Manual de prácticas de *Computadors*

Fase 2

CandyNDS 2

Versión gráfica

2º curso de Grado en Ingeniería Informática

Universitat Rovira i Virgili

Profesor responsable:	Santiago Romaní (santiago.romani@urv.cat)
Profesores de prácticas:	Pere Millán, Cristina Romero, Carlos Soriano, Víctor Navas

Tabla de contenidos

1 Descripción general de la fase 2

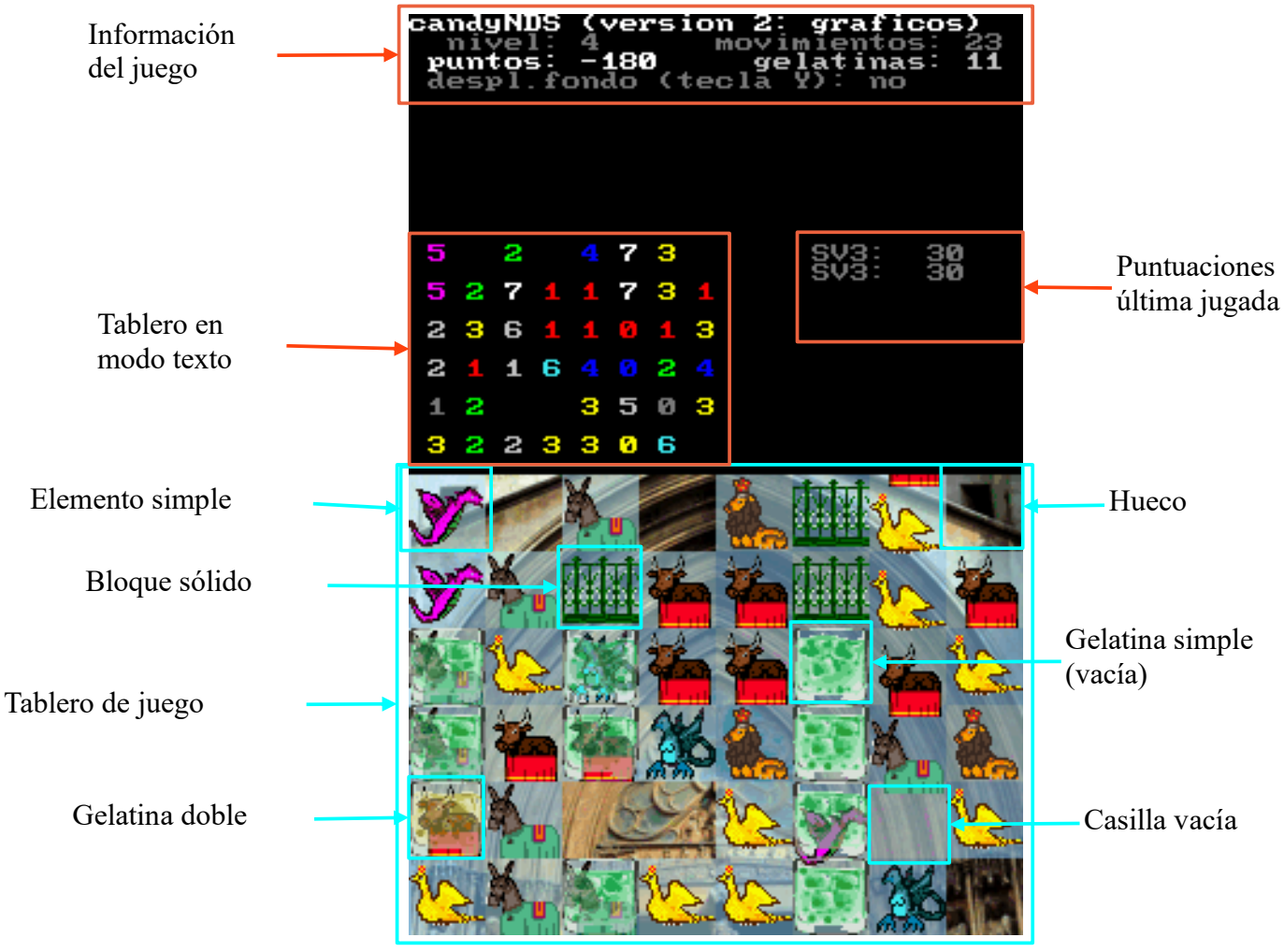
1.1 Funcionamiento básico

Se propone completar la fase 1 de la práctica descrita en el **Manual de Prácticas 1**, que consistía en implementar una versión del juego ***Candy Crush*** representada exclusivamente con mensajes de texto (matriz de números).

En la fase 2 de la práctica se utilizarán los recursos gráficos de la NDS para representar los elementos del juego, los bloques sólidos y las gelatinas simples y dobles, más un patrón ajedrezado de cuadros para marcar cada casilla del tablero (exceptuando los huecos) y una imagen (foto) como fondo general de pantalla.

Por otro lado, habrá que animar dichos gráficos, lo cual requiere interrupciones periódicas para conseguir que la posición y forma de todos los elementos gráficos cambie progresivamente.

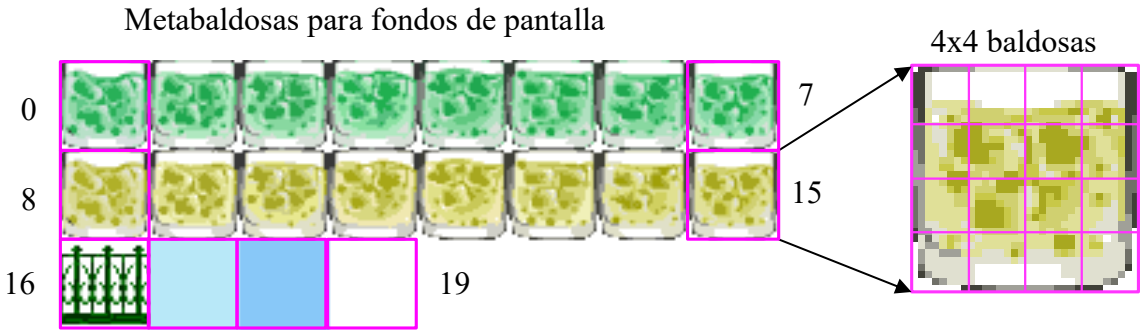
La siguiente figura representa un ejemplo de visualización gráfica, utilizando las dos pantallas de la NDS:



Los componentes gráficos que se utilizarán serán los siguientes:

- **Elementos**: los elementos del juego se representarán con *sprites* de 32x32 píxeles, a partir de un conjunto determinado de metabaldosas (grupos de 4x4 baldosas de 8x8 píxeles cada una), por ejemplo:

- **Bloques del tablero**: a partir de un determinado conjunto de metabaldosas, también de 32x32 píxeles, se definirán los gráficos de las gelatinas simples y dobles (8 versiones por tipo de gelatina), el bloque sólido, los cuadros de las casillas y el hueco (transparente), que se cargarán en los mapas de baldosas de los fondos gráficos de pantalla:



- **Imagen de fondo**: como fondo general de la pantalla gráfica, se cargará una imagen bitmap de 256 filas x 512 columnas, en color real (16 bits/píxel), por ejemplo:

Para realizar los movimientos y cambios de forma de los objetos gráficos anteriores, se proponen 4 tipos de animación:

- **Movimiento de elementos**: los *sprites* se deben mover gradualmente (a nivel de píxeles) por la pantalla para visualizar los cambios de posición, las bajadas verticales, las bajadas laterales y la recombinación del tablero, además de gestionar la creación y eliminación de los elementos (mostrar/ocultar *sprites*).

- **Escalado de elementos**: para representar las sugerencias de combinaciones, los *sprites* de los 3 elementos propuestos se deben reducir y aumentar gradualmente, usando el escalado de *sprites*.

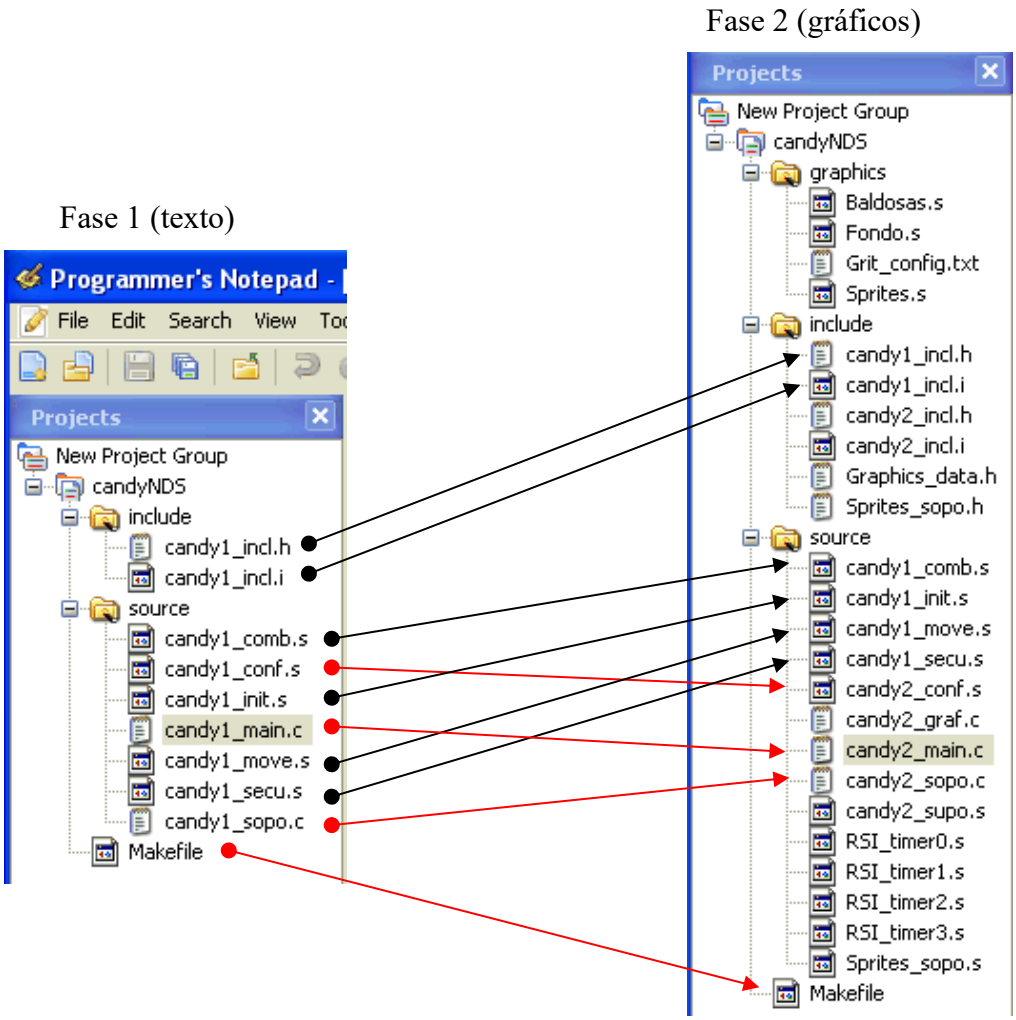
- **Animación de gelatinas**: la representación de las gelatinas deberá cambiar periódicamente entre un conjunto de 8 metabaldosas consecutivas, según el tipo de gelatina.

- **Desplazamiento de fondo**: la imagen de fondo general se girará 90º en sentido horario y deberá desplazarse periódicamente para mostrar todos sus píxeles, alternando el sentido de desplazamiento arriba/abajo cuando se llegue al límite correspondiente.

Como última consideración general, hay que observar que el tablero de juego quedará reducido en esta segunda fase a **6 filas por 8 columnas**, debido a las dimensiones máximas de la pantalla y el tamaño de las metabaldosas.

1.2 Estructura definitiva del proyecto

En esta segunda fase del proyecto, el conjunto de ficheros evoluciona del siguiente modo:



En el esquema anterior, los ficheros que se heredan de la fase 1 se han conectado con flechas con los ficheros correspondientes de la fase 2, resaltando en rojo las flechas de aquellos ficheros que se proporcionan de nuevo con un cambio de nombre (`candy1_` por `candy2_`), además de una nueva versión del `Makefile` y del fichero de proyecto `candyNDS.pnproj` (no mostrado en el gráfico anterior).

Dentro del directorio comprimido `Compus.zip`, accesible en el espacio *Moodle* de la asignatura, hay contenida una carpeta de nombre `support_files_2` con todos los ficheros de soporte necesarios para realizar la segunda fase de la práctica.

Atención: como hay que seguir trabajando sobre el repositorio `git` asignado, será necesario incorporar todos los ficheros adicionales (`git add`) y actualizados sobre el último *commit* de la rama `master`, y después fusionar en nuevo *commit* (`git merge`) con el resto de ramas de los programadores.

A continuación se describe el uso de todos los nuevos ficheros para la fase 2, resaltando en negrita aquellos ficheros que contendrán el código de las tareas a realizar:

`Baldosas.s`:	datos de las baldosas de gelatinas, bloque sólido y cuadros
`Fondo.s`:	datos de la imagen de fondo general del juego
`Sprites.s`:	datos de las baldosas para los elementos del juego
`candy2_incl.h`:	declaraciones en C para la fase 2
`candy2_incl.i`:	declaraciones en ensamblador para la fase 2
`Graphics_data.h`:	declaraciones en C de los datos gráficos (baldosas, fondo, *sprites*)
`Sprites_sopo.h`:	declaraciones en C de las rutinas de `Sprites_sopo.c`
`candy2_conf.s`:	adaptación de las variables globales de configuración del juego
`candy2_graf.c`:	funciones de inicialización de los gráficos del juego
`candy2_main.c`:	adaptación del programa principal de control del juego
`candy2_sopo.c`:	adaptación de las funciones de soporte al programa principal
`candy2_supo.s`:	rutinas de soporte a la gestión de los gráficos del juego
`RSI_timer0.s`:	rutinas para controlar el movimiento de los *sprites*
`RSI_timer1.s`:	rutinas para controlar el escalado de los *sprites*
`RSI_timer2.s`:	rutinas para controlar la animación de las gelatinas
`RSI_timer3.s`:	rutinas para controlar el desplazamiento de la imagen de fondo
`Sprites_sopo.s`:	rutinas para gestionar los *sprites* de la NDS
`Makefile`:	adaptación de los comandos de compilación (incluyen los gráficos)

1.3 Tareas de la fase 2

A continuación se listan las tareas a realizar, divididas en tres conjuntos: **inicializaciones gráficas (2A-2D)**, **animaciones gráficas (2E-2H)** y **actualización de rutinas de la fase 1 (2I)**. Cada tarea se desglosa en varias subtareas, cuyo funcionamiento se detallará más adelante en otros apartados de este manual:

Inicializaciones gráficas

Tarea 2A (`candy2_graf.c`)

- **Tarea 2Aa** → `init_grafA()`: la parte de la función de inicialización del procesador gráfico principal encargada de reservar la memoria gráfica para las baldosas de los *sprites*, además de cargar dichas baldosas y la paleta de colores asociada,
- **Tarea 2Ab** → `genera_sprites(*mat)`: función para generar los *sprites* de los elementos de la matriz de juego que se pasa por parámetro, así como los datos correspondientes en el vector de elementos.

