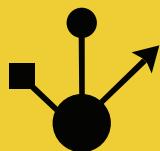




Escuela
Politécnica
Superior

Golden Sacra



Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Ángel Jesús Terol Martínez

Tutor/es:

Francisco José Gallego Durán

Mayo 2020



Universitat d'Alacant
Universidad de Alicante

Golden Sacra

Videojuego para Gameboy en ensamblador gbZ80

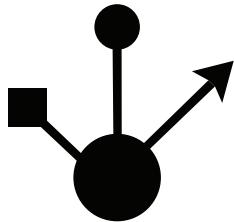
Autor

Ángel Jesús Terol Martínez

Tutor/es

Francisco José Gallego Durán

Departamento de la Ciencia de la Computación e Inteligencia Artificial



Grado en Ingeniería Multimedia



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Mayo 2020

Agradecimientos

El grado de Ingeniería Multimedia llega a su fin para mi después de 5 años de constante esfuerzo. Sin duda ha sido una de las etapas más importantes de mi vida y que siempre recordaré con una sonrisa en la cara. Estos dos últimos años han sido los que más he disfrutado y de los cuales, con lástima, toca despedirse.

A inicios de curso no sabía dónde meterme... Me sentía perdido en la mayor parte de las asignaturas y no tenía la certeza de tener los conocimientos adecuados para sacarlas adelante. Ahora puedo decir que me siento seguro de mí mismo al tener que enfrentarme a los problemas que se me puedan presentar a lo largo del camino. Y esto es gracias al profesorado del cual la Universidad de Alicante dispone (en especial mi tutor) los cuales siempre han tratado de sacar lo mejor de nosotros.

Agradecer esta experiencia también a Carla Maciá, Raquel González, Yaiza Panizo, José Malagón, Alejandro Domenech y Alberto Berenguer, con los que he compartido risas, ilusiones, proyectos, y discusiones. También agradecerles por sacarme (en diferentes ocasiones) de un apuro. No sé qué sería de mí de no haber podido contar con ellos. Tengo el anhelo de poder trabajar juntos de nuevo, pero el futuro es un incierto.

Y como no, a mis padres y hermanas. Siempre me han estado apoyando de todas las maneras que han tenido a su alcance y no hay palabras que puedan expresar lo agradecido que estoy por ello.

*Los videojuegos deben ser una cosa.
Divertidos. Divertidos para todos.*

Satoru Iwata

Índice general

1 Resumen	1
2 Objetivos	3
3 Terminología	5
4 Metodología y Planificación	7
4.1 Mínimo Producto Viable	8
5 Game Design Document	9
5.1 Características	9
5.2 Historia	9
5.3 Mecánicas	9
5.4 Personajes	11
5.4.1 Protagonista	11
5.4.2 Enemigos	12
5.4.2.1 Básicos	12
5.4.2.2 Jefe	13
5.5 Controles	14
5.5.1 Mapa de Mundo	14
5.5.2 Mazmorras	15
5.5.3 Menús	15
5.6 Pantallas	16
5.7 Estados de Juego	18
6 Estado del Arte	19
6.1 Game Boy	19
6.1.1 Especificaciones Técnicas	20
6.2 Análisis del Mercado	21
6.3 Principales Referentes	21
6.3.1 Pokémon Rojo/Azul - Game Boy	21
6.3.1.1 Dibujado de Ventanas Emergentes en Pokémon Rojo/Azul	23
6.3.2 Pokémon Mundo Misterioso: Equipo de Rescate Rojo - Game Boy Advance	24
6.3.3 The Legend Of Zelda: Oracle of Ages - Game Boy Color	25
7 Manual	27
7.1 Creación de una ROM	27
7.2 Herramientas	30
7.2.1 RGBDS	30

7.2.2	GBTD y GBMB	32
7.2.3	Emuladores	33
7.2.4	Flash Cartridge	34
7.2.5	GBSound Sample Generator	34
7.2.6	Sublime Text 3	35
7.3	Tiles	36
7.4	Sprites	37
8	Desarrollo	39
8.1	Iteración 0 - Iniciación en la Programación para Game Boy	39
8.1.1	Hello World	39
8.1.2	Sprites y Scroll	41
8.1.3	Tilemaps	43
8.1.4	Conclusión	43
8.2	Iteración 1 - Primer Prototipo	44
8.2.1	Colisiones	44
8.2.2	Enemigos	46
8.2.3	Paletas	48
8.2.4	Interrupción V-Blank	51
8.2.5	Acceso Directo a Memoria	52
8.2.6	Ventanas	55
8.2.7	Sonido	56
8.2.8	Conclusión	57
8.3	Iteración 2 - Reestructuración y Reimplementación de Código	58
8.3.1	Entity Component System	58
8.3.2	Array de Entidades	59
8.3.3	Conclusión	61
8.4	Iteración 3 - Elementos para un Mínimo Producto Viable	62
8.4.1	HUD	62
8.4.2	Animaciones	63
8.4.3	Colisiones - Casos Especiales	65
8.4.4	Mapas y Estados de Juego	67
8.4.5	Conclusión	68
8.5	Iteración 4 - Implementación de Elementos de Producto	69
8.5.1	Colisiones	69
8.5.2	Progresión de Juego	70
8.5.3	Transiciones entre Estados de Juego	72
8.5.4	Objetos y Hambre	74
8.5.5	Ventanas - Diálogos y Menús	76
8.5.6	Conclusión	79
8.6	Iteración 5 - Diseño de Niveles para Enseñar al Usuario a Jugar	80
8.6.1	Actualización de RGBDS	80
8.6.2	Utilización de Objetos	81
8.6.3	Múltiples Tilesets	82
8.6.4	Inserción de Objetos y Enemigos	85

8.6.5	Conclusión	86
8.7	Iteración 6 - Mejoras Gráficas, Gameplay y Efectos de Sonido	87
8.7.1	Música y Efectos de Sonido	87
8.7.2	Arreglo Visual en Ventanas Emergentes	90
8.7.3	Mejoras en el Flujo de Juego	92
8.7.4	Conclusión	95
9	Anexo	97
9.1	CPU	97
9.1.1	Registros	97
9.1.2	Instrucciones	98
9.2	Mapa de memoria	99
9.3	Organización de la ROM	101
9.4	Inputs	102
9.5	Interrupciones	102
9.6	GPU	104
9.6.1	LCD control	105
9.6.2	LCD STAT	105
9.6.3	LCD Color Display (Solamente CGB)	106
9.6.3.1	LCD Palettes	106
9.7	Sonido	107
9.7.1	Canal 1 - Tono & Portamento	108
9.7.2	Canal 2 - Tono	109
9.7.3	Canal 3 - Onda Programable	109
9.7.4	Canal 4 - Ruido Blanco	110
9.7.5	Registros de Uso General	111
10	Conclusiones Finales	113
10.1	Estado del Producto	113
10.2	Posibles Futuras Mejoras	113
10.3	Opiniones Personales	114
Bibliografía		115

Índice de figuras

4.1 Tabla Trello - Iteración 0	8
5.1 Mockup - Avance de Juego	10
5.2 Mockup - Feedback de Hambre	10
5.3 Diseño de la Protagonista	11
5.4 Diseño los Sprites de la Protagonista	12
5.5 Enemigos Base	12
5.6 Mockup - Diseño de Enemigo Jefe	13
5.7 Mapa de Mundo	14
5.8 Mazmorras	15
5.9 Mockup - Pantalla de Título	16
5.10 Mockup - Diseño de Mapa de Mundo	16
5.11 Mockup - Diseño de Vivienda	17
5.12 Mockup - Diseño de Menú e Inventory	17
5.13 Diagrama de Flujo	18
6.1 Game Boy	19
6.2 Pokemon Azul/Rojo	21
6.3 Pantallas de Pokémon Rojo/Azul	22
6.4 Dibujado de Ventanas en Pokémon Rojo/Azul	23
6.5 Ventanas Emergentes en Pokémon Rojo/Azul	23
6.6 Pokemon Mundo Misterioso: Equipo de Rescate Rojo	24
6.7 Captura de Pantalla In-Game de Pokémon Mundo Misterioso	25
6.8 The Legend Of Zelda: Oracle Of Ages	25
6.9 Pasado y Presente en Oracle Of Ages	26
6.10 Capturas de Pantalla In-Game del Oracle Of Ages	26
7.1 ROM básica ejecutada en emulador	30
7.2 Game Boy Tile Designer	32
7.3 Game Boy Map Builder	32
7.4 Emulador BGB	33
7.5 Emulador NO\$GMB	33
7.6 Game Boy Flash Cartridge	34
7.7 GBSound Sample Generator	34
7.8 Interfaz Sublime Text 3	35
7.9 Capas de la Game Boy	36
7.10 Descripción Gráfica de la OAM	37
8.1 Hello World	39
8.2 Sprite Animado en NO\$GMB	41

8.3	Colisiones Pixel a Pixel	44
8.4	Tamaño de Bloques en el Juego	45
8.5	Primer Diseño de Enemigo	47
8.6	Ataque del Enemigo	48
8.7	Pokémon Rojo/Azul en GBC y GB	49
8.8	Primera Iteración del HUD	55
8.9	Diagrama ECS	58
8.10	Barra de Vida	62
8.11	Sprites Enemigo	64
8.12	Casos Especiales en Colisiones	65
8.13	Mapas Principales	67
8.14	Muestra Gráfica de Progresión de Juego	71
8.15	Problema con la Paleta de Color en GB	72
8.16	Transición en Game Boy Original/Pocket	73
8.17	Estructura de Color en GBC	74
8.18	Problemas Dibujado de Ventanas Emergentes	77
8.19	Dibujado de Textos	79
8.20	Actualización 0.4.0 RGBDS - Código Obsoleto	80
8.21	Inventario	82
8.22	Falta de Memoria ROM	84
8.23	Cargado Erróneo de Tileset	84
8.24	Pantalla Principal de Juego Definitiva	85
8.25	Diseño de Nuevos Enemigos	85
8.26	Diseño de Niveles	86
8.27	Marc Davis - Compositor Musical en Fiverr	90
8.28	Problema Visual Ventanas Emergentes	91
8.29	Arreglo Visual Ventanas Emergentes	93
9.1	Visualización del VRAM	100
9.2	Distribución del PAD	102
9.3	Modos de dibujado de la GPU	104
9.4	Representación Gráfica de los Registros de Especificación y Escritura en Modo CGB	106
9.5	Ejemplo Gráfico de Ciclo Útil de una Onda	107
9.6	Ejemplo Gráfico de Envoltoriente de Amplitud	108

Índice de tablas

6.1	Especificaciones técnicas de la Game Boy original	20
9.1	Función de los bits del registro F o Flags	98
9.2	Mapa de Memoria de la Game Boy	99
9.3	Esquema Memoria ROM	101
9.4	Interrupciones de la Game Boy	103
9.5	Duraciones del Portamento	108
9.6	Ciclos de trabajo	108
9.7	Niveles de volumen	110
9.8	Niveles de volumen	111

Índice de Códigos

7.1	Código base de una ROM	27
7.2	Cabecera del cartucho	28
7.3	Comprobación de la ROM	29
7.4	Crear una ROM con RGBDS	31
7.5	Makefile básico	31
7.6	Inclusión de un Fichero Binario	36
8.1	Copia de Memoria	40
8.2	Inserción de un Sprite a la OAM	40
8.3	Lectura del PAD	41
8.4	Encontrar dirección de memoria del tile de personaje	45
8.5	Comprobación de Colisiones del Enemigo	47
8.6	Paleta para la GB	49
8.7	Paleta para la GBC	50
8.8	Espera al rango V-Blank: Método 1	51
8.9	Activar Interrupción V-Blank	51
8.10	Espera al Rango V-Blank: Método 2	51
8.11	Reserva de Memoria para DMA	52
8.12	Inicio del proceso DMA	53
8.13	Copiado de Proceso DMA	53
8.14	Rutina de Copiado en Interrupción para DMA	54
8.15	Encendido de la Segunda Pantalla	55
8.16	Inicialización Registros de Sonido de Uso General	56
8.17	Sonido en Canal 2	57
8.18	Reserva Memoria	59
8.19	Get Enemies Array	61
8.20	Rangos de Vida	63
8.21	Atributos de Sprites	64
8.22	Atributos de Sprites	65
8.23	Comprobar Tiles y Denegar Movimiento	66
8.24	Mapas de Juego	67
8.25	Bucle de Juego	68
8.26	Definición de Espacio para la Reserva de Casillas	69
8.27	Reserva de Casillas	69
8.28	Carga de Niveles	71
8.29	Paleta de Color en GB	72
8.30	Paleta de Color Invertida en GB	73
8.31	Conseguir Componente G de la Paleta de Color en GBC	74
8.32	Vector de Objetos	75

8.33 Nuevo Tamaño del Jugador	75
8.34 Actualizaciones a Final de Movimiento	75
8.35 Actualización del Hambre	76
8.36 Dibujado de Ventana	77
8.37 Textos en ROM	78
8.38 Delay	78
8.39 Cambio de GLOBAL por EXPORT	81
8.40 Comprobación de Objeto Seleccionado	81
8.41 Muestreo	82
8.42 Múltiples Tilesets	83
8.43 Múltiples Tilesets	83
8.44 Método de Sonido en Canal 1	87
8.45 Método de Música en Canal 1	88
8.46 Música y SFX en ROMX	89
8.47 Escondido de Sprites Según su Tile	91
8.48 Escondido de Sprites Según su Tile	92
8.49 Rango de Visión en los Enemigos	93
8.50 Velocidad de los Enemigos	94

1 Resumen

Golden Sacra es un juego de aventuras, RPG o nethack, para la consola Game Boy, el cual recibe influencia de títulos como **Pokémon Mundo Misterioso** o **Shiren the Wanderer**.

El juego funciona por **turnos**: habrá un turno para la acción del jugador y otro para cada enemigo. Cada vez que ataquemos, nos movamos o usemos un objeto, nuestro turno se consumirá. El **gameplay** se desarrolla principalmente en el **interior de una mazmorra**, compuesta por **distintos pisos** a las cuales podremos acceder gracias a unas escaleras.

Estará realizado completamente en lenguaje ensamblador o Z80, y en este documento se recogerá todo el proceso de diseño y desarrollo, junto a las herramientas o frameworks utilizados para ello.

Abstract

Golden Sacra is an adventure, RPG or nethack game, for the original Game Boy, which is influenced by titles such as **Pokémon Mystery Dungeon** or **Shiren the Wanderer**.

The game works by **turns**: there will be a turn for player action and another for each enemy. Each time we attack, move or use an object, our turn will be consumed. The **gameplay** is mainly developed inside a dungeon, composed of **different floors** that we can access thanks to some stairs that can be found in each one.

It will be done entirely in assembly language or Z80, and this document will cover the entire design and development process, along with the tools or frameworks used for it.

2 Objetivos

A día de hoy es difícil encontrar desarrolladores de videojuegos que se preocupen realmente de **qué está ocurriendo en la máquina** con cada instrucción que escriben gracias a la facilidad que ofrecen los entornos modernos actuales y la cantidad de información que se puede encontrar en las redes. Un **ingeniero** debe ser capaz en todo momento de **resolver los problemas** que se le planteen por si mismo y no basarse en buscar la solución de alguien anónimo.

Las arquitecturas de las máquinas actuales son **complejas y muy potentes** para que una persona les pueda sacar todo el potencial en un corto período de tiempo. Por ello, lo ideal es empezar por una consola más antigua como punto de partida, como lo puede ser la propia **Game Boy**.

La razón de escogerla como la consola sobre la que desarrollar este proyecto ha sido **subjetiva** debido al afecto que le tengo. Perfectamente podría haber escogido cualquier otra como la *NES* o la *Master System*. Por otro lado, con la documentación de esta memoria pretendo **ser de ayuda para más personas** que se propongan en un futuro realizar un juego para dicha consola.

A grandes rasgos, los objetivos serían los siguientes:

- **Analizar la Nintendo Game Boy y entender sus capacidades y limitaciones.**
- **Analizar librerías y frameworks existentes.**
- **Aprender programación en ensamblador.**
- **Comprender el funcionamiento del hardware.**
- **Diseñar y desarrollar un videojuego completo.**

Objetivos secundarios:

- **Realizar una publicación física del videojuego.**
- **Distribuir el videojuego.**

3 Terminología

A lo largo del documento se van a utilizar varias nomenclaturas para hacer la lectura más sencilla:

- **GB:** Game Boy.
- **GBC:** Game Boy Color.
- **SGB:** Super Game Boy.
- **RGBDS:** Rednex Game Boy Development System. Ensamblador y enlazador para la Game Boy y Game Boy Color.
- **Bit:** Unidad mínima de información empleada en informática.
- **Byte:** Unidad de información equivalente a 8 bits.
- **CPU:** Central Processing Unit. Hardware que interpreta las instrucciones del programa.
- **GPU:** Graphics Processing Unit. Hardware dedicado al procesamiento de gráficos.
- **RAM:** Random Access Memory. Memoria de trabajo donde almacenamos nuestras variables.
- **ROM:** Read Only Memory. Zona de memoria donde se almacena el código del programa.
- **VRAM:** Video RAM. Zona de memoria utilizada por el controlador gráfico para representar información de manera visual por pantalla.
- **HRAM:** High RAM. Zona de memoria accesible en el proceso DMA.
- **DMA:** Direct Memory Access. Característica de ciertos sistemas informáticos que permite acceder a RAM a un subsistema, independientemente de la CPU.
- **OAM:** Object Attribute Memory. Espacio de memoria en el que se almacenan los atributos de los sprites.
- **Sprite:** Elemento visual activo en pantalla.
- **Tile:** Conjunto de pixeles de tamaño 8x8.
- **GBTD:** Game Boy Tile Designer. Programa para diseñar tiles.
- **GBMB:** Game Boy Map Builder. Programa para construir tilemaps.
- **BGB:** Emulador de GB, GBC y SGB.
- **NO\$GMB:** Equivalente al BGB, pero obsoleto.

4 Metodología y Planificación

El **tipo de metodologías** aplicadas a la hora de planificar el proyecto son las **ágiles**. Si bien este tipo de metodologías suelen aplicarse a proyectos en grupo, es importante tener calculados los tiempos que se debe emplear a cada tarea con el fin de **obtener el mejor resultado en el plazo de tiempo correspondiente**.

El proyecto está dividido en distintas etapas/iteraciones, sobre las cuales se van aprendiendo y poniendo en práctica nuevos aspectos respectivos al proyecto. Además, al finalizar cada iteración, se hace una revisión del mismo para ver en detalle que necesita mejorar o arreglar, e intentar de esta manera perder el mínimo tiempo posible. Las tareas cambian y se modifican constantemente a lo largo del desarrollo, debido a que, al partir de 0, los conocimientos aumentan progresivamente. Esto implica que lo que al principio parecía estar bien implementado al final hay que, en la mayoría de casos, rehacerlo.

El proyecto se ha dividido en las siguientes etapas/iteraciones:

- **Diseño**

Lo principal es tener en mente lo que se quiere llevar a cabo. Se hará un pequeño análisis de los distintos juegos y géneros de los que dispone el catálogo de la consola, así como la posible dificultad técnica que tengan, a partir del cual obtener una idea sólida del proyecto a realizar. Realizar un pequeño GDD también nos ayudará a que estas ideas no se modifiquen o pierdan con el paso del tiempo.

- **Aprendizaje**

Esta iteración se utilizará exclusivamente para analizar en profundidad las limitaciones de la consola, así como realizar distintas pruebas que puedan llegar a servir como base a la hora de dar comienzo con el desarrollo.

- **Desarrollo**

La etapa principal del proyecto en la que vamos a invertir más tiempo. Se irá desarrollando el producto y, al mismo tiempo, realizando la documentación pertinente.

- **Revisión y Maquetación**

Esta última etapa servirá para revisar tanto el producto como la documentación de la memoria y su maquetación. También se utilizará para arreglar posibles errores del videojuego o realizar pequeñas mejoras.

Las **herramientas principales** que se van a utilizar son **Trello** y **Toggl**. Además, como herramienta de **control de versiones**, se va a hacer uso de **Git**.

En Trello se ha creado una tabla con la que poder visualizar de manera sencilla qué tareas están pendientes de realizar, tanto del producto como de la memoria. Todas estas tareas pasarán a la fase de revisión y, una vez dado el visto bueno, terminarán en el estado de finalizado.



Figura 4.1: Tabla Trello - Iteración 0

4.1 Mínimo Producto Viable

Lo normal en todo proyecto es que ocurran imprevistos que hagan al programador perder más tiempo en una tarea o incluso paralizar por completo el proyecto. Además, esto se junta con el hecho de que aquí no hay nadie que pueda ocupar nuestro puesto mientras ese problema se soluciona. Por esta razón, es importante tener en mente un **producto mínimo viable**, con el cual obtener un producto jugable de inicio a fin en el tiempo disponible.

En nuestro caso, el producto mínimo sería **una mazmorra, enemigos** que te puedan matar, **objetos** que usar, **y las mecánicas básicas** del jugador.

Más que producto se le debería de calificar de **prototipo**, pero no deja de ser un **proyecto cerrado y terminado** con el que la gente pueda jugar.

5 Game Design Document

5.1 Características

- **Título:** Golden Sacra
- **Plataforma:** Game Boy - Game Boy Color
- **Género:** Aventuras, RPG/Nethack
- **Idioma:** Inglés
- **Público Objetivo:** Cualquier persona mayor de 3 años.

5.2 Historia

La historia transcurre en una **pequeña aldea** conocida como *Elry*. La vida es tranquila y plácida, menos para **nuestra protagonista Ashia**, cuyo **padre** padece de una **enfermedad extraña** y su cura procede únicamente de una planta llamada *Inis*. Este tipo de plantas solamente crecen en las profundidades de las mazmorras más inhóspitas y terroríficas que un humano pueda pisar.

Sin llegar a ver mejoría alguna en su padre y que ninguna persona cercana podía proporcionarles el susodicho remedio, **decidió adentrarse en las cuevas cercanas con el objetivo de encontrarlo**. Sin experiencia de combate, lo único que la acompañaba era el valor.

5.3 Mecánicas

Para avanzar en *Golden Sacra*, deberemos adentrarnos en las distintas mazmorras y **llegar hasta el piso más profundo** de todos. Para ello, el jugador deberá ir derrotando a los enemigos que le corten el paso y encontrar las escaleras que le conduzcan al siguiente piso.

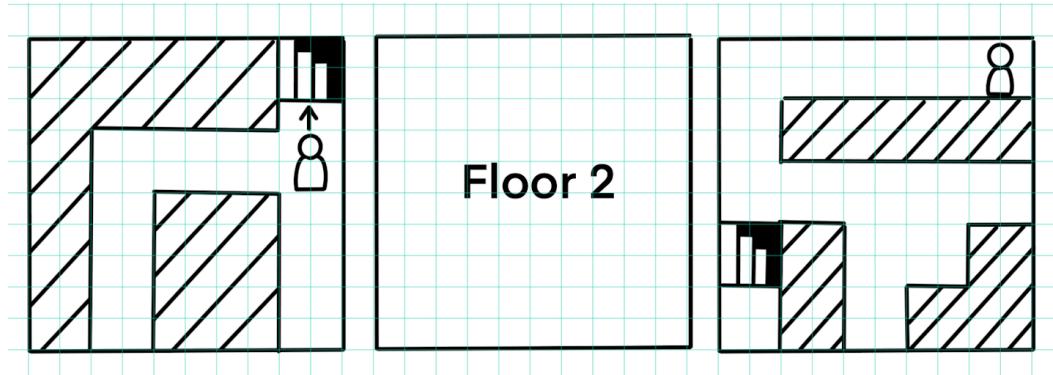


Figura 5.1: Mockup - Avance de Juego

El movimiento y el sistema de combate funcionan por **turnos**. El primer turno será para el jugador, donde podrá decidir entre las acciones de moverse, atacar o utilizar un objeto, para luego dar paso al turno de los enemigos. **No todas las acciones consumen el turno:** cambiar el equipamiento, cambiar la dirección en la que el personaje mira, etc.

El jugador dispondrá de un valor de hambre, el cual irá disminuyendo conforme gastemos nuestro turno. Si la barra llega a 0, **por cada turno la vida irá disminuyendo**. Para evitar que esto ocurra, deberá encontrar objetos que pueda comer para satisfacerse. En el HUD no se dispondrá de una barra de hambre, equivalente a la que habrá con la vida. Conforme el hambre disminuya, llegados a un punto saldrán dos mensajes de advertencia: un primer mensaje para cuando está por debajo del 20%, y otro para cuando ha llegado al 0% (teniendo en cuenta que no tener nada de hambre es un valor del 100%).

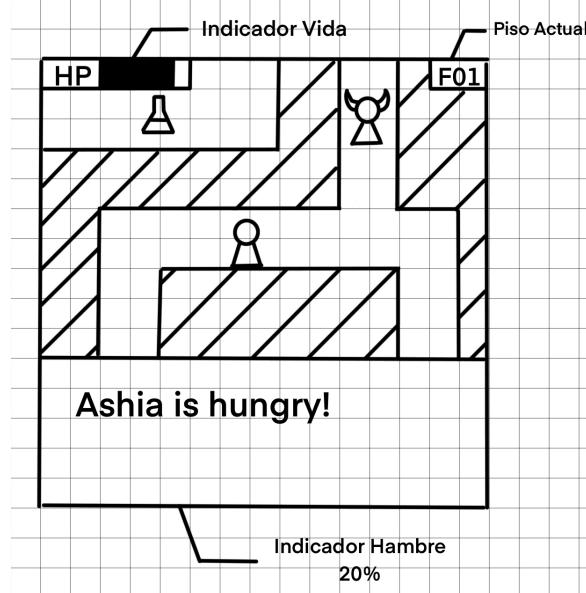


Figura 5.2: Mockup - Feedback de Hambre

Cuando se elimine a un enemigo, el jugador obtendrá **experiencia**. Con ella, podrá **subir de nivel y aumentar sus estadísticas base**. Por otro lado, gracias a unos **colecciónables**, podrá **mejorar el equipamiento y las armas que porte**.

5.4 Personajes

5.4.1 Protagonista

Nuestra protagonista, **Ashia**, será el **personaje que controle en todo momento el jugador**. Conforme avance en los niveles y consiga nuevo equipo, podrá recibir mejoras que la ayuden a enfrentarse a los enemigos.

Se realizó, tras varias opciones en formato tradicional, un **diseño completo del personaje** en formato digital. El resultado es el siguiente:



Figura 5.3: Diseño de la Protagonista

Los sprites, al tener que ser de 4x4 tiles, pierden bastantes detalles en comparación a la fuente original. Sin embargo, captan perfectamente la esencia del personaje. A la hora de hacerlo, hubo una **inspiración importante** por parte de los sprites originales de las entregas **Pokémon Rojo/Azul**.

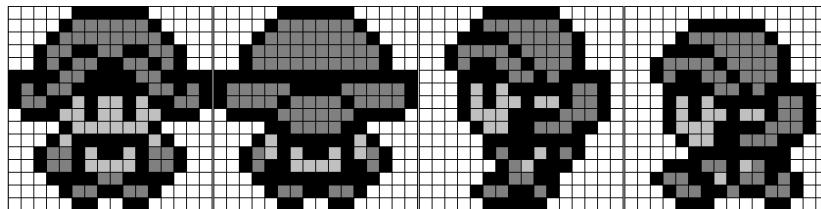


Figura 5.4: Diseño los Sprites de la Protagonista

5.4.2 Enemigos

En cuanto a los **enemigos**, los vamos a dividir esencialmente en **dos apartados**: los enemigos **básicos** que vayan apareciendo en los niveles y el **jefe final**, el cual encontraremos llegados al piso más profundo de cada mazmorra.

5.4.2.1 Básicos

Todos los enemigos básicos poseerán la **misma inteligencia artificial**. Se distinguirán esencialmente por sus **sprites y estadísticas** (vida, resistencia, fuerza, etc.).

En la mayoría de casos su **comportamiento** se basará en intentar **atrapar al jugador y debilitarlo**. Podrán obtener, sin embargo, estados en los que sientan la necesidad de huir o utilizar algún objeto.

Los enemigos van a ser los siguientes:

- **Enemigo Ladrón:** El enemigo más débil, pero de los más resistentes. Tiene 5 puntos de vida, 1 de velocidad y 5 de daño.
- **Enemigo Esqueleto:** El enemigo que más nos vamos a encontrar. Tiene 3 puntos de vida, 3/4 de velocidad y 10 de daño.
- **Enemigo Orco:** Un enemigo más feroz que el Esqueleto, con 3 puntos de vida, 2/3 de velocidad y 15 de daño.
- **Enemigo Armado:** El enemigo más poderoso de los 4, con 5 puntos de vida, 1/2 de velocidad y 15 de daño.

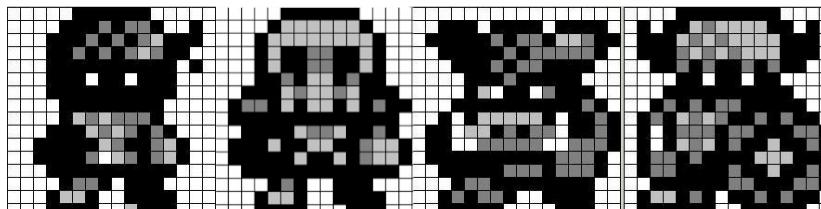


Figura 5.5: Enemigos Base

5.4.2.2 Jefe

El **jefe** se halla en el **piso final de la mazmorra**. Ocupa el doble de casillas que cualquier enemigo básico y sus estadísticas son mucho mayores que las que pueda llegar a tener cualquier enemigo básico. Para derrotarlo, será esencial el uso de objetos que vaya encontrando por el camino o, por el contrario, tener el nivel necesario como para no necesitar el uso de estos.

