Tactics DS

Víctor Grau Moreso
Mark Holland
Lluís Ulzurrun de Asanza Sàez

12 de abril de 2015

Índice

Introducción

Figura : Fire Emblem

Figura : Advance Wars

Introducción

Problemas

1. Muchos juegos recurren a tableros o rejillas.
2. `libnds` es confuso y requiere mucho esfuerzo por parte del desarrollador.
3. ¡Queremos hacer un juego!

Soluciones

1. Clase `Grid` que nos permita interactuar con comodidad.
2. Desarrollaremos un framework que nos permita trabajar cómodamente.
3. ¡Haremos nuestro propio Fire Emblem!

Índice

Framework

- No queremos pensar en el hardware mientras programamso el juego. En serio, no queremos.
- Tiene que ser muy eficiente (procesadores ARM7 a 33Mhz y ARM9 a 67 Mhz, 4MB de RAM, poca memoria de vídeo...).
- Queremos que programar sea cómodo y que el framework se encargue de gestionar todas las complicaciones.
- No queremos un framework sólo para la Nintendo DS, nos gustaría portarlo en un futuro a otras plataformas.
- **Solución:** Diseñamos las APIs de manera que sean muy cómodas de usar pero también teniendo en cuenta las limitaciones del hardware.

Compilando en múltiples plataformas

- Los `Makefile` de `libnds` son algo misteriosos.
- No podemos compilar de forma recursiva.
- Podemos cambiar las flags del compilador.
- ¿Cómo podemos hacer el framework portable?
- **Solución:** Declaración del framework en cabeceras `.h` bien documentadas. Archivos `.cpp` que importan el archivo con la implementación en función de las flags del compilador.

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

¿A qué nos referimos con cómodo?

```
auto prsB = []() { FMAW::printf("Se pulsa B"); };
int IDprsB = FMAW::Input::onButtonBPressed(prsB);
auto whlB = []() { FMAW::printf("B sigue pulsado"); };
int IDwhlB = FMAW::Input::whileButtonBPressed(whlB);
auto rlsB = []() { FMAW::printf("Se suelta B"); };
int IDrlsB = FMAW::Input::onButtonBReleased(rlsB);

auto tClbk = [IDprsB, IDwhlB, IDrlsB](int x, int y) {
  if (FMAW::pointInArea({x, y},
      MAIN_MENU_NEW_GAME_AREA)) {
    FMAW::printf("Nueva partida!");
    FMAW::Input::unregisterCallback(IDprsB);
    FMAW::Input::unregisterCallback(IDwhlB);
    FMAW::Input::unregisterCallback(IDrlsB);
  }
};
FMAW::Input::onTouchReleased(tClbk);
```

¿A qué nos referimos con cómodo?

Código real de Tactics DS

Este ejemplo es parte de nuestra implementación de la clase `Grid`.

```
bool Grid::moveCharacterFromCellToCell(IndexPath from,
                        IndexPath to, unsigned int dur) {
    Cell *f = this->cellAtIndexPath(from);
    Cell *t = this->cellAtIndexPath(to);
    if (!t->isOccupied() && f->isOccupied()) {
        Unit *c = f->getCharacter();
        t->setCharacter(c);
        c->setPosition(f->getCenter());
        c->animateToPosition(t->getCenter(), dur);
        f->setCharacter(nullptr);
        return true;
    }
    return false;
}
```

