GameboyAdvance Programming Assignment 3

Cangming Liu

March 15, 2008

1 Overview

In Part4, the final project of this course, you will be given an API for a game-engine-like library for the GBA hardware. This system will take care of the hardware details of GBA programming, so you, the programmer, may focus on the design of your game.

In this assignment, you will be provided an incomplete version of this game engine, and it is your task to fill in the missing part: sprite functionalities. The following sections give the necessary background information for this assignment.

1.1 Hardware

When the GBA tries to render a sprite, there are three sections of memory that it looks into:

Video Memory: This section starts at 0x0601:0000 and stores the pixel information of each sprite.

OAM Memory: This section starts at 0x0700:0000 and stores attribute information about each sprite.

Palette Memory: Starts at 0x0500:0200. This section of memory stores color information.

Each rendering cycle roughly goes as follows:

As you have learned in section, each OAM Memory entry is a set of three 16-bit registers, and the lower 10 bits of the third attribute is the tile number. For each cycle, the GBA will look at all OAM entries and try to render those with non-zero tile number.

For each non-zero sprite, the GBA will look at attribute 0 and 1 for information such as position and size, and it will then fetch the correct tile from the video memory. The tile to fetch is determined by the tile number. The video memory can be thought of an array of length 1024, and each element in the array is a 32-byte block of data. For a 256-colored sprite, which is what we will use for this project, each pixel corresponds to one byte. Thus for a normally sized sprite (8x8), it takes two blocks of data to store all the pixel information.

For each block of data, the GBA will go into the palette memory and fetch the correct colors. Again, we will think of the palette memory as an array of length 256, with each element representing a single color. So for each block of data, each byte will represent a pixel, and the value of that byte is the index into the array of colors.

The palette memory is a straight forward array of 256 values, so it is not interesting to us. The video memory and the OAM memory, however, need to be managed carefully. For example, when a new sprite is created, it will require a number of blocks of video memory. There must be some code to guard this piece of memory so no other sprites can claim this section. Similarly, when a sprite is destroyed, the blocks of video memory it used is no longer useful, so we must recycle that piece of memory to prevent quickly filling it up with useless data.

1.2 The Game Engine

It is designed with these ideas:

1.the logic for games are inherently simple and repetitive. Recall Mario from Part2. Once you had all the sprites set up and showing up on screen, the rest of the logic is simple: if keyleft, move left; if keyright, move right; if collide with the... bad guy, die.

2.the general setup for sprites and backgrounds are similar. Different sprites really only differ in appearance. So if provided the tile data, constructing a sprite should be easy.

There will be a more detailed document for this game engine, but for this part, you only need to know of these classes:

Kernel: the core of the game engine. It coordinates all the pieces of the game so they work coherently.

Module: This is an abstract superclass for the pieces of your game. Each Module can be attached to the Kernel, and once all necessary Modules are attached, calling the `run()` method of the Kernel will make the game start running. There are three functions that must be implemented:

`void insert();`

This function is called when a module is first attached to the Kernel. It can contain certain setup instructions.

`void step();`

This function is called at every cycle of the game while the Kernel is running.

`void clean();`

This function is called when the module is removed from the Kernel. It should contain instructions to any special clean up code.

2 Your task

As mentioned above, your task is to implement the sprites part of the game engine. Please be sure to familiarize with how sprites work on the GBA. Your Part2 sprite.h and sprite.cpp give a good idea.

2.1 The Sprite Interface

Each sprite should act like a module in this game engine. Therefore we mandate that, in your submission, there must be a Sprite class which inherits from Module. Other that this requirement, you are free to hand in any *reasonable* interface for a sprite. We leave the design of this class as an open-ended problem to you, but here are a few things to consider:

- Sprite will inherit the three abstract functions from Module. What should each function do in the case of your Sprite function? Notice the step function: what does a sprite usually do at every cycle of the game?

- What are some of the things you would like to do with a sprite? Think of sample use-cases and determine what functionalities are needed by these cases.

Keep in mind that the game engine must be general so it can accommodate the different varieties of games. So your sprite class must be able to take any combination of inputs and produce the expected results on screen.

2.2 The Implementation

Once you have decided on an interface for the sprite class, work on a primitive idea for the implementation. Basic functionalities for the sprites such as setting the position or size should be similar to what you did in Part2. What is different for this part is this: your sprite class is supposed to "take any combination of inputs and produce the expected result." This means that sprites can be created, destroyed, inserted to the Kernel, removed from it. There could be any number of sprites, and those actions can be performed in any order. Again, points to consider:

- The video memory, which holds the tile data, is limited. If certain sprites are removed from the Kernel, there can be gaps in the video memory. Since the video memory isn't technically full, we should still add new sprites to it. Thus it could be necessary to defragment your video memory.

- The same story goes for the OAM memory... or does it?

- In video modes 3,4, and 5, only the upper half of the video memory is available for sprite use. There should be a way to pass this argument into your game engine.

- Note that when you move things in video memory, the corresponding OAM entry must be updated with the new tile number.

- What are some possible errors and how would you detect them?

We highly recommend that you put the memory management logic in a separate class, so the sprite class focuses on moving things on screen, and this class takes care of the custodial work. This allows easier maintenance of code and readability.

Now, before going on to step 3, write down your thoughts of how to go about doing this project. Don't write code. Diagrams are usually helpful to draw your different classes, or you can have a flow chart of your program logic. Include function stubs and specifications. Include class headers and documentations. Explain your thoughts. Compile (pun intended) all these into a neat and organized document, and submit it. One measure of how good your design document is is to imagine that if a team of hackers received only this document and is not allowed to ask questions, would they be able to write the code you intended?

2.3 Making It All Work

Now that you have thoroughly thought about your project and already have method stubs and class headers written down, simply implement them.