Tactics DS

Víctor Grau Moreso
Álvaro Ponce Arévalo
Lluís Ulzurrun de Asanza Sàez

11 de mayo de 2015

Índice

Motivación

Figura : Fire Emblem

Figura : Advance Wars

Motivación

1. ¿Por qué un juego de estrategia?

Motivación

1. ¿Por qué un juego de estrategia?
2. ¿Por qué un ambiente similar al de Fire Emblem?

¿Por qué un juego de estrategia?

1. Es más sencillo de programar por la ausencia de físicas complicadas.

¿Por qué un juego de estrategia?

1. Es más sencillo de programar por la ausencia de físicas complicadas.
2. No tiene mecánicas de juego de elevada complejidad.

¿Por qué un juego de estrategia?

1. Es más sencillo de programar por la ausencia de físicas complicadas.
2. No tiene mecánicas de juego de elevada complejidad.
3. Sencillez a la hora de implementar una IA.

¿Por qué un juego de estrategia?

1. Es más sencillo de programar por la ausencia de físicas complicadas.
2. No tiene mecánicas de juego de elevada complejidad.
3. Sencillez a la hora de implementar una IA.
4. No es imprescindible un guión o historia muy elaborada.

¿Por qué el ambiente de Fire Emblem?

1. Todos nosotros llevamos jugando a esta saga desde GBA.

¿Por qué el ambiente de Fire Emblem?

1. Todos nosotros llevamos jugando a esta saga desde GBA.
2. Hay páginas con todos los sprites necesarios para crear una mini versión del juego.

¿Por qué el ambiente de Fire Emblem?

1. Todos nosotros llevamos jugando a esta saga desde GBA.
2. Hay páginas con todos los sprites necesarios para crear una mini versión del juego.
3. Es más fácil programar un caballero que un avión.

Índice

Mapas

- Distintos tipos de terrenos (montañas, ríos, bosques...).
- Archivos especiales para definir diferentes mapas.
- El tamaño del mapa es siempre el mismo.
- ¡Puedes crear tus propios mapas!

Formato de un fichero de mapa

En cada fichero definimos el escenario por completo, tanto unidades como terreno:

Formato de un fichero de mapa

En cada fichero definimos el escenario por completo, tanto unidades como terreno:

1. **Primera línea:** dos enteros X e Y, separados por un espacio, que definen el número de filas y columnas del tablero.

Formato de un fichero de mapa

En cada fichero definimos el escenario por completo, tanto unidades como terreno:

1. **Primera línea:** dos enteros X e Y, separados por un espacio, que definen el número de filas y columnas del tablero.
2. **Siguientes X líneas:** Y enteros separados por espacios. Cada número indica el tipo de terreno de la celda correspondiente.

Formato de un fichero de mapa

En cada fichero definimos el escenario por completo, tanto unidades como terreno:

1. **Primera línea:** dos enteros X e Y, separados por un espacio, que definen el número de filas y columnas del tablero.
2. **Siguientes X líneas:** Y enteros separados por espacios. Cada número indica el tipo de terreno de la celda correspondiente.
3. Línea en blanco.

Formato de un fichero de mapa

En cada fichero definimos el escenario por completo, tanto unidades como terreno:

1. **Primera línea:** dos enteros X e Y, separados por un espacio, que definen el número de filas y columnas del tablero.
2. **Siguientes X líneas:** Y enteros separados por espacios. Cada número indica el tipo de terreno de la celda correspondiente.
3. Línea en blanco.
4. Un **número indeterminado de líneas** con el siguiente formato: $DT\,(x, y)$, siendo todas las variables enteros, donde:
   - D indica el jugador dueño de la unidad.
   - T indica el tipo de la unidad.
   - x e y indican la celda del tablero donde está la unidad.

Formato de un fichero de mapa

En cada fichero definimos el escenario por completo, tanto unidades como terreno:

1. **Primera línea:** dos enteros X e Y, separados por un espacio, que definen el número de filas y columnas del tablero.
2. **Siguientes X líneas:** Y enteros separados por espacios. Cada número indica el tipo de terreno de la celda correspondiente.
3. Línea en blanco.
4. Un **número indeterminado de líneas** con el siguiente formato: $DT\,(x, y)$, siendo todas las variables enteros, donde:
   - D indica el jugador dueño de la unidad.
   - T indica el tipo de la unidad.
   - x e y indican la celda del tablero donde está la unidad.
5. Fin del fichero.

Ejemplo de mapa: Misty Mountain

```
12 16
2 4 4 2 4 4 4 4 4 0 0 0 0 0 0 0
2 4 2 4 3 4 2 4 3 0 0 0 0 0 0 0
3 4 0 5 4 2 3 2 0 0 0 0 0 4 0 0
4 0 0 5 3 3 2 2 0 0 0 0 0 0 0 0
2 4 0 5 2 2 2 2 0 4 4 0 0 0 0 0
2 4 0 5 2 2 2 3 0 4 4 0 0 0 0 0
4 3 2 5 3 3 0 0 0 5 0 0 0 0 0 0
4 0 0 5 2 2 0 0 0 5 3 2 0 0 0 0
4 4 0 5 2 3 0 0 0 5 3 3 0 0 0 0
3 2 4 5 2 3 0 0 0 1 2 0 0 0 0 0
2 2 4 5 0 0 0 0 0 5 2 0 0 0 0 0
3 3 4 5 0 0 0 0 0 5 0 0 0 0 0 0
```

Ejemplo de mapa: Misty Mountain

```
12 16
2 4 4 2 4 4 4 4 4 0 0 0 0 0 0 0
2 4 2 4 3 4 2 4 3 0 0 0 0 0 0 0
3 4 0 5 4 2 3 2 0 0 0 0 0 4 0 0
4 0 0 5 3 3 2 2 0 0 0 0 0 0 0 0
2 4 0 5 2 2 2 2 0 4 4 0 0 0 0 0
2 4 0 5 2 2 2 3 0 4 4 0 0 0 0 0
4 3 2 5 3 3 0 0 0 5 0 0 0 0 0 0
4 0 0 5 2 2 0 0 0 5 3 2 0 0 0 0
4 4 0 5 2 3 0 0 0 5 3 3 0 0 0 0
3 2 4 5 2 3 0 0 0 1 2 0 0 0 0 0
2 2 4 5 0 0 0 0 0 5 2 0 0 0 0 0
3 3 4 5 0 0 0 0 0 5 0 0 0 0 0 0
```

```
2 1 (2, 2)
2 1 (1, 3)
2 1 (2, 4)
2 1 (1, 5)
1 2 (10, 10)
1 3 (9, 11)
1 1 (11, 11)
```

Ejemplo de mapa: Misty Mountain

```
12  16
2  4  4  2  4  4  4  4  4  0  0  0  0  0  0  0
2  4  2  4  3  4  2  4  3  0  0  0  0  0  0  0
3  4  0  5  4  2  3  2  0  0  0  0  0  4  0  0
4  0  0  5  3  3  2  2  0  0  0  0  0  0  0  0
2  4  0  5  2  2  2  2  0  4  4  0  0  0  0  0
2  4  0  5  2  2  2  3  0  4  4  0  0  0  0  0
4  3  2  5  3  3  0  0  0  5  0  0  0  0  0  0
4  0  0  5  2  2  0  0  0  5  3  2  0  0  0  0
4  4  0  5  2  3  0  0  0  5  3  3  0  0  0  0
3  2  4  5  2  3  0  0  0  1  2  0  0  0  0  0
2  2  4  5  0  0  0  0  0  5  2  0  0  0  0  0
3  3  4  5  0  0  0  0  0  5  0  0  0  0  0  0
```

```
2  1  (2,  2)
2  1  (1,  3)
2  1  (2,  4)
2  1  (1,  5)
1  2  (10,  10)
1  3  (9,  11)
1  1  (11,  11)
```

Mecánicas

- Movimiento por el mapa restringido por el tipo de celda.