Tarea 2B (`candy2_graf.c`)

- **Tarea 2Ba** → `init_grafA()`: la parte de la función de inicialización del procesador gráfico principal encargada de reservar la memoria gráfica para el fondo 2, además de cargar las baldosas y la paleta de colores asociadas,
- **Tarea 2Bb** → `genera_mapa2(*mat)`: función para generar el mapa de baldosas asociado al fondo 2, que debe representar el tablero de juego con un patrón de cuadros ajedrezado, con los huecos pertinentes.

Tarea 2C (`candy2_graf.c`)

- **Tarea 2Ca** → `init_grafA()`: la parte de la función de inicialización del procesador gráfico principal encargada de reservar la memoria gráfica para el fondo 1, además de cargar las baldosas y la paleta de colores asociadas (mismas baldosas y paleta de la tarea 2Ba, compartidas),
- **Tarea 2Cb** → `genera_mapa1(*mat)`: función para generar el mapa de baldosas asociado al fondo 1, que debe representar los bloques sólidos y las gelatinas simples y dobles, así como los datos correspondientes en la matriz de gelatinas.

Tarea 2D (`candy2_graf.c`)

- **Tarea 2Da** → `init_grafA()`: la parte de la función de inicialización del procesador gráfico principal encargada de reservar la memoria gráfica para el fondo 3, además de cargar los píxeles de la imagen de fondo general del juego,
- **Tarea 2Db** → `ajusta_imagen3()`: función para rotar y desplazar la imagen del fondo 3, de manera que se sitúe en vertical y con la primera columna de píxeles de la imagen alineada con la primera fila de píxeles de la pantalla.

Animaciones gráficas

Tarea 2E (`RSI_timer0.s`)

- **Tarea 2Ea** → `rsi_vblank()`: la parte de la RSI del retroceso vertical encargada de actualizar los registros de control de los *sprites*,
- **Tarea 2Eb** → `activa_timer0(init)`: rutina para activar el *timer* 0, inicializando o no el divisor de frecuencia según el parámetro `init`,
- **Tarea 2Ec** → `desactiva_timer0()`: rutina para desactivar el *timer* 0,
- **Tarea 2Ed** → `rsi_timer0()`: RSI del *timer* 0, encargada de actualizar las coordenadas `(px,py)` de todos los *sprites* activos según su velocidad `(vx,vy)`, además de actualizar el divisor de frecuencia para acelerar el movimiento.

Tarea 2F (`RSI_timer1.s`)

- **Tarea 2Fa** → `rsi_vblank()`: la parte de la RSI del retroceso vertical encargada de actualizar los registros de control de los *sprites* (igual a 2Ea),
- **Tarea 2Fb** → `activa_timer1(init)`: rutina para activar el *timer* 1, inicializando el sentido de escalado según el parámetro `init`,
- **Tarea 2Fc** → `desactiva_timer1()`: rutina para desactivar el *timer* 1,
- **Tarea 2Fd** → `rsi_timer1()`: RSI del *timer* 1, encargada de actualizar el factor de escalado de los *sprites* a resaltar, en el sentido actual.

Tarea 2G (`RSI_timer2.s`)

- **Tarea 2Ga** → `rsi_vblank()`: la parte de la RSI del retroceso vertical encargada de cambiar cíclicamente las metabaldosas que representan el dibujo de las gelatinas (a modificar en el fichero `RSI_timer0.s`),
- **Tarea 2Gb** → `activa_timer2()`: rutina para activar el *timer* 2,
- **Tarea 2Gc** → `desactiva_timer2()`: rutina para desactivar el *timer* 2,
- **Tarea 2Gd** → `rsi_timer2()`: RSI del *timer* 2, encargada de decrementar el código de cambio de cada gelatina del tablero, comunicando a la `rsi_vblank()` cuando haya que cambiar las metabaldosas de al menos una gelatina.

Tarea 2H (`RSI_timer3.s`)

- **Tarea 2Ha** → `rsi_vblank()`: la parte de la RSI del retroceso vertical encargada de actualizar los registros de control de posición del fondo 3 (a modificar en el fichero `RSI_timer0.s`),
- **Tarea 2Hb** → `activa_timer3()`: rutina para activar el *timer* 3,
- **Tarea 2Hc** → `desactiva_timer3()`: rutina para desactivar el *timer* 3,
- **Tarea 2Hd** → `rsi_timer3()`: RSI del *timer* 3, encargada de actualizar el desplazamiento del fondo 3 según el sentido actual, comunicando a la `rsi_vblank()` cuando haya que actualizar el registro de control pertinente.

Actualización de rutinas de la fase 1

Tarea 2I

- **Tarea 2Ia** (`candy1_init.s`) → `recombina_elementos()`: activar el movimiento de los *sprites* de acuerdo con la reubicación de los elementos de la matriz de juego,
- **Tarea 2Ib** (`candy1_secu.s`) → `elimina_secuencias()`: eliminar los elementos (ocultar *sprites*) y las gelatinas (sustituir metabaldosas) de las secuencias que se eliminen en cada jugada,

- **Tarea 2Ic** (`candy1_move.s`) → `baja_verticales()`: activar el movimiento de los *sprites* de acuerdo con la bajada de elementos en vertical, además de inicializar los *sprites* asociados a los nuevos elementos que se van generando al eliminar secuencias,

- **Tarea 2Id** (`candy1_move.s`) → `baja_laterales()`: activar el movimiento de los *sprites* de acuerdo con la bajada de elementos en diagonal.

1.4 Distribución de tareas

Como en la primera parte de la práctica, la implementación de todas las tareas de esta segunda parte habrá que repartirlas entre los componentes de cada grupo de prácticas, atendiendo a los siguientes requisitos:

- cada programador tendrá que realizar **2 tareas completas**, una del conjunto de **inicializaciones gráficas** y otra del conjunto de **animaciones gráficas**,

- las 2 tareas a elegir deben de corresponder a un mismo bloque, tal como se describe en el apartado sobre valoración de las tareas,

- las subtareas de la tarea **2I** (**actualización de rutinas de la fase 1**) se podrán repartir libremente entre los programadores del grupo de prácticas, aunque es preferible asignar las rutinas a modificar al programador que las haya implementado en la fase 1, lógicamente,

- cada tarea tiene una valoración diferente, según su grado de dificultad (ver apartado sobre valoración de las tareas),

- los grupos de menos de 4 programadores podrán presentar una versión incompleta de la práctica, aunque esto implicará una limitación de la nota máxima que se podrá obtener (ver apartado sobre valoración de las tareas).

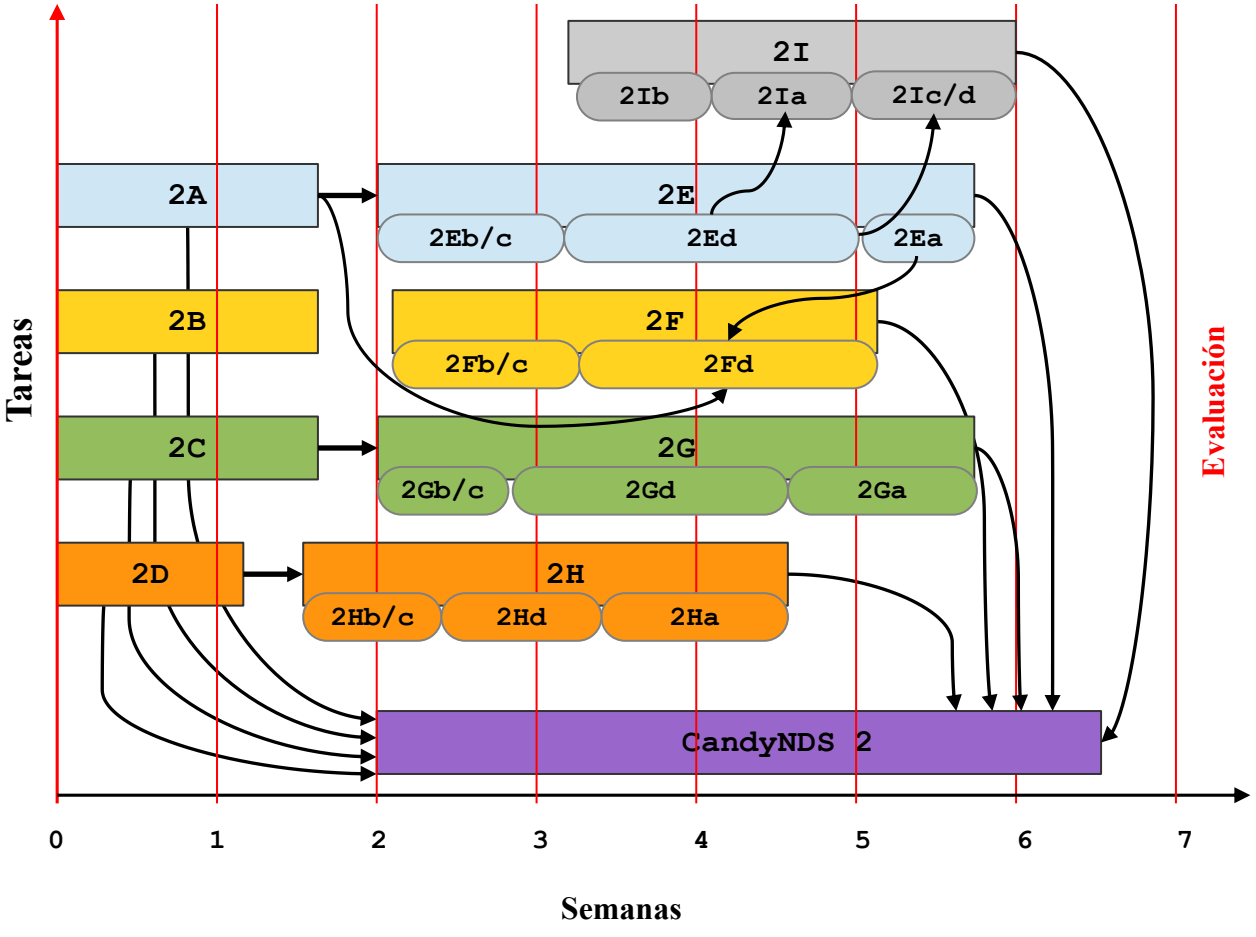
Además, hay que tener en cuenta que las rutinas escritas por todos los programadores se tendrán que unir en **una única versión definitiva** del proyecto. Este trabajo se tendrá que distribuir entre todos los componentes del grupo.

1.5 Planificación temporal de las tareas

En esta segunda fase de la práctica no existe ninguna dependencia directa entre las rutinas de las distintas subtareas, es decir, no hay ninguna rutina de un programador que invoque directamente a otra rutina de otro programador.

Sin embargo, **sí existen dependencias indirectas**, es decir, subtareas que no pueden funcionar si otra subtarea no está completa. Por supuesto, la integración de todas las subtareas realizadas es la última dependencia indirecta que todo proyecto debe resolver.

Por lo tanto, insistimos en que para acabar un proyecto a tiempo es imprescindible ceñirse a una planificación temporal de sus distintas tareas y subtareas. A continuación se propone una planificación para 7 semanas desde que se explica la segunda parte de la práctica hasta que se realizará la evaluación (sin contar períodos de vacaciones intermedios):



Las tareas **2A**, **2C** y **2D** suponen dependencias para las tareas **2E**, **2G** y **2H**, respectivamente, lo cual implica que es preferible asignar las tareas del mismo color a un mismo programador. Además, las tareas **2A** y las subtareas **2Ea** y **2Ed** resultan imprescindibles para otras subtareas relacionadas con los *sprites* (**2Fd**, **2Ia**, **2Ic**, **2Id**).

La tarea CandyNDS 2 representa la versión completa del proyecto, sobre la cual hay que ir incorporando paulatinamente el código generado por todos los programadores. Por tanto, es necesario **acabar las tareas de las rutinas antes de la sexta semana**, con el fin de tener tiempo suficiente para ajustar la integración de todas las rutinas dentro de la versión definitiva del proyecto, es decir, poder solucionar los posibles fallos de ejecución que pueden aparecer en dicha integración.

1.6 Valoración de las tareas

A continuación se muestra una tabla con la valoración máxima (sobre 10) de cada tarea y sus subtareas, agrupadas en 4 bloques que deben ser asignados cada uno a un mismo programador, además de la tarea **2I**, cuyas subtareas se pueden repartir entre diversos programadores:

Bloque 1	**2A** (`candy2_graf.c`)	**4 puntos**	**2E** (`RSI_timer0.s`)	**7 puntos**
	2Aa: `init_graf()`	1 punto	2Ea: `rsi_vblank()`	1 punto
	2Ab: `genera_sprites()`	3 puntos	2Eb: `activa_timer0()`	1,5 puntos
			2Ec: `desactiva_timer0()`	0,5 puntos
			2Ed: `rsi_timer0()`	4 puntos
Bloque 2	**2B** (`candy2_graf.c`)	**4 puntos**	**2F** (`RSI_timer1.s`)	**6 puntos**
	2Ba: `init_graf()`	1 punto	2Fb: `activa_timer1()`	1,5 puntos
	2Bb: `genera_mapa2()`	3 puntos	2Fc: `desactiva_timer1()`	0,5 puntos
			2Fd: `rsi_timer1()`	4 puntos
Bloque 3	**2C** (`candy2_graf.c`)	**4 puntos**	**2G** (`RSI_timer2.s`)	**7 puntos**
	2Ca: `init_graf()`	1 punto	2Ga: `rsi_vblank()`	2 puntos
	2Cb: `genera_mapa1()`	3 puntos	2Gb: `activa_timer2()`	1,5 puntos
			2Gc: `desactiva_timer2()`	0,5 puntos
			2Gd: `rsi_timer2()`	3 puntos
Bloque 4	**2D** (`candy2_graf.c`)	**3 puntos**	**2H** (`RSI_timer3.s`)	**6 puntos**
	2Da: `init_graf()`	1 punto	2Ha: `rsi_vblank()`	2 puntos
	2Db: `ajusta_imagen3()`	2 puntos	2Hb: `activa_timer3()`	1,5 puntos
			2Hc: `desactiva_timer3()`	0,5 puntos
			2Hd: `rsi_timer3()`	2 puntos

	2I (diversos ficheros)	**6 puntos**
	2Ia: *`recombina_elementos()`	1 punto
	2Ib: *`elimina_secuencia()`	2 puntos
	2Ic: *`baja_verticales()`	2 puntos
	2Id: *`baja_laterales()`	1 punto

La suma de los puntos por cada bloque varía entre 9 y 11 puntos. Estas diferencias se pueden compensar con los puntos que se pueden acumular con la realización de las subtareas **2Ix**.

