UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Gestión de los sprites en la implementación del tutorial "The Bouncing Ball" para NDS

Apellidos, nombre	**Agustí Melchor, Manuel** (magusti@disca.upv.es)
Departamento	**Dpto. de Ing. De Sistemas y Computadores (DISCA)**
Centro	**Escola Tècnica Superior d'Enginyeria Informàtica** Universitat Politècnica de València

1 Resumen de las ideas clave

En este artículo vamos a hablar del uso de los *sprites* en aplicaciones para la videoconsola Nintendo DS (NDS). Para ello hemos escogido revisar el trabajo de Mukunda Johnson [1] quien, entre otras cosas, realizó un tutorial[1] sobre el desarrollo de aplicaciones para NDS.

Sobre el trabajo de Johnson, Jaén [2] realizaría un trabajo de actualización y ampliación para seguirlo junto al tutorial original, al tiempo que se corrigieron algunas erratas del texto original, se hicieron algunos cambios del código debidos a modificaciones de las librerías en que se basa y se ampliaron algunos apartados con pequeñas "demos" de conceptos.

Ya no es posible consultar el tutorial original en la red, así que **el presente trabajo ofrece una versión del apartado de uso de los *sprites* actualizada** a las versiones existentes de las librerías sobre las que se basa. Manteniendo estos documentos disponibles queremos rendir un homenaje a estos desarrolladores que han contribuido con su código y con sus explicaciones a que otros puedan adentrarse en este difícil campo de desarrollo de aplicaciones para videoconsolas, que son plataformas muy populares y diferentes del típico ordenador de sobremesa.

Para facilitar el seguimiento de este trabajo se han dispuesto todos los elementos del proyecto que aquí se comenta en GitHub [3]. Nos ocuparemos en este artículo de comentar las acciones necesarias para mover un objeto gráfico sobre la escena, utilizando la botonera de la NDS para interactuar con él, al tiempo que aparecerán otros factores que influirán en su dinámica como la "gravedad" y el "rozamiento con el aire" que lo harán avanzar en una trayectoria casi parabólica y las colisiones con el suelo lo harán deformarse al rebotar en el mismo.

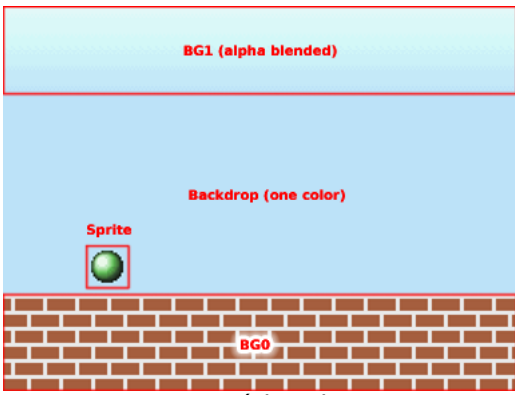


Figura 1: Imagen esquemática de proyecto. Fuente [1].

Asumiremos que se dispone ya de un escenario creado como el que muestra la Figura 1. En él podemos observar dos capas (fondos o *backgrounds*) que hacen de cielo (el BG1 que está en segundo plano) y de suelo (el BG0 que está "delante" de BG1). Sobre esta segunda plataforma rebota la pelota (el objeto o *sprite*) que controlamos y que podrá: moverse hacia la derecha e izquierda sin fin y "saltar" hacia arriba, viéndose afectada por la "gravedad".

[1] Publicado originalmente en "How to Make a Bouncing Ball Game for Nintendo DS" <http://ekid.nintendev.com/bouncy/>, este trabajo ya no está disponible en la red; pero existe una copia de fecha de 2015 en [3] que es al que nos referiremos al hablar del "tutorial original".

2 Objetivos

Una vez que el lector haya leído con detenimiento este documento:

- Podrá experimentar con un código que utiliza los conceptos básicos de desarrollo de aplicaciones de videojuegos en la consola NDS relativos al uso de *sprites*.
- Dispondrá de un ejemplo de código que introduce la interacción con el *sprite* a través de la botonera de la NDS.
- Podrá experimentar con un ejemplo de código que introduce la interacción con el *sprite* a través del uso de parámetros de modelos físicos como gravedad y rozamiento, así como la detección de colisiones.

No es un objetivo instalar las herramientas que permiten la creación del ejecutable así como la carga del mismo en la consola, de hecho, nosotros utilizaremos un emulador: *DeSmuME* [4]. Para ello se puede recurrir a los trabajos de [5] y [6]. Tampoco es un objetivo de este trabajo entrar a ver las características hardware de la NDS como plataforma optimizada para tareas de videojuego. Si el lector tiene curiosidad o necesidad de profundizar en los elementos de la NDS es recomendable consultar [7]. Antes de ver un ejemplo práctico, hemos de hablar un poco de teoría, en concreto el uso de los *sprites* que hace la NDS. Veamos qué consideraciones hay que tener en cuenta para cargarlos como recursos en la memoria de la videoconsola.

