

Juegos Retro

Cómo programar un juego
de NES y no morir en el
intento



Protagonistas



Edgardo Gho



Carlos Maidana

Disclaimer: Toda imagen usada es propiedad de sus respectivos dueños. Se usan con fines didácticos.



Temario

01

Quienes somos

Miembros del grupo de investigación en lógica programable

02

Nintendo (NES)

Arquitectura, PPU (video), APU (audio) y controles

03

Herramientas

Cosas que nos simplifican la vida

04

Ejemplos básicos

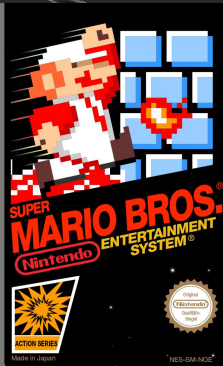
¿Se acuerdan de ASM?
Volvió! en forma de juego de Nintendo



Nintendo x Entertainment System

Creada por Nintendo en los 80s

- Procesador Ricoh RP2A03 (6502)
- PPU Ricoh 2C02
 - Resolución 256x240
 - 2KB Video RAM
 - Render, Scroll , Sprites
- APU
 - 5 Canales (2 SQ, Tri, Noise, PCM)
- 2KB Ram Integrada
- Soporta bancos de memoria
 - Permite direccionar una cantidad de memoria mayor de los 64KB originales del 6502.
- Chip de protección contra piratería



Cartucho



- El cartucho contiene el “programa” (juego). Como mínimo está compuesto por 3 chips
 - Chip de seguridad: Solo es fabricado por Nintendo, y si no está presente la consola rechaza el juego y se reinicia permanentemente.
 - Chip de programa (PGR): Contiene el binario del juego.
 - Chip de caracteres (CHR): Contiene las imágenes al estilo ROM de caracteres.
- Existen otros modelos de PCB con chips NVRAM y batería para almacenar el progreso del juego. Otros soporta un chip MMC (Memory Controller) que permite extender el direccionamiento por fuera de los 64KB.



Arquitectura de programación

Registros

Registro acumulador (8 bits)

Registros índice X,Y (8 bits)

Stack Pointer (8 bits) \$100~\$1ff

CCR: N,Z,V,C

Memoria Direccionable

2KB RAM (\$000~\$7FF)

Registros PPU (\$2000 ~ \$2007)

Registros APU (\$4000~\$4017)

ROM ~49K (\$4020~\$FFFF)

RAM Video 2KB (\$2000~\$2FFF*)

Direccionamiento (modos)

Inmediato, absoluto directo,
inherente, Base+offset (X,Y),
página cero, Indirecto mediante
página cero y offset (X,Y)

Instrucciones

66 Instrucciones

El modo BCD NO está incluido
en el Ricoh RP2A03

Una imagen completa



La resolución es de 256x240 "píxeles". Soporta 64 colores (6 bits).

$256 \times 240 = 61440$ píxeles

$61440 \times 6 \text{ bits} = 368640 \text{ bits por pantalla} = 46080 = \sim \text{46KB} !!!!!$

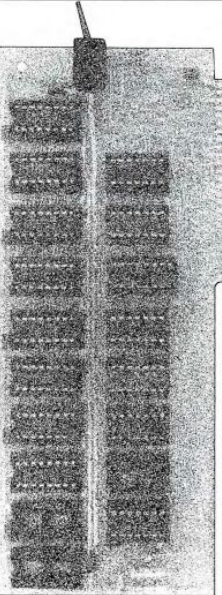
Una imagen completa

La resolución es de 256x240 "píxeles". Soporta 64 colores (6 bits).

$256 \times 240 = 61440$ píxeles

$61440 \times 6 \text{ bits} = 368640 \text{ bits por pantalla} = 46080 = \sim \text{46KB}!!!!$

16KB de RAM en los 80s costaban \$195. El NES en 1986 costaba \$200, por ende no podía tener 46K de RAM.



**16K Ram
Expansion
Board for the
Apple II*
\$195.00**

- expands your 48K Apple to 64K of programmable memory
- works with Microsoft Z-80 card, Visicalc, LISA ver 2.0 and other software
- eliminates the need for an Applesoft* or Integer Basic ROM Card
- switch selection of RAM or mother board ROM language
- includes installation and use manual
- fully assembled and tested

 Visa and MasterCard accepted

Shipping and handling will be added unless the order is accompanied by a check or money order
N.C. residents add 4% sales tax

*Apple II and Applesoft are trademarks of Apple Computer, Inc.

ANDROMEDA
★ INCORPORATED**
P.O. Box 19144
Greensboro, NC 27410
(919) 852-1482

**Formerly Andromeda Computer Systems

Una imagen completa



La resolución es de 256x240 "píxeles". Soporta 64 colores (6 bits).

$256 \times 240 = 61440$ píxeles

$61440 \times 6 \text{ bits} = 368640 \text{ bits por pantalla} = 46080 = \sim \text{46KB} !!!!!$

¡El NES solo tiene 2KB de memoria de vídeo, por ende no podemos almacenar ni una sola pantalla completa!



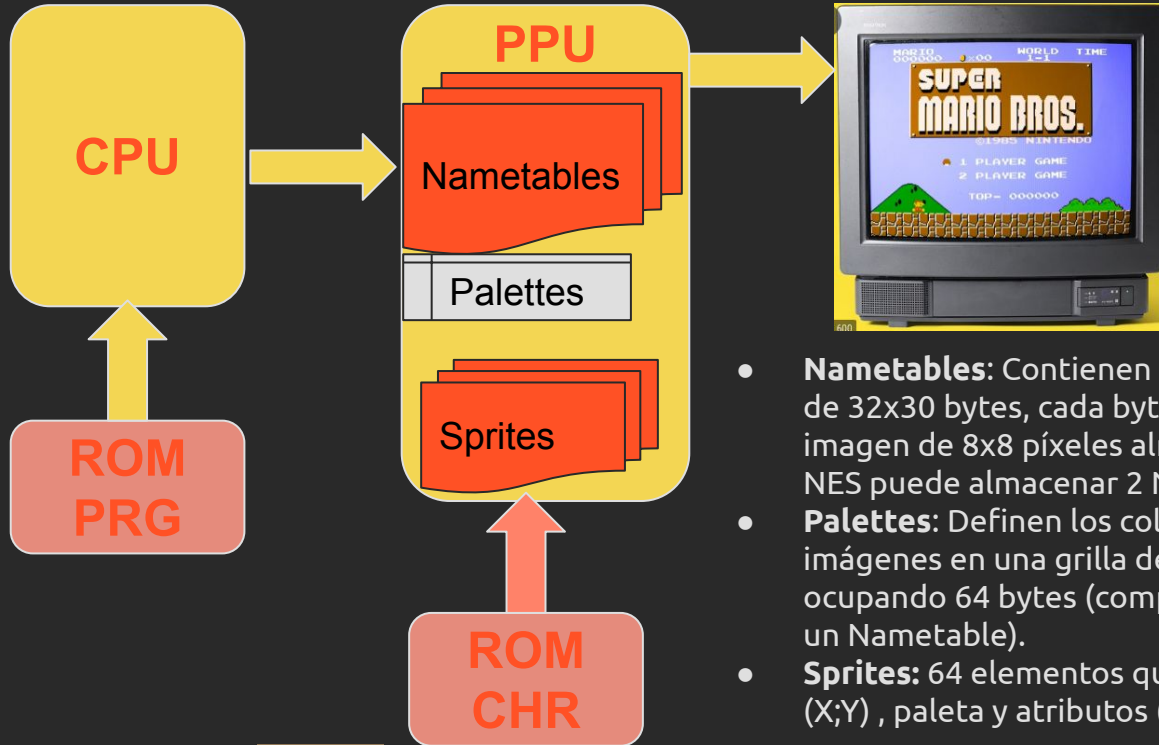
Pero muchos elementos se repiten, inclusive algunos cambian de color para representar elementos distintos (nubes vs arbustos). Algunos son inclusive simétricos.