Independientemente del número de bloques y subtareas **2Ix** que se realicen, la nota máxima final vendrá limitada por el número de bloques que se presenten fusionados en la versión definitiva, según la siguiente tabla:

1 bloque + 1 subtarea 2Ix	máx. 7 puntos
2 bloques + 2 subtareas 2Ix	máx. 9 puntos
3 bloques + 3 subtareas 2Ix	máx. 10 puntos
4 bloques + 4 subtareas 2Ix	máx. 11 puntos

La modulación del máximo de puntos está pensada para permitir presentar proyectos incompletos, al mismo tiempo que se tiene en cuenta la **menor** dificultad para fusionar menos rutinas del proyecto. De este modo, se pueden formar grupos de menos de 4 alumnos.

Sin embargo, **no se contabilizarán las tareas y subtareas acabadas pero no fusionadas**. Es decir, si se presentan 3 bloques acabados (junto con el correspondiente número de subtareas **2Ix**), 2 fusionados y uno no, los alumnos que hayan fusionado sus rutinas podrán llegar a 9 puntos, mientras que el alumno que presente sus rutinas no fusionadas con el resto del proyecto solo podrá llegar a 7 puntos.

Los grupos de prácticas compuestos por menos de 4 alumnos también pueden optar a máximos superiores si completan (y fusionan) bloques adicionales. Por ejemplo, un grupo de 2 alumnos que presente 3 bloques fusionados (junto con el correspondiente número de subtareas **2Ix**) puede optar a un máximo de 10 puntos.

Así mismo, si alguno de los componentes de un grupo **no** cumple con su trabajo, el resto de los componentes pueden asumir sus tareas, con lo cual podrán optar a la nota máxima correspondiente de los bloques fusionados, además de acumular los puntos de las tareas extra que realicen (hasta el máximo correspondiente). En este caso, el alumno que no realice una tarea se quedará sin los puntos correspondientes.

Se pide, además, realizar **un programa de test** (función `main()` específica + juego de pruebas) para cada bloque de tareas principales implementado (bloques del 1 al 4). Por ejemplo, el programa de test para el bloque 3 deberá testear **todas** las rutinas de las tareas **2C** y **2G**.

Al realizar dichos programas de test aparecerán dependencias entre tareas de distintos bloques, por ejemplo, entre las tareas **2F** (bloque 3) y **2A** (bloque 1), ya que se necesita la inicialización de los *sprites* realizada en **2A** para que las rutinas de la tarea **2F** puedan visualizar el escalado de los *sprites*. La aparición de este tipo de dependencias es prácticamente inevitable en los proyectos informáticos. Para salvar estas situaciones, no quedará más remedio que estrechar la **colaboración** entre los programadores encargados de las rutinas que tienen dichas dependencias.

El test de las tareas **2Ix** se incluirá en los programas de test de los bloques que las utilicen.

Atención: en caso de **no** proporcionar el programa de test con su juego de pruebas adjunto, **la práctica se considerará suspendida**. Por supuesto, dicho programa de test debe compilar, *linkar* y ejecutarse correctamente, para poder considerarlo como válido para la evaluación.

2 Contexto de la fase 2

2.1 Ficheros de definiciones

Tal como se comentó en el manual de la fase 1 de la práctica, el número de filas y columnas del tablero de juego cambia en esta fase 2, por lo cual se recomendaba utilizar símbolos para representar los límites del tablero, en vez de usar número fijos.

Por lo tanto, uno de los primeros cambios a realizar es el valor de dichos límites, en `candy1_incl.i`:

```
@;== candy1_incl.i: definiciones comunes para ficheros en ensamblador  ===

@; Rango de los números de filas y de columnas -> mínimo: 3, máximo: 11
ROWS = 6
COLUMNS = 8
```

y en `candy1_incl.h`:

```
/*------------------------------------------------------------------------

     $Id: candy1_incl.h $

     Definiciones externas en C para la versión 1 del juego (modo texto)

------------------------------------------------------------------------*/

// Rango de los números de filas y de columnas:
// mínimo: 3, máximo: 11
#define ROWS     6               // dimensiones de la matriz de juego
#define COLUMNS  8
#define DFIL     (24-ROWS*2)     // desplazamiento vertical de filas

#define MAXLEVEL 9               // nivel máximo (niveles 0..MAXLEVEL-1)
```

En teoría, la recompilación de todos los ficheros fuente que incluyen estos ficheros de definiciones debería producir un nuevo ejecutable adaptado a un tablero de 6 filas x 8 columnas. Es **necesario** hacer un `make clean` y recompilar para que todos los ficheros de código fuente utilicen las nuevas dimensiones definidas en los ficheros *include*.

Para esta segunda fase se proporcionan dos ficheros de definiciones más, uno para código ensamblador y otro para código C.

El fichero de definiciones en ensamblador se llama `candy2_incl.i`, y contiene lo siguiente:

```
@;== candy2_incl.i: definiciones comunes para ficheros en ensamblador  ===
@;==                    (versión 2)                                     ===

.include "../include/candy1_incl.i"

@; Píxeles por casilla del tablero de juego
MTWIDTH = 256 / COLUMNS                 @; núm. píxeles de ancho (e.g. 32)
MTHEIGHT = 192 / ROWS                   @; núm. píxeles de alto (e.g. 32)

@; Dimensiones de las metabaldosas
MTROWS = MTHEIGHT / 8                   @; núm. filas metabaldosa (e.g. 4)
MTCOLS = MTWIDTH / 8                    @; núm. columnas metabaldosa (4)
MTOTAL = MTROWS * MTCOLS                @; núm. total de baldosas simples

@; Estructura 'elemento' (ver fichero 'candy2_incl.h")
ELE_II = 0
ELE_PX = 2
ELE_PY = 4
ELE_VX = 6
ELE_VY = 8
ELE_TAM = 10

@; Estructura 'gelatina' (ver fichero 'candy2_incl.h")
GEL_II = 0
GEL_IM = 1
GEL_TAM = 2
```

Las definiciones `MT...` sirven para definir las dimensiones de las metabaldosas que se utilizan para generar los gráficos de los *sprites* y los fondos del juego. Aunque sus valores serán constantes durante todo el desarrollo de la práctica (32x32 píxeles, 4x4 baldosas, etc.), siempre es conveniente usar símbolos que permiten que el código fuente sea lo más adaptable posible.

Las definiciones `ELE_...` y `GEL_...` servirán para acceder a los campos de las estructuras `elemento` y `gelatina`, que se describen en el fichero `candy2_incl.h`:

```
/*------------------------------------------------------------------------

     $Id: candy2_incl.h $

     Definiciones externas en C para la versión 2 del juego (modo gráfico)

------------------------------------------------------------------------*/

#include <candy1_incl.h>

// Pixeles por casilla del tablero de juego
#define MTWIDTH  (256/COLUMNS)          // núm. píxeles de ancho (e.g. 32)
#define MTHEIGHT   (192/ROWS)           // núm. píxeles de alto (e.g. 32)

// Dimensiones de las metabaldosas:
#define MTROWS   (MTHEIGHT/8)           // núm. filas metabaldosa (e.g. 4)
#define MTCOLS   (MTWIDTH/8)            // núm. columnas metabaldosa (4)
#define MTOTAL   MTROWS*MTCOLS          // núm. total de baldosas simples

// estructura de datos relativos a un elemento
typedef struct
{
     short  ii;             // número de interrupciones pendientes (0..32)
                            // o (-1) si está inactivo
     short  px;             // posición x (0..256)
     short  py;             // posición y (-32..192)
     short  vx;             // velocidad x
     short  vy;             // velocidad y
} elemento;

// estructura de datos relativos a una gelatina
typedef struct
{
     char  ii;              // número de interrupciones pendientes (0..10)
                            // o (-1) si está inactivo
     char  im;              // índice de metabaldosa (0..7/8..15)
} gelatina;
```

La estructura `elemento` se usará para gestionar la posición y la velocidad de los *sprites*, mientras que la estructura `gelatina` se usará para gestionar la animación de las gelatinas.

Por el momento, hay que entender que estas estructuras disponen de ciertos campos con un tamaño concreto para cada campo (1 o 2 bytes), y que las definiciones para lenguaje ensamblador relativas a dichos campos y al tamaño de toda la estructura tienen el propósito de parametrizar el acceso a memoria de los campos de dichas estructuras.

2.2 Fichero de configuración

El fichero `candy1_conf.s` se debe sustituir por el nuevo fichero `candy2_conf.s`, donde se proporcionan configuraciones del tablero de juego de 6 filas por 8 columnas, con la misma codificación presentada en la primera parte de la práctica.

Por ejemplo, el nuevo mapa para el nivel 4 que se proporciona es el siguiente:

```
…
            .global mapas
      mapas:
…
      @; mapa 4: gelatinas dobles (+ elementos prefijados)
            .byte 0,15,0,15,0,7,0,15
            .byte 0,0,7,0,0,7,0,0
            .byte 10,3,8,1,1,8,3,3
            .byte 10,1,9,0,0,20,3,4
            .byte 17,2,15,15,3,19,4,3
            .byte 3,2,10,0,0,20,0,15
```

Una vez más, informamos que los números contenidos en estos mapas son solo valores de ejemplo, es decir, no hay ningún inconveniente en que se cambien para definir **juegos de pruebas propios**.

2.3 El programa principal y sus funciones de soporte

El fichero `candy2_main.c` contiene el nuevo código del programa principal que controla todo el juego. Es muy similar al proporcionado en la fase anterior, de modo que a continuación solo resaltamos las diferencias introducidas:

```
int main(void)
{     …          // definición de variables locales e inicializaciones
      init_grafA();
      inicializa_interrupciones();
      do                    // bucle principal del juego
      {
            if (initializing)   //// SECCIÓN DE INICIALIZACIÓN ////
            {
              // inicializa matriz;
              genera_sprites(matrix);
              genera_mapa1(matrix);
              genera_mapa2(matrix);
              // comprobar y eliminar secuencias iniciales
            }
            else if (falling)   //// SECCIÓN BAJADA DE ELEMENTOS ////
            {
              // bajar elementos (1 posición)
              // si bajando, activa_timer0(fall_init);
                             while (timer0_on) swiWaitForVBlank();
              // si final de bajada, comprobar y eliminar secuencias
            }
            else                //// SECCIÓN DE JUGADAS ////
            {
              // detectar jugada (touchscreen)
              // comprobar y eliminar secuencias
            }
            if (!falling)       //// SECCIÓN DE DEPURACIÓN ////
            {
              // detectar botones B/START para cambio manual de nivel
              // si se pulsa botón Y, if (timer3_on) desactiva_timer3();
                                      else activa_timer3();
            }
            if (change)         //// SECCIÓN CAMBIO DE NIVEL ////
            {
              // detectar fin movimientos o combinaciones o objetivos
              // si no hay combinación, recombina elementos
            }
            if (lapse >= 192)   //// SECCIÓN DE SUGERENCIAS ////
            {
              // si tiempo > 8 segundos, sugiere combinación
              // cada segundo, reduce_elementos(matrix);
                               aumenta_elementos(matrix);
            }
      } while (1);        // bucle infinito
}
```

Básicamente se han introducido las inicializaciones gráficas, la inicialización de las interrupciones, la activación del *timer* 0 (movimiento de *sprites*) cuando hay que bajar elementos, la activación y desactivación del *timer* 3 (desplazamiento del fondo 3) cuando se pulsa **Y**, y la invocación de las funciones de soporte `reduce_elementos()` y `aumenta_elementos()` que, a su vez, activarán el *timer* 1 (escalado de *sprites*) para el efecto de resaltado de los tres elementos de una combinación sugerida.

Por otro lado, el fichero `candy2_sopo.c` contiene el nuevo código de las funciones de soporte al programa principal. A continuación se describen las modificaciones respecto a la versión anterior:

La función `procesar_touchscreen()` se ha actualizado para adaptar la detección de la pulsación sobre las casillas del tablero al espacio total de la pantalla táctil, utilizando las definiciones `MTWIDTH` y `MTHEIGHT`:

```
int procesar_touchscreen(char mat[][COLUMNS],
                                 int *p1X, int *p1Y, int *p2X, int *p2Y)
{
      …
      scanKeys();
      if (keysHeld() & KEY_TOUCH)      // detecta pulsación en pantalla
      {
            touchRead(&posXY);         // capta posición (x,y), en píxeles
            v1X = posXY.px / MTWIDTH;  // convierte a posición matriz
            v1Y = posXY.py / MTHEIGHT;
      …
```

Las nuevas funciones `reduce_elementos()` y `aumenta_elementos()` activan el *timer* 1 para iniciar el escalado de los *sprites*, cuyas coordenadas (`col`, `fil`) se encuentran almacenadas en el vector de posiciones sugeridas `pos_sug[]`, y esperan el final del efecto consultando la variable global `timer1_on`:

```
void reduce_elementos(char mat[][COLUMNS])
{
      …
      for (i=0; i<3; i++)
      {
            x = pos_sug[i*2];
            y = pos_sug[i*2+1];
            activa_escalado(y, x);
            …
```

```
	}
	…
	activa_timer1(0);
	while (timer1_on) swiWaitForVBlank();
}

void aumenta_elementos(char mat[][COLUMNS])
{
	…
	activa_timer1(1);
	while (timer1_on) swiWaitForVBlank();
	for (i=0; i<3; i++)
	{
		x = pos_sug[i*2];
		y = pos_sug[i*2+1];
		desactiva_escalado(y, x);
		…
	}
	…
}
```

Por último, la función `intercambia_posiciones()` se ha modificado para realizar la animación de la posición de los *sprites* correspondientes a los elementos a intercambiar en una jugada, llamando a la función `activa_elemento()` y esperando el final del efecto consultado la variable global `timer0_on`:

```
void intercambia_posiciones(char mat[][COLUMNS],
                                    int p1X, int p1Y, int p2X, int p2Y)
{
	…
	activa_elemento(p1Y,p1X,p2Y,p2X);
	activa_elemento(p2Y,p2X,p1Y,p1X);
	activa_timer0(1);
	while (timer0_on) swiWaitForVBlank();
}
```

2.4 Rutinas para la gestión de los sprites

El fichero `Sprites_sopo.s` contiene una serie de rutinas en lenguaje ensamblador creadas específicamente para manejar los *sprites* de la NDS. El fichero `Sprites_sopo.h` declara las cabeceras de estas rutinas para su invocación desde código fuente escrito en lenguaje C:

```
/*--------------------------------------------------------------------

     $Id: Sprites_sopo.h $

     Declaraciones de funciones globales de 'Sprites_sopo.s'

--------------------------------------------------------------------*/

extern void SPR_actualizarSprites(u16* base, int limite);
extern void SPR_crearSprite(int indice, int forma, int tam, int baldosa);
extern void SPR_mostrarSprite(int indice);
extern void SPR_ocultarSprite(int indice);
extern void SPR_ocultarSprites(int limite);
extern void SPR_moverSprite(int indice, int px, int py);
extern void SPR_fijarPrioridad(int indice, int prioridad);
extern void SPR_activarRotacionEscalado(int indice, int grupo);
extern void SPR_desactivarRotacionEscalado(int indice);
extern void SPR_fijarEscalado(int igrp, short sx, short sy);
```

Todas las rutinas, excepto la primera, guardan el resultado de sus cálculos sobre una variable global que se llama `oam_data`, declarada al principio del fichero `Sprites_sopo.s`:

```
@;-- .bss. data section ---
.bss
          .align 1
     oam_data:  .space 128 * 8     @; espacio de trabajo para 128 sprites
```

El propósito de las rutinas `SPR_...` es guardar los parámetros en los campos de los registros de control del *sprite* referenciado por su índice (número entre 0 y 127). Sin embargo, estos cambios se almacenan temporalmente sobre la variable `oam_data`, que replica la estructura de los registros de control de los *sprites*, ubicados a partir de la dirección de memoria `0x07000000` para el procesador gráfico principal. La rutina

`SPR_actualizarSprites()` copia el contenido de la variable `oam_data` sobre las correspondientes posiciones de memoria de dichos registros de control.