3 Introducción

Para dibujar con rapidez en las dos pantallas de la NDS, ofrece una aceleración por *hardware* que se basa en la existencia de los dos motores gráficos que se asignan a cada pantalla, la división de la **memoria de vídeo** (**VRAM**, de 656 MiB) en bancos de memoria, cuyo propósito está preestablecido y cuya asignación a uno de los motores es configurable.

La VRAM de la NDS, véase la Figura 2, se gestiona en base a un esquema de particiones de tamaño fijo, denominadas **bancos**. Estos, tienen asignados su posible contenido y cometido. En el caso que nos ocupa, los *sprites*, solo se pueden ubicar en los bancos que están etiquetados como “OBJ GRAPHICS”. Hay dos, puesto que hay dos subsistemas o motores gráficos (*Main* y *Sub*) que incorporan acciones especializadas para la gestión de estos “objetos” (los*sprites*). Cada motor puede acceder a una cantida de memoria diferente para los *sprites*: el *Main* a 352 KiBytes (utilizando los bancos A, B, E, F y G. mientras que el *Sub* a 144 KiBytes (bancos D e I); sin que sea posible compartirla entre ellos. Se puede consultar [10] para ver más detalles de bajo nivel.

La información de los sprites se agrupa en la memoria que se conoce como la **OAM** (*Object Attribute Memory*) y que asocia a cada uno de los sprites que puede direccionar cada motor, cuatro registros o valores de 16 bits que codifican los atributos y operaciones a realizar sobre cada *sprite*.

Así que trabajar con *sprites* en la NDS significa que hemos de llevar los *sprites* a la memoria correspondiente del motor de gráficos que escojamos. Además, habrá que asignar los valores a los atributos de cada *sprite* creado en la OAM. Veamos en qué se traduce esta forma de trabajar con estos elementos.

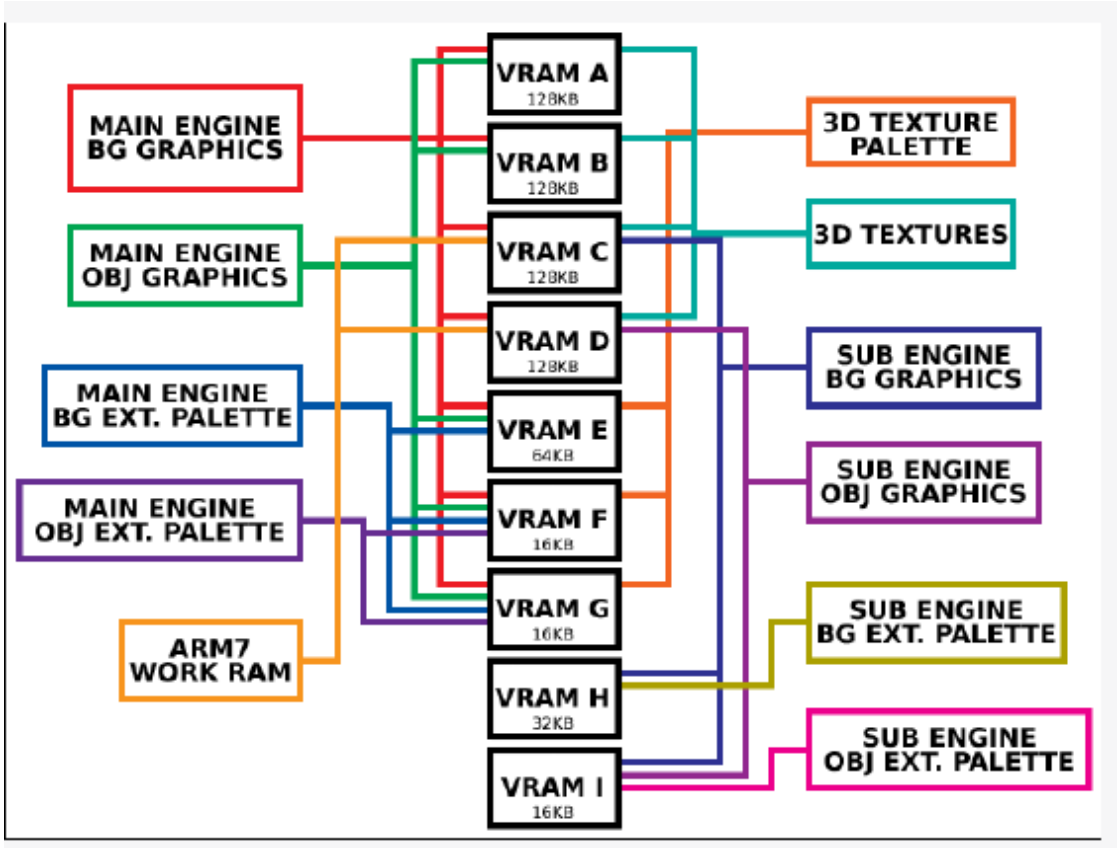
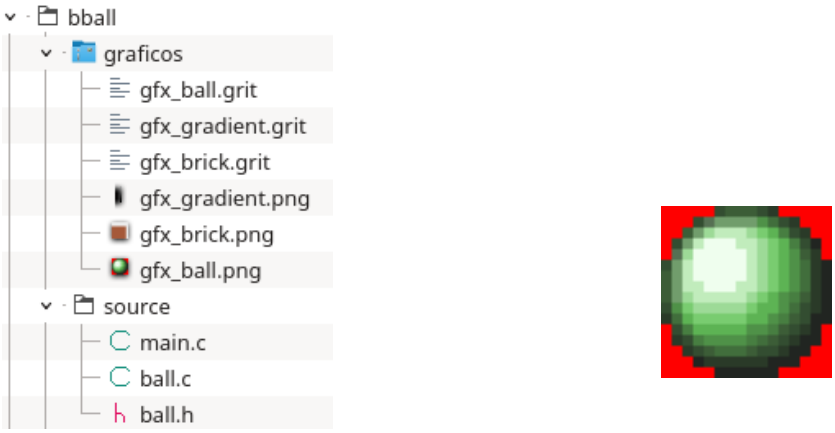


