The PERN Project

The Nintendo Reverse Engineering Project

Day 3

Things covered in day 3 will be:
- What is a sprite?
- Memory and sprites.
- Displaying your first sprite.
- Multiple sprites
- 16 Color sprites
- Sprites in bitmap Modes (yes they are different)

What is a sprite?

What exactly do small, fast moving, flying faeries have to do with computer graphics? The term sprite has been around for a long time and although I have no idea who coined the term it has certainly stuck.

Sprites generally refer to any object that moves freely from the background. An obvious example is the Mario character in Super Mario Bros. Sprites can be any size often ranging from a few pixels to half the screen. They also can have any number of frames of animation. Be it a small, fast moving bullet or an imposing Street Fighter II character, sprites are what make games fun.

The most common way of rendering sprites is very similar to the way we drew a picture in day two. Each frame of the sprite is stored in a bitmap in off screen memory and is drawn to the screen pixel by pixel every frame. There are several problems with this. First, if you do not take precautions the background is destroyed. You must preserve it, either by redrawing the background every frame or storing the contents of the background that you are going to overwrite in a bitmap in memory, then, when the sprite has moved, replace the background you just copied over. The real problem is not this method's complexity, but its speed.
Fortunately for us the GBA has hardware rendered sprites freeing us from tying up the processor with bit-blitting code. Not to mention that the hardware supports such features as rotation, scaling, and alpha blending.

In order to render a sprite all you need to do is provide the graphics for the image and supply the GBA with a few pieces of information and you will have little Links and Marios running all over the place.

Memory and sprites

There are two areas of concern when it comes to sprite memory. The first is called OAM (Object Attribute Memory). The second is character memory. OAM is where the characteristics of each sprite are defined. You can have up to 128 sprites defined in OAM at any one time. OAM is arranged as a linear array starting at memory location 0x07000000. There are 128 entries in this array each consisting of four 16-bit attributes for a total of 1024 bytes of data.

The first 3 attributes deal with sprite attributes like position, size, shape, color depth, etc. The last attribute is for rotation data but is not actually part of that sprite (that probably makes no sense at this point but that is ok, I will cover in more detail in a moment). The most common way to define OAM is with a structure and most people I have seen do it like this:

```
typedef struct tagSprite
{
    u16 attribute0;
    u16 attribute1;
    u16 attribute2;
    u16 attribute3;
}OAM_Entry;

OAM_Entry OAM_Copy[128];
```

This allows you to access your copy of OAM with an array and then once each frame copy it to the real OAM. There are two reasons for using a copy. First OAM is locked when the screen is actually being drawn, meaning you can only access it during the Vblank and Hblank (and the Hblank only under certain circumstances). The second reason is that if you update OAM while your screen is not finished, you may actually update it when your sprite is half drawn. This will usually cause undesired effects like tearing (top half and bottom half don't exactly line up).

I actually use a slightly different structure for my OAM because attribute 3 really has nothing to do with that sprite. My OAM looks something like this.

```
typedef struct
{
    u16 attribute[3];
    u16 filler;
}OAM_Entry;

typedef struct
{
    u16 filler1[3];
    u16 pa;
    u16 filler2[3];
    u16 pb;
    u16 filler3[3];
    u16 pc;
    u16 filler4[3];
    u16 pd;
}RotData;

OAM_Entry OAMCopy[128];
RotData* rotData = (RotData*) OAMCopy;
```

Both my rotData array and my OAMCopy array point to the same location in memory causing them to overlap. This has two major benefits. One, it allows me to access the rotation attributes independently from the sprite attributes without the extra memory for another array. Two, it allows me to just copy the OAMCopy array into OAM every frame. Because attribute three is automatically updated when I access the rotData array, this copy takes care of both rotation attributes and sprite attributes. Below is a table that represents the layout of the OAM array. This is similar to the unioning the two sets of data but I feel this method is much more readable.

OAMCopy[]	**rotData[0]**
OAMCopy[0].attribute[0]	filler1[0]
OAMCopy[0].attribute[1]	filler1[1]
OAMCopy[0].attribute[2]	filler1[2]
OAMCopy[0].filler	pa
OAMCopy[1].attribute[0]	filler2[0]
OAMCopy[1].attribute[1]	filler2[1]
OAMCopy[1].attribute[2]	filler2[2]
OAMCopy[1].filler	pb
OAMCopy[2].attribute[0]	filler3[0]
OAMCopy[2].attribute[1]	filler3[1]
OAMCopy[2].attribute[2]	filler3[2]
OAMCopy[2].filler	pc
OAMCopy[3].attribute[0]	filler4[0]
OAMCopy[3].attribute[1]	filler4[1]
OAMCopy[3].attribute[2]	filler4[2]
OAMCopy[3].filler	pd

You may be wondering what the hell this pa, pb, pc, pd thing is. This is covered in the rotation portion of DAY 9. One thing you should notice though is that my rotData structure is 4 times as large as the OAMEntry structure. If you ignore the fillers the rotData structure accesses the attribute3 (the filler) of each OAMEntry structure when the arrays are pointed to the same area of memory. The pa, pb, pc, pd variable are the rotation attributes. There are a total of 32 sets of 4 rotation attributes in OAM. There can be 32 different rotation parameters that can be applied to any number of sprites. All the sprites can use the same rotation attributes or you can have 32 independently rotating and scaling sprites. Which sprites go with which attributes? Well, that is up to you. Part of one of the attributes in each sprite tells the GBA which set of rotData to use. Because the 32 rotation attributes don't correspond to any one OAM, we split it into two arrays even though the arrays access the same physical memory.