Mecánicas

- Movimiento por el mapa restringido por el tipo de celda.
  - Ríos que se pueden atravesar.

Mecánicas

- Movimiento por el mapa restringido por el tipo de celda.
  - Ríos que se pueden atravesar.
  - Montañas y bosques con costes de desplazamiento mayor.

Mecánicas

- Movimiento por el mapa restringido por el tipo de celda.
  - Ríos que se pueden atravesar.
  - Montañas y bosques con costes de desplazamiento mayor.
- Ataques a melee o a distancia.

Mecánicas

- Movimiento por el mapa restringido por el tipo de celda.
  - Ríos que se pueden atravesar.
  - Montañas y bosques con costes de desplazamiento mayor.
- Ataques a melee o a distancia.
- Niebla de guerra que te impide ver el mapa al completo.

Mecánicas

- Movimiento por el mapa restringido por el tipo de celda.
  - Ríos que se pueden atravesar.
  - Montañas y bosques con costes de desplazamiento mayor.
- Ataques a melee o a distancia.
- Niebla de guerra que te impide ver el mapa al completo.
- Modos de juego HvH y HvIA y sistema de reproducir partidas guardadas.

IA

- La IA está sometida a las mismas limitaciones que cualquier jugador humano.

IA

- La IA está sometida a las mismas limitaciones que cualquier jugador humano.
    - Atravesar una montaña cuesta más que atravesar un prado.

IA

- La IA está sometida a las mismas limitaciones que cualquier jugador humano.
  - Atravesar una montaña cuesta más que atravesar un prado.
  - La niebla de guerra le impide ver unidades enemigas.

IA

- La IA está sometida a las mismas limitaciones que cualquier jugador humano.
  - Atravesar una montaña cuesta más que atravesar un prado.
  - La niebla de guerra le impide ver unidades enemigas.
  - Sus unidades tienen el mismo campo de visión, movimiento, salud y ataque que las de los jugadores humanos.
- La IA no tiene tiempo límite para tomar decisiones.

IA

- La IA está sometida a las mismas limitaciones que cualquier jugador humano.
  - Atravesar una montaña cuesta más que atravesar un prado.
  - La niebla de guerra le impide ver unidades enemigas.
  - Sus unidades tienen el mismo campo de visión, movimiento, salud y ataque que las de los jugadores humanos.
- La IA no tiene tiempo límite para tomar decisiones.
- Se compensa con un sistema de toma de decisiones sencillo.

Decisiones de la IA

```
repeat
    haHechoAlgo ← false;
    foreach u ∈ unidadesIA, u tiene acciones restantes do
        pAtacar ← e ∈ unidadesEnemigo, sólo u puede atacar a e;
        if |pAtacar| == 0 then
            pAtacar ← e ∈ unidadesEnemigo, u puede atacar a e;
        end
        if |pAtacar| > 0 then
            u.atacar( randomIn( pAtacar ) );
        else
            e ← enemigoMásCercano[ u ];
            u.irA( celdaAlcanzableMásCercanaA( e.posicion, u ) );
        end
        haHechoAlgo ← true;
    end
until haHechoAlgo == false;
```


DEMO
TIME

Índice

La clase `Grid`

- Contiene el estado del escenario y los métodos que lo alteran (acciones de personajes, niebla de guerra, cursor...).
- El mapa es una matriz de celdas, donde cada celda contiene el tipo de terreno y la unidad (si la hay) de una posición del mapa.

Cálculo de movimientos posibles

- Al seleccionar una unidad, se llama a un método de la clase `Grid` que calcula mediante un algoritmo A los caminos óptimos a todas las celdas alcanzables por la unidad.
- El resultado es una matriz de booleanos del mismo tamaño que el mapa que indica a qué celdas puede y no puede moverse la unidad.

Cálculo de movimientos posibles

```
Data: unidad, tablero
Result: matriz alcanzables
inicializar matriz coste a infinito;
initializar matriz alcanzable a false;
porRevisar ← unidad.celda;
while !porRevisar.vacía() do
    celda ← porRevisar.extraer();
    alcanzable[ celda ] ← true;
    adyacentes ← adyacentesA( celda );
    foreach a∈ adyacentes do
        nuevo ← coste[celda] + costeTerreno[a];
        if coste[a] > nuevo ∧ nuevo ≤ unidad.maxMov then
            coste[a] ← nuevo;
            porRevisar.insertar( a );
        end
    end
end
```

Niebla de guerra

- La niebla de guerra tiene varios modos: `Desactivada`, `Por Defecto`, `Siempre J1`, `Siempre J2`, `Siempre activada`.

Niebla de guerra

- La niebla de guerra tiene varios modos: `Desactivada`, `Por Defecto`, `Siempre J1`, `Siempre J2`, `Siempre activada`.
- Para el cómputo de las celdas visibles en el modo `Por Defecto`, se obtienen todas las unidades del jugador que tiene el turno, y se aplica un algoritmo prácticamente idéntico al del cálculo de celdas alcanzables.

Niebla de guerra

- La niebla de guerra tiene varios modos: `Desactivada`, `Por Defecto`, `Siempre J1`, `Siempre J2`, `Siempre activada`.
- Para el cómputo de las celdas visibles en el modo `Por Defecto`, se obtienen todas las unidades del jugador que tiene el turno, y se aplica un algoritmo prácticamente idéntico al del cálculo de celdas alcanzables.
- En el modo HvIA, la niebla de guerra está en el modo `Siempre J1`, por lo que independientemente del turno se mostrará el mapa en pantalla como si fuese el del jugador 1.

Niebla de guerra

- La niebla de guerra tiene varios modos: `Desactivada`, `Por Defecto`, `Siempre J1`, `Siempre J2`, `Siempre activada`.
- Para el cómputo de las celdas visibles en el modo `Por Defecto`, se obtienen todas las unidades del jugador que tiene el turno, y se aplica un algoritmo prácticamente idéntico al del cálculo de celdas alcanzables.
- En el modo HvIA, la niebla de guerra está en el modo `Siempre J1`, por lo que independientemente del turno se mostrará el mapa en pantalla como si fuese el del jugador 1.
    - ¡Aunque en los cálculos de la IA sí que se tiene en cuenta la niebla de guerra!

Mover o atacar a los personajes

- Al seleccionar a una unidad se calcula su rango de movimiento así como su rango de ataque.

Mover o atacar a los personajes

- Al seleccionar a una unidad se calcula su rango de movimiento así como su rango de ataque.
- Cada unidad cuenta con un número de acciones disponibles por turno.