¿Necesitamos realmente memoria RAM para almacenar imágenes que siempre son las mismas? NO. Podemos almacenar en ROM esas imágenes y luego la PPU lee los píxeles directamente de la ROM.

Picture Processing Unit (PPU)

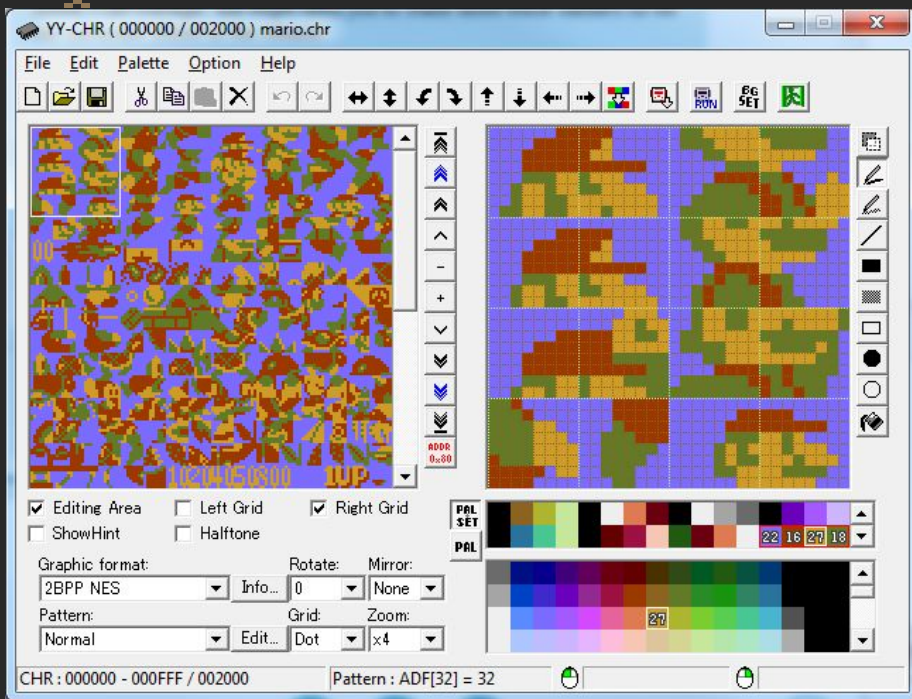
Lo que diferencia a una consola de una PC tradicional

Diagrama en bloques



- **Nametables:** Contienen el “fondo”. Se componen de 32x30 bytes, cada byte es una referencia a una imagen de 8x8 píxeles almacenados en CHR. El NES puede almacenar 2 Nametables (960 bytes).
- **Palettes:** Definen los colores a usarse por las imágenes en una grilla de 8x7.5 (32x32 píxeles) ocupando 64 bytes (completando 1KB luego de un Nametable).
- **Sprites:** 64 elementos que almacenan dirección (X;Y) , paleta y atributos (espejado, rotado, etc).

Como esta formada una CHR ROM



Contiene hasta 256 imágenes de 8x8 píxeles. Estas imágenes pueden dibujarse en el fondo (Nametable) o en un Sprite. Si se usara un bit por píxel, entonces 8x8 se almacena con 8 bytes.

Sprite



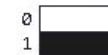
Color Indices

0	0	0	1	1	0	0	0
0	0	1	1	1	1	0	0
0	1	1	1	1	1	1	0
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1
0	0	1	0	0	1	0	0
0	1	0	1	1	0	1	0
1	0	0	0	0	0	1	1

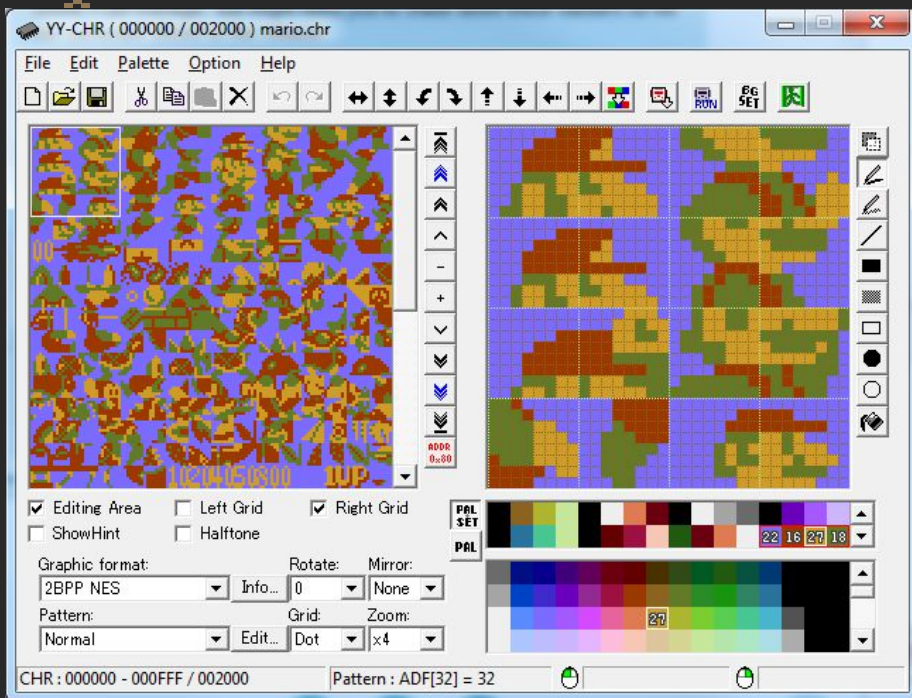
Mem.

