How to Make a Bouncing Ball Game

Hello! And welcome to the world of DS homebrew programming! In this tutorial, I will explain how to make a dumb game that features a bouncing ball!! Don't be fooled though, this tutorial will teach you how to use *backgrounds*, *sprites*, *alpha blending*, *fixed point arithmetic*, *sprite scaling*, and more!

Your Supplies

Here are all the things you need to make this project.

First, you are going to need a *compiler*. Get devkitARM from devkitPro. If you're on windows, devkitPro makes things easy with their installer, download that and get devkitARM and *libnds* with it. Oh, also get the examples! This tutorial will start with a template from the examples.

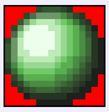
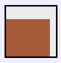
devkitARM is the *toolchain* that we can use to compile code for the DS.

libnds is a somewhat low-level library for DS programming. We will use it in this tutorial.

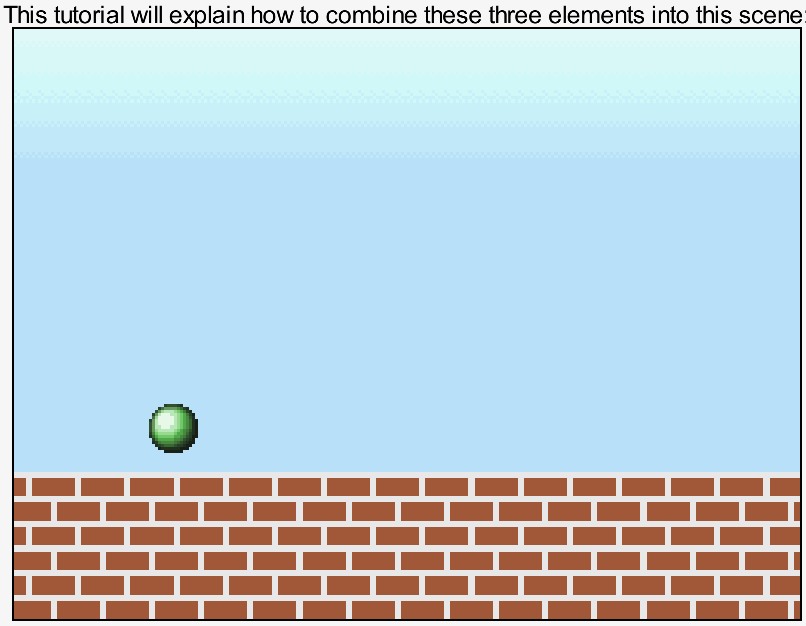
Also, how are you going to test your project? Luckily, **for this tutorial** *you don't even need a DS or flashcard*. All testing can be done with no$gba, a GBA/DS emulator. Scroll down a bit and grab the freeware version. If you're on linux, I've heard that it runs OK with *wine*.

Warning: Although this game happens to work in no$gba, do not--ahem, *DO NOT* trust emulators. Emulators are expert at hiding things in your program that are broken (because they do not emulate the hardware exactly). There is a good chance of something not working correctly on hardware if you test only in an emulator. When working on your projects, you should frequently test on hardware to make sure you haven't screwed anything up!

Finally, I will give you some graphics that you will make the game with:

Bouncy ball sprite	Funny shape	A gradient
		

This tutorial will explain how to combine these three elements into this scene:

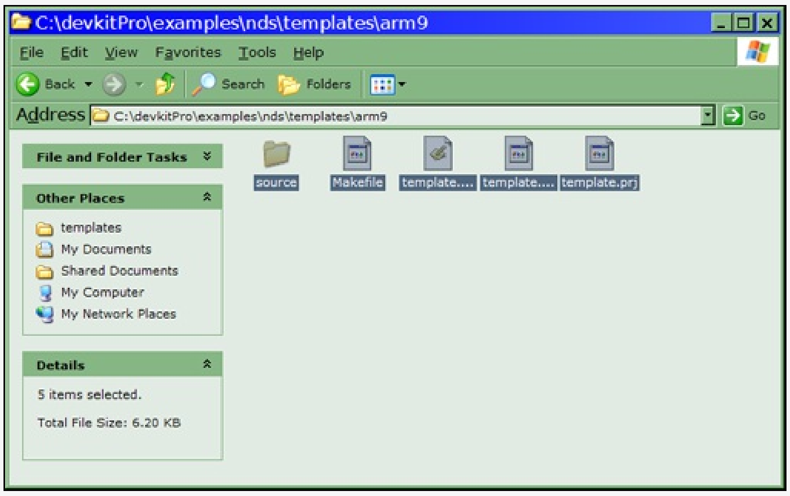


Download the graphics here!

Contents | Next: Setting up your project

Setting up Your Project

Copy the template



Now that you have devkitARM & friends setup, navigate to the examples folder (I hope you downloaded the examples!). Inside this folder there is plenty of example source code to learn from. For now, we are only interested in the DS *template*. Navigate to examples/nds/templates/arm9.

Make a little folder somewhere on your computer to hold your project files. Copy all of the *arm9* template files into it.

The basics

Now it's time to write a little bit of code. Browse to the *source* folder of your project and open up main.c. Start by deleting everything :P, keystroke: Ctrl+A, Delete.

Now with our fresh source file, we want to add the basics to it. Start by **#including** the *libnds* definitions.

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

Next, we will create the *main* function of the program.

```
int main( void )
{
    irqInit();                // initialize interrupts
    irqEnable( IRQ_VBLANK );  // enable vblank interrupt
    while(1)
        swiWaitForVBlank();

    return 0;
}
```

In the above code, we make two calls to the libnds irq handler functions. The first one enables the interrupt handler. If you don't know what interrupts are, don't worry, you'll learn more about them later. Anyway, the second call enables the *vertical blanking* interrupt, it is required for the swiWaitForVBlank function to work.

What does swiWaitForVBlank do??? It is a special function that is built into the DS BIOS (basic in/out system (some standard functions for doing things)). What it does is halts the program until the next *VBlank* interrupt occurs.

Some newbies write code like this to wait for the next vblank:

```
void vblank_wait( void )
{
    while( REG_VCOUNT >= 160 ) {}
    while( REG_VCOUNT < 160 ) {}
}
```

Whenever I see code like the example above, I go curl up into a little corner and cry. Although this code will wait for the next VBlank state, it does not halt the CPU! While the CPU is *halted* it can be considered to be in a sleeping state, waiting to be woken up by an interrupt. In the sleeping state, the CPU will consume less power. swiWaitForVBlank will *halt* the CPU until the Vblank interrupt. *swiWaitForVBlank makes your program more energy efficient!!*

If you don't know what a "Vblank" is, don't worry! It will be explained soon. Basically, the code we have here will initialize interrupts, and wait in an (energy efficient) infinite loop.

Previous: Your supplies | Contents | Next: Designing

Designing

Okay... we have our *skeleton* code, we are ready to add things to it. Before we start coding, we must think about *what* exactly we are going to make.

First lets overview some things. The DS screen resolution is 256x192 pixels, there are two screens. We have an input directional pad, A/B/X/Y buttons, two shoulder buttons, start/select buttons, and a *touchscreen* button.

Now, unlike the GBA, which only has one display engine that goes with one screen, the DS basically duplicated the display engine so it has display engine A and B. Display engine A is more powerful than B so it can be considered to be the *main* display engine. Display engine B can be called the *sub* display engine. In this tutorial we will only use the main display engine.

Both display engines can be switched into a bunch of video modes. There is a very very good document about DS hardware called GBATEK. It's written by the author of the no$gba emulator. It's called **GBA**TEK because it dates back to when GBA homebrew was very popular. Now it has a complete DS section that explains the inner workings all of the hardware in great detail. One of the sections of GBATEK contain a list of the video modes each display engine can use. I pretty much copied that section to make the table below.

Video modes

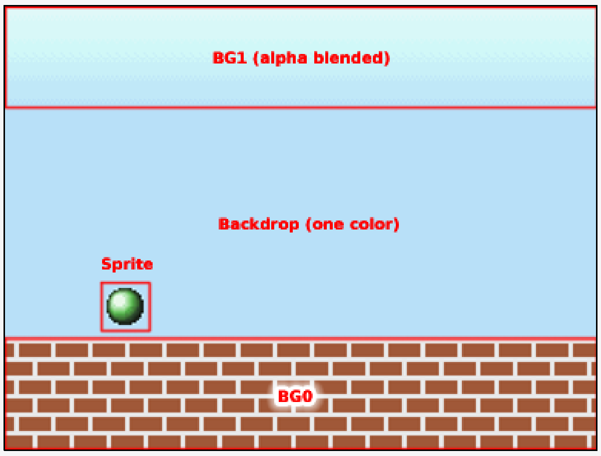
The video mode is composed of multiple settings. First, we have 7 background modes. Here is a list of the BG modes the DS can use, each BG mode has different behavior for each of the *backgrounds*.

Mode	BG0	BG1	BG2	BG3
0	Text/3D	Text	Text	Text
1	Text/3D	Text	Text	Affine
2	Text/3D	Text	Affine	Affine
3	Text/3D	Text	Text	Extended
4	Text/3D	Text	Affine	Extended
5	Text/3D	Text	Extended	Extended
6	3D	Not used	Large	Not used

Each display engine has 4 backgrounds, BG0, BG1, BG2, and BG3. We see in mode 2 that BG2 and BG3 are set to *Affine* mode. This mode similar to the one available on the GBA, it allows 8-bit tile display that can be rotated and/or scaled.

Extended BGs have superpowers and may be used to display rot/scale (rot/scale means it can be rotated and scaled!) bitmaps.

Mode 0 will be okay for our purpose in this game. Here is another picture of the game highlighting the different entities.



BG0 will be used to draw the bricks. BG1 will be used to draw the top gradient (alpha blended with backdrop). And the ball will use *sprites*. All of the BGs will use 16 color tiles, and will have no rotation effects applied, so a *Text* (plain) BG will work for us. It's called a *Text* BG because the tiles can also be called *characters*. So the *text* bg displays an array of tiles/characters on the screen.

Now, along with the BG mode, there is another setting that affects the display. GBATEK calls it the *display mode*. There are 4 display modes that the main engine can use.

Display Mode	Description

0	Display off. Screen becomes white. It is still rendered, but not displayed.
1	Normal display (backgrounds+sprites)
2	VRAM display (bitmap display from VRAM memory block)
3	Main memory display (view bitmap in main ram)

We will use mode 1 to display our backgrounds and sprite.

Now, the *sub* display engine is not as powerful as the main display engine. The sub display cannot use display modes 2 and 3, it also cannot render any 3D stuff so BG mode 6 is off limits, and BG0 is always text. The main display engine can use BG0 to display 3D graphics.

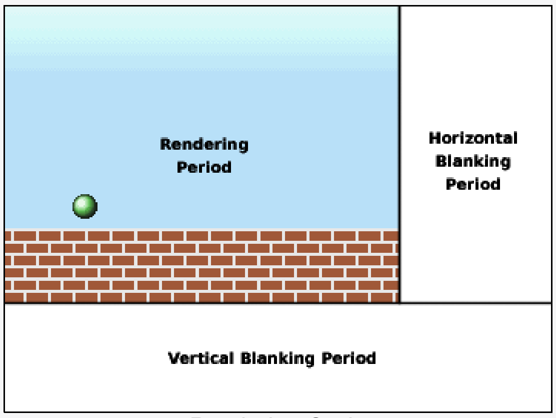
Input

The input is simple, we will use only the directional pad to tweak the velocity of the bouncing ball.

Previous: Setting up your project | Contents | Next: Program Flow

Program Flow

This is a very important chapter, many times I have seen newbies use disturbing ways of timing the flow of their program.

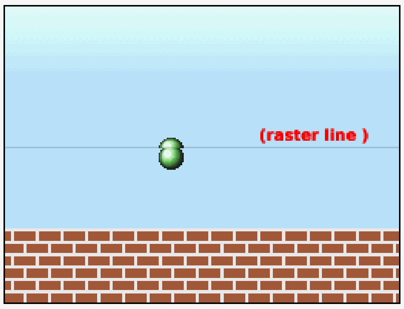


Rendering Cycle

Now, the *normal* way to do things is to time everything to the rendering cycle. The rendering cycle can be split up into 3 states, *active*/rendering, *HBlank*/horizontal blanking period, and *VBlank*/vertical blanking period. It goes through these states 60 times a second, thus giving you a fixed 60fps frame rate (well, actually about 59.8261 fps :P). The rendering cycle is always looping through these states regardless of what the program is doing.

I think with some displays, HBlank was the period of time it took to move the rendering thingy back to the left side of the screen, and the VBlank period was the time it took to move the raster line back to the top. (*I'm not sure if this is done with LCDs but it can be emulated easily*)

Anyway, during the blanking periods, the display engine isn't rendering anything! This gives you two advantages: you have fast access to the video memory, and the display engine isn't drawing anything. When the display engine is drawing, you don't want to mess with the positioning/data of certain elements. For example, see what happens when you move the ball down 5 pixels when the display engine is busy rendering it:



Yikes! The ball will be displayed like that for one frame. What happened is that half of the ball was already drawn in it's original position, and then the ball was moved down 5 pixels. The display engine then started rendering the ball again in the new position. This effect is called *shearing*, it can sometimes be used to create cool graphical effects, and it can be accidentally used to create undesirable effects.

If you don't know what a raster line is, it is the current line that is being rendered by the display engine.

Now, to avoid this problem, we can take advantage of the VBlank period. During the VBlank period, the display isn't drawing anything! Thus it is safe to change any of the graphical information. The VBlank period is limited, so your program has to be fast enough to change all of the rendering information before the screen starts drawing again.

During the rendering period, the program can do other things, like update all of the program logic. When it's done updating that, there isn't much else to do, so the program can enter a *halted* state until the next VBlank period. This is done using the swiWaitForVBlank function described in previous chapters.

One other thing is the HBlank period. This is a short period when the screen is finished drawing a single line. Sometimes you can achieve cool *raster effects* by changing certain data while the screen is drawing. To make sure that the effect runs smoothly, the HBlank period can be used to update data safely mid-frame. This is quite an advanced topic and will not be explained further throughout this tutorial.

In conclusion, your main loop should take advantage of these rendering states to ensure a smooth graphical display and *program flow*. An example main loop should look like this:

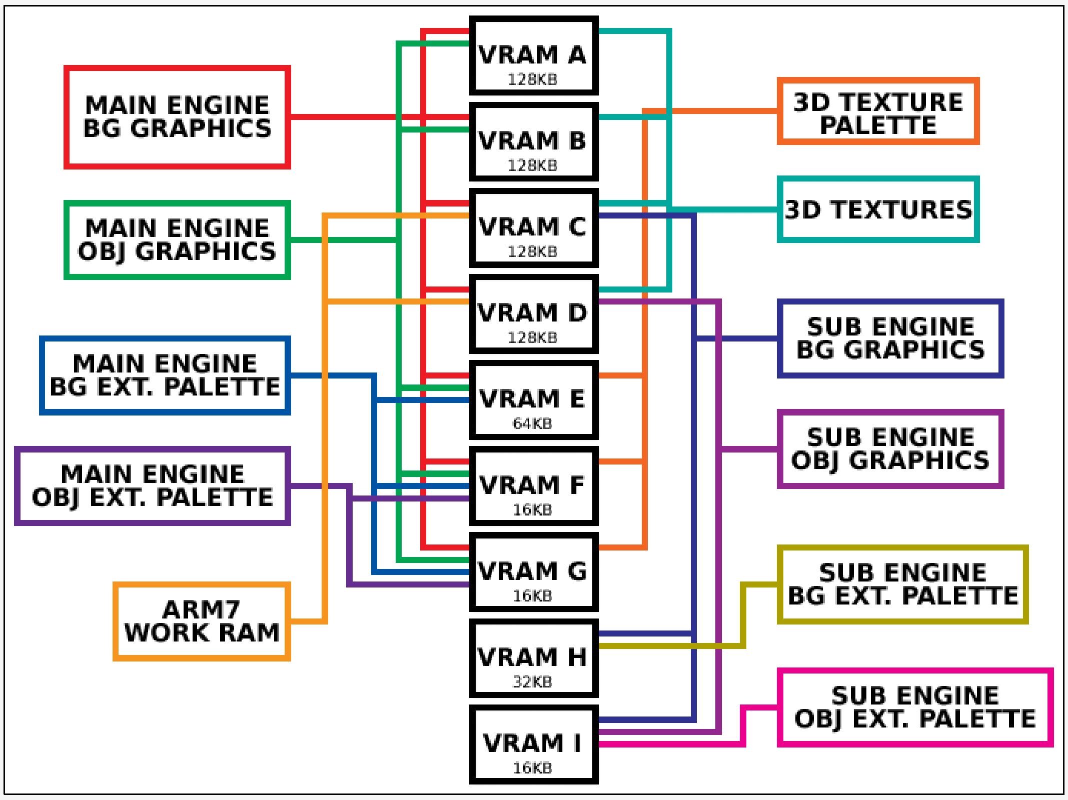
```
int main(void)
{
    irqInit();
    irqEnable( IRQ_VBLANK ); // enable vblank irq for swiWaitForVBlank
    while(1)  // infinite loops are perfectly normal when coding for consoles
    {
        // Rendering period:
        // update game objects (move things around, calculate velocities, etc)
        update_logic();

        // wait for the vblank period
        swiWaitForVBlank();

        // VBlank Period: (safe to modify graphics)
        // move the graphics around
        update_graphics();
    }
}
```

Previous: Designing | Contents | Next: VRAM banks

VRAM Banks

Before we start loading graphics, we must understand the VRAM bank settings. This is one of the things that I was stuck on for a while when switching to DS homebrew from GBA.