Figura 2: Memoria en la NDS: bancos y particiones de la VRAM (fuente [3])..

4 Estructura y contenido del proyecto

El ejemplo práctico sobre el que trabajamos, veíamos su diseño en la Figura 1, tiene la estructura de ficheros que muestra la Figura 3a y vamos a asumir, por brevedad de esta exposición, que se ha desarrollado ya la escena (los elementos BG0 y BG1), por lo que aquí nos centraremos en el fichero gráfico (*gfx_ball.png*, *Figura 3b*) que le da la apariencia al sprite y el contenido de los ficheros dentro del directorio *source* (Figura 3a) que corresponda a esta tarea.



a) *b)*

Figura 3: Ficheros del proyecto: (a) jerarquía y (b) el sprite para la "pelota".

El tutorial original se desarrolla en base a **cuatro** apartados: carga de la imagen para el *sprite*, generación de varios *sprites*, *movimiento de un sprite*, control del usuario, incorporación del efecto de la física y colisiones. Se podrá ver la evolución de estas etapas en el GitHub [3] asociado, pero aquí nos centraremos en la versión final.

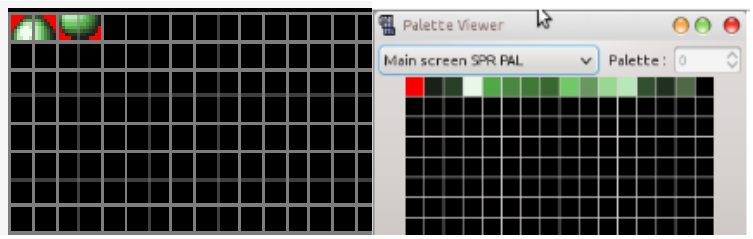
4.1 Conversión a teselas de los *sprites*

Los *sprites* se descomponen en teselas (tiles), que son pequeños mapas de bits de tamaño cuadrado. La NDS admite [1] de 8x8 píxeles y 4bpp[2] (16 colores) en una de 16 posibles paletas y de 8bpp (en cuyo caso solo hay una paleta para todos los sprites de 256 colores).

```
$ cat graficos/gfx_ball.grit

# convert ball graphic: # 16 colors
-pn16

# 4bpp
-gB4
```



a) *b)* *c)*

Figura 4: Conversión del sprite a teselas: (a) instrucciones para GRIT y representación gráfica del contenido de la VRAM al almacenar el "sprite" (b, extraída de [1]) y (c) paleta de color.

La imagen de partida es una imagen en formato PNG de 16x16 píxels indexada de 8 bits (o 256 colores), por lo que es necesario trocear la imagen en teselas de 8x8 y generar una paleta de solo 16 valores y ubicarlo en memoria en el banco correspondiente, al estilo de lo que muestra la Figura 4b. Para ello utilizaremos la aplicación GRIT [9] a la que instruiremos con el fichero gfx_ball.grit, vease la Figura 4b. Observamos que, Figura 4c, en la paleta cargada en memria, el primer color (en este caso rojo), siempre es el color "transparente"por lo que no lo veremos al pintarlo en la escena.

