Shonumi's Blog

Just another 6bit.net Gaming & Emulation site

- Home
-

Type text to search here...

Home > Emulation Development > Emulating the GBA

Emulating the GBA

April 19th, 2015 shonumi Leave a comment Go to comments

Like 0 Share

- Stur

Emulating the GBA

Most of it was written from scratch with a few bits taken from the old GB Enhanced project. After constant programming, I've come away with something that will decently handle a number of GBA games. In this article, I will share my experiences, trials, and knowledge that I gained this past year.

Advanced Emulation 201

After more or less completing DMG and GBC emulation in GB Enhanced, my Gameboy emulator (GBE for short), I actually began to feel a bit bored. At the time I was working on implementing custom graphics for GBC games, which required a more sophisticated approach than what I previously wrote about. Custom graphics is a fascinating subject, but I'd been working on it for months, and I wanted something to break up the pace. Having tackled the Gameboy, what would be more worthwhile than tackling its successor? I had been looking at Martin Korth's GBATEK for some time, and I felt ready to emulate the GBA in some capacity.

Unlike the DMG and GBC, however, there were no easy coding tutorials online. Articles like Imran Nazar's GB Emulation in Javascript or Codeslinger's series of articles about the same subject simply don't exist for the GBA. I'd be going in blind with no hand-holding. Granted, there are many open source GBA emulators to reference, but looking at someone else's code doesn't magically mean you understand how a game console actually works. I'd just have to rely on what I'd previously learned, apply it to a new system, make a few mistakes along the way, and ultimately progress from there.

Probably the first major step anyone takes when emulating a system is to establish a working CPU. Without that, nothing else will run as it was designed to. The Gameboy Advance uses a 32-bit ARM7TDMI CPU, running an ARMv4 architecture, clocking in at a whopping 16.78MHz or 2^24 Hz (impressive for its time, especially as a mobile device). It also had a Z80 style CPU for backwards compatibility with DMG and GBC games, but that was unavailable to GBA software. The ARM7TDMI is also based on Reduced Instruction Set Computing (RISC).

This CPU, like many ARM CPUs, has two modes of processing instructions called ARM and THUMB. ARM instructions are encoded as 32-bit opcodes, while THUMB instructions are encoded as 16-bit opcodes. Most GBA games generally used ARM instructions when inside the GBA's 32KB Work RAM, while THUMB was used everywhere else, especially in memory areas with 16-bit buses (a 32-bit ARM instruction would require two reads while using a 16-bit bus, leading to reduced performance).

ARM opcodes provide a higher level of flexibility as it concerns the instruction itself. A single ARM instruction could, for example, copy one register shifted by another register into a third register. THUMB can match ARM's functionality for the most part, except it may take multiple THUMB instructions to achieve the same result in some cases. Furthermore, THUMB opcodes only have access to a limited number of the CPU's registers. THUMB opcodes always set the CPU's condition flags (such as when a mathematical instruction results in zero), but for ARM instructions this is optional.

ARM instructions are always executed conditionally as well, that is to say the instruction does not run unless certain CPU flags are set based on the outcome of previous operations. To some extent, this can be used for psuedo-branching code, but it shouldn't be abused; unexecuted instructions still run for one CPU cycle. Overall, in layman's terms, ARM instructions can be thought of as longer but more advanced with lots of features; THUMB instructions can be thought of as smaller with concise features.

One more thing about the CPU I should mention: it has a surprising number of registers. It has 13 General Purpose Registers (R0-R12), a Stack Pointer (R13), a Link Register (R14), and a Program Counter (R15), a Current Program Status Register and a Saved Program Status Register. But wait, there's more! Some of these registers are banked depending on the state of the CPU. For example, the Stack Pointer used during normal operations will be different from the Stack Pointer used during interrupt handling. It sounds confusing, but in

Coming from the Gameboy's Z80-style CPU, I was pretty overwhelmed by the GBA's CPU and ARM in general, especially when it came to decoding the instructions. I expected something like the GB's extended **0xCBxx** 16-bit opcodes, but nothing like ARM/THUMB. Instruction encodings are all over the map to say the least. At any rate, I started trying to implement some ARM instructions, but I eventually switched to THUMB due to its simplicity. I chose to decode each instruction, determine if it belonged to a particular opcode group, then call a function specifically written to handle that opcode group. Opcode groups describe a family of instructions such as THUMB.1, THUMB.2, ARM.5, and ARM.6, as specified in ARM's documentation.

