de base de notre personnage sont situées aux positions $32, $33, $34 et $35 sur notre tileset. Les tiles de l'animation suivante se trouvent juste après, aux positions $36, $37, $39 et $39. Cela va nous permettre de changer l'animation simplement en utilisant une valeur de décalage que l'on va ajouter au moment d'aller chercher le sprite à afficher.

Animer la marche

Dans un premier temps, nous ajoutons 2 variables au début de notre programme. La première variable, mario_anim, va nous servir d'identifiant pour l'animation à afficher. Comme nous avons deux animations, on peut les représenter respectivement par les valeurs 0 et 1. Ainsi, si la variable mario_anim vaut 0, on affiche la première animation et si elle vaut 1, on affiche la seconde animation. On pourrait théoriquement appliquer la même logique avec plus d'animations. Avec seulement 2 animations on peut se permettre d'incrémenter le compteur continuellement et ne regarder que le premier bit qui va varier entre 0 et 1. Par contre, avec plus d'animations, on aura besoin que la valeur de compteur fasse des aller retour si on ne veut pas que l'animation ait un rendu de boucle étrange.
La seconde variable, current_anim, va contenir la valeur de décalage à ajouter lorsque nous irons charger les sprites. Dans notre cas, elle vaudra 0 lorsqu'on voudra charger l'animation de base, et 4 pour charger l'animation suivante.

```
mario_anim    DS.B 1  ; le compteur qui nous indique quelle animation utiliser
current_anim  DS.B 1  ; valeur du décalage pour chercher les sprites de l'animation en cours
```

Ensuite nous allons incrémenter le compteur à chaque déplacement du personnage, donc dans la partie du code où l'on modifie l'offset.

```
INC mario_anim ; on incrémente le compteur d'animation à chaque mouvement
```

On va ensuite effectuer le calcul du décalage et la recherche de sprite dans draw_mario :

```
LDA mario_anim ; On charge le compteur d'animation
```

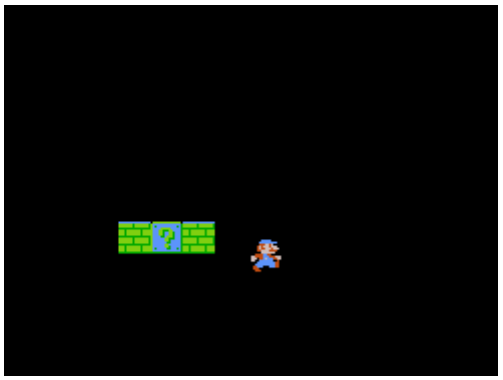
```
LSR          ; Ce décalage permet de diviser par deux la fréquence de mise à jour de l'animation
             ; On peut en ajouter plusieurs si l'animation est encore trop rapide
CLC          ; On remet à 0 le Carry Flag au cas où le décalage
AND #1       ; On ne garde que la valeur 0 ou 1
ASL                                          ; Comme on veut décaler notre recherche de 4 tiles, on multiplie le compteur par 4
ASL          ; Ainsi si le compteur vaut 0, elle ne change pas et on ira toujours chercher le sprite de base
             ; Si le compteur vaut 1, on le multiplie deux fois de suite pour créer une multiplication par 4
STA current_anim ; On stocke cette valeur de décalage
```

Il ne nous reste plus qu'à ajouter le décalage à chaque fois que l'on va chercher un sprite du personnage. Il suffit pour cela d'utiliser un ADC après le chargement de l'adresse du sprite.

```
LDA #$32    ; Adresse de la tile (en hex)
ADC current_anim ; Ajout du décalage
```

Il faut prendre en compte le fait que cette méthode fonctionne grâce à la façon dont est créé le tileset et n'est pas toujours la bonne solution. Par exemple avec le personnage de Mario, la 3ème animation ne contient pas de nouveaux sprites pour le visage. De ce fait, le décalage n'est pas constant. Il faut alors alors trouver une alternative et parfois il faudra faire du cas par cas.

Dans notre cas, le résultat de l'animation donne ceci :



Résultat de l'animation

Changer la direction

Actuellement, notre sprite ne possède qu'une animation de marche vers la droite quel que soit sa direction. Nous allons maintenant modifier le code pour pouvoir mettre à jour le sprite lorsque le personnage se retourne vers la gauche.

Nous avons vu dans le cours sur les tilesets que chaque sprite était accompagné d'un octet d'attribut dont les bit 6 et 7 permettent d'inverser le sprite horizontalement ou verticalement. Cela nous permet de gagner de l'espoir mémoire en stockant une seule fois une tile et en la retournant simplement en même temps que le personnage. Cependant il faut bien prendre en compte que cette inversion s'effectue à l'échelle d'une tile et non du sprite complet. Si ce dernier est composé de plusieurs tiles et qu'on se contente de les inverser, on peut donc se retrouver avec des résultats étranges comme dans l'exemple suivant :

Flip naïf du sprite

Pour éviter cela, il faut donc penser à inverser l'ordre de chargement des sprites. Dans notre exemple relativement simple, nous allons vérifier l'état de la direction et utiliser un branchement en fonction de cet état. Si la direction est à 0 (ce qui signifie que le personnage va vers la droite), on continue d'exécuter le code que nous avions jusque là. Si la direction vaut 1 (ce qui signifie que le personnage va vers la gauche) on modifie l'ordre d'exécution. Le code sera alors très similaire, à la différence que certaines adresses pour charger les sprites seront interverties et que l'octet d'attribut sera #01000000. Ainsi, la tuile sera inversée horizontalement.

