# CAMLNES

Émulateur de NES en OCaml

Projet long 2022-2023

Adrian Heouairi

# Introduction

- Objectif : programmer un émulateur de NES pour PC en OCaml en paradigme fonctionnel, sans mapper et sans support audio
- Console sortie à partir de 1983
- Émulateur = interpréter un programme fait pour un autre ordinateur
- L’émulateur programmé fonctionne, à une vitesse proche de la normale

# Introduction

- Compatibilité avec les jeux les plus connus utilisant le mapper 0 :
- Bomberman (USA).nes
- Donkey Kong (World) (Rev A).nes
- Donkey Kong Jr. (World) (Rev A).nes
- Excitebike (Japan, USA).nes
- Mario Bros. (World).nes
- Space Invaders (Japan).nes
- Donkey Kong 3 (World).nes
- Duck Hunt (World).nes
- Ice Climber (USA, Europe).nes
- Pac-Man (USA) (Namco).nes
- Super Mario Bros. (World).nes

# Démonstration

- Contrôles au clavier :
  - ZQSD/WASD/flèches : se déplacer
  - K et L pour A et B
  - Entrée pour start
  - Retour arrière pour select
  - P pour mettre en pause l’émulation
  - R pour reset le jeu

# Scénarii d’utilisation

- On peut jouer au clavier à un jeu NES utilisant le mapper 0 (il y en a environ 130)
- On ne peut pas jouer à la plupart des jeux NES
- On ne peut pas jouer avec une manette

# Vue d’ensemble du code

- cartridge.ml : parseur de cartouches
- bus.ml : tableau d’entiers qui contient RAM, PRG, $2000-$2007, $4016 + état de la puce graphique + miroir
- cpu.ml : exécute des instructions, possède des registres, résout selon le mode d’adressage
- ppumem.ml : tableau d’entiers qui contient CHR, nametables, pattern tables, palettes + miroir

# Vue d’ensemble du code

- ppu.ml : fonctions qui interprètent le contenu de la mémoire graphique pour afficher l’écran : draw_next_pixel(), render_sprites()
- init.ml : charger le jeu, réinitialiser le CPU, etc
- main.ml : rendu graphique dans une fenêtre, interprétation des touches de clavier, boucle principale du programme : NMI, VBlank

# Justification

- Le programme a été codé en style impératif et non fonctionnel. Plusieurs raisons : difficile, pas assez de temps, code peut-être plus optimal en impératif

# Déroulement

- 1 – CPU et parseur de cartouches codés en même temps car les tests sont importants pour débugger
- 2 – main.ml basique avec affichage graphique
- 3 – la puce graphique a été codée en dernier
-
- Le bus a été codé tout au long du projet

# Compétences requises

- Énormément de manipulations de bits et d’octets
- Il faut lire beaucoup de documentation et savoir la trouver
- Il faut savoir comparer le log de son CPU à un log de référence (maîtrise de bash, sed, grep, etc.)

# Difficultés rencontrées

- Obtenir un CPU qui se comporte correctement est difficile
- Le rendu graphique est complexe
- Optimisation : malgré des efforts, mon code est 5 à 10 fois plus lent que les émulateurs connus, les jeux arrivent à peine à tourner à vitesse normale

# Présentation d’un algorithme

- Algorithme qui dessine les sprites (= personnages qui bougent). Il doit être lancé 60 fois par seconde.
- Mémoire de 256 octets appelée OAM : chaque sprite fait 4 octets donc 64 sprites.
- Octet 1 = position y, octet 2 = numéro de tuile, octet 3 = attributs, octet 4 = position x
- La NES fournit un signal vidéo de 256x240 pixels

# Présentation d’un algorithme

- Chaque tuile est un tableau de 8x8 pixels où chaque case prend une valeur entre 0 et 3 = couleur
- Chaque tuile est représentée par 16 octets :
- octet 1 : 01011101  octet 2 : 11101001 … octet 8 : 11111111
- octet 9 : 10110111  octet 10 : 10111100 … octet 16 : 01011101
- 2 1 x x x x x x
- 3 x x x x x x x
- …
- x x x x x x x x

