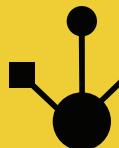




Escuela
Politécnica
Superior

Touch & Brush



Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Carla Macia Diez

Tutor/es:

Francisco José Gallego Durán

Mayo 2020



Universitat d'Alacant
Universidad de Alicante

Touch & Brush

Videojuego para Nintendo DS y manual de desarrollo

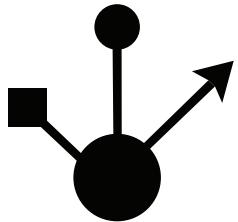
Autor

Carla Macia Diez

Tutor/es

Francisco José Gallego Durán

Departamento de Ciencia de la Computación e Inteligencia Artificial



Grado en Ingeniería Multimedia



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Mayo 2020

Resumen

Touch & Brush es un videojuego de agilidad mental para la consola **Nintendo DS**, desarrollado en **C++** y el conjunto de herramientas de **devkitPro**, basado principalmente en **Magic Cat Academy** de Google (Doodle de Halloween 2016).

En cada **nivel** el jugador deberá vencer a un número de enemigos para poder continuar y estos irán apareciendo alrededor de la pantalla e irán moviéndose hacia el jugador, el cual estará en el centro. Dichos enemigos poseerán un **patrón** visible que el jugador deberá **dibujar** correctamente en la pantalla táctil para poder derrotarlos antes de que ellos le alcancen y le quiten vida.

Abstract

Touch & Brush is a mental agility video game for the **Nintendo DS** console, developed in **C ++** and using **devkitPro** toolkit, based mainly on Google's **Magic Cat Academy** (2016 Halloween Doodle).

On each **level** the player must defeat a number of enemies in order to continue and they will appear around the screen and move towards the player, who will be in the center. These enemies will have a **pattern** that the player must correctly **draw** in the touch screen in order to defeat them before they reach and damage him.

Justificación y objetivos

Desde que comencé el grado en ingeniería multimedia hace ya 5 años he podido participar en el desarrollo de un gran número de proyectos enfocados en los videojuegos que, aunque se traten de simples demos básicas con fines totalmente didácticos, me han servido para darme cuenta de que no solo me apasiona jugar a videojuegos, sino que también adoro crearlos. Es por eso que mi último proyecto como estudiante de este grado no podía ser otro que crear un videojuego para la consola que más horas de diversión me ha brindado en mi infancia: la Nintendo DS.

Este proyecto supone una gran ilusión para mí, pues ser capaz de crear un juego para una plataforma en la que años atrás yo disfrutaba y pasaba horas y horas jugando tanto a sus grandes títulos es un sentimiento que no puedo llegar a expresar con palabras, y que por supuesto, influirá a la hora de perfeccionar cada detalle. Además, para poder desarrollar un juego de una manera correcta debemos **entender cómo trabaja la máquina** a bajo nivel, así, siempre que cometamos un fallo seremos capaces de localizar el error más rápido y solucionarlo de una forma eficiente. En los ordenadores y consolas modernos esto puede llegar a ser un gran dolor de cabeza para el desarrollador, pues son **máquinas muy complejas**. Sin embargo, las **consolas retro**, al tratarse de una versión **simplificada**, son unas **excelentes candidatas** para comenzar desarrollando pequeños juegos que nos permitan entender su funcionamiento y que, al fin y al cabo, la base de las máquinas modernas también está ahí.

Por otro lado, la Nintendo DS es una consola con *hardware* específico para el desarrollo de juegos a la que se le puede sacar muchísimo partido y, obviamente, las grandes empresas que antiguamente desarrollaban para esta máquina no iban a andar compartiendo el código fuente ni enseñando a terceros. Es por ello que si como fanático deseas introducirte en el mundo del desarrollo para Nintendo DS y buscas información, lo más probable es que pases horas y horas leyendo artículos hechos por personas que han creado manuales con pequeñas demos y ejemplos pero que llevan muchos años sin actualizarse o no terminen de explicar del todo lo que tú esperas. Así pues, espero poder enfocar esta memoria a modo de **manual** para que le sirva a cualquier persona que desee desarrollar para la **Nintendo DS**.

Dicho esto, los objetivos a cumplir en la realización de este proyecto son:

- **Analizar la Nintendo DS y entender sus capacidades y limitaciones.**
- **Analizar librerías y frameworks existentes.**
- **Aprender/Profundizar programación en C++.**
- **Comprender el funcionamiento del hardware.**

- Diseñar y desarrollar un videojuego completo.
 - Realizar la producción física del videojuego en cartucho.
 - Elaborar un manual de desarrollo referenciable que aúne el conocimiento disponible sobre desarrollo en Nintendo DS.
-

Agradecimientos

Antes de nada, he de agradecer a todas las personas que me han ayudado, no solo durante todo el desarrollo de este proyecto, sino también durante toda mi etapa como estudiante de este grado. Ellos han sido gran parte de mi motivación a seguir adelante y mi apoyo en los momentos más difíciles.

En primer lugar gracias a mis padres, por proporcionarme los medios necesarios para poder formarme en un campo que tanto me gusta y por interesarse por éste a pesar de no entender mucho del tema. También, por saber detectar cuándo lo estaba pasando mal e intentar ayudarme al instante, a pesar de que yo sea de pocas palabras en ese aspecto.

A mi pareja y amigos, no solo a los antiguos, también a los que he conocido en el grado y que me han permitido aprender mucho de ellos. Ángel, Leire, Franck, Yaiza, Raquel, Mer, Alberto... con ellos y muchos más he podido compartir gran cantidad de trabajos, *hobbies*, buenos y malos momentos que sin duda espero seguir compartiendo en el futuro.

Y por último, gracias a mi tutor y al profesorado del grado, en especial a los profesores de cuarto curso. Ellos me han ayudado a ver la luz al final del túnel y hacerme creer que puedo dedicarme a esto después de unos primeros años de caos y de estar perdida entre tantas asignaturas distintas. Gracias a su dedicación y esfuerzo los alumnos de multimedia hemos podido disfrutar al máximo nuestros últimos años de carrera.

A todos y cada uno de vosotros, gracias.

Índice general

Resumen	v
Justificación y objetivos	vii
Agradecimientos	ix
Terminología	xxiii
1 Introducción	1
2 Marco teórico	3
2.1 Nintendo DS	3
2.1.1 Versiones	4
2.1.1.1 Nintendo DS Lite	4
2.1.1.2 Nintendo DSi	4
2.1.1.3 Nintendo DSi XL	6
2.1.2 Especificaciones	6
2.2 Estudio de mercado	7
2.3 Algoritmo de reconocimiento de escritura	7
2.3.0.1 Graffiti de PAlib	8
2.3.0.2 Graffiti de Palm	10
2.4 Referentes	14
2.4.1 Magic Cat Academy	14
2.4.2 Una pausa con... Brain Training Ciencias	18
2.4.3 Lost Magic	20
2.4.4 Wario Ware: Touched!	22
3 Metodología	25
3.1 Planificación	25
3.1.1 Iteración 0	25
3.1.2 Iteración 1 - 4	26
3.1.3 Iteración 5 - 6	26
3.2 Mínimo producto viable	26
4 Diseño del juego (Game Design Document)	29
4.0.1 Características	29
4.0.2 Historia	29
4.0.3 Ambientación y estilo	29
4.0.4 Jugabilidad y mecánicas	31

4.0.5	NPCs	32
4.0.5.1	Enemigos comunes	32
4.0.5.2	Aliado	32
4.0.5.3	Jefe	33
4.0.6	Niveles	34
4.0.7	Controles	35
4.0.8	Estados del juego	36
4.0.9	Pantallas	36
5	Desarrollo	41
5.1	Herramientas	41
5.1.1	Entorno de desarrollo	41
5.1.2	Emuladores	41
5.1.3	Cartuchos Flash	43
5.1.3.1	R4	44
5.1.4	DevkitPro	44
5.1.5	Libnds	45
5.1.6	Grit	45
5.2	Iteraciones	47
5.2.1	Iteración 0 - Investigación previa y primeros pasos en NDS	47
5.2.1.1	Fondos	47
5.2.1.2	Sprites	52
5.2.1.3	Input	56
5.2.2	Iteración 1 - Pantallas del juego y lógica de los enemigos	57
5.2.3	Iteración 2 - Mecánicas y primera versión del algoritmo de reconocimiento de gestos	59
5.2.4	Iteración 3 - Algoritmo de reconocimiento de gestos	61
5.2.5	Iteración 4 - Jugabilidad y ajustes de aparición de enemigos	68
5.2.5.1	Algoritmo de Bresenham	77
5.2.6	Iteración 5 - Retoque al algoritmo de reconocimiento de gestos y animaciones	84
5.2.7	Iteración 6 - Mejoras visuales, corrección de niveles y mejora del producto final	92
6	Conclusiones	103
6.1	Estado del juego	103
6.2	Mejoras	105
6.3	Lecciones aprendidas	106
6.4	Conclusión personal	107
Bibliografía	109	
7	Anexo I - Detalles técnicos de Nintendo DS	111
7.1	Procesadores	111
7.1.1	ARM9	111
7.1.2	ARM7	112

7.2	Memoria de vídeo	113
7.3	OAM	114
7.4	DMA	115
7.5	Sonido	116
7.6	ROM	116

Índice de figuras

2.1	Nintendo DS	3
2.2	Nintendo DS Lite	4
2.3	Nintendo DSi	5
2.4	Menú de Nintendo DS (izquierda) frente al de Nintendo DSi (derecha)	5
2.5	Nintendo DSi XL	6
2.6	Caracteres y su trazo correspondiente de PA Graffiti	8
2.7	Correspondencia entre caracteres y los 32 rangos de ángulos.	9
2.8	Ejemplo de asignación de una cadena de caracteres a un trazo.	10
2.9	Gestos usados en PalmOS para definir los distintos caracteres	11
2.10	Ejemplo gráfico del cálculo de la longitud total en el eje x	13
2.11	Ejemplo gráfico del cálculo de la longitud total en el eje y	14
2.12	Doodle Halloween 2016	15
2.13	Cuarto nivel del juego, el gimnasio	15
2.14	Primera fase del jefe donde aparecen enemigos.	16
2.15	Segunda fase del jefe donde le atacamos diréctamente	17
2.16	Pantalla que aparece al completar el juego	17
2.17	Pantalla que aparece al perder todas tus vidas.	18
2.18	Menú principal de Una pausa con... Brain Training Ciencias	19
2.19	Minijuego Suma y Sigue en su modo normal	19
2.20	Minijuego Suma y Sigue en su modo arcade	20
2.21	Principal mecánica del Lost Magic. Arriba vemos los conjuros aprendidos y abajo en la pantalla táctil podemos dibujarlos.	21
2.22	Elementos del juego típicos de un RPG.	22
2.23	Pantalla de cambio entre minijuegos y minijuego específico de Wario Ware: Touched!.	23
3.1	Estado del tablero de Trello durante la iteración 0.	26
3.2	Concepto del mínimo producto viable.	28
4.1	Fondo de la pantalla de juego que representa la sala de un museo vacía y destrozada.	30
4.2	Concept art de Cherry, la protagonista del juego, los cuadros.	30
4.3	Sprites de Cherry y los principales enemigos.	31
4.4	Tipos de patrones con los que pueden aparecer los enemigos.	32
4.5	Tipos de patrones con los que pueden aparecer los enemigos.	33
4.6	Mockup del enfrentamiento con el jefe.	34
4.7	Posiciones iniciales desde las que pueden aparecer enemigos.	35
4.8	Diagrama de flujo entre estados	36
4.9	Mockup de la pantalla de título	37

4.10 Mockup del juego	38
4.11 Mockup de la pantalla de fin de partida	39
4.12 Mockup de la pantalla de victoria	39
4.13 Mockup de la pantalla de pausa	40
5.1 Desmume emulando WarioWare: Touched!	42
5.2 Versión debug de no\$gba emulando WarioWare: Touched!	43
5.3 Herramienta de depurado para fondos (izquierda) y sprites (derecha) de no\$gba	43
5.4 Cartucho R4	44
5.5 Wingrit, versión GUI para Windows de Grit	46
5.6 Grit versión comandos	47
5.7 Carpeta del proyecto	48
5.8 Matriz identidad	50
5.9 Diferencias entre la representación normal binaria y la coma fija	50
5.10 Resultado en el emulador Desmume	52
5.11 Reordenar paleta de colores desde GIMP.	55
5.12 Prueba de dibujado de sprites.	56
5.13 Diagrama de los estados del juego y sus clases.	58
5.14 Resultado de la ejecución y elementos gráficos vistos desde el OAM Viewer de no\$gba	59
5.15 Boceto y acabado del sprite del jugador	60
5.16 Boceto y acabado del sprite del enemigo	61
5.17 Visualización del trazo ideal y del usuario sin aplicar el desplazamiento	64
5.18 Visualización del trazo ideal y del usuario con el desplazamiento aplicado	65
5.19 Resultado de la prueba del algoritmo desarrollado probando	67
5.20 Resultado de la prueba del algoritmo desarrollado probando	68
5.21 Pantalla de inicio de Wario Ware: Touched!	69
5.22 Resultado de la consulta con la mala configuración de fondos	71
5.23 Resultado de la consulta con una buena configuración de fondos	72
5.24 Cambio de imagen a modo indexado y selección de paleta	73
5.25 Fondo dibujado con un problema en la paleta	74
5.26 Superposición de tonos grises en la paleta de los fondos de la pantalla inferior	75
5.27 Paletas de los sprites que se salen de los límites	75
5.28 Comparativa de disposición de pixeles en pantalla y en memoria	76
5.29 Resultado de usar la función drawPointSub para dibujar una línea	77
5.30 Aplicación del algoritmo de Bresenham para dibujar líneas en un medio discreto	78
5.31 Dibujado en la pantalla táctil usando el algoritmo de Bresenham	81
5.32 Diagrama de la relación entre las clases Level y PatternManager	82
5.33 Definición visual de las posiciones iniciales y caminos de los enemigos	83
5.34 Resultado de esta implementación	84
5.35 Ejemplo visual del cálculo de la desviación entre los puntos de dos trazos	85
5.36 Desplazamiento erróneo del trazo del usuario (rosa) con respecto al trazo ideal (azul)	87
5.37 Dibujado de un trazo y señalización de su centro de masas	89
5.38 Frames de la animación del jugador	90

5.39	Frames de la animación del enemigo	92
5.40	Boceto del fondo de nivel y versión acabada	94
5.41	Icono y título por defecto de un proyecto de NDS que use devkitARM	94
5.42	Erronea visualización del ícono de la aplicación (firmware de NDSi emulado en no\$gba)	95
5.43	Mensaje de conversión del Makefile	96
5.44	Frames de la animación del enemigo	96
5.45	Posición en memoria del color específico	97
5.46	Valores RGB asignados al espectro HSV	98
5.47	Implementación de Wario Ware: Touched de las animaciones para las cinematáticas	99
5.48	Aspecto final de la pantalla del menú principal	100
5.49	Posiciones finales de los enemigos.	101
5.50	Portada de Touch & Brush.	102
6.1	Menú principal y nivel de Touch & Brush	104
6.2	Pantalla de victoria y previsualización en la R4 de Touch & Brush	104
7.1	Representación gráfica de la tubería de procesos del ARM9	112
7.2	Representación gráfica de la tubería de procesos del ARM7	113
7.3	Interior de un cartucho de juego para NDS. El superior (NTR-031) añade el sensor infrarrojo.	117

Índice de tablas

2.1	Especificaciones técnicas de la NDS	7
2.2	Datos de cada letra estudiados.	12
5.1	Máscaras de teclas.	57
7.1	Tamaños de los 9 bancos de VRAM de la NDS.	114

Índice de Códigos

5.1	Bucle principal del juego	48
5.2	Mapeo de memoria de video	49
5.3	Bucle principal del juego	49
5.4	Bucle principal del juego	49
5.5	Registros de control para la matriz afín de los fondos	50
5.6	Comandos de grit por archivo	51
5.7	Copia de la imagen en la memoria de video	51
5.8	Mapeo del banco E a memoria de sprites	52
5.9	Copia de la información de sprites a memoria de video	53
5.10	Declaración de la copia de la OAM y	53
5.11	Declaración de la copia de la OAM y	54
5.12	Función para comprobar si el jugado mantiene pulsado el pad de direcciones hacia abajo	56
5.13	Implementación del patrón Singleton	58
5.14	Implementación del patrón Singleton	60
5.15	Función consoleInit con los parámetros adecuados para crear un fondo que nos sirva para depurar	61
5.16	Inserción de valores que conforman el trazo	62
5.17	Algoritmo de reconocimiento de gestos	65
5.18	Configuración de los registros controladores de los fondos de la pantalla inferior	69
5.19	Función de dibujado de pixeles dada una entrada de usuario	76
5.20	Implementación del algoritmo de Bresenham	79
5.21	Llamada a Bresenham	81
5.22	Búsqueda del punto más cercano con la función findNearest	85
5.23	Cálculo del centro de masas con la función calculateCenterOfMass()	88
5.24	Animación del sprite del jugador	91
5.25	Animación del sprite del enemigo al moverse	92
5.26	Cambios en el Makefile para establecer el título y el ícono	94
5.27	Conversión del archivo BMP a GRF en el Makefile	96
5.28	Animación del texto	98

Terminología

A lo largo de este documento se van a utilizar una serie de abreviaturas con el fin de hacer la lectura más amena. Todos estos términos están explicados a continuación.

- **NDS:** Nintendo DS.
- **NDSi:** Nintendo DSi.
- **GBA:** Game Boy Advance.
- **ARM9:** Procesador ARM946E-S de la Nintendo DS.
- **ARM7:** Procesador ARM7TDMI de la Nintendo DS.
- **R4:** Revolution for NDS.
- **RPG:** Role Playing Game o videojuego de rol.
- **VRAM:** Memoria RAM de vídeo.
- **RAM:** Memoria de Acceso Aleatorio.
- **NPC:** Non Playable Characters, hace referencia a todos los personajes que aparecen en el juego que no son el jugador.
- **HUD:** Head-Up Display, es la información que, por lo general en un videojuego se muestra siempre por pantalla.
- **GRF:** Formato de archivo gráfico usado por Microsoft GraphEdit.

1 Introducción

Este proyecto trata sobre el desarrollo de un **juego para la consola** de Nintendo, **Nintendo DS** y sus posteriores versiones (hasta la Nintendo DSi XL).

El videojuego, titulado Touch & Brush, se trata de un juego de categoría **agilidad mental** y en el cual se progresar por **niveles**. En cada nivel, aparecerán una serie de **enemigos** que tendrán un **patrón específico** e irán a atacar al jugador. Este, si no quiere perder todas sus vidas y por tanto la partida, debe ser más rápido y **dibujar dichos patrones** en la pantalla táctil para derrotarlos.

Esta es una idea que puede parecer sencilla, pero que a nivel técnico lleva bastante trabajo, no solo por desarrollar todas las **mecánicas y aspectos visuales** para una consola limitada como es la Nintendo DS, sino también porque el propio proyecto requiere analizar, investigar, e incluso desarrollar un **algoritmo de reconocimiento de gestos**, para que el juego sea capaz de identificar el trazo del usuario. Y eso no es un trabajo que pueda hacerse en una tarde.

También, requiere trabajo tanto de **diseño del juego** completo, como **planificación** del proyecto, y sobre todo la parte de **documentación**, pues uno de los objetivos es que esta memoria sirva de referencia o manual a cualquiera con una base de programación que quiera desarrollar para NDS. Por ello, debemos plasmar cada concepto y procedimiento de la manera más detallada posible.

Por otro lado, la Nintendo DS, tal y como hemos dicho, se trata de una consola con limitaciones técnicas, es por ello que debemos **analizarla** previamente así como conocer las diferencias entre sus versiones. Tiene dos procesadores ARM que ejecutan los juegos originales y retrocompatibles, doble pantalla (siendo una de ellas táctil), zonas hardware dedicadas a los gráficos (VRAM, OAM), soporte para sonido estéreo...

Todos y cada uno de esos aspectos los describiremos en los apartados siguientes.

2 Marco teórico

2.1 Nintendo DS

La **Nintendo DS** es una videoconsola portátil de la compañía de origen japonés **Nintendo**. Creada para suceder a la Game Boy Advance, se lanzó al mercado el **21 de noviembre de 2004 en Estados Unidos**, retrasándose su fecha de lanzamiento en **Japón al 2 de diciembre** y en **Europa al 5 de marzo del año siguiente** con un **precio de 149.99€**. La consola original y sus versiones posteriores alcanzaron un total de **154,90 millones unidades vendidas** en todo el mundo y su videojuego más vendido fue el **New Super Mario Bros**, con un total de 30,80 millones de copias vendidas.



Figura 2.1: Nintendo DS
Fuente: Nintendo

La consola consta de **dos pantallas LCD** retroiluminadas con brillo ajustable, siendo una de ellas (la inferior) **táctil**, un **pad de direcciones**, **seis botones de acción** (A,B,X,Y, R y L), **dos botones de control** (Start y Select) y un **micrófono**. En su interior posee **dos procesadores**, un **ARM946E-S** de 67 MHz que, en general, se encarga de la mayor parte del trabajo ejecutando los **juegos de NDS** y un **ARM7TDMI** de 33 MHz, que mueve los juegos de **GBA** y algunas funciones Wi-Fi. Profundizaremos en ambos procesadores más adelante.

Como curiosidad, las siglas DS significan **Developer's System** (Sistema de desarrolladores) ya que según la compañía, este sistema ofrece muchas herramientas para que los desarrolladores puedan innovar en sus creaciones. No obstante, también hicieron oficial que estas siglas también hacían referencia a **Dual Screen**, por su doble pantalla.

2.1.1 Versiones

Ahora que conocemos un poco más la Nintendo DS original, es importante que analicemos sus posteriores ediciones, ya que nuestro juego debe funcionar en cualquiera de ellas. Debemos conocer qué diferencias significativas existen y tenerlas en cuenta a la hora de desarrollar el juego. Así pues vamos a ir revisando la familia de consolas de Nintendo DS y conociendo sus principales cambios.

2.1.1.1 Nintendo DS Lite

Fue creada en **2006** por Nintendo, es ligeramente **más pequeña** que su predecesora e incluía una **carcasa** más elegante. Algunos **botones** fueron **relocalizados** y añadía la función de poder elegir entre **4 niveles de brillo** a diferencia de la original, que solo podía ajustarse al mínimo o al máximo.



Figura 2.2: Nintendo DS Lite

Fuente: Nintendo

2.1.1.2 Nintendo DSi

Salió a la venta a finales de **2008 en Japón** y en **2009 en el resto del mundo** y se trata de una **revisión** del modelo de la Nintendo DS Lite. Si bien mantiene muchas características intactas, hay algunas otras que se deben destacar.

El primer cambio notable y que a mucha gente no le agradó, fue que **dejaba de ser compatible** con los juegos de **GBA**, pues ya no poseía ranura para los cartuchos. Introdujeron una **cámara delantera** y otra **trasera**, ambas de 0.3 megapíxeles, con las que podías hacer fotos y editarlas mediante un programa llamado **Cámara DSi** que venía instalado en la consola por defecto. Poseía también una ranura para **tarjetas SD**, en las que podías guardar las



Figura 2.3: Nintendo DSi
Fuente: Nintendo

fotos de la cámara así como los programas o juegos que descargases de la **Tienda Nintendo DSi**.

Por último, otras mejoras o diferencias frente a sus modelos originales son una **mejora de la calidad del audio**, **aumento de la memoria interna y RAM**, el procesador principal **ARM9** pasó a ser de 133 MHz (**aumentando su velocidad**), un **aumento** ligero del **tamaño de las pantallas** y **reducción** de la **duración de la batería** a 16 horas, siendo de 18 horas y media las originales.

El **menú principal** de la consola es completamente distinto al de la DS original, pasando a ser bastante **más personalizable** y con una gran variedad de programas, los cuales algunos de ellos se mantenían desde la DS como el **Picto-Chat** o la **Descarga DS**.

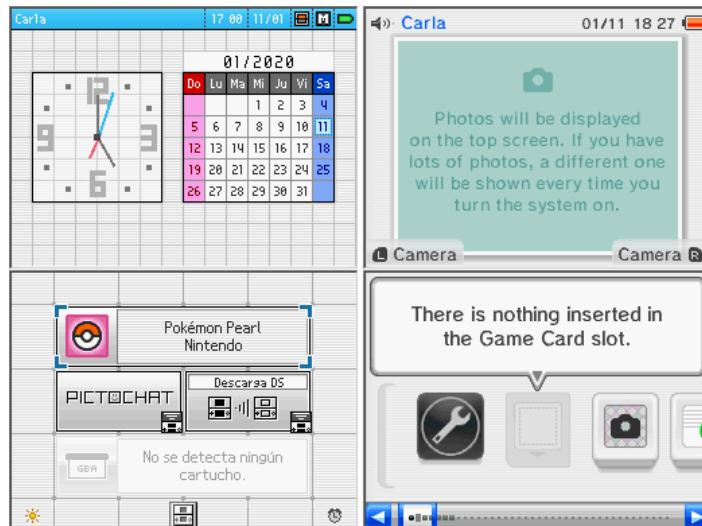


Figura 2.4: Menú de Nintendo DS (izquierda) frente al de Nintendo DSi (derecha)

Fué la primera consola portátil en tener **bloqueo regional**, no permitiendo que juegos desarrollados en países extranjeros se ejecutasesen en la consola si esta era originaria del mismo país.

2.1.1.3 Nintendo DSi XL

Se trata de un modelo adicional de NDSi que salió a la venta en **Japón en 2009**, donde originalmente se bautizó como Nintendo DSi LL, llegando a **España en 2010** donde pasó a llamarse Nintendo DSi XL.



Figura 2.5: Nintendo DSi XL

Fuente: Nintendo

El principal cambio destacable es el considerable aumento del tamaño de la consola como consecuencia del **aumento del tamaño de las pantallas** (estas tienen aproximadamente **una pulgada más** comparadas con la NDSi). En general es una consola perfecta para aquellos que tengan alguna discapacidad visual o les sea difícil o incómodo jugar con las anteriores por su tamaño.

2.1.2 Especificaciones

En esta sección vamos a conocer los detalles técnicos de la consola y profundizar en los más interesantes para poder entender qué la compone. Podemos ver a continuación una tabla a modo de resumen.

CPU	ARM946E-S 32bit RISC CPU ARM7TDMI 32bit RISC CPU
Velocidad de reloj	(ARM9) 66MHz (ARM7) 33MHz (16MHz en modo GBA)
RAM	4096KB
VRAM	656KB
Pantalla	2 pantallas LCD (256x192 px, 3"). Una de ellas táctil.
Paleta de colores	18-bit (262144 colores)
Sonido	16 canales de sonido estéreo
Comunicación	Wifi IEEE 802.11b
Alimentación	Batería recargable de ion de litio 850mAh
Peso	275g
Dimensiones	148.7mm × 84.7mm × 28.9mm

Tabla 2.1: Especificaciones técnicas de la NDS

Para profundizar más acerca de los procesadores de la consola, su memoria de vídeo, zonas hardware específicas para gráficos, sonido y ROM, visita el **Anexo I**.

2.2 Estudio de mercado

Si bien la NDS fue una consola revolucionaria por su gran número de ventas así como el éxito de sus grandes títulos, hoy en día ya no se producen juegos para ésta, pues ha sido **sucedida por la 3DS**. Es por esto que el mercado a día de hoy se reduce a fanáticos que desean comprar en buen estado estos juegos y lo hacen mediante la venta de **segunda mano**.

No obstante, existe la llamada **Scene**, conformada por los programadores e informáticos que desarrollan **aplicaciones y juegos no oficiales** para sistemas como es la NDS entre muchas otras. La existencia de este tipo de comunidades son muy buenas ya que gracias a foros o *hashtags* como el de #dsdev en Twitter puedes llegar a más público, haciendo que quizás se interesen por tu proyecto.

2.3 Algoritmo de reconocimiento de escritura

Algo esencial que necesitaremos para llevar a cabo este proyecto es un **algoritmo** que nos permita **reconocer el patrón que dibuje el usuario** en la pantalla táctil de entre unos que nosotros tengamos definidos.

El desarrollo de un *software* de estas características es sin duda algo que puede hacer que algunos desarrolladores se lleven las manos a la cabeza. Si bien es cierto que a día de

hoy existen gran número de programas de pago o incluso integrados en nuestros teléfonos y ordenadores que son capaces de distinguir caracteres dibujados por usuarios, hay muy pocos de ellos de código abierto. Por suerte, tenemos un ejemplo de un *software* así en el contexto que nos interesa.

Para el desarrollo en NDS existía una librería llamada **PAlib** que surgió en los primeros años de desarrollo *homebrew* cuyo objetivo, al igual que **libnds**, era facilitar las tareas al programador. No obstante, a pesar de haber sido bien acogida por los usuarios por su sencillez y fácil uso, los encargados de mantener el conjunto de librerías y herramientas de **devkitPro** afirmaban que esta librería estaba **mal diseñada y provocaba serios fallos** en la consola tales como corrupciones de la tarjeta SD. Es por ello que devkitPro, del cual hablaremos más adelante, dejó de dar soporte a esta librería¹ y ésta tras un tiempo esta cerró su página web y dejó de actualizarse. A día de hoy, parte de su documentación y código fuente siguen publicados en internet gracias a usuarios que lograron recuperarlas del archivo.

Lo que nos interesa a nosotros de esta librería es precisamente que implementó un **algoritmo de reconocimiento de escritura manuscrita** similar a otro más conocido como es **Graffiti de Palm**, pues en ambos la forma de dibujar los caracteres es la misma. Así pues, vamos a comprender a continuación cómo solucionan el problema ambos algoritmos.

2.3.0.1 Graffiti de PAlib

Como hemos comentado, este algoritmo de PAlib nos da como resultado el carácter al que más se parezca el dibujado por el usuario. Los trazos y sus respectivos caracteres son los mostrados en la siguiente figura, donde como podemos ver, no hay números ni algunos caracteres especiales como la 'ñ'. No obstante, esto es compensado con la posibilidad de poder incorporar **nuestros propios trazos**.



Figura 2.6: Caracteres y su trazo correspondiente de PA Graffiti
Fuente: Documentación de PAlib (Recuperada del archivo)

¹<https://devkitpro.org/wiki/PAlib>

El algoritmo funciona de la siguiente manera:

- Guarda en un array cada par de coordenadas $(x_1, y_1), \dots, (x_n, y_n)$ que conforman el **trazo** introducido por el usuario.
- De toda esa lista de puntos, escoge **17** muestras para computarlas.
- Calcula el **ángulo entre cada par de puntos** y le **asigna un carácter** dependiendo del rango en el que caiga dicho ángulo.
- **Compila todos los caracteres** de los ángulos calculados y genera una cadena de 16 caracteres que define dicho trazo.
- Busca dentro de unos trazos que ya tiene definidos, aquella **cadena de caracteres** que sea **más parecida** a la introducida por el usuario.