Mover o atacar a los personajes

- Al seleccionar a una unidad se calcula su rango de movimiento así como su rango de ataque.
- Cada unidad cuenta con un número de acciones disponibles por turno.
- Una unidad puede ejecutar acciones, a elegir entre mover y/o atacar, siempre que no haya consumido todas las acciones disponibles en un turno.

Mover o atacar a los personajes

- Al seleccionar a una unidad se calcula su rango de movimiento así como su rango de ataque.
- Cada unidad cuenta con un número de acciones disponibles por turno.
- Una unidad puede ejecutar acciones, a elegir entre mover y/o atacar, siempre que no haya consumido todas las acciones disponibles en un turno.
- Por defecto, una unidad puede atacar, mover y atacar, o mover y mover una segunda vez una distancia menor.

Mover o atacar a los personajes

- Al seleccionar a una unidad se calcula su rango de movimiento así como su rango de ataque.
- Cada unidad cuenta con un número de acciones disponibles por turno.
- Una unidad puede ejecutar acciones, a elegir entre mover y/o atacar, siempre que no haya consumido todas las acciones disponibles en un turno.
- Por defecto, una unidad puede atacar, mover y atacar, o mover y mover una segunda vez una distancia menor.
- Una excepción es el espadachín, que posee una acción adicional por turno.

Partidas guardadas

- Una de las funciones de la clase `Grid` es guardar un listado de los cambios acontecidos en el escenario paso a paso.

Partidas guardadas

- Una de las funciones de la clase `Grid` es guardar un listado de los cambios acontecidos en el escenario paso a paso.
- Pausando el juego y escogiendo la opción *Cargar Partida* podemos ver una reproducción de la partida hasta el momento.

Partidas guardadas

- Una de las funciones de la clase `Grid` es guardar un listado de los cambios acontecidos en el escenario paso a paso.
- Pausando el juego y escogiendo la opción *Cargar Partida* podemos ver una reproducción de la partida hasta el momento.
- Entre otras cosas, sirve de base para un posible modo online.

Clase `Player`

- Guarda un identificador único que es usado por el `TurnManager` para:

Clase `Player`

- Guarda un identificador único que es usado por el `TurnManager` para:
  - Identificar qué unidades deben restablecer su contador de acciones.

Clase `Player`

- Guarda un identificador único que es usado por el `TurnManager` para:
  - Identificar qué unidades deben restablecer su contador de acciones.
  - Saber qué jugador juega en cada turno.

Clase `Player`

- Guarda un identificador único que es usado por el `TurnManager` para:
  - Identificar qué unidades deben restablecer su contador de acciones.
  - Saber qué jugador juega en cada turno.
  - Proclamar a un vencedor.

Clase `Player`

- Guarda un identificador único que es usado por el `TurnManager` para:
  - Identificar qué unidades deben restablecer su contador de acciones.
  - Saber qué jugador juega en cada turno.
  - Proclamar a un vencedor.
- La IA se vale de la clase `PlayerIA`:

Clase `Player`

- Guarda un identificador único que es usado por el `TurnManager` para:
  - Identificar qué unidades deben restablecer su contador de acciones.
  - Saber qué jugador juega en cada turno.
  - Proclamar a un vencedor.
- La IA se vale de la clase `PlayerIA`:
  - Extiende a la clase `Player`.

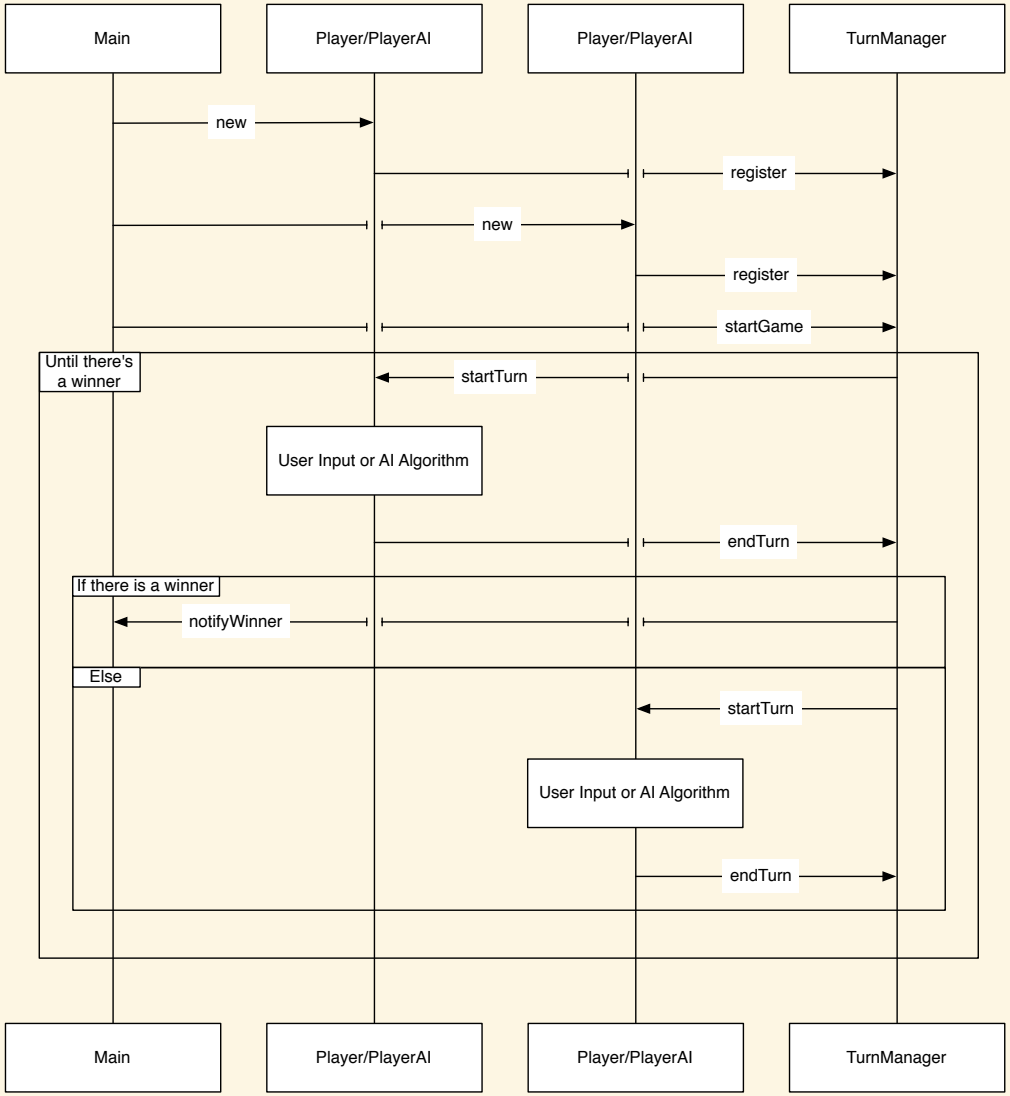
Clase `Player`

- Guarda un identificador único que es usado por el `TurnManager` para:
  - Identificar qué unidades deben restablecer su contador de acciones.
  - Saber qué jugador juega en cada turno.
  - Proclamar a un vencedor.
- La IA se vale de la clase `PlayerIA`:
  - Extiende a la clase `Player`.
  - A nivel del juego la IA es tratada como un jugador más.