4.2 La OAM

No basta con cargar los *sprites* en teselas y las paletas de los mismos a la VRAM. Para que el hardware de la NDS sepa qué hacer con ellos hay que asignar los valores oportunos a los "atributos" de los sprites que están contenidos en la OAM de 1024 bytes. Esta alberga [3] un vector cuyos elementos son cuatro valores de 2 bytes (véase la Figura 5) por lo que caben (1024/(4*2)) = 128 *sprites* como máximo en un mismo momento. Estos atributos permiten establecer la posición de la esquina superior izquierda del *sprite* en pantalla, si está visible o no, su escalado o rotación, etc.

4.3 Uso de valores en coma fija

Para la gestión de la posición y otros factores que afectan a la dinámica del *sprite*, se podrían utilizar valores decimales para obtener una precisión mayor que la resolución en píxeles de las pantallas de la NDS. En nuestro caso [3], si quisiéramos mover un píxel un

[2] *Bits per pixel* (bpp), es el número de bits para un índice de color asociado a cada punto de las teselas. En este caso la tesela ocupa 8x8x4bits = 256 bits = 32 Bytes.

sprite cada vez que se pinta en pantalla, como la velocidad de este refresco es de unos 60Hz, la velocidad sería muy alta (el ancho de pantalla se recorrería en 256/60 o aproximadamente 4 seg.). Así, la solución es incrementar en fracciones como, p. ej. 0,25 píxeles cada cuadro.

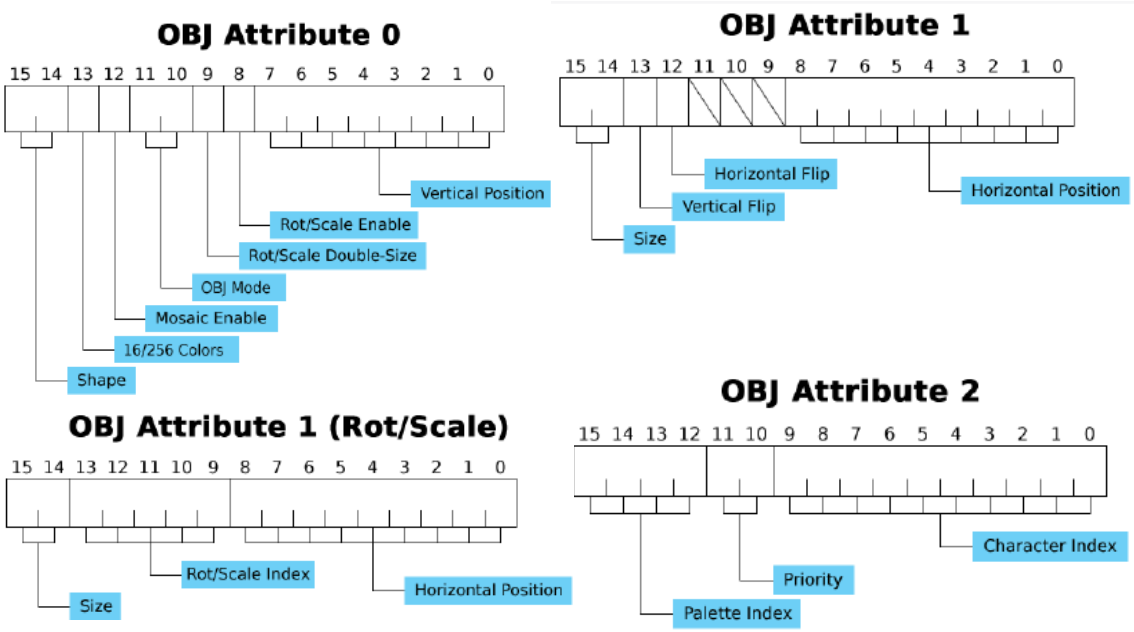


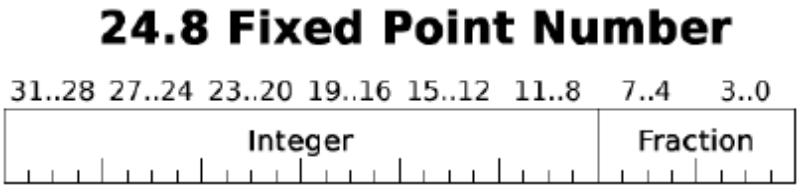
Figura 5: Para cada sprite hay 4 "atributos" que lo caracterizan. Fuente [1].

Puesto que la NDS no tiene un operador de coma flotante *hardware*, todos sus valores han de ser enteros si se quiere rapidez, entonces ¿cómo representamos valores fraccionales como 13,37 metros[3]? Lo haremos utilizando valores en centímetros, p. ejs. 1337 cm. De las cuatro cifras de este valor, dos se utilizan para los decimales, por lo que se dice que este número está en formato de "coma fija X.2". En el ejemplo que nos ocupa, en lugar de valores reales en coma flotante (con formato de mantisa y exponente), se utilizan valores de coma fija en los que se ha de decidir cuántos bits se utilizan para la parte entera y cuántos para la decimal.

