CS 212 GBA
The Anatomy of a Barebones GBA Program
Spring 2007

1 Programming for the Gameboy Advance

Programming for the GBA is a bit of a change up from what you are probably used to dealing with in writing programs in Java, though there are huge overlaps in the syntactical conventions of Java and C. You'll find this to be really useful if you mainly or only are familiar with Java. The main differences that you will be made aware of is that C is not object oriented, however we will make liberal use of pointers and structures to make up for this lacking, and you will find these are quite ample for our purposes. We will discuss structures and pointers in depth in later chapters. For now we will dive right in to some very basic GBA programming, to give you a feel for what manipulating the GBA actually means in a programming sense.

2 A Single Sprite

We'll start by just having something that compiles to the GBA ROM format and puts a very simple sprite on the screen. Lets look at a very simple main method.

```
int main () {
    SetMode ( MODE_1 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE );
    InitializeSprites ();
    // The game loop
    while (1){
        sprites[0].attribute0 = COLOR_256 | SQUARE | ( 16 );
        sprites[0].attribute1 = SIZE_8 | ( 24 );
        sprites[0].attribute2 = 52;
    }
}
```

The first thing that happens is that `SetMode()` is called, with a parameter that is defined by a bitwise operation, which is what defines many of the configuration options when interfacing with the GBA. In this case `MODE_1 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE` sets the graphic mode to be 1, enables sprites, says that sprite data will be stored in a 1 dimensional array, and that background 2 is enabled. There are 6 graphics modes, modes 0 through 2 are tile based modes (which means we will use (perhaps multiple) layered backgrounds for putting tiles and sprites on the screen), and modes 3 through 5 are bitmap based modes (which means we draw directly to pixels on the screen). You will almost completely be using tiled background modes, so we will focus on that for now. Bitmapped modes are useful for making title screens, things like that, though they could be applied all over a game. There are 4 different backgrounds which can be used simultaneously (when in the correct mode) and they all have some different properties. The important information is summarized in this table:

Tiled Video Modes and The Backgrounds They Work With

Mode	Background	Rotation/Scaling
0	0,1,2,3	No
1	0,1,2	Yes(2)
2	2,3	Yes(2,3)

Backgrounds and Their Sizes

Background	Max Resolution	Rotation/Scaling
0	512x512	No
1	512x512	No
2	1024x1024	Yes
3	1024x1024	Yes

In these tables there is an extra column that says whether or not this mode will work with rotation and scaling matrix. We won't go into this yet, but it's important to be aware of how `SetMode()` affects what the GBA is capable of.

The next thing that happens is `InitializeSprites()` is called. Lets look at the code for initialize sprites so we can understand just what it does.

```
void InitializeSprites() {
  u16 loop;
  //transfer palette data into memory (16 colors)
  //only 5 are used in this sprite set
  for(loop = 0; loop < 5; loop++){
    OBJ_PaletteMem[loop] = sprite_palette[loop];
  }

  //transfer ball sprite into memory (2x8)
  //loop starts at 16 (after pallete data)
  for(loop = 16; loop < 32; loop++) {
    OAM_Data[loop] = ball[loop-16];
  }
  // Transfer paddle sprite into memory.
  // Since the paddles are all white,
  // the same block of four white pixels
  // is copied to the right dimensions of
  // the paddle
  for (loop = 32; loop < 96; loop++) {
    OAM_Data[loop] = paddle[0];
  }
  // transfer numbers (0-9) into memory ()
  for (loop = 96; loop < 256; loop++) {
    OAM_Data[loop] = numbers[loop-96];
  }
  // transfer letters (A-Z) into memory ()
  u16* temp_alphabet;
  temp_alphabet = (u16*)alphabet;

  for (loop = 256; loop < (256 + (26*32)); loop++) {
    OAM_Data[loop] = temp_alphabet[loop-256];
  }
}
```

Basically what this does, a number of times, is copy raw sprite and palette data into the actual GBA memory (known as OAM_Data). Different sprites are different sizes, which is why each loop is executed different numbers of times. There are also a few peculiarities. Firstly the first 16 bits of OAM are used to store the palette information, so we begin storing our sprite data accounting for that, which is why the first loop (which transfers the ball sprite to memory) starts at 16, and not zero. Another interesting thing here is that the paddle sprite comes from a single line of sprite data. Looking in `Sprites.h` we see:

```
paddle[( 1 )] =
{
  0x1111,
},
```

What is happening is that single bar is being copied into memory multiple times in successive locations, so we end up with the full paddle, all from just one line of sprite data.

