GUIA DE DESARROLLO DE APLICACIONES SOBRE WII

Proyecto final de carrera de la Escuela Técnica Superior de Ingeniería Informática

Autor: Víctor Díaz Bernal

Director: Manuel Agustí i Melchor

Septiembre de 2011

Índice

Capítulo 1. Introducción....5
1.1. Presentación....5
1.2. Desarrollo en Nintendo Wii sin el SDK oficial....6
1.3. Descripción de la Nintendo Wii....7
1.3.1. Hardware interno....7
1.3.2. El sistema de arranque....8
1.3.3. El Wiimote....8
1.3.4. El System Menu....9
1.4. Instalación del Homebrew Channel y ejecución de aplicaciones caseras....9
1.4.1. Aplicaciones en el Homebrew Channel....13

Capítulo 2. El entorno de desarrollo....15
2.1. Introducción....15
2.2. DevkitPro. Instalación....15
2.3. Compilación. La consola Msys....17
2.4. Ficheros Makefile....18
2.5. Tipos de datos específicos en Wii....19
2.6. Primer ejemplo: "¡Hola Mundo!"....20

Capítulo 3. Manejo de los controles....23
3.1. Introducción....23
3.2. Inicialización del Wiimote....23
3.3. Detección de botones....24
3.4. Lectura de los infrarrojos....26
3.4.1. Información adicional del Wiimote. Orientación y aceleración....26
3.4.2. Obtención de datos del Wiimote de forma general. La función WPAD_DATA....27
3.5. Control de la vibración del Wiimote....27
3.6. Uso de extensiones....27
3.7. Ejemplos anexos a la memoria relativos al capítulo 3....29

Capítulo 4. El sistema de ficheros....30
4.1. Introducción....30
4.2. Inicializando el sistema de ficheros....30
4.3. Funciones habituales de acceso a ficheros....30
4.3.1. Apertura de un fichero....30
4.3.2. Cierre de ficheros....31
4.3.3. Lectura de ficheros....31
4.3.4. Escritura en ficheros....32
4.4. Operaciones con ficheros XML. La librería mxml....32
4.4.1. Escritura en un fichero XML....33
4.4.1.1. Creación de un fichero XML nuevo....33
4.4.1.2. Creación de un nuevo elemento....33
4.4.1.3. Atributos....33
4.4.1.4. Añadiendo contenido a un elemento....34
4.4.1.5. Salvando un fichero XML....34
4.4.2. Lectura de un fichero XML....35

4.4.2.1. Cargando el fichero XML....35
4.4.2.2. Búsqueda de un elemento....35
4.4.2.3. Obtención de información de un elemento....36
4.5. Ejemplos anexos a la memoria relativos al capítulo 4....37

Capítulo 5. Imágenes....38
5.1. Introducción. La librería wiisprite....38
5.2. Inicializando el subsistema de vídeo....38
5.3. Carga de imágenes. La clase Image....39
5.3.1. Carga de imágenes desde un directorio....39
5.3.2. Carga de imágenes desde *buffer*. La herramienta raw2c....40
5.4. Manipulación de imágenes. La clase Sprite....40
5.4.1. Posicionamiento de las imágenes....40
5.4.2. Manipulación de las imágenes....41
5.4.2.1. Manipulación de las dimensiones de la imagen....42
5.4.2.2. Manipulación de la rotación de la imagen....42
5.4.2.3. Desplazamiento de la imagen....42
5.4.2.4. Manipulación de la transparencia y visibilidad de la imagen....42
5.4.3. Imágenes como puntero del Wiimote....43
5.4.4. Píxel de referencia y posicionamiento....43
5.5. Gestión de capas. La clase LayerManager....44
5.6. Quads....46
5.6.1. Funciones específicas de la clase Quad....46
5.7. Detección de colisiones....46
5.7.1. Rectángulo de colisión....47
5.7.1.1. Rectángulos de colisión y Quads....48
5.7.1.2. Rectángulos de colisión y zonas de la pantalla....48
5.8. *Sprite sheets*. La clase TiledLayer....49
5.8.1. Creación de un objeto TiledLayer....50
5.8.2. Renderizado de los objetos TiledLayer....51
5.8.3. Celdas animadas....54
5.8.4. Otras funciones de la clase TiledLayer....55
5.9. Ejemplos anexos a la memoria relativos al capítulo 5....55

Capítulo 6. Sonidos....57
6.1. Introducción....57
6.2. Inicialización del subsistema de audio....57
6.3. Reproducción de MP3. La librería Mp3Player....57
6.3.1. Reproducción de un fichero MP3 desde *buffer*....57
6.3.2. Reproducción de un fichero MP3 desde un directorio....58
6.3.3. Otras funciones de la librería Mp3Player....59
6.4. Reproducción de ficheros MOD. La librería ModPlay....59
6.4.1. Reproducción de un fichero MOD desde *buffer*....59
6.4.2. Otras funciones de la librería ModPlayer....60
6.5. Ejemplos anexos a la memoria relativos al capítulo 6....61

Capítulo 7. Conclusiones....62

Bibliografía y referencias....64

Lista de figuras

Figura 1.1. Material de trabajo....5
Figura 1.2. Esquema del hardware de la Nintendo Wii....7
Figura 1.3. Advertencia sobre estafas de HackMii....9
Figura 1.4. Instalación de BootMii y Hombrew Channel....10
Figura 1.5. Instalación de Homebrew Channel....11
Figura 1.6. Instalación de BootMii....11
Figura 1.7. Menú de BootMii....12
Figura 1.8. Menú de restauración....12
Figura 1.9. Restauración del sistema mediante *backup*....13
Figura 1.10. Inicio del Homebrew Channel....13
Figura 1.11. Menú "Home" del Homebrew Channel....14
Figura 2.1. Instalador de DevkitPro....15
Figura 2.2. Selección de descarga e instalación de archivos de DevkitPro....15
Figura 2.3. Selección de conservación de los ficheros descargados....16
Figura 2.4. Selección de componentes a instalar....16
Figura 2.5. Selección del directorio de instalación....17
Figura 2.6. Estructura habitual del directorio de un proyecto....17
Figura 2.7. Compilación con la consola MSYS....18
Figura 2.8. Resultado de la ejecución de "Hello World!" (Resolución de pantalla 640x480)....22
Figura 5.1. Sentido de crecimiento de las coordenadas x e y....41
Figura 5.2. Situación de un *sprite* en las coordenadas x e y....41
Figura 5.3. Comportamiento del píxel de referencia....44
Figura 5.4. Resultados de la función Draw....45
Figura 5.5. Posición de las capas según su índice....46
Figura 5.6. Ejemplo de colisiones de *sprites*....47
Figura 5.7. División de la pantalla en zonas de colisión....49
Figura 5.8. Ejemplo de *sprite sheet*....49
Figura 5.9. Indice de las filas y columnas....50
Figura 5.10. División de la imagen mediante la función SetStaticTileset....51
Figura 5.11. Asignación de índices equivalentes en imágenes diferentes....51
Figura 5.12. Imagen dividida....52
Figura 5.13. Resultado de las sentencias SetCell....53
Figura 5.14. Resultado de la sentencia FillCells....53
Figura 5.15. Imagen con índices animados....54
Figura 5.16. Resultado de la sentencia SetAnimatedTile....55

Capítulo 1. Introducción

1.1. Presentación

El objetivo de esta memoria es que sirva como guía de introducción al desarrollo de aplicaciones para la Nintendo Wii (en adelante Wii). En los siguientes capítulos se describe el uso de varias librerías y herramientas, algunas incluidas en el kit de desarrollo no oficial (devkitPro) y otras importadas, que permiten el desarrollo de aplicaciones que hagan uso de las funcionalidades básicas de la consola, tales como gráficos, sonidos e interacción con los mandos.

En esta memoria se asume que el lector ya dispone de unos conocimientos básicos de programación en C y C++, por lo que se evitará explicar cómo programar en estos lenguajes y sólo se explicará el uso de las funciones de cada librería y las peculiaridades de la programación en esta plataforma. El objetivo es introducir a programadores con conocimientos básicos de C y C++ en el desarrollo de aplicaciones no oficiales para Wii, explicando cómo usar algunas de las diversas librerías y herramientas proporcionadas por el kit de desarrollo DevkitPro. De esta forma, se muestra cómo manejar la funcionalidad más característica de la consola, su peculiar mando, y se presentan las herramientas para incluir y manejar elementos multimedia tales como imágenes, audio y texto, que permitan crear aplicaciones atractivas para esta plataforma, destinada inicialmente a video-juegos. La principal motivación a la hora de realizar el proyecto ha sido la de reunir y presentar esta información ya que, muchas veces, las librerías disponibles no incluyen documentación y la información existente suele encontrarse dispersa en Internet.

En el proyecto se incluye el código de aplicaciones sencillas que sirvan como ejemplo ilustrativo de lo explicado, mostrando de forma práctica el uso de las funciones de las librerías presentadas a lo largo de la guía. El desarrollo de estos ejemplos se ha realizado sobre un sistema Windows, aunque también puede realizarse sobre sistemas Linux y Mac OSX. Estas demostraciones se han probado en una consola Wii, con una versión de firmware 4.2, con un mando Wiimote y una expansión Nunchuk disponibles, mostrados en la **figura 1.1**. Para ejecutar software no oficial en Wii también es necesaria una tarjeta de memoria SD de hasta 2 gigabytes.

Figura 1.1. Material de trabajo.

El software usado, principalmente las librerías incluidas en devkitPro así como las herramientas para instalar y ejecutar software no oficial en Wii, son gratuitas y de libre acceso. Por otra parte, Nintendo intenta eliminar el uso de software no oficial en su plataforma así que es posible que estas herramientas no funcionen en consolas con una versión de firmware superior a la 4.2.

1.2. Desarrollo en Nintendo Wii sin el SDK oficial

Las siglas SDK se refieren a un kit de desarrollo de software (del inglés, *Software Development Kit*), que es un conjunto de librerías que da acceso a los programadores a las principales funcionalidades de una plataforma. Normalmente, la compañía propietaria de la plataforma pone a la venta un kit de desarrollo oficial disponible para las compañías desarrolladoras. Desde la web www.warioworld.com se ofrece apoyo a los desarrolladores autorizados por Nintendo.

Se conoce como SDK no oficial a un conjunto de librerías creadas por personas ajenas a la compañía propietaria de la plataforma. Al contrario que con el SDK oficial, el kit no oficial puede tener un acceso limitado a las funcionalidades de la plataforma. La creación de las librerías no oficiales depende normalmente de programadores particulares sin intereses lucrativos, por lo que el desarrollo de estas librerías suele ser lento, dependiente de procesos de ingeniería inversa y, en ocasiones, puede quedar abandonado. Sin embargo, la distribución de estas librerías suele realizarse bajo licencias que permiten su libre distribución y modificación. Además, suele existir una comunidad en Internet detrás el desarrollo de las librerías no oficiales que puede encargarse de mantener las librerías actualizadas y ayudar a los programadores.

Las aplicaciones desarrolladas con un SDK no oficial se conocen comúnmente con el término *homebrew*, que significa 'hecho en casa'. El desarrollo de aplicaciones no oficiales en la videoconsola Nintendo Wii se realiza con el SDK conocido como **devkitPro**. El kit de desarrollo devkitPro incluye una versión de los compiladores GNU para C y C++ adaptados a la arquitectura PowerPC. También incluye librerías estándar del lenguaje C y un conjunto de librerías de bajo nivel específicas para Wii llamado **libogc** <1>. Una alternativa al desarrollo de aplicaciones para Wii es el Wii Opera SDK (www.wiioperasdk.com), un conjunto de librerías javascript que permite crear aplicaciones web adaptada al navegador Opera de Wii, pudiendo interactuar con los Wiimote. Obviamente estas aplicaciones se ejecutan desde el navegador, por lo que devkitPro es la opción para desarrollar software que se ejecute de forma nativa.

Las aplicaciones no oficiales para Wii se ejecutan forzando diversos fallos en el software de la consola. También es posible ejecutar software no oficial mediante el uso de modificaciones en el hardware conocidas como *modchips*. La legalidad del uso de *modchips* es confusa (en España existen sentencias tanto a favor como en contra)<2>. En cambio, el uso de aplicaciones no oficiales no es ilegal <3>, aunque algunas compañías (como Nintendo) están en su contra por su uso potencial para cargar copias de juegos y suelen tomar medidas para evitarlo, como actualizaciones del *firmware* que eliminan cualquier aplicación no oficial. Por ello es posible que el *homebrew* en Wii no funcione a partir de algunas versiones más recientes del *firmware*.

La forma habitual de ejecutar aplicaciones no oficiales en Wii es mediante el **Homebrew Channel**, una aplicación que se instala como un canal en el menú principal y que permite administrar y ejecutar otras aplicaciones no oficiales. Existen varios métodos para instalar este canal, en el apartado 1.4 se describe cómo hacerlo usando el método conocido como Bannerbomb.

Se debe tener en cuenta que el uso poco responsable de aplicaciones no oficiales puede dañar la consola, incluso inutilizarla. Es muy importante no usar aplicaciones que modifiquen la región de la consola (de PAL a NTFS, etc), ya que es una de las principales causas de daños irreversibles en el sistema<4>. También puede ser dañino instalar versiones modificadas del *firmware*, así como ejecutar aplicaciones que cambien la versión del *firmware* a una más antigua. La instalación de canales no oficiales, salvo el Homebrew Channel, puede ser otra fuente de peligro. Al instalar aplicaciones no oficiales, se anula la garantía, por lo que se pierde la posibilidad de una reparación por parte de Nintendo.

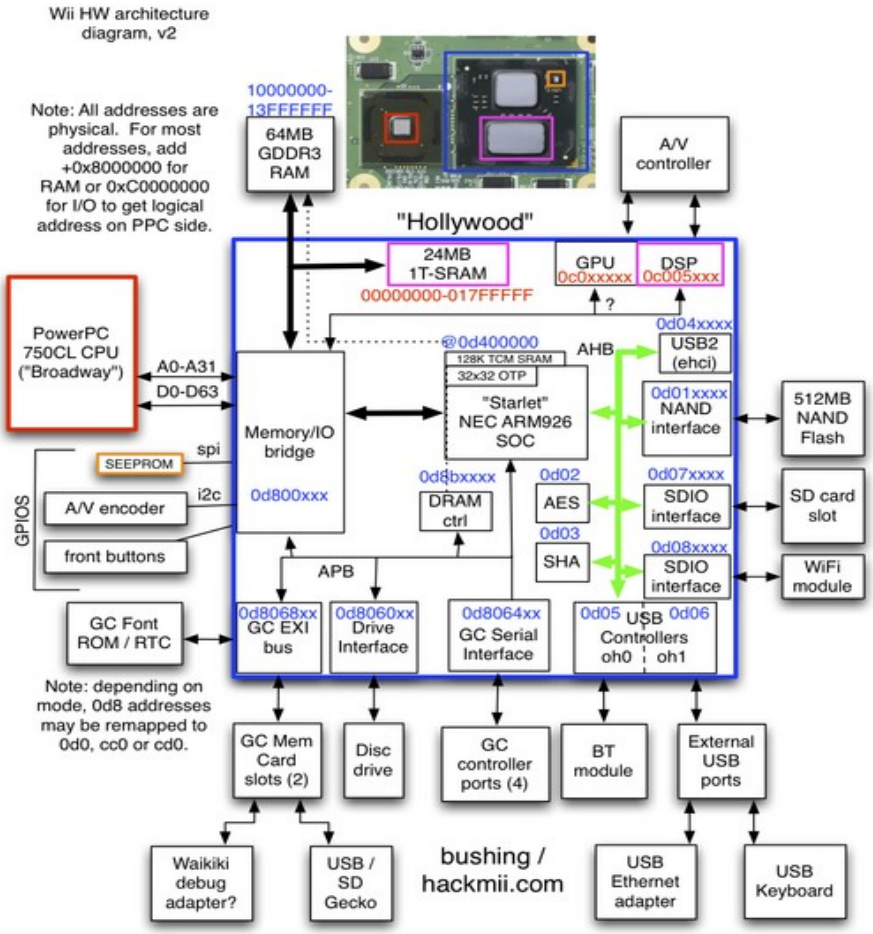
1.3. Descripción de la Nintendo Wii

1.3.1. Hardware interno

El procesador principal de Wii es el IBM PowerPC de 729 Mhz, conocido como *Broadway*. El *Broadway*, que usa tecnología CMOS de 90 nm es el descendiente del procesador PowerPC Gekko de 180 nm, usado en la consola GameCube <5><6>.

La Wii dispone de una GPU diseñada por AMD, conocida cómo Hollywood . El Hollywood es un módulo multi-chip dividido en dos partes, Napa y Vegas. Napa controla las funciones de entrada y salida, el acceso a la RAM y contiene el procesador gráfico junto con su RAM dinámica. Por otro lado, Vegas contiene el procesador de audio (DSP).

Figura 1.2. Esquema del hardware de la Nintendo Wii.



El Hollywood contiene también un procesador ARM926, conocido cómo *Starlet*. El *Starlet* se encarga procesar las funciones de entrada y salida, como la conexión inalámbrica, el lector de DVD, el USB, etc. Este procesador se encarga de ejecutar el sistema operativo IOS y es el primero en funcionar cuando se enciende la consola, ya que se encarga de los procesos de arranque. El sistema operativo IOS, propiedad de Nintendo, está diseñado para sistemas empotrados. Está constituido por un núcleo y una serie de módulos independientes<7>.

Además se dispone de un módulo de memoria RAM GDDR3 de 64 MB y una memoria flash NAND de 512 MB, que almacena los datos del sistema, los canales, las partidas guardadas y la información de configuración de la consola. Otros componentes son dos puertos USB, una ranura para tarjetas de memoria SD, un dispositivo wifi y otro bluetooth, cuatro puertos para mandos de GameCube y dos ranuras para tarjetas de memoria de esta consola<8>. La **figura 1.2** muestra un esquema de todo este hardware.

1.3.2. El sistema de arranque

Como se ha explicado, es el procesador *Starlet* el que se encarga de los procesos de arranque. El sistema de arranque en Wii consta de 3 procesos. El primero, boot0, se encarga de verificar una clave de seguridad única grabada en la consola y lanza el siguiente proceso de arranque. El proceso boot1, que reside en el primer bloque de la memoria NAND, se encarga de lanzar el proceso boot2 que reside el los bloques del 1 al 7 de la NAND. El proceso boot2 se encarga de leer en la NAND y cargar el módulo del IOS responsable de arrancar el System Menu, que se describe más adelante, y pasa el control al procesador *Broadway*. Este proceso boot2 puede ser modificado y utilizado para lanzar aplicaciones no oficiales<9>.

1.3.3. El Wiimote

El Wii Remote, comúnmente conocido como Wiimote, es el principal dispositivo de entrada de la consola y funciona como mando de control para juegos. Se comunica con la consola de forma inalámbrica mediante un dispositivo bluetooth Broadcom BCM2042 integrado.

El Wiimote dispone de nueve botones y un pad direccional o cruceta. El botón principal, llamado botón A, es el de mayor tamaño, situado en la parte superior. El botón B con forma de gatillo, está situado en la parte inferior del mando. El botón Home, situado en el centro de la parte superior del mando, suele usarse para salir al menú, mientras que el resto de botones suelen usarse para funciones secundarias. Un botón especial es el botón Power, que sirve para apagar o encender la consola.

La principal característica del Wiimote es su capacidad de captura de movimientos. Para ello dispone de un acelerómetro que puede detectar movimiento en tres ejes. Adicionalmente, incluye una cámara de infrarrojos en uno de sus extremos con capacidad para rastrear hasta cuatro objetos en movimiento. La barra de sensores incluida con la consola dispone de dos grupos de LEDs infrarrojos que son usados como referencia por la cámara de infrarrojos para obtener información sobre la posición apuntada por el mando.

El Wiimote funciona con dos pilas AA, situadas bajo una tapa en su parte inferior. Bajo esta tapa también se encuentra el botón de sincronización, que permite conectar el mando con la consola. El mando también incluye una memoria interna de 16 kB, que almacena el *firmware* y otra información adicional. En el extremo contrario a la cámara de infrarrojos, se encuentra un puerto para ampliaciones externas, tales como el Nunchuk o el Mando Clásico.<10>

1.3.4. El System Menu

El menú principal, llamado System Menu, es la principal interfaz gráfica de la consola y el primer código en ejecutarse al iniciarse el procesador PowerPC.