# Présentation d’un algorithme

- Entrées de l’algorithme :
- OAM : mémoire des sprites de 256 octets
- avant_plan : tableau 2D d’entiers de 240 lignes et 256 colonnes
- tuiles : tableau d’entiers de 4096 octets qui contient 256 tuiles
- palettes : tableau d’entiers de 16 octets, contient 4 palettes de couleurs à donner aux pixels, une couleur est un octet qui vaut entre 0 et 63

- Sortie de l’algorithme : avant_plan doit contenir la couleur de chaque pixel visible ou la constante PIXEL_TRANSPARENT

```
procédure rendu_sprites():

pour i allant de 0 à 239:
    pour j allant de 0 à 255:
        avant_plan[i][j] = PIXEL_TRANSPARENT

pour k allant de 63 à 0:
    position_y = OAM[k * 4] + 1
    if position_y = 256: position_y = 255

    position_x = OAM[k * 4 + 3]

    numero_tuile = OAM[k * 4 + 1]

    attributs = OAM[k * 4 + 2]
    b7, b6, …, b0 = les bits de attributs
    numero_palette = b1 b0 // Entier entre 0 et 3

    debut_tuile = numero_tuile * 16
```

Suite de « pour k allant de 63 à 0: » :

```
pour i allant de 0 à 7:
    pour j allant de 0 à 7:
        bit1 = bit(7 - j, tuiles[debut_tuile + i])
        bit2 = bit(7 - j, tuiles[debut_tuile + 8 + i])

        si ¬bit1 et ¬bit2:
            continue la boucle d'indice j
        sinon si bit1 et ¬bit2:
            couleur = palettes[numero_palette * 4 + 1]
        sinon si ¬bit1 et bit2:
            couleur = palettes[numero_palette * 4 + 2]
        sinon:
            couleur = palettes[numero_palette * 4 + 3]

        essayer: avant_plan[position_y + i]
                                    [position_x + j] = couleur
        attraper exception "Indice invalide":
            ne rien faire
```

# Testabilité

- Tests automatisés du CPU avec des cartouches spéciales de test avec pipeline GitLab
- nestest.nes : exécute environ 5000 instructions et compare le log
- Suite instr_test_v5 : écrit un code de retour et un message en mémoire
- Lancer manuellement Donkey Kong et Super Mario Bros., souvent avec log CPU, pour débugger
- Les jeux fonctionnent, ce qui prouve le succès du projet
- Cependant, la vitesse est à peine x1 alors que les émulateurs connus peuvent aller de x5 à x10

- Le profiling montre que la majorité du temps est passée à dessiner l’arrière-plan et résoudre des adresses de la mémoire graphique

| Usage CPU | Fonction | Description |
|---|---|---|
| 15,27% | Ppu.write_CHR_tile_colors | Écrit une tuile 8x8 par exemple dans l’avant-plan |
| 8,87% | Ppumem.read | Lit un octet de la mémoire graphique + miroir |
| 6,63% | Ppu.draw_next_pixel | Dessine un pixel + appelle rendu fg et bg |
| 4,48% | Ppumem.resolve_mirror | Résout une adresse de la mémoire graphique |
| 2,53% | Ppu.get_bg_pixel | Choisit le bon pixel entre deux arrière-plans |
| 2,48% | Utils.nth_bit | Donne le bit n d’un entier sous forme de booléen |
| 2,00% | Ppu.get_base_nametable… | Indique quel arrière-plan utiliser |
| 1,77% | Bus.read_raw | Lit un octet du bus, sans déclencher d’action |
| 1,70% | Ppu.write_to_bigarray | Écrit un pixel RGB sur l’écran |
| 1,27% | Ppu.render_sprites | Algorithme vu précédemment |
| 1,25% | Cpu.run_next_instruction | Lance une instruction CPU |
| 0,49% | Bus.read | Lit un octet du bus avec actions spéciales |
| 0,42% | Bus.resolve_mirror | Résout une adresse du bus |

# Conclusion

- J’ai appris beaucoup sur la NES, l’architecture d’un CPU, Dune, OCaml, pipeline CI GitLab, profiler du code
- Version 2.0 : optimisations selon profiling, rendu graphique parallélisé, redessiner uniquement ce qui a changé, limiteur de vitesse, son, mappers, interface graphique, savestates, manettes
- Si c’était à refaire : coder en C, CPU cycle-accurate, puce graphique fidèle au hardware (registres, cycle-accurate), représentation mémoire différente (pour bank switching)