Clase `Player`

- Guarda un identificador único que es usado por el `TurnManager` para:
  - Identificar qué unidades deben restablecer su contador de acciones.
  - Saber qué jugador juega en cada turno.
  - Proclamar a un vencedor.
- La IA se vale de la clase `PlayerIA`:
  - Extiende a la clase `Player`.
  - A nivel del juego la IA es tratada como un jugador más.
  - Las acciones que toma la IA utilizan los mismos métodos que las de un jugador humano.

Interacción entre `Player` y `TurnManager`

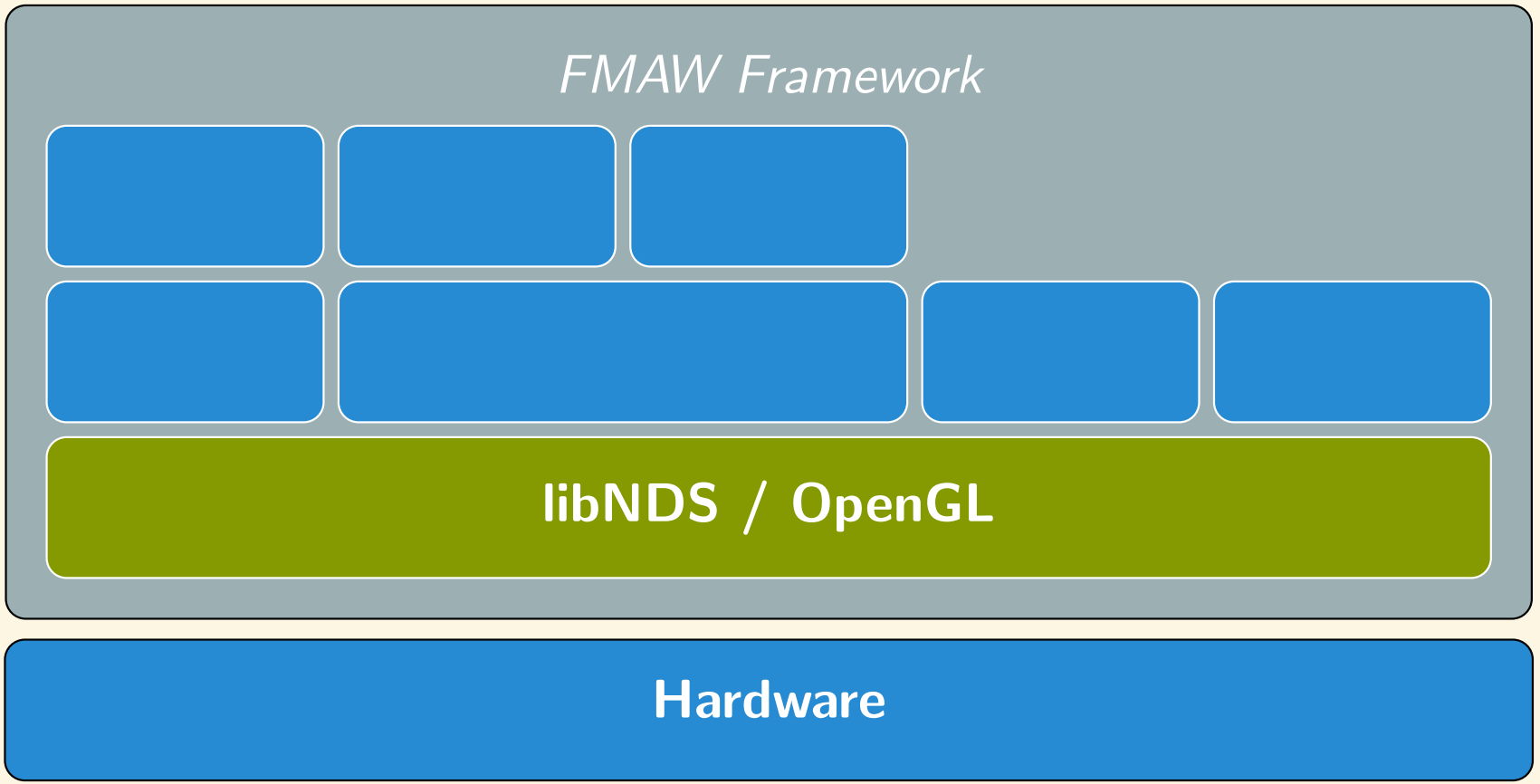


Índice

Objetivos

- Abstracción completa del hardware: para facilitar el desarrollo y la depuración del juego.
- Eficiente: procesadores ARM7 a 33Mhz y ARM9 a 67 Mhz, 4MB de RAM, poca memoria de vídeo...
- Útil: además de abstraer debe aportar nuevas herramientas.
- Portable: queremos pensar en el futuro y poder usarlo en otras plataformas.
- **Solución:** Diseñamos las APIs de manera que sean muy cómodas de usar pero también teniendo en cuenta las limitaciones del hardware.

Arquitectura



Coma fija

- **Problema:** Las operaciones en coma flotante son costosas pero con enteros son más sencillas.
- **Idea:** ¿Por qué no manejar los números decimales con un entero donde algunos bits están reservados para representar la parte decimal del número?
- **Solución:** Clase `FixedReal`.
- La hora de la verdad: actualizad este fragmento.

```
typedef double RealNumber;

RealNumber acc    = -2.5;
RealNumber iniVel = 20.0;
RealNumber iniX   =  0.0;

RealNumber positionAtTime( RealNumber time ) {
    return iniX + iniVel * time + 0.5 * acc * time *
        time;
}
```

Coma fija

- **El truco:** sobrecargar operadores en C++.
- **Curiosidad:** usar mal el operador `new` en la implementación de los operadores introdujo uno de los *memory leak* más complicados de depurar del proyecto.

```
class FixedReal {
private:
    int num;
    short int fraction_size;
public:
    FixedReal operator+(const FixedReal x);
    FixedReal operator-(const FixedReal x);
    FixedReal operator*(const FixedReal x);
    FixedReal operator/(const FixedReal x);
    // ...
    operator int() const;
    operator double() const;
    bool operator<(const FixedReal x) const;
    // ...
};
```