Figura 6: Un reloj como ejemplo de formato en coma fija: hay parte entera y fraccional, que se recalca con el ':'. Fuente [1].

Imaginemos un ejemplo como el reloj de la Figura 6. El valor en memoria sería 1337 que se puede entender como las 13 horas y 17 minutos y así se representa en la figura con los dos puntos separando cada parte del valor. Las conversiones de este valor a horas o minutos involucran operaciones de división o multiplicación con un valor constante (60), por lo que también son operaciones lentas. Así que, escoger un formato de coma fija donde la parte fraccional sea potencia de dos, facilita estas operaciones al convertirlas en desplazamientos con el número de bits dedicado a la parte decimal. Veremos que se utiliza el formato 20.10 y el 24.8 (descrito en la Figura 7, junto a las operaciones para la conversión entre formatos), donde la parte fraccional permite abordar un rango de 256 valores que es el valor de resolución horizontal de las pantallas de la NDS y, así, obtener la precisión de centésimas de píxel.

[3] Parafraseando el ejemplo de <http://www.coranac.com/tonc/text/fixed.htm>.



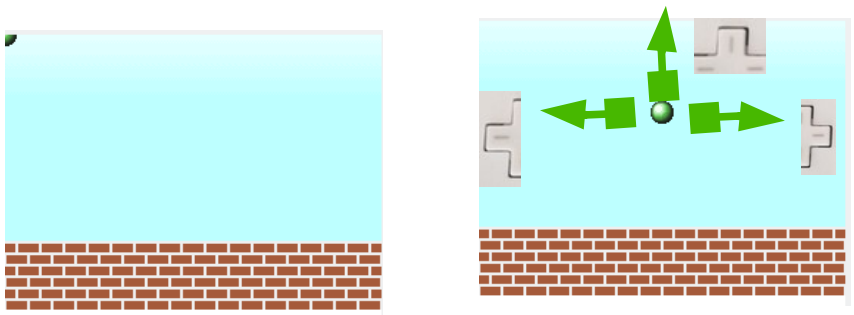
Fórmulas de conversión [3] coma flotante – coma fija (I.F de 24.8) :
decimal (d) a coma fija: round (d * (2*F)) → x
y de coma fija (x) a decimal: ((x*2)-F) → d

Figura 7: El formato de coma fija 24.8

5 Desarrollo e implementación

El ejemplo de código que se lista a continuación permite cargar y mover un *sprite* en la pantalla simulando la existencia de una plataforma o suelo que se mueve siguiendo la dirección de la pelota al alcanzar uno de los bordes de la pantalla. Por brevedad en la exposición se han destacado las instrucciones que en los ficheros *main.c* y *ball.c (Figura 10 a la Figura 12)* que hacen referencia al trabajo con *sprites*, manteniendo y ampliando los comentarios. El código del programa principal se encarga de:

- Configurar la memoria y cargar los gráficos (*setupGraphics*) en la función *main*.
- Posicionar la pelota (*resetBall*), como muestra la Figura 8a, que si no lo haces te esperará en la esquina superior izquierda,
- Hacerla subir y moverse en horizontal (*processInput*), en función de lo que el usuario decida al pulsar los botones, como indica la Figura 8b.



a) *b)*

Figura 8: Resultado de inicializar la posición de la pelota y opciones de movimiento.

También veremos la operativa de la pelota en *ball.c* (*Figura 13 y Figura 14*)[4], que incorpora a la caída "básica" (lineal), como muestra la Figura 9a:

- El efecto de la física del movimiento (*ballUpdate*), donde la gravedad y el rozamiento hacen que la trayectoria de descenso no sea rectilínea, Figura 9b.
- Los límites de la escena (*ballRender*), las colisiones y deformaciones, Figura 9c.

[4] Cuidado si estás siguiendo el tutorial original, porque en *main.c*, la función *update_logic*; pasará por llamarse *updateLogic* y acabará siendo *processLogic*. Así como *update__graphics* pasará a ser *updateGraphics*.

