UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Plataforma de emulación multiconsola con soporte para juego online basado en Rollback Netcode

Informe de Proyecto de Grado presentado por

Carlos Daniel Acuña Di Giorno

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisor

Héctor Cancela

Montevideo, 2 de marzo de 2025

Agradecimientos

Quiero agradecer, en primer lugar, a los docentes supervisores que me ayudaron a lo largo de este proyecto: a Santiago Iturriaga, con quien lo inicié y que me ayudó a definirlo, y a Héctor Cancela, cuyo apoyo fue fundamental en la etapa final y en la elaboración de este documento.
En segundo lugar, agradezco a todos mis compañeros, amigos y docentes de la Facultad de Ingeniería con quienes he trabajado y que han sido parte de este camino.
También quiero expresar mi gratitud a todas aquellas personas que dedicaron su tiempo a estudiar y documentar el funcionamiento de la NES, y a Tony Cannon por desarrollar el método que utilicé en este proyecto. Sin el trabajo de todos ellos, este proyecto no habría sido posible.
Gracias también a mi amigo Sebastián García que escuchaba con paciencia mis lamentos cuando habían dificultades en el proyecto. Por último, y principalmente, agradezco a mi familia por su apoyo incondicional durante todos estos años de carrera.
Gracias.

Resumen

Este proyecto consiste en la implementación de una plataforma de emulación desarrollada en el lenguaje C++ para el sistema operativo Windows, diseñada con una arquitectura x64, un diseño modular y una interfaz gráfica de usuario de uso intuitivo. La plataforma esta diseñada para la integración sencilla de emuladores de sistemas de los años 80 y 90, como Sega Genesis y Super Nintendo, facilitando su extensión a otros sistemas o dispositivos sin requerir modificaciones complejas al código base.

Como parte de la plataforma, se desarrolló un emulador del sistema Nintendo Entertainment System (NES), conocido como Family Game en Uruguay. Este emulador fue creado desde cero e integrado exitosamente en la plataforma.

El proyecto también implementó una funcionalidad de juego online P2P basada en técnicas predictivas, utilizando el *Rollback Netcode* para reducir la latencia percibida entre dos jugadores. Esto requirió un diseño optimizado tanto del emulador como de la plataforma para garantizar un desempeño adecuado en este contexto.

Adicionalmente, se desarrolló una funcionalidad de entrada/salida que permite la interacción programática con la plataforma, habilitando proyectos futuros que experimenten con inteligencia artificial aplicada a videojuegos clásicos.

El desarrollo incluye una introducción a los fundamentos de la emulación, un repaso de la arquitectura del NES y una revisión del estado del arte en *Gaming Networking*. Esto permitió identificar y adoptar soluciones como el *Rollback Netcode* en el diseño de la plataforma y del emulador, asegurando un enfoque moderno y funcional.

Palabras clave: Emulación, Gaming Networking, Rollback, Arquitectura de Computadoras, Redes de Computadoras

Índice general

Capítulo 1

Introducción

En este informe queremos presentar el proceso de investigación llevado a cabo para desarrollar un emulador de Nintendo Entertainment System integrado a una plataforma de emulación con soporte a juego online utilizando un sistema predictivo para enmascarar la latencia entre ambos jugadores, escalable a múltiples sistemas de videojuegos de los años 80 y 90.

1.1. Contexto y Motivación

En los últimos años, la preservación tanto de hardware como de software de sistemas clásicos de videojuegos y microcomputadoras ha adquirido una relevancia creciente. Sistemas icónicos como el Nintendo Entertainment System (NES), el Commodore 64 o diversas máquinas recreativas arcade que marcaron a generaciones enteras, llevan décadas fuera de producción comercial. A medida que el tiempo avanza, cartuchos de juego, los circuitos integrados y otros componentes esenciales comienzan a deteriorarse, algunos más rápidamente debido a defectos de diseño o materiales poco duraderos, dificultando el acceso a estos sistemas.
La emulación emerge como una solución frente a este problema, al permitir que el software original de estos sistemas pueda ejecutarse en plataformas modernas. Esto se logra simulando el comportamiento del hardware original mediante software, lo que elimina la necesidad de migrar manualmente cada pieza de software a sistemas contemporáneos. (Johnson y Antonelli, 2020)
Más allá del disfrute recreativo, la emulación abre las puertas a nuevas posibilidades, como el estudio técnico de estos sistemas, su uso en investigaciones de inteligencia artificial aplicadas a videojuegos, o implementando mejoras y funcionalidades que estos sistemas no tenían, por ejemplo, un sistema de juego online ya que estas solo tenían multijugador de manera local. Las motivaciones para desarrollar un emulador son múltiples, además del factor nostálgico y apreciación por la historia de la tecnología, implementar un emulador presenta desafíos tales como comprender detalladamente la arquitectura de la máquina

objetivo incluyendo procesadores, memoria, buses, unidades gráficas,sistemas de entrada/salida, y otras unidades de hardware. Esto proporciona una base sólida en conceptos de bajo nivel, como el funcionamiento de los registros, las interrupciones y el manejo de ciclos de reloj, la sincronización entre distintos componentes y programar y operar a nivel de bits.

Una vez decidido a desarrollar un emulador que incluya multijugador online solo queda elegir el sistema objetivo que cumpla los requerimientos de que la complejidad de su arquitectura sea abarcable para que una persona desarrolle el emulador y que tenga soporte para al menos dos jugadores. La Nintendo Entertainment System (NES) o más conocida comúnmente en Uruguay como Family Game es una de las consolas más conocidas y presentes en la cultura de nuestro país y tiene la capacidad para dos jugadores, lo cual la hace una gran elección para experimentar esta técnica en un emulador.
Además, si pensamos un poco más en grande y a futuro, desarrollar la plataforma de manera modular nos permitiría incluir a futuro más sistemas que aprovechen esta funcionalidad reutilizando el código, además de traer otros beneficios como:

- Separación de responsabilidades y un código más claro.
- Fomentar la colaboración y contribución externa
- Aislamiento de problemas y mejoras continuas
- Preparación para tecnologías futuras
- Escalabilidad para diferentes propósitos entre otros beneficios, y como muestra del ultimo mencionado, incluir un sistema de entrada salida para usos en investigación como por ejemplo una Inteligencia Artificial .

1.2. Objetivos

Por lo tanto, los objetivos y resultados esperados de este proyecto son :

1. Diseñar una plataforma con diseño modular y fácilmente escalable en el lenguaje C++ para que funcione en sistemas x86 con sistema operativo Windows.
2. Soporte para juego multijugador a través de internet con un modelo P2P, utilizando Rollback Netcode para reducir la percepción de latencia..
3. Capacidad de emular eficientemente la consola Nintendo Entertainment System (NES) de 8 bits.
4. Sistema de entrada salida que permita la interacción programática con la plataforma.

1.3. Organización del Documento

El documento está organizado para tener una visión clara y progresiva del tema tratado. Comienza con una introducción a los conceptos fundamentales de la emulación, así como una revisión del estado del arte en esta área, lo que se aborda en el Capítulo 2. En el Capítulo 3 se presenta en detalle el sistema que vamos a emular, el Nintendo Entertainment System (NES), exponiendo su arquitectura y características principales.
Posteriormente, el Capítulo 4 introduce el tema del Gaming Networking, dando una introducción y destacando los avances actuales en esta disciplina, también se revisa los antecedentes de los emuladores más influyentes y explora si son compatibles con sistemas de juego online. El Capítulo 5 se enfoca en el método Rollback, explicando su funcionamiento y relevancia en el contexto del juego online.
Con esta base, el Capítulo 6 se dedica a la presentación de nuestra propia plataforma, detallando sus características y objetivos.
La validación y evaluación del desempeño de la plataforma se realiza en el Capítulo 7, donde se presentan las pruebas realizadas y sus resultados. Finalmente, en el Capítulo 8 se exponen las conclusiones junto con posibles direcciones para trabajos futuros.

Capítulo 2

Introducción a la Emulación y Relevamiento del Estado del Arte.

En este capítulo vamos explorar el concepto de emulación, describiendo su importancia. También vamos revisar las técnicas clásicas de emulación para los componentes de hardware más importantes y los desafíos técnicos asociados, por ultimo daremos un repaso de las técnicas más modernas.

2.1. ¿Qué es un Emulador?

La definición estándar de emulación es "intentar ser igual o mejor que algo o alguien". En el contexto de la informática, la emulación se refiere a replicar el comportamiento de un dispositivo de hardware mediante software o a replicar el comportamiento de un software utilizando otro software o hardware.(del Barrio, 2001) Esta definición es algo amplia, ya que la emulación puede abarcar desde sistemas operativos hasta dispositivos específicos, como tarjetas de sonido.(Johnson y Antonelli, 2020)

No se debe confundir con el termino simulador ya que ambos se refieren a distintos objetivos, un simulador se define como una aplicación cuyo objetivo principal es modelar con precisión el estado interno de un sistema. Por ejemplo un simulador de vuelo, donde diversos fenómenos físicos son representados matemáticamente, complementados con una representación visual interactiva. Por otro lado, un emulador no busca reproducir el estado interno del sistema, sino replicar su comportamiento externo tal como lo percibiría el usuario.(Richeson, 2017) Claramente las líneas entre emulación y simulación pueden ser borrosas y un emulador puede tender a ser simulador tanto como el desarrollador quiera.
La prioridad en un emulador es interpretar instrucciones de manera eficiente y generar una salida visual fiel al sistema original, sin necesidad replicar exacta-

mente los métodos empleados por el hardware real para lograrlo.

Las técnicas utilizadas para emular hardware son diferentes a las de emular software. Cuando se emula un dispositivo de hardware con otro hardware, el objetivo principal es que los valores de salida del nuevo dispositivo coincidan con los del original para los mismos valores de entrada. Esta tarea se asocia a la ingeniería electrónica y el diseño de circuitos integrados. En el caso de cuando se emula un sistema operativo sobre otro, se trata de un problema principalmente de software. El concepto de emuladores está relacionado con el de máquinas virtuales (VM), como Java o Docker. Una máquina virtual es un entorno que no existe como hardware físico, sino que se implementa por software sobre una computadora real.

En este documento haremos énfasis en emuladores de computadoras o consolas en software. Las computadoras de origen, que son las que estaríamos tratando de emular, pueden ser desde microcomputadoras de 8 bits como el Sinclair Spectrum, Commodore 64 o Amiga, hasta consolas de videojuegos como la Nintendo NES, Sega Genesis, o Super Nintendo, incluso maquinas Arcade (conocidas popularmente como maquinitas o fichines).

Estas máquinas, aunque varían mucho en sus especificaciones, comparten una serie de características comunes, como el uso de procesadores baratos de 8 bits (6502, Zilog Z80, Intel 8080 and 8085, Motorola 6809), 16 bits (Motorola 68000), y en sistemas más modernos arquitecturas RISC de gama media (como ARM, MIPS o PowerPC) lo cual permite la reutilizacion de codigo al emular estos procesadores para emular otros sistemas que los usen.
Las computadoras de destino suelen ser PCs modernas basadas en x86 o x64, que pueden ejecutar sistemas operativos como Windows, Linux o macOS, o incluso emuladores portados a consolas de videojuegos como PlayStation, Xbox, Nintendo Switch y dispositivos móviles como smartphones. Las capacidades de la computadora de destino determinan el hardware que se puede emular y el rendimiento general de la emulación. Con el avance de las PCs modernas, que ahora tienen procesadores multihilo de alto rendimiento y GPUs especializadas, es posible emular consolas de la era 3D, como la Nintendo 64, Dreamcast o consolas de la generación PlayStation 2 y posteriores. Estos emuladores requieren técnicas avanzadas que iremos exponiendo en las siguientes secciones.(del Barrio, 2001)

2.2. El Proceso de Emulación

Antes de abordar el proceso de emulación, es importante establecer un contexto fundamental: los sistemas que buscamos emular están construidos, en su esencia, sobre la arquitectura de Von Neumann.

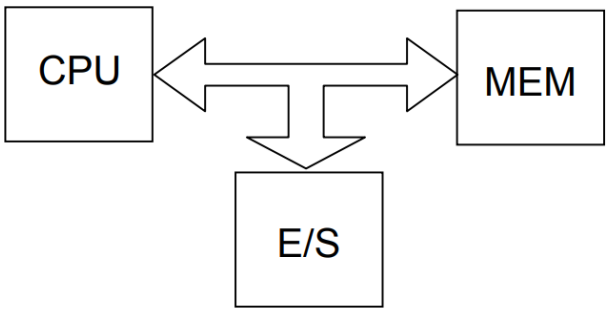
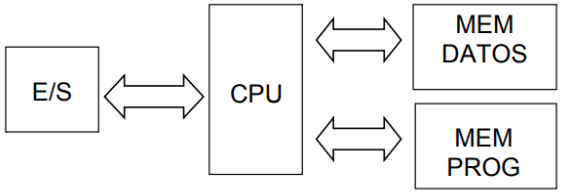


Figura 2.1: Arquitectura de Von Neumann

Aunque algunos de estos sistemas pueden incorporar características propias de la arquitectura Harvard, como la separación de memoria para datos e instrucciones en componentes específicos, esto no altera el hecho de que Harvard es, en última instancia, una extensión o especialización de los principios básicos de Von Neumann (INCO, 2011).



[H]

Figura 2.2: Arquitectura de Harvard

Esto es muy importante, ya que nos ayuda a entender tanto la forma en que estos sistemas operan como a identificar los componentes que tendremos emular.

La CPU (Unidad Central de Procesamiento) es el componente encargado de controlar el funcionamiento de todo el sistema, realizar la mayoría de los cálculos y ejecuta las tareas más demandantes.

La memoria es el componente donde se almacenan los datos e instrucciones con los que la CPU realiza calculas y ejecuta. Existen varios tipos de memoria, siendo los más básicos la RAM, que permite leer y escribir datos, y la ROM, que solo permite lectura. Además, hay muchas variantes de estas memorias, cada una adaptada a necesidades específicas.

Los buses son conjuntos de líneas eléctricas que permiten la comunicación entre la CPU y los demás dispositivos conectados al sistema. Normalmente hay más de un bus. Estos se dividen comúnmente en tres categorías: el bus de direcciones, que especifica qué dispositivo o parte del dispositivo debe ser accedido; el bus de datos, que transporta la información entre la CPU y los dispositivos; y el bus de control, que se encarga de transmitir señales adicionales para coordinar y regular el acceso al bus. Como el bus es compartido por todos los dispositivos, solo uno puede ser accedido por la CPU a la vez, o bien, dos dispositivos pueden intercambiar información cuando la CPU no lo está utilizando.

Por otro lado, los dispositivos de entrada y salida (I/O) abarcan una amplia gama de funciones y equipos. Incluyen desde dispositivos de almacenamiento secundario, como discos duros o impresoras, hasta monitores, parlantes, dispositivos de entrada como teclados, ratones o controles, hardware de comunicación como tarjetas de red. En esencia, cualquier hardware que pueda ser controlado por la CPU entra en esta categoría.

A la hora de implementar un emulador o los componentes del mismo existen dos paradigmas, Low-Level Emulation (LLE) y High-Level Emulation (HLE). En un enfoque de LLE, se intenta replicar con precisión el comportamiento del hardware original, procesando cada instrucción tal como lo haría el hardware. En cambio, en un enfoque de HLE, se abstraen estas operaciones, reemplazando funcionalidades específicas del hardware o del sistema operativo por implementaciones nativas de alto nivel en el sistema anfitrión. (EmulationGeneralWiki, 2024) La elección entre ambos enfoques depende del objetivo del emulador y del nivel de fidelidad o rendimiento que se desee lograr. Es importante mencionar estos enfoques antes de zambullirnos en el tema de emulación, sin embargo profundizaremos un poco más en ellos más adelante cuando hablemos de emulación en sistemas más modernos, por el momento introduzcamos las ideas de manera simple con modelos tradicionales.

El esqueleto central de un emulador es un bucle, ya que así funciona una computadora. La CPU opera constantemente siguiendo un ciclo de fetch, decode y execute. Dado que la CPU actúa como el núcleo de la emulación, la emulación de todos los demás dispositivos se organiza en torno a su funcionamiento. En la realidad, todos los dispositivos del sistema trabajan en paralelo y estos se van sincronizando mediante escrituras en registros o direcciones especiales de la memoria, la CPU ejecuta instrucciones mientras el hardware gráfico genera la imagen en pantalla y el hardware de sonido reproduce audio. Sin embargo, la mayoría de los emuladores están diseñados para ejecutarse en máquinas de

un solo procesador, lo que impide emular estos dispositivos de forma paralela además de que dificultaría el desarrollo y en muchos casos la diferencia de poder de computo entre el dispositivo siendo emulado y el dispositivo emulando es tan grande que hacerlo de manera secuencial es humanamente indistinguible.

Por lo tanto el bucle principal de un emulador tendría la siguiente forma:

```
while (isRunning) {
    // 1. Obtener la instruccion actual desde la memoria
    Instruction instruction = fetch(CPU.PC);

    // 2. Decodificar la instruccion
    DecodedInstruction decoded = decode(instruction);

    // 3. Ejecutar la instruccion
    execute(decoded);

    // 4. Actualizar dispositivos I/O (graficos, sonido, etc.)
    updateGraphics();
    updateSound();
    handleInput();

    // 5. Sincronizar con el tiempo de frecuencia real
    synchronize();
}
```

Entremos en detalle a hablar de cada uno de estos componentes.

2.2.1. Emulación de CPU

Como mencionamos anteriormente el CPU es el núcleo del sistema que buscamos emular, y su correcta emulación es una de las tareas más vitales en este proceso. Además si estamos emulando un sistema de videojuegos, estos son máquinas "en tiempo real", lo que significa que los juegos dependen de que la capacidad de ejecutar instrucciones por segundo sea igual o mayor que la máquina original. Normalmente la emulación de una CPU requiere mucho mayor poder de computo en comparación a la CPU que se esta emulando y su funcionamiento dependerá de que tan eficiente sea nuestro código y que tan precisa sea la emulación(Kuchera, 2011), por lo que optimizar el rendimiento es fundamental.

En la mayoría de los sistemas, se emula una sola CPU, pero algunos dispositivos cuentan con más de una unidad de procesamiento. En ciertos casos, existe una relación maestro-esclavo entre las CPU, mientras que en otros operan de forma similar a un sistema multiprocesador. Para implementar esto, el algoritmo básico simple seria integrar múltiples llamadas a funciones de fetch decode y execute que intercalen la emulación de las diferentes CPUs. Sin embargo, en algunos casos es necesario emular también el hardware que gestiona

la sincronización y comunicación entre ellas lo cual pueden resultar en arquitecturas increíblemente complejas como es el caso de Sega Saturn(Copetti, 2022) .

La emulación de una CPU se centra únicamente en los aspectos relacionados con sus cálculos y su forma de operar. Normalmente, esto significa emular solo la ejecución de las instrucciones del programa . Aunque las CPUs modernas poseen características avanzadas, como la ejecución fuera de orden, la superescalaridad o los pipelines, estas complejidades internas no suelen ser necesarias para la emulación.

Existen dos enfoques principales para emular una CPU que vamos a exponer a continuación.

Interpreter

Este método consiste en tomar el código fuente (representado como bytes) y, mediante un ciclo de "fetch-decode-execute", leer cada byte, *interpretar* la instrucción correspondiente y ejecutarla (del Barrio, 2001). Es el enfoque más simple y directo para desarrollar un emulador, pero también es el más demandante de poder en términos de uso de CPU. En el Listing 2.1 podemos ver la estructura de este enfoque.

Listing 2.1: Bucle principal de un emulador interpreter

```
while ejecutando:
    // Obtengo la opcode del program counter actual
    opcode = obtenerInstruccion(pc++)
    //Decodifico el opcode y lo ejecuto.
    switch opcode:
        caso OPCODE_A:
            ejecutarOperacionA()
        caso OPCODE_B:
            ejecutarOperacionB()
        caso OPCODE_C:
            ejecutarOperacionC()
        ...
        caso OPCODE_N:
            ejecutarOperacionN()
```

Un emulador *interpreter* es más lento en comparación con otras formas de emulación de CPU. Esto se debe al costo adicional de la decodificación y planificación de las instrucciones. Los emuladores escritos en lenguajes de alto nivel como Java o C++ suelen tener un mayor costo adicional debido al costo de implementar ciertas instrucciones básicas de la CPU. Por ejemplo, las operaciones que implican el cálculo de banderas (flags) son muy costosas y es complicado aprovechar al máximo las capacidades del procesador anfitrión. Si hablamos de CPUs cuyas frecuencias están en el rango de 1-20 MHz con computadoras mo-

dernas el overhead de este método es más que aceptable y tiene sentido utilizar este método más simple para emularlos. (mamedev, 2008)

A partir del intérprete básico, existen implementaciones que buscan aumentar su rendimiento. Dado que uno de los mayores costos en un intérprete es overhead de la decodificación de instrucciones, la primera optimización apunta a reducir este costo. Aprovechando la localidad del código (es decir, que el mismo código se ejecuta varias veces), tiene sentido almacenar información sobre las instrucciones decodificadas y reutilizarla para evitar decodificar la misma instrucción más de una vez. Este tipo de intérpretes se conocen como emuladores *threaded*.

Los emuladores *threaded* están basados en un tipo de código llamado "threaded code". El flujo de ejecución se gestiona mediante un arreglo de punteros a funciones. Un emulador *threaded* decodifica la instrucción la primera vez que se encuentra y almacena un puntero a la función o código que implementa dicha instrucción en un arreglo. La próxima vez que se encuentra la misma instrucción, se accede al arreglo y se realiza un salto directo al código.

Binary Translation

Con la llegada de los DSPs (Procesadores de Señal Digital) y los procesadores RISC de alta velocidad, el overhead del método *Interpreter* se ha vuelto demasiado elevado incluso para los procesadores modernos. El gran problema con esto es que estos procesadores tienden a operar a velocidades mucho más altas, que van desde los 20 MHz hasta cientos de MHz. Y para empeorar el problema, cada instrucción emulada a menudo se organiza en un pipeline, de modo que solo requiere un ciclo emulado por instrucción, por lo que pasa de necesitar ejecutar varios cientos de miles de instrucciones por segundo a necesitar ejecutar varias decenas o cientos de millones de instrucciones por segundo. Para solucionar este problema nace este segundo enfoque que es la traducción binaria de código. Más rápido que simplemente obtener cada opcode de la CPU emulada y ejecutar una función o código que implemente la función de esa instrucción, sería traducir ese opcode a un opcode de la CPU de destino. Eso es lo que pretende hacer este método.
La traducción se realiza en bloques de código. Las razones para traducir en bloques, en lugar de traducir todo el programa de una vez, son diversas. Desde el hecho de que no todo el código se puede conocer de antemano, hasta el hecho de que debe haber puntos para detener la emulación y facilitar el proceso de traducción, que se hace más rápido trabajando en bloques.

Existen dos formas de traducir el código: estática o dinámica.
La forma estática se refiere a que antes de comenzar a usar el programa en el emulador, el programa es completamente traducido a la CPU de destino. Funciona como si el programa fuera un código fuente de un lenguaje de alto nivel que se estuviera compilando por el compilador de este lenguaje.
La recopilación estática no se utiliza mucho en emuladores, ya que la idea detrás de un emulador es ser dinámico en lugar de estático. Y en la mayoría de los casos,

una traducción estática no es posible o es difícil de implementar. Por ejemplo, traducir estáticamente los miles de programas posibles que una computadora puede ejecutar sería un problema. En otros sistemas como los arcade, esta técnica podría ser útil ya que cada arquitectura solía correr un solo juego o una librería relativamente pequeña.

En la forma dinámica la traducción se realiza en tiempo de ejecución, se traduce el bloque actual y se ejecuta sobre la marcha.Cuando el código se traduce por primera vez, se almacena en una caché de traducción y se reutiliza cada vez que se tiene que ejecutar el mismo código más tarde. El proceso de traducción puede ser lento en comparación con el tiempo de simplemente ejecutar el código o interpretarlo, pero si solo se realiza una vez (o se realiza antes de la ejecución, como en la traducción estática) y el código se ejecuta muchas veces, el rendimiento final aumenta y es mucho más rápido que solo un intérprete porque las capacidades de la CPU de destino se utilizan mejor que en un intérprete. La sobrecarga debida a la decodificación de cada instrucción solo se realiza una vez, y el código para enlazar la ejecución de cada instrucción con la siguiente (el bucle de ejecución) no necesita ser emulado.

Pero el método también tiene problemas que un emulador intérprete no tiene. Uno de ellos es que no puede trabajar fácilmente con código generado dinamicamente, por ejemplo, mientras que un intérprete estándar no se ve afectado. Un traductor estático no puede detectar ni traducir código generado dinamicamente, porque este código solo se crea en tiempo de ejecución, y en un traductor estático la traducción se realiza antes de la ejecución. Un traductor dinámico podría detectar regiones de memoria con código traducido que se modifiquen. Pero la sobrecarga del proceso de recompilar muchas veces del mismo fragmento de código, y el proceso de detección de esas modificaciones, podría disminuir mucho el rendimiento del emulador.(mamedev, 2008)

Dynamic Recompilation

El método de recompilación dinámica implica recompilar en tiempo de ejecución el código máquina del programa objetivo en el código máquina de la máquina destino. En comparación con el método de interpreter más básico, ambos comparten el principio de tener que capturar un opcode, decodificarlo y realizar la función. La diferencia clave entre los dos, sin embargo, es que la recompilación dinámica guarda el código traducido en regiones de memoria conocidas como cachés, que se recuperan según sea necesario, mientras que los intérpretes pueden realizar el mismo opcode repetidamente, teniendo que traducir el opcode cada vez.

Comparado con la emulación con interpreter, ofrece mejoras muy significativas, pero también introduce problemas más difíciles de abordar. El mayor beneficio de utilizar este método es que es significativamente mas rápido que el método interpreter básico . No hay necesidad de traducir cada opcode en cada ciclo, ya que el resultado ya ha sido almacenado en caché antes. Esto significa

que el emulador simplemente puede apuntar a la caché y ejecutar el código, lo que resulta en un gran aumento de velocidad. Realmente no hay otra razón por la que usar este método en lugar de un interpreter. Si no es necesario el aumento de velocidad es mejor utilizar el método interpreter ya que introduce una muchas complejidades al emulador, como:

- Mucho más difícil de depurar: A menudo no sabremos dónde está el problema incluso si el código parece correcto. Podría ser tan simple como usar la función nativa equivocada en cuanto a bits (por ejemplo, usar de 8 bits en lugar de 16 bits), o un problema que solo surge cuando se activan las optimizaciones del compilador. Generalmente, también hay una base de código mucho más grande a mantener, lo cual tiene una mayor probabilidad de contener errores.

- Problemas relacionados con las ubicaciones de salto: Es habitual que las arquitecturas empleen instrucciones de salto para el control del flujo del programa. Esto crea un problema en una arquitectura cliente como el x86, donde las instrucciones pueden tener cualquier longitud hasta 16 bytes, lo que significa que no se sabe a dónde hay que saltar.

- Problemas con tiempos/sincronización entre componentes: En sistemas complejos, rápidamente nos toparemos con problemas al intentar sincronizar datos o con un sistema que depende de tiempos precisos. (Satti, 2016)

2.2.2. Emulación de la Unidad de Procesamiento Gráfico

En todos los sistemas destinados para interactuar con los usuarios, el sistema gráfico desempeña un papel fundamental. La visión humana es el sentido que permite procesar información con mayor velocidad y capacidad, lo que convierte a los dispositivos visuales de I/O en una de las formas más eficiente de que una computadora transmita datos al usuario. En el caso de los videojuegos, que son el enfoque principal de nuestra emulación, el sistema gráfico es tan importante como la interacción mediante controles. Aunque también exista el sonido, es secundario: se puede jugar sin sonido, pero difícilmente sin imagen y sin inputs, al menos no como entendemos los videojuegos.Los métodos de salida gráfica más comunes en la actualidad: monitores LCD o LED .

Cuando estos monitores terminan de dibujar un frame, sincronizan con el hardware de video, lo que genera una señal conocida como Vertical Sync (VSync), Vertical Interrupt (VInt) o VBlank. Esta sincronización se indica mediante una interrupción y el cambio de banderas en un registro de estado. Algunos sistemas también son capaces de generar interrupciones después de dibujar cada línea, conocidas como interrupciones de línea (Line Interrupts) o interrupciones horizontales (Horizontal Interrupts).

La actualización completa de la pantalla se realiza comúnmente a una frecuencia fija. La frecuencia más común es la de 60 Hz (60 imágenes por segundo), aunque hace pocos años en Europa la frecuencia solía ser de 50Hz debido a que

usaban la norma PAL. Por eso es común ver en sistemas no tan antiguos como Playstation 2 que vengan con selector de norma NTSC 60Hz o PAL 50Hz, aunque estos ultimos generlamente venian con problemas y funcionaban más lento que la versión NTSC. (Babalon, 2023) En consolas antiguas este dato no es un detalle menor ya que definirá la frecuencia a la funcione la consola y también puede modificar la arquitectura o componentes. El VBlank es la principal (y muchas veces única) fuente de sincronización con otros componentes como el CPU. Este intervalo de tiempo entre que termina de dibujar un frame y empieza a dibujar otro, es vital ya que será nuestro intervalo para correr la lógica del juego con los nuevos inputs de usuario, actualizar nuestro game state y cargar los gráficos para dibujarlo en el frame siguiente.

En segundo lugar, suele estar el hardware de sonido o la CPU, siendo más frecuente que el sonido sea más importante que la CPU. Algunos sistemas tienen CPUs poco potentes (como el Super Nintendo, que usa una CPU de 1 MHz), pero poseen hardware gráfico y de sonido muy avanzado. En otros sistemas, la diferencia entre la CPU y el hardware gráfico/sonido es menor (como en la Sega Mega Drive/Genesis). El hardware gráfico (y también el de sonido) está diseñado para descargar a la CPU de gran parte del esfuerzo de generación gráfica, reservándola para tareas de control y, en algunos casos, para cálculos o implementación de efectos visuales.

Hay una diferencia entre las máquinas arcade y las consolas domesticas. Las consolas domesticas usan hardware más barato y menos potente que las máquinas arcade de la época. Las máquinas arcade implementan hardware más potente, muchas veces duplicándolo o mejorándolo (por ejemplo, duplicando el número de CPUs, los chips de gráficos y sonido, mayor memoria, ROMs más grandes para los datos del juego). Esto se debe a que una consola debe ser económica para ser vendida al público general, mientras que el número de máquinas arcade es limitado y solo se utilizan en las salas arcade.

La emulación gráfica en la mayoría de las consolas ocupa más del 50 % del tiempo total de emulación, llegando a veces al 80-90 % . Esto significa que una emulación correcta y eficiente de este componente es incluso más importante que la emulación de la CPU. (del Barrio, 2001)

Cuando se comienza la emulación de un sistema, generalmente se empieza con la CPU ya que el resto de los componentes dependen de este, pero hasta que no se aborda la emulación gráfica, no se enfrenta la tarea realmente difícil de la emulación.A diferencia de la CPU, es muy difícil mostrar un bucle genérico de funcionamiento de un emulador ya que hay un enorme número de unidades gráficas diferentes que hacen las cosas de distinta manera y hacen que esta tarea sea casi imposible. A modo de introducción al problema, en la siguiente subsección presentaremos brevemente dos enfoques comunes para el manejo de gráficos 2D. Nos centraremos especialmente en los gráficos basados en tiles, ya que este es el método utilizado en el emulador de NES que estamos desarrollando.

Plain 2D Graphic Generation

En la actualidad para renderizar gráficos 2D se utiliza este tipo de hardware basado en el uso de framebuffers, donde la memoria de video almacena una matriz de píxeles que representan la pantalla. Cada píxel puede definirse con índices de paleta (usando menos memoria) o valores RGB (mayor profundidad de color y calidad). Esto contrasta con los años 80 y 90, cuando los motores basados en tiles eran necesarios debido a las limitaciones de memoria y CPU. El "blit" (transferencia de bloques) es clave en este hardware, permitiendo copiar datos gráficos al framebuffer. Aunque antes se realizaba con la CPU, ahora los GPUs lo hacen más rápido y eficientemente usando hardware especial para estas operaciones. Técnicas como el blitting transparente permiten manejar partes transparentes en imágenes, esenciales para animaciones. Sin embargo este método de almacenado y dibujado pixel a pixel en un buffer también fue utilizado en el clasico Space Invaders (del Barrio, 2001)y dado su costo computacional podíamos ver que los enemigos no eran todos movidos al mismo tiempo y a medida que estos eran destruidos el juego se aceleraba no necesariamente por un diseño del juego sino porque el CPU tenia que mover menos elementos y reducía el tiempo para ejecutar un frame. (Osbourn, 2020)

Tiled Based Graphics

Los motores gráficos basados en tiles (también conocido como patrones) y sprites fueron los predominantes en los sistemas de videojuegos de los años 80 y 90 debido a su capacidad para producir gráficos y animaciones de buena calidad sin requerir un uso excesivo de la CPU o la memoria que era muy cara en la época. Diferentes sistemas de videojuegos implementaron motores basados en tiles de maneras diversas. Para entender bien como funciona este modelo comencemos mostrando lo que es un tile.

