Toggle navigation N3S

- Main page
- Recent changes
- Search Search

How It Works

Contents

Graphic Data

How the NES does it

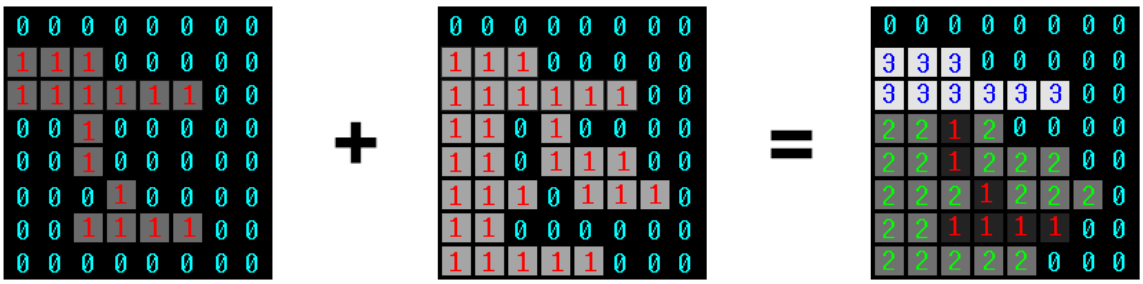
Most NES cartridges store all **pattern data** (**tiles**) in an area of ROM called **CHR**. These are individual chips that are 8KB or larger, and there are sometimes several of them on the PCB. Some games compress tile data into PRG (where audio and game logic is stored) and load it dynamically instead.

Each tile is 8x8 pixels. Most in-game characters are therefore made up of several (screenshot from YY-CHR (https://web.archive.org/web/20190523120018/http://www.romhacking.net/utilities/119/)):

CHR data

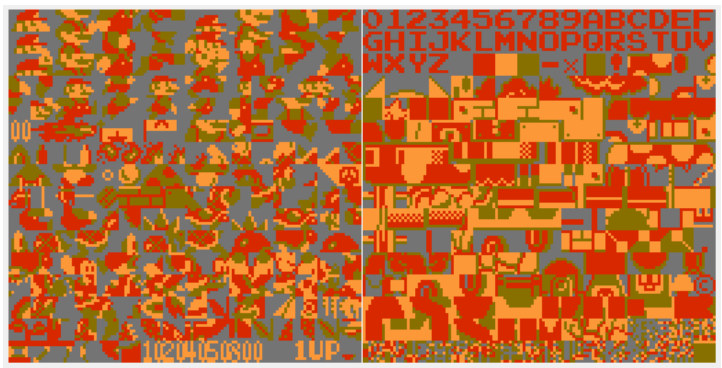
Tiles are 16 bytes each. Each pixel is either transparent, or 1 of 3 colors. In base zero, this means each pixel can have any value between 0-3.

The data is stored in an interesting way, where each row is two bytes and the color is 1-3 if the first, second, or both bytes contain 1:



Pattern tables

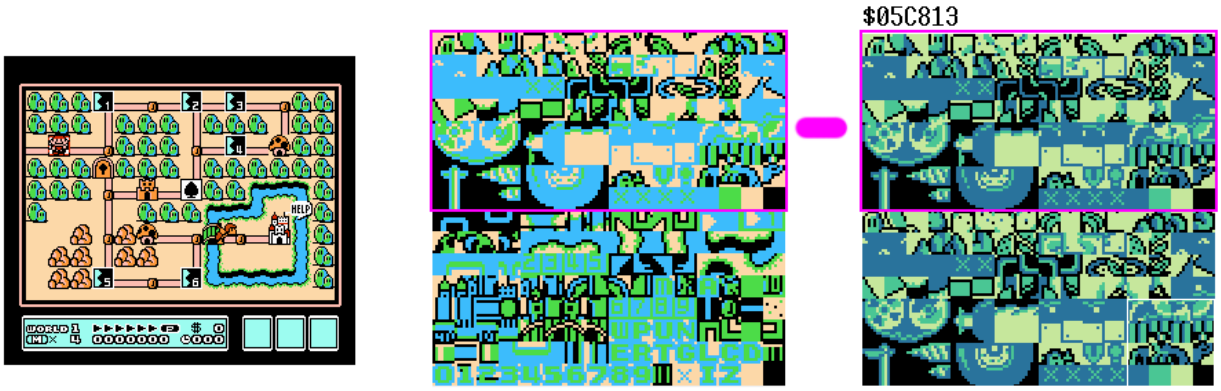
For most games, the available tiles are stored in 8KB of memory known as the **pattern table**. Though this is read as continuous chunk of memory, the foreground (OAM) and background (nametable) sprites can only point to one half of it to use at any given time. This causes most debuggers to display it as two 4KB halves typically labelled "left" and "right". Here is how the FCEUX debugger displays it:



