



ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

**LIBWIIESP: BIBLIOTECA LIBRE DE DESARROLLO DE
VIDEOJUEGOS PARA NINTENDO WII**

Ezequiel Vázquez de la Calle

8 de julio de 2011



ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

**LIBWIIESP: BIBLIOTECA LIBRE DE DESARROLLO DE VIDEOJUEGOS PARA
NINTENDO WII**

- Departamento: Lenguajes y Sistemas Informáticos
- Directores del proyecto: Manuel Palomo Duarte y Antonio García Domínguez
- Autor del proyecto: Ezequiel Vázquez de la Calle

Cádiz, 8 de julio de 2011

Fdo: Ezequiel Vázquez de la Calle

A mis padres y a mi hermana,
a mis compañeros de fatigas:
Agu, Dioni y Guti,
a mis tutores, Manuel y Antonio,
y, por supuesto, especialmente a Noe.
Sin todos ellos, este proyecto
no habría sido posible.

Licencia

Este documento ha sido liberado bajo Licencia GFDL 1.3 (GNU Free Documentation License). Se incluyen los términos de la licencia en inglés al final del mismo.

Copyright (C) 2011 Ezequiel Vázquez de la Calle

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Las imágenes de las figuras [4.16](#), [4.20](#), [B.2](#), [B.4](#) y [B.6](#) pertenecen a Space Penguin.

La imagen de la figura [4.7](#) pertenece a Travis E. Wirth.

La imagen de la figura [3.1](#) pertenece a R. S. Shaw.

Índice general

1. Introducción	1
1.1. Objetivos	1
1.2. Alcance	2
1.2.1. Identificación del producto	3
1.2.2. Funcionalidades	3
1.2.3. Aplicaciones del software	3
1.3. Definiciones, abreviaturas y acrónimos	4
1.4. Visión general de la memoria	5
2. Desarrollo del calendario	6
3. Descripción general del proyecto	10
3.1. Perspectiva del producto	10
3.2. Requisitos	11
3.2.1. Requisitos funcionales	11
3.2.2. Requisitos de interfaces externas	12
3.2.3. Requisitos de rendimiento	12
3.2.4. Atributos del sistema software	13
3.3. Características de los usuarios	13
3.4. Restricciones generales	13
3.5. Suposiciones y dependencias	15
4. Desarrollo del proyecto	17
4.1. Diagramas del sistema	17
4.2. Sistema de vídeo	20
4.2.1. Módulos <i>video</i> y <i>GX</i>	20
4.2.2. Identificación de funcionalidad necesaria	21
4.2.3. Diseño e implementación	22
4.2.4. Pruebas	24
4.3. Acceso al lector de tarjetas	24
4.3.1. Módulo <i>wisd_io</i> y <i>libFat</i>	25
4.3.2. Identificación de funcionalidad necesaria	25
4.3.3. Diseño e implementación	26
4.3.4. Pruebas	27
4.4. Los mandos	27
4.4.1. Módulo <i>Wpad</i>	27

4.4.2. Identificación de funcionalidad necesaria	29
4.4.3. Diseño e implementación	29
4.4.4. Pruebas	30
4.5. Recursos multimedia	31
4.5.1. Módulos <i>asndlib</i> , <i>libmad</i> y biblioteca <i>FreeType2</i>	31
4.5.2. Identificación de funcionalidad necesaria	33
4.5.3. Diseño e implementación	33
4.5.4. Pruebas	36
4.6. Trabajar con XML	36
4.6.1. Biblioteca <i>TinyXML</i>	36
4.6.2. Identificación de funcionalidad necesaria	37
4.6.3. Diseño e implementación	37
4.6.4. Pruebas	38
4.7. Organización de los recursos multimedia	38
4.7.1. Identificación de funcionalidad necesaria	39
4.7.2. Diseño e implementación	39
4.7.3. Pruebas	40
4.8. Soporte de internacionalización (i18n)	40
4.8.1. Identificación de funcionalidad necesaria	40
4.8.2. Diseño e implementación	41
4.8.3. Pruebas	43
4.9. Animaciones	43
4.9.1. ¿Qué es una animación?	43
4.9.2. Identificación de funcionalidad necesaria	44
4.9.3. Diseño e implementación	44
4.9.4. Pruebas	45
4.10. Registro de mensajes	46
4.10.1. Excepciones	46
4.10.2. Identificación de funcionalidad necesaria	46
4.10.3. Diseño e implementación	47
4.10.4. Pruebas	48
4.11. Detección de colisiones	48
4.11.1. Identificación de funcionalidad necesaria	48
4.11.2. Diseño e implementación	49
4.11.3. Pruebas	52
4.12. Plantillas	52
4.12.1. Actores	53
4.12.2. Niveles	55
4.12.3. Juego	58
4.13. Ejemplos de <i>LibWiiEsp</i>	59
4.13.1. Arkanoid	60
4.13.2. Duck Hunt	61
4.13.3. Wii Pang	62

5. Pruebas	65
5.1. Plan de pruebas y validación del sistema	65
5.2. Especificación del diseño de pruebas	65
5.3. Especificación de los casos de prueba	67
5.4. Documentación de la ejecución de las pruebas	67
6. Conclusiones	68
6.1. Aspectos generales	68
6.2. Conocimientos adquiridos	68
6.3. Posibles mejoras	69
6.4. Futuro del proyecto	70
Bibliografia y referencias	70
A. Software utilizado	73
B. Manual de instalación y uso	76
B.1. Instalación del entorno de desarrollo	76
B.2. Estructura de un proyecto con <i>LibWiiEsp</i>	78
B.2.1. Estructura de directorios	78
B.2.2. El archivo <i>makefile</i>	79
B.2.3. Ejecución del programa	82
B.3. Consideraciones sobre programar con <i>LibWiiEsp</i>	82
B.3.1. Big Endian	83
B.3.2. Los tipos de datos	83
B.3.3. La alineación de los datos	84
B.3.4. Alineación de datos que usan DMA	85
B.3.5. Depuración con <i>LibWiiEsp</i>	86
B.4. Plantillas de <i>LibWiiEsp</i>	88
B.4.1. Sistemas de coordenadas	88
B.4.2. Actores	90
B.4.3. Niveles	93
B.4.4. Juego	100
C. Manual de referencia	102
D. GNU Free Documentation License	224
1. APPLICABILITY AND DEFINITIONS	224
2. VERBATIM COPYING	226
3. COPYING IN QUANTITY	226
4. MODIFICATIONS	227
5. COMBINING DOCUMENTS	229
6. COLLECTIONS OF DOCUMENTS	229
7. AGGREGATION WITH INDEPENDENT WORKS	229
8. TRANSLATION	230
9. TERMINATION	230

10. FUTURE REVISIONS OF THIS LICENSE	230
11. RELICENSING	231
ADDENDUM: How to use this License for your documents	231

Índice de figuras

1.1. Organización de una textura en <i>tiles</i> , y orden de sus píxeles en el flujo de datos	4
2.1. Diagrama de Gantt con la planificación del proyecto	9
3.1. Diferencia entre las representaciones <i>Big Endian</i> y <i>Little Endian</i>	14
3.2. Ejemplo sobre la necesidad de alineación de los datos	14
4.1. Diagrama de clases del sistema	18
4.2. Diagrama de componentes del sistema	19
4.3. Esquema que ilustra el funcionamiento del sistema gráfico de Wii	23
4.4. Interfaz pública de la clase Screen	24
4.5. Interfaz pública de la clase Sdcard	26
4.6. Operación AND para conocer si hay pulsación de un botón	28
4.7. Ángulos de giro en la orientación de un <i>Wii Remote</i>	28
4.8. Interfaz pública de la clase Mando	31
4.9. Interfaz pública de la clase Imagen	35
4.10. Interfaz pública de la clase Sonido	35
4.11. Interfaz pública de la clase Musica	35
4.12. Interfaz pública de la clase Fuente	36
4.13. Interfaz pública de la clase Parser	38
4.14. Interfaz pública de la clase Galeria	40
4.15. Interfaz pública de la clase Lang	43
4.16. Animación formada por 5 fotogramas	43
4.17. Interfaz pública de la clase Animacion	45
4.18. Interfaz pública de la clase Logger	48
4.19. Sencillo ejemplo de implementación de la técnica de <i>Double Dispatch</i>	50
4.20. Ejemplo para ilustrar el desplazamiento de las figuras de colisión	51
4.21. Interfaz pública de la clase Figura	52
4.22. Captura de pantalla de Arkanoid	61
4.23. Captura de pantalla de Duck Hunt	63
4.24. Captura de pantalla de Wii Pang	64
B.1. Ejemplo de pantalla de error en tiempo de ejecución	87
B.2. Distintos sistemas de coordenadas en el universo del juego	89
B.3. Sencillo autómata de ejemplo para el comportamiento de un actor	92

B.4. Distintas partes de un escenario	94
B.5. Ejemplo de <i>tileset</i> , formado por 6 <i>tiles</i> de 32x32 píxeles	94
B.6. Ejemplo de escenario con <i>tiles</i> atravesables y no atravesables	95

Índice de cuadros

B.1. Tipos de datos que se deben emplear al programar para Wii	84
--	----

Capítulo 1

Introducción

Actualmente, el sector de los videojuegos está en alza, llegando en algunos países europeos a facturar más que la industria cinematográfica. A lo largo de los años, han visto la luz multitud de sistemas especializados en la ejecución de estas aplicaciones software de entretenimiento, las videoconsolas. Sin embargo, es común en este mundillo que los fabricantes no permitan la ejecución de código no firmado por ellos en dichas plataformas. Gracias a la labor de muchas personas a lo largo y ancho del planeta, se ha hecho posible el desarrollo de software casero en algunas videoconsolas (por ejemplo, en la *Sony PlayStation 3*, o en la *Nintendo Wii*). Generalmente, las herramientas para construir y ejecutar este software suele ser de muy bajo nivel, y, en muchas ocasiones, de código cerrado y sin documentación técnica alguna disponible, por lo que se hace patente la necesidad de facilitar al gran público herramientas libres que cubran las posibilidades de desarrollo en estas videoconsolas.

A continuación, se describe de forma general el contenido del proyecto y de esta memoria.

1.1. Objetivos

Los objetivos de este proyecto son varios, entre los cuales destacan especialmente el afán de conocimiento sobre el funcionamiento de una videoconsola (en este caso concreto, la Nintendo Wii), la puesta en práctica de los conocimientos adquiridos durante las asignaturas de la titulación de *Ingeniería Técnica en Informática de Gestión*, la profundización en los métodos de programación de videojuegos de dos dimensiones (ampliando lo aprendido en la asignatura *Diseño de Videojuegos*), y el deseo de aportar una herramienta completa, libre, de alto nivel y documentada en español para el desarrollo de videojuegos en la plataforma Nintendo Wii.

La lista completa de objetivos que se persiguen con la realización del proyecto son los siguientes:

- **Aportar una herramienta completa de desarrollo de videojuegos 2D en Wii:** se pretende ofrecer al mundo del *software libre* una forma de desarrollar videojuegos en dos dimensiones para la consola Nintendo Wii, que resulte

sencilla de utilizar, pero que a su vez permita construir juegos completos.

- **Proporcionar una visión general sobre el funcionamiento de la consola:** una videoconsola no es más que un ordenador dedicado exclusivamente a la ejecución de videojuegos; sin embargo, existen numerosas diferencias en el desarrollo de un programa para una consola respecto a hacerlo para un ordenador personal. Si bien no se desea profundizar al máximo en este asunto, sí se quiere aportar una idea más o menos completa del cambio que supone elegir una plataforma de desarrollo u otra.
- **Ofrecer una documentación completa en español:** actualmente, existen pocas herramientas de desarrollo libres para Nintendo Wii, y las que hay son de muy bajo nivel, además de estar poco o nada documentadas. Con este proyecto se busca proporcionar una documentación completa, detallada y en español, de tal manera que cualquier persona con ciertos conocimientos sobre programación de videojuegos y orientación a objetos pueda desarrollar fácilmente un videojuego para la videoconsola.
- **Desarrollar tres juegos de ejemplo:** la mejor manera de dominar una herramienta software es practicar con ella, pero siempre es conveniente tener un producto acabado que sirva de referencia. En este proyecto se quiere aportar tres juegos, totalmente funcionales y relativamente completos, que ilustren los resultados hasta los que se puede llegar con la biblioteca *LibWiiEsp*.

Arkanoid: clon del clásico de *Taito*, en el que el jugador debe destruir ladrillos, golpeándolos con una pelota que rebota en las paredes del escenario, y debe evitar también que la pelota caiga hacia la zona inferior de la pantalla.

Duck Hunt: basado en el clásico de *Nintendo*. En lugar del comportamiento del videojuego de 1984, en éste participan dos jugadores a la vez. El objetivo de una partida consistirá en ser el primero de los dos en abatir un número concreto de patos.

Wii Pang: clon del clásico *Pang* de *Mitchel Co.* de 1989. El jugador controla a un personaje que debe evitar ser aplastado por unas pompas de colores que rebotan en el escenario del juego. Este personaje lanza ganchos verticales para romper las pompas en otras dos más pequeñas, que se vuelven a dividir en dos cuando reciben otro impacto sucesivamente hasta desaparecer al llegar a su tamaño más pequeño.

1.2. Alcance

Este proyecto pretende cubrir la escasez de herramientas libres que permiten desarrollar videojuegos en dos dimensiones para la consola Nintendo Wii, aportando una biblioteca con la que resulte fácil, pero a la vez eficiente, la construcción de juegos para esta plataforma. También proporciona una documentación útil y completa, enteramente en español, en contraposición a la poca información disponible sobre este tema, y que en general sólo puede encontrarse en inglés.

1.2.1. Identificación del producto

El producto resultante de este proyecto es *LibWiiEsp*, una biblioteca libre y completa, pensada para hacer posible, de una forma sencilla y eficaz, el desarrollo de videojuegos libres en dos dimensiones para Nintendo Wii.

1.2.2. Funcionalidades

Esta biblioteca, escrita en C++ [8] y publicada bajo licencia GPLv3, proporciona una interfaz que permite interactuar con los mandos, el sistema gráfico, el sistema de sonido y el lector de tarjetas SD de la consola. Además, incluye un *parser* de XML sencillo pero efectivo, un sistema de soporte de internacionalización basado en ficheros XML, un sistema de gestión de contenido multimedia (imágenes, efectos de sonido, pistas de música y fuentes de texto), registro de eventos del sistema (*logging*), creación de animaciones a partir de una imagen organizada en rejilla (*spritesheet*) y un módulo de detección de colisiones basado en figuras planas fácilmente ampliable.

Como último punto a destacar, *LibWiiEsp* incluye tres clases abstractas pensadas para ser utilizadas como plantillas para la creación de actores, niveles y la clase principal del videojuego (en la que se controla el bucle principal). La plantilla para niveles permite crear éstos con la herramienta libre Tiled [11], de tal manera que se facilita la creación de escenarios nuevos una vez terminada la programación del juego.

Parte importante del proyecto es también la documentación, que incluye un manual de instalación del entorno y de uso de las plantillas, y un manual de referencia completo.

A pesar de que es posible utilizar *LibWiiEsp* en sistemas *Windows*, *GNU/Linux* y *Mac*, la documentación sólo contempla los sistemas *GNU/Linux*. Además, cabe destacar que el proceso de aprendizaje a la hora de utilizar la biblioteca requiere un esfuerzo moderado en un principio, debido al deseo de cubrir todos los aspectos del desarrollo de un videojuego en dos dimensiones.

1.2.3. Aplicaciones del software

El principal beneficio que aporta *LibWiiEsp* es el de proporcionar, a cualquier persona con conocimientos de C++ y orientación a objetos, la posibilidad de desarrollar videojuegos en dos dimensiones en la plataforma Nintendo Wii.

El aporte de documentación completamente en español y el empleo de técnicas de programación adquiridas durante el transcurso de la titulación de *Ingeniería Técnica en Informática de Gestión*, unido a la sencillez que aporta a la hora del desarrollo, hacen que el producto sea ideal para aprender y poner en práctica los procesos implicados en el diseño y creación de un videojuego en dos dimensiones.

Por otra parte, la evolución natural de una herramienta como es *LibWiiEsp* podría producir la creación de una comunidad hispana de desarrolladores de videojuegos libres para la consola de Nintendo; si bien este hecho puede considerarse algo ambicioso, es una posibilidad a contemplar.

1.3. Definiciones, abreviaturas y acrónimos

- **Scener:** persona que ha colaborado en la obtención de información sobre cómo ejecutar código casero en un sistema cerrado, como una videoconsola o un *smartphone*. Un *scener* suele desarrollar aplicaciones libres basadas en la posibilidad de ejecución de software no firmado digitalmente por el fabricante (por ejemplo, un equipo de cuatro personas ha desarrollado un reproductor multimedia para la Nintendo Wii).
- **Libogc:** biblioteca de muy bajo nivel, escrita en C, y desarrollada por varios *sceners*. Brinda acceso a todo el hardware de la consola, pero es bastante compleja de utilizar.
- **Spritesheet:** imagen o textura, organizada en rejilla, en la que cada recuadro de la rejilla contiene un fotograma de una animación.
- **Textura organizada en tiles:** una textura se representa en memoria como un flujo de datos que contiene la información de los píxeles de una imagen. La organización lineal de una textura consiste en que, en el flujo de datos, se recibe la información de los píxeles en un orden de izquierda a derecha y de arriba hacia abajo (es decir, en un recorrido de los píxeles por filas y columnas). Sin embargo, cuando Nintendo Wii trabaja con una textura, requiere que la información de los píxeles se distribuya organizada en *tiles* o grupos de píxeles, de forma que, si un par de puntos son adyacentes en la imagen, también lo sean en su representación en memoria. Como consecuencia de esto, la información de los píxeles de una textura se reciben, en el flujo de datos, en grupos de 4x4 píxeles que se organizan como puede apreciarse en la figura 1.1.

01 02 03 04 05 06 07 08
09 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64

Datos de textura organizados de forma lineal.
Se reciben en el orden:
01 02 03 04 05 06 07 08 09 10 11 12 ...

01 02 03 04 05 06 07 08
09 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64

Datos de textura organizados en tiles.
Se reciben en el orden:
01 02 03 04 09 10 11 12 17 18 19 20 ...

Figura 1.1: Organización de una textura en *tiles*, y orden de sus píxeles en el flujo de datos

- **GPU:** *Graphics Processing Unit* o Unidad de Procesamiento de Gráficos, hace referencia al chip dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central.
- **EFB:** *Embedded Frame Buffer*. Es el búffer interno con el que trabaja el procesador gráfico (GPU) de Nintendo Wii, el cual recibe un flujo de datos y se encarga de dibujar la información en la pantalla a cada fotograma.
- **Proyección ortográfica:** la proyección ortográfica es un sistema de representación gráfica, consistente en representar elementos geométricos o volúmenes en un plano (en nuestro caso, en la pantalla), mediante proyección ortogonal; se obtiene de modo similar a la sombra generada por un foco de luz procedente de una fuente muy lejana (en el infinito). En una proyección de este tipo, parece haber únicamente dos dimensiones, y la profundidad no se tiene en cuenta: un objeto situado en primer plano tiene la misma proporción que otro que se encuentre en un punto alejado del plano donde se proyecta.

1.4. Visión general de la memoria

Este documento sigue, de una manera más o menos fiel, las pautas recogidas por varios profesores del Departamento de Lenguajes y Sistemas Informáticos en el documento *Recomendaciones para la realización de la Documentación del Proyecto de Fin de Carrera*. A continuación se describen de forma general los capítulos que componen la memoria:

- **Introducción:** Identificación de objetivos, alcance y aplicaciones del proyecto. Vistazo general de la memoria.
- **Planificación temporal:** Desarrollo del calendario que se ha seguido a la hora de realizar el proyecto.
- **Descripción general del proyecto:** Visión general del proyecto, ampliando la información aportada en la sección introductoria.
- **Desarrollo del Proyecto:** Descripción en profundidad de todos los aspectos relativos al desarrollo del proyecto.
- **Pruebas:** Descripción de las distintas pruebas que se han realizado para comprobar y validar los componentes de la biblioteca.
- **Conclusiones:** Valoración global del trabajo realizado en el proyecto, posibles mejoras y ampliaciones.
- **Bibliografía y referencias:** En este capítulo se indican todas las fuentes de información consultadas para realizar el proyecto.
- **Apéndices:** Se incluyen como apéndices un listado de software utilizado en la elaboración del proyecto, el manual de instalación y uso de la biblioteca, el manual de referencia completa y el texto de la licencia bajo el que se libera este documento, que es la GFDLv1.3.

Capítulo 2

Desarrollo del calendario

En este capítulo puede consultarse el desarrollo temporal del proyecto, reflejado en un diagrama de Gantt. Cabe destacar que el proyecto se ha elaborado en el periodo que abarca desde octubre de 2010 hasta julio de 2011, pero sufrió un parón de un mes entre noviembre y diciembre de 2010, por lo que el tiempo de desarrollo aproximado constó de 245 días. A continuación se resumen las actividades realizadas en cada una de las fases de ejecución del proyecto:

- **Fase de planificación** [35 días]: Se trata de la primera fase, en la que se pone en marcha el proyecto.
 - Idea de proyecto [3 días]: en este punto se plantean varias opciones sobre en qué consistirá el proyecto, sin tener en cuenta detalles sobre el desarrollo, descartando las ideas que se excedían en su complejidad o no llegaban a un mínimo nivel de contenido. En un primer momento se decide construir un videojuego para la consola Nintendo Wii.
 - Estudio de viabilidad [7 días]: tras acotar aproximadamente el contenido del proyecto, se procede a recabar información general sobre si es posible o no desarrollar para Nintendo Wii, y en qué condiciones. Tras la recogida de datos desde diversas fuentes, se llega a la conclusión de que es necesaria una inmersión en la programación para la videoconsola, con el objetivo de conocer hasta qué punto es viable desarrollar con las herramientas de bajo nivel encontradas.
 - Pruebas de viabilidad [10 días]: se pone en práctica todo lo encontrado en el punto anterior, llegando a la conclusión de que no es suficiente contar con las herramientas proporcionadas por los *sceners*, ya que operaciones relativamente sencillas como cargar un recurso multimedia desde la tarjeta SD requiere una cantidad enorme de líneas de código.
 - Cambio de planteamiento [1 día]: tras las conclusiones obtenidas a lo largo de los puntos anteriores, se decide cambiar el objetivo del proyecto; en lugar de implementar un videojuego, se desarrollará una herramienta completa, con documentación amplia y en español, que sirva para desarrollar videojuegos en dos dimensiones para Nintendo Wii.

- Búsqueda de documentación adicional [10 días]: desde el momento en el que se decide construir una biblioteca completa para desarrollar videojuegos para Nintendo Wii, se profundiza en la documentación encontrada, obteniendo nuevas fuentes de información, pero todo en inglés. Se decide que, como complemento para la herramienta, se debe generar una documentación apropiada en español.
 - Entrevista con los tutores [1 día]: tras tener clara la idea de proyecto que se quiere llevar a cabo, se concierta una reunión con los directores del proyecto para exponerles la situación. Los tutores aceptan la idea, y se firman los correspondientes documentos.
 - Planificación temporal del proyecto [3 días]: una vez el proyecto comienza a andar, se realiza una estimación temporal del desarrollo, siendo ésta bastante flexible en el sentido de que, en un principio, no se conoce el alcance que tendrá la herramienta de desarrollo para Nintendo Wii.
- **Fase de ejecución** [130 días]: Una vez establecidos los objetivos del proyecto se procede a comenzar el desarrollo propiamente dicho.
- Especificación de requisitos [15 días]: se pulsa la idea de herramienta de desarrollo para Nintendo Wii, acotando qué funcionalidad se proporcionará a los usuarios de la biblioteca, y descartando una serie de puntos que, aunque podrían ser interesantes, aumentan exponencialmente la complejidad del sistema. A pesar de ello, posteriormente se irán añadiendo más requisitos a medida que se van concluyendo los objetivos marcados en un primer momento, consecuencia ello de la metodología de desarrollo seguida. Por otro lado, se decide crear tres sencillos juegos de ejemplo para ilustrar la utilidad de la herramienta una vez creada.
 - Iteraciones de desarrollo [95 días]: una vez decidida la funcionalidad general del producto, se procede a desarrollar cada uno de los módulos que cubrirán los distintos aspectos de la biblioteca. La construcción de cada módulo se descompone en cuatro fases bien diferenciadas:
 1. Análisis: partiendo de los requisitos indicados en la especificación, se marca exactamente qué se quiere conseguir con el desarrollo del módulo correspondiente, y qué no se podrá ofrecer por aumentar excesivamente la complejidad. Se deciden las tecnologías, bibliotecas externas y otros detalles necesarios para comenzar la implementación de cada apartado de la herramienta.
 2. Diseño: a la hora de definir cómo cumpliría su cometido cada uno de los módulos de la biblioteca, se tuvieron en cuenta todas las fuentes de información consultadas previamente, además de otras tantas que se iban localizando a medida que era necesario. Debido a ello, la fase de diseño de algunos módulos (como el módulo de vídeo, por ejemplo) se alargó bastante en el tiempo.
 3. Codificación: tras establecer claramente cómo trabajaría cada componente de la herramienta, llegaba el momento de implementar el módulo.

lo correspondiente. Se tuvieron que continuar realizando indagaciones sobre la forma de trabajar de Nintendo Wii, ya que en cada fase de implementación surgían errores desconocidos y conceptos particulares de la videoconsola.

4. Pruebas: después de terminar la codificación de cada módulo, se dedicaba un cierto tiempo a las comprobaciones y validaciones necesarias de ese módulo concreto, más las pruebas oportunas para comprobar cómo se incorporaba el nuevo módulo al conjunto de los ya existentes.
- Juegos de ejemplo [20 días]: tras dar por finalizada la construcción de *Lib-WiiEsp*, se desarrollaron los tres juegos que acompañan a la herramienta. La creación de estas tres aplicaciones constó también de las clásicas etapas de análisis, diseño, codificación y pruebas, y se requirió relativamente poco tiempo (menos de un mes para los tres ejemplos), demostrando que la biblioteca facilita enormemente el desarrollo de videojuegos.
- **Fase de documentación** [60 días]: Ya durante el desarrollo de la biblioteca se fue generando documentación, especialmente tras concluir la construcción de cada uno de los módulos. Sin embargo, una vez finalizada la herramienta se redactaron el manual de instalación y uso y la memoria del proyecto, suponiendo una inversión considerable de tiempo, y alternándose la construcción de los juegos de ejemplo con la generación de documentación.
- **Fase de finalización** [20 días]: En este último punto se produce la revisión final del proyecto y la documentación por parte de los tutores, se preparan los materiales necesarios para la presentación del trabajo, y se realiza la defensa ante el tribunal.

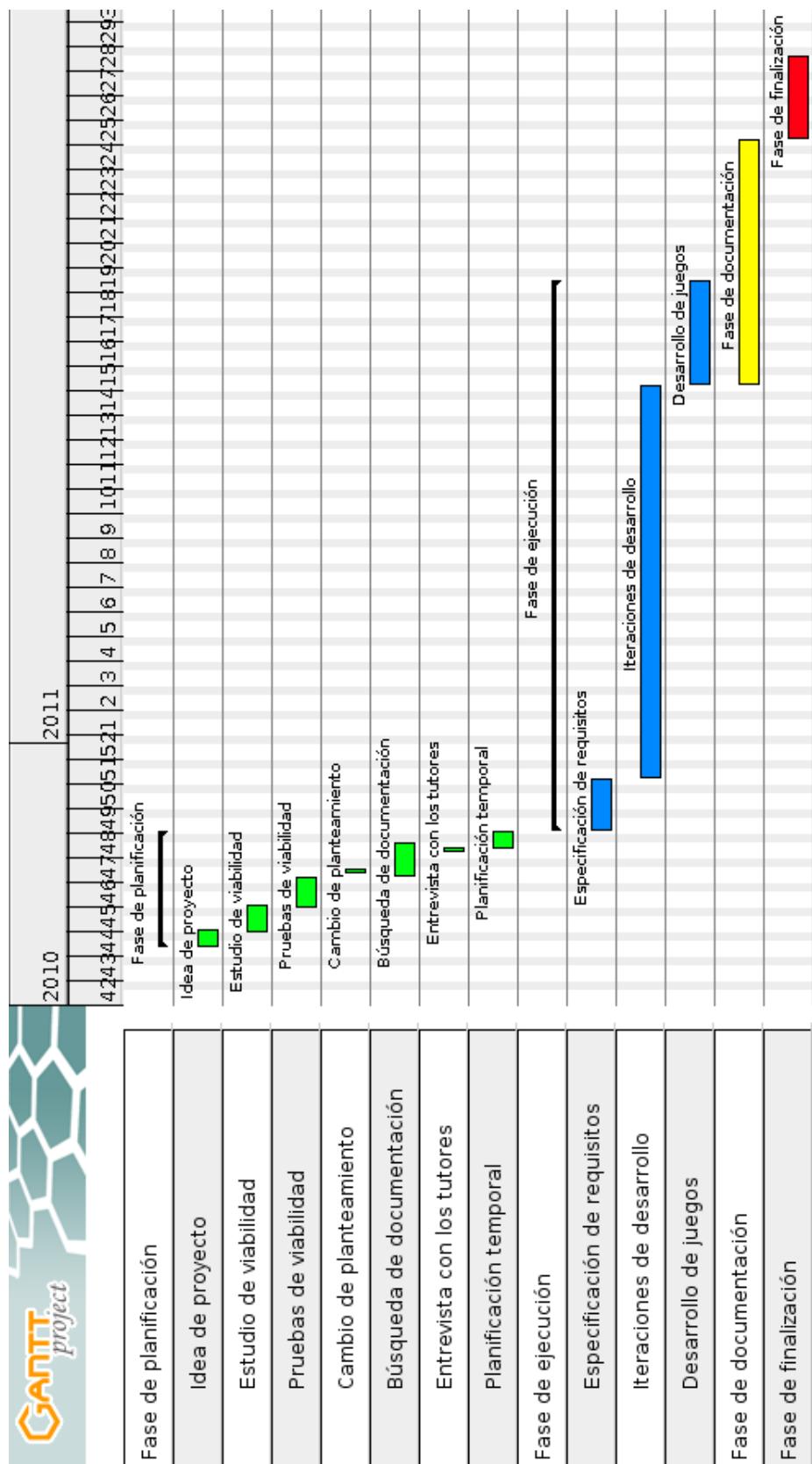


Figura 2.1: Diagrama de Gantt con la planificación del proyecto

Capítulo 3

Descripción general del proyecto

En este capítulo se profundiza en el proyecto, definiendo algunos de los aspectos más importantes de *LibWiiEsp*.

3.1. Perspectiva del producto

La biblioteca es un producto software independiente, es decir, no forma parte de ningún otro proyecto de mayor envergadura, aunque sí depende de las herramientas de bajo nivel proporcionadas por los *sceners*. La herramienta más importante de éstas, y sobre la que trabaja *LibWiiEsp*, es la biblioteca *libogc*. Esta biblioteca, desarrollada mediante ingeniería inversa, ofrece acceso a todo el hardware de Wii, pero realiza las operaciones a muy bajo nivel, resultando relativamente incómoda para el programador.

LibWiiEsp no necesita comunicarse con otros sistemas, y carece de interfaz gráfica de usuario. Las interfaces de que dispone son de tipo hardware, y consisten en el acceso a los distintos componentes de la videoconsola, como son los mandos (vía *Bluetooth*), la tarjeta SD para el almacenamiento, y sus distintos subsistemas.

Un programa desarrollado con *LibWiiEsp* podrá acceder al sistema gráfico de la consola, al de sonido, a la tarjeta SD y podrá leer el estado de hasta cuatro mandos conectados permanentemente al aparato. En su momento se decidió dejar para más adelante la posibilidad de acceder al sistema de *WiFi* de la videoconsola, así como a los puertos *USB* traseros y las ranuras para tarjetas de memoria de *Game Cube*; en futuras versiones se añadirán las estructuras necesarias para ello. De todas formas, sigue siendo posible acceder a estos subsistemas mediante las funciones que proporciona *libogc*.

Un detalle más a comentar, es que la biblioteca está pensada para hacer juegos en dos dimensiones, pero en una primera versión las plantillas pueden resultar incómodas para trabajar con un videojuego que no tenga utilice una perspectiva ortográfica.

3.2. Requisitos

A continuación se describen de forma general los requisitos que deberá cumplir la biblioteca para hacer posible el desarrollo de videojuegos completos.

3.2.1. Requisitos funcionales

En este apartado se describe la funcionalidad que incluye *LibWiiEsp*, mencionando los distintos módulos que se aportan al usuario:

- **Controlar el sistema de vídeo:** el aspecto más importante de un videojuego es su visualización, y hay que conseguir reproducir gráficos en la pantalla a partir de las escasas herramientas que proporciona *libogc*. Hay que facilitar el uso de texturas creadas a partir de una imagen y de formas geométricas con colores planos. Además, hay que proporcionar un sistema de doble *búffer* para evitar posibles problemas a la hora de dibujar gráficos y crear un mecanismo para finalizar cada fotograma, así como optimizar el rendimiento de todo el sistema gráfico de la videoconsola.
- **Acceder al lector de tarjetas SD:** se necesita garantizar el acceso a un medio de almacenamiento desde donde poder cargar los recursos y los datos que sean necesarios, en este caso, se ha seleccionado el lector de tarjetas SD por existir una mayor compatibilidad de las tarjetas con el software que permite ejecutar los videojuegos creados con *LibWiiEsp*, el *Homebrew Channel* [5].
- **Utilizar hasta cuatro mandos:** para no limitar la creación de los usuarios a juegos de un único jugador, es importante aportar la posibilidad de utilizar todos los mandos que llega a soportar Nintendo Wii, que son cuatro a la vez.
- **Utilizar recursos multimedia:** se proporcionará la posibilidad de cargar imágenes para crear texturas, efectos de sonido, pistas de música y fuentes de texto desde el medio de almacenamiento elegido, el lector de tarjetas. Una vez cargado en memoria el contenido multimedia, la utilización de éstos debe ser lo más sencilla posible. Ha de permitirse dibujar una textura o una parte de ella, reproducir varios efectos de sonido a la vez, reproducir una pista de música y escribir texto con cualquier carácter soportado por la fuente de texto seleccionada.
- **Organizar recursos multimedia:** todo el contenido multimedia debe organizarse de una forma que resulte sencilla, para facilitar el acceso a estos recursos desde cualquier punto de la aplicación.
- **Soporte de internacionalización (i18n):** en un mundo globalizado como el actual, es muy importante que una aplicación (en este caso, un videojuego) pueda obtenerse en múltiples idiomas, no sólo en el del creador. Se proporcionará un soporte de idiomas sencillo de utilizar, pero suficientemente potente y eficiente, además de facilitar la adición de idiomas nuevos.

- **Reproducir animaciones:** a partir de una textura que contenga un *spritesheet* se debe poder dibujar en la pantalla la animación determinada por los distintos cuadros de la rejilla de la imagen.
- **Registro de mensajes del sistema:** debido a las dificultades que entraña la depuración de código en la plataforma Nintendo Wii, será útil ofrecer un sistema de registro de mensajes con distintos niveles (avisos, errores...) para hacer menos compleja la detección de errores y creación de *logs* del sistema.
- **Detectar las colisiones:** otro aspecto importantísimo de la creación de videojuegos es la detección de colisiones. Para ello, *LibWiiEsp* tendrá un módulo especializado en detectar colisiones entre formas geométricas planas, de forma eficiente y fácilmente escalable.
- **Creación de actores:** en un videojuego, todos los elementos no estáticos son llamados actores, y aunque puede darse una gran variedad de comportamientos en ellos, hay varios aspectos comunes en todos. Se dará la posibilidad de utilizar una plantilla (una clase abstracta) que recoja todos estos detalles compartidos entre los distintos actores, y reduzca la complejidad de creación de éstos.
- **Diseño de escenarios:** tan importante como los actores es el escenario en el que se desarrolla el juego. Se debe facilitar la creación de escenarios basados en mapas de *tiles*, y que resulten sencillos de cargar en el programa.
- **Inicialización de la videoconsola:** todos los subsistemas de la consola que se emplean al ejecutar un videojuego necesitan ser inicializados y configurados antes de entrar en funcionamiento, así que se aportará una forma cómoda de realizar estas operaciones.

3.2.2. Requisitos de interfaces externas

Como ya se ha comentado, se necesita acceder al lector de tarjetas SD y montar la partición en la que se encuentran los recursos multimedia.

Por otro lado, es necesaria una separación efectiva de código fuente y datos, de tal manera que se puedan añadir niveles a un juego, modificar los parámetros de configuración, añadir idiomas nuevos y ajustar la información de los distintos actores que participan en el videojuego sin necesidad de recompilar cada vez que se requiera un cambio. Para ello, lo más adecuado es utilizar el formato XML, por lo que es imprescindible que *LibWiiEsp* ofrezca una manera sencilla y eficiente de trabajar con este tipo de archivos.

3.2.3. Requisitos de rendimiento

Respecto al rendimiento, es muy importante cuidar el rendimiento de los distintos módulos de la biblioteca, ya que, por una parte, el hardware de la Nintendo Wii es relativamente limitado en comparación con las otras dos videoconsolas de su misma generación (*Microsoft Xbox 360* y *Sony PlayStation 3*). Concretamente, Wii dispone

de un procesador de 729 MHz, 64 megabytes de memoria principal y 24 de memoria de vídeo, debido a ello hay que optimizar el máximo posible el producto.

3.2.4. Atributos del sistema software

Las características indispensables que deben conseguirse con el desarrollo del sistema son la fiabilidad, escalabilidad y mantenibilidad.

La biblioteca debe ser completamente fiable, ya que la depuración es un proceso bastante complejo en la Nintendo Wii. Hay que conseguir que todos los componentes de *LibWiiEsp* respondan con amplia fiabilidad, y no provoquen errores, de tal manera que el usuario pueda estar seguro de que es su programa el que produce el fallo, en caso de que lo hubiera.

Es deseable conseguir la característica de la escalabilidad, debido a que *LibWiiEsp* cubrirá una parte importante de las operaciones que realiza un videojuego en dos dimensiones, pero hay otras que no tocará. Estas partes son, entre otras, las conexiones de red y la detección de colisiones (siempre es bueno aumentar la cantidad de figuras de colisión). Para obtener esta propiedad, es necesario que desde un principio se guarde cuidado respecto a otra característica como es la mantenibilidad: si *LibWiiEsp* va a crecer con el tiempo (y espero que así sea) es muy importante que pueda ser mantenida y corregida de una manera sencilla.

3.3. Características de los usuarios

LibWiiEsp está orientada a usuarios con un mínimo conocimiento sobre el desarrollo de videojuegos en dos dimensiones, e incluso podría llegar a utilizarse como herramienta de iniciación en este campo, pero esto último no se recomienda. Por supuesto, es una herramienta con suficiente potencia como para desarrollar videojuegos de una complejidad media-alta, por lo que también es adecuada para programadores que tengan una amplia experiencia en el campo. Al estar implementada en C++ y utilizando el paradigma de la orientación a objetos, se aconseja a los posibles usuarios que adquieran previamente una base de conocimiento suficiente en los dos aspectos mencionados.

3.4. Restricciones generales

La programación para Nintendo Wii tiene implícitas ciertas restricciones relacionadas con la arquitectura hardware concreta de la videoconsola. A continuación se enumeran estas limitaciones:

- **Big Endian:** el procesador de Nintendo Wii trabaja con una representación de la información en memoria con *Big Endian*, al contrario que las plataformas de arquitectura Intel. Esto obliga a que se compruebe la estructura de toda información que se cargue desde un medio de almacenamiento externo y que se haya

tratado con un sistema *LittleEndian* (como por ejemplo, cualquier ordenador personal de hoy en día). Para una explicación gráfica sobre las diferencias de representación entre los distintos tipos de *Endian*, ver figura 3.1.

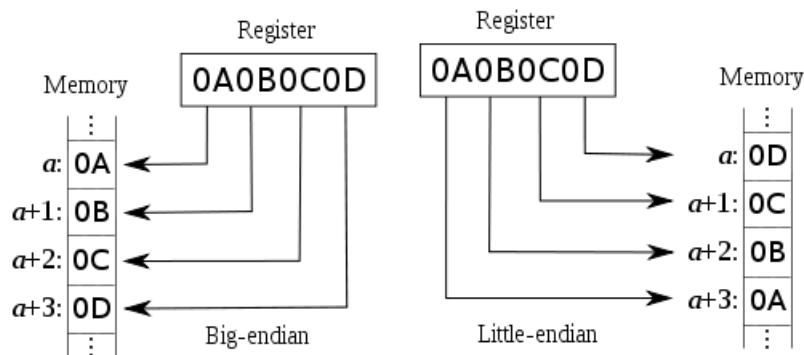


Figura 3.1: Diferencia entre las representaciones *Big Endian* y *Little Endian*

- **Alineación de los datos:** otra limitación referente a la arquitectura en la que está construida Wii consiste en que cada dato en memoria debe estar alineado a su tamaño. Así, un entero de 32 bits sólo podrá encontrarse en las posiciones 0, 4, 8, etc., un carácter (8 bits) podrá encontrarse en cualquier posición . . . Si, por ejemplo, reservamos memoria para un carácter, y justo después para un entero de 32 bits, entonces el procesador rellenará las tres posiciones de memoria que restan hasta la cuarta con ceros, produciéndose una pérdida de espacio considerable si no se cuida este detalle. En la figura 3.2 puede observarse el estado de la memoria si no se respeta esta indicación, comparando la situación con una declaración de variables adecuada.



Figura 3.2: Ejemplo sobre la necesidad de alineación de los datos

- **Relleno y alineación al leer desde un periférico:** el mecanismo de la caché de lectura desde periféricos que utiliza Nintendo Wii emplea un *búffer*

de 32 bytes, que requiere que la zona de memoria principal donde se escribe esté alineada a esta cantidad, y que además tenga un tamaño exactamente múltiplo de 32 bytes. De no respetarse esta limitación, se podrían producir errores de pérdida de datos por *machacamiento*, ya que la lectura se hace en bloques de 32 bytes alineados exactos, sin comprobar que esos 32 bytes pertenezcan al mismo fichero. Es decir, puede darse el caso (bastante común) de que la información de un archivo tenga su parte inicial en una zona de memoria, y el resto en otra distinta, lo cual provocaría un error.

- **Tipos de datos:** A la hora de programar en la consola Wii deben utilizarse los tipos de datos definidos por *libogc*, que incluyen variantes para todo tipo de datos numéricos (enteros con y sin signo, y decimales de coma flotante). Para el resto de datos (booleanos, caracteres, etc.) no existen redefiniciones de tipos.
- **Cantidad de memoria:** un aspecto importante a tener en cuenta a la hora de utilizar *LibWiiEsp* es la limitación de memoria que tiene la videoconsola. La Nintendo Wii dispone sólo de 64 megabytes de memoria principal, por lo que es obligatorio que el usuario tenga especial cuidado de no sobrecargar la consola (por ejemplo, cargando demasiadas pistas de música).

3.5. Suposiciones y dependencias

Toda la funcionalidad descrita anteriormente precisa de una serie de condiciones y dependencias que, si bien no son demasiadas, son todas necesarias para la construcción y ejecución de un videojuego desarrollado con *LibWiiEsp*.

En la propia Nintendo Wii se debe tener instalado algún método que permita la ejecución de código sin firmar digitalmente por el fabricante, siendo la mejor de las posibilidades *Homebrew Channel* [5]. Este software se instala como un canal en el menú principal de la consola, y permite correr ejecutables almacenados en la tarjeta SD.

Para poder utilizar hasta cuatro mandos, éstos deben estar sincronizados permanentemente con la consola; en caso contrario, sólo se reconocerá el mando principal asociado. Para más información sobre cómo sincronizar permanentemente un mando a Nintendo Wii, consultar el manual de la consola [10].

Respecto a la propia herramienta *LibWiiEsp*, ésta tiene varias dependencias de código. Concretamente son las bibliotecas *libogc* (proporciona acceso al hardware de Wii), *libfat* (permite montar y desmontar particiones en los sistemas de archivos FAT y FAT32), *TinyXML* (para operar con archivos de datos en formato XML) y *FreeType2* (proporciona una interfaz para poder trabajar con fuentes de texto). Todas excepto *libogc* necesitan estar portadas (compiladas específicamente) para trabajar con Nintendo Wii. Pueden obtenerse todas desde la forja del proyecto [6].

Otra herramienta, quizá la más importante, es un conjunto de compiladores y bibliotecas de C/C++, modificados para generar ejecutables que puedan ser lanzados en Nintendo Wii. Este conjunto de herramientas y utilidades varias se llama *DevKitPPC*, y se encuentra englobado dentro de un proyecto denominado *DevKit-Pro* [17] que incluye este paquete y otros dos similares para la programación en *Sony PlayStation Portable* y *Nintendo DS*. *LibWiiEsp* utiliza la versión *r21* de este paquete de herramientas.

Capítulo 4

Desarrollo del proyecto

LibWiiEsp ha sido construida siguiendo una metodología iterativa basada en análisis, diseño, implementación y pruebas para cada requisito, consiguiéndose tras cada iteración un módulo totalmente funcional. En este capítulo se desarrolla todo el proceso separado en fases, abordando en cada una de ellas el proceso completo realizado para cubrir cada uno de los requisitos indicados en la descripción general del proyecto.

Cada iteración, o fase, consta de los cuatro puntos principales ya mencionados, donde en el análisis se describe cómo trabaja *libogc* a bajo nivel y qué es lo que se pretende conseguir, el diseño profundiza en el cómo se ha conseguido cubrir el requisito, en el apartado de implementación se indican los detalles más relevantes sobre la construcción del módulo, y en el epígrafe de pruebas se recopilan las diversas pruebas realizadas a cada módulo, tanto individuales, como de cohesión con los demás módulos anteriores.

A pesar de que en este capítulo se mencionan las pruebas individuales aplicadas a cada uno de los módulos de *LibWiiEsp*, en el siguiente se recoge la documentación completa sobre el plan de pruebas de la biblioteca. Para ampliar información sobre el funcionamiento y los parámetros de las clases que componen *LibWiiEsp*, consultar los manuales de la biblioteca.

Debido a la falta de material de consulta en forma de bibliografía, toda la información sobre el funcionamiento de Nintendo Wii ha sido recopilada a partir de un buen número de fuentes localizadas a través de la red [7] [14] [15] [16].

4.1. Diagramas del sistema

En la figura 4.1 se puede observar el diagrama de clases completo del sistema y en la figura 4.2 el diagrama de componentes del sistema. Ambos diagramas representan el estado del sistema (a nivel lógico interno y a nivel físico de los archivos de cabeceras) tras finalizar el desarrollo.

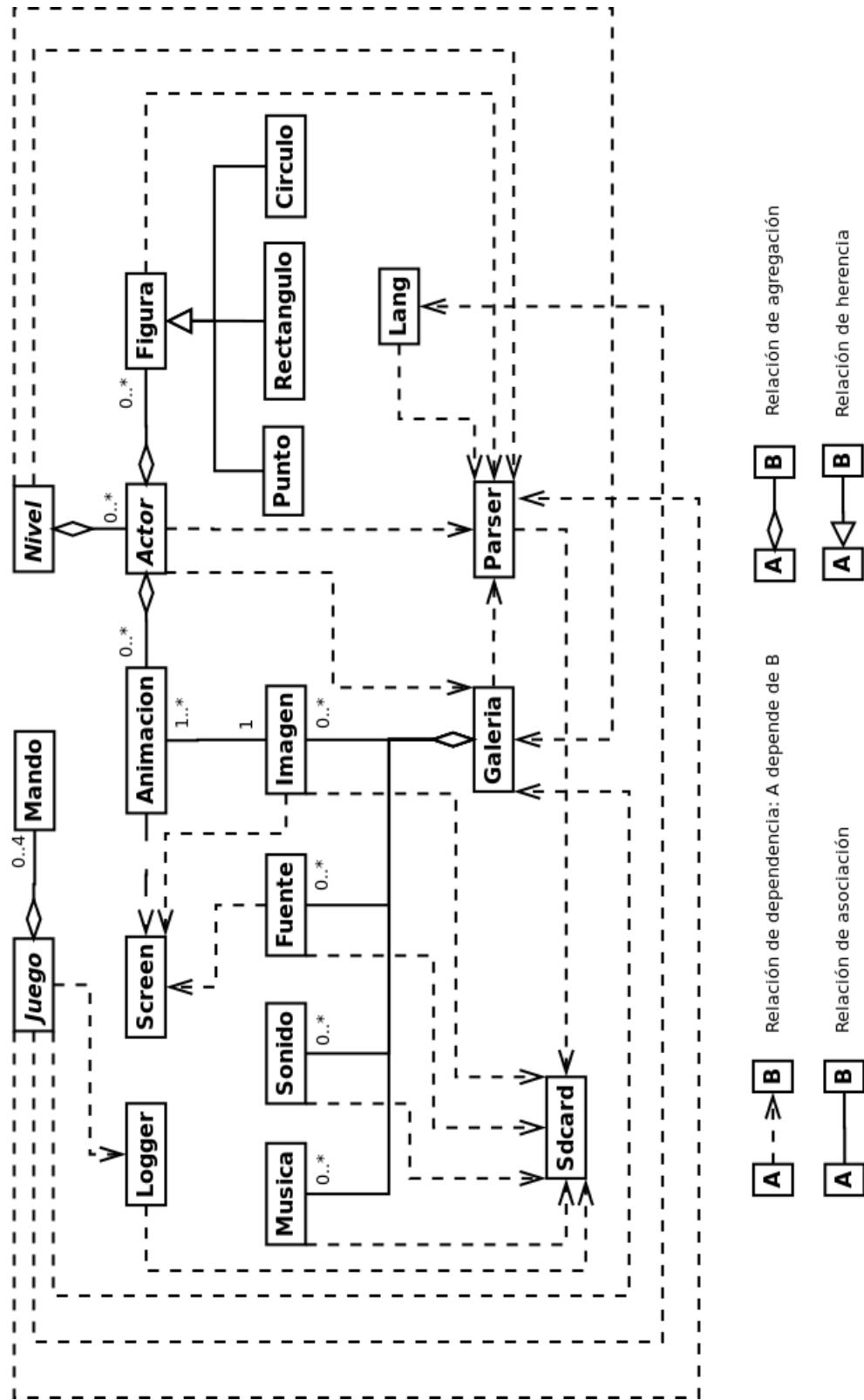


Figura 4.1: Diagrama de clases del sistema

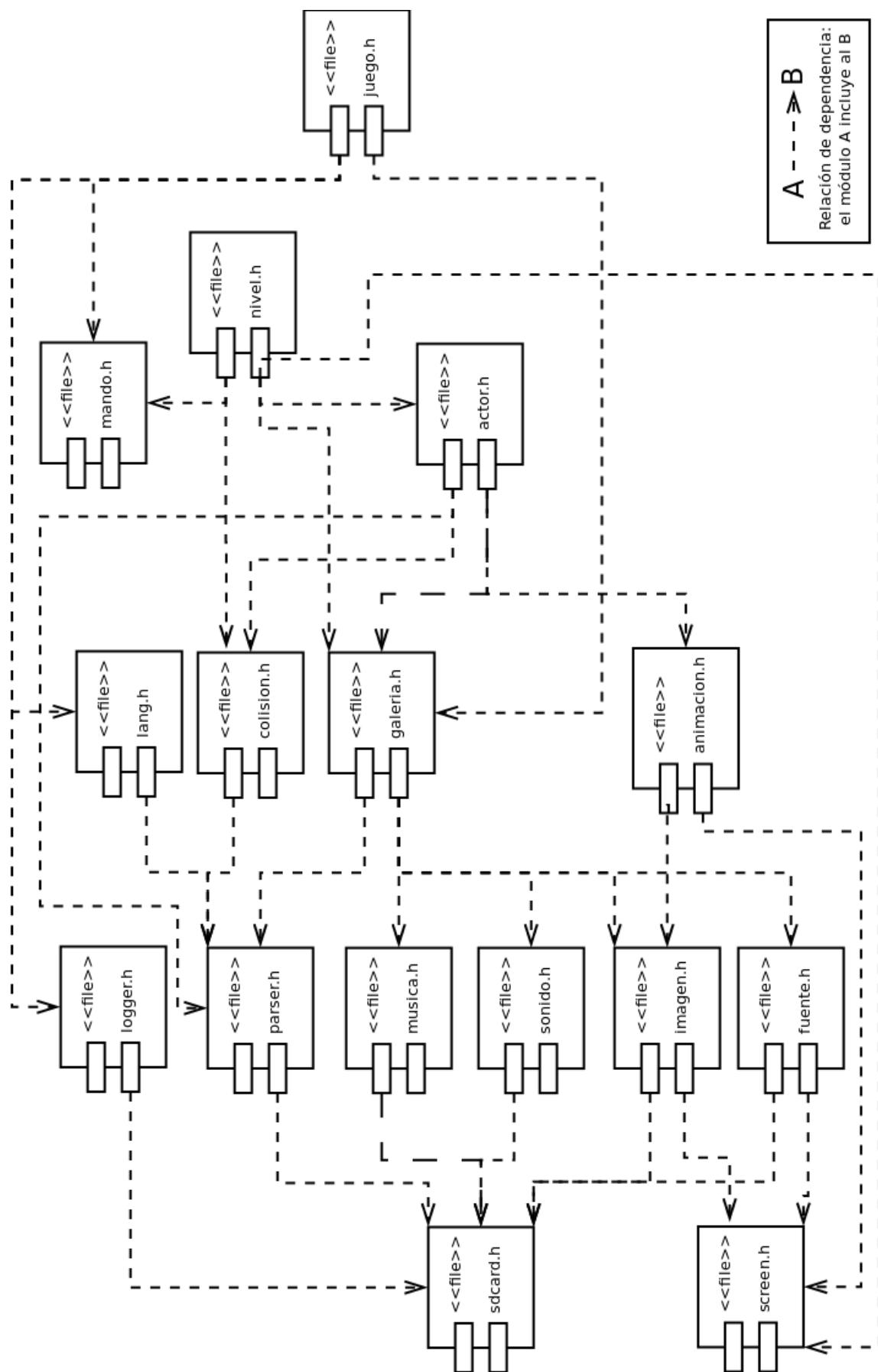


Figura 4.2: Diagrama de componentes del sistema

4.2. Sistema de vídeo

El primer paso para abordar el desarrollo de *LibWiiEsp* fue investigar sobre la forma de trabajar del sistema gráfico de la videoconsola. Como ya se ha comentado, la base del proyecto será *libogc* [12], una biblioteca de bajo nivel que permite acceder a prácticamente todo el hardware de Nintendo Wii.

4.2.1. Módulos *video* y *GX*

Libogc trabaja gráficamente con dos módulos, llamados *video* y *GX*. El módulo *video* es el que controla las funciones básicas del chip gráfico de la consola, encargándose de detectar el modo de vídeo, enviar los datos que se quieren dibujar al bus de la *GPU* y esperar la sincronización vertical. Este componente requiere para trabajar un *búffer* de información situado en la memoria principal, y que debe almacenar, en cada fotograma de un programa, los datos que se quieren dibujar en la pantalla.

Sobre el módulo *video* que controla directamente el hardware, opera el otro componente, la librería *GX*, la cual ofrece muchas más posibilidades de trabajo. En concreto, proporciona una serie de estructuras y funciones que facilitan enormemente el trabajo con el sistema de video, siendo las más interesantes para nuestro objetivo las siguientes:

- **Dibujo de primitivas:** la *GX* trabaja con una serie de primitivas de dibujo, que son polígonos que se pueden dibujar en el *búffer* del sistema básico de vídeo. Cada uno de estos polígonos se dibuja a partir de un número concreto de vértices; por ejemplo, para dibujar un triángulo son necesarios tres vértices, para una línea dos, y así. Las primitivas más útiles para nosotros serán el punto (*GX_POINTS*), la línea recta (*GX_LINES*), el triángulo (*GX_TRIANGLES*) y el rectángulo (*GX_QUADS*).
- **Trabajo con texturas en crudo y paletas:** una textura no es más que una imagen preparada para ser dibujada dentro de un polígono que tenga área (en el caso de las primitivas antes mencionadas, triángulos y rectángulos). Esta imagen puede tener su información de forma explícita (color directo) o implícita (cada píxel es una referencia a un color de la paleta de colores del sistema). *GX* proporciona las estructuras de datos necesarias para almacenar la información de los colores, texturas y paletas, aunque el trabajo directo con ellas es un tanto engoroso.
- **Formatos de texturas:** se soportan varios formatos de textura, cada uno de los cuales tiene sus ventajas e inconvenientes, pero hay uno de ellos con el que el procesador de Wii trabaja de forma nativa: este formato es RGB5A3, que utiliza 16 bits para cada píxel y da la posibilidad de trabajar con el canal *alpha* para las transparencias. Concretamente, RGB5A3 representa un píxel con 5 bits para cada canal de color y uno para *alpha*, siendo determinada la componente del rojo por los 5 bits de mayor peso, el verde por los 5 siguientes y el azul por los 5 bits, dejando el de menor peso para indicar el canal *alpha*.