Un tile es una imagen muy pequeña, generalmente de tamaño fijo, es este caso de 8x8 pixeles que puede usarse sola o en conjunto de otros tiles para formar una imagen más grande y compleja. En la Figura 2.3 podemos ver como usando estos 4 tiles se forma la Figura 2.4.

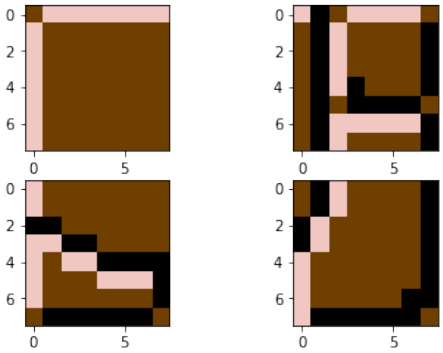


Figura 2.3: Ejemplo de un conjunto de tiles (Earl, 2018)

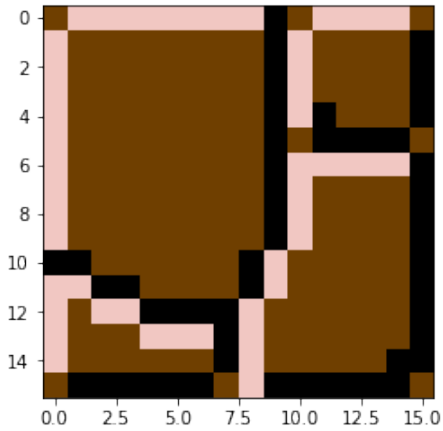


Figura 2.4: Ejemplo de imagen formada por tiles (Earl, 2018)

La motivación de poder usar estos tiles, es que son fácilmente almacenables en una memoria ROM ya que en aquella época era mucho más barata y accesible y además estos tiles puede ser usados para representar distintas cosas según la paleta que se utiliza como veremos en la Figura 2.4 del juego Super Mario Bros. donde se puede observar que tanto las nubes como los arbustos comparten los mismos tiles.

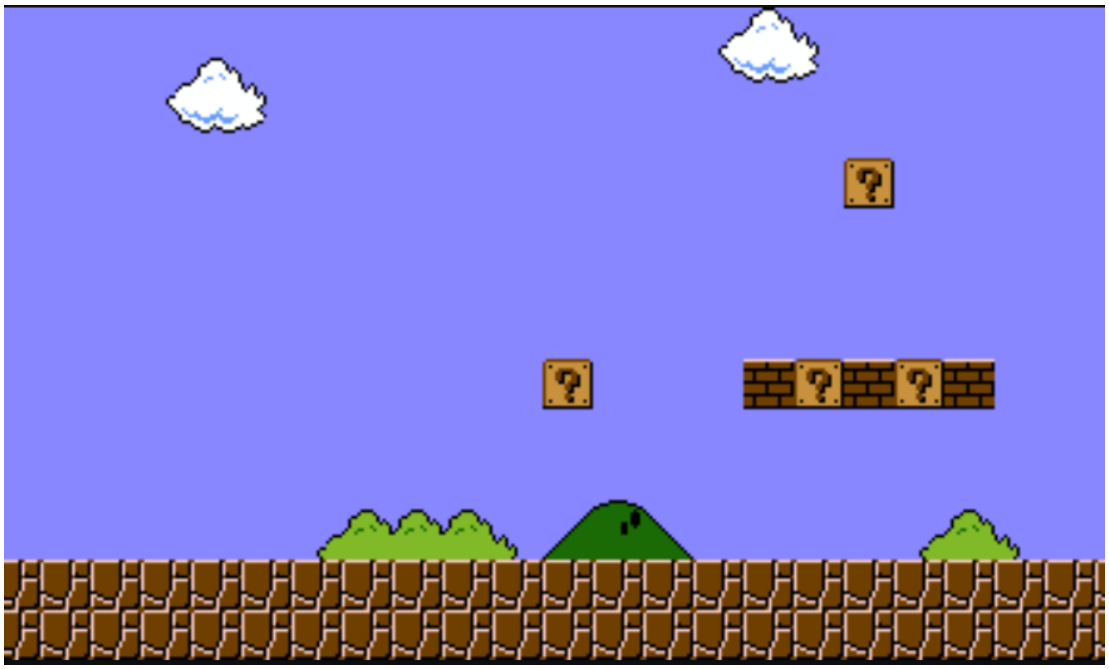


Figura 2.5: Super Mario Bros. ejemplo de multiples usos del mismo tile (Earl, 2018)

Como mencionamos anteriormente estos tiles vienen alojados en una ROM en forma de mapa de tiles a la que llamaremos también Pattern Table, cada tile tendrá un index que los identifica dentro del mapa de tiles. Estos tiles son utilizados en la *Name Table*, una Name Table es básicamente una tabla (una matriz en memoria) que describe qué tile se debe mostrar en cada posición de la pantalla. Cada entrada en la Name Table apunta a un índice dentro de la Pattern Table. Es decir, la pantalla se divide en una cuadrícula de celdas del tamaño de un tile y cada entrada en la Name Table indica qué tile (número o índice) debe mostrarse en esa posición. En la Figura 2.6 podemos ver un ejemplo gráfico de una Name Table.

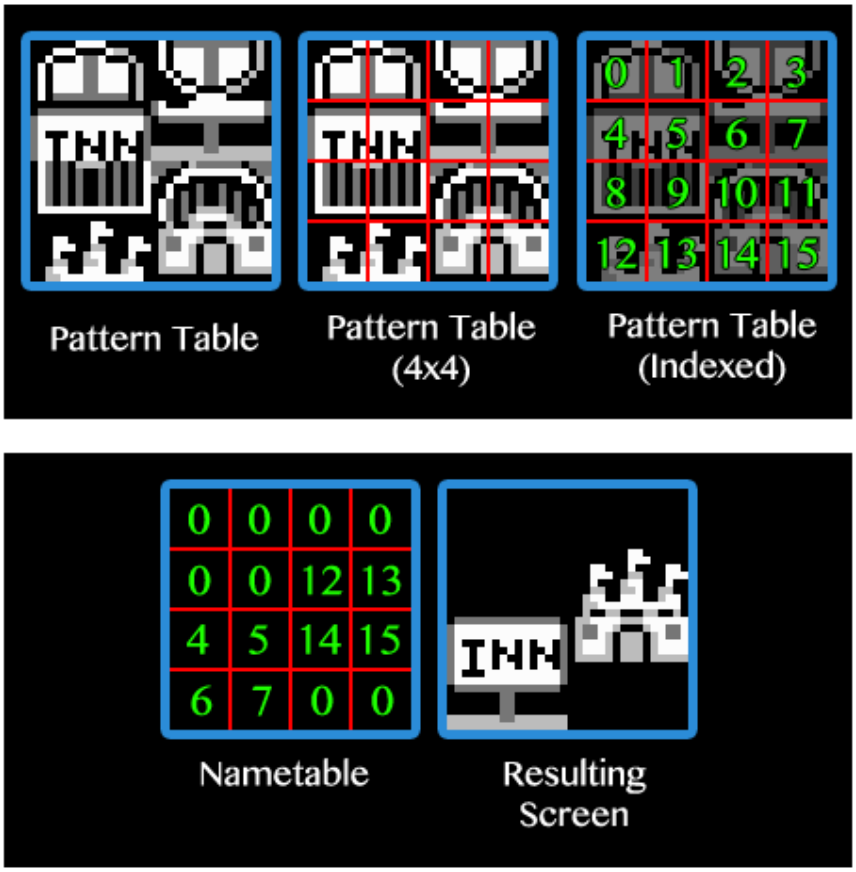


Figura 2.6: Ejemplo de una *Name Table*. (Parkes, 2013)

Los sistemas arcade utilizaron motores muy potentes con sprites muy grandes, mucha memoria y varias tablas de patrones, mientras que las consolas de videojuegos implementaron motores más simples y utilizaron menos memoria de video. Sin embargo, todos estos motores compartían los mismos principios básicos.

2.2.3. Emulación de Memoria

La memoria es uno de los componentes centrales de cualquier sistema computacional. En un hardware real, el espacio de direccionamiento de memoria no es simplemente una región plana y accesible de manera uniforme, está estructurada en diferentes regiones con propósitos específicos. Estas pueden incluir memoria RAM principal, memoria de video, registros de dispositivos, acceso a memorias

ROM, entre otros. En un emulador, replicar esta estructura es clave para lograr una emulación funcional. En cuanto a espacio de direccionamiento de memoria se pueden diferenciar, a grandes rasgos, dos tipos de sistemas:

- **Consolas antiguas basadas en cartuchos (ROM)**: Las consolas antiguas que usan cartuchos ROM organizan su espacio de memoria en dos regiones principales: una para la ROM, donde están almacenados el código y los datos del juego, y otra para la RAM, que se usa para manejar variables y datos temporales. La ROM es de solo lectura y está protegida para evitar problemas causados por errores en el código o sistemas antipiratería, mientras que la RAM se puede usar libremente. Además, estas consolas a menudo tienen regiones de memoria que son "espejos" de otras, debido a limitaciones en el hardware, y áreas que no se utilizan pero que deben responder con valores predeterminados al ser leídas.
- **Computadoras y consolas modernas basadas en CD**: Las computadoras y consolas modernas, como las que usan unidades ópticas de almacenamiento (como CDs o BlueRays), tienen un diseño de memoria más similar a una computadora estándar. Su ROM suele contener el BIOS o el sistema operativo interno, mientras que la RAM, mucho más amplia, sirve para cargar los datos y el código desde medios como CD-ROMs.

El mapeo de memoria de un sistema define cómo se organiza el espacio de direcciones entre los distintos componentes, por ejemplo en la Figura 2.7 podemos ver el mapeo de memoria de la GameBoy.

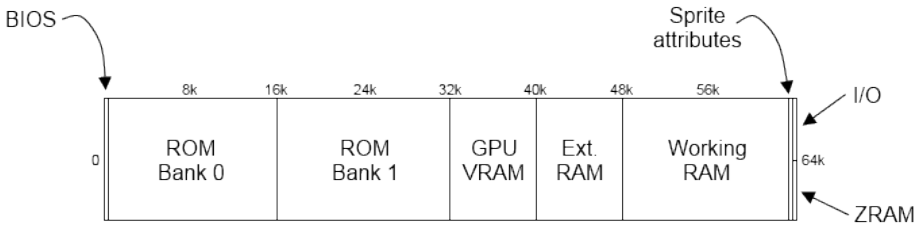


Figura 2.7: Mapeo de memoria de GameBoy (Nazar, 2010)

El espacio de direcciones de memoria en una computadora puede utilizarse tanto para acceder a memoria (ya sea memoria del sistema o de un dispositivo, como la memoria de video) como para acceder a dispositivos I/O. Este tipo de acceso se conoce como I/O mapeado en memoria. En este sistema, los registros de los dispositivos están directamente asignados en el espacio de direcciones de memoria, gracias a decodificadores que utilizan algunas o todas las líneas de dirección del bus para determinar qué dispositivo o chip de memoria debe ser accedido. En algunos sistemas (como los basados en arquitecturas como el Z80 o x86), existe un espacio de direcciones separado específicamente para el acceso I/O. Cada dirección en este espacio está vinculada a un registro en un

dispositivo, y este espacio de direcciones suele ser más pequeño (por ejemplo, de solo 8 bits).(del Barrio, 2001)
En el Listing 2.2 se muestra a modo de ejemplo como emular un espacio de direccionamiento genérico de 16 bits.

Listing 2.2: Emulacion de memoria

```
// Crear el espacio de direccionamiento
UINT8 memoria[2^16] // Espacio de direccionamiento total de 16 bits

// Definir mapeo de memoria
ROM_START = 0x0000   // Region de ROM desde 0x0000 hasta 0x7FFF (32KB)
ROM_END = 0x7FFF
RAM_START = 0x8000   // Region de RAM desde 0x8000 hasta 0xBFFF (16KB)
RAM_END = 0xBFFF
IO_START = 0xC000   // Region de IO desde 0xC000 hasta 0xCFFF (4KB)
IO_END = 0xCFFF

LEER_MEMORIA(direccion):
    IF direccion >= ROM_START AND direccion <= ROM_END:
        retornar memoria[direccion]   // ROM (solo lectura)
    IF direccion >= RAM_START AND direccion <= RAM_END:
        retornar memoria[direccion]   // RAM (lectura/escritura)
    IF direccion >= IO_START AND direccion <= IO_END:
        retornar memoria[direccion]   // IO (lectura/escritura)
    ELSE:
        imprimir "Error: Direccion fuera de rango"

ESCRIBIR_MEMORIA(direccion, valor):
    IF direccion >= ROM_START AND direccion <= ROM_END:
        imprimir "Error: No se puede escribir en ROM"
    IF direccion >= RAM_START AND direccion <= RAM_END:
        memoria[direccion] = valor   // RAM (lectura/escritura)
    IF direccion >= IO_START AND direccion <= IO_END:
        memoria[direccion] = valor   // IO (lectura/escritura)
    ELSE:
        imprimir "Error: Direccion fuera de rango"
```

2.2.4. Emulación de Interrupciones y Sincronización Entre Componentes

Las interrupciones permiten que los dispositivos envíen señales a la CPU para interrumpir su ejecución y realizar tareas específicas. Las interrupciones pueden tener diferentes prioridades, y las de mayor prioridad, como la NMI (Non Maskable Interrupt), no pueden ser interrumpidas ni deshabilitadas.

En los emuladores de CPU, se implementa una lógica para manejar estas interrupciones, emulando la respuesta de la CPU a las señales de interrupción de dispositivos como teclado, disco, y hardware gráfico o de sonido. El emulador debe ser capaz de almacenar el contexto de la CPU (como el valor del contador de programa y los registros) cuando se recibe una interrupción, y luego "salta" a la dirección donde se encuentra la rutina de la interrupción correspondiente.

En sistemas de videojuegos como consolas o arcades, los dispositivos gráficos y de sonido son los principales generadores de interrupciones. Por ejemplo, las interrupciones de sincronización vertical y horizontal en el hardware gráfico permiten al emulador manejar el tiempo de la pantalla y actualizar la memoria gráfica sin causar distorsiones. El emulador debe manejar con precisión estas interrupciones, ya que afectan directamente a la sincronización visual y auditiva del juego.

En la mayoría de los sistemas de videojuegos, las interrupciones gráficas se utilizan como el principal mecanismo de sincronización del juego. La interrupción de Vertical Blank (VBlank) indica cuándo comienza y termina un frame. Entre el final de un cuadro y el comienzo del siguiente, se puede acceder a la memoria gráfica sin causar corrupción en la pantalla, por lo que es en este momento cuando el código del juego actualiza la información de video para el próximo cuadro.

En la mayoría de estos sistemas, la interrupción gráfica es la única forma confiable de sincronización, además de un conteo preciso de los ciclos de la CPU ejecutados. Este comportamiento de los sistemas de videojuegos tiene un impacto profundo en la estructura del emulador. Dado que la interrupción de video es el principal evento periódico en el emulador, la emulación se organiza en torno a ella. En el Listing 2.3 podemos ver como se estructura el ciclo de emulación en torno a interrupciones gráficas. Este es el algoritmo general, que se modifica en algunos sistemas debido a la existencia de otros eventos, como una emulación de sonido más precisa o una interrupción horizontal, que implica una emulación línea por línea del frame. (del Barrio, 2001)

Listing 2.3: Ciclo de emulación basado en interrupciones gráficas

```
while true:
    // Ejecuta ciclos de la CPU para el frame.
    emular_ciclos_cpu()

    // Ejecuta ciclos del hardware grafico para el frame.
    emular_ciclos_de_unidad_grafica()

    // Renderiza el frame en pantalla
    renderizar()

    // Procesar audio del cuadro
    procesar_audio()
```

```
// Genera la interrupcion grafica.
generar_interrupcion_vblank()

// Espera los milisegundos correspondientes para
//que no funcione m s rapido que la maquina original.
esperar_tiempo()
```

2.2.5. Búsqueda de Documentación y las Dificultades Presentes

Una de las tareas más importantes en el proceso de emular un sistema cualquiera es recopilar la mayor cantidad de información posible sobre cómo funciona ese sistema. Esto no es una tarea fácil ya que la gran mayoría de las veces no existe una documentación oficial o completa. Normalmente esta documentación es de uso privado y se mantiene en secreto por la empresa que diseñó el sistema. Esto suele ser la norma para consolas de videojuegos y máquinas arcade ya que son sistemas cerrados con estrictas reglas de protección para los desarrolladores de juegos que obtienen kits oficiales de desarrollo. En algunos casos, solo la empresa que diseñó la máquina produce programas para ella.

Aunque la tarea es difícil, hay algunas consideraciones iniciales que nos pueden ayudar. Cuanto más popular y antigua sea una máquina, más fácil será encontrar información. Un sistema antiguo implica que muchas personas ya habrán trabajado con él e investigando sus características ocultas. Incluso puede que las reglas de protección ya no estén vigentes o a la empresa ya no le importe. La documentación oficial podría haberse "filtrado" y estar disponible gratuitamente. Lo mismo ocurre con los sistemas populares, aunque en un período más corto de tiempo. Un sistema popular es aquel en el que muchas personas están investigando como funciona y desarrollando juegos y programas. Esto ayuda a que se genere más documentación aunque no sea oficial. Una de las mejores fuentes de información para sistemas nuevos, con poca documentación, siempre han sido los grupos de "demos". Una demo es un pequeño programa con gráficos y sonido que intenta aprovechar las características de una máquina para causar impacto en el usuario o llevar las capacidades de la máquina al límite. Antes de que las personas intenten crear sus propios juegos caseros para una plataforma, los programadores de demos suelen trabajar con ella. El último recurso es utilizar la ingeniería inversa para descubrir cómo funciona el hardware de la máquina emulada. Esta es una tarea difícil y requiere mucha experiencia y conocimiento sobre hardware. El código de los programas para la máquina emulada se analiza para intentar descubrir los comandos enviados al hardware y determinar qué hacen dichos comandos. Esto es más sencillo si se tiene acceso a la máquina original y a herramientas para desarrollar programas para ella. Obtener información del código original y usarla con el hardware original es una buena manera de recopilar datos. El trabajo de ingeniería inversa a menudo se realiza también con el propio emulador, integrando un debugger y un desensamblador. Los errores del emulador y las diferencias entre la salida del

emulador y la de la máquina real se rastrean hasta encontrar las configuraciones correctas. Es una tarea muy demandante que requiere mucho tiempo y paciencia. La ingeniería inversa es el único método legal para emular una máquina cuyos derechos de protección abarcan todo su hardware y documentación. (del Barrio, 2001)

2.2.6. Testeo y Depuración de un Emulador

La tarea de testear y debugguear un emulador es igual o incluso más difícil que recopilar la información necesaria para hacer el emulador. En algunos casos, cuando es necesario aplicar ingeniería inversa, ambas tareas se realizan simultáneamente. Las diferentes partes del emulador deben ser testeadas y depuradas, pero cada una requiere esfuerzos específicos en este proceso. No es lo mismo probar la CPU, que ejecuta decenas o cientos de miles de instrucciones por segundo, que el hardware gráfico o de sonido. Se deben realizar pruebas de integración y pruebas separadas. Por lo general, se prueba la CPU de manera aislada (usando pequeños programas en el lenguaje ensamblador emulado), mientras que las otras partes se desarrollan, prueban y depuran como parte del emulador completo. La CPU es probablemente donde más errores pueden ocultarse debido a la cantidad de código necesario para emular todas sus instrucciones. Es más rápido encontrar errores simples (porque el juego o programa simplemente no funciona) y más difícil detectar errores pequeños debido a instrucciones o entradas que rara vez se usan. Una ventaja de la CPU es que se puede construir un sistema de pruebas para realizar evaluaciones sistemáticas de todas las instrucciones. Hay dos formas básicas de hacerlo:

1. Usar una herramienta que genere pequeños programas para probar unas pocas instrucciones con parámetros aleatorios, comparando los resultados del núcleo de la CPU emulada con los de una CPU real, si se tiene acceso a una. Este es el mejor enfoque, ya que se compara directamente con el hardware real. Sin embargo, algunas CPU son difíciles de encontrar o utilizar, y crear una placa dedicada para las pruebas puede ser costoso.

2. Una alternativa más accesible es comparar nuestra implementación del CPU con otra ya implementada y confiable. Aunque este enfoque es útil, no es perfecto, ya que las diferencias de ambas implementaciones pueden generar discrepancias.

El hardware gráfico y de sonido puede ser más o menos difícil de probar. Los errores grandes son fáciles de detectar porque se perciben visual o auditivamente. Comparar colores, imágenes mostradas, música y sonidos con los de la máquina original ayuda a identificar problemas rápidamente, aunque encontrar la causa del error puede ser complicado. Un modelo preciso del sistema gráfico y de sonido emulado es casi imposible sin conocer las características exactas del hardware. Sin herramientas especializadas como osciloscopios o comparadores de imágenes, es difícil detectar pequeñas diferencias en sonido y gráficos. Otros problemas, como la velocidad percibida de un programa o juego, pueden ser más sutiles y

suelen estar relacionados con una emulación inexacta de las frecuencias de los componentes, lo cual es difícil de resolver.

El tema de la ingeniería inversa, las pruebas y la depuración es vasto y complejo, al punto de requerir cientos de páginas para abordarlo en detalle. Abordar los desafíos asociados con estas actividades exige experiencia, profundo conocimiento sobre electrónica, práctica constante y un enfoque metódico. La perfección en los emuladores es prácticamente inalcanzable, ya que lograr que todos los juegos o programas de una máquina funcionen correctamente resulta extremadamente difícil, principalmente por la imposibilidad de probarlos todos en su totalidad.

La tarea se complica aún más cuando ciertos cartuchos ROM incorporan hardware adicional que no está bien documentado, o ciertas placas arcades, que incluían sistemas de autodestrucción al detectar variación de voltaje o por incluso tocar la parte inferior de la placa mientras esta prendida como ocurre con la popular *Capcom Play System III* de Capcom (ArcadeOtakuWiki, 2023). En este contexto, el proceso de depuración se asemeja a un esfuerzo interminable, al igual que ocurre con otros productos de software. Solo aquellos sistemas que poseen características limitadas, documentación extensa y un catálogo pequeño de programas completamente probados pueden considerarse verdaderamente emulados. (del Barrio, 2001)

2.3. Técnicas Modernas de Emulación

Hasta ahora hemos mencionado los métodos tradicionales de emulación donde se emula el comportamiento de los componentes de hardware que forman un sistema en particular, es decir una aproximación LLE que funciona muy bien para sistemas clásicos debido a que eran más simples y limitados. Pero el crecimiento exponencial en el poder de cómputo de los sistemas modernos y su creciente complejidad han llevado a la necesidad de abstraernos del hardware para lograr una emulación más fluida. Es aquí donde nace el paradigma HLE (High-Level Emulation).

2.3.1. HLE

En el paradigma HLE la barrera se establece entre el núcleo del sistema operativo (kernel) y las aplicaciones que corren sobre él (como juegos, programas, etc.). Las aplicaciones funcionan normalmente (después de que se traducen las instrucciones específicas del sistema de usuario), pero cuando necesitan interactuar con el sistema operativo (como abrir archivos, mapear memoria o crear procesos), esas solicitudes (conocidas como *syscalls*) son interceptadas y gestionadas por el código creado por los desarrolladores del emulador. El primer emulador conocido en usar esta técnica fue el **UltraHLE** para emular el sistema Nintendo 64 en el año 1998, que era la consola más potente del momento. Lo que hacia UltraHLE era tomar los llamados a la N64 3D API y traducirlos a la API gráfica que usara la maquina destino, esto aceleraba enormemente la

velocidad de emulación permitiendo correr (aunque con errores gráficos) estos complejos juegos 3D en las Pentium 2 de la época si se contaba con una tarjeta aceleradora 3D. (del Barrio, 2001) A medida que los sistemas de consolas avanzan en complejidad, la importancia del enfoque HLE (High-Level Emulation) aumenta. Las consolas modernas ya son demasiado complejas y poderosas para implementar su emulación utilizando el enfoque tradicional. Además, algunos sistemas (como la Xbox 360) no son más que un sistema operativo estándar de PC, lo que hace que sea ineficiente intentar recrear el hardware utilizando una PC como máquina anfitriona. Por lo tanto, el enfoque HLE se convierte cada vez más en la única opción razonable. El estado de los PCs de consumo también ha cambiado. Las computadoras actuales son mucho más rápidas que las de hace 20 años, y la emulación LLE ya es posible para algunas de las primeras consolas y CPUs que en los años 90 tuvieron que ser emuladas mediante HLE. Como resultado, muchos emuladores ahora pueden optar por la precisión y la replicación exacta de los ciclos de los componentes, lo que da lugar a entornos de software muy precisos que finalmente pueden reemplazar a las consolas y computadoras antiguas. Sin embargo, HLE ha encontrado un nuevo propósito en los teléfonos inteligentes, dispositivos portátiles y otros dispositivos electrónicos que tienen especificaciones mucho más bajas que la computadora promedio, y para estos dispositivos el paradigma HLE se traduce en framerates más altos. (EmulationGeneralWiki, 2024)

Ventajas y Desventajas de HLE

La técnica de HLE ofrece varias ventajas, entre las cuales destaca la capacidad de aprovechar de manera más eficiente los recursos del sistema anfitrión. Además, permite optimizar los resultados a medida que el hardware y el software mejoran, y reduce el esfuerzo necesario para lograr el resultado deseado, siempre que la funcionalidad adecuada ya esté disponible en el sistema anfitrión, como ocurre con las funciones gráficas 3D. Otra ventaja es que la implementación de HLE no depende tanto de la documentación detallada del hardware emulado, sino que solo requiere acceder a las funciones que ofrece el kit de desarrollo del sistema anfitrión. Además, HLE elimina algunos problemas legales comunes, ya que no requiere que los usuarios proporcionen el software de arranque o los componentes del sistema original, como ROMs o imágenes de cartuchos, que normalmente podrían generar dudas legales relacionadas con los derechos de autor. Sin embargo, HLE tiene desventajas, como una mayor dependencia de la estandarización entre las aplicaciones emuladas y de los mecanismos de alto nivel presentes en la plataforma original. Si no existen estos mecanismos o las aplicaciones no los utilizan correctamente, pueden fallar al ejecutarse, aunque otras aplicaciones similares funcionen bien. Esto significa que a menudo se requieren ajustes adicionales para garantizar que todos los juegos y aplicaciones funcionen correctamente.

Aunque HLE es más fácil de implementar y puede ofrecer un rendimiento más rápido incluso en hardware limitado, lo hace sacrificando la precisión. El enfoque HLE no puede lograr el mismo nivel de precisión que una emulación de

bajo nivel (LLE), ya que se basa en emular los resultados deseados, en lugar de replicar con exactitud los tiempos y comportamientos del hardware original. En algunos casos, un juego puede ejecutarse correctamente al 90 % de su precisión original, pero en otros puede no funcionar en absoluto debido a que depende de una sincronización o función muy precisa. En contraste, en LLE, la emulación se realiza replicando los chips de hardware originales, incluyendo sus errores y tiempos de espera, lo que asegura una mayor precisión y menor necesidad de ajustes o hacks específicos por juego.

Por lo tanto, mientras que HLE suele ser una opción más rápida y fácil, la emulación de bajo nivel (LLE) suele ser más precisa y fiable, especialmente cuando se busca mantener la compatibilidad y la precisión de juegos que dependen de una emulación exacta del hardware original. (EmulationGeneralWiki, 2024)

2.3.2. Otras Consideraciones

LLE y HLE no son los únicos paradigmas existentes en emulación ni es una obligación elegir entre uno y otro, de hecho la aproximación híbrida ha ganado una popularidad significativa en los emuladores de software recientes para sistemas más nuevos. Esta popularidad se debe a su capacidad para combinar ambas técnicas: Emulación LLE para los servicios críticos del sistema operativo y Emulación HLE para componentes complejos o de alta demanda. [1] Esta combinación ofrece varios beneficios:

- **Reducción del esfuerzo de desarrollo**: Al centrar el enfoque LLE en los aspectos esenciales, los desarrolladores pueden simplificar el proceso.
- **Mejora del rendimiento**: HLE simplifica la emulación de componentes no críticos, lo que resulta en una ejecución más rápida.
- **Mantenimiento de la precisión**: LLE asegura una emulación precisa de las funcionalidades principales.

Los emuladores más conocidos que usan este enfoque híbrido son **Dolphin** que emula Nintendo GameCube/Wii y **RPCS3** que emula Playstation 3. (Sanchez, 2018)

Como dijimos hace anteriormente, LLE y HLE no son los únicos paradigmas existentes. Si tomamos a LLE como pararse entre el kernel y el hardware y a HLE como pararse entre el kernel y la aplicaciones, estos no son los únicos puntos donde un emulador puede ponerse. El emulador podría pararse en un nivel aún más alto entre el juego y el motor del juego, esta idea no es tan descabellada, ya que es precisamente lo que hace **ScummVM**(scummvm.org, 2024), emulador que nos permite jugar clásicas aventuras gráficas como *Monkey Island* o *Day of Tentacle*. Sin embargo, los motores de videojuegos actuales son increíblemente complejos, con varios millones de líneas de código, lo que nos llevaría siglos como desarrollador escribir un emulador que funcionara a un nivel tan alto. El tiempo de desarrollo sería enorme, pero el rendimiento del

emulador sería bastante bueno, ya que todas las tareas complejas se habrían re-implementado de manera nativa para el sistema anfitrión. El emulador también podría pararse por debajo del punto LLE a un nivel de transistores y tampoco sería tan descabellado para plataformas antiguas, como lo demuestra el proyecto **Visual 6502** (visual6502.org, 2012). Suponiendo que estamos equipados con todo los instrumentos necesarios podríamos generar fácilmente un código que simule nuestro CPU objetivo, lo que implicaría un tiempo de desarrollo muy reducido. Sin embargo, el rendimiento sería extremadamente pobre debido a que tenemos que emular miles de millones de transistores. (Sanchez, 2018)

En conclusión, no siempre subir de nivel disminuye la cantidad de trabajo perdiendo precisión, ni siempre bajar de nivel aumenta la cantidad de trabajo. A la hora de encarar un proyecto de emulación y definir un paradigma, todo debe ser considerado caso a caso dependiendo que sistema se quiera emular y en que sistema se quiera correr el emulador.

Capítulo 3

Nintendo Entertainment System

En este capítulo profundizamos sobre la NES que fue el sistema que emulamos. Presentamos una breve historia de la consola y una descripción detallada de los componentes principales de su arquitectura, incluyendo la CPU, PPU, manejo de inputs del gamepad y el Game Pak.

3.1. Un poco de historia

En plena era del entretenimiento en casa, cuesta imaginar que hace unas pocas decadas era impensado tener un sistema de videojuegos propia en el living. Solo unos pocos afortunados —muy pocos y muy privilegiados— podían darse el lujo de tener una máquina arcade para ellos solos. El resto tenía que conformarse con ir a los salones de maquinas arcade, esos lugares medio oscuros, llenos de humo y con personajes pintorescos que era mejor tenerlos lejos que cerca. Aunque desde los años 70 ya existían consolas hogareñas que generaban un negocio bastante rentable, la variedad y calidad de los juegos no tenían punto de comparación con lo que ofrecían las arcade. De hecho, la decepción que causaron algunos títulos, como la desafortunadamente célebre adaptación de **E.T.** de **Atari**, sumada a la saturación del mercado de dispositivos y juegos de baja calidad, terminó haciendo que, a principios de los 80, todo el mercado de videojuegos hogareños se viniera abajo. Entre 1982 y 1983, las ventas en Estados Unidos pasaron de tres mil millones de dólares a apenas cien millones en cuestión de meses. (Kent, 2016)

El golpe fue tan grande que muchas empresas fabricantes terminaron quebrando, dejando a la industria en ruinas. Invertir en videojuegos se había convertido en algo impensable y nadie apostaba por su recuperación. Pero, mientras en ese lado del mundo todo parecía perdido, en Japón estaba por salir a la luz un sistema de entretenimiento doméstico que salvaría a la industria de los videojuegos y los llevaría al éxito que conocemos hoy.