El System Menu muestra los canales instalados en la consola, aplicaciones que pueden tener distintas funcionalidades. Desde este menú se pueden arrancar los juegos introducidos en el lector de DVD, abrir el navegador web, etc. Es posible instalar canales no oficiales, cómo el Homebrew Channel. Desde el System Menu también se cambia la configuración de la consola, y se administran los datos de juegos guardados.

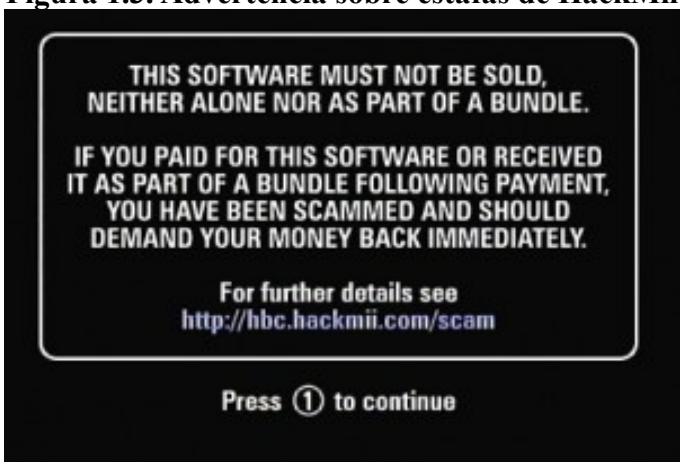
1.4. Instalación del Homebrew Channel y ejecución de aplicaciones caseras

En este apartado se describe cómo instalar el Homebrew Channel, el canal que permite administrar y ejecutar aplicaciones no oficiales de forma sencilla, usando el método conocido como **Bannerbomb**.

Bannerbomb es un *exploit* para Wii que permite la ejecución de un fichero con extensión .dol o .elf desde una tarjeta de memoria SD. Existen dos versiones de Bannerbomb, la v1 para versiones del *firmware* de la 3.0 a la 4.1, y la v2 para la versión del *firmware* 4.2 <11> . La versión del *firmware* en la Wii utilizada para las pruebas realizadas en este proyecto es la 4.2 y por tanto se mostrará como ejemplo la instalación del Homebrew Channel para esta versión.

El primer paso es descargar la versión de Bannerbomb correspondiente a la versión de nuestro *firmware* de la página web oficial, http://bannerbomb.qoid.us/. Una vez descargado, se copia la carpeta "private" en la raíz de una tarjeta SD formateada, junto con un archivo boot.dol o boot.elf que se desee ejecutar. Hasta el momento se sabe que Wii acepta tarjetas de hasta 2 GB. Una vez introducida la tarjeta SD en la ranura para tarjetas de memoria en la Wii, desde el menú principal hay que entrar en el menú Tarjeta SD, situado en la esquina inferior izquierda (en la versión 4.2). Esto hará aparecer una ventana que preguntará si se desea ejecutar el archivo boot.dol o boot.elf.

Figura 1.3. Advertencia sobre estafas de HackMii

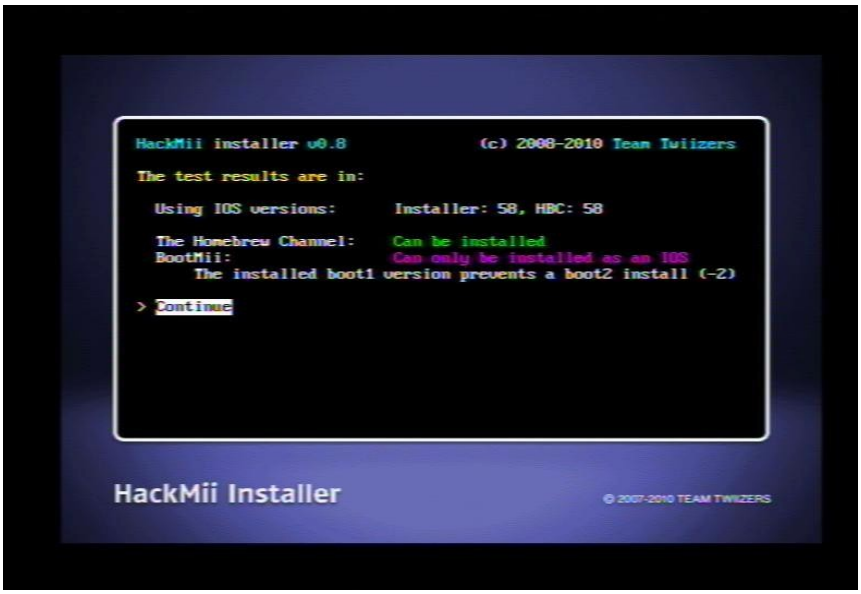


Para instalar el Homebrew Channel, el ejecutable que se usará será el de la aplicación HackMii Installer<12>, que puede descargarse de su página web oficial http://www.bootmii.org.

HackMii Installer es una aplicación que, aprovechando varios fallos de seguridad en la consola, permite instalar diversas utilidades, entre ellas el Homebrew Channel y la aplicación BootMii. La aplicación BootMii puede sustituir el proceso de arranque boot2, permitiendo cargar el Homebrew Channel sin entrar en el menú principal. En la raíz de la tarjeta SD se copia el fichero boot.elf incluido en la descarga, que será el fichero que ejecutara Bannerbomb.

Si todo funciona correctamente aparecerá en pantalla un texto de advertencia sobre posibles estafas (HackMii es gratuito), tras el cual hay que pulsar el botón 1 del Wiimote para continuar (**figura 1.3**) y aparecerá una pantalla donde se pueden observar varios apartados (**figura 1.4**):

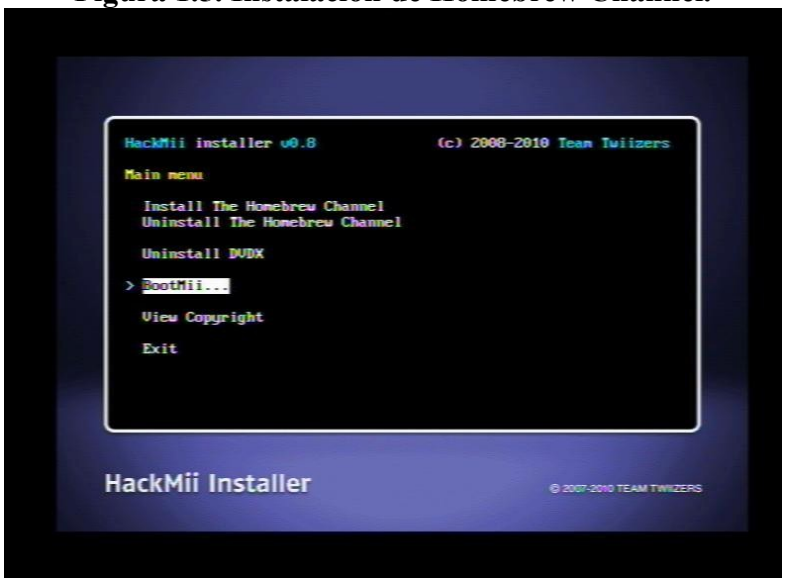
Figura 1.4. Instalación de BootMii y Homebrew Channel



- Using IOS versions: HackMii se ejecuta cómo un módulo del IOS cuyo número aparecerá aquí. Es el mismo módulo que usará el Homebrew Channel en caso de instalarse.
- The Homebrew Channel: Aquí aparecerá un texto indicando si el Homebrew Channel puede ser instalado.
- BootMii: Aquí aparece un texto indicando si la aplicación BootMii puede ser instalada como boot2 o sólo como un módulo del IOS.

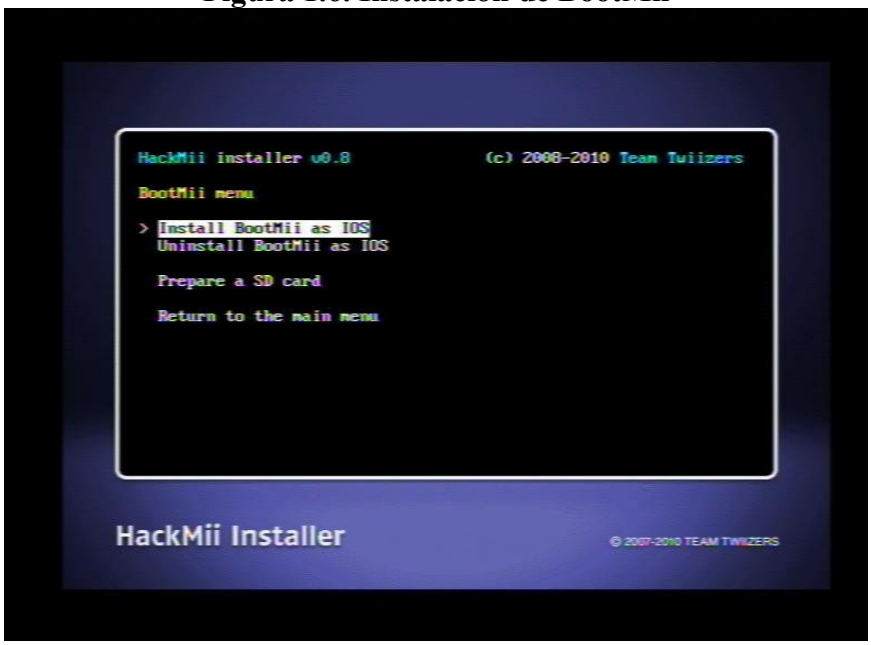
A continuación aparecerá el menú para instalar las utilidades (**figura 1.5**). Para navegar por este menú se usa el pad direccional del Wiimote y el botón A para confirmar la selección. Seleccionando la opción "Install The Homebrew Channel" instalará de forma automática el Homebrew Channel que, en caso de encontrarse ya instalado, se puede desinstalar con la opción "Uninstall The Homebrew Channel".

Figura 1.5. Instalación de Homebrew Channel.



La opción "BootMii..." abre un nuevo menú con las distintas opciones de instalación de la herramienta BootMii (**figura 1.6**). La opción "Install BootMii as boot2", en caso de estar disponible, instalará la aplicación BootMii de forma que se ejecutará al encender la consola. Esta es la opción recomendable. La opción "Install BootMii as IOS" instalará BootMii como un módulo del IOS. Esto es necesario para la ejecución del Homebrew Channel.

Figura 1.6. Instalación de BootMii



La ventaja de instalar BootMii como boot2 es que le permite actuar en lugar del sistema de arranque oficial, teniendo acceso a la consola antes de que se cargue el sistema de ficheros <13>. Si los ficheros necesarios para su ejecución no se encuentran en la tarjeta SD, o la tarjeta SD no se encuentra introducida en la ranura para tarjetas de memoria, el System Menu arrancará de manera habitual. La principal ventaja consiste en que BootMii permite realizar una copia de seguridad de la memoria NAND que puede utilizarse para restaurar el sistema en caso de resultar dañado por alguna aplicación no oficial.

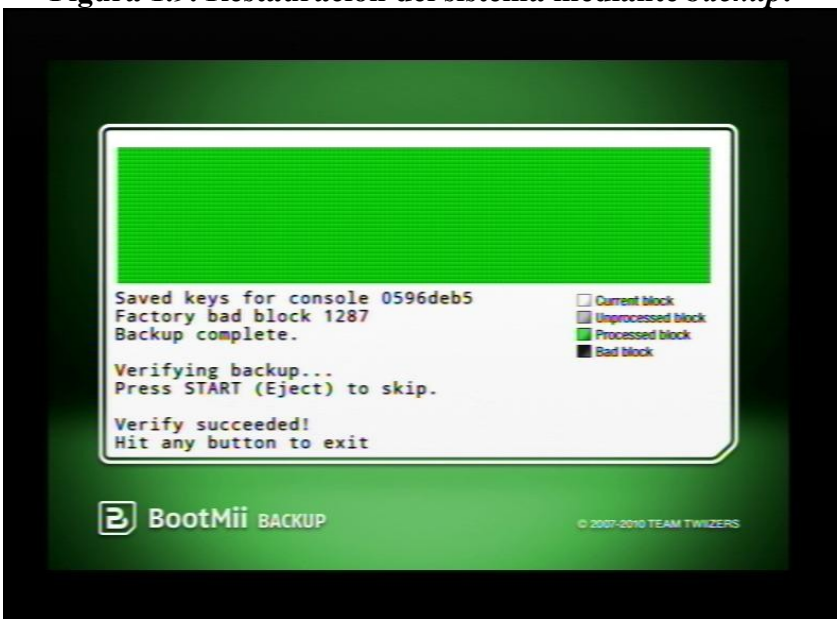
Figura 1.7. Menú de BootMii.

Para navegar por el menú de BootMii se usan los botones del panel frontal de Wii. El botón Power permite navegar por las distintas opciones, mientras que el botón Reset confirma la selección. En la **figura 1.7** se muestran las distintas opciones del menú principal de BootMii. De izquierda a derecha, la primera opción arranca el System Menú, la segunda arranca el Homebrew Channel, la tercera permite navegar por el sistema de ficheros de la tarjeta SD y ejecutar aplicaciones y la cuarta opción abre un nuevo menú que permite realizar una copia de seguridad de la memoria NAND.

Figura 1.8. Menú de restauración.

En el menú de restauración de la **figura 1.8**, la primera opción empezando por la izquierda, permite copiar a la tarjeta SD una imagen del contenido de la memoria NAND, que puede ser restaurada con la segunda opción. Al realizar una copia de seguridad de la memoria NAND, aparecerá una pantalla como la de la **figura 1.9**, donde se mostrará el proceso de copia y verificación.

Figura 1.9. Restauración del sistema mediante *backup*.



Este proceso de copia de seguridad generará dos ficheros nand.bin y keys.bin en la raíz de la tarjeta SD que deben ser conservados en caso de ser necesarios para una posterior restauración. Se recomienda usar una tarjeta de memoria SD de al menos 1 GB para poder realizar todo este proceso.

1.4.1. Aplicaciones en el Homebrew Channel

Una vez instalado el Homebrew Channel, este aparecerá como un canal en el menú principal de Wii (**figura 1.10**). Este canal mostrará de forma gráfica una lista con las aplicaciones no oficiales almacenadas en la tarjeta SD.

Figura 1.10. Inicio del Homebrew Channel.

Las aplicaciones deben guardarse en una carpeta "apps" situada en la raíz de la tarjeta SD. Dentro de este directorio "apps", debe haber una carpeta para cada aplicación, donde se encontrará el ejecutable de la aplicación, que debe tener el nombre "boot.dol" o "boot.elf". Adicionalmente, esta carpeta puede contener una imagen PNG de nombre "icon.png" con una resolución de 128x48 píxeles que se mostrará como icono en la lista del Homebrew Channel, y un fichero XML de

nombre "meta.xml" que permite mostrar información sobre la aplicación y tiene la siguiente forma:

```
<?xml version=”1.0” encoding=”UTF-8” standalone=”yes” ?>
<app version="N° de la versión de la aplicación">
<name>Nombre de la aplicación</name>
<coder>Autor de la aplicación</coder>
<version>N° extendido de la versión</version>
<release_date>Fecha en formato AAAAMMDDhhmmss
(AñoMesDíaHoraMinutoSegundo)</release_date>
<short_description>Descripció la aplicación </short_description>
<long_descrition>Descripción extendida de la aplicación que se
muestra al seleccionar la aplicación en el Homebrew Channel
</long_description>
</app>
```

Para ejecutar una aplicación basta con pulsar sobre ella con el Wiimote en la lista y pulsar en el botón "CARGAR" de la ventana emergente. El Homebrew Channel puede conectarse a Internet automáticamente para mantenerse actualizado y para actualizar la aplicación BootMii. Pulsando el botón "Home" del Wiimote se abre un nuevo menú que permite salir del Homebrew Channel o apagar la consola, tal como aparece en la **figura 1.11**.

Figura 1.11. Menú "Home" del Homebrew Channel.

Capítulo 2. El entorno de desarrollo

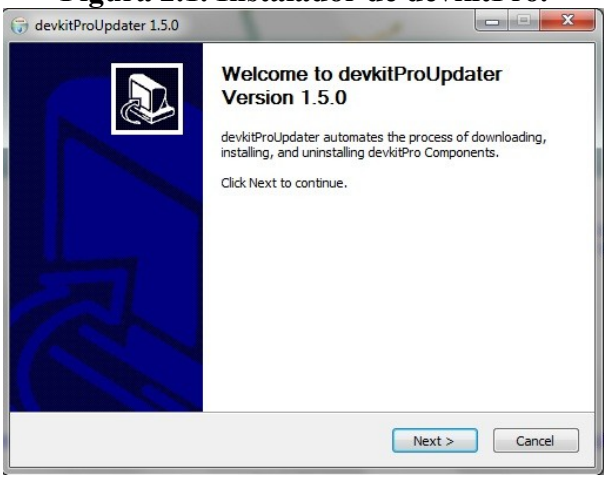
2.1. Introducción

En este capítulo se describe cómo preparar el entorno de desarrollo de software no oficial (*homebrew*) para Wii, así como el proceso de compilación y se presenta un ejemplo de aplicación sencilla. Se describe la realización de estas acciones para un sistema Windows, pero DevkitPro puede usarse igualmente en sistemas Linux u OSX <14>.

2.2. DevkitPro. Instalación

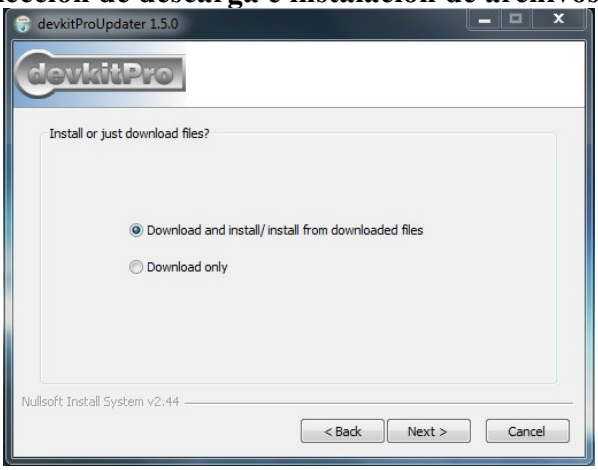
Cómo se ha descrito en el capítulo anterior, el desarrollo de aplicaciones no oficiales para Wii se realiza con el kit devkitPro, cuyo instalador oficial para Windows puede ser descargado de http://sourceforge.net/projects/devkitpro/. La versión de devkitPro utilizada durante el desarrollo de esta guía es la 1.5.0.

Figura 2.1. Instalador de devkitPro.



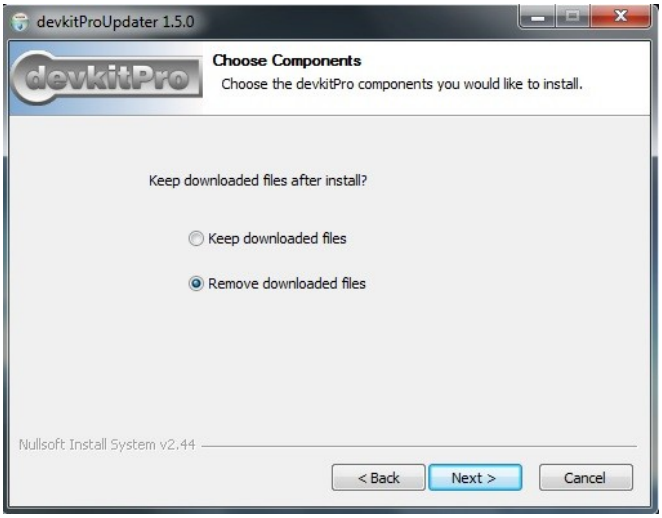
Tras ejecutar el asistente de instalación (**figura 2.1**) y pulsar en "Siguiente" (Next), se pregunta si se desea descargar e instalar los archivos, o sólo descargarlos. En este caso se escoge descargar e instalar (**figura 2.2**).

Figura 2.2. Selección de descarga e instalación de archivos de devkitPro.



