



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



Escola Tècnica  
Superior d'Enginyeria  
Informàtica

Escuela Técnica Superior de Ingeniería Informática  
Universitat Politècnica de València

## **Tutorial práctico para desarrollo de videojuegos sobre plataforma Nintendo NDS**

Proyecto Final de Carrera  
I.T. Informática de Sistemas

**Autor:** Jose David Jaén Gomariz  
**Director:** Jose Manuel Agustí Melchor  
29/09/2015

# Índice de contenido

1	Introducción.....	3
1.1	Presentación.....	3
1.2	Objetivos del trabajo.....	4
1.3	Material en el que me he basado.....	4
2	Instalación de programas necesarios.....	5
2.1	Preparativos.....	5
2.2	Instalación.....	7
3	Añadidos personales.....	8
3.1	Manipulación del texto y uso de Main y Sub Engines.....	8
3.2	Acceso al sistema de archivos de la NDS.....	17
4	Desarrollo.....	19
4.1	Preparación.....	19
4.2	Construyendo la escena.....	20
	Primera Parte del Resumen.....	20
	Primer añadido: cambiar el color del fondo.....	26
	Segunda Parte del Resumen.....	29
	Segundo añadido: mover, eliminar y cambiar la prioridad de los BGs.....	31
	Tercera Parte del resumen.....	34
4.3	Sprites.....	35
	Resumen.....	35
4.4	Avanzado.....	37
	Primera parte del resumen.....	37
	Primer añadido: Mover, escalar y rotar el sprite.....	41
	Segunda parte del Resumen.....	45
	Segundo añadido: Score de saltos, tiempo de juego y guardado en data.....	52
	Tercera Parte del resumen.....	57
5	Conclusiones y trabajos futuros.....	60
6	Bibliografía/webgrafía.....	61
7	Anexos.....	62

# 1 Introducción

En este proyecto que vamos a realizar llamado “**Tutorial práctico para el desarrollo de videojuegos sobre plataforma Nintendo DS**” nos vamos a centrar en el seguimiento de un tutorial para desarrollar un videojuego para la **Nintendo DS**.

La idea es mantener el enfoque del tutorial exponiendo detalles del hardware de la plataforma, junto a los propios de programación con las herramientas que ofrece el proyecto **devkitPro** [1] manteniendo un desarrollo práctico. Cuando se ha visto conveniente, se ha creado un ejemplo concreto para exponer parte del API de **devkitPro** [1]. Para ello seguiremos como guía el tutorial ***How to Make a Bouncing Ball Game*** [2], donde añadiremos contenido ya no accesible, y añadiendo nuevas funcionalidades que le den una interacción sobre lo que se está probando.

## 1.1 Presentación

En primer lugar, comprobaremos si el código sigue funcionando en la versión **1.5.4** (Es la última versión disponible al momento de comenzar este proyecto) de las librerías utilizadas en **devkitPro**[1] y encontrar una manera de solucionar cualquier problema hallado, si este se encuentra, cambiaremos el código obsoleto por uno nuevo que funcione, además añadiremos nuevo código para hacerlo más interactivo, basándonos en otros tutoriales y añadiendo trabajos prácticos personales. Para finalizar, acabaremos con una Demo técnica donde estarán incluidos todos los elementos descritos anteriormente.

El desarrollo se puede realizar sobre cualquier plataforma, se detallará la forma de instalar las librerías de desarrollo en plataformas **GNU/Linux** y **MS/Windows** y se comentarán posibles herramientas para llevarlo a cabo en cada caso.

Utilizaremos emuladores para el desarrollo y finalmente probaremos sobre la plataforma física que toda la funcionalidad está operativa.

A lo largo de este trabajo se mencionaran los archivos de código que recogen el desarrollo y que acompañarán a esta memoria del trabajo como material complementario, permitiendo así descargar este documento de muchas líneas de código repetido y ofrecer al lector una serie de ejemplos operativos.

El resto de la memoria se estructura entorno al eje principal de reconstruir el tutorial existente [2]. Para ello, se centrará en sus sucesivos capítulos.

En el capítulo uno, prepararemos todo los elementos principales tales como las librerías, el editor, los emuladores y las imágenes que necesitaremos.

En el capítulo dos, se aborda el comienzo del diseño del juego, empezando por el *backdrop*, los *backgrounds* (**BGs**) y los bancos **VRAM**.

En el capítulo tres, se describe la carga y renderizado de *sprites*.

En el capítulo cuatro, abordaremos la aritmética de punto fijo, los efectos del *sprite*, la manipulación del *sprite* y el sonido del juego.

## 1.2 Objetivos del trabajo

Los objetivos a realizar a lo largo del proyecto serán :

- Manejo de texto para presentar leyendas o información.
- Seguir el tutorial original y adaptarlo si hay algo obsoleto.
- Completar la parte del audio que no está disponible en el sitio web.
- Añadir la opción de acceso al sistema de ficheros para guardar información de una ejecución para la siguiente, con fines de configuración o de listado de las mejores puntuaciones.
- Explicar cómo se hace una sesión de depuración del código.
- Creación de una Demo funcional que muestre todo lo anterior.

Como objetivos secundarios, se van a crear un conjunto de ejemplos, independientes entre sí, que mostrarán una funcionalidad concreta y que permitirán al usuario variar algunos parámetros para entender su funcionamiento.

## 1.3 Material en el que me he basado

Como se ha explicado anteriormente, este proyecto está basado en el tutorial “**How to Make a Bouncing Ball Game**” [2] en el que se nos propone la creación de un juego basado en una bola que está continuamente rebotando, enseñando desde la creación de los *backgrounds* y *sprites* hasta su modificación y interacción con el entorno creado.

Los trabajos y prácticas de la asignatura Arquitectura y entorno de desarrollo de videojuegos (AEV) [3][4][5][6][7][8][9][10][11], me han ayudado a la hora de obtener información sobre la utilización de las librerías y las funciones necesarias para el proyecto.

Los ejemplos de **devkitPro** [1], que aunque la gran mayoría obsoletos, siempre han dado alguna pista de como hacer funcionar elementos de los añadidos y entenderlos mejor.

## 2 Instalación de programas necesarios

En este trabajo se asume que el lector tiene un cierto conocimiento de la consola y de las posibilidades de crear ejecutables para ella. De no ser así se sugiere la lectura de “Baby Steps in Nintendo DS “Homebrew”Hacking” [12] .

### 2.1 Preparativos

Aquí prepararemos lo mínimo necesario para poder llevar a cabo el proyecto de *“How to make a bouncy ball game”* [2]. Para ello necesitaremos un compilador, una librería, un emulador (en este caso podemos utilizar también una consola **Nintendo DS** pero por seguridad (y comodidad) es más aconsejable el emulador) y los gráficos a utilizar en el proyecto.

Para el compilador utilizaremos el “**devkitARM**”. Donde escogeremos la versión **devkitARM release 44 and libraries** que es la que utilizaremos en este proyecto. En concreto usaremos la versión de **Windows**, aunque explicaré como es la instalación para la versión de **Linux** ya que a la hora de programar, compilar y ejecutar los **.nds** en los emuladores escogidos no importará en que sistema operativo (**SO**) te encuentres.

Por la parte de **Windows** una vez accedamos a la página oficial del **devkitPro**[1] encontraremos el instalador en el rectángulo en negro que muestro en la fig. 1.

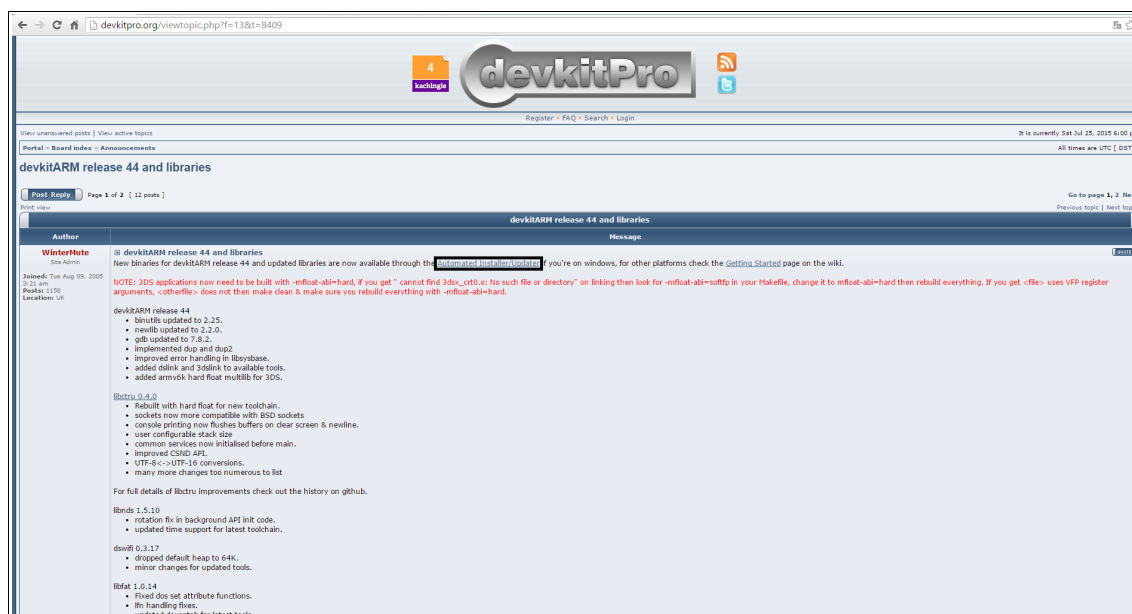


Fig. 1: Instalador Windows. Fuente [1].

Después seleccionamos en la siguiente ventana fig. 2 la versión **devkitProUpdater-1.5.4.exe** en el caso de **Windows** y **devkitARMupdate.pl** para **Linux**. Para la librería utilizaremos **Libnds**[13] que se trata de una librería de bajo nivel para programar en NDS.

devkitPro			
Homebrew toolchains for wii, gamecube, 3ds, ds, gba, gp32 and psp			
Brought to you by: <a href="#">wntrmute</a>			
Summary	Files	Reviews	Support
External Link	Mailing Lists	Wiki	Tickets
New			
Looking for the latest version? <a href="#">Download devkitProUpdater-1.5.4.exe (220.2 kB)</a>			
Home / Automated Installer			
Name	Modified	Size	Downloads / Week
Parent folder			
previous	2012-04-13		1
devkitARMupdate.pl	2014-11-24	6.6 kB	111
devkitProUpdater-1.5.4.exe	2014-11-20	220.2 kB	341
devkitProUpdater-1.5.3.exe	2012-04-13	220.8 kB	5
README.md	2012-04-11	372 Bytes	21
devkitPPCupdate.pl	2012-04-11	5.7 kB	17
Totals: 6 Items		453.7 kB	495

Fig. 2: Versión devkitPro

Y, finalmente, para el emulador tenemos 2 opciones para el emulador, **no\$gba**[14][15] que solo funciona en **Windows** y **DeSmuME**[16] que funciona tanto en **Windows** como en **Linux**:

En el caso de **no\$gba**[14][15], ya que en la página oficial el enlace está roto, tendremos que descargarlo de la página del creador donde descargaremos la versión **debug**. Este emulador, puede ejecutarse nativamente en **Windows** y con la herramienta **wine** puede ejecutarse en **Linux**.

Para **DeSmuME** [16] descargaremos la versión para el **SO**, que hayamos escogido.

Para los gráficos utilizaremos los que se encuentran en la carpeta materiales *\sprites*.

En la tabla 1 se muestran los componentes gráficos que utilizaremos en el proyecto aunque estos son meramente visuales, ya que están agrandados para que se vean los detalles.




Bouncy Ball sprite	Funny Shape	A gradient
		

Tabla 1: Gráficos proyecto original [6].

## 2.2 Instalación

Para instalar *devkitPro*[1] en **Windows**, una vez arranquemos el instalador **devkitProUpdater-1.5.4.exe**, es aconsejable instalarlo en una dirección lo más sencilla posible (básicamente en la raíz de algún disco duro como **C:\devkitPro** ya que las direcciones largas o complejas suelen dar problemas, luego seleccionaremos las siguientes opciones que se muestran en la fig. 3, que es lo necesario para el proyecto en **Nintendo DS**.

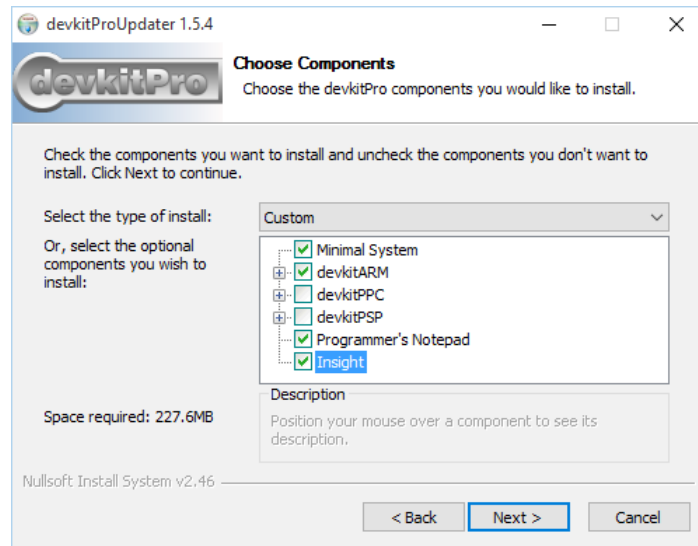


Fig. 3: Opciones del instalador

Para su instalación en **Linux**, ejecutaremos **devkitARMupdate.pl**, al terminar nos indicará que declaremos dos variables de sistema con un contenido similar a la tabla 2 también deberá declarar y exportar esas variables desde un terminal a (tabla 3), así como incluirlas en su *.bash\_profile* o *bashrc*, dependiendo de la distribución **UNIX** que utilices.

Una vez instalado, dentro del directorio raíz, comprobar si existe el subdirectorio *libnds* y dentro de este los subdirectorios *include* y *lib*.

```
Checking DEVKITPRO...Please set DEVKITPRO in your environment as
/home/usuario/devkitPro

Checking DEVKITARM...Please set DEVKITARM in your environment as

${DEVKITPRO}/devkitARM
```

Tabla 2: Variables sistema

```
$ export DEVKITPRO=${HOME}/devkitPro

$ export DEVKITARM=${DEVKITPRO}/devkitARM
```

Tabla 3: Donde exportar las variables

## 3 Añadidos personales

### 3.1 Manipulación del texto y uso de Main y Sub Engines

En este apartado, vamos a tratar un tema que no se ve en el proyecto de “*How to make a bouncy ball game*”[2], que es el tratamiento de texto que voy a utilizar en los añadidos de cada apartado, para mostrar la leyenda e información que pueda ser útil, además mostrare el uso de diferentes *engines* (*Main o Sub*) para mostrar como se crean.

Para ello, vamos a necesitar crear una nueva carpeta para este miniproyecto, en el que trataremos la manipulación y posicionamiento de cadenas de texto, tanto básicas como con fuentes personalizadas.

