Whyrm, videojuego para Game Boy

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
Víctor Conejero Vicente

Tutor/es:
Francisco José Gallego Durán

Julio 2022

1. Justificación y Objetivos

Las tecnologías y lenguajes de programación han evolucionado de tal forma, que, hoy en día, son accesibles para un amplio sector de la población. Cada vez es más fácil desarrollar software de forma independiente y sin disponer de vastos conocimientos.

Uno de los sectores donde más se evidencia esta evolución es en el de los videojuegos. No es difícil encontrar herramientas que permiten el desarrollo íntegro de un videojuego sin programar una sola línea de código.

Ser accesible para el mayor número posible de personas trae claros beneficios para el arte, pero, el hecho de que el número de desarrollos que aprovechan realmente las capacidades del hardware es cada vez menor es, a su vez, limitante. Es necesario un profundo conocimiento de las bases para sacar partido a las tecnologías. Como ingenieros, es nuestro trabajo conocer y saber explotar todo el potencial de las tecnologías de las que disponemos para el desempeño de nuestra profesión.

Para un programador, la mejor forma de conocer los fundamentos y capacidades de una plataforma es trabajar al más bajo nivel, en contacto con el Hardware. Aunque seguir este principio para llevar a cabo un proyecto en una consola actual es una tarea demasiado compleja para un solo programador, las consolas retro son un entorno ideal para un desarrollo de estas características.

Por estas razones, el proyecto plasmado en este documento consiste en el desarrollo en ensamblador de un prototipo de videojuego para la Game Boy (consola portátil de Nintendo lanzada en el año 1989). El objetivo es entender el funcionamiento de la consola al detalle, a la vez que se aplican de forma práctica los conocimientos adquiridos.

Para alcanzar este objetivo, será necesario abordar los siguientes aspectos:

- Analizar la Nintendo Game Boy y entender sus capacidades y limitaciones.

- Analizar librerías y frameworks existentes.

- Aprender programación en ensamblador.

- Diseñar y desarrollar el prototipo de videojuego.

2. Agradecimientos

Agradecer a Francisco José Gallego Durán por introducirme al *RetroDev* y por la divulgación constante y de libre acceso que realiza en YouTube como Profesor Retroman.

Agradecer a Alejandro Calatayud Mocholí por diseñar el sprite del personaje principal.

Agradecer a toda la comunidad de *Game Boy Development* por facilitar el acceso a la información necesaria para que este proyecto fuera posible.

3. Dedicatoria

A mi familia, mis amigos y mi tutor, por vuestra paciencia y confianza en mí.

4. Índice de contenidos

1. Justificación y Objetivos ... 3
2. Agradecimientos ... 4
3. Dedicatoria ... 5
4. Índice de contenidos ... 6
5. Terminología ... 9
6. Introducción ... 10
7. Estado del arte ... 12
   7.1 Kid Icarus: Of Myths and Monsters ... 13
   7.1.1 Aspectos de la jugabilidad ... 13
   7.1.2 Mecánicas de movimiento ... 14
   7.1.3 Aspectos técnicos ... 14
   7.2 Super Mario Land ... 17
   7.2.1 Aspectos de la jugabilidad ... 17
   7.2.2 Mecánicas de movimiento ... 18
   7.2.3 Aspectos técnicos ... 19
   7.3 Super Meat Boy ... 22
   7.3.1 Aspectos de la jugabilidad ... 23
   7.3.2 Mecánicas de movimiento ... 23
   7.4 Hollow Kinght ... 26
   7.4.1 Aspectos de la jugabilidad ... 27
   7.4.2 Mecánicas de movimiento ... 28
8. Whyrm. Diseño del juego ... 33
   8.1 Visión general ... 33
   8.2 Entidad controlada por el jugador. El caballero ... 33
   8.2.1 Mecánicas de movimiento ... 34
   8.2.2 Mecánicas de vida ... 41
   8.3 Entidades controladas por IA ... 41
   8.3.1 Abeja ... 41
   8.3.2 Sierra ... 42
   8.3.3 Lanza ... 42
   8.4 Niveles ... 43
   8.5 Transición entre niveles ... 45

8.6 Pantalla de inicio 45
8.7 Pantalla final 45
9. Introducción a la programación en Game Boy 46
9.1 Mapa de memoria de Game Boy 46
9.1.1 ROM 46
9.1.2 VRAM y OAM 48
9.1.3 WRAM 52
9.1.4 Registros de entrada/salida 53
9.1.5 HRAM 53
9.2 Introducción a los registros e instrucciones de la CPU 54
9.3 Introducción al ensamblador RGBDS 55
9.4 Creación y ensamblado de una ROM básica 57
9.4.1 Para empezar 57
9.4.2 Prerrequisitos 57
9.4.3 Fichero "Hola Mundo" 57
9.4.4 Ensamblando con RGBDS 59
9.4.5 Test del resultado con emulador BGB 60
10. Whyrm. Desarrollo 62
10.1 Hito 0. Introducción a la programación en Game Boy 63
10.2 Hito 1. Creación y manejo de entidades con motor básico 64
10.2.1 Mánager de entidades 65
10.2.2 Sistema de render 66
10.2.3 Sistema de input 68
10.2.4 Sistema de físicas 70
10.2.5 Mánager de nivel 70
10.2.6 Otros desarrollos 71
10.2.7 Estado final 72
10.3 Hito 2. Prototipo de nivel 73
10.3.1 Mánager de entidades 73
10.3.2 Sistema de colisiones 75
10.3.3 Sistema de cámara 81
10.3.4 Sistema de físicas 84
10.3.5 Mánager de estados 85
10.3.6 Sistema de input 88
10.3.7 Mánager de nivel 89
10.3.8 Estado final 91

10.4 Hito 3. Diseño visual y contenido 92
10.4.1 Diseño visual 92
10.4.2 Sistema de eventos de nivel 96
10.4.3 Sistema de animaciones 97
10.4.4 Mánager de nivel 99
10.4.5 Mánager de juego 99
10.4.6 Estado final 101
11. Conclusiones 102
12. Anexo I: Especificaciones técnicas Game Boy 103
13. Anexo II: transferencia DMA. 104
14. Anexo III: Optimización VBLANK. 107
15. Bibliografía 114

5. Terminología

Bit: en informática, unidad mínima de información, que puede tener solo dos valores (cero o uno). (Definición de Oxford Languages)

Byte: Conjunto de 8 bits que recibe el tratamiento de una unidad y que constituye el mínimo elemento de memoria direccionable de una computadora. (Definición de Oxford Languages)

Píxel: Unidad básica de una imagen digitalizada en pantalla a base de puntos de color o en escala de grises. (Definición de Oxford Languages)

Sprite: Gráfico de ordenador que puede moverse en la pantalla y manipularse como una entidad única. (Definición de Oxford Languages)

Tile: en videojuegos, pequeña imagen de forma regular que se coloca junto a otras para construir el mundo de un juego o el mapa de fondo de un nivel concreto.

Frame: en videojuegos, unidad de tiempo en cuyo transcurso se leen las entradas de mando o teclado, realizan cálculos y dibuja una imagen en la pantalla.

ROM: en videojuegos, es un fichero que contiene una copia de los datos de la memoria de solo lectura.

CPU: Siglas de la expresión inglesa central processing unit, 'unidad central de proceso', que es la parte de una computadora en la que se encuentran los elementos que sirven para procesar datos. (Definición de Oxford Languages)

Registro (CPU): memoria de alta velocidad y poca capacidad contenida en un microprocesador para almacenar los valores de datos, comandos, instrucciones o estados binarios que ordenan qué dato debe procesarse y la forma en la que se debe hacer.

Para la representación de números **hexadecimales** y **binarios** se utilizarán, a lo largo de todo el documento, los símbolos **$** y **%** respectivamente. Los números sin símbolo se deben interpretar como números decimales. Ejemplo: 66 = $42 = %01000010 .

6. Introducción

Este proyecto consiste en el desarrollo de un prototipo de **videojuego** en dos dimensiones tipo plataformas de scroll lateral.

El género **plataformas** engloba todos aquellos videojuegos en los que la jugabilidad gira en torno al control de un personaje que corre y salta sobre plataformas, suelos, salientes, escaleras u otros objetos representados en una pantalla de juego única o desplazable (de manera horizontal o vertical).

Los juegos de **scroll lateral** son aquellos en los que la acción se ve desde un ángulo de cámara lateral y la pantalla sigue al jugador mientras este se mueve a lo largo y ancho del nivel.

New Super Mario Bros. Wii (Nintendo EAD 2009), juego de plataformas en scroll lateral.

El videojuego está destinado a funcionar en la **Game Boy** de Nintendo, consola portátil lanzada en el año 1989.

El desarrollo se ha hecho íntegramente en el lenguaje ensamblador de la CPU de Game Boy y está centrado la optimización de los limitados recursos de la consola mediante la programación bajo la **arquitectura ECS** (Entity-Component-System).

La arquitectura ECS sigue las bases del **diseño orientado a datos**. El diseño orientado a datos, mayormente utilizado en el desarrollo de videojuegos, es un enfoque centrado en la ubicación de los datos, para separar y ordenar los campos de datos en función del momento en que se accede a ellos para su lectura o transformación (Fabian R 2018).

Tradicionalmente más extendido, el **diseño orientado a objetos** es un enfoque centrado en las relaciones entre los datos (Booch G 1991).

En videojuegos, frente al diseño orientado a objetos, el diseño orientado a datos supone mejoras notables de rendimiento de la CPU, facilidades para la reutilización de código gracias a su estructura simple y atómica de los módulos del proyecto y, también, facilidades en el mantenimiento (K Fedoseev et al 2020).

Para la implementación del proyecto se han seguido los planteamientos de la metodología ágil Scrum, pensada para ofrecer resultados en períodos muy limitados de tiempo e iterar sobre los resultados obtenidos.

Como reto adicional, se han intentado adaptar las mecánicas de movimiento de un juego de plataformas actual a nuestro prototipo de juego en Game Boy. Esto nos ofrece la oportunidad de discernir las mecánicas que han surgido como resultado de una evolución del medio en cuanto a experiencia jugable, de las mecánicas que son resultado de las nuevas capacidades que proporciona el hardware y software moderno.

Cabe aclarar que el objetivo de este documento no es servir como guía para el desarrollo de un videojuego, sino que tiene como objetivo plasmar el transcurso del desarrollo de un prototipo de videojuego para Game Boy programado en ensamblador por una persona sin conocimientos previos del lenguaje ni el hardware de la consola.

7. Estado del arte

Para dar forma a nuestra idea ha sido necesario estudiar, por un lado, algunos de los videojuegos de plataformas que se desarrollaron originalmente en Game Boy, y, por otro lado, algunos de los videojuegos de tipo plataformas que han destacado en los últimos años. Estos títulos sirven, además, como referencias para el diseño de nuestro juego.

En la parte de **videojuegos retro**, se han analizado los títulos *Super Mario Land* y *Kid Icarus* lanzados para Game Boy en los años 1989 y 1991 respectivamente.

En lo que respecta a **videojuegos modernos**, se han analizado los exitosos *Super Meat Boy* y *Hollow Knight* lanzados en los años 2010 y 2017.

Para facilitar el análisis técnico de los videojuegos retro nos hemos servido de las herramientas de depuración y visualización de memoria que incorpora el emulador BGB (Bircd 2021).

El análisis de los videojuegos modernos se centrará principalmente en las mecánicas de movimiento del personaje principal de cada juego.

7.1 Kid Icarus: Of Myths and Monsters

Kid Icarus se trata de videojuego tipo plataformas en scroll lateral, centrado en la exploración de niveles, donde controlamos a un personaje con la capacidad de disparar proyectiles para deshacerse de los enemigos que aparezcan.

Menú y nivel del juego Kid Icarus (Nintendo R&D1 1991).

7.1.1 Aspectos de la jugabilidad

De este título quiero destacar los que considero sus puntos fuertes:

- Divide sus niveles en una amplia sala principal y pequeñas salas secundarias con distintos encuentros (tiendas de mejoras, oleadas de enemigos, …). Esta distribución incita al jugador a la **exploración**. Recorrer la totalidad de la sala principal recompensa al jugador con puntos y mejoras para el personaje.

- La variedad de enemigos y las cantidades en las que van apareciendo generan una **acción constante** con la que se consigue atrapar la atención del jugador a lo largo de la amplitud de los niveles, sobretodo en la sala principal donde los enemigos reaparecen cada poco tiempo.

- Los niveles cambian en estética, aumentan la dificultad e introducen nuevos enemigos. Estos aspectos mantienen una **sensación de progreso** a lo largo de la partida. A esta sensación suma también una evolución de las mecánicas de movimiento y acción. Llega un punto en el que los saltos del personaje se convierten en la capacidad de volar, y los proyectiles que dispara el personaje se vuelven

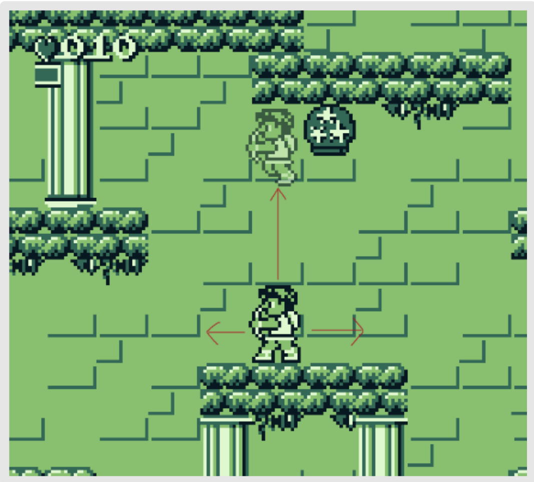
mucho más poderosos. Además, diferentes enemigos únicos de final de nivel (jefe/boss) con mecánicas y patrones de ataque diferentes, aportan frescura y nuevos aspectos a la acción.

En resumen, el principal punto **a favor** del juego es su acción, que caracteriza mayormente la jugabilidad del título: es rápida, constante y evoluciona a lo largo de todo el juego.

En contra, el diseño de niveles: la estructura que los conforma es muy similar entre niveles y no saca demasiado partido a las mecánicas de movimiento del personaje.

7.1.2 Mecánicas de movimiento

Al inicio de la partida, el personaje principal se puede mover lateralmente y con velocidad constante y, además, realizar un salto. Con el progreso de la partida, se pueden enlazar saltos de manera sucesiva en el aire en un movimiento que recuerda a volar.



7.1.3 Aspectos técnicos

La amplia sala principal de cada nivel se trata de un mapa de tiles que se carga en la memoria de vídeo por partes. La zona cargada se va actualizado con el avance por el mapa del área visible en pantalla.

Posición del personaje

Área visible en pantalla

El personaje se mueve siempre en una pequeña zona en el centro de la pantalla, a partir de estos márgenes que delimitan ésta zona, el personaje deja de moverse y es el área visible lo que se arrastra para mover al personaje por el mapa del nivel.

Posición del personaje

Área visible en pantalla

Área de movimiento del personaje

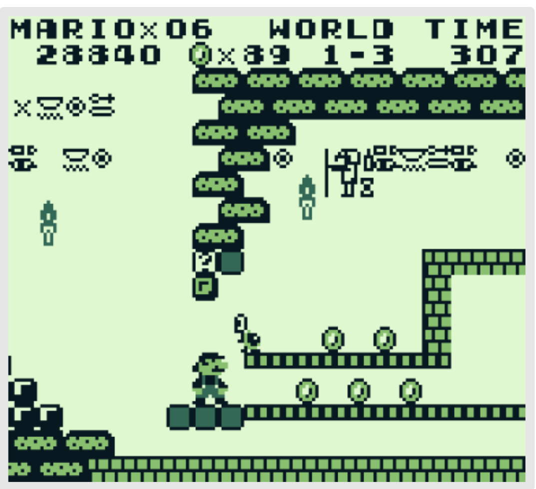
Las plataformas se construyen como una composición rectangular de tiles de 8x8 píxeles para simplificar significativamente el cálculo de colisiones del personaje con la plataforma.

El contador de corazones recogidos, el personaje principal y los enemigos son composiciones de sprites de 8x8 píxeles que se almacenan en una sección de la memoria de

vídeo diferente a la zona donde se almacena el mapa de fondo. Los elementos de esta nueva zona se dibujan en pantalla encima del mapa de fondo.

7.2 Super Mario Land

Super Mario Land es otro videojuego tipo plataformas en scroll lateral para Game Boy. A día de hoy las entregas del título Super Mario siguen considerándose la principal referencia e insignia del género plataformas.



Menú y nivel del juego Super Mario Land (Nintendo EAD 1989).

7.2.1 Aspectos de la jugabilidad

Tanto este como el resto de juegos Super Mario centran por completo su jugabilidad en el movimiento del personaje, Mario. Sin embargo, es su diseño de niveles donde alcanzan la excelencia: cada plataforma, cada enemigo y cada moneda (objeto recolectable), en todos los niveles, se encuentran posicionados con las capacidades de movimiento de Mario en mente.

El control de Mario es preciso y, sumado al recién mencionado diseño de niveles y una encantadora y pegadiza banda sonora, genera en el jugador un alto nivel de inmersión.

Los niveles de Super Mario Land no son tan amplios como los de Kid Ikarus, pero tampoco necesitan serlo, combinan dificultad y duración de tal manera que completar un nivel genera la satisfacción de haberlo finalizado, pero incita a comenzar el siguiente.

En este juego, un jugador experimentado disfrutará de llevar las mecánicas de movimiento de Mario a su máximo potencial y superar el nivel de la manera más rápida posible. Un

jugador novato, por el contrario, podrá disfrutar de explorar las diferentes secciones de cada nivel a la vez que se familiariza con las mecánicas del juego.

Hemos mencionado la exploración porque este juego tampoco se queda corto en ese aspecto:

- Salas ocultas a las que se accede a través de tuberías.
- Bloques sorpresa con distintas mejoras temporales para Mario.
- Rutas alternativas para superar cada nivel.

Es muy difícil sacar puntos negativos respecto a la jugabilidad de Super Mario Land. Quizá se pueda mencionar, para esta entrega de la saga, que las diferentes estéticas de nivel no cohesionan demasiado entre sí.

7.2.2 Mecánicas de movimiento

Mario dispone de los mismos movimientos a lo largo de toda la partida. Estos son:

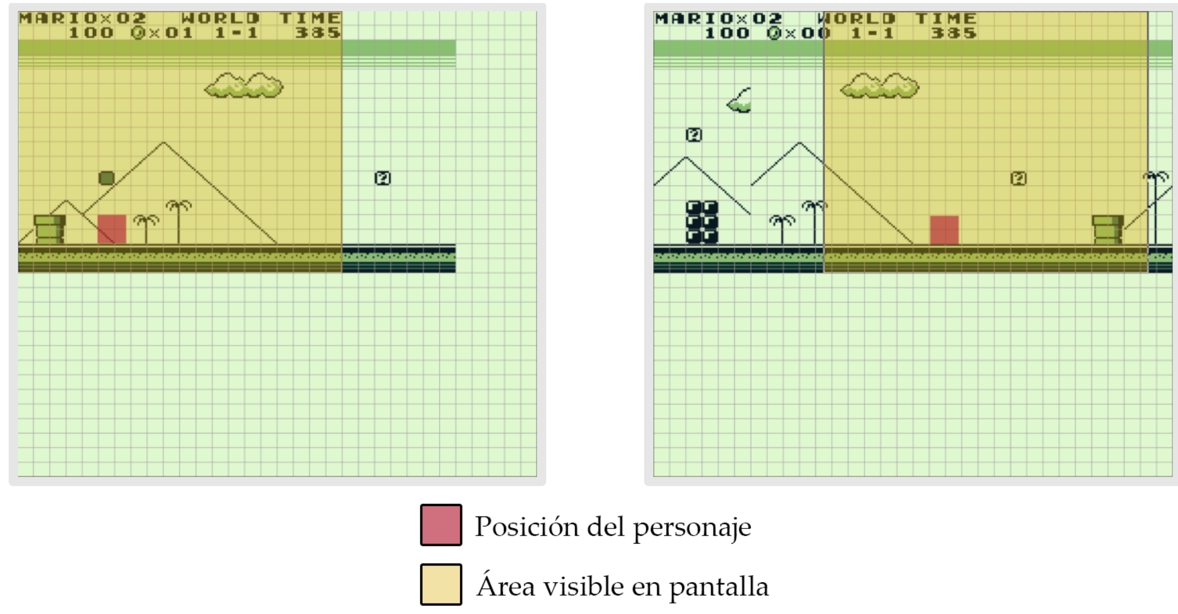
Movimiento lateral básico. Lento a izquierda y derecha. Con aceleración y deceleración.

Movimiento de salto para alcanzar plataformas. Sirve también para eliminar enemigos, que mueren cuando Mario aterriza sobre ellos.

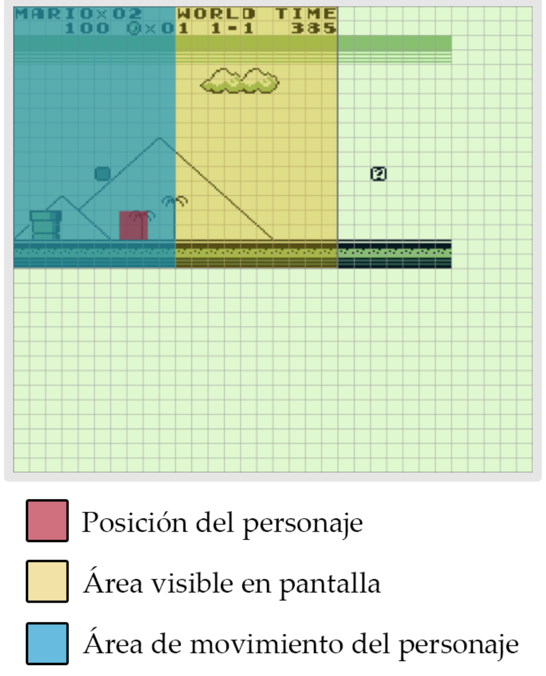
Sprint. Mario puede esprintar para moverse aún más rápido en el eje X. Además, si se salta durante el sprint Mario alcanza mayor altura.

7.2.3 Aspectos técnicos

Los niveles se cargan por partes en memoria como un mapa de tiles. Las zonas de la memoria de vídeo correspondientes a los tiles que no se están dibujando en pantalla se actualizan con la parte del nivel que sigue a la que hay actualmente dibujada.



Mario dispone de una zona fija de movimiento dentro del área visible en pantalla. El área visible nunca se mueve en el eje Y, y en el eje X se mueve solo hacia la derecha cuando Mario alcanza el punto medio en de esa área y continúa avanzando hacia la derecha.

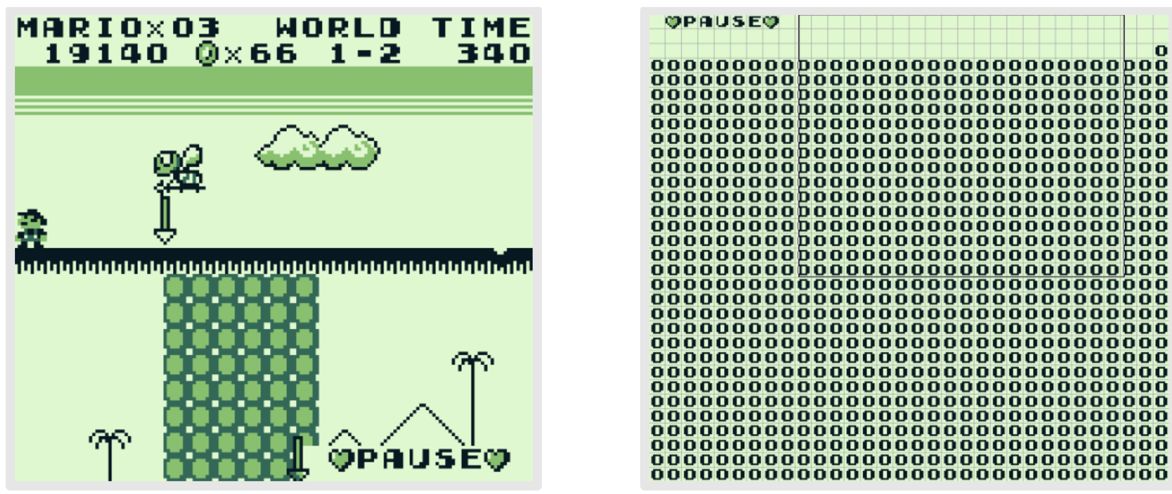


Como ya veíamos en Kid Ikarus, las plataformas se construyen como una composición rectangular de tiles de 8x8 píxeles para simplificar significativamente el cálculo de colisiones del personaje con la plataforma.

Mario y los enemigos son composiciones de sprites de 8x8 píxeles que se almacenan en una sección de la memoria de vídeo diferente a la zona donde se almacena el mapa de fondo. Los elementos de esta nueva zona se dibujan en pantalla encima del mapa de fondo.

El marcador de la zona superior de la pantalla es parte del mapa de fondo.

Cuando pausamos el juego, se dibuja en pantalla parte de un área alternativa de la memoria de vídeo con la palabra *"PAUSE"*. Los tiles de esta área se dibujan por encima del mapa de fondo, pero por debajo de Mario y los enemigos.



Nivel en pausa y estado del área alternativa de la memoria de vídeo.

7.3 Super Meat Boy

Super Meat Boy es un videojuego 2D tipo plataformas en scroll lateral lanzado inicialmente en 2010 para las plataformas Xbox y Windows. El juego se aleja de los estándares que gobernaban la época para traer de vuelta la esencia de los plataformas clásicos de principios de los 90, donde el argumento pierde peso en la obra para centrarse puramente en la jugabilidad.

Menú principal de Super Meat Boy (Team Meat 2010).

Nivel de Super Meat Boy.

7.3.1 Aspectos de la jugabilidad

Meat Boy, el personaje principal, es un pequeño cubo de carne cuya misión en el juego es rescatar a su novia, Bandage Girl, de las manos de un doctor malvado.

El jugador controlará a Meat Boy a lo largo de más de 300 pequeños pero muy exigentes **niveles**.

En estos niveles, aparte de plataformas, encontraremos pequeños enemigos y multitud de obstáculos mortales que matan instantáneamente a Meat Boy al entrar en contacto con él.

Los niveles van aumentando en **dificultad** hasta alcanzar, en las últimas fases, el punto en el que los márgenes de error para completar un nivel son mínimos o casi inexistentes, obligando al jugador a dominar el movimiento de Meat Boy a la perfección.

El movimiento de Meat Boy es mucho más rápido que en el resto de plataformas al uso, y es que la **velocidad** cobra un importante papel en este juego, ya que se recompensa al jugador con puntuaciones más altas por finalizar un nivel de la forma más rápida posible. Las puntuaciones sirven más tarde para desbloquear el acceso a niveles extra.

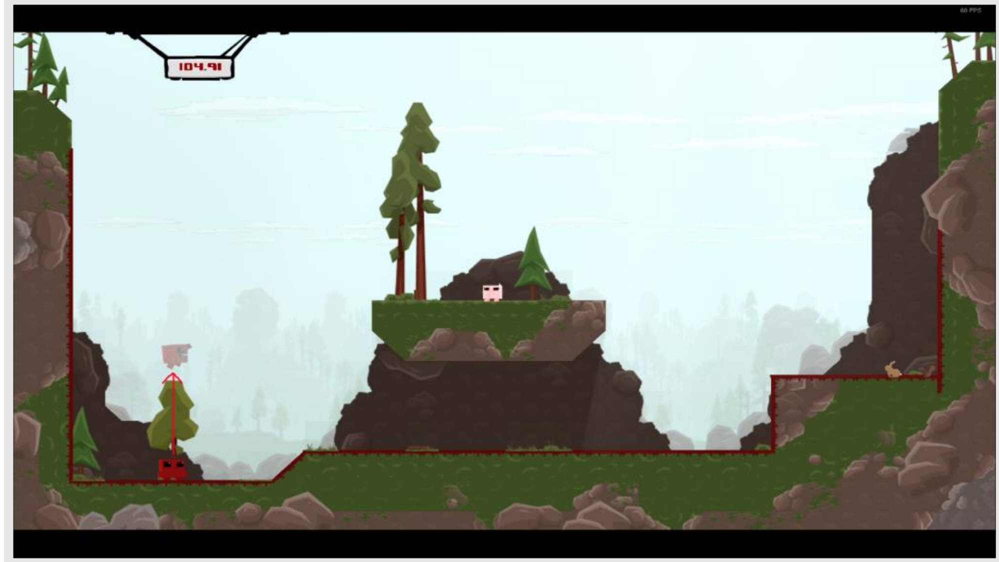
Superar cada nivel se acaba convirtiendo en todo un reto y, por ello, el elemento insignia de Super Meat Boy es su dificultad, que a la vez se convierte en uno de sus mayores atractivos y uno de los factores decisivos por los que los jugadores menos experimentados dejen de jugarlo.