Après avoir calculé le décalage du sprite, on effectue le test sur la direction

```
LDA direction
CMP #0
BNE animation_left
CLC
```

Si la comparaison n'est pas bonne, on change l'ordre d'exécution à l'adresse de animation_left. Sinon on continue. Il faut cependant bien penser à rajouter une ligne CLC dans les deux cas, pour réinitialiser le Carry Flag après la comparaison.

```
animation_left:
  CLC
  LDA #0
  STA OAMADDR

  LDA #128
  STA OAMDATA
  LDA #$33
  ADC current_anim
  STA OAMDATA
  LDA #%01000000
  STA OAMDATA
  LDA #128
  STA OAMDATA

  LDA #128    ; Y
  STA OAMDATA
  LDA #$32
  ADC current_anim
  STA OAMDATA ; Tile
  LDA  #%01000000
  STA OAMDATA ; Attr
  LDA #136    ; X
  STA OAMDATA

  LDA #136    ; Y
  STA OAMDATA
  LDA #$35
  ADC current_anim
  STA OAMDATA ; Tile
  LDA  #%01000000
  STA OAMDATA ; Attr
  LDA #128    ; X
  STA OAMDATA

  LDA #136    ; Y
  STA OAMDATA
  LDA #$34
  ADC current_anim
  STA OAMDATA ; Tile
  LDA  #%01000000
  STA OAMDATA ; Attr
  LDA #136    ; X
  STA OAMDATA
  RTS
```

Et voilà, avec code nous pouvons animer la marche de notre personnage et l'orienter dans la bonne direction !

Animation orienté

Contrôleurs, périphériques

Les contrôles sur la NES

Les instructions correspondant aux contrôles des manettes de la NES sont envoyées sur les adresses $4016 et $4017. Ces adresses correspondent respectivement aux manettes 1 et 2; dans le développement d'un jeu à 1 joueur on aura donc seulement besoin de l'adresse $4016.

Pour un contrôleur, on a accès à 8 bits correspondant aux 8 boutons d'une manette NES :

bit	7	6	5	4	3	2	1	0
bouton	A	B	Select	Start	Haut	Bas	Gauche	Droite

Une manette de NES

On n'accède pas à ces 8 bits en lisant un octet de données. Pour lire l'état d'un Joypad (manette) de la NES, il faut lire successivement à la même adresse (donc $4016 pour le Joypad 1), pour chaque bouton, donc 8 fois. Dans l'ordre, on lit les boutons A, puis B, Select, Start, Haut, Bas, Gauche et enfin Droite. A chaque lecture on reçoit un bit à 1 si le bouton est pressé et 0 sinon. Pour indiquer qu'on souhaite récupérer l'état du Joypad, on commence d'abord par envoyer à l'adresse $4016 un 1 suivi d'un 0, ce qui permettra de faire les 8 lectures ensuite. En code assembleur, ça donne ça :

```
JOY1 EQU $4016      ; raccourci vers l adresse $4016

; On signale qu on va lire le joypad (1 puis 0 en $4016)
LDA #1
STA JOY1
LDA #0
STA JOY1

; On peut commencer a lire les statuts des boutons
LDA JOY1    ; bouton A
LDA JOY1    ; bouton B
LDA JOY1    ; bouton Select
LDA JOY1    ; bouton Start
LDA JOY1    ; bouton Haut
LDA JOY1    ; bouton Bas
LDA JOY1    ; bouton Gauche
LDA JOY1    ; bouton Droite
```

A chaque lecture, l'état du bouton est stocké dans l'accumulateur A, on peut donc par exemple le lire en faisant une opération logique simple comme "A ET 1" : si A vaut 1 le résultat vaudra 1, sinon 0. Exemple :

```
  ; ne pas oublier le bloc pour réinitialiser le joypad avant

  ; puis on lit (bouton A)
  LDA JOY1 ; JOY1 est dans A
  AND #1   ; compare A et #1 -> met le zero flag à 0 si A AND 1 = 1, 1 sinon
  BNE boutonA ; branche si le zero flag est à 0 (donc si le bouton est pressé)

boutonA:
  ; code executé quand le bouton A est pressé
```

La Four Score

La NES Four Score est un périphérique permettant de connecter jusqu'à quatre manettes sur la console, alors que celle-ci est normalement limitée à deux manettes.

La NES Four Score

On a vu que la NES ne disposait que de deux adresses pour lire les entrées des manettes 1 et 2, ce qui nous amène à nous poser la question suivante : comment récupérer les informations des quatre manettes ?

