Notas sobre Programación de la Consola NES

Carlos Jacanamejoy

5 de mayo de 2018

Nota del autor: Esta es una publicación preliminar, deseaba publicar la versión cuando estuviera completa, pero debido a diferentes proyectos no se cuando continue con esta, por lo tanto decidí publicarlo así como está con la esperanza de que le pueda ser de utilidad para los curiosos del mundo retro.

1. Introducción

La consola NES, a pesar de utilizar un procesador de 8 bits permitió el desarrollo de centenares de horas de diversión para mucha gente. El interés de programar sobre la misma radica en la ilusión que tengo de crear un juego original, creado desde cero, pasando por la crearción de la parte gráfica, sonidos, hasta la programación de las diferentes mecánicas del juego. Dado que me agradan las historias, también, quisiera contar una desarrollada en un mundo mágico, donde pueda plasmar mi creatividad.

Todo el trabajo se desarrolla sobre la plataforma Linux (Ubuntu 14.04 LTS). Se utiliza el compilador cc65 del proyecto con su mismo nombre cc65. Agradecimientos a Doug [1], creador del lugar que me ha sido de gran ayuda, porque es donde están diferentes tutoriales para aprender a programar la NES.

2. Sistema de memoria de la NES

La NES tiene dos mapas de memoria separados, uno para la CPU, y otro para la PPU. A continuación los detalles sobre las diferentes memorias.

2.1. Mapa de direcciones para la CPU

Las direcciones con las que puede trabajar el procesador de la NES son de 16 bits (0-65535, 0x0000 - 0xFFFF). Las diferentes operaciones que realice

el procesador las realizará utilizando los valores de la dirección que se halla en el bus de datos. El mapa general de direcciones se puede consultar en la tabla 1.

Rango	Tamaño	Dispositivo asociado
0x0000-0x07FF	0x0800	2KB internal RAM
0x0800-0x0FFF	0x0800	Mirrors of 0x0000-0x07FF
0x1000-0x17FF	0x0800	
0x1800-0x1FFF	0x0800	
0x2000-0x2007	0x0008	NES PPU registers
0x2008-0x3FFF	0x1FF8	Mirrors of 0x2000-2007 (repeats every 8 bytes)
0x4000-0x4017	0x0018	NES APU and I/O registers
0x4018-0x401F	0x0008	APU and I/O functionality that is normally disabled. See CPU Test Mode.
0x4020-0xFFFF	0xBFE0	Cartridge space: PRG ROM, PRG RAM, and mapper registers (See Note)

Tabla 1: Mapa de direcciones para la CPU [2]

2.2. Mapa de direcciones para la PPU

La PPU es el chip encargado de dibujar. Posee direcciones de 16KB de espacio (0x0000-0x3FFF). En la tabla 2 se presenta el detalle del mapa de memoria. El bus de la PPU está completamente separado al de la CPU, aunque desde la última se puede acceder ajustando los registros de mapeo de memoria en 0x2006 y 0x2007 [3].

2.3. Acerca de los cartuchos para la NES

Algo importante a saber es que existen básicamente dos tipos de cartuchos para la NES. El primero tiene dos chips para la ROM, 1 para la PRG-ROM (código ejecutable), y uno para CHR-ROM (gráficos). Este tipo mapea automáticamente los gráficos a la memoria de la PPU en 0-0x1FFF. Para dibujar sobre la pantalla se debe escribir el número de tile sobre la tabla de nombres (nametable). El otro tipo de cartuchos tiene un chip PRG-ROM y en lugar de un chip CHR-ROM, tienen un chip CHR-RAM de 0x2000 bytes. Los gráficos se encuentran en algún lugar del PRG-ROM, y el programa tiene que cargarlos en el CHR-RAM (escribiendo las direcciones 0-1fff del PPU) antes de que se pueda acceder a ellos.

3. Herramientas disponibles

Para la compilación se utilizan los makefile de los ejemplos [1]. Para la edición de los tiles se desarrolló un programa de edición gráfica en python, y para pasarlo a formato de la CHR-ROM se utiliza un script de conversión de formato basados en [4]. Alternativamente hay una herramienta muy versatil para editar tiles YY-CHR [5], en Linux (Ubuntu 14.04 lts) usando Wine-1.6.2 funciona correctamente. También hay herramientas variadas creadas por Shiru [6] para el desarrollo de juegos para la NES.