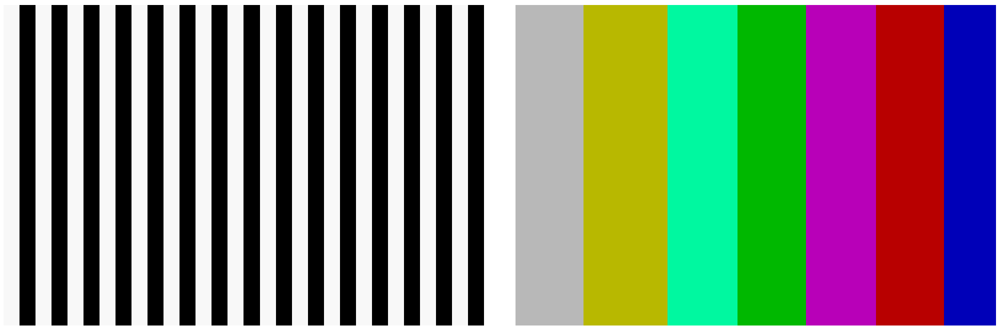
This was perhaps the hardest and least interesting part of emulator development for me. I spent a lot of hours just plugging away at my keyboard trying to recreate a CPU from the ground up. It's just not sexy work. There are no graphics, no sound, and hardly anything but console-based output. You can't get to the exciting things however until you've built up the emulated CPU enough. At this time, I was only doing GBA emu-dev as a side project, something to do whenever I wasn't playing video games. From April to August, not a lot happened. My emulated CPU could run at least, but a lot of instructions were missing or incorrectly implemented. At the end of the summer though, I felt it was enough to take the next step: graphical output.

Graphics: The Fun Stuff

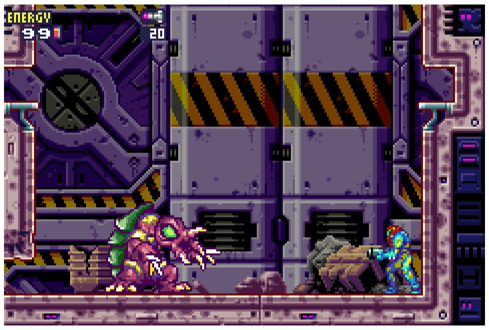
Obviously I could have tried running my emulator on any number of commercial games, but since the emulated CPU was incomplete, it was bound to exit due to an opcode that couldn't be handled. I wanted to see something on screen, to know that my efforts were at least heading in the right direction. The answer, I decided, was to whip up some homebrew of my own. Usually it helps to be involved with homebrew before jumping into an emulator, since you have to learn the system in order to successfully program on it. The reverse holds true to some degree, however; in making an emulator, you learn how to make homebrew.

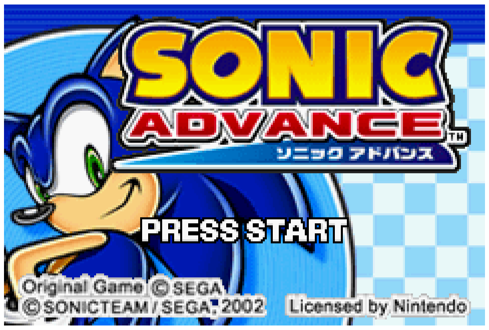
After studying the GBA in GBATEK, I figured what was necessary to get the screen to change colors. From there, it was only a matter of writing the ARM/THUMB assembly code. Fun fact, I did this all in a hex editor, using a text file to keep track of what I'm doing. While this seems silly, my emulator at the time could only run a handful of instructions; I had to be sure the ROM file I was creating executed only the exact instructions I needed.

