Tactics DS

MEMORIA

Víctor Grau Moreso

Mark Holland

Lluís Ulzurrun de Asanza Sàez

2014-2015

Índice

1. Introducción

1.1. FMAW Framework

El desarrollo de *homebrew* sobre Nintendo DS supone un reto importante al programador, quien además de trabajar a un nivel menos abstracto y más cercano a los componentes físicos de la consola se enfrenta a una capacidad de cálculo y memoria reducida incluso para la época en la que la consola salió al mercado[1]. La falta de una abstracción adecuada de los componentes es una fuente de errores que si no son detectados y corregidos a tiempo pueden suponer importantes retrasos en el desarrollo de cualquier proyecto.

Debido a estas limitaciones decidimos que para poder llevar a cabo un desarrollo adecuado del proyecto era necesario disponer de antemano de una serie de herramientas y librerías para facilitarnos una mínima abstracción del *hardware* del equipo, de manera que diseñamos y desarrollamos una librería que ejerce de intermediaria entre las aplicaciones y las utilidades de bajo nivel ofrecidas por `libnds`; a la cual hemos llamado `FMAW framework`.

`FMAW framework` dispone no únicamente de una capa de abstracción de los componentes físicos, sino también de algunas herramientas sencillas de debug y una implementación de la aritmética básica[2] en coma fija (implementada sobrecargando los operadores aritméticos básicos). Además está diseñado para poder ser portado a cualquier otra plataforma sin necesidad de rediseñar ninguna de las interfaces existentes.

1.2. Tactics DS

Históricamente han abundado los juegos basados en tableros o rejillas. Desde el ajedrez[3] hasta los actuales Fire Emblem[4], pasando por un sinfín de juegos de mesa y videojuegos. Esta clase de juegos resultan de especial interés por su componente estratégico y su afinidad a videoconsolas portátiles, en especial las de Nintendo[5]. Resultan igualmente interesantes desde el punto de vista del programador ya que una arquitectura bien diseñada para este tipo de juegos permite un desarrollo más sencillo.

`Tactics DS` pretende ser un juego comparable a algunos de los clásicos del género rol táctico por lo que incluye características como variedad de unidades, inteligencia artificial que pueda jugar contra jugadores humanos, niebla de guerra que impida la visión completa del escenario, diferentes tipos de terreno con sus costes de visión y movimiento asociados, variedad de mapas y capacidad de reproducción de partidas previas, entre otras.

En esta memoria detallamos el funcionamiento de diversos componentes de nuestra librería así como documentamos el desarrollo de `Tactics DS`, recopilamos los problemas encontrados durante el desarrollo, exponemos nuestras conclusiones y finalmente proponemos trabajo futuro a realizar sobre las herramientas que detallamos.

[1] Japón, 24 de noviembre de 2004. Sony lanzó la PlayStation Portable el 12 de diciembre de 2004, con un procesador con una frecuencia diez veces mayor y cuadruplicando la capacidad de la memoria RAM respecto a la Nintendo DS.

[2] Suma, diferencia, producto y cuociente

[3] Siglo XV.

[4] 19 de abril de 2012.

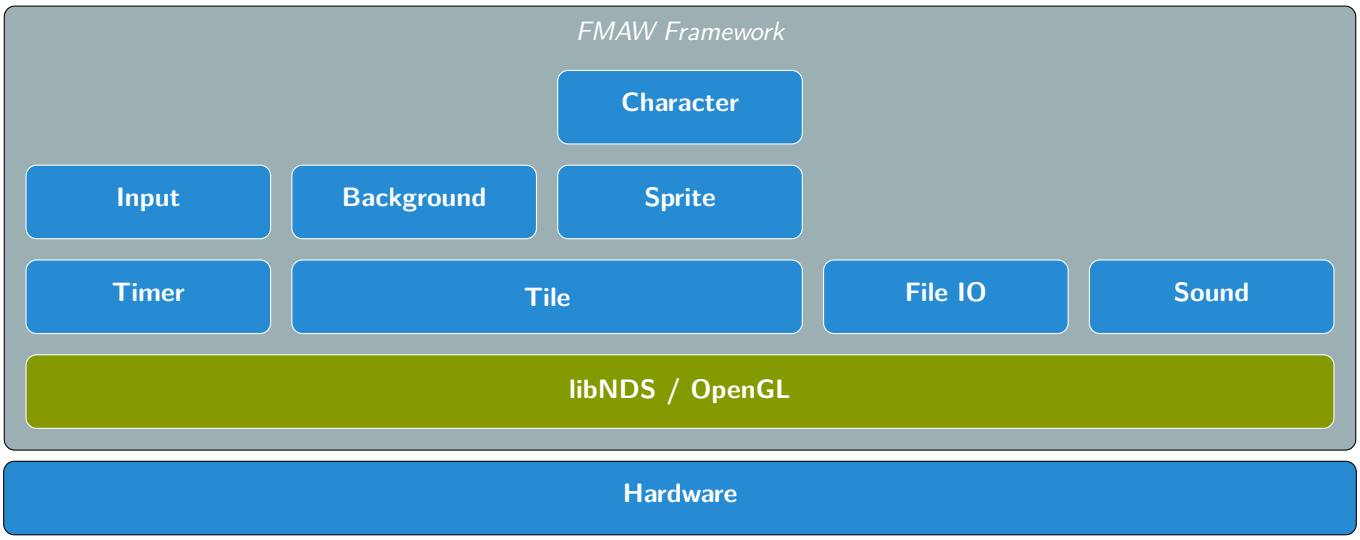
[5] "Fire Emblem: The Binding Blade" (2002, Game Boy Advance), "Fire Emblem" (2003, Game Boy Advance), "Fire Emblem: The Sacred Stones" (2004, Game Boy Advance), "Fire Emblem: Shadow Dragon" (2008, Nintendo DS) "Fire Emblem Awakening" (2012, Nintendo 3DS), "Fire Emblem If" (2015, Nintendo 3DS), "Advance Wars" (2001, Game Boy Advance), "Advance Wars 2" (2003, Game Boy Advance), "Advance Wars: Dual Strike" (2005, Nintendo DS), "Advance Wars: Days of Ruin" (2008, Nintendo DS).

2. FMAW Framework

Bajo el espacio de nombres `FMAW` se encuentra el componente fundamental sobre el que se ha desarrollado `Tactics DS`. Esta librería ha sido diseñada y desarrollada desde cero por nosotros y toma prestados conceptos usados en otras librerías como `SpriteKit` o `UIKit`. Se compone de cuatro componentes fundamentales sobre los que se construyen utilidades de alto nivel:

1. `Timer`: ofrece un API para encolar tareas que serán ejecutadas en el futuro, de forma repetida o no.
2. `Tile`: permite abstraer el concepto de *tile* que maneja internamente la Nintendo DS, lo que facilita la compilación de los proyectos basados en `FMAW framework` en otras plataformas.
3. `IO`: habilita el manejo de un fichero de archivos, virtual o no (de especial utilidad cuando se trabaja con emuladores de Nintendo DS que no disponen de sistema de archivos real).
4. `Sound`: maneja todos los efectos de sonido y música de fondo.

Sobre `Timer` se construye `Input`, una abstracción del sistema de entrada de usuario de la Nintendo DS que permite seguir patrones de diseño orientados a eventos. A su vez descansan sobre `Tile` las abstracciones `Background` y `Sprite`; y sobre la última, `Character`. Independiente del resto de componentes se halla la clase `FixedReal`, una implementación eficiente de números en coma fija que permite operar de forma rápida y con poca pérdida de precisión[6].



[6]En función de cuántos bits decidan destinarse a la parte decimal la precisión será mayor o menor, en detrimento del valor máximo de la parte entera.

2.1. Componentes

2.1.1. Timer

`Timer` ofrece un API para encolar tareas que se deben realizar en el futuro, de forma repetida o no. Las tareas son instancias de la plantilla `std::function<void(int)>`[7].

Listing 1: API ofrecida por `Timer`.

```
namespace FMAW {
namespace Timer {
/**
 * Enqueues given function so it will be called
 * @param  callback   Function to be called. Method will be given its ID as
 *                    unique parameter.
 * @param  delta      Time in ms to wait before calling the function.
 *                    Note that precision is 5ms!
 * @param  repetitive Whether this callback should be enqueued again after
 *                    being called.
 * @return            An identifier to cancel enqueued action later.
 */
int enqueue_function(std::function<void(int)> callback, unsigned int delta,
                     bool repetitive);

/**
 * Dequeues given function so it won't be called again.
 * @param  id Function to be dequeued.
 * @return    Whether function was dequeued properly or not.
 */
bool dequeue_function(int id);
}  // namespace Timer
}  // namespace FMAW
```

Las tareas forman parte de una estructura que almacena entre otros datos el momento en el que debe ser ejecutada por primera vez. Se mantienen en una lista instancias de esta estructura. Cada 5ms se recorre la lista de tareas y se comprueba si es necesario ejecutar o no cada tarea. En caso de ser ejecutada se verifica si debe volver a ser ejecutada de nuevo o no y dependiendo del caso se retira la tarea de la lista o se encola de nuevo con un momento de inicio actualizado. La precisión de este sistema es suficiente para la mayoría de los casos de uso y durante el desarrollo de `Tactics DS` no hemos tenido ningún problema de falta de precisión ni caídas de rendimiento por el uso de esta API de tareas programadas.

Cada tarea encolada recibe un identificador numérico diferente que puede usarse para cancelar la ejecución de ésta en cualquier momento. La propia tarea recibe el identificador al ser llamada para que ésta pueda desencolarse a sí misma si detecta que su ejecución ya no es necesaria.

Esta API requiere que al inicio de la ejecución del programa se invoque el método `init` del espacio de nombres `FMAW::Timer`. Esto es debido a que el sistema necesita alguna manera de obtener la hora actual para poder calcular el intervalo de tiempo transcurrido entre cada comprobación de la lista y para ello recurre a uno de los relojes físicos de la Nintendo DS[8]. Otro requisito para poder usar esta interfaz es que se invoque al método `check` del espacio de nombres `FMAW::Timer` con suficiente frecuencia. Esto se logra ejecutando la llamada necesaria antes de dibujar cada fotograma.

[7] `std::function<...>` es una de las nuevas plantillas disponibles en C++11.
[8] Cabe señalar que la Nintendo DS cuenta con únicamente cuatro relojes, de manera que quedan tres restantes para otros menesteres.