Es interesante la herramienta FCEUX[7], que es muy útil para hacer debug de los juegos. Posee un visor de las Nametables (tablas de nombre de tiles) para verla mientras se juega. Tambien se puede ver los tiles cargados.

4. Modo de trabajo del cc65

El compilador cc65 requiere diferentes directivas, y en diferentes pasos, para compilar de manera simple, se utiliza un archivo makefile. El compilador cc65 genera el código objeto (asembler). Entonces ld65 enlaza los ficheros objeto utilizando el archivo de configuración (*.cfg), de esta forma se obtiene un solo archivo *.nes.

La NES tiene por procesador el chip 6502, que trabaja con instrucciones de 8 bits. No hay forma simple para trabajar con variables de más de 8 bits, por lo que se recomienda trabajar con variables sin signo de 8 bits (unsigned byte). Las direcciones son de 16 bits, pero casi todo el resto se procesa a 8 bits. Las únicas operaciones matemáticas posibles son la adición, sustracción y la multiplación/división utilizando desplazamiento de bits (factores de 2, cocientes de 2).

Para escribir código en C, se deben tener ciertas consideraciones:

1. Las variables deben ser de 8 bits (unsigned char), por lo cual se limitan a valores de 0 a 255.
2. Tratar de no pasar valores a las funciones, o utilizar un fastcall, que almacena las variables pasadas en los registros A, X, Y.
3. Las matrices, idealmente, deben tener un máximo de 256 bytes.
4. Muchas de las cosas a las que estás acostumbrado (printf) no funcionarán.
5. Utilizar ++g en lugar de g++ (es más rápido).
6. Cc65 no puede pasar estructuras por valor, ni devolver una estructura.
7. Las variables declaradas a nivel global se compilarán mucho más rápido que las locales. Incluso las estructuras declaradas a nivel global funcionarán mucho más rápido que las locales.

Al compilar se debe utilizar la directiva **-O** para optimizar el código. Existen otras directivas adicionales que pueden ayudar a optimizar el código, i,r,s, que se pueden combinar en algo como **-Oirs**. Se debe tener cuidado con los bugs generados por las optimizaciones. Por ejemplo, la lectura de datos de un registro de hardware, y los datos no son utilizados por el programa, se optimizará lejos el hardware leído . Mas sugerencias se pueden hallar en [8]

Se debe optimizar el código debido a que el programa puede volverse muy lento, así como tambien muy grande y no alcanzar en el reducido espacio que se dispone.

Los módulos asm (ca65) pueden compartir etiquetas/variables entre si con importación, exportación, importzp, exportzp. Cc65 puede acceder a variables y arrays desde módulos asm declarando una variable “extern unsigned char foo;” (y si es un símbolo zeropage, agregue la línea “#pragma zpsym (“foo”) ;”. Cuando compile el código C, añadir una definición de importación para él. El uso de “extern” es para cuando se tiene un gran archivo binario, pues es más fácil incluirlo en el código asm como el siguiente :

```
.export _foo
_foo:
.incbin "foo.bin"
```

A continuación, para acceder desde el código C, haga esto “extern unsigned char foo [];". Tenga en cuenta el subrayado. Por alguna razón, cuando cc65 compila, agrega un subrayado antes de cada símbolo. Por lo tanto, en el lado asm usted tiene que agregar un guión bajo a cada etiqueta/variable exportada.

Es posible llamar a funciones en cc65 escritas en asm con un __fastcall__. Esto almacenará las variables pasadas en los registros A, X, Y, en lugar de la pila C. Algunos bits de código no se pueden hacer en C, así que podríamos usar una biblioteca de instrucciones ASM especiales que pueden ser importadas y llamadas por el código C. (Como el código de inicio). Se debe tratar de no pasar valores, si es posible, porque en caso contrario se producirá código muy lento en asm.

También es posible escribir código asm en el código C. Se vería así:

```
  asm ("Z: bit $2002") ;
  asm ("bpl Z") ;
```

En los ejemplos tomados de [1], se cambió el código de inicio llamado crt0.s, por reset.s. Tambien se cambia el fichero *.cfg. Los dos archivos mencionados cambian en las diferentes lecciones. Se añade -add-source a la línea de comandos, por lo que recompilar generará un archivo .s (asm) con código C incluido, en caso de que lo desee ver en el código generado asm.