Como observamos, lo que básicamente hace el algoritmo es **caracterizar la línea con una cadena de caracteres** sirviéndose del ángulo que forman los puntos de esta misma. Una vez calcula el ángulo que forman dos puntos, para saber qué carácter asignarle se vale de una circunferencia la cual está dividida en **32 rangos** como se ven en la siguiente figura.

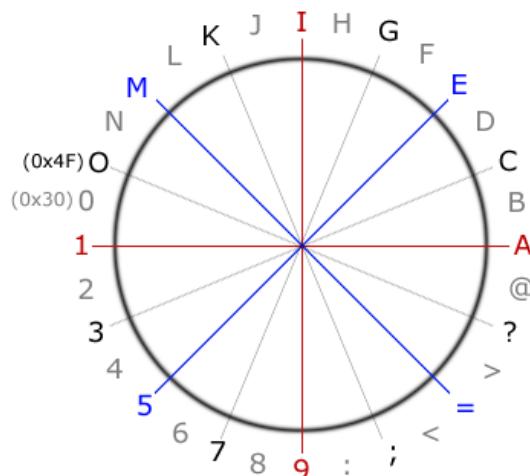


Figura 2.7: Correspondencia entre caracteres y los 32 rangos de ángulos.

Fuente: azyDream

Así pues, un carácter como por ejemplo la letra 'i' equivaldría a la cadena '9999999999999999', pues al dibujarse completamente en vertical desde arriba hacia abajo todos los ángulos calculados caerían en el rango que equivale al 9.

En la figura siguiente veríamos un ejemplo de cuál sería la cadena asignada al trazo mostrado en blanco. Las líneas azules representan los 17 puntos de muestra. Como se ve, los primeros caracteres del trazo son A, pues el ángulo entre puntos es de 0 grados y conforme el trazo va descendiendo se le van asignando las letras de los rangos consecuentes.

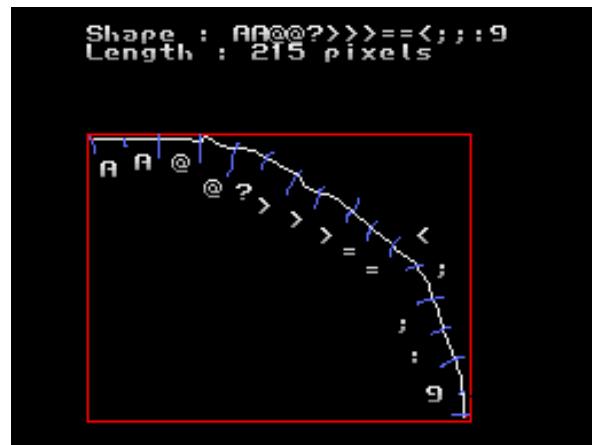


Figura 2.8: Ejemplo de asignación de una cadena de caracteres a un trazo.

Fuente: azyDream

Esta forma de trabajar del algoritmo es lo que permite que puedas **introducir tus propios diseños de forma sencilla para el programador**, pues solo tienes que dibujar el patrón que deseas en un proyecto a parte y haciendo llamadas a las funciones de crear una forma te devolverían una cadena que identifica dicho trazo. De hecho, el algoritmo te da la posibilidad de analizar formas en base a unas que tiene predefinidas o las tuyas propias.

2.3.0.2 Graffiti de Palm

Graffiti es un *software* de reconocimiento de escritura manuscrita desarrollado por el ingeniero informático **Jeff Hawkins** en **Palm** para las **PDAs** con sistema operativo PalmOS. Dicho *software* crea una ligera modificación y simplificación de los caracteres alfabéticos, como podemos ver en la siguiente figura, para que puedan ser dibujados con un lápiz táctil sobre una pantalla y sin levantar lo hasta completar el trazo.

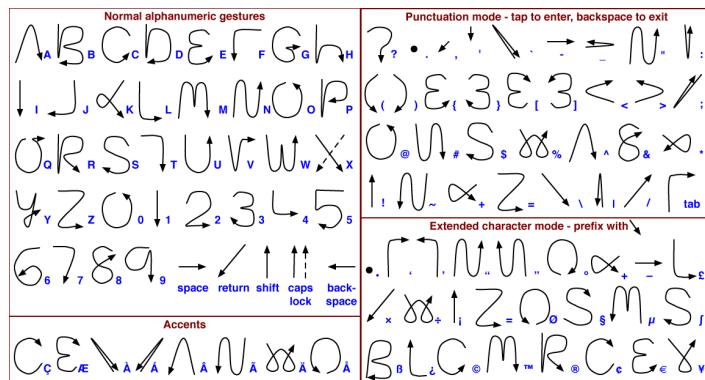


Figura 2.9: Gestos usados en PalmOS para definir los distintos caracteres
Fuente: IMewbot

No obstante, Graffiti tuvo que ser **retirado y más tarde sustituido** por una nueva versión de este ya que la compañía **Xerox demandó a Palm** porque su *software* de reconocimiento de escritura violaba la patente de Xerox sobre **tecnología Unistrokes**. Esto fue así porque Palm consiguió una demo de dicha tecnología antes de desarrollar su propio sistema.

Esto es importante comentarlo, pues a continuación explicaremos cómo funciona el algoritmo de la tecnología Unistrokes tal y como está reflejado en la pantente USPTO nº 5596656², pero debemos tener en cuenta que la primera versión de Graffiti funciona de la misma manera.

Así pues, vamos a describir brevemente lo que hace el algoritmo para identificar el trazo:

- Guardar en una lista ordenada cada par de coordenadas $(x_1, y_1), \dots, (x_n, y_n)$ que conforman el **trazo**.
- **Filtrar dicha lista** para eliminar el posible ruido y suavizar así el trazo para facilitar el cálculo.
- Comprobar si se trata de una **línea recta** y, si es así, calcular la pendiente de esta. Usar esa pendiente para comprobar los caracteres que se generan con una línea, como por ejemplo i, 1, espaciado... Y terminar la búsqueda.
- Si no se trata de una línea recta, **normalizar** el trazo para que quepa en una caja cuadrada y calcular las siguientes características:

El **desplazamiento en el eje x** entre el origen y el final

$$dx = xn - x1 \quad (2.1)$$

El **desplazamiento en el eje y** entre el origen y el final

$$dy = yn - y1 \quad (2.2)$$

²<https://patents.google.com/patent/US5596656>

El desplazamiento entre el origen y el punto medio del trazo en el eje x

$$dx_{x1-xn/2} = x(n/2) - x1 \quad (2.3)$$

El desplazamiento entre el origen y el punto medio del trazo en el eje y

$$dy_{y1-yn/2} = y(n/2) - y1 \quad (2.4)$$

La suma de las longitudes de las líneas que unen cada punto con su predecesor **proyectado en el eje x**

$$lengthxtot = length_x(x1, x2) + length_x(x2, x3) + \dots + length_x(xn - 1, xn) \quad (2.5)$$

La suma de las longitudes de las líneas que unen cada punto con su predecesor **proyectado en el eje y**

$$lengthytot = length_y(y1, y2) + length_y(y2, y3) + \dots + length_y(yn - 1, yn) \quad (2.6)$$

- Calculadas estas características, elegir el carácter que más se parece de una **tabla** como la siguiente:

	dx	dy	dx(x1-xn/2)	dy(y1-yn/2)	length x tot	length y tot
a	0.0	-1.0	0.0	-0.5	0.0	1.0
b	0.0	1.0	1.0	0.5	2.0	1.0
c	0.0	-1.0	-1.0	-0.5	2.0	1.0
d	0.0	1.0	-1.0	0.5	2.0	1.0
e	-1.0	0.0	-0.5	0.0	1.0	0.0
[...]	[...]	[...]	[...]	[...]	[...]	[...]

Tabla 2.2: Datos de cada letra estudiados.

- Si el resultado es una U o una O, determinar si el trazo gira en el **sentido horario o antihorario** para afirmar si es una U o una O respectivamente.

Como podemos ver, el algoritmo calcula una serie de **valores que caracterizan la línea** y los compara con los que han estimado para los distintos caracteres que pueden dibujar. No obstante, debemos detenernos a entender dos de ellos pues son los que más confusión pueden provocar: **lengthxtot** y **lengthytot**.

Tal y como se explicaba anteriormente, este valor representa para cada eje, la suma de las longitudes de unas líneas imaginarias desde un punto hasta su predecesor. Es importante no confundir este concepto con la distancia del punto inicial al final, pues si bien pueden coincidir en algunos casos, no significan lo mismo. Por ejemplo, imaginemos un trazo conformado por 3 puntos completamente colocados en vertical y separados en una unidad. En este caso,

ambos valores serían iguales, 3, pero si el trazo lo modificamos para que tenga un cuarto punto inferior al tercero, ahora la longitud sería mayor a 3 y la distancia menor a tres, pues habríamos añadido un segmento más en el caso de la longitud y por el contrario, la distancia entre el punto inicial y final ahora es menor.

La siguiente figura muestra un ejemplo gráfico de cómo se representa estos valores:

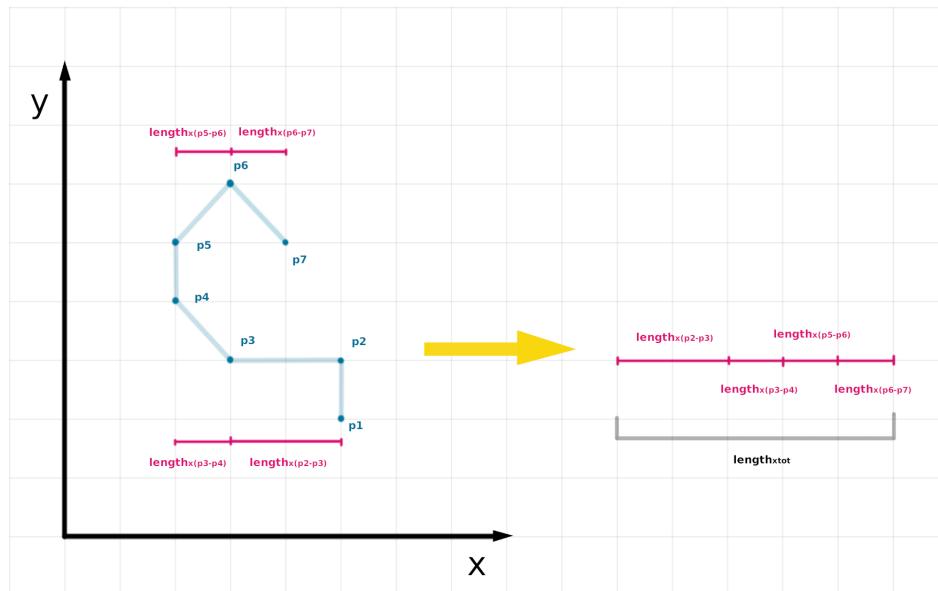


Figura 2.10: Ejemplo gráfico del cálculo de la longitud total en el eje x

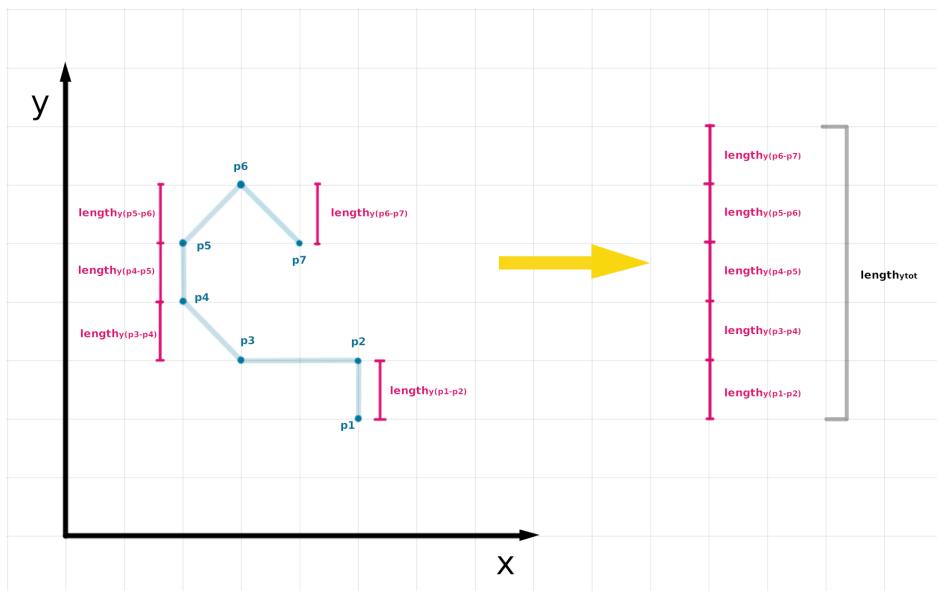


Figura 2.11: Ejemplo gráfico del cálculo de la longitud total en el eje y

Una vez comprendemos estos algoritmos un poco más en detalle podemos valorar si implementarlos o crear el nuestro propio. Si bien implementarlos otorgaría más facilidades y sería más eficaz y rápido a la hora de funcionar, se ha decidido crear uno propio, con el fin de **aprender** un poco más sobre las bases **matemáticas** que hacen que todos estos algoritmos puedan funcionar. En un entorno más laboral es muy importante que no nos dedicemos a reinventar la rueda, sin embargo este entorno sigue siendo académico, con lo cual es más enriquecedor aprender un poco más sobre estos aspectos. No obstante, es muy importante analizar estas soluciones y tenerlas presentes ya que en el caso de no poder llegar a crear un algoritmo propio, estos podrían servir de colchón.

2.4 Referentes

En esta sección analizaremos los **juegos en los que nos hemos basado** para elaborar un diseño de nuestro juego, ya sea en las mecánicas, arte, pantallas, o incluso las formas en la que los desarrolladores implementaron las funcionalidades.

2.4.1 Magic Cat Academy

Se trata de un juego para **navegador** que fue creado como **Doodle** de Google para celebrar **Halloween** del año 2016.



Figura 2.12: Doodle Halloween 2016

En este juego seremos Momo, una gata cuyo **objetivo es recuperar el libro de hechizos mágicos** de su escuela que los espíritus han robado. Para ello, debemos enfrentarnos a ellos dibujando en la pantalla con ayuda del ratón los **símbolos** que llevan justo en la parte superior para derrotarlos, avanzar por las salas de la escuela y dar con el jefe. Si algún fantasma nos **alcanza** perderemos una vida de un **total de 5** y al quedarnos sin ellas acabará la partida. Además, cuanto más rápido y mejor realicemos los patrones de los enemigos conseguiremos acumular **combos** que nos aumentarán la **puntuación**.

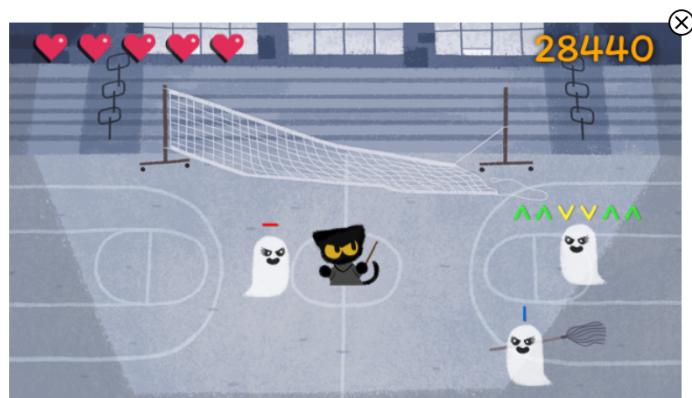


Figura 2.13: Cuarto nivel del juego, el gimnasio

El juego consta de **5 niveles** más un nivel de **jefe final**. En cada nivel hemos de derrotar a una serie de fantasmas y a un jefe con un comportamiento **único y característico**. A medida que los niveles van avanzando la **dificultad aumenta**, pues la **velocidad** de los fantasmas es mayor y también empiezan a aparecer enemigos con **combinaciones de patrones más complejas y largas**.

Los tipos de patrones que nos podemos encontrar son:

- **Línea vertical:** Por sí solo no tiene efecto diferente, puede combinarse con los demás.
- **Línea horizontal:** Por sí solo no tiene efecto diferente, puede combinarse con los demás.
- **Flecha hacia arriba (^):** Por sí solo no tiene efecto diferente, puede combinarse con los demás.
- **Flecha hacia abajo (v):** Por sí solo no tiene efecto diferente, puede combinarse con los demás.
- **Rayo:** Este patrón es más fuerte que los anteriores, si lo dibujamos cuando hay varios enemigos en pantalla afectará a todos, quitándole un par de símbolos a cada enemigo. Puede combinarse con los demás
- **Corazón:** Aparece de vez en cuando siempre que no tengamos la vida a tope, si lo dibujamos conseguiremos recuperar una unidad de vida.

El **nivel del jefe final** es algo distinto a los demás ya que no sigue la estructura que estos mantenían. En este nivel deberemos derrotar primero a una serie de fantasmas y después dibujar una combinación de patrones muy larga que el jefe posee para hacerle daño antes de que nos alcance. Esto se deberá **repetir tres veces**, y por supuesto la velocidad y complejidad aumenta cuanto más cerca estemos de derrotarlo.

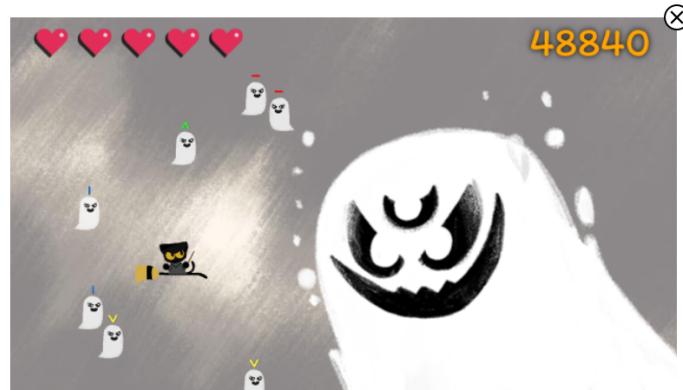


Figura 2.14: Primera fase del jefe donde aparecen enemigos.

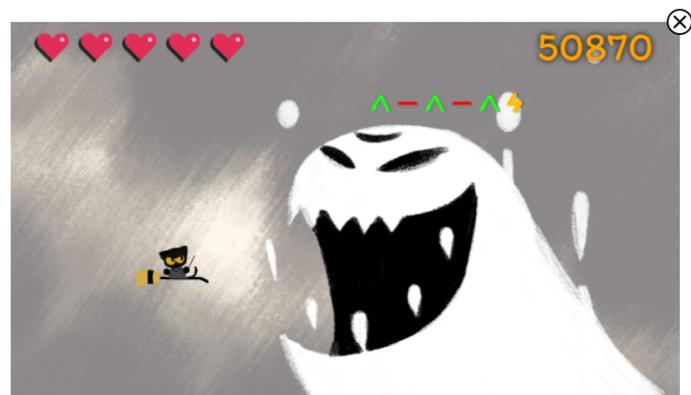


Figura 2.15: Segunda fase del jefe donde le atacamos diréctamente

Como ya se ha comentado, el jugador puede ir acumulando combos para mejorar su puntuación. Tanto si ganamos la partida como si la perdemos, podremos **compartir en las redes sociales nuestra puntuación**. Esto es una buena idea ya que permite que los usuarios comparten sus experiencias con otros y les inciten a jugar ya que aporta cierto factor de **competitividad**.

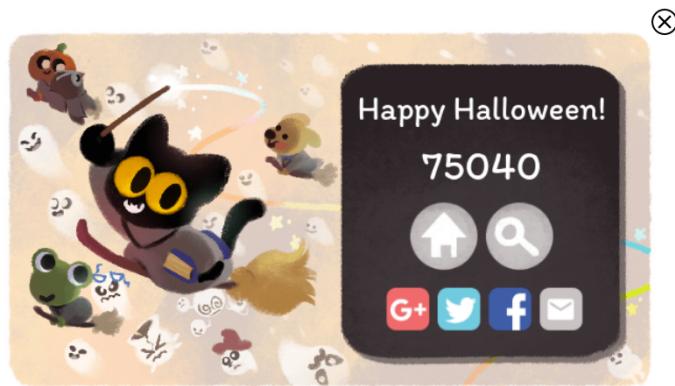


Figura 2.16: Pantalla que aparece al completar el juego

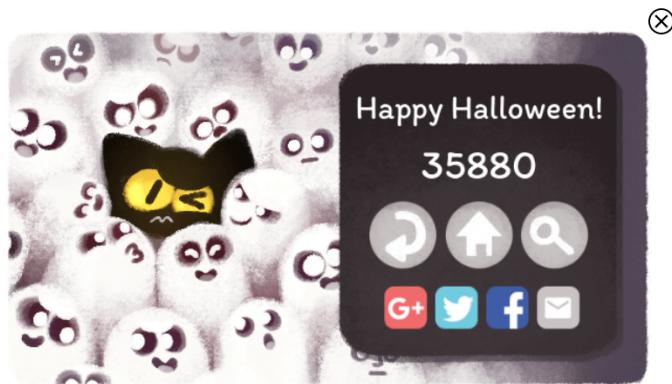


Figura 2.17: Pantalla que aparece al perder todas tus vidas.

Como curiosidad, el juego fue desarrollado por 4 grupos de aproximadamente 4 o 5 personas cada uno. Constaban de un grupo de arte, otro de producción, uno de ingenieros y uno de “ayuda extra” donde se realizaba la música.

Ahora bien, hemos elegido este juego como **principal referente** en cuanto a las **mecánicas**. Diseñaremos el juego de modo que la esencia y jugabilidad sean prácticamente las mismas. Creemos que un juego de estas características es algo sencillo pero que puede ser muy **escalable**, facilitando el desarrollo por iteraciones. También es **divertido** y un candidato perfecto para la NDS ya que parece estar hecho para jugarlo con una **pantalla táctil**.

2.4.2 Una pausa con... Brain Training Ciencias

Una pausa con... Brain Training o también conocido como Brain Age Express en América, se trata de una serie **tres juegos educacionales de resolver puzzles** desarrollados por Nintendo y publicados en el servicio **DSi Ware** de manera **gratuita** en 2009 en Europa.

Estos tres juegos se tratan de un **recopilatorio** de los puzzles que más les habían gustado a los jugadores de los **títulos previos** como Brain Training ¿Cuántos años tiene tu cerebro? y su secuela Más Brain Training para NDS. No obstante, le daban una vuelta de tuerca a estos puzzles haciéndolos **más desafiantes y divertidos**, incluso integraban el uso de la **cámara** al ser un proyecto para DSi.



Figura 2.18: Menú principal de Una pausa con... Brain Training Ciencias

Fuente: Nintendo

Todos los títulos de esta saga tienen un mismo **objetivo**: jugar unos minutos **diariamente** a una serie de puzzles de **cálculo, memorización de imágenes o datos** e incluso ejercicios de **ortografía y lectura**, con el fin de **ejercitar el cerebro**. No obstante, también proporciona la opción de jugar a dichos puzzles de manera individual e incluso tienen un **modo sudoku**.

Volviendo a nuestro referente, como hemos comentado se trata de una serie de tres juegos divididos en **categorías de ciencias, letras y sudoku**. Por una razón que no se hizo oficial, Nintendo quitó del servicio de DSi Ware la edición de sudokus, mientras que las otras dos se mantuvieron e incluso venían preinstaladas en las NDSi americanas. En concreto nos vamos a centrar en la edición de ciencia, pues hay un puzzle particular que tiene bastante relación con nuestro juego: el **Suma y Sigue**.



Figura 2.19: Minijuego Suma y Sigue en su modo normal

Fuente: Nintendo

En este minijuego debemos realizar la **suma** de los **números** que veremos esparcidos por la pantalla y escribir el resultado en la pantalla táctil **lo antes posible**. Posee **dos modos de juego**, uno normal que es el que acabamos de explicar y otro **arcade**, el que nos interesa. El objetivo del modo arcade es exactamente el mismo, realizar la suma de números, pero con el añadido de que los números ahora no están colocados aleatoriamente en la pantalla, si no forman el cuerpo de un **enemigo** que se va acercando hacia el jugador para quitarle vida. Este modo otorga un factor más **desafiante** al juego ya que en el primer modo si fallamos la operación obtendremos peor resultado en la prueba, mientras que en el modo arcade al tener un máximo de tres vidas y ver cómo los enemigos se acercan provoca **presión** en el jugador.



Figura 2.20: Minijuego Suma y Sigue en su modo arcade
Fuente: Nintendo

La razón por la que se ha elegido este minijuego como referente es por poseer algunas **características adicionales** que Magic Cat Academy no posee con respecto a los enemigos y que podrían aportar más diversión si se **mezclan**. Por ejemplo, los enemigos a veces se esconden haciendo que **no se puedan ver** parte de sus números, también dichos números **se mueven** alrededor del cuerpo de dicho enemigo, haciendo que sea más complicado realizar el cálculo.

2.4.3 Lost Magic

Lost Magic es un juego **RPG de estrategia en tiempo real** desarrollado por **Taito** para NDS y publicado en 2006.

En este juego seremos el mago Isaac, cuyo padre le otorga una varita poderosa con la cual lanzar hechizos potentes. Isaac deberá avanzar en la historia **derrotando unos enemigos** en un **tiempo determinado**. Para ello, puede moverse por el mapa y al encontrarse a un enemigo mantener pulsado el botón L, esto abrirá un menú donde puede invocar hechizos dibujándolos en la pantalla táctil, siempre y cuando antes los haya aprendido. Una vez dibujado el hechizo simplemente debe tocar a qué enemigo desea lanzárselo y cuanto **más rápido y mejor lo dibujemos**, más **efectivo** será. Por último, para poder realizar estos

hechizos necesita **poder mágico**, cada vez que realice uno este disminuirá y para que vuelva a aumentar simplemente debemos esperar unos segundos.



Figura 2.21: Principal mecánica del Lost Magic. Arriba vemos los conjuros aprendidos y abajo la pantalla táctil podemos dibujarlos.

Como muchos RPGs, el jugador también irá subiendo de nivel, mejorando sus características mágicas, y obteniendo objetos.



Figura 2.22: Elementos del juego típicos de un RPG.

La razón por la que hemos elegido este juego es por estudiar cómo **implementa el dibujado en tiempo real de los patrones** de hechizos en la pantalla de la NDS y el **reconocimiento** de estos. Al tratarse de un juego para la **misma consola** para la que estamos desarrollando y estar desarrollado por una gran empresa como es Taito, podemos hacernos una idea del **resultado** que podríamos obtener y las limitaciones que existen.

2.4.4 Wario Ware: Touched!

Wario Ware: Touched! es un juego basado en **minijuegos** de la saga Wario Ware que **Nintendo** desarrolló para la NDS en 2004.

En este juego tendremos distintos **niveles representados por personajes** tales como Wario, Mona, Kat y Ana, Ashley... En **cada nivel** debemos pasarnos una **serie de minijuegos muy cortos en un tiempo determinado** (unos 15 segundos) que suelen ser de dibujar algo en la pantalla táctil, soplar al micrófono, etc. Si no lo logramos, **perderemos una vida**, lo cual es muy importante pues al pasarnos una serie de minijuegos llegará el **minijuego de final de nivel** que deberemos completar para completarlo.



Figura 2.23: Pantalla de cambio entre minijuegos y minijuego específico de WarioWare: Touched!.

La razón de escoger este juego como referente es la misma que la de Lost Magic. Me gustaría estudiar cómo es capaz de **dibujar en tiempo real trazados del usuario** así como gran cantidad de *sprites* y fondos en pantalla que aparecen y desaparecen en cuestión de segundos, **optimizando así los recursos de la consola**. También, personalmente me gusta mucho el **estilo del arte** en los *sprites* y diseño de personajes ya que son bastante carismáticos y únicos, y en cierto modo me gustaría inspirarme en ello para poder crear los de este proyecto.

3 Metodología

La metodología que se ha usado para llevar a cabo el proyecto se basa en las **metodologías ágiles** como **Scrum**.

Este tipo de metodologías suelen aplicarse en **proyectos que se realizan en grupo** para así separar bien las tareas, que todos los integrantes puedan trabajar desacopladamente e ir obteniendo resultados lo antes posible. No obstante, este proyecto se trata de un trabajo que se va a realizar únicamente por una persona, pero aún así sigue siendo una buena idea aplicar este tipo de metodologías ya que este es un proyecto donde queremos **tener un producto lo antes posible**, y que nos permita ir mejorándolo, añadiéndole funcionalidades o características extras que lo mejoren.

Así pues, se ha **adaptado** la metodología Scrum al desarrollo de este videojuego para NDS aplicando un **desarrollo iterativo**. Al principio, se planificó una serie de iteraciones teniendo en cuenta el tiempo disponible, asignándoles los objetivos o tareas generales que se debían realizar en éstas. A pesar de que en las metodologías ágiles las iteraciones son de períodos cortos de tiempo, en este caso se decidió ampliar esos tiempos ya que el equipo de trabajo es de únicamente una persona. No obstante las iteraciones se planeó que duraran como máximo un mes.

Por otro lado, después de cada iteración, se extraía una **conclusión** del trabajo desarrollado en ésta y se valoraba **modificar la planificación inicial** si surgía algún problema.

3.1 Planificación

A continuación se detallan las iteraciones que se establecieron, con las tareas que se iban a realizar en estas:

3.1.1 Iteración 0

- Documentarse sobre cómo desarrollar para NDS.
- Realizar pruebas de dibujado simples que nos den una práctica y conocimiento necesarios para empezar a crear el proyecto.
- Analizar los referentes y realizar un diseño del juego.

3.1.2 Iteración 1 - 4

- Desarrollo del juego, de sus mecánicas y algoritmos necesarios.

3.1.3 Iteración 5 - 6

- Mejoras visuales y de retroalimentación para el usuario.
- Realización de pruebas para encontrar fallos tanto en el funcionamiento como en el diseño.

Por último, para llevar un **seguimiento de la planificación** se ha utilizado la herramienta **Trello**. Se creó un tablero con la listas de tareas generales para especificar las tareas que el proyecto requería, la lista de iteración actual, para especificar qué tareas se debían realizar en esa iteración, la lista WIP¹, que contiene las tareas en las que se están trabajando actualmente, y la lista finalizado, que contiene las que ya se han acabado. Además también tiene una lista de problemas, para ir apuntando los inconvenientes que surjan durante el desarrollo y asegurar así que no se quedaban sin resolver.

Se trata de una herramienta muy cómoda, pues de un simple vistazo puedes saber el estado del proyecto, y, además, es muy cómoda de utilizar.