2.1.2. Input

`Input` implementa una capa de abstracción para la entrada del usuario, con una interfaz similar a la de `Timer`. En esta ocasión la API está disponible en el espacio de nombres `FMAW::Input` y ofrece tres formas en encolar por cada botón disponible en la consola: al pulsar, mientras está pulsado y al soltar.

Listing 2: API ofrecida por `Input`.

```
namespace FMAW {
namespace Input {
/**
 * Registers a callback so it will be called when button A is pressed.
 * @param callback  Function to be called when button A is pressed.
 * @return Identifier of the callback so it can be registered later.
 */
int onButtonAPressed(std::function<void(void)> callback);

/**
 * Registers a callback so it will be called while button A is pressed.
 * @param callback  Function to be called while button A is pressed.
 * @return Identifier of the callback so it can be registered later.
 */
int whileButtonAPressed(std::function<void(void)> callback);

/**
 * Registers a callback so it will be called when button A is released.
 * @param callback  Function to be called when button A is released.
 * @return Identifier of the callback so it can be registered later.
 */
int onButtonAReleased(std::function<void(void)> callback);

/**
 * Registers a callback that will be called when a touch is detected.
 * Coordinates of the touch will be given as parameters.
 * @param callback  Function to be called. Receives x and y cooordinates of
 *                  touch as parameters.
 * @return Identifier of the callback so it can be registered later.
 */
int onTouchPressed(std::function<void(int, int)> callback);

/**
 * Registers a callback that will be called when a touch is moved.
 * New coordinates of the touch will be given as parameters.
 * @param callback  Function to be called. Receives new x and y cooordinates of
 *                  touch as parameters.
 * @return Identifier of the callback so it can be registered later.
 */
int onTouchMoved(std::function<void(int, int)> callback);

/**
 * Registers a callback that will be called when a touch finishes.
 * Coordinates of the last position of finished touch will be given as parameters.
 * @param callback  Function to be called. Receives x and y cooordinates of
 *                  last position of touch as parameters.
 * @return Identifier of the callback so it can be registered later.
 */
int onTouchReleased(std::function<void(int, int)> callback);
```

```
    /**
     * Unregisters callback with given identifier so it won't be called again.
     * @param identifier Identifier of callback to be unregistered.
     */
    bool unregisterCallback(int identifier);

    }  // namespace Input
    }  // namespace FMAW
```

En el fragmento de código 2 se muestran métodos para encolar manejadores para cada tipo de interacción con el botón A. Los toques en la pantalla táctil o los movimientos y clicks del ratón en un ordenador se gestionan también mediante tres tipos de eventos diferentes: al tocar la pantalla (o al pulsar el botón del ratón), al desplazarse por la pantalla (o al mover el cursor del ratón) y al liberar la pantalla (o soltar el botón del ratón).

La implementación actual no realiza ningún seguimiento intensivo de los movimientos del *stylus* por la pantalla ni tampoco detecta cuándo se libera la pantalla. Los métodos `onTouchMoved` y `onTouchReleased` son en realidad una forma alternativa de encolar un manejador para los eventos de "se mantiene pulsada la pantalla" y "se pulsa la pantalla". Adaptar la implementación actual para cambiar este comportamiento es trivial sin embargo por falta de tiempo no hemos podido desarrollar un framework más completo de seguimiento de *stylus*[9].

Esta API, de manera análoga a `Timer`, requiere que se llame al método `check` con cierta frecuencia, de manera que en nuestra implementación recurrimos a `Timer` para encolar la ejecución de `Input::check` cada 5ms. De forma similar a como se planteó en `Timer`, el método `check` recorre la lista de manejadores registrados y ejecuta los que corresponda en función del historial de pulsaciones de botones de la consola.

Al registrarse un manejador éste recibe un identificador numérico que permite desregistrarlo en cualquier momento, aunque a diferencia de `Timer` los manejadores no reciben dicho identificador, de modo que recae en el desarrollador la responsabilidad de almacenarlo para su posterior uso.

Cabe destacar que los manejadores para las pulsaciones de los botones físicos y los manejadores para reaccionar ante el *stylus* tienen firmas diferentes. Los primeros son funciones que no reciben ningún parámetro mientras que los segundos reciben dos argumentos, a saber, la posición en el eje horizontal y la situación en el eje vertical del punto que ha sido (o está siendo) tocado con el *stylus*, expresada en desplazamiento en píxeles desde la esquina superior izquierda de la pantalla inferior de la consola.

Listing 3: Menú básico implementado mediante las utilidades de `libnds`.

```
bool menuIsActive           = true;
bool buttonAWasPressed      = false;
bool buttonBWasPressed      = false;
bool buttonSelectWasPressed = false;
bool buttonStartWasPressed  = false;

void processInput() {
    scanKeys();
    int keysh = keysHeld();

    // If menu is active.
    if (menuIsActive) {
        // If start is pressed, close menu.
        if (keysh & KEY_START) {
```

[9]Tampoco nos pareció de especial interés invertir más tiempo en este modo de interacción ya que la pantalla táctil de la Nintendo DS utiliza tecnología que no ofrece una precisión comparable a la que se encuentra en cualquier teléfono inteligente o tableta actual.

```
            buttonStartWasPressed = true;
        } else if (buttonStartWasPressed) {
            // Only change menu when button start is released.
            menuIsActive = false;
            buttonAWasPressed = false;
            buttonBWasPressed = false;
            buttonStartWasPressed = false;
            buttonSelectWasPressed = false;
        } else {
            // We are in the menu and we are not closing it.
            // If A is pressed mark it as pressed.
            if (keysh & KEY_A) buttonAWasPressed = true;
            // If A was pressed but it is not pressed anymore, trigger handler.
            if (buttonAWasPressed && !(keysh & KEY_A)) {
                menuIsActive = false;
                buttonAWasPressed = false;
                buttonBWasPressed = false;
                buttonStartWasPressed = false;
                buttonSelectWasPressed = false;
                _newGame();
            }
            // If A is not pressed uncheck it.
            if (!(keysh & KEY_A)) buttonAWasPressed = false;

            // If B is pressed mark it as pressed.
            if (keysh & KEY_B) buttonBWasPressed = true;
            // If B was pressed but it is not pressed anymore, trigger handler.
            if (buttonBWasPressed && !(keysh & KEY_B)) {
                menuIsActive = false;
                buttonAWasPressed = false;
                buttonBWasPressed = false;
                buttonStartWasPressed = false;
                buttonSelectWasPressed = false;
                _loadGame();
            }
            // If B is not pressed uncheck it.
            if (!(keysh & KEY_B)) buttonBWasPressed = false;

            // If Select is pressed mark it as pressed.
            if (keysh & KEY_SELECT) buttonSelectWasPressed = true;
            // If Select was pressed but it is not pressed anymore, trigger handler.
            if (buttonSelectWasPressed && !(keysh & KEY_SELECT)) {
                menuIsActive = false;
                buttonAWasPressed = false;
                buttonBWasPressed = false;
                buttonStartWasPressed = false;
                buttonSelectWasPressed = false;
                _versusGame();
            }
            // If A is not pressed uncheck it.
            if (!(keysh & KEY_SELECT)) buttonSelectWasPressed = false;
        }
    }
}
```

Listing 4: Menú básico implementado mediante las utilidades de `FMAW framework`.

```
int newID, loadID, versusID;
bool menuActive = false;

auto newGame    = [](){
    _newGame();
};
auto loadGame   = [](){
    _loadGame();
};
auto versusGame = [](){
    _versusGame();
};

auto enqueueMenu = [newGame, loadGame, versusGame, &newID, &loadID,
&versusID](){
    newID    = FMAW::Input::onButtonAReleased(newGame);
    loadID   = FMAW::Input::onButtonBReleased(loadGame);
    versusID = FMAW::Input::onButtonSelectReleased(versusGame);
};

auto dequeueMenu = [&newID, &loadID, &versusID](){
    FMAW::Input::unregisterCallback(newID);
    FMAW::Input::unregisterCallback(loadID);
    FMAW::Input::unregisterCallback(versusID);
    newID    = -1;
    loadID   = -1;
    versusID = -1;
};

auto toggleMenu = [enqueueMenu, dequeueMenu, &menuActive](){
    if (menuActive) dequeueMenu();
    else enqueueMenu();
    menuActive = !menuActive;
};

FMAW::Input::onButtonStartReleased(toggleMenu);
```

Esta API resulta particularmente útil cuando se trabaja con manejadores que únicamente deben ser activados bajo ciertas circunstancias. El fragmento de código 3 muestra la implementación de un manejador para un menú mediante las utilidades originales de `libnds`, mientras que el fragmento 4 muestra la misma funcionalidad implementada mediante `FMAW framework`. Puede observarse cómo esta tarea requiere alrededor de 70 líneas de código mediante `libnds` mientras que con la ayuda de `FMAW framework` esto mismo se puede realizar en apenas 30 líneas de código, siendo éste también más sencillo[10].

[10] Únicamente contiene una expresión condicional, se utilizan únicamente cuatro variables además de los manejadores, siendo tres de éstas los identificadores de los manejadores, el bloque más largo tiene únicamente seis líneas y el mayor nivel de identación es uno.

2.1.3. Tile

`Tile` es una abstracción del sistema de teselas de la Nintendo DS. Permite cargar en memoria de vídeo imágenes a través de un API más sencilla que la ofrecida por `libnds`, haciéndose ésta cargo del manejo de la posición de memoria correspondiente a cada imagen y paleta. Es un componente esencial de `FMAW framework` puesto que cualquier contenido que se muestre en pantalla debe pasar antes por este componente y ser registrado como *tile*. Uno de los puntos pendientes de este componente es permitir registrar de forma individual paletas e imágenes, de manera que diversas imágenes compartan una misma paleta.

2.1.4. Background

`Background` es una abstracción del mecanismo de fondos de la Nintendo DS. Soporta todas las funciones habilitadas por `libnds` a través de una interfaz más sencilla basada en instancias de la clase `Background` en lugar de acceso directo a los registros de memoria de vídeo.

Cada fondo tiene asociado un conjunto de 2^{10} *tiles* aunque únicamente 768 de éstas se muestran en pantalla[11] y permite acceder a estos de forma directa. Internamente no se almacena un vector de direcciones de memoria sino que se aprovecha la estructura de los registros de memoria permitiendo guardar únicamente la dirección de memoria del primer *tile* de cada fondo.