7.3.2 Mecánicas de movimiento

Meat Boy dispone de los mismos movimientos a lo largo de toda la partida. Estos son:

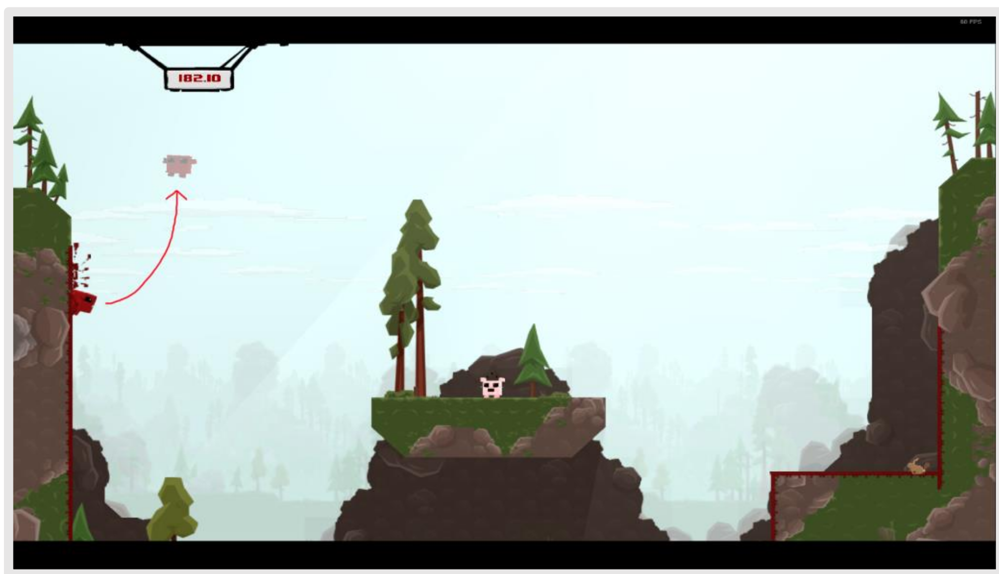
Movimiento lateral básico. Lento a izquierda y derecha. Con aceleración, pero parada en seco.

Movimiento de salto para alcanzar plataformas.



Sprint. Meat Boy puede esprintar para moverse mucho más rápido en el eje X.

Rebote con pared. Si Meat Boy se encuentra pegado a un muro podrá realizar un movimiento similar al salto, aunque no tenga ninguna plataforma a sus pies.



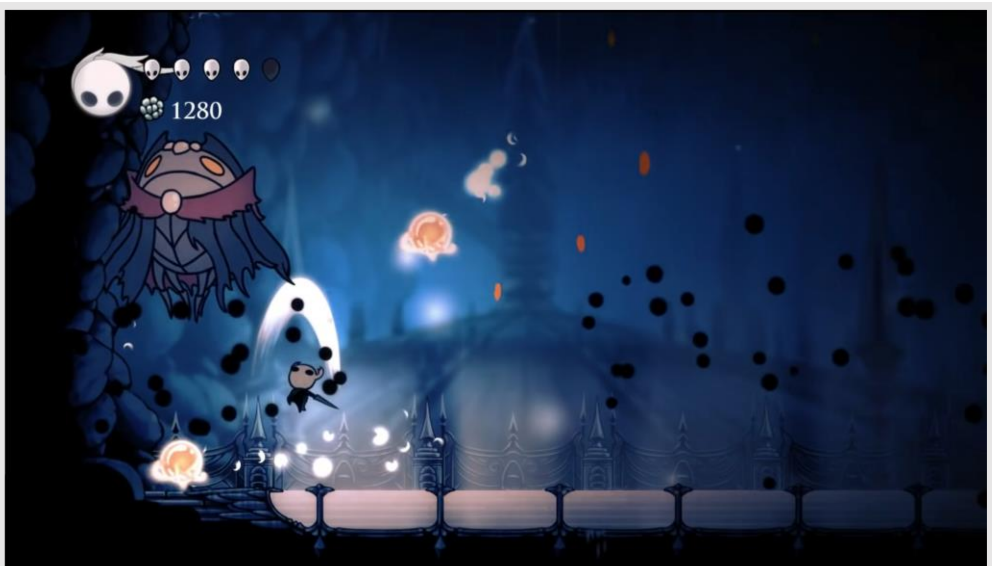
7.4 Hollow Kinght

Hollow Knight es un videojuego 2D tipo acción y plataformas en scroll lateral lanzado inicialmente en 2017 para Windows. Con un más que destacable apartado artístico y sonoro y alrededor de tres millones de copias venidas, es uno de los juegos de plataformas más exitosos de los últimos años.

Este título es un claro ejemplo de la evolución del género y el medio, pues, frente a los juegos de Game Boy anteriormente vistos, que no tenían mucho más remedio que centrarse puramente en la jugabilidad a la que, en ocasiones, se acompañaba con una muy sencilla historia, Hollow Knight es capaz de combinar diferentes géneros (principalmente acción y plataformas) a la vez que cuenta una historia completa en la que intervienen numerosos personajes y en un mundo enorme y que se siente vivo.

Menú principal de Hollow Knight (Team Cherry 2017).

Hollow Knight, zona de plataformas.

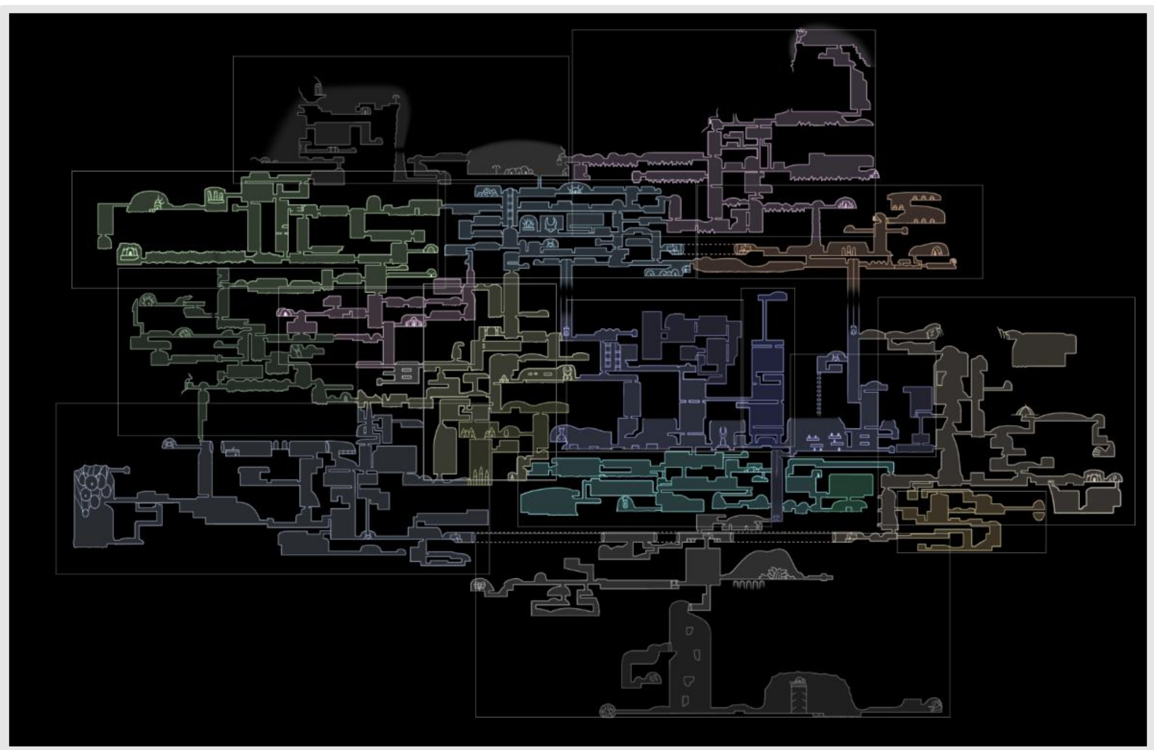


Hollow Knight, batalla contra jefe de final de zona.

7.4.1 Aspectos de la jugabilidad

En este juego, el jugador controla al **Caballero**, un guerrero insectoide sin nombre, en su aventura para explorar **Hallownest**, un reino caído plagado de una misteriosa enfermedad sobrenatural. El juego se desarrolla en diversas localizaciones subterráneas, y cuenta con personajes amistosos con los que dialogar y numerosos enemigos en forma de bichos y jefes de final de zona a los que combatir.

Hollow Knight cuenta con 16 amplias zonas o **niveles** de diferente estética pero que cohesionan perfectamente y conforman un inmersivo mundo.



Mapa de zonas de Hollow Knight.

Cada zona posee una gran variedad de diferentes enemigos con diferentes tipos de ataque. Además, cada una de las zonas dispone de uno o más jefes finales en cuyos enfrentamientos se exige al jugador un claro domino de las mecánicas de movimiento y combate.

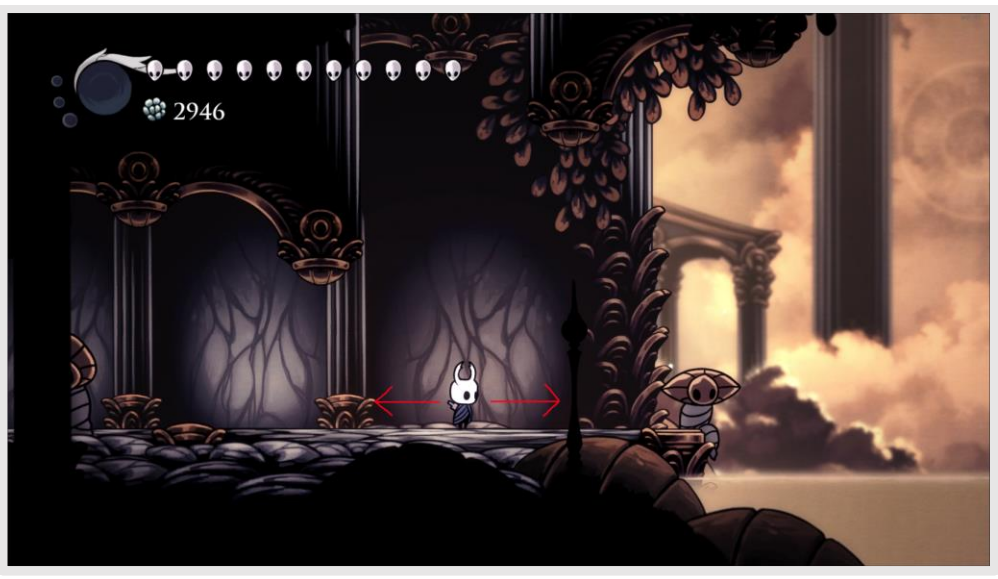
A lo largo de estos niveles el jugador irá encontrándose con diferentes objetos coleccionables que desbloquean el acceso a nuevas zonas, nuevas habilidades para el personaje y pequeños relatos de la historia del reino.

Con el **avance de la partida** el jugador adquiere nuevos recursos para la exploración y el combate como nuevos movimientos, nuevos tipos de ataque y mejoras de salud para el personaje.

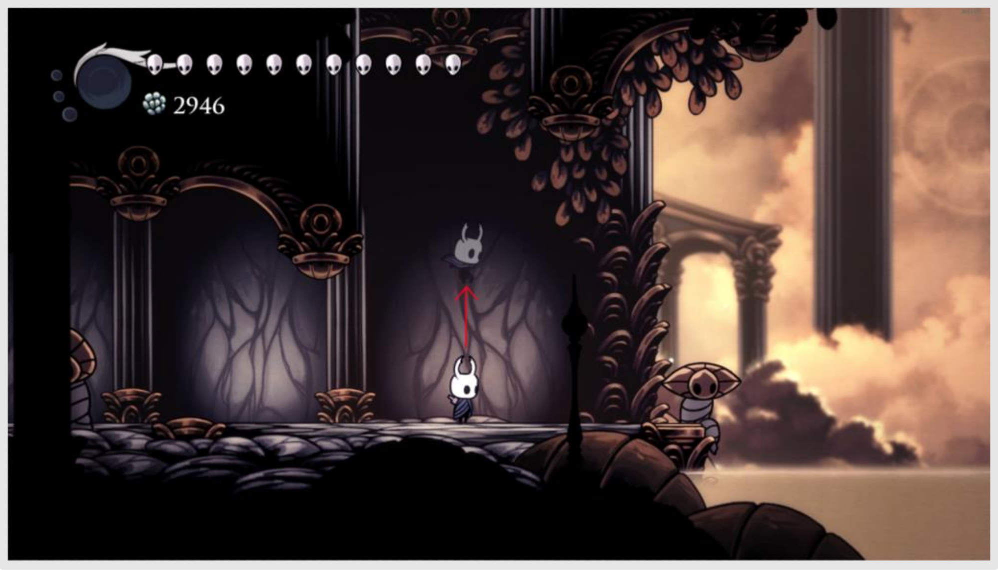
7.4.2 Mecánicas de movimiento

Al final de la partida, adquiridas todas las mejoras para el personaje, los movimientos disponibles son:

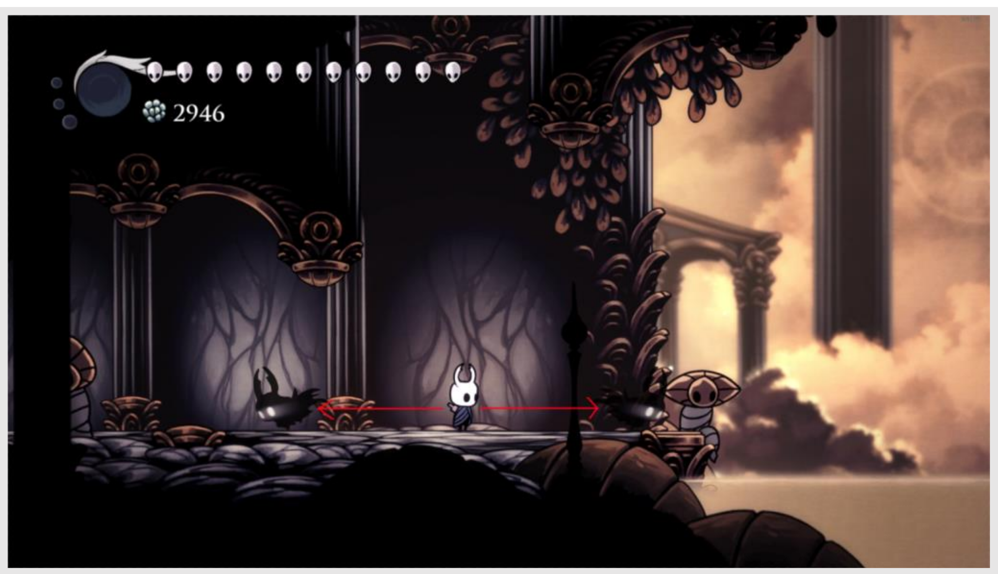
Movimiento lateral básico. Lento a izquierda y derecha. Con velocidad constante.



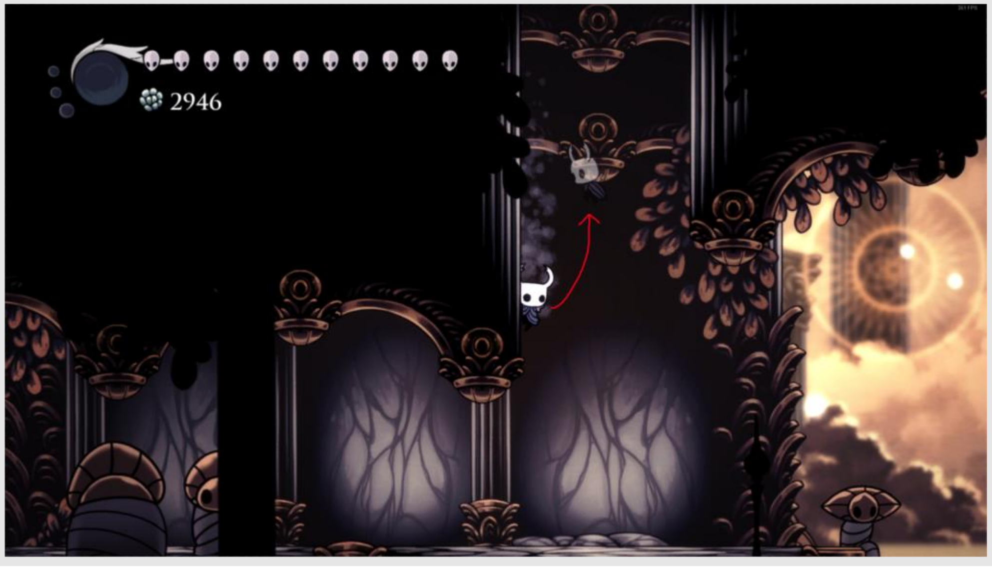
Movimiento de salto para alcanzar plataformas y evitar ataques enemigos.



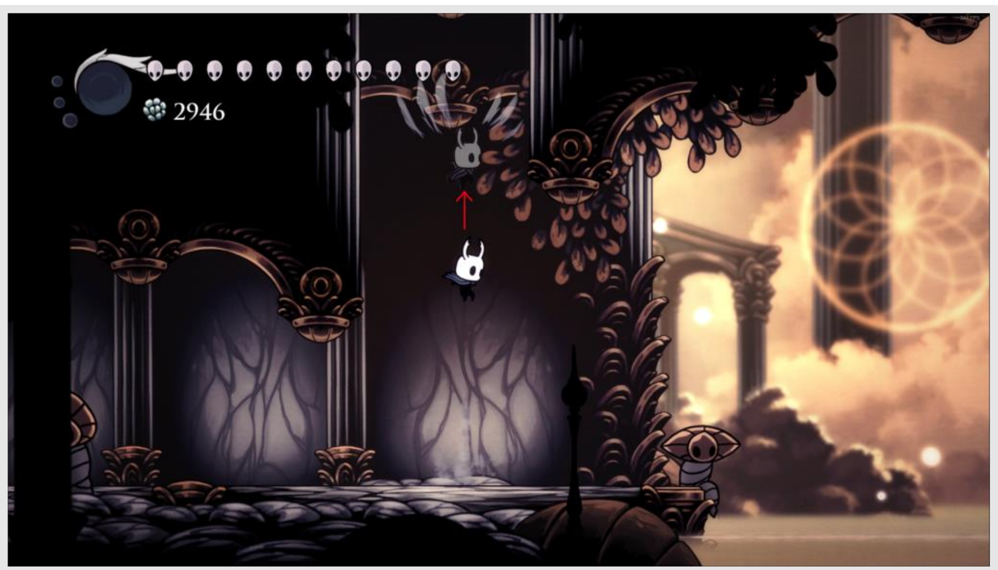
Embestida. Movimiento rápido en X hacia la dirección a la que mira el personaje.



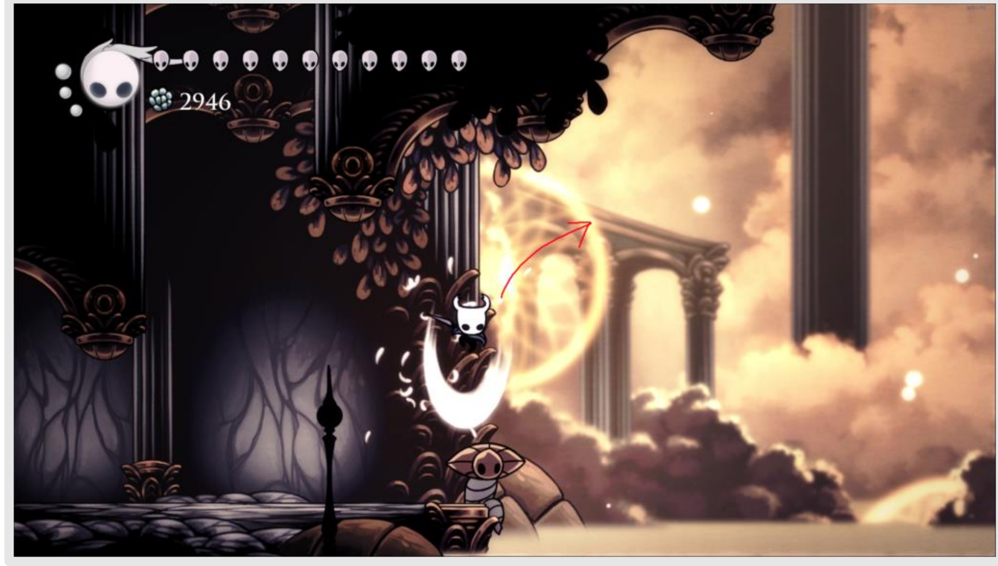
Rebote con pared. Si el personaje principal se encuentra pegado a un muro podrá realizar un movimiento similar al salto, aunque no tenga ninguna plataforma a sus pies.



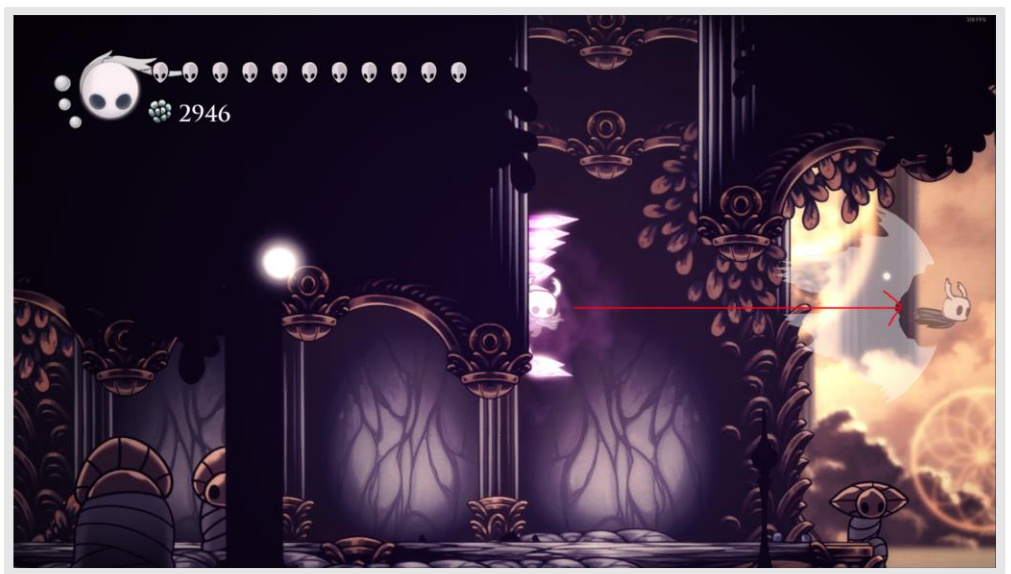
Doble salto. En el aire, el personaje podrá saltar de nuevo una única vez.



Rebote aéreo. En el aire, si se realiza un movimiento de ataque hacia abajo y golpea a un enemigo, el personaje principal saldrá impulsado como si de un salto se tratase.



Embestida cargada. Si el personaje principal se encuentra pegado a un muro podrá detenerse unos segundos para cargar energía que se libera en forma de embestida en el eje X. Es un movimiento que no se detiene a no ser que el personaje colisione con algún obstáculo o enemigo, por lo que sirve para recorrer grandes distancias rápidamente.


2946

8. Whyrm. Diseño del juego

Este punto tiene como objetivo describir el diseño del prototipo de juego que se va a desarrollar. En los diferentes apartados se especificarán todos los aspectos de su jugabilidad y los elementos que lo componen.

8.1 Visión general

El juego, de título provisional "Whyrm", se trata de un plataformas dos dimensiones de *scroll* lateral para la consola Game Boy.

El jugador controlará a una entidad, el caballero, con distintas mecánicas de movimiento de las que se tendrá que servir para superar diferentes secciones o niveles repletas de obstáculos para dificultar su paso. Como veremos más adelante, en estos niveles encontraremos también entidades controladas por IA que servirán, a la vez, para aumentar la dificultad del recorrido y como útiles elementos para avanzar en el nivel. La combinación de mecánicas de movimiento y entidades ofrecerán al jugador numerosas opciones y vías para completar los niveles.

Concepto del título del juego con paleta de colores Game Boy.

8.2 Entidad controlada por el jugador. El caballero

El **caballero** será nuestra única entidad controlada por el jugador y tendrá dimensiones de 8x13 píxeles para el cálculo de físicas. Visualmente podrá tener dimensiones de mayor tamaño, sin superar el límite de 16x16 píxeles.

Boceto del caballero.

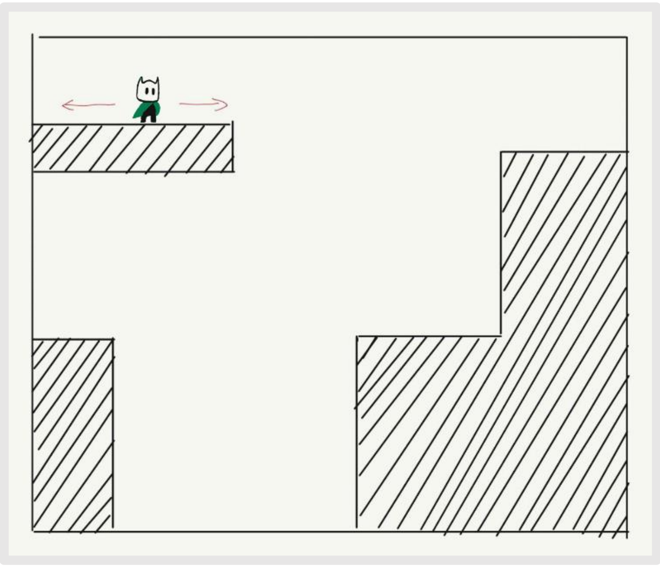
8.2.1 Mecánicas de movimiento

Desde el inicio de la partida, el caballero dispondrá de todas sus mecánicas de movimiento. Estas son:

- **Movimiento lateral**. Es un movimiento lento a izquierda y derecha que el caballero podrá realizar a no ser que colisione con algún obstáculo en el eje X o el personaje se encuentre en un estado de movimiento que no lo permita. Si el caballero está muerto no se le aplicará este movimiento ni ningún otro.

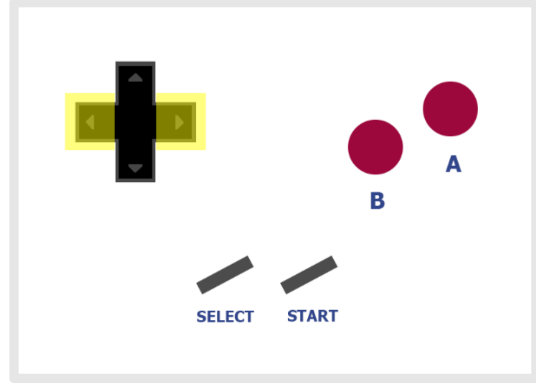
  Es el movimiento básico por excelencia en los videojuegos tipo plataformas en scroll lateral.

  - Impide: no impide nada.
  - Interrumpe: no interrumpe nada.

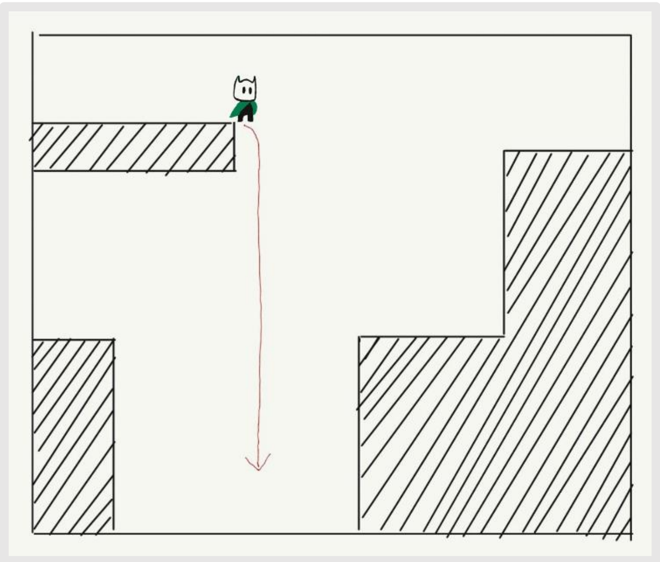


Boceto del movimiento lateral del caballero en un nivel.