Let us look at the OAM attributes and see if we can discern how to display sprites.

bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Attribute[0]	Shape		C	M	Mode		SD	R	Y Coordinate							

Bits 0-7 form an 8bit signed integer (range = [-128, 127]) denoting the top of your sprite. ANDing your sprite's top with 255 to wrap around the screen. For example, say that the top is int y = -1 (one pixel above the screen). In hex, this is 0xFFFFFFFF; -1 & 255 = 0xff, which is indeed -1 for signed 8bit integers.
Bit 8 is the rotation scaling flag. If set it will draw the sprite with the scaling/rotation parameters specified.
Bit 9 is something called the size double flag and I will talk about this when I

describe sprite rotation. One interesting effect of the SD flag is that if it is set, but the rotation flag is not, then that sprite will not be displayed. This is how we will turn off all of the unused sprites and why this flag is sometimes referred to as the TURN_OFF flag.
Bits A-B are the mode flags and deal with alpha blending. I will talk about that more in chapter 9.
Bit C is the mosaic flag. When set the sprite will have the mosaic value applied to it. This topic is also saved for day 9.
Bit D is the color mode. If set it defines the sprite as 256-colors. If cleared it uses a 16-color palette.
Bit E-F is the object's shape and will be described in more detail when we actually draw our first sprite.

bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Attribute[1]	Size		VF	HF	RotData index			X Coordinate								

Bits 0-8 form the left of the sprite. Similar to the top, these 9 bits form a signed integer, with the range [-256, 255]. This time, AND with 511 for correct wrapping.
Bits 9-13 serve a dual purpose. If the rotation scaling flag is set in attribute[0] then they are the 5-bit index (0-31) into the rotData array and define which set of rot/scaling parameters are going to be used with that sprite. If the rotation scaling flag is not set then bits 9-11 are not used.
Bit 12 is the horizontal flip flag (the sprite will be flipped along the y axis) and **bit 13** is the vertical flip flag (x axis). These attributes are not used when the rotation flag is set in attribute[0].
Bit 14-15 is the size and I will give you a nice little table that explains the use of this flag and the shape flag when we draw our first sprite.

bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Attribute[2]	Palette number				Priority		Character Name									

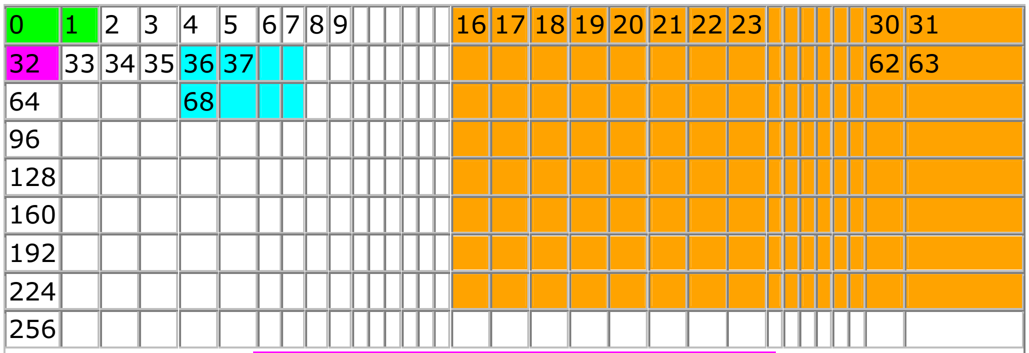
Bit 0-9 is the index into sprite character memory of the first 8x8 tile of the sprite (to fully understand this flag wait until I describe the second area of sprite memory).
Bit 10-11 is the priority. As with backgrounds sprites can be assigned a priority of 0-3. Higher numbered sprites are drawn first, meaning that a sprite of priority 0 will be drawn over the top of a sprite with priority of 3. Also, a sprite priority is always higher than that of the corresponding background priority meaning that if both a sprite and a background have the same priority then the sprite will be drawn on top. One more thing: for sprites with the same priority then sprites with the lowest OAM number are drawn on top. **Bits 12-15** are the palette number. If your sprite is a 16-color sprite then this value will determine which of the 16 16-color palettes are used. If it is a 256-color sprite then these bits are ignored.

Well that pretty much sums up OAM memory, except it does nothing for rotation attributes; that will come later when I talk about rotation and scaling. The next area is character memory and this is where your sprite character data is stored (the bitmaps of your sprite).