2.1.5. Sprite

`Sprite` es una abstracción del sistema de *sprites* de la Nintendo DS. Su motivación original fue agrupar todas las operaciones con máscaras de bits en un único lugar para evitar errores al manipularlas erróneamente[12]. La clase `Sprite` tiene asociada un `Tile` y permite realizar todas las operaciones comunes sin necesidad de acceder al `Tile` subyacente.

2.1.6. Character

La clase `Character` no abstrae ningún elemento de `libnds` sino que es una construcción sobre `Sprite` enfocada a ofrecer métodos útiles para trabajar por personajes de un juego. Ofrece, entre otros, métodos para animar el cambio de posición de un personaje. Este método internamente recurre a `Timer` para realizar una transición entre estados fluida.

2.1.7. File IO

El espacio de nombres `FMAW::IO` ofrece una versión propia de los métodos tradicionales de lectura y escritura de ficheros: `fopen`, `fclose`, `fprintf`, `fscanf` y `fflush`. Tiene su origen en un problema de compatibilidad de los emuladores con el sistema de ficheros de los cartuchos de la Nintendo DS.

[11] A falta de una confirmación oficial por parte de Nintendo nuestra hipótesis es que cada fondo dispone de 2^{10} *tiles* debido a que para acceder a los 768 visibles en pantalla es necesario un puntero de 10 bits (con nueve únicamente se podría acceder a un máximo de 512 *tiles*) y con la finalidad de facilitar el trabajo a los ingenieros que desarrollaron el *hardware* y el *firmware* de la consola se añadieron los 256 *tiles* adicionales. Esta teoría es consistente con la existencia de bits desplazamiento vertical que permiten mostrar dichos *tiles* en pantalla y de bits de desplazamiento horizontal que recurren los *tiles* del lado opuesto en lugar de tener un conjunto de *tiles* adicionales en ambos lados.

[12] Ciertas funciones del sistema de teselas de la Nintendo DS se habilitan mediante manipulación de diversos bits. Este tipo de operaciones son muy propensas a provocar fallos cuyas consecuencias son difícilmente previsibles y siendo una ardua tarea encontrar el origen del error.

Sobre una consola real la librería `libfat` permite leer y escribir contenidos en la `flashcard`[13], sin embargo sobre un emulador es necesario recurrir a `nitrofs` para poder escribir ficheros. `nitrofs` no es compatible con las consolas reales.

Existiendo esta limitación nuestro objetivo con `FMAW::IO` es superarla ofreciendo una interfaz idéntica a la tradicional pero que podamos en un futuro implementar de tal manera que dependiendo de si el *software* se está ejecutando en una consola real o en un emulador cargue una librería u otra o incluso almacene en un sistema de ficheros en red los datos.

2.1.8. Sound

El espacio de nombres `FMAW::Sound` ofrece una capa de abstracción sobre `maxmod`, la librería *de facto* para reproducir sonido en `libnds`. Este espacio de nombres ofrece métodos para registrar efectos de sonido y reproducir tanto música de fondo como efectos especiales. La finalidad de `FMAW::Sound` únicamente es abstraer `maxmod` para facilitar la implementación de `FMAW framework` en otras plataformas.

2.2. Soporte multi plataforma

Al diseñar el *framework* uno de los puntos que más se tuvo en cuenta fue que debía ofrecer un API independiente de la plataforma en la que se fuese a compilar y ejecutar el videojuego, de manera que cualquier tipo de identificador de las librerías de bajo nivel debía ser abstraído a un identificador de `FMAW framework` y todas las estructuras de datos usadas por estas librerías debían ser reemplazadas por otras nuevas que además pudiesen ser extendidas en un futuro. Esto se refleja en la existencia de ficheros `.cpp`, `.fds` y `.fgl`.

Para permitir compilación en múltiples plataformas debíamos reescribir el `Makefile` de manera que se compilase un conjunto de ficheros diferentes en función de si se deseaba compilar para Nintendo DS y otra plataforma, sin embargo no fue posible adaptarlo debido a las dependencias del mismo. Llegamos a una solución alternativa: dado que todos los ficheros con extensión `.cpp` iban a ser compilados para la Nintendo DS, y ante la incapacidad de crear subdirectorios o modificar la jerarquía del proyecto, definimos la constante del preprocesador `NDS` únicamente si se iba a compilar el proyecto para la Nintendo DS y la constante `OPENGL` en caso contrario; de manera que nuestros archivos `.cpp` tan sólo deben comprobar cuál de las dos constantes está definida e incluir el fichero correspondiente a la implementación de `FMAW framework` para la plataforma deseada.

Listing 5: `fmaw_io.cpp`.

```
#ifdef NDS
#include "./fmaw_io.fds"
#elif OPENGL
#include "./fmaw_io.fgl"
#endif
```

Debe tenerse en cuenta que la Nintendo DS es una consola peculiar por su doble pantalla, siendo sólo una de éstas táctil. Esto provoca que los desarrolladores diseñen los juegos de forma diferente, combinando ambas pantallas o dándoles usos completamente diferentes. Al diseñar un juego compatible con Nintendo DS y PC debe tenerse en cuenta que en PC no se va a disponer de una segunda pantalla, de modo que si bien a nivel binario no sería necesario hacer ningún cambio ya que `FMAW framework` se encargaría de simular la doble pantalla, a nivel de experiencia de usuario sería conveniente hacer ciertos ajustes para que el disfrute sea el mismo independientemente de la plataforma. Para este fin también se puede recurrir a las constantes `NDS` y `OPENGL`, adaptando el juego ligeramente en función de la plataforma.

[13] No sabemos qué librería usan los juegos comerciales, probablemente una diseñada por `Intelligent Systems`, empresa desarrolladora interna de Nintendo a cargo del kit de desarrollo oficial de la Nintendo DS.

2.3. Debug

`libnds` carece de herramientas de debug. Si bien es cierto que se puede utilizar `gdb` para analizar la ejecución de una aplicación, requiere aprender a usar una nueva tecnología bastante compleja y nuestra disponibilidad de tiempo no nos permitía hacer tal cosa, de modo que era imperativo desarrollar un conjunto de utilidades más sencillas que nos permitiesen localizar errores con más facilidad y perfilar el uso de memoria.

Concretamente desarrollamos una función variádica `printf` con firma idéntica a la función `printf` de C, de manera que todos los mensajes que queramos mostrar por consola pasan por una misma función que se puede reimplementar para escribir en fichero, enviar datos a un servidor o utilizar como destino cualquier dispositivo de salida en función de las necesidades de cada momento. Durante el desarrollo hemos utilizado esta función como abstracción de `nocashMessage`, añadiendo soporte para cadenas con formato, del que carece `nocashMessage`.

También se ha desarrollado un pequeño conjunto de utilidades para imprimir por pantalla el valor de los registros internos de la consola en formato binario, de manera que es más sencillo localizar errores al aplicar máscaras de bit y otras operaciones de bajo nivel que suelen provocar errores en lugares inesperados.

Por último se ha incorporado un sistema de seguimiento del uso de memoria, `memtrack`, de manera que en tiempo de ejecución es posible saber el uso de memoria actual del `software`. Esto es especialmente útil para detectar fugas de memoria que fácilmente pueden saturar la memoria del dispositivo y detener la ejecución de la aplicación. Esta librería ha sido adaptada para usar las herramientas de salida de `FMAW::Debug` en lugar de la salida estándar.

2.3.1. Testing

Inicialmente nuestra intención era disponer de una librería totalmente cubierta por tests unitarios, sin embargo eso probó ser impracticable por la falta de tiempo, de manera que únicamente hicimos tests unitarios del componente más matemático del *framework*: la implementación de aritmética en coma fija. El resto de componentes permanecen sin ningún tipo de tests unitarios ni ningún mecanismo automático de verificación de corrección del código.

Para definir los tests unitarios recurrimos a la librería `Catch`, que define un conjunto de *macros* mediante el cual se pueden definir los tests unitarios. La librería da soporte para sistemas de test mucho más complejos que el nuestro sin embargo es lo suficientemente sencilla como para poder llevar a cabo los tests de nuestra implementación de aritmética en coma fija sin mucha dificultad.

2.4. Nuestro primer juego con FMAW framework

En esta sección desarrollamos un pequeño juego para mostrar cómo utilizar FMAW framework. Nos basamos en el juego que se desarrolla en «How to Make a Bouncing Ball Game» [4]. Partiremos de que devkitpro está instalado y ya se pueden compilar aplicaciones para *homebrew*[14]. Además asumiremos que los archivos multimedia propuestos en el tutorial están disponibles en el directorio de trabajo, junto a sus correspondientes gritfiles.

2.4.1. Esquema básico

Descargaremos el esqueleto del proyecto desde https://github.com/Sumolari/TacticsDS/archive/tutorial/start.zip y partiremos del siguiente archivo main.cpp:

Listing 6: main.cpp.

```
//---------------------------------------------------------------------------
// Graphic references
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Background...
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Game objects
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Main code section
//---------------------------------------------------------------------------

/// Sets up graphics.
void setupGraphics(void) { }

/// Updates game logic.
void update_logic() { }

/// Renders graphics.
void update_graphics() { }

/// Main.
int main(void) {
    return 0;
}
```

Lo primero que debemos hacer es importar FMAW framework añadiendo la siguiente línea al inicio del fichero.

```
#include "./FMAW.h"  // Import our awesome framework!
```

A continuación debemos iniciar el framework. Modificaremos la función main de tal manera que se llame a FMAW::init, pasándole las funciones update_graphics y update_logic como *callbacks*. A continuación llamaremos a la función setupGraphics, que cargará en memoria las imágenes que usaremos y por último llamaremos a la función FMAW::start que iniciará el bucle principal del juego.

[14] Hay tutoriales disponibles en http://devkitpro.org.

Listing 7: Función main.

```
/// Main.
int main(void) {
    FMAW::init(update_graphics, update_logic);
    setupGraphics();

    FMAW::start();

    return 0;
}
```

La función update_logic de momento sólo llamará al método FMAW::Timer::check para poder encolar tareas.