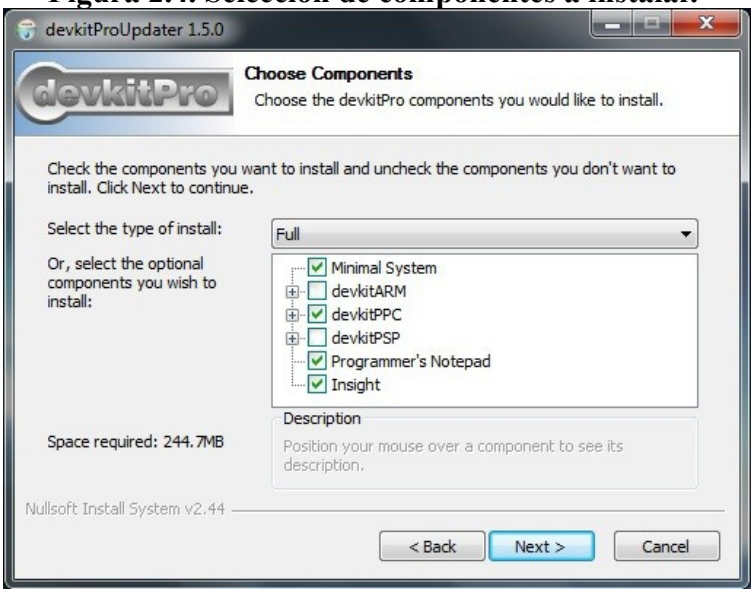
En la siguiente ventana (**figura 2.3**) se elige una de las dos opciones en función de si se quieren conservar los ficheros para la instalación descargados o no.

Figura 2.3. Selección de conservación de los ficheros descargados.



Tras continuar, se deben seleccionar los componentes a instalar (**figura 2.4**). Para el desarrollo en Wii, basta con seleccionar "Minimal System" y "devkitPPC". Minimal System o MSYS es un conjunto de utilidades GNU para Windows tales como bash o make <15>. La opción "Programmer's Notepad" instalará esta aplicación, que es un editor que puede usarse para facilitar la escritura de código.

Figura 2.4. Selección de componentes a instalar.



Por último, hay que elegir el directorio donde se instalarán las librerías y herramientas de devkitPro (**figura 2.5**). El directorio por defecto es "c:\devkitPro" y en esta guía se asumirá que es el directorio elegido.

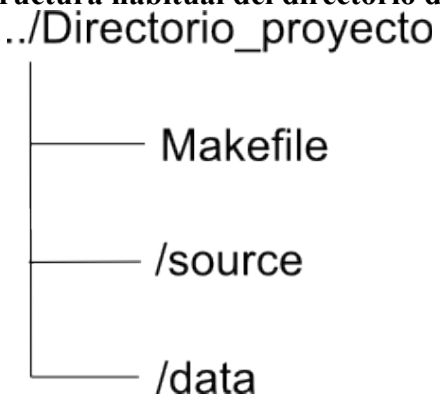
Figura 2.5. Selección del directorio de instalación.

Una vez finalizada la instalación, se habrá creado un directorio "devkitPro" en la raíz de la unidad c: que contendrá un directorio "devkitPPC" y otro "libogc", además de otros correspondientes a herramientas de devkitPro. En el directorio "libogc" se encuentran las librerías que se usarán para el desarrollo en Wii. Para incluir librerías externas ya compiladas para la arquitectura PPC a devkitPro basta con copiar los archivos compilados (.a) en el directorio "lib/wii/" y los archivos de cabeceras (.h) en el directorio"include", ambos dentro del directorio "libogc".

2.3. Compilación. La consola Msys

La estructura del directorio de trabajo en el que se esté desarrollando, si se usan las plantillas de ficheros Makefile incluidas en los ejemplos, debe tener la siguiente forma (**figura 2.6**): un directorio raíz con el nombre del proyecto, en su interior el fichero Makefile, cuyo funcionamiento se describe en el siguiente apartado, y un directorio "source", que contendrá los ficheros de código fuente del proyecto.

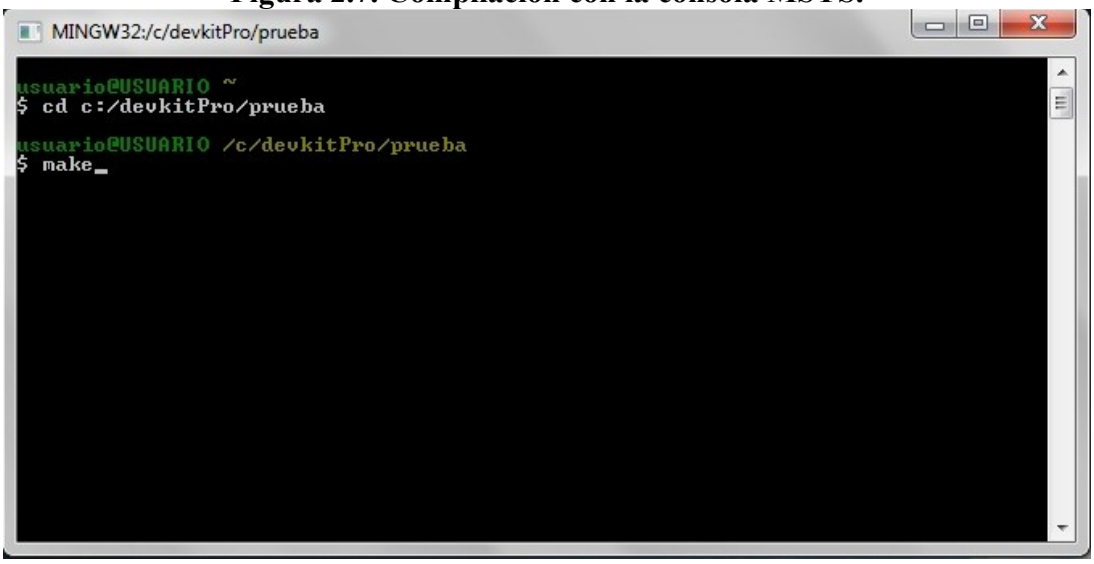
Figura 2.6. Estructura habitual del directorio de un proyecto.



Al compilar, se generará un directorio build, con ficheros generados por el compilador y dos ejecutables, con las extensiones .dol y .elf, con el nombre de la aplicación, que serán los ficheros que se ejecutarán en la Wii (cualquiera de ellos). Hay que recordar que el ejecutable que se vaya a usar debe renombrarse como "boot" al copiarlo en el directorio de la aplicación en la tarjeta SD.

Un método sencillo para compilar es el uso de la consola Msys, que permite utilizar órdenes habituales de sistemas UNIX como cd, make o grep <15>. La consola Msys puede ejecutarse desde el directorio c:\devkitPro\msys\. Para compilar con ella, hay que trasladarse hasta el directorio del proyecto, donde se encuentra el fichero Makefile, mediante la orden cd y ejecutar la orden make (figura 2.7).

Figura 2.7. Compilación con la consola MSYS.



2.4. Ficheros Makefile

Un fichero Makefile es, a grandes rasgos, un fichero que, tras ejecutar la orden make, indica al compilador GNU los pasos a seguir. No es necesario conocer el funcionamiento completo de los ficheros Makefile, pero es conveniente conocer algunos detalles para aprovecharlos. Se pueden encontrar plantillas de ficheros Makefile en los ejemplos incluidos en devkitPro, así como en los ejemplos incluidos en este proyecto.

Por defecto, todos los ficheros Makefile incluidos en devkitPro tratan de compilar los ficheros de código fuente que se encuentren en un directorio "source" y el ejecutable toma el nombre del directorio donde se encuentre el Makefile. Tras compilar, también se genera un directorio "build", que contiene ficheros objeto creados por el compilador. Estos ficheros ayudan a recompilar más rápidamente y pueden ser borrados si se quiere volver a compilar limpiamente, ya que, en ocasiones, pueden producir errores.

La primera línea a tener en cuenta, que debe aparecer en cualquier fichero Makefile para compilar ejecutables para Wii, es la siguiente:

```
include $(DEVKITPPC)/wii_rules
```

que indica al compilador que se está compilando para Wii (el compilador para PowerPC también puede compilar para GameCube). Tras esta línea se encuentra una serie de variables donde se indican ciertos directorios a tener en cuenta durante la compilación. La primera variable, TARGET, es el nombre del ejecutable generado por el compilador. Por defecto, en las plantillas de los ficheros Makefile, este nombre es el del directorio donde se encuentra el Makefile.

La variable BUILD es el nombre del directorio donde se almacenarán los ficheros objeto y otros ficheros intermedios generados por el compilador. Por defecto, el nombre de este directorio es "build" y se encuentra en el directorio del Makefile:

```
BUILD		:=	build
```

La variable SOURCES indica el nombre del directorio donde se encuentran los ficheros de código fuente. Por defecto, el nombre de este directorio es “source”:

```
SOURCES	:=	source
```

Opcionalmente, pueden aparecer otras variables como DATA, que suele usarse para indicar un directorio donde se almacenen imágenes y ficheros de audio que vayan a ser incluidos en el ejecutable. Otra posible variable es INCLUDES, que se usa a para indicar los nombres de los directorios donde se encuentren ficheros de cabeceras adicionales.

La siguiente línea de interés es donde aparece la variable LIBS. En esta variable se especifican las librerías que van a ser usadas en la compilación. Las librerías se indican mediante un prefijo -l y el nombre de la librería, que puede encontrarse en el directorio libogc/lib/wii. Así, la librería libfat.a se incluiría cómo -lfat. En el siguiente ejemplo se indicaría que se van a incluir la librería libmath de C y la librería básica para compilar para Wii, la librería ogc, que contiene las funciones de bajo nivel que usarán el resto de librerías en Wii:

```
LIBS	:= -logc -lm
```

El orden de las librerías es importante. Puede haber dependencia entre librerías que usan funciones de otras, así que, si no están correctamente ordenadas, pueden provocar errores. El orden en el que se indican las librerías en la variable LIBS es el inverso, es decir, las primeras librerías de la lista son las que puedan tener dependencias con las últimas. Por ejemplo, libwiiuse depende de libogc y libbte, luego el orden correcto sería el siguiente:

```
LIBS	:= -lwiiuse -lbte -logc -lm
```

2.5. Tipos de datos específicos en Wii

En Wii pueden usarse los tipos de datos habituales de C pero, además, se añaden otros tipos que son abreviaciones de los tipos de datos habituales. Estos tipos de datos se describen en la siguiente tabla:

Tabla 2.1. Tipos de datos en Wii.

Tipo de dato	Equivalente en Wii	Descripción
char	s8	Carácter. Entero de 8 bits con signo. Rango de -128 a 127.
unsigned char	u8	Entero de 8 bits sin signo. Rango de 0 a 255.
short	s16	Entero de 16 bits con signo. Rango de -32768 a 32767.
unsigned short	u16	Entero de 16 bits sin signo. Rango de 0 a 65535

int	s32	Entero de 32 bits con signo. Rango de -0x80000000 a 0x7fffffff.
unsigned int	u32	Entero de 32 bits sin signo. Rango de 0 a 0xffffffff.
long	s32	Equivalente a int.
unsigned long	u32	Equivalente a int.
long long	s64	Entero de 64 bits con signo. Rango de -0x8000000000000000 a 0x7fffffffffffffff.
unsigned long long	u64	Entero de 64 bits sin signo. Rango de 0 a 0xffffffffffffffff.
float	f32	Número en coma flotante de 32 bits.
double	f64	Número en coma flotante de 64 bits.

Estos tipos se encuentran definidos en C:\devkitPro\libogc\include\gctypes.h. Ambas nomenclaturas son equivalentes y pueden usarse indistintamente.

2.6. Primer ejemplo: "Hello World!"

A continuación se presenta un primer ejemplo explicado de aplicación para Wii, que muestra por pantalla el texto "Hello World!". Este ejemplo puede encontrarse en los anexos adjuntos a esta memoria como **ejemplo 1**.

El primer paso es incluir las librerías necesarias. En este caso se incluyen las librerías estándar de C y las librerías gccore, que contiene funciones de bajo nivel básicas para el funcionamiento de las aplicaciones en Wii y de la que dependen el resto de librerías, y wiiuse, que contiene las funciones para el manejo del mando de Wii, el Wiimote, que se explica en el capítulo 3.

```
#include <stdio.h>
#include <stdlib.h>
#include <gccore.h>
#include <wiiuse/wpad.h>
```

A continuación se definen dos variables globales necesarias para el funcionamiento del vídeo y una función que encapsula todas las funciones de inicialización, tanto del sistema de vídeo como de la librería wiiuse.

```
static void *xfb = NULL;
static GXRModeObj *rmode = NULL;

void funcion_inicio(){
     //Inicializa el sistema de vídeo.
     VIDEO_Init();
```

```
        //Inicializa los controles.
        WPAD_Init();

        //Obtiene el modo de video determinado en la configuración de
        //la consola
        rmode = VIDEO_GetPreferredMode(NULL);

        // Aloja el buffer en memoria
        xfb = MEM_K0_TO_K1(SYS_AllocateFramebuffer(rmode));

        // Inicializa la consola, necesario para la función printf
        console_init(xfb,0,0,rmode->fbWidth,rmode->xfbHeight,
                        rmode->fbWidth*VI_DISPLAY_PIX_SZ);

        // Configura el vídeo con el modo predeterminado
        VIDEO_Configure(rmode);

        // Le dice al hardware de video donde se encuentra el buffer.
        VIDEO_SetNextFramebuffer(xfb);

        // Vuelve visible la escena.
        VIDEO_SetBlack(FALSE);

        // Limpia los registros de vídeo del hardware
        VIDEO_Flush();

        // Espera a que termine la configuración del vídeo.
        VIDEO_WaitVSync();
        if(rmode->viTVMode&VI_NON_INTERLACE) VIDEO_WaitVSync();
}
```

Tras esto, se define la función main.

```
int main(int argc, char **argv) {

        funcion_inicio();
```

La consola acepta codigos de escape de terminal VT <16>. La siguiente línea coloca el cursor en la fila 2, columna 0.

```
printf("\x1b[2;0H");
```

En general, el cursor puede desplazarse con el siguiente código: printf("\x1b[%d;%dH", fila, columna). A continuación se imprime la línea "Hello World!":

```
        printf("Hello World!");
```

El siguiente bucle refresca la pantalla y espera a que se pulse el botón "Home" del Wiimote para salir del programa:

```
while(1){

            WPAD_ScanPads();
```

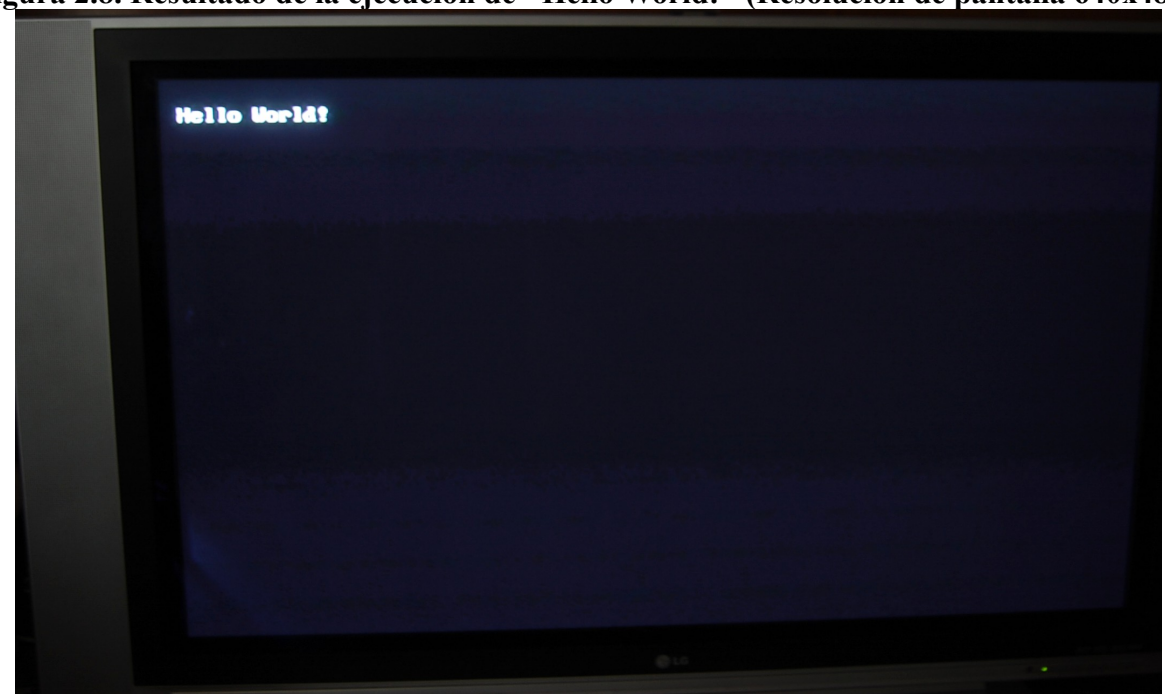
```
            u32 pressed = WPAD_ButtonsDown(0);

            if( pressed & WPAD_BUTTON_HOME) exit(0);

            VIDEO_WaitVSync();
        }

        return 0;
    }
```

Todas las funciones correspondientes al control de los mandos (aquellas con el prefijo WPAD) se describen en el capítulo 3.

Para compilar este ejemplo, puede usarse el fichero Makefile que se incluye como plantilla en el anexo adjunto a la memoria. El fichero con el código debe almacenarse en un directorio "source" que se encuentre en el mismo directorio que el Makefile. Con la consola Msys, hay que desplazarse hasta la ubicación del fichero Makefile, supóngase que es el directorio "hello_world" y ejecutar la orden "make", tras lo cual el compilador generará un directorio "build", un fichero hello_world.dol y otro fichero hello_world.elf.

Tras esto, se copia uno de los ficheros ejecutables renombrado "boot" en una carpeta dentro del directorio apps de la tarjeta de memoria SD. En el Homebrew Channel aparecerá un botón con el nombre de la aplicación o la ruta de ubicación del ejecutable, si no se ha incluido el fichero meta.xml. Al pulsar sobre el botón, se ejecutará la aplicación, que imprimirá el texto "Hello World!" en letras blancas sobre una pantalla negra.

Figura 2.8. Resultado de la ejecución de "Hello World!" (Resolución de pantalla 640x480).



Capítulo 3. Manejo de los controles

3.1. Introducción. La librería wiiuse

En este capítulo se describe como utilizar los controles de Wii, leer la información de entrada del Wiimote y sus extensiones.

Para controlar el Wiimote, se hará uso de la librería wiiuse, que contiene las estructuras de datos donde se almacena la información de los periféricos, así como una serie de constantes usadas como flags. Para usar esta librería es necesario incluir el fichero de cabeceras wpad.h, que contiene las funciones necesarias para el control del wiimote.

```
#include <wiiuse/wpad.h>;
```

En el fichero Makefile del proyecto se debe añadir -lwiiuse en la variable LIBS.

3.2. Inicialización del wiimote

Para inicializar la librería, se debe usar la función **WPAD_init()** al inicio del programa. Por el contrario, se puede usar la función **WPAD_Shutdown()** para salir de la librería antes de terminar la ejecución del programa.

Se dispone de cuatro canales para los cuatro Wiimotes que pueden estar conectados a la vez, numerados del 0 al 3. Así, el primer mando conectado se corresponde con el canal 0, el canal 1 para el segundo mando, etc.

Para establecer el formato en el que se deseen obtener los datos se usa la función **WPAD_SetDataFormat** cuya sintaxis es la siguiente:

```
WPAD_SetDataFormat(chan, fmt);
```

Donde `chan` es el canal del wiimote y `fmt` es el modo de operación. Los modos de operación se describen en la siguiente tabla:

Tabla 3.1. Modos de operación del Wiimote.

Modo de operación	Descripción
WPAD_FMT_BTNS	Modo por defecto, se corresponde sólo a los botones.
WPAD_ FMT_BTNS_ACC	Botones y acelerómetro.
WPAD_FMT_BTNS_ACC_IR	Botones, acelerómetro e infrarrojos.

Por otro lado, los canales disponibles son los mostrados en la tabla 3.2 :

Tabla 3.2. Constantes para los canales del Wiimote.

Canal	Descripción
WPAD_CHAN_0	Constante para el canal 0.

WPAD_CHAN_1	Constante para el canal 1.
WPAD_CHAN_2	Constante para el canal 2.
WPAD_CHAN_3	Constante para el canal 3.
WPAD_CHAN_ALL	Constante para todos los canales.

