エミュレータはソフトなマホウ。

ハード解析“なし”で作るファミコンエミュレータ

Stage.2

PPU ・画面出力

ψ（プサイ）の興味関心空間 http://ledyba.org/

目次

0x00 ソースコードとPDF

この本は、「エミュレータはソフトなマホウ。」の第二巻です。前巻では、基本的な設計とCPUとRAMを作成しました。

第一巻も含めたPDFと、エミュレータのソースコードはこちらにあります。どちらも誤植やバグを見つけ次第直しています。この本はC87でのスナップショットといった感じです。

PDF：https://github.com/ledyba/NesBook

ソースコード：https://github.com/ledyba/Cycloa

どちらもPull Requestをお待ちしています！

0x08 PPU：グラフィックを実装し、見栄えをそれっぽくしよう。

今回はPPU、つまりグラフィック機能の実装に入ります。Stage.1で、CPUからどのように見えるか、を中心に考えていましたが、今回はもう少し広い視点からみてみましょう。

0x0801 CPUから、PPUはどう見える？

今まで、RAM、カートリッジ、CPUと作ってきました。これから作るPPUもこれらの部品と協調して動くわけですから、できている部品とどう接続されるのかを見ていくのがやはり自然でしょう。まずは、CPUからどう見えるかを考えます。

CPUから見たPPU＝メモリを隔てた、むこうがわ

やはり、まずは表示される内容をどうやって読み書きするのかが気になります。

CPUのデータやプログラムが格納されているメモリ空間と、PPUの画像データの格納されているメモリ空間は、直接は繋がっていません。つまり、画面に表示されているデータを「普通の変数」のように直接CPUから読み書きすることは出来ません（図 1）。

その代わり、アドレスはメモリマップドIOを使って特定のあるアドレス（0x2006か0x2003）に、**データとして**PPUのメモリ空間のアドレスの値を書き込むことで指定し、書き込むデータや読み込むデータも、同様に特定のアドレス(0x2007か0x2004)に読み書きすることでPPUとデータの交換を行います。メモリマップドIOを細いトンネルにしてPPUとCPUの世界がつながっているイメージでしょうか（適当）。

具体的に言うと、次のアドレスがそうなります。読み込みと書き込みでは値が異なるので、注意してください。

これらのアドレスを、「PPUレジスタ」と呼ぶことがあります。

アドレス	アドレスを読んだ時に得られる情報
0x2000	未使用
0x2001	未使用
0x2002	PPUの状態を表す値を読むことができます。
0x2003	未使用
0x2004	スプライトRAMの内容を読むことができます。アドレスは別箇所で指定。
0x2005	未使用
0x2006	未使用
0x2007	グラフィックRAMの内容を読むことができます。アドレスは別箇所で指定。

表 *1:* 読み込みする場合の*PPU*メモリ領域

アドレス	アドレスへ書いた時に得られる情報
0x2000	PPUの無効・有効などの基本的な情報を設定します
0x2001	画面の周りのマスクの設定を行います
0x2002	未使用
0x2003	0x2004からスプライトのデータを読み書きするときのアドレスを設定します
0x2004	スプライトRAMに値を書き込みます
0x2005	ハードウェアスクロールの機能を設定します
0x2006	0x2007からグラフィックRAMの内容を読み書きするときのアドレスを指定します。
0x2007	グラフィックRAMに値を書き込みます

表 2: 書き込みする場合のPPUメモリ領域

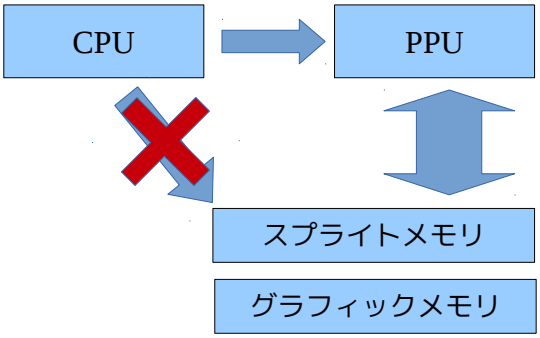

図 1: PPU経由でアクセス

もう一度、今度は詳しく説明します。PPUは、CPUとは違った、別のメモリ空間を**2つ**持っています。このメモリ空間には、以前作ったRAMモジュールとはまた違うRAM部品が乗っており、どちらも描画内容を決定するために使用されます。これが、「スプライトメモリ」と「グラフィックメモリ」です。この内容もCPUから書き換えることができますが、CPUから直接書き込むことはできず、メモリマップドIOを使ってPPUを仲介して読み書きしなければなりません（図 1）。これが0x2004や0x2007で、他のアドレスと違って読み込みも、書き込みもできます。