At any given time there should be 512 tiles available to be drawn, with 256 coming from each half.

The interesting thing about the pattern table is that it is not stored in the NES VRAM. The PPU actually maps the CHR data right from the cartridge. For Super Mario Bros (as seen above) this is simple, as there is 8kb of CHR-ROM data in a chip that contains all the tiles the game has to offer.

However, for games with more than 8KB of CHR-ROM (the average is much higher, sometimes 128KB-256KB) the cartridges need to be capable of **bank switching**, which routes requests for tiles to independent sections in the CHR-ROM. This allows games to switch pattern tables without having to "load" the data, with more advanced **mappers** capable of switching finer / smaller parts of the memory. This can be used to display parts of the game with different levels and enemies, or sometimes for animations like in Super Mario Bros 3:



How N3S does it

Tile Graphics

N3S renders **voxel-based 3D meshes** that are 8x8 and 32 voxels deep.

Until the editor is built, N3S just generates "3D" extrusions of tiles taken directly from CHR-ROM. So every mesh, for now, is just the original tile (but several voxels deep). Reading tiles from CHR-ROM is fairly simple, as shown above.

N3S also stores every mesh in the order it appears in CHR-ROM. So for a game with 1024 tiles (16KB CHR-ROM), the last one N3S reads is stored as #1023 (zero-based).

Pattern tables

Making sure that N3S is rendering the sprite the game wants is actually the hardest part. Every sprite referenced in OAM and nametable memory (see sections below) has a byte (0-255) that specifies which tile it wants to draw. The pattern table selection bits in CTRL specify which half of the pattern table to take it from (so if the OAM has its pattern table selection bit set to 1, I'd add 256 to the tile it's asking for).

If I were to always assume that, say, tile 122 referred to the 122nd tile in all of the game's CHR-ROM, I'd likely wind up drawing the wrong tile in games that use CHR bank switching.

One way around this would be to look deeper into what Nestopia is doing, and make sure I'm taking CHR bank switching into consideration. However, this wouldn't work with games that use other methods such as VRAM.

The approach I took instead was to find how fine bank switching was on the game I was testing and step through all the CHR data, generating a simple hash of each section at that size. (This doesn't hash the whole section, it just samples an even distribution of bytes). So a game with 16KB of CHR data that could bank-switch 1KB sections had 16 hashes. These are stored in a table along with the actual tile offset that section starts at (indicating where in all the CHR memory that tile actually sits). Then, for every frame emulated, I grab hashes of each bank-switchable section in the loaded pattern table, and generate a "virtual" pattern table that returns the true tile ID of any tile requested, taking the actual offset into account.

This might not be the perfect solution, and I might change it down the line. If a game is capable of switching every tile individually in VRAM then I'm hashing 512 sprites to find out what 3D mesh to draw in their place, which could have serious performance consequences (despite how crude and simple my hash algorithm is).

Another possible issue is that some games store a lot of redundant tile data in CHR memory to, for example, present different combinations of enemies on a cart that can't bank-switch very finely. Therefore N3S runs the risk of hash-collisions for different sections if I'm not hashing them thoroughly enough. To get around this, I have the hash complexity increase (to a certain limit) if hash collisions happen during hash generation.

Another issue with redundant tiles is that redundant 3D meshes are created. However, a tile that might exist in several places in CHR-ROM (such as

Some games also generate sprites (fully programmatically or by combining other sprites). These games will likely be incompatible with 3D emulation as N3S cannot generate accurate meshes ahead of time.