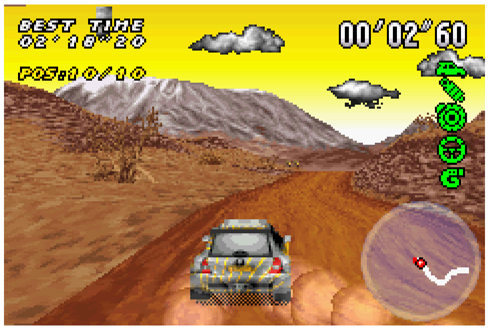
The first thing I did was change the whole screen to green. After that, I started working with the GBA's tile-based graphics to create simple black and white stripes, and an image in the vein of old TV test patterns, specifically the SMPTE color bars. After that, I messed around with implementing interrupts on a small scale, mainly the VBlank interrupt. The demo for this simply cycled through every possible color the GBA can display in an endless loops, going through ~60 colors a second. Initial errors effectively lead to potentially seizure inducing explosions of color, unfortunately.


Some of my earlier tests

tile-based. Most games operate in Mode 0, which allows for 4 BG layers. Mode 1 allows for the 3rd BG layer to be scaled and rotated (but not the others, 4th is disabled). Mode 2 allows both the 3rd and 4th to be scaled and rotated (disabling the 1st and 2nd BGs, however). Modes 3-5 are different implementations of bitmap based graphics. These were used for displaying detailed still images or in some cases used for software rendering graphics on-screen.





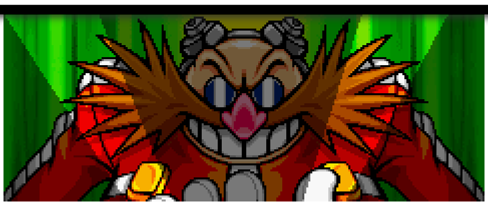


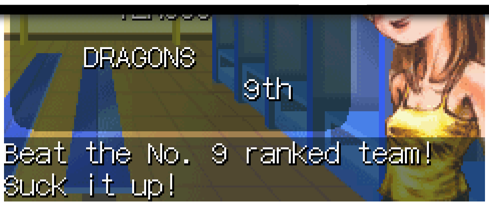
Metroid Fusion uses Mode 0, allowing for many complex layers and scenes. Sonic Advance uses Mode 1 for the title screen; the logo rapidly shrinks creating a zooming effect. V-Rally 3 renders its 3D graphics entirely in software and uses Mode 4 to display the frame as a bitmap.

The GBA had an impressive number of sprites (or OBJs), 128, doubling the amount found on the DMG and GBC. The newer handheld does away with old sprite limit of 8 per-scanline. Under the most ideal circumstances all 128 could be rendered on a single scanline, but more often than not the number was less due to several factors (how many CPU cycles were available for OBJ rendering at any given time, how many cycles it takes the hardware to process OBJs of various types).

All sprites on the GBA can be rotated and scaled as well, however, the hardware only had 32 unique "slots" that defined scaling/rotation parameters. That is to say, while all 128 sprites could rotate at an angle, you could only define 32 different angles at any given moment. Also of great advantage were new variable sprite sizes. On the DMG and GBC, sprites were only 8×8 or 8×16. On the GBA you get 8×8, 8×16, 16×8, 16×16, 32×8, 8×32, 32×32, 32×16, 16×32, 64×64, 64×32, and 32×64! That's quite a lot of options, and the increased flexibility went on to allow developers to be more creative in art styles, overall game design, and programming.