The DS has 9 VRAM banks, they have varied sizes and purposes. Here is a diagram that shows what each VRAM bank size is, and what they can be used for.



This is a confusing diagram ;). There is some information that is not shown, like what memory address is used when the VRAM is mapped for certain functions. If you want a *complete* description of each vram bank and it's settings, read the GBATEK article named "DS Memory Control - VRAM".

I shall now explain what the purpose is for each VRAM bank setting.

LCDC

This is a somewhat special mode that *every* VRAM bank can use (not shown in diagram). It maps the memory for fast access by the ARM9 cpu. There are a few exceptions where the video hardware can use it too though. The *display capture* hardware can write to banks A, B, C or D when they are allocated for LCDC access. The rendering engine can also display a 16-bit bitmap from banks A, B, C, or D when they are allocated to this mode.

MAIN ENGINE, BG GRAPHICS

Almost every bank can be allocated to hold graphical information for the *main* engine's backgrounds. When banks are allocated to this mode, they can be used for holding tile data for backgrounds, tilemaps, bitmap data for *extended* BGs, and maybe something else that I missed. Banks A, B, C, D, E, F, and G can use this mode. We want to use one of those banks to hold the background graphics for this game.

MAIN ENGINE, OBJ GRAPHICS

OBJ means *object* or more commonly *sprite*. Banks allocated to this mode can hold tile graphics (or bitmaps) to be displayed by sprites (or objects). We want one of these banks to hold our bouncy ball sprite graphic.

MAIN ENGINE, BG EXT. PALETTE

I haven't played with this much, in this mode, the bank can be used to hold an extended palette for use by backgrounds (with special settings). This mode is not mapped for ARM9 CPU access, so you must allocate the bank to another mode (like LCDC) to write the palette data to it.

MAIN ENGINE, OBJ EXT. PALETTE

I have never used this mode. It is like the BG EXT. PALETTE but it's for sprites.

ARM7 WORK RAM

In this mode, the vram bank is assigned to a memory region that the ARM7 CPU can access. This mode is used in GBA mode for the GBA's 256KB of external work ram.

3D TEXTURE PALETTE

When VRAM banks are allocated for this purpose, they can hold palette data for 3D stuff. The 3D textures may be in a special paletted format that will use data from banks allocated for this purpose.

3D TEXTURES

This mode is used for holding 3D textures.

SUB ENGINE, BG GRAPHICS

This is just like the *main* engine's mode, but it is a bit more limited. Only three banks can be assigned to this mode. But only 128KB max can be used with this mode; Banks H and I overlap bank C when they are allocated together.

SUB ENGINE, OBJ GRAPHICS

This one is just like the *main* engine's mode for holding sprite graphics. Notice that it is also very limited on the amount of memory that can be allocated, although this is actually quite huge compared to the GBA's 32KB of OBJ vram :).

SUB ENGINE, BG EXT. PALETTE

The sub-engine can use extended palettes too, this is for the sub-engine's backgrounds.

SUB ENGINE, OBJ EXT. PALETTE

And finally, this one is for the sub-engine's sprites that use extended palettes.

Changing the settings

Now, to change the VRAM bank *mapping*, you can use the libnds *vramSetBankX* function where X is replaced by the letter associated with the bank. There are many libnds definitions for each bank setting, you can look at *video.h* in the libnds headers to see the enumerations for them. In this tutorial we will use only 2 bank settings: **VRAM_E_MAIN_BG**, and **VRAM_F_MAIN_SPRITE**. The bank setup code should look something like this:

```
vramSetBankE( VRAM_E_MAIN_BG );
vramSetBankF( VRAM_F_MAIN_SPRITE );
```

Previous: Program flow | Contents | Next: Converting the graphics

Converting the Graphics

It's almost time to start writing some code. Before we do that, we first need to get our graphics linked into the project!

Grit

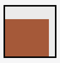
Grit = *GBA Raster Image Transmogrifier*. It is a very good program to convert graphics into a format usable by the GBA. Since DS graphics hardware is *very* similar, grit can be used to convert our graphics (and the current versions *do* have DS specific stuff).

Grit is shipped with devkitARM, so you should already have it installed. To check out all the features grit offers, open up a command prompt and type 'grit' and push enter. You should be flooded with all the options grit takes.

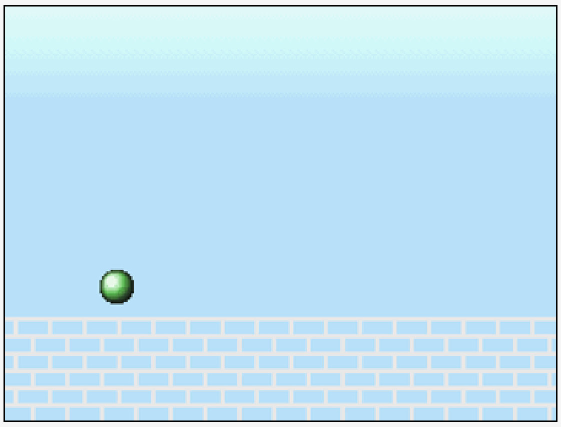
Now... we need to modify our project to automatically convert and *link* the graphics into our project. To do this, first we need a *flag file* for each of our images. A *flag file* holds all the flags for conversion for a single image, .grit is the extension used. Grit will automatically look for a flag file with the same name as the image it is converting.

Converting the Graphics

I hope you downloaded the graphics. Let's start with the *brick* graphic.



At first glance, we see that the image uses only 2 colors, BUT there is one more hidden color in the palette. If we view the palette in an editor we will see three colors: 🟥🟫⬜. The last two are the ones that form the image. The first one is the *transparent* color. When the DS is rendering graphics, the first color is not rendered. It can be used to create *transparent* parts of an image. Since the brick does not have any transparent parts, the red color is not used in the image; it is only used to reserve the first color index. If we were to ignore this rule and put one of our two colors in that spot (like the body color), our image would end up being rendered like this:

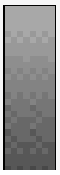


The first color of each palette is not rendered. It is transparent. So.. finally, we will make a flag file that will tell grit to make a 3 color, 4-bit image. A 4-bit image can use 16 colors, but we are only going to the first 3 of them. Make a new text file called *gfx_brick.grit*, inside, put the following:

```
# <- 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
```

This script should be sufficient for converting *gfx_brick.png* into a usable format. There are many other options that can be used, but their default settings will work for converting this image.

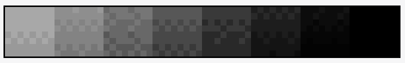
gfx_gradient.png



Next image... Here we have an 8x64 image of a gradient. This is the image that will be *blended* onto the backdrop. The DS (and GBA) has special *alpha blending* hardware that can *add* two images together, each one having a factor that the color components are multiplied by. What we are going to do with this image is give it a small blending factor, and add it to the backdrop at the top of the screen.

Now, when we convert the image, it will be transformed into linear data that can be transferred directly to VRAM. Grit will cut

the image into 8x8 pieces for us. It should probably look more like this when it is loaded into tile VRAM (8 tiles).



If you're wondering why we want all of our images cut into 8x8 pieces, it's because the backgrounds construct an image by putting these pieces together. Another way to draw things is to just render bitmap data onto the screen, but this method is much slower and wastes the capability of the graphics hardware. Don't worry if you are still confused, the next chapter will explain more about the tiles.

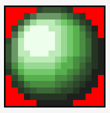
Okay, let's make a flag file for the gradient:

```
# the gradient has 16 colors
-pn16

# the gradient is a 4-bit image
-gB4
```

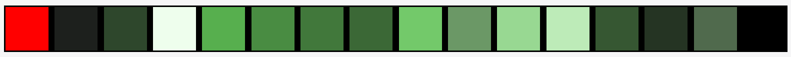
Quite simple really, almost the same as the brick script, but with a wider palette. Save this file as *gfx_gradient.grit*.

gfx_ball.png

The last graphic we have is our bouncy ball!

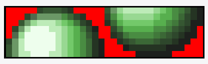


Here is a picture of the palette it uses:



Notice the first bright red color. Since it is in the first entry of the palette, it will not be shown when the ball is rendered onto the screen. It will be transparent!

We need a flag file to convert this 16x16 image into 8x8 tiles, it needs to look like this when we load it into video memory:



The rendering engine accepts this format when drawing sprites larger than 8x8 (it goes left->right, top->bottom). There is also another mode, "2D mode" where you can load non-linear graphics and treat the obj vram like a 32x32 tile image. We will use 1D linear mode in this tutorial though.

In the flag file we need to tell grit to export a 16 color, 4-bit image, just like the gradient one.

```
# convert ball graphic

# 16 colors
-pn16

# 4bpp
-gB4
```

Save this file as *gfx_ball.grit*.

Linking the Graphics

Okay! Now that we have our graphics and *flag files*, we can give our makefile some rules to tell grit to convert our graphics.

Start by making a sub-directory in your project folder, call it *gfx*. The default makefile will list *gfx* in the source files.

Open up the makefile, scroll down to the part that looks like this:

```
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)))
```

Add another line to make a list of PNG files for conversion.

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

Next, modify the OFILES part by adding our PNG files list to it. (Add the underlined text)

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

Now scroll down to the area right below the *main targets* section. Add this special grit *rule* for converting our png files.

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

This tells the makefile that it can use grit to convert a PNG & GRIT file into a source file and a C/C++ header file. The -fts flag tells grit to output an assembly source file.

Now, when you build your project, your graphic files should be automatically detected and converted into object files and linked into the project.

Previous: VRAM banks | Contents | Next: Loading the graphics

Loading the Graphics

Phew, it's finally time to write some code! Open up your *main.c*, it should look like this: (from earlier chapters)

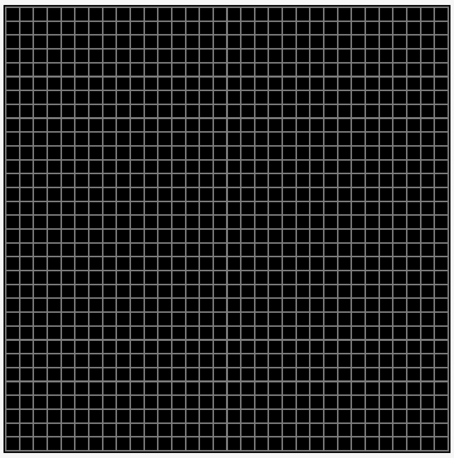
```
#include <nds.h>

int main( void )
{
    irqInit();
    irqEnable( IRQ_VBLANK );

    while(1)
    {
        swiWaitForVBlank();
    }
    return 0;
}
```

Create a new function, it will be used to load all of the graphics and setup the display mode. I'll call it **setupGraphics**(). Add this function to main() after setting up the interrupts.

The first thing we want to do in it is setup the VRAM banks we will use. Bank E will be used for the backgrounds (next chapter), and Bank F will be used for sprites (upcoming chapters). Setup both to be used by the main engine.

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

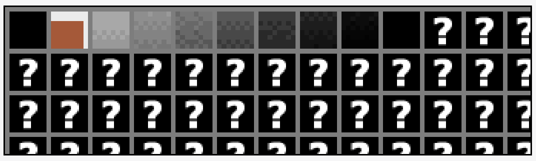
Here is a visual representation of *half* of VRAM bank E when no data is loaded.



Each of those cells represent a 16-color tile entry. Each 8x8 16 color tile uses up 32 bytes of memory (8 * 8 * 1/2 bytes).

BIG WARNING: Although the memory may be thought of as blank when your program starts, it may not really be blank! It may contain random garbage, or leftovers from whatever booted your code. This is one of the biggest issues that occur when switching from emulator to hardware. When programming, do *not* expect anything to be initialized to zero.

Now, lets zoom in on the upper left corner of this block. This is what we can imagine the VRAM will look like up there when we load our tiles.



We will clear the first tile to zero during the setup function. We will copy the brick tile into tile 1, and the gradient tiles into tiles 2->9. The question marks represent uninitialized data (see above warning).

To copy the tiles, we will use the dmaCopyHalfWords() function. This function uses the DMA (direct memory access) hardware to copy blocks of data from one place to another. We are going to copy the data from our program into VRAM.

First, include the references that were produced by grit, also make some definitions of where the data will be loaded.

```
//-----------------------------------------------
// graphic references
//-----------------------------------------------
#include "gfx_ball.h"
#include "gfx_brick.h"
#include "gfx_gradient.h"

//-----------------------------------------------
// tile entries
//-----------------------------------------------

#define tile_empty     0 // tile 0 = empty
#define tile_brick     1 // tile 1 = brick
#define tile_gradient  2 // tile 2 to 9 = gradient

// macro for calculating BG VRAM memory
// address with tile index
#define tile2bgram(t) (BG_GFX + (t) * 16)
```

Now in your setup code, copy the graphics into vram...

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

    // generate the first blank tile by clearing it to zero
    int n;
    for( n = 0; n < 16; n++ )
        BG_GFX[n] = 0;

    // copy bg graphics
    dmaCopyHalfWords( 3, gfx_brickTiles, tile2bgram( tile_brick ), gfx_brickTilesLen );
    dmaCopyHalfWords( 3, gfx_gradientTiles, tile2bgram( tile_gradient ), gfx_gradientTilesLen );

    ...
}
```

There are a few things here that need explaining. First of all, we have *BG_GFX*. This is a pointer defined by libnds that gives the base address for the main engine's BG VRAM.

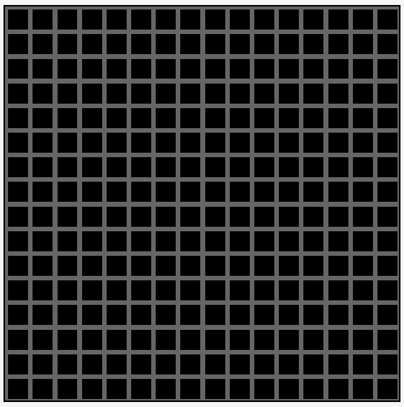
tile2bgram is a small macro that calculates the offset of a certain tile in the BG graphics memory. It adds *t * 16* which really adds t * 32 bytes because BG_GFX uses 16-bit entries.

We also see direct access to BG_GFX when we write the first empty tile, each tile is 32 bytes, so this will write 16 half-words (16-bits) to the first tile in bg memory.

The last thing is the parameters of dmaCopyHalfWords. The first parameter is the DMA *channel* selection. There are 4 DMA channels (0->3) that can be used for different things. Channel 3 may be used for all general purpose transfers (so we will use it here). Channels 1 & 2 have higher priority than channel 3 (with GBA, 1&2 were used for transferring sound data). Channel 0 has the highest priority and can be used to transfer critical data. The second parameter of dmaCopyHalfWords is the source address. The third parameter is the destination address. The last parameter is how much data to copy. X bytes of data (replace X with the last parameter) will be copied from the source address to the destination address.

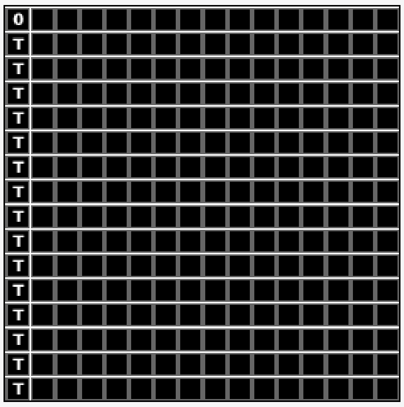
Palettes

When 16 or 256 color graphics are used, they are rendered using a *palette*. The main engine's palette memory is mapped for access at address 0x5000000->0x50003FF. Each entry is a 16-bit color value. Here is a visual representation of the first 256 colors of the palette memory when it is empty (but remember that it may *not* be empty when you load the game on hardware!).



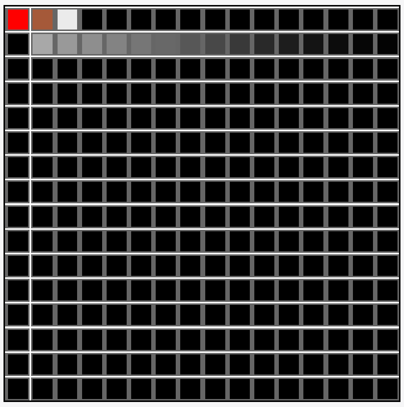
The first 256 color block is used by the backgrounds, the other 256 color block after is used by the sprites. Back in the SNES days, you only had one 256 color palette for both :(.

Now, in 16 color mode, this palette is divided up into 16 pieces, you can select any one of them to render any 16 color graphic. With 256 color graphics, there is no palette selection, because they can use the whole palette. Here is the palette divided into 16 sub-palettes.

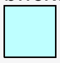


Now, the first color of each palette is special. It is the *transparent* color, this color is not ever seen on the screen, it is used to make *see through* parts of an image. To make parts of your image transparent, paint the transparent parts with a unique color, and fix that color at index 0! In 256 color mode, only index 0 of 255 is transparent.

One more exception though, although colors 16,32,48,64,etc are never rendered when using 16 color graphics, color 0 may be rendered! (top left color 0, as seen in image) This color is the *backdrop*. If the screen isn't completely covered by graphics, anything left over will be filled with the *backdrop* color. Here is what the palette will look like when we are done loading it.



There is something wrong here, it's the first backdrop color. Right now it's loaded with the hidden red transparent color for the bricks, but it will be used for the backdrop! After we are done loading the palettes, we will overwrite this entry with a blueish color.



Now lets copy the palettes, start by adding a few definitions for placement.