Sin embargo, cuando el bit que identifica al canal *alpha* tiene el valor 1 (*alpha* activado), se utilizan 4 bits para cada canal, siendo los cuatro de menor peso los que determinan el nivel de transparencia (es decir, la distribución de los bits con el canal *alpha* activado sería RRRRGGGGBBBAAA (*red* o rojo, *green* o verde, *blue* o azul, *alpha*), con el rojo situado en los cuatro bits de mayor peso). Este sistema, además de ser el más eficiente para trabajar con la *GPU* de Nintendo Wii, proporciona una flexibilidad enorme respecto al trabajo con transparencias. Por otra parte, se observa que la construcción de texturas se realiza a partir de una imagen organizadas en *tiles* de 4x4 píxeles.

- **Sistema de descriptores:** antes de realizar cualquier operación de dibujo, *GX* espera recibir una serie de parámetros en los que se indiquen qué se va a dibujar; estos parámetros reciben el nombre de descriptores. Entre otras muchas funciones, los descriptores sirven para indicar si se va a dibujar una primitiva con color directo o rellena con una textura, la proyección que se va a utilizar a la hora de visualizar la imagen en la pantalla y el orden y tipo de los parámetros de los vértices.
- **Operaciones directas sobre la *GPU*:** el resto de operaciones de interés son relativas a la finalización de escritura en el *búffer* gráfico, el volcado de datos desde éste hacia el *EFB* del chip gráfico de la videoconsola y la fijación de estos datos en este *búffer* interno de la *GPU*, operación que evita el posible *machacamiento* de información en caso de sobrecarga del sistema gráfico.

Un detalle más, *GX* utiliza como formato para el color una variable entera de 32 bits sin signos, donde cada 8 bits representan la componente de un canal de la imagen. La distribución hexadecimal es 0xRRGGBBAA, siendo RR el byte que corresponde al color rojo, GG los 8 bits para el verde, BB la componente azul y AA el canal *alpha*.

4.2.2. Identificación de funcionalidad necesaria

Una vez reunida toda esta información, y teniendo claro qué estructuras y funciones de *GX* y *video* nos serán útiles, procedemos a identificar qué queremos conseguir para trabajar de una forma cómoda con el sistema gráfico de la consola.

Lo primero que tenemos claro es que se necesita controlar la inicialización del sistema gráfico de Wii, preparando ambos módulos (*video* y *GX*) para que trabajen de acuerdo a nuestras necesidades, pero abstrandéndonos de todos los mecanismos implicados en este proceso. También es necesario un mecanismo que permita dar por finalizado un fotograma y enviarlo a la *GPU* para que procese la información de éste y la dibuje en la pantalla, y que además enviara nueva información al *EFB* únicamente si existieran cambios en los gráficos a dibujar (de este modo, se consigue evitar la sobrecarga innecesaria del chip gráfico). Sería conveniente utilizar un sistema de doble *búffer*, es decir, alternar entre dos flujos de datos la hora de dibujar los gráficos y pasarlo al *EFB*, de tal manera que evitariamos posibles problemas de

parpadeos al dibujar en la pantalla.

Por otro lado, nos interesa realizar el procesamiento necesario para crear una textura a partir de la información de una imagen que se encuentre en memoria, organizando los bits que componen dicha imagen en *tiles* de 4x4, y también sería adecuado abstraer al usuario de los mecanismos implicados en el trabajo con los descriptores de *GX*.

Como tercer y último bloque de funcionalidad, se encuentra el objetivo de dibujar en la pantalla texturas previamente procesadas, tanto de forma completa como únicamente una parte de ellas. Además, se dará la oportunidad de dibujar las siguientes formas geométricas llenas de un color plano: un punto (o píxel), una recta con anchura, un rectángulo determinado por cuatro puntos y un círculo.

Por supuesto, todas las operaciones aquí descritas deberán ser eficientes, ya que no interesa sobrecargar el procesador gráfico de la videoconsola.

4.2.3. Diseño e implementación

Como resultado de toda la información concretada en los dos puntos anteriores, se llega a la conclusión de que lo más apropiado para satisfacer el requisito *Controlar el sistema de vídeo* es construir una clase que implemente el patrón *Singleton*, es decir, que únicamente exista una instancia de esta clase en el sistema, y que sea accesible desde cualquier punto de la aplicación. La decisión sobre la utilización de este patrón de diseño se basó, principalmente, en que la Nintendo Wii trabaja con sólo una pantalla, y por tanto no tendría sentido que coexistieran en memoria varios objetos de la clase.

En esta clase, debería existir un método que se encargue de inicializar los módulos de *libogc* descritos con anterioridad. El proceso de inicialización debe establecer el modo de vídeo, crear los dos flujos de datos que conformarían el sistema de doble *búffer*, configurarlos para que trabajen con el modo de vídeo detectado, e indicar a la *GPU* dicho modo de vídeo. A continuación, debe reservarse una zona de memoria, alineada a 32 bytes, que se utilizará para copiar el contenido del búffer activo en cada fotograma al *EFB*. En la figura 4.3 puede observarse un esquema de cómo queda la organización de los flujos de datos del sistema de doble *búffer*.

Justo después se inicializará la librería *GX*, aportándole los parámetros de configuración necesarios para establecer una proyección ortográfica, calcular la resolución de pantalla, el tratamiento de color y el canal *alpha*, y otros detalles más.

Para evitar que se envíe al chip gráfico información que no cambia de un fotograma al siguiente (y por tanto, conseguir una mayor eficiencia al procesar los gráficos únicamente cuando hay cambios en el *EFB*) nuestra clase necesita una bandera interna, representada como una variable de tipo booleano, que indique si hay cambios

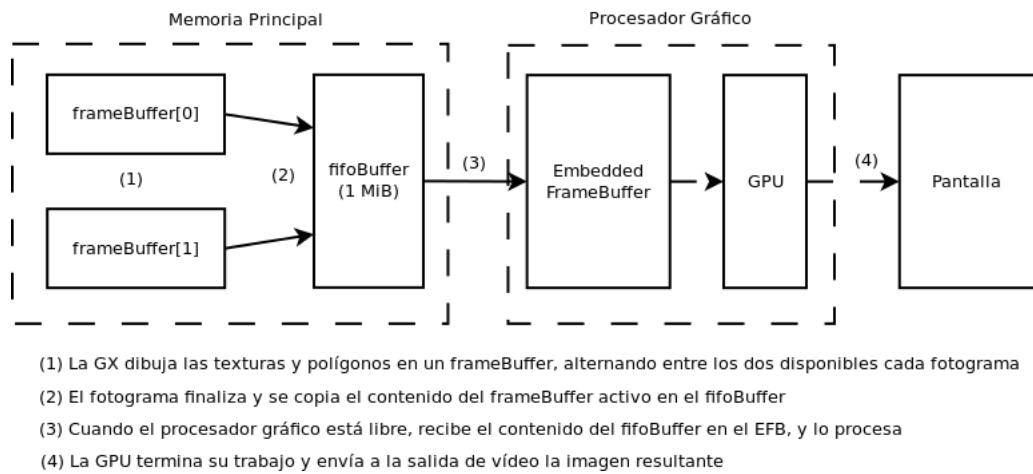


Figura 4.3: Esquema que ilustra el funcionamiento del sistema gráfico de Wii

respecto al estado anterior del *EFB*. Esta bandera se activará cuando se utilice cualquier método de dibujo, y se desactivará en el método que se encarga de enviar los datos a la *GPU*.

Una vez implementados los métodos de inicialización del sistema gráfico y de finalización de fotograma, pasamos al siguiente punto, el trabajo con texturas. Se necesita un método que, a partir de una zona de memoria que contenga información de una imagen en formato RGB5A3, se encargue de crear una textura organizada en *tiles* de 4x4 e inicialice una variable que contenga la estructura que necesita *GX* para trabajar con ella.

A continuación, se escriben los métodos privados de la clase que abstraen el trabajo de los descriptores, indicando uno de ellos los descriptores necesarios para utilizar color directo, y el otro para utilizar una textura previamente procesada.

Por último, se crean los métodos de dibujo que necesitamos para cubrir la funcionalidad identificada. El dibujo de una textura es trivial, ya que consiste en dibujar un rectángulo (a partir de cuatro puntos) relleno con la textura, lo cual se consigue ajustando los descriptores. El punto se consigue dibujando un píxel de color, la recta con anchura, con dos triángulos rectángulos que tengan la hipotenusa común, y el rectángulo relleno de color es parecido a dibujar una textura, pero ajustando los descriptores de *GX* para que en lugar de utilizar ésta, se rellene con un color liso. Para dibujar parte de una textura, se delimita dicha parte con un rectángulo indicado por un punto sobre la textura, y un ancho y un alto en píxeles, siendo las coordenadas del punto relativas a la esquina superior izquierda de la textura original. Por último, para dibujar un círculo relleno de color, se utilizan 32 triángulos que comparten un vértice (que coincide con el centro del círculo).

La interfaz pública de la clase puede observarse en la figura 4.4.

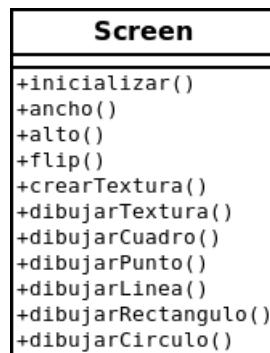


Figura 4.4: Interfaz pública de la clase Screen

4.2.4. Pruebas

La batería de pruebas diseñada para esta clase en particular consta de dos grupos de comprobaciones: por un lado, había que probar que el sistema gráfico se inicializa, trabaja correctamente con el sistema de doble *búffer*, se crean bien las texturas a partir de una imagen cargada en memoria con formato RGB5A3 y que se finalizan adecuadamente los fotogramas. El otro conjunto de pruebas iba dirigido a confirmar que los métodos de dibujo realizan bien su trabajo.

La inicialización del sistema gráfico, el trabajo con el doble *búffer* y la finalización de fotogramas se comprobaron utilizando el subsistema de consola que incorpora *libogc*, imprimiendo mensajes en la pantalla a medida que se completaban las operaciones.

Por otro lado, los métodos de dibujo se probaron directamente, una vez validada la funcionalidad descrita en el párrafo anterior. Sin embargo, cabe destacar que, al no disponer aún de acceso a la tarjeta SD para cargar imágenes, las pruebas de creación de texturas y dibujo de las mismas tuvieron que realizarse con la utilidad *raw2c* (incluida en *DevKitPPC* [17]), un pequeño software que transforma cualquier archivo binario (imágenes, efectos de sonido, etc.) en un archivo de cabecera de C con toda su información en forma de *array* de datos. De esta manera, y a partir de una imagen BMP sin compresión, se diseñó un pequeño programa que creaba la textura a partir del flujo de datos generado por *raw2c* y la dibujaba en la pantalla, tanto completa como una parte de ella.

4.3. Acceso al lector de tarjetas

Una vez cubierto el requisito de controlar el sistema de vídeo, era momento de investigar sobre el funcionamiento del lector de tarjetas de Nintendo Wii.

4.3.1. Módulo *wiisd_io* y *libFat*

Libogc proporciona acceso al dispositivo hardware del lector de tarjetas a través de la variable global `__io_wiisd`, la cual está incluida en el fichero de cabecera `sdcard/wiisd_io.h`. Esta variable global es una instancia de la estructura `DISC_INTERFACE`, incluida en la cabecera `ogc/disc_io.h`, y que se utiliza como interfaz única para todos los dispositivos de almacenamiento de los que dispone la videoconsola. La estructura está formada por dos variables que contienen información (indican el tipo de dispositivo) y otras seis, que son punteros a función. Estas seis funciones se definen para cada tipo de dispositivo, y son las siguientes:

- **startup**: inicializa el dispositivo.
- **isInserted**: comprueba si el medio de almacenamiento está disponible en el lector correspondiente.
- **readSectors**: lee un grupo de sectores desde el dispositivo a un flujo de datos.
- **writeSectors**: escribe un flujo de datos en un grupo de sectores.
- **clearStatus**: indica si el dispositivo está listo para realizar una operación.
- **shutdown**: apaga el dispositivo.

Cuando el dispositivo esté listo para ser accedido, hay que montar la partición en la que se encuentre la información que se quiere cargar. Las tarjetas de memoria SD trabajan con el sistema de ficheros FAT o FAT32, por lo que es necesario utilizar *libFat* adaptada para trabajar con Nintendo Wii. La interfaz de esta biblioteca es muy sencilla, ya que se puede montar un sistema de ficheros FAT con la función `fatMountSimple`, que recibe como parámetros el nombre que queremos asignarle a la unidad montada, y la estructura `DISC_INTERFACE` correspondiente al dispositivo (en el caso de la tarjeta SD será, como se comentó anteriormente, la variable `__io_wiisd`). Desmontar una partición que esté montada es tan sencillo como llamar a la función `fatUnmount`, indicándole el nombre de la unidad montada.

Hay que tener en cuenta una limitación de este sistema, y es que sólo es accesible la primera partición de la tarjeta SD; por tanto, se debe tener especial cuidado de que la tarjeta tenga su primera partición formateada con FAT o FAT32.

4.3.2. Identificación de funcionalidad necesaria

Tras revisar toda la documentación y las cabeceras de los archivos implicados, hay que definir de una forma clara qué se quiere conseguir para cumplir el requisito de acceder al lector de tarjetas. Básicamente, se necesita poder montar una partición FAT/FAT32 que se encuentre en la tarjeta SD, asegurando que quede accesible desde cualquier punto de la aplicación, y también poder desmontar esta partición.

Por otro lado, no estaría de más poder conocer en todo momento si la partición está montada o no, y el nombre asignado que tiene la unidad montada.

4.3.3. Diseño e implementación

Al igual que ocurre con el sistema gráfico de la videoconsola, lo más adecuado para tratar el acceso al lector de tarjetas es implementar una clase con el patrón *Singleton*, ya que sólo existe en el hardware un lector de tarjetas (al contrario ocurre con los dos puertos USB traseros o las dos ranuras para tarjetas de memoria de *Game Cube*, pero no es el caso que nos atañe). Por otro lado, la implementación del mencionado patrón debe dejar accesible la instancia de la clase desde cualquier punto del sistema.

La clase tendrá un método de inicialización, en el que reciba el nombre que se quiere asignar a la unidad una vez montada. Dicho método intentará inicializar el dispositivo del lector de tarjetas (utilizando la función *startup* de la estructura descrita antes) y, en caso de tener éxito la operación, tratará de montar la primera partición que se encuentre en la tarjeta SD con un sistema de ficheros FAT. Si algo saliera mal, el proceso se repetirá hasta 10 veces. Superado ese número de intentos sin éxito, el programa interrumpiría su ejecución, saliendo con un código de error 1.

Además, la clase contará con métodos consultores sobre una bandera (variable booleana) que indique si la partición está montada o no, y sobre el nombre asignado a la unidad. Por último, se proporcionará también un método que intente desmontar la partición, en caso de que se encuentre activa.

Para acceder al sistema de ficheros montado mediante esta clase, basta con utilizar las funciones estándar de lectura y escritura de archivos. Es recomendable que siempre se empleen rutas absolutas, precedidas del nombre de la unidad seguida de los dos puntos (:). Por ejemplo, sabiendo que la unidad se llama "SD", para cargar un archivo cuya ruta sea */directorio/archivo.bmp* se podría utilizar el siguiente código:

```

1 |     ifstream archivo;
2 |     archivo.open("SD:/directorio/archivo.bmp", ios::binary);

```

Por último, en la figura 4.5 puede observarse la interfaz pública de la clase:



Figura 4.5: Interfaz pública de la clase Sdcard

4.3.4. Pruebas

Las pruebas realizadas para validar esta clase fueron sencillas, ya que consistieron en un programa que utilizaba el subsistema de consola de *libogc* para ir imprimiendo mensajes a medida que se intentaba acceder al hardware del lector de tarjetas y, una vez activado, realizar el montaje y desmontaje de la partición. También se incluyó en el programa de prueba la apertura y escritura en un archivo de texto plano.

4.4. Los mandos

En la tercera iteración del proceso de desarrollo del proyecto se entra de lleno en la gestión de los dispositivos de entrada, es decir, los mandos. La mayor innovación de Nintendo Wii respecto a las demás videoconsolas fue precisamente su revolucionario sistema de control de los juegos, denominado *Wii Remote* (o, coloquialmente, *wiimote*), que permitían ir mucho más allá de participar en el videojuego apretando botones.

4.4.1. Módulo *Wpad*

A la hora de gestionar los mandos de la consola, *libogc* utiliza el módulo llamado *Wpad*, que brinda acceso a las estructuras de control y las funciones dedicadas a leer el estado de los controles. Cada *wiimote* que quiera utilizarse, como ya se comenta en el capítulo de descripción general de este documento, debe estar sincronizado permanentemente a la videoconsola.

Wpad asigna a cada mando que esté conectado una estructura de tipo *WPAD-Data*, en la cual se encuentra toda la información relacionada con su estado en un momento dado. Cada *wiimote* se identifica mediante un número entero, denominado *chan*, que se genera automáticamente según el orden en el que se conectan los mandos (el primero tendrá *chan* cero, el segundo *chan* uno, y así).

La información sobre la pulsación de los botones se obtiene con variables de tipo entero de 32 bits sin signo, en la que cada dígito binario del entero representa el estado de un botón. Para saber si un botón concreto está pulsado basta con comparar (a nivel de bit, con un AND) la variable correspondiente con el valor binario único del botón (ver figura 4.6). Existen cuatro variables que indican si los botones acaban de ser pulsados, si se están manteniendo pulsados, si se acaban de soltar, o si no están pulsados. Por otro lado, hay variables también numéricas que indican el estado de la batería (menor carga cuanto menor es la cifra) o si el mando tiene alguna nueva información pendiente de recibir (normalmente, efectos de sonido para reproducir en el altavoz del *wiimote*).

Los últimos elementos que constituyen esta estructura son, a su vez, cinco estructuras más que almacenan los datos de orientación del mando (ángulos de viraje, cabeceo y rotación), de los acelerómetros, la fuerza con la que se mueve, las coorde-

```
00000000000000101000001000010000
```

```
0000000000000000000000000000000010000
```

La primera variable es la contenida en `WPADDData`.

La segunda es el valor de pulsación de un botón.

Se comprueba la pulsación mediante una operación AND

Figura 4.6: Operación AND para conocer si hay pulsación de un botón

nadas en pantalla del puntero infrarrojo y la expansión conectada al *wiimote*.

Respecto a los ángulos que determinan la orientación del mando, el cabeceo es el ángulo que forma éste respecto a un plano horizontal y perpendicular a la pantalla si se le hace girar sobre el eje X, el viraje es el ángulo formado por el *wiimote* respecto a un plano vertical y perpendicular a la pantalla si se le hace girar sobre el eje Z, y la rotación es el ángulo que mide el giro de éste sobre el eje Y, como si estuviera dando una vuelta de campana. En la figura 4.7 puede apreciarse un esquema donde se describen gráficamente estos ángulos.

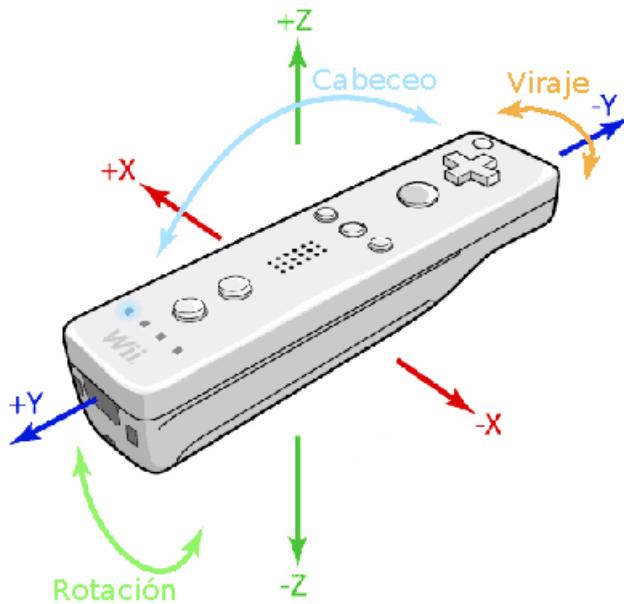


Figura 4.7: Ángulos de giro en la orientación de un *Wii Remote*

En el módulo *Wpad* se definen varias funciones que permiten trabajar con la estructura `WPADDData`, siendo la más importante de ellas `WPAD_ScanPads`, que lee el estado de todos los mandos conectados y actualiza la información de cada uno en su respectiva estructura. Por otro lado, existen funciones consultoras para conocer

el estado de pulsación de los botones, pero no para el resto de información relativa a los *wiimotes*, así pues, hay que acceder directamente a la estructura para conocer la orientación del mando, el puntero infrarrojo, etc.

El resto de funciones del módulo permite activar o desactivar la vibración, inicializar o desconectar el sistema de *Bluetooth* con el que funcionan los mandos, desactivar uno de éstos conociendo su número identificador o enviar un flujo de datos con un efecto de sonido para reproducirlo en el altavoz del *wiimote*.

4.4.2. Identificación de funcionalidad necesaria

A partir de toda la información recopilada en el epígrafe anterior, se debe decidir qué funcionalidad se quiere obtener para cubrir el requisito de *Utilizar hasta cuatro mandos*.

Aunque sería muy interesante proporcionar control sobre todos los aspectos del *wiimote*, existe un problema conocido en la lectura de los acelerómetros utilizando el módulo *Wpad*, por lo que se descarta, de momento, esta funcionalidad. Igualmente, la utilización de la *Wii Balance Board* (más conocida como la tabla de *Wii Fit*), la guitarra de títulos como *Guitar Hero* o *Rock Band*, el *Wii Motion Plus* y el mando clásico podrían aportar inmensas posibilidades a *LibWiiEsp*, pero al no disponer de estos periféricos no puedo desarrollar un módulo completo para ello, por tanto, se descarta su uso. Sin embargo, sí puedo aportar la funcionalidad relativa a la extensión del *wiimote* por excelencia, el *Nunchuck*.

Recapitulando, el módulo de gestión de entrada de la biblioteca va a permitir conectar hasta cuatro *Wii Remotes* simultáneos, cada uno con su respectiva expansión *Nunchuck*. Se van a mapear todos los botones de ambos periféricos, para conocer tanto si se encuentran pulsados o no, si se acaban de presionar o si se acaban de soltar en un momento determinado.

Respecto a las características especiales de los mandos de la videoconsola, se ofrecerán métodos consultores sobre la orientación de los mandos y sobre la situación del puntero infrarrojo sobre la pantalla, quedando el acceso a la información de los acelerómetros pendiente para una futura versión (cuando una actualización del módulo *Wpad* consiga que la lectura de dicha información sea correcta). Se controlará además el estado de la palanca del *Nunchuck*, y la vibración del *wiimote*.

4.4.3. Diseño e implementación

Una vez fijados los objetivos a conseguir, se crea una clase que gestione todos los aspectos relativos a la gestión de los *wiimotes*. La clase debe tener un puntero a la estructura de tipo *WPADDData* del mando que tendrá asociado cada instancia, además de almacenar el *chan* (número identificador único para el dispositivo), una variable que indique la expansión conectada y un diccionario en el que se guarde el mapeo de los botones, tanto del propio *wiimote* como del *Nunchuck*. El sistema de

Bluetooth bajo el que trabajan los mandos con la videoconsola tiene que inicializarse antes de poder usarse, así que se crea un método de clase que se encargue de ello.

La identificación de cada botón se realiza mediante un tipo enumerado que se utiliza como clave en el diccionario de botones. A cada una de estas claves se le asigna el valor binario de su botón correspondiente. Por otra parte, se crean dos *arrays* de valores booleanos, del mismo tamaño que el diccionario, y que guardarán el estado actual de todos los botones (pulsado o no), y el estado anterior. De esta forma se podrá identificar si se acaba de pulsar o soltar un botón.

A la hora de actualizar el estado del *wiimote*, se definen dos funciones. Una de ellas será un método de clase que realice la lectura y actualización de la estructura *WPADData* de todos los controles conectados (no se puede hacer de forma individual), y la otra función obtendrá, a partir del contenido de la estructura actualizada, toda la información relativa al mando que esté asociado con la instancia concreta de la clase. En este último método se comprueba también el estado de pulsación de los botones, refrescando la información de los *arrays* de datos booleanos.

Prácticamente la totalidad del resto de métodos son de consulta, tomando la información desde los *arrays* de booleanos (los métodos que comprueban el estado de un botón concreto) o bien desde la estructura *WPADData* (orientación, puntero infrarrojo, palanca del *Nunchuck*, si el mando está conectado o no, si el *Nunchuck* está conectado...). El único método restante no observador es el que activa la vibración del *wiimote* durante la cantidad de microsegundos que se le indique.

Un detalle a mencionar es la implementación de los métodos relacionados con el *Nunchuck*, ya que se ofrece el valor crudo de los ejes de la palanca (son enteros sin signo, de 8 bits), donde el centro es, aproximadamente, un valor de 128. Pero ocurre que esto no es exacto, y por eso se proporcionan dos métodos (uno para cada eje) que indican si la palanca está pulsada hacia arriba o abajo, hacia la derecha o la izquierda, o si está en centrada en el eje.

Por último, mencionar que no se permite la copia ni asignación de una instancia, debido a que cada una está ligada a un único *wiimote*, y no es posible asignar un mismo dispositivo a distintos objetos de la clase, ni utilizar una misma instancia para controlar más de un mando.

En la figura 4.8 se muestra la interfaz pública de la clase Mando.

4.4.4. Pruebas

Comprobar la detección de pulsaciones de los botones fue sencillo, se hizo a partir de un programa que escribía en el sistema de consola que aporta *libogc* un mensaje distinto según el botón que se pulsaba. Los métodos observadores se validaron im-

Mando
<pre>+Mando() +~Mando() +static leerDatos() +static inicializar() +conectado() +chan() +actualizar() +pressed() +newPressed() +released() +punteroX() +punteroY() +punteroEnPantalla() +vibrar() +cabeceo() +viraje() +rotacion() +nunConectado() +nunPalancaEstadoX() +nunPalancaEstadoY() +nunPalancaValorX() +nunPalancaValorY() +nunPalancaCentroX() +nunPalancaCentroY()</pre>

Figura 4.8: Interfaz pública de la clase Mando

primiendo directamente en la pantalla los valores que devolvían, y comprobando que fueran correctos. En cambio, para probar los métodos observadores para el estado del puntero infrarrojo sobre la pantalla, se utilizó el método *dibujarPunto* de la clase *Screen* con las coordenadas que devolvían las mencionadas funciones.

4.5. Recursos multimedia

Tras obtener la funcionalidad necesaria para gestionar el sistema gráfico de Nintendo Wii, el lector de tarjetas y el soporte para hasta cuatro mandos, es el momento de trabajar en la carga y funcionamiento de los recursos multimedia. En este punto es especialmente necesario que se optimice al máximo, ya que de no hacerlo podemos provocar que los tiempos de carga se alarguen demasiado.

Tal y como ya se ha comentado, *LibWiiEsp* ofrece soporte para utilizar imágenes, efectos de sonido, pistas de música y fuentes de texto.

4.5.1. Módulos *asndlib*, *libmad* y biblioteca *FreeType2*

Como primer detalle a tener en cuenta, sabemos que la clase *Sdcard* nos permite acceder a cualquier archivo que se encuentre en la tarjeta SD insertada en la videoconsola mediante, por ejemplo, un flujo de datos de entrada de ficheros (*ifstream*). Otra cuestión ya mencionada, pero muy importante, es que toda reserva de memoria que se realice para almacenar un flujo de bytes debe estar alineado a 32 bytes y tener un tamaño múltiplo exacto de 32 bytes (consultar el capítulo 2 para más información).

En la primera iteración del proceso de desarrollo, la que cubre la construcción

de la clase *Screen*, se detalla todo lo relacionado con *GX*, el módulo de *libogc* que se encarga del sistema gráfico. Gracias a los avances conseguidos en aquel punto, ahora disponemos de un método que, a partir de una imagen almacenada en un flujo de datos, con sus bits organizados en RGB5A3 (formato de vídeo nativo de Nintendo Wii), crea un objeto de textura de tipo *GXTexObj*, con el que trabaja *GX*. Por otra parte, también disponemos en esta clase de dos funciones que nos permiten dibujar en la pantalla una textura de este tipo al completo, o una parte de ella. Respecto al formato de archivo de la imagen, se necesita, para una primera versión, un formato de archivo de imagen que no tenga compresión y que tenga la información de color de forma directa, sin utilizar paletas de color. El formato adecuado según estas premisas sería el mapa de bits (BMP) de 24 bits de color por píxel (8 bits por componente en cada píxel). Un detalle importante es la limitación que sufre el formato *GXTexObj*, que consiste en que las medidas de toda imagen que vaya a ser transformada en textura deben ser múltiplo de 8 (tanto el ancho como el alto en píxeles); en caso de no respetarse, se produciría un error en el sistema.

Respecto al sonido, *libogc* incorpora el módulo *asndlib*, que es un conjunto de funciones que procesan flujos de bytes con efectos de sonido. *Asndlib* utiliza un dispositivo hardware especial para mezclar el sonido, llamado DSP, y que soporta hasta 16 voces simultáneas (una voz es flujo de bytes que contiene un sonido). Nintendo Wii trabaja a 48000Hz de frecuencia, a 16 bits por *sample* y en estéreo de forma nativa. Este módulo de gestión de sonido debe inicializarse antes de ser usado, y apagarse antes de finalizar la ejecución del programa. *Asndlib* incorpora dos funciones de reproducción, una para reproducir una sola vez un efecto de sonido, y otra para reproducirlo infinitamente.

Las pistas de música son aún más fáciles de reproducir que los efectos de sonido, gracias a que *libogc* incorpora el módulo *libmad*. A partir de una zona de memoria que contenga un archivo de música en un formato MP3 válido, *libmad* reproduce la pista de música en una voz reservada del DSP, y además abstrae de la necesidad de cambiar la forma de representación del fichero MP3, ya que se encarga de realizar el cambio desde *LittleEndian* a *BigEndian* si fuera necesario.

En lo que concierne a la escritura en pantalla utilizando fuentes de texto, *libogc* no ofrece ningún módulo para trabajar con ellas, pero existe una adaptación de la biblioteca *FreeType2* para operar con Nintendo Wii, y que abstrae también del cambio de *Endian* al cargar las fuentes y de la reserva de memoria alineada y con tamaño múltiplo de 32 bytes. Una vez cargada una fuente con *FreeType2*, se puede extraer un carácter con una sencilla función que devuelve una imagen bitmap con dicho carácter dibujado. La biblioteca permite extraer la imagen del carácter en formato monocromo, es decir, en blanco y negro puro; por tanto, para dibujar el carácter bastaría con recorrer píxel a píxel esta imagen, dibujando en la pantalla los píxeles que correspondan en el color seleccionado (utilizando el método *dibujarPunto* de la clase *Screen*). Como último punto de interés, *FreeType2* soporta cualquier carácter de cualquier codificación, siempre que esté soportado en la fuente de texto

que se utilice.

4.5.2. Identificación de funcionalidad necesaria

Con respecto a las texturas, se necesita cargar desde la SD una imagen de mapa de bits de 24 bits de profundidad de color directo y almacenarla en un flujo de bytes en formato RGB5A3. Después de eso, se necesita crear una textura que se pueda dibujar con los métodos de la clase *Screen*, es decir, de tipo *GXTexObj*. Hay que permitir que se indique un color transparente (es decir, que no se dibuje) para que las texturas dibujadas en pantalla no se dibujen únicamente con forma rectangular, y ofrecer métodos observadores para el ancho y el alto en píxeles de la imagen. Por supuesto, también es necesario un método que dibuje la textura en la pantalla.

Los efectos de sonido únicamente necesitan ser cargados desde la tarjeta SD a un flujo de bytes, y poder ser reproducidos en cualquier momento. Como son sonidos estéreo (con dos canales), se debería permitir controlar el volumen de cada canal de forma independiente, para que así el usuario pueda conseguir efectos de sonido 3D.

Una pista de música debe ser cargada desde la tarjeta SD y almacenada en una zona de memoria, se debe poder reproducir, detener, y mantener en repetición si así se desea, así como controlar su volumen de reproducción.

Para trabajar con las fuentes de texto, primero hay que cargarlas desde la tarjeta SD y almacenarlas. Después de eso, hay que proporcionar al menos un método que permita escribir una cadena de caracteres, independientemente del juego de caracteres que utilice. Para ello, se puede usar el tipo de C++ *wstring*, que es idéntico a *string* salvo por el tipo de carácter base que utiliza, *wchar_t*, que permite trabajar con caracteres de hasta 32 bits.

Por supuesto, los cuatro recursos multimedia deben respetar las condiciones de utilizar alineación a 32 bytes cuando se reserve memoria, y se debe llenar dichas zonas de memoria para que su tamaño sea múltiplo de 32 bytes. Además, deben cuidarse todas las situaciones que puedan derivar en error, comprobando las condiciones necesarias antes de cada operación.

4.5.3. Diseño e implementación

Con toda la información recopilada, procedemos a crear cuatro clases, una para representar cada uno de los recursos multimedia presentados. A continuación se aporta una descripción de cómo cumple los objetivos marcados cada una de ellas.

La clase **Imagen** es la que se encarga de proporcionar la funcionalidad necesaria para dibujar texturas en la pantalla. Dispone de un atributo de clase público que indica el color considerado como transparente, que es compartido por todas las instancias de la clase, y que al cargar la imagen corresponde con un píxel con el canal *alpha* activado (es decir, un píxel que en la imagen tenga el mismo color que

el transparente será un píxel invisible en la textura). Se ofrecen métodos consultores para el ancho y el alto en píxeles de la imagen, así como a la textura *GXTexObj* creada a partir de ella.

En la primera versión de *LibWiiEsp* sólo se dará soporte a imágenes de mapa de bits de 24 bits de profundidad de color directo, como ya se mencionó antes. Como se pretende que sea sencillo implementar soporte para un abanico amplio de formatos de imagen, el constructor de la clase *Imagen* únicamente inicializará las variables internas de la instancia creada. Para cargar una imagen se utilizará el método *cargarXXX*, donde XXX corresponde con la extensión de la imagen en la tarjeta SD; así, para cargar *archivo.bmp* se implementa el método *cargarBmp*. Con esto se consigue que si en un futuro se quisiera implementar la carga de archivos PNG, bastaría con crear el método *cargarPng*.

Para trabajar con archivos BMP son necesarias dos estructuras, que corresponden con las dos cabeceras de todo archivo que siga este formato: la primera cabecera ocupa 14 bytes, e indica los detalles del archivo; la segunda son 40 bytes, y contiene información relativa a la imagen en sí.

Entrando en la implementación del método de carga de mapas de bits, el proceso que sigue es sencillo: primero se comprueba que la tarjeta SD está montada y accesible, y se lee el archivo BMP a un flujo de entrada de ficheros. Después, se lee la primera cabecera y se comprueba que el archivo sigue efectivamente un formato de mapa de bits. Seguidamente, se lee la segunda cabecera, y de ahí se extraen los datos de ancho y alto de la imagen (recordemos, deben ser ambos múltiplo de 8) y la profundidad del color (que, de momento, sólo será válido si es 24 bits). A continuación, se recorre el resto del archivo con un bucle, leyendo en cada iteración 3 bytes (que corresponden con un píxel, siguiendo el formato BBGRR, es decir, componente azul de 8 bits, después verde y por último rojo). La información de cada píxel se transforma al formato RGB5A3, haciendo que si el píxel coincide con el color invisible, tenga el canal *alpha* activo al máximo (transparencia total). Como detalle, una imagen BMP se lee desde abajo a la izquierda, hacia la derecha y hacia arriba, pero las texturas *GXTexObj* necesitan la información desde la esquina superior izquierda, hacia la derecha y hacia abajo. Una vez leída la imagen al flujo de bytes, se crea la textura utilizando el método correspondiente de la clase *Screen* y se da por finalizado el proceso.

Para dibujar la textura, se proporciona un método que llama a *dibujarTextura* de *Screen*, pasándole como parámetro la textura asociada con la imagen. La interfaz pública de la clase se muestra en la figura 4.9.

Respecto a la clase **Sonido**, se proporciona un método de clase que inicializa el módulo *asndlib*, y en el constructor se comprueba que la tarjeta SD está accesible y se lee el archivo de sonido (que debe tener *samples* de 16 bits con signo, estéreo, una frecuencia de 48000 Hz y representación *BigEndian*, formato que puede obtenerse

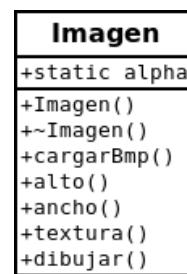


Figura 4.9: Interfaz pública de la clase Imagen

procesando el archivo con SoX [3]) a una zona de memoria alineada. Se ofrecen dos métodos para controlar el volumen de los dos canales del sonido por separado, y otro para reproducir el efecto. En la figura 4.10 puede observarse la interfaz pública de la clase.

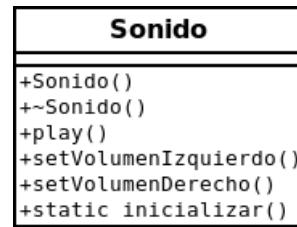


Figura 4.10: Interfaz pública de la clase Sonido

Las pistas de música funcionan con la clase **Musica**, que tiene una implementación parecida a la anterior, solo que también incorpora un método que detiene la reproducción, y otro que permite mantener la reproducción en bucle (el método comienza la reproducción de la pista si ésta ya hubiera terminado). Se muestra la interfaz pública de la clase en la figura 4.11.



Figura 4.11: Interfaz pública de la clase Musica

La clase **Fuente**, por último, utiliza el método de carga del archivo de fuentes que proporciona la adaptación de *FreeType2*, de tal manera que no hay que preocuparse por la memoria alineada y el *Endian*. Se aportan dos métodos de escritura en pantalla, uno recibe una cadena de caracteres *string*, y el otro una *wstring*, de tal manera

que se cubren todos los posibles caracteres (de hasta 32 bits). Para escribir el texto, se recorre la cadena recibida, cargando el bitmap asociado a cada carácter, y dibujándolo en la pantalla píxel a píxel en el color que se indique. También se ofrece un método de clase que inicializa la biblioteca *FreeType2*, y que debe ser llamado antes de poder utilizarla. La interfaz pública de la clase se puede observar en la figura 4.12.

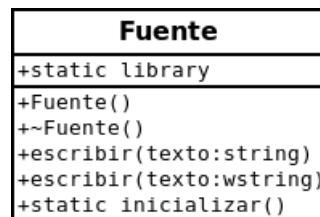


Figura 4.12: Interfaz pública de la clase Fuente

Todos los destructores de estas clases se encargan de liberar la memoria ocupada por todos los flujos de bytes y zonas de memoria reservada.

4.5.4. Pruebas

La batería de pruebas relativa a los recursos multimedia consistieron en un programa que cargaba una imagen desde un archivo, en distintos formatos (soportados o no), y trataba de dibujarla en pantalla. Las comprobaciones de las condiciones necesarias para la carga y utilización de imágenes BMP se cumplieron escrupulosamente, y sólo se dibujó en pantalla la que tenía el formato adecuado.

El sonido y la música se probaron en el mismo programa, cargando los archivos correspondientes y reproduciéndolos al pulsar un determinado botón del mando. Las fuentes de texto, por otra parte, se probaron escribiendo textos en la pantalla en distintos idiomas (inglés, español, ruso y japonés), resultando positivas todas las comprobaciones. Los caracteres no latinos se mostraban correctamente.

4.6. Trabajar con XML

En los requisitos sobre interfaces externas se hace referencia a la utilización de la tecnología XML para conseguir una separación efectiva entre código fuente y datos. En esta quinta fase del desarrollo se recoge información sobre la creación de un módulo que permita trabajar cómodamente con este formato de archivos.

4.6.1. Biblioteca *TinyXML*

El primer paso para cubrir la funcionalidad deseada consiste en una investigación sobre las distintas herramientas ya existentes que permitan cargar información desde un archivo XML y, por supuesto, que sean compatibles con Nintendo Wii. Tras la lectura de diversas fuentes, se encuentra que la única herramienta disponible

actualmente para procesar este tipo de ficheros en la videoconsola es *TinyXML*.

Esta pequeña utilidad proporciona abstracción sobre la lectura del árbol formado por los nodos del documento XML, y además, ya viene adaptada para su funcionamiento bajo la plataforma de Nintendo, por lo que no hay que preocuparse sobre la alineación de datos ni del *Endian*. Trabaja cargando el documento en memoria, y accediendo a los distintos nodos que lo componen mediante punteros. Da acceso a elementos, atributos y contenido de nodos, además de permitir una fácil navegación descendente (es decir, de padre a hijo), por lo que cubre las necesidades básicas para nuestros propósitos.

Al poderse cargar completamente el árbol XML con esta herramienta, podemos implementar fácilmente un *parser* que siga el método de trabajo DOM, que funciona precisamente cargando el árbol completo en memoria y accediendo posteriormente a los elementos necesarios.

4.6.2. Identificación de funcionalidad necesaria

Para trabajar cómodamente con archivos XML, necesitamos una interfaz accesible desde cualquier punto del sistema, y que permita cargar un documento completo y acceder al nodo raíz. A partir de este elemento (o de cualquier otro que hayamos localizado previamente), debemos poder buscar otro elemento bajo él a partir de su valor, y sería muy interesante poder navegar entre los hermanos (elementos que se encuentran bajo el mismo padre en el árbol) del mismo valor. Por supuesto, también se requiere poder leer atributos de tipo cadena de caracteres, entero y decimal de coma flotante, y al texto contenido en un elemento.

Por último, también es interesante que todas estas operaciones se realicen eficientemente, para no cargar demasiado el sistema al trabajar con este tipo de archivos de datos.

4.6.3. Diseño e implementación

Para obtener la funcionalidad especificada en el epígrafe anterior, se decide crear una clase que implemente el patrón *Singleton*, así podemos tener la certeza de que sólo habrá una instancia de esta herramienta en el sistema, ahorrando costes de memoria (ya que sólo se permite tener cargado un único árbol XML a la vez) y teniendo la utilidad siempre disponible en cualquier punto del sistema.

Se implementa un método que, a partir de la ruta absoluta de un archivo XML en la tarjeta SD, carga un árbol XML completo en memoria. Este método sobre-escribe cualquier otro documento XML que estuviera previamente cargado, por lo que se consigue que no haya fugas de memoria en este aspecto. También se crean varios métodos consultores que proporcionan acceso al nodo raíz del documento, al valor y el contenido de un elemento dado, y a un atributo de un elemento a partir de su nombre y el elemento al que pertenece (existiendo métodos de consulta de

atributos para cadenas de caracteres, números enteros y decimales de coma flotante).

Para facilitar la navegación por el árbol del documento, se crea un método que, dado un valor y un elemento, buscará y devolverá el primer elemento cuyo valor coincida con el indicado. Si no se le indica un elemento a partir del cual buscar, recorrerá todo el documento a partir del elemento raíz. Además, existe otro método que, a partir de un elemento, busca entre los elementos hermanos (que tienen el mismo nodo padre) por el siguiente elemento con el que comparta valor.

El recorrido del árbol XML del documento realizado por estos métodos de navegación se realiza en preorden, de ahí los criterios de ordenación de elementos (por ejemplo, "siguiente hermano" se refiere al elemento, hijo del mismo parente, que es posterior al dado en preorden). Comentar también que los posibles errores de inexistencia de un elemento con un valor dado o de un atributo solicitado están controlados haciendo uso de las comprobaciones pertinentes.

En la figura 4.13 puede observarse la interfaz pública de la clase Parser, que nace como resultado de esta fase de desarrollo.

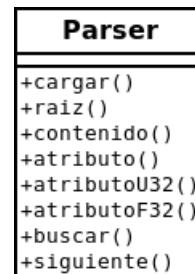


Figura 4.13: Interfaz pública de la clase Parser

4.6.4. Pruebas

La batería de pruebas de esta clase consistió en la carga de varios documentos XML distintos, a partir de los cuales se realizaron varias lecturas y recorridos en los árboles, imprimiendo en el sistema de consola de *libogc* los valores de cada elemento visitado, así como los atributos solicitados. Se probaron tanto casos que se esperaban correctos como situaciones en las que se solicita un atributo inexistente, la búsqueda de un elemento no válido o la obtención del valor o el contenido de elementos que no existen.

4.7. Organización de los recursos multimedia

Tras construir un módulo que nos permita trabajar cómodamente con archivos de datos en formato XML y de disponer de cuatro clases para hacer uso de recursos multimedia en el sistema, llega el momento de gestionar eficazmente dichos recursos.

4.7.1. Identificación de funcionalidad necesaria

Se pretende desarrollar un módulo que gestione todos los recursos multimedia existentes en el sistema, teniéndolos localizados desde cualquier punto de éste, y clasificados por tipo. Cada recurso debe estar identificado por un código único dentro de su tipo. Además, para obtener una separación efectiva entre código fuente y datos, se contempla la posibilidad de indicar, mediante un listado en formato XML, la relación de todos los recursos multimedia asociados con su código identificador.

4.7.2. Diseño e implementación

Para satisfacer la funcionalidad arriba indicada, se procede a diseñar una clase que implemente el patrón *Singleton*, ya que sólo necesitamos una instancia que gestione todos los recursos multimedia que se encuentren en el sistema. Además, la utilización de este patrón de diseño nos aporta la ventaja de que la instancia única de esta clase será accesible desde cualquier punto del sistema.

La clase dispondrá de un diccionario para cada tipo de recurso: imágenes, efectos de sonido, pistas de música y fuentes de texto. Cada entrada en el diccionario estará formada por una clave (una cadena de caracteres de tipo *string*), que será única en el diccionario, y asociado a ella habrá un puntero al objeto que contenga al recurso.

La carga de todos los recursos se realizará desde un método de inicialización que no será el constructor, debido a la implementación del patrón *Singleton*. Dicho método comprobará que la tarjeta SD está accesible, cargará un archivo XML cuya ruta absoluta en la tarjeta reciba por parámetro y recorrerá el documento XML, cargando todos los recursos multimedia que se indiquen en el archivo. Se espera que cada elemento hijo del nodo raíz del documento contenga la información de creación de un recurso, siendo identificado el tipo de recurso por el valor del elemento. A continuación se muestra un ejemplo válido del formato esperado para el documento XML:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <galeria>
3      <imagen codigo="fondo" formato="bmp" ruta="/directorio/fondo.bmp" />
4      <musica codigo="rock" volumen="128" ruta="/directorio/rock.mp3" />
5      <sonido codigo="bang" volumen="255" ruta="/directorio/bang.pcm" />
6      <fuente codigo="arial" ruta="/directorio/arial.ttf" />
7  </galeria>
```

El método inicializador se encargará de llamar al constructor de la clase del recurso, utilizando como parámetros los atributos que acompañan a cada elemento en el archivo XML, y almacenará la dirección del objeto creado junto a su código en el diccionario correspondiente.

Para acceder a los recursos, la clase proporciona un método observador para cada tipo de recurso (es decir, para cada diccionario). Estos métodos reciben el código de

identificación del recurso que se desea utilizar, y devuelven una referencia constante a él, abstrayendo al usuario de la lógica de punteros. El destructor de la clase se encarga de destruir todos los objetos de recursos almacenados en los diccionarios y liberar la memoria ocupada por éstos.

En la figura 4.14 se muestra la interfaz pública de la clase Galeria.



Figura 4.14: Interfaz pública de la clase Galeria

4.7.3. Pruebas

La clase Galeria se ha probado con varios ficheros XML, tanto en casos de documentos mal formados como válidos, cargando los recursos indicados y utilizándolos en un programa de prueba. También se probó el caso de intentar cargar un recurso que no existe en la tarjeta SD, y el de código identificador repetido.

4.8. Soporte de internacionalización (i18n)

Continuando con las posibilidades que nos ofrece el disponer de un módulo que nos ayude a trabajar cómodamente con el formato de datos XML, y dado que la clase *Fuente* puede escribir en la pantalla cualquier carácter gracias a las cadenas de caracteres anchos (de tipo *wstring*), vamos a aplicar una idea parecida a la de la clase *Galeria*, pero enfocada a proporcionar un buen soporte para la internacionalización de los juegos desarrollados con *LibWiiEsp*.

4.8.1. Identificación de funcionalidad necesaria

Lo que se quiere conseguir con este módulo es que, en lugar de escribir texto en la pantalla directamente desde el código fuente, se asigne una *etiqueta* o código identificador al punto del programa en el que se quiere mostrar el texto. La idea es que, en tiempo de ejecución, se sustituya dicha etiqueta por el texto correspondiente a ella en el idioma que se indique.

Un idioma, pues, debe ser un conjunto de textos, escritos todos en la misma lengua, identificado cada texto por un código o nombre de etiqueta. Todos los idiomas deben tener las mismas etiquetas, solo que el valor de la misma etiqueta para distintos idiomas será la misma cadena de caracteres traducida al que corresponda.

Para permitir una ampliación sencilla y rápida de los idiomas disponibles en un videojuego (además de separar código fuente y datos), se necesita que todas las etiquetas se definan en un archivo externo que sea cargado en tiempo de ejecución. Lo ideal es utilizar la tecnología XML y el módulo de *LibWiiEsp* que permite trabajar con ella.

Se requiere también que sea sencillo cambiar entre idiomas, para lo que parece adecuado tener un idioma marcado como activo en un momento dado, y que se permita conocer qué idioma se encuentra activo en un momento dado, la existencia o no de un idioma, seleccionarlo y alternar entre todos los idiomas disponibles de forma circular.

Por último, es necesario que el módulo que se encargue de gestionar el valor de las etiquetas de texto se encuentre disponible desde cualquier punto del sistema, y que exista únicamente una instancia en éste.

4.8.2. Diseño e implementación

Para cubrir la funcionalidad descrita se construye una clase que implementa el patrón *Singleton*, ya que uno de los requisitos del módulo consiste en disponer de una instancia de la clase, y que sea accesible desde cualquier punto del sistema.

La estructura de datos diseñada para almacenar un idioma consiste en un diccionario, donde su primera componente es una cadena de caracteres de alto nivel (de tipo *string*) que almacena el nombre o código de una etiqueta de texto (y mediante la cual se identifica la etiqueta), y su segunda componente es otra cadena de caracteres de alto nivel, pero de caracteres anchos (de tipo *wstring*), en la cual se guarda el texto asociado a la etiqueta traducido al idioma correspondiente.

Cada idioma se almacena en otro diccionario en la segunda componente de éste, identificado por su nombre (cadena de caracteres anchos *wstring*) en la primera componente del diccionario.

El método de inicialización del soporte de internacionalización requiere como parámetro una cadena de caracteres con la ruta absoluta en la tarjeta SD de un archivo XML que contenga la información de las etiquetas de texto para todos los idiomas que se utilizarán. Este archivo se carga en memoria y se recorre, creando una entrada en el diccionario de idiomas para cada uno que encuentre, y guardando cada etiqueta de texto asociada a él. A continuación se muestra un ejemplo de archivo de idiomas soportado:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <lang>
3   <idioma nombre="español">
4     <tag nombre="PUNTOS" valor="Puntuación" />
5     <tag nombre="VIDAS" valor="Vidas" />
6     <tag nombre="NIVEL" valor="Nivel" />
```

```

7  </idioma>
8  <idioma nombre="english">
9   <tag nombre="PUNTOS" valor="Score" />
10  <tag nombre="VIDAS" valor="Lives" />
11  <tag nombre="NIVEL" valor="Level" />
12 </idioma>
13 <idioma nombre="français">
14  <tag nombre="PUNTOS" valor="Points" />
15  <tag nombre="VIDAS" valor="Vies" />
16  <tag nombre="NIVEL" valor="Niveau" />
17 </idioma>
18 </lang>

```

Al leer el contenido de una etiqueta, éste se almacena temporalmente en una cadena de caracteres *string*. Antes de guardarlo en el diccionario del idioma correspondiente, se transforma a *wstring*, para así asegurar que los caracteres son los correctos. La explicación de esta transformación es sencilla: cuando se lee un texto con caracteres anchos (que necesitan un espacio mayor que 1 byte por carácter), cada carácter ancho se 'rompe' en trozos de 1 byte cada uno. Un carácter ASCII normal ocupará el espacio esperado, es decir, 1 byte. Durante la transformación se crea una cadena de caracteres anchos (de tipo *wchar_t*), y se rellena con el valor de la etiqueta después de pasarlo por la función de C llamada *mbstowcs*. Esta función se encarga de detectar los caracteres anchos que están 'rotos' en trozos de 1 byte, los une, y devuelve toda la cadena recibida como una cadena de caracteres anchos. Por último, se almacena lo devuelto en una variable *wstring*, y se introduce en el idioma correspondiente.

Se proporciona un método observador para conocer el nombre del idioma activo, otro que comprueba si un nombre de idioma introducido existe como clave en el diccionario que almacena los idiomas, y dos métodos para cambiar el idioma: uno que recibe directamente como parámetro el nombre del idioma a activar, y otro que alterna entre todos los idiomas registrados a través de una función que va activando cada uno según se lo encuentre al recorrer el diccionario. Este último método obtiene un iterador al idioma activo actual dentro del diccionario, y lo avanza una posición. Si no se ha llegado al final del diccionario, se activa el idioma encontrado; en caso contrario, se activa el primero del diccionario.

Por último, para obtener el texto asociado a una etiqueta concreta en el idioma activo actual, existe una función que, dado el nombre de la etiqueta mediante una cadena de caracteres *string*, devuelve el texto asociado a ésta mediante una cadena de caracteres anchos *wstring*, el cual se puede dibujar en pantalla (si la fuente seleccionada para ello tiene soporte para el juego de caracteres necesario), guardar en un fichero, o utilizar en cualquier operación deseada.

En la figura 4.15 se muestra la interfaz pública de la clase Lang.

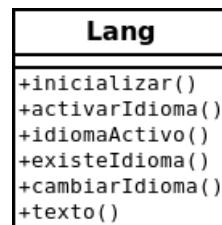


Figura 4.15: Interfaz pública de la clase Lang

4.8.3. Pruebas

Respecto a las pruebas realizadas para validar este módulo de soporte de internacionalización, se han realizado comprobaciones que consistían en imprimir en la pantalla las etiquetas de texto en diversos idiomas, como son español, inglés y ruso (para los caracteres no latinos). También se probaron los métodos de cambio de idioma activo, y la reacción del módulo al pedirle encontrar un nombre de idioma o etiqueta que no existen.

4.9. Animaciones

Llegados a este punto del desarrollo, prácticamente hemos cubierto todas las herramientas necesarias para crear un videojuego, pero de momento, únicamente disponemos de la posibilidad de dibujar una textura estática, lo que nos limita a la hora de crear elementos dinámicos en nuestro juego. En este capítulo, se consigue un nuevo módulo, partiendo de la clase *Imagen*, con el que se pueden crear animaciones fácilmente.

4.9.1. ¿Qué es una animación?

En primer lugar, vamos a definir qué tipo de animación queremos conseguir. La idea de animación que se persigue conseguir es una simulación de movimiento mediante la visión de una secuencia de imágenes (denominadas fotogramas), una detrás de otra, pero con leves variaciones entre sí. Es la misma idea que se utiliza en los dibujos animados, medio en el que se muestra una serie de imágenes estáticas con leves diferencias entre sí que, al ser observadas secuencialmente, consiguen que el cerebro humano interprete un movimiento reconstruyendo los "huecos" que existen entre una imagen y la siguiente.



Figura 4.16: Animación formada por 5 fotogramas

Partiendo de este concepto de animación, se va a construir un módulo para *LibWiiEsp* que, tomando como base la información de una textura contenida en un objeto de la clase *Imagen*, se encargue de animar los fotogramas que se indiquen en dicha textura.

4.9.2. Identificación de funcionalidad necesaria

La funcionalidad principal para este módulo es dibujar en pantalla una secuencia de imágenes, tomadas todas ellas a partir de una misma textura en la que se encuentren todos los fotogramas de la secuencia. La animación se reproducirá continuamente, es decir, después del último paso se volverá a comenzar por el primero. Además, se debe permitir que entre un cambio de paso y el siguiente cambio puedan existir algunos fotogramas en los que se dibuje el mismo paso de la animación, para poder regular así la velocidad de reproducción de ésta.

Es importante también que se permita dibujar la animación tal cual, o que cada paso de ésta pueda ser invertido respecto al eje vertical; esto se utilizará para optimizar el espacio en memoria en los casos en que dos animaciones distintas sean idénticas, pero con distinta orientación (izquierda y derecha), ya que permitirá utilizar la misma textura para reproducir ambas animaciones.

Por último, se necesitará acceder al ancho y alto en píxeles de la animación, y al objeto de la clase *Imagen* que sirve de base para la animación, así como conocer en qué paso de la animación se encuentra ésta en un momento dado.

4.9.3. Diseño e implementación

Se crea una clase nueva para cubrir la funcionalidad descrita en el epígrafe anterior.

Cuando se crea una animación, se le deben pasar al constructor de la clase una serie de parámetros, que son un puntero a la imagen (previamente cargada en memoria) que contenga la rejilla de fotogramas, una cadena de caracteres de alto nivel (de tipo *string*) que contenga una secuencia de números enteros positivos, separados por comas y sin espacios (del tipo "0,1,2,3,4,5"), el número de filas y de columnas que hay en la rejilla, y el retardo a aplicar a la visualización de la animación (el retardo indica el número de fotogramas consecutivos en los que se dibuja el mismo paso de la animación; el valor por defecto 1 indica que se dibuja un nuevo cuadro a cada fotograma).

