

**UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERA TÉCNICA EN INFORMÁTICA DE GESTIÓN**

Un Juego de lucha para Nintendo DS: implementación y desarrollo de tutorial.

Realizado por
Antonio Jesús Narváez Corrales

Dirigido por
Pablo López Olivas
y
Antonio José Fernández Leiva

Departamento
Lenguajes y Ciencias de la Computación

MÁLAGA, Septiembre, 2015

**UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN**

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente/a Dº/Dª.

Secretario/a Dº/Dª.

Vocal Dº/Dª.

para juzgar el proyecto Fin de Carrera titulado: **Un juego de lucha para Nintendo DS: implementación y desarrollo de tutorial.**

realizado por: **Dº Antonio Jesús Narváez Corrales.**

tutorizado por: **Dº Pablo Lopez Olivas y Dº Antonio José Fernández Leiva**

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN
DE _____ Y PARA QUE CONSTE, SE
EXTIENDE FIRMADA POR LOS COMPARCIENTES DEL TRIBUNAL, LA
PRESENTE DILIGENCIA.

Málaga a _____ de_____ del 2015

El/La Presidente/a: El/La Secretario/a: El/La Vocal:

Fdo:

Fdo:

Fdo:

Índice de ilustraciones

Figura 2.01: Logo Nintendo 1950.	20
Figura 2.02: Consola Game & Watch.	21
Figura 2.03: Juego Donkey Kong.	22
Figura 2.04: Consola Super Nintendo.	23
Figura 2.05: Consola Wii.	24
Figura 2.06: Amiibo de Mario.	25
Figura 2.07: Nintendo DS original.	26
Figura 2.08: Nintendo DS Lite.	27
Figura 2.09: Nintendo DSi.	29
Figura 2.10: Nintendo DSi XL.	29
Figura 2.11: Nintendo 3DS.	31
Figura 2.12: Nintendo 2DS.	32
Figura 2.13: New Nintendo 3DS.	33
Figura 2.14: Juego Warrior.	37
Figura 2.15: Juego Karate Champ.	38
Figura 2.16: Juego Urban Champion.	39
Figura 2.17: Juego Typhoon Gal.	40
Figura 2.18: Juego Street Fighter 1.	41
Figura 2.19: Juego Pit Fighter.	41
Figura 2.20: Juego Street Fighter 2.	42
Figura 2.21: Juego Mortal Kombat.	43
Figura 2.22: Juego Virtual Fighter.	44
Figura 2.23: Juego The King of Fighter.	45
Figura 2.24: Juego Tekken 1.	46
Figura 2.25: Juego Soul Edge.	47
Figura 2.26: Juego Bloody Roar.	48

Figura 2.27: Juego Rivals Schools United by Fate.....	48
Figura 2.28: Juego Street Fighter 3.....	49
Figura 2.29: Juego Guilty Gear.....	50
Figura 2.30: Juego Super Smash Bros.....	51
Figura 2.31: Juego Dragon Ball Z: Budokai Tenkaichi.....	52
Figura 2.32: Juego Street Figther 4.....	53
Figura 2.33: Juego Pokken Tournament.	55
Figura 2.34: Juego Touhou 11.....	56
Figura 2.35: Juego Touhou 10.5.....	57
Figura 2.36: Juego Touhou 12.3.....	58
Figura 3.01: Programa Visual Studio.	60
Figura 4.01: Ventana de instalación de DevKitPro	63
Figura 4.02: Ejemplo de fuente.	66
Figura 4.03: Fuente personalizada.....	68
Figura 4.04: Ejemplo imagen de animación.	75
Figura 4.05: Mapa de colisiones.	79
Figura 4.06: Pruebas de colisiones	82
Figura 5.01: Ambientación Touhou DS	91
Figura 5.02: Boceto de interfaz de Touhou DS	92
Figura 5.03: Ejemplo imagen de animación.	95
Figura 6.01: Clase Math en tutorial 1	101
Figura 6.02: Diagrama de Clases Tutorial 1. Menu Inicial.....	102
Figura 6.03: Version 0.1. Menu Inicial	103
Figura 6.04: Clase Sprite en tutorial 1.	104
Figura 6.05: Clase Palette en tutorial 1.....	105
Figura 6.06: Clase Background en tutorial 1.....	106
Figura 6.07: Clase Input en tutorial 1..	107

Figura 6.08: Clase Text en tutorial 1.....	108
Figura 6.09: Clase Engine en tutorial 1.....	109
Figura 6.10: Clase Menú en tutorial 1.....	110
Figura 6.11: Clase Button en tutorial 1.....	111
Figura 6.12: Diagrama de Clases Tutorial 2. Escena Combate.....	112
Figura 6.13: Version 0.2. Escena combate.....	112
Figura 6.14. Clase Sprite en tutorial 2.	113
Figura 6.15: Clase SpriteAnimated en tutorial 2.....	114
Figura 6.16: Clase Input en tutorial 2.	114
Figura 6.17: Clase Movement en tutorial 2.	115
Figura 6.18: Clase Character en tutorial 2.	116
Figura 6.19: Clase Battle en tutorial 2.	117
Figura 6.20: Diagrama de Clases Tutorial 3. Movimientos.	118
Figura 6.21: Version 0.3. Escena Combate	118
Figura 6.22: Clase XMLParser en tutorial 3.....	119
Figura 6.23: Clase Movement en tutorial 3.....	121
Figura 6.24: Clase Input en tutorial 3	122
Figura 6.25: Clase Character en tutorial 3	123
Figura 6.26: Diagrama de Clases Tutorial 4. Movimientos Proyectiles.....	125
Figura 6.27: Version 0.4. Escena combate	125
Figura 6.28: Clase Movement en tutorial 4.....	126
Figura 6.29: Clase ProjectileMovement en tutorial 4.....	127
Figura 6.30: Clase Projectile en tutorial 4.	129
Figura 6.31: Clase Character en tutorial 4	130
Figura 6.32: Diagrama de Clases Tutorial 5. Colisiones.	132
Figura 6.33: Version 0.5. Menu Inicial	133
Figura 6.34: Version 0.5. Escena Combate	133

Figura 6.35: Clase Math en tutorial 5	133
Figura 6.36: Clase Character. en tutorial 5.....	135
Figura 6.37: Clase Movement en tutorial 5.....	136
Figura 6.38: Clase Collision en tutorial 5.	137
Figura 6.39: Clase Battle en tutorial 5.	138
Figura 6.40: Diagrama de Clases Tutorial 6 Menus y modos de juego.....	138
Figura 6.41: Version 0.6. Menu Inicial	139
Figura 6.42: Version 0.6. Selección de personaje	139
Figura 6.43: Version 0.6. Escena combate.....	140
Figura 6.44: Clase Scene en tutorial 6.	140
Figura 6.45: Clase Options en tutorial 6.	141
Figura 6.46: Clase CharacterSelector en tutorial 6.....	141
Figura 6.47: Clase Battle en tutorial 6.	142
Figura 6.48: Clase Menu en tutorial 6.	143
Figura 6.49: Clase Engine en tutorial 6.	143

Índice general

Agradecimientos	13
1. Introducción.....	15
1.1. Contexto.....	15
1.2. Objetivo.....	16
1.3. Estructura de la memoria.	16
2. Nintendo DS, Nintendo 3DS y Juegos de Lucha.....	19
2.1. Nintendo	19
2.2. Historia de Nintendo.....	19
2.3. Nintendo DS.	25
2.4. Nintendo 3DS.	30
2.5. Homebrew y FlashCart.....	34
2.6. Juegos de Lucha	35
2.6.1. Historia Juegos de Lucha.....	36
2.7. Touhou	56
2.7.1. Touhou: Scarlet Weather Rhapsody 10.5.....	57
2.7.2. Touhou: Hisoutensoku 12.3.	58
3. Tecnología empleada.	59
3.1. Devkitpro.....	59
3.2. Librería Libnds.....	59
3.3. Librería Palib.	59
3.4. Librería NightFox lib's.....	59
3.5. Visual Studio.	60
3.6. Lenguaje C.....	60
3.7. Lenguaje C++.....	61
3.8. XML.	61
4. Tutoriales.	63

4.1. Instalación de entorno.	63
4.2. Herramientas a usar (emuladores, grit).....	64
4.3. Hello world.	64
4.4. Texto por pantalla.	65
4.4.1 Colores de fuentes.	67
4.4.2 Fuentes personalizadas.	67
4.5. Imágenes.....	68
4.5.1. Fondos (Backgrounds).....	69
4.5.1.1. Simples (Tiled y 256 colores).	69
4.5.1.2. Complejos (8 bits y 16 bits).	70
4.5.1.3. Affine.....	71
4.5.2. Imágenes (Sprites).	72
4.5.2.1. Simples.....	73
4.5.2.2. Animados.....	74
4.5.2.3. Sprites 3D.....	76
4.6. Input.....	77
4.7. Sonidos.....	78
4.8. Colisiones.....	78
4.8.1. Mapa colisiones.....	79
4.8.2. Colisiones Manuales.....	81
4.9. Wifi.....	82
5. Diseño	85
5.1. Diseño.	85
5.1.1. Patrón Documento Concepto.	87
5.1.2. Patrón Documento Diseño	88
5.1.3. Documento Concepto Touhou DS.....	90
5.2. Diseño TouhouDS.....	91

5.2.1. Ambientación.....	91
5.2.2. Combate.....	91
5.2.3. Personajes.....	93
5.2.3.1. Movimientos.....	93
5.2.4. Modos de juego.....	95
6. Implementación.....	97
6.00. Metodología.....	97
6.00.1. Metodologías ágiles	97
6.00.2. Extreme Programming	98
6.00.3. Metodología.....	99
6.01 Introducción	101
6.01.1 Conocimientos previos.....	101
6.01.2 Clase Math.	101
6.02. Tutorial 1. Menú inicial.	102
6.02.1. Clase Sprite.....	103
6.02.2. Clase Palette.....	105
6.02.3. Clase Background.....	106
6.02.4. Clase Input.....	107
6.02.5. Clase Text.....	108
6.02.6. Clase Engine.	109
6.02.7. Clase Menú.....	110
6.02.8. Clase Button.	111
6.03. Tutorial 2. Escena Combate.	111
6.03.1. Clase Sprite.....	113
6.03.2. Clase SpriteAnimated.....	113
6.03.3. Clase Input.....	114
6.03.4. Clase Movement.	114

6.03.5. Clase Character.....	116
6.03.6. Clase Battle.....	117
6.04. Tutorial 3. Movimientos.....	118
6.04.1. Clase XMLParser	119
6.04.2. Clase Movement	121
6.04.3. Clase Input.....	122
6.04.4. Clase Character.....	123
6.05. Tutorial 4. Movimientos Proyectiles.....	123
6.05.1. Clase Movement.	124
6.05.2 Clase ProjectileMovement.....	126
6.05.3 Clase Projectile.....	127
6.05.4. Clase Character.....	129
6.06. Tutorial 5. Colisiones.....	131
6.06.1. Clase Math	133
6.06.2. Clase Character.....	135
6.06.3. Clase Movement.	136
6.06.4. Clase Collision.	137
6.06.5. Clase Battle.....	137
6.07. Tutorial 6. Menus y modos de juego	138
6.07.1. Clase Scene.....	140
6.07.2. Clase Options.....	140
6.07.3. Clase CharacterSelector.....	141
6.07.4. Clase Battle.....	142
6.07.5. Clase Menu.....	142
6.07.6. Clase Engine	143
Conclusiones.....	145
Bibliografía.....	147

Agradecimientos

Se que no podría haber llegado hasta aquí sin la ayuda de mi familia. Por lo tanto este proyecto solo se lo puedo dedicar a ellos: A mi padre, a mi madre y a mis hermanos.

Capítulo 1. Introducción

En este capítulo se detallará el plan a seguir para desarrollar esta memoria, así como el objetivo de ésta.

1.1. Contexto

En la actualidad, los videojuegos son parte importante de la cultura y son una de las formas de ocio preferidas de los jóvenes y no tan jóvenes. Los videojuegos han acompañado a la informática prácticamente desde su nacimiento convirtiéndose en una muestra de los avances en hardware y software que se iban realizando en el sector. Al inicio se convirtieron en una de las maneras de entretenérse para aquellos que estaban relacionados con la informática. Sin embargo, una vez que se empieza a ver la popularidad que alcanzan los videojuegos, comienzan a formarse grandes empresas que darán lugar a la llamada industria de los videojuegos. Hoy día es una industria consolidada que mueve mucho dinero y está en continuo aumento.

El desarrollo de videojuegos es la actividad por la cual se diseña y crea un videojuego, desde el concepto inicial hasta la versión final. Ésta es una actividad multidisciplinaria, que involucra profesionales de la informática, el diseño, el sonido, la actuación, etc. El proceso es similar a la creación de software en general, aunque difiere en la gran cantidad de aportes creativos (música, historia, diseño de personajes, niveles, etc.) necesarios. El desarrollo también varía en función de la plataforma objetivo (PC, móviles, consolas), el género (estrategia en tiempo real, rol, aventura gráfica, plataformas, etc.) y la forma de visualización (2D, 2.5D y 3D). Los estudios en informática sobre: inteligencia artificial, redes neuronales, modelado de procesos físicos, etc. tienen aplicación directa en el desarrollo de videojuegos.

Actualmente la mayoría de las grandes empresas desarrollan sus propias herramientas de desarrollo y además, las que poseen alguna videoconsola en el mercado disponen de sus propias herramientas de desarrollo con el hardware y software necesario para probar las aplicaciones. Sin embargo, existen alternativas a las herramientas (SDK, Software Development Kit) propietarias de las compañías, que surgen gracias al aporte de personas que programan y distribuyen librerías abiertas que permiten el desarrollo para videoconsolas.

Nintendo DS es la consola portátil más vendida de la historia y por poco no es la consola, contando portátil y de sobremesa, más vendida de la historia, según las cifras que maneja el popular portal "VGChartz" [Vgchartz, 2015].

La plataforma Nintendo DS lleva en el mercado 10 años aproximadamente y aunque hay numerosos tutoriales para adquirir los conocimientos necesarios para desarrollar aplicaciones, ninguno es definitivo, ya que siempre generan dudas acerca de ciertos aspectos sobre su desarrollo.

El género de lucha es un género amado por todos los jugadores. Éste tuvo su momento de gloria en los años noventa con juegos como “Street Fighter 2”, “Marvel Super Heroes” o “Mortal kombat”; pero aun así hoy en día goza de cierta popularidad entre el público en general con sagas como “Tekken”, “Soul calibur” o “Mortal Kombat” entre otras.

1.2. Objetivo

El fin de este proyecto es llevar a cabo el desarrollo de un juego de lucha con un desarrollo incremental y desarrollar un tutorial para la programación para Nintendo DS.

El lenguaje de programación usado en la Nintendo DS es C, C++ o Lua. En este caso nos decantamos por el segundo debido a que su manejo de clases facilitará la comprensión y estructuración del código.

El lenguaje C es el más usado por todos los desarrolladores amateurs. Para programar en Nintendo DS es necesario una librería específica. Existe dos librerías que destacan por encima de todas las demás por su fácil manejo e instalación, éstas son “Palib” y “Libnds”.

Tiempo atrás “Palib” [PALib, 2015] era la más usada porque era muy sencillo llevar a cabo aplicaciones, es decir, con poco código se obtenían bastantes resultados. Poco a poco ha perdido fuerza debido a sus limitaciones de bajo nivel, en ocasiones necesarios para el correcto desarrollo en la consola. Aun así la librería “Libnds” [Libnds, 2015] es bastante compleja de manejar dificultando el trabajo a los programadores.

En realidad hay una tercera alternativa en la programación para la Nintendo DS, es NFlib [NightFox Lib, 2015]. Ésta hace uso directo de Libnds facilitando las funciones de ésta adaptándolas para así evitar el difícil manejo de la librería original. Debido a su facilidad de manejo y potencia, la librería NFlib es la elegida para nuestro desarrollo.

El juego de lucha que realizaremos está basado en un juego de la saga Touhou [Touhou Wiki, 2015], más concretamente estará basado en el Touhou 12.3. Este juego es un juego de lucha clásico sin ningún aditivo especial. En este juego luchan dos luchadores, uno frente a otro en un entorno 2D y estos personajes tienen ataques de corto alcance y lanzamiento de ciertas magias.

En definitiva, realizaremos una serie de tutoriales, cuyos ejemplos estarán escritos en C, para enseñar cómo se programa en Nintendo DS con la librería NFlib y para culminar realizaremos un juego de lucha pero esta vez desarrollándolo en C++ y NFlib.

1.3. Estructura de la memoria

A continuación se detalla cómo se ha organizado la presente memoria:

Capítulo 1: Introducción. En el capítulo introductorio explicamos el principal objetivo del proyecto, así como la organización de este documento.

Capítulo 2: Nintendo DS, Nintendo 3DS y juegos de lucha. En este capítulo se explican los detalles técnicos que pueden ser de utilidad sobre la videoconsola en la que se va a desarrollar los tutoriales y la aplicación objetivo de este proyecto. También se habla un poco sobre la empresa creadora de dicha consola y sus antecedentes, así como detalles sobre los juegos de lucha y acerca del juego que ha servido de inspiración para desarrollar el nuestro.

Capítulo 3: tecnología empleada. En el capítulo 3 describimos el software a utilizar para el desarrollo del proyecto, detallando información sobre algunas librerías y lenguajes de programación.

Capítulo 4: Tutoriales. En este capítulo nos centraremos en el desarrollo de los tutoriales con la librería Night Fox. Se abarcará todos los puntos posibles, desde la carga de un simple texto en pantalla, hasta la carga de complejas imágenes.

Capítulo 5: Diseño. Aquí se explica brevemente cómo realizar un documento de diseño y realizaremos el documento del juego que se desarrollará en el siguiente capítulo.

Capítulo 6: Implementación. El capítulo 6 se centra en la implementación de la aplicación, basándose en los capítulos anteriores. Antes de entrar de lleno en la implementación, se hablará sobre el plan de trabajo usando la metodología "Extreme Programming". A continuación se detalla la implementación del menú inicial, personajes, movimientos, colisiones, comunicación inalámbrica... Llevaremos un desarrollo incremental, por lo que para el tutorial 2 se partirá del tutorial uno y así sucesivamente.

Conclusiones. Éste capítulo se muestran las conclusiones del proyecto, se detallan los conocimientos adquiridos y sus posibles ampliaciones y mejoras que podrían implementarse en versiones futuras.

Bibliografía. Por último, se recopila la bibliografía referenciada a lo largo de este documento así como bibliografía adicional relacionada con el proyecto.

Capítulo 2. Nintendo DS, Nintendo 3DS y Juegos de Lucha

Este capítulo recopila la historia de la compañía Nintendo [Historia Nintendo, 2015], así como su plataforma Nintendo DS [Xataka Nintendo DS, 2015] y sucesivas. También se profundizará en qué significa un juego de lucha y su historia. Y para terminar se explicará el título en el que nos hemos basado, Touhou [Touhou Wiki, 2015], para el juego desarrollado en el capítulo 6.

2.1. Nintendo

"Nintendo Company Limited" es una empresa dedicada al sector de los videojuegos, tanto en la creación de distintas consolas como en el desarrollo de juegos.

Nintendo es una de las empresas más grandes del mundo en cuanto a investigación y desarrollo de videojuegos se refiere, tratando siempre de innovar en este mercado tan competitivo. Además, el grado de satisfacción del cliente es muy elevado. Desde 1975, cuando diera su primer paso dentro de este sector, se ha dedicado a la producción de "software" (juegos) y "hardware" (consolas), creciendo hasta convertirse en una de las compañías más exitosas en la industria.

Con sede en Kyoto (Japón), actualmente fabrica y distribuye "hardware" y "software" para su última consola de sobremesa "Wii U" y para la familia de consolas portátiles "Nintendo 3DS". Desde 1983, cuando lanzó al mercado la consola "Nintendo Entertainment System" (NES), Nintendo ha vendido más de 4.200 millones de videojuegos y más de 669 millones de consolas en todo el mundo. En su haber tiene un listado de consolas que incluye las actuales consolas Wii U y Nintendo 3DS, así como Wii, Nintendo DS, Nintendo DSi y Nintendo DSi XL, Game Boy, Game Boy Advance, Super NES, Nintendo 64 y Nintendo GameCube. Entre sus juegos han creado personajes tan conocidos para la cultura como Mario, Donkey Kong, Metroid, Zelda y Pokémon.

En España, "Nintendo Ibérica, S.A.", es la subsidiaria encargada del mercado nacional de la compañía Nintendo.

2.2. Historia de Nintendo

La siguiente información ha sido obtenida de la web oficial de Nintendo [Historia Nintendo, 2015] y de diversos artículos de Wikipedia [Wikipedia, 2015]

Año 1889, Nintendo dio sus primeros pasos como una empresa japonesa creada por "Fusajiro Yamauchi" bajo el nombre de "Nintendo Koppai". Estableciendo su sede en Kyoto, la empresa producía y distribuía "Hanafuda", un juego de cartas. Estas cartas eran hechas a mano y pronto ganaron popularidad. Yamauchi tuvo que contratar asistentes para poder producir mayor cantidad de cartas y así satisfacer la demanda.

En el año 1902, el señor Yamauchi comienza a fabricar las primeras barajas de cartas estilo occidental en Japón. En principio estaban destinadas a la exportación, pero el producto alcanza tanta popularidad en Japón como en el resto del mundo.

En el año 1933 se funda la sociedad Yamauchi Nintendo Co. Ltd.

Llegando el 1950 Hiroshi Yamauchi, bisnieto del creador de la compañía, es nombrado presidente y se hace cargo de la manufacturación de Yamauchi Nintendo Co. Ltd. Poco tiempo después la compañía pasa a llamarse Nintendo Playing Card Co. Ltd. Es la primera compañía en tener éxito con la producción masiva de cartas de plástico en Japón.



Figura 2.01: Logo Nintendo 1950

En 1955 Hiroshi Yamauchi visita EE. UU. para mantener negociaciones con la "United States Playing Card Company", empresa dominante en el mercado de fabricantes de cartas estadounidenses en 1995 pero que operaba en una pequeña oficina. Esto fue un golpe de realidad, donde Yamauchi notó las limitaciones del negocio de las cartas. Finalmente logró hacer un trato con Walt Disney y pudo utilizar los personajes de esa empresa en sus cartas logrando incrementar las ventas de cartas abriéndose al mercado infantil.

Llegando 1963, Yamauchi renombró la empresa a "Nintendo Company Limited". También comenzó a experimentar en otras áreas de negocio. Destacan una compañía de taxis, una cadena de "Hoteles del amor", una cadena de TV, una compañía de comida, etc. Todos estos intentos fueron un fracaso excepto uno, los juguetes.

En los Juegos Olímpicos de Tokio 1964 las ventas de cartas cayeron, haciendo que las ganancias de Nintendo fueran menores.

Debido a la caída en ventas de cartas, en 1969 se amplía el departamento de juegos y se construye una planta de producción a las afueras de Kyoto.

Un día cualquiera de 1970, Hiroshi Yamauchi se encontraba en la fábrica de Nintendo. Allí observó un brazo extensible que había sido creado por Gunpei Yokoi, uno de sus ingenieros de mantenimiento, para su diversión. Yamauchi le ordenó a Yokoi que desarrollara su producto para

poder lanzarlo en la campaña de Navidad. La "Ultra Hand" fue un enorme éxito vendiendo aproximadamente 1,2 millones de unidades. Yokoi fue ascendido al área de desarrollo de productos. Se inicia la venta de la serie Beam Gun que utilizaba opto-electrónica. Se incorporaba, por primera vez en Japón, la electrónica a la industria del juguete.

En cooperación con Mitsubishi Electric, en 1975 se desarrolla para Japón un sistema de videojuegos utilizando reproductor de vídeo electrónico (EVR). Nintendo se dio cuenta del éxito que tenían los videojuegos en esta época, y comenzó a introducirse en el mercado. Su primer paso en este campo fue el de asegurarse su derecho a distribuir la Magnavox Odyssey, una consola, en Japón.

En 1977 se introduce al desarrollo de videojuegos para uso doméstico en cooperación con Mitsubishi Electric. Desarrolla sus primeras consolas domésticas "TV Game 15" y "TV Game 6". También en este año Shigeru Miyamoto fue contratado como artista gráfico para pintar máquinas recreativas.

En marzo de 1978, Nintendo publica un juego de sobremesa arcade basado en el "Othello". En una pantalla negra, de un solo color, piezas de Othello verdes, blancas y negras se sustituyen por cuadrados y signos de suma respectivamente.

En el año 1979, Minoru Arakawa, yerno del director de Nintendo Hiroshi Yamauchi, decide probar suerte en el mercado americano e inaugura "Nintendo of America" en la ciudad de Nueva York. Nintendo inicia un departamento de máquinas de juegos operadas con monedas.

Pasado un año de la inclusión americana, en 1980 ya consigue establecerse definitivamente bajo el nombre Nintendo of America Inc. Se comienza a vender en Japón la línea de productos "GAME & WATCH", los primeros videojuegos portátiles LCD con microprocesador.



Figura 2.02: Consola "Game & Watch"

El artista gráfico de Nintendo, Shigeru Miyamoto, crea el juego Donkey Kong. A Jumpman, protagonista del juego, se le cambia posteriormente el nombre en las oficinas centrales de Nintendo of America como homenaje por su parecido con el propietario de las oficinas, Mario

Segali. Por lo que se pasó a llamar "Mario".

En 1981, Nintendo se enfoca en el desarrollo y distribución por todo el mundo del videojuego de funcionamiento con moneda "Donkey Kong". Este videojuego se convierte rápidamente en el de mayor venta entre los juegos para salas recreativas.

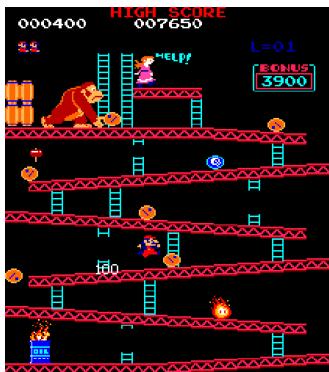


Figura 2.03: Juego "Donkey Kong"

En Julio de 1983, Nintendo entra a la Bolsa de Tokio. También se comienza a trabajar en la consola denominada "Family Computer" (Famicom), utilizando una CPU (Central Processing Unit) y PPU (Picture Processing Unit).

Tan solo un año después del comienzo de su desarrollo, en 1984 se lanza el sistema Famicom en Japón. Llamado Nintendo Entertainment System, NES, cuando se lanza en el resto del mundo. Entre los títulos disponibles están Excitebike, Super Mario Bros., Metroid, The Legend of Zelda y Punch-Out! Mario y su hermano Luigi se hacen tan famosos como NES y el juego para NES "Super Mario Bros." es un éxito en todo el mundo.

Ya en 1989, se lanza al mercado la Game Boy en Japón, la primera consola portátil con cartuchos de juego intercambiables. En los siguientes años Game Boy es lanzada junto a Tetris y en todo el mundo logran un enorme éxito.

Tan solo un año después, 1990, Nintendo se introduce en el mercado de 16 bits lanzando Super Famicom. También este año se instalan las oficinas centrales de Nintendo of Europe como filial en Alemania.

En 1991 Super Nintendo Entertainment System, conocida como Super Famicom en Japón, aparece en Europa con gran éxito.



Figura 2.04: Consola "Super Nintendo"

En 1993 se establecen mas filiales de Nintendo en Francia, Reino Unido, España, Bélgica y Australia.

En 1996 Nintendo da el salto al 3D. En junio de este mismo año se lanza en Japón la Nintendo 64, primera videoconsola de 64 bits. Sorprende por su calidad gráfica y por su extraño mando.

Después del éxito monocromo de Game Boy, en 1998 Nintendo lanza al mercado Game Boy Color.

En el año 2000 Nintendo se traslada de edificio en Kyoto y se establece en Reino Unido con Nintendo UK

En 2001 Nintendo lanza Game Boy Advance impresionando con la potencia gráfica que ofrecía la portátil. Este mismo año Nintendo lanza otra consola de sobremesa, Nintendo GameCube.

En 2002 Nintendo crea otra filial más, esta vez en Italia. También este año Nintendo, Sega y Namco anuncian el desarrollo conjunto del tablero gráfico informático en 3D "TRIFORCE", para las máquinas recreativas de la próxima generación.

A mediados de este mismo año y tras 52 años al frente de Nintendo Co., Ltd, el Presidente Hiroshi Yamauchi anuncia su retirada y nombra sucesor a Satoru Iwata.

Aunque tan solo habían pasado dos años desde que Game Boy saliese a la venta, en 2003 Nintendo lanza una revisión de la consola, Game Boy Advance SP. El secreto de su éxito fue su pantalla abatible y su retroiluminación.

En el año 2005 se lanza Nintendo DS y alcanza un éxito inmediato. Esta consola se caracteriza por su doble pantalla, una de ellas táctil. Terminado este año Nintendo lanza Game Boy Micro, reedición reducida de la Game Boy Advance.

En 2006 aparece una reedición de la Nintendo DS, la Nintendo DS Lite. El año culmina con el lanzamiento de Wii. La nueva consola de sobremesa con su innovador sistema de control que

tiene muy buena acogida tanto entre los fans de Nintendo como entre un público poco acostumbrado a los videojuegos.



Figura 2.05: Consola "Wii"

En los siguientes años se crean mas filiales de Nintendo, esta vez en Sudáfrica y en Lisboa.

En 2009 la consola portátil Nintendo DS, sufre otro rediseño quedando como resultado la Nintendo DSi. La portátil incluía nuevas funciones de sonido, dos cámaras y acceso al contenido digital que poseía Wii. La Nintendo DSi es lanzada con un hermano mayor, Nintendo DSi XL cuyas características son similares pero es de mayor tamaño.

Dos años después de la ultima reedición de Nintendo DS, en 2011 Nintendo lanza Nintendo 3DS. Esta consola sigue el mismo patrón de dos pantallas donde una de ellas es táctil. La característica principal de esta consola es la inclusión de una pantalla 3D sin gafas, aunque no termina de funcionar bien del todo.

En el año 2012 Nintendo lanza una nueva consola doméstica, Wii U, y una nueva consola portátil, Nintendo 3DS XL. La portátil es semejante a su anterior versión pero de mayor tamaño, como ya ocurriera con Nintendo DSi. En cambio la Wii U es mucho mas q una simple rediseño. Esta consola trae como principal característica la inclusión de un mando con pantalla táctil como se de la pantalla de la Nintendo DS se tratase. Esto otorgó una jugabilidad nunca antes vista.

Tan solo un año después de Nintendo 3DS XL, Nintendo lanza Nintendo 2DS. Esta portátil que permite jugar a todos los juegos de Nintendo 3DS pero en 2D y a un precio mas económico.

A finales de 2014 Nintendo lanza al mercado las figuras Amiibo con un tremendo inesperado éxito. Figuras que mediante la tecnología NFC permite interactuar con la consola Wii U.



Figura 2.06: "Amiibo" de Mario

A principios de 2015 Nintendo lanza al mercado una reedición de la Nintendo 3DS, llamándose New Nintendo 3DS. Las principales diferencias con su predecesora son la inclusión de una pantalla 3D mas real, mas botones y la inclusión de tecnología NFC para la interacción con los Amiibo.

2.3. Nintendo DS

El 12 de noviembre de 2003 Nintendo anunció su desarrollo y su lanzamiento se produjo en 2004. Esta fue anunciada bajo el nombre clave "Nintendo DS" (DS significa Dual Screen, doble pantalla). El nombre de esta consola suele abreviarse como "NDS" o simplemente como "DS".

La larga experiencia de Nintendo en el campo de las consolas creó una consola exclusiva, no sólo a nivel físico, sino también en cuanto a software se refiere. Técnicamente la consola incluyó algunas características innovadoras para la época, como la propia pantalla táctil controlable mediante un stylus, el micrófono incorporado o la conectividad WiFi.

La evolución de Nintendo ha sido crear la primera consola con pantalla táctil que permite interactuar con la máquina como nunca se había hecho en una consola, acercándola más a las PDAs que al resto de las consolas. La pantalla inferior de la Nintendo DS está superpuesta con una pantalla resistiva lo cual la hace táctil. La otra pantalla es un LCD estándar sin función táctil. El uso de dos pantallas produce mayor complejidad en los programas de software al tener que especializar las funciones de cada pantalla.

En un extremo está el pad direccional, desarrollado por Nintendo, que permite el

desplazamiento por los menús y el control de algunos juegos. En el lado contrario se sitúan cuatro botones que se corresponden con el estándar marcado por los modelos de sobremesa como Super Nintendo, que se denominan en la Nintendo DS: X, Y, B y A, incluyendo también los botones SELECT y START. Sobre la consola, y cubierto tras la pantalla al levantarla se sitúan dos botones más, los botones L y R.



Figura 2.07: "Nintendo DS" original

Especificaciones técnicas

Dos pantallas retroiluminadas de 3 pulgadas, con una resolución máxima de 256x192 píxeles cada una, con hasta 260000 colores (5 bits para cada canal). La pantalla inferior es táctil y reconoce varias pulsaciones simultáneas, devolviendo un único resultado, el baricentro.