La sala en la que se encuentran dista mucho del resto de pisos. En este caso no encontraremos ninguna escaleras u objetos. Será un área cuadrada, en el que solamente se encontrará el protagonista frente a frente con el enemigo.

El sprite que lo forma será de 4x4 tiles, a diferencia del jugador, compuesto de 2x2 tiles. Esto además, quiere decir que el rango de ataque del enemigo es mayor, como se muestra a continuación en el mockup:

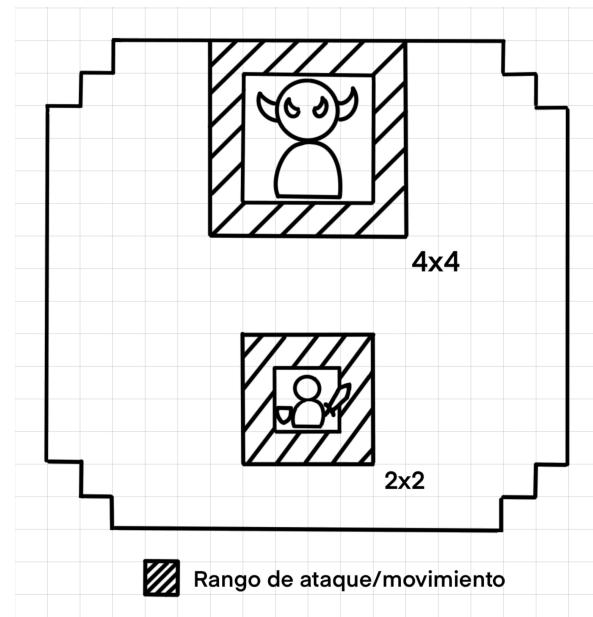


Figura 5.6: Mockup - Diseño de Enemigo Jefe

5.5 Controles

Los controles de la Game Boy son muy sencillos y fáciles de entender. Lo único que vamos a tener que hacer es diferencias entre los dos estados del juego (mapa de mundo y mazmorras), ya que dependiendo de en la que el jugador se halle, deberemos activar o desactivar mecánicas como la de atacar o el consumo de turnos.

5.5.1 Mapa de Mundo

Vamos a definir el Mapa de Mundo como todas aquellas zonas por las que el jugador pueda moverse libremente y no hayan enemigos.



Figura 5.7: Mapa de Mundo

Los controles van a ser los siguientes:

- **PAD:** Movimiento del personaje.
- **Botón A:** Interactuar con el entorno (personajes u objetos) y avanzar en los diálogos.
- **Botón Start:** Acceso al menú de juego.
- **Botón Select:** Acceso a la vista del mapa del mundo.

5.5.2 Mazmorras

Accederemos a las mazmorras a través de distintas puertas localizadas en zonas concretas del mapa de mundo.



Figura 5.8: Mazmorras

Los controles serán los mostrados a continuación:

- **PAD:** Movimiento del personaje.
- **Botón A:** Avanzar en los diálogos.
- **Botón B:** Atacar.
- **Botón Start:** Acceso al menú de juego.
- **Botón Select:** Abrir mapa de la mazmorra.

5.5.3 Menús

Para el sistema de menús, los principales botones serán el A (aceptar) y el B (denegar). También se podrá salir inmediatamente (salvo algún caso específico) utilizando el botón Start.

5.6 Pantallas

A continuación se van a mostrar una serie de *mockups* de los diferentes estados de juego en los que se puede encontrar el jugador.

La pantalla de título es el estado principal con el que comienza el videojuego cada vez que se encienda la consola. Se indicará la tecla que se debe pulsar para poder dar comienzo y simplemente se mostrará un fondo con algún dibujo, el título por encima de este y el nombre del autor junto a la fecha de creación.

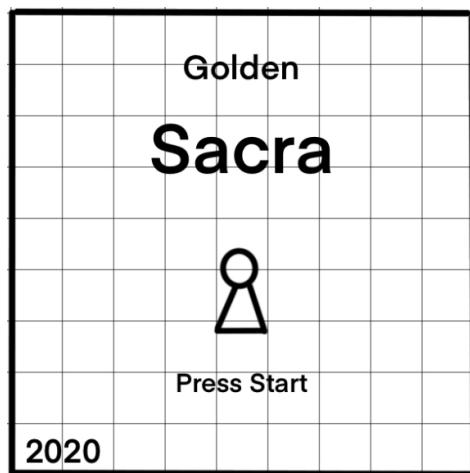


Figura 5.9: Mockup - Pantalla de Título

El segundo estado, inmediatamente después de la pantalla de título, es la del mapa del mundo. Aquí el jugador va a poder explorar un poco la zona, hablar con distintos NPC's y decidir a qué localizaciones acceder (viviendas o mazmorras).

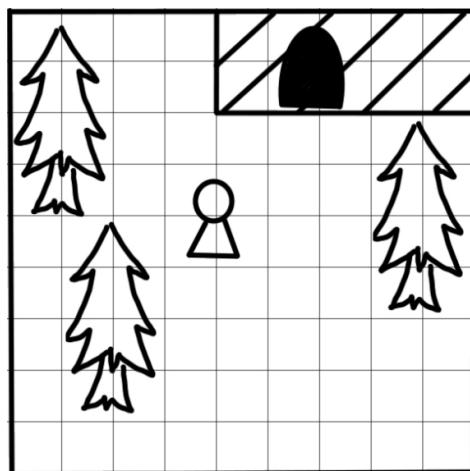


Figura 5.10: Mockup - Diseño de Mapa de Mundo

Las viviendas se compondrán principalmente de uno o varios NPC's con los que poder interaccionar y diversos muebles que decorarán el interior. En alguna ocasión, el jugador encontrará objetos escondidos en estos sitios que le serán de ayuda en su aventura. También existirá la posibilidad de comprar objetos a un mercader el cual se podrá encontrar en una de estas zonas.

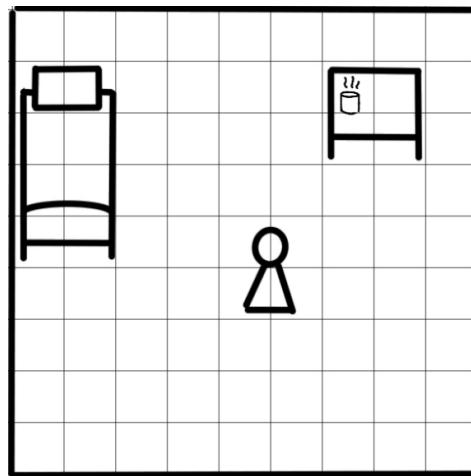


Figura 5.11: Mockup - Diseño de Vivienda

Otra pantalla será la del menú y la del inventario. A esta segunda podremos acceder a través de la primera, además de guardar la partida. El inventario mostrará todos los objetos y las cantidades de las que se disponen en ese momento.

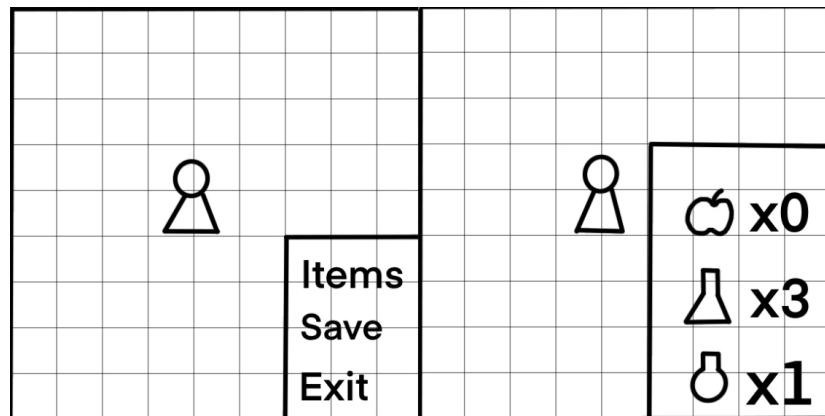


Figura 5.12: Mockup - Diseño de Menú e Inventario

5.7 Estados de Juego

Vamos a ver cómo se relacionan entre sí todos los estados del juego mediante un diagrama de flujo. Esto es útil para que sepamos, a la hora de programar, desde qué estados se puede o debe acceder a otro.

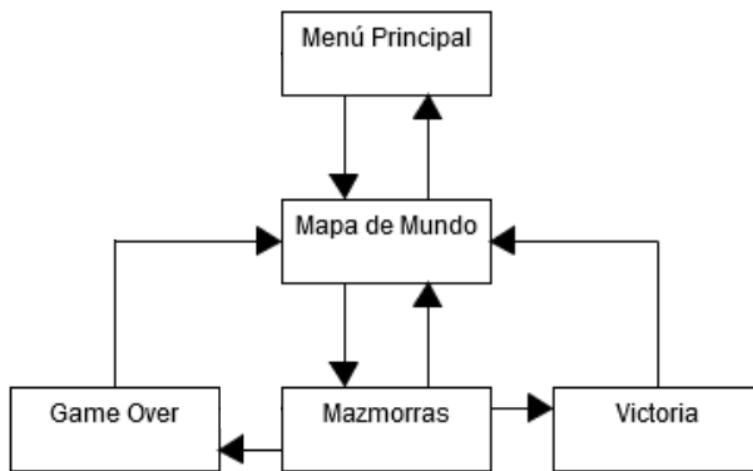


Figura 5.13: Diagrama de Flujo

Como ya se ha comentado, el primer estado es el de pantalla de título. Desde aquí solamente podremos acceder al estado de mapa de mundo, en el cual el jugador podrá decidir explorar o adentrarse en las mazmorras. Una vez aquí dentro, podremos volver al mapa del menú siempre que muramos o lleguemos al último piso de todos. Se podrá salir también desde la propia mazmorra guardando la partida, lo que provocará que se teletransporte el jugador a este estado.

6 Estado del Arte

6.1 Game Boy

La **Game Boy** es una video consola portátil de **8 bits** desarrollada y manufacturada por Nintendo. Es la segunda consola portátil de la compañía siguiendo a su familia antecesora *Game & Watch*, saliendo al mercado el **21 de Abril de 1989 en Japón**, 3 meses más tarde en América y el 28 de Septiembre de 1990 en Europa.



Figura 6.1: Game Boy

Se caracteriza por constar de un **procesador Z80**, una pantalla **LCD** monocroma con contraste ajustable, un **pad de 8 direcciones**, dos **botones de acción** (A y B), y dos **botones de control** (Start y Select).

Fue **duramente criticada** por diversas compañías debido al **tamaño de su pantalla y sus dos únicos colores**. Pese a ello, el hecho de que solamente se necesitasen **cuatro pilas AA** y pudieses **jugar durante días sin tener que cambiarlas** fue lo que propició su rotundo **éxito** frente a consolas como la *Game Gear* de SEGA. A ello se sumó que la consola saliese a la venta en pack junto al aclamado título **Tetris**. **Tanto niños como adultos** quisieron hacerse con su propia Game Boy.

Por último, mencionar que si la Game Boy tuvo **tantos años de vida** fue gracias al **modelo de negocio** que aún mantiene hoy en día Nintendo, sacando **revisões** de esta misma, **retrocompatibles** las unas con las otras, en vez de consolas completamente nuevas. Dichas consolas fueron las siguientes: **Game Boy Pocket** en 1996, y tanto **Game Boy Light** como **Game Boy Color** en 1998.

6.1.1 Especificaciones Técnicas

Cuando uno piensa en programar para cierta máquina lo primero que debe hacer es conocer exactamente **cómo es por dentro y cómo funciona**. En la siguiente tabla quedan reflejadas todas las **características de la Game Boy original**:

Game Boy	
CPU	Sharp LR35902, 8-bit
RAM	8 Kb
VRAM	8 Kb
Velocidad de Reloj	4,19 MHz
OAM	160 bytes
Pantalla	LCD, 2,6"
Resolución	160 x 144 pixeles
Alimentación	4 pilas AA
Sonido	4 canales estéreo
Paleta de colores	2-bit (4 tonos de gris)

Tabla 6.1: Especificaciones técnicas de la Game Boy original

Estas son las **características que más nos interesan** a la hora de programar en la consola. Puede que ahora mismo te parezcan **datos sin ningún valor**, pero más tarde nos servirán de guía para conocer de **cuánta memoria disponemos en cada momento y qué podemos o no hacer**.

6.2 Análisis del Mercado

Si pudiésemos viajar atrás en el tiempo (unos 20 años aproximadamente), comprobaríamos cómo la GB fue una de las consolas con **mayor auge en ventas de la historia**.

Lamentablemente, en la actualidad, la consola ha pasado a mejor vida. En donde antes se necesitaba de un cartucho del tamaño de la palma de mi mano, ahora es posible **almacenarlos en masa** en una misma micro SD. Las personas a las que les gustan estos juegos no se van a pensar dos veces el **descargar un emulador** para su teléfono móvil, donde pueden jugarlos sin ningún tapujo.

Sin embargo, existe el **mercado al por menor**, donde algunas personas siguen buscando juegos originales por puro **colecciónismo**, o las *scenes*, donde la gente saca diferentes demos o prototipos con los que probar los límites de la consola o simplemente pasar un buen rato.

6.3 Principales Referentes

En esta sección se encuentran todas aquellas entregas que, tras una revisión de los mismos, se cree que **han sido de utilidad** por su parecido a la hora de desarrollar este proyecto:

6.3.1 Pokémon Rojo/Azul - Game Boy

Juegos RPG conocidos en Japón como *Pocket Monsters: Aka & Midori*, desarrollados por la compañía **Game Freak** y publicados por **Nintendo**.

Son las **primeras entregas** de la conocida saga, siendo lanzados al mercado entre los años 1996 y 1999.



Figura 6.2: Pokemon Azul/Rojo

En el juego, el personaje es manejado desde una **perspectiva aérea**. El objetivo principal es llegar a conseguir/coleccionar todos los monstruos (conocidos como *pokémons*) que aparecen en las distintas áreas de la región ficticia de *Kanto*. Con ellos podrás completar una enciclopedia interna del juego llamada *Pokédex*, la cual dispone información de las 151 especies. Todas las versiones se componen del mismo argumento y son independientes la una

de la otra, pero es necesario hacer intercambios (mediante el *Game Link Cable*¹) con otros amigos para poder completarla.

Por otro lado, el segundo objetivo principal es que el jugador se convierta en el **entrenador pokémon** más fuerte de la región, teniendo que derrotar a todos los NPC's² (o al menos a la mayor parte) que se lleguen a interponer por su camino.

Todo consta de **tres pantallas de juego** diferentes: el **mapa general**, donde el jugador maneja al protagonista desde la perspectiva aérea ya mencionada, una **pantalla de batalla** con vista lateral, donde solamente se ve nuestro *pokémon* y el del rival, y la **interfaz gráfica/sistema de menús** donde se podrán configurar los *pokémons* que tengamos en el equipo.



Figura 6.3: Pantallas de Pokémon Rojo/Azul

La **razón de escoger este título** como referente se debe a la manera en la que implementa distintos aspectos, ya sea al **renderizado de la interfaz gráfica**, el **muestreo de diálogos**, la recolección de objetos esparcidos por el mapa, o incluso el método de carga de mapas de fondo.

La manera en que implementa las colisiones del jugador con el mapa de fondo se van intentar replicar respecto a la forma en que este lo hace. Básicamente disponemos de distintas casillas (el jugador siempre tiene guardado su casilla actual), las cuales poseen un valor que nos indicará si se trata de una obstáculo o no.

También tiene implementado en su código (siendo una de las pocas entregas que lo tienen) el cambio de paletas de color. Este es un aspecto muy interesante ya que le hacía cobrar vida en la Game Boy Color. Lo veremos en el capítulo de **Desarrollo** del documento.

Además, de forma subjetiva, el **estilo artístico** llevado a cabo es uno de mis favoritos. Esto hará que, de forma inconsciente, la mayor parte de mis sprites estén inspirados en los suyos y pueda apreciarse a simple vista.

¹También conocido como **Cable Link**, es un accesorio que permite conectar más de una Game Boy entre sí para jugar en modo multijugador.

²NPC o *Non Playable Character*, hace referencia a todos aquellos personajes que el jugador no puede controlar.

6.3.1.1 Dibujado de Ventanas Emergentes en Pokémon Rojo/Azul

Un análisis que se ha hecho de estas entregas es del **cómo eran capaces de dibujar múltiples ventanas emergentes**, sin necesidad de apagar la pantalla LCD.



Figura 6.4: Dibujado de Ventanas en Pokémon Rojo/Azul

Como podéis observar en las imágenes, por un lado tenemos la **pantalla principal** donde el tilemap está cargado y en la segunda, lo que nos encontramos curiosamente, es un **copiado y pegado** de todos los tiles de fondo que el scroll es capaz de mostrar en la primera. Con ello consiguen tener un **escena completamente idéntica**, lo que implica que, al superponer la segunda pantalla a la primera, no se va a captar diferencia alguna. La **ventaja** que tienen es la de **ser capaces de mostrar todas las ventanas emergentes que necesiten**, como se muestra en la siguiente imagen:



Figura 6.5: Ventanas Emergentes en Pokémon Rojo/Azul

Para conseguir crear este efecto sin necesidad de apagar la pantalla realizan lo siguiente: por cada N tiles a dibujar, **esperan previamente a un intervalo V-Blank o H-Blank** para que la memoria de vídeo no se corrompa. Además, al haber dibujado solamente en la pantalla secundaria, para volver al juego solo tienen que esconderla, sin necesidad de más.

6.3.2 PokéMón Mundo Misterioso: Equipo de Rescate Rojo - Game Boy Advance

Entrega de la saga PokéMón lanzada en el año 2006 para Europa. Existe una **versión paralela** lanzada para la consola **Nintendo DS**.



Figura 6.6: Pokemon Mundo Misterioso: Equipo de Rescate Rojo

Es un **RPG de aventuras** donde, a diferencia de otras entregas de la saga, innova con la premisa de que el jugador ahora es un *pokémon*. El objetivo consta en rescatar al mundo de las desgracias naturales formando un equipo de rescate junto a otros *pokémons*.

El concepto "mazmorra" ya había sido utilizada previamente en otras entregas como *Shiren the Wanderer*, pero nunca en la saga *Pokemon*. Todos los elementos "humanos" de otras entregas se han eliminado y ha conseguido diferenciarse de la competencia por su **argumento único y especial**.

La saga *Mundo Misterioso* funciona mediante **turnos**. De esta manera, cada vez que el jugador se mueva, ataque, o realice cualquier otra acción, consumirá su turno, dando paso al turno de los *pokémons* aliados y rivales.

Las **misiones se realizarán siempre en el interior de una mazmorra**, donde rescataremos *pokémons* que se hayan perdido, recolectar distintos objetos importantes, o derrotar un jefe. Dichas mazmorras están **formadas por distintos pisos** a los cuales podremos acceder mediante unas escaleras que deberemos encontrar en cada uno de ellos. Los pisos se generan de **forma procedimental**, por lo que todo cambiará cada vez que entremos a la misma mazmorra, beneficiando así la rejugabilidad.

El juego se ha escogido como **referente** por ser exactamente el **mismo género** al cual pretendemos hacer. Las **mecánicas** van a ser **prácticamente idénticas**, con la diferencia de intentar hacerlas mucho **más simples**. Hay que tener en cuenta que este juego fue desarrollado para una consola mucho superior sobre la que vamos a trabajar.



Figura 6.7: Captura de Pantalla In-Game de Pokémon Mundo Misterioso

La mecánica de turnos, mazmorras generadas proceduralmente, ataques, objetos, etc..., van a estar inspiradas principalmente en esta entrega, dejando de lado temas más técnicos como los mencionados en el anterior referente.

6.3.3 The Legend Of Zelda: Oracle of Ages - Game Boy Color

Junto a *The Legend Of Zelda: Oracle Of Seasons*, fueron dos entregas de acción/aventura desarrolladas por la compañía *Flagship* y publicadas por *Nintendo* para la consola Game Boy Color en el año 2001.



Figura 6.8: The Legend Of Zelda: Oracle Of Ages

Como la mayoría de los *Zelda* clásicos, el combate y la exploración se realizan desde una perspectiva aérea (similar a la de *Pokémon Rojo/Azul*). El arma principal es la espada, con la que podremos golpear los enemigos en un radio determinado de distancia. También obtendremos

dremos a lo largo del transcurso del juego **distintos objetos que aumentarán el número de mecánicas disponibles** y ayudarán a resolver puzzles.

En esta entrega en particular, el protagonista deberá viajar entre el pasado y el presente, conectados por agujeros temporales, para desbloquear nuevas áreas del mapa. Esto lo podremos hacer nada más conseguir el *Harpa del Tiempo* al inicio del juego.



Figura 6.9: Pasado y Presente en Oracle Of Ages

También encontraremos **colecciónables** que nos serán de utilidad a la hora de **mejorar el equipamiento**. En este caso, principalmente, serán anillos, que proporcionarán habilidades extras (como mejora de defensa o ataque). El poder de la espada y el escudo que el protagonista porta se podrán mejorar de forma similar hasta un máximo de dos veces.



Figura 6.10: Capturas de Pantalla In-Game del Oracle Of Ages

La entrega fue un **éxito tanto en lo que ventas se refiere como en crítica**. En la revista *Nintendo Power*, fue posicionado en el número 39 de los 200 mejores juegos de la consola.

La razón de escoger esta entrega como referente es el **diseño artístico de sus enemigos**, cargado de mapas de fondo y movimiento del scroll (distintos a como se implementó en *Pokémon*). El hecho de que puedas obtener colecciónables para futuras mejoras y el poder obtener distintas armas también es un factor que voy a tener en cuenta para mejorar la experiencia de juego.

7 Manual

La **información** que podemos encontrar por internet acerca de cómo programar en Game Boy puede llegar a ser **escasa e incluso confusa**. Si bien hay varios tutoriales que ayudan a iniciarse, ninguno llega a explicar con profundidad todos los aspectos. En este apartado veremos lo mejor posible desde **cómo crear un ejecutable vacío**, a qué **herramientas** nos pueden ser **de utilidad** a la hora de empezar un proyecto de estas características:

7.1 Creación de una ROM

Nuestro primer paso será **crear un ROM vacío** que pueda ser ejecutado en la Game Boy. Esto es muy sencillo de entender y lo básico a la hora de crear un juego.

Código 7.1: Código base de una ROM

```
1 INCLUDE "hardware.inc"
2
3 SECTION "Cartridge Header",ROM0[$100]
4     nop
5     jp Start
6     INIT_HEADER
7
8 SECTION "Start",ROM0[$0150]
9 Start::
10    halt
11    jr Start
```

Repasemos paso a paso el código. La primera línea **incluye el fichero de hardware**, el cual contiene todas las direcciones de memoria importantes definidas (dándole un nombre equivalente). No es necesario hacerlo, pero es de gran ayuda a la hora de programar.

En la línea 3 se especifica al programa que, el código que viene justo a continuación, lo debe guardar a partir de la dirección de memoria número \$0100. Como se puede apreciar en la tabla 9.3, es aquí donde se da la **entrada a la ejecución del programa**. Esto se resume en hacer una pausa y saltar al inicio de nuestro código. En ensamblador esto son **4 bytes (00 C3 00 01)**, con lo que **la siguiente instrucción se va a guardar** en la posición **\$0104**, es decir, la **cabecera del cartucho**.

En la línea 6 tenemos una llamada a una macro, cuya función es definir bytes y **rellenar las distintas posiciones de memoria que componen la cabecera**:

Código 7.2: Cabecera del cartucho

```

1 INIT_HEADER: MACRO
2
3     NINTENDO_LOGO
4
5     DB "GOLDEN SACRA",0,0,0
6     DB $00 ; $00 - DMG
7             ; $80 - DMG/GBC
8             ; $C0 - GBC Only cartridge
9     DB $00
10    DB $00
11    DB $00 ; $00 - GameBoy Only, $03 - SGB
12    DB CART_ROM_MBC1
13    DB $01 ; ROM size, $01 - 512Kbit = 64Kbyte = 4 banks
14    DB $00 ; RAM size
15    DB $01 ; $01 - All others
16            ; $00 - Japan
17    DB $33 ; $33 - Check $0144/$0145 for Licensee code.
18    DB $00
19    DB $00
20    DW $00
21
22 ENDM

```

Su función consta en llamar, en la línea 4, a otra macro (ya definida dentro del archivo de hardware que se ha incluido previamente) que define **48 bytes**, los cuales forman el **bitmap del logo de Nintendo** que aparece nada más encender la Game Boy. Si se altera cualquier byte de los que lo componen, nuestro juego no funcionará en una consola física, ya que el proceso de inicio comprueba que todos sean correctos. Si no lo son, se bloquea.

A continuación tenemos 15 bytes (**\$0134-\$0143**) para definir el **título de nuestro videojuego**. Si no llegamos a utilizar todos los caracteres, los espacios sobrantes se rellenan con ceros de forma automática. También lo podemos hacer manualmente, como se muestra en el ejemplo.

El siguiente byte (**\$0143**) indica si el programa **soporta funciones específicas de la Game Boy Color**. Con el valor \$80 se puede indicar que sí, pero así además funcionaría en todos los modelos.

Los dos siguientes bytes (**\$0144-\$0145**) sirven para indicar la **compañía o la productora del juego**. No es más que un ejemplo así que se va a quedar con valores nulos.

El byte **\$0146** se utiliza para **activar o desactivar las funciones SGB**. Estas funciones tienen como objetivo el poder **conectar una Game Boy a una SNES**. Tiene dos valores posibles, \$00 para dejarlas desactivadas o, por el contrario, \$03 para activarlas.

Ahora en el byte **\$0147** se debe especificar el **tipo de cartucho**. Si se dejase a valor nulo, diríamos que solo nos interesa aprovechar los 32Kb como ROM. El cartucho **MBC1**¹ seleccionado nos va a permitir un **máximo de 2Mb como ROM y otros 32Kb como RAM**. Existen **29 tipos** distintos, y queda a elección del programador elegir cuál utilizar.

Los dos próximos bytes, **\$0148 y \$0149**, sirven para definir el **tamaño de la ROM y la RAM**, consecutivamente. En el ejemplo, la ROM ha quedado dividida en 4 bancos (valor \$01). Esto puede ser de gran ayuda a la hora de almacenar música o gráficos. La RAM, por otro lado, se queda en valor nulo ya que se le ha dado un tamaño previo en el tipo de cartucho.

El byte **\$014A** se utiliza para especificar **dónde se debe vender el juego**. Las opciones son dos: Japón o cualquier otro país.

El siguiente byte (**\$014B**) quedó en desuso al poco de sacar nuevos chips. Nuevamente, al igual que los bytes **\$0144-\$0145**, especificaba la **compañía o productora**. Sin embargo, **el valor \$33 significaba que esto ya se había especificado en los bytes nombrados**, por lo que se usaban sus valores en su lugar. Las funciones SGB (si las activamos) no funcionarán si el valor de este byte es distinto de \$33.

Los **últimos cuatro bytes (\$014D - \$014F)** contienen el **proceso que comprueba toda la ROM**. El código que ejecuta el primer byte es el siguiente (en C++):

Código 7.3: Comprobación de la ROM

```
1  x=0;
2  for(int i=0134h; i < 014Ch; i++){
3      x = x-[i]-1;
4 }
```

¹MBC son las siglas de **Memory Bank Controller**. Son chips alojados en la cabecera del cartucho que se utilizan para expandir la memoria mediante la técnica del *banking*.

Volviendo al **código 7.1**, lo único que resta es poner un **bucle infinito** para mantener la Game Boy en modo reposo. Si se ejecuta el fichero *.gb* en un emulador o una consola física, el **resultado** será el siguiente:



Figura 7.1: ROM básica ejecutada en emulador

7.2 Herramientas

A la hora de crear un videojuego se necesitan muchos recursos como **arte** o **música**. También se necesita de la ayuda de **librerías que ensamblen y enlacen los distintos ficheros de los que el proyecto se compone**. En esta sección se muestran **herramientas** que pueden ser de ayuda y cuyo uso es recomendado:

7.2.1 RGBDS

RGBDS (Rednex Game Boy Development System) es una librería gratuita cuya función es **ensamblar y enlazar los ficheros que componen un proyecto**.

Viene con **cuatro herramientas**:

- **RGBASM:** Ensamblador.
- **RGBLINK:** Enlazador que creará nuestra ROM.
- **RGBGFX:** Conversor de imágenes .PNG a gráficos de Game Boy.
- **RGBFIX:** Modificador de la cabecera del cartucho para que cumpla con los requisitos.