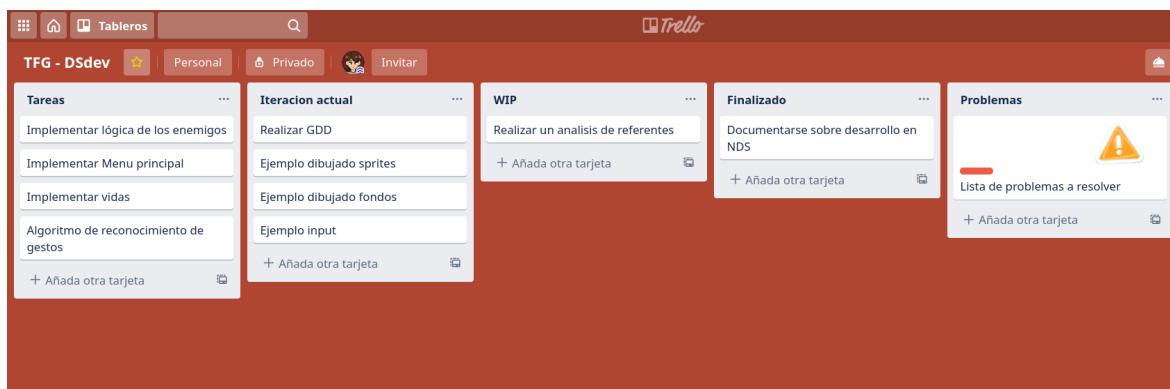


Figura 3.1: Estado del tablero de Trello durante la iteración 0.

3.2 Mínimo producto viable

El diseño realizado en el apartado Diseño del juego está pensado para que, en el mejor de los casos se puedan implementar todas las mecánicas, tipos de enemigos, pantallas, etc. en el tiempo disponible de desarrollo, sin embargo, es poco probable que así sea.

¹WIP: del inglés, *Work In Progress*.

Surgirán problemas, tanto externos como internos, de eso no cabe duda, y no queremos vernos en la situación de estar a pocas semanas de la fecha final y **solo tener un prototipo**. Es por ello que hay que estar preparado para ese problema y definir una versión simplificada, que siga funcionando como juego y sea un **producto cerrado** de principio a fin, algo que la gente pueda probar para ver si le gustan las mecánicas o no y a partir de ahí **trabajar iterativamente** sobre ello añadiéndole las funcionalidades explicadas anteriormente con un **orden de prioridad**.

Así pues, el mínimo producto que se plantea es el siguiente:

Tendrá un **único y primer nivel**, donde los enemigos que aparezcan solo tengan **un patrón** que será al azar entre **dos tipos distintos** (horizontal y vertical).

Se implementará la **muerte** del jugador, así como la **victoria**, de modo que el juego pueda **rejugarse** tantas veces como se quiera. No obstante, no sería necesario implementar aún las respectivas pantallas de fin de juego y victoria, con volver a la pantalla principal bastaría.

Por último carecería de animaciones y sonido, pues implementar las **mecánicas** es **prioritario** frente a perfeccionar lo visual. Sin embargo, sí que se implementará el dibujado del **rastro del lápiz en la pantalla táctil**, así nos servirá como **método de depurado** en caso de que el reconocimiento de gestos nos de problemas.

En la siguiente imagen se muestra un mockup de lo que sería el mínimo producto viable:

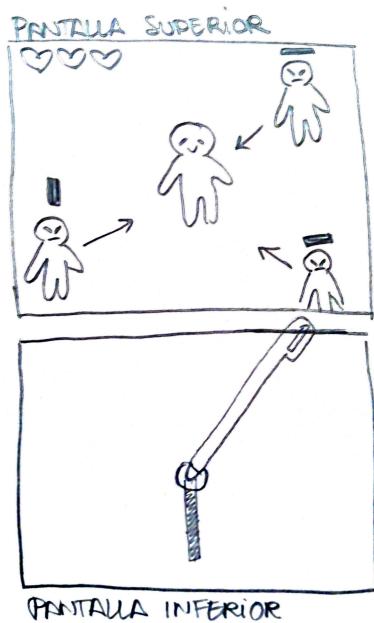


Figura 3.2: Concepto del mínimo producto viable.

4 Diseño del juego (Game Design Document)

4.0.1 Características

- **Título:** Touch & Brush.
- **Plataforma:** Familia de consolas Nintendo DS.
- **Género:** Agilidad mental.
- **Audiencia:** Todas las edades.
- **Idioma:** Inglés.

4.0.2 Historia

Un día cualquiera en un **museo de arte tradicional** sucedió una catástrofe. Un malvado pincel robó toda la pintura de los **cuadros** de la exposición haciendo que estos se volviesen **furiosos**, se separasen de las paredes que los sostenían y empezaran a atacar a la gente.

Cherry, una pequeña niña que disfrutaba de una agradable excursión al museo ese día con su clase, al ver la situación de pánico decide actuar. Es entonces cuando se encuentra con otro pincel, llamado **Celio**, que le asegura a la chica que la única manera de devolver los cuadros a la normalidad es pintarles lo que ellos quieran. Así pues, Cherry y Celio deciden trabajar juntos para salvar el museo.

4.0.3 Ambientación y estilo

Al tratarse de un juego 2D para la consola Nintendo DS, el estilo que adoptarán los escenarios y personajes de Touch & Brush será **pixelart¹** y **cartoon²**, ya que el tamaño de nuestra pantalla y sprites es reducido.

El juego se desarrolla en el espacio cerrado de un museo de arte, así que el fondo principal donde se desarrolla el gameplay se trata de una de sus salas. Esta misma poseerá las características más típicas de un museo de arte, como puede ser un reluciente parqué, una tenué

¹Estilo de arte digital que trabaja con las imágenes a nivel de pixel.

²Estilo de dibujo y estilización de personajes de personajes con proporciones irreales de tal manera que llega incluso a tener un carácter humorístico y simpático.

iluminación con puntos de luz hacia la pared donde deberían ir los cuadros y un fondo azul oscuro para que los sprites y elementos por encima de éste resalten sin problema.



Figura 4.1: Fondo de la pantalla de juego que representa la sala de un museo vacía y destrozada.

Tanto la protagonista como los enemigos tendrán características desproporcionadas tales como gran cabeza, ojos y extremidades, típicas de la estética cartoon. Además, al tratarse de una historia de fantasía y magia, los sprites del juego poseerán **gran variedad de colores** y bastante saturados, para destacar sobre el fondo y llamar la atención del jugador.



Figura 4.2: Concept art de Cherry, la protagonista del juego, los cuadros.

Un toque que posee la ambientación Touch & Brush y que ayuda a **unificar el estilo visual** de todo el producto es el uso de **manchas de pintura de color arcoíris** tratando de simular una pintura mágica. El uso de este recurso se utiliza en diversas ocasiones como por ejemplo, en el pincel Celio, en el logo del juego, en el texto de la pantalla del título y en la mecánica de dibujar en la pantalla táctil y en el propio fondo del museo. Por último, a pesar de que para los gráficos no usaremos una paleta de colores específica, sino más bien todos los que nos permita el hardware, lo que sí haremos será usar el mismo tono lila para hacer las sombras de sprites y fondos. Esto ayudará a cohesionar todos los gráficos y dotar al juego de una consistencia visual.



Figura 4.3: Sprites de Cherry y los principales enemigos.

4.0.4 Jugabilidad y mecánicas

Para **progresar** en los niveles de Touch & Brush, el jugador deberá ir **dibujando el patrón** o conjunto de patrones que los **enemigos** poseen encima de ellos antes de que éstos lleguen hasta él. El jugador se encontrará en el **centro superior de la pantalla** y los enemigos aparecerán por los bordes y, al **alcanzarle**, le **quitarán una vida**. Este mismo poseerá **6 vidas** que podrá consultar en todo momento ya que se visualizarán en un **contador de vidas** en la parte superior izquierda de la pantalla.

Como algunos enemigos se moverán más lento que otros, el reto del jugador es poder decidir rápidamente qué patrones dibujar y en qué orden para avanzar al siguiente nivel con el mayor número de vidas posibles. Además, si somos capaces de **concatenar varios patrones correctos seguidos** aumentará nuestra **puntuación**, también visible en cualquier momento en la parte superior derecha de la pantalla.

Por otro lado, si nos encontramos en un nivel de una dificultad superior y hemos perdido más de la mitad de nuestra vida, aparecerá un **aliado** con un **patrón de corazón**. Si lo

dibujamos correctamente nos aumentará la vida en una unidad.

Al llegar al último nivel encontraremos al **jefe final**, que debemos derrotar de manera similar a los enemigos normales pero con una serie de fases. Su comportamiento se explicará detalladamente en el apartado de Enemigos.

4.0.5 NPCs

Dentro del juego distinguiremos tres tipos de NPCs, siendo dos de estos enemigos y uno de ellos un aliado. A continuación se detalla cada uno en profundidad.

4.0.5.1 Enemigos comunes

Como hemos comentado, los enemigos comunes del juego se tratan de los cuadros del museo, que al haberles sido arrebatada su pintura han cobrado vida y se han vuelto furiosos.

Este es el único tipo de enemigo que habrá en los niveles normales del juego y su comportamiento siempre será el mismo: **ir hacia el jugador** y al alcanzarle le quitará vida. Sin embargo, entre varios enemigos existirán diferencias como por ejemplo la **velocidad** a la que se mueven, el **tipo de patrón** que debe dibujar el jugador para matarlos y la **posición** desde la que aparecen. Estas características se comentarán más adelante en el apartado de niveles.

Los enemigos podrán aparecer con un patrón de entre **tres** posibles, los cuales se muestran en la siguiente figura. Se tratan de una **línea horizontal**, una **línea vertical** y un **triángulo**. Estos patrones además están representados con **distintos colores** cada uno para que al usuario, a la hora de visualizarlos en el juego, le sea más fácil distinguirlos.

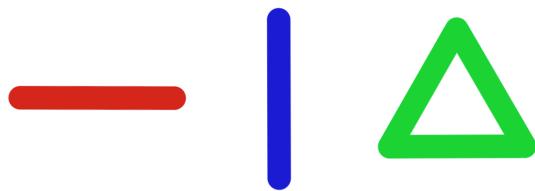


Figura 4.4: Tipos de patrones con los que pueden aparecer los enemigos.

4.0.5.2 Aliado

Este NPC tiene la misma apariencia que un enemigo normal, pues también se trata de un cuadro del museo. Sin embargo, **no es agresivo**, de hecho es beneficioso para el jugador. Su

patrón será un dibujo con forma de corazón y si el usuario consigue dibujarlo cuando este aparezca en un nivel, **aumentará su vida** en una unidad.

Ahora bien, tanto su aparición como su comportamiento no serán igual que los enemigos normales. Este aliado solo aparecerá **una vez por nivel**, siempre a partir del nivel 3 y siempre que nuestra vida sea igual o inferior a la mitad. Además, no correrá hacia el jugador, sino que aparecerá desde el borde izquierdo de la pantalla y **avanzará en línea recta hasta la mitad de la pantalla**, donde se detendrá.



Figura 4.5: Tipos de patrones con los que pueden aparecer los enemigos.

4.0.5.3 Jefe

Este enemigo se trata del jefe final del juego, el cual representa al **cuadro más prestigioso** y valorado del museo. En cuanto a su apariencia, es igual que los enemigos normales pero de mayor tamaño, ojos rojizos y porta una corona de rey.

Se podrá encontrar después de haber superado el último nivel y su comportamiento será distinto. En primer lugar, se encontrará en la parte derecha de la pantalla, avanzando hacia la izquierda donde se encuentra el jugador. También, su daño será mayor, quitando de un solo golpe 2 corazones.

La forma en la que debemos matarlo también será diferente a los demás enemigos. Su comportamiento estará dividido en **3 fases**, y en cada una de ellas generará una **secuencia de 5 patrones** que el jugador deberá dibujar **en orden** mientras dicho jefe se va moviendo a una velocidad concreta hacia este. Si el jugador lo logra, el jefe volverá a su posición inicial, pasando a la siguiente fase, donde aumentará su velocidad y la secuencia de patrones volverá a generarse.

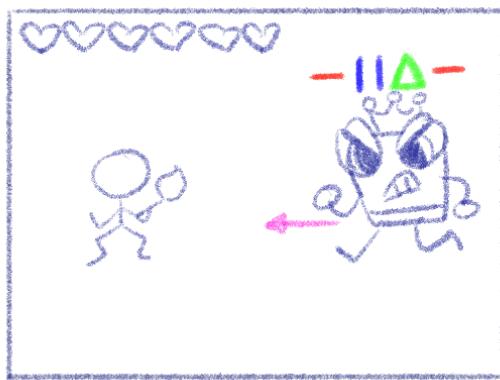


Figura 4.6: Mockup del enfrentamiento con el jefe.

4.0.6 Niveles

El juego constará de **4 niveles normales** y **un nivel de jefe final**.

En cuanto a los niveles normales, visualmente no serán muy distintos entre sí, pero sí que hay **5 parámetros** que podemos ajustar para dotar a los niveles de Touch & Brush de una curva de complejidad y aprendizaje interesantes.

En primer lugar, tenemos el **tiempo** que transcurre entre la aparición de enemigos. A medida que avancemos en los niveles, este tiempo se irá reduciendo para que aparezcan enemigos con más frecuencia y el jugador deba dibujar los patrones más rápido.

Después, tenemos la **posición** desde la que aparecen los enemigos y la **velocidad** a la que se mueven. Al encontrarse el jugador en el centro de la pantalla, los enemigos podrán salir desde 6 posiciones distintas y dirigirse hacia él. Estas posiciones son: esquinas superiores derecha e izquierda, centro derecha e izquierda y esquinas inferiores derecha e izquierda. En cuanto a la velocidad, es interesante que algunos enemigos sean capaces de moverse más rápido que otros para despistar al jugador. Por ejemplo, si sale un enemigo desde un lado a una velocidad lenta el jugador se fijará en ese e intentará derrotarlo, pero entonces es cuando aparece un enemigo que se mueve más rápido, le alcanza antes de que pueda darse cuenta o reaccionar y le quita una vida.

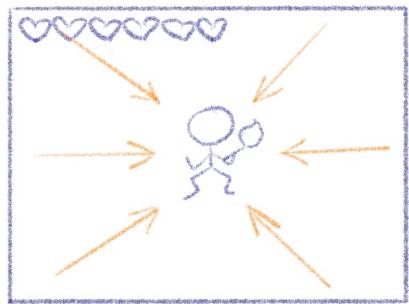


Figura 4.7: Posiciones iniciales desde las que pueden aparecer enemigos.

Por otro lado tenemos el **número de enemigos que debemos derrotar** para pasar al siguiente nivel, que irá aumentando según avancemos. Y por último, tenemos la **complejidad de los patrones** de los enemigos. Es bastante más fácil de dibujar una línea recta que un triángulo, así que los primeros niveles comenzarán con esos patrones más simples y a medida que avancemos se incorporarán los más complejos.

Así pues, una vez explicado esto, los 4 niveles de Touch & Brush tendrían las siguientes características:

- **Nivel 1:** Enemigos lentos que únicamente aparecen por los laterales centrales de la pantalla, moviéndose lento y sus patrones serán la línea horizontal y vertical. Será necesario derrotar aun total de 6 enemigos para pasar al siguiente nivel y la frecuencia con la que estos aparecen será la más baja.
- **Nivel 2:** En este nivel se incorporan los enemigos que salen desde las esquinas superiores e inferiores, llevando estos una velocidad mayor. Además, el número de enemigos a derrotar aumenta a 10 así como aumenta también la frecuencia con la que éstos aparecen.
- **Nivel 3:** Se mantienen las mismas posiciones y velocidades de los enemigos, así como sus patrones. Aumenta el número de enemigos que derrotar a 13 y también lo hace la frecuencia con la que éstos aparecen. Se incorpora el aliado que te permite recuperar vida.
- **Nivel 4:** Nuevamente, aumenta el número de enemigos que derrotar a 16 y la frecuencia con la que aparecen. Además también aparecen los enemigos con patrón de triángulo.

4.0.7 Controles

En cuanto a los controles, al tratarse de un juego para una consola con pantalla táctil la gran mayoría de interacciones tanto para el **gameplay principal** como la **selección de opciones en los menús** se harán mediante esta. No obstante, cabe destacar que para **abrir el menú de pausa** se pulsará el botón **START**.

4.0.8 Estados del juego

Aquí veremos un resumen de todos los estados del juego, así como el flujo entre ellos para que a la hora de desarrollar tengamos en cuenta desde qué estados se puede llegar a uno en concreto.

En la siguiente imagen se puede observar dicho resumen, y más adelante veremos las pantallas que se corresponden con cada estado.

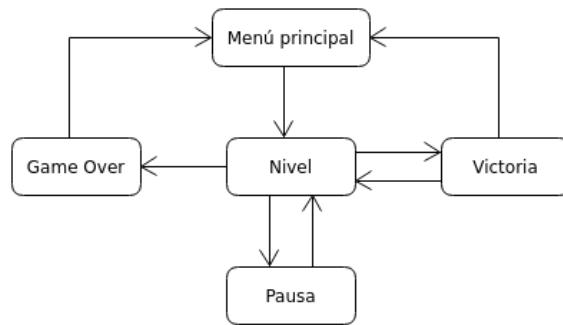


Figura 4.8: Diagrama de flujo entre estados

Al iniciar el juego se cargará el **menú principal**, que nos dará la opción de empezar a jugar desde el primer nivel ya que no habrá guardado de partida. Si completamos un nivel, pasaremos al estado de **victoria**, que nos mostrará un mensaje de que hemos completado el nivel y nos dará la opción tanto de volver a jugar ese nivel, continuar al siguiente o volver al menú principal.

Por otro lado, si mientras jugamos perdemos, independientemente del nivel en el que nos encontrasemos pasaremos al estado de **fin de partida**, que nos mostrará un mensaje de que hemos muerto, la puntuación obtenida y únicamente nos permitirá volver al menú principal.

Por último, mientras estemos jugando a cualquier nivel podremos cambiar al **estado de pausa** si deseamos dejar en espera nuestro juego y así mismo desde este menú de pausa podremos volver al estado donde dejamos la partida.

4.0.9 Pantallas

A continuación se muestran una serie de **bocetos de pantallas** asociadas a los distintos estados del juego, así como también los elementos visuales que estos poseen, qué información nos dan y si son interactivos.

En primer lugar, la pantalla del menú principal. En ella no habrán muchos elementos,

teniendo en la pantalla superior el logo del juego y, en la inferior un mensaje que nos indicará que debemos tocar la pantalla para comenzar a jugar. Además, en esta misma pantalla también se mostrará el nombre del autor del juego.



Figura 4.9: Mockup de la pantalla de título

Por otro lado tenemos la pantalla de nivel, siendo esta donde el jugador más tiempo pasará y más información le aportará. En la pantalla superior, haciendo uso de sprites se podrán visualizar los enemigos moviéndose por la pantalla, el jugador y la vida de éste mismo. Además, en la parte derecha superior tendremos un contador que nos indicará nuestra puntuación y se irá actualizando mientras jugamos.

En la pantalla inferior es donde el jugador irá dibujando los patrones, por ello mediante un fondo simularemos una especie de lienzo y cuando el jugador esté tocando la pantalla táctil iremos dibujando el trazo que éste realiza.



Figura 4.10: Mockup del juego

Después, en cuanto al los estados de victoria o final de partida, ambos son muy similares. En la pantalla superior ambos mostrarán un mensaje indicando si han ganado o perdido, y después un texto con la puntuación obtenida. En la pantalla inferior, habrá un botón común en ambos para volver al menú principal y otro en el caso de la pantalla de victoria que nos llevará al siguiente nivel. Estos botones serán interactuables tocando sobre ellos en la pantalla táctil. Cabe destacar también, que en caso de haberlos pasado el juego entero el botón de pasar al siguiente nivel no estará disponible.

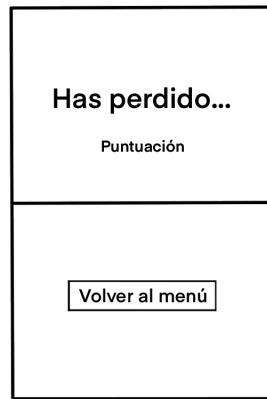


Figura 4.11: Mockup de la pantalla de fin de partida



Figura 4.12: Mockup de la pantalla de victoria

Por último, la pantalla de pausa como ya hemos comentado no aportará gran información al usuario. Esta simplemente le mostrará en la pantalla superior un texto de que ha pausado la partida y en la pantalla inferior un botón para reanudarla.

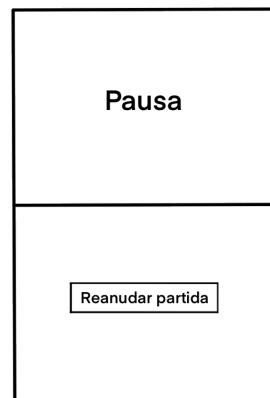


Figura 4.13: Mockup de la pantalla de pausa

5 Desarrollo

En este capítulo se describirá detalladamente el proceso realizado para conseguir el juego que se ha diseñado previamente. Se explicará desde todas las herramientas que se necesitan hasta el producto final, pasando por unas primeras demos básicas de manejo de la consola y su hardware, mínimo producto viable e iteraciones y problemas del mismo, de esta manera conseguimos que cualquier persona que busque información sobre desarrollo en NDS le sea lo más útil posible y sea capaz de buscar las partes que más le interesan.

5.1 Herramientas

5.1.1 Entorno de desarrollo

El lenguaje de programación que vamos a utilizar es **C++** debido a varios motivos, entre ellos que la librería es para dicho lenguaje y además es el perfecto candidato para dejarnos aprovechar la máquina en su totalidad.

Por último, para programar necesitaremos un editor de texto. En mi caso he utilizado **Sublime Text 3** por comodidad, pero cualquiera es completamente válido.

5.1.2 Emuladores

Para probar el juego que vamos a desarrollar es indispensable disponer de programas que **emulen virtualmente la consola** en nuestro ordenador de trabajo. Esto no solo nos **ahorra el tiempo** que cuesta en probarlo en una máquina real, también nos proporciona un **entorno seguro** donde no importa si metemos demasiado la pata haciendo nuestras pruebas, ya que no hay riesgo alguno de romper una máquina real.

Existen gran cantidad de emuladores de Nintendo DS, pero nos vamos a centrar en dos de ellos.

El primero de ellos es **Desmume**. Este es bastante **cómodo** ya que se encuentra disponible tanto para Windows, Mac y Linux, y también nos permite cambiar los controles y unos pequeños ajustes que nos pueden servir para depurar nuestros programas. Es una muy buena opción si lo que queremos es probar nuestros programas de una manera **rápida** o incluso ejecutar los ejemplos que nos proporciona la librería, de la cual hablaremos más adelante.



Figura 5.1: Desmume emulando WarioWare: Touched!

Por otro lado tenemos **no\$gba**, una excelente opción y sin duda la que más vamos a **necesitar** usar. Permite ejecutar tanto ROMs de GBA como de NDS y además tiene disponible una **versión debug** con muchísimas más opciones que Desmume. Entre ellas un visualizador de toda la memoria de la consola así como herramientas de **depurado visual** para **fondos, sprites y paletas**.

5.1. HERRAMIENTAS

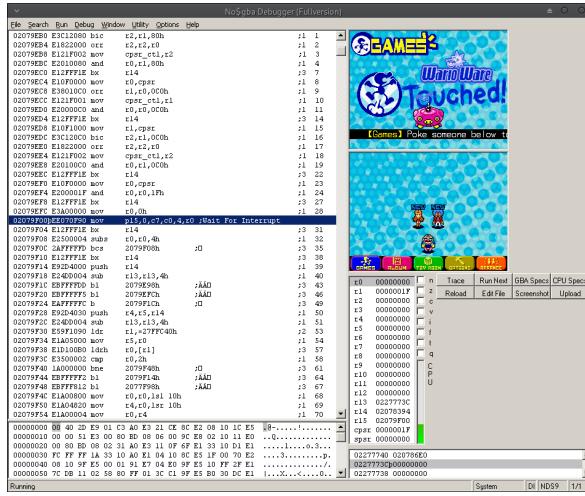


Figura 5.2: Versión debug de no\$gba emulando WarioWare: Touched!

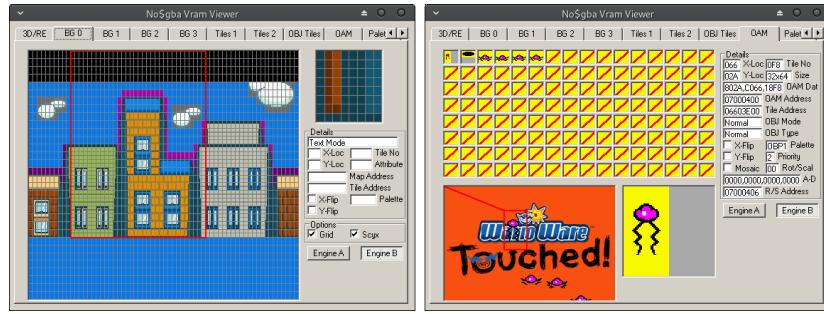


Figura 5.3: Herramienta de depurado para fondos (izquierda) y sprites (derecha) de no\$gba

Como única pega es que **solo** está **disponible** para el sistema operativo **Windows**, pero sin mucho problema podemos hacer que funcione en Linux gracias al uso de la herramienta **Wine**.

5.1.3 Cartuchos Flash

Aunque dispongamos de los emuladores para la gran mayoría del trabajo, es imprescindible probar los resultados de vez en cuando en una **máquina real**. Para ello evidentemente deberemos disponer de la consola en sí, pero también necesitaremos un **cartucho** al cual introduciremos **nuestro programa** y éste será ejecutado por la consola. Estos son los **cartuchos flash**.

Se denominan de esa forma ya que poseen una memoria flash, que permite **simultáneas**

lecturas y escrituras en una misma operación. Esta tecnología se emplea en las unidades de **almacenamiento externos** (USB) o **discos sólidos** (SSD)

A continuación vamos a centrarnos en una marca de cartucho flash concreta, ya que es la que poseemos para desarrollar el proyecto.

5.1.3.1 R4

Revolution for NDS o **R4 DS** nació principalmente para la **piratería** de juegos de NDS o para ejecutar aplicaciones ilegales en ella. Se trata de un cartucho similar a los originales de los juegos de NDS con una **ranura** para una tarjeta **MicroSD**, a la cual nosotros copiaríamos nuestro fichero binario con extensión nds y eso bastaría para probar el juego. Posee también una **interfaz agradable**, permitiendo también la **reproducción de contenidos multimedia** en una sección de esta.



Figura 5.4: Cartucho R4

5.1.4 DevkitPro

DevkitPro es una **colección** de gran cantidad de **herramientas** de ayuda para **desarrollar** en sistemas como NDS, GBA, GameCube, Wii, Nintento Switch... Digamos que es como Adobe. Nosotros no usamos Adobe para editar imágenes digitales, en su lugar usamos Photoshop, desarrollado por este. En nuestro caso, nosotros usaremos una herramienta de devkitPro para poder **compilar** nuestro programa y **generar los binarios** adecuados para ejecutarlos en los procesadores ARM de la consola, esta herramienta es **devkitARM**.

5.1.5 Libnds

La librería que utilizaremos es **libnds**. Inicialmente fue desarrollada por Michael Noland (alias joat) y Jason Rogers (alias rovoto), pero después de unos meses pasó a manos de Dave Murphy (alias WinterMute), quien a día de hoy es la principal persona que la mantiene.

Libnds empezó siendo un simple **conjunto de macros y definiciones** de zonas de memoria de **recurrente acceso** para **facilitar** el uso y legibilidad del código a los **programadores**. Por ejemplo, es lo mismo escribir SPRITE_GFX que 0x6400000, ambas hacen referencia al lugar en memoria donde comienza el almacenamiento de datos referente a los sprites, pero la primera es sin duda **más limpia y fácil de entender**, y sobre todo le quita peso al programador ya que no le obliga a conocerse todas estas posiciones con exactitud.

Más tarde y según los desarrolladores homebrew comenzaban a usar libnds, se le empezaron a **añadir elementos** de gran utilidad, como por ejemplo **estructuras de datos** o incluso **APIs para crear sprites** y alocarlos en memoria, simplificando el trabajo del programador a simplemente llamar a un par de **funciones**. A día de hoy cubre gran cantidad de aspectos como gráficos, sonido, pulsaciones de botones y pantalla táctil, operaciones matemáticas complejas... y es usada por el 90% de la comunidad.

5.1.6 Grit

GBA Raster Image Transmogrifier o Grit para los amigos, es una herramienta de **conversión de mapas de bits para el desarrollo en GBA y NDS**. Es decir, es una herramienta que nos ayudará a **convertir nuestras imágenes** de fondos, sprites o tiles en algo que la **máquina** pueda **entender**.

Acepta muchos tipos de **formatos** de imágenes como BMP, PNG, GIF, JPG... y también nos permite **trabajar con ellas** para cambiar la profundidad de bit, separar nuestros fondos en tiles de manera rápida, soporte para transparencias y mucho más.

Lo que más nos va a interesar a parte de que nos convierta las imágenes en binarios que la NDS entienda, es que además nos proporciona un **archivo de cabecera** con definiciones que podemos **usar en nuestro programa** como el tamaño de la imagen en bytes, anchura y altura y e información relativa a su paleta. Esto ya lo veremos más adelante en el desarrollo, pero ya adelanto que es una herramienta muy útil y nos va a ahorrar muchos problemas.

Aunque grit compila en todas las plataformas, para sistemas que no se traten de Windows debemos **crearnos nuestro propio makefile**. Esto no será mucho problema, pues en los **ejemplos de libnds** ya vienen makefiles que incluyen el uso de esta herramienta. En **Windows** disponemos de la versión **Wingrit** con una interfaz que nos permitirá trabajar con la imagen de manera más sencilla y visual. Para el resto de sistemas operativos tenemos la **versión por comandos**, que es la que utilizaremos nosotros la mayor parte del tiempo. Una vez instalado grit en nuestro sistema (Linux Manjaro en este caso), podemos abrir una

terminal y ejecutar el comando "grit" para leer una pequeña descripción de todos los comandos que se encuentran disponibles.