Lastly, the GBA provided hardware to process so called "special" effects, comparable to the color math operations from the SNES. The GBA could decrease an image's brightness (making the pixels closer to black), increase its brightness (making the pixels closer to white), or perform alpha blending between two layers for transparency. The first two are often used for fade-ins and fade-outs; previously games on the DMG and GBC had to manually manipulate palettes themselves to recreate the same results. Some GBA games do this anyway (see Riviera: The Promised Land's logo screens), but not often. Alpha transparency, as one can imagine, proves very helpful in creating a number of visual effects; the very first scenes of Golden Sun impressively use it for lighting that even interacts the player's sprites.

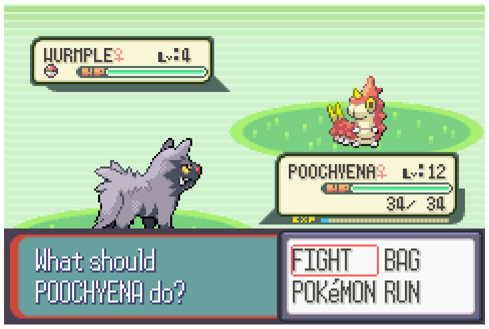




Sonic Advance actually darkens the screen, leaving some sections at normal brightness to create the impression of lights. Super Dodge Ball Advance shows alpha transparency at its full potential: fancy text boxes!

While these special effects can be applied to the entire screen, often they were limited to certain areas of the screen thanks to the GBA's Window functionality. For those that know the DMG/GBC Window, the GBA's Window system is quite different. A window basically dictates a virtual rectangle on the LCD. The GBA then decides what will be displayed inside that window (which background layers are enabled, if sprites are enabled, and if the special effects applies to this area). Even more powerfully, windows determine what is enabled/disabled **outside** the rectangle as well! The GBA has 2 rectangular windows, but it also offers a third called the Object Window. Some sprites may change into Object Window Mode based on their parameters in OAM; the sprite is not displayed, however, any pixels that the GBA would have displayed basically act as a mini window.

This was often used to create masking effects, most notably done in Pokemon Ruby/Emerald/Sapphire and Fire Red/Leaf Green. These games actually have 2 copies of a Pokemon's sprite in battle, one in Object Window Mode on top, and a regular one on bottom. The top one is not displayed, but instead it is used as a window that draws a colored background in the shape of the sprite. This is then alpha blended over the normal Pokemon's sprite on the bottom. That's how Pokemon change colors whenever a stat changing move like "Growl" or "Leer" are performed. Pretty neat, huh?



Step 1: Draw a regular Pokemon + an invisible "shadow" copy on top. Step 2: Take the BG (shown here for demonstration, not like that on a real GBA). Step 3: Draw the BG in the form of the "shadow" copy. Step 4: Blend the result with the regular Pokemon. Huzzah!

Black Screens of Death

So at this point of my adventure, I had limited working homebrew. It was time to take the dive into booting commercial games. To me this was a major step, one that I took seriously. I gave myself a deadline for October 15th, 2014, roughly 2 and a half weeks. The thing about emulating video game systems is that there are a number of "moving parts" that all need to work together in order to get anywhere.

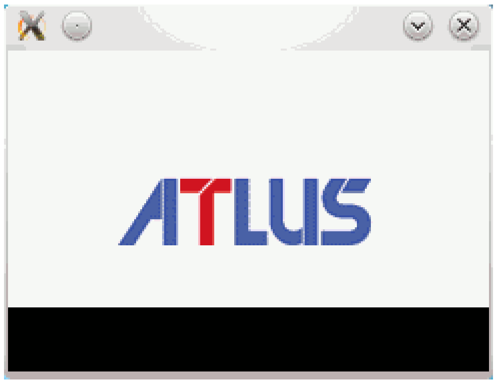
In an emulator's infancy, a lot of those parts are marked TODO or skipped over for the sake of getting something working, even at the cost of glitches. The foremost thing for some emu-devs is to put something, anything, on-screen. If emulators were ships, mine was in no shape to sail, but I set off anyway.