- Dos procesadores que pueden ejecutar código simultáneamente:
 - ARM9: ARM946E-S, la CPU principal, 67 MHz.
 - ARM7: ARM7TDMI, coprocesador, 33 MHz.
- 4 megabytes de RAM.
- 656 KB de RAM para vídeo y una pequeña memoria no volátil para las preferencias del usuario.
- Una GPU compuesta de dos sistemas de renderizado 2D (uno para cada pantalla) y un sistema de renderizado 3D que puede renderizar hasta 120.000 triángulos por segundo a una tasa de 60 fps y una tasa de relleno de 30 millones de píxeles por segundo. La GPU está integrada en el mismo chip que ambos procesadores.
- Un pad de dirección y 8 botones (A, B, X, Y en forma de cruz; L, R en los laterales; y Start y Select, a cada lado de la pantalla).
- NiFi (Nintendo wifi).

- 802.11b wifi.
- Una salida de audio de 16 canales con altavoces estéreo y una salida estándar para auriculares.
- Un micrófono.
- Dos aberturas para cartuchos flash, el "Slot-1" para cartuchos específicos de DS y el "Slot-2" compatible con cartuchos de GameBoy Advance.
- Batería de ión-litio de 850 mAh.
- Reloj integrado de 33 MHz, que se encarga de mantener la hora y fecha incluso cuando la NDS está apagada.
- Peso: 275 gramos.
- Dimensiones, 148,7 mm de ancho, 84,7 mm de largo y 28,9 mm de alto

Nintendo DS Lite

Ya se había creado la marca Nintendo DS y era una consola bien tratada, económica (en España se puso a la venta por 150 euros) y con un generoso catálogo de juegos. Poco más de un año tras su llegada al mercado la compañía japonesa hizo oficial la Nintendo DS Lite, un modelo similar a la original aunque con pequeños cambios.



Figura 2.08: "Nintendo Ds Lite"

Las pantallas, por ejemplo, eran algo más nítidas, aunque con la misma resolución, 256x192 píxeles. Mantuvo la retrocompatibilidad con Game Boy Advance, mejoró la batería y redujo el peso, desde los 275 gramos de la original a los 218 gramos de la DS Lite. Nintendo mantuvo su precio, en España 150 euros, y fue una más que digna sucesora.

La Nintendo DS Lite es un 21% más ligera que el modelo original. La posición de los botones ha sido cambiada ligeramente, como el botón ON/OFF, que ha pasado de estar encima de la

cruceta de control y a ser un botón en el lado derecho. Los botones SELECT y START, que estaban situados encima de los cuatro botones, se encuentran ahora debajo de los mismos. Las luces de encendido y de carga han sido cambiadas a la parte superior derecha y el micrófono ahora está en el centro, para que al soplar no haya que mover la vista y desviarse del juego.

El puntero de la consola es ligeramente más grande y ahora se guarda a la derecha de la consola. Las pantallas también han sido mejoradas, aunque siguen del mismo tamaño, ahora cuentan con regulación de la luminosidad, al igual que la Game Boy Micro. Las cubiertas son perfectamente rectangulares, por lo que no se pueden rayar las pantallas ni donde están los botones.

Especificaciones técnicas

- Los únicos cambios con respecto a la original, son:
- Pantallas de 3,12 pulgadas
- 4 niveles de brillo en las pantallas
- Batería de Ión litio de 1000 mAh.
- Peso: 218 gramos.
- Dimensiones, 148,7 mm de ancho, 84,7 largo de mm y 28,9 mm de alto

Nintendo DSi y DSi XL

"Nintendo DSi" llegó tres años después de la DS Lite. Incluyeron dos cámaras, una frontal y otra trasera, y una vez más se incrementó el tamaño de ambas pantallas hasta las 3,25 pulgadas en esta ocasión pero con la misma resolución y añadiendo un slot para tarjetas SD. A su vez, Nintendo eliminó la retrocompatibilidad con Game Boy Advance. Más que la mejora de hardware, Nintendo hizo una gran mejora en el software de la consola. Esta consola incluye un sistema similar a su homónima de sobremesa, Wii, permitiéndole descargar juegos mediante pago.



Figura 2.09: "Nintendo DSi"

Más importante que la DSi fue la "Nintendo DSi XL", en la que Nintendo apostó por unas pantallas de un tamaño bastante más grande, 4,2 pulgadas, pero con el resto de características idénticas respecto a la DSi.



Figura 2.10: "Nintendo DSi XL"

Especificaciones técnicas

- Dos pantallas retroiluminadas con 5 niveles de intensidad lumínica:
 - Pantalla superior de 3.25 pulgadas para DSi.
 - Pantalla Inferior táctil de 4.2 para DSi XL.
- Dos procesadores que pueden ejecutar código simultáneamente:

- ARM9: ARM946E-S, la CPU principal, 133 MHz.
- ARM7: ARM7TDMI, coprocesador, 33 MHz.
- 16 megabytes de RAM.
- 1 Mb de RAM para vídeo.
- Un pad de dirección y 8 botones (A, B, X, Y en forma de cruz; L, R en los laterales; y Start y Select, a cada lado de la pantalla).
- 802.11b/g wifi,
- Una salida de audio de 16 canales con altavoces estéreo y una salida estándar para auriculares.
- Micrófono.
- Camera frontal de 0.3 megapixeles.
- Camera trasera de 0.3 megapixeles.
- Slot-1 para cartuchos específicos de DS.
- Slot 2 para tarjetas SD.
- Batería de ión-litio de 840 mAh DSi y 1050 mAh DSi XL.
- Peso: 214 gramos DSi y 314 gramos DSi XL.
- Dimensiones:
 - 137 mm de ancho, 74,9 largo de mm y 18,9 mm de alto DSi.
 - 161 mm de ancho, 91,4 largo de mm y 21,2 mm de alto DSi XL.

2.4. Nintendo 3DS

A principios de 2011 Nintendo 3DS llegó a todo el mundo, y lo hizo con un sistema 3D sin gafas.

Mantiene el mismo formato de doble pantalla que su predecesora, pero esta vez una 3D, la superior, y una táctil, la inferior. Sigue teniendo compatibilidad con los juegos de las anteriores de DS pero con notables mejoras en lo relativo a las características. Las pantallas seguían siendo de un tamaño de 3,53 pulgadas pero esta vez con una resolución mayor, 400x240 píxeles la de arriba; 320x240 la de abajo, hardware evolucionado y mejor gestión de WiFi. En definitiva, es el mismo formato de consola pero adaptada a los tiempos más modernos.



Figura 2.11: Nintendo 3DS

El 3D de la consola es un poco delicado ya que hay estar en la posición adecuada para que se aprecie en su totalidad. El 3D ha sido muy criticado de la consola, ya que muchos lo consideran innecesario.

Un año después de su salida, repite fórmula hizo con su predecesora sacando al mercado la "Nintendo 3DS XL" manteniendo todas las características pero con unas pantallas de mayor tamaño, 4,88 en lugar de 3,53.

Especificaciones técnicas

- Pantalla superior con tecnología autostereoscopica otorgándole 3D.
- Dos pantallas LCD retroiluminadas:
 - Pantalla superior de 3,53 pulgadas para 3DS y 4.88 pulgadas para 3DS XL y con resolución máxima de 400x240.
 - Pantalla inferior 3,02 pulgadas para 3DS y 4.18 pulgadas para 3DS XL y con resolución máxima de 320x240.
- Dos procesadores:
 - ARM11: dual core 266 Mhz.
 - ARM9: 133 MHz.
- Procesador Gráfico (GPU): Pica 20010 a 133 Mhz.
- 128 Mb de RAM.
- Un pad analógico, Un pad de dirección y 9 botones (A, B, X, Y en forma de cruz; L, R en los laterales; y Start, Select y Home).
- Wifi 2.4 Ghz, soporta 802.11b/g.

- Una salida de audio con altavoces estéreo y una salida estándar para auriculares.
- Micrófono.
- Giroscopio.
- Cámara interna de 0.3 megapixeles.
- Dos cámaras traseras de 0.3 megapixeles para fotos y videos en 3D.
- Slot-1 para cartuchos específicos de DS.
- Slot-2 para tarjetas SD.
- Batería de ión-litio de 1300 mAh 3DS y 1750 mAh 3DS XL.
- Peso: 235 gramos 3DS y 336 gramos 3DS XL.
- Dimensiones:
 - 134 mm de ancho, 74 mm de largo y 22 mm de alto 3DS.
 - 156 mm de ancho, 93 mm de largo y 22 mm de alto 3DS XL.

Nintendo 2DS

Cuando ya parecía que la consola no daba más de sí, Nintendo sorprende con la salida al mercado de "Nintendo 2DS".

Es como si tuviésemos una 3DS pero le quitásemos la bisagra, manteniendo las dos pantallas, de "3,53" pulgadas la superior y "3,02" la pulgadas inferior, eliminando el modo 3D y recolocando los controles para una mayor comodidad dado el nuevo formato.



Figura 2.12: "Nintendo 2DS"

Cambio en especificaciones técnicas respecto a 3DS:

- Peso: 235 gramos 3DS y 336 gramos 3DS XL.
- Un altavoz con sonido mono en lugar de dos estéreos.
- Dimensiones 144 mm ancho, 127 largo mm y 20.3 mm alto 3DS.

'New' Nintendo 3DS y 3DS XL

La última al salir al mercado, ha sido la denominada "New 3DS". Nuevamente repite la fórmula de dos tamaños distintos que tan bien le ha funcionado en el pasado.

En general esta consola es en todos los aspectos superior a su precursora aportando un stick para control auxiliar, dos botones adicionales, mejor CPU, más RAM y NFC para darle compatibilidad a las figuras Amiibo. Aunque todo esto destaque, la principal nueva función es el 3D mejorado. Ya no es necesario estar en una posición concreta para que el 3D sea perceptible creando un efecto real de 3D que sorprende a quien lo prueba.



Figura 2.13: "New Nintendo 3DS"

Especificaciones técnicas

- Pantalla superior con tecnología autostereoscopica para 3D.
- Dos pantallas LCD retroiluminadas:
 - Pantalla superior de 3,88 pulgadas para new 3DS y 4,88 pulgadas para 3DS XL y con una resolución máxima de 400x240.
 - Pantalla inferior 3,33 pulgadas para 3DS y 4,18 pulgadas para 3DS XL y con una resolución máxima de 320x240.

- Dos procesadores:
 - ARM11: Quad core 268 Mhz.
 - ARM9: 133 MHz.
- Procesador Gráfico (GPU): Pica 20010 at 268 Mhz.
- 256 Mb de RAM.
- Un pad analógico, Un pad de dirección, un stick de control y 11 botones (A, B, X, Y en forma de cruz; L, R, LZ y RZ en los laterales; y Start, Select y Home).
- Wifi 2.4 Ghz, soporta 802.11b/g.
- Una salida de audio con altavoces estéreo y una salida estándar para auriculares.
- Micrófono.
- Giroscopio.
- Acelerómetro
- Sensor infrarrojo frontal.
- Cámara interna de 0.3 megapixeles.
- Dos cámaras traseras de 0.3 megapixeles para fotos y vídeos en 3D.
- Slot-1 para cartuchos específicos de 3DS y de DS.
- Slot 2 para tarjetas SD.
- Batería de ión-litio de 1400 mAh 3DS y 1700 mAh 3DS XL.
- Peso: 253 gramos new 3DS y 329 gramos new 3DS XL.
- Dimensiones:
 - 142 mm ancho, 80,6 largo mm y 21,6 mm alto new 3DS.
 - 160 mm ancho, 93,5 largo mm y 21,5 mm alto new 3DS XL.

2.5. Homebrew y FlashCart

Homebrew es el nombre dado a los programas o juegos creados con las herramientas extraoficiales y no están avalados por las empresas propietarias de la plataforma. Cada tutorial que realicemos será considerado Homebrew para Nintendo DS.

Casi en el mismo momento que se publicó la Nintendo DS surgieron herramientas para poder desarrollar para ella. Gracias a esto se creó rápidamente una gran comunidad de programadores

que se aventuraban a programar algo para la plataforma.

Para poder cargar el software desarrollado para la consola se usa cartucho especial de juego denominado "FlashCart". La "FlashCart" se puede conectar a la consola y al ordenador, no simultáneamente, para copiar en el software programado.

Hay que diferenciar dos tipos de "FlashCart", las que se ubican en el slot 2 de la consola, cartuchos de Game Boy Advance, y las de slot 1, Cartucho de Nintendo DS.

Slot 2(GBA): Fueron los primero cartuchos en salir, no permite ejecutar software de Nintendo DS que requiera funciones táctiles o propias de la consola. Es muy útil tanto para ejecutar software básico como para ejecutar emuladores de GBA, Nintendo, Mega-drive.

Slot 1(NDS): Tardaron algo mas en salir que las del slot 2. Estos FlashCart si permiten cargar software usando toda la funcionalidad que la Nintendo DS nos brinda. A cambio perderemos la compatibilidad y la posibilidad de ejecutar copias de seguridad de GBA y de los emuladores.

2.6. Juegos de lucha

Los juegos de lucha son amados por casi todos. ¿Quién no ha echado unas peleas con algún amigo? Pero, ¿qué es exactamente un juego de lucha? A continuación intentaremos definir qué elementos ha de poseer un juego para que éste sea considerado como tal.

Como bien indica su nombre en estos juegos hay luchas entre dos o más contendientes. Normalmente estos estarán situados uno frente a otro para combatir en un espacio delimitado. Esto no tiene por que ser siempre así, pues podría haber más de un luchador en la pantalla al mismo tiempo formando una batalla a 3, 4 o incluso 8 personajes.

También puede ser que nuestro luchador sea en realidad un equipo de 2 o más. Esto quiere decir que quizás a la hora de elegir, se podrá seleccionar más de un luchador y para obtener la victoria será necesario derrotar a todos los luchadores del equipo contrincante.

Cada personaje poseerá una manera de cuantificar su salud, normalmente son barras de vida situadas en la parte superior de la pantalla, y cuando ésta se agota, ese personaje cae derrotado, la barra disminuye con cada golpe que recibe el personaje una cantidad proporcional al daño de éste.

Cada batalla se lleva a cabo en intervalos de tiempo definidos, normalmente treinta, sesenta o noventa segundos, aunque este tiempo puede ser ilimitado.

Para salir victorioso de una batalla, se ha de derrotar al contrincante un número determinado de veces, cada una de ellas se denomina "round". El estándar es a dos "round", pero puede ser otro.

Cada batalla suele tener lugar en un lugar concreto, dichos lugares pueden ser desde un ring o sobre unas rocas bajo una cascada. El escenario puede poseer objetos para ser recogidos como salud o armas. El escenario no tiene por qué ser estático ya éste puede cambiar debido a roturas de elementos, cambiar drásticamente por un ataque especial o simplemente porque ha transcurrido cierto tiempo.

Los personajes poseerán un elenco de golpes, desde puñetazos o patadas, hasta cabezazos o espaldados, culazos, etc. Estos movimientos podrán ser combinados para formar una sucesión de golpes, denominados "combos". Estas sucesiones son muy poderosas y en ocasiones son muy difíciles de realizar. Cuando se consigue realizar un "combo" con éxito, nuestro personaje suele quedar vulnerable al finalizar, por lo que si no hemos conseguido impactar el combo, estaremos a merced de nuestro oponente.

Los combatientes también podrán bloquear los ataques de su oponente, recibiendo menos daño o directamente no recibiendo daño alguno. También los contendientes suelen poder realizar un movimiento de agarre, donde en caso de ser realizado con éxito, el otro luchador no podrá más que observar cómo es golpeado durante un instante de tiempo, normalmente con una cinemática.

Los personajes pueden poseer magias. Estos movimientos especiales pueden lanzar un proyectil y para realizarlos es necesario una pulsación concreta de botones. No todas las magias han de ser proyectiles, una magia puede ser que nuestro personaje ejecute una sucesión de golpes. La magia más famosa es la realizada en street fighter por "Ryu" y "Ken" entre otros, el Hadouken. Para llevarlo a cabo se han de pulsar abajo, seguido de la diagonal abajo frontal, seguido de frontal y pulsando a la vez que este frontal y un botón de puños ( + P).

También algunos juegos incluyen un sistema para romper los combos, esto significa que si se realiza un combo se puede cancelar sin haberse completado para inmediatamente comenzar otro golpe u otro combo.

Ni que decir cabe, que éstas son solo unas pautas generales y como en todo, las reglas están para romperse. Por ejemplo, en la saga de juegos de Nintendo "smash bros" en vez de vida, los personajes tienen un porcentaje indicando cuan probable es que el personaje salga fuera del escenario con un ataque poderoso, siendo derrotados.

2.6.1. Historia Juegos de lucha

Los juegos de lucha hoy en día gozan notable popularidad, y son queridos por los jugadores. Esto no siempre fue así, pues hubo una época en que estos juegos no gozaban de la popularidad que hoy en día tienen. Este género ha tenido sus altibajos a lo largo de la historia. Este punto es la unión de los artículos [SoyunJugon, 2015], [RacketBoy, 2015], [vicbengames, 2015], [GameFili,

2015], [IGN, 2015] y [Hardcoregaming101, 2015] entre otros.

La historia comienza en 1979, cuando aparece un juego para recreativas llamado "**Warrior**", éste era una versión muy simple de lo que luego llegaría a ser el género, pero ya era una primera aproximación a todo lo que estaba por llegar.

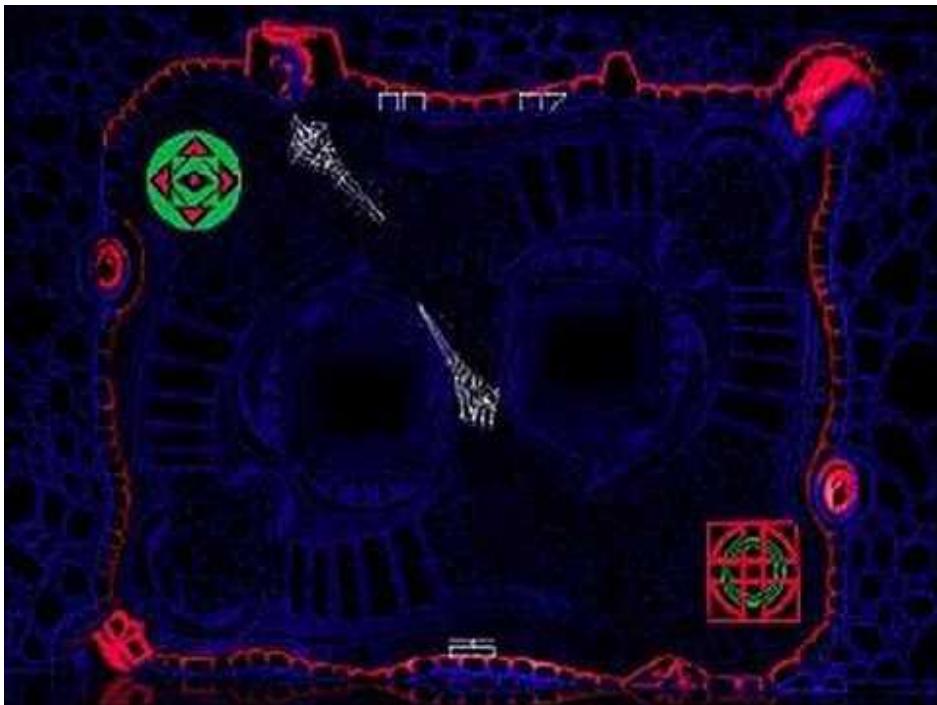


Figura 2.14: Juego "Warrior"

En este juego tenemos una vista cenital de dos luchadores con espadas dibujados en blanco en la pantalla. El objetivo era acercarse al rival y golpearle con la espada para así acabar con su vida de un solo golpe.

Hubo que esperar cuatro años, al 1983 para que otro juego de género similar destacase, esta vez para la plataforma Commodore 64. The "**Attack of the Phantom Karate Devils**" era un juego que no pasó a la historia y muchas veces se obvia cuando se habla de juegos de lucha. Solo lo mencionamos para hacer comprender que los inicios fueron difíciles.

Más adelante en ese mismo año, apareció uno que sí está más en la mente de los aficionados al género y que ya sí cosechó éxito, "**Karate dou**" (o "**Karate Champ**" fuera de Japón).



Figura 2.15: Juego "Karate Champ"

Este título ya incluye aspectos propios del género, como son puntuación, combates con los personajes situados a los lados de la pantalla, movimientos horizontales para acercarnos y poder golpear al enemigo, etc. El juego consistía en que en un tiempo determinado teníamos que golpear al enemigo y acumular puntos. Al finalizar el combate, aquél con más puntos ganaba la batalla. Entre batalla y batalla el juego poseía ciertos mini juegos para evitar la repetición de batallas y que el jugador se aburriese. Mini juegos como evadir objetos voladores, romper objetos o noquear a un toro.

A raíz de este juego el género fue creciendo y haciéndose popular. Esto condicionó que apareciesen múltiples títulos que eran prácticamente copias del "Karate Champ". Entre tanta copia hay algunos que consiguieron asomar la cabeza.

Uno de ellos fue la apuesta de Nintendo al género, "**Urban Champion**". Este título se desarrollaba en un ambiente menos formal que el karate, las peleas callejeras.



Figura 2.16: Juego "Urban Champion"

Este título bebía mucho del Karate Champ, que había empezado a marcar género. En este juego nos encontramos a dos rivales, uno frente al otro, los cuales avanzan hacia adelante o hacia atrás para acercarse al enemigo y golpearlo. Este juego destaca por incluir evasivas, es decir, este título introdujo mecánicas para poder esquivar los golpes del adversario, ya sea dando un paso antes de ser golpeados o directamente bloqueando el impacto. La victoria en este juego se conseguía expulsando al oponente del terreno de lucha, condición de victoria hasta ahora inexistente en el género. Finalmente también aportó también al género el mareo, con el cual después de golpear varias veces seguidas al oponente éste permanecerá quieto durante unos segundos.

Los juegos de lucha empezaron a hacerse un hueco en los salones recreativos y eran cada vez más demandados.

En 1985 y a pesar de no ser el primer título de lucha desarrollado por la empresa "Taito", "**Onna Sansirou - Typhoon Gal**" innovó mucho en el género con mecánicas que perduran hasta nuestros días y sin las cuales no se conciben los juegos de luchas.



Figura 2.17: "Typhoon Gal"

En este juego tomamos el papel de Yuki, que debía de ir sala por sala enfrentándose a múltiples enemigos. Es el primer título en incluir agarres y lanzamientos de enemigos lo cual es la mecánica básica de los juegos de lucha. También incluye la lucha con múltiples enemigos al mismo tiempo, situación no vista hasta este momento.

También en este año destacan dos títulos más, "**Shanghai Kid**" ("Hokuha Syourin Hiryu no Ken" en japon) y "**Galactic Warriors**". El primero destaca por ser el primer juego de lucha en incluir un sistema de combos y golpes especiales pulsando varios botones simultáneamente. El segundo título fue desarrollado por Konami y fue el primer juego en el que podemos seleccionar entre varios personajes, dando así mucha más libertad al jugador. También este título aportó mecánicas como daño aunque bloquee un ataque, ataque de proyectiles, ejecutar varios movimientos en el salto, un sistema de niveles de fuerza en los ataques y personajes con movimientos diferentes. Este juego sí innovó mucho en el género de lucha aunque no goza de la fama que sin duda se merece.

En 1987 nació la leyenda, llega al mercado "**Street Fighter I**" y rápidamente se convirtió en un éxito en los salones recreativos.



Figura 2.18: "Street Fighter 1"

Destacaba por muchos factores, entre otros por sus personajes dotados de un carisma y personalidad propia, características nunca antes vistas en el género. Los personajes poseían movimientos únicos y magias que realizaban por medio de combinaciones de movimientos y botones. También en este juego destacaban los escenarios y el arte en general. Este juego comenzó a popularizar los juegos de lucha gracias a sus constantes enfrentamientos entre amigos. Fue muy exitoso, pero lo podría haber sido mucho más si no hubiese apostado por el sistema de seis botones. El título requería de seis botones para jugar y por aquel entonces lo normal es que las máquinas no tuviesen en esa cantidad de botones, por lo que no fue fácil su exportación y no llegó a todos los lugares que debería.

Ya en 1990 apareció en el mercado otro juego con unos gráficos nunca antes vistos, "**Pit Fighter**", este título apostaba por unos gráficos digitalizados y gran violencia y espectacularidad.



Figura 2.19: Juego "Pit Fighter"

Lucha, acción y violencia se daban la mano en este título para romper los moldes establecidos por "Street Fighter", no con mucho éxito, pero hizo el suficiente ruido para que muchos

recuerden este título y se le considere el precursor de algunos otros.

El año 1991 es para muchos cuando realmente todo comienza. En este año se lanza el título estandarte del género de lucha y que todos conocen y han jugado al menos una vez, "**Street Fighter 2**". Este juego cambió todo, cambió la percepción que se tenían de los juegos de lucha y su modo de realizarlos, fue un adelantado a su época.



Figura 2.20: Juego "Street Fighter 2"

Absolutamente todo era un avance en este título. Las animaciones eran mucho más fluidas que cualquier otro título hasta la fecha. El sonido era soberbio, con voces y bandas sonoras personalizadas por personajes. El elenco de personajes era numeroso con sus ocho personajes jugables, donde cada personaje poseía movimientos únicos que se adaptaban a diferentes estilos de lucha. También destaca sus bonus entre niveles, especialmente uno donde debemos destruir un coche a golpes. Pero lo mejor de todo y por lo que obtuvo el podio de mejor juego de lucha fue por su jugabilidad. Una jugabilidad que hacia que jugar fuera una delicia, con movimientos precisos y con muy buena respuesta a los controles. En definitiva daba gusto jugar por que no se trababa en absoluto y el personaje hacia exactamente lo que querías que hiciese.

"Street Fighter 2" corrió como la pólvora por los salones recreativos y terminó de afianzar las máquinas de seis botones, ya que si un establecimiento no poseía este título, la gente no acudía en masa. Este título a día de hoy sigue siendo una delicia jugarlo, ya que no ha envejecido casi nada.

A raíz de este título la compañía sacó al mercado algunos títulos derivados de éste, como "**Street Fighter II Champion Edition**", el cual añadía algunos personajes más jugables elevando el elenco a doce y también corría errores insignificantes. La versión definitiva salió al mercado en

1994, "**Super Street Fighter II Turbo**" (Super Street Fighter II X en Japón) donde incluía otros cuatro personajes dando un total de dieciséis personajes jugables.

También en 1991 salió al mercado el que sería la apuesta en el género de lucha de "SNK", "**Fatal Fury**". Este título fue de los primeros que bebió directamente de Street Fighter II y se aprecia. Este título poseía unos gráficos similares pero con personajes de mayor tamaño en pantalla y un elenco limitado a tres personajes seleccionables. Su segunda versión tan solo lanzada un año después superó con creces al primer título, dándole una calidad soberbia a los gráficos, incluyendo dieciséis personajes seleccionables y mejorando con creces los movimientos.

En el año 1992 salió al mercado uno de los títulos que ha día de hoy sigue teniendo su versión en cada nueva consola que se lanza, "**Mortal Kombat**". Este título destaca por su violencia y sangre extremas, así como por sus personajes variopintos. Pero lo que destaca principalmente son sus fatalities, que aunque ya se había visto en juegos anteriores aquí es donde se explota hasta su máximo exponente.



Figura 2.21: Juego "Mortal Kombat"

Apostando por los gráficos digitalizados, este título buscó competir con "Street Fighter II" y en cierta medida lo logró debido a su más que notable aspecto visual bien diferenciado. El control, aunque muy diferente a lo que Street Fighter nos tenía acostumbrado, gustó a muchos y se hizo popular. Pero lo que lanzó al estrellato a este título fueron los fatalities, extremadamente sangrientos y violentos. Esta violencia provocó que los medios se hicieran eco del juego y que incluso se prohibiera en muchos países fomentando así que todos los niños quisieran jugar debido a la publicidad indirecta.

Debido a la inmensa popularidad que estaba consiguiendo el género de lucha, aparecieron mil y un juegos diferentes, donde destaca "**Art of Fighting**" con una calidad gráfica increíble y

"King of Monster" con sus personajes gigantes luchando al mismo tiempo que destruían la ciudad.

El año 1993 fue nuevamente un año muy importante para el género con la aparición de grandes títulos como "**Samurai Shodown**", que destacaba por el uso de armas en el combate y el zoom en ciertos momentos de la batalla. Aparece también "**Dragon ball Z super Butoden 1**" y su secuela, destacando especialmente el segundo. Los dos juegos de "Dragon Ball" destacaban en que su pantalla se partía en dos cuando los personajes se separaban lo suficiente y que era muy fiel a la serie en la que se basaba.

Pero si por algo destacara 1993 en el género de lucha, es tener su primera incursión seria en el mundo 3D con "**Virtual Fighter**". Este no es ni mucho menos el primer juego 3D de lucha realizado, pero si el primero el conseguir cierta fama.



Figura 2.22: Juego "Virtual Fighter"

Anteriormente se crearon juegos en 3D, pero su calidad dejaba mucho que desear debido a que las plataformas no poseían la potencia técnica suficiente. "**Virtual Fighter**" consiguió con pocos polígonos realizar un destacable juego de lucha, con todos los clichés del género.

En 1994, el género de lucha estaba en pleno apogeo y debido a esto aparecieron en el mercado multitud de juegos, muchos de ellos con una gran calidad tanto gráfica y gran jugabilidad. Es conveniente destacar títulos como "**Art Of Fighting 2**" que mejoraba la formula del primero aportando más personajes y calidad en general pero sin deslumbrar. Destacamos también "**Samurai Shodown 2**" que también usaba la misma formula que el primer título pero aportando mucha más jugabilidad para dejar un producto bastante redondo. Otro título fue "**Darkstalkers**" ("**Vampire The Night Warriors**" en su versión japonesa) donde se daban lugar combatientes de las historias de terror clásicas como Drácula, un hombre lobo o una momia entre otros, destacando en este juego su magnífico acabado y sus similitudes con Street Fighter

pero dándole la vuelta de tuerca para mejorar aun más si cabe. Especial mención al juego "**X-Men: Children of the Atom**" donde por primera vez se daba lugar un combate entre los personajes de la franquicia de cómics Marvel, los X-Men más concretamente, con un juego similar a Street Fighter 2 en cuanto a jugabilidad y solo incluyendo unas pocas mecánicas como un super salto y poco más. Este juego fue el primero y el precursor de lo que más adelante se convertiría en la saga "Marvel VS Capcom".

Pero sin duda, si este año hay que elegir un juego de lucha, ese es "**The King of Fighters 94**". El golpe sobre la mesa de este juego fue casi equivalente a lo que ya hiciera "Street Fighter 2" allá por el 1991.



Figura 2.23: Juego "The King of Fighters 94"

Este título es considerado el primero en el que se lleva a cabo un "crossover" en un juego de lucha. Un "crossover" consiste en incluir a personajes de otros juegos y juntarlos en uno solo. En este juego encontramos personajes de "Fatal Fury" y "Art of Fighting", entre otros. En este juego no se elige a un solo luchador, en él se elige a un país que ésta representado por tres luchadores ya preestablecidos, resultando en el juego un total de ocho países y de 24 luchadores distintos, tres luchadores por equipo. Para vencer el combate has de vencer a los tres luchadores del equipo rival sin perder a tus tres luchadores. Esto suponía una renovación del género, aunque aún contaba con limitaciones como cambiar de personajes a placer. También destaca su completísimo apartado de efectos y sonidos, dando lugar a un juego redondo y que supo plantarle cara a "Street Fighter 2". Esta misma franquicia pasaría a ser casi anual, mejorando cada año al anterior apareciendo hasta 2010 con un total de 23 juegos en su haber.

Debido a la potencia gráfica que permitían las consolas, muchas empresas empezaron a apostar por el 3D.

Recién empezado 1995, más bien finalizando 1994, apareció un título muy a tener en cuenta, "**Killer Instinct**". Este título bebía mucho del ya popular "Mortal Kombat" pero aportando quizás

más carisma a sus diez personajes seleccionables y eliminando sangre. Destaca especialmente por su sistema de combos y su posibilidad de cortarlos dando lugar a la frase "combo breaker" que es recordada por muchos con nostalgia.

Muy temprano en este mismo año aparece en el mercado un título que sigue cosechando fama con cada nuevo título que aparece y que ya va camino del séptimo, sin contar "crossovers" y líneas paralelas. No es otro que "**Tekken**", que con su 3D bastante avanzado para la época supo encandilar a muchos.



Figura 2.24: Juego "Tekken 1"

Este título sorprendió a todos por su buen uso del 3D, aportando una calidad y estilo casi únicos. También modificaba un poco la jugabilidad ya que cada botón controla una extremidad. También siguió la estela de muchos otros títulos ofreciendo personajes que dominaban estilos de lucha existentes en la realidad.

Este mismo año es lanzado un juego de estética similar a "Tekken" pero con una jugabilidad totalmente distinta, "**Soul Edge**". Título que a su salida no fue muy afamado pero que tendría mucho que decir con sus secuelas.