La secuencia de números enteros que se recibe indica el orden en el que se van dibujando los distintos cuadros de la animación. Cada cuadro de la rejilla corresponde con un número entero positivo, siendo el cero el fotograma de arriba a la izquierda, y avanzando de derecha a izquierda y de arriba abajo. Por ejemplo, una rejilla de tres columnas y dos filas tendrá en su primera fila (de izquierda a derecha) los cuadros cero, uno y dos; y la segunda fila, los cuadros tres, cuatro y cinco (igualmente, de izquierda a derecha). Un mismo cuadro se puede repetir cuantas veces se quiera en

la secuencia de una misma animación.

Un objeto de animación proporciona varios métodos observadores para conocer y acceder fácilmente a las variables que lo controlan, por ejemplo, la imagen, el ancho y alto en píxeles de un cuadro (todos los cuadros se consideran iguales), y cuáles son los índices del primer cuadro y del cuadro actual. También existen dos métodos para modificar el estado de la animación, concretamente son reiniciar (método que establece el paso actual al primero de la secuencia) y avanzar (que se encarga de calcular el siguiente cuadro a dibujar teniendo en cuenta el retardo y la propia secuencia de cuadros).

El método dibujar se encarga de, como su propio nombre indica, plasmar en la pantalla el fotograma correspondiente al cuadro actual de la animación. Se da la opción de invertir el fotograma respecto al eje vertical, funcionalidad que proporciona la propia clase Screen a través de su método *dibujarCuadro* (que es el método que sirve de base para dibujar la parte correspondiente de la textura origen). Este método realiza una llamada a avanzar, con lo que no es necesario preocuparse por la gestión de cuadros desde el exterior.

Una cosa más a tener en cuenta es que el destructor de la clase es el predeterminado, por lo que no se destruye la imagen asociada a la animación (se utiliza un puntero constante a la imagen), y hay que destruirla manualmente en caso de que se quiera liberar la memoria ocupada por ésta.

En la figura 4.17 puede verse la interfaz pública de la clase Animacion.

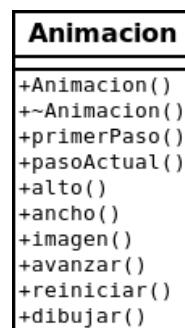


Figura 4.17: Interfaz pública de la clase Animacion

4.9.4. Pruebas

Las pruebas realizadas a este módulo consistieron en crear varias animaciones a partir de otras tantas texturas previamente cargadas en objetos de la clase *Imagen*, y dibujarlas en pantalla con distintos retardos, con inversión respecto al eje vertical o sin ella, y con distintas secuencias de pasos.

4.10. Registro de mensajes

Debido a la complejidad de la depuración de código en la plataforma Nintendo Wii (especialmente en lo referente a depuración en tiempo de ejecución), se hace patente la necesidad de un módulo que permita al desarrollador conocer el estado de las variables y expresiones cuando se está ejecutando un programa en el que se detecta un error. Para cubrir estas situaciones, se crea un sistema de gestión de errores basado en excepciones, que se aplica a todas las clases implementadas hasta el momento, y un módulo que permita generar un registro de mensajes, o *logs*, a partir de la información que el programador estime oportuna.

4.10.1. Excepciones

LibWiiEsp proporciona una jerarquía de excepciones que permite controlar los distintos comportamientos erróneos que sucedan durante la ejecución de un programa. Todas las excepciones derivan de la clase *Expcion*, que a su vez hereda de la clase *std::exception*, de tal manera que es compatible con una gestión de excepciones estándar.

Para conocer el mensaje de error que contienen todas las excepciones, puede utilizarse el método *what*, que devuelve una cadena de caracteres de tipo *string* con la información que se haya indicado al lanzar la excepción.

Las excepciones que se contemplan son las siguientes:

- **Expcion:** Clase de excepción que sirve como base a todas las demás.
- **ArchivoEx:** Excepción para controlar errores relacionados con la apertura de archivos.
- **CodigoEx:** Sirve para gestionar los fallos que puedan producirse al trabajar con códigos identificativos.
- **ImagenEx:** Para tratar los errores concretos de la clase Imagen.
- **NunchuckEx:** Esta clase de excepción se utiliza para informar de situaciones relacionadas con el *Nunchuck*.
- **TarjetaEx:** Generalmente sólo se utiliza para indicar que la tarjeta SD no está disponible.
- **XmlEx:** Excepción que permite identificar cuándo sucede un error al trabajar con un árbol XML.

4.10.2. Identificación de funcionalidad necesaria

La gestión de mensajes del sistema debe estar centralizada en un punto del sistema, que además resulte accesible desde cualquier punto de un programa. Lo que se pretende con la gestión de errores y el registro de mensajes consiste en almacenar

textos con información relevante sobre el estado del sistema en un momento determinado. Dicha información se guardará temporalmente en un *búffer* en memoria y será posteriormente volcada a un archivo de texto plano en la tarjeta SD.

Se permitirán tres niveles de prioridad para los mensajes: errores (sólo se muestran los mensajes de situaciones que generen una interrupción en la ejecución del programa), avisos (se muestran los anteriores y también aquellos derivados de los casos en que, sin llegar a interrumpir la ejecución, provocan un comportamiento anómalo en el sistema), e información, que muestra todos los mensajes definidos.

También debe poderse apagar el sistema de errores, de tal manera que no genere un archivo de *logs* o registro cuando un programa ya esté en producción.

4.10.3. Diseño e implementación

Para el registro de mensajes del sistema, se crea una clase que implementa el patrón *Singleton*, ya que queremos centralizar toda la gestión de mensajes en un único punto del sistema, y tenerlo siempre accesible. Esta clase tendrá como atributos privados el nivel de los mensajes que quieren observar, un *búffer* para almacenar todos los textos que se escribirán en el archivo de *logs* (que es una cadena de caracteres *string*) y un flujo de salida de bytes a fichero, en el que se realizará el volcado de información desde el *búffer* hacia el archivo.

Los niveles de registro están agrupados en una enumeración (con los valores OFF, ERROR, AVISO, INFO) que, a su vez, se define como tipo público de la clase para facilitar su utilización. El orden es desde el nivel más restrictivo (OFF, no se registra nada) a más amplio (INFO, se registra todo). Un nivel más amplio incluye a los más restrictivos que él, lo cual quiere decir que si, por ejemplo, se activara el nivel AVISO, se registrarían tanto mensajes de aviso como de error.

Cuando se inicializa el sistema de registro de mensajes, se guarda el nivel de *log* introducido como parámetro, y si éste no es OFF (no registrar nada), se sigue adelante. Se comprueba que la tarjeta SD esté disponible para realizar operaciones de entrada/salida, se abre el archivo de *log* indicado en el primer parámetro en modo escritura (si no existiera, se crea), y se establece el *búffer* para los mensajes. La instancia irá registrando mensajes en el búfalo, cada uno en una línea, y comenzando por las cadenas [INFO] para el nivel de información, [AVISO] para el nivel de aviso, y [ERROR] para una cadena que contenga un mensaje con nivel de registro error.

El guardado de mensajes se realiza por orden de ejecución, es decir, si un mensaje se encuentra por debajo de otro en el archivo de texto plano, significa que se ejecutó posteriormente al segundo. Por último, hay que tener en cuenta que el volcado del *búffer* en el archivo se realiza cuando se destruye la instancia de la clase. La ventaja de esta decisión es que, al realizarse un único guardado, el rendimiento del programa no se ve afectado con las muchas operaciones de escritura que pueden darse durante

la ejecución. Sin embargo, existe un inconveniente, ya que si ocurriera un error que no estuviera controlado en el código, se generaría un *pantallazo* y se finalizaría la ejecución sin guardar el registro de mensajes en el archivo. En versiones posteriores se mejorará este aspecto, haciendo posible que se puedan guardar la información generada hasta un cierto momento, a voluntad del programador.

En la figura 4.18 se muestra la interfaz pública de la clase Logger.

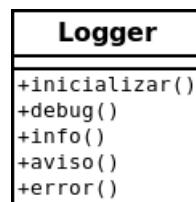


Figura 4.18: Interfaz pública de la clase Logger

4.10.4. Pruebas

Para comprobar el buen funcionamiento de esta clase, se creó un pequeño programa de prueba que escribía mensajes de sistema en distintas prioridades, y se probó este programa con los distintos niveles posibles para la clase (OFF, ERROR, AVISO, INFO).

4.11. Detección de colisiones

La décima etapa de desarrollo del proyecto aborda uno de los módulos más importantes a la hora de crear un videojuego en dos dimensiones, ya que debe ser escrupulosamente eficiente y, a la vez, permitir una gran flexibilidad. En esta sección se detallan todos los pormenores del sistema de colisiones de *LibWiiEsp*.

Este módulo de detección de colisiones no depende de ninguna otra parte del sistema, así que puede ser reutilizado para otros desarrollos no relacionados con Nintendo Wii.

4.11.1. Identificación de funcionalidad necesaria

Las colisiones se calcularán en base a figuras geométricas planas, y una colisión ocurrirá siempre entre dos de estas figuras. Se necesita que, cuando se evalúe una colisión, se indique únicamente si efectivamente ha ocurrido o no, no es necesario identificar el punto o puntos de colisión. Las figuras entre las que se calcularán colisiones serán un punto, un círculo y un rectángulo cuyos lados estén alineados a los ejes.

Las figuras de colisión estarán normalmente asociadas a entidades propias del juego que tendrán unas coordenadas respecto a un punto (0,0) determinado por la

esquina superior izquierda del escenario, por lo que se ha planteado que existen dos formas de representar la posición de las figuras de colisión asociadas:

1. Figuras de colisión con posición absoluta respecto al mismo origen de coordenadas que las entidades del juego.
2. Que las figuras tengan su posición relativa al punto superior izquierdo del objeto del juego con el que están asociadas.

Ambas aproximaciones tienen sus ventajas e inconvenientes. El primer punto de vista facilita los cálculos, pero necesita que se actualice la posición de las figuras cada vez que el objeto al que están asociadas modifique la suya. Por otro lado, la segunda opción complica ligeramente la detección de la colisión (ya que habría que tener en cuenta la posición de los objetos contenedores de las figuras), pero elimina la necesidad de actualización de la posición de las figuras.

Se ha elegido la segunda opción por considerarla más eficiente, así que será necesario que en los cálculos matemáticos se tengan en cuenta las posiciones de los objetos asociados a las figuras de colisión.

Por último, se necesita que una figura de colisión pueda ser cargada desde un elemento de un árbol XML.

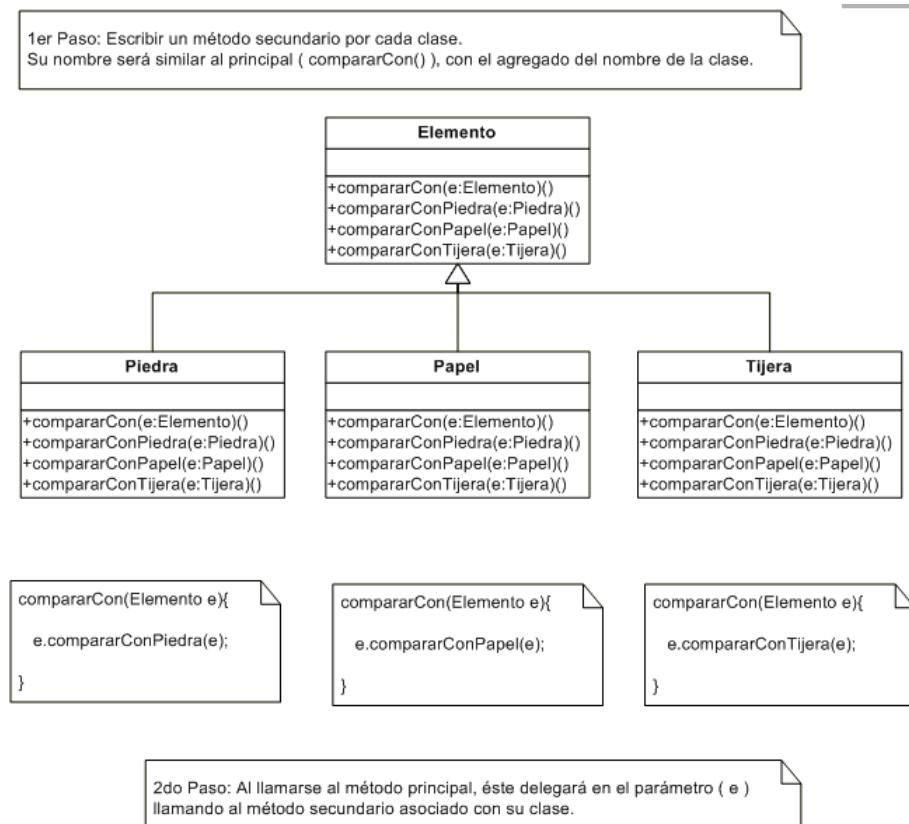
4.11.2. Diseño e implementación

En base a lo decidido en el punto anterior, se crea una clase abstracta pura, llamada Figura, y de la cual derivarán todas las demás clases que representen los distintos polígonos de colisión. Concretamente, de la clase Figura derivarán las clases Punto, Rectángulo y Círculo.

La clase punto tendrá dos enteros para indicar sus coordenadas respecto a la esquina superior izquierda del objeto del sistema con el que estará asociado, un círculo estará determinado por un punto y un número decimal que indique el radio, y un rectángulo estará formado por cuatro puntos. A pesar de que los rectángulos tendrán sus lados alineados a los ejes de coordenadas, se utiliza la representación de cuatro puntos en lugar de un punto y las longitudes de los lados para que, en el futuro, se pueda eliminar fácilmente la restricción de los lados paralelos a los ejes.

La clase abstracta Figura es la base del módulo de gestión de colisiones integrado en *LibWiiEsp*. Se basa en una implementación de la técnica *double dispatch* (que a su vez es una implementación del patrón *Visitante*) consistente en una especie de polimorfismo en tiempo de ejecución, donde la elección del método a ejecutar depende no sólo del objeto que lo ejecuta, si no del tipo del parámetro que recibe. En la práctica, al emplear esta técnica se consigue evitar la comprobación de tipos mediante estructuras de tipo condicional (*if*, *switch* ...) cuando se quiere evaluar una colisión entre dos objetos de clases derivadas de Figura.

Entrado un poco más en detalle, *double dispatch* implica que toda clase derivada de Figura implementará un método denominado *hayColision*, en el que recibirá un puntero a zona de memoria donde se almacenará otra Figura. Este método devolverá un valor booleano, que será verdadero si hay colisión entre las dos figuras, o falso en caso contrario. Internamente, devuelve el resultado de llamar al método *hayColision* de la segunda figura, pasando como parámetro la figura actual (es decir, el objeto *this* en el contexto de la primera figura). Con este intercambio de parámetros se consigue que esta segunda llamada se realice conociendo el tipo exacto del parámetro que se pasa, ya que en el contexto de ejecución (se hace desde un método de la primera figura) se conoce el tipo del objeto *this*. Al llamarse el método *hayColision* de la segunda figura con un parámetro con el tipo perfectamente identificado, esta segunda figura ya sabe los tipos de ambos objetos (conoce su propio tipo por el contexto de ejecución, y el de la primera figura por lo explicado anteriormente), de tal manera que se ejecuta el método adecuado con los cálculos necesarios para averiguar si hay colisión entre ambas figuras o no. En la figura 4.19 se puede observar un ejemplo de implementación de la técnica.



El otro aspecto importante relativo a las figuras de colisión es el desplazamiento de éstas. Una figura de colisión está determinada, como mínimo, por una pareja de coordenadas cuyo punto de origen (es decir, el (0,0)) corresponde con el punto superior izquierdo de un objeto del sistema; objeto que tendrá, a su vez, una pareja de coordenadas (x,y) respecto al límite izquierdo del escenario (coordenada X, cuanto mayor sea, más alejado hacia la derecha estará el objeto de este límite izquierdo) y al límite superior del escenario (coordenada Y, cuanto mayor sea, más alejado hacia abajo estará el objeto de este límite superior). Es decir, las coordenadas de una figura de colisión son relativas a un punto que no tiene por qué ser el origen (0,0) de las coordenadas de los objetos del sistema.

Por ejemplo, suponiendo un personaje en un juego de plataformas que tenga asociado un rectángulo de colisiones. Este personaje tiene unas coordenadas (x,y) respecto al punto superior izquierdo del escenario, y que determinan su posición en dicho escenario. Esta pareja de coordenadas variará en función del comportamiento del personaje durante la ejecución del juego. Sin embargo, las parejas de coordenadas de los puntos que determinan el rectángulo de colisiones se mantendrán fijas en todo momento, ya que son relativas al punto superior izquierdo del personaje, y no al del escenario. Esto provoca que, para conocer la posición absoluta de un punto del rectángulo respecto al origen de coordenadas del escenario, haya que sumar el valor de la coordenada X del personaje con el de la coordenada X del punto; y análogamente para las coordenadas Y. En la figura 4.20 se puede observar un ejemplo del sistema de coordenadas de las figuras de colisión, donde el origen de coordenadas es el punto superior izquierdo del objeto con el que están las figuras asociadas.



Figura 4.20: Ejemplo para ilustrar el desplazamiento de las figuras de colisión

Tomando en consideración todo lo explicado, se ha creado una clase abstracta pura, llamada Figura, que será la base para todas las clases derivadas que representen a una figura concreta. Esta clase Figura incluye métodos de clase (estáticos) que se encargan de crear una figura concreta a partir de un elemento de un árbol XML, así como los métodos virtuales puros que implementan el patrón *Visitante* y la técnica del *double dispatch*.

A partir de esta clase, se crean las clases Punto, Rectangulo y Círculo, cada una con los métodos necesarios para evaluar las posibles colisiones entre sí. También incluyen los métodos observadores y modificadores para todos sus atributos. La interfaz pública de la clase Figura pueden observarse en la figura 4.21.

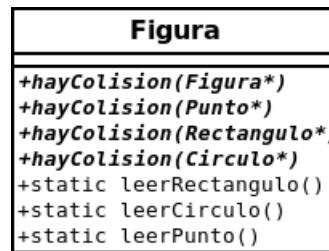


Figura 4.21: Interfaz pública de la clase Figura

Por último, comentar que se ha conseguido una escalabilidad alta, ya que para añadir una figura nueva (por ejemplo, un triángulo), bastaría con crear una nueva clase que herede de Figura, y añadir a las existentes los métodos necesarios para detectar colisiones entre ellas y la nueva.

4.11.3. Pruebas

Las comprobaciones realizadas a este módulo consistieron en un programa que generaba varias figuras y comprobaba las colisiones entre ellas. Previamente a la ejecución del programa, se sabía qué figuras debían colisionar con otras. A la hora de probar la figura Punto se creó un programa interactivo, en el que se asignaban las coordenadas del puntero infrarrojo del mando a un objeto de la clase, y se actualizaba su posición en cada fotograma. Además, se incluían varias figuras (rectángulos y círculos) que cambiaban de color según apuntáramos el puntero infrarrojo sobre ellas o no (es decir, si se detectaba una colisión entre el punto y las figuras).

4.12. Plantillas

Con todo lo desarrollado hasta el momento, ya se dispone de una herramienta completa para construir videojuegos para Nintendo Wii. A partir de este punto, se considera necesario la creación de una serie de plantillas (clases abstractas) para facilitar la implementación de videojuegos. El uso o no de estas plantillas son totalmente opcionales, pero suponen una ayuda para aquellos programadores que no

tengan mucha experiencia en el desarrollo de videojuegos en dos dimensiones.

4.12.1. Actores

Un actor es la unidad básica en el desarrollo de un videojuego en dos dimensiones, y consiste en un objeto con entidad propia que se puede visualizar en la pantalla, y que almacena información sobre sus distintas características. Estas características son las siguientes: posición actual dentro del universo del juego (pareja de coordenadas (x,y)), posición previa del actor en el universo del juego (otro par de coordenadas (x,y)), velocidad de movimiento horizontal y vertical (número de píxeles que avanza el actor en un fotograma en cada uno de los ejes), estado actual del actor, estado previo del actor y el tipo de actor (una cadena de caracteres). Por supuesto, al derivar esta clase abstracta, se pueden añadir más características según se crea necesario.

Realizando un acercamiento más técnico y concreto, un actor no es más que un conjunto de las características antes mencionadas, además de dos diccionarios donde, a cada estado del actor, le corresponde una animación y un conjunto de cajas de colisión, y que proporciona una manera sencilla y efectiva de cargar todas estas características desde un archivo XML almacenado en la tarjeta SD de la consola.

Existen dos referencias para las coordenadas de un actor. En primer lugar, están las coordenadas de posición del actor dentro del escenario que conforma el propio juego, y que se refieren al desplazamiento en píxeles hacia la derecha del actor respecto al límite izquierdo del escenario (coordenada X), y al desplazamiento hacia abajo, también en píxeles, del actor respecto al límite superior de dicho escenario (coordenada Y). Por otra parte, existe otra referencia para las coordenadas de dibujo de un actor en la pantalla, y que se refieren a la posición respecto a los límites izquierdo y superior de la pantalla, además de la capa en la que se dibuja el actor (consultar la documentación de la clase Screen para más información respecto al dibujo de texturas en la pantalla).

El comportamiento de un actor viene definido por los distintos estados que puede adoptar a lo largo de la ejecución del juego. Cada estado tiene asociado un comportamiento concreto, y que se ejecutará en cada iteración del bucle principal del juego en la que el actor tenga activo ese estado concreto. Las transiciones entre estados se realizan externamente (generalmente, desde la clase que gestiona el escenario, y que deriva de la clase abstracta Nivel). Los estados se identifican mediante una cadena de caracteres de alto nivel (de tipo *string*), y se crean según aparezcan en el archivo XML que define las animaciones y las cajas de colisión para cada estado, es decir, un estado sólo se creará si aparece en algún momento en el mencionado archivo XML. El comportamiento del actor según su estado actual debe definirse en el método virtual puro *actualizar* cuando se derive la clase Actor.

Un detalle muy importante es que existe un estado obligatorio que hay que incluir forzosamente, y es el estado "normal", que es el que se toma por defecto. Si este estado faltara en el archivo XML del actor, se produciría un error y/o un com-

portamiento inesperado.

La cadena de caracteres que identifica a un estado se toma como clave para dos diccionarios, uno que almacena una animación asociada al estado concreto, y otra que almacena un conjunto (de tipo *set<Figura*>*) de figuras de colisión también asociadas al mismo estado. Estos diccionarios que tienen códigos de estado como clave identificativa se utilizan para que, en cada iteración del bucle principal, se dibuje la animación del actor correspondiente con el estado actual de éste, y sólo se tengan en cuenta las cajas de colisión asociadas al estado actual del actor para evaluar sus colisiones. Se proporcionan métodos para evaluar colisiones entre dos actores de una manera muy sencilla, y también para dibujar el actor en la pantalla en base a unas coordenadas de pantalla que se reciben desde fuera de la clase (la diferencia entre las coordenadas de posición y las de pantalla se explican unos párrafos más arriba).

Otro detalle más es que se almacena un identificador de tipo de actor, es decir, todos los actores que estén gestionados por la misma clase derivada de Actor deberán tener este atributo con el mismo valor.

Como ya se ha comentado, se consigue una separación completa de código fuente y datos, de tal manera que para cambiar las figuras de colisión, los estados o las animaciones de un actor, basta con modificar el archivo XML desde el cual se lee toda esta información. Este archivo tendrá una estructura parecida a esta:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <actor vx="3" vy="3" tipo="jugador">
3      <animaciones>
4          <animacion estado="normal" img="chief" sec="0" filas="1" columnas="5" retardo="3" />
5          <animacion estado="mover" img="chief" sec="0,1,2,3,4" filas="1" columnas="5" retardo="3" />
6          <animacion estado="muerte" img="chief" sec="5" filas="1" columnas="6" retardo="0" />
7      </animaciones>
8      <colisiones>
9          <rectangulo estado="normal" x1="27" y1="21" x2="55" y2="21" x3="55" y3="96" x4="27" y4="96" />
10         <circulo estado="normal" cx="41" cy="13" radio="8" />
11         <rectangulo estado="mover" x1="27" y1="21" x2="55" y2="21" x3="55" y3="96" x4="27" y4="96" />
12         <circulo estado="mover" cx="41" cy="13" radio="8" />
13         <sinfigna estado="muerte" />
14     </colisiones>
15 </actor>

```

En el elemento raíz se observan tres atributos, que son la velocidad de movimiento horizontal del actor en píxeles por fotogramas, la velocidad de movimiento vertical, y el tipo de actor.

Se pueden apreciar dos grandes bloques, uno para las animaciones, y otro para las cajas de colisión. Cada animación y figura incluye la información necesaria para ser creada, además de un estado al que se asociará. Sólo puede haber una animación por estado; pero no ocurre así para las cajas de colisión, de las cuales puede haber todas las que se deseen en cada estado. Cade destacar que, en ambos bloques, aparece el estado obligatorio "normal", como ya se indicó anteriormente.

Por último, cabe destacar que al derivar la clase Actor, se le pueden añadir nuevos atributos y métodos según se considere necesario, consiguiendo así partir de una base como es la propia clase Actor, pero pudiendo llegar a la complejidad que se deseé. También hay que mencionar que a la hora de crear un actor, es apropiado diseñar los cambios entre estados como un autómata finito determinado, que como ya se ha explicado, realizaría sus transiciones entre estados de forma externa. Si se realiza de esta manera, es muy fácil desarrollar un actor en un período de tiempo relativamente pequeño.

4.12.2. Niveles

Un nivel, desde el punto de vista de *LibWiiEsp*, no es más que un escenario donde los actores participan en el juego. Está formado por tres elementos principales que son los actores, el conjunto de *tiles* que componen el propio nivel y una imagen de fondo. También tiene asociada una pista de música propia, una imagen desde la que se toman los *tiles* que se dibujan, y unas coordenadas de *scroll* para la pantalla, así como un ancho y un alto.

El escenario que presenta un nivel es de forma rectangular, y tiene un ancho y un alto definidos por el número de *tiles* que lo componen. A pesar de ello, en todo momento están disponibles sus medidas en píxeles. Dentro de este escenario, todos los elementos tienen una pareja de coordenadas que indican la distancia horizontal hacia la derecha respecto al límite izquierdo del nivel (coordenada X), y la distancia vertical hacia abajo respecto al límite superior del nivel (coordenada Y). Así pues, el origen de coordenadas del nivel (el punto (0,0)) es el vértice superior izquierdo del rectángulo que forma el escenario.

En la pantalla sólo puede dibujarse una parte del escenario del nivel, ya que ésta tiene un tamaño determinado por el modo de vídeo. La parte del nivel que se dibuja en la pantalla en un momento determinado es la ventana del nivel, y es un rectángulo con las mismas medidas que la pantalla, y que tiene unas coordenadas respecto al punto (0,0) del escenario. Estas coordenadas son el *scroll* del nivel, y se puede modificar mediante el método *moveScroll*.

La imagen de fondo del nivel debe tener el tamaño de la pantalla completa (para una televisión PAL de proporciones 4:3, las medidas en píxeles son 640x528), y será fija durante todo el transcurso del nivel. La idea de esta imagen de fondo estática es para situar un paisaje que acompañe al nivel, ya que la sensación de avance se

produce con los propios *tiles* que componen la estructura del nivel.

Existen dos tipos de *tiles* en un nivel, que se distinguen únicamente en que los *tiles* no atravesables tienen un rectángulo de colisión con el mismo tamaño y posición, y por lo tanto, provocará que un actor que colisione con él pueda reaccionar de una manera; sin embargo, los *tiles* atravesables no disponen de esta figura de colisión, y tienen únicamente un objetivo decorativo, para dotar de mayor detalle al escenario del nivel, pero no provocarán una colisión cuando un actor los toque. Todos estos *tiles* se almacenan en la misma estructura (de tipo *Escenario*), y para comprobar si un actor concreto colisiona con algún elemento del nivel se proporciona el método *colision* (información útil para saber, por ejemplo en un juego de plataformas, si el actor está cayendo o está sobre una plataforma). Si el programador necesita un mayor detalle en la detección de colisiones entre actores y escenario, siempre puede añadir los métodos y atributos que considere necesarios para ello al crear la clase derivada de Nivel. La detección de colisiones entre un actor y los *tiles* del escenario está optimizada para que sólo se evalúe la colisión con los *tiles* sobre los que se encuentre el actor, evitando cálculos innecesarios.

Se proporciona un método virtual puro *actualizarEscenario* en el que se pueden implementar comportamientos para el escenario (como por ejemplo, plataformas destructibles, o en movimiento, además de cambiar las coordenadas del *scroll* de la pantalla), y un método que dibuja la ventana del nivel en la pantalla de la consola.

Los actores que participan en un nivel se distinguen entre los actores que son dirigidos por los jugadores, y los actores que controla la máquina. Cada uno de ellos se almacena en una estructura distinta, y dispone de un método concreto para actualizarlo. Concretamente, el método para actualizar los actores no jugadores es *actualizarNpj*, en el que se supone que se debe implementar la actualización de todos los actores controlados por la máquina (la decisión sobre cómo llevar a cabo esta operación queda, por entero, en manos del programador). Igualmente, para actualizar los actores jugadores, se dispone del método *actualizarPj*, que debe actualizar a un único jugador en cada llamada, y lo hace recibiendo el código identificador único del jugador, y el mando asociado a él.

Hay que destacar especialmente la filosofía de esta clase Nivel junto con la clase Actor. En cada clase derivada de Actor se debe definir el comportamiento de cada actor dependiendo de su estado actual. Sin embargo, las transiciones entre estados de un actor pueden realizarse desde el Nivel en el que el actor está participando, o bien en el propio actor, ya que éste dispone de una referencia constante al Nivel en el que se encuentra, aunque lo que se recomienda es tomar la primera opción, y situar las transiciones entre estados en el método de actualización del Nivel, de tal manera que los actores respondan con un comportamiento u otro, dependiendo de cómo se les haya definido, y siempre según el estado que tengan activado.

Una vez descritos los elementos que conforman el escenario de un nivel, hay que

detallar especialmente el proceso de carga de un nivel desde un archivo TMX creado con el editor de mapas de *tiles* Tiled [11]:

- Cuando se crea un nivel, el constructor recibe un archivo TMX que se carga desde la tarjeta SD gracias a la clase Parser. Una vez en memoria, se lee el ancho y alto del nivel en *tiles*, y el ancho y el alto en píxeles de un único *tile* (las medidas de un *tile* deben ser múltiplos de 8, ya que llevarán una imagen asociada). A partir de estos datos, se calcula el ancho y alto del escenario del nivel en píxeles.
- El proceso de lectura desde el archivo TMX continúa leyendo las propiedades del mapa, que son *imagen_fondo* (que es el código que debe tener la imagen de fondo del nivel en la galería de medias), *imagen_tileset* (código que la imagen del tileset deberá tener asociado en la galería de medias del sistema) y *musica* (código identificador de la pista de música en la galería de medias del sistema). Tras leer las propiedades del mapa de *tiles*, se procede a leer las tres capas que debe contener el archivo TMX, que son las siguientes:
 1. Capa 'escenario': capa de patrones del editor Tiled, debe contener los *tiles* atravesables.
 2. Capa 'plataformas': capa de patrones del editor Tiled, que contiene los *tiles* no atravesables (los que tendrán figura de colisión asociada).
 3. Capa 'actores': capa de objetos del editor Tiled. Cada objeto definido en esta capa debe tener la propiedad XML con la dirección absoluta en la tarjeta SD del archivo XML de descripción del actor como valor. Si además, el actor es un actor jugador, se espera que tenga otra propiedad llamada *jugador*, y cuyo valor debe ser el código identificador del jugador. También debe tener el identificador de su tipo en el campo *Tipo*.
- Hay que tener en cuenta otro detalle importante, y es que, al llamar al constructor, la información de los actores se guardan en una estructura temporal. Por ese motivo, existe el método virtual puro *cargarActores*, que ha de ser implementado por el programador al derivar la clase Nivel, y que se debe encargar de recorrer esta estructura temporal, crear cada actor utilizando el constructor de la clase correspondiente al tipo del actor, y almacenarlo en la estructura correspondiente (dependiendo de si es un actor jugador o no jugador). Por último, es conveniente que este método vacíe la estructura temporal para no malgastar memoria.

Esto último es necesario ya que, en la propia clase abstracta Nivel que forma parte de *LibWiiEsp* no se conoce qué clases derivadas habrá de Actor, y por tanto, se deja en manos del programador implementar la creación de Actores en el nivel. A continuación, se establece el *scroll* del nivel al valor (0,0), y se da por concluida la carga del nivel. Para conocer todos los detalles sobre la creación de niveles con el editor Tiled, consultar el manual de *LibWiiEsp*.

4.12.3. Juego

Esta clase abstracta proporciona una plantilla sobre la que construir el objeto principal de toda aplicación desarrollada con *LibWiiEsp*. Consiste en dos apartados bien diferenciados, que son la inicialización de todos los subsistemas de la consola y de la biblioteca (esto se realiza desde el constructor), y la ejecución del bucle principal de la aplicación. Como clase abstracta que es, no se puede instanciar directamente, si no que hay que crear a partir de ella una clase derivada en la que se definan los métodos que el programador considere oportunos para gestionar el programa.

Además de la inicialización de la consola y el control del bucle principal del programa, esta clase también se encarga de gestionar la entrada de hasta cuatro mandos en la consola. Para ello, dispone de un diccionario en el que cada jugador, que está identificado por un código único (de tipo *string*), tiene asociado un mando concreto de la consola. Para acceder al mando de un jugador, basta con buscar en el diccionario mediante el código identificativo del éste.

El constructor de la clase recibe como parámetro un archivo XML con un formato concreto, en el que se deben especificar las opciones de configuración para el programa, como son el nivel de registro de mensajes deseado y la ruta completa del archivo de registro que se generará en el caso de estar el subsistema activado, el color considerado transparente y que no se dibujará en la pantalla (en formato 0xRRGGBBAA), el número de fotogramas por segundo (FPS) que se quiere que tenga el videojuego, la ruta absoluta hasta el archivo XML donde se especifican los recursos multimedia que deben ser cargados en la galería de medias, la ruta absoluta hasta el archivo XML donde se especifican las etiquetas de texto del sistema de internacionalización y el nombre del idioma por defecto, y por último, la configuración de jugadores, consistente en cuatro atributos de tipo cadena de caracteres en los que se deben especificar los códigos identificadores para cada uno de los jugadores.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <conf>
3   <log valor="/apps/wiipang/info.log" nivel="3" />
4   <alpha valor="0xFF00FFFF" />
5   <fps valor="25" />
6   <galeria valor="/apps/wiipang/xml/galeria.xml" />
7   <lang valor="/apps/wiipang/xml/lang.xml" defecto="english" />
8   <jugadores pj1="pj1" pj2="pj2" pj3="" pj4="" />
9 </conf>
```

El constructor de la clase Juego, que debe ser llamado en el constructor de la correspondiente clase derivada, se encarga en primer lugar de montar la primera partición de la tarjeta SD (debe ser una partición con sistema de ficheros FAT), cargar el árbol XML del archivo de configuración en memoria mediante la clase Parser, e inicializar el sistema de registro de mensajes según se haya especificado en el archivo

de configuración. Si en alguna de estas operaciones se produjera un error, se saldría del programa mediante una llamada a la función *exit*.

A partir de ese momento, el constructor inicializará los sistemas de vídeo (clase Screen), controles (clase Mando), sonido (clase Sonido) y la biblioteca *FreeType2* de gestión de fuentes de texto (clase Fuente). A continuación, establece el color transparente del sistema, el número de FPS, leerá la configuración de los jugadores, y creará una instancia de Mando para cada jugador que, en el archivo de configuración, no tenga una cadena vacía como identificador. Posteriormente, guarda cada mando asociado al código del jugador correspondiente en el diccionario de controles.

Por último, se cargan todos los recursos media en la galería, y las etiquetas de texto para los idiomas del sistema en la clase Lang.

Después de haber inicializado la consola partiendo del archivo de configuración, el método virtual *run* proporciona una implementación básica del bucle principal del programa, en la que se controlan las posibles excepciones que puedan lanzarse (las cuales captura y almacena su mensaje en el archivo de mensajes), se mantiene constante la tasa de fotogramas por segundo, y se actualizan todos los mandos conectados a la consola y se gestiona correctamente la actualización de los gráficos a cada fotograma. Junto a este método también se proporciona un método virtual puro, llamado *cargar*, y que se ejecuta antes de entrar en el bucle principal. Este método permite realizar las operaciones que se estimen necesarias antes de comenzar la ejecución del bucle, en el caso de que fueran necesarias (en caso contrario, bastaría con definir el método como una función vacía a la hora de derivar la clase Juego).

Igualmente, al ser el método *run* virtual, si el programador necesita otro tipo de gestión para el bucle principal de la aplicación, basta con que lo redefina en la clase derivada; a pesar de ello, tendrá disponible un sencillo ejemplo de control del bucle de la aplicación en la definición del método en la clase Juego.

Por último, especificar un detalle, y es que el destructor de la clase Juego se encarga de liberar la memoria ocupada por los objetos de la clase Mando, de tal manera que no hay que preocuparse por ello.

4.13. Ejemplos de *LibWiiEsp*

Una vez concluido el desarrollo de *LibWiiEsp*, quise aportar una serie de ejemplos para ilustrar hasta qué punto es útil la herramienta con la intención añadida de que sirvieran como iniciación al uso de la biblioteca. En esta sección se trata el funcionamiento de los tres juegos que acompañan a *LibWiiEsp*, explicando los detalles más relevantes de su implementación.

4.13.1. Arkanoid

Este primer juego es un clon de clásico de 1986, publicado por la empresa japonesa Taito. Consiste en una bola que se mueve por la zona de juego, rebotando en las paredes, y con la cual el jugador debe intentar destruir una serie de ladrillos que se encuentran en el escenario. Para conseguir su objetivo, el jugador controla una pala con la que debe evitar que la bola caiga por la zona inferior de la pantalla, en cuyo caso pierde una oportunidad. Además, cuando se destruye un ladrillo puede generarse aleatoriamente un objeto que cae verticalmente, y que si es recogido con la pala, puede aportar beneficios al jugador (una oportunidad extra, un multiplicador temporal de puntuación obtenida, o un aumento del tamaño de la pala).

La novedad de este desarrollo respecto al original es que se permite controlar la pala con los botones de la cruceta del mando, o bien mediante el ángulo de cabeceo (al más puro estilo del Mario Kart Wii de Nintendo).

Entrando en el apartado técnico, el juego tiene cuatro actores, que son la bola, los ladrillos, la pala y los objetos, que tienen sus animaciones y figuras de colisión organizadas por estados, información que se carga desde sendos archivos XML. Los escenarios, que tienen toda la lógica del juego implementada en la clase Escenario (derivada de Nivel), están diseñados con el editor Tiled [11], y una última clase, llamada Juego, se encarga de controlar el bucle principal.

La pala se mueve horizontalmente según se indique mediante el mando hasta tocar los límites laterales de la zona de juego; por otro lado, los objetos simplemente caen hasta la zona inferior de ésta, y cuando llegan simplemente desaparecen. Los ladrillos no se mueven.

El detalle más reseñable es el movimiento de la bola, que se compone de dos variables, velocidad y dirección, que son el módulo y el ángulo de un vector de movimiento, respectivamente. Cada vez que hay un rebote de la bola, se calcula un nuevo vector de movimiento para ésta, y cada fotograma la bola por la zona de juego se mueve en horizontal y en vertical un número de píxeles equivalentes a la parte entera de las componentes horizontal y vertical del vector. Cuando se destruye un ladrillo, el módulo del vector aumenta un poco, hasta llegar al límite establecido para la velocidad máxima en el archivo XML de la bola. Cabe destacar también el hecho de que el punto de impacto de la bola con la pala influye en el nuevo vector que tendrá tras el rebote.

Este juego resulta muy sencillo de ampliar mediante escenarios nuevos, e incorpora el soporte de internacionalización (i18n) que incluye *LibWiiEsp*.

Por último, mencionar que durante el desarrollo de Arkanoid se tuvieron que efectuar algunas modificaciones en la biblioteca, principalmente relacionadas con la clase abstracta Actor. En su momento, se diseñó esta clase de tal forma que se facilitaba el

desarrollo de juegos de plataforma, pero rápidamente se comprobó que atributos como la dirección no resultarían útiles como tipos enumerados, lo que ocasionó que se eliminara esa característica. También hubo que corregir la representación del mapa de *tiles* del escenario, ya que antes se guardaba en un conjunto *set*, y por tanto había que recorrer todos los *tiles* para identificar si un actor colisionaba con el escenario. Con la implementación actual, se consigue optimizar mucho el rendimiento de la detección de colisiones, ya que sólo se evalúan los *tiles* que se encuentren alrededor del actor.

En la figura 4.22 puede observarse una captura de pantalla de este juego.

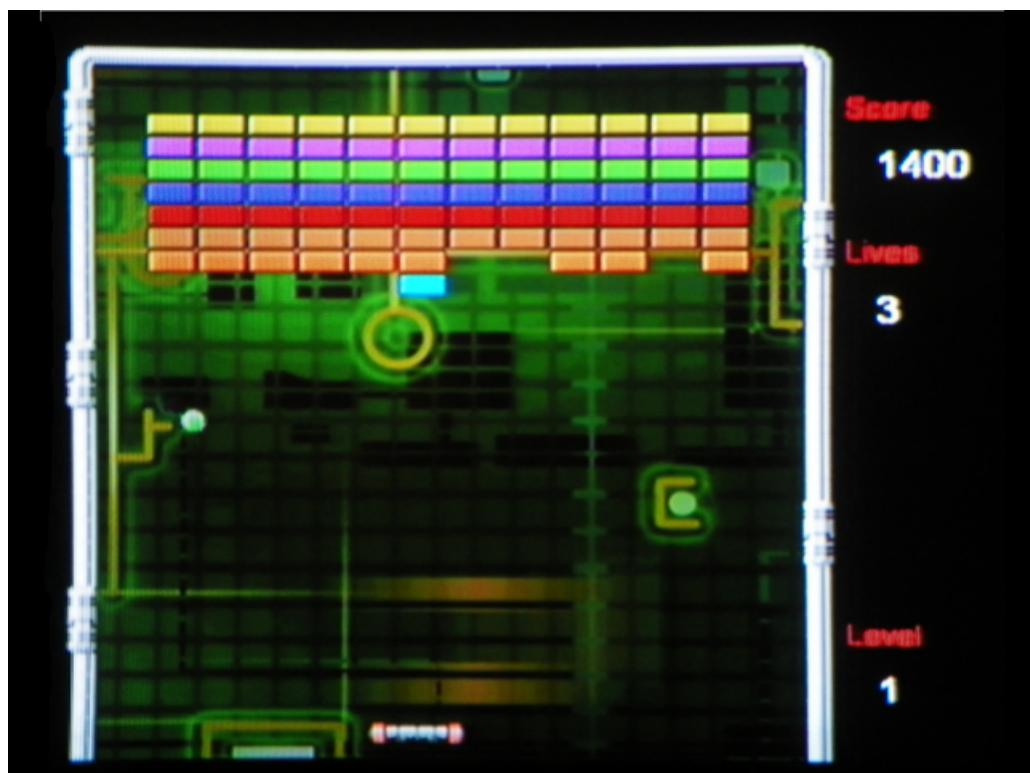


Figura 4.22: Captura de pantalla de Arkanoid

4.13.2. Duck Hunt

El segundo juego desarrollado es una versión del juego de Nintendo de 1984. A diferencia del original, en esta implementación de Duck Hunt pueden jugar dos personas simultáneamente, y los patos salen cada poco tiempo desde uno de los laterales de la pantalla hasta el otro, con un ángulo de inclinación. El objetivo del juego es llegar a abatir una cierta cantidad de patos, configurable desde el archivo XML del escenario.

En esta ocasión, son sólo dos los actores que participan en el juego, siendo uno

los patos y el otro el que controla los dos puntos de mira.

La clase Escenario, derivada de Nivel, se encarga de generar un pato nuevo a uno de los lados de la pantalla cada cierto tiempo, siendo este periodo un intervalo que va desde el segundo y medio a los tres segundos. El lateral del escenario en el que aparece el pato es aleatorio, al igual que el ángulo que tomará en su vuelo, que puede ir desde -30 grados hasta +30 grados. Si el pato escapa sin ser abatido, se elimina el objeto para no congestionar la memoria, sin embargo, cuando se dispara a un pato, éste queda suspendido durante medio segundo en el lugar del impacto (con una animación que indica que ha sido alcanzado), y después cae verticalmente hacia la zona inferior de la pantalla.

Por otro lado, las coordenadas de los puntos de mira equivalen al puntero infrarrojo de su correspondiente mando asociado, y un pato será derribado cuando un punto de colisión que coincide con el puntero de un mando colisione con el pato y además se haya pulsado el botón B del mando. Tras efectuar un disparo, no se puede volver a disparar hasta pasados 0,8 segundos, que es el tiempo que dura la recarga de la escopeta. Esto supone un cambio de estado para el actor del punto de mira, que vuelve al estado normal tras agotarse el periodo de tiempo indicado.

Al desarrollar este juego se realizaron modificaciones a la clase Nivel de *LibWiiEsp*, ya que los patos no se dibujaban bien cuando se acercaban al lateral izquierdo de la pantalla. El error consistía en que, para determinar si un pato estaba en pantalla o no, se tomaba únicamente la referencia de su punto superior izquierdo, en lugar de utilizar ambos laterales del actor. Al solucionar este error, se aprovechó para solucionar la misma situación que ocurría con los componentes del mapa de *tiles*, de tal manera que se dibujara correctamente el lateral izquierdo de un escenario compuesto por *tiles*.

En la figura 4.23 puede observarse una captura de pantalla de este juego.

4.13.3. Wii Pang

El tercer y último juego completo de ejemplo que acompaña a *LibWiiEsp* es Wii Pang, una versión sencilla del clásico Pang publicado por Mitchell Co. en 1989. En este juego controlamos a un personaje que puede moverse horizontalmente sobre el escenario, hasta tocar las paredes laterales de éste, y que debe evitar ser aplastado por unas bolas que rebotan con los límites del nivel. Para destruir las bolas, puede lanzar ganchos, que van subiendo hacia la parte superior de la pantalla con una dirección vertical desde el punto donde se han lanzado. Cuando un gancho colisiona con una bola, ésta se deshace, produciendo que dos bolas de un tamaño inferior aparezcan en su lugar, así hasta que se destruyen las bolas de tamaño más pequeño, que no generan ninguna otra nueva.



Figura 4.23: Captura de pantalla de Duck Hunt

Las bolas tienen un movimiento horizontal con velocidad constante, y su movimiento vertical se ve modificado por una aceleración de gravedad, de tal manera que resulta una parábola. Para implementar este movimiento, una bola que caiga verá incrementada su velocidad vertical cada fotograma, hasta que colisione con el suelo, momento en el que se le asigna su velocidad inicial, pero hacia arriba. Esta velocidad va descendiendo hasta que llega a cero, momento en el que alcanza el punto más alto de rebote, y de nuevo comienza a bajar, ya que la velocidad cambia de signo y comienza a aumentar de nuevo en función de la aceleración de gravedad. Cada tamaño de bola llegará a una altura determinada por una velocidad inicial diferente del resto de tamaños de bola.

El personaje podrá lanzar dos ganchos simultáneos. Estos ganchos empiezan siendo un objeto pequeño a nivel del suelo del escenario, pero cada fotograma van creciendo de tamaño hacia arriba hasta que llegan a tocar la parte superior del escenario, momento en el que desaparecen. Si colisionan con una bola, desaparecen también. El método de dibujo de este actor recorre toda la extensión vertical de éste, de arriba hacia abajo, dibujando en cada paso una parte del cable que acompaña al gancho. También su figura de colisión asociada va aumentando de tamaño a medida que crece.

Otro detalle interesante es el movimiento del personaje cuando es aplastado por una bola, momento en el que, con una animación de muerte, inicia un movimiento

parabólico similar (pero no igual) al de las bolas, rebotando con el lateral del escenario y después cayendo hacia fuera de la pantalla.

Un nivel acaba cuando no quedan más bolas en el escenario, y cuando el personaje muere, tiene que volver a empezar el nivel desde el principio. Al igual que ocurre con Arkanoid, resulta muy sencillo añadir escenarios nuevos al juego.

Respecto a las modificaciones implementadas a *LibWiiEsp* en base al desarrollo de este juego, el principal cambio constó en dar la posibilidad de redefinir el método de dibujo de los actores, ya que los ganchos son actores cuyo tamaño no se mantiene constante durante la ejecución del programa. También hubo que añadir un método nuevo para saber si un actor colisiona con los cuatro bordes del escenario, ya que el que había hasta ese momento disponible únicamente indicaba si un actor chocaba con un componente del escenario.

En la figura 4.24 puede observarse una captura de pantalla de este juego.



Figura 4.24: Captura de pantalla de Wii Pang

Hay que mencionar que las capturas de pantalla de los tres juegos no tienen una alta calidad por haberse realizado mediante fotografías disparadas a la pantalla de una televisión CRT (es decir, de tubo de rayos catódicos).

Capítulo 5

Pruebas

5.1. Plan de pruebas y validación del sistema

El plan de pruebas se ha realizado durante el desarrollo de la biblioteca y una vez se finalizó el producto, y ha incluido los siguientes tipos de pruebas:

- **Pruebas de clase:** Tras completar el desarrollo de cada una de las clases o módulos que componen *LibWiiEsp* se procedió a realizar una serie de pruebas concretas para el apartado correspondiente.
- **Pruebas de sistema:** Una vez completada la biblioteca, se realizaron pruebas de conjunto, en las que se utilizaban prácticamente todos los módulos desarrollados, y en los que se comprobó la cohesión entre todas las partes del sistema.
- **Pruebas de *makefile*:** Este apartado merece una mención aparte, y es que el archivo *makefile* instala, desinstala y empaqueta para la distribución a la biblioteca, además de realizar una recompilación eficiente.
- **Pruebas de juegos:** Las pruebas relacionadas con los tres juegos de ejemplo desarrollados se realizaron al completar éstos.

5.2. Especificación del diseño de pruebas

- **Pruebas de clase:** Estas pruebas se fueron realizando durante el desarrollo, y se han ido comentando en los puntos correspondientes a cada uno de los módulos implementados. A modo de vista general, estas pruebas, en lo general comprobaciones de *caja negra* y *caja blanca*, han ido enfocadas a probar cada uno de los métodos implementados en cada clase, de tal manera que se pudieran comprobar los resultados para datos de entrada válidos y no válidos.
- **Pruebas entre módulos:** Una vez se finalizó el desarrollo, la batería de pruebas realizada en ese momento estuvo enfocada a comprobar las relaciones entre distintas clases que deben trabajar haciendo uso las unas de las otras. Concretamente, las pruebas de sistema se realizaron atendiendo a las siguientes relaciones entre clases:

- Screen (4.2), Imagen (4.5) y Animación (4.9): pruebas orientadas a comprobar que todas las funciones de dibujo de texturas operaban correctamente.
 - Screen (4.2) y Fuente (4.5): conjunto de validaciones que evaluaban la escritura de textos en la pantalla, haciendo uso de fuentes de texto y caracteres con diversas codificaciones.
 - Sonido y Música (4.5): pruebas que comprobaron que la reproducción de pistas de música y efectos de sonido se producía correctamente, sin provocar errores en el procesador DSP.
 - Sdcard (4.3) y los recursos multimedia (4.5): esta batería de pruebas comprobó que las clases que gestionan los recursos multimedia realizaban un uso acceso correcto a la tarjeta SD.
 - Sdcard (4.3) y Parser (4.6): se realizaron pruebas para verificar que el parser XML trabajaba correctamente al cargar archivos desde la tarjeta SD.
 - Sdcard (4.3) y Logger (4.10): pruebas que comprobaban la correcta escritura del archivo de registro de mensajes en la tarjeta SD.
 - Parser (4.6), Galería (4.7), Lang (4.8), Actor, Nivel y Juego (4.12): estas pruebas estaban orientadas a comprobar que las clases que utilizan XML lo hacían de una forma correcta.
 - Galería (4.7) y recursos multimedia (4.5): este conjunto de pruebas se encargó de demostrar que la galería de recursos operaba correctamente con las clases que gestionan éstos.
 - Actor y Nivel (4.12): multitud de pruebas se encargaron de validar el tratamiento que da la clase Nivel a los distintos Actores.
 - Nivel y Juego (4.12): pruebas enfocadas a validar el tratamiento que la clase Juego da a los objetos de clases derivadas de Nivel.
 - Juego (4.12) y Mando (4.4): conjunto de validaciones que comprobaban la cohesión entre la clase Juego y los distintos objetos de la clase Mando.
- **Análisis estático de código:** Además de comprobar la cohesión entre los distintos módulos que componen *LibWiiEsp*, se utilizó la herramienta *CppCheck* para validar todo el código fuente generado de forma estática. Este software realiza comprobaciones sobre los archivos fuente, asegurando que no ocurran fallos típicos como fugas de memoria, variables no inicializadas, reservas de memoria no utilizadas, punteros nulos, uso de funciones obsoletas, así como varios errores comunes sobre el uso del lenguaje y la biblioteca estándar de plantillas (STL).
- **Pruebas de *makefile*:** estas pruebas se realizaron a lo largo de todo el desarrollo, ya que han habido muchas ocasiones en las que se ha añadido funcionalidad al archivo de recompilación. Las comprobaciones realizadas se realizaron objetivo por objetivo.

- **Pruebas de juegos:** la batería de pruebas de juegos se condensaron en un plan intensivo de *testing*, en el que varias personas se dedicaron a jugar a las tres demostraciones para comprobar su funcionamiento.

5.3. Especificación de los casos de prueba

A la hora de realizar las comprobaciones por módulos, se disponía de unos datos de entrada que se proporcionaban a los métodos probados de la clase correspondiente. Se probaron todos los métodos de todas las clases. Esta entrada incluía tanto información válida, de la que se había previamente calculado el resultado que debían producir, como datos no válidos, que debían generar una excepción controlada y su correspondiente entrada en el registro de mensajes del sistema.

Las pruebas de sistema estuvieron basadas en esta misma metodología, y supusieron una ampliación de los casos de prueba para módulos individuales. Se hizo especial hincapié en el intercambio de datos entre distintas clases, en forma de paso de parámetros, mensajes o activación y desactivación de banderas. Las comprobaciones se realizaban tanto a la salida de un primer módulo como a la entrada del siguiente en la ejecución. La validación con *CppCheck* se realizó con la opción de activar todas las comprobaciones, y se lanzó para las carpetas *include* y *src*.

Las pruebas realizadas al archivo *makefile* se fueron realizando a medida que se desarrollaba nueva funcionalidad en él, y consistieron en comprobar que los comandos que se ejecutaban realizaban su tarea correctamente.

Respecto a las pruebas realizadas a los juegos, se realizaron por parte de personas que ya tenían experiencia en los juegos clásicos en los cuales se basan las demostraciones de *LibWiiEsp*, y se centraron principalmente en la detección de comportamientos anómalos de los juegos.

5.4. Documentación de la ejecución de las pruebas

La batería de pruebas individual de cada una de las clases del sistema arrojó, para algunas de ellas, diversos fallos de baja consideración, que fueron resueltos en poco tiempo. Principalmente, dieron problemas las funciones de dibujo de texturas, ya que se producían fallos al dibujarlas en la pantalla debido a errores en la creación del formato RGB5A3. Los efectos de sonido no se reproducían correctamente, al no estar bien tratados con el programa SoX [3], pero esto se resolvió leyendo en profundidad la documentación del software. Otra situación redundante ocurría con errores de memoria, en los que los punteros no se estaban gestionando correctamente, pero que pudo resolverse tras utilizar la herramienta *CppCheck*. En general, prácticamente todos los módulos del sistema necesitaron ajustes tras realizarse las pruebas.

Aparte de los típicos errores por descuido a la hora de implementar el código, no ocurrieron incidentes reseñables durante el desarrollo de las pruebas.

Capítulo 6

Conclusiones

6.1. Aspectos generales

La biblioteca desarrollada como resultado del proyecto es una herramienta completa para la creación de videojuegos en dos dimensiones para Nintendo Wii, utilizando como base el lenguaje de programación C++. Además de proporcionar abstracción sobre todos los aspectos que dificultan el desarrollo en esta videoconsola, supone una gran fuente de información para todo aquel interesado en el funcionamiento de los distintos sistemas de ésta, debido principalmente a la enorme cantidad de documentación generada, tanto en forma de manual como las descripciones de las clases recogidas en la referencia y en este propio documento.

Por supuesto, *LibWiiEsp* necesita continuar su desarrollo con el objetivo de refinar cada vez más sus componentes, aportar funcionalidad nueva y revisar la existente. Sin embargo, a partir de una herramienta como ésta puede surgir una comunidad que la perfeccione y le de uso, hecho que se hace más real por cuanto se trata de una biblioteca totalmente libre.

Un último aspecto a destacar es lo útiles que resultan, didácticamente hablando, tanto *LibWiiEsp* como su documentación y los tres juegos de ejemplo. Se ha conseguido aportar no sólo una herramienta de desarrollo, si no una vía para acercar el desarrollo de software para videoconsolas a la gente *de a pie*, demostrando que es posible crear videojuegos para plataformas cerradas gracias al trabajo (prácticamente en todas las ocasiones, **libre**) de muchos *sceners* a lo largo del mundo.