Este mecanismo se ha diseñado para permitir que la escritura de los registros de control de los *sprites* (registros E/S) se produzca justo después de un retroceso vertical (*Vertical Blank*), con el fin de realizar un acceso sincronizado. De este modo se evitará modificar los registros de E/S cuando se estén accediendo por parte del procesador gráfico principal, aunque las rutinas de gestión de *sprites* proporcionadas (excepto la primera) se invoquen en cualquier momento (acceso des-sincronizado). El resultado final es que se pueden realizar múltiples actualizaciones de los *sprites* que serán transmitidas al procesador gráfico en el momento adecuado, sin necesidad de sincronizar cada actualización individualmente.

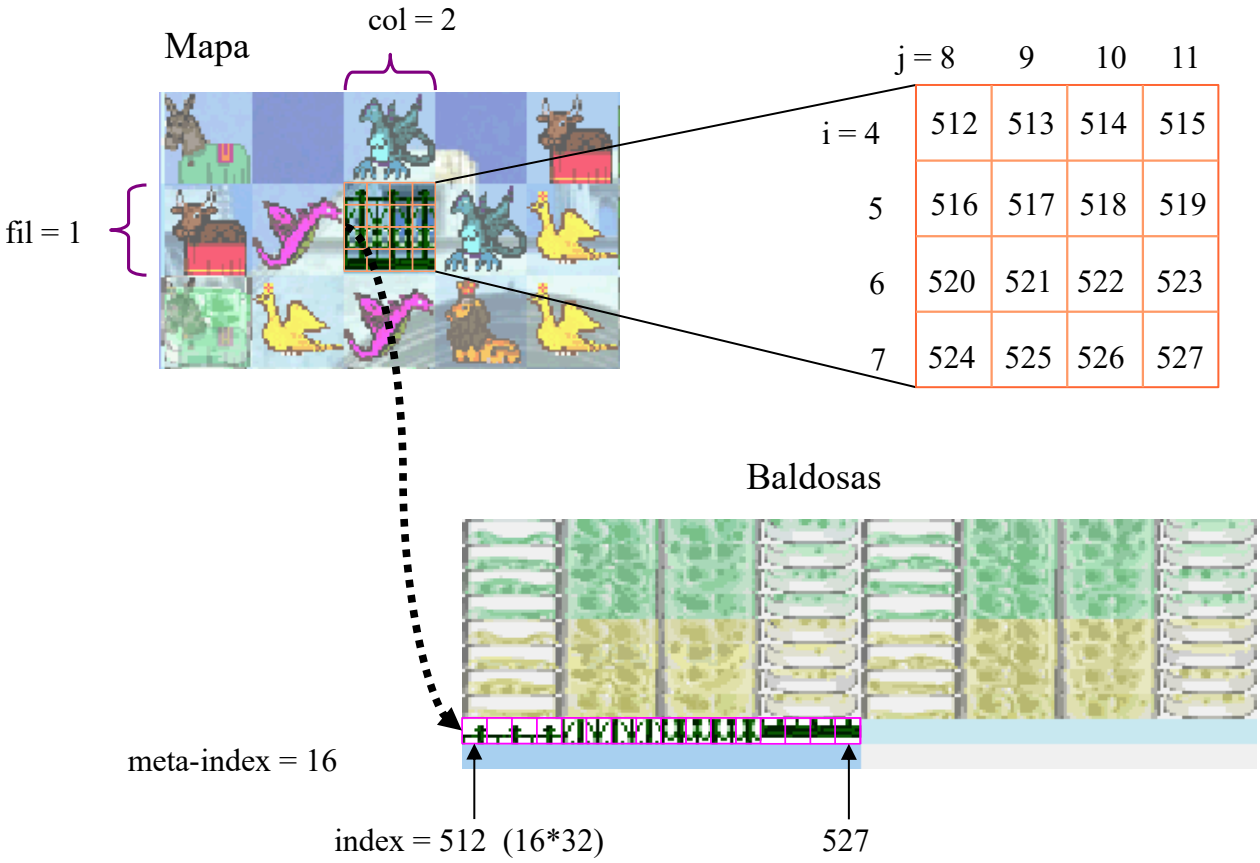
2.5 Rutinas de soporte a las tareas de la práctica

El fichero `candy2_supo.s` contiene una serie de rutinas en lenguaje ensamblador para ayudar a la implementación de las tareas de la práctica. El fichero `candy2_incl.h` declara las cabeceras de estas rutinas para su invocación desde código escrito en lenguaje C:

```
     // candy2_supo.s //
extern int busca_elemento(int fil, int col);
extern int crea_elemento(int tipo, int fil, int col);
extern int elimina_elemento(int fil, int col);
extern int activa_elemento(int fil, int col, int f2, int c2);
extern int activa_escalado(int fil, int col);
extern int desactiva_escalado(int fil, int col);
extern void fija_metabaldosa(u16 * mapaddr, int fil, int col, int imeta);
extern void elimina_gelatina(u16 * mapaddr, int fil, int col);
```

En las descripciones de las rutinas, dentro del fichero `candy2_supo.s`, se describe el funcionamiento de cada una de ellas. No obstante, en este apartado vamos a explicar algunos conceptos clave para entender cómo operan.

Empezaremos por la rutina `fija_metabaldosa()`, que permite fijar los índices de las baldosas de una metabaldosa dentro de las posiciones de un mapa de baldosas correspondientes a una casilla del tablero de juego. El siguiente gráfico ilustra el proceso de fijar las posiciones de la casilla (1, 2) a los índices de las baldosas de la metabaldosa 16:

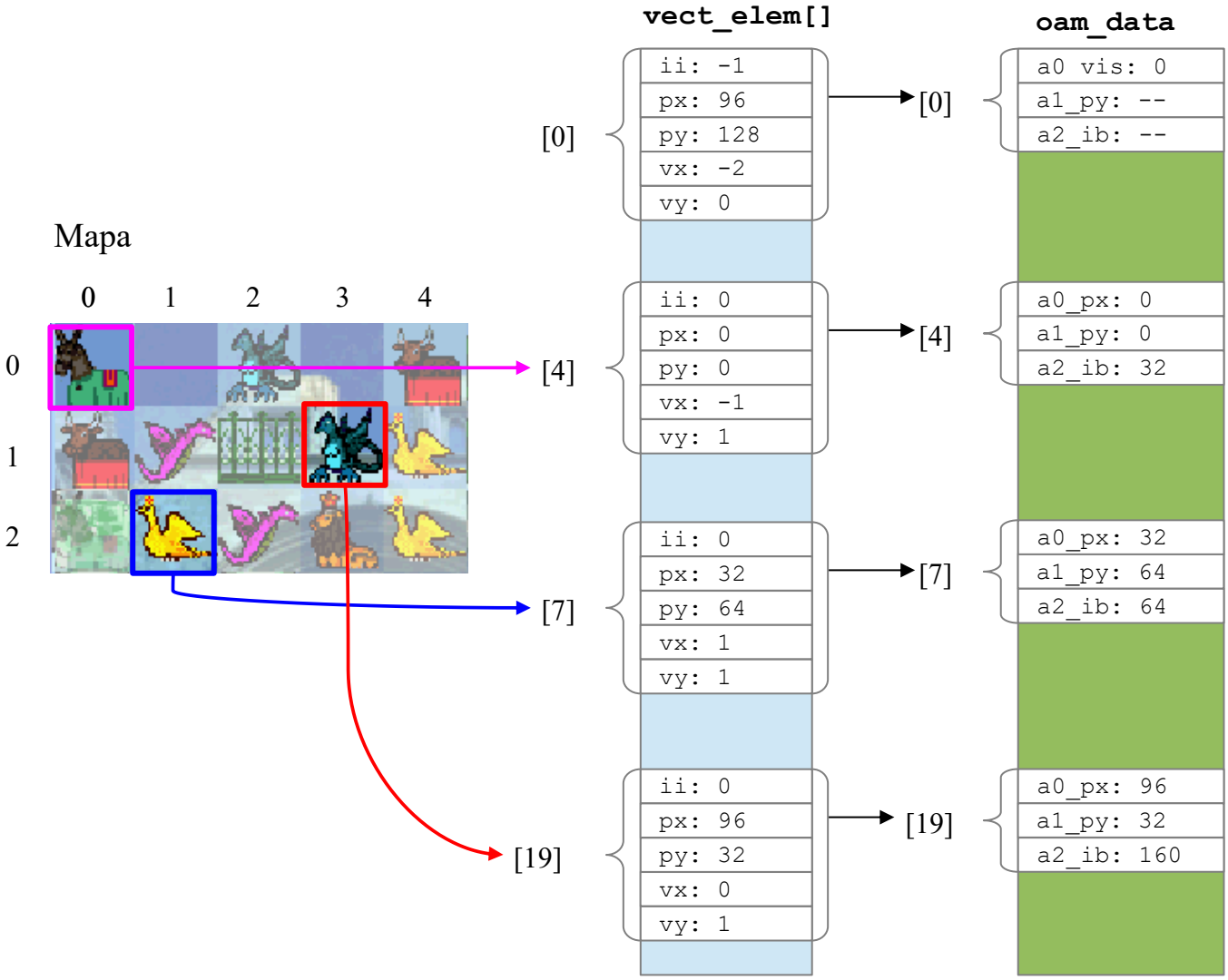


El resultado de fijar la metabaldosa 16 en la casilla (1,2) es guardar los índices del 512 al 527 en las posiciones del mapa de baldosas comprendidas entre las filas i=4..7 y las columnas j=8..11. Aunque la rutina `fija_metabaldosa()` realiza todos los cálculos necesarios para modificar el mapa de baldosas, resulta útil comprender su funcionamiento para la implementación de las tareas de crear el fondo ajedrezado (con huecos), generar los bloques sólidos (verja) y realizar la animación de las gelatinas. Además, la rutina espera como primer parámetro la dirección inicial del mapa de baldosas que se tiene que modificar, o sea que debemos ser conscientes de dónde está almacenada toda la información.

Por otro lado, se proporcionan las rutinas de soporte que operan con los elementos, `busca_elemento()`, `crea_elemento()`, `elimina_elemento()` y `activa_elemento()`, que trabajan con la información almacenada en el vector `vect_elem[]`, el cuál registra la posición y la velocidad de los elementos activos del juego. Cada entrada del vector sigue la estructura `elemento`, definida en el fichero

`candy2_incl.h` (ver apartado 2.1). El campo `ii` sirve para determinar si dicha entrada corresponde a un elemento activo (>=0) o no (-1).

La siguiente figura muestra un ejemplo de cómo se almacena la información de 3 elementos:

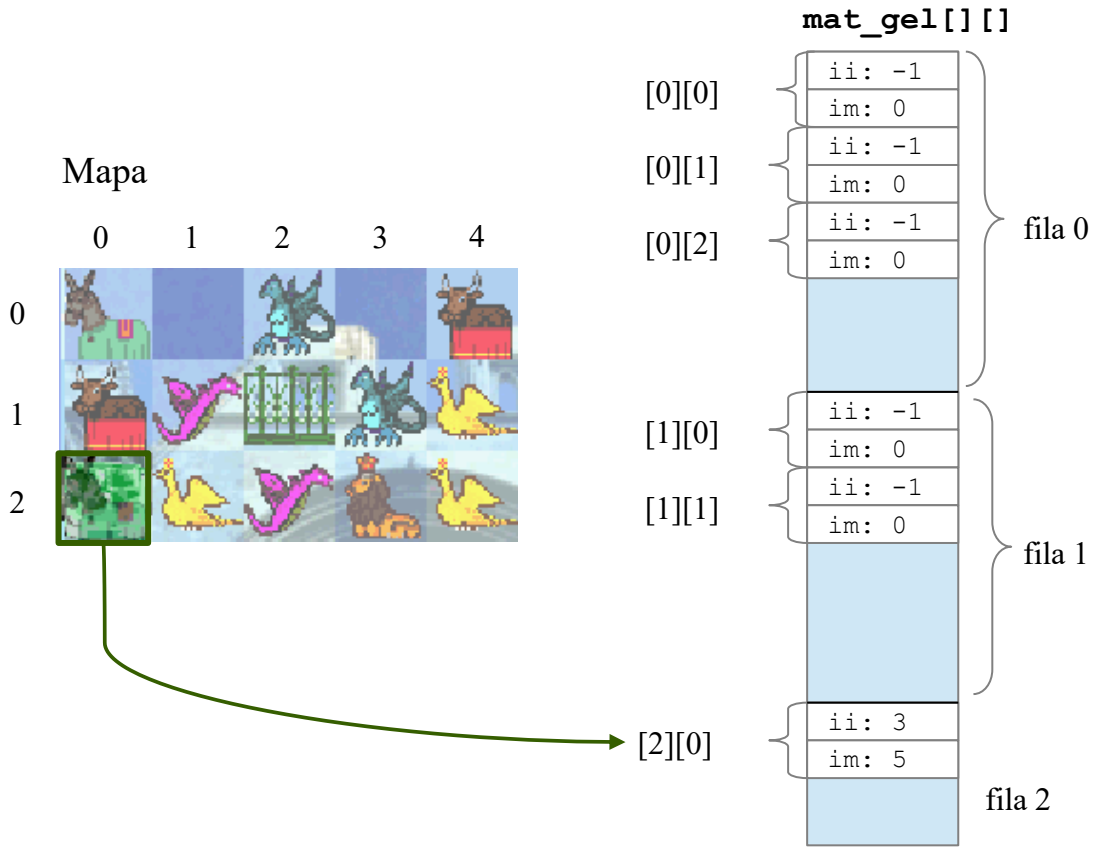


Salvo al inicio de la partida, los elementos del juego se encuentran **desordenados** dentro del vector `vect_elem[]`, puesto que cuando se eliminan elementos sus entradas en el vector se reutilizan para crear nuevos elementos, cuya posición no coincidirá con la de los elementos eliminados. La rutina `busca_elemento()` devuelve el índice de la entrada del vector `vect_elem[]` correspondiente al elemento que se encuentra en la posición `(fil, col)` que se pasa por parámetro, buscando las coordenadas equivalentes en píxeles almacenadas en los campos `(px, py)` de las entradas activas del vector (entradas con el campo `ii` > -1).