Otras APIs

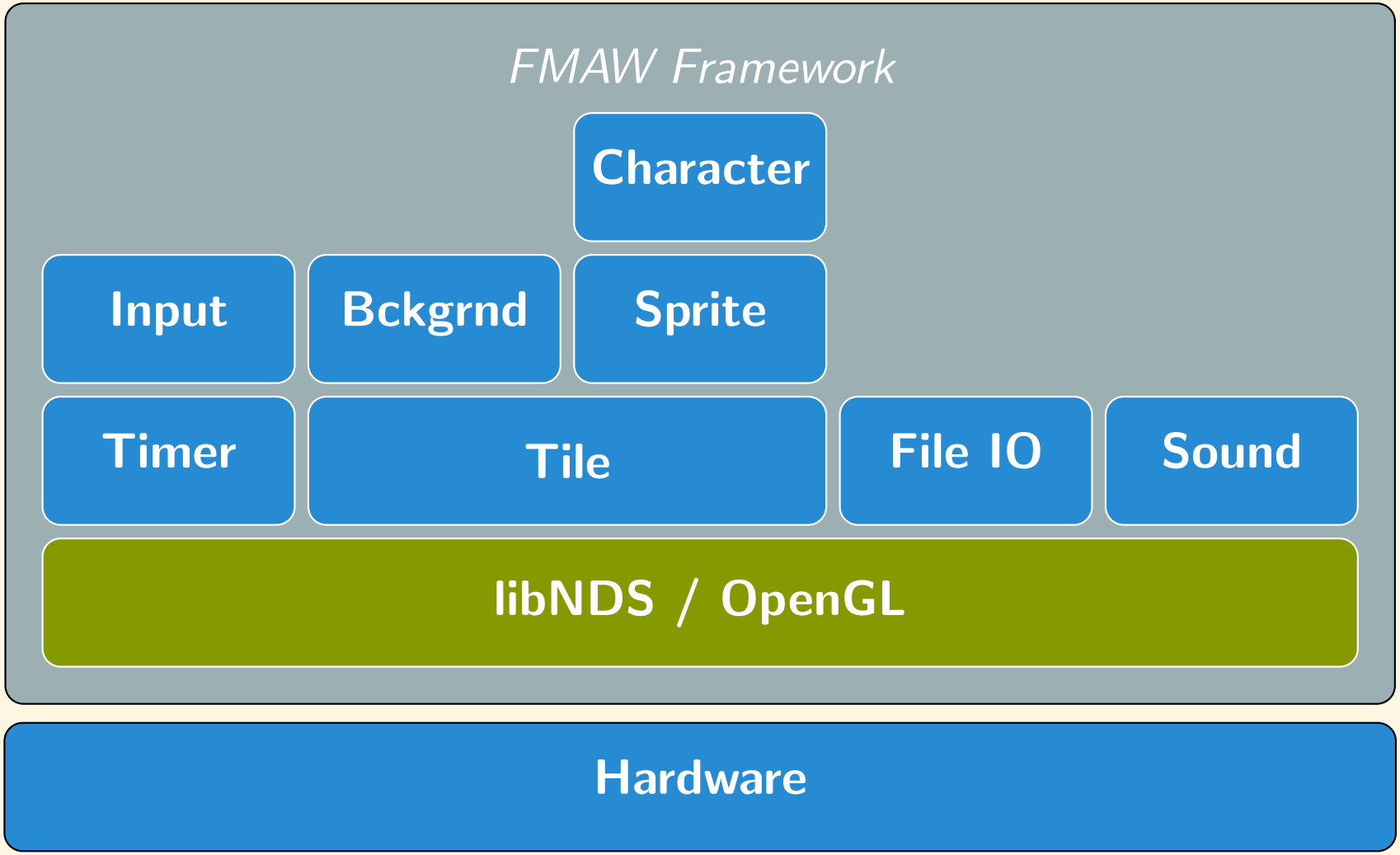
Nuestro framework incluye APIs para, entre otras cosas...

- Operar en coma fija con los operadores tradicionales: `+ / * -`
- Gestionar los fondos, `tile` a `tile` o como una imagen fija.
- Gestionar los `sprites` y paletas, gestionando de forma automática la asignación de memoria de vídeo.
- Gestión de `timers` y programación de tareas para ejecución repetida o futura.
- Banda sonora y efectos de sonido.
- Herramientas para facilitar la depuración.

Además de las ya vistas...

- Abstracción completa del sistema de `input`.
- API de animación.

Arquitectura



¿Cómo funciona la magia? - I

```
/**
 * Moves this character to given position animatedly.
 * @param position Final position of the character.
 * @param duration Time it'll take to move the it.
 * @param type     Type of animation to be performed.
 * @param callback Function that will be called when
 *                 animation finishes. It'll receive
 *                 whether animation finished
 *                 successfully (true) or not.
 */
animation_id animateToPosition(
  Point position,
  unsigned int duration,
  AnimationType type,
  std::function<void(bool)> callback
);
```

¿Cómo funciona la magia? - II

```
animation_id id = this->animations.size();

std::function<void(int)> func = [this, id](int
    callback_id) {
    // MAGIC HAPPENS HERE!
};

int callback_id = Timer::enqueue_function(func,
    ANIMATION_STEP_TIME, true);

Animation animation = {
    id, callback_id,
    this->getPosition(), position,
    duration, duration, type,
    callback
};
this->animations[id] = animation;

return id;
```

¿Cómo funciona la magia? - III

```
this->animations[id].remainingDuration -=
    ANIMATION_STEP_TIME;

if (this->animations[id].remainingDuration <= 0) {
    // Set final position.
    this->setPosition(this->animations[id].
        finalPosition);
    // Dequeue enqueued function.
    Timer::dequeue_function(callback_id);
    // Call completion callback.
    if (this->animations[id].callback != NULL)
        this->animations[id].callback(true);
} else {
    // MORE MAGIC!
}
```