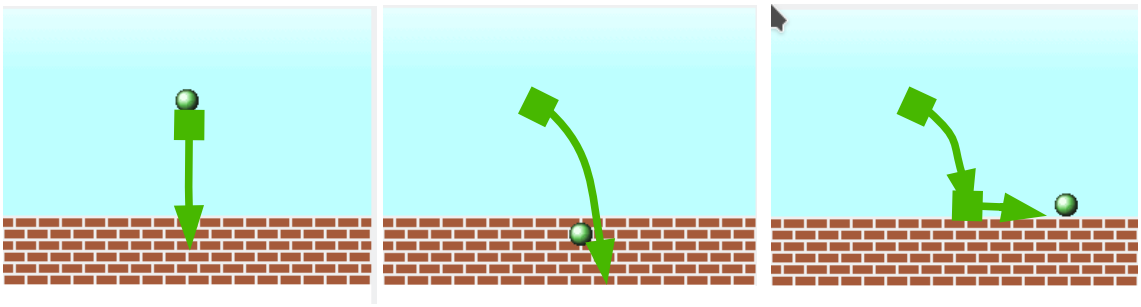


Figura 9: Efectos de física de modelos: (a) le afecta la gravedad, (b) cae en parábola desde el punto inicial y (c) la colisión con el "suelo" detiene la caída.

```
1       #include <nds.h>
2       #include <stdio.h>
3
4       #include <gfx_ball.h>
5       #include <gfx_brick.h>
6       #include <gfx_gradient.h>
7       #include "ball.h"
8
9       ball g_ball;
10      void setupInterrupts( void );
11      void resetBall( void );
12      void setupGraphics();
13      void updateLogic();
14      void updateGraphics();
15      void processLogic( void );
16
17      int main( void ) {
18         setupGraphics();
19         resetBall();
20         while(1)   {
21                 processLogic();
22                 swiWaitForVBlank();
23                 updateGraphics();
24         }
25      }
26      void resetBall( void ) {              // reset ball attributes
27        g_ball.sprite_index = 0;            // use sprite index 0 (0->127)
28        g_ball.sprite_affine_index = 0;     // use affine matrix 0 (0->31)
29        g_ball.x = 128 << 8;                // X = 128.0
30        g_ball.y = 64 << 8;                 // Y = 64.0
31        g_ball.xvel = 100 << 4;             // start X velocity a bit to the right
32        g_ball.yvel = 0;                    // reset Y velocity
33      }
...
```

```
typedef struct t_ball
{
  int x;                    //24.8 punto fijo
  int y;                    //24.8 punto fijo
  int xvel;                 //20.12 punto fijo
 int yvel;                  //24.8 punto fijo
  u8 sprite_index;          //entrada OAM (0->127)
  u8 sprite_affine_index;   //entrada OAM affine (0->31)
  int height;               //ancho del balón
} ball;
```

Figura 10: Código del ejemplo, parte 1 . Partes del código extraído de [1] y [2].

```
...
34      //BGs: Índices para las teselas (“tiles”)
35      #define tile_empty          0       //tile 0 = empty
36      #define tile_brick          1       //tile 1 = brick
37      #define tile_gradient       2       //tile 2 = gradient
38      //BGs Índices para las paletas
39      #define pal_bricks          0       //paleta brick (entrada 0->15)
40      #define pal_gradient        1       //paleta gradient (entrada 16->31)
41      #define backdrop_colour  RGB8( 190, 255, 255 )
42
43      u16 *bg0map, *bg1map;
44
45      // Sprites
46      #define tiles_ball          0       // ball tiles (16x16 tile 0->3)
47      #define sprites (           (SpriteEntry*)OAM)
48      #define pal_ball            0       // ball palette (entry 0->15)
49
50
51      void setupGraphics( void ) {
52       int n;
53      // Inicialización de memoria y carga de los elementos gráficos de fondo
54      videoSetMode( MODE_0_2D | DISPLAY_BG0_ACTIVE | DISPLAY_BG1_ACTIVE );
55
56      // Sprite
57      oamInit(&oamMain, SpriteMapping_1D_32, false); //!extPalette
58        dmaCopy( gfx_ballTiles, SPRITE_GFX, gfx_ballTilesLen );
59        dmaCopy( gfx_ballPal, SPRITE_PALETTE, gfx_ballPalLen );
60      } // Fi de setupGraphics
61
62      int g_camera_x;
63      int g_camera_y;
64
65      void updateGraphics( void ) {  // Menejar pilota i càmera: update graphical information
66        // update ball sprite, camera = 0, 0
67        ballRender( &g_ball, 0, 0 );
68
69        REG_BG0HOFS = g_camera_x >> 8;
70      } // Fi de updateGraphics
71      #define x_tweak (1<<2)          // for user input  (Ho baixe de (2<<8)
72      #define y_tweak 50              // for user input: quant més gran més salta
73
74      void processInput( void ) {
75        scanKeys();
76       int keysh = keysHeld();        // process user input
77       if( keysh & KEY_UP ) {         // if UP is pressed --> tweak y velocity of ball
78        g_ball.yvel -= y_tweak; }
79        if( keysh & KEY_DOWN ){       // if DOWN is pressed --> tweak y velocity of ball
80        g_ball.yvel += y_tweak; }
81       if( keysh & KEY_LEFT ) {       // if LEFT is pressed --> tweak x velocity
82        g_ball.xvel -= x_tweak;  }
83       if( keysh & KEY_RIGHT ) {      //  if RIGHT is pressed --> tweak y velocity
84        g_ball.xvel += x_tweak;  }
85       } // Fi de processInput
...
```