Primero, empezaremos desde la base, así que iremos dentro de la carpeta de la instalación **DevkitPro**, **devkitPro\examples\nds\templates\arm9** y copiaremos todo en una nueva carpeta, que usaremos para el miniproyecto. Borraremos el contenido del fichero *template.c* y lo cambiamos por este código básico mostrado en la tabla 4 con el que comenzaremos.

```
#include <nds.h>
#include <stdio.h>

int main(void) {

    while(1)
    {

        swiWaitForVBlank();

    }

}
```

Tabla 4: Código básico para texto

Desde aquí comenzaremos añadiendo partes del código mientras lo voy explicando más detalladamente, empezando por añadir las líneas de código que muestra la tabla 5 dentro de *main*, justo antes de llegar al *while*.

```
consoleDemoInit(); //Inicializa la consola de texto
consoleClear(); //Borra la pantalla
setBrightness(3, 0); //Establece el brillo de las dos pantallas
iprintf("Hola Mundo!"); //Imprime por la consola de texto
iprintf("Texto seguido\n"); //\n -> Salto de línea
iprintf("Texto en otra linea");
iprintf("\x1b[10;2HTexto en posicion elegida"); //\x1b[<linea>;<columna>H   linea entre 0-23 y
columna entre 0-31(Estan en Tiles)
```

Tabla 5: Inicialización consola y lineas de texto

Si le damos a compilar nos creará un **.nds**, si lo ejecutamos con **no\$gba**[14][15], tendremos en pantalla una imagen como la de la fig. 4. En ella se puede ver el contenido visualizado en las dos pantallas de la consola: en la superior el nivel de brillo la hace verse muy iluminada y, en la inferior, que es la única que ha inicializado la función “*consoleDemoInit*”, el texto.



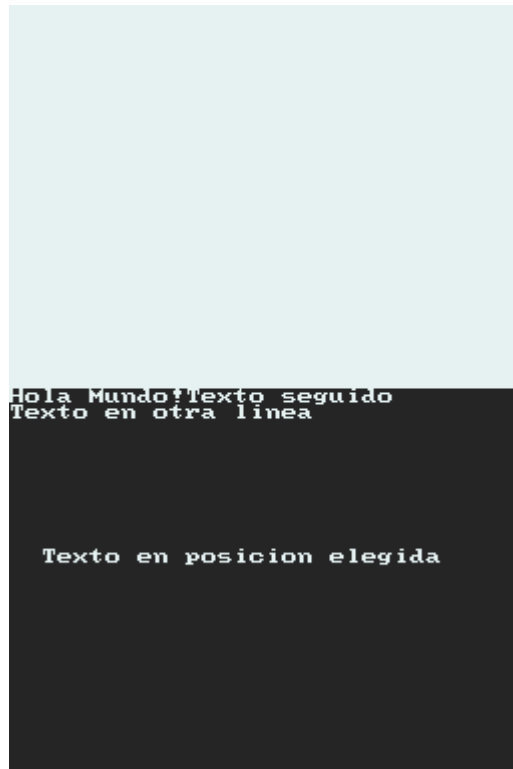


Fig. 4: Prueba texto1

```
//Inicializa la variable tipo PrintConsole
PrintConsole mainConsola, subConsola;

//Inicia el motor de gráficos 2D, imprescindible para mostrar por ejemplo texto.
videoSetMode(MODE_0_2D);//(main system)
videoSetModeSub(MODE_0_2D);//(sub system)
```

Tabla 6: Variable PrintConsole y selección de vídeo

Como podemos observar aquí hemos probado distintas formas de colocar un texto básico con la consola por defecto **consoleDemoInit()** y en diferentes posiciones mediante añadidos a las líneas del *iprint*, como el **\n** que permite el salto de línea para escribir en la línea siguiente, o el código siguiente **\x1b[10;2H** que nos permite, colocar la línea de texto en los *tiles* correspondientes al primer número (vertical) y al segundo número (Horizontal), desde donde comenzará a escribir la línea de texto.

Después de tratar texto de modo muy básico, pasaremos a algo un poco más complejo. Empezaremos primero con la utilización de diferentes *engines* para mostrar texto, tanto en la pantalla superior como en la pantalla inferior a la misma vez.

Para ello debemos borrar lo anteriormente escrito y dejar el código como al principio (tabla 4), luego añadimos las siguientes líneas de código de la tabla 6.

Aquí creamos las variables que utilizaremos cuando queramos cambiar de consola, y de paso seleccionamos los modos de vídeo tanto del *main system* como del *sub system*.

Ahora añadiremos la inicialización de las 2 consolas que vamos a utilizar con este código de la tabla 7.

```
consoleInit(&mainConsola, 3, BgType_Text4bpp, BgSize_T_256x256, 31, 0, true, true);
consoleInit(&subConsola, 3, BgType_Text4bpp, BgSize_T_256x256, 31, 0, false, true);
```

*Tabla 7: Inicialización consolas*

Dentro de *consoleInit()* cada dato representa de izquierda a derecha, 1- Consola a inicializar, 2- Capa **Bg** donde se imprimirá, 3- //Tipo de fondo, 4- //Tamaño del fondo, 5- //Base del mapa, 6- //Base del *tile* gráfico, 7- //Sistema gráfico a usar(main System) -> false(sub system) y 8- No cargar gráficos para la fuente.

Ahora, nos tocará seleccionar qué consola queremos utilizar y qué línea de texto ponerle, para ello seguidamente al código anterior escribiremos las líneas que muestra la tabla 8.

```
consoleSelect(&mainConsola);//Permite seleccionar la consola deseada
iprintf("\x1b[12;1HPueba Texto pantalla superior");

consoleSelect(&subConsola);//Permite seleccionar la consola deseada
iprintf("\x1b[12;1HPueba Texto pantalla inferior");
```

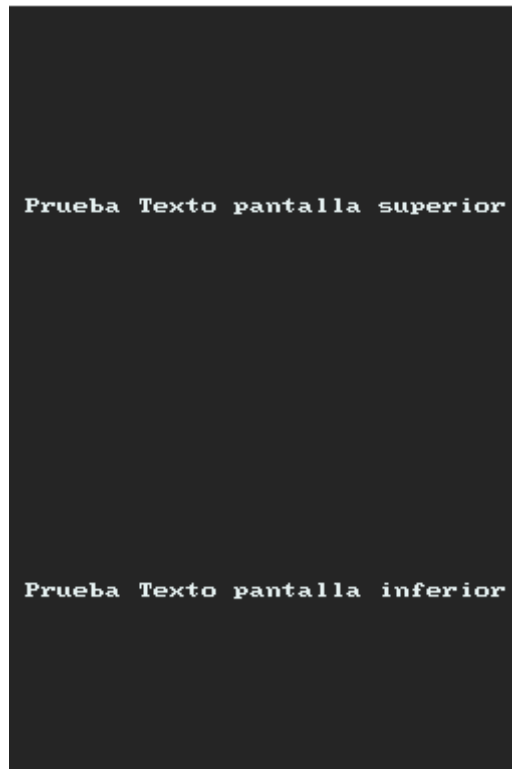
*Tabla 8: Selección Consola*

Para finalizar esta parte, le damos a compilar y ejecutamos el **.nds** resultante, debe mostrar una imagen como la figura 6.

Para la tercera prueba vamos a implementar una fuente distinta que la que viene por defecto en la librería, de esta forma podemos conseguir distintas formas de mostrar un texto. Pero primeramente crearemos una carpeta en nuestro proyecto de prueba, con el nombre de **gfx** y dentro copiaremos la fuente (fig. 5) que vamos a utilizar (puede ser la que pongo de muestra o una que hayáis creado).



*Fig. 5:  
Fuente*



*Fig. 6: Prueba texto2*

Lo primero será modificar ciertos aspectos del *makefile* o revisar que estén introducidos, en la siguiente tabla 9 muestra la parte del código que hay que revisar y la parte subrayada en amarillo la que hay que poner en caso de que no este, ya que permite que el programa lea los gráficos que se encuentren en ella.

TARGET	:=	\$(notdir \$(CURDIR))
BUILD	:=	build
SOURCES	:=	source
DATA	:=	data
INCLUDES	:=	include
GRAPHICS	:=	gfx

*Tabla 9: Makefile parte 1*

Para continuar, un poco más abajo encontramos el siguiente trozo de código (tabla 10), donde tenemos que revisar que las líneas subrayadas en amarillo estén, ya que son necesarios para procesar los archivos gráficos **.bmp** que se encuentren dentro de la carpeta gfx.

```

export VPATH := $(foreach dir,$(SOURCES),$(CURDIR)/$(dir)) \
               $(foreach dir,$(DATA),$(CURDIR)/$(dir)) \
               $(foreach dir,$(GRAPHICS),$(CURDIR)/$(dir))

export DEPSDIR := $(CURDIR)/$(BUILD)

CFILES      := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.c)))
CPPFILES    := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.cpp)))
SFILES      := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.s)))
BINFILES    := $(foreach dir,$(DATA),$(notdir $(wildcard $(dir)/*.*)
BMPFILES    := $(foreach dir,$(GRAPHICS),$(notdir $(wildcard $(dir)/*.bmp)))

```

Tabla 10: Makefile parte 2

Seguimos un poco mas abajo hasta el siguiente trozo de código (tabla 11), otra vez revisamos si esta el texto subrayado y si no, lo introducimos, ya que nos permite pasar todos los **.bmp** a **.o**.

```

export OFILES := $(addsuffix .o,$(BINFILES)) \
                 $(BMPFILES:.bmp=.o) \
                 $(CPPFILES:.cpp=.o) $(CFILES:.c=.o) $(SFILES:.s=.o)

```

Tabla 11: Makefile parte 3

Finalmente solo queda revisar la parte del código donde esta la conversión del *grit* (tabla 12), donde revisaremos que exista la línea para la conversión de *grit*.

```

#-----
# This rule creates assembly source files using grit
# grit takes an image file and a .grit describing how the file is to be processed
# add additional rules like this for each image extension
# you use in the graphics folders
#-----
%.s %.h : %.bmp %.grit
#-----

grit $< -fts -o$*

```

Tabla 12: Makefile parte 4

Una vez preparado el *makefile* toca crear el archivo **font.grit** que permitirá al programa leer la **fuentes.bmp** que hemos preparado, para ello crearemos un nuevo fichero vacío donde escribiremos las siguientes líneas de la tabla 13, no me meteré en que es cada cosa, si queréis saber cuantas opciones permite *grit* podéis verlas aquí [7].

```
#-----
# graphics in tile format
#-----
-gt

#-----
# output first 16 colors of the palette
#-----
-pw16

#-----
# no tile reduction
#-----
-mR!

#-----
# no map output
#-----
-m!

#-----
# graphics bit depth is 4 (16 color)
#-----
-gB4
```

*Tabla 13: Grit del archivo gráfico que define la fuente de letra a utilizar.*

Una vez preparado lo anterior, comenzaremos añadiendo el *header* de la fuente con esta línea de la tabla 14.

```
#include "font.h"
```

*Tabla 14: Include del Header de la fuente*

Y seguido, como en la anterior vez volvemos al código inicial (tabla 4), después comenzamos a introducir estas líneas de código de la tabla 15.

```
//Inicia el motor de gráficos 2D, imprescindible para mostrar por ejemplo texto.
videoSetMode(MODE_0_2D); //(main system)
videoSetModeSub(MODE_0_2D); //(sub system)

//Asignación del Banco de Memoria C al sub system BG
vramSetBankC(VRAM_C_SUB_BG);
```

*Tabla 15: Asignación VRAM y selección de vídeo*

Como podemos observar, hemos introducido una nueva línea *vramSetBankC()*, que nos permitirá asignar la nueva fuente en la memoria **VRAM**.

Luego, será necesario crear tanto la variable de la consola a utilizar, como de la fuente que gastaremos. Para ello introduciremos seguidamente estas líneas de la tabla 16.

```
PrintConsole laConsola; //Inicializa la variable tipo PrintConsole
ConsoleFont fuente; //Inicializa la variable tipo ConsoleFont
```

*Tabla 16: Variables PrintConsole y ConsoleFont*

Una vez introducidas, lo siguiente es inicializar la consola como hemos hecho anteriormente con esta

línea de la tabla 17.

```
consoleInit(&laConsola, 0, BgType_Text4bpp, BgSize_T_256x256, 20, 0, false, false);
```

*Tabla 17: Incialización de la consola*

Y por fin llegamos a la parte nueva, por así decirlo, donde nos toca configurar la nueva fuente en base al **.bmp** que queremos usar, para ello añadiremos estas líneas de código de la tabla 18.

```
fuelle.gfx = (u16*)fontTiles;  
fuelle.pal = (u16*)fontPal;  
fuelle.numChars = 95;  
fuelle.numColors = fontPalLen / 2;  
fuelle.bpp = 4;  
fuelle.asciiOffset = 32;  
fuelle.convertSingleColor = false;
```

*Tabla 18: Atributos ConsoleFont*

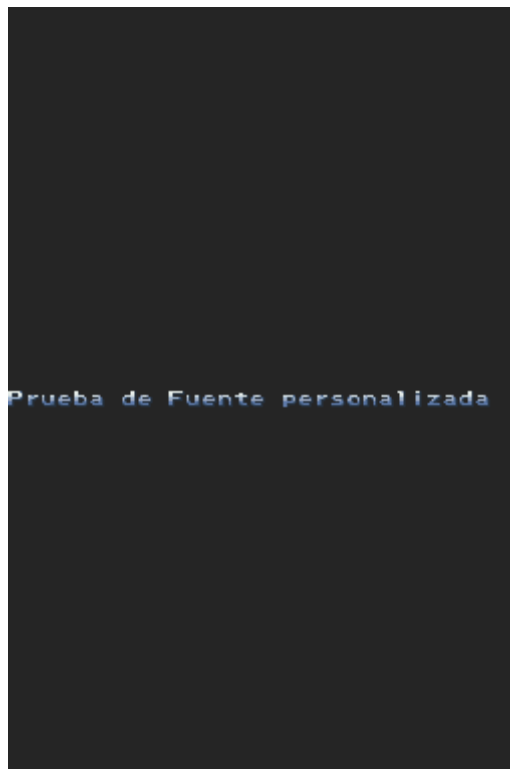
Todos estos atributos están explicados en *ConsoleFont Struct Reference* de la librería **console.h** que se encuentra en *libnds* [2].

Para finalizar añadimos el código para asignar la fuente que hemos escogido a la consola que utilizaremos y el texto a mostrar mostrado en la tabla 19.

```
consoleSetFont(&laConsola, &fuelle);  
iprintf("Prueba de Fuente personalizada");
```

*Tabla 19: Asignar Fuente a la Consola, más línea de texto*

El resultado sería como esta imagen de la fig. 7.



*Fig. 7: Prueba texto3*

Para la cuarta y última prueba, cogeremos todo lo anteriormente, más la manipulación del texto.