L'astuce de la NES Four Score est de stocker les informations de deux manettes sur une adresse, de la même manière que les boutons d'une manette sont lus à la même adresse. Ainsi, on pourra récupérer les informations des manettes 1 et 3 sur l'adresse $4016, et les informations des manettes 2 et 4 sur l'adresse $4017. Le code suivant permet ainsi de récupérer les informations de quatre manettes connectées sur une Four Score, en lisant tous les boutons de la manette 1, puis tous les boutons de la manette 3, et de même pour les deux autres :

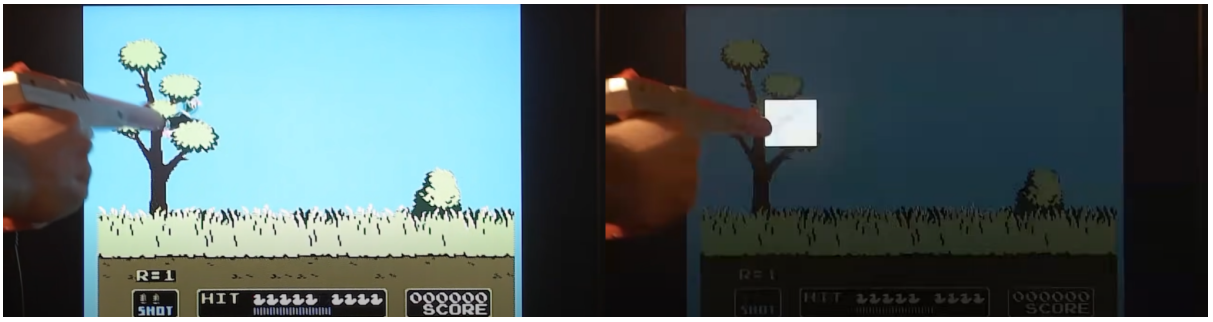
```
JOY1 EQU $4016      ; raccourci vers l adresse $4016
JOY2 EQU $4017      ; raccourci vers l adresse $4017

; On signale qu on va lire le joypad 1 (1 puis 0 en $4016)
LDA #1
STA JOY1
LDA #0
STA JOY1

; On peut commencer a lire les statuts des boutons des manettes 1 et 3
LDA JOY1     ; manette 1, bouton A
...
LDA JOY1     ; manette 1, bouton Droite
LDA JOY1    ; manette 3, bouton A
...
LDA JOY1    ; manette 3, bouton Droite


; On signale qu on va lire le joypad 2 (1 puis 0 en $4017)
LDA #1
STA JOY2
LDA #0
STA JOY2

; On peut lire les boutons des manettes 2 et 4
LDA JOY2     ; manette 2, bouton A
...
LDA JOY2     ; manette 2, bouton Droite
LDA JOY2    ; manette 4, bouton A
...
LDA JOY2    ; manette 4, bouton Droite
```

Le NES Zapper

Un dernier périphérique que l'on vous présentera rapidement dans ce cours est le NES Zapper, qui se présente sous la forme d'un pistolet branché à la console. Celui-ci est utilisé dans certains jeux pour tirer (virtuellement) sur l'écran. Il est sorti en deux versions, ci-dessous en gris, la version originale sortie au Japon, et en orange la version occidentale :

Le NES Zapper ou Famicon Light Gun au Japon

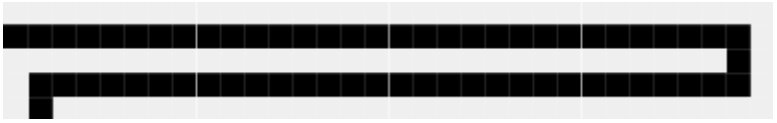
Le NES Zapper fonctionne de la façon suivante : au moment où le joueur appuie sur la détente, la NES affiche un écran noir avec un carré blanc à la place des cibles pendant une image. Un capteur sur le pistolet identifie alors les différences d'intensité sur l'écran (ici, sur le jeu Duck Hunt) :



Un joueur tire avec le NES Zapper, avant et pendant le tir

Gérer les collisions

Avec les parties précédentes, nous pouvons désormais déplacer un sprite en utilisant les contrôles d'une manette, ou animer ce sprite. Cependant il passera à travers tous les obstacles qu'il rencontrera, on a donc besoin d'apprendre à gérer les collisions.
On a besoin pour cela de savoir à un endroit précis de notre décor s'il y a un mur ou un quelconque obstacle. Une façon de faire est de représenter les murs d'une map dans une autre table, avec un bit à 0 pour indiquer que la case est vide, et 1 pour s'il y a un mur. Par exemple, on veut utiliser le décor suivant :



En blanc, les murs

On peut le représenter en mémoire de cette façon :

En blanc, les murs

Ce qui donne, en code :

```
wall_mask:
 DC.B %11111111,%11111111,%11111111,%11111111 ; ligne 0
 DB.B %00000000,%00000000,%00000000,%00000001 ; ligne 1
 DC.B %11111111,%11111111,%11111111,%11111101 ; ligne 2
 DC.B %10000000,%00000000,%00000000,%00000001 ; ligne 3
 DC.B %10111111,%11111111,%11111111,%11111111 ; ligne 4
```

Dans la suite du code il va falloir accéder à la bonne case pour un endroit donné. On pourra trouver l'octet qui correspond à une case en (x, y) en prenant celui d'adresse de la table y * 4 + x / 8. Le bon bit sera alors celui en x modulo 8 ième position dans l'octet sélectionné, en partant de la gauche.
Pour effectuer un équivalent de l'opération modulo, en assembleur, on peut utiliser un masque et effectuer une opération AND : par exemple `%10000000 AND valeur` renverra la valeur du bit à la première position de `valeur`. Il peut donc être utile de stocker en amont les 8 masques dont on risque d'avoir besoin, pour accéder aux 8 bits de l'octet. Puisqu'on pourra avoir souvent besoin d'accéder à chacun des bits d'un octet pour gérer les collisions, on pourra les stocker de cette façon :

```
bits:
 DC.B %10000000
 DC.B %01000000
 DC.B %00100000
 DC.B %00010000
 DC.B %00001000
 DC.B %00000100
 DC.B %00000010
 DC.B %00000001
```