Por otro lado, las rutinas `crea_elemento()`, `elimina_elemento()` y `activa_elemento()`, aparte de modificar el contenido del vector `vect_elem[]`, invocan a las rutinas de gestión de *sprites* `SPR_...` para mostrar, ocultar, mover y cambiar el dibujo de los *sprites* que representan gráficamente a los elementos del juego.

Una vez más, aunque estas rutinas realizan todo el trabajo automáticamente, es necesario saber cómo se estructura la información de los elementos, además de comprender que el índice de cada elemento dentro del vector coincide con el índice de los atributos del *sprite* dentro de los datos `oam_data`. Concretamente, será necesario recorrer el vector de elementos para mover los elementos activos invocando a la rutina `SPR_moverSprite()`, además de la correspondiente llamada a `SPR_actualizarSprites()`.

Vamos a dejar la explicación de las rutinas `activa_escalado()` y `desactiva_escalado()` para cuando se detallen las tareas de resaltado de elementos, y vamos a terminar este apartado explicando cómo se representan las gelatinas simples y dobles con la matriz `mat_gel[][COLUMNS]`, la cual se modifica por la rutina `elimina_gelatina()`. El siguiente gráfico muestra un ejemplo de representación de una gelatina simple:



Como se puede observar, la matriz de gelatinas sí que está **ordenada** del mismo modo que la matriz de juego, pudiendo acceder a sus entradas a partir de los índices de fila y columna de las casillas del tablero.

Si en una casilla no hay gelatina, el campo `ii` de la entrada asociada será -1. Si hay una gelatina simple o doble, el campo `ii` será un número entre 0 i 10, que se usará para retardar la animación de sus baldosas.

El campo `im` indica el índice de metabaldosa que se está visualizando en el momento actual, que será un número de 0 a 7 para una gelatina simple, o un número de 8 a 15 para una gelatina doble. La animación de las gelatinas consiste en incrementar cíclicamente el índice de su metabaldosa y actualizar las baldosas correspondientes.

La rutina `elimina_gelatina()` desactiva la entrada de la matriz `mat_gel[][COLUMNS]` correspondiente a la gelatina que se encuentra en la posición `(fil, col)` que se especifica por parámetro, además de eliminar las baldosas correspondientes del mapa de baldosas, cuya dirección de memoria virtual se debe indicar también por parámetro. El resto de acciones relativas a los gráficos de gelatinas (inicialización y animación) corresponde a tareas que se deben asignar a un miembro del grupo de prácticas, de modo que le será necesario conocer esta representación de la información sobre las gelatinas.

3 Tareas de inicialización gráfica

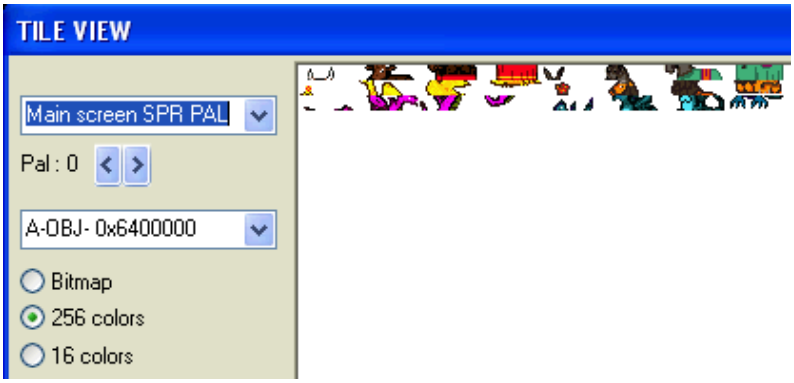
3.1 Tarea 2A: inicialización de los sprites

3.1.1 Subtarea 2Aa: init_grafA()

Implementar la parte de la función de inicialización del procesador gráfico principal `init_grafA()` encargada de realizar las siguientes acciones:

- reservar la memoria gráfica para las baldosas de los *sprites*, asignando el banco de memoria **VRAM_F** a partir de la dirección `0x06400000`,
- cargar las baldosas para los *sprites*, almacenadas en la variable global `SpritesTiles[]`, y su paleta de colores asociada, almacenada en la variable global `SpritesPal[]`, ambas definidas dentro de `graphics/Sprites.s`.

Después de dichas inicializaciones, se tienen que poder observar las baldosas con la herramienta **View Tiles** del ***DeSmuME***:



3.1.2 Subtarea 2Ab: genera_sprites(char mat[][COLUMNS])

Inicializar los *sprites* correspondientes a los elementos de la matriz de juego que se pasa por parámetro, realizando los siguientes pasos:

- ocultar todos los 128 *sprites* y desactivar todos los `ROWS*COLUMNS` elementos del vector `vect_elem[]`,
- recorrer la matriz del juego e invocar a la función `crea_elemento()` para todas las casillas donde exista un elemento (con o sin gelatina),

- actualizar la variable global `n_sprites`, de acuerdo con el número de *sprites* creados.
- actualizar OAM según el número de *sprites* creados.

Después de dichas inicializaciones, al cargar un nivel de juego se tiene que poder observar los *sprites* con la herramienta **View OAM** del ***DeSmuME***:

matrix[6][8]:

1	15	5	15	6	7	2	15
4	3	7	6	6	7	5	2
10	3	13	1	1	14	3	3
10	1	9	3	2	20	3	4
17	2	15	15	3	19	4	3
3	2	10	6	5	20	1	15

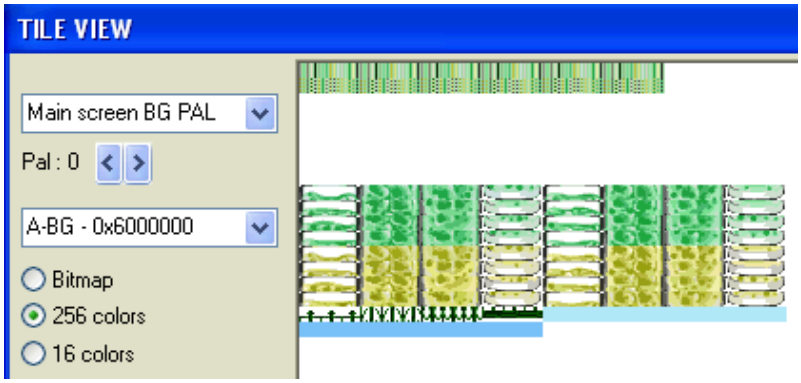
3.2 Tarea 2B: inicialización del fondo 2 (patrón ajedrezado)

3.2.1 Subtarea 2Ba: init_grafA()

Implementar la parte de la función de inicialización del procesador gráfico principal `init_grafA()` encargada de realizar las siguientes acciones:

- reservar la memoria gráfica para los mapas y las baldosas de los fondos 1 y 2, asignando el banco de memoria **VRAM_E** a partir de la dirección `0x06000000`,
- inicializar el fondo 2 en modo ***Text*** (8bpp), con un tamaño del mapa de 32x32 baldosas, fijando la base de los gráficos de las baldosas y del mapa de baldosas donde se considere oportuno (pero sin colisiones con otros programadores/as),
- fijar la prioridad del fondo 2 al nivel 2,
- cargar las baldosas para los fondos 1 y 2, almacenadas (en formato comprimido LZ77) en la variable global `BaldosasTiles[]`, y su paleta de colores asociada, almacenada en la variable global `BaldosasPal[]`, ambas definidas dentro de `graphics/Baldosas.s`.

Después de dichas inicializaciones, se tienen que poder observar las baldosas con la herramienta **View Tiles** del ***DeSmuME***:



3.2.2 Subtarea 2Bb: genera_mapa2(char mat[][COLUMNS])

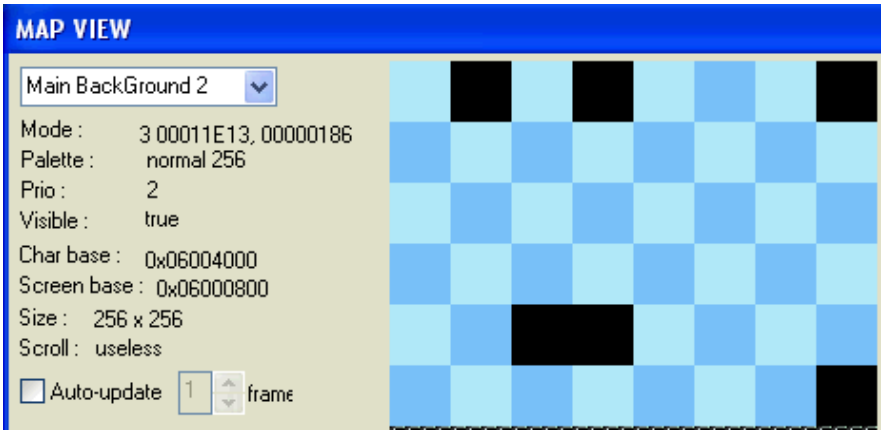
Generar un mapa de baldosas sobre el fondo 2 que represente un patrón de cuadros ajedrezado con las posiciones del tablero de juego que se pasa por parámetro, marcando las posiciones donde existen elementos (con o sin gelatina) o bloques sólidos, dejando cuadros transparentes en las casillas que contengan un hueco, realizando los siguientes pasos:

- recorrer la matriz del juego e invocar la función `fija_metabaldosa()` para fijar una metabaldosa en cada posición de la pantalla de acuerdo con el contenido del tablero de juego:
    - donde exista un elemento (con o sin gelatina), un espacio vacío o un bloque sólido, copiar una metabaldosa sólida de color azul claro o azul intenso (índice 17 o 18), de forma alternada para formar un patrón ajedrezado,
    - donde exista un hueco, copiar una metabaldosa transparente (índice 19), de forma que se pueda observar el fondo general del juego (fondo 3) a través de los huecos del tablero.

Después de dichas inicializaciones, al cargar un nivel de juego se tiene que poder observar el contenido del mapa 2 con la herramienta **View Map** del ***DeSmuME***:

matrix[6][8]:

1	15	5	15	6	7	2	15
4	3	7	6	6	7	5	2
10	3	13	1	1	14	3	3
10	1	9	3	2	20	3	4
17	2	15	15	3	19	4	3
3	2	10	6	5	20	1	15



3.3 Tarea 2C: inicialización del fondo 1 (gelatinas)

3.3.1 Subtarea 2Ca: init_grafA()

Implementar la parte de la función de inicialización del procesador gráfico principal `init_grafA()` encargada de realizar las siguientes acciones:

- reservar la memoria gráfica para los mapas y las baldosas de los fondos 1 y 2, (mismo código que la subtarea **2Ba**),
- inicializar el fondo 1 en modo ***Text*** (8bpp), con un tamaño del mapa de 32x32 baldosas, fijando la base de los gráficos de las baldosas y del mapa de baldosas donde se considere oportuno (pero sin colisiones con otros programadores/as),
- fijar la prioridad del fondo 1 al nivel 0,
- cargar las baldosas para los fondos 1 y 2 (mismo código que la subtarea **2Ba**).

Después de dichas inicializaciones, se tienen que poder observar las mismas baldosas que las mostradas en el apartado 3.2.1.

3.3.2 Subtarea 2Cb: genera_mapa1(char mat[][COLUMNS])

Generar un mapa de baldosas sobre el fondo 1 que represente inicialmente los bloques sólidos y las gelatinas simples y dobles del tablero de juego que se pasa por parámetro, realizando los siguientes pasos:

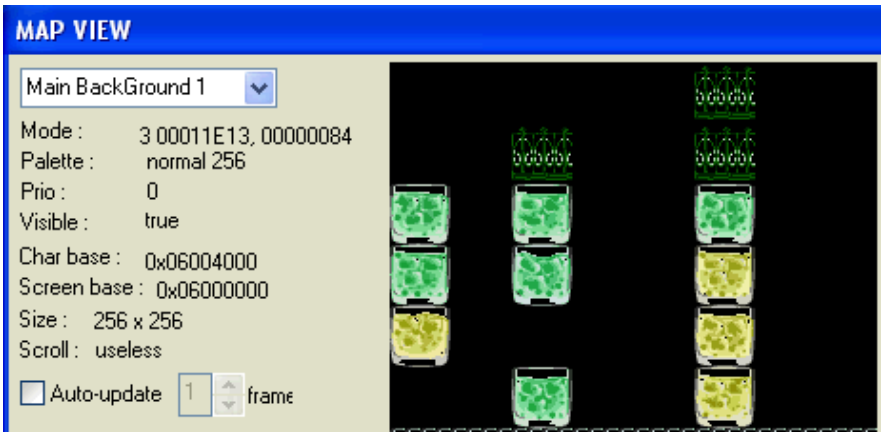
- recorrer la matriz del juego e invocar la función `fija_metabaldosa()` para fijar una metabaldosa en cada posición de la pantalla de acuerdo con el contenido del tablero de juego:
    - donde no exista bloque sólido ni gelatina, copiar una metabaldosa transparente (índice 19),
    - donde exista un bloque sólido, copiar la metabaldosa que representa una verja (índice 16),
    - donde exista una gelatina simple o doble (con o sin elemento), generar un valor aleatorio entre 0 y 7 para calcular una metabaldosa de animación aleatoria, según sea simple o doble, y copiarla en el mapa,

- para las casillas con gelatina, activar la posición correspondiente de la matriz de gelatinas `mat_gel[][COLUMNS]`, con un índice de animación (campo `ii`) aleatorio entre 1 y 10; además, hay que almacenar el índice de la metabaldosa actual en el campo `im`; el resto de las posiciones deben estar desactivadas (campo `ii` a -1).