Coma fija

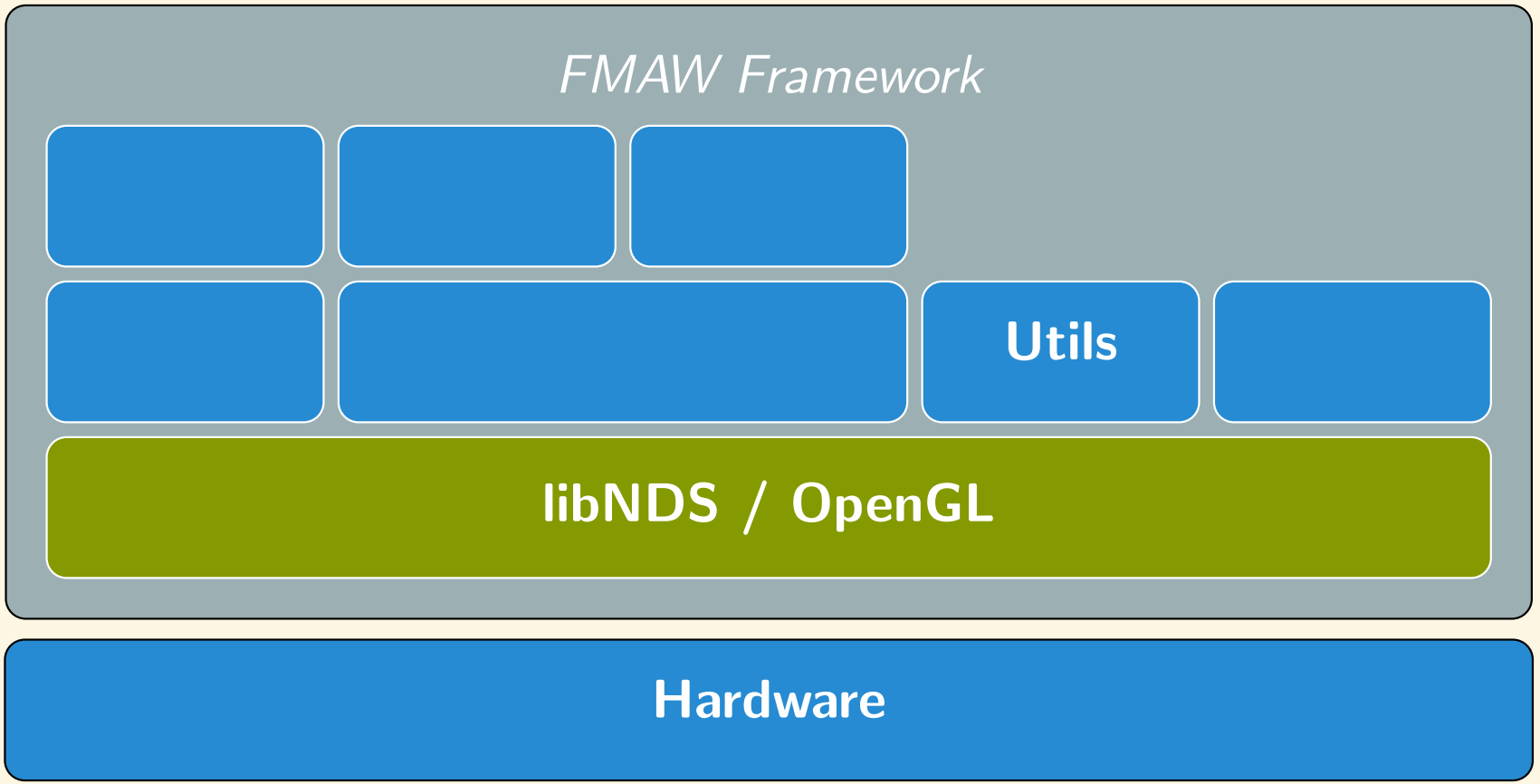
- La hora de la verdad: fragmento actualizado.

```
typedef FixedReal RealNumber;

// Queremos 8 bits de precision.
RealNumber acc    { -2.5, 8 };
RealNumber iniVel { 20.0, 8 };
RealNumber iniX   {  0.0, 8 };

RealNumber positionAtTime( RealNumber time ) {
    return iniX + iniVel * time + 0.5 * acc * time *
        time;
}
```

Arquitectura



Tiles, paletas, backgrounds y sprites

- **Tile:** una imagen, cuyo color está indexado.
- **Paleta:** los colores de una imagen.
- **Background:** un conjunto de 128×128 *tiles* y paletas que forman una rejilla cuadrada (algunos *tiles* se salen de la pantalla).
- **Sprite:** un pareja de *tile* y paleta que puede aparecer en cualquier posición de la pantalla y aplicársele transformaciones diversas.
- **El problema:** cada vez que queremos mostrar una imagen en pantalla hay que copiarla a la memoria de vídeo, establecer su paleta, buscar una posición libre en el registro de *sprites* y establecer los bits correspondientes a 1...
- **La propuesta de `libnds`:** funciones auxiliares para tareas repetitivas.

Tiles, paletas, backgrounds y sprites

- **Antes:** ¿qué hace este código?

```
typedef struct t_spriteEntry {
    u16 attr0, attr1, attr2, affine_data;
} spriteEntry;
#include "./gfx_ball.h"
#define sprites         ((spriteEntry*)OAM)
#define tiles_ball       0
#define pal_ball         0
#define tile2objram(t) (SPRITE_GFX     + (t) * 16)
#define pal2objram(p)  (SPRITE_PALETTE + (p) * 16)
dmaCopyHalfWords( 3, gfx_ballTiles, tile2objram(
    tiles_ball), gfx_ballTilesLen );
dmaCopyHalfWords( 3, gfx_ballPal,   pal2objram(
    pal_ball),    gfx_ballPalLen );
sprites[0].attr0 = 50;
sprites[0].attr1 = 20 + ATTR1_SIZE_16;
sprites[0].attr2 = tiles_ball + (pal_ball << 12);
```

Tiles, paletas, backgrounds y sprites

- **El problema de `libnds`:** no es portable y se puede mejorar (opera con los registros a bajo nivel, es propenso a *overflows*, etc).
- **Nuestra propuesta:** clases `Tile`, `TileAttributes`, `Background` y `Sprite`.
- `TileAttributes` únicamente sirve para definir un `Tile`, de manera que se pueden guardar de forma global y reutilizar las imágenes.
- `Tile` es una abstracción de *tile* con utilidades como un constructor que mezcla la imagen de una *tile* con la paleta de otra.
- `Background` y `Sprite` son abstracciones de *backgrounds* y *sprites* con algunas utilidades adicionales para depuración y reducción de código.