Así por ejemplo, si se desea obtener información de los infrarrojos de todos los Wiimotes, se usaría la función de la siguiente forma:

```
WPAD_SetDataFormat(WPAD_CHAN_ALL, WPAD_FMT_BTNS_ACC_IR);
```

Las constantes para cada canal tienen el valor numérico correspondiente al número del canal. Por ejemplo, la constante WPAD_CHAN_0 vale cero. Por lo tanto, es más sencillo indicar el canal con su valor numérico. En la siguiente linea se asigna el uso de infrarrojos al canal 0:

```
WPAD_SetDataFormat( 0, WPAD_FMT_BTNS_ACC_IR);
```

Es conveniente hacer un uso inteligente de esta función y ceñirse solo a los canales y las funcionalidades que vayan a ser necesarias para la aplicación, ya que pueden llevar a un gasto innecesario de batería en el Wiimote.

Se puede establecer el tiempo por defecto tras el cual los mandos se apagarán automáticamente, mediante la función **WPAD_SetIdleTimeout**, cuya sintaxis es:

```
WPAD_SetIdleTimeout(seconds);
```

Donde `seconds` son los segundos tras los cuales todos los mandos conectados se apagarán si ninguno ha sido usado. Es posible desconectar cada mando por separado con la función **WPAD_Disconnect(chan)**, donde **chan** es el canal del mando que se quiera desconectar.

Por último, si se está usando la cámara de infrarrojos, es necesario indicar la resolución a la que se desee ajustar la cámara de infrarrojos. Para ello se usa la función WPAD_SetVRes cuya sintaxis se muestra a continuación:

```
WPAD_SetVRes(chan, xres, yres);
```

Donde `chan` es el canal del wiimote al que queremos ajustar la resolución, `xres` es la resolución horizontal e `yres` la resolución vertical. Así, si queremos ajustar los infrarrojos de todos los mandos a una resolución de 640x480, usaremos la función como se muestra a continuación:

```
WPAD_SetVRes(WPAD_CHAN_ALL, 640, 480);
```

3.3. Detección de botones

Para leer los datos de todos los mandos conectados, se usa la función **WPAD_ScanPads()**. Esta función debe emplearse cada vez que se quiera comprobar el estado de los mandos y los botones que estén siendo pulsados.

Se dispone de tres funciones para detectar los eventos de botón. La función **WPAD_ButtonsHeld(chan)** indica si un botón (cualquiera, tanto del wiimote como de las expansiones) se encuentra pulsado, la función **WPAD_ButtonsDown(chan)** informa si se produce el evento de que un botón acaba de ser pulsado y la función **WPAD_ButtonsUp(chan)** indica si un botón acaba de ser liberado, siendo el parámetro **chan** es el canal del mando donde se desee comprobar el evento. Las tres funciones devuelven un entero mayor que cero si se ha producido uno de los respectivos eventos, o cero en caso contrario.

Para averiguar si el evento se produce en un botón concreto, en el fichero de cabeceras C:\devkitPro\libogc\wiiuse\wpad.h se definen las constantes con el código para cada botón, tanto del Wiimote como de sus expansiones. La comprobación se realizaría como se muestra en el siguiente ejemplo:

```
if(WPAD_ButtonsDown(0) & WPAD_BUTTON_HOME) exit(0);
```

En el ejemplo anterior, se comprueba si algún botón a sido pulsado y, si ese botón es el botón "Home" del Wiimote, la aplicación termina.

Tabla 3.3. Constantes para los botones del Wiimote.

Botón del Wiimote	Constante de wpad.h	Icono
Botón 2	WPAD_BUTTON_2	
Botón 1	WPAD_BUTTON_1	
Botón B	WPAD_BUTTON_B	
Botón A	WPAD_BUTTON_A	
Botón "menos" o "-"	WPAD_BUTTON_MINUS	
Botón "Home"	WPAD_BUTTON_HOME	
Dirección izquierda del pad direccional o cruceta (con el Wiimote en posición vertical)	WPAD_BUTTON_LEFT	
Dirección derecha del pad direccional o cruceta (con el Wiimote en posición vertical)	WPAD_BUTTON_RIGHT	
Dirección abajo del pad direccional o cruceta (con el Wiimote en posición vertical)	WPAD_BUTTON_DOWN	
Dirección arriba del pad direccional o cruceta (con el Wiimote en posición vertical)	WPAD_BUTTON_UP	

Botón "más" o "+"	WPAD_BUTTON_PLUS	

3.4. Lectura de los infrarrojos

La librería wiiuse almacena los datos de los infrarrojos en una estructura ir_t, definida en C:\devkitPro\libogc\include\wiiuse\wiiuse.h. Para leer la información de la cámara de infrarrojos, se debe declarar una estructura de este tipo, que se pasará como parámetro a la función **WPAD_IR**, que tiene la siguiente sintaxis:

```
WPAD_IR(chan, ir);
```

Donde `chan` es el canal del mando del que se desea obtener los datos e `ir` es un puntero a una estructura ir_t donde se almacenarán los datos que se lean de los infrarrojos en ese momento.

Una vez almacenados los datos en una estructura ir_t, se puede obtener la posición de la pantalla a donde apunte el Wiimote. Suponiendo que se ha usado una estructura ir_t de nombre 'ir', en ir.x se encuentra almacenada la coordenada horizontal respecto a la resolución que se haya indicado con la función **WPAD_SetVRes**, mientras que en ir.y se encuentra la coordenada vertical. También puede obtenerse el ángulo (en grados) de rotación del Wiimote respecto a la barra de sensores en ir.angle y la distancia en metros a la barra de sensores en ir.z. Por otro lado, la variable ir.valid indica, mediante un valor booleano, si el Wiimote se encuentra apuntando a la pantalla, con un valor *true* en caso afirmativo o *false* en caso negativo. Todos son valores en coma flotante.

3.4.1 Información adicional del Wiimote. Orientación y aceleración

Gracias al acelerómetro integrado en el Wiimote, es posible obtener información sobre su orientación y, evidentemente, su aceleración.

Para obtener los datos de aceleración, primero debe crearse una estructura de tipo vec3w_t. Esta estructura simula un vector de tres dimensiones, con información sobre la aceleración en los ejes x, y, z. El siguiente paso es usar la función **WPAD_Accel**, que tiene la siguiente sintaxis:

```
WPAD_Accel(chan, vector);
```

Donde `chan` es el canal del Wiimote, y `vector` es un puntero a una estructura vec3w_t. Tras esto, en la estructura vec3w_t se encontrará almacenada la información sobre la aceleración. Suponiendo que se ha creado una estructura vec3w_t de nombre 'acel', pueden usarse las expresiones acel.x, acel.y y acel.z para acceder al valor de la aceleración en x, y o z, respectivamente. Todas son variables de 32 bits sin signo.

Para obtener datos de la orientación del Wiimote, primero debe crearse una estructura de tipo orient_t, que se pasará como parámetro a la función **WPAD_Orientation**, cuya sintaxis es:

```
WPAD_Orientation(chan, orient);
```

Donde `chan` es el canal del Wiimote y `orient` es un puntero a una estructura de tipo orient_t. Una vez almacenada la información, se puede acceder a las distintas variables de la estructura orient_t para obtener los datos. Suponiendo que se ha creado una estructura orient_t de

nombre 'orient', la variable orient.roll indica el ángulo (en grados) de rotación del Wiimote a lo largo de su eje longitudinal. La variable orient.pitch indica el ángulo (en grados) de inclinación del Wiimote respecto a su posición horizontal. La variable orient.yaw indica el ángulo horizontal (en grados) que forma el Wiimote con respecto a la pantalla. Todas son variables en coma flotante.

3.4.2. Obtención de datos del Wiimote de forma general. La función WPAD_Data

Es posible obtener la información del Wiimote (orientación, aceleración o posición respecto a los infrarrojos) de forma general mediante la función **WPAD_Data**. Esta función recibe cómo parámetro el canal del Wiimote a usar y devuelve un puntero a una estructura de tipo WPADData. Esta estructura almacena otras estructuras de tipo vec3w_t para la aceleración, orient_t para la orientación, ir_t para la información de los infrarrojos y una estructura de tipo expansion_t para los datos de la expansión conectada.

Suponiendo que se crea un puntero a una estructura WPADData llamada wd, se accedería a estas estructuras con expresiones como wd->accel para la estructura vec3w_t, wd->orient para la estructura orient_t, wd->ir para la estructura ir_t y wd->exp para la estructura expansion_t. Los datos de cada una de estas estructuras serían accedidos de la forma habitual. Por ejemplo, la expresión wd->ir.x indicaría la posición en el eje x del Wiimote con respecto a los infrarrojos.

3.5. Control de la vibración del Wiimote

Es posible controlar la función de vibración del Wiimote mediante la función WPAD_Rumble. Esta función recibe los parámetros cómo se muestra a continuación:

WPAD_Rumble(chan, vibra);

donde el parámetro chan determina el canal del mando y el parámetro vibra activa la vibración en caso de valer 1 o la desactiva en caso de valer 0.

3.6. Uso de extensiones

Los datos de las expansiones del Wiimote se almacenan en varias estructuras definidas en wiiuse.h. La estructura para la extensión más común, el Nunchuk, se define cómo una estructura de tipo nunchuk_t. Las estructuras para el resto de expansiones se describen en la siguiente tabla.

Tabla 3.4. Estructuras para las expansiones del Wiimote.

Tipo de expansión	Descripción
nunchuk_t	Estructura para el nunchuk.
classic_ctrl_t	Estructura para el mando clásico.
guitar_hero_3	Estructura para la guitarra del juego Guitar Hero 3.
wii_boar_t	Estructura para la Wii Board.
motion_plus_t	Estructura para el Motion Plus.

Existe una estructura, expansion_t, donde se almacena la información sobre cual de las expansiones se encuentra conectada actualmente al wiimote.

```
typedef struct expansion_t {
      int type;
      union {
            struct nunchuk_t nunchuk;
            struct classic_ctrl_t classic;
            struct guitar_hero_3_t gh3;
            struct wii_board_t wb;
            struct motion_plus_t mp;
      }; } expansion_t;
```

Esta estructura almacena un entero `type`, que indica el tipo de expansión, y una estructura de datos correspondiente al tipo de expansión conectado. Las constantes para cada tipo de expansión se describen en la siguiente tabla:

Tabla 3.5. Constantes para cada tipo de expansión.

Tipo de expansión.	Descripción.
EXP_NONE	Valor de type cuando no hay ninguna expansión conectada.
EXP_NUNCHUK	Valor de type cuando se encuentra conectado el nunchuk.
EXP_CLASSIC	Valor de type cuando se encuentra conectado el mando clásico.
EXP_GUITAR_HERO_3	Valor de type cuando se encuentra conectada la guitarra del juego Guitar Hero 3.
EXP_WII_BOARD	Valor de type cuando se encuentra conectada la Balance Board.
EXP_MOTION_PLUS	Valor de type cuando se encuentra conectada la expansión Motion Plus.

La función **WPAD_Expansion**, cuya sintaxis es:

```
WPAD_Expansion(chan, exp);
```

Donde `chan` es el canal del Wiimote y `exp` es un puntero a una estructura de tipo expansion_t. Esta función asocia a la estructura `exp` de tipo **expansion_t** los datos de la expansión conectada al wiimote del canal `chan`. Al igual que la función **WPAD_ScanPads**, esta función debe llamarse para cada nueva comprobación. Una vez almacenada la estructura, se puede acceder a sus datos internos para comprobar, por ejemplo, que botones de la expansión están siendo pulsados. Por ejemplo, para comprobar si se ha pulsado el botón C del nunchuk conectado al wiimote del canal 0, se utilizarían las siguientes instrucciones:

```
expansion_t exp;
WPAD_Expansion(0, &exp);
if(exp.type == EXP_NUNCHUK){
      if(exp.nunchuk.btns_held & NUNCHUK_BUTTON_C){
            //Acciones a realizar.} }
```

La variable `exp.nunchuk.btns_held` valdrá 1 si se ha presionado un botón cualquiera del Nunchuk, `exp.nunchuk.btns_released` valdrá 1 si un botón ha sido liberado. La variable `exp.nunchuk.btns_last` contendrá el valor constante del último botón pulsado.

La estructura nunchuk_t también contiene información similar a la del Wiimote, ya que incluye otro acelerómetro. Siguiendo con el ejemplo anterior, en exp.nunchuk.accel se encuentra almacenada una estructura de tipo vec3w_t con información sobre la aceleración del Nunchuk, mientras que, en exp.nunchuk.orient, se encuentra almacenada una estructura orient_t con información sobre su orientación.

Tabla 3.6. Constantes para los botones del Nunchuk.

Botón del Nunchuk	Constante de wpad.h	Icono
Botón C	NUNCHUK_BUTTON_C	
Botón Z	NUNCHUK_BUTTON_Z	

Adicionalmente, la estructura nunchuk_t contiene otra estructura de tipo joystick_t, con información sobre la posición del joystick del Nunchuk. En el caso del ejemplo anterior, se accedería a esta estructura mediante la expresión exp.nunchuk.js. En js.ang, se almacena el ángulo de posición del joystick, siendo 0 el valor para la posición superior, 90 para la derecha, 180 para abajo y 270 para la izquierda.

Obviamente, también puede usarse la función WPAD_Data para comprar si el Nunchuk se encuentra conectado. El siguiente ejemplo muestra cómo hacerlo:

```
WPADData *wd;
bool nunchuk_conectado = false;
wd = WPAD_Data(0);
if(wd->exp.type == EXP_NUNCHUK){
      nunchuk_conectado = true;
}
```

3.7. Ejemplos adjuntos a la memoria relativos al capítulo 3

Ejemplo 2: Muestra cómo obtener datos del Wiimote, tales como la orientación, la aceleración y la posición del puntero de infrarrojos, así como los botones que se estén pulsando y los imprime en pantalla. También detecta si se encuentra conectado el Nunchuk y, en caso de estarlo, muestra en pantalla la misma información sobre esta expansión.

Capítulo 4. El sistema de ficheros

4.1. Introducción

En este capítulo se describe cómo acceder al sistema de ficheros de la tarjeta de memoria SD para usar las funciones más habituales de manejo de ficheros de la librería estándar de C, stdio. También se describe cómo leer y escribir ficheros XML con la librería mxml.

4.2. Inicializando el sistema de ficheros. La librería libfat

Para poder acceder al sistema de ficheros en Wii, es necesario usar la librería libfat. Para ello, hay que añadir la sentencia `#include <fat.h>` en el código fuente y -lfat a la variable LIBS del fichero Makefile.

La función **fatInitDefault()** inicializa el sistema de ficheros por defecto, el de la tarjeta de memoria SD, devolviendo el valor booleano *false* en caso de producirse algún error o el valor *true* en caso contrario. De esta forma, es posible verificar la correcta inicialización del sistema de ficheros para evitar errores en la aplicación. Es necesario llamar a esta función antes de usar cualquier otra función que opere con ficheros.

4.3. Funciones habituales de acceso a ficheros

A continuación se describen las funciones de uso más habitual para manejar ficheros. Estas funciones forman parte de la librería estándar de C, stdio, que debe incluirse en el código fuente mediante la sentencia `#include <stdio.h>`.

4.3.1. Apertura de un fichero

Para abrir un fichero ya existente o crear uno nuevo se utiliza la función fopen, que devuelve un puntero a una variable FILE. Su sintaxis general es:

```
fopen(nombreFichero, modo);
```

Donde `nombreFichero` es una cadena con el nombre del fichero y, opcionalmente, su ruta. Si no se especifica la ruta, se supondrá que el fichero se encuentra en el directorio local. Por ejemplo, "sd:/texto.txt" indica que el fichero se encuentra en el directorio raíz de la tarjeta SD. En cambio, la cadena "texto.txt" indicaría que el fichero se encuentra en el directorio local donde se encuentre el ejecutable. Si la cadena fuera "datos/texto.txt", indicaría que el fichero se encuentra en un directorio "datos" que se encontraría en el directorio local.

El parámetro `modo` es una cadena que contiene una serie de caracteres que configuran el modo de apertura o creación del fichero. En la tabla 4.1 se indican los modos más habituales.

Tabla 4.1. Modos de apertura de ficheros.

Modo	Descripción
"r" o "rt"	Modo de lectura en modo texto. En este caso, si el fichero no existe la función devuelve NULL.

"w" o "wt"	Modo de escritura en modo texto. Si el fichero no existe, crea uno nuevo y lo deja abierto para escribir en él y, si ya existe, lo sobrescribe
"a" o "ar"	Modo append en modo texto. Si el fichero no existe, crea uno nuevo y lo deja abierto para escribir en él y, si ya existe, permite escribir al final de este, respetando los datos anteriores.
"rb"	Igual que "r", pero para escribir datos en binario.
"wb"	Igual que "w", pero para escribir datos en binario.
"ab"	Igual que "a", pero para escribir datos en binario.

Tras el uso de la función fopen, es conveniente comprobar si el fichero se ha abierto correctamente. Esto podría hacerse con una serie de sentencias como la siguiente:

```
FILE *fichero;
fichero = fopen("texto.txt", "r");
if(fichero == NULL){
     /*Acciones a realizar en caso de no abrirse correctamente
     "texto.txt".*/
}
```

4.3.2. Cierre de ficheros

Es conveniente cerrar los ficheros cuando dejen de ser usados o se podrían alterar los datos por error, provocando futuros fallos imprevisibles. Para cerrar un fichero se utiliza la función **fclose**, que se usa como se muestra a continuación:

```
fclose(punteroFichero);
```

Donde `punteroFichero` es el identificador de una variable puntero a FILE, que almacena la dirección del fichero abierto. Si el fichero apuntado por `punteroFichero` no existe, la función fclose devolverá el valor NULL.

4.3.3. Lectura de ficheros

Para leer ficheros se utiliza la función se utiliza la función **fread**, cuya sintaxis es la siguiente:

```
fread(punteroDato, tamanyo, veces, punteroFichero);
```

Donde `punteroDato` es la dirección de la variable o elemento donde se quiera almacenar los datos leídos del fichero indicado por la variable apuntada por `punteroFichero`. En `tamanyo` se indica el tamaño en bytes de que se quieran leer del fichero, normalmente el tamaño del elemento apuntado por `punteroDato`, mientras que el parámetro `veces` indica cuantos elementos de tamaño `tamanyo` se van a leer. Una forma de obtener el tamaño de una variable es mediante la función sizeof, cuya sintaxis es:

```
sizeof(elemento);
```

La función sizeof devolvería el tamaño en bytes de `elemento`.

4.3.4. Escritura de ficheros

Para escribir en un fichero se utiliza la función fwrite, cuya sintaxis es idéntica a la de la función fread:

```
fwrite(punteroDato, tamanyo, veces, punteroFichero);
```

De nuevo, `punteroDatos` es la dirección de la variable o elemento que, en este caso, se quiera escribir. El parámetro `tamanyo` es el tamaño en bytes del elemento apuntado por `punteroDato`, mientras que veces indica cuantos elementos de tamaño `tamanyo` van a ser escritos. Por último, `punteroFichero` es la variable puntero a FILE que hace referencia al fichero sobre el que se va a escribir.

4.4. Operaciones con ficheros XML. La librería mxml

XML es un estándar de definición de lenguajes de marcado, que permite el almacenamiento de datos estructurados. En un fichero XML, la información se organiza mediante etiquetas. Cada etiqueta define un elemento, que puede estar contenido dentro de otro elemento (padre) y que puede contener otros elementos (hijos). Cada elemento puede contener datos en forma de texto y puede tener varios atributos. Un ejemplo de un documento XML es el fichero "meta.xml" que se puede incluir en la carpeta de la aplicación para configurar su apariencia en el Homebrew Channel:

```
<?xml version=”1.0” encoding=”UTF-8” standalone=”yes” ?>
<app version="N° de la versión de la aplicación">
<name>Nombre de la aplicación</name>
<coder>Autor de la aplicación</coder>
<version>N° extendido de la versión</version>
<release_date>Fecha en formato AAAAMMDDhhmmss
(AñoMesDíaHoraMinutoSegundo)</release_date>
<short_description>Descripción la aplicación </short_description>
<long_descrition>Descripción extendida</long_description></app>
```

El uso del estándar XML está ampliamente extendido y permite compartir información de forma sencilla entre distintas plataformas. En el caso del desarrollo de aplicaciones en Wii, XML permite almacenar información de forma ordenada y fácil de comprender, además de permitir que el usuario pueda personalizar de forma sencilla la configuración de la aplicación. El ejemplo expuesto anteriormente, es un caso en el que el usuario (o el creador de una aplicación) puede personalizar la información que el Homebrew Channel muestra sobre una aplicación concreta. El uso de las etiquetas hace que sea sencillo saber que información se está modificando, sin tener que conocer cómo trabaja la aplicación que va a leer el fichero (el Homebrew Channel en este caso).