```
// BG PALETTES
#define pal_bricks        0    // brick palette (entry 0->15)
#define pal_gradient      1    // gradient palette (entry 16->31)

#define backdrop_colour   RGB8( 190, 225, 255 )
#define pal2bgram(p)      (BG_PALETTE + (p) * 16)
```

Now at the next section of *setupGraphics*:

```
// palettes goto palette memory
dmaCopyHalfWords( 3, gfx_brickPal, pal2bgram(pal_bricks), gfx_brickPalLen );
dmaCopyHalfWords( 3, gfx_gradientPal, pal2bgram(pal_gradient), gfx_gradientPalLen );

// set backdrop color
BG_PALETTE[0] = backdrop_colour;
```

BG_PALETTE is a libnds pointer that addresses the main engine's background palette. We set index 0 in the end to our backdrop color.

We can have a chance to see some results now, let's setup a test video mode. We'll use bg mode 0 without any backgrounds enabled, and normal display.

```
swiWaitForVBlank();
videoSetMode( MODE_0_2D );
```

Some people don't care about waiting for a new frame before they set the video mode. It's not so important, but it prevents a tiny bit of crap from showing when you set the display.

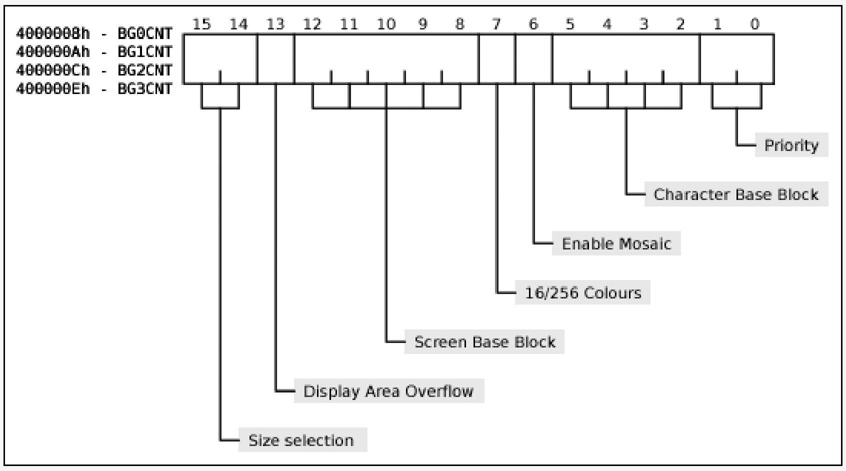
Compile the code (run make from the program directory/alt+1 if you are using programmers notepad). Run the produced .nds file with no$gba. You should see your blue backdrop displayed.

Previous: Converting the graphics | Contents | Next: The background

Backgrounds

Finally, a somewhat interesting chapter... Here we will discuss how to setup the 2 backgounds we will use.

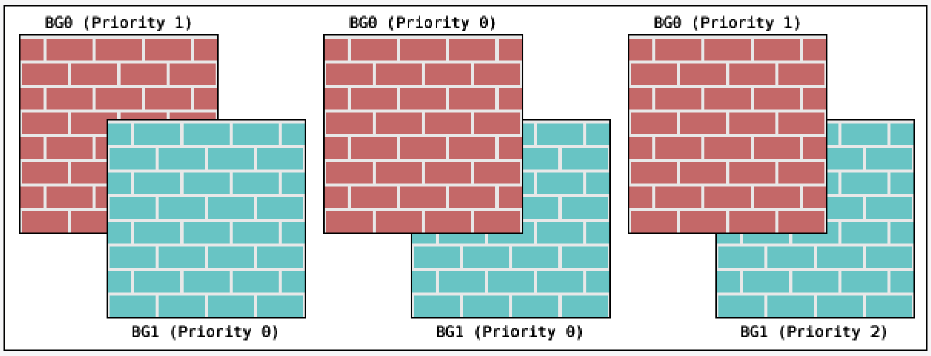
Now, to configure the background settings, we must mess around with some *hardware registers*. Hardware registers are things that may be accessed by the program to control certain hardware activity. For the DS and GBA, the hardware registers (or in/out ports) may be accessed at memory region 0x4000000 -> 0x4FFFFFF. Not every address is assigned to a hardware register (there aren't *that* many!). The register we will modify to control the BG behavior is the BGxCNT registers. There are 4 of them, BG0CNT, BG1CNT, BG2CNT, and BG3CNT. They hold the controls for each background.



Here we see the BGxCNT registers picked apart into the different values they hold. Each value uses a certain number of *bits* in the register. Here is an explanation of what each setting affects.

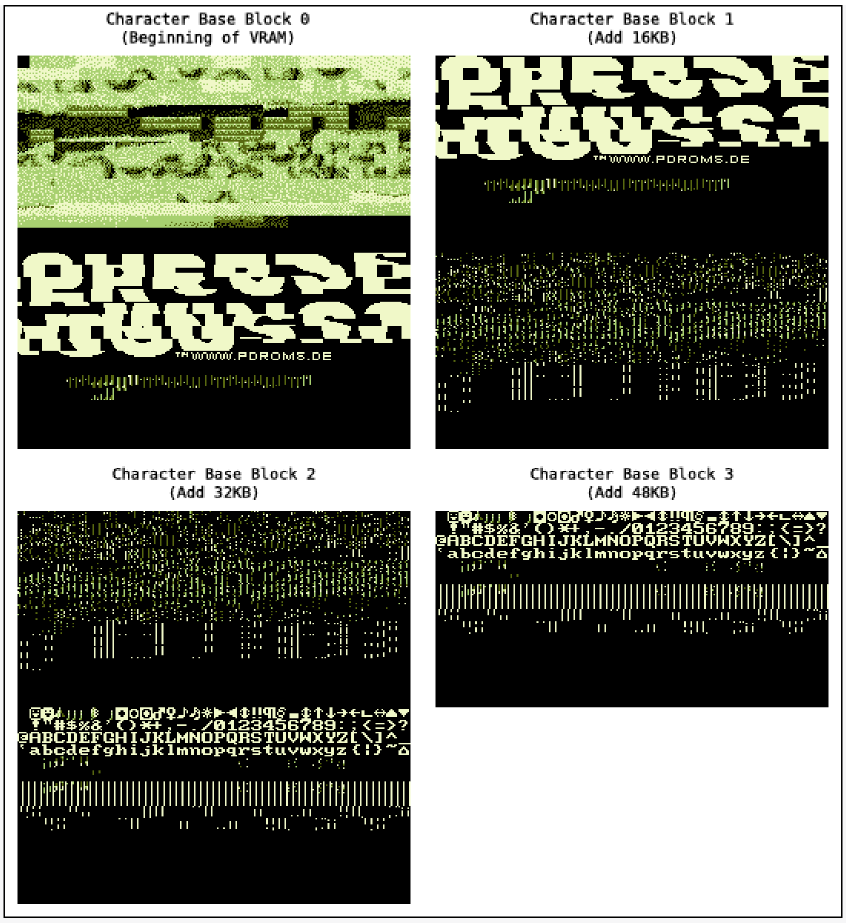
Priority (0..3)

This is the backgrounds *priority*. When mixing backgrounds together without setting *priority*, BG0 will be displayed on top of everything, BG1 will be displayed under BG0, BG2 under BG1, and BG3 under BG2. BUT, if you change the priority value, you can modify this ordering. *Lower* priority values indicate *higher* priority. If BG0 has priority 1, and BG1 has priority 0, then BG1 will be displayed on top of BG0.



Character Base Block (0..15)

Now.. one limitation of 16-color backgrounds is that each map entry has a 10 bit number to select a tile. This gives you 1024 tiles to choose from. 1024 tiles can fit in 32KB of memory, but the large VRAM banks are 128KB of memory each! Now, although you can only use 1024 tiles at a time for a single BG, you can control WHERE in the memory the tiles are picked. This is done with the *character base block*. The character base block adds N * 16KB to the tile base offset. Here is a picture of increasing character base blocks for my GBA game (works just about the same on DS).



On the GBA, you only had 64KB of BG video memory, so the character base block could only be 0->3, bits 4-5 of BGxCNT were reserved. For DS, the two bits were used to extend the character base block selection so you could access much more memory.

In the above picture, we see some interesting things. In block 0, the tiles for the terrain below were stored. We see some remnants from the title screen in the unused bottom half.

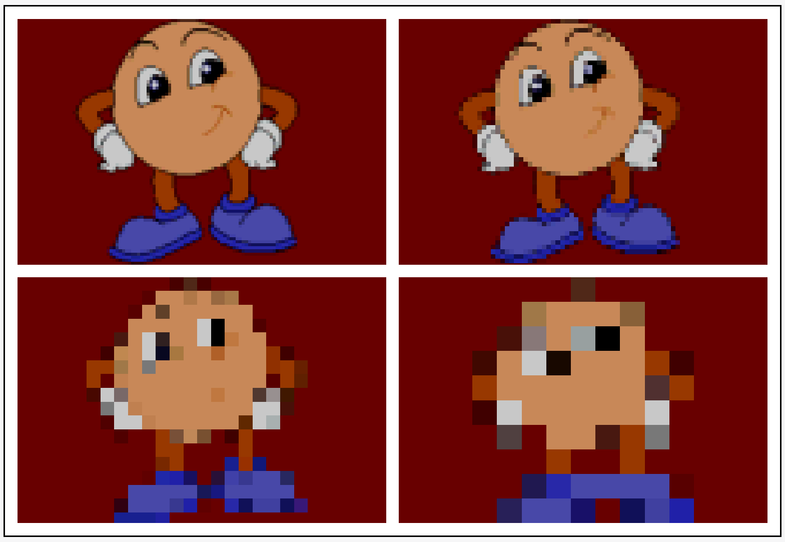
In the next block (CCB 1), we have scrolled down 16KB. Now we can see another 16KB of memory. This block is mostly filled with BG *map* data (the corrupted looking stuff).

In the next block, we are at the 32KB mark (2 * 16KB). Here we see the bottom half of CCB 1, and we see the next region, which is filled with our text characters!

In the last block, the base is set directly to the address of the text characters, this block (block3) was used by one of the BGs in the game to draw text on the screen. I set it up like this so I could simply write an ASCII number into the tile entry, and it would select the associated character.

Mosaic (0..1)

The seventh in the control register enables *mosaic* effects. The GBA and DS (and snes!) have special hardware that can apply a "mosaic" effect to the image. You can see this effect in Super Mario when you enter doors, or start a level. See below different levels of *mosaic* (yes I just applied the effect over one of the devkitPro GBA examples).



The actual mosaic levels are set in another register.

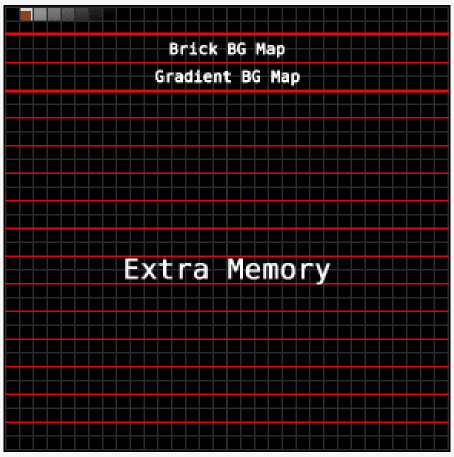
LOL, it looks like he dropped his pants!

16/256 Colours (0..1)

This register controls the tile format of the BG. If it is 1, then the BG will display 256-color tiles. We will set this bit to 0 to use 16-color tiles.

Screen Base Block (0..31)

This is another important setting, this is like the Character Base Block, but it selects the *Screen* Base Block. The Screen Base Block is the offset added to the base of video memory to select where the background will read the *map data*. The map data for a single 32x32 BG will use 2KB of memory. The Screen Base Block is added to the base of VRAM in 2KB steps. Specifying 5 will use the memory starting at the 10KB mark.



Above we see an image of our VRAM marking each screen base block in red lines. Each block is 2KB wide, so each uses the space that can hold 64 16-color tiles.

We need to select blocks to hold the data for our two backgrounds. We can't use block 0 (our tiles are stored there!). We can use block 1 for storing the "bricks" BG, and we can use block 2 for storing the "gradient" BG. There is *alot* of extra video memory left that may be used for more complex games.

Display Area Overflow

This bit isn't used in *text* BGs. For bitmap/affine BGs this bit controls what is seen when outside of the BG area is shown. If it is zero, the area will be transparent. If it is one, the background will *loop* when you scroll outside its boundaries. Text BGs will always *loop*, so we don't need to care about this bit for our purposes.

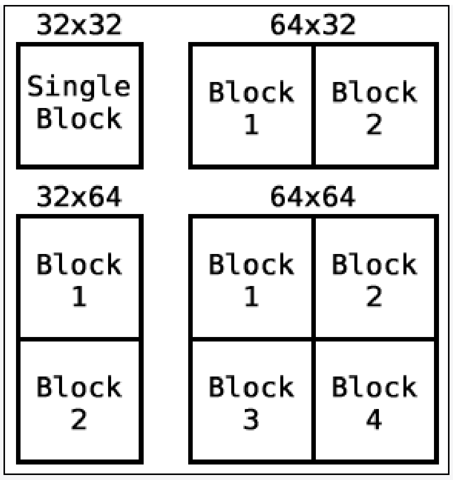
Size (0..3)

The last attribute controls the *size* of the background.

Size	Dimensions
0	32 by 32 tiles (256x256 pixels)
1	64 by 32 tiles (512x256 pixels)
2	32 by 64 tiles (256x512 pixels)
3	64 by 64 tiles (512x512 pixels)

This table only applies to text backgrounds. When rot/scale backgrounds are used, there are different sizes and behavior of the map data.

When sizes other than 0 are used, the text background will use more than 1 screen base block. Also, dont expect a nice linear block when using sizes other than 0! When using other sizes, the background is composed of multiple 32x32 blocks.



Setting up the Controls

What settings will we write to our BG controls? Well, the BG graphics aren't going to overlap, so we don't have to use the priority bits. The tiles for both BGs are at the very beginning of memory, so we don't have to use a character base block offset. We aren't using Mosaic. We are using 16 color tiles (0 setting). We want a normal 32x32 size bg (also 0). Display Area Overflow isn't used. The only thing we need to set (the only non-zero value) is the Screen Base Block. libnds defines this as BG_MAP_BASE(n); it shifts n to the left 8 bits so it corresponds to bits 8->12 in the BG control.

We will set BG0 to use block 1 and BG1 to use block 2.

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

All that explanation for 2 lines??? Well, the explanation of the other stuff may come in handy later. :P

The Tilemap

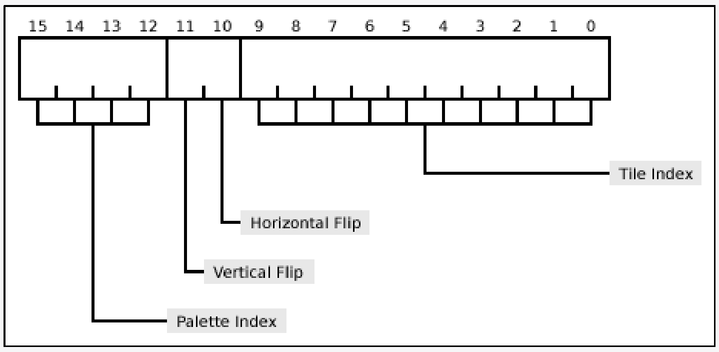
Before the BGs can display anything, we must fill up their tilemap with data!

Let's add two definitions to point to VRAM where our BG Screen Base Blocks point to.

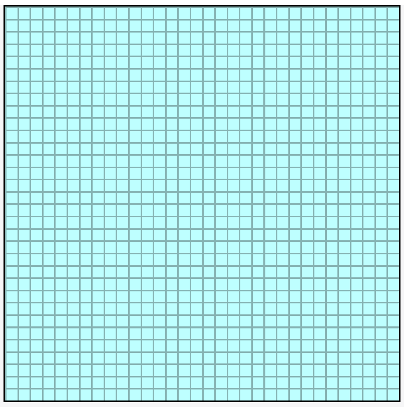
```
#define bg0map    ((u16*)BG_MAP_RAM(1))
#define bg1map    ((u16*)BG_MAP_RAM(2))
```

BG_MAP_RAM is a libnds definition that gets a pointer to VRAM at with a specified screen base block.

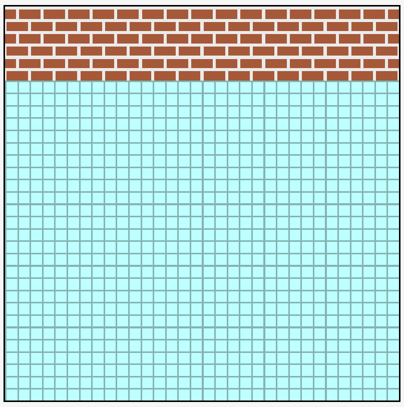
Each entry in the BG *map* is 16 bits wide, let us analyze these bits!



Now, in the 32KB screen block we can fit 1024 16-bit tile entries, the 32x32 tilemap is stored in the block linearly from left->right, top->bottom. Here is a picture of the layout when the tilemap is empty (using our empty tile 0, our backdrop, and a light grid).

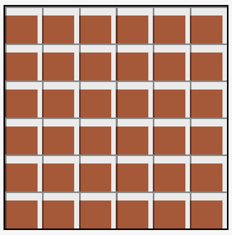


We will write data into the tile entries so we get something that looks like this:

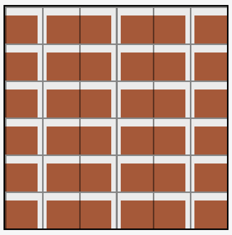


The top 6 lines will be filled with brick data, the other tiles will be cleared to zero.

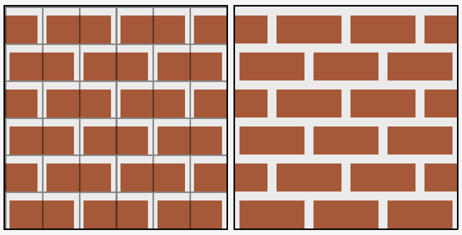
Now, you may be confused how those bricks were made with our single tile! This is when the horizontal flip bit come in handy. Below we see a picture of what the bricks will look like without *flipping* the tiles.



Pretty ugly huh, now see what happens when we apply the "horizontal flip" flag to every odd column.



Still not quite right, but better. Let's stagger the image by flipping odd columns for even rows, and even columns for odd rows.



That's more like it :P. Let's write the code!

Generating the Bricks' Tilemap

We will start by clearing the entire bricks' tilemap to zero.

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

Now let's draw the 32x6 image of bricks.

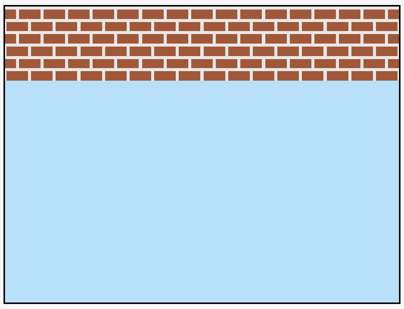
```
    int x, y;
    for( x = 0; x < 32; x++ )
    {
        for( y = 0; y < 6; y++ )
        {
            // magical formula to calculate if the tile needs to be flipped.
            int hflip = (x & 1) ^ (y & 1);