Figura 2.25: Juego "Soul Edge"

Este título se caracteriza por que cada personaje lucha con un arma distinta, dando lugar a un elenco muy peculiar. Nuevamente los controles varían del sistema clásico ya que este juego posee un botón para el ataque en vertical y otro para el horizontal, además de cobertura y una patada. Este juego fue el primero de una saga de juegos que luego pasaría a llamarse "Soul Calibur" en 1998, dando lugar a otra franquicia que sobrevive a día de hoy.

Por supuesto en este año también salieron al mercado numerosos juegos en 2D para afianzar aun más el género, género que empezaba a desinflarse poco a poco. Títulos como "**Far East Of Eden**" con una jugabilidad increíble y unos gráficos increíbles. También otro título en un intento de relanzar la saga "Street Fighter" con "**Street Fighter Alpha**" aportando nuevos gráficos estilo japonés y uniendo estos personajes con algunos de otros juegos de lucha.

En 1996 el género de lucha comenzó su declive poco a poco y ya pocos títulos llamarían la atención como ocurrió anteriormente. Aun así, apareció otro título en 3D que si destacaría un poco, más por su controversia que por su jugabilidad, "**Dead or Alive**". Este título desató cierta polémica debido al busto de sus luchadoras. Estos bustos poseían un movimiento exagerado realizado únicamente para llamar la atención y lo consiguió. También en este año aparecieron múltiples secuelas de juegos ya pasados como "**Art of Fighting 3**", "**Samurai Shodown 4**" o "**King of Fighter 96**". Especial mención a que el rey, "Street Fighter", intentó este año arañar las tres dimensiones con su juego "**Street Fighter Ex**", pero se dio de bruces con ella.

1997 junto con 1998 supondría unos de los últimos empujones por intentar revitalizar el género.

Para empezar en 1997 y ahora que el 3D estaba tan en boca de todos, aparece por la puerta pequeña un título que sorprendió, "Bloody Roar". Sus transformaciones aportaron algo diferente al género que empezaba a estancarse.



Figura 2.26: Juego "Bloody Roar"

En este título y después de llenar una barra de energía que se recarga al golpear o ser golpeado, podemos transformarnos en un animal u otro dependiendo del personaje. De este modo cambia los ataques y la jugabilidad en general de nuestro personaje.

No mucho después de este título, aparece en el mercado un título que para muchos es uno de esos grandes títulos olvidados, "Rivals Schools United By Fate". Sin aportar demasiado, es un título muy a tener en cuenta.



Figura 2.27: Juego "Rivals Schools United by Fate"

La principal novedad en este título es que además de seleccionar dos personajes para la batalla, podemos realizar ataques combinados, quedando espectacular en la pantalla.

Este año también el rey da un golpe en la mesa y harto de versiones de su obra culmen como "**Street Fighter II Turbo**" y "**Street Fighter Champion**", harto de sus intentos con los crossovers como "Street Figher Alpha 1" y "Street Figher Alpha 2", y sus intentos en el mundo 3D con "Street Fighter EX", lanza al mercado la continuación de la saga, "**Street Fighter 3**". Con unos gráficos 2D nunca vistos hasta la época debido a su fluidez y estilo propio.



Figura 2.28: Juego "Street Fighter 3"

En este juego se descartan casi todos los personajes que le dieron la fama a la saga apostando por nuevos personajes y estilos de lucha. La suavidad y el detallado de los "sprites" es máximo, dando una experiencia visual nunca antes vista para un juego de lucha en dos dimensiones aunque la jugabilidad se mantuviese respecto a "Street Fighter II". No tardarían mucho tiempo en rectificar, lo que para muchos fue considerado un error, e incluir todos los personajes en las reediciones del título, quedando como "**Street Fighter III 3rd Strike**", lanzado en 1999, como la culminación de esta saga.

Ya a finales de 1997 al rey le aparece un duro competidor. Esta vez en 3D, como venía siendo habitual de la saga de juegos, aparece el que es considerado por muchos el mejor juego de la saga "Tekken", el "**Tekken 3**".

También este año aparecen otro títulos en 2D destacables como "**The Last Blade**", "**Darkstalkers 3**" y "**Marvel Super Heroes vs. Street Fighter**".

En 1998 hay nuevamente un aluvión de juegos y este intento de relanzar el género de 1997 va perdiendo fuelle, no sin antes despuntar con algunos títulos.

Nuevamente nos encontramos con títulos de segundas, terceras o cuartas versiones como

pueden ser el respectivo el "King of Fighter" de todos los años, "**Last Blade 2**" o el primer "**Marvel vs Capcom**".

Pero hay un título que destaca este año, "**Guilty Gear**". Apostando por una calidad gráfica soberbia y un increíble trasfondo a los personajes.



Figura 2.29: Juego "Guilty Gear"

Este juego rápidamente consiguió adoradores, aunque no demasiados. Destacaba especialmente el trasfondo de los personajes que luego serían ampliados con más historias, cómics, etc. y también destacaban sus gráficos con multitud de efectos que lo dotaban de una visibilidad increíble. Esto unido a sus trece personajes bien diferenciados hicieron un buen título que vino para quedarse y que a día de hoy siguen desarrollando nuevas ediciones y reediciones.

Y finalmente ocurrió, esta cuesta arriba que había comenzado cuando en 1991 cuando se lanzara aquel "Street Fighter 2" había llegado a su fin, ya no se podía seguir elevando este género y cayó en picado.

En los años 1999 y 2000 solo destaca la apuesta de Nintendo con "**Super Smash Bros**", que más que intentar aportar al género basándose en el estándar, eligió despuntarse y tomar otro rumbo aportando su sello personal, como Nintendo siempre hace.



Figura 2.30: Juego "Super Smash Bros"

En este título podían jugar hasta cuatro jugadores al mismo tiempo y su plantel de personajes estaba compuesto por personajes de otros juegos de la talla de Mario, Link o Donkey Kong entre otros. Supuso un éxito bastante notorio pero todo quedó en casa de Nintendo y su Nintendo 64.

Por supuesto estos años también aparecieron secuelas de algunos juegos anteriores destacando **"Fatal Fury Wild Ambition"** en tres dimensiones, **"Guilty Gear X"** que mejoraba lo realizado en el primer juego, **"Marvel vs Capcom 2"** y **"Capcom vs SNK"**. Capcom le había cogido gusto a esto de los crossover como intento de atraer a público procedente de distintos juegos.

Lo que sí se es digno de mención sobre el 1999 es que una pequeña empresa realiza un motor juegos de lucha bajo licencia libre, y lo deja para que todos lo puedan usar, **"M.U.G.E.N."**. Este motor ofrece muchas facilidades para añadir personajes nuevos, dejando su uso a la comunidad y a quien deseara hacer su propio juego de lucha. Esto nos viene a decir que aunque el género carecía de la popularidad conseguida en el pasado, aun había un público fiel que quería seguir disfrutando de juegos de lucha.

En los dos años siguientes, 2001 y 2002, ocurrió más de lo mismo, algunos títulos destacaban sobre otros, pero sobre todo eran títulos que ya tenían su público fijo y las compañías apostaban por ellos sabiendo que no obtendrían un exitazo pero sí ciertas ventas. "Capcom" repite con **"Capcom vs SNK 2"**, "Tekken" intenta conseguir un éxito similar a "Tekken 3" con **"Tekken 4"**, sin conseguirlo. Continua "King of Fighters" sin cansarse e innovando mínimamente entre versiones. Y **"Dead or Alive 3"** aparece con el mismo revuelo que la primera y segunda edición.

Los únicos que consiguen recuperar algo de gloria fueron "Mortal Kombat", que después de algunos juegos consiguen recuperar algo de su gloria pasada con **"Mortal Kombat Deadly**

Alliance" (también llamado "**Mortal Kombat Vengeance**"), para muchos el mejor título de la saga. La saga "Soul Calibur" sacó al mercado un digno título de su saga iniciada allá por el 1996 y con solo dos en su haber, "**Soul Calibur 2**" que es para muchos el mejor de toda la saga. Para terminar este año no podemos olvidar que la saga "Super Smash Bros" saca al mercado "**Smash Bros Melee**", que aunque la consola no era muy popular, el juego consiguió una popularidad destacable.

Años 2003 y 2004. La historia se repite como lleva pasando en los últimos años, ningún título destaca por encima de otros y se suceden secuelas de los títulos que aún quieren morder esta manzana que está bastante mordida. Solo pueden destacar títulos como "King of Fighters". Aparece otro "Guilty Gear", afianzando a su público y definiéndose como un título destacable. Otro título es "**Samurai Shadowns V**" saga a la cual el 3D nunca terminó de encajarle del todo bien. Y se lanzan infinitas reediciones de la saga "Street Fighter" tanto del segundo como del tercero.

En 2003 y 2004, lo único destacable es un título que gana este honor por su fidelidad con su anime, "**Dragon Ball Z: Budokai Tenkaichi**". Títulos basados en animes ha habido desde el principio del género, pero este consigue trasmitir la especial esencia de este anime.



Figura 2.31: Juego "**Dragon Ball Z: Budokai Tenkaichi**"

Este juego logró sorprender por su calidad gráfica usando el estilo "cell shading" para que se pareciese al anime y a su elenco de personajes. Todo esto unido con su jugabilidad y los personajes con habilidades similares a su homónimo animado, hicieron un juego ampliamente recordado para los fans del género. No tanto el primero de la saga sino sus sucesores fueron los mejores. En especial "**Dragon Ball Budokai Tenkaichi 3**" poseyendo este la increíble cifra de 161

personajes jugables, la mayor de un juego de basado en la franquicia "Dragon Ball" y una de las mayores cifras en un juego de lucha.

En los dos siguientes años, 2005 y 2006 vuelve a ocurrir lo que venía pasando unos años atrás, aparecen versiones sin especial glorias y refritos de juegos para contentar a ese pequeño colectivo amante de los juegos de lucha. Aparece nuevamente un "Samurai Shodown", "**Samurai Showdown VI**". Aparece otro "Guilty Gear", "**Guilty Gear XX Slash**". Nuevamente la saga "King Of Fighter" publicó otro título, "**King Of Fighter Maximum Impact 2**". Sí que aparecen nuevos títulos como "**The Melty Blood**" y "**Arcana Heart**". Y como juegos más destacables de estos años son las continuaciones de las sagas más populares como "Soul Calibur", con su "**Soul Calibur III**"; "Virtual Fighter", con su "**Virtual Fighter 5**"; y "Tekken", con su "**Tekken 5**". Quizás destacando este ultimo por que consiguió cierta popularidad.

En los siguientes dos años, 2007 y 2008, ocurre un poco como ya pasara en 1998 y se intenta relanzar el género con múltiples apuestas de buen nivel. "Tekken" pule su "Tekken 5" y lanza su "**Tekken 6**", juego muy similar al anterior pero con mejoras en casi todo. "Smash Bros" saca su nuevo juego para la Wii, "**Super Smash Bros Brawl**", destacando su elenco de personajes cada vez más completo. También la saga "Soul Calibur" saca al mercado "**Soul Calibur IV**" siendo éste un muy buen juego que rivaliza con "Tekken 6". "Mortal Kombat" saca un título en "crossover" con los comics de "DC Universe", "**Mortal Kombat vs. DC Universe**" con un discreto éxito. Incluso en estos años, 2008 más concretamente, aparece una nueva franquicia que da un golpe en la mesa para hacerse destacar como ya hiciera "Guilty Gear", hablamos de "**BlazBlue: Calamity Trigger**" que aporta algo de frescura al género.

Pero sin lugar a dudas el título del año en cuanto a género de lucha se refiere fue la vuelta del rey, "**Street Fighter IV**". Apostando por fin por unos gráficos en 3D pero de un modo poco convencional y sin perder un atisbo de jugabilidad.



Figura 2.32: Juego "Street Fighter 4"

Este título ofrecía al amante de la saga otro título para disfrutar con sus personajes favoritos sin que sufriera en el cambio a un nuevo juego, y también ofrecía un aire fresco para los que se encontraran con la saga por primera vez. Ofrecía todas las mejoras que había ido consiguiendo a lo largo de los años, un gran elenco de personajes muy bien diferenciados entre ellos y un sistema de súper ataques muy bien llevado. Se convierte en un indispensable de juego de lucha en cuanto aparece, generando torneos por todo el globo.

En 2009 y 2010 y después de un 2008 despuntando, la situación vuelve a su cauce y nuevamente nos encontramos en unos años donde ningún título destaca. Esto no impide que aparezcan títulos en el mercado. "Street Fighter" sigue con sus refritos pero esta vez de "Street fighter IV", aportando más personajes y mejorando pocas cosas. "King of Fighter" saca dos títulos en estos años, "**King of Fighter XII**" y "**King Of Fighter 2002 Unilimited Match**". "BlazBlue" saca al mercado su segundo y tercer juego, "**BlazBlue Continuum Shift**" y "**BlazBlue Continuum Shift II**", mejorando su formula. Este año Capcom saca al mercado otro de sus "crossover", "**Tatsunoko vs. Capcom: Unilimite All-Stars**", pero con modesto éxito.

2011 y 2012 nuevamente son años escasos de títulos, tan solo destaca uno sobre los demás. A pesar de que "Mortal Kombat" saca un título, "**Mortal Kombat 2011**", éste no logra destacar y pasa al olvido casi inmediatamente. "Soul Calibur" saca al mercado su quinta edición, "**Soul Calibur V**", pero le ocurre algo parecido a Mortal Kombat y no destaca para nada. También en estos dos años ocurre un crossover muy deseado por los aficionados de dos sagas, "Street Fighter" y "Tekken", resultando el título "**Street Fighter X Tekken**" que adapta la jugabilidad de "Street Fighter" a "Tekken" quedando un título que suena mejor de lo que realmente es, dejando un sabor agridulce a quien lo juega. Otro crossover que ocurre estos años es "**PlayStation All-Stars Battle Royale**" que reúne a personajes de juegos de "Play Station". Con este juego "Sony" pretende emular el éxito de Smash Bros pero con penoso resultado.

El título que sí llama un poco más la atención este año fue "Marvel vs. Capcom 3: Fate of Two Worlds", que una vez más trae a todos los personajes de "Marvel" y "Capcom" a la contienda. Con buen acabado y aprovechando un poco la reciente popularidad de "Marvel" conseguiría un éxito aceptable.

2013 y 2014 han sido años aceptables para género. Aparecen algunas secuelas aceptables de "Killer Instinct" con el "**Killer Instinct 3**" o "Dead or Alive" con su "**Dead or Alive 5**", que aunque no destaquen excesivamente, sirve para contentar a los aficionados a estas sagas. Aparecen algunos juegos nuevos en el plantel como "**Injustice: Gods Among Us**" que es un juego con los personajes del universo "DC" y muy inspirado en la saga "Mortal Kombat" pero con mucha menos violencia y sangre. También aparece un juego para los muy aficionados al género debido a su dificultad y complejidad, "**Persona 4 Arena Ultimax**", juego basado en el juego de rol

"Persona 4". Estos años destacan especialmente por salida al mercado del esperado "**Super Smash Bros for Nintendo 3DS y Wii U**" que consigue una gran fama en el mercado y se posiciona casi como el mejor de la saga.

Ya para terminar, en el año 2015 ocurre la vuelta de algunos de los grandes del género. Para empezar y ya lanzado tenemos a "**Mortal Kombat X**", el cual y según los aficionados ha conseguido estar a la altura de lo esperado ofreciendo un título de calidad. También este año vuelve "Tekken", con "**Tekken 7**", que sigue la estela de "Tekken 6" en cuanto a gráficos y jugabilidad, mejorando poco a éste y aportando poco y algunos personajes. Un título que sí dio la sorpresa y que está causando buena expectación es "**Pokken Tournament**", donde se llama a la batalla a los "pokemon".



Figura 2.33: Juego "Pokken Tournament"

En este título los personajes son "pokemon" que luchan entre sí pero abandonando el sistema por turnos por un control directo y en tiempo real. Es desarrollado por la misma compañía de "Tekken". Otro título que sin duda dará que hablar es la vuelta del "Street Fighter" con "**Street Fighter V**". De este título poco se sabe por ahora, solo que tendrá un estilo visual similar a su antecesor, pero sin duda dará que hablar.

2.7. Touhou

Touhou [Touhou Wiki, 2015] es una saga de juegos de naves, "shoot'em up", creada por "Team Shanghai Alice", empresa compuesta por un solo integrante. Su primer título fue el "Touhou 1", lanzado en 2006, y aunque este título pasó un poco desapercibido, la saga no tardó mucho tiempo en alcanzar la modesta popularidad que posee actualmente.

Los juegos de esta saga tienen la jugabilidad clásica de los juegos de naves con movimiento vertical, donde nuestro personaje, una nave, avanza por el escenario y los enemigos van apareciendo frente a nosotros. Estos títulos tienen la peculiaridad de que es más importante

evitar los disparos enemigos que atacarles y derrotarlos. Es común en estos juegos que llegue un momento donde tan solo haya un espacio posible por el que huir de los disparos enemigos. Otra característica de todos los títulos es que la gran mayoría de los personajes son femeninos.

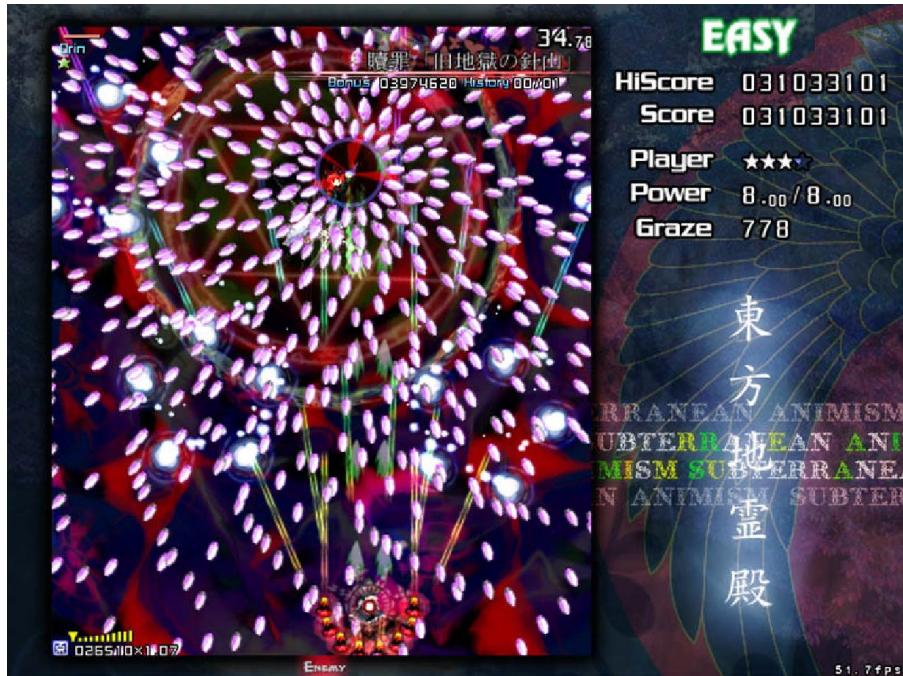


Figura 2.34: Juego "Touhou 11"

Aunque esta saga era solo de juegos de naves en sus inicios, hoy en día hay multitud de otros productos como música, novelas ligeras, novelas visuales, manga, anime y juegos de lucha. Estos últimos son los que nos resultan interesantes, ya que para realizar el juego de lucha del capítulo seis nos hemos basado en uno de ellos.

2.7.1. Touhou 10.5. Scarlet Weather Rhapsody

Segundo título de lucha de la saga "Touhou". Como ya ocurriera con el "Touhou 7.5", "Team Shanghai Alice" se alía con la compañía "Twilight Frontier" para el desarrollo de este título, aportando la primera compañía la historia y música y la segunda el resto en su desarrollo. Fue lanzado al mercado en el año 2008. Este juego se titula "Touhou 10.5" debido a que en la historia, se sitúa entre el "Touhou 10" y el "Touhou 11".

Este juego entra dentro del corte clásico de juego de lucha. En él, los personajes tienen barra de vida, barra de poder que se va acumulando con el tiempo, se juega a un tiempo preestablecido y hay super ataques. Los controles de este juego pueden resultar algo distintos a lo habitual, ya que este título dispone de un botón de golpear y dos botones para lanzar proyectiles.



Figura 2.35: Juego "Touhou 10.5"

Este juego posee quince personajes, todos femeninos, y algunos tienen movimientos y jugabilidad especiales que los diferencian del resto. Otra peculiaridad de este título es la inclusión de un clima en batalla. Este clima altera las condiciones de la batalla dependiendo de qué clima está activo, permitiendo por ejemplo hacer más daño o drenar vida del oponente entre otras propiedades.

Una característica que ya incluía el título de peleas anterior es la inclusión de cartas. Cada personaje tiene un elenco de cartas que puede personalizar antes de la batalla y condicionan qué ataques especiales posee el personaje.

2.7.2. Touhou 12.3. Touhou Hisotensoku

Tercer título de lucha de la saga Touhou. Como ya ocurriera con el título anterior, "Team Shanghi Alice" colabora con "Twilight Frontier" para el desarrollo de este título, el primer estudio aportando la historia y música y el segundo el resto. Fue lanzado al mercado en el año 2009. Este título es nombrado "Touhou 12.3" debido a que si historia se sitúa entre los títulos "Touhou 12" y "Touhou 12.5".

Este juego, más que como un juego propio funciona como expansión del juego anterior "Touhou 10.5". Debido a que este juego extiende al juego anterior posee todas las mismas características pero solucionando algunos problemas y aportando nuevos personajes.



Figura 2.36: Juego "Touhou 12.3"

Este juego añade cinco personajes a su predecesor, quedando a un total de veinte contendientes a elegir, todos femeninos nuevamente. La mecánica de clima cambiante no es modificada en este título, quedando exactamente igual que en el "Touhou 10.5". Sí se añaden multitud de cartas y nuevas animaciones para los personajes existentes quedando así un título más redondo que su predecesor y dejando mejores sensaciones en general.

Capítulo 3. Tecnología empleada

En este capítulo se explica brevemente el software y las herramientas usadas para el desarrollo del proyecto. Se explicará de forma breve para no sobrecargar con datos innecesarios y se profundizará en aquel software más desconocido a nivel general.

3.1. DevKitPro

DevkitPro [DevKitPro, 2015] es un descargable que incluye una serie de herramientas "open source" de programación para las plataformas "Wii", "Game Cube", "Nintendo DS", "Game Boy Advance", "GP 32" y "Play Station Portable".

Estas herramientas incluyen librerías para poder programar con el lenguaje C para cada plataforma mencionada, emuladores para sus pruebas en el ordenador y una serie de editores para mayor facilidad de trabajo.

En el capítulo 4.1 explicamos en detalle cómo llevar a cabo su instalación para comenzar a trabajar.

3.2. Librería Libnds.

Librería open source para el lenguaje C creada por Michael Noland y Jason Rogers. Ésta da soporte a la consola Nintendo DS y pretende ser una alternativa a la librería oficial de Nintendo. Es la librería más completa de todas las existentes para la plataforma, dando soporte a todas las funcionalidades de la consola: pantalla táctil, botones, micrófono, carga elementos 3D y carga elementos 2D.

Es una librería muy completa y optimizada, siendo la librería a la que al final se recurrirá para realizar una óptima programación para la plataforma.

3.3. Librería Palib.

Esta librería toma como referencia a "libnds" facilitando su uso mediante nuevas funciones más sencillas. También está desarrollada para el lenguaje C. De este modo pretende ser una vía de entrada para nuevos programadores que deseen realizar juegos simples para la consola Nintendo DS. Aunque sí es cierto que a veces se queda algo corta y hay que recurrir a "libnds" directamente para algunas funcionalidades.

Esta librería no cuenta con soporte actualmente y no es muy buena alternativa debido a su poca potencia final.

3.4. Librería NightFox lib.

De igual modo que librería PAlib, esta librería [NightFoxLib, 2015] toma como referencia a

libnds, facilitando su acceso e incluso en ocasiones mejorándola. Esta librería ha sido realizada por Cesar Rincón para el lenguaje C.

Esta librería es mucho mejor que la antes mencionada, "PAlib", ya que está mejor implementada y es mucho más óptima. Nuevamente esta librería facilita las funciones ya dadas por libnds. Esta librería será explicada con detalle en el capítulo 4.

3.5. Visual Studio

"Visual Studio" [Visual Studio, 2015] es un software propietario de Microsoft para el desarrollo de aplicaciones en los lenguajes C++ y C#, entre otros.

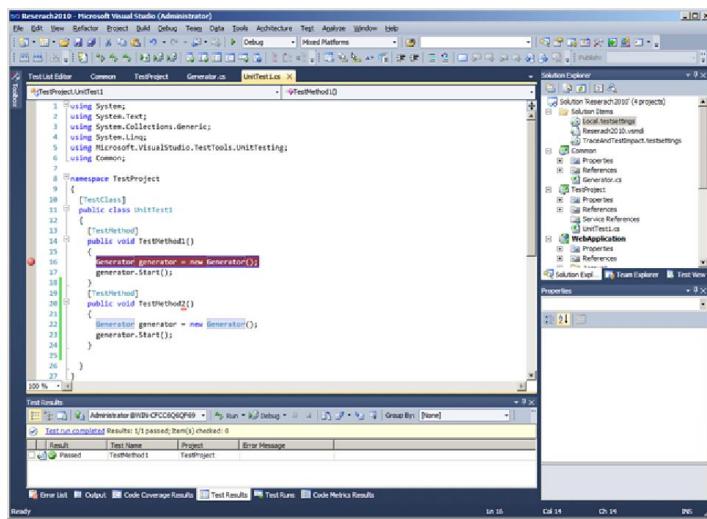


Figura 3.01: Programa "Visual Studio"

Este programa destaca por su depurador, el cual es bastante potente y accesible para los programadores. El "software" incluye lo estándar que incluyen todos los entornos de desarrollo, como es auto completado de funciones, una buena detección de errores y un potente compilado. Muy flexible a la hora de compilar por partes, para no tener que compilar todo el código en el caso de que éste sea de gran envergadura.

3.6. Lenguaje C.

Lenguaje de programación no orientado a objetos creado en 1972 por Dennis M. Ritchie como evolución del lenguaje B. Su primer estándar aparece en 1989.

Segun [C, 2015] en primera instancia era un lenguaje orientado a crear sistemas operativos pero a derivado a mucho más. Es un lenguaje de datos estático, tipificado y de medio nivel.

Este lenguaje es la base de muchos otros lenguajes debido a su gran potencia y flexibilidad, quizás no sea el mas cómodo de usar pero lo compensa con creces debido a la posibilidad de trabajar a muy bajo nivel accediendo a la memoria directamente o gracias a la capacidad de

escribir código maquina con él.

A lo largo de la historia surgen nuevos estándares para el lenguaje otorgándole mas flexibilidad y potencia. La última de ésta ha sido el estándar publicado en 2011 que ha añadido la posibilidad de "multi-threading".

3.7. Lenguaje C++

Lenguaje de programación orientado a objetos basado en el lenguaje C diseñado en 1980 por Bjarne Stroustrup.

Como bien indica [CPlusPlus, 2015] su principal diferencia respecto a C es la inclusión de clases, otorgándole gran potencia. Aunque ésta es la principal diferencia no es la única que presenta en contraposición a su hermano menor C. Algunas otras diferencias son cambios en la librería estándar de entrada y salida, ciertas palabras reservadas, plantillas, sobrecarga de operadores, contenedores, etc.

Al igual que ocurre con C, tiene un estándar el cual rige qué es correcto y qué no en el lenguaje. El último estándar es el C++14, que fue lanzado en 2014 y ya se habla de un nuevo estándar para 2017.

3.7. XML

Siglas en inglés de "eXtensible Markup Language", es un formato de texto para almacenar información de forma legible. Segun [XML, 2015] existe desde 1986 y está basado en otro formato llamado "SGML".

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Edit_Mensaje SYSTEM "Edit_Mensaje.dtd">
<Edit_Mensaje>
  <Mensaje>
    <Remitente>
      <Nombre>Nombre del remitente</Nombre>
      <Mail> Correo del remitente </Mail>
    </Remitente>
    <Destinatario>
      <Nombre>Nombre del destinatario</Nombre>
      <Mail>Correoo del destinatario</Mail>
    </Destinatario>
    <Texto>
      <Asunto>
        Este es mi documento con una estructura muy sencilla
        no contiene atributos ni entidades...
      </Asunto>
      <Texto value="2" />
    </Texto>
  </Mensaje>
</Edit_Mensaje>
```

Como se aprecia, el formato es similar a HTML y esta basado en "tags", solo que no hay tags

especiales, es decir, los tags son inventados a conveniencia. También estos tags pueden tener atributos.

Capítulo 4. Tutoriales

En este capítulo, se explicarán todos los pasos para aprender a programar en la NDS con la librería NFL. Se llevará a cabo una serie de tutoriales para cada funcionalidad concreta. Se explicarán las funciones y su uso, y se afianzará los conocimientos con pequeños ejemplos.

Estos ejemplos serán realizados en C, usando la librería antes mencionada, NightFox Lib.

4.1. Instalación de entorno

Antes de empezar a programar, es necesario instalar las herramientas para ello. Comenzamos instalando el devkitpro [DevkitPro, 2015] de su página Web. Lo descargamos e instalamos como un programa más de Windows.

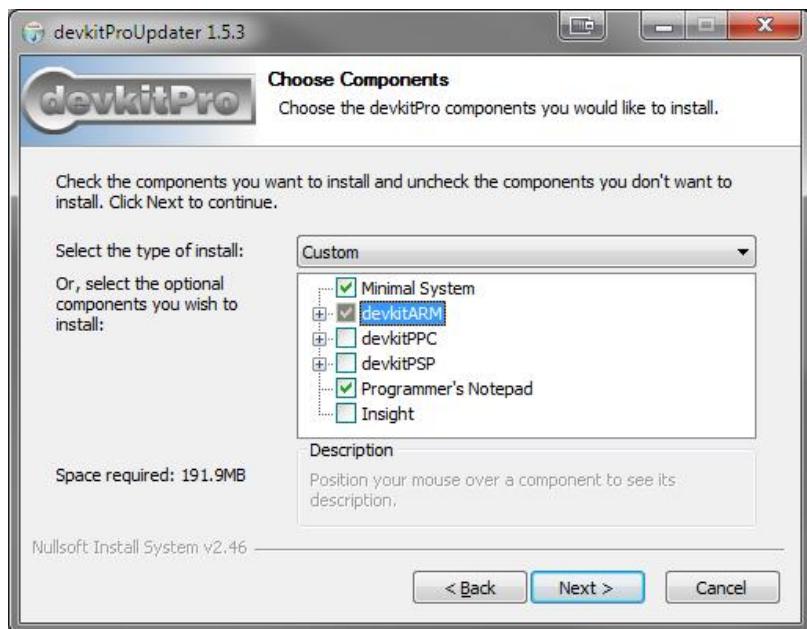


Figura 4.01: Ventana de instalación de "DevKitPro"

En este menú quitamos todo lo referente a las librerías de programación para "Game Boy Advance", para "Game Cube" y para "Play Station Portable".

En nuestro caso lo instalaremos en el directorio "nds" situado en la raíz del disco duro, "C". La instalación descargará los archivos necesarios y los instalará.

Ahora procederemos a la descarga de la librería "NightFox". Para ello accedemos a su Web [NightFox Lib, 2015] y nos descargamos la última versión. Ésta la descomprimimos en la carpeta "nds" junto a la carpeta "devkitpro".

Ya para terminar, en la localización "C:\nds" creamos otra carpeta, junto a las dos mencionadas, llamada "Projects" y dentro de ésta otra llamada "Tutorials". En este directorio es

donde albergaremos nuestro trabajo. Las carpetas donde ubiquemos nuestros proyectos deben poseer un nombre sin espacios en blanco.

Una vez finalizado, es mejor reiniciar el equipo para prevenir posibles errores derivados de las variables de entorno que "devkit" añade.

4.2. Herramientas a usar

Usaremos diversas herramientas a lo largo del desarrollo del proyecto. Las más relevantes son descritas a continuación:

Emuladores: En la carpeta de la librería NFL ("C:/nds/nflib") hay varios emuladores para poder probar el código escrito en el ordenador, aunque para comprobar el correcto funcionamiento se recomienda la NDS, ya que puede variar el resultado.

En nuestro caso, creamos otra carpeta en la ubicación "nds" llamada "emulators" y ahí copiamos el contenido de la carpeta "nflib/tools/emu".

Para usar los emuladores, abrimos el ejecutable que trae cada emulador y abrimos el archivo con extensión ".nds" que nos genera al compilar.

Conversores: La NDS no puede leer directamente de los formatos populares (bmp, png, etc). Por este motivo NFL también incluye un conversor de imágenes. Este conversor es un programa llamado "grit" que se encuentra en la carpeta "nflib/tools/grit". Se han facilitado herramientas para su uso mediante ejecutables, con extensión ".bat", para cada caso concreto.

Para usar "grit" hay que introducir las imágenes, con extensión "bmp" y modo indexado, en el directorio "bmp", situado donde está el ejecutable. Luego hay que usar el "bat" correspondiente, "Convert_Sprites" para sprites, "Convert_Backgrounds" para fondos, etc. Esto nos generará un archivo por cada imagen usada y una adicional con la extensión "pal" en la carpeta correspondiente, "sprites" para imágenes sprites, "backgrounds" para los fondos, etc.

4.3. Hello World.

Para comenzar creamos una carpeta en el directorio "projects" para tener nuestro primer tutorial diferenciado del resto. A este directorio lo nombramos como el número de tutorial que corresponde, "T.4.3.Hello World". En cada tutorial crearemos una carpeta específica para albergarlo.

De la carpeta de la librería NFL, copiamos el contenido de la carpeta "template" y lo pegamos en nuestra carpeta recién creada, de este modo tenemos un proyecto de ejemplo. Para evitar problemas futuros, copiamos también el archivo "Makefile" ubicado en la carpeta "nflib/makefiles/NifroFs", sustituyendo al que ya habíamos copiado procedente de la carpeta "template". Esto lo hacemos para poder compilar más adelante en C++ ya que el makefile nativo

no nos da dicha opción.

Para obtener el ejecutable hacemos clic en "compile.bat". Después de esto aparecerán dos archivos llamados con el mismo nombre de nuestra carpeta, uno con la extensión "elf" y otro con "nds". Este último es el que debemos probar en nuestro emulador o consola.

El código está ubicado en la carpeta "source", en un archivo llamado "main.c". Analizando dicho archivo ya encontramos ciertas funciones propias de la librería.

Lo primero que encontramos en el código, son las inclusiones necesarias. La librería "nds" es la librería nativa de NDS y la siguiente librería es la propia de la NFL, "nf_lib.h".