Tiles, paletas, backgrounds y sprites

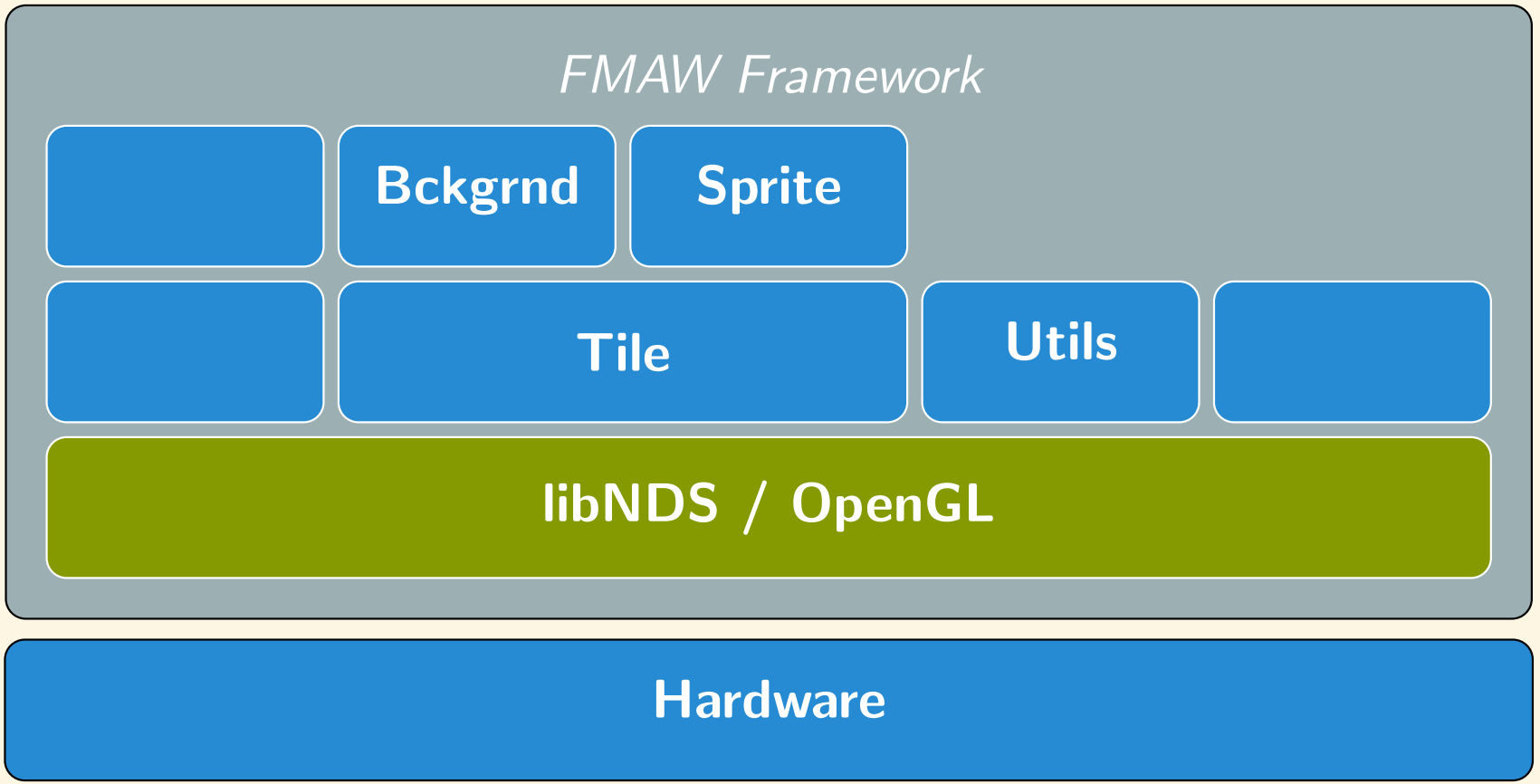
- **Después:** ¿qué hace este código?

```
#include "./gfx_ball.h"
FMAW::TileAttributes ball_attributes{
  gfx_ballTiles,     // Imagen.
  gfx_ballTilesLen, // Tamanyo.
  gfx_ballPal,       // Paleta.
  gfx_ballPalLen,    // Tamanyo.
  FMAW::TypeSprite, // Tipo: fondo o sprite.
  FMAW::ScreenMain  // Pantalla superior o inferior.
};
FMAW::Tile ball_tile(ball_attributes);  // A memoria.
FMAW::Sprite pelota{ball_tile};         // Sprite.
pelota.setPosition( 20, 50 );           // Posicion.
pelota.setSizeMode(FMAW::square16x16); // Tamanyo.
pelota.enable();                        // Visible.
```

Tiles, paletas, backgrounds y sprites

- **Curiosidad:** la memoria de paletas es mucho más reducida que la de *sprites*. Al principio no permitíamos reutilizar paletas y enseguida vimos cómo nos faltaba memoria para las paletas y se corrompían las imágenes.
- **Solución:** añadimos a la clase `Tile` constructores para permitir reutilizar paletas, algo que no habíamos pensado originalmente.

Arquitectura



Temporizadores

- **El problema:** hay tareas que deben ejecutarse de forma periódica o con un retardo en el futuro.
- **La propuesta de `libnds`:** acceso a los 4 relojes físicos de la consola pudiendo iniciarlos y detenerlos, así como ejecutar una función tras pasar un número concreto de *ticks* del procesador.
- **El problema de `libnds`:** las funciones son muy confusas y poco documentadas, además son demasiado explícitas (iniciar reloj, detener reloj, etc).
- **Nuestra solución:** construyamos una API de temporizadores por encima de la de `libnds` que sea más flexible y cómoda de usar.

Temporizadores

- La hora de la verdad: funciones públicas.

```
namespace FMAW {
namespace Timer {

void init();

int enqueue_function(
  std::function<void(int)> callback,
  unsigned int delta,
  bool repetitive
);

bool dequeue_function(int id);

void check();

}  // namespace Timer
}  // namespace FMAW
```

Temporizadores

- La hora de la verdad: funciones públicas.

```
typedef struct t_callback {
  int ID;
  unsigned int  init, delta;
  bool repetitive, toBeRemoved;
  std::function<void(int)> function;
} Callback;

std::map<int, Callback> registered_functions {};
int next_registered_function_id = 0;
unsigned int ticks = 0;

void init() {
  if (ticks == 0) {
    ticks++;
    // Ojo al NULL!
    timerStart(0, ClockDivider_1024, 0, NULL);
  }
}
```

Temporizadores

- **¿Por qué `NULL`?:** Porque no usamos el *callback* del reloj de *hardware*, sino que usamos el reloj sólo para calcular cuánto tiempo ha transcurrido.
- El desarrollador debe manualmente llamar a la función `Timer::check()` antes de cada fotograma.
- De esta manera evitamos posibles condiciones de carrera y otros problemas de la computación paralela.
- Nuestro sistema de temporizadores se asemeja más al modelo asíncrono de JavaScript que a hilos de ejecución.

Temporizadores

- La hora de la verdad: la función `Timer::check()`.