Figura 3.1: NES Action Set. Fuente: nintendo.fandom.com

El 15 de julio de 1983 se lanza en Japon la **Famicom**, que llegaría luego a Estados Unidos bajo el nombre de **Nintendo Entertainment System** el 18 de octubre de 1985 con un re-diseño curioso ya que se asemejaba a un reproductor de VHS, incluso los juegos se insertaban de manera similar, esto era debido a que Nintendo intentaba venderla no tanto como una consola de videojuegos sino como una suerte de juguete usando un televisor, para reforzar más esta suerte de caballo de troya, Nintendo incluía una pistola de luz conocida como Zapper que puede verse en la Figura 3.1 y hasta un robot llamado R.O.B. que reaccionaba a los juegos mirando la pantalla de la TV. En la Figura 3.2 se puede ver a R.O.B. y los accesorios que utiliza para interactuar con el juego *Gyromite*.

Figura 3.2: R.O.B. Fuente: nes.fandom.com

Al principio solo se vendía en Nueva York, donde Nintendo distribuyó cien mil unidades de cara a las fiestas navideñas para tantear el interés de los usua-

rios. Aunque el éxito fue considerable, Nintendo prefirió mantener la cautela y seguir presentando su consola poco a poco. A principios de 1986, la NES se lanzó en Los Ángeles, posteriormente en Chicago, luego en San Francisco, y no sería hasta septiembre de 1986 cuando la consola de Nintendo se pudo comprar en cualquier lugar de Estados Unidos y resultó en un éxito rotundo sin rival .

En 1987, la NES se convirtió en el juguete más vendido de Estados Unidos, mientras que *The Legend of Zelda* se convirtió en el primer juego de NES en alcanzar un millón de unidades vendidas. Solo en Estados Unidos, los ingresos de *Super Mario Bros. 3* superaron los 500 millones de dólares, con más de 7 millones de copias vendidas y 4 millones adicionales en Japón. En 1991, Nintendo generaba alrededor de 1,5 millones de dólares por cada uno de sus 5,000 empleados. Las ganancias de la compañía a principios de los años 90 superaron las de la industria cinematográfica estadounidense. Tal fue el impacto de Nintendo en la cultura estadounidense que una encuesta de 1990 reveló que Mario era más reconocido por los niños que el propio Mickey Mouse. (Diskin, 2004)
La NES, fue descontinuada en Occidente en 1995. En cambio, la división japonesa de la marca aún la mantendría comercialmente viva hasta principios de los 2000. (Bravo, 2023)

3.2. Arquitectura

A la hora de diseñar la NES, **Hiroshi Yamauchi**, presidente de Nintendo en aquella época, quería una consola más económica que la competencia lo que llevo a Nintendo a decidir usar una CPU ya bastante obsoleta para esos años. Aunque un procesador de 16 bits habría sido mejor eleccion tecnologicamente hablando, para mantener el costo bajo, decidieron utilizar la variante del procesador de 8 bits MOS 6502-2A03, desarrollado por MOS Technology en 1975. El chip era suficiente para ejecutar los programas, pero no podía generar los gráficos requeridos, por lo que la compañía decidió diseñar un segundo chip llamado PPU, responsable de calcular y mostrar los gráficos. Sin embargo tuvo dificultades para encontrar una empresa dispuesta a producir chips tan personalizados a un precio tan bajo. Ricoh accedió a fabricarlos después de que Nintendo garantizara un pedido de tres millones de chips.

Ambos chips cuentan con su propia memoria RAM. Los juegos solían almacenarse en chips ROM dentro de los cartuchos de juego, los cuales podían ser accedidos por la CPU y el PPU al insertarse en el sistema.

La comunicación del procesador a los dispositivos I/O como la PPU y los dispositivos de entrada es través de mapeo en memoria, los datos se transfieren desde el CPU a un dispositivo mediante una escritura en una ubicación específica de la memoria y viceversa mediante una lectura a una ubicación específica de la memoria.

Exponer la totalidad de la arquitectura de la NES en este documento excede el alcance y el propósito del mismo por lo que nos vamos a enfocar en los componentes fundamentales y necesarios que uno debe conocer para poder hacer

un emulador funcional, estos son la CPU, la PPU, el Game Pak y el manejo de inputs.

3.2.1. CPU

El 2A03 es un circuito integrado personalizado que funciona como el corazón de la NES y la Family Computer. Para evitar costosos circuitos de lógica adhesiva (*Glue logic definition*, s.f.), Nintendo incorporó una gran cantidad de hardware (mucho para la época, que era alrededor de 1982) dentro de este chip. (Taylor, 2004) Este chip se diferenciaba de un 6502 estándar en que tenía la capacidad de manejar sonido, funcionando como una pAPU (pseudo Unidad de Procesamiento de Audio) además de CPU, y carecía del modo BCD (numeros decimales en binario). Para efectos de programación, el 2A03 es un procesador little endian y utiliza el mismo conjunto de instrucciones que el 6502 estándar.

Mapeado de memoria

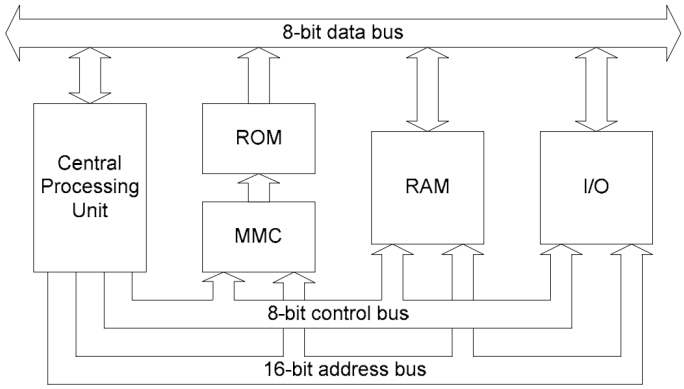


Figura 3.3: Diagrama de CPU

El mapeado de memoria se puede dividir en tres partes, el acceso a la memoria ROM en el cartucho (notar que el CPU solo accede al ROM por medio del MMC o mapper), acceso a la memoria RAM y acceso a los dispositivos de I/O. El bus de datos se usa para leer o escribir un byte en la dirección seleccionada, el de control para informar a los componentes si la operación es de lectura o escritura y el bus de direcciones se utiliza para establecer la dirección de la ubicación requerida. Como se puede ver en la Figura 3.3 el bus de direcciones es de 16 bits y direcciona dentro del rango $0000-$FFFF. Estos 64 KBs no están completos de memoria física En la Figura 3.4 podemos ver el layout:

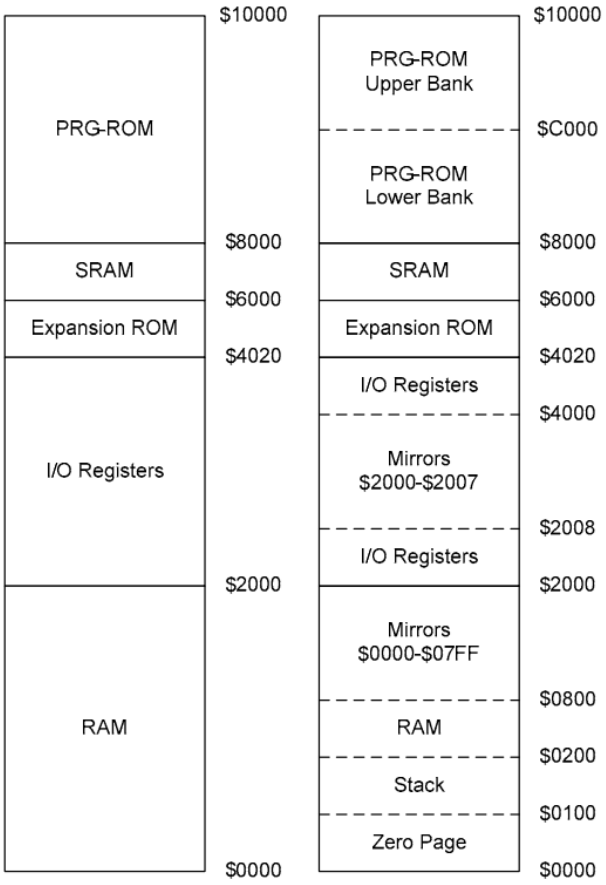


Figura 3.4: Mapeo de memoria del CPU

- **$0000 - $1FFF**: Este rango pertenece a la memoria RAM que es 2KB, incluyendo dentro el stack y el Zero Page que es un rango especial de $0000-$00FF utilizada por algunos modos de direccionamiento de la NES para acceder en menos ciclos y con menos bytes de dirección necesarios. Luego están los "mirrors", estos se indican que la memoria RAM está espejada, esto quiere decir que el bloque real de memoria RAM $0000-$07FF se vuelve a repetir en $0800 - $1FFF , y acceder a la direccion $0800 seria lo mismo que acceder a $0000. Estos mirrors están presentes para completar el espacio de direccionamiento y simplificar el diseño.
- **$2000**–**$2007** : Registros de PPU, de estos hablaremos más adelante.
- **$2008**–**$3FFF**: De nuevo mirrors, donde se espejan los registros de PPU.
- **$4000**–**$401F**:Registros de I/O, vamos a nombrar los que más nos in-

teresan a la hora de emular, $4000–$4013 y el $4015 son registros para controlar distintos aspectos de la APU, el registro $4014 activa el OAM DMA (hablaremos más adelante de esto ya que tiene que ver con los sprites), y los registros $4016 y $4017 son para leer los inputs del joystick más adelante detallaremos cómo estos son leídos.

- **$4020–$FFFF**: Normalmente estos espacios no están mapeados directamente por el procesador sino que son accedidos a través del MMC (Memory management controller) también conocido como Mapper que esta alojado dentro del cartucho por lo que ahondaremos más este rango cuando hablemos de los cartuchos también llamados Game Pak. (Diskin, 2004)

Registros

Los registros presentes en el 6502 son los siguientes:

Program Counter (PC): Registro de 16 bits que apunta a la siguiente instrucción a ejecutar. En cada instrucción ejecutada el valor de este registro es afectado, también es afectado por operaciones de jump, branch(saltos condicionales por ejemplo al valor de una flag) e interrupciones.

Stack Pointer (SP): Registro de 8 bits que es un puntero a la pila. La pila se encuentra en las ubicaciones de memoria $0100-$01FF. El puntero de pila actúa como un desplazamiento desde $0100. La pila funciona de arriba hacia abajo, por lo que se inicializa en el valor $FF y cuando se aloja un byte, el puntero de pila se decrementa, y cuando se extrae un byte de la pila, el puntero de pila se incrementa. No hay detección de desbordamiento de pila, por lo que el puntero de pila simplemente volverá a comenzar desde $00 hasta $FF.

Accumulator (A): Registro de 8 bits que almacena los resultados de operaciones aritméticas y lógicas.

Index Register (X): Registro de 8 bits, típicamente usado como contador o desplazamiento en ciertos modos de direccionamiento. Puede usarse también para almacenar un valor recuperado de la memoria y obtener o establecer el valor del puntero de pila.

Index Register Y (Y): Registro de 8 bits que se utiliza de manera similar al registro X, como contador o para almacenar un desplazamiento en ciertos modos de direccionamiento pero a diferencia del registro X, no puede afectar el puntero a la pila.

Status Register (P): Registro de 8 bits que contiene flags de un solo bit que se activan o desactivan al ejecutar instrucciones. Las flags son las siguientes:

- **Carry Flag (C)**: Se activa si la última instrucción produjo un desbordamiento por ejemplo al realizar 255+1 da como resultado 0 y activa el

flag de carry. Esto permite realizar cálculos en números de más de 8 bits, usando el bit de carry en el siguiente cálculo.

- **Zero Flag (Z)**: Se activa si el resultado de la última instrucción fue cero.
- **Interrupt Disable (I)**: Este flag se usa para evitar que el sistema responda a interrupciones IRQ.
- **Decimal Mode (D)**: Dado que no hay modo decimal en este modelo esta flag es ignorada.
- **Break Command (B)**: Indica que se ejecutó una instrucción `BRK(break)`, lo que genera una interrupción IRQ.
- **Overflow Flag (V)**: Se activa si la última instrucción produjo un resultado no válido. Esto ocurre cuando por ejemplo, sumar dos números positivos da un numero negativo, por ejemplo haciendo la suma 64 + 64 da como resultado −128, activando el flag.
- **Negative Flag (N)**: El bit más significativo de un byte representa su signo, donde 0 es positivo y 1 es negativo. Esta flag se activa si ese bit es 1.

Modos de direccionamiento

El 6502 tiene un total de 13 modos de direccionamiento.

- **Implied**: Este modo de direccionamiento no accede a memoria, se usa por ejemplo en instrucciones que desactivan una flag del registro de status , o guardan un registro en la pila.
- **Accumulator**: Este modo de direccionamiento indica que la instrucción se hará en el registro acumulador, es como el direccionamiento implícito donde la dirección implícita es siempre la misma.
- **Immediate**: La instrucción usará como operando el byte que se encuentra la siguiente posición de memoria del opcode (PC + 1).
- **Zero Page**: Este modo de direccionamiento indica que la dirección de memoria tendrá la forma $00XX por lo que solo necesitamos un byte para tener la dirección final, este byte se obtiene leyendo la siguiente posición de memoria del opcode (PC + 1) .
- **Zero Page, X**: Igual que Zero Page pero a $00XX se le suma el valor contenido en el registro X.
- **Zero Page, Y**: Igual que Zero Page X, pero en lugar de sumar el registro X se suma el valor contenido en el registro Y.

- **Relative**: Este tipo de direccionamiento se usa para saltos condicionales (branching), a la siguiente posición de memoria del opcode (PC + 1) se provee un byte a modo de offset que será agregado al PC actual si se cumple la condición del branch, este offset debe estar en el rango -128 a 127.

- **Absolute** : En este tipo de direccionamiento se provee los dos bytes para formar la dirección de 16 bits, en PC +1 se provee la parte baja de la dirección y en PC + 2 la parte alta, por ejemplo si en PC+1 tengo 00 y en PC + 2 tengo FF se formaría la dirección $FF00.

- **Absolute X** : Igual que absolute pero se le suma el valor contenido en el registro X.

- **Absolute Y** : Igual que Absolute X pero sumando el valor del registro Y.

- **Indirect** : Este modo direccionamiento solo se utiliza en instrucciones de salto. Los dos bytes proporcionados después del opcode (igual que en absoluto) forman la dirección de 16 bits bits en el que estará alojado el byte menos significativo de otra dirección de 16 bits. Esa es la direccion en la que haremos el salto. Por ejemplo supongamos que recibimos los operandos AA Y BB, y $BBAA contiene el valor CC y en $BBAA + 1 esta el valor DD, entonces la direccion final a la que vamos a hacer el salto será $DDCC. Este modo de direccionamiento tiene un bug en el hardware donde si el low byte es FF no cambiara de pagina sino que se mantiene en la misma pagina dándose vuelta y volviendo al 00 y devolviendo una dirección invalida.

- **Indexed Indirect** : Este modo de direccionamiento toma un solo operando y le agrega el valor del registro X y el resultado será el offset para indexar dentro de la pagina zero y obtener la dirección de 16 bits que buscamos, luego funciona igual que indirect.

- **Indirect Indexed**: Se provee un byte como operando que será el indice dentro de la pagina zero, leyendo esa celda y la siguiente obtenemos la dirección efectiva de 16 bits a la que se le sumara el valor contenido en el registro Y, si esta suma causa un cambio de pagina la operación demorara un ciclo más de CPU.

Instrucciones

El 6502 tiene 56 instrucciones diferentes, algunas se puedan usar con distintos modos de direccionamiento lo cual forma un total de 151 opcodes “legales” de los 256 posibles. El largo de una instrucción varia entre 1 y 3 bytes dependiendo de que modo de direccionamiento utilice, siendo el primer byte el opcode y los restantes los operandos. Los tipos de instrucciones se puede calificar de la siguiente manera:

- **Access**: Carga valores de memoria a un registro y viceversa.
- **Transfer**: Copia el valor de un registro en otro.
- **Arithmetic** : Sumas, restas, incrementos y demás operaciones aritméticas.
- **Shift**: Operaciones de corrimiento de bits hacia la izquierda o hacia la derecha.
- **Bitwise**: Operaciones logicas como AND, OR, EOR (XOR), etc.
- **Compare** : Se hacen comparaciones entre dos valores sin modificarlos pero activando flags.
- **Branch**: Saltos condicionales como por ejemplo saltar a $00XX si la flag zero esta activada.
- **Jump**: Saltos a otras direcciones del programa como por ejemplo una subrutina.
- **Stack** : Operaciones de stack como push y pull.
- **Flags** : Operaciones para activar y desactivar alguna flag en particular.
- **Other** : Dentro de esta categoria esta solo la operación NOP (NO Operation) que es para gastas ciclos de CPU sin hacer nada. (*NES Instruction reference*, 2024)

Se puede encontrar la tabla entera de instrucciones en (Hors, 2023)

Interrupciones

Las interrupciones como su nombre lo indica, interrumpen la secuencia normal de ejecución del programa y el CPU debe encargarse de atenderlas. Generalmente las interrupciones son causadas por otros componentes de hardware que requieren la atención del CPU aunque también puede ser causadas por el software. La NES tiene 3 tipos de interrupciones cuyas direcciones de rutinas de atención están indicadas dentro del Interrupt Vector Table (MicrocontrollersLab, 2022) en el rango $FFFA-$FFFF dentro del código del juego.

Las IRQs, o interrupciones enmascarables, son generadas por el software utilizando la instrucción BRK (break), para eso el cartucho debe contener un mapper con la capacidad de generar una IRQ (mas adelante ahondaremos sobre ellos). Estos son ignorados si el CPU tiene la flag ***I*** de deshabilitar interrupciones, habilitada, en caso de estar habilitadas si ocurre una IRQ el CPU guarda su estado en la pila y salta a la dirección formada por los bytes alojados en $FFFE y $FFFF.

Las NMI o interrupciones no inmascarables, son interrupciones que el CPU no puede ignorar a diferencia de las IRQ y debe atenderlas cuanto antes, este tipo de interrupciones solo son generadas por el PPU cuando ocurre un Vblank al final de cada frame. Esto le da aviso al CPU de que la PPU termino de dibujar el frame actual y puede proceder a actualizar el game state y con los nuevos inputs para el siguiente frame. La dirección a la rutina de NMI se encuentra en $FFFA y $FFFB.

Por ultimo esta la interrupcion de Reset, esta es disparada cuando el sistema esta iniciando o cuando el usuario presiona el boton de reset en la consola. La dirección de la rutina se encuentra en $FFFC y $FFFD. La CPU demora 7 ciclos en empezar a ejecutar la rutina de interrupción y en el raro caso de que ocurrieran las 3 al mismo tiempo, el orden de prioridad es Reset seguida de NMI y por ultimo IRQ. (Diskin, 2004)

3.2.2. PPU

A diferencia de la 6502 que, como se mencionó anteriormente, era bastante obsoleta, la PPU (su nombre era **2C02**) era considerada un chip de manejo de gráficos de videojuegos bastante avanzado para la época que generaba una señal de video compuesto de 240 líneas de pixeles para ser recibida por un televisor con esa entrada de video analógica (NFI, 2024).

PPU Registers

El CPU se comunica con la PPU por medio de los registros de PPU que se encuentran en el espacio de direccionamiento del CPU, concretamente en el rango **$2000–$2007** , estos registros son :

- **PPUCTRL ($2000): PPU Control Register**
  - Controla la configuración básica de la PPU, como la selección de *Name Tables* tamaño del sprite y habilitar el NMI.
  - **Significado Bits :**
    - **6**: Habilitar NMI al final del frame
    - **7**: Activa el modo master/slave (esto no se usa en nada, si se activara en una consola real dañaria la PPU) (NesDev, 2024b)
    - **5:** Tamaño del sprite (1 = 8x16, 0 = 8x8).
    - **4:** Tabla de patrones para sprites (1 = $1000, 0 = $0000).
    - **3:** Tabla de patrones para fondos (1 = $1000, 0 = $0000).
    - **2:** Define si la dirección de la VRAM incrementa en 1 (modo horizontal) o 32 (modo vertical).
    - **1-0:** Selección de Name Table (00 = $2000, 01 = $2400, 10 = $2800, 11 = $2C00).

- **PPUMASK ($2001): PPU Mask Register**
  - Controla qué partes de la imagen se renderizan y cómo se procesan los colores.
  - **Significado Bits:**
    - **7:** Intensificar tonos rojos.
    - **6:** Intensificar tonos verdes.
    - **5:** Intensificar tonos azules.
    - **4:** Dibujar sprites.
    - **3:** Dibujar fondo.
    - **2:** Mostrar sprites en los 8 pixeles más a la izquierda de la pantalla.
    - **1:** Mostrar fondo en los 8 pixeles más a la izquierda de la pantalla.
    - **0:** Modo blanco y negro (Greyscale).
- **PPUSTATUS ($2002): PPU Status Register**
  - Proporciona información sobre el estado de la PPU.
  - **Significado Bits:**
    - **7:** Bandera de VBlank (1 = PPU está en VBlank).
    - **6:** Sprite zero-hit (1 = el sprite zero ha colisionado con un fondo no transparente).
    - **5:** Sprite Overflow Flag (Se activa si hay más de un sprite en la misma scanline).
    - **4–0:** En el contexto de emulación son ignorados.
- **OAMADDR ($2003): OAM Address Register** Especifica la dirección dentro de la memoria OAM (Object Attribute Memory) para leer o escribir datos relacionados con los sprites.
- **OAMDATA ($2004): OAM Data Register** Permite leer o escribir datos de la memoria OAM. Se utiliza para definir atributos de los sprites (posición, tile, color, etc.) aunque es más lento que el OAM DMA por lo que se suele usar este ultimo.
- **PPUSCROLL ($2005): PPU Scroll Register** Configura la posición de scroll. Los valores escritos definen que pixel estará en la esquina superior izquierda del próximo frame.
- **PPUADDR ($2006): PPU Address Register** Dado que el CPU y el PPU estan en buses separados. El CPU solo puede escribir en la VRAM a través de un par de registros, en este registro se carga la dirección de VRAM que se quiere escribir.

- **PPUDATA ($2007): PPU Data Register** Permite leer o escribir datos en la memoria de video VRAM cuya dirección se define en el registro **PPUADDR**. Después de cada lectura/escritura, la dirección se incrementa automáticamente según el valor en el bit 2 del registro **PPUCTRL**.

- **OAMDMA ($4014): OAM DMA Register** Usado para transferir rápidamente datos a la memoria OAM desde la RAM del sistema. El CPU escribe en el registro el byte que define la pagina que será transferida (dirección $XX00-$XXFF) al OAM del PPU. Luego el CPU es suspendido y empieza a hacer la transferencia copiando 256 bytes.

Mapeado de Memoria y Manejo del background

La PPU tiene su propio espacio de direcciones completamente separado de la CPU, conformado por 10 KBs de memoria, 8 KBs de ROM o RAM en el Game Pak y 2 KBs de memoria RAM (VRAM) en la consola, sumado a 32 Bytes de Palette Ram . En la Figura 3.5 podemos ver el el mapeo de memoria que, como la CPU, también mapea 64 KBs de memoria.

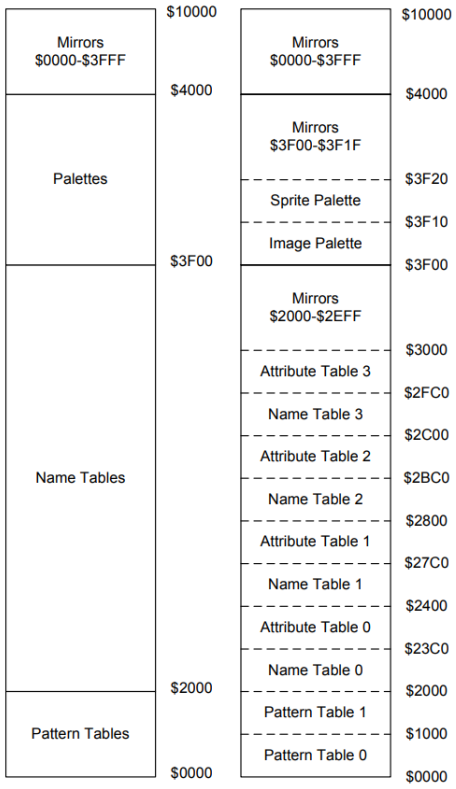


Figura 3.5: PPU mapeo de memoria

Para explicar cada componente de esta parte no iremos en orden como hicimos en la sección de mapeo de memoria de la CPU sino que se hará de una forma que, creemos, facilitara la compresión de toda esta estructura que puede llegar a ser mareadora y complicada, también omitiremos detallar los mirrors ya que son el mismo concepto visto en la sección de mapeo de memoria de CPU.

- **$3F00–$3F1F Palettes**: La NES contiene un total de 52 colores en su paleta a pesar de tener espacio para 64. Sin embargo la NES no es capaz de mostrar todos estos colores a la vez ya que la Palette RAM cuenta con 32 Bytes de memoria y es insuficiente para hacerlo. En esos 32 Bytes se manejan dos tablas de 16 Bytes de paletas, la Image Palette con las paletas usadas para el Background y la Sprite Palette para las paletas usadas en los sprites, Dentro de cada una de estas dos tablas se manejan 4 paletas con 4 colores cada una y en cada entrada de estas paletas se encuentra alojado el Index de un color (no el valor del color propiamente dicho), además en todas las paletas su primer color será el valor que corresponde a la transparencia, por lo que en verdad solo hay 3 colores en cada paleta. Es decir que para direccionar un color dentro de la Palette RAM son necesarios 5 bits, por ejemplo 10101, el bit marrón indica si es sprite o background, los azules indican que paleta se usa, y los rojos indican finalmente el color. (Diskin, 2004) Vamos a ilustrar esto en la Figura 3.6 para ayudar a entender la estructura.

		Paleta 0				Paleta 1				Paleta 2				Paleta 3			
		$x0	$x1	$x2	$x3	$x4	$x5	$x6	$x7	$x8	$x9	$xA	$xB	$xC	$xD	$xE	$xF
Background	$3F0x	*0**	1	2	3	*0*	1	2	3	*0*	1	2	3	*0*	1	2	3
Sprite	$3F1x		1	2	3		1	2	3		1	2	3		1	2	3

Figura 3.6: Estructura de la Palette Ram

- **$0000–$1FFF Pattern Tables**: La NES tiene dos Pattern Tables: la Pattern Table 0 y la Pattern Table 1. Estas almacenan tiles de 8×8 píxeles. En una de ellas se almacenan los tiles usados para el background, y en la otra, los tiles para los sprites. La asignación de qué tabla será utilizada para sprites y cuál para background depende de cada juego. La mayoría de los juegos almacenan estas tablas en la CHR-ROM del Gampe Pak, aunque algunos juegos más complejos utilizan una RAM que van llenando en tiempo de ejecución. Cada entrada de estos tiles contienen los dos bits de color que usarían de una paleta cualquiera, si dijimos que para obtener el index de un color necesitamos 5 bits x xx xx, en el Pattern Table se guardan los de color rojo, y se lo hace cómo se ve en la Figura 3.7

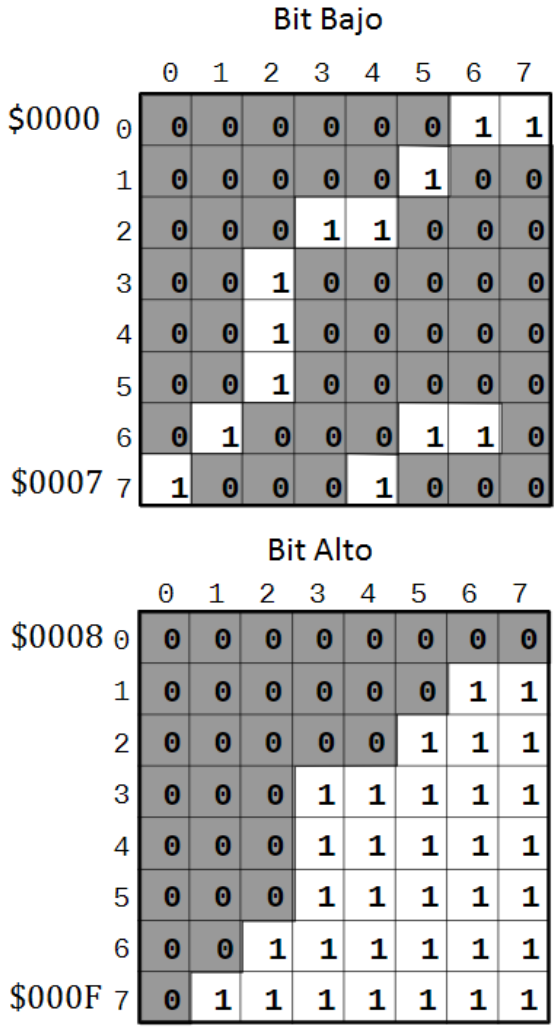


Figura 3.7: Almacenamiento de un Tile

De los 5 bits `x xx xx`, que necesitamos para obtener el index de un color ya sabemos cómo obtener el bit marrón de sprite/background y donde se almacenan los dos bits rojos de color, lo siguiente será explicar donde se utilizan estos tiles y en donde se obtienen los bits azules de paleta. (Morlan, 2019)

- **$2000–$2EFF Name Tables / Attribute Tables** : Las *Name Tables* son básicamente, matrices de 32x30 donde cada entrada es un Tile Index, y cada *Name Table* representa un área gráfica de 256x240 pixeles donde

se dibujara el background como se ve en la Figura 3.8.

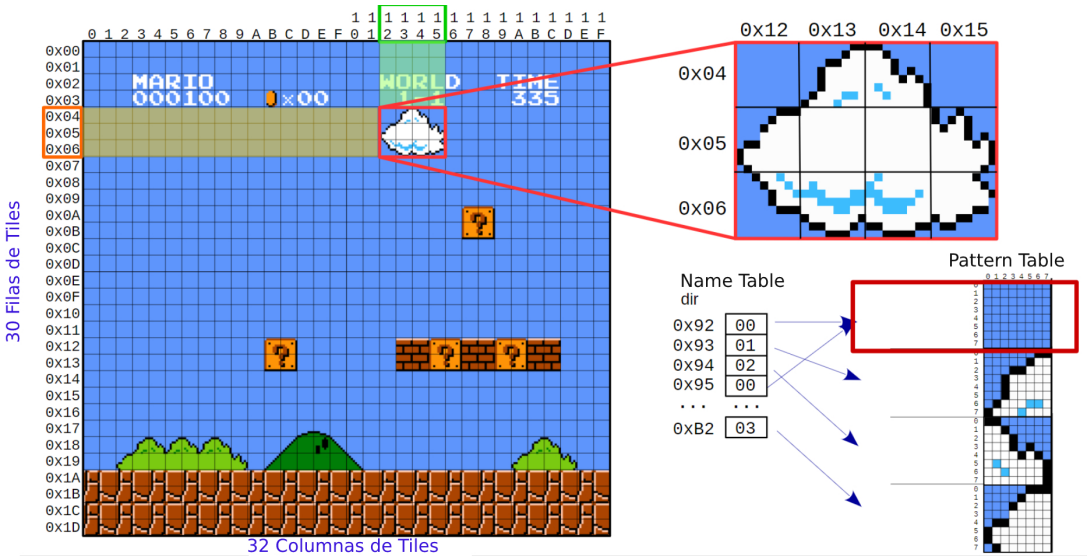


Figura 3.8: *Name Table* NES (Machado, 2001)

Cada *Name Table* tiene asociada un attribute table de tamaño 64 bytes que se aloja al final de cada *Name Table*, los attributes tables son los que almacenan los dos bits para definir la paleta que nos estaban faltando (los azules), aunque la forma de hacerlo no es tan sencilla como a uno le gustaría. Los tiles de una *Name Table* se organizan en bloques de 4x4 . Cada bloque se divide a su vez en 4 cuadrantes de 2x2, como se muestra a continuación:

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Cada bloque es representado por un byte en la attribute table. Este byte contiene 2 bits para cada cuadrante, indicando qué paleta de colores utilizar:

- **Bits 0-1:** Paleta del cuadrante *superior izquierdo* (tiles A, B, E, F).
- **Bits 2-3:** Paleta del cuadrante superior derecho (tiles C, D, G, H).
- **Bits 4-5:** Paleta del cuadrante inferior izquierdo (tiles I, J, M, N).
- **Bits 6-7:** Paleta del cuadrante inferior derecho (tiles K, L, O, P).

Por ejemplo:
Supongamos que el byte de la attribute table para un bloque contiene el valor binario `10 01 00 11`. Esto significa:

- `11` (Paleta 3): Cuadrante inferior derecho.
- `00` (Paleta 0): Cuadrante inferior izquierdo.
- `01` (Paleta 1): Cuadrante superior derecho.
- `10` (Paleta 2): Cuadrante superior izquierdo.

