Sprite Multiplexing

on Nintendo DS

Daniel -MsK`- Borges

2007/03/15

0.1

I. The beginning

I'm currently developing a shooting game on Nintendo DS. I like shooting games with tons of bullets on screen to be dodged, those shooters called bullet hell, manic shooters, 弾幕 (danmaku), barrage shooters, etc.
I also like my Nintendo DS, so I decided to make my game on it, as it's pretty easy to get stuff to run code on it.

Nintendo DS is a wonderful handheld machine for 2D games, the 2D engine included does much of the work needed to make 2D games :

- 128 sprites per screen with rotate/scaling options
- 4 scrolling backgrounds with rotate/scaling options
- alpha blending
- etc.

But the problem for my upcoming game appears just here : ONLY 128 SPRITES !!I can barelly display my bullets with that. Because I read somewhere in the past that it was possible to get more sprites than this 128 on GameBoy Advance (Nintendo DS is just an evolution of that hardware), I began browsing the web to learn those techniques and found this :

1. using a 16colors background and draw bullets manually on it
2. use DS 3D Hardware to draw polygons which will do the sprite job
3. sprite multiplexing

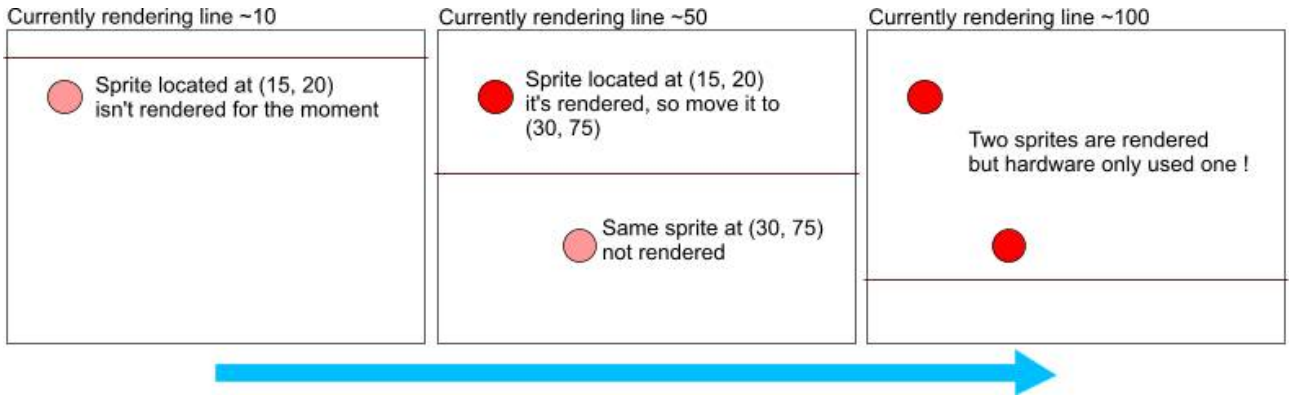
Problems with 1 and 2 :

1. it uses one of the 4 backgrounds, it doesn't take the power of the 2D engine and uses CPU.
2. DS can render 3D but only on one screen, to make 3D on both screens, you have to render a screen and then the other and so on, so you downgrade your game to 30FPS (which is not so bad but, I dislike the idea and 3D engine is not for 2D, after all)

So I decided to use the third technique.
My reference on implementing this technique is http://gba.pqrs.org/tips/sprite-multiplexer.html and it's all in Japanese... I « translated » it, now I understand how hard is the job of a translator.
This document isn't a translation of it but it's pretty much like it, as I use the same technique adapted to DS (and on both screens !) and this presentation will use the same plan.
Many thanks to Takayama Fumihiko !

II. Sprite Multiplexing ?

It's a technique that was used on old hardwares, those ones only having 8 sprites handled by the hardware like MSX, Atari, Amiga, NES/Famicom, etc.
It's pretty simple : as the screen is rendered line per line, the idea is to move a sprite between two lines to get it displayed two times !



Repeat this multiple times for a higher number of sprites and you can display hundreds of sprites without using all the sprites handled by the hardware.

The problem : here is a critical point, if you modify the sprite position while it is rendered, only half of it will be rendered ! This can be good for effects but it also can be dangerous as a sprite can become invisible.

Solution : only use the multiplexer to display sprites that are not really important visually. (bullets are important for gameplay but if only 5 lines of it are rendered on 8, for example, it's not a big problem, but an half drawed enemy, boss or maybe worse, a player, is not acceptable)

III.Seems interesting, how can I use it on my DS ?

