SNES PPU Tutorial

Written by *Squall (aka Squall_FF8)*

Version 1.0, from 20.12.2017

This tutorial emerged as an attempt to explain to fellow FinalFantasy modder the basics of SNES (Super Nintendo) PPU (Picture Processing Unit). I will try to give a definition to each of basic elements (abstractions) used to produce an image on SNES, to give short explanation and examples that will illustrate the subject. I will try not to dive too deep, yet mention the specifics needed for real application (mod).

OK forget about all technical details that you know or don't know and start from a scratch:

I. Mode

Modern or old all graphic system use a mode in which to display whatever we see on the screen. Some cards have only 1 mode, but most, more than 1. Think of SNES PPU as a graphic card that you plug in your computer.

1. So, what is a video mode?

It's a sum of number of characteristics:

- Resolution: usually defined in pixels - how long and wide is our screen (in pixels)
- Max number of colors that could be simultaneously seen on the screen
- BPP: Bits per pixels - how many bits we need to define the color of the screen
- Video Memory: the mode dictate how to interpret the memory in order to compose the screen
- many other things like layers, buffers, graphical objects (like Sprites), ...

2. Types of Modes:

2.1. Depend on how video memory is interpreted.

- **Bit-planes** - the video memory contains the color for each pixel. Modern cards use primary 1 plane of bits, which is very easy to ASM. Unfortunately, almost all old cards use more than 1 plane to organize bits for pixels. I'm sure there was a hardware need for that, but it's quite complicated to manage this in ASM. For now, I'm not going to delve in this, but will mention that SNES uses 4 planes.
- **Tile based** - the main video memory contains index of a tile rather raw pixels. That is pretty much as text mode in DOS/Linux - the video memory contains ASCII (or other pages) index of letters and there is a separate place that hold the definition of each letter. In SNES is the same but is called Tile.

2.2. Depending on how color is stored:

- **Direct Color** - each color contains the necessary bits for R/G/B components. In SNES its 5-5-5 - meaning 5 bits for each component
- **Palette based** - each color is an index into predefined palette of colors (defined in their absolute RGB value)

3 OK that was theory, what about some examples?

Mode is more abstract, so not many operation could be done:

- We can set desired mode (0..7). Usually this is one of the first things in the initialization code.
- Normal resolution is 256x240 (PAL) or 256x224 (NTSC) with 50/60Hz. We can't set the frequency, but we can read it. We can set V resolution.

```
$2105 (W): bit 2-0 BG Screen Mode
$2133 (W): bit 2 - BG V-Direction Display (0=224 Lines, 1=239 Lines) (for NTSC/PAL)
```

```
$213F (R): bit 4 - Frame Rate (0=NTSC/60Hz, 1=PAL/50Hz)
```
* In SNES Mode 0-6 are all Tile based, Mode 7 is bit-planes. All modes use palettes to designate the color.
** For specifics like resolution and BPP better check documents.

II. Layers

The second abstraction is Layer. You may see it as Background or BG in documentation. It works the same way as in specialized graphic applications: Photoshop or Gimp - the final image is composed by plotting couple of images 1 by 1, each on top of the next, each containing transparent pixels trough which you may see things from the image under it.

SNES contain 4 such layers (usually called background) + 1 sprite layer (which function in similar way) for a total of 5 layers. Unlike Photoshop where each layer can have 256 levels of transparency, here we have only 2 levels - fully transparent or not. Using color with index 0 mark transparent parts.

The other difference from Photoshop is that in each layer we can define priority for individual Tiles. For BG1-4 we have priority 0 and 1, while for Sprites (OBJ) we have 0-3. The order in which final image is rendered depend on the mode, so check the documentations for specifics (or experiment in vSNES). While usually the mode define things like BPP or Max Colors, in SNES each layer has its own definition and is determined by the mode. For example, in Mode 0 - all 4 BG are 2 BPP (4 color), but in Mode 1 - 4bpp for BG1,2 and 2bpp for BG3. Sprites (OBJ) are always in 4bpp, regardless of the Mode!

```
$212C - control which layer is visible or not
$2105 - tilesize for each leayer
$2107-$210A - set layer's base address in VRAM and its size. For now we all work with
32x32
```
Now that we know so many things about layers, what exactly we have in a layer?
Each layer is a two-dimensional array of 16 bit values. There are 4 possible dimensions, but for now we will talk about 32x32. Each value has this format (exception in Mode 7 or "Offset-per-Tile" modes: 2,4,6):
```
  Bit 0-9   - Tile ID     (000h-3FFh)
  Bit 10-12 - Palette ID  (0-7)
  Bit 13    - BG Priority (0=Lower, 1=Higher)
  Bit 14    - X-Flip      (0=Normal, 1=Mirror horizontally)
  Bit 15    - Y-Flip      (0=Normal, 1=Mirror vertically)
```

Clipping???
Scrolling????