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

See how *hflip* was shifted left 10 spaces, and *pal_bricks* 12 spaces. This matches the diagram shown before. We also use the tile index of the brick that we defined in the *loading* chapter.

Let's see how this looks, add **DISPLAY_BG0_ACTIVE** to your video mode to enable your shiny new BG.

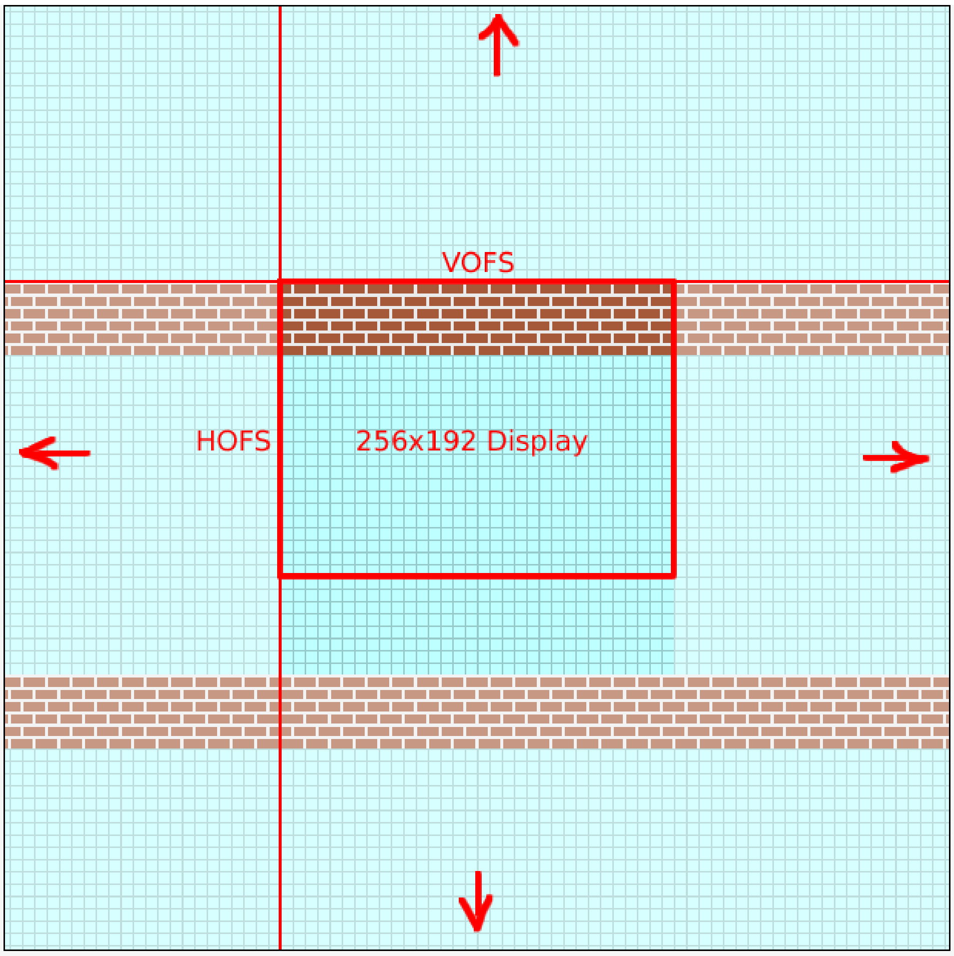
```
    videoSetMode( MODE_0_2D | DISPLAY_BG0_ACTIVE );
```

Compile the project and open the NDS file in no$gba. You should see this image on the top screen.



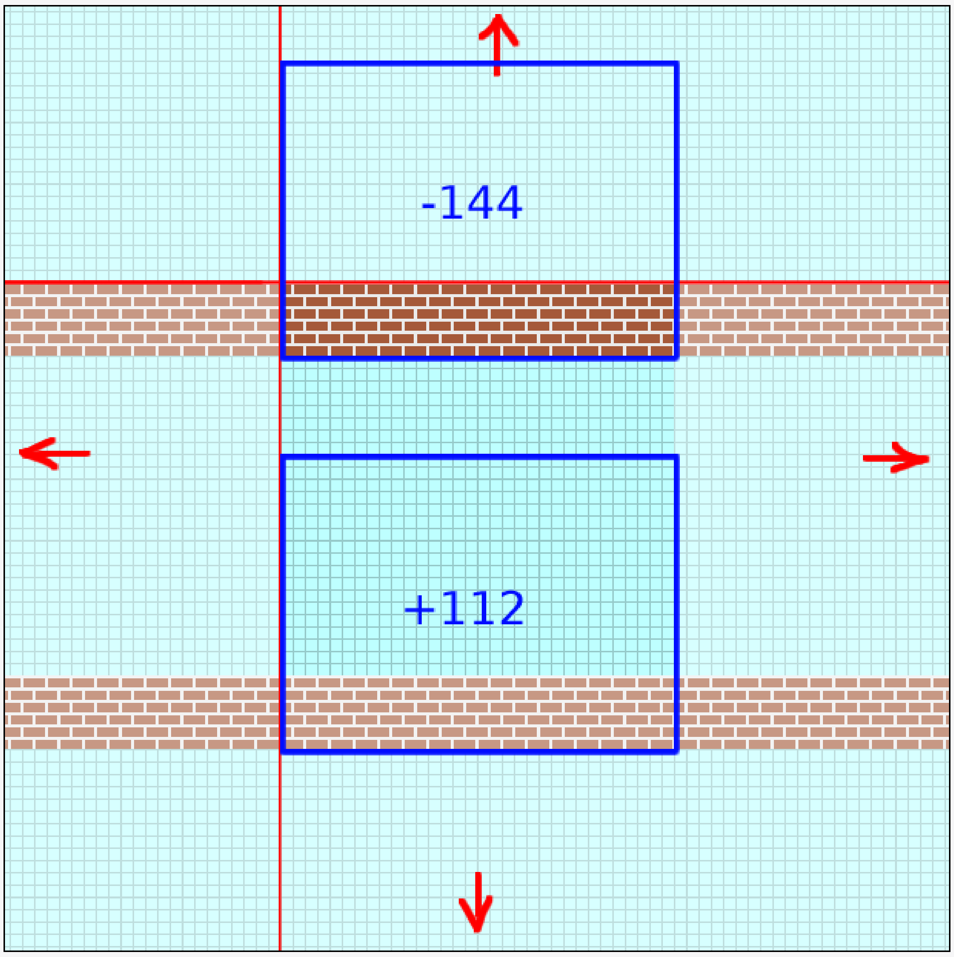
Whoops, the brick platform is at the *top* instead of the *bottom*! Let's fix this by using the *scrolling* registers.

We have two scroll registers for each bg, they are called BGxHOFS and BGxVOFS where x is replaced by 0->3. They mean *BG Horizontal Offset* , and *BG Vertical Offset*. Technically they aren't really offsets of the BGs. They are more like offsets for the display. Look at the image below.



When we move the display to show a part that is outside of our 256x256 background region, the region will loop back to the opposite side and start drawing from there. The faded parts in the picture represent the "looped" parts of the background.

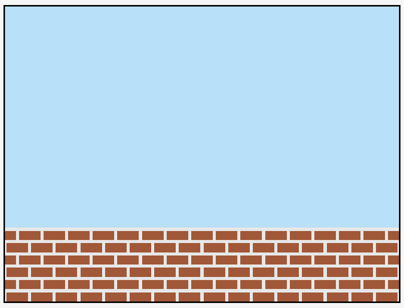
Now, we can solve our problem with the vertical offset register in two ways. We could move the display upwards 144 pixels (offset = -144), or we could move the display down 112 pixels (offset = 112). Both will look exactly the same.



Let's use the 112 option. Add this single line to your setup code.

```
    REG_BG0VOFS = 112;
```

Compile the code and run the NDS file in no$gba. You should see this now:



Ah, much better. ;)

The Gradient

We will use BG1 to display the gradient. Start by clearing the map data.

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

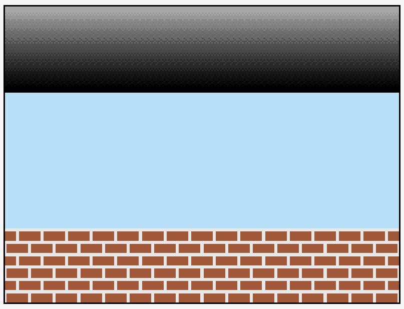
Next, fill the upper 256x64 region (32x8 tiles) with the gradient tiles. Choose a gradient tile according to the row number, and repeat it for each column.

```
    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);
        }
    }
```

And enable the background in your videoSetMode call...

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

Compile the code and run the game... You should get a ugly looking gradient pasted over your scene.

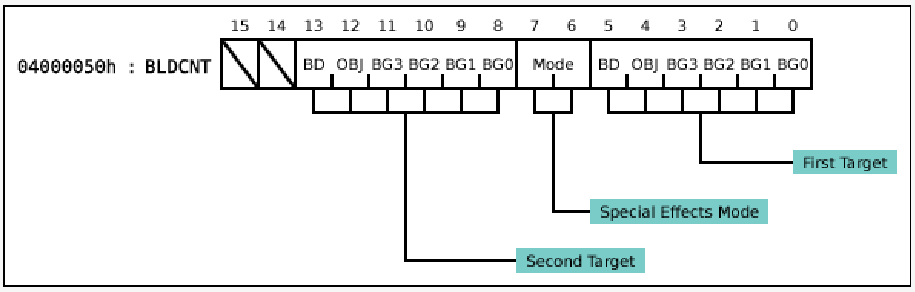


The next chapter will explain how to blend the gradient onto the backdrop.

Previous: Loading the graphics | Contents | Next: Alpha blending

Alpha Blending

Let's start with the BLDCNT register. This register controls the special effects that will be applied to the video.



BLDCNT modes

There are 4 modes you can select from (bits 6..7).

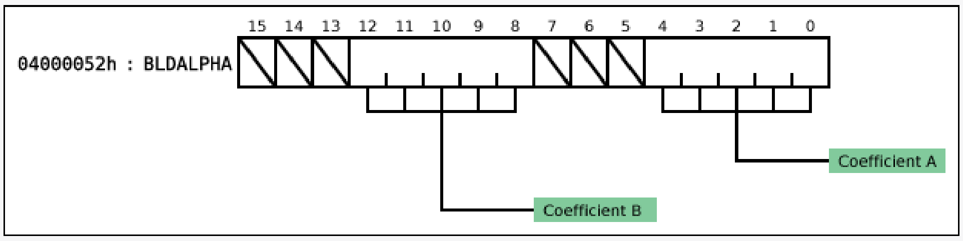
Mode 0: Disabled

This mode is the *disabled* mode. No effects will be applied to the output.

Mode 1: Alpha Blending

In this mode, the elements that are selected in the targets are *alpha blended* with each other. There are a few limitations that must be followed. First of all, only 2 elements for each pixel can actually be blended together. The top-most pixel will be blended with the second top-most pixel. The top-most element(s) must be selected in the first target. While the elements below it must be selected in the second target.

To blend the pixels, the hardware multiplies the RGB components by a value specified through the BLDALPHA register. Each target has a coefficient to be multiplied by.



The coefficient values are 5 bits, but the value ranges from 0->16. Values 17..31 are read as 16.

The first target is multiplied by coefficient A and the second by coefficient B. The forumla is something like this: *Pixel = MAX(31, (1st * A + 2nd * B) / 16)*.

We will use this mode to blend the gradient against the backdrop.

Modes 2 & 3: Fade Image

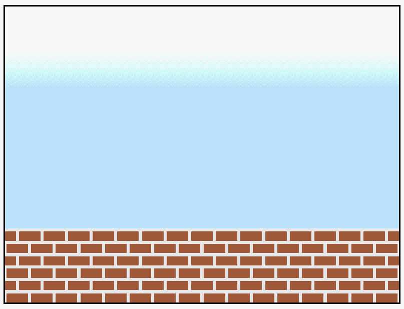
These modes only use the first elements selected in the first target. Mode 2 will increase the brightness of the image, and mode 3 will decrease the brightness of the image. The brightness/darkness coefficient is specified in the BLDY register. The BLDY coefficient ranges from 0->16. The output formula may look like this (mode 2): *Pixel = A + ((31-A) * Y) / 16*, and mode 3: *Pixel = A - (A * Y) / 16.*

Blending the Image

Okay, now lets try to blend BG1 against the backdrop. Select BG1 in the bits of the first target, and the backdrop in the bits of the second target. Also set the alpha blending mode.

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

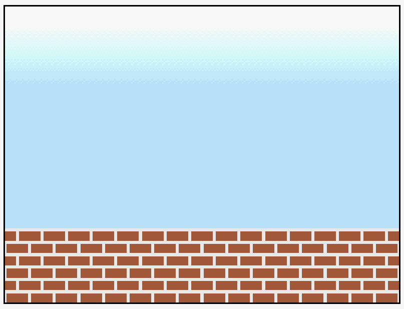
Source/SRC and *Dest/DST* are alternate names for the first and second targets. BLDALPHA was set to 100% (16/16) for both coefficients, so this will fully add both elements together. Compile the code and view the output:



Oops, that's a little bright, lets try 50% for the first target.