I picked one game that didn't crash or exit or otherwise explode in my emulator: Super Dodge Ball Advance. It seemed to keep running in an infinite loop, which at that time was a good sign. It didn't abort due to

The first time I ran Super Dodge Ball Advance, not much of the emulated code was correct. Register values in the CPU were off, memory values were seemingly random, and weird instructions were being executed. Some of the ARM/THUMB instructions I did emulate weren't even accurate, and that caused a lot of headaches early on.

The process of debugging basically involved sitting in front of my would-be emulator and a working emulator such as VBA-M or no$gba, stepping through various instructions, trying to see what was going wrong. It was a seemingly never-ending quest to vanquish all sorts of nasty bugs. I thought I'd never see this thing running; the black screen began to taunt me day after day. Eventually, however, I reached a breakhrough.

After enabling two or three Software Interrupts (SWIs, also known as BIOS functions) via High Level Emulation, and after adding basic Direct Memory Accesses (DMAs, essentially quick/automated memory transfers), I got my first image, the Atlus title screen. After that, my emulator promptly showed nothing but garbage, but it was a start nonetheless. From there, I clawed my way to the title screen by improving LCD emulation. After that, I implemented input so I could actually go in game. I could in fact start matches, but the sprites were badly rendered and the ball could not be thrown due to unimplemented math-related SWIs.



Ah, the good ol' glitchy days, when nothing worked…

Still, with this success, I started working on other games. A lot of my early efforts were focused on simply getting games to boot. I figured I'd take care of any outstanding issues once the emulator was advanced enough to tackle more games. One of the best motivations for emu-devs is seeing their favorite games brought back to life. Of course there are plenty of existing solutions, but it's a completely different feeling when you're seeing code that **you** wrote running games like the real thing.

I went after Kirby: Nightmare in Dreamland, Bomberman Tournament, and Riviera: The Promised Land, the classics of my teenage years. After much hard work, I saw them booting and playing, although imperfectly at first. Many games continued to plague me with their black screens, and it was only after hours of debugging that I saw progress at all. Today, most games will run with minor issues thanks to all the time I spent going through hell during these months.

Some of the specific highlights and milestones of my journey
* Booted Super Dodge Ball Advance -> October 9th, 2014
* Booted Bomberman Tournament -> October 22nd, 2014
* Booted Final Fantasy Tactics Advance -> November 2nd, 2014
* Booted Advance Wars, Megaman Zero, and Megaman & Bass -> November 14th, 2014
* Added SRAM saving -> November 15th, 2014
* Booted Megaman Battle Network 1 & 2 -> November 22nd, 2014
* Booted Final Fantasy: DoS, Lunar Legend, Golden Sun -> December 12th, 2014

Last time I tackled sound, I saved it for last. Relatively speaking, although sound is one of the easier things to get right, personally I like to wait until I know most of the emulator is in good shape. Also, sound programming is not my forte. Pixels, sure, I get them, but I when I programmed the sound code for the original GBE project, I was new to things like "frequencies" and "amplitudes" and "periods". So to say the least, I avoided it because it's not my strong point; I didn't feel like bashing my head trying to get things right when I could have been working on other parts of the emulator. Eventually, I ran out of other parts to code, so around March of this year I took a breath and dove right in.

The GBA has an almost exact replica of the DMG/GBC sound controller as it relates to its Sound Channels 1-4. Channel 1 generates a square wave with an envelope and sweep function. Channel 2 generates a plain square wave, no envelope, no sweep. Channel 3 plays 4-bit digital samples. Channel 4 generates white noise with an envelope function. For the most part, I copy+pasted my old code from GBE and renamed variables as necessary. This proved enough 9/10 times.

What's interesting about the GBA is that Sound Channel 3 was actually improved in functionality. The DMG/GBC only allowed for a total of 32 4-bit samples to be stored in RAM. The GBA expands this by adding a second bank. Sound Channel 3 can freely switch between banks, meaning two different sounds can rapidly play without much effort once the sample data is loaded. Channel 3 can also play all 64 samples as a single sound. Aside from those changes though, the Sound Channels do not significantly deviate from their predecessors. Some GBA games don't even use them; instead they exclusively rely on the DMA sound channels.