Para ello, aprovechando el código final del anterior apartado, comenzaremos añadiendo las variables que vamos a utilizar. Añadimos las nuevas variables, donde añadiremos también la **subconsola** pasando a llamar **laConsola** como **mainConsola** código de la tabla 20.

```
const int tile_base = 0;
const int map_base = 20;

PrintConsole mainConsola, subConsola; //Inicializa la variable tipo PrintConsole

ConsoleFont fuente; //Inicializa la variable tipo ConsoleFont
```

*Tabla 20: Variables*

Luego, cambiamos la iniciación del modo *subsystem* para permitir modificaciones del texto (tabla 21).

```
//Inicia el motor de gráficos 2D, imprescindible para mostrar por ejemplo texto.
videoSetMode(MODE_0_2D); // (main system)
videoSetModeSub(MODE_5_2D); // (sub system)
vramSetBankC(VRAM_C_SUB_BG);
```

*Tabla 21: Inicializando vídeo y asignando VRAM*

Ahora inicializaremos la subconsola con este código de la tabla 22.

```
consoleInit(&subConsola, 3, BgType_ExRotation, BgSize_ER_256x256, map_base, tile_base, false, false);
```

*Tabla 22: Inicialización SubConsola*

Modificamos la asignación de consola y fuente, código de la tabla 23.

```
consoleSetFont(&laConsola, &fuente);
```

*Tabla 23: Anterior*

Por la nueva con la subConsola, código tabla 24.

```
consoleSetFont(&subConsola, &fuente);
```

*Tabla 24: Posterior*

A partir de aquí, ya cambia la cosa. Primero eliminamos el *iprint()*, y nos preparamos para seleccionar la *mainConsola*, escribir texto en ella y luego pasar a la subconsola donde escribiremos otro texto. Todo gracias al método *consoleSelect()*, que tendremos que añadir con este código de la tabla 25.

```

consoleSelect(&mainConsola); //Permite seleccionar la consola deseada
iprintf("\n\n\n\n");
iprintf("\tBoton A: Disminuye Ancho\n\n");
iprintf("\tBoton B: Aumenta Ancho\n\n");
iprintf("\tBoton X: Disminuye Alto\n\n");
iprintf("\tBoton Y: Aumenta Alto\n\n");
iprintf("\tBoton L: Rota a Izquierda\n\n");
iprintf("\tBoton R: Rota a Derecha\n\n");
iprintf("\tPad: Mueven la frase\n");

consoleSelect(&subConsola); //Permite seleccionar la consola deseada
iprintf("\x1b[10;5HPrueba de Movimiento");
iprintf("\x1b[12;5HRotacion y Escalado");

```

*Tabla 25: Selección de la pantalla y texto*

Para el control del texto necesitaremos crear una serie de variables mostradas en la tabla 26 que nos permitirán escalarlo, moverlo y rotarlo .

```

unsigned int angle = 0;
int scrollX = 0;
int scrollY = 0;
int scaleX = intToFixed(1,8);
int scaleY = intToFixed(1,8);

```

*Tabla 26: Variables para el manejo del texto*

Dentro del *while* escribiremos todo lo relacionado con la lectura de *inputs* y los cambios que estos generarán en el texto, código tabla 27.

```

scanKeys();
u32 keys = keysHeld();

if ( keys & KEY_L ) angle+=64;
if ( keys & KEY_R ) angle-=64;

if ( keys & KEY_LEFT ) scrollX++;
if ( keys & KEY_RIGHT ) scrollX--;
if ( keys & KEY_UP ) scrollY++;
if ( keys & KEY_DOWN ) scrollY--;

if ( keys & KEY_A ) scaleX++;
if ( keys & KEY_B ) scaleX--;

if ( keys & KEY_X ) scaleY++;
if ( keys & KEY_Y ) scaleY--;

```

*Tabla 27: Código para leer los cambios en el texto*

Por último, después del *swiWaitForBlank()* necesitaremos añadir el código de la tabla 28 que, dados los datos obtenidos en el código anterior (tabla 27) nos permitirá, escalar, mover y rotar el texto.

```

//Asigna al BG seleccionado un ángulo y una escala en X,Y
bgSetRotateScale(bg3, angle, scaleX, scaleY);

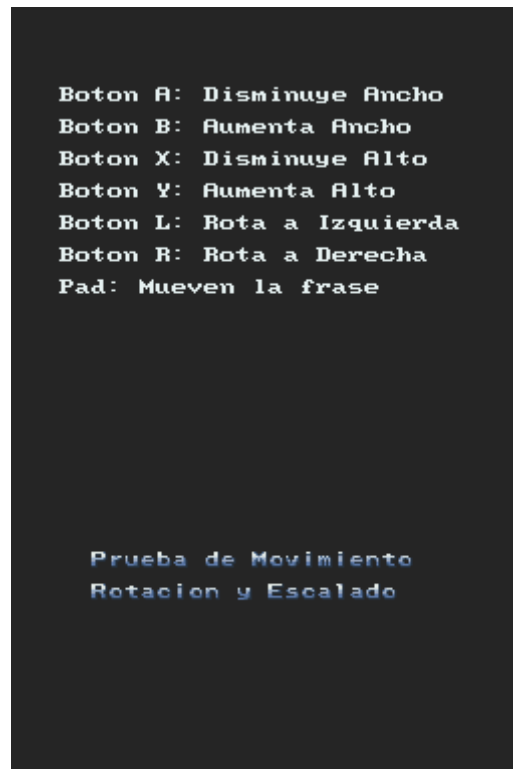
bgSetScroll(bg3, scrollX, scrollY); //Permite mover el bg a otras posiciones
bgUpdate(); //actualiza el BG

```

*Tabla 28: Código modificación bg*



El resultado fig. 8 nos permitirá modificar el texto conforme a los comandos que se muestran en la pantalla superior.



*Fig. 8: Prueba texto4*

### 3.2 Acceso al sistema de archivos de la NDS

Para el acceso, tanto en lectura como escritura de archivos, ajenos a la **ROM**, **devkitPro** nos permite ver dos opciones, una es *nitrofs* y la otra *fat*. De las dos *nitrofs* no soporta escritura, así que solo sirve para lectura, mientras tanto *fat* soporta ambas, aunque para ello crea ficheros nuevos. En **Linux** con **DeSmuMe**[16] y *fat* se puede simular una tarjeta de memoria, llamando al emulador, desde la dirección donde se encuentra el **.nds** con la siguiente línea de la tabla 29, en la terminal de **Linux**.

```
$ desmume --cflash-path=./ libfatrw.nds
```

*Tabla 29: Llamada al ejemplo fat con DeSmuMe*

Dando como resultado, la siguiente imagen de la fig. 9.



Fig. 9: Prueba sistema de archivos fat

Como vemos en el resultado, *fat* se inicializa correctamente con con la comprobación *fatInitDefault()*, luego abre el fichero *test.txt* donde lee y escribe en la pantalla la línea de texto que contiene “Jo mateix” para posteriormente escribir dentro de *test.txt* la palabra “Manolo”. Finalmente accede en modo binario, primero escribiendo “0x78” en *Tetris.sav*, para leer lo guardado obteniendo el dato de el.

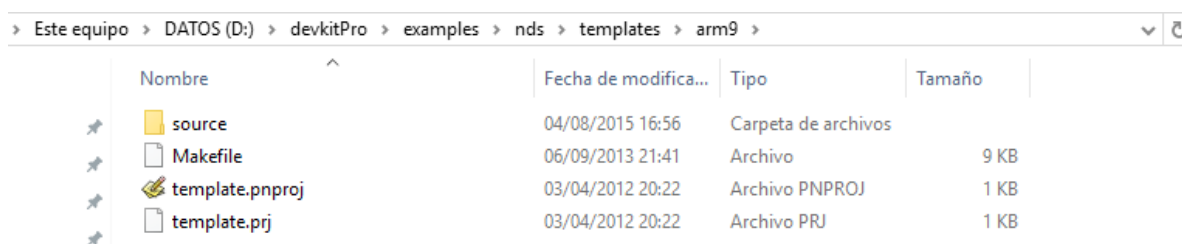
## 4 Desarrollo

Para mucha más información sobre el proyecto base de “*How to make a bouncy ball game*”[2], es aconsejable ver mejor el proyecto original donde se explica todo más detalladamente, sobre todo esta parte de el, que es casi en su totalidad mucha teoría sobre las características de la programación en **Nintendo DS**.

El tutorial original se desarrolla en base a cuatro apartados (Preparation, Building the Scene, Sprites y Advanced) que aquí mantendremos y sobre los que se indicarán los cambios introducidos o los ejemplos independientes desarrollados.

### 4.1 Preparación

Para prepara nuestro proyecto basado en el trabajo de “*How to make a bouncy ball game*”[2] tendremos que meternos en la siguiente carpeta dentro de la instalación de **DevkitPro**, **devkitPro\examples\nds\templates\arm9** fig. 10.



Nombre	Fecha de modifica...	Tipo	Tamaño
source	04/08/2015 16:56	Carpeta de archivos	
Makefile	06/09/2013 21:41	Archivo	9 KB
template.pnproj	03/04/2012 20:22	Archivo PNPROJ	1 KB
template.prj	03/04/2012 20:22	Archivo PRJ	1 KB

Fig. 10: Template arm9

Y copiamos el contenido en la carpeta que crearemos para el proyecto (aconsejo crear la carpeta para el proyecto en una ruta igual de sencilla que la de la instalación del **DevkitPro**[1]).

Luego abrimos el archivo *template.pnproj* y en *main.c* borramos el código que viene y lo sustituys por este de la tabla 30.

```
#include <nds.h> //definiciones de libnds(librerías de nds)
int main( void ){
    //irqEnable → Habilita el manejador de interrupciones
    //IRQ_VBLANK → (vertical blanking) es decir sincronización vertical
    irqEnable( IRQ_VBLANK )
    while(1) //Bucle que simula el funcionamiento de un juego.
    {
        swiWaitForVBlank(); //Espera a sincronización vertical
    }
    return 0;
}
```

Tabla 30: Código Básico

Así creamos el *main* básico para nuestro programa. Esta parte se corresponde a la carpeta **BouncingBallFase 0**.

## 4.2 Construyendo la escena

### Primera Parte del Resumen

Primero en el apartado de **diseño** se nos explica como comenzar la codificación inicial en base a los elementos de la **Nintendo DS** que engloban las pantallas, botones, el táctil, los motores de visualización sus modos de vídeo que muestro en esta tabla 31.

Modo	BG0	BG1	BG2	BG3
0	Texto/3D	Texto	Texto	Texto
1	Texto/3D	Texto	Texto	Affine
2	Texto/3D	Texto	Affine	Affine
3	Texto/3D	Texto	Texto	Extendido
4	Texto/3D	Texto	Affine	Extendido
5	Texto/3D	Texto	Extendido	Extendido
6	3D	No usado	Grande	No usado

Tabla 31: Modos BGs

Donde se explica los 4 *backgrounds* que podemos usar y sus características dependiendo el modo de vídeo utilizado. También no muestra una imagen fig. 11 donde podemos apreciar los diferentes *backgrounds* del proyecto:

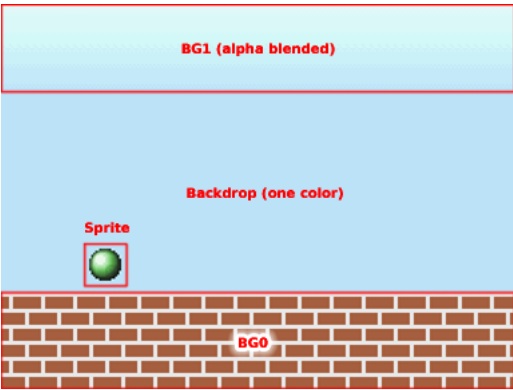


Fig. 11: Imagen Final Proyecto

Donde nos explica a que corresponde cada **BG** y luego los diferentes modos de visualización mostrados en esta tabla 32.

Modo de Visualización	Descripción
0	Visualización apagada. La pantalla se muestra en blanco. Sigue procesando pero no lo muestra.
1	Visualización normal (fondos y sprites)
2	Visualización VRAM(vista del mapa de bits del bloque de memoria de la VRAM)
3	Visualización de la memoria principal(vista del mapa de bits del bloque de memoria de la RAM principal)

Tabla 32: Modos Visualización

En el apartado de **flujo de programa** nos explica la teoría para entender como funciona el ciclo de renderizado, sus estados y los periodos de borrado tanto vertical como horizontal. Para evitar cosas como esta 12.

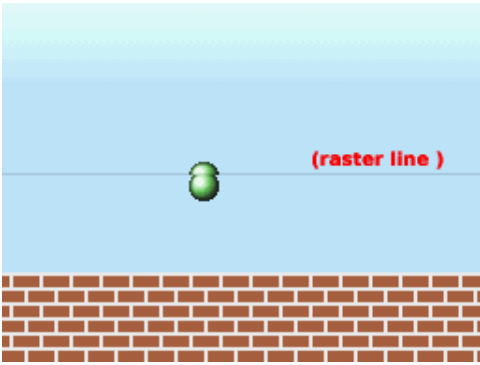


Fig. 12: Bug imagen

Para ello necesitaremos añadir las siguientes líneas de la tabla 33 de código a nuestro *main*, para dejarlo preparado a las nuevas funciones que vamos a necesitar en este proyecto.

```
int main( void )
{
    irqEnable( IRQ_VBLANK );    //Habilitar interrupcion vblank
    setupGraphics();
    while(1)
    {
        //Periodo de Renderizado
        //Actualizacion objetos del juego (Moverlos alrededor,
        //calcular velocidades, etc)
        update_logic();

        //Espera para periodo vblank
        swiWaitForVBlank();

        //Periodo vblank: (seguro modificar graficos)
        //mover graficos alrededor
        update_graphics();
    }
}
```

Tabla 33: Configuración Código Básico

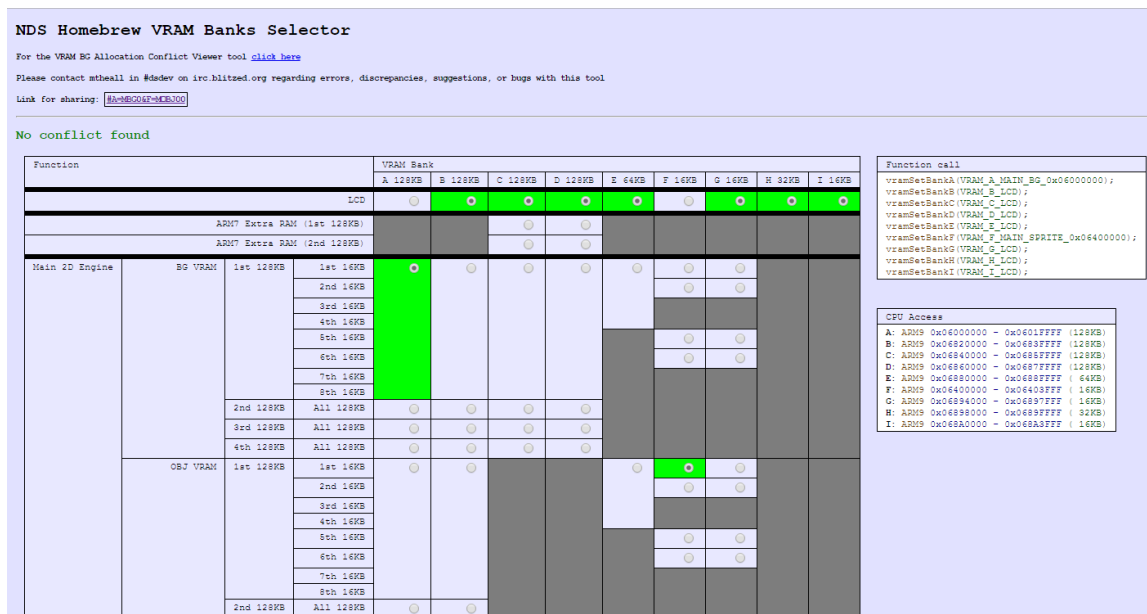


Fig. 13: Capacidades de la VRAM

En la parte de **Bancos VRAM** se nos explica la teoría de la configuración de la **RAM** en la **Nintendo DS**, para qué sirve cada banco de memoria y sus propósitos. Explicado cada función o propósito de cada uno de ellos. Si necesitamos más información y además saber cuánto ocupa o cómo se va a distribuir los datos en la **VRAM** de la **Nintendo DS** en estas páginas puedes verlo utilizando el **VRAM Bank Selector** [18]. Estas dos imágenes fig. 13 Y fig. 14 muestran la información que nos ofrece .