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

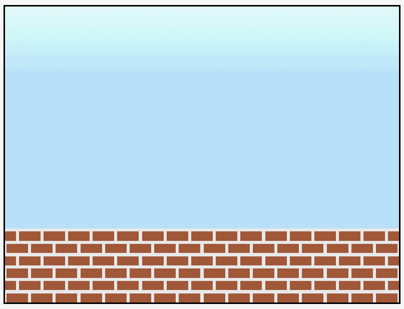
New output:



Heh, that's still a bit bright. You should tweak the coefficient until it looks okay. I settled with 25%:

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

And.. that should look like this:

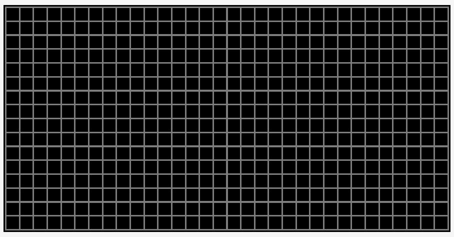


It still looks a bot wierd, but it'll have to do. :)

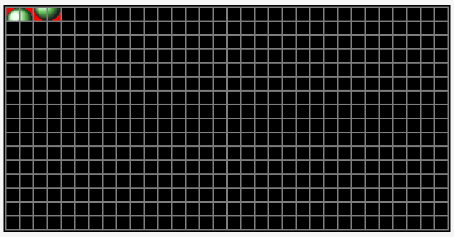
Previous: Backgrounds | Contents | Next: Loading sprites

Loading Sprite Graphics

We have allocated VRAM Bank F for holding our sprite graphics in a previous chapters. Bank F is 16KB big, so it can hold up to 512 16-color tiles.



We will load the 16x16 graphic into tiles 0->3.



First, make some definitions to where your going to load things. Also add a neat definition to translate tile numbers into VRAM addresses.

```
#define tiles_ball      0 // ball tiles (16x16 tile 0->3)

#define tile2objram(t)  (SPRITE_GFX + (t) * 16)
```

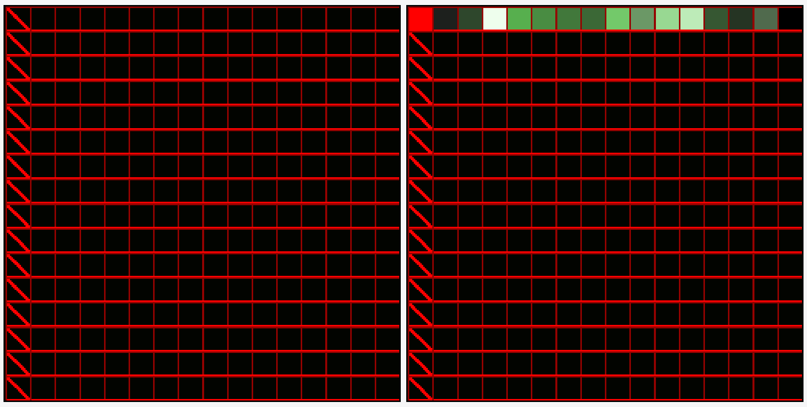
Now DMA copy the tiles into VRAM (in your setupGraphics functions).

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

And the Palette...

We need to copy the palette to the sprite palette ram. The main engine's sprite palette is located at addresse 0x5000200->0x50003FF. The layout is the same as the BG one, with the first color of each 16 color palette transparent (and only color 0 is transparent in 256 color mode).

We will load our palette to the first 16-color sprite palette.



Add this definition to calculate an address in the sprite palette. SPRITE_PALETTE is the libnds definition for this region. Also add the palette index you will use.

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

Now DMA copy the palette into the memory.

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

We are now ready to start adding sprites to the display.... In the next chapter!

Previous: Alpha blending | Contents | Next: Displaying sprites

Sprites

Okay, now it's time to add some sprites to our scene!

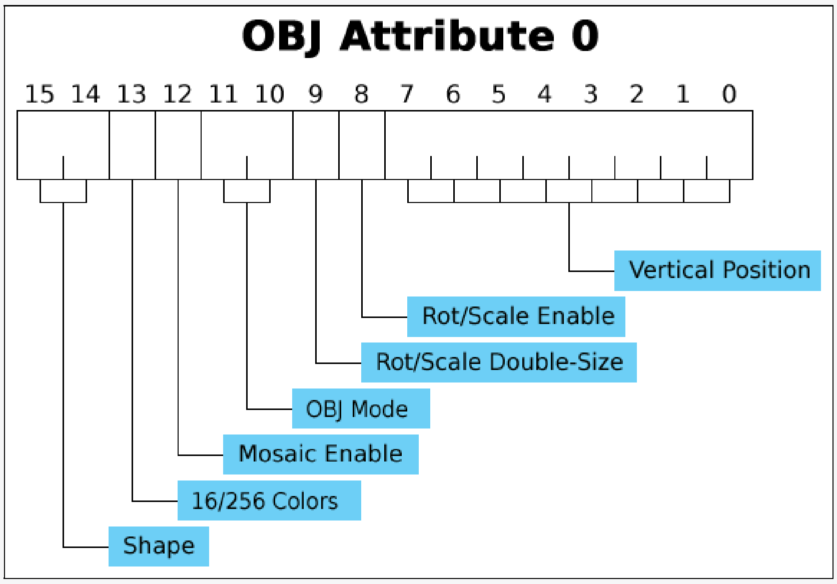
How do the Sprites Work?

Well, first of all, you have a big array of *sprite attributes*. The area of memory that holds this is called the *OAM* (Object Attribute Memory). The main engine's OAM is mapped to memory address 0x7000000->0x70003FF, it's 1024 bytes large. Each sprite entry takes up 8 bytes (actually 6, but we'll round up), so you can have 128 sprites defined in the OAM at a time. (wee you can draw 128 sprites on the screen!)

Now, for each sprite entry there are 4 Hwords of data. The first Hword in the sprite entry is *Attribute0*. The second, *Attribute1*. And the third, *Attribute2*. The last Hword isn't a sprite attribute, but it is used to define rotation/scaling parameters.

Each of the OBJ attributes define how the sprite will look when it is to be displayed. Let's pick apart the attributes.

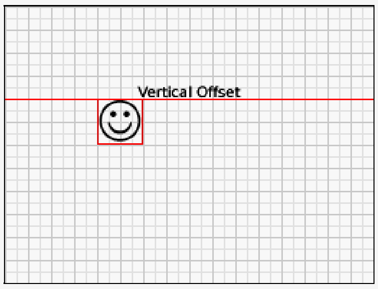
Attribute 0



Here is the first attribute... It controls a bunch of things.

Vertical Position

This is where the top-left coordinate of the sprite will be when it is drawn on the screen (measured in pixels).



This number can be negative too, to show the sprite crossing the top boundary. For negative numbers, the value is specified in twos-complement format (just AND the value with 0xFF and you'll be fine).

Rot/Scale Enable

This *flag* enables rotation/scaling effects to be applied when rendering the sprite. We'll look into this more later.

Rot/Scale Double Size

When the Rot/Scale flag is 1, then this flag will double the boundary size of the sprite. This is useful to prevent the edges of the sprite from being clipped.

IMPORTANT: When the Rot/Scale flag is 0, then this is the *disable* flag. Setting this bit for non rot/scale sprites will *disable* the sprite from being rendered. Back in the early GBA days, it seems people didn't know about this bit much, so to hide the sprites, they would move the vertical position off of the screen (or worse, the horizontal position). That method will effectively *hide* the sprite, but it does not stop it from using the rendering cycles. Setting this bit to disable the sprite will save you some rendering cycles!

OBJ Mode

This value ranges from 0->3.

When this value is 0, the sprite will act *normally*. :P

When this value is 1, then this sprite will be selected for the alpha blending, first target, regardless of the OBJ setting in the BLDCNT register.

When this value is 2, then the shape of this sprite will be added to the OBJ window. The OBJ window is something that masks certain parts of some backgrounds/other sprites with pretty shapes (shapes made by sprites put together).

When this value is 3, then this is a *bitmap* sprite. These can display 16-bit direct color images.

Mosaic Enable

This bit enables mosaic effects to be applied to the sprite. See the *backgrounds* chapter for an example of what mosaic looks like.

16/256 Colors

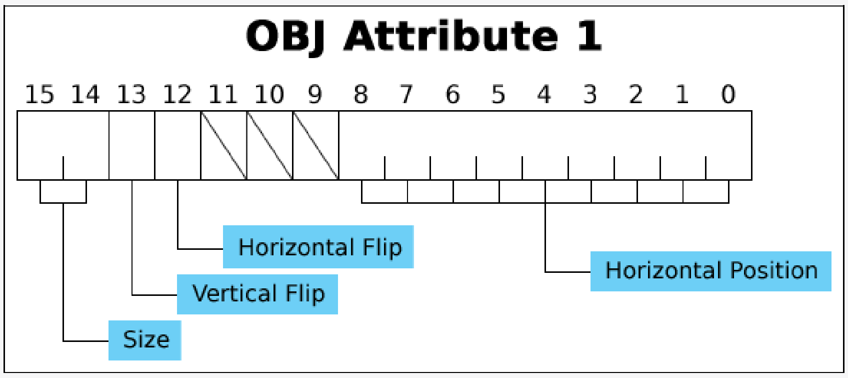
If this bit is 1, then the sprite will use 256 color tile data. We will use 0, for 16-color sprites.

Shape

This bit changes the shape of the rectangular drawing region for the sprite. 0 = Square, 1 = Wide, 2 = Tall, 3 = Do not use.

See the table for the *Size* attribute below.

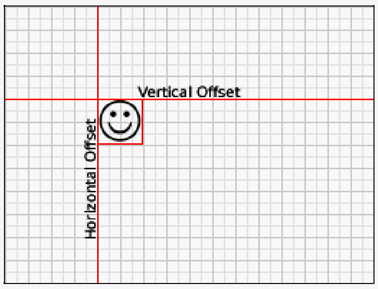
Attribute 1



This attribute has two different modes, depending on whether the Rot/Scale enable flag was set in attribute0.

Horizontal Position

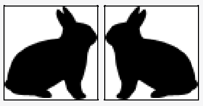
This is the X position of the top left corner of the sprite rectangle from the left side of the screen (measured in pixels).



This number can be negative too, mask negative values with 0x1FF before writing it to the sprite attribute!

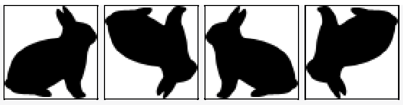
Horizontal Flip

Setting this bit will make the sprite image flipped horizontally. Like this: (flipped one is on the right)



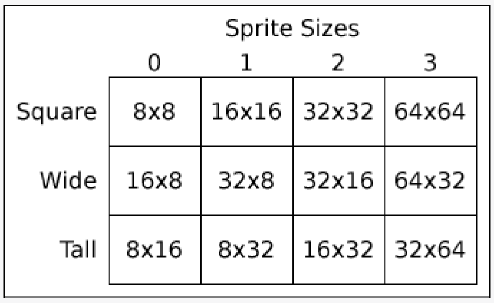
Vertical Flip

Setting this bit will make the sprite image flipped vertically. You can also mix the two flipping bits together.



Size

This setting controls the size of the drawing rectangle and how many tiles will be used for rendering the sprite.

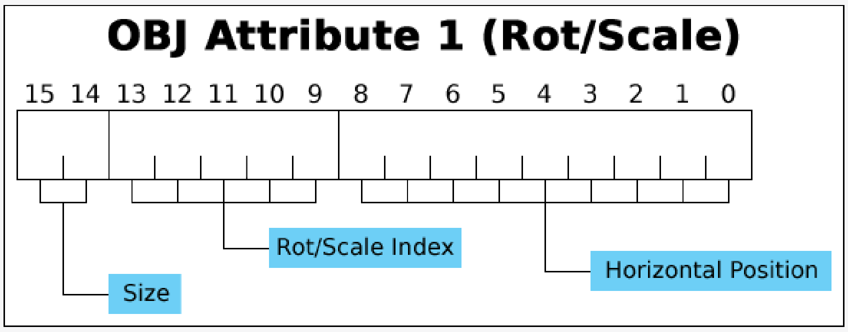


Sprite Sizes

	0	1	2	3
Square	8x8	16x16	32x32	64x64
Wide	16x8	32x8	32x16	64x32
Tall	8x16	8x32	16x32	32x64

Each setting will draw (width/8) * (height/8) tiles.

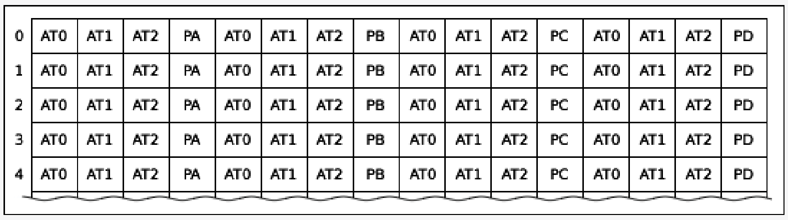
Attribute1 (Rot/Scale Enabled)



In this mode, there are no horizontal/vertical flipping flags, and we have a new setting.

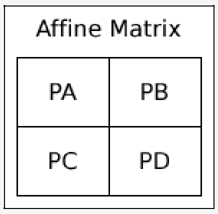
Rot/Scale Index

This value selects one of the rot/scale parameters for transforming the pixels (0..31). The parameter data is mixed into the OAM memory, it uses the 4th unused Hword of each sprite entry.



0	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD
1	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD
2	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD
3	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD
4	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD

In the diagram, we see a picture of the OAM memory with each cell representing 2 bytes (1 Hword). AT0, AT1, and AT2 are the three sprite attributes for each sprite entry. The last Hword of each entry form the *affine matrices*, each matrix has four values: PA, PB, PC, and PD. These values may also be called dx, dmx, dy, and dmy. Personally, I like to call them XX, XY, YX, and YY. They form a 2x2 matrix that is used to transform the pixels of a sprite.

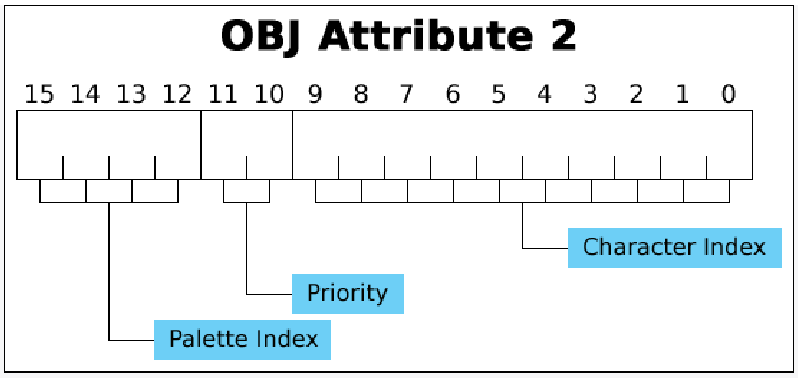


Affine Matrix

PA	PB
PC	PD

More info on this in a later chapter.

Attribute 2



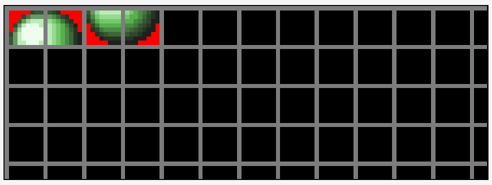
Character Index

This is the number of the first tile that the sprite will draw.

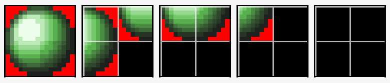
Which tiles are used depends on the layout of the tile vram (layout is selected in the video mode).

1D Layout

This is the mode we are using, here is a closup of our OBJ VRAM at the beginning.



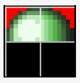
Now, in 1D mode, the tiles are chosen linearly, with the tiles placed in the sprite rectangle in the order left->right, top->bottom. Look what images the sprite will draw if we choose tiles 0->4:



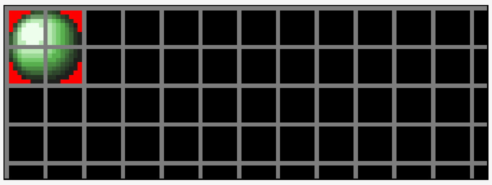
I hope you understand what's going on there.. :)

2D Layout

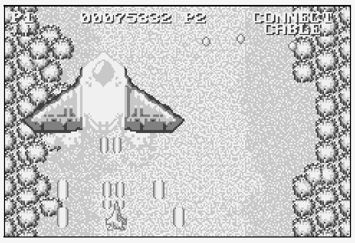
In this mode, the tiles are arranged in a way so the VRAM block looks *normal*. This main disadvantage of this is that the data isn't linear, so you have to piece your tiles together like a puzzle in VRAM :P. This will be displayed if we try to draw our current sprite in 2D mode.