\$18
\$3c
\$7e
\$db
\$ff
\$24
\$5a
\$81


Palette



Como esta formada una CHR ROM







El Nintendo usa 2BPP , y se almacenan en dos tablas separadas (MSB y LSB). Luego el color se resuelve mirando la paleta de colores definida para esa imagen.

Sprite


Color Indices

0	1	1	1	1	1	0
1	1	1	1	1	1	1
1	2	2	2	2	2	1
2	2	2	2	2	2	2
2	3	1	3	3	1	3
2	3	2	3	3	2	3
3	3	3	3	3	3	3
0	3	3	2	2	3	0

Palette

0	
1	
2	
3	

bitplane 1 (MSB)	bitplane 0 (LSB)
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	



CHR ROM (8KB)

La CHR ROM es de 8KB. Cada “tile” o patrón se define como 8x8 píxeles, codificados en 2BPP, por ende son 16 bytes por tile. $8192 \text{ bytes} / 16 \text{ bytes} = 512$ tiles posibles. Se divide entonces la CHR ROM en dos partes.. La parte A (0) y la parte B(1). Cada una de las partes almacena 256 tiles (se direccionan con un byte). Piensen en la CHR ROM como dos vectores de 256 elementos (cada elemento es un tile). Generalmente se utiliza una parte para almacenar las partes del “fondo” y la otra para almacenar las partes de los sprites.

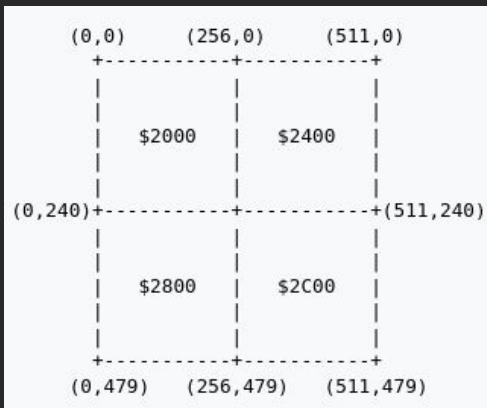


Nametable

32 x 30 referencias a imágenes almacenadas en CHR ROM utilizando así 960 bytes.

Cada 4x4 imágenes se define una paleta de colores que se almacena luego de las referencias (64 bytes).

Cada Nametable consume 1024 bytes. El espacio de memoria VRAM contiene 2KB, por ende puede almacenar como máximo 2 Nametables. Estos pueden utilizarse en horizontal o en vertical. Se definen en total 4 espacios para Nametable aunque solo existe memoria para 2.



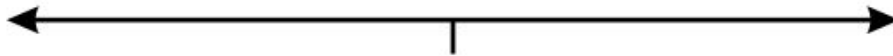
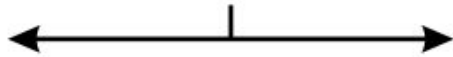
Nametable

Podemos armar un mapa mayor a 512 píxeles de ancho. En VRAM solo podemos almacenar 512, pero a medida que el personaje se desplaza, en el caso de Mario a la derecha, podemos ir cambiando el fondo y reemplazando por otros valores. Solo 256 píxeles se ven a la vez. Para lograr dibujar otras secciones se utilizan dos registros para hacer scrolling.



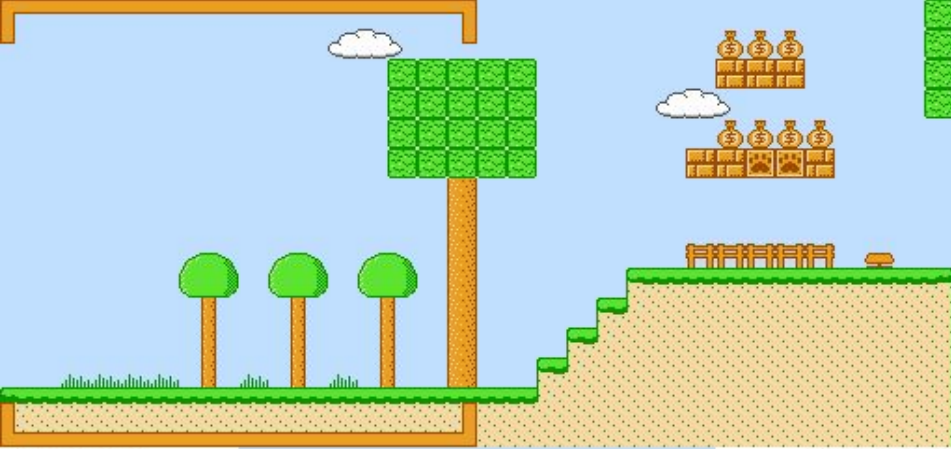
Nametable

The player sees this

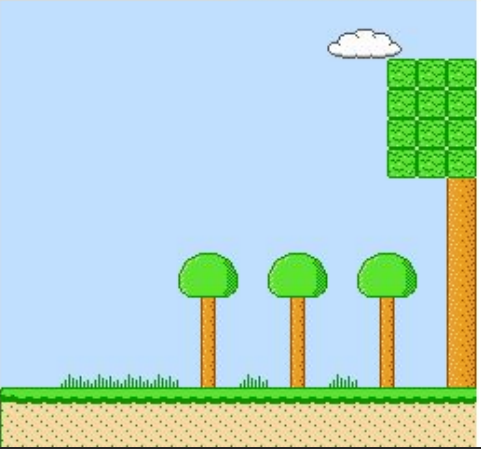


But the NES sees this

Este efecto es similar a una “cámara” que hace foco en cierta parte de la memoria. Parte del procesamiento que debe hacerse durante el juego es ir “desplazando” la cámara para que pueda seguir al personaje.



\$2000



\$2400

Output:

La "cámara" se puede configurar para que comience en cualquier pixel (256x240). Si sale del alcance de un nametable se mueve al siguiente (Horizontal o Vertical según la configuración). Se hace mirroring en los nametables restantes.



Sprites

Combinando imágenes de 8x8 podemos lograr representar elementos de mayor tamaño. Cada Sprite se define mediante 4 bytes en OAM (Object Attribute Memory).

1. Posición Y
2. Índice CHR ROM (referencia a la imagen)
3. Atributos
 - a. Paleta de colores (0,1,2 o 3)
 - b. Foreground o Background
 - c. Flip horizontal
 - d. Flip vertical
4. Posición X

En el caso de no querer usar el sprite, se dibuja “fuera” de la pantalla (pasando Y=240).

Existe memoria para almacenar 64 Sprites (256KB).