Listing 8: Función update_logic.

```
/// Updates game logic.
void update_logic() { FMAW::Timer::check(); }
```

2.4.2. Creación del fondo

Necesitamos cargar las imágenes, para ello debemos importar el código binario de las imágenes. Nos situaremos en la sección de *Graphic references* y añadiremos allí los siguientes *includes*:

Listing 9: Importar gráficos.

```
//---------------------------------------------------------------------------
// Graphic references
//---------------------------------------------------------------------------

#include "./gfx_ball.h"
#include "./gfx_brick.h"
#include "./gfx_gradient.h"
```

También declararemos una constante que almacene el color de fondo que queremos para nuestra aplicación:

Listing 10: Declarar color de fondo.

```
//---------------------------------------------------------------------------
// Background...
//---------------------------------------------------------------------------

#define BACKDROP_COLOR RGB8(190, 255, 255)
```

Además será necesario modificar el método setupGraphics para cargar los *tiles* necesarios y establecerlos como fondo del juego. En primer lugar definiremos cuatro *tiles* en el método setupGraphics: uno para la imagen de la pelota, otro para el color de fondo (transparente), otro para los ladrillos de la base y otro para el degradado de la parte superior de la pantalla.

Listing 11: Definir *tiles*.

```
/**
 * Sets up graphics.
 */
void setupGraphics(void) {
    FMAW::Tile ball_tile(FMAW::TileAttributes(
                            gfx_ballTiles,
                            gfx_ballTilesLen,
                            gfx_ballPal,
                            gfx_ballPalLen,
                            FMAW::TypeSprite,
                            FMAW::ScreenMain));

    FMAW::Tile bg_tile(FMAW::TileAttributes(
                            gfx_bgTiles,
                            gfx_bgTilesLen,
                            gfx_bgPal,
                            gfx_bgPalLen,
                            FMAW::TypeBackground,
                            FMAW::ScreenMain));

    FMAW::Tile brick_tile(FMAW::TileAttributes(
                              gfx_brickTiles,
                              gfx_brickTilesLen,
                              gfx_brickPal,
                              gfx_brickPalLen,
                              FMAW::TypeBackground,
                              FMAW::ScreenMain));

    FMAW::Tile gradient_tile(FMAW::TileAttributes(
                                 gfx_gradientTiles,
                                 gfx_gradientTilesLen,
                                 gfx_gradientPal,
                                 gfx_gradientPalLen,
                                 FMAW::TypeBackground,
                                 FMAW::ScreenMain));
```

En segundo lugar estableceremos el color de fondo y haremos que aparezcan los *tiles* del muro de ladrillos. Queremos que haya seis filas de ladrillos. Los ladrillos queremos que estén volteados en el eje vertical de tal manera que parezcan el doble de largos y además queremos que dos filas adyacentes no tengan el volteado idéntico para que el muro quede más realista. Por último queremos que sean las últimas filas las que tienen ladrillos y no las superiores. Esto lo haremos por partes, siempre en la función `setupGraphics`

1. Definiremos el color de fondo e instanciaremos el fondo `0` y limpiaremos su región de memoria (no podemos asegurar que esté limpia al cargarse nuestra aplicación).

Listing 12: Definir color de fondo y preparar fondo 0.

```
    FMAW::setBackgroundColor(BACKDROP_COLOR);

    FMAW::Background bgBricks(0);
    bgBricks.setScreenBaseBlock(1);
    bgBricks.clearAllTiles();
```

2. Cargaremos ladrillos en las 6 primeras filas, teniendo en cuenta que queremos voltearlas siguiendo un patrón determinado: original, volteado, original en las filas pares y volteado, original, volteado en las impares:

Listing 13: Mostrar ladrillos.

```
// Set tilemap entries for 6 first rows of background 0 (bricks).
for (int y = 0; y < 6; y++) {
    for (int x = 0; x < 32; x++) {
        int tile_id = x + y * 32;  // Product optimized at compile time!

        // Either odd columns of even rows or even columns of odd rows...
        // if (( x % 2 == 0 && y % 2 == 1 ) || (x % 2 == 1 && y % 2 == 0 ))
        if ((x & 1) ^ (y & 1))
            bgBricks.enableHorizontalFlip(tile_id);

        bgBricks.setTile(tile_id, brick_tile);
    }
}
```

5. Desplazaremos el fondo 0 hacia abajo de manera que el muro quede en la parte inferior.

```
// Did we say 6 first rows? We wanted 6 LAST rows!
bgBricks.setVerticalOffset(112);
```

Si ejecutamos el programa ahora veremos que hay un fondo azul. Esto se debe a que los fondos por defecto utilizan el *tile* `0` como fondo. Nosotros hemos cargado un *tile* transparente como *tile* `0`, de manera que el color de fondo pasa a ser el color de *backdrop*.

A continuación añadiremos el degradado de la parte superior de la pantalla. Para ello utilizaremos el fondo `1`. Debemos tener en cuenta que el degradado es una imagen que excede el tamaño de un *tile* (8×8px) de manera que será necesario jugar un poco con las direcciones de memoria. El degradado es una imagen más alta que ancha de manera que en función de la fila que estemos dibujando querremos dibujar una parte de la tesela más o menos desplazada en el eje vertical. Esto podemos lograrlo añadiendo un desplazamiento a la dirección de memoria del *tile*.

Esta solución puede parecer poco portable pero realmente no supone ningún impedimento a la hora de trabajar en diferentes plataformas: podemos imponer la limitación de tamaño de las teselas que tiene el *hardware* de la Nintendo DS en la implementación para cualquier plataforma, de manera que `FMAW framework` emulase el comportamiento de la Nintendo DS en otras plataformas.

Listing 14: Cargar degradado.

```
FMAW::Background bgGradient(1);
bgGradient.setScreenBaseBlock(2);
bgGradient.clearAllTiles();
// Set tilemap entries for 8 first rows of background 1 (gradient).
for (int y = 0; y < 8; y++) {
    int tile_index = gradient_tile.imgMemory + y;
    for (int x = 0; x < 32; x++) {
        int tile_id = x + y * 32;
        bgGradient.setTile(tile_id, tile_index);
        bgGradient.setPalette(tile_id, gradient_tile.palMemory);
    }
}
```

Por último es necesario indicar el tipo de mezcla de colores que queremos. Para ello recurrimos a las abstracciones de FMAW framework. Establecemos el color de fondo como destino de la mezcla, el degradado como origen de la mezcla, el tipo de mezcla como mezcla de opacidad y finalmente los dos coeficientes usados para el cálculo de la mezcla.

Listing 15: Mezcla de opacidad.

```
        FMAW::Background::useBackdropAsAlphaDst();
        bgGradient.useAsAlphaBlendingSrc();
        FMAW::Background::setAlphaBlendingMode(FMAW::babmAlphaBlending);
        FMAW::Background::setAlphaBlendingCoefficientOne(4);
        FMAW::Background::setAlphaBlendingCoefficientTwo(16);
    }
```

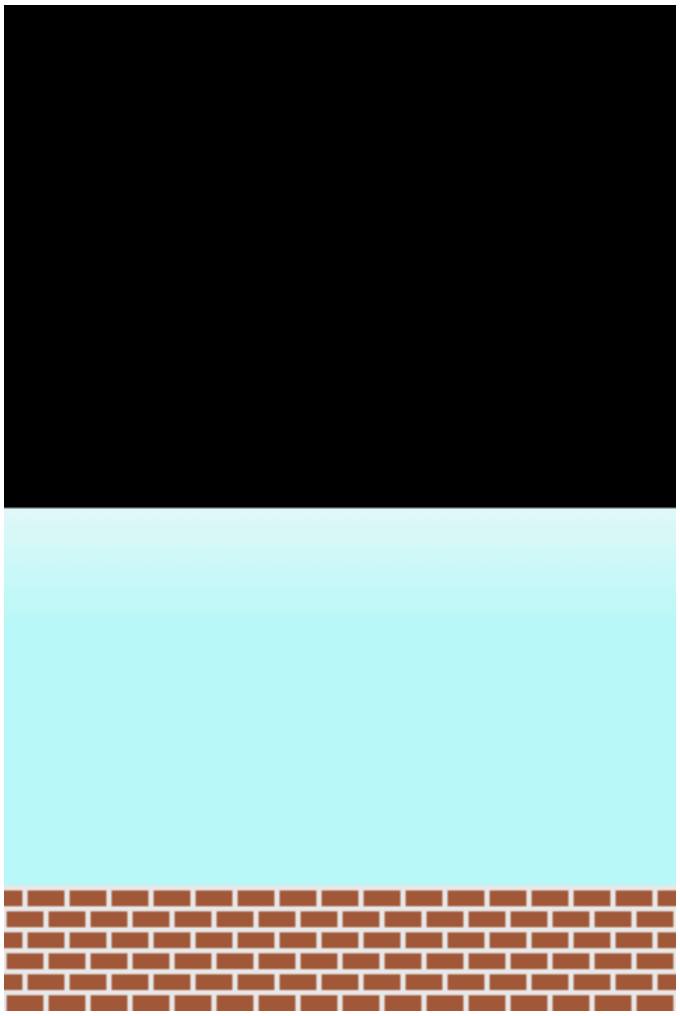


Figura 1: Estado actual del juego.

Si ejecutásemos el juego ahora mismo obtendríamos el resultado de la figura 1.

2.4.3. Inserción de la pelota

Antes de insertar la pelota añadiremos al principio del fichero `main.cpp` una serie de constantes que usaremos más adelante y definen, entre otros parámetros la constante de gravitación universal, la fricción del aire, la fricción del suelo, el nivel de la plataforma, la fuerza del rebote, la velocidad máxima en cada eje, el tamaño de la pelota...

Listing 16: Definir constantes.

```
//------------------------------------------------------------------------------
// Constants
//------------------------------------------------------------------------------

// Gravity constant (add to vertical velocity).
#define GRAVITY         FMAW::FixedReal(0.3125, 8)

// Friction in the air...
#define AIR_FRICTION    FMAW::FixedReal(0.3, 12)
// Friction when the ball hits ground.
#define GROUND_FRICTION FMAW::FixedReal(0.12, 8)
// The level of the brick platform.
#define PLATFORM_LEVEL  FMAW::FixedReal(144, 8)
// The amount of Y velocity that is absorbed when you hit the ground
#define BOUNCE_DAMPER   FMAW::FixedReal(0.08, 8)

#define BALL_RADIUS     FMAW::FixedReal(8, 8)
#define BALL_DIAM       16

#define MIN_HEIGHT      FMAW::FixedReal(4.7, 8)

#define MIN_YVEL        FMAW::FixedReal(4.7, 8)
#define MAX_XVEL        FMAW::FixedReal(3.9, 12)

#define X_TWEAK         FMAW::FixedReal(0.125, 12)
#define Y_TWEAK         FMAW::FixedReal(0.1, 8)
```

A continuación crearemos una clase `Ball`, que hereda de `Character` y representa una pelota. Tendrá como atributos una velocidad en el eje horizontal y otra en el vertical y únicamente requiere implementar dos métodos virtuales de la clase `Character`: `update` y `render`.