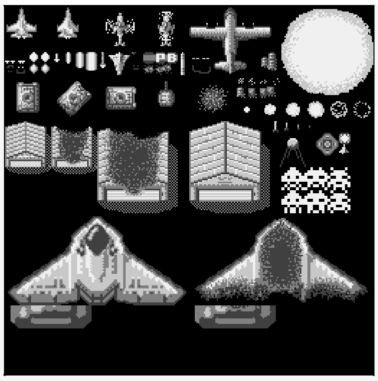
It'll look normal if we move the lower half of the sprite into the next row (and for larger sprites, move each row down to the next):



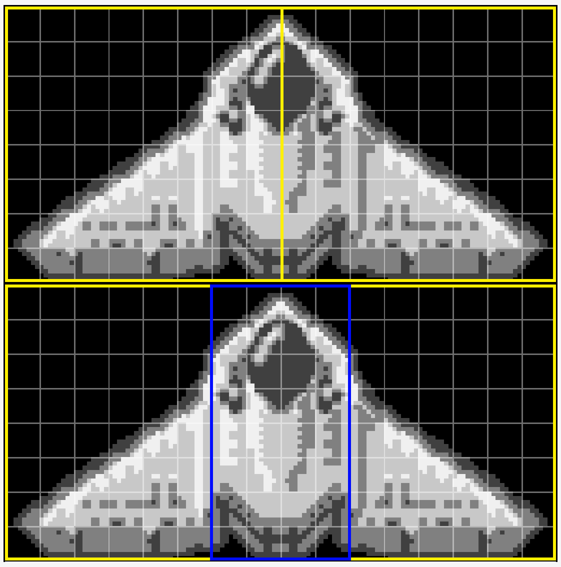
One advantage of 2D mode, is that you can use parts of 2 or more different images loaded in vram to make some sprites (you can't do this if both images are stored in linear format). Have a closer look at what happens when I shoot the center of the second boss in *Super Wings*.



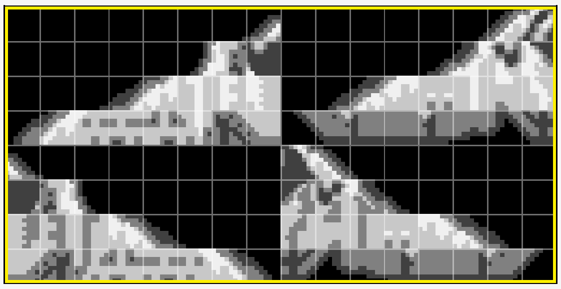
And here is a snapshot of all the tile data, everything is clearly visible because of the 2D layout.



Now, although the 128x64 ship could be rendered with 2 64x64 sprites, I additionally pasted another 32x64 sprite on top of it. This is the sprite that flashes when you shoot the center.



In 1D mode, the ship would have to be arranged like something like this:



Can't really get a center piece from that. :P (would have to load/generate separate tile data).

Bitmap layout

For bitmap sprites, the layout is a bit screwy. ;)

I don't want to clutter up this tutorial with bitmap stuff, so *maybe* I'll explain it later in another tutorial.

Priority

Here is the priority setting, it's just like the one for BG. Sprites have a higher base priority than BGs. If a sprite has the same priority number as a BG, then the sprite will be rendered on top of the BG.

Palette

This is the index of the 16-color palette to use to render the 16-color tile data. In 256 color mode, this value isn't used.

For direct color bitmap sprites, this entry is used as a per-sprite *alpha* value. It is used together with the alpha blending effects controlled by BLDCNT.

Displaying Some Sprites

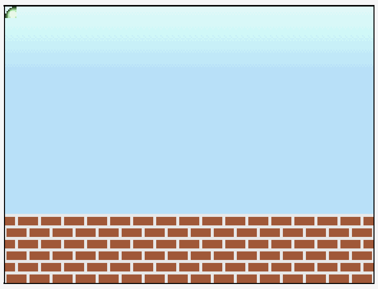
We'll want to modify the OAM entries easily, add a definition to make a reference of the OAM casted to a struct containing the attributes.

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

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

Okay. Go to your setupGraphics function again. Let's start by enabling sprites (and 1D layout)!! Change your video mode.

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

Compile and run the game, you should see this:

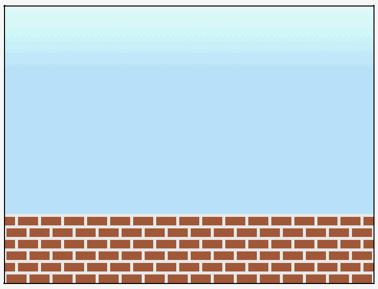


Notice the little crap in the top-left corner. That is a sprite! Actually, it's *128 uninitialized sprites stacked on top of each other*. ;)

To fix this, let's set all the sprite entries to *disabled* before setting the video mode.

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

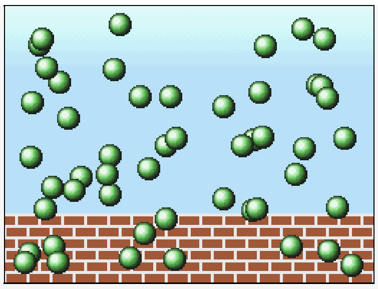
ATTR0_DISABLED is an alias for the double size flag (used as disable bit when theres no rotation/scaling). Compile and run the game. The sprites in the corner should be gone now.



Let's add some of our own sprites! We'll add balls all over the screen! [that sounded a little wrong...]

```
    // code to test out the sprite engine
    int n;
    for( n = 0; n < 50; n++ )
    {
        // attribute0: set vertical position 0->screen_height-sprite_height,
        // other default options will be okay (default == zeroed)
        sprites[n].attr0 = rand() % (192 - 16);

        // attribute1: set horizontal position 0->screen_width-sprite_width
        // also set 16x16 size mode
        sprites[n].attr1 = (rand() % (256 - 16)) + ATTR1_SIZE_16;

        // attribute0: select tile number and palette number
        sprites[n].attr2 = tiles_ball + (pal_ball << 12);
    }
```

Compile and run...etc. You should see the amazing image below:



Whee! :D

Previous: Loading sprite graphics | Contents | Next: Fixed point arithmetic

Fixed Point Arithmetic

The Problem

Say we want to scroll the background left a little bit every frame. Our program loop is operating at ~60hz so adding 1 pixel at a time would scroll it 60 pixels/second, that *might* be too fast! The solution is to add a *fraction* of a pixel every frame, like 0.25 instead of 1.

Now, the obvious easy method to do this with *PC* programming is to use floating point numbers. Floating point arithmetic may take advantage of an FPU (floating point unit) to calculate special numbers with a fractional part.

The problem with floating point on the DS is that the DS does not have an FPU. Floating point operations are emulated by the CPU. This makes them very *very slow*, and it eats up your processing time.

The Solution

Here is a great alternative to floating point numbers: Fixed point numbers!! :D

Fixed point numbers are in a special numbering format where the fractional part and the whole 'integer' part are separated. An example... hmm, look at your clock (I hope it's digital).

When the minute reaches *60* then the hour will increment, this is like a fixed point format, with the fractional range being 0->60.

Now, to represent the clock value (hours and minutes only) in a single value, the value can be 13*60 + 37. Anything we add to this value will be in the scale of 'minutes'.

But, now we have a problem, the clock value is mixed together... To retreive the value, we can do:

$$\mathit{hours} = \mathit{INTEGER}(\mathit{value} / 60)$$
$$\mathit{minutes} = \mathit{value} \,\%\, 60$$

This causes a big problem when doing stuff on handheld consoles that aren't very powerful (it's also a problem when programming for any computer). The above formulas require a divide operation to calculate the value, divides are usually quite slow, no matter what computer you are programming for.

To fix this problem, we can be smart and use a power of 2 as the fractional range (you can stop looking at the clock now). If the fractional range is a power of 2, then we can use bit-shifts/masking instead of division to compute the two values!

For example, I have this number here: 1502. I wonder what *real* value it represents?? Well, it depends on what format the number is in. If we are using fixed point with a fractional range of 2^8, then the value will be:

$$\mathit{whole} = \mathit{INTEGER}(1502 / 2^8) = 1502 >> 8 = 5$$
$$\mathit{fraction} = 1502 \,\%\, 2^8 = 1502 \,\&\, 255 = 222$$
$$\mathit{VALUE} = 1502 / 256 = 5.8671875\ (!)$$

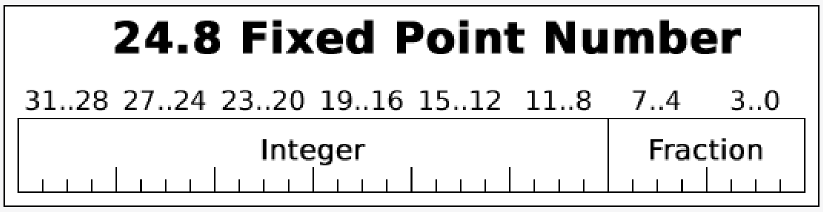
These calculations can be done *very* fast by computers. Bit shifts can be done in a single instruction (1 clock cycle, you get over 60 million cycles/second).

From here onward, when I refer to fixed point numbers, I am only referring to those which have fractional ranges in powers of 2.

Different Formats

There's probably many ways people represent the integer and fractional size for a fixed point number. The one I see often is the I.F format, where I is the number of *bits* that represent the integer portion, F is the number of bits that represent the fractional portion, and I+F is the size of the *container*. If a number has 5 bits representing integer part, and 5 bits representing fractional part, then it would be called "5.5". Sometimes a little prefix is added too, like "f5.5" or "fx5.5". There isn't really a standard notation.

A commonly used format is 24.8, where the fraction can be incremented 256 times before the integer part is incremented. The first *byte* (8 bits) of data hold the fractional part. The other 3 bytes (24 bits) contain the integer value.



The 8 bits of fraction is usually enough data for most operations. People sometimes increase the number of bits for the fractional part when accuracy is an issue. (And they also have to be cautious when decreasing the integer size).

In a perfect world, we will probably prefer the fixed point format with an infinite integer size (and a super super huge fractional size). :)

Converting Between Fixed Point and Decimal

This is pretty easy to understand. To convert a decimal number to a fixed point value, multiply the decimal number by 2^f where *f* is the number of bits that represent the fractional part in the fixed point value. After converting you must throw away the fraction; round to the nearest integer. So that's $x = ROUND(2d^f)$.

To convert the opposite way, simply divide instead of multiplying, or negate the exponent. $d = x2^{-f}$

Accuracy

Now, after converting to fixed point, the number must discard the value after the decimal point. This introduces a little bit of an innacuracy problem. Most of the time an 8 bit fraction gives good enough accuracy, but sometimes more is needed.

For an example, let's convert 1.11 to fixed point using 8 bits fraction.

$$284.16 = 1.11 \times 2^8$$

We have to throw away the part after the decimal point so it's an integer value; we round to *284*.

This is the actual value we get, BUT, it is slightly off from its source. When we convert back to decimal, we get:

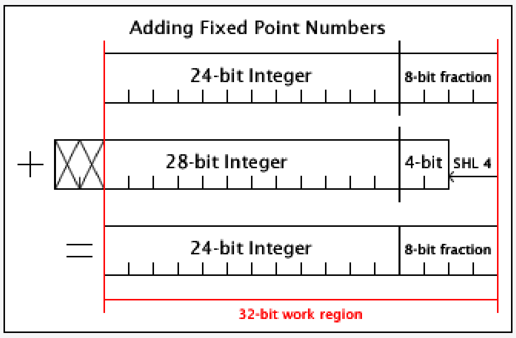
$$1.109375 = 284 \times 2^{-8}$$

The error is *0.000625*. That's a pretty small error, but it may [will] add up if you accumulate the results. Keep accuracy in mind when you program some advanced fixed point stuff!

Addition and Subtraction

Let's start with the basics. You can add and subtract fixed point numbers just like any regular integer. BUT, if the two numbers are in different *formats* you must convert one format to the other before adding/subtracting.

If you go back to fourth grade, you might remember learning how to align the decimal points before you add something. We can use this concept here.



In this example, we are adding a 24.8 fixed point value to a 28.4 one. We must align the fractional parts before adding the two numbers (like aligning the decimal points). We shift the second value left 4 spaces to match the fraction of the other addend. Since we are using 32-bit operations, shifting the 28.4 value overflows past the 32-bit work area. If those top 4 bits of the number have data in them, the data will be lost and there will be an error in the calculation. This limits the integer size of the second number to only 24 bits. The result is a 24.8 fixed point value.

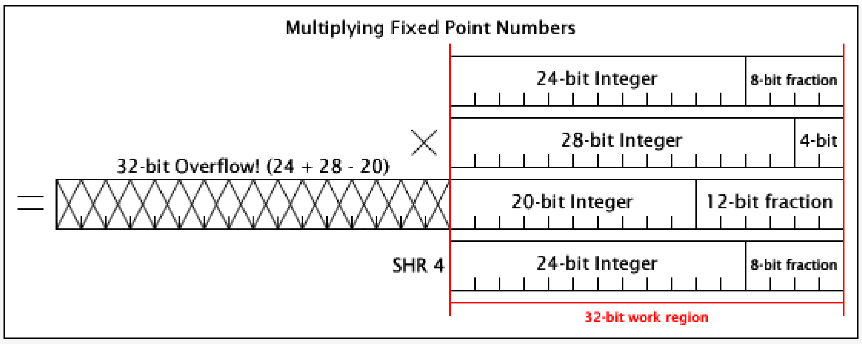
To avoid the overflowing problem, and have less accuracy instead, we can shift the first value right 4 bits to align, instead of the second value shifted left 4 bits. This would affect the fractional size of the result too, giving us a 28.4 number.

You can also shift *both* of the values to match the fractional parts, you need to decide which method is neccesary for the operation in question.

Subtraction is exactly the same as addition, except replace PLUS with MINUS. :P

Multiplication

Now it gets a little more complicated. When we multiply two fixed point numbers the fractional and integer sizes get *added* to each other, we must ensure that the product does not overflow past the 32-bit working area.



In this example, we multiply a 24.8 number with a 28.4 number. The result we get is a 52.12 number, *but*, since the number exceeds the 32-bit region, the top 32 bits cannot be used; the integer part must be truncated to 20 bits. This limits the value of the result significantly. If there is data in those top 32 bits, then it will be lost, and the result will be invalid. There isn't usually a need to multiply two huge numbers by each other.

After the multiplication, the example shifts the truncated result right 4 bits to restore 24.8 format.

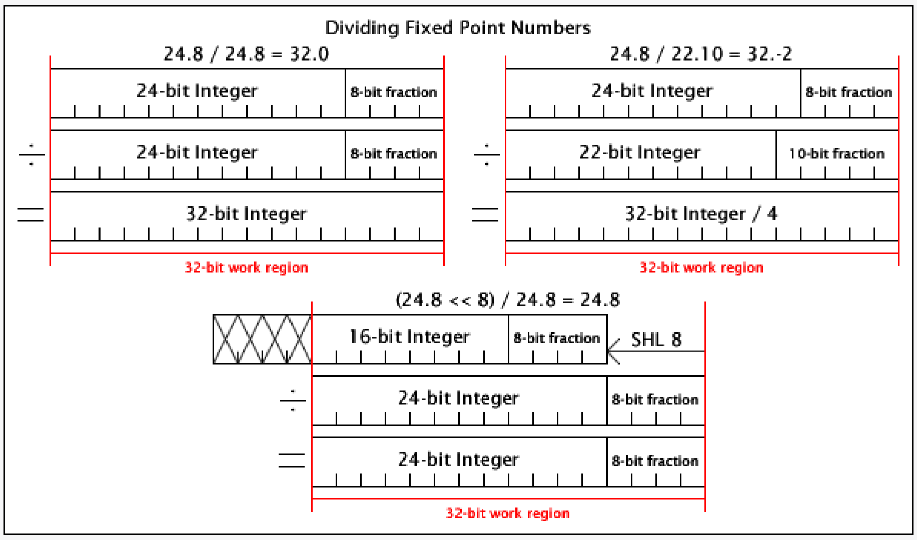
Now, the shifting can be done before or after multiplication, or maybe before AND after, or even... not at all. It all depends on how much accuracy you need, how big the numbers will be (usually the 24 bit integer isn't fully used), and/or what output format you want. If you shift the value before multiplication, you will lose data in the fractional part, causing some error in the result. If you shift afterwards, the product will be somewhat limited due to the 32-bit work boundaries. If you shift before and after, then you'll get a fraction of both bad stuff.

Sometimes you don't want to shift at all, or you want to shift to another amount. Say you need a *.8 fixed point number from 2 factors. You have a *.0 whole number, and another *.8 fixed point number. You multiply the *.0 by *.8; you end up with a *.8 in the result. The result doesn't need any shifting because this is already the desired format.

If you really need the accuracy *and* big numbers multiplied, you can use 64-bit operations to double the work boundaries. 64-bit operations may be a little bit slower, but sometimes they are needed.

Division

Division is a bit interesting. After you divide two fixed point numbers, the fractional size of the dividend gets subtracted by the divisor's fractional size.



Now, when you divide 2 numbers with the same format. You will lose *all* of the fractional part (top-left example). This usually isn't desired! This inaccuracy is prevented by shifting the dividend left by some amount (usually equal to the divisors fractional size). This is shown in the bottom example.

Another interesting effect is when you divide by a number with a greater fractional size. The result will be the quotient divided by some amount. The value needs to be shifted left after the division to get the real integer number.

Heh

Anyway, you'll get the hang of it some day. I hope you got a little grasp of it, we'll be applying a bunch of it in the next chapter.

