PPU scrolling

From Nesdev wiki

Scrolling is the movement of the displayed portion of the map. Games scroll to show an area much larger than the 256x240 pixel screen. For example, areas in *Super Mario Bros*. are many screens wide. The NES's first major improvement over its immediate predecessors (ColecoVision and Sega Mark 1) was pixel-level scrolling of playfields.

Contents

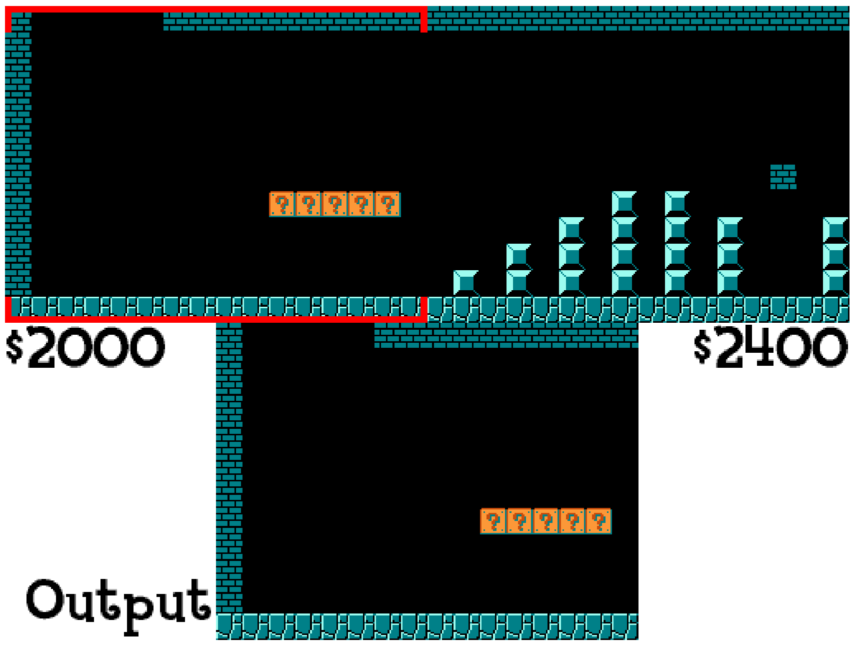
The common case

Ordinarily, a program writes to two PPU registers to set the scroll position in its NMI handler:

1. Find the X and Y coordinates of the upper left corner of the visible area (the part seen by the "camera")
2. Write the X coordinate to PPUSCROLL ($2005)
3. Write the Y coordinate to PPUSCROLL
4. Write the starting page (high order bit of X and Y) to bits 0 and 1 of PPUCTRL ($2000)

The scroll position written to PPUSCROLL is applied at the end of vertical blanking, just before rendering begins, therefore these writes need to occur before the end of vblank. Also, because writes to PPUADDR ($2006) can overwrite the scroll position, the two writes to PPUSCROLL must be done after any updates to VRAM using PPUADDR.

By itself, this allows moving the camera within a usually two-screen area (see Mirroring), with horizontal and vertical wraparound if the camera goes out of bounds. To scroll over a larger area than the two screens that are already in VRAM, you choose appropriate offscreen columns or rows of the nametable, and you write that to VRAM before you set the scroll, as seen in the animation below. The area that needs rewritten at any given time is sometimes called the "seam" of the scroll.



Frequent pitfalls

Don't take too long
: If your NMI handler routine takes too long and PPUSCROLL ($2005) is not set before the end of vblank, the scroll will not be correctly applied this frame. Most games do not write more than 64 bytes to the nametable per NMI, more than this may require advanced techniques to fit this narrow window of time.

Set the scroll last
: PPUSCROLL must always be set after using PPUADDR ($2006). They have a shared internal register and using PPUADDR will overwrite the scroll position.

PPU internal registers

If the screen does not use split-scrolling, setting the position of the background requires only writing the X and Y coordinates to $2005 and the high bit of both coordinates to $2000.

Programming or emulating a game that uses complex raster effects, on the other hand, requires a complete understanding of how the various address registers inside the PPU work.