And now we get into the meat of any GBA program, the while loop. A GBA works by setting up the environment that you want to use, and then updating that many times per second, looking for user input, updating enemy sprites positions and things like that. In this very primitive program all we are going to do is place a sprite in a fixed position on the screen. We do this by manipulating the `sprites[]` array. Each sprite in the sprite array has three attributes that you should be mindful of. `attribute0` controls the color mode underwhich the sprite was made, the shape that you want for the sprite, and the y position of the sprite, all put together by a bitwise OR. `attribute1` controls the size of the sprite, and its x position. `attribute2` controls which place in sprite memory that it should be getting the sprite data to display on the screen from. In the case of our simple program we have:

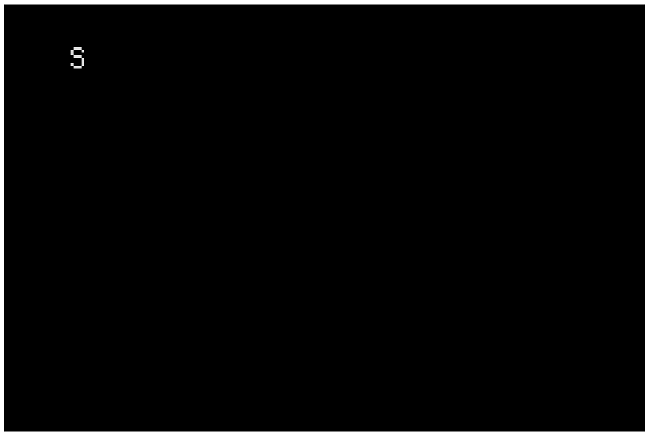
```
  sprites[0].attribute0 = COLOR_256 | SQUARE | ( 16 );
  sprites[0].attribute1 = SIZE_8 | ( 24 );
  sprites[0].attribute2 = 52;
```

Here we are setting the 0th sprite's attribute0 so that we are using 256 colors, the shape of the sprite will be square (tall, wide are some other common choices) and the y position of this sprite will be 16. The sprite's attribute1 is set so that the size is 8x8, and the x position is 24. Lastly attribute2 is set to 52, which is the 52th position in OAM_Data, which in this particular case is the letter S. There are alot of different values that you can set to these attributes, but for now just focus on the ones we have given to you in this table:

Index	Sprite	Attribute0	Attribute1	Attribute2
1	Ball	COLOR_16 | SQUARE | y	SIZE_8 | x	1
2	Paddle	COLOR_16 | TALL | y	SIZE_16 | x	2
. . .				
6	0	COLOR_16 | SQUARE | y	SIZE_8 | x	6
. . .	1-9			7-16
16	A	COLOR_256 | SQUARE | y	SIZE_8 | x	16
. . .	B-Z	*		18-66

256 color sprites take up twice the space per-pixel, so we index with alternating numbers

So when we compile this, we get something like the following:



3 Moving The Sprite

Now that we have put a sprite on the screen, lets do something useful with it. The beginning of doing anything useful, and of implementing a game, is being able to manipulate our sprites based on user input. This is very easy with the macros that have been setup in the GBA header files we have provided for you. To do this we use the `keyDown()` macro along with the assorted defines for the various keys (the following is from `gba_keys.h`):

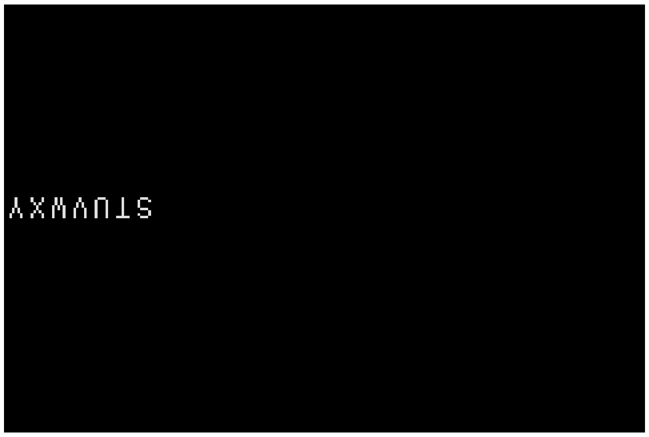
```
#define KEY_A                 0x001
#define KEY_B                 0x002
#define KEY_SELECT            0x004
#define KEY_START             0x008
#define KEY_RIGHT             0x010
#define KEY_LEFT              0x020
#define KEY_UP                0x040
```

```
#define KEY_DOWN            0x080
#define KEY_R               0x100
#define KEY_L               0x200
```