Games that store graphics in PRG-ROM

If a game stores graphics (compressed or not) in PRG-ROM, then N3S can't (currently) play them. N3S currently generates meshes from CHR-ROM when the game loads, not when the game uncompresses the sprites into pattern tables.

However, even with predefined 3D models, PRG graphics data poses another issue for manually editing 3D models: My current idea of how someone would create 3D models for a game would be to load it and have N3S generate meshes from CHR that are a single voxel deep, as well as generate hashes for all the CHR sections, to store in an N3S file. These would make perfect starting points for people to push / pull / sculpt full 3D sprites while maintaining the correct side profile.

But for games that don't have CHR-ROM, I have no way of knowing exactly what the graphics are ahead of time. One solution might be to run a TAS and grab each unique CHR bank that comes up during play, although that will take some effort. This issue requires more research.

Colors

How the NES does it

The NES uses **palettes** to display a limited number of colors.

The NES is capable of displaying 64 different colors, though the whole screen can be tinted (or set to greyscale) with flags set in the PPU's MASK register ($2001) to produce a larger variety. These initial 64 colors differ slightly with NES hardware revisions and clones.

At any point in time, there are 8 palettes of 3 colors that can be used, as well as a background color. The FCEUX debugger displays them like so:

4 of these palettes belong to OAM sprites, and 4 to background (nametable) sprites.

As seen above, tiles are made up of pixels which can hold values between 0-3. 0 means transparent, so the background color is drawn (assuming there are no other sprites on that pixel). 1-3 draws the corresponding color in the palette that the sprite is assigned to.

How N3S does it

N3S actually uses **shader code** to render colors.

The 3D meshes that N3S draws are made up of **vertices**, like in any other game. Each vertex can contain a variety of data, which is up to the program to specify. For some games, that data might include 2D coordinates of a texture. For N3S, vertices have position and RGBA (red/green/blue/alpha) color data. Typically, this sets which color the 3D model should be at the vertex, but the shader code in N3S interprets it as **how much that vertex should blend each color on the palette**.

At the beginning of every frame, a **constant buffer (https://web.archive.org/web/20190523120018/https://msdn.microsoft.com/en-us/library/windows/desktop/ff476898(v=vs.85).aspx)** is filled with all of the true RGB values of the palettes and background. For each sprite (unless the palette is the same as the last) another constant buffer is updated that simply selects which palette the sprite is using.

```
This info used to be stored and calculated per-pixel in the pixel shader. However I realized that for each polygon there could only be one whole co
```

The vertex shader then uses the RGBA values to render that vertex as a blend of colors 0-4 in the palette. RGB correspond to 1-3, and A (normally the alpha channel) corresponds to the background color. This is necessary as some sprites use the background color as outlines or shading, like in Metroid.

The RGBA values are actually a range of 0 to 1 (an RGB value for an even grey would be { 0.5, 0.5, 0.5 }), not 0-255 like in most raster applications. Because "branching" (using if-else statements) in shader code hurts performance, I avoided using an integer to represent which palette to use and assume all RGBA values that the program hands over won't add up to more than 1. The shader simply adds all the palette RGB values multiplied by RGBA values together. The vertex shader outputs this final value to the pixel shader, which for now outputs it directly (there is no lighting in N3S currently).

```
Another advantage of this method is that, in the future, I can let N3S creators make voxels a blend of different palette colors, allowing for smoo
```

Rendering Sprites

How the NES does it

OAM

The NES can store data on where and how to draw up to 64 sprites in **Object Attribute Memory (OAM)**. OAM is a 256-byte area in VRAM, where each sprite takes up 4 bytes.

Each byte specifies the following:

- **0**: Y position
- **1**: Tile
  - Which sprite from the pattern table to draw
- **2**: Extra data
  - Palette selection
  - Horizontal flip
  - Vertical flip
  - Render priority
- **3**: X Position

Rendering

The NES actually does some complicated math with how it draws each sprite. One particular quirk is that it needs to load the sprites into another 32-byte chunk of memory for each scanline. Therefore it is limited to 8 sprites per scanline, though it tries to load more and usually fails (https://web.archive.org/web/20190523120018/http://wiki.nesdev.com/w/index.php/PPU_sprite_evaluation#Sprite_overflow_bug).