```
#include <nds.h>          // Includes propietarios NDS
#include <nf_lib.h>        // Includes librerías propias
```

Ya una vez en el main, encontramos tres llamadas a funciones básicas. Estas funciones son de libNDS y no de NFL. Son las siguientes:

```
consoleDemoInit();           // Inicializa la consola de texto
consoleClear();              // Borra la pantalla
setBrightness(3, 0);         // Restaura el brillo
```

La primera de ellas inicia el modo texto para el modo debug mediante printf. La segunda línea borra la pantalla y reinicia todas las variables de la librería. La tercera cambia el brillo de pantalla, siempre que sea posible.

Como todo sistema de juegos, la programación para NDS consiste en un bucle infinito.

```
printf("\n Hello World!");
While(1) {
    swiWaitForVBlank();      // Espera al sincronismo vertical
```

La línea "swiWaitForVBlank" se usa para esperar a que se termine de actualizar la pantalla. Dicha línea es esencial y aparecerá al final de todos los bucles infinitos a lo largo de los tutoriales.

Si cargamos el fichero generado por el "bat", con la extensión "nds", obtendremos una simulación de una Nintendo DS, donde en la pantalla inferior aparece el texto "Hello World!".

4.4 Texto en pantalla

En el tutorial anterior, hemos aprendido cómo escribir un texto en pantalla, pero este texto era de "debug". Para escribir texto normal será necesario algo más de trabajo.

Para empezar con las fuentes, debemos tener la fuente que se refleja en la siguiente imagen. Ésta se encuentra en la carpeta "nitrofiles/fnt" de este mismo proyecto.

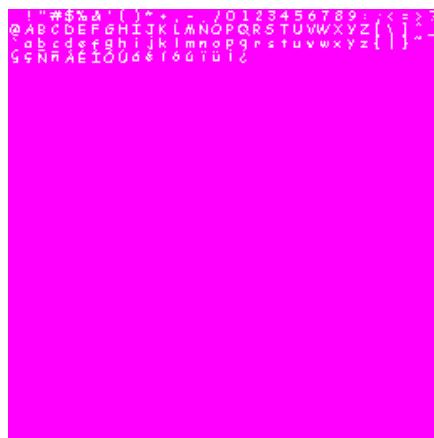


Figura 4.02: Ejemplo de fuente

Para convertir esta imagen con formato "bmp" a la imagen con el formato que reconoce la consola, se ha usado el programa grit, mencionado en el tutorial 4.2, y más concretamente el ejecutable "fontConverter.bat".

Antes de comenzar a posicionar el texto, debemos iniciar el motor inicializando los buffers temporales para "Backgrounds", ya que este mismo buffer es el que se usa en los textos.

```
// Inicializa los buffers para almacenar fondos
NF_InitTiledBgBuffers();
// Inicializa los fondos para la pantalla superior
NF_InitTiledBgSys(0);
// para la pantalla inferior
NF_InitTiledBgSys(1);
```

Una vez inicializados los buffers, comenzamos con las inicializaciones propias de los textos.

```
NF_InitTextSys(0); // Inicializa el texto
```

A continuación tenemos que cargar la fuente que se usará, ubicada en la carpeta "nflib/fnt/nombreFuente". Ésta es 'default' en nuestro caso.

```
// fuente | nombre que se le dara a la fuente | ancho | alto | rotacion (0 ninguna, 1 derecha, 2 izquierda)
NF_LoadTextFont("fnt/default", "normal", 256, 256, 0);
```

El texto se sitúa en diferentes capas (layers). De este modo se pueden hacer cambios a todos los textos de la misma capa de manera simultánea. Al crear la capa se le asignará un nombre, el cual será su identificador.

```
// 0,1 pantalla | id de capa | rotacion (0 ninguna, 1 derecha, 2 izquierda) | nombre de la fuente
NF_CreateTextLayer(0, 0, 0, "normal");
```

Después de todas las inicializaciones, la carga de la fuente y de la capa, ya estamos listos para

poder publicar texto en pantalla.

```
// pantalla (0,1) | id de capa | posicion X | posicion Y | texto  
NF_WriteText(0, 0, 1, 1, "Hola Mundo!\nHello World!");
```

Llegados a este punto ya le hemos dicho al buffer que muestre un texto por pantalla. Pero para que este se muestre hay que transmitir la información del buffer a la pantalla. Para esto tan solo falta decirle a la memoria gráfica que lo muestre.

```
[ NF_UpdateTextLayers();
```

Importante mencionar que las inicializaciones, la carga de la fuente y el layer tan solo es necesario hacerlo una vez en caso de que no se descargue de la memoria. Por lo que con una fuente y capa podemos escribir múltiples textos.

4.4.1 Colores de fuentes

Ya tenemos texto escrito en la pantalla, pero todo el texto generado es blanco. Cambiar el color de éste es muy sencillo.

Una vez creada la fuente y su capa correspondiente, podemos asignarle un color a la capa.

Primero hemos de crear un color.

```
// pantalla | layer (0 - 3)| id de color (0-15) | color rojo (0-31) | color verde (0-31) | color azul (0-31)  
NF_DefineTextColor(1, 0, 1, 31, 0, 0);
```

Una vez definido el color, ya podemos asignarlo a la capa.

```
[ NF_SetTextColor(1, 0, 1); // pantalla | layer | color
```

4.4.2 Fuentes personalizadas

Para crear una nueva fuente, obtendremos la fuente de ejemplo que hay en el tutorial de fuentes (Tutorial 4.4.0.) y lo cambiaremos según convenga.

Una fuente está distribuida en una cuadrícula de 8 píxeles de alto y ancho por cada letra. Una vez modificada la fuente, hay que recurrir al programa "grit" (introducido en el capítulo 4.2) y convertirla en una fuente válida.

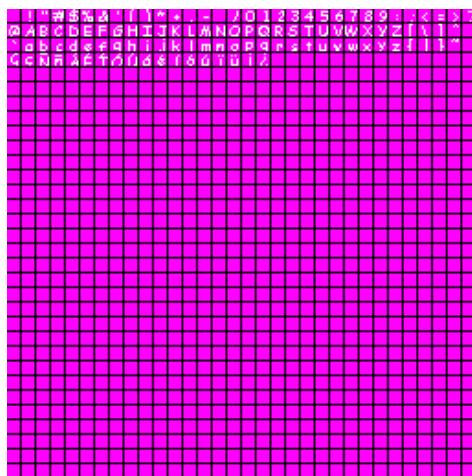


Figura 4.03: Fuente personalizada

El programa está situado en la carpeta “nflib/tools/grit”. En esta ubicación hay numerosos archivos ".bat" y multitud de carpetas. Entre todas éstas hay una carpeta llamada "fonts" y otra llamada "bmp" y un ".bat" llamado "Convert_Fonts".

Una vez se tenga modificado es necesario copiar el archivo de fuente (con extensión bmp) a la carpeta bmp del directorio grit antes mencionado. Ejecutamos el bat "Convert_Fonts" e introducimos el nombre que queremos que tenga la fuente. Una vez hecho esto nos aparecerá en la carpeta “fonts” el archivo ".fnt" y en la carpeta “sprites” dos archivos más, uno con la extensión ".pal" y otro con la extensión ".img".

Una vez que tengamos los archivos necesarios, lo copiamos a la carpeta "nitrofiles\fnt" de nuestro proyecto y lo usamos del mismo modo que hicimos en el tutorial 4.4.

4.5. Imágenes

En esta serie de tutoriales aprenderemos a cargar imágenes, parte fundamental para cualquier juego. Primero se abarcará la carga de fondos de pantalla y luego se enseñará a cargar otras imágenes (sprites).

Para la carga de imágenes, tanto fondos como sprites, volveremos a usar la aplicación grit, mencionada en el tutorial 4.2.

Con las imágenes hay que tener ciertas consideraciones en cuenta. Para empezar, es importante mencionar que el primer píxel de todas las imágenes indica cuál es el color asignado como transparente. Usaremos como norma general el color magenta (255,0,255).

La extensión con la que trabaja grit es ".bmp", puesto que no admite transparencias. Por lo tanto se recomienda trabajar con ".bmp" directamente o, en su defecto, con ".png" y luego pasarlo a ".bmp".

4.5.1. Fondos

Los fondos son las imágenes que se sitúan detrás de todo lo demás y suelen ocupar toda la pantalla.

Poseemos tres tipos de fondo a nuestro alcance: simples, complejos o los denominados affine. La principal diferencia visual que tienen es que los simples son de 256 colores, los complejos de 8 o 16 bits y los affines tienen las propiedades de rotarse y cambiar de tamaño.

4.5.1.1. Fondos Simples

En este capítulo describimos los fondos más simples que usaremos. Estos fondos tienen 256 colores por lo que puede ser la mejor opción en cuanto a memoria se refiere. Estos fondos no pueden ser girados ni rotados. Hemos realizado dos tutoriales para ilustrar su uso. La diferencia entre ambos tutoriales es que el primero carga un fondo simple y en el segundo se le adjudica a éste un movimiento cuando se pulse las teclas (adelantándonos al tutorial 4.6. Inputs)

Para cargar las imágenes, nuevamente se usará el programa grit. Se ejecutará el ".bat" de "Convert_Backgrounds" en este caso.

En el primer tutorial, para empezar hay que indicarle a la consola que usaremos gráficos 2d, y el modo de color que tendrá.

```
// pantalla | Modo (0 tileados 256 colores; 2 affine til. 8 bits en capa 2 y 3; 5 fondos bitmaps de 8 o 16 bits)
NF_Set2D(0, 0);
```

De este modo preparamos el motor para recibir fondos simples. Por lo que ahora cuando iniciemos los buffers y el sistema de fondos el sistema sabrá cómo ha de hacerlo. Estas inicializaciones son las mismas que las usadas en el tutorial 4.4. de textos en pantalla.

```
NF_InitTiledBgBuffers();           // Inicializa los buffers para almacenar fondos
NF_InitTiledBgSys(0);             // Inicializa los fondos Tileados para la pantalla superior
NF_InitTiledBgSys(1);             // Inicializa los fondos Tileados para la pantalla inferior
```

Con esto ya están inicializados los buffers, ahora hay que cargar la imagen de fondo en la memoria RAM.

```
NF_LoadTiledBg("bg/layer3", "moon", 256, 256);    // nombre de archivo | identificador | ancho | alto.
```

Ahora que ya esta en memoria RAM, el siguiente paso es pasarlo a pantalla.

```
NF_CreateTiledBg(0, 3, "moon"); // pantalla | id capa (0 - 3) | nombre del fondo.
```

Ya con esto tendremos un fondo estático cargado en la pantalla.

En el segundo tutorial se hace uso de una función especial para realizar el movimiento de un

fondo.

```
NF_ScrollBg(1, 1, pos_x, pos_y); // pantalla | id capa | posición del fondo en X | posición del fondo en Y
```

4.5.1.2. Fondos Complejos

Los fondos complejos son iguales que los simples, con la diferencia de que éstos tienen mayor calidad gráfica. En la librería NFLib hay dos tipos de fondos complejos, de 8 bits y de 16 bits. Las funciones para ambos tipos son exactamente iguales, tan solo es necesario cambiar el número 16 en el método por un 8. En nuestro tutorial se ha usado un fondo de 16 bits.

Para cargar las imágenes, nuevamente se usará el programa grit, salvo que esta vez se ejecutará el bat de "Convert_bitmap16" o "Convert_bitmap8", según convenga.

Para empezar, hemos de iniciar el 2d en modo 5, modo 8 o 16 bits

```
NF_Set2D(0, 5); // Modo 2D_5 en ambas pantallas  
NF_Set2D(1, 5);
```

A continuación iniciamos el modo bitmap para ambas pantallas.

```
// pantalla || modo ( 0, 8 bits; 1, 16 bits).  
NF_InitBitmapBgSys(0, 1);  
NF_InitBitmapBgSys(1, 1);
```

Inicializamos los buffers tanto para la pantalla inferior.

```
NF_Init16bitsBgBuffers();
```

En la pantalla superior vamos a hacer uso de un segundo buffer, que será copia del primero. Esto nos dará mucho más control sobre el fondo y nos permitirá modificarlo en el caso que quisiésemos hacerlo. Por lo que lo iniciamos y lo activamos

```
NF_Init16bitsBackBuffer(1);  
NF_Enable16bitsBackBuffer(1);
```

Ahora que ya está todo inicializado, procedemos con la carga en memoria RAM de la imagen.

```
NF_Load16bitsBg("bmp/bitmap16", 0); // nombre imagen || slot (0-15)
```

Ya tan solo queda transferir de la memoria RAM a la memoria gráfica VRAM y automáticamente a la pantalla.

```
NF_Copy16bitsBuffer(0, 0, 0); // pantalla | destino(0, VRAM; 1, backbuffer)|slot a cargar
```

Para la pantalla inferior se ha cargado en la VRAM, y por lo tanto en la pantalla. Pero en la

pantalla superior, se cargará la misma imagen en memoria en el backbuffer previamente inicializado.

```
| NF_Copy16bitsBuffer(1, 1, 0);
```

En la pantalla superior no aparecerá nada por pantalla, ya que la imagen está cargada en el backbuffer y no en VRAM directamente.

Existe una variable global para acceder al backbuffer, esta es `NF_16BITS_BACKBUFFER[][]`, donde el primer parámetro de la matriz es la pantalla y el segundo es un float con todas las posiciones de pantalla, donde los 8 primeros bits especifican la posición "y" en la pantalla y los 8 últimos la posición "x". Esta matriz almacena colores en el formato "Alpha + RGB555".

Esto que puede resultar confuso, da mucha potencia ya que tenemos acceso a toda la imagen cargada en el back buffer para poder modificarla a nuestro antojo.

Una vez realizado los cambios deseados, para traer el backbuffer a la memoria gráfica, y por lo tanto a la pantalla, se usa la instrucción:

```
| NF_Flip16bitsBackBuffer(1);
```

4.5.1.3. Affine

Los fondos affine son unos fondos especiales, los cuales a diferencia de los fondos simples y los fondos complejos, pueden ser escalados y rotados. Estos son especialmente útiles para llevar a cabo efectos de transición entre estados del juego.

Para convertir las imágenes affine usaremos de nuevo el grit, pero esta vez el ".bat" nombrado como "Convert_Affine". Estas imágenes tienen que tener un tamaño de 256x256 o 512x512 en su defecto.

Se empieza cargando el 2D en su modo concreto. En este caso hay que iniciar el 2D en el modo affine.

```
| NF_Set2D(0, 2);           // Modo 2D_2 en la pantalla superior  
| NF_Set2D(1, 2);           // Modo 2D_2 en la pantalla inferior
```

Luego iniciamos los buffers de affine para ambas pantallas.

```
| NF_InitTiledBgBuffers();      // Inicializa los buffers para almacenar fondos  
| NF_InitAffineBgSys(0);        // Inicializa los fondos affine para la pantalla superior  
| NF_InitAffineBgSys(1);        // Iniciaiza los fondos affine para la pantalla inferior
```

Una vez todo inicializado, cargamos las imágenes en memoria RAM.

```

// nombre imagen | nombre asignado | ancho | alto
NF_LoadAffineBg("bg/waves512", "waves", 512, 512);
NF_LoadAffineBg("bg/navygrid", "grid", 256, 256);
NF_LoadAffineBg("bg/flag512", "flag", 512, 512);

```

Ahora que ya están en memoria RAM, las cargamos en la VRAM.

```

// pantalla | capa (2-3) | nombre a cargar | infinito (1, si; 0, no)
NF_CreateAffineBg(0, 3, "waves", 1);
NF_CreateAffineBg(0, 2, "grid", 0);
NF_CreateAffineBg(1, 3, "flag", 1);

```

Ya con esto es suficiente para que se muestre ambos fondos, pero en caso de que el fondo no fuera a ser movido o escalado en algún momento, pierde un poco el sentido haber usado este tipo de fondo.

Así que se ha hecho una lógica con los botones de la consola para el escalado, movimiento y el zoom del fondo.

Hay tres transformaciones que podemos hacer con los fondos affines:

1. Modificando la matriz de transformación, es decir, escalado e inclinación de los ejes (rotación).

```

// pantalla | capa (2-3) | escala X (0-512) | escala Y (0-512) | inclinación X (0-512) | inclinación Y (0-512)
NF_AffineBgTransform(n, 3, zoom, zoom, 0, 0);

```

2. Cambio del centro o pivote, se usa para que el objeto rote diferente.

```

NF_AffineBgCenter(n, 3, x, y);      // pantalla | capa(2-3) | centro X (0-ancho) | centro Y (0-alto)

```

3. Movimiento y rotación.

```

// pantalla | capa | posición X (-2048 a 2048) | posición Y | ángulo rotación (-2048 a 2048)
NF_AffineBgMove(n, 3, 0, 0, angle);

```

4.5.2. Imágenes (Sprites)

Las imágenes, gráficos o sprites son el pilar de todo juego 2D. En “NightFox Lib” hay diferentes tipos de imágenes en función de sus propiedades, simples, complejas o 3D.

Del mismo modo que ocurría con los fondos, las simples son gráficos de 256 colores y un tamaño máximo de 128x128 y las complejas son de 8 o 16 bits y mayor tamaño. Las complejas tienen soporte en la librería en la que esta basada NFLib, libnds, pero NFLib no le ha dado soporte debido a que estas imágenes suelen ocupar mucha memoria y no se puede realizar juegos complejos. Debido a que NFLib no le da soporte, nosotros nos centraremos en las simples

y en las 3D.

Los Sprites simples tienen un tamaño limitado a 128x128 y en el caso que queramos cargar imágenes de mayor tamaño, se usa las imágenes 3D. Las imágenes 3D no son más que un polígono plano (2 triángulos) con la textura del sprite sobre ella, de este modo se le da soporte para imágenes de 256x256.

Debido a que a las imágenes simples poseen muchas funciones para su trato, se ha dividido los tutoriales en dos partes, con y sin animaciones.

4.5.2.1. Imágenes simples

Aquí hablamos sobre las imágenes simples. Para este tutorial se ha usado un solo sprite, una pelota, la cual rebota por los bordes de la pantalla superior. Se ha prescindido de fondos o inputs para simplificar lo importante de este tutorial, los sprites.

La imágenes traen consigo algo llamado paletas. Éstas poseen la información de los colores de una imagen dada. De este modo, para cambiar el color de un sprite basta con cambiar su paleta de colores.

Para convertir las imágenes se usa, una vez más, el programa grit. Pero esta vez el ".bat" "Convert_Sprites". Este ejecutable generará múltiples archivos de imágenes y una sola paleta (con extensión ".pal"). El ancho y el alto debe de ser múltiplo de dos, y un tamaño máximo de 128 y un tamaño mínimo de 8 píxeles.

Lo primero de todo es iniciar el 2D.

```
// Modo 2D_0 en ambas pantallas  
NF_Set2D(0, 0);  
NF_Set2D(1, 0);
```

Luego hay que inicializar los buffers.

```
NF_InitSpriteBuffers();           // Inicializa los buffers para almacenar sprites y paletas  
NF_InitSpriteSys(0);             // Inicializa los sprites para la pantalla superior  
NF_InitSpriteSys(1);             // Inicializa los sprites para la pantalla inferior
```

Se debe cargar todas las imágenes y sus paletas a memoria RAM.

```
NF_LoadSpriteGfx("sprite/bola", 1, 32, 32); // nombre fichero | id en RAM (0-255) | ancho | alto.  
NF_LoadSpritePal("sprite/bola", 1);          // nombre paleta | id en RAM (0 - 63)
```

Nuevamente está todo cargado en memoria RAM, pero si queremos que se muestre hay que pasarlo a la memoria gráfica VRAM.

```
NF_VramSpriteGfx(0, 1, 0, false); // pantalla | id RAM | id VRAM | true todo en memoria, false no.  
NF_VramSpritePal(0, 1, 0); // pantalla | id en RAM | id en VRAM
```

Ahora ya tenemos todo cargado en memoria gráfica VRAM. A diferencia de los fondos, al cargarla en VRAM no se carga directamente en la pantalla, hay que especificarlo.

```
// pantalla | id de gráfico | id gráfico en VRAM | id paleta en VRAM | posición NF_CreateSprite(0, 0, 0, 0,  
bola_x, bola_y);
```

Con esto, tendremos un sprite estático en la pantalla. Hay que usar una instrucción para moverlo.

```
NF_MoveSprite(0, 0, bola_x, bola_y); // pantalla | id de gráfico | nueva posición X | nueva posición Y
```

La librería trabaja con un sistema de buffers temporales, por lo que hay que actualizar la pantalla volcando estos buffers. Una vez volcado hay que indicarlo. Como consecuencia, hay que añadir antes y después del sincronismo vertical estas funciones.

```
NF_SpriteOamSet(0); // Actualiza el buffer  
swiWaitForVBlank(); // Espera al sincronismo vertical  
oamUpdate(&oamMain); // Actualiza el buffer
```

En el tutorial también hemos añadido una lógica muy simple para que la pelota se mueva rebotando por la pantalla.

4.5.2.2. Imágenes Animadas

Este tutorial toma el tutorial anterior como referencia, es decir, una vez llegado aquí, ya sabemos cómo cargar un sprite en la pantalla.

Para generar el gráfico con múltiples sprites, éste ha de ser creado con los frames situados en la vertical. Como se puede apreciar, esta imagen tiene cada uno de sus estados de animación, frames, situados cada uno bajo el anterior, creando así una secuencia. De este modo es como funciona el motor.



Figura 4.04: Ejemplo imagen de animación

La conversión mediante el programa grit, se hace de forma similar a un sprite sin animaciones.

A la hora de cargar el gráfico en memoria, solo especificamos el tamaño de un frame, y no del tamaño real del archivo.

```
NF_LoadSpriteGfx("sprite/personaje", 0, 64, 128);
```

Cuando queremos cargarlo en VRAM, hay dos modos de hacerlo, cargando todos los frames en la memoria gráfica o no. La principal diferencia es que en el momento de cambiar entre frame puede sufrir un mínimo retraso.

```
NF_VramSpriteGfx(1, 0, 0, true);
```

Para gestionar las animaciones solo hay que indicar en que frame de la animación se situará el gráfico.

```
NF_SpriteFrame(1, 0, pj_frame); // pantalla || id gráfico (VRAM) || frame
```

Como añadido, para cuando a un mismo gráfico queremos hacerle un volteado horizontal.

```
NF_HflipSprite(1, 0, false); q // pantalla || id gráfico (vram) || volteado
```

También existe la función de volteado vertical. Es similar a la anterior salvo que sustituimos la H por una V.

Este ejemplo posee casi todo lo necesario para un buen comienzo de un juego. Este tutorial contiene dos fondos, dos sprites, y algo de lógica para gestionar las animaciones en la pantalla.

4.5.2.3. Imágenes 3D

Para terminar con la carga de imágenes, tenemos las imágenes 3D. La principal diferencia es que éstas permiten mayores dimensiones en los sprites. Realmente se crea un plano 3D y se pinta la imagen sobre ella.

El modo de conversión del gráfico es exactamente igual que en el tutorial 4.5.2.1, Imágenes simples.

El modo de inicializar el motor es diferente ahora.

```
NF_Set3D(0, 0); // Modo 3D_0 en la pantalla superior  
NF_Set2D(1, 0); // Modo 2D_0 en la pantalla inferior
```

También para los buffers es distinto.

```
NF_Init3dSpriteSys();
```

Debido a que solo imprime la imagen en el plano, el modo de carga de imágenes y paletas en RAM es exactamente igual que en los gráficos simples

```
NF_LoadSpriteGfx("sprite/numbers", 0, 16, 16);  
NF_LoadSpritePal("sprite/numbers", 0);
```

Para añadirlos a la VRAM sí cambian los métodos

```
NF_Vram3dSpriteGfx(0, 0, true);  
NF_Vram3dSpritePal(0, 0);
```

Y ya para terminar y cargarlos en la pantalla.

```
NF_Create3dSprite(0, 0, 0, x, y);
```

Al igual que los sprites que tenían métodos para actualizar los buffers, los 3D no van a ser menos

```
NF_Draw3dSprites();  
glFlush(0);  
swiWaitForVBlank();  
NF_Update3dSpritesGfx();
```

Ya con esto tenemos el gráfico cargado en la pantalla.

Al igual que con los sprites simples, estos gráficos pueden moverse por la pantalla.

```
NF_Move3dSprite(0, x, y);
```

También pueden animarse

```
NF_Set3dSpriteFrame(0, frame);
```

4.6. Input

En la consola hay que diferenciar dos tipos de input, los botones y la pantalla táctil.

Botones.

Lo primero a realizar es pedirle a la consola que actualice los buffers de input.

```
scanKeys(); // permite capturar refrescar el buffer interno de teclas
```

Ahora dependiendo de lo que queramos, se ha de usar unos métodos u otros.

```
U16 keys = keysDown(); // captura en key las nuevas pulsaciones, ignorando mantenidas  
U16 keys = keysHeld(); // captura en key los botones mantenidos  
U16 keys = keysUp(); // captura cuando se suelta un boton.
```

La información sobre los inputs se guarda en una variable de 16 bits (u16) en modo bits. La librería trae constantes con los valores posibles.

```
KEY_A, KEY_B, KEY_SELECT, KEY_START, KEY_RIGHT, KEY_LEFT, KEY_UP, KEY_DOWN, KEY_R, KEY_L, KEY_X,  
KEY_Y, KEY_TOUCH
```

Por lo que si queremos comprobar si se ha pulsado la A, después de obtener el input actual se ha de hacer una comprobación tipo "and" lógico con la constante.

Del siguiente modo comprobamos si se ha pulsado arriba en el pad.

```
if( keys & KEY_UP){}
```

Pantalla Táctil

Ya con esto tenemos capturados los botones, detectar la pantalla táctil es ligeramente diferente.

```
touchPosition touchscreen;  
// modificamos los valores de touchscreen  
touchRead(&touchscreen);
```

Mediante "touchscreen.px" y "touchscreen.py" obtenemos las coordenadas "x" e "y" de la pantalla donde hemos pulsado. Si se usa esta función después de "keysDown" solo detectará el primer clic, y si lo hacemos después de "keysHeld" detectará el movimiento continuo.

Con esto ya tenemos el control sobre las pulsaciones de la consola.

4.7. Sonidos

Los sonidos pueden no ser una parte fundamental de un juego, pero no por ello hay que restarles importancia.

El formato para la NDS es el ".raw". NFLib no nos facilita una herramienta para su conversión a este formato, por lo que hemos usado la aplicación "Audacity" [Audacity, 2015]. Todos los audios serán ubicados en la carpeta "nitrofiles/sfx".

Para su uso lo primero es habilitar el sonido.

```
[ soundEnable(); // Inicializa el engine de audio de la DS ]
```

A continuación hay que inicializar los buffers.

```
[ NF_InitRawSoundBuffers(); // Inicializa los buffers de sonido ]
```

Una vez inicializado todo, ya podemos proceder a la carga del audio.

```
[ // nombre, id memoria, frecuencia, bits usados ( 0 – 8 bits, 1 – 16 bits)  
NF_LoadRawSound("sfx/sample", 0, 11025, 0); ]
```

Para descargarlo usar el "NF_UnloadRawSound(id)".

Ya estamos listo para reproducir sonidos. Este método devuelve un id de sonido, el cual usaremos posteriormente para silenciarlo o continuarlo.

```
[ // id memoria, volume ( 0 – 127), balance ( 0/64/127), loop ( true/false), punto donde empieza loop.  
sound_id = NF_PlayRawSound(1, 127, 64, true, 0); ]
```

Ya con esto tenemos audios sonando. Éstos pueden pausar o continuar con la reproducción en cualquier momento.

```
[ soundPause(sound_id); // id sonido  
soundResume(sound_id); // id sonido ]
```

4.8. Colisiones

Todo juego, tarde o temprano, necesita un sistema de colisiones. NFLib incluye un sistema que consiste en un fondo especial, que contendrá las zonas de colisión, de este modo podremos movernos por ciertas zonas del fondo detectando peculiaridades en este fondo. Este sistema es especialmente útil en juegos con vista cenital.

Para complementar a este sistema, crearemos nuestro propio sistema de colisiones simple

basado en rectángulos.

4.8.1. Mapa Colisiones

La librería NFlib, trae un sistema propio para hacer colisiones mediante un fondo especial, del cual se leerá la información del color. Dependiendo del color leído, se llevará a cabo una acción u otra.

El mapa de colisiones ha de ser convertido también con el programa grit, esta vez con el ejecutable "Convert_CMaps.bat".

El tutorial, consiste en unas pelotas que caen por una pasarela. Estas imágenes son la comparación entre el mapa de colisión que usamos (situado a la izquierda) y el mapa que se muestra (situado a la derecha).

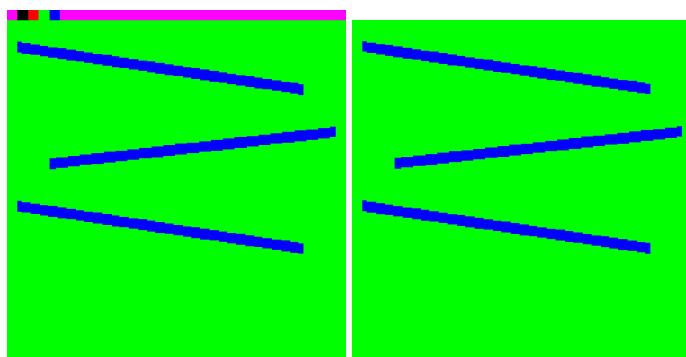


Figura 4.05: Mapa de colisiones

Todo es exactamente igual que con los fondos, tan solo hay que añadir ciertas funciones para el trato con el fondo de colisiones.

Para empezar hay que añadir la inicialización del mapa de colisiones a continuación de los fondos y los sprites.

```
-----  
NF_InitTiledBgBuffers();           // Inicializa los buffers para almacenar fondos  
NF_InitTiledBgSys(0);             // Inicializa los fondos Tileados para la pantalla superior  
NF_InitSpriteBuffers();           // Inicializa los buffers para almacenar sprites y paletas  
NF_InitSpriteSys(0);              // Inicializa los sprites para la pantalla superior  
NF_InitCmapBuffers();            // Inicializa los buffers de mapas de colisiones  
-----
```

Ya con esto tenemos el sistema inicializado, también para las colisiones. A la hora de cambiar los fondos tenemos que añadir el mapa de colisiones.

```
-----  
NF_LoadTiledBg("bg/pdemo_bg", "bg3", 256, 256); // Carga el fondo  
NF_LoadSpriteGfx("sprite/whiteball", 0, 16, 16);    // Pelota  
NF_LoadSpritePal("sprite/whitepal", 0);  
// Carga el fondo de colisiones. Nombre, capa, ancho, alto.  
NF_LoadColisionBg("maps/pdemo_colmap", 0, 256, 256);  
-----
```

Ya una vez tenemos todo listo, podemos leer los gráficos del fondo, y en función del píxel leído hacer una cosa u otra. En nuestro ejemplo se ha usado este método para indicar si la pelota ha de descender o avanzar.

```
// capa de fondo, coordenada x, coordenada y  
u8 color = NF_GetPoint(0, x, y)
```

Esta función devuelve el color leído. En el código se hace uso de 3 pelotas, los cuales descenderán por las pasarelas. Se crean.