Para ensamblar y enlazar con la librería se debe ejecutar los siguientes tres comandos en una terminal:

Código 7.4: Crear una ROM con RGBDS

```

1 rgbsasm -ogame.obj game.z80
2 rgblink -mgame.map -ngame.sym -ogame.gb game.obj
3 rgbfpx -p0 -v game.gb

```

Pero existe **un problema**: imagina que el proyecto dispone de 10 ficheros distintos. Ejecutar los **3 comandos por cada uno** de ellos **no es viable**. Una solución es crear un fichero *Makefile* para ensamblarlos todos de forma continua y enlazarlos. Un **ejemplo sencillo** podría ser el que se muestra a continuación:

Código 7.5: Makefile básico

```

1  ASM = rgbsasm
2  LINK = rgblink
3  FIX = rgbfpx
4  Mkdir_P := mkdir -p
5  Obj_DIR := ../obj/
6  GB_DIR := ../
7  ROM_NAME = Game_Title
8  SOURCES = $(wildcard *.asm)
9  FIX_FLAGS = -v -p 0
10
11 OBJECTS = $(SOURCES:.asm=$(OBJ_DIR)%.o)
12
13 all: dirs $(ROM_NAME)
14     $(warning Compiled Project)
15
16 $(ROM_NAME): $(OBJECTS)
17     $(LINK) -o $(GB_DIR)$@.gb -m $(OBJ_DIR)$@.map -n
18         $(OBJ_DIR)$@.sym $(OBJECTS)
19     $(FIX) $(FIX_FLAGS) $(GB_DIR)$@.gb
20
21 $(OBJ_DIR)%.o: %.asm
22     $(ASM) -o $@ $<
23
24 clean:
25     rm -r $(OBJ_DIR)
26     $(warning Cleaned Project)
27
28 dirs:
29     $(Mkdir_P) $(OBJ_DIR)
30     $(warning Obj Folder Make)

```

7.2.2 GBTD y GBMB

GBTD (Game Boy Tile Designer) y **GBMB (Game Boy Map Builder)** son unos programas gratuitos con los cuales podremos crear nuestros tiles y tilemaps, ambos **desarrollados por Harry Mulder**. Además, son programas que exportan directamente al lenguaje ensamblador compatible con RGBDS.

El primer programa, **GBTD**, tiene como utilidad **la creación de tiles** de diferentes tamaños (8x8, 8x16, 16x16...), tanto con los colores de la Game Boy original como los de la SGB y Game Boy Color.

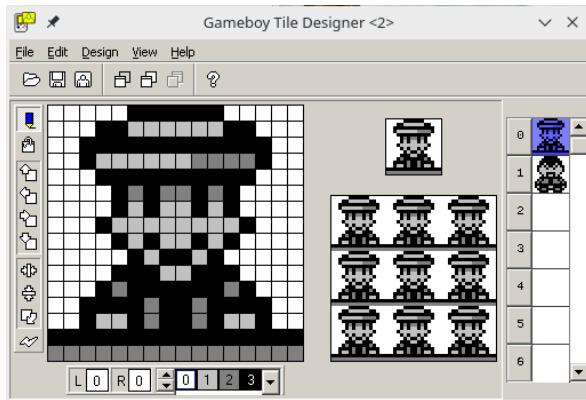


Figura 7.2: Game Boy Tile Designer

Por otro lado, con **GBMB** podremos utilizar los ficheros generados por el GBTD y crear distintos tilemaps, con un tamaño máximo de 1024x1024 píxeles. Además, se puede importar cualquier tile que GBTD soporte.

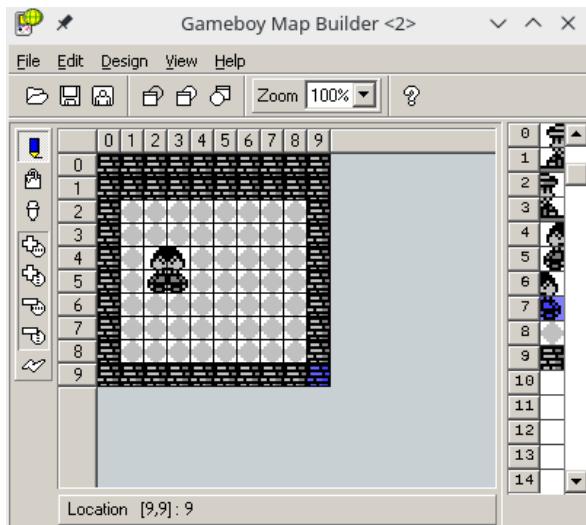


Figura 7.3: Game Boy Map Builder

7.2.3 Emuladores

Existen diversos emuladores con los que podemos probar a la perfección nuestros juegos de GB. Sin embargo, **pocos son los que traen un debugger** con los que podamos visualizar en todo momento lo que está ocurriendo con la memoria.

La mejor opción es el uso de **BGB**. Permite emular juegos de GB, SGB y GBC. Cuenta con **muchas herramientas** con las que mejorar el gameplay y, además, es **muy preciso** a la hora de emular, por lo que cualquier juego que funcione bien aquí también lo hará en una Game Boy real.

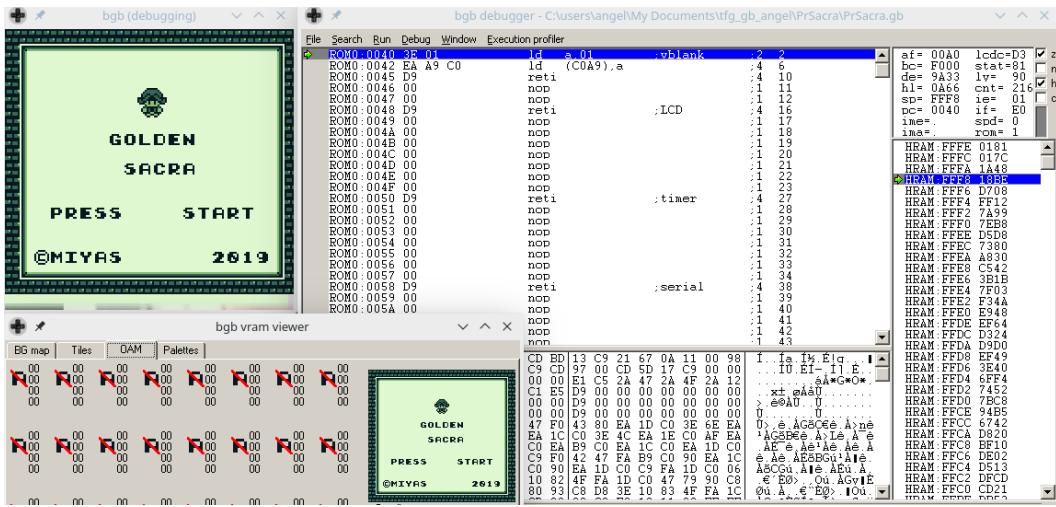


Figura 7.4: Emulador BGB

Otro emulador que podemos usar de manera complementaria es **NO\$GMB**. Su debugger es **más cómodo** de utilizar, pero existen diferencias cruciales entre el funcionamiento esperado y el resultado. Existen **problemas en la emulación del intervalo V-Blank**, por lo que al usarlo en una Game Boy física puede que se den corrupciones del bus de datos.

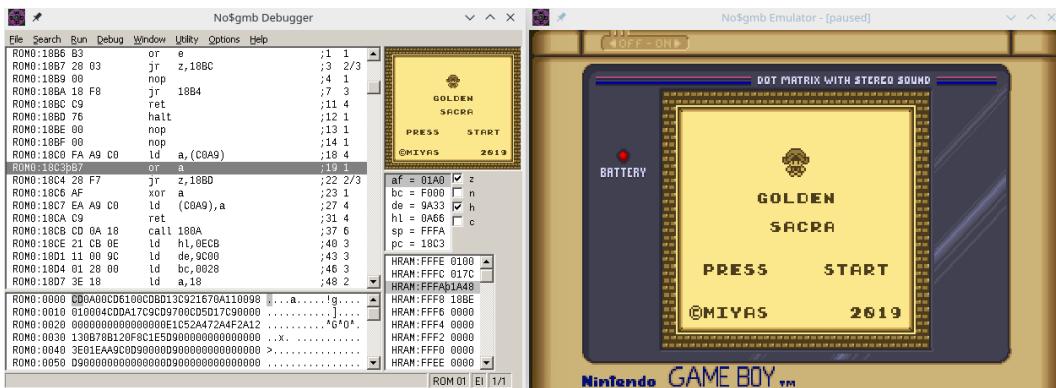


Figura 7.5: Emulador NO\$GMB

7.2.4 Flash Cartridge

La mejor opción para estar completamente seguros de que nuestro juego funciona es probarlo en una Game Boy física. Para ello tendremos que hacer uso de un **cartucho flash**, al cual podemos introducir ROM's de diversos juegos. No todo el mundo se lo puede permitir, pero puede resultar de lo más útil.



Figura 7.6: Game Boy Flash Cartridge

El *Everdrive* que se muestra en la imagen es el **modelo más barato** (hablamos alrededor de los 60€), pero funciona perfectamente. Nos permite guardar nuestras partidas y utilizarla en cualquier modelo de Game Boy. Además, **dispone de todos los mappers**, lo cual significa que, asignemos el tipo de cartucho que asignemos en la cabecera de la ROM, vamos a poder hacer uso de él.

A la hora de comprar uno, lo único **necesario es descargar**, desde la página web que se muestra en el propio cartucho, **los drivers** que el fabricante nos marca.

7.2.5 GBSound Sample Generator

GBSound Sample Generator es un programa que podemos ejecutar tanto en emulador como en hardware real, **bastante sencillo de usar**, en el que podemos modificar los valores de los registros de sonido y probar directamente su resultado.

Sound Mode #1	Sound Mode #2	Sound Mode #3	Sound Mode #4
♦Sup_Time 0	♦Pat_Duty 2	♦Sound_On/Off 1	♦Sound_Len 58
Sup_Mode 0	Sound_Len 1	Sound_Len 0	Env_Init 10
Sup_Shifts 0	Env_Init 8	Sel_Out_Level 0	Env_Mode 0
Pat_Duty 2	Env_Mode 0	Frequency 1750	Env_Nb_Step 1
Sound_Len 1	Env_Nb_Step 4	Cons_Sel 0	Poly_Cnt_Freq 0
Env_Init 4	Frequency 1751	Out_to_S01 1	Poly_Cnt_Step 0
Env_Mode 0	Cons_Sel 0	Out_to_S02 1	Poly_Cnt_Div 0
Env_Nb_Sup 3	Out_to_S01 1	On/Off 0	Cons_Sel 1
Frequency 1651	Out_to_S02 1		Out_to_S01 1
Cons_Sel 0	On/Off 0		Out_to_S02 1
Out_to_S01 1			On/Off 0
Out_to_S02 1			
On/Off 0			

Figura 7.7: GBSound Sample Generator

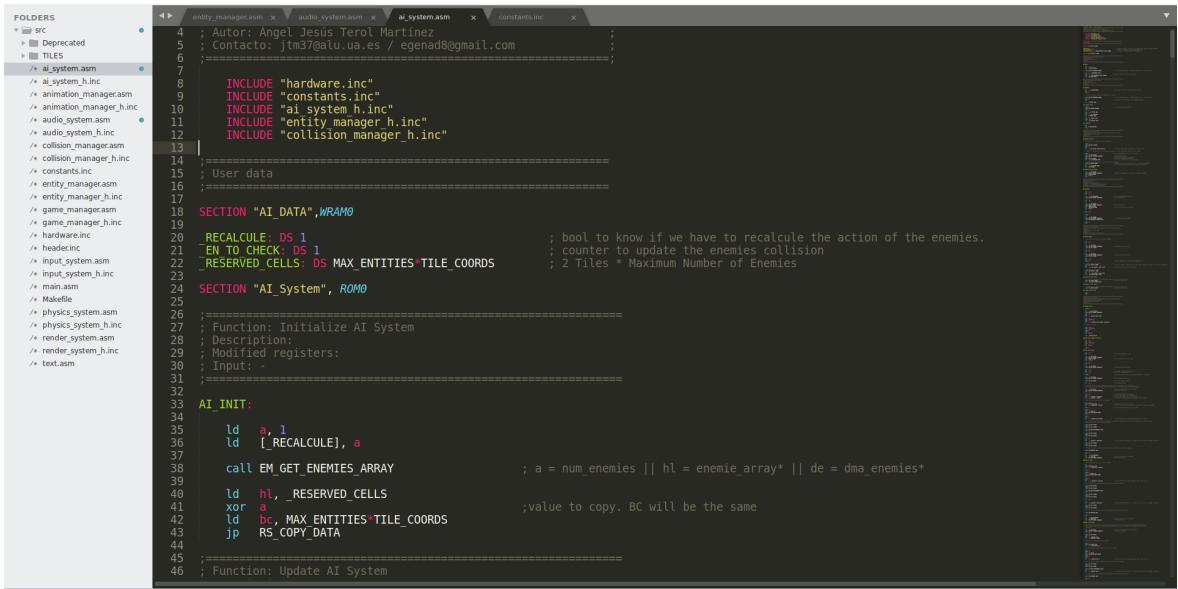
En vez de establecer el valor bruto de los registros en el código, **definiremos una frecuencia de sonido, un nivel de volumen, una envolvente, un ciclo de trabajo, etc.** Como los valores se muestran por pantalla, si queremos reproducir ese sonido exacto **solo** tenemos que copiar los 3 o 4 bytes que el programa nos da.

7.2.6 Sublime Text 3

Por último, vamos a utilizar como editor de texto **Sublime Text 3**. Es una herramienta muy **sencilla, ligera y con un abanico de comandos muy extenso**.

La **interfaz gráfica** consta de una **barra lateral izquierda**, donde se muestra el **sistema de ficheros**, comenzando desde la carpeta raíz que se haya indicado, y una **barra lateral derecha** donde se encuentra una **copia del documento, pero en forma de slider y en miniatura**. Este último es muy útil a la hora de navegar por el documento.

Además, tiene la opción de **instalar paquetes** que añadan funcionalidades al editor. Estos paquetes pueden ser descargados de otros usuarios o, por el contrario, programados. Para este proyecto se recomienda instalar el paquete de **RGBDS**, el cual se va a encargar de pintar cada palabra de un color, según la sintaxis indicada por el propio ensamblador.



```

FOLDERS
SRC
Depreciated
TILES
ai_system.asm
ai_system.h.inc
animation_manager.asm
audio_system.asm
audio_system.h.inc
entity_manager.asm
entity_manager.h.inc
collision_manager.asm
collision_manager.h.inc
constants.inc
entity_manager.asm
entity_manager.h.inc
game_manager.asm
game_manager.h.inc
hardware.inc
header.inc
input_system.asm
input_system.h.inc
main.asm
Makefile
physics_system.asm
physics_system.h.inc
render_system.asm
render_system.h.inc
text.asm

entity_manager.asm x audio_system.asm x ai_system.asm x constants.inc x

4 ; Autor: Angel Jesus Terol Martinez
5 ; Contacto: jtm3@alu.ua.es / egenad8@gmail.com
6 ;
7
8 INCLUDE "hardware.inc"
9 INCLUDE "constants.inc"
10 INCLUDE "ai_system.h.inc"
11 INCLUDE "entity_manager.h.inc"
12 INCLUDE "collision_manager.h.inc"
13 |
14 =====
15 ; User data
16 ;
17
18 SECTION "AI_DATA",NRAMO
19
20 RECALCULE: DS 1 ; bool to know if we have to recalculate the action of the enemies.
21 EN_TO_CHECK: DS 1 ; counter to update the enemies collision
22 RESERVED_CELLS: DS MAX_ENTITIES*TILE_COORDS ; 2 Tiles * Maximum Number of Enemies
23
24 SECTION "AI_System", ROM0
25
26
27 ; Function: Initialize AI System
28 ; Description:
29 ; Modified Registers:
30 ; Input:
31 ;
32
33 AI_INIT:
34
35 ld a, 1
36 ld [_RECALCULE], a
37
38 call EM_GET_ENEMIES_ARRAY ; a = num_enemies || hl = enemy_array* || de = dma_enemies*
39
40 ld hl, _RESERVED_CELLS
41 xor a
42 ld bc, MAX_ENTITIES*TILE_COORDS ; value to copy. BC will be the same
43 jp RS_COPY_DATA
44
45 ; Function: Update AI System
46

```

Figura 7.8: Interfaz Sublime Text 3

7.3 Tiles

Ya se han mencionado y visto como generarlos usando distintas herramientas pero, ¿conocemos en detalle realmente qué son o cómo funcionan?

La primera diferencia a tener en cuenta es que **un tile no es un sprite**. Se puede afirmar, sin embargo, que **los sprites están compuestos por un tile** (como mínimo).

Así pues, un tile es **conjunto de píxeles** que, de manera general, forman un tamaño de 8x8. Para utilizarlos en un proyecto, lo primero que se debe hacer es incluirlos (después de haberlos generado con la herramienta GBMB):

Código 7.6: Inclusión de un Fichero Binario

```

1 SECTION "Tiles", ROM0
2 Tiles:
3   INCLUDE "./TILES/tilemap.z80"
4 Fin_tiles:
```

La Game Boy funciona por “capas”, donde cada una se dibuja por encima de la anterior. Como se ha podido ver, existen 2 fondos de pantalla sobre los cuales dibujar tiles. La **pantalla principal** es la que sirve de **fondo** (árboles, casas, ríos... Cualquier cosa que no vaya a necesitar actualizarse constantemente). La **segunda pantalla**, la cual se puede esconder, se utiliza de forma común para el **HUD** (ventanas emergentes del inventario o guardado, por ejemplo). La **tercera capa** es la de los **sprites**. En esta última capa es donde va a estar el personaje, enemigos, etc.

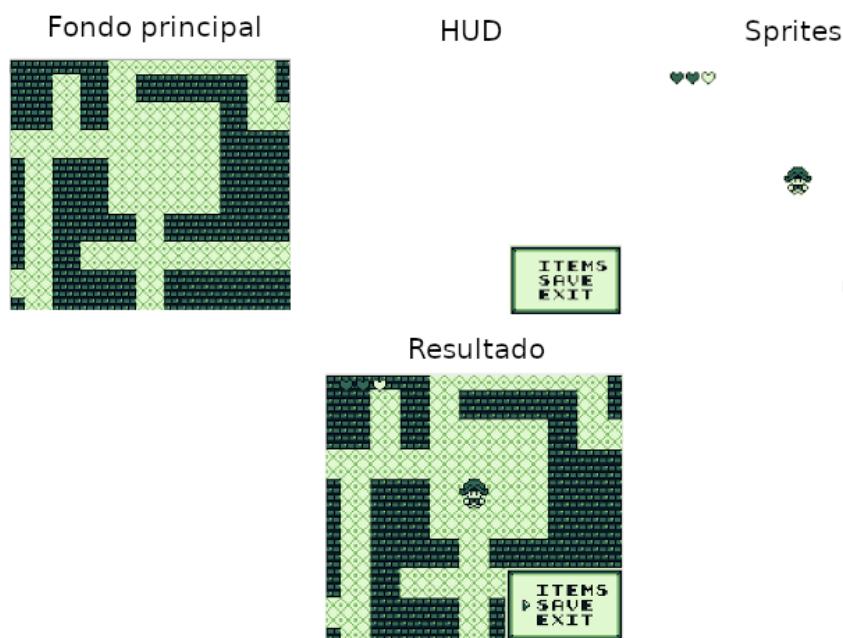


Figura 7.9: Capas de la Game Boy

7.4 Sprites

Sabemos diferenciarlos de los tiles y cómo podemos generarlos. Lo único que falta por conocer es el **cómo se estructuran en memoria**.

Los sprites se almacenan en el **Object Array Memory (OAM)**, y **ocupan siempre 4 bytes cada uno**. El primer byte es la posición Y, el segundo la posición X, el tercero el número de tile y el cuarto el atributo (que veremos más en detalle).

Lo que se llega a visualizar en memoria una vez se ha creado (como mínimo) un sprite, es un **vector de bytes**. Estos valores se deberán saber diferenciar empezando desde la primera posición de todas. A continuación se expone una **representación gráfica**:

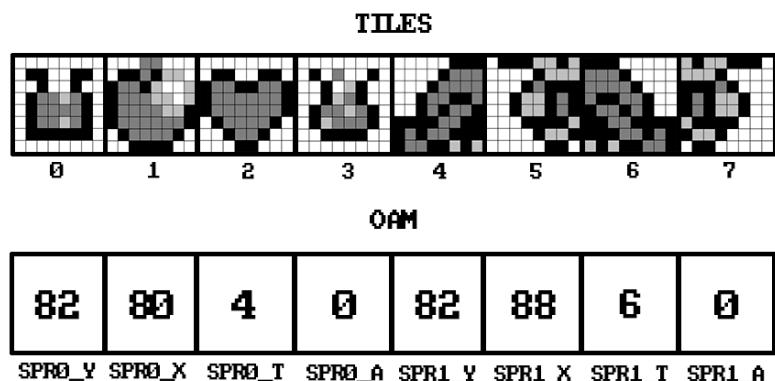


Figura 7.10: Descripción Gráfica de la OAM

Los bytes de la OAM que se representan en la imagen darían como resultado **media cabeza de un personaje**, puesto que el primero tiene indicado que muestre el tile número 4 y el segundo el número 6. Remarcar que **ambos sprites se separan por 8 píxeles en el eje X**, el cual es el ancho de un tile.

8 Desarrollo

8.1 Iteración 0 - Iniciación en la Programación para Game Boy

Esta iteración dio **comienzo** de manera previa a la propuesta de TFG, con la idea de ir preparado al inicio de desarrollo y llevar los conceptos básicos aprendidos. Esto **ha ayudado a conocer** mejor **cómo ensamblar** un trozo de código (y hacerlo funcionar en una GB real) o **cómo meter un sprite en memoria y moverlo por pantalla**.

8.1.1 Hello World

El **primer experimento** que se ha conseguido desarrollar ha sido el de mostrar por pantalla el conocido **"Hello World"** que todos hemos programado alguna vez al empezar con un lenguaje nuevo. A pesar de tener **"basura visual"**, se podían llegar a distinguir las letras.



Figura 8.1: Hello World

No tiene mucho misterio: primero de todo **cargamos en memoria los tiles correspondientes**, que en este caso no eran otros que las letras del abecedario inglés. Lo segundo que necesitamos es **introducir en memoria de vídeo (OAM) los sprites que consideremos oportunos manualmente**, uno tras otro, y dejando entre ellos una separación de 8 pixels (recordemos que cada tile son 8x8 pixels).

¿Cómo y dónde se deben cargar los tiles para posteriormente utilizarlos? Si observamos el mapa de memoria (9.2), comprobaremos que la posición **\$8000** hace referencia a la VRAM interna. Y precisamente es aquí donde **la consola busca el tile correspondiente cada vez que insertemos un sprite en la OAM**.

El método que a crear es el equivalente al *ldir* de *Amstrad*, con el que podremos **insertar filas de bytes de un origen a un destino de forma directa**.

Código 8.1: Copia de Memoria

```

1 COPY_MEM:
2   ld a, [hl]    ; cargamos el dato en A
3   ld [de], a    ; copiamos el dato al destino
4   dec bc        ; uno menos por copiar
5
6   ld a, c      ; comprobamos si bc es cero
7   or b
8   ret z        ; si es cero, volvemos
9
10  inc hl
11  inc de
12  jr RS_COPY_MEM

```

La función tiene como **parámetros** la **dirección origen** en el registro HL, la **dirección destino** en DE, y el **número de bytes** a copiar en BC. Simplemente va insertando en HL el contenido de DE, incrementa ambas direcciones, decrementa el contador para saber si aún quedan bytes restantes por copiar y, si no es así, terminar y volver.

Lo siguiente es **saber cómo crear un sprite** por cada letra que se desee mostrar:

Como ya se ha visto en el **apartado 7.4**, lo que debemos hacer es **indicar el sitio en el que lo queremos y qué número de tile mostrar** (del atributo no nos vamos a preocupar de momento, por lo que lo dejamos a nulo). Estos datos se van a guardar en cualquier lugar de la ROM (según vea conveniente el ensamblador) para, una vez ejecutado el juego, leer y copiarlos.

Código 8.2: Inserción de un Sprite a la OAM

```

1 MAKE_LETTER:
2   ld hl, letter_H
3   ld de, _VRAM
4   ld bc, $4
5   jp RS_COPY_MEM
6
7 letter_H: DB 5, 5, 7, 0

```

Con este ejemplo ya aparecerá la letra H por pantalla. Destacar que el código es muy mejorable ya que **no es reescalable** y habría que tener el mismo trozo de código para cada letra que queramos meter en RAM.

También dejar claro que el mensaje de "Hello World" **podría haberse hecho sin utili-**

zar RAM (simplemente insertándolo de background) ya que no se va a modificar en ningún momento. Se ha realizado de esta forma, de todos modos, para facilitar explicaciones futuras.

8.1.2 Sprites y Scroll

La **segunda prueba** se basó en conseguir **animar y mover un sprite por pantalla y aprender el funcionamiento del scroll**.

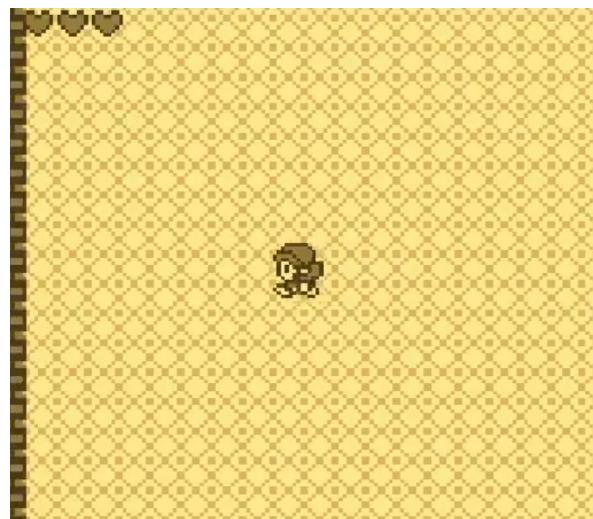


Figura 8.2: Sprite Animado en NO\$GMB

Sabemos crear sprites (lo único que debemos hacer realmente es meter datos a una variable situada en la OAM), por lo que **animarlos no es muy complicado**. Necesitaremos un "reloj" o **contador**, que **cada N ciclos nos modifique los números de tiles de cada sprite**.

Para moverlos, sin embargo, se necesita **comprobar el estado del PAD** (en el capítulo **9.4** se puede hallar más información). Una vez comprobada la dirección que el jugador nos indica, dependiendo de esta moveremos los 4 sprites que forman al personaje, **decrementando o incrementando una de sus coordenadas**.

Es **necesario leer el PAD varias veces seguidas**, ya que debido al efecto *bouncing* se pueden producir resultados que no concuerden con la realidad.

Código 8.3: Lectura del PAD

```

1 =====
2 = Valores correspondientes
3 =====
4 A_BUTTON EQU %00000001
5 B_BUTTON EQU %00000010
6 SELECT EQU %00000100
7 START EQU %00001000
8 RIGHT_JP EQU %00010000
9 LEFT_JP EQU %00100000

```

```

10 UP_JP    EQU %01000000
11 DOWN_JP   EQU %10000000
12 =====
13 =====
14 = Función de lectura
15 =====
16 READ_PAD:
17
18 ld  a, P1F_5    ;(cruzeta activada, botones no)
19 ld  [rP1], a
20 ld  a, [rP1]
21 ld  a, [rP1]
22 ld  a, [rP1]
23 ld  a, [rP1]    ;leemos varias veces
24
25 and $0F      ;solo queremos los 4 bits bajos
26 swap a       ;cambiamos los valores altos por los bajos
27 ld  b, a       ;insertamos el resultado en el registro b
28
29 ld  a, P1F_4    ;(botones activados, cruzeta no)
30 ld  [rP1], a
31 ld  a, [rP1]
32 ld  a, [rP1]
33 ld  a, [rP1]
34 ld  a, [rP1]
35
36 and $0F      ;nuevamente solo queremos los bits bajos
37 or b        ;hacemos una operación or con b para obtener los 8 bits.
38 cpl         ;complementario de a.
39 ld  [_PAD], a
40           ; pad but
41 ret        ;result = [0000] [0000]

```

Para **comprobar la dirección** solamente tenemos que coger el valor del PAD y hacer una **operación AND con la constante que queramos**. Si el resultado no da valor nulo, quiere decir que esta pulsada en ese instante.

Ahora lo que queda es saber cómo queremos mover al personaje. En este caso, simplemente decrementamos o incrementamos las coordenadas Y o X del scroll, menos en el caso de que se llegue a un extremo de la pantalla, donde lo que modificamos son los valores de los sprites. Es decir, mientras la X del scroll no sea 0, decrementaremos su coordenada X, pero si lo es, las coordenadas X que se van a decrementar son las de cada sprite que forman al personaje.

Esto es **cuestión de diseño**, por lo que cada uno puede implementarlo de la forma que guste. En juegos como **Pokémon Rojo/Azul**, lo que se hacía siempre era mover el scroll. Sin embargo, en juegos como **The Legend Of Zelda: Oracle of Ages**, únicamente se movían los sprites.

8.1.3 Tilemaps

Ahora queda conocer cómo mostrar por pantalla un tilemap:

Una vez se disponga de un tilemap exportado gracias al uso de la herramienta GBMB, lo que se debe hacer es **copiar los bytes, uno a uno, a memoria de vídeo**.

Para ello **debe estar apagado el LCD** (insertar un 0 en la dirección de memoria \$FF40), ya que **de otra forma** es posible que la consola se salga del rango V-Blank y produzca **resultados no deseados**.

A continuación, incluiremos el fichero generado en cualquier zona del código, para que una vez compilado se guarde en zona de ROM, igual que se hace con los tiles. Utilizamos por último la función de copiado de memoria, encendemos de nuevo el LCD, y listo: hemos obtenido un tilemap en pantalla.

8.1.4 Conclusión

Esta iteración me ha servido para **entrar con unos conocimientos previos al desarrollo del juego**. Si bien los temas tocados son muy básicos, son fundamentales. Hay muchas **partes** que son **totalmente mejorables** y que en un futuro deberé de modificar, pero pienso que **lo aprendido me va a beneficiar mucho de cara a las próximas etapas**.