How N3S does it

Rendering sprites is easy! If there is no mirroring, and the sprite isn't off-screen, I just render the full mesh at the X and Y coordinates given.

However, as a result I'm ignoring the 8-sprite limit per-scanline. This will be implemented in the future.

Rendering mirrored sprites

Mirroring is handled on the vertex-shader level. Each voxel-mesh sprite in N3S is 8x8x32, with the origin in the top-left. Therefore, by multiplying each vertex's X or Y value by -1 I can reverse it (so long as I compensate by moving it one sprite over when updating the world matrix).

I have a constant buffer in the vertex shader that simply stores the horizontal/vertical mirrored states as 0 or 1. To avoid branching in my shader code, I just multiply the position by whatever that value is every time. If the horizontal mirroring is set to 0, then "x * (0 * -1)" won't make any change.

To minimize constant buffer writes, I only update the mirror values when they've actually changed.

One problem with this technique, however, is that some polygons would normally get culled as a result of backface culling (https://web.archive.org/web/20190523120018/https://en.wikipedia.org/wiki/Back-face_culling) when the model is reversed. This is because the video card decides which triangles are facing towards or away from the camera by whether or not the 3 vertices appear in a clockwise or counter-clockwise pattern, and when I reverse a model that order changes. With back-face culling enabled, the video card won't bother to render triangles facing away from the camera.

As a result I have to disable back-face culling. Alternatively, I could generate each mesh in all 4 possible states of mirroring, however for a game with 256kb of CHR data that would mean 16,384 * 4 = 65,536 unique meshes. This would require a massive amount of VRAM, which would be silly for an NES emulator... So I take the performance hit of drawing some redundant pixels instead.

```
This was actually a dealbreaker for the HoloLens, performance-wise, so I had to generate extra meshes instead.
```

Rendering partial sprites

Because sprites are not always entirely on screen, N3S needs to render partial sprites. This can be seen on the nearest side of the "screen" in this screenshot (https://web.archive.org/web/20190523120018/https://s3.amazonaws.com/n3s/screenshots/2.png).

To do this, N3S makes an additional 64 meshes (if needed) for each "pixel" of each sprite that I call Z-meshes. Each of these meshes are made of all the voxels along the Z-axis at that pixel's X&Y. This could theoretically lead to a huge amount of vertex buffers being stored in V-RAM, especially for games that have 128KB of CHR-ROM (8,192 sprites) or more. To assist with this, N3S makes a crude hash of the voxel data long the Z-axis and only creates new meshes where it is truly necessary. Because a large amount of sprites have transparent areas, those areas do not

need any actual mesh. Also, many sprites (such as those that display text) will likely be made up of the exact same meshes (typically just 1 voxel at a certain depth).

Because N3S currently extrudes meshes directly from CHR, and each pixel can only be one of 4 values, there are only ever 3 actual Z-meshes that need to be stored. However, I'll have to see what happens when the editor is built in and more complex 3D models are created.

Another performance flaw with this method, because I'm re-using these Z-meshes and therefore they contain no relational positional data within the sprites themselves, is that I have to update the world matrix up to 56 times and make 56 individual draw calls. I'm running a mid-range video card and it's having no issues so far, but I'll have to benchmark other devices.

Rendering The Background

How the NES does it

Nametables

The NES background consists of 8x8 sprites arranged on a fixed grids called **nametables**. Any given nametable is 32 sprites wide and 30 sprites tall, or 256x240 pixels (the exact size of one screen).

Each nametable also specifies one palette for each 4x4 (16x16 pixel) set of sprites in an accompanying data structure called an **attribute table**. This is why background objects (like the ? block in Super Mario Bros) are mostly the same size and uniform color.

Nametable sprites cannot be mirrored.

The NES can store 2 nametables in memory. A NES cartridge can also provide RAM for 1 or 2 additional nametables.