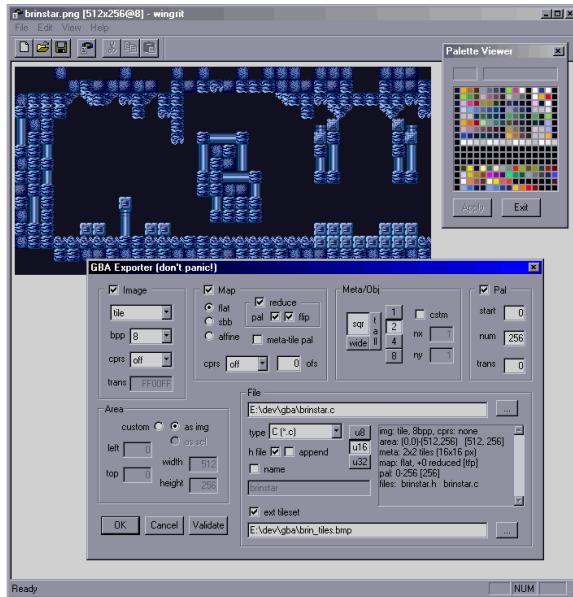


Figura 5.5: Wingrit, versión GUI para Windows de Grit

Fuente: Coranac

```

Terminal - shiro@shiro-pc:~
Archivo Editar Ver Terminal Pestañas Ayuda
[shiro@shiro-pc ~]$ grit
---grit v0.8.15 ---
GRIT: GBA Raster Image Transmogrifier. (grit v0.8.15)
    Converts bitmap files into something the GBA can use.
usage: grit srcfile(s) [args]

--- Graphics options (base: "-g") ---
-g | -g!      Include or exclude gfx data [inc]
-gu{8|16|32}   Gfx data type: u8, u16, u32 [u32]
-gz{lhr0}     Gfx compression: off, lz77, huff, RLE, off+header [off]
-gb{fmt}      Gfx format, bitmap or tile [tile]
-gb{fmt}      Gfx format / bit depth (1, 2, 4, 8, 16, a5i3, a3i5) [img bpp]
-gx            Enable texture operations
-gS            Shared graphics
-gT{n}        Transparent color; rrgbb hex or 16bit BGR hex [FF00FF]
-aL{n}        Area left [0]
-aR{n}        Area right (exclusive) [img width]
-aW{n}        Area width [img width]. Overrides -ar
-aT{n}        Area top [0]
-aB{n}        Area bottom (exclusive) [img height]
-aH{n}        Area height [img height]. Overrides -ab

--- Map options (base: "-m") ---
-m | -m!      Include or exclude map data [exc]
-mu{8|16|32}   Map data type: u8, u16, u32 [u16]
-mz{lhr0}     Map compression: off, lz77, huff, RLE, off+header [off]
-ma{n}        Map-entry offset n (non-zero entries) [0]
-mp{n}        NEW: Force mapsel palette to n
-mB{n}:((iphv[n])) NEW: Custom mapsel bitformat
-mR{t,p,f}    Tile reduction: (t)iles, (p)al, (f)lipped

```

Figura 5.6: Grit versión comandos

Además, se trata de un proyecto *open-source*, por lo que si poseemos del conocimiento adecuado podemos añadir nuestro propio código para personalizar el uso.

5.2 Iteraciones

5.2.1 Iteración 0 - Investigación previa y primeros pasos en NDS

Esta primera iteración se dedicó a documentarse sobre el desarrollo en NDS, la consola y su hardware, así como conocer todas las herramientas necesarias para comenzar a trabajar en el proyecto. También se realizaron una serie de pruebas para aprender cómo pintar gráficos en pantalla. A esta iteración se le dedicó especialmente más tiempo ya que era la que quería conocer con mayor detalle, pues luego se quería plasmar bien en esta memoria esos pasos de modo que se pudiese entender de la forma más clara posible.

5.2.1.1 Fondos

Una vez ya tenemos todas las herramientas necesarias podemos ponernos manos a la obra a programar. Sin embargo, antes de ponernos a programar nuestros juegos vamos a realizar una serie de pruebas y así ir conociendo el hardware de la NDS. En concreto, en esta sección vamos a aprender a **dibujar fondos en ambas pantallas**.

Primero que todo vamos a crear la carpeta del proyecto. Esta carpeta se llamará "Fondos", y dentro contendrá una serie de carpetas dedicadas a distintos ficheros. En una carpeta "gfx"

almacenaremos todas las imágenes del juego. En una carpeta "include", introduciremos los archivos de cabecera y, en una carpeta "source" todos los archivos .cpp del proyecto. Esto lo haremos así ya que vamos a usar el Makefile que viene en los ejemplos de devkitPro para compilar nuestro proyecto y que nos genere un archivo ejecutable. Este sería el aspecto de la carpeta:

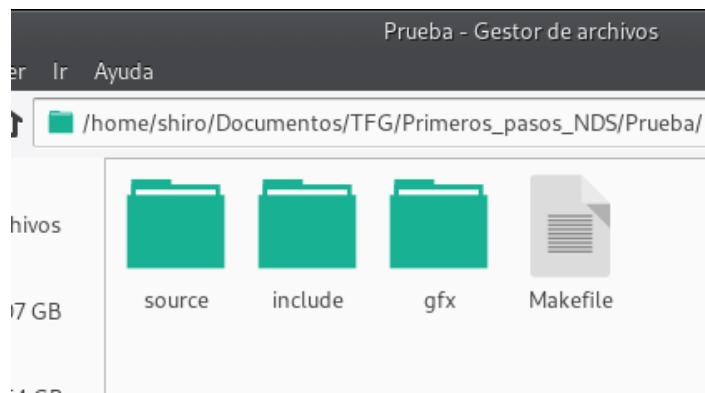


Figura 5.7: Carpeta del proyecto

Una vez lo tenemos, creamos el archivo main.cpp, que contendrá el bucle principal del juego. Necesitamos mantener la consola este ya que de otro modo nuestro programa llegará a su fin, la consola se reiniciará y no nos dará tiempo a ver los resultados de nuestras pruebas. Nuestro main contendrá el siguiente código:

Código 5.1: Bucle principal del juego

```

1
2 #include <n3ds.h>
3
4 int main(void){
5     while(1){ //bucle infinito
6         //actualizamos el juego
7     }
8 }
```

Como se puede ver en el **Anexo I.**, el hardware de esta consola posee **dos motores gráficos**, uno para **cada pantalla**. Por lo general, libn3ds asigna el primer motor (al que llamaremos **motor A**) a la **pantalla superior** y el segundo (**motor B**) a la **pantalla táctil**, pero nosotros podemos cambiarlo si así lo deseamos.

De entre todos los modos en los que puede operar cada motor, el que más nos interesa es el **modo 5**. Este modo permite que dos de los cuatro fondos disponibles en cada pantalla sean del tipo "*Extended Rotation Background*". Esto quiere decir que tienen la capacidad de ser **rotados, escalados o trasladados** mediante una **matriz afín** de transformaciones, que

veremos más adelante.

Vamos a comenzar preparando ambos motores para usarlos, para ello libnds nos lo pone muy simple ofreciéndonos la función powerOn(), que "enciende" un hardware específico. En este caso bastaría con simplemente enviarle como parámetro POWER_ALL_2D.

Después debemos realizar el mapeo de la memoria de video para que la consola sepa en qué zonas debe copiar los datos de los fondos. Como ya se ha comentado anteriormente, la NDS posee 9 bancos de VRAM, en este caso vamos a mapear el banco A como almacenamiento de fondos para la pantalla superior y el banco C para la inferior, pero podemos asignarlos a nuestro gusto. Para ello, usaremos las funciones vramSetBank de libnds.

Código 5.2: Mapeo de memoria de video

```
1 vramSetBankA(VRAM_A_MAIN_BG_0x06000000);
2 vramSetBankC(VRAM_C_SUB_BG_0x06000000);
```

Una vez hecho esto debemos establecer a ambos motores el modo de vídeo deseado, así como especificarles las capas de fondos que van a estar activas. Esto lo haremos con las funciones videoSetMode y videoSetModeSub.

Código 5.3: Bucle principal del juego

```
1 videoSetMode(MODE_5_2D | DISPLAY_BG3_ACTIVE);
2 videoSetModeSub(MODE_5_2D | DISPLAY_BG3_ACTIVE);
```

Casi hemos terminado preparando los fondos para dibujar, pero aún debemos dar valor a una serie de registros de control específicos de cada fondo para indicarle qué datos le vamos a copiar, dónde y cómo los posiciona en pantalla.

Código 5.4: Bucle principal del juego

```
1 REG_BG3CNT = BG_BMP16_256x256 | BG_BMP_BASE(0) | BG_PRIORITY(3);
```

En concreto, en esta última línea le hemos proporcionado la siguiente información:

- **BG_BMP16_256x256:** El fondo sera un bitmap de 16 bits de profundidad y de 256 píxeles de ancho y alto como máximo.
- **BG_BMP_BASE(0):** Lugar en memoria donde se localizará el fondo.
- **BG_PRIORITY(3):** Le asigna el nivel 3 de prioridad. La prioridad en los fondos va de 0 (encima) a 3 (debajo) y se usa para decidir qué fondos se visualizan por encima de los demás.

Ahora volvemos al tema que habíamos dejado antes, la matriz afín. Los registros que vienen a continuación sirven para modificar esa matriz y aplicar así transformaciones a los fondos. Como de momento no queremos aplicarle transformación alguna, lo ideal es que la dejemos como una matriz identidad.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figura 5.8: Matriz identidad

Código 5.5: Registros de control para la matriz afín de los fondos

```

1 REG_BG3_PA = 1 << 8;
2 REG_BG3_PB = 0;
3 REG_BG3_PC = 0;
4 REG_BG3_PD = 1 << 8;
```

Como podemos ver, en los registros donde van los unos a continuación aparece “«“. Este es el operador desplazamiento, sirve para desplazar los bits en el sentido al que apuntan tanto como indique el valor de la derecha. En ese caso, el 0000000000000001 (en binario) que asignabamos a los registros pasa a ser 0000000100000000. Pero, ¿para qué queremos eso?

Esto tiene que ver con cómo interpreta la NDS los bits. En el sistema binario hay varias formas de representar un número, como la coma fija. Esta nos permite representar números fraccionarios, lo cual nos interesa al para poder expresar senos o cosenos que nos servirán para rotar la imagen.

Normal	$2^{15} 2^{14} 2^{13} 2^{12} 2^{11} 2^{10} 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$
Coma fija	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} 2^{-6} 2^{-7} 2^{-8}$

Figura 5.9: Diferencias entre la representación normal binaria y la coma fija

Lo último que faltaría para ver nuestros fondos en la consola sería copiar los bytes de las imágenes a la zona de memoria que corresponde. Aquí es donde entra Grit para facilitarnos

el trabajo. En la carpeta gfx copiaremos nuestra imagen, así como también incluiremos un archivo con extensión .grit que contendrá lo siguiente:

Código 5.6: Comandos de grit por archivo

```

1 #Nombre del símbolo. Indica el nombre dentro de nuestro programa
2 -s ground
3
4 # Warning/log level. [1-3] Muestra todos los mensajes de error o warnings. Nos ↪
  ↪ interesa tenerlos activados para depurar.
5 -W3
6
7 # Indica si el fondo tiene transparencia. Si no tiene -> !. Si tiene indicar ↪
  ↪ el valor en hexadecimal del color transparente.
8 -gT!
9
10 # Bitmap image. Indica si es un mapa de bits o tiles, en este caso de bits
11 -gb
12
13 # Bit depth. Indica la profundidad de bit
14 -gB16

```

Así pues, al compilar grit realizará la conversión de todas las imágenes que se encuentren en la carpeta gfx de nuestro proyecto. Esta conversión se hará utilizando las reglas que se establecen en el archivo anterior, pero si no existe uno se aplicarán las de por defecto. Por último creará una cabecera con datos de nuestra imagen.

Con esto ya solo nos faltaría copiar los bytes de la imagen a la memoria de video de la consola. Esto lo haremos usando la función de C++ memcpy.

Código 5.7: Copia de la imagen en la memoria de video

```

1 memcpy((uint16*)BG_BMP_RAM(0), groundBitmap, groundBitmapLen);
2           //destino //origen //tamano a copiar

```

BG_BMP_RAM(BASE) es una macro que dado un valor en BASE calcula una dirección de memoria. Por ejemplo, para BASE igual a 2, calculará $2 * 4000$ y el resultado se lo sumará a 6000000 (todas las operaciones en base 16, es decir, hexadecimal). Como nosotros queremos copiar nuestro fondo en 0x06000000 como valor a BASE tenemos que pasarle un 0. Sin embargo, si en lugar del banco A hubiésemos elegido el banco B (0x06020000) debería valer 8 ($8 * 4000 + 6000000 = 0x06020000$).

Ya podemos compilar nuestro programa y, si todo ha salido bien deberíamos tener un resultado como el siguiente:

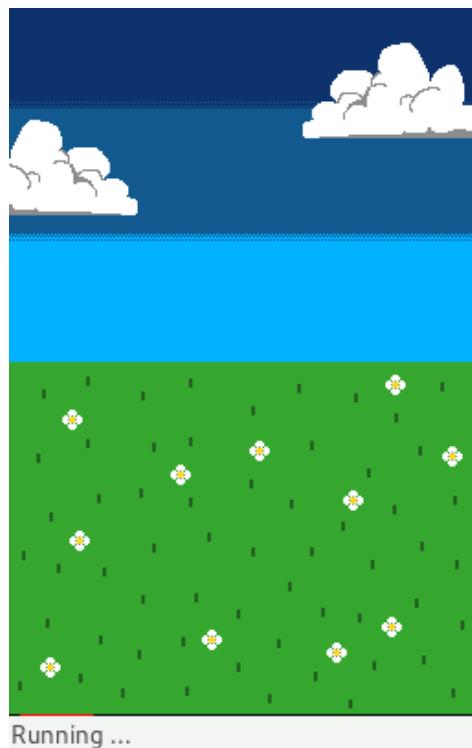


Figura 5.10: Resultado en el emulador Desmume

5.2.1.2 Sprites

A la hora de dibujar sprites debemos realizar una serie de tareas adicionales, pues vamos a trabajar con la OAM.

Lo primero es, como anteriormente, mapear un banco de memoria de vídeo a memoria para sprites. Esto se puede hacer con la función `vramSetBankE()` y la macro `VRAM_E_MAIN_SPRITE`. Además, debemos especificar también en el modo de vídeo que se visualicen los sprites y que su forma de guardarlos en memoria va a ser lineal (`DISPLAY_SPR_1D`).

Código 5.8: Mapeo del banco E a memoria de sprites

```

1  vramSetBankE(VRAM_E_MAIN_SPRITE);
2  [...]
3  videoSetMode(MODE_5_2D | DISPLAY_SPR_ACTIVE | DISPLAY_SPR_1D);
```

Una vez hecho esto, debemos copiar la información del sprite a memoria, tal y como hacíamos con los fondos, con el añadido de que ahora sí necesitamos copiar también la paleta. Grit también nos proporciona la paleta del sprite, siempre que le pasemos el parámetro `-p`. En el archivo de cabecera del sprite tenemos las variables `spritePalLen` como el tamaño que ocupa la paleta y `spritePal`, un puntero a dónde comienza esta.

Código 5.9: Copia de la información de sprites a memoria de video

```
1 memcpy(&SPRITE_GFX[0], spriteTiles, spriteTilesLen);
2 memcpy(&SPRITE_PALETTE[0], spritePal, eyes_1PalLen);
```

Una vez hemos hecho esto, ya podemos trabajar con la OAM para crear nuestros sprites.

La OAM está formada objetos de tipo SpriteEntry y SpriteRotation. El primero de ellos guarda información relativa al sprite en sí, como su posición en pantalla, tamaño en pixeles, modo de color, etc. Por otro lado, el segundo de ellos guarda información relativa a las transformaciones afines que se aplican a ese sprite en concreto, tal y como hacíamos con los fondos.

Ahora bien, durante la ejecución normal de un juego, lo típico es que tengamos varios sprites, y en un mismo ciclo cambiemos algo de casi todos ellos, por ejemplo si tenemos muchos enemigos a cada uno le tendremos que cambiar su posición o lo tendremos que animar. Es por ello que trabajar diréctamente sobre la OAM puede no ser la mejor idea, pues acceder a memoria supone un coste elevado. En su lugar, podemos crearnos una copia de la OAM que nosotros guardaremos en caché y, una vez por ciclo, volcar toda esa copia en la memoria real. Pero es muy importante que no pasemos por alto inicializar dicha copia.

Esto podemos llevarlo a cabo gracias a la estructura OAMTable de libnds, que contiene tanto los objetos de tipo SpriteEntry como SpriteRotation.

Código 5.10: Declaración de la copia de la OAM y

```
1 OAMTable* oam; //Nuestra copia de la OAM
2
3 [...]
4
5 //Inicializamos la copia de la OAM poniendo todos sus atributos a 0
6 void Graphics::initOAM(){
7
8     SpriteEntry* se;
9     SpriteRotation* sr;
10
11    for(int i = 0; i < SPRITE_COUNT; i++){
12        se = &oam->oamBuffer[i];
13        se->attribute[0] = ATTR0_DISABLED;
14        se->attribute[1] = 0;
15        se->attribute[2] = 0;
16    }
17
18    for(int j = 0; j < MATRIX_COUNT; j++){
19        sr = &oam->matrixBuffer[j];
20        sr->hdx = 1 << 8;
21        sr->hdy = 0;
22        sr->vdx = 0;
```

```

23     sr->vdy = 1 << 8;
24 }
25 }
26
27 [...]
28
29 //volcamos la copia de la OAM en la OAM real
30 void Graphics::updateOAM(){
31     memcpy(OAM,oam->oamBuffer,SPRITE_COUNT * sizeof (SpriteEntry));
32 }
33

```

Por último, solo quedaría crear un sprite y ver si se dibuja en pantalla. Esto lo haremos creando un objeto de tipo SpriteEntry, inicializar sus valores a los deseados y seguidamente actualizar la OAM.

Código 5.11: Declaración de la copia de la OAM y

```

1
2 int Graphics::createSprite(int x, int y){
3
4     //Creamos el objeto
5     SpriteEntry* sprite = &oam->oamBuffer[0];
6
7     //Inicializamos sus valores
8     sprite->y = y;
9     sprite->x = x;
10    sprite->gfxIndex = 0;
11    sprite->palette = 0;
12    sprite->size = OBJSIZE_64;
13    sprite->priority = OBJPRIORITY_0;
14    sprite->shape = OBJSHAPE_SQUARE;
15    sprite->isRotateScale = false;
16    sprite->isSizeDouble = false;
17    sprite->blendMode = OBJMODE_NORMAL;
18    sprite->isMosaic = false;
19    sprite->colorMode = OBJCOLOR_16;
20
21    //y actualizamos la OAM
22    updateOAM();
23
24 }

```

Como vemos, SpriteEntry tiene gran cantidad de variables y vamos a detenerlas a explicar las más importantes que afectan en que nuestro sprite se vea correctamente.

- **x, y:** Coordenadas en pantalla del sprite.
- **gfxIndex:** Coordenadas en pantalla del sprite.

- **palette:** Paleta que le corresponde (0-15).
- **size:** Tamaño del sprite. Los tamaños admitidos son 16x16, 32x32 y 64x64, aunque no tienen por qué ser cuadrados, pero la dimensión más grande debe ser de uno de estos valores.
- **priority:** Prioridad en pantalla (0 alta prioridad, se ve por encima-3 baja prioridad).
- **shape:** La forma del sprite, si es cuadrado, rectangular o de una forma más compleja.
- **colorMode:** Modo de color del sprite, si usa una paleta de 16 colores o 256.
- **isRotateScale:** Especifica si es un sprite al que se le pueden aplicar transformaciones afines. Es muy importante que tengamos en mente que aún si establecemos la transformación afín en el objeto SpriteRotation pero este valor está a false, no veremos el resultado de aplicar dicha transformación.

Por último, solo quedaría un aspecto relevante a la imagen de los sprites. Hemos especificado que el sprite ha de ser de una paleta de 16 colores, así que debemos generarla usando GIMP. Para ello, abrimos nuestro sprite en GIMP y seleccionamos Imagen > Modo > Indexado, y seleccionamos la opción Generar paleta óptima de 16 colores. Para establecer el color que será el transparente, grit escoje el primero de la paleta a no ser que le especifiquemos otro. Para el orden de los colores de la paleta, podemos ir a Colores > Mapa > Reordenar el mapa de colores, y en este caso especificar el color fucsia como transparente.

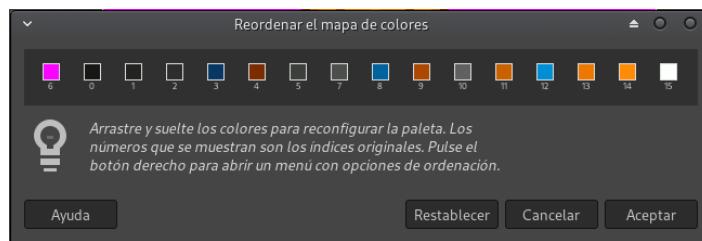


Figura 5.11: Reordenar paleta de colores desde GIMP.

Por último, si compilamos el juego y lo ejecutamos, deberíamos tener algo como lo siguiente:

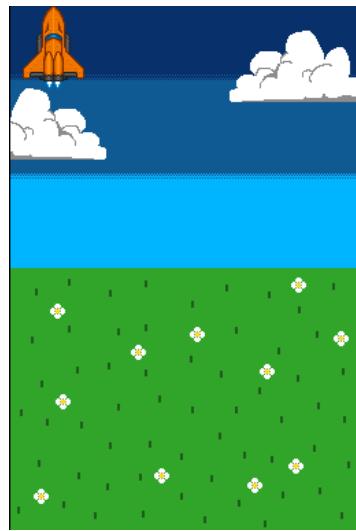


Figura 5.12: Prueba de dibujado de sprites.

5.2.1.3 Input

Vamos a ver ahora cómo conocer y **gestionar la entrada del usuario**. Esta tarea es bastante más sencilla que lo anterior que hemos estado viendo gracias a libnds, que nos facilita el trabajo. Para gestionar todo esto haremos una función **handleInput** que deberá ser llamada una vez en cada iteración del programa.

En esta función llamaremos a la función **scanKeys()** de libnds. Esto hará que se **actualice el estado de todos los botones de la consola**, pantalla táctil incluída. Después, podemos llamar a la función de evento que queramos de entre tres: **keysHeld**, si queremos comprobar las teclas que están siendo presionadas continuamente, **keysDown**, las que se acaban de pulsar, y **keysUp**, las que se acaban de soltar.

Lo único que nos faltaría es especificar qué tecla queremos comprobar exactamente para ese evento. Para ello usaremos las máscaras de libnds.

Código 5.12: Función para comprobar si el jugado mantiene pulsado el pad de direcciones hacia abajo

```

1 void handleInput(){
2     scanKeys();
3
4     if (keysHeld() & KEY_DOWN){
5         //ha pulsado la tecla
6     }

```

	Máscara de bit	Botón de la consola
KEY_A	1 <<0	A
KEY_B	1 <<1	B
KEY_SELECT	1 <<2	Select
KEY_START	1 <<3	Start
KEY_RIGHT	1 <<4	Derecha (Pad de direcciones)
KEY_LEFT	1 <<5	Izquierda (Pad de direcciones)
KEY_UP	1 <<6	Arriba (Pad de direcciones)
KEY_DOWN	1 <<7	Abajo (Pad de direcciones)
KEY_R	1 <<8	R
KEY_L	1 <<9	L
KEY_X	1 <<10	X
KEY_Y	1 <<11	Y
KEY_TOUCH	1 <<12	Pantalla táctil (sin coordenadas)
KEY_LID	1 <<13	Consola plegada

Tabla 5.1: Máscaras de teclas.

5.2.2 Iteración 1 - Pantallas del juego y lógica de los enemigos

Una vez ya adquiridos los conocimientos y la práctica necesarios para programar los gráficos en la NDS, comencé a desarrollar el producto.

Lo primero que quise implementar fueron las pantallas de juego, es decir, que de un menú principal pulsase una tecla y cargase un nivel. Para ello, primero hice un diagrama de clases que reflejan los distintos estados que tendría el juego.

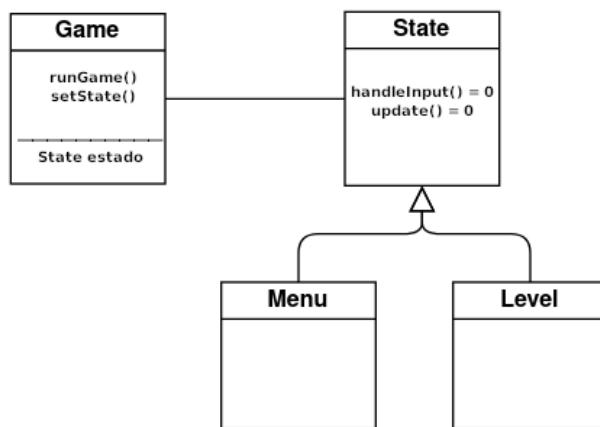


Figura 5.13: Diagrama de los estados del juego y sus clases.

Como vemos en la anterior imagen, nuestra clase principal sería Game, que tendría un objeto de tipo State. Este sería el estado actual de juego, el cual al comienzo se inicializaría con el estado Menu. Menu y Level son dos clases que heredan de State, y ésta es una interfaz cuyos métodos virtuales son los típicos de un estado: handleInput() y update(). Así, todos los estados tendrán esos métodos pero los implementarán de forma distinta, por ejemplo, el handleInput() de Menu simplemente debe comprobar que se pulse una única tecla para comenzar el juego, mientras que el Level requiere de otros inputs y es más complejo. Toda esta implementación de estados cumple el patrón state de programación orientada a objetos.

Por otro lado, para que los estados fuesen accesibles desde muchos contextos (por ejemplo, desde la futura clase player cuando muera querrá llamar a que se cambie al estado GameOver) estos debían ser únicos. Para ello, todos los estados tendrían un método Instance(), que devuelve un puntero al objeto de ese tipo de estado. Pero ¿cómo asegurarnos de que solo existe un único objeto de cada estado? Esto lo haremos mediante una variable estática.

Código 5.13: Implementación del patrón Singleton

```

1
2 Level* Level::Instance(){
3     static Level pinstance;
4     return &pinstance;
5 }
```

Como podemos ver en el anterior código, en cada método Instance() de cada estado habrá una variable estática del tipo del estado. Una variable estática se crea al principio del programa y su vida es toda la ejecución de éste hasta que acaba. Es decir, que la variable pinstance

se inicializará la primera vez que nosotros llamemos a ese método, creando así un objeto del tipo del estado, pero cuando volvamos a llamarla ésta ya estará creada así que simplemente nos la devuelve sin volver a crear una nueva. Esto en cuanto a diseño, es conocido como patrón Singleton.

También creé una clase Graphics, que se encargaría de realizar todas las tareas gráficas. Esto es para abstraer de la lógica del juego todas las funciones de libnds y hacer así un código más limpio. Esto en cuanto a diseño, es conocido como patrón Facade, pues la clase Graphics serviría a modo de fachada entre libnds y nuestro juego. Así, si en un futuro deseamos cambiar la librería es mucho más sencillo de hacer.

Por último, lo que hice durante esta iteración fue la gestión de un array de enemigos. Estos enemigos se crean al principio para evitar reservar memoria durante el juego, pero están inactivos. Para que fuesen saliendo uno a uno, creé un reloj que, al llegar a 0, recorre dicho array de enemigos buscando el siguiente inactivo para activarlo. Una vez están activos, estos enemigos se empiezan a mover hacia la derecha hasta el centro de pantalla.

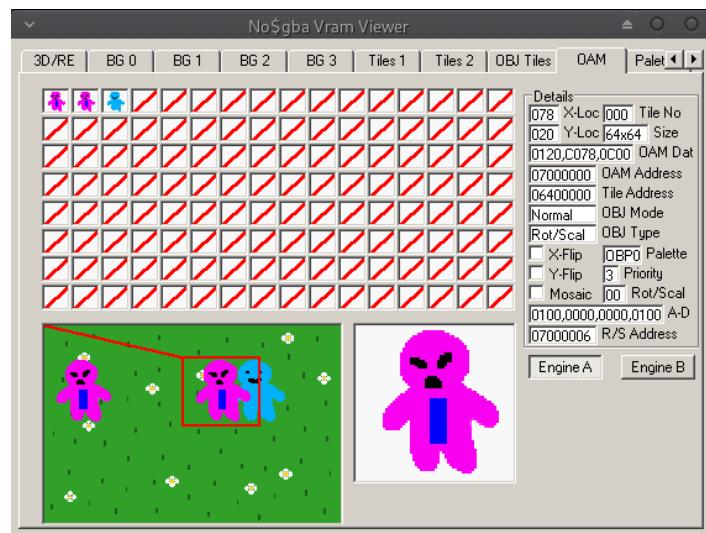


Figura 5.14: Resultado de la ejecución y elementos gráficos vistos desde el OAM Viewer de no\$gba

5.2.3 Iteración 2 - Mecánicas y primera versión del algoritmo de reconocimiento de gestos

En esta segunda iteración, me centré en el jugador y en el algoritmo de reconocimiento de gestos.

En cuanto al jugador, creé una clase Player que tiene variables como la vida, posición, etc. Para la vida, hice que los enemigos cada vez que colisionasen con el jugador redujesen la vida de este. También creé la vida del jugador, de la siguiente manera:

Código 5.14: Implementación del patrón Singleton

```

1 //create the life sprites
2 for(int i=0; i < MAX_LIFES; i++){
3     p->setHeartSpriteId(i,g->createLifeSprites(0+(i*16),0));
4 }
5 }
```

Como podemos ver, el jugador tiene un número máximo de vidas (6). Este trozo de código lo que hace es ir creando los sprites de corazones para que la vida sea visible, pero al tener distinta posición en x, cada vez que crea uno se va aumentando. Esto hará que se creen corazones pegados uno al lado del otro.

Otra cosa a la que le dediqué tiempo fue a una primera versión del algoritmo de reconocimiento de gestos. Al principio solo quería distinguir entre una línea horizontal y otra vertical, así que lo que se me ocurrió fue guardarme la primera vez que toca la pantalla y la última. Para comprobar si esta era una línea horizontal restaba sus valores en X y si estos eran mayores de un umbral significaba que era una línea horizontal, y lo mismo para la vertical. Sin embargo, esto no funcionaba del todo correcto, pues las líneas inclinadas cumplían ambas condiciones.

Dejé esto de lado de momento e hice que los enemigos tuvieran un patrón, así, al dibujar una línea se comprobaría si esa línea corresponde con el patrón del enemigo y, si es así, lo mataría.

Por último, lo que se realizó durante esta iteración fueron los sprites finales tanto de los enemigos como el jugador. Realicé el boceto en Procreate y el pixelart en GIMP.



Figura 5.15: Boceto y acabado del sprite del jugador



Figura 5.16: Boceto y acabado del sprite del enemigo

5.2.4 Iteración 3 - Algoritmo de reconocimiento de gestos

En esta iteración quería centrarme en el desarrollo de un algoritmo de reconocimiento de gestos, pues era algo vital para la jugabilidad del proyecto.

Así pues, lo primero que hice fue pensar en qué necesitaría para poder comparar dos trazos, uno ideal y otro que fuese el que el usuario ha dibujado. Basándome en el algoritmo Graffiti, decidí que lo que debía hacer era guardar cada par de coordenadas que el usuario pulsaba mientras dibujaba un trazo en una estructura del tipo array. Para ello, primero pensé en crear una estructura llamada Vector2D, que almacena en ella dos enteros, la coordenada x y la y. Después, decidí usar como array std::vector, por varias razones. La primera es que para poder guardar una colección de pares de números que el usuario va introduciendo en tiempo de ejecución necesitaremos una estructura dinámica, pues algunos usuarios dibujan el trazo más lento, lo que implica más muestras y viceversa. Por otro lado, esta herramienta nos ofrece datos y funciones mu cómodas para trabajar con ella (funciones para insertar números, acceso a su tamaño y capacidad actuales, redimensionado...).