「スプライトメモリ」や「グラフィックメモリ」に置かれる具体的な内容は、このあとまたじっくり見ていくことにしましょう。

ちなみに、0x2008～0x2fffまでは、同じ8バイトの内容がひたすらミラーリングされています。

CPUの値の書き込み処理はVirtualMachineが仲介していましたね。ここにあとで追記することになるでしょう。

このメモリバスの読み書きによって、「CPUからPPU」の情報の受け渡しが行えるようになります。

CPUから見たPPU＝割り込み発生元

今まで作ってきたパーツは基本的にCPUが完全な主導権を持っていましたが、PPUはCPUから一方的に指示されるだけでなく、PPUからCPUへ情報を能動的に伝えることができます。それが、「割り込み」です。割り込みについては、Stage1で解説を行なっていますので参考にしてください。

実は割り込みにはいくつか種類があるのですが、PPUが発生させるのはNMI（Non Maskable Interrupt）という割り込みです。もう一つの割り込みである「IRQ;Interrupt ReQuest」では、CPUのレジスタのIフラグが1になっていると割り込みが無視される（リクエスト＝あくまでお願い）のですが、NMIでは「Non Maskable」からもわかる通り、割り込みの無視を行うことができません。

CPUへ割り込みが発生されると、CPUはそれまで一旦実行していたプログラムを停止して、「割り込みハンドラ」を実行し始めるのでした。割り込みハンドラはいくつかあり、NMIはIRQとは別の割り込みハンドラを実行します。その割り込みハンドラの指定は、右の表に書かれたアドレスを使います。

アドレス	割り込み
0xFFFA-0xFFFB	NMI
0xFFFC-0xFFFD	リセット時
0xFFFE-0xFFFF	IRQ/BRK

表 3: 割り込みハンドラ

この表にはNMIとIRQの割り込み以外にも書かれていますね。「リセット」も一種の割り込みのようなもので、今まで実行していたプログラムを停止し、CPUの状態を特定の状態に「リセット」した上で、このハンドラが実行されます。

BRKは、6502の持つ命令の一つです。CPUの実行する命令でありながら、IRQ割り込みと同じようにハンドラの実行に移ります。

さて、PPUが割り込みを起こして特定のハンドラの実行を発生させることで情報を伝えることができる、とわかりましたが、ではどんな「情報」を伝えるのでしょうか？

PPUが伝える情報は、「VBLANKが発生したこと」です。ここで、グラフィックチップが画像をレンダリングする仕組みを、少し見てみましょう。

0x0802 PPUが描画するしくみ、VBLANKはPPUのコーヒーブレイク

PPUは画面の絵を作り、それをテレビに送信しますが、この時、一気に画像を１枚出力するわけではなく、少しずつレンダリングしていきます。

その時、まず左上から右に向かってピクセルの列を出力し、右端に達したら一つ下の行の左端に移り、更に右に向かって出力していきます。一番右下に達したら、また左上に戻るのですが、この時すぐに戻るわけではなく、右下から左上に戻るまでにしばらく時間があります。

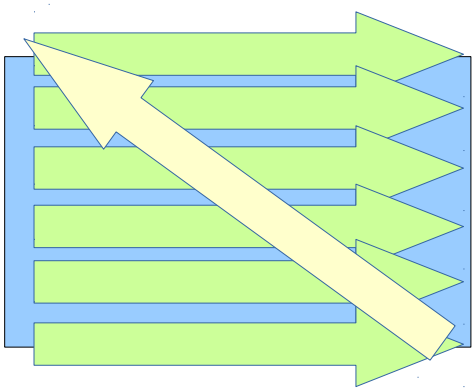

図 2: レンダリング順序

なんで？と思うかもしれませんが、これは少し昔の時代に思いを馳せてみると納得が行くと思います。つまり、昔は今のようにテレビは液晶だった訳ではなく、「ブラウン管」というのを使っていました[1]。ブラウン管では、テレビの奥のほうに電子を発射する銃があり、発射された電子が画面表面に塗ってある蛍光物質が反応すると光る、という仕組みになっていました。この発射される電子線は電界と磁界で曲がって画面の所定の位置に当たるのですが、この光線が右下から左上に戻り、再度描画を始めるための時間が必要なのです。この時間を「垂直帰線期間」「Vブランク」と言います。

さて、この「戻る」「また始める」タイミングを制御しているのがvsync[2]という信号で、この信号が１（High