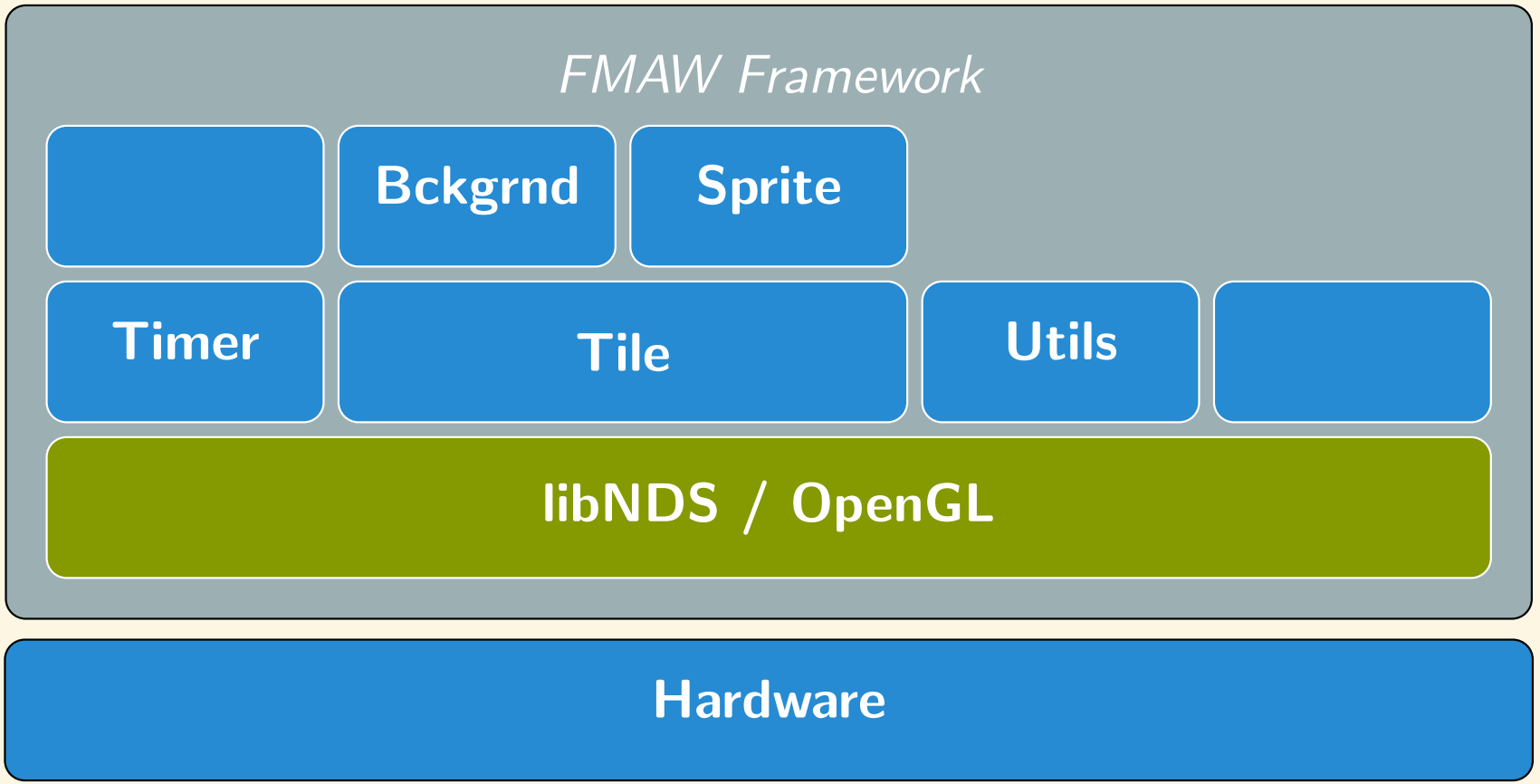
```
void check() {
  std::vector<int> toBeRemoved;
  ticks = RANGE(ticks + timerElapsed(0));

  for ( auto &it : registered_functions.begin() ) {
    Callback &entry = it.second;

    if (!entry.toBeRemoved && ticks > entry.init) {
      entry.function(entry.ID);
      if (entry.repetitive)
        entry.init = RANGE(ticks + entry.delta);
      else dequeue_function(entry.ID);
    }

    if (entry.toBeRemoved)
      toBeRemoved.push_back(entry.ID);
  }
  // ... quitar las callbacks de toBeRemoved
```

Arquitectura



Entrada del usuario

- **Problema:** queremos que se ejecute una *callback* cuando el usuario pulse/mantenga pulsado/suelte ciertos botones.
- **Propuesta de `libnds`:** *polling*. Antes de dibujar cada fotograma el desarrollador debe llamar a la función `scanKeys()`, esta función preparará un registro de manera que al llamar a la función `keysHeld()` se obtenga un vector con el estado (pulsado o no) de cada botón.
- **El problema de `libnds`:** demasiado explícito, es necesario hacer *polling*, la estructura de control para activar o desactivar *callbacks* es compleja, no hay manera directa de saber si un botón ha sido liberado recientemente o no (debemos mantener una estructura de datos adicional en el código del juego para este fin), etc.

Entrada del usuario

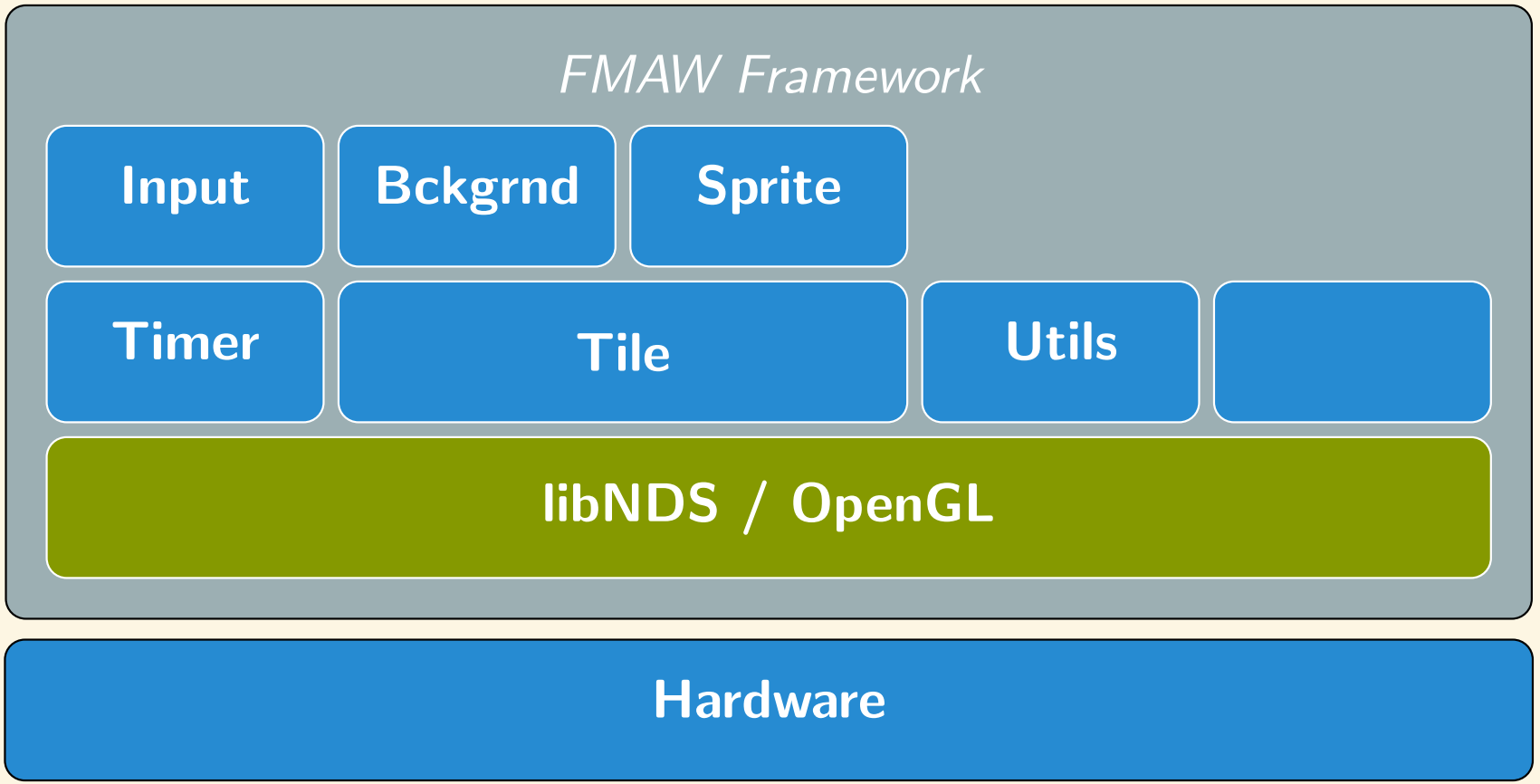
- **Nuestra solución:** una API similar a la de `Timer` que permita registrar y borrar *callbacks* según sea necesario, además de poder registrarlas para los eventos de pulsar botón, mantener botón pulsado o liberar botón.
- La implementación hace todas las consultas que haría el desarrollador de forma explícita pero agrupadas en un único lugar.
- También se encarga de mantener información sobre el estado de los botones (recién pulsado, recién soltado o pulsado) y de permitir registrar y eliminar *callbacks*.
- El *polling* se realiza mediante la API de `Timer`.