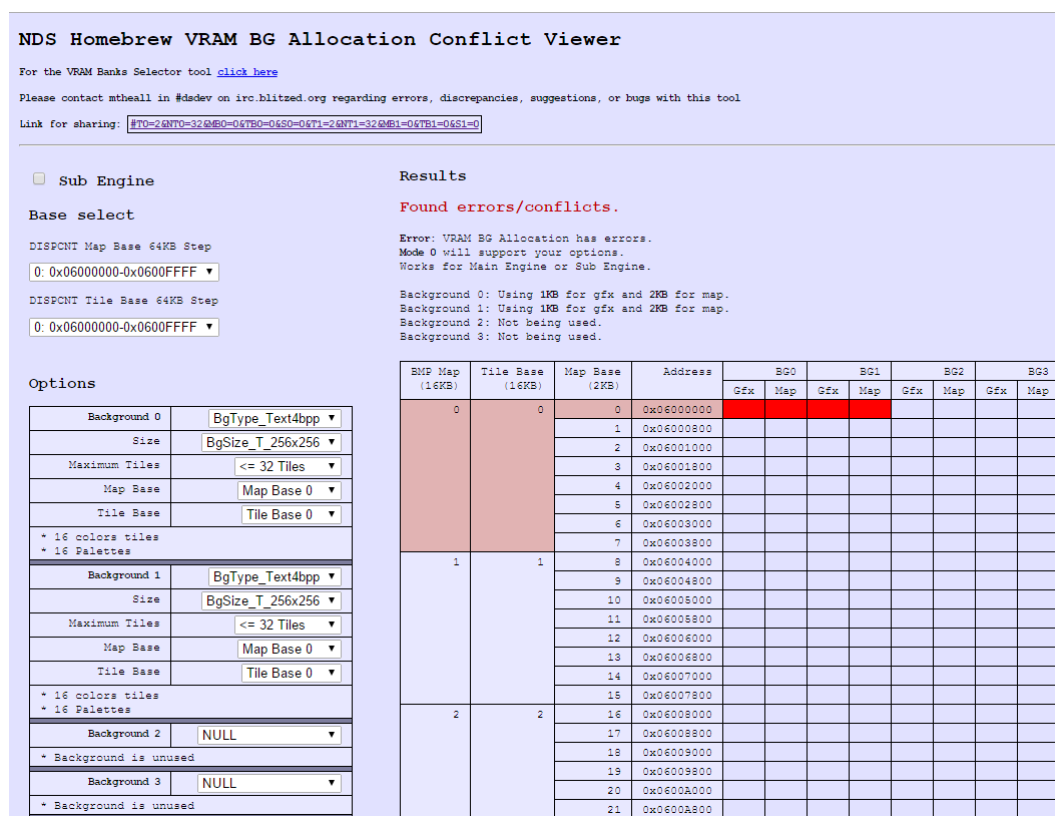


Fig. 14: Registros VRAM

Luego en **Conversión de Gráficos** se nos explica como **GRIT**[18] permite convertir los gráficos que vamos a utilizar en un formato que la **Nintendo DS** puede leer. En este apartado tendremos que crear los *.grit* de cada elemento que usaremos para los *backgrounds* y el *sprite* de la bola, es decir los ladrillos y el gradiente. Para ello crearemos nuevos archivos llamados *gfx\_brick.grit* (tabla 34), *gfx\_gradient.grit* (tabla 35) y *gfx\_ball.grit* (tabla 36).

**gfx\_brick.grit:**

```
# <- this is a comment! →  
# the -pn{n} rule specifies the number of entries in the palette  
# 3 colors  
-pn3  
# the -gB{n} rule specifies the output graphic format  
# 4-bit (16 color palette)  
-gB4
```

*Tabla 34: Código Grit para Ladrillos*

**gfx\_gradient.grit:**

```
# the gradient has 16 colors  
-pn16  
# the gradient is a 4-bit image  
-gB4
```

*Tabla 35: Código Grit para Gradiente*

**gfx\_ball.grit:**

```
# convert ball graphic  
# 16 colors  
-pn16  
# 4bpp  
-gB4
```

*Tabla 36: Código Grit para la Bola*

Para vincular los nuevos gráficos necesitaremos darle algunas reglas a nuestro *makefile* para decirle al *grit* como convertir nuestros gráficos. Para ello primero, crearemos una carpeta dentro de nuestro proyecto con el nombre **gfx**, donde introduciremos tanto los **png** como los **grit** recién creados. Luego en el *makefile* añadiremos las siguientes líneas:

Nos desplazamos hasta esta sección mostrada en la tabla 37.

```

CFILES    := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.c)))
CPPFILES  := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.cpp)))
SFILES    := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.s)))
BINFILES  := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.bin)))

```

Tabla 37: Sección 1 Makefile

Y añadimos esta línea de la tabla 38 para la conversión de los **PNG**.

```

PNGFILES := $(foreach dir,$(SOURCES),$(notdir $(wildcard $(dir)/*.png)))

```

Tabla 38: Conversión PNG

En la sección de **OFILES**, añadimos la lista de archivos **PNG**, tabla 39.

```

export OFILES := $(PNGFILES:.png=.o) $(BINFILES:.bin=.o) \
$(CPPFILES:.cpp=.o) $(CFILES:.c=.o) $(SFILES:.s=.o)

```

Tabla 39: Conversión .png a .o

Si nos desplazamos hacia abajo hasta justo debajo de la sección *main targets*, tendremos que añadir esta regla especial de *grit* mostrada en la tabla 40 para la conversión de los **PNG**.

```

#-----# GRIT rule for converting the PNG
files
#-----
%.s %.h : %.png %.grit
#-----
@grit $< -fts

```

Tabla 40: Código Grit

Ahora crearemos la función *setupGraphics()* que previamente habíamos definido en el *main* y añadimos las siguientes líneas de la tabla 41 que nos permiten asignar 2 bancos de memoria, uno para los *backgrounds* y otro para los *sprites*.

```

void setupGraphics( void )
{
    vramSetBankE( VRAM_E_MAIN_BG );
    vramSetBankF( VRAM_F_MAIN_SPRITE );
    ...
}

```

Tabla 41: Reserva de banco VRAM

Seguido se nos explica como sería una visualización de un banco de **VRAM** y lo que es un *tile*.

También se nos explica la función *dmaCopyHalfWord()* que usaremos después.

Pero primero vamos a añadir más código, en este caso referencias para el *grit* y a la carga de datos, código de la tabla 42.



```

//Referencias Graficas
#include "gfx_ball.h"
#include "gfx_brick.h"
#include "gfx_gradient.h"

//Entradas tile
#define tile_empty          0 //tile 0 = empty
#define tile_brick          1 //tile 1 = brick
#define tile_gradient       2 //tile 2 = gradient

//macro para calcular memoria BG VRAM
//direccion con el indice de tile
#define tile2bgram(t) (BG_GFX + (t) * 16)

```

Tabla 42: Referencias

Luego necesitaremos ampliar la función *setupGraphics()* para poder copiar los gráficos en la **VRAM**, código de la tabla 43. Seguidamente viene una explicación de algunas cosas del código que hemos introducido anteriormente.

```

void setupGraphics( void )
{
    vramSetBankE( VRAM_E_MAIN_BG );
    vramSetBankF( VRAM_F_MAIN_SPRITE );

    //generar el primer banco de tile por borrado a cero
    int n;
    for( n = 0; n < 16; n++ )
    {
        BG_GFX[n] = 0;
    }
    //Copiando graficos
    dmaCopyHalfWords( 3, gfx_brickTiles, tile2bgram( tile_brick ), gfx_brickTilesLen );
    dmaCopyHalfWords( 3, gfx_gradientTiles, tile2bgram( tile_gradient ), gfx_gradientTilesLen );
    ...
}

```

Tabla 43: Código de *dmaCopyHalfWords*

Después de la carga de gráficos, toca también todo lo relacionado con la paleta, que tras una descripción y explicación sobre ella, tenemos que añadir a nuestro *main* el siguiente código de la tabla 44.

```

//Paletas de entrada
#define pal_bricks          0 //paleta brick (entrada 0->15)
#define pal_gradient       1 //paleta gradient (entrada 16->31)

#define backdrop_colour    RGB8( 190, 255, 255 )
#define pal2bgram(p)      (BG_PALETTE + (p) * 16)

```

Tabla 44: Definición de la Paleta

Y en *setupGraphics()* este otro de la tabla 45.

```
//Paleta direccionada a la memoria de paleta
dmaCopyHalfWords( 3, gfx_brickPal, pal2bgram( pal_bricks ), gfx_brickPalLen );
dmaCopyHalfWords( 3, gfx_gradientPal, pal2bgram( pal_gradient ), gfx_gradientPalLen );

//Asignar Color de fondo
BG_PALETTE[0] = backdrop_colour;
```

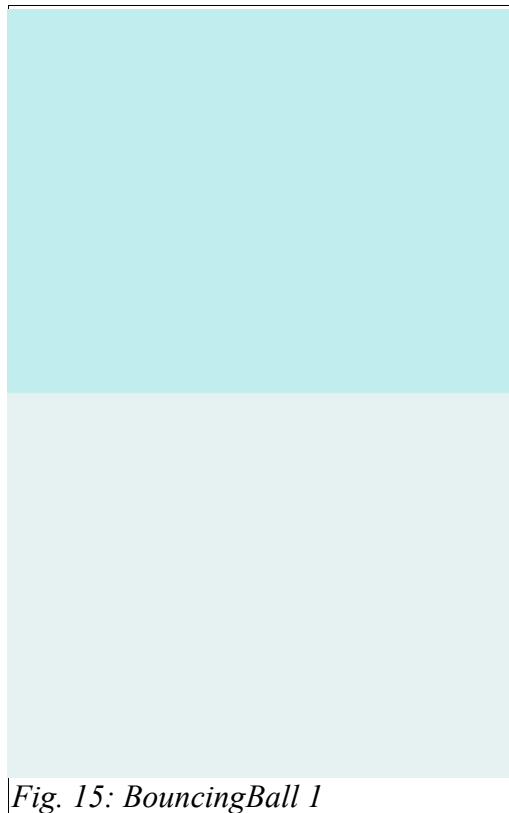
*Tabla 45: Código paleta de fondo*

Finalmente añadimos el modo de vídeo que utilizaremos con la siguiente línea de la tabla 46, dentro de *setupGraphics()*.

```
videoSetMode( MODE_0_2D );
```

*Tabla 46: Código modo de video*

Y ya podemos compilar y ver el resultado en el archivo generado **.nds**, mostrándonos este resultado de la fig. 15.



*Fig. 15: BouncingBall 1*

Esta parte se Corresponde a la carpeta **BouncingBallFase 1**.

### **Primer añadido: cambiar el color del fondo**

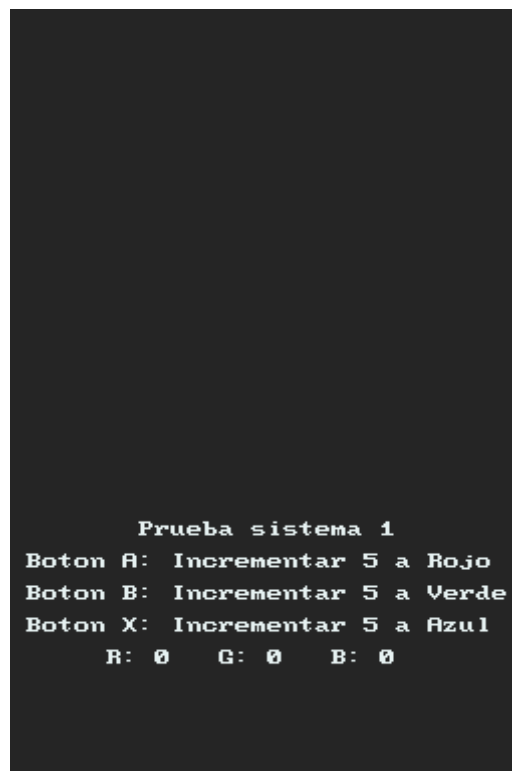
Para darle más funcionalidad al proyecto, he añadido en esta parte del proyecto la forma de modificar el *backdrop* (se trata del fondo de pantalla, independiente de los BGs) mediante las teclas **A**, **B** y **X**. Así como el aprovechamiento de la pantalla inferior para mostrar texto informativo de los cambios producidos.

Por lo tanto, he tenido que añadir una serie de código nuevo y la necesidad de añadir la función *updateGraphics()*, que ya habíamos definido dentro del bucle. Primero añadimos las variables para el manejo de la entrada de los botones y las variables que guardaran los valores del color, código de la tabla 47.

```
u32 keys;  
int red = 0;  
int green = 0;  
int blue = 0;
```

*Tabla 47: Variables*

A continuación, necesitaremos definir cómo va a quedar nuestro *updateGraphics()*. Para ello vamos a necesitar añadir la distintas entradas de botones, definiendo cada una de ellas y la información que vamos a necesitar añadir, para que sea más fácil ver lo que pasa y que botón utilizar para cada necesidad. En definitiva vamos a añadir este código de la tabla 48 a nuestro proyecto.



```
Prueba sistema 1  
Boton A: Incrementar 5 a Rojo  
Boton B: Incrementar 5 a Verde  
Boton X: Incrementar 5 a Azul  
R: 0 G: 0 B: 0
```

*Fig. 16:Añadido 1.0.5*

```

void update_graphics()
{
    /**Añadido al original**/
    BG_PALETTE[0] = RGB8( red, green, blue );//Añadido Para cambiar el color de fondo

    scanKeys();

    int keys = keysDownRepeat(); // Se lee el estado actual de todos los botones.

    if(KEY_A & keys) //Solo si se pulsa el boton A
    {
        red = red + 5;
        if(red > 255)
        {
            red = 0;
        }
    }
    if(KEY_B & keys) //Solo si se pulsa el boton B
    {
        green = green + 5;
        if(green > 255)
        {
            green= 0;
        }
    }
    if(KEY_X & keys) //Solo si se pulsa el boton X
    {
        blue = blue + 5;

        if(blue > 255)
        {
            blue = 0;
        }
    }

    consoleDemoInit();

    //Texto a representar por defecto se procesan en la pantalla inferior
    iprintf("\x1b[8;8HPrueba sistema 1");//\x1b[<linea>;<columna>H linea entre 0-23 y
columna entre 0-31
    iprintf("\x1b[10;1HBoton A: Incrementar 5 a Rojo");
    iprintf("\x1b[12;1HBoton B: Incrementar 5 a Verde");
    iprintf("\x1b[14;1HBoton X: Incrementar 5 a Azul");
    iprintf("\x1b[16;6HR: %d", red);
    iprintf("\x1b[16;13HG: %d", green);
    iprintf("\x1b[16;20HB: %d", blue);
    /**Añadido al original**/
}

```

Tabla 48: Código *updateGraphics*

Una vez acabemos debería mostrarnos una pantalla igual a la fig. 16. Donde podemos apreciar, que cada vez que pulsamos un botón de los mostrados, se incrementa el valor al cual esta asociado en 5, permitiendo una mejor variedad a la hora de escoger nuestro color de *backdrop*.