La siguiente tabla ilustra cómo se asignan las paletas:

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

El sistema tiene 2KBs de VRAM para manejar 2 *Name Tables* con su attribute table, sin embargo como se ve el en el layout de mapeo de memoria, se direccionan 4. Esto es porque 2 de ellas se espejan de las "*reales*" , existen distintos tipos de mirroring:

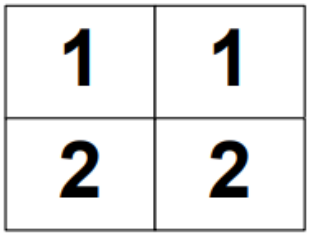


Figura 3.9: Mirroring Horizontal

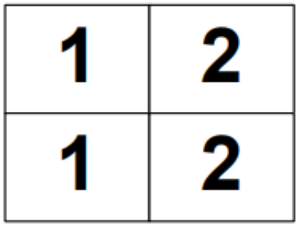


Figura 3.10: Mirroring Vertical

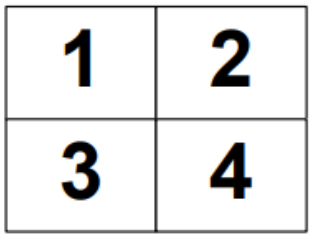


Figura 3.12: Mirroring Four-Screen

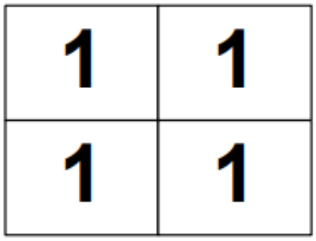


Figura 3.11: Mirroring single-screen

En esta ultima no hay mirroring y es necesario que el cartucho cuente con 2KBs más de memoria RAM para poder soportar las dos name y attribute tables adicionales. (Diskin, 2004) Hasta aquí explicamos los componentes que se muestran en el layout de mapeo de memoria de la PPU y adicionalmente de cómo se esta almacena los gráficos del background, ahora pasaremos a hablar de los sprites.

Sprites

Los sprites son los personajes o objetos animados que vemos en el juego y estos se manejan de forma distinta a al background, normalmente estos personajes estan conformados por varios sprites y es lo que permite que tengan tamaños grandes y animaciones varias utilizando poca memoria como vemos en la Figura 3.13.

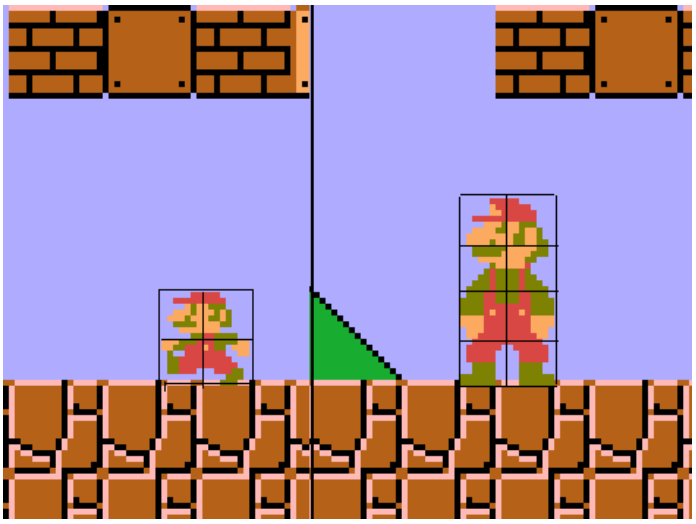


Figura 3.13: Personaje hecho con múltiples sprites. Fuente: nesdoug.com

Cada sprite es representado por 4 bytes:

1. **Y Position (Coordenada Vertical)**
   - Indica la posición del sprite en el eje Y, específicamente de la parte superior del sprite en la pantalla. El valor debe ser de 0 a 239 (las posiciones fuera de este rango no se renderizan).
2. **Tile Index**
   - Identificador del Tile para poder ir luego a obtenerlo en la Pattern Table.
3. **Attributes** Especifica las propiedades visuales del sprite:
   - Bits 0-1: Definen la paleta que se usa de las 4 disponibles en la palette table.
   - Bit 2-4: No utilizados (deben estar en 0).
   - Bit 5: Prioridad (determina si el sprite se dibuja delante o detrás del fondo).
   - Bit 3: Flip horizontal (invierte el sprite horizontalmente).
   - Bit 4: Flip vertical (invierte el sprite verticalmente).
4. **X Position (Posición horizontal)**
   - Indica la posición del sprite en el eje X, específicamente la coordenada de la esquina izquierda del sprite en la pantalla. Rango válido de 0 a 255.

Esta información se almacena en una memoria especial llamada **OAM** (Object Attribute Memory) que tiene un tamaño fijo de 256 bytes (64 sprites) y está diseñada específicamente para manejar los datos de los sprites que se dibujan en pantalla.

La OAM no es accesible directamente por la CPU de la NES durante el funcionamiento normal; en su lugar, se utiliza el registro **$4014 OAMDMA** del bus de la CPU para cargar datos desde la memoria principal a la OAM (como mencionamos en la sección de registro). Esto permite actualizar rápidamente los datos de los sprites entre cuadros. Además, dado que la NES solo puede dibujar 8 sprites por scanline, la información en la OAM también se utiliza para determinar cuáles sprites serán visibles cuando haya más de 8 en una misma línea.

En la NES, los sprites pueden tener un tamaño de 8x8 píxeles o 8x16 píxeles. Cuando se utiliza el modo de 8x16 píxeles, un sprite ya no se representa con un único tile, sino con dos tiles consecutivos. El primer tile se utiliza para la parte superior del sprite y el segundo para la parte inferior.

Este modo es útil para representar objetos grandes, como personajes o enemigos principales, sin necesidad de usar múltiples entradas de OAM para construir un objeto mayor, lo que puede ahorrar tiempo de CPU. Sin embargo también implica que su manejo es más complicado y no es posible usar operaciones de rotación. (M-Tee, 2024)

La prioridad de dibujado de los sprites viene definida por su posición dentro de la OAM, el primer sprite conocido como sprite zero es el que tiene la mayor prioridad, es decir si algún otro sprite colisiona con el sprite zero, este se dibujara encima del otro.

Ya hablamos del background y de los sprites, si la NES solo tuviera juegos single-screen esto ya seria suficiente, pero afortunadamente esto no es así que pasemos al ultimo ítem de este repaso sobre la PPU.

Scrolling

Cuando hablamos de scrolling nos referimos al movimiento de la pantalla o del fondo de un juego para dar la sensación de que nuestro personaje u objeto avanza por un mundo mucho más que lo que una pantalla puede mostrar y lo hace de manera suave, esto lo hace gracias a que la NES maneja multiples *Name Tables*, la NES es capaz de mostrar 960 tiles pero almacena el doble en la VRAM. El scrolling puede ser tanto Horizontal como Vertical o incluso ambos como podemos ver con juegos como Super Mario Bros 3, en la Figura 3.14 se puede ver que, en un momento dado, el fondo que veamos en pantalla estará compuesto por fracciones de 2 o más *Name Tables* (en caso de que haya scroll multidireccional o diagonal).

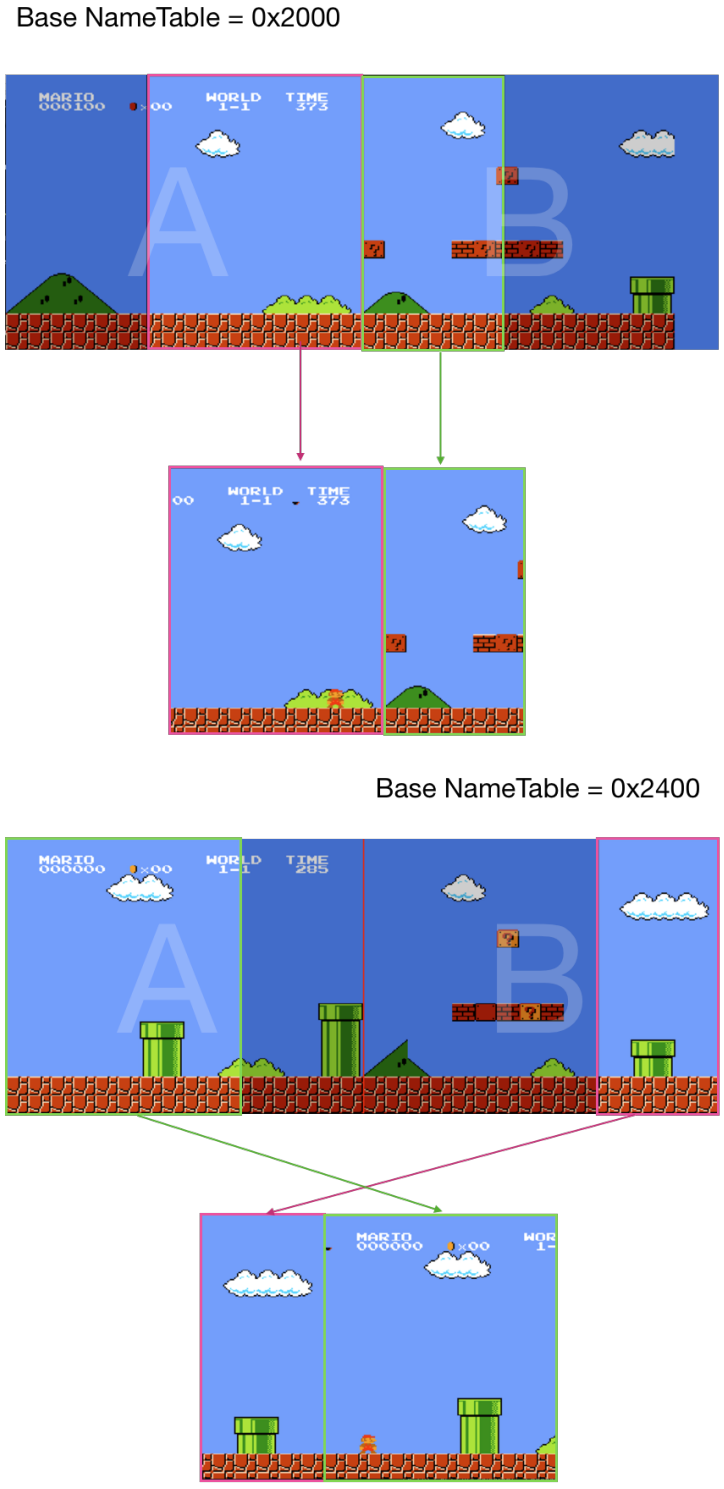


Figura 3.14: Diagrama Scrolling (Bagmanov, 2020)

A medida que nuestro personaje "avanza" y nuestro campo de visión se va moviendo y se van actualizando los tiles contenidos en la *Name Table* por fuera de nuestra vista para un scrolling limpio y sin glitches **siempre y cuando el modo de mirroring sea el adecuado**. Por ejemplo para el caso de un juego con scroll horizontal, es conveniente que el mirroring sea del tipo vertical, si un juego con mirroring vertical intenta scrollear verticalmente veremos glitches como por ejemplo el que se muestra en la Figura 3.15

Figura 3.15: Ejemplo de un glitch por scrolling sin mirroring adecuado. Fuente contra.fandom.com

Si volvemos a mirar el diagrama de mirroring vertical vemos que la *Name Table* que tenemos debajo de 1 es un mirror de este mismo *Name Table* por lo que el glitch que vemos en la Figura 3.15 es que en la fila de tiles del borde superior del frame vemos los mismo que en la fila de tiles del borde inferior.

Para implementar el scrolling el sistema mantiene un registro de 16 bits (se usan 15 en realidad) que direcciona cada una de las entradas a las 4 *Name Tables*. Este registro es comúnmente llamado **Loopy Register** en el mundo de la emulación, en honor a este entusiasta que hizo un documento llamado "***The Skinny On NES Scrolling***" exponiendo el manejo del scrolling de la NES en el año 1999. Puede encontrarse una versión traducida de este documento en (Loopy, 1999)

El Loopy Register tiene la siguiente forma.

Fine Y (Bits 12-14)	Name Table (Bits 10-11)	Coarse Y (Bits 5-9)	Coarse X (Bits 0-4)

- Fine Y: Coordenada Y a nivel de pixel dentro de un tile, su valor va de 0-7 (3 bits).
- Name Table: Selección de una de las 4 Name Table (2 bits).
- Coarse Y: Coordenada Y a nivel de tiles dentro de la Name Table (5 bits).
- Coarse X: Coordenada X a nivel de tiles dentro de la Name Table X (5 bits).

Existe otro registro temporal de 16 bits de direccionamiento donde se almacena los 3 bits del Fine X como coordenada a nivel de pixel dentro de un tile.

El valor de estos registros es seteado por medio del registro **$2006 PPUCTRL** donde en la primer escritura define el Coarse X y el Fine X y la segunda escritura el Coarse Y y Fine Y, definiendo las coordenadas del pixel que será la esquina superior izquierda del nuevo frame, haciendo esto en cada frame es como se implementa el suave movimiento del scrolling.(Diskin, 2004)

Continuar explorando todos los detalles de la PPU extendería en demasía este documento. Por eso, se han cubierto únicamente aspectos esenciales para comprender cómo se manejan los gráficos. Para un análisis más profundo, se recomienda consultar el proceso de rendering (NesDev, 2022c) y su diagrama de timing (Ulfalizer, 2022) . Ahora pasemos a hablar de cómo se maneja el input del gamepad.

3.2.3. Inputs del Gamepad

Figura 3.16: Gamepad NES. Fuente: Amazon.com

La NES usaba el control mostrado en la Figura 3.16 que tenia 4 botones y un dpad de 4 direcciones. Para obtener el estado de un control la NES lee 8

veces el registro de I/O a modo de shift register () obteniendo el estado de un botón a la vez, siguiendo el orden A, B, Select, Start, Up, Down, Left, Right. El primer control se lee desde la dirección $4016 y el segundo en la $4017. Para iniciar el proceso de lectura de ambos controles, la NES escribe el valor 1 en la direccion $4016. Esto recarga el registro de entrada continuamente hasta que el CPU vuelve a escribir en esta puerto el valor 0, eso detiene la recarga del registro y captura el estado del control para proceder a leerlo. (NesDev, 2022d) Afortunadamente es un proceso sencillo, pasemos ahora a hablar del Game Pak

3.2.4. Game Pak

Los juegos de NES venían en cartuchos conocidos como Game Pak. La primer version de los cartuchos eran simples. Estos traían dos bancos de memoria ROM, el PGR-ROM donde se alojaba el codigo del juego y el CHR-ROM donde se alojaban los graficos. Cuando el cartucho se inserta en la consola el PGR-ROM se conecta al bus de CPU y el CHR-ROM al de PPU.

Figura 3.17: Game Pak. Fuente: nintendo.fandom.com

Mappers

La limitada memoria era suficiente para los primeros juegos, pero a medida que empezaron a ser más ambiciosos la memoria era insuficiente. Para que los cartuchos puedan tener más memoria ROM, dado que la CPU de la NES solo puede acceder máximo a 32KB de PRG ROM y PPU a 8KBs de CHR-ROM, se incluyo hardware en los cartuchos para hacer switch banks conocidos como memory mappers o MMC (Memory Management Chip). La memoria de un cartucho grande se divide en bloques, llamados **bancos o banks**, de un tamaño

fijo 16 KB para el código del juego(PGR-ROM) o 4 KB para los gráficos(CHR-ROM). Solo un banco o una combinación específica de bancos puede ser visible para la consola en un momento dado. Por ejemplo, si el jugador avanzaba a un nuevo nivel, el juego podía escribir un valor en una dirección especial en el rango $8000-$FFFF que llegaría al mapper y en respuesta cambiar los bancos visibles para la NES, cargando el nuevo nivel o gráficos mientras mantenía otras partes del cartucho "ocultas". Esto daba la posibilidad de que los juegos puedan ser mucho más grandes y complejos de los que la consola podría manejar directamente. Existe una lista con un total de 767 tipos de mappers distintos (NesDev, 2022a), vamos a nombrar algunos de los más usados. (Diskin, 2004)

MMC1 (Mapper 1)

- **Características principales:**
  - Soporte para PRG-ROM y CHR-ROM con switch banks.
  - Configuración flexible de mirroring cambiando el tipo en tiempo de ejecución (horizontal, vertical o control manual).
  - Soporte para baterías de respaldo (Save RAM), permitiendo guardar partidas (normalmente se usaban pilas similares a las que usan los motherboards de pc, estas mantenían con energía una RAM con el juego guardado).
  - Máximo de 8 bancos de 16 KB de PRG-ROM (128 KB en total).
- **Ventaja:**
  - Versatilidad para juegos más grandes y complejos con gráficos detallados.
  - Primera implementación de guardado en juegos permitiendo juegos mucho más largos.
- **Limitación:** Configuración lenta debido a su sistema de switch basado en escritura en registros de 1 bit.
- **Juegos que lo usaban:** Metroid, The Legend of Zelda, Kid Icarus.

UNROM (Mapper 2)

- **Características principales:**
  - Soporta switch banks de PRG ROM.
  - No tiene soporte para switch banks de CHR-ROM, por lo que utiliza CHR RAM, donde los gráficos se cargan directamente desde la PRG ROM al inicio.
  - Máximo de **8 bancos de 16 KB** de PRG ROM (128 KB en total).

- **Ventaja:** Permite juegos más grandes que los que podían caber en los 32 KB de PRG ROM estándar del NES.
- **Limitación:** Gráficos menos sofisticados debido a la falta de soporte para CHR-ROM.
- **Juegos que lo usaban:** Mega Man, Castlevania.

CNROM (Mapper 3)

- **Características principales:**
  - Permite switch banks de CHR-ROM, lo que significa gráficos más detallados y variados.
  - No tiene soporte para switch banks de PRG ROM; el tamaño del código del programa permanece limitado a 32 KB.
- **Ventaja:** Habilita juegos con gráficos más variados gracias al switch banks de CHR-ROMs.
- **Limitación:** El código del juego sigue limitado a los 32 KB.
- **Juegos que lo usaban:** Arkanoid, Cybernoid.

MMC3 (Mapper 4)

- **Características principales:**
  - Soporte avanzado para conmutación de PRG ROM y CHR-ROM.
  - Scrolling selectivo, permitiendo desplazar partes específicas de la pantalla, mientras se deja otras partes estáticas.
  - Generación de IRQs facilitando efectos gráficos avanzados y sincronización.
  - Máximo de 32 bancos de 16 KB de PRG ROM (512 KB en total).
- **Ventaja:**
  - Permite gráficos y mecánicas complejas, haciendo posible juegos de calidad superior.
  - Soporte para mapas de mayor tamaño y efectos como animaciones de parallax.
- **Limitación:** Más caro y complejo de implementar en cartuchos.
- **Juegos que lo usaban:** Super Mario Bros. 2, Super Mario Bros. 3, Kirby's Adventure.

Formato de la Imagen de un Cartucho de NES

Los archivos que contienen la imagen de los los cartuchos y que son utilizados para ejecutarse en un emulador suelen denominarse **ROMs**, en referencia a los chips ROM utilizados para almacenarlo. Una simple copia del contenido del cartucho no es suficiente para ejecutarlos, ya que no ofrece una manera de identificar qué significa cada parte del archivo. Por ello ha surgido el formato iNES.

El formato de archivo **iNES** fue definido originalmente por **Marat Fayzullin** para su uso en el emulador iNES. Desde entonces, este formato ha sido adoptado por la mayoría de los emuladores y es el formato más común para las imágenes ROM. Los archivos en formato iNES usan la extensión de archivo ***.nes**. El formato incluye un encabezado de 16 bytes al inicio del archivo, que contiene información con la estructura mostrada en la Tabla 3.1. (Diskin, 2004)

El formato iNES es simple y tiene una amplia compatibilidad con emuladores, sin embargo, tiene limitaciones al no ser suficientemente flexible para juegos más complejos o juegos homebrew, ya que no soporta bien características avanzadas de la consola o mappers de memoria más sofisticados, por esto nació un formato retrocompatible con iNES llamado NES2.0. (NesDev, 2022b)

Byte(s)	Descripción
0-3	**Debe contener la palabra "NES" seguido del valor** `$1A`: para identificar que el archivo está en formato iNES.
4	**Tamaño de PRG-ROM**: Número de bancos de 16 KB de PRG-ROM.
5	**Tamaño de CHR-ROM**: Número de bancos de 8 KB de CHR-ROM (0 indica uso de CHR-RAM).
6	**Flags 6 (Mapper y características):** ▪ Bit 0: Mirroring (0 = horizontal, 1 = vertical). ▪ Bit 1: Presencia de PRG-RAM con batería para guardar partidas. ▪ Bit 2: Presencia de un trainer (bloque de 512 bytes opcional en los juegos de NES que se usa para trucos, depuración o modificaciones) ▪ Bit 3: Si este bit esta seteado, indica que se trata de un mirroring de 4 pantallas e ignora el bit 0. ▪ Bits 4-7: Bits bajos del número identificador del mapper.
7	**Flags 7:** ▪ Bits 0-3: Reservado., deberia ser siempre 0. ▪ Bits 4-7: Bits altos del número identificador del mapper.
8	**Tamaño de PRG RAM**: Número de bancos de 8 KB de PRG-RAM (0 indica 8 KB por defecto).
9	**Flags 9**: Región de TV (0 = NTSC, 1 = PAL).
10	**Flags 10**: Características adicionales (poco utilizadas).
11-15	**Reservado**: Deben rellenarse con ceros para compatibilidad futura.

Tabla 3.1: Descripción de los bytes del encabezado iNES.

Capítulo 4

Introducción al Gaming Networking y Relevamiento de Plataformas de Emulación Existentes

En este capítulo haremos un repaso por los fundamentos del Gaming Networking y expondremos los modelos existentes, dando un breve detalle de cómo funciona cada uno y cuándo utilizarlos. Por ultimo haremos una relevamiento de algunas plataformas de emulación que sean capaces de emular NES.

4.1. Fundamentos del Gaming Networking

El Gaming Networking es un campo técnico que combina principios de redes informáticas con mecánicas específicas de videojuegos. Antes de discutir los modelos de implementación que existen, es esencial establecer una base conceptual sólida. Por lo que haremos un sumario de elementos y conceptos básicos que se manejan en este campo.

4.1.1. ¿Qué es Gaming Networking?

Gaming Networking se refiere al proceso de habilitar la comunicación en tiempo real entre múltiples dispositivos o clientes en un entorno de juego. Esto implica la transmisión y recepción de paquetes de datos entre clientes y servidores, asegurando una experiencia de juego sincronizada y receptiva. Esta área de la computación es compleja y requiere una cuidadosa consideración de factores como la latencia, el ancho de banda y la pérdida de paquetes.

4.1.2. Sumario de conceptos básicos

Netcode

El término *netcode* es una expresión coloquial utilizada tanto por jugadores como por desarrolladores para referirse al gaming networking.

Ping

El *Ping* o RTT es el tiempo total que tarda un paquete de datos en viajar de un jugador A a un jugador B y regresar. En la Figura 4.1 se puede ver un ejemplo visual del concepto.

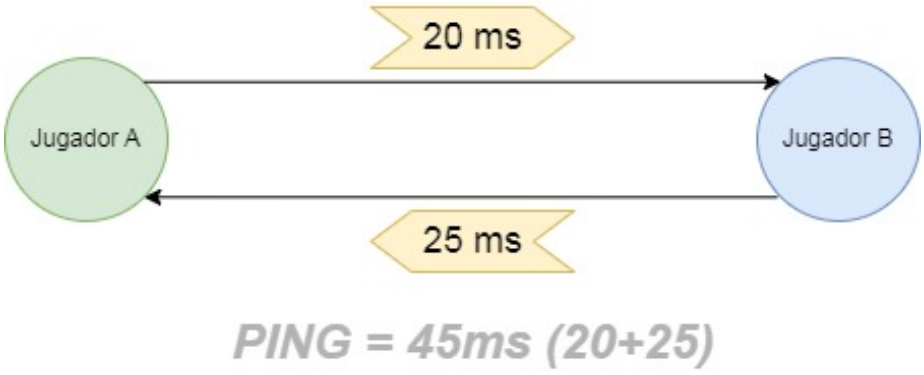


Figura 4.1: Ejemplo visual de ping. Fuente propia

Cuando una PC o consola "hace ping" a un servidor, envía una solicitud de eco ICMP (Protocolo de Mensajes de Control de Internet) al servidor del juego, que responde a esta solicitud devolviendo una respuesta de eco ICMP. El tiempo transcurrido entre el envío de la solicitud y la recepción de la respuesta es el tiempo de ping al servidor del juego. Valores de ping más altos indican más latencia o lag, lo que afecta negativamente la experiencia de juego. Por ello, es preferible jugar en conexiones con valores de ping bajos para que los juegos se sientan ágiles y receptivos. (Battle(non)sense, 2017)

Routing

El tiempo que tarda un paquete de datos en llegar a su destino no solo depende de la distancia física entre el jugador y el servidor, sino también de las características de la ruta que toma, es decir, depende de la topología de la red. Los cables de cobre y fibra óptica no siempre siguen un camino directo, lo que significa que un servidor más lejano podría tener una conexión más rápida debido a una ruta más eficiente. En la Figura 4.2 podemos ver un ejemplo donde el jugador remoto 2 esta más cerca físicamente al jugador local que el Remoto

1 pero a causa de como esta hecha la topología de la red, el ping entre ellos es mayor.

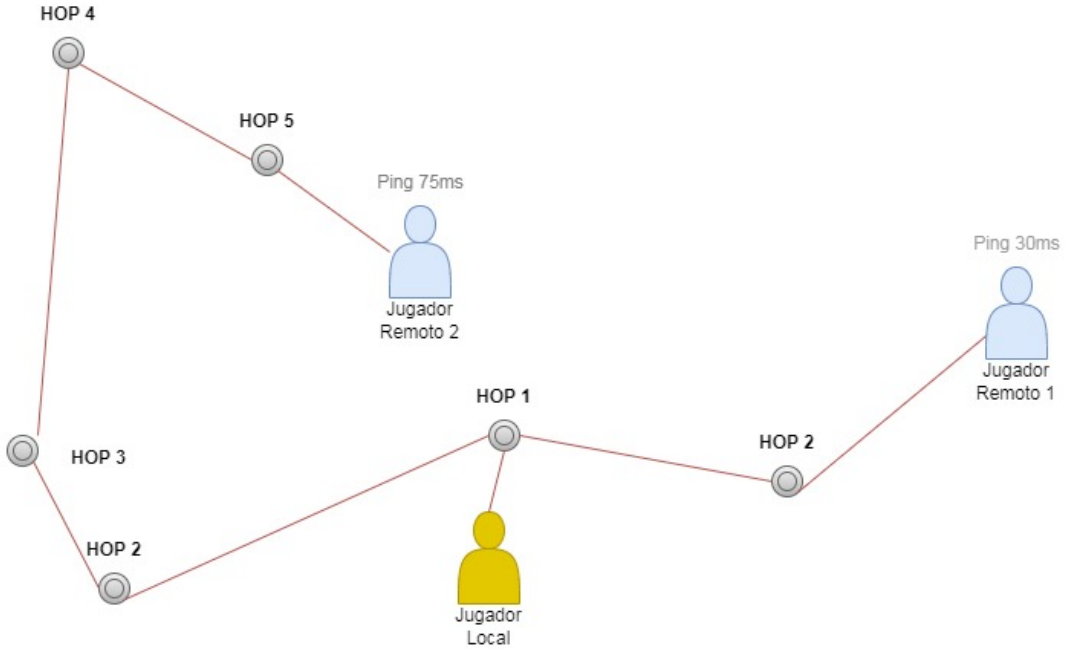


Figura 4.2: Ejemplo de Routing. Fuente propia

Además, los datos deben pasar por varios puntos intermedios, conocidos como saltos (HOPS) antes de llegar al servidor. Cada salto adicional puede aumentar el ping y el riesgo de pérdida de datos.

Existen servicios que intentan optimizar estas rutas, reduciendo el número de saltos y encontrando trayectos más rápidos. Sin embargo, la utilidad de estos servicios depende de la calidad de la infraestructura del proveedor de Internet (ISP) del usuario. Si el ISP ya ofrece rutas rápidas y optimizadas, estas herramientas podrían incluso empeorar la latencia. Por eso es recomendable probar antes de contratar un servicio de este tipo. (Battle(non)sense, 2017)

Pérdida de paquetes

La pérdida de paquetes ocurre cuando un paquete de datos no logra llegar a su destino, lo cual puede afectar significativamente aplicaciones en tiempo real como los videojuegos, ya que enviar nuevamente el paquete perdido aumenta la latencia.

Entre las causas más comunes se encuentran problemas en la red local, como tarjetas de red defectuosas, interferencias en conexiones WiFi o Powerline,

o cables y puertos de red dañados. Por otro lado, el router también puede ser responsable, ya sea por fallos de hardware, firmware desactualizado o un consumo excesivo del ancho de banda disponible. En estos casos, reiniciar el router, actualizar su firmware o cambiar a uno que priorice aplicaciones en tiempo real podría ser útil.

Si la pérdida de paquetes ocurre fuera de la red doméstica, en las infraestructuras externas gestionadas por el ISP, lo único que se puede hacer es contactar a su soporte técnico y esperar que puedan solucionar el problema. (Battle(non)sense, 2017)

Update Rate

La frecuencia con la que un juego envía y recibe datos afecta directamente al ping. Si un juego actualiza a una frecuencia de 30Hz (30 actualizaciones por segundo), habrá más tiempo entre actualizaciones en comparación con una frecuencia de 60Hz, lo que agrega un retraso adicional.

Al aumentar el update rate, se puede reducir este retraso adicional. Las tasas bajas no solo afectan la latencia de la red, también pueden causar problemas como los llamados "super bullets" en el contexto de juegos First Person Shooter(FPS). Esto sucede cuando un juego con bajo update rate combina el daño de múltiples disparos en una sola actualización, haciendo que parezca que un disparo único causó mucho más daño de lo esperado. Por ejemplo, si un servidor de un juego envía 10 actualizaciones por segundo, hay 100ms de juego "contenido" en cada actualización. Esto coincide con el tiempo entre disparos de un arma que dispara a 600 balas por minuto. Sin embargo, si un arma dispara a 750 balas por minuto, no hay suficientes actualizaciones para registrar cada disparo individual. En su lugar, varios disparos se agrupan en una sola actualización, haciendo que el daño se acumule y se perciba como si una "super bala" hubiera causado un daño desproporcionado.

Esto deja claro por qué un update rate alto no solo es esencial para mantener la latencia de la red baja, sino también para ofrecer una experiencia online consistente. Cuando el update rate es alto, un pequeño porcentaje de pérdida de paquetes no representa un problema significativo. Sin embargo con tasas bajas la importancia de cada paquete es mucho mayor, si estos se pierden van a degradar considerablemente la experiencia. (Battle(non)sense, 2017)

NAT

La *NAT* (Network Address Translation) permite a varios dispositivos en una red local compartir una única dirección IP pública al reescribir las direcciones en los paquetes que entran y salen del router. Esto proporciona seguridad al bloquear tráfico entrante no solicitado pero trae inconvenientes para conectar con otro jugador para jugar. La NAT funciona asignando direcciones IP "privadas" especiales a las computadoras de un lado del router, mientras conserva una sola dirección IP pública en el lado "público". Cuando los paquetes pasan por el router hacia el exterior, el router reescribe la dirección de origen en los paque-

tes, reemplazando la dirección privada con la IP pública. Cuando los paquetes regresan desde afuera, el router reescribe la dirección de destino en los paquetes, restaurando la dirección IP privada. Por lo general, el router bloquea todo el tráfico entrante, aunque debe permitir cierto tráfico para que, por ejemplo, el navegador pueda cargar páginas web. El router decide qué tráfico dejar pasar basándose en si fue él quien inició la conexión. Si el router envió un paquete primero, recuerda esa conexión y permite que los paquetes de respuesta pasen. El router reconoce los paquetes de respuesta usando no solo la dirección IP sino también un número de puerto que se adjunta a los paquetes salientes. Cuando el servidor responde, los paquetes se envían a la IP pública del router y al puerto asociado. El router utiliza este puerto para decidir qué dirección IP privada debe colocar de nuevo en los paquetes antes de enviarlos a la red local, permitiendo que nuestra computadora los reciba. Para esto el router mantiene una tabla de asignaciones para rastrear qué puerto corresponde a cada dirección IP privada y pública. (Johnston, 2014) En la Figura 4.3 podemos ver un ejemplo de como funciona una NAT.