Figura 11: Código del ejemplo, parte 2 . Partes del código extraído de [1] y [2].

```
...
...
86        void updateCamera( void ) {
87        int cx = ((g_ball.x)) - (128 << 8);      // cx = desired camera X (en format 24.8)
88        int dx;                                  // dx = difference between desired and current position
89        dx = cx - g_camera_x;
90        if( dx > 10 || dx < -10 )  //10 is the minimum threshold
91          dx = (dx * 50) >> 1; // scale the value by some amount // 50 i 10
92        g_camera_x += dx;           // add the value to the camera X position
93        g_camera_y = 0;             // camera Y is always 0
94        } // Fi de updateCamera
95
96        void processLogic( void ) {
97        processInput();
98        ballUpdate( &g_ball );
99        /* "Whoops, the ball went out of the screen." To fix this, we will add a 'camera' that will
100       follow the ball around the screen.Make two more global variables.*/
101       updateCamera();
102       } // fi de processLogic
```

Figura 12: Código del ejemplo, parte 3 . Partes del código extraído de [1] y [2].

```
1         #include <nds.h>
2         #include "ball.h"
3         #define c_radius (8<<8) // the radius of the ball in *.8 fixed point
4         #define c_diam   16     // the diameter of the ball (integer)
5
6         void ballRender( ball* b, int camera_x, int camera_y ) {
7         // Cada entrà de la taula de OAM ocupa 4* 2bytes == 16 bits = u16
8         u16* sprite = OAM + b->sprite_index * 4;
9         int x, y;
10        x = ((b->x - c_radius*2) >> 8) - camera_x;
11        y = ((b->y - c_radius*2) >> 8) - camera_y;
12        if( x <= -16 || y <= -16 || x >= SCREEN_WIDTH || y >= SCREEN_HEIGHT)  {
13              //sprites fuera de rango. Deshabilitar el sprite
14              oamSetHidden( &oamMain, b->sprite_index, true);
15              return;
16        }
17        sprite[0] = (y & 255) | ATTR0_ROTSCALE_DOUBLE;
18        sprite[1] = (x & 511) | ATTR1_SIZE_16 | ATTR1_ROTDATA( b->sprite_affine_index );
19        sprite[2] = 0;
20        u16* affine;
21        affine = OAM + b->sprite_affine_index * 16 + 3;
22        affine[4] = 0;
23        affine[8] = 0;
24         affine[0] = (b->height * (65536/c_diam)) >> 16;       // int pa
25        affine[12] = 65536 / pa;                                //  int pd
26        }
27
...
```

Figura 13: Código de ball.c. Partes del código extraído de [1] y [2].

```
...
28
29        #define c_gravity 80          // gravity constant (add to vertical velocity) (*.8 fixed)
30        #define c_air_friction 1       //  friction in the air... multiply X velocity by (256-f)/256
31        #define c_ground_friction 30    //  friction when the ball hits the ground, multiply X by (256-f)/
256
32        #define c_platform_level ((192-48) << 8)    //  the level of the brick platform in *.8 fixed point
33        #define c_bounce_damper 20      //  the amount of Y velocity that is absorbed when you hit
the ground
34
35        #define min_height  (1200)   // the minimum height of the ball (when it gets squished) (*.8)
36        #define min_yvel (1200) // the minimum Y velocity (*.8)
37        #define max_xvel (1000<<4)  // the maximum X velocity (*.12)
38
39        // clamp integer to range
40        static inline int clampint( int value, int low, int high ) {
41          if( value < low ) value = low;
42          if( value > high) value = high;
43          return value;
44        }
45
46        void ballUpdate( ball* b ) { // Físicas: update ball object (call once per frame)
47         b->x += (b→xvel>>4);                  // add X velocity to X position xvel is 20.12 while x is 24.8
48         b->xvel = (b->xvel * (256-c_air_friction)) >> 8;          // Gravetat: apply air friction to X vel.
49         b->xvel = clampint( b->xvel, -max_xvel, max_xvel );   // clamp X velocity to the limits
50         b->yvel += c_gravity;                                     // add gravity to Y velocity
51         b->y += (b→yvel);                                         // add Y velocity to Y position
52         if( b->y + c_radius >= c_platform_level )     {           // Bounce on the platform
53           b->xvel = (b->xvel * (256-c_ground_friction)) >> 8;    // apply ground friction to X velocity
54           if( b->y > c_platform_level - min_height )  {// the ball has been squished to min. height?
55            b->y = c_platform_level – min_height;                // mount Y on platform
56            // negate Y velocity, also apply the bounce damper
57            b->yvel = -(b->yvel * (256-c_bounce_damper)) >> 8;
58            // clamp Y to mininum velocity (minimum after bouncing, so the ball does not settle)
59            if( b->yvel > -min_yvel )   b->yvel = -min_yvel;
60           }
61           b->height = (c_platform_level - b->y) * 2;              // calculate the height
62         }
63         else {  b->height = c_diam << 8; }}
64        } // Fi ballUpdate
```