III Tiles

We used that word so much, so I hope you got some intuitive understanding, but now is it a time to get a more formal one :D

1. What is a Tile?

Tile is the basic element to a tile engine. It is the atom of the chemistry so to speak. It is the brick of a wall.
A tile is a 2D MxN matrix (block/array) of points/pixels**. In SNES a tile is an 8x8 block of pixels!** The colors of each pixels could be an index in a palette or Direct Color (R/G/B values) and that is defined by the Mode/Layer.

2. Internal storage (or Bit-planes)

A tile is stored in the memory as a Bit-Planes. Depending on how many BPP is a layer, the tile uses the same number of Bit-planes. Since SNES can use only 2/4/8 BPP, the tiles use 2/4/8 Bit-planes. From hacking standpoint it’s important to know - how many bytes a Tile occupy:

- in 2bpp, we have 2 x 8x8 = 128 bits = 16 bytes = $10 bytes
- in 4bpp, we have 4 x 8x8 = 256 bits = 32 bytes = $20 bytes

3. Tile map

A Tile Map is a 2D matrix MxN where each value is a Tile ID. Wait a minute, that looks dangerously similar to the definition of a layer. YES, it is, a layer is an instance (custom case) of a tile map. And not just that, a Sprite is another instance of a TileMap.
Because of the way SNES works (Tiles don't hold palette info), the values of this 2D matrix (TileMap) are 16 bits, with the same meaning as layers:

```
Bit 0-9   - Tile ID      (000h-3FFh)
Bit 10-12 - Palette ID   (0-7)
Bit 13    - BG Priority  (0=Lower, 1=Higher)
Bit 14    - X-Flip       (0=Normal, 1=Mirror horizontally)
Bit 15    - Y-Flip       (0=Normal, 1=Mirror vertically)
```

4. Tile Size

SNES uses 2 sizes for tiles - 8x8 and 16x16. Wait a minute, didn't you say earlier is only 8x8? Yes, I did! 16x16 tiles are actually 4 tiles (2x2) stored in a certain order.

5. Important registers (soon)

III.2. Tile internal storage (Bit-planes) EXTENDED

2.1 What is a Bit-plane

A Bit-Plane is a 2D MxN matrix/array/block of 1 bit values, yes only 0 or 1!
In graphics, if we take bit 0 (rightmost) of each pixel of a picture, the 2d matrix of all bit 0 is a Bit-Plane0. Similarly, of all bit1 of pixels of a picture is Bit-Plane1 and so on.

Lets talk specifically about Tiles in 4BPP:
If we take all bit 0 of all 8x8 pixels, we will have 8x8 1bit matrix - Bit-Plane0
If we take all bit 1 of all 8x8 pixels, we will have 8x8 1bit matrix - Bit-Plane1
If we take all bit 2 of all 8x8 pixels, we will have 8x8 1bit matrix - Bit-Plane2
If we take all bit 3 of all 8x8 pixels, we will have 8x8 1bit matrix - Bit-Plane3
* each Bit-Plane is 8 bytes, each byte contain a row of bits
* Number of BPP = Number of Bit-Planes!