¿Cómo funciona la magia? - IV

```
Point fPos = this->animations[id].finalPosition;
Point iPos = this->animations[id].initialPosition;
// Set position computed on the fly.
int deltaX = fPos.x - iPos.x;
int deltaY = fPos.y - iPos.y;

int deltaDur = this->animations[id].duration - this->
    animations[id].remainingDuration;

FixedReal dTimePercent = FixedReal(deltaDur, 8), dtd =
     FixedReal(this->animations[id].duration, 8);

dTimePercent = dTimePercent / dtd;

this->setXPosition(dTimePercent * deltaX + iPos.x);
this->setYPosition(dTimePercent * deltaY + iPos.y);
```

Índice

La clase `Grid`

- La pantalla de la Nintendo DS es de 256x192 px, suficiente para un tablero de 16x12 casillas de 16x16 px.
- Cada casilla ocupará 4 tiles de `background`.
- Necesitaremos funciones auxiliares para manejar algunas cosas cómodamente, entre otras:
  - Personaje ocupando la celda.
  - Coste de ir de una celda a otra.
  - Resaltar la celda seleccionada.
  - Mover un personaje de una celda a otra
  - Dibujar las celdas.
  - Iniciar el tablero a partir de un archivo de definición de mapa.
  - Guardar y reproducir movimientos.

Acceso a las casillas

- Las casillas se indexan mediante `IndexPath`, una pareja de fila y columna.
- Se mantiene un `IndexPath` especial que apunta a la celda seleccionada en cada momento (la que está resaltada por el cursor).
- Métodos auxiliares permiten desplazar el cursor.
- La propia `Grid` encola y desencola callbacks para los botones de desplazamiento y el *stylus*.

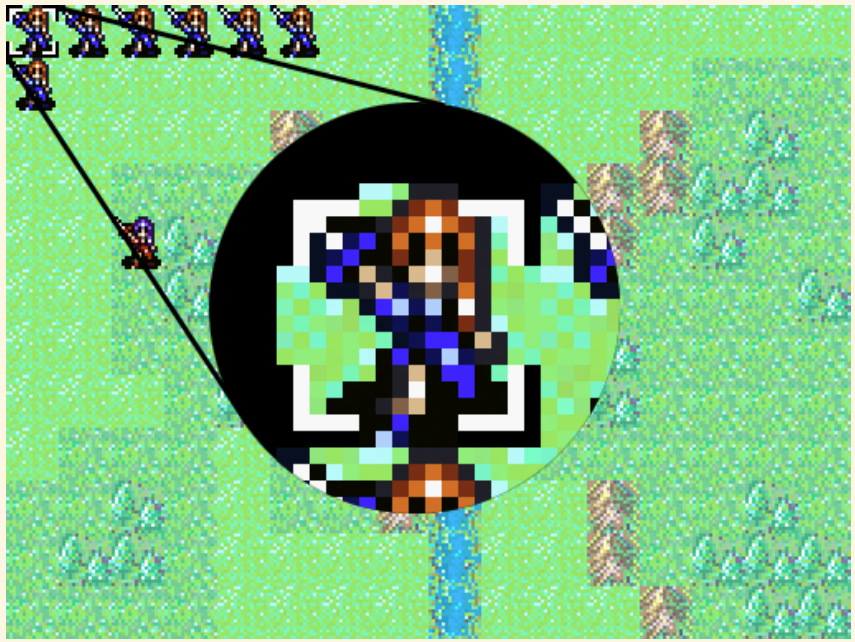


Figura : Cursor en TacticsDS

Acceso a las casillas

```
class Grid {
    Cell *cellAtIndexPath(IndexPath ip);

    Cell *cellAtSelectedPath();
    IndexPath getSelectedPath();

    bool moveCharacterFromCellToCell(IndexPath from,
        IndexPath to, unsigned int duration);

    bool selectCellAtIndexPath(IndexPath path);

    void enqueueCallbacks();
    void dequeueCallbacks();

    // ...
};
```

Casillas

- Cada casilla tiene un tipo de terreno y un personaje asociado (opcional).
- Cada casilla conoce su centro y el coste de desplazarse a ella desde una adyacente.
- Además cada casilla tiene asignadas las 4 `tiles` correspondientes del `background`.
- ¡Esto sigue siendo portable! `tile` = `quad` (OpenGL)
- Tanto casillas como tablero son parte del juego, no del framework, por tanto no tienen conocimiento del hardware sobre el que se ejecutan (ni importan `libnds`).