```
for (b = 0; b < 3; b++) {  
    NF_CreateSprite(0, b, 0, 0, -16, -16);  
    NF_SpriteLayer(0, b, 3);
```

Luego se le indican su posición inicial, en coordenadas x e y

```
s16 x[3];  
s16 y[3];  
x[0] = 32;  
y[0] = -16;  
x[1] = 228;  
y[1] = 32;  
x[2] = 10;  
y[2] = 100;
```

Y para terminar, se indica por cada uno de los 16 píxeles de ancho de cada pelota, su píxel de colisión de la variable coordenada "y". Esta hecho un poco "a lo bruto" pero ilustra el comportamiento deseado.

```
s16 py[16];  
py[0] = 11;  
py[1] = 13;  
py[2] = 14;  
py[3] = 15;  
py[4] = 15;  
py[5] = 16;  
py[6] = 16;  
py[7] = 16;  
py[8] = 16;  
py[9] = 16;  
py[10] = 16;  
py[11] = 15;  
py[12] = 15;  
py[13] = 14;  
py[14] = 13;  
py[15] = 11;
```

Una vez que tenemos todo esto, por cada pelota obtenemos el píxel que esta tocando el

centro, añadiéndole el desplazamiento de "y". Con esto sabemos si esta cayendo o no. El color 4, corresponde al color azul.

```
for (n = 0; n < 16; n++) {
    if (NF_GetPoint(0, (x[b] + n), (y[b] + py[n]))) == 4)
        down = false;
}
```

Luego comprobamos si va hacia la izquierda o hacia la derecha.

```
// Caida hacia la izquierda
if (NF_GetPoint(0, (x[b] - 1), (y[b] + 16)) == 4) left = false;
// Caida hacia la derecha
if (NF_GetPoint(0, (x[b] + 16), (y[b] + 16)) == 4) right = false;
// Si hay caida libre, no te muevas en horizontal
if (left && right) {
    right = false;
    left = false;
}
```

Y ya para terminar movemos la pelota.

```
NF_MoveSprite(0, b, x[b], y[b]); // Posicion del Sprite
```

4.8.2. Colisiones Manuales

Aparte de la colisiones del mapa de NFlib, ésta no dispone de ningún otro sistema de colisiones. Debido a esto, hemos creado nuestro propio sistema de colisiones basado en rectángulos.

Este tutorial está basado en el tutorial “4.5.2.1 Sprites simples” y se le ha añadido otra pelota y una colision basada en cuadrados.

Para gestionar si un cuadrado colisiona con otro, tan solo hay que realizar la comprobación de sus extremos.

```
if(
    (bola_x[i] <= bola_x[otra_bola] + bola_w[otra_bola]) &&
    (bola_x[i]+bola_w[i] >= bola_x[otra_bola]) &&
    (bola_y[i] <= bola_y[otra_bola] + bola_h[otra_bola]) &&
    (bola_y[i]+bola_h[i] >= bola_y[otra_bola])){
```

De este modo tenemos la comprobación realizada. Mostramos en la imagen algunos ejemplos para ilustrar que está funcionando. El pivote del cuadrado está en la esquina inferior izquierda.

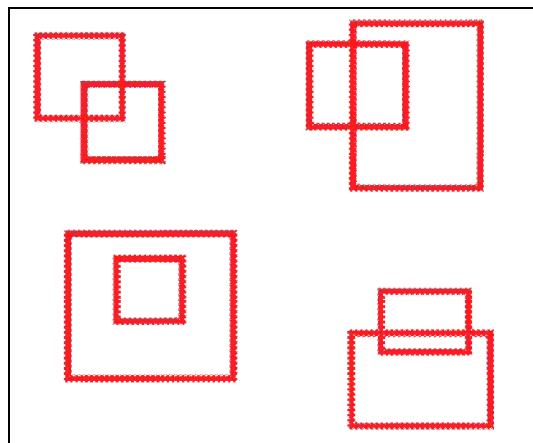


Figura 4.06: Pruebas de colisiones

Éste es el sistema que hemos decidido realizar, se puede usar muchos otros sistemas de colisiones, siempre que se programen desde cero.

4.9. Wifi

Aprovechar la tecnología "Wifi" de la consola no es sencillo. Éste es uno de los tutoriales más complejos antes de entrar en materia con el último tutorial. Aquí solo explicaremos algunas funciones, se recomienda consultar del código para una mayor comprensión de funcionamiento.

A parte de las librerías de siempre, hay que incluir otra librería

```
#include <nf_wifi.h>
```

Para hacerlo más sencillo, se han creado funciones en el código. En la función "_ConnectWIFI" conectamos a la conexión WIFI previamente configurado con el software de Nintendo DS. Esto lo intentaremos cada frame hasta que se conecte.

```
if (NF_WiFiConnectDefaultAp())
    return true;}
```

El siguiente paso es elegir si somos cliente o servidor. Si somos cliente nos conectamos al puerto especificado.

```
NF_WIFI_CreateUdpListener(myport); // puerto donde escuchar
```

Y si somos servidor creamos el puerto de comunicación.

```
NF_WIFI_CreateUdpSender(server_ip, myport); // char* con la IP || puerto a usar
```

Para escuchar usaremos

```
status = NF_WIFI_UdpListen(300000); // time out de espera
```

Para enviar información usaremos

```
NF_WIFI_UdpSend(buffer); // char * cadena a enviar
```

Una vez hayamos terminado, cerramos el socket y desconectamos el WIFI

```
close(NF_SOCKET);
NF_WiFiDisconnectAp();
```


Capítulo 5. Diseño

En este capítulo se explicará en qué consiste un documento de diseño y su vital importancia y se realizarán un documento de concepto y un documento de diseño sobre el juego que desarrollaremos en el capítulo 6.

5.1. Diseño

Cuando se habla de diseño de videojuegos, en lo primero que se suele pensar es en los gráficos y el arte en general de un juego, pero esto no es del todo correcto. El diseño de un videojuego consiste en planificar cómo será el juego, incluyendo en este plan todo el juego al completo y todo lo que lo conforma: gráficos, música y mecánicas.

El problema acerca del diseño es que es un arte difícil de calificar a priori, por ejemplo, para saber si alguien realiza una acción mejor que otro en otras materias, tan solo hay que ver los resultados y comparar, pero con el diseño de videojuegos no es tan fácil ya que a veces los resultados no reflejan la calidad del diseño.

Existen múltiples libros de referencia que intentan explicar en qué consiste el diseño de un videojuego y el enfoque necesario para llevarlo a cabo. Algunos libros de interés son: "Art of game design: A book of lenses" [Art Game Design, 2014] y "Theory of Fun for Game Design" [Theory of Fun, 2013]. Hay que tener cuidado con ciertos libros, ya que algunos solo consiguen confundir aún más.

El diseño de un juego suele acabar plasmado en un documento, de texto normalmente, donde se detallan todas las características del juego de modo que si alguien completamente desconocido leyese dicho documento, éste puede hacerse una idea con bastante detalle sobre en qué consiste el juego. Suele hacerse un documento de unas dos páginas, recomendablemente, para presentar el juego, es decir, este sería el documento que le darías a alguien que quieras convencer de que el juego puede estar bien. Una vez que estás decidido a hacer el juego se suele realizar un documento especificando los detalles del título punto por punto siendo este segundo documento es mucho más extenso.

Existen múltiples maneras de enfocar un diseño pero particularmente distinguimos dos bien diferenciadas. La primera y el modo tradicional es tener una idea de una mecánica o varias, por ejemplo un juego de lucha, un juego de coches, etc. y luego construir el resto alrededor de esta mecánica o tipo de juego, dando los detalles de éste y otorgándole personalidad propia. El otro modo es olvidarnos al completo de mecánicas o tipos de juego y pensar en sensaciones, ¿qué queremos trasmitir al jugador? y una vez que tengamos la respuesta a esta pregunta, ya pensar en el género y mecánica se adecua más a ésta. Por supuesto, ésto dos métodos no son definitivos y hay mil y un maneras de tener una idea para un juego. Lo que sí es cierto y siempre

se cumple es que una vez que tengamos mecánica, género o sensación a trasmitir, hemos de preguntarnos acerca del juego, ¿cómo se juega? ¿qué características tiene nuestro personaje? ¿qué tipos de enemigos existen? "¿por qué mola?" (citando a Juan De Miguel Contreras, diseñador español y profesor del Master en diseño de videojuegos en la universidad complutense de Madrid). Éstas son solo algunas pocas cuestiones que debemos realizarnos a la hora de pensar en un juego.

Hoy en día es muy difícil crear algo completamente original y todo está referenciado por algún otro juego o producto. Esto no es malo, ya que si se desea mejorar algo o trasmitir la misma sensación que se trasmítia con cierto título, qué mejor que basarnos en él y otorgarle personalidad propia. Aun así cada año surgen ideas nuevas que dan una vuelta de tuerca a algo que ya existía, quizás sea usando algo ya existente de un modo que nunca se había usado o ya sea llevando a otra dimensión algo existente dándole un lavado de cara.

Antes comentaba que es difícil comparar el diseño de dos juegos, pero esto no es del todo cierto ya que hay ciertas pautas que se deben evitar, a continuación indico algunas pautas. No por tener más mecánicas, será mejor juego, un buen juego es aquél que funciona con unas pocas mecánicas pero exprimidas al máximo (Ejemplo "Super Mario Bros"). El juego ha de tener una duración determinada, no por ser un juego más largo será un mejor juego (Ejemplo "Portal"). Se debe ser justo con el jugador, esto no significa que deba ser fácil (Ejemplo "Demon Soul").

Hemos dicho que el diseño es pensar el juego al completo, plasmarlo y luego programarlo. Esto no siempre es así, de hecho la mayoría de las veces no lo es y el diseño sufre cambios en medio del desarrollo. Un perfecto diseñador tendría todo pensado, a falta de modificar longitudes, cantidad de daño, vidas, etc. Aunque esto entra dentro de lo normal, hay ciertas decisiones que deben hacerse antes que otras si éstas condicionan el resto del desarrollo. Por ejemplo, imagina un plataformas y el personaje salta, decidimos la distancia de salto del personaje y gracias a este salto diseñamos los niveles. Después de dos meses realizando niveles resulta que el diseñador piensa que es mejor que el personaje tenga doble salto fastidiando así todos los saltos peligrosos y las zonas donde el personaje no alcanzaba en los niveles ya creados, teniendo que rediseñarlos todos de nuevo. El ejemplo anterior ilustra un mal diseño, ya que si se preveía esa posibilidad tendría que haberse pensado casi lo primero para aceptarla o descartarla pronto. Tenerlo todo decidido es muy idealista. Supongamos que en un proceso avanzado del desarrollo comenzamos con las demos públicas, y apreciamos que el salto simple no gusta al público objetivo al cual queríamos venderle nuestro producto, pues en este caso quizás sí haya que rediseñar y añadir el doble salto aunque con ello se pierdan meses de trabajo.

Por supuesto hay juegos que requieren más diseño en un aspecto y otros que lo requieren en otro. Por ejemplo, el tetris tiene un diseño muy diferente del portal, ya que aunque en el portal

lo más importante son los niveles, en el tetris lo más importante fue la decisión de las piezas a incluir. A raíz de esto el diseño se suele dividir en partes y se puede hablar de diseño de mecánicas, diseño de personajes, diseño de niveles, etc.

El diseño también abarca la temática del título, condicionando la música y el arte del juego. Por ejemplo el título "Crash Bandicoot" es de temática alegre y fuertemente referenciada por la cultura polinesia. Imaginemos que si en lugar de esta temática, el juego tuviera una temática seria e influenciada por la cultura nipona, el juego sería muy diferente. Por lo que el diseño está presente el todo el juego, en cómo se ve, en cómo se oye, en cómo se siente. Si en algún título hay discordancia entre alguno de estos campos, habrá sido problema del diseño.

El proceso de diseño de un videojuego no es algo que se pueda realizar en poco tiempo ya que requiere pensar al detalle las características del juego.

Existen multiples plantillas para plasmar tanto el documento de concepto (el documento de dos paginas mencionado) como el documento de diseño de un videojuego, con intención de facilitar la realización de dicho documento.

5.1.1. Patrón de documento de concepto

Ficha de juego

Título: Título del juego

Género: género al que pertenece el juego.

Plataforma: Plataforma del juego.

Modos: número de jugadores o modos de juego.

Público objetivo: público al que esta destinado el juego.

Descripción

Breve descripción sobre el juego. Este texto ha de ser de tres o cuatro líneas máximo. Éste es el párrafo con el que se vende el juego a alguien ajeno al juego y deberá contener toda la esencia sobre el título.

Ambiente

Aquí se explicará la historia sobre el juego y la estética que tomará éste.

Mecánicas Centrales

En este apartado, separados por puntos o simplemente por párrafos, se indicarán los puntos importantes para entender el juego. Debido a que este documento ha de ser escueto, en ocasiones se omitirán ciertos detalles sobre las mecánicas, siempre y cuando no sean esenciales

para comprender el juego correctamente. En este apartado también se suele intentar evitar los números directamente, debido a que estos números seguro variarán a lo largo del desarrollo. Los números se evitarán siempre y cuando no estén totalmente claros. Por ejemplo, si sabemos que el juego tendrá ocho personajes, pues está bien indicarlo en este documento pero no es apropiado decir que los personajes tiene cien puntos de salud a menos que sea algo esencial e indispensable para el juego.

En este apartado se suele hablar también, a parte de las mecánicas, de las condiciones de victoria y derrota. También en ciertas ocasiones se habla de las pantallas que el juego posee.

En definitiva, aquí se ha de indicar lo indispensable para que alguien que no conozca el juego en absoluto pueda hacerse una idea sobre el juego.

Referentes

En este apartado se especificarán posibles referentes o inspiraciones para el juego. Éstos no tienen por qué ser otros juegos necesariamente, pueden ser cuadros, imágenes, canciones o cualquier otro producto. Se suele indicar por cada referente que aspecto es aquel del que obtenemos la referencia.

Riesgos

Aquí se indican los riesgos y dificultades que se pueden encontrar a la hora de desarrollar el juego. Se hablará tanto de limitaciones técnicas, de limitaciones de personal o de limitaciones de tiempo, entre otras.

5.1.2. Patrón de documento de diseño

Hablar de patrón para un documento de diseño es algo muy complejo, ya que depende completamente del juego que se este diseñando. Lo que si es cierto que suele haber ciertos puntos que se repiten en cada diseño. El orden de los apartados no tiene por qué ser del mismo orden aquí especificado. Los apartados aquí indicados son solo un posible patrón, pero dependiendo del juego quizás sea necesario destacar y añadir nuevos apartados o prescindir de algunos.

Estos apartados son solo una referencia e incluso hay ciertos puntos que dependiendo quien realice el documento preferirá añadir el contenido en un apartado u otro. Cada apartado tendrá la extensión necesaria para que quede claro el concepto. Es importante, debido a que este documento es de uso interno en el desarrollo, usar un lenguaje comprensible y sencillo a fin de evitar posibles ambigüedades y confusiones.

Introducción

Aquí se escribirá una breve introducción sobre el juego y se explicarán los apartados que

poseerá el documento de diseño.

Ambientación

En este apartado se explicará la ambientación para el juego, es casi indispensable aportar material gráfico como referencia para que aquellos que se lean este documento se hagan una idea del estilo y estética deseada para el juego. También en este apartado se hablará sobre la historia del juego en el caso de que ésta no sea demasiado extensa, en cuyo caso la historia poseerá un apartado propio. También si el juego contiene muchos diálogos tendrán apartados separados.

Personajes

Suele haber un apartado dedicado a los personajes relevantes y no tan relevantes para el juego. De estos se explicará su historia y su contexto, además de aportar material gráfico de éstos. A veces también se indica ciertos aspectos importantes para el juego. Por ejemplo, en un juego de lucha se puede especificar si el personaje tendrá mucha salud, o poca, o incluso si lo queremos detallar aquí y no donde las mecánicas, podemos incluir el elenco de los movimientos de dicho personaje.

Mecánicas centrales

Éste es el apartado más crítico del documento. En éste se explicará toda la jugabilidad del juego de forma tan clara como sea posible. También se suele aportar materiales gráficos para una mejor comprensión de la mecánica. Este apartado casi seguro tendrá subapartados, unos u otros dependiendo enteramente del tipo de juego que se este diseñando. En nuestro caso particular hemos añadido un apartado de combate y otro de movimientos para explicar la funcionalidad de éstos. Si estuviésemos diseñando el “super mario bros” por ejemplo, en este apartado explicaríamos el funcionamiento de los niveles, la mecánica de acabar con los enemigos, la mecánica de la inercia y saltos, entre otras.

Interfaz

Todo documento posee un apartado donde se define, con mayor o menor detalle, tanto la interfaz de juego como las diferentes interfaces que el juego posee en los diferentes menús del juego. Normalmente al inicio del proyecto se usan bocetos para ilustrar acerca del posicionamiento y quizás la estética más que definir completamente las interfaces y a media que el proyecto avanza se va completando con las interfaces totalmente definidas.

Controles

También suele haber un apartado indicando como se juega al juego, especificando para que se usa cada botón.

5.1.3. Documento de concepto Touhou DS

Aquí se realizará el documento de concepto de dos páginas sobre el juego que se programará en el capítulo 6.

Ficha de juego

Título: Touhou DS

Género: Lucha

Plataforma: Nintendo DS

Modos: 1 o 2 jugadores

Público Objetivo: Amantes de los juegos de lucha.

Descripción

Touhou es un juego de lucha en dos dimensiones de corte clásico, donde todos los personajes son femeninos y de estética japonesa, donde priman los ataques a distancia con magias en lugar de los ataques cuero a cuerpo.

Ambiente

La reina del país nipón, ha creado un torneo para conocer a la luchadora más fuerte del país. La ganadora de dicho torneo conseguirá una suculenta recompensa y el honor de ser reconocida como la luchadora más fuerte.

Mecánicas Centrales

Combates uno contra uno en un entorno 2D, donde cada luchador se sitúa en un extremo de la pantalla y siempre miran hacia su oponente.

Cada personaje tiene una barra de vida que desciende al ser golpeado.

Los luchadores se podrán desplazar de izquierda a derecha y podrán saltar y agacharse.

Los personaje poseen dos botones de ataques, uno a corta distancia y otro que lanza uno o varios projectiles.

Pantallas: En la pantalla superior se ubicará el combate y en la inferior se ubicará el tiempo y los "rounds".

Modos de juego: El juego poseerá un "modo historia" que será una sucesión de combates, un "modo arcade" para luchar combates sueltos y un modo dos jugadores para jugar por conexión inalámbrica.

Referentes

Touhou hisoutensoku: Clara inspiración total en este título.

Street Fighter 2: Se intenta que el juego trasmita una sensación similar.

Riesgos

Limitaciones técnicas de la consola: baja memoria Ram (4 MB), poca memoria de video (1 MB) y baja resolución de pantallas.

5.2. Diseño Touhou DS

Este capítulo contiene un diseño superficial de la prueba conceptual que se desarrolla en el capítulo 6. Este documento de diseño es previo al desarrollo. En un caso real, este documento sufriría modificaciones menores a lo largo del desarrollo. Estas modificaciones incluirían el balanceo de los personajes y la finalización ciertos puntos. Es importante destacar que para este juego nos hemos basado en otro juego ya existente, pero que al decir “basado en”, nos referimos sólo a concepto e imágenes de este.

5.2.1. Ambientación

En la era Meiji, había múltiples torneos de lucha para saber qué hombre era el más poderoso. La emperadora Masako harta de tanto derroche de testosterona decide hacer un torneo, bajo las burlas de su esposo, donde solo se permitía combatir a luchadoras. Contrariamente a lo que esperaba su marido Meiji, el anuncio de este torneo ha generado gran expectación y es esperado por muchos.

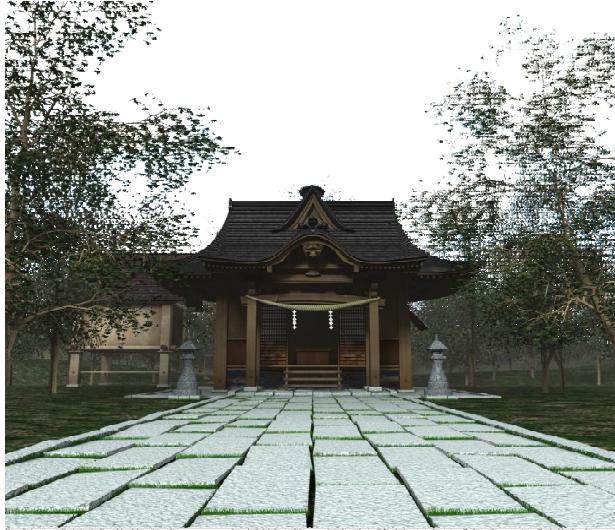


Figura 5.01: Ambientacion "Touhou DS"

5.2.2. Combate

El combate será en un entorno de dos dimensiones, donde los personajes siempre mirarán a

su oponente.

Los combates están divididos en varias rondas. Aquél que gane un número determinado de rondas, ganará el combate. Las rondas se podrán cambiar en el menú de opciones pudiendo elegirse entre dos o una ronda.

El combate terminará cuando uno de los personajes no tenga puntos de vida o el tiempo se agote. El tiempo será de treinta, sesenta, noventa segundos o ilimitado y se podrá cambiar en el menú de opciones. El temporizador va desde el mayor valor hasta cero, es decir, hacia atrás.

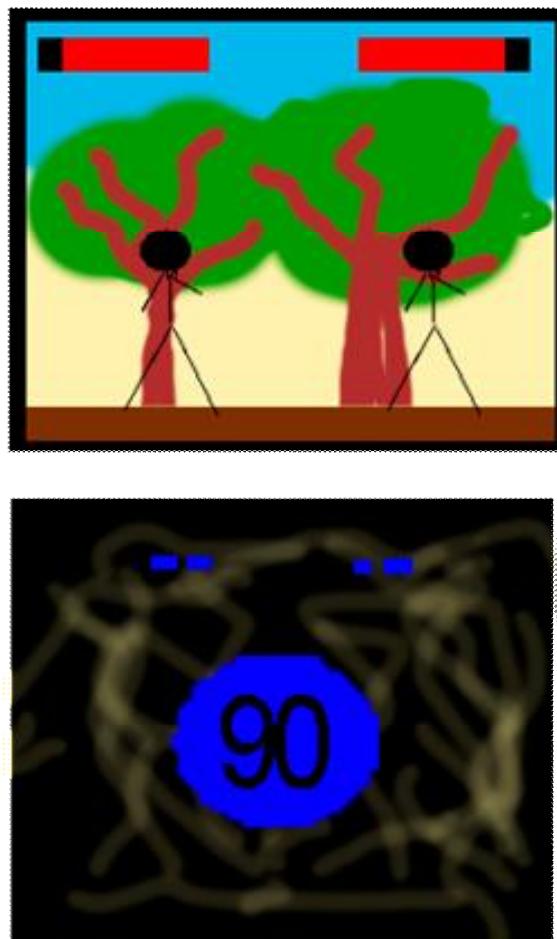


Figura 5.02: Boceto de interfaz de "Touhou DS"

La vitalidad de los luchadores se situará en la parte superior de la pantalla. En cambio el tiempo restante de combate y las rondas ganadas o perdidas se situarán en la pantalla inferior.

En el combate contaremos con múltiples escenarios, los cuales serán aleatorios a la hora de combatir. Estos serán estáticos. La pantalla inferior siempre tendrá el mismo fondo.

En cualquier momento durante el combate, podremos pulsar el botón Start de la consola para

pausar el juego, entonces aparecerá una pequeña ventana donde podremos elegir entre abandonar el combate y volver al menú principal.

5.2.3. Personajes

El juego cuenta con cuatro personajes femeninos bien diferenciados entre ellos, y cada personaje cuenta con dos vestimentas.

Sakuya Izayoi



Aya Shameimaru



Hong Meiling



Tenshi Hinanawi



A efectos de desarrollo, estos personajes serán todos iguales cambiando únicamente los valores numéricos. Los personajes poseen un nombre y una vitalidad.

Estos personajes son tomados directamente del juego en el que nos hemos basado. Los personajes contarán con dos gamas de colores distintas a modo de trajes y se seleccionarán en la pantalla de selección de personajes.

5.2.3.1. Movimientos

Llamamos movimiento a toda acción del personaje en pantalla, es decir, desde el movimiento de izquierda a derecha a los ataques.

Aunque es cierto que cada personaje tiene movimientos únicos, todos estos movimientos llevan un patrón que se cumple para todos los personajes.

Las pulsaciones de los botones en la consola se traducirán en un identificador único con el que se especificará qué movimiento es. Cada personaje tendrá movimientos en todas las direcciones referenciadas con un identificador tal que "BackDown", "Down", "FrontDown", "Back", "Stand", "Front", "BackUp", "Up", "FrontUp". De este modo tenemos controladas las ocho direcciones posibles, y a estas direcciones, en el caso que se esté pulsando algún botón simultáneamente, se le añadirá también dicho botón al identificador, quedando tal que:

Ataque normal: AttackBackDown, AttackDown, AttackFrontDown, AttackBack, AttackStand, AttackFront, AttackBackUp, AttackUp, AttackFrontUp,

Ataque Proyectil: ProjBackDown, ProjDown, ProjFrontDown, ProjBack, ProjStand, ProjFront, ProjBackUp, ProjUp, ProjFrontUp,

También si el personaje está saltando tendremos los mismos índices anteriores pero añadiendo la palabra "Jump" delante.

Gracias a todo esto tenemos un identificador para controlar todos los movimientos posibles del personaje y de este modo es muy sencillo añadir movimientos.

Cada personaje posee un XML creado para este proyecto, donde se especifica los movimientos y sus características.

```
<character>
  <data>
    <health>100</health>
    <maxJumpHeight>60</maxJumpHeight>
  </data>
  <movements>
    <movement>
      <name>Stand</name>
      <command>Stand</command>
      <priority>0</priority>
      <sprite>
        <nameSprite>Stand</nameSprite>
        <width>32</width>
        <height>64</height>
        <numFrames>6</numFrames>
      </sprite>
      <totalDuration>0.6</totalDuration>
      <loopable>true</loopable>
      <canBeBlocked>true</canBeBlocked>
      <movementData>
        <frame>
          <duration>0.066</duration>
          <damage>0</damage>
          <positionX>0</positionX>
          <positionY>0</positionY>
          <offsetX>0</offsetX>
          <offsetY>0</offsetY>
          <hitX>20</hitX>
          <hitY>30</hitY>
          <hitWidth>20</hitWidth>
          <hitHeight>20</hitHeight>
        </frame>
      </movementData>
    </movement>
    <movement>
      ...
    </movement>
  </movements>
</character>
```

En primer lugar tenemos las características (salud y altura del salto) de cada personaje y a continuación un listado de todos los movimientos del personaje divididos en cada frame.

Cada movimiento posee un índice único, usado en la etiqueta "command", que indica qué movimiento será de los identificadores antes mencionados. Gracias a esto una vez que detectemos qué se debe hacer en este movimiento, sabremos directamente qué movimiento ejecutar. A parte del índice cada movimiento tiene otras características tales qué la duración, si es repetible, si puede ser cancelado, etc. Gracias a estos campos podemos controlar qué ocurre si usamos un movimiento y recibimos que se ha de hacer otro.

Cada movimiento posee "frames" y éstos representan cada instante de la animación, y como todo va relacionado con esto, pues también indica qué daño hará, si se ha de desplazar el personaje, etc.



Figura 5.03: Ejemplo imagen de animación

Las imágenes de los movimientos se identificarán por el nombre del movimiento y estarán en celdas verticales del tamaño especificado del movimiento. Cada "frame" indica cada instante del movimiento.

De este modo, dos personajes tan solo tendrán de diferencia el archivo de datos XML y las imágenes de los movimientos. Gracias a esto es muy flexible añadir nuevos personajes aunque es un gran trabajo de balanceado de personajes para ajustar daños, velocidades, etc. Gran parte del balanceo se obtendrá de los datos del juego en el que nos basamos, pero debido a que los datos son distintos que los que se han establecido para este proyecto, no es trivial.

5.2.4. Modos de Juego

En el juego tenemos varios modos de juego que definen algunas propiedades de los

combates.

Story Mode: Este modo consiste en que una vez seleccionado nuestro personaje, lucharemos 4 combates uno tras otro. Se llevará a cabo un combate contra cada luchadora. Si se ganan todos los combates se habrá completado este modo y se guardará el tiempo total que se ha tardado en finalizar todos los combates. Estos combates se juegan a sesenta segundos y a dos rondas.

Arcade Mode: En este modo es donde jugaremos combates por diversión. En la pantalla de selección de personajes elegiremos nuestro personaje y también a nuestro contrincante. Luego podremos elegir el mapa en el que se desarrollará el combate.

VS Mode: Este modo será como el "Arcade Mode" pero con la diferencia de que nos uniremos o crearemos una en red local para luchar contra otra persona. En este modo elegiremos a nuestro personaje y en caso de ser el anfitrión, también elegiremos el mapa.

Option Mode: No es un modo propiamente dicho. Aquí podremos modificar las propiedades del juego como el volumen de sonido, de efectos, el tiempo de lucha, número de "rounds" por combate y el brillo de la pantalla.

En todos los modos de juego, salvo en el de opciones que no se considera un modo propiamente dicho, hemos hablado de la pantalla de selección de personaje, pero no se ha dado información sobre ella.

La pantalla de selección de personaje es una pantalla que antecede al combate, en ella hemos de elegir a nuestros personajes y el escenario donde lucharemos. Esta pantalla mostrará en la pantalla inferior el rostro de los personajes a modo de botones y al pulsar sobre ellos aparecerá en la pantalla superior dicho personaje. Debajo de estos personajes aparecerán unos cuadros especificando los distintos escenarios que podemos elegir. Una vez todo éste seleccionado accederemos al combate pulsando el respectivo botón.

Capítulo 6. Implementación.

Después de toda la teoría sobre Nintendo y los juegos de lucha, después de los tutoriales sobre programación en Nintendo DS y después de hablar un poco de diseño, hemos llegado a la implementación del juego de lucha.

En este capítulo se explica los distintos tipos de metodologías y a continuación se explican los pasos a seguir para realizar dicho juego paso a paso.

6.00 Metodología

En este capítulo se explicará el concepto de metodología ágil, la metodología elegida y en el modo en que se aplicará.

6.00.1. Metodologías ágiles

Las metodologías ágiles [Metodología Ágil, 2015] surgen en contraposición a las largas metodologías donde todo está pensado y escrito en un enorme documento de referencia. En este tipo de metodologías se aboga por una rápida respuesta, una colaboración con el cliente muy dinámica y una gran respuesta al cambio. En la metodología ágil se indican unas pautas a la hora de realizar un software. Estas pautas son las siguientes:

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.
- Respuesta ante el cambio sobre seguir un plan.

Este tipo de metodologías ha ido ganado popularidad con el tiempo, sobre todo para grupos pequeños con poco presupuesto donde cada día es crucial. Existen múltiples metodologías ágiles, cada una con sus ventajas y carencias pero todas ellas siguen las pautas antes mencionadas y a dichas carencias les aportan algunas otras que las caracterizan. Algunas de las más conocidas son:

- Scrum. Desarrollo incremental, solapamiento diferentes etapas, asignación de roles y continuas reuniones cortas.
- Programación Extrema. Desarrollo incremental, programación en parejas, pruebas unitarias cooperación continua con el cliente.
- Lean Software Development. Eliminar desperdicios, ampliar aprendizaje, reaccionar tan rápido como sea posible.

En nuestro caso particular se ha elegido extreme programming por el desarrollo incremental y por la filosofía

6.00.2. Extreme Programming

Extreme programming [Extreme Programming, 2015] es una metodología de programación ágil, esto quiere decir que se intenta no perder demasiado tiempo con la documentación y se intenta dejar a los programadores hacer lo que mejor saben, programar.

Aquéllos que usan "extreme programming" saben que el código está sujeto a continuos cambios y que hay que saber adaptarse a éstos. También saben que hay que trabajar codo con codo con el cliente y también que los trabajadores deben trabajar como un gran equipo, intentando evitar las individualidades.

"Extreme programming" especifica ciertos valores:

Simplicidad: el código ha de ser claro, se intenta evitar los comentarios cuando no aportan demasiado y se espera que el código esté auto comentado con nombres de función y variables acordes a su uso. También se intenta modular todo lo posible para que todo gran problema quede reducido a funciones muy simples y puedan ser fácilmente reutilizadas. Gracias a esto se consigue que de un rápido vistazo se pueda entender qué hace una porción de código. También se aboga a veces por la programación por parejas, asegurando de este modo que todo código pensado por dos personas es programado y revisado al mismo tiempo. Pero esto último no suele ocurrir.

Comunicación: La comunicación se hace gracias al código simple y legible. También se fomenta un buen ambiente de trabajo y en caso de la programación por parejas, se procura que las parejas roten cada cierto tiempo. Otras medidas para fomentar la comunicación son la realización de pruebas unitarias y el contacto constante con el cliente a quien a veces se le exige que trabaje a tiempo total como si un miembro más de la empresa se tratase.

Retroalimentación: al estar el cliente en continua comunicación o a veces trabajando directamente en la empresa, hay una continua crítica al trabajo realizado. Gracias a esto suele ser sencillo dar prioridad a una cosa u otra dependiendo del cliente. También con cada nueva operación que se hace al código se suelen realizar nuevamente las pruebas unitarias por lo que el código está en constante análisis. Por lo tanto estos proyectos suelen llevar un desarrollo iterativo e incremental, donde se parte de una base y se le va añadiendo mejoras unas tras otras.

Coraje o valentía: Debido a que se programa para hoy, y no para mañana, es necesario valentía para afrontar las nuevas partes a desarrollar desechando código si es necesario. No debe dar miedo tocar un código existente para cambiarlo si con esto ganaremos en un futuro.

Respeto: Los programadores han de respetarse y por supuesto respetar las pruebas unitarias que existen, pudiendo modificar lo que sea pero solo cuando el resultado sea mejor que lo que ya había antes programado. También se pide un respeto mutuo por sus compañeros de trabajo.

6.00.3. Metodología

"Extreme programming" es la metodología que se llevará a cabo, pero debido a que solo hay una persona implicada en el proyecto y que no hay un cliente directo, no se pueden llevar a cabo algunas de los valores de la metodología.

En cada iteración, se mostrará un diagrama de clases sobre las clases contenidas en ese tutorial. Para indicar que se clases se van modificando en cada tutorial se usara el siguiente sistema de colores.

- Azul: Clase no modificada.
- Amarillo: Clase modificada respecto al anterior tutorial
- Verde: Clase nueva añadida en este tutorial.

Aunque se vaya a programar un juego, el objetivo sigue siendo didáctico y por este motivo se ha tomado el desarrollo incremental que predica "Extreme Programming" como referente. Los siguientes puntos quedan tal que:

Tutorial 1. Menú inicial. En este capítulo crearemos las clases bases de las que haremos uso más adelante como, por ejemplo, una clase para manejar gráficos. Estas clases no se desarrollarán completamente, si no que se irá ampliando su funcionalidad cuando sea necesario. También crearemos un menú simple y el bucle principal.

Tutorial 2. Escena Combate. Ya en este capítulo abordaremos el combate, incluyendo a un personaje y el principio de los movimientos. Al final de este tutorial tendremos un personaje moviéndose por el escenario de manera un poco cableada y poco flexible.

Tutorial 3. Movimientos. Aquí ya abordamos los XML para facilitar los movimientos a los personajes. Hasta este punto tan solo poseeremos movimientos de corta distancia que no requieran proyectiles. Al finalizar este tutorial ya se poseerá la potencia para que nuestro personaje tenga todos los movimientos ya que éstos serán leídos desde el XML.

Tutorial 4. Movimientos Proyectiles. Una vez que ya tenemos los movimientos a corta distancia, el siguiente paso a realizar será crear los movimientos a larga distancia, en este capítulo ya tendremos los movimientos de proyectiles pudiendo así completar el elenco de movimientos de los personajes.

Tutorial 5. Colisiones. Hasta este punto tenemos los movimientos pero aún no impactan en

nada, tan solo son gráficos. En este tutorial realizaremos el sistema de colisiones para así poder impactar con nuestro movimientos.

Tutorial 6. Menú de modos de juego. Aquí realizaremos la distinción entre los distintos modos de juegos de "Story Mode", "Arcade Mode", "VS Mode"(Sin funcionalidad aun) y "Option Mode".

Version	Funcionalidad	Fecha
Version 0.1.	Clases base del juego: clases para Sprites, fondos, textos, botones, etc. Escena de menú sin arte.	Diciembre 2014
Version 0.2.	Escena de combate y clases para gestión de personajes y movimientos.	Marzo 2015
Version 0.3.	Extensión de la clase de combate y clases para lectura de datos del XML.	Mayo 2015
Version 0.4.	Clase para gestión de proyectiles y modificación de personajes y XML para su lectura.	Junio 2015
Version 0.5.	Creación clase para gestión de colisiones mediante cuadrados y añadida esta colisión a los movimientos y personajes.	Julio 2015
Version 0.6.	Gran cantidad de recursos artísticos añadidos. Añadido también más movimientos.	Agosto 2015

6.01. Introducción

Como ya se ha dicho, llevaremos a cabo un desarrollo incremental, donde cada tutorial y versión del juego, continuará donde lo ha dejado el anterior. Lo mas básico de todo es tener una base y en nuestro caso es una librería matemática.

6.01.1 Conocimientos previos

Antes de entrar en materia es importante conocer cierta información dada por las librerías.

A lo largo del tutorial, se hará uso de los tipos u8 (unsigned char), u16 (unsigned int), s16 (int).

6.01.2. Clase Math

Esta clase será la base matemática de nuestro proyecto, en ésta incluiremos todo lo relacionado con operaciones no básicas.

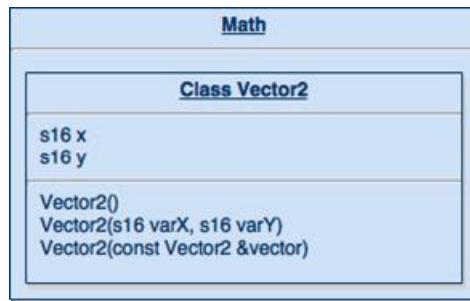


Figura 6.01: Clase Math en tutorial 1

Para empezar hemos creado una clase para el manejo de Vectores de dos dimensiones. La pantalla en Nintendo DS no tiene decimales, por lo que para el vector 2 se ha decidido hacer sin decimales. La clase solo está compuesta por dos enteros.