Here are the related registers:

v
: Current VRAM address (15 bits)

t
: Temporary VRAM address (15 bits); can also be thought of as the address of the top left onscreen tile.

x
: Fine X scroll (3 bits)

w
: First or second write toggle (1 bit)

The PPU uses the current VRAM address for both reading and writing PPU memory thru $2007, and for fetching nametable data to draw the background. As it's drawing the background, it updates the address to point to the nametable data currently being drawn. Bits 10-11 hold the base address of the nametable minus $2000. Bits 12-14 are the Y offset of a scanline within a tile.

The 15 bit registers *t* and *v* are composed this way during rendering:

```
yyy NN YYYYY XXXXX
||| || ||||| +++++-- coarse X scroll
||| || +++++-------- coarse Y scroll
||| ++-------------- nametable select
+++----------------- fine Y scroll
```

Register controls

In the following, *d* refers to the data written to the port, and *A* through *H* to individual bits of a value.

$2005 and $2006 share a common write toggle, so that the first write has one behaviour, and the second write has another. After the second write, the toggle is reset to the first write behaviour. This toggle may be manually reset by reading $2002.

$2000 write

```
t: ...BA.. ........ = d: ......BA
```

$2002 read

```
w:                  = 0
```

$2005 first write (*w* is 0)

```
t: ....... ...HGFED = d: HGFED...
x:              CBA = d: .....CBA
w:                  = 1
```

$2005 second write (*w* is 1)

```
t: CBA..HG FED..... = d: HGFEDCBA
w:                  = 0
```

$2006 first write (*w* is 0)

```
t: .FEDCBA ........ = d: ..FEDCBA
t: X...... ........ = 0
w:                  = 1
```

$2006 second write (*w* is 1)

```
t: ....... HGFEDCBA = d: HGFEDCBA
v                   = t
w:                  = 0
```

At dot 256 of each scanline

If rendering is enabled, the PPU increments the vertical position in *v*. The effective Y scroll coordinate is incremented, which is a complex operation that will correctly skip the attribute table memory regions, and wrap to the next nametable appropriately. See Wrapping around below.

At dot 257 of each scanline

If rendering is enabled, the PPU copies all bits related to horizontal position from *t* to *v*:

```
v: ....F.. ...EDCBA = t: ....F.. ...EDCBA
```

During dots 280 to 304 of the pre-render scanline (end of vblank)

If rendering is enabled, at the end of vblank, shortly after the horizontal bits are copied from *t* to *v* at dot 257, the PPU will repeatedly copy the vertical bits from *t* to *v* from dots 280 to 304, completing the full initialization of *v* from *t*:

```
v: IHGF.ED CBA..... = t: IHGF.ED CBA.....
```

Between dot 328 of a scanline, and 256 of the next scanline

If rendering is enabled, the PPU increments the horizontal position in *v* many times across the scanline, it begins at dots 328 and 336, and will continue through the next scanline at 8, 16, 24... 240, 248, 256 (every 8 dots across the scanline until 256). The effective X scroll coordinate is incremented, which will wrap to the next nametable appropriately. See Wrapping around below.

$2007 reads and writes

Outside of rendering, reads from or writes to $2007 will add either 1 or 32 to *v* depending on the VRAM increment bit set via $2000. During rendering (on the pre-render line and the visible lines 0-239, provided either background or sprite rendering is enabled), it will update *v* in an odd way, triggering a coarse X increment and a Y increment simultaneously (with normal wrapping behavior). Internally, this is caused by the carry inputs to various sections of *v* being set up for rendering, and the $2007 access triggering a "load next value" signal for **all** of *v* (when not rendering, the carry inputs are set up to linearly increment *v* by either 1 or 32). This behavior is not affected by the status of the increment bit. The Young Indiana Jones Chronicles uses this for some effects to adjust the Y scroll during rendering, and also Burai Fighter (U) to draw the scorebar. If the $2007 access happens to coincide with a standard VRAM address increment (either horizontal or vertical), it will presumably *not* double-increment the relevant counter.