5. Manejo de la PPU

La PPU se encarga de gestionar los gráficos sobre la pantalla. La zona de dibujo o pantalla consiste en un arreglo de 32 tiles ancho, por 24 tiles de

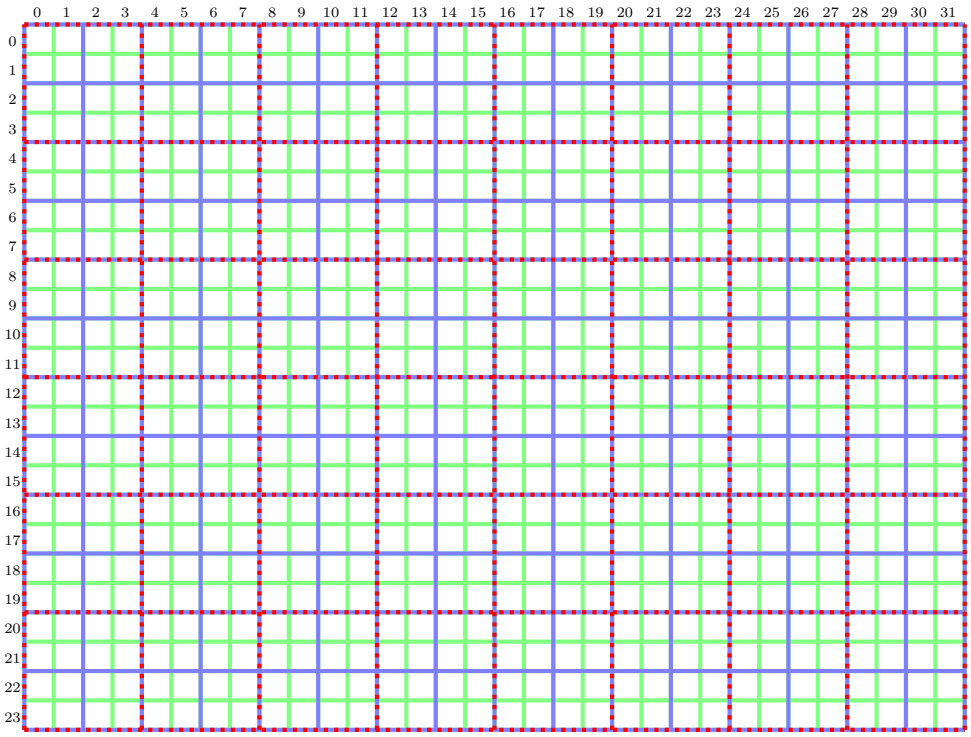

Figura 1: Distribución de la pantalla .

alto. Cada tile corresponde a un cuadrado de 8 pixeles de lado. En la figura 1 se aprecia la forma en que se divide la pantalla.

Address range	**Size**	**Description**
0x0000-0x0FFF	0x1000	Pattern table 0
0x1000-0x1FFF	0x1000	Pattern Table 1
0x2000-0x23FF	0x0400	Nametable 0
0x2400-0x27FF	0x0400	Nametable 1
0x2800-0x2BFF	0x0400	Nametable 2
0x2C00-0x2FFF	0x0400	Nametable 3
0x3000-0x3EFF	0x0F00	Mirrors of 0x2000-0x2EFF
0x3F00-0x3F1F	0x0020	Palette RAM indexes
0x3F20-0x3FFF	0x00E0	Mirrors of 0x3F00-0x3F1F

Tabla 2: Mapa de direcciones para la PPU [3]

En un juego que se desarrolla en una pantalla estática solo se requiere trabajar con un nametable, el cero, que corresponde a la memoria en PPU de 0x2000-0x23FF. Se usan 0x400 (1024) bytes por nametable, de los cuales 0x300 (32x24=768) se dedican para referenciar al tile a dibujar, y 0x100 para configurar las diferentes paletas de 4 colores en las diferentes zonas de la pantalla.

5.1. Modificar registros de la PPU

Para modificar registros en la PPU se debe indicar su dirección inicial en 0x2006. Una vez establecida la dirección de inicio, se envian los datos escribiendo en 0x2007. La dirección del registro en la PPU sobre el cual se escribirá se incrementa automáticamente en cada escritura.