On pourra alors vérifier si un mur est présent en position (position_x, position_y), de cette façon :

```
; on calcule y * 4
LDA position_y
ASL
ASL
STA tmp     ; tmp contient maintenant position_y * 4
; puis x / 8
LDA position_x
LSR
LSR
LSR
CLC
; et on ajoute les deux
ADC tmp     ; tmp contient maintenant position_y * 4 + position_x / 8
; on peut maintenant récupérer l'octet numéro tmp de nos murs
TAX
LDA wall_mask,X
STA tmp

; maintenant il nous reste à choisir le bon bit
LDA position_x
AND #7   ; %00000111 -> position_x modulo 8
TAX
LDA bits,X
AND tmp  ; le flag Z = 1 s'il n'y a pas de mur
```

Remarque : ASL et LSR permettent respectivement de décaler les bits d'un cran vers la gauche et vers la droite, ce qui revient à faire une multiplication et une division par 2. Donc 2 fois ASL = multiplication par 4 et 3 fois LSR = division par 8.

Son & musique

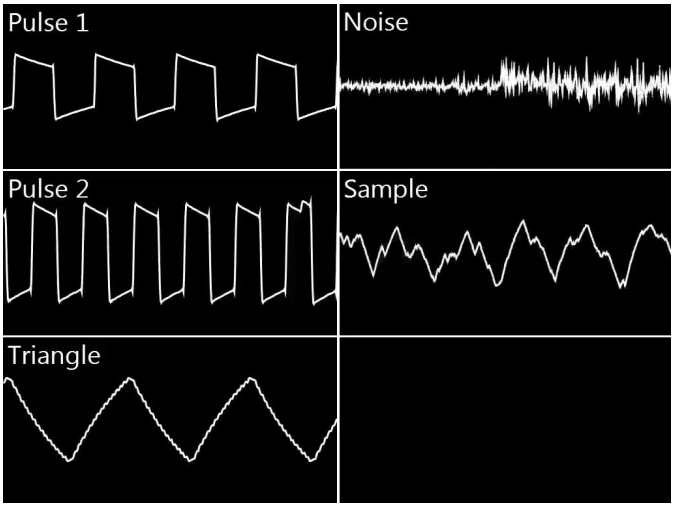
Ajoutez du son à votre programme

Présentation de l'APU

La puce de génération du son de la Nintendo NES est une puce 2A03. Cette puce produite par *Ricoh* au Japon est une puce qui a été construite spécialement pour les besoins de la NES et est basée sur la puce 6502. Avec l'aide de *MOS Technology*, Nintendo a réussi à modifier le fonctionnement d'une puce 6502 pour en adapter son fonctionnement à de la gestion d'audio. D'ailleurs, celle-ci est plus communément appelée pAPU ou "pseudo Audio Processing Unit" car celle-ci n'est en réalité pas simplement destinée au traitement du son mais aussi d'autres informations, comme la gestion des inputs manette.

La Nintendo NES dispose de 5 channels audios :

- 2 canaux produisant un son rectangulaire appelés Pulse
- 1 canal produisant un son triangulaire appelé Triangle
- 1 canal produisant un son aléatoire (du bruit) appelé Noise
- 1 canal pouvant lire des fichiers audio appelé DMC pour Delta Modulation Channel



Représentation des channels audios

Voici des exemples de signaux produits par la puce pAPU 2A03. Comme vous pouvez le voir, les capacités du 2A03 étant moindres comparées à nos machines actuelles, le signal carré et le signal triangulaire ne sont pas parfaits. Ce sont par ailleurs ces imperfections qui rendent le son de la NES si particulier et c'est une des particularités principales que les producteurs de musique "8-bit" ou dit de Chiptune vont rechercher.

Etant construits sur la même base matérielle, le 6502 et le 2A03 fonctionnent de la même manière d'un point de vue programmation. En effet, tout comme pour réaliser des calculs ou afficher des éléments à l'écran, pour générer du son il faut utiliser le même jeu de commandes Assembleur.

"Mais alors comment est ce que je contrôle le 2A03 et pas le 6502 dans mon code ASM6502 ?"

Tout se joue sur une plage d'adresses mémoire précise. Ce sont les adresses de $4000 à $4015. Ce jeu de 21 adresses est tout ce qu'il vous faut pour faire du son !

Les adresses sont réunies en 5 groupes + 1 adresse. 1 groupe de 4 adresses par canal, et une adresse supplémentaire.
Pour contrôler un canal, la NES dispose d'en moyenne 10 contrôles différents pour manipuler le son et le moduler.

"10 contrôlés par canaux mais 4 adresses seulement ?"

Et oui ! tous les contrôles n'ont pas forcément besoin d'un octet pour transmettre leur information ! Par exemple, l'instruction d'activation ou non du son peut être résumée à un booléen. Soit un seul bit. C'est pourquoi chaque adresse peut contenir plusieurs informations sur un seul octet. Voici un exemple extrait de l'annexe listant toutes les adresses :

Adresse	Fonction	Bits	Détails
$4000	Canal APU 1 (Square) Volume et Decay (W)	DDLEVVVV	**V**olume, **E**nveloppe constante, compteur de **L**ongueur, **D**uty cycle

A l'adresse $4000 commence la première adresse de contrôle de l'APU. C'est d'ailleurs la première adresse de contrôle du canal 1, le premier des deux canaux qui génèrent un son carré.
Comme vous pouvez le voir, l'octet est divisé en 4 parties que l'on nommera avec des lettres pour vous aider à vous repérer :
DD L E VVVV

V : Contrôle le **volume**, sur 4 bits. 2_4 = 16 : 16 valeurs sont donc possibles pour ce paramètre.
E: **Enveloppe** sonore constante. Permet de demander à la NES de maintenir le volume. Valeur booléenne.
L : Permet de demander d'interrompre le compteur de **longueur**. Valeur booléenne.
D : **Duty Cycle**. Permet d'altérer la forme du signal carré pour en changer sa texture (l'aspect sonore du son). Sur 2 bits. 22 = 4.
4 valeurs sont possibles :

%00 12.5%,
%01 25%,
%10 50%,
%11 75%.