En amarillo los botones para iniciar el movimiento:



- **Gravedad**. Si no existen obstáculos en el eje Y con los que los pies del caballero colisionen, se aplicará, para este, un movimiento de descenso en el eje Y. El movimiento será acelerado hasta alcanzar un valor máximo de velocidad de caída. Existen estados de movimiento del caballero en los que no se aplica la gravedad.
    - Impide: Salto.
    - Interrumpe: no interrumpe nada.



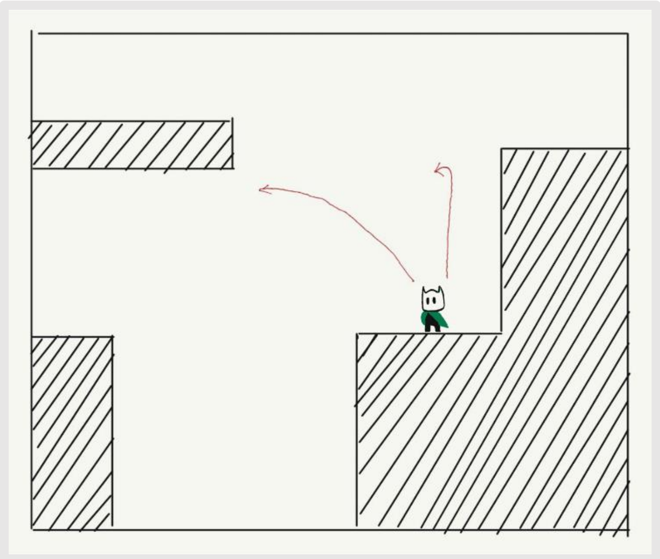
Boceto de la gravedad en un nivel.

- **Salto**. Es el movimiento que impulsa al caballero de manera ascendente en el eje Y. Será también un movimiento acelerado hasta alcanzar un máximo de velocidad. Además, este movimiento está limitado a recorrer una distancia en Y concreta. El movimiento se detendrá si la parte superior de la cabeza del caballero colisiona con

algún obstáculo durante el ascenso. Algunos estados de movimiento también pueden interrumpir el salto. Habrá también estados que no permitan iniciarlo.

Es un movimiento imprescindible para un videojuego de plataformas.

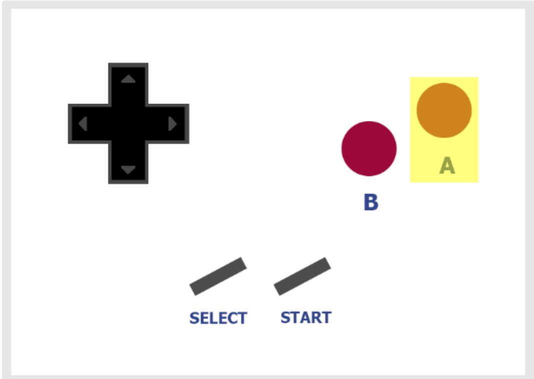
- Impide: Gravedad.
- Interrumpe: no interrumpe nada.



Boceto del salto del caballero en un nivel.

En la imagen se puede observar el salto del caballero cuando se suma, o no, movimiento lateral durante el salto.

En amarillo los botones para iniciar el movimiento:

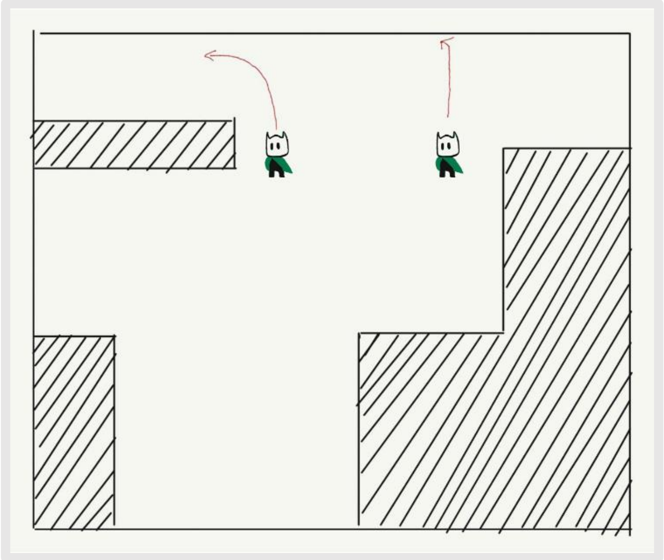


- **Doble Salto**. En el aire (sin colisiones en ninguno de los ejes), el caballero podrá realizar un movimiento similar al salto. No se puede realizar la acción de doble Salto consecutivamente. Para poder volver a ejecutar un doble Salto, el jugador tendrá

que realizar antes algún otro movimiento (sin incluir gravedad y movimiento lateral).

Es un movimiento poco común en videojuegos retro pero muy usado actualmente en videojuegos de todo tipo de géneros.

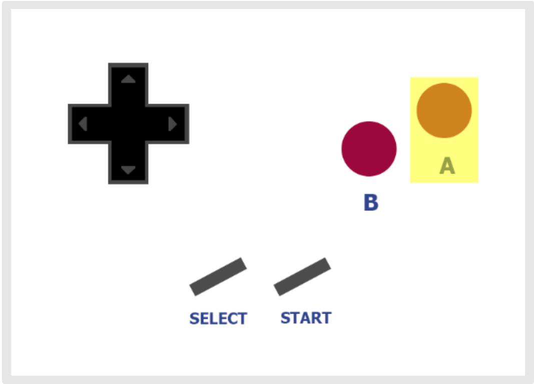
- Impide: Gravedad, Salto.
- Interrumpe: Gravedad.



Boceto del doble salto del caballero en un nivel.

En la imagen se puede observar el doble salto del caballero desde las posiciones donde habría finalizado el salto del boceto anterior.

En amarillo los botones para iniciar el movimiento:

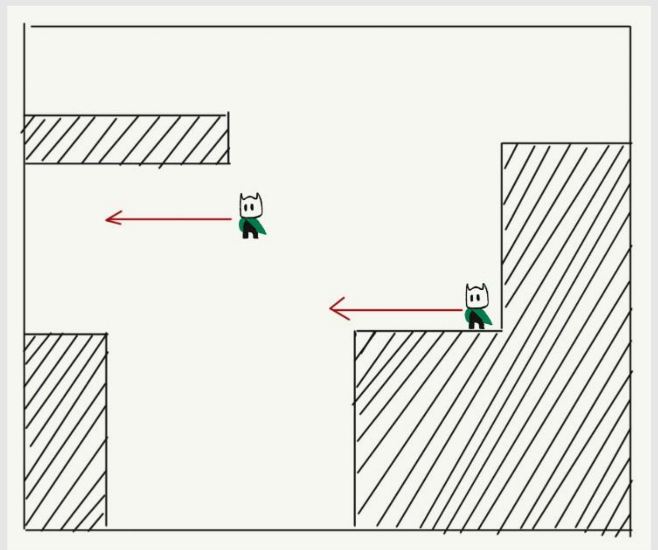


- **Embestida**. Se trata de un desplazamiento rápido y distancia fija en el eje X. Este movimiento se realizará en la dirección hacia la que el caballero mira en el momento

en el que se presiona el botón correspondiente. Este movimiento se interrumpe ante una colisión en el eje X.

Es un movimiento poco común en videojuegos retro.

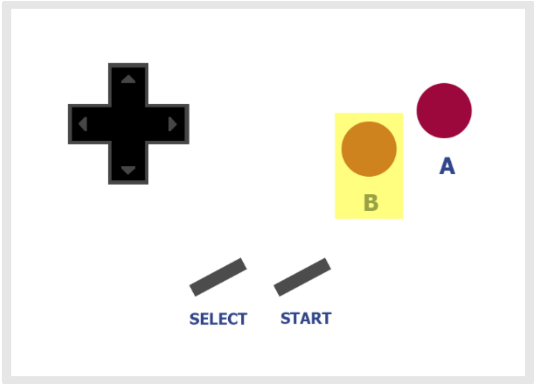
- Impide: Gravedad, Salto, Doble Salto.
- Interrumpe: Gravedad, Salto, Doble Salto.



Boceto de la embestida del caballero en un nivel.

En la imagen se muestra la embestida desde el aire (sin colisiones del caballero en ninguno de los ejes) y desde el suelo (colisión en eje Y del caballero con obstáculo).

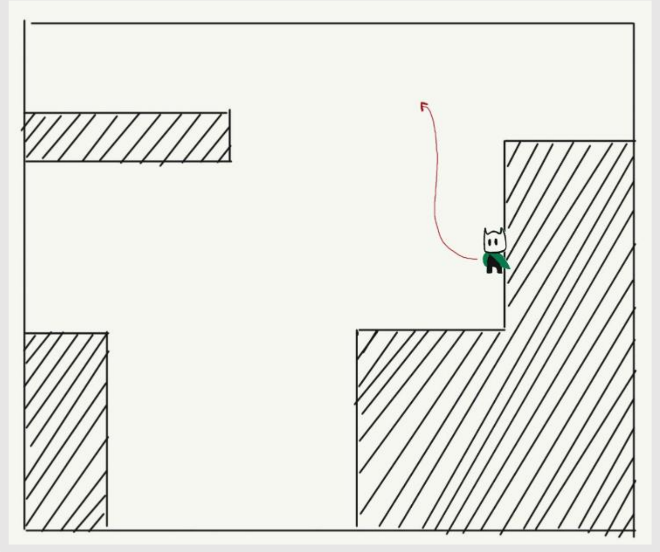
En amarillo los botones para iniciar el movimiento:



- **Rebote con pared**. Mientras el caballero se encuentre colisionando en el eje X con algún obstáculo, podrá realizar un movimiento similar al salto en cuanto desplazamiento y comportamiento.

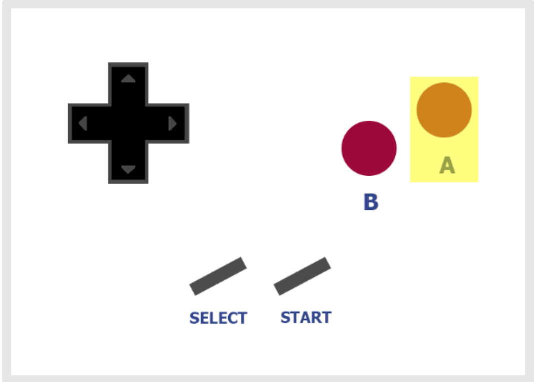
Es un movimiento muy poco común en videojuegos retro.

- Impide: Salto, Gravedad.
- Interrumpe: Gravedad.



Boceto del rebote con pared del caballero en un nivel.

En amarillo los botones para iniciar el movimiento:



- **Rebote aéreo**. Al presionar el botón correspondiente, el caballero iniciará la primera fase de este movimiento. Esta primera fase (sin desplazamiento en ninguno de los ejes) dura unos pocos frames desde que se presiona el botón. Si durante esos frames el caballero colisiona con alguna otra entidad, desencadenará la segunda fase del movimiento. Si durante la primera fase no se produce colisión con otras entidades, el rebote se detendrá sin haber producido desplazamiento y habrá que esperar unos

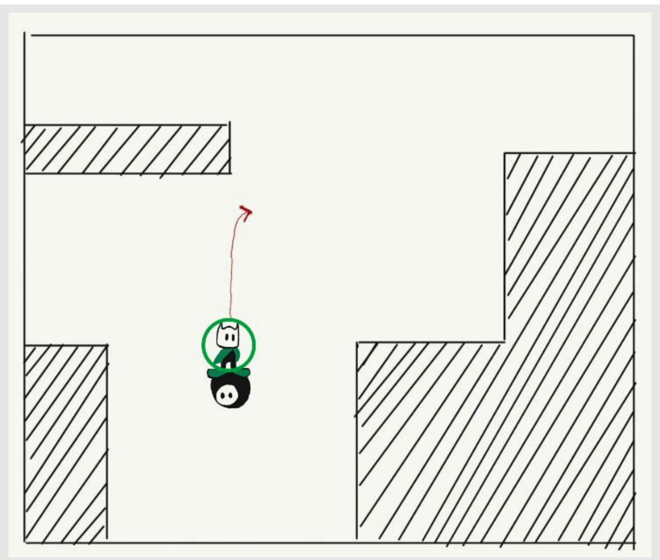
instantes hasta que el jugador pueda repetirlo presionando el botón correspondiente.

Es un movimiento poco común en videojuegos retro. Aunque el clásico Super Mario ya introduce una pequeña versión de este movimiento al aterrizar sobre los enemigos, no es un rebote lo suficientemente alto para superar obstáculos.

La segunda fase consiste en un movimiento similar al salto en cuanto a desplazamiento y comportamiento.

La primera fase del rebote aéreo no impide ni interrumpe ningún tipo de movimiento. La segunda fase, sin embargo:

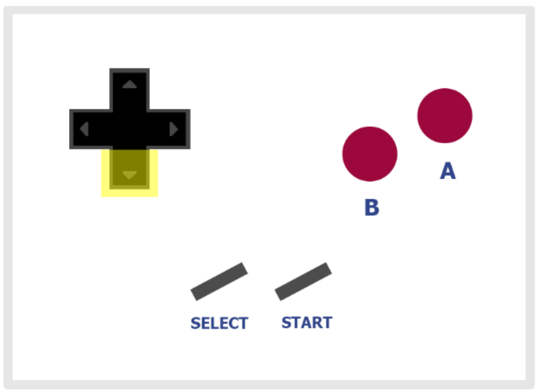
- Impide: Gravedad, Salto.
- Interrumpe: Gravedad, Salto, Doble Salto, Embestida, Rebote con pared.



Boceto de la embestida del caballero en un nivel.

El círculo verde que rodea al caballero en la imagen representa que este se encontraba en la primera fase del rebote aéreo cuando se ha producido la colisión con la otra entidad. El desplazamiento que indica la flecha roja corresponde al aplicado por la segunda fase del rebote aéreo.

En amarillo los botones para iniciar el movimiento:



8.2.2 Mecánicas de vida

El caballero morirá siempre de manera instantánea al recibir cualquier tipo de daño. Cuando el caballero muera, renacerá instantes después desde el punto inicial del nivel en el que ha muerto. Ese punto inicial estará definido en los propios datos del nivel y variará entre niveles. El juego almacenará el dato con el número de muertes del caballero acumulado a lo largo de toda la partida.

8.3 Entidades controladas por IA

Dispondremos de tres tipos de entidades no controlables por el jugador: abejas, sierras y lanzas. Todas compartirán tamaño y tendrá que ser mayor al del personaje, menor a 16x16 píxeles y de igual ancho que alto. Si el caballero colisiona con alguna de estas entidades recibirá daño y morirá, excepto si el caballero se encuentra en la fase inicial de "Rebote aéreo", en cuyo caso se iniciará el desplazamiento correspondiente a la segunda fase de ese movimiento.

8.3.1 Abeja

La abeja es una entidad estática, no se desplaza en ninguno de los ejes. Si el caballero colisiona con esta entidad durante la primera fase del "Rebote aéreo", esta entidad pasará a estar muerta. Pasados unos instantes, la abeja muerta volverá a su estado normal (viva) sin modificar su posición en el mapa.

Boceto de abeja.

8.3.2 Sierra

La sierra es una entidad con desplazamiento en uno de los ejes (exclusivamente en uno de ellos para evitar desplazamientos diagonales). Tendrá velocidad constante y cambiará de dirección al colisionar con obstáculos del mapa específicos que delimitan su recorrido. En ningún caso la sierra morirá.

Boceto de sierra.

8.3.3 Lanza

La lanza es una entidad similar a la sierra. Las únicas diferencias con la sierra son que tendrá una velocidad mayor, pero en recorridos de distancia mucho menor. Además, al encontrarse con su obstáculo delimitador correspondiente, la lanza se tomará unos instantes de parada antes de cambiar la dirección de su desplazamiento. Las lanzas tampoco morirán.

Boceto de lanza.

8.4 Niveles

Cada uno de los niveles tendrá un tamaño de 256x256 píxeles y tendrán distribuidos en ese espacio una serie de obstáculos (paredes, suelos, techos, plataformas, …), un punto de inicio y un punto de fin.

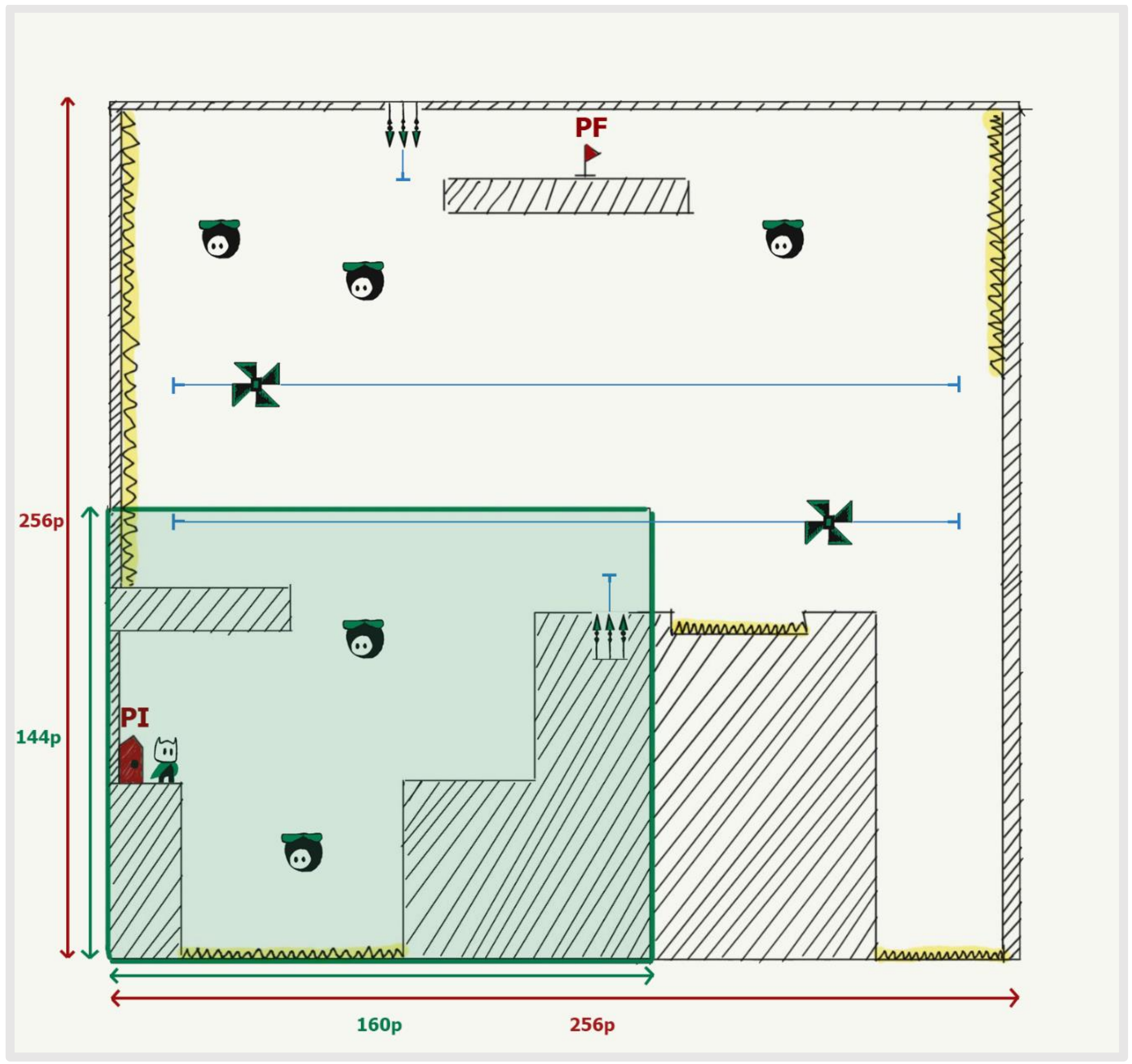
Los obstáculos podrán ser o no mortales, la diferencia reside en que, ante una colisión del caballero con ese obstáculo, este último detendrá su desplazamiento o lo matará.

El punto de inicio es la posición del nivel en la que el caballero comienza el nivel y en la que reaparece tras morir.

El punto de fin es la posición del nivel que el caballero tendrá que alcanzar para avanzar al siguiente nivel.

Adicionalmente, cada nivel contendrá un conjunto de entidades controladas por la IA. El máximo número de estas entidades en cada nivel es de nueve y su disposición y tipo variará entre niveles.

El área visible en pantalla (o cámara) del nivel tiene un tamaño de 160x144 píxeles y se desplazará junto al caballero a lo largo y ancho del nivel, manteniendo, cuando sea posible, al caballero en su centro.



Boceto de nivel.

El boceto muestra el espacio de un nivel completo y, en verde, el área visible en pantalla.

Las áreas rellenadas con rayas diagonales negras representan los obstáculos no mortales, y las resaltadas en amarillo son los obstáculos mortales.

En la imagen, las letras **PI** referencian al punto inicial del nivel y las letras **PF** al punto final del mismo.

A lo largo del nivel hay situadas diferentes entidades controladas por la IA. Las líneas azules representan, para este tipo de entidades, el espacio en el que se moverán.

8.5 Transición entre niveles

Tras la finalización de un nivel y previo al inicio del siguiente, se mostrará una pantalla donde el jugador podrá ver el número del nivel que ha completado, el número de muertes del caballero para ese nivel y el número total de muertes durante toda la partida.

8.6 Pantalla de inicio

Cada vez que se cargue el juego en la consola se mostrará, de inicio, una pantalla con el título del juego e indicaciones para comenzar/retomar la partida.

8.7 Pantalla final

Al superar todos los niveles se mostrará una pantalla felicitando al jugador y mostrando el número total de muertes del caballero durante toda la partida.

9. Introducción a la programación en Game Boy

9.1 Mapa de memoria de Game Boy

Tabla representativa de la memoria de Game Boy (Nintendo 1999):

Dirección de inicio	Dirección final	Descripción
$0000	$3FFF	**ROM** banco 0. 16 kB
$4000	$7FFF	**ROM** banco intercambiable. 16 kB
$8000	$9FFF	RAM de vídeo, **VRAM**. 8 kB
$A000	$BFFF	**RAM del cartucho** (si tiene). 8 kB
$C000	$CFFF	RAM de trabajo, **WRAM**. Banco 0. 4 kB
$D000	$DFFF	**WRAM**. Banco intercambiable. 4 kB
$E000	$FDFF	**Espejo de WRAM**, no usable. 8 kB
$FE00	$FE9F	Memoria de atributos de objetos, **OAM**.
$FEA0	$FEFF	Área no usable.
$FF00	$FF7F	**Registros de entrada/salida**.
$FF80	$FFFE	RAM alta o High RAM (**HRAM**)
$FFFF	$FFFF	Registro para habilitar las interrupciones (**IE**)
		64 kB de espacio total.

9.1.1 ROM

La ROM es un espacio de 32 kB de memoria dividida en dos áreas de 16 kB. La primera, ROM banco 0, es fija y siempre se puede acceder a ella. La segunda sirve como acceso al resto de bancos del cartucho (el número de estos bancos varía entre cartuchos). Cada uno

de esos bancos es un bloque de 16 kB a los que no podemos acceder de manera simultánea. En un momento determinado, solo podremos acceder a uno de esos bancos mediante el área de memoria comprendida entre las direcciones $4000 y $7FFF, y será necesario lanzar una instrucción al procesador para modificar el banco al que queremos acceder.

$0000	Dirección de inicio	Dirección final	Descripción
	$0000	$00FF	Vectores de salto (RST e interrupciones).
	$0100	$0103	Punto de entrada al programa.
	$0104	$014F	Cabecera del cartucho.
	$0150	$3FFF	Resto de la ROM banco 0.
	$4000	$7FFF	ROM banco intercambiable.

$7FFF

Vectores de salto (RST e interrupciones). Son una serie de direcciones de memoria a las que nuestro programa podrá saltar mediante la instrucción RST de la CPU o automáticamente cuando se detecte algún tipo de interrupción del procesador (siempre y cuando se hayan habilitado previamente las interrupciones). Estas direcciones son:

- Para la instrucción RST: $0000, $0008, $0010, $0018, $0020, $0028, $0030, $0038
- Según el tipo de interrupción: $0040, $0048, $0050, $0058, $0060

Sobre las interrupciones, entramos más en detalle en el **Anexo III: Optimización VBLANK**.

Punto de entrada al programa. Son tres bytes para la instrucción de salto que especifica a la CPU la dirección de la ROM donde comienza el código de nuestro programa.

Cabecera del cartucho. Esta área contiene sumas de comprobación, información sobre el chip MBC utilizado, los tamaños de la ROM y la RAM, etc. La mayoría de los bytes de esta área deben ser especificados correctamente.

9.1.2 **VRAM y OAM**

VRAM

$8000	Dirección de inicio	Dirección final	Descripción
	$8000	$9FFF	Tabla de datos de **Tiles**.
	$9800	$9BFF	Por defecto, **mapa de fondo**.
	$9C00	$9FFF	Por defecto, **ventana**.

$9FFF

Al igual que en otros sistemas retro, los píxeles no se manipulan individualmente, ya que esto sería muy costoso para la CPU. En su lugar, los píxeles se agrupan en cuadrados de 8x8, denominados "tiles, a menudo considerados como la unidad base en los gráficos de Game Boy.

Tabla de Tiles. Es el área de la memoria donde se almacenan cada uno de los tiles.

Un tile en Game Boy ocupa 16 bytes, donde cada línea está representada por 2 bytes. Es deducible entonces que la información del color de cada píxel es dada por 2 bits.

La paleta por defecto está compuesta por cuatro colores que corresponden a los siguientes valores:

Color	Valor Binario
Negro	11
Verde Oscuro	10
Verde Claro	01
Blanco	00

Para cada línea, el primer byte especifica el bit menos significativo del ID de color de cada píxel, y el segundo byte especifica el bit más significativo. En ambos bytes, el bit 7 representa el píxel más a la izquierda, y el bit 0 el más a la derecha.

			abcdefgh		ijklmnop		ia jb kc ld me nf og ph
$3C	$7E		00111100		01111110		00 10 11 11 11 11 10 00
$42	$42		01000010		01000010		00 11 00 00 00 00 11 00
$42	$42		01000010		01000010		00 11 00 00 00 00 11 00
$42	$42		01000010		01000010		00 11 00 00 00 00 11 00
$7E	$5E	=	01111110	+	01011110	=	00 11 01 11 11 11 11 00
$7E	$0A		01111110		00001010		00 01 01 01 11 01 11 00
$7C	$56		01111100		01010110		00 11 01 11 01 11 10 00
$38	$7C		00111000		01111100		00 10 11 11 11 10 00 00
			abcdefgh		ijklmnop		ia jb kc ld me nf og ph

Ejemplo de datos de un Tile (Antonio Niño Díaz et al 2021).

Mapa de fondo, área compuesta por un mapa de tiles, una gran cuadrícula de tiles. Sin embargo, los tiles no se escriben directamente en los mapas de tiles, sino que simplemente contienen referencias a los tiles de la tabla de tiles. Esto hace que la reutilización de tiles sea poco costosa, tanto en tiempo de CPU como en espacio de memoria.