5.2. Dibujo básico de tiles

Para dibujar tiles sobre la pantalla de la forma mas simple, se debe modificar un nametable. Los tiles están ubicados en el fihcero Alpha.chr, el cual es invocado en el fichero reset.s. Para dibujar sobre la pantalla con los tiles se hará la siguiente serie de pasos:

1. Apagar la pantalla.
2. Asignar una paleta de colores.
3. Modificar el nametable con el patrón a mostrar.
4. Resetear el scroll.
5. Encender la pantalla.
6. Iniciar un ciclo infinito

Se debe apagar la pantalla (paso 1) porque solo se puede modificar cuando la consola está en estado de V-Blank o pantalla apagada. En otros casos, los efectos causados no serán los esperados. Modificar la pantalla incluye cambiar la paleta, los azulejos de fondo o cambiar los sprites.

Para añadir la paleta de colores (paso 2) , se deben escribir en la PPU a partir de la dirección 0x3f00. Por defecto se usa en toda la pantalla la primera paleta de 4 colores, los cuales pueden ser escogidos entre todos los colores posibles (ver figura 2).

Para dibujar en la pantalla (paso 3) se debe modificar un nametable de la PPU. Cada nueva escritura en 0x2007, será un tile hacia adelante, eventualmente al finalizar la linea se irá a la siguiente.

Al escribir en la PPU, también se arruina la posición de desplazamiento del fondo, por lo que necesita ser restablecido (paso 4) .

Finalmente se enciende la pantalla y se inicia un ciclo infinito para que siempre se muestre el patrón de tiles definido en el nametable cero. En la lección uno de Doug[1], se puede ver el código realizando el dibujo básico de tiles mencionado.

5.3. V-blank: el momento para cambiar la PPU

El PPU es un procesador gráfico que envía una señal al televisor, u obtiene nueva información de la CPU (a través de un lento único byte en un bus de tiempo). No puede hacer las dos cosas al mismo tiempo. Por lo tanto, la única vez que se puede enviar nueva información es cuando está apagado el envío de datos al televisor (pantalla), o durante v-blank.

La PPU pasa el 90 % de su tiempo enviando píxeles al televisor, una línea a la vez. Inicia en la esquina superior izquierda y va hacia la derecha, baja una linea,y nuevamente de izquierda a derecha, baja una, de izquierda a derecha, etc, hasta que llega al fondo, luego se detiene, salta de nuevo hacia arriba a la izquierda y comienza de nuevo, repitiéndolo 60 veces por segundo. La ligera pausa en la parte inferior es el período en blanco vertical (v-blank). Cuando la pantalla está 'encendida', es la única vez que puede enviar información nueva sin dibujar lineals deformes (weired glitchy) extrañas a través de la pantalla.

El período V-blank es muy corto. Tan corto, que sólo da para escribir de 2 a 4 columnas de tiles de 8x8, y actualizar los sprites. Para un juego de desplazamiento, tendrá que actualizar los tiles de fondo a medida que vaya avanzando, pero, sólo durante V-blank. Por lo tanto, el v-blank es muy importante durante el tiempo que tarda.

Hay 2 maneras de saber cuando hay un v-blank. La PPU establece una bandera que se puede comprobar (el alto bit de 0x2002). O bien, puede activar las interrupciones NMI (non-maskable interrupt). Cuando NMI está activado, el programa detendrá lo que está haciendo cuando recibe una señal de que V-blank ha comenzado, y saltará al código NMI. Puede utilizar este tiempo para escribir en la PPU, y temporizar el código de música junto con los movimientos de sprites. Se sabe que saltará a NMI cada 1/60 de segundo, por lo que todo el juego usará eso como una referencia para determinar el tiempo que ha pasado entre los eventos.

En la sección What's a V-blank? de Doug, hay un excelente ejemplo de código sobre cómo usar v-blank. Para trabajar con v-blank, en la lección dos se cambia el módulo reset.s insertando las referencias a NMI_flag (++NMI_flag) y Frame_count (++Frame_count) en el controlador de NMI.