2.2. Example

```
   Tile in Decimal:                          Same Tile in Binary (4bit):
00 01 02 03 04 05 06 07         0000 0001 0010 0011 0100 0101 0110 0111
08 09 10 11 12 13 14 15         1000 1001 1010 1011 1100 1101 1110 1111
00 01 02 03 04 05 06 07         0000 0001 0010 0011 0100 0101 0110 0111
08 09 10 11 12 13 14 15   ---\  1000 1001 1010 1011 1100 1101 1110 1111
00 01 02 03 04 05 06 07   ---/  0000 0001 0010 0011 0100 0101 0110 0111
08 09 10 11 12 13 14 15         1000 1001 1010 1011 1100 1101 1110 1111
00 01 02 03 04 05 06 07         0000 0001 0010 0011 0100 0101 0110 0111
08 09 10 11 12 13 14 15         1000 1001 1010 1011 1100 1101 1110 1111
```

```
  Bit-Plane 0:       Bit-Plane 1:       Bit-Plane 2:       Bit-Plane 3:
0 1 0 1 0 1 0 1   0 0 1 1 0 0 1 1   0 0 0 0 1 1 1 1   0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1   0 0 1 1 0 0 1 1   0 0 0 0 1 1 1 1   1 1 1 1 1 1 1 1
0 1 0 1 0 1 0 1   0 0 1 1 0 0 1 1   0 0 0 0 1 1 1 1   0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1   0 0 1 1 0 0 1 1   0 0 0 0 1 1 1 1   1 1 1 1 1 1 1 1
0 1 0 1 0 1 0 1   0 0 1 1 0 0 1 1   0 0 0 0 1 1 1 1   0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1   0 0 1 1 0 0 1 1   0 0 0 0 1 1 1 1   1 1 1 1 1 1 1 1
0 1 0 1 0 1 0 1   0 0 1 1 0 0 1 1   0 0 0 0 1 1 1 1   0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1   0 0 1 1 0 0 1 1   0 0 0 0 1 1 1 1   1 1 1 1 1 1 1 1
```

2.3 How Bit-Planes are stored in SNES

In the perfect world we will be done, but in SNES things are even more complicated. In memory, this is present as:

```
  p -  Bit-Plane,   pX - Bit-Plane X
  r -  row,  rX - row X
  pX:rY - a byte value made by the 8 bits of a row Y in Bit-Plane X
p0:r0  p1:r0  p0:r1  p1:r1  p0:r2  p1:r2  p0:r3  p1:r3  p0:r4  p1:r4  p0:r5  p1:r5  p
0:r6  p1:r6  p0:r7  p1:r7
p2:r0  p3:r0  p2:r1  p3:r1  p2:r2  p3:r2  p2:r3  p3:r3  p2:r4  p3:r4  p2:r5  p3:r5  p
2:r6  p3:r6  p2:r7  p3:r7
```

2.4 Fill the missing XXs:

As a practical task, I will give you the value of few pX:rY. Your task is to tell me the missing ones marked as XX. All values are in hex!

```
55 33 XX XX XX XX XX XX XX XX XX XX XX XX XX XX
0F 00 XX XX XX XX XX XX XX XX XX XX XX XX XX XX
```

2.5 Some interesting cases:

- 2bpp? It's very easy, we have only 2 bit-planes and only 1 row (16 bytes). Its not by chance I used 2 rows to put bytes of a Tile :)
- 3bpp? First row is the same. Second row is only 8 bytes and contain bytes of Bit-Plane 2.
- 3bpp->4bpp? (please tell, how do you think it will happen)

IV. Palette

1. What is a Palette?

Our next abstraction is a Palette. Think of an artist painting a picture - although the picture may contain millions of colors, while painting the artist take only couple of colors on a palette (tray/board) to draw specific areas of the picture. The same way, when the PPU draw a Tile, it uses **a selection of colors named a Palette**.

The number of colors in a Palette is tightly connected with the Layer BPP:
- If a Layer is 4bpp, then the Palette contain 2^4 = 16 colors with indexes 0..15.
- If a Layer is 2bpp, then the Palette contain 2^2 = 4 colors with indexes 0..3
- How many colors we have in 8bpp?

2. Color (Palette Entry)

In each Palette entry 0 is a special - it is used for transparency. All colors are 16 bit in the format: 0BBBBBGGGGGRRRRR.
That is 5 bits for each component - Red/Green/Blue. In specialized books you will see this abbreviated as 5-5-5.

3. Palette memory

PPU contain dedicated memory of 512 bytes for Palettes, abbreviated as CGRAM. That memory can hold 256 colors. Depending of the Layer's BPP that memory is used differently:
- in 8bpp - we have 1 Palette containing 256 colors, entries: $00..$FF
- in 4bpp - we have 8 Palettes, occupying entries: $00..$7F
- in 2bpp (Mode 0) we have 8 Palettes for each layer: $00..$1F (BG1), $20..$3F (BG2), $40..$5F (BG3) and $60..7F (BG4)

* Sprites use 8 Palettes, occupying entries: $80..$FF

4. PPU Registers:

```
$2121: Address for accessing CGRAM (1b/W). This register selects the word location
(byte address * 2)
```

```
$2122: Data write to CGRAM (1b/W)
$213B: Data read from CGRAM (1b/R)
*Since read/write return only 1 bite, you need 2 consecutive operations to get the
whole color (16bits)
```

V. Sprite

1. What is a Sprite?

A computer graphic element (bitmap) which may be manipulated as a single entity. It's interesting that earlier they were called - movable objects, stamp or simply objects. I guess the way we imagine sprites (ghosts) as floating transparent entity over, is the reason for calling them sprites.