Otras funciones del tablero

- Cada movimiento de un personaje en el tablero se guarda en un fichero de historial.
- El historial puede ser reproducido en cualquier momento como si de un vídeo se tratase.
- Alcanzar una casilla tiene un coste diferente dependiendo del tipo de terreno (montaña, río, hierba...) y cada personaje tiene una capacidad de movimiento.
- El tablero impide que un personaje se desplace de una casilla a otra si no dispone de suficiente capacidad de movimiento (calcula al vuelo las casillas accesibles).

Índice

Tactics DS

- Modo multijugador contra otro jugador humano.
- Modo individual contra la IA.
- Posibilidad de guardar y reproducir partidas.
- Posibilidad de jugar en diversos escenarios.
- Posibilidad de jugar con niebla de guerra (¡pero no usarla al reproducir partidas!).
- Variedad de personajes diferentes (ataques a distancia o cuerpo a cuerpo, movimiento rápido o lento, más o menos salud...).
- Animaciones de los personajes en reposo.
- Menú y tablero con control táctil (*stylus*).

Tactics DS

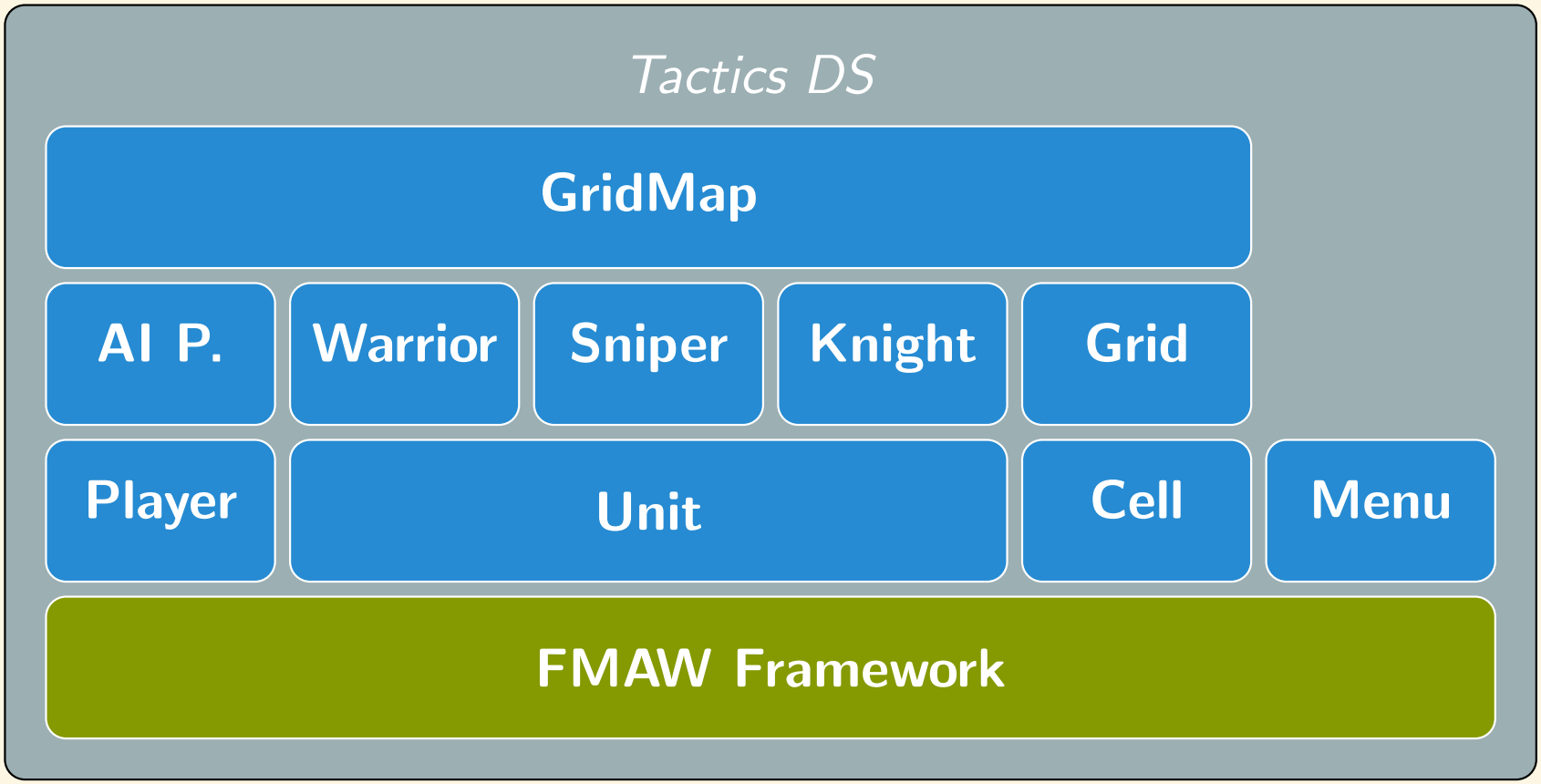
Problemas

1. Posibilidad de guardar y reproducir partidas.
2. Posibilidad de jugar en diversos escenarios con o sin niebla de guerra.
3. Animaciones de los personajes en reposo.
4. Interacción con el usuario.

Soluciones

1. Clase `Grid`.
2. Clase `Grid`.
3. Framework (API de animación).
4. Framework (API de `input`).

Arquitectura



Multijugador contra humano

- Cada jugador está representado por una instancia de la clase `Player`.
- Un coordinador de turnos llama al método `startTurn` de cada `Player` al comienzo de su turno.
- Al pulsar el botón `B` se cambia el jugador.
- En el turno del jugador azul sólo se permite seleccionar personajes de este jugador.
- Un personaje azul se puede mover a una casilla vacía o puede atacar a un enemigo desplazándose a una casilla ocupada por un personaje rojo.
- La niebla de guerra permite ver sólo las unidades del jugador actual.
- El cursor cambia para indicar la acción a realizar.

Multijugador contra IA

- La IA es una instancia de la clase `PlayerAI`.
- El turno de un `PlayerAI` termina cuando éste llama al callback que recibe en el constructor. Este callback hace lo mismo que haría cuando un jugador humano pulsa el botón `B`.
- La IA está sometida a las mismas limitaciones de movimiento que cualquier jugador humano.