Character data starts at 0x06010000 and extends for 32KB. It consists of 1024 8x8 16 color tiles (256 color tiles actually take two 8x8 slots). The character name bits in attribute[2] refer to one of these tiles. This makes loading graphics for sprites that are larger than 8x8 awkward. Fortunately, I have written a tool that will cut your sprite bitmap into 8x8 tiles that can be loaded much easier into sprite memory.

The problem with this set up is that bitmap modes (modes 3, 4, and 5) actually extend into the character data area of memory cutting it in half. This means that only character names greater than 511 are allowed in the bitmap modes.
The tiles can be arranged in one of two user definable ways: 1D (1 dimensional) and 2D. If the 1D flag in REG_DISPCNT is set then the layout is 1D, otherwise it is 2D. First I will explain 2D mode.

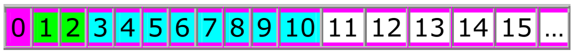
2D mode is laid out as a big square of 8x8 tiles. The table is 32 tiles wide and 32 tall. The Character Name in attribute[2] points to one of these tiles. Here is a picture that may help.



8x8 16-color sprite; Character name= 32
8x8 256-color sprite; Character name = 0
16x16 256-color sprite; Character name = 36
64x64 256-color sprite; Character name = 16

To compute the actual memory address of a character the following equation will work. 0x06010000 + (CharName * 8x4) = addressOfChar. 0x06010000 is the start address of character memory and each character is 32 bytes due to the fact that they are 16-color by default. Storing of the graphics into the character memory must be done one tile at a time which is why a tool to strip the bitmaps into a strip of tiles is necessary.

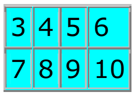
The next mode is just a big stack of tiles that are only one wide and 1024 tall (512 if 256 color). This mode may seem less intuitive, but is actually the most common. In fact it is easier. The major benefit to this mode comes into play when you consider the work necessary to copy a sprite from your ROM to video memory. In 2D mode you have to copy your bitmap tile by tile into video memory keeping track of the width of the video memory (32 tiles). This adds a separate step that can be a significant slowdown when you need to change out sprite graphics often. With 1D mode you can copy the data straight into video memory, assuming your sprite is already broken down. This enables the easy use of DMA (direct memory access is a very fast memory copy done by hardware and will be discussed later) to copy sprite data into memory. I have found no real use for 2D mode and do not use it myself. I am sure there is a reason, but I do not know what that could be. Here is a picture of 1D mode:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...

8x8 16-color sprite; Character name = 0
8x8 256-color sprite; Character name = 1
16x16 256-color sprite; Character name = 3

The 16x16 256-color sprite is arranged like this:



3	4	5	6
7	8	9	10

This means that a tile by tile copy of the sprite graphics into character memory is not interrupted by the checking for the width of character memory. In 2D you would have to load the first 4 tiles into memory and then recomputed the offset into character memory for the next row of tiles, but 1D always wraps around.
Well that about sums up the memory and formats used by sprites; now I suppose you want to draw a few.

Drawing your first sprite
First let's do a 1D demo and then a 2D one. Before we do anything, we need to define all these bits we just talked about so we can set our attributes in some readable fashion. You will find these defines in "gba_video.h"

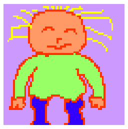
```
//sprite defines
//Atribute0 stuff
#define ROTATION_FLAG       BIT8
#define SIZE_DOUBLE         BIT9
#define SPRITE_OFF          BIT9
#define MODE_NORMAL         0
#define MODE_TRANSPARENT    BIT(0xA)
#define MODE_WINDOWED       BIT(0xB)
#define MOSAIC              BIT(0xC)
#define COLOR_16            0
#define COLOR_256           BIT(0xD)
#define SQUARE              0
#define TALL                BIT(0xF)
#define WIDE                BIT(0xE)

//Atribute1 stuff
#define ROTDATA(n)           ((n) << 9)
#define HORIZONTAL_FLIP     BIT(0xC)
#define VERTICAL_FLIP       BIT(0xD)
#define SIZE_8              0
#define SIZE_16             BIT(0xE)
#define SIZE_32             BIT(0xF)
#define SIZE_64             BIT(0xF) | BIT(0xE)

//atribute2 stuff
#define PRIORITY(n)          ((n) << 10)
#define PALETTE(n)           ((n) << 12)
```

These defines should be self explanatory; if not, refer to the descriptions of the attributes above. To clarify the relation between the size and shape flags here is a handy table.

	SIZE_8	SIZE_16	SIZE_32	SIZE_64
SQUARE	8x8	16x16	32x32	64x64
TALL	8x16	8x32	16x32	32x64
WIDE	16x8	32x8	32x16	64x32

DEMO 1

The first thing we must do is create some graphics for our sprite. To do that, we are again going to use a PCX file, but this time we are going to let the PC process our data. You will need PCX2Sprite if you have not already downloaded it. This file takes as an input a PCX file and produces an object file, a raw binary, or a c file that contains the sprite graphics chopped up to fit into sprite memory. It also produces a header file for easy access to the data. Open your favorite graphics editor and create a 64x64 image. Ensure the palette is 256 colors and then save as sprite.pcx. Here is my image:



(yes, I drew it myself)
(no, it is not a self portrait)

Color zero, no matter its RGB value, will not be rendered onto the GBA screen so use this to color the portions of your image that you do not want blitted. PCX2Sprite.exe takes several parameters. For a single sprite with one frame of animation just use: pcx2Sprite input.pcx. An object file will be produced with a header file for easy access. Look in the readme to see the other options.