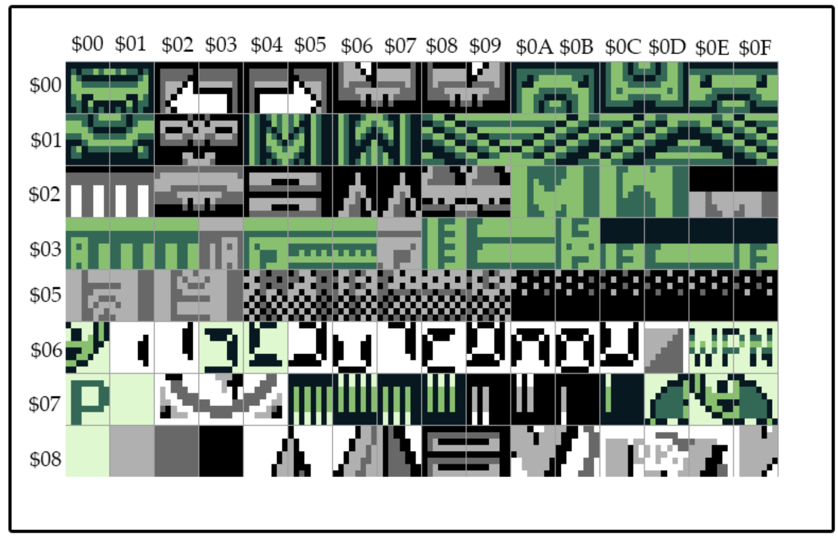
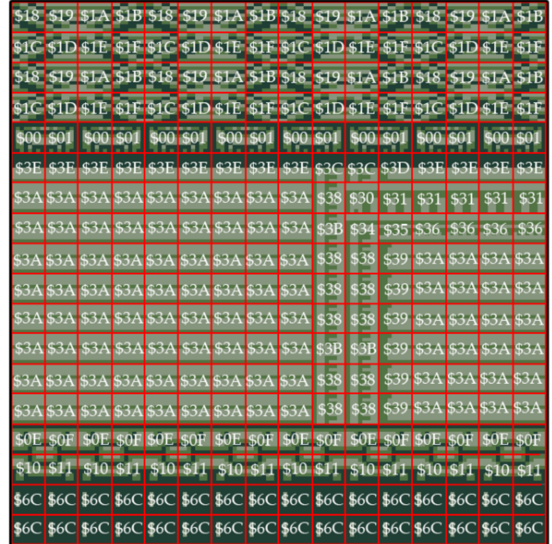
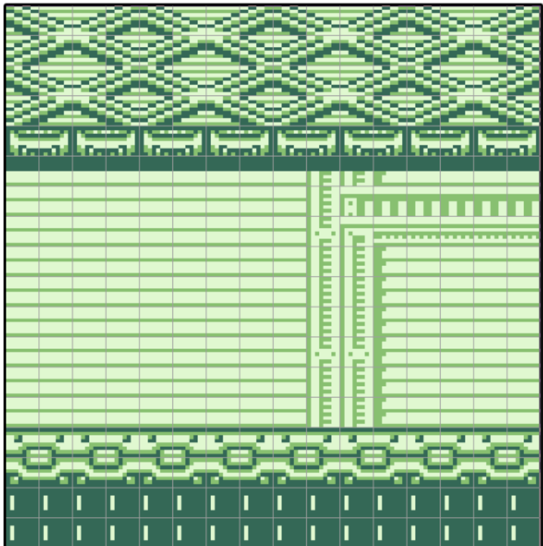


Tabla de Tiles

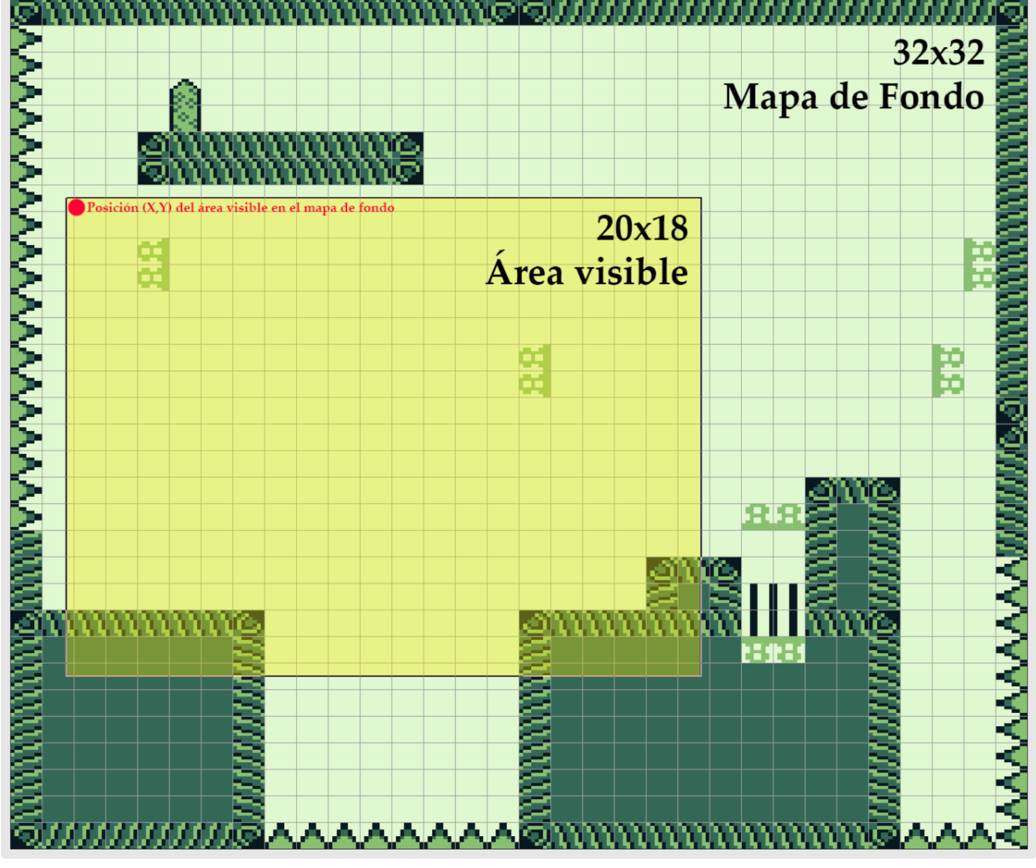


Mapa de fondo (Datos)



Mapa de fondo (Visual)

Esta área tiene un tamaño de 256x256 píxeles o, lo que es lo mismo, 32x32 tiles. En pantalla, sin embargo, sólo podemos visualizar un área más pequeña, de 20x18 tiles. Esta área visible almacena en 2 bytes su posición en el mapa de tiles y basta con modificar esos bytes para arrastrar el área por el espacio del mapa de tiles.



Ventana. La ventana, al igual que le mapa de fondo, es un mapa de tiles. Mediante un registro de control, podremos activarla en una posición concreta para mostrar ciertos tiles de la ventana sobre el mapa de fondo.

OAM

$FE00	Dirección de inicio	Dirección final	Descripción
	$FE00	$FE9F	Memoria de atributos de objetos, **OAM.**

$FE9F

OAM es el área de la memoria dónde se almacenan los datos de cada uno de los objetos o sprites que se muestran y mueven por la pantalla.

Cada sprite ocupa cuatro bytes en memoria y la OAM puede almacenar hasta un total de 40 sprites. Esos cuatro bytes corresponden a:

Byte nº	Descripción
0	Posición Y en píxeles del sprite en el área visible.
1	Posición X en píxeles del sprite en el área visible.
2	Posición del tile del sprite en la tabla de tiles.
3	Atributos: volteado horizontal, volteado vertical, paleta.

Para los sprites, el identificador de color '00' es siempre el color de transparencia.

A no ser que se especifique lo contrario, los sprites se dibujarán en pantalla sobre el mapa de fondo y la ventana.

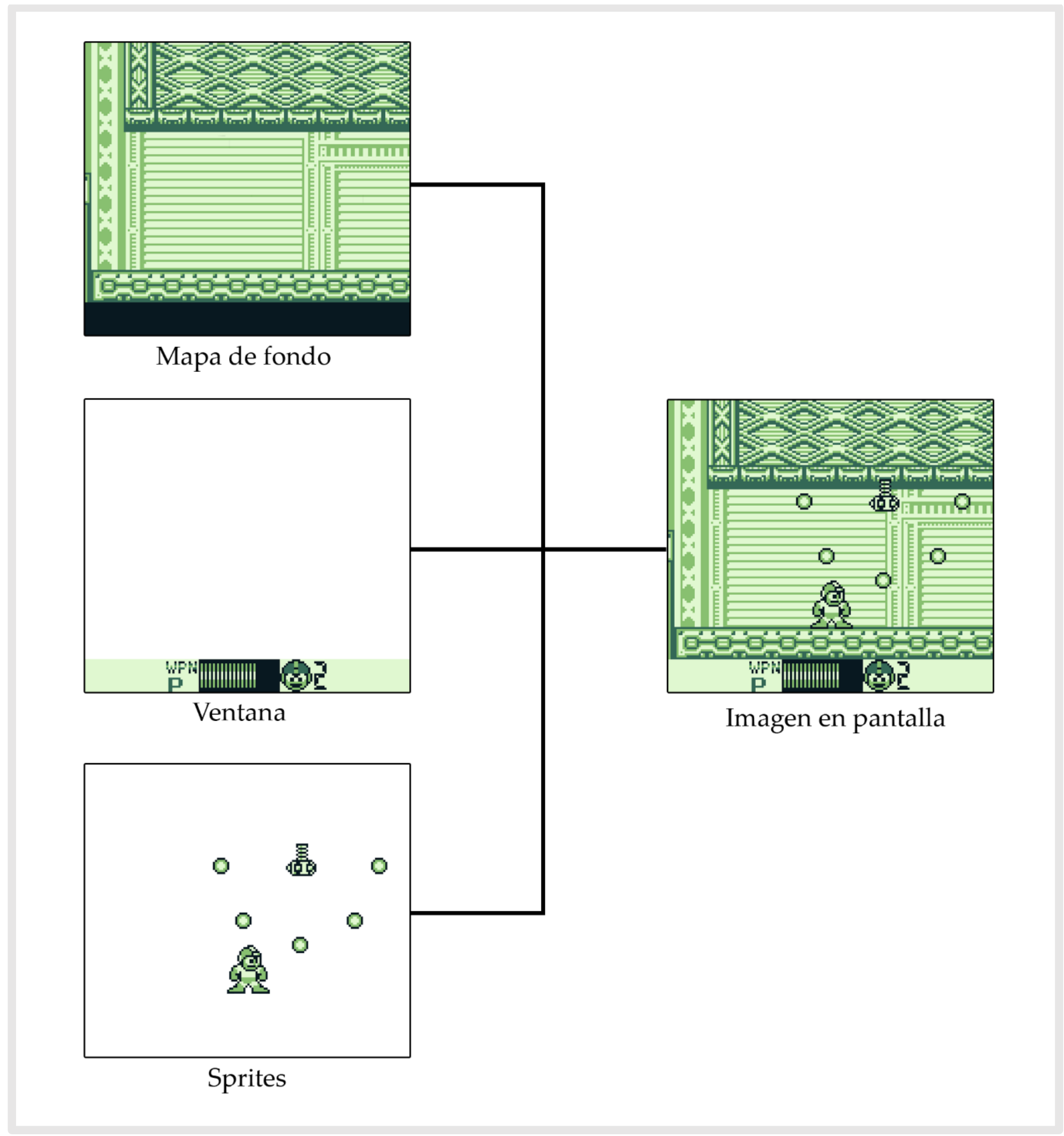


Diagrama de composición de imagen en pantalla en Megaman (Capcom 1991).

La imagen se muestra en una pantalla LCD integrada con una resolución de, como ya hemos mencionado, 160×144 píxeles y el color de cada píxel realmente varía entre 4 tonos de gris (blanco, gris claro, gris oscuro y negro). Sin embargo, la Game Boy original, al tener una pantalla LCD verde, muestra los gráficos en tonos verdosos (Copetti 2019).

9.1.3 WRAM

Se trata del espacio de memoria donde tendremos que definir todas las variables de trabajo que se requieran durante el tiempo de ejecución de nuestro programa o juego.

9.1.4 Registros de entrada/salida

$FF00	Dirección de inicio	Dirección final	Propósito
	$FF00		Datos sobre las entradas físicas de la consola (botones).
	$FF01	$FF02	Datos de entrada y salida mediante 'Serial transfer', el método de comunicación entre dos Game Boys.
	$FF04	$FF07	Registros de temporizador y divisor.
	$FF10	$FF26	Datos de sonido.
	$FF30	$FF3F	Patrón de ondas.
	$FF40	$FF4B	Control del LCD, estados de la PPU, scrolling, paletas y transferencia DMA a OAM.
	$FF4F		Selector de banco VRAM. Solo para Game Boy Color.
	$FF50		Para deshabilitar la ejecución de la ROM.
	$FF51	$FF55	Transferencia VRAM DMA. Solo para Game Boy Color.
	$FF68	$FF69	Paletas de fondo y objetos. Solo para Game Boy Color.
	$FF70		Selector de banco WRAM. Solo para Game Boy Color.

$FF70

9.1.5 HRAM

Área de la memoria útil para la transferencia DMA de datos a la OAM. La transferencia DMA se explica y aplica en el **Anexo II: transferencia DMA**.

9.2 Introducción a los registros e instrucciones de la CPU

Para la construcción de Game Boy, en lugar de colocar muchos chips estándar en la placa base, Nintendo optó por un único chip para alojar la mayoría de los componentes, incluida la CPU. Este tipo de chip se denomina "System On Chip" (SoC) y el que se encuentra en la Game Boy se conoce como Sharp LR35902 (Copetti 2019).

Este chip incluye los registros de su predecesor, el 8080 (Nintendo 1999; Pastraiser n.d).

15 … 8	7 … 0	Posición del bit
Registros Principales		
A	F	Acumulador y Flags
B	C	
D	E	
H	L	Dirección indirecta
Registros índice		
SP		Puntero a pila
Contador de programa		
PC		

Bits del registro flag (F)

7	6	5	4	3	2	1	0	Bit
Z	N	H	C	0	0	0	0	Flag (0 = sin uso)

En cuanto a las instrucciones que incluye, estas son algunas (Nintendo 1999; Pastraiser n.d):

Instrucciones lógicas: AND, OR, XOR.

Instrucciones aritméticas: ADC, ADD, CP, DEC, INC, SBC, SUB.

Instrucciones de operaciones de bit: BIT, RES, SET, SWAP.

Instrucciones de desplazamiento de bits: RL, RLA, RLC, RR, SLA, SRA, SRL.

Instrucciones de carga: LD, LDH.

Saltos y subrutinas: CALL, JR, JP, RET, RETI.

Instrucciones para las operaciones de pila: POP, PUSH.

Instrucciones varias: DI, EI, HALT, NOP, STOP.

9.3 Introducción al ensamblador RGBDS

RGBDS (Rednex Game Boy Development System) se trata de un conjunto de herramientas para la traducción del código ensamblador de un juego de Game Boy a código máquina ejecutable por la consola.

Las tres herramientas de RGBDS son:

- **rgbds** (ensamblador), encargado generar por cada fichero fuente en código ensamblador, un fichero de formato específico (fichero objeto) para que el linker pueda interpretarlo.
- **rgblink** (linker), encargado de enlazar los ficheros objetos resultado del ensamblado de la herramienta anterior en un único fichero ROM en código máquina.
- **rgbfix** (fixer), encargado de revisar y corregir la cabecera del fichero ROM resultante del linkado con rgblink.

En cuanto a la sintaxis admitida por el ensamblador, estos son los puntos introductorios clave:

La sintaxis está basada en líneas, como en cualquier otro ensamblador, lo que significa que se hace una instrucción o directiva por línea. Ejemplo:

```
[etiqueta]        [instrucción] [; comentario]
John:             LD A,87        ;Weee
```

Antes de empezar a escribir, hay que definir una sección. Esto indica al ensamblador el tipo de información que sigue y, si es código, en que área de memoria situarlo.

```
SECTION "nombre", tipo
SECTION "nombre", tipo, opciones
SECTION "nombre", tipo[dirección de memoria]
SECTION "nombre", tipo[dirección de memoria], opciones
```

Los tipos de sección posibles son:

ROM0	Sección del banco 0 de la ROM. *Dirección de memoria* entre $0000 y $3FFF.
ROMX	Sección de banco intercambiable ROM. *Dirección de memoria* entre $4000 y $7FFF.
VRAM	Sección de banco intercambiable VRAM. *Dirección de memoria* entre $8000 y $9FFF.
SRAM	Sección de RAM externa. *Dirección de memoria* entre $A000 y $BFFF.
WRAM0	Sección del banco 0 de la WRAM. *Dirección de memoria* entre $C000 y $CFFF.
WRAMX	Sección de banco intercambiable WRAM. *Dirección de memoria* entre $D000 y $DFFF.

OAM	Sección de la OAM. *Dirección de memoria* entre $FE00 y $FE9F
HRAM	Sección de la HRAM. *Dirección de memoria* entre $FF90 y $FFFE.

Las *opciones* pueden incluir:

BANK[*banco*]	Especifica en banco en el que el linker debe colocar la sección..
ALIGN[*n*, *compensación*]	Coloca la sección en una dirección cuyos *n* bits menos significativos son iguales a *compensación*.

Ejemplos de sección:

```
SECTION "Sistema de físicas",ROM0[$2000]
; Instrucciones en la ROM0
SECTION "Niveles extra",ROMX[$4567],BANK[3]
; Instrucciones en el banco 3 de la ROM intercambiable
```

Para la definición de datos en ROM:

DB define una lista de bytes que serán almacenados en la imagen final. Ideal para tablas y textos.

Se puede usar también **DW** para almacenar una lista de palabras (16 bits) o **DL** para almacenar una lista de dobles palabras (32 bits)

```
DB 1,2,3,4
DW "Hola!"
```

En cuanto a la reserva de espacio en RAM, el método preferido es utilizar **DS** para reservar un número específico de bytes vacíos.

```
DS 42 ; Reserva 42 bytes
```

El espacio vacío reservado en RAM no es inicializado por el ensamblador.

Toda la información para la configuración y ejecución de las herramientas, así como, la sintaxis completa admitida por el ensamblador se encuentra disponible en la documentación oficial de RGBDS (Bentley J et al 2021).

9.4 Creación y ensamblado de una ROM básica

Abordaremos ahora los pasos necesarios para la creación de una ROM básica de Game Boy.

9.4.1 Para empezar

Esta prueba de ROM básica se desarrollará en **Linux** y nos serviremos del recién mencionado ensamblador **RGBDS** en su versión **v0.5.2**.

9.4.2 Prerrequisitos

- Disponer del paquete **RGBDS** instalado.
- Disponer de un **emulador Game Boy/Game Boy Color** (BGB, SameBoy, No Cash, …). Nosotros optamos por el emulador BGB.
- El **editor de textos** por el que se tenga preferencia. En nuestro caso, SublimeText con el plugin de sintaxis z80 para mejorar la visualización del código.

9.4.3 Fichero "Hola Mundo"

El primer paso será crear un fichero de ensamblador (.asm) esqueleto para realizar la prueba.

Es necesario, para cualquier juego/programa en Game Boy incluir una cabecera como la siguiente:

```
SECTION "Cabecera Cartucho",ROM0[$0100]
        NOP
        JP      START   ; Salto al inicio de nuestro programa

        ;; [$0104 - $0133] LOGO DE NINTENDO
        DB $CE,$ED,$66,$66,$CC,$0D,$00,$0B,$03,$73,$00,$83,$00,$0C,$00,$0D
        DB $00,$08,$11,$1F,$88,$89,$00,$0E,$DC,$CC,$6E,$E6,$DD,$DD,$D9,$99
        DB $BB,$BB,$67,$63,$6E,$0E,$EC,$CC,$DD,$DC,$99,$9F,$BB,$B9,$33,$3E

        ;; [$0134 - $013E] + [$013F - $0142] TITULO DEL JUEGO
        DB      "HELLO WORLD", "    "
                ;0123456789A    ;0123
                ;NOMBRE         ;CODIGO DE PRODUCTO
                ;11 ASCII chrs ;4 ASCII chrs

        ;; [$0143] COMPATIBILIDAD DE COLOR
        DB $00

        ;; [$0144 - $0145] CODIGO DE CREADOR
        DB $00
        DB $00

        ;; [$0146] INDICADOR PARA SUPER GAME BOY
        DB $00

        ;; [$0147] TIPO DE CARTUCHO
        DB $00

        ;; [$0148] TAMAÑO DE ROM
        DB $00

        ;; [$0149] TAMAÑO DE RAM
        DB $00

        ;; [$014A] DESTINACION
        DB $01

        ;; [$014B] CODIGO DE LICENCIA
        DB $33

        ;; [$014C] VERSION DE MASK ROM / MANEJADA POR RGBFIX
        DB $00

        ;; [$014D] COMPROBACION COMPLEMENTARIA / MANEJADA POR RGBFIX
        DB $00

        ;; [$014E - $014F] SUMA DE CONTROL DEL CARTUCHO / MANEJADA POR RGBFIX
        DW $00
```

Esta cabecera en concreto define un cartucho únicamente compatible con Game Boy (ni Color, ni Super Game Boy) de 32KB de sólo memoria ROM. Se puede encontrar más información sobre la definición de cabeceras en el manual oficial (Nintendo 1999) o los Pan Docs (Antonio Niño Díaz et al 2021).

En ese mismo fichero que hemos creado, a parte de la cabecera, necesitaremos un pequeño programa con el que comprobar que todo está funcionando correctamente, por ejemplo:

```
SECTION "Start", ROM0[$0150]
START:
        ; Iniciamos un bucle de espera
        LD      BC, $0200

STOP_SCREEN_OUT:
        LD      DE, $0200

STOP_SCREEN_IN:
        DEC     DE
        LD      A, D
        OR      E
        JR      NZ, STOP_SCREEN_IN

        DEC     BC
        LD      A, B
        OR      C
        JR      NZ, STOP_SCREEN_OUT

        ;Apaga la pantalla después del bucle de espera
        LD      A,0             ; ┐
        LDH     [$FF40],A       ; └─> Apaga el LCD

Loop:
        HALT
        NOP
        NOP
        JR      Loop            ; Bucle infinito
```

Este pequeño programa se dedica a apagar el LCD (mostrar en blanco la pantalla de Game Boy) tras una pequeña espera durante la que veremos el logo de Nintendo dibujado.

9.4.4 Ensamblando con RGBDS

Para generar una ROM funcional con RGBDS desde un ÚNICO fichero fuente bastará con los siguientes tres comandos de consola, que se encargan del ensamblado, linkado y corrección de la ROM.

```
$ rgbasm -o salida.obj tuFichero.asm

$ rgblink -o nombreROM.gb salida.obj

$ rgbfix -v -p 0 nombreROM.gb
```

Cuando el código fuente se disponga en más de un fichero los pasos serán:

```
$ rgbasm -o sal1.obj Fichero1.asm
$ rgbasm -o sal2.obj Ficheor2.asm
$ rgbasm -o sal3.obj Fichero3.asm

$ rgblink -o nombreROM.gb -m mapa.map -n simbolo.sym *.obj

$ rgbfix -v -p 0 nombreROM.gb
```

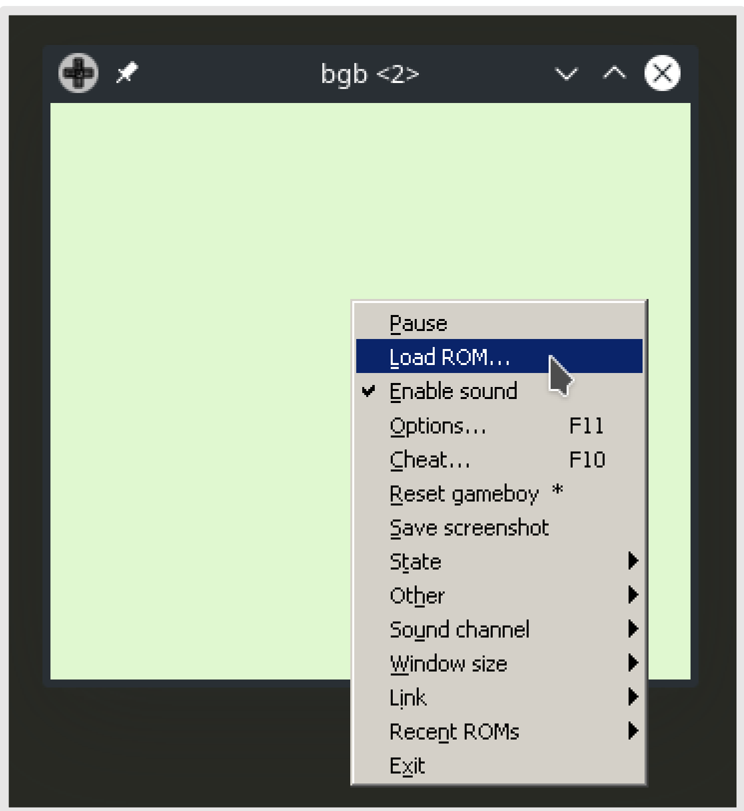
Las opciones de `-m mapa.map` y `-n simbolo.sym` nos proporcionan los ficheros *map* y *symbol* de nuestra ROM.

El fichero **map** enumera cómo se asignaron las secciones y los símbolos. También nos indica el total de memoria en ROM y RAM que ocupa nuestro código.

El fichero **symbol** enumera la dirección de todos los símbolos exportados. Varios programas externos pueden utilizar esta información, por ejemplo, para ayudar a depurar las ROMs.

9.4.5 Test del resultado con emulador BGB

Llegados a este punto dispondremos de nuestra ROM en formato .gb lista para cargar en un emulador.



Pantalla principal emulador BGB.

Una vez abierto el emulador, seleccionamos la opción de cargar ROM y buscamos nuestro fichero .gb generado.

Si hemos utilizado el código del programa proporcionado en los pasos anteriores, veremos, al cargar la ROM en el emulador, el logo de Nintendo el cual desaparecerá tras unos segundos, cuando se alcance la operación que apaga el LCD de la consola.

10. Whyrm. Desarrollo

Con el diseño listo, comenzamos la fase de desarrollo para Whyrm, nuestro juego. En este punto recorreremos, de manera detallada y en orden, toda la implementación que se lleve a cabo, así como, los problemas (y su resolución) que surjan durante el desarrollo. El desarrollo parte de cero: con un fichero en blanco y sin conocimientos previos sobre el desarrollo en Game Boy.

Para la implementación del proyecto se ha seguido los planteamientos de la **metodología ágil** Scrum, pensada para ofrecer resultados en períodos muy limitados de tiempo e iterar sobre los resultados obtenidos. De esta forma, el tiempo de desarrollo se divide en sprints: iteraciones breves de, generalmente, una semana de duración (Trigás 2012).

Además, hemos definido cuatro puntos claves del estado del proyecto, a los que denominamos **hitos**, con el objetivo de alcanzar cada uno de ellos en tres sprints. Como veremos a continuación, estos cuatro hitos definen la estructura de este apartado.

El desarrollo se ha hecho, y como ya mencionamos en la introducción, bajo la arquitectura ECS (Entity-Component-System).

Para nuestra implementación, sin embargo, son necesarias ligeras modificaciones de la arquitectura adaptadas a la tecnología y el tiempo del que disponemos para el desarrollo.

- En nuestro ECS, los **componentes** serán los datos, cada uno de los bytes (posición, velocidad, puntero a sprite, estado de animación...).
- **Entidades** son cada uno objetos del juego con apariencia y comportamiento. Son simplemente grupos de datos (grupos de componentes).
- Cada uno de los **sistemas** corresponderá a código que transforma los datos de una manera concreta. Un sistema se diseñará para funcionar solo con determinados componentes. Por ejemplo, el sistema de físicas transformará, para todas las entidades, los componentes físicos: velocidad, posición, …
- Además, como añadido, dispondremos de **mánagers**: código con la responsabilidad de poseer datos (tendrá la definición de esos datos) y gestionarlos. Por ejemplo, el

mánager de entidades poseerá todos los datos de las entidades y las estructuras de esos datos, y gestionará la creación de nuevas entidades y borrado de entidades ya existentes.