5.4. Tabla de atributos: Uso de colores

En la figura 2 se aprecian los 50 colores posibles que se pueden usar en la NES. Estos colores pueden variar dependiendo del tipo de salida de video. Según el emulador, también se puede ajustar la paleta de los 50 colores, algunas paletas se pueden hallar en [9]. La paleta de colores en la NES tiene asignadas 32 direcciones de memoria reservadas (0x3f00-0x3f1f). 16 para el fondo (0x3f00-0x3f0f), y 16 para sprites (0x3f10-0x3f1f). Cada paleta se divide en secciones de a 4 colores, y el primer color de cada una es ignorada y se toma el color de fondo (primer color de la primera paleta, 0x3f00). Por

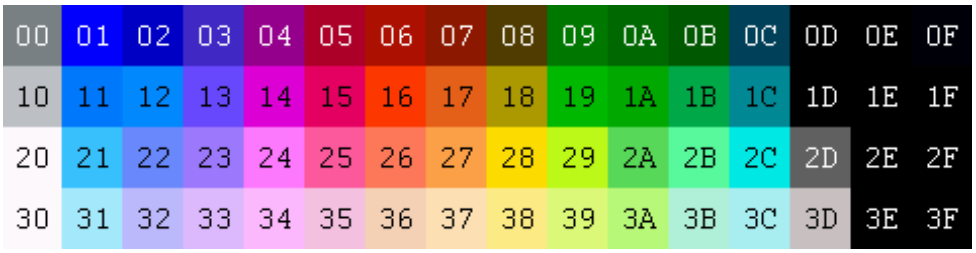

Figura 2: Tabla de colores posibles [10].

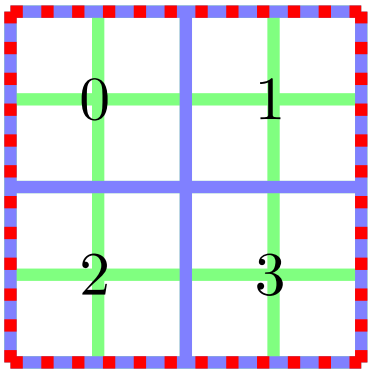

Figura 3: Tiles cubiertos por cada byte de la TAB-AT.

lo tanto un sprite puede utilizar solamente 3 colores. Y cada tile puede usar 3 colores mas el color de fondo.

En pantalla solo se pueden tener como máximo 25 colores de los 50 posibles (figura 2). Según Doug, es recomendable usar los colores 0F,1F,2F,3F para el negro, pues por ejemplo 0D suele presentar problemas con algunos televisores.

Solamente se puede tener una paleta de 4 colores en una zona de 16x16 pixeles. La herramienta Nes Screen Tool [6] es bastante util para crear fondos.

Cada pantalla de fondo (nametable) tiene 64 bytes dedicados a “atributos” ... es decir, qué paleta utilizar. El nametable 0 tiene su tabla de atributos (TAB-AT) en 23c0-23ff. En la figura 3 se ilustra como se divide cada zona representada por un byte de la TAB-AT, cubre una zona de 32x32 pixeles, es decir 4x4 tiles, que es dividida en 4 partes , cada parte (cuadrado azul) tiene asignada una paleta de 4 colores. Cada 2 bits indican la paleta seleccionada para la zona de 16x16 pixeles. El orden en que se toman los pares de bits para empaquetarlos en el byte es 33221100, es decir, los dos bits menos significativos para la zona 0, y asi en adelante (ver las zonas numeradas en la figura 3).

Por ejemplo si se requiere usar la paleta 1 en el área 3, el byte de atributo sería algo como 0b01xxxxxx.

En [10] se halla la lección 3 que presenta como realizar el ajuste de la TAB-AT. También se presenta como ubicar variables en la zona zeropage utilizando una directiva #pragma. Las variables de la zona zeropage son