Listing 17: Definición de la clase `Ball`.

```
//------------------------------------------------------------------------------
// Ball class
//------------------------------------------------------------------------------

class Ball : public FMAW::Character {
  public:
    /// X, Y velocity.
    FMAW::FixedReal xvel, yvel;
    /// Creates a new ball using a new sprite.
    Ball(): FMAW::Character(FMAW::Sprite())  {};
    /// Updates this ball's position based on its velocity.
    virtual void update();
    /// Renders this ball in given frame.
    virtual void render(int camera_x, int camera_y);
};
```

El método **update** se encargará de actualizar la posición de la pelota en función de su velocidad, además impedirá que la velocidad exceda ciertos límites y que la pelota atraviese el suelo.

Listing 18: Método update de la clase Ball.

```
void Ball::update() {
    this->xvel *= FMAW::FixedReal(1, 12) - AIR_FRICTION;
    if (this->xvel < FMAW::FixedReal(0, 12) - MAX_XVEL) {
        this->xvel = FMAW::FixedReal(0, 12) - MAX_XVEL;
    } else if (this->xvel > MAX_XVEL) {
        this->xvel = MAX_XVEL;
    }
    this->x += FMAW::FixedReal(this->xvel.toDouble(), 8);

    this->yvel += GRAVITY;
    this->y += this->yvel;

    if (this->y + BALL_RADIUS >= PLATFORM_LEVEL) {
        this->xvel *= (FMAW::FixedReal(1, 8) - GROUND_FRICTION);
        // Check if the ball has been squished to minimum height.
        if (this->y > PLATFORM_LEVEL - MIN_HEIGHT) {
            // Mount Y on platform.
            this->y = PLATFORM_LEVEL - MIN_HEIGHT;
            // Negate Y velocity, also apply the bounce damper.
            this->yvel = FMAW::FixedReal(0, 8) -
                         (this->yvel * (FMAW::FixedReal(1, 8) - BOUNCE_DAMPER));
            if (this->yvel > FMAW::FixedReal(0, 8) - MIN_YVEL)
                this->yvel = FMAW::FixedReal(0, 8) - MIN_YVEL;
        }
        this->height = ((PLATFORM_LEVEL - this->y) * FMAW::FixedReal(2, 8)).toInt();
    } else {
        this->height = BALL_DIAM;
    }
}
```

El método **render** se encargará de animar la pelota al rebotar contra el suelo, aplicando un efecto de escalado, además ocultará la pelota en caso de que quede fuera de la pantalla (aunque esto no sucederá nunca ya que la pantalla acompañará a la pelota).

Listing 19: Método render de la clase Ball.

```
void Ball::render(int camera_x, int camera_y) {
    int x, y;
    x = (this->getXPosition() - BALL_RADIUS * 2).toInt() - camera_x;
    y = (this->getYPosition() - BALL_RADIUS * 2).toInt() - camera_y;

    if (x <= -16 || y <= -16 || x >= 256 || y >= 192) {
        this->sprite.disable();
        return;
    }

    this->sprite.setPosition(x, y);
    this->sprite.setSizeMode(FMAW::square16x16);

    this->sprite.enableDoubleSize();

    FMAW::Transform t = FMAW::Transform(0);
```

```
        t.setIdentity();
        FMAW::FixedReal div = FMAW::FixedReal(1, 8) / FMAW::FixedReal(BALL_DIAM, 8);
        FMAW::FixedReal pa = FMAW::FixedReal(this->height, 8) * div;
        FMAW::FixedReal pd = FMAW::FixedReal(1, 8) / pa;
        t.applyScaling(pa, pd);

        this->sprite.applyTransform(t);
    }
```

Añadiremos nuevas variables globales para almacenar la posición de la cámara y la propia pelota:

Listing 20: Variable globales.

```
//------------------------------------------------------------------------------
// Game objects
//------------------------------------------------------------------------------

FMAW::FixedReal g_camera_x;
FMAW::FixedReal g_camera_y;

Ball g_ball;
```

Actualizaremos la función `update_logic` para que actualice el estado de la pelota. Podríamos usar una tarea encolada pero debido a que queremos que sea ejecutada una vez por fotograma es mejor llamarla en esta función y ahorrarnos la sobrecarga de tener una tarea encolada cuya ejecución es continua y se ejecuta cada 5ms.

Listing 21: Función `update_logic`.

```
/**
 * Updates game logic.
 */
void update_logic() {
    FMAW::Timer::check();
    g_ball.update();
}
```

Implementaremos por primera vez la función `update_graphics`, que dibujará la pelota en pantalla y desplazará la cámara a la posición deseada:

Listing 22: Función `update_graphics`.

```
/**
 * Renders graphics.
 */
void update_graphics() {
    g_ball.render(g_camera_x.toInt(), g_camera_y.toInt());

    FMAW::Camera::setHorizontalOffset(g_camera_x);
}
```

Por último actualizaremos la función `main` para que encole una función que determine la posición deseada de la cámara en función de la posición de la pelota:

Listing 23: Función main.

```
/**
 * Main.
 */
int main(void) {
    FMAW::init(update_graphics, update_logic);
    setupGraphics();

    auto update_camera = [](int ID) {
        // Desired camera X:
        FMAW::FixedReal cx = g_ball.getXPosition() - FMAW::FixedReal(128, 8);

        // Difference between desired and current position.
        FMAW::FixedReal dx = cx - g_camera_x;

        if ((dx > FMAW::FixedReal(0.04, 8)) || (dx < FMAW::FixedReal(-0.04, 8))) {
            dx *= FMAW::FixedReal(0.005, 8);
        }

        g_camera_x += dx;
        g_camera_y = 0;
    };
    FMAW::Timer::enqueue_function(update_camera, 5, true);
```

2.4.4. Movimiento de la pelota

Si ejecutásemos ahora el juego veríamos una pelota verde caer del cielo y rebotar contra el suelo repetidamente. No hay manera de desplazarse sobre el eje horizontal. Corregiremos esto añadiendo manejadores para las flechas de dirección de tal manera que al mantenerse pulsada una flecha la pelota se mueva en la dirección correspondiente. Añadiremos a la función `main`:

Listing 24: Manejadores de las flechas de dirección.

```
    auto moveUp = []() { g_ball.yvel -= Y_TWEAK; };
    FMAW::Input::whileButtonArrowUpPressed(moveUp);

    auto moveDown = []() { g_ball.yvel += Y_TWEAK; };
    FMAW::Input::whileButtonArrowDownPressed(moveDown);

    auto moveLeft = []() { g_ball.xvel -= X_TWEAK; };
    FMAW::Input::whileButtonArrowLeftPressed(moveLeft);

    auto moveRight = []() { g_ball.xvel += X_TWEAK; };
    FMAW::Input::whileButtonArrowRightPressed(moveRight);

    FMAW::start();

    return 0;
}
```

El resultado final será muy similar al del tutorial de «How to Make a Bouncing Ball Game» [4] pero con un código mucho más legible. Puede descargarse el resultado desde `https://github.com/Sumolari/TacticsDS/archive/tutorial/results.zip`.

3. Tactics DS

`Tactics DS` es el mejor exponente de las posibilidades que ofrece `FMAW framework` y nuestro objetivo a desarrollar desde el inicio de la asignatura. Su desarrollo ha ido de la mano del de `FMAW framework`, ampliando la librería según necesitábamos un API para acceder a los elementos de más bajo nivel todavía no disponibles[15].

3.1. Propuesta de desarrollo

Al inicio del proyecto redactamos una lista de características que podríamos añadir a `Tactics DS` de tal manera que siempre tuviésemos ideas y en ningún momento nos atascásemos sin saber qué añadir a continuación. Esta lista no pretendía ser una colección de objetivos a corto plazo por tanto era necesario seleccionar un subconjunto de elementos de la lista para componer los objetivos a corto plazo del proyecto, a los que dedicaríamos todo el tiempo de desarrollo disponible. Sólo tras completar totalmente estos objetivos nos plantearíamos el desarrollo de otros objetivos adicionales. La lista de ideas y características quedó como sigue:

Sprint 1: Desarrollo de tablero de ajedrez y pieza peón (movimiento en vertical, mata a una ficha adyacente en vertical).

Sprint 2: Desarrollo de tipos de terreno. Cada casilla es de un terreno diferente y cada ficha tiene una velocidad diferente en cada tipo de terreno de modo que dependiendo del terreno la ficha avanza más o menos casillas. El peón se puede mover en cualquier dirección. Pierde el jugador que se quede sin fichas.

Sprint 3: Desarrollo de unidad a distancia. Se añade un nuevo tipo de ficha, "arquero", que en lugar de matar una ficha que se encuentre adyacente a ella, mata a una ficha que está a una distancia Manhattan[16] concreta, por ejemplo, 3 casillas.

Sprint 4: Añadir salud a las fichas. En lugar de matar, las fichas atacan. Cada ataque resta un poco de salud a la ficha enemiga en función al atributo ataque de la ficha atacante (por simplicidad).

Sprint 5: Ficha caballero. Daña *a melee*, tiene atributos ligeramente diferentes al peón.

Sprint 6: Guardado de partida. Cada movimiento se guarda.

Sprint 7: Reproducir partida. A partir de una partida guardada se reproduce la partida como si de un vídeo se tratase.

Sprint 8: Multijugador local. Cuando termina el turno de un jugador la consola se puede pasar a otro jugador humano para que juegue el turno del rival, en lugar de la IA.

Sprint 9: Mejorar un poco la IA, que hasta ahora jugaba de manera completamente aleatoria.

Sprint 10: Multijugador en red. Cuando termina el turno de un jugador el otro puede jugar, pero vía red.

Sprint 11: Compartir partidas en red. Se pueden subir las partidas guardadas a un servidor para poder incluso reproducirlas online.

Sprint 12: Añadir más variedad de personajes.

Sprint 13: Selección de mapas precargados. Existe una pequeña variedad de mapas entre los que se puede seleccionar el escenario en el que se quiere jugar.

Sprint 14: Mapas procedurales. Hasta ahora el mapa era siempre fijo, permitamos uno aleatorio, quizá con algunos ajustes como desactivar agua.

[15] Hemos llevado a cabo un pequeño blog documentando los avances del proyecto que puede consultarse en `http://sumolari.github.io/TacticsDS/`.

[16] Suma de diferencia de coordenadas en el eje horizontal y vertical.