Esta parte se Corresponde a la carpeta **BouncingBallFase 1.0.5**.

## Segunda Parte del Resumen

Para empezar esta parte, nos explica la configuración de los 4 registros **BGxCNT** que posee la **NintendoDS**, en el que cada uno de ellos lleva el control de un *background*. A continuación, nos da una breve información de cada uno de los ajustes en el registro **BGxCNT**. Al llegar al apartado de configuración de los controles, explica un poco la configuración que necesitarán nuestros 2 *backgrounds*. Por tanto tendremos que añadir las siguientes líneas de la tabla 49.

```
// libnds prefixes the register names with REG_BGxCNT
REG_BG0CNT = BG_MAP_BASE(1);
REG_BG1CNT = BG_MAP_BASE(2);
```

Tabla 49: Registros BG

Luego, hay que rellenar el *tilemap* por lo que toca añadir dos definiciones más mostradas en la tabla 50.

```
#define bg0map ((u16*)BG_MAP_RAM(1))
#define bg1map ((u16*)BG_MAP_RAM(2))
```

Tabla 50: Tilemap

Seguidamente nos explica un poco como va configurar el *tilemap* para que los ladrillos se dibujen de forma correcta. Pero primero, tocara poner el *tilemap* a 0, así que tenemos que escribir estas líneas de la tabla 51.

```
int n;
for( n = 0; n < 1024; n++ ) bg0map[n] = 0;
```

Tabla 51: Tilemap borrado a 0

Segundo, dibujar una imagen de 32x6 ladrillos, código mostrado en la tabla 52.

```
int x, y;
for( x = 0; x < 32; x++ )
{
    for( y = 0; y < 6; y++ )
    {
        // magical formula to calculate if the tile needs to be flipped.
        int hflip = (x & 1) ^ (y & 1);

        // set the tilemap entry
        bg0map[x + y * 32] = tile_brick | (hflip << 10) | (pal_bricks << 12);
    }
}
```

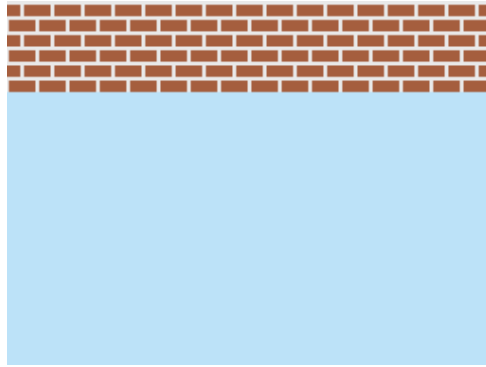
Tabla 52: Código para dibujar los ladrillos

Y finalmente añadimos a **videoSetMode()** este código de la tabla 53 para poder mostrar el nuevo **BG**.

```
videoSetMode( MODE_0_2D | DISPLAY_BG0_ACTIVE );
```

Tabla 53: Configuración *videoSetMode()*

Al compilar el proyecto y ejecutar el **.nds** nos mostrara la imagen fig. 17.



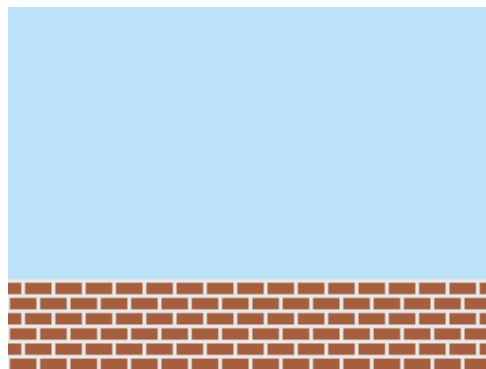
*Fig. 17: Figura 1*

Comprobaremos que los ladrillos se encuentran en la parte superior, así que nos enseñara como funciona los registro **BGxHOFS** y **BgxVOFS** para mover el **BG** por la pantalla. Para ello, tendremos 2 opciones o mover la pantalla 144 píxeles hacia arriba o 112 hacia abajo, esta última la que utilizaremos mediante este código de la tabla 54.

```
REG_BG0VOFS = 112;
```

Tabla 54: Registro BGxVOFS

Compilamos y ejecutamos el **.nds** nos mostrará esta imagen fig. 18 donde los ladrillos ya estan colocados donde necesitamos.



*Fig. 18: Figura 2*

Para el gradiente tendremos que usar **BG1** y comenzamos borrando a 0 el *tilemap*, código de la tabla 55.

```
int n;  
for( n = 0; n < 1024; n++ ) bg1map[n] = 0;
```

Tabla 55: Borrado Tilemap gradiente

Seguidamente, llenaremos la parte superior con 32x8 *tiles* del gradiente con este código de la tabla 56.

```

int x, y;
for( x = 0; x < 32; x++ )
{
    for( y = 0; y < 8; y++ )
    {
        int tile = tile_gradient + y;
        bg1map[ x + y * 32 ] = tile | (pal_gradient << 12);
    }
}

```

Tabla 56: Rellenado TileMap gradiente

Y añadimos a **videoSetMode()** el código de la tabla 57 para mostrar **BG1**.

```

videoSetMode( MODE_0_2D | DISPLAY_BG0_ACTIVE | DISPLAY_BG1_ACTIVE );

```

Tabla 57: *VideoSetMode con BG1 activo*

Compilamos y ejecutamos el **.nds** que nos mostrará esta imagen fig. 19.

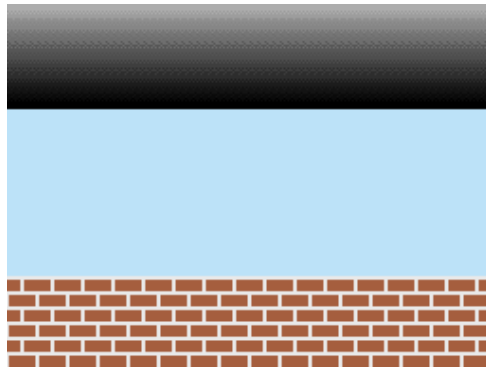


Fig.19: Figura 3

Esta parte se Corresponde a la carpeta **BouncingBallFase 2**.

### Segundo añadido: mover, eliminar y cambiar la prioridad de los BGs

Siguiendo la línea del anterior añadido, en esta parte del proyecto vamos a añadir una nueva funcionalidad. En este caso nos permitirá poder mover arriba y abajo tanto el **BG** de ladrillos como el **BG** del gradiente, así como ocultarlos, volverlos a mostrar o cambiar su prioridad para que se superpongan uno o otro. Utilizaremos tanto las flechas del **PAD** de arriba y abajo como los botones **A**, **B**, **L** y **R**, también pondremos una línea de texto que nos indique que **BG** esta como prioritario.

Lo primero que haremos será introducir este código de la tabla 58 en las referencias para permitir la utilización de *iprintf()*.

```

#include <stdio.h>

```

Tabla 58: *Librería stdio.h*

Ahora nos toca escribir las nuevas variables que necesitaremos para controlar las posiciones de los dos **BGs**, como quien tiene la prioridad, mostradas en la tabla 59.

```
int punteroL = 112;
int punteroG = 0;
int fondo = 0;
```

Tabla 59: Variables añadido 2

En la función *setupGraphics()*, necesitaremos introducir la leyenda que mostraremos en este añadido, mediante el siguiente código de la tabla 60 al inicio de esta función.

```
consoleDemoInit();

iprintf("\x1b[4;8HPrueba Sistema 2");
iprintf("\x1b[8;1HBoton L: Seleccionar Fondo");
iprintf("\x1b[10;1HBoton R: Asignar Max. Prioridad");
iprintf("\x1b[12;1HBoton A: Quita Fondo Elegido");
iprintf("\x1b[14;1HBoton B: Pone Fondo Elegido");
iprintf("\x1b[16;1HUP: Sube Fondo Elegido");
iprintf("\x1b[18;1HDOWN: Baja Fondo Elegido");
```

Tabla 60: Leyenda añadido 2

Por último, rellenaremos la función *updateGraphics()* con todo el código de la tabla 61 que nos permitirá interactuar con los **BG**s de esta fase del proyecto, además de contener los *iprint* que nos dicen que **BG** es prioritaria.

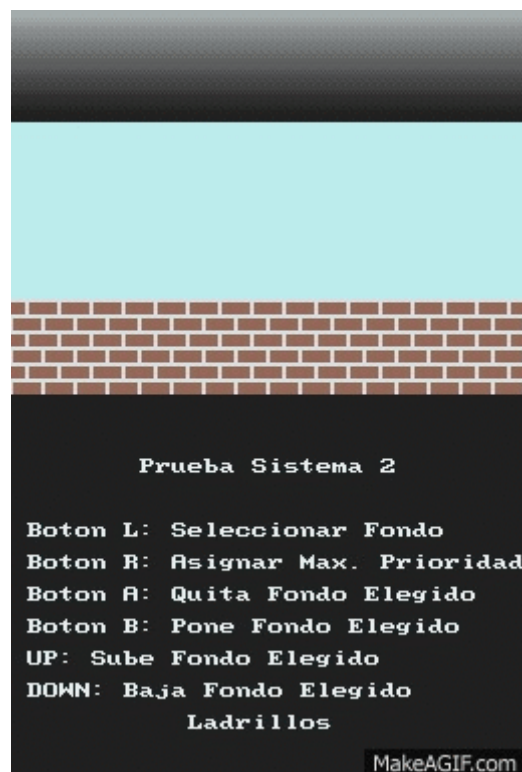


Fig. 20: Añadido 2.0.3

En esta parte, vemos nuevas llamadas como, *bgSetPriority()* que nos permite cambiar la prioridad del **BG** que pongamos (El primer número es el BG, y el segundo el nivel de prioridad, que va del 0-3). También tenemos las llamadas *bgHide()* y *bgShow()* que ocultan o muestran respectivamente, la **BG** que le pongamos.

Una vez terminado, compilamos y ejecutamos el **.nds**, permitiendonos cosas como las de la fig. 20.



```

scanKeys();
int keysd = keysDown();
int keysh = keysHeld();

if(keysd & KEY_L){
    if (fondo == 0 ) fondo = 1;
    else fondo = 0;
}
if(keysd & KEY_R){
    if (fondo == 0 ){
        bgSetPriority(0,0);
        bgSetPriority(1,1);
    }
    else{
        bgSetPriority(0,1);
        bgSetPriority(1,0);
    }
}
if(keysd & KEY_A){
    if(fondo == 0) bgHide(0);
    else bgHide(1);
}
if(keysd & KEY_B){
    if(fondo == 0) bgShow(0);
    else bgShow(1);
}
if(keysh & KEY_UP)
{
    if(fondo == 0){
        REG_BG0VOFS = punteroL;
        punteroL++;
    }
    else{
        REG_BG1VOFS = punteroG;
        punteroG++;
    }
}
if(keysh & KEY_DOWN){
    if(fondo == 0){
        REG_BG0VOFS = punteroL;
        punteroL--;
    }
    else{
        REG_BG1VOFS = punteroG;
        punteroG--;
    }
}
if(fondo == 0) iprintf("\x1b[20;1HLadrillos");
else iprintf("\x1b[20;1HGradiente");

```

*Tabla 61: Código updateGraphics()*

Esta parte se Corresponde a la carpeta **BouncingBallFase 2.0.3**.

### Tercera Parte del resumen

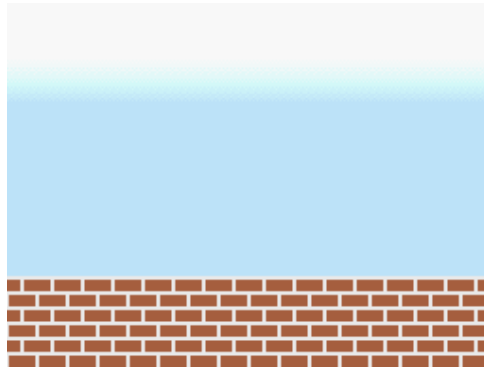
En esta parte del proyecto, nos explican el registro **BLDCNT** que controla los efectos espaciales que se aplican al vídeo. Existen 4 Modos de vídeo Modo 0: Desactivado, Modo 1: Alfa Blending y Modos 2 y 3: Fundido de imagen. Todos están detallados en el trabajo original [6]. Nosotros usaremos el modo 1, que mezclara nuestro gradiente con el *backdrop*.

Ahora trataremos de combinar nuestro **BG1** (Gradiente) con el *backdrop*, para ello comenzaremos con este código de la tabla 62.

```
REG_BLDCNT = BLEND_ALPHA | BLEND_SRC_BG1 | BLEND_DST_BACKDROP;  
REG_BLDALPHA = (16) + (16<<8);
```

*Tabla 62: Registros BLDCNT*

Esto, sumará ambos coeficientes al 100%, dando como resultado la imagen de la fig. 21.



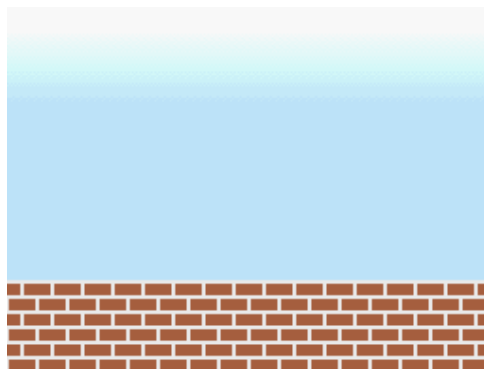
*Fig. 21: Primera BLDALPHA*

Como se puede comprobar, la parte superior ha quedado muy brillante por lo que, reduciremos la suma de los dos coeficientes al 50%. Para ello modificaremos el código de **REG\_BLDALPHA** por este otro de la tabla 63.

```
REG_BLDALPHA = (8) + (16<<8);
```

*Tabla 63: Modificación BLDALPHA*

Y este es el resultado, en la fig 22.



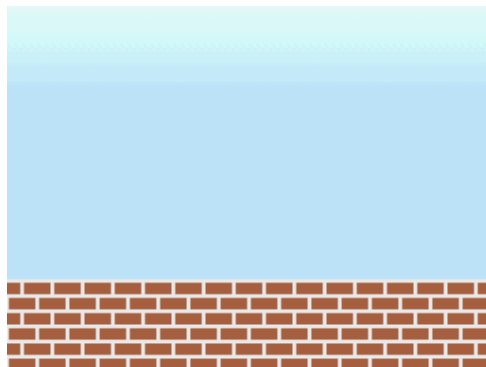
*Fig. 22: Segunda BLDALPHA*

Como aún se ve demasiado brillante en la parte superior, ajustaremos más aún el coeficiente, así que otra vez modificaremos el código de **REG\_BLDALPHA** por este otro de la tabla 64.

```
REG_BLDALPHA = (4) + (16<<8);
```

*Tabla 64: Última modificación BLDALPHA*

Por lo que definitivamente conseguimos el ajuste del coeficiente adecuado, que vemos en la siguiente imagen de la fig. 23.



*Fig. 23: Tercera BLDALPHA*

Esta parte se Corresponde a la carpeta **BouncingBallFase 3**.