Previous: Sprites | Contents | Next: The Bouncy Ball

The Bouncy Ball

Ah, finally time to program this.

Start by making a new source file: *ball.c*. Add it to your *source* directory.

Also, make a new directory in your project, call it *include*. The default makefile will add this directory to the list of folders containing files to be #include(d) in the source. Make a new file called *ball.h* and add it to the include directory.

The Ball Object

We need to make a struct to define what variables the ball needs. Open up ball.h.

Now, just to be safe, you should always wrap the code in your header files to make sure that it's only included once in any source file. If the header file is included multiple times in a singe source file, errors will probably result.

Like this:

```
// ball.h

#ifndef BALL_H
#define BALL_H

// insert code

#endif
```

Since BALL_H was defined inside the #ifndef (this preprocessor code is to include code ONLY if the condition is false), if the #ifndef happens again (if the source file includes the header twice somehow), then it will fail and protect your code from breaking.

Now, when I say "// insert code" I do NOT mean actual source code, source code belongs in source files only. Headers should only contain references to the source code and other definitions like structures. It may contain static source code though (usually inline, for small helper functions, etc).

Let's make a structure to hold the variables for the ball object.

```
#ifndef BALL_H
#define BALL_H

typedef struct t_ball
{
    int x;     // 24.8 fixed point
    int y;     // 24.8 fixed point
    int xvel;  // 20.12 fixed point
    int yvel;  // 24.8 fixed point

    u8  sprite_index;        // OAM entry (0->127)
    u8  sprite_affine_index; // OAM affine entry (0->31)

    int height; // height of ball
} ball;

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

#endif
```

This struct contains all the elements for our ball object. 'x' holds the horizontal position, measured in 24.8 fixed point, or 256 units per pixel. 'y' is the vertical position, in the same format.

xvel is the horizontal *velocity*. This will be added to the horizontal position every frame. *yvel* is the same, but for the vertical position. yvel is in the same 24.8 format, but xvel is in a higher precision 20.12 format.

sprite_index is the entry in the OAM/sprite this object will use for rendering itself. There are 128 entries, so this value ranges from 0->127.

sprite_affine selects an entry in the OAM affine data. There are 32 entries, so this value ranges from 0->31. This won't be used until next chapter.

height is the current height of the ball. This will be used in the 'squishing' chapter. :P

We also have two other definitions here, two references to the update functions [written in section below]. These functions will be called by main.c.

Programming the Ball

Open up *ball.c* now. Before we start writing code, we want to include 2 files. The libnds definitions, and our new ball reference.

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

We will have 2 functions, one for updating the ball physics, and one for rendering the ball. The update one should be called once per frame in the 'logic' section of the main loop. And the other should be called during VBlank.

```
void ballUpdate( ball* b )
{

}

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

}
```

Each function takes a pointer to the ball object to be updated/rendered, we can make multiple ball objects like this! ballRender takes two other parameters specifying the coordinates of the *camera* ...in pixels.

We will start with ballRender.

Rendering the Ball

In our render function, we want to convert the position of the ball to pixels, and transform it with the camera position. With the new coordinates, we will modify an entry of the OAM to render the sprite.

Let's make a pointer to the sprite OAM entry.

```
void ballRender( ball* b, int camera_x, int camera_y )
{
    u16* sprite = OAM + b->sprite_index * 4;
    ...
```

Each sprite entry is 4 Hwords wide. This gets the base address of the sprite entry to use.

Next, we'll translate the X/Y coordinates to sprite coordinates.

```
    ...

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

    ...
```

c_radius should be defined somewhere at the top of the source code, it holds the radius of the ball, which is 8. Since the coordinates are in 24.8 fixed point, radius needs to be in that format too.

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

The coordinates are then shifted into an integer value, and the camera offsets are subtracted.

Now, before we set the sprite entry, let's check if our values are actually in the screen boundaries. Although you can specify coordinats that are off the screen, if the coordiantes stretch too far our there, they will wrap around to the other side of the screen, not very desired!

```
    ...

    if( x <= -16 || y <= -16 || x >= 256 || y >= 192 )
    {
        // sprite is out of bounds
        // disable the sprite
        sprite[0] = ATTR0_DISABLED;
        return;
    }

    ...
```

With the above code, if the sprite position is out of bounds, then it will *disable* the sprite target, and the function will exit.

We can set the sprite attributes now. We remember that attribute0 holds the Y position. We must also mask the value to 8-bits to truncate any sign-extension.

```
    sprite[0] = y & 255;
```

The X coordinate is stored in attribute1, along with the *size* setting. We want to draw a 16x16 image. Don't forget to mask the X value to cut off the sign-extension.

```
    sprite[1] = (x & 511) | ATTR1_SIZE_16;
```

The last attribute controls the tile entry, priority, and palette.

Now, although it's bad practice, I am feeling really really lazy. I'm just going to *hardcode* the tile and palette number to 0. (priority can be 0 too).

```
    sprite[2] = 0;
```

The problem with this is that if you ever load the tiles somewhere else, you'll have to change this value too. You should always #define stuff that is to be used more than one time in the source code. It saves alot of headache when tweaking things.

This is enough code to produce some results. Let's go back to main.c.

Cleaning Time

Our main.c function is getting a little clutterd. Let's reorganize some of it. Start by replacing your main function with this.

```
//-------------------------------------------------------------
// program entry point
//-------------------------------------------------------------
int main( void )
//-------------------------------------------------------------
{
    // setup things
    setupInterrupts();
    setupGraphics();
    resetBall();

    // start main loop
    while( 1 )
    {
        // update game logic
        processLogic();

        // wait for new frame
        swiWaitForVBlank();

        // update graphics
        updateGraphics();
    }
    return 0;
}
```

In the beginnin of the function, we setup things.

setupInterrupts is our new function to setup the interrupt handler and enable interrupts. We will move the irqInit/irqDisable combo in there.

```
//-------------------------------------------------------------
// setup interrupt handler with vblank irq enabled
//-------------------------------------------------------------
void setupInterrupts( void )
//-------------------------------------------------------------
{
    // initialize interrupt handler
    irqInit();

    // enable vblank interrupt (required for swiWaitForVBlank!)
    irqEnable( IRQ_VBLANK );
}
```

setupGraphics is the same function that we wrote in previous chapters. Oh, don't forget to remove the 'testing' code in setupGraphics. :P (the part that adds balls all over the screen)

resetBall is a new function to initialize our ball object. We will only have one ball object, so define a global variable to hold the ball attributes. Remember that the attributes [structure] are defined in ball.h, include that in main.c too.

```
#include "ball.h"

ball g_ball;
```

In resetBall, we will initialize our new object.

```
//-------------------------------------------------------------
// reset ball attributes
//-------------------------------------------------------------
void resetBall( void )
//-------------------------------------------------------------
{
    // use sprite index 0 (0->127)
    g_ball.sprite_index = 0;

    // use affine matrix 0 (0->31)
    g_ball.sprite_affine_index = 0;

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

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

    // start X velocity a bit to the right
    g_ball.xvel = 100 << 4;

    // reset Y velocity
    g_ball.yvel = 0;
}
```

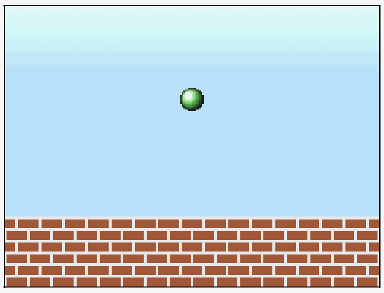
Now we have two more functions, *processLogic* and *updateGraphics*. The *updateGraphics* one will update graphical elements on the screen, so it *must* be called during the VBlank period to ensure a smooth display. *updateLogic* is where keypad input, ball physics, and other non graphic-related routines will go.

Right now, since we haven't programmed the ball physics yet, we'll just write the updateGraphics function. Just make processLogic an empty function to be filled in later.

```
//-------------------------------------------------------------
// update graphical information (call during vblank)
//-------------------------------------------------------------
void updateGraphics( void )
//-------------------------------------------------------------
{
    // update ball sprite, camera = 0, 0
    ballRender( &g_ball, 0, 0 );
}
```

The project can be compiled now, to see our premature output. :D

You should see the ball on the screen (where resetBall put it). We need to program the behavior now.



Programming the Ball (really)

Before doing anything, open up main.c and add a ballUpdate call in processLogic. This is the update function that must be called every frame to update behavior.

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

Open up ball.c again, and this time we'll write the contents of ballUpdate. This is the kind of fun part.

Start by writing a line to add the X velocity to the balls X position.

```
//-------------------------------------------------------------------
// update ball object (call once per frame)
//-------------------------------------------------------------------
void ballUpdate( ball* b )
//-------------------------------------------------------------------
{
    // add X velocity to X position
    b->x += (b->xvel>>4);
}
```

Notice the >>4 performed on xvel. This is because xvel is 20.12 fixed point format, while *x* is 24.8 format. xvel must be aligned to *x*'s format before addition.

Compile and run the game, you should see the ball moving to the right. :D

Before we continue, let's define some constant values for the physics. I hand-tweaked these values until the ball's behavior felt somewhat right. :)

I also added a small function to *clamp* a value.

```
#define c_gravity           80                // gravity constant (add to vertical velocity) (*.8 fixed)

#define c_air_friction      1                 // friction in the air... multiply X velocity by (256-f)/256
#define c_ground_friction   30                // friction when the ball hits the ground, multiply X by (256-f)/256
#define c_platform_level    ((192-48) << 8)   // the level of the brick platform in *.8 fixed point
#define c_bounce_damper     20                // the amount of Y velocity that is absorbed when you hit the ground

#define c_radius            (8<<8)            // the radius of the ball in *.8 fixed point
#define c_diam              16                // the diameter of the ball (integer)

#define min_height          (1200)            // the minimum height of the ball (when it gets squished) (*.8)

#define min_yvel            (1200)            // the minimum Y velocity (*.8)
#define max_xvel            (1000<<4)         // the maximum X velocity (*.12)

//------------------------------------------------------------------
// clamp integer to range
//------------------------------------------------------------------
static inline int clampint( int value, int low, int high )
//------------------------------------------------------------------
{
    if( value < low ) value = low;
    if( value > high) value = high;
    return value;
}
```

Let's apply the "air friction" to the X velocity. Also, let's clamp it to the maximum velocity setting.

```
    // apply air friction to X velocity
    b->xvel = (b->xvel * (256-c_air_friction)) >> 8;

    // clamp X velocity to the limits
    b->xvel = clampint( b->xvel, -max_xvel, max_xvel );
```

Compile and run. The ball should slowly stop in the air now.

Next we'll add the *gravity* constant to the Y velocity, and add the Y velocity to the Y position.

```
    // add gravity to Y velocity
    b->yvel += c_gravity;

    // add Y velocity to Y position
    b->y += (b->yvel);
```

Run the code, the ball should drop off the screen now! We'll make it bounce on the platform.

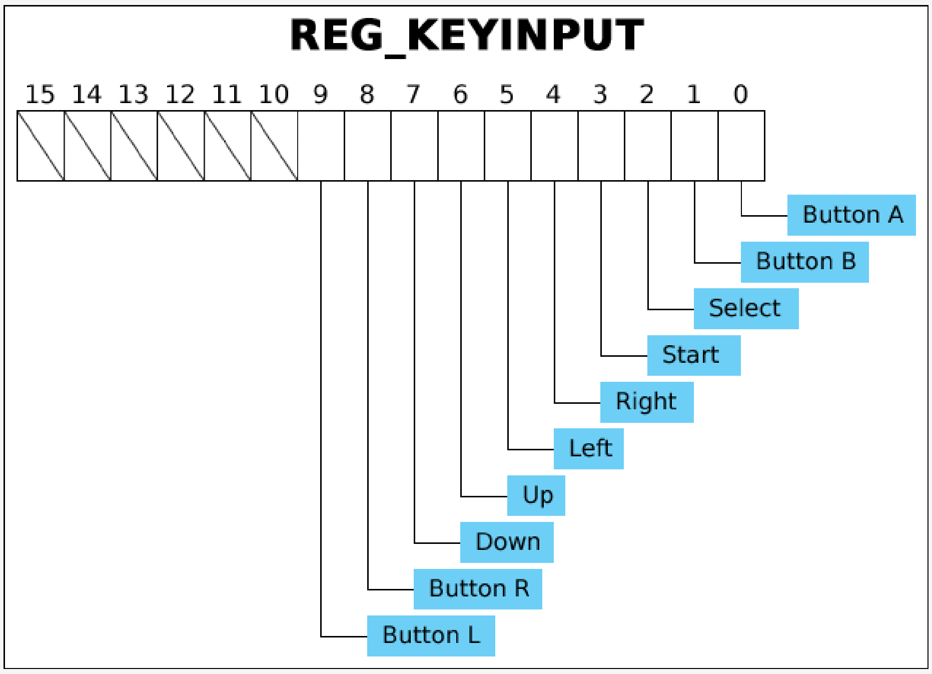
```
    if( b->y + c_radius >= c_platform_level )
    {
        // apply ground friction to X velocity
        b->xvel = (b->xvel * (256-c_ground_friction)) >> 8;

        // mount Y on platform
        b->y = c_platform_level - c_radius;

        // negate Y velocity, also apply the bounce damper
        b->yvel = -(b->yvel * (256-c_bounce_damper)) >> 8;

        // clamp Y to mininum velocity (minimum after bouncing, so the ball does not settle)
        if( b->yvel > -min_yvel )
            b->yvel = -min_yvel;
    }
```

First the code checks if Y + radius is greater than the brick platform level. If it is, it multiplies the X velocity by the ground friction, mounts the Y position on top of the platform, and reverses the Y velocity after multiplying by the damper.

Compile&run, the ball should be bouncing on the platform!! :D

Input

This is pretty boring if we don't get to interact with the ball. Let's program some input routines to modify the ball's velocity with the directional pad. For GBA/DS reading the keypad is fairly easy. Most of it is done with the KEYINPUT register (this was the only register for GBA).



If the bit corresponding to the certain key is *zero* then the key is *pressed* (yes, zero). If the bit is 1 then the key is released. I can't express how easy using this register is compared to the methods used before GBA/DS were available. For SNES, you had to read the register at a certain number of clock cycles after the VBlank starts, I never could get the input right on that thing.

Anyway, if you haven't noticed, two of the keys are missing (3 actually). The X/Y buttons and the 'pen' button. These values are read on the ARM7 side.

libnds has some functions that can read the keypad input for us. The first one we'll use is scanKeys().

```
void processInput( void )
{
    scanKeys();
    ...
```

The scanKeys function reads the KEYINPUT register into some internal variables. It also uses the previously read data to check if the key was just pressed, or just released.

To check if a key was just pressed, we can use the keysDown() function. It returns a bitmask of what keys were just clicked.

keysUp() returns a bitmask of what keys were just released.

keysHeld() returns a bitmask of what keys are held.

One more advantage of using the libnds keypad functions is that they retrieve the X, Y, and the PEN button from the ARM7 side too (a bit of a hassle saved).

We'll just use the keysHeld function for the keypad input. We'll make the directional keys modify the horizontal/vertical velocities of the ball.

```
    int keysh = keysHeld();
    // process user input
    if( keysh & KEY_UP )      // check if UP is pressed
    {
        // tweak y velocity of ball
        g_ball.yvel -= y_tweak;
    }
    if( keysh & KEY_DOWN )    // check if DOWN is pressed
    {
        // tweak y velocity of ball
        g_ball.yvel += y_tweak;
    }
    if( keysh & KEY_LEFT )    // check if LEFT is pressed
    {
        // tweak x velocity
        g_ball.xvel -= x_tweak;
    }
    if( keysh & KEY_RIGHT )   // check if RIGHT is pressed
    {
        // tweak y velocity
        g_ball.xvel += x_tweak;
    }
```

The KEY_??? symbols are libnds definitions to mask the keypad bitmask with.

Also, the x_tweak, and y_tweak values must be defined somewhere, it's how much will be added to the velocity when you press a key.

```
#define x_tweak    (2<<8)  // for user input
#define y_tweak    25      // for user input
```

Finally, add processInput() to your *processLogic* routine.

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

Compile and run the game, you should be able to control the ball's movement with the D-Pad now!! :D

The Camera

"Whoops, the ball went out of the screen."

To fix this, we will add a 'camera' that will follow the ball around the screen. Make two more global variables.

```
int g_camera_x;
int g_camera_y;
```

These are the X and Y position of our *camera*. They will be in 24.8 fixed point format. We will need a function to update the camera.

```
void updateCamera( void )
{
    // cx = desired camera X
    int cx = ((g_ball.x)) - (128 << 8);

    // dx = difference between desired and current position
    int dx;
    dx = cx - g_camera_x;

    // 10 is the minimum threshold
    if( dx > 10 || dx < -10 )
        dx = (dx * 50) >> 10; // scale the value by some amount

    // add the value to the camera X position
    g_camera_x += dx;

    // camera Y is always 0
    g_camera_y  = 0;
}
```

First this function calculates the position that the horizontal position of the camera *desires*, which is the ball in the center of the screen (ball - screen_width/2).