8.2 Iteración 1 - Primer Prototipo

Llegados a la **primera iteración**, el **objetivo** era el de conseguir implementar **colisiones**, un **enemigo básico**, y alguna **ventana de HUD**. Además de las acciones de **atacar o ser atacado** y, por ende, **matar o morir**.

8.2.1 Colisiones

Las **colisiones** son un **aspecto esencial** en cualquier juego. Sin ellas, no hay límites que impidan al jugador ir por distintas zonas (las cuales puedan contener comportamientos indeterminados).

En el caso de *Amstrad* (y prácticamente cualquier sistema), las colisiones se pueden desarrollar **píxel a píxel**, es decir, comprobando las coordenadas X y Y de cada entidad junto a la anchura y altura, respectivamente, de sus sprites. Una imagen representativa de lo descrito es la siguiente:

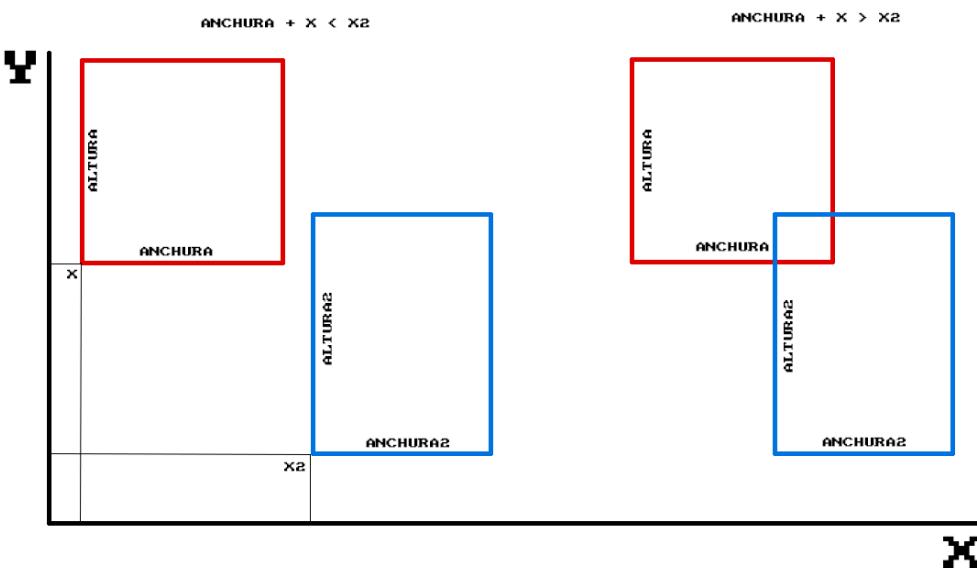


Figura 8.3: Colisiones Pixel a Pixel

De hecho, en la GB se puede comprobar las colisiones entre sprites con este método sin problema alguno. ¿Qué ocurre entonces? A pesar de que los sprites tengan coordenadas, el fondo de pantalla o background va por tiles. O dicho de otro modo, **el fondo no está compuesto por sprites**, los cuales posean coordenadas de pantalla para, posteriormente, nosotros hacer las comprobaciones necesarias.

¿Cómo lo implementamos entonces? Esta es una **pregunta que muchas personas llegado el momento se plantean constantemente** durante un cierto tiempo. "Dadas las

coordenadas de pantalla de nuestro personaje, calculemos en qué tile está situado exactamente al inicio de partida". Esta es la **primera respuesta alguien se puede dar**. El resultado de ello va a ser **desastroso**, ya que se necesitan demasiados cálculos (y no solo de 8 bits, sino de 16 bits) para poder lograrlo. Además, el código se hace infinito y muy poco óptimo.

Lo mejor es **optar por las ideas más sencillas** (y con más sentido), por ejemplo, **guardar desde un principio el tile** (precalculado) en el que se encuentra el personaje. Digamos que si su posición en coordenadas es la (80,80), siendo el valor del scroll (0,0), el tile del personaje es el (9,9). **Los cálculos son sencillos**: se dividen las coordenadas del personaje entre 8 (tamaño de un tile) y le resto 1. Esta última operación es por como he diseñado el juego y por comodidad, ya que todos los bloques son de 2x2 tiles, menos los límites de pantalla que los tengo diseñados de 1x1 tiles.

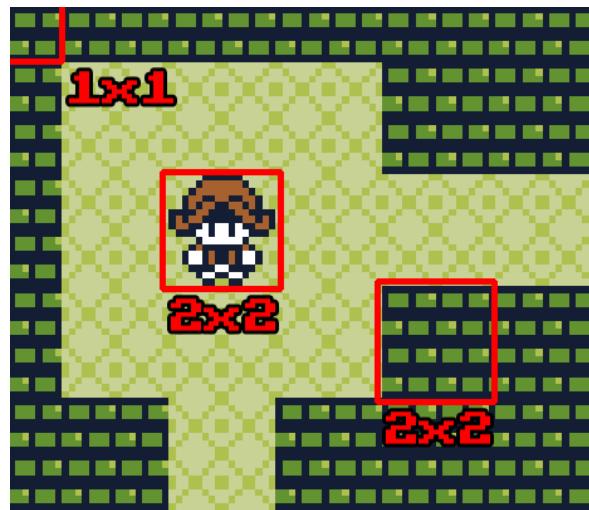


Figura 8.4: Tamaño de Bloques en el Juego

A continuación, lo que debemos hacer es un poco más complejo: comprobar el tile al que nos vamos a mover, dependiendo de la dirección que nos indique el PAD. Lo primero a encontrar es la dirección de memoria de vídeo que nos indican los tiles del personaje. Empezando desde la dirección inicial de la pantalla, se le sumará en bucle en valor 32 hasta que el tile Y del jugador se quede a 0 (obviamente, sin borrar el dato inicial). Cuando esto último ocurra, se le suma directamente el tile X, y de esta manera ya habremos conseguido la posición de memoria en la que se encuentra el personaje.

Código 8.4: Encontrar dirección de memoria del tile de personaje

```

1 CM_CHECK_TILE:
2
3     ld hl, _SCRNO      ;32x32 tiles
4     ld e, 32
5     ld d, 0
6     ld b, 0
7     ld a, [_TILE_Y]

```

```

8
9 cnt_yloop:
10
11    ;Moverse primero en el eje Y --> añadir 32
12    add hl, de
13    inc b
14    cp b
15    jr nz, cnt_yloop
16
17    ;Moverse en el eje X --> añadir 1
18    ld a, [_TITLE_X]
19    ld e, a
20    add hl, de
21    ret

```

¿Queremos comprobar el tile izquierdo? Le restamos 1 a esa dirección. ¿Queremos comprobar el tile de abajo? Le sumamos 32.

Por último se deberá de comprobar si podemos mover los sprites a ese tile. Lo que haremos será implementar un **listado de bytes** (seguidos, uno tras otro) de los **números de tile a los que el personaje se puede mover**, y comprobar si coincide con alguno de esos. En caso contrario, denegaremos el movimiento.

Como podemos observar, el proceso es algo más complejo que las dos comprobaciones que se hacen habitualmente en cualquier juego 2D, pero no por ello imposible. **Lo único que hay que tener en cuenta en todo momento es que el bus de datos no se corrompa**. Es decir, que cuando vayamos a comprobar el valor de la susodicha dirección de vídeo, el LCD se encuentre en V-Blank o H-Blank. Si necesitas algo más de información ve al apartado **9.6**.

8.2.2 Enemigos

La inteligencia artificial de los enemigos en esta iteración se puede describir como un 'pilla-pilla', sin mucho más. De forma general, en la mayor parte de los juegos *RPG* el comportamiento de los enemigos es ese, con añadidos como el sentido de visión o las diferentes estadísticas.

Lo primero que vamos a hacer es **diseñar unos sprites sencillos** del típico esqueleto que te encuentras en cualquier juego *RPG*. La única condición que vamos a tener es que el sprite debe ser de 2x2 tiles, para que luego no hayan problemas con los cálculos de colisiones.

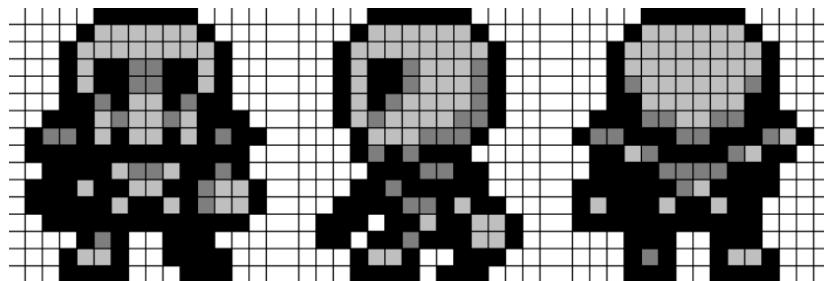


Figura 8.5: Primer Diseño de Enemigo

Ahora que tenemos un sprite lo siguiente es conseguir que se mueva por la pantalla. ”¿En qué dirección?”, es la pregunta que nos debemos hacer. Aunque la IA persiga al jugador, **si no le controlamos las colisiones va a poder moverse por donde más le convenga**. Dicho pues, el primer paso va a ser calcularle las colisiones, exactamente **igual que se ha explicado con el jugador**. La única **diferencia** es que en vez de coger el valor de los inputs, debemos **usar el valor pasado por parámetro** al cual la entidad intente moverse.

Código 8.5: Comprobación de Colisiones del Enemigo

```

1 CM_CHECK_NEXT_TILE_EN:
2
3     ld c, a
4     call CM_CHECK_TILE    ; Devuelve en HL la dirección de memoria del tile.
5
6     ld a, c
7     or a
8     jr z, cnt_top      ; Move enemy up
9     dec a
10    jr z, cnt_bot      ; Move enemy down
11    dec a
12    jr z, cnt_left     ; Move enemy left
13    jr cnt_right       ; Move enemy right

```

¿Qué input le pasamos? Es lo ”único” que nos queda por calcular. Realmente necesitaremos programar varias funciones que calculen la dirección a seguir de manera distinta. El mayor problema de la GB es el del **tiempo**, en el sentido de que si nos salimos del intervalo V-Blank no vamos a poder continuar comprobando el valor de las casillas. Y ,a diferencia del jugador, en este caso no vamos a comprobar una dirección en un mismo ciclo de cómputo: **en el peor de los casos vamos a querer comprobar las cuatro direcciones posibles**.

Así pues, vamos a tener que simplificar lo máximo posible todos los cálculos, dejándolo, por ejemplo, de la siguiente manera:

- **1:** Comparamos coordenadas Y. Si son idénticas, pasamos directamente a comprobar las coordenadas X.
- **2:** Si las coordenadas Y eran distintas, miramos si el tile del enemigo es mayor que la del jugador. Si lo es, intentamos moverlo hacia arriba, si no lo es, hacia abajo.

- **3:** Si no hemos podido moverlo en Y por la condición del punto 1 o por colisiones en el punto 2, comparamos las coordenadas X. Si hemos podido moverlo, saltamos al punto 6.
- **4:** Si la coordenada X del enemigo es mayor, intentamos moverlo hacia la izquierda. En caso contrario, hacia la derecha.
- **5:** Si se han encontrado colisiones, salimos de la función y no actualizamos el movimiento del enemigo.
- **6:** Guardamos en la variable del enemigo la dirección resultante para, en el siguiente ciclo, iniciar el movimiento.

Ya tenemos el movimiento terminado. ¿Qué nos queda? Obviamente, **conseguir que ataque al jugador y le haga daño**. Esto es muy simple: simplemente comprobamos, previo a realizar los pasos expuestos, si el jugador está en una casilla adyacente. En caso de estarlo, **iniciamos una animación de ataque y al terminar le restamos vida al jugador**.



Figura 8.6: Ataque del Enemigo

8.2.3 Paletas

Este es un tema que muchos comenzamos a estudiar por mera curiosidad. En nuestra infancia siempre nos hemos preguntado por qué algunos juegos se veían en blanco y negro en la GB, mientras que, si los ponías en la GBC, repentinamente obtenían color. Y cómo no, esto sucedía solamente en casos específicos como el Pokémon Plata/Oro.

Esto se debía a que, según qué desarrolladora, **se molestaron en programar en el susodicho juego las paletas de colores** para que fuese más vistoso jugarlo en la GBC.

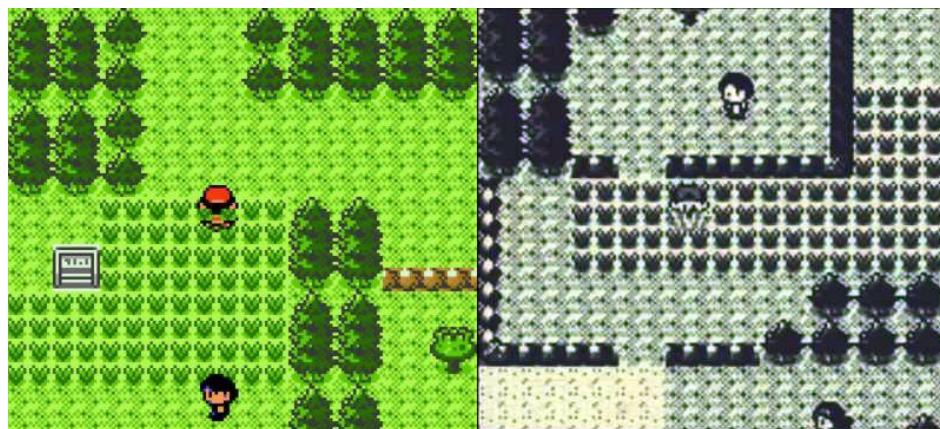


Figura 8.7: Pokémon Rojo/Azul en GBC y GB

A día de hoy aún es complicado encontrar información al respecto para poder conseguir este efecto. Por suerte, **el manual oficial de la CPU de Nintendo describe en detalle el proceso a seguir**. No incluye ningún tipo de ejemplo, pero ya es más que suficiente.

Lo primero que debemos hacer es comprobar qué Game Boy está utilizando el jugador. Esta parte es bastante sencilla, ya que **el hardware nos inserta en el registro 'a' un booleano que nos lo indica**. Con este valor ya podemos pasar a insertar las paletas.

Empecemos por el caso de la GB, el cual es el más sencillo de implementar:

Código 8.6: Paleta para la GB

```

1  BCK_PALETTE EQU %11100100
2  SPR_PALETTE EQU %11100010
3  GBC     EQU $11
4
5  [...]
6
7  cp GBC
8  jr z, .gbc
9
10 ld a, BCK_PALETTE ;cargar dirección de memoria de la paleta en a
11 ld [rBGP], a ;paleta de fondo
12 ld a, SPR_PALETTE
13 ld [rOBP0], a ;paleta para sprites

```

Como podéis apreciar, lo único que hacemos es **insertar un byte en los registros rBGP y ROBP0 para la paleta de fondo y de sprites, respectivamente**. La GB interpreta el valor que le hemos introducido como grupos de 2 bits, siendo 00 el blanco y 11 el negro. **Los bytes en el registro indican el rango del color más oscuro al más claro. El primer byte de la paleta de sprites, independientemente del valor que le demos, va a ser transparente**.

Por otro lado, en el caso de la GBC, la cosa se complica. Nos tenemos que hacer dos fun-

ciones, una para la paleta de fondo y otra para los sprites. **La GB posee dos direcciones de memoria que funcionan como si de un array se tratase.** Tenemos por un lado el registro **\$FF68 o rBCPS** que funciona como índice, mientras que **\$FF69 o rBCPD** son las **posiciones donde meter los valores correspondientes**. Dejar claro que estas direcciones de memoria mencionadas son las pertinentes a la paleta de fondo.

Código 8.7: Paleta para la GBC

```

1 dungeon_palette:
2 DB $32, $AF, $0F, $0F, $07, $06, $02, $10
3
4 Set_Dungeon_Palette:
5
6   ld b, 4
7   ld hl, dungeon_palette
8
9   xor a
10  set 7, a
11  ld [rBCPS], a    ;Auto Increment. This is a Write-Specification register.
12
13 set_dbp:
14
15  ld a, [hl+]
16  ld [rBCPD], a    ;This is a Write-Data register.
17  ld a, [hl+]
18  ld [rBCPD], a
19
20  dec b
21  jr nz, set_dbp
22
23  ret

```

En este caso disponemos de 16 bytes que representan los 4 colores. Descontando el byte mayor (5 bytes para cada canal de color), tenemos $2^{15} = 32768$ colores distintos. Los primeros 5 bytes son para el canal rojo, los siguientes para el verde, y los más altos para el azul.

En cualquier caso, **si necesitas más información** al respecto visita el **apartado 9.6.3** del anexo.

8.2.4 Interrupción V-Blank

Leyendo la documentación oficial de la CPU, podemos llegar a un interesante apartado en el que explican **cómo optimizar las comprobaciones de llegada al rango V-Blank**.

Hasta ahora, lo único que habíamos llegado a ver era el siguiente ejemplo:

Código 8.8: Espera al rango V-Blank: Método 1

```

1 WAIT_VBLANK:
2   ld a,[rLY]      ;get current scanline
3   cp 145          ;Are we in v-blank yet?
4   jr nz,WAIT_VBLANK ;if A-91 != 0 then loop

```

Es un método bastante sencillo que nos puede ayudar en una primera instancia para esperar al rango V-Blank. Sin embargo, **es una mala práctica** ya que **consume mucha más batería** de la consola que haciéndolo de la manera que veremos a continuación. Además, sin ir más lejos, la comprobación la realizan (por lo general) con el valor 145, cuando el rango de V-Blank comienza en el 144 (por minúsculo que parezca, ya es un microsegundo que perdemos por ciclo de juego).

¿Cómo recomiendan hacer estas comprobaciones? Mediante el **uso de su interrupción** (puedes ver **más información** en el **apartado del anexo 9.5**).

La idea es simple: **cuando queramos esperar al V-Blank, hacemos un "halt"** para dejar la consola en stand-by, a la espera de que se produzca cualquier interrupción. La cuestión es que no queremos que eso ocurra, **solo queremos que salga de la espera cuando la interrupción sea la apropiada (la del V-Blank)**. Dicho pues, lo primero que hacemos en nuestra configuración es activar solamente la ya mencionada:

Código 8.9: Activar Interrupción V-Blank

```

1 IEF_VBLANK EQU %00000001
2
3 [...]
4
5 ;enable vblank interrupt
6   ei
7   ld a,IEF_VBLANK
8   ld [rIE],a

```

Al llegar al vector \$40, vamos a **insertar un 1 en una variable global** que nos indica cuándo hemos accedido a la interrupción. Esta variable global la iremos comprobando en la función de espera:

Código 8.10: Espera al Rango V-Blank: Método 2

```

1 RS_WAIT_VBLANK:
2
3   halt
4   nop
5

```

```

6    ld  a,[vblank_flag]
7    or  a
8
9    jr  z, RS_WAIT_VBLANK
10
11   xor a
12   ld [vblank_flag],a
13
14   ret

```

De esta manera, **no optimizamos en bytes de código**, pero **la batería** de la GB del jugador **va a durar mucho más** (puede llegar a extender su vida en un 50%).

8.2.5 Acceso Directo a Memoria

Optimizar el código siempre es una buena práctica y más en este tipo de proyectos en los que disponemos de recursos tan limitados. Aquí es donde el proceso de **Acceso Directo a Memoria o Direct Memory Access (DMA)** entra en juego.

Sabemos que para crear un sprite necesitamos variables guardadas en la OAM, las cuales más tarde la consola interpreta y muestra. El problema con el que nos vamos a topar en cuanto tengamos distintas entidades actualizándose simultáneamente es el de que, para hacerlo, necesitamos estar dentro del rango V-Blank (ya que, de otra forma, la memoria de vídeo es inaccesible). Si además de esto, queremos actualizar la lógica de nuestro juego en este período corto de tiempo, el problema obtiene un efecto *bola de nieve*.

El DMA nos ahorra todo este trabajo, ya que es capaz de enviar bytes de información desde la ROM/RAM a la OAM cada cierto tiempo. Remarcar antes que nada que esta transferencia de datos **dura apenas 160 ciclos**, lo cual es un tiempo diminuto.

Lo primero que deberemos hacer es **reservar e inicializar un bloque de memoria RAM**. Concretamente, este bloque debe ocupar 160 bytes, ya que el proceso transfiere un byte de RAM a OAM por ciclo. Es recomendable reservarlos comenzando en la posición \$C000.

Código 8.11: Reserva de Memoria para DMA

```

1 SECTION "DMA_VARS", WRAMO[$C000]
2
3 ;PLAYER SPRITES
4
5 _SPR0_Y: DS 1
6 _SPR0_X: DS 1
7 _SPR0_NUM: DS 1
8 _SPR0_ATT: DS 1
9
10 [...]
11
12 free_space: DS 108

```

A continuación, vamos a ver el código necesario para que el DMA ocurra:

Existe un **registro especial (\$FF46)** en el que tendremos que indicar la dirección en la que empiezan los datos a transferir. Como solamente son dos bytes, por defecto los bajos serán 00. Una vez metamos el valor en el registro, el DMA comenzará inmediatamente, por lo que tendremos que mantenernos dentro de un bucle a la espera de que termine.

Código 8.12: Inicio del proceso DMA

```

1 ld a, $C0      ;Carga la dirección de memoria $C000
2 ld [$FF46], a
3
4 ld a, $28      ;Espera 160 microsegundos
5 .loop:
6 dec a
7 jr nz, .loop
8 ret

```

El problema es que **mientras el proceso de DMA ocurre, el hardware solo puede acceder a HRAM**. Si implementásemos la función tal y como la he descrito en el código de arriba, se almacenaría en ROM, por lo que **no llegaría a ejecutarse nunca**.

Lo que vamos a hacer es **guardar todo ese código en la dirección \$FF80**, la cual es HRAM, de la siguiente manera:

Código 8.13: Copiado de Proceso DMA

```

1 ld de,$FF80
2 rst $28
3 DB $00,$0D
4
5 DB $F5, $3E, $C0, $EA, $46, $FF, $3E, $28, $3D, $20, $FD, $F1, $D9
6 ret

```

Como ves, lo primero que hacemos es cargar en los registros 'DE' la dirección ya mencionada. Lo siguiente es **provocar un reset de la consola**, lo cual llamará a la interrupción correspondiente (y ahí meteremos el código). ¿Por qué no hacer una función y llamarla con "call"? Porque el opcode "rst" ocupa 1 byte, lo que equivale a ser más rápido.

Los siguientes 2 bytes los utilizaremos de contador para indicar cuánta información copiar. Y por último tenemos 13 bytes que no son, ni más ni menos, que el código de **Inicio del proceso DMA**. A la hora de copiarlo es más sencillo disponer de una tira de bytes a tener lenguaje ensamblador.

Lo único que nos falta es escribir nuestra rutina de copiado en la interrupción ya mencionada, la cual **tiene como dirección de memoria \$0028**:

Código 8.14: Rutina de Copiado en Interrupción para DMA

```

1 SECTION "Reset Table", ROM0[$28]
2 COPY_DMA_DATA:
3   pop hl      ;here we get the return address onto hl. It is the top of the ↵
4     ↪ stack
5   push bc      ;we save the value of bc just in case it has something ↵
6     ↪ important
7
8   ; here we get the number of bytes to copy
9   ; hl contains the address of the bytes following the "rst $28" call
10
11  ld a,[hli]
12  ld b,a
13  ld a,[hli]
14  ld c,a
15
16  ;now bc contains the total bytes to copy ($000D)
17  ;and hl points to the first byte of our assembled subroutine (which is $F5)
18
19 COPY_DMA_DATA_LOOP:
20
21  ld a,[hli]    ;save the data of hl in a and increase it
22  ld [de],a      ;save the data of a in de (FF80)
23  inc de        ;increase the memory address of the destination
24  dec bc        ;decrease the byte counter
25  ld a, b
26  or c
27  jr nz, COPY_DMA_DATA_LOOP
28  pop bc        ;restore the initial value of bc
29  push hl      ;push on to the stack the memory address which is the ret of ↵
30     ↪ the DMA_COPY function
31  reti         ;return to the top of the stack memory address

```

Paso a paso: hacemos un pop a la pila para obtener el valor que tenía de vuelta, el cual es el primer byte \$00 de nuestro **Copiado de Proceso DMA**. De paso, guardamos en la pila el valor que tuviese los registros 'BC' por si hubiese información importante.

Nos guardamos en 'BC' los dos bytes que nos indicaban el tamaño de datos a copiar para usarlo de contador, y comenzamos a meter en las direcciones de los registros 'de' los valores de 'HL' hasta que 'BC' llegue a 0. Una vez terminado, devolvemos a 'BC' su valor inicial sacándolo de la pila y hacemos un push de la dirección de memoria actual en 'HL' para, a continuación, volver con un "reti".

Y para terminar, lo único que tenemos que hacer es, en nuestro bucle principal del juego, hacer un call a la dirección de HRAM para dejar paso al proceso DMA.

8.2.6 Ventanas

Las ventanas de HUD se han intentado implementar en esta iteración. Pese a que se ha implementado tanto la del menú general como la de diálogo (sin texto), **se ha llegado a la conclusión de que hay que rehacerlo**. Aún así, no está de más dejar por escrito el proceso que se ha seguido:



Figura 8.8: Primera Iteración del HUD

Lo primero que hacemos, al igual que con los mapas de fondo, es **hacernos nuestros propios tilemaps** con las ventanas que queramos mostrar. El problema con el que nos vamos a encontrar es el del intervalo V-Blank y su pequeño lapso de tiempo.

Copiar y pegar una tira tan larga de bytes, sin apagar el LCD, y que no se produzca basura visual, es simple y llanamente imposible. ¿Cómo lo solventamos? Podemos modificar la función de copiado de memoria con el fin de, en vez de copiar un tilemap entero, copiar una porción (usando para ello un contador).

Sin ir más lejos, la función obtiene como input el valor del contador (ancho de la ventana a copiar) y un offset (cuantos bytes quedan hasta llegar a la siguiente línea de pantalla). De esta manera, **sólo copiamos la cantidad exacta de bytes** que ocupan esa ventana y no malgastamos tiempo.

El **dónde copiarlo**, sin embargo, no va a ser la misma dirección de memoria que con el mapa de fondo, si no la **segunda pantalla** (escondida por defecto) de la que dispone la Game Boy, cuyo **rango va desde la dirección \$9C00** (la cual indicaremos ahora como destino) **hasta la \$9FFF**.

Para mostrar esta pantalla, deberemos hacer lo siguiente:

Código 8.15: Encendido de la Segunda Pantalla

```

1 SHOW_SCRN1:
2
3     ;Encendemos la pantalla LCD
4     ld a, [rLCD]
5     or LCDCF_WINON
6     ld [rLCD], a

```

```

7
8 ;La coordenada Y de pantalla será el parámetro B
9 ld a, b
10 ld [rWY], a
11
12 ;La coordenada X de pantalla será el parámetro C
13 ld a, c
14 ld [rWX], a
15 ret

```

Como parámetros de entrada necesitaremos las coordenadas X e Y que queremos que tenga la pantalla. Lo que resta es ponerle a la **dirección de memoria \$FF40 (rLCDL)** el valor %00100000 (LCDL_WINON), que **especifica el encendido de la segunda pantalla**.

El problema del rango V-Blank, después de todo, **seguía ocurriendo**, por lo que la única solución que quedaba era apagar y encender la pantalla LCD. Esto provoca que se perciba un **parpadeo poco agradable a la vista**.

8.2.7 Sonido

Los **efectos de sonido y la música** en la actualidad son **esenciales** en un juego. Sin ellos, no solo eres incapaz de producir cualquier estímulo al jugador, si no que no das ningún tipo de *feedback* auditivo sobre lo que está ocurriendo.

Viniendo de sistemas modernos donde simplemente reproducimos un fichero de texto, puede ser un apartado bastante complejo.

En el caso de la GB, dispondremos de **4 canales por los que ser capaces de emitir sonidos**. El propio hardware se encargará de mezclarlos y dar el output que corresponde. No vamos a verlos en este apartado, si necesitas **más información** visita el punto del **anexo 9.7**.

Lo primero que vamos a hacer es **inicializar los registros de uso general**. Tenemos, por ejemplo, **rNR52 (\$FF26)**, que activa o desactiva por completo el sistema de audio.

Código 8.16: Inicialización Registros de Sonido de Uso General

```

1 ld a, %00000111 ; Volumen Máximo del Output 1 y 2.
2 ld [rNR50], a
3
4 ld a, %00000010 ; El Canal 2 solamente saldrá por el Output 2.
5 ld [rNR51], a
6
7 ld a, %10000000
8 ld [rNR52], a      ; Encendido del sistema de audio.

```

Ahora, por cada ciclo de ejecución, vamos a **actualizar la "nota"** que nuestro sistema de audio va a hacer sonar. Como estas notas musicales van a estar en la misma octava, longitud, ciclo de trabajo y envolvente, podemos inicializar solamente una vez los registros de uso general en las siguientes direcciones de memoria: rNR21, rNR22 y rNR24. La frecuencia

"baja" de nuestra nota va a estar guardada en rNR23. También vamos a disponer de un **tempo** por el cual podamos **conseguir distintos ritmos**.