Después de dichas inicializaciones, al cargar un nivel de juego se tiene que poder observar el contenido del mapa 1 con la herramienta **View Map** del ***DeSmuME***:

matrix[6][8]:

1	15	5	15	6	7	2	15
4	3	7	6	6	7	5	2
10	3	13	1	1	14	3	3
10	1	9	3	2	20	3	4
17	2	15	15	3	19	4	3
3	2	10	6	5	20	1	15



3.4 Tarea 2D: inicialización del fondo 3 (imagen de fondo)

3.4.1 Subtarea 2Da: init_grafA()

Implementar la parte de la función de inicialización del procesador gráfico principal `init_grafA()` encargada de realizar las siguientes acciones:

- reservar la memoria gráfica para los píxeles del fondo 3, asignando los bancos de memoria **VRAM_A** y **VRAM_B** a partir de la dirección `0x06020000`,
- inicializar el fondo 3 en modo ***Extended/bitmap 16bpp***, con un tamaño de la imagen de 512 columnas por 256 filas, fijando la base de los píxeles a partir de la dirección `0x06020000`,
- fijar la prioridad del fondo 3 al nivel 3,
- cargar los píxeles de la imagen para el fondo 3, almacenados (en formato comprimido LZ77) en la variable global `FondoBitmap[]`, definida dentro del fichero `graphics/Fondo.s`,
- invocar a la función `ajusta_imagen3()` para rotar y alinear la imagen convenientemente, tal como se describe en la siguiente subtarea.

Después de dichas inicializaciones, se tienen que poder observar los píxeles del fondo 3 con la herramienta **View Map** del ***DeSmuME***:

3.4.2 Subtarea 2Db: ajusta_imagen3(int ibg)

Rotar 90 grados a la derecha la imagen del fondo cuyo identificador se pasa por parámetro (fondo 3 de procesador principal) y desplazarla para que se visualice en vertical a partir del primer píxel, realizando los siguientes pasos:

- fijar el centro de la imagen como centro de rotación, invocando la función de ***libnds*** `bgSetCenter()`,
- rotar el fondo 90 grados a la derecha, invocando la función `bgSetRotate()` para la rotación y la función `degreesToAngle()` para convertir los grados en el formato propio del parámetro de rotación de `bgSetRotate()`,
- desplazar el fondo para que coincida la parte alta de la pantalla con la parte alta de la imagen, invocando la función `bgSetScroll()`,
- actualizar los parámetros de rotación-desplazamiento fijados para el fondo 3, invocando la función `bgUpdate()`.

Después de dichas inicializaciones, al iniciar el programa se tiene que poder observar la parte alta de la imagen en la pantalla inferior:

4 Tareas de animaciones gráficas

4.1 Tarea 2E: movimiento de elementos

4.1.1 Subtarea 2Ea: rsi_vblank()

Implementar la parte de la RSI del retroceso vertical `rsi_vblank()` que debe actualizar los *sprites*, llamando a la rutina `SPR_actualizarSprites()` cuando se haya producido un cambio en la posición o forma de alguno de los *sprites* del juego, lo cual se podrá detectar a través de la variable global `update_spr` declarada dentro del fichero `RSI_timer0.s`.

Después de dicha actualización, habrá que desactivar (poner a cero) la variable `update_spr`.

4.1.2 Subtarea 2Eb: activa_timer0(int init)

Rutina para inicializar el *timer* 0, cargando o no el valor del divisor de frecuencia del *timer* según el parámetro `init`:

- si `init` no es 0, el valor de la variable `divFreq0` se copiará en otra variable de nombre `divF0`, además de en el registro E/S de datos del *timer* 0,
- si `init` es 0, no se modificará de ningún modo la variable `divF0` ni el registro E/S de datos del *timer* 0, ya que se debe mantener el divisor de frecuencia anterior en dichas entidades (variable y registro de E/S).

El propósito de dicho comportamiento es permitir simular un efecto de aceleración del movimiento de los elementos, ya que el divisor de frecuencia irá disminuyendo a medida que se muevan los *sprites* correspondientes, lo cual permitirá generar más interrupciones por unidad de tiempo y, por lo tanto, más velocidad.

El contenido de la variable `divFreq0` se tiene que calcular para conseguir mover un elemento de una casilla a otra en menos de 0,35 segundos. Ésta será la velocidad inicial, que se irá incrementando a medida que los elementos se muevan, hasta que se paren todo el movimiento.

Además, hay que activar (poner a 1) la variable global `timer0_on`, la cual se pondrá a cero cuando se hayan terminado todos los desplazamientos de los *sprites* (ver apartado 4.1.4).

Por último, la rutina `activa_timer0()` debe activar dicho *timer* mediante sus registro E/S de datos y control, seleccionando la frecuencia de entrada adecuada y permitiendo la activación de las interrupciones asociadas.

4.1.3 Subtarea 2Ec: desactiva_timer0()

Rutina para desactivar el *timer* 0 a través de su registro E/S de control, además de desactivar (poner a 0) la variable global `timer0_on`.

4.1.4 Subtarea 2Ed: rsi_timer0()

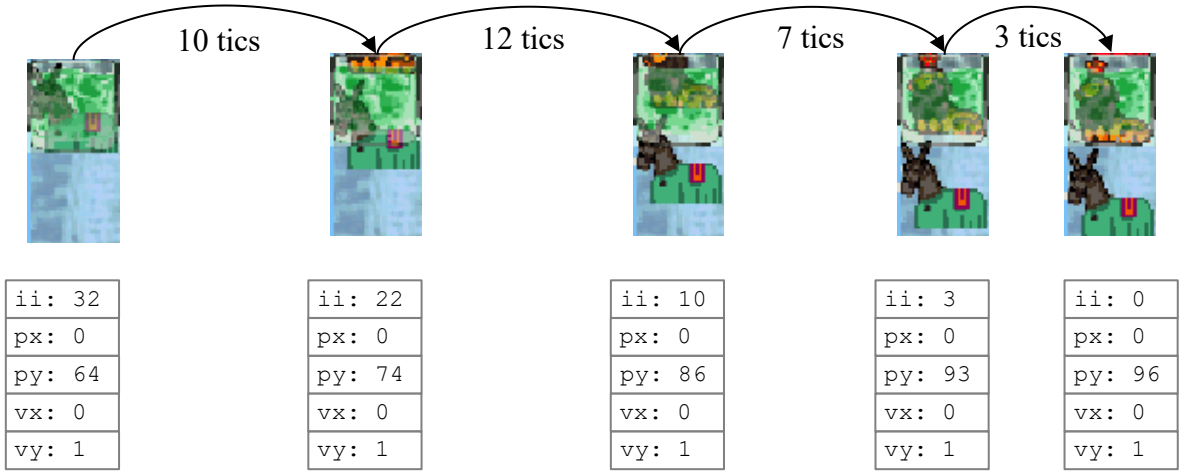
Rutina para atender las interrupciones periódicas del *timer* 0. Su propósito básico es, a cada interrupción de tiempo (tic), recorrer los `n_sprites` iniciales del vector de elementos `vect_elem[]` y actualizar la posición de los *sprites* activos, a partir de los siguientes pasos:

- si el campo `ii` del elemento analizado está desactivado (-1) o vale 0, ignorar ese elemento,
- en otro caso, decrementar el valor del campo `ii` y actualizar los campos de posición `px` y `py` según el valor de los campos de velocidad `vx` y `vy`,
- actualizar la posición del *sprite* correspondiente al elemento analizado, llamando a la rutina `SPR_moverSprite()`,
- si se ha movido algún elemento, activar la variable `update_spr` y reducir el divisor de frecuencia del *timer* para provocar el efecto de aceleración, evitando superar cierto límite de velocidad máxima (a fijar según criterio del programador),
- si no se ha movido ningún elemento, invocar a `desactivar_timer0()`, lo cual provocará la puesta a cero de la variable `timer0_on`.

La comunicación de las rutinas que activan el movimiento de los elementos (tarea **2I**) con las rutinas de la tarea **2E** se realiza mediante el vector de elementos, más

concretamente, a través a la rutina de soporte `activa_elemento()`. Dicha rutina calcula los valores de los campos `vx` y `vy` según las casillas origen y destino del movimiento, y asigna el valor 32 al campo `ii` del elemento a mover. De este modo, si hay que bajar una casilla en vertical, se fijarán `vx` = 0 y `vy` = 1, de modo que, después de 32 interrupciones, la coordenada `py` se habrá incrementado 32 píxeles, y el elemento habrá llegado a la casilla inferior.

Los siguientes gráficos muestran algunos de los 32 desplazamientos (correspondientes a 32 tics del *timer* 0) de un mismo movimiento vertical:



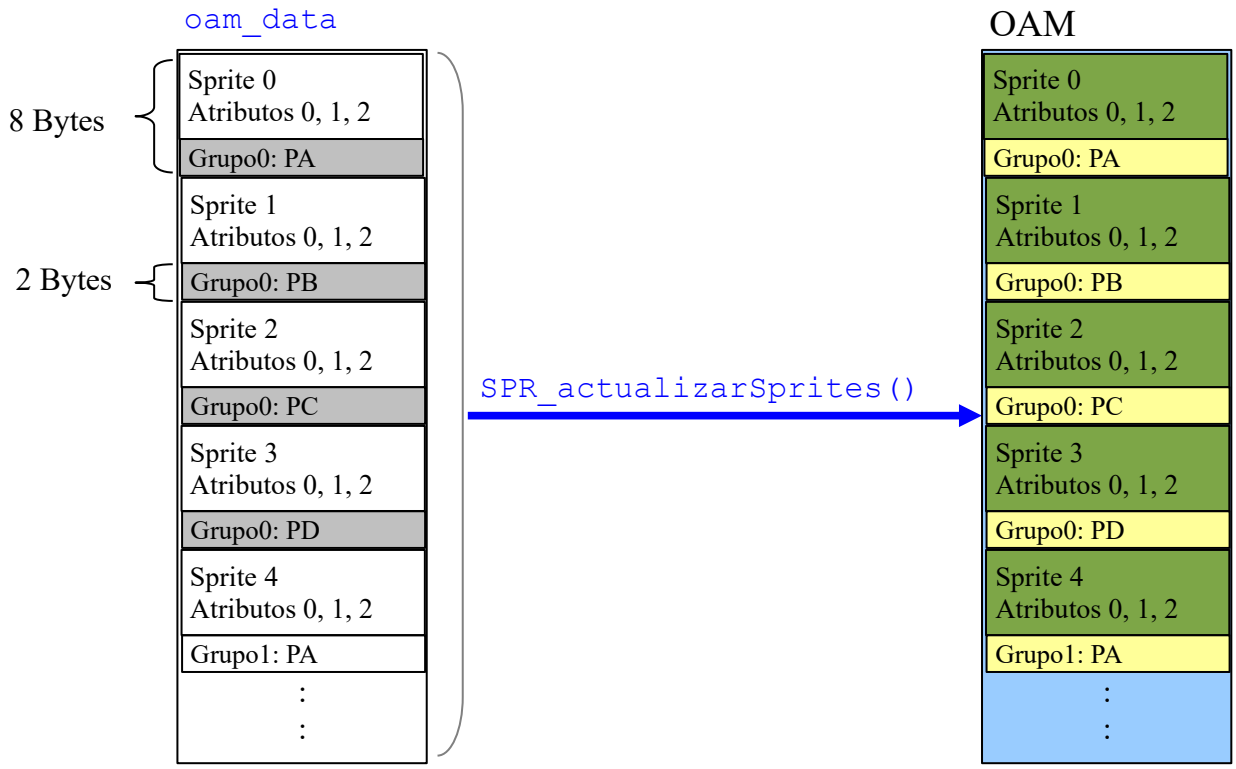
4.2 Tarea 2F: escalado de elementos

4.2.1 Subtarea 2Fa: rsi_vblank()

Esta subtarea no existe en la lista de subtareas puntuables, ya que su implementación es idéntica a la de la subtarea **2Ea** (apartado 4.1.1), por lo tanto, solo se puntuará una vez. Sin embargo, hay que entender por qué el mismo código permite actualizar la visualización de la posición de los *sprites* en movimiento, así como la visualización del escalado de los *sprites* correspondientes a los elementos de una sugerencia de combinación.

En primer lugar, hay que entender que la parte **2Ea** de la `rsi_vblank()`, implementada dentro del fichero `RSI_timer0.s`, llama a la rutina `SPR_actualizarSprites()` cuando la variable global `update_spr` vale 1. Dicha actualización provoca la copia del contenido de la variable `oam_data` dentro de los registros E/S de control OAM de los *sprites* activos en ese momento.

En segundo lugar, el programa principal llamará a la rutina de soporte `activa_escalado()`, implementada dentro del fichero `candy2_supo.s`, para las tres posiciones del tablero de juego retornadas por la rutina `sugiere_combinacion()`. Esto provocará que los *sprites* de dichas casillas se activen en modo rotación-escalado, usando el **grupo 0 de registros PA-PD**. Tal como se explica en los apuntes de teoría del tema 4, estos registros E/S de control contienen los parámetros de rotación-escalado de dicho grupo, y se encuentran intercalados entre los atributos de los primeros *sprites* de las estructuras `oam_data` y OAM:



En resumen, para escalar los elementos sugeridos habrá que modificar las posiciones de la variable `oam_data` correspondientes a los parámetros PA-PD del grupo 0, y poner a 1 la variable `update_spr`.

La rutina `rsi_vblank()` llamará a `SPR_actualizarSrites()`, la cual copiará los parámetros PA-PD del grupo 0 en OAM si el número de *sprites* activos es mayor o igual a 4, ya que hay que copiar 4 posiciones correspondientes a los parámetros del primer grupo de rotación-escalado. La estructura del juego garantiza que siempre existirán, como mínimo, 4 *sprites* activos.

4.2.2 Subtarea 2Fb: activa_timer1(int init)

Rutina para inicializar el *timer* 1, además de copiar el parámetro `init` dentro de la variable `escSen` (declarada dentro del fichero `RSI_timer1.s`), la cual indica el sentido del escalado:

- si `escSen` es 0, se trata de un ciclo de decremento del escalado; hay que fijar la variable `escFac`, que guardará el factor de escalado actual, a 1,0 (en formato decimal de coma fija 0.8.8), y trasladar dicho factor sobre los parámetros PA y PD del grupo 0 (PB y PC se mantendrán a cero), lo cual se puede conseguir mediante una llamada a la función `SPR_fijarEscalado()`,

- si `escSen` es 1, se trata de un ciclo de incremento del escalado; no hay que modificar la variable `escFac` ni fijar los parámetros PA y PD del grupo 0, porque se supone que contienen el último valor del ciclo de decremento.