Existen limitaciones en cuantos Sprites pueden compartir una misma línea.

Se suele utilizar DMA para copiar los Sprites de RAM a VRAM durante el barrido vertical.



Paletas

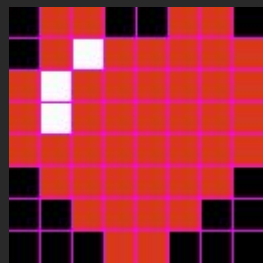
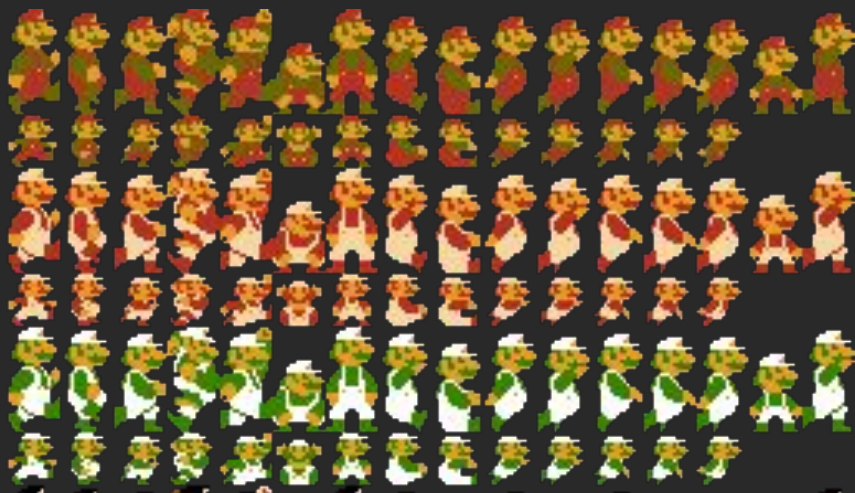
El NES soporta 64 colores, aunque algunos son todo negro, por ende efectivamente son 55.

Los juegos hacen uso de estos para reutilizar gráficos simplemente cambiando la paleta.

Los colores se pueden ver distintos si se utiliza un NES PAL o NTSC.

savtool's NES palette

00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f

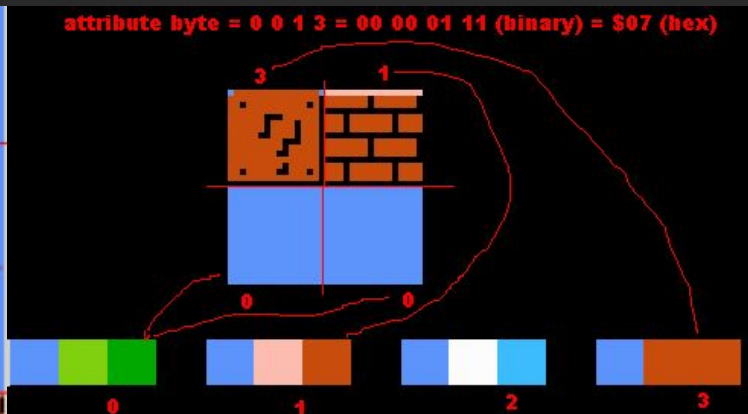
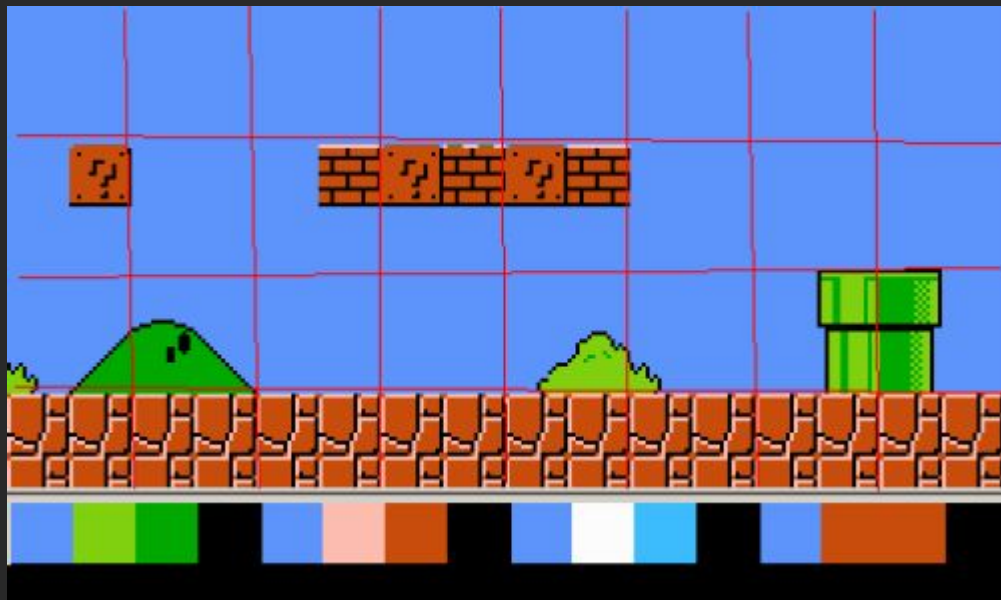


0	3	3	0	0	3	3	0
0	3	1	3	3	3	3	3
3	1	3	3	3	3	3	3
3	1	3	3	3	3	3	3
3	3	3	3	3	3	3	3
0	3	3	3	3	3	3	0
0	0	3	3	3	3	0	0
0	0	0	3	3	0	0	0

Address	Purpose
\$3F00	Universal background color
\$3F01-\$3F03	Background palette 0
\$3F05-\$3F07	Background palette 1
\$3F09-\$3F0B	Background palette 2
\$3F0D-\$3F0F	Background palette 3
\$3F11-\$3F13	Sprite palette 0
\$3F15-\$3F17	Sprite palette 1
\$3F19-\$3F1B	Sprite palette 2
\$3F1D-\$3F1F	Sprite palette 3