Código 8.17: Sonido en Canal 2

```

1      call    Check_Tempo    ; Comprobar que podemos cargar la siguiente nota
2      or     a
3      ret    nz
4      call    Reset_Tempo   ; Reiniciar el contador del tempo
5
6      ld a, [_NOTA] ; Cargamos la nota a tocar
7      ld c, a
8      ld b, 0        ; BC = Número de nota a tocar
9
10     ld hl, Music_Notes ; HL = Dirección ROM donde están nuestras notas ↪
11         ↪ musicales
12     add hl, bc    ; ahora tenemos la dirección de la nota a tocar
13     ld a, [hl]
14     ld [rNR23], a ; la escribimos en el registro
15
16     call    Save_Note    ; Comprobamos si es la última nota musical (de ser ↪
17         ↪ así reiniciamos a la primera). En caso contrario cargamos la ↪
18         ↪ siguiente.

```

8.2.8 Conclusión

Como conclusión de esta iteración, comentar que **se han conseguido con creces todas las tareas propuestas**. Ya disponemos de un prototipo en el cual nos podemos mover, matar, morir, coger un objeto, y escuchar algún que otro sonido. Quedan **asuntos pendientes** como que el juego sea extremadamente lento, que no podamos tener más de un enemigo, o que para mostrar la ventana de texto tengamos que apagar y encender la pantalla. Sin embargo, puedo decir que **estoy contento con el resultado obtenido**.

8.3 Iteración 2 - Reestructuración y Reimplementación de Código

Esta iteración se ha basado principalmente en la **reestructuración del código**, basándose ahora en el modelo **Entity Component System** (ECS), dejando lo conseguido en la iteración 2 como prototipo del cual poder avanzar, y **centrándonos en hacer reescalable el código** por medio de la creación de **arrays de entidades**.

8.3.1 Entity Component System

En el prototipo de la iteración 1 podíamos tener, única y exclusivamente, un enemigo. Además, se hacía cuesta arriba el poder realizar su actualización al mismo tiempo que la del jugador, ya que cada función contenía tanto las decisiones lógicas de la IA como el cálculo de colisiones, animaciones, etc. Por ello, a la hora de jugar, se podía ver cómo el enemigo solamente tomaba decisiones en acabar la ronda del jugador, lo que además provocaba una **experiencia lenta**.

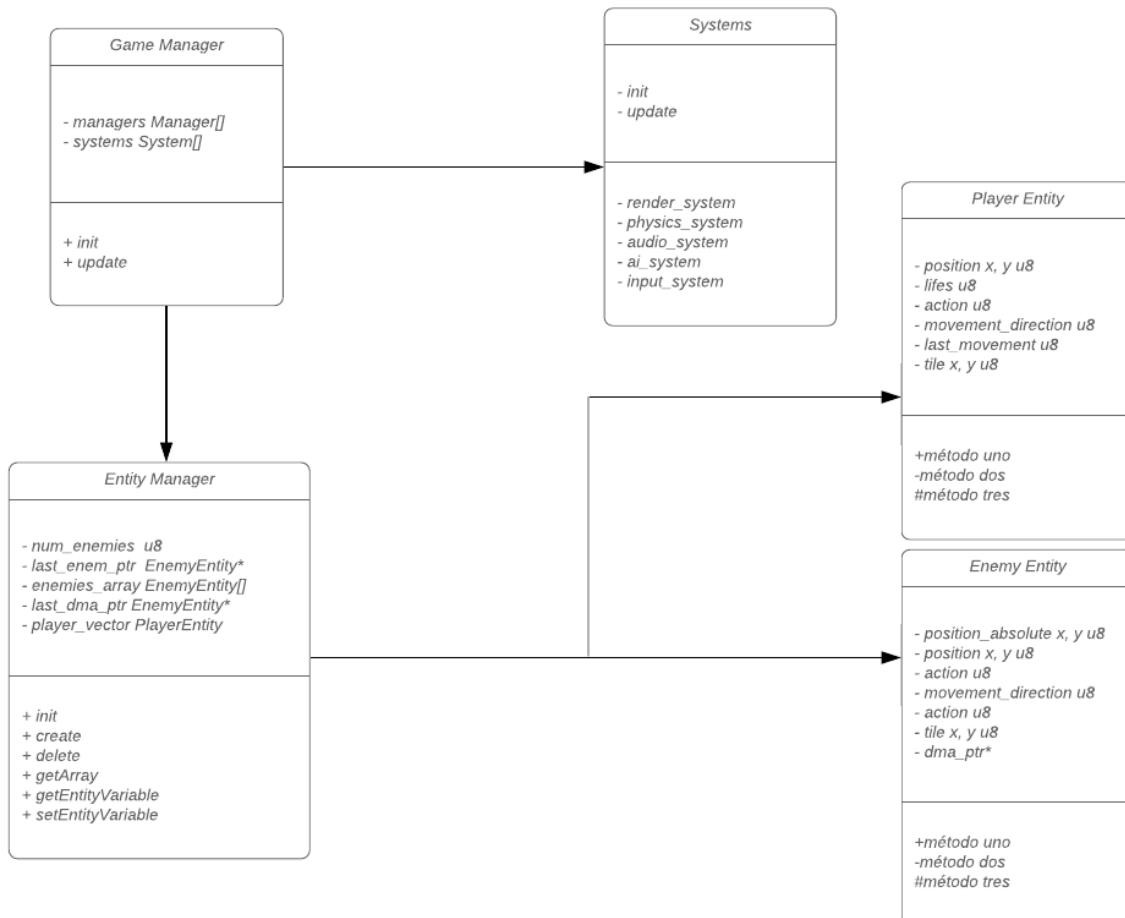


Figura 8.9: Diagrama ECS

Con el modelo ECS, sin embargo, **podemos independizar todas las tareas en sistemas y manejadores de datos**, con lo que seremos capaces de actualizar todo lo que contenían antes tanto el jugador como el enemigo, en un preciso instante de tiempo. Es decir, podemos actualizar la inteligencia artificial de los enemigos por un lado, para luego actualizar el renderizado, y más tarde pasar a las físicas de todas las entidades.

Ya no solo nos ayuda a corregir estos problemas, sino que además habremos conseguido tener el código mucho más legible y ordenado de lo que en un principio estaba. **Las tareas se han conseguido independizar las unas de las otras y centralizar aquellas operaciones que crean, modifican o destruyen datos en los manejadores**.

8.3.2 Array de Entidades

Para la creación y manejo de un array veremos que el proceso un tanto complejo:

A la hora de crear el array, debemos que **tener claras dos cosas: cuántas entidades queremos que hayan al mismo tiempo y cuántos datos va a contener cada entidad**. Una vez hecho esto, simplemente le tenemos que decir al compilador que nos reserve un **Número de entidades * Tamaño de Entidad**. También querremos un **puntero a la última posición en la que podemos crear un enemigo**, la cual es simplemente la etiqueta con la que accederemos al primer valor y el que va a ir incrementando cada vez que creamos otra entidad.

Código 8.18: Reserva Memoria

```

1      ;=====
2      ; Compilation time constants
3      ;=====
4
5
6      enemy_size = 9
7      max_entities = 3
8
9      SECTION "Entities_Data", WRAM0[$C000]
10
11     ; Each sprite needs 4 bytes, that means we can store 160/4 = 40 sprites
12     dma_enemies: DS 64 ; Each enemy consists of 4 sprites
13
14     num_enemies: DS 1
15
16     last_dma_ptr: DS 2
17     last_enem_ptr: DS 2
18
19     player_vector: DS player_size
20     enemie_array: DS enemy_size*max_entities
21     block_array: DS block_size*max_items

```

Sin embargo, para el caso de la Game Boy vamos a tener que crear **dos arrays distintos**. **Uno al cual pueda acceder el DMA** (el cual contendrá solamente datos para almacenar en la OAM) y un array en el cual poder almacenar los **datos restantes como la acción que tiene que hacer o la dirección de movimiento**.

Por ello, tanto a la hora de crear como destruir una entidad, hemos tenido que trabajar con dos arrays distintos, los cuales contienen distintos tipos de datos, y solamente con instrucciones de 8 bits (esto en *Amstrad* es bastante más sencillo al disponer del registro 'IX').

Para **crear una entidad** deberemos que **seguir los siguientes pasos**: coger la dirección del último puntero, leer y copiar los datos desde ROM (con nuestra función de copiado de memoria), poner la siguiente posición como último puntero en la que podemos crear una entidad y aumentar el número de entidades del que disponemos.

En el caso de este proyecto, hemos realizado algo más complejo (y no por ello mejor). Primero, obtenemos el puntero del cual tendríamos que copiar datos por parámetro. Estos datos eran las **posiciones absolutas** (recordemos que los enemigos tienen dos posiciones, una de pantalla y otra de tiles), el número de tile que va a tener el primer sprite de cuatro (arriba a la izquierda), vidas, posición precalculada de tiles, dirección de movimiento y acción.

Con los tres primeros datos lo que tengo que hacer es llenar los datos pertenecientes al array del DMA. Es decir, debo hacer los cálculos apropiados para, a partir de las posiciones absolutas que me dan (recordemos que cada entidad son cuatro sprites juntos), calcular las posiciones de pantalla. También, con el número de tile, debemos incrementarlo y ponérselo al sprite que toque. Esto se traduce en un trozo de código poco legible y que a la larga puede suponer problemas, pero de momento nos sirve.

Con el puntero de la última posición donde podemos crear un enemigo, y con el dato de cuánto ocupa cada entidad (que en este caso sería Tamaño de Entidad-2, ya que los dos primeros datos se guardan junto a los cálculos previos), llamamos a una función de copiado de memoria, el cual ya tenía el origen (el primer puntero pasado por parámetro), destino y tamaño de datos a copiar. Solo queda actualizar ambos punteros de última entidad, aumentar el número de enemigos y listo, ya tenemos una entidad creada.

Para el borrado fue algo más sencillo, ya que simplemente modificamos el tamaño de las entidades y les añadimos un puntero a la posición que les toque en el array de DMA. Con el puntero del array de entidades pasado por parámetro, nos vamos al puntero del DMA de esa entidad. **Cada entidad en DMA ocupa 16 bytes (4 bytes por sprite)**, con lo que ya sabemos el tamaño de datos con el que vamos a lidiar. Ahora obtenemos el puntero al último enemigo del array de DMA y, nuevamente, copiamos y pegamos datos de uno al otro. En este caso tendremos que asegurarnos de que los datos de la última entidad se borren ya que lo que se quede en este array se va a visualizar por pantalla. Actualizamos el último puntero de DMA a la entidad copiada y una cosa menos de la que preocuparse.

Volviendo al otro array, la idea es exactamente la misma, con la diferencia de que no hace falta poner a 0 todos los bytes de la entidad eliminada. Simplemente con decrementar el número de entidades, esos datos ya no se van a usar.

Y por último, para poder iterar sobre todos los enemigos y actualizarlos de forma continua, hemos creado una pequeña función que devuelve los punteros de ambos arrays y el número de enemigos que tenemos en ese preciso instante. Simplemente, al terminar de actualizar una entidad, sumamos al puntero el tamaño de la entidad para actualizar la siguiente en un bucle.

Código 8.19: Get Enemies Array

```
1  EM_GET_ENEMIES_ARRAY:  
2      ld  a, [num_enemies]  
3      ld  hl, enemie_array  
4      ld  de, dma_enemies  
5      ret
```

8.3.3 Conclusión

Si bien parece que, más que avanzar en esta iteración, lo que hayamos hecho haya sido retroceder, creo que era un **trabajo completamente necesario**. Este sistema de entidades además de facilitar la generación de enemigos, va a ayudar también con los objetos, elementos de HUD, creación de distintos niveles, etc. Lo que debo hacer en la siguiente iteración es **intentar aprovechar al máximo el tiempo** del que dispongo para que los resultados se hagan visibles.

8.4 Iteración 3 - Elementos para un Mínimo Producto Viable

En esta tercera iteración se han **seguido añadiendo elementos que ya estaban en el primer prototipo** de finales de iteración 1, pero que tocaban rehacer desde cero porque no eran reescalables.

8.4.1 HUD

Un **elemento imprescindible** en cualquier videojuego: **matar y que te maten**. Y para hacer saber cuánto le falta al jugador para morir, lo mejor es una barra de vida a modo de HUD.

En este caso lo que hemos decidido ha sido cambiar los sprites de corazones que tenía al principio por una **barra** con la que poder representar **mayores rangos numéricos** y que, claro está, sean intuitivos para el jugador. Los corazones lo que le indicaban era más bien que tenía un máximo de 6 puntos de vida y nada más. En este tipo de juegos, el personaje va ganando *stats* o estadísticas conforme sube de nivel, entre los que destaca el aumento de este último mencionado.

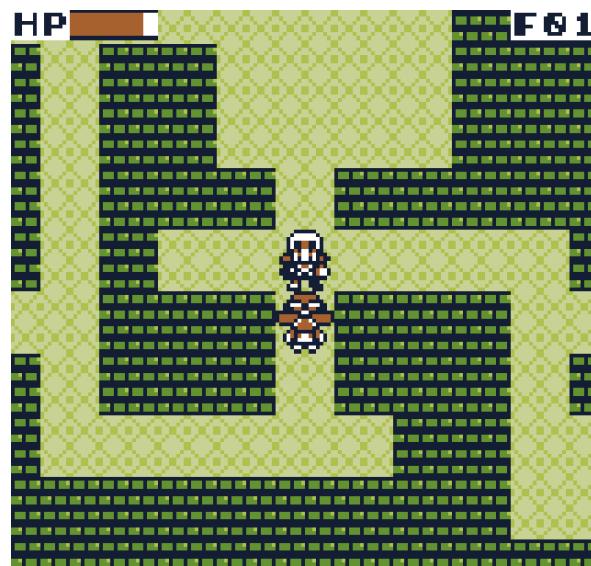


Figura 8.10: Barra de Vida

En temas de cómo se muestran los sprites: simplemente son **variables que almacenamos en la memoria del DMA**. No necesitamos ningún dato en ROM al respecto. Y tampoco tendremos que actualizar la posición absoluta, ya que al ser elementos del HUD **nos interesa que estén siempre visibles en pantalla**.

La actualización de la barra de vida es sencilla: en la inteligencia artificial de nuestro enemigo, comprobaremos primero si el jugador está en un tile contiguo. En el caso de que lo esté, decrementaremos la vida del jugador (de momento lo haremos en un valor fijo, la idea es que puedan darse golpes críticos o varíe en función de la cantidad de daño), y seguido comproba-

remos en qué rango de valores se encuentra nuestro personaje. Esto lo hacemos en divisiones de 6, para poder tener tanto los rangos de los 3 tiles llenos (máxima vida) como vacíos (muerto).

Computacionalmente hablando, no hace falta realizar divisiones en tiempo real para ello, simplemente asignamos de cuánto a cuánto va cada rango, y al valor mínimo de cada uno le restamos la vida actual. **Si la operación nos genera un acarreo o cero, significa que estamos justo ahí.**

Código 8.20: Rangos de Vida

```

1 ; Ranges are: 42-35 | 34 - 28 | 27 - 21 | 20 - 14 | 13 - 7 | 6 - 1 | 0
2
3 ld a, 35
4 sub b           ; This take-away generates carry if we are above the range.
5 jr c, vida_completa
6
7 ld a, 28
8 sub b
9 jr c, dosenteros_unomitad
10
11 [...]

```

También hemos aprovechado para añadir en la esquina derecha superior el **indicador del piso** en el que nos encontramos. Conforme vayamos avanzando por las mazmorras, el contador irá aumentando.

8.4.2 Animaciones

En todo videojuego son necesarias las animaciones. En el caso de no haberlas, hoy en día la primera reacción del jugador va a ser dejarlo completamente de lado sin pensárselo dos veces.

Si bien en el prototipo que se realizó en la iteración 1 contenía ya un enemigo completamente animado, nos sucede (una vez más) que toca rehacer todo de cero porque el código no se podía reescalar.

Vamos a intentar **optimizar lo máximo posible** (al menos, dentro de lo que mi conocimiento actual me permite). Lo fácil para hacer animaciones siempre es cambiar el sprite, cada cierto tiempo, sin complicarse mucho más. El problema es que la Game Boy dispone un **espacio muy limitado de tiles** que podamos usar de forma simultánea, por lo que cuantos menos usemos, mejor.

Dicho pues, el enemigo solamente tiene **4 sprites**. Y efectivamente, el enemigo se nos puede quedar totalmente animado con solamente esos sprites. Vamos a ver cómo es esto posible:

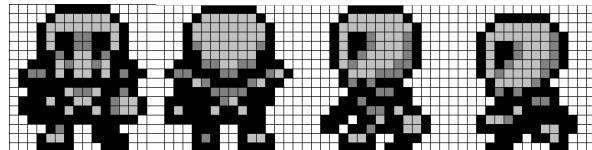


Figura 8.11: Sprites Enemigo

Ya se ha mostrado con anterioridad estos sprites, pero no está de más recordar cómo son. Si nos fijamos en el primer y segundo sprite, comenzando por la izquierda, podrás darte cuenta de la siguiente particularidad: tienen una pierna hacia delante y otra hacia atrás. Lo mismo ocurre con los brazos. Esto lo hemos hecho a propósito para que pudiésemos crear una animación simplemente **invirtiendo los sprites**.

En el caso del **movimiento lateral** ya sí que no nos queda otra que **hacer uso de dos sprites**, e ir cambiándolos según pase un tiempo. Sin embargo, para el movimiento **lateral derecho**, lo que haremos será, de la misma manera, **invertirlos**.

Hacer esto en la GameBoy no tiene mucha complejidad (más allá de gestionarlo de forma correcta). Como ya hemos visto, los sprites en la memoria de vídeo (OAM) constan de **cuatro valores**: posiciones X e Y, número de tile a usar, y un atributo. Hasta ahora este último siempre lo hemos dejado a 0. Pues bien, toca pasar a manejarlos si queremos tener **animaciones optimizadas** (en cuanto a memoria de tiles, pues el código para ello nos va a ocupar algo más en ROM).

Principalmente vamos a usar **dos atributos distintos**, uno para los sprites de arriba, izquierda y abajo, y otro para la derecha (que será el que nos invierta el sprite).

Código 8.21: Atributos de Sprites

```
RIGHT_ATTRIBUTE EQU %00100000
LEFT_ATTRIBUTE EQU %11011111
```

A grosso modo, al iniciar un movimiento se comprueba la dirección y seguido se le inserta, a todos los atributos de los cuatro sprites que conforman el enemigo, el valor que corresponda. Además de esto, tendremos que hacer un **"intercambio" de coordenadas X entre sprites**, de la forma que el sprite superior-izquierda pase a ser el superior-derecha y viceversa. Realmente **no necesitamos cambiar las coordenadas** de todos ellos **en memoria de vídeo**, con saber que los sprites están intercambiados **a la hora de calcular las posiciones absolutas** **lo podemos falsificar**.

Obviamente, esto no es tan fácil como aparenta ser, y más cuando el enemigo tiene distintas acciones o estados que puede realizar al mismo tiempo. Por ejemplo, el enemigo puede estar atacando con los sprites intercambiados o no, y dependiendo de ello tendremos que saber manejar la forma en la que se actualiza el comportamiento general ("¿Debo volver a

intercambiar los sprites?", "¿O tengo que volver a poner los atributos de la izquierda?").

En cualquier caso, el método principal de animación para los movimientos de arriba y abajo lo podemos ver en el siguiente código. Como podrás comprobar, lo que hacemos es indicar que queremos intercambiar los sprites a la hora de dibujarlos y cambiarles los atributos:

Código 8.22: Atributos de Sprites

```

1 ld a, [_EN_TIMER]
2 dec a
3 ret nz
4
5 push hl          ; Guardamos dirección en HL
6 ld e, en_action
7 call EM_GET_ENEMY_VARIABLE ; Obtenemos acción del enemigo
8 xor SWAP_ACTION      ; Realizamos operación XOR con el SWAP
9 ld [hl], a          ; Insertamos resultado en memoria ROM
10 pop hl            ; Recuperamos dirección en HL
11
12 push hl
13 ld b, XOR_RIGHT
14 call EM_CHANGE_EN_ATTRIBUTES ; Cambiamos los atributos
15 pop hl
16
17 ret

```

Como veréis, la acción del intercambio se indica con una operación XOR, ya que lo que quiero de esta animación es que, si los sprites están invertidos, los vuelva a dejar en su estado original y viceversa cada vez que entre a la función y el tiempo de animación nos indique que podemos realizarlo.

8.4.3 Colisiones - Casos Especiales

Si bien los enemigos no se meten en la casilla de otra entidad, hay un par de **casos especiales que conviene corregir**, ya que prácticamente dejan el juego no funcional. Estos dos casos específicos son los que se muestran en la siguiente imagen:

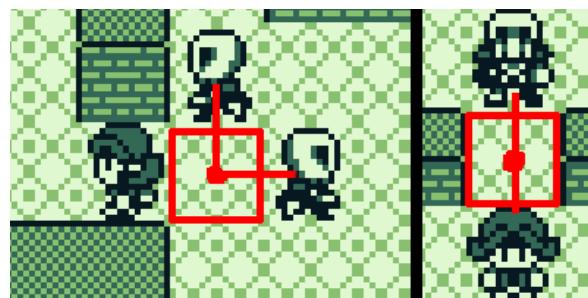


Figura 8.12: Casos Especiales en Colisiones

En la captura de la **izquierda** lo que sucede es que **los dos enemigos calculan que deben ir a la misma casilla**, lo que provoca que terminen juntándose en ella. En la imagen de la **derecha**, el problema emerge cuando **el jugador quiere ir hacia arriba, pero el enemigo ya ha calculado que debe ir hacia abajo**.

El segundo caso lo vamos a conseguir corregir a tiempo para esta iteración. Lo que hacemos es lo siguiente: en el sistema de físicas, una vez recibimos un input, actualizamos al jugador y luego iteramos sobre todos las entidades. Esto nos deja una pequeña vía de escape en la cual podemos detectar que el jugador se está moviendo a la misma casilla que el enemigo tiene calculada, y poder impedir así que este último se mueva.

Código 8.23: Comprobar Tiles y Denegar Movimiento

```

1 ld a, [_NPIXM]
2 dec a
3 jr nz, .ps_move_en_up ; Comprobamos que estamos en la primera iteración.
4
5 call AI_CHECK_TILES      ; Comprueba que el jugador no está en la misma ↪
   ↪ casilla
6
7 ld b, h
8 ld c, l
9
10 or a          ; Si no es 0, impedimos el movimiento.
11 jp nz, ps_en_mov_deny

```

Lo que se muestra en el código no es ni más ni menos que el **método de movimiento del enemigo**. Al comprobar en qué dirección se mueve el jugadore/a (y si es el primer frame del enemigo) se procede a comprobar si los tiles que tiene el jugador (los cuales ya están actualizados al haber realizado una iteración) son los mismos que a los que se tiene que mover.

El método de comprobación de tiles es muy simple: cogemos los tiles del jugador, los guardamos en el manejador de colisiones, y con la dirección del vector de enemigos apropiada comprobamos cada coordenada de los tiles. Si ambos son iguales, devolvemos un 1 para hacer saber al sistema de físicas que debe finalizar la ejecución.

8.4.4 Mapas y Estados de Juego

Como se diseñó en el GDD, el juego iba a constar de dos estados principales: el **mundo exterior y las mazmorras**. Con esta idea en mente, se van a incorporar al juego 3 mapas principales:



Figura 8.13: Mapas Principales

Dependiendo de en qué mapa se encuentre el jugador, **activaremos unas mecánicas u otras**. Por ejemplo, en el mundo exterior el usuario no va a poder atacar, perder vida, pasar hambre, etc.

La implementación que se ha seguido para conseguir esto es la siguiente: una vez se detecta una casilla de tipo puerta, se pasa a comprobar su número de tile y se carga el mapa correspondiente. Como es probable que la pila siga teniendo returns pendientes que pueden afectar al gameplay, lo que hacemos es esperar una iteración entera, consiguiendo vaciarla. Para ello dispondremos de **dos bytes en RAM que indican si hay que cambiar de un estado a otro y a cuál**.

Código 8.24: Mapas de Juego

```

1 _STATE:    DS 1          ; 0 = Overworld, 1 = Dungeon, 2 = Transition
2 _CHANGE:   DS 1          ; bool to know if we have to change state
3
4 [...]
5
6 GM_SET_GAME_STATE:
7     ld  [_STATE], a      ; Cambiamos el estado.
8     ld  a, 1
9     ld  [_CHANGE], a      ; Indicamos que tenemos que cambiar de estado.
10    ret

```

Lo que hacemos a continuación, nada más entrar al bucle de juego, es **comprobar si debemos cambiar a ese estado** previo al comienzo de actualizar la inteligencia artificial, las físicas, animaciones, etc.

Código 8.25: Bucle de Juego

```
1 [...]  
2 .gameloop:  
3  
4     call GM_CHECK_STATE      ; Compruebo estado.  
5     call GM_UPDATE_INPUT     ; Actualizo inputs  
6     call RS_UPDATE          ; Actualizo renderizado  
7     call AS_UPDATE          ; Actualizo animaciones  
8     call AI_UPDATE          ; Actualizo inteligencia artificial  
9  
10    call RS_WAIT_VBLANK     ; Espero al intervalo V-Blank  
11    call RS_DRAW            ; Dibujo sprites en OAM mediante DMA.  
12    call PS_UPDATE          ; Actualizo físicas.  
13    jp .gameloop
```

8.4.5 Conclusión

En este caso creo que **el trabajo resultante no ha sido el esperado**. Me he encontrado con distintos problemas a lo largo de este período de tiempo que han perjudicado al desarrollo del proyecto. Esto quiere decir que, probablemente, voy a tener que **descartar alguna que otra tarea de la lista** para asegurarme de que lo más importante queda bien plasmado.

8.5 Iteración 4 - Implementación de Elementos de Producto

En esta iteración el **objetivo principal** ha sido el de **conseguir un producto**. Con esto veremos que el enfoque principal es el de hacer posible la muerte del jugador y el reinicio de partida, el paso de nivel, el poder matar enemigos, usar objetos, etc.

8.5.1 Colisiones

Nuevamente volvemos a hacer una **revisión de este aspecto** del juego. Ya no solamente para terminar lo que nos quedó por arreglar, si no para **rehacer el código de 0**.

Tras una mejor **analización del problema**, se llegó a la conclusión de que **tratar esta serie de problemas como casos especiales generaba demasiados "parches" en el código**. A parte, ya existían **métodos mucho más funcionales** que solucionaban todos los problemas de una vez.

La clave está en **guardarse en un array todas las casillas en las que los enemigos van a moverse o se han movido**. De esta forma, las entidades "reservan" casillas. Puede que parezca lo más lógico pero realmente es complicado caer en la cuenta de poder hacerlo de esta manera. **Juegos como Pokemon: Mundo Misterioso usan este método**.

Lo primero que debemos hacer es, como ya hemos dicho, crearnos un array donde vayamos a guardar todas las casillas reservadas. Es muy simple, **reservamos un espacio de 2 * Número de Entidades**.

Código 8.26: Definición de Espacio para la Reserva de Casillas

```
1 _RESERVED_CELLS: DS MAX_ENTITIES*TILE_COORDS ; 2 Tiles * Maximum Number of ↵
    ↵ Entities
```

Ahora, de manera **previa a que un enemigo calcule una casilla a la que ir**, va a **hacer la comprobación correspondiente para conocer si ya está reservada**. En caso de estarlo, pasaría a comprobar si puede moverse a otra.

A la hora de iterar sobre el array de casillas reservadas, no es conveniente malgastar memoria RAM poniendo a cada entidad un puntero a la posición de su casilla correspondiente. Conforme vayamos iterando por entidad, con el índice del bucle sabremos qué posición exacta del vector corresponde a cada uno.

Una vez sabe la entidad a qué casilla debe moverse, la reservará para que, una vez llegados a la actualización de las físicas, comience a moverse **sin que haya otra entidad que pueda hacerlo también a esa misma posición**.

Código 8.27: Reserva de Casillas

```
1 AI_RESERVE_CELL:
2
3     ld hl, _RESERVED_CELLS ; Posición inicial del vector de casillas ↵
        ↵ reservadas.
```

```

4 ld a, [_TOTAL_EN]      ; Índice de la entidad actual.
5 ld de, 2                ; Número de coordenadas por casilla.
6
7 ai_reserve_cell_loop:
8
9 dec a
10 jr z, ai_reserve_cell_do ; Si nuestro índice es 1, vamos directamente ←
    ↪ a coger coordenadas.
11
12 add hl, de            ; En caso contrario, sumamos el tamaño de casilla a la ←
    ↪ dirección de memoria.
13 jr ai_reserve_cell_loop
14
15 ai_reserve_cell_do:
16     ; Get the saved tiles from the collision manager
17
18 call CM_GET_TILEY      ; Cogemos coordenada Y
19 ld [hl], a              ; Guardamos coordenada Y
20 inc hl
21 call CM_GET_TILEX      ; Cogemos coordenada X
22 ld [hl], a              ; Guardamos coordenada X
23 ret

```

Como es lógico, en la iteración de la siguiente entidad lo que vamos a hacer es recorrer este mismo vector, de manera que comprobemos las coordenadas X e Y de la casilla a la que se intenta mover con todas las que ya hay reservadas. **En caso de que ambas coordenadas coincidan con alguna de ellas, impedimos el movimiento y probamos con otra.**

8.5.2 Progresión de Juego

Para conseguir un **producto** era de suma importancia tener algún tipo de **progresión dentro del juego**. En este tipo de juegos se consigue mediante el acceso a "pisos inferiores/superiores" del cual se encuentra actualmente el jugador (como podemos observar en la **figura 8.10**). En este proyecto la propuesta inicial es la de conseguir **3 pisos en total**, más la última de todas donde se pueda pelear contra un enemigo final.