En cualquiera de los dos casos, hay que fijar la variable `escNum` a 0, ya que será el contador de las veces que se ha cambiado el factor de escalado. Dichos cambios se producirán en la RSI del *timer* 1 (ver apartado 4.2.4).

Además, hay que activar (poner a 1) la variable global `timer1_on`, la cual se pondrá a cero cuando se hayan terminado todas las variaciones de escala de un ciclo de reducción o de aumento (ver apartado 4.2.4).

También hay que inicializar el divisor de frecuencia del *timer* 1, copiando en su registro E/S de datos el contenido de la variable `divFreq1`, que se tiene que calcular para conseguir 32 tics en menos de 0,35 segundos.

Por último, la rutina `activa_timer1()` debe activar dicho *timer* mediante su registro E/S de control, seleccionando la frecuencia de entrada adecuada y permitiendo la activación de las interrupciones asociadas.

4.2.3 Subtarea 2Fc: desactiva_timer1()

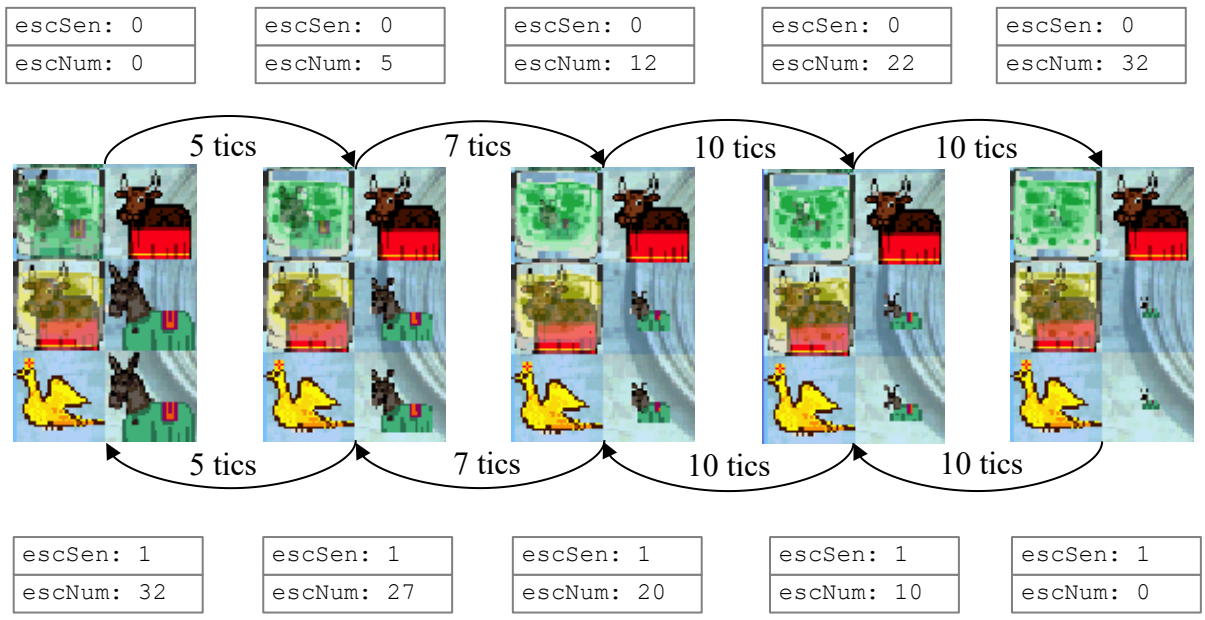
Rutina para desactivar el *timer* 1 a través de su registro E/S de control, además de desactivar (poner a 0) la variable global `timer1_on`.

4.2.4 Subtarea 2Fd: rsi_timer1()

Rutina para atender las interrupciones periódicas del *timer* 1. Su propósito básico es, a cada interrupción de tiempo (tic), actualizar el escalado de los *sprites* correspondientes a los elementos sugeridos a partir de los siguientes pasos:

- incrementar la variable `escNum`; si dicha variable llega a 32, invocar a `desactivar_timer1()`, lo cual provocará la puesta a cero de la variable `timer1_on`.
- en otro caso, incrementar o decrementar (según el sentido de escalado que indica `escSen`) el factor de escalado actual almacenado en `escFac`, en un valor diferencial fijo, a determinar por el programador,
- actualizar el factor de escalado actual sobre los parámetros PA y PD del grupo 0, llamando a la rutina `SPR_fijarEscalado()`,
- activar la variable `update_spr`, lo cual provocará (en la próxima interrupción de *VBlank*) la actualización de los registros E/S de los *sprites* en OAM.

Los siguientes gráficos muestran algunos de los 32 cambios de escalado (correspondientes a 32 tics del *timer* 1) de un efecto de reducción (hacia la derecha) seguido por un efecto de aumento (hacia la izquierda):



4.3 Tarea 2G: animación de gelatinas

4.3.1 Subtarea 2Ga: rsi_vblank()

Implementar la parte de la RSI del retroceso vertical (en el fichero `RSI_timer0.s`) encargada de actualizar las metabaldosas de las gelatinas que se tengan que cambiar, a partir de los siguientes pasos:

- si la variable `update_gel` está desactivada (igual a 0), ignorar todos los pasos siguientes,
- en caso contrario, recorrer toda la matriz de gelatinas `mat_gel[][COLUMNS]` y, para cada posición del tablero de juego, aplicar los siguientes pasos:
    - si el campo `ii` de la posición analizada está desactivado (-1) o es mayor que 0, ignorar dicha posición,
    - en caso contrario, actualizar la metabaldosa de la gelatina correspondiente invocando la rutina `fija_metabaldosa()` con el valor de metabaldosa que indique el campo `im` de la posición analizada, y reinicializar el campo `ii` a 10.
- finalmente, desactivar la variable `update_gel`.

Este proceso permite actualizar las metabaldosas de las gelatinas en instantes diferentes, suponiendo que los valores del campo `ii` de las gelatinas presentan divergencias entre sí (ver apartado 3.3.2), ya que no todos estos valores serán 0 en la misma interrupción de retroceso vertical. Esto evitará sobrecargar la `rsi_vblank()`, lo cual podría suceder si todas las gelatinas del tablero se tuvieran que actualizar simultáneamente.

Por otro lado, la RSI del *timer* 2 es la encargada de actualizar los valores de los campos `ii` e `im` de la matriz de gelatinas, a la frecuencia que se indique para el *timer* 2 (ver apartado 4.3.4).

4.3.2 Subtarea 2Gb: activa_timer2()

Rutina para inicializar el *timer* 2, además de poner a 1 la variable global `timer2_on`. Esta rutina se llama desde la función `contar_gelatinas()` del fichero `candy2_sopo.c`, en el caso de que la matriz de juego contenga alguna gelatina.

También hay que inicializar el divisor de frecuencia del *timer* 2, copiando en su registro E/S de datos el contenido de la variable `divFreq2`, que se tiene que calcular para conseguir 10 cambios de metabaldosa por segundo.

Por último, la rutina `activa_timer2()` debe activar dicho *timer* mediante su registro E/S de control, seleccionando la frecuencia de entrada adecuada y permitiendo la activación de las interrupciones asociadas.

4.3.3 Subtarea 2Gc: desactiva_timer2()

Rutina para desactivar el *timer* 2 a través de su registro E/S de control, además de desactivar (poner a 0) la variable global `timer2_on`. Esta rutina se llama desde la función `contar_gelatinas()` del fichero `candy2_sopo.c`, en el caso de que la matriz de juego no contenga ninguna gelatina.

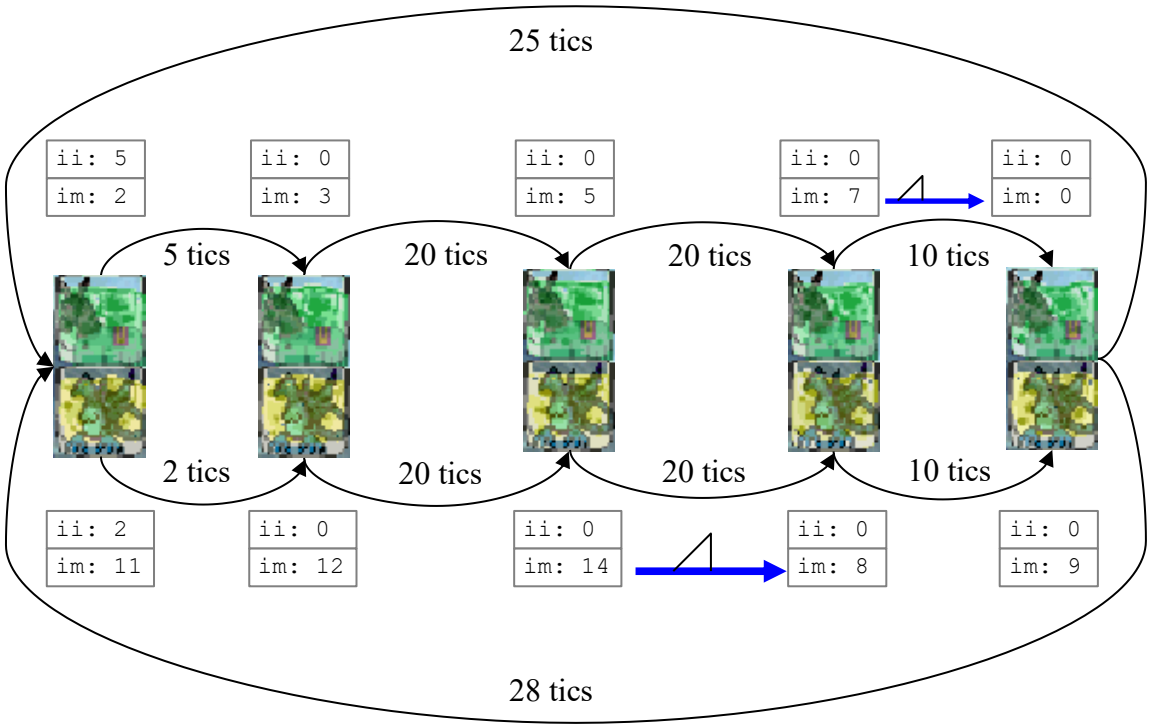
4.3.4 Subtarea 2Gd: rsi_timer2()

Rutina para atender las interrupciones periódicas del *timer* 2. Su propósito básico es, a cada interrupción de tiempo (tic), actualizar el estado de las metabaldosas de las gelatinas registradas en el mapa de gelatinas `mat_gel[][COLUMNS]`, a partir de los siguientes pasos:

- analizar el campo `ii` de cada posición de la matriz de gelatinas; si dicho valor es mayor que cero, decrementarlo y continuar con la siguiente posición,
- en caso de que el campo `ii` de la posición analizada sea cero, aumentar el índice de metabaldosa de la gelatina correspondiente, volviendo al índice inicial en caso de que se haya llegado al máximo para el tipo de gelatina detectado (simple: 7 → 0, doble: 15 → 8),
- activar la variable `update_gel`, en el caso de que se requiera actualizar la metabaldosa de alguna gelatina del tablero de juego (en la próxima interrupción de *VBlank*).

Los siguientes gráficos muestra el efecto de animación de una gelatina simple y una doble, con distintos tiempos de actualización (desfase de tics), y con sus respectivos

cambios cíclicos de índices de metabaldosas (las flechas azules indican reinicio del rango):



4.4 Tarea 2H: desplazamiento de fondo

4.4.1 Subtarea 2Ha: rsi_vblank()

Implementar la parte de la RSI del retroceso vertical (en el fichero `RSI_timer0.s`) encargada de actualizar el registro E/S de desplazamiento X del fondo 3, a partir de los siguientes pasos:

- si la variable `update_bg3` está desactivada (igual a 0), ignorar todos los pasos siguientes,
- en caso contrario, copiar el valor de la variable global `offsetBG3X` (declarada en el fichero `RSI_timer3.s`) sobre el registro E/S de desplazamiento **BG3X**, ajustando el valor del número entero de la variable al formato de coma fija 0.20.8 del registro,

- finalmente, desactivar la variable `update_bg3`.

Por otro lado, la RSI del *timer* 3 será la encargada de actualizar el valor de `offsetBG3X`, a la frecuencia que se indique para el *timer* 3 (ver apartado 4.4.4).

4.4.2 Subtarea 2Hb: activa_timer3()

Rutina para inicializar el *timer* 3, además de poner a 1 la variable global `timer3_on`. Esta rutina se llama desde la función principal `main()`, en el caso de que el *timer* esté desactivado y se pulse el botón **Y**.

También hay que inicializar el divisor de frecuencia del *timer* 3, copiando en su registro E/S de datos el contenido de la variable `divFreq3`, que se tiene que calcular para conseguir una velocidad de desplazamiento de la imagen de fondo de 10 píxeles por segundo.

Por último, la rutina `activa_timer3()` debe activar dicho *timer* mediante su registro E/S de control, seleccionando la frecuencia de entrada adecuada y permitiendo la activación de las interrupciones asociadas.

4.4.3 Subtarea 2Hc: desactiva_timer3()

Rutina para desactivar el *timer* 3 a través de su registro E/S de control, además de desactivar (poner a 0) la variable global `timer3_on`. Esta rutina se llama desde la función principal `main()`, en el caso de que el *timer* esté activado y se pulse el botón **Y**.