Entrada del usuario

- La hora de la verdad: funciones públicas.

```
namespace FMAW {
namespace Input {
// Boton A.
int     onButtonAPressed(std::function<void(void)> c );
int whileButtonAPressed(std::function<void(void)> c );
int    onButtonAReleased(std::function<void(void)> c );
// ...
// Tactil y raton.
int  onTouchPressed(std::function<void(int, int)> c );
int     onTouchMoved(std::function<void(int, int)> c );
int onTouchReleased(std::function<void(int, int)> c );
// Utilidades.
void check();
bool unregisterCallback(int identifier);
}  // namespace Input
}  // namespace FMAW
```

Arquitectura



Sonido

- **Problema:** por una vez las APIs de `libnds` están bastante bien, pero no son portables.
- **Nuestra solución:** encapsular estas APIs de manera que la implementación para otras plataformas utilice un *framework* diferente.

Animaciones

- **Problema:** los personajes no suelen teletransportarse en los videojuegos, y no hay ningún soporte para animaciones en `libnds`.
- **Nuestra solución:** aprovechar la API `Timer` y la clase `Sprite` para dar soporte a animaciones en ciertas operaciones: como el desplazamiento.

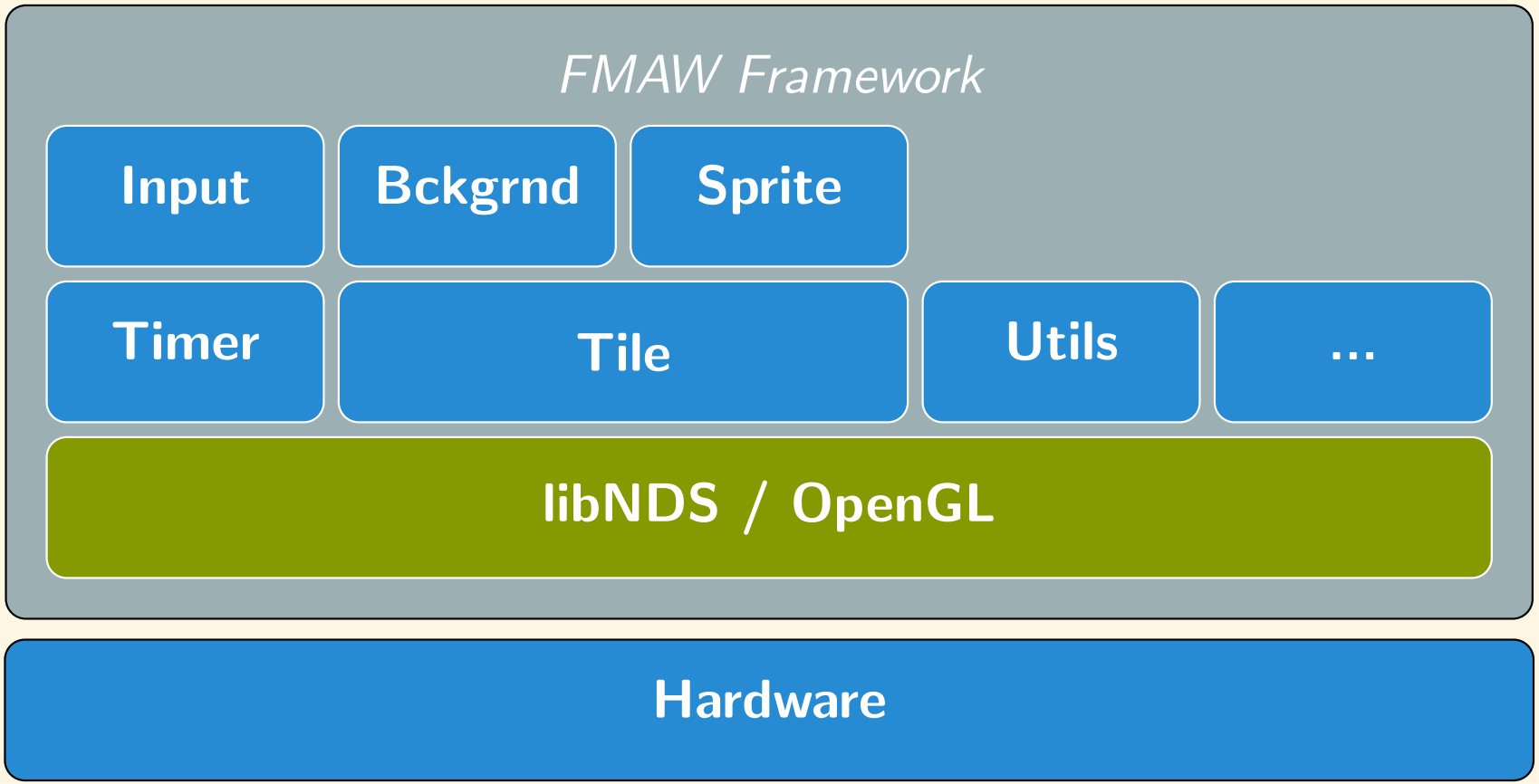
```
animation_id animateToPosition(
  Point position,
  unsigned int duration,
  AnimationType type,
  std::function<void(bool)> callback
);
```

Animaciones

- **¿Cómo funciona?**

Data: Sprite, destino, duración y callback
t_fin$\leftarrow$ t_actual + duración;
origen$\leftarrow$ sprite.posición;
each 5 *ms* **do**
    if *t_actual* $<$ *t_fin* **then**
        per_remaining $\leftarrow$ t_actual / t_fin;
        sprite.posición $\leftarrow$ per_remaining $\times$ (destino $-$ origen);
    else
        sprite.posición $\leftarrow$ destino;
        stopRepeating();
        callback(true);
    end
end

Arquitectura



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.

```
// Copyright 2015 FMAW

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

Conclusiones

- C++ es tremendamente flexible y eficiente.
  Trabajar con él ha sido una gozada.

Conclusiones

- C++ es tremendamente flexible y eficiente.
  Trabajar con él ha sido una gozada.
- Programar sin herramientas de depuración es un pequeño infierno.

Conclusiones

- C++ es tremendamente flexible y eficiente.
  Trabajar con él ha sido una gozada.
- Programar sin herramientas de depuración es un pequeño infierno.
- Programar sin documentación es un infierno mayor.

Conclusiones

- C++ es tremendamente flexible y eficiente.
  Trabajar con él ha sido una gozada.
- Programar sin herramientas de depuración es un pequeño infierno.
- Programar sin documentación es un infierno mayor.
- Podéis descargar nuestro proyecto desde GitHub:
  `https://github.com/Sumolari/TacticsDS`.

Conclusiones

- C++ es tremendamente flexible y eficiente.
  Trabajar con él ha sido una gozada.
- Programar sin herramientas de depuración es un pequeño infierno.
- Programar sin documentación es un infierno mayor.
- Podéis descargar nuestro proyecto desde GitHub:
  `https://github.com/Sumolari/TacticsDS`.
- `new` es el origen de todo mal.