Sprint 15: Creador de mapas. Eliges el tipo de terreno de cada celda y montas tu propio el mapa.

Sprint 16: Selección de unidades. Puedes elegir las unidades con las que jugar antes de comenzar la partida. Dispones de una cantidad limitada de dinero para elegir. Cada unidad cuesta una cantidad. No se podía ni se pueden crear unidades durante la partida.

Sprint 17: Más de dos jugadores. Ahora puede haber hasta 3 oponentes, todos contra todos.

Sprint 18: Hasta 4 equipos. Los jugadores de un mismo equipo no se pueden atacar.

Sprint 19: Nivel de personajes. Cada personaje al matar a otro consigue experiencia. Al obtener cierta cantidad de experiencia el personaje sube de nivel y se hace más fuerte.

Sprint 20: Habilidades no ofensivas. Se introduce una nueva unidad, "mago", que en lugar de hacer daño a los enemigos es capaz de curar a los aliados que estén a cierta distancia Manhattan.

Sprint 21: Lugares de interés. Ciertas casillas del tablero son lugares de interés y capturarlas da un bonus al jugador que la capturó. Pueden ser capturadas dejando una unidad encima al final de tu turno.

Sprint 22: Niebla de guerra.

Sprint 23: Scroll. El tablero es más grande que la pantalla y para poder verlo entero hay que hacer scroll.

Sprint 24: Unidad *scout*, más débil pero con más visibilidad.

El subconjunto de elementos seleccionados de esta lista que conformaron los objetivos de `Tactics DS` son los que siguen:

Sprint 1: Desarrollo de tablero de ajedrez y pieza peón (movimiento en vertical, mata a una ficha adyacente en vertical).

Sprint 3: Desarrollo de unidad a distancia. Se añade un nuevo tipo de ficha, "arquero", que en lugar de matar una ficha que se encuentre adyacente a ella, mata a una ficha que está a una distancia Manhattan[17] concreta, por ejemplo, 3 casillas.

Sprint 4: Añadir salud a las fichas. En lugar de matar, las fichas atacan. Cada ataque resta un poco de salud a la ficha enemiga en función al atributo ataque de la ficha atacante (por simplicidad).

Sprint 5: Ficha caballero. Daña *a melee*, tiene atributos ligeramente diferentes al peón.

Sprint 6: Guardado de partida. Cada movimiento se guarda.

Sprint 7: Reproducir partida. A partir de una partida guardada se reproduce la partida como si de un vídeo se tratase.

Sprint 8: Multijugador local. Cuando termina el turno de un jugador la consola se puede pasar a otro jugador humano para que juegue el turno del rival, en lugar de la IA.

Sprint 9: Multijugador en red. Cuando termina el turno de un jugador el otro puede jugar, pero vía red.

Sprint 10: Compartir partidas en red. Se pueden subir las partidas guardadas a un servidor para poder incluso reproducirlas online.

No todos los objetivos iniciales han podido ser implementados. Los objetivos que dependen de la conectividad inalámbrica de la consola han sido especialmente conflictivos y finalmente los hemos descartado, siendo reemplazados por otras características cuya implementación es más estable y no depende en tanta medida de la calidad del emulador sobre el que se esté ejecutando la aplicación[18]. Finalmente la lista de características implementadas ha quedado como sigue:

[17] Suma de diferencia de coordenadas en el eje horizontal y vertical.

[18] El único emulador en nuestro conocimiento que dio soporte a la interconexión de consolas a través de redes inalámbricas fue `DeSmuME` pero se vio forzado a retirar esta característica en 2010 debido a diversas acciones tomadas por parte de Nintendo.

Sprint 1: Desarrollo de tablero de ajedrez y pieza peón (movimiento en vertical, mata a una ficha adyacente en vertical).

Sprint 2: Desarrollo de tipos de terreno. Cada casilla es de un terreno diferente y cada ficha tiene una velocidad diferente en cada tipo de terreno de modo que dependiendo del terreno la ficha avanza más o menos casillas. El peón se puede mover en cualquier dirección. Pierde el jugador que se quede sin fichas.

Sprint 3: Desarrollo de unidad a distancia. Se añade un nuevo tipo de ficha, "arquero", que en lugar de matar una ficha que se encuentre adyacente a ella, mata a una ficha que está a una distancia Manhattan[19] concreta, por ejemplo, 3 casillas.

Sprint 4: Añadir salud a las fichas. En lugar de matar, las fichas atacan. Cada ataque resta un poco de salud a la ficha enemiga en función al atributo ataque de la ficha atacante (por simplicidad).

Sprint 5: Ficha caballero. Daña *a melee*, tiene atributos ligeramente diferentes al peón.

Sprint 6: Guardado de partida. Cada movimiento se guarda.

Sprint 7: Reproducir partida. A partir de una partida guardada se reproduce la partida como si de un vídeo se tratase.

Sprint 8: Multijugador local. Cuando termina el turno de un jugador la consola se puede pasar a otro jugador humano para que juegue el turno del rival, en lugar de la IA.

Sprint 9: Mejorar un poco la IA, que hasta ahora jugaba de manera completamente aleatoria.

Sprint 13: Selección de mapas precargados. Existe una pequeña variedad de mapas entre los que se puede seleccionar el escenario en el que se quiere jugar.

Sprint 22: Niebla de guerra.

3.2. Mecánicas del juego

Las mecánicas del juego son similares a las de la franquicia Fire Emblem de Nintendo: se dispone de un mapa dividido en celdas, donde cada celda puede tener un tipo de terreno que afecta a la unidad situada en ella o que la atraviesa. Dos equipos, aliados y enemigos, se enfrentan en combate por turnos en estos escenarios. Cada equipo tiene a su disposición varias unidades, a saber: arqueros, espadachines o caballeros. Cada turno las unidades pueden moverse y atacar, o únicamente atacar. De efectuarse un segundo movimiento éste será más corto que el original. El primer jugador que elimine a todas las unidades de su oponente gana la partida.

3.2.1. Tipos de terreno

Hay siete tipos de terreno visualmente diferenciables:

1. Hierba: no posee ninguna característica especial.
2. Hierba alta: no posee ninguna característica especial.
3. Puente: no posee ninguna característica especial.
4. Río: no puede ser atravesado por las unidades, debe rodearse o buscar un puente por el cual cruzar.
5. Bosque: reduce mucho la visión a través de él y otorga 1 punto de defensa adicional a las unidades situadas en él.
6. Montaña: reduce la visión a través de ella. Aumenta notablemente el rango de visión de las unidades que estén en ella, además de otorgarles 2 puntos de defensa adicional.
7. Castillo: otorga 2 puntos de defensa adicional a la unidad que se halla en él.

[19] Suma de diferencia de coordenadas en el eje horizontal y vertical.

3.2.2. Tipos de unidad

Existen tres tipos de unidad bien diferentes, cada una con su propio conjunto de *sprites* y sus mecánicas propias. En la tabla 1 se muestran las estadísticas de todas las unidades disponibles en el juego.

1. Espadachín: unidad cuerpo a cuerpo con poco daño pero mucha resistencia.
   Habilidad especial: Berserker (tiene una acción de mover o atacar adicional por turno).
2. Arquero: unidad a distancia con buen daño y rango, pero baja resistencia y movimiento.
   Habilidad especial: Puntería (puede atacar a distancia a los enemigos).
3. Caballero: unidad cuerpo a cuerpo con daño y resistencia moderadas, pero con un gran rango de movimiento.
   Habilidad especial: Carga (puede desplazarse a más distancia que cualquier otra unidad).

	Ataque	**Rango ataque**	**Vida**	**Movimiento**	**Visión**
Espadachín	3	1	10	6	5
Arquero	5	2-4	4	4	6
Caballero	4	1	5	7	6

Cuadro 1: Estadísticas de las unidades

3.3. Diseño de la interfaz

3.3.1. Tablero

Nuestro juego está fuertemente inspirado en *Advance Wars*, *Fire Emblem* y *Final Fantasy Tactics*, de modo que diseñar la interfaz no era un gran problema. En primer lugar recopilamos algunos *sprites* de Fire Emblem[20] y compusimos una rejilla de 16 casillas en horizontal y 12 en vertical.

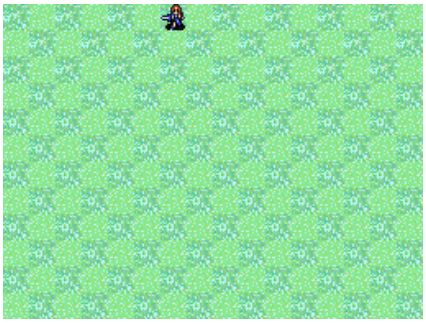


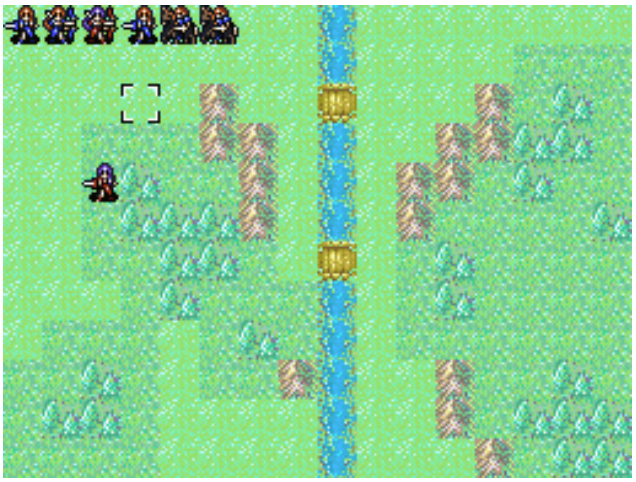
Figura 2: Tablero con dos colores.

La división de la rejilla no es arbitraria: los *tiles* son de 8×8px, de manera que era necesario hacer una división en tamaños múltiplos de 8. Las casillas de 8 × 8px son demasiado pequeñas y el fondo es apenas distinguible. Era importante, además, mantener las proporciones cuadradas de las teselas para hacer más sencillo diseñar los mapas. Hicimos pruebas con casillas de 16 × 16px y de 32 × 32px y la rejilla que nos resultó más apropiada fue la de 16 × 16px, de manera que cada casilla del tablero utiliza cuatro *tiles* para dibujar su fondo.