Empecé entonces por desarrollar la entrada del usuario. Para ello, me creé un proyecto aparte con el fin de probar este algoritmo en una pantalla con salida de texto para depurar, a diferencia de nuestro proyecto que ya tenía las pantallas con sus fondos ya cargados y cambiarlo supondría mucho trabajo.

Código 5.15: Función consoleInit con los parámetros adecuados para crear un fondo que nos sirva para depurar

```

1
2     [...]
3
4     /*Parámetros:
5     0: Puntero a la terminal que usa
6     0: Capa de fondo que usa
7     BgType_Text4bpp: Tipo de fondo
8     BgSize_T_256x256: tamanyo del fondo

```

```

9   31: Comienzo en memoria
10  0:
11  false: Usa el motor de la pantalla inferior (si es verdadero usa la ↪
12    ↪ superior)
12  true: Carga la fuente por defecto para ser usada*/
13
14  consoleInit(0, 0,BgType_Text4bpp, BgSize_T_256x256, 31,0, false, true);
15
16  [...]
17
18  //17: Fila donde comenzar a escribir
19  //5H: Columna donde comenzar a escribir
20  printf("\x1b[17;5H Linea Vertical \n");
21
22  [...]

```

El código anterior muestra cómo preparé la consola para poder mostrar mensajes por pantalla que me sirviesen de sistema de depuración.

Para insertar los valores del usuario, como ya vimos anteriormente en el apartado de Input en nuestra función handleInput() llamaremos primero a scanKeys() y touchread() para que nos actualice el estado actual de todos los botones incluyendo la pantalla táctil. Después, si el usuario está pulsando la pantalla deberemos distinguir dos casos: si es la primera vez que pulsa o no. Para ello, comprobaremos el tamaño del vector. Si este es menor de 1 significa que es la primera vez que pulsa y si no, ya lleva manteniendo pulsado un rato. Esto lo hago para poder ahorrarme introducir un par de valores nuevos en el array si estos son iguales al anterior.

Código 5.16: Inserción de valores que conforman el trazo

```

1
2  [...]
3
4  if(keysHeld() & KEY_TOUCH){
5
6      //Guardamos el punto en el array
7      if(pattern.size()>1){ //si lleva un rato pulsando
8
9          auto* v = pattern[pattern.size()-1]; //nos guardamos el punto ↪
10            ↪ anterior (el ultimo)
11
11      if(v->getX() != stylus.px || v->getY() != stylus.py){ //y lo ↪
12        ↪ comprobamos para que no sea igual al que vamos a introducir
13          Vector2D* pos = new Vector2D(stylus.px, stylus.py);
14          pattern.push_back(pos);
14          pos = nullptr;
15      }
16

```

```

17         v = nullptr;
18
19     }else{ //si es la primera vez que pulsa
20         Vector2D* pos = new Vector2D(stylus.px, stylus.py);
21         pattern.push_back(pos);
22         pos = nullptr;
23     }
24
25 }
26
27 [...]

```

Una vez hecho esto, cuando detectemos que el jugador deja de pulsar con la función keyUp() y la macro KEY_TOUCH, tendríamos todas las coordenadas almacenadas y listas para trabajar con ellas.

Ahora bien, ¿cómo podemos comparar los dos trazos (ideal y del usuario) para saber si son parecidos? Matemáticamente podemos usar la desviación estándar, ya que tenemos un conjunto de valores y queremos ver su dispersión frente a los ideales. Esta desviación, a la que llamaremos σ , se calcula con la media cuadrática de los valores o RMS (del inglés root mean square), de la siguiente forma:

$$x_{RMS} = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + x_3^2 + \dots + x_N^2}{N}} \quad (5.1)$$

Así pues, cuanto menor sea σ , más se parecerán ambos trazos pues su desviación entre los puntos es menor.

Durante la ejecución del algoritmo lo que haremos será ir punto por punto calculando dicha desviación del punto ideal con el del usuario, acumulando todas las desviaciones para tener finalmente una desviación total que indicará cuánto se asemejan ambos patrones. Debemos tener en cuenta que estos cálculos losharemos independientemente por cada eje, así pues computaremos las coordenadas x por una parte, dandonos una desviación en dicho eje, y lo mismo con el eje y, sumando finalmente dichas desviaciones. Así pues, los cálculos queharemos por cada punto son los siguientes:

Para el eje X:

$$\sigma_{xn_{ideal} \rightarrow xn_{usuario}} = \sqrt{\frac{xn_{ideal}^2 + xn_{usuario}^2}{2}} \quad (5.2)$$

Para el eje Y:

$$\sigma_{xn_{ideal} \rightarrow xn_{usuario}} = \sqrt{\frac{xn_{ideal}^2 + xn_{usuario}^2}{2}} \quad (5.3)$$

Ahora bien, si bien calculando esto nos puede dar unos valores bastante fiables, hay un aspecto que tenemos que tener en cuenta primero.

El usuario, aunque por lo general dibujará el patrón lo más centrado que pueda, pero es posible que a veces al hacerlo rápido lo dibuje con tendencia a uno de los bordes de la pantalla. Por eso, si un patrón dibujado está muy desplazado con respecto al que debería ser al hacer el cálculo de la desviación esta puede aumentar, habiendo el riesgo de que entonces el algoritmo no te de el resultado correcto. Para solventar esto, he elegido como punto de referencia el inicio del trazo, tal y como hace Graffiti, y he añadido que por cada punto le reste la distancia tanto en x como en y para desplazar todo el trazo. Esto se muestra mejor en la siguiente figura:

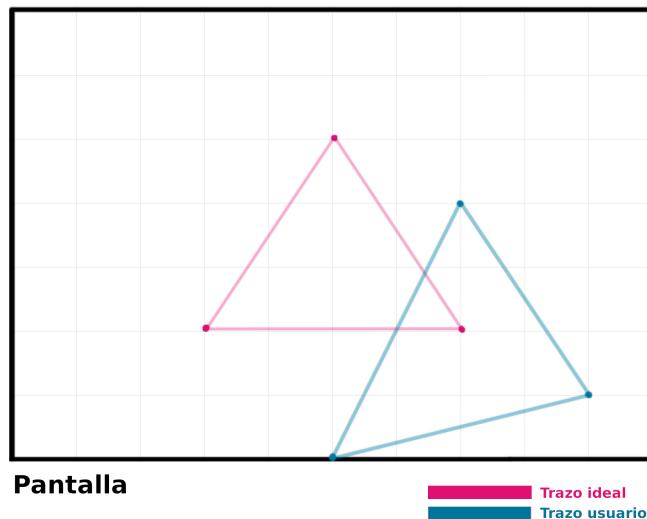


Figura 5.17: Visualización del trazo ideal y del usuario sin aplicar el desplazamiento

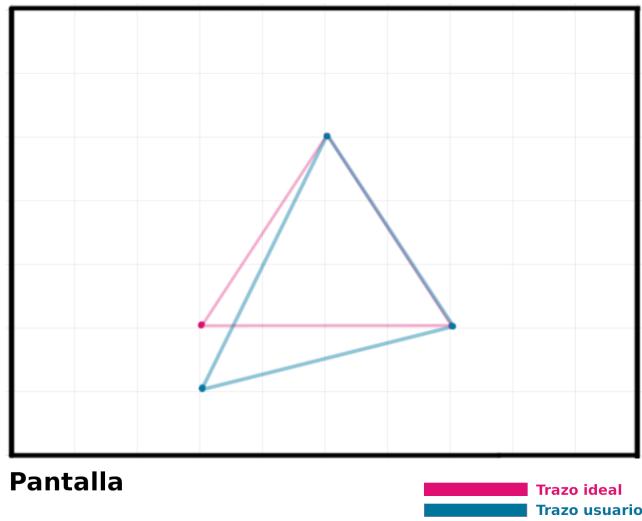


Figura 5.18: Visualización del trazo ideal y del usuario con el desplazamiento aplicado

Las fórmulas entonces de estos desplazamientos serían las siguientes:

$$dx = x_{ideal} - x_{usuario} \quad (5.4)$$

$$dy = y_{ideal} - y_{usuario} \quad (5.5)$$

Y esto es todo lo que deberíamos tener en cuenta, así que el algoritmo sería el siguiente:

Código 5.17: Algoritmo de reconocimiento de gestos

```

1 float checkPattern(const Vector2D shape[]){
2
3     //calcule displacement in both axis
4     //of the user pattern from the ideal pattern
5     int dx,dy;
6
7     Vector2D s = shape[0];
8     Vector2D* u = pattern[0];
9
10    dx = s.getX() - u->getX();
11    dy = s.getY() - u->getY();
12
13    //user pattern iterator

```

```

15     float inc = (pattern.size()-1)/(ANCHOR_POINTS-1);
16     float j = 0.0;
17
18
19     //total deviation
20     float o_tot = 0.0;
21
22     for(int i=0;i<ANCHOR_POINTS;i++){
23
24         u = pattern[j];
25         s = shape[i];
26
27         //displace point
28         int x_ = u->getX() + dx;
29         int y_ = u->getY() + dy;
30
31
32         //Calculate RMS and deviation for x and y
33         float rms_x, rms_y, aux;
34         aux = 0.0;
35
36         //x coordinate
37         aux = pow(s.getX(),2) + pow(x_,2); // x12 + x22
38         aux = aux/2.0;           // -----
39
40         rms_x = sqrt(aux);
41
42         o_tot = o_tot + abs(s.getX() - rms_x);
43
44
45         //y coordinate
46         aux = pow(s.getY(),2) + pow(y_,2); // y12 + y22
47         aux = aux/2.0;           // -----
48
49         rms_y = sqrt(aux);
50
51         o_tot = o_tot + abs(s.getY() - rms_y);
52
53
54         if(i == ANCHOR_POINTS-1){
55             j = pattern.size();
56         }else{
57             j = j + inc;
58         }
59
// ←
// → 2←
// →
// ←
// → ←
// → ←
// → 2←
// →

```

```
60     }
61
62     return o_tot;
63
64 }
```

La variable ANCHOR_POINTS se trata de una macro que establece cuántos puntos de muestra se usan para trabajar con el algoritmo, al igual que hace el algoritmo de PAlib. Esto hace que la velocidad del algoritmo sea mayor, aunque también aumenta el error.

Por último, una vez introducidos los datos de los patrones ideales, el algoritmo funcionaba correctamente como se ve en la siguiente imagen:

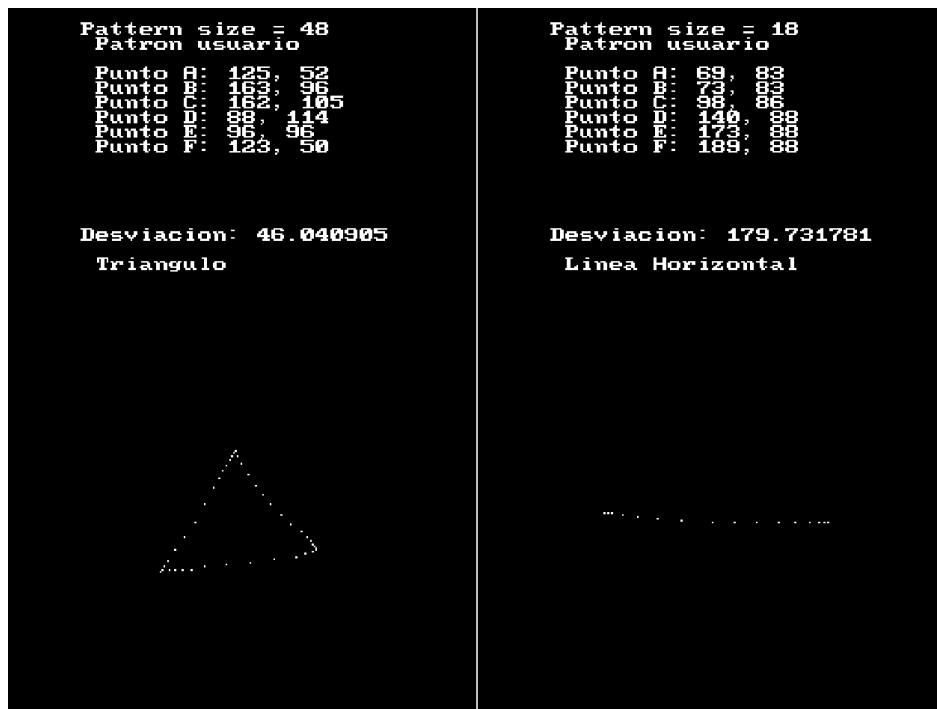


Figura 5.19: Resultado de la prueba del algoritmo desarrollado probando

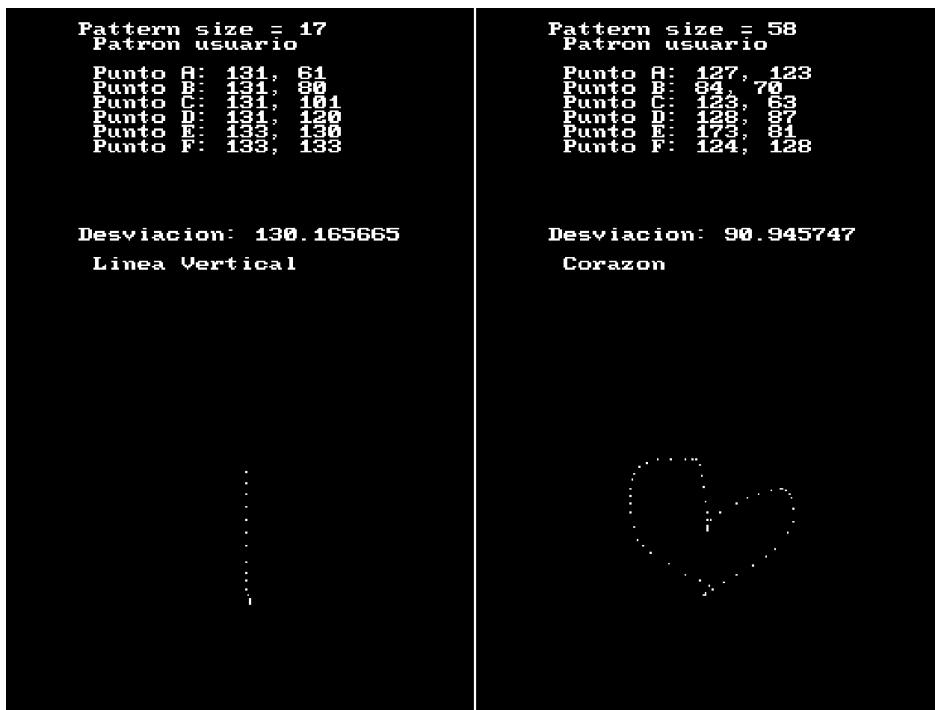


Figura 5.20: Resultado de la prueba del algoritmo desarrollado probando

5.2.5 Iteración 4 - Jugabilidad y ajustes de aparición de enemigos

Durante esta iteración me centré en avanzar en la parte jugable del juego, es decir en las mecánicas y niveles.

Lo primero en lo que me centré fue en el dibujado en la pantalla táctil para que el usuario pudiese ver de una manera agradable lo que estaba pintando. Así pues, quise conseguir un efecto similar al que tiene el juego Wario Ware: Touched! en su pantalla de inicio.

En esta pantalla, aparecen un montón de elementos casi aleatoriamente y, de vez en cuando, aparece una pelota que el usuario puede arrastrar y al moverse crea un rastro verde que desaparece con el tiempo.



Figura 5.21: Pantalla de inicio de Wario Ware: Touched!

Mi idea era parecida a esto, solo que quería que el rastro fuese permanente hasta que el usuario dibujase el patrón por completo y quería además que dicho rastro fuese con muchos colores, como si se tratase de un arcoíris para dar la sensación de que estuvieses dibujando con un pincel mágico.

Para implementar esto rápidamente se me ocurrió una idea, pues como ya he comentado, por cada pantalla somos capaces de dibujar hasta 4 fondos y estos pueden tener un color de transparencia. Así pues, mi idea era dibujar un fondo con degradado de arcoíris, encima de este otro simulando un lienzo y, por cada pixel que el usuario tocase en la pantalla táctil, pintar el color transparente en el fondo superior para que dejase ver el inferior.

Así pues, comencé por dibujar los dos fondos en la pantalla táctil, lo cual ya me causó problemas. Para mapear la memoria de vídeo como memoria de fondos, asigné el banco C usando la instrucción `vramSetBankC(VRAM_C_SUB_BG_0x06200000)`. De este modo tenemos 128 kilobytes disponibles para memoria de fondos, lo cual es suficiente para lo que necesitamos.

Para dibujar los fondos, como ya he comentado anteriormente primero inicialicé los registros controladores de los fondos.

Código 5.18: Configuración de los registros controladores de los fondos de la pantalla inferior

```

1 //set control registers for canvas background on the sub screen
2 // Size of the bg | Initial position in memory | priority (higher so ↪
3 // ↪ appears on top)
4 REG_BG3CNT_SUB = BG_BMP16_256x256 | BG_BMP_BASE(0) | BG_PRIORITY(2);
5
6 //affine matrix of the background
7 REG_BG3PA_SUB = 1 << 8; REG_BG3PC_SUB = 0;
8 REG_BG3PB_SUB = 0; REG_BG3PD_SUB = 1 << 8;
9
10 //position of the background
11 REG_BG3X_SUB = 0;
12 REG_BG3Y_SUB = 0;
13
14 //set control registers for the rainbow background on the sub screen
15 REG_BG2CNT_SUB = BG_BMP8_256x256 | BG_BMP_BASE(8) | BG_PRIORITY(3);
16
17 //affine matrix of the background
18 REG_BG2PA_SUB = 1 << 8; REG_BG2PC_SUB = 0;
19 REG_BG2PB_SUB = 0; REG_BG2PD_SUB = 1 << 8;
20
21 //position of the background
22 REG_BG2X_SUB = 0;
23 REG_BG2Y_SUB = 0;

```

Sin embargo, al ejecutar el juego no conseguía ver el fondo superior. Pasé un rato analizando qué estaba ocurriendo, pues a pesar de que estaba copiando donde esperaba el fondo, no se visualizaba ni en el juego, ni en las ventanas de depuración de no\$gba. Al final, opté por usar una herramienta que me ayudaría a verificar si lo que estaba intentando hacer tenía o no sentido.

Dicha herramienta es un visualizador de conflictos de asignación de VRAM para fondos en un navegador¹. En ella puedes introducir las configuraciones de tus fondos (tamaño, profundidad y posición inicial en memoria), la pantalla en la cual vas a dibujar los fondos y te proporciona si se van a producir conflictos e incluso qué bancos deberías usar. Pues bien, introducidos los datos de mi configuración dió el resultado de la figura 5.22.

Fué ahí entonces cuando entendí lo que estaba pasando. Con la configuración actual, un solo fondo ocupaba todo el espacio disponible (los 128KB) de memoria de vídeo, y el siguiente se quedaba fuera del banco, con lo cual no se iba a ver nunca. Para solucionar esto, cambié los fondos para que tuviesen una profundidad de 8 bits por pixel, en lugar de 16 y así ambos cabrían en el espacio de memoria sin problemas, como muestra la siguiente figura.

¹[#](https://mtheall.com/vram.html)

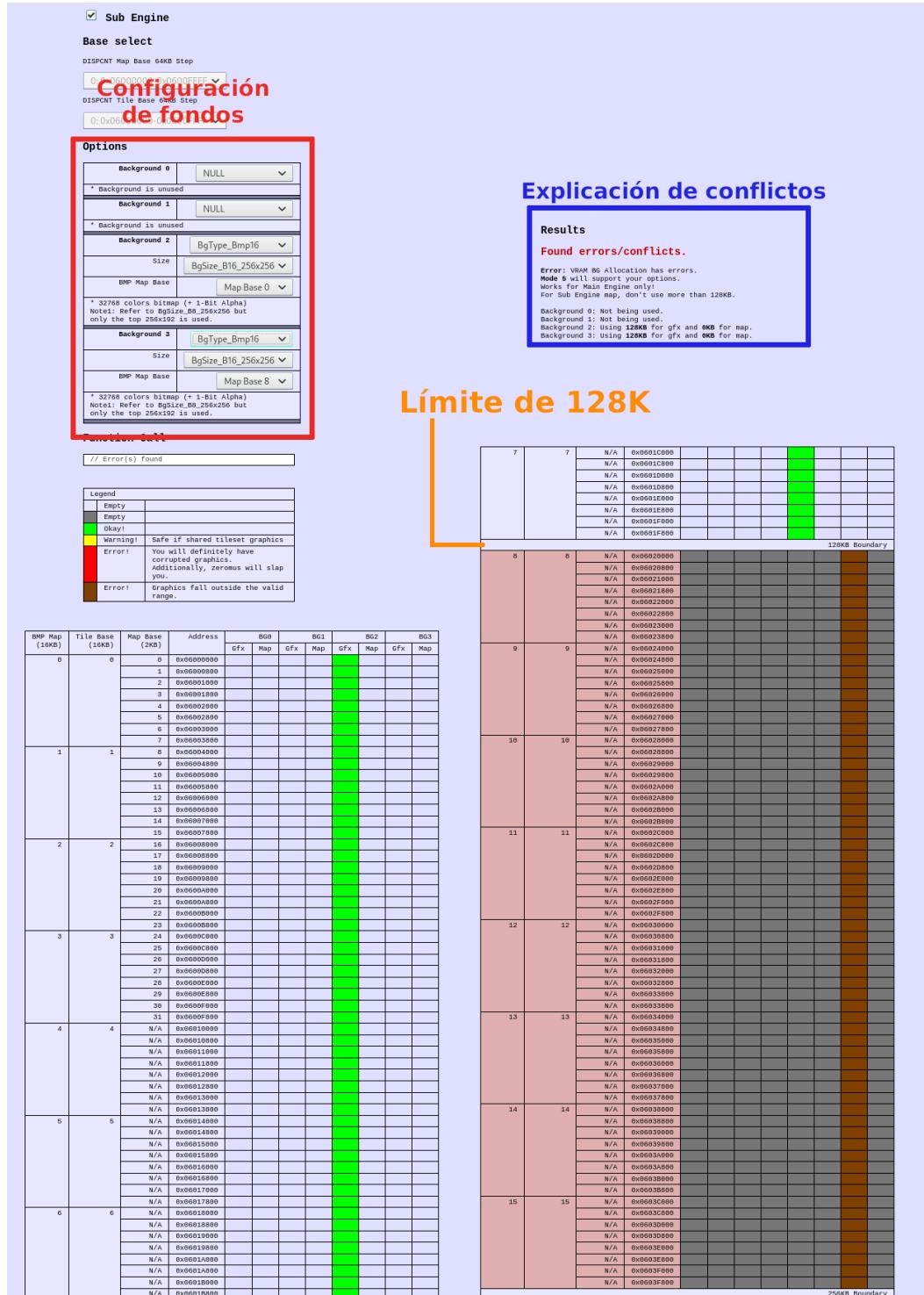


Figura 5.22: Resultado de la consulta con la mala configuración de fondos

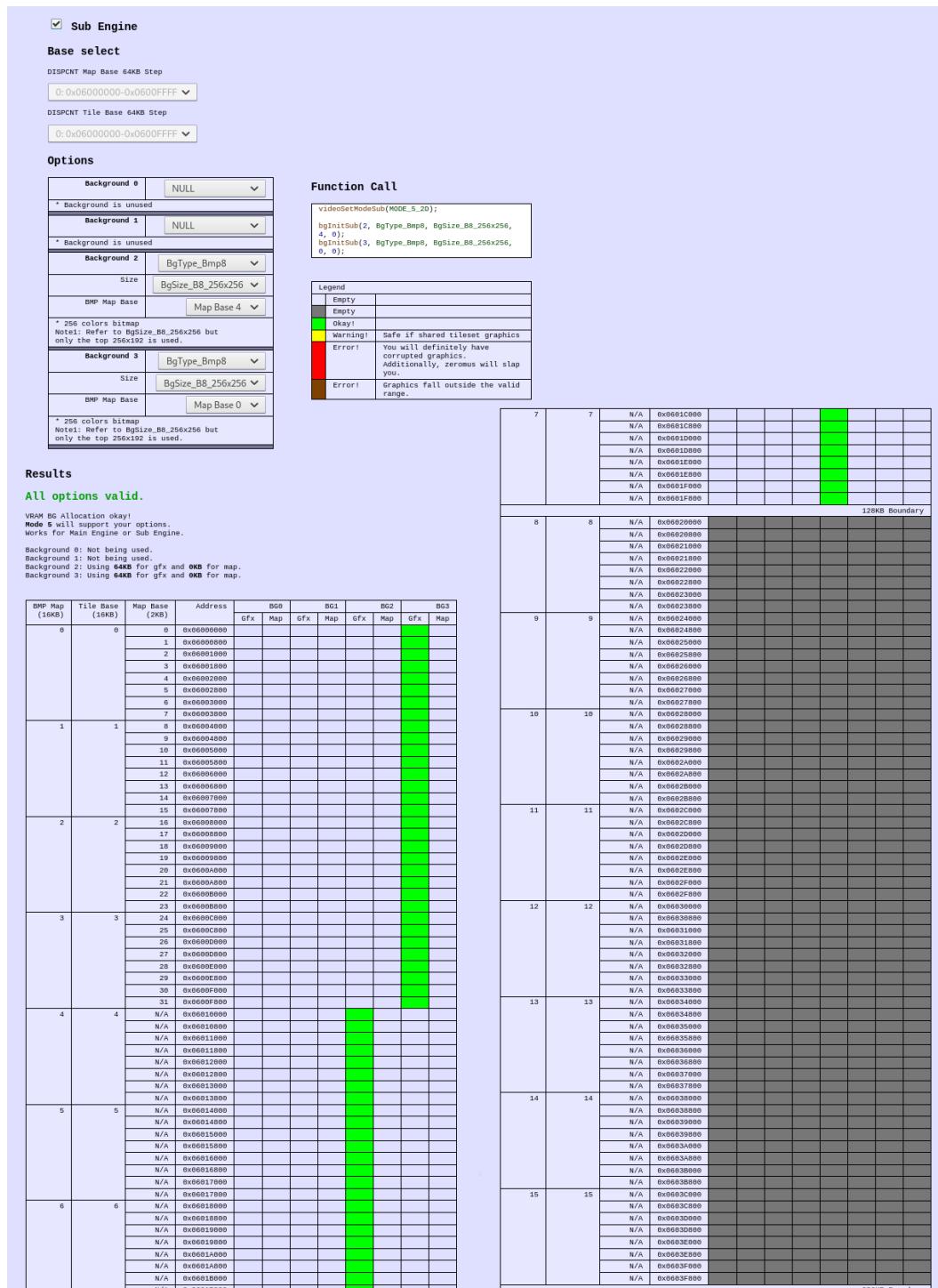


Figura 5.23: Resultado de la consulta con una buena configuración de fondos

Los cambios que tuve que realizar para solucionar esto no supusieron gran complicación, simplemente cambiando los registros controladores de fondos de 16 a 8 bpp usando la macro BG_BMP8_256x256, también en los archivos de configuración de grit de cada fondo, específicamente también la profundidad de pixel con el parámetro -gB8 y por último, como ambos fondos debían usar la misma paleta de X colores, abrí ambos fondos en GIMP y los exporté para que usaran la misma paleta de colores. Esto puede hacerse sencillamente abriendo ambos fondos en el mismo archivo de gimp, cambiando el modo de color a indexado y en el menú emergente seleccionar la opción de "Generar paleta óptima" con 16 colores, como ya hemos comentado en otras ocasiones.

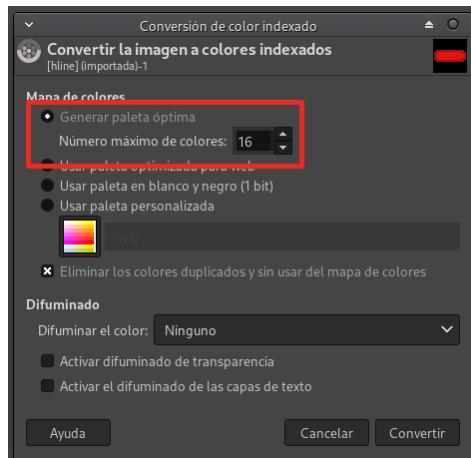


Figura 5.24: Cambio de imagen a modo indexado y selección de paleta

También, otra cosa que tuve que tener en cuenta fué especificar dónde comenzaba el segundo fondo en memoria. Antes, usaba la macro BG_BMP_BASE(8) para indicar que el segundo fondo comenzaba en la dirección 0x06020000, lo cual ya hemos visto que se nos quedaba fuera del rango disponible. Sin embargo, ahora el primer fondo ocupa la mitad de espacio, acabando en la posición 0x0600FFFF, con lo cual podía copiar el segundo a partir de la posición 0x06010000. Para representar esto con dicha macro basta con pasarle por parámetro el número 4, pues indica que, de los 8 bloques (numerados del 0 al 7) de 16KB cada uno en los que están divididos los 128KB de la memoria, el fondo comenzaría en el cuarto. Esto se puede ver gráficamente en la figura anterior, fijándonos en la columna BMP Map.

Una vez hecho esto, ya conseguí que se viesen ambos fondos, no obstante, aún había un pequeño problema. Como se ve en la siguiente figura, en la esquina superior izquierda del fondo se ven unos colores grises que no deberían aparecer, pues en su lugar era un tono de rojo. Esto me hizo pensar que quizás había algún problema con las paletas, así que usé las herramientas de depuración de no\$gba para ver qué ocurría.

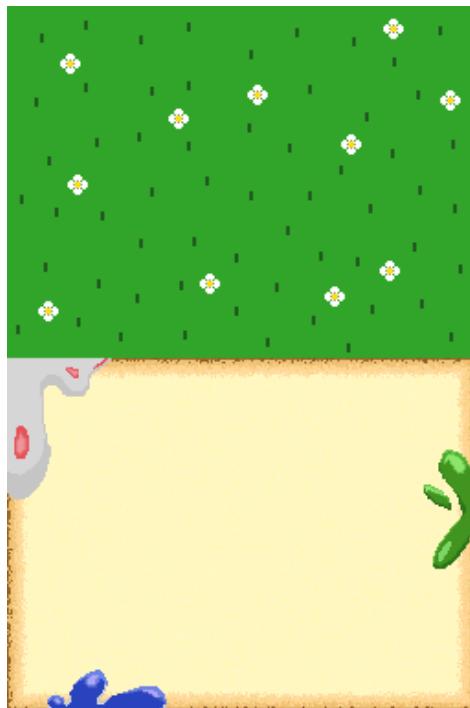


Figura 5.25: Fondo dibujado con un problema en la paleta

Así pues, como se ve en la figura 5.26 vi que habían unos 48 colores en escala de grises que no parecían pertenecer a la paleta de los fondos, era como si se hubiesen copiado por encima de ésta. Sospeché que el problema venía de la zona de memoria donde se copian las paletas de los sprites, pues la zona de memoria consecutiva a ésta es precisamente la dedicada a paletas de fondos de la pantalla inferior. Al comprobar las paletas de sprites, vi que también estaba presente ese degradado en tonos grises, lo cual me indicaba más aún de que alguna de las paletas de los sprites había sobreescrito las de los fondos al salirse de los límites. Comprobé entonces todos los sprites y efectivamente, el sprite del jugador tenía una paleta de 256 colores, seguramente por un despiste al exportar el archivo desde GIMP. Arreglado esto ya se veía el fondo perfectamente.