Explanation

- The implementation of scrolling has two components. There are two fine offsets, specifying what part of an 8x8 tile each pixel falls on, and two coarse offsets, specifying which tile. Because each tile corresponds to a single byte addressable by the PPU, during rendering the coarse offsets reuse the same VRAM address register (*v*) that is normally used to send and receive data from the PPU. Because of this reuse, the two registers $2005 and $2006 both offer control over *v*, but $2005 is mapped in a more obscure way, designed specifically to be convenient for scrolling.
- $2006 is simply to set the VRAM address register. This is why the second write will immediately set *v*; it is expected you will immediately use this address to send data to the PPU via $2007. The PPU memory space is only 14 bits wide, but *v* has an extra bit that is used for scrolling only. The first write to $2006 will clear this extra bit (for reasons not known).
- $2005 is designed to set the scroll position before the start of the frame. This is why it does not immediately set *v*, so that it can be set at precisely the right time to start rendering the screen.
- The high 5 bits of the X and Y scroll settings sent to $2005, when combined with the 2 nametable select bits sent to $2000, make a 12 bit address for the next tile to be fetched within the nametable address space $2000-2FFF. If set before the end of vblank, this 12 bit address gets loaded directly into *v* precisely when it is needed to fetch the tile for the top left pixel to render.
- The low 3 bits of X sent to $2005 (first write) control the fine pixel offset within the 8x8 tile. The low 3 bits goes into the separate *x* register, which just selects one of 8 pixels coming out of a set of shift registers. This fine X value does not change during rendering; the only thing that changes it is a $2005 first write.
- The low 3 bits of Y sent to $2005 (second write) control the vertical pixel offset within the 8x8 tile. The low 3 bits goes into the high 3 bits of the *v* register, where during rendering they are not used as part of the PPU memory address (which is being overridden to use the nametable space $2000-2FFF). Instead they count the lines until the coarse Y memory address needs to be incremented (and wrapped appropriately when nametable boundaries are crossed).

See also: PPU Frame timing

Summary

The following diagram illustrates how several different actions may update the various internal registers related to scrolling. See Examples below for usage examples.

Action	Before				Instructions	After				Notes
	t	v	x	w		t	v	x	w	
$2000 write	....	....	...	.	LDA #$00 (%00000000) STA $2000	...00..	....	...	.	
$2002 read	...00..	....	...	.	LDA $2002	...00..	....	...	0	Resets paired write latch *w* to 0.
$2005 write 1	...00..	....	...	0	LDA #$7D (%01111101) STA $2005	...00.. ...01111	....	101	1	
$2005 write 2	...00.. ...01111	....	101	1	LDA #$5E (%01011110) STA $2005	1100001 01101111	....	101	0	
$2006 write 1	1100001 01101111	....	101	0	LDA #$3D (%00111101) STA $2006	0111101 01101111	....	101	1	Bit 14 (15th bit) of *t* gets set to zero
$2006 write 2	0111101 01101111	....	101	1	LDA #$F0 (%11110000) STA $2006	0111101 11110000	0111101 11110000	101	0	After *t* is updated, contents of *t* copied into *v*

Wrapping around

The following pseudocode examples explain how wrapping is performed when incrementing components of *v*. This code is written for clarity, and is not optimized for speed.

Coarse X increment

The coarse X component of *v* needs to be incremented when the next tile is reached. Bits 0-4 are incremented, with overflow toggling bit 10. This means that bits 0-4 count from 0 to 31 across a single nametable, and bit 10 selects the current nametable horizontally.

```
if ((v & 0x001F) == 31) // if coarse X == 31
  v &= ~0x001F          // coarse X = 0
  v ^= 0x0400           // switch horizontal nametable
else
  v += 1                // increment coarse X
```

Y increment

If rendering is enabled, fine Y is incremented at dot 256 of each scanline, overflowing to coarse Y, and finally adjusted to wrap among the nametables vertically.

Bits 12-14 are fine Y. Bits 5-9 are coarse Y. Bit 11 selects the vertical nametable.

```
if ((v & 0x7000) != 0x7000)        // if fine Y < 7
  v += 0x1000                      // increment fine Y
else
  v &= ~0x7000                     // fine Y = 0
  int y = (v & 0x03E0) >> 5        // let y = coarse Y
  if (y == 29)
    y = 0                          // coarse Y = 0
    v ^= 0x0800                    // switch vertical nametable
  else if (y == 31)
    y = 0                          // coarse Y = 0, nametable not switched
  else
    y += 1                         // increment coarse Y
  v = (v & ~0x03E0) | (y << 5)     // put coarse Y back into v
```

Row 29 is the last row of tiles in a nametable. To wrap to the next nametable when incrementing coarse Y from 29, the vertical nametable is switched by toggling bit 11, and coarse Y wraps to row 0.