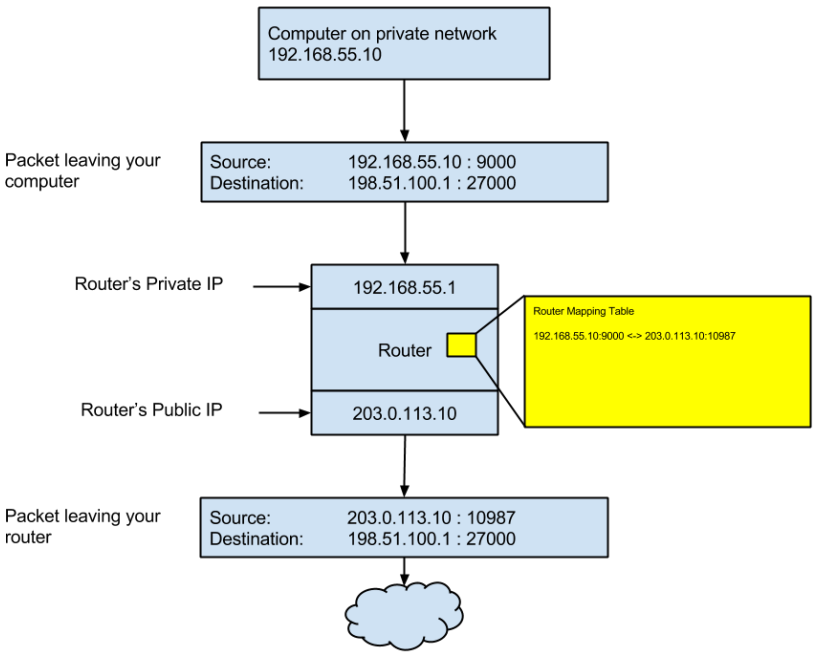


Figura 4.3: Ejemplo de como funciona una NAT. (Johnston, 2014)

4.1.3. UDP vs TCP: ¿Cuál es Mejor Para Implementar un Netcode?

Al desarrollar un juego multijugador, uno de los primeros pasos es decidir qué protocolo de comunicación utilizar. Esta es una decisión crítica, ya que una elección incorrecta puede tener efectos no deseables en la experiencia multijugador online. Repasemos las características de cada protocolo.

TCP:

- **Fiabilidad**: TCP se encarga de confirmar la recepción de los mensajes y retransmitir los paquetes perdidos, garantizando que todos los datos lleguen correctamente.
- **Transferencia de datos en orden**: Los paquetes de datos pueden tomar diferentes rutas para llegar al destino. TCP reorganiza estos paquetes en el orden original en que fueron enviados.
- **Verificación de errores**: Utiliza sumas de verificación (checksums) para garantizar que los datos no se corrompan durante la transmisión.
- **Control de flujo**: TCP regula el flujo de datos para evitar que los recursos de la red se saturen, ajustando la velocidad de envío de los paquetes.
- **Control de congestión**: Si la red está sobrecargada, TCP reduce la tasa de transferencia de datos para ayudar a aliviar la congestión.

UDP:

- **Velocidad**: UDP no necesita establecer una conexión antes de enviar datos, lo que reduce los retrasos.
- **Sin confirmación**: No requiere que el receptor confirme la recepción de datos, evitando la sobrecarga de gestionar confirmaciones y retransmisiones.
- **Flexibilidad**: Los desarrolladores pueden manejar la pérdida de paquetes de manera personalizada, creando capas de confiabilidad específicas para las necesidades de su juego.

TCP es adecuado para juegos que requieren una entrega precisa y ordenada de datos, como los juegos por turnos. Sin embargo, esta fiabilidad tiene un costo: añadir pasos para garantizar que los datos lleguen en orden y sin errores genera retrasos adicionales. En juegos rápidos y en tiempo real, este retraso se traduce en más latencia. En cambio UDP no garantiza la entrega, el orden ni la protección contra duplicados. Pero es más rápido y flexible.

Uno podría verse tentado a combinar ambos protocolos ya que en el estado del juego, solo los datos más recientes son relevantes. Pero para otros tipos de datos, como una secuencia de comandos enviados de una máquina a otra, la confiabilidad y el orden pueden ser muy importantes. La tentación entonces

es usar UDP para los estados del jugador, y TCP para los datos confiables y ordenados. En primera instancia esto parece una buena idea. El problema es que ambos, TCP y UDP, están construidos sobre el protocolo IP, los paquetes subyacentes enviados por cada protocolo se afectan mutuamente. La forma en que se afectan es bastante compleja y está relacionada con cómo TCP maneja la confiabilidad y el control de flujo, pero fundamentalmente TCP tiende a inducir la pérdida de paquetes UDP. (Hori, 1997).

Por lo tanto lo recomendable es usar solo UDP para implementar el netcode, si es necesario comunicarse con alguna API REST, unas pocas conexiones TCP no deberían afectar el juego, pero no hay que usar ambos para el netcode del juego. Si necesitamos algunas de las características de TCP como un protocolo de conexión, o que tenga fiabilidad y verificación de errores, estas deberían ser implementadas por el desarrollador sobre el protocolo UDP.(Fiedler, 2008)

4.2. Modelos

A continuación vamos a repasar los modelos de Gaming Networking existentes y exponer cuándo usarlos y cuándo no, sus ventajas y desventajas y por ultimo nombrar algunos ejemplos de juegos que los utilizan.

4.2.1. P2P Deterministic Lockstep

En este modelo de red, los jugadores se conectan directamente entre sí en un esquema P2P (peer-to-peer) formando una red con topología de malla y envían inputs o comandos entre ellos. Los pares solo avanzan la simulación cuando han recibido las entradas/comandos de todos los jugadores para el cuadro actual, de ahí el origen del término "lockstep". Esta espera introduce una latencia inevitable, igual a la del jugador con más lag. Para que este enfoque funcione, el juego debe ser completamente determinista para que no haya divergencias entre las simulaciones.(Fiedler, 2024)

Ventajas:

- No se necesita escribir mucho netcode. "Simplemente funciona".
- Solo se deben enviar entradas o comandos, por lo que el uso del ancho de banda es muy bajo. Por eso era el modelo utilizado en los 90s cuando se usaban modems de 28.8 Kbps.
- No se necesitan servidores, lo que significa que los juegos no tienen costo de infraestructura.
- Se puede grabar el flujo de entradas o comandos de un juego y reproducirlo por completo.

Desventajas:

- Hacer que el juego sea determinista puede ser increíblemente difícil.
- El juego se siente realmente mal cuando uno de los jugadores tiene mucho lag.
- Depurar las causas de la no determinación puede ser extremadamente frustrante.
- Es difícil/imposible lograr una determinación perfecta entre diferentes compiladores y plataformas.
- Se requiere NAT punch-through (Johnston, 2014) para que los jugadores detrás de una NAT puedan conectarse, lo cual puede no ser posible entre todos los jugadores, dependiendo de si están detrás de una NAT más restrictiva o no. Por lo general, esto termina utilizando un sistema de retransmisión como alternativa cuando la NAT punch-through no funciona.
- Todos los jugadores pueden ver las direcciones IP públicas de todos los demás jugadores con los que están jugando, lo cual es un potencial problema de seguridad ya que esto aumenta el riesgo de ser víctimas de ataques DDoS.
- La capacidad del desarrollador para implementar mecánicas contra trampas y mantener el juego limpio es limitada.
- Los lag switches (dispositivos o software para causar retraso) pueden usarse para retrasar a otros jugadores en el juego indefinidamente.
- Solo es realmente adecuado para juegos con un número reducido de jugadores (por ejemplo, de 4 a 8 jugadores como máximo).

Cuándo considerarlo:

- Cuando realmente no sea posible otra opción. Es un modelo que tiene problemas muy grandes tanto en jugabilidad como en seguridad.

Cuándo evitarlo:

- Siempre.

Ejemplos:

- Age of Empires
- Command and Conquer
- Ultra Street Fighter IV

4.2.2. Deterministic Lockstep with Relay Server

Similar al modelo anterior, pero en lugar de conectar a los jugadores de manera P2P, estos intercambian paquetes a través de un servidor de retransmisión. Este servidor de retransmisión tiene un manejo de update rate, en lugar de ser simplemente un reflejo pasivo. Ahora, si un cliente intenta usar un lag switch o manipula demasiado el tiempo de las inputs, esto puede ser detectado y el jugador puede ser expulsado del juego. (Fiedler, 2024)

Ventajas:

- No hay que preocuparse por problemas de NAT.
- Los ataques con lag switches ya no son efectivos.
- Conserva todos los beneficios del lockstep determinista.

Desventajas:

- Hay que esperar al jugador con más lag, o ese jugador pierde sus inputs o es expulsado del juego.

Cuándo considerarlo:

Cuando se trate de un juego de género *RTS* (estrategia en tiempo real) y tenga demasiadas unidades para sincronizarlas de otra manera.

Cuándo evitarlo:

Los jugadores de juegos de lucha esperan rollback netcode. Usar este modelo no los dejaría conformes.

Ejemplos:

- Starcraft II
- Cualquier juego moderno de género *RTS*.

4.2.3. Deterministic Lockstep with Simulation Server

En este modelo el servidor no sirve solo para retransmisión de inputs. Ahora también simula el juego por su cuenta (como si fuera otro jugador, pero invisible). Esto significa que, además de enviar los inputs de los jugadores a los demás, el servidor corre el mismo juego que los jugadores y se asegura de que todo esté sincronizado. Así, las cosas importantes del juego, como otorgar puntos, ítems o progreso en el metajuego (algo fuera del juego principal, como desbloquear niveles o recompensas),se hacen desde el servidor donde los jugadores no pueden hacer trampas. (Fiedler, 2024)

Ventajas:

- Conserva todos los beneficios del lockstep determinista, y ahora se tiene una base segura para otorgar ítems y puntajes a los jugadores.
- Se puede implementar un verdadero metajuego y progresión, los jugadores no pueden hackearlo fácilmente.

Desventajas:

- Hay que pagar por servidores dedicados para ejecutarse junto a los jugadores, lo que aumenta significativamente los costos de infraestructura.
- Solo es viable para juegos con pocos jugadores.

Cuándo considerarlo:

- Si se necesita alta seguridad y es crítico que los jugadores no puedan hacer trampa manipulando el código del juego o su memoria, por ejemplo, en juegos monetizados con micro transacciones donde se otorgan recompensas importantes como ítems, monedas o puntos para la progresión.
- Juegos con pocos jugadores por partida.

Cuándo evitarlo:

- Juegos con muchos jugadores por partida o que manejan demasiadas unidades.
- Si se tiene un presupuesto limitado.
- Cuando el juego es más simple y menos competitivo, este nivel de seguridad puede ser innecesario.

Ejemplos:

- Clash of Clans
- Clash Royale

4.2.4. Distributed Authority

En este modelo, cada jugador maneja su propio personaje y ciertos objetos en su propia máquina, como si fuera "el jefe" de esos elementos (por ejemplo, su personaje, un vehículo o algún objeto que interactúa). Otros elementos, como enemigos controlados por IA, también se dividen entre las máquinas de los jugadores, de modo que todos contribuyen a simular el mundo del juego. De forma básica, el retraso de inputs se elimina al simular cada jugador localmente en su propia máquina, donde cada cliente tiene "autoridad" (actúa como servidor)

sobre su propio personaje. La idea principal es repartir la carga de trabajo (simulación y cálculos) entre las máquinas de los jugadores en lugar de depender de un servidor central.(Fiedler, 2024)

Ventajas:

- No se requiere determinismo.
- No necesita servidores dedicados.
- Distribuir la simulación permite distribuir el costo de simular el mundo entre varias máquinas, lo que hace posible mundos más grandes.
- Si no tiene servidores dedicados, la alternativa es que un jugador aloje la partida (y pueda hacer trampas) o que todos los jugadores puedan hacer trampas.

Desventajas:

- Es inseguro ya que confía en el cliente.
- Implementar la unión tardía de otro jugador puede ser difícil, y aunque funcione, no siempre será confiable.
- Las cosas se ejecutan en máquinas diferentes. La programación distribuida es más difícil que ejecutar todo en un servidor.
- Cada jugador simula algunos objetos y necesita enviarlos a los demás. Esto puede ser un problema en conexiones asimétricas (más descarga que subida). A medida que aumenta el número de jugadores, el costo crece de manera **O(n*m)**, donde n es el número de jugadores y m el número de objetos. Esto se puede mitigar parcialmente sincronizando objetos cercanos con mayor frecuencia que los lejanos o usando una simulación física "casi determinista" que permita actualizarse con menor frecuencia y aún verse de manera correcta visualmente.

Cuándo considerarlo:

- Cuando se convierte un juego de un solo jugador a multijugador y no queremos reescribir todo el código para la red.
- Cuando no hay presupuesto o no queremos pagar servidores dedicados.
- Si el juego tiene un mundo abierto grande.
- Si el juego no es determinista y reescribirlo para que lo sea no es factible.
- Si el juego consume muchos recursos de CPU y no es posible rehacer y correr múltiples ciclos cada frame porque usaría demasiado CPU.
- Si el juego es cooperativo o poco competitivo.

Cuándo evitarlo:

- Cuando el juego es muy competitivo.

Ejemplos:

- *Journey*
- *God of War: Ascension*
- *Mercenaries 2*
- *GTA: Online*
- *Dark Souls*
- *Uncharted*
- *Destiny*

4.2.5. Pure Client/Server

En este modelo, el juego se ejecuta exclusivamente en el servidor. Cada cliente solo envía sus inputs al servidor y recibe de vuelta una serie de snapshots que representan una versión interpolada del estado visual del mundo en un tiempo determinado. El cliente se encarga de renderizar este estado.(Fiedler, 2024)

Ventajas:

- El cliente solo envía inputs, lo cual reduce los datos enviados al servidor y el uso de banda ancha.
- Bajo uso de CPU en el cliente ya que no ejecuta simulaciones ni cálculos físicos, todo el poder de procesamiento se usa para renderizar gráficos.
- Solo se sincronizan los componentes visuales de los objetos (posición, orientación, animación, etc.) y no su estado interno más complejo.
- Todos los objetos están en una misma línea de tiempo, lo que facilita interacciones consistentes entre jugadores y el mundo.

Desventajas

- Existe un retraso entre que el cliente envía sus inputs y ve la acción reflejada, este retraso es como mínimo igual al lag entre cliente y servidor.
- Es complicado lograr que el cliente envíe inputs adelantados al servidor para que este los use justo a tiempo, sin agregar demasiado retraso adicional. Lo ideal es que estos lleguen cuando el servidor los necesita, si llegan muy adelantados habrá latencia agregada. Esta sincronización es complicada de manejar debido a las condiciones cambiantes de internet.

- Dependiendo donde se encuentre el servidor y el routing que maneje el proveedor de internet, la latencia puede ser mayor a la normal.

Cuándo considerarlo:

- Si el juego no es muy sensible a la latencia o es aceptable una pequeña tasa fija de latencia.

Cuándo evitarlo:

- Cuando no hay presupuesto o no queremos pagar servidores dedicados.

Ejemplo:

- Quake

4.2.6. Client Side Prediction and Lag Compensation

Este modelo mejora y complejiza el modelo anterior. Ahora se adapta para que cada jugador opere en su propia línea de tiempo. En lugar de que todo el mundo del juego avance en pasos de tiempo fijos, el servidor actualiza a los jugadores individualmente cuando recibe sus inputs más el tiempo delta correspondiente desde sus clientes, siendo el tiempo delta el tiempo transcurrido entre que el cliente envio el input y el servidor lo recibe y procesa.

En el cliente, los objetos remotos (controlados por otros jugadores) se interpolan para suavizar su movimiento y a veces, aunque no es lo ideal, se extrapolan (se predicen). Sin embargo, los objetos controlados localmente por el jugador tienen un tratamiento especial. Estos se simulan completamente en el cliente utilizando los inputs locales del jugador, lo que elimina la sensación de latencia en acciones como moverse o disparar.

Para mantener la autoridad del servidor, este envía correcciones periódicas del estado profundo al cliente. Estas correcciones incluyen información como el inventario, la munición restante, el arma equipada y los retrasos entre disparos. Una vez que el cliente recibe esta información, aplica las correcciones y resimula el estado del juego desde el pasado hasta el presente, ajustando cualquier discrepancia. Este enfoque se conoce como **client side prediction** (predicción del cliente).

El manejo de disparos y daño se realiza exclusivamente en el servidor. En el cliente, el disparo de armas es solo visual y sirve para dar una respuesta inmediata al jugador. Gracias a que el cliente y el servidor comparten una sincronización de tiempo, el servidor puede reconstruir el estado del mundo desde la perspectiva del cliente en el momento en que este disparó. Esto se conoce como **lag compensation** (compensación de latencia) y permite que los impactos se registren con precisión desde el punto de vista del jugador que disparó.

Para implementar la compensación de latencia, el servidor utiliza un búfer circular que almacena el estado visual del juego durante aproximadamente un

segundo. Este búfer contiene información como la posición, orientación y animaciones de los objetos, y el servidor interpola estos datos para coincidir con lo que el cliente ve. De esta forma, los disparos pueden impactar con precisión, incluso si el objetivo estaba en movimiento. Sin embargo, desde la perspectiva del jugador impactado, esto puede parecer que fue alcanzado mientras ya estaba detrás de una cobertura, lo que refleja las limitaciones del sistema. (Fiedler, 2024)

Ventajas:

- Buena calidad de partidas entre jugadores.
- Bajo uso de ancho de banda desde el cliente al servidor ya que solo se envían inputs.
- Casi todo corre en el servidor, liberando CPU en el cliente para el renderizado.

Desventajas:

- Los servidores dedicados son costosos.
- Requiere reestructurar el código del juego para compartir lógica entre cliente y servidor, y que funcione con predicción y compensación de latencia.
- La compensación de latencia puede causar situaciones confusas, como recibir disparos "detrás de la cobertura".
- Es vulnerable a trampas.
- No es adecuado para juegos con interacciones cuerpo a cuerpo complejas entre jugadores.
- Si el mundo del juego es muy dinámico o destructible, reconstruir estados para la compensación de latencia puede ser prohibitivo.

Cuándo considerarlo:

- Cuando estamos creando un FPS (First Person Shooter) competitivo de primer nivel.

Cuándo evitarlo:

- Cuando no hay presupuesto o no queremos pagar servidores dedicados.
- Cuando el juego incluye mecánicas cuerpo a cuerpo muy complejas.

Ejemplos:

- Counter-Strike
- Call of Duty
- Titanfall 1 y 2
- Apex Legends
- Valorant

4.2.7. Client side prediction with Client Simulation in the Remote View of Objects

En lugar de que los objetos remotos se interpolen en el servidor, una parte de la simulación de esos objetos se ejecuta en el cliente. Esto permite que los objetos sigan moviéndose y actualizándose en el cliente, incluso entre las actualizaciones que se reciben desde la red.

Las actualizaciones de los objetos se priorizan de manera que no todos los objetos se envían en cada paquete de datos, y se actualizan a diferentes frecuencias. Por lo general, los objetos más cercanos al jugador reciben actualizaciones más frecuentes. Para gestionar esto, cada objeto tiene un acumulador de prioridad que aumenta cada vez que no se envía una actualización para ese objeto. Este acumulador se ajusta con un multiplicador que puede depender de diversos factores como el tipo de objeto, su estado o la distancia desde el jugador.

Dado que la vista del cliente sobre los jugadores remotos no es tan precisa como en juegos como *Counter Strike* , los disparos a menudo deben adelantarse para compensar la latencia (esto se llama "lead shots"). En lugar de esperar a que la actualización del servidor llegue, el cliente puede "adelantar" el disparo para tratar de acertar donde el jugador enemigo debería estar, en lugar de donde está exactamente en ese momento. Sin embargo, el cliente solo recibe crédito por los disparos que, según el servidor, son "creíbles", es decir, si el disparo se encuentra dentro de un margen de tolerancia aceptable con respecto a la posición en el servidor. Sin embargo, esta solución no es tan precisa ni robusta como la compensación de latencia utilizada en el modelo que mencionamos anteriormente.(Fiedler, 2024)

Ventajas:

- Menor latencia en la visualización de objetos remotos en el cliente, ya que no hay búfer para la interpolación.
- El uso de ancho de banda se puede escalar fácilmente ajustando el tamaño máximo del paquete enviado. El acumulador de prioridad maneja el resto.

Desventajas:

- Los jugadores disparan a una predicción.
- El servidor no puede verificar completamente que el cliente acertó con precisión.
- Mayor costo de CPU en el cliente, ya que ejecuta parte de la simulación y detección de colisiones para objetos remotos, en lugar de solo interpolarlos.
- Puede haber una complejidad significativa, ya que se pueden recibir actualizaciones para un objeto en un paquete sin recibir la actualización de otro objeto del que depende.

Cuándo considerarlo:

- Si se usa Unreal Engine (E. Games, 2024) para desarrollar un FPS competitivo más casual o un shooter cooperativo ya que este es modelo de red predeterminado de este famoso motor.
- Cuando el juego tiene un mundo grande con muchos jugadores y objetos, y queremos priorizar actualizaciones más frecuentes para los objetos cercanos.
- Si los objetos tienen un movimiento altamente predecible, como proyectiles, aviones o cuerpos rígidos que pueden extrapolarse entre las actualizaciones de red.

Ejemplos:

- Unreal Tournament
- Fortnite
- Halo
- Battlefield

4.3. Revisión de plataformas de emulación existentes

Dado que existe un océano de emuladores independientes tanto de NES como de otros sistemas y no es humanamente posible estudiarlos todos vamos a limitarnos a tomar algunos emuladores de NES y plataformas más conocidas y revisar que tipo de online implementan (si lo hacen).

4.3.1. MAME (Multiple Arcade Machine Emulator)

MAME se lanzó por primera vez en 1997 y es unas de las plataformas (si no la más famosa) de emulación, su propósito inicial era la de documentar y preservar videojuegos y hardware de sistemas arcade por medio de la emulación, aunque posteriormente su alcance se expandió a consolas, computadoras personales antiguas, calculadoras, etc. Además de ser un recurso para la preservación histórica, MAME también se utiliza para investigar cómo funcionaban estas máquinas y garantizar que no se pierdan con el tiempo, al proporcionar una plataforma para experimentar sus contenidos de manera educativa. Dado que MAME apunta a la preservación su enfoque es el de emular el sistema de la manera más precisa que razonablemente se pueda hacer.(MAME, 2024) Es posible jugar mame online por medio de un middleware llamado Kaillera lanzado en el año 2001 el cual usa un modelo cliente-servidor donde un jugador será el servidor y el resto los clientes con soporte hasta 8 jugadores, los inputs se manejan con Delay Based Netcode. (Thibault, 2004)

4.3.2. Mesen

Mesen es un emulador de código abierto para NES y Famicom, diseñado con un enfoque en la precisión y funcionalidades avanzada. Es compatible con Windows y Linux y está escrito en C++ y C#. Este es conocido por ser uno de los emuladores más precisos de NES y también por tener unos de los debuggers más completos para el sistema lo cual lo hace una excelente herramienta de desarrollo. Mesen cuenta con multijugador online aunque en su documentación no se especifica que método usa, solo podemos deducir por lo presente en la documentación que usa un modelo cliente-servidor. (SourMesen, 2020)

4.3.3. Mednafen

Mednafen (abreviatura de "My Emulator Doesn't Need A Frickin' Excellent Name") es un emulador de múltiples plataformas que admite una amplia variedad de consolas, como NES, SNES, Genesis, PlayStation, Neo Geo y más. Es conocido por su precisión de emulación y su enfoque en la calidad sobre la simplicidad en la interfaz. Al igual que en Mesen, en la documentación de Mednafen no se menciona como se implementa su Netplay pero si se deduce que utiliza un modelo cliente-servidor. (MednafenTeam, 2024)

4.3.4. Nintendo Switch

Nintendo ofrece acceso a un catalogo de juegos de NES con la posibilidad de multijugador online a los usuarios que cuentan con una suscripción de pago a Nintendo Switch Online, desgraciadamente no esta especificado que método se utiliza, lo único que se menciona al respecto es una funcionalidad "*Low Latency Mode*" que reduce el input delay al jugar online y solo puede activarse luego de iniciar el juego. (Nintendo, s.f.)

4.3.5. RetroArch

RetroArch es un frontend lanzado en mayo de 2010 para la plataforma libretro, diseñado para unificar la emulación y facilitar el acceso a múltiples sistemas desde una sola interfaz. Su propósito es actuar como un punto de encuentro para emuladores, motores de juegos y aplicaciones multimedia, proporcionando una experiencia personalizable y versátil. Es compatible con una amplia gama de sistemas operativos, como Windows, macOS, Linux, Android, iOS e incluso consolas como PlayStation y Xbox. Lo que hace único a RetroArch es su capacidad para cargar "cores", que son módulos que emulan consolas, arcades y otros dispositivos. Cada core se basa en libretro, una API que estandariza la emulación y permite que los desarrolladores porten su software a múltiples plataformas. Además, RetroArch ofrece características avanzadas como shaders, rebobinado, netplay, soporte para runahead y guardado/cargado de estados, etc.

El netcode de RetroArch está basado en el "replay" y proporciona juego online sobre redes no fiables sin latencia de entrada en la configuración predeterminada. El netplay admite hasta 16 jugadores y muchos espectadores, garantizando su funcionamiento con sincronización perfecta, para esto requiere:

- Que el núcleo es determinista,
- Los únicos dispositivos de entrada con los que el núcleo interactúa son el gamepad y los sticks analógicos.
- Tanto el núcleo como el contenido cargado son idénticos en el host y el cliente.

Se espera que los núcleos soporten serialización para un comportamiento adecuado de netplay, en caso de no hacerlo la experiencia será menos fluida.

El netplay en RetroArch funciona esperando que la entrada llegue con retraso desde la red, luego retrocediendo y reproduciendo con la entrada retrasada para obtener un estado consistente. En un momento dado, todos los clientes de netplay pueden estar en estados inconsistentes, pero una vez que reciben los datos retrasados de los demás, retroceden invisiblemente al último momento en que estuvieron consistentes, reproducen con la nueva entrada y alcanzan un nuevo estado, que en teoría estará más cerca del estado "correcto" que el estado anterior. Para implementar esto, Retroarch usa el modelo cliente - servidor bajo el protocolo TCP para asegurar la fiabilidad y el orden correcto de los paquetes. (Libretro, 2024)

Capítulo 5

Introduccion a la técnica Rollback Netcode

En este capítulo presentamos la técnica Rollback Netcode, explicando su historia, funcionamiento y los beneficios que aporta en comparación con métodos tradicionales como el Delay Input. También hablamos sobre los requisitos para implementar Rollback en emuladores.

5.1. Un Poco de Historia

El concepto de rollback surgió a finales de la década del 2000, impulsado por **Tony Cannon**, entusiasta de los videojuegos de género lucha graduado en Ciencias de la Computacion en la Universidad de Standford en 1995, cofundador del evento **Evo (Evolution Championship Series)** (EVO, 2022) y creador del middleware **GGPO (Good Game Peace Out)**(Cannon, 2024). Cannon estaba preocupado por la desaparición de las salas arcade en los años 90, que eran el centro de la comunidad de jugadores competitivos. Su interés por preservar esa cultura se intensificó cuando Capcom lanzó una versión online de *Street Fighter II: Hyper Fighting* en 2005, que prometía revivir la escena competitiva pero resultó ser un fracaso debido a su netcode basado en input lag y los glitches. Para los jugadores casuales quizás era aceptable, pero para los más dedicados, la experiencia era "injugable". Esto llevó a Cannon a pensar en una solución específica para juegos donde el tiempo y la precisión son críticos, como los de lucha. Así nació la idea del Rollback Netcode, una técnica que prioriza la fluidez de los inputs locales del jugador al predecir los movimientos del oponente y corregir posibles desajustes de manera retroactiva. Aunque puede haber pequeños glitches visuales, el sistema garantiza que el tiempo de respuesta del jugador sea consistente, lo que es crucial para este género. Después de varios meses de trabajo, Cannon lanzó en 2009 **GGPO**, una herramienta que marcó un antes y un después en la forma en que los juegos de lucha multijugador manejan la latencia. Este desarrollo no solo ayudó a revitalizar la comunidad de juegos de

lucha online, sino que también se convirtió en un estándar que hoy en día influye en los desarrollos de grandes estudios y franquicias de renombre.

5.2. El método Rollback

El método más común para implementar el Netplay de un juego 1 VS 1 como puede ser un juego de género lucha consiste en ejecutar una instancia separada del juego en cada dispositivo y mantenerlas sincronizadas asegurándose de que cada instancia reciba exactamente los mismos inputs (ver Figura 5.1). Si los estados del juego se determinan únicamente a partir de los inputs, reproducir los mismos inputs en ambos sistemas producirá el mismo resultado. Este enfoque presenta numerosas ventajas: las computadoras son, por naturaleza, deterministas y, asumiendo que el juego no dependa de elementos externos que afecten este determinismo, los ingenieros y diseñadores pueden trabajar en la jugabilidad sin preocuparse por los detalles relacionados con la red. Esto permite agregar soporte multijugador online de manera retroactiva a juegos que ya han sido lanzados. Además, este método no requiere componentes basados en servidores, salvo para la funcionalidad básica de matchmaking, que generalmente es proporcionada por el publicador del juego .

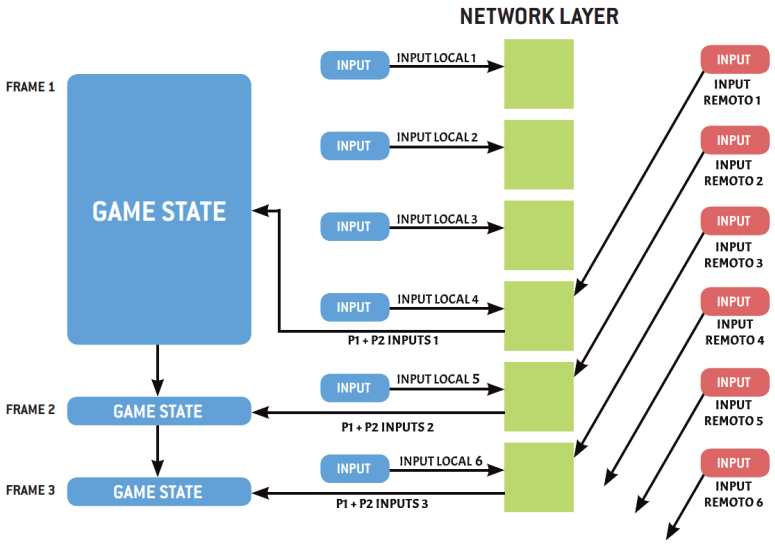


Figura 5.1: Diagrama Delay Based Netcode (Cannon, 2024)

Desafortunadamente, este método tiene **un problema importante**: estos juegos suelen estar diseñados para tomar los inputs antes de cada actualización

del game state. Un frame del juego no puede ejecutarse hasta que se hayan recibido todos los inputs de los jugadores remotos. En la práctica, esto se manifiesta como un delay en los inputs equivalente al tiempo que toma enviar un paquete de un sistema al otro. En resumen, los juegos no son responsivos. La respuesta instantánea para la que el juego fue diseñado se ve reemplazada por controles lentos y poco precisos que en latencias mayores al equivalente de 2 o 3 frames (alrededor de 65 milisegundos) arruinan la experiencia del jugador. En muchos casos, el retraso introducido por la capa de red puede cambiar por completo la forma de jugar el juego. En juegos de lucha como *Street Fighter*, se le conocen como "tácticas de lag" el aprovechar el conocimiento de que, para cuando nuestro oponente vea nuestro movimiento, será demasiado tarde para responder y arruinan la competitividad del juego.

En el método Rollback se utiliza una simulación, es decir una ejecución especulativa para seguir corriendo el juego y eliminar el retraso percibido en los controles de cada jugador local, en lugar de esperar a que todos los inputs lleguen a la instancia de un jugador antes de ejecutar un frame (ver Figura 5.2). Para esto se adivina lo que harán los jugadores remotos basándose en sus acciones anteriores. Esto elimina el lag que experimenta el jugador local con el método tradicional de input delay y el personaje del jugador local responde tan rápido como si estuviera jugando offline. Aunque las acciones de los otros jugadores no pueden conocerse hasta que lleguen sus inputs, el método de predicción que usa el rollback asegura que la simulación del juego sea correcta la mayor parte del tiempo.

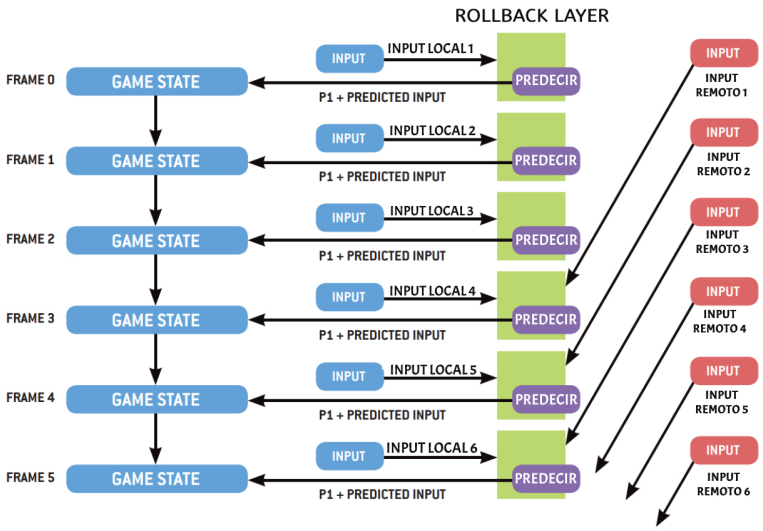


Figura 5.2: Diagrama Rollback (Cannon, 2024)

Cuando se reciben los inputs remotos, estos se comparan con los predichos. Si se detecta una discrepancia, la simulación retrocede al primer frame incorrecto, se corrigen los inputs y se avanza nuevamente hasta el frame actual utilizando los nuevos valores (ver Figura 5.3).