6.2. Conocimientos adquiridos

Durante el desarrollo se han intentado aplicar todos los conocimientos adquiridos a lo largo del plan de estudios cursado, concentrándolos en la creación de una herramienta software útil. Además de todo lo relacionado con la programación en lenguaje C++, se ha utilizado el enfoque de la programación orientada a objetos (ya que el producto final es una biblioteca escrita por completo haciendo uso de este paradigma), técnicas de optimización, se han empleado patrones de diseño según las circunstancias lo requerían y se ha utilizado una metodología de desarrollo

iterativa. Los nuevos conocimientos incluyen una profundización exhaustiva en los archivos de órdenes de recompilación (*makefile* [13]), la investigación y obtención de información sobre un sistema cerrado como es Nintendo Wii, la creación de un sistema sobre otro existente partiendo de poca o nula documentación, el trabajo con la tecnología XML, las transformaciones entre formatos de textura a bajo nivel (bit a bit) y todos los conceptos relacionados con los formatos de vídeo y audio, la utilización de periféricos distintos a los comunes ratón y teclado, el uso de bibliotecas externas para tareas tan comunes como acceder a un dispositivo de almacenamiento (utilizando *libFat*) o escribir textos en una pantalla (*FreeType2*) y los conceptos de *Endian* y alineación de datos y zonas de memoria.

Respecto a la redacción de la memoria, se ha profundizado en la generación de documentación técnica, tanto en lo que a manuales se refiere como con las descripciones de los distintos módulos que componen la herramienta. Al no tratarse de un proyecto de creación de un software de gestión, se ha intentado aplicar un punto de vista diferente: la documentación recogida en esta memoria trata, en primer lugar, de mostrar cómo trabaja a bajo nivel Nintendo Wii con la biblioteca *libogc* [12], y después se hace una descripción completa sobre la funcionalidad que aporta *Lib-WiiEsp*. Durante la redacción de toda la documentación se ha aprendido a utilizar L^AT_EX, un sistema de composición de textos muy útil y extendido, así como a trabajar con el diseñador de diagramas *Dia*. Además, para la generación automática de la documentación relativa a los métodos y atributos públicos definidos en cada módulo, se ha utilizado el software *Doxygen* [4].

6.3. Posibles mejoras

A continuación se listan algunas mejoras que sería deseable añadir a la biblioteca en futuras versiones:

- **Sistema de sonido 3D:** implementar un sistema de sonido 3D, jugando con el volumen de los dos canales de audio de los que se dispone a la hora de reproducir un sonido.
- **Controlar los acelerómetros de los mandos:** en este punto, se depende de que los creadores de *libogc* implementaran correctamente la lectura de estos dispositivos.
- **Soporte para todos los periféricos de Wii:** como la guitarra de *Guitar Hero*, el Wii Fit, el mando clásico o el mando de Nintendo *Game Cube*.
- **Soporte para utilizar los puertos USB traseros:** así como el soporte de almacenamiento más sencillo de utilizar es el lector de tarjetas SD del que dispone la videoconsola, los puertos USB de Nintendo Wii tienen tecnología USB 2.0, por lo que se conseguiría un mayor rendimiento a la hora de realizar operaciones de lectura y escritura.

6.4. Futuro del proyecto

Aunque a día de hoy es una herramienta completa, *LibWiiEsp* necesita seguir creciendo, tanto para corregir algunos defectos que puedan presentarse a medida que se utilice la herramienta, como para proporcionar nueva funcionalidad, por ejemplo, dar soporte a un mayor número de formatos para los recursos multimedia, aumentar el número de figuras de colisión disponibles y proporcionar soporte para hilos y utilización de protocolos de red.

A corto plazo se quiere crear una comunidad de usuarios alrededor de este producto, de tal manera que se fomente su utilización y se perfeccione a partir de más puntos de vista que no únicamente el de su creador original.

También se ha pensado utilizar el mismo conjunto de módulos que componen *LibWiiEsp* y adaptarlo para que trabaje con otras videoconsolas actuales, como son *PlayStation Portable*, *Nintendo DS* o incluso *PlayStation 3*.

Bibliografía

- [1] Bernardo Cascales Salinas.
El libro de LaTeX.
Pearson Educación, 2003.
- [2] Bienvenido Trillo Sáez, Pedro Muñoz Rodríguez.
Iniciación a Gimp.
Grupo Editorial Universitario, 2007.
- [3] Cbagwell, Robs.
Forja de SoX.
<http://sox.sourceforge.net>.
- [4] Dimitri Van Heesch.
Pagina oficial de Doxygen.
<http://www.doxygen.org>.
- [5] Equipo Hackmii.
Pagina oficial de Homebrew Channel.
<http://hbc.hackmii.com>.
- [6] Ezequiel Vázquez de la Calle.
Forja de *LibWiiEsp* en RedIris.
<https://forja.rediris.es/projects/libwiewsp>.
- [7] Francisco Muñoz (Hermes).
Tutorial de Hermes de programación para Wii.
http://www.elotrolado.net/wiki/Curso_de_programacion.
- [8] G. Aburruzaga García, I. Medina Bulo, F. Palomo Lozano.
Fundamentos de C++.
Servicio de Publicaciones de la Universidad de Cádiz, 2001.
- [9] Gerardo Aburruzaga García.
Make. Un programa para controlar la recompilación.
<http://www.uca.es/softwarelibre/publicaciones/make.pdf>.
- [10] Nintendo Co.
Manual oficial de Nintendo Wii.
http://www.nintendo.es/NOE/es_ES/support/ayuda_wii_5251_7840.html.

- [11] Thorbjørn Lindeijer.
Pagina oficial de Tiled.
<http://www.mapeditor.org>.
- [12] Varios Autores.
Documentación de *libogc*.
<http://libogc.devkitpro.org>.
- [13] Varios Autores.
Manual completo de Make.
<http://www.gnu.org/software/make/manual>.
- [14] Varios Autores.
Página oficial de WiiBrew.
<http://wiibrew.org>.
- [15] Varios Autores.
Tutoriales de Scene Beta.
<http://wii.scenebeta.com/tutoriales/wii>.
- [16] Varios Autores.
Wiki de GameDev.
<http://wiki.gamedev.net>.
- [17] WinterMute.
Página oficial de DevKitPro.
<http://devkitpro.org>.

Apéndice A

Software utilizado

Doxxygen

Doxxygen [4] realiza una documentación automática de código fuente. Puede generar esta documentación en varios formatos, incluyendo HTML y L^AT_EX.

Dia

Dia es un editor de gráficos vectoriales el cual incluye distintas plantillas para distintos tipos de gráficos, como pueden ser UML, ERe, diagramas de flujo, esquemas Cisco de red y un larguísimo etcétera.

GNU Make

GNU Make [13] [9] es el programa de recompilación y de control de dependencias por excelencia. Se puede utilizar para compilar proyectos software en diversos códigos.

Tiled

Tiled [11] es un editor de mapas de *tiles* que permite construir escenarios para juegos en dos dimensiones. La gran ventaja que proporciona este software consiste en que exporta el escenario construido en formato XML, facilitando enormemente el proceso de carga y utilización de los niveles que se diseñen utilizando esta herramienta. Está licenciado con GPLv2.

Homebrew Channel

Homebrew Channel [5] es una aplicación no oficial para Nintendo Wii cuya principal funcionalidad es la de permitir la ejecución de software sin firmar digitalmente por Nintendo. Aunque no se ha publicado su código fuente, es de distribución gratuita, y su gran ventaja consiste en que utiliza el lector de tarjetas SD de la

videoconsola como medio de almacenamiento para los ejecutables. Tiene un aspecto gráfico realmente cuidado.

SoX

SoX [3] es una aplicación libre, que se ejecuta en línea de comandos, y que está considerada como la *navaja suiza de los formatos de sonido*. Permite cambiar prácticamente todos los aspectos de un archivo de audio, pudiendo trabajar con multitud de formatos. También puede aplicar varios efectos a estos archivos de sonido, e incluso reproducirlos.

Subversion

Subversion es un sistema de control de revisiones de código libre, y que permite gestionar eficiente y cómodamente los cambios aplicados a un proyecto por uno o varios desarrolladores.

DevKitPro

DevKitPro [17] es un conjunto de compiladores, bibliotecas estándar de C y C++ y varias utilidades que permiten compilar código fuente escrito en esos lenguajes para poder ser ejecutado en varias videoconsolas actuales. Concretamente, la versión para Nintendo Wii que he utilizado se denomina *DevKitPPC*.

LibOgc

LibOgc [12] es una biblioteca de muy bajo nivel, escrita en C, y que permite acceder a todo el hardware de la videoconsola Nintendo Wii. Está desarrollada por varios *sceners*, y trabaja junto con la herramienta *DevKitPPC*.

LibFat

Es una biblioteca libre que permite realizar operaciones con particiones cuyo sistema de ficheros sea FAT o FAT32. Se ha utilizado una versión adaptada para trabajar con Nintedo Wii.

LibFreeType2

Biblioteca libre con la que se pueden utilizar todo tipo de fuentes de texto en las aplicaciones, ya que proporciona una interfaz completa para cargarlas en memoria a partir de un archivo y trabajar con ellas. Se trata de una versión adaptada a Nintedo Wii.

TinyXML

TinyXml es una sencilla biblioteca libre, adaptada para trabajar con Nintendo Wii, y cuya utilidad reside en poder operar fácilmente con árboles XML cargados desde un archivo de datos que utilice este formato de almacenamiento.

LATEX

LATEX [1] es un sistema de composición de textos, orientado especialmente a la creación de libros, documentos científicos y técnicos que contengan fórmulas matemáticas. Se ha utilizado para la redacción de este documento y del manual de instalación y uso de la biblioteca. *LATEX* es software libre bajo licencia LPPL.

Gimp

Gimp [2] es un programa de edición de imágenes digitales en forma de mapa de bits, tanto dibujos como fotografías. Es un programa libre y gratuito. Forma parte del proyecto GNU y está disponible bajo la Licencia pública general de GNU.

Gantt Project

Gantt Project es un software escrito en Java y libre, cuyo propósito es el de realizar diagramas de Gantt que reflejen la planificación temporal de un proyecto.

Cppcheck

Cppcheck es una utilidad libre que se encarga de realizar un exhaustivo análisis estático de cualquier código escrito en C++. Indica desde posibles fugas de memoria hasta el uso de funciones obsoletas, pasando por analizar el uso de la biblioteca estándar de plantillas y la seguridad del sistema de excepciones.

Apéndice B

Manual de instalación y uso

B.1. Instalación del entorno de desarrollo

Esta sección del manual detalla paso a paso la instalación de estas dependencias en un sistema *GNU/Linux* de 32 bits, siendo el proceso prácticamente el mismo para sistemas de 64 bits (únicamente cambia la versión de las herramientas base, que deberá ser en este caso la adecuada para 64 bits). Para sistemas Windows también es posible desarrollar con *LibWiiEsp*, pero este manual no cubre este tipo de arquitecturas.

En primer lugar, hay que crear una carpeta accesible para todos los usuarios del sistema, donde irán emplazadas las herramientas y dependencias de *LibWiiEsp*. Lo ideal es crear un directorio dentro de /opt y otorgarle a éste los permisos necesarios, pero es posible realizar la instalación en nuestro directorio /home, de tal manera que sólo nuestro usuario pueda acceder a estas herramientas.

Suponiendo que se escoge la primera opción, se crea el directorio en /opt y se le asignan permisos para que todos los usuarios del sistema puedan hacer uso de lo que en él se almacene:

```
sudo mkdir /opt/devkitpro  
sudo chmod 777 /opt/devkitpro
```

A continuación, hay que descargar las herramientas que sirven de base para *LibWiiEsp*, que son el conjunto de compiladores, bibliotecas y binarios *DevKitPPC*, y la biblioteca de bajo nivel *libOgc*, en su versión 1.8.4, junto con la modificación de *libFat* 1.0.7 compatible con ella. Ambos recursos se encuentran disponibles en la forja del proyecto, en el paquete *Dependencias*. Se deben descargar en el directorio que acabamos de crear:

```
cd /opt/devkitpro  
wget http://forja.rediris.es/frs/download.php/2316/devkitPPC_r21-i686-linux.tar.bz2  
wget http://forja.rediris.es/frs/download.php/2315/libogc-1.8.4-and-libfat-1.0.7.tar.gz
```

El siguiente paso es descomprimir ambos ficheros y, si no queremos tener ocupado espacio innecesariamente, eliminarlos. Las instrucciones para ejecutar estas acciones

son:

```
tar -xvzf devkitPPC_r21-i686-linux.tar.bz2
tar -xvzf libogc-1.8.4-and-libfat-1.0.7.tar.gz
rm devkitPPC_r21-i686-linux.tar.bz2 libogc-1.8.4-and-libfat-1.0.7.tar.gz
```

Después de esto, es necesario establecer algunas variables de entorno para que el sistema sepa dónde localizar estas herramientas que acabamos de instalar. Las dos primeras consisten en la ruta hasta el directorio base *devkitpro*, y hasta el directorio donde se encuentran las herramientas como tal, concretamente *devkitpro/devkitPPC*. La tercera variable de entorno indica la dirección IP de nuestra Nintendo Wii dentro de la red local, dato necesario para ejecutar correctamente las aplicaciones que se desarrollen sin que haga falta copiar el ejecutable en la tarjeta SD de la consola cada vez que se genere uno nuevo.

Para establecer estas variables de entorno, basta con editar el archivo de configuración *./bashrc* del usuario con el que vayamos a desarrollar utilizando *LibWiiEsp*. Si por alguna causa fuera necesario que todos los usuarios del sistema tengan que usar estas herramientas, el archivo de configuración a editar sería */etc/bashrc*, o su equivalente en el sistema (por ejemplo, para una distribución *Ubuntu 10.10* de 32 bits, el archivo es */etc/bash.bashrc*).

Suponiendo que sólo nuestro usuario va a desarrollar utilizando *LibWiiEsp*, las instrucciones para establecer estas variables de entorno serían:

```
echo export DEVKITPRO=/opt/devkitpro >> ~/.bashrc
echo export DEVKITPPC=$DEVKITPRO/devkitPPC >> ~/.bashrc
echo export WILOAD=tcp:192.168.X.X >> ~/.bashrc
```

Un detalle, en estas instrucciones, *192.168.X.X* se debe sustituir por la dirección IP que tiene asignada la consola Nintendo Wii en la red local.

A continuación, podemos recargar el fichero de configuración *./bashrc* para tener listas las variables de entorno mediante la orden:

```
source ~/.bashrc
```

Llegados a este punto, el último paso para que el entorno de desarrollo funcione correctamente con *LibWiiEsp* es instalar la biblioteca propiamente dicha. Existen dos maneras de realizar esto, o bien podemos descargar la versión estable publicada en la forja, o también compilando el código disponible desde ésta. Para el primer caso, basta con descargar el paquete comprimido desde la página principal de la Forja del proyecto [6], copiarlo al directorio */opt/devkitpro*, y descomprimirlo allí.

Para compilar la biblioteca a partir de los fuentes, debemos ejecutar las siguientes líneas, siempre que se hayan instalado previamente las dependencias:

```
svn co https://forja.rediris.es/svn/libwiiesp/trunk libwiiesp
cd libwiiesp
make install
```

Y después de la instalación de la biblioteca, ya tenemos preparado nuestro sistema para comenzar con el desarrollo de videojuegos en dos dimensiones para Nintendo Wii.

B.2. Estructura de un proyecto con *LibWiiEsp*

Una vez instalado correctamente el entorno de desarrollo para utilizar *LibWiiEsp*, el siguiente paso es crear la estructura de archivos y directorios necesarios para trabajar con un nuevo proyecto para Nintendo Wii. Por supuesto, queda en manos del programador la decisión final sobre esta estructura de directorios, pero, debido a todo lo que hay que tener en cuenta a la hora de desarrollar con *LibWiiEsp* (sobre todo, en lo referente a la compilación y enlazado de los archivos que componen un proyecto), voy a presentar en esta sección un ejemplo de proyecto vacío que quedaría listo para comenzar el desarrollo. Además de tratar la estructura de directorios, también mostraré y explicaré el código de un archivo *makefile* que pueda trabajar con esta organización para el proyecto.

B.2.1. Estructura de directorios

En primer lugar, hay que crear un directorio base donde almacenar todo lo relativo a nuestro proyecto. La localización de este directorio en el sistema es indiferente, de tal manera que, para el ejemplo, lo haremos en nuestro directorio */home* con las instrucciones:

```
mkdir ~/juego  
cd ~/juego
```

Una vez dentro, la siguiente estructura de directorios sería más que suficiente para albergar todos los componentes del proyecto:

- *doc*: Documentación del proyecto.
- *lib*: Bibliotecas externas que se vayan a utilizar en el proyecto. Aquí se debe guardar cada biblioteca en un directorio separado. En cada directorio debe existir un archivo *makefile* el cual compile la biblioteca, y genere un archivo de enlazado estático *.a*, además los archivos de cabecera de la biblioteca externa tienen que estar justo bajo este directorio principal de la biblioteca externa.
- *media*: En este directorio se colocarán todos los archivos que contengan recursos multimedia que vayan a ser empleados en el proyecto. De momento, sólo están soportados imágenes *bitmap* de 24 bits de color directo, fuentes de texto soportadas por *FreeType2*, efectos de sonido en formato *PCM*, y pistas de música *MP3*.
- *src*: Aquí van los archivos fuente del proyecto.
- *xml*: Archivos de datos del proyecto. Como mínimo, aquí se encontrarán el archivo que describe los recursos multimedia que se cargarán en la galería de

medias, el archivo del soporte de internacionalización, y el archivo de configuración del programa.

B.2.2. El archivo *makefile*

La estructura de directorios, y los detalles que la acompañan (como las restricciones de estructuración a la hora de utilizar bibliotecas externas en el proyecto), se han definido así para trabajar con un archivo de compilación *makefile* concreto.

Este archivo de compilación implica una serie de modificaciones considerables en comparación con uno que genere un ejecutable para PC en entornos *GNU/Linux*, por lo que va a detallarse su funcionamiento sección a sección. El objetivo de este *makefile* es generar un ejecutable con extensión *.dol*, que puede lanzarse en una videoconsola Nintendo Wii. A continuación, se muestra un sencillo ejemplo de *makefile* compatible con la estructura de directorios planteada en el punto anterior:

```

1 # Informacion configurable
2 LOCALLIBS = tinyxml bullet
3 PROJECT = Wii Pang
4 TARGET = boot
5 BUILD = build
6 SOURCE = src
7 DEPSDIR = $(BUILD)
8
9 # Reglas de compilacion
10 .SUFFIXES:
11 include $(DEVKITPPC)/wii_rules
12 include $(DEVKITPPC)/libwiiesp_rules
13
14 # Generar una lista con todos los ficheros objeto del proyecto
15 CPPFILES = $(notdir $(wildcard $(SOURCE)/*.cpp))
16 OFILES = $(CPPFILES:.cpp=.o)
17 OBJS = $(addprefix $(CURDIR)/$(BUILD)/,$(OFILES))
18
19 # Variables para la compilacion
20 INCLUDE = $(foreach dir,$(LOCALLIBS),-I$(dir)) -I$(LIBOOGC_INC)
21 LIBPATHS = -L$(LIBOOGC_LIB) -L$(CURDIR)/$(BUILD)
22 VPATH = $(SOURCE)$(foreach dir,$(LOCALLIBS),:$(dir))
23 LIBS = $(LIBWIIESP_LIBS) -ltinyxml -lbullet
24 OUTPUT = $(CURDIR)/$(TARGET)
25 LD = $(CXX)
26
27 # Flags para la compilacion
28 CXXFLAGS      = -g -ansi -Wall $(MACHDEP) $(INCLUDE) -std=c++0x
29 OPTIONS       = -MMD -MP -MF
30 LDFLAGS        = -g $(MACHDEP) -Wl,-Map,$(notdir $@).map
31
32 # Objetivos
33 .PHONY: all $(BUILD) libs $(LOCALLIBS) listo run clean
34
35 all: $(BUILD) libs $(OUTPUT).dol listo
36
37 $(BUILD):
```

```

38      @ [ -d $(BUILD) ] || mkdir -p $(BUILD)
39
40 libs: $(LOCALLIBS)
41     @echo Bibliotecas externas listas
42
43 $(LOCALLIBS):
44     $(MAKE) --no-print-directory --silent -C lib/$@ \
45     mv lib/$@/*.a $(BUILD)
46
47 $(CURDIR)/$(BUILD)/%.o: $(CURDIR)/$(SOURCE) /%.cpp
48     $(CXX) -MMD -MP -MF $(DEPSDIR) /$*.d $(CXXFLAGS) -c $< -o $@
49
50 listo:
51     $(RM) $(SOURCE) /-g $(OUTPUT).elf.map
52
53 run:
54     wiiload $(TARGET).dol
55
56 clean:
57     for dir in $(LOCALLIBS); do $(MAKE) clean -C lib/$$dir; done
58     $(RM) -fr $(BUILD) $(OUTPUT).elf $(OUTPUT).dol *~ $(SOURCE)/*~
59
60 DEPENDS :=      $(OBJS:.o=.d)
61
62 $(OUTPUT).dol: $(OUTPUT).elf
63
64 $(OUTPUT).elf: $(OBJS)
65
66 -include $(DEPENDS)

```

En el primer bloque que aparece en el ejemplo, se definen una serie de variables que indican los directorios que forman parte del proyecto. *LOCALLIBS* debe recoger los nombres, separados por un espacio, de los directorios principales de cada biblioteca externa que se vaya a emplear en la compilación. La variable *PROJECT* indica el nombre completo del proyecto, y *TARGET* el nombre, sin extensión, del ejecutable que se generará. Para que el programa pueda ser ejecutado con *Homebrew Channel*, se recomienda que se nombre *boot*. Por último, *BUILD* indica el directorio que se creará cuando se ordene la compilación del proyecto para almacenar todos los ficheros objeto del proyecto, *SOURCE* es el directorio donde están todos los archivos fuente, y *DEPSDIR* es donde se generarán los archivos que indican las dependencias entre módulos del sistema, que debe ser el mismo directorio que *BUILD*.

A continuación, se deben importar las reglas de compilación para Nintendo Wii, tanto las necesarias para utilizar las herramientas de *DevKitPPC*, como las propias de *LibWiiEsp*. Para ello, antes hay que limpiar las reglas implícitas existentes, lo cual se consigue con la instrucción *.SUFFIXES*:

El bloque de tres instrucciones que viene a continuación se encarga de almacenar en variables una lista de todos los archivos *.cpp* que se encuentran en el directorio de fuentes, y otra lista con la ruta absoluta de los ficheros objeto que se generarán al compilar (es decir, archivos con extensión *.o* en la carpeta indicada por la variable

BUILD).

El cuarto bloque de instrucciones recoge los directorios donde se encuentran los archivos de cabeceras externas (los almacena en la variable *INCLUDE*), los directorios en los cuales hay que buscar las bibliotecas de enlazado estático (guardados en la variable *LIBPATHS*), establece el *VPATH*, crea la lista de bibliotecas a enlazar estáticamente (tomando la variable *LIBWIIESP_LIBS* para contar con todo lo que necesita un ejecutable generado con *LibWiiEsp*, pero se le debe añadir además lo necesario para enlazar también las bibliotecas externas), y por último, indica la ruta absoluta hasta el ejecutable sin extensión y define el enlazador que se utilizará.

El bloque siguiente establece los *flags* de compilación necesarios para generar un ejecutable de extensión *.elf*, que posteriormente se transformará a otro con extensión *.dol*. Y justo después comienzan los objetivos del *makefile*, que se describen a continuación:

- *all*: Crea un ejecutable *.dol* a partir del código fuente que se encuentre en el directorio indicado en la variable *SOURCE*. Es el objetivo por defecto al ejecutar *make*.
- *\$(BUILD)*: Crea el directorio donde se crearán todos los ficheros objetos del proyecto.
- *libs* y *\$(LOCALLIBS)*: Se encargan de compilar las bibliotecas externas, de empaquetarlas en ficheros de enlace estático y mover éstos al directorio de los ficheros objeto.
- *\$(CURDIR)/\$(BUILD)/%.o*: Genera un fichero objeto en el directorio *\$(BUILD)* por cada módulo que se encuentre en el directorio de los ficheros fuentes.
- *listo*: Objetivo que limpia un archivo de extensión *.elf.map*.
- *run*: Lanza la utilidad *wiiload* para enviar el ejecutable a la Nintendo Wii. Ésta debe tener abierto el *Homebrew Channel* y estar conectada a la red local con la misma dirección IP que se indicó en la variable de entorno *WIILOAD*, de otro modo no ejecutará el programa.
- *clean*: Objetivo que limpia de archivos temporales el proyecto. Elimina la carpeta donde se almacenan los ficheros objeto, y ejecuta también los objetivos *clean* de cada biblioteca externa.

La última parte de este ejemplo se encarga de comprobar que se cumplen las dependencias de los módulos a compilar, de generar un ejecutable *.elf* a partir de los módulos objeto del proyecto, y en último lugar, de crear el ejecutable definitivo *.dol* a partir del archivo *.elf*.

B.2.3. Ejecución del programa

Una vez generado nuestro programa, existen dos maneras de ejecutarlo en la videoconsola. La primera, ya descrita, es lanzarlo a través del objetivo *make run*, para lo cual necesitamos tener correctamente conectada la Nintendo Wii a la red local, y *wiiload* bien configurado con la IP de la consola.

La otra manera de hacerlo, que es la necesaria para poder distribuir el programa, es guardar el ejecutable en un directorio de la tarjeta SD de la consola, cuya ruta debe ser */apps/XXX/boot.dol*, donde XXX es el nombre unix de nuestra aplicación (importante que no contenga espacios). Nótese que tanto el nombre *boot.dol* como el hecho de que el directorio que lo contiene esté dentro del directorio */apps* es obligatorio.

Debido a las características de *Homebrew Channel*, podemos acompañar nuestro ejecutable con una imagen que servirá de ícono para la aplicación (debe tener formato PNG, llamarse *icon.png*, y tener un tamaño de 128 píxeles de ancho y 48 de alto), y un fichero XML con la información que deseemos aportar sobre el programa (debe ser un XML con un formato concreto, llamado *meta.xml*). Ambos ficheros, la imagen y el archivo de datos, deben ir en el mismo directorio que el ejecutable.

Una referencia sobre los nodos del fichero de datos *meta.xml* que acompaña a una aplicación se puede encontrar a continuación:

```

1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <app version="1">
3      <name>El nombre de la aplicacion</name>
4      <coder>Autor o autores del codigo de la aplicacion.</coder>
5      <version>La version de la aplicacion.</version>
6      <release_date>Dia de lanzamiento. Formato: AAAAMMDD</release_date>
7      <short_description>Comentario que se muestra en el menu principal. No
        se recomienda que sea mayor de 30 caracteres</short_description>
8      <long_description>Descripcion detallada del programa<
        long_description>
9  </app>
```

B.3. Consideraciones sobre programar con *LibWiiEsp*

A la hora de programar para una plataforma cerrada como es Nintendo Wii, hay que tener una serie de aspectos en cuenta, debido a la arquitectura concreta que tiene el hardware de la consola. Es muy importante tener en cuenta estos detalles, ya que en caso contrario se pueden producir errores o comportamientos inesperados, como texturas que se pisán entre sí en la memoria principal, o excepciones propias del sistema.

A continuación se describen todos esos pequeños detalles a tener en cuenta, que si bien sólo suponen un cambio en algunos hábitos a la hora de programar, nos

aseguran que todo irá bien (al menos, en lo que al funcionamiento del hardware se refiere).

B.3.1. BigEndian

Lo primero que hay que tener en cuenta es que la representación de los datos en la consola Nintendo Wii se realiza con *BigEndian*, al contrario que las plataformas Intel (la arquitectura, no la marca del procesador), que utilizan *LittleEndian* para almacenar la información. Cuando un dato ocupa más de un byte, se puede organizar de mayor a menor peso (esto sería *BigEndian*), o bien de menor a mayor peso (que es la organización en *LittleEndian*).

Un ejemplo muy claro es la representación del *número mágico* que emplean los archivos de imágenes formadas por mapas de bits (también conocidos como *bitmaps*). Este número es, en su representación decimal, 19778. Si esta cifra la representamos en sistema hexadecimal con *LittleEndian* el resultado sería 0x4D42; sin embargo, en un sistema que utilice *BigEndian*, esta cifra se representaría como el hexadecimal 0x424D en la memoria principal.

Así pues, como consecuencia de esto, siempre que queramos cargar un archivo binario (una imagen, una pista de música, etc.) en la consola Nintendo Wii, tenemos dos opciones: o bien modificamos la representación del archivo en la plataforma de origen (que normalmente será un ordenador con arquitectura Intel) para que el recurso esté ya representado como *BigEndian*, o podemos utilizar las funciones del espacio de nombres *Endian* que se proporcionan en el archivo de cabecera *util.h* de *LibWiiEsp*. Este espacio de nombre aporta las siguientes dos funciones, que se encargan de transformar variables de 16 o 32 bits entre *LittleEndian* y *BigEndian* (soportan ambas transformaciones):

```

1 u16 inline swap16(u16 a) {
2     return ((a<<8) | (a>>8));
3 }
4
5 u32 inline swap32(u32 a) {
6     return ((a)<<24 | (((a)<<8) & 0x00FF0000) | (((a)>>8) & 0x0000FF00) | (a)
7         >>24);
}
```

Así pues, únicamente utilizando estas dos funciones con cada dos o cuatro bytes cuando queramos cargar un recurso desde la tarjeta SD podremos evitar los problemas derivados de los distintos sistemas de representación. Para un ejemplo práctico sobre el uso de estas dos funciones, ver el código fuente de la clase Imagen de *LibWiiEsp*.

B.3.2. Los tipos de datos

A la hora de programar para Nintendo Wii se utilizan siempre estos tipos de datos:

Tipo de dato	Descripción	Rango
u8	Entero de 8 bits sin signo	0 a 255
s8	Entero de 8 bits con signo	-127 a 128
u16	Entero de 16 bits sin signo	rango 0 a 65535
s16	Entero de 16 bits con signo	-32768 a 32767
u32	Entero de 32 bits sin signo	0 a 0xffffffff
s32	Entero de 32 bits con signo	-0x80000000 a 0x7fffffff
u64	Entero de 64 bits sin signo	0 a 0xffffffffffffffffffff
s64	Entero de 64 bits con signo	-0x8000000000000000 a 0x7fffffffffffffff
f32	Flotante de 32 bits	-
f64	Flotante de 64 bits	-

Cuadro B.1: Tipos de datos que se deben emplear al programar para Wii

Todos los tipos de datos disponibles se encuentran en `/opt/devkitpro/libogc/include/gctypes.h`.

B.3.3. La alineación de los datos

Otro detalle importante que se debe contemplar a la hora de programar para Nintendo Wii es que el procesador de la consola necesita alinear los datos conforme a su tamaño. Es decir, si vamos a trabajar con un entero de 32 bits (4 bytes), sólo se pueden almacenar en posiciones de memoria múltiplos de cuatro: 0, 4, 8, 12... Lo mismo ocurre con variables de 16 o 64 bits.

Para ilustrar el comportamiento del procesador respecto a la alineación de los datos, podemos considerar el siguiente ejemplo:

```

1 struct ejemplo {
2     u8 entero;
3     u32 otro;
4 }
```

La estructura *ejemplo* no se representará igual compilando en una plataforma Intel que en la Wii, ya que en la primera, *entero* se situará en la posición 0 de la memoria, y *otro* ocupará de la posición 1 hasta la 5. Sin embargo, en nuestra consola *entero* ocupará el mismo lugar, la posición 0, pero la variable *otro* necesitará estar alineada a su tamaño, es decir, ocupará las posiciones 4 a 7 de la memoria, y las posiciones entre *entero* y *otro* se llenarán con ceros para asegurar que el entero de 32 bits está alineado.

Esto revierte tanto en la cantidad de memoria ocupada, como en el hecho de que si el procesador encontrara un dato desalineado e intentara leerlo, se produciría un error en el sistema.

Para solventar este inconveniente, se debe jugar inteligentemente con las declaraciones de las variables, de tal manera que se organicen los datos de forma alineada.

B.3.4. Alineación de datos que usan DMA

Este es otro caso en el que influye la necesidad de alinear los datos con los que se trabaja. El procesador de la Nintendo Wii trabaja con datos cacheados, pero no así los periféricos (la tarjeta SD, los dispositivos USB o el lector DVD). Además, estos periféricos trabajan con una alineación fija de 32 bytes, y si no se contempla este detalle, se pueden producir errores de *machacamiento* de datos al realizar dos lecturas consecutivas desde periféricos.

Además, cuando leemos un dato desde un periférico, se corre el riesgo de machacar el contenido de la caché del procesador, por lo que es imprescindible volcar explícitamente los datos leídos en la memoria, es decir, asegurar de que la lectura se ha realizado completamente antes de realizar cualquier otra acción.

Es muy sencillo evitar esta situación, y es preparando la memoria en la que se almacenarán los datos leídos desde el periférico, de tal manera que esté alineada a 32 bytes y su tamaño sea múltiplo de 32. Para ilustrar cómo hacer esto, se muestra un ejemplo a continuación:

```

1 // Abrimos el archivo mediante un flujo
2 ifstream archivo;
3 archivo.open("SD:/apps/wiipang/media/sonido.pcm", ios::binary);
4
5 // Obtener el tamaño del sonido
6 archivo.seekg(0, ios::end);
7 u16 size = archivo.tellg();
8 archivo.seekg(0, ios::beg);
9
10 // Calcular el relleno que hay que aplicar a la memoria reservada
11 // para que su tamaño sea múltiplo de 32
12 u8 relleno = (size * sizeof(s16)) % 32;
13
14 // Reservar memoria alineada: utilizamos memalign en lugar de malloc
15 s16* sonido = (s16*)memalign(32, size * sizeof(s16) + relleno);
16
17 // Realizar la lectura de datos desde el periférico
18 // Utilizar char* como tipo de lectura es por compatibilidad
19 // con libFat
20 archivo.read((char*)sonido, size);
21
22 // Fijar los datos leídos en la memoria, evita machacamiento
23 // en la cache
24 DCFlushRange(sonido, size * sizeof(s16) + relleno);
```

Como puede verse en el ejemplo, se calcula en primer lugar el relleno necesario para que la memoria que ocupa el archivo binario a cargar sea múltiplo de 32 bytes, a continuación se reserva la memoria alineada a 32 bytes utilizando la función *memalign* en lugar de *malloc*, y el último paso, tras leer la información desde el archivo, consiste en realizar el volcado explícito de información desde la caché de lectura a la memoria, utilizando la función *DCFlushRange* (esta función recibe la dirección de

memoria a la que se quiere realizar el volcado, y el tamaño de ésta).

B.3.5. Depuración con *LibWiiEsp*

Con toda la información anterior descrita en este manual y la referencia completa de la biblioteca, se puede comenzar a desarrollar un videojuego sencillo. Como en todo proceso de desarrollo de software, ocurrirán errores, y aquí es donde se hace patente la falta de medios disponibles para depurar el código.

Existen dos tipos de errores que podremos encontrarnos a la hora de programar para Nintendo Wii, que corresponden con las fases de compilación y ejecución. A continuación, vamos a plantear cómo solucionar los posibles errores que pueden surgir en cada uno de estos momentos:

Errores de compilación

En la fase de compilación suelen ocurrir, sobretodo, errores de sintaxis, aunque también es posible que ocurran errores de enlazado si las rutas hasta los archivos de cabeceras o las bibliotecas de enlace estático no son correctas. El compilador nos avisará de cualquier tipo de error que ocurra, tanto en el preprocesado, como en la compilación propiamente dicha y en el enlazado. Así pues, prestando atención a los mensajes que pueda proporcionarnos el compilador sobre los errores o avisos que se den, podremos depurar el código fuente de nuestra aplicación.

Sobre los mensajes de enlazado, el enlazador nos proporcionará información suficiente para saber qué ha fallado durante esta operación, pero si se sigue al pie de la letra este manual (especialmente, la instalación del entorno y la creación del *makefile*) no debería haber ningún problema.

Errores de ejecución

En la fase de ejecución es cuando tenemos verdaderos problemas para depurar nuestra aplicación, y es que apenas hay herramientas que puedan facilitarnos el conocer el estado de los objetos del sistema, el contenido de las variables, etc.

LibWiiEsp proporciona un sistema de registro de eventos del sistema, la clase *Logger*, que permite que escribamos un *log* de errores, avisos e información variada. La forma de trabajar con esta clase es muy sencilla, y todos los detalles pueden consultarse en su documentación (ver referencia completa de la biblioteca). Pero hay ocasiones en que los errores en tiempo de ejecución requieren más precisión que una serie de mensajes que aportemos desde nuestro propio código, ya que el uso de la clase *Logger* está recomendado en casos de comportamiento inesperado del programa, pero no de errores como tal.

Cuando ocurre un error de ejecución en la Nintendo Wii, ésta nos presentará una pantalla de error parecida a la imagen de la figura B.1. En esta pantalla de

```

Exception (DSI) occurred!
GPR00 SEC0D82A6 GPR08 804FBE98 GPR16 0000151E GPR24 00000001
GPR01 801B2398 GPR09 803BBB40 GPR17 800936E8 GPR25 00000037
GPR02 8008AA68 GPR10 80095040 GPR18 FFFFFFFF GPR26 803BC7A8
GPR03 80181AE8 GPR11 DF093DE6 GPR19 8007F028 GPR27 803BC950
GPR04 803BBB48 GPR12 88200028 GPR28 8018A860 GPR29 0000000C
GPR05 803BBC38 GPR13 80097A20 GPR21 8007F03B GPR29 0000000D
GPR06 00000000 GPR14 0000151E GPR22 0000000D GPR30 803BBB48
GPR07 SEC0D82A6 GPR15 00000037 GPR23 801B23E4 GPR31 80192000
LR 8005a158 SRR0 8005a178 SRR1 0000a032 MSR 00000000
DAR DF093DEA DSISR 04000000

STACK DUMP:
8005a178 --> 8005a158 --> 80011cc0 --> 80011f54 -->
80012d84 --> 8000d40c --> 800043cc --> 80033374 -->
80033314

CODE DUMP:
8005a178: 810B0004 5506003A 419E0174 70E60001
8005a188: 910B0004 38E90000 40820034 80FEFFFF8
8005a198: 38AA0008 7D274850 70003A14 80C90008

```

Figura B.1: Ejemplo de pantalla de error en tiempo de ejecución

error pueden apreciarse tres partes principales. A la hora de localizar en qué parte de nuestro código se ha provocado este error, necesitamos fijarnos en la sección *STACK DUMP*, es decir, la segunda. Esta parte del mensaje de error nos detalla la traza de llamadas a función que se han realizado hasta llegar a la llamada que ha producido el error, estando cada elemento de la traza representado por una dirección de memoria en hexadecimal.

Esta información es muy útil, ya que podemos localizar a qué línea de nuestro código fuente corresponde cada llamada con una utilidad incluida en *DevKitPPC*, y esta es la utilidad *powerpc-eabi-addr2line*. Se puede localizar en la carpeta */opt/-devkitpro/devkitPPC/bin* si se ha seguido este manual para instalar *LibWiiEsp*. Lo más cómodo es crear un enlace simbólico a esta utilidad en el directorio principal de nuestro proyecto, aunque también se puede añadir el directorio *devkitPPC/bin* a la ruta donde el sistema busca ejecutables.

Siguiendo el ejemplo de la figura 1, si queremos saber a qué archivo y qué línea de código pertenece la dirección *0x80011f54*, basta con ejecutar la siguiente línea de código en el directorio principal del proyecto:

```
./powerpc-eabi-addr2line -e boot.elf -f 0x80011f54
```

Para que esta utilidad funcione, debemos tener en el directorio donde la ejecutamos una copia del ejecutable *.elf* generado por nuestro proyecto (el mismo que hemos ejecutado en la consola y que nos ha dado el mensaje de error de la Figura 1). El parámetro *-e* recibe el nombre de este ejecutable *.elf*, y el parámetro *-f* es la dirección de memoria (en hexadecimal) que ha producido el error.

Un ejemplo del resultado de la ejecución de la utilidad *addr2line* puede ser el

siguiente:

```
Actor  
/home/rabbit/Escritorio/libwiiesp/src/actor.cpp:27
```

Lo cual nos indica que el error se ha producido en la línea 27 del fichero fuente *actor.cpp* de *LibWiiEsp*.

Un detalle a comentar es que, para salir de la pantalla de error que se muestra en la Figura 1, basta con pulsar el botón *Reset* de la consola, lo que nos devolverá al *Homebrew Channel*.

En resumen, con la herramienta *addr2line* y la clase *Logger* podemos ir realizando una decente depuración de nuestra aplicación, que aunque hay que reconocer que no es muy cómodo, es mejor que ir dando palos de ciego.

B.4. Plantillas de *LibWiiEsp*

Con todo lo visto hasta el momento en este manual es posible comenzar el desarrollo de nuestra aplicación, ya que tenemos las herramientas preparadas, sabemos qué hay que tener en cuenta a la hora de utilizarlas, y además contamos con las clases de *LibWiiEsp*, que nos facilitan (mucho) la vida a la hora de cargar recursos multimedia, establecer los idiomas del módulo de internacionalización, acceder a la tarjeta SD de la consola, dibujar texturas y animaciones en pantalla, etc.

Pero el punto más interesante de *LibWiiEsp*, además de la abstracción que nos proporciona a la hora de trabajar con estos subsistemas de la videoconsola, está formado por las tres plantillas (clases abstractas) que ofrece para facilitar el desarrollo de un videojuego. Estas plantillas, que son *Actor*, *Nivel* y *Juego*, permiten crear los distintos tipos de actores y niveles, además de la clase principal de nuestro programa, de una manera sencilla y efectiva.

Cabe destacar que, en el caso de que las plantillas ofrecidas no se adapten a las necesidades del videojuego que tenemos en mente, siempre podemos personalizar la clase que corresponda al derivarla, o bien modificando la clase original desde el código fuente de la biblioteca.

En esta sección del manual vamos a desgranar los detalles necesarios para poder sacar el máximo jugo a estas tres plantillas que nos facilitarán el desarrollo de nuestro videojuego:

B.4.1. Sistemas de coordenadas

En primer lugar, hay que comprender cómo funcionan los distintos sistemas de coordenadas que nos encontraremos a la hora de crear el universo de nuestro videojuego. La figura B.2 ilustra los tres sistemas que pueden darse a la vez en el juego.

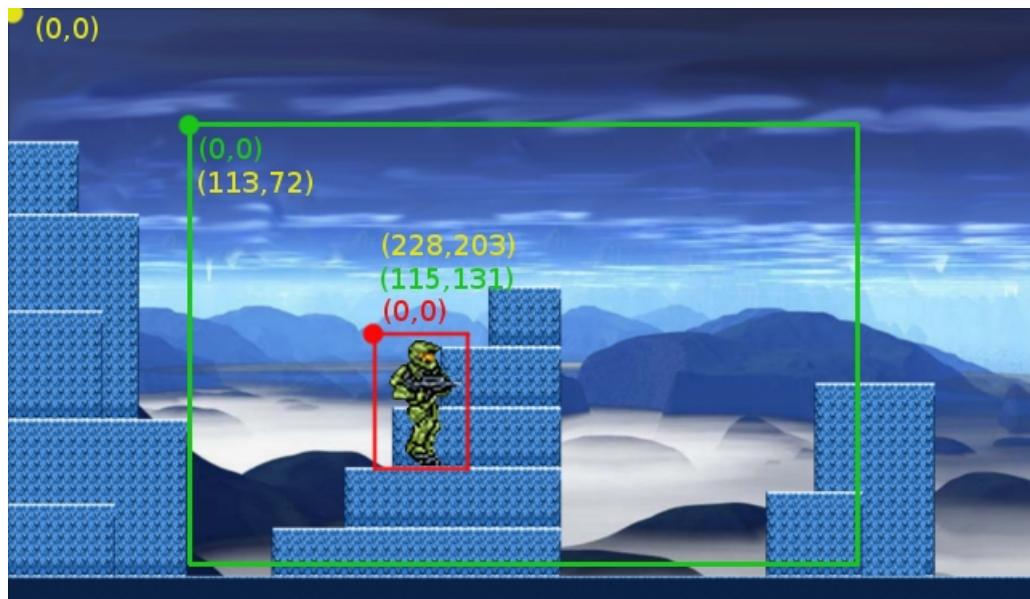


Figura B.2: Distintos sistemas de coordenadas en el universo del juego

La imagen al completo representa un escenario completo de un juego. Este escenario es el universo del juego en sí, tiene forma rectangular, y su esquina superior izquierda es el origen de coordenadas, siendo la X positiva hacia la derecha, y la Y hacia abajo. Todos los pares de coordenadas, en cualquiera de los tres sistemas que se emplean, se refieren al punto superior izquierdo del objeto al que pertenecen las coordenadas. En la imagen, se distinguen las coordenadas relativas al escenario con el color amarillo.

El rectángulo verde que se aprecia en el centro de la imagen representa la parte del escenario que se muestra en la pantalla (actúa como si fuera una ventana desplazable). La pareja de coordenadas (x,y) en amarillo de este rectángulo indica la posición de desplazamiento de la pantalla, o *scroll*, respecto al punto (0,0) del escenario. Jugando con el *scroll* conseguimos que la pantalla *se mueva* sobre el escenario, y muestre la parte que queramos de éste. Por otro lado, el vértice superior izquierdo de la pantalla es el origen de otro sistema de coordenadas, que indica la posición de un objeto (un actor, por ejemplo) dentro de la visualización en pantalla.

Teniendo en cuenta estos dos sistemas de coordenadas, se entiende fácilmente que el actor (el rectángulo rojo) tenga una pareja de coordenadas que indiquen su posición en el escenario (coordenadas en amarillo), y otra pareja de coordenadas (en verde) que denotan su posición en la pantalla. Sin embargo, las coordenadas de un actor respecto a la pantalla no se almacenan, si no que se calculan restando su posición en el escenario (amarillo) menos la posición del desplazamiento de la pantalla (coordenadas amarillas de la pantalla).

Además, el vértice superior izquierdo del rectángulo que representa al actor es el origen de un tercer sistema de coordenadas, que se utiliza como referencia para las cajas de colisión asociadas al actor.

B.4.2. Actores

Un actor es un objeto que tiene entidad propia dentro del universo del videojuego. En este sentido, son actores tanto los protagonistas manejados por los jugadores, como los enemigos controlados por la máquina, los *items* que recogemos, cada una de las balas (en el caso de un juego de disparos) también es un actor...

Entrando en el apartado técnico, un actor se representa como un objeto que tiene una pareja de coordenadas (x,y) respecto al origen del escenario, una velocidad en píxeles por fotograma (tanto vertical como horizontal), un conjunto de estados, cada uno de los cuales tiene asociada una animación y varias figuras de colisión, y un indicador sobre qué estado de los posibles es el actual.

A continuación se explican los diversos aspectos a tener en cuenta a la hora de crear un actor utilizando la clase abstracta que proporciona *LibWiiEsp*. En primer lugar, tenemos que crear una clase derivada de *Actor*. En el constructor de nuestra clase derivada, no debemos olvidar pasarle al constructor de *Actor* la cadena de caracteres con la ruta absoluta hasta el archivo de datos desde el que se cargan los datos del actor, y una referencia al nivel en el que participará. Además, tenemos que definir el método *actualizar*, que es un método virtual puro, y en el cual tenemos que definir el comportamiento del actor dependiendo de su estado actual.

Ambos aspectos se explican con detalle en los siguientes apartados:

Cargando los datos de un actor

Cada actor que se cree derivando la clase *Actor* cargará toda la información relativa a él a través del método *cargarDatosIniciales*, definido en la propia clase base *Actor* (al cual se llama desde el constructor de ésta clase), y que recibe la ruta absoluta, en la tarjeta SD, de un archivo XML con un formato como el siguiente:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <actor vx="3" vy="3" tipo="jugador">
3      <animaciones>
4          <animacion estado="normal" img="chief" sec="0" filas="1" columnas="6" retardo="3" />
5          <animacion estado="mover" img="chief" sec="0,1,2,3,4" filas="1" columnas="6" retardo="3" />
6          <animacion estado="muerte" img="chief" sec="5" filas="1" columnas="6" retardo="0" />
7      </animaciones>
8      <colisiones>
```

```

9   <rectangulo estado="normal" x1="27" y1="21" x2="55" y2="21" x3="55"
10  y3="96" x4="27" y4="96" />
11  <circulo estado="normal" cx="41" cy="13" radio="8" />
12  <rectangulo estado="mover" x1="27" y1="21" x2="55" y2="21" x3="55"
13  y3="96" x4="27" y4="96" />
14  <circulo estado="mover" cx="41" cy="13" radio="8" />
15  <sinfiria estado="muerte" />
</colisiones>
</actor>
```

En el archivo XML anterior pueden observarse dos grandes bloques, uno para las animaciones y otro para las figuras de colisión. A cada estado (en el ejemplo hay tres, *normal*, *mover* y *muerte*) le corresponde una única animación, pero puede tener ninguna, una o varias figuras de colisión. En el caso de que un estado concreto no tenga ninguna figura de colisión asociada, basta con introducir un nodo con el nombre *sinfiria* y su correspondiente atributo *estado*. Para más información sobre las animaciones o las figuras de colisión, consultar las secciones correspondientes en la referencia de la biblioteca.

Muy importante: Los estados en los cuales puede encontrarse un actor vienen definidos por los que aparezcan en este archivo de datos, y es **imprescindible** que definamos el estado *normal* (al menos, en las animaciones), ya que es el estado que un actor toma por defecto, y si no se encontrara entre los datos del actor, se produciría un error en el sistema.

Para más información sobre la carga de datos iniciales, consultar la documentación de la clase Actor.

Definir el comportamiento de un actor

El comportamiento de un actor, como ya se ha comentado, depende del estado en el que se encuentre. A la hora de crear una clase derivada de *Actor* se deberían implementar tantos métodos como estados se hayan definido en el archivo de datos del actor, de tal manera que cada uno de estos métodos corresponda con el comportamiento esperado en cada uno de los estados.

Por ejemplo, si tenemos un actor con tres estados (*normal*, *caer* y *mover*), tendríamos tres nuevos métodos llamados *estado_normal*, *estado_caer* y *estado_mover*. En cada una de estas funciones habría que implementar el comportamiento deseado de nuestro actor para ese estado concreto. En el siguiente código se muestra un ejemplo del método *estado_mover*, que se encargaría de desplazar horizontalmente al actor:

```

1 void estado_mover(void) {
2     mover(_x + _vx, _y);
3 }
```

Implementando de esta manera un método por cada estado, únicamente habría que definir el método virtual puro *actualizar* para que, según el estado actual del

actor, se ejecute la función correspondiente.

Lo ideal es organizar el comportamiento del actor en un autómata finito determinado, donde se especifiquen los estados posibles, y las transiciones que pueden darse entre los distintos estados. Hay que mencionar que los cambios de estado se realizarán desde una clase derivada de *Nivel*, que será el escenario donde los actores se encontrarán. El motivo de esta decisión no es otro que la falta de conocimiento que tiene un actor sobre lo que ocurre a su alrededor en el escenario del juego, información que sí está disponible en todo momento en la clase que se encarga de gestionar éste.

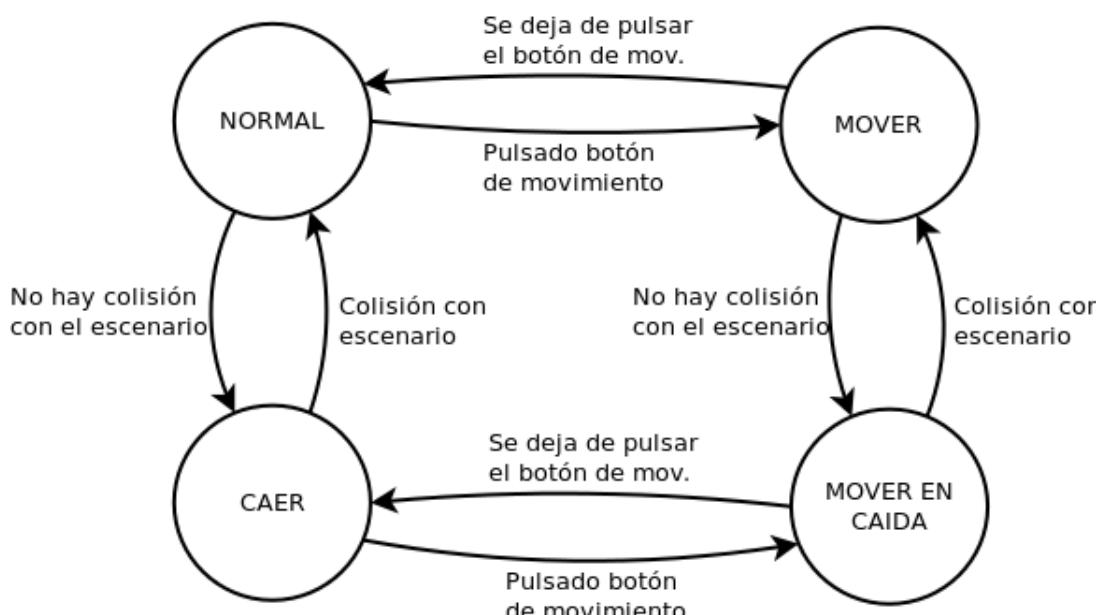


Figura B.3: Sencillo autómata de ejemplo para el comportamiento de un actor

En la figura B.3 puede apreciarse un sencillo ejemplo de autómata finito determinado, formado por cuatro estados, y que define el comportamiento de un actor controlado por un jugador a través de un mando. El actor comienza en el estado denominado *NORMAL*, en el que no sufre ninguna modificación de sus variables internas de posición. Desde este estado, y dependiendo de las condiciones que se cumplan, se puede pasar a los estados *MOVER* (si el jugador pulsa el botón de movimiento) o *CAER* (si no existe colisión entre el actor y el escenario). En el primero, nuestro actor modifica su posición horizontal en base a su velocidad en este eje, y en el segundo, se cambia la posición vertical hacia abajo, en base también a la velocidad del actor respecto al eje vertical. Desde estos dos estados puede llegarse a *MOVER EN CAIDA*, que es una combinación de ambos (movimiento en ambos ejes).

Puede comprobarse que, en cada estado, se proporciona un comportamiento para el actor y una serie de condiciones para que sucedan cambios de estado del actor. La manera de comportarse del actor para cada estado debe programarse en el mé-

todo *actualizar* de la clase derivada de *Actor* que controla a éste; sin embargo, las comprobaciones sobre el cumplimiento de condiciones que deben satisfacerse para ejecutar un cambio desde un estado a otro puede realizarse en el método correspondiente de la clase que controle el escenario (*actualizarPj* para los actores jugadores, o *actualizarNpj* en el caso de los actores controlados por la máquina), o bien en el propio método de actualización del actor, ya que éste tiene un atributo que es una referencia al nivel en el que participa. Se recomienda optar por la primera opción, para así separar el comportamiento del actor según el estado y las transiciones posibles entre éstos.

B.4.3. Niveles

Un nivel representa el escenario donde los actores, ya estén controlados por un jugador o por la máquina, interactúan entre sí y con los componentes de dicho nivel. Al igual que ocurre con los actores, *LibWiiEsp* proporciona una clase base para crear niveles de una manera sencilla y rápida, que permite diseñar cada nuevo escenario utilizando el software *Tiled*, editor de mapas de *tiles*.

Hay que distinguir entre los conceptos de escenario y nivel. Para *LibWiiEsp*, un nivel es una clase derivada de la clase abstracta *Nivel*, y que define varios escenarios que se comportan de la misma forma, siendo cada escenario un mapa de lites generado con *Tiled* en el que se especifica la disposición de los *tiles* y los actores que participan.

Con esto se consiguen varias ventajas. En primer lugar, definiendo una sola vez el comportamiento de un tipo de escenario en una clase derivada de *Nivel*, se pueden generar múltiples escenarios cuya lógica sea la implementada en esta clase. Por otro lado, este sistema permite que, en un mismo videojuego, se puedan intercalar fácilmente escenarios con comportamientos distintos (como ejemplo, se puede pensar en las típicas fases de *bonus* de clásicos como *Street Fighter*, en las que se debe destruir un coche o varios barriles en lugar de luchar contra otro oponente controlado por la máquina).

Partes de un nivel

Un nivel se compone de tres partes, que son los distintos tipos de objetos que se cargan en un escenario. En primer lugar, se encuentran las *propiedades* del nivel, que son una serie de cadenas de caracteres que indican códigos de recursos en la galería de medias del sistema; las propiedades del nivel son la imagen de fondo (que es fija, y permite dibujar un paisaje estático en la última capa de dibujo), la pista de música asociada a un nivel y la imagen del *tileset*. Pueden añadirse más propiedades según el videojuego que estemos desarrollando, pero la lectura y carga de esta información deberá ser programada en el constructor de la clase derivada. Las partes básicas del escenario se pueden apreciar en la figura B.4.

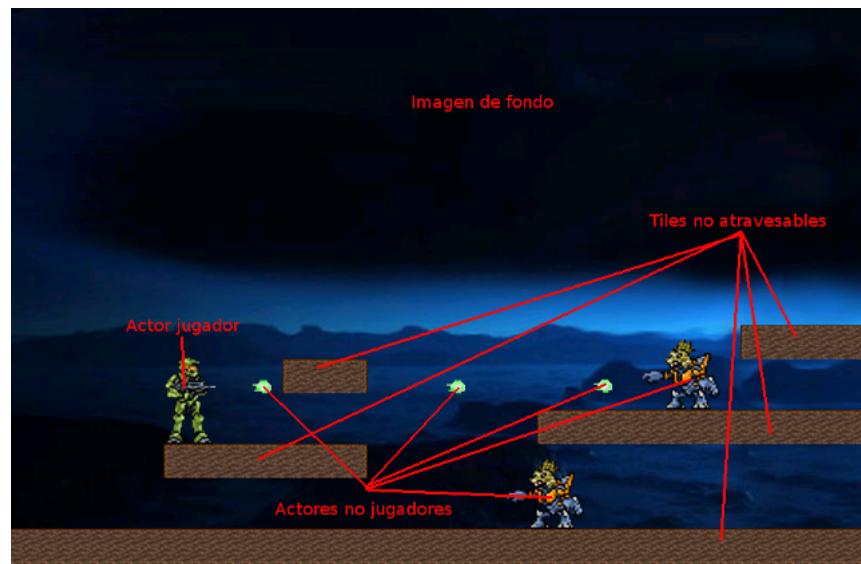


Figura B.4: Distintas partes de un escenario

Llegados a este punto, es necesario explicar brevemente los conceptos de *tile* y *tileset*. Un *tile* es una imagen pequeña, generalmente cuadrada o rectangular, que se utiliza para componer un escenario en un videojuego en dos dimensiones. En un mismo escenario, se emplean numerosos *tiles* que se recogen en una única imagen organizados como una tabla (es decir, en filas y columnas). Esta imagen que almacena todos los *tiles* utilizados en un mismo escenario es lo que conocemos por *tileset*. En la figura B.5 hay un ejemplo de un *tileset* con el que se podría dibujar un escenario para un juego de vista cenital.

Figura B.5: Ejemplo de *tileset*, formado por 6 *tiles* de 32x32 píxeles

Continuando con los tipos de objeto que componen un escenario, en segundo lugar tenemos la propia composición de éste, que está formada por dos capas de dibujo en las que se define la composición del escenario en sí a partir de los *tiles* del *tileset* asociado al nivel. Internamente, cada capa de dibujo se divide en una rejilla de cuadros (filas y columnas), donde cada celda tiene las mismas dimensiones que un *tile*, y en ella se coloca un *tile* concreto. Una de las dos capas del escenario, la llamada *PLATAFORMAS*, se caracteriza en que, cuando *LibWiiEsp* la carga en memoria, asigna a cada *tile* una figura de colisión de su mismo tamaño, de tal manera que los *tiles* que componen esta capa de dibujo pueden interactuar con los actores del juego. Por otro lado, la capa denominada *ESCENARIO* está compuesta por *tiles*

que no tienen asociada ninguna figura de colisión, y únicamente se añaden al nivel con el objetivo de mejorar visualmente el escenario.

Generalmente, esta distinción entre *tiles* con figura de colisión asociada y sin ella se utiliza para definir los *tiles* que pueden ser atravesados por los actores del juego (aquellos que no tienen asociada una figura de colisión), y los que no permiten que los actores los atraviesen, que son los que sí tienen figura de colisión asociada. Puede verse un ejemplo en la figura B.6.

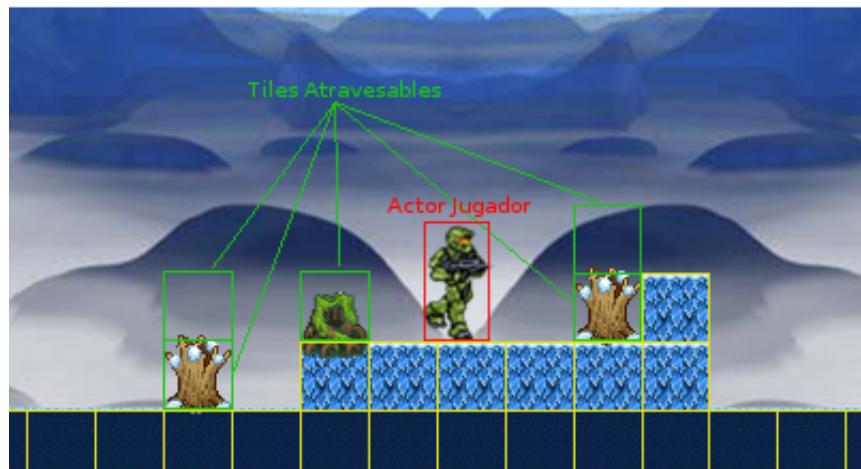


Figura B.6: Ejemplo de escenario con *tiles* atravesables y no atravesables

Por último, nos encontramos con el último tipo de objetos que necesitamos para definir un escenario, que son los actores que participan en el nivel. Se distinguen los actores controlados por los jugadores, y los que son dirigidos por la máquina.

Como ya se ha comentado en la sección de los actores, éstos tendrán definido su comportamiento según el estado en el que se encuentren en un momento determinado. Sin embargo, las transiciones entre distintos estados es conveniente realizarlas en el correspondiente método de actualización de la clase que gestiona el escenario, a pesar de que se pueden hacer desde la clase del actor.

Creación de un escenario con *Tiled*

El proceso de creación de un escenario a partir de la herramienta *Tiled* [11] es muy sencillo. En su página oficial podemos descargar la última versión o, dependiendo de si el sistema en el que nos encontramos (en mi caso al redactar este manual, una Ubuntu 10.10) dispone de esta utilidad en los repositorios, se puede instalar con la siguiente orden:

```
sudo apt-get install tiled
```

Una vez instalado, ejecutamos *Tiled* y pulsamos en el botón *Nuevo*. Se abrirá una ventana en la que se nos preguntan algunos parámetros del nuevo mapa de

tiles. Seleccionamos proyección ortogonal, el ancho y alto en píxeles de cada *tile*, y el ancho y alto medido en número de *tiles* que tendrá el escenario. Las medidas de un *tile* deben ser múltiplos de 8 píxeles, debido a los requisitos para la carga de texturas con *LibWiiEsp*, siendo un tamaño recomendable es 32x32 píxeles por *tile*. La resolución que utiliza la Nintendo Wii en un sistema PAL y proporción 4:3 es de 640 píxeles de ancho por 528 de alto, por lo que el tamaño mínimo del escenario debería ser (si utilizamos la medida 32x32 píxeles por *tile*) de 20 *tiles* de ancho, por 16 *tiles* de alto. Una vez establecidos estos parámetros, pulsamos en aceptar.

A continuación, hay que cargar en la herramienta la imagen que contiene el *tile-set*. Para ello, pulsamos en *Mapa*, y después en *Nuevo conjunto de patrones*. Se abrirá un diálogo en el que debemos introducir el nombre que le daremos a la imagen (esto es un dato interno de *Tiled*, y el valor introducido aquí no es relevante), la propia imagen, y las medidas que tendrá un *tile* (ancho y alto en píxeles). Un detalle a tener en cuenta, en nuestra imagen no debe haber separación entre los *tiles*, ni tampoco margen. Al introducir estos datos, hacemos clic en *Aceptar* y ya tendremos el *tileset* listo para dibujar el mapa.

Como siguiente punto, hay que introducir las tres propiedades del escenario. Nos vamos a *Mapa* y entonces a *Propiedades del Mapa*, e introducimos en la lista estas propiedades:

- *imagen_fondo*: Su valor es el código que tendrá la imagen de fondo del escenario en la galería de recursos del sistema.
- *imagen_tileset*: Su valor es el código que tendrá la imagen del *tileset* del escenario en la galería de recursos del sistema.
- *musica*: Su valor es el código que tendrá la pista de música del escenario en la galería de recursos del sistema.

Tras introducir las propiedades, pulsamos en *Aceptar*, y procedemos a preparar las tres capas de dibujo de nuestro escenario, que deben llamarse obligatoriamente como se describe en la lista:

- *escenario*: Capa de patrones donde dibujaremos los *tiles* atravesables.
- *plataformas*: Capa de patrones donde dibujaremos los *tiles* no atravesables, es decir, los que tendrán asociada una figura de colisión.
- *actores*: Capa de objetos donde situaremos todos los actores que participen en el nivel.

En este punto, ya estamos en disposición de comenzar a dibujar nuestro escenario. Es recomendable comenzar por la capa de *tiles* no atravesables, continuar decorando con los *tiles* atravesables, y por último, establecer la posición de los actores. Para situar un actor en el escenario, insertamos un objeto en el lugar que deseemos, y lo redimensionamos adecuadamente para que se vea correctamente cómo quedaría

su posición inicial. Seguidamente, hacemos clic con el botón derecho en el actor, y pulsamos en *Propiedades del objeto*. Se mostrará un diálogo en el que debemos indicar, en el campo *Tipo*, el tipo del actor que comenzará aquí (dato necesario a la hora de distinguir qué clase derivada de *Actor* hay que instanciar), y además la propiedad *xml*, cuyo valor debe ser la ruta absoluta hasta el archivo XML que contiene la información de este tipo de actor en la tarjeta SD de la Nintendo Wii. Además, si el actor es controlado por un jugador, debemos añadir también la propiedad *jugador* con el código identificativo del jugador que jugará con este actor como valor.

A la hora de definir los distintos actores, se puede utilizar el campo *Nombre* del diálogo de propiedades de un actor para identificarlo en tiempo de diseño del escenario.

Implementando una clase que controle escenarios

Una vez tenemos creados uno o varios escenarios cuyo comportamiento (la gestión de transiciones entre los estados de los distintos actores que participan en él, el movimiento o no del *scroll* de la pantalla sobre el escenario, etc.) es común, el último paso para poder disfrutar de ellos es definir una clase derivada de la abstracta *Nivel* que controle todos los detalles del nivel. Cada clase derivada definirá una gestión distinta de un grupo de varios escenarios.

El primer paso para crear una clase derivada de la abstracta *Nivel* es implementar el método virtual *cargarActores*. Como se detalla en la documentación de la clase, en el constructor se cargan todos los *tiles* del escenario, se toma la imagen de fondo y del *tileset* desde la galería, y se leen los datos de inicialización de cada actor participante, almacenándose en una estructura temporal. La definición de este método debe recorrer esta estructura temporal de datos de actores, creando, para cada ocurrencia, un actor de la clase derivada de *Actor* correspondiente, tal y como se aprecia en el siguiente ejemplo:

```

1 void cargarActores(void) {
2     for(Temporal::iterator i = _temporal.begin(); i != _temporal.end();
3         ++i)
4     {
5         if(i->tipo_actor == "jugador") {
6             Personaje* p = new Personaje(i->xml, this);
7             p->mover(i->x, i->y);
8             _jugadores.insert(std::make_pair(i->jugador, p));
9         } else if(i->tipo_actor == "bicho") {
10            Bicho* p = new Bicho(i->xml, this);
11            p->mover(i->x, i->y);
12            _actores.push_back(p);
13        }
14     }
15 }
```

En el ejemplo, el programador ha definido dos clases derivadas de *Actor*, deno-

minadas *Personaje* y *Bicho*. En el método implementado, se recorre la estructura temporal de datos de actores del escenario, y se crea un actor a partir de la clase correspondiente (según el tipo de actor que se haya indicado desde *Tiled*), se mueve el actor hasta su posición en el escenario, y por último se inserta en la estructura adecuada (*_jugadores* en el caso de los actores controlados por un jugador, en la que hay que indicar también el código identificador del jugador concreto; o *_actores* para los actores controlados por la máquina).

Es importante recordar que este método *cargarActores* se debe llamar desde el constructor de la clase derivada de *Nivel* con la intención de que la creación de los actores se realice en el momento de cargar el escenario. Además, es recomendable vaciar la estructura temporal cuando se finalice el proceso.

El siguiente paso en la generación de esta clase derivada es implementar los métodos de actualización del nivel, que se deberían llamar en cada fotograma del programa. A continuación se indican cuáles son, y qué funcionalidad se espera que tengan.

En el método *actualizarPj* se debe actualizar el estado de un único actor jugador, atendiendo tanto al mando concreto que tenga asociado en la estructura *_jugadores*, como a la situación del escenario. Un ejemplo sencillo podría ser:

```

1 void actualizarPj(const std::string& jugador, const Mando& m) {
2
3     // Estado NORMAL: puede pasar a MOVER
4     if(_jugadores[jugador]->estado() == "normal") {
5         if(m.pressed(Mando::BOTON_ARRIBA) or m.pressed(Mando::BOTON_ABAJO) )
6             _jugadores[jugador]->setEstado("mover");
7     }
8
9     // Estado MOVER: puede pasar a NORMAL
10    if(_jugadores[jugador]->estado() == "mover") {
11        if(m.pressed(Mando::BOTON_ARRIBA)) {
12            _jugadores[jugador]->invertirDibujo(true);
13            s16 vel_x = _jugadores[jugador]->velX();
14            if(vel_x > 0)
15                vel_x *= -1;
16            _jugadores[jugador]->setVelX(vel_x);
17        } else if(m.pressed(Mando::BOTON_ABAJO)) {
18            _jugadores[jugador]->invertirDibujo(false);
19            s16 vel_x = _jugadores[jugador]->velX();
20            if(vel_x < 0)
21                vel_x *= -1;
22            _jugadores[jugador]->setVelX(vel_x);
23        } else
24            _jugadores[jugador]->setEstado("normal");
25    }
26
27    // Actualizar el actor en base a su nuevo estado actual
28    _jugadores[jugador]->actualizar();
29 }
```

Por otro lado, el método *actualizarNpj* debe recorrer la estructura en la que se almacenan los actores controlados por la máquina y actualizar los que se consideren oportunos (aquí se deja en manos del programador el actualizar todos los actores, sólo los que están en pantalla, o los que cumplan un determinado criterio). Como ejemplo, se muestra la siguiente función que actualizaría el estado de todos los actores no jugadores:

```

1 void actualizarNpj(void) {
2     for(Actores::iterator i = _actores.begin() ; i != _actores.end() ; ++
3         i) {
4         // Estado NORMAL: puede pasar a CAER
5         if ((*i)->estado() == "normal")
6             if(not colision((*i)))
7                 (*i)->setEstado("caer");
8
9         // Estado CAER: puede pasar a NORMAL
10        if ((*i)->estado() == "caer")
11            if(collisionVertical((*i)))
12                (*i)->setEstado("normal");
13
14        // Actualizacion del actor
15        (*i)->actualizar();
16    }
17 }
```

El último método a implementar es *actualizarEscenario*, en el que se espera que se implementen todos los demás detalles relativos al escenario que necesiten ser actualizados a cada fotograma del juego. El siguiente ejemplo muestra una implementación que únicamente actualiza el *scroll* de la pantalla sobre el escenario, según la posición horizontal del jugador cuyo código identificador es *pj1*:

```

1 void actualizarEscenario(void) {
2
3     if(_jugadores["pj1"]->x() - _scroll_x >= screen->ancho() / 2)
4         moverScroll(_scroll_x + abs(_jugadores["pj1"]->velX()), _scroll_y);
5     else if(_jugadores["pj1"]->x() - _scroll_x <= screen->ancho() / 4)
6         moverScroll(_scroll_x - abs(_jugadores["pj1"]->velX()), _scroll_y);
7
8     if(_jugadores["pj1"]->y() - _scroll_y >= screen->alto() / 2)
9         moverScroll(_scroll_x, _scroll_y + abs(_jugadores["pj1"]->velY()));
10    else if(_jugadores["pj1"]->y() - _scroll_y <= screen->alto() / 4)
11        moverScroll(_scroll_x, _scroll_y - abs(_jugadores["pj1"]->velY()));
12 }
```

Por supuesto, quiero remarcar que los métodos de ejemplo son precisamente eso, ejemplos muy sencillos cuya finalidad es que sirvan de guía para comprender cómo se trabaja con la plantilla de niveles de *LibWiiEsp*, y a partir de los cuales poder desarrollar los métodos de actualización necesarios (al derivar la clase abstracta *Nivel* se pueden añadir los métodos que se consideren necesarios).

B.4.4. Juego

La clase abstracta *Juego* es la tercera y última plantilla que *LibWiiEsp* ofrece para facilitar el desarrollo de videojuegos para Nintendo Wii. Es muy sencilla, y consiste en dos partes principales. El constructor se encarga de inicializar la consola a partir de la información que se introduzca en el archivo de configuración de la aplicación, y el método *run* ejecuta el bucle principal del programa. A continuación se aportan todos los detalles relativos a esta plantilla para construir la clase principal de nuestro videojuego.

Inicialización de la consola

Como ya se ha comentado, la inicialización de todos los sistemas de la consola Nintendo Wii se realiza en el constructor de la clase *Juego*, que recibe como parámetro la ruta absoluta en la tarjeta SD de un archivo XML de configuración. Esta inicialización consiste en montar la primera partición de la tarjeta SD de la consola (debe tener un sistema de ficheros FAT), establecer el sistema de *logging*, leer el archivo de configuración y, a partir de éste, iniciar todos los aspectos de la consola que vamos a utilizar.

El proceso de inicialización, llegados a este punto, es el siguiente:

1. Inicializar la pantalla, el sistema de mandos, el sonido y las fuentes de textos (en este orden).
2. Establecer el color transparente, y los fotogramas por segundo que tendrá la aplicación.
3. Cargar los identificadores de los jugadores y asociar un mando con cada uno de ellos.
4. Cargar en memoria todos los recursos multimedia que se indiquen en el archivo XML de la galería.
5. Cargar en memoria las etiquetas de texto del soporte de internacionalización.

Cuando creamos una clase derivada de *Juego* hay que llamar al constructor de la clase base, pasándole como parámetro la ruta absoluta en la tarjeta SD del archivo de configuración. Por otro lado, el destructor de la clase base se encarga de liberar la memoria ocupada por la estructura que almacena los objetos de la clase *Mando*, y de llamar a la función *exit*, hecho obligatorio para que la pila de la función *atexit* se ejecute al salir del programa (esto es muy importante, ya que en caso contrario nos encontraremos con una pantalla de error por no haber apagado los sistemas de la consola).

Un ejemplo del archivo XML de configuración es el siguiente:

```
1 | <?xml version="1.0" encoding="UTF-8"?>
2 | <conf>
```

```

3 | <log valor="/apps/wiipang/info.log" nivel="3" />
4 | <alpha valor="0xFF00FFFF" />
5 | <fps valor="25" />
6 | <galeria valor="/apps/wiipang/xml/galeria.xml" />
7 | <lang valor="/apps/wiipang/xml/lang.xml" defecto="english" />
8 | <jugadores pj1="pj1" pj2="pj2" pj3="" pj4="" />
9 | </conf>

```

En este archivo de configuración se establece que el sistema de *logging* registrará todos los eventos que sucedan en el sistema, el color transparente será el magenta, se correrá la aplicación a 25 fotogramas por segundo, se indican los archivos XML de la galería y el sistema de internacionalización, se establece el inglés como idioma por defecto, y se prepara la consola para trabajar con dos mandos, asociados respectivamente a un jugador identificado por el código *pj1* y otro identificado por *pj2*.

El bucle principal

La clase base *Juego* proporciona, además, un método que ejecuta un bucle principal sencillo. Este método es virtual, de tal manera que si el programador necesita otra forma de gestionar su aplicación, se le permite redefinirlo en su clase derivada.

El método *run* es el que se encarga de controlar este bucle principal. En primer lugar, llama al método *cargar*, que es virtual puro y debe ser definido en la clase derivada de *Juego*. En esta función debe ejecutarse todo lo que se necesite **antes** de que se entre en el bucle. Después, se inicializa la bandera de salida con un valor falso y se establece el contador de ciclos del procesador a cero (este contador se utiliza para mantener constante el valor de los fotogramas por segundo), tras lo cual se entra en el bucle principal.

El bucle principal actualiza, al principio de cada fotograma, el estado de todos los mandos conectados a la consola, ejecuta el método virtual puro *frame*, y después finaliza el fotograma y controla la tasa de FPS. En este método *frame* se incluirán todos los detalles de la ejecución de cada fotograma, y devolverá un valor booleano falso si la ejecución debe continuar, siendo el valor de retorno verdadero en el caso de que el programa deba terminar.

Un detalle más, en el caso de que ocurriera una excepción en el transcurso de la ejecución del programa, ésta será registrada por el sistema de *logging* (siempre que éste esté activado, al menos, en el nivel *error*, identificador 1), y después se saldrá de la aplicación.

Por último, destacar el hecho de que tanto si se necesita una gestión del bucle principal diferente, o un mayor número de funciones, el hecho de tener que derivar de la clase *Juego* implica la posibilidad de crear tantos métodos como sea necesario, y la redefinición opcional del método *run* nos permite ejecutar estos nuevos métodos de la manera que mejor se adecúe a nuestras necesidades.

Apéndice C

Manual de referencia

En este apéndice se incluye el manual de referencia completo de la biblioteca. Este manual aporta una visión en profundidad sobre todos los métodos, atributos y estructuras que componen la herramienta, y está generado con la herramienta Doxygen [4].

Por otro lado, cada una de las clases documentadas en este manual viene acompañada por una exhaustiva explicación sobre su funcionamiento interno, así como ejemplos sencillos que ilustran cómo se debe emplear cada módulo a la hora de darle uso dentro de un videojuego desarrollado con *LibWiiEsp*.

Nótese que, aunque se respetan los números de página dentro de este documento, la numeración de este apéndice difiere de la empleada durante el resto del escrito, debido a que este manual de referencia se ha importado directamente desde el archivo PDF generado con Doxygen.

Capítulo 1

LibWiiEsp

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

LibWiiEsp es una biblioteca libre de desarrollo de juegos en dos dimensiones para Nintendo Wii. Pretende ser una aproximación sencilla, pero efectiva y genérica, al desarrollo para la consola de Nintendo, proporcionando una serie de subsistemas útiles a la hora de desarrollar todo tipo de juegos.

LibWiiEsp está construida sobre el kit de desarrollo DevKitPro, y más concretamente su versión para PowerPC, denominada DevKitPPC (en su versión r21). DevKitPro ofrece una serie de compiladores, enlazadores, reglas de compilación, bibliotecas de C++ y útiles (wiiload addr2line, etc.) para compilar y ejecutar programas en una consola Nintendo Wii que disponga de HomeBrew Channel instalado (la versión del menú de la consola o los iOS de que disponga la consola no influyen en la ejecución del software generado con LibWiiEsp). Como base de desarrollo, se toma la biblioteca de bajo nivel libOgc (en su versión 1.8.4), junto con la adaptación al sistema de la consola de libfat (versión 1.0.7). Ambas herramientas están disponibles en la forja del proyecto.

LibWiiEsp está pensada para ser un conjunto de herramientas que faciliten la entrada del programador en el desarrollo para Nintendo Wii. La biblioteca, junto con su documentación completa, conforma un primer paso en la programación para la consola, pero esta primera versión no está pensada para ir mucho más allá que ser una introducción en el mundillo.

Un último apunte; esta biblioteca no hubiera sido posible sin la ayuda de Hermes (elotrolado.net), quien es uno de los mayores sceners de Wii, y que ha desarrollado (junto a otros muchos) la biblioteca de bajo nivel libOgc, además de publicar varios tutoriales de iniciación a la programación para Wii.

1.1. Licencia GPLv3

libWiiEsp es software libre: usted puede redistribuirlo y/o modificarlo bajo los términos de la Licencia Pública General GNU publicada por la Fundación para el Software Libre, ya sea la versión 3 de la Licencia, o (a su elección) cualquier versión posterior.

libWiiEsp se distribuye con la esperanza de que sea útil, pero SIN GARANTÍA ALGUNA; ni siquiera la garantía implícita MERCANTIL o de APTITUD PARA UN PROPÓSITO DETERMINADO. Consulte los detalles de la Licencia Pública General GNU para obtener una información más detallada.

1.2. Autoría

LibWiiEsp está desarrollada por Ezequiel Vázquez De la calle (ezequielvazq@gmail.com)

1.3. Detalles sobre instalación y uso

Consultar el manual de la biblioteca en la forja del proyecto.

1.4. Enlaces

Página del proyecto: <http://libwiiesp.forja.rediris.es/>

Forja del proyecto: <https://forja.rediris.es/projects/libwiiesp/>

Licencia GPLv3: <http://www.gnu.org/licenses/gpl.html>

Capítulo 2

Documentación de namespaces

2.1. Referencia del Namespace endian

Grupo de funciones para transformar variables desde big endian a little endian, y vice-versa.

Funciones

- u16 swap16 (u16 a)
- u32 swap32 (u32 a)

2.1.1. Descripción detallada

Grupo de funciones para transformar variables desde big endian a little endian, y vice-versa.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

El procesador de la Nintendo Wii trabaja con big endian como representación interna de los bytes, al contrario que las plataformas Intel, que utilizan little endian. Esto significa que, para poder leer información de un fichero importado desde una plataforma Intel, hay que invertir el orden de los bytes que se vayan leyendo. Las funciones que se presentan en este espacio de nombres sirven para invertir el orden de variables de 16 y 32 bytes, de tal manera que la Nintendo Wii pueda leer ficheros binarios importados desde un PC.

2.1.2. Documentación de las funciones

2.1.2.1. `u16 endian::swap16(u16 a) [inline]`

Cambia el endian de una variable de 16 bits

Parámetros

<code>a</code>	Variabile de 16 bits a la que se desea cambiar el endian
----------------	--

Devuelve

Valor de la variable con el endian cambiado

2.1.2.2. `u32 endian::swap32(u32 a) [inline]`

Cambia el endian de una variable de 32 bits

Parámetros

<code>a</code>	Variabile de 32 bits a la que se desea cambiar el endian
----------------	--

Devuelve

Valor de la variable con el endian cambiado

2.2. Referencia del Namespace pixel

Grupo de funciones para manejar los colores con el formato 0xRRGGBBAA.

Funciones

- `u32 color (u8 r, u8 g, u8 b)`
- `u8 rojo (u32 color)`
- `u8 verde (u32 color)`
- `u8 azul (u32 color)`

2.2.1. Descripción detallada

Grupo de funciones para manejar los colores con el formato 0xRRGGBBAA.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Gracias a este grupo de funciones, resulta muy cómodo trabajar con los colores de los píxeles, ya que permiten tanto formar un color a partir de sus tres componentes, como obtener un componente (rojo, verde o azul) a partir de un color completo.

2.2.2. Documentación de las funciones

2.2.2.1. `u8 pixel::azul (u32 color) [inline]`

Función para obtener el componente azul de un color en formato 0xRRGGBBAA

Parámetros

<code>color</code>	Color del cual se quiere obtener el componente azul
--------------------	---

Devuelve

Componente azul del color que se recibe

2.2.2.2. `u32 pixel::color (u8 r, u8 g, u8 b) [inline]`

Función para formar un color a partir de sus tres componentes

Parámetros

<code>r</code>	Componente rojo del color que se quiere formar
<code>g</code>	Componente verde del color que se quiere formar
<code>b</code>	Componente azul del color que se quiere formar

Devuelve

Color en formato 0xRRGGBBAA, con el componente alpha visible al 100%

2.2.2.3. `u8 pixel::rojo (u32 color) [inline]`

Función para obtener el componente rojo de un color en formato 0xRRGGBBAA

Parámetros

<code>color</code>	Color del cual se quiere obtener el componente rojo
--------------------	---

Devuelve

Componente rojo del color que se recibe

2.2.2.4. `u8 pixel::verde (u32 color) [inline]`

Función para obtener el componente verde de un color en formato 0xRRGGBBAA

Parámetros

<code>color</code>	Color del cual se quiere obtener el componente verde
--------------------	--

Devuelve

Componente verde del color que se recibe

2.3. Referencia del Namespace tiempo

Grupo de funciones para controlar el tiempo en una aplicación.

Funciones

- `void controlarFps (const u8 fps)`

Variables

- `static u32 tick`

2.3.1. Descripción detallada

Grupo de funciones para controlar el tiempo en una aplicación.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Con las funciones incluidas en este espacio de nombres, se pueden controlar fácilmente el tiempo transcurrido entre dos puntos de la aplicación; aplicado a un videojuego, se puede retardar la ejecución del programa el periodo necesario para mantener una tasa de fotogramas por segundo constante.

2.3.2. Documentación de las funciones**2.3.2.1. `void tiempo::controlarFps (const u8 fps) [inline]`**

Función que retarda la ejecución del programa el tiempo necesario para mantener los fps constantes

Parámetros

<i>fps</i>	Frames por segundo que se quieren mantener constantes
------------	---

2.3.3. Documentación de las variables

2.3.3.1. u32 tiempo::tick [static]

Variable para almacenar una medida de tiempo. Definirla a cero antes de entrar en el bucle principal

2.4. Referencia del Namespace utf32

Grupo de funciones para convertir cadenas de caracteres a distintos formatos.

Funciones

- std::wstring convertir (const std::string &cadena)

2.4.1. Descripción detallada

Grupo de funciones para convertir cadenas de caracteres a distintos formatos.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Con este grupo de funciones se puede cambiar el formato a una cadena de caracteres de alto nivel, de tal manera que se pueda representar en UTF32, y por tanto, pueda almacenar caracteres de otros alfabetos que no sean los contemplados en la tabla ASCII original.

2.4.2. Documentación de las funciones

2.4.2.1. std::wstring utf32::convertir (const std::string & cadena) [inline]

Función para cambiar la codificación a un std::string, de tal manera que se represente en un std::wstring con formato UTF32. La conversión se realiza reservando la memoria alineada necesaria para almacenar el mismo número de caracteres que tiene la cadena std::string original, pero con el tipo base wchar_t. Se copia la cadena original entera a esta zona de memoria reservada con la función mbstowcs, que cambia la codificación, se añade un carácter de fin de cadena al final y se crea una cadena de alto nivel std::wstring a

partir de esta cadena de bajo nivel de caracteres anchos. Por último, se libera la memoria ocupada por la cadena de bajo nivel temporal.

Parámetros

<i>cadena</i>	Cadena de caracteres que se quiere codificar en UTF32
---------------	---

Devuelve

Cadena de caracteres con codificación utf32

Capítulo 3

Documentación de las clases

3.1. Referencia de la Clase Actor

Clase que proporciona una base para definir actores (también llamados sprites) de una manera fácil.

```
#include <actor.h>
```

Tipos públicos

- `typedef std::set< Figura * > CajasColision`
- `typedef std::map< std::string, CajasColision > Colisiones`
- `typedef std::map< std::string, Animacion * > Animaciones`

Métodos públicos

- `Actor (const std::string &ruta, const Nivel *nivel) throw (ArchivoEx, CódigoEx, TarjetaEx, XmlEx)`
- `virtual ~Actor (void)`
- `u32 x (void) const`
- `u32 y (void) const`
- `u32 xPrevio (void) const`
- `u32 yPrevio (void) const`
- `s16 velX (void) const`
- `s16 velY (void) const`
- `const std::string & estado (void) const`
- `const std::string & estadoPrevio (void) const`
- `const CajasColision & cajasColision (void) const`
- `u16 ancho (void) const`
- `u16 alto (void) const`
- `const std::string & tipoActor (void) const`

- void mover (u32 x, u32 y)
- void setVelX (s16 vx)
- void setVelY (s16 vy)
- bool setEstado (const std::string &e)
- void invertirDibujo (bool inv)
- virtual void dibujar (s16 x, s16 y, s16 z)
- bool colision (const Actor &a)
- virtual void actualizar (void)=0

Métodos protegidos

- void cargarDatosIniciales (const std::string &ruta) throw (ArchivoEx, TarjetaEx, CódigoEx, XmlEx)
- void leerAnimaciones (TiXmlElement *nodo) throw (CódigoEx, XmlEx)
- void leerColisiones (TiXmlElement *nodo)

Atributos protegidos

- s16 _vx
- s16 _vy
- u32 _x
- u32 _y
- u32 _x_previo
- u32 _y_previo
- std::string _estado_actual
- std::string _estado_previo
- std::string _tipo_actor
- bool _invertida
- Colisiones _map_colisiones
- Animaciones _map_animaciones
- const Nivel * _nivel

3.1.1. Descripción detallada

Clase que proporciona una base para definir actores (también llamados sprites) de una manera fácil.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Un actor es la unidad básica en el desarrollo de un videojuego en dos dimensiones, y consiste en un objeto con entidad propia que se puede visualizar en la pantalla, y que almacena información sobre sus distintas características. Para facilitar la creación de actores que participen en el universo de un videojuego, libWiiEsp proporciona esta clase abstracta Actor, que permite controlar desde un mismo sitio los distintos estados por los que puede pasar un actor durante el transcurso del videojuego.

Las características de que dispone un actor son las siguientes: posición actual dentro del universo del juego (pareja de coordenadas (x,y)), posición previa del actor en el universo del juego (otro par de coordenadas (x,y)), velocidad de movimiento horizontal y vertical, estado actual del actor, estado previo del actor, y el tipo de actor. Por supuesto, al derivar esta clase abstracta, se pueden añadir más características según se crea necesario.

Realizando un acercamiento más técnico y concreto, un actor no es más que un conjunto de las características antes mencionadas, además de dos diccionarios donde, a cada estado del actor, le corresponde una animación y un conjunto de cajas de colisión, y que proporciona una manera sencilla y efectiva de cargar todas estas características desde un archivo XML almacenado en la tarjeta SD de la consola.

Existen dos referencias para las coordenadas de un actor. En primer lugar, están las coordenadas de posición del actor dentro del escenario que conforma el propio juego, y que se refieren al desplazamiento en píxeles hacia la derecha del actor respecto al límite izquierdo del escenario (coordenada X), y al desplazamiento hacia abajo, también en píxeles, del actor respecto al límite superior de dicho escenario (coordenada Y). Por otra parte, existe otra referencia para las coordenadas de dibujo de un actor en la pantalla, y que se refieren a la posición respecto a los límites izquierdo y superior de la pantalla, además de la capa en la que se dibuja el actor (consultar la documentación de la clase Screen para más información respecto al dibujo de texturas en la pantalla). Para ilustrar esta diferencia, nada mejor que un ejemplo: supongamos un escenario de 4000 píxeles de ancho y 2000 de alto. La pantalla de la consola Nintendo Wii con una configuración PAL y proporción 4:3 tendrá un tamaño de 640 píxeles de ancho, y 528 de alto. Si el actor a dibujar se encuentra en las coordenadas (1700, 1000) del escenario, se utilizarán estos valores para los cálculos de colisiones, de cambios de posición y de estados. Sin embargo, a la hora de dibujar al actor, hay que tener en cuenta el propio desplazamiento de la pantalla sobre el escenario; si la pantalla tuviera su punto (0,0) en el píxel (1600, 900) del escenario, entonces las coordenadas de dibujo del actor serán (100,100). El desplazamiento de la pantalla respecto a los límites del escenario se controla desde la clase que controla el propio escenario, que será una clase derivada de la clase abstracta Nivel (ver documentación de esta clase para más información).

El comportamiento de un actor viene definido por los distintos estados que puede adoptar a lo largo de la ejecución del juego. Cada estado tiene asociado un comportamiento concreto, y que se ejecutará en cada iteración del bucle principal del juego en la que el actor tenga activo ese estado concreto. Las transiciones entre estados se realizan externamente (generalmente, desde la clase que gestiona el escenario, y que deriva de la clase abstracta Nivel). Los estados se identifican mediante una cadena de caracteres de alto nivel (de tipo std::string), y se crean según aparezcan en el archivo XML que define las animaciones y las cajas de colisión para cada estado, es decir, un estado sólo se creará si aparece en algún

momento en el mencionado archivo XML. El comportamiento del actor según su estado actual debe definirse en el método virtual puro actualizar cuando se derive la clase Actor.

Un detalle muy importante es que existe un estado obligatorio que hay que incluir forzosamente, y es el estado "normal", que es el que se toma por defecto. Si este estado faltara en el archivo XML del actor, se produciría un error y/o un comportamiento inesperado.

La cadena de caracteres que identifica a un estado se toma como clave para dos diccionarios, uno que almacena una animación asociada al estado concreto (consultar documentación de la clase Animación para cualquier duda sobre ésta), y otra que almacena un conjunto (de tipo std::set<Figura*>) de figuras de colisión también asociadas al mismo estado. Estos diccionarios que tienen códigos de estado como clave identificativa se utilizan para que, en cada iteración del bucle principal, se dibuje la animación del actor correspondiente con el estado actual de éste, y sólo se tengan en cuenta las cajas de colisión asociadas al estado actual del actor para evaluar sus colisiones. Se proporcionan métodos para evaluar colisiones entre dos actores de una manera muy sencilla, y también para dibujar el actor en la pantalla en base a unas coordenadas de pantalla que se reciben desde fuera de la clase (la diferencia entre las coordenadas de posición y las de pantalla se explican unos párrafos más arriba).

Otro detalle más es que se almacena un identificador de tipo de actor, es decir, todos los actores que estén gestionados por la misma clase derivada de Actor deberán tener este atributo con el mismo valor.

Como ya se ha comentado, se consigue una separación completa de código fuente y datos, de tal manera que para cambiar las figuras de colisión, los estados o las animaciones de un actor, basta con modificar el archivo XML desde el cual se lee toda esta información. Este archivo tendrá una estructura parecida a esta:

```
<?xml version="1.0" encoding="UTF-8"?>
<actor vx="3" vy="3" tipo="jugador">
    <animaciones>
        <animacion estado="normal" img="chief" sec="0" filas="1" columnas="5" retard
o="3" />
        <animacion estado="mover" img="chief" sec="0,1,2,3,4" filas="1" columnas="5"
retardo="3" />
        <animacion estado="muerte" img="chief" sec="5" filas="1" columnas="6" retardo=
"0" />
    </animaciones>
    <colisiones>
        <rectangulo estado="normal" x1="27" y1="21" x2="55" y2="21" x3="55" y3="96" x
4="27" y4="96" />
        <circulo estado="normal" cx="41" cy="13" radio="8" />
        <rectangulo estado="mover" x1="27" y1="21" x2="55" y2="21" x3="55" y3="96" x4
="27" y4="96" />
        <circulo estado="mover" cx="41" cy="13" radio="8" />
        <sinfigna estado="muerte" />
    </colisiones>
</actor>
```

En el elemento raíz se observan tres atributos, que son la velocidad de movimiento horizontal del actor en píxeles por fotogramas, la velocidad de movimiento vertical, y el tipo de actor.

Se pueden apreciar dos grandes bloques, uno para las animaciones, y otro para las cajas de colisión. Cada animación y figura incluye la información necesaria para ser creada, además de un estado al que se asociará. Sólo puede haber una animación por estado; pero no ocurre así para las cajas de colisión, de las cuales puede haber todas las que se deseen en cada estado. Cade destacar que, en ambos bloques, aparece el estado obligatorio "normal", como ya se indicó anteriormente. Para más información sobre los parámetros de los constructores de cajas de colisión o animaciones, consultar la documentación de cada una de estas clases.

Por último, cabe destacar que al derivar la clase Actor, se le pueden añadir nuevos atributos y métodos según se considere necesario, consiguiendo así partir de una base como es la propia clase Actor, pero pudiendo llegar a la complejidad que se deseé.

Hay que mencionar que a la hora de crear un actor, es apropiado diseñar los cambios entre estados como un autómata finito determinado, que como ya se ha explicado, realizaría sus transiciones entre estados de forma externa. Si se realiza de esta manera, es muy fácil desarrollar un actor en un período de tiempo relativamente pequeño.

3.1.2. Documentación de los 'Typedef' miembros de la clase

3.1.2.1. `typedef std::map<std::string, Animacion*> Actor::Animaciones`

Diccionario que asocia un estado con una animación.

3.1.2.2. `typedef std::set<Figura*> Actor::CajasColision`

Conjunto de cajas de colisión.

3.1.2.3. `typedef std::map<std::string, CajasColision> Actor::Colisiones`

Diccionario que asocia un estado con un conjunto de cajas de colisión.

3.1.3. Documentación del constructor y destructor

3.1.3.1. `Actor::Actor (const std::string & ruta, const Nivel * nivel) throw (ArchivoEx, CodigoEx, TarjetaEx, XmlEx)`

Constructor de la clase Actor. Establece el estado "normal" y la posición a cero.

Parámetros

<i>ruta</i>	Ruta absoluta en la tarjeta SD del archivo XML de datos del actor.
<i>nivel</i>	Puntero constante al nivel en el que se mueve el actor.

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir un archivo
------------------	---

<i>CodigoEx</i>	Se lanza si se detectan dos animaciones para un mismo estado.
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD
<i>XmlEx</i>	Se lanza si hay un error relacionado con un árbol XML

3.1.3.2. **virtual Actor::~Actor(void) [virtual]**

Destructor virtual de la clase Actor. Libera la memoria ocupada por las cajas de colisión y por las animaciones asociadas a cada estado del actor.

3.1.4. Documentación de las funciones miembro

3.1.4.1. **virtual void Actor::actualizar(void) [pure virtual]**

Método virtual puro, en el que se debe implementar el comportamiento del actor (la actualización de las variables internas del actor) según el estado actual de éste. Las modificaciones sobre el estado actual del actor vendrán dadas desde fuera mediante el método setEstado.

3.1.4.2. **u16 Actor::alto(void) const**

Método consultor que devuelve el alto en píxeles de un cuadro de la animación que corresponde al estado actual del actor.

Devuelve

Alto en píxeles de un cuadro de la animación del estado actual del actor.

3.1.4.3. **u16 Actor::ancho(void) const**

Método consultor que devuelve el ancho en píxeles de un cuadro de la animación que corresponde al estado actual del actor.

Devuelve

Ancho en píxeles de un cuadro de la animación del estado actual del actor.

3.1.4.4. **const CajasColision& Actor::cajasColision(void) const**

Método consultor que devuelve una referencia constante al conjunto de figuras de colisión que corresponden al estado actual del actor.

Devuelve

Referencia constante al conjunto de cajas de colisión del estado actual del actor.

3.1.4.5. void Actor::cargarDatosIniciales (const std::string & ruta) throw (ArchivoEx, TarjetaEx, CódigoEx, XmlEx) [protected]

Método que, a partir de la ruta de un archivo XML con un formato concreto, abre éste y lanza los métodos que leen los datos iniciales: las animaciones y las cajas de colisión asociadas a cada estado.

Parámetros

<i>ruta</i>	Ruta absoluta al archivo XML que contiene la información del actor.
-------------	---

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir un archivo
<i>CódigoEx</i>	Se lanza si se detectan dos animaciones para un mismo estado.
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD
<i>XmlEx</i>	Se lanza si el árbol XML con los datos de la animación esuviera incompleto.

3.1.4.6. bool Actor::colision (const Actor & a)

Método para saber si el actor al que pertenece la función colisiona con otro externo. Un actor colisiona con otro si lo hacen entre sí, al menos, una figura de colisión de cada uno de ellos.

Parámetros

<i>a</i>	Actor externo con el que se quiere evaluar si hay colisión.
----------	---

Devuelve

Verdadero si hay colisión entre los actores, y falso en caso contrario.

3.1.4.7. virtual void Actor::dibujar (s16 x, s16 y, s16 z) [virtual]

Método que dibuja el cuadro correspondiente de la animación asociada al estado actual del actor. Las coordenadas que recibe son coordenadas de la pantalla, no del escenario del juego donde se encuentra el actor; y hacen referencia al punto superior izquierdo de un rectángulo imaginario que contendría al actor. Se permite redefinir el método por si fuera necesario para dibujar un tipo de actor especial.

Parámetros

<i>x</i>	Coordenada X de la pantalla donde se dibujará el actor.
<i>y</i>	Coordenada Y de la pantalla donde se dibujará el actor.
<i>z</i>	Capa en la que se dibujará el actor. Más atrás cuanto mayor sea Z. Entre 0 y 999.

3.1.4.8. const std::string& Actor::estado (void) const

Método consultor que devuelve el estado actual en el que se encuentra el actor.

Devuelve

Estado actual en el que se encuentra el actor.

3.1.4.9. const std::string& Actor::estadoPrevio (void) const

Método consultor que devuelve el estado del actor en la iteración anterior del bucle principal.

Devuelve

Estado del actor en la iteración anterior del bucle principal.

3.1.4.10. void Actor::invertirDibujo (bool inv)

Método que establece la orientación de la imagen del actor en la pantalla, sobre el eje vertical.

Parámetros

<i>inv</i>	Verdadero si se debe dibujar la imagen invertida, falso en caso contrario.
------------	--

3.1.4.11. void Actor::leerAnimaciones (TiXmlElement * nodo) throw (CodigoEx, XmlEx)
[protected]

Método que, a partir de un elemento de un árbol XML, lee las animaciones de un actor. Cada animación se espera que tenga una serie de elementos en el árbol XML, y se asocia con un estado concreto del actor. Un estado sólo puede tener una animación asociada.

Parámetros

<i>nodo</i>	Elemento de un árbol XML que contiene las animaciones de un actor.
-------------	--

Excepciones

<i>CodigoEx</i>	Se lanza si se detectan dos animaciones para un mismo estado.
<i>XmlEx</i>	Se lanza si el árbol XML con los datos de la animación esuviera incompleto.

3.1.4.12. void Actor::leerColisiones (TiXmlElement * nodo) [protected]

Método que, a partir de un elemento de un árbol XML, lee las cajas de colisión de un actor. Cada cajas de colisión se espera que tenga una serie de elementos en el árbol XML, y se asocia con un estado concreto del actor.

Parámetros

<i>nodo</i>	Elemento de un árbol XML que contiene las cajas de colisión de un actor.
-------------	--

3.1.4.13. void Actor::mover (u32 x, u32 y)

Método que modifica la posición del actor estableciendo sus nuevas coordenadas. Se toma como origen el punto superior izquierdo del escenario. Si aumenta la X, más a la derecha estará el actor respecto del punto $x = 0$; y si aumenta la Y, más abajo estará el actor respecto del punto $y = 0$.

Parámetros

<i>x</i>	Nuevo valor para la coordenada X del actor.
<i>y</i>	Nuevo valor para la coordenada Y del actor.

3.1.4.14. bool Actor::setEstado (const std::string & e)

Método que modifica el estado del actor. Almacena el estado actual, que pasa a ser el anterior. Si el estado que se recibe no se encuentra en los diccionarios de animaciones y cajas de colisión no se modifica el estado, y se devuelve un valor falso.

Parámetros

<i>e</i>	Nuevo estado en el que se encuentra el actor.
----------	---

Devuelve

Verdadero si se ha cambiado el estado del actor, y falso en caso contrario.

3.1.4.15. void Actor::setVelX (s16 vx)

Método que modifica la velocidad de desplazamiento horizontal del actor. Indica el número de píxeles que se cambia la posición al realizar un movimiento horizontal.

Parámetros

<i>vx</i>	Nuevo valor para la velocidad de desplazamiento horizontal.
-----------	---

3.1.4.16. void Actor::setVelY(s16 vy)

Método que modifica la velocidad de desplazamiento vertical del actor. Indica el número de píxeles que se cambia la posición al realizar un movimiento vertical.

Parámetros

<i>vy</i>	Nuevo valor para la velocidad de desplazamiento vertical.
-----------	---

3.1.4.17. const std::string& Actor::tipoActor(void) const

Método consultor que indica el tipo de actor.

Devuelve

Tipo de actor.

3.1.4.18. s16 Actor::velX(void) const

Método consultor que devuelve el número de píxeles que se desplaza horizontalmente el actor en cada iteración del bucle principal.

Devuelve

Valor de la velocidad horizontal del actor.

3.1.4.19. s16 Actor::velY(void) const

Método consultor que devuelve el número de píxeles que se desplaza verticalmente el actor en cada iteración del bucle principal.

Devuelve

Valor de la velocidad vertical del actor.

3.1.4.20. u32 Actor::x(void) const

Método consultor que devuelve el valor de la coordenada X actual del actor.

Devuelve

Coordenada X del actor.

3.1.4.21. `u32 Actor::xPrevio(void) const`

Método consultor que devuelve el valor de la coordenada X del actor en la iteracion anterior del bucle principal del programa.

Devuelve

Valor de la coordenada X del actor en la iteracion anterior del bucle principal

3.1.4.22. `u32 Actor::y(void) const`

Método consultor que devuelve el valor de la coordenada Y actual del actor.

Devuelve

Coordenada Y del actor.

3.1.4.23. `u32 Actor::yPrevio(void) const`

Método consultor que devuelve el valor de la coordenada Y del actor en la iteracion anterior del bucle principal del programa.

Devuelve

Valor de la coordenada Y del actor en la iteracion anterior del bucle principal

3.1.5. Documentación de los datos miembro**3.1.5.1. `std::string Actor::_estado_actual [protected]`**

Estado actual del actor.

3.1.5.2. `std::string Actor::_estado_previo [protected]`

Estado anterior del actor.

3.1.5.3. `bool Actor::_invertida [protected]`

Indica si se debe dibujar la animación del actor invertida o no.

3.1.5.4. `Animaciones Actor::_map_animaciones [protected]`

Diccionario de animaciones del actor, que asocia un estado con una animación.

3.1.5.5. Colisiones Actor::_map_colisiones [protected]

Diccionario de colisiones del actor, que asocia un estado con un conjunto de cajas de colisión.

3.1.5.6. const Nivel* Actor::_nivel [protected]

Referencia al nivel en el que está participando el actor.

3.1.5.7. std::string Actor::_tipo_actor [protected]

Tipo de actor.

3.1.5.8. s16 Actor::_vx [protected]

Número de píxeles que se desplaza horizontalmente el actor en cada actualización

3.1.5.9. s16 Actor::_vy [protected]

Número de píxeles que se desplaza verticalmente el actor en cada actualización

3.1.5.10. u32 Actor::_x [protected]

Coordenada X de la posición del actor en el escenario del juego. Representa la distancia en píxeles del punto superior izquierdo del actor respecto al límite izquierdo del escenario.

3.1.5.11. u32 Actor::_x_previo [protected]

Coordenada X de la posición del actor en la anterior actualización.

3.1.5.12. u32 Actor::_y [protected]

Coordenada Y de la posición del actor en el escenario del juego. Representa la distancia en píxeles del punto superior izquierdo del actor respecto al límite superior del escenario.

3.1.5.13. u32 Actor::_y_previo [protected]

Coordenada Y de la posición del actor en la anterior actualización.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/actor.h

3.2. Referencia de la Estructura Nivel::actor

Estructura para almacenar de forma temporal la información de un actor.

```
#include <nivel.h>
```

Atributos públicos

- std::string **tipo_actor**
- std::string **xml**
- u32 **x**
- u32 **y**
- std::string **jugador**

3.2.1. Descripción detallada

Estructura para almacenar de forma temporal la información de un actor. Se compone de un identificador del tipo de actor al que pertenece la información, la ruta hasta el archivo XML desde donde se cargan los datos iniciales del actor, las coordenadas (X,Y) del actor respecto al origen de coordenadas del nivel, y una cadena de caracteres que guarda el código identificador del jugador al que representa el actor (si este último atributo es una cadena vacía se asume que el actor no es controlado por ningún jugador).

La documentación para esta estructura fue generada a partir del siguiente fichero:

```
▪ /home/rabbit/Escritorio/libwiiesp/include/nivel.h
```

3.3. Referencia de la Clase Animacion

Clase que permite crear una animación a partir de una imagen cargada en memoria.

```
#include <animacion.h>
```

Métodos públicos

- Animacion (const Imagen &i, const std::string &secuencia, u8 filas=1, u8 columnas=1, u8 retardo=1)
- bool primerPaso (void) const
- u8 pasoActual (void) const
- u16 alto (void) const
- u16 ancho (void) const
- const Imagen & imagen (void) const
- void avanzar (void)
- void reiniciar (void)
- void dibujar (s16 x, s16 y, s16 z, bool invertir=false)

3.3.1. Descripción detallada

Clase que permite crear una animación a partir de una imagen cargada en memoria.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

La clase Animación es un mecanismo para, a partir de una imagen previamente cargada en memoria, conseguir una simulación de movimiento mediante una secuencia de imágenes en dos dimensiones. Se basa en la idea de tener, en una misma textura, todos los fotogramas o cuadros que conforman la animación, y el orden por el que se organizan para mostrar un movimiento fluido.

La misma idea de animación es la que se utiliza en el cine o la televisión: se muestra una secuencia de imágenes, una detrás de otra, pero con leves variaciones entre sí. Lo que se consigue con esto es que el cerebro humano interprete un movimiento reconstruyendo los "huecos" que existen entre una imagen y la siguiente.

Como ya se ha mencionado, esta clase crea una animación a partir de una única imagen. Esta imagen debe organizar los fotogramas en forma de rejilla, de tal manera que todos los cuadros que vayan a componer el movimiento se encuentren en ella formando filas y columnas. Se recomienda ajustar lo mejor posible la cantidad de filas y columnas respecto al número de cuadros que contenga la animación, por ejemplo, si se van a emplear 20 fotogramas, es razonable tener cuatro o cinco filas y cinco o cuatro columnas respectivamente; por el contrario, resultaría menos práctico (sobretodo a la hora de editar el archivo de imagen) tener una única fila con 20 columnas. Pero esto es sólo un consejo, ya que la clase Animación soporta tanto el primer caso como el segundo.

Otro concepto importante sobre las animaciones es el control del tiempo: el ojo humano sólo percibe 24 imágenes por segundo, de tal manera que un movimiento fluido tendrá ese número de fotogramas por segundo, o fps. Sin embargo, es posible crear una animación con un mayor o menor número de cuadros. En todo caso, la cantidad de cuadros que se utilicen para una animación determina tanto su velocidad como su fluidez: por ejemplo, en un sistema con una tasa de 24 fps, una animación que tenga 24 cuadros se apreciará totalmente fluida, y tardará un segundo en completar su ciclo; otra animación construida en el mismo sistema sobre 12 cuadros, por otra parte, tardará medio segundo en concluir, pero será menos fluida que la anterior.

Teniendo en cuenta que, en un sistema, la tasa de fps es constante, y con el objetivo de intentar homogeneizar la duración de las animaciones, independientemente del número de cuadros que la compongan, surge el concepto de retardo. Esta idea consiste en esperar a que pasen algunos fotogramas dibujando el mismo cuadro de la animación antes de pasar a dibujar el siguiente. Siguiendo el mismo ejemplo anterior, se consigue que las dos animaciones tarden lo mismo en completarse si se incluye un retardo de dos fotogramas en la segunda de ellas. Es decir, la animación de 24 cuadros dibujaría uno de éstos en cada

actualización, pero la de 12 lo haría cada dos avances (en una actualización se dibuja un cuadro nuevo, y en la inmediatamente siguiente, se vuelve a dibujar este mismo), de tal manera que ambas animaciones concluirían en el mismo periodo de tiempo.

Esta técnica del retardo consigue, además, que la velocidad de todos los movimientos del programa dependan únicamente de la tasa de fotogramas por segundo, centralizando el control de la velocidad en un punto.

Un último detalle a mencionar es que esta clase está construida sobre las clases Imagen y Screen. Hay que tener en cuenta, por ejemplo, que para cargar una imagen en memoria, existe la limitación de que sus medidas en píxeles deben ser múltiplos de 8. Todos estos detalles se pueden consultar en la documentación de cada una de las clases.

Funcionamiento interno

Cuando se crea una animación, se le deben pasar al constructor de la clase una serie de parámetros, que son un puntero a la imagen (previamente cargada en memoria) que contenga la rejilla de fotogramas, una cadena de caracteres de alto nivel (de tipo std::string) que contenga una secuencia de números enteros positivos, separados por comas y sin espacios (del tipo "0,1,2,3,4,5"), el número de filas y de columnas que hay en la rejilla, y el retardo a aplicar a la visualización de la animación (como ya se ha mencionado, esta cifra indica cada cuantas actualizaciones de la animación se va a dibujar un nuevo cuadro: el valor por defecto 1 indica que se dibuja un nuevo cuadro a cada fotograma).

La secuencia de números enteros que se recibe indica el orden en el que se van dibujando los distintos cuadros de la animación. Cada cuadro de la rejilla corresponde con un número entero positivo, siendo el cero el fotograma de arriba a la izquierda, y avanzando de derecha a izquierda y de arriba abajo. Por ejemplo, una rejilla de tres columnas y dos filas tendrá en su primera fila (de izquierda a derecha) los cuadros cero, uno y dos; y la segunda fila, los cuadros tres, cuatro y cinco (igualmente, de izquierda a derecha). Un mismo cuadro se puede repetir cuantas veces se quiera en la secuencia de una misma animación.

Un objeto de animación proporciona varios métodos observadores para conocer y acceder fácilmente a las variables que lo controlan, por ejemplo, la imagen, el ancho y alto en píxeles de un cuadro (todos los cuadros se consideran iguales), y cuáles son los índices del primer cuadro y del cuadro actual. También existen dos métodos para modificar el estado de la animación, concretamente son reiniciar (método que establece el paso actual al primero de la secuencia) y avanzar (que se encarga de calcular el siguiente cuadro a dibujar teniendo en cuenta el retardo y la propia secuencia de cuadros).

El método dibujar se encarga de, como su propio nombre indica, plasmar en la pantalla el fotograma correspondiente al cuadro actual de la animación, en las coordenadas (x,y,z) que se indiquen. Se da la opción de invertir el fotograma respecto al eje vertical, funcionalidad que proporciona la propia clase Screen a través de su método dibujarCuadro. Este método realiza una llamada a avanzar, con lo que no es necesario preocuparse por la gestión de cuadros desde el exterior.

Una cosa más a tener en cuenta es que el destructor de la clase es el predeterminado, por lo que no se destruye la imagen asociada a la animación, y hay que destruirla manualmente en caso de que se quiera liberar la memoria ocupada por ésta.

Ejemplo de uso

```
// Crear una animación a partir de una imagen de 3 filas y 4 columnas (12 cuadro
// s),
// con un retardo de 2 (pasar a un nuevo cuadro cada dos actualizaciones)
Imagen* rejilla = new Imagen();
rejilla->cargarBmp( "/apps/wiipang/media/rejilla.bmp" );
Animacion anima( rejilla, "0,1,2,3,4,5,6,7,8,9,10,11", 3, 4, 2 );
// Bucle principal del juego
while( 1 )
{
    // Dibujar animación
    anima.dibujar( 100, 100, 50 );
    // Dibujar animación invertida
    anima.dibujar( 250, 100, 51, true );
    // ...
    screen->flip();
}
```

3.3.2. Documentación del constructor y destructor

3.3.2.1. Animacion::Animacion (const Imagen & i, const std::string & secuencia, u8 filas = 1, u8 columnas = 1, u8 retardo = 1)

Constructor de la clase Animacion.

Parámetros

<i>i</i>	Imagen que será base de la animación
<i>secuencia</i>	Enteros separados por comas, indica los distintos pasos correlativos de una animación
<i>filas</i>	Número de filas en la rejilla de la imagen i
<i>columnas</i>	Número de columnas en la rejilla de la imagen i
<i>retardo</i>	Número de frames que tienen que pasar para que se produzca un avance en la animación

3.3.3. Documentación de las funciones miembro

3.3.3.1. u16 Animacion::alto (void) const

Devuelve el alto, en píxeles, de cada cuadro de la animación

Devuelve

Alto, en píxeles, de cada cuadro de la animación

3.3.3.2. u16 Animacion::ancho (void) const

Devuelve el ancho, en píxeles, de cada cuadro de la animación

Devuelve

Ancho, en píxeles, de cada cuadro de la animación

3.3.3.3. void Animacion::avanzar (void)

Avanza al paso siguiente del actual. Si el actual es el último, se sitúa en el primero.

3.3.3.4. void Animacion::dibujar (s16 x, s16 y, s16 z, bool invertir = false)

Dibuja en la pantalla la imagen asociada con el paso actual de la animación, y avanza al paso siguiente.

Parámetros

<i>x</i>	Coordenada X donde se dibujará la imagen asociada al paso actual de la animación
<i>y</i>	Coordenada Y donde se dibujará la imagen asociada al paso actual de la animación
<i>z</i>	Capa donde se dibujará la imagen asociada al paso actual de la animación (entre 0 y 999)
<i>invertir</i>	Verdadero si se quiere dibujar el cuadro de la textura invertido respecto al eje vertical

3.3.3.5. const Imagen& Animacion::imagen (void) const

Devuelve la imagen de la animación.

Devuelve

Referencia constante a la imagen de la animación.

3.3.3.6. u8 Animacion::pasoActual (void) const

Devuelve el número que indica el paso actual de la animación. El primero es cero.

Devuelve

Número que indica el paso actual de la animación.

3.3.3.7. bool Animacion::primerPaso (void) const

Indica si la animación se encuentra en el primer paso.

Devuelve

Verdadero si la animación se encuentra en el primer paso, o falso en caso contrario.

3.3.3.8. void Animacion::reiniciar (void)

Sitúa el contador de pasos de la animación en el primer paso.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/animacion.h

3.4. Referencia de la Clase Circulo

Clase que representa un círculo en el sistema de gestión de colisiones.