Coarse Y can be set out of bounds (> 29), which will cause the PPU to read the attribute data stored there as tile data. If coarse Y is incremented from 31, it will wrap to 0, but the nametable will not switch. For this reason, a write >= 240 to $2005 may appear as a "negative" scroll value, where 1 or 2 rows of attribute data will appear before the nametable's tile data is reached. (Some games use this to move the top of the nametable out of the Overscan area.)

Tile and attribute fetching

The high bits of *v* are used for fine Y during rendering, and addressing nametable data only requires 12 bits, with the high 2 CHR addres lines fixed to the 0x2000 region. The address to be fetched during rendering can be deduced from *v* in the following way:

```
tile address      = 0x2000 | (v & 0x0FFF)
attribute address = 0x23C0 | (v & 0x0C00) | ((v >> 4) & 0x38) | ((v >> 2) & 0x07)
```

The low 12 bits of the attribute address are composed in the following way:

```
NN 1111 YYY XXX
|| |||| ||| +++-- high 3 bits of coarse X (x/4)
|| |||| +++------ high 3 bits of coarse Y (y/4)
|| ++++---------- attribute offset (960 bytes)
++--------------- nametable select
```

Examples

Single scroll

If only one scroll setting is needed for the entire screen, this can be done by writing $2000 once, and $2005 twice before the end of vblank.

1. The low two bits of $2000 select which of the four nametables to use.
2. The first write to $2005 specifies the X scroll, in pixels.
3. The second write to $2005 specifies the Y scroll, in pixels.

This should be done after writes to $2006 are completed, because they overwrite the *t* register. The *v* register will be completely copied from *t* at the end of vblank, setting the scroll.

Note that the series of two writes to $2005 presumes the toggle that specifies which write is taking place. If the state of the toggle is unknown, reset it by reading from $2002 before the first write to $2005.

Instead of writing $2000, the first write to $2006 can be used to select the nametable, if this happens to be more convenient (usually it is not because it will toggle *w*).

Split X scroll

The X scroll can be changed at the end of any scanline when the horizontal components of *v* get reloaded from *t*: Simply make writes to $2000/$2005 before the end of the line.

1. The first write to $2005 alters the horizontal scroll position. The fine *x* register (sub-tile offset) gets updated immediately, but the coarse horizontal component of *t* (tile offset) does not get updated until the end of the line.
2. An optional second write to $2005 is inconsequential; the changes it makes to *t* will be ignored at the end of the line. However, it will reset the write toggle *w* for any subsequent splits.
3. Write to $2000 if needed to set the high bit of X scroll, which is controlled by bit 0 of the value written. Writing $2000 changes other rendering properties as well, so make sure the other bits are set appropriately.

Like the single scroll example, reset the toggle by reading $2002 if it is in an unknown state. Since a write to $2005 and a read from $2002 are equally expensive in both bytes and time, whether you use one or the other to prepare for subsequent screen splits is up to you.

The first write to $2005 should usually be made as close to the end of the line as possible, but before the start of hblank when the coarse x scroll is copied from *t* to *v*. Because about 4 pixels of timing jitter are normally unavoidable, $2005 should be written a little bit early (once hblank begins, it is too late). The resulting glitch at the end of the line can be concealed by a line of one colour pixels, or a sprite. To eliminate the glitch altogether, the following more advanced X/Y scroll technique could be used to update *v* during hblank instead.

Split X/Y scroll

Cleanly setting the complete scroll position (X and Y) mid-screen takes four writes:

1. Nametable number << 2 (that is: $00, $04, $08, or $0C) to $2006
2. Y to $2005
3. X to $2005
4. Low byte of nametable address to $2006, which is ((Y & $F8) << 2) | (X >> 3)

The last two writes should occur during horizontal blanking to avoid visual errors.