Nametable Mirroring

Because games need to scroll more than one screen-length, the NES has built-in functions to render nametables side-by-side so that a background can be spread across them.

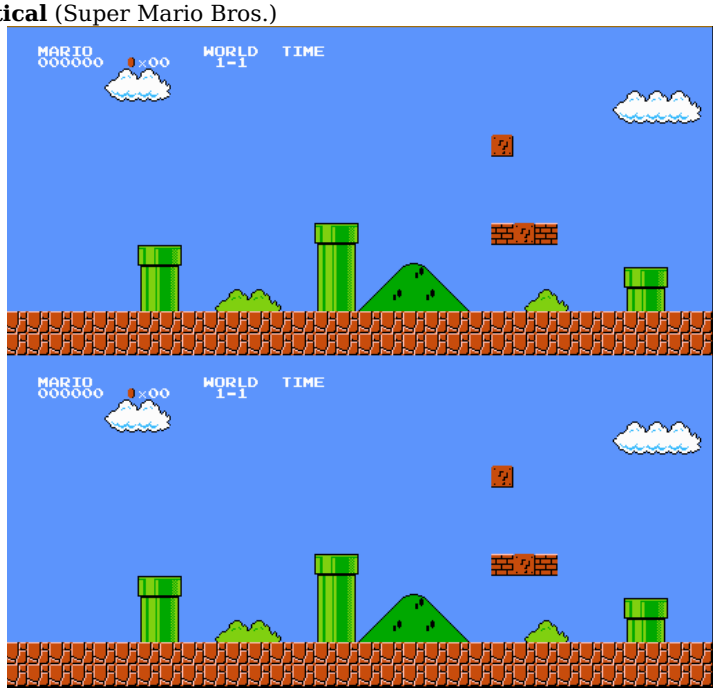
The way that the nametables are arranged is called **mirroring**.

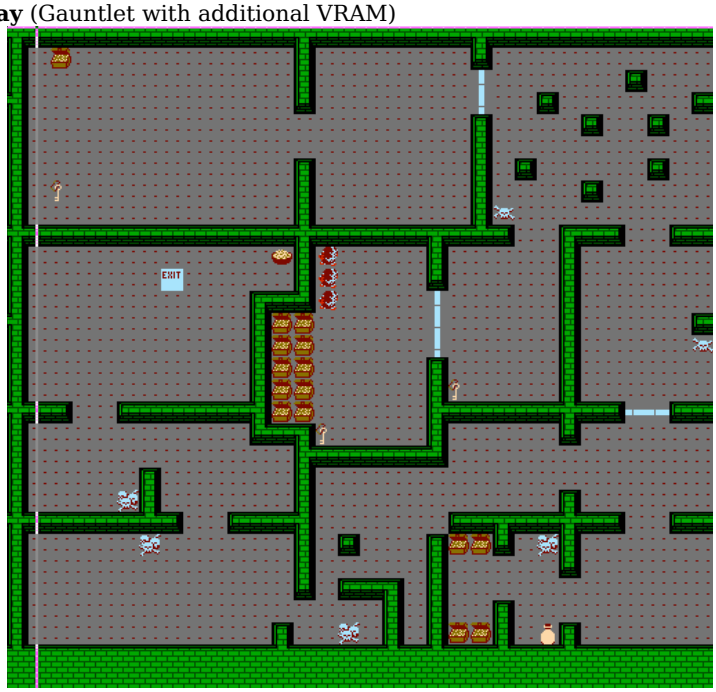
The mirroring type can be dynamically updated by the game, or permanently set into the PCB of the cart during manufacturing.

Only with 4 nametables (two provided by additional VRAM in the cartridge) can a game implement full-screen diagonal scrolling.

There are 4 commonly used mirroring arrangements:

- **Vertical** (Super Mario Bros.)