Una vez tenía ambos fondos dibujados, lo siguiente que quería hacer era conseguir que al pulsar en la pantalla táctil en una coordenada (x,y) , calcular dónde estaría ese pixel en memoria y cambiar ese pixel a negro ya que sería el color transparente, dejando así ver el fondo de arcoíris que está debajo de él.

Los píxeles en pantalla están representados por una coordenada (x,y) , donde x toma valores en un rango de $[0-254]$ e y en un rango $[0-191]$. Sin embargo, en memoria esto no es así, pues están representados en 1 dimensión, colocándose una pixel consecutivamente al que le precede.

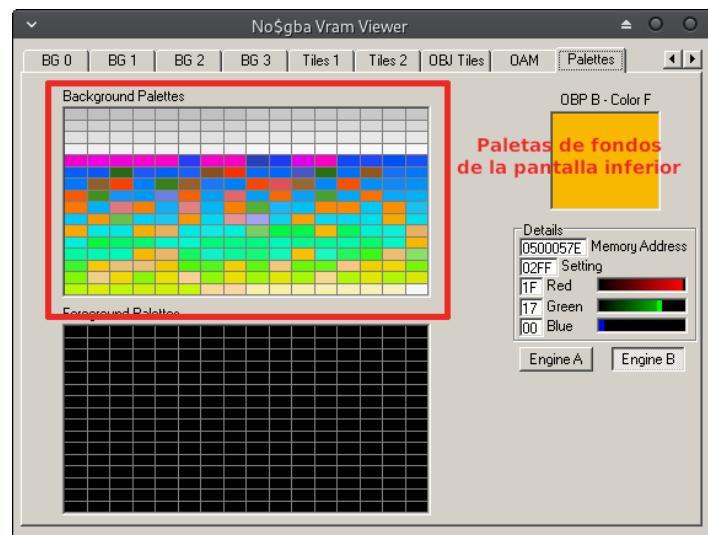


Figura 5.26: Superposición de tonos grises en la paleta de los fondos de la pantalla inferior

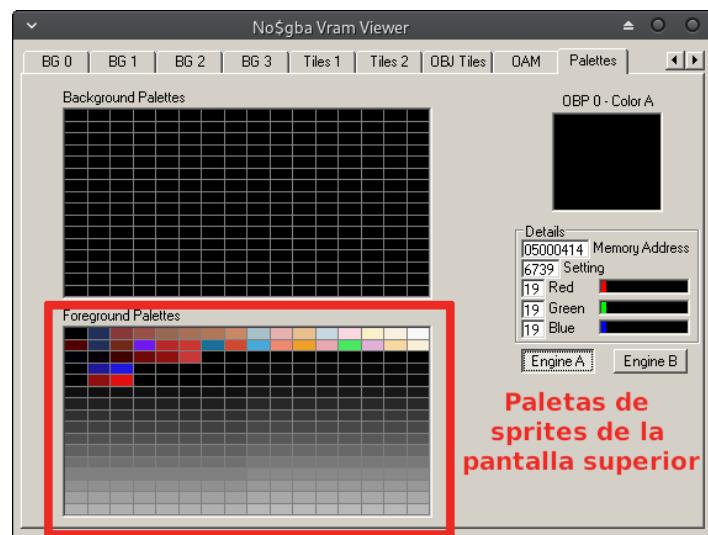


Figura 5.27: Paletas de los sprites que se salen de los límites

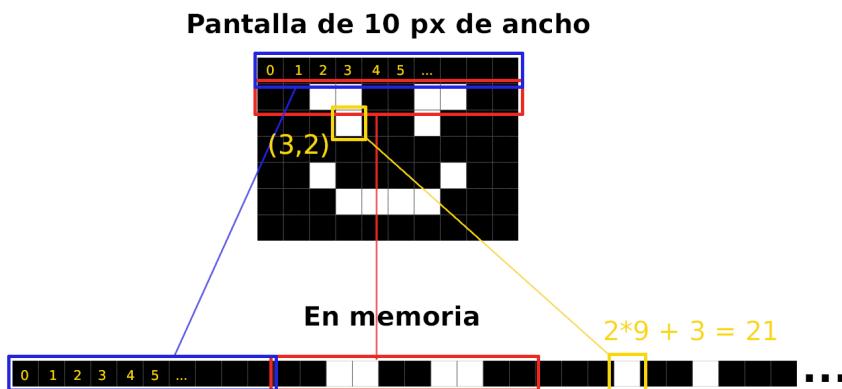


Figura 5.28: Comparativa de disposición de píxeles en pantalla y en memoria

Es por esto que si queremos saber qué posición ocuparía un pixel en memoria, bastaría con realizar el siguiente cálculo:

$$\text{indice} = x + \text{ANCHO}_{\text{pantalla}} * y \quad (5.6)$$

Por ejemplo, para el pixel en coordenadas de pantalla (0,1) siendo el ancho de pantalla 255, el pixel se encontraría en la posición número 255 ($0 + 255 * 1$), ya que por delante de él hay 255 píxeles que conforman la primera fila de pantalla.

Con esto ya sabríamos en qué posición encontraríamos nuestro pixel, pero esto no es todo, pues nuestro fondo no está en la posición 0x00000000, sino que está donde hemos asignado el banco de memoria de video (0x06200000). Por eso, debemos incrementarle a nuestro resultado dicha posición inicial. Además, también tenemos que tener en cuenta que debemos dividir el resultado entre 2, debido a que los fondos pasan a ser de 8bpp.

El siguiente código muestra una implementación de una función llamada `drawPointSub` que hace lo explicado anteriormente tomando como parámetros las coordenadas (x,y) de pantalla donde el usuario pulsa. Además de pintar dicho pixel, también pinta los adyacentes formando un círculo como se puede ver en la siguiente figura.

Código 5.19: Función de dibujado de píxeles dada una entrada de usuario

```

1
2 #define SCREEN_START ((u16*)0x06200000) //start of the background video memory
3 const unsigned int color = 0; //pixel color
4
5 [...]
6
7 void Graphics::drawPointSub(int x, int y){

```

```

8
9    //calculate the memory address
10   int add = x + SCREEN_WIDTH * y;
11   add = add/2;
12
13   //draw the pixel
14   dmaCopy(&color, SCREEN_START + add , 2);
15
16   [...]
17 }
```

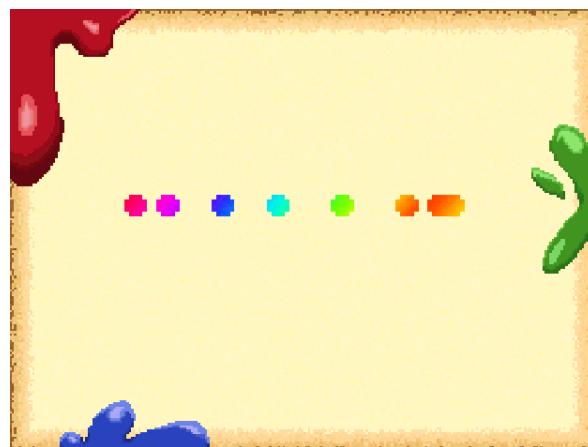


Figura 5.29: Resultado de usar la función drawPointSub para dibujar una línea

Esta función era llamada cada vez que el usuario comenzaba a pulsar o mantenía pulsada la pantalla táctil y debería dejar el rastro de éste, sin embargo como vemos en la figura anterior parece que no realiza todo el recorrido, pues dibujé una línea recta. Esto era debido al framerate de la consola, el cual está ajustado a 60 debido a que en cada iteración siempre espera a que se de la interrupción VBLANK (es decir, cada vez que se termina de dibujar un frame).

Para solucionar esto lo que hice fue, entre frame y frame, en lugar de dibujar un punto concreto dibujar una línea que unía el punto que el usuario estaba pulsando actualmente y el que había pulsado en la anterior iteración. Esto lo solventé implementando el algoritmo de Bresenham para dibujado de líneas.

5.2.5.1 Algoritmo de Bresenham

El algoritmo de Bresenham sirve para dibujar en medios discretos gráficos tales como líneas, círcunferencias y otras curvas. A toda la familia de algoritmos se les conoce bajo el mismo

nombre ya que parten de la misma idea y pero tienen ligeras modificaciones.

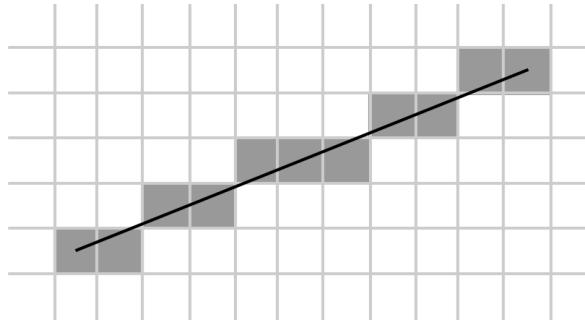


Figura 5.30: Aplicacion del algoritmo de Bresenham para dibujar líneas en un medio discreto

Fuente: ProjectPrinter2D

El algoritmo de Bresenham para dibujado de lineas se basa en lo siguiente:

- Dados un punto inicial y final de la recta, calculamos las distancias entre los puntos tanto en el eje horizontal como vertical.
- Comprobamos qué distancia es mayor, pues la mayor hará que en cada iteración avance siempre en una unidad. Por ejemplo, si una linea es casi o plenamente vertical esta claro que siempre avanzará en una unidad en el eje Y.
- En cuanto a la distancia que sea menor, esta a veces avanzará o no avanzará en su eje. Para saber cuándo hacerlo o no, se vale de un valor que llamaremos 'e', que se calcula como la relación entre las distancias por ejes.

$$e = dx/dy \quad (5.7)$$

- Cuando este valor es inferior a la unidad, no se avanza y cuando la supera (en cada iteración se acumula), se le resta una unidad y se avanza en ese eje. Esto se puede ver mejor en la siguiente figura.

Una vez entendido esto, procedí a implementar el algoritmo. Para ello, dentro de mi clase que maneja todos los gráficos creé una función a la que le pasaría las coordenadas del punto inicial y final y se encarga de llamar a la función que he mencionado antes (`drawPointSub`) para pintar un punto en la pantalla.

Al principio podemos distinguir dos bloques condicionales, que se encargan de calcular los incrementos tanto en el eje X, como en el Y, en función de la orientación de la línea. Esta puede saberse coprobaro si las distancias son negativas. Después, un tercer bloque comprueba la inclinación de esta ($si dx \geq dy$), para ajustar cuál será el incremento que

siempre aumentará una unidad en cada iteración. Por último, se hacen los cálculos de los errores para la posterior decisión de incrementar o no el eje de menor distancia.

Una vez calculados estos valores, se inicializan a nuestro punto inicial de la recta unos puntos que llamaremos los 'actuales'. Estos nos servirán de iteradores y se irán actualizando según el algoritmo. Así pues, comenzamos a iterar, siempre dibujando el punto al principio y después realizando los cálculos de los errores para ver si incrementamos una unidad en el eje de menor distancia (cuando av sea mayor o igual a 0) y por último actualizamos los puntos actuales. Repetiremos esto hasta que el punto actual coincida con el punto final.

Código 5.20: Implementación del algoritmo de Bresenham

```

1
2 void Graphics::drawLineSub(int x_i, int y_i, int x_f, int y_f){
3
4     int dx = x_f - x_i;
5     int dy = y_f - y_i;
6
7     int inc_x_i, inc_x_r, inc_y_i, inc_y_r = 0;
8
9     //Calculamos los desplazamientos en el eje X
10    if(dx >= 0){
11        inc_x_i = 1;
12    }else{
13        dx = -dx;
14        inc_x_i = -1;
15    }
16
17    //Calculamos los desplazamientos en el eje Y
18    if(dy >= 0){
19        inc_y_i = 1;
20    }else{
21        dy = -dy;
22        inc_y_i = -1;
23    }
24
25    //Decidimos cual es el eje de incremento siempre en base a la inclinacion ↪
26    ↪ de la recta
27    if(dx >= dy){
28
29        inc_x_r = inc_x_i;
30        inc_y_r = 0;
31    }else{
32
33        inc_x_r = 0;
34        inc_y_r = inc_y_i;
35
36        int aux = dx;
37        dx = dy;

```

```
38     dy = aux;
39 }
40
41 //Calculamos los errores
42 int av_r = 2 * dy;
43 int av = av_r - dx;
44 int av_i = av - dx;
45
46 //Punto iterador
47 int x_act = x_i;
48 int y_act = y_i;
49
50 //Bucle
51 do{
52
53     //Función de dibujar un punto
54     drawPointSub(x_act, y_act);
55
56     if(av >= 0){
57
58         //Nos movemos en dos ejes
59
60         //Actualizar el punto
61         x_act = x_act + inc_x_i;
62         y_act = y_act + inc_y_i;
63         //Actualizar el error
64         av = av + av_i;
65
66     }else{
67
68         //Nos movemos solo en un eje
69
70         //Actualizar el punto
71         x_act = x_act + inc_x_r;
72         y_act = y_act + inc_y_r;
73         //Actualizar el error
74         av = av + av_r;
75
76     }
77
78     //Repetir hasta llegar al punto final
79 }while(x_act != x_f || y_act != y_f);
80 }
```

Una vez hecho esto, solo falta hacer una pequeña distinción. Llamaremos a esta función siempre que el usuario mantenga pulsado, así pues deberemos tener unas variables que nos permitan saber qué coordenadas se pulsaron en la anterior iteración e ir actualizándolas. Esto nos lleva a un caso, ¿qué pasa si es la primera vez que el usuario pulsa la pantalla? Pues bien, para eso, incializamos estas variables a un valor negativo y comprobaremos ese

valor a la hora de dibujar. Si lo son, dibujaremos un único punto, inicializaremos los puntos y seguiremos y, si no, se aplicará Bresenham. Esto se puede ver en el siguiente código:

Código 5.21: Llamada a Bresenham

```
1
2 [...]
3
4 //Si es la primera vez que pulsa la pantalla táctil
5 if(x_in == -1 && y_in == -1){
6
7     //Dibujamos un punto
8     g->drawCircleSub(stylus.px, stylus.py);
9
10 //Si no
11 }else{
12
13     //Nos ahorraremos el cálculo si el punto que vamos a dibujar es el mismo que ↵
14     ↵ la iteración anterior
15     if(x_in != stylus.px || y_in != stylus.py){
16
17         //Calcular Bresenham
18         g->drawLineSub(x_in, y_in, stylus.px, stylus.py);
19     }
20
21 //Actualizamos el punto
22 x_in = stylus.px;
23 y_in = stylus.py;
24
25 [...]
```

Finalmente, este es el resultado del dibujado de una varias líneas usando dicho algoritmo en la pantalla táctil:



Figura 5.31: Dibujado en la pantalla táctil usando el algoritmo de Bresenham

Una vez conseguido el dibujado, lo siguiente que hice fue implementar en nuestro proyecto el algoritmo de reconocimiento de gestos desarrollado en la iteración anterior, ya que si bien aún necesitaba algunos retoques, funcionaba bastante mejor que se desarrolló en las primeras iteraciones y ya permitía jugar al usuario.

Para hacer esto, volví a recurrir al diseño de clases del proyecto, pensando cómo podría incluir el uso de este algoritmo de una forma abstraída y limpia. Entonces mi solución fué crear una clase llamada `PatternManager`, que fuese accesible desde `Level`. Esta clase contendría tanto los patrones ideales como el patrón usuario, es decir, unas estructuras de datos (arrays y vectores) y unos métodos para trabajar con ellos, como són los siguientes:

- `checkPattern()`: Toma como parámetro el patrón ideal con el que quiere comparar el del usuario y realiza el algoritmo descrito en la anterior iteración, devolviendo un número decimal que representa el error total entre ambos patrones.
- `insertPoint()`: Pasandole como parámetro la coordenada x e y del punto de pantalla que el usuario ha pulsado, inserta un nuevo elemento en el array de puntos que conforman el trazo del usuario.
- `clearPattern()`: Borra todos los elementos del array de puntos que conforman el trazo del usuario.

A continuación se muestra un esquema de diagrama de clases que muestra la relación de las clases `PatternManager` y `Level`:

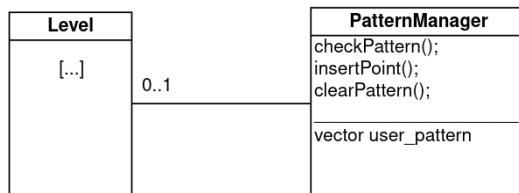


Figura 5.32: Diagrama de la relación entre las clases `Level` y `PatternManager`

Por último, lo que implementé durante esta iteración fue una mejora en los enemigos. Para que fuesen similares a nuestro referente, Magic Cat Academy, jugué varias veces a éste y me percaté de que más o menos, los enemigos siempre salían por las mismas zonas, las cuales muestro en la siguiente figura:

Así pues, lo que hice principalmente fue definir 6 posiciones fijas y al crear un enemigo, se le asigna una de estas posiciones. No obstante, como ahora el update de cada enemigo sería

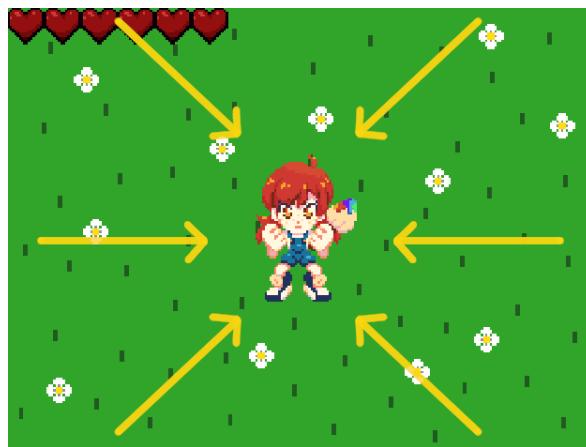


Figura 5.33: Definición visual de las posiciones iniciales y caminos de los enemigos

distinto (unos se mueven recto, otros inclinados, etc), tuve que definir dos nuevas variables que se iniciarían justo al definir esta posición del enemigo. Estas variables son inc_x e inc_y, las cuales definen el camino que seguirá el enemigo.

Ahora bien, lo último que quedaba era asegurarse de que todos los enemigos que saliesen de la parte izquierda de la pantalla, tuviesen el sprite invertido horizontalmente, para que mirasen hacia la dirección en la que avanzan. Esto se puede hacer fácilmente gracias a libnds y la estructura SpriteEntry, de la cual ya hemos hablado, tan solo debemos asignar la propiedad hFlip a verdadero, sin embargo, nos esto nos pone un impedimento, y es que libnds nos dice que para poder invertir un sprite es necesario que una de sus otras propiedades, isRotateScale sea falso. Esto quiere decir que no podemos aplicar sobre el sprite transformaciones afines de escalado y rotado, con lo cual debemos tenerlo en cuenta.

Lo último que hice fue asignar bien las prioridades de los sprites para que algunos se viesen por encima de otros, de la siguiente manera:

- Prioridad 0: Los sprites de la vida del jugador, siempre por encima de todo.
- Prioridad 1: Los enemigos que vienen desde la línea inferior
- Prioridad 2: El jugador.
- Prioridad 3: Los enemigos que vienen desde la línea central o la superior.

Y este fue el resultado de los enemigos con el nuevo cambio en funcionamiento:



Figura 5.34: Resultado de esta implementación

5.2.6 Iteración 5 - Retoque algoritmo de reconocimiento de gestos y animaciones

Durante esta quinta iteración planifiqué centrarme en varios aspectos relativos al algoritmo de reconocimiento de gestos para así poder dejarlo completamente terminado.

Si bien el algoritmo ya estaba implementado en el juego y sus resultados eran buenos, aún quedaban algunos aspectos por revisar y perfeccionar, pues probándolo me di cuenta de dos problemas que tenía. El primero de ellos es, que según la forma de realizar el cálculo de la desviación entre los puntos de los trazos, el trazo dibujado por el usuario debía tener el mismo sentido que el que nosotros tomábamos por ideal. Tanto el trazo del usuario como el ideal están almacenados en estructuras de tipo array, donde los puntos son accesibles gracias a un índice que va de 0 hasta el tamaño total de la estructura. Pues bien, el algoritmo al comenzar a iterar los puntos del trazo del usuario lo hace siguiendo ese orden, desde 0 hasta el tamaño máximo, calculando las desviaciones entre pares de puntos que comparten el mismo índice, es decir, el punto 0 con el 0, el 1 con el 1, etc.

Esto podemos visualizarlo mejor gracias a la siguiente figura. Supongamos un trazo ideal que sea una línea horizontal de 3 puntos: $p1(0,0)$, $p2(1,0)$ y $p3(2,0)$, por simplificación. En nuestro código, hemos decidido que se almacenarán en ese mismo orden, correspondiendo el índice 0 a $p1$, el 1 a $p2$ y el 2 a $p3$. Ahora el usuario, al ver que para poder jugar debe dibujar una línea horizontal la dibuja siguiendo el orden inverso por preferencia suya, es decir, que su línea almacenada en nuestro programa sería $p1(2,0)$ en el índice 0, $p2(1,0)$ en el 1 y $p3(0,0)$ en el 2. Si realizamos el cálculo en el orden de los índices la desviación total que nos da es aproximadamente 2,82, lo cual es erróneo pues ambas líneas son exactamente iguales y la desviación debería ser nula. En la imagen está representado mediante flechas rojas los cálculos que actualmente realiza el algoritmo y que son erróneos y, en verde los que debería realizar por lógica.

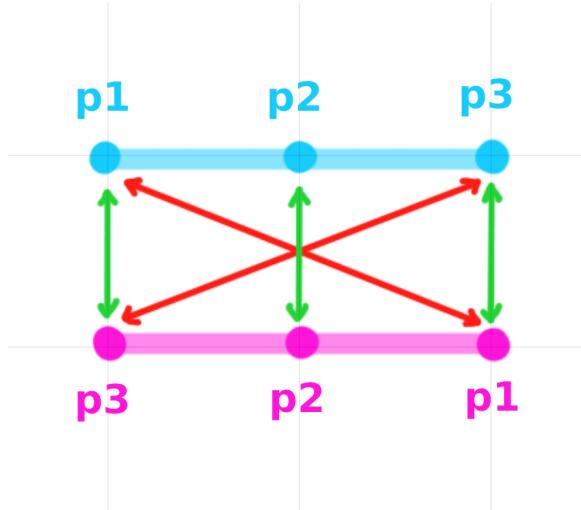


Figura 5.35: Ejemplo visual del cálculo de la desviación entre los puntos de dos trazos

Si nos fijamos bien en los algoritmos que hemos estudiado previamente como PAlib y Graffiti, vemos que estos sí tienen en cuenta el sentido de dibujado del trazo y diferencian, por ejemplo, en PAlib las líneas que han sido dibujadas de izquierda a derecha y viceversa representan distintos caracteres dado que sus ángulos son distintos. Además, por lo general la gran mayoría de trazos que representan letras se dibujan de izquierda a derecha. No obstante, nuestro referente en cuanto a mecánicas y jugabilidad, Magic Cat Academy, ignora el sentido del trazo y se centra en que la forma en general sea la correcta. Esto me parece más intuitivo, pues al tratarse de un juego al que pueden jugar tanto personas zurdas, diestras, o incluso alguna persona con discapacidad motora, obligarle a realizar el trazo en un sentido concreto puede afectar negativamente a su experiencia de usuario. Así pues, decidí tratar este problema y resolverlo haciendo que el cálculo de la desviación se realizase entre dos puntos cuya distancia sea mínima.

Para hacer el cálculo entre los puntos de menor distancia hice una función llamada `findNearest` que toma como argumentos unas coordenadas x e y, y el trazo ideal del cual se quiere buscar el más cercano. Así pues, cada vez que desde nuestro algoritmo principal estemos procesando un punto del trazo del usuario se llamará a esta función para que nos devuelva el punto del trazo ideal de menor distancia a ese.

Código 5.22: Búsqueda del punto más cercano con la función `findNearest`

```

1 int PatternManager::findNearest(int x, int y, const Vector2D shape[]){
2
3     float dist = FLT_MAX;
4     int nearest = -1;
5
6     for(int i = 0; i<ANCHOR_POINTS; i++){
7
8         Vector2D pos = shape[i];

```

```

9
10    //calculamos la distancia
11    float dist_act = sqrt(pow((pos.getX()-x),2)+pow(pos.getY()-y,2));
12
13    if(dist_act < dist){ // actualizamos el punto de menor distancia
14        nearest = i;
15        dist = dist_act;
16    }
17}
18
19 return nearest;
20
21}

```

El código anterior muestra el funcionamiento de la función `findNearest`. Lo que hace básicamente es comenzar a recorrer el array del trazo ideal calculando la distancia entre el punto sobre el que está iterando y el punto pasado por parámetro. Si esta distancia es menor, se actualizan tanto una variable que guarda dicho valor de la distancia para ser comparado con los demás y el índice de ese punto, que será lo que la función da como resultado.

El cálculo de la distancia puede resultar algo engorroso de ver tal y como está escrito en el código, por ello se muestra en la siguiente fórmula:

$$d(AB) = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2} \quad (5.8)$$

Las funciones `pow()` y `sqrt()` de la librería de sistema `math.h` nos proporcionan el cálculo de la potencia y raíz cuadrada respectivamente.

Con esto y tendríamos solucionado este problema, no obstante al haber hecho una modificación tan importante como añadir un bucle más al algoritmo hemos de tener en cuenta que ahora su tiempo de ejecución ha aumentado, pues ha pasado de ser un simple bucle (complejidad lineal) a ser un bucle anidado (complejidad cuadrática). Esto puede ser un problema si decidimos aumentar el número de puntos de muestra de cada patrón o incluso si añadimos muchos más patrones distintos. Recordemos que este cálculo es algo que se debe realizar con la mayor rapidez posible, pues de lo contrario podría provocar tirones en el juego y repercutiría negativamente en la experiencia del usuario. No obstante, como de momento tenemos pocos puntos de muestra y pocos patrones vamos a centrarnos primero en resolver el siguiente problema.

En cuanto al segundo problema, este se debe a la forma en la que desplazamos el trazo del usuario hacia el trazo ideal para realizar posteriormente el cálculo de la desviación. Como ya hemos comentado, elegimos el punto inicial de ambos trazos, calculábamos el desplazamiento y después cuando iterabamos sobre los puntos del trazo del usuario los ibamos desplazando. El hecho de usar el punto inicial tenía sentido si, nuevamente, todos los trazos tuvieran que

ser hechos con el mismo sentido que los ideales. Sin embargo, al no tener en cuenta esto y dejando el algoritmo tal y como estaba, el trazo del usuario se desplazaria de tal forma que el cálculo de la desviación podría dar unos valores incorrectos. Esto se puede ver mejor en la siguiente figura.

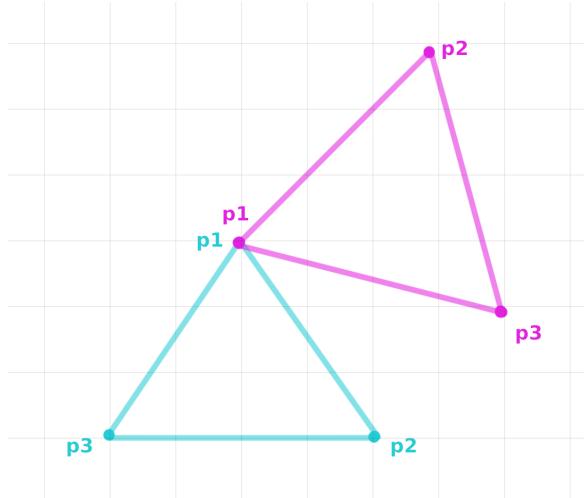


Figura 5.36: Desplazamiento erróneo del trazo del usuario (rosa) con respecto al trazo ideal (azul)

Como solución a este problema, debía desplazar todo el trazo a un punto que representase el centro del mismo y que pueda ser calculado para cualquier nube de puntos, este es el centro de masas.

La fórmula del cálculo del centro de masas es la siguiente, donde r representa el vector de posición (x, y) del punto, m su masa y M el valor total de la masa del sistema.

$$r_{cm} = \frac{\sum_i m_i r_i}{\sum_i m_i} = \frac{1}{M} \sum_i m_i r_i \quad (5.9)$$

No obstante, si bien el centro de masas es un concepto con el que podemos trabajar en ámbitos donde los puntos u objetos de un sistema posean un peso, en nuestro caso podemos asumir que todos nuestros puntos tienen el mismo peso o importancia, en este caso la unidad. Así pues, la fórmula quedaría de la siguiente manera:

$$r_{cm} = \frac{1}{N} \sum_{i=0}^N r_i = \frac{r_0 + r_1 + r_2 + \dots + r_n}{N} \quad (5.10)$$

Una vez entendido esto, para implementarlo en el código creé una función llamada calculateCenterOfMass() que recorre el vector que conforma el trazo del usuario, selecciona los puntos de muestra y los acumula para calcular así el centro de masas y almacenar sus coordenadas luego en dos variables, x_cm e y_cm.

Código 5.23: Cálculo del centro de masas con la función calculateCenterOfMass()

```

1 void PatternManager::calculateCenterOfMass(){
2
3     int sum_x = 0;
4     int sum_y = 0;
5
6     x_cm = 0;
7     y_cm = 0;
8
9     //iterador sobre los puntos de muestra
10    float inc = (user_pattern.size()-1)/(ANCHOR_POINTS-1);
11    float j = 0.0;
12
13    for(int i=0;i<ANCHOR_POINTS;i++){
14        if(user_pattern[i]){
15
16            Vector2D* pos = user_pattern[j];
17
18            sum_x = pos->getX() + sum_x; //acumulamos la coordenada X
19            sum_y = pos->getY() + sum_y; //acumulamos la coordenada Y
20
21        }
22
23        if(i == ANCHOR_POINTS-1){
24            j = user_pattern.size()-1;
25        }else{
26            j = j + inc;
27        }
28    }
29
30    //dividimos entre el total de puntos
31    x_cm = sum_x/ANCHOR_POINTS;
32    y_cm = sum_y/ANCHOR_POINTS;
33
34}
```

A pesar de haber añadido nuevamente una tarea más al algoritmo esto no supone un gran problema como puede ser el anterior a nivel de coste, pues este cálculo se va a realizar una única vez cada vez que el usuario dibuje un patrón ya que el centro de masas es el mismo para realizar los cálculos con todos los patrones ideales que tengamos.