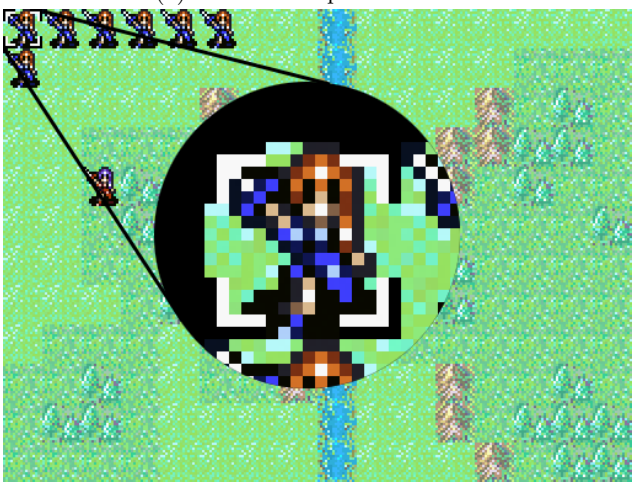
En la figura 2 se puede ver el aspecto que tiene el tablero cuando sólo se utilizan dos tipos de terreno diferentes. Éste es el aspecto casi final del tablero, al que se le añadieron más tipos de terreno (hasta un total de siete tipos de terreno diferentes, como ríos, puentes, montañas, bosques o castillos, entre otros), más unidades, un cursor que indica el tipo de acción a realizar y la posición actualmente seleccionada y finalmente se le introdujo niebla de guerra. Puede verse el tablero en diferentes etapas del desarrollo de `Tactics DS` en la figura 3.

[20]Disponibles en `http://www.feplanet.net/sprites-archive`, y tomados para este proyecto en base al derecho de *fair use* de la ley de *copyright* de Estados Unidos de América.

(a) Diferentes tipos de terreno.



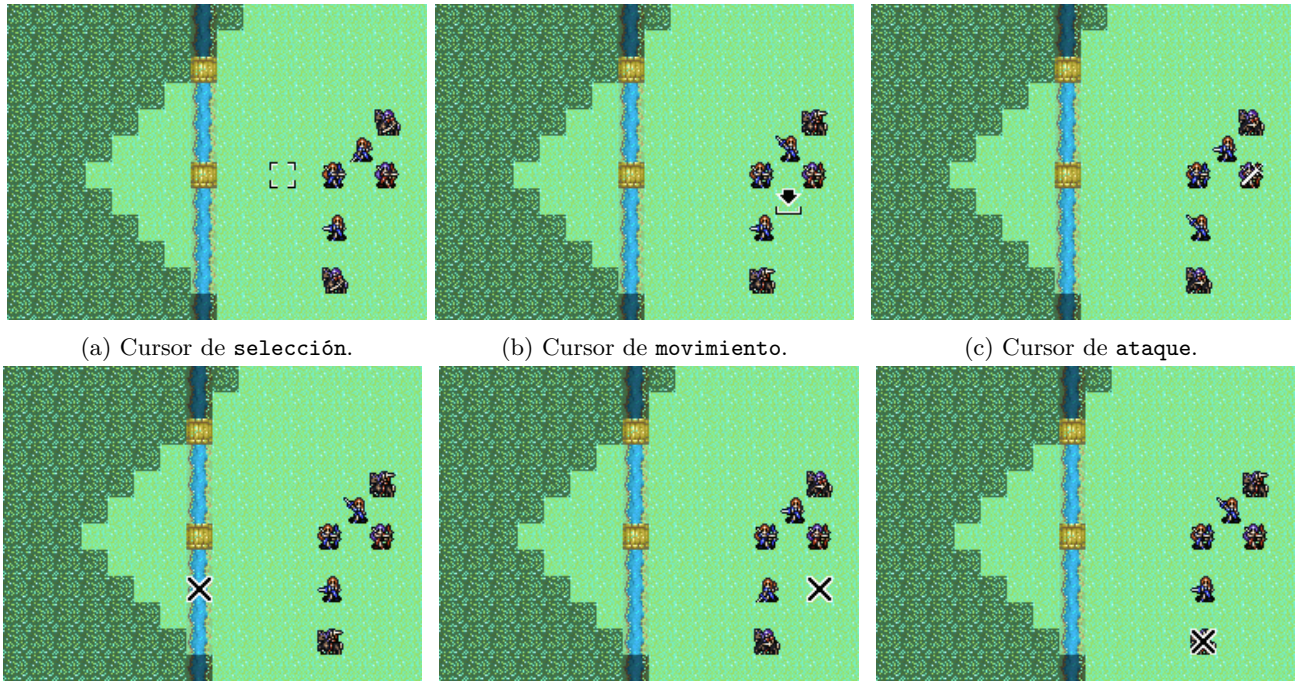
(b) Primer mapa completo.



(c) Cursor en el tablero.

(d) Niebla de guerra.

Figura 3: Avances en el diseño del tablero.



(a) Cursor de `selección`. (b) Cursor de `movimiento`. (c) Cursor de `ataque`.

(d) Cursor de `bloqueo`. (e) Cursor de `bloqueo`. (f) Cursor de `bloqueo`.

Figura 4: Diferentes representaciones del cursor.

3.3.2. Cursor

El cursor es una de las piezas fundamentales del tablero. Permite al jugador saber con qué celda está interactuando y qué tipo de acción se puede realizar. El cursor se desplaza por el interior del tablero, ocupando siempre una casilla, y se puede mostrar en 4 estados diferentes: `selección`, `desplazamiento`, `ataque` y `bloqueo`. Además al desplazarse el cursor se reproduce un sonido, que varía en función de si se puede interactuar con la casilla a la que se desplaza o no. En la figura 4 pueden verse las diferentes representaciones del cursor en función del tipo de interacción con la celda seleccionada.

El cursor de `selección` (figura 4a) es el cursor por defecto e indica que la celda sobre la que se encuentra puede ser `marcada`. Al `marcarse` una celda la unidad que se encuentra en ella pasa a ser la `unidad seleccionada` de manera que el cursor cambia al cursor de `movimiento` (figura 4b), `ataque` (figura 4c) o `bloqueo` (figuras 4d, 4e y 4f):

- Se mostrará el cursor de `movimiento` si la celda apuntada por el cursor está dentro del rango de movimiento de la `unidad seleccionada`.
- Se mostrará el cursor de `ataque` si la celda apuntada por el cursor está dentro del rango de ataque de la `unidad seleccionada` y en ésta hay una unidad enemiga.
- Se mostrará el cursor de `bloqueo` si la celda apuntada por el cursor está fuera del rango de movimiento o de ataque de la unidad.
- Se mostrará el cursor de `selección` si la celda apuntada por el cursor está ocupada por otra unidad controlada por el jugador, de manera que pueda marcarla y liberar la `unidad seleccionada` previa.

(a) Menú con la opción de `Nueva partida` seleccionada.

(b) Menú con la opción de `Cargar partida` seleccionada.

(c) Menú con la opción de `Versus` seleccionada.

(d) Pantalla de información para el jugador azul.

(e) Pantalla de información para el jugador rojo.

Figura 5: Menú principal y pantalla de información.

3.3.3. Menú

El menú es un elemento al que generalmente se le presta menos atención que al resto del juego pero que es igualmente importante. En `Tactics DS` el menú principal permite iniciar una partida contra la IA, contra otro jugador humano o reproducir la última partida jugada. Además durante la partida el menú se transforma en un panel de información donde aparecen tanto los controles como el color del jugador que está jugando el turno actual. El menú de `Tactics DS` soporta interacción con el *stylus* de la Nintendo DS, de manera que se puede marcar cualquier opción tocando el botón correspondiente, además de desplazando la selección mediante las flechas verticales.

3.4. Diseño de la lógica

`Tactics DS` utiliza en gran medida clausuras y mecanismos de delegación y *callback* para gestionar la lógica del juego. Las clases `Grid` y `MainMenu` incluyen métodos para encolar y desencolar todos sus manejadores, de manera que alternar entre la partida y el menú es tan sencillo como desencolar los manejadores de uno y encolar los del otro.

Por su parte el sistema de turnos se gestiona mediante `TurnManager`. En primer lugar deben registrarse los jugadores que habrá en la partida (actualmente sólo se permiten partidas de hasta dos jugadores pero el sistema soporta partidas

con un número arbitrario de jugadores) y a continuación llamando al método `finishTurn` se cambia el turno de cada jugador. Internamente lo que sucede es que al registrarse un jugador se mantiene un puntero a la instancia de la clase `Player` registrada, de tal manera que cuando comienza el turno de un jugador se llama al método `startTurn` de éste, que habilita la interacción del jugador con el tablero.

Tras cada ataque de un jugador se comprueba si quedan enemigos en el campo de batalla y de no ser así se reproduce el sonido de victoria y se anuncia el ganador de la partida.

3.5. Diseño de la IA

La inteligencia artificial del videojuego es un aspecto muy importante en este género y si bien podríamos haber dedicado gran parte del desarrollo diseñando una inteligencia artificial infalible hemos preferido desarrollar una inteligencia artificial sencilla para centrarnos en otros aspectos del juego.

Para cada unidad bajo su control, nuestra IA actúa siguiendo los siguientes criterios, ordenados

1. Si hay unidades enemigas a las que sólo puede atacar esta unidad, ataca a una de ellas seleccionada de forma aleatoria.
2. Si hay diversas unidades enemigas a las que pueden esta y otras unidades, ataca a una aleatoria.
3. Si no hay ninguna unidad a la que pueda atacar se mueve a la celda alcanzable más cercana a las unidades enemigas.

Por simplicidad todos los tipos de unidad actúan del mismo modo, de manera que unidades con diferentes características comparten un único patrón de comportamiento, evitando tener que implementar una lógica específica para cada tipo de unidad.

Los agentes están situados en entorno caracterizado por las siguientes cuatro propiedades fundamentales:

1. **No accesible:** las unidades de la IA desconocen toda la información del mapa debido a la implementación de la niebla de guerra.
2. **Determinista:** todas y cada una de las acciones tienen un único resultado posible, tanto el movimiento de una unidad como el ataque que ésta puede realizar.
3. **Dinámico:** dado que el entorno tiene distintos agentes, éste se ve influenciado por distintos procesos. Debido a esto un agente no sólo se verá afectado por sus decisiones, sino que también lo hará con las de otros.
4. **Discreto:** aunque existan muchas posibles acciones que un agente puede llegar a realizar, el número de estas sigue siendo finito.