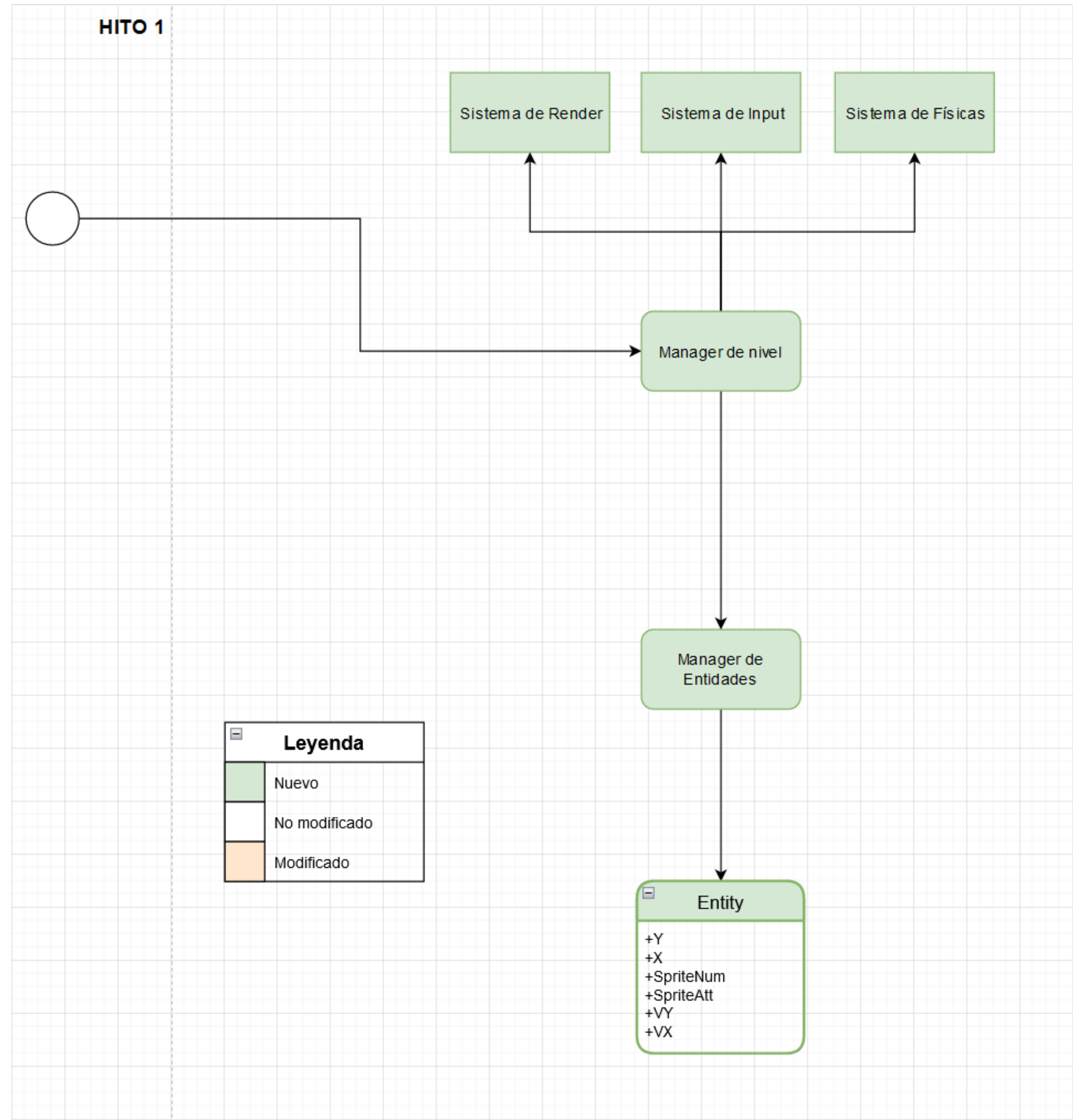
10.1 Hito 0. Introducción a la programación en Game Boy

El **hito 0** consiste en adquirir los conocimientos básicos necesarios para arrancar el desarrollo de un juego en Game Boy. Para esto hemos extraído de los manuales oficiales de Game Boy, toda la información conveniente para familiarizarnos y conocer de manera general la consola y, además, ser capaces de desarrollar una ROM básica y funcional. Para esta última parte ha sido necesario también aproximarnos de manera introductoria a las instrucciones de la CPU de Game Boy y a un ensamblador (en nuestro caso, el ensamblador RGBDS).

La información y conocimientos adquiridos en este hito, así como el desarrollo de una ROM básica, son los expuesto a lo largo del punto 9.

10.2 Hito 1. Creación y manejo de entidades con motor básico

Como **hito 1** hemos planteado desarrollar un motor ECS básico que nos permita generar entidades visibles en pantalla. Además, el movimiento de una de las entidades, la del caballero, podrá ser controlada con los botones físicos de la consola.



Estructura del motor ECS al término del hito 1.

10.2.1 Mánager de entidades

Es el encargado de la gestión de todos los datos relacionados directamente con las entidades y sus componentes.

A través de este mánager:

- Reservaremos espacio en memoria y almacenaremos los datos de los componentes de una entidad cuando esta se cree.
- Borraremos los datos y liberaremos el espacio en memoria de una entidad cuando esta sea borrada.

Todas las entidades ocupan el mismo espacio, es decir, todas tienen los mismos componentes (siendo cada componente un byte en memoria). Además, la creación de entidades queda limitada a 10 como el máximo número de entidades coexistentes.

Las entidades se almacenan en una estructura de datos similar a un *array* de un lenguaje de programación moderno. En adelante, nos referiremos a esta estructura de datos como **array de entidades**.

Al iniciar nuestro juego, el mánager de entidades se encarga de reservar un espacio vacío en memoria para el array de entidades. El tamaño de ese espacio viene especificado por la siguiente fórmula:

*EntityArray.Size = Entity.Size * MaxEntities*

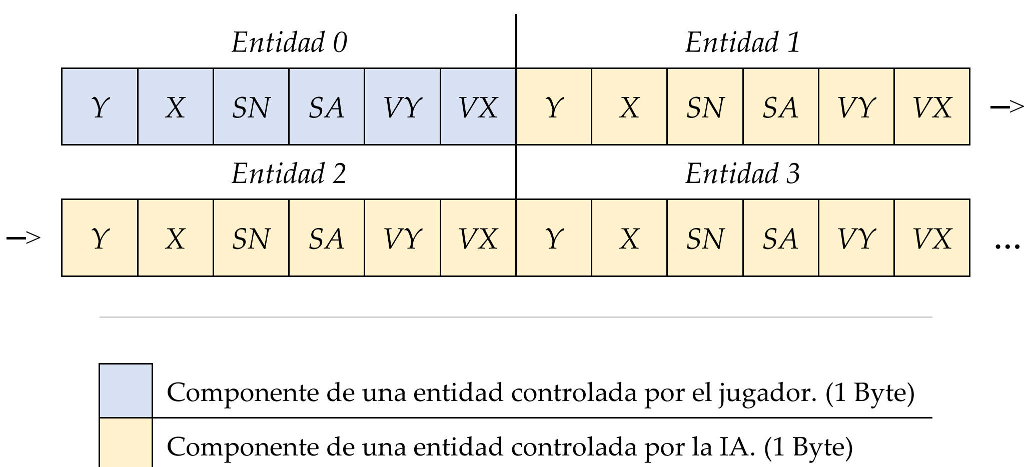
Siendo:

- *EntityArray.Size*, el tamaño en bytes del array de entidades.
- *Entity.Size*, el tamaño en bytes de una entidad. Equivale al número de componentes de una entidad.
- *MaxEntities*, el máximo número de entidades coexistentes.

Definimos, a continuación, los componentes de una entidad para el hito actual:

Componente	Significado
Y	Posición en pantalla (píxel) de la entidad en el eje Y.
X	Posición en pantalla (píxel) de la entidad en el eje X.
SpriteNum	Posición del tile de la entidad en la tabla de tiles.
SpriteAtt	Atributos del tile (orientación, paleta, …) de esa entidad.
VY	Velocidad en el eje Y de la entidad.
VX	Velocidad en el eje X de la entidad.

Sabiendo entonces que podemos tener hasta un máximo de 10 entidades y que cada entidad posee seis componentes diferentes, el array de entidades ocupará un espacio de 60 bytes y se ve en memoria de la siguiente manera:



La primera entidad del array (entidad 0) será siempre la entidad controlable por el jugador, el caballero.

10.2.2 Sistema de render

Es el sistema encargado de transformar los componentes necesarios para que las entidades se muestren en pantalla. Estos componentes son: *X*, *Y*, *SpriteNum* y *SpriteAtt*.

La transformación, en nuestro caso, consiste simplemente en copiar, para cada entidad, esos cuatro componentes a la memoria de objetos de la Game Boy (**OAM**).

Recordemos que la OAM es una sección de la memoria de vídeo donde se almacena la información necesaria de los objetos o entidades para que estos puedan ser dibujados en

pantalla. Esa información corresponde exactamente, y con esa intención se ha diseñado, con los cuatro primeros componentes de nuestras entidades.

Sería lógico, sabiendo esto, codificar el sistema de render para que, comenzando por la primera entidad de nuestro array, copiara uno a uno los bytes correspondientes a esos cuatro primeros componentes y repetir esta operación para el resto de las entidades.

Esta solución es correcta y totalmente viable para otros sistemas o consolas, pero, en Game Boy, existe una manera mucho más eficiente de copiar esa información: la **transferencia DMA**.

La transferencia DMA (Direct Memory Access), es capaz de copiar bloques de memoria de la ROM o la RAM a la OAM en los momentos oportunos y de manera muy eficiente (aproximadamente 160 microsegundos). En el **Anexo II: transferencia DMA** podemos ver en detalle la implementación de la transferencia DMA a OAM.

La implementación de la transferencia DMA trae consigo, por desgracia, un pequeño inconveniente respecto al planteamiento inicial que habíamos hecho para los componentes y entidades: al copiar bloques de memoria en conjunto, no tenemos manera de especificar cuáles son los componentes que queremos que se copien y cuáles no. Por ello, tendremos que separar nuestro array de entidades en dos.

Por un lado, tendremos los componentes que se corresponden directamente con la OAM (para transferencia DMA), y por otro, todo el resto de los componentes. En adelante, referenciaremos a estos dos arrays como **sOAM** (Sombra OAM) y array **RC** (resto de componentes).

<u>sOAM</u>

Entidad 0				*Entidad 1*				
Y	*X*	*SN*	*SA*	*Y*	*X*	*SN*	*SA*	…

<u>RC</u>

Entidad 0		*Entidad 1*		
VY	*VX*	*VY*	*VX*	…

El sistema de render se encarga también de las tareas de transferencia de datos de ROM a VRAM.

Tarea	Dirección destino
Cargar los tiles del juego.	$8000 (tabla de tiles)
Cargar el mapa de tiles correspondiente al nivel actual	$9800 o $9C00, dependiendo del área de memoria que se esté utilizando para dibujar el mapa de fondo.

El sistema de render, como sistema que es, no almacena ni posee ningún tipo de dato. Para poder llevar a cabo las tareas vistas previamente, el mánager de nivel (expuesto más adelante) ofrece al sistema de render cuatro parámetros. Dos de ellos se pasan una vez al inicio del juego:

- **Puntero** a la dirección de memoria ROM donde se almacenan los **tiles del juego**.
- **Puntero** a la dirección de memoria ROM donde se almacena el **mapa de tiles** correspondiente al nivel actual.

Los otros dos se pasa una vez cada frame:

- **Puntero** a la dirección de memoria RAM correspondiente al primer componente del **array sOAM**.
- **Número de entidades** que existen en ese momento.

10.2.3 Sistema de input

El sistema de input se encarga de consultar el registro de entrada/salida **$FF00** que contiene la información de qué botones del *joypad* están siendo pulsados.

Joypad es comúnmente el término que se utiliza para referirse a los botones físicos de la Game Boy.

Game Boy y joypad. Imagen por Evan Amos.

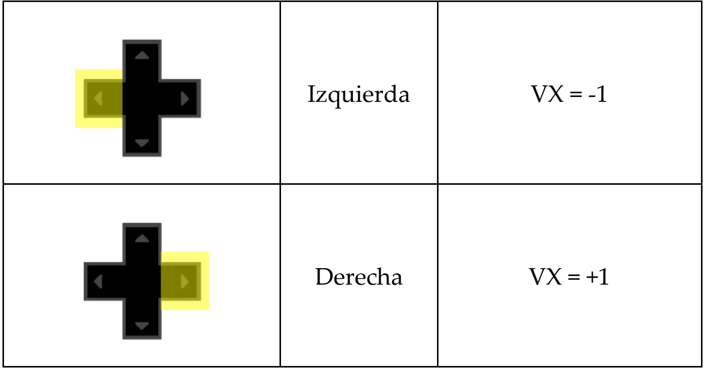
Es más que recomendado leer de la dirección $FF00 varias veces seguidas, para que las entradas se estabilicen, y utilizar realmente el valor de la última lectura.

Adicionalmente, para este hito, el sistema de input modificará los componentes *VY* y *VX* de la primera entidad (caballero) del array RC cuando se detecten pulsaciones en los botones de dirección.

Para poder acceder a los componentes de velocidad del caballero, el mánager de nivel ofrece al sistema de input, cada frame, el **puntero** a la dirección de memoria RAM correspondiente al primer componente del **array RC**.

La velocidad del caballero se actualiza de acuerdo a la siguiente tabla:

Botón	Dirección	Valor de la velocidad
	Arriba	VY = -1
	Abajo	VY = +1



	Izquierda	VX = -1
	Derecha	VX = +1

10.2.4 Sistema de físicas

Para este hito, el sistema de físicas se encarga en cada frame de actualizar la posición (X, Y) de cada una de las entidades sumando a esa posición la velocidad de la entidad en ambos ejes.

$$Entity.Y(n) = Entity.Y(n-1) + Entity.VY(n-1)$$

$$Entity.X(n) = Entity.X(n-1) + Entity.VX(n-1)$$

Siendo:

- *n*, el frame actual, el que corresponde a la próxima imagen en ser dibujada en pantalla.
- *n-1*, el frame inmediatamente anterior al actual.

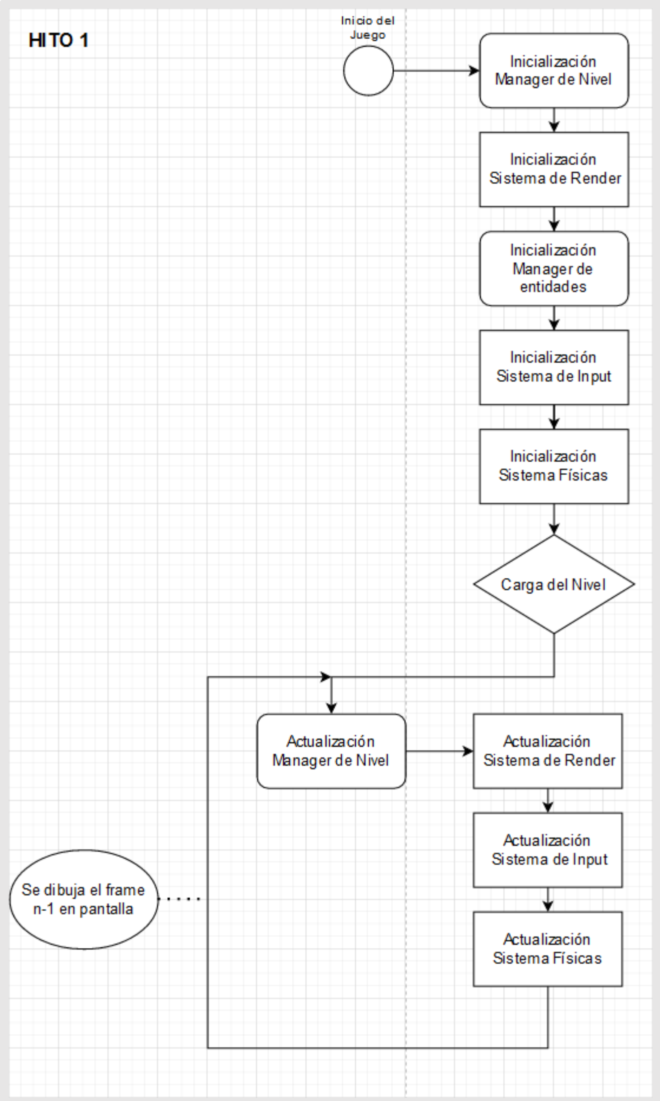
Para cumplir con esta tarea, el mánager de nivel ofrece en cada frame tres parámetros al sistema de físicas:

- Para acceder a la posición de las entidades, **puntero** a la dirección de memoria RAM correspondiente al primer componente del **array sOAM**.
- Para acceder a la velocidad de las entidades, **puntero** a la dirección de memoria RAM correspondiente al primer componente del **array RC**.
- **Número de entidades** que existen en ese momento.

10.2.5 Mánager de nivel

Es el encargado de realizar la llamada de inicialización y actualización para los mánagers y sistemas a los que posee. En este punto del desarrollo el mánager de nivel posee a todos los

sistemas y mánagers creados. Es, por tanto, el encargado de iniciar y mantener el flujo del juego.



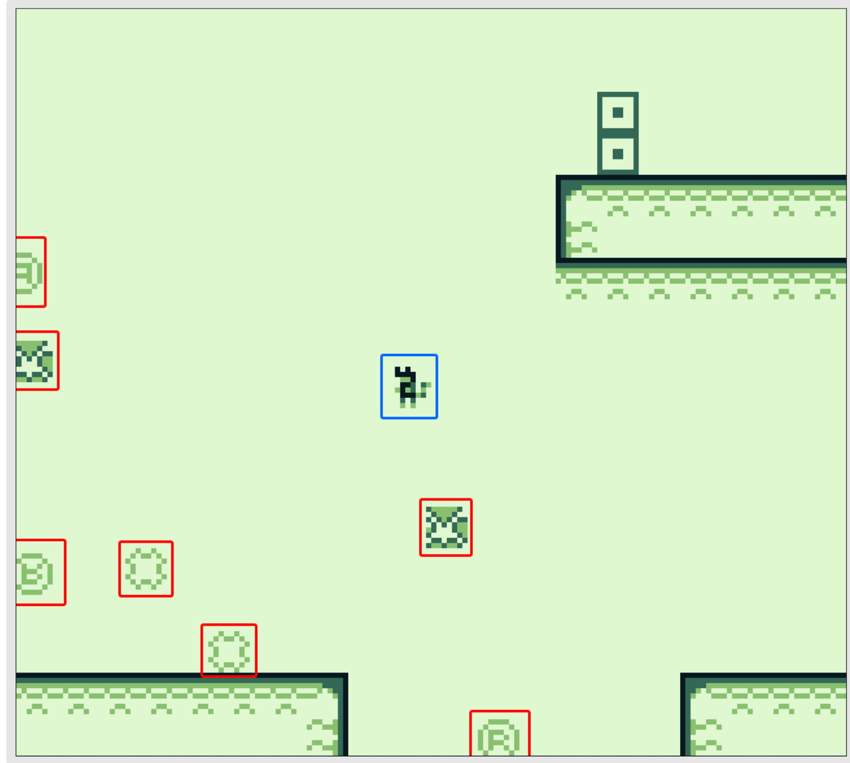
Flujo del juego al término del hito 1.

El mánager de nivel también posee la definición del espacio de memoria ROM donde se encuentran los datos del mapa de tiles a cargar como mapa de fondo y el espacio de memoria ROM donde se encuentran los datos de las entidades a crear.

10.2.6 Otros desarrollos

Adicionalmente, en este hito se ha implementado una optimización sobre el período VBlank. El desarrollo de la optimización se explica al detalle en el **Anexo III: Optimización VBLANK**.

10.2.7 Estado final



Captura del estado del juego al término del hito 1.

Los sprites marcados en rojo en la imagen corresponden a las **entidades no controlables** que se mueven por el mapa con la velocidad con la que fueron creadas.

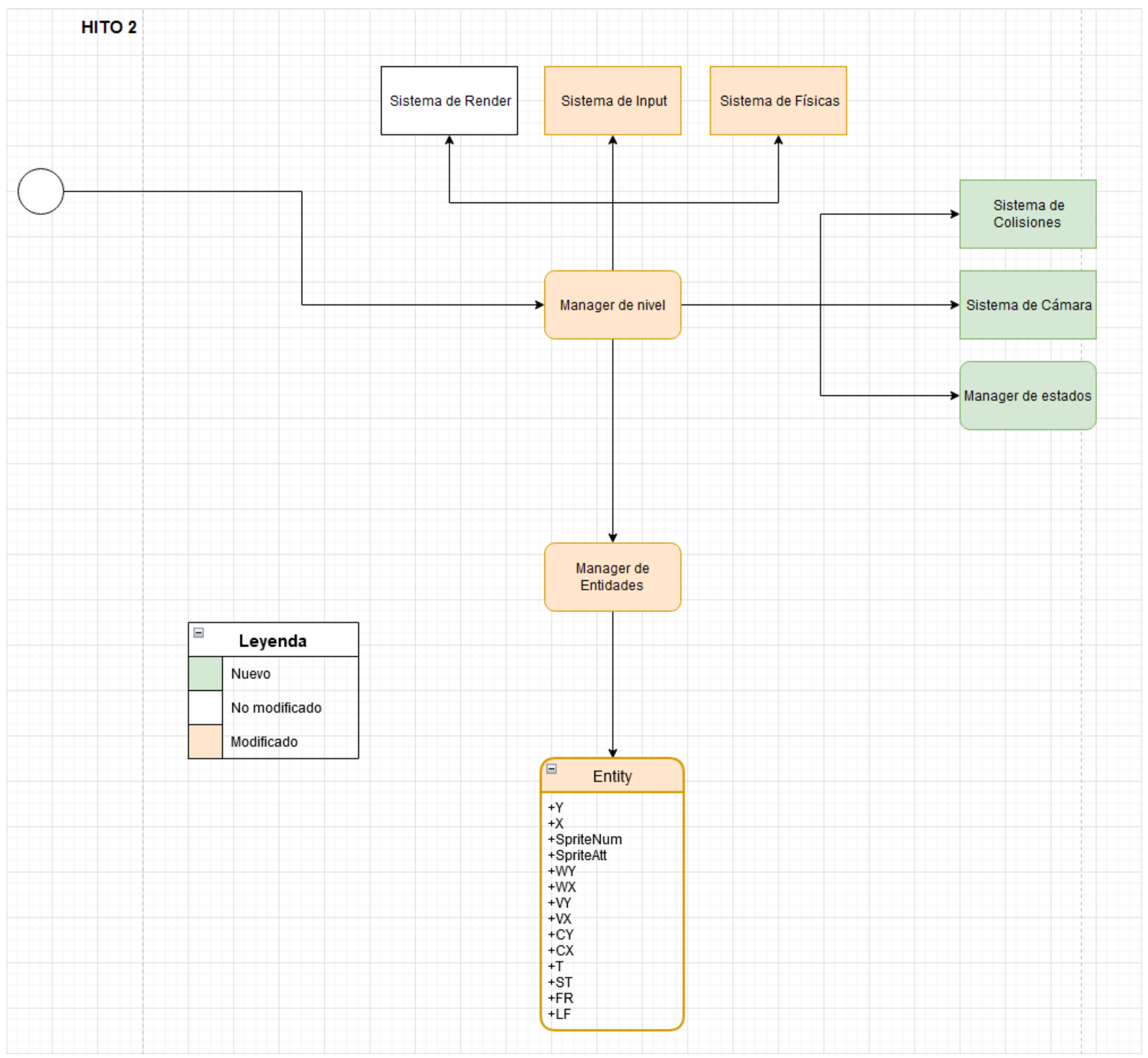
El **caballero**, sprite marcado en azul, sin embargo, se mueve por el mapa a izquierda o derecha y arriba o abajo dependiendo de los botones pulsados del joypad.

Hemos cargado para la prueba también un mapa de fondo con algunos tiles que ahora mismo no obstaculiza de ninguna manera a las entidades.

El área visible en pantalla, en este punto del desarrollo, todavía no se mueve y es siempre la misma.

10.3 Hito 2. Prototipo de nivel

El **hito 2** es una primera versión de un nivel del juego. En este nivel aparecen todos los elementos posibles de un nivel del juego final, al menos, en la medida mínima posible.



Estructura del motor ECS al término del hito 2.

10.3.1 Mánager de entidades

Para cumplir con el hito, serán necesarias una serie de modificaciones en el mánager de entidades.

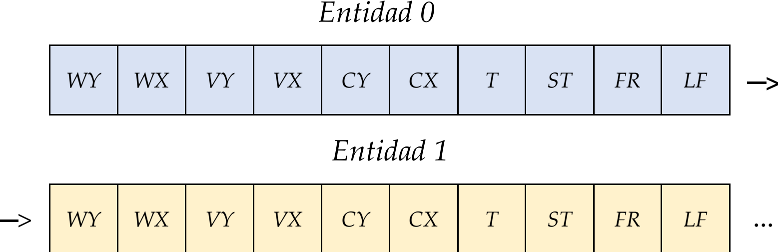
Primeramente, necesitamos nuevos componentes para el array RC:

Componente	Significado
WY	Posición en el nivel (píxeles) de la entidad en el eje Y.

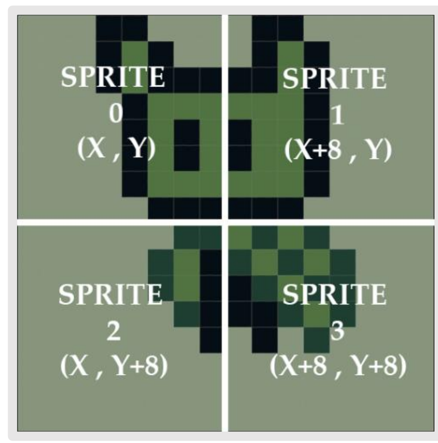
WX	Posición en el nivel (píxeles) de la entidad en el eje X.
CY	Tipo de colisión detectada en el eje X para la entidad.
CX	Tipo de colisión detectada en el eje Y para la entidad.
T	Tipo de entidad. Caballero o enemigo.
ST	Estado de la entidad.
FR	Frame del estado de la entidad.
LF	Dirección hacia donde mira la entidad.

El array RC será ahora:

RC

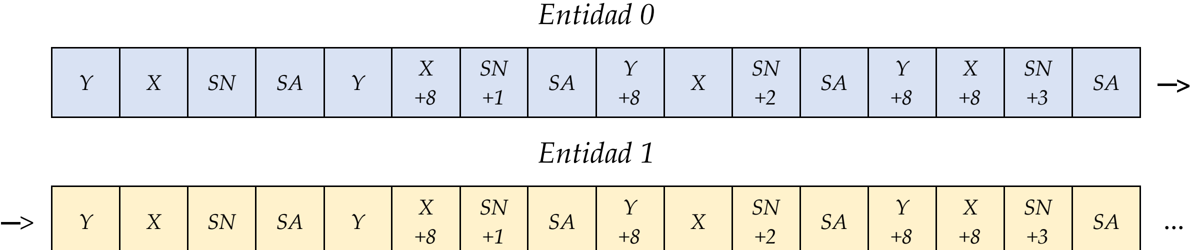


Al final del hito anterior, veíamos en la captura de nuestro juego entidades con sprites de 8x8 píxeles. Para seguir con mayor facilidad el movimiento de las entidades y ganar detalle en el diseño de sprites hemos llevado a cabo una modificación para que las entidades puedan alcanzar tamaños de hasta 16x16 píxeles. Ahora, cada entidad es una composición de cuatro sprites de 8x8 píxeles. Esto significa que en el array de sOAM cada entidad pasa de ocupar cuatro bytes a ocupar 16.



Entidad compuesta por cuatro sprites.

sOAM



A 16 bytes cada entidad, 10 entidades suman un total de 160 bytes o, lo que es lo mismo, el total del espacio de memoria reservado para la OAM. Esta es la razón por la que el número máximo de entidades coexistentes permanecerá fijo en 10.

10.3.2 Sistema de colisiones

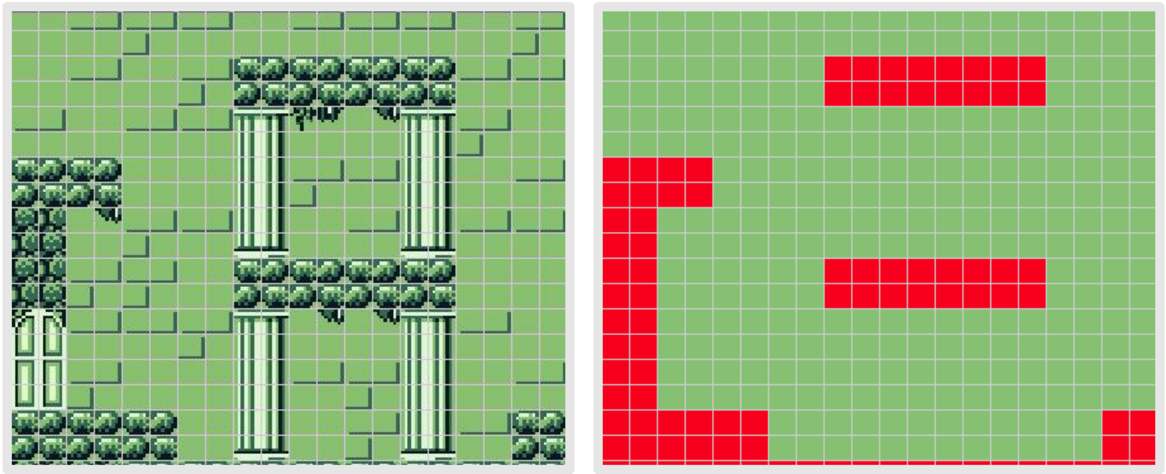
Por un lado, comprobará, para cada entidad, con que tiles del mapa de fondo va a colisionar y que tipo de tile son. Por otro, comprobará si el caballero está colisionando con alguna de las entidades controladas por la IA.