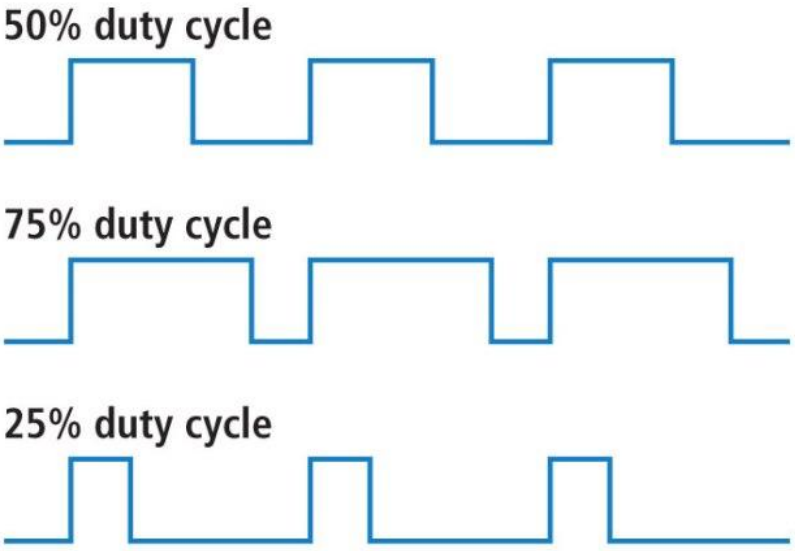


Schéma de l'effet du Duty Cycle sur le signal

Comme vous pouvez le voir, en 1 octet on peut envoyer 4 instructions à l'APU. Tout se limite donc à construire cet octet et l'écrire en **$4000** pour que l'APU l'interprète.

Premier son

Pour produire notre premier son, nous allons utiliser le canal Pulse 1 (le premier canal carré).
Voici la structure des 4 adresses (de $4000 à $4003) :
$4000 → DDLEVVVV
$4001 → AUUUDBBB
$4002 → LLLLLLLL
$4003 → CCCCCHHH

Voici les valeurs que nous choisirons pour $4000 (DDLEVVVV)
V (volume) : **%1111** (16) pour produire un son au volume maximum
E (enveloppe constante) : **%1** pour activer un volume constant
L (halte du compteur de longueur) : **%0** cette fonctionnalité ne nous est pas utile.
D (Duty Cycle) : **%10** On va choisir un Duty Cycle à 50% pour avoir un signal régulier.

en construisant notre octet final, on obtient donc : **%10001111**

Pour $4001 on ne va utiliser aucune des options de celle-ci. on va le set à : %00000000
On passe maintenant à **$4002** (LLLLLLLL):
L (Fréquence) : L correspond aux 8 bits les plus bas qui correspondent au timer qui va contrôler la fréquence de l'audio. En effet, ce paramètre est envoyé sur 11 bits, les 8 plus faibles sur $4002 et les 3 plus hauts sur $4003.
En musique, l'une des choses les plus importantes c'est de jouer juste. La hauteur d'une note (ou est ce qu'elle est aigüe ou grave) est déterminée en Hertz (Hz).
En d'autres termes, un son n'est qu'un motif sonore répété à une fréquence précise.
Le calcul de la fréquence d'update est dépendant de la vitesse d'actualisation de la puce et donc varie en fonction de la région (PAL ou NTSC). En annexe vous trouverez une formule pour calculer précisément la valeur à envoyer à l'APU pour obtenir une note exacte.
Dans mon cas, je veux produire un La à 440Hz, la valeur sera donc 253. soit **$00011111101** sur 11 bits.

Pour finir, l'adresse $4003 (DDDDDHHH)
D (Durée) : Cette partie correspond aux bits du compteur de durée. Plus la valeur est longue plus la note va durer longtemps. Dans notre cas, on veut une note la plus longue possible. On va donc mettre tous les bits à 1.
H (Fréquence) : les 3 bits les plus hauts
La valeur finale sera donc **%11111000**

Maintenant que nous avons nos 4 valeurs de prêtes, il ne nous reste plus qu'à les appeler.
Vous pouvez déclencher un son à un moment particulier, par exemple lors d'un appui sur un bouton ou pendant une collision.
Dans notre cas on veut déclencher le son au moment du chargement du programme et ce avant la boucle principale.

Code Assembleur

Pour cela nous devons charger la valeur dans A puis qu'on stocke le contenu de A dans les adresses respectives.