3.6. Metodología de trabajo

Para asegurarnos un desarrollo agradable establecimos un sistema bien definido de tareas y control de código. La gestión de tareas se llevó a cabo mediante la plataforma Phabricator[21] de Facebook y el control de versiones mediante Git, alojando los repositorios en Bitbucket[22] y en GitHub[23].

Cada tarea recibía un identificador numérico y se trabaja en ella en una rama independiente creada a partir de la rama `development`. Una vez terminada la tarea se mezclaba con la rama `development` y se continuaba el desarrollo tomando una nueva tarea. Las tareas contenían pequeñas modificaciones del programa, como añadir una nueva unidad, implementar un nuevo método o corregir algún error. Cuando una tarea estaba terminada antes de mezclar su rama con `development` se creaba una *pull request*, que únicamente tras ser aprobada por todos los miembros del equipo era integrada en la rama de desarrollo.

Además disponíamos[24] de un servidor de integración continua en MagnumCI[25] que verificaba que cada versión del proyecto se podía compilar. Cada prueba que se realizaba utilizaba una máquina virtual nueva, de manera que tuvimos que automatizar los *scripts* de instalación de `devKitPro`.

[21] `http://phabricator.org`
[22] `https://bitbucket.org`
[23] `https://github.com/Sumolari/TacticsDS`
[24] `https://magnum-ci.com/public/4c98e728354f98f9761b/builds`
[25] `https://magnum-ci.com`

4. Dificultades

El desarrollo de `Tactics DS` y de `FMAW framework` ha sido duro no tanto por la dificultad de los objetivos planteados como por la falta de facilidades ofrecidas por la plataforma. Es destacable que `FMAW framework` no nació para dar soporte a `Tactics DS` sino que su desarrollo comenzó al seguir uno de los tutoriales de introducción al desarrollo para Nintendo DS propuesto, «How to Make a Bouncing Ball Game» [4], que además es el documento de introducción más completo al que hemos llegado a tener acceso.

La documentación es escasa y generalmente incompleta o confusa, teniendo que recurrir a revisar el código fuente de las librerías para saber realmente qué hacen ciertas funciones. La comunidad de usuarios suele recomendar malas prácticas conocidas y antipatrones de diseño, como *hard-code*, utilizar desplazamiento de bits en lugar de divisiones sin documentar el motivo o nunca usar enumerados o constantes para mantener el código legible y mantenible; entre otros. Todas estas malas prácticas nos resultan tremendamente desagradables y hemos querido evitarlas en `FMAW framework` y, sobre todo, en `Tactics DS`.

Junto a la mala documentación y los malos consejos de la comunidad debe añadirse el terrible soporte de herramientas de debug, llegando a ver errores tan pintorescos como que el compilador genere código máquina que no pueda interpretar el procesador de la consola, pasando por toda clase de errores desde fugas de memoria cuasi indetectables hasta saturación de la memoria de vídeo y paletas.

Estas dificultades han sido especialmente importantes durante las primeras semanas de desarrollo de `FMAW framework` ya que una vez abstraído el *hardware* de la consola el desarrollo de `Tactics DS` tuvo menos complicaciones de este tipo. Sin embargo sí que hubo algunos problemas recurrentes con `Grit`[26] y con la conversión del audio al formato requerido por `maxmod`[27].

Cabe destacar que a pesar de las limitaciones de *hardware* tanto la velocidad del procesador como la capacidad de la memoria principal fue más que suficiente y no encontramos ningún tipo de limitación debido a estos. No obstante la memoria de vídeo y de paletas sí que supuso un impedimento importante.

4.1. Red

La interfaz WiFi de la Nintendo DS está situada en la propia placa base de la consola y se trata de una implementación totalmente casera por parte de Nintendo. Se asume que la interfaz es de 16 bits y mediante la experimentación se ha concluido que soporta transmisiones de hasta 2.0 Mbit.

DevKitPro proporciona varios ejemplos para el uso del WiFi en la Nintendo DS pero hemos tenido múltiples problemas a la hora de conectar a los puntos de acceso que tenemos a nuestra disposición, posiblemente porque estos ejemplos no dan soporte a la seguridad `WPA`, que es la más común hoy en día pero no lo era en la época de la Nintendo DS[28].

Nos hubiera gustado realizar varias pruebas de red con la Nintendo DS y posiblemente incorporarlo de alguna manera a nuestro proyecto pero debido a la escasa documentación accessible para un usuario con nuestro nivel y el tiempo restante cara a la entrega del proyecto hemos decidido no continuar con la comunicación inalámbrica.

[26] Herramienta utilizada para convertir las imágenes al formado de la Nintendo DS.
[27] Es un formato antiguo y ya prácticamente no hay *software* que permita convertir audio a dicho formato.
[28] Sin embargo da soporte a la seguridad `WEP`, desaconsejada en la actualidad.

5. Conclusiones

El desarrollo de *homebrew* para Nintendo DS es una ardua tarea si se desean resultados de calidad. Las herramientas son escasas, la documentación confusa y la comunidad ofrece dudosos consejos. Por tanto la decisión de desarrollar nuestra propia librería, a pesar del enorme esfuerzo que supuso, ha resultado ser una de las decisiones más acertadas del proyecto y nos ha permitido desarrollar en mucho menos tiempo del esperado un prototipo de juego totalmente jugable.

Centrarnos en el desarrollo de un juego basado en tablero ha sido también un acierto importante, permitiéndonos dedicar la mayor parte del tiempo de desarrollo del juego a perfeccionar la interfaz e implementación del tablero y haciendo sencillo añadir mejoras al juego, como variedad de personajes o nuevas mecánicas (niebla de guerra, ataques a distancia, etc).

5.1. Futuro

Todavía queda mucho trabajo por hacer en `Tactics DS` para que llegue a tener la calidad de un juego comercial de Nintendo DS. Durante el desarrollo hemos topado con algunas limitaciones tanto de `Tactics DS` como el `FMAW framework` que nos habría gustado superar pero que no hemos podido por falta de tiempo. A continuación listamos algunos de los puntos que creemos que deberían ser mejorados en futuras versiones.

1. Limitación del número de paletas y *sprites*. Hemos descubierto que únicamente podemos mantener en memoria de vídeo 8 *sprites* diferentes (limitados en realidad por las paletas). Esto probablemente se deba a que únicamente tenemos activado uno de los bancos de memoria de vídeo disponibles, sin embargo en las pruebas que hemos llevado a cabo no hemos logrado activar y usar correctamente otros bancos. Sería realmente interesante seguir investigando esta opción y mejorar `FMAW framework` para que éste fuese capaz de activar bancos de memoria según fuese necesario.

2. Optimizar el consumo de memoria de las paletas. La Nintendo DS soporta una paleta de hasta 256 colores, de manera que resultaría muy útil que `FMAW framework` compactase las paletas según se registrasen, evitando duplicidad de colores en la paleta. Actualmente la solución que hemos tomado ha sido manualmente preprocesar las imágenes y registrarlas en un orden específico de manera que podamos evitar cargar algunas paletas y se reutilicen las que ya están en memoria. Esta solución evidentemente no es la ideal y mejoras en este aspecto permitirían una abstracción mayor y un uso de memoria mucho más eficiente.

3. Mejoras en sistema de tareas programadas. Actualmente el sistema de tareas programadas tiene cierta sobrecarga de memoria. Si bien esto no supone un problema sería interesante poder reducir el uso de memoria de cara al desarrollo de juegos más complejos. Además es necesario llamar al método `FMAW::Timer::check` antes de dibujar cada fotograma, algo que podría evitarse utilizando los manejadores de interrupción de reloj que ofrece la Nintendo DS. Si bien ninguna de las dos mejoras supondrá un cambio drástico en las posibilidades que brinda `FMAW framework` ambas nos parecen importantes de cara a mantener una base de código limpia y sostenible.

4. Añadir más variedad de unidades. Actualmente hay tres tipos de unidades pero sería interesante disponer de más unidades y de un sistema de selección que permita al jugador disponer del ejército con el que se sienta más cómodo jugando. Son mejoras que harían de `Tactics DS` un juego más entretenido y variado y que no costaría demasiado implementar, aunque ha sido imposible añadir esto en el primer prototipo.

5. Añadir más jugadores y equipos. Aunque ya es jugable con únicamente dos jugadores, permitir jugar a más jugadores y formar equipos cambiaría la forma de jugar a `Tactics DS` y ofrecería un nuevo conjunto de posibilidades muy interesantes para el jugador. Desde el punto de vista de desarrollar un producto comercial este punto es esencial.

6. Añadir juego en red. La Nintendo DS tiene capacidades de red y no aprovecharlas es una lástima. Sin embargo los problemas surgidos durante el desarrollo de las funciones de red de `Tactics DS` han hecho imposible disponer de esta característica a tiempo. Seguir trabajando en este campo aportaría una mejora sustancial al juego.

7. Portar `FMAW framework` a OpenGL. `FMAW framework` está diseñado por completo pensando en que pudiese ser portado a cualquier plataforma y el desarrollo de la versión en OpenGL está comenzado aunque en una fase muy temprana del desarrollo y requiere implementar la mayoría de las clases todavía. No es un trabajo especialmente complicado pero requiere tiempo y conocimientos de OpenGL y aunque contábamos con los segundos no hemos dispuesto del tiempo suficiente para implementar esta versión.

Referencias

[1] Bartley, C. Memtrack: Tracking memory allocations in c++, 2002. http://www.almostinfinite.com/memtrack.html.

[2] (Eltoshen), C., (Ellis), J., and (Zerxer), N. fire emblem planet, 2008. http://www.feplanet.net/sprites-archive.

[3] Feldman, S., and desarrolladores de GNU. Gnu make manual, 2014. https://www.gnu.org/software/make/manual/.

[4] Johnson, M. e. How to make a bouncing ball game, 2009. http://ekid.nintendev.com/bouncy/.

[5] Lavelle, S. Bfxr: make sound effects for your games., 2011. http://www.bfxr.net/.

[6] Miessner, B. e. a. Free ringtones on myfreeringtones.org, 2009. http://myfreeringtones.org/title/Final

[7] Nash, P. e. a. Catch, 2010. https://github.com/philsquared/Catch.

[8] Noland, M. j., Rogers, J. d., and Murphy, D. W. libnds documentation, 2014. http://libnds.devkitpro.org/.

[9] Odgen, S. A. v. Ambient click 2, 2006. http://www.flashkit.com/soundfx/Interfaces/Clicks/ambient_-agent_vi-8701/index.php.

[10] Stroustrup, B. A tour of c++, 2014. http://www.stroustrup.com/Tour.html.