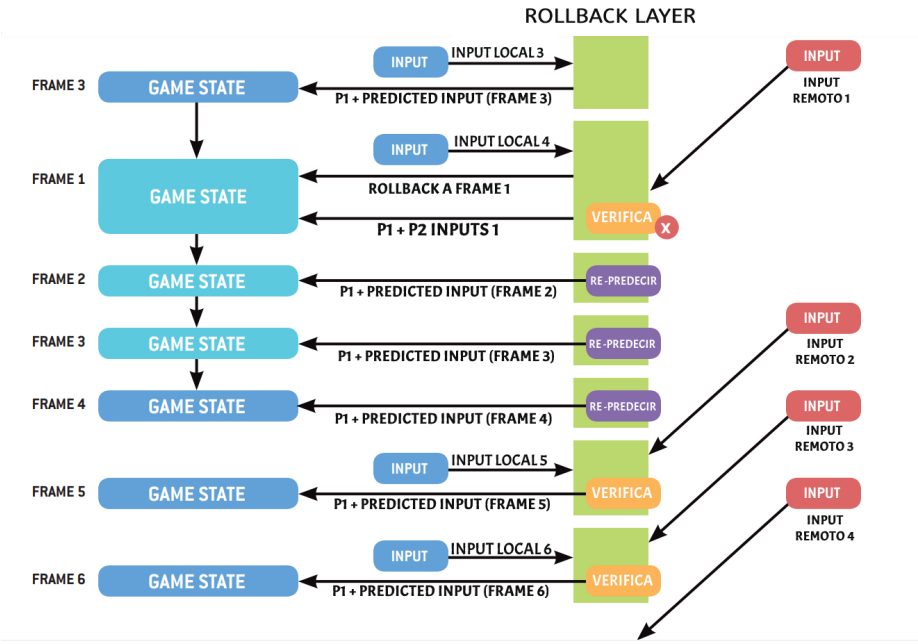


Figura 5.3: Diagrama Rollback con predicción incorrecta (Cannon, 2024)

Las acciones locales realizadas por el jugador local siempre ocurren al instante y siempre son correctas. Esto es muy conveniente, ya que la capacidad de respuesta del personaje al control suele ser el aspecto más importante en la experiencia de juego online. Además, cualquier efecto a largo plazo causado por inputs de jugadores remotos que ya se hayan recibido también será correcto. Volviendo a usar de ejemplo a *Street Fighter* si nuestro oponente lanzó un proyectil hace unos frames, el comportamiento de ese proyectil es completamente determinista y no puede ser afectado por inputs futuros que lleguen remotamente. Por lo tanto, el proyectil se moverá correctamente tan pronto como la simulación local determine que efectivamente nuestro oponente lanzó el proyectil. Esto hace que el tiempo y la experiencia de enfrentar un proyectil online sean idénticos a la experiencia fuera de línea, lo cual es importante, ya que lidiar con proyectiles es una parte fundamental de *Street Fighter*. De manera similar ocurre con otros movimientos como los saltos, ya que generalmente los jugadores no pueden cambiar el arco de sus saltos después de iniciarlos. Por lo que lidiar con nuestro oponente descendiendo de un salto con un golpe an-

tiaéreo bien colocado, es idéntico tanto online como jugando localmente. Dicho más abstractamente, cuando nuestro oponente introduce ciertos inputs el game state suele entrar en un estado constante durante cierta ventana de frames que no serán afectados por los inputs remotos que vayan llegando, se puede ver un ejemplo gráfico de esto en la Figura 5.4 donde una vez presionado el botón de patada fuerte el juego ejecutara la animación del personaje que dura 32 frames (mas de medio segundo) sin poder alterarlo.

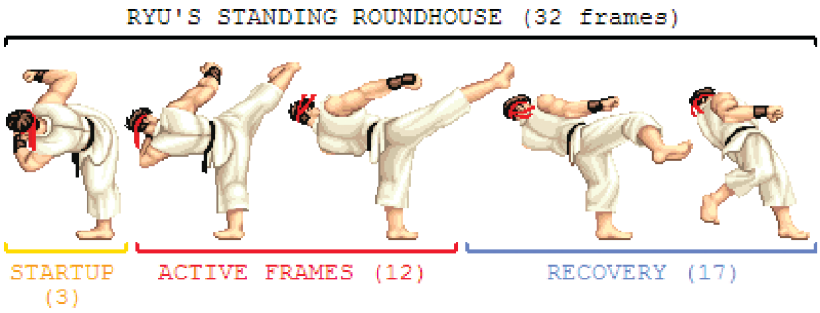


Figura 5.4: Ejemplo de animación larga e inalterable una vez presionado un botón (Cannon, 2012)

La latencia se oculta en la ventana de tiempo entre que nuestro oponente inicia una acción y cuando nuestra simulación se da cuenta de que se realizó dicha acción. El tiempo perdido en esa ventana es omitido por la simulación. Por ejemplo, supongamos que hay 45 ms de latencia con nuestro oponente, cuando este ejecuta un movimiento como el de la Figura 5.4, su simulación procesará los inputs de inmediato. Para él, el movimiento ocurre al instante, ya que los inputs locales se envían a la simulación de inmediato y siempre son correctos. Sin embargo, nuestra simulación no notará que nuestro oponente realizó un movimiento hasta que llegue un paquete de la red con ese input 45 ms después. Cuando esto ocurra, el juego deberá rebobinar 45 ms (unos 3 frames), corregir el input de nuestro oponente y avanzar la simulación 45 ms nuevamente al tiempo actual. Como resultado, no veremos los primeros 45 ms de animación de lo que hizo nuestro oponente, usando como ejemplo la Figura 5.4, no veremos los 3 frames de Startup sino que directamente veremos el primer frame de Active Frames.

Esto no parece ser lo ideal, pero la alternativa sería retrasar toda la simulación 45 ms, incluidos los inputs locales. En la práctica, perder esos 45 ms de animación suele dar como resultado una experiencia de usuario mucho mejor. Esto se debe en parte a la capacidad de respuesta inmediata de las acciones locales, pero también porque la mayoría de las veces esos 45 ms no son tan importantes ya que solo se esta omitiendo el Startup. En caso de que la latencia sea alta y los saltos visuales notorios, es posible combinar este método con

Delay Input, si agregamos un delay de uno o dos frames a nuestro input, esto permitiría que se manden inputs "por adelantado" aunque el juego no vaya a responder a ellos todavía, lo cual mitiga un poco la latencia y los saltos visuales sacrificando un poco de responsividad, como si se estuvieran restando 16 ms de latencia por cada frame de input delay y reduciendo la distancia de los rollbacks.

5.2.1. Modo de predicción

Un buen algoritmo de predicción es indispensable para minimizar los saltos visuales que pueden ocurrir durante un rollback. El trabajo del algoritmo de predicción es anticipar los inputs del jugador que llegan desde la red en el frame $N + \frac{L}{F}$, dados los inputs de todos los frames previos $1...N$, donde N es el último input recibido del jugador remoto, L es el tiempo de latencia, y F es la frecuencia con la que el juego actualiza su estado ($\frac{1}{60}$ ms normalmente).

La calidad del algoritmo de predicción depende de la frecuencia y la gravedad de los errores de renderizado causados por una predicción fallida. Por ejemplo, en un juego de *Pong*, un error de predicción hará que la posición de la paleta del oponente salte a una nueva ubicación en la pantalla. Equivocarse en la posición de la paleta es importante, pero un algoritmo que siempre esté equivocado pero que solo se desvíe por unos pocos píxeles es preferible a uno que solo se equivoque el 10 % del tiempo pero que haga que la paleta del oponente salte un cuarto de la longitud de la pantalla.

Naturalmente, el mejor algoritmo de predicción posible es específico del juego, y probablemente también del jugador.

El algoritmo de predicción que se usa normalmente asume que los inputs futuros serán idénticos a los inputs más recientes recibidos del jugador remoto. Esto limita el número de errores de predicción al número de veces que cambia el estado de los inputs en cada intervalo. Aunque es simple, esto ha demostrado ser muy eficaz para evitar los efectos visuales más bruscos. Por ejemplo, un personaje que se mueve hacia adelante o hacia atrás hará que la pantalla se desplace hacia la izquierda o hacia la derecha. Equivocarse en esa posición provocará que toda la pantalla salte hacia la izquierda o la derecha durante un rollback. Al asumir que el joystick se mantiene en una dirección, aseguramos que la pantalla se desplace de manera fluida en el caso más común: correr hacia la derecha. El peor caso (cambiar rápidamente el joystick de izquierda a derecha) rara vez ocurre en el juego real.

Un método alternativo podría ser usar el estado del juego de los frames previos para proporcionar un mejor algoritmo de predicción. Algo similar pero más sofisticado probó Anton Ehlert en el articulo (Ehlert, 2021) utilizando una red neuronal entrenada con un dataset de miles de secuencias de inputs grabados de jugadores, sin embargo los resultados mostraron que aunque podría haber una pequeña mejora, el aumento de demanda en cuanto a poder de computo y memoria no lo hacen conveniente. Por lo tanto en términos de método de predicción, se sigue usando el mismo y simple método hasta día de hoy.

5.2.2. Sincronización

Para garantizar que ambos jugadores experimenten la misma secuencia de eventos en el juego, es necesario:

- **Mantener el Game State en ambas instancias**:

Es vital que el netcode tenga formas de lidiar con paquetes desordenados o paquetes que se pierden. Si un solo input se perdiera, el Game State de ambas instancias podría divergir.

- **Equidad entre jugadores**:

No solo es importante asegurar que los inputs lleguen correctamente para mantener los Game States iguales, también es crucial que ambos jugadores estén sincronizados en el mismo frame. Por diversos motivos, no es raro que una instancia del juego se quede "un poco atrás". Si esto ocurre y no se corrige, el juego funcionará correctamente para el jugador atrasado, pero habrá muchos de rollbacks para el jugador adelantado resultando en una partida caótica. Por ejemplo, imaginemos que el Jugador A está en el frame 15, mientras que el Jugador B está en el frame 20 como se muestra en la Figura 5.5. Como el Jugador B ya alcanzó el frame 20, esté le envió a A los frames que necesita y continúa enviando frames adelantados. En la instancia del Jugador A, no será necesario hacer predicciones. Sin embargo, el Jugador B ha tenido que predecir continuamente sus frames, llegando a ejecutar más de 5 frames especulados. Cada vez que un paquete del Jugador A llega, si la predicción no fue precisa (a medida que se predicen más frames, aumenta la probabilidad de errores), B tendrá que hacer rollbacks de 5 frames. Esto será claramente visible y resultará en una experiencia desagradable.

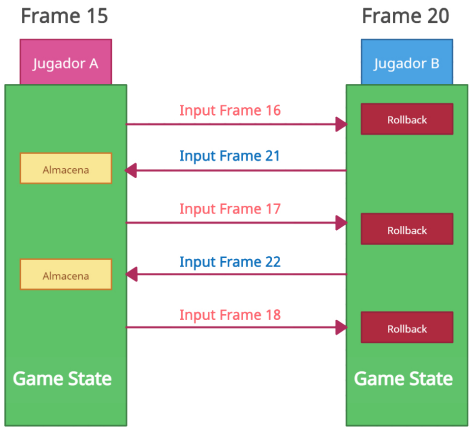


Figura 5.5: Ejemplo de Rift. Fuente propia

Este problema es conocido como *The Rift*, y es un problema fácil de pasar por alto si se está dando los primeros pasos con esta metodología y no se testea exhaustivamente en redes con condiciones adversas, de hecho, este fue un enorme problema en el juego *Street Fighter V* lanzado en el año 2016, donde la saga principal estrenaba su primera entrega usando Rollback Netcode (Pusch, 2019) (ya se había hecho una primera aproximación en el Spin-off Street Fighter X Tekken), el juego tenia continuas quejas de gente que decía que desde su lado el juego era injugable, mientras que desde el otro lado usando conexiones mucho menos fiables como conexiones inalámbricas y jugando en dispositivos más modestos y propensos a retrasarse, el juego funcionaba "bien". Algunos entusiastas y el propio Cannon estudiaron el caso desde fuera y todos concluyeron que el problema era que los desarrolladores nipones no manejaban el Rift, afectando la reputación del método.

A modo de ejemplo para ver de como se pueden solucionar estos problemas usaremos al propio **GGPO** (Cannon, 2024) desarrollado por Canon, GGPO utiliza un protocolo simple y eficiente para sincronizar los inputs entre jugadores en una sesión, como podemos ver en el Listing 5.1 el encabezado de cada paquete contiene un identificador de sesión específico de 16 bits, seguido por el tipo de carga útil. Los tres tipos principales de paquetes son `quality_report`, `quality_reply` e `input`.

Listing 5.1: Descripción simplificada del payload de los paquetes UDP en GGPO.

```
struct packet {
  struct {
    uint16 magic; // Id de la sesion
    uint8 type; // Tipo de paquete: input, quality report o quality reply
  } hdr;
  union {
    struct {
      uint32 start_frame;
      uint32 ack_frame;
      uint16 num_bits;
      uint8 bits[MAX_COMPRESSED_BITS];
    } input;
    struct {
      int8 frame_advantage;
      uint32 ping;
    } quality_report;
    struct {
      uint32 pong;
    } quality_reply;
  } u;
};
```

Cada payload de tipo `input` se envía en cada frame. Los inputs se comprimen utilizando una máquina de estados para alternar botones virtuales. La

máquina comienza con un vector de inputs vacío en el frame 0 y para cada frame se codifican en el array `bits` siguiendo este esquema:

- `1 + <5-bits de n>`: Indica que el botón identificado por el número `n` ha cambiado de estado en comparación con el input anterior.
- `0`: Indica el final de los cambios de estado en los botones.

Por ejemplo supongamos que solo el Boton 1 estaba presionado en el frame anterior y en el actual se suelta el Boton 1 y se presiona el Boton 2 y el Boton 3, bits tendra la siguiente forma

```
100001xx | 100010xx | 100011xx | 0
```

y si en el siguiente frame se mantienen apretados los mismos botones solo se envía el valor 0.

Esta codificación permite que la mayoría de los inputs dentro de un flujo se compriman en un solo bit (`0`). Incluso enviando inputs distintos en cada frame, el flujo se comprime a aproximadamente 300 bits por segundo. Este tamaño reducido permite incluir todos los inputs desde el último frame confirmado en cada paquete, lo que simplifica el manejo de paquetes perdidos o desordenados.

GGPO mantiene un historial de los últimos `x` inputs recibidos de un jugador remoto, donde `x` es una ventana mayor que la barrera de predicción, es decir es más grande que los frames que han sido predichos. Cada vez que se recibe un input de un jugador remoto, se carga del historial el input correspondiente al `start_frame` indicado en el payload y se ejecuta la máquina de estados para recuperar los inputs subsecuentes. Si se obtienen inputs adelantados al frame actual se almacenan en el historial para no tener que predecirlo. Finalmente se actualiza el campo `ack_frame` del siguiente paquete `input`.

GGPO envía periódicamente un paquete de tipo `quality_report` para medir la calidad de la conexión. Este paquete incluye un timestamp generado en la máquina local que se setea en la variable *ping* y una medida de "ventaja de frames" local. Al recibir un paquete `quality_report`, el par envía inmediatamente una respuesta `quality_reply`, copiando el valor de `ping` del reporte en la respuesta. El endpoint original mide el tiempo de ida y vuelta restando el timestamp actual del valor `pong` del último paquete `quality_reply`.

Ya mencionamos el problema conocido como *The rift* y en GGPO se lidia con el con la variable `FRAME_ADVANTAGE`. donde se miden cuantos frames adelantados tiene el jugador local frente a su par. Este dato se calcula utilizando la siguiente fórmula:

$$\text{Frame Advantage} = \left(\text{Last Remote Frame} + \frac{\text{Ping} \cdot \text{Frame Frequency}}{2}\right) - \text{Last Local Frame}$$

Esto significa que estimamos el frame que nuestro oponente está renderizando actualmente sumando la mitad del tiempo de ida y vuelta de los paquetes al

último paquete recibido, y luego restamos el frame que estamos renderizando. El resultado es la cantidad de frames que estamos adelantados a nuestro oponente.

Hay varios factores que pueden afectar el cálculo preciso de la ventaja de frames, como tiempos de transmisión de paquetes asimétricos, problemas intermitentes de conexión, entre otros. Sin embargo, esta fórmula parece funcionar bien en la práctica.

Cada jugador en una sesión de GGPO siempre está al tanto de su propia ventaja de frames local y recibe actualizaciones periódicas sobre el cálculo de la ventaja de fotogramas de su oponente. GGPO intentará mantener el juego "justo" asegurándose de que las ventajas de frames local y remota se mantengan dentro de un margen de tolerancia de un frame.

Por ejemplo, si en un extremo GGPO calcula consistentemente una ventaja de frames local de cinco, mientras que su par calcula una ventaja de tres, GGPO intentará igualar esta disparidad reduciendo la velocidad de ejecución del extremo local en un frame. Esto debería resultar en la pérdida de un fotograma para el extremo local y una ganancia de un frames para el extremo remoto, igualando la ventaja de ambos extremos a cuatro frames. La implementación de cómo ralentizar el extremo local es manejada por el desarrollador del juego al recibir una *callback* por parte de GGPO. (Cannon, 2012)

5.2.3. Condiciones para Implementar Rollback

En las partes anteriores hablamos de como funciona el método y porque es bueno enmascarando la latencia de la red, ahora nos toca hablar sobre las condiciones que un juego debe cumplir para ser capaz de implementar Rollback y cuan complicado puede llegar a ser hacerlo, más si no se tiene en cuenta desde el principio.

Guardar y cargar estados

En primer lugar el juego debe poder guardar y cagar estados previos para ser capaz de hacer rollbacks, para esto la estructura de datos que definen el game state deben ser **serializables**, a la vez esta serialización debe ser hecha de manera rápida por lo que es vital separar y encapsular el game state de lo que no es game state para reducir el tamaño y costo de la misma, por ejemplo si estamos jugando *Mortal Kombat* no es necesario guardar la posición de las partículas de fuego en el escenario pero si la posición de los personajes por lo que ambos segmentos deberían estar separados.

Determinismo

Una vez tengamos resuelto el game state, hay que asegurarse de que el siguiente estado se calcule únicamente a partir de las entradas del juego. Esto debería suceder de forma natural si encapsulamos bien el game state y las entradas, pero a veces puede ser complicado. Estos son las consideraciones que hay que tener presentes:

- Tener cuidado con los generadores de números aleatorios, muchos juegos utilizan números aleatorios para calcular el próximo estado del juego. Si se usa uno, hay que asegurarse de que sea completamente determinista, es decir que la semilla del generador de números aleatorios debe ser la misma en el frame 0 para ambos jugadores y el estado del generador de números aleatorios debe incluirse en el game state. Hacer ambas cosas garantiza que los números generados para un frame en particular sean siempre los mismos, independientemente de cuántas veces se deba retroceder a ese frame.
- Evitar fuentes de tiempo externas. Si se utiliza la hora actual del sistema como generador de números aleatorios en el cálculo del game state puede llevar a divergencias entre las instancias ya que el tiempo en dos computadoras o consolas casi nunca está sincronizado y usarlo en los cálculos del estado del juego puede causar problemas de sincronización. El uso de fuentes de tiempo externas en cálculos que no afecten el game state está bien, por ejemplo para calcular la duración de efectos en pantalla.
- Manejar bien las referencias si el game state contiene memoria asignada dinámicamente y variables estáticas.

El cuanto de tiempo

Cuando se hacen rollbacks es necesario avanzar el juego frame por frame. Esto puede ser difícil si el estado del juego avanza con una tasa de actualización variable. Por eso el game state debería avanzar un cuanto de tiempo fijo por frame. (Cannon, 2019)

Separación de la lógica y el rendering

Al avanzar el juego frame a frame luego de un rollback, este se debe hacer en segundo plano sin renderizar, de lo contrario arruinaría visualmente el juego, para esto es necesario que la actualización del game state esta separada del rendering.

Costo de avanzar un frame

Si el juego rebobina un gran número de frames, una vez que el estado previo se ha deserializado y cargado, el juego debe volver a serializar el frame confirmado y re-simular todos esos frames hasta el presente , esto debe hacerse rápidamente y en segundo plano dentro del tiempo que dura correr un frame normal. Nuevamente separar lo que es game state con lo que son efectos visuales y demás datos que no influyen en el game state es sumamente importante. Normalmente en un juego con que corra la lógica de juego una vez en la frecuencia de frames esperada es suficiente, pero con rollback debe ser capaz de correr hasta 8 ciclos en un frame. Dependiendo del motor que se este usando y cuánto se pueda personalizar este, separar y desactivar estos subsistemas puede ser muy difícil. (Pusch, 2019)

Casos borde

Además de las dificultades ya expuestas vale la pena mencionar brevemente algunos de los casos borde que uno enfrenta al querer dar soporte a este método. Crear y destruir objetos se vuelve más complicado. Por ejemplo si el jugador local lanza una bola de fuego y el oponente mantiene presionado "abajo-atrás" frente a ella, listo para bloquearla, Rollback va a predecir ese mismo inputs y simulara al oponente bloqueando esta bola de fuego en la pantalla del jugador local. La bola de fuego se destruye al ser bloqueada y se muestran efectos de partículas a su alrededor. Pero luego llegan los inputs remotos y resulta que el oponente realizó un movimiento invencible a través de la bola de fuego justo antes de bloquearla. Esto significa que en realidad no fue destruida y debe continuar viajando por la pantalla. En muchos juegos, recrear la bola de fuego después de haber sido destruida es costoso y puede generar efectos visuales extraños. Una solución que muchos juegos con rollback adoptan es simplemente guardar objetos durante varios cuadros después de ser destruidos, en caso de que el rollback necesite "des-destruirlos", lo que puede requerir ajustes en la lógica del juego. Los sistemas de partículas también necesitan mucha atención especial.

El audio es otro problema particularmente difícil en los juegos con rollback. Los sonidos que se reprodujeron en la versión predicha a menudo deben cortarse y detenerse en la versión retrocedida. Además, es posible que sonidos que no se reprodujeron en la versión local ahora deban reproducirse varios milisegundos después debido al cambio en el estado del juego por el rollback. Esto es especialmente problemático si se considera que la lógica del juego debe estar separada de todos los demás aspectos del bucle del juego. En las primeras aproximaciones de rollback en juegos modernos como *Street Fighter X Tekken*, había importantes errores de sonido donde el audio se cortaba durante segundos, o los sonidos reproducidos se interrumpían bruscamente o se desincronizaban.

También puede haber momentos en los que nunca queramos hacer rollback como durante eventos con cambios drásticos de cámara, eventos donde aparece una alerta visual llamativa que dice ***YOU WIN*** pero la predicción era errónea y el juego no había terminado, o en transiciones de escenarios. Por lo tanto, el juego debe evitar que estos eventos se ejecuten hasta que el jugador local esté 100 % seguro de que el evento ocurrió y no puede ser revertido. Esto genera un esfuerzo adicional de ingeniería y depuración para el equipo de desarrollo del juego, y por lo tanto, un mayor costo.

Si un juego no fue inicialmente desarrollado con rollback en mente, adaptarlo luego puede ser un enorme reto técnico, así ocurrió con el juego *Mortal Kombat X* en el año 2016. Inicialmente el juego contaba con el clásico Delay Based Netcode para implementar su juego online y la gente no estaba nada contenta con el, por lo que decidieron cambiarlo a Rollback Netcode. Así lo expuso el ingeniero de *NetherRealm Studios* **Michael Stallone** en la *Game Developers Conference* del año 2019. Este comentaba que el juego con su netcode antiguo gastaba 10ms de

los 16ms disponibles para correr un frame, cuando hicieron el primer cambio a rollback con una aproximación naive el costo de correr un frame se triplico hasta los 32ms lo cual forzó al los desarrolladores a rediseñar casi cada aspecto in-game del juego en pos de lograr ganancias de performance. Para lograrlo fue necesario un gran equipo de ingenieros que trabajaron concurrentemente durante 10 meses y cuyas horas de trabajo suman un total equivalente a 8 años. (Pusch, 2019) Esta interesante charla puede encontrarse completa en la siguiente referencia. (Stallone, 2019)

5.2.4. Rollback en emuladores

A la hora de implementar este método en emuladores el enfoque cambia un poco. Para empezar no se estaría implementando el método a los juegos en si, sino que la implementación seria a nivel de hardware emulado. La enorme mayoría de los juegos probablemente no vengan diseñados para jugarse online ni mucho menos diseñados para usarse con este método en mente (mas si hablamos de juegos de los 80s y 90s), por lo que para dar soporte de Rollback Netcode a la librería de un sistema de juegos en particular tenemos que tratar los juegos tal cual son, como cajas negras con poca o nula capacidad de modificación. No habrá capacidad de separar el game state del no - game state ni habrá capacidad de evitar algunos casos bordes como los mencionados anteriormente.

Este hermetismo tiene su parte positiva y su parte negativa. La parte positiva es que simplifica mucho la implementación del método, en lugar de guardar, game states guardaremos "hardware state", es decir tendremos que ir guardando y restaurando el estado de nuestros componentes de hardware emulados lo cual no debería ser un problema para sistemas antiguos basados en tiles ya que serializar el estado de sus componentes tiene bajo costo y el tamaño es pequeño. La parte negativa es que, esto a su vez, es una limitante de los sistemas emulados a los que podremos implementar rollback, ya que estos se han vuelto progresivamente más complejos, usando más registros y memoria, o incluso varios CPUs, haciendo más costoso el proceso de serialización. Para tener una referencia copiar un estado de 8 MBs tiene 1ms de costo (Dehaene, 2024). Una posible aproximación a este problema es usar una idea similar a la que usa GGPO con el manejo de inputs, es decir, a la hora de guardar y cargar estados no hacerlo en su totalidad sino guardar y restaurar solo las diferencias lo cual reduciría el costo de serializacion y manejo de estados. Si esto no es suficiente se podrían explorar alternativas más avanzadas como el uso de tecnologías GPGPU, aunque estas son simples ideas que necesitarían ser propiamente investigadas.
Más allá de todo, en emuladores de sistemas modernos como *RPCS3*, emulador de PlayStation 3, cuyos save states requirieron un enorme esfuerzo para implementarse y pueden llegar a ocupar varios GBs (Bailey, 2022), la implementación de rollback parece estar fuera de nuestro alcance en el contexto tecnológico actual.

Capítulo 6

FingEmulator

Al trabajo central de este proyecto le dimos el nombre de **FingEmulator**. ***FingEmulator*** es una plataforma de emulación diseñada para ser simple, ligera y escalable. Esta desarrollada para arquitecturas x64 con sistema operativo Windows de 64 bits. Viene integrado con un emulador de Nintendo Entertainment System capaz de emular alrededor de 80 juegos de NES que utilizan el Mapper 0. En este capítulo vamos a exponer sus funcionalidades principales,las herramientas utilizadas y el proceso de desarrollo.

Figura 6.1: FingEmulator corriendo la demo MegaManJet de **Chris Covell**. Fuente propia

6.1. Características Principales

6.1.1. Gestión de Juegos

En FingEmulator el usuario carga el ROM que va a jugar usando el dialogo de selección de archivo del sistema operativo. Esto facilita al usuario navegar por su sistema y seleccionar el juego de manera intuitiva y rápida. Si se carga un juego estando otro corriendo, se cerrará el juego previo y empezara a correr el juego recién cargado. También implementa una opción para cerrar el juego sin más.

6.1.2. Opciones del Sistema

FingEmulator implementa algunas funciones básicas para el manejo del sistema que se está emulando como son pausa, reanudar y resetear. También cuenta con la posibilidad de guardado rápido y cargado rápido de un juego por si se quiere hacer un poco de *trampa* dado que los juegos de esta época solían tener una dificultad endiablada que obligaban al jugador a empezar de nuevo el juego una vez perdían la partida. Esta practica era común en pos de aumentar su duración ya que solían ser bastante cortos. Con estas opciones el jugador puede guardar el juego en un punto conveniente y si pierde volver a cargar este punto de guardado.

6.1.3. Gestión de Inputs

En FingEmulator es posible jugar tanto con teclado como con Gamepad luego de configurarlos por separado como se ve en la Figura 6.2. Se puede usar ambos de manera intercambiable. También es posible usar distintos controles para un mismo jugador, por ejemplo el Player 1 puede mapear el Boton A para el Gamepad 1 y el Boton B para el Gamepad 2. Esto permite una configuración flexible. De manera experimental se implementó una opción de **Runahead**, Runahead es una técnica utilizada en emuladores para reducir la latencia percibida por el jugador entre que este aprieta un botón y la acción se muestra en pantalla. Esto ayuda a mitigar la latencia que existe al jugar estos juegos en TVs modernas ya que algunas suelen tener un delay entre que reciben la imagen y la dibujan. Considerando que estos juegos fueron pensados para jugarse en tecnologías analógicas con delay despreciable, esto puede llegar a ser molesto. Por lo que esta opción restara un frame de input lag, pero como mencionamos es un agregado experimental y puede distorsionar un poco el sonido.

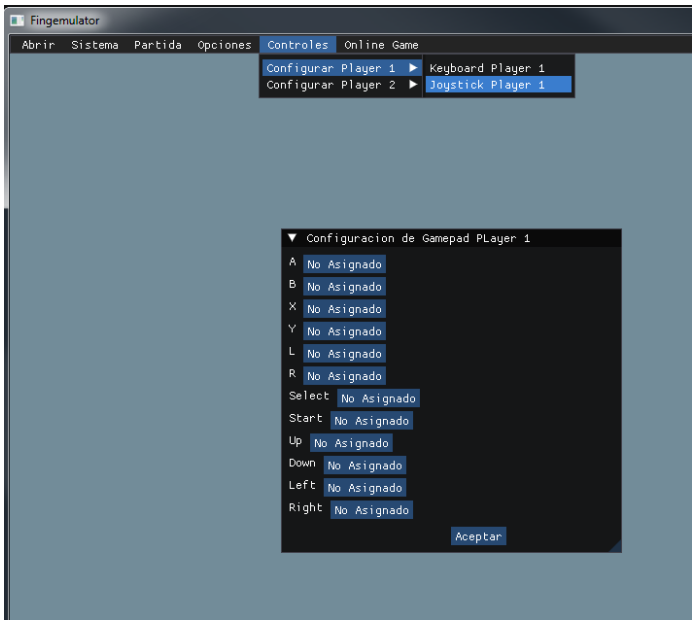


Figura 6.2: Soporte Gamepad. Fuente propia.

6.1.4. Opciones Gráficas

En cuanto a configuraciones gráficas FingEmulator tiene opciones para modo Pantalla Completa y modo Ventana, la ventana de juego se ajusta automáticamente al tamaño de la ventana. Como se ve en la figura 6.3 también hay una opción para activar un filtro lineal de imagen para suavizar un poco el aspecto de los pixeles y darle un aspecto más retro.

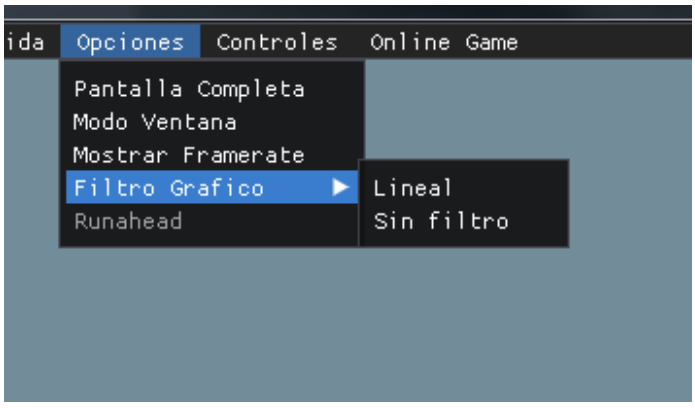


Figura 6.3: Opciones. Fuente propia

6.1.5. Gestión de Juego Online

FingEmulator incluye la posibilidad de jugar online utilizando Rollback Netcode, para esto se debe configurar los parametros que se muestran en la Figura 6.4, incluye opciones de Input Delay para reducir la distancia de los rollbacks .