Para ello crearemos **una variable RAM en el manejador de partida**, que nos pueda indicar en qué piso nos encontramos actualmente. Dado este índice, **una vez detectemos que estamos en un tile especial** (el cual ya hemos nombrado en algún momento como escalera), **pasaremos a cargar el tilemap correspondiente**.

Para el cargado de tilemaps, al ser tan grandes, **debemos apagar y encender la pantalla LCD para evitar posibles corrupciones en el bus de vídeo**. El resultado que nos da es un efecto parpadeo no muy agradable a la vista, pero el cual **podemos solucionar con transiciones** (como veremos en el siguiente apartado).

Por último, todos los tilemaps (igual que con el que ya teníamos), los guardamos en ROM realizando su inclusión, y la cantidad de enemigos y objetos las creamos una vez comprobados en qué nivel nos encontramos:



Figura 8.14: Muestra Gráfica de Progresión de Juego

Código 8.28: Carga de Niveles

```

1 GM_INCREMENT_ACTUAL_FLOOR:
2
3     ld a, [_ACTUAL_FLOOR]      ; Cogemos indice de piso actual
4     inc a                      ; Incrementamos el piso actual
5     ld [_ACTUAL_FLOOR], a       ; Guardamos el nuevo valor
6
7     jp RS_UPDATE_FLOOR_HUD    ; Actualizamos el HUD.
8
9 GM_GO_NEXT_FLOOR:
10
11    call GM_INCREMENT_ACTUAL_FLOOR ; Llamamos a la función de arriba
12
13    ; Eliminamos objetos y enemigos actuales.
14    call EM_DELETE_ALL_ENEMIES
15    call EM_DELETE_ALL_BLOCKS
16    ld a, [_ACTUAL_FLOOR]        ; Comprobamos que nivel tenemos que cargar
17    cp 2
18    jr z, .floor2
19    cp 3
20    jr z, .floor3
21    ret
22    [...]
23
24.floor2:                 ; Cargamos piso 2
25    ld hl, Floor2             ; Dirección de inicio del tilemap
26    ld bc, Fin_Floor2-Floor2   ; Cantidad de bytes que ocupa el tilemap
27    call RS_INIT_FLOOR         ; Dibujado del tilemap
28
29    ld hl, enemy_entity2      ; Creación de entidades
30    call EM_CREATE_ENEMY
31    [...]
```

8.5.3 Transiciones entre Estados de Juego

Como hemos comentado en el apartado anterior, el apagado y encendido de la pantalla LCD genera un efecto visual desagradable para el usuario. La mejor manera de solventarlo es mediante una **transición *fade-in* y *fade-out* de los colores**. Este es un método que juegos como **Pokémon Rojo/Azul** ya utilizaban de manera constante para hacer pasar por desapercibidos los tiempos de carga de los distintos tilemaps.

La idea es sencilla: **por cada frame vamos a ir eliminando un color de nuestra paleta**. Esto quiere decir, por otro lado, que deberemos hacer distinción entre la Game Boy Original/Pocket y la Game Boy Color, puesto que la cantidad de colores (como ya hemos visto) es mucho más amplia en este último modelo.

Veamos el caso de las primeras Game Boy:

Como sabemos, la GB contiene un máximo de **cuatro colores**, los cuales van de más claro a más oscuro (teniendo en cuenta que los bits se cuentan de derecha a izquierda). Cada color se representa con 2 bits, siendo 11 el negro y 00 el blanco. Además, la paleta de sprites tiene la particularidad de que el color más claro, es en realidad transparente.

Código 8.29: Paleta de Color en GB

```
BCK_PALETTE EQU %11100100 ; GB background palette
SPR_PALETTE EQU %11100000 ; GB sprite palette
```

Vamos a tener la necesidad de gestionar los primeros dos colores de la paleta de fondo de manera inversa. Esto se debe precisamente porque el primer color de la paleta de sprites es transparente, por lo que pongamos los bits que pongamos, el resultado va a ser el mismo. Al poner las dos paletas con los mismos colores, va a suceder lo siguiente:

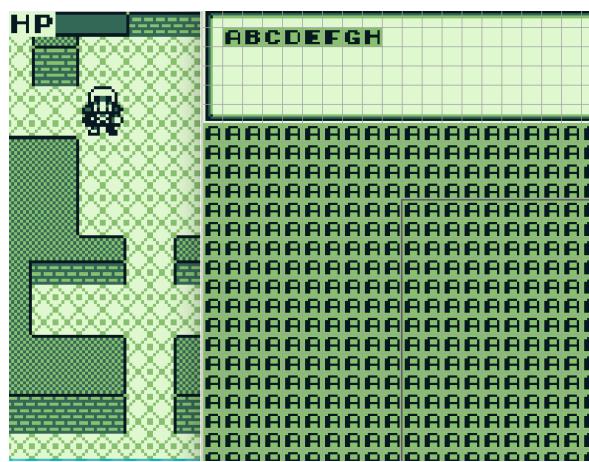


Figura 8.15: Problema con la Paleta de Color en GB

Si nos fijamos bien, veremos que **los tiles de las letras, en el HUD tienen un fondo blanco, mientras que en el recuadro de la derecha tienen un fondo verde**. Esto se

debe a que el color gris en la paleta de fondo es el color blanco en la de sprites.

Al final la solución es la de, **en el programa GBTD, intercambiar los colores de blanco a gris y viceversa en todos aquellos tiles que se utilizasen para el fondo y el resto dejarlos igual**. Ahora necesitaremos intercambiar estos dos mismos colores pero en el código. Todo esto provocaba que los tiles que utilizamos de sprites se viesen de la misma manera que si las utilizase de fondo, y que los demás tiles se viesen como tocaban.

Código 8.30: Paleta de Color Invertida en GB

```
1BCK_PALETTE EQU %11100001 ; GB background palette
2SPR_PALETTE EQU %11100000 ; GB sprite palette
```

Volviendo al tema principal, lo que debemos hacer para conseguir la **transición a blanco** es, **por cada frame o iteración (en un total de 4 iteraciones)**, restar 1 a todos los **colores**. De esta forma, el color negro pasa a gris, el gris a blanco, etc.

Por otro lado, en la **transición de blanco a color**, debemos ir **sumando 1 a todos los valores, mientras que los comparamos con el valor de la paleta original**. Una vez una componente sea igual que la de la paleta original, dejamos de incrementarla.

El **resultado** es el siguiente:



Figura 8.16: Transición en Game Boy Original/Pocket

Para el caso de la GBC la idea es exactamente la misma tanto para el *fade-out* como para el *fade-in*. La dificultad está en el comprender cómo gestiona los colores.

El mayor problema es la componente G, debido a que se halla intercalada entre el primer y el segundo byte. Como vemos además, cada componente dispone de 5 bits, lo cual nos da un total de 2^5 **posibilidades**, una vez más, **por componente**. Esto quiere decir que si, al igual que en GB, queremos conseguir el blanco absoluto en 4 iteraciones, **deberemos**

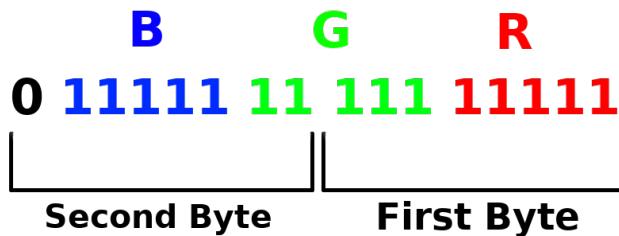


Figura 8.17: Estructura de Color en GBC

restar 1 a cada componente durante 32-1 iteraciones (contamos el 0).

Para restar o sumar la componente G por separado, deberemos coger el primer byte, hacerle una operación AND a los 3 últimos bits, efectuar 3 veces la operación RLC (o por el contrario, un SWAP y un RRC) para pasar los tres bits al principio del byte, y guardarnos el resultado. A continuación cogemos el segundo byte y nuevamente aplicamos un AND para guardarnos los dos primeros bits. Hacemos otro SWAP y otro RRC para no machacar los que tenemos en el segundo registro, y con una operación OR de ambos conseguimos la componente completa, la cual ya podemos decrementar. Para guardar de nuevo el valor tendríamos que efectuar justo los pasos marcados, pero a la inversa.

Código 8.31: Conseguir Componente G de la Paleta de Color en GBC

```

1 ld a, [hl]           ; a = GGGRRRRR
2 and %11100000          ; a = GGG00000
3 swap a               ; a = 0000GGG0
4 rrc a                ; a = 00000GGG
5 ld b, a              ; b = a
6 inc hl
7 ld a, [hl]           ; a = 0BBBBBGG
8 and %000000011         ; a = 000000GG
9 swap a               ; a = 00GG0000
10 rrc a                ; a = 000GG000
11 or b                 ; or = 000GGGGG
12 ld [_GBC_PAL_G], a    ; save green component in ram.

```

Como podéis apreciar, **esto es solamente para coger la componente verde** (a falta de todo lo demás), por lo que se genera mucho más código. Por ello es **importante comentar en cada operación el resultado esperado** y, de esta manera, evitar tener que ponernos a hacer trazas en caso de que algo vaya mal.

8.5.4 Objetos y Hambre

Este apartado estaba, al igual que los enemigos, hecho ya desde el primer prototipo a finales de la iteración 1. Esto conlleva, sin embargo, acarrear exactamente los mismos problemas.

La solución, por ende, es **crearnos nuevamente un array** (tanto en RAM como en DMA) de entidades que sean específicamente objetos. Y esto hay que cogerlo con pinzas porque es

una decisión de diseño y estructuración del código.

Código 8.32: Vector de Objetos

```

1 block_size = 7 ; AbsY, AbsX, Type, TileX, TileY, DMA Address [x2] (Little ↪
    ↪ Endian).
2 block_dma_size = 4
3
4 dma_blocks: DS 48 ; A 'block entity' could be an item, which consists only of 1 ↪
    ↪ sprite
5
6 num_blocks: DS 1
7
8 last_dmabl_ptr: DS 2
9 last_block_ptr: DS 2
10
11 block_array: DS block_size*MAX_BLOCKS

```

Dentro del código este tipo de entidades las vamos a definir como "bloques" ya que **solo constan**, a diferencia del resto, **de un sprite** y pueden interactuar con el jugador (aquí la diferencia con los sprites de HUD).

La lógica de creación de la entidad, eliminación, etc. es la misma que la de los enemigos, por lo que no creo conveniente volver a explicarlo.

Por ahora, **conforme al Game Design Document**, vamos a disponer de **tres posibles objetos**, los cuales se encuentran esparcidos por el mapa. Esto nos obliga a agrandar el tamaño del jugador en **3 bytes más**, para saber con certeza cuántos disponemos de cada uno. Además, vamos a añadir un **cuarto byte como indicador de hambre** que tiene el jugador (mecánica descrita en el GDD).

Código 8.33: Nuevo Tamaño del Jugador

```

1 player_size = 10 ; Lifes, TileX, TileY, MovementDirection, LastMovement, ↪
    ↪ Action, Hunger, Basic Potions, Super Potions, Basic Apples.

```

Como podemos observar, ahora disponemos de un byte respectivo para cada tipo de objeto.

Lo siguiente que debemos hacer es, **cada vez que el jugador termina un movimiento, comprobar si ha colisionado con alguno de los objetos que el array contiene**. Esto lo hacemos de una manera muy sencilla: comprobando las coordenadas X e Y de la casilla en la que se encuentran ambos. En el momento que demos con una coincidencia, comprobaremos el tipo de objeto y sumaremos 1 al byte correspondiente del jugador.

Código 8.34: Actualizaciones a Final de Movimiento

```

1 PS_UPDATE_NPIXM:
2     [...]
3     call EM_INCREASE_PLAYER_LIFE      ; aumentamos la vida del jugador
4     call EM_DECREASE_PLAYER_HUNGER   ; decrementamos el hambre
5
6     call AM_RESET_PLAYER_SPRITES    ; reseteamos los sprites

```

```

7
8   call EM_GRAB_ITEM           ; comprobamos si podemos coger un objeto.
9
10  jp  AI_INIT

```

Una vez hemos obtenido el objeto, simplemente llamamos a la lógica de eliminación para que deje de ser visible en pantalla.

En el caso del hambre, por otro lado, lo que hacemos es decrementar el valor actual que tiene guardado el jugador y, de llegar a 0, comenzar a restar un valor mayor del que hemos sumado previamente al terminar el turno.

Código 8.35: Actualización del Hambre

```

1 EM_DECREASE_PLAYER_HUNGER:
2
3   ld  e, pl_hunger
4   call EM_GET_PLAYER_VARIABLE    ; Cogemos el hambre actual.
5   or  a             ; Comprobamos que sea 0.
6   jr  z, EM_DECREASE_PLAYER_LIFE ; Si es 0, decrementamos la vida actual ↪
                                     ↪ del jugador.
7
8   dec a             ; En caso de no ser 0, decrementamos el hambre.
9   ld  [hl], a
10
11  ret

```

A falta de los métodos de recuperar vida dado un input (para según qué objeto sumar un valor u otro) y el de recuperar hambre, este apartado se podría llegar a dar por finalizado. Para poder utilizar los objetos, **necesitamos un sistema de ventanas desde el cual dar la respectiva accesibilidad al jugador**.

8.5.5 Ventanas - Diálogos y Menús

La siguiente preocupación es el renderizado de ventanas. Como ya hemos comentado en la iteración 1, para poder dibujar una ventana debemos apagar y encender la pantalla LCD. Esto, al igual que ocurre con el cargado de niveles, **no se puede arreglar con una transición**.

Las primeras pruebas consistieron en esperar a un intervalo H-Blank o V-Blank y pintar de forma seguida 3 tiles en la segunda pantalla. Lo que sucedía era que al entrar **en V-Blank las pintaba sin problemas, pero en un intervalo H-Blank no le daba tiempo**.

La solución fue dejar a 1 la cantidad de tiles que se dibuja por intervalo H-Blank o V-Blank, dando por asegurado que se iban a dibujar todas y cada una de las ventanas que se quisiesen.

Finalmente, la lógica del dibujado de ventanas funciona de la siguiente: nos pasan como **parámetros un ancho y un alto**. La esquina superior izquierda de la segunda pantalla siempre va a ser la esquina de una ventana, por lo que la iniciamos al tile que toca, y ponemos los tiles de borde superior un total de *ancho* – 2 veces (descontamos las dos esquinas).



Figura 8.18: Problemas Dibujado de Ventanas Emergentes

Ahora simplemente insertamos la esquina que falta. Hasta que no lleguemos a la última línea de todas, vamos a efectuar la misma operación: *bordeizquierdo - blanco * N - bordederecho*, (siendo $N = \text{ancho} - 2$). La última línea la hacemos idéntica a la primera, pero con los tiles inferiores. A continuación se muestra el código descrito:

Código 8.36: Dibujado de Ventana

```

1 RS_DRAW_WINDOW:
2
3 ld a, 33      ; Cargamos el ancho total + 1.
4 sub e          ; Le restamos el ancho pasado por parámetro.
5 ld [_OFFSET], a ; Nos lo guardamos en RAM.
6
7 ld hl, _SCRN1 ; Cargamos la primera posición de la segunda pantalla.
8 call RS_WAIT_MODE_01 ; Esperamos H-Blank o V-Blank.
9 ld [hl], CORNER_LU ; Cargamos el tile de esquina superior izquierda
10    ↪ .
11 inc hl        ; Incrementamos 1 vez la dirección de memoria.
12
13 ld d, e      ; Cargo en 'd' el Offset
14 dec d        ; Le resto 2
15 dec d
16
17 .rs_draw_window_loop_up:
18
19 ld c, 1      ; Cargo en 'bc' el número de bytes a copiar.
20 ld b, 0
21 call RS_WAIT_MODE_01 ; Esperamos H-Blank o V-Blank
22 ld a, WINDOW_UP ; Cargamos el tile de borde superior

```

```

23  call RS_COPY_DATA          ; Input: a(var), hl(origin), bc(number of bytes)
24                                ; Modified registers: hl, ↪
                                ;           bc, af, e
25  dec d                      ; Decremento 'd'
26  jr nz, .rs_draw_window_loop_up ; Si 'd' no es 0, continuamos con el bucle ↪
                                ; .
27
28  call RS_WAIT_MODE_01        ; Esperamos H-Blank o V-Blank
29  ld [hl], CORNER_RU         ; Insertamos la esquina superior derecha
30
31  ld a, [_OFFSET]            ; Recuperamos el Offset y lo insertamos en 'de'
32  ld e, a
33  xor a
34  ld d, a
35  add hl, de                ; Añadimos el Offset a la posición actual para llegar ↪
                                ; a la siguiente línea.

```

La **primera prueba**, como podréis comprobar en las capturas posteriores, fue **mostrar un pequeño texto en el momento que el usuario obtiene un objeto**, con el nombre de este. Para que todos los textos estuviesen organizados, hicimos un manejador nuevo que los guardase en ROM. Por lo general, lo que se suele hacer es guardar las cadenas de texto igual que en C++, pues el ensamblador lo pasa a bytes en tiempo de compilación. Después de eso, en tiempo de ejecución nos quedaría restarle al byte obtenido la primera posición de nuestro abecedario en memoria para coger el tile correspondiente. En este caso, **metimos directamente los bytes** correspondientes a los tiles que queríamos que saliesen.

Código 8.37: Textos en ROM

```

1; "Ashia got"
2Item:::
3  DB 0, 18, 7, 8, 0, WHITE_TILE, 6, 14, 19
4End_Item:::
5
6; "Basic Apple"
7BasicApple:
8  DB 1, 0, 18, 8, 2, WHITE_TILE, 0, 15, 15, 11, 4
9End_Basic_Apple:
10
11 [...]

```

Cada vez que el jugador escoja un objeto, sacaremos el texto "Ashia got" seguido del nombre específico. Vamos a hacerlo de la misma manera que hemos creado la ventana, con la única diferencia de que por cada tile que dibujemos, **aplicamos un delay de N frames** (al gusto del programador). En resumen: esperamos un número determinado de veces al intervalo V-Blank. Esta técnica la implementaban muchos juegos a la hora de mostrar diálogos.

Código 8.38: Delay

```

1RS_DELAY_FRAMES:
2  ld a, c                  ; Cargamos en a el input (frames a esperar)
3  or a                     ; Comprobamos que el parámetro sea distinto de 0.
4  jr nz, rs_delay_frames_vblank ; Si no es 0, pasamos al bucle.

```

```

5
6   jp RS_WAIT_MODE_01      ; En caso de ser 0 esperamos a un H-Blank.
7
8 rs_delay_frames_vblank:
9
10  call RS_WAIT_VBLANK     ; Esperamos al intervalo V-Blank.
11  dec c                  ; Decrementamos el contador.
12  jr nz, rs_delay_frames_vblank ; Si no es 0 volvemos al inicio del bucle.
13  ret

```

El resultado es el siguiente:



Figura 8.19: Dibujado de Textos

Por último, algo que estaría bien comentar es la implementación realizada en este proyecto. Como ya hemos analizado previamente en el Estado del Arte, la **implementación de las entregas Pokémon Rojo/Azul se basan en el "copia y pega" de la pantalla secundaria a la primaria**. Por nuestra parte, lo que hacemos es **crear una única ventana que ocupe un ancho y un alto, con su esquina superior izquierda en la coordenada (0,0) de la segunda pantalla**. La ventaja es que el proceso es mucho **más rápido** a la hora de dibujarlo, pero tiene como inconveniente el de ser **incapaz de mostrar más de una ventana emergente** de forma simultánea.

8.5.6 Conclusión

Sin duda alguna siento que, hasta ahora, esta ha sido la iteración en la que **más trabajo se ha llevado a cabo**. Con esto prácticamente tengo el juego terminado a **falta de algunas mejoras** como pueden ser **efectos de sonido, retoques visuales, o alguna mejora en cuanto a inteligencia artificial**. Son, sin embargo, **tareas muy secundarias** que si se quedasen en el tintero no afectarían de manera grave al proyecto. Lo que sí necesito asegurarme ahora para que el producto sea totalmente jugable es que exista la posibilidad de utilizar los objetos esparcidos por el mapa.

8.6 Iteración 5 - Diseño de Niveles para Enseñar al Usuario a Jugar

En esta iteración el **objetivo principal** es terminar el **diseño de niveles** (objetos, obstáculos, enemigos...) de manera definitiva. Además, los primeros niveles del juego se deben pensar de forma que puedan guiar al jugador y enseñarle a jugar sin necesidad de más feedback.

8.6.1 Actualización de RGBDS

Como es lo normal en todo proyecto, con el paso del tiempo **las herramientas** que utilizamos **van obteniendo actualizaciones** que ofrecen al usuario todo tipo de mejoras y optimizaciones. De la misma forma, hay aspectos o nomenclaturas que tenemos en nuestro código y que los desarrolladores han creído conveniente dejar como obsoleto y reemplazarlos por otros.

Esto es precisamente lo que ha sucedido con este proyecto. Por suerte, los cambios no han afectado en gran medida.

Lo que se debe hacer en estas circunstancias es ir directamente a la **documentación oficial del desarrollador** y buscar los cambios que más nos interesen, como las eliminaciones:

Deprecations and removals:

- As part of a cleanup effort, features previously marked as deprecated have been removed, such as the `CODE` section type
- Labels not starting with a dot nor followed by a colon have been deprecated; in a future version, they will be treated as macro invocations
- Deprecated `OPT z` in favor of new and more consistent `OPT p`
- Deprecated `GLOBAL` symbol (and its synonym `XDEF`), as it has the same effect as `EXPORT`
- Removed "section-local" charmap (deprecated in 0.3.9)

Figura 8.20: Actualización 0.4.0 RGBDS - Código Obsoleto

Las **modificaciones importantes** fueron la **segunda y la cuarta**. La segunda lo que nos dice es que las etiquetas que empiecen con un punto se van a tratar como llamadas a una macro. Estos puntos los poníamos más por cuestiones estéticas que por otra cosa, ya que ayudaban a diferenciarlas como etiquetas interiores de una función.

La cuarta modificación cambió la forma de globalizar funciones y variables. **La nomenclatura "GLOBAL" dejó de existir** para dar paso a "EXPORT". A grosso modo no hay gran diferencia entre ellas y nuestro código debería funcionar perfectamente una vez hagamos el cambio.

Código 8.39: Cambio de GLOBAL por EXPORT

```

1 IF !DEF(ENTITY_INC)
2 ENTITY_INC SET 1
3
4 ;FUNCTIONS
5 EXPORT EM_INIT
6 EXPORT EM_CREATE_PLAYER
7 EXPORT EM_CREATE_ENEMY
8
9 [...]

```

8.6.2 Utilización de Objetos

Esta es una **tarea** que quedó **pendiente de la anterior iteración** y que era esencial para conseguir un producto jugable. La mayor complicación que podemos encontrarnos aquí es el dibujado de pantallas, ya que a parte de los iconos (utilizados para diferenciar los objetos entre sí), es el conseguir mostrar cuántas unidades de cada uno tiene el jugador en ese preciso instante. Por otro lado, el **dibujado de las ventanas quedó reducido al llamado de una función**, pasando por parámetro el ancho y el alto, así que en ese aspecto no deberíamos encontrar problema alguno.

Como ya se indicó en iteraciones previas, el jugador disponía de tres direcciones de memoria RAM, cada una para ir almacenando la cantidad que dispone de cada objeto. Cada vez que el jugador selecciona el objeto que desea utilizar, se comprueba esa variable dinámica, decrementándola en caso de ser distinta de 0. Justo después procedemos a aumentar la vida o el hambre del jugador, según qué caso.

Código 8.40: Comprobación de Objeto Seleccionado

```

1 ; THE PLAYER PRESSED A, so we check at which option he did.
2
3 ld a, [_OPTION]
4 cp 1
5 jp z, rs_draw_inventory_check_apples ; El jugador selecciona la manzana.
6 cp 2
7 jp z, rs_draw_inventory_check_potions ; El jugador selecciona la poción.
8 cp 3
9 jp z, rs_draw_inventory_check_spoptions ; El jugador selecciona la súper ↪
   ↪ poción.
10 jp RS_DRAW_MENU_BOX

```

¿Cómo mostramos el número exacto de unidades de cada objeto de las que disponemos? Si tenemos el número actual de un objeto en concreto, y tenemos el tile numérico inicial (el tile con el dibujo de un 0), lo único que debemos hacer es sumar el número actual al número de tile para hallar el tile final que debemos mostrar:

Código 8.41: Muestreo

```

1  call EM_GET_PLAYER_VARIABLE      ; a = Número actual de manzanas
2  ld d, TILE_NUM_0              ; d = tile 0
3  add d                      ; Tile_0 + número de manzanas
4  inc bc
5  ld d, a
6  call RS_WAIT_MODE_01          ; Modified registers: a
7  ld a, d
8  ld [bc], a                  ; Inserción del resultado en memoria de vídeo.
9  ret

```

El resultado final se puede ver en la siguiente imagen:



Figura 8.21: Inventario

8.6.3 Múltiples Tilesets

Ahora la idea es **introducir nuevos enemigos**. Esto implica añadir más tiles en nuestro tileset (bastantes más). Lamentablemente, la GB es **incapaz de tener en memoria tal cantidad** (recordemos que en nuestro tileset disponemos por el momento tanto de los caracteres alfabéticos, numéricos, sprites del personaje principal y sus animaciones, muros de la mazmorra, árboles, muebles, etc...).

Hay ocasiones en las que muchos de esos tiles no se utilizan para nada y son memoria desperdiciada. Por ejemplo, dentro de la mazmorra no vamos a encontrar nunca tiles de árboles. Por este motivo, tenemos la obligación de diseñar 4 tilesets distintos y un

cargado propio que, en función del estado de juego, cargue uno u otro.

Código 8.42: Múltiples Tilesets

```

1;=====
2; Tiles
3;=====
4Persistent_Tiles:
5    INCLUDE "./TILES/persistent_tiles.z80"
6Persistent_Tiles_End:
7
8Overworld_Tiles:
9    INCLUDE "./TILES/overworld_tiles.z80"
10Overworld_Tiles_End:
11
12Dungeon_Tiles:
13    INCLUDE "./TILES/dungeon_tiles.z80"
14Dungeon_Tiles_End:
15
16Title_Screen_Tiles:
17    INCLUDE "./TILES/title_screen_tiles.z80"
18Title_Screen_Tiles_End:
```

Además, aprovechando que dividimos el tileset principal en varios, podemos añadir más tiles que puedan ayudarnos a decordar los distintos paisajes del juego.

La forma en la que cargamos uno u otro es la siguiente: teniendo en cuenta que siempre tendremos en memoria los tiles del personaje, los números y las letras, nos creamos un **tileset llamado "persistent"**, que se va a mantener constante en todos los estados de juego. Ahora, sabiendo que ese tileset **ocupa un total de N bytes**, podemos **cargar el siguiente a partir de la posición inicial de nuestra VRAM** más ese tamaño N:

Código 8.43: Múltiples Tilesets

```

1RS_GET_NONPERSISTENT_TILES_MA:
2    ld hl, _VRAM           ; Memoria VRAM inicial
3    ld de, Persistent_Tiles_End-Persistent_Tiles ; Tamaño de N bytes
4    add hl, de             ; Dirección resultante en HL
5    ret
```

Un problema con el que podemos llegar a encontrarnos es el siguiente: tenemos más tiles, y nos caben en la VRAM, ¿pero qué pasa si no nos cabe en la memoria ROM? Es decir, al fin y al cabo, todos esos bytes van a parar en la memoria de lectura, en algún momento nos quedaremos sin espacio, ¿no? Si esto sucede se nos mostrará en la terminal el siguiente mensaje:

```
rgblink -o ../PrSacra.gb -m ../obj/PrSacra.map -n ../obj/PrSacra.sym ../obj/a
i_system.o ../obj/animation_manager.o ../obj/audio_system.o ../obj/collision_
manager.o ../obj/entity_manager.o ../obj/game_manager.o ../obj/input_system.o
../obj/main.o ../obj/physics_system.o ../obj/render_system.o ../obj/text.o
error: Unable to place "Animations" (ROM0 section) anywhere
make: *** [Makefile:24: PrSacra] Error 1
```

Figura 8.22: Falta de Memoria ROM

La solución es meterlo todo en ROMX, ese banco de memoria que tenemos disponible y que hasta ahora no habíamos utilizado para nada. Poniendo las directrices de "SECTION", indicando el banco de memoria y poniendo a continuación los bytes a almacenar en ella, ya tendríamos el problema resuelto (al menos de momento).

El segundo problema: el **control de qué tileset está cargado y si hemos cargado los bytes que tocan**. Esto realmente no es un problema como tal, sino un descuido por parte de nosotros, los programadores. Hay que llevar cuidado en todo momento ya que se pueden producir efectos tan adversos como el que se muestra a continuación:

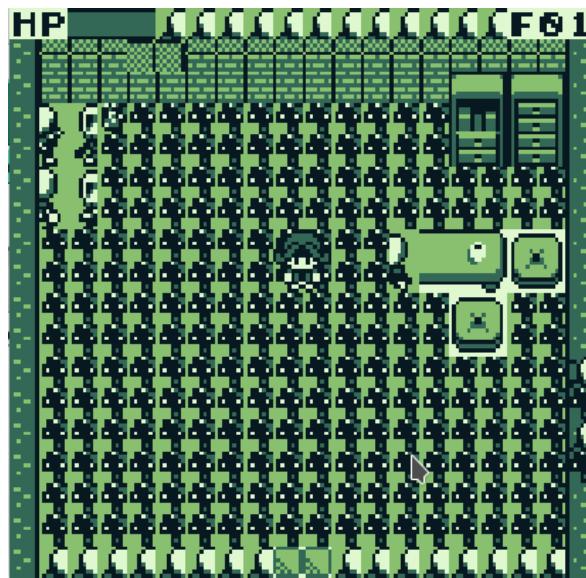


Figura 8.23: Cargado Erróneo de Tileset

El tercer y último problema: si en nuestro tileset inicial tenemos unos números de tile específicos por los cuales permitimos dejar al jugador interaccionar de alguna forma, **¿qué ocurre si tenemos más de un tile con el mismo número?** Por ejemplo, supongamos que el tile de suelo es el 103 en nuestro tileset inicial. Ahora lo dividimos en varios y tenemos que nuestro tile de suelo es el 50. Perfecto, esto se debe a que tenemos menos tiles en general. Pero, lamentablemente, en el otro tileset también tenemos un tile con número 50, y a este último también nos vamos a poder mover. ¿Cómo lo solucionamos? Es cuestión de diseño más que de programación. En este caso colocaremos en ese número otros tiles a los que el

jugador no pueda llegar nunca, o que (por el contrario) también sea un suelo.