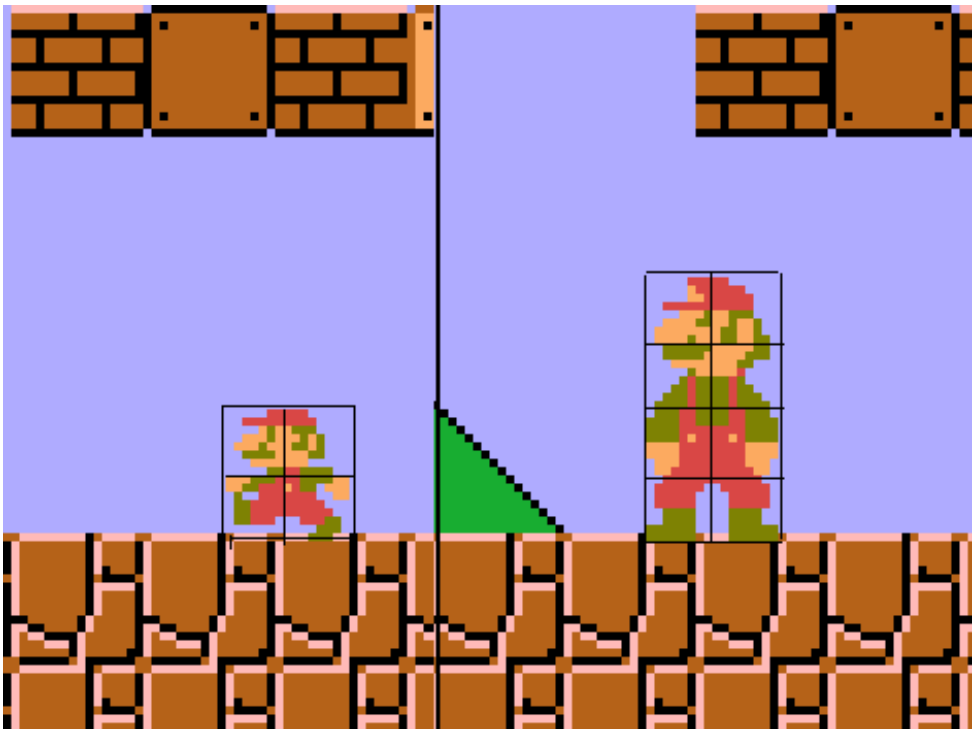

Figura 4: Metasprites en Super Mario.

importantes porque trabajan mas rápido.

Las zonas 32x32 inician a contarse de forma similar a los tiles, puede darse buena referencia observando en la figura 1.

5.5. Manejo de Sprites

Un sprite es un objeto de 8x8 pixeles que se puede mover libremente por toda la pantalla. Tambien se pueden usar sprites de 8x16. Casi todos los personajes son construidos a partir de sprites, salvo algunas excepciones en ciertos juegos.

Dado que 8x8 es un espacio reducido, se suelen conectar muchos sprites juntos en un Metasprite. Por ejemplo, en la figura 4, un bloque de 2×2 de sprites se usan para un Mario pequeño. Un bloque de 2×4 de sprites para un Mario grande.

La NES únicamente puede manejar 64 sprites, y sólo se puede poner 8 sprites en la misma línea de exploración (de izquierda a derecha). Además, los sprites de menor prioridad no aparecerán en esa línea de exploración. Son usuales juegos con sprites parpadeando, en realidad la NES no parpadea naturalmente, lo que ocurre es que es una manera de mantenerlos visibles en la pantalla cuando hay demasiados. El programador hace esto al mezclar las prioridades del sprite en cada fotograma, porque un sprite que parpadea es mejor que un sprite invisible.

Una prioridad de sprite refleja cual se dibuja primero. Un Sprite 0 utiliza

las direcciones OAM (Object Attribute Memory) 0-3. Sprite 1 utiliza las direcciones OAM 4-7. Sprite 2 utiliza direcciones OAM 8-B. Etc. El Sprite 0 siempre tendrá prioridad sobre los otros sprites. Sprite 1 tendrá la siguiente prioridad más alta, luego Sprite 2, etc. Esto significa que si ocupan el mismo espacio, los sprites de número inferior (es decir, de mayor prioridad) aparecerán en la parte superior. Si hay más de 8 sprites en una línea de exploración, aparecerán los 8 sprites de menor número (es decir, de mayor prioridad) [en esa línea de exploración] y los otros desaparecerán.

Para que los sprites aparezcan en la pantalla se establecen sus coordenadas para que estén en la zona visible. Cada sprite utiliza 4 bytes de memoria (algunos llaman a este grupo de bytes OAM). El OAM consiste en coordenada *y*, indice del Tile, atributos, coordenada *x* [11].