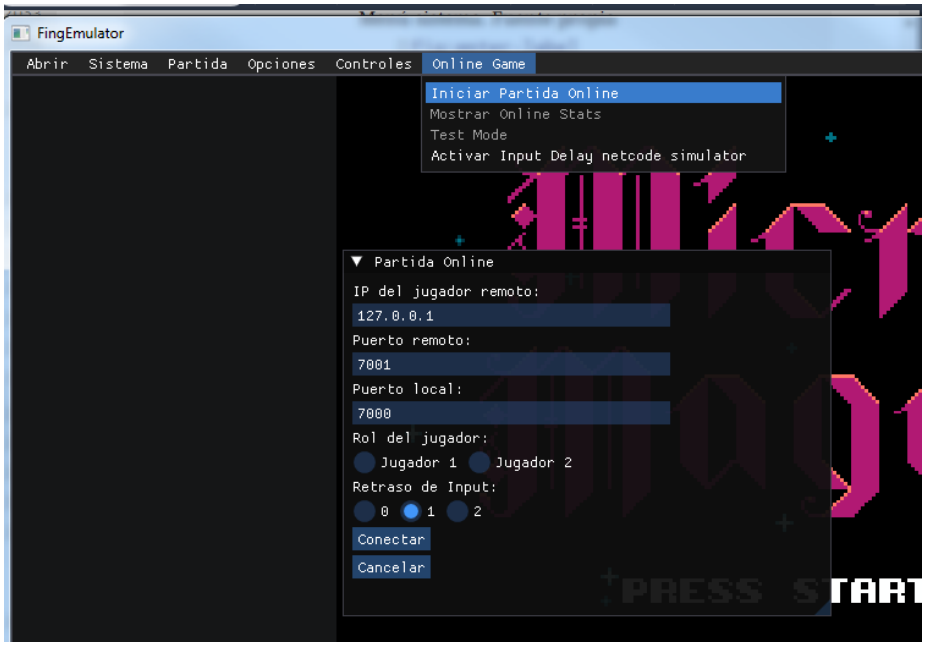


Figura 6.4: Configuración Online. Fuente propia

6.1.6. Manejo Programático a través de una API

Paralelamente a la versión implementada para el uso humano de la plataforma, se desarrollo una versión orientada al manejo programático de la plataforma que permite a otros programas cagar juegos e introducirles inputs para que estos se vayan jugando entre otras cosas. Esta versión fue implementada como una librería dinámica que puede ser agregada a cualquier proyecto que se desee. En la figura 6.5 se pueden ver las funciones que ofrece la API para el manejo programático de la plataforma.

```
// Funciones de la API
FINGEMULATOR_API void init(); // Crea e inicializa una instancia del sistema
FINGEMULATOR_API void set_system(int system); // Setea el sistema que se usara.
FINGEMULATOR_API bool load_rom(const char* path); // Carga el rom.
FINGEMULATOR_API void update_inputs(uint16_t inputs); //Envia nuevos inputs al juego
FINGEMULATOR_API uint32_t* advanceFrameOffline(bool muted); // Corre la logica del emulador. Devuelve el Frame resultante
FINGEMULATOR_API uint8_t read_memory(uint16_t address); // Funcion para poder leer direcciones de memoria.
FINGEMULATOR_API void destroy(); //Destruye la instancia del sistema
```

Figura 6.5: Funciones disponibles en la API. Fuente propia

6.2. Herramientas de Desarrollo

6.2.1. Lenguaje

La plataforma fue desarrollada en **C++**, este lenguaje fue elegido por su extenso ecosistema de librerías, porque es flexible a la hora de usar múltiples paradigmas y sobre todo por su buen rendimiento y control de los recursos, esto ultimo es esencial si queremos implementar un emulador con rollback ya que como vimos, este puede hacer uso intenso del CPU.

6.2.2. Entorno de Desarrollo

Se utilizó **Visual Studio 2019**(Microsoft, 2019) como entorno de desarrollo (IDE) debido a sus potentes herramientas que aceleraron la escritura del código y, principalmente, a su poderoso debugguer.

6.2.3. Debugging del emulador

Para optimizar el proceso de depuración del emulador, se utilizó el modo debug de **Mesen**(SourMesen, 2020) que posee una gran cantidad de herramientas de análisis tanto del código que se esta ejecutando como del estado de los componentes que emula.

6.2.4. Manejo de I/O

Para el manejo de renderizado, audio e inputs se utiliza la librería **SDL2** , SDL (Simple DirectMedia Layer) (SDL, 2024) es una biblioteca de desarrollo multiplataforma diseñada para proporcionar acceso de bajo nivel al hardware. Es muy fácil de usar y muy amigable para videojuegos y emuladores debido a su capacidad para manejar casi todo lo necesario en una sola librería. SDL abstrae las complejidades del hardware, permitiendo a los desarrolladores crear aplicaciones consistentes en diferentes sistemas operativos como Windows, macOS o Linux lo cual facilitaría en un futuro dar soporte a otros sistemas.

6.2.5. Interfaz de usuario

Para la interfaz de usuario optamos por **Dear ImGui**(Cornut, 2024), una biblioteca diseñada para interfaces gráficas inmediatas. Dear ImGui dibuja la interfaz directamente en cada frame, lo que la hace extremadamente flexible y compatible con APIs gráficas como SDL. Además, su rendimiento y facilidad de integración la convierten en una herramienta ideal para este tipo de proyectos.

6.2.6. Multijugador Online

Para implementar el multijugador online usamos el SDK de GGPO (Cannon, 2024) que nos facilita la sincronía de inputs y maneja el envió de inputs a través de internet, entre otras cosas, ya profundizamos sobre el previamente en el capítulo de Rollback.

6.3. Desarrollo

Según la cronología inicial, luego de la fase de relevamiento, estaba previsto desarrollar primero la plataforma, sin embargo se consideró apropiado empezar por el emulador ya que era la parte más compleja.

6.3.1. Desarrollo del Emulador

Inicialmente se comenzó el desarrollo del emulador bajo la arquitectura x86 ya que así estaba en los objetivos, lo primero que se empezó a desarrollar fue la CPU **MOS-6502** usando el método Interpreter dado que se trata de una CPU bastante antiguo no había necesidad de complicarse con métodos más avanzados, posteriormente se decidió por cambiar al método threaded interpreter donde se maneja una gran array a modo de opcode table cuyas entradas son punteros a las funciones correspondientes a ese opcode, haciendo el código más claro y evitando tener que iterar en un enorme switch. Si bien la CPU es más sencilla de entender y de implementar que otros componentes como la PPU, su implementación esta lejos de ser trivial. Las operaciones más complicadas de implementar del CPU fueron las de suma y resta ya que al manejar enteros con signo hay que manejar bien la flag de overflow. Las demás operaciones son más sencillas, pero es muy fácil equivocarse en algún detalle, principalmente cuando se trata de guardar y recuperar estados en la pila o en manejar la cantidad de ciclos de una operación, estos son pequeños errores que se pueden pagar **muy caro** en tiempo de depuración posteriormente. La CPU de NES tiene una serie de opcodes ilegales que algunos juegos utilizan, pero dado que son muy pocos y varios de ellos son juegos no licenciados, se decidió solo implementar los opcodes legales.

Sin lugar a dudas la parte más compleja fue la PPU, una gran cantidad de

meses fueron necesarios para empezar a tener un entendimiento claro y suficiente para poder comenzar su implementación, debido por una parte al poco conocimiento previo que teníamos sobre hardware gráfico y motores basados en tiles, y por otra parte por la naturaleza de la documentación. Como se mencionó anteriormente, estos suelen ser sistemas comerciales donde los detalles de su arquitectura son mantenidos en secreto por las empresas propietarias por lo que todos los documentos existentes son hechos por entusiastas usando ingeniería inversa. Estos documentos pueden tener errores u omisiones que obligan a tener que revisar y estudiar varias fuentes distintas para tener una visión consistente de lo que se debe hacer, además muchos conceptos o funcionalidades se presentan de manera fragmentada y en un principio es realmente difícil unir todos estos fragmentos y tener un entendimiento global de como funciona todo. Para complicar más las cosas la PPU es una gran maquina de estados que uno simplemente no puede ir implementando y probando de manera procedural y unitaria, es necesario que gran parte de esta esté implementada para poder comenzar a testearla.

Aunque no estaba especificado dentro de los objetivos del proyecto el implementar la emulación del chip de sonido de la NES, había intenciones iniciales de hacerlo con pretensiones de que sonara aceptablemente. Pero dado que solo el trato de la PPU había llevado **muchísimo** más tiempo que el estipulado por el cronograma para el proyecto entero y que la emulación de un chip de sonido requiere conocimientos específicos de la materia y es también una de las partes más complicadas, se descartó esta posibilidad. Sin embargo para no dejar el emulador mudo se decidió integrar un emulador del chip de sonido de la NES de código libre y libre distribución desarrollado por ***James Athey*** llamado *Nes_Snd_Emu* (Athey, 2024). Esto trajo luego algunas complicaciones en la implementación del Rollback, dado que no teníamos control total sobre como funcionaba el APU pero pudieron ser sorteadas.

6.3.2. Desarrollo de la Plataforma

Una vez terminado el emulador empezamos a desarrollar la plataforma en si. Para la interfaz en un comienzo intentamos usar **WinForms** (Microsoft, 2024) debido a que pueden crearse fácilmente en Visual Studio y de manera visual haciendo *Drag and Drop* de los elementos que necesitamos. Desafortunadamente había un gran problema, usar WinForms con una ventana de SDL no era una combinación sencilla de implementar debido a la diferencia en cómo ambas tecnologías manejan las ventanas y el renderizado. WinForms está diseñado para trabajar con controles y eventos dentro de su propio sistema de ventanas, mientras que SDL gestiona su propia ventana de manera independiente. Esto dificulta la integración de las ventanas, es decir, que la ventana de SDL se renderice correctamente dentro de un formulario de WinForms. Además esta debe estar centrada, debe moverse junto al formulario y debe ajustarse a su tamaño. También podía haber dificultades en el manejo de eventos y conflictos entre ellos.

Ante estos problemas decidimos volver a investigar posibles alternativas hasta que dimos con **Dear ImGui** (Cornut, 2024) que es la herramienta que terminamos usando para nuestro proyecto. Usar Dear ImGui tiene varias ventajas, especialmente si se usa SDL, ya que Dear ImGui está diseñado para ser un sistema de interfaz gráfica que se dibuja en cada frame de la ventana gestionada por SDL. Esto permite una integración directa y fluida con las funcionalidades gráficas de SDL, eliminando conflictos entre el manejo de ventanas y eventos. Empezar con la plataforma a ciegas podría haber supuesto importantes recambios luego de desarrollar el emulador y fue la decisión correcta ya que probablemente se hubiera pasado por alto las complicaciones que suponían usar WinForms con una ventana SDL embebida como se mencionó anteriormente.

Tras elegir Dear ImGui como biblioteca de interfaz, el desarrollo del resto de la plataforma avanzó de manera más clara y estructurada, integrar el emulador que habíamos hecho fue relativamente fácil. Lo ultimo en ser integrado fue la biblioteca de GGPO y aquí se presentó un problema. En el repositorio de GGPO no existen releases oficiales precompiladas disponibles para su descarga. En cambio, es necesario descargar el código fuente y compilar la biblioteca manualmente. El proyecto viene por defecto para ser compilado para x64, se intento compilar el proyecto el proyecto para arquitecturas x86 pero no tuvimos éxito, para no seguir gastando tiempo en seguir intentado, optamos por migrar el proyecto a x64 y usar la versión x64 de GGPO. Luego de migrar la plataforma e integrar GGPO se comenzó a testear el modo online. Al inicio había problemas de desincronizaciones y de Rift, estos se fueron resolviendo agregando métodos para balancear la ventaja de frames y ajustando los datos que debíamos serializar.

6.3.3. Implementación de la API

Para implementar el manejo programático de la plataforma se manejaron varias posibilidades como usar memoria compartida o implementarse como una API basada en sockets. Esta ultima aproximación fue como se intentó implementar en un inicio ya que era con la que estábamos más familiarizados. La idea era usar sockets TCP donde una hipotética IA enviaría los inputs por medio de estos sockets y la plataforma respondería devolviendo el frame resultante, luego este quedaría pausado hasta que la IA volviera a enviar inputs. Aunque la solución funcionaba tenia inconvenientes y un gran problema. Uno de los inconvenientes es que podía haber dificultades en el manejo del ***endianness*** lo cual afectaba los colores del frame recuperado desde el cliente. Pero el principal problema es que funcionaba realmente **lento**, hablamos de 1 frame por segundo o cada dos segundos, era inviable de usar. Por lo que se descartó esta solución y también el uso de memoria compartida y se procedió por una tercera alternativa que fue implementarla como una librería dinámica que luego sea agregada al proyecto que se quiera usar tal cual como se integra SDL o GGPO, lo cual lo haría muchísimo más eficiente y además muchísimo más fácil de usar. Para esto se creó un proyecto aparte dentro de la misma solución de la plataforma en Visual Studio, así se separa la API del resto del proyecto pero sin tener que

copiar los módulos existentes y que a la hora de compilar, lo haga todo junto.

6.4. Arquitectura Final de la Plataforma

En la Figura 6.5 podemos ver la arquitectura final de la plataforma.

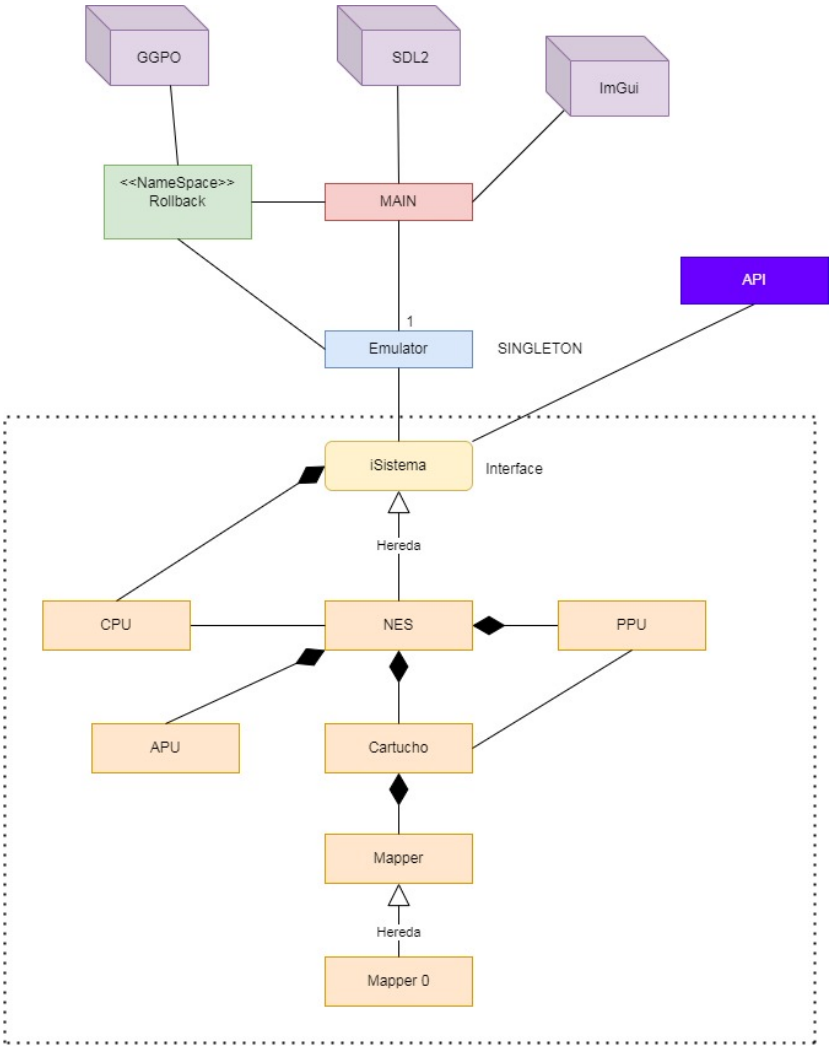


Figura 6.6: Diagrama Arquitectura general de FingEmulator. Fuente propia

Vamos a describir algunos aspectos que no son auto-explicativos al mirar el diagrama.

- **iSistema**: Esta interfaz es la encargada de separar el sistema emulado del

resto de la plataforma, dejando los emuladores encapsulados como se ve en el cuadro punteado, cada emulador que se implemente para agregar a la plataforma debe heredar esta interfaz. A su vez iSistema sirve para encapsular al propio CPU, entre NES y CPU existe una inclusión circular ya que en NES se corre la lógica principal y se encarga de comunicar los distintos componentes, por lo que NES incluye la a CPU, sin embargo CPU no incluye directamente a NES sino que incluye un puntero a una clase que herede ISistema, esto encapsula al CPU y si el día de mañana se quiere emular otro sistema que use el MOS-6502 se pueda reutilizar esta misma clase que ya tenemos implementada. En resumen, es conveniente que cada CPU que se llegue a implementar en el futuro siga esta practica para poder ser reutilizado en otros sistemas que lo usen, la misma idea se puede aplicar a los chips de sonido que se sepa que son utilizados por varios sistemas.

- **Emulator**: Emulator es la clase encargada de comunicar la interfaz con el sistema emulado, así como también de crear las instancias del sistema que se va a emular e implementar las distintas funcionalidades sobre el emulador que la plataforma ofrece, también implementa todas las funciones necesarias para el uso de GGPO. Emulator esta implementado bajo el patrón singleton ya que no queremos que haya más de una instancia existente de forma simultánea.

- **Rollback**: Rollback es la clase que se encarga de inicializar las estructuras de GGPO y de implementar las callbacks que este necesita. Para paliar el problema de Rift que se mencionó en el capítulo de Rollback, cada vez que se detecta que hay x frames de ventaja sobre el par, se pausa el emulador x frames para que queden igualados. Esto no se suele notar en una partida normal, solo es un problema si por alguna razón la instancia del emulador que está corriendo el par está funcionando por debajo de los 60 frames por segundo, lo cual causaría un Rift constante y pausas continuas.

- **MAIN**: En main se implementa el bucle principal de la plataforma, donde incluye toda la interfaz de usuario así como el manejo de eventos de entrada de inputs o interacción del usuario con la interfaz y el renderizado del frame obtenido desde emulator.

Capítulo 7

Experimentación y Pruebas

En este capítulo exponemos las pruebas realizadas para validar el correcto funcionamiento de la plataforma desarrollada. Se abordan los tres partes más importantes: la emulación, la funcionalidad de rollback para juego online y la API. Describimos las herramientas y recursos utilizados, los procedimientos seguidos y los resultados obtenidos. Tener en cuenta que al tratarse de videojuegos, algunas de estas pruebas son de carácter sensorial y de como se siente el juego.

7.1. Pruebas del Emulador

El objetivo de estas pruebas es garantizar que la CPU y la PPU del emulador funcionen correctamente, replicando fielmente el comportamiento del sistema original.

7.1.1. Prueba de CPU

- **Objetivo de la prueba**: Verificar la correcta ejecución de instrucciones, el estado de los registros y las flags de la CPU.

- **Herramienta utilizada**: ROM *nestest* desarrollada por **Kevin Horton** (NesDev, 2024a). Esta ROM hace un testeo exhaustivo de la CPU testeando, según el autor, casi todas las combinaciones de flags, instrucciones y registros.

- **Metodología**: Se corre la ROM *nestest*.

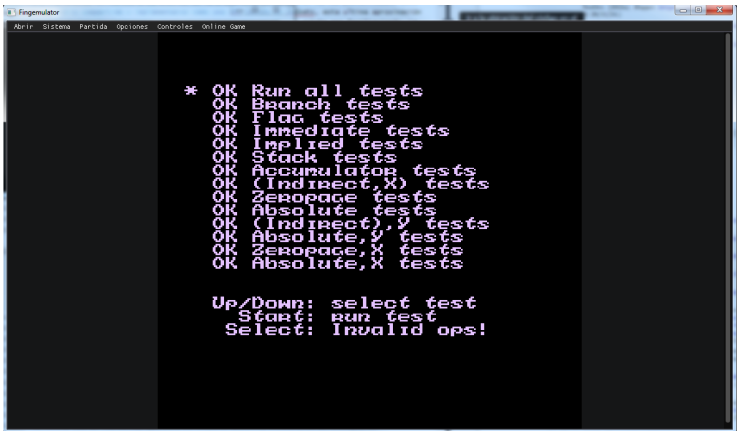


Figura 7.1: FingEmulator corriendo nestest. Fuente propia

- **Resultados**: Como puede verse en la figura 7.1 todos los test dieron resultado *OK* mostrando un funcionamiento mínimamente aceptable de la CPU. Si bien este test no asegura que el CPU carezca de fallas importantes, es un excelente punto de partida antes de empezar a probar juegos.

7.1.2. Pruebas de PPU

Prueba 1

- **Objetivos**:
  - Verificar la correcta salida de video del emulador.
  - Verificar la correcta generación de un background estatico y simple.
  - Verificar la correcta generación de sprites sencillos sin animación.
  - Verificar la correcta salida de colores.
- **Herramientas utilizadas**:
  - ROM *color_test* (Smith, 2015):
- **Metodología**: Corremos la ROM *color_test*.

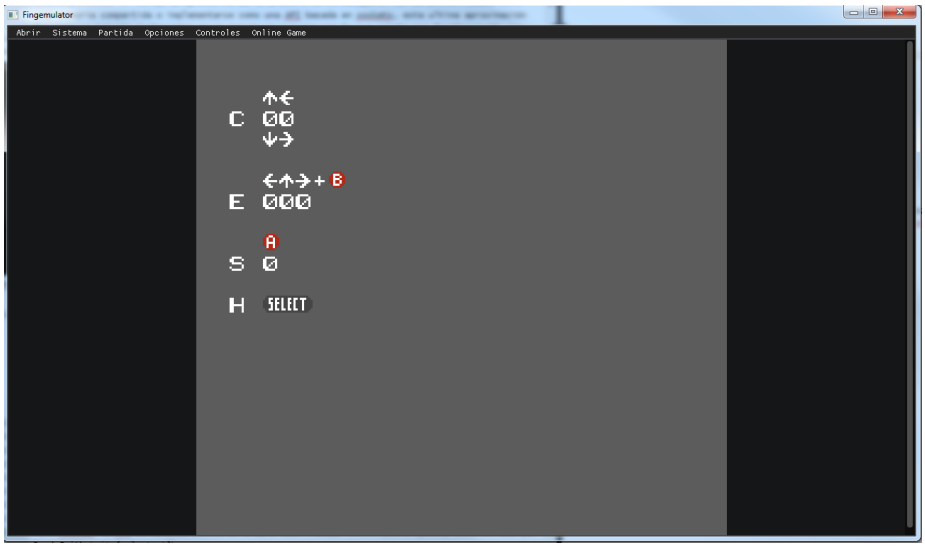


Figura 7.2: FingEmulator corriendo *color_test*. Fuente propia

- **Resultados**: La PPU mostró la salida esperada como se ve en la Figura 7.2, el background está correctos, también lo están los sprites y los colores.

Prueba 2

- **Objetivos**:
  - Verificar la correcta generación de un background más complejo y cambiante.
  - Verificar el correcto funcionamiento del scrolling horizontal.
- **Herramientas utilizadas**:
  - ROM *MegaManJet* de **Chris Covell** (Covell, 2003):
- **Metodología**: Corremos la ROM *MegaManJet*.

Figura 7.3: FingEmulator corriendo *MegaManJet*. Fuente propia

- **Resultados**: La PPU mostró la salida esperada como se ve en la Figura 7.3, el background estaba correcto y el scrolling horizontal funcionaba bien.

7.1.3. Prueba conjunta de todos los componentes del emulador de NES

- **Objetivos**:
  - Verificar el correcto funcionamiento del scrolling vertical y de todos los componentes.
  - Verificar la estabilidad de la plataforma en prolongados periodos de ejecución.
- **Herramientas utilizadas**:
  - ROM del juego *Micro Mages* (M. Games, 2019) de NES (el autor tiene una copia del juego comprada).
- **Metodología**: Jugamos por un tiempo prolongado (una hora) un juego de NES.

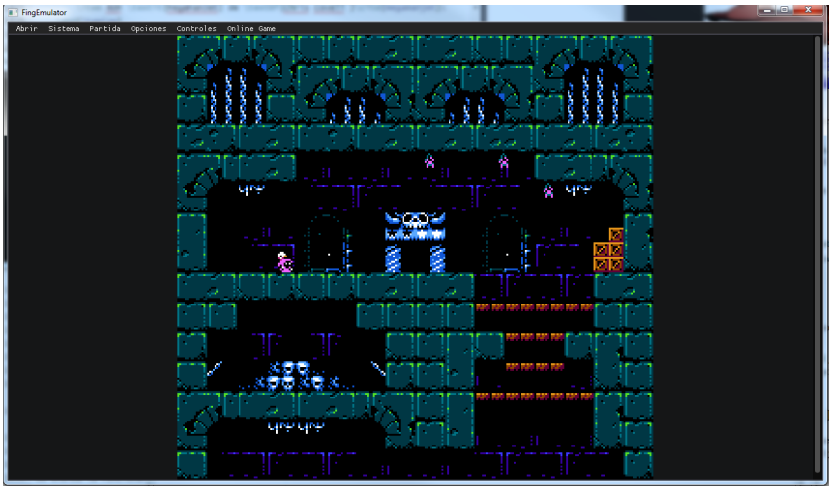


Figura 7.4: FingEmulator corriendo *Micro Mages*. Fuente propia

- **Resultados**: Estuvimos jugando el juego durante una hora, en la Figura 7.4 podemos ver que el juego se ve correctamente, el scrolling vertical y el juego en general funcionó bien todo el tiempo.

7.2. Pruebas de Juego Online

Se evalúa la funcionalidad del juego online basado en Rollback para analizar su desempeño en condiciones adversas de red y se compara con el clásico método Delay Input.

7.2.1. Prueba de rendimiento de Rollback Netcode

- **Objetivos**:
  - Verificar que el juego online es jugable en condiciones de red adversas.
  - Verificar que el juego online esté funcionando bien sin desincronizaciones en condiciones de red adversas.
  - Verificar que el emulador no baje de los 60 FPS al hacer rollbacks de 8 frames por cada frame ejecutado.
  - Verificar que no haya fugas de memoria en el guardado de game states durante el juego online.
  - Verificar que no haya problemas de Rift.
  - Verificar la estabilidad de la plataforma en prolongados periodos de ejecución de juego online.
- **Herramientas utilizadas**:

- ROM del juego *Micro Mages* (M. Games, 2019) de NES (el autor tiene una copia del juego comprada).
- Funcionalidades implementadas para mostrar el frame rate al que está corriendo el emulador y stats de la calidad de conexión.
- Administrador de tareas de Windows para monitorizar el uso de memoria RAM de la plataforma.
- Modo test implementado para que, de manera automática, se genere un input distinto cada frame y fuerce un rollback en el jugador remoto.
- Un programa de código libre llamado Clumsy (Tao, 2021) que simula comportamiento de internet de manera local.
- Un PC con CPU i5-3470T a 2.9GHz, 8 GBs de RAM y Windows 7 64-Bits y otro PC más potente. Ambos conectados por red local.

- **Metodología**: Jugamos por un tiempo prolongado un juego de NES en modo online usando dos PCs conectados por red. En el PC más débil jugamos normal con inputs de origen humano y se evalúa el rendimiento. En el otro más potente se ejecuta Clumsy usando la configuración que se muestra en la Figura 7.4 y el modo test para forzar rollbacks en el PC más débil.

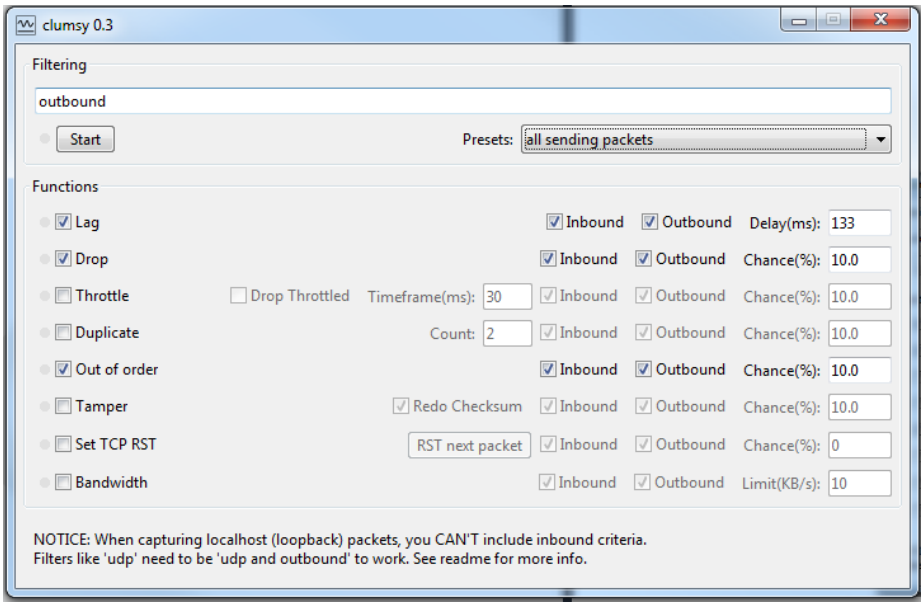


Figura 7.5: Configuración de Clumsy para la prueba de rendimiento de Rollback Netcode

- **Resultados**: Ejecutamos la prueba alrededor de 40 minutos, ambas instancias se mantuvieron sincronizadas y en el PC menos potente el frame rate se mantuvo estable en los 60 FPS. El uso de memoria RAM también se mantuvo estable y el Rift se mantuvo controlado. A pesar de los notables saltos por los rollbacks y las pausas para balancear el Rift el juego era responsivo y jugable. Por lo que todos los objetivos de esta prueba tuvieron buenos resultados.

7.2.2. Comparación del método Rollback con el método Delay Input

- **Objetivos**:
  - Comparar en términos sensoriales el método Delay Input con el método Rollback bajo condiciones similares a la anterior prueba.
- **Herramientas utilizadas**:
  - ROM del juego *Micro Mages* (M. Games, 2019) de NES (el autor tiene una copia del juego comprada).
  - La funcionalidad Input Delay Netcode Simulator implementada para simular como respondería un juego bajo una latencia configurable.
- **Metodología**: Jugar un juego de NES con el simulador activado y configurado como se ve en la Figura 7.5.

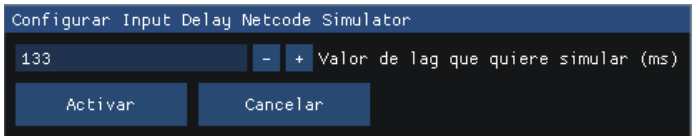


Figura 7.6: Input Delay Netcode Simulator

- **Resultados**: El juego visualmente corre más fluido y sin saltos ni errores de sonido, pero los controles se sienten muy pesados y poco responsivos, muy incomodo de jugar.

7.3. Prueba API

- **Objetivos**:
  - Verificar que el uso programático del emulador está funcionando como se espera.
  - Verificar la estabilidad de la API en prolongados periodos de ejecución.

- **Herramientas utilizadas**:
  - ROM del juego *Micro Mages* (M. Games, 2019) de NES (el autor tiene una copia del juego comprada).
  - Un test implementado que utiliza el DLL de la API, este comienza a correr un juego e introduce inputs aleatorios simulando una IA, al final de cada frame se dibuja el frame para corroborar que todo funciona correctamente.
- **Metodología**: Corremos el test por un tiempo prolongado (30 minutos).
- **Resultados**: Ejecutamos el test por 30 minutos. El frame que se obtenía de la API estaba correcto porque se veía al ser dibujado, el personaje manejado por el test con inputs aleatorios realizaba distintas acciones y no hubo problemas de ejecución.

7.4. Resumen de resultados obtenidos

Todas las pruebas realizadas arrojaron resultados satisfactorios. La plataforma mostró un comportamiento estable, sin cierres inesperados y con un uso de memoria acotado, lo que brinda cierta confianza en su buen funcionamiento, tanto en su versión normal como en su uso como librería.

En cuanto a la emulación de NES, todas las demos y juegos testeados funcionaron como se esperaba. Esto sugiere un buen nivel de precisión en la emulación. Sin embargo, como mencionamos en el Capítulo 2 sección 2.2.6, es difícil garantizar una emulación completamente funcional, ya que esto requeriría probar la totalidad de los juegos de NES de principio a fin. Por lo tanto, es posible que algunos juegos no funcionen correctamente.

Por último, la funcionalidad online mostró un desempeño muy sólido. Incluso bajo condiciones de red adversas y con inputs que cambiaban en cada frame, la partida online nunca se desincronizó y el juego respondió de manera fluida.

Estos resultados, mas allá de las limitaciones de las pruebas, nos permiten tener cierto grado de confianza en el buen funcionamiento de la plataforma en general.