Para trabajar con ficheros XML se va a usar la librería mxml, abreviatura de "minimal-xml", en su versión adaptada a PowerPC. Esta librería contiene las funciones básicas para leer y escribir ficheros XML. Puede descargarse de http://sourceforge.net/projects/devkitpro/files/portlibs/mxml-2.6-ppc.tar.bz2/download.

La librería mxml trabaja con nodos, definidos por la estructura **mxml_node_t**. Cada nodo puede tener un padre y varios hijos. El nodo padre será el elemento que lo contenga, y los nodos hijos, los elementos contenidos. De esta forma, el documento XML se define como un árbol de nodos, siendo la raíz el nodo que se corresponde con la totalidad del documento, y sus hijos los nodos correspondientes a los elementos XML que contenga. Además, la estructura mxml_node_t, alberga el contenido de cada elemento y sus atributos. Al trabajar con la librería mxml será necesario definir uno o varios punteros a estructuras mxml_node_t.

4.4.1. Escritura en un fichero XML

4.4.1.1 Creación de un fichero XML nuevo

El primer paso para crear un fichero XML nuevo es usar la función **mxmlNewXML** que devuelve un puntero a una estructura mxml_node_t. El nodo representado por esta estructura será el nodo raíz del documento XML, y será el padre de todos los demás nodos. La sintaxis de la función mxmlNewXML es la siguiente:

```
nodo_raiz = mxmlNewXML(version);
```

Donde `version` es una cadena con la versión del estándar XML usada. Si `version` es NULL, la versión por defecto será la 1.0. La variable `nodo_raiz` es un puntero a una estructura mxml_node_t que se corresponde con el nodo raíz y es el padre del resto de nodos del documento.

4.4.1.2. Creación de un nuevo elemento

Para crear un nuevo elemento en el documento XML se utiliza la función **mxmlNewElement** que devuelve un puntero al nodo creado. La sintaxis de la función es la siguiente:

```
nodo = mxmlNewElement(nodo_padre, nombre);
```

Donde `nodo_padre` es el nodo donde se quiere contener el nuevo nodo y `nombre` es una cadena con el nombre del elemento. La variable `nodo` es un puntero al nodo creado.

La función **mxmlSetElement**, permite cambiar el nombre de un elemento ya existente. La función requiere dos parámetros, siendo el primero un puntero al nodo que se quiere modificar y el segundo, una cadena con el nuevo nombre del elemento.

4.4.1.3. Atributos

Para añadir atributos a un elemento se utiliza la función **mxmlElementSetAttr** cuya sintaxis es la siguiente:

```
mxmlElementSetAttr(nodo, nombre_atributo, valor);
```

El parámetro `nodo` es un puntero al nodo al que se va a añadir el nuevo atributo. El parámetro `nombre_atributo` es una cadena con el nombre del nuevo atributo y el parámetro `valor` es una cadena con el contenido de dicho atributo.

Si se quiere dar formato a la cadena del contenido del nuevo atributo, se puede utilizar la función **mxmlElementSetAttrf**, que tiene la siguiente sintaxis:

```
mxmlElementSetAttrf(nodo, nombre_atributo, formato, ...);
```

Donde el parámetro `formato` será una cadena que determinará el formato del contenido del atributo. Tras este parámetro se pasarán como parámetros las variable o valores que se hayan definido en la cadena formato. Por ejemplo:

```
mxmlElementSetAttrf(nodo, "href", "%s/%s", ruta, fichero);
```

Esta sentencia añadirá al elemento apuntado por `nodo` un atributo "href" con la ruta a un fichero.

4.4.1.4. Añadiendo contenido a un elemento

Para añadir contenido a un elemento se usa la función **mxmlNewText**, que tiene la siguiente sintaxis:

```
mxmlNewText(nodo, blancos, texto);
```

Donde `nodo` es un puntero al nodo al que se quiera añadir contenido, `blancos` es un entero con el número de espacios en blanco a insertar antes del texto, y `texto` es una cadena con el texto a añadir al nodo.

Para modificar el texto de un nodo ya existente se usa la función **mxmlSetText**, cuya sintaxis es idéntica a la de la función **mxmlNewText**.

Al igual que para los atributos, se puede establecer un formato para el contenido de un elemento con la función **mxmlNewTextf**, cuya sintaxis es idéntica a la función **mxmlElementSetAttrf** explicada en el apartado **4.4.1.3.**. Si se desea modificar el contenido ya existente de un elemento dándole formato, se utilizará la función **mxmlSetTextf** que, de nuevo, tiene la misma sintaxis que la función **mxmlNewTextf**.

4.4.1.5. Salvando un fichero XML

Para salvar el fichero XML creado se usa la función **mxmlSaveFile**, cuya sintaxis es la siguiente:

```
mxmlSaveFile(nodo_raiz, fichero, callback);
```

Donde `nodo_raiz` es un puntero a una estructura mxml_node_t que se corresponde con el nodo raíz del documento XML. El parámetro `fichero` es un puntero a FILE que apunte a un fichero abierto o creado para escritura con la función **fopen**. El parámetro `callback` se corresponde con una constante definida en la librería mxml. Este parámetro determina cómo deben tratarse los datos almacenados en el documento XML. Normalmente, es suficiente con darle el valor MXML_NO_CALLBACK, que tratará todos los datos como texto. Si se le da el valor MXML_INTEGER_CALLBACK, todos los datos serán tratados como números enteros. Devuelve 0 si la operación se produce con éxito o -1 si falla.

En el siguiente ejemplo se muestra cómo crear un documento XML sencillo. Puede verse en funcionamiento en el **ejemplo 3**:

```
fatInitDefault();
mxml_node_t *raiz_xml;
mxml_node_t *padre;
mxml_node_t *hijo;

raiz_xml = mxmlNewXML("1.0");

padre = mxmlNewElement(raiz_xml, "padre");
mxmlElementSetAttr(padre, "atributo", "valor_atributo");
hijo = mxmlNewElement(padre, "hijo");
mxmlNewText(hijo, 0, "Texto del hijo");

FILE  *f;
f = fopen("sd:/documento.xml", "wb");
mxmlSaveFile(raiz_xml, f, MXML_NO_CALLBACK);
```

La anterior serie de sentencias creará un fichero XML en el directorio raíz de la tarjeta de memoria SD llamado "documento.xml" que tendrá el siguiente aspecto:

```
<?xml version="1.0" ?>
<padre atributo="valor_atributo">
<hijo>Texto del hijo</hijo>
</padre>
```

4.4.2. Lectura de un fichero XML

4.4.2.1. Cargando el fichero XML

Para cargar el fichero XML se utiliza la función **mxmlLoadFile**, cuya sintaxis es la siguiente:

```
nodo_raiz = mxmlLoadFile(nodo, fichero, callback);
```

Donde `nodo` es un puntero a un nodo padre, si es que existe alguno. Normalmente este parámetro será NULL. El parámetro `fichero` es un puntero a FILE que apunta a un fichero abierto en modo lectura con la función **fopen**. El parámetro `callback` es el mismo que se ha explicado en el apartado **3.4.1.5.**, que normalmente será MXML_NO_CALLBACK. La función devuelve un puntero a una estructura mxml_node_t, en este caso el puntero es `nodo_raiz`, que será el nodo que englobará al documento al completo.

4.4.2.2. Búsqueda de un elemento

Para navegar a través de un árbol de nodos, se utilizan las funciones **mxmlWalkPrev** y **mxmWalkNext**, cuya sintaxis es la siguiente:

```
mxmlWalkPrev(actual, padre, modo_busqueda);

mxmlWalkNext(actual, padre, modo_busqueda);
```

Estas funciones navegarán a través de los hijos del nodo `padre` , cuya referencia se pasa cómo parámetro. Este nodo normalmente será el nodo raíz del documento. El parámetro `actual` es el nodo a partir del cual se quiere comenzar la búsqueda. El tercer parámetro es una constante definida en la librería mxml. Esta constante puede tener los valores que se definen en la **tabla 4.2** e indican cómo debe comportarse la función de búsqueda al navegar por la jerarquía de nodos.

Tabla 4.2. Constantes para el modo de búsqueda.

Constante	Descripción
MXML_NO_DESCEND	Indica que la búsqueda no debe descenderse en ningún hijo en la jerarquía, solo en nodos al mismo nivel. Es decir, solo se navegará a través de los hijos directos de `padre`.
MXML_DESCEND_FIRST	Indica que se puede descender un nivel en la jerarquía de nodos "hijos".
MXML_DESCEND	Indica que se puede descender a cualquier nivel de la jerarquía Es decir, se navegarán todos los nodos del documento, independientemente de su nivel.

Ambas funciones devuelven un puntero a una estructura mxml_node_t, con el nodo encontrado. La primera función asciende del nodo actual al anterior nodo en la jerarquía. La segunda función desciende al siguiente nodo posterior a actual.

Para encontrar un nodo con un nombre concreto, se usa la función **mxmlFindElement**, que devuelve un puntero a una estructura mxml_node_t correspondiente al nodo encontrado, o NULL si el elemento no existe. Su sintaxis es la siguiente:

```
mxmlFindElement( nodo, raiz, nombre, atributo, valor,
                 modo_busqueda);
```

Donde `nodo` es el nodo a partir del cual comenzar la búsqueda, y `raiz` es el nodo raíz de la jerarquía de nodos del documento. Si se desea realizar una búsqueda en todo el documento, ambos parámetros corresponderán al nodo raíz. El parámetro `nombre` es una cadena con el nombre del elemento que se esté buscando. Los parámetros `atributo` y `valor` sirven para filtrar la búsqueda por elementos con un atributo igual a `atributo`, y un valor de dicho atributo correspondiente al parámetro `valor`; ambos pueden ser NULL. El parámetro `modo_busqueda` se corresponde con una de las constantes definidas en la tabla 4.2. Por ejemplo, la siguiente sentencia encontraría el primer elemento "a" con un atributo "href" con un valor correspondiente a una URL concreta:

```
nodo = mxmlFindElement(raiz, raiz, "a", "href","http://www.upv.es",
                       MXML_DESCEND);
```

4.4.2.3. Obtención de la información de un elemento

Para obtener el valor de un atributo, se utiliza la función **mxmlElementGetAttr**, que requiere dos parámetros, el primero es un puntero al nodo, y el segundo una cadena con el nombre del atributo de dicho nodo cuyo valor se quiera obtener. La función devolverá una cadena con el

valor del atributo o NULL si no existe.

El contenido de un elemento se encuentra almacenado en la estructura del nodo correspondiente. Si existe una estructura mxml_node_t llamada nodo, su contenido se encuentra en:

```
nodo->value.text.string
```

4.5. Ejemplos adjuntos a la memoria relativos al capítulo 4

Ejemplo 3: Muestra cómo crear un fichero XML sencillo, mediante el uso de las funciones de la librería mxml.

Capítulo 5. Imágenes

5.1. Introducción. La librería wiisprite

En este capítulo se muestra cómo añadir imágenes en las aplicaciones usando la librería wiiprite <18>. Wiisprite es una librería escrita en C++ que hace uso de la GPU para la aceleración de gráficos y trabaja con imágenes PNG haciendo uso de la librería libpng en su versión para PowerPC.

La librería wiiprite es una librería externa al paquete devkitPPC. La versión más reciente puede descargarse de http://wiibrew.org/wiki/Libwiisprite. Para usarla en un proyecto, se debe incluir el fichero de cabeceras wiisprite.h y enlazarla en el Makefile añadiendo -libwiisprite en la variable LIBS.

Todas las clases de la librería wiisprite usan el espacio de nombres wsp, por lo que en el código fuente es necesario añadir la sentencia:

```
using namespace wsp;
```

5.2. Inicializando el subsistema de vídeo. La clase GameWindow

La clase GameWindow es una de las clases incluidas en la librería wiisprite, y contiene las funciones necesarias para inicializar el subsistema de vídeo. Para usar la clase GameWindow, se debe crear una instancia de la misma, sobre la que se llamarán a las funciones de la clase.

```
GameWindow gw; //siendo gw un nombre arbitrario.
```

Para inicializar el subsistema de vídeo se usa la función **InitVideo()**, que debe ser invocada antes que cualquier otra función de wiiprite. Si fuese necesario comprobar si el subsistema de vídeo ya se encontrase inicializado, se puede usar la función **IsInitialized()**, que devolverá un booleano con valor *true* en caso de encontrarse inicializado, o *false* en caso contrario.

Para establecer un color de fondo, se usa la función **SetBackground** que, siguiendo con el ejemplo anterior, tendría la siguiente sintaxis:

```
gw.SetBackground(bgcolor);
```

Donde `bgcolor` es una variable de tipo GXColor que tiene el siguiente formato: **{ r, g, b, a }**, siendo **r** el valor para el rojo (de 0 a 255), **g** el valor para el verde, **b** el valor para el azul, y **a** el valor para el canal *alpha* (o transparencia). Así, para establecer un color de fondo azul, se debería añadir la siguiente sentencia:

```
gw.SetBackground((GXColor){0, 0, 255, 255});
```

Si se desea obtener la resolución de la pantalla, se deben usar las funciones **GetWidth()** y **GetHeight()**, que devuelven un entero indicando la anchura y la altura, respectivamente. En caso de que el subsistema de vídeo no se encuentre inicializado, ambas funciones devolverán el valor 0.

La función **Flush()** finaliza el renderizado de la escena. Esta función debe invocarse después del resto de funciones de dibujo y deberá ser llamada siempre que se realice alguna modificación en la escena. Por último, la función **StopVideo()** detiene el subsistema de vídeo. Es recomendable invocar esta función al final del programa, para evitar cuelgues inesperados del sistema.

El siguiente ejemplo muestra como usar las funciones de la clase GameWindow para mostrar un fondo blanco en pantalla, y se puede ver su funcionamiento en el **ejemplo 4** del material complementario a esta memoria:

```
#include <wiiuse/wpad.h>
#include <wiisprite.h>

using namespace wsp;

int main(int argc, char **argv){
           GameWindow gw;
           gw.InitVideo();
           gw.SetBackground((GXColor){ 255, 255, 255, 255 });

           WPAD_Init();

           while(1){
                WPAD_ScanPads();
                if(WPAD_ButtonsDown(WPAD_CHAN_0)&WPAD_BUTTON_HOME)
                          break;
                gw.Flush();
           }
     gw.Flush();
     gw.StopVideo();
           return 0;
}
```

5.3. Carga de imágenes. La clase Image

La clase Image nos permite cargar una imagen en un programa, para posteriormente manipularla o dibujarla en pantalla. Esta clase dispone de las funciones **LoadImage**, que permiten cargar imágenes desde un directorio o desde un *buffer*. También dispone de otras funciones básicas como **GetWidth()** y **GetHeight()**, que devuelven un entero con el valor de la anchura y altura de la imagen, respectivamente, la función **DestroyImage()**, que limpia la imagen cargada en un objeto de tipo Image para poder cargar otra, y la función **IsInitialized()**, que comprueba si ya existe una imagen cargada en el objeto Image sobre el que se invoca y devuelve en booleano con valor *true* en caso de ser cierto, y *false* en caso contrario. Algo a tener en cuenta es que solo es posible cargar imágenes cuya anchura y altura sea un múltiplo de 4, de lo contrario la función LoadImage devolverá un error IMG_LOAD_ERROR_WRONG_SIZE. Si la imagen se carga correctamente, devolverá el valor IMG_LOAD_ERROR_NONE.

5.3.1 Carga de imágenes desde un directorio

Para cargar imágenes desde un directorio es necesario crear un objeto de tipo Image y usar la función **LoadImage** pasando como parámetro un *string* con la ruta donde se encuentre la imagen. Por ejemplo:

```
Image imagen;
if(imagen.LoadImage("imagenes/ImagenDirectorio.png")       !=
IMG_LOAD_ERROR_NONE) exit(0);
```

Las dos sentencias anteriores crean un objeto de la clase Image y carga sobre él una imagen "ImagenDirectorio.png", guardada en un directorio "imagenes" en la misma ubicación donde se encuentre el ejecutable de la aplicación. En caso de que la imagen no se encuentre en la ubicación especificada, la función devolverá un error y el programa terminará su ejecución.

5.3.2. Carga de imágenes desde buffer. La herramienta raw2c

Para integrar las imágenes en la aplicación cargándolas desde un *buffer* en el código fuente, el procedimiento es similar a cargarlas desde un directorio, salvo que en este caso basta con pasar a la función de carga el nombre del *buffer*. Así, si existe un *buffer* llamado ImagenBuffer, se cargará en un objeto `imagen` de la clase Image como se muestra en la siguiente sentencia:

```
imagen.LoadImage(ImagenBuffer);
```

DevkitPro incluye la herramienta *raw2c*, que permite transformar una imagen en código fuente. Esta herramienta se encuentra en c:\devkitpro\devkitPPC\bin\. Al arrastrar una imagen sobre la herramienta, esta generará un fichero de código C con un array que contiene los datos de la imagen y un fichero .h para incluir en el código fuente de la aplicación. Sí, por ejemplo, se arrastra una imagen "ImagenBuffer.png" a la aplicación *raw2c*, esta generará un fichero de código .c con el nombre "ImagenBuffer.c" y un fichero de cabeceras "ImagenBuffer.h", que contendrá la definición de un *buffer* "ImagenBuffer" y una variable "ImagenBuffer_size" con el valor de la talla del *buffer*. Tras esto, bastaría con añadir la sentencia `#include "ImageBuffer.h"` en el código fuente.

5.4. Manipulación de imágenes. La clase Sprite

Los *sprites* son un elemento clave en el uso de la librería wiisprite. El término *sprite*, procedente de los primeros juegos en sistemas como Atari 400/800 o Commodore 64, se usa para referirse a un mapa de bits, normalmente pequeño y con transparencias <19>. La librería wiisprite proporciona la clase Sprite, que dispone de funciones que permiten manejar y manipular fácilmente estas imágenes o *sprites*.

Una vez cargadas las imágenes en una aplicación, es necesario asignarlas a un objeto Sprite, que nos permitirá manipularlas. Para asignar una imagen previamente cargada a un Sprite, se usa la función **SetImage**. Así si existe un objeto Sprite mi_sprite y un objeto Image mi_imagen, con una imagen ya cargada, se asociarían como se muestra a continuación:

```
mi_sprite.SetImage(&mi_imagen);
```

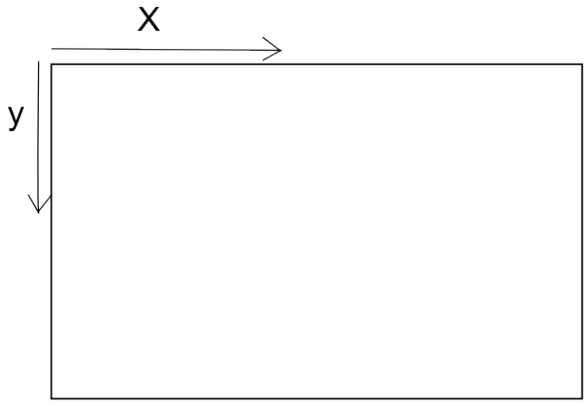
5.4.1 Posicionamiento de las imágenes

Para establecer la posición de la imagen en la pantalla se dispone de la función **SetPosition** que, siguiendo con el ejemplo del apartado anterior, se usaría como sigue:

```
mi_sprite.SetPosition(x, y);
```

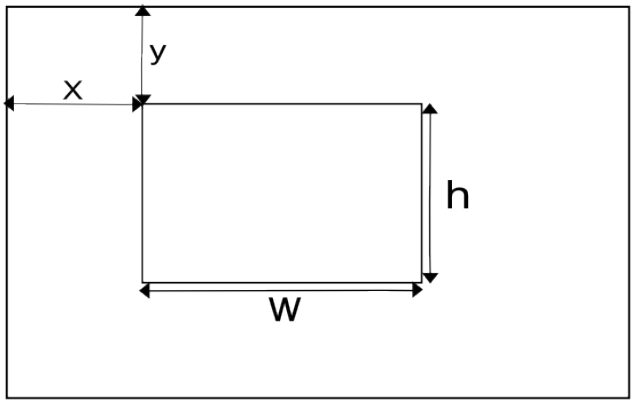
Donde `x` e `y` son las coordenadas donde se situaría la imagen en la pantalla. La coordenada 'x' crece desde la esquina superior izquierda en horizontal hacia la derecha, mientras que la coordenada 'y' crece desde la esquina superior izquierda en vertical hacia abajo, cómo se muestra en la **figura 5.1**.