Let us delve into the world of sprites with our first single sprite demo. This demo places a single 256-color 64x64 sprite on the screen and allows you to move it about with your direction pad. It also enables or disables the vertical and horizontal flip flags with the A and B buttons. As usual here is the full source to the demo after which we will step through the code.

```
#include "gba.h"
#include "sprite.h"

//global data
OAMEntry OAMCopy[128];

//a simple sprite structure
typedef struct
{
   int x,y;
   OAMEntry* oam;
   int gfxID;
}Sprite;

//from day 2, our wait for Vblank
void WaitForVblank(void)
{
   while(! (REG_DISPSTAT & DISPSTAT_VB));
   while(REG_DISPSTAT & DISPSTAT_VB);
}

//turn all the sprites off
```

```
void InitOAM(void)
{
   int i;
   for(i = 0; i < 128; i++)
      OAMCopy[i].attribute[0] = SPRITE_OFF;
}

// Copy our OAMcopy to OAM
void UpdateOAM(void)
{
   int i;
   for(i = 0; i < 128 * sizeof(OAMEntry) / 4 ; i++)
      ((u32*)OAMMem)[i] = ((u32*)OAMCopy)[i];
}

// Test for key presses and move the sprite
void GetInput(Sprite* sp)
{
   if(!(REG_KEYS & KEY_UP))
      sp->y--;

   if(!(REG_KEYS & KEY_DOWN))
      sp->y++;

   if(!(REG_KEYS & KEY_LEFT))
      sp->x--;

   if(!(REG_KEYS & KEY_RIGHT))
      sp->x++;

   if(!(REG_KEYS & KEY_A))
      sp->oam->attribute[1] ^= HORIZONTAL_FLIP;

   if(!(REG_KEYS & KEY_B))
      sp->oam->attribute[1] ^= VERTICAL_FLIP;
}

// Update the sprites OAM entry to reflect new position
void MoveSprite(Sprite* sp)
{
   sp->oam->attribute[1] &= 0xFE00;
   sp->oam->attribute[1] |= (sp->x & 0x01FF);

   sp->oam->attribute[0] &= 0xFF00;
   sp->oam->attribute[0] |= (sp->y & 0x00FF);
}

// main entry point
int main(void)
{
   Sprite sprite1;
   int i;

   SetMode(MODE_0 | OBJ_ENABLE | OBJ_MAP_1D);

   InitOAM();
   //init our sprite
   sprite1.oam = &OAMCopy[0];
   sprite1.x = 10;
   sprite1.y = 10;
   sprite1.gfxID = 0;
   //set up our sprites OAM entry attributes
   sprite1.oam->attribute[0] = COLOR_256 | SQUARE;
   sprite1.oam->attribute[1] = SIZE_64;
   sprite1.oam->attribute[2] = sprite1.gfxID;

   //copy the sprite grahics in obj graphics mem
   for(i = 0; i < 64 * 64 / 2; i++)
      OBJ_GFXMem[sprite1.gfxID * 16 + i] = spriteData[i];

   //copy in the palette
   for(i = 0; i < 256; i++)
      OBJPaletteMem[i] = spritePalette[i];

   //main loop
   while(1)
   {
      GetInput(&sprite1);
      MoveSprite(&sprite1);
      WaitForVblank();
      UpdateOAM();
   }
 }
```

Let us step through the code.

```
//global data
OAMEntry OAMCopy[128];

//a simple sprite structure
typedef struct
{
   int x,y;
   OAMEntry* oam;
   int gfxID;
}Sprite;
```

First we declare our OAM copy that holds our mirror of the OAM entries. Again the reason this is necessary is because there are times when OAM attribute memory is locked and this allows us to write attributes anytime without worrying about the state of the screen update.

Next we define a simple sprite structure that holds all the data our sprite will need. An x and a y value and a pointer to its OAM entry. Also we have an ID that will point to its graphics (the ID is just a representation of the offset into sprite graphics memory of the first tile of the sprite).

```
//turn all the sprites off
void InitOAM(void)
{
   int i;

   for(i = 0; i < 128; i++)
      OAMCopy[i].attribute[0] = SPRITE_OFF;
}
```

```
// Copy our OAMcopy to OAM
void UpdateOAM(void)
{
   int i;

   for(i = 0; i < 128 * sizeof(OAMEntry) / 4 ; i++)
      ((u32*)OAMMem)[i] = ((u32*)OAMCopy)[i];
}
```

These two functions initialize and update our sprites. The first just sets all the OAM entries to "off" so that only the ones we turn on will be displayed. If this is not done then all of the unused entries will be active and end up showing up in the upper left hand corner of the screen as a random 8x8 box of crap.

The second function copies our OAM copy to actual OAM memory so our sprites will be updated. It is cast to a 32 bit pointer so we can copy 4 bytes at a time. We will convert this to a DMA call in a later chapter as well as place the copy into internal working ram to speed things up.

Next is the getInput() function which modifies a sprite based on user input. The only interesting thing it does is clear and set the flip flags to cause our sprite to be rendered upside down or sideways depending on key press.

```
// main entry point
int main(void)
{
   Sprite sprite1;
   int i;

   SetMode(MODE_0 | OBJ_ENABLE | OBJ_MAP_1D);

   InitOAM();

   //init our sprite
   sprite1.oam = &OAMCopy[0];
   sprite1.x = 10;
   sprite1.y = 10;
   sprite1.gfxID = 0;

   //set up our sprites OAM entry attributes
   sprite1.oam->attribute[0] = COLOR_256 | SQUARE;
   sprite1.oam->attribute[1] = SIZE_64;
   sprite1.oam->attribute[2] = sprite1.gfxID;
 }
```

The code for main is pretty straightforward. We set our mode and enable the sprites and set the mapping mode to 1D. We then initialize our sprite by assigning its OAM and its position, and let it know we are going to use the beginning of sprite graphics memory to place our sprite graphics (gfxID = 0). Finally, we set the attributes for the OAM to create a 64x64 256-color sprite. Notice that we assign attribute[2] the value for the beginning of sprite graphics; this location is often referred to as the sprite (or object) character name.

```
//copy the sprite grahics in obj graphics mem
for(i = 0; i < 64 * 64 / 2; i++)
   OBJ_GFXMem[sprite1.gfxID * 16 + i] = spriteData[i];

//copy in the palette
for(i = 0; i < 256; i++)
   OBJPaletteMem[i] = spritePalette[i];
```

The last thing we do is copy in our sprite graphics and palette. OBJ_GFXMem is defined in gba.h as 0x06010000. We then offset that address the correct number of sprite characters by multiplying the starting character by 16 (each character is based on an 8x8 16 color character, this is 64 pixels at 4 bits apiece for 32 bytes, or 16 shorts which is the type of our arrays). In our case the gfxID is 0 so this is not needed but when we do more sprites this will become very important.

The palette copy is straightforward enough that it deserves little comment. Remember, we imported the sprite data from a PCX file that was converted with PCX2sprite to an object file that we could link right in. The last part of main sticks us in an infinite loop where we grab input and update the sprite.

Demo 2

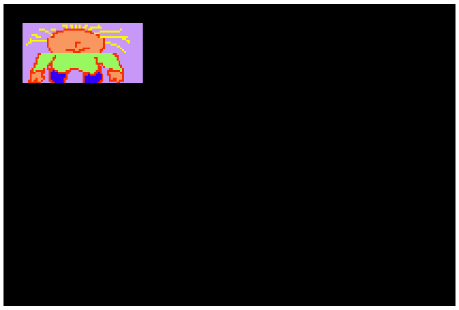
Demo 2 simply changes the mapping mode to 2D so we can see the difference. The only change will be the way in which we copy in the sprite data.
The first thing we must change is this:

```
SetMode(MODE_0 | OBJ_ENABLE | OBJ_MAP_1D);
```

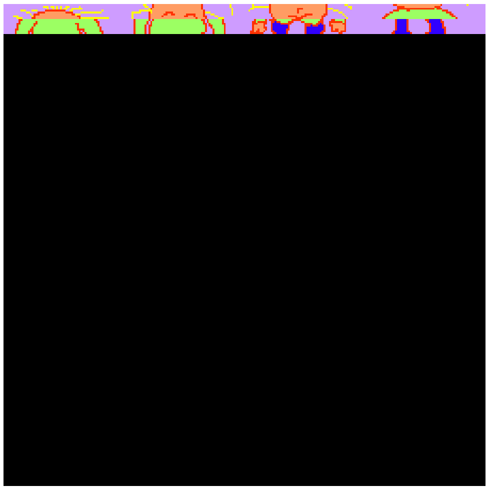
To

```
SetMode(MODE_0 | OBJ_ENABLE | OBJ_MAP_2D);
```

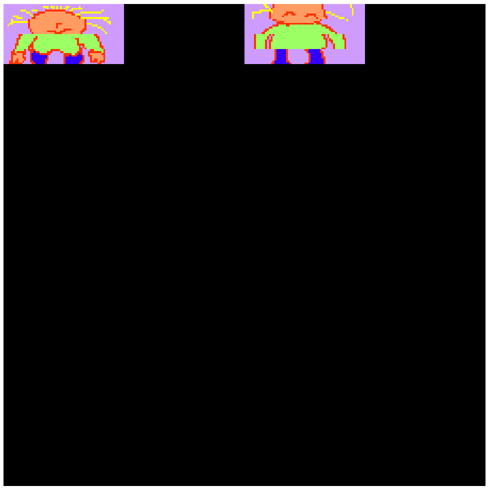
Let us see what this change alone gives us:



Hmm … Not exactly what we want is it. If you recall, we must copy the data in tile row by tile row in order for 2D to work. The reason for this is that when we just copy in all our data we get tiles that look like this in tile memory:



This is a screen shot using Mappy VM's tile viewer that lets us look inside sprite graphics memory. For 1D mode, this *is* the way we want our tiles loaded in, but not for 2D. This is how we want our tiles in 2D mode:



Unfortunately, Mappy VM's sprite tile viewer is set up for 16 color mode. Sprite graphics memory is 32 tiles wide and 32 tall, making it 256 pixels wide in 16 color mode. In 256 color mode it should only be 128 pixels wide but the video viewer still wraps it at 256 pixels so we do not get to see the nice 2D layout that is actually occurring. So, how do we get our tiles in memory this way? Simple: we just copy in the first 8 tiles then move down a full 32 tiles, then copy the next 8, and so on till we have all 64. Here is some sample code that does just that. Besides the change to SetMode, this is the only difference between demo 1 and demo 2.

```
offset = sprite1.gfxID * 8 * 4 / 2;

//copy the sprite grahics in obj graphics mem
for(iy = 0; iy < 8; iy++) {
   for(ix = 0; ix < 8 * 8 * 8 / 2; ix++)    {
      OBJ_GFXMem[offset + ix] = spriteData[ix + iy * 8*8*8 / 2];
   }
   offset += 32 * 8 * 4 / 2;
}
```

First we calculate the offset into gfxMem by multiplying the sprite ID by the size of a tile (8 x 4, since the tiles are based on 16-color), then divide by two because our arrays are 16 bit. We then loop through all the tiles and copy in 8 at a time. Every row, we increment our offset by the size of 32 8x8 tiles.
Now let us add multiple sprites.

DEMO 3 - Multiple Sprites

Let us place a 50 or so sprites on the screen with this demo. First they will all use the same graphics. In the next demo we will make them use different graphics.
The first thing we are going to need is some way to send these guys off at different speeds because I certainly don't want to move 50 sprites around with the keyboard. It would be nice if we had a random number generator. Let's make one.

The following is based on the Mersenne Twister algorithm.

```
unsigned short r256table[256];
unsigned char r256index;

void r256init(void)
{
   int i,j,msb;
   j=42424;
   for(i=0;i<256;i++)
   {
      r256table[i]=(unsigned short)(j=j*65539);
   }

   msb=0x8000;

   j=0;

   for(i=0;i<16;i++)
   {
      j=i*5+3;
      r256table[j]|=msb;
      r256table[j+1]&=~msb;
      msb>>=1;
   }
}

unsigned short r256(void)
{
   int r;
   r=r256table[(r256index+103)&255]^r256table[r256index];
   r256table[r256index++]=r;
   return r;
}
```

For details on how and why it works, be sure and follow the link provided. Since we have no system timer to seed the function with we are not going to worry about the fact that is going to give the same sequence of random numbers every time. Later we will initialize it and then call it repeatedly while waiting for the user to get past the title screen. This way it should be random enough by the time it gets used.

Now that we have a means let us get on to the ends. Here is the changed code. We added the random functions which I am not going to re-paste. We also changed our sprite struct to hold speed.

```
typedef int FIXED32;
//a simple sprite structure
typedef struct
{
  FIXED32 x,y;
  OAMEntry* oam;
  int gfxID;
  FIXED32 dx,dy;
}Sprite;
```

dx and dy now will track our speed. These will be fixed 24.8 values for now and in order to use them on x and y it would be best if x and y were also fixed. For a fixed point review see appendix A.

The next altered function will be MoveSprite. This needs to be fixed to handle the fixed point x and y and it does that with a simple shift of x and y back to ints.

```
// Update the sprites OAM entry to reflect new position
void MoveSprite(Sprite* sp)
{
   sp->oam->attribute[1] &= 0xFE00;
   sp->oam->attribute[1] |= ((sp->x >> 8) & 0x01FF);

   sp->oam->attribute[0] &= 0xFF00;
   sp->oam->attribute[0] |= ((sp->y >> 8 ) & 0x00FF);
}
```

I hope this code is clear and if not please refer to the appendix. Finally, we get to main and are able to alter our code to handle lots of sprites.

```
#define MAX_SPRITES 50

int main(void) // main entry point
{
   Sprite sprites[MAX_SPRITES];
   int i;
   int offset;

   SetMode(MODE_0 | OBJ_ENABLE | OBJ_MAP_1D);

   InitOAM();
   r256init();

   //init our sprite
   for(i = 0; i < MAX_SPRITES; i++)
   {
      sprites[i].x = 10 << 8;
      sprites[i].y = 10 << 8;
```

```
     sprites[i].dx = r256() >> 7;
     sprites[i].dy = r256() >> 7;

     sprites[i].gfxID = 0;

     //set up our sprites OAM entry attributes
     sprites[i].oam = &OAMCopy[i];

     sprites[i].oam->attribute[0] = COLOR_256 | SQUARE;
     sprites[i].oam->attribute[1] = SIZE_64;
     sprites[i].oam->attribute[2] = sprites[i].gfxID;
  }

  offset = sprites[0].gfxID * 8 * 4 / 2;

  //copy the sprite grahics in obj graphics mem
  for(i = 0; i < 8 * 8 * 64 / 2; i++)
  {
     OBJ_GFXMem[offset + i] = spriteData[i];
  }

  //copy in the palette
  for(i = 0; i < 256; i++)
     OBJPaletteMem[i] = spritePalette[i];

  //main loop
  while(1)
  {
     for(i = 0; i < MAX_SPRITES; i++)
     {
        sprites[i].x += sprites[i].dx;
        sprites[i].y += sprites[i].dy;

        MoveSprite(&sprites[i]);
     }

     WaitForVblank();
     UpdateOAM();
  }
}
```

First you will notice that we declare an array of sprites and initialize them with a for loop. We assign x and y a value of 10 (in fixed point) and then assign a random speed to the dx and dy. These values are shifted down by 7 so they will be no more than 9 bits. Since they are in the form 24.8, 8 bits will be fraction giving them a speed between 0 and 2 pixels per frame in either direction.

You should also note that all sprites are assigned the same gfxID making them all use the same graphics. The next demo will assign different graphics to all the sprites.

Finally we get to the main loop. The call to getInput has been replaced by a for loop that adds the speed to x and y and then moves the sprite. We don’t need to worry about going off screen because that is handled in the MoveSprite function.

DEMO 4

And now we do the same, but with unique graphics for all 128 sprites. You will notice that, since there is 32KB of sprite graphics memory and 128 OAM entries, we can fit exactly 128 16x16 sprites in memory. (128 * 16 * 16 = 32KB). If we were to use 16 color sprites instead we could fit 128 32x16 or 16x32 sprites. When we cover advanced sprite topics, we will put this relation to good use by designing memory management that takes advantage of same-sized sprites.

First I need some sprite graphics. I open up paint shop and create an image that is 240x160 and enable my grid view at 16x16 so I can see where I am drawing. I then fill up the spaces for 128 sprites, which is nearly a full screen of graphics data. This is the image I created.

My only goal was to make certain that each 16x16 block had some graphics in it. Now I convert it using pcx2sprite with the following options:

```
Pcx2sprite data/sprite.pcx –h:16 –w:16
```

This lets the converter now the size of my sprites so it groups their tiles together appropriately. Now let's check the changes we must make to the source in order for this to work like we want. The only change is to main().

```
#define MAX_SPRITES 128
// main entry point
int main(void)
{
   Sprite sprites[MAX_SPRITES];
   int i;
   int offset;

   SetMode(MODE_0 | OBJ_ENABLE | OBJ_MAP_1D);

   InitOAM();
   r256init();

   //init our sprite
   for(i = 0; i < MAX_SPRITES; i++)
   {
      sprites[i].x = ((i % 15) * 16) << 8;
      sprites[i].y = ((i / 15) * 16) << 8;

      sprites[i].dx = r256() >> 7;
      sprites[i].dy = r256() >> 7;

      sprites[i].gfxID = i * 8;

      //set up our sprites OAM entry attributes
      sprites[i].oam = &OAMCopy[i];
      sprites[i].oam->attribute[0] = COLOR_256 | SQUARE;
      sprites[i].oam->attribute[1] = SIZE_16;
      sprites[i].oam->attribute[2] = sprites[i].gfxID;

      MoveSprite(&sprites[i]);
   }

   offset = sprites[0].gfxID * 8 * 4 / 2;

   //copy the sprite graphics in obj graphics mem
   for(i = 0; i < 16 * 16 * 128 / 2; i++)
   {
      OBJ_GFXMem[offset + i] = spriteData[i];
   }

   //copy in the palette
   for(i = 0; i < 256; i++)
      OBJPaletteMem[i] = spritePalette[i];

   //main loop
   while(1)
   {
      if(!(REG_KEYS & KEY_A))
      {
         for(i = 0; i < MAX_SPRITES; i++)
         {
            sprites[i].x += sprites[i].dx;
            sprites[i].y += sprites[i].dy;

            MoveSprite(&sprites[i]);
         }
      }
      WaitForVblank();
      UpdateOAM();
   }
}
```

The first thing you should notice is that all 128 sprites are enabled. You can edit demo 3 and display all 128 sprites as well, but keep in mind there are hardware limits to the amount of sprites per scan line that can be rendered. These limits are an advanced topic and will be covered on day 9.

We then initialize the sprites as before with the following differences: we set x and y so they fill up the screen in a nice pattern and recreate our original image. Also, we want each sprite to use different graphics this time.

All we have to do is point the OAM attribute[2] to the correct place. Since each sprite is exactly 4 characters from the last (16x16 needs 4 tiles since tiles are 8x8) and each character counts as 2 tiles in 256-color mode we simply multiply the sprite index by 8. This is done in the following line:

```
      sprites[i].gfxID = i * 8;
```

If the sprites were instead 8x8 then they would only take up 2 characters in 256-color mode and the line could be modified to:

```
      sprites[i].gfxID = i * 2;
```

Another change is made to ensure our sprite is 16x16 by making the shape square and the size 16.

Since we have 128 sprites to copy in this time instead of 1, our for loop gets a slight modification to allow for more data (for(i = 0; i < 128 * 16 * 16 / 2; i++)).

Finally, we come to the main loop which is nearly identical to before. The only change I made was that the sprites will not move unless you press the A key. This way you can see them in their nice, orderly rows that I took the time to set up with:

```
for(i = 0; i < MAX_SPRITES; i++)
   {
      sprites[i].x = ((i % 15) * 16) << 8;
      sprites[i].y = ((i / 15) * 16) << 8;
```

That is it for simple 256-color sprites. Let's do one more demo for 16-color sprites and call it a day.

DEMO 5 - 16-Color Sprite

Really the major change necessary for this demo will be in graphics creation and conversion. I actually had to sit down and add support to my converter for 16-color sprites and that took longer than changing the demo.

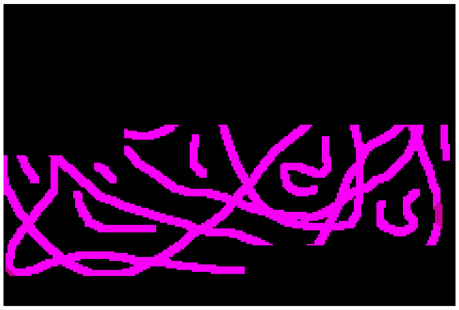
```
Pcx2sprite data/sprite.pcx –h:16 –w:16 –16
```

The only change to the code is the init of the sprites to 16-color and the change of gfxID index to i * 4 instead of i * 8 since the 16x16 sprites use half the tiles in 16-color mode. Here is the init code; no explanation follows because it is identical to the operation of the above code.

One thing of note is that in order for my converter to process 16 color tiles only colors from the first 16 palette entries can be used. There is no palette quantizing or any such thing in my simple converter. This is a limitation that I may fix one of these days. All changes from the 256 color demo are highlighted in red.

```
#define MAX_SPRITES 128
// main entry point
int main(void)
{
   Sprite sprites[MAX_SPRITES];
   int i;
   int offset;

   SetMode(MODE_0 | OBJ_ENABLE | OBJ_MAP_1D);
   InitOAM();
   r256init();

   //init our sprite
   for(i = 0; i < MAX_SPRITES; i++)
   {
      sprites[i].x = ((i % 15) * 16) << 8;
      sprites[i].y = ((i / 15) * 16) << 8;
      sprites[i].dx = r256() >> 7;
      sprites[i].dy = r256() >> 7;
      sprites[i].gfxID = i * 4;

    //set up our sprites OAM entry attributes
      sprites[i].oam = &OAMCopy[i];
      sprites[i].oam->attribute[0] = COLOR_16 | SQUARE;
      sprites[i].oam->attribute[1] = SIZE_16;
      sprites[i].oam->attribute[2] = sprites[i].gfxID;

      MoveSprite(&sprites[i]);
   }

   offset = sprites[0].gfxID * 8 * 4 / 2;

   //copy the sprite graphics in obj graphics mem
   for(i = 0; i < 16 * 16 * 128 / 4; i++)
   {
      OBJ_GFXMem[offset + i] = spriteData[i];
   }

   //copy in the palette
   for(i = 0; i < 16; i++)

    OBJPaletteMem[i] = spritePalette[i];

 //main loop
 while(1)
 {
    if(!(REG_KEYS & KEY_A))
    {
       for(i = 0; i < MAX_SPRITES; i++)
       {
          sprites[i].x += sprites[i].dx;
          sprites[i].y += sprites[i].dy;

          MoveSprite(&sprites[i]);
       }
    }
    WaitForVblank();
    UpdateOAM();
 }
 }
```

Sprites in Bit Map Modes (mode 3, 4, and 5)
For an interesting test, change the call to setmode in demo 4 so that it places the GBA into one of the 3 bit map modes. If you do this, this is what you will see:



What happened to the firs half of our sprites?! Well, it turns out that bitmap modes are greedy, and in order to fit all their data into video memory they need to borrow from sprite graphics memory. How much do they take? Exactly half. This means that any sprite gfx ID you use that is below 512 will now be in regular video memory and hence not be rendered by the GBA. Your sprites are still there, but the GBA can't find any graphics for them, so they are blank.

In bitmapped modes, ensure that your character name (or sprite ID or Object name what ever you choose to call it) does not reference the first 512 tiles of sprite graphics memory as these will be unavailable to you. Instead, begin ID assigning at 512.

That is all we have time for today. We will revisit sprites and cover the more advanced topics on Day 9. Those topics will include: Rotation, Scaling, Animation, Mosaic, memory management, and exceeding the 128 sprite limit.