```
#include <colision.h>
```

Métodos públicos

- Circulo (Punto centro, f32 radio)
- bool hayColision (Figura *f, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Circulo *c, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Rectangulo *r, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Punto *p, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- Punto & centro (void)
- const Punto & centro (void) const
- f32 & radio (void)
- const f32 & radio (void) const

3.4.1. Descripción detallada

Clase que representa un círculo en el sistema de gestión de colisiones.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Se representa mediante un punto (su centro) y un radio, que es un decimal en coma flotante de 32 bits. Deriva de la clase Figura.

3.4.2. Documentación del constructor y destructor

3.4.2.1. Circulo::Circulo (Punto centro, f32 radio) [inline]

Constructor de la clase Círculo.

Parámetros

<i>centro</i>	Centro del círculo
<i>radio</i>	Radio del círculo

3.4.3. Documentación de las funciones miembro

3.4.3.1. Punto& Circulo::centro (void) [inline]

Método modificador que devuelve una referencia al centro del circulo

Devuelve

Referencia al centro del circulo

3.4.3.2. const Punto& Circulo::centro (void) const [inline]

Método consultor que devuelve el centro del circulo

Devuelve

Punto centro del circulo

3.4.3.3. bool Circulo::hayColision (Figura * f, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [inline, virtual]

Método para saber si una figura de tipo desconocido que se recibe por parámetro colisiona con el círculo. Implementación de la técnica de double dispatch.

Parámetros

<i>f</i>	Puntero a una figura con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del círculo
<i>dy1</i>	Desplazamiento vertical del círculo
<i>dx2</i>	Desplazamiento horizontal de la figura que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical de la figura que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.4.3.4. bool Circulo::hayColision (Punto * p, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [virtual]

Método para saber si un punto que se recibe por parámetro colisiona con el círculo.
Implementación de la técnica de double dispatch.

Parámetros

<i>p</i>	Puntero a un punto con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del círculo
<i>dy1</i>	Desplazamiento vertical del círculo
<i>dx2</i>	Desplazamiento horizontal del punto que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del punto que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.4.3.5. bool Circulo::hayColision (Circulo * c, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [virtual]

Método para saber si un círculo que se recibe por parámetro colisiona con el círculo.
Implementación de la técnica de double dispatch.

Parámetros

<i>c</i>	Puntero a un círculo con el que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del círculo
<i>dy1</i>	Desplazamiento vertical del círculo
<i>dx2</i>	Desplazamiento horizontal del círculo que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del círculo que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.4.3.6. bool Circulo::hayColision (Rectangulo * r, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [virtual]

Método para saber si un rectángulo desconocido que se recibe por parámetro colisiona con el círculo. Implementación de la técnica de double dispatch.

Parámetros

<i>r</i>	Puntero a un rectángulo con el que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del círculo
<i>dy1</i>	Desplazamiento vertical del círculo
<i>dx2</i>	Desplazamiento horizontal del rectángulo que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del rectángulo que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.4.3.7. `f32& Circulo::radio(void) [inline]`

Método modificador que devuelve una referencia al radio del circulo

Devuelve

Referencia al radio del circulo

3.4.3.8. `const f32& Circulo::radio(void) const [inline]`

Método consultor que devuelve el centro del circulo

Devuelve

Punto centro del circulo

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/collision.h

3.5. Referencia de la Clase Expcion

Clase de excepción básica.

```
#include <excepcion.h>
```

Métodos públicos

- Expcion (const std::string &e)
- const char * what () const throw ()
- virtual ~Expcion (void) throw ()

3.5.1. Descripción detallada

Clase de excepción básica.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Cuando se captura, puede informar de qué error se ha producido mediante el método what().

3.5.2. Documentación del constructor y destructor

3.5.2.1. Expcion::Expcion (const std::string & e) [inline]

Constructor de la clase de excepción

Parámetros

e	Cadena de caracteres que indica el error ocurrido
---	---

3.5.2.2. virtual Expcion::~Expcion (void) throw () [inline, virtual]

Destructor de la clase de excepción

3.5.3. Documentación de las funciones miembro

3.5.3.1. const char* Expcion::what () const throw () [inline]

Método observador que devuelve una descripción del error que ha provocado la excepción

Devuelve

Descripción del error que ha provocado la excepción

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/expcion.h

3.6. Referencia de la Clase Figura

Clase abstracta que sirve como base a todas las figuras de colisión definidas en el sistema.

```
#include <colision.h>
```

Métodos públicos

- virtual bool hayColision (Figura *f, s16 dx1, s16 dy1, s16 dx2, s16 dy2)=0
- virtual bool hayColision (Punto *f, s16 dx1, s16 dy1, s16 dx2, s16 dy2)=0
- virtual bool hayColision (Circulo *f, s16 dx1, s16 dy1, s16 dx2, s16 dy2)=0
- virtual bool hayColision (Rectangulo *f, s16 dx1, s16 dy1, s16 dx2, s16 dy2)=0
- virtual ~Figura (void)

Métodos públicos estáticos

- static Figura * leerRectangulo (TiXmlElement *nodo)
- static Figura * leerCirculo (TiXmlElement *nodo)
- static Figura * leerPunto (TiXmlElement *nodo)

3.6.1. Descripción detallada

Clase abstracta que sirve como base a todas las figuras de colisión definidas en el sistema.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

La clase abstracta Figura es la base del módulo de gestión de colisiones integrado en LibWiiEsp. Se basa en una implementación de la técnica double dispatch (que a su vez es una implementación del patrón Visitante) consistente en una especie de polimorfismo donde la elección del método a ejecutar depende no sólo del objeto que lo ejecuta, si no del tipo del parámetro que recibe. En la práctica, al emplear esta técnica se consigue evitar la comprobación de tipos mediante estructuras de tipo condicional (if, case, etc.) cuando se quieren comparar dos objetos de clases derivadas de Figura.

Entrado un poco más en detalle, double dispatch implica que toda clase derivada de Figura implementará un método hayColision, en el que recibirá un puntero a zona de memoria donde se almacenará otra Figura. Este método devolverá un valor booleano, que será verdadero si hay colisión entre las dos figuras, o falso en caso contrario. Internamente,

devuelve el resultado de llamar al método `hayColision` de la segunda figura, pasando como parámetro la figura actual (es decir, el objeto `this` en el contexto de la primera figura). Con este intercambio de parámetros se consigue que esta segunda llamada se realice conociendo el tipo exacto del parámetro que se pasa, ya que en el contexto de ejecución (se hace desde un método de la primera figura) se conoce el tipo del objeto `this`. Al llamarse el método `hayColision` de la segunda figura, recibiendo un parámetro con el tipo perfectamente identificado, esta segunda figura ya sabe los tipos de ambos objetos (conoce su propio tipo por el contexto de ejecución, y el de la primera figura por lo explicado anteriormente), de tal manera que se ejecuta el método adecuado con los cálculos necesarios para averiguar si hay colisión entre ambas figuras o no.

Gracias a esta técnica de double dispatch se consigue una mayor escalabilidad del código de las colisiones, ya que resulta muy sencillo añadir nuevas figuras que detecten colisiones entre sí y entre las ya existentes. Únicamente basta con derivar la clase abstracta `Figura`, definir el método `hayColision` que recibe un puntero a `Figura` (tal y como se ha descrito e implementan las figuras existentes), y crear los métodos `hayColision` que calculen si se dan colisiones entre los distintos tipos de figuras. El inconveniente de esta técnica es que se debe duplicar algo de código, debido a que un método de `Figura1` que compruebe la colisión entre `Figura1` y `Figura2` debe estar definido exactamente igual al método de `Figura2` que evalúe la colisión entre esos dos tipos de figuras.

El otro aspecto importante relativo a las figuras de colisión es el desplazamiento de éstas. Una figura de colisión está determinada, como mínimo, por una pareja de coordenadas cuyo punto de origen (es decir, el `(0,0)`) corresponde con el punto superior izquierdo de un objeto del sistema; objeto que tendrá, a su vez, una pareja de coordenadas `(x,y)` respecto al límite izquierdo del escenario (coordenada X, cuanto mayor sea, más alejado hacia la derecha estará el objeto de este límite izquierdo) y al límite superior del escenario (coordenada Y, cuanto mayor sea, más alejado hacia abajo estará el objeto de este límite superior). Es decir, las coordenadas de una figura de colisión son relativas a un punto que no tiene por qué ser el origen `(0,0)` de las coordenadas de los objetos del sistema.

Por ejemplo, suponiendo un personaje que tenga asociado un rectángulo de colisiones, este personaje tiene unas coordenadas `(x,y)` respecto al punto superior izquierdo del escenario, y que determinan la posición del personaje en dicho escenario. Esta pareja de coordenadas variará en función del comportamiento del personaje durante la ejecución del juego. Sin embargo, las parejas de coordenadas de los puntos que determinan el rectángulo de colisiones se mantendrán fijas en todo momento, ya que son relativas al punto superior izquierdo del personaje, y no al del escenario. Esto provoca que, para conocer la posición de un punto del rectángulo respecto al origen de coordenadas del escenario, haya que sumar el valor de la coordenada X del personaje con el de la coordenada X del punto; y análogamente para las coordenadas Y.

Ciertamente, es más complicado gestionar dos sistemas de referencia (uno para los objetos del sistema, y otro para las figuras de colisión), pero se ha decidido que cada figura de colisión esté asociada a un objeto del sistema, y que sus coordenadas sean relativas al punto superior izquierdo de su objeto asociado, porque supone un coste muy alto en procesamiento el tener que estar actualizando las posiciones absolutas de todas las figuras de colisión en cada iteración del bucle principal.

En base a todo lo explicado, para poder realizar los cálculos de colisiones correctamente hay que contar con la posición absoluta del objeto que tiene asociado cada figura de colisión. Esto se consigue con el concepto de desplazamiento, y que no es más que dos parejas de valores que se le pasan al método `hayColision` de una figura cuando se quiere saber si colisiona con otra. Cada pareja de valores son las coordenadas del objeto asociado a una de las figuras, para que se tengan en cuenta a la hora de calcular si hay colisión o no. Concretamente, la primera pareja de valores `dx1` y `dy1` indican el desplazamiento de la figura de la cual se llama a su método `hayColision`, y los valores `dx2` y `dy2`, el desplazamiento de la figura que se recibe por parámetro.

3.6.2. Documentación del constructor y destructor

3.6.2.1. `virtual Figura::~Figura(void) [inline, virtual]`

Destructor virtual de la clase Figura.

3.6.3. Documentación de las funciones miembro

3.6.3.1. `virtual bool Figura::hayColision (Figura * f, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [pure virtual]`

Método virtual puro para saber si una figura que se recibe por parámetro colisiona con la figura a la que pertenece el método. La definición de este método debe ser siempre la misma, ya que con ello se consigue la implementación de la técnica de double dispatch. Es la siguiente: `return f->hayColision(this, dx2, dy2, dx1, dy1);`

Parámetros

<code>f</code>	Puntero a una figura con la que se quiere comprobar si existe colisión
<code>dx1</code>	Desplazamiento horizontal de la figura actual
<code>dy1</code>	Desplazamiento vertical de la figura actual
<code>dx2</code>	Desplazamiento horizontal de la figura que se recibe por parámetro
<code>dy2</code>	Desplazamiento vertical de la figura que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementado en Punto, Rectangulo y Circulo.

3.6.3.2. `virtual bool Figura::hayColision (Punto * f, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [pure virtual]`

Método virtual puro para saber si un punto que se recibe por parámetro colisiona con la figura a la que pertenece el método.

Parámetros

<i>f</i>	Puntero a una figura con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal de la figura actual
<i>dy1</i>	Desplazamiento vertical de la figura actual
<i>dx2</i>	Desplazamiento horizontal de la figura que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical de la figura que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementado en Punto, Rectangulo y Circulo.

3.6.3.3. virtual bool Figura::hayColision (Rectangulo * *f*, s16 *dx1*, s16 *dy1*, s16 *dx2*, s16 *dy2*) [pure virtual]

Método virtual puro para saber si un rectángulo que se recibe por parámetro colisiona con la figura a la que pertenece el método.

Parámetros

<i>f</i>	Puntero a una figura con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal de la figura actual
<i>dy1</i>	Desplazamiento vertical de la figura actual
<i>dx2</i>	Desplazamiento horizontal de la figura que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical de la figura que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementado en Punto, Rectangulo y Circulo.

3.6.3.4. virtual bool Figura::hayColision (Circulo * *f*, s16 *dx1*, s16 *dy1*, s16 *dx2*, s16 *dy2*) [pure virtual]

Método virtual puro para saber si un círculo que se recibe por parámetro colisiona con la figura a la que pertenece el método.

Parámetros

<i>f</i>	Puntero a una figura con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal de la figura actual
<i>dy1</i>	Desplazamiento vertical de la figura actual
<i>dx2</i>	Desplazamiento horizontal de la figura que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical de la figura que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementado en Punto, Rectangulo y Circulo.

3.6.3.5. static Figura* Figura::leerCirculo (TiXmlElement * nodo) [static]

Método que lee un círculo de colisión desde un elemento de un árbol XML.

Parámetros

<i>nodo</i>	Elemento de un árbol XML que contiene la información de un círculo de colisión.
-------------	---

Devuelve

Puntero a un círculo de colisión creado a partir de la información del elemento XML.

3.6.3.6. static Figura* Figura::leerPunto (TiXmlElement * nodo) [static]

Método que lee un punto de colisión desde un elemento de un árbol XML.

Parámetros

<i>nodo</i>	Elemento de un árbol XML que contiene la información de un punto de colisión.
-------------	---

Devuelve

Puntero a un punto de colisión creado a partir de la información del elemento XML.

3.6.3.7. static Figura* Figura::leerRectangulo (TiXmlElement * nodo) [static]

Método que lee un rectángulo de colisión desde un elemento de un árbol XML.

Parámetros

<i>nodo</i>	Elemento de un árbol XML que contiene la información de un rectángulo de colisión.
-------------	--

Devuelve

Puntero a un rectángulo de colisión creado a partir de la información del elemento XML.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/collision.h

3.7. Referencia de la Clase Fuente

Clase que permite gestionar una fuente de texto a través de FreeType2.

```
#include <fuente.h>
```

Métodos públicos

- Fuente (const std::string &ruta) throw (ArchivoEx, TarjetaEx)
- void escribir (const std::string &texto, u8 tam, s16 x, s16 y, s16 z, u32 color) const
- void escribir (const std::wstring &texto, u8 tam, s16 x, s16 y, s16 z, u32 color) const
- ~Fuente (void)

Métodos públicos estáticos

- static void inicializar (void)
- static void apagar (void)

Atributos públicos estáticos

- static FT_Library library

3.7.1. Descripción detallada

Clase que permite gestionar una fuente de texto a través de FreeType2.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Esta clase está relacionada con la carga de archivos y utilización de fuentes de texto, con el objetivo de proporcionar de una manera sencilla y eficaz las herramientas necesarias para escribir textos en la pantalla. La biblioteca que sirve de base para trabajar con los archivos de fuentes es un port de FreeType2 a Nintendo Wii, de tal manera que estarán soportados todos los formatos de archivos de fuentes que estén contemplados en esta biblioteca.

Cada instancia de esta clase representa un único archivo de fuentes de texto cargado en memoria y listo para ser utilizado. Los archivos de fuentes se almacenarán en la tarjeta SD de forma dinámica, de tal manera que en cualquier punto del programa se podrá acceder a la tarjeta SD y cargar una nueva fuente de texto en el sistema.

Las cadenas "tradicionales" std::string trabajan con un byte por carácter (es decir, con un carácter base de tipo char, 8 bits). En el formato Unicode, sin embargo, un carácter

puede ocupar hasta cuatro caracteres, con lo cual se hace evidente que hay que utilizar otra estructura de datos para poder dar soporte a otros alfabetos además del latino. La solución es utilizar cadenas con caracteres base que puedan almacenar varios bytes por carácter, con el tipo `wchar_t`.

Se ha implementado soporte para escribir en pantalla caracteres Unicode de hasta 32 bits, para lo cual se utiliza la clase `std::wstring`, que no es más que una `std::string` pero utilizando, en lugar de `char` como carácter base, el tipo `wchar_t`. Debido a ello, se pueden escribir en la pantalla todos los caracteres existentes, desde el alfabeto latino, hasta kanjis japoneses, pasando por cirílico y más.

Para transformar una `std::string` en una `std::wstring` se reserva memoria alineada de tipo `wchar_t`, para tantos caracteres como tenga la cadena original. Después se utiliza la función `mbstowcs()`, que transforma una cadena con caracteres de un sólo byte (pero que puede estar almacenando caracteres multibyte, sólo que éstos estarían repartidos entre varios caracteres) en una cadena de caracteres de 32 bits por carácter.

Una vez tenemos una cadena de caracteres de 32 bits por carácter con el texto que se quiere escribir en la pantalla, el proceso para hacerlo es sencillo. Se trata de recorrer el texto, carácter a carácter, extrayendo el bitmap asociado a un carácter cada vez (esta funcionalidad la proporciona FreeType2, ya que un archivo de fuentes contiene un gráfico vectorial por cada carácter, a partir del cual se obtiene el mapa de bits), y dibujando el bitmap en las coordenadas correspondientes de la pantalla. Este dibujo se realiza píxel a píxel, ya que la imagen extraída tiene píxeles coloreados (los que componen el propio carácter) y píxeles con valor cero (un mapa de bits es rectangular, de tal manera que hay píxeles invisibles, y éstos son los que tiene el valor nulo).

Es muy importante tener claro que, para poder utilizar un juego de caracteres concreto, éste debe estar contemplado en el archivo de fuentes (es decir, que si se quiere escribir un texto en chino mandarín, el archivo de fuentes debe soportar chino mandarín, en otro caso, se escribirán caracteres no esperados en la pantalla).

Por último, un detalle que se ha contemplado, es que si el archivo de fuentes soporta kerning, éste se utiliza para todos los caracteres. El kerning es una funcionalidad que permite que los caracteres no tengan todos la misma separación, sino que, según el carácter, la separación entre estos sea la adecuada (para ilustrar esta situación, basta con pensar que el carácter ‘i’ no tiene el mismo ancho que el carácter ‘m’, y si la separación entre caracteres es la misma, puede no quedar bien el que algunas letras se vean con más separación que otras en una misma palabra).

Funcionamiento interno

Antes de poder utilizar ninguna fuente de texto, hay que inicializar la biblioteca FreeType2, mediante el método de clase `Fuente::inicializar()`.

El constructor de la clase se ocupa de cargar un archivo de fuentes alojado en la SD (lo primero que hace es comprobar que la unidad de la tarjeta SD esté montada y lista para ser utilizada). Se utiliza una función de la propia biblioteca FreeType2 para realizar la carga, ya que el port de ésta a Wii abstrae de tener que preocuparse por el Endian, la memoria alineada y demás detalles.

Una vez cargada la fuente, para escribir un texto, ya sea desde un std::string o desde un std::wstring, basta con indicar el texto a escribir, las coordenadas (x, y, z) en las cuales se quiere escribir el texto (las coordenadas corresponden a la esquina superior izquierda del texto, para cualquier duda sobre el sistema de coordenadas, consultar la documentación de la clase Screen), el tamaño de los caracteres, y el color de relleno de éstos, en formato hexadecimal 0xRRGGBBAA (consultar clase Screen para el formato de color).

Es importante tener en cuenta que si el texto escrito se sale de la pantalla, la clase no proporciona ninguna manera de saberlo, así que si se quiere controlar esta situación, se tienen que hacer los cálculos antes de llamar a los métodos de escritura.

Cada píxel del mapa de bits generado por FreeType2 que se vaya a dibujar, se hará con el método dibujarPunto() de la clase Screen.

Por último, el destructor de la clase se encarga de liberar la memoria ocupada por las estructuras internas (concretamente, una estructura propia de FreeType2 que se genera en el constructor).

Sencillo ejemplo de uso

```
// Inicializar la clase
Fuente::inicializar();
// Cargar la fuente Verdana
Fuente verdana( "/apps/wiipang/media/verdana.ttf" );
// Escribir un texto con caracteres latinos, tamaño 30, coordenadas (50,50,4) y
// en color rojo
verdana.escribir( "Hola", 30, 50, 50, 4, 0xFF0000FF );
```

3.7.2. Documentación del constructor y destructor

3.7.2.1. Fuente::Fuente (const std::string & ruta) throw (ArchivoEx, TarjetaEx)

Constructor de la clase Fuente. Lee un archivo de fuentes desde la tarjeta SD

Parámetros

<i>ruta</i>	Ruta absoluta hasta el archivo de fuentes
-------------	---

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir el archivo al que apunta la ruta aportada
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD

3.7.2.2. Fuente::~Fuente (void)

Destructor de la clase Fuente

3.7.3. Documentación de las funciones miembro

3.7.3.1. static void Fuente::apagar (void) [inline, static]

Método de clase para "apagar" la biblioteca de fuentes.

3.7.3.2. void Fuente::escribir (const std::wstring & texto, u8 tam, s16 x, s16 y, s16 z, u32 color) const

Método para escribir en pantalla un texto de 32 bits por carácter con la fuente de la instancia.

Parámetros

<i>texto</i>	Cadena de texto de alto nivel de 32 bits que se quiere escribir en la pantalla
<i>tam</i>	Tamaño en píxeles con el que se quiere dibujar el texto
<i>x</i>	Coordenada X en pantalla del punto superior izquierdo del primer carácter que se dibujara
<i>y</i>	Coordenada Y en pantalla del punto superior izquierdo del primer carácter que se dibujara
<i>z</i>	Capa en la que se dibujará el texto (primer plano es 0, fondo es 999)
<i>color</i>	Color en el que se quiere dibujar el texto, en formato 0xRRGGBAA

3.7.3.3. void Fuente::escribir (const std::string & texto, u8 tam, s16 x, s16 y, s16 z, u32 color) const

Método para escribir en pantalla un texto con la fuente almacenada en la instancia.

Parámetros

<i>texto</i>	Cadena de texto de alto nivel que se quiere escribir en la pantalla
<i>tam</i>	Tamaño en píxeles con el que se quiere dibujar el texto
<i>x</i>	Coordenada X en pantalla del punto superior izquierdo del primer carácter que se dibujara
<i>y</i>	Coordenada Y en pantalla del punto superior izquierdo del primer carácter que se dibujara
<i>z</i>	Capa en la que se dibujará el texto (primer plano es 0, fondo es 999)
<i>color</i>	Color en el que se quiere dibujar el texto, en formato 0xRRGGBAA

3.7.3.4. static void Fuente::inicializar (void) [inline, static]

Método de clase para inicializar la biblioteca de fuentes.

3.7.4. Documentación de los datos miembro

3.7.4.1. FT_Library Fuente::library [static]

Variable de clase que almacena el manejador de la biblioteca FreeType2

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/fuente.h

3.8. Referencia de la Clase Galeria

Clase que gestiona los recursos media del sistema, y facilita su carga en memoria y el acceso a ellos.

```
#include <galeria.h>
```

Métodos públicos

- void inicializar (const std::string &ruta) throw (ArchivoEx, ImagenEx, TarjetaEx, XmlEx)
- const Imagen & imagen (const std::string &codigo) throw (CodigoEx)
- const Musica & musica (const std::string &codigo) throw (CodigoEx)
- const Sonido & sonido (const std::string &codigo) throw (CodigoEx)
- const Fuente & fuente (const std::string &codigo) throw (CodigoEx)

Métodos públicos estáticos

- static Galeria * get_instance (void)
- static void destroy (void)

Métodos protegidos

- Galeria (void)
- ~Galeria (void)
- Galeria (const Galeria &g)
- Galeria & operator= (const Galeria &g)

3.8.1. Descripción detallada

Clase que gestiona los recursos media del sistema, y facilita su carga en memoria y el acceso a ellos.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Esta clase consiste en un conjunto de diccionarios (en C++, maps), uno por cada tipo de recurso media que se vaya a utilizar en el desarrollo del juego. La carga de éstos se realiza al principio de la ejecución a partir de un archivo XML con un formato concreto. Además, esta clase implementa un patrón Singleton, de tal manera que los recursos media están disponibles en todo momento, y en cualquier parte del sistema.

En estos momentos, los cuatro recursos que se contemplan son: imágenes, sonidos, pistas de música y fuentes de texto. Cada uno de estos recursos se encuentra disponible mediante un código único de identificación, en formato std::string, y que se lee desde el ya mencionado archivo de datos XML. Si en un futuro se decidiera implementar un nuevo tipo de recurso, basta con implementar su correspondiente diccionario, la función de lectura desde el XML, y las funciones consultoras.

Dos recursos de distinto tipo pueden tener identificadores idénticos, ya que se archivan cada uno en su diccionario. Por otra parte, los recursos se crean mediante su constructor, y con los parámetros que cada uno de ellos espera (como mínimo, un recurso debe tener su código identificativo y la ruta hasta el archivo que contiene el media que se va a cargar). Por ejemplo, si se va a cargar un sonido, los parámetros que se esperan son el código (en formato std::string), la ruta absoluta hasta el archivo situado en la tarjeta SD (también en formato std::string), y los valores numéricos (enteros de 8 bits sin signo) que indican el volumen de cada uno de los dos canales de reproducción.

Funcionamiento interno

Los diccionarios sobre los que trabaja esta clase consisten en parejas formadas por una cadena de caracteres de alto nivel (del tipo ya mencionado, std::string), que actúa como clave identificativa, y un puntero a la zona de memoria donde se encuentra el recurso correspondiente. Cuando se solicita a la clase un recurso media, esta operación se hace a partir del código del media. El método comprueba que el código recibido corresponde a un recurso, lo busca y devuelve el puntero que apunta a él; si el código no estuviera registrado en el diccionario, se lanzaría una excepción.

La clase necesita que la tarjeta SD esté montada para poder acceder a los archivos que contienen los recursos; en caso contrario, se lanzará una excepción al intentar acceder a éstos.

A la hora de inicializar la galería de recursos, se le debe pasar como parámetro la ruta absoluta (en la tarjeta SD) hasta el archivo XML en el que se encuentra la información de los media. Este archivo debe tener un nodo raíz cuyos hijos serán, cada uno de ellos, un recurso media. El método que se encarga de recorrer la información e ir creando cada recurso a partir de ésta irá leyendo cada etiqueta de recurso, la identificará por su tipo, y según sea éste, intentará leer unos u otros atributos de la etiqueta. A continuación se muestra un ejemplo de etiqueta válida para cada uno de los cuatro tipos de recursos contemplados en estos momentos por la clase:

```
<imagen codigo="fondo" formato="bmp" ruta="/apps/wiipang/media/fondo.bmp" />
```

```
<musica codigo="rock" volumen="128" ruta="/apps/wiipang/media/rock.mp3" />
<sonido codigo="sound" volumen="255" ruta="/apps/wiipang/media/sound.pcm" />
<fuente codigo="arial" ruta="/apps/wiipang/media/arial.ttf" />
```

Añadir un nuevo recurso media al sistema es tan sencillo como copiar el archivo a la tarjeta SD (se recomienda mantener una carpeta donde se almacenen todos los recursos, por ejemplo, una llamada 'media'), y después añadir la etiqueta correspondiente al archivo que contenga la información de la galería. De esta forma se evita tener que volver a compilar el proyecto en caso de querer añadir o modificar un recurso, consiguiendo así una separación de código y datos.

Por último, al crear la galería, se establece que, cuando se salga del programa, se llame a su destructor que, a su vez, llama a los destructores de cada uno de los recursos almacenados, de tal manera que se evitan fugas de memoria.

Ejemplo de uso

```
// Inicialización de la galería a partir de un XML
galeria->inicializar("/apps/wiipang/xml/galeria.xml");
// Ejemplo de acceso a cada uno de los recursos
galeria->sonido("sound")->play();
galeria->fuente("arial")->escribir(";Funciona!", 30, 80, 100, 10, 0x00FF00FF);
galeria->imagen("fondo")->dibujar(0, 0, 900);
galeria->musica("rock")->play();
```

3.8.2. Documentación del constructor y destructor

3.8.2.1. Galeria::Galeria(void) [inline, protected]

Constructor de la clase Galeria. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.8.2.2. Galeria::~Galeria(void) [protected]

Destructor de la clase Galeria. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.8.2.3. Galeria::Galeria(const Galeria & g) [protected]

Constructor de copia de la clase Galeria. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.8.3. Documentación de las funciones miembro

3.8.3.1. static void Galeria::destroy(void) [inline, static]

Función estática que destruye la instancia activa de la galeria de medias, llamando a su destructor.

3.8.3.2. const Fuente& Galeria::fuente (const std::string & *codigo*) throw (CodigoEx)

Método que devuelve un puntero a la fuente que tenga el código que se indica

Parámetros

<i>codigo</i>	Código de la fuente que se quiere obtener
---------------	---

Devuelve

Puntero a la fuente que corresponde al código introducido

Excepciones

<i>CodigoEx</i>	Se lanza si el código recibido no corresponde a ninguna fuente de texto
-----------------	---

3.8.3.3. static Galeria* Galeria::get_instance (void) [inline, static]

Función estática que devuelve la instancia activa de la galeria de medias en el sistema. En el caso de no haber ninguna instancia, se crea y se devuelve. Implementación del patrón Singleton.

Devuelve

Puntero a la instancia activa de la galeria de medias en el sistema

3.8.3.4. const Imagen& Galeria::imagen (const std::string & *codigo*) throw (CodigoEx)

Método que devuelve un puntero a la imagen que tenga el código que se indica

Parámetros

<i>codigo</i>	Código de la imagen que se quiere obtener
---------------	---

Devuelve

Puntero a la imagen que corresponde al código introducido

Excepciones

<i>CodigoEx</i>	Se lanza si el código recibido no corresponde a ninguna imagen
-----------------	--

3.8.3.5. void Galeria::inicializar (const std::string & *ruta*) throw (ArchivoEx, ImagenEx, TarjetaEx, XmlEx)

Método para inicializar la galería, y cargar los medias en la biblioteca de medias.

Parámetros

<i>ruta</i>	Ruta absoluta hasta el archivo XML en el que se encuentra la información de los media
-------------	---

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir un archivo
<i>ImagenEx</i>	Se lanza si sucede un error relacionado con la carga de una imagen
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD
<i>XmlEx</i>	Se lanza si hay un error relacionado con un árbol XML

3.8.3.6. const Musica& Galeria::musica (const std::string & *codigo*) throw (CodigoEx)

Método que devuelve un puntero a la pista de música que tenga el código que se indica

Parámetros

<i>codigo</i>	Código de la pista de música que se quiere obtener
---------------	--

Devuelve

Puntero a la pista de música que corresponde al código introducido

Excepciones

<i>CodigoEx</i>	Se lanza si el código recibido no corresponde a ninguna pista de música
-----------------	---

3.8.3.7. Galeria& Galeria::operator= (const Galeria & *g*) [protected]

Operador de asignación de la clase Galeria. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.8.3.8. const Sonido& Galeria::sonido (const std::string & *codigo*) throw (CodigoEx)

Método que devuelve un puntero al sonido que tenga el código que se indica

Parámetros

<i>codigo</i>	Código del sonido que se quiere obtener
---------------	---

Devuelve

Puntero al sonido que corresponde al código introducido

Excepciones

<i>CodigoEx</i>	Se lanza si el código recibido no corresponde a ningún sonido
-----------------	---

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/galeria.h

3.9. Referencia de la Clase Imagen

Clase que representa una imagen cargada en memoria, lista para ser dibujada en la pantalla.

```
#include <imagen.h>
```

Clases

- struct **tagBitmapFileHeader**
- struct **tagBitmapInfoHeader**

Métodos públicos

- Imagen (void)
- void cargarBmp (const std::string &ruta) throw (ArchivoEx, ImagenEx, TarjetaEx)
- bool dibujar (s16 x, s16 y, s16 z) const
- u16 ancho (void) const
- u16 alto (void) const
- GXTexObj * textura (void) const
- ~Imagen (void)

Atributos públicos estáticos

- static u32 alpha

Métodos protegidos

- Imagen (const Imagen &i)
- Imagen & operator= (const Imagen &i)

3.9.1. Descripción detallada

Clase que representa una imagen cargada en memoria, lista para ser dibujada en la pantalla.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Se puede decir que esta clase Imagen es una abstracción de todos los mecanismos del sistema de la consola Nintendo Wii que es necesario conocer para dibujar una textura en la pantalla. Su sencillo uso hace que, en lugar de complicadas funciones de dibujo, rectángulos, píxeles y demás, únicamente se deba tener en cuenta un archivo de mapa de bits, y unas coordenadas (x,y,z).

Cada instancia de esta clase representa y gestiona la imagen de un archivo bitmap cargado en la memoria de la consola. Dicho bitmap se carga desde la tarjeta SD dinámicamente, pero existen ciertas limitaciones a la hora de trabajar con una imagen concreta.

Para empezar, las medidas de la imagen (alto y ancho), en píxeles, deben ser, obligatoriamente, múltiplos de 8. Si este detalle no se tiene en cuenta, se produce un error del sistema en tiempo de ejecución. Esta restricción está relacionada con la organización de las texturas en tiles de 4×4 píxeles (ver clase Screen para más información), y el problema de cargar una textura cuyas medidas no sean múltiplos 8 es que, seguramente, se rompan los tiles formados, o que incluso no se pudiera reorganizar la información de píxeles de la textura en forma de tiles.

El máximo tamaño para una textura es de 1024×1024 píxeles, y ello se debe a que ése es el tamaño máximo del buffer FIFO que se encarga de enviar la información de un frame al procesador gráfico (consultar la clase Screen). El formato para indicar el color transparente es el mismo que se utiliza en la clase Screen, es decir, un entero de 32 bits, sin signo, y con la forma 0xRRGGBBAA.

Otra limitación importante, aunque esta ya no es de la consola si no de la propia biblioteca, es que, de momento, sólo se pueden cargar imágenes en formato bitmap (BMP), de 24 bits de color directo, y 8 bits por cada componente de color. El por qué de esta restricción es exclusivamente el tiempo que habría que dedicar a desarrollar métodos de carga para cualquier formato de BMP (algunos formatos de bitmap trabajan con color indirecto y paletas de color), JPG o PNG (como ejemplos más comunes). De todas formas, en sucesivas versiones de la clase se irán desarrollando estas funcionalidades para dotarla de mayor potencia.

El formato de color para un píxel que utiliza de forma nativa la Nintendo Wii es RGB5A3. Este formato, de 16 bits, consiste en que cada componente del píxel se representa con 5 bits, y el último bit es el canal alpha (la transparencia). Esto sería, sobre el papel, formato RGB5A1 (y de hecho, generalmente se comporta de la misma manera), pero la

diferencia está en que, si el canal alpha está activado, la consola trabajará como si cada componente del píxel ocupara 4 bits (es decir, 4 bits para rojo, 4 bits para verde, 4 bits para azul, 3 bits para el canal alpha, y 1 bit para indicar que éste está activo). Los tres bits ‘extra’ del canal alpha se toman, cada uno, de una componente de color.

Por último, hay que tener en cuenta que la Wii trabaja en Big Endian, y que por lo tanto, hay que tratar todo aquello que se lea desde un dispositivo para que el Endian no ocasione conflictos. Para más información, consultar el módulo de utilidades.

Funcionamiento interno de la clase Imagen

La clase Imagen se compone de cuatro atributos internos, y uno estático y público. Éste último es el color que se considerará como color transparente, y es compartido por todas las instancias de la clase. Los atributos internos son dos enteros de 16 bits sin signo, que indican el ancho y el alto de la imagen en píxeles, una dirección que apunta hacia la memoria donde se almacena la información de color de los píxeles que componen la imagen, y la dirección de un objeto GXTexObj, que es el atributo que realmente se utilizará para dibujar en la pantalla.

El proceso de cargar una imagen es sencillo. En primer lugar, se crea una instancia de la clase, y después se llama al método cargarBMP(), pasándole como parámetro un std::string que indique la ruta absoluta hasta el archivo de imagen a cargar (debe estar en la SD). Cuando se vaya a implementar el soporte para otro formato de imagen, habrá que definir un método similar, pero específico para el formato nuevo, como por ejemplo cargarPNG() o cargarJPG().

El método de carga de bitmaps comprueba primero que la tarjeta SD esté montada y operativa (ver clase Sdcard). A continuación, abre el archivo que se le ha indicado en la ruta, y trata de leer los 14 primeros bytes del archivo (que corresponden con la cabecera de archivo de todo bitmap). Se comprueba, mediante el campo de tipo de la cabecera, que el archivo es efectivamente un BMP, en caso contrario, se cierra el flujo de entrada y se lanza una excepción. Si todo va bien, se procede a leer los siguientes 40 bytes, que son la cabecera de la propia imagen (que no del archivo, que es la otra). Se toman de ésta los valores de ancho y alto en píxeles de la imagen, y la profundidad de color. Como ya se ha comentado antes, si el ancho y el alto no son múltiplos de ocho, no se puede continuar, y por tanto, se lanza una excepción.

El dato de la profundidad de color es importante, ya que permite diferenciar entre los distintos formatos de BMP que existen. Ya se ha comentado que, de momento, sólo se da soporte a los bitmaps de 24 bits de color, así que, si el dato es correcto, se reserva la memoria alineada y se llama al método correspondiente; en caso contrario, se lanza una excepción.

El método cargarBmp24() se encarga de la tarea específica de leer desde la entrada un bitmap de 24 bits. Si se quisiera ampliar la clase, dotándola de la posibilidad de leer otros formatos de bitmap, únicamente habría que añadir la comprobación en el método cargarBmp(), y crear el método específico de carga del formato.

Un bitmap de 24 bits de color, con 8 bits de color directo por componente, se carga leyendo la información de la imagen de 3 bytes en 3 bytes. Posteriormente, se trabaja a

nivel de bit (con operadores de desplazamiento) para transformar el formato de píxel del BMP (BGR) en RGB5A3, y se van guardando, en orden de llegada, los píxeles con este último formato en la zona de memoria reservada.

Por último, se fijan los datos en la memoria desde la caché, y se crea el objeto de textura haciendo uso de la función que proporciona la clase Screen. Dibujar una imagen es algo trivial, ya que se recurre a los métodos dibujarTextura() o dibujarCuadro(), también de la clase Screen. Finalmente, mencionar que el destructor de la clase se encarga de liberar la memoria ocupada por la textura y la información de los píxeles.

Un sencillo ejemplo de uso:

```
// Crear una instancia de Imagen
Imagen i;
// Cargar un bitmap
i.cargarBMP( "/apps/wiipang/media/imagen.bmp" );
// Dibujar la textura en unas coordenadas (x, y, z)
i.dibujar( 100, 200, 15 );
```

3.9.2. Documentación del constructor y destructor

3.9.2.1. **Imagen::Imagen (void)**

Constructor de la clase Imagen. El objeto creado tiene todos sus atributos a cero.

3.9.2.2. **Imagen::~Imagen (void)**

Destructor de la clase Imagen. Libera la memoria ocupada por la textura y la información de los píxeles.

3.9.2.3. **Imagen::Imagen (const Imagen & i) [protected]**

Constructor de copia de la clase Imagen. Se encuentra en la zona protegida para no permitir la copia.

3.9.3. Documentación de las funciones miembro

3.9.3.1. **u16 Imagen::alto (void) const**

Método observador para el alto en píxeles de la imagen.

Devuelve

Número de píxeles de alto de la imagen.

3.9.3.2. u16 Imagen::ancho (void) const

Método observador para el ancho en píxeles de la imagen.

Devuelve

Número de píxeles de ancho de la imagen.

3.9.3.3. void Imagen::cargarBmp (const std::string & ruta) throw (ArchivoEx, ImagenEx, TarjetaEx)

Función para cargar una imagen desde un archivo bitmap que debe estar en la tarjeta SD. De momento, sólo se da soporte a los bitmaps de 24 bits de color directo (sin paleta).

Parámetros

<i>ruta</i>	Ruta absoluta hasta el fichero que contiene la imagen.
-------------	--

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir el archivo al que apunta la ruta aportada
<i>ImagenEx</i>	Se lanza si sucede un error relacionado con la propia imagen
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD

3.9.3.4. bool Imagen::dibujar (s16 x, s16 y, s16 z) const

Dibuja la imagen en la pantalla en unas coordenadas (x,y,z).

Parámetros

<i>x</i>	Coordenada X del punto superior izquierdo del lugar donde se quiere dibujar la imagen.
<i>y</i>	Coordenada Y del punto superior izquierdo del lugar donde se quiere dibujar la imagen.
<i>z</i>	Coordenada Z (capa) en la que se dibujará la imagen. Entre 0 y 999.

Devuelve

Verdadero si se ha dibujado la imagen, y falso en caso contrario.

3.9.3.5. Imagen& Imagen::operator= (const Imagen & i) [protected]

Operador de asignación de la clase Imagen. Se encuentra en la zona protegida para no permitir la asignación.

3.9.3.6. GXTexObj* Imagen::textura(void) const

Método observador para la textura de la imagen.

Devuelve

Objeto de textura de la imagen.

3.9.4. Documentación de los datos miembro**3.9.4.1. u32 Imagen::alpha [static]**

Variable de clase que determina el color considerado transparente, y que por tanto será invisible.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/imagen.h

3.10. Referencia de la Clase Juego

Clase que proporciona una base para definir el objeto principal del programa de manera fácil.

```
#include <juego.h>
```

Métodos públicos

- Juego (const std::string &ruta)
- virtual ~Juego (void)
- virtual void run (void)

Tipos protegidos

- typedef std::map< std::string, Mando * > Controles

Métodos protegidos

- virtual void cargar (void)=0
- virtual bool frame (void)=0

Atributos protegidos

- Controles _mandos
- u8 _fps

3.10.1. Descripción detallada

Clase que proporciona una base para definir el objeto principal del programa de manera fácil.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Esta clase abstracta proporciona una plantilla sobre la que construir el objeto principal de toda aplicación desarrollada con LibWiiEsp. Consiste en dos apartados bien diferenciados, que son la inicialización de todos los subsistemas de la consola y de la biblioteca (esto se realiza desde el constructor), y la ejecución del bucle principal de la aplicación. Como clase abstracta que es, no se puede instanciar directamente, si no que hay que crear a partir de ella una clase derivada en la que se definan los métodos que el programador considere oportunos para gestionar el programa.

Además de la inicialización de la consola y el control del bucle principal del programa, esta clase también se encarga de gestionar la entrada de hasta cuatro mandos en la consola. Para ello, dispone de un diccionario (de tipo std::map) en el que cada jugador, que está identificado por un código único (de tipo std::string), tiene asociado un mando concreto de la consola. Para acceder al mando de un jugador, basta con buscar en el diccionario mediante el código identificativo del éste.

El constructor de la clase recibe como parámetro un archivo XML con un formato concreto, en el que se deben especificar las opciones de configuración para el programa, como son el nivel de logging deseado y la ruta completa del archivo de log que se generará en el caso de estar el sistema de log activado, el color considerado transparente y que no se dibujará en la pantalla (en formato 0xRRGGBBAA, ver documentación de la clase Screen para más información), el número de fotogramas por segundo (FPS) que se quiere que tenga el videojuego, la ruta absoluta hasta el archivo XML donde se especifican los recursos multimedia que deben ser cargados en la galería de medias (más información en la descripción de la clase Galeria), la ruta absoluta hasta el archivo XML donde se especifican las etiquetas de texto del sistema de idiomas y el nombre del idioma por defecto (más detalles en la descripción de la clase Lang), y por último, la configuración de jugadores, consistente en cuatro atributos de tipo cadena de caracteres en los que se deben especificar los códigos identificadores para cada uno de los jugadores.

A continuación, se muestra un ejemplo del archivo de configuración esperado:

```
<?xml version="1.0" encoding="UTF-8"?>
<conf>
    <log valor="/apps/wiipang/info.log" nivel="3" />
    <alpha valor="0xFF00FFFF" />
    <fps valor="25" />
    <galeria valor="/apps/wiipang/xml/galeria.xml" />
```

```
<lang valor="/apps/wiipang/xml/lang.xml" defecto="english" />
<jugadores pj1="pj1" pj2="pj2" pj3="" pj4="" />
</conf>
```

El constructor de la clase Juego, que debe ser llamado en el constructor de la correspondiente clase derivada, se encarga en primer lugar de montar la primera partición de la tarjeta SD (debe ser una partición con sistema de ficheros FAT), cargar el árbol XML del archivo de configuración en memoria mediante la clase Parser, e inicializar el sistema de logging según se haya especificado en el archivo de configuración. Si en alguna de estas operaciones se produjera un error, se saldría del programa mediante una llamada a la función exit().

A partir de ese momento, el constructor inicializará los sistemas de vídeo (clase Screen), controles (clase Mando), sonido (clase Sonido) y la biblioteca FreeType de gestión de fuentes (clase Fuente). A continuación, establece el color transparente del sistema y el número de FPS, después de lo cual, leerá la configuración de los jugadores, y creará una instancia de Mando para cada jugador que, en el archivo de configuración, no tenga una cadena vacía como identificador. Posteriormente, guarda cada mando asociado al código del jugador correspondiente en el diccionario de Controles.

Por último, se cargan todos los recursos media en la Galería, y las etiquetas de texto para los idiomas del sistema en la clase Lang.

Después de haber inicializado la consola partiendo del archivo de configuración, el método virtual run() proporciona una implementación básica del bucle principal del programa, en la que se controlan las posibles excepciones que puedan lanzarse (las cuales capture y almacena su mensaje en el archivo de log), se mantiene constante la tasa de fotogramas por segundo, y se actualizan todos los mandos conectados a la consola y se gestiona correctamente la actualización de los gráficos a cada fotograma. Junto a este método run(), también se proporciona un método virtual puro, llamado cargar(), y que se ejecuta antes de entrar en el bucle principal. Este método permite realizar las operaciones que se estimen necesarias antes de comenzar la ejecución del bucle, en el caso de que fueran necesarias (en caso contrario, bastaría con definir el método como una función vacía a la hora de derivar la clase Juego).

Igualmente, al ser el método run() virtual, si el programador necesita otro tipo de gestión para el bucle principal de la aplicación, basta con que lo redefina en la clase derivada; a pesar de ello, tendrá disponible un sencillo ejemplo de control del bucle de la aplicación en la definición del método en la clase Juego.

Por último, especificar un detalle, y es que el destructor de la clase Juego se encarga de liberar la memoria ocupada por los objetos de la clase Mando, de tal manera que no hay que preocuparse por ello.

3.10.2. Documentación de los 'Typedef' miembros de la clase

3.10.2.1. `typedef std::map<std::string, Mando*> Juego::Controles` [protected]

Estructura para almacenar los mandos conectados a la consola, asociado cada uno al código de identificación del jugador al que corresponde el mando. Si en el XML de configuración aparece `pj1=jugador1`", entonces se creará un mando para el primer jugador con el código "jugador1". Si en el XML, por ejemplo, aparece `pj4=""`, entonces no se crea mando para el cuarto jugador.

3.10.3. Documentación del constructor y destructor

3.10.3.1. `Juego::Juego (const std::string & ruta)`

Constructor de la clase abstracta Juego. Carga un archivo XML con la configuración que se quiere aplicar al programa. El archivo debe estar situado en la tarjeta SD. Después de leer el archivo XML, inicializa la consola Nintendo Wii según los parámetros establecidos en la configuración.

Parámetros

<code>ruta</code>	Ruta absoluta hasta el archivo de configuración.
-------------------	--

3.10.3.2. `virtual Juego::~Juego (void)` [virtual]

Destructor virtual de la clase abstracta Juego. Llama a la función `exit(0)` después de liberar la memoria ocupada. Esto es obligado para que se disparen las funciones de la pila `atexit()`, y que de no ser contemplado, produciría un error en la consola.

3.10.4. Documentación de las funciones miembro

3.10.4.1. `virtual void Juego::cargar (void)` [protected, pure virtual]

Método virtual puro en el que se debe implementar toda la lógica que se necesite ejecutar antes de que comience el bucle principal de la aplicación.

3.10.4.2. `virtual bool Juego::frame (void)` [protected, pure virtual]

Método virtual puro en el que se debe implementar toda la lógica que se necesite ejecutar durante el bucle principal de la aplicación.

3.10.4.3. `virtual void Juego::run (void)` [virtual]

Método que contiene el bucle principal del juego. Establece el control de tiempo para mantener los FPS constantes (según el valor leído desde el archivo de configuración), y se

encarga de actualizar los mandos conectados a la consola a cada iteración. Toda la lógica del programa que deba ir en este bucle principal se debe implementar en el método virtual frame().

3.10.5. Documentación de los datos miembro

3.10.5.1. u8 Juego::_fps [protected]

Tasa de fotogramas por segundo que tendrá la aplicación.

3.10.5.2. Controles Juego::_mandos [protected]

Estructura que almacena los mandos conectados a la consola.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/juego.h

3.11. Referencia de la Clase Lang

Soporte para múltiples idiomas en la misma aplicación, basado en un archivo de idiomas en formato XML.