To make use of these, we will look for user input during each iteration of our main while loop. We put the following code in, so that our `main()` looks like:

```
int main () {
    SetMode ( MODE_1 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE );
    InitializeSprites ();
    // Position variables and their bit - masks .
    // You can use the same masks for all sprites
    s16 x = 24;
    s16 y = 16;
    // The game loop
    while (1){
        // Changes the values of x and y according to
        // input from the D - Pad
        if ( keyDown ( KEY_LEFT )) x--;
        if ( keyDown ( KEY_RIGHT )) x++;
        if ( keyDown ( KEY_UP )) y--;
        if ( keyDown ( KEY_DOWN )) y++;
        // Applies the updated x and y values to the sprite
        sprites[0].attribute0 = COLOR_256 | SQUARE | ( y );
        sprites[0].attribute1 = SIZE_8 | ( x );
        sprites[0].attribute2 = 52;
    }
}
```

The things to note here are that we have added two new variables x and y, that will hold our sprites x and y positions on the screen. We can see that these variables are used to update the *x* and *y* position as they are put into the sprite's `attribute0` and `attribute1`. We also see that the we look for user input from each of the four directional buttons during each iteration of the main while loop. `if ( keyDown ( KEY_LEFT )) x-;` is saying, if the left key is currently down (*NOT* wait for the left key to be pressed) then decrement *x*. The functionality of the other statements works the same. Compiling and running this, we see we are now able to move our S around the screen. However, if we move too far to the left of the screen, something unexpected happens:



What exactly is happening here? The answer is not obvious, but what is going is that when our x value is decremented below 0 it begins to underflow into the other bits of the sprite's OAM_Data, setting some unexpected flags, or changing the size to something undesirable. How can we fix this? The obvious solution is to fix our x and y positions so that they are always between 0-256. The usual mathematical solution to do this would be with the modulus operator, which is unfortunately too slow to use on the GBA hardware, so we must use the bitwise AND operator. If we AND the x and y positions with the hex numbers 0x01FF and 0x00FF then we will ensure that they are always between 0 and 255, and will never effect the other bits in the registers will be altered undesirably. The reason we use 0x01FF on x is because in `attribute1` the x position is actually 9 bits. In `attribute0` the y position is only 8 bits. Thus the use of 0x01FF and 0x00FF.

So we create two new variables:

```
u16 xMask = 0x01FF;
u16 yMask = 0x00FF;
```

and bitwise AND then with the x and y positions when we update `attribute0` and `attribute1`, like so:

```
sprites[0].attribute0 = COLOR_256 | SQUARE | ( yMask & y );
sprites[0].attribute1 = SIZE_8 | ( xMask & x );
sprites[0].attribute2 = 52;
```

Now if we compile and run our game, it runs smoothly and without totally strange behavior. Our final main method looks like this:

```
int main () {
    SetMode ( MODE_1 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE );
    InitializeSprites ();
    // Position variables and their bit - masks .
    // You can use the same masks for all sprites
    s16 x = 24;
    s16 y = 16;
    u16 xMask = 0x01FF ;
    u16 yMask = 0x00FF ;
    // The game loop
    while (1){
        // Changes the values of x and y according to
        // input from the D - Pad
        if ( keyDown ( KEY_LEFT )) x--;
        if ( keyDown ( KEY_RIGHT )) x++;
        if ( keyDown ( KEY_UP )) y--;
        if ( keyDown ( KEY_DOWN )) y++;
        // Applies the updated x and y values to the sprite
        sprites[0].attribute0 = COLOR_256 | SQUARE | ( yMask & y );
        sprites[0].attribute1 = SIZE_8 | ( xMask & x );
        sprites[0].attribute2 = 52;
    }
}
```

And with that, you should be able to create a simple game of Pong.