## 4.3 Sprites

### Resumen

Para empezar, nos explica un poco la **VRAM** que asignamos anteriormente a *sprites* y que permite x azulejos. Y en primer lugar, podremos una definiciones mostradas en la tabla 65 para cargar los gráficos de la bola

```
#define tiles_ball    0 // ball tiles (16x16 tile 0->3)
#define tile2objram(t) (SPRITE_GFX + (t) * 16)
```

*Tabla 65: Definiciones tiles*

En segundo lugar, copiaremos los *tiles* en la **VRAM**, con el siguiente código de la tabla 66.

```
dmaCopyHalfWords( 3, gfx_ballTiles, tile2objram(tiles_ball), gfx_ballTilesLen );
```

*Tabla 66: dmaCopyHalfwords de la bola*

En el apartado de la paleta, se nos explica que debemos copiar la paleta del *sprite* en la **RAM** y nos muestra unas imágenes de como se debe ver la **RAM** y nos da el código de la tabla 67 para calcular la dirección de la paleta y el índice que utilizará.

```
#define pal2objram(p) (SPRITE_PALETTE + (p) * 16)
#define pal_ball    0 // ball palette (entry 0->15)
```

*Tabla 67: Definiciones paleta*

Luego, copiaremos la paleta en la memoria, con el código de la tabla 68.

```
dmaCopyHalfWords( 3, gfx_ballPal, pal2objram(pal_ball), gfx_ballPalLen );
```

Tabla 68: *dmaCopyHalfWords de la paleta*

En la parte **como funcionan los sprites** nos explicará como funciona el área de la memoria llamada **OAM** (*Object Attribute Memory*) y como están configurados cada uno de los 3 atributos que los componen y sus características internas.

Al llegar al apartado, **viendo algunos sprites** nos tocará añadir una definición a la estructura del **OAM** y sus atributos, con el código de la tabla 69.

```
typedef struct t_spriteEntry
{
    u16 attr0;
    u16 attr1;
    u16 attr2;
    u16 affine_data;
} spriteEntry;

#define sprites ((spriteEntry*)OAM)
```

Tabla 69: *Atributos y definición para el sprite*

Ahora, en *setupGraphics()*, modificaremos la llamada a la función *videoSetMode()* con el código de la tabla .

```
videoSetMode( MODE_0_2D | DISPLAY_BG0_ACTIVE | DISPLAY_BG1_ACTIVE |  
DISPLAY_SPR_ACTIVE | DISPLAY_SPR_1D_LAYOUT );
```

Compilamos y ejecutamos el **.nds**, veremos una imagen como la fig. 24.

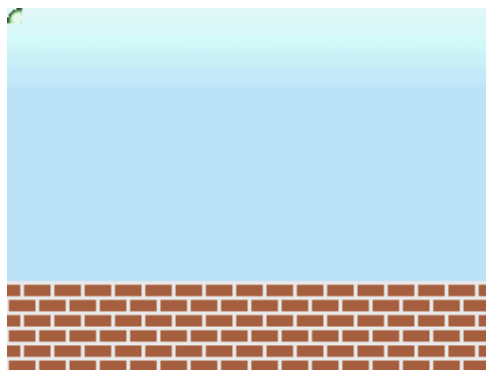


Fig. 24: *Sprite 1*

Se puede observar en la imagen fig. 24 Que solo se muestra un *tile* del sprite, aunque realmente son los 128 apilados, para solucionar esto configuraremos todas las entradas de sprites antes del ajuste de vídeo, con el siguiente código de la tabla 70.

```
int n;  
for( n = 0; n < 128; n++ )  
    sprites[n].attr0 = ATTR0_DISABLED;
```

Tabla 70: *Configuración atributo 0 del sprite*

Con esto el *sprite* desaparecerá de la pantalla. Compilamos y ejecutamos el **.nds**, nos mostrará una imagen como al fig. 23.

Ahora añadiremos el código de la tabla 71, para añadir bolas por toda la pantalla.

```
//Codigo de test para probar el motor de sprites
for(n = 0; n < 50; n++)
{
    //atributo0: fijar la posicion vertical 0-> pantall_alto-sprite_alto
    //otras opciones por defecto que estarían bien (default == zeroed)
    sprites[n].attr0 = rand() % (192-16);

    //atributo1: fijar la posicion horizontal 0-> pantalla_ancho-sprite_ancho
    //establecer modo de tamaño 16x16
    sprites[n].attr1 = (rand() % (256-16)) + ATTR1_SIZE_16;

    //atributo2: seleccionar numero de tile y numero paleta
    sprites[n].attr2 = tiles_ball + (pal_ball << 12);
}
```

Tabla 71: Código test

Por último compilamos y ejecutamos el **.nds**, la imagen resultante sería como la fig. 25.

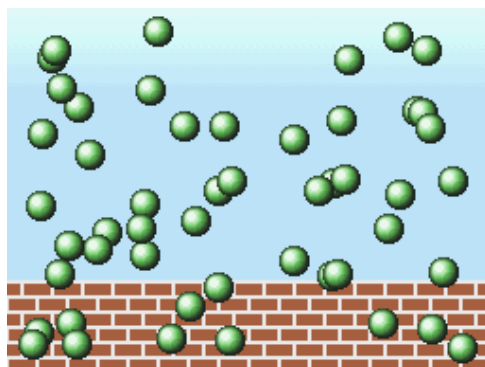


Fig. 25: Imagen de prueba de sprites

Esta parte se corresponde a la carpeta **BouncingBallFase 4.0**.

## 4.4 Avanzado

### Primera parte del resumen

En la parte de **aritmética de coma entera** (*fixed point arithmetic*), el autor habla de como solucionar el problema de que la consola **Nintendo DS** no tenga una **FPU** (unidad de coma flotante) y solucionar el problema con las operaciones que requieres de una **FPU**, para ello nos explica como usar la aritmética de coma entera para solucionar estos problemas.

Por fin, llegamos al apartado del **balon hinchable** (*the bouncy ball*). Para ello primero tendremos que crear un nuevo archivo y lo llamaremos *ball.c* que lo añadiremos en la carpeta **source**. También crearemos un nuevo archivo llamado *ball.h* que lo agregaremos en el directorio *include*.

En el apartado **el objeto bola**, necesitaremos definir las variables de la bola, así que abriremos *ball.h* y añadiremos el código de la tabla 72.

```
//ball.h

#ifndef BALL_H
#define BALL_H

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 heigth;    //ancho del balon
} ball;

void ballUpdate( ball* b );
void ballRender( ball* b, int camera_x, int camera_y );

#endif
```

Tabla 72: Código *ball.h*

Luego nos explica para que sirve cada variable que manejará nuestra bola y las dos funciones que necesitaremos.

En el apartado de **programación de la bola**, tendremos que abrir *ball.c* para comenzar incluyendo las definiciones del mismo, mediante el código de la tabla 73.

```
#include <nds.h>
#include "ball.h"
```

Tabla 73: Librerías *ball.c*

Después, necesitaremos las dos funciones que pusimos en *ball.h*, tanto *ballUpdate()*, como *ballRender()*, para manejar las físicas de la bola como su renderizaje. Para ello escribiremos el siguiente código de la tabla 74.

```
void ballUpdate( ball* b )
{
}

void ballRender( ball* b, int camera_x, int camera_y )
{
}

}
```

Tabla 74: Funciones *ballUpdate()* y *ballRender()*

Una vez tenemos nuestros esqueletos de las funciones, empezaremos por *ballRender()*. En el cual tendremos que introducir este código mostrado en la tabla 75 y además añadir la definición del radio

de la bola con el código de la tabla 76.

```
u16* sprite = OAM + b->sprite_index * 4;

int x, y;
x = ((b->x - c_radius) >> 8) - camera_x;
y = ((b->y - c_radius) >> 8) - camera_y;

if( x <= -16 || y <= -16 || x >= 256 || y >= 192 )
{
    //sprites fuera de rango
    //Deshabilitar el sprite
    sprite[0] = ATTR0_DISABLED;
    return;
}

sprite[0] = y & 255;
sprite[1] = (x & 511) | ATTR1_SIZE_16;
sprite[2] = 0;
```

Tabla 75: Código dentro de *ballRender()*

```
#define c_radius (8<<8)
```

Tabla 76: Radio de la Bola

Ahora nos toca volver al *main.c* para modificar ciertas partes del código, empezando por cambiar el *main()* por este código de la tabla 78.

Ahora, necesitaremos crear la función de *setupInterrupts()* para configurar el manejador de interrupciones mediante el código de la tabla 77.

```
void setupInterrupts( void )
{
    //Habilitar interrupcion vblank (necesario para swiWaitForVBlank)
    irqEnable( IRQ_VBLANK );
}
```

Tabla 77: Función *setupInterrupts*

```

int main( void )
{
    //Cosas de configuracion
    setupInterrupts();
    setupGraphics();
    resetBall();

    //Bucle Principal
    while(1)
    {
        //Periodo de Renderizado
        //Actualizacion objetos del juego (Moverlos alrededor, calcular velocidades, etc)
        updateLogic();

        //Espera para periodo vblank
        swiWaitForVBlank();

        //Periodo vblank: (seguro modificar graficos)
        //mover graficos alrededor
        updateGraphics();
    }
    return 0;
}

```

Tabla 78: Nuevo main

La función *setupGraphics()* la mantendremos igual que en el apartado anterior, salvo que tendremos que eliminar el código que nos genera bolas por todos lados.

Mientras *resetBall()* es una nueva función para inicializar nuestro objeto bola. Por tanto, primero añadiremos al principio del código entre las librerías/definiciones y la estructura del *sprite*, el siguiente código de la tabla 79.

```

#include "ball.h"

ball g_ball;

```

Tabla 79: Librería y objeto bola

Ahora, ya podemos rellenar la función *resetBall()* con el código de la tabla 81.

Por último, rellenaremos la función *updateGraphics()* con el código de la tabla 80.

```

//Informacion Actualizacion grafica (llamar durante vblank)
void updateGraphics( void )
{
    //Actualizar sprite del balon, camara = 0, 0
    ballRender( &g_ball, 0, 0);
}

```

Tabla 80: Código updateGraphics()



```

//Resetear atributos del balon
void resetBall( void )
{
    //usar indice sprite 0 (0->127)
    g_ball.sprite_index = 0;

    //usar matriz de afinamiento 0 (0->131)
    g_ball.sprite_affine_index = 0;

    //X = 128.0
    g_ball.x = 128 << 8;

    //Y = 64.0
    g_ball.y = 64 << 8;

    //iniciar velocidad X un bit a la derecha
    g_ball.xvel = 100 << 4;

    //resetear velocidad Y
    g_ball.yvel = 0;
}

```

Tabla 81: Código resetBall()

Finalmente, compilamos y ejecutamos el **.nds**, nos mostrará una imagen como la fig. 26.

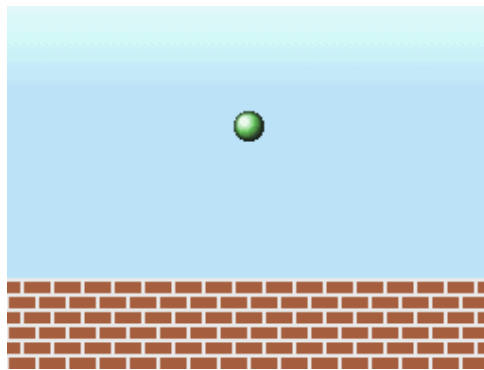


Fig. 26: Resultado

Esta parte se Corresponde a la carpeta **BouncingBallFase 5.0**.

### Primer añadido: Mover, escalar y rotar el sprite

Para este añadido, la nueva funcionalidad nos permitirá poder mover, escalar o rotar el sprite a nuestro antojo por la pantalla, eso si, añadiendo límites en la pantalla a modo de límite de escenario. Para ello usaremos el **PAD** para mover el sprite, los botones **A** y **B** para escalar y los botones **X** e **Y** para rotarlo.

Así que lo primero que haremos será, añadir los nuevos atributos de *ball.h*, que son el control de ángulos y de escala, para ello necesitaremos añadir en la parte de *typedef struct t\_ball* debajo de **int height** el código de la tabla 82.

```

int alpha; //Ángulo de la bola
bool scale; //Mira si esta con rotacion o escalado: true:escalado, false: rotacion

```

Tabla 82: Nuevos atributos ball.h

Ahora pasaremos al interior de *ball.c*, donde primero escribiremos las nuevas definiciones que marcarán los límites de la pantalla y diámetro del *sprite*, mediante el código de la tabla 83.

```
#define c_both_level    ((192 - 8) << 8) //límite inferior
#define c_top_level    ( (0 - 8) << 8) //límite superior
#define c_izq_level    ((0 - 8) << 8) //límite superior
#define c_der_level    ((256 - 8) << 8) //límite superior

#define c_diam          32 //el diametro del balon (entero)
```

Tabla 83: Nuevas definiciones *ball.c*

Después, nos centraremos en rellenar la función *ballUpdate()* para controlar el renderizado de la bola dentro de los límites de la pantalla y también controlar tanto su escala máxima como mínima, mediante el código de la tabla 84.

```
//Control del limite de la pantalla
if(b->y + c_radius >= c_both_level ) b->y = c_both_level - c_radius;
if(b->y - c_radius <= c_top_level ) b->y = c_top_level + c_radius;
if(b->x - c_radius <= c_izq_level ) b->x = c_izq_level + c_radius;
if(b->x + c_radius >= c_der_level ) b->x = c_der_level - c_radius;

//Control del tamaño de la bola
if(b->height <= (16 << 8)) b->height = (16 << 8);
if(b->height >= (64 << 8)) b->height = (64 << 8);
```

Tabla 84: Contenido *ballUpdate()*

Para finalizar con *ball.c*, añadiremos unas líneas en *ballrender()*, para poder mostrar tanto el escalado como la rotación, manejando la matriz *affine*, para ello escribiremos los trozos de código que están subrayados, de la tabla 86.