There are two such DMA sound channels on the GBA, simply called Direct Sound A and B. The GBA has 4 DMAs channels, DMA0, DMA1, DMA2, and DMA3. DMA1 and DMA2, however, have a special mode that the others don't called Sound FIFO. Essentially, every time one of the GBA's hardware timers overflows (always Timer 0 or Timer 1), the GBA pushes 8-bit signed samples to the sound controller from a FIFO buffer. DMAs add data to the FIFO automatically, meaning no intervention is necessary from software. Whereas the 4-bit samples on Sound Channel 3 can only express 16 different amplitudes, the 8-bit samples for the DMA channels can express 256 different amplitudes making for much richer sound quality. Their output frequency was determined by the hardware timers, so it was possible to have 22KHz audio, albeit at the cost of cartridge space.

The DMA channels were new territory for me, but they actually turned out to be much easier than expected. All I had to do was keep feeding my code the samples and play them back. Having said that, sound quality is only acceptable at this time; it's not perfect, and the code could definitely use a lot more love. Even so, with sound roughly complete, I largely had a functional, albeit basic GBA emulator in my hands. I could play many of my old games from start to finish.

Why do all of this?

There are already plenty of great GBA emulators on many platforms. What's the purpose in spending hours of my time to make yet another one? I'm not looking for people to use my work over VBA-M or mGBA. I have no pretensions about becoming "the next big emu" or usurping popular ones. I don't care about that; it's of absolutely zero interest to me.

I'm just someone who loves video games, loves emulation, and loves a challenge. I wouldn't have learned half as much as I did through this ongoing experience if I had just read GBATEK and looked at someone else's source code. I've now gained valuable insight into how one of my favorite systems from yesteryear works on a low-level. When it comes down to it, the process was thrilling, frustrating, wonderous, daunting, **but ultimately fun**.

However, I'm a little more ambitious than just that. In my first article, I wrote about true "internal resolution" in Nintendo DS emulators at a time before we had amazing work like Desmume X432R. I said I had a pretty strong DIY attitude, so naturally the next step from the GBA is the DS. I have every intent of taking on the Nintendo DS eventually, and I want to push what's been done in DS emulation, primarily in the areas of custom textures and custom shaders. Don't get me wrong, I use Desmume X432R a lot, but this is just a dream I got, to make my own emu. That's all years down the road, but I think I'm on the right track.

Now, hear me out on the last of my crazy plans. I'm going to aim to make a 4-in-1 emulator, from the DMG to GBC to GBA to DS. I'm already in the process of figuring out how to make the first three work together. Once DS emulation starts up, I'll go from there and see how to integrate it. Like I said, I don't care who uses it or if it ever gets noticed on the scene; emu-dev is a hobby and a passion now.

For anyone interested, all my progress starts with GB Enhanced+ or just GBE+ for short. The old GBE project is dead, long live the GBE+ project. It's the same as the old project (custom tile replacement) just with better code and a GUI (no more CLI only) and it's going to be the base for everything else I want to do.

Well, that's all I got this time. Was it worth a 2 year wait for an update? I don't know, but at least everyone knows I haven't been sitting on my hands. We'll see how and where things go from here ;)

Categories: Emulation Development Tags:
Comments (0) Trackbacks (0) Leave a comment Trackback

1. No comments yet.

1. No trackbacks yet.

Name (required)
E-Mail (will not be published) (required)
Website

Subscribe to comments feed
Submit Comment
Emulating Game Boy Games With Custom/Colorized Sprites
RSS

- Entries RSS
- Comments RSS
- WordPress.org

Categories

- Emulation Development

Blogroll

- Documentation
- Plugins
- Suggest Ideas
- Support Forum
- Themes
- WordPress Blog
- WordPress Planet

Archives

- April 2015
- July 2013
- April 2013
- March 2013
- February 2013

Meta

- Log in

Top WordPress

Bad Behavior has blocked **13** access attempts in the last 7 days.