```
#include <lang.h>
```

Métodos públicos

- void inicializar (const std::string &ruta, const std::wstring &idioma_activo) throw (ArchivoEx, CódigoEx, TarjetaEx, XmlEx)
- void activarIdioma (const std::wstring &idioma) throw (CódigoEx)
- const std::wstring & idiomaActivo (void) const
- bool existeIdioma (const std::wstring &idioma) const
- void cambiarIdioma (void)
- const std::wstring & texto (const std::string &tag) throw (CódigoEx)

Métodos públicos estáticos

- static Lang * get_instance (void)
- static void destroy (void)

Métodos protegidos

- Lang (void)
- ~Lang (void)

- Lang (const Lang &l)
- Lang & operator= (const Lang &l)

3.11.1. Descripción detallada

Soporte para múltiples idiomas en la misma aplicación, basado en un archivo de idiomas en formato XML.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Esta clase proporciona soporte de idiomas en cualquier aplicación que se desarrolle para Nintendo Wii, pero también es reutilizable en cualquier aplicación de C++, ya que no tiene dependencia del sistema de la consola. Implementa un patrón Singleton, de tal manera que el soporte de idiomas está disponible en cualquier momento y en cualquier parte del sistema.

La idea sobre la que se trabaja es que cada vez que se quiera escribir un texto en el programa, en lugar de ponerlo directamente desde el código fuente, se asignará una etiqueta de texto. Dicha etiqueta de texto será sustituida, en tiempo de ejecución, por la cadena de caracteres asociada a ella, en el idioma que esté marcado como activo en ese momento en la clase Lang. Así, el texto que se muestra en una etiqueta determinada cambiará dinámicamente según se active un idioma u otro.

Los idiomas, las etiquetas y el contenido de cada una de éstas en cada uno de los idiomas se especifican desde un archivo de datos XML, con un formato concreto. A continuación se muestra un ejemplo de XML válido para el soporte de idiomas:

```
<?xml version="1.0" encoding="UTF-8"?>
<lang>
    <idioma nombre="español">
        <tag nombre="MENU1" valor="Jugar" />
        <tag nombre="MENU2" valor="Puntuación" />
        <tag nombre="MENU3" valor="Salir" />
    </idioma>
    <idioma nombre="english">
        <tag nombre="MENU1" valor="Play" />
        <tag nombre="MENU2" valor="Ranking" />
        <tag nombre="MENU3" valor="Exit" />
    </idioma>
</lang>
```

Se trabaja con cadenas de caracteres de alto nivel que utilizan como base caracteres anchos (en C/C++, el tipo wchar_t, que almacena caracteres de 1, 2 ó 4 bytes); así se consigue que cualquier carácter Unicode esté soportado. Cada idioma viene identificado por su nombre (que es, a su vez, una cadena de caracteres anchos), y consiste en un

diccionario de etiquetas de texto. Una etiqueta de texto tiene dos componentes, que son su código identificativo (una cadena de caracteres de alto nivel) y el texto que se sustituirá en la etiqueta en el idioma correspondiente. Es muy importante que cada etiqueta tenga su correspondencia en cada uno de los idiomas soportados, ya que la clase Lang asume que si una etiqueta aparece en un idioma, aparecerá en todos los demás.

Funcionamiento interno

La estructura de datos diseñada para almacenar un idioma consiste en un diccionario (en C++, un std::map), donde su primera componente es una cadena de caracteres de alto nivel (de tipo std::string) que almacena el nombre de una etiqueta de texto (y mediante la cual se identifica la etiqueta), y su segunda componente es otra cadena de caracteres de alto nivel, pero de caracteres anchos (de tipo std::wstring), en la cual se guarda el texto asociado a la etiqueta traducido al idioma correspondiente.

Cada idioma se almacena en otro diccionario en la segunda componente de éste, identificado por su nombre (cadena de caracteres de tipo std::wstring) en la primera componente del diccionario.

Cuando se inicializa el soporte de idiomas, se le pasa como parámetro una cadena de caracteres con la ruta absoluta en la tarjeta SD del archivo de idiomas anteriormente descrito. Este archivo se carga en memoria y se recorre, creando cada idioma que se encuentre, y guardando cada etiqueta de texto asociada a él.

Al leer el contenido de una etiqueta, ésta se almacena temporalmente en una std::string. Antes de guardarla en el diccionario del idioma correspondiente, se transforma a UTF32, mediante una función definida en la cabecera util.h. La explicación de esta transformación es sencilla: cuando se lee un texto con caracteres anchos (que necesita un espacio mayor que 1 byte), cada carácter ancho se ‘rompe’ en trozos de 1 byte cada uno. Un carácter ASCII normal ocupará el espacio esperado, es decir, 1 byte. Lo que ocurre dentro de la función antes mencionada es que se crea una cadena de caracteres anchos (de tipo wchar_t), y se rellena con el contenido leído desde el archivo después de pasarla por la función de C llamada mbstowcs. Esta función se encarga de detectar los caracteres anchos que están ‘rotos’ en trozos de 1 byte, los une, y devuelve toda la cadena recibida como una cadena de caracteres anchos. Por último, se almacena lo devuelto en una std::wstring, y se introduce en el idioma correspondiente.

Otra funcionalidad que ofrece esta clase consiste en activar un idioma concreto mediante su nombre, conocer cuál está activo en un momento determinado, saber si un idioma concreto existe, y alternar entre todos los idiomas registrados a través de una función que va activando cada uno según se lo encuentre al recorrer el diccionario. Este último método obtiene un iterador al idioma activo actual dentro del diccionario, y lo avanza una posición. Si no se ha llegado al final del diccionario, se activa el idioma encontrado; en caso contrario, se activa el primero del diccionario.

Por último, para obtener el texto asociado a una etiqueta concreta en el idioma activo actual, existe una función que, dado el nombre de la etiqueta mediante un std::string, devuelve el texto asociado a ésta mediante un std::wstring, el cual se puede dibujar en pantalla (si la fuente seleccionada para ello tiene soporte para el juego de caracteres necesario), guardar en un fichero, o utilizar en cualquier operación deseada.

Ejemplo de uso

```
// Inicializar el soporte de idiomas mediante el archivo y el idioma activo por
// defecto (ruso)
lang->inicializar( "/apps/wiipang/xml/lang.xml", "english" );

// Escribir en pantalla el texto de una etiqueta (ver clase Fuente y clase Galer
ia
// Se escribiría en pantalla ' Play '
galeria->fuente( "arial" )->escribir( lang->texto( "MENU1" ), 30, 100, 100, 5, 0
xFF0000FF );

// Cambiar el idioma activo a 'español'
lang->activarIdioma( "español" );

// Escribir la misma etiqueta de texto que antes
// Se escribiría en pantalla ' Jugar '
galeria->fuente( "arial" )->escribir( lang->texto( "MENU1" ), 30, 100, 100, 5, 0
xFF0000FF );
```

3.11.2. Documentación del constructor y destructor

3.11.2.1. Lang::Lang(void) [inline, protected]

Constructor de la clase Lang. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.11.2.2. Lang::~Lang(void) [protected]

Destructor de la clase Lang. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.11.2.3. Lang::Lang(const Lang & I) [protected]

Constructor de copia de la clase Lang. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.11.3. Documentación de las funciones miembro

3.11.3.1. void Lang::activarIdioma (const std::wstring & idioma) throw (CodigoEx)

Activa un idioma para que, a partir de este momento, se utilice en todas las etiquetas del sistema.

Parámetros

<i>idioma</i>	Nombre del idioma que se quiere utilizar
---------------	--

Excepciones

<i>CodigoEx</i>	Se lanza si el nombre del idioma no corresponde a ninguno existente
-----------------	---

3.11.3.2. void Lang::cambiarIdioma (void)

Método que va cambiando el idioma activo de forma circular entre los idiomas disponibles

3.11.3.3. static void Lang::destroy (void) [inline, static]

Función estática que destruye la instancia activa del soporte de idiomas, llamando a su destructor.

3.11.3.4. bool Lang::existeldioma (const std::wstring & idioma) const

Función consultora que indica si existe un idioma concreto en el sistema.

Parámetros

<i>idioma</i>	Nombre del idioma que se desea saber si existe en el sistema.
---------------	---

Devuelve

Verdadero si el idioma existe, o falso en caso contrario.

3.11.3.5. static Lang* Lang::get_instance (void) [inline, static]

Función estática que devuelve la instancia activa del soporte de idiomas en el sistema. En el caso de no haber ninguna instancia, se crea y se devuelve. Implementación del patrón Singleton.

Devuelve

Puntero a la instancia activa del soporte de idiomas en el sistema

3.11.3.6. const std::wstring& Lang::idiomaActivo (void) const

Función para conocer el idioma que actualmente está activo en el sistema.

Devuelve

Nombre del idioma que está activo en el sistema

3.11.3.7. void Lang::inicializar (const std::string & ruta, const std::wstring & idioma_activo) throw (ArchivoEx, CódigoEx, TarjetaEx, XmlEx)

Método para inicializar el soporte de idiomas

Parámetros

<i>ruta</i>	Archivo XML con la información del soporte de idiomas
<i>idioma_activo</i>	Nombre del idioma que estará activo en el sistema desde el principio

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir un archivo
<i>CódigoEx</i>	Se lanza si se intenta activar un idioma que no existe
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD
<i>XmlEx</i>	Se lanza si hay un error relacionado con un árbol XML

3.11.3.8. Lang& Lang::operator= (const Lang & l) [protected]

Operador de asignación de la clase Lang. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.11.3.9. const std::wstring& Lang::texto (const std::string & tag) throw (CódigoEx)

Función que devuelve el texto, en el idioma activo, asociado a la etiqueta que se recibe.

Parámetros

<i>tag</i>	Etiqueta de la cual se quiere obtener el texto asociado en el idioma activo.
------------	--

Devuelve

Texto, en el idioma activo del sistema, asociado con la etiqueta que se recibe

Excepciones

<i>CódigoEx</i>	Se lanza si la etiqueta solicitada no existe en el idioma activo
-----------------	--

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/lang.h

3.12. Referencia de la Clase Logger

Sistema de registro de mensajes de información, aviso o error en un archivo de texto plano en la SD.

```
#include <logger.h>
```

Tipos públicos

- enum nivel_log { **OFF**, **ERROR**, **AVISO**, **INFO** }
- typedef enum Logger::nivel_log Nivel

Métodos públicos

- void inicializar (const std::string &ruta, Nivel nivel=OFF) throw (ArchivoEx, TarjetaEx)
- bool debug (void) const
- void info (const std::string &texto)
- void aviso (const std::string &texto)
- void error (const std::string &texto)

Métodos públicos estáticos

- static Logger * get_instance (void)
- static void destroy (void)

Métodos protegidos

- Logger (void)
- ~Logger (void)
- Logger (const Logger &l)
- Logger & operator= (const Logger &l)

3.12.1. Descripción detallada

Sistema de registro de mensajes de información, aviso o error en un archivo de texto plano en la SD.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Esta clase pretende ser una forma sencilla pero eficaz de registrar los eventos que se deseen durante la ejecución de un programa. Al ejecutarse un programa directamente en la videoconsola, no se dispone de ninguna herramienta que permita conocer el valor de expresiones o variables en tiempo de ejecución. Logger es de ayuda cuando sucede algún comportamiento anómalo o no esperado en la ejecución, ya que permite almacenar cadenas de texto en un búffer, y guardar éste en un archivo situado en la tarjeta SD para poder consultarlo posteriormente.

La clase proporciona cuatro niveles de registro, cada uno de los cuales se centra en un tipo concreto de información. Estos niveles son apagado (no se registra ningún evento) información (aquí se incluye cualquier tipo de información relevante sobre el comportamiento del sistema durante la ejecución), aviso (en este nivel se debería incluir información concreta sobre comportamientos que, sin llegar a provocar un error, no son esperados), y error (nivel de registro para almacenar mensajes sobre errores que pueden provocar la parada de la ejecución).

Logger implementa un patrón Singleton, de tal manera que se tendrá una única instancia fácilmente localizable en el sistema. Además, está pensada para ser combinada con un sistema de gestión de excepciones, ya que todos los eventos e informaciones que se registran en el archivo de log son cadenas de texto de alto nivel (de tipo std::string). Esto último facilita enormemente el poder registrar los mensajes de toda excepción que disponga de un método a la usanza de what() de las excepciones estándar (toda aquella que derive de std::exception), que devuelve una cadena de texto con el mensaje de la excepción.

Se debe tener en cuenta que las operaciones de lectura, y más aún las de escritura, en la tarjeta SD de la consola son muy lentas. Es por este motivo por el que se permite desactivar la función de log, mediante el nivel de registro OFF, cuando el programa no necesite información de depuración. Desactivando el registro de eventos se consigue que, al final del programa (que es cuando se produce la escritura del búffer en el archivo de log), no se sobrecargue el sistema con la operación de salida, y la ejecución pueda terminar en un intervalo de tiempo normal.

Funcionamiento interno

Los niveles de registro están agrupados en una enumeración que, a su vez, se define como tipo público de la clase para facilitar su utilización. El orden es desde el nivel más restrictivo (OFF, no se registra nada) a más amplio (INFO, se registra todo). Un nivel más amplio incluye a los más restrictivos que él, lo cual quiere decir que si, por ejemplo, se activara el nivel AVISO, se registrarían tanto mensajes de aviso como de error. Es por esto que se recomienda, en caso de necesitar representar los niveles como enteros, se utilice el cero para OFF, uno para ERROR, dos para AVISO y tres para INFO, con la intención de respetar la jerarquía de niveles. Por supuesto, la decisión final queda en manos del desarrollador.

Cuando se inicializa el sistema de registro de eventos, se guarda el nivel de log introducido como parámetro, y si éste no es OFF (no registrar nada), se sigue adelante. Se comprueba que la tarjeta SD esté disponible para realizar operaciones de entrada/salida, se abre el archivo de log indicado en el primer parámetro en modo escritura (si no existiera, se crea), y se establece una cadena de caracteres de alto nivel como búffer para los mensajes

de log.

La clase irá registrando eventos en el búffer, cada uno en una línea, y comenzando por las cadenas [INFO] para el nivel de información, [AVISO] para el nivel de aviso, y [ERROR] para una cadena que contenga un mensaje con nivel de registro error. El guardado de eventos se realiza por orden de ejecución, de tal manera que cuanto más adelante se encuentre una línea en el archivo de log, será posterior en su ejecución a los eventos reflejados en líneas anteriores.

Por último, hay que tener en cuenta que el volcado del búffer en el archivo se realiza cuando se destruye la instancia de la clase. La ventaja de esta decisión es que, al realizarse un único guardado, el rendimiento del programa con Logger activado no se ve afectado durante la ejecución, hecho que merece la pena aún sabiendo que, al finalizar la aplicación, habrá que esperar a que finalice la operación de salida.

Ejemplo de uso

```
// Inicializar Logger con nivel de registro de avisos
logger->inicializar( "/apps/wiipang/info.log", Logger::AVISO );
// Registrar un mensaje de error
logger->error( "Esto es un mensaje de error" );
// Registrar un mensaje de aviso
logger->aviso( "Esto es un mensaje de aviso" );
// Registrar un mensaje de información, que no se guardará en el archivo de log
logger->info( "Esto es un mensaje de información, no se guardará" );
```

3.12.2. Documentación de los 'Typedef' miembros de la clase

3.12.2.1. **typedef enum Logger::nivel_log Logger::Nivel**

Niveles de log, desde el más amplio (INFO) hasta el más restrictivo (OFF, no se registra nada)

3.12.3. Documentación de las enumeraciones miembro de la clase

3.12.3.1. **enum Logger::nivel_log**

Niveles de log, desde el más amplio (INFO) hasta el más restrictivo (OFF, no se registra nada)

3.12.4. Documentación del constructor y destructor

3.12.4.1. **Logger::Logger(void) [inline, protected]**

Constructor de la clase Logger. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.12.4.2. `Logger::~Logger(void) [protected]`

Destructor de la clase Logger. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.12.4.3. `Logger::Logger(const Logger & l) [protected]`

Constructor de copia de la clase Logger. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.12.5. Documentación de las funciones miembro**3.12.5.1. `void Logger::aviso(const std::string & texto)`**

Método para introducir un evento de log en la categoría de aviso

Parámetros

<i>texto</i>	Texto que se quiere introducir en el log en la categoría de aviso
--------------	---

3.12.5.2. `bool Logger::debug(void) const`

Método consultor para conocer si está activado el modo debug

Devuelve

Verdadero si el modo debug está activado, y falso si no lo está

3.12.5.3. `static void Logger::destroy(void) [inline, static]`

Función estática que destruye la instancia activa del log, llamando a su destructor.

3.12.5.4. `void Logger::error(const std::string & texto)`

Método para introducir un evento de log en la categoría de error

Parámetros

<i>texto</i>	Texto que se quiere introducir en el log en la categoría de error
--------------	---

3.12.5.5. `static Logger* Logger::get_instance(void) [inline, static]`

Función estática que devuelve la instancia activa del log en el sistema. En el caso de no haber ninguna instancia, se crea y se devuelve. Implementación del patrón Singleton.

Devuelve

Puntero a la instancia activa del log en el sistema

3.12.5.6. void Logger::info (const std::string & texto)

Método para introducir un evento de log en la categoría de información

Parámetros

<i>texto</i>	Texto que se quiere introducir en el log en la categoría de información
--------------	---

3.12.5.7. void Logger::inicializar (const std::string & ruta, Nivel nivel = OFF) throw (ArchivoEx, TarjetaEx)

Método que inicializa la clase, abriendo el archivo que recibe por parámetro. La clase sólo registra logs si el segundo parámetro es verdadero; esto se hace así para poder desactivar el modo debug de una forma fácil.

Parámetros

<i>ruta</i>	Ruta absoluta del archivo de logs en la SD
<i>nivel</i>	Nivel de severidad en el registro de logs

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir un archivo
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD

3.12.5.8. Logger& Logger::operator= (const Logger & l) [protected]

Operador de asignación de la clase Logger. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/logger.h

3.13. Referencia de la Clase Mando

Clase que permite leer información desde un Wii Remote.

```
#include <mando.h>
```

Tipos públicos

- enum boton_str {
 BOTON_A, BOTON_B, BOTON_C, BOTON_Z,
 BOTON_1, BOTON_2, BOTON_MAS, BOTON_MENOS,
 BOTON_HOME, BOTON_ARRIBA, BOTON_ABAJO, BOTON_DERECHA,
 BOTON_IZQUIERDA }
- enum nunchuck_x_str { PALANCA_CENTRO_X, PALANCA_DERECHA,
 PALANCA_IZQUIERDA }
- enum nunchuck_y_str { PALANCA_CENTRO_Y, PALANCA_ARRIBA,
 PALANCA_ABAJO }
- typedef enum Mando::boton_str Boton
- typedef enum Mando::nunchuck_x_str Nunchuck_x
- typedef enum Mando::nunchuck_y_str Nunchuck_y

Métodos públicos

- Mando (void)
- ~Mando (void)
- bool conectado (void) const
- u8 chan (void) const
- void actualizar (void)
- bool pressed (Boton boton) const
- bool released (Boton boton) const
- bool newPressed (Boton boton) const
- s16 punteroX (void) const
- s16 punteroY (void) const
- bool punteroEnPantalla (void) const
- void vibrar (u16 tiempo)
- f32 cabeceo (void) const
- f32 viraje (void) const
- f32 rotacion (void) const
- bool nunConectado (void) const
- Nunchuck_x nunPalancaEstadoX (void) const throw (NunchuckEx)
- Nunchuck_y nunPalancaEstadoY (void) const throw (NunchuckEx)
- u16 nunPalancaValorX (void) const throw (NunchuckEx)
- u16 nunPalancaValorY (void) const throw (NunchuckEx)
- u16 nunPalancaCentroX (void) const throw (NunchuckEx)
- u16 nunPalancaCentroY (void) const throw (NunchuckEx)

Métodos públicos estáticos

- static void leerDatos (void)
- static void inicializar (u16 ancho_pantalla, u16 alto_pantalla)

3.13.1. Descripción detallada

Clase que permite leer información desde un Wii Remote.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Clase que representa al Wii Remote, permitiendo acceder en cada momento al estado de pulsación de los botones, la situación del puntero infrarrojo mediante sus coordenadas, y el estado de la palanca de la expansión Nunchuck.

Funcionamiento interno

La libogc proporciona una estructura (definida como WPADDData), en la cual se plasma toda la información relativa a todos los mandos cuando se llama a la función WPAD_ScanPads(). Importante: Una llamada a WPAD_ScanPads() actualiza la información de todos los mandos, así que hay que tener cuidado de llamarla sólo una vez por frame (a menos que se le quiera dar otro uso distinto al habitual de 'leer la entrada una vez por frame y actualizar el mundo del juego en consecuencia'). En un mismo frame, hay que llamar una vez a WPAD_ScanPads() para actualizar la estructura WPADDData, y después actualizar cada mando por separado (mapeo de botones, estado del puntero infrarrojo, etc.).

Cada mando conectado a la consola se identifica en esta estructura WPADDData con un número denominado chan, que tiene los valores desde 0 a 3, y que es único para cada mando. Importante: La libogc sólo tiene acceso a los mandos que tienen sincronización permanente, con la consola; es decir, que los mandos que se vayan a utilizar no deben estar conectados a la consola en modo invitado (para más información sobre la sincronización permanente y el modo invitado, consultar el manual de la propia consola Wii). Además, podemos saber la extensión que tiene conectada el mando (Nunchuck, mando clásico, o incluso la guitarra del Guitar Hero 3) mediante la función WPAD_Probe(u32 chan, u32* type); se guarda la información en type.

El mapeado de los botones se realiza aparte, haciendo uso de dos valarrays booleanos, uno de los cuales representa las pulsaciones de todos los botones en el momento actual, y el otro, las pulsaciones en el momento anterior. De esta manera, en cualquier frame podemos saber si un botón está siendo pulsado, si se acaba de soltar, o si lo acabamos de pulsar. En realidad, la detección de las pulsaciones se hace llamando a la función WPAD_BButtonsDown(u32 chan), y almacenando el resultado en una variable u32 llamada _pulsado. Dicho resultado es un entero de 32 bits, donde cada bit indica una pulsación de

un botón concreto, y si se compara mediante un operador AND de bits (&) con los valores de cada botón que proporciona libogc, nos dirá si dicho botón ha sido pulsado. Así pues, si para cada botón, comparamos su valor conocido con `_pulsado` y lo almacenamos en su lugar correspondiente en el valarray de pulsaciones, tendremos un mapeo completo de todos los botones del mando.

Un detalle más, muy importante, y es que cada vez que se crea una instancia de la clase, se toma como chan el número actual de mandos activos (es decir, si ya tengo dos mandos, uno con chan 0 y otro con chan 1, la nueva instancia tendrá, sí o sí, el chan 2). Hay que tener cuidado de no crear indiscriminadamente instancias de Mando, porque sólo habrá una instancia en todo el sistema con un chan determinado; y un mando, una vez inicializado con un chan, no puede modificarlo. Es decir, que si se crea una instancia de Mando para controlar el primer mando, hay que utilizar esa instancia en todo el sistema, nada de crear una instancia en otro sitio para intentar controlar el primer mando.

Uso de la clase

Antes de utilizar una instancia de Mando, hay que inicializar la clase, esto se hace con la llamada

```
// Inicializar la clase Mando
Mando::inicializar(u16 ancho, u16 alto);
```

donde los parámetros son el alto y ancho de la pantalla en píxeles.

La mayoría de los métodos que componen la clase Mando no son más que métodos observadores sobre el estado descrito las estructuras de datos internas, y que se describen por sí solos en la documentación que acompaña a la clase.

Métodos de orientación del mando

Hay tres métodos observadores (cabeceo, viraje y rotacion), que devuelven un valor entre 180 y -180. Estos valores representan el ángulo de giro respecto a tres ejes (x, y, z) del mando, suponiendo que el (0,0,0) se da cuando el mando está apuntando perpendicularmente a la pantalla, y con los botones hacia arriba.

Para más información sobre estos tres métodos, consultar su descripción.

Ejemplos de uso:

```
// Actualizar los dos mandos:
Mando::leerDatos();
m1.actualizar();
m2.actualizar();

// Saber las coordenadas en pantalla del puntero infrarrojo del mando 1:
m1.punteroX();
m1.punteroY();

// Saber si el segundo mando ha pulsado el botón A:
if ( m2.newPressed( Mando::BOTON_A ) )
// ... Hacer algo

// Hacer vibrar el primer mando durante 300 milisegundos:
```

```
m1.vibrar( 300000 );  
  
// ¿Tiene el Nunchuck del segundo mando la palanca pulsada hacia la derecha?  
if ( m2.nunPalancaEstadoX() == Mando::PALANCA_DERECHA )  
// ... Hacer algo
```

3.13.2. Documentación de los 'Typedef' miembros de la clase

3.13.2.1. **typedef enum Mando::boton_str Mando::Boton**

Enumeración en la que cada elemento representa uno de los botones mapeados del Wii Remote

3.13.2.2. **typedef enum Mando::nunchuck_x_str Mando::Nunchuck_x**

Enumeración que representa los estados horizontales de la palanca del Nunchuck

3.13.2.3. **typedef enum Mando::nunchuck_y_str Mando::Nunchuck_y**

Enumeración que representa los estados verticales de la palanca del Nunchuck

3.13.3. Documentación de las enumeraciones miembro de la clase

3.13.3.1. **enum Mando::boton_str**

Enumeración en la que cada elemento representa uno de los botones mapeados del Wii Remote

3.13.3.2. **enum Mando::nunchuck_x_str**

Enumeración que representa los estados horizontales de la palanca del Nunchuck

3.13.3.3. **enum Mando::nunchuck_y_str**

Enumeración que representa los estados verticales de la palanca del Nunchuck

3.13.4. Documentación del constructor y destructor

3.13.4.1. **Mando::Mando (void)**

Constructor predeterminado de la clase Mando.

3.13.4.2. Mando::~Mando (void)

Destructor de la clase Mando.

3.13.5. Documentación de las funciones miembro**3.13.5.1. void Mando::actualizar (void)**

Actualiza el estado del Wii Remote. Mapea los botones pulsados (conservando el estado anterior, para detectar nuevas pulsaciones, o botones soltados), los acelerómetros y el puntero infrarrojo.

3.13.5.2. f32 Mando::cabeceo (void) const

Valor del cabeceo del Wii Remote. El cabeceo del mando es el ángulo que forma éste con un plano horizontal perpendicular a la pantalla, si se le hace girar sobre el eje X. El ángulo es positivo si la punta del mando queda por encima de la parte trasera, y negativo en caso contrario.

Devuelve

Valor del ángulo de cabeceo del Wii Remote.

3.13.5.3. u8 Mando::chan (void) const

Método observador para conocer el mando concreto (de 0 a 3) con el que se está trabajando.

Devuelve

Número de mando (de 0 a 3) con el que se está trabajando en la instancia.

3.13.5.4. bool Mando::conectado (void) const

Método observador para saber si el Wii Remote está conectado o no.

Devuelve

True si el mando está conectado, o False si no lo está

3.13.5.5. static void Mando::inicializar (u16 ancho_pantalla, u16 alto_pantalla) [inline, static]

Método de clase para inicializar el sistema de bluetooth de los mandos.

Parámetros

<i>ancho_pantalla</i>	Ancho en píxeles de la pantalla
<i>alto_pantalla</i>	Alto en píxeles de la pantalla

3.13.5.6. static void Mando::leerDatos (void) [static]

Método de clase para leer los datos de estado de todos los Wii Remotes conectados. Debe llamarse una única vez por cada frame.

3.13.5.7. bool Mando::newPressed (Boton *boton*) const

Método observador para saber si un botón ha sido pulsado en un momento determinado.

Parámetros

<i>boton</i>	Botón que se desea saber si ha sido pulsado o no.
--------------	---

Devuelve

Devuelve True si el botón se acaba de pulsar, o False en caso contrario.

3.13.5.8. bool Mando::nunConectado (void) const

Método observador que indica si el Nunchuck está conectado al Wii Remote.

Devuelve

True si el Nunchuck está conectado, o False si no lo está.

3.13.5.9. u16 Mando::nunPalancaCentroX (void) const throw (NunchuckEx)

Método observador para el valor central horizontal de la palanca del Nunchuck. Este valor es constante, pero la palanca suelta (supuestamente centrada) no siempre devuelve el mismo valor, por lo que es recomendable dar un margen de +/- 15 a la hora de utilizar este valor.

Devuelve

Valor central de la coordenada X de la palanca del Nunchuck.

Excepciones

<i>NunchuckEx</i>	Se lanza si el Nunchuck no está conectado al WiiRemote
-------------------	--

3.13.5.10. u16 Mando::nunPalancaCentroY (void) const throw (NunchuckEx)

Método observador para el valor central vertical de la palanca del Nunchuck. Este valor es constante, pero la palanca suelta (supuestamente centrada) no siempre devuelve el mismo valor, por lo que es recomendable dar un margen de +/- 15 a la hora de utilizar este valor.

Devuelve

Valor central de la coordenada Y de la palanca del Nunchuck.

Excepciones

<i>NunchuckEx</i>	Se lanza si el Nunchuck no está conectado al WiiRemote
-------------------	--

3.13.5.11. Nunchuck_x Mando::nunPalancaEstadoX (void) const throw (NunchuckEx)

Método observador para conocer el estado básico horizontal de la palanca del Nunchuck. Este método devuelve PALANCA_CENTRADA, PALANCA_IZQUIERDA o PALANCA_DERECHA, dependiendo del estado de pulsación horizontal de la palanca. Si se necesita un control más exacto de los valores de la coordenada X de la palanca del Nunchuck, es conveniente utilizar los métodos 'nunPalancaValorX' y 'nunPalancaCentroX'

Devuelve

Estado básico horizontal de la palanca del Nunchuck

Excepciones

<i>NunchuckEx</i>	Se lanza si el Nunchuck no está conectado al WiiRemote
-------------------	--

3.13.5.12. Nunchuck_y Mando::nunPalancaEstadoY (void) const throw (NunchuckEx)

Método observador para conocer el estado básico vertical de la palanca del Nunchuck. Este método devuelve PALANCA_CENTRADA, PALANCA_ARRIBA o PALANCA_ABAJO, dependiendo del estado de pulsación vertical de la palanca. Si se necesita un control más exacto de los valores de la coordenada Y de la palanca del Nunchuck, es conveniente utilizar los métodos 'nunPalancaValorY' y 'nunPalancaCentroY'

Devuelve

Estado básico vertical de la palanca del Nunchuck

Excepciones

<i>NunchuckEx</i>	Se lanza si el Nunchuck no está conectado al WiiRemote
-------------------	--

3.13.5.13. u16 Mando::nunPalancaValorX (void) const throw (NunchuckEx)

Método observador para el valor de la coordenada horizontal de la palanca del Nunchuck. Si este valor es superior al valor central, se considera que la palanca está siendo pulsada hacia la derecha, en caso contrario, está siendo pulsada hacia la izquierda.

Devuelve

Valor de la coordenada horizontal de la palanca del Nunchuck

Excepciones

<i>NunchuckEx</i>	Se lanza si el Nunchuck no está conectado al WiiRemote
-------------------	--

3.13.5.14. u16 Mando::nunPalancaValorY (void) const throw (NunchuckEx)

Método observador para el valor de la coordenada vertical de la palanca del Nunchuck. Si este valor es superior al valor central, se considera que la palanca está siendo pulsada hacia abajo, en caso contrario, está siendo pulsada hacia arriba.

Devuelve

Valor de la coordenada vertical de la palanca del Nunchuck

Excepciones

<i>NunchuckEx</i>	Se lanza si el Nunchuck no está conectado al WiiRemote
-------------------	--

3.13.5.15. bool Mando::pressed (Boton *boton*) const

Método observador para saber si un botón está pulsado en un momento determinado.

Parámetros

<i>boton</i>	Botón que se desea saber si está pulsado o no.
--------------	--

Devuelve

Devuelve True si el botón está pulsado, o False en caso contrario.

3.13.5.16. bool Mando::punteroEnPantalla (void) const

Método observador que permite conocer si el puntero infrarrojo del Wii Remote se encuentra dentro de los límites de la pantalla.

Devuelve

True si el puntero infrarrojo está dentro de la pantalla; en caso contrario, False

3.13.5.17. s16 Mando::punteroX (void) const

Método observador para la coordenada X del puntero infrarrojo del Wii Remote. La coordenada X tiene su posición cero en la izquierda de la pantalla, y aumenta a medida que se acerca a la derecha de la pantalla.

Devuelve

Coordenada X, en píxeles, del puntero infrarrojo del Wii Remote

3.13.5.18. s16 Mando::punteroY (void) const

Método observador para la coordenada Y del puntero infrarrojo del Wii Remote. La coordenada Y tiene su posición cero en la parte superior de la pantalla, y aumenta a medida que se acerca a la parte inferior de la pantalla.

Devuelve

Coordenada Y, en píxeles, del puntero infrarrojo del Wii Remote

3.13.5.19. bool Mando::released (Boton botón) const

Método observador para saber si un botón ha sido soltado en un momento determinado.

Parámetros

<i>botón</i>	Botón que se desea saber si ha dejado de estar pulsado o no.
--------------	--

Devuelve

Devuelve True si el botón se ha soltado, o False en caso contrario.

3.13.5.20. f32 Mando::rotacion (void) const

Valor de la rotación del Wii Remote. La rotación del mando es un ángulo que mide el giro de éste sobre el eje Y (como si estuviera dando una vuelta de campana). El ángulo es positivo si la rotación es hacia la derecha, y negativo en caso contrario.

Devuelve

Valor del ángulo de rotación del Wii Remote.

3.13.5.21. void Mando::vibrar (u16 tiempo)

Método que hace vibrar el Wii Remote durante una cantidad concreta de tiempo.

Parámetros

<i>tiempo</i>	Tiempo, en microsegundos, que se quiere hacer vibrar el Wii Remote
---------------	--

3.13.5.22. f32 Mando::viraje (void) const

Valor del viraje del Wii Remote. El viraje del mando es el ángulo que forma éste con un plano vertical perpendicular a la pantalla, si se le hace girar sobre el eje Z. El ángulo es positivo si la punta del mando queda a la derecha de la parte trasera, y negativo en caso contrario.

Devuelve

Valor del ángulo de viraje del Wii Remote.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/mando.h

3.14. Referencia de la Clase Musica

Clase que representa una pista de música que puede ser reproducida en la consola Nintendo Wii.

```
#include <musica.h>
```

Métodos públicos

- Musica (const std::string &ruta, u8 volumen=255) throw (ArchivoEx, TarjetaEx)
- ~Musica (void)
- void play (void) const
- void stop (void) const
- void loop (void) const
- bool reproduciendo (void) const
- void setVolumen (u8 volumen)

Métodos protegidos

- Musica (const Musica &m)
- Musica & operator= (const Musica &m)

3.14.1. Descripción detallada

Clase que representa una pista de música que puede ser reproducida en la consola Nintendo Wii.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Esta clase, como todas las clases que representan recursos media, proporciona una abstracción de las funciones y estructuras de datos necesarias para utilizar dichos recursos, en este caso, la clase Musica permite cargar y reproducir archivos de audio en formato MP3, y está pensada para las pistas de música. La biblioteca de bajo nivel que utiliza esta clase para poder utilizar dichas pistas es un port de libmad.

Cada instancia de la clase representa una pista de música, independiente de todas las demás, y preparada para ser reproducida en cualquier punto del programa. Se crea partir de un archivo de audio en formato MP3, cargando éste desde la tarjeta SD de la consola. En principio, cualquier formato (frecuencia, bitrate) de MP3 puede ser utilizado con esta clase, pero se recomienda reducir el bitrate a 128 kbps y mantener la frecuencia a 44100 Hz, así como no almacenar demasiadas pistas a la vez en memoria, ya que la memoria de la que dispone la consola es limitada (64 megas), y cada pista se carga tal cual mediante un flujo de entrada desde ficheros (con lo que el consumo de espacio es bastante alto).

En la descripción de la clase Sonido puede obtenerse más información sobre cómo trabaja el subsistema de sonido de la consola. A partir de ahí, se sabe que existe una voz (un flujo de datos que se pasan directamente al procesador de sonido, el DSP) reservada exclusivamente para pistas de música. La implementación de libmad sobre la que se trabaja con la clase Musica hace uso de dicha voz, de tal manera que sólo se puede reproducir una única pista de música a la vez. Por otro lado, es seguro que no se interferirá con la reproducción de un efecto de sonido encapsulado en una instancia de la clase Sonido. Además, al ser un port, no hay que preocuparse de detalles como el Endian al leer, ya que las funciones de libmad están preparadas para encargarse de dichos detalles.

Al reproducir una pista, por defecto sólo se reproduce una vez. De hecho, libmad no da soporte a la reproducción ininterrumpida como tal. Si se quiere mantener una pista en reproducción infinita (hasta que se detenga explícitamente o se salga del programa) hay que indicarlo al utilizar la función play() mediante su parámetro booleano (hay que especificarlo en cada reproducción; por defecto, se toma el valor false), y llamar a loop() en cada iteración del bucle principal (este método se encarga de iniciar la reproducción de nuevo cuando se acabe la pista en reproducción, pero sólo lo hace si se indicó reproducción continua al iniciarla).

Funcionamiento interno

Esta clase consiste únicamente en un flujo de bytes (la propia pista de música), atributos simples (tamaño, indicador de reproducción infinita y volumen) y la interfaz de la libmad.

A la hora de crear una instancia de la clase, lo primero que se hace es comprobar que la tarjeta SD está montada y lista para ser utilizada. En caso de no ser así, se lanza una excepción indicando el error. Posteriormente, se abre el archivo que se recibe como parámetro mediante la ruta absoluta de éste en la tarjeta. Hay otro parámetro opcional, un entero de 8 bits sin signo (u8), que representa el volumen de la pista de audio (siendo 0 el volumen mínimo, y 255 el máximo), y que se puede modificar en tiempo de ejecución mediante el método modificador setVolumen(). El archivo MP3 se abre mediante un flujo de entrada de bytes ifstream, y se lee en bloque sobre una zona de memoria reservada previamente, de tipo puntero entero de 16 bits con signo (s16), alineada a 32 bytes, y convenientemente ajustada a un tamaño múltiplo de 32 bytes también. Si todo va bien, se cierra el flujo de entrada, y queda lista la pista de sonido para ser reproducida.

Para iniciar la reproducción de la pista, basta con llamar al método play() de la instancia, indicando si se quiere reproducir una vez, o de forma ininterrumpida (tal y como se ha explicado previamente). Este método se encarga de parar cualquier reproducción que haya en curso, establecer el volumen de la nueva reproducción y enviar el flujo de bytes que compone la pista de audio a la voz reservada para música, realizándose estas dos últimas tareas a través de la interfaz que proporciona libmad.

En cualquier momento se puede detener la reproducción de una pista de música llamando al método stop(), y también conocer si se está reproduciendo una pista de música o no mediante el método reproduciendo(). Estas dos funciones también trabajan directamente con la interfaz de libmad.

Por último, el destructor se encarga de detener la reproducción y de liberar la memoria ocupada por la pista de música.

Hay que mencionar que esta clase tiene una "limitación" importante, y es que si se está reproduciendo una pista de audio A, y se realiza una llamada al método stop() de una instancia B, la reproducción de A se detendría. Este comportamiento es intencionado, ya que interesa que, si se está reproduciendo una pista al iniciar la reproducción de otra distinta, la primera se detenga para poder iniciar la nueva.

Ejemplo de uso

```
// Crear una instancia a partir de un archivo MP3
Musica rock( "/apps/wiipang/media/rock.mp3", 255 );
// Iniciar la reproducción
rock.play();
// Detener la reproducción, en el caso de que se esté reproduciendo
if ( rock.reproduciendo() )
    rock.stop();
// Iniciar una reproducción infinita
rock.play( true );
// Mantener la reproducción infinita en cada iteración del bucle principal
while( 1 ) {
    // ...
    rock.loop();
    // ...
}
```

3.14.2. Documentación del constructor y destructor

3.14.2.1. **Musica::Musica (const std::string & ruta, u8 volumen = 255) throw (ArchivoEx, TarjetaEx)**

Constructor predeterminado de la clase Musica.

Parámetros

<i>ruta</i>	Ruta absoluta del fichero MP3 desde el que se cargará la pista de música
<i>volumen</i>	Volumen de la pista de sonido

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir el archivo al que apunta la ruta aportada
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD

3.14.2.2. **Musica::~Musica (void)**

Destructor de la clase Musica.

3.14.2.3. **Musica::Musica (const Musica & m) [protected]**

Constructor de copia de la clase Musica. Se encuentra en la zona protegida para no permitir la copia.

3.14.3. Documentación de las funciones miembro

3.14.3.1. **void Musica::loop (void) const**

Método que mantiene la reproducción infinita de la pista de música. Se debe llamar dentro del bucle principal de la aplicación para mantener la pista reproduciéndose de forma continua. Si no se quiere reproducir infinitamente, basta con no llamar a este método.

3.14.3.2. **Musica& Musica::operator= (const Musica & m) [protected]**

Operador de asignación de la clase Musica. Se encuentra en la zona protegida para no permitir la asignación.

3.14.3.3. **void Musica::play (void) const**

Método que reproduce una pista de música, con el volumen prefijado.

3.14.3.4. bool Musica::reproduciendo (void) const

Método que indica si se está reproduciendo la pista actualmente.

Devuelve

Verdadero si se está reproduciendo la pista, o falso en caso contrario.

3.14.3.5. void Musica::setVolumen (u8 volumen)

Función modificadora del volumen de la pista de música.

Parámetros

<i>volumen</i>	Nuevo valor para el volumen de la pista de música (entre 0 y 255)
----------------	---

3.14.3.6. void Musica::stop (void) const

Método que detiene la reproducción de una pista de música.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/musica.h

3.15. Referencia de la Clase Nivel

Clase que proporciona una base para definir niveles (escenarios donde participan actores) de manera fácil.

```
#include <nivel.h>
```

Clases

- struct actor

Estructura para almacenar de forma temporal la información de un actor.

- struct tile

Estructura que almacena los datos de un tile.

Tipos públicos

- enum Capa { **PLATAFORMAS**, **ESCENARIO** }
- typedef struct Nivel::tile Tile

- `typedef std::map< Capa, Tile ** > Escenario`
- `typedef std::vector< Actor * > Actores`
- `typedef std::map< std::string, Actor * > Jugadores`

Métodos públicos

- `Nivel (const std::string &ruta) throw (ArchivoEx, TarjetaEx)`
- `virtual ~Nivel (void)`
- `u32 ancho (void) const`
- `u32 alto (void) const`
- `u32 xScroll (void) const`
- `u32 yScroll (void) const`
- `const Musica & musica (void) const`
- `void moverScroll (u32 x, u32 y)`
- `bool colision (const Actor *a)`
- `bool colisionBordes (const Actor *a)`
- `virtual void actualizarPj (const std::string &jugador, const Mando &m)=0`
- `virtual void actualizarNpj (void)=0`
- `virtual void actualizarEscenario (void)=0`
- `void dibujar (void)`

Tipos protegidos

- `typedef struct Nivel::actor ActorTemp`
- `typedef std::vector< ActorTemp > Temporal`

Métodos protegidos

- `void leerPropiedades (TiXmlElement *propiedades)`
- `void leerEscenario (TiXmlElement *escenario)`
- `void leerPlataformas (TiXmlElement *plataformas)`
- `void leerActores (TiXmlElement *actores)`
- `virtual void cargarActores (void)=0`

Atributos protegidos

- `std::string _imagen_fondo`
- `std::string _imagen_tileset`
- `std::string _musica`
- `u32 _ancho_un_tile`
- `u32 _alto_un_tile`
- `u32 _ancho_tiles`

- u32 _alto_tiles
- u32 _scroll_x
- u32 _scroll_y
- u32 _ancho_nivel
- u32 _alto_nivel
- u16 _filas_tileset
- u16 _columnas_tileset
- Escenario _escenario
- Actores _actores
- Jugadores _jugadores
- Temporal _temporal

3.15.1. Descripción detallada

Clase que proporciona una base para definir niveles (escenarios donde participan actores) de manera fácil.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Un nivel, desde el punto de vista de LibWiiEsp, no es más que un escenario donde los actores participan en el juego. Está formado por tres elementos principales que son los actores, el conjunto de tiles que componen el propio nivel y una imagen de fondo. También tiene asociada una pista de música propia, una imagen desde la que se toman los tiles que se dibujan, y unas coordenadas de scroll para la pantalla, así como un ancho y un alto.

El escenario que presenta un nivel es de forma rectangular, y tiene un ancho y un alto definidos por el número de tiles que lo componen. A pesar de ello, en todo momento están disponibles sus medidas en píxeles. Dentro de este escenario, todos los elementos tienen una pareja de coordenadas que indican la distancia horizontal hacia la derecha respecto al límite izquierdo del nivel (coordenada X), y la distancia vertical hacia abajo respecto al límite superior del nivel (coordenada Y). Así pues, el origen de coordenadas del nivel (el punto (0,0)) es el vértice superior izquierdo del rectángulo que forma el escenario.

En la pantalla sólo puede dibujarse una parte del escenario del nivel, ya que ésta tiene un tamaño determinado por el modo de vídeo. La parte del nivel que se dibuja en la pantalla en un momento determinado es la ventana del nivel, y es un rectángulo con las mismas medidas que la pantalla, y que tiene unas coordenadas respecto al punto (0,0) del escenario. Estas coordenadas son el scroll del nivel, y se puede modificar mediante el método moverScroll().

La imagen de fondo del nivel debe tener el tamaño de la pantalla completa (para una televisión PAL de proporciones 4:3, las medidas en píxeles son 640×528), y será fija durante

todo el transcurso del nivel. La idea de esta imagen de fondo estática es para situar un paisaje que acompañe al nivel, ya que la sensación de avance se produce con los propios tiles que componen la estructura del nivel.

Existen dos tipos de tiles en un nivel, que se distinguen únicamente en que los tiles no atravesables tienen un rectángulo de colisión con el mismo tamaño y posición, y por lo tanto, provocará que un actor que colisione con él pueda reaccionar de una manera; sin embargo, los tiles atravesables no disponen de esta figura de colisión, y por lo tanto tienen únicamente un objetivo decorativo, para dotar de mayor detalle al escenario del nivel, pero no provocarán una colisión cuando un actor los toque. Todos estos tiles se almacenan en la misma estructura (de tipo Escenario), y para comprobar si un actor concreto colisiona con algún elemento del nivel se proporciona el método `colision()` (información útil para saber, por ejemplo en un juego de plataformas, si el actor está cayendo o está sobre una plataforma). Si el programador necesita un mayor detalle en la detección de colisiones entre actores y escenario, siempre puede añadir los métodos y atributos que considere necesarios para ello al crear la clase derivada de Nivel. La detección de colisiones entre un actor y los tiles del escenario está optimizada para que sólo se evalúe la colisión con los tiles sobre los que se encuentre el actor, evitando cálculos innecesarios.

Se proporciona un método virtual puro `actualizarEscenario()` en el que se pueden implementar comportamientos para el escenario (como por ejemplo, plataformas destructibles, o en movimiento, además de cambiar las coordenadas del scroll de la pantalla), y un método que dibuja la ventana del nivel en la pantalla de la consola.

Los actores que participan en un nivel se distinguen entre los actores que son dirigidos por los jugadores, y los actores que controla la propia CPU. Cada uno de ellos se almacena en una estructura distinta, y dispone de un método concreto para actualizarlo. Concretamente, el método para actualizar los actores no jugadores es `actualizarNpj()`, en el que se supone que se debe implementar la actualización de todos los actores controlados por la máquina (la decisión sobre cómo llevar a cabo esta operación queda, por entero, en manos del programador). Igualmente, para actualizar los actores jugadores, se dispone del método `actualizarPj()`, que debe actualizar a un único jugador en cada llamada, y lo hace recibiendo el código identificador único del jugador, y el mando asociado a él.

Hay que destacar especialmente la filosofía de esta clase Nivel junto con la clase Actor. En cada clase derivada de Actor se debe definir el comportamiento de cada actor dependiendo de su estado actual. Sin embargo, las transiciones entre estados de un actor pueden realizarse desde el Nivel en el que el actor está participando, o bien en el propio actor, ya que éste dispone de una referencia constante al Nivel en el que se encuentra, aunque lo que se recomienda es tomar la primera opción, y situar las transiciones entre estados en el método de actualización del Nivel, de tal manera que los actores respondan con un comportamiento u otro, dependiendo de cómo se les haya definido, y siempre según el estado que tengan activado.

Una vez descritos los elementos que conforman el escenario de un nivel, hay que detallar especialmente el proceso de carga de un nivel desde un archivo TMX creado con el editor de mapas de tiles Tiled (<http://www.mapeditor.org/>):

Cuando se crea un nivel, el constructor recibe un archivo TMX que se carga desde la tarjeta SD gracias a la clase Parser. Una vez en memoria, se lee el ancho y alto del nivel en tiles, y el ancho y el alto en píxeles de un único tile (las medidas de un tile deben ser múltiplos de 8, ya que llevarán una imagen asociada, ver documentación de Imagen para más información). A partir de estos datos, se calcula el ancho y alto del escenario del nivel en píxeles.

Después, el proceso de lectura desde el archivo TMX continúa leyendo las propiedades del mapa, que son `imagen_fondo` (que es el código que debe tener la imagen de fondo del nivel en la Galería de medias), `imagen_tileset` (código que la imagen del tileset deberá tener asociado en la Galería de medias del sistema) y `musica` (código identificador de la pista de música en la Galería de medias del sistema). Tras leer las propiedades del mapa de tiles, se procede a leer las tres capas que debe contener el archivo TMX, que son las siguientes:

1. Capa ‘escenario’ (nombre obligatorio): capa de patrones del editor Tiled, debe contener los tiles atravesables.
2. Capa ‘plataformas’ (nombre obligatorio): capa de patrones del editor Tiled, que contiene los tiles no atravesables (los que tendrán figura de colisión asociada)
3. Capa ‘actores’ (nombre obligatorio): capa de objetos del editor Tiled. Cada objeto definido en esta capa debe tener la propiedad `xml` con la dirección absoluta en la tarjeta SD del archivo XML de descripción del actor como valor. Si además, el actor es un actor jugador, se espera que tenga otra propiedad llamada `jugador`, y cuyo valor debe ser el código identificador del jugador. También debe tener el identificador de su tipo en el campo `Tipo`.

Hay que tener en cuenta otro detalle importante, y es que, al llamar al constructor, la información de los actores se guardan en una estructura temporal. Por ese motivo, existe el método virtual puro `cargarActores()`, que ha de ser implementado por el programador al derivar la clase Nivel, y que se debe encargar de recorrer esta estructura temporal, crear cada actor utilizando el constructor de la clase correspondiente al tipo del actor, y almacenarlo en la estructura correspondiente (dependiendo de si es un actor jugador o no jugador). Por último, es conveniente que este método vacíe la estructura temporal para no malgastar memoria.

Esto último es necesario ya que, en la propia clase abstracta Nivel que forma parte de LibWiiEsp no se conoce qué clases derivadas habrá de Actor, y por tanto, se deja en manos del programador implementar la creación de Actores en el nivel. A continuación, se establece el scroll del nivel al valor (0,0), y se da por concluida la carga del nivel. Para conocer todos los detalles sobre la creación de niveles con el editor Tiled, consultar el manual de LibWiiEsp.

Ejemplo de uso

Con todo lo descrito, una vez cargado un nivel a partir de su archivo TMX, ya está listo para empezar a jugar. Un ejemplo de bucle principal utilizando un único nivel sería el siguiente:

```
Mando m();  
NivelPrueba* nivel = new NivelPrueba( "/apps/wiipang/xml/prueba.tmx" );  
bool salir = false;
```

```

while(not salir) {
    // Actualizar el mando del jugador
    Mando::leerDatos();
    m.actualizar();
    // Salir al pulsar HOME
    if(m.newLinePress(Mando::BOTON_HOME))
        salir = true;
    // Actualizar el jugador en base a su mando (cambios de estado)
    nivel->actualizarPj("pj1", m);
    // Actualizar los personajes no jugadores (cambios de estado)
    nivel->actualizarNpj();
    // Actualiza los elementos del nivel que no son actores (scroll)
    nivel->actualizarEscenario();
    // Mantener la reproducción continua de la pista de música del nivel
    nivel->musica().loop();
    // Dibujar el nivel
    nivel->dibujar();
}

```

En este ejemplo, se considera que se ha derivado la clase abstracta Nivel en la clase NivelPrueba, a la que se le han implementado los métodos actualizarPj(), actualizarNpj() y actualizarEscenario(), y que los actores del nivel se vuelcan desde la estructura temporal con la definición del método cargarActores().

3.15.2. Documentación de los 'Typedef' miembros de la clase

3.15.2.1. `typedef std::vector<Actor*> Nivel::Actores`

Vector que almacena todos los actores no jugadores que participan en un nivel.

3.15.2.2. `typedef struct Nivel::actor Nivel::ActorTemp [protected]`

Estructura para almacenar de forma temporal la información de un actor.

Se compone de un identificador del tipo de actor al que pertenece la información, la ruta hasta el archivo XML desde donde se cargan los datos iniciales del actor, las coordenadas (X,Y) del actor respecto al origen de coordenadas del nivel, y una cadena de caracteres que guarda el código identificador del jugador al que representa el actor (si este último atributo es una cadena vacía se asume que el actor no es controlado por ningún jugador).

3.15.2.3. `typedef std::map<Capa, Tile> Nivel::Escenario`**

Diccionario que almacena todos los tiles que componen un nivel.

3.15.2.4. `typedef std::map<std::string, Actor*> Nivel::Jugadores`

Diccionario que almacena los actores jugadores que participan en un nivel, asociando cada jugador con un código identificativo único.

3.15.2.5. `typedef std::vector<ActorTemp> Nivel::Temporal` [protected]

Vector para almacenar de forma temporal la información de todos los actores el nivel. Se cargan los datos en el constructor del nivel. Para crear cada actor como tal hay que definir el método virtual cargarActores(), el cual se deja en manos del programador, y que debe recoger la información almacenada en este vector y crear cada actor utilizando el constructor de la clase derivada de Actor que corresponda al tipo de actor de cada registro temporal.

3.15.2.6. `typedef struct Nivel::tile Nivel::Tile`

Estructura que almacena los datos de un tile.

Se compone de una pareja de coordenadas (X,Y), que especifican la distancia del punto superior izquierdo del tile respecto al origen de coordenadas (0,0) del nivel, un entero de 32 bits que indica el número identificador del tile dentro del mapa de tiles generado con la aplicación Tiled, y un puntero a un rectángulo de colisión asociado al tile (puede ser NULL).

3.15.3. Documentación de las enumeraciones miembro de la clase

3.15.3.1. `enum Nivel::Capa`

Nombres de las distintas capas de Tiles que componen un escenario

3.15.4. Documentación del constructor y destructor

3.15.4.1. `Nivel::Nivel (const std::string & ruta) throw (ArchivoEx, TarjetaEx)`

Constructor de la clase Nivel. Carga un archivo TMX generado con el editor de mapas de tiles Tiled, creando todas las estructuras necesarias del nivel, los actores no jugadores y los actores jugadores.

Parámetros

<i>ruta</i>	Ruta absoluta hasta el archivo TMX generado con Tiled que almacena la información del nivel
-------------	---

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir un archivo
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD

3.15.4.2. `virtual Nivel::~Nivel (void) [virtual]`

Destructor virtual de la clase. Destruye todos los tiles y actores.

3.15.5. Documentación de las funciones miembro

3.15.5.1. **virtual void Nivel::actualizarEscenario (void) [pure virtual]**

Método virtual puro en el que se debe implementar la actualización de todos los aspectos del escenario del nivel que no correspondan a ningún actor, por ejemplo, el scroll, las zonas de escenario destructibles, etc.

3.15.5.2. **virtual void Nivel::actualizarNpj (void) [pure virtual]**

Método virtual puro en el que se debe implementar la actualización de un actor no jugador. En este método se deben gestionar las transiciones entre estados del autómata finito determinado que controla el comportamiento del Actor. En este caso, no se depende de un mando, si no del estado del nivel.

3.15.5.3. **virtual void Nivel::actualizarPj (const std::string & jugador, const Mando & m) [pure virtual]**

Método virtual puro en el que se debe implementar la actualización de un actor jugador en base al estado del mando asociado a él. En este método se deben gestionar las transiciones entre estados del autómata finito determinado que controla el comportamiento del Actor.

Parámetros

<i>jugador</i>	Código identificador único del actor jugador que se quiere actualizar.
<i>m</i>	Mando asociado al actor jugador que se quiere actualizar.

3.15.5.4. **u32 Nivel::alto (void) const**

Método consultor para conocer el alto en píxeles del escenario que compone el nivel.

Devuelve

Alto en píxeles del escenario que compone el nivel.

3.15.5.5. **u32 Nivel::ancho (void) const**

Método consultor para conocer el ancho en píxeles del escenario que compone el nivel.

Devuelve

Ancho en píxeles del escenario que compone el nivel.

3.15.5.6. virtual void Nivel::cargarActores (void) [protected, pure virtual]

Método virtual puro que hay que definir en las clases derivadas, y que se debe encargar de volcar la información de los actores, almacenada en la estructura temporal que se rellena en el constructor del nivel, en las estructuras definitivas de los actores. Para cada actor, debe identificar su tipo y llamar al constructor de la clase adecuada con los parámetros de inicialización que se proporcionan desde esta estructura temporal. Es recomendable que vacíe la estructura temporal cuando finalice. Debe ser llamado en el constructor de la clase derivada de Nivel.

3.15.5.7. bool Nivel::colision (const Actor * a)

Método para conocer si un actor externo colisiona con, al menos, un tile del escenario del nivel.

Parámetros

<i>a</i>	Puntero constante al actor del cual se quiere saber si colisiona con el escenario del nivel.
----------	--

Devuelve

Verdadero si hay colisión, o falso en caso contrario.

3.15.5.8. bool Nivel::colisionBordes (const Actor * a)

Método para conocer si un actor externo colisiona con uno de los límites del nivel.

Parámetros

<i>a</i>	Puntero constante al actor del cual se quiere saber si colisiona con los límites del nivel.
----------	---

Devuelve

Verdadero si hay colisión, o falso en caso contrario.

3.15.5.9. void Nivel::dibujar (void)

Método que dibuja la parte del escenario del nivel que marque las coordenadas del scroll en la pantalla. Las coordenadas de scroll indicarán el punto superior izquierdo de la parte rectangular del escenario del nivel a dibujar. Se dibujan también todos los actores que se encuentren dentro de esta sección.

3.15.5.10. void Nivel::leerActores (TiXmlElement * *actores*) [protected]

Método que, a partir de un elemento de un árbol XML, se encarga de leer todos los actores que participan en el nivel, tanto los jugadores como los no jugadores, y los almacena en la estructura temporal de tipo Temporal.

Parámetros

<i>actores</i>	Elemento XML donde se almacena la información de los actores del nivel.
----------------	---

3.15.5.11. void Nivel::leerEscenario (TiXmlElement * *escenario*) [protected]

Método que, a partir de un elemento de un árbol XML, se encarga de leer los tiles que componen el escenario del nivel. Un tile forma parte del escenario si no tiene asociado ninguna caja de colisión, es decir, si es un tile atravesable.

Parámetros

<i>escenario</i>	Elemento XML donde se almacenan los tiles del escenario del nivel.
------------------	--

3.15.5.12. void Nivel::leerPlataformas (TiXmlElement * *plataformas*) [protected]

Método que, a partir de un elemento de un árbol XML, se encarga de leer los tiles que componen las plataformas del nivel. Un tile forma parte de las plataformas si tiene asociada una caja de colisión, es decir, si es un tile no atravesable.

Parámetros

<i>plataformas</i>	Elemento XML donde se almacenan los tiles de las plataformas del nivel.
--------------------	---

3.15.5.13. void Nivel::leerPropiedades (TiXmlElement * *propiedades*) [protected]

Método que, a partir de un elemento de un árbol XML, se encarga de leer todas las propiedades del nivel.

Parámetros

<i>propiedades</i>	Elemento XML donde se almacenan las propiedades del nivel.
--------------------	--

3.15.5.14. void Nivel::moverScroll (u32 *x*, u32 *y*)

Método que modifica la posición del scroll estableciendo sus nuevas coordenadas. Se toma como origen el punto superior izquierdo del escenario. Si aumenta la X, más a la derecha estará el scroll respecto del punto x = 0; y si aumenta la Y, más abajo estará el

scroll respecto del punto y = 0.

Parámetros

<i>x</i>	Nuevo valor para la coordenada X del scroll.
<i>y</i>	Nuevo valor para la coordenada Y del scroll.

3.15.5.15. **const Musica& Nivel::musica (void) const**

Método consultor para obtener la pista de música del nivel.

Devuelve

Referencia constante a la pista de música del nivel.

3.15.5.16. **u32 Nivel::xScroll (void) const**

Método consultor para saber la posición del scroll horizontal del nivel. El scroll horizontal es un entero de 32 bits sin signo, que indica la distancia, desde el límite izquierdo del escenario y hacia la izquierda, a la que se encuentra la esquina superior izquierda de la parte del nivel que se dibujará en la pantalla de la consola.

Devuelve

Posición del scroll horizontal.

3.15.5.17. **u32 Nivel::yScroll (void) const**

Método consultor para saber la posición del scroll vertical del nivel. El scroll vertical es un entero de 32 bits sin signo, que indica la distancia, desde el límite superior del escenario y hacia abajo, a la que se encuentra la esquina superior izquierda de la parte del nivel que se dibujará en la pantalla de la consola.

Devuelve

Posición del scroll vertical.

3.15.6. Documentación de los datos miembro

3.15.6.1. **Actores Nivel::_actores [protected]**

Estructura que almacena todos los actores no jugadores del nivel.

3.15.6.2. **u32 Nivel::_alto_nivel [protected]**

Alto del nivel medido en píxeles

3.15.6.3. `u32 Nivel::_alto_tiles` [protected]

Alto, medido en tiles, del nivel

3.15.6.4. `u32 Nivel::_alto_un_tile` [protected]

Alto en píxeles de un tile

3.15.6.5. `u32 Nivel::_ancho_nivel` [protected]

Ancho del nivel medido en píxeles

3.15.6.6. `u32 Nivel::_ancho_tiles` [protected]

Ancho, medido en tiles, del nivel

3.15.6.7. `u32 Nivel::_ancho_un_tile` [protected]

Ancho en píxeles de un tile

3.15.6.8. `u16 Nivel::_columnas_tiles` [protected]

Número de columnas de tiles que componen la imagen del tiles.

3.15.6.9. `Escenario Nivel::_escenario` [protected]

Estructura que almacena todos los tiles del nivel, tanto los atravesables como los no atravesables.

3.15.6.10. `u16 Nivel::_filas_tiles` [protected]

Número de filas de tiles que componen la imagen del tiles.

3.15.6.11. `std::string Nivel::_imagen_fondo` [protected]

Código con el que se almacena la imagen de fondo del nivel en la Galería.

3.15.6.12. `std::string Nivel::_imagen_tiles` [protected]

Código con el que se almacena la imagen del tiles del nivel en la Galería.

3.15.6.13. Jugadores Nivel::_jugadores [protected]

Estructura que almacena todos los actores jugadores del nivel.

3.15.6.14. std::string Nivel::_musica [protected]

Código con el que se almacena la pista de música del nivel en la Galería.

3.15.6.15. u32 Nivel::_scroll_x [protected]

Posición del scroll horizontal del nivel.

3.15.6.16. u32 Nivel::_scroll_y [protected]

Posición del scroll vertical del nivel.

3.15.6.17. Temporal Nivel::_temporal [protected]

Estructura que guarda temporalmente todos los actores del nivel. Esto se hace en el constructor, y se hace para que el constructor no deba llamar a una función virtual pura de carga de actores (el compilador daría error). Como solución a este problema, se ha decidido que el constructor carga la información de los actores en esta estructura temporal, y después, en el constructor de la clase derivada de Nivel, se vuelca esta información en las estructuras definitivas para los actores.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/nivel.h

3.16. Referencia de la Clase Parser

Clase que facilita la lectura de información desde archivos XML situados en la tarjeta SD de la consola.