Aprovechando que podemos insertar muchos más tiles, nos podemos permitir el lujo de gastar algo de tiempo en el diseño de la pantalla principal del juego, la cual ha quedado así:



Figura 8.24: Pantalla Principal de Juego Definitiva

8.6.4 Inserción de Objetos y Enemigos

Este apartado es el que **más trabajo** ha tenido y el que **más corto va a ser de explicar**. Principalmente teníamos la idea de crear una inteligencia artificial, pero analizando el tiempo del que disponemos y el resto de entregas del mismo género, es preferible tirar de elementos básicos en cualquier RPG como pueden ser la cantidad de vida y el daño de un enemigo.

Con esto en mente, hemos diseñado 3 enemigos más para poder hacer una **dificultad ascendente** según el jugador fuese progresando por los distintos niveles.

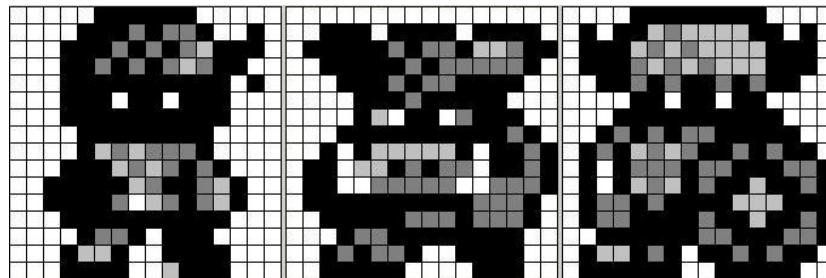


Figura 8.25: Diseño de Nuevos Enemigos

De esta manera, **el primer enemigo tendría más vida pero menos daño que el enemigo inicial, el segundo tendrá la misma vida pero más daño, y el tercero tendrá más vida y más daño**. Así mismo, mostrando cada enemigo en un nivel nuevo, podemos tener esa dificultad escalonada que buscamos.

Lo único que nos queda es calcular en qué sitios meter qué objetos para no hacer impos-

sible el avance en el juego, pero sí que te haga decidir si es rentable o no el hacerte con él. Y lo más importante de todo: procurar que el juego sea divertido.



Figura 8.26: Diseño de Niveles

¿Por qué se ha comido tanto tiempo esta tarea? Porque se necesita de mucho diseño y prueba y error para **comprobar que todo funciona como debe y que el juego no es frustrante ni imposible**. Además, para meter cada enemigo u objeto se necesita calcular de manera previa las coordenadas X e Y que van a tener, así como los tiles en los que se va a encontrar.

8.6.5 Conclusión

Pese a la situación en la que nos encontramos la mayoría de personas actualmente, me puedo dar con un canto en los dientes al haber conseguido sacar el trabajo adelante. La motivación ha sido un factor clave a la hora de seguir con el desarrollo del proyecto.

Me quedan dos iteraciones más a lo sumo y no tengo duda alguna de que el proyecto está en buenas condiciones. Como dije en la conclusión de la iteración 4, las tareas que faltan son secundarias, pero van a servir para dejar un producto completo y cerrado, el cual se pueda meter a un cartucho físico y vender.

8.7 Iteración 6 - Mejoras Gráficas, Gameplay y Efectos de Sonido

Esta iteración (de unas dos semanas de duración) se ha dedicado a la mejora tanto de elementos gráficos pertenecientes al HUD, como de inteligencia artificial y música. Al ser la penúltima iteración antes de la entrega del proyecto, a parte de lo ya nombrado, otro objetivo ha sido el de arreglar distintos errores que hacía que el juego se congelase en distintas ocasiones.

8.7.1 Música y Efectos de Sonido

Ya habíamos visto en la iteración 1, podemos hacer sonar un "Do Re Mi Fa Sol La Si" constante a través del canal 1. Sin embargo, es un sistema que nos limita mucho a la hora de ser capaces de implementar música o distintos efectos de sonido según que evento se dé dentro del juego.

Vamos a utilizar ahora todos los canales de audio (a excepción del tercero), y además no vamos a utilizar parámetros fijos de frecuencias, longitud, ciclo de trabajo o envolventes. Cada nota tendrá sus 3 o 4 bytes (dependiendo del canal) que nos ayudarán a tener una mayor variedad de resultados.

Dicho esto, lo primero que he modificado ha sido la manera en la que se toca una nota. Como vamos a hacer uso de los canales 1, 2 y 4, he creado los tres métodos distintos en los que, dada una dirección de memoria, coge los 3 o 4 bytes contiguos para meterlos en sus registros correspondientes:

Código 8.44: Método de Sonido en Canal 1

```

1 AS_PLAY_NOTE_CHANNEL_1:
2
3     ld a, [hl]
4     ld [rNR10], a
5     inc hl
6     ld a, [hl]
7     ld [rNR11], a
8     inc hl
9     ld a, [hl]
10    ld [rNR12], a
11    inc hl
12    ld a, [hl]
13    ld [rNR13], a
14    inc hl
15    ld a, [hl]
16    ld [rNR14], a
17
18    ret

```

Con esto, para generar los efectos de sonido, solamente tendríamos que hacerle la llamada en cualquier otra parte del código con la dirección de memoria correspondiente en HL. Aquí el tempo o la nota actual a tocar son completamente innecesarias ya que queremos que se

produzcan de forma inmediata.

Para hacer la banda sonora del juego, por otro lado, sí que vamos a necesitar de un sistema algo más complejo. Lo que tomábamos antes como "nota actual" era el número a sumar a una dirección de memoria específica. En esa dirección de memoria resultante, es donde realmente estaba la nota a tocar. ¿Qué ocurre si para llegar a la siguiente nota necesitamos sumar un número de más de 8 bits? ¿Y más de 16 bits? La escalabilidad es mínima, por lo que tendremos que cambiar lo que la "nota" simbolizaba a, simple y llanamente, una dirección de memoria fija. Vamos a seguir con el ejemplo del canal 1:

Código 8.45: Método de Música en Canal 1

```

1 AS_PLAY_MUSIC_CHANNEL_1:
2
3     ld a, [_NOTA] ; Cargamos la dirección de memoria de la nota actual a tocar
4     ld l, a
5     ld a, [_NOTA+1]
6     ld h, a
7     ld a, [hl]
8
9     cp _KEEPNOTE ; Comprobamos si debemos cambiar de nota, hacer un silencio←
10    ↘ o mantenerla.
11     jr z, as_note_end_keep
12
13     call AS_PLAY_NOTE_CHANNEL_1
14
15 as_note_end_keep:
16
17     ; Mantenemos la nota.
18     [...]
19
20     ; Comprobamos si era la última nota. Reseteamos o continuamos a la ←
21     ↘ siguiente
22     [...]
23     jr z, as_note_end_reset
24
25     ; Actualizamos la dirección de memoria de la nota a la siguiente.
26     [...]
27
28     ret
29
30 as_note_end_reset:
31
32     ; Reseteamos la dirección de memoria de la nota a la primera.
33     [...]
34
35     ret

```

El único parámetro de entrada que le tenemos que pasar a la función es la dirección de

memoria final de la canción que está sonando en BC. Al ser música de fondo que suena constantemente en el juego, el único momento en el que nos vamos a preocupar de cambiar la nota inicial de la canción es en los cambios de estado de juego (pantalla de título, mazmorra, etc...).

Al igual que hacíamos con los gráficos, las canciones y efectos de sonido también los vamos a guardar en el banco de memoria secundario ROMX:

Código 8.46: Música y SFX en ROMX

```

1 TITLE_SCREEN1:
2   DB $00, $00, $00, _SILENCE, _SILENCE
3   DB $00, $00, $00, _SILENCE, _SILENCE
4   DB $00, $81, $4B, _A, _OCT6_NR
5   DB _KEEPNOTE
6   DB $00, $8A, $47, _G, _OCT6_NR
7   DB $00, $00, $00, _SILENCE, _SILENCE
8   DB $00, $81, $4B, _C, _OCT7_NR
9   DB _KEEPNOTE
10  DB _KEEPNOTE
11  [...]
12 END_TITLE_SCREEN1:
13
14 [...]
15
16 OBJECT_SOUND:::
17   DB $00, $C0, $43, $44, $87
18 TEXT_OBJECT_SOUND:::
19
20 [...]
21
22 ; CHANNEL 4
23
24 DAMAGE_SOUND:::
25   DB $00, $F1, $61, $C0
26 END_DAMAGE_SOUND:::

```

¿Y cómo sabemos qué combinación de bytes colocar para que suene un sonido en concreto? Aquí es donde entra la herramienta de GBSound Sample Generator, como bien muestro en la sección de **herramientas utilizadas**.

Como podemos observar, en los bytes de **_TITLE_SCREEN1** disponemos de distintas nomenclaturas como **_SILENCE** o **_OCT6_NR**. Simplemente se tratan de distintas frecuencias que nos dan las octavas y notas que queremos. Por ejemplo, un **_A** seguido de un **_OCT6_NR** nos daría la nota La en Octava 6. Las últimas dos letras se utilizan para activar o desactivar el bit que impide que la nota se mantenga constante a lo largo del tiempo, sin llegar a efectuar su estado de relajación.

Una pregunta que nos podemos llegar a hacer es la siguiente: ¿Por qué no analizar el código de juegos como Pokémon Rojo para saber cómo tienen programado el sistema de audio? Aquí entramos en temas de tiempo y necesidad. En este proyecto el realizar un sistema de

audio completamente funcional es algo que se quedaba en un plano muy secundario. Como partíamos de una base (lo realizado en la iteración 1) solo se ha necesitado comprobar cuáles eran las necesidades y efectuar las respectivas modificaciones. Obviamente, esto no quiere decir que el analizado de distintas entregas sea una pérdida de tiempo. Seguramente haciendo esto último podamos llegar a tener un sistema mucho más complejo, el cual nos permita disponer de más funcionalidades. Aún así lo reitero: no es el principal objetivo de este proyecto.

Todo lo que necesitamos ahora es imaginación y gusto musical con el que componer una canción y transcribirla a bytes. En el caso de que nos falte alguna de las dos cualidades, siempre podemos recurrir a páginas web como Fiverr en la que podremos encontrar distintos artistas que nos van a hacer un buen trabajo. Marc Davis fue, en este caso, el artista que compuso las canciones de este proyecto. Todo lo que él necesitaba era alguna captura del proyecto y referentes sobre los que inspirarse.

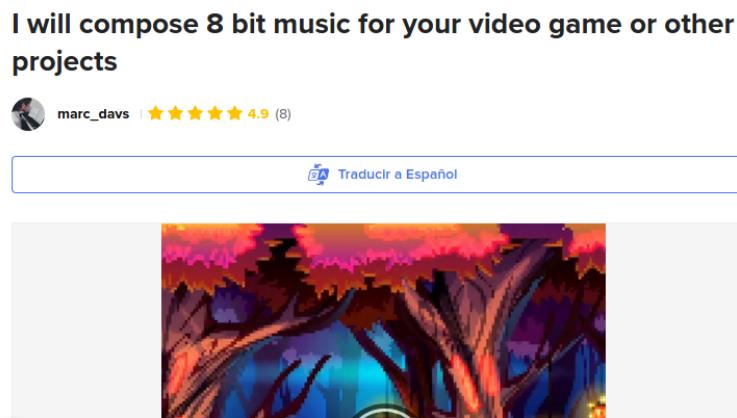


Figura 8.27: Marc Davis - Compositor Musical en Fiverr

Este es un trabajo de ingeniería y, por tanto, no es nuestro trabajo el realizar la composición de una banda sonora. Es por ello que, en casos como estos, podemos hacer uso de fuentes externas (siempre y cuando tengamos en cuenta los derechos de uso).

8.7.2 Arreglo Visual en Ventanas Emergentes

Las ventanas emergentes que se realizaron en la iteración anterior no quedaron del todo usables debido a un problema visual en el que los sprites, al estar siempre por encima de las dos pantallas de fondo, ocultaban parte de los textos:

Para solucionar este problema, lo que debemos hacer es comprobar las coordenadas en tiles de cada sprite, tanto del jugador, como enemigos y objetos. Por ejemplo, en la imagen 8.28 tendríamos que comprobar que la coordenada Y fuese menor de N, mientras que (en este caso) la coordenada X nos es indiferente.

Con esto en mente, ya sabemos que la función que hagamos va a necesitar dos parámetros: las coordenadas X e Y mínimas por las cuales, si un sprite está por encima de ellas, lo tendremos que esconder.



Figura 8.28: Problema Visual Ventanas Emergentes

Código 8.47: Escondido de Sprites Según su Tile

```

1 RS_HIDE_SPRITES:
2
3     xor a
4     call PS_SET_UPDATE_ABSOLUTES ; Impedimos que se actualicen en el render.
5
6     call EM_GET_ENEMIES_ARRAY
7     call rs_hide_sprites_start    ; Escondemos los enemigos
8
9     call EM_GET_BLOCKS_ARRAY
10    call rs_hide_sprites_start    ; Escondemos los objetos
11
12    jp RS_DRAW                 ; Una vez terminado, redibujamos todo.
13
14 rs_hide_sprites_start:
15
16    or a                      ; Comprobamos si hay al menos 1 entidad.
17    ret z
18
19    [...]
20
21 rs_hide_sprites_loop:
22
23    ld a, [hl]                 ; HL = Y component
24    sub c                      ; E_Y - Minimum_Y
25    jr c, rs_hide_sprites_next ; Si el resultado no da acarreo, significa ←
                                ; → que la entidad no está por encima del mínimo Y.
26
27    inc hl                     ; HL = X component
28    ld a, [hl]
29    sub b                      ; E_X - Minimum_X

```

```

30 jr c, rs_hide_sprites_next ; Si el resultado no da acarreo, significa ←
    ↪ que la entidad no está por encima del mínimo X.
31
32 ; Si las coordenadas son mayores a los mínimos, lo escondemos. Para ello ←
    ↪ ponemos a 0 sus coordenadas en el DMA.
33
34 xor a
35 ld [hl], a          ; X = 0
36 dec hl
37 ld [hl], a          ; Y = 0
38
39 rs_hide_sprites_next:
40
41 ; Sumamos a la dirección HL actual un número específico que nos de el ←
    ↪ primer valor de la siguiente entidad.
42 [...]
43
44 rs_hide_sprites_next_entity:
45
46 ; Comprobamos si nos quedan más entidades. En caso afirmativo, actualizamos←
    ↪ la dirección HL.
47 [...]

```

En el código expuesto no se muestra cómo se esconde al jugador. Esto se debe a que, mientras que a los enemigos y objetos se les actualizan las coordenadas de pantalla (o coordenadas absolutas) por ciclo de ejecución, al jugador no. Quiere decir que, a las dos primeros, con poner en el DMA los valores de las coordenadas a 0 van a dejar de aparecer en pantalla y una vez terminado todo, el juego va a coger las coordenadas guardadas en cada entidad y las va a volver a poner en la OAM. Si hiciésemos esto con el jugador veríamos que los sprites se esconden, pero no se vuelven a poner en su sitio. Para poder esconderlo tendremos que manipular la OAM de forma directa:

Código 8.48: Escondido de Sprites Según su Tile

```

1 RS_HIDE_PLAYER_SPRITE:
2
3 ld de, _OAMRAM      ; Dirección inicial de la OAM
4 ld a, 4
5 ld [_COUNT2], a
6 ld a, 1          ; Número de entidades a comprobar
7
8 jr rs_hide_sprites_start ; Llamada al método anterior.

```

El resultado lo podemos ver en el siguiente par de imágenes:

8.7.3 Mejoras en el Flujo de Juego

Las mejoras que se van a implementar son dos: el radio de visión de los enemigos y su velocidad.

Implementar el rango de visión es sencillo: dado un valor máximo, comprobamos si las coor-



Figura 8.29: Arreglo Visual Ventanas Emergentes

denadas X e Y del jugador se encuentran dentro de ese radio respecto al enemigo. Como las comprobaciones no van solamente en una dirección, tendremos primero que hacer la resta de ambos y ver si da acarreo. En caso contrario le restamos el radio de visión máximo. Si el resultado ha vuelto ha dar acarreo o 0, significa que está dentro del rango de visión en ese eje.

Código 8.49: Rango de Visión en los Enemigos

```

1 [...]  

2  

3 ld b, a           ; B = Coordenada X del jugador  

4 [...]  

5  

6 ld d, a           ; D = Coordenada X del enemigo  

7 sub b  

8 jr c, ai_see_player_swap_x    ; Si hay carry significa que la X del ←  

→ jugador es mayor.  

9  

10 ; Si la coordenada del jugador no es mayor, comprobamos si está dentro del ←  

→ rango.  

11 sub SIGHT_RADIUS  

12 jr c, ai_see_player_y    ; Si está dentro del rango pasamos a comprobar la ←  

→ Y.  

13 jr z, ai_see_player_y  

14  

15 jr ai_see_player_unsuccess ; Si no está dentro del rango devolvemos falso←  

→ .  

16  

17 [...]  

18  

19 ai_see_player_swap_x:  

20  

21 ; Si la coordenada X del jugador es mayor, llegamos aquí. Realizamos la ←  

→ operación inversa y le restamos el rango de visión.

```

```

23
24 ld a, b
25 sub d
26 sub SIGHT_RADIUS
27 jr c, ai_see_player_y
28 jr z, ai_see_player_y
29 jr ai_see_player_unsuccess

```

El método nos devuelve falso si el enemigo no es capaz de ver al jugador, y 1 en caso contrario. De esta manera no vamos a tener enemigos persiguiéndonos de manera constante independientemente de donde nos encontremos.

La segunda mejora da mucho juego, y se trata de la velocidad de cada enemigo. Esta velocidad va a ser distinta para cada uno de ellos, y va a dotar al juego de más posibilidades a la hora que el jugador tome una decisión u otra.

Esto es muy sencillo y lo hemos hecho ya en algún otro momento a lo largo del desarrollo. Lo único que tenemos que hacer es añadir un byte más al tamaño de los enemigos que simbolice esta velocidad. Si queremos que el enemigo se mueva un tercio del jugador, lo que hará será moverse tres casillas y luego dejar pasar un turno. Con esto en mente, por ejemplo, podríamos ponerle a ese enemigo que su velocidad es 3. Por cada turno que pase, esa velocidad se decrementará y al llegar a 0 se reiniciará y es cuando hará el parón. Esto quiere decir que, la velocidad de este enemigo en específico, es de 3/4, ya que se mueve 3 de cada 4 casillas.

Como solamente estamos usando un byte, este mismo va a ser el que nos aporte la información del valor inicial y el valor actual. Si un byte son 8 bits, podemos usar los 4 primeros para uno y los 4 siguientes para otro. Esto nos impide tener velocidades mayores de 15, pero creo que es completamente innecesario.

La comprobación de este valor, finalmente, lo haremos en la toma de decisiones del enemigo. Si no puede atacar al jugador, comprobaremos si se puede mover según su velocidad.

Código 8.50: Velocidad de los Enemigos

```

1 ld e, en_velocity
2 call EM_GET_ENEMY_VARIABLE ; Obtenemos velocidad del enemigo
3 ld d, a ; La guardamos en D y la decrementamos
4 dec a
5 jr c, .decide_move_keep ; Si da acarreo (== 0) podemos movernos
6
7 ld a, d ; Obtenemos los 4 bits de la velocidad actual
8 and %00001111
9 dec a ; Decrementamos el valor
10 jr nz, .decide_move_update_velocity ; Si no da 0 significa que podemos ↪
    ↩ movernos
11
12 ; En caso contrario tenemos que resetear la velocidad
13
14 ld a, d

```

```
15    and %11110000      ; Obtenemos los 4 bits que nos da el valor por ↪
     ↪ defecto.
16    ld d, a           ; Lo guardamos en D
17    swap a            ; Hacemos un swap al registro a (tendremos %00001111)
18    or d              ; Aplicamos una operación de tipo OR
19    ld [hl], a         ; Guardamos el valor resultante
20
21    jp .decide_not_m ; Impedimos el movimiento.
```

8.7.4 Conclusión

Esta iteración ha servido para dejar algunas de las tareas secundarias terminadas. Pienso que el trabajo en estas dos semanas ha sido bueno y productivo, y que he conseguido dejar un producto cerrado y bien acabado. El trabajo que queda de aquí en adelante (el poco tiempo que me queda de aquí a la entrega) se va a enfocar en meter más niveles y alguna canción más. En cuanto a inteligencia artificial, por ejemplo, no da tiempo a hacer nada demasiado complejo.

9 Anexo

9.1 CPU

Como ya has podido comprobar, la CPU que la Game Boy utiliza es la **Sharp LR35902**, variante de la **Zilog Z80 e Intel 8080**. Es un procesador de **8 bits** que puede realizar **cálculos de 16 bits y manejar hasta 64Kb de memoria externa** para el control del Joypad, el LCD o el control de sonido. A pesar de ello, se queda lejos de poseer todas las características de la Zilog Z80. Las operaciones de 16 bits que posee son muy reducidas, pues no posee instrucciones con prefijos **ED**, **FD** ni **DD**¹. Esto implica que **no disponemos** tampoco de los registros **IX**.

9.1.1 Registros

Para poder empezar a programar en ensamblador **debemos saber qué registros tenemos y cómo usar cada uno de ellos**. Los nombres de los registros de los que disponemos son A, F, B, C, D, E, H, y L, los cuales pueden contener **1 byte** de información. También **pueden ser agrupados** en pares de forma que puedan contener **2 bytes**: AF, BC, DE y HL. También están los registros SP y PC, los cuales tienen una **función específica** dentro de la CPU.

En primer lugar tenemos el registro **A**, también denominado **Acumulador**. Es donde va a estar la mayor parte de la información procesada. Es un registro al que se le puede **introducir valores de forma directa**, al igual que B, C, D, E, H y L.

Los registros **B** y **C** se suelen usar para **contadores** y los registros **D** y **E** como un par de 2 bytes para almacenar **direcciones de memoria** sobre las que copiar distintos datos. Obviamente, sus usos no están limitados a las dos que acabo de mencionar, depende de lo que el programador vea conveniente en cada momento.

El registro **F** (o **Flags**) es donde se almacena en todo momento el **estado del procesador**. Es un registro de **solo lectura**, aunque puede utilizarse junto a A para distintas operaciones. Se suele usar a la hora de **comprobar el resultado de la instrucción anterior**. En la siguiente tabla puedes ver la función de cada uno de los bits:

Los registros **H** y **L** se utilizan para **acceso indirecto a una dirección de memoria**. Con acceso indirecto nos referimos al valor de 16 bits que contiene la dirección HL. Es muy útil a la hora de recorrer los valores de un determinado array.

Los registros **SP (Stack Pointer)** se utilizan para **conocer la posición de memoria**

¹Denominados “Undocumented opcodes”, los cuales no están en la lista oficial del Zilog.

Flags				
Bit	Nombre	0	1	Definición
7	zf	Z	NZ	Flag Zero
6	n	-	-	Flag de suma/resta
5	h	-	-	Flag de semi acarreo
4	cy	C	NC	Flag de acarreo
3-0	-	-	-	Sin uso.

Tabla 9.1: Función de los bits del registro F o Flags

desde la cual se hizo una llamada. Es decir, cuando hacemos una instrucción “call”, la pila aumenta, y al hacer un “ret”, disminuye de tamaño.

Y por último, los registros **PC (Program Counter)**, los cuales indican la dirección de memoria de la **próxima instrucción** que se va a ejecutar.

9.1.2 Instrucciones

No voy a entrar en detalle a cerca de las instrucciones que existen, porque para estos casos lo mejor es visitar cualquier tabla con todos los opcodes² que tiene la CPU (en la bibliografía hay un enlace a una de ellas que me ha resultado muy útil).

A grosso modo, las instrucciones las podemos calificar de la siguiente manera:

- **Operaciones de carga:** LD, PUSH, POP...
- **Manipulación de bits:** BIT, SET y RES.
- **Operaciones aritméticas de 8 bits:** AND, OR, XOR, ADD...
- **Operaciones aritméticas de 16 bits:** RRA, RLA, RRC, SLA...
- **Operaciones de salto:** JP, JR, RET, RETI y CALL.
- **Definiciones de espacio:** DB, DS, DW...
- **Operaciones de uso general:** HALT, DI, EI, STOP...

Para una **información más detallada** de lo que hace cada operación lo mejor es consultar el **manual oficial de la CPU** o similares. En la bibliografía podrás encontrarlos.

²También conocido como **Operation Code**. Es una porción de código máquina que especifica la instrucción a realizar.

9.2 Mapa de memoria

Como hemos comentado con anterioridad, la Game Boy nos permite direccionar directamente un total de **64Kb de memoria** (esto es, en hexadecimal, desde la posición \$0000 hasta la \$FFFF). Esta memoria está **dividida en distintos bloques**, mapeados por los diseñadores de aquel entonces, dejando a la **ROM con dos bloques de 16Kb, y un bloque de 8Kb para la RAM**. Aún así, estos espacios de memoria eran insuficientes para algunos videojuegos, con lo que se pasó a emplear la técnica del ***Banking***. Esta técnica consistía en dividir una vez más la ROM en distintos bloques, cada uno independiente del otro, de forma que siempre tendríamos un **bloque fijo** de 16Kb donde almacenaríamos toda la **lógica del juego** y, mediante ciertas instrucciones, **cambiar el segundo bloque** de 16Kb por otro de igual tamaño según las necesidades del programador.

En la siguiente tabla podemos ver los distintos bloques ya mencionados:

Mapa de memoria		
Inicio	Final	Descripción
0000	3FFF	Banco 00, 16Kb ROM
4000	7FFF	Banco 01 - NN, 16Kb ROM
8000	9FFF	8Kb Video RAM
A000	BFFF	8Kb RAM externa
C000	CFFF	Banco 00, 4Kb RAM interna
D000	DFFF	Banco 01 - NN, 4Kb RAM interna
E000	FDFF	Espejo de la RAM interna
FE00	FE9F	OAM
FEAO	FEFF	No usable
FF00	FF7F	Registros Entrada/Salida
FF80	FFFE	High RAM
FFFF	FFFF	Registro de activación de interrupciones

Tabla 9.2: Mapa de Memoria de la Game Boy

Como podemos ver, los dos primeros bloques (**\$0000 - \$7FFF**) se utilizan para acceder a la ROM del cartucho. La mayor parte del código de nuestro juego se va a almacenar en el primer bloque de 16Kb, llamado **HOME BANK**. Este área de memoria es siempre accesible, y contiene la cabecera de nuestro videojuego (como veremos más adelante). El otro bloque de 16Kb es el llamado **PAGED BANK**, debido a que es un bloque intercambiable o paginado. Dependiendo de qué bloque paginemos, podremos o no acceder a distintos bancos de memoria.

El siguiente bloque de memoria es la **Video RAM (\$8000 - \$9FFF)**, la cual no funciona como una tira de bytes donde cada uno de ellos representa un pixel distinto de la pantalla (como sí sucede, por ejemplo, con *Amstrad*). La VRAM, en este caso, contiene dos bloques fácilmente diferenciables: una sección para los **tiles**, y otra con la que se define el fondo (**Background Map o BGMAP**), el cual es un array de 32x32 bytes. Es decir, en el BGMAP lo que tendremos, simplemente, son números que representen qué tile queremos en cada sitio.

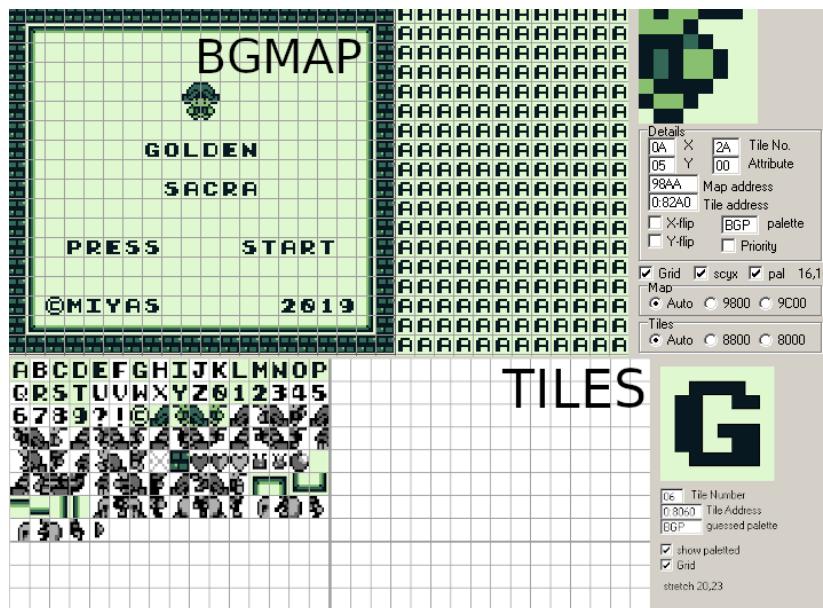


Figura 9.1: Visualización del VRAM

El área que viene a continuación es la **RAM**. Primero tenemos la sección que pertenece al **cartucho (\$A000 - \$BFFF)**, por lo que puede (o no) que lo tengamos disponible. No es un requisito que los cartuchos tengan este espacio de RAM, pero de hacerlo siempre nos viene bien. Si no estás seguro de disponer de él, mejor no lo uses. Luego está la **RAM interna o Work RAM (\$C000 - \$DFFF)**, donde podremos almacenar los valores que necesitemos mantener a lo largo de los distintos ciclos de cómputo.

