Es evidente que, cuando los videojuegos apenas se estaban empezando a desarrollar, se debieron de hacer acomodaciones correspondientes para poder hacer más evidente la innovación que se estaba llevando a cabo con cada salida de una consola nueva. Por ejemplo:

Mega Man 1:

Mega Man 1 presentaba algunas peculiaridades a la hora que se juega el juego: el sonido traba las ocurrencias en la pantalla, los enemigos se recargan cuando dejamos y volvemos a ver el punto en el que aparecen, las transiciones de niveles son lentas y no podemos ver a los elementos de la pantalla hasta que la transición se detiene, así como hay elementos limitados en la pantalla a la vez. ¿Cómo se hace esto? ¿Por qué ocurre?

Para investigar con respecto al funcionamiento de la gestión de pantallas en videojuegos 8-bit, nos enfocaremos en una plataforma para jugar videojuegos que era de las más populares en su día: la Nintendo NES.

El procesamiento de sonido:

En el caso de la Nintendo NES, la CPU estaba basado en un procesador MOS Technology 6502, el cuál contaba con unas cuántas funcionalidades que permitían interactuar con el audio, pues había una unidad de procesamiento de audio (Audio Processing Unit, or APU) que venía integrada en el chipset RPA203, donde se también se encontraba el procesador 6502. Esto significaba que, aparte de todas las direcciones de memoria y registros mapeados directamente a memoria que ya vienen de por sí en el procesador 6502, también se dedicaron 22 registros total y completamente únicos para el procesamiento de sonido (desde cuestiones como volumen y generación de sonidos, hasta reproducción de muestras de sonido).

Originalmente, teníamos pensado que el procesamiento de sonidos afectaba directamente a los procesos mostrados en pantalla en el juego de Mega Man, más que nada por el efecto de cuando se llenan las barras de vida, en el cuál podemos apreciar que toda la pantalla se congela. Sin embargo, un poco de investigación mostró que realmente esto es un efecto programado por los desarrolladores, y que lo único que el sonido afecta es a otros sonidos (la música de fondo se distorsiona cuando escuchamos efectos en pantalla, por lo que la música siempre puede sonar diferente de cómo se compuso). Cada uno de los sprites se sigue moviendo de manera fluida incluso si hay sonido de fondo, mostrando una falta de dependencia entre las instrucciones de sonido y las instrucciones mostradas a pantalla.

Sin embargo, sigue habiendo complicaciones técnicas: por ejemplo, en el nivel de Electric Man, hay una sección donde debemos de subir unas escaleras que tienen como obstáculos renglones en donde se activan campos de electricidad. El efecto de estos campos es uno parpadeante, pero el problema viene del hecho que, cuando se muestra la electricidad “parpadeante”, también parpadean tanto las barras de poder

y de vida de nuestro personaje, así como el puntaje alto mostrado en la parte superior de la pantalla.

Entonces, la pregunta es: ¿Cómo es que los gráficos se procesan en la NES?

La PPU:

La unidad de procesamiento de imágenes (Picture Processing Unit, o por sus siglas en inglés, PPU) es uno de los componentes que constituye al hardware de la NEs. Es un componente diseñado específicamente para el envío de señales a televisiones, y que contiene 10 kilobytes de memoria.

Es fácil asumir que estos 10 kilobytes de memoria son los que se utilizan para poder procesar todo lo que compete con lo que se ve en la pantalla, ¿no es así? Sin embargo, esto se encuentra lejos de la verdad completa, pues 8 kilobytes de memoria están destinados al Game Pak (cartuchos en los que se guardaba el software de los juegos), significando que se guarda la información de cada "teja" y el fondo de la escena que se está mostrando, mientras que los 2 kilobytes que sobran son utilizados un par de mapas como máximo.

No hemos mencionado temas como la cuestión de la presentación de colores, o la información en la que se guardan los sprites como su posición, los colores que abarcan los mismos, esto se debe a que la NES ya tiene espacios de direccionamiento más pequeños que se encargan de esto, uno de estos siendo el OAM (Object Attribute Memory).

Evaluación de sprites de la PPU:

La PPU se encarga de revisar y evaluar qué sprites deben de aparecer en la pantalla por línea de escaneo durante cada fotograma que se procesa, significando que está calculando 262 (240 efectivos) líneas de escaneo por fotograma procesado. Esto lo hace en tándem con el OAM, y es un paso por separado a la renderización de sprites. (**Nota de importancia:** la cantidad de sprites por línea de escaneo

En resumen, lo que la OAM hace por cada vez que evalúa los sprites es:

1. Borra todos los sprites en la lista de sprites por ser conseguidos.
2. Empieza a revisar la información que se encuentra guardada en el OAM, buscando primordialmente la información sobre los sprites disponibles que se van a imprimir a pantalla durante esta línea de escaneo. Agarra específicamente 8 por scanline (como máximo, si encuentra menos entonces esta cantidad será la agarrada).
3. Si y sólo si se encuentran 8 sprites en una scanline, revisa si hay más sprites para determinar si se debe de activar la bandera de desbordamiento de sprites (importante).
4. Habiendo obtenido los detalles de los sprites obtenidos, determina cuáles tienen información en la línea de escaneo actual y dónde se ubica cada pixel.

La razón por la que el paso tres es particularmente importante es porque, si bien la lógica del algoritmo en sí no está mal, el problema viene siendo la implementación. En cuanto se tienen 8 sprites procesados, el procesador de imágenes comienza a buscar subsecuentemente si es que se encuentra un noveno sprite. Sin embargo, una mala incrementación de una variable m (en donde m es un contador en el arreglo OAM[n][m]) da como resultado que se empiecen a hacer escaneos de manera "diagonal", lo cuál empieza a arrojar resultados erróneos.