Comenzaremos listando los distintos tipos de tile:

Tipo de Tile	Tipo de colisión	Máscara
No colisionable	No produce ningún tipo de colisión.	%00000000 = $00
Delimitador	Marca un cambio de sentido para las entidades con desplazamiento controladas por la IA.	%01110000 = $70
Colisionable normal	Detiene el movimiento del caballero.	%10000000 = $80
Mortal	Mata al caballero.	%11100000 = $90
Victoria	Marca el punto final de un nivel. Cuando el caballero colisiona con un tile de este tipo el nivel se da por finalizado.	%10100000 = $A0

Para la detección de colisiones de entidades con el mapa existe la opción de trabajar con dos submapas paralelos, el de tiles y el de colisiones. El primero contendría la información necesaria para que la consola pueda determinar que tile tiene que dibujar en cada posición del mapa, y el segundo contendría, para cada posición del mapa, la información sobre el tipo de correspondiente al tile situado en esa misma posición.

Área visible en pantalla para el primer nivel del juego Kid Icarus (Nintendo R&D1 1991).



A la izquierda, mapa de tiles. A la derecha, mapa de colisiones.

Esta solución, sin embargo, requiere, por cada mapa de 32x32 tiles (256x256 píxeles), dos submapas de 32x32 bytes, es decir, un total de 2.048 bytes. Un tamaño bastante alto teniendo en cuenta los limitados espacios de memoria ROM de los cartuchos de Game Boy.

Nosotros conseguimos agrupar la información de tiles y colisiones en un único mapa de 1.024 bytes de la siguiente manera:

Los tiles se cargarán en la tabla de tiles a partir de la posición concretas que marca la máscara del tipo de tile al que pertenecen.

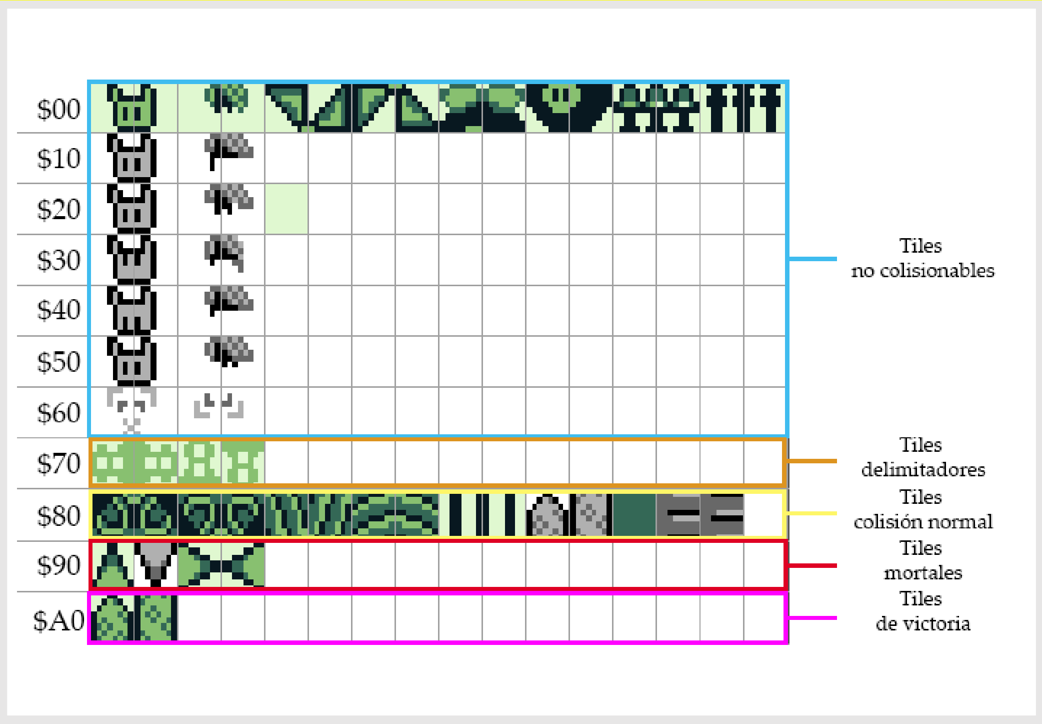
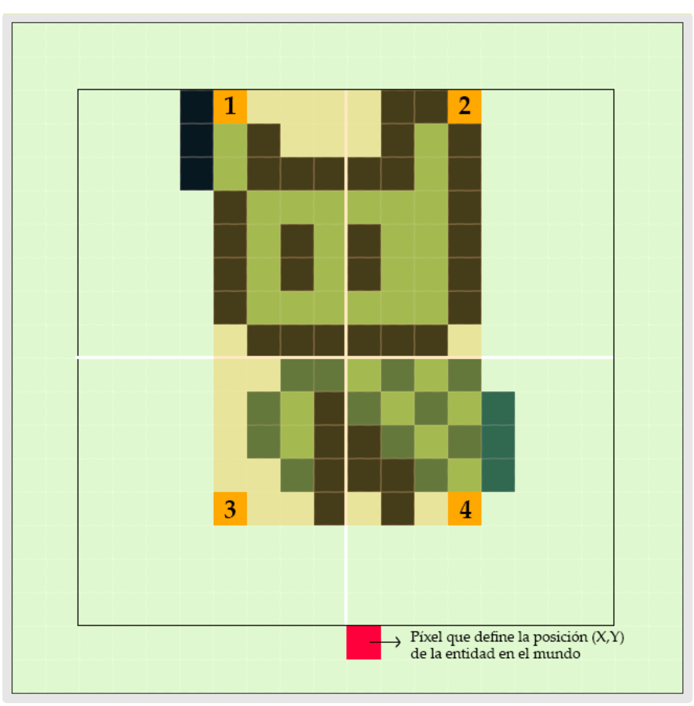


Tabla de tiles cargada según tipo de tile.

De esta forma, la parte alta del byte identificador de un tile en el mapa de tiles nos indicará qué tipo de tile es, pues corresponde a cada una de las máscaras.

A la hora de resolver, en nuestro código, el tipo de colisión que se ha producido, basta con realizar una operación **AND** entre el valor del tile al que la entidad quiere avanzar y el valor binario **%11110000**. Tras la operación nos quedará solo la parte alta del byte del tile, su tipo.

Las entidades se mueven por el mapa a **nivel de píxel** y no de tile. Para determinar con que tiles está colisionando una entidad tenemos que definir cuatro píxeles referencia que conforman su área de colisión.



Área de colisión para la entidad del caballero.

La posición del píxel 1 se calcula cómo:

$$Píxel(1).X = Entiy(n).X - (Entity(n).W/2)$$

$$Píxel(1).Y = Entiy(n).Y – Entity.MaxH$$

La posición del píxel 2 se calcula cómo:

$$Píxel(2).X = Entiy(n).X + (Entity(n).W/2) - 1$$

$$Píxel(2).Y = Entiy(n).Y – Entity.MaxH$$

La posición del píxel 3 se calcula cómo:

$$Píxel(3).X = Entiy(n).X - (Entity(n).W/2)$$

$$Píxel(3).Y = Entiy(n).Y – (Entity.MaxH – Entity(n).H) - 1$$

La posición del píxel 4 se calcula cómo:

$$Píxel(4).X = Entiy(n).X + (Entity(n).W/2) - 1$$

$$Píxel(4).Y = Entiy(n).Y – (Entity.MaxH – Entity(n).H) - 1$$

Siendo, para todas las fórmulas:

- *Entity(n).W*, el ancho dado para la entidad.
- *Entity(n).H*, el alto dado para la entidad.
- *Entity.MaxH*, el alto máximo que puede tomar una entidad. Fijado en 16 píxeles.

Los anchos y altos de las entidades son constantes. El caballero tiene un ancho y alto propios. Todas las entidades controladas por la IA comparten un mismo ancho y alto.

Dependiendo de la dirección en la que se esté moviendo la entidad tendremos que comprobar en qué tile del mapa de tiles se encuentran tres de los cuatro píxeles que conforman el área de colisión de una entidad.

Condición	**Puntos a comprobar**
$VY > 0$ y $VX > 0$	2, 3 y 4
$VY > 0$ y $VX < 0$	1, 3 y 4
$VY < 0$ y $VX > 0$	1, 2 y 4
$VY < 0$ y $VX < 0$	1, 2 y 3

Para saber a qué tiles del mapa corresponden las posiciones de los tres píxeles a comprobar es necesario convertir esa posición en píxeles a un formato de dirección de memoria del mapa de tiles. La conversión se realiza de la siguiente manera:

*TileMemDir = TileMap.InitDir + (Píxel(n).X/8) + (Píxel(n).Y/8*32)*

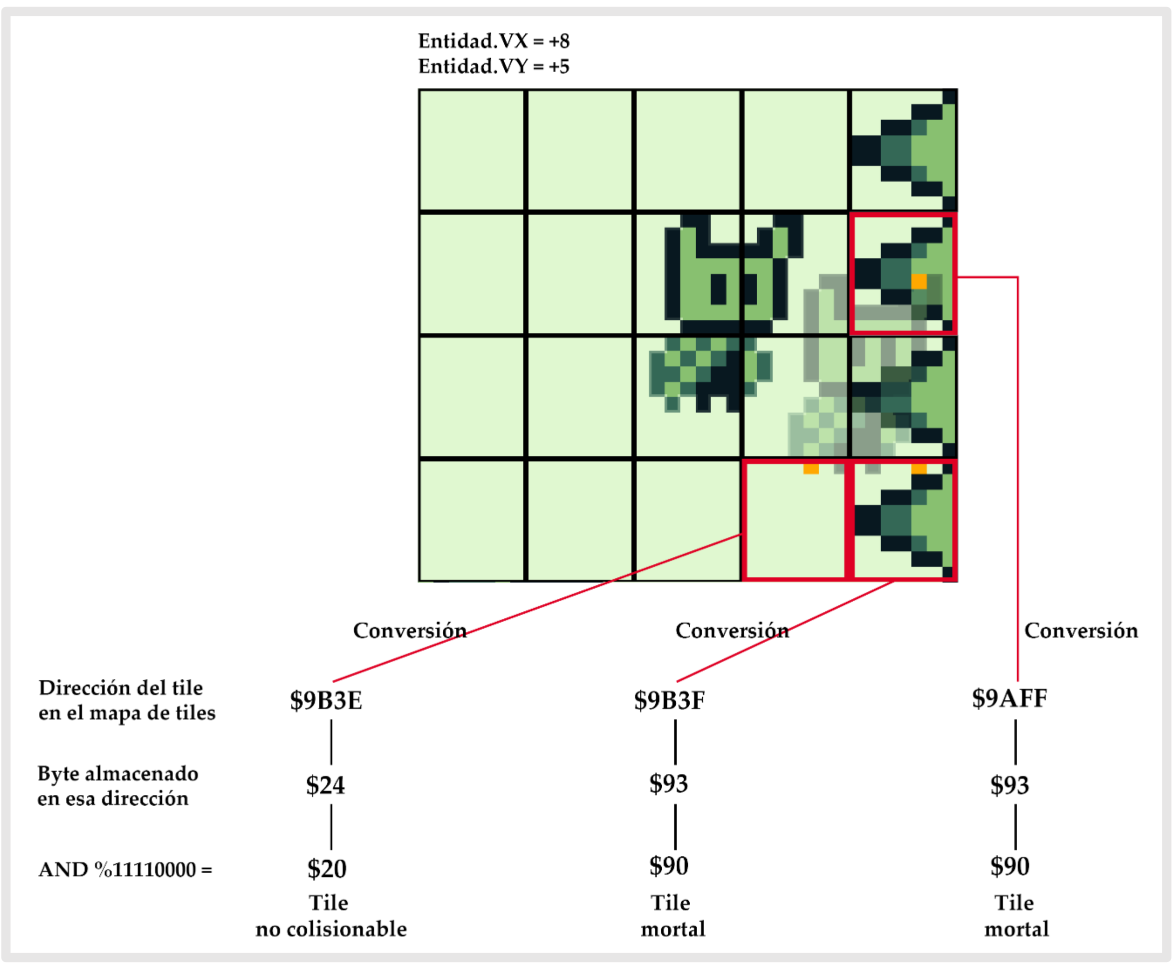
Siendo:

- *TileMemDir*, la dirección de memoria en el mapa de tiles correspondiente al tile en el que se encuentra el píxel a convertir.
- *TileMap.InitDir*, la dirección de memoria ROM donde se encuentra almacenado el primer tile del mapa de fondo actual.
- *Píxel(X),* valor de la posición en el eje X del píxel que se quiere convertir. Se divide entre ocho porque cada tile tiene un ancho de ocho píxeles.

- *Píxel(Y)*, valor del eje Y en píxeles de la posición que se quiere convertir. Se divide entre ocho porque los tiles tienen un alto de ocho píxeles. Se multiplica por 32 porque, en la memoria de vídeo, entre un tile y el tile situado bajo este hay una diferencia de 32 bytes, el equivalente al ancho en tiles de la pantalla de Game Boy.

Cabe recordar que, operando con números enteros, *Y*4* no tiene por qué generar el mismo resultado que *Y/8*32*.

Ejemplifiquemos este cálculo con una situación real en nuestro juego:



Representación del cálculo de colisiones de las entidades con el mapa de fondo.

Los tipos de colisiones detectados se almacenarán para cada entidad en los nuevos componentes ***CX*** y ***CY***.

Adicionalmente, algunos tipos de colisiones lanzan una petición de cambio de estado para la entidad.

Tipo colisión	Tipo entidad	Tipo petición
Delimitador	Enemigo	Cambio de sentido
Mortal	Caballero	Cambio a muerto

Para detectar las colisiones entre el caballero y el resto de las entidades hemos seguido una codificación similar a la de la detección de colisiones con el mapa de fondo, pero ahorrándonos las conversiones de posiciones y operaciones para averiguar el tipo de tile, pues todas las entidades operan con su posición en píxeles.

Este tipo de colisiones no se almacenan. Cuando se detectan lanzan una petición de cambio de estado al mánager de estados para actualizar el estado del caballero a "rebote aéreo fase 2" o "muerto" dependiendo de si en ese momento el caballero se encuentra o no en estado "rebote aéreo fase 1".

Para cumplir con sus tareas, el mánager de nivel ofrece en cada frame dos parámetros al sistema de colisiones:

- **Puntero** a la dirección de memoria RAM correspondiente al primer componente del **array RC**.
- **Número de entidades** que existen en ese momento.

Al inicio del nivel le ofrece también:

- **Puntero** a la dirección de memoria RAM donde se encuentran almacenados los datos del mapa de fondo.

10.3.3 **Sistema de cámara**

En el hito anterior trabajábamos únicamente con la posición de la entidad en pantalla, pero ahora vamos a habilitar el desplazamiento del área visible en pantalla y esto nos obliga a manejar también las entidades con una posición sobre el nivel/mundo si no queremos que el movimiento de la cámara afecte también al movimiento de las entidades.

Este sistema se encarga fundamentalmente de dos tareas:

1. Para cada entidad, convertir su posición mundo en posición pantalla.

2. Actualizar la posición del área visible en pantalla atendiendo al movimiento de la entidad controlable por el jugador, el caballero.

Como veíamos en el apartado de introducción a la programación en Game Boy, la posición del área visible en pantalla dentro del mapa de tiles viene dada por el registro de entrada/salida **$FF42** (posición en Y) y **$FF43** (posición en Y). Modificar estos registros modifica la posición del área visible o, como nos referiremos a partir de ahora, cámara.

La conversión de posición mundo a posición pantalla de una entidad se realiza con la siguiente fórmula:

Sprite 0

$$Entity(n).X(0) = Entity(n).WX - Cam.X$$

$$Entity(n).Y(0) = Entity(n).WY - Cam.Y$$

Siendo:

- *Entity(n).X(0)*, el valor actualizado de la posición X en pantalla del sprite 0 de la entidad.
- *Entity(n).X(0)*, el valor actualizado de la posición Y en pantalla del sprite 0 de la entidad.
- *Entity(n).WX*, el valor de la posición X en mundo de la entidad.
- *Entity(n).WY*, el valor de la posición Y en mundo de la entidad.
- *Cam.X*, el valor de la posición X en mundo de la cámara. Valor del registro $FF34.
- *Cam.Y*, el valor de la posición X en mundo de la cámara. Valor del registro $FF32.

Tendremos que hacer esta operación para los tres sprites restantes de la composición de sprites que forma una entidad.

Sprite 1

$$Entity(n).X(1) = Entity(n).WX + 8 - Cam.X$$

$$Entity(n).Y(1) = Entity(n).WY - Cam.Y$$

Sprite 2

$$Entity(n).X(2) = Entity(n).WX - Cam.X$$

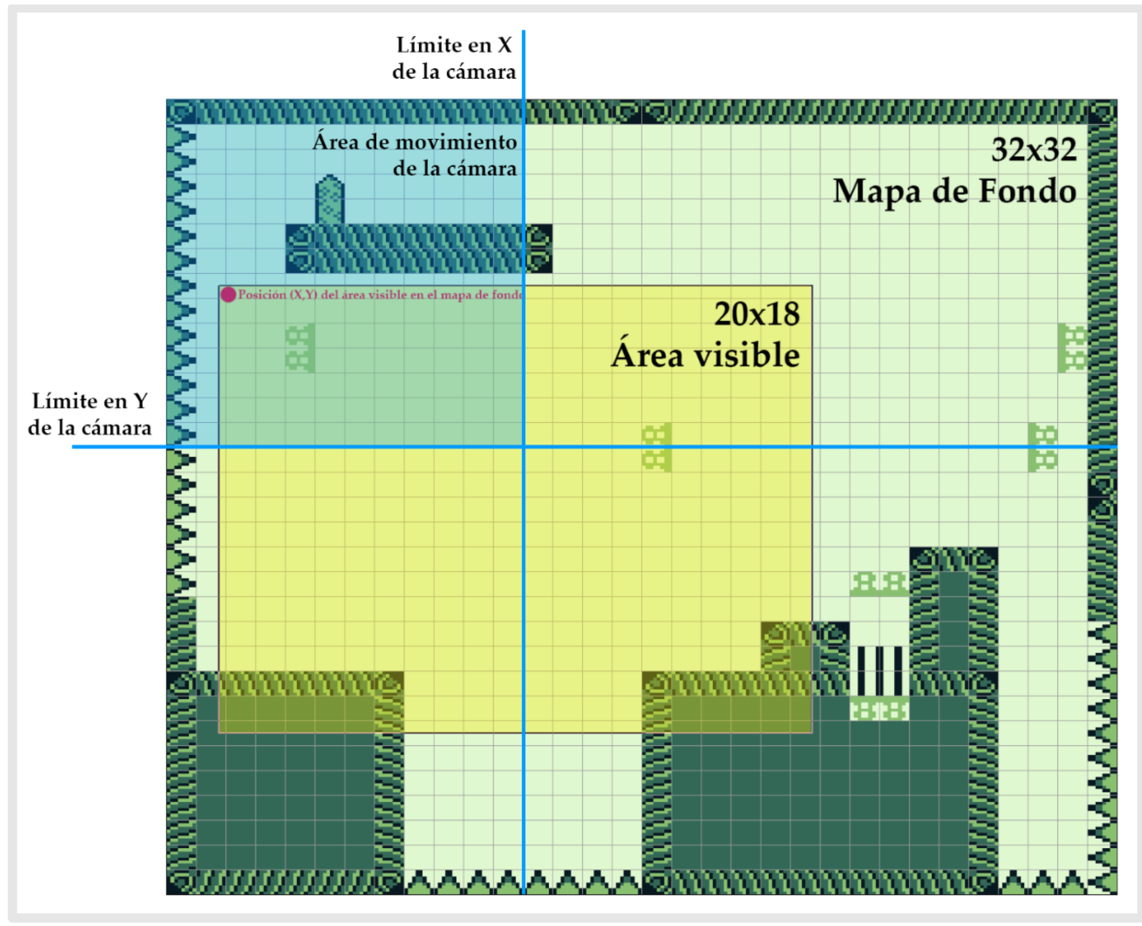
$$Entity(n).Y(2) = Entity(n).WY + 8 - Cam.Y$$

Sprite 3

$$Entity(n).X(3) = Entity(n).WX + 8 - Cam.X$$

$$Entity(n).Y(3) = Entity(n).WY + 8 - Cam.Y$$

El movimiento de la cámara tiene como objetivo seguir al caballero a lo largo del nivel, manteniendo a este en el centro de la pantalla siempre que sea posible.



Representación de los límites de movimiento de la cámara.

Los límites se calculan de la siguiente forma:

$$Cam.MaxX = Map.W - Cam.X - 1$$

$$Cam.MaxY = Map.H - Cam.Y - 1$$

Siendo:

- *Cam.MaxX*, la posición en píxeles más alta en X que la cámara tomará.
- *Cam.MaxY*, la posición en píxeles más alta en Y que la cámara tomará.
- *Map.W*, el ancho en píxeles del mapa (256).
- *Map.H*, el alto en píxeles del mapa (256).

En estos cálculos restamos 1 porque la posición (X, Y) inicial de la cámara es (0, 0).

Para cumplir con sus tareas, el mánager de nivel ofrece en cada frame dos parámetros al sistema de cámara:

- **Puntero** a la dirección de memoria RAM correspondiente al primer componente del **array RC**.
- **Número de entidades** que existen en ese momento.

10.3.4 Sistema de físicas

A las tareas que desempeñaba en el hito anterior se le suma actualizar el componente ***LF*** de todas las entidades.

Se actualizará el componente *LF* dependiendo del valor de la velocidad en X de la entidad en el momento de la actualización del sistema de físicas.

Valor de *VX*	**Valor de *LF***
Positivo	1
Negativo	0

Además, con la introducción del componente posición mundo (*WX*, *WY*) el sistema de físicas dejará de actualizar la posición en pantalla de las entidades para modificar su posición en el mundo.

Los parámetros que recibe ahora el sistema de físicas son:

- Para acceder a la posición de mundo de las entidades, **puntero** a la dirección de memoria RAM correspondiente al primer componente del **array RC**.

- Para acceder a la velocidad de las entidades, **puntero** a la dirección de memoria RAM correspondiente al tercer componente (velocidad en Y) del **array RC**.
- **Número de entidades** que existen en ese momento.

En el caso concreto de este sistema, es más eficiente para la CPU trabajar con dos punteros a dos posiciones diferentes de un mismo array. De esta forma evitamos parte de los desplazamientos entre componentes del array.

10.3.5 Mánager de estados

Controla los nuevos componentes ***ST*** y ***FR*** de nuestras entidades. Estos dos bytes contienen la información del estado y el instante (frame) de ese estado en el que se encuentra una entidad.

Para cada entidad se han definido en este mánager una serie de estados. Las entidades serán diferenciadas unas de otras por el nuevo componente ***T*** de las entidades.

Cada estado está compuesto de 5 bytes que representan: duración en frames del estado (1 byte), puntero a la tabla de velocidades en X para ese estado (2 bytes) y puntero a la tabla de velocidades en Y para ese estado (2 bytes).

Tipo de entidad	Estado	Tipo de dato	Valor
Caballero	Estático (*)	Frames duración	0
		Tabla VX	Tabla estático
		Tabla VY	Tabla de gravedad
	Muerto	Frames duración	90
		Tabla VX	Tabla velocidad 0
		Tabla VY	Tabla velocidad 0
	Salto	Frames duración	25
		Tabla VX	Tabla estático
		Tabla VY	Tabla de salto

	Embestida a izquierda	Frames duración	8
		Tabla VX	Tabla embestida izq.
		Tabla VY	Tabla velocidad 0
	Embestida a derecha	Frames duración	8
		Tabla VX	Tabla embestida dcha.
		Tabla VY	Tabla velocidad 0
	Rebota muro a izquierda	Frames duración	25
		Tabla VX	Tabla rebote pared izq.
		Tabla VY	Tabla de salto
	Rebota muro a derecha	Frames duración	25
		Tabla VX	Tabla rebote pared dcha.
		Tabla VY	Tabla de salto
	Rebote aéreo fase 1	Frames duración	15
		Tabla VX	Tabla estático
		Tabla VY	Tabla estático
	Rebote aéreo fase 2	Frames duración	25
		Tabla VX	Tabla estático
		Tabla VY	Tabla de salto
Sierra	Movimiento positivo (*)	Frames duración	0
		Tabla VX	Tabla movimiento dcha.
		Tabla VY	Tabla velocidad 0
	Movimiento negativo	Frames duración	0
		Tabla VX	Tabla estático
		Tabla VY	Tabla de gravidad

Lanza	Movimiento positivo (*)	Frames duración	0
		Tabla VX	Tabla velocidad 0
		Tabla VY	Tabla movimiento izq.
	Movimiento negativo	Frames duración	0
		Tabla VX	Tabla velocidad 0
		Tabla VY	Tabla movimiento dcha.
Abeja	Estático (*)	Frames duración	0
		Tabla VX	Tabla velocidad 0
		Tabla VY	Tabla velocidad 0

Los estados marcados con (*) son **el estado por defecto** de la entidad y al que se cambia automáticamente cuando la entidad agota los frames de su estado actual.

Los estados con **duración** 0 frames no finalizan de manera automática, lo harán cuando el mánager de estados procese una petición de cambio de estado válida para la entidad.

Las **tablas** son un conjunto de bytes que corresponden a la nueva velocidad de la entidad en uno de los ejes para cada frame del estado. Veamos como ejemplo la **tabla de salto**:

```
JUMP_TABLE:
.START:
DB  $00,-$00,-$01,-$01,-$01,-$01,-$01,-$01,-$01,-$01 ; Frames 16 a 25
DB -$02,-$02,-$02,-$02,-$02,-$02,-$02,-$02,-$02,-$02 ; Frames 6 a 15
DB -$02,-$02,-$02,-$02,-$02                          ; Frames 1 a 5
.END:
```

El byte marcado en azul corresponde al valor de la nueva velocidad en Y del primer frame de los estados "Salto", "Rebota muro a izquierda", "Rebota muro a derecha" y "Rebote aéreo fase 2". El byte marcado en naranja corresponde al valor de la nueva velocidad en Y del frame final de esos mismos estados.

En la **tabla velocidad 0** los bytes son siempre **$00**, es decir sirve para detener el movimiento de la entidad.

En la **tabla estático** los bytes son **$80**, este valor lo utilizamos cuando queremos que la velocidad de la entidad no sea modificada.

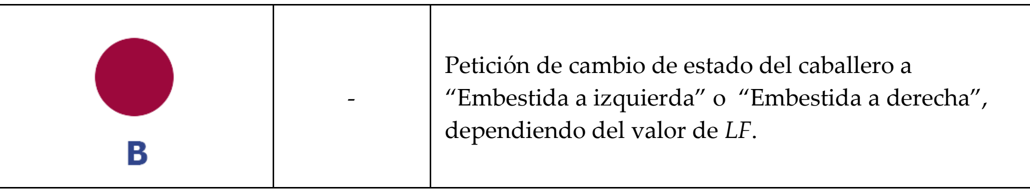
Para cumplir con sus tareas, el mánager de nivel ofrece en cada frame dos parámetros al sistema de animaciones:

- **Puntero** a la dirección de memoria RAM correspondiente al primer componente del **array sOAM**, para actualizar los cuatro *SpriteNum* del caballero.
- **Puntero** a la dirección de memoria RAM correspondiente al primer componente del **array RC,** para consultar el estado (*ST*) del caballero.