Figura 5.1. Sentido de crecimiento de las coordenadas x e y.



La posición indicada a la función establece dónde se situará el píxel de la esquina superior izquierda, a partir del cual se dibujará el resto de la imagen. Así, una imagen situada en la coordenada *(x, y)*, con una anchura *w* y una altura *h*, tras el renderizado quedaría en la posición mostrada en la **figura 5.2**.

Figura 5.2. Situación de una imagen en las coordenadas x e y.



5.4.2. Manipulación de las imágenes

La clase Sprite dispone de varias funciones para modificar las dimensiones y la rotación de la imagen, así como para desplazarla.

5.4.2.1. Manipulación de las dimensiones de la imagen

La función **SetZoom(zoom)** permite escalar la imagen a lo ancho y a lo alto con un factor determinado por el parámetro **zoom**, siendo este un valor en coma flotante de 32 bits. Para obtener el factor de escalado actual se usa la función **GetZoom()**. Este factor vale 1 si la imagen se encuentra en su tamaño original.

Para modificar las dimensiones de la imagen por separado se utilizan las funciones **SetStretchWidth(stretchWidth)** y **SetStretchHeight(stretchHeight)**, que modifican la anchura y la altura, respectivamente. Los parámetros **stretchWidth** y **stretchHeight**, ambos valores en coma flotante, determinan el factor con el que se modifican estas dimensiones. Para obtener el factor de escalado actual del la anchura y la altura, se usan las funciones **GetStretchWidth()** y **GetStretchHeight()**, respectivamente. De nuevo, este factor valdrá 1 si la anchura y la altura de la imagen son las originales.

5.4.2.2. Manipulación de la rotación de la imagen

La función **SetRotation(rotation)** permite establecer el ángulo de rotación de la imagen, con respecto a su posición original. La rotación se mide en grados/2, por lo que si se desea rotar una imagen 90 grados, el parámetro **rotation** sera 45. La función **GetRotation()** devuelve el ángulo de rotación actual de la imagen.

5.4.2.3. Desplazamiento de la imagen

La función **Move(deltaX, deltaY)** permite desplazar la posición de una imagen. El parámetro **deltaX** determina el desplazamiento en el eje de coordenadas x, mientras que **deltaY** determina el desplazamiento en el eje y, siendo ambos valores en coma flotante. Por ejemplo, la siguiente sentencia desplazaría el objeto Sprite 10 píxeles hacia la derecha y 10 píxeles hacia arriba:

```
sprite.Move(10.0, -10.0);
```

5.4.2.4. Manipulación de la transparencia y la visibilidad de la imagen

Para modificar la transparencia de una imagen, se dispone de la función **SetTransparency**, cuya sintaxis es:

```
sprite.SetTransparency(alpha);
```

Donde el valor de `alpha` puede variar de 0 (imagen totalmente transparente) a 255 (imagen totalmente opaca). Para obtener el valor *alpha* de una imagen, se usa la función **GetTransparency()**. Otra forma de modificar la visibilidad de una imagen es mediante la función **SetVisible**, que se usa de la siguiente forma:

```
sprite.SetVisible(visible);
```

Esta función volverá invisible la imagen cuando `visible` tenga el valor *false*, y la hará de nuevo visible cuando `visible` tenga el valor *true*. La función **IsVisible()** informa de la visibilidad de la imagen, devolviendo el valor *true* si el objeto es visible, o *false* en caso contrario.

5.4.3. Imágenes como puntero del Wiimote

Al usar la librería wiisprite para establecer una imagen como puntero para el Wiimote, es necesario realizar unos ajustes a las coordenadas de los infrarrojos. Para ello, la librería dispone de dos constantes, WSP_POINTER_CORRECTION_X y WSP_POINTER_CORRECTION_Y, que deben restarse a las coordenadas x e y de los infrarrojos. En el siguiente ejemplo, se muestra cómo hacer que un objeto Sprite de nombre `puntero`, siga la posición y rotación de los infrarrojos del Wiimote:

```
ir_t ir;
WPAD_IR(WPAD_CHAN_0, &ir);

puntero.SetPosition(ir.x-WSP_POINTER_CORRECTION_X,
                    ir.y-WSP_POINTER_CORRECTION_Y);

puntero.Move(-((f32)puntero.GetWidth()/2),
              -(f32)puntero.GetHeight()/2));

puntero.SetRotation(ir.angle/2);
```

La función **Move** sitúa el píxel central de `puntero` sobre la posición a la que apunte el Wiimote.

5.4.4. Píxel de referencia y posicionamiento

Cuando se escala o se rota una imagen, se hace en torno a un píxel llamado píxel de referencia. Este píxel, por defecto, se sitúa en el centro de la imagen. Es posible cambiar la posición de este píxel con la función **SetRefPixelPosition**, cuya sintaxis es:

```
SetRefPixelPosition(x, y);
```

Donde `x` es la posición en el eje de coordenadas 'x' e `y` es la posición en el eje de coordenadas 'y'.

Es posible cambiar la posición del píxel de referencia en cada uno de los ejes por separado con la función **SetRefPixelX(x)** para la posición en el eje 'x' y **SetRefPixelY(y)** para la posición en el eje 'y'.

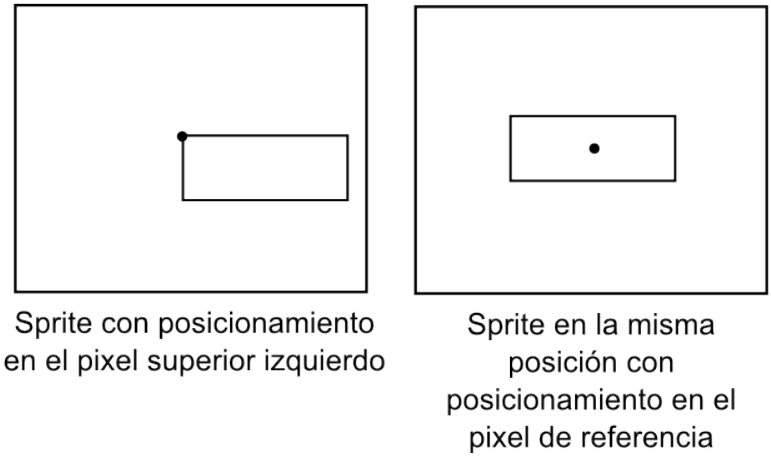
Como se ha explicado anteriormente, la posición de una imagen por defecto esta basada en el píxel superior izquierdo. Se puede alterar este comportamiento por defecto con la función **SetRefPixelPositioning**, cuya sintaxis es:

```
sprite.SetRefPixelPositioning(Metodo_Posicionamiento);
```

Metodo_Posicionamiento debe tener uno de los dos siguientes valores constantes, REFPIXEL_POS_TOPLEFT y REFPIXEL_POS_PIXEL. El primero de ellos establece el comportamiento por defecto, situar el píxel superior izquierdo en la posición indicada con la función **SetPosition**, y dibujar el resto de la imagen a partir de él. En cambio, si se pasa como parámetro el valor REFPIXEL_POS_PIXEL, el posicionamiento se realizará conforme a la posición del píxel de referencia. Ya que este píxel, por defecto, se sitúa en el centro de la imagen, esta será

dibujada alrededor del píxel en las coordenadas indicadas en la función SetPosition, como se muestra en la **figura 5.3**.

Figura 5.3. Comportamiento del píxel de referencia.



5.5. Gestión de capas. La clase LayerManager

Otro de los elementos clave en el uso de la librería wiisprite son las capas. Las capas son necesarias para mostrar las imágenes en pantalla, situando unas por encima de otros. Para gestionar las capas se debe crear un objeto LayerManager como se muestra en el siguiente ejemplo:

```
LayerManager manager(3);
```

En el ejemplo anterior, se crea un objeto `manager`, de la clase LayerManager. El entero pasado como parámetro indica el número de capas va a contener. En el caso del ejemplo, `manager` gestionará tres capas distintas. Para obtener el número de capas del objeto `manager`, se usa la función **GetSize()**:

```
int tamanyo = manager.GetSize();
```

En este caso, `tamanyo` tendrá el valor 3.

Se puede asignar un objeto Sprite a cada una de las capas. Para hacerlo, se usa la función **Append** sobre el objeto `manager`. Suponiendo que existe un objeto de la clase Sprite llamado sprite1, se asignaría como se indica:

```
manager.Append(&sprite1);
```

Los objetos LayerManager funcionan esencialmente como un array, estando las capas numeradas empezando por cero. La función **Append** añade un Sprite en la última posición libre. De esta forma si, tras el ejemplo anterior, se usa de nuevo la función **Append** con un objeto Sprite sprite2, sprite1 ocuparía la posición 0 y sprite2 la posición 1. Para insertar un Sprite en una capa concreta se utilizaría la función **Insert** como se muestra a continuación:

```
manager.Insert(&sprite2, 0);
```

En este caso, `sprite2` se inserta en la posición 0, sustituyendo al *sprite* que se encontrase en esa posición.

Para obtener el *sprite* que se encuentre asignado a una capa concreta, se utiliza la función **GetLayerAt(u32 index)**, donde **index** es el índice de la capa. La función devolverá la dirección de memoria de dicha capa.

La función **Remove**, permite borrar el elemento cuya dirección se le pase como parámetro. Por ejemplo:

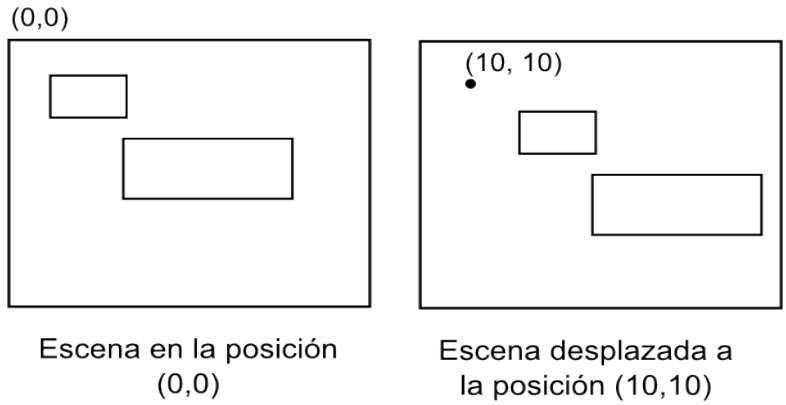
```
manager.Remove(&sprite2);
```

eliminará al elemento `sprite2` de su posición, que pasará a ser NULL. Para borrar todos los elementos a la vez, basta con usar la función **RemoveAll()**, que no necesita parámetros. Por último, para dibujar todas las capas, se usará la función **Draw**, cuya sintaxis es:

```
manager.Draw(x, y);
```

Los parámetros `x` e `y` indican la posición de referencia a partir de la cual se dibujarán las capas. Estos dos parámetros normalmente tendrán el valor 0 para ajustar las capas a la pantalla, aunque pueden situarse en cualquier lugar, lo que desplazará la escena completa a la posición indicada, como se muestra en la **figura 5.4**.

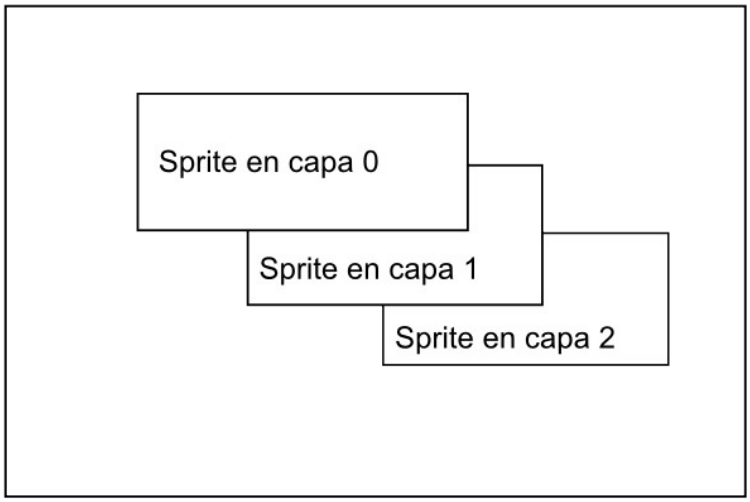
Figura 5.4. Resultados de la función Draw.



La función **Draw** dibuja las capas en orden, comenzando por la de mayor índice, cómo se muestra en la **figura 5.5**. Las capas con un indice mayor se sitúan al fondo, mientras que las capas de menor indice se sitúan al frente. De esta forma, la capa con el índice 0 se sitúa al frente de todas. Tras esta función, es necesario llamar a la función **Flush()** para completar el renderizado. Es posible dibujar varias escenas simultáneas creando varios objetos LayerManager. En este caso, las escenas se dibujarán en el orden en el que se invoque la función **Draw** sobre los diversos objetos LayerManager, siendo el último que se dibuje el que se sitúe en la posición superior. Hay que tener

en cuenta que solo se detectarán colisiones entre imágenes que se encuentren en el mismo Layer Manager. Las colisiones entre imágenes se describen en el apartado 5.7.

Figura 5.5. Posición de las capas según su índice.



5.6. Quads

Un quad es un elemento gráfico de forma rectangular. Los objetos Quad en la librería wiisprite son similares a los objetos Sprite, salvo que no se les asocia ninguna imagen, teniendo un color de relleno y un borde. La clase Quad dispone de funciones similares a las de la clase Sprite como **SetWidth**, **SetHeight**, **SetRotation**, **GetRotation** y **SetPosition**.

5.6.1. Funciones específicas de la clase Quad

Para establecer el color de relleno del Quad, se utiliza la función **SetFillColor(GXColor fillColor)**, donde **fillColor** se asigna como se describe en la función **SetBackground** de la clase GameWindow. Por defecto el color será negro.

Se puede establecer un borde con las siguientes funciones. La función S**etBorderWidth(u16 width)** le da al borde un grosor **width**. Con la función **GetBorderWidth()** se puede obtener dicho valor. La función **SetBorder(boolean border)** mostrará el borde si el valor de **border** es *true*, o lo ocultará si el valor de **border** es *false*. La función **IsBorder()** devuelve un booleano indicando si el borde se encuentra actualmente visible o no. Por último, la función **SetBorderColor(GXColor borderColor)** establece un color para el borde, funcionando de manera similar a la función **SetFillColor**, explicada en el apartado **5.2**.

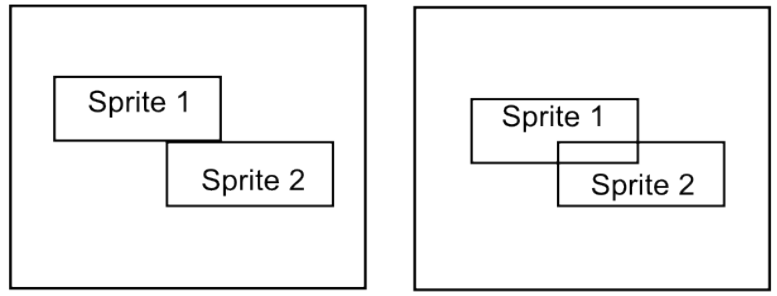
5.7. Detección de colisiones

Para detectar colisiones entre imágenes, wiisprite dispone de la función **CollidesWith**, disponible para la clase Sprite. Esta función devuelve un booleano con valor *true* si se detecta una colisión entre dos imágense, o *false* en caso contrario. La forma habitual de usar la función **CollidesWith** se muestra a continuación:

```
if(sprite.CollidesWith(&sprite2)){
      //Acciones a realizar.
}
```

Suponiendo que `sprite` y `sprite2` son dos objetos Sprite, la función **CollidesWith** devolverá *true* si `sprite` colisiona con `sprite2`, como se muestra en la **figura 5.6**.

Figura 5.6. Ejemplo de colisiones de sprites.



No hay colisión. La función CollidesWith devuelve *false*.

Hay colisión. La función CollidesWith devuelve *true*.

En este caso, a la hora de calcular la colisión, solo se tiene en cuenta la posición de las imágenes, no la rotación o el escalado. Si se quiere añadir al cálculo de la colisión la rotación y el escalado, la función **CollidesWith** se usaría de la siguiente forma:

```
if(sprite.CollidesWith(&sprite2, true)){
      //Acciones a realizar.
}
```

Ahora se tendrá en cuenta la rotación y el escalado de ambas imágenes, además de su posición, para calcular la colisión. Por supuesto, esto puede aumentar considerablemente el rendimiento de la aplicación, así que sólo es recomendable su uso cuando sea necesario.

5.7.1. Rectángulo de colisión

Para detectar colisiones, wiisprite emplea una estructura Rectangle que define un rectángulo, llamado rectángulo de colisión, que se asocia a un objeto Sprite cuando este es creado. La estructura Rectangle se muestra a continuación:

```
struct Rectangle{
      f32 x, y;
      f32 width, height;
};
```

La posición del rectángulo se indica en `x` e `y`, mientras que `width` y `height` indican la anchura y la altura, respectivamente. Por defecto, estas dimensiones se corresponden con las de la

imagen al que se encuentre asociado. Para modificar los datos del rectángulo, se dispone de la función **DefineCollisionRectangle**, cuya sintaxis es:

```
DefineCollisionRectangle(x, y, width, height);
```

Los parámetros determinan la posición en x, la posición en y, la anchura y la altura, en ese orden. Es posible obtener el rectángulo asociado a una imágen mediante la función **GetCollisionRectangle()**, que devuelve un puntero al rectángulo asociado.

5.7.1.1. Rectángulos de colisión y Quads

Es posible asociar un rectángulo de colisión a un objeto Quad mediante la función SetRectangle, que recibe como parámetro un puntero a una estructura de tipo Rectangle. Mediante el uso de esta función, el objeto Quad adquiere las dimensiones y la posición del rectángulo. Para ello es necesario haber creado antes una estructura de tipo Rectangle.

```
Rectangle rect;
Quad quad;
rect.x = 10;
rect.y = 10;
rect.width = 50;
rect.height = 50;
quad.SetRectangle(&rect);
```

La anterior serie de sentencias creará un objeto Quad de 50x50 píxeles situado en el punto (10,10), con un rectángulo de colisión de las mismas dimensiones asociado, que podrá ser utilizado para detectar colisiones con el objeto Quad.

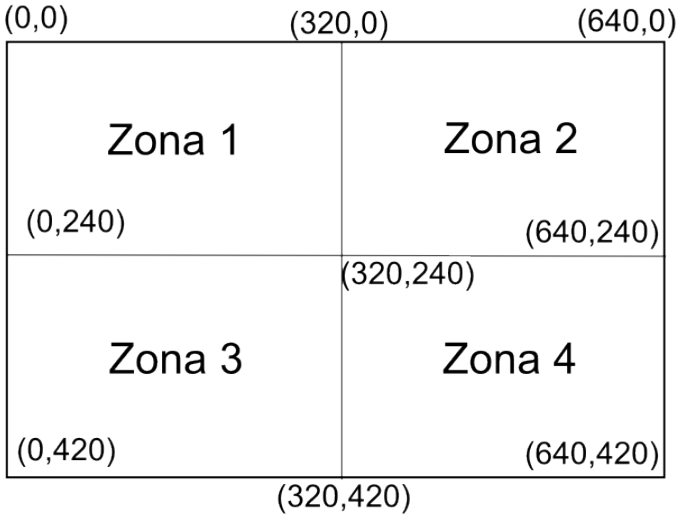
5.7.1.2. Rectángulos de colisión y zonas de la pantalla

También es posible usar los rectángulos de colisión sin asociarlos a otro objeto, usualmente para detectar colisiones de una imagen sobre una zona concreta de la pantalla. Esto es especialmente útil para mejorar el rendimiento de una aplicación. La detección de colisiones puede ser un proceso muy costoso, especialmente si hay muchos elementos en la pantalla.