Details

To split both the X and Y scroll on a scanline, we must perform four writes to $2006 and $2005 alternately in order to completely reload *v*. Without the second write to $2006, only the horizontal portion of *v* will loaded from *t* at the end of the scanline. By writing twice to $2006, the second write causes an immediate full reload of *v* from *t*, allowing you to update the vertical scroll in the middle of the screen. Because of the background pattern FIFO, the visible effect of a mid-scanline write is delayed by 1 to 2 tiles.

The writes to PPU registers are done in the order of $2006, $2005, $2005, $2006. This order of writes is important, understanding that the write toggle for $2005 is shared with $2006. As always, if the state of the toggle is unknown before beginning, read $2002 to reset it.

In this example we will perform two writes to each of $2005 and $2006. We will set the X scroll (X), Y scroll (Y), and nametable select (N) by writes to $2005 and $2006. This diagram shows where each value fits into the four register writes.

```
N: %01
X: %01111101 = $7D
Y: %00111110 = $3E
```

```
$2005.1 = X                                                       = %01111101 = $7D
$2005.2 = Y                                                       = %00111110 = $3E
$2006.1 = ((Y & %11000000) >> 6) | ((Y & %00000011) << 4) | (N << 2) = %00010100 = $14
$2006.2 = ((X & %11111000) >> 3) | ((Y & %00111000) << 2)            = %11101111 = $EF
```

However, since there is a great deal of overlap between the data sent to $2005 and $2006, only the last write to any particular bit of *t* matters. This makes the first write to $2006 mostly redundant, and we can simplify its setup significantly:

```
$2006.1 = N << 2                                                  = %00000100 = $04
```

There are other redundancies in the writes to $2005, but since it is likely the original X and Y values are already available, these can be left as an exercise for the reader.

Before				Instructions	After				Notes
t	v	x	w		t	v	x	w	
....	....	...	0	LDA #$04 (%00000100) STA $2006	0000100	....	...	1	Bit 14 of *t* set to zero
0000100	....	...	1	LDA #$3E (%00111110) STA $2005	1100100 111.....	....	...	0	Behaviour of 2nd $2005 write
1100100 111.....	....	...	0	LDA #$7D (%01111101) STA $2005	1100100 11101111	....	101	1	Behaviour of 1st $2005 write
1100100 11101111	....	101	1	LDA #$EF (%11101111) STA $2006	1100100 11101111	1100100 11101111	101	0	After *t* is updated, contents of *t* copied into *v*

Timing for this series of writes is important. Because the Y scroll in *v* will be incremented at dot 256, you must either set it to the intended Y-1 before dot 256, or set it to Y after dot 256. Many games that use split scrolling have a visible glitch at the end of the line by timing it early like this.

Alternatively you can set the intended Y after dot 256. The last two writes ($2005.1 / $2006.2) can be timed to fall within hblank to avoid any visible glitch. Hblank begins after dot 256, and ends at dot 320 when the first tile of the next line is fetched.

Because this method sets *v* immediately, it can be used to set the scroll in the middle of the line. This is not normally recommended, as the difficulty of exact timing and interaction of tile fetches makes it difficult to do cleanly.

Quick coarse X/Y split

Since it is the write to $2006 when *w*=1 that transfers the contents of *t* to *v*, it is not strictly necessary to perform all 4 writes as above, so long as one is willing to accept some trade-offs.

For example, if you only write to $2006 twice, you can update coarse X, coarse Y, N, and the bottom 2 bits of fine y. The top bit of fine y is cleared, and fine x is unchanged.

$2006's contents are in the same order as *t*, so you can affect the bits as:

```
   First      Second
/¯¯¯¯¯¯¯¯¯\ /¯¯¯¯¯¯¯\
0 0yy NN YY YYY XXXXX
  ||| || || ||| +++++-- coarse X scroll
  ||| || ++-+++-------- coarse Y scroll
  ||| ++--------------- nametable select
  +++------------------ fine Y scroll
```

References

- The skinny on NES scrolling (http://nesdev.com/loopyppu.zip) original document by loopy, 1999-04-13
- Drag's X/Y scrolling example (http://forums.nesdev.com/viewtopic.php?p=78593#p78593) from the forums
- VRAM address register chip photograph analysis by Quietust