10.3.6 Sistema de input

Las acciones que tomará el sistema de input respecto al hito anterior, son ahora, algo diferentes:

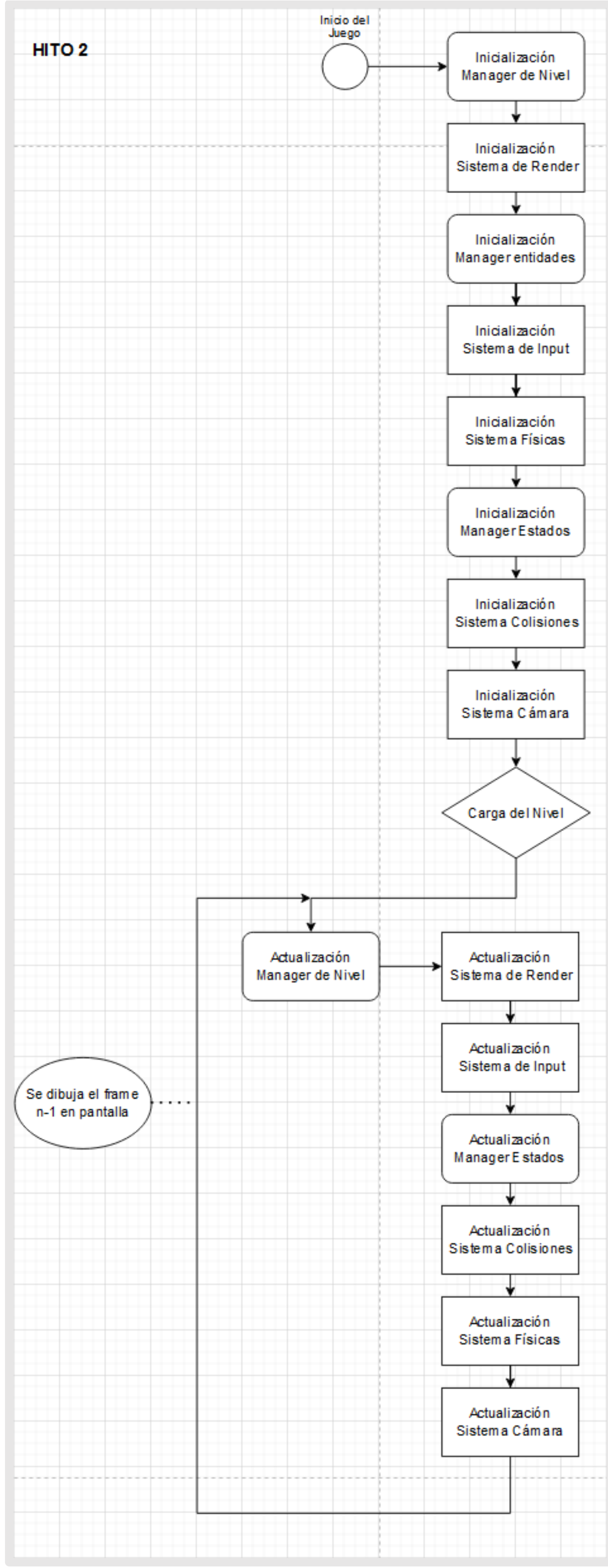
Botón	Dirección	Valor de la velocidad
	Arriba	Sin acción.
	Abajo	Petición de cambio de estado del caballero a "Rebote aéreo fase 1"
	Izquierda	VX = -1
	Derecha	VX = +1
A	-	Petición de cambio de estado del caballero a "Salto", "Rebote muro a izquierda" o "Rebote muro a derecha", dependiendo de los valores *CX* y *CY*.



B	-	Petición de cambio de estado del caballero a "Embestida a izquierda" o "Embestida a derecha", dependiendo del valor de *LF*.

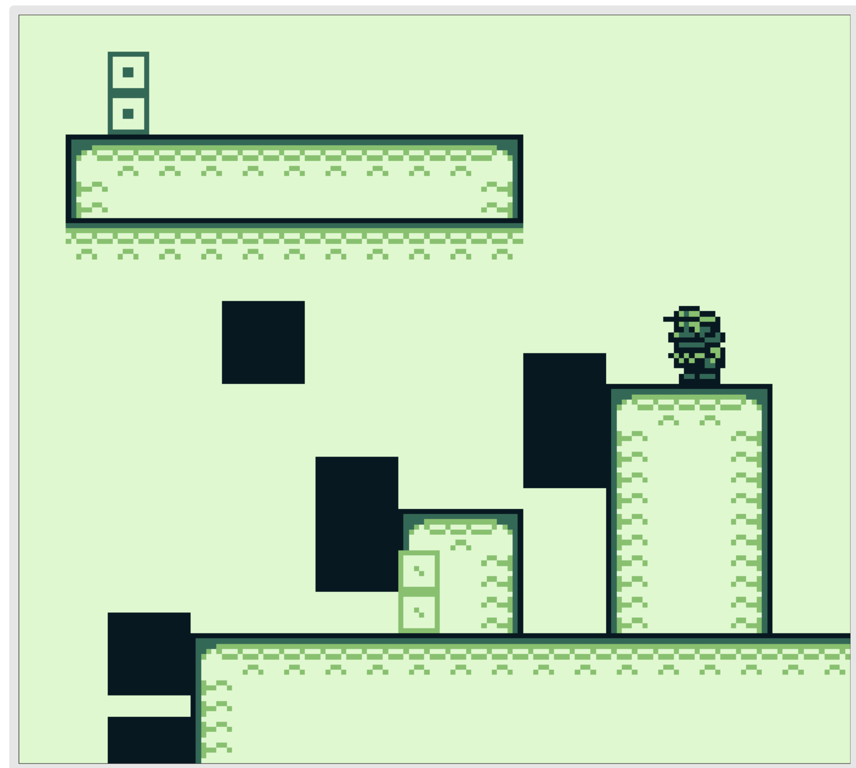
10.3.7 **Mánager de nivel**

Respecto al hito anterior, esto son los cambios en el flujo del juego:



Flujo del juego al término del hito 2.

10.3.8 **Estado final**



Captura del estado del juego al término del hito 2.

Todas las entidades detectan colisiones con los tiles del mapa de fondo.

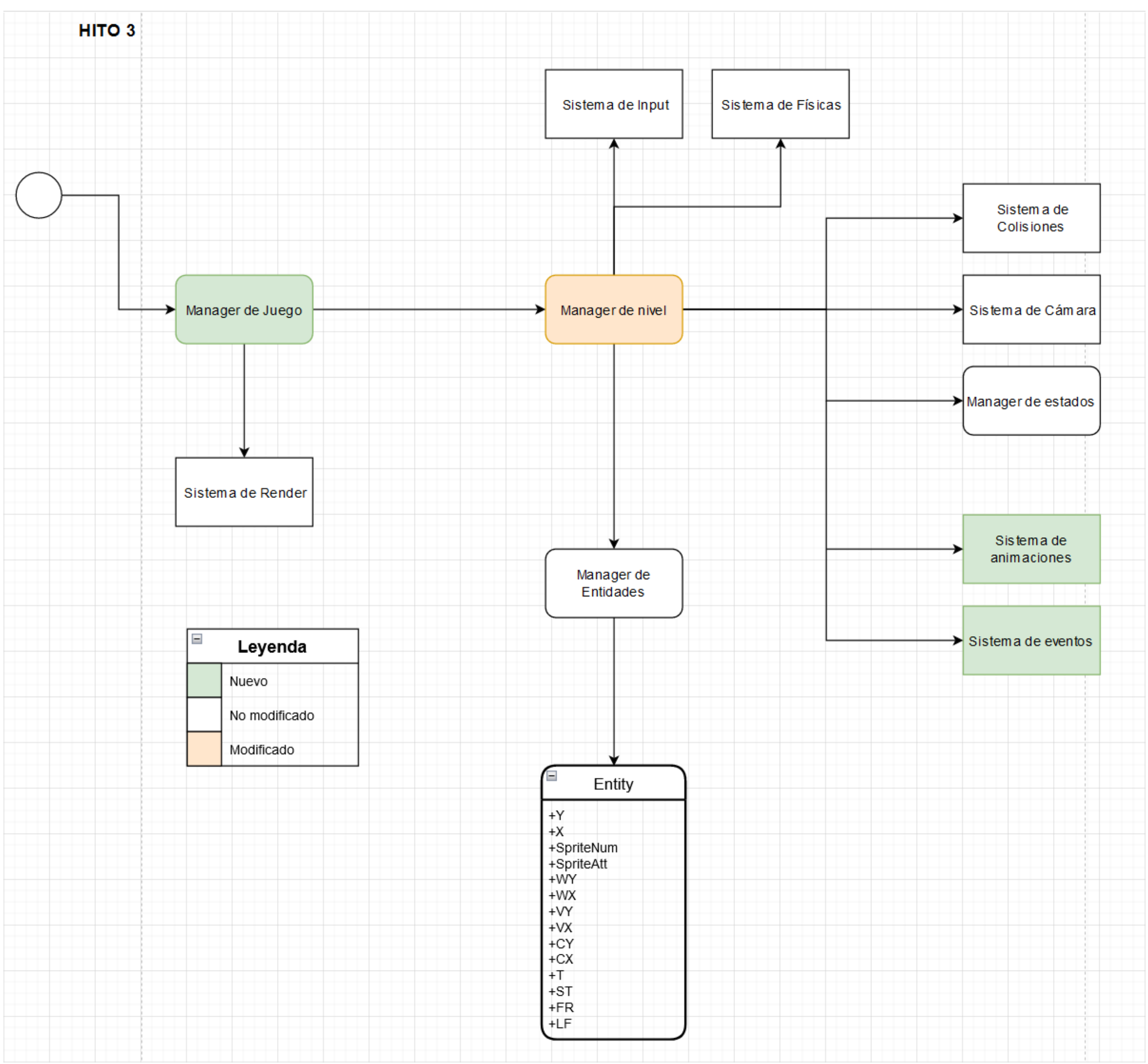
El caballero, para el que estamos utilizando temporalmente un sprite de Mario, puede recorrer el nivel con los movimientos lateral, de salto, de embestida, rebote con pared y rebote aéreo. También es afectado por la gravedad. El área visible en pantalla sigue el movimiento del jugador siempre que es posible.

En el mapa de fondo que utilizamos en este punto no se han llegado a definir tiles de tipo delimitador, por lo que, las entidades controladas por la IA (en negro) recorren el nivel con velocidad constante hasta detenerse al colisionar con un bloque de colisión tipo normal.

Cuando el caballero colisiona fuera del movimiento rebote aéreo con alguna de las entidades controladas por la IA, o si colisiona con tiles mortales del mapa de fondo, entra en estado de muerto. El estado muerto detiene todo movimiento del personaje durante los frames que se han establecido para su duración.

10.4 Hito 3. Diseño visual y contenido

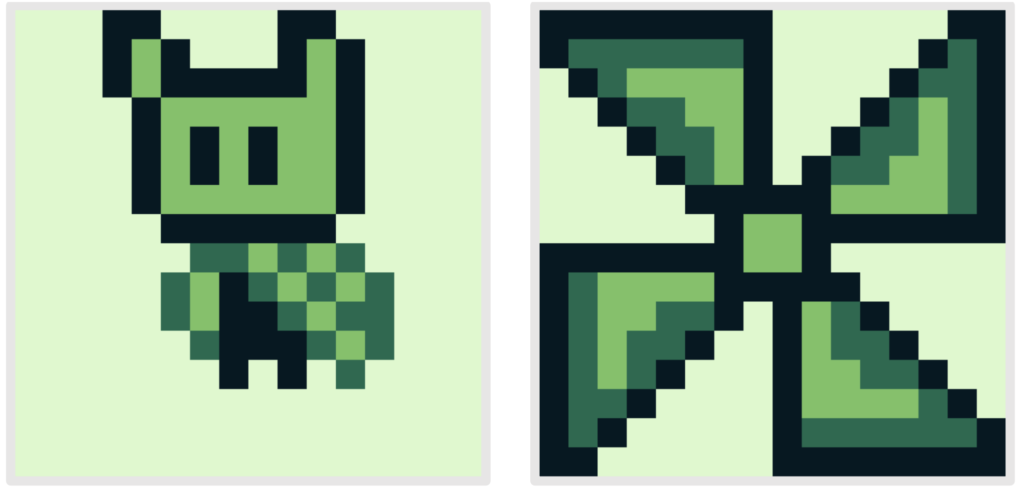
El **hito 3** tiene como objetivo disponer de una versión prototipo del juego donde se vean todos sus elementos funcionando en conjunto. También se introducirá en este hito una primera versión del diseño visual de entidades y niveles.



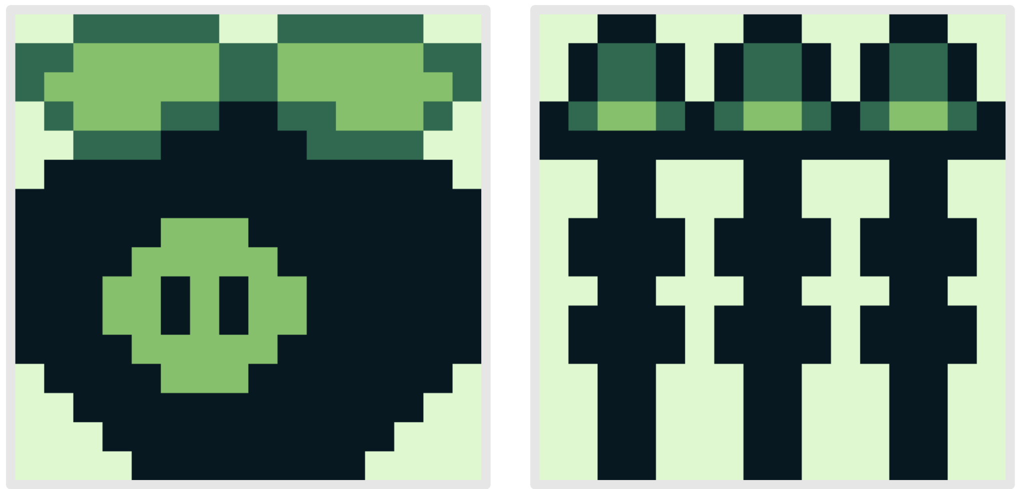
Estructura del motor ECS al término del hito 3.

10.4.1 Diseño visual

Se han diseñado sprites para los cuatro tipos de entidades de nuestro juego:

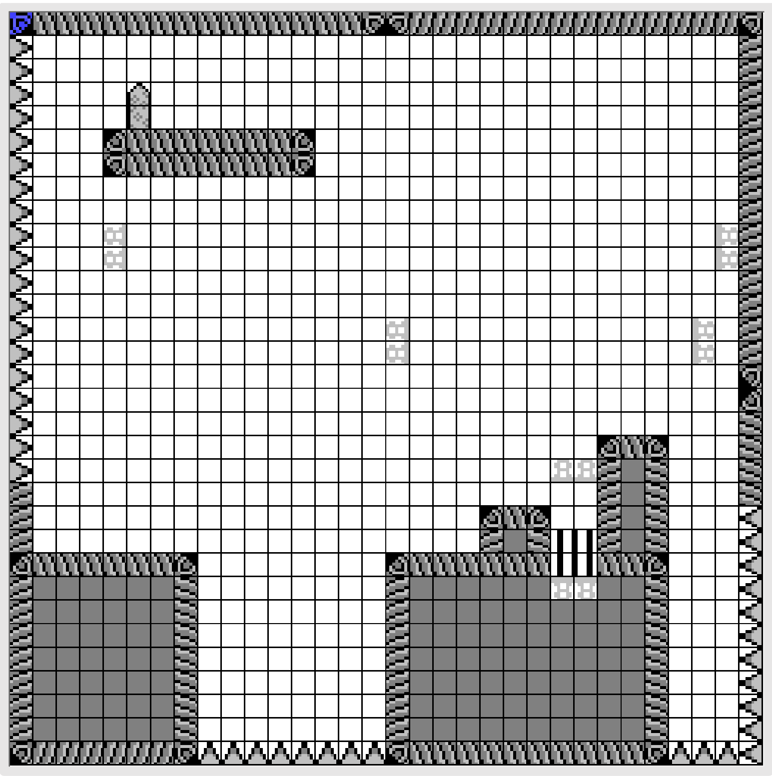


Diseños del caballero y la sierra.

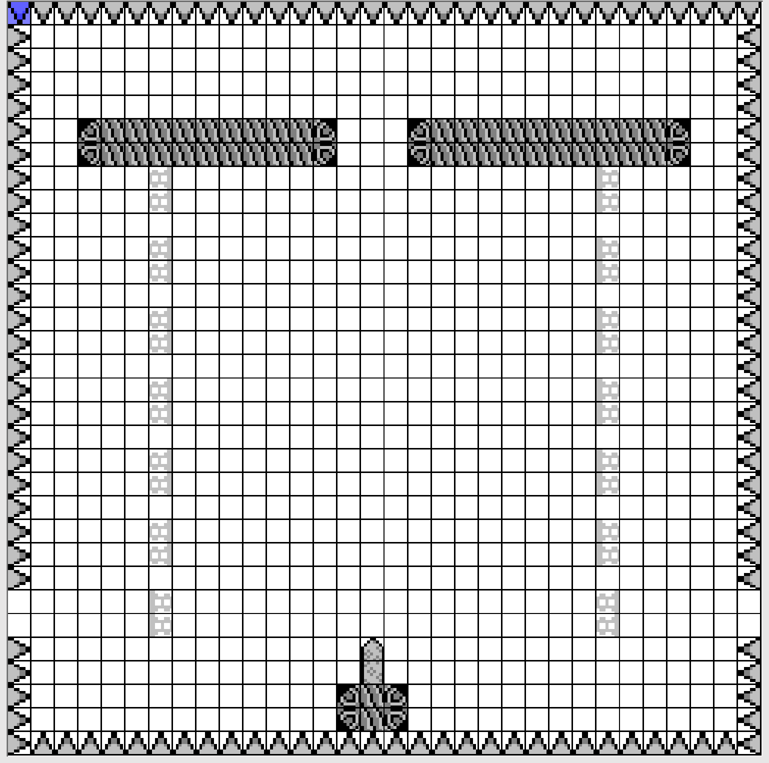


Diseños la abeja y las lanzas.

Se han diseñado también diferentes mapas de fondo para cada nivel. Estos son dos:

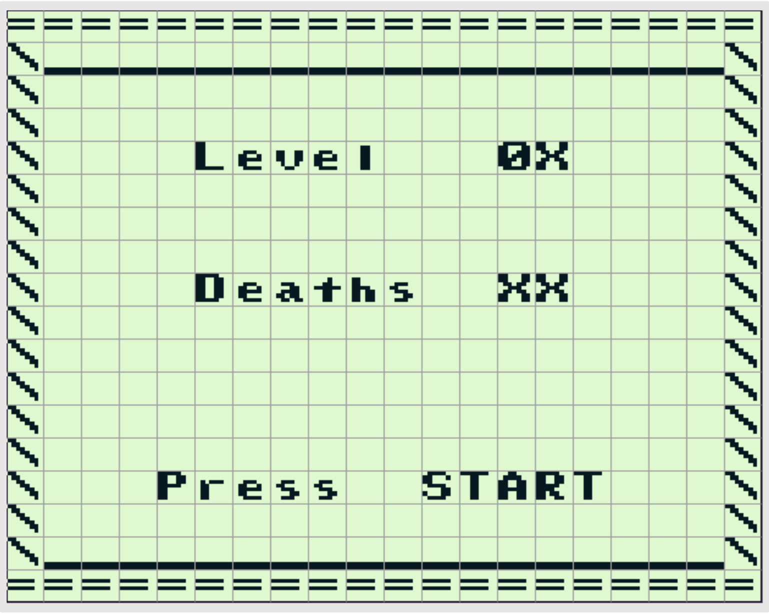


Mapa fondo 1.



Mapa fondo 2.

Para la ventana, que hemos comenzado a utilizar para este hito, hemos diseñado el siguiente mapa de tiles:



Mapa de tiles correspondiente a la ventana.

La tabla de tiles al término del hito es la siguiente:

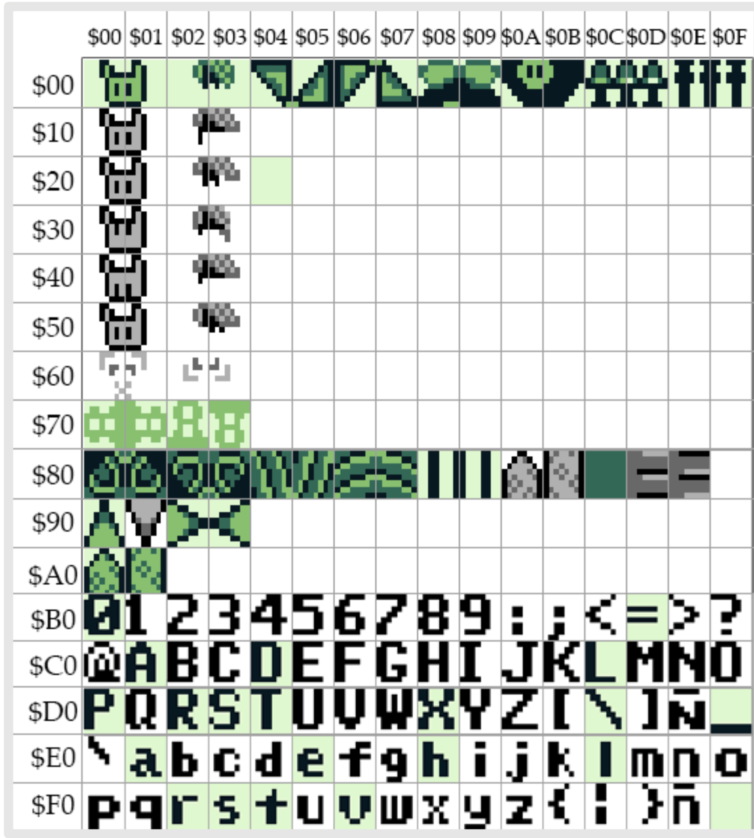


Tabla de tiles al término del hito 3.

10.4.2 **Sistema de eventos de nivel**

Se encarga, en cada frame, de revisar el estado de ejecución del nivel.

Los dos estados de ejecución del nivel son: **en juego** y **finalizado**.

Estado de nivel	Valor de *LF*
En juego	0
Finalizado	1

Cuando el estado de ejecución del nivel cambia, el sistema de eventos de nivel lanza un evento. Actualmente disponemos de dos tipos de evento:

Transición	Evento
En juego -> Finalizado	Mostrar ventana en pantalla e iniciar carga de siguiente nivel.
Finalizado -> En juego	Quitar la ventana de la pantalla.

Al inicio del juego, habremos cargado en la ventana el mapa de tiles visto en el punto anterior. Cuando el estado del nivel "En juego" finaliza se cargan en las posiciones que ocupan las "X" los valores almacenados en RAM con el nivel y número de muertes del caballero actuales.

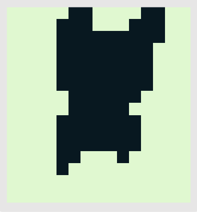
Área visible en pantalla al finalizar un nivel. Mostrando la ventana.

10.4.3 **Sistema de animaciones**

Hemos creado el sistema de animaciones para que se encargue de modificar los sprites que forman la entidad del caballero en pantalla dependiendo del estado en el que se encuentra.

Estado	Composición de sprites	
	VX igual a 0	*VX* distinta de 0
Estático		
Salto Rebote muro a izquierda Rebote muro a derecha Rebote aéreo fase 2		
Embestida a izquierda Embestida a derecha		
Muerto		

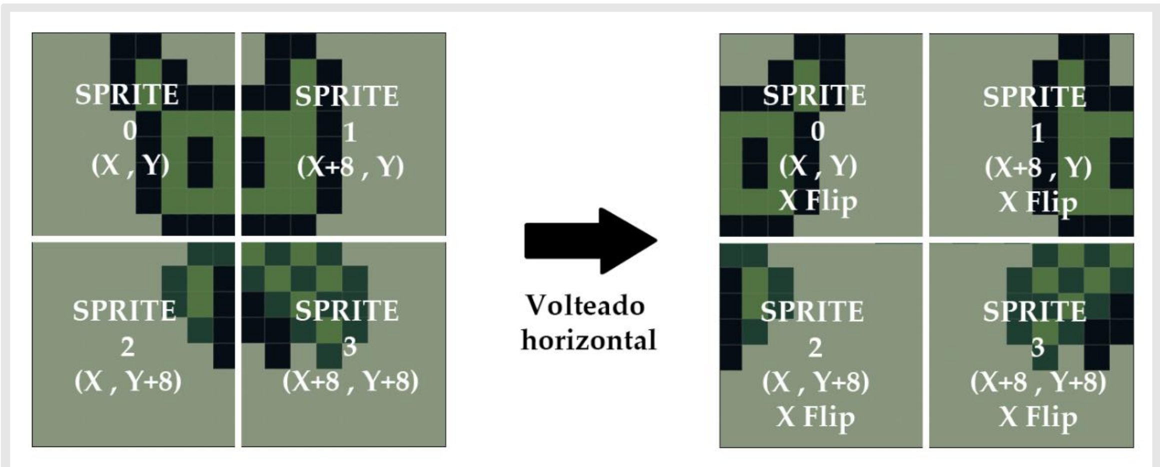
Si el caballero se encuentra en estado "Rebote aéreo fase 1" no se modifican los sprites que componen a la entidad, pero sí la paleta con la que se dibuja (a travñes del componente *SpriteAtt*) para que el personaje se vea completamente en negro:



Sprites correspondientes al estado salto con paleta modificada.

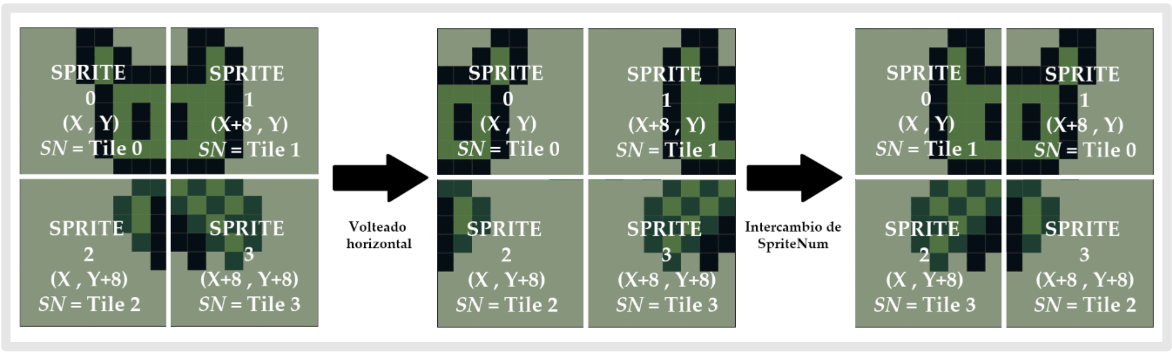
El volteado horizontal de los sprites se realiza atendiendo al valor del componente *LF*, con la información de la dirección a la que mira la entidad.

Simplemente con un volteado de cada uno de los cuatro sprites de una entidad no obtendremos el resultado que buscamos.



Volteado horizontal de los cuatros sprites de una entidad.

Para la correcta aplicación de esta transformación es necesario intercambiar el valor *SpriteNum* de los sprites 0 y 1 y los sprites 2 y 3.



Volteado horizontal + Intercambio de SpriteNum.

Para desempeñar esta terea, el level manager ofrece al sistema de animaciones:

- Para acceder a la posición de mundo de las entidades, **puntero** a la dirección de memoria RAM correspondiente al primer componente del **array RC**.
- Para acceder a la velocidad de las entidades, **puntero** a la dirección de memoria RAM correspondiente al tercer componente (velocidad en Y) del **array RC**.
- **Número de entidades** que existen en ese momento.

10.4.4 Mánager de nivel

Al inicio del nivel almacenará en cuatro bytes de RAM las posiciones iniciales X e Y en el mapa de fondo del caballero y la cámara. El mánager de estados se sirve de estos cuatro bytes para restablecer las posiciones del caballero y la cámara cuando el caballero muere.