Capítulo 8

Conclusiones y Trabajo Futuro

En este capítulo final damos cierre al documento con un resumen de los logros del proyecto, las conclusiones que sacamos del trabajo realizado y proponemos posibles direcciones para extender la plataforma.

8.1. Conclusiones

Este proyecto tenía cuatro objetivos iniciales, siendo los primeros tres los que el autor consideraba de máxima importancia.

El primer objetivo consistía en diseñar una plataforma modular y fácilmente escalable en el lenguaje C++ para que funcione en sistemas x86 con sistema operativo Windows. Este objetivo fue cumplido casi en su totalidad, salvo por el detalle de la arquitectura x86, que terminó siendo implementada para arquitecturas x64, que es el estándar actualmente. La plataforma se presenta en el capítulo 6 junto a su arquitectura (sección 6.4). En el Anexo 1 se encuentra el enlace al repositorio para su descarga.
El segundo objetivo era dar soporte para juego multijugador a través de Internet, utilizando un modelo P2P y Rollback Netcode para reducir la percepción de latencia. Este objetivo también se cumplió; en la sección 6.1.5 se muestra la funcionalidad implementada y en la sección 6.3.1 se comenta su implementación.
El tercer objetivo consistía en que la plataforma fuese capaz de emular eficientemente la consola Nintendo Entertainment System (NES) de 8 bits. Este objetivo fue alcanzado, como se puede ver en el capítulo 7 de pruebas, en el que se demuestra que la plataforma es capaz de correr tanto demos como juegos de NES sin errores gráficos aparentes. El cuarto objetivo consistía en implementar un sistema de entrada/salida que permitiera la interacción programática con la plataforma. Este objetivo se alcanzó mediante la implementación de la API

expuesta en la sección 6.1.6, cuyo desarrollo se comenta en la sección 6.3.3. Por medio de las funciones disponibles, cualquier programa puede cargar un juego de los sistemas disponibles, "jugar" enviando inputs, recibir los frames resultantes y leer el estado de la memoria.
Por lo tanto, todos los objetivos fueron cumplidos, aunque para alcanzarlos fue necesario muchísimo más tiempo del planificado, debido a lo ambicioso del proyecto y a la inexperiencia del autor programando emuladores. El proceso de desarrollo resultó considerablemente más difícil de lo esperado, principalmente por la dificultad de construir un emulador de un sistema complejo como la NES sin experiencia previa.

El primer gran desafío fue determinar en detalle qué se debía hacer. La documentación fragmentada sobre distintas partes de un mismo componente, los detalles imprecisos, omitidos o incompletos que obligaban a consultar múltiples fuentes, y la propia complejidad de la máquina hicieron que la curva de aprendizaje fuera muy dura de atravesar, requiriendo invertir una enorme cantidad de tiempo para lograr un entendimiento global del funcionamiento del sistema. Afortunadamente, el conocimiento previo adquirido en los cursos de Arquitectura de Computadoras y Redes de Computadoras fue de gran ayuda para agilizar la comprensión de ciertas estructuras y conceptos.

El segundo gran desafío fue depurar errores, principalmente aquellos relacionados con la CPU. Hay que tener en cuenta que, al depurar normalmente un programa, el error suele estar en el propio software; sin embargo, en el caso de un emulador se está implementando hardware en el que correrá un software cuyo funcionamiento y lógica desconocemos. Esto implica que, si el sistema falla, el error estará en el hardware que implementamos y no en el software que se ejecuta. La CPU ejecuta miles de instrucciones por segundo, y cualquier pequeño error puede desencadenar la corrupción total del sistema, obligándonos a rastrear el fallo de manera regresiva, lo cual es difícil y consume mucho tiempo. Las pruebas realizadas con demos y tests, expuestos en la sección 7.1, fueron de gran ayuda, ya que al tratarse de programas significativamente más pequeños y simples que un juego, era más fácil y rápido rastrear los errores de implementación en las instrucciones más básicas de la CPU. Esto permitió un procedimiento de depuración más progresivo que, de haber usado solo juegos, hubiera resultado mucho más complejo. Lo mismo ocurrió con las pruebas de la PPU: empezar a depurar con demos que presentaban gráficos sencillos e ir progresando hacia demos y juegos con elementos más avanzados como el scrolling, permitió identificar más rápidamente en qué parte de la implementación de la PPU había problemas. También hay otras dificultades menores pero siempre presentes a la hora de programar a nivel de bits como son la baja legibilidad que tiene el código y que también es fácil equivocarse en alguna operación de enmascaramiento, afortunadamente C++ tiene algunas funcionalidades para facilitar un poco esto. En resumen, programar un emulador es muy difícil y lleva un montón de tiempo.

Sin embargo no todo es sufrimiento, programar un emulador proporciona va-

liosa experiencia profundizando los conocimientos sobre arquitectura de computadoras, aprendiendo a interpretar documentación de hardware, y mejorando las habilidades de programación a nivel de bits y en general.

Implementar la plataforma de manera modular y utilizar GGPO fue un gran acierto, este método utilizado solamente en juegos de lucha funciona sorprendentemente bien con estos juegos antiguos. Principalmente aquellos de carácter cooperativo donde ambos jugadores pasan desafíos juntos, lo cual hace emocionante pensar en el potencial que puede tener la plataforma si se agregan más sistemas. A pesar de las complejidades que pueden presentarse a la hora de querer implementar este método, las pruebas realizadas en la sección 7.2 muestran lo robusto que es en condiciones muy desfavorables de la red y cuan superior es al clásico método basado en input delay, por lo que vale totalmente la pena hacerlo. Crear esta plataforma ayudó a profundizar el conocimiento adquirido en redes de computadoras y particularmente en el Gaming NetWorking, mejorar habilidades en cuanto a diseño de software y familiarizarse con distintas tecnologías como son SDL, Dear ImGui y GGPO . Esperamos que la API y el proyecto en general sea útil para otros estudiantes para aprender sobre emulación o para usar en sus proyectos de Inteligencia Artificial. A pesar de todas las dificultades fue un proyecto divertido y muy satisfactorio.

8.2. Trabajo Futuro

Hay bastante trabajo con el que mejorar y robustecer la plataforma, vamos a nombrar algunas tareas:

- Hacer una implementación propia de la APU que permita flexibilidad para manejar mejor el sonido cuando se usa GGPO o se juega con el modo Runahead.
- Implementar los opcodes ilegales del MOS 6502 para aumentar la cantidad de juegos compatibles.
- Implementar más mappers para aumentar la cantidad de juegos compatibles.
- Mejorar la robustez en general y manejo de errores en la plataforma.
- Agregar más sistemas emulados a la plataforma.
- Agregar más opciones gráficas como soporte de filtros CRT para un aspecto más similar al tipo de imagen que ofrecían esas TVs.
- Agregar persistencia en los configuraciones y poder guardar estados de juego en archivos para poder cagarlos luego de cerrar el programa.
- Agregar opciones de manejo de Audio.

- Implementación de un servidor que sirva para Matchmaking (para encontrar a otros jugadores) y que implemente UDP hole punching para servir como intermediario a la hora de conectar dos jugadores que estén detrás de distintas NATs, ya que como está el emulador ahora es necesario que cada jugador haga redirecionamiento de puertos (port forwarding) manualmente para poder jugar a través de internet con la plataforma.

Glosario

- ***CPU (Central Processing Unit)***: Unidad Central de Procesamiento.
- ***PPU (Picture Processing Unit)***: Unidad de Procesamiento de Imagen, responsable de generar los datos de imagen.
- ***APU (Audio Processing Unit)***: Unidad de Procesamiento de Audio, responsable de generar datos de audio.
- ***ROM (Read-Only Memory)***: Memoria de solo lectura. En el contexto del desarrollo de emuladores, contiene los datos del juego.
- ***RAM - Random access memory***: Memoria de acceso aleatorio. A diferencia de la ROM, la RAM puede ser tanto leída como escrita. Sin embargo, los datos almacenados en la RAM son volátiles: la memoria conserva los datos solo mientras tiene energía.
- **Little endian** es un formato de almacenamiento de datos en memoria donde el byte menos significativo de un valor se almacena primero en memoria
- **Sprite**: Es una imagen o conjunto de imágenes que representan personajes u objetos móviles en un videojuego, generalmente animados y colocados sobre el fondo.
- **Shift Register**: circuito digital que almacena datos y los desplaza secuencialmente hacia la izquierda o derecha, bit a bit.
- ***Ping***: Medida del tiempo, expresado en milisegundos (ms), que tarda un paquete de datos en viajar desde el dispositivo A al dispositivo B y regresar. Es un indicador clave de la latencia en conexiones de red.
- ***Jitter***: Variación en el tiempo de llegada de paquetes de datos en una red, causada por fluctuaciones en la latencia.
- ***Netpause***: El juego se pausa debido a no estar recibiendo paquetes remotos por demasiado tiempo.
- ***Input Delay***: Latencia existente entre el presionado de un botón y la respuesta del emulador.

- ***Frame***: Una unidad individual de imagen que compone una animación o un video. En el contexto de videojuegos, un frame representa un estado visual del juego en un momento específico. La velocidad a la que los frames se muestran en pantalla, medida en frames por segundo (FPS), determina la fluidez de la experiencia visual. También se usa para referirse al procesamiento de todos los cálculos necesarios para representar un instante del juego, incluidos gráficos, audio y lógica del sistema.
- ***FrameRate***: Tasa de frames por segundo.
- ***Glitch*** : es un error visual que ocurre en un juego o programa, donde los gráficos se muestran de manera incorrecta o inesperada.
- ***Game State***: Representación completa de todas las variables y datos relevantes que describen el momento actual de un videojuego, incluyendo aspectos como posiciones de objetos, valores internos de las mecánicas, estado de los personajes y cualquier otra información necesaria para definir el progreso y las condiciones del juego en un instante específico.
- ***Determinismo***: Propiedad de un sistema o programa que garantiza que, dado un mismo estado inicial y las mismas entradas, siempre producirá exactamente el mismo resultado.
- ***Serializable***: capacidad de un objeto o estructura de datos para ser convertido en un formato que pueda almacenarse o transmitirse (como un archivo o un flujo de datos) y posteriormente reconstruirse en su estado original.
- ***Desincronización***: Situación en la que dos o más instancias de un juego o emulador, que deberían estar en el mismo estado, divergen debido a discrepancias en los datos, entradas o condiciones de ejecución., es decir, no está siendo determinista.
- ***UDP hole punching*** : Técnica de redes que permite a dos dispositivos detrás de enrutadores de Traducción de Direcciones de Red (NAT) separados establecer una conexión directa entre sí.

Referencias

ArcadeOtakuWiki. (2023). *CPS3.* https://wiki.arcadeotaku.com/w/CPS3#Suicide. (Visitado el: 12/12/24.)

Athey, J. (2024). *Nes_snd_emu.* https://github.com/jamesathey/Nes_Snd_Emu/tree/master. (Visitado el: 06/12/24.)

Babalon. (2023). *NTSC vs PAL differences - which video standard is better for gaming.* https://altarofgaming.com/ntsc-vs-pal-differences/. (Visitado el: 21/11/24.)

Bagmanov, R. (2020). *PPU scrolling.* https://bugzmanov.github.io/nes_ebook/chapter_8.html. (Visitado el: 28/11/24.)

Bailey, D. (2022). *PS3 emulator RPCS3 finally adds save states.* https://www.gamesradar.com/ps3-emulator-rpcs3-finally-adds-save-states/. (Visitado el: 19/12/24.)

Battle(non)sense, C. (2017). *Netcode explained.* https://www.pcgamer.com/netcode-explained/. (Visitado el: 13/12/24.)

Bravo, E. (2023). *Nintendo NES, la consola que revolucionó los videojuegos, cumple 40 años.* https://www.revistagq.com/articulo/nintendo-nes-historia-aniversario. (Visitado el: 22/11/24.)

Cannon, T. (2012, September). Fight the lag! the trick behind GGPO's low latency netcode [Articulo de Prensa]. *Game Developer Magazine's*, 8-13.

Cannon, T. (2019). *GGPO guide.* https://github.com/pond3r/ggpo/blob/master/doc/DeveloperGuide.md. (Visitado el: 02/12/24.)

Cannon, T. (2024). *GGPO.* https://www.ggpo.net/. (Visitado el: 29/11/24.)

Copetti, R. (2022). *Sega Saturn architecture a practical analysis.* https://www.copetti.org/writings/consoles/sega-saturn/. (Visitado el: 19/11/24.)

Cornut, O. (2024). *Dear ImGui.* https://github.com/ocornut/imgui. (Visitado el: 05/12/24.)

Covell, C. (2003). *Mega Man Jet.* https://chrismcovell.com/videogames.html. (Visitado el: 21/11/24.)

Dehaene, D. (2024). *Delta rollback: New optimizations for rollback netcode.* https://medium.com/@david.dehaene/delta-rollback-new-optimizations-for-rollback-netcode-7d283d56e54b. (Visitado el: 04/12/24.)

del Barrio, V. M. (2001). Study of the techniques for emulation programming [Tesis de Grado]. *Facultad de Informática de Barcelona - Universidad*

Politécnica de Cataluña..

Diskin, P. (2004). Nintendo entertainment system documentation [System Documentation]. https://www.nesdev.org/NESDoc.pdf.

Earl, M. (2018). *Extracting super mario bros levels with python.* https://matthewearl.github.io/2018/06/28/smb-level-extractor/. (Visitado el: 21/11/24.)

Ehlert, A. (2021). Improving input prediction in online fighting games [Tesis de Grado]. *KTH Royal Institute of Technology*.

EmulationGeneralWiki. (2024). *High/low level emulation.* https://emulation.gametechwiki.com/index.php/High/Low_level_emulation. (Visitado el: 13/12/24.)

EVO. (2022). *EVO.* https://www.evo.gg/about-evo. (Visitado el: 29/11/24.)

Fiedler, G. (2008). *NAT punch-through for multiplayer games.* https://www.gafferongames.com/post/udp_vs_tcp/. (Visitado el: 14/12/24.)

Fiedler, G. (2024). *Choosing the right network model for your multiplayer game.* https://mas-bandwidth.com/choosing-the-right-network-model-for-your-multiplayer-game/. (Visitado el: 17/12/24.)

Games, E. (2024). *Unreal engine 5.5 documentation.* https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-5-documentation. (Visitado el: 18/12/24.)

Games, M. (2019). *Micro mages.* http://morphcat.de/micromages/. (Visitado el: 19/12/24.)

Glue logic definition. (s.f.). https://www.pcmag.com/encyclopedia/term/glue-logic. (Visitado el: 22/11/24.)

Hori, Y. (1997, June). Characteristics of UDP packet loss: Effect of TCP traffic. *Proceedings of INET '97, Engineering 3-1, June 1997*.

Hors, A. (2023). *Writing an NES emulator: Part 1 - the 6502 CPU.* https://analog-hors.github.io/site/pones-p1/. (Visitado el: 23/11/24.)

INCO. (2011). Notas de teórico arquitectura de Von Neumann [Notas Teoricas. Curso Arquitectura de Computadoras]. *Facultad de Ingenieria UDELAR*. https://eva.fing.edu.uy/pluginfile.php/313136/mod_folder/content/0/2019/ArquitecturaDeVonNeumann.pdf.

Johnson, D., y Antonelli, W. (2020). *Emulators can turn your PC into a Mac, let you play games from any era, and more here's what you should know about the potential benefits and risks of using one.* https://www.businessinsider.com/what-is-an-emulator?r=US&IR=T. (Visitado el: 15/11/24.)

Johnston, K. (2014). *NAT punch-through for multiplayer games.* https://keithjohnston.wordpress.com/2014/02/17/nat-punch-through-for-multiplayer-games/. (Visitado el: 13/12/24.)

Kent, S. L. (2016). *La gran historia de los videojuegos* [Libro]. NOVA.

Kuchera, B. (2011). *Accuracy takes power: one man's 3GHz quest to build a perfect SNES emulator.* https://arstechnica.com/gaming/2011/08/accuracy-takes-power-one-mans-3ghz-quest-to-build-a-perfect-snes-emulator/. (Visitado el: 20/11/24.)

Libretro. (2024). *Netplay.* https://docs.libretro.com/development/retroarch/netplay/. (Visitado el: 04/12/24.)
Loopy. (1999). *NES scroll.* https://www.nesdev.org/nesscroll-sp.pdf. (Visitado el: 29/11/24.)
Machado, F. (2001). Unidad grafica basada en mapas de teselas [Presentación]. *Universidad Rey Juan Carlos*. https://zenodo.org/records/5393956.
MAME. (2024). *MAME.* https://www.mamedev.org/. (Visitado el: 04/12/24.)
mamedev. (2008). *Core concepts.* https://wiki.mamedev.org/index.php/Core_Concepts. (Visitado el: 17/11/24.)
MednafenTeam. (2024). *Mednafen.* https://mednafen.github.io/documentation/netplay.html#Section_intro. (Visitado el: 06/12/24.)
MicrocontrollersLab. (2022). *What is interrupt vector table?* https://microcontrollerslab.com/what-is-interrupt-vector-table/. (Visitado el: 23/11/24.)
Microsoft. (2019). *Visual Studio.* https://visualstudio.microsoft.com/es/. (Visitado el: 05/12/24.)
Microsoft. (2024). *Windows Forms.* https://learn.microsoft.com/es-es/dotnet/desktop/winforms/overview/?view=netdesktop-9.0. (Visitado el: 05/12/24.)
Morlan, A. (2019). *An overview of NES rendering.* https://austinmorlan.com/posts/nes_rendering_overview/. (Visitado el: 28/11/24.)
M-Tee. (2024). *NES 8x16 sprites overview.* https://www.videogamesage.com/forums/topic/4831-an-overview-of-8x16-sprite-mode-for-the-nes/. (Visitado el: 28/11/24.)
Nazar, I. (2010). *Gameboy emulation in JavaScript: Memory.* https://imrannazar.com/series/gameboy-emulation-in-javascript/memory. (Visitado el: 12/12/24.)
NesDev. (2022a). *Mapper.* https://www.nesdev.org/wiki/Mapper. (Visitado el: 29/11/24.)
NesDev. (2022b). *NES2.0.* https://www.nesdev.org/wiki/NES_2.0. (Visitado el: 29/11/24.)
NesDev. (2022c). *PPU rendering.* https://www.nesdev.org/wiki/PPU_rendering. (Visitado el: 29/11/24.)
NesDev. (2022d). *Standard controller.* https://www.nesdev.org/wiki/Standard_controller. (Visitado el: 29/11/24.)
NesDev. (2024a). *Emulator tests.* https://www.nesdev.org/wiki/Emulator_tests. (Visitado el: 06/12/24.)
NesDev. (2024b). *PPU registers.* https://www.nesdev.org/wiki/PPU_registers#PPUCTRL. (Visitado el: 28/11/24.)
NES instruction reference. (2024). https://www.nesdev.org/wiki/Instruction_reference#NOP. (Visitado el: 23/11/24.)
NFI. (2024). *Composite video – everything you need to know.* https://www.nfi.edu/composite-video/. (Visitado el: 27/11/24.)
Nintendo. (s.f.). *How to activate/deactivate low-latency mode for NES - Nintendo Switch Online games.* https://www.nintendo.com/en-gb/Support/Nintendo-Switch/How-to-Activate-Deactivate-Low-Latency

-Mode-for-NES-Nintendo-Switch-Online-Games-1499214.html#:~:text=Start%20an%20online%20game%20from,turn%20on%20low%2Dlatency%20mode. (Visitado el: 04/12/24.)

Osbourn, T. (2020). *Space invaders and the fun reason it gets progressively harder*. https://tosbourn.com/space-invaders/. (Visitado el: 10/12/24.)

Parkes, J. (2013). *nametable*. https://jarrodparkes.com/2013/12/20/nes-nametables/. (Visitado el: 21/11/24.)

Pusch, R. (2019). *Explaining how fighting games use delay-based and rollback netcode*. https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/. (Visitado el: 01/12/24.)

Richeson, J. M. (2017). The anatomy of a hardware emulator [Tesis de Maestria]. *University of Texas at Austin*.

Sanchez, A. (2018). *LLE vs HLE and their tradeoffs*. https://alexaltea.github.io/blog/posts/2018-04-18-lle-vs-hle/. (Visitado el: 13/12/24.)

Satti, M. (2016). Introduction to dynamic recompilation in emulation. https://github.com/marcosatti/Dynarec_Guide/tree/master.

scummvm.org. (2024). *ScummVM*. https://www.scummvm.org/. (Visitado el: 18/12/24.)

SDL. (2024). *Simple DirectMedia layer*. https://www.libsdl.org/. (Visitado el: 05/12/24.)

Smith, B. (2015). *Color test*. https://forums.nesdev.org/viewtopic.php?p=155593#p155593. (Visitado el: 06/12/24.)

SourMesen. (2020). *Mesen*. https://www.mesen.ca/docs/tools.html. (Visitado el: 06/12/24.)

Stallone, M. (2019). *8 frames in 16ms: Rollback networking in mortal kombat and injustice 2*. https://youtu.be/7jb0FOcImdg?si=iLBQrM3hIgt2pQeD. (Visitado el: 03/12/24.)

Tao, C. (2021). *Clumsy*. https://github.com/jagt/clumsy. (Visitado el: 11/12/24.)

Taylor, B. (2004). *2A03 technical reference*. https://www.nesdev.org/2A03%20technical%20reference.txt. (Visitado el: 22/11/24.)

Thibault, C. (2004). *Kaillera*. http://www.kaillera.com/faq.php#1. (Visitado el: 04/12/24.)

Ulfalizer. (2022). *NTSC PPU frametiming*. https://www.nesdev.org/w/images/default/4/4f/Ppu.svg. (Visitado el: 29/11/24.)

visual6502.org. (2012). *Visual transistor-level simulation of the 6502 CPU*. http://www.visual6502.org/. (Visitado el: 18/12/24.)

Anexo 1. Requerimientos y Manual de Uso

1. **Donde Descargarlo:** La plataforma se encuentra disponible aquí junto a una pequeña guía de compilación.

2. **Requerimientos:** Para correr Fingemulator solo es necesario una PC x64 con Sistema Operativo Windows de 64 bits.

3. **Ejecutar Fingemulator:** Para ejecutar la plataforma debemos hacer doble click en el ejecutable desde el explorador de Windows.

4. **Abrir un juego :** Para abrir un juego, ir al menú superior y apretar el botón *Abrir* - *Juego NES* como se ve en la Figura 1.

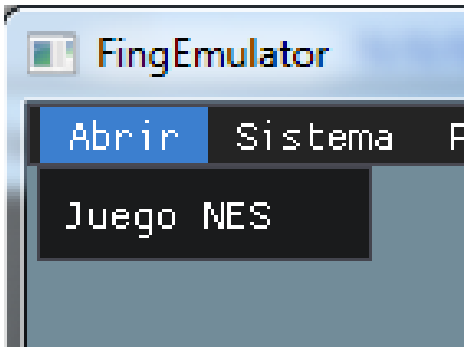


Figura 1: Abrir juego. Fuente propia

5. **Configurar Inputs:** Para configurar los Inputs ir al *Controles* en el menú superior y elegir qué jugador vamos a configurar y cuál dispositivo. En la Figura 2 podemos ver el caso de configuración de teclado, para configurar un Input hacemos click en el botón azul que tiene a su derecha y luego presionamos la tecla que queremos usar. Alternativamente hay un botón Default donde se hace una asignación por defecto a cada Input. En caso

de querer configurar un Joystick, la ventana es similar como se ve en la Figura 3 y la metodología es la misma.

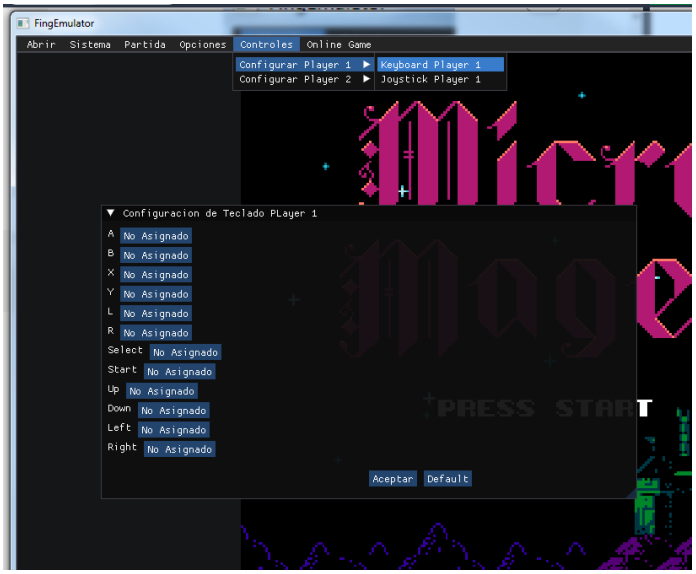


Figura 2: Configuración Teclado. Fuente propia.

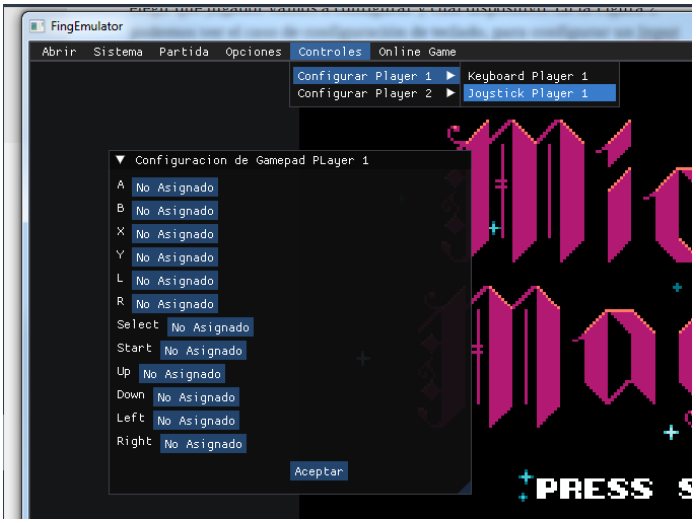


Figura 3: Configuración GamePad. Fuente propia

6. **Modo Pantalla, Filtro y Runahead:** Para el modo pantalla completa o ventana y el manejo de filtros debemos ir a *Opciones* en el menú superior y

elegir la opción que deseamos como se ve en la Figura 4. Para el Runahead es necesario que el juego ya esté abierto.

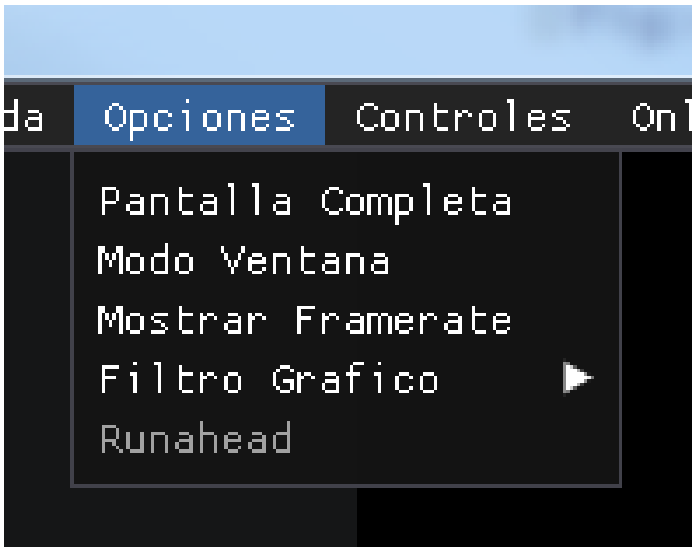


Figura 4: Menú opciones. Fuente propia

7. **Manejo del Sistema:** Si queremos pausar el juego, resetearlo o cerrarlo debemos ir a Sistema en el menú superior como se ve en la Figura 5.

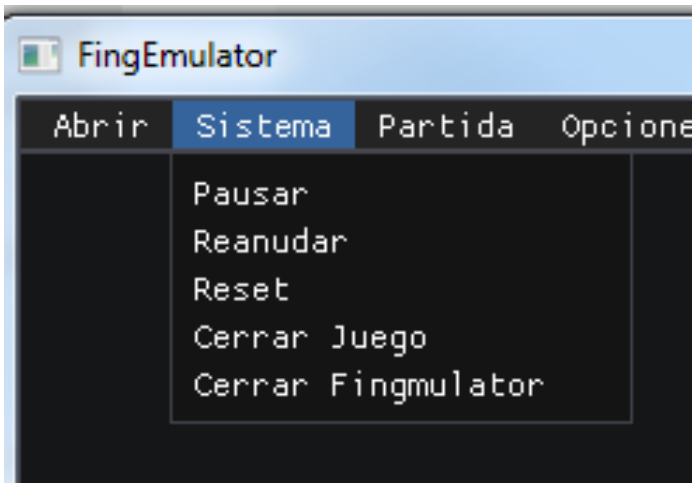


Figura 5: Menú sistema. Fuente propia

8. **Guardar y Cargar Partida:** Para hacer un guardado y cargado rápido

de la partida que estamos jugando ir a Partida en el menú superior y elegir la opción que deseamos como se ve en la Figura 6.

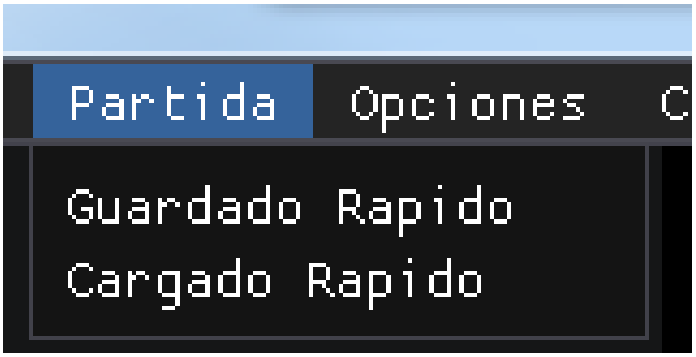


Figura 6: Menú Partida. Fuente propia

9. **Iniciar Partida Online:** Para iniciar una partida online es necesario primero tener el juego que vamos a jugar abierto, luego vamos al menú superior y hacemos click en *Online game* - *Iniciar Partida Online* , en la ventana de configuración que podemos ver en la Figura 6 debemos definir la IP a la que nos vamos a conectar y los puertos de acá jugador, así como el rol de jugador y que input delay vamos a usar.

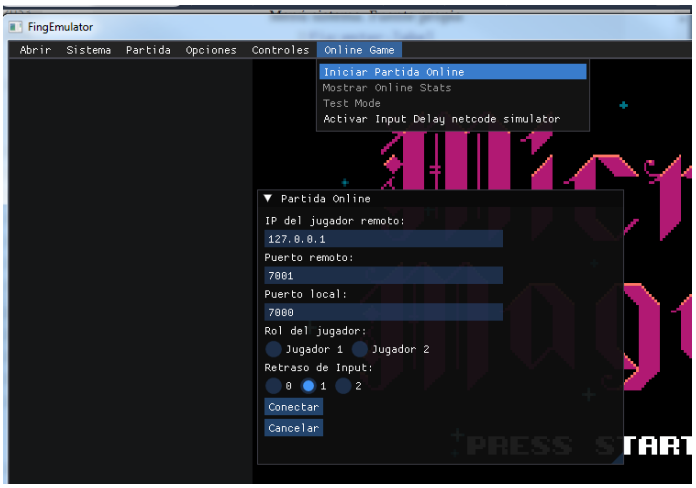


Figura 7: Configuración Partida Online. Fuente Propia

Anexo 2. Cronograma Inicial y Tiempo Final de Desarrollo

El cronograma inicial era el siguiente:

- **Mes 1 a 2:** Relevamiento, búsqueda de información, estudio de técnicas y del estado del arte.
- **Mes 3:** Diseño e implementación de la plataforma de emulación.
- **Mes 4 a 5:** Implementación del emulador NES e integración con la plataforma.
- **Mes 6:** Evaluación de desempeño de la plataforma.
- **Mes 7 a 8:** Finalización de la redacción del informe final y documentación del código.

El tiempo final de desarrollo terminó siendo el siguiente:

- Mes 1 a 4:
  - Relevamiento de requisitos.
  - Búsqueda de información y análisis de tecnologías existentes.
  - Estudio de técnicas de emulación y del estado del arte.
- Mes 5-18:
  - Desarrollo del emulador de NES.
- Mes 19-24:
  - Diseño de la arquitectura de la plataforma de emulación.
  - Implementación de la plataforma .

        - Integración del emulador NES con la plataforma de emulación.
    - Mes 25:
        - Evaluación del desempeño de la plataforma.
    - Mes 26-27:
        - Redacción del informe final del proyecto.
        - Limpieza de código para ser subido al repositorio de Facultad de Ingeniería UDELAR.

En el tiempo final de desarrollo no se tuvieron en cuenta los meses que el autor no trabajó en el proyecto.