Sin embargo, toda esta información nos permite, por lo menos, formular una respuesta hipotética a la pregunta que teníamos desde antes: es posible que la razón por la que los sprites en la pantalla parpadeen cuando se presentan los rayos de electricidad es debido a que hay tantos objetos en pantalla que simplemente haya una sobrecarga de sprites en pantalla y por tanto estos conflictúen los unos con los otros, y por ende se dé el fenómeno de parpadeo.

Un usuario en Stack Exchange, *NobodyNada (2016),* acierta lo siguiente: "La PPU de la NES solo puede amasar 64 sprites por pantalla y 8 por línea de escaneo. Tratar de amasar más hará que algunos se tornen invisibles. Arruinaría el juego si algunos enemigos fueran invisibles por el hecho de que hay demasiados de estos, por lo que los desarrolladores programaron a los sprites para que cambiaran de orden en la lista de sprites. En vez de que algunos sprites *siempre* sean invisibles, los sprites invisibles cambian cada fotograma, causando el efecto de parpadeo."

Es decir, que el fenómeno de parpadeo también es un efecto intencional causado por los desarrolladores de juegos para la NES. Sin embargo, no hace falta mencionar que la intencionalidad de este efecto viene de una necesidad de flanquear un problema, mientras que la del audio es más un efecto para dar al usuario a entender ciertos cambios de estado. Por encima de todo esto viene la pregunta: ¿Cómo es que los sprites van cambiando en pantalla? ¿Cómo se decide qué sprites aparecen en determinado momento y cuáles no?

Prioridad de sprites de la PPU:

En concreto, esta parte no necesariamente responde a la pregunta anteriormente planteada, pues el efecto de parpadeo se consigue cambiando la lista de sprites constantemente. Sin embargo, esta sección hablará un poco más sobre toda la cuestión que tiene que ver con decidir: De los sprites que actualmente tenemos seleccionados durante el fotograma actual, ¿cuáles se muestran en frente de otros? ¿Cómo determinamos a los objetos del fondo comparado con los del frente de la pantalla?

La forma en la que los sprites del fondo son decididos es de la siguiente manera: El OAM se puede ver como un arreglo bidimensional *OAM[n][m].* Con base en esto, el

mismo tiene “reglas de prioridad” con las cuáles decide el posicionamiento de los objetos. Estas dos son:

1. ¿Qué objeto tiene el menor valor de n? Es decir, ¿qué objeto está “más a la izquierda” en el arreglo de la OAM?
2. ¿Está puesto el bit opaco de fondo?

Estas dos reglas permiten que se muestren objetos en pantalla según su prioridad de fondo, y podemos visualizar estos ejemplos fácilmente en la primera pantalla del nivel de “Ice Man”, en el cuál podemos ver que hay unos enemigos flotantes que a pesar de su movimiento, se muestran frente a la pantalla en todo momento hasta que estos son vencidos.

Finalmente, en Mega Man podemos ver que, cuando cambiamos de pantalla, debemos de esperar a que la pantalla se mueva hasta su nueva posición, y luego se encarga de cargar a los enemigos en pantalla. ¿Cómo es que funciona esto?

Scrolling en la NES:

El caso más simple de scrolling que vemos se puede resumir algorítmicamente en pasos que consisten en lo siguiente:

1. Se encuentran las posiciones X y Y de la esquina superior izquierda (la parte visible del mapa en el que nos encontramos).
2. Se escriben los bits bajos de X y Y a PPUSCROLL (registro de la PPU que se encarga de cambiar la posición de la pantalla por medio de la selección de pixeles en la nametable).
3. Los novenos bits de X y Y se escriben en PPUCTRL.

Este es el proceso algorítmico más básico que se tiene para generar un efecto de movimiento de pantallas, donde se representa el movimiento básico de un personaje en un plataformero 2D.

Sin embargo, el método de scrolling utilizado en las transiciones de pantalla en Mega Man es diferente al scroll básico, ya que emplea una técnica llamada Split X Scrolling, en la que podemos visualizar la principal diferencia de que el valor de X utilizado se actualiza de inmediato, permitiendo dar un efecto de mayor velocidad y reposicionamiento de los sprites según se lleve a cabo el movimiento. Según esto, podemos inferir que el registro $2000, o el PPUCTRL, es el que lleva a cabo el procesamiento de los enemigos en pantalla una vez acabe el scroll.

Referencias:

1. *2A03*. (2023). NESdev Wiki. https://www.nesdev.org/wiki/2A03
2. (n.d.). *MOS 6500 Microprocessors* [Review of *MOS 6500 Microprocessors*]. Www.webarchive.org; Web Archive. Retrieved April 3, 2025, from

https://web.archive.org/web/20220629085756if_/http://archive.6502.org/

datasheets/mos_6500_mpu_nov_1985.pdf

3. *APU*. (n.d.). NESdev Wiki. https://www.nesdev.org/wiki/APU
4. *PPU*. (2023). NESdev Wiki. https://www.nesdev.org/wiki/PPU

5. Copetti, R. (2019, January 25). *NES Architecture | A Practical Analysis*. Rodrigo's Stuff. https://www.copetti.org/writings/consoles/nes/
6. Diskin, P. (2004). *Nintendo Entertainment System Documentation*. https://www.nesdev.org/NESDoc.pdf