Figura 14: Código de ball.c (continuación). Partes del código extraído de [1] y [2].

5.1 Etapas del proyecto

Para entender el progreso en la implementación del "personaje" y la interacción con el usuario, se ha seguido el guión del tutorial original [3] y se han generado tres etapas:

Ya hemos dicho que los contenidos del tutorial original y las versiones del código actualizadas se han ido disponiendo en *GitHub* [3]. El resultado final es el que se puede ver en este artículo, pero para ver cómo el proyecto va creciendo se han guardado las etapas intermedias que dan lugar a las versiones (las que corresponden a este artículo están en el directorio *tutorial_Mukunda_JoseDavid/bball__versions/bball__subVersionsSprites/*) y corresponden a:

- bball__v04. Es la que inicia este apartado de los sprites. Se ocupa de la configuración de memoria y carga de los recursos asociados al gráfico que hace de "pelota", el *sprite*.
- bball__v05. Generará 51 *sprites*, utilizando el inicial para generar cincuenta copias de él que se moverán "alegremente" (aleatoriamente) por la pantalla.
- bball__v06. Final. Que vuelve a dejar solo uno y que incorpora la interacción del usuario con la aplicación, mediante el uso de la botonera de la NDS. Esta es la versión que se muestra en este artículo y contiene las las simplificaciones y normalizaciones de código que hemos introducido. Se muestra en las figuras *10 a la 14*, donde iremos haciendo comentarios brevemente junto a las mismas líneas de código.

6 Conclusión y cierre

Con este artículo hemos vuelto a poner a disposición de los interesados en el desarrollo de aplicaciones *homebrew* para la NDS (las que utilizan el kit de desarrollo no oficial) un recurso que creemos es de utilidad para servir de apoyo a los que se inician en este complejo contexto de desarrollo de aplicaciones para plataformas de videoconsolas.

Las actualizaciones necesarias del código y la ubicación en un repositorio en *GitHub* son nuestra modesta aportación. Quiero agradecer desde aquí el trabajo de los autores que figuran en la bibliografía. Sirva este trabajo como homenaje a estos desarrolladores que han contribuido con su código y con sus explicaciones a que otros puedan adentrarse en este apasionante y difícil campo de desarrollo de aplicaciones para computadores con una arquitectura tan especializada y poco documentada.

Esperamos que el lector se anime a descargar el proyecto desde el *GitHub* [3], leer el original y probar las amplicaciones propuestas en el trabajo de Jose David [2]. Con el emulador DeSmuME [4] es posible ejecutar los desarrollos expuestos. Y, por supuesto, experimentar y modificar el código que se publica en el repositorio del proyecto.

7 Bibliografía

[1] Mukunda Johnson. Sitio web. Disponible en <http://mukunda.com/projects.html>.

[2] Jaén Gomariz, JD. (2015). Tutorial práctico para desarrollo de videojuegos sobre plataforma Nintendo NDS. Disponible en <http://hdl.handle.net/10251/56433>.

[3] Repositorio del proyecto Escena del tutorial "The bouncing ball". Disponible en <https://github.com/magusti/NDS-hombrew-development>.

[4] DeSmuME. Página web del proyecto. Disponible en <http://desmume.org/>.

[5] J. Amero (Patater). (2008). Introduction to Nintendo DS Programming. Disponible en <https://patater.com/files/projects/manual/manual.html>.

[6] Boudeville, O. (2008). A guide to homebrew development for the Nintendo DS. Disponible en <http://osdl.sourceforge.net/main/documentation/misc/nintendo-DS/homebrew-guide/HomebrewForDS.html >.

[7] F. Moya y M. J. Santofimia. (2011). Laboratorio de Estructura de Computadores empleando videoconsolas Nintendo DS. Ed. Bubok Publishing. ISBN. 978-84-9981-039-3.

[9] Grit. GBA Raster Image Transmogrifier. Disponible en <http://www.coranac.com/projects/grit >.

[10] Stair, S. (2006). GBATEK. Gameboy Advance / Nintendo DS - Technical Info. Disponible en <https://www.akkit.org/info/gbatek.htm>.