Paletas



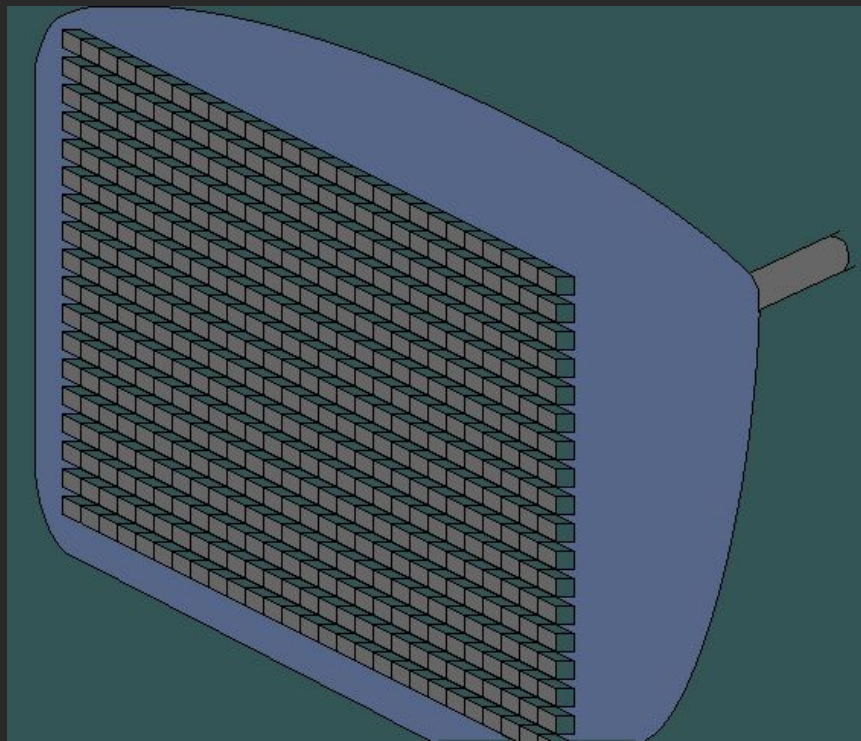


Como se dibuja la pantalla (NTSC)

Barrido horizontal (256 píxeles más un pequeño tiempo extra)

Barrido vertical (240 píxeles más un tiempo extra). Efectivamente se ven solo 224 líneas.

En total el tiempo de VBLANK es 2387 ciclos de CPU. Cuando se dibuja la línea 240 se produce una NMI para indicar que es seguro actualizar la PPU.



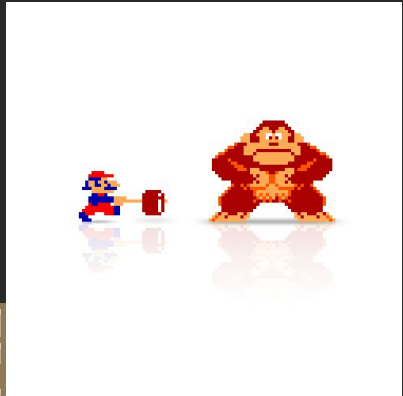
Como se dibujan los fondos

Si tenemos que dibujar toda la pantalla (1024 bytes), leer el valor al acumulador, luego copiarlo a VRAM toma aprox 8 ciclos, así que 8192 ciclos de CPU para toda la pantalla. **¡¡¡Pero solo hay 2387!!!!.** Es por esto que solo se actualiza “parte” de la pantalla. En el cambio de pantalla completa (por ejemplo cuando Mario pasa por un tubo), se puede apagar el render durante un ciclo completo.



Como se dibujan los sprites

El NES tiene 64 sprites (cada sprite se representa con 4 bytes). Los sprites probablemente cambian todo el tiempo. Se suele definir un área de RAM (0x200~0x2FF) para guardar el estado de los Sprites. Luego en cada ciclo NMI se copian los 256 bytes a VRAM (OAM). Dado que perder 2048 ciclos de CPU sería muy costoso, el Nintendo posee un controlador DMA. Este puede copiar desde 0x200 hasta 0x2FF y depositar el valor en OAM. Para esto detiene a la CPU y consume aprox 512 ciclos de CPU (o sea un 25% del tiempo).



Audio



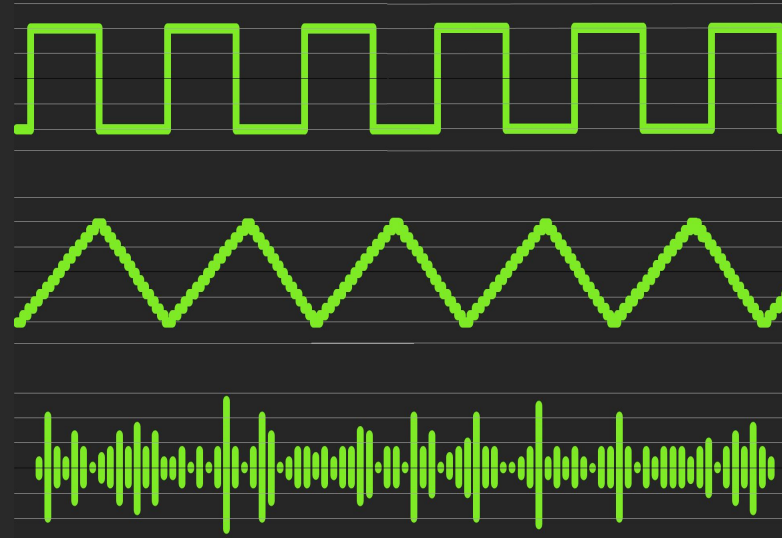
5 canales

Posee 5 canales:

- 2 de tipo Square Wave
- 1 tipo Triangle
- 1 tipo Noise
- 1 tipo Delta PCM

La frecuencia base se define como una división de la frecuencia de CPU. Se controla:

- Frecuencia / Período
- Volumen
- Envolvente
- Vibrato



Herramientas



Herramientas modernas

cc65

Ensamblador y linker

yychr

Crear CHRs
completos

FCEUX (win)

Emulador / Debugger

Nes Screen Tool

Dibujar pantallas y
exportarlas

FamiTracker

Tracks de audio



CC65

Es un compilador de C para 6502. Es multiplataforma. Su

■ código es libre bajo licencia Zlib. Utilizamos dos
■ programas:

- ca65 = Ensamblador
- ld65 = Linker

Para ensamblar un programa simplemente:

`ca65 archivo.s -o archivo.o -l archivo.lst`

Para fabricar el archivo .nes compatible con el emulador:

■ **`ld65 -C link.x archivo.o -o archivo.nes`**

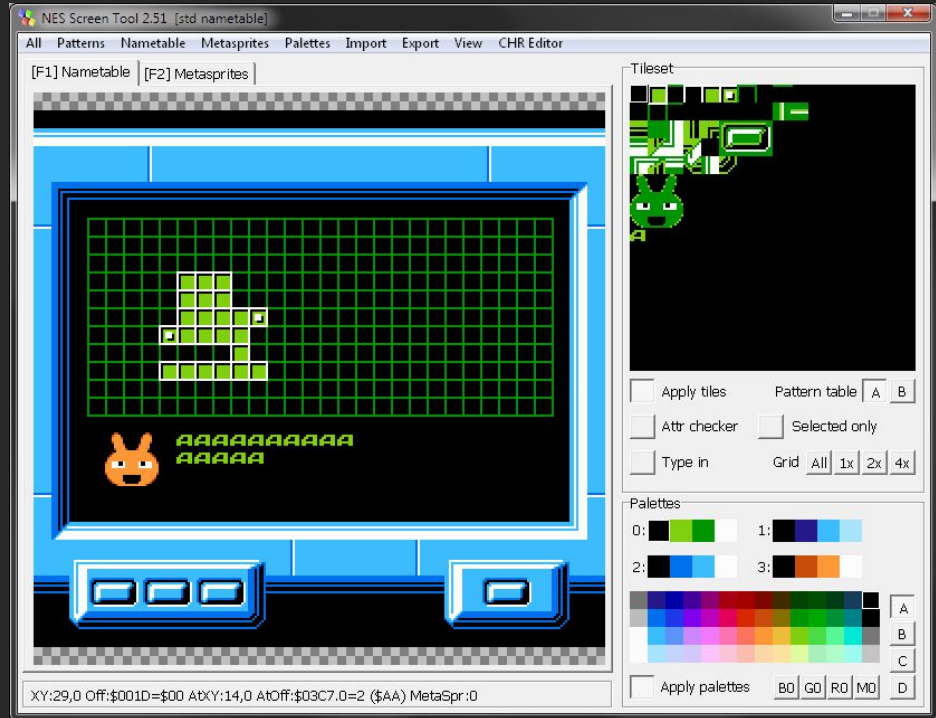
El archivo de linker (link.x) define los
segmentos de memoria y su tipo.

```
MEMORY {  
  ZP: start = $00, size = $0100, type = rw, file = "";  
  HDR: start = $0000, size = $0010, type = ro, file = %O, fill = yes, fillval = $00;  
  PRG: start = $8000, size = $8000, type = ro, file = %O, fill = yes, fillval = $00;  
  CHR: start = $0000, size = $2000, type = ro, file = %O, fill = yes, fillval = $00;  
}
```

```
SEGMENTS {  
  ZEROPAGE: load = ZP, type = zp;  
  HEADER: load = HDR, type = ro;  
  CODE: load = PRG, type = ro, start = $8000;  
  VECTORS: load = PRG, type = ro, start = $FFFA;  
  IMG: load = CHR, type = ro;  
}
```

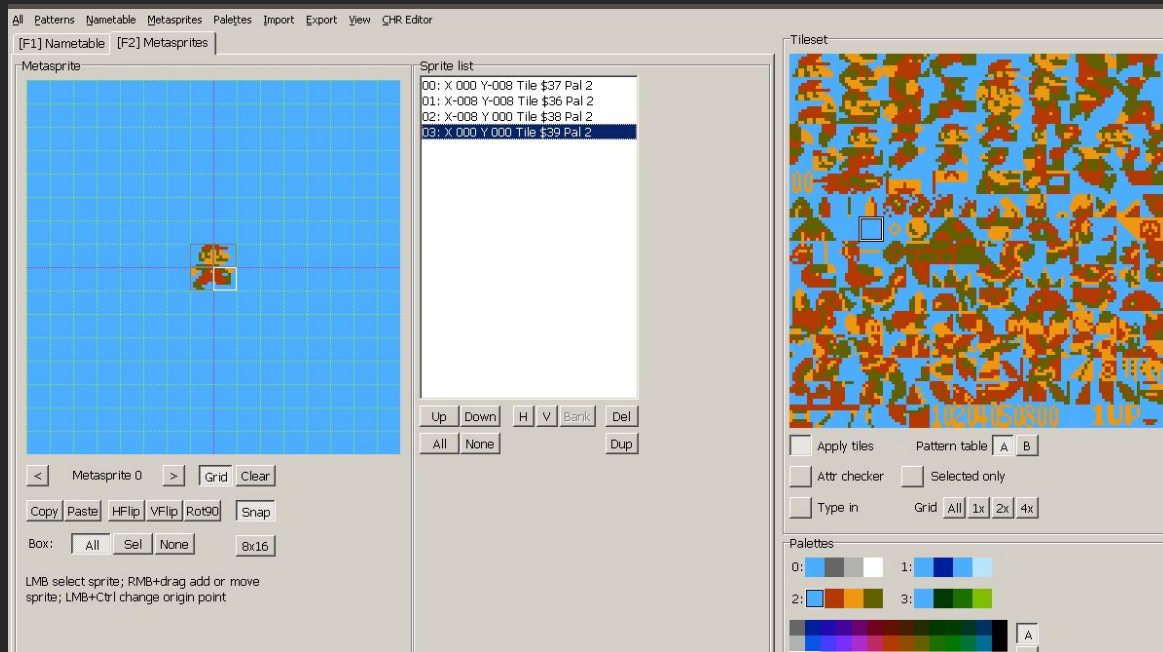
NES Screen Tool

Luego del ensamblador, es la herramienta más utilizada. Permite abrir archivos CHR y diagramar un Mapa (Nametable) definiendo las paletas de colores. También permite armar combinaciones de Sprites obtenidos del CHR. La herramienta de edición de tiles es muy básica. Permite exportar el Nametable.