```
#include <parser.h>
```

Métodos públicos

- void cargar (const std::string &ruta) throw (ArchivoEx, TarjetaEx)
- TiXmlElement * raiz (void)
- std::string contenido (const TiXmlElement *el) const
- std::string atributo (const std::string &nombre, const TiXmlElement *el) const
- u32 atributoU32 (const std::string &nombre, const TiXmlElement *el) const
- f32 atributoF32 (const std::string &nombre, const TiXmlElement *el) const

- TiXmlElement * buscar (const std::string &valor, TiXmlElement *el=0)
- TiXmlElement * siguiente (TiXmlElement *el)

Métodos públicos estáticos

- static Parser * get_instance (void)
- static void destroy (void)

Métodos protegidos

- Parser (void)
- ~Parser (void)
- Parser (const Parser &p)
- Parser & operator= (const Parser &p)

3.16.1. Descripción detallada

Clase que facilita la lectura de información desde archivos XML situados en la tarjeta SD de la consola.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

La clase Parser no es más que una interfaz sencilla y directa para la lectura de información desde archivos XML situados en la tarjeta SD de la consola Nintendo Wii. Abstiene las estructuras de datos y funciones de la biblioteca externa TinyXML, de tal manera que no hay que preocuparse por las comprobaciones y posibles errores que puedan surgir en el proceso de lectura. Implementa un patrón Singleton, de tal manera que es accesible desde cualquier punto del programa, y sólo existe una instancia en el sistema.

Comentar que no hay que preocuparse por los detalles de BigEndian, reserva de memoria alineada a 32 bytes y demás, porque la versión de TinyXML que se utiliza esta portada a Nintendo Wii, lo cual significa que ya contempla de base estos asuntos.

Este parser funciona cargando el árbol del documento por completo en memoria, y después realizando búsquedas y lecturas sobre éste, lo cual es una implementación de DOM (Document Object Model, o modelo de objetos del documento en la lengua de Cervantes), que es una interfaz estándar para manipular y acceder a un archivo XML. Al trabajar con árboles, se dispone de un elemento raíz en el documento, a partir del cual van desplegándose elementos hijos, y pueden aplicarse todas las técnicas y algoritmos relativos a árboles. El documento XML, una vez cargado en memoria, se encontrará almacenado dentro de la propia clase hasta que, o bien se destruya ésta, o se cargue un nuevo árbol XML.

Derivado de esto último, es importante tener en cuenta que si se está trabajando con un árbol XML, no se debe cargar otro nuevo hasta haber finalizado con el primero, ya que el cargar un segundo árbol implica la destrucción del existente, y por tanto, para volver a operar sobre el original, es necesario cargarlo otra vez. Sin embargo, al cargar por segunda vez un mismo archivo XML, las variables que apuntaban a elementos del primer árbol quedan apuntando a zonas de memoria liberadas; de ahí el cuidado necesario al trabajar con varios archivos XML.

Funcionamiento interno

La clase únicamente tiene como atributo privado un documento XML de tipo TiXmlDocument, que viene de serie con la biblioteca TinyXML (la definición del tipo, no el atributo). Cuando se realiza una llamada al método cargar, éste recibe como parámetro una cadena de caracteres de alto nivel (de tipo std::string) que indica la ruta absoluta hasta el archivo XML del cual se quiere cargar en memoria su árbol de elementos. Primero comprueba que la tarjeta SD está preparada para trabajar, y después de eso, intenta cargar el árbol completo en el atributo privado que almacenará el documento. Si algo fallara, se lanzaría una excepción con un texto descriptivo sobre el error. Un detalle a mencionar es que la codificación de caracteres con la que se intenta abrir el archivo XML es UTF8.

Una vez se ha abierto correctamente el archivo, y se ha cargado el árbol de elementos en memoria, hay varias posibilidades para extraer información de éste. En primer lugar, se puede acceder al elemento raíz del árbol mediante el método raiz. También, dado un elemento del árbol, se puede extraer el contenido de éste (si lo tuviera) mediante el método contenido, que lo devuelve en forma de std::string. Si no existiera contenido en el elemento, se devuelve una cadena vacía.

Para acceder a los atributos de un elemento concreto, puede hacerse mediante los métodos atributo (que intenta devolver el valor del atributo como una cadena de caracteres de alto nivel; si no existe el atributo o el elemento, devuelve una cadena vacía), atributoU32 y atributoF32 (que funcionan de la misma manera, sólo que devolviendo el valor como un entero de 32 bits sin signo, o un decimal de coma flotante de 32 bits, respectivamente. Si no existieran el elemento o el atributo, se devuelve el valor cero).

La navegación por los elementos del árbol es sencilla, basta con llamar al método buscar con el valor del elemento que se quiere localizar. Se puede especificar otro elemento a partir del cual se comenzará la búsqueda, que se realiza recursivamente en preorden, y devuelve la primera ocurrencia de elemento que coincide con el valor indicado. Si no se determina un elemento como raíz de la búsqueda, se toma el elemento raíz del árbol como inicio. Por último, si no se localizara ningún elemento coincidente en valor con el deseado, se devuelve el valor NULL.

Otra manera de localizar un elemento es, a partir de uno dado, buscar el siguiente elemento hermano con el mismo valor. Se considera que un elemento es hermano de otro si tienen el mismo padre (elemento inmediatamente superior en el árbol). El siguiente hermano con el mismo valor es aquel elemento que, estando entre los hermanos del elemento original, tiene el mismo valor, y está situado a la derecha del primero. El método siguiente intenta localizar el hermano inmediatamente consecutivo con el mismo valor a un elemento dado, y es útil para evitar bucles que recorran todos los hijos de un elemento dado, en el

caso de que se estén buscando sólo los hijos que tengan un valor determinado. Al igual que en la búsqueda, si no se localizara un elemento que coincida con los criterios del método, se devuelve el valor NULL.

Ejemplo de uso

```
// Cargar un archivo XML
parser->cargar( "/apps/wipang/xml/galeria.xml" );
// Buscar el primer elemento con valor 'musica' a partir del elemento raíz
TiXmlElement* musica = parser->buscar( 'musica', parser->raiz() );
// Leer un atributo llamado codigo, con formato cadena de caracteres
std::string ruta = parser->atributo( "codigo", musica );
// Leer un atributo llamado volumen, con formato de entero de 32 bits sin signo
u32 volumen = parser->atributo( "volumen", musica );
// Buscar el primer hermano de 'musica' con el mismo valor ("musica" también)
TiXmlElement* otra_musica = parser->siguiente( musica );
```

3.16.2. Documentación del constructor y destructor

3.16.2.1. Parser::Parser (void) [inline, protected]

Constructor de la clase Parser. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.16.2.2. Parser::~Parser (void) [inline, protected]

Destructor de la clase Parser. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.16.2.3. Parser::Parser (const Parser & p) [protected]

Constructor de copia de la clase Parser. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.16.3. Documentación de las funciones miembro

3.16.3.1. std::string Parser::atributo (const std::string & nombre, const TiXmlElement * el) const

Método que devuelve el texto contenido en un atributo de un elemento XML que se recibe como parámetro. Si el elemento o el atributo no existen, se devuelve una cadena vacía.

Parámetros

<i>nombre</i>	Nombre del atributo cuyo valor se quiere conocer
<i>el</i>	Elemento XML del árbol del cual se quiere conocer el valor de un atributo

Devuelve

Cadena de texto con el contenido del atributo solicitado

3.16.3.2. f32 Parser::atributoF32 (const std::string & *nombre*, const TiXmlElement * *el*) const

Método que devuelve el contenido de un atributo de un elemento XML que se recibe como parámetro, en formato de coma flotante de 32 bits sin signo. Si el elemento o el atributo no existen, se devuelve el valor 0.0.

Parámetros

<i>nombre</i>	Nombre del atributo cuyo valor se quiere conocer
<i>el</i>	Elemento XML del árbol del cual se quiere conocer el valor de un atributo

Devuelve

Número decimal en coma flotante de 32 bits con el contenido del atributo solicitado

3.16.3.3. u32 Parser::atributoU32 (const std::string & *nombre*, const TiXmlElement * *el*) const

Método que devuelve el contenido de un atributo de un elemento XML que se recibe como parámetro, en formato entero de 32 bits sin signo. Si el elemento o el atributo no existen, se devuelve el valor 0.

Parámetros

<i>nombre</i>	Nombre del atributo cuyo valor se quiere conocer
<i>el</i>	Elemento XML del árbol del cual se quiere conocer el valor de un atributo

Devuelve

Entero de 32 bits sin signo con el contenido del atributo solicitado

3.16.3.4. TiXmlElement* Parser::buscar (const std::string & *valor*, TiXmlElement * *el* = 0)

Método que busca un elemento cuyo valor sea el que se recibe por parámetro. La búsqueda es recursiva entre todos los hijos del elemento que se reciba (si no se recibe ninguno, la búsqueda comienza en el elemento raíz del árbol XML). Si no se encuentra el elemento solicitado, se devuelve NULL.

Parámetros

<i>valor</i>	Valor del elemento a buscar
<i>el</i>	Elemento del árbol XML a partir del que se comenzará la búsqueda

Devuelve

Elemento cuyo valor coincide con el introducido por parámetro

3.16.3.5. void Parser::cargar (const std::string & ruta) throw (ArchivoEx, TarjetaEx)

Método que carga en memoria un archivo XML que esté almacenado en la tarjeta SD. El archivo queda listo para ser accedido en cualquier momento, y desde cualquier parte del programa (gracias a la implementación del patrón Singleton).

Parámetros

<i>ruta</i>	Ruta absoluta hasta el archivo XML a cargar, debe estar en la tajeta SD
-------------	---

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir el archivo al que apunta la ruta aportada
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD

3.16.3.6. std::string Parser::contenido (const TiXmlElement * el) const

Método que devuelve el texto contenido en un elemento XML que se recibe como parámetro. Si el elemento no existe, o no tiene contenido, se devuelve una cadena vacía.

Parámetros

<i>el</i>	Elemento XML del árbol del cual se quiere conocer su contenido
-----------	--

Devuelve

Cadena de texto con el contenido del elemento recibido

3.16.3.7. static void Parser::destroy (void) [inline, static]

Función estática que destruye la instancia activa del parser XML, llamando a su destructor.

3.16.3.8. static Parser* Parser::get_instance (void) [inline, static]

Función estática que devuelve la instancia activa del parser XML en el sistema. En el caso de no haber ninguna instancia, se crea y se devuelve. Implementación del patrón Singleton.

Devuelve

Puntero a la instancia activa del parser XML en el sistema

3.16.3.9. Parser& Parser::operator= (const Parser & p) [protected]

Operador de asignación de la clase Parser. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.16.3.10. TiXmlElement* Parser::raiz (void)

Método que devuelve el elemento raíz del árbol XML del archivo que esté cargado en memoria.

Devuelve

Elemento raíz del árbol XML del archivo que está actualmente cargado

3.16.3.11. TiXmlElement* Parser::siguiente (TiXmlElement * el)

Método que devuelve el siguiente elemento hermano (es decir, que se encuentra en la misma profundidad en el árbol XML, y que tiene el mismo elemento padre) con el mismo valor que el recibido por parámetro. Si no se encontrara, se devuelve el valor NULL.

Parámetros

<i>el</i>	Elemento del árbol XML a partir del que se busca el siguiente hermano
-----------	---

Devuelve

Elemento hermano consecutivo al recibido, con el mismo valor que éste.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/parser.h

3.17. Referencia de la Clase Punto

Clase que representa un punto en el sistema de gestión de colisiones.

```
#include <colision.h>
```

Métodos públicos

- Punto (s16 x=0, s16 y=0)

- bool hayColision (Figura *f, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Circulo *c, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Rectangulo *r, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Punto *p, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- s16 & x (void)
- const s16 & x (void) const
- s16 & y (void)
- const s16 & y (void) const

3.17.1. Descripción detallada

Clase que representa un punto en el sistema de gestión de colisiones.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Se representa mediante dos enteros de 16 bits con signo, y deriva de la clase Figura.

3.17.2. Documentación del constructor y destructor

3.17.2.1. Punto::Punto (s16 x = 0, s16 y = 0) [inline]

Constructor de la clase Punto

Parámetros

<i>x</i>	Coordenada X del punto. Por defecto, tiene valor 0.
<i>y</i>	Coordenada Y del punto. Por defecto, tiene valor 0.

3.17.3. Documentación de las funciones miembro

3.17.3.1. bool Punto::hayColision (Figura * f, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [inline, virtual]

Método para saber si una figura de tipo desconocido que se recibe por parámetro colisiona con el punto. Implementación de la técnica de double dispatch.

Parámetros

<i>f</i>	Puntero a una figura con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del punto

<i>dy1</i>	Desplazamiento vertical del punto
<i>dx2</i>	Desplazamiento horizontal de la figura que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical de la figura que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.17.3.2. bool Punto::hayColision (Circulo * *c*, s16 *dx1*, s16 *dy1*, s16 *dx2*, s16 *dy2*) [virtual]

Método para saber si un círculo que se recibe por parámetro colisiona con el punto actual. Implementación de la técnica de double dispatch.

Parámetros

<i>c</i>	Puntero a un círculo con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del punto
<i>dy1</i>	Desplazamiento vertical del punto
<i>dx2</i>	Desplazamiento horizontal del círculo que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del círculo que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.17.3.3. bool Punto::hayColision (Punto * *p*, s16 *dx1*, s16 *dy1*, s16 *dx2*, s16 *dy2*) [inline, virtual]

Método para saber si un punto que se recibe por parámetro colisiona con el punto actual. Implementación de la técnica de double dispatch.

Parámetros

<i>p</i>	Puntero a un punto con el que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del punto
<i>dy1</i>	Desplazamiento vertical del punto
<i>dx2</i>	Desplazamiento horizontal del punto que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del punto que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.17.3.4. `bool Punto::hayColision (Rectangulo * r, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [virtual]`

Método para saber si un rectángulo que se recibe por parámetro colisiona con el punto actual. Implementación de la técnica de double dispatch.

Parámetros

<i>r</i>	Puntero a un rectángulo con el que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del punto
<i>dy1</i>	Desplazamiento vertical del punto
<i>dx2</i>	Desplazamiento horizontal del rectángulo que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del rectángulo que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.17.3.5. `s16& Punto::x (void) [inline]`

Método modificador que devuelve una referencia a la coordenada X del punto.

Devuelve

Referencia a la coordenada X del punto.

3.17.3.6. `const s16& Punto::x (void) const [inline]`

Método consultor que devuelve la coordenada X del punto.

Devuelve

Valor de la coordenada X del punto.

3.17.3.7. `const s16& Punto::y (void) const [inline]`

Método consultor que devuelve la coordenada Y del punto.

Devuelve

Valor de la coordenada Y del punto.

3.17.3.8. s16& Punto::y(void) [inline]

Método modificador que devuelve una referencia a la coordenada Y del punto.

Devuelve

Referencia a la coordenada Y del punto.

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/collision.h

3.18. Referencia de la Clase Rectangulo

Clase que representa un rectángulo en el sistema de gestión de colisiones.

```
#include <collision.h>
```

Métodos públicos

- Rectangulo (Punto p1, Punto p2, Punto p3, Punto p4)
- bool hayColision (Figura *f, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Circulo *c, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Rectangulo *r, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- bool hayColision (Punto *p, s16 dx1, s16 dy1, s16 dx2, s16 dy2)
- Punto & p1 (void)
- const Punto & p1 (void) const
- Punto & p2 (void)
- const Punto & p2 (void) const
- Punto & p3 (void)
- const Punto & p3 (void) const
- Punto & p4 (void)
- const Punto & p4 (void) const
- Punto & centro (void)
- const Punto & centro (void) const

3.18.1. Descripción detallada

Clase que representa un rectángulo en el sistema de gestión de colisiones.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Se representa mediante cuatro puntos de tipo Punto, y deriva de la clase Figura. Se considera que sus lados son paralelos a los ejes, pero se representa con los 4 puntos en lugar de con un punto y el ancho y alto para, en un futuro, permitir rectángulos no alineados con los ejes.

3.18.2. Documentación del constructor y destructor**3.18.2.1. Rectangulo::Rectangulo (Punto *p1*, Punto *p2*, Punto *p3*, Punto *p4*) [inline]**

Constructor de la clase Rectángulo. El orden de los puntos esperado es superior izquierdo (*p1*), superior derecho (*p2*), inferior derecho (*p3*) e inferior izquierdo (*p4*).

Parámetros

<i>p1</i>	Punto superior izquierdo del rectángulo.
<i>p2</i>	Punto superior derecho del rectángulo.
<i>p3</i>	Punto inferior izquierdo del rectángulo.
<i>p4</i>	Punto inferior derecho del rectángulo.

3.18.3. Documentación de las funciones miembro**3.18.3.1. Punto& Rectangulo::centro (void) [inline]**

Método modificador que devuelve el centro del rectángulo

Devuelve

Referencia al centro del rectángulo

3.18.3.2. const Punto& Rectangulo::centro (void) const [inline]

Método consultor que devuelve el centro del rectángulo

Devuelve

Centro del rectángulo

3.18.3.3. bool Rectangulo::hayColision (Circulo * *c*, s16 *dx1*, s16 *dy1*, s16 *dx2*, s16 *dy2*) [virtual]

Método para saber si un círculo que se recibe por parámetro colisiona con el rectángulo. Implementación de la técnica de double dispatch.

Parámetros

<i>c</i>	Puntero a un círculo con el que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del rectángulo
<i>dy1</i>	Desplazamiento vertical del rectángulo
<i>dx2</i>	Desplazamiento horizontal del círculo que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del círculo que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.18.3.4. bool Rectangulo::hayColision (Punto * *p*, s16 *dx1*, s16 *dy1*, s16 *dx2*, s16 *dy2*) [virtual]

Método para saber si un punto que se recibe por parámetro colisiona con el rectángulo. Implementación de la técnica de double dispatch.

Parámetros

<i>p</i>	Puntero a un punto con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del rectángulo
<i>dy1</i>	Desplazamiento vertical del rectángulo
<i>dx2</i>	Desplazamiento horizontal del punto que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del punto que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.18.3.5. bool Rectangulo::hayColision (Rectangulo * *r*, s16 *dx1*, s16 *dy1*, s16 *dx2*, s16 *dy2*) [virtual]

Método para saber si un rectángulo desconocido que se recibe por parámetro colisiona con el rectángulo. Implementación de la técnica de double dispatch.

Parámetros

<i>r</i>	Puntero a un rectángulo con el que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del rectángulo
<i>dy1</i>	Desplazamiento vertical del rectángulo
<i>dx2</i>	Desplazamiento horizontal del rectángulo que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical del rectángulo que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.18.3.6. `bool Rectangulo::hayColision (Figura * f, s16 dx1, s16 dy1, s16 dx2, s16 dy2) [inline, virtual]`

Método para saber si una figura de tipo desconocido que se recibe por parámetro colisiona con el rectángulo. Implementación de la técnica de double dispatch.

Parámetros

<i>f</i>	Puntero a una figura con la que se quiere comprobar si existe colisión
<i>dx1</i>	Desplazamiento horizontal del rectángulo
<i>dy1</i>	Desplazamiento vertical del rectángulo
<i>dx2</i>	Desplazamiento horizontal de la figura que se recibe por parámetro
<i>dy2</i>	Desplazamiento vertical de la figura que se recibe por parámetro

Devuelve

Verdadero si hay colisión entre las figuras, o falso en caso contrario

Implementa Figura.

3.18.3.7. `Punto& Rectangulo::p1 (void) [inline]`

Método modificador que devuelve una referencia al punto superior izquierdo del rectángulo

Devuelve

Referencia al punto superior izquierdo del rectángulo

3.18.3.8. `const Punto& Rectangulo::p1 (void) const [inline]`

Método consultor que devuelve el punto superior izquierdo del rectángulo

Devuelve

Punto superior izquierdo del rectángulo

3.18.3.9. `Punto& Rectangulo::p2 (void) [inline]`

Método modificador que devuelve una referencia al punto superior derecho del rectángulo

Devuelve

Referencia al punto superior derecho del rectángulo

3.18.3.10. const Punto& Rectangulo::p2(void) const [inline]

Método consultor que devuelve el punto superior derecho del rectángulo

Devuelve

Punto superior derecho del rectángulo

3.18.3.11. const Punto& Rectangulo::p3(void) const [inline]

Método consultor que devuelve el punto inferior derecho del rectángulo

Devuelve

Punto inferior derecho del rectángulo

3.18.3.12. Punto& Rectangulo::p3(void) [inline]

Método modificador que devuelve una referencia al punto inferior derecho del rectángulo

Devuelve

Referencia al punto inferior derecho del rectángulo

3.18.3.13. Punto& Rectangulo::p4(void) [inline]

Método modificador que devuelve una referencia al punto inferior izquierdo del rectángulo

Devuelve

Referencia al punto inferior izquierdo del rectángulo

3.18.3.14. const Punto& Rectangulo::p4(void) const [inline]

Método consultor que devuelve el punto inferior izquierdo del rectángulo

Devuelve

Punto inferior izquierdo del rectángulo

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/collision.h

3.19. Referencia de la Clase Screen

Clase que gestiona los gráficos en la Nintendo Wii.

```
#include <screen.h>
```

Métodos públicos

- void inicializar (void)
- u16 ancho (void) const
- u16 alto (void) const
- void flip (void)
- void crearTextura (GXTexObj *textura, void *pixeles, u16 ancho, u16 alto)
- void dibujarTextura (GXTexObj *textura, s16 x, s16 y, s16 z, u16 ancho, u16 alto)
- void dibujarCuadro (GXTexObj *textura, u16 texAncho, u16 texAlto, s16 x, s16 y, s16 z, s16 cuadroX, s16 cuadroY, u16 cuadroAncho, u16 cuadroAlto, bool invertido=false)
- void dibujarPunto (s16 x, s16 y, s16 z, u32 color)
- void dibujarLinea (s16 x1, s16 y1, s16 x2, s16 y2, s16 z, u16 ancho, u32 color)
- void dibujarRectangulo (s16 x1, s16 y1, s16 x2, s16 y2, s16 x3, s16 y3, s16 x4, s16 y4, s16 z, u32 color)
- void dibujarCirculo (s16 x, s16 y, s16 z, f32 r, u32 color)

Métodos públicos estáticos

- static Screen * get_instance (void)
- static void destroy (void)

Métodos protegidos

- Screen (void)
- ~Screen (void)
- Screen (const Screen &s)
- Screen & operator= (const Screen &s)

3.19.1. Descripción detallada

Clase que gestiona los gráficos en la Nintendo Wii.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

La libogc proporciona una biblioteca de bajo nivel para trabajar con los gráficos, llamada GX. La GX trabaja, de base, en tres dimensiones, y soporta varias proyecciones gráficas, pero como este es un proyecto en 2D, sólo se utilizará la proyección ortográfica. Básicamente, se trata de que todo lo que hay en el mundo del juego se proyecta sobre la pantalla de TV de forma perpendicular (como si el foco de luz estuviera en el infinito), de tal manera que se muestra como si sólo hubiera dos dimensiones, y además, la profundidad no se tiene en cuenta: se ve al mismo tamaño un objeto, esté en la capa 0, o en la 999.

GX espera que, a cada frame, se le pase un buffer con lo que tiene que dibujar en ese momento. Esta clase Screen utiliza un buffer (de ahora en adelante FIFO, First Input First Output, nombre descriptivo sobre cómo funciona) de 1MB, que es el tamaño máximo que está soportado, y es un tamaño más que sobrado para dibujar toda la pantalla en un frame concreto. Además, implementa un sistema de doble buffer de una forma sencilla, de tal manera que se alternan: cada vez que hay que copiar datos al FIFO, se hace desde un buffer distinto. Es decir, que habrá un framebuffer activo en cada frame, que será en el que se dibuje todo, y al final del frame, el contenido de dicho buffer se copia al FIFO.

El modo de pantalla (PAL, NTSC, PAL60) se toma automáticamente de la configuración que tenga la consola en el momento de inicializar la clase, de tal manera que no hay que preocuparse. Además, cuando se configura el modo de vídeo, pasamos a tener disponibles el ancho y el alto de la pantalla en píxeles. Se proporciona también el uso de capas de profundidad, que van desde 0 (primer plano) hasta 999 (capa más profunda). Nota: de momento, esta clase sólo trabaja con proporción 4:3, en un futuro no muy lejano se implementará soporte para 16:9.

Cada punto que se vaya a dibujar en la pantalla consta de tres coordenadas (x,y,z), donde X es la distancia horizontal en píxeles hasta el lado izquierdo de la pantalla, Y es la distancia vertical en píxeles hasta el borde superior de la pantalla, y Z es la capa en la que se va a dibujar (como se ha dicho ya, es un número entero positivo que, cuanto más se acerca de 0, más cerca del primer plano estará el punto). El tipo de estas tres coordenadas es s16 (entero con signo de 16 bits).

Cuando se dibuja un punto, hay que darle color para que se vea (si no, estaríamos hablando de un píxel invisible). El formato de color esperado es un entero sin signo de 32 bits (tipo u32 en la biblioteca), con el formato hexadecimal 0xRRGGBBAA, donde cada pareja representa un componente del color: RR es rojo representado en 8 bits, GG es verde, BB es azul, y AA es el canal alpha. Este canal alpha tiene una peculiaridad, y es que todo lo que tenga un valor inferior a 0x08 será considerado invisible.

La GX de libogc proporciona un serie de funciones de dibujo primitivas para plasmar en la pantalla figuras geométricas. Las que se utilizan en la clase Screen son las siguientes: 1. GX_POINTS: Dibuja un sólo punto. 2. GX_LINES: Dibuja una línea recta entre dos puntos, de un píxel de ancho. 3. GX_TRIANGLES: Dibuja un triángulo a partir de tres puntos. 4. GX_QUADS: Dibujar un rectángulo a partir de cuatro puntos.

Utilizar estas funciones primitivas para dibujar figuras geométricas es muy sencillo. Primero, hay que establecer una serie de descriptores, o dicho de otro modo, parámetros que le pasamos a la GX para que sepa qué vamos a hacer. Estos descriptores incluyen información sobre qué vamos a mandarle y cómo, por ejemplo, se le puede decir "se van a

pintar puntos representados por tres variables s16, con color directo utilizando una variable u32, y no se va a utilizar textura". Posteriormente, con la instrucción GX_Begin(primitiva, GX_VTXFMT0, numero_de_puntos_a_dibujar), comenzamos el bloque de dibujo propiamente dicho. Esta función indica qué primitiva se va a utilizar, y cuántos puntos se emplearán. Por ejemplo, para dibujar dos triángulos, la llamada sería:

```
// Primitiva GX_TRIANGLES, y 6 vértices, para dibujar dos triángulos  
GX_Begin(GX_TRIANGLES, GX_VTXFMT0, 6)
```

Seguidamente, se procede a dibujar los seis puntos, siguiendo el mismo formato que se ha indicado antes con los descriptores. Para dibujar un punto, la llamada es:

```
// Dibujar un punto invisible en las coordenadas (x,y,z)  
GX_Position3s16(s16 x, s16 y, s16 z);  
// Rellenar el último punto dibujado con color rojo (0xRR0000AA)  
GX_Color1u32(0xFF0000FF);
```

Finalmente, hay que cerrar el bloque de dibujo con la función GX_End()

Descripción de la clase Screen

La clase Screen implementa un patrón Singleton, ya que es lógico pensar que sólo se va a utilizar una pantalla en un juego. Tiene tres partes bien diferenciadas, que son las funciones que gestionan la pantalla propiamente dicha, las funciones relativas a texturas, y los métodos que se encargan de dibujar figuras geométricas en el framebuffer.

Las funciones de gestión de pantalla son, principalmente, la de inicialización (inicializar(), llamar al principio del programa una sola vez), la de fin de frame (flip(), que se encarga de copiar lo que tengamos en el framebuffer actual al FIFO y de enviarlo al chip gráfico de la consola), y dos métodos consultores para saber, en todo momento, el ancho y alto de la pantalla en píxeles.

El punto fuerte de la clase Screen tiene que ver con las texturas. Se proporciona una función, crearTextura(), que a partir de una zona de memoria en la cual haya píxeles de 8 bits por color y en formato RGB, creará un objeto de textura GXTexObj, con el formato que utiliza la GX, y preparado para trabajar con él. El formato de imagen que utiliza la Wii es RGB5A3, de 16 bits. Se representa con 5 bits para cada color, y uno para el canal alpha, pero se puede utilizar un píxel de cada color para determinar niveles de transparencia más complejos. Esta función de creación de texturas modifica la zona de memoria que almacena la información de los píxeles, convirtiéndola en tiles de 4×4.

Una vez creada la textura, se puede dibujar tal cual en la pantalla utilizando la función dibujarTextura(), indicando sus coordenadas y su ancho y alto. También se puede dibujar una parte de esa textura, utilizando la función dibujarCuadro(), a la cual hay que pasarle las coordenadas de dibujo, el ancho y alto de la textura completa, y tanto las coordenadas X e Y de la parte de la textura que se quiere dibujar, como el ancho y el alto del cuadro. Internamente, estas dos funciones de dibujo hacen uso de una función privada de la clase, configurarTextura(), que se encarga de establecer los descriptores de la GX para dibujar texturas.

El tercer bloque de funciones de dibujo de la clase Screen son, básicamente, funciones que permiten dibujar formas geométricas (como son un punto, una línea recta, un rectángulo y un círculo), en un color plano, y a partir de las primitivas que se describieron anteriormente. Importante: al dibujar un rectángulo, el orden de los puntos debe ser, sí o sí, izquierda-arriba, derecha-arriba, derecha-abajo, e izquierda-abajo. Si no se sigue esta recomendación, existe riesgo de provocar un cuelgue en el sistema. Internamente, estas funciones de dibujo hacen uso de una función privada de la clase, `configurarColor()`, que se encarga de establecer los descriptores de la GX para dibujar con color directo de relleno.

Para más información sobre los métodos de la clase Screen, consultar su descripción.

Ejemplo de uso:

```
screen->inicializar();

// Bucle principal
while(1) {
    // Gestionar entrada, realizar cálculos, etc.
    // ...

    // Fase de dibujo: un circulo rojo y un rectangulo azul
    screen->dibujarCirculo(100, 100, 5, 30.0f, 0xFF0000FF);
    screen->dibujarRectangulo(300, 200, 400, 200, 400, 250, 300, 250, 10, 0x0
000FFFF);

    // Fin de frame
    screen->flip();
}
```

3.19.2. Documentación del constructor y destructor

3.19.2.1. `Screen::Screen(void) [inline, protected]`

Constructor de la clase Screen. En la zona protegida debido a la implementación del patrón Singleton. La llamada al constructor inicializa el sistema de vídeo de la Nintendo Wii. Fija la constante `FIFO_SIZE`, que será el tamaño del bus que envía los datos al procesador gráfico en cada frame.

3.19.2.2. `Screen::~Screen(void) [protected]`

Destructor de la clase Screen. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.19.2.3. `Screen::Screen(const Screen & s) [protected]`

Constructor de copia de la clase Screen. Se encuentra en la zona protegida debido a la implementación del patrón Singleton (evita la copia).

3.19.3. Documentación de las funciones miembro

3.19.3.1. `u16 Screen::alto (void) const`

Método observador para el alto de la pantalla en píxeles

Devuelve

Alto de la pantalla en píxeles

3.19.3.2. `u16 Screen::ancho (void) const`

Método observador para el ancho de la pantalla en píxeles

Devuelve

Ancho de la pantalla en píxeles

3.19.3.3. `void Screen::crearTextura (GXTexObj * textura, void * pixeles, u16 ancho, u16 alto)`

Método que, a partir de una zona de memoria con información de píxeles, crea un objeto de textura GXTexObj, con el cual puede trabajar la biblioteca de bajo nivel GX. El formato de la zona de memoria debe ser una secuencia de píxeles, con 24 bits por píxel (ocho bits por color, primero rojo, seguido de verde y finalmente azul). Las medidas de la imagen formada por la información de estos píxeles debe tener medidas (alto y ancho) múltiplos de 8.

Parámetros

<i>textura</i>	Dirección de memoria a un objeto GXTexObj recién creado.
<i>pixeles</i>	Dirección a la zona de memoria donde se encuentra la información de la imagen.
<i>ancho</i>	Ancho en píxeles que tiene la imagen que se convertirá en textura.
<i>alto</i>	Alto en píxeles que tiene la imagen que se convertirá en textura.

3.19.3.4. `static void Screen::destroy (void) [inline, static]`

Método de clase que limpia la memoria ocupada por la instancia activa en el sistema de la clase Screen, llamando a su destructor.

3.19.3.5. `void Screen::dibujarCirculo (s16 x, s16 y, s16 z, f32 r, u32 color)`

Método que dibuja un círculo relleno de un color concreto a partir de un punto y un radio.

Parámetros

<i>x</i>	Coordenada X del centro del círculo.
<i>y</i>	Coordenada Y del centro del círculo.
<i>z</i>	Coordenada Z (capa) donde se dibujará el círculo. Entre 0 y 999.
<i>r</i>	Radio, en píxeles, del círculo.
<i>color</i>	Entero de 32 bits sin signo que indica el color del círculo. Formato 0xRRGGBB-BA

3.19.3.6. void Screen::dibujarCuadro (GXTexObj * *textura*, u16 *texAncho*, u16 *texAlto*, s16 *x*, s16 *y*, s16 *z*, s16 *cuadroX*, s16 *cuadroY*, u16 *cuadroAncho*, u16 *cuadroAlto*, bool *invertido = false*)

Método que dibuja una parte de una textura en formato GXTexObj en unas coordenadas (x,y,z) de la pantalla. En esas coordenadas se dibujará la esquina superior izquierda de la zona de la textura que se quiera dibujar. Dicha zona viene determinada por unas coordenadas (x,y), que indican, respectivamente, la distancia en horizontal al lado izquierdo de la textura, y en vertical al lado superior de ésta, del punto superior izquierdo del cuadro de la textura que se va a dibujar. Este cuadro también se define por su alto y ancho en píxeles.

Parámetros

<i>textura</i>	Dirección de memoria de la textura que se va a dibujar.
<i>texAncho</i>	Ancho en píxeles de la textura a dibujar
<i>texAlto</i>	Alto en píxeles de la textura a dibujar
<i>x</i>	Coordenada X donde se comenzará a dibujar la textura.
<i>y</i>	Coordenada Y donde se comenzará a dibujar la textura.
<i>z</i>	Coordenada Z (capa) donde se comenzará a dibujar la textura. Entre 0 y 999.
<i>cuadroX</i>	Coordenada X (interna a la textura) donde comienza el cuadro a dibujar.
<i>cuadroY</i>	Coordenada Y (interna a la textura) donde comienza el cuadro a dibujar.
<i>cuadroAncho</i>	Ancho en píxeles del cuadro de la textura que se va a dibujar.
<i>cuadroAlto</i>	Alto en píxeles del cuadro de la textura que se va a dibujar.
<i>invertido</i>	Verdadero si se quiere dibujar el cuadro de la textura invertido respecto al eje vert.

3.19.3.7. void Screen::dibujarLinea (s16 *x1*, s16 *y1*, s16 *x2*, s16 *y2*, s16 *z*, u16 *ancho*, u32 *color*)

Método que dibuja una recta rellena de un color concreto entre dos puntos.

Parámetros

<i>x1</i>	Coordenada X del punto origen de la recta.
<i>y1</i>	Coordenada Y del punto origen de la recta.
<i>x2</i>	Coordenada X del punto final de la recta.
<i>y2</i>	Coordenada Y del punto final de la recta.

<i>z</i>	Coordenada Z (capa) donde se dibujará la recta. Entre 0 y 999.
<i>ancho</i>	Ancho en píxeles de la recta
<i>color</i>	Entero de 32 bits sin signo que indica el color de la recta a dibujar. Formato 0xRRGGBBAA

3.19.3.8. void Screen::dibujarPunto (s16 *x*, s16 *y*, s16 *z*, u32 *color*)

Método que dibuja un punto de un color concreto en unas coordenadas (x,y,z).

Parámetros

<i>x</i>	Coordenada X donde se dibujará el punto.
<i>y</i>	Coordenada Y donde se dibujará el punto.
<i>z</i>	Coordenada Z (capa) donde se dibujará el punto. Entre 0 y 999.
<i>color</i>	Entero de 32 bits sin signo que indica el color del punto a dibujar. Formato 0xRRGGBBAA

3.19.3.9. void Screen::dibujarRectangulo (s16 *x1*, s16 *y1*, s16 *x2*, s16 *y2*, s16 *x3*, s16 *y3*, s16 *x4*, s16 *y4*, s16 *z*, u32 *color*)

Método que dibuja un rectángulo relleno de un color concreto a partir de cuatro puntos. El orden de los puntos debe ser, obligatoriamente, izquierda-arriba, derecha-arriba, derecha-abajo, izquierda-abajo. Si no se sigue esta recomendación, puede darse un comportamiento inesperado.

Parámetros

<i>x1</i>	Coordenada X del punto superior izquierdo del rectángulo.
<i>y1</i>	Coordenada Y del punto superior izquierdo del rectángulo.
<i>x2</i>	Coordenada X del punto superior derecho del rectángulo.
<i>y2</i>	Coordenada Y del punto superior derecho del rectángulo.
<i>x3</i>	Coordenada X del punto inferior derecho del rectángulo.
<i>y3</i>	Coordenada Y del punto inferior derecho del rectángulo.
<i>x4</i>	Coordenada X del punto inferior izquierdo del rectángulo.
<i>y4</i>	Coordenada Y del punto inferior izquierdo del rectángulo.
<i>z</i>	Coordenada Z (capa) donde se dibujará el rectángulo. Entre 0 y 999.
<i>color</i>	Entero de 32 bits sin signo que indica el color del rectángulo. Formato 0xRRGGBBAA

3.19.3.10. void Screen::dibujarTextura (GXTexObj * *textura*, s16 *x*, s16 *y*, s16 *z*, u16 *ancho*, u16 *alto*)

Método que dibuja una textura en formato GXTexObj en unas coordenadas (x,y,z) de la pantalla. En esas coordenadas se dibujará la esquina superior izquierda de la textura.

Parámetros

<i>textura</i>	Dirección de memoria de la textura que se va a dibujar.
<i>x</i>	Coordenada X donde se comenzará a dibujar la textura.
<i>y</i>	Coordenada Y donde se comenzará a dibujar la textura.
<i>z</i>	Coordenada Z (capa) donde se comenzará a dibujar la textura. Entre 0 y 999.
<i>ancho</i>	Ancho en píxeles de la textura a dibujar
<i>alto</i>	Alto en píxeles de la textura a dibujar

3.19.3.11. void Screen::flip(void)

Método que da por finalizado un frame. Si se ha marcado el flag interno de actualización de gráficos, se copia al buffer FIFO el contenido del framebuffer activo, y se establece el otro como activo para el siguiente frame.

3.19.3.12. static Screen* Screen::get_instance(void) [inline, static]

Método de clase que devuelve la dirección de memoria de la instancia activa en el sistema de la clase Screen. Si aún no existe dicha instancia, la crea y la devuelve. Además, establece que al salir del programa, se destruya por medio de una llamada al destructor.

3.19.3.13. void Screen::inicializar(void)

Método que inicializa todo el sistema de vídeo de la consola Nintendo Wii. Se debe llamar al comienzo del programa una sola vez.

3.19.3.14. Screen& Screen::operator=(const Screen & s) [protected]

Operador de asignación de la clase Screen. Se encuentra en la zona protegida debido a la implementación del patrón Singleton (evita la copia)

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/screen.h

3.20. Referencia de la Clase Sdcard

Clase que gestiona la ranura para tarjetas SD de la consola Wii.

```
#include <sdcards.h>
```

Métodos públicos

- void inicializar (const std::string &unidad="SD")

- const std::string & unidad (void) const
- bool montada (void) const
- void desmontar (void)
- bool existe (const std::string &archivo)

Métodos públicos estáticos

- static Sdcard * get_instance (void)
- static void destroy (void)

Métodos protegidos

- Sdcard (void)
- ~Sdcard (void)
- Sdcard (const Sdcard &l)
- Sdcard & operator= (const Sdcard &l)

3.20.1. Descripción detallada

Clase que gestiona la ranura para tarjetas SD de la consola Wii.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Esta clase implementa un patrón Singleton, debido a que la consola Nintendo Wii sólo dispone de una ranura lectora de tarjetas SD. Además, para desarrollar juegos y aplicaciones relativamente sencillas en dos dimensiones, sólo se necesita acceder a un único dispositivo de almacenamiento masivo, por lo que se ha decidido no desarrollar más allá que esta clase.

Es cierto que la libogc puede permitir acceder a dispositivos USB de almacenamiento (comúnmente conocidos como "lápices usb"), pero se ha descartado implementar esta funcionalidad en la biblioteca, tanto por simplificar su utilización, como por ahorro de tiempo. Está planeado, en un futuro, implementar soporte para dispositivos USB.

Siguiendo con la norma de la sencillez, la clase Sdcard sólo puede montar una única partición. El formato en el que debe estar dicha partición es FAT o FAT32, ya que la biblioteca auxiliar que utiliza libogc para montar unidades y particiones es una implementación para Wii de libfat.

Funcionamiento interno

La mecánica interna de esta clase se basa en realizar diez intentos de montar la primera partición de la tarjeta SD insertada en la ranura de la consola. Si fallaran los diez intentos, la clase sale del programa con un código de error 1.

El proceso que se sigue al intentar montar una partición consiste en inicializar el hardware del lector de tarjetas. Si esto se consigue, se intenta montar, con un sistema de archivos FAT o FAT32, la primera partición válida de la tarjeta SD. Si no se consigue, se apaga el hardware de la ranura, se contabiliza el intento, y se prueba de nuevo con el mismo proceso.

Hay que tener en cuenta que una tarjeta puede tener varias particiones, y éstas pueden ser de distintos sistemas de archivos. Pues bien, esta clase espera que la primera partición esté formateada como FAT/FAT32, porque es la única a la que puede acceder. Así pues, queda bajo la responsabilidad del usuario la correcta preparación de la SD para poder utilizar esta clase.

Utilización de la clase

Cuando todo sale bien, se monta la partición con el nombre de unidad que se recibe por parámetro en el método inicializador, y se pone a disposición del resto del sistema una serie de funciones para conocer el nombre de esta unidad, saber si la partición está montada o no, y para desmontarla en cualquier momento. Si se desmonta (por el motivo que sea) la unidad, se puede volver a utilizar inicializándola de nuevo.

Generalmente, esta clase se inicia al principio del programa, y se desmonta al final, ya que si se utiliza también la clase Logger (consultar documentación de dicha clase), ésta realiza la escritura al final de la ejecución.

Para realizar una operación de lectura y/o escritura, basta con anteponer el nombre de la unidad, seguida de dos puntos (:), y después la dirección absoluta del recurso a cargar/modificar en formato UNIX (por ejemplo, string(sdcards->unidad() + ':' + '/apps/wiipang/xml/media.xml') sería la forma de abrir el archivo media.xml).

Unos últimos detalles. La propia implementación del patrón Singleton se encarga de que se destruya la instancia activa de la clase en el sistema al salir del programa mediante la instrucción exit(). También hay que tener en cuenta que las clases que encapsulan recursos media (como la clase Imagen, o la clase Musica) requieren que Sdcards haya montado previamente la unidad FAT/FAT32 sobre la que se van a realizar las operaciones de entrada y salida.

Pequeño ejemplo de uso:

```
// Inicializar el lector de tarjetas, y montar una unidad llamada 'SD'  
sdcards->inicializar("SD");  
  
// Guardar en un std::string el nombre de la unidad  
std::string unidad = sdcards->unidad();  
  
// Saber si está montada la partición; si lo está desmontarla  
if ( sdcards->montada() )  
    sdcards->desmontar();
```

3.20.2. Documentación del constructor y destructor

3.20.2.1. **Sdcard::Sdcard (void) [inline, protected]**

Constructor de la clase Sdcard. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.20.2.2. **Sdcard::~Sdcard (void) [protected]**

Destructor de la clase Sdcard. Se encuentra en la zona protegida debido a la implementación del patrón Singleton. Desmonta la partición si estuviera montada.

3.20.2.3. **Sdcard::Sdcard (const Sdcard & I) [protected]**

Constructor de copia de la clase Sdcard. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.20.3. Documentación de las funciones miembro

3.20.3.1. **void Sdcard::desmontar (void)**

Método que desmonta la partición de la unidad. Si no estuviera montada, no hace nada.

3.20.3.2. **static void Sdcard::destroy (void) [inline, static]**

Método de clase que limpia la memoria ocupada por la instancia activa en el sistema de la clase Sdcard, llamando a su destructor.

3.20.3.3. **bool Sdcard::existe (const std::string & archivo)**

Método que comprueba si un archivo existe en la tarjeta SD.

Parámetros

<i>archivo</i>	Ruta absoluta del archivo que se quiere comprobar.
----------------	--

3.20.3.4. **static Sdcard* Sdcard::get_instance (void) [inline, static]**

Método de clase que devuelve la dirección de memoria de la instancia activa en el sistema de la clase Sdcard. Si aún no existe dicha instancia, la crea y la devuelve. Además, establece que al salir del programa, se destruya por medio de una llamada al destructor.

3.20.3.5. void Sdcard::inicializar (const std::string & unidad = "SD")

Método que inicializa la clase. Intenta montar la primera partición válida de la tarjeta SD insertada en la consola, utilizando un sistema de archivos FAT/FAT32. Se realizan diez intentos, si no se ha conseguido un montaje correcto tras dichos intentos, se sale de programa con código de error 1.

Parámetros

<i>unidad</i>	Nombre de unidad que se le asignará a la partición montada.
---------------	---

3.20.3.6. bool Sdcard::montada (void) const

Método consultor para saber si la unidad está montada o no.

Devuelve

Devuelve verdadero si la unidad está montada, o falso en caso contrario.

3.20.3.7. Sdcard& Sdcard::operator= (const Sdcard & l) [protected]

Operador de asignación de la clase Sdcard. Se encuentra en la zona protegida debido a la implementación del patrón Singleton.

3.20.3.8. const std::string& Sdcard::unidad (void) const

Método consultor para el nombre de la unidad asignado a la partición montada

Devuelve

Nombre de la unidad asignado a la partición montada

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/sdcard.h

3.21. Referencia de la Clase Sonido

Clase que representa un efecto de sonido que se puede reproducir.

```
#include <sonido.h>
```

Métodos públicos

- Sonido (const std::string &ruta, u8 volder=255, u8 volizq=255) throw (ArchivoEx, TarjetaEx)

- ~Sonido (void)
- bool play (void) const
- void setVolumenIzquierdo (u8 volumen)
- void setVolumenDerecho (u8 volumen)

Métodos públicos estáticos

- static void inicializar (void)

Métodos protegidos

- Sonido (const Sonido &m)
- Sonido & operator= (const Sonido &m)

3.21.1. Descripción detallada

Clase que representa un efecto de sonido que se puede reproducir.

Autor

Ezequiel Vázquez de la Calle

Versión

0.9.2

Al igual que el resto de clases que representa a un recurso multimedia, la clase Sonido proporciona abstracción sobre las funciones y estructuras de datos necesarias para reproducir efectos de sonido en la Wii. La base sobre la que se trabaja es la libASND, que forma parte de la libogc.

Cada instancia de la clase representa un efecto de sonido concreto, que puede ser reproducido en cualquier momento de la ejecución. Se crea a partir de un archivo de sonido, con un formato bastante concreto, pero que es muy fácil de obtener a partir de prácticamente cualquier archivo de audio mediante el uso de SoX, una herramienta libre, que consiste en un conversor de formatos de audio muy potente, y que se utiliza en línea de comandos de cualquier sistema UNIX (de hecho, se incluye en los repositorios de Ubuntu).

El mencionado formato al que necesitamos convertir todos nuestros efectos de sonido (que no música, reitero), es PCM crudo (es decir, sólo la información del sonido, sin cabeceras de archivo), con distribución de bytes en big endian, samples de 16 bits con signo, frecuencia de 48000 Hz y 2 canales. Evidentemente, la consola puede reproducir muchos más formatos de sonido (mono y estéreo, samples de 8 y 16 bits, y frecuencias en un rango que va desde 1 Hz hasta los 144000Hz), pero se ha seleccionado éste porque, según la documentación consultada, es el formato nativo con el que trabaja la consola.

Para transformar un sonido input.wav en otro con el formato explicado output.pcm, se utilizaría la siguiente orden en la consola de comandos:

```
sox input.wav -V3 --encoding signed-integer --type raw --bits 16 --endian big --
channels 2 --rate 48000 output.pcm
```

Una vez finalizada la conversión, basta con copiar el archivo output.pcm a la tarjeta SD, y trabajar con la clase Sonido para tener el efecto disponible para su reproducción.

Existe un detalle importante a tener en cuenta, y es relativo al sistema de sonido de la Nintendo Wii. La libASND puede mezclar 16 voces de sonido, una de las cuales está reservada para la pista de música. Esto es así porque la cantidad de instrucciones a procesar para mezclar 16 voces con la diversidad de formatos expuesta antes es muy elevada.

En la libASND, una voz no es más que un flujo de datos, cuyo estado se comprueba con una interrupción de audio que es llamada cada 21,333 ms. Si hubiera entrada en una voz, ésta pasaría directamente al DSP, que es un procesador cuya única tarea es la de procesar los efectos de sonido. Con esto se consigue descongestionar al procesador principal, y obtener una mayor potencia a la hora de trabajar con el audio.

Cada vez que un sonido es reproducido, se busca una voz desocupada (que no esté reproduciendo nada en ese momento), y se envía allí la secuencia de datos que componen el sonido. Evidentemente, esto quiere decir que sólo se pueden reproducir, en un mismo momento, 15 efectos de sonido y una pista de música, cantidad más que suficiente para el desarrollo de juegos en dos dimensiones. Si en algún momento, se intentaran reproducir más sonidos de los que se permiten, no ocurriría ningún error, porque existe un mecanismo que evita que haya conflictos: si no hay ninguna voz libre, el sonido no se intenta reproducir.

Funcionamiento interno de la clase Sonido

Esta clase es muy sencilla en su implementación, ya que únicamente carga desde un archivo alojado en la tarjeta SD un flujo de bytes, que se almacena tal cual en memoria, y que se envía a la función de reproducción cuando se le solicita. Para realizar la carga a memoria, se comprueba en primer lugar que la unidad de la tarjeta esté montada y disponible para realizar una operación de lectura. Posteriormente, se comprueba el tamaño del archivo, y se reserva la correspondiente memoria alineada a 32 bytes, con formato de enteros de 16 bits con signo (s16 como tipo del puntero). En último lugar, se lee el propio archivo a la zona de memoria alineada, y se cierra el flujo de bytes.

Un sonido dispondrá también de dos valores enteros de 8 bits sin signo, que representan el volumen de cada uno de los dos canales de audio. Estos dos valores se le pueden pasar al constructor junto con la ruta absoluta del archivo a cargar, y también se pueden modificar en tiempo de ejecución, de tal manera que, en ese sentido, la clase es bastante flexible.

Para escuchar un sonido, se busca la primera voz para efectos que esté libre, y se vuelca en ella la información del sonido, comenzando la reproducción. Como esta clase está pensada para pequeños efectos de sonido, no se proporciona un método de parada o pausa de la reproducción, ya que se sobreentiende que un efecto de sonido dura, a lo sumo, cinco o seis segundos.

Existe una función estática y pública, llamada Sonido::inicializar(), a la cual es necesario llamar antes de trabajar con la clase. Dicha función también establece que, al salir del programa, se apague el subsistema de audio de la consola.

Por último, un pequeño fragmento de código para ilustrar el uso de la clase:

```
// Cargar el sonido, previamente formateado con SoX, desde la SD. Volumen de los
// dos canales por defecto
Sonido s( "/apps/wipang/media/output.pcm" );
//Modificar los volúmenes de ambos canales, dando mayor volumen al canal derecho

s.setVolumenDerecho(255);
s.setVolumenIzquierdo(100);
// Reproducir el sonido con los valores actuales de volumen
s.play();
```

3.21.2. Documentación del constructor y destructor

3.21.2.1. Sonido::Sonido (const std::string & ruta, u8 volder = 255, u8 volizq = 255) throw (ArchivoEx, TarjetaEx)

Constructor predeterminado de la clase Sonido.

Parámetros

<i>ruta</i>	Ruta absoluta del fichero de sonido desde el que se cargará el efecto de sonido
<i>volder</i>	Volumen del canal derecho
<i>volizq</i>	Volumen del canal izquierdo

Excepciones

<i>ArchivoEx</i>	Se lanza si hay algún error al abrir el archivo al que apunta la ruta aportada
<i>TarjetaEx</i>	Se lanza si ocurre un error relacionado con la tarjeta SD

3.21.2.2. Sonido::~Sonido (void)

Destructor de la clase Sonido.

3.21.2.3. Sonido::Sonido (const Sonido & m) [protected]

Constructor de copia de la clase Sonido. Se encuentra en la zona protegida para no permitir la copia.

3.21.3. Documentación de las funciones miembro

3.21.3.1. static void Sonido::inicializar (void) [inline, static]

Método de clase para inicializar el sistema de sonido de la consola.

3.21.3.2. Sonido& Sonido::operator= (const Sonido & m) [protected]

Operador de asignación de la clase Sonido. Se encuentra en la zona protegida para no permitir la asignación.

3.21.3.3. bool Sonido::play (void) const

Método que reproduce un sonido una sola vez, con el volumen prefijado para cada canal. Si el sonido pudiera reproducirse correctamente, se devuelve un valor True, en caso contrario, el método devuelve un valor False.

Devuelve

Se devuelve verdadero si el sonido se reproduce correctamente, en caso contrario, falso.

3.21.3.4. void Sonido::setVolumenDerecho (u8 volumen)

Función modificadora del volumen del canal derecho del sonido.

Parámetros

<i>volumen</i>	Nuevo valor para el volumen del canal derecho (entre 0 y 255).
----------------	--

3.21.3.5. void Sonido::setVolumenIzquierdo (u8 volumen)

Función modificadora del volumen del canal izquierdo del sonido.

Parámetros

<i>volumen</i>	Nuevo valor para el volumen del canal izquierdo (entre 0 y 255).
----------------	--

La documentación para esta clase fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/sonido.h

3.22. Referencia de la Estructura Nivel::tile

Estructura que almacena los datos de un tile.

```
#include <nivel.h>
```

Atributos públicos

- u32 x
- u32 y

- u32 **gid**
- Rectangulo * **colision**

3.22.1. Descripción detallada

Estructura que almacena los datos de un tile. Se compone de una pareja de coordenadas (X,Y), que especifican la distancia del punto superior izquierdo del tile respecto al origen de coordenadas (0,0) del nivel, un entero de 32 bits que indica el número identificador del tile dentro del mapa de tiles generado con la aplicación Tiled, y un puntero a un rectángulo de colisión asociado al tile (puede ser NULL).

La documentación para esta estructura fue generada a partir del siguiente fichero:

- /home/rabbit/Escritorio/libwiiesp/include/nivel.h

Apéndice D

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “**publisher**” means any person or entity that distributes copies of the Do-

cument to the public.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of

added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in

the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later

version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with ... Texts.” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.