Además, los centros de masas de los patrones ideales los almacené en un array y así ir usando los que me convenga. Por ejemplo, el usuario dibuja un trazo, seguidamente calculo el centro de masas de ese trazo, paso a realizar el cálculo de desviaciones entre ambos y en

ese proceso uso ese array para obtener el centro de masas del patrón ideal en concreto.

Por último lo que realicé para ver que efectivamente estaba calculando bien el centro de masas fue utilizar la función `drawCircleSub()` para dibujar en pantalla dicho centro, siendo este el resultado:

Como se puede ver, parece que el centro está algo desplazado, pero esto es normal pues hemos dibujado dos patrones que se tratan de figuras cerradas, haciendo que dos de sus puntos, en concreto el inicial y final, sean prácticamente el mismo y por tanto haciendo que este centro se desvío hacia donde se encuentran estos. Esto no es gran problema, pues este desvío será así para todas las figuras y no afectará negativamente al cálculo.

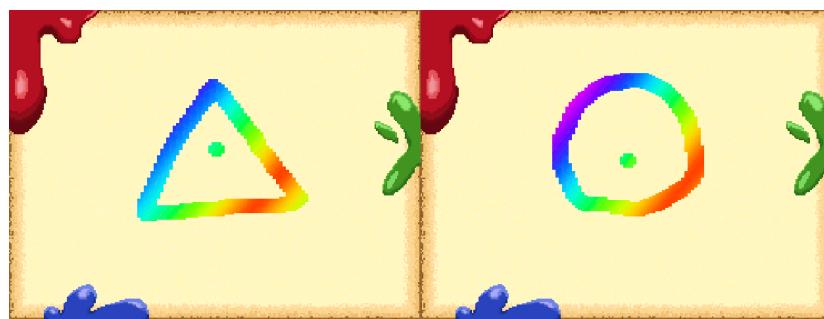


Figura 5.37: Dibujado de un trazo y señalización de su centro de masas

Lo último que realicé durante esta iteración fue comenzar a darle algunos toques visuales al juego, comenzando por las animaciones de los sprites.

Comencé por la animación de la protagonista que, a pesar de encontrarse en el centro de la pantalla y no moverse, decidí darle un toque más de vida y hacerle una animación para que pareciese que está preparada para luchar.

Usando el sprite inicial, en GIMP separé por capas los distintos elementos que quería mover, como la cabeza, cuerpo, manos y pies, y los fui colocando y creando los diferentes frames. El resultado fue el siguiente:



Figura 5.38: Frames de la animación del jugador

Para implementar la animación en el juego simplemente cargué todos los frames consecutivamente. Después, en la clase Player añadí una variable de tipo entero llamada animation_clock, que es inicializada a un valor concreto. En cada iteración este valor se decrementará en una unidad y cuando llegue a 0 llamaremos a una función de la fachada para que se encargue de cambiar el sprite al siguiente frame.

Código 5.24: Animación del sprite del jugador

```

1 void Player::update(){
2
3     if(animation_clock == 0){
4         animation_clock = 8;
5         Graphics::Instance()->animatePlayer(getOAMid());
6     }else{
7         animation_clock--;
8     }
9 }
10
11 [...]
12
13 void Graphics::animatePlayer(int id){
14     SpriteEntry* sprite = &oam->oamBuffer[id];
15
16     if(sprite->gfxIndex < 320){
17         sprite->gfxIndex+= 64;
18     }else{
19         sprite->gfxIndex = 64;
20     }
21
22     updateOAM();
23 }
```

Ahora bien, si estamos hablando de animaciones, ¿por qué no he usado un reloj para saber cuánto tiempo ha transcurrido entre ciclos? Pues porque después de valorarlo, no me parecía necesario. Nuestro juego va a ser un programa que se ejecute siempre en el mismo procesador, en el ARM9 y ARM7 de la NDS, es por ello que siempre nos vamos a asegurar de que en todas las consolas nuestro programa funciona exactamente igual, pues sus componentes son los mismos. Sin embargo hay una excepción a esto: la NDSi.

Como ya hemos comentado anteriormente, la NDSi fue un modelo actualizado de la familia NDS que no solo añadía características innovadoras como las dos cámaras, sino que también posee mejoras a nivel de hardware. El procesador ARM9 es más veloz que el que posee la NDS original, mientras que el ARM7 se mantiene igual. Pero si esto es así, ¿significaría que nuestro juego funcionará más rápido en la NDSi? Pues no, porque Nintendo a la hora de desarrollar esta consola creó el llamado "DSi mode", que es una característica que porta el cartucho del programa y que le dice a la consola si dicho programa puede usar todos los nuevos recursos de DSi. Por supuesto, los juegos desarrollados hasta la fecha de salida de la NDSi no tenían este

modo, así que a efectos prácticos funcionaban igual en ambas consolas. En cuanto a nuestra parte, lo único que nosotros podemos usar para poder probar nuestro juego en una NDS real son los cartuchos flash, y el único que tiene soporte para este modo se trata de CycloDSi Evolution, el cual fue inhabilitado por Nintendo mediante una actualización al software de DSi debido a temas de piratería.

En resumen, nuestro juego va a funcionar siempre con la misma frecuencia de procesador, por ello no tenemos que preocuparnos de que las animaciones puedan verse afectadas al no usar un reloj.

5.2.7 Iteración 6 - Mejoras visuales, corrección de niveles y mejora del producto final

Al tratarse esta de la penúltima iteración, me centré en acabar algunos aspectos de feedback visual y jugabilidad, así como también de producto.

En primer lugar, siguiendo donde dejé la iteración pasada las animaciones, quise incorporar las animaciones de los enemigos al moverse.

Al igual que hice con el jugador, primero abrí en GIMP el sprite del enemigo para que me sirviese de base para crear los frames de la animación, aunque en este caso al tratarse de un movimiento más complejo sí que tuve que ir deteniéndome en cada frame a dibujar sus piernas en cada posición. El resultado fue el siguiente:



Figura 5.39: Frames de la animación del enemigo

Seguidamente, para incorporarlo al juego usé nuevamente un contador a modo de temporizador, sin embargo la diferencia de éste es que solo se iba a actualizar si el enemigo se estaba moviendo, es decir, que aún no hubiese alcanzado al jugador.

Código 5.25: Animación del sprite del enemigo al moverse

```
1 void Enemy::update(){
2     if(!checkCollision()){

3 }
```

```
4      x = x + inc_x;
5      y = y + inc_y;
6
7      //actualizamos la animación al igual que haciamos con el jugador
8      if(animation_clock == 0){
9          animation_clock = 8;
10         Graphics::Instance()->animateEnemyRun(getOAMid());
11     }else{
12         animation_clock--;
13     }
14
15 }else{
16     attack();
17     //Como ha llegado hasta el jugador, le asignamos el sprite de estar ←
18     ↗ parado
19     Graphics::Instance()->resetEnemySprite(getOAMid());
20 }
21
22 }
23
24 [...]
25
26 void Graphics::animateEnemyRun(int id){
27     SpriteEntry* sprite = &oam->oamBuffer[id];
28
29     if(sprite->gfxIndex < 256){
30         sprite->gfxIndex+= 64;
31     }else{
32         sprite->gfxIndex = 64;
33     }
34 }
35 }
```

Después de incorporar esto al juego, cambié el fondo del nivel al que sería el definitivo. Para ello, como siempre primero realicé un boceto previamente y una vez estaba satisfecha con la idea, lo realicé en pixelart con la herramienta GIMP. Una vez terminado, lo incluí al juego simplemente sustituyendo el archivo por el que previamente estaba usando como fondo de nivel.

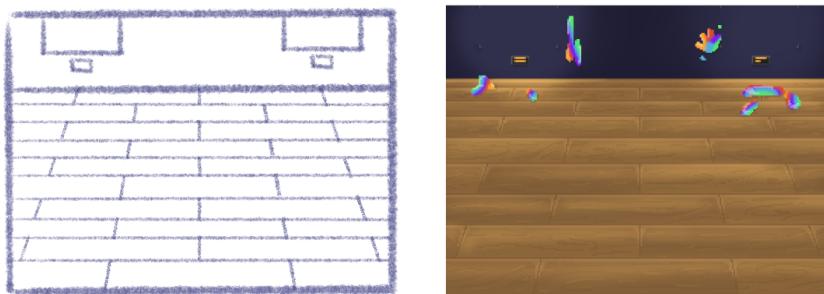


Figura 5.40: Boceto del fondo de nivel y versión acabada

Una vez terminado esto, la siguiente tarea que realicé fue una que contribuiría al que el juego se empezase a ver como un producto más acabado. La NDS, al tener un firmware que se ejecuta al encenderla nos deja ver algunas características de los juegos que tiene insertados en sus ranuras de NDS y GBA. Estas características son el título del juego y un pequeño ícono.

Hasta ahora, al estar usando el Makefile que venía en los ejemplos de devkitPro nos asignaba un nombre y un ícono por defecto como podemos ver en la siguiente figura.

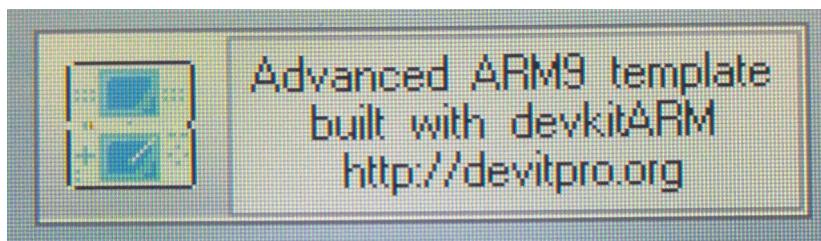


Figura 5.41: Ícono y título por defecto de un proyecto de NDS que use devkitARM

No obstante, dicho archivo ya va preparado para que puedas cambiarlos sin tener que indagar mucho en el código. En concreto, tenemos que buscar las variables con nombre GAME_TITLE y GAME_SUBTITLE1 para cambiar el título y la variable ICON para cambiar el ícono de la aplicación. A esta última debemos asignarle la ruta a un archivo de 32x32 píxeles y formato BMP de 16 colores (siendo el primer color el transparente). Así pues creé un ícono de esas características nuevamente en GIMP y lo incluí al proyecto. El siguiente código muestra los cambios que realicé al Makefile para añadir estas nuevas propiedades al proyecto.

Código 5.26: Cambios en el Makefile para establecer el título y el ícono

```

1 [...]
2
3 ICON := ../icon.bmp
4
5 [...]
6
7 # These set the information text in the nds file
8 GAME_TITLE := Touch & Brush
9 GAME_SUBTITLE1 := Carla Macia Diez
10 GAME_SUBTITLE2 :=
11
12 [...]

```

Al probar este cambio en la consola real, vi que el título se había cambiado sin problema, sin embargo la imagen parecía tener los colores muy diferentes, aunque la forma se podía visualizar.

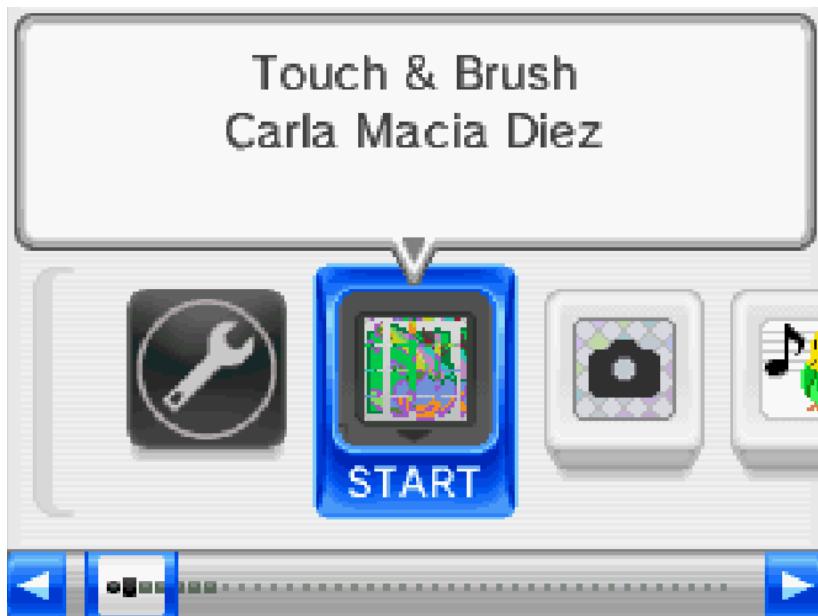


Figura 5.42: Erronea visualización del ícono de la aplicación (firmware de NDSi emulado en no\$gba)

Pensé que esto podía ser un problema de mi imagen, así que por asegurarme la reemplacé por la que usa por defecto y el problema seguía persistiendo. Fue entonces cuando pensé que a la hora de compilar, debía estar haciendo algún cambio a la imagen original para que se viese de esa forma. Me fijé en la terminal a la hora de compilar el proyecto, y vi que aparecía un mensaje como el siguiente:

```

devkitpro/portlibs/nds/include -I/opt/devkitpro/
/shiro/Escritorio/pbrush_tfg/build -DARM9 -fno-ro-
ro/Escritorio/pbrush_tfg/source/Vector2D.cpp -o
linking pbrush_tfg.elf
convert icon.bmp
Nintendo DS rom tool 2.1.2 - Jun 13 2018
by Rafael Vuijk, Dave Murphy, Alexei Karpenko
built ... pbrush_tfg.nds

```

Figura 5.43: Mensaje de conversión del Makefile

Esto ya me parecía raro, pues si la propia documentación te explica que el formato debe ser BMP, ¿por qué lo convierte? Así pues busqué en qué parte del Makefile realizaba esa conversión y encontré lo siguiente:

Código 5.27: Conversión del archivo BMP a GRF en el Makefile

```

1
2 #-----→
3 # Convert non-GRF game icon to GRF if needed →
4 #-----→
5 $(GAME_ICON): $(notdir $(ICON)) →
6 ##-----→
7   @echo convert $(notdir $<)
8   @grit $< -g -gt -gB4 -gT FF00FF -m! -p -pe 16 -fh! -ftr
9
10 -include $(DEPSDIR)/*.d

```

Fue entonces cuando se me ocurrió probar a quitar dicha conversión a GRF por ver si el formato de mi archivo era válido, así que comenté esa parte del código y asigné directamente a la variable GAME_ICON la ruta a mi archivo. Y en efecto, ahí estaba el problema, ahora el ícono se veía sin ningún problema.

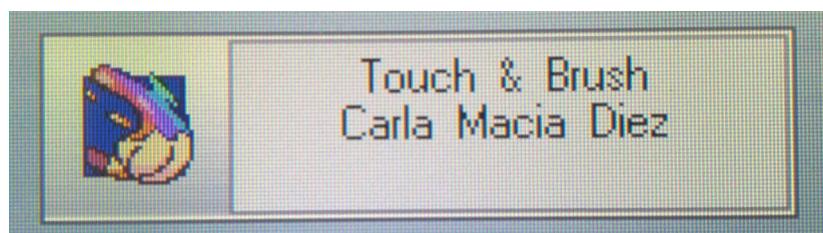


Figura 5.44: Frames de la animación del enemigo

Ya finalizada esta tarea, seguí con otro aspecto visual de la pantalla del menú principal. Quería asignarle los fondos definitivos tal y como se especifican en el apartado de Pantallas del Diseño del juego.

Comencé por la pantalla inferior, donde quería colocar un fondo con un mensaje de "Tocar la pantalla para jugar". Además, quería que este siguiese la estética también definidos en el apartado mencionado del Diseño del juego, por ello quise que este texto fuese cambiando de color pasando por todos los tonos, dando así un aspecto de arcoíris.

Para lograr esto, primero hice el fondo base con el texto sobre él y dicho texto le asegné un color llamativo como es el fucsia. Después, incluí el fondo al proyecto e hice que se dibujase también cuando se carga el menú principal en la función `loadMenu()` de la clase `Graphics`. Una vez hice esto, usando la herramienta de depurado visual de la gráficos de `no$gba`, vi en qué posición o posiciones de la memoria de paletas se encontraba este color.

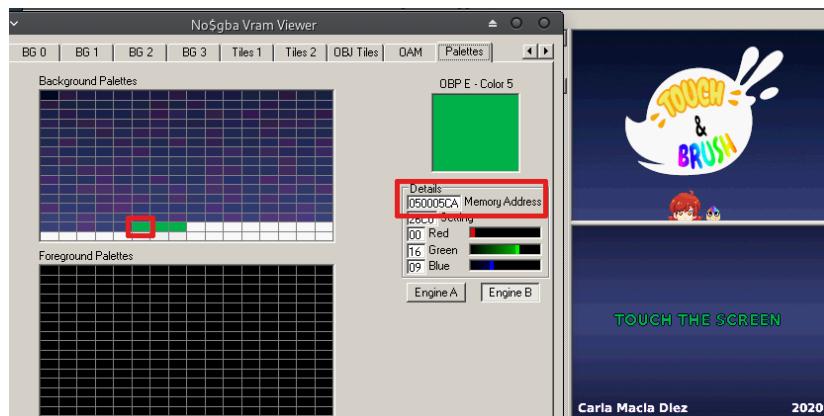


Figura 5.45: Posición en memoria del color específico

Esta posición de memoria la guardé en mi código y mi idea era, en la función `update()` de `Menu`, que iba a ser llamada una vez por ciclo, cambiar dicho color. Pero ahora bien ¿cómo podía cambiar dicho color de manera que crease un gradiente y no fuese brusco?

Para ello, busqué información acerca de algoritmos de degradados de colores RGB, y encontré que lo que hacían era usar las componentes del color roja, azul y verde y las hacían aumentar o decrementar para conseguir convertirlos en un valor del espectro HVS. En concreto, seguían los valores de la siguiente figura:

Aprendido esto creé unas variables de tipo entero `r`, `g` y `b` en mi clase `Graphics` y creé también una función llamada `fadeTitleText()` que implementa el cambio de dichos valores. Cabe destacar que el código original de esta implementación no es propio mío, sino se encuentra publicado en la web Codepen, y es el siguiente:

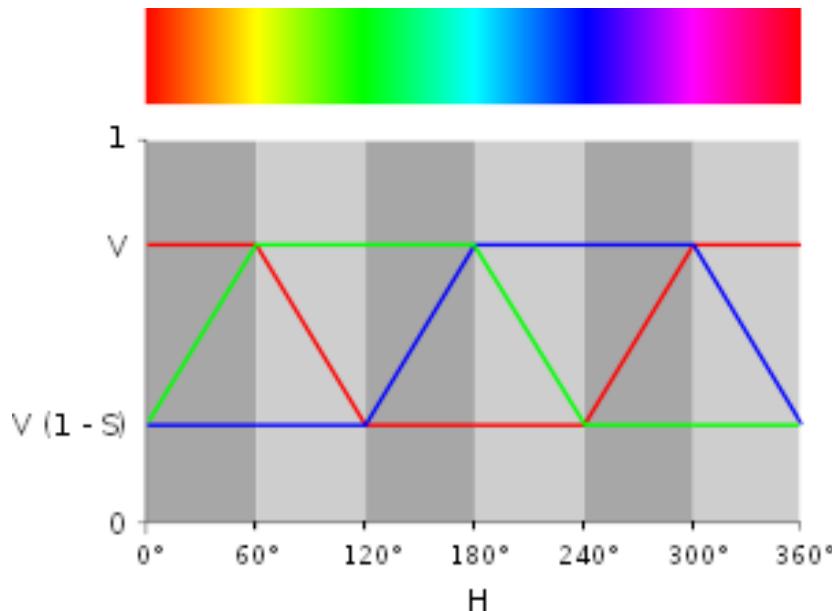


Figura 5.46: Valores RGB asignados al espectro HSV

No obstante aún faltaba una cuestión, cómo transformamos estos tres valores en algo que podamos copiar en la memoria de paletas de la NDS. Para ello, libnds nos lo pone fácil, pues posee una macro llamada `RGB15`, que convierte los valores `r` `g` y `b` de 5 bits a un conjunto de 15 bits. Cabe destacar, que como estamos trabajando con componentes de 5 bits, el valor máximo de cada componente es 31, y no 255 como en el algoritmo original.

Código 5.28: Animación del texto

```

1 void Graphics::fadeTitleText(){
2
3     //source of the algorythm: CodePen
4     //https://codepen.io/Codepixl/pen/ogWWaK/
5
6     if(r > 0 && b == 0){
7         r--;
8         g++;
9     }if(g > 0 && r == 0){
10        g--;
11        b++;
12    }if(b > 0 && g == 0){
13        r++;
14        b--;
15    }
16
17     unsigned short color = RGB15(r,g,b);
18     memcpy(TITLE_TEXT_COLOR,&color,sizeof(color));
19     memcpy(TITLE_TEXT_COLOR+1,&color,sizeof(color));
20     memcpy(TITLE_TEXT_COLOR+2,&color,sizeof(color));

```

21 }

Al probar esto y ver que efectivamente funcionaba bien, pasé a la pantalla superior. La idea de ésta era mucho más simple, pues solo quería que se viese el fondo con el logotipo y los personajes de la historia parpadeando.

Para realizar esto, me basé en cómo uno de nuestros referentes: Wario Ware: Touched, resuelve un problema similar. En las animaciones de las cinemáticas se utiliza mucho el recurso de separar la cara de los personajes entre el fondo y los sprites, así, las partes que siempre sean estáticas como por ejemplo la cara se mantienen ligadas a un fondo que siempre es el mismo y, las partes dinámicas como los ojos, boca, nariz, etc, se desarrollan como sprites, para que sea más fácil trabajar con ellos y ahorrar recursos.

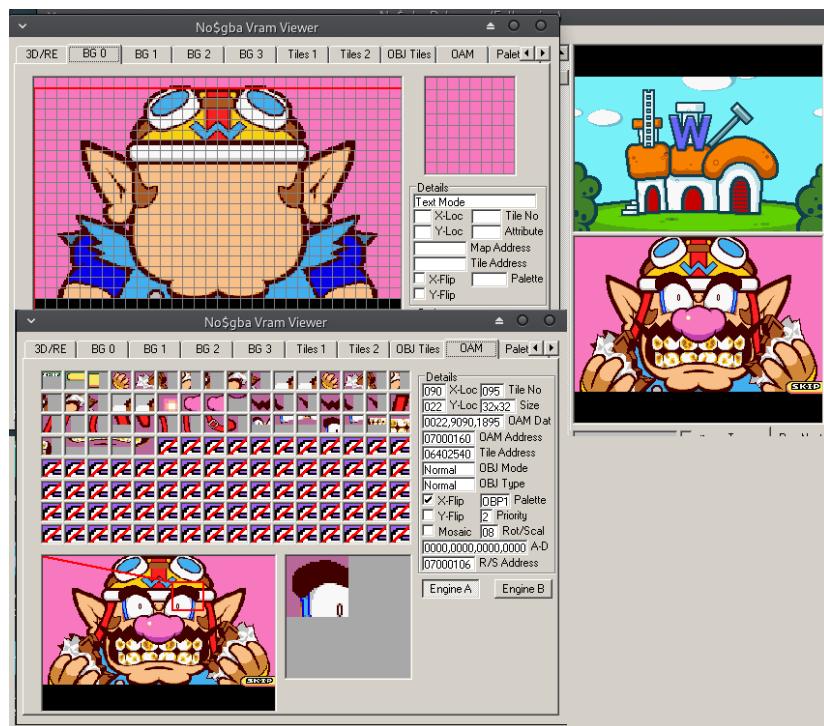


Figura 5.47: Implementación de Wario Ware: Touched de las animaciones para las cinemáticas

Entonces así lo hice, creé el fondo en GIMP que tenía el logotipo y las caras vacías de los protagonistas y, por separado, creé un sprite de los ojos de ambos. Incluí tanto el fondo como el sprite al menú, y mi idea para ahorrar recursos de la consola era hacer que el sprite de los ojos se escalase en el eje Y para dar la apariencia de estar pestañeando. No obstante, esto no funcionó, pues el centro de transformaciones de los sprites está situado en el centro y al escalarlo los ojos se movían de sitio. Al no poder cambiar este centro a diferencia del de los fondos, decidí hacer dicha animación por frames. Esta animación la realicé de la misma

manera que las animaciones del jugador y los enemigos.

En la siguiente figura se muestra el resultado final de la pantalla de inicio:

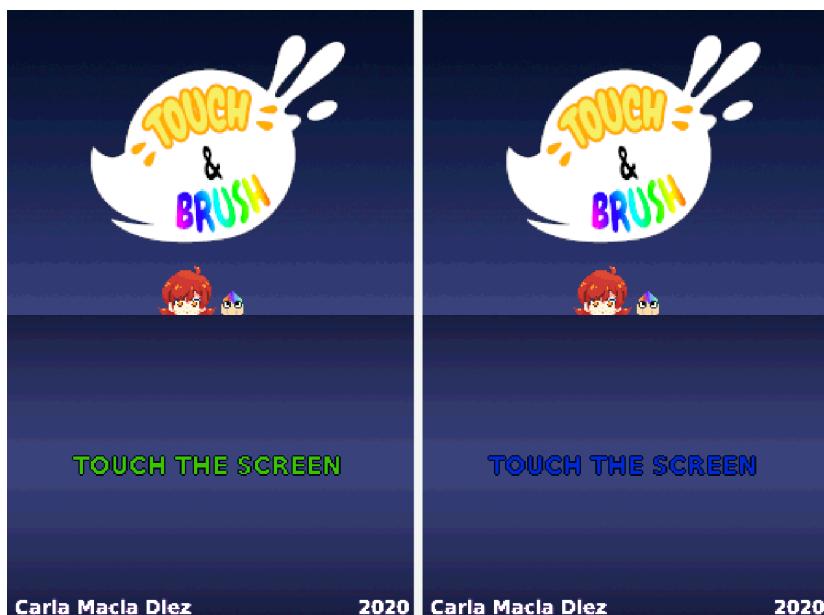


Figura 5.48: Aspecto final de la pantalla del menú principal

Una vez terminé los aspectos visuales, pasé a centrarme en aspectos de la jugabilidad y niveles. En concreto, implementé los niveles que se diseñaron en el apartado de Diseño del juego pero para poder hacerlo tuve que hacer un pequeño cambio a los enemigos.

En cada iteración del juego, los enemigos se mueven hacia el centro de la pantalla en una unidad, pues trabajamos con coordenadas de pantalla que representan números enteros. Ahí fue donde me surgió la duda de cómo implementar las distintas velocidades, pues si aumentaba estos incrementos podrían llegar a verse saltos bruscos en los enemigos. La solución que yo pensé para esto fue hacer que tanto el incremento de los enemigos como su posición en la lógica del juego fuesen variables que representen números decimales. Sin embargo, a la hora de asignar estos valores x e y a los sprites en cada iteración los convertiremos a un entero. Esto se puede ver mejor en un ejemplo.

Imaginemos un enemigo cuya velocidad es 0.5 en el eje X y se encuentra en la coordenada 0 en ese eje. En la primera iteración, su posición pasará a ser 0.5 en ese eje, sin embargo, a la hora de asignar la posición al sprite haremos un truncamiento a ese valor decimal y pasará a ser 0. En la siguiente iteración, al sumar nuevamente el incremento, esta vez la posición decimal será 1, con lo que el sprite sí avanzaría de posición.

Esta solución personalmente me parece efectiva y simple, pues conseguimos resolver el problema, establecer velocidades de forma más sencilla y no cambiar drásticamente la lógica de los enemigos ni añadirles más variables.

Por otro lado, lo último que realicé durante esta iteración fué un cambio en el diseño de estos niveles y donde aparecían los enemigos. La idea de establecer las posiciones desde las que salían los enemigos en seis puntos en la pantalla es una buena idea para un juego como Magic Cat Academy, donde la pantalla de juego es amplia y los enemigos pueden verse sin problema.

Sin embargo, en nuestro caso tenemos una pantalla bastante reducida y los enemigos que salen por la parte superior tienen la desventaja de que el patrón que poseen se llega a ver casi cuando estos están alcanzando al jugador. Esto podría ser un punto que jugase a nuestra contra en la experiencia de usuario, pues si le mata un enemigo de estos podría sentir que no ha sido justo ya que él no ha podido llegar a ver el patrón a tiempo.

Por esto, decidí cambiar dichas posiciones iniciales y reducirlas a 4 posibles. El jugador ahora estará colocado en el centro superior de la pantalla y, como antes, saldrán enemigos de los lados y de las esquinas inferiores.

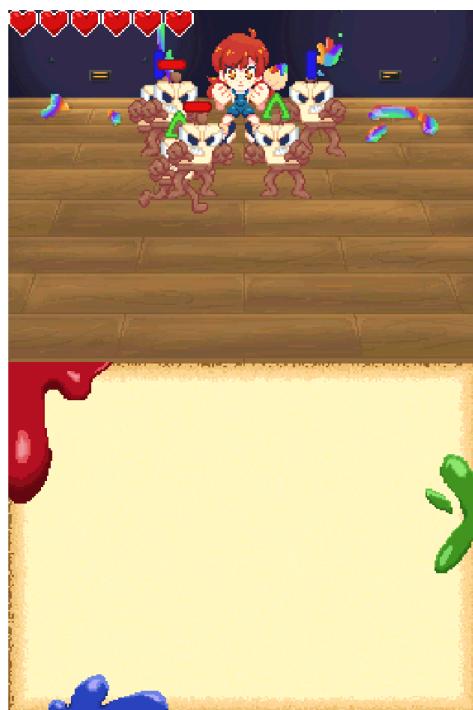


Figura 5.49: Posiciones finales de los enemigos.

Dediqué finalmente tiempo a probar muchas veces el juego, tanto yo como amigos y familia-

res, y mejoraré su calidad visual incluyendo sprites de ataque y recibir daño a la protagonista. También, cambié los sprites de la vida, ya que al ser tan apagados casi no se apreciaban.

Por otro lado, arreglé una serie de fallos, en concreto que cuando conseguías pasarte los niveles del juego y volvías al menú principal se quedaba el fondo del lienzo dibujado en la pantalla inferior. Esto es por el propio flujo del programa, ya que una vez detectamos que el usuario ha dejado de tocar la pantalla táctil, primero matamos a los enemigos y, si se considera que se ha completado el nivel, cargamos en memoria los fondos del menú principal. Sin embargo, después de eso también se llamaba al método de volver a dibujar la pantalla inferior para borrar el patrón que había dibujado el usuario. Esto estaba sobreescritiendo el fondo previamente cargado en memoria, por eso se visualizaba el del lienzo. Arreglé este fallo simplemente teniendo en cuenta que el número de enemigos a matar no fuese 0 (que significaría que ya se ha completado el nivel) para redibujar la pantalla inferior.