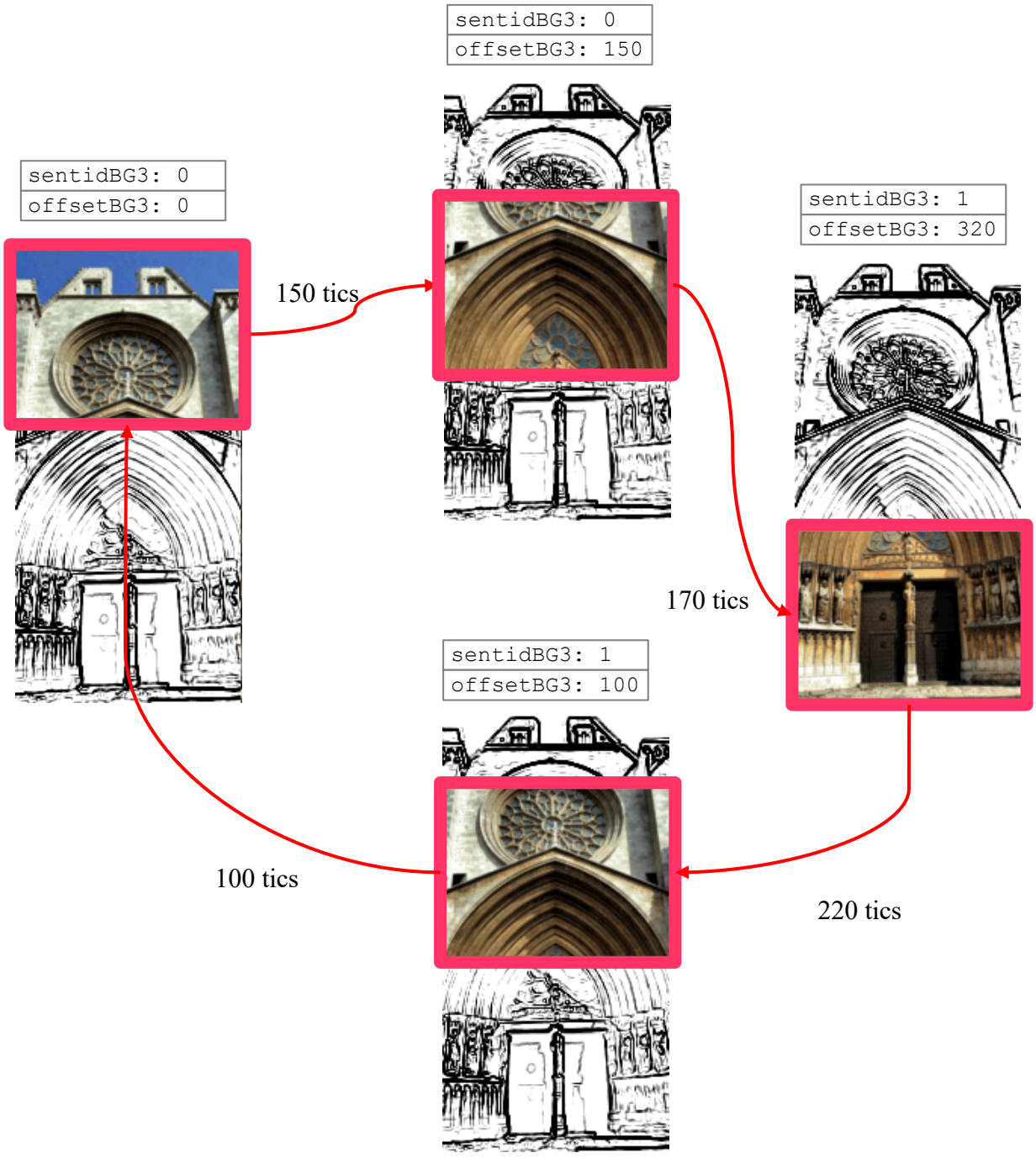
4.4.4 Subtarea 2Hd: rsi_timer3()

Rutina para atender las interrupciones periódicas del *timer* 3. Su propósito básico es, a cada interrupción de tiempo (tic), actualizar el desplazamiento del fondo 3 contenido de la variable `offsetBG3X` según el sentido que indique la variable `sentidBG3`, a partir de los siguientes pasos:

- si `sentidBG3` es 0, incrementar 1 píxel el desplazamiento; si `sentidBG3` es 1, decrementar 1 píxel el desplazamiento,

- si el desplazamiento ha llegado a su límite en el sentido actual, cambiar `sentidBG3` al sentido contrario,
- activar la variable `update_bg3` para que en la próxima activación de `rsi_vblank()` se transfiera el desplazamiento actual al registro de control correspondiente.

Los siguientes gráficos representan el efecto de animación del fondo, incluidos los dos tipos de cambio de sentido:



5 Tareas restantes

5.1 Tarea 2I: actualización de las rutinas de la fase 1

5.1.1 Subtarea 2Ia: recombina_elementos()

Modificar la rutina `recombina_elementos()` para que active el movimiento de los *sprites* de acuerdo con la reubicación de los elementos de la matriz de juego, llamando a la rutina de soporte `activa_elemento()`, indicándole por parámetro la posición inicial y final de cada reubicación.

5.1.2 Subtarea 2Ib: elimina_secuencias()

Modificar la rutina `elimina_secuencias()` para que elimine los elementos y actualice las gelatinas de las posiciones de las secuencias formadas en cada jugada, a partir de los siguientes pasos:

- si alguna de las posiciones de la secuencia contiene una gelatina simple o doble, llamar a la rutina de soporte `elimina_gelatina()`,
- para todas las posiciones de la secuencia a eliminar, llamar a la rutina de soporte `elimina_elemento()`.

5.1.3 Subtarea 2Ic: baja_verticales()

Modificar la rutina `baja_verticales()` para que gestione el movimiento y la creación de los *sprites* de acuerdo con la bajada de elementos en vertical, a partir de los siguientes pasos:

- para los nuevos elementos generados aleatoriamente, llamar a la rutina de soporte `crea_elemento()` para crear los *sprites* correspondientes;

Atención: es muy recomendable crear los nuevos elementos "encima" del tablero de juego (fila: –1) y animarlos para que bajen hasta la casilla vacía que corresponda, para simular que caen desde arriba, en vez de que aparezcan repentinamente.

- para todos los elementos que se tengan que mover, llamar a la rutina de soporte `activa_elemento()`.

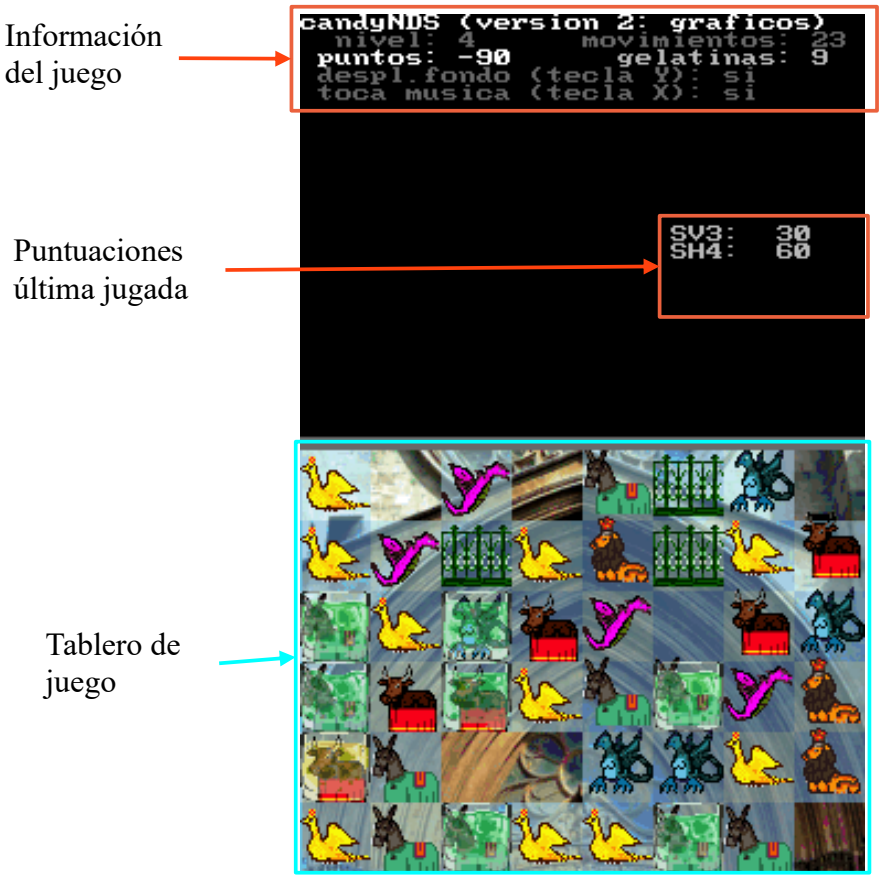
5.1.4 Subtarea 2Id: baja_laterales()

Modificar la rutina `baja_laterales()` para que gestione el movimiento de los *sprites* de acuerdo con la bajada de elementos en diagonal, llamando a la rutina de soporte `activa_elemento()`.

5.2 Finalización de la fase 2

La versión definitiva del programa tendrá que eliminar todo el código fuente que solo tenga un propósito de depuración, como por ejemplo, la visualización de la matriz en formato texto o la posibilidad de superar o repetir el nivel pulsando los botones **B** y **START**. Esta eliminación significa tanto modificar el programa principal como borrar completamente el código fuente de las funciones de soporte que no se utilicen, como `escribe_matriz()` o `copia_mapa()`.

La siguiente figura muestra un ejemplo de la visualización final del programa, sin la matriz de números:



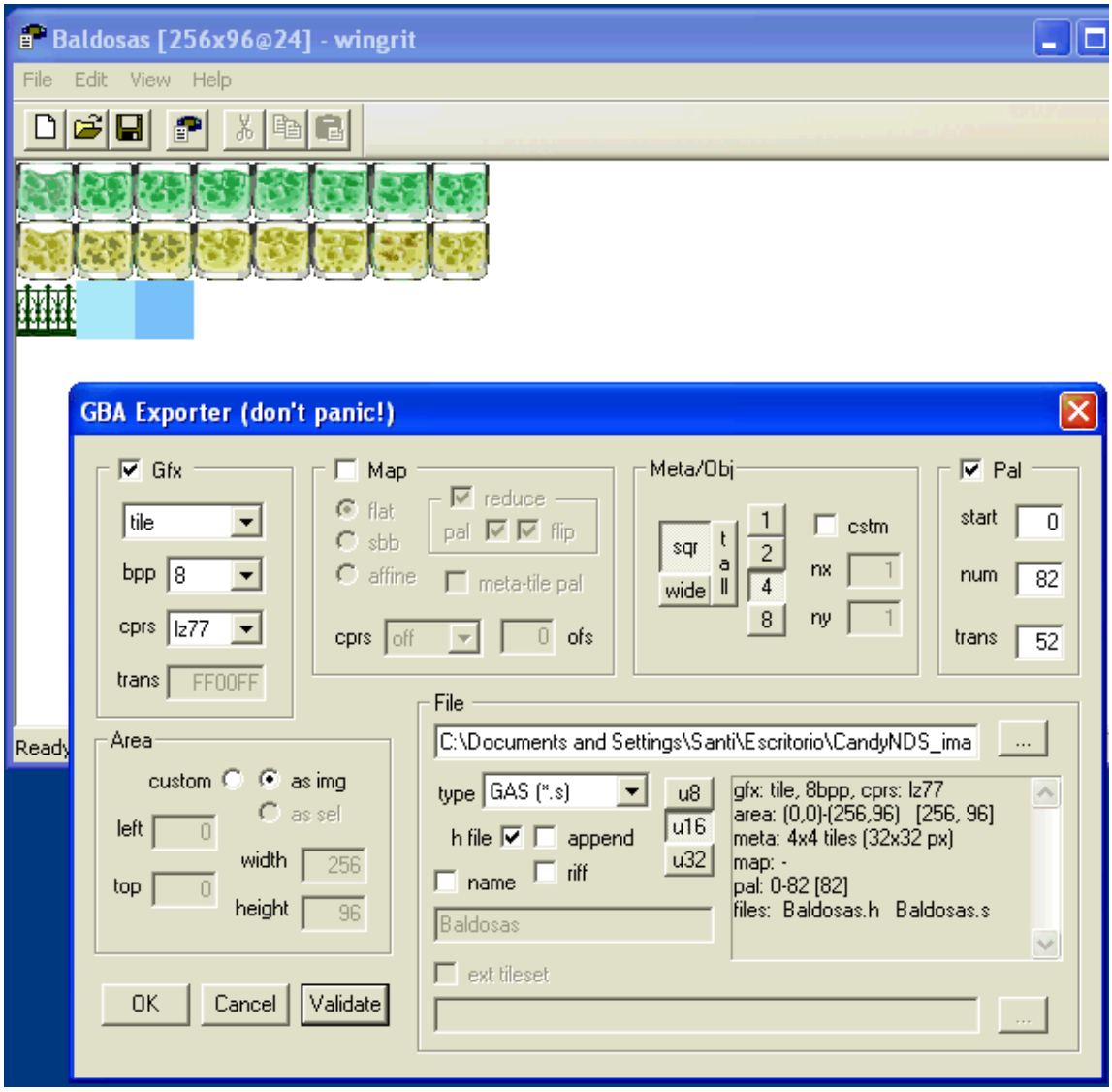
Opcionalmente, se puede añadir una opción para activar y desactivar la reproducción de una música de fondo mientras se desarrolla el juego.

6 Generación de ficheros gráficos

Para generar los ficheros que contienen los datos de los gráficos (directorio `graphics/` del proyecto), hemos utilizado el programa ***Grit***, que se puede descargar gratuitamente de la siguiente página web: http://www.coranac.com/projects/grit/

Existe una versión para ejecutarse desde línea de comandos y otra que presenta una interfaz gráfica para *Windows*, que se llama ***Wingrit***, y que permite cargar ficheros gráficos en múltiples formatos (ver manuales) y transformarlos en declaraciones de números en lenguaje ensamblador (ficheros `.s`) correspondientes a píxeles de baldosas, las paletas de colores y los mapas, así como en imágenes *bitmap*.

La siguiente imagen muestra un ejemplo de carga del fichero `Baldosas.tif`, y de la ventana de configuración de la transformación **GBA Exporter** (que se activa desde el menú **View**), con los parámetros correspondientes:



En este ejemplo se generan los ficheros `Baldosas.h` y `Baldosas.s`, que contendrán las variables `BaldosasTiles[]` y `BaldosasPal[]`, que definen los píxeles de las baldosas y la paleta de los colores de referencia, y que se cargarán en las inicializaciones gráficas (tareas **2B** y **2C**).

Para simplificar el número de ficheros y directivas `#include`, hemos fusionado manualmente los ficheros de cabecera `Fondo.h`, `Baldosas.h` y `Sprites.h` dentro de un único fichero `include/Graphics_data.h`.

Por último, en el directorio `graphics/` se ha añadido un fichero de texto `Grit_config.txt` para registrar los parámetros del ***Grit*** que se han usado para transformar cada fichero gráfico.

Con este ejemplo se pretende mostrar cómo se pueden generar los gráficos para la NDS con relativa facilidad, y se deja abierta la posibilidad de sustituir los ficheros gráficos proporcionados para esta práctica (fichero `CandyNDS_images.zip` en el espacio *Moodle* de la asignatura) por otros que utilicen otras representaciones de los elementos, los bloques sólidos, las gelatinas animadas, el tablero de juego y la imagen de fondo.

Además, en el espacio *Moodle* se pueden consultar el guion de los laboratorios sobre gráficos de la NDS, el segundo de los cuales contiene más detalles acerca de la creación de escenarios y conversión de ficheros gráficos en sus respectivos ficheros fuente, usando comandos textuales del ***Grit***.