Juste avant le premier appel de mainloop on ajoutera donc ce code :

```
;      DDLEVVVV
  lda #%10001111
  sta $4000
;      AUUUDBBB
  lda #%00000000
  sta $4001
;      LLLLLLLL
  lda #%11111101
  sta $4002
;      CCCCCHHH
  lda #%11111000
  sta $4003
```

Si vous compilez ce code, vous verrez qu'aucun son n'est produit. C'est tout simplement parce qu'on a pas encore demandé à l'APU d'activer le canal Pulse 1. Il ne produit donc aucun son. Il faut donc ajouter avant ou après nos instructions à l'APU la ligne suivante :

```
  ;54321
  lda #%00000001;dans A on stocke un octet demandant l'activation du canal 1
  sta $4015; on envoie le contenu de A dans la 21ème adresse de l'APU
```

Une fois compilé.... Ça y est ! On a du son !

Comment calculer la valeur à envoyer à l'APU pour la fréquence d'une note

Comme vu dans le document précédent, l'APU a besoin de recevoir une valeur pour faire varier la fréquence d'un son (mesurée en Hertz).
Mais avant de déterminer quelle valeur on doit lui envoyer, il faut d'abord qu'on comprenne comment fonctionne la fréquence :

$$t = 1/f$$
$$f = 1/t$$

Comme vous pouvez le voir, la fréquence est directement liée à t, le temps écoulé entre deux vibrations.
S'il s'écoule 0.5s entre deux vibrations :

$$f = 1/0.5 = 2Hz$$

La fréquence est donc de 2Hz.
Plus **t** est grand, plus f est petit et inversement. En d'autres termes, un son grave aura un T plus grand qu'un son aigu.

La fréquence étant perçue par la NES comme un intervalle de temps et non comme une fréquence, c'est donc T qu'il faut qu'on lui transmette.

Dans le cas de la NES, voici le calcul exact pour calculer notre T :

$$T = (CPU/(16 * f)) - 1$$

Pour un processeur NTSC, CPU=1789773.0 CPU=1789773.0 alors que pour un processeur PAL, CPU=1662607.0

Voici un exemple de fonction en C++ pour déterminer la valeur de T facilement:

```
int frequencyToNes(float frequency, bool isPAL){
   float cpu = isPAL ? 1662607.0 : 1789773.0;
   return (int)((cpu/(16*frequency)) -1);
}
```

Adresses mémoires liées à l'APU

Adresse	Fonction	Bits	Détails
$4000	Canal APU 1 (Square) Volume et Decay (W)	DDLEVVVV	**V**olume, **E**nveloppe, compteur de **L**ongueur, **D**uty cycle
$4001	Canal APU 1 (Square) Balayage (W)	AUUUDBBB	**B**alayage, **D**irection, fréquence d'**U**pdate, **A**ctivation
$4002	Canal APU 1 (Square) Fréquence (W)	LLLLLLLL	Octet de fréquence **L**
$4003	Canal APU 1 (Square) Longueur (W)	CCCCCHHH	Octet de fréquence **H**, registre du **C**ompteur de chargement
$4004	Canal APU 2 (Square) Volume et Decay (W)	DDLEVVVV	**V**olume, **E**nveloppe, compteur de **L**ongueur, **D**uty cycle
$4005	Canal APU 2 (Square) Balayage (W)	AUUUDBBB	**B**alayage, **D**irection, fréquence d'**U**pdate, **A**ctivation
$4006	Canal APU 2 (Square) Fréquence (W)	LLLLLLLL	Octet de fréquence **L**
$4007	Canal APU 2 (Square) Longueur (W)	CCCCCHHH	Octet de fréquence **H**, registre du **C**ompteur de chargement
$4008	Canal APU 3 (Triangle) Compteur linéaire (W)	DLLLLLLL	**D**ésactivation de l'horloge, registre de chargement du compteur **L**inéaire
$4009	Canal APU 3 (Triangle) Non utilisé (-)	--------	Inutilisé
$400A	Canal APU 3 (Triangle) Fréquence (W)	LLLLLLLL	Octet de fréquence **L**
$400B	Canal APU 3 (Triangle) Longueur (W)	CCCCCHHH	Octet de fréquence **H**, registre du **C**ompteur de chargement
$400C	Canal APU 4 (Noise) Volume et Decay (W)	DDLEVVVV	**V**olume, **E**nveloppe, compteur de **L**ongueur, **D**uty cycle
$400D	Canal APU 4 (Noise) Non utilisé (-)	--------	Inutilisé
$400E	Canal APU 4 (Noise) Fréquence (W)	LLLLLLLL	Octet de fréquence **L**
$400F	Canal APU 4 (Noise) Longueur (W)	CCCCCHHH	Octet de fréquence **H**, registre du **C**ompteur de chargement
$4010	Canal APU 5 (DMC) Mode de lecture et Fréquence DMA (W)	IB--FFFF	Activation **I**rq, **B**oucle, **F**réquence
$4011	Canal APU 5 (DMC) Registre de chargement du compteur Delta (W)	-DDDDDDD	Compteur **D**elta ou **D**onnées PCM 7bits
$4012	Canal APU 5 (DMC) Registre de chargement d'adresse (W)	AAAAAAAA	**A**dresse = $C000+(A'$40)
$4013	Canal APU 5 (DMC) registre de longueur (W)	LLLLLLLL	Longueur = (L'$10)+1 octet
$4015	DMC/IRQ/statut du compteur de longueur/ registre d'activation du canal de son (RW)	DF-54321	Statut irq du **D**mc / statut de la **F**rame irq. Canal 12345 on

Adresses mémoires du pAPU

Vidéo : Comment réaliser une musique pour NES avec Famitracker

vidéo : Comment intégrer une mélodie de Famitracker en Assembleur avec Famitone2

Conclusion

Ça y est ! Nous avons vu ensemble les bases pour réaliser un jeu sur NES !