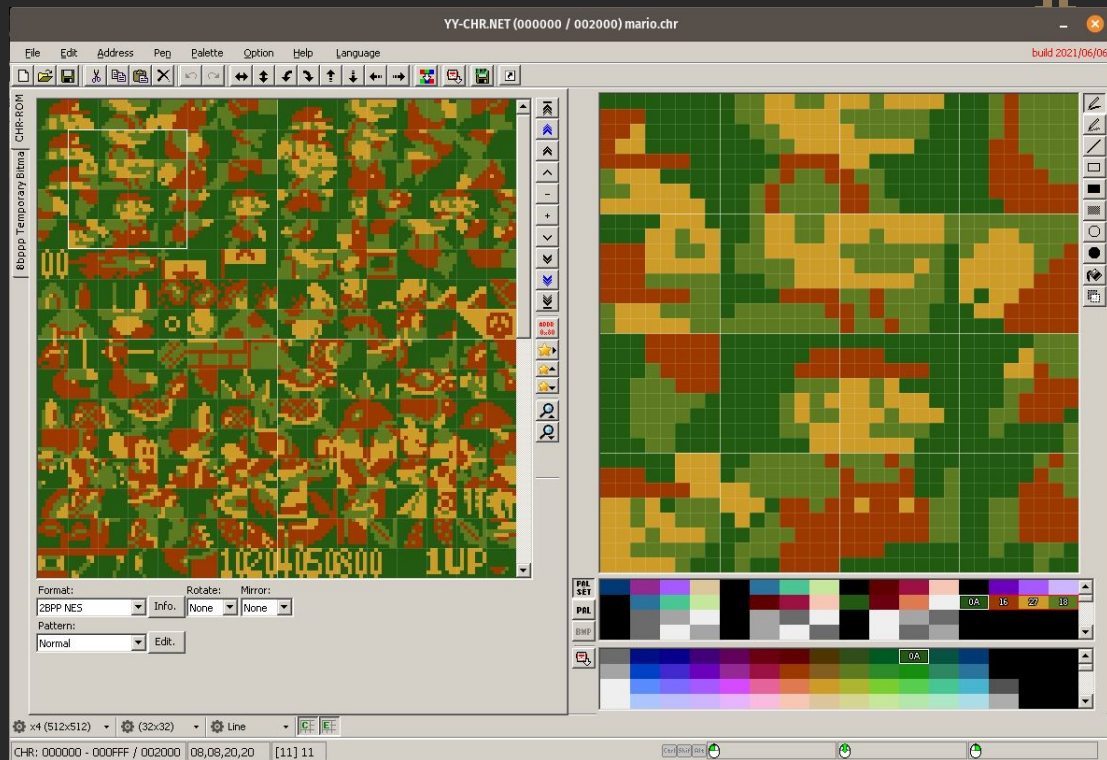
NES Screen Tool

La lista de Sprites que conforma un personaje puede fácilmente traducirse a código.



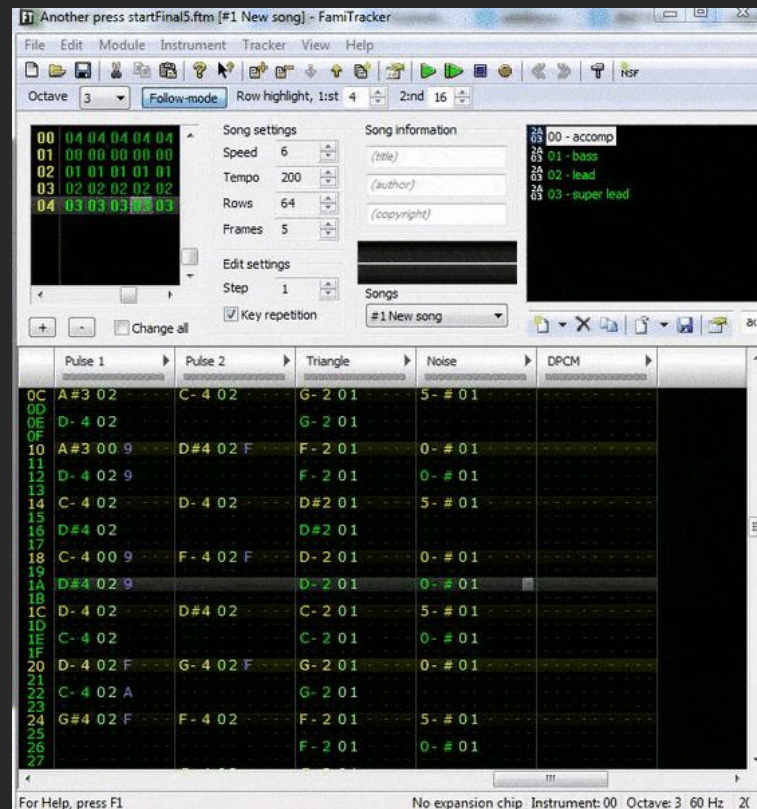
YYchar

Es un editor de tiles o sprites. Soporta varios formatos (NES, SNES, etc). Permite tomar imágenes BMP y transformarlas a tiles o sprites ajustando la paleta de colores. Hay dos versiones, una legacy (0.99) escrita en C y una .NET que es la más moderna.

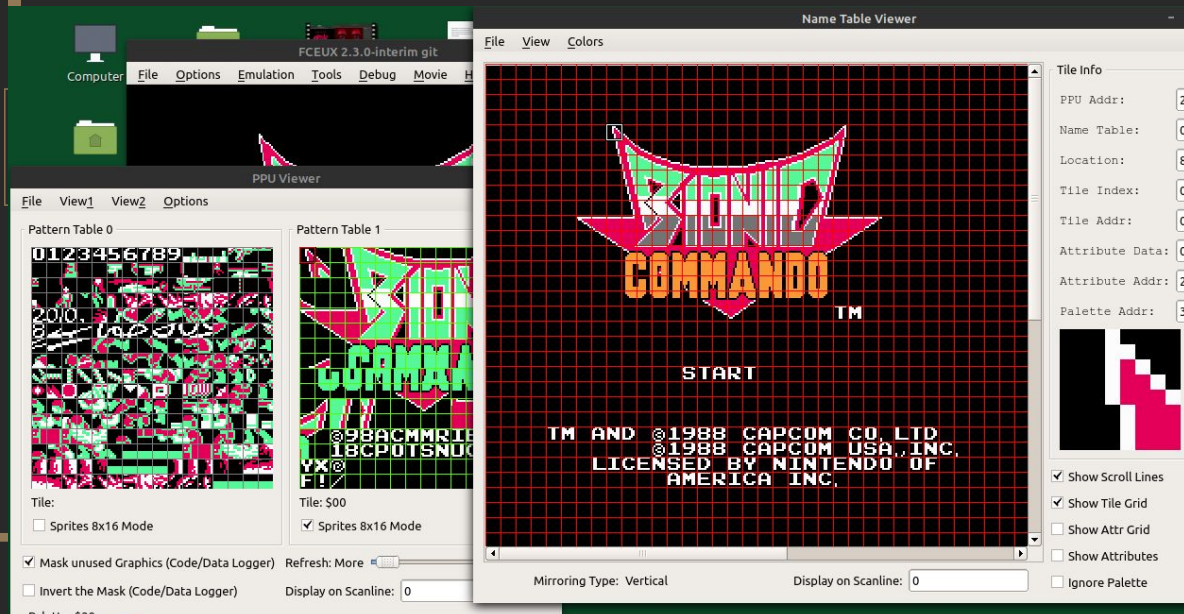


FamiTracker

- Permite crear tracks de música o efectos. Soporta definir instrumentos (volumen, envolvente, vibrato, etc) y utilizar esos instrumentos en distintas notas. Soporta muchos formatos para exportar el track, desde ASM directo (muy útil para efectos) hasta binarios completos (música de fondo) acompañados por el código reproductor en ASM/C.



FCEUX (win)



Si bien existen muchos emuladores, FCEUX en Windows posee previews de Nametables, PPU, y un debugger completo. Reproduce los juegos .nes generados por el linker.

Estructura de un Juego



No tenemos una ROM o Everdrive

Existen cartuchos llamados que soportan conectar una memoria SD que contenga binarios de ROM para que el NES ejecute.

Nosotros vamos a usar un emulador entonces tenemos que empaquetar todo en un archivo .nes que el emulador entienda.