```
class Vector2{
public:
    int x;
    int y;
```

Tenemos tres constructores

```
Vector2(){}
    x = 0;
    y = 0;}
Vector2(int varX, int varY){
    x = varX;
    y = varY;}
Vector2(const Vector2 &vector){
    x = vector.x;
    y = vector.y;}
```

También poseemos métodos para acceder a los datos y para modificarlos

```

int getX(){return x;}
int getY(){return y;}

void setX(int xVar){x = xVar; }
void setY(int yVar){y = yVar; }

void setXY(int xVar, int yVar){setX(xVar);setY(yVar);}
void setXY(const Vector2 &pos){setX(pos.x);setY(pos.y);}

```

Y también hemos implementado los métodos para comparar, añadir, etc.

```

bool operator ==(const Vector2 &v){
    return (x == v.x) && (y == v.y);
} // operator == vector2
Vector2& operator+=(const Vector2 &v2){
    x += v2.x;
    y += v2.y;
    return *this;
} // operator+= vector2

```

6.02. Tutorial 1. Menú Inicial

En este primer tutorial se empezarán con las clases básicas para el juego. En estas primeras clases se realizará la carga de un gráfico, tanto para el fondo como para cualquier otra cosa. El texto básico y un botón también serán creados en este tutorial. Aunque lo más importante es que se sentarán las primeras bases para el motor mediante la clase "Engine". También se creará la primera escena del juego, el menú principal.

Dicir que de estas clases solo se programará lo necesario para que se puedan ejecutar con el menú. Por ahora se evitará sobrecargar las clases para facilitar su comprensión.

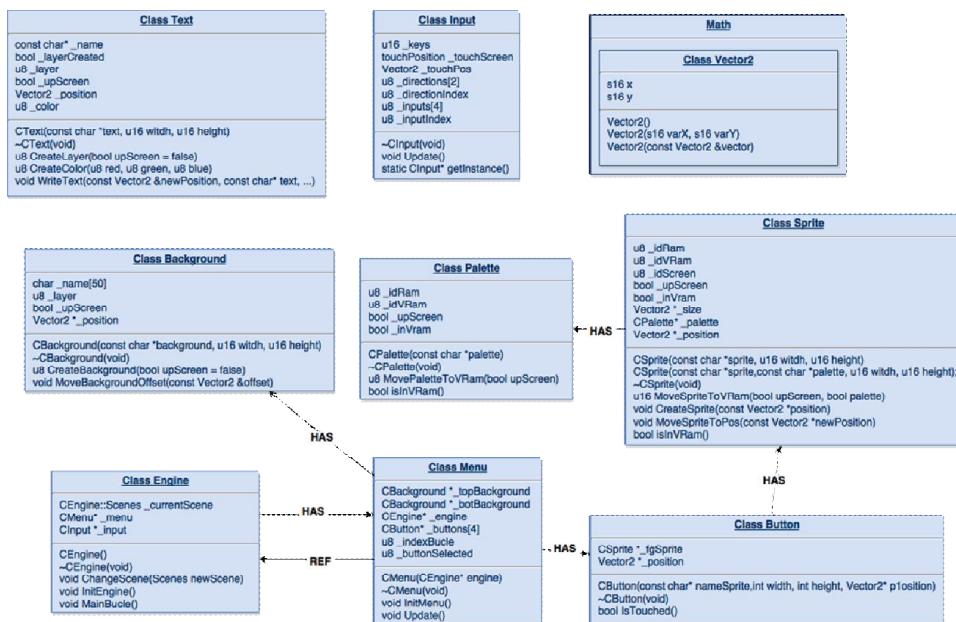


Figura 6.02: Diagrama de Clases Tutorial 1. Menú Inicial

Las clases que se abordarán en este tutorial son:

Clase Sprite: Gráfico genérico.

Clase Palette: Paleta de un gráfico genérica.

Clase Background: Fondo genérico.

Clase Input: Recoge las pulsaciones y las convierte a un enumerado.

Clase Text: Texto genérico.

Clase Engine: Todo lo relacionado con el motor.

Clase Menú: Primera escena del juego.

Clase Button: Botón genérico.

Al finalizar este capítulo, el juego estaba tal que así:



Figura 6.03: Versión 0.1. Menú inicial

6.02.1. Clase Sprite

Esta clase será la base de todos los gráficos del juego. Todos los imágenes que aparezcan y no sean tan solo parte del fondo serán de esta clase (o de clases derivadas de esta). Estos gráficos se pueden mover, escalar y rotar. Todo Sprite tiene una paleta asociada, la cual define los colores que ésta tendrá.

<u>Class Sprite</u>
u8 _idRam u8 _idVRam u8 _idScreen bool _upScreen bool _inVram Vector2 * _size CPalette* _palette Vector2 * _position
CSprite(const char *sprite, u16 width, u16 height) CSprite(const char *sprite,const char *palette, u16 width, u16 height); ~CSprite(void) u16 MoveSpriteToVRam(bool upScreen, bool palette) void CreateSprite(const Vector2 *position) void MoveSpriteToPos(const Vector2 *newPosition) bool isInVRam()

Figura 6.04: Clase Sprite en tutorial 1

La clase sprite está compuesto por los atributos

```

u8 _idRam;      // ID del sprite en Ram
u8 _idVRam;     // ID del sprite en VRam
u8 _idScreen;   // ID del sprite en VRam
bool _upScreen; // ID del sprite en VRam
bool _inVram;
Vector2 * _size;
CPalette* _palette;
Vector2 * _position;

```

Poseemos dos constructores, la principal diferencia entre ambos es que uno creará la paleta y otro no.

```

CSprite(const char *sprite, u16 width, u16 height);
CSprite(const char *sprite,const char *palette, u16 width, u16 height);

```

Este método sirve para transferir la información del Sprite de la RAM a la VRAM.

```

u16 MoveSpriteToVRam(bool upScreen, bool palette);

```

Donde el segundo parámetro indica si debemos pasar la paleta también a la VRAM o no.

Este método trasfiere la información de la VRAM a la pantalla, pintándolo en la pantalla seleccionada en el método anterior.

```

void CreateSprite(const Vector2 *position);

```

Para terminar, este método mueve el sprite a la posición dada.

```
void MoveSpriteToPos(const Vector2 *newPosition);
```

Por ahora éstas son las funciones para mostrar un gráfico por pantalla. Para pintar satisfactoriamente un gráfico hay que realizar la siguiente invocación de métodos:

```
CSprite sprite = new CSprite("nameSprite", "nameSprite", 128, 128);
sprite ->MoveSpriteToVRam(false,true);
sprite ->CreateSprite(new Vector2(4,5));
```

De este modo habremos pintado en la posición (4,5) de la pantalla inferior el gráfico cuyo nombre es nameSprite, con la paleta con el mismo nombre.

Se ha dividido la carga del gráfico en varios métodos en vistas a futuras mejoras de rendimiento.

6.02.2. Clase Palette

Ésta es una clase que tiene la información acerca de la paleta usada por el gráfico. La paleta de un gráfico (sprite) es la información del color de esta, por lo que de un mismo gráfico puede mostrar unos colores u otros tan solo cambiando la paleta. Toda paleta está asociada a un Sprite, o varios, dándole la información del color requerida por ésta.

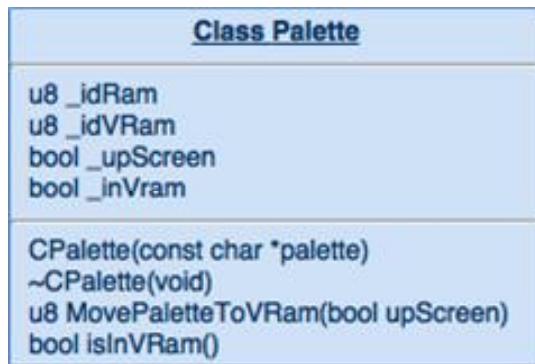


Figura 6.05: Clase Palette en tutorial 1

A continuación se explicaran los métodos de esta clase:

Solo tiene un constructor, este constructor es llamado normalmente en el constructor de la clase Sprite, aunque también se le puede llamar directamente.

```
CPalette(const char *palette);
```

Copia la información de la paleta contenida en la RAM a la VRAM

```
u8 MovePaletteToVRam(bool upScreen);
```

Por ahora estas son las funciones para el trato de una paleta. Un ejemplo de uso podría ser el

siguiente:

```
CPalette palette = new CPalette ("namePalette");
palette->MovePaletteToVRam(false);
```

De este modo habremos creado la paleta con nombre “namePalette” en la pantalla inferior.

Se ha dividido la carga de la paleta en dos pasos, para poder usar una misma paleta en varios gráfico y también por futuras mejoras de rendimiento.

6.02.3. Clase Background

Esta clase posee la información acerca de las imágenes de fondo, estas imágenes tienen la peculiaridad de que son estáticas, no se escalan y no se rotan. Aunque estas imágenes sí se pueden desplazar por la pantalla. Todo Sprite cargado siempre se situará sobre este fondo.

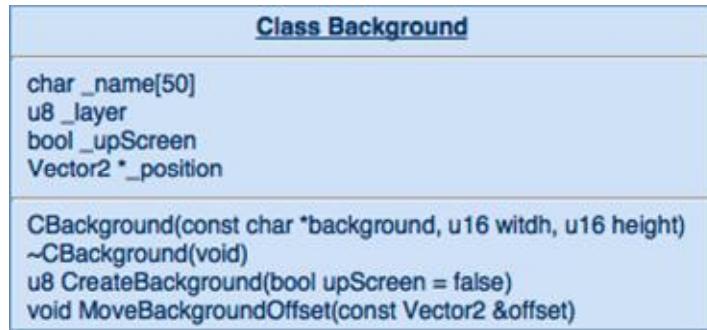


Figura 6.06: Clase Background en tutorial 1

A continuación se explicarán los métodos de esta clase:

Éste es su constructor, el tamaño puede ser superior al de la pantalla pero nunca inferior.

```
CBackground(const char *background, u16 width, u16 height);
```

Este método carga el fondo en la pantalla especificada

```
u8 CreateBackground(bool upScreen = false);
```

Este método desplaza el fondo una cantidad de x e y dada a través del Vector2.

```
void MoveBackgroundOffset(const Vector2 &offset);
```

Por ahora éstas son las funciones para la gestión del fondo. De este modo para pintar un fondo solo es necesario realizar estas llamadas:

```
CBackground background = new CBackground("nameBackground");
background->CreateBackground(true);
```

De este modo crearemos el fondo con nombre “nameBackground” en la pantalla superior.

Se ha dividido la carga del fondo en dos pasos, para poder usar un mismo fondo en ambas pantallas y también por futuras mejoras de rendimiento.

6.02.4. Clase Input

Esta clase será la encargada de obtener todo tipo de interacción con la consola, tanto los botones como la pantalla. Esta será una de las clases que mas crecerán a lo largo de proyecto, sobre todo en cuando se alcancen los combos. Esta clase está realizada con el patrón "Singleton", de este modo es accesible desde cualquier parte del código sin necesidad de referencia directa.

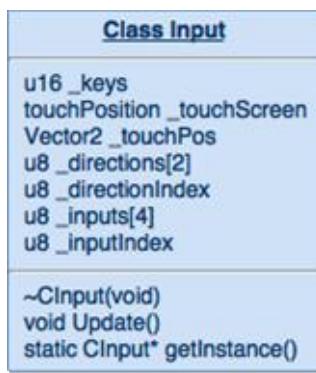


Figura 6.07: Clase Input en tutorial 1

Se han establecido una serie de enumerados para la detección de los controles. Estos enumerados han sido diferenciados en dos enumerados.

```
enum Direction{NoDir, Up, Down, Left, Right};  
enum Inputs{NoInput, Attack, WeakProj, StrongProj, Jump, Dash, Start, Select};
```

El primero de ellos solo se encargará de las direcciones. El segundo es más complejo, se encargara de guardar los tipos de ataques hechos en función de los botones pulsados. Esta clase se encarga de recoger todo, y establecer los patrones para saber qué ataque se ha realizado. Se tiene dos métodos para comprobar qué tipo de input se ha usado. Una vez comprobado se añade a la lista y luego se podrá leer.

```
void checkCombos();  
void checkDirections();
```

Y ya luego se tiene ciertos métodos para obtener la información de los inputs.

```
u8* getDirections(){return _directions;}  
u8* getInputs(){return _inputs;}  
Vector2* getTouchPos(){return &_touchPos;}
```

Por el momento, estos son los métodos públicos que posee (además de algunos privados)
Esta clase no ha hecho mas que empezar, y por ahora con detectar el touchPos nos basta.

Para hacer la llamada desde cualquier lugar del código, se usa el siguiente modo:

```
Cinput::getInstance()->metodoALlamar();
```

6.02.5. Clase Text

Esta clase será la encargada de escribir texto en la pantalla, para esto antes hay que cargar una fuente. Se pueden tener varias fuentes a la vez en la pantalla.

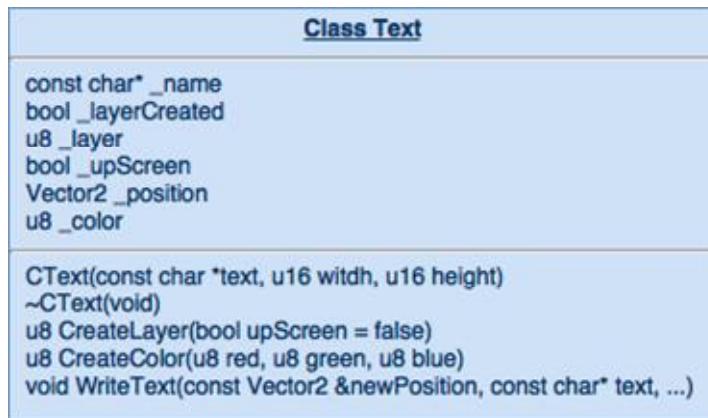


Figura 6.08: Clase Text en tutorial 1

El constructor es el que se encarga de cargar en memoria la fuente

```
CText(const char *text, u16 width, u16 height);
```

Este método crea la capa en la pantalla especificada.

```
u8 CreateLayer(bool upScreen = false);
```

Este método es para poder cambiar el color del texto en caso que sea necesario.

```
u8 CreateColor(u8 red, u8 green, u8 blue);
```

Este método escribe el texto en la posición especificada.

```
void WriteText(const Vector2 &newPosition, const char* text, ...);
```

Para escribir texto por pantalla, tan solo hay que hacer la llamada a los métodos del siguiente modo:

```

CText text = new CText("font01",266,256);
text->CreateLayer(true);
text->WriteText(new Vector2(5,2)," texto en pantalla ");

```

6.02.6. Clase Engine

Esta clase es el pulmón del juego, esta clase llevará la gestión de las diferentes escenas del juego y será la encargada de cambiar entre ellas, inicializándolas y eliminándolas. También esta clase se encargará de inicializar el motor con las funciones necesarias. El bucle principal del juego estará en esta clase, llamando a su vez a la clase del input y de la escena correspondiente.

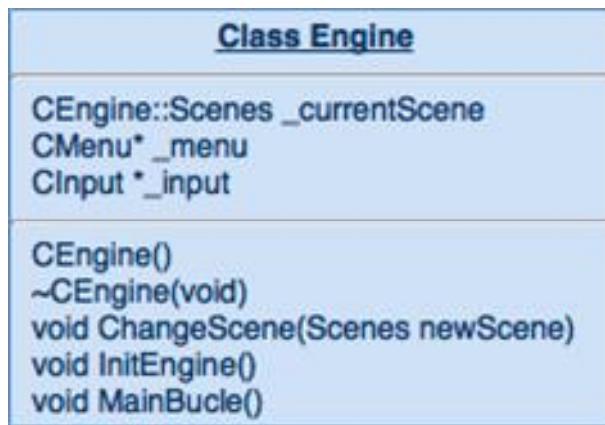


Figura 6.09: Clase Engine en tutorial 1

Esta clase posee un enumerado, para llevar un mejor control de las diferentes escenas del juego.

```

enum Scenes{MENU, ARCADE, VERSUS, OPTION};

```

A continuación se explicarán los métodos básicos de esta clase:

En el constructor es donde se inicializa el motor, llamando a todas las funciones necesarias.

```

Cengine();

```

Este método es el llamado por las distintas escenas cuando es necesario cambiar de escena. Este método llamará al destructor de la escena actual e inicializará la nueva escena.

```

void ChangeScene(Scenes newScene);

```

Este método establece la escena menú como la primera escena y llama a su inicialización. También inicializa el input.

```

void InitEngine();

```

método llamado una sola vez en el juego, dentro de éste está el bucle principal del juego, llamando al update de la clase correspondiente y llamando al motor para que actualice los datos de la pantalla.

```
| void MainBucle();
```

En el main.cpp del código, se llama a estas funciones una sola vez de la siguiente manera:

```
| CEngine *engine = new CEngine();
| engine->InitEngine();
| engine->MainBucle();
```

6.02.7. Clase Menú

Ésta es una de las clases de escenas del juego, esta escena muestra el menú principal del juego, con los botones para acceder a las diferentes escenas del juego. Este menú será navegable tanto con las flechas de dirección como con la pantalla táctil. Aunque por ahora solo es accesible mediante el touch.

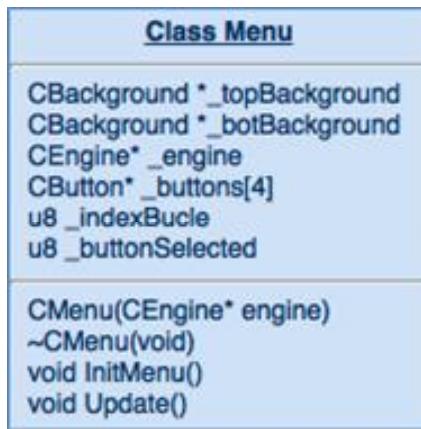


Figura 6.10: Clase Menu en tutorial 1

En el constructor recibe por referencia el motor, para poder cambiar de escena cuando sea necesario.

```
| CMenus(CEngine* engine);
```

Como todas las escenas, tendrá su método de inicialización. Aquí es donde se inicializan los elementos del menú principal, en este caso son los botones y los fondos

```
| void InitMenu();
```

Todos las escenas tendrá su método update, en el cual se hará lo que requiera una actualización constante.

```
void Update();
```

6.02.8. Clase Button

Esta clase es la usada para interactuar con la aplicación, tanto a través de la pantalla táctil como de los botones de la consola.

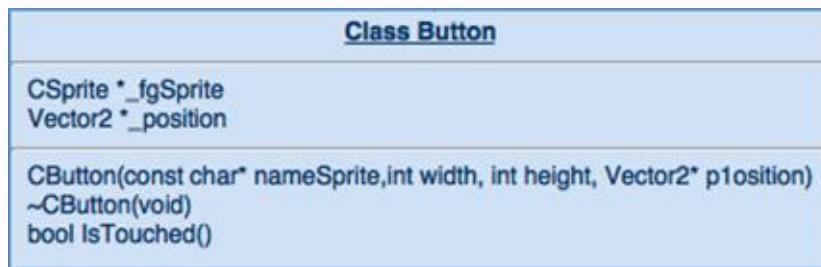


Figura 6.11: Clase Button en tutorial 1

En el constructor, se carga un gráfico, dado su nombre, su altura, su ancho y su posición. Dentro del constructor se creará el "Sprite".

```
CButton(const char* nameSprite,int width, int height, Vector2* position);
```

Dado el alto y el ancho del gráfico, este método detecta si se ha pulsado sobre el botón.

```
bool IsTouched();
```

Este método será muy usado a lo largo del juego, y aunque no posee por ahora interacción con los botones, la poseerá mas adelante.

6.03. Tutorial 2. Escena Combate

Seguimos con la escena principal del juego, la escena de combate. Esta escena tendrá, por ahora, dos personajes. Ésta será la escena donde los personajes interactuarán entre ellos golpeándose.

En este tutorial, haremos el movimiento básico, de izquierda a derecha, de las unidades de manera un poco arcaica. Para llevar esto a cabo, crearemos un sprite animado y empezaremos con la creación de los denominados movimientos, los cuales serán contenidos por un "Character".

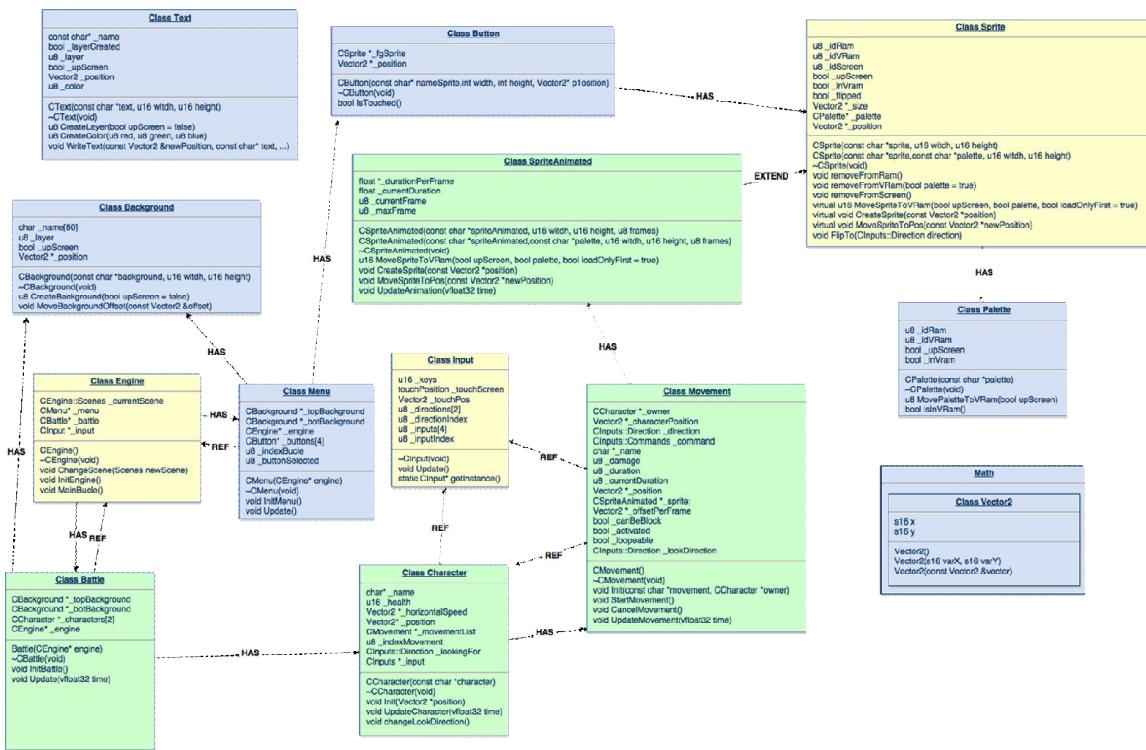


Figura 6.12: Diagrama de Clases Tutorial 2. Escena Combate

Clase Sprite: añadiremos mejoras para la carga y descarga.

Clase SpriteAnimated: clase extensión de Sprite para gestionar animaciones;

Clase Input: se mejorará la captura para usar los enumerados.

Clase Movement: gestionará los diferentes movimientos (izq, ataque a, etc).

Clase Character: contendrá lo esencial del personaje: movimientos y sprites.

Clase Battle: gestionará a los personajes y al escenario.

Al finalizar este capítulo, el juego estaba tal que así:



Figura 6.13: Version 0.2. Escena Combate

6.03.1. Clase Sprite

Primera clase a la que añadimos funcionalidad. En esta ocasión tan solo hemos cambiado algunos métodos para convertirlos en virtuales. Gracias a esto ahora podremos extender de esta clase a la clase Animated Sprite. También se ha añadido un método para voltear horizontalmente este sprite.

Class Sprite	
u8 _idRam u8 _idVram u8 _idScreen bool _upScreen bool _inVram bool _flipped Vector2 *_size CPalette *_palette Vector2 *_position	
CSprite(const char *sprite, u16 width, u16 height) CSprite(const char *sprite,const char *palette, u16 width, u16 height) ~CSprite(void) void removeFromRam() void removeFromVRam(bool palette = true) void removeFromScreen() virtual u16 MoveSpriteToVRam(bool upScreen, bool palette, bool loadOnlyFirst = true) virtual void CreateSprite(const Vector2 *position) virtual void MoveSpriteToPos(const Vector2 *newPosition) void FlipTo(CInputs::Direction direction)	

Figura 6.14: Clase Sprite en tutorial 2

Se han creado métodos para eliminar la información del sprite de los diferentes lugares en los que se carga (Ram, VRam, Screen).

```
void removeFromRam();  
void removeFromVRam(bool palette = true);  
void removeFromScreen();
```

Se ha añadido un método para realizar un volteado horizontal. (solo funciona con los parámetros de entrada “CInputs::Right” y “CInputs::Left”

```
void FlipTo(CInputs::Direction direction);
```

Y los siguientes métodos pasan a ser virtuales.

```
virtual u16 MoveSpriteToVRam(bool upScreen, bool palette, bool loadOnlyFirst = true);  
virtual void CreateSprite(const Vector2 *position);  
virtual void MoveSpriteToPos(const Vector2 *newPosition);
```

6.03.2. Clase SpriteAnimated

Esta clase extiende de la clase Sprite y le añade funcionalidad para las imágenes que tienen animación. La funcionalidad de esta clase es exactamente igual que con el sprite y tan solo añade un método para la actualización de la animación.

Class SpriteAnimated
<pre>float *_durationPerFrame float _currentDuration u8 _currentFrame u8 _maxFrame CSpriteAnimated(const char *spriteAnimated, u16 width, u16 height, u8 frames) CSpriteAnimated(const char *spriteAnimated,const char *palette, u16 width, u16 height, u8 frames) ~CSpriteAnimated(void) u16 MoveSpriteToVRam(bool upScreen, bool palette, bool loadOnlyFirst = true) void CreateSprite(const Vector2 *position) void MoveSpriteToPos(const Vector2 *newPosition) void UpdateAnimation(vfloat32 time)</pre>

Figura 6.15: Clase SpriteAnimated en tutorial 2

Los constructores son semejantes, tan solo hay que añadir el número de frames que tiene la animación

```
CSpriteAnimated(const char *name, u16 width, u16 height, u8 frames);
CSpriteAnimated(const char *name,const char *palette, u16 width, u16 height, u8 frames);
```

El siguiente será el método encargado de cambiar el gráfico actual al siguiente frame, también indicará que el gráfico se ha terminado en el caso de que no sea un gráfico repetible y de volver al primer frame en el caso de que si lo sea.

```
void UpdateAnimation(vfloat32 time);
```

6.03.3. Clase Input

No ha habido grandes cambios en esta clase, tan solo se ha mejorado la detección del input en lo que a las direcciones corresponde. En el siguiente tutorial será donde se realizará el grueso de esta clase, haciendo la detección de los comandos.

Class Input
<pre>u16 _keys touchPosition _touchScreen Vector2 _touchPos u8 _directions[2] u8 _directionIndex u8 _inputs[4] u8 _inputIndex ~CInput(void) void Update() static CInput* getInstance()</pre>

Figura 6.16: Clase Input en tutorial 2

6.03.4. Clase Movement

Esta clase contiene toda la información referente a los movimientos del personaje, entendiendo por movimiento por todas las acciones de éste, como todos los tipos de golpes y desplazamiento del personaje por la pantalla. Éste tendrá varias características.

<u>Class Movement</u>
<pre>CCharacter *_owner Vector2 *_characterPosition CInputs::Direction _direction CInputs::Commands _command char *_name u8 _damage u8 _duration u8 _currentDuration Vector2 *_position CSpriteAnimated *_sprite; Vector2 *_offsetPerFrame bool _canBeBlock bool _activated bool _loopable CInputs::Direction _lookDirection</pre>
<pre>CMovement() ~CMovement(void) void Init(const char *movement, CCharacter *owner) void StartMovement() void CancelMovement() void UpdateMovement(float32 time)</pre>

Figura 6.17: Clase Movement en tutorial 2

Un movimiento poseerá una determinada duración, podrá ser interrumpido o no, un daño, y será en bucle o no. También poseerá un comando de ejecución, un desplazamiento respecto a la posición del personaje y aplicará un desplazamiento al personaje.

Su constructor queda vacío y no se hará nada en él.