El **espejo de RAM (\$E000 - \$FDFF)** es un espacio de memoria desperdiciado. Es altamente recomendado **no guardar nada aquí**.

Ahora llegamos a la **OAM (\$FE00 - \$FE9F)**, también conocido como Sprite RAM. Aquí es donde vamos a guardar todos nuestros sprites.

Con el área de **HRAM (\$FF80 - \$FFFE)** veremos más adelante que se le puede dar buen uso. Y finalmente, el último byte de todos (**\$FFFF**), es la **activación de interrupciones**.

9.3 Organización de la ROM

Una pregunta que nos deberíamos hacer siempre es: **¿dónde debemos programar?** Está claro que nuestro código se va a almacenar en la ROM pero, ¿en cualquier sitio? La respuesta es no, puesto que tiene una organización específica que, si no se respeta, nuestro juego puede que no llegue ni tan siquiera a ejecutar.

Lo mejor es definir en una tabla las distintas posiciones de memoria en los que está compuesta la ROM y que no debemos pasar por alto:

Esquema Memoria ROM		
Inicio	Final	Descripción
00	100	Vectores de interrupción.
100	103	Entrada a la ejecución del programa.
104	14E	Cabecera del cartucho.
14F	3FFF	Lógica del programa.
4000	7FFF	Segundo banco. Intercambiable.

Tabla 9.3: Esquema Memoria ROM

El rango **\$0000-\$0100** son los llamados **vectores de interrupción**. Según vayamos necesitándolo, veremos que la Game Boy dispone de distintas **funciones** las cuales pueden ser **llamadas en cualquier momento del programa**.

La **cabecera del cartucho** (**\$0104 - \$014E**) se puede descomponer mucho más, pero lo veremos en profundidad cuando comencemos a programar, por lo que no es necesario explicarlo aquí.

9.4 Inputs

A la hora de controlar cuándo se han pulsado los botones de la Game Boy tenemos un único registro (\$FF00) que contiene la información de los diferentes 8 botones. Este byte funciona como una especie de matriz:

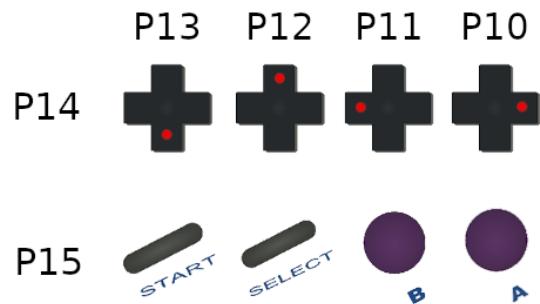


Figura 9.2: Distribución del PAD

La nomenclatura P1N que se ha seguido en la imagen no significa otra cosa que “Player 1” seguido del número del bit del registro. De esta forma, si quisiésemos guardarnos los valores de los botones A, B, Start o Select pondríamos el bit 5 con valor 1 y el bit 4 con valor 0.

Hay que tener en cuenta que los valores que nos devuelve este registro están al revés de como solemos pensar. Con esta afirmación me refiero a que si un botón está pulsado, el valor que aparecerá es 0 y no 1. No es un problema muy serio a tener en cuenta ya que podemos hacer un complementario para tener los valores de la forma habitual (1 = si, 0 = no).

9.5 Interrupciones

Las interrupciones, como su nombre indica, son **procesos por lo que la CPU deja en pausa la ejecución** de la lógica del juego para saltar a una posición de memoria determinada, también llamada vector. Son útiles ya que, entre otros, se pueden dar en cualquier parte de nuestro código.

La mayoría de las CPU's tienen un “**master flag**” para las interrupciones. La Z80 que lleva la Game Boy no es distinta, pero hay registros adicionales específicos de la consola. El registro maestro es el **IME (Interrupt Master Enable)** el cual es **desactivado con el opcode DI**, con lo que no se daría ninguna interrupción. Por el contrario, **se activa con EI**, produciéndose **todas las interrupciones** que nosotros especifiquemos **en el registro IE (Interrupt Enable, \$FFFF)**. De manera complementaria, tenemos el **registro IF (Interrupt Flag, \$FF0F)**, el cual indica qué interrupción se ha activado.

El **proceso** que se da cada vez que se da una interrupción es el siguiente:

1. Cuando una interrupción se produce, su bit del registro IF se pone a 1.
2. Si el IME y el bit representativo de la interrupción en el registro IE están activos, se producen los siguientes tres pasos.
3. El registro IME se desactiva para prevenir otra interrupción mientras se procesa ya una.
4. El registro PC (Program Counter) se pushea al stack.
5. Se hace un salto al vector de la interrupción.

Hay **5 tipos** de interrupciones:

Interrupción	Dirección de memoria
Vertical Blank	0040
Estado del LCD	0048
Temporizador	0050
Serial Link	0058
Pulsación del Joypad	0060

Tabla 9.4: Interrupciones de la Game Boy

- **V-Blank:** Se produce alrededor de 60 veces por segundo. Ocurre cuando se redibuja la última línea de la pantalla, que es cuando el hardware no esta haciendo uso de la memoria de vídeo. Dura alrededor de 1'1ms.
- **Estado del LCD:** Hay varias razones por las que esta interrupcion se puede dar, descritas en el registro STAT (\$FF40).
- **Temporizador:** Se produce cuando el registro TIMA (\$FF05) cambia su valor de \$FF a \$00.
- **Serial Link:** Se produce cuando una trasferencia de datos se ha completado a través del puerto del cable link.
- **Pulsación del Joypad:** Se produce cuando el usuario presiona cualquier botón. Debido al efecto “*bouncing*”³, la parte del software debería tener presente que esta interrupción puede producirse varias veces de forma consecutiva.

³El “*bouncing*” es aquel efecto que, a nivel microscópico, cuando se pulsa un botón, los contactos rebotan. Esto genera pequeños saltos de señal que el hardware lee.

9.6 GPU

La **GPU** (**G**raphics **P**rocessing **U**nit), tambien referenciada en la Game Boy como **PPU** (**P**icture **P**rocessing **U**nit), es uno de los principales componentes junto a la CPU. Es la principal vía para mostrar gráficos por pantalla, y gran parte del trabajo del procesador se basa en generarlos para pasárselos a esta.

Tiene las siguientes **especificaciones**:

- **2 fondos de pantalla**, una principal y otra secundaria utilizada de ventana.
- **40 sprites**, máximo 10 por línea.
- Tiles de tamaño 8x8.
- **4 modos** de dibujado.
- 456 ciclos por línea, 70224 por frame.
- **4 colores** en tonos de gris (2-bit).
- LCD de 160x144 píxeles (20x18 tiles).

Los **4 modos de dibujado** son los siguientes:

- **Modo 0 - \$00:** H-Blank.
- **Modo 1 - \$01:** V-Blank.
- **Modo 2 - \$10:** Escaneo de OAM.
- **Modo 3 - \$11:** H-Draw. Tiempo en el que se transfieren datos al driver del LCD.

Los modos 0, 2 y 3 no ocurren durante el V-Blank, solo durante el V-Draw. En la siguiente imagen puedes ver cuando se da cada una, a excepción del H-Draw que se da siempre que se dibuje un pixel por pantalla:

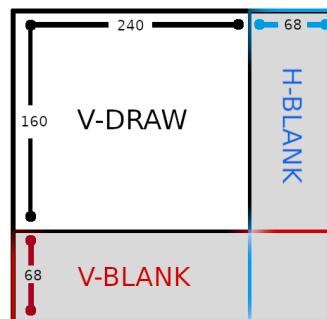


Figura 9.3: Modos de dibujado de la GPU

9.6.1 LCD control

El LCD control se encuentra en la dirección de memoria \$FF40, y es el encargado de controlar todo lo que se dibuja por pantalla. Sus bits tienen los siguientes usos:

- **Bit 7:** Apaga o enciende el LCD.
- **Bit 6:** Selección de la posición del fondo de pantalla secundaria (0 = \$9800-\$9BFF, 1 = \$9C00-\$9FFF).
- **Bit 5:** Dibuja o esconde la pantalla secundaria.
- **Bit 4:** Selecciona la posición de los tiles que componen las 2 pantallas (0 = \$8800-\$97FF, 1 = \$8000-\$8FFF).
- **Bit 3:** Selecciona la posición de la pantalla principal (0 = \$9800 - \$9BFF, 1 = \$9C00 - \$9FFF).
- **Bit 2:** Tamaño de los sprites (0 = 8x8, 1 = 8x16). En caso de activarlo, cada sprite usará su tile asignado más el siguiente en memoria.
- **Bit 1:** Dibuja o no los sprites (0 = no, 1 = sí).
- **Bit 0:** Dibuja o no los dos fondos de pantalla (0 = no, 1 = sí).

9.6.2 LCD STAT

Como hemos comentado previamente, el STAT es el registro que nos muestra en todo momento el estado del LCD.

Sus bits funcionan de la siguiente manera:

- **Bit 7:** Sin uso.
- **Bit 6:** Interrupción cuando LYC = LY⁴.
- **Bit 5:** Modo 10 (OAM Scan).
- **Bit 4:** Modo 01 (V-Blank).
- **Bit 3:** Modo 00 (H-Blank).
- **Bit 2:** Flag de coincidencia. Se activa si LYC = LY.
- **Bit 0-1:** Modo de dibujado del LCD (Ver apartado 9.6).

Los bits que van del 3 al 6 selecciona las condiciones por la que la interrupción STAT pueda ser dada. Seleccionar más de una de manera simultánea puede generar un bloqueo.

⁴LY es la línea por la que el scanline está dibujando y LYC es la seleccionada por el usuario.

9.6.3 LCD Color Display (Solamente CGB)

En caso de utilizar una GBC, la pantalla LCD puede llegar a mostrar un total de 32 tonos en cada canal de color RGB, dando como resultado un total de 32768 colores diferentes. Las paletas constan solamente de 4 colores, escogidos del total ya mencionado.

Igual que ocurre con el modo DMG, podemos utilizar paletas independientes para el BG y el OBJ. Sin embargo, como el canal OBJ dispone de un canal transparente, este solamente tendrá 3 colores en total.

9.6.3.1 LCD Palettes

Cada paleta del modo CGB son representadas mediante 2 bytes, en los cuales cada canal de color RGB viene a su vez representado por 5 bits. El bit más alto queda en desuso.

Los datos se escriben mediante los registros de especificación y los registros de escritura. Los 6 bits del registro de especificación, como su nombre indica, precisan la dirección de escritura en la que van a acabar guardados los datos que sean introducidos en el registro de escritura. Si el bit más alto del registro de especificación se pone a 1, el registro de escritura se incrementará de forma automática (la dirección a incrementar es la especificada en los 6 bits bajos).

Los registros de especificación a utilizar para las paletas de color en modo CGB son \$FF68 y \$FF6A, y los registros de escritura \$FF69 y \$FF6B, siendo utilizadas para las paletas de BG y OBJ respectivamente.

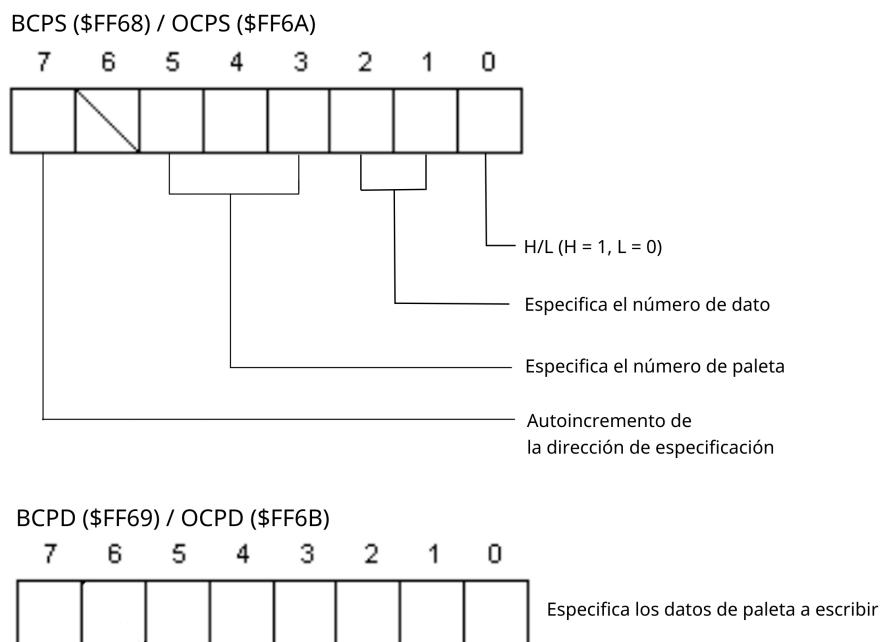


Figura 9.4: Representación Gráfica de los Registros de Especificación y Escritura en Modo CGB

9.7 Sonido

La Game Boy dispone de 4 canales diferentes, los cuales son programables e independientes los unos de los otros:

- **Canal 1:** Tono⁵ y portamento⁶. Se trata de una onda cuadrada a la que le podemos modificar el régimen de trabajo.
- **Canal 2:** Tono. Al igual que el Canal 1, es una onda cuadrada de régimen de trabajo modificable.
- **Canal 3:** Onda programable. Tiene una RAM de onda de 32 valores (4 bits). No dispone de envolvente de amplitud, a diferencia de los otros 3 canales.
- **Canal 4:** Ruido blanco.

Vamos a explicar varios conceptos teóricos de manera que podamos entenderlas sin necesidad de consultar otros documentos:

El régimen de trabajo o ciclo útil se refiere al porcentaje de tiempo que dura el nivel alto respecto al nivel bajo de una onda cuadrada.

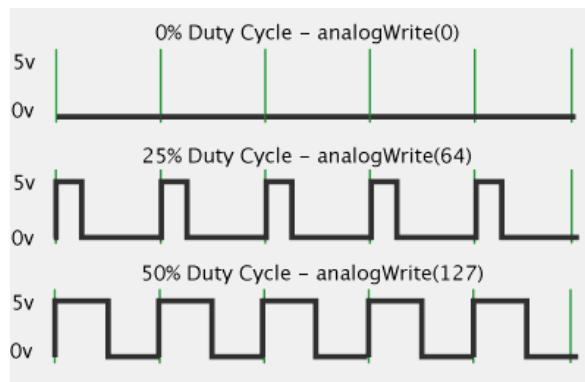


Figura 9.5: Ejemplo Gráfico de Ciclo Útil de una Onda

Por otro lado, la envolvente de amplitud es un gráfico en el que se muestra la evolución a lo largo del tiempo del ya nombrado atributo correspondiente a una onda. Esto es, desde el momento de su emisión, hasta su desaparición.

Esta envolvente nos da la información de si el tiempo de ataque es abrupto, cuánto tiempo se mantiene el sonido, y si su extinción se va produciendo poco a poco o de manera inmediata.

A continuación vamos a ver, como no, los registros que nos van a hacer falta del sistema de sonido:

⁵El tono o altura del sonido es el atributo que nos hace percibir los sonidos más agudos o graves.

⁶El portamento es la transición entre sonidos graves y agudos para impedir percibir una discontinuidad.

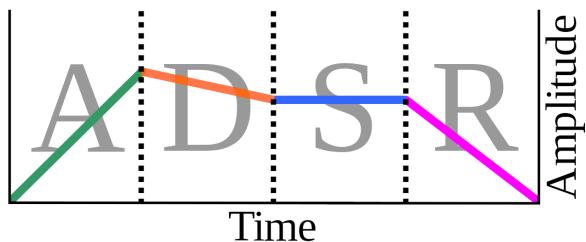


Figura 9.6: Ejemplo Gráfico de Envolvente de Amplitud

9.7.1 Canal 1 - Tono & Portamento

- **\$FF10:** Registro de portamento.

Los bits del 6 al 4 especifican la duración del portamento. El bit 3 indica el incremento/decremento. Los bits del 2 al 0 indican, por último, el desplazamiento.

000	Apagado
001	7.8 ms (1/128Hz)
010	15.6 ms (2/128Hz)
011	23.4 ms (3/128Hz)
100	31.3 ms (4/128Hz)
101	39.1 ms (5/128Hz)
110	46.9 ms (6/128Hz)
111	54.7 ms (7/128Hz)

Tabla 9.5: Duraciones del Portamento

- **\$FF11:** Duración del ciclo de trabajo.

Los bits 7 y 6 se utilizan para indicar el ciclo de trabajo. Los bits restantes sirven para especificar la longitud de onda.

00	12.5%
01	25%
10	50%
11	75%

Tabla 9.6: Ciclos de trabajo

- **\$FF12:** Envolvente de amplitud.

Los bits del 7 al 4 indican el volumen inicial (desde \$00 hasta \$0F). El bit 3, la dirección de la envolvente. Los bits 2-0 son el periodo.

- **\$FF13:** Frecuencia baja.

Los 8 bits nos especifican la frecuencia baja. En total son 11 bits, pero los 3 restantes están en la dirección de memoria \$FF14.

- **\$FF14:** Frecuencia alta.

El bit 7 es el iniciador (cuando se le da valor 1, el sonido se reinicia). El bit 6 activa o desactiva la longitud. Este bit es el único que se puede leer, el resto son de escritura. Los bits 2-0 son los correspondientes a la frecuencia baja.

9.7.2 Canal 2 - Tono

- **\$FF16:** Duración del ciclo de trabajo.

Los bits 7 y 6 se utilizan para indicar el ciclo de trabajo. Los bits restantes sirven para especificar la longitud de onda. Los valores correspondientes son los mismos a la de la tabla **9.6**.

- **\$FF17:** Envolvente de amplitud.

Los bits del 7 al 4 indican el volumen inicial (desde \$00 hasta \$0F). El bit 3, la dirección de la envolvente. Los bits 2-0 son el periodo.

- **\$FF18:** Frecuencia baja.

Los 8 bits nos especifican la frecuencia baja. En total son 11 bits, pero los 3 restantes están en la dirección de memoria \$FF14.

- **\$FF19:** Frecuencia alta.

El bit 7 es el iniciador (cuando se le da valor 1, el sonido se reinicia). El bit 6 activa o desactiva la longitud. Este bit es el único que se puede leer, el resto son de escritura. Los bits 2-0 son los correspondientes a la frecuencia baja.

9.7.3 Canal 3 - Onda Programable

- **\$FF1A:** Encendido/Apagado.

El valor del bit 7 nos indica si el sonido está encendido o apagado.

- **\$FF1B:** Longitud.

Los 8 bits especifican la longitud de onda. Solamente se utiliza si el bit 6 de \$FF1E esta a 1.

- **\$FF1C:** Nivel de volumen.

Los bits 6 y 5 indican el nivel de salida.

00	Apagado
01	100%
10	50%
11	25%

Tabla 9.7: Niveles de volumen

- **\$FF1D:** Frecuencia baja.

Los 8 bits nos especifican la frecuencia baja. En total son 11 bits, pero los 3 restantes están en la dirección de memoria \$FF14.

- **\$FF1E:** Frecuencia alta.

El bit 7 es el iniciador (cuando se le da valor 1, el sonido se reinicia). El bit 6 activa o desactiva la longitud. Este bit es el único que se puede leer, el resto son de escritura. Los bits 2-0 son los correspondientes a la frecuencia baja.

- **\$FF30:** Patrón de onda.

Almacena la forma de la onda para generar datos de sonido aleatorios. Contiene 32 muestras de 4 bits que se reproducen con los 4 bits altos en primer lugar.

9.7.4 Canal 4 - Ruido Blanco

El ruido blanco surge variando de manera aleatoria la amplitud de una frecuencia.

- **\$FF20:** Longitud.

Los 8 bits especifican la longitud de onda. Solamente se utiliza si el bit 6 de \$FF1E esta a 1.

- **\$FF21:** Envolvente de amplitud.

Los bits del 7 al 4 indican el volumen inicial (desde \$00 hasta \$0F). El bit 3, la dirección de la envolvente. Los bits 2-0 son el periodo.

- **\$FF22:** Contador polinómico.

Los bits del 7 al 4 sirven para especificar la frecuencia de reloj con la que la amplitud va a cambiar de forma aleatoria. El bit 3 especifica la medida del contador. Los bits restantes (2-0) hacen referencia al radio de división de las frecuencias.

- **\$FF23:** Disparador o Iniciador.

El bit 7 es el iniciador (cuando se le da valor 1, el sonido se reinicia). El bit 6 activa o desactiva la longitud.

9.7.5 Registros de Uso General

- **\$FF24:** Control de volumen.

Controla tanto el volumen de los canales izquierdos como los derechos.

- **\$FF25:** Selección de salida de canal.

Indica si un canal debe salir por la izquierda o por la derecha. Las terminales de salida se nombran como SO1 y SO2.

Bit 7	Canal 4 - SO2
Bit 6	Canal 3 - SO2
Bit 5	Canal 2 - SO2
Bit 4	Canal 1 - SO2
Bit 3	Canal 4 - SO1
Bit 2	Canal 3 - SO1
Bit 1	Canal 2 - SO1
Bit 0	Canal 1 - SO1

Tabla 9.8: Niveles de volumen

- **\$FF26:** Apagado o encendido del sonido.

El bit 7 en el controlador general (apaga o enciende todos los canales). El bit 3 controla el canal 4, el bit 2 el canal 3, etc. Desactivar el sonido puede ahorrar hasta un 16% de batería.

10 Conclusiones Finales

En esta sección voy a hacer unas breves conclusiones respecto al resultado definitivo del proyecto y todo lo que se ha aprendido a lo largo de su desarrollo.

10.1 Estado del Producto

Existen muchas funcionalidades definidas en el **Game Design Document** que no se han llegado a finalizar, como el poder subir de nivel al ganar experiencia, el poder interactuar con distintos NPC's, o que existan distintas mazmorras con un jefe final en cada una de ellas. Podemos decir entonces que el juego, en el estado que se encuentra actualmente, es muy mejorable.

Sin embargo, es un **proyecto cerrado, divertido y con sentido**, el cual cumple con su cometido perfectamente. Por ende, es un proyecto que en el futuro se puede enseñar en un portfolio con mucho orgullo.

Por otro lado, como es normal, a lo largo del desarrollo han surgido **nuevas ideas** que pueden mejorar mucho la experiencia de usuario. Aún así, lo propio fue centrarse en las tareas marcadas como principales desde principio de desarrollo y, en caso de dar tiempo, hacerlas. Se van a nombrar en el siguiente apartado.

10.2 Posibles Futuras Mejoras

Como hemos comentado, Golden Sacra dispone de un amplio abanico de posibles mejoras para mejorar la experiencia de juego:

- **Generación Procedimental:** Esto nos puede servir para tener un número indefinido de pisos y mazmorras. Al disponer de una consola muy limitada en cuanto a memoria, hacer un algoritmo de semilla (por ejemplo) para generar todos los pisos que compongan una mazmorra, nos puede ahorrar tanto en espacio como en tiempo. Además, esto conseguiría que los pisos se volviesen a generar cada vez que el jugador entre a la misma mazmorra, obteniendo un resultado diferente y, por tanto, aumentando de este modo la rejugabilidad.
- **Animaciones:** Si bien en el producto se han conseguido introducir todas las animaciones básicas, faltan muchas que aumentan la satisfacción general del usuario, como las de morir o consumir un objeto.
- **Mejorar la Inteligencia Artificial:** Como se ha explicado a lo largo del desarrollo, el comportamiento de los enemigos no se pueden considerar como una "inteligencia

artificial". Habría que definirlos más bien como patrones. Posibles mejoras son la de que un enemigo, al ver al jugador en un radio, decida cerrar una puerta para que este tenga que proceder por otro camino algo más largo. Detalles como este que acabo de describir pueden dotar al proyecto de una mayor variedad a la hora de tomar decisiones.

- **Más Niveles:** Actualmente solo disponemos de una mazmorra con 3 pisos. Una de las mejoras más claras es la de expandir este número de niveles disponibles, los cuales se vayan desbloqueando según el jugador avance y consiga llegar al último piso de una mazmorra específica.
- **Nuevos Objetos:** Disponer solamente de 3 objetos está bien desde el punto de vista a corto plazo. Si queremos hacer un proyecto entretenido en todas las horas de juego que este pueda ofrecer, necesitamos disponer de una amplia gama de objetos que puedan decantar la balanza entre morir o salvarse.

10.3 Opiniones Personales

Creo que como ingeniero he conseguido avanzar y mejorar bastante a lo largo del desarrollo. El lenguaje ensamblador no solo te hace entender mejor el funcionamiento de la máquina objetivo (una Game Boy en este caso), sino que además te ayuda a entender mejor lo que está ocurriendo por debajo a la hora de programar en cualquier otro lenguaje de alto nivel.

Cuando entré en la carrera siempre tuve la incógnita de cómo se podría llegar a hacer un juego para esta consola, y a parte de poder decir "ya se ha hecho", puedo decir también "ya lo he hecho".

Comenzar proyectos nuevos siempre da miedo, y más cuando los debes hacer por tu cuenta, sin nadie que te pare los pies ante las posibles malas decisiones, ni que te pueda ayudar ante los problemas que se te pongan de frente. Sin embargo, al igual de que el riesgo es mayor, la recompensa también lo es. Mis conocimientos han aumentado en gran medida, y esto se puede comprobar con el simple hecho de que volvería a hacer todo desde 0. Y aún así espero que, en caso de hacerlo, una vez termine pueda volver a decir exactamente lo mismo.

Bibliografía

- Anthony J. Bentley. (2010). *RGBDS Documentation*. <https://rednex.github.io/rgbds/>.
- Computing History. (2013). *Nintendo Game Boy*. <http://www.computinghistory.org.uk/det/4033/Nintendo-Game-Boy/>.
- David DeGraw. (2017). *Catskull Games*. <https://catskullgames.com/>.
- David Pello González. (2015). *LaDecadence*. http://wiki.ladecadence.net/doku.php?id=tutorial_de_ensamblador.
- Doctor Ludos. (2018). *Cómo hacer un juego para la Game Boy original en pleno 2018: 'Sheep It Up!'* <https://www.xataka.com/videojuegos/como-hacer-un-juego-para-la-game-boy-en-2018-sheep-it-up>.
- Execin. (2017). *DMA Transfers on Game Boy*. <https://exez.in/gameboy-dma>.
- Felipe Alfonso (bitnenfer). (2008). *FlappyBoy*. <https://github.com/bitnenfer/flappy-boy-asm>.
- Luis Colomer Blasco. (2016). *Acústica Musical*. <http://cursodeacusticamusical.blogspot.com/2015/12/capitulo-9-envolventes-de-amplitud-y-de.html>.
- Marat Fayzullin, P. R., Pascal Felber, y Korth, M. (1998). *Pan Docs*. <http://bgb.bircd.org/pandocs.htm>.
- Marc Davis. (2020). *I will compose 8 bit music for your video game or other projects*. https://es.fiverr.com/marc_davs/compose-8-bit-music-for-your-video-game-or-other-projects?context_referrer=search_gigs&source=main_banner&ref_ctx_id=185d161a-e2f0-4ebf-b235-bc4c14845747&pckg_id=1&pos=3.
- Marc Rawer. (1999). *Game Boy CPU Manual*. <http://marc.rawer.de/Gameboy/Docs/GBCPUMAN.pdf>.
- Nintendo. (1999). *Manual de Programación para Game Boy*. <http://index-of.es/Varios-2/Game%20Boy%20Programming%20Manual.pdf>.
- nitro2k01. (2008). *Game Boy Development Wiki*. https://gbdev.gg8.se/wiki/articles/Main_Page.
- Pret. (2019). *Pokémon Red/Blue Disassembly Code*. <https://github.com/pret/pokered>.
- Randy Mongenel. (2010). *Duo's GameBoy ASMSchool*. <http://gameboy.mongenel.com/index.html#>.

- Wikipedia. (2004). *The Legend Of Zelda: Oracle Of Seasons y Oracle Of Ages*. https://es.wikipedia.org/wiki/The_Legend_of_Zelda:_Oracle_of_Seasons_y_Oracle_of_Ages.
- Wikipedia. (2008a). *Pokémon Mundo Misterioso: Equipo de Rescate Rojo/Azul*. https://es.wikipedia.org/wiki/Pok%C3%A9mon_Mundo_Misterioso:_Equipo_de_Rescate_Azul_y_Equipo_de_Rescate_Rojo.
- Wikipedia. (2008b). *Pokémon Rojo/Azul*. https://es.wikipedia.org/wiki/Pok%C3%A9mon_rojo_y_Pok%C3%A9mon_azul.
- Wikipedia. (2011). *Game Boy Family*. https://en.wikipedia.org/wiki/Game_Boy_family.
- Zalo. (2015). *GB Sound Sample Generator*. <https://github.com/Zal0/GBSoundDemo>.