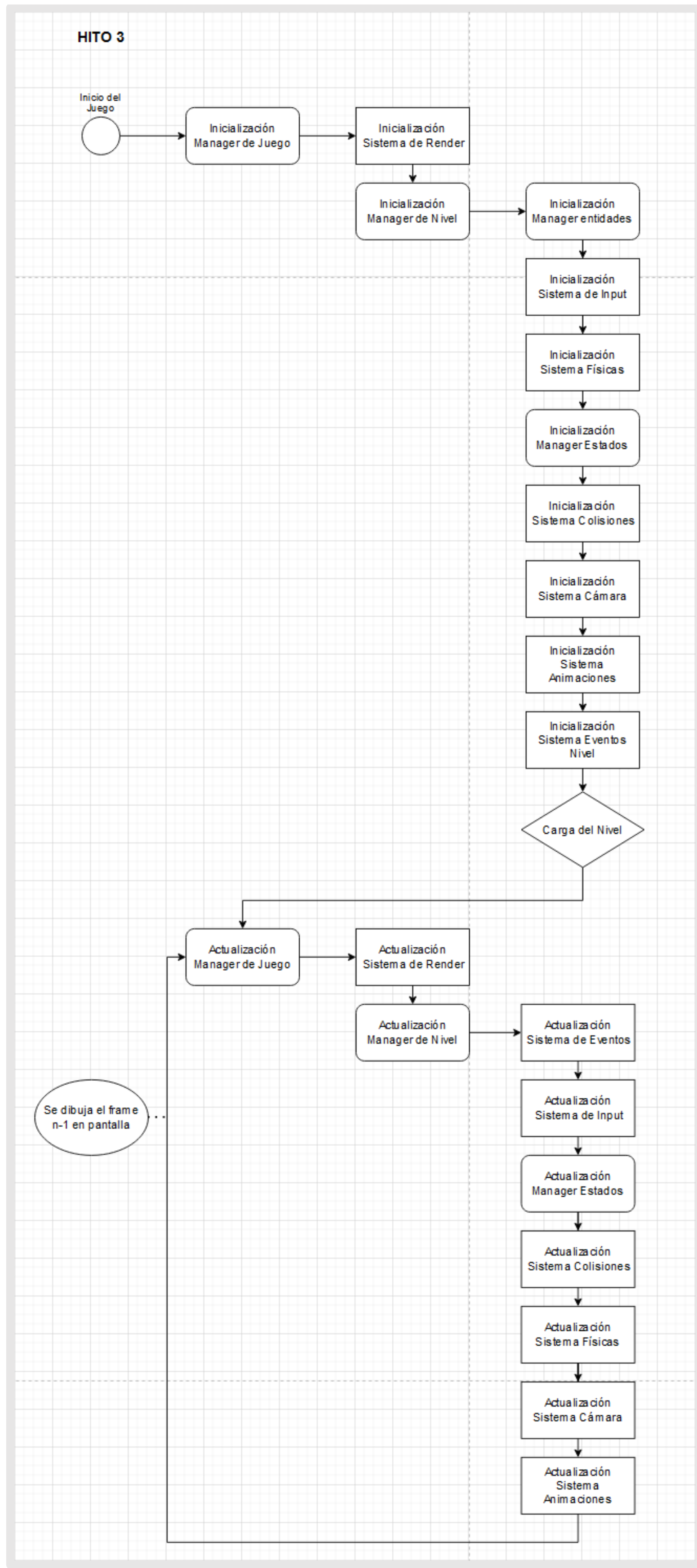
Almacena también el contador de muertes del caballero para el nivel actual. El contador aumenta en uno con cada muerte.

10.4.5 Mánager de juego

El mánager de juego controla el número de nivel actual y tiene acceso al índice de niveles que relaciona cada nivel con las direcciones de memoria ROM donde se encuentran los datos del mapa de tiles a cargar y las entidades a crear al inicio de ese nivel. Al cargar un nivel, le pasa al mánager de nivel un puntero a la dirección de memoria ROM que almacena el mapa de tiles a cargar en el fondo para el nivel actual, y otro puntero a la dirección de memoria ROM que almacena los datos de las entidades a cargar para el nivel actual.

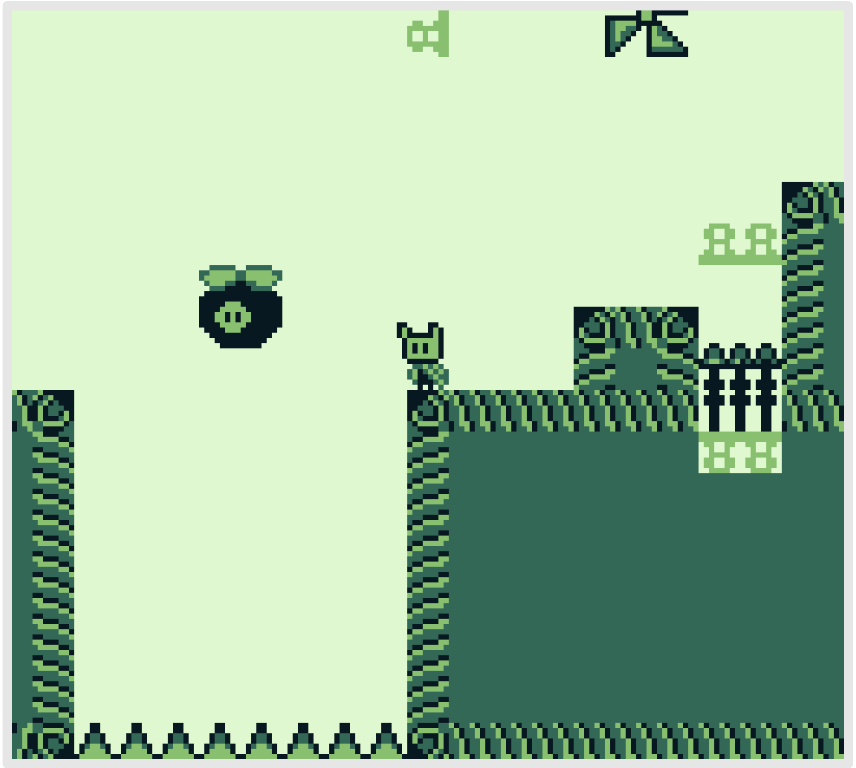
Además, almacena el contador total de muertes del caballero en la partida. Cuando un nivel entra en estado "Finalizado" el mánager de nivel actualiza el valor de ese contador sumándole el valor del contador de muertes del nivel finalizado.

Suplanta al mánager de nivel en el papel de controlador del inicio y flujo del juego.



Flujo del juego al término del hito 3.

10.4.6 **Estado final**



Captura del estado del juego al término del hito 3.

El jugador puede recorrer el nivel haciendo uso de todos los movimientos del caballero con el objetivo de alcanzar el punto final de nivel, marcado por los tiles de tipo victoria. Una vez alcanzado el punto final podrá ver el número de muertes que ha sufrido hasta superar el nivel.

En cada nivel se distribuyen una serie de obstáculos que dificulten la llegada del caballero al punto final. Estos obstáculos pueden ser tiles de colisión normal y mortal del mapa de fondo o entidades enemigas, con y sin movimiento.

El juego en su estado actual está compuesto de seis niveles diferentes.

11. Conclusiones

La idea de desarrollar un videojuego íntegramente en ensamblador para una consola retro puede sonar alocada para muchos. Incluso a mí me lo parecía en un inicio, pero el reto era lo suficientemente atractivo como para mitigar el miedo a tomar el proyecto. Hoy día no puedo estar más feliz de haber tomado finalmente la decisión de realizar el proyecto pues este abarca gran parte de los ámbitos aprendidos a lo largo de la carrera y me ha servido como afianzador de todos esos conocimientos a la vez que me ha permitido desarrollar nuevas facetas como programador.

Para el proyecto, un objetivo imprescindible era analizar y entender las capacidades y limitaciones de la Game Boy. En mi desarrollo he intentado plasmar que no solo he cumplido este objetivo, sino que también me he atrevido a ir más allá. Desde el inicio del proyecto y teniendo en mente las ventajas que ofrece trabajar a bajo nivel he tomado una mentalidad centrada en la **optimización y explotación de los recursos que ofrece la consola**.

El programar bajo la arquitectura ECS me ofrece ahora la **oportunidad de trasladar mi juego a otras plataformas retro** con costes bajos de re-implementación, como puede ser el Amstrad CPC, con un lenguaje ensamblador similar en muchos aspectos al de Game Boy.

Este proyecto también me ha servido como **refuerzo para mi capacidad de adaptación a nuevas tecnologías**. Investigar a fondo y preparar un desarrollo para una consola como la Game Boy puede suponer una tarea ardua para muchos, pues en comparación a los entornos de desarrollo moderno, las fuentes son mucho más escasas y se requiere un esfuerzo extra en construir un entorno de trabajo cómodo y fluido.

En definitiva, puedo afirmar que un proyecto de esta índole es una experiencia de aprendizaje de la que se beneficiaría cualquier aspirante a programador de videojuegos.

12. Anexo I: Especificaciones técnicas Game Boy

Microprocesador	Sharp LR35902
Frecuencia de reloj	4,19 MHz
RAM	8 kB
Vídeo RAM	8 kB
Dimensiones	148x90x32 mm
Peso	220 g
Consumo de energía	78 – 80 mA hr. 4 pilas AA (1,5 V)
Tiempo de juego	Hasta 30 horas
Tipo de pantalla	LCD
Tamaño de la pantalla	4,7x4,3 cm
Resolución de la pantalla	160x144 píxeles
Colores	4 tonos de verde
Sonido	Estéreo, 4 canales

Especificaciones técnicas Game Boy (Nintendo 1999)

13. Anexo II: transferencia DMA.

La transferencia DMA (Direct Memory Acces) a OAM se trata de un copiado en bloque de una determinada área de la memoria ROM o RAM al espacio de memoria OAM. Es el método de carga de datos a OAM recomendado porque, con una duración de solo 160 microsegundos, es el más rápido y eficiente.

La transferencia DMA a OAM se inicia cuando se escriben datos al registro de entrada/salida **$FF46.**

```
 ; Primero cargamos $C1 al registro DMA ($FF46)
  LD      A, $C1
  LD      [$FF46], A

  ; Comienza la transferencia DMA. Esperamos 160 microsegundos a que acabe
  ; el bucle siguiente tiene exactamente esa duración
  LD      A, 40
.bucle:
  DEC     A
  JR      NZ, .loop
  RET
```

El valor cargado a la dirección $FF46 es el identificador de la dirección de memoria inicial de los datos que queremos copiar a la OAM. En el caso de ejemplo se ha cargado el valor $C1 que se resuelve por la transferencia como la dirección $C100. Siempre resuelve el valor del byte bajo de la dirección como $00, por lo que los datos que queremos copiar a OAM tienen que estar siempre situados a partir de una dirección cuyos ocho bits más a la derecha valen 0. Tras cargar el valor correspondiente en $FF46 tendremos que esperar 160 microsegundos a que esta finalice.

Una vez da comienzo una transferencia DMA, y durante su duración, para la CPU solo es accesible el espacio de memoria **HRAM**. Por ello, para que el código expuesto arriba pueda funcionar correctamente, necesitamos almacenarlo en la HRAM.

```
SECTION "Rutina OAM DMA", ROM0
CopiaRutinaDMA:
   LD   HL, RutinaDMA
   LD   B,  RutinaDMA.end - RutinaDMA   ; Número de bytes a copiar
   LD   C,  LOW(hOAMDMA)                 ; Byte bajo de la dirección destino
.copia:                                  ; Bucle de copia a HRAM de la rutina
   LD   A, [HL+]
   LDH  [C], A
   INC  C
   DEC  B
   JR   NZ, .copia
   RET

RutinaDMA:
   LDH  [$FF46], A
   LD   A, 40
.bucle:
   DEC  a
   JR   NZ, .bucle
   RET
RutinaDMA.end:


SECTION "OAM DMA", HRAM
hramRutinaDMA::
   DS RutinaDMA - RutinaDMA.end ; Reservamos espacio al que copiar la rutina
```

Este código se encarga de trasladar la rutina DMA (marcada en azul) a un espacio reservado de la memoria HRAM. La etiqueta "hramRutinaDMA" equivale a la dirección de memoria HRAM que hemos reservado especialmente para almacenar la rutina (líneas marcadas con morado).

El código con las marcas naranjas se trata de la rutina que copia byte a byte toda la rutina DMA al espacio HRAM reservado.

En el caso de nuestro juego, ejecutaremos la rutina de copia "CopiaRutinaDMA" en el momento en que iniciemos nuestro sistema de render.

```
RENDERSYS_INIT::
; ---- Routine to initialize LCD state, Palettes, OAM & VRAM.
; DESTROYS:
;       A, HL, DE, BC
;
    ; Move DMA subroutine into HRAM
    CALL    CopiaRutinaDMA

    ; ...

    RET
```

La actualización del sistema de render se encargará de llamar a la rutina DMA en HRAM para que copie a OAM nuestro array sOAM.

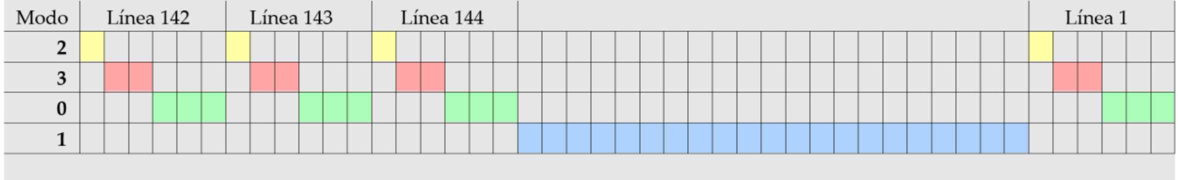
```
RENDERSYS_UPDATE::
; ---- Updates the Object Attribute memory.
; PARAMETERS:
;       HL = Pointer to first component in the sOAM array.
; DESTROYS:
;       A
;
    LD      A, HIGH(HL)
    CALL    hramRutinaDMA     ; CALL HRAM DMA routine
    RET
```

14. Anexo III: Optimización VBLANK.

Para controlar el LCD y realizar los cálculos correspondientes para el dibujado/renderizado de imagen en pantalla, Game Boy dispone de la **PPU** (**Picture Processing Unit**), un microprocesador que trabaja de manera paralela a la CPU.

La PPU pasa por cuatro modos de trabajo (modos 0, 1, 2 y 3) a lo largo del tiempo entre el dibujado de una imagen completa (frame) y la siguiente (Nintendo 1999; Antonio Niño Díaz et al 2021):

- Para el dibujado de cada una de las 144 filas de píxeles que componen la pantalla de Game Boy, la PPU cambia (en un orden fijo) entre los **modos 0, 2 y 3**. El modo 0 se conoce también como **HBlank**.
- Al **modo 1** se accede una vez se han dibujado el total de 144 filas de píxeles y tiene una duración equivalente al dibujado de 10 de esas filas. El modo 1 se conoce también como **VBlank**.



Segmento del comportamiento de la PPU

En la figura se observa el período de tiempo en el que se dibujan las últimas filas de píxeles de la pantalla. Tras esto, y como ya hemos mencionado, la PPU entra en modo 1 o VBlank durante un tiempo equivalente al dibujado de 10 filas de píxeles y luego comienza con el dibujado del siguiente frame, por la primera fila de la pantalla. En el dibujado de una línea el orden en el que la PPU accede a cada uno de los modos es el que se ve en la figura:

1. Modo 2, la línea va a ser dibujada.
2. Modo 3, la línea se está dibujando.
3. Modo 0, la línea ya se ha dibujado.

Las acciones, duración y restricciones de cada uno de los modos son las siguientes:

Modo	Acción de la PPU	Duración	Memoria de vídeo accesible por CPU
2	Buscando en la OAM por los objetos con coordenada Y que coincida con la línea actual	**80 dots** (19 µs)	VRAM, paletas CGB
3	Leyendo OAM y VRAM para generar la imagen	**168 to 291 dots** (entre 40 y 60 µs), dependiendo del número de sprites a dibujar	Ninguna
0	Ninguna (HBlank)	**85 to 208 dots** (entre 20 y 49 µs), dependiendo de la duración del modo 3 inmediatamente anterior	VRAM, OAM, paletas CGB
1	Ninguna (VBlank)	**4560 dots** (1087 µs, 10 scanlines)	VRAM, OAM, paletas CGB

Siendo un *dot* el período más corto durante el cual la PPU puede generar un píxel.

Teniendo en cuenta esto, presentamos ahora el pseudocódigo del bucle principal generalmente recomendado para un juego de Game Boy:

```
; **** Main Game Loop ****
Main:
        halt                    ; stop system clock
                                ; return from halt when interrupted
        nop                     ; No operation

        ld      a,(VblnkFlag)
        or      a               ; V-Blank interrupt ?
        jr      z,Main          ; No, some other interrupt

        xor     a
        ld      (VblnkFlag),a   ; Clear V-Blank flag

        call    Controls        ; button inputs
        call    Game            ; game operation

        jr      Main

; **** V-Blank Interrupt Routine ****
Vblnk:                          ; Memory position $0040
        push    af
        push    bc
        push    de
        push    hl

        call    SpriteDma       ; Do sprite updates

        ld      a,1
        ld      (VblnkFlag),a   ; VblnkFlag = 1

        pop     hl
        pop     de
        pop     bc
        pop     af
        reti
```

Como se puede ver en el bucle *Main* del código presentado, concretamente en la primera de las líneas marcadas en azul, se utiliza la instrucción `HALT`. Esta instrucción obliga a la CPU a entrar en modo de bajo consumo hasta que el momento en que se tiene que atender una interrupción. El curso del programa se detiene en la instrucción `HALT` hasta atender una interrupción.

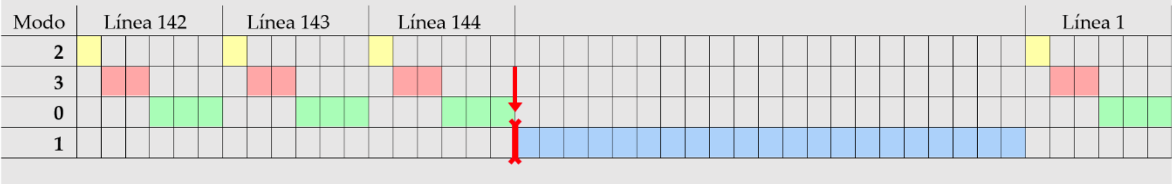
Una **interrupción** es una solicitud para que el procesador interrumpa el código que se está ejecutando actualmente. En Game Boy se pueden activar hasta cinco tipos de interrupciones:

<table>
<tr><th>Interrupción</th><th colspan="2">Solicitud</th><th>Manejador / Interrupt Handler</th></tr>
<tr><td>VBlank</td><td colspan="2">Cada vez que la Game Boy entra en VBlank (modo 1 de la PPU)</td><td>$0040</td></tr>
<tr><td rowspan="4">STAT</td><td rowspan="4">Según configuración</td><td>Cada vez que se va a dibujar la fila de píxeles especificada en la posición de memoria $FF45</td><td rowspan="4">$0048</td></tr>
<tr><td>Cada vez que la PPU entra en modo 2</td></tr>
<tr><td>Cada vez que la PPU entra en modo 1 (VBlank)</td></tr>
<tr><td>Cada vez que la PPU entra en modo 0 (HBlank)</td></tr>
<tr><td>Timer</td><td colspan="2">Cada vez que el temporizador incorporado se desborda (es decir, cuando el contador del temporizador excede el valor 255)</td><td>$0050</td></tr>
<tr><td>Serial</td><td colspan="2">Cuando se completa la transferencia de un byte en la transferencia en serie de datos entre dos Game Boy mediante un Link Cable.</td><td>$0058</td></tr>
<tr><td>Joypad</td><td colspan="2">Cada vez que se presiona un botón del joypad</td><td>$0060</td></tr>
</table>

Al atender una solicitud de interrupción, la CPU realiza una llamada (equivalente a la instrucción `CALL`) a la dirección de memoria del manejador de dicha interrupción. En esta dirección podremos especificar un conjunto de instrucciones específico para que sean ejecutadas cada vez que se produce la interrupción. Cada interrupción tiene asociada un manejador cuya dirección de memoria especificamos en la columna "*Manejador / Interrupt Handler*" de la tabla anterior.

Volviendo al código expuesto anteriormente, la rutina de nombre "VBlank" (líneas marcadas en naranja) correspondería al código situado en el manejador de la dirección $0040. La rutina se ejecutará cuando la PPU entre en modo 1 (VBlank) y finaliza cuando se alcanza la instrucción `RETI`. Tras esa instrucción final, el procesador continuará ejecutando el código inmediatamente posterior a la instrucción `HALT`.

Las líneas marcadas en azul se encargan de comprobar que la interrupción que se ha atendido ha sido, en específico, la interrupción VBlank. En resumidas cuentas, con este código, los cálculos de nuestro juego comenzarán al inicio del período VBlank, donde la memoria de vídeo es completamente accesible (flecha roja en la siguiente figura).



A partir de ese momento, el tiempo para acceder a la memoria de vídeo sin restricciones es limitado. Una vez finaliza el período VBlank, si se sigue intentando acceder a la memoria de vídeo, los datos que se leídos o escritos en la memoria de vídeo durante los modos 2 y 3 estarán corrompidos.

Si no se desarrolla una forma de gestión del trabajo en los espacios HBlank o modo 0 (de compleja implementación), donde sí se puede acceder a la memoria de vídeo al completo, solo dispondremos del espacio VBlank para leer y escribir en esa memoria.

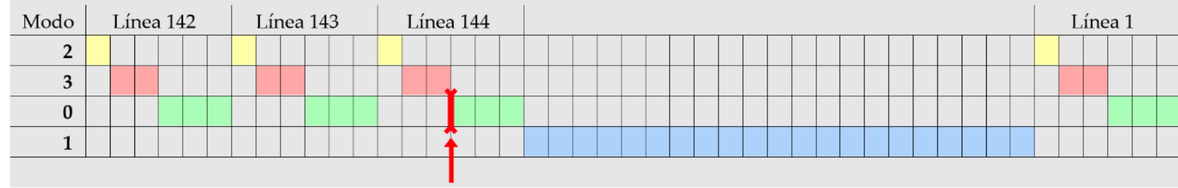
A continuación, planteamos una optimización de sencilla implementación que tiene como objetivo situar el período HBlank de la línea 144 como inicio para los cálculos de nuestro juego, ampliando así el espacio temporal continuado en el que la memoria de vídeo es completamente accesible.

```
;┌───────────────────────────────────────────────────────────────
;│ INTERRUPTIONS
;└
SECTION "STATInterrupt",ROM0[$0048]
    RETI

;┌───────────────────────────────────────────────────────────────
;│ Program START
;└
SECTION "Start", ROM0[$0150]
MAIN:
    DI                          ; Disable interruptions

    LD      SP, $FFFF           ; Point STACK to memory's last position

MAIN.INIT:
    CALL    GAMEMAN_INIT

    ; Set up Interruptions
    LD      A, %00001000        ; ┐
    LDH     [$FF41],A           ; └─> Set up STAT interrupt to proc on mode 0
    LD      A, %00000010        ; ┐
    LDH     [$FFFF], A          ; └─> Enable STAT interrupt
    EI                          ; Enable interruptions
    XOR     A                   ; ┐
    LDH     [$FF0F], A          ; └─> Reset Interrupt flag

MAIN.GAMELOOP:
    HALT    ; Stop system clock
            ; return from halt when interrupted
    ; NOP   ; No Operation (NOP is necessary to avoid HALT skipping)
            ; └─> no need with RGBDS (already adds a NOP after HALT)

    LD      A, [$FF44]          ; ┐
    CP      143                 ; └─> Last line HBLANK ?
    JR      NZ, .GAMELOOP       ; No, some other line

    CALL    GAMEMAN_RENDER      ; Render Game
    CALL    GAMEMAN_UPDATE      ; Update Game

    JR .GAMELOOP
```

Al arranque de nuestro juego, y como vemos en las líneas con la marca azul, configuramos la interrupción STAT para que se lance cuando la PPU entre en modo 0 (HBlank) y la activamos a través de las direcciones de memoria correspondientes ($FF41 y $FFFF). Con la instrucción `EI` activamos las interrupciones y, a continuación, en la dirección $FF0F, que es la dirección con los indicadores de solicitud para las diferentes interrupciones, cargamos 0, es decir, reseteamos los indicadores.

En el bucle principal de nuestro programa, en concreto, en las líneas marcadas en naranja, consultamos la dirección de memoria $FF44, que contiene siempre el valor en el eje Y de la fila de píxeles que se va a dibujar, se está dibujando, o acaba de dibujarse. Cuando este valor sea igual a 143, teniendo en cuenta que su valor inicial es 0, significará que se ha dibujado

la última línea de la pantalla (línea 144) y que nos encontramos en el último espacio HBlank, inmediatamente antes del espacio VBlank (flecha roja en la figura siguiente).



Comenzando a trabajar a partir de este punto, habremos conseguido una "ampliación" del período VBlank de 120 ciclos de CPU, número que disminuirá si aumenta el número de sprites que se tienen que dibujar en la última fila de la pantalla para el frame actual (consultar duración del modo 0 en la primera tabla del anexo). Ese número de ciclos se traduce en una ampliación de entre el **1,9%** y el **4,6%** del período VBlank.

Existe una manera alternativa de implementar esta optimización, sin servirnos de las interrupciones.

```
;┌──────────────────────────────────────────────────────────────────────
;│ Program START
;└
SECTION "Start", ROM0[$0150]
MAIN:
    DI                              ; Disable interruptions

    LD      SP, $FFFF               ; Point STACK to memory's last position

MAIN.INIT:
    CALL    GAMEMAN_INIT

MAIN.GAMELOOP:
    CALL    WAIT_LAST_HBLANK        ; Wait until last line mode 0

    CALL    GAMEMAN_RENDER          ; Render Game
    CALL    GAMEMAN_UPDATE          ; Update Game

    JR .GAMELOOP

;┌──────────────────────────────────────────────────────────────────────
;│ LOCAL FUNCTIONS
;└
WAIT_LAST_HBLANK:
    LD      A, [$FF44]              ; ┐
    CP      143                     ; └─> Last line ?
    JR      NZ, WAIT_LAST_HBLANK    ; No, some other line
.WAIT_MODE0:
    LD      A, [$FF41]              ; ┐
    AND     %00000011               ; └─> Last line HBLANK ?
    JR      NZ, .WAIT_MODE0         ; No, some other mode
    RET
```

Con esta implementación, comprobamos, al inicio de cada iteración del bucle principal de nuestro juego, si nos encontramos en el período HBlank posterior al dibujado de la última fila de píxeles. Para ello, llamamos a la rutina marcada en naranja en el código. La rutina consiste de dos bucles:

- El primero, en naranja, consulta la dirección $FF44 que, como ya hemos mencionado en este anexo, contiene siempre el valor en el eje Y de la fila de píxeles que se va a dibujar, se está dibujando, o acaba de dibujarse. La ejecución se mantendrá en el bucle hasta alcanzar la última fila de píxeles.
- El segundo, en azul, consulta la dirección $FF41 cuyos bits 0 y 1 (siendo el bit 7 el de mayor peso) nos dan la información del modo actual de la PPU. La ejecución se mantendrá en el bucle hasta alcanzar HBlank (modo 0).

Aunque esta segunda implementación es más eficiente en cuanto a ocupación de la **CPU**, pues nos ahorramos en cada frame 143 llamadas al manejador de la interrupción con su respectiva instrucción `RETI`, hay que aclarar que, si cargáramos una ROM con esta segunda implementación a un cartucho y la reprodujéramos con una Game Boy, esta será menos eficiente en cuanto a consumo de **batería** y expandir la vida máxima de la batería. Hacer `HALT` reduce el consumo de energía de la CPU y la ROM (Nintendo 1999), pero, por la naturaleza de este segundo caso, omitimos la instrucción.

15. Bibliografía

Fabian R 2018. *Data-oriented design*.
https://www.dataorienteddesign.com/dodbook/

Booch G 1991. *Object-Oriented Design With Applications*. Benjaming/Cummings.

K Fedoseev et al 2020. *Application of Data-Oriented Design in Game Development*. J. Phys.: Conf. Ser. 1694 012035.
https://iopscience.iop.org/article/10.1088/1742-6596/1694/1/012035/meta

Nintendo 1999. *Game Boy Programming Manual Version 1.1*.

Bircd 2021. *BGB version 1.5.9 Manual*.
https://bgb.bircd.org/manual.html

Copetti R 2019. *Game Boy Architecture. A practical analysis*.
https://www.copetti.org/writings/consoles/game-boy/

Pastraiser n.d. *Intel 8080 instruction set* y *Gameboy CPU (LR35902) instruction set*.
https://www.pastraiser.com/cpu/i8080/i8080_opcodes.html
https://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html

Antonio Niño Díaz et al 2021. *Pan Docs*.
https://gbdev.io/pandocs/About.html

Bentley J et al 2021. *RGBDS v0.5.2 release*.
https://rgbds.gbdev.io/docs/v0.5.2

Trigás M 2012. *Metodología Scrum*.
https://openaccess.uoc.edu/webapps/o2/bitstream/10609/17885/1/mtrigasTFC0612memoria.pdf

Gbdev 2022. *OAM DMA Tutorial*.
https://gbdev.gg8.se/wiki/articles/OAM_DMA_tutorial