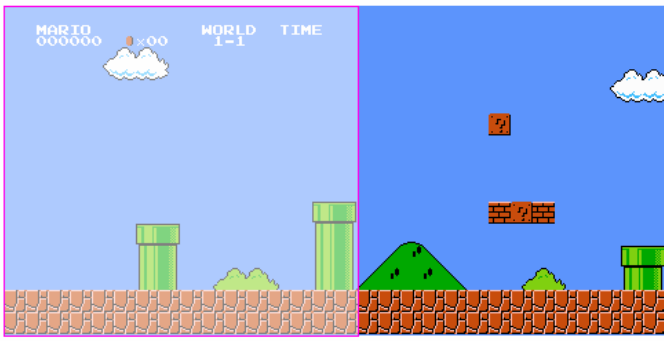


- **Horizontal** (Vertical rooms in Metroid)
  - 
- **4-way** (Gauntlet with additional VRAM)
  - 
- **Single** (Tetris)

Scrolling

Unless nametable rendering is disabled, the NES will render an entire 256x240 pixel section of the background specified by the **scroll values** set by the game.

Scroll values can be set at any point during rendering (though the effects are unpredictable if not set between rendering of individual scanlines). Many games change the scroll mid-frame to show a status bar or menu in part of the screen:



Scroll values are set by the game through a combination of:

- The nametable selection bits from the **CTRL** register ($2000)
- X and Y values sent to the **SCROLL** register ($2005)
- Some crazy math as a result of writing directly to the **ADDR** register ($2006)

Writing to $2005 and $2006 causes a shared toggle to switch back and forth, which changes the behavior of writing to either. (Initial write to $2005 sets the X coordinate, then Y, then X again...)

When scroll values are written, the NES does not store the X or Y in its own variable, but instead updates the memory address of the pixel that is ready to be drawn at $2006. Assuming this happens before each frame is rendered, the game is selecting which pixel of the background is in the very top-left of the screen.

Because of the way that nametable tiles are stored, this is fast and efficient because different bits of a 15-bit address imply fine x and y coordinates within a tile, and coarse x and y coordinates to select the tile itself.

Rendering

During rendering, the NES makes a temporary copy of the address and loops through a single scanline,

continiously writing the pixel at that address and incrementing it to select the next pixel along the X axis. This may involve jumping to the next nametable to the right (or wrapping around), which is handled automatically by the hardware. When it reaches the end of the scanline, it grabs the X value from the original address (which represents screen coordinate 0) and increments the Y to draw the next scanline.

If the X value of the original address has been changed since the last scanline started, then the screen will have effectively scrolled however many pixels over. This can be used for parallax scrolling effects or for a status bar like in the Mario example above. **However if the Y value is updated it is ignored, unless certain writes to $2006 convince the PPU to reload the address entirely.**

How N3S does it

Nametable rendering is by far the most difficult part of emulating the PPU.

The namestables themselves are simple data structures and are easy to pull into structs / classes in code. And mirroring was easy to implement by wrapping the nametables in a class and providing functions for grabbing tiles that took it the mirroring arrangement into account.

Scrolling, however, is infamously difficult to implement, especially if you're emulating a game that sets the Y coordinate after the start of each frame. NES homebrew developers also tend to have trouble figuring out how to change the scroll mid-frame, as the logic becomes significantly more complex from the game's side.

It also forced me to change my approach to N3S. I originally assumed I could grab the VRAM and render its contents at the end of every frame, but I actually had to dig into the Nestopia source code and make it grab scroll values as rendering occurred. I then leave N3S a breadcrumb trail of values written at certain times to make sense of later.

I also had to emulate the scroll address functionality exactly, otherwise I would end up with bizarre results. This was difficult as the exact behavior of the addresses, particularly with writes to $2006, is seemingly random at first glance. One write might clear the highest bit of the coarse Y selection and overwrite another value partially, and only under certain conditions would an earlier write to Y actually effect rendering at some later point. Ultimately, the Nesdev wiki page (https://web.archive.org/web/20190523120018/http://wiki.nesdev.com/w/index.php/PPU_scrolling) on it proved essential.

There are also a lot of random writes to these values that can be ignored. This is partially due to the fact that NES developers didn't know the exact effects and some guesswork was involved, but also because some of the registers have other uses and are written to for other purposes during or between frames. However, after N3S builds an exact timeline of what the scroll was at any given scanline, it removes redundant entries starting from bottom to top to speed up rendering.