Header

Todo archivo .nes debe

- contener un header que lo
- identifique ante el emulador.

Define la firma "NES"\$1A y luego indica el formato de PRG ROM, CHR ROM y si usa bancos (mapper). El header debe estar al principio del archivo .nes,

- esto se logra indicando al linker (ld65) donde colocar cada
- segmento mediante el archivo link.x

```
.segment "HEADER"

; Configurar con Mapper NROM 0 con bancos fijos
.byte 'N', 'E', 'S', $1A      ; Firma de NES para el emulador
.byte $02                     ; PRG tiene 16k
.byte $01                     ; CHR tiene 8k (archivo chr)
.byte %00000000               ; NROM Mapper 0
.byte $0, $0, $0, $0, $0, $0

; Fin del header
;; Este Header se encuentra definido en el archivo link.x el cual
;; define donde va a quedar en el archivo .NES final
```


CHR ROM

```
;Incluir el binario con las imagenes de la rom de caracteres  
.segment "IMG"  
.incbin "letras.chr"
```

El archivo con la ROM de caracteres (CHR) se define en un segmento (llamado IMG por ejemplo) y se incluye con **``.incbin archivo.chr``**

De esta forma el linker puede ubicar el contenido de forma que el emulador pueda encontrarlo.

- El tamaño máximo es de 8KB.
- En el caso de necesitar más tiles entonces hay que hacer bank switching.

RAM de datos (zeropage) y DMA

- Las variables se almacenan en RAM, y el NES posee 2KB. La página zero (\$000 ~ \$0ff) suele usarse para punteros o variables. Luego \$100~\$1ff contienen el stack. Después \$200~\$2ff suele ser utilizado para almacenar los 64 sprites (64x4). El DMA copia estos datos, desde \$0200 hasta \$02ff, deteniendo al procesador.

```
;Declaracion de variables en la pagina 0
;; Esto es RWM (RAM), por ende se "reservan" bytes
;; para luego ser usados como variables.
.segment "ZEROPAGE"
    posX: .res 1
    posY: .res 1
```

```
;Dado que en $0200~$02ff tenemos cargados los sprites
;Utilizamos el DMA para transferir estos 256 bytes a memoria
;de video en la ubicacion de los sprites.
LDA #$00
STA $2003 ; cargamos en el DMA la parte baja de 0200
LDA #$02
STA $4014 ; cargamos en el DMA la parte alta de 0200 y comienza.
;Esto deberia bloquear el procesador hasta que termina.
```


Vectores

Se cargan los vectores para indicar donde se encuentran las ISR para NMI, Reset e IRQ. Los dos importantes son NMI (que se produce cada vez que se completa una pantalla) y Reset, que va a quedar en un loop infinito ejecutando la lógica del juego.

```
; Direcciones para las ISR
.segment "VECTORS"
.word nmi
.word reset
.word irq
```

Reset

- Cuando se enciende el NES, se dispara la ISR de reset. La secuencia de encendido es fija, y consiste en esperar que la PPU se encienda y esté lista para su uso. Se inicializa el stack pointer, y se limpia la RAM mientras se espera a que la PPU esté lista. Luego de esto es un buen momento para cargar las paletas de colores y cargar el primer fondo del juego en VRAM.

```
;;Rutina de interrupcion (Reset)
;; Esta rutina se dispara cuando el nintendo se enciende
;; o se aprieta el boton de reset. Se encarga de inicializar
;; el hardware
reset:
    SEI                ; desactivar IRQs
    CLD                ; desactivar modo decimal

;;Durante el encendido del Nintendo hay que respetar unos tiempos
;;hasta que el PPU se encuentra listo para ser utilizado.
;;A continuacion se siguen los pasos sugeridos en:
;; https://wiki.nesdev.com/w/index.php/Init_code
    LDX #$40
    STX $4017          ; disable APU frame IRQ
    LDX #$FF
    TXS                ; Set up stack
    INX                ; now X = 0
    STX $2000          ; disable NMI
    STX $2001          ; disable rendering
    STX $4010          ; disable DMC IRQs

vblankwait1:          ; First wait for vblank to make sure PPU is ready
    BIT $2002
    BPL vblankwait1

clrmem:
    LDA #$00
    STA $0000, x
    STA $0100, x
    STA $0300, x
    STA $0400, x
    STA $0500, x
    STA $0600, x
    STA $0700, x
    LDA #$FE
    STA $0200, x
    INX
    BNE clrmem

vblankwait2:          ; Second wait for vblank, PPU is ready after this
    BIT $2002
    BPL vblankwait2
```

Reset

- Con el primer fondo ya cargado,
- encendemos la PPU (indicando en qué parte de la CHR ROM están los fondos y en cual los sprites). Podemos encender el scroll. Por último habilitamos las interrupciones (si es que vamos a utilizarlas) y nos quedamos en un loop infinito donde se va a desarrollar el juego propiamente dicho. En este loop NO debería actualizarse la pantalla salvo que estemos en VBLANK.

```
;; Encendemos el PPU
;; y el barrido vertical
;; y apuntamos el PPU a
;; la tabla 0 de sprites
;; y 1 para fondos

LDA #%10000000
STA $2000

;; Encendemos Sprites, Background y sin clipping en lado izquierdo
LDA #%00011110
STA $2001

;;Apagamos el scroll del background (fondo)
LDA #$00
STA $2005
STA $2005

;;Habilitamos las interrupciones
CLI

fin:
jmp fin
```

NMI

- La NMI se dispara cuando se produce el VBLANK. Debe guardar los registros que modifique, como el Acumulador y los flags y recuperarlos antes de RTI. Dentro de NMI podemos actualizar la VRAM sin que se produzca flickering en la pantalla.

```
;Rutina de interrupcion (NMI)
;Esta rutina se dispara cuando la pantalla
;se dibujo por completo, y el barrido vertical
;esta volviendo al inicio. Deberia poder utilizarse
;solo por 2250 ciclos aprox. Deberia dispararse 25 veces
;por segundo o 50 con interlaceado
nmi:
    ;;Guardamos en el stack el estado del CPU (flags y acumulador)
    PHA
    PHP

    ;;Ahora recuperamos el estado del CPU (flags y acumulador) y listo
    PLP
    PLA
    RTI
```



¡Hola Mundo!

Demos en
github.com/edgardogho/NESHolaMundo
github.com/edgardogho/NES6502

Referencias

La biblia del NES:

https://www.nesdev.org/wiki/Nesdev_Wiki

YY-CHR:

<https://w.atwiki.jp/vychr/>

NES Screen tool

<https://shiru.undergrund.net/software.shtml>

FamiTracker

<http://famitracker.com/>

cc65

<https://cc65.github.io/>