1. Let's remember some things about Nintendo DS.

About sprites :
There is a distinction between Sprite data (what is rendered) and Sprite Information (where, what data, what palette, what effects). We only care about Sprite Information, with libnds, this information is stored in the SpriteEntry structure :

```
typedef struct sSpriteEntry {
     uint16 attribute[3];
     uint16 filler;
} SpriteEntry, * pSpriteEntry;
```

Refer to http://nocash.emubase.de/gbatek.htm to know the job of each of the 3 attributes but remember that X and Y position are stored in attribute[1] and attribute[0] respectively.
An array[128] of this structure is stored at OAM and OAM_SUB (for main and sub screen) by the hardware, so to draw a sprite you basically just have to copy one SpriteEntry to an entry of this array, the OAM (Object Attribute Memory).

About interrupts :
Nintendo DS have a lot of interrupts but what we need here is only the VBLANK (interrupt triggered when the screen rendering is finished) and HBLANK (triggered when line rendering is finished). Well, we'll not use HBLANK as it triggers every line but else VCOUNT, that triggers every line you want (use `SetYtrigger()`).

2. Objective : more than 400 sprites using only 64 true sprites

As you may already spotted, we need to draw our sprites sorted ordered by their Y-axis position. So a first approach would be to make an array of 400 sprites, fill it with your bullets, sort it (with something really fast...) and then draw the first 64 sprites, when they are drawed, replace the 64 old sprites by 64 new ones at the good HBLANK, and so on.
I tried. It doesn't work. Quicksort is way too slow !
Here comes Takayama Fumihiko !
The technique he used in his bulletgba is the better I saw, I only have one reproach I'll discuss later.
We still keep our array of 400 sprites (well, he used an array of 512 and I use an array of 1024 (-:) but we use more stuff : we divide the screen in blocks of lines, add an array to count sprites per block, set an array of pointers to the first array and fill it !
Hummmm, this is not clear, let's see :

1. Count sprites per block
2. Set pointers to the big sprite array to know where starts each block
3. Effectively add each sprite to the big array using the block-pointers

Doing that this way, sprites are sorted automatically per block !

This is a two pass technique. I have to work on that to make it one pass only, this is the only thing I don't like in it.

3. Let's code a bit (how I^WTakayama implemented it)

I started reading `spritedoubler.hpp` and `spritedoubler.cpp` of bulletgba and... I almost did the same... I just find my code easier but I'm new to C++ so it maybe a lot improved. I just doubled everything for dual screen and did faster HBL interrupt handling functions.

First thing to do : doubler buffering. If you try to modify what is currently drawed, it would be ugly, so you need 2 structures to handle your multiplexing data, here is mine :

```
#define MAX_SPRITES 1024
#define MAX_SPRITES_PER_BLOCK 64
#define BLOCK_LINES 4
#define BLOCKS (SCREEN_HEIGHT / BLOCK_LINES)
...
struct SET { // S.E.T. : Sprite Entry Table
     SpriteEntry Sprites[MAX_SPRITES];
     u16 registeredInBlock[BLOCKS];
     u16 addedInBlock[BLOCKS];
     SpriteEntry * Starter[BLOCKS];
     SpriteEntry * last;
};
```

Sprite Entry Table, what an ugly name, don't you think ?
You'll have 4 of these, two to double buffer the main screen and more two for the sub screen.
`Sprites` is ...our sprites.
`registeredInBlock` contains the sprite count per block, `addedInBlock` contains the same information, it's resulting of the two pass system, one reason to dislike it.
`Starter` is the pointers to Sprites giving us the distinction between each block.
`last` is a pointer to `Sprites` with the last `MAX_SPRITES_PER_BLOCK` sprites in `Sprites`, so the last sprites on screen will always be displayed without any glitch, good.

Now that you have this structure, you just have to fill it and display it, so let's see how to display it.
Firstly, you need to swap your buffers on the Vblank interrupt, not too hard.
Next, on each MAX_SPRITES_PER_BLOCK Hblank, you may think, just copy from Starter[current line] to OAM, but this will not work : only the first `BLOCK_LINES` lines of each sprites will be displayed !
Here comes the `optimize()` function. This function takes each Starter and drives it back of `MAX_SPRITES_PER_BLOCK – registeredInBlock`[current line], thus, each HBL interrupt, you'll display your new sprites and a maximum of old sprites to complete them. Here is some problem, some sprites will not be displayed correctly if you have too many sprites in a block, just keep that in mind when you make your game to avoid ugly things.

IV. Conclusion

From what I brain-stormed and what I saw on the web, this is the best technique so far to display LOTS of sprites (and not too big ones, like bullets). The performances are good, my demo uses ~Vblank time to multiplex 800 sprites (well a bit more, as some sprites are displayed on both screens when they go from one screen to the other). This could be better with a one-pass system, I believe but for the moment I still prefer the two-pass version to the one-pass versions I did.

Hope this document will help you !

You can also use « sprite moving when drawing » to make some effects like moving vertically a sprite while drawing to make a laser or moving it from left to right while displayed for wave effects, etc.

Have fun !