Para que aparezca un sprite, se asigna coordenadas, un tile y se escoge una paleta. Se podría hacer esto escribiendo la dirección de Sprite en 0x2003 y después escribiendo los datos de Sprite a 0x2004. Esto se hace raramente, pues hay una opción mucho más eficiente de la copia de la memoria (DMA) para los sprites. Por lo tanto, se suele utilizar 0x200-0x2ff de la memoria RAM para almacenar los valores, a continuación, esperar a V-Blank, a continuación, escribir 0 a 0x2003, a continuación, escribir 2 en 0x4014 (DMA de RAM 200, y todos los datos de sprite se transferirá a la memoria de sprite (OAM).

Doug expone lo anteriormente mensionado en la sección Sprites, de la cual se basa el presente contenido. El ejemplo de código corresponde a la lección 4.

6. Lectura de controles, mandos o Joypads

El joypad uno corresponde a la direccón 0x4016, y el joypad dos a 0x4017. El orden de lectura de los bits de mayor a menor peso es A, B, SELECT, START, UP, DOWN, LEFT, RIGHT. Un ejemplo básico se explica en Input, cuyo código corresponde a la lección 5.

7. La tabla de tiles CHR

El área de la memoria que se conecta a la PPU, que contiene las imágenes usadas para dibujar los fondos y sprites.

La memoria se divide en secciones de 16 bytes llamadas tiles, se dan en dos planos de 8 bytes para describir la imagen de 8x8 pixeles, esto se puede ver en detalle en la tabla 3.

La tabla de patrones (CHR) se divide en dos secciones de 256-tiles, en 0000-0FFF la sección izquierda, y 1000-1FFF la derecha, estos nombres se dan debido a como los trabaja un emulador. Cada sección se entiende como

Planos	Patrón	Resultado
0xx0=41	01000001	
0xx1=C2	11000010	
0xx2=44	01000100	
0xx3=48	01001000	
0xx4=10	00010000	
0xx5=20	00100000	. 1 3
0xx6=40	01000000	11 . . . . 3 .
0xx7=80	10000000	. 1 . . . 3 . .
		. 1 . . 3 . . .
0xx8=01	00000001	. . . 3 . 22 .
0xx9=02	00000010	. . 3 2
0xxA=04	00000100	. 3 . . . . 2 .
0xxB=08	00001000	3 222
0xxC=16	00010110	
0xxD=21	00100001	
0xxE=42	01000010	
0xxF=87	10000111	

Tabla 3: Patrones de pixeles en un tile [4]

una imagen de 128x128 pixeles o 16x16 tiles con 2 bits de profundidad (4 indices de color).

Referencias

[1] Doug (dougeff) , *NES Programming with cc65*. https://nesdoug.com/. Consultada el 21 de agosto de 2017.

[2] Nesdev wiki , *CPU memory map*. http://wiki.nesdev.com/w/index.php/CPU_memory_map. Consultada el 21 de agosto de 2017.

[3] Nesdev wiki , *PPU memory map*. http://wiki.nesdev.com/w/index.php/PPU_memory_map. Consultada el 21 de agosto de 2017.

[4] Nesdev wiki , *PPU pattern tables*. http://wiki.nesdev.com/w/index.php/PPU_pattern_tables. Consultada el 21 de agosto de 2017.

[5] YY , *YY-CHR*. http://www.romhacking.net/utilities/119/. Consultada el 21 de agosto de 2017.

[6] Shiru , *Shiru's Stuff*. https://shiru.untergrund.net/software.shtml. Consultada el 21 de agosto de 2017.

[7] FCEUX team , *FCEUX The all in one NES/Famicom/Dendy Emulator* . http://www.fceux.com/web/home.html. Consultada el 21 de agosto de 2017.

[8] Ullrich von Bassewitz , *cc65 coding hints*. http://www.cc65.org/doc/coding.html. Consultada el 21 de agosto de 2017.

[9] FirebrandX , *The NES Composite Palette Project*. http://www.firebrandx.com/nespalette.html. Consultada el 21 de agosto de 2017.

[10] Doug (dougeff) , *A little color*. https://nesdoug.com/2015/11/19/5-a-little-color/. Consultada el 21 de agosto de 2017.

[11] Nesdev wiki , *PPU OAM*. http://wiki.nesdev.com/w/index.php/PPU_OAM. Consultada el 23 de agosto de 2017.