Comme vous avez pu le voir, le développement de jeux sur la Nintendo NES n'a rien à voir avec les méthodes de développement modernes. On est loin des Unity ou Unreal Engine, deux moteurs de jeux très répandus sur le marché et grâce auxquels un petit prototype de jeu peut être réalisé en une petite demi-heure. Là, sur la NES tout est différent. Toutes les fonctionnalités auxquelles nous étions habitués doivent être développées par nous. Comme vous l'avez vu durant ce cours, nous avons eu la chance de pouvoir utiliser des outils brillants développés par des génies de l'Assembleur 6502 et tout particulièrement de la NES comme Famitracker et Famitone2 pour l'audio, YY-CHR et NES Screen Tool pour le visuel ou encore FCEUX, un excellent émulateur NES. Ces outils n'étaient pas disponibles en 1983 lorsque les développeurs de l'époque devaient développer sur vim, sans coloration syntaxique, et encore moins avec tous les avantages que nous avons aujourd'hui. Chaque outil que nous avons dû utiliser, ils ont dû se les développer eux-même, et ce afin de nous faciliter la vie car comme vous avez pu le voir... Développer sur NES ça n'est pas une partie de plaisir !

Malgré la grande difficulté que représente le développement en ASM 6502 sur NES, nous avons pu voir ensemble de nombreux points.

- Nous avons commencé par nous familiariser avec la console, sur la partie électronique mais aussi sur la partie plus technique, afin de comprendre son fonctionnement.
- Nous avons ensuite dû mettre les pieds dans le plat et découvert l'assembleur 6502 avec par exemple ses contraintes et ses instructions, mais aussi toutes les nuances du développement de la NES. Des suites de cette introduction nous avons réalisé notre premier programme : un programme assez simple et pourtant complexe au premier abord qui permettait d'afficher Bonjour ! à l'écran.
- Ensuite, nous avons approfondi le fonctionnement de l'affichage sur la NES jusqu'à pouvoir afficher un arrière-plan et un personnage, que nous avons rapidement réussi à animer et à contrôler.
- Enfin, nous avons appris à contrôler l'APU pour faire du son et nous avons réussi à ajouter de la musique à notre jeu.

Tous ces points sont les points de base nécessaires pour pouvoir se lancer dans le développement sur NES. Une fois que vous aurez maîtrisé celles-ci , pourquoi ne pas continuer et aller encore plus loin ?

Nous vous suggérons par exemple d'essayer d'optimiser votre code pour potentiellement, charger un niveau complet contenu sur plusieurs écrans !

Vous pourriez aussi utiliser Famitone2 afin de déclencher des effets spéciaux en même temps que la musique.

Enfin, ce serait intéressant de réussir à intégrer un élément collectable ou même un ennemi !

Nous vous avons fait découvrir les bases du développement sur la NES mais c'est un monde entier qui vient de s'ouvrir à vous !

Travaux pratique

L'objectif de ce TP est de pouvoir déplacer un sprite horizontalement et verticalement à l'aide des boutons directionnels (Haut, Bas, Droite, Gauche) de la manette de la NES, sur un décor. Ce TP sera organisé en plusieurs étapes pour vous guider dans le développement, puis une correction sera proposée.

Hello world

Pour commencer, nous vous proposons un premier fichier source qui contient une base de programme avec seulement le nécessaire pour débuter ce TP. Votre première tâche consiste donc à assembler ce fichier pour pouvoir le lancer sur un émulateur, comme vu dans la vidéo d'installation des logiciels. Nous vous rappelons la commande pour compiler votre fichier source :

```
asm6 source.asm nomdujeu.nes
```

Afficher un background

Une fois la première compilation réalisée, changeons le background ! Pour cela il faudra importer quelques fichiers dans notre projet.

Tout d'abord vous devrez créer (IV - Graphismes - Présentation de YY CHR) ou récupérer le fichier CHR fourni dans le dossier ressource, puis l'importer dans le projet.

Ensuite, vous aurez besoin de palettes pour les couleurs de vos sprites. Vous pouvez également créer vos propres palettes (IV - Graphismes - Présentation de Nes Screen Tool) ou bien utiliser celles fournies dans les ressources.

L'étape suivante consiste à créer la nametable qui définit les tiles à charger sur l'écran. Encore une fois, vous pouvez créer des nametables de toute pièce (IV - Graphismes - Présentation de Nes Screen Tool) ou récupérer celle que nous vous proposons.

Enfin, Il faut charger les palettes et les nametables dans la mainloop. Il n'y a pas besoin de faire quoi que ce soit pour le CHR.

A la fin de cette étape, vous devriez pouvoir afficher un background sur votre écran.

Afficher un sprite

Pour cette étape, tout est déjà chargé en mémoire. Il faut simplement regarder l'adresse mémoire du sprite qui nous intéresse comme montré dans la vidéo IV -

Graphismes - Présentation de Nes Screen Tool) et le charger dans une sous routine et l'appeler dans la main loop

Scroll horizontal

Maintenant que vous avez choisi et afficher votre sprite, cette étape consiste à lire l'appui sur les touches gauche/droite et faire se déplacer le sprite horizontalement lorsque l'on appuie sur ces boutons (à droite lorsque l'on appuie sur le bouton droite, à gauche lorsque l'on appuie sur le bouton gauche). La partie de ce cours qui peut vous aider est *6 - Contrôleurs, périphériques*.