Por último, para mejorar la calidad del producto final y en cierto modo celebrar que el juego ya estaba completo, realicé una ilustración para la portada del videojuego.



Figura 5.50: Portada de Touch & Brush.

6 Conclusiones

En este apartado vamos a repasar el estado actual del proyecto, las posibles mejoras y ampliaciones a realizar en un futuro, todo lo que hemos aprendido y las conclusiones que hemos obtenido una vez ha finalizado este proyecto.

6.1 Estado del juego

Después de todo el trabajo realizado durante estos meses, Touch & Brush puede considerarse un **juego completo y acabado**, con un objetivo sencillo, que todos los usuarios pueden jugar y disfrutar de principio a fin tantas veces como deseen.

Se ha logrado implementar la gran mayoría de **mecánicas y aspectos del juego que se diseñaron previamente** y también se han logrado cumplir los objetivos que se marcaron al comienzo de este proyecto. Entre ellos, se ha logrado **estudiar la NDS y su hardware**, a la vez que se ha conseguido **diseñar y programar un juego para esta**. También, se ha **explicado toda la fase de desarrollo** del mismo en este documento, de modo que pueda servirle de referencia a cualquier persona que esté interesada en desarrollar para esta consola.

Además, el proyecto ha podido pasar por una última **fase de pruebas** para encontrar fallos, tanto a nivel de programación como de diseño, perfeccionando así el resultado final como producto.

A continuación se adjuntan una serie de imágenes que muestran el juego funcionando en una Nintendo DS Lite.



Figura 6.1: Menú principal y nivel de Touch & Brush

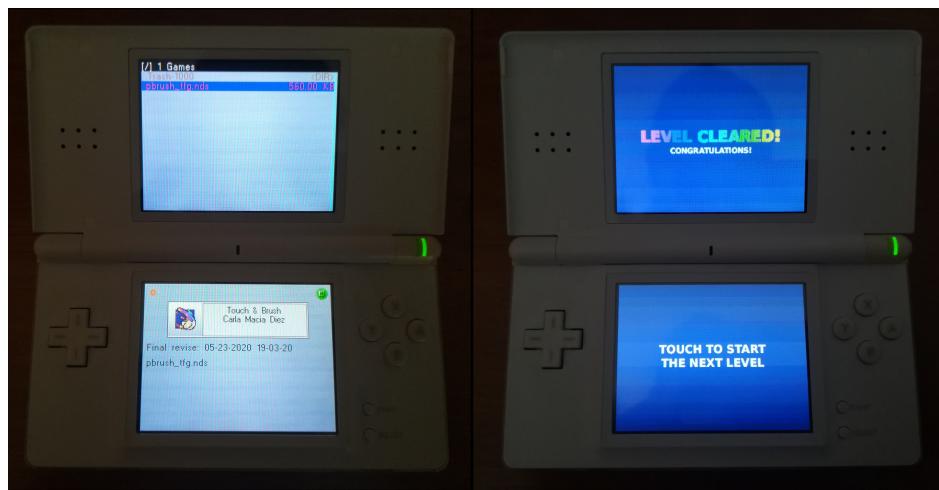


Figura 6.2: Pantalla de victoria y previsualización en la R4 de Touch & Brush

No obstante, para este proyecto existen diversas mejoras y ampliaciones las cuales pueden hacer que Touch & Brush se convierta en un juego que los usuarios disfruten más. Dichas mejoras se centran **principalmente en adición de contenido**, ya que como hemos comentado, el juego actualmente se encuentra acabado. Estas se mencionan en el siguiente apartado.

6.2 Mejoras

Como hemos comentado, Touch & Brush aún puede **mejorar** bastante más, si bien añadiendo **características que no se han podido implementar** o incluyendo **nuevas** ideas que han surgido durante el desarrollo. A continuación se detallan todas ellas.

- **Incluir los enemigos y funcionalidades diseñadas que no se han podido implementar:** La principal mejora que se le podría aplicar al juego es la de incluir algunos aspectos que se diseñaron pero que no se han podido implementar. Por ejemplo, entre ellos están el sistema de **puntuación** en los niveles, el nivel de **jefe final** y el **aliado** que te permite recuperar vida durante la partida. Sin duda alguna, estas mejoras tendrían mucha prioridad, pues aportarían al juego mucha más diversión y dinamismo.
- **Mejora de pantallas:** Se podrían añadir las pantallas que fueron diseñadas, principalmente la pantalla de fin de partida y la de victoria. A pesar de que la de victoria está implementada en el juego, no es exáctamente igual a cómo se diseñó debido a un desajuste en la planificación, pues faltaría dotarle de más opciones (volver al menú y continuar), así como la visualización de la puntuación.
- **Añadir más niveles:** Otra mejora sería la de incorporar más niveles al juego, con enemigos que posean patrones más complejos, para así aumentar el contenido y la duración del juego.
- **Música:** La incorporación de una banda sonora es algo que puede aportar mucho al resultado final de un juego, ya sea por las canciones principales, las cuales ayudan al jugador a manifestar emociones de tensión, felicidad, miedo, ... o por los efectos de sonido, que ayudan a informarle al usuario de qué está ocurriendo en el juego.
- **Mejorar las animaciones:** A pesar de que la parte gráfica no era algo de gran peso en este proyecto, hay que considerar que para futuras revisiones podrían mejorarse las animaciones existentes y añadir algunas más (por ejemplo, el ataque de los enemigos). Esto es importante ya que mejoraría la calidad del producto final y aportaría más información al usuario sobre lo que pasa durante la partida. Además, también se podrían añadir incluso **cinemáticas** que cuenten la historia de cómo la protagonista ha llegado a esa situación.
- **Incluir un sistema de ayuda:** Una posible mejora para el juego, que beneficiaría la experiencia de usuario, sería la inclusión de un pequeño **tutorial** al principio del primer nivel que le enseñase al usuario que debe dibujar los patrones de los enemigos, tal y como hace Magic Cat Academy.
- **Menú de ajustes:** Para permitir al usuario ajustar algunas opciones como el idioma, volumen de la música y efectos de sonido, e incluso dificultad. Esta idea surgió durante las últimas iteraciones, al dejar a amigos y familiares probar el juego casi acabado. Algunos de ellos, sobre todo aquellos que no estaban muy relacionados con los videojuegos, no entendían qué debían hacer y acababan perdiendo en el primer nivel. No obstante, una vez lo intentaban por segunda vez y comenzaban a probar cosas (generalmente a

tocar la pantalla), rápidamente ya comenzaban a matar a los enemigos y a avanzar en el juego.

- **Realizar la producción física del juego:** A pesar de que Touch & Brush es un juego acabado, la producción de un juego en un cartucho es algo costoso, y si bien no se ha realizado todavía, es algo que acabará ocurriendo. Es por ello que es preferible invertir ese esfuerzo en un producto que considere que está mejor acabado visualmente y posee más contenido.

6.3 Lecciones aprendidas

Enfrentarme a un proyecto como este ha sido una experiencia que me ha permitido **aprender muchísimo**, no solo de los aspectos que ya esperaba, sino de muchos otros más.

He podido aprender mucho sobre la NDS y su arquitectura, y tener que ajustarme a hacer un juego para una **máquina específica con limitaciones** ha hecho que en varias ocasiones haya tenido que tomar una serie de decisiones, que más tarde y al haber obtenido más conocimiento según proseguía el desarrollo, he podido evaluar.

También, como en cualquier otro proyecto de ingeniería, he aprendido a **dar solución a una serie de problemas** usando lo que previamente he investigado. Si bien para desarrollos de proyectos que usan librerías o *frameworks* más nuevos y con más soporte hay gran cantidad de ayuda e información en internet, para el caso de desarrollar para NDS es poco común ver eso. La gran mayoría de foros y librerías se encuentran desactualizados desde hace bastantes años, incluso la propia documentación de libnds me ha resultado escasa en muchas ocasiones. Esto ha hecho que a la hora de querer buscar más información acerca de, por ejemplo, una función, haya tenido que **ir también al propio código de la librería** y entender qué pretende hacer cada línea de código.

Por otro lado, este proyecto me ha permitido aprender sobre los **algoritmos de reconocimiento de gestos y escritura manuscrita**, así como las bases matemáticas que los sostentan. Solía ver este tipo de software una especie de caja negra y no pensaba que podía ser capaz de desarrollar el mío propio y funcionarse correctamente. Si bien esto ha sido algo que no esperaba al principio cuando planteé hacer un juego para NDS, estoy bastante contenta de haber podido aprender de ello.

Además, también he aprendido sobre la importancia de un **buen diseño del juego inicial**, pues durante la fase de desarrollo el hecho de tener todos los aspectos del juego detallados (pantallas, mecánicas, comportamiento de los enemigos), me ha facilitado su implementación.

Por último, otro aspecto de mucha importancia durante este proyecto ha sido la **planificación**. Enfrentarse a un proyecto como este en solitario ha sido algo duro, y han surgido contratiempos en muchas ocasiones. Sin embargo, gracias a una buena planificación y dotar

a las tareas de un orden de prioridad de modo que se podría conseguir un producto lo antes posible es lo que ha permitido que el proyecto haya tenido un buen resultado.

6.4 Conclusión personal

Personalmente, al haber podido llevar a cabo este proyecto me siento **muy satisfecha**, pues como ya he comentado, hacer un juego para NDS o crear un algoritmo de reconocimiento de escritura eran algo que hace un año me preguntaba cómo se hacían y veía fuera de mi alcance. Esto además, también me aporta la **confianza** necesaria para enfrentarme a **proyectos futuros** sin miedo a su complejidad.

Este ha sido un proyecto largo y en muchos momentos duro de afrontar. Especialmente porque ha habido momentos en los que mi motivación se ha visto muy reducida, ya sea por problemas personales, familiares o incluso la situación de la pandemia global provocada por el SARS-CoV-2. En cuanto a éste último, me gustaría señalar y agradecer la dedicación y esfuerzo de los profesores de la Universidad de Alicante, y en especial a mi tutor, Francisco Gallego, por su disponibilidad y ayuda a pesar de toda la situación.

No cabe duda de que seguiré mostrando a las personas Touch & Brush para que puedan probarlo y obtener así nuevas opiniones e ideas para mejorarlo, haciendo que se convierta en un juego digno de competir con algunos títulos para la misma consola. El resultado de este proyecto me ha dejado tan contenta que mis ganas e ilusión de seguir trabajando en él en un futuro son casi las mismas que tuve al comenzarlo.

Como ya he comentado anteriormente, la NDS es una consola a la que siempre le he tenido mucho cariño, llegando a ser mi consola portátil preferida. He invertido mucho tiempo de mi infancia jugando a muchos de sus grandes títulos, y sin duda alguna haber conseguido hacer un juego completo para una cosa tan especial me hace inmensamente feliz.

Bibliografía

- ARM. (2000). *ARM9E-S (Rev 1) Technical Reference Manual*. <http://ww1.microchip.com/downloads/en/DeviceDoc/doc6178.pdf>.
- ARM. (2001). *ARM7TDMI (Rev 3) Technical Reference Manual*. http://ww1.microchip.com/downloads/en/DeviceDoc/DDI0029G_7TDMI_R3_trm.pdf.
- azyDream. (2008). *Palib - Shape Recognition Principle*. <https://azydream.tistory.com/55>.
- Chuan Ji (jichu4n). (2018). *Archive of Palm OS SDKs*. <https://github.com/jichu4n/palm-os-sdk>.
- devkitPro. (2008). *PAlib - devkitPro*. <https://devkitpro.org/wiki/PAlib>.
- Google. (2016). *Halloween 2016 (Doodle Magic Cat Academy)*. <https://www.google.com/doodles/halloween-2016>.
- Jaeden Amero. (2008). *Introduction to Nintendo DS Programming*. <https://patater.com/files/projects/manual/manual.html#id2614177>.
- Jasper Vijn. (2004). *Tonc*. <http://www.coranac.com/tonc/text/log.htm>.
- Joel Sherrill (joelsherrill). (2014). *PAlib Recognition algorithm from RTEMS*. <https://github.com/atgreen/RTEMS/blob/master/c/src/lib/libbsp/arm/nds/touchscreen/reco.c>.
- Martin Korth. (2014). *GBA/NDS Technical Info*. <http://problemkaputt.de/gbatek.htm#aboutthisdocument>.
- Nintendo. (2005). *WarioWare: Touched!* <https://www.nintendo.es/Juegos/Nintendo-DS/WarioWare-Touched--273564.html>.
- Nintendo. (2009). *Una pausa con... Brain Training Ciencias*. <https://www.nintendo.es/Juegos/Nintendo-DSiWare/Una-pausa-con-Brain-Training-Ciencias-261960.html>.
- Nintendo. (2020a). *Nintendo DSi*. <https://www.nintendo.es/Nintendo-DS/Nintendo-DSi/Nintendo-DSi-Pagina-web-oficial-de-Nintendo-Iberica--116441.html>.
- Nintendo. (2020b). *Nintendo DSi XL*. <https://www.nintendo.es/Nintendo-DS/Nintendo-DSi-XL/Nintendo-DSi-XL-Pagina-web-oficial-de-Nintendo-Iberica--116442.html>.

- Nuria Martínez Soriano. (2011). *DESARROLLO DE APLICACIÓN SOBRENINTENDO DS PARA TRANSFERENCIA DE DATOS VÍA WI-FI*. <https://riunet.upv.es/bitstream/handle/10251/11576/memoria.pdf?sequence=1>.
- Olivier Boudeville. (2008). *A guide to homebrew development for the Nintendo DS*. <http://osdl.sourceforge.net/main/documentation/misc/nintendo-DS/homebrew-guide/HomebrewForDS.html#andnow>.
- PAlib. (2008). *PAlib: Shape Recognition*. http://cravesoft.free.fr/PAlibDocEng/html/group___reco.html#ga925cec3c4c9e826e8c07ee5a2366feb.
- Thomas Farrell, Connor Petilli. (2015). *ARCHITECTURE OF THE NINTENDO DS*. <http://meseec.ce.rit.edu/551-projects/fall2015/3-1.pdf>.
- Vandal. (2006). *Análisis Lost Magic*. <https://vandal.elespanol.com/analisis/nds/lost-magic/5265#p-29>.
- Wikipedia. (2002). *ARM architecture*. https://en.wikipedia.org/wiki/ARM_architecture#32-bit_architecture.
- Wikipedia. (2004). *Nintendo DS*. https://en.wikipedia.org/wiki/Nintendo_DS.
- Wikipedia. (2006a). *Comunicación entre procesos*. https://es.wikipedia.org/wiki/Comunicaci%C3%B3n_entre_procesos.
- Wikipedia. (2006b). *Lost Magic*. <https://en.wikipedia.org/wiki/LostMagic>.
- Wikipedia. (2006c). *Nintendo DS Lite*. https://en.wikipedia.org/wiki/Nintendo_DS_Lite.
- Wikipedia. (2006d). *WarioWare: Touched!* https://es.wikipedia.org/wiki/WarioWare:_Touched!
- Wikipedia. (2008a). *Brain Age Express*. https://en.wikipedia.org/wiki/Brain_Age_Express.
- Wikipedia. (2008b). *Nintendo DSi*. https://en.wikipedia.org/wiki/Nintendo_DSi.
- Wikipedia. (2009). *Graffiti (Palm OS)*. [https://es.wikipedia.org/wiki/Graffiti_\(Palm_OS\)](https://es.wikipedia.org/wiki/Graffiti_(Palm_OS)).
- Wikipedia. (2010). *Scene*. <https://es.wikipedia.org/wiki/Scene>.
- Wikipedia. (2011). *Reduced instruction set computing*. https://es.wikipedia.org/wiki/Reduced_instruction_set_computing.
- Wikipedia. (2013). *Nintendo DS Family*. https://en.wikipedia.org/wiki/Nintendo_DS_family.
- Xerox Corp. (1995). *Unistrokes for computerized interpretation of handwriting*. <https://patents.google.com/patent/US5596656>.

7 Anexo I - Detalles técnicos de Nintendo DS

7.1 Procesadores

Como hemos mencionado la Nintendo DS posee dos unidades de procesamiento que realizan tareas diferentes y específicas. Vamos a profundizar continuación un poco en dichos procesadores para conocer algunos datos técnicos sobre ellos y los trabajos que llevarán a cabo en cualquier juego que se ejecute en la NDS.

7.1.1 ARM9

Su nombre completo es **ARM946E-S** y pertenece a la familia de procesadores de **32 bits**, con una arquitectura **RISC**¹, conocida como ARM9. Aunque oficialmente se conozca ARM9 como el nombre que se le da a esta familia de procesadores, nosotros durante este documento usaremos ese término para referirnos al procesador ARM946E-S en concreto.

Este procesador es capaz de trabajar a una frecuencia de **66MHz**, haciéndolo el más potente de los dos procesadores de la NDS. Además, su diseño está basado en la **arquitectura Harvard** para el acceso a memoria, lo cual quiere decir que la memoria está separada en memoria de datos y memoria de instrucciones, permitiéndole acceder simultáneamente a ambas y mejorando su rendimiento a cambio de una mayor complejidad.

Por otro lado, el ARM9 usa un *pipeline*² de **5 pasos**. Primero, obtiene la instrucción a realizar de una cola de prioridad, después la decodifica, seguidamente la ejecuta, accede a datos en memoria (si la instrucción lo requiere), y finalmente vuelve a escribir en los registros de las instrucciones.

Por último debemos comentar las tareas de este procesador. En general, se encarga del **90% de la ejecución** de los juegos de NDS, tanto toda la lógica, algoritmos, IA e incluso la parte gráfica. Sin embargo, en la ejecución de **juegos de GBA no se usa**, dejándole esa tarea al ARM7.

¹RISC: del inglés *Reduced Instruction Set Computer*, es un tipo de diseño de procesadores que tiene como objetivo reducir los accesos a memoria y con ello hacer más eficiente la ejecución de instrucciones.

²En informática, un *pipeline* o tubería se refiere a una lista de procesos consecutivos donde la entrada de uno es la salida del que le precede.

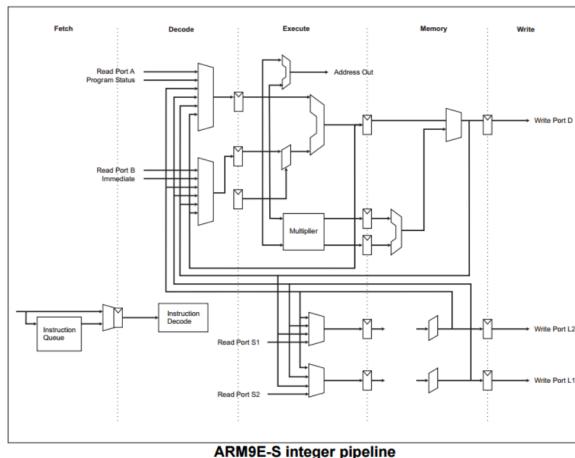


Figura 7.1: Representación gráfica de la tubería de procesos del ARM9

Fuente: Thomas Farrell y Connor Petilli

7.1.2 ARM7

Al igual que con el ARM9, su nombre completo es **ARM7TDMI**, pero durante este documento nos referiremos a él como ARM7. Este procesador es también de la familia de procesadores ARM7, de arquitectura **RISC** de **32 bits**.

Es algo **menos potente** que el anterior, pues su velocidad es de **33MHz** en la ejecución de **juegos de NDS** y **16MHz** en los de GBA. Además, se diferencia con respecto al ARM9 en su diseño en lo referente al acceso a memoria, pues este sigue una **arquitectura de Von Neumann**. Mientras que el ARM9 separaba entre memoria de datos y memoria de instrucciones teniendo dos buses de datos distintos, el ARM7 tiene un único bus de 32 bits para estos.

Por otro lado, su *pipeline* se basa únicamente en **3 pasos**, siendo la obtención de instrucciones de memoria, decodificación y ejecución de estas. No posee una memoria caché³ para instrucciones o datos, pero esto se ve recompensado con una mayor velocidad en el flujo de las instrucciones.

Por último, este procesador no juega un papel tan importante como el ARM9, pero sí realiza tareas de gran peso, como **ejecutar los juegos de GBA**, encargarse del **sonido** (incluido del uso del micrófono), **conexión Wi-Fi** y acceso a la **pantalla táctil**. Sin embargo, **los desarrolladores no pueden hacer uso de este procesador**, de modo que deben darle las instrucciones al ARM9 y éste es quien se comunica con el ARM7 para encargarle las tareas a realizar. Esta comunicación entre procesadores se lleva a cabo gracias al **protocolo IPC**⁴

³Componente hardware que guarda datos de recurrente uso en una memoria más accesible para el procesador de modo que en futuras solicitudes de estos datos el tiempo de lectura de estos será menor.

⁴IPC: del inglés *Inter-Process Communication*, hace referencia a los mecanismos que un sistema operativo ofrece para que varios procesos puedan acceder a la vez a memoria compartida.

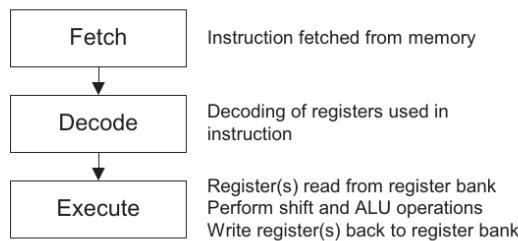


Figura 7.2: Representación gráfica de la tubería de procesos del ARM7

Fuente: ARM

y haciendo uso de una serie de registros con estructura **FIFO**⁵.

7.2 Memoria de vídeo

La NDS tiene dos pantallas, las cuales son manejadas por el ARM9 haciendo uso de **dos motores gráficos** que pueden renderizar gráficos 2D y 3D (este último solo el motor de la pantalla superior).

Para que estos motores puedan operar, necesitarán hacer uso de imágenes que se usarán para fondos o texturas, *tiles*⁶, *sprites*⁷ e incluso coordenadas de vértices. Estos datos se encuentran en una zona de memoria específica para estos datos: la memoria de vídeo o **VRAM**.

En total, la memoria de vídeo tiene una **capacidad de 656Kb** distribuída en **9 bancos** diferentes que son nombrados de la A a la I. Cada banco posee un tamaño distinto ya que están pensados para almacenar datos con distinto fin. Por ejemplo, los bancos de mayor capacidad están diseñados para guardar datos referentes a los fondos, mientras que los más pequeños se suelen usar para memoria de tiles o sprites. A continuación se muestra una tabla con todos los bancos y sus tamaños.

Así pues, antes de copiar nuestros datos a memoria y usarlos, debemos mapear la memoria de vídeo a los bancos que vayamos a usar y especificar también a qué se van a dedicar.

⁵FIFO: del inglés *First In, First Out*, es un concepto que se refiere a que las instrucciones se realizan por orden de llegada.

⁶Los *tiles* son un conjunto de segmentos o divisiones de un mapa de bits y con un ancho y alto determinado, que al colocarlos todos juntos y en orden forman la imagen completa. Serían como las piezas de un puzzle vistas individualmente.

⁷Un *sprite* es, en un videojuego, cualquier mapa de bits que posea un tamaño pequeño y represente a todos los objetos que no forman parte del fondo (enemigos, jugador, vidas, interfaz...).

Banco	Tamaño (Kb)
VRAM_A	128
VRAM_B	128
VRAM_C	128
VRAM_D	128
VRAM_E	64
VRAM_F	16
VRAM_G	16
VRAM_H	32
VRAM_I	16

Tabla 7.1: Tamaños de los 9 bancos de VRAM de la NDS.

Por otro lado, cada motor puede operar en **5 modos** distintos (el motor correspondiente a la pantalla superior posee un modo extra para mapas de bits de mayor tamaño y la posibilidad de renderizar gráficos 3D).

Los modos se diferencian entre ellos de la manera o formato en el que **describen los pixeles**. Así, un modo puede definirse según cómo caracteriza los siguientes valores del pixel:

- **Tamaño.** Por ejemplo, cada pixel ocupa 16 bits.
- **Tipo.** Referente al color. Si por ejemplo posee los valores RGB directamente en su información o una referencia a la memoria de paletas.
- **Orden en memoria de los componentes del color.** Por ejemplo, si es RGB, el azul es el menos significativo, con lo cual su valor estaría codificado en los bits menos significativos.
- **Tamaño de cada componente del color.** Por ejemplo, si a cada componente le damos 5 bits, su valor iría de 0 a 31. En binario, veríamos que un pixel de estas características tendría la siguiente distribución: xRRRRRGGGGGGBBBBB.
- **Rol del bit más significativo.** Hay un bit "sobrante" en el ejemplo anterior, y por lo general se puede especificar si ese bit hace referencia a la transparencia del pixel (0 si no se debe visualizar, 1 si se debe visualizar).

7.3 OAM

La **OAM**, del inglés, *Object Attribute Memory* es una sección del hardware de la NDS donde se almacenan **datos relativos a los sprites de nuestro juego**.

Como hemos comentado en la sección de Memoria de vídeo, nosotros podemos mapear un banco de VRAM para usarlo como almacenamiento de los datos referentes a las imágenes de los *sprites*, mientras que en la OAM guardaremos información relativa a dónde se encuentran esos *sprites* dibujados en pantalla (coordenadas x e y), si están o no visibles, si están invertidos en uno de los ejes o en ambos, si se les puede aplicar transformaciones afines... En definitiva, en la OAM almacenamos datos para trabajar con esos *sprites*.

En un mismo instante podemos tener **hasta 128 sprites**, numerados del 0 al 127, y a pesar de que podemos especificarles una prioridad para que se dibujen unos por encima de otros, para un mismo nivel de prioridad (supongamos que dos *sprites* tienen el nivel de prioridad 2), aquel *sprite* cuyo número sea menor aparecerá por encima. Además, de esos 128 *sprites*, podemos tener hasta **32 de ellos afines**, lo cual quiere decir que se les puede aplicar operaciones de rotación y escalado.

Por otro lado, en cuanto al uso de **paletas de color** en los *sprites*, podemos elegir entre tres opciones:

- Podemos tener **16 paletas**, cada una de 16 colores. Cada *sprite* puede usar una de estas paletas.
- Podemos tener **una única paleta** de 256 colores, todos los *sprites* usan la misma.
- **No establecemos una paleta**, el color viene especificado en la información del *sprite*. Esta no es una muy buena opción, pues consume muchísima más memoria que las otras dos anteriores.

7.4 DMA

Hemos estado hablando de las zonas de memoria dedicadas al almacenamiento de nuestras imágenes y datos relativos a la parte gráfica, pero para poder almacenarlos debemos tener un mecanismo que nos permita **copiar o mover esos datos** desde el propio cartucho ROM (en el cual profundizaremos más adelante) hasta estas zonas *hardware*.

El **DMA** significa *Direct Access Memory* y es, en efecto, un **método de copia rápido y eficiente para la CPU** que funciona por hardware. Disponemos de **4 canales** por los cuales podemos transferir datos, numerados del 0 al 3, y pueden trabajar de manera **asíncrona**. Para una copia a través de DMA, la CPU (el ARM9) inicializa esta transferencia y, si la copia no requiere acceder a la memoria principal, el procesador puede seguir ejecutando instrucciones. Una vez se haya realizado la transferencia completa, mediante una interrupción se notifica al ARM9 de que la copia se ha realizado con éxito. Esta es una buena forma de asegurarnos de que, por ejemplo, si vamos a dibujar un fondo en pantalla, sepamos que ese fondo ha sido copiado para evitarnos dibujar lo que fuese que había en memoria antes de su copia.

Por supuesto, este no es solo el único método de copia que tenemos disponible, también hay otros métodos clásicos que funcionan por *software* como memcpy, el cual es algo más ineficiente, pero estos pueden venirnos bien para según qué situaciones. Por ejemplo, si estamos inicializando la partida antes de comenzar el juego, nos puede ser irrelevante usar uno u otro, pero si debemos hacer uso de la copia mientras el jugador está jugando, deberíamos hacer uso de los métodos más eficientes para asegurarnos de que no se producen tirones en el juego.

7.5 Sonido

En lo referente al sonido, como hemos comentado, la NDS posee **dos altavoces** desde los cuales se pueden reproducir la música en **estéreo**, además de una **conexión de audio analógica**.

Como ya hemos comentado, todo lo referente a sonido está controlado por el ARM7, así que debemos de darle órdenes al ARM9 y éste es quien se encarga de enviarle instrucciones al ARM7. Es por ello que para poder reproducir sonidos necesitamos que éstos puedan estar bien sincronizados.

Posee **16 canales** a través de los cuales podemos enviar datos para que se reproduzcan. Para un mismo canal, debemos especificar la **frecuencia de la muestra (hasta 32768Hz)**, si supera este valor se asigna al máximo), el tamaño de la muestra, y otros datos como el volumen, si queremos que se reproduzca en bucle, etc.

En cuanto a los canales, los cuales son numerados del 0 al 15, por todos ellos podemos enviar sonidos grabados. Sin embargo, los canales 8 al 13 son los únicos que pueden reproducir **sonidos sintetizados**, y los canales 14 y 15 son los que pueden reproducir **ruido**.

Para poder pasarle los datos de la muestra, se debe hacer en formato **raw o bruto**. Esto no es así para la gran mayoría de formatos de sonido que conocemos a día de hoy (MP3, WAV, etc), así que debemos de buscar una manera de realizar esta conversión, y eso es una tarea que fácilmente puede llevarnos a consumir bastante CPU.

7.6 ROM

Como vamos a hacer un juego que funcionará en una consola real, debemos conocer el formato de **cartuchos** los cuales introducirán nuestro juego en la consola para que pueda ser ejecutado.

Toda la información de un juego para NDS y todas las imágenes y sonidos de éste, al

final son compiladas y enlazadas dando como resultado un **archivo binario** de un formato específico. Este formato es .nds, un archivo binario que además incorpora un encabezado con información adicional del juego (un nombre, logo, y una descripción corta) y opcionalmente algunos archivos adjuntos como, por ejemplo, un sistema de ficheros.

Este archivo se coloca en unos cartuchos **ROM**⁸ y se distribuyen las copias a las personas que compran los juegos oficiales.

Estos cartuchos son de unas dimensiones de 35mm de alto, 33mm de ancho, 3.8 mm de profundidad y tienen un peso aproximado de 3.5 g. Se distribuyen con unas pequeñas variaciones según el juego lo requiera y el fabricante así lo haya especificado, por ejemplo, los tamaños de esta memoria ROM van de **8Mb a 512Mb**. Además, a veces incluyen una pequeña **memoria flash** para guardar datos relacionados a la partida (si es el caso de que el jugador pueda guardar su progreso). También, si el juego hace uso de señales infrarrojas, el cartucho **incorpora el sensor** para ello, ya que la consola no tiene *hardware* específico para ello.



Figura 7.3: Interior de un cartucho de juego para NDS. El superior (NTR-031) añade el sensor infrarrojo.

Fuente: Thomas Farrell y Connor Petilli

⁸ROM: Del inglés *Read-Only Memory*, hace referencia a un tipo de memoria que sólo se usa para leer de ella datos.