Ahora, le toca el turno a *main.c*, empezaremos por las definiciones, que en este caso se tratará de nuevo de la librería *stdio.h* (tabla 58), Luego pasamos a la introducción de la leyenda que usaremos en este apartado, esta irá al final de *setupGraphics()*, mediante el código de la tabla 85.

```
//Leyenda
consoleDemoInit();

//Texto a representar por defecto se procesan en la pantalla inferior
iprintf("\x1b[8;8HPueba sistema 4");
iprintf("\x1b[10;3HBoton A: Aumentar la Bola");
iprintf("\x1b[12;3HBoton B: Disminuir la Bola");
iprintf("\x1b[14;3HBoton X: Rotar a derecha");
iprintf("\x1b[16;3HBoton Y: Rotar a izquierda");
iprintf("\x1b[18;3HBoton START: Reiniciar Bola");
iprintf("\x1b[20;5HPad: Control de la Bola");
```

Tabla 85: Leyenda prueba 4

```

u16* sprite = OAM + b->sprite_index * 4;

u16* affine = OAM + b->sprite_affine_index * 16 + 3;

int x, y;
x = ((b->x - c_diam) >> 8) - camera_x;
y = ((b->y - c_diam) >> 8) - camera_y;

int pa, pb, pc, pd;

if(b->scale)
{
    pa = (b->height * (65536/c_diam)) >> 16;
    pd = (b->height * (65536/c_diam)) >> 16;
    pb = 0;
    pc = 0;
}
else
{
    pa = cosLerp(b->alpha) >> 4;
    pd = cosLerp(b->alpha) >> 4;
    pb = sinLerp(b->alpha) >> 4;
    pc = -sinLerp(b->alpha) >> 4;
}

sprite[0] = (y & 255) | ATTR0_ROTSCALE_DOUBLE;
sprite[1] = (x & 511) | ATTR1_SIZE_16 | ATTR1_ROTDATA( b->sprite_affine_index );
sprite[2] = 0;

affine[0] = pa;
affine[4] = pb;
affine[8] = pc;
affine[12] = pd;

```

Tabla 86: Contenido modificado de *ballrender()*

En la función *resetball()*, añadiremos al final de esta, el siguiente código de la tabla 87, para dejar unos valores por defecto en los nuevos atributos.

```

//resetear la altura de la bola a c_diam 32
g_ball.height = 32 << 8;

//resetear angulo de la bola
g_ball.alpha = 0 << 8;

//resetear rotacion o escalado: por defecto escalado
g_ball.scale = true;

```

Tabla 87: Añadido a *resetball()*

Luego en *updateGraphics()*, escribiremos el código de la tabla 88, que nos permitirá modificar la línea de la matriz *affine* (Esto está un poco cogido con pinzas, ya que he conseguido también que funcione desde *ball.c* y no desde *main.c* ), dependiendo de si estamos modificando la escala o la rotación.

```

if(g_ball.scale)
{
    g_ball.sprite_affine_index = 0;
}
else
{
    g_ball.sprite_affine_index = 1;
}

```

Tabla 88: Añadido *updateGraphics()*

Después, escribiremos la parte de lectura de las entradas, para ello, crearemos una nueva función llamada *processInput()*, donde escribiremos el siguiente código de la tabla 89.

```

scanKeys();

int keysh = keysHeld();
int keysp = keysDown();
//proceso uso de entradas
if( keysh & KEY_UP)           //Chequea si UP esta presionado
{
    g_ball.y -= 1 << 8;
}
if( keysh & KEY_DOWN )       //chequea si DOWN esta presionado
{
    g_ball.y += 1 << 8;
}
if( keysh & KEY_LEFT )       //chequea si LEFT esta presionado
{
    g_ball.x -= 1 << 8;
}
if( keysh & KEY_RIGHT )      //chequea si RIGHT esta presionado
{
    g_ball.x += 1 << 8;
}
if( keysh & KEY_A )          //chequea si A esta presionado
{
    g_ball.height -= 1 << 8;
    if(!g_ball.scale)g_ball.scale = true;
}
if( keysh & KEY_B )          //chequea si B esta presionado
{
    g_ball.height += 1 << 8;
    if(!g_ball.scale)g_ball.scale = true;
}
if(keysp & KEY_START)
{
    resetBall();
}
if(keysp & KEY_X)
{
    g_ball.alpha += 9 << 8;
    if(g_ball.scale)g_ball.scale = false;
}
if(keysp & KEY_Y)
{
    g_ball.alpha -= 9 << 8;
    if(g_ball.scale)g_ball.scale = false;
}

```

Tabla 89: Código de *processInput()*

Una vez terminado de escribir el código de *processInput()*, necesitaremos llamarlo en cada iteración, para ello tanto *processInput()* como *ballUpdate()* los añadiremos en la función *updateLogic()* con las líneas de la tabla 90.

```
processInput();  
ballUpdate( &g_ball );
```

Tabla 90: Código *update\_Logic()*

Una vez terminado, compilamos y ejecutamos el **.nds**, el resultado nos permitirá hacer cosas como los de la figura fig. 27.

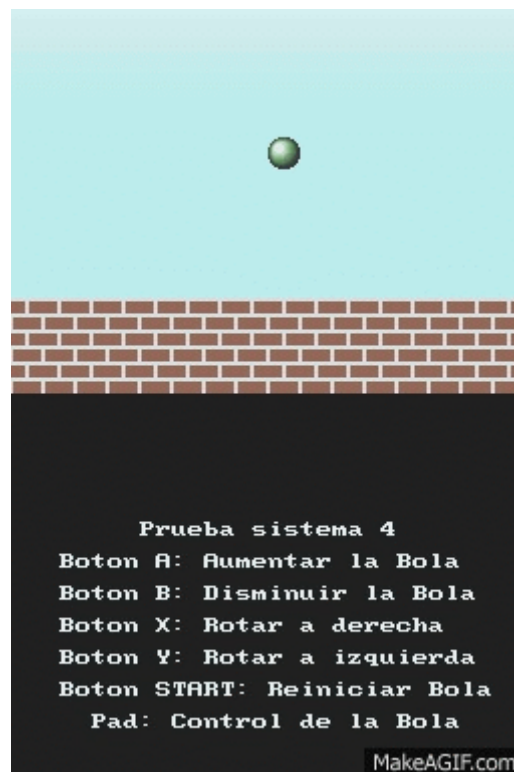


Fig. 27: Añadido 5.0.2

Esta parte se Corresponde a la carpeta **BouncingBallFase 5.0.2**.

## Segunda parte del Resumen

En el apartado de **programación de la bola** (*Programming the ball*), tendremos que añadir la llamada a *ballUpdate()*, dentro de *processLogic()* ya que el comportamiento de la bola debe actualizarse en cada fotograma, para ello introduciremos el código de la tabla 91 dentro de *processLogic()*.

```
void processLogic( void )
{
    ballUpdate( &g_ball );
}
```

*Tabla 91: Código processLogic()*

Ahora nos tocará ir a *ball.c* para escribir el contenido de *ballUpdate()*, con el código de la tabla 92.

```
//Actualizar objeto balon (una llamada por frame)
void ballUpdate( ball* b )
{
    //Añadir X velocidad a la posicion X
    b->x += (b->xvel>>4);
}
```

*Tabla 92: Código ballUpdate()*

Si observas el código donde sale `>> 4` se debe a que **xvel** 20,12 es formato de punto fijo, mientras **x** es de 24,8, así que **xvel** debe estar alineado con **x** antes de la adición.

Compilamos y ejecutamos el **.nds**, nos mostrará un movimiento de la bola hacia la derecha.

Esta parte se Corresponde a la carpeta **BouncingBallFase 5.1**.

Para continuar, definiremos algunos valores constantes de la física de la bola. Mediante el código de la tabla 94.

Vamos a añadir la “fricción del aire” a la velocidad **x** dentro de la función *ballUpdate()*, mediante el código de la tabla 93.

```
//aplicar friccion del aire a la velocidad X
b->xvel = (b->xvel * (256-c_air_friction)) >> 8;

//fijar velocidad X en los limites
b->xvel = clampint( b->xvel, -max_xvel, max_xvel );
```

*Tabla 93: Fricción con el aire y límites*

```

#define c_gravity          80                      //constante de gravedad
(añadir velocidad vertical) (*.8 fixed)

#define c_air_friction      1                      //friccion con el aire...
Multiplicador velocidad X por (256-f)/256
#define c_ground_friction 30                      //friccion cuando el balon choca con
el suelo, multiplicador X por (256-f)/256
#define c_platform_level ((192-48) << 8) //El nivel de la plataforma de ladrillos en *.8 fixed point
#define c_bounce_damper    20                      //la cantidad de velocidad Y
que es absorbida cuando golpeas el suelo

#define c_radius            (8<<8)                //El radio del balon en *.8 fixed point
#define c_diam              16                    //el diametro del
balon (entero)

#define min_height          (1200)                //la minima anchura del
balon (cuando es aplastado) (*.8)

#define min_yvel            (1200)                //la minima velocidad Y (*.8)
#define max_xvel            (1200<<4)            //la maxima velocidad X
(*.12)

//fijar en rango un numero entero
static inline int clampint(int value, int low, int high)
{
    if( value < low ) value = low;
    if( value > high ) value = high;
    return value;
}

```

*Tabla 94: Físicas de la bola*

Compilamos y ejecutamos el **.nds**, deberemos notar como la bola se frena hasta pararse,

Esta parte se Corresponde a la carpeta **BouncingBallFase 5.2**.

Ahora añadiremos la constante de gravedad y la velocidad Y con el código de la tabla 95.

```

//añadir gravedad a la velocidad Y
b->yvel += c_gravity;

//añadir velocidad Y a la posicion Y
b->y += (b->yvel);

```

*Tabla 95: Gravedad y velocidad Y*

Compila y ejecuta el **.nds**, deberías ver caer la bola hasta fuera de la pantalla.

Esta parte se Corresponde a la carpeta **BouncingBallFase 5.3**.

Para solucionar esto, haremos que la bola rebote en la plataforma mediante el código de la tabla 96 que seguiremos poniendo en *ballUpdate()*.

```

if(b->y + c_radius >= c_platform_level )
{
    //aplicar friccion de suelo a la velocidad X
    b->xvel = (b->xvel * (256-c_ground_friction)) >> 8;

    //montaje plataforma Y
    b->y = c_platform_level - c_radius;

    //Negar velocidad Y, mientras aplicar amortiguador de rebote
    b->yvel = -(b->yvel * (256-c_bounce_damper)) >> 8;

    //fijar la minima velocidad Y (minimo despues de rebotar, para que el balon no se
    desestabilice
    if( b->yvel > -min_yvel )
    {
        b->yvel = -min_yvel;
    }
}

```

Tabla 96: Código para rebotar en los ladrillos

Compila y ejecuta el **.nds**, ahora sí, deberías ver como rebota la bola continuamente sobre la plataforma de ladrillos.

Esta parte se Corresponde a la carpeta **BouncingBallFase 5.4**.

Una vez conseguido que la bola rebote en la plataforma, pasaremos al apartado de **entrada** (*Input*), para ello necesitaremos leer el teclado (Botones y PAD de la consola). Por tanto, necesitaremos crear una nueva función *processInput()*, donde escribiremos el siguiente código de la tabla 98.

Los valores *x\_tweak* y *y\_tweak* tendrán la cantidad de velocidad que se debe añadir a la velocidad cuandos e pulsa una tecla, por tanto, deberemos definir las en algún lado. Así que, nos vamos a la parte superior del código y escribimos justo debajo del último *#define*, el siguiente código de la tabla 97.

```

#define x_tweak          (2<<8) //para uso de entradas
#define y_tweak          25      //para uso de entradas

```

Tabla 97: Definiciones

Por último, añadimos *processInput()* a la función *processLogic()*.



```

void processInput( void )
{
    scanKeys();

    int keysh = keysHeld();
    //proceso uso de entradas
    if( keysh & KEY_UP)                //Chequea si UP esta presionado
    {
        //ajustar la velocidad Y negativa del balon
        g_ball.yvel -= y_tweak;
    }
    if( keysh & KEY_DOWN )            //chequea si DOWN esta presionado
    {
        //ajustar la velocidad Y positiva del balon
        g_ball.yvel += y_tweak;
    }
    if( keysh & KEY_LEFT )            //chequea si LEFT esta presionado
    {
        //ajustar velocidad X negativa del balon
        g_ball.xvel -= x_tweak;
    }
    if( keysh & KEY_RIGHT )           //chequea si RIGHT esta presionado
    {
        //ajustar velocidad X positiva del balon
        g_ball.xvel += x_tweak;
    }
}

```

*Tabla 98: Lectura de botones*

Compilamos y ejecutamos el **.nds**, ahora ya podemos manejar la bola y moverla de un lado para otro.

Esta parte se Corresponde a la carpeta **BouncingBallFase 5.5**.

Ahora, ya podemos mover la bola a nuestro antojo, pero cuando esta atraviesa los bordes laterales la pantalla no la sigue, por tanto tendremos que crear una cámara que siga a la bola en su movimiento.

Para ello, en el apartado **la cámara (The Camera)**, vamos a crearla. Para ello necesitaremos crear las variables para manejar su posición, mediante el código de la tabla 99 que escribiremos debajo de la línea `ball g_ball;`.

```

int g_camera_x;
int g_camera_y;

```

*Tabla 99: Variables de la cámara*

Ahora, necesitaremos una nueva función para actualizar la cámara, para ello escribiremos el código de la tabla 100.

```

void updateCamera( void )
{
    // cx = posicion X camara deseada
    int cx = ((g_ball.x)) - (128 << 8);

    // Dx = diferencia entre la posición deseada y actual
    int dx;
    dx = cx - g_camera_x;

    // 10 es el umbral mínimo
    if( dx > 10 || dx < -10 )
    {
        dx = (dx * 50) >> 10; //escalar el valor de la cantidad dx
    }

    // Agregar el valor a la posición X de la cámara
    g_camera_x += dx;

    // Cámara Y es siempre 0
    g_camera_y = 0;
}

```

Tabla 100: Código de *updateCamera()*

Esta lectura y posicionamiento de la cámara debemos controlarla en cada fotograma, por lo que la incluiremos dentro de *processLogic()*, justo después de *ballUpdate()*.

A continuación, modificaremos *updateGraphics()* para que use la posición de la cámara. Así que modificaremos el código existente por el de la tabla 101.

```

//Actualizar sprite del balon, camara = 0, 0
ballRender( &g_ball, 0, 0);

REG_BG0HOFS = g_camera_x >> 8;

```

Tabla 101: Modificación *updateGraphics()*

Compilamos y ejecutamos el **.nds**, ahora la cámara seguirá a la bola cuando esta se mueva.

Esta parte se Corresponde a la carpeta **BouncingBallFase 5.6**.

Ahora, en el apartado **transformando sprites** (*Transforming sprites*), para jugar un poco con la forma de la bola, para ello nos explica como la matriz affine, nos permite rotar, escalar o cortar el sprite y tras explicar como funciona cada uno de ellos, nos propone hacer que la bola se aplaste cada vez que choque con la plataforma de ladrillos. Para ello, comenzaremos cambiando el código de la tabla 96 dentro de *ballUpdate()*, por el de la tabla 102.

```

if(b->y + c_radius >= c_platform_level )
{
    //aplicar friccion de suelo a la velocidad X
    //(si, esto puede hacerse multiples veces)
    b->xvel = (b->xvel * (256-c_ground_friction)) >> 8;

    //comprobar si el balon a sido aplastado a la altura mínima
    if(b->y > c_platform_level - min_height)
    {
        //montaje plataforma Y
        b->y = c_platform_level - min_height;

        //Negar velocidad Y, mientras aplicar amortiguador de rebote
        b->yvel = -(b->yvel * (256-c_bounce_damper)) >> 8;

        //fijar la minima velocidad Y (minimo despues de rebotar, para que el balon no se
desestabilice
        if( b->yvel > -min_yvel )
        {
            b->yvel = -min_yvel;
        }
    }

    //Calcular la altura
    b->height = (c_platform_level - b->y) * 2;
}
else
{
    b->height = c_diam << 8;
}

```

Tabla 102: Nuevo código *ballUpdate()*

Luego explica, que debido a que la caja de renderizado estándar que tiene el *sprite* solo permite visualizar el 50% al aplastarse la bola, por lo tanto necesitamos aumentar al doble el tamaño de dicha caja en el atributo 0, para ello necesitaremos cambiar el código de *ballRender()* de la tabla 75, por el código de la tabla 103.