Pour cela, vous pouvez stocker la position du sprite en x en variable, et brancher sur un bloc de code si on appuie sur le bouton gauche. Dans ce bloc de code on décrémentera la valeur de cette position pour que le sprite affiche un pixel de plus à gauche. De même pour le bouton droit qui branchera sur un bloc de code qui va lui incrémenter la position du sprite en x.

Scroll vertical

Une fois le déplacement horizontal fait, vous pouvez vous occuper du deuxième axe et faire se déplacer le sprite verticalement lorsque l'on appuie sur le bouton Haut ou sur le bouton Bas.

De la même façon que pour le scroll horizontal, vous pouvez brancher sur deux nouveaux blocs de code qui changeront une variable de position du sprite en y. Attention cependant, pour le scroll vertical, puisque la position en haut à gauche de l'écran est en (0,0), on incrémente la position en y pour faire descendre le sprite et on la décrémenter pour faire remonter le sprite.

Animer le sprite

La touche finale va consister à animer le sprite pour avoir un rendu plus dynamique. Vous pouvez vous reportez au cours 5 sur les animations pour avoir des précisions sur le code à implémenter.

Si vous avez déjà créé votre propre sprite pour l'étape 3, vous pouvez également créer des animations personnalisées. Prenez garde cependant à la manière dont est rangée votre tileset. Dans ce TP nous allons partir du principe que les animations qui composent votre sprite sont rangées les unes après les autres. Par exemple, si votre sprite de base est composé des tuiles 32, 33, 34 et 35, alors l'animation suivante est composée des tuiles 36, 37, 38 et 39. Sinon vous pouvez toujours utiliser le fichier mario.chr fourni avec le TP.

Maintenant que les sprites sont prêts, vous pouvez créer l'animation dans le script. Pour cela, vous aurez besoin d'une variable de compteur, qui servira d'indicateur de l'animation en cours, ainsi que d'une variable de décalage, qui indiquera la valeur à ajouter à l'adresse de recherche des sprites. Ce décalage dépend de l'animation en cours (donc de la variable de compteur) et de la distance entre chaque tile dans le tileset. En reprenant l'exemple précédent, la tile du haut gauche de notre sprite de base (appelons ça l'animation 0) est à l'adresse 32. Pour afficher l'animation 1, il faut afficher la tile 36, il y a donc un décalage de 4 tile par rapport à la valeur de base. Si on ajoutait une animation 2 aux tuiles, 40, 41, 42 et 43, il y aurait alors un décalage de 8 tiles par rapport à celle de base. A chaque animation, on se décale de 4 valeurs supplémentaires. Ainsi l'adresse de la tile à afficher équivaut à : Adresse de base + (décalage * compteur).

Une fois que l'animation de marche est fonctionnelle, vous pouvez également faire en sorte que le sprite se tourne dans la bonne direction lorsqu'il se déplace. Selon la façon dont vous avez organisé votre code, vous pouvez stocker la direction dans une variable, ou utiliser les inputs. Vous devrez également manipuler l'octet d'attribut pour inverser les tiles tout en prenant garde à adapter les adresses à charger.

ANNEXE

Vidéos

Playlist des vidéos
https://youtube.com/playlist?list=PLEYOBYO-39QZSNw0jTOwaZjlwqDzJwsTC

Code source

Github : https://github.com/Ronan-senpi/NES-Base-Game-Dev

Sources

Blance, Andrew. (2020) An Introduction to 6502 Assembly and Low Level Programming. Codeburst.
https://codeburst.io/an-introduction-to-6502-assembly-and-low-level-programming-7c11fa6b9cb9

Maltais, Sylvain. (2016) Langage de programmation - Assembleur 6502. Gladir.
https://www.gladir.com/CODER/ASM6502/index.htm

David, Odin. (2019) Programmation avec le 6502. Connect.
https://connect.ed-diamond.com/Hackable/hk-031/programmation-avec-le-6502

David, Odin. (2020) Programmation avec le 6502 ; Découverte de la NES. Connect.
https://connect.ed-diamond.com/Hackable/hk-034/programmation-avec-le-6502-decouverte-de-la-nes

(2021) NesDev Wiki
https://wiki.nesdev.org

Parker, Christopher. (2019) Nerdy Night Mirror.
https://nerdy-nights.nes.science/#main_tutorial-0

(2009) Adressage mémoire sur NES et utilisation des mappers. T.R.A.F Wiki.
https://wiki.romhack.org/index.php?title=Adressage_m%C3%A9moire_sur_NES_et_utilisation_des_mappers

(2022) Nintendo Entertainment System. Wikipedia.
https://en.wikipedia.org/wiki/Nintendo_Entertainment_System

Copetti, Rodrigo. (2019) NES Architecture. Rodrigo's Stuff.
https://www.copetti.org/writings/consoles/nes/

Safiire (2019) Creating Sound on the NES
https://irkenkitties.com/blog/2015/03/29/creating-sound-on-the-nes/

Chibiakumas (2019) learn 6502 Assembly Lesson partie 26
https://www.chibiakumas.com/6502/platform3.php

jsr (2005) Famitracker.
http://famitracker.com/

Shiru (2010) Famitone2 library
https://shiru.untergrund.net/code.shtml