- Se aplican las mismas mecánicas que para jugadores humanos.
- Para el resto del juego el funcionamiento es igual haya dos jugadores humanos o dos IA jugando.
- La niebla de guerra permite ver sólo las unidades del jugador humano.

Reproducir partida

- Cada jugador está representado por una instancia de la clase `Player`.
- Se desactivan todos los *callbacks* de interacción hasta terminar de reproducir la partida.
- Se encola una función que ejecutará movimiento a movimiento hasta que todos los movimientos hayan sido ejecutados.
- Se desactiva la niebla de guerra.


DEMO
TIME

Conclusiones

- Gracias al framework hemos podido centrarnos en desarrollar el juego... ¡una vez lo hemos terminado!
- Hacer el desarrollo para Nintendo DS más cómodo no es imposible. ¿Será el SDK oficial similar a nuestro framework?
- Diseñar las APIs de manera que sean cómodas de usar pero también considerando las limitaciones del hardware permite que el framework sea tanto potente como eficiente.
- ¡Disponible en Github!
  `https://github.com/Sumolari/TacticsDS`
- En la memoria del trabajo hay una versión del tutorial «*How to make a bouncing ball game*» usando FMAW Framework.

Bibliografía y recursos I


Curtis Bartley.
Memtrack: Tracking memory allocations in c++, 2002.
http://www.almostinfinite.com/memtrack.html.


Christopher (Eltoshen), Jeff (Ellis), and Nick (Zerxer).
fire emblem planet, 2008.
http://www.feplanet.net/sprites-archive.


Stuart Feldman and desarrolladores de GNU.
Gnu make manual, 2014.
https://www.gnu.org/software/make/manual/.


Mukunda (eKid) Johnson.
How to make a bouncing ball game, 2009.
http://ekid.nintendev.com/bouncy/.

Bibliografía y recursos II


Stephen Lavelle.
Bfxr: make sound effects for your games., 2011.
http://www.bfxr.net/.


Bernardine et al. Miessner.
Free ringtones on myfreeringtones.org, 2009.
http://myfreeringtones.org/title/Final


Phil et al Nash.
Catch, 2010.
https://github.com/philsquared/Catch.


Michael (joat) Noland, Jason (dovoto) Rogers, and Dave (WinterMute) Murphy.
libnds documentation, 2014.
http://libnds.devkitpro.org/.

Bibliografía y recursos III

Stephen (Agent vivid) Odgen.
Ambient click 2, 2006.
http://www.flashkit.com/soundfx/Interfaces/Clicks/ambient_-agent_vi-8701/index.php.

Bjarne Stroustrup.
A tour of c++, 2014.
http://www.stroustrup.com/Tour.html.