```
CMovement();
```

La iniciación del movimiento se llevará a cabo a través del método "Init". En él se establecerán todas sus propiedades.

```
void Init(const char *movement, CCharacter *_owner);
```

Que el movimiento éste iniciado no significa que esté en ejecución, para que comience la ejecución hay que llamar al siguiente método.

```
void StartMovement();
```

Cuando el movimiento ha terminado, se deberá llamar a este método, el cual borrará el gráfico y hará terminar el movimiento

```
void CancelMovement();
```

Este método será llamado cada frame cuando el movimiento esté en ejecución, de este modo se actualizará el tiempo del movimiento y a su vez actualizará el gráfico.

```
void UpdateMovement(vfloat32 time);
```

Éstas son por ahora las características y métodos de los movimientos. Por ahora la carga de las variables se hace de un modo muy pobre. La carga se ampliará en el siguiente tutorial.

6.03.5. Clase Character

La información del personaje se encuentra en esta clase. Un personaje tiene salud y nombre. También posee una lista de movimientos. Esta lista es el aspecto principal del personaje, ya que dictaminará qué podrá hacer el personaje y cómo lo hará. La clase Character leerá el input dado por la clase Input y lo usará para activar un movimiento u otro, de este modo el personaje realizará una acción u otra. El movimiento dictaminará el resto de la información sobre lo que hay que hacer, quedando relegado el personaje como un gestor de dichos movimientos.

Class Character
char* _name u16 _health Vector2* _horizontalSpeed Vector2* _position CMovement* _movementList u8 _indexMovement CInputs::Direction _lookingFor CInputs* _input

CCharacter(const char *character)
~CCharacter(void)
void Init(Vector2* position)
void UpdateCharacter(vfloat32 time)
void changeLookDirection()

Figura 6.18: Clase Character en tutorial 2

Para construir un personaje solo será necesario el nombre de éste.

```
CCharacter(const char *character);
```

Toda la iniciación de las variables del personaje y la lectura de los distintos movimientos, se realiza en el método Init.

```
void Init(Vector2* position);
```

Cada frame será necesario llamar al "update" del personaje, de este modo podrá desplazarse por la pantalla, hacer una ataque o recibir un daño. Dentro de este método se actualizará el movimiento activo y también se comprobará si es necesario cambiar de movimiento.

```
void UpdateCharacter(vfloat32 time);
```

Una vez se lee el input, se comprobará si es necesario cambiar de movimiento con el método

siguiente.

```
void checkMovement();
```

Ya para terminar tenemos dos métodos menores, que funcionan como auxiliares de otros, para quitar carga de trabajo y modular más el código.

El primero para cambiar la dirección hacia la que está mirando el personaje.

```
void changeLookDirection();
```

Y el segundo para evitar activar el movimiento actual dos veces consecutivas, como ocurriría por ejemplo si mantenemos pulsado hacia la derecha.

```
checkAndChangeIfDifferent(u8 newIndex);
```

Mas adelante será necesario cargar los datos del jugador de algún modo, pero por ahora se han puesto datos fijos.

6.03.6. Clase Battle

Ésta es otra de las escenas del juego, en este caso esta escena es la encargada de la batalla entre los luchadores. Esta escena contendrá a los dos personajes, un fondo, la interfaz referente a los personajes como las barras de vida y un contador de tiempo. La interacción con los personajes será mediante los botones de la consola, quedando la pantalla táctil relegada a un segundo plano.

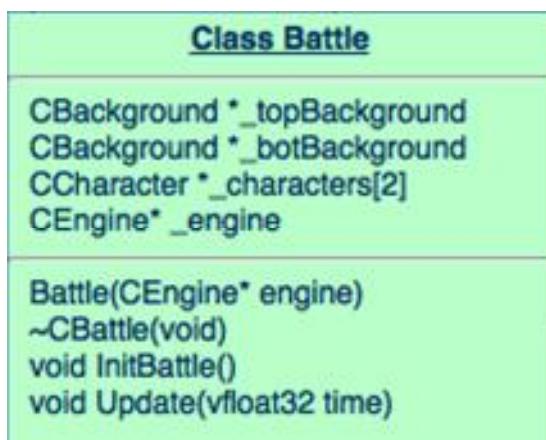


Figura 6.19: Clase Battle en tutorial 2

El constructor de esta clase es similar a la del menú.

```
CBattle(CEngine* engine);
```

En el método "InitBattle" es donde se crean cada uno de los personajes, y se les asigna su posición inicial, así como los fondos y el resto de elementos.

```
void InitBattle();
```

Método donde se llamará al update de cada personaje y del resto de elementos.

```
void Update(vfloat32 time);
```

Ésta es la primera instancia sobre la escena de la batalla, más adelante se modificará para incluir al segundo personaje y al movimiento del fondo de la pantalla. También se añadirán las barras de vida y el tiempo.

6.04. Tutorial 3. Movimientos

Seguimos trabajando con la escena principal del juego, la escena de combate. En esta escena ya habíamos situado un personaje y realizado el movimiento lateral muy básico.

En este tutorial, se realizará los movimientos que no son proyectiles del personaje. Se establecerán las bases para que tan solo reste añadir contenido en un XML ya que la información de los movimientos será leída desde uno.

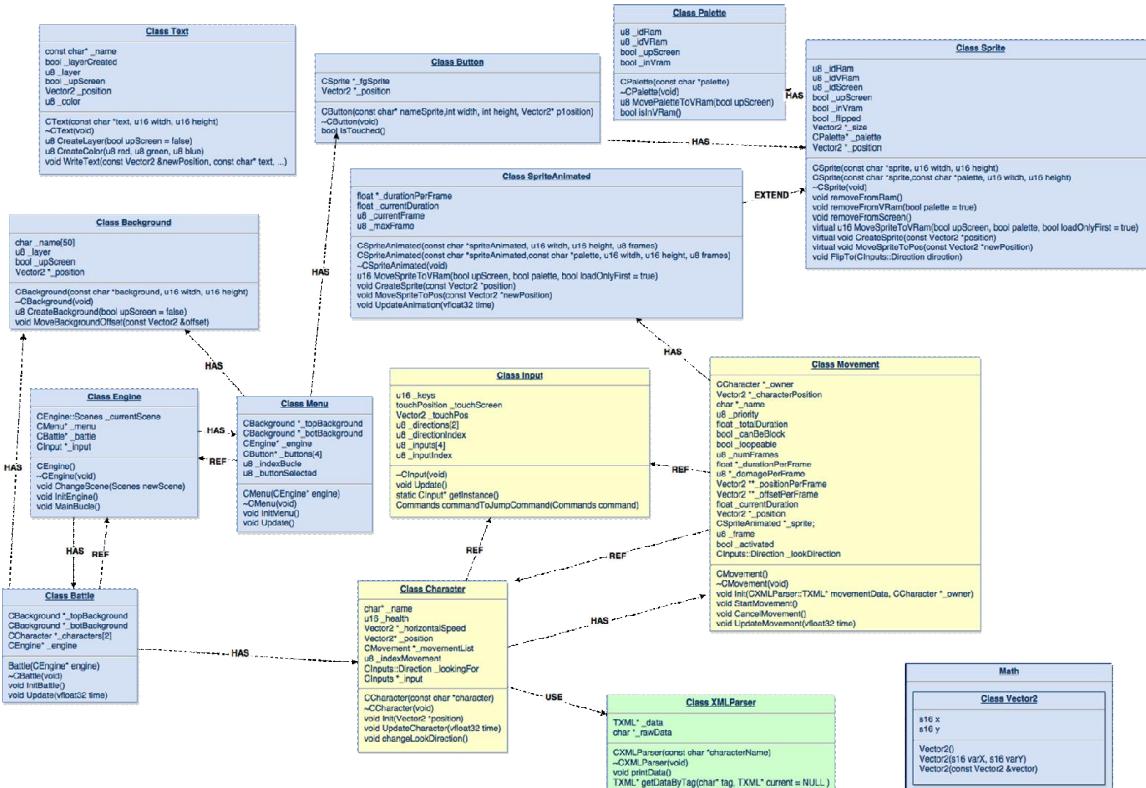


Figura 6.20: Diagrama de Clases Tutorial 3. Movimientos

Clase ParseXML: Se encargará de la lectura del XML.

Clase Movement: Se modificará para la gestión de los datos leídos por XML.

Clase Input: Se mejorará el input para albergar todas las posibilidades.

Clase Character: Se añadirá gestión para los movimientos y el salto.

Al finalizar este capítulo, el juego estaba tal que así:



Figura 6.21: Version 0.3. Escena Combate

6.04.1. Clase XMLParser

Ésta es la clase que leerá todos los datos de los personajes. Para la lectura de los datos se usará el formato XML. El formato del fichero XML tendrá una estructura personalizada acorde a nuestras exigencias.

Class XMLParser
TXML* _data char * _rawData
CXMLParser(const char *characterName) ~CXMLParser(void) void printData() TXML* getDataByTag(char* tag, TXML* current = NULL)

Figura 6.22: Clase XMLParser en tutorial 3

Estos datos se guardarán en la memoria con una estructura personalizada.

```

struct TXML{
    char *tag;
    char *value;
    u8 numChilds;
    TXML **childs;
    TXML *father;
};

```

En esta estructura, el “tag”, representa el “tag” en el XML. El “value” representa el valor de dicho tag. “numchilds” indica el número de hijos que tiene una capa. “childs” contiene múltiples elementos con esta misma estructura sobre cada uno de los hijos en el XML. Y “father”, contiene una referencia al padre en el caso de no ser la raíz.

El constructor de esta clase tendrá por referencia el nombre del carácter. Gracias a esto leerá el fichero XML con el nombre del personaje y le aplicara el “parser” para transformarlo a la estructura mencionada.

```
CXMLParser(const char *characterName);
```

Para leer el fichero no se ha usado ninguna función especial, tan solo se ha usado funciones de C tales como fopen, fseek, fread y fclose.

```
char* ReadFile(const char *file);
```

La función principal de esta clase es “Parse”. Esta función se encarga de hacer toda la lectura necesaria sobre los datos en el fichero y los trasforma a la estructura dada. Esta función hará uso de las otras funciones “createTXML”, “readTag”, “readValue”, “addChildToXML” y “isValidChar”. Gracias a estas funciones tendremos el código más estructurado

```
TXML* Parse(const char *rawData);
```

Para crear una estructura vacía del tipo TXML se usará este método, introduciéndole el tag que lo contiene.

```
TXML* createTXML(char *tag);
```

Para leer los valores del XML, tanto de los tags como los valores, se usan dos funciones muy similares entre ellas. El valor dado lo devuelve a través de un puntero a dicha clase, es la variable llamada outWord. Este método también adelanta el índice por donde se va leyendo el fichero.

```

void readTag(const char *rawData, int *index, char *&outWord);
void readValue(const char *rawData, int *index, char *&outWord);

```

Para la creación de nuevos hijos leídos del XML y por lo tanto, hijos que han de ser añadidos a

la estructura, se usa este método. Donde se le especifica un puntero al elemento de la estructura que estamos leyendo, que hará de padre, el tag del hijo y un puntero a un array que ya tenemos, y que usamos para que no intente destruir lo leído al finalizar el método.

```
void addChildToTXML(TXML *&currentStruct,char* childToAdd, TXML***&temp);
```

Para comprobar si es un carácter válido, se ha creado un pequeño método.

```
bool isValidChar(volatile char c);
```

Esta clase posee también un método de utilidad, que no influye en el Parse, pero sí en su acceso posterior. Este método devuelve la estructura TXML dado un tag.

```
TXML* getDataByTag(char* tag, TXML* current = NULL );
```

6.04.2. Clase Movement

Esta clase ya poseía lo básico para crear un movimiento. Ahora se ha añadido la carga de datos, mediante XML.

Class Movement
CCharacter *_owner Vector2 *_characterPosition char *_name u8 _priority float _totalDuration bool _canBeBlock bool _loopable u8 _numFrames float *_durationPerFrame u8 *_damagePerFrame Vector2 **_positionPerFrame Vector2 **_offsetPerFrame float _currentDuration Vector2 *_position CSpriteAnimated *_sprite; u8 _frame bool _activated CInputs::Direction _lookDirection
CMovement() ~CMovement(void) void Init(CXMLParser::TXML* movementData, CCharacter *_owner) void StartMovement() void CancelMovement() void UpdateMovement(vfloat32 time)

Figura 6.23: Clase Movement en tutorial 3

Para la carga de datos, una vez llamado a su constructor, hay que llamar al método “Init”. Este método se ha modificado pasándole ahora un TXML y el personaje al que pertenece. Este método leerá los datos del personaje y por cada movimiento que éste posee, llamará a dos

métodos, uno para la carga del sprite del movimiento y otro para los datos de cada frame este.

```
void Init(CXMLLoader::TXML* movementData, CCharacter *_owner);
```

Para cargar los gráficos de cada movimiento, se usa este método. Este método leerá el gráfico en memoria estando listo para proceder a su carga en la VRAM y pantalla cuando sea necesario.

```
void initSprite(CXMLLoader::TXML* spriteData);
```

Los datos del movimientos vienen dados por los frames que posee. Cada frame poseerá una duración, un daño, etc. El método siguiente solo rellena la estructura de datos.

```
void initFrames(CXMLLoader::TXML* frameData);
```

En esta clase también se ha modificado el método "StartMovement", y el "UpdateMovement", pero son cambios menores.

6.04.3. Clase Input

En esta clase se han realizado multitud de cambios.

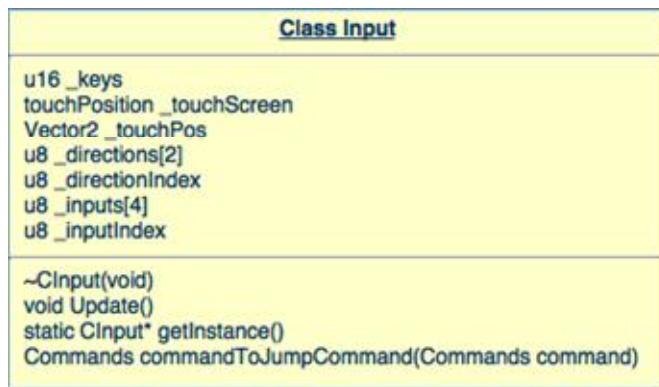


Figura 6.24: Clase Input en tutorial 3

Para empezar se han añadido al enumerado denominado "Commands" todas los posibles comandos que se pueden introducir. Se han usado las nueve direcciones posibles solas y con cada botón posible. También se han añadido todas las anteriores cuando el personaje ésta saltando. De este modo introducir movimientos será mucho más sencillo.

El método "checkCombos", detecta los botones y establece qué comando se ha pulsado.

```
void checkCombos();
```

Disponemos de una función auxiliar para transformar un "input" de tipo "Commands" que no sea Jump a su homónimo de tipo Jump. Éstos son todos los comandos posibles.

```

enum Commands{
    BackDown, Down, FrontDown, Back, Stand, Front, BackUp, Up, FrontUp,
    AttackBackDown, AttackDown, AttackFrontDown, AttackBack, AttackStand, AttackFront, AttackBackUp,
    AttackUp, AttackFrontUp,
    ProjBackDown, ProjDown, ProjFrontDown, ProjBack, ProjStand, ProjFront, ProjBackUp, ProjUp, ProjFrontUp,
    StrongProjBackDown, StrongProjDown, StrongProjFrontDown, StrongProjBack, StrongProjStand,
    StrongProjFront, StrongProjBackUp, StrongProjUp, StrongProjFrontUp,
    DashBackDown, DashDown, DashFrontDown, DashBack, DashStand, DashFront, DashBackUp, DashUp,
    DashFrontUp,
    JumpBackDown, JumpDown, JumpFrontDown, JumpBack, JumpStand, JumpFront, JumpBackUp, JumpUp,
    JumpFrontUp,
    JumpAttackBackDown, JumpAttackDown, JumpAttackFrontDown, JumpAttackBack, JumpAttackStand,
    JumpAttackFront, JumpAttackBackUp, JumpAttackUp, JumpAttackFrontUp,
    JumpProjBackDown, JumpProjDown, JumpProjFrontDown, JumpProjBack, JumpProjStand, JumpProjFront,
    JumpProjBackUp, JumpProjUp, JumpProjFrontUp,
    JumpStrongProjBackDown, JumpStrongProjDown, JumpStrongProjFrontDown, JumpStrongProjBack,
    JumpStrongProjStand, JumpStrongProjFront, JumpStrongProjBackUp, JumpStrongProjUp,
    JumpStrongProjFrontUp,
    JumpDashBackDown, JumpDashDown, JumpDashFrontDown, JumpDashBack, JumpDashStand,
    JumpDashFront, JumpDashBackUp, JumpDashUp, JumpDashFrontUp,
    Hit, HitAir, HitGround,
    Start, Select, NoCommand,
    Size};
}

```

6.04.4. Clase Character

En esta clase los cambios que se han hecho están relacionados con los datos leídos del XML y con el salto. Esta clase es la encargada de llamar al "Parser".

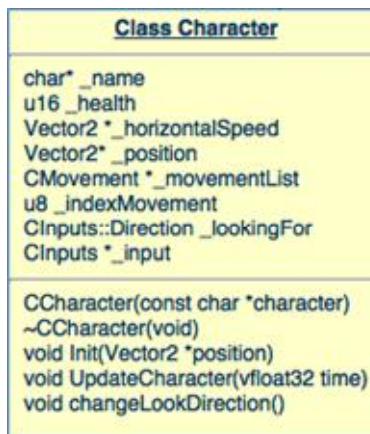


Figura 6.25: Clase Character en tutorial 3

Para cargar los datos de esta clase se hace uso del método "loadAttributes" que leerá los atributos del personaje. Estos serán la vida y la altura de salto.

```

void loadAttributes(CXMLParser::TXML* data)

```

A continuación se hace uso de la función "loadMovement" para cargar los movimientos del personaje. Todos los datos se leerán desde el XML. Este array estará indexado por el identificador de comando para ejecutar el movimiento. De este modo será muy sencillo cambiar de movimiento.

```
void loadMovements(CXMLParser::TXML* data);
```

El método checkMovement hará el cambio de movimiento oportuno.

```
void checkMovement();
```

El método anterior intentará cambiar el movimiento, solo en el caso que se pueda. Para cambiarlo se ayudará del siguiente método.

```
bool checkChangeCommand(u8 newIndex, bool force = false);
```

En cada update, la última comprobación será si se está saltando o no, en caso afirmativo, se ejecuta el movimiento de salto.

```
void executeJump();
```

La clase Character no está del modo más óptimo que se pueda llevar a cabo, y el salto podría haber sido implementado de mejor modo. Pero para nuestra demostración es más que suficiente.

6.05. Tutorial 4. Movimientos de Proyectiles

Una vez que tenemos los movimientos, ya poseemos todos los golpes de contacto que necesitamos para hacer un juego de lucha. Para tener la jugabilidad completa tan solo nos queda añadir los movimientos a distancia, también llamados magias.

En este tutorial, se realizará los movimientos con proyectiles del personaje. Nuevamente se sentarán las bases para que se puedan añadir movimientos de proyectiles de forma sencilla a través del XML.

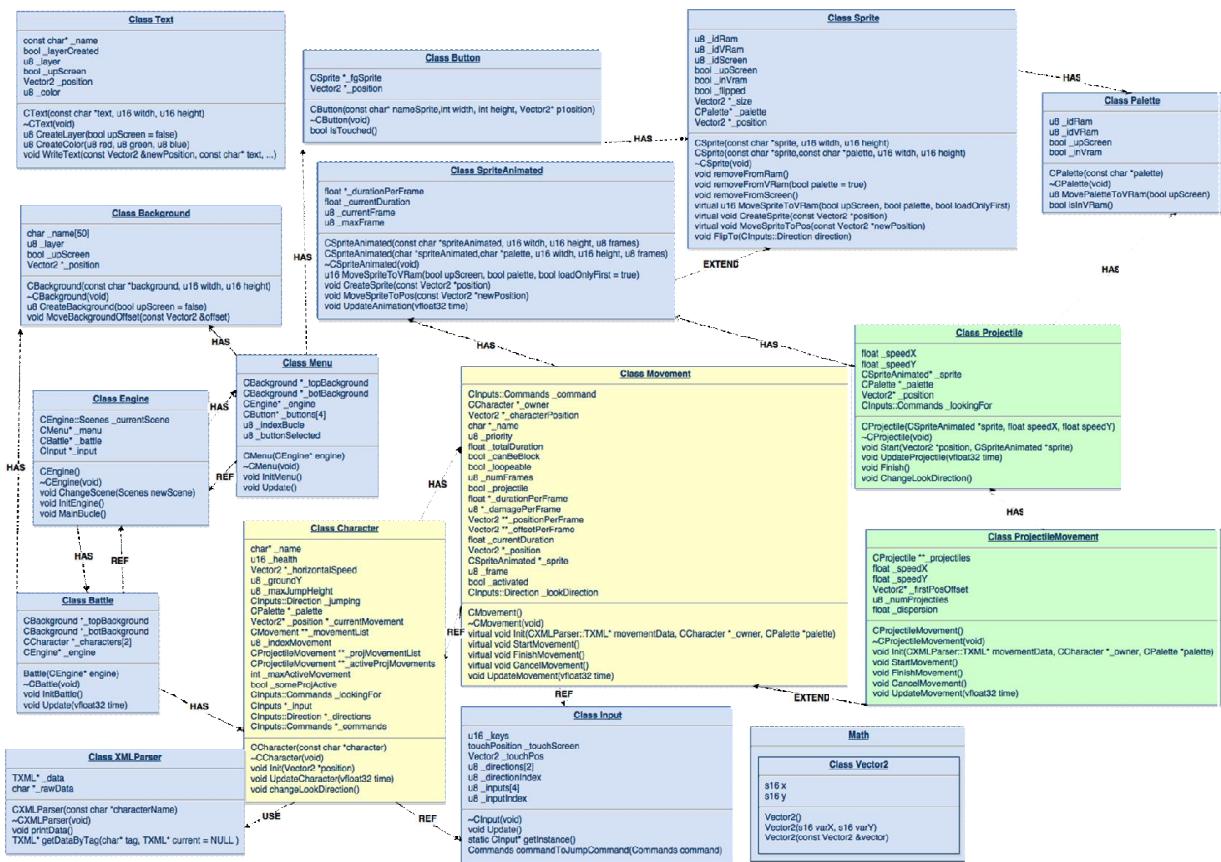


Figura 6.26: Diagrama de Clases Tutorial 4. Movimiento Proyectiles

Clase Movement: Se modificará para habilitar la herencia.

Clase ProjectilMovement: Se crearán basándose en los movimientos normales.

Clase Projectile: Se crearán los proyectiles que se disparan.

Clase Character: Se añadirá gestión para los movimientos de proyectiles.

Al finalizar este capítulo, el juego estaba tal que así:

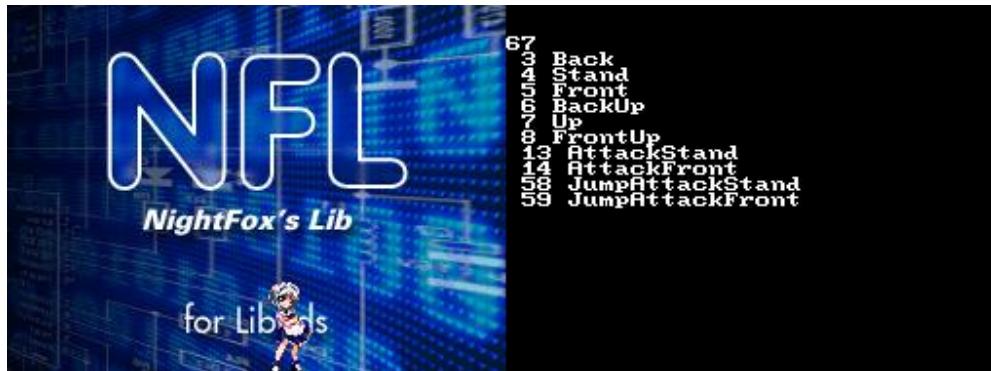


Figura 6.27: Version 0.4. Escena Combate

6.05.1 Clase Movement

Esta clase ya tenía funcionalidad completa para los movimientos. Ahora se han realizado cambios en la clase para poder extender de ella y poder adaptar mas fácilmente los movimientos de proyectiles.

<u>Class Movement</u>
CInputs::Commands _command CCharacter *_owner Vector2 *_characterPosition char *_name u8 _priority float _totalDuration bool _canBeBlock bool _loopable u8 _numFrames bool _projectile float *_durationPerFrame u8 *_damagePerFrame Vector2 **_positionPerFrame Vector2 **_offsetPerFrame float _currentDuration Vector2 *_position CSpriteAnimated *_sprite u8 _frame bool _activated CInputs::Direction _lookDirection
CMovement() ~CMovement(void) virtual void Init(CXMLParser::TXML* movementData, CCharacter *_owner, CPalette *palette) virtual void StartMovement() virtual void FinishMovement() virtual void CancelMovement() void UpdateMovement(vfloat32 time)

Figura 6.28: Clase Movement en tutorial 4

Se han cambiado las funciones y las variables privadas a "protected". También muchos métodos han pasado a ser virtuales para poder ser redefinidos.

El cambio principal realizado en esta clase ha sido dividir el "UpdateMovement" en dos partes, diferenciando ahora cuándo se actualiza el tiempo y cuándo se actualiza el sprite.

```
virtual void updateSprite(float time);  
virtual void updateTime(float time);
```

De este modo tenemos más flexibilidad. El contenido de "UpdateMovement" es diferente ahora y solo hace la llamada a los otros dos métodos.

```
void UpdateMovement(vfloat32 time)
```

Primero calcula y comprueba el tiempo por si el movimiento ha terminado.

```
void updateTime(vfloat32 time)
```

Y segundo se calcula la posición y el "frame" del sprite.

```
void updateSprite(vfloat32 time)
```

6.05.2 Clase ProjectileMovement

Esta clase es la principal de este tutorial, gracias a esta completaremos lo necesario para tener un juego de lucha.

Class ProjectileMovement
CProjectile **_projectiles float _speedX float _speedY Vector2* _firstPosOffset u8 _numProjectiles float _dispersion
CProjectileMovement() ~CProjectileMovement(void) void Init(CXMLParser::TXML* movementData, CCharacter *_owner, CPalette *palette) void StartMovement() void FinishMovement() void CancelMovement() void UpdateMovement(vfloat32 time)

Figura 6.29: Clase ProjectileMovement en tutorial 4

Esta clase extiende de la clase "Movement", por lo que se usará mucho de ella. En cuanto a datos en el XML, los movimientos estarán compuestos de dos tipos de datos, los movimientos del personaje y los movimientos del proyectil.

La parte referente al movimiento es exactamente igual que para los movimientos comunes, por lo que no modificamos absolutamente nada en el XML. La parte del proyectil sí tiene más cambios. Esta clase cargará el gráfico en "Ram" y en "VRam" como si para el personaje se tratara, pero el encargado de pintarlo en la pantalla será la clase "Projectile".

En el XML, a parte de una sección para movimientos tendremos ahora una sección para "projMovements" donde se especificarán los datos para los proyectiles.

```

<projMovement>
    <name>ProjectileProjStand</name>
    <command>ProjStand</command>
    <sprite>
        <nameSprite>ProjectileProjStand</nameSprite>
        <width>32</width>
        <height>16</height>
        <numFrames>1</numFrames>
    </sprite>
    <totalDuration>0.5</totalDuration>
    <loopable>false</loopable>
    <canBeBlocked>false</canBeBlocked>
    <speedX>0.004</speedX>
    <speedY>0</speedY>
    <firstPositionX>16</firstPositionX>
    <firstPositionY>32</firstPositionY>
    <numProjectiles>1</numProjectiles>
    <dispersión>0.04</dispersión>
    <projMovementData>
        <frame>
            <duration>1</duration>
            <hitX>0</hitX>
            <hitY>0</hitY>
            <hitWidth>20</hitWidth>
            <hitHeight>10</hitHeight>
        </frame>
    </projMovementData>
</projMovement>

```

Es muy similar a un movimiento normal pero añadiendo ciertos datos que tenemos que manejar para los proyectiles. Estos datos son la cantidad de proyectiles que tiene el ataque, la dispersión que tendrán éstos y su posición inicial. También tiene un campo de velocidad que especifica el movimiento de los proyectiles. La velocidad viene dada por cada cuánto tiempo avanzan un "pixel" en la pantalla.

La llamada del método Init queda exactamente igual que la anterior, e incluso se llama al Init del padre para cargar los datos. Solo que además se le añade la carga de los nuevas variables.

```

void Init(CXMLParser::TXML* movementData, CCharacter *_owner);

```

Se ha añadido un método llamado "Init Projectiles" que se encarga de crear los objetos "CProjectiles" para controlar las posiciones de éste. En este método es donde se calcula la velocidad que debe tener cada proyectil en función de la dispersión.

```

void initProjectiles();

```

Los métodos de "StartMovement", "FinishMovement" y "CancelMovement" quedan casi igual que en "Movement", haciendo la llamada a éste salvo que se incluye la llamada a "Start" y

"Finish" de los proyectiles.

```
void updateSprite(vfloat32 time);
```

La funcionalidad de este método ha cambiado por completo, ya no actualiza los gráficos si no que ahora esta llamada se propaga al "Update" a todos los proyectiles.

Por supuesto también posee varios métodos para obtener el listado de proyectiles

```
CProjectile** getProjectiles(){return _projectiles;}
u8 getNumProjectiles(){return _numProjectiles;}
```

6.05.3 Clase Projectile

Ya creado el movimiento proyectil y el movimiento, tan solo falta tener el gráfico del proyectil moviéndose por la pantalla.

Class Projectile
float _speedX float _speedY CSpriteAnimated* _sprite CPalette * _palette Vector2* _position CInputs::Commands _lookingFor

CProjectile(CSpriteAnimated *sprite, float speedX, float speedY)
~CProjectile(void)
void Start(Vector2 *position, CSpriteAnimated *sprite)
void UpdateProjectile(vfloat32 time)
void Finish()
void ChangeLookDirection()

Figura 6.30: Clase Projectile en tutorial 4

Para crear un proyectil será necesario especificar el gráfico y la velocidad que tendrá el proyectil. Se recibe el gráfico ya cargado en "Ram" por el movimiento de proyectil correspondiente. Entonces en esta clase se hace una copia de dicho sprite usando el mismo gráfico de "Ram" para ahorrar recursos.

```
CProjectile(CSpriteAnimated *sprite, float speedX, float speedY);
```

Una vez creado y especificado la velocidad del proyectil, cuando se inicializa se le pasa de nuevo el sprite así como la posición inicial donde situaremos el proyectil. Se recibe de nuevo el sprite para copiar los datos que éste posee en la "VRam" a nuestro gráfico del proyectil, por lo que es prácticamente una copia del gráfico anterior.

```
void Start(Vector2 *position, CSpriteAnimated *sprite);
```

La velocidad indica cuánto tiempo ha de pasar para que avancemos un "píxel" en el eje indicado. Debido a esto, en el update hacemos exactamente eso además de llamar al "Update" del gráfico para que actualice la animación si es preciso.

```
void UpdateProjectile(vfloat32 time);
```

Una vez que el movimiento de proyectil nos indique que hemos de terminar, quitaremos el sprite de la pantalla.

```
void Finish();
```

6.05.4 Clase Character

En la clase del personaje se ha añadido toda la gestión relacionada con los movimientos de proyectiles.

Class Character
<pre>char* _name u16 _health Vector2 * _horizontalSpeed u8 _groundY u8 _maxJumpHeight CInputs::Direction _jumping CPalette * _palette Vector2* _position *_currentMovement CMovement ** _movementList u8 _indexMovement CProjectileMovement ** _projMovementList CProjectileMovement ** _activeProjMovements int _maxActiveMovement bool _someProjActive CInputs::Commands _lookingFor CInputs * _input CInputs::Direction * _directions CInputs::Commands * _commands</pre>
<pre>CCharacter(const char *character) ~CCharacter(void) void Init(Vector2 *position) void UpdateCharacter(vfloat32 time) void changeLookDirection()</pre>

Figura 6.31: Clase Character en tutorial 4

Del mismo modo que antes obteníamos y almacenábamos los movimientos se hace ahora para los movimientos de proyectiles. También se ha añadido una pequeña lista para llevar la cuenta de los movimientos activos ya que puede haber varios proyectiles diferentes en la pantalla. También se ha añadido un variable "bool" para saber si hay algún movimiento actual.

Del mismo modo que esta clase ya poseía un método para cargar los movimientos, ahora hay un método para cargar los movimientos de proyectiles. Este método hace casi lo mismo que el de los movimientos pero guardando y leyendo los datos correspondientes.