Supóngase que se está desarrollando un juego, con una gran multitud de enemigos, proyectiles u otros elementos mostrándose a la vez en la pantalla; o la interfaz de una aplicación con multitud de elementos gráficos, botones, etc. Comprobar si el *sprite* del jugador colisiona con alguno de los demás en cada iteración del programa podría ser muy costoso, además, la mayoría de estas comprobaciones resultarían en un valor *false*, por lo que se estaría desperdiciando mucho tiempo de computación.

Una manera de evitar esto y mejorar el rendimiento sería usar rectángulos para dividir la pantalla en zonas. Suponiendo que la pantalla tiene una resolución de 640x480, se dividiría como se muestra en la **figura 5.7**.

Figura 5.7. División de la pantalla en zonas de colisión.



En cada iteración se comprobaría cuál de las zonas colisiona con el *sprite* del jugador o la imagen del puntero del Wiimote y, en caso de producirse, se pasaría a comprobar las colisiones con los objetos de esa zona, ignorando lo que se encuentre en el resto de zonas. Para aumentar la efectividad de esta técnica, se podría aumentar el número de zonas en las que dividir la pantalla.

Para detectar la colisión entre un objeto Sprite y un rectángulo, se utiliza la función CollidesWith de forma similar a cómo se describe en el apartado 4.7. salvo que se pasa como parámetro una referencia al objeto Rectangle. Por ejemplo, la siguiente sentencia comprobaría si se produce una colisión entre un objeto Sprite `sprite` y un objeto Rectangle `rectangulo`:

```
sprite.CollidesWith(&rectangulo);
```

5.8. *Sprite sheets*. La clase TiledLayer

En ocasiones, puede ser necesario crear una imagen compuesta de una multitud de imágenes más pequeñas, como podría ser el fondo de pantalla de un video-juego, o puede que se quiera crear una animación compuesta de una gran cantidad de frames. Para estas tareas sería necesario cargar un gran número de imágenes en la aplicación, lo que podría llegar a ser muy laborioso. Para simplificar esta tarea, es habitual el uso de las llamadas *sprite sheets* (**figura 5.8**). Estas *sprite sheets* están compuestas de varios sprites de menor tamaño, a modo de mosaico, que se dividen en celdas del mismo tamaño creando una multitud de *sprites* diferentes.

Figura 5.8. Ejemplo de s*prite sheet*.

Para manejar *sprite sheets*, la librería wiisprite dispone de la clase TiledLayer, que contiene las funciones necesarias para componer y manejar estos *sprites*.

5.8.1. Creación de un objeto TiledLayer

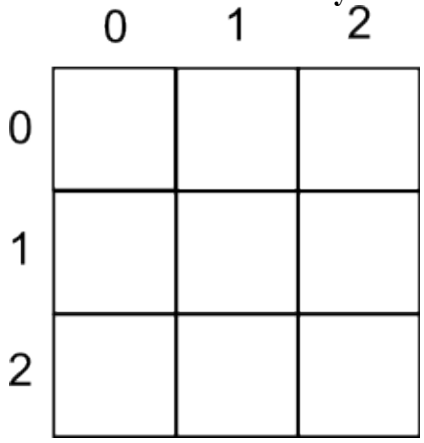
Para crear un nuevo objeto TiledLayer, se invoca el constructor de la clase, que tiene la siguiente sintaxis:

```
TiledLayer objetoTiledLayer( columnas, filas, animados);
```

Esta función creará una matriz de celdas cuyo número de columnas estará determinado por el argumento `columnas`, y el número de filas estará determinado por el argumento `filas`. El argumento `animados` determina la cantidad posible de celdas con animaciones, cuya utilidad se describe más adelante. El índice de filas y columnas comienza en 0 y avanza consecutivamente de izquierda a derecha para las columnas y de arriba hacia abajo para las filas. Por ejemplo, la siguiente sentencia crearía un objeto TiledLayer de nombre mosaico con 3 filas y 3 columnas, sin celdas animadas, mostrado en la **figura 5.9**:

```
TiledLayer mosaico(3, 3, 0);
```

Figura 5.9. Indice de las filas y columnas.



El siguiente paso consiste en asignar una *sprite sheet* al objeto TiledLayer. Para ello, la *sprite sheet* ha debido ser cargada de la forma habitual, utilizando la función **LoadImage** de la clase Image. Tras haber cargado la imagen, se asocia al objeto TiledLayer mediante la función **SetStaticTileset**, cuya sintaxis es la siguiente:

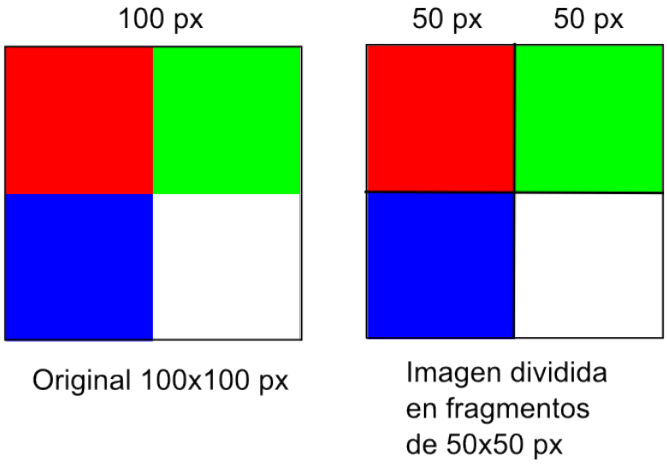
```
objetoTiledLayer.SetStaticTileset( imagen, anchura, altura);
```

Donde `imagen` es la referencia a una imagen cargada. Los parámetros `anchura` y `altura` determinan las dimensiones de las celdas en las que se va a dividir la *sprite sheet*. Por ejemplo, si se carga una *sprite sheet* de 100x100 píxeles llamada "imagen", y se desea dividirla en cuatro *sprites*, la función **SetStaticTileset** se utilizaría como se muestra a continuación:

```
mosaico.SetStaticTileset( &imagen, 50, 50);
```

Esto dividiría la *sprite sheet* con cuatro *sprites* de 50x50 píxeles, cómo se muestra en la **figura 5.10**.

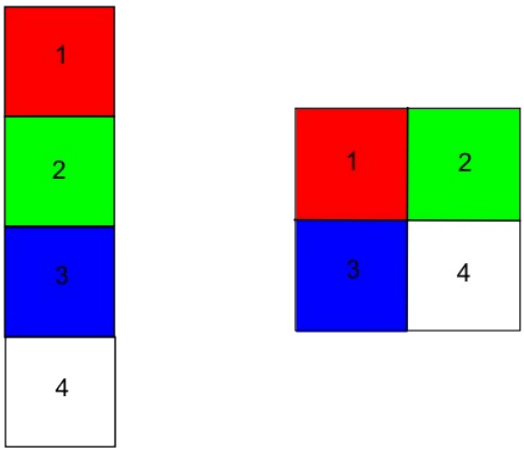
Figura 5.10. División de la imagen mediante la función SetStaticTileset.



El tamaño de los *sprites* es lo que determinará el número de fragmentos en las que se dividirá la imagen, de la misma forma que determinará el tamaño de las celdas del objeto TiledLayer al que se asocie la imagen.

La función **SetStaticTileset** asigna un índice a cada *sprite*, comenzando por asignar un 1 en al primer *sprite* de la izquierda de la primera fila, y continuando de forma consecutiva a lo largo de esa fila hasta pasar a la siguiente. En la siguiente **figura 5.11** se muestran varios ejemplos de asignación de índices para varias imágenes con distintas dimensiones, pero resultados equivalentes.

Figura 5.11. Asignación de índices equivalentes en imágenes diferentes.



5.8.2. Renderizado de los objetos TiledLayer

Una vez determinadas las dimensiones del objeto TiledLayer y asociado una imagen divida en *sprites*, hay que "rellenar" las celdas del objeto TiledLayer con los fragmentos de la imagen. Para ello se dispone de dos funciones, las funciones **SetCell** y **FillCells**.

La función **SetCell** permite asignar *sprites* a cada celda, individualmente. Su sintaxis es la siguiente:

```
objetoTiledLayer.SetCell( columna, fila, indice);
```

Donde `columna` y `fila` determinan la posición de la celda e `indice` se corresponde con el indice del *sprite* que se quiera asignar a esa celda.

Por otro lado, la función **FillCells** permite asignar un único *sprite* a un grupo de celdas. Su sintaxis es la siguiente:

```
objetoTiledLayer.FillCells( columnaInicio, filaInicio, columnas,
                                    filas, indice);
```

Esta función asignará el *sprite* correspondiente a `indice` a un rectángulo de una cantidad `columnas` de celdas a lo ancho y una cantidad `filas` de celdas a lo alto, comenzando desde la celda cuya posición determinan los parámetros `columnaInicio` y `filaInicio`.

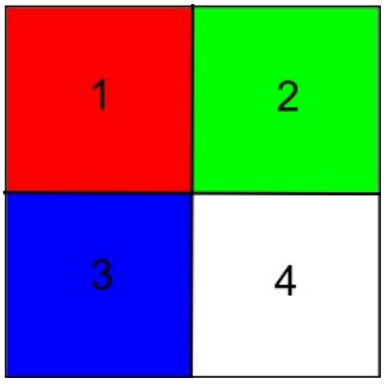
Como ejemplo práctico de lo anterior y siguiendo con el ejemplo del apartado anterior, supóngase que se ha creado un objeto TiledLayer de 3x3 celdas:

```
TiledLayer mosaico(3,3,0);
```

Ha este objeto se le asigna la imagen de la siguiente **figura 5.12** divida en cuatro *sprites* mediante las sentencias:

```
Image imagen;
imagen.LoadImage("FiguraX_x.png");
mosaico.SetStaticTiledset(&imagen, 50, 50);
```

Figura 5.12. Imagen dividida.

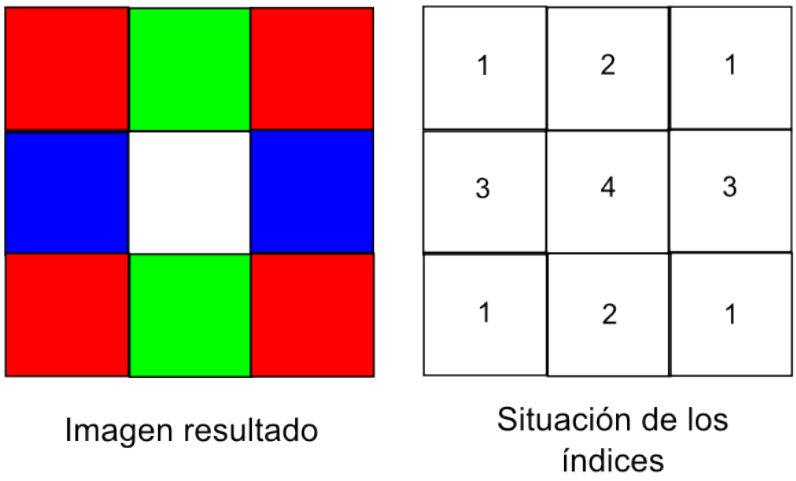


Tras esto, mediante la función **SetCell**, se puede asignar un sprite a cada celda del objeto TiledLayer. Las siguientes sentencias crearían una imagen como la de la **figura 5.13**:

```
mosaico.SetCell(0,0,1);
mosaico.SetCell(1,0,2);
```

```
mosaico.SetCell(2,0,1);
mosaico.SetCell(0,1,3);
mosaico.SetCell(1,1,4);
mosaico.SetCell(2,1,3);
mosaico.SetCell(0,2,1);
mosaico.SetCell(1,2,2);
mosaico.SetCell(2,2,1);
```

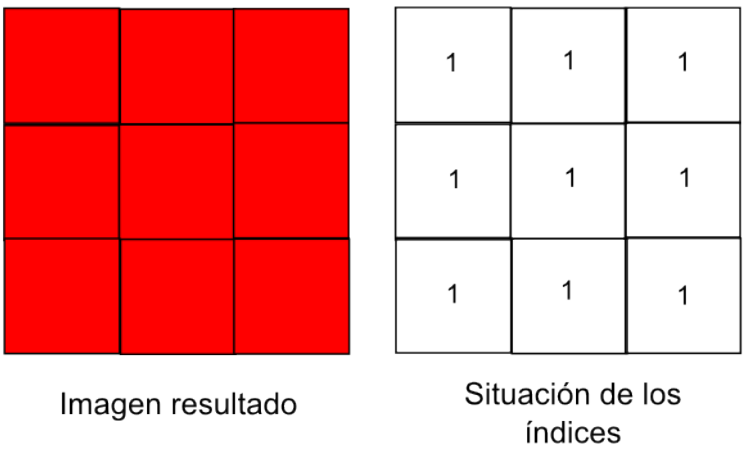
Figura 5.13. Resultado de las sentencias SetCell.



También puede emplearse la función **FillCells** para asignar el mismo *sprite* a un conjunto de celdas. La siguiente sentencia daría como resultado la imagen de la **figura 5.14**:

```
mosaico.FillCells(0,0,3,3,1);
```

Figura 5.14. Resultado de la sentencia FillCells.



Una vez asignados los *sprites* a sus respectivas celdas, la imagen se renderiza de la forma habitual, asignando el objeto TiledLayer a una capa de un objeto LayerManager y usando la función Draw.

5.8.3. Celdas animadas

La clase TiledLayer permite asignar índices especiales a las celdas. Estos índices, de valor negativo, indican que la celda es una celda animada. El número total de posibles celdas animadas debe indicarse en el constructor de la clase TiledLayer, como se ha explicado en el apartado 4.8.1.

La función **CreateAnimatedTile()** crea un nuevo índice para una celda animada, cuyo valor devuelve. Los índices para celdas animadas comienzan en -1 y decrecen de forma consecutiva cada vez que se usa la función **CreateAnimatedTile** hasta que no quedan posibles celdas para animar, tras lo que devolverá el valor 0, siendo la cantidad máxima de índices animados la especificada por el tercer parámetro de la función constructora de la clase TiledLayer. Estos índices pueden asignarse a las celdas de la forma habitual, mediante las funciones **SetCell** y **FillCells**.

Mediante la función **SetAnimatedTile** asocia uno de los índices de celdas animadas con uno de los índices de los *sprite* estáticos. Su sintaxis es la siguiente:

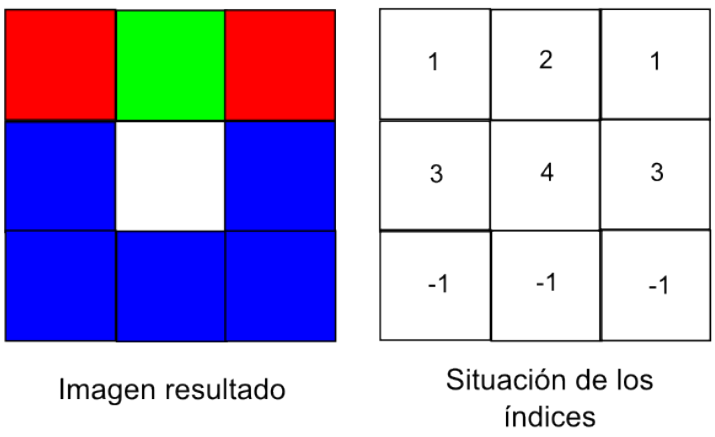
```
objetoTiledLayer.SetAnimatedTile( indiceAnimada, indiceEstatico);
```

Donde `indiceAnimada` es el índice de la celda animada e `indiceEstatica` es el índice del sprite. Siguiendo con el ejemplo anterior, si se hace la siguiente asociación, tras haber creado un índice animado con la función **CreateAnimatedTile**:

```
mosaico.SetAnimatedTile(-1, 3);
```

El *sprite* con índice 3 quedará asociado al índice de celda animada -1. De esta forma, si durante la asignación de indices de la **figura 5.13** se asigna el índice -1 a la última fila, la imagen tendrá el aspecto que se muestra en la **figura 5.15**.

Figura 5.15. Imagen con índices animados.

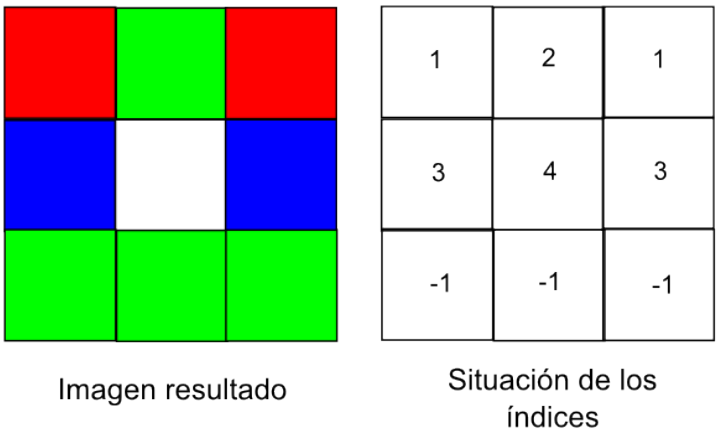


Estos índices facilitan la animación de las celdas, ya que basta usar la función SetAnimatedTile para cambiar el sprite asociado al índice negativo. Por ejemplo:

```
mosaico.SetAnimatedTile(-1,2);
```

Esta sentencia cambiaría todos los *sprites* de las celdas con índice -1 por el *sprite* con índice 2. El resultado se muestra en la **figura 5.16**.

Figura 5.16. Resultado de la sentencia SetAnimatedTile.



Por último, la función **GetAnimatedTile** devuelve el índice negativo asociado a un *sprite*, en caso de tenerlo. Su sintaxis es la siguiente:

```
objetoTiledLayer.GetAnimatedTile(indice);
```

Donde `indice` es el índice del *sprite*.

5.8.4. Otras funciones de la clase TiledLayer

Las funciones **GetColumns()** y **GetRows()** devuelven el número de columnas y filas, respectivamente, de un objeto TiledLayer. Las función **GetImage()** devuelve un puntero al objeto Image asociado al objeto TiledLayer. Las funciones **GetCellWidth()** y **GetCellHeight()** devuelven, respectivamente, la anchura y la altura de las celdas de un objeto TiledLayer. Por último, las función **SetTransparency(alpha)**, donde **alpha** es un valor de 0 a 255, establece la transparencia del objeto TiledLayer, mientras que la función **GetTransparency()** devolvería el valor *alpha*.

5.9. Ejemplos anexos a la memoria relativos al capítulo 5

Ejemplo 4: Muestra cómo inicializar el sistema de vídeo con la librería wiisprite, estableciendo un fondo de pantalla blanco.

Ejemplo 5: Muestra cómo cargar imágenes, desde fichero y desde *buffer*, cómo usar una imagen como puntero del Wiimote, cómo modificar estas imágenes con las funciones de la librería

wiisprite, como cambiar el píxel de posicionamiento de la imagen y el uso de un objeto Quad para crear un efecto de *fade in* y *fade out*.

Ejemplo 6: Muestra cómo detectar colisiones entre *sprites* y el uso de la clase TiledLayer creando un sencillo juego consistente en disparar a una diana apuntando con el Wiimote. Este juego también usa XML para guardar información, tal como se ha presentado en el capítulo 4 y reproduce efectos de sonido en formato mp3. La reproducción de audio mp3 se explica en el siguiente capítulo.

Ejemplo 7: Juego de puzzle consistente en adivinar una clave de colores. Se usan las funciones de la clase Quad para crear parte de la interfaz del juego, con el que se interacciona mediante el Wiimote.

Capítulo 6. Sonidos

6.1. Introducción

En este capítulo se describe como reproducir sonidos en los formatos MP3 y MOD, reproduciéndolos desde un fichero o desde el código fuente.

6.2. Inicialización del subsistema de audio

Para inicializar el subsistema de audio es necesario incluir la librería ansdlib. Para ello hay que añadir la sentencia `#include <asndlib.h>` en el código fuente y -lasnd en la variable LIBS del fichero Makefile.