Termina explicando, como funciona tanto la rotación como escalado de *sprites*, aunque solo se centra en el escalado. Y muestra como con la matriz *affine*, se puede modificar la forma del *sprite*.

```

u16* sprite = OAM + b->sprite_index * 4;
u16* affine = OAM + b->sprite_affine_index * 16 + 3;

int x, y;
x = ((b->x - c_radius * 2) >> 8) - camera_x;
y = ((b->y - c_radius * 2) >> 8) - camera_y;

int pa = (b->height * (65536/c_diam)) >> 16;
int pd = 65536 / pa;

if( x <= -16 || y <= -16 || x >= 256 || y >= 192 )
{
    //sprites fuera de rango
    //Deshabilitar el sprite
    sprite[0] = ATTR0_DISABLED;
    return;
}

sprite[0] = (y & 255) | ATTR0_ROTSCALE_DOUBLE;
sprite[1] = (x & 511) | ATTR1_SIZE_16 | ATTR1_ROTDATA( b->sprite_affine_index );
sprite[2] = 0;

affine[0] = pa;
affine[4] = 0; //No se usa en el escalado
affine[8] = 0; //No se usa en el escalado
affine[12] = pd;

```

Tabla 103: Nuevo código ballRender()

Finalmente si compilamos y ejecutamos el **.nds**, verás como al llegar al suelo la bola se aplasta.

Esta parte se Corresponde a la carpeta **BouncingBallFase 6.1**.

## Segundo añadido: Score de saltos, tiempo de juego y guardado en data.

Para el último añadido, trataremos unos poco aspectos de un juego, como puede ser el control del tiempo de juego, una tabla de *score* (Como podría ser acumulación de puntos, monedas etc. ) en este caso contando los botes que da la bola contra el suelo, la visualización del score guardado en un archivo **.txt**.

Lo primero que haremos sera añadir las librerías nuevas que vamos a necesitar mediante el código de la tabla 104.

```

#include <filesystem.h> //Para nitrofs
#include <fat.h> //PAra FAT
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <time.h>

```

Tabla 104: Librerías añadido 4

Luego necesitaremos escribir las nuevas variables que vamos a tener que utilizar. Estas están en el código de la tabla 105.

```

uint ticks = 0;    //Variable que controla el tiempo de juego
int minutos = 0;
int tiempo;
int len;

FILE* inf;
u8 c;

```

*Tabla 105: Variables añadido 4*

Ahora pasaremos al *main()*, donde modificaremos las siguientes líneas marcadas mostradas en el código de la tabla 106, añadiendo o cambiando según lo necesario.

```

//Cosas de configuracion
setupInterrupts();
setupData();
setupGraphics();
resetBall();

//Bucle Principal
while(!(keysDown() & KEY_START))
{
    //Calculo del tiempo de juego
    processTime();

    //Periodo de Renderizado
    //Actualizacion objetos del juego (Moverlos alrededor, calcular velocidades, etc)
    processLogic();

    //Espera para periodo vblank
    swiWaitForVBlank();

    //Periodo vblank: (seguro modificar graficos)
    //mover graficos alrededor
    updateGraphics();
}

saveGame();

return 0;

```

*Tabla 106: Añadidos de main()*

En el siguiente paso, crearemos la función *setupData()* mediante el código siguiente de la tabla 107.

```

void setupData(void)
{
    consoleDemoInit();

    FILE* fp;

    if (nitroFSInit(NULL))
    {

        fp = fopen("save.txt", "r");
        if(fp)
        {
            fseek(fp,0,SEEK_END);
            len = ftell(fp);
            fseek(fp,0,SEEK_SET);
            iprintf("\x1b[10;9HTop Scores:");
            char *entireFile = (char*)malloc(len+1);
            entireFile[len] = 0;
            if(fread(entireFile,1,len,fp) != len)
            iprintf("\x1b[20;0HError al leer el archivo!");
            else
            {

                iprintf("\n\n\t\t\t%s", entireFile);

            }
            free(entireFile);

            fclose(fp);

        }

    }
    else iprintf("\x1b[20;0HnitroFSInit fallo: terminando\n");

    //leyenda
    iprintf("\x1b[4;8HPrueba Sistema 5");
    iprintf("\x1b[6;4HStart: Cierra el juego");
}

```

Tabla 107: Código de *setupdata()*

Ahora, calcularemos el contados del tiempo en la función *processTime()*, para ello escribiremos el código de la tabla 108.

```

void processTime(void)
{
    timerStart(0, ClockDivider_1024, 0, NULL);

    ticks += timerElapsed(0);

    tiempo = ticks/TIMER_SPEED;

    if(tiempo < 60)
    {

        iprintf("\x1b[1;20HTime: %02i:%02i\n", minutos, tiempo);

    }
    else
    {
        minutos += 1;
        ticks=0;
    }
}

```

Tabla 108: Función *processTime()*

En esta función 108, vemos tanto el conteo del tiempo con *timerStart()* y luego pasado a segundos el la variable **tiempo**, como la parte donde se muestra en pantalla y de paso se calcula los minutos.

Ahora necesitaremos poner el **score** del juego para ello tendremos que añadir en *processLogic()*, ya que queremos refrescar el tiempo cada vez que actualicemos la bola, para ello añadiremos la siguiente línea de la tabla 109 justo después de *ballUpdate()*.

```

iprintf("\x1b[1;1HScore: %d", g_ball.score);

```

Tabla 109: Añadido en *processLogic()*

Luego, vamos a necesitar poder guardar nuestros scores y tiempos, para ello escribiremos la función *saveGame()* con el código de la tabla 110.

```

void saveGame(void)
{
    //Guardado save con fat
    consoleDemoInit();
    FILE *fp;

    if(!fatInitDefault()) iprintf("\x1b[20;0HInit FAT: Error!\n");
    else
    {
        // Ejemplo de acceso en modo texto
        fp = fopen("save.txt", "w");
        if(!fp) printf("\x1b[20;0HEscribint fat:/test.txt: Error!\n");
        else
        {
            fprintf(fp, "%d\t%d:%d\n", g_ball.score, minutos, tiempo);
            fclose(fp);
        }
    }
}

```

Tabla 110: Código de *saveGame()*

Para finalizar con *main.c*, sólo necesitamos añadir dentro de la función *resetBall()*, la inicialización del atributo de la bola *score*, con la línea **g\_ball.score = 0;**.

Dado que es necesario al inicializar el atributo **g\_ball.score**, añadirlo en *ball.h*, así que añadiremos la línea de código **int score;** a los atributos de la estructura **t\_ball**.

Para finalizar solo nos queda escribir en *ball.c* la línea de código que llevará el conteo de cada bote que da la bola, por ello tendremos que añadir esta línea marcada en la sección de código que muestro en la tabla 111.

```
if(b->y + c_radius >= c_platform_level )
{
    //aplicar friccion de suelo a la velocidad X
    //(si, esto puede hacerse multiples veces)
    b->xvel = (b->xvel * (256-c_ground_friction)) >> 8;

    //comprobar si el balon a sido aplastado a la altura mínima
    if(b->y > c_platform_level - min_height)
    {
        //montaje plataforma Y
        b->y = c_platform_level - min_height;

        //Negar velocidad Y, mientras aplicar amortiguador de rebote
        b->yvel = -(b->yvel * (256-c_bounce_damper)) >> 8;

        //fijar la minima velocidad Y (minimo despues de rebotar, para que el balon no se
desestabilice
        if( b->yvel > -min_yvel )
        {
            b->yvel = -min_yvel;
        }

        //aumentar score
        b->score +=1;
    }

    //Calcular la altura
    b->height = (c_platform_level - b->y) * 2;
}
```

Tabla 111: Añadido en *ball.c*

Una vez terminado, compilamos y ejecutamos el **.nds**, nos mostrará lo de la fig. 28.



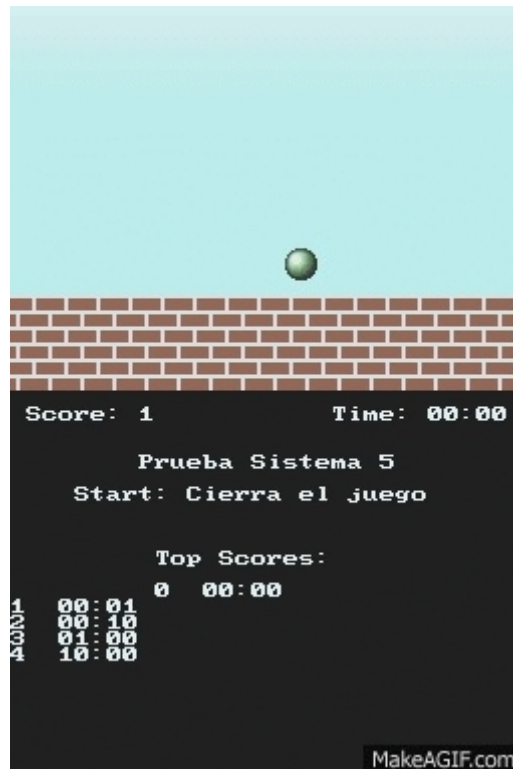


Fig. 28: Añadido 6.2.1

Esta parte se Corresponde a la carpeta **BouncingBallFase 6.2.1**.

### Tercera Parte del resumen

El último apartado, de este trabajo es el **sonido boing** (*Boingy sound*), por desgracia el contenido de la web del autor estaba roto, por tanto es una trabajo que ha tocado hacerlo propio e investigando.

Lo primero que debemos hacer, es crear una carpeta nueva en el proyecto llamada **music**, donde guardaremos los sonidos a utilizar, tanto el de fondo como el efecto de sonido, también tendremos que añadir en el *makefile* la carpeta en el apartado **AUDIO** como se muestra en la tabla 112.

```
TARGET := $(shell basename $(CURDIR))
BUILD := build
SOURCES := source
INCLUDES := include
DATA :=
GRAPHICS := gfx
AUDIO := music
ICON :=
```

Tabla 112: Añadido del audio al makefile

Después, debemos incluir las librerías y referencias para poder usar tanto música de fondo como sonidos FX , a partir de ahora **sfx** en el proyecto. Así que añadiremos en *main* el siguiente código de la tabla 113.

```
#include <maxmod9.h>

//Referencias Sonidos
#include "soundbank_bin.h"
#include "soundbank.h"
```

*Tabla 113: Librería y referencias de sonido*

Ahora, nos vamos hasta el *main()* donde necesitaremos inicializar el *maxmod*, esto nos permitirá empezar a utilizar sonidos de fondo, así como sonidos **sfx**.

Primero, inicializamos *maxmod* con el siguiente código de la tabla 114.

```
//Iniciamos Maxmod
mmInitDefaultMem((mm_addr)soundbank_bin);
```

*Tabla 114: Inicializar Maxmod*

Segundo, crearemos la función *processMusic()* donde cargaremos y comenzaremos la música de fondo y cargaremos el sonido **sfx**, para ello escribiremos el código de la tabla 115.

```
void processMusic( void )
{
    //Cargamos modulo de musica
    mmLoad(MOD_BSMUSIC);
    //Cargamos sonido FX
    mmLoadEffect( SFX_BOING );
    //Comenzamos la musica de fondo
    mmStart(MOD_BSMUSIC, MM_PLAY_LOOP);
}
```

*Tabla 115: Código de processMusic*

La parte donde pone *configuramos el sonido FX* la deje comentada, porque al final utilice la que viene por defecto.

Tendremos que añadir en el *main()*, la llamada a *processMusic()* en la zona de **//Cosas de configuración**.

Seguido, en el *main()* al final de este añadiremos unas líneas para parar la música en caso de que el juego acabe, cerrando el *maxmod*. Para ello añadiremos el código de la tabla 116.

```
mmStop();
mmUnload(MOD_BSMUSIC);
mmUnloadEffect(SFX_BOING);
```

*Tabla 116: Parar y descargar los sonidos*

Por último, necesitamos que cada vez que rebote la bola suene el **sfx**, para ello tenemos que irnos a *ball.c*, donde primero añadiremos también la librería y la referencia de sonidos (aunque `#include "soundbank_bin.h"` no hace falta ponerlo) de la tabla 113. Y la línea de código que ejecutará el sonido **sfx**, esta se tendrá que escribir dentro del *if* donde se calcula si la bola ha tocado la plataforma, con el código de la tabla 117

```
mmEffect(SFX_BOING); //Cargamos el sonido de bote
```

*Tabla 117: Activar sonido SFX*

Compilamos y ejecutamos el **.nds**, ahora podrás escuchar tanto la música que no parará de tocar, hasta el rebote de la bola cada vez que toca la plataforma, si haces que la bola tarde más o menos en tocar la plataforma verás como el sonido de bote suena antes o después.

Esta parte se Corresponde a la carpeta **BouncingBallFase 7.1**.

## 5 Conclusiones y trabajos futuros

Con el desarrollo del presente trabajo se han podido completar los objetivos de manejo de texto, adaptar el tutorial original, completar la parte del audio, añadir el acceso al sistema de ficheros. No se han podido finalizar por 2 de los objetivos, debido en gran parte a la gran ausencia de material explicativo de que hace cada cosa, por lo que la investigación de como hacer funcionar muchos de los objetivos anteriores como de funciones que hemos necesitado para llevarlas a cabo se ha comido más tiempo de lo debido.

Como conclusión puedo decir que ha sido todo un reto alcanzar casi todos los objetivos iniciales y aunque no he podido finalizar el resto no estoy para nada desanimado, ya que sigo pensando en formas de alcanzarlas en un futuro y también de otras muchas ideas que luego listaré que se han quedado en objetivos futuros como : rotar y escalar a la vez utilizando las funciones nuevas **OAM**, generación de obstáculos, introducción de un teclado para poner un nick, uso de versiones multijugador en red o proyecto de niveles o fases .

Para finalizar puedo decir que el proyecto esta lejos de completarse, pero puede dar cabida a futuros proyectos que le añadan esas partes que no pude implementar.

## 6 Bibliografía/webgrafía

- [1] Página web del compilador devkitPro <http://devkitpro.org/>
- [2] Tutorial: How to make a bouncing ball - Mukunda Johnson -  
<http://ekid.nintendev.com/bouncy/index.php> <http://mukunda.com/projects.html>
- [3] Práctica 1 de AEV - M. Agustí - Instalación del entorno de desarrollo devkiPro para la arquitectura de la videoconsola NDS
- [4] Práctica 2 de AEV - M. Agustí - Acceso a información en el hardware de la videoconsola NDS
- [5] Práctica 3 de AEV – M. Agustí - Entrada y salida de datos utilizando el texto
- [6] Práctica 4 de AEV – M. Agustí - Uso del audio en la videoconsola NDS
- [7] Práctica 5 de AEV – M. Agustí - Uso de la imagen en la videoconsola NDS
- [8] Memoria – Scroll y transiciones de escenario en NintendoDS
- [9] Memoria – Tutorial para el uso de las funciones de manejo de backgrounds y sprites
- [10] Memoria – Tutorial – Las funciones de manejo de paletas de color
- [11] Memoria Treball BomberGuy
- [12] [http://survivalengineer.blogspot.com/2011\\_09\\_01\\_archive.html](http://survivalengineer.blogspot.com/2011_09_01_archive.html)
- [13] Página web Librería libnds <http://libnds.devkitpro.org/>
- [14] Página web del creador de no\$gba <http://problemkaputt.de/gba.htm>
- [15] Página web del emulador no\$gba <http://www.nogba.com/>
- [16] Página web del emulador DeSmuME <http://desmume.org/>
- [17] Visualizador de la VRAM de Nintendo DS <http://mtheall.com/vram.html#>  
<http://mtheall.com/banks.html#>
- [18] *Grit GBA Raster Image Transmogrifier* <http://www.coranac.com/projects/grit/>

**7 Anexos**