```
void loadProjMovements(CXMLParser::TXML* data);
```

Debido a que los movimientos de proyectiles tienen un componente de movimiento normal también, cuando el movimiento se activa hay que activar el proyectil en caso de que el movimiento tenga proyectiles.

```
void activateProjMovements(u8 index);
```

También en el Update se ha añadido la gestión para los movimientos de proyectiles en caso de que haya alguno activo

```
void UpdateCharacter(vfloat32 time);
```

6.06. Tutorial 5. Colisiones

Ya tenemos todo tipo de movimientos y de proyectiles, pero aún no disponemos de nada a qué golpear.

En este tutorial añadiremos un segundo personaje en pantalla y añadiremos las colisiones entre los elementos, para así provocar golpes.

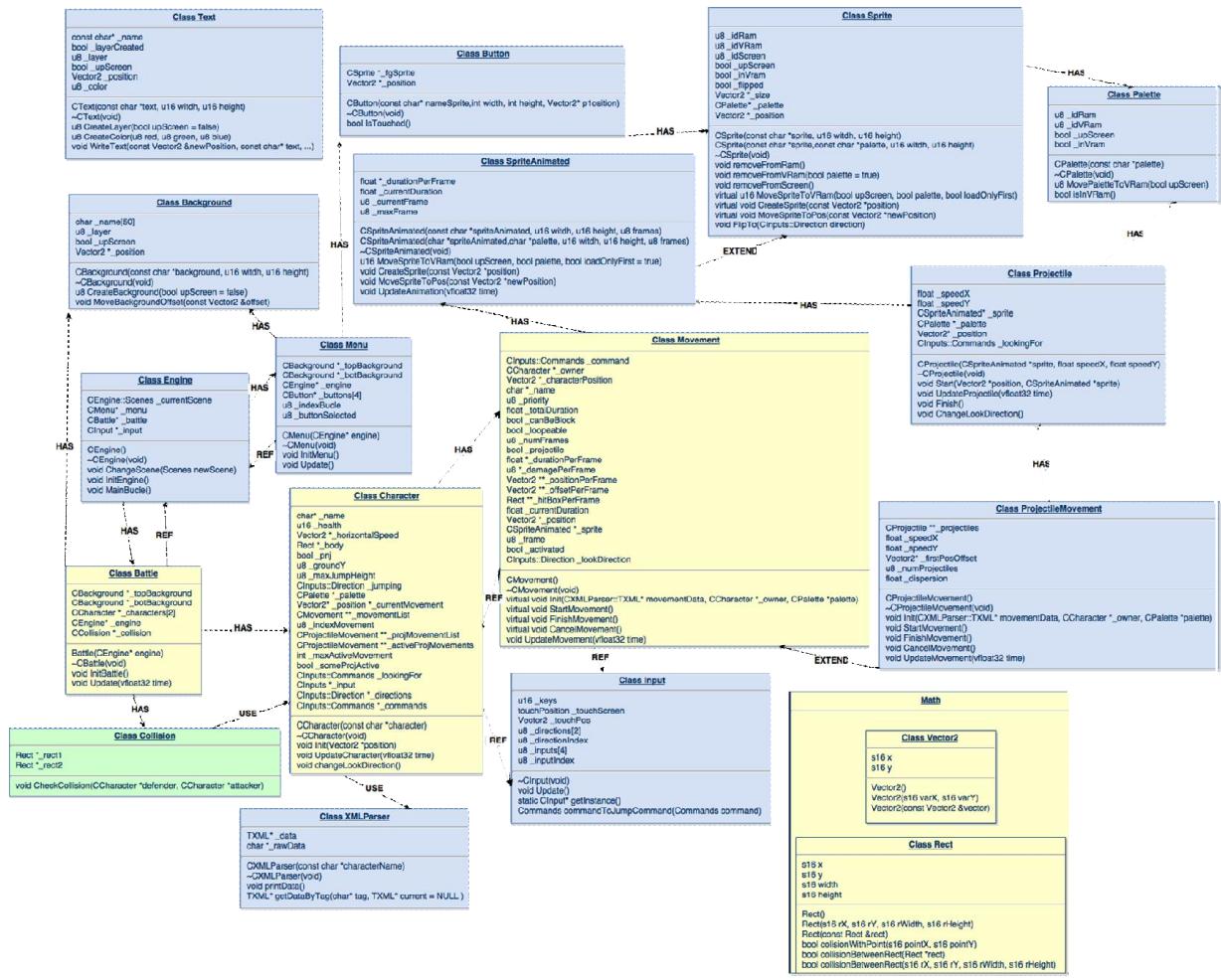


Figura 6.32: Diagrama de Clases Tutorial 5. Colisiones

Clase Math: Se añadirá un "Rect" para controlar las colisiones

Clase Character: Se creará un método para quitar vida y se añadirá un "Rect".

Clase Movement: Se añadirá un "Rect" para colisiones.

Clase Collision: Controlará si hay colisiones entre los elementos del mapa.

Clase Battle: Contiene a la clase collision para llamarla.

Al finalizar este capítulo, el juego estaba tal que así:

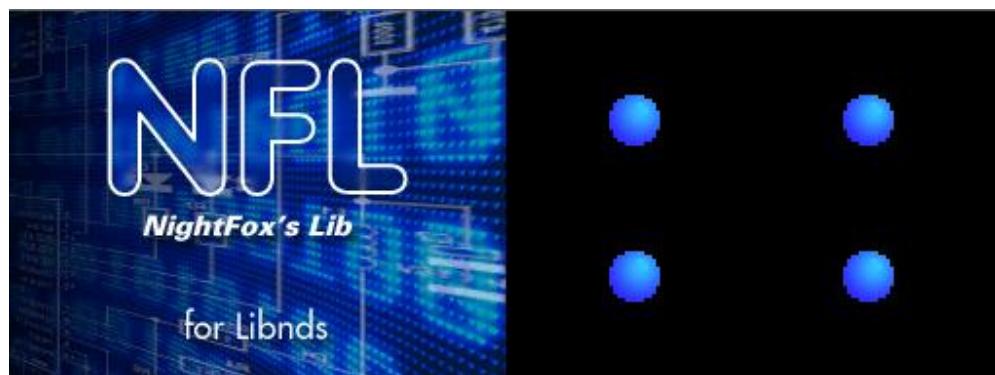


Figura 6.33: Versión 0.5. Menú Inicial

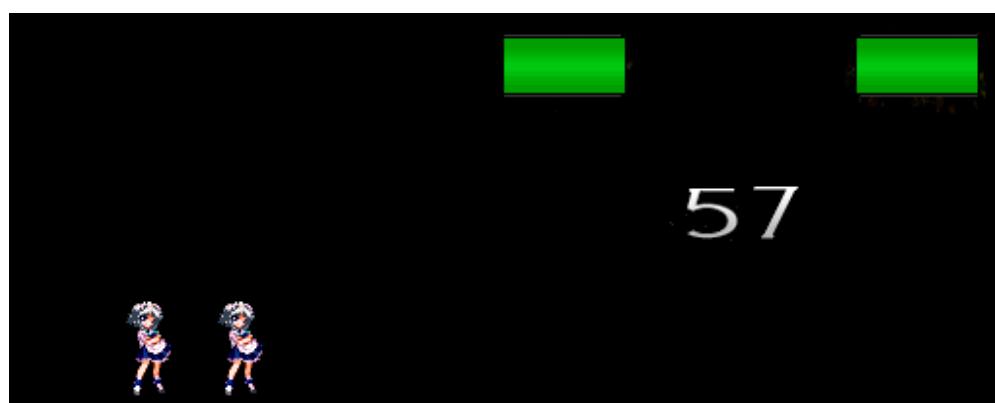


Figura 6.34: Versión 0.5. Escena Combate

6.06.1 Clase Math

Ya teníamos el "Vector2" para controlar posiciones en pantalla y coordenadas en general, ahora haremos una estructura para controlar cuadros de colisión

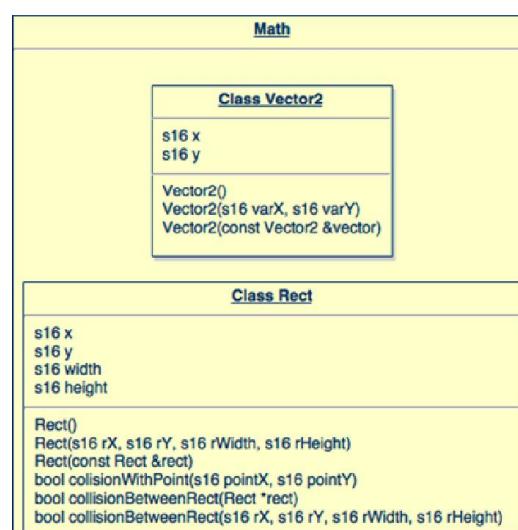


Figura 6.35: Clase Math en tutorial 5

Para controlar las colisiones entre objetos hemos usado una estructura llamada "Rect". Ésta representa un rectángulo que está representado por la coordenada "X" de la esquina superior izquierda, la coordenada "Y" de la esquina superior izquierda, el ancho y el alto del cuadrado. El cubo tendrá que ser dado con coordenadas enteras.

```
class Rect{  
    s16 x;  
    s16 y;  
    s16 width;  
    s16 height;
```

Para su manejo poseemos tres constructores.

```
Rect(){  
    x = 0;  
    y = 0;  
    width = 0;  
    height = 0; }  
Rect(s16 rX, s16 rY, s16 rWidth, s16 rHeight){  
    x = rX;  
    y = rY;  
    width = rWidth;  
    height = rHeight; }  
Rect(const Rect &rect){  
    x = rect.x;  
    y = rect.y;  
    width = rect.width;  
    height = rect.height; }
```

También poseemos métodos para acceder a los datos y para modificarlos

```
s16 getX(){return x;}  
s16 getY(){return y;}  
s16 getWidth(){return width;}  
s16 getHeight(){return height;}  
  
void setX(s16 newX){x = newX;}  
void setY(s16 newY){y = newY;}  
void setWidth(s16 newWidth){width = newWidth;}  
void setHeight(s16 newHeight){height = newHeight;}
```

También se han creado tres métodos para detectar las colisiones entre dos "Rect" y entre un "Rect" y un punto.

```

bool colisionWithPoint(s16 pointX, s16 pointY){
    return x < pointX && pointX < x+width &&
           y < pointY && pointY < y+height;

    bool collisionBetweenRect(Rect *rect){
        return collisionBetweenRect(rect->getX(), rect->getY(), rect->getWidth(), rect->getHeight());

        bool collisionBetweenRect(s16 rX, s16 rY, s16 rWidth, s16 rHeight){
            return x <= rX + rWidth &&
                   x + width >= rX &&
                   y <= rY + rHeight &&
                   y + height >= rY ;

```

6.06.2. Clase Character

Para el personaje, ya teníamos un valor que representaba su vida.

<u>Class Character</u>
char* _name u16 _health Vector2 *_horizontalSpeed Rect *_body bool _pnj u8 _groundY u8 _maxJumpHeight CInputs::Direction _jumping CPalette *_palette Vector2* _position *_currentMovement CMovement **_movementList u8 _indexMovement CProjectileMovement **_projMovementList CProjectileMovement **_activeProjMovements int _maxActiveMovement bool _someProjActive CInputs::Commands _lookingFor CInputs *_input CInputs::Direction *_directions CInputs::Commands *_commands
CCharacter(const char *character) ~CCharacter(void) void Init(Vector2 *position) void UpdateCharacter(vfloat32 time) void changeLookDirection()

Figura 6.36: Clase Character en tutorial 5

Se ha creado una variable para identificar si es un personaje no jugador, "PNJ", para diferenciarlo del personaje que controlaremos. Esto se ha añadido en el constructor

```
CCharacter(const char *character, bool pnj);
```

También se ha añadido en el XML los parámetros para poder ser leídos y crear un "Rect",

llamado body en nuestro caso.

```
Rect* getBody(){return _body;}
```

Para poder quitarle vida al personaje se ha añadido un método.

```
void TakeHitPoints(u16 damage);
```

6.06.3. Clase Movement

En los movimientos no se puede hacer como en el personaje y añadirle una sola colisión.

Class Movement
<pre>CInputs::Commands _command CCharacter *_owner Vector2 *_characterPosition char *_name u8 _priority float _totalDuration bool _canBeBlock bool _loopeable u8 _numFrames bool _projectile float *_durationPerFrame u8 *_damagePerFrame Vector2 **_positionPerFrame Vector2 **_offsetPerFrame Rect **_hitBoxPerFrame float _currentDuration Vector2 *_position CSpriteAnimated *_sprite u8 _frame bool _activated CInputs::Direction _lookDirection</pre>
<pre>CMovement() ~CMovement(void) virtual void Init(CXMLParser::TXML* movementData, CCharacter *_owner, CPalette *palette) virtual void StartMovement() virtual void FinishMovement() virtual void CancelMovement() void UpdateMovement(vfloat32 time)</pre>

Figura 6.37: Clase Movement en tutorial 5

En este caso se ha añadido una lista de colisiones y se rellenará dependiendo de la colisión que se especifica en cada "frame". En caso de que no haya colisiones pues no se rellena.

Se ha creado un método para obtener el "Rect" del movimiento en el "frame" actual.

```
Rect* getCurrentHitBox(){return _hitBoxPerFrame[_frame];}
```

Todos los proyectiles de los movimientos de proyectiles tienen la misma colisión por lo que no es necesario un método en especial para obtenerlos.

6.06.4. Clase Collision

Esta clase será la encargada de comprobar si se ha producido colisión alguna.

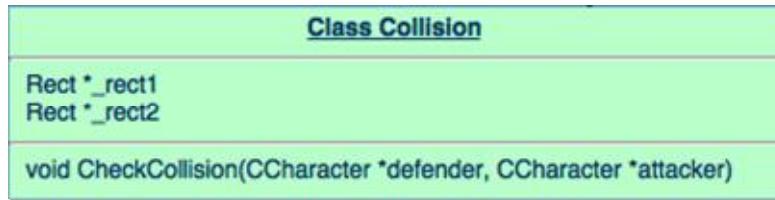


Figura 6.38: Clase Collision en tutorial 5

Las comprobaciones de colisión se realizaran dos sentidos, primero se comprobará si el jugador uno tiene colisión con el jugador dos y luego al revés. Para cada tipo de colisión que se puede producir se ha creado un método específico, aunque solo hay un método público que se encargará de realizar todas las comprobaciones.

```
void CheckCollision(CCharacter *defender, CCharacter *attacker);
```

Para empezar comprobaremos si hay colisión entre projectiles.

```
void checkCollisionBetweenProjectiles(CProjectileMovement **projectilesDef, CProjectileMovement **projectilesAttk, u8 maxElements);
```

A continuación se comprueba si el personaje ha sido impactado por algún proyectil del oponente, en caso afirmativo le restaremos la vida al personaje.

```
void checkCollisionProjectile(CCharacter *defender, CProjectileMovement **projectiles, u8 maxElements);
```

Después de los projectiles se comprueba si ha habido contacto con un ataque cuerpo a cuerpo. Nuevamente en caso afirmativo le restamos vida.

```
bool checkCollisionBetweenAttacks(CCharacter *defender, CCharacter *attacker);
```

Y para finalizar se comprueba si los personajes se están chocando cuerpo con cuerpo. Esto se hace para que no puedan solaparse.

```
bool checkCollisionBody(CCharacter *defender, CCharacter *attacker);
```

6.06.5. Clase Battle

Ya que hemos añadido las colisiones, hay que incluirlas al combate.

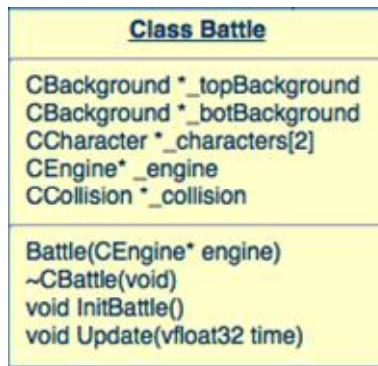


Figura 6.39: Clase Battle en tutorial 5

Se ha modificado el "Init" y "Update". En ambos métodos se incluye el "Collision". También en el init se ha creado otro jugador del mismo modo que creamos el primero.

Tutorial 6.07. Tutorial 6. Menús y modos de juegos

Una vez llegados a éste punto, tenemos el movimiento básico de nuestro personaje.

En este apartado añadiremos interacción por los diferentes menús del juego y añadiremos contenido a la batalla, personajes y escenarios.

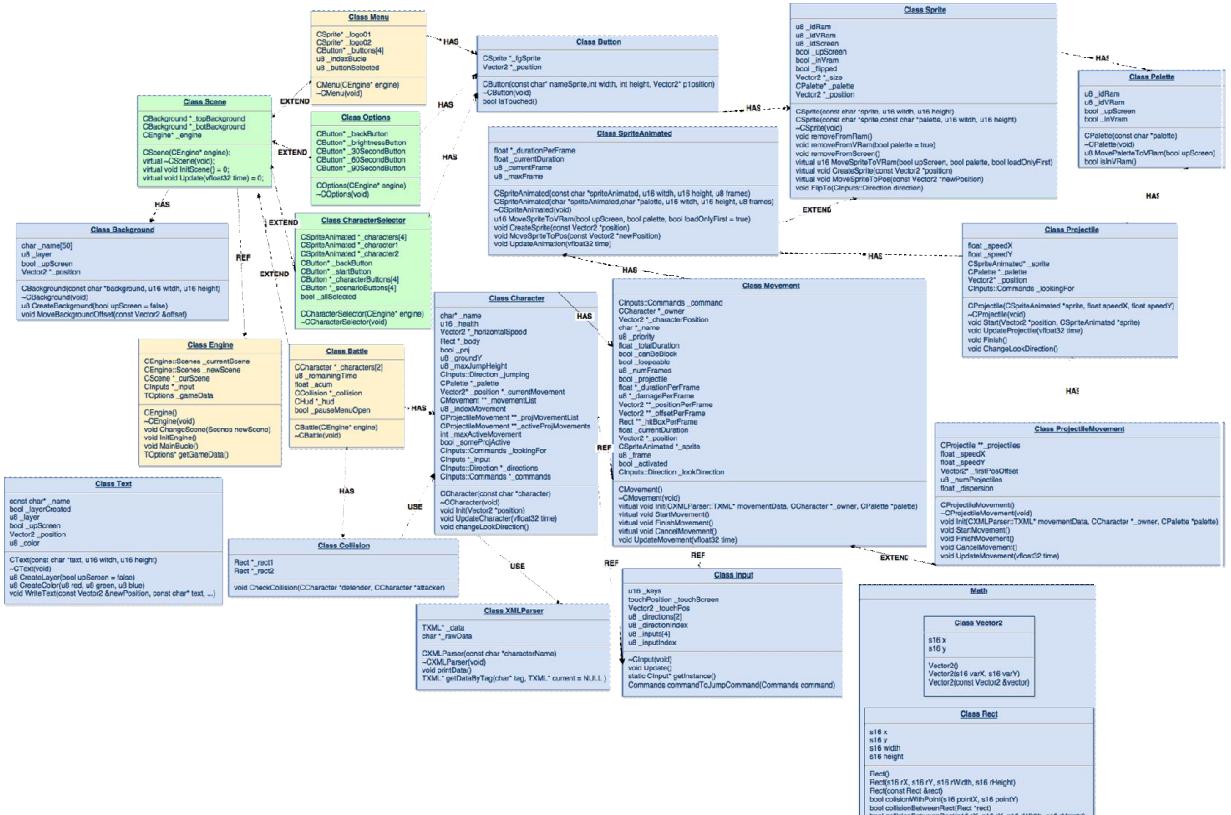


Figura 6.40: Diagrama de Clases Tutorial 6 Menus y modos de juego

Clase Scene: se ha creado una clase padre para facilitar las escenas.

Clase Options: contendrá el menú de opciones.

Clase Character Selector: contendrá el menú de selección de personaje.

Clase Battle: Se ha añadido las diferentes cargas de escenario y personajes.

Clase Menu: Se ha retocado su aspecto visual.

Clase Engine: Se han añadido variables globales y mejorado la interacción de escenas.

Al finalizar este capítulo, el juego estaba tal que así:



Figura 6.41: Versión 0.6. Menú Inicial



Figura 6.42: Versión 0.6. Selección de personajes



Figura 6.43: Versión 0.6. Escena Combate

6.07.1. Clase Scene

Esta clase es la generalización de una escena.

<u>Class Scene</u>
<pre>CBackground * _topBackground CBackground * _botBackground CEngine* _engine</pre>
<pre>CScene(CEngine* engine); virtual ~CScene(void); virtual void InitScene() = 0; virtual void Update(vfloat32 time) = 0;</pre>

Figura 6.44: Clase Scene en tutorial 6.

Posee un método abstracto para iniciar la escena. Este método deberá ser redefinido por las clases heredadas.

`virtual void InitScene() = 0;`

También posee un método, también abstracto, para llevar a cabo la actualización de los elementos en la escena.

`virtual void Update(vfloat32 time) = 0;`

6.07.2. Clase Options

Clase hija de "Scene" encargada de mostrar el menú de opciones. En este menú se podrá cambiar la duración de una batalla.

<u>Class Options</u>
CButton* _backButton CButton* _brightnessButton CButton* _30SecondButton CButton* _60SecondButton CButton* _90SecondButton
COptions(CEngine* engine) ~COptions(void)

Figura 6.45: Clase Options en tutorial 6.

En el método inicial redefinido, se añaden todos los botones del menú de opciones. Los cambios efectuados modificarán ciertas variables de la clase "Engine".

```
void InitScene();
```

En el método "Update" redefinido, se comprueba si se han pulsado los botones y se actualizan las variables globales.

```
void Update(vfloat32 time);
```

6.07.3. Clase Character Selector

Clase hija de "Scene" encargada de mostrar el menú de selección de personajes.

<u>Class CharacterSelector</u>
CSpriteAnimated * _characters[4] CSpriteAnimated * _character1 CSpriteAnimated * _character2 CButton* _backButton CButton* _startButton CButton * _characterButtons[4] CButton * _scenarioButtons[4] bool _allSelected
CCharacterSelector(CEngine* engine) ~CCharacterSelector(void)

Figura 6.46: Clase Character Selector en tutorial 6.

En el método inicial redefinido, se crearán todos los elementos para esta escena. Estos elementos son los personajes seleccionables y los mapas.

```
void InitScene();
```

En el método Update redefinido, se comprueba si se han pulsado los botones y se actualizan las variables de "Engine".

```
void Update(vfloat32 time);
```

Para modular el código, se han creado dos métodos privados para detectar cuando se selecciona un persona o un escenario.

```
void selectCharacter(u8 idCharacter);
void selectScenario(u8 idScenario);
```

6.07.4. Clase Battle

Esta escena ahora extiende de la clase "Scene" y se han redefinido algunos métodos.

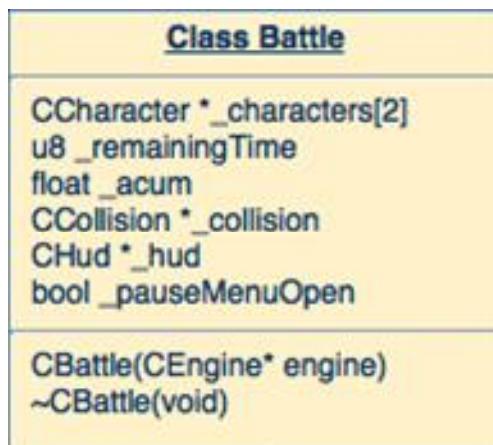


Figura 6.47: Clase Battle en tutorial 6.

El método que antes era "InitBattle" ahora pasa a ser "InitScene" respetando la herencia.

```
void InitScene();
```

Se han añadido dos métodos privados a los que se llama en el "Init". Estos métodos te indican que se ha de cargar, leyendo los datos de las variables de la clase "Engine".

```
char *getPlayerNameById(u8 id);
char *getScenarioById(u8 id);
```

6.07.5. Clase Menu

Clase hija de "Scene" encargada de mostrar el menú principal del juego.

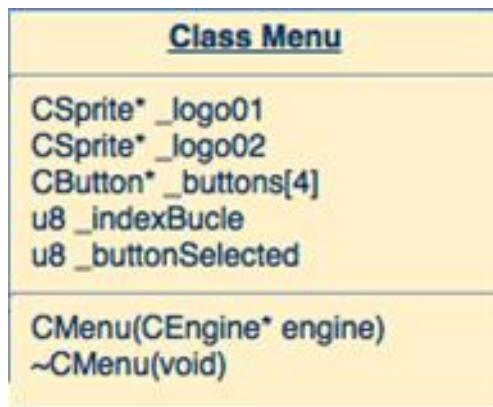


Figura 6.48: Clase Menu en tutorial 6.

El método "initMenu" es ha renombrado como "InitScene" respetando la herencia.

```
void InitScene();
```

6.07.6. Clase Engine

En esta clase solo se ha realizado un pequeño cambio, la inclusión de una variable estática para las opciones del juego.

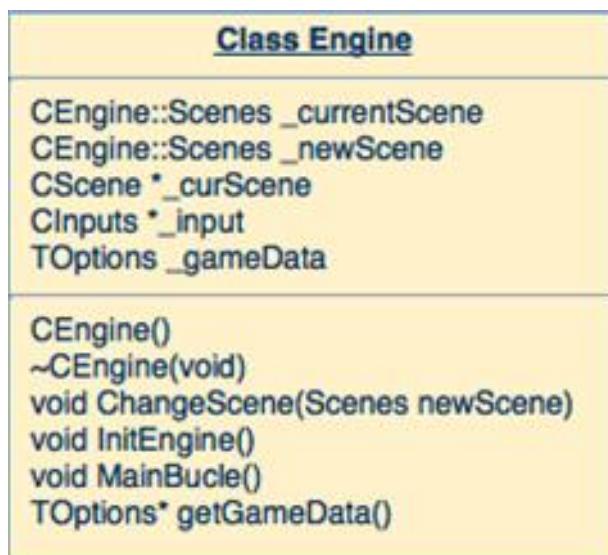


Figura 6.49: Clase Engine en tutorial 6.

Estructura que contiene las opciones del juego. Estas serán accedidas desde casi todo el juego.

```
struct TOptions{
    u8 level;
    u8 character1;
    u8 character2;
    u8 time;
};
```

Conclusiónes

Cuando me aventuré a realizar este proyecto, no sabía que mi vida iba a dar tantas vueltas, vueltas que me han impedido realizar este proyecto en el tiempo que me hubiese gustado, pero aun así he afrontado este proyecto con motivación y optimismo.

A pesar de haber aprendido mucho sobre videojuegos antes de realizar este proyecto, me alegra haber realizado este proyecto ya que gracias a éste, me he acercado al mundo de los juegos de lucha. Nunca había tenido la oportunidad de acercarme a este género y aunque el juego no ejemplifique un completo desarrollo de un juego, considero que si es una buena aproximación para alguien novato en este mundo.

Problemas técnicos

Lo más complicado sin duda ha sido el adaptar el input de los controles para que éstos se transformaran en movimientos. Ha sido complejo realizar el sistema de movimientos, pero una vez realizado tiene casi infinitas posibilidades. Este proyecto puede parecer fácil una vez que se aprecia en una vista general, pero programar para Nintendo DS tiene problemas derivados del software usado en su desarrollo, ya que es muy arcaico y que para los problemas que surgen no hay documentación donde acudir y todo debe ser solucionado por uno mismo en una consola.

Aprendizaje personal

Programar para la Nintendo DS te obliga a prestar especial atención a la eficiencia de tu código y cada byte es importante debido a las limitaciones técnicas que la consola presenta. A día de hoy parece que tenemos recursos ilimitados ya que los sistemas tienden a tener grandes volúmenes de memoria, potencia de computación etc. Si conseguimos programar como si tuviéramos muchos menos recursos podremos añadir mucho más a nuestro juego final.

Gracias a todo esto he de decir que he disfrutado mucho con este proyecto, ya que gracias a él conozco mucho más sobre los juegos de lucha y aunque este juego es tan solo una aproximación a lo que es realmente un gran juego de lucha, ahora me siento capaz de afrontar un juego de lucha de gran envergadura. También programar para Nintendo DS me ha ayudado a pensar en la eficiencia. Me ha ayudado a que después de que algo funcione correctamente, ser muy crítico con mi propio código y preguntarme cómo puedo mejorar esto y si es eficiente o no. Quizás en proyectos pequeños estos detalles no sean necesarios pero en los grandes juegos sí lo son, y estos pequeños detalles son los que definen a un buen programador.

Mejoras y añadidos

En lo que respecta a los tutoriales falta por añadir aun más ejemplos y añadir más funcionalidad. Hay ciertas funciones que se han omitido y que están en la documentación oficial

de la librería. Tampoco se ha profundizado en la librería en la que está basada la que hemos usado, hemos usado la librería "NightFox" que se basa en "libnds". Esta librería es muy extensa y tiene una complejidad mayor que la explicada en este tutorial.

En lo que respecta al juego de lucha, falta añadir contenido en el "XML" para poder así tener mucha más diversidad de movimientos. Resta añadir una interfaz amigable para poder apreciar el combate y resta añadir la condición de victoria. Tampoco se ha realizado un módulo de inteligencia artificial, ya que solo este punto podría ser otro proyecto y no está directamente relacionado con la programación en la Nintendo DS. Tampoco se ha hecho nada relacionado con la comunicación inalámbrica debido a la elevada complejidad que conlleva.

En definitiva, he disfrutado con el proceso y he aprendido a ser mucho más eficiente y darle importancia a los detalles.

Bibliografía

[Vgchartz, 2015] Vgchartz

- <http://www.vgchartz.com/>

[PALib, 2015] Librería para Nintendo DS

- <https://sites.google.com/site/palibwiki/>

[Libnds, 2015] Proyecto en Github sobre la librería

- <https://github.com/devkitPro/libnds>

[DevkitPro, 2015] Devkitpro.

- devkitpro.org/

[NightFox Lib, 2015] Página oficial Night Fox lib.

- www.nightfoxandco.com/

[Hobby consolas lucha, 2015] Artículo de Hobby consolas sobre juegos de lucha.

- www.hobbyconsolas.com/reportajes/nintendo-125-anos-historia-traves-su-logo-86164

[Historia Nintendo, 2015] Artículo de Nintendo sobre su historia.

- www.nintendo.es/Empresa/La-historia-de-Nintendo/La-historia-de-Nintendo-625945.html

[Xataka Nintendo DS, 2015] Artículo de xataka sobre nintendo DS.

- www.xataka.com/consolas-y-videojuegos/diez-anos-de-nintendo-ds-que-nos-dejan-un-total-de-nueve-modelos-diferentes

[Wikipedia, 2015] Diversos artículos en Wikipedia.

- en.wikipedia.org

[GameFili , 2015] Artículo de "gamefili" sobre historia juegos de lucha.

- <http://blogs.gamefilia.com/mhtdtr/04-08-2011/44302/historia-de-los-juegos-de-lucha-i-pioneros>

[SoyUnJugon, 2015] Artículo de "soyunjugon" sobre historia juegos de lucha.

- <https://soyunjugon.wordpress.com/2010/12/12/videojuegos-de-lucha-la-consagracion-los-16-bits-parte-3/>

[IGN, 2015] Diversos artículos en IGN.

- <http://me.ign.com/en/>

[RacketBoy, 2015] Artículo de "racketboy" sobre juegos de lucha.

- <http://www.racketboy.com/retro/fighting/fighting-games-101-all-you-need-to-know-to-battle>

[Hardcoregaming101, 2015] Artículo de "hardcoregaming101" sobre historia juegos de lucha.

- <http://blog.hardcoregaming101.net/2010/10/brief-history-of-2d-fighting-games.html>

[vicbengames, 2015] Artículo de "vicbengames" sobre historia juegos de lucha.

- <http://vicbengames.blogspot.ae/2011/10/historia-de-los-juegos-de-lucha.html>

[Touhou Wiki, 2015] Diversas páginas de la "wiki" no oficial de Touhou.

- http://en.touhouwiki.net/wiki/Touhou_Wiki

[Cplusplus, 2015] Referencia de lenguaje C++

- <http://www.cplusplus.com/>

[C, 2015] Referencia de lenguaje C

- <http://www.open-std.org/jtc1/sc22/wg14/>

[Visual Studio, 2015] Referencia de Visual Studio

- <http://www.visualstudio.com>

[XML, 2015] Referencia de XML

- <http://www.xml.com/>

[Extreme Programming, 2015] Referencia a la metodología "Extreme Programming"

- <http://www.extremeprogramming.org/>

[Audacity, 2015] Software open source para tratamiento audio

- <http://audacity.es/>

[Art Game Design, 2014] Libro que enfoca el diseño de videojuegos mediante la observación de las ideas a través de distintos filtros.

- [The Art of Game Design: A book of lenses](#)

[Theory of Fun, 2013] Libro realmente interesante sobre como divertirse y sobre como proponer retos, enfocado al diseño de videojuegos.

- Theroy of Fun for Game Design

[Metodología Ágil, 2015] Manifiestos para la metodología ágil.

- <http://agilemanifesto.org/iso/es/>