Second, this function gets the difference between the camera's X position and the desired position.

Third, we check if the distance is greater than a small threshold. This probably isn't needed, but it will ensure the camera stops exactly at the right position, instead of a *little* [unnoticably little :P] bit to the left/right.

Next we add the scaled difference to the camera's X position. This makes the camera trail a little bit behind the ball's movement.

The Y position of the camera will be set to zero.

Add updateCamera to your *logic* functions.

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

Next, we'll modify updateGraphics to use the camera position.

```
//-----------------------------------------------------------
// update graphical information (call during vblank)
//-----------------------------------------------------------
void updateGraphics( void )
```

```
//------------------------------------------------------------
{
    // update ball sprite
    ballRender( &g_ball, g_camera_x >> 8, g_camera_y >> 8 );

    REG_BG0HOFS = g_camera_x >> 8;
}
```

We pass the camera position (the integer part/pixels) to ballRender. There's also something else that's needed, we need to modify the horizontal offset of the *bricks* BG, so we set the HOFS register with the camera position.

Compile and run the code... the *camera* should follow the ball now.

The next chapter will describe how to make the ball squishy. :D

Previous: Fixed point arithmetic | Contents | Next: Squishing stuff

Transforming Sprites

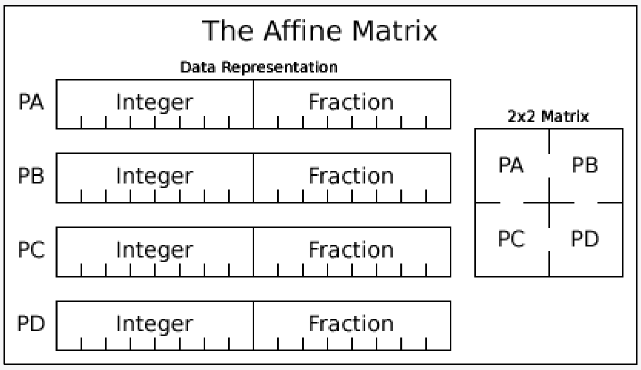
Now we are going to play with a fun hardware aspect of the sprites. The sprites (and sometimes backgrounds!) have a special feature that allows the pixel coordinates to be transformed by a 2x2 matrix (affine matrix).

The affine matrix can be used to rotate, scale, and/or even skew the sprite image.

We are going to take advantage of this hardware feature to *squish* the ball when it hits the ground.

The Affine Matrix

The affine matrix is composed of four values: PA, PB, PC, and PD. For DS (and GBA), The values are in 8.8 fixed point format.



The DS uses the affine matrix to transform pixel coordinates.

For each pixel of the sprite, the image coordinate is calculated with the formulas:

$$u = x * PA + y * PB + width / 2$$
$$v = x * PC + y * PD + height / 2$$

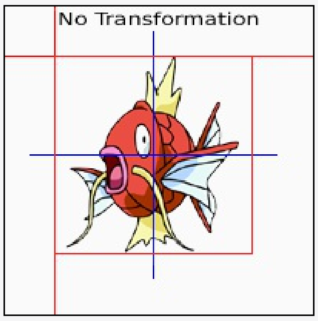
Where x and y are the distance from the *center* of the sprite. u and v select a pixel from the OBJ vram source.

There are a few basic types of matrices, explained below.

Identity

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The above diagram shows an *identity* matrix. If we use this matrix then the u and v coordinates will be equal to the x and y coordinates, thus not affecting the image output from the source.



Transformation by Identity Matrix (no change)

Rotation

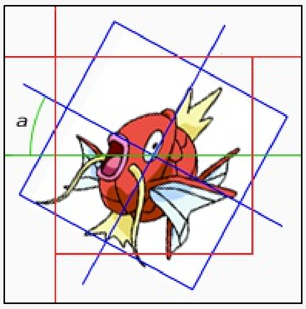
Another basic type of matrix is the *rotation* matrix.

$$\begin{bmatrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{bmatrix}$$

2x2 Rotation Matrix

In the above diagram, *a* represents the desired angle. For a practical application, cos and sin can be done with a LUT (look-up table/array containing a bunch of precalculated values) containing 8.8 fixed point values.

When using this kind of matrix, the image will appear rotated by the angle specified.

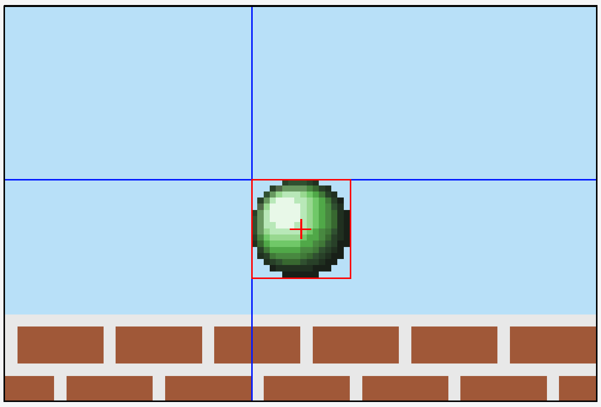


Notice the parts of the image that reach outside of the red boundary, this is a problem that needs to be checked when transforming sprites (explained along with scaling).

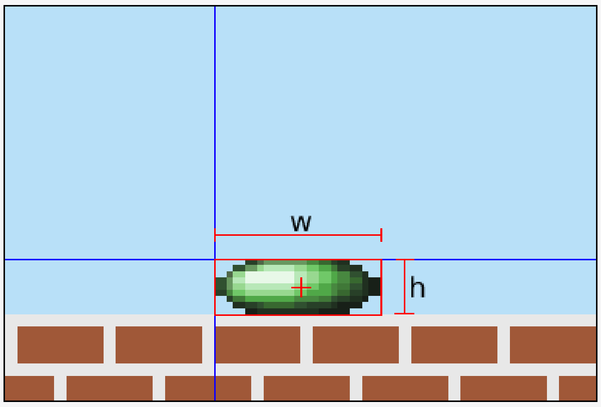
Scaling

$$\begin{bmatrix} x\_scale & 0 \\ 0 & y\_scale \end{bmatrix}$$

This is the one we're going to use, we want to scale the ball by some amount to make it look cooler when it bounces. x_scale/y_scale are the factors that will be multiplied by the x/y coordinates to get the scaled u/v coordinates.



This is the ball in the air...

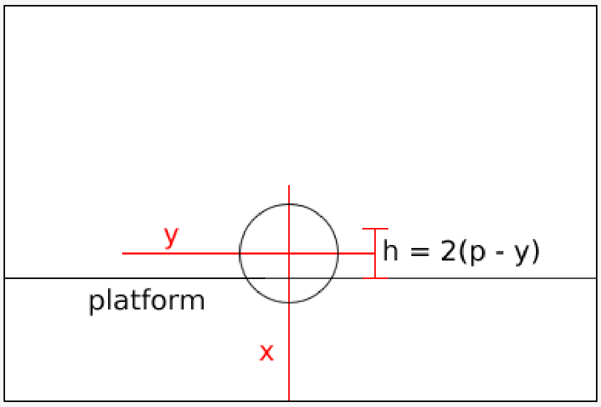


We will scale [squish] it when it touches the ground.

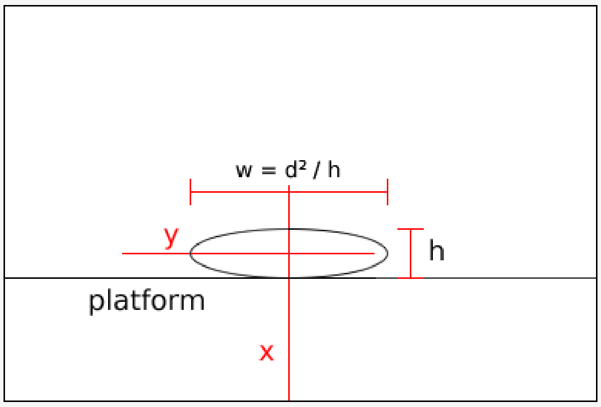
I know it makes the ball look a bit deformed, but the image will only be seen for a single frame.

Scaling the Ball

Now let's apply this stuff to our bouncy ball. The first thing we are going to do is...allow the ball to fall somewhat *past* the platform level.



We then measure the vertical distance between the platform and the top of the ball. This is our *height*. When the ball is touching the platform, we will scale the height of the ball to this difference, so the ball appears squished against the platform.



With a height value, we can also calculate a width value from it to preserve the area of the ball. We can do this with the formula:

width = diameter^2 / height

To translate this width value to a value for PA (x_scale) we do:

PA = diameter / width

For y_scale, it's:

PD = diameter / height

Let's start by editing ballUpdate to let the ball fall somewhat past the platform before it bounces. Here's the old code:

```
    if( b->y + c_radius >= c_platform_level )
    {
        // apply ground friction to X velocity
        b->xvel = (b->xvel * (256-c_ground_friction)) >> 8;

        // mount Y on platform
        b->y = c_platform_level - c_radius;

        // negate Y velocity, also apply the bounce damper
        b->yvel = -(b->yvel * (256-c_bounce_damper)) >> 8;

        // clamp Y to mininum velocity (minimum after bouncing, so the ball does not settle)
        if( b->yvel > -min_yvel )
            b->yvel = -min_yvel;
    }
```

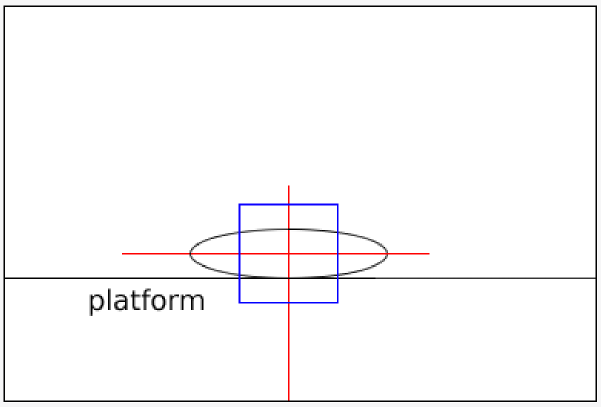
We will modify it to only bounce when the ball reaches the point where it's past minimum height (min_height, as defined earlier). We also calculate the height value, if the ball isn't touching the platform, we will specify the regular height (diameter). *height* is in *.8 fixed point.

```
    if( b->y + c_radius >= c_platform_level )
    {
        // apply ground friction to X velocity
        // (yes this may be done multiple times)
        b->xvel = (b->xvel * (256-c_ground_friction)) >> 8;

        // check if the ball has been squished to minimum height
        if( b->y > c_platform_level - min_height )
        {
            // mount Y on platform
            b->y = c_platform_level - min_height;

            // negate Y velocity, also apply the bounce damper
            b->yvel = -(b->yvel * (256-c_bounce_damper)) >> 8;

            // clamp Y to mininum velocity (minimum after bouncing, so the ball does not settle)
            if( b->yvel > -min_yvel )
                b->yvel = -min_yvel;
        }

        // calculate the height
        b->height = (c_platform_level - b->y) * 2;
    }
    else
    {
        b->height = c_diam << 8;
    }
```

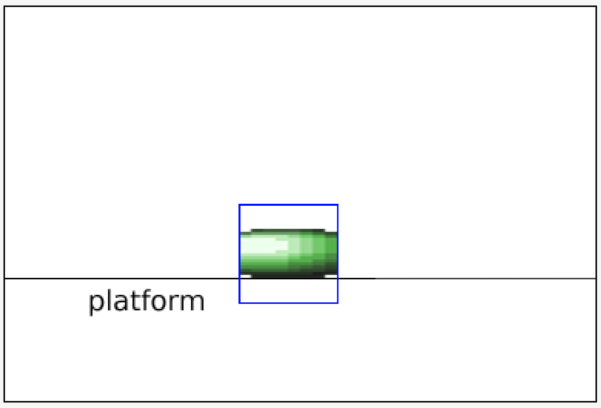
Now let's modify our ballRender code.

What we want to do is modify the routine to setup an affine matrix containing the scaling parameters from the height value. We must understand one more concept before we do this though...

Have a look at the picture below, it's the ball squished to 50% height, it also shows the sprite's rendering box! The rendering box's top left corner is the sprite's X and Y offsets.

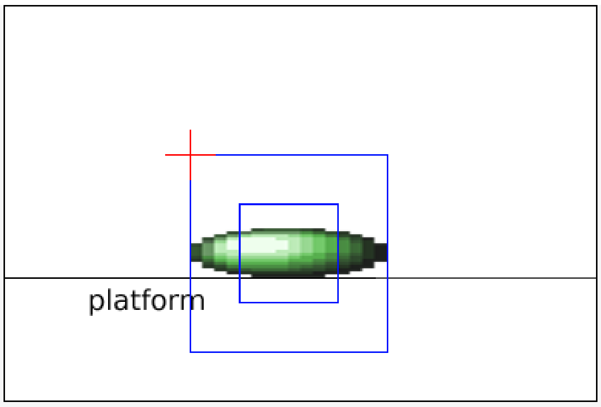


For our ball, the rendering region is 16x16. We have a problem here, when the ball gets squished, the width of the ball extends past the rendering box. The sprite will be rendered somewhat like this:



Thankfully, there is an easy solution to this -- the *double-size* flag (in attribute0).

The advantage of the double-size flag is that the sprite's rendering size will be doubled, giving you twice the space to render the object. The disadvantage is that the sprite will consume double the amount of rendering cycles.



Now we will be able to squish the ball to half it's height. Note that the X/Y coordinates of the sprite will change. (subtract width/2!)

Let's write the code finally. First thing we want to change is the coordinates, we must subtract c_radius * 2 instead of * 1 to compensate for the double-sized sprite.

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

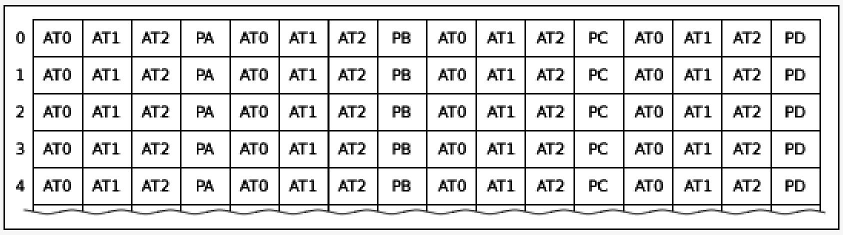
Now enable rotation/scaling with the double-size flag in attribute0.

```
    sprite[0] = (y & 255) | ATTR0_ROTSCALE_DOUBLE;
```

In rot/scale mode, attribute1 changes a bit, there are no more X/Y flip flags. Instead, there is a 5 bit affine matrix selection. Select the affine matrix specified in the ball structure.

```
    sprite[1] = (x & 511) | ATTR1_SIZE_16 | ATTR1_ROTDATA( b->sprite_affine_index );
```

Now we can setup the affine matrix values. As mentioned before, the data is interleaved into the sprite parameters with every 4th Hword.



0	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD
1	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD
2	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD
3	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD
4	AT0	AT1	AT2	PA	AT0	AT1	AT2	PB	AT0	AT1	AT2	PC	AT0	AT1	AT2	PD

Make a pointer to the affine matrix we want to modify.

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

We multiply the affine selection by 16 (Hwords) and add 3 to point to *PA*. affine[0] = PA, affine[4] = PB, affine[8] = PC, affine[12] = PD.

The scale matrix will not use PB and PC, we will clear them to zero.

```
    affine[4] = 0;
    affine[8] = 0;
```

Now we have a bit of fixed point math to deal with. The affine parameters are in 8.8 fixed point, our height value is in *.8 too (* means a sane amount).

We must give PA our horizontal scaling value. As mentioned above the formula is:

$$PA = diameter / width$$

To get width, we do:

$$width = d^2 / height$$

To do this in fixed point, we must shift d^2 left by 16 bits. Then, when we divide, we will end up with a *.8 value.

Hey wait a minute, this is too much math, we can simplify the PA operation a bit.

If we replace *width* with the width formula, we see the following:

$$PA = d / (d^2 / h)$$

We do a little bit of math to it and end up with:

$$PA = h / d$$

Now, as mentioned before, divides can be a little slow for the DS. Since *d* is a constant value, we can flip this operation into a multiplication problem.

$$PA = h * (1/d)$$

In fixed point, we will scale 1/d by shifting the value left 16 bits. We will then shift the result right 16 bits, this gives us much *much* better precision.

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

The format is in the correct 8.8 fixed point. The compiler will optimize the 65536/constant into a single value. We need the PD value too, which is:

$$PD = diameter / height$$

We already have *height / diameter*, so we just need the reciprocal.

```
    int pd = 65536 / pa;
```

65536 is *.16 fixed point, we divide by *.8 fixed point, we get the correct *.8 fixed point value.

Load the parameters into the affine matrix.

```
    affine[0] = pa;
    affine[12] = pd;
```

That should do it! Compile and run the game, you should see a much better boucing effect!

Previous: The bouncy ball | Contents | Next: Boingy sound