La función **ASND_Init()** inicializa el subsistema de audio, y debe ser usada antes que cualquier otra función relacionada con este subsistema. La función **ASND_Pause(pause)** pausa el subsistema de audio. Si **pause** vale 1, detiene cualquier sonido que se esté reproduciendo. Si vale 0, reanuda la reproducción. La función **ASND_Is_Paused()** permite comprobar si el subsistema se encuentra pausado, devolviendo un entero con valor 1 si lo está, o con valor 0 en caso contrario.

Por último, la función **ASND_End()** detiene el subsistema de audio. Es conveniente usar esta función antes de finalizar la aplicación.

6.3. Reproducción de MP3. La librería Mp3Player

Para poder reproducir ficheros de audio en formato MP3, es necesario incluir la librería Mp3Player, añadiendo la sentencia `#include <mp3player.h>` en el código fuente y -lmad a la variable LIBS del fichero Makefile.

Para inicializar la librería, se utiliza la función **MP3Player_Init()**, que debe ser usada antes que cualquier otra función de la librería.

6.3.1. Reproducción de un fichero MP3 desde el código fuente

El primer paso es añadir la sentencia `%.mp3.o : %.mp3` tras la sección "main targets" del fichero Makefile que se encuentra cerca del final del fichero, como se muestra a continuación:

```
#---------------------------------------------------------
# main targets
#---------------------------------------------------------
$(OUTPUT).dol: $(OUTPUT).elf
$(OUTPUT).elf: $(OFILES)

%.mp3.o    :    %.mp3
#---------------------------------------------------------

      @echo $(notdir $<)
      $(bin2o)

-include $(DEPENDS)
```

Con esto, se ordena al compilador que debe convertir cualquier fichero con la extensión .mp3 que encuentre en el directorio indicado en la variable DATA en un fichero de código objeto (un fichero con extensión .o). El compilador guardará este fichero en un directorio "build", junto con un fichero de cabeceras que tendrá el nombre del fichero mp3, seguido de _mp3. Por ejemplo, si el fichero mp3 original tenía el nombre de "sample", el compilador creará dos ficheros llamados "sample_mp3.o" y "sample_mp3.h". Por supuesto, también puede usarse la herramienta *raw2c* descrita en el apartado 5.3.2.

El siguiente paso consiste en incluir el fichero de cabeceras generado en el código fuente. Siguiendo con el ejemplo anterior, habría que añadir la sentencia `#include "sample_mp3.h"`. El fichero "sample_mp3.h" contendrá un *buffer* con el nombre "sample_mp3" y una variable "sample_mp3_size" con la talla de dicho *buffer*.

Para reproducir el audio en mp3 desde un *buffer*, la librería Mp3Player dispone de la función **MP3Player_PlayBuffer**:

```
MP3Player_PlayBuffer(buffer, talla_buffer, NULL);
```

Donde `buffer` es el puntero al *buffer* de datos y `talla_buffer` es la talla de `buffer`. El tercer parámetro puede ser NULL. De esta forma la siguiente sentencia reproducirá el audio en mp3 desde el *buffer* "sample_mp3":

```
MP3Player_PlayBuffer(sample_mp3, sample_mp3_size, NULL);
```

6.3.2. Reproducción de un fichero mp3 desde un directorio

Para reproducir un fichero mp3 desde un directorio, la librería Mp3Player dispone de la función MP3Player_PlayFile:

```
MP3Player_PlayFile(fichero, lector, NULL);
```

Donde `fichero` es un puntero al fichero mp3 y `lector` es la función a usar para leer el fichero. El tercer parámetro puede ser NULL. Se puede definir la función `lector` como se muestra en el siguiente ejemplo:

```
s32 lector(void *f, void *dat, s32 size)
{
     return fread(dat, 1, size, (FILE*) f);
}
```

Así, para reproducir un fichero mp3, se usarían las siguientes sentencias (tras haber inicializado el sistema de ficheros y el audio):

```
FILE *f;
f=fopen("sample.mp3", "r");
MP3Player_PlayFile(f, lector, NULL);
```

Esto reproducirá el fichero "sample.mp3", que se encuentra en el mismo directorio que el ejecutable, usando la función `lector`, definida anteriormente.

6.3.3. Otras funciones de la librería Mp3Player

La función **MP3Player_Volume(volume)** permite establecer el volumen general del audio mp3, siendo **volume** un entero positivo.

La función **MP3Player_Stop()** detiene cualquier sonido que la librería Mp3Player esté reproduciendo. Por otro lado, la función **MP3Player_IsPlaying()** devolverá un booleano con valor *true* si la librería se encuentra reproduciendo algún sonido, y *false* en caso contrario.

6.4. Reproducción de ficheros MOD. La librería ModPlay

El formato de fichero MOD, con extensión .mod, es un formato de fichero para representación de música diseñado para secuenciadores. Este formato apareció en la década de 1980 y fue difundido principalmente por las computadoras Amiga y Atari ST. Es similar a MIDI, salvo que MOD incluye sus propias muestras de audio, con lo que suena siempre igual, independientemente de la máquina donde se reproduzca. Es usado habitualmente para la música en video-juegos para videoconsolas, debido a su bajo coste en memoria y uso de CPU y se sigue usando hoy en día en consolas como la Nintendo DS <20>.

Para reproducir audio en formato MOD en las aplicaciones para Wii, es necesario incluir la librería ModPlay. Para esto hay que añadir la sentencia `#include <gcmodplay.h>` en el código fuente y -lmodplay en la variable LIBS del fichero Makefile.

6.4.1. Reproducción de un fichero MOD desde *buffer*

Debido a su uso tradicional para música en video-juegos, la librería está pensada para reproducir el audio en formato MOD integrado en el código fuente.

Al igual que se ha explicado en el apartado sobre el formato MP3, hay que indicar al fichero Makefile que se quiere convertir los ficheros con extensión .mod en ficheros de código objeto. Para ello, hay que añadir la sentencia `%.mod.o : %.mod` tras la sección "main targets":

```
#---------------------------------------------------------
# main targets
#---------------------------------------------------------
$(OUTPUT).dol: $(OUTPUT).elf
$(OUTPUT).elf: $(OFILES)

%.mod.o    :     %.mod
#---------------------------------------------------------

      @echo $(notdir $<)
      $(bin2o)

-include $(DEPENDS)
```

Esto convertirá cualquier fichero con extensión .mod en la carpeta indicada en DATA en ficheros de código objeto, que será depositado en el directorio "build" con el nombre del fichero original seguido de _mod. Por ejemplo, a partir del fichero "sample.mod" se creará "sample_mod.o", que contendrá un buffer "sample_mod" y una variable con el valor de la talla de dicho *buffer*, "sample_mod_size".

La librería ModPlay define una estructura **MODPlay**, que almacena la información necesaria para reproducir el audio en formato MOD:

```
typedef struct _modplay {
      MOD mod;
      BOOL playing,paused;
      BOOL bits,stereo,manual_polling;
      u32 playfreq,numSFXChans;
      MODSNDBUF soundBuf;
} MODPlay;
```

Esta estructura contiene distintas variables que permiten controlar el estado de la pista de audio MOD, como saber si se encuentra actualmente en reproducción o pausada (con las variables `playing` y `paused`), saber si la pista se está reproduciendo en estéreo o conocer la frecuencia del audio. La librería dispone de varias funciones para manejar parte de esta información de manera más intuitiva, que se describirán más adelante.

Es necesario crear una variable de la estructura **MODPlay** para poder reproducir una pista de audio en formato MOD. Una vez creada, se pasa su dirección como parámetro a la función **MODPlay_Init**:

```
MODPlay mod;
MODPlay_Init(&mod);
```

Esto inicializará las variables de la estructura **MODPlay** a sus valores por defecto. Una vez inicializado, la función **MODPlay_SetMOD** asignará a la estructura el buffer con los datos del audio. La sintaxis de esta función es la siguiente:

```
MODPlay_SetMOD(estructura_MODPlay, buffer);
```

Donde `estructura_MODPlay` es la dirección de la estructura a la que se quiere asociar el *buffer* con los datos del audio. Así, siguiendo con el ejemplo anterior, si se ha creado un fichero "sample_mod.o" a partir de un fichero "sample.mod" y ha sido incluido en el código fuente, la siguiente sentencia asociaría el buffer "sample_mod" con una estructura MODPlay mod:

```
MODPlay_SetMOD(&mod, sample_mod);
```

Para iniciar la reproducción del audio, se usa la función **MODPlay_Start**, a la que se le pasa como parámetro la dirección de una estructura MODPlay. Esta función reproduce el audio en bucle. Continuando el ejemplo anterior, la sentencia sería:

```
MODPlay_Start(&mod);
```

Al ejecutarse esta sentencia, el audio del fichero "sample.mod" comenzará a reproducirse.

6.4.2. Otras funciones de la librería ModPlayer

La librería ModPlayer dispone de otras funciones para modificar las características del audio. La función **MODPlay_SetFrequency** permite modificar la frecuencia del sonido reproducido por la librería ModPlayer. Su sintaxis es la siguiente:

```
MODPlay_SetFrequency(estructura_MODPlay, frecuencia);
```

Donde `estructura_MODPlay` es la dirección de la estructura MODPlay a la que se ha asociado la reproducción del sonido y `frecuencia` es el valor al que se quiera establecer la frecuencia a la que se reproducirá el sonido, siendo está un número entero con un rango de 1 Hz a 144000 Hz, aunque lo conveniente es trabajar a la frecuencia por defecto de 48000 Hz.

Por defecto, el sonido en Wii es estéreo. Con la función **MODPlay_SetStereo** se puede modificar este parámetro. La función tiene esta sintaxis:

```
MODPlay_SetStereo(estructura_MODPlay, estéreo);
```

Donde, de nuevo, `estructura_MODPlay` es la estructura a la que se ha asociado el sonido a reproducir, y `estéreo` es un parámetro de tipo BOOL que, en caso de ser *false*, el sonido se reproducirá en mono y, en caso de ser *true*, se reproducirá en estéreo.

Para modificar el volumen del sonido se utiliza la función **MODPlay_SetVolume**, cuya sintaxis se muestra a continuación:

```
MODPlay_SetVolume(estructura_MODPlay, volumen, volumensfx);
```

Donde `volumen` es el volumen general para la música y `volumensfx` es el volumen para efectos de sonido. Generalmente, estos dos parámetros tendrán el mismo valor.

Se puede pausar la reproducción del sonido con la función MODPlay_Pause, que tiene la siguiente sintaxis:

```
MODPlay_Pause(estructura_MODPlay, pausa);
```

Donde `pausa` es un parámetro de tipo BOOL que, en caso de tener el valor *true*, pausará la reproducción del sonido asociado a `estructura_MODPlay`. En cambio, en caso de que `pausa` tenga el valor *false*, la reproducción del sonido se reanudará, si se encontraba pausada.

Por último, la función **MODPlay_Stop**, que recibe cómo parámetro la referencia de la estructura MODPlay que se esté utilizando, detendrá la reproducción del sonido.

6.5. Ejemplos anexos a la memoria relativos al capítulo 6

Ejemplo 8: Muestra el uso de las funciones de la clase ModPlay reproduciendo audio en formato MOD.

Ejemplo 9: Muestra cómo reproducir varios ficheros mp3 al pulsar con el puntero del Wiimote sobre una imagen mediante un sencillo juego para niños “Wii Kids Farm” que reproduce sonidos de animales. Hace uso de wiisprite para crear las imágenes, como se ha explicado en el capítulo 5.

Capítulo 7. Conclusiones

El objetivo de esta memoria es el de servir de guía para comenzar a desarrollar aplicaciones en la consola Nintendo Wii. A lo largo de sus capítulos, se han mostrado algunas de las librerías disponibles para usar las distintas funcionalidades de la consola, comenzando por el uso de los controles, y añadiendo posteriormente diferentes elementos multimedia. En el capítulo 3 se ha mostrado como usar las funciones de la librería wiipad, que permiten utilizar el Wiimote y sus expansiones. En el capítulo 4 se ha mostrado cómo acceder al sistema de ficheros y cómo manejar ficheros XML, útiles para almacenar datos de las aplicaciones. En el capítulo 5 se ha mostrado el uso de la librería wiisprite para trabajar con imágenes en forma de *sprites*, pudiendo cargar estas imágenes desde ficheros o incluyéndolas en el código fuente, así como la aplicación de transformaciones tales como desplazamientos, rotación y escalado y añadiendo la posibilidad de detectar colisiones entre distintas imágenes. Por último, en el capítulo 6 se ha mostrado como incluir audio en las aplicaciones en dos formatos distintos. Con estos conocimientos es posible desarrollar juegos y aplicaciones con gráficos 2D, capaces de manejar datos en ficheros y que sean controlables con el Wiimote, pudiendo usar la mayoría de funcionalidades de este peculiar mando, siendo la más característica la detección de movimiento.

Quedan sin cubrir otros aspectos y funcionalidades de la consola Nintendo Wii, tales como la comunicación en red mediante wifi. Existen además otras librerías gráficas aparte de la librería wiisprite, mostrada en el capítulo 5, muchas de ellas no incluidas en devkitPro (por ejemplo, libwiigui y GRRLIB). Dada la variedad y cantidad de librerías existentes en la comunidad de desarrollo de software no oficial que pueden ser importadas al paquete devkitPro, se ha decidido dejar estas librerías (salvo wiisprite) fuera de la guía y limitarla a explicar la programación de las funcionalidades más básicas de esta plataforma de juegos. Queda como posible vía de futuros trabajos el estudio del resto de librerías gráficas disponibles, así como la documentación y estudio de librerías que permitan trabajar con gráficos 3D. También quedan como posible vía para futuros trabajos el estudio de librerías que permitan la comunicación por red, así como otras librerías de devkitPro que controlan funcionalidades de bajo nivel, tales como el lector óptico o los puertos USB, todas ellas bajo el grupo de librerías libogc, que por su complejidad y escasa documentación, se ha decidido dejar fuera de esta guía.

La idea original era la de proporcionar una guía que resultara útil en general a cualquier lector interesado en iniciarse en el desarrollo de aplicaciones para Wii teniendo unos conocimientos básicos de programación, y dar a conocer la existencia de la comunidad existente en la red dedicada al desarrollo de software no oficial para la plataforma de Nintendo.

Resumen de posibles vías para futuros trabajos:

- Comunicación por red mediante la librería network, incluida en devkitPro (.../libogc/include/network.h).
- Estudio del soporte para programación multihilo otorgado por el conjunto de librerías libogc incluidas en devkitPro, en concreto la librería lwp (.../libogc/include/ogc/lwp.h).
- Documentación y estudio de otras librerías gráficas distintas a wiisprite, que permitan trabajar con primitivas gráficas. Aunque wiisprite puede ser usada para crear una interfaz para una aplicación, existen alternativas más adecuadas como la librería libwiigui <http://wiibrew.org/wiki/Libwiigui/tutorial>. Otra librería gráfica para Wii es GRRLIB, que permite trabajar con primitivas gráficas 3D y 2D <http://wiibrew.org/wiki/GRRLIB>.

- Obtención de datos de los “Mii” mediante la librería libmii y estudio de sus posibilidades <http://wiibrew.org/wiki/Libmii>.
- Estudio del uso de otras expansiones de los controles, tales como la Balance Board o el Wiimote Plus.
- Uso de las librerías libfreetype y libtremor en su versión para PPC para el renderizado de fuentes de texto y reproducción de audio OGG, respectivamente.

Bibliografía y referencias

<1> *devkitPPC* [en línea]. Wiibrew [Consulta: 25 de julio de 2011]. Disponible en web: <http://wiibrew.org/wiki/Devkitpro>.

<2> Bufet Almeida. *PS-JAILBREAK: se alzan las medidas cautelares acordadas en favor de Sony, a la que se condena en costas* [en línea]. [Consulta: 30 de junio de 2011]. Disponible en web: <http://www.bufetalmeida.com/612/psjailbreak.html>.

Bufet Almeida. *Caso Chipspain.com – Swap Magic y Chips Multisistema para Playstation no constituyen delito* [en línea]. [Consulta: 30 de junio de 2011]. Disponible en web: <http://www.bufetalmeida.com/614/chipspain.html>.

<3> *WiiBrew:FAQ* [en línea]. Wiibrew [Consulta: 30 de junio de 2011]. *Is it illegal to install the Homebrew Channel?* Disponible en web: <http://wiibrew.org/wiki/WiiBrew:FAQ>.

<4> *Brick* [en línea]. Wiibrew [Consulta: 25 de julio de 2011]. *Prevention*. Disponible en web: <http://wiibrew.org/wiki/Brick>.

<5> *IBM Broadway* [en línea]. Wikipedia [Consulta: 25 de julio de 2011]. Disponible en web: <http://es.wikipedia.org/wiki/IBM_Broadway>.

<6> *IBM Ships First Microchips for Nintendo's Wii Video Game System* [en línea]. IBM [Consulta: 2 de julio de 2011]. Disponible en web: <http://www-03.ibm.com/press/us/en/pressrelease/20213.wss>.

<7> *Hollywood (graphics chip)* [en línea]. Wikipedia [Consulta: 25 de julio de 2011]. Disponible en web: <http://en.wikipedia.org/wiki/Hollywood_(graphics_chip)>.

<8> *Hardware* [en línea]. Wiibrew [Consulta: 25 de julio de 2011]. *Nintendo Wii Hardware Summary.* Disponible en web: <http://wiibrew.org/wiki/Wii_Hardware>.

Wii [en línea]. Wikipedia [Consulta: 25 de julio de 2011]. *Especificaciones técnicas*. Disponible en web: <http://es.wikipedia.org/wiki/Wii>.

<9> *Hardware/Starlet* [en línea]. Wiibrew [Consulta: 25 de julio de 2011]. *Boot*. Disponible en web: <http://wiibrew.org/wiki/Starlet>.

<10> *Wiimote* [en línea]. Wiibrew [Consulta: 2 de julio de 2011]. Disponible en web: <http://wiibrew.org/wiki/Wiimote>.

<11> *Bannerbomb* [en línea]. ElOtroLado.net [Consulta: 4 de julio de 2011]. Disponible en web: <http://www.elotrolado.net/wiki/Bannerbomb>.

<12>*HackMii Installer* [en línea]. ElOtroLado.net [Consulta: 4 de julio de 2011]. Disponible en web: <http://www.elotrolado.net/wiki/HackMii Installer>.

<13> *Bootmii* [en línea]. ElOtroLado.net [Consulta: 5 de julio de 2011]. Disponible en web: <http://www.elotrolado.net/wiki/Bootmii>.

<14> *Getting Started/devkitPPC* [en línea]. DevkitPro.org [Consulta: 7 de julio de 2011]. Disponible en web: <http://devkitpro.org/wiki/Getting_Started/devkitPPC>.

<15> *MinGW Minimalist GNU for Windows* [en línea]. [Consulta: 7 de julio de 2011]. Disponible en web: <http://www.mingw.org/wiki/MSYS>.

<16> *vt100 terminal escape codes* [en línea]. [Consulta: 11 de julio de 2011]. Disponible en web: <http://ilkerf.tripod.com/cdoc/vt100ref.html>.

<17> *Mini-XML Programmers Manual, Version 2.6* [en línea]. [Consulta: 9 de julio de 2011]. Disponible en web: <http://www.minixml.org/mxml.html>.

<18> *Libwiisprite/tutorial* [en línea]. Wiibrew [Consulta: 9 de julio de 2011]. Disponible en web: <http://wiibrew.org/wiki/Libwiisprite/tutorial>.

<19> *Sprite (videojuegos)* [en línea]. Wikipedia [Consulta: 9 de julio de 2011]. Disponible en web: <http://es.wikipedia.org/wiki/Sprite_(videojuegos)>.

<20> *MOD (formato de archivo)* [en línea]. Wikipedia [Consulta: 25 de julio de 2011]. Disponible en web: <http://es.wikipedia.org/wiki/MOD_(formato_de_archivo)>.

Otras referencias bibliográficas de utilidad:

CRAVIOTO BABÉ, Ingancio y CAPILLA HERRERO, Marian. "Arquitectura y programación de consolas de Juegos: Wii". Trabajo para la asignatura Integración de medios digitales. Escuela Técnica Superior de Informática, UPV, 2009.

ZOMEÑO PERONA, José y DEL HIERRO HERNÁNDEZ, Cristina. "Aplicación de audio para Nintendo Wii". Trabajo para la asignatura Integración de medios digitales. Escuela Técnica Superior de Informática, UPV, 2010.