2. SNES specifics:

- All sprites are in 4bpp (16 colors)
- They have dedicated entries in the CGRAM: $80..$FF
- There 128 Sprites max with ID 0..127
- There are only 2 sizes (dimensions) for sprite called small and large. Small could be 8x8, 16x16, 32x32 pixels. Large could be 16x16, 32x32, 64x64
- Sprites may share the same tile-set
- Sprites may share the same Palette

3. Sprite properties (attributs)

SNES has dedicated memory of 512+32 (544) bytes called OAM (Object Attribute Memory). This memory contains 128 entries of 4 bytes + 2 bits (34 bits per entry)

```
  Byte 0 - X-Coordinate (lower 8bit) (upper 1bit at end of OAM)
  Byte 1 - Y-Coordinate (all 8bits)
  Byte 2 - Tile Number  (lower 8bit) (upper 1bit within Attributes)
  Byte 3 - Attributes:
    Bit7   Y-Flip (0=Normal, 1=Mirror Vertically)
    Bit6   X-Flip (0=Normal, 1=Mirror Horizontally)
    Bit5-4 Priority relative to BG (0=Low..3=High)
    Bit3-1 Palette Number (0-7) (OBJ Palette 4-7 can use Color Math via CGADSUB)
    Bit0   Tile Number (upper 1bit)
```

After above 512 bytes, additional 32 bytes follow, containing 2-bits per Sprite:

```
    Bit7    OBJ 3 OBJ Size     (0=Small, 1=Large)
    Bit6    OBJ 3 X-Coordinate (upper 1bit)
    Bit5    OBJ 2 OBJ Size     (0=Small, 1=Large)
    Bit4    OBJ 2 X-Coordinate (upper 1bit)
    Bit3    OBJ 1 OBJ Size     (0=Small, 1=Large)
    Bit2    OBJ 1 X-Coordinate (upper 1bit)
    Bit1    OBJ 0 OBJ Size     (0=Small, 1=Large)
    Bit0    OBJ 0 X-Coordinate (upper 1bit)
```

4. Sprite tile-set

When a Sprite is 8x8 it uses only 1 Tile. But when the sprite is larger, it uses couple of Tiles, called tile-set. This tiles are hold in VRAM as layers. Since we give only the TileID of the top-left tile, let's see what other tiles are used and how. For example - sprite 32x32 pixels (4x4 tiles) with TileID 00:

```
  Normally:     SNES:
00 01 02 03   00 01 02 03
04 05 06 07 \ 10 11 12 13
08 09 0A 0B / 20 21 22 23
0C 0D 0E 0F   30 31 32 33
```

- We lay Tiles of row1, after that of row2,... If OAM has TileID = N, then that is N, N+1, N+2,...
- Each row is separated by 16($10) tiles. If OAM has TileID = n, then row I looks like: N+$10*I, N+$10*I+1, N+$10*I+2,...

- Since 1 tile has 32 bytes, then in VRAM each row is separated by 16*32 = 512 bytes.

So what will happen with tiles with ID $04..$0F, $14..$1F, $24..$2F, $34..3F? Well we can put 3 extra Sprites. This is called interleaving.

5. Sprite Registers:

```
$2101: OAM params - sssnnbbb
   sss - Sprite size:
        0 = 8x8    16x16 ;Caution:
        1 = 8x8    32x32 ;In 224-lines mode, OBJs with 64-pixel height
        2 = 8x8    64x64 ;may wrap from lower to upper screen border.
        3 = 16x16  32x32 ;In 239-lines mode, the same problem applies
        4 = 16x16  64x64 ;also for OBJs with 32-pixel height.
        5 = 32x32  64x64
        6 = 16x32  32x64 (undocumented)
        7 = 16x32  32x32 (undocumented)
   nn - gap???
   bbb - Base Address for Sprite Tiles (8K-word steps) (16K-byte steps)
$2102/03: OAM address
$2104: Data write to OAM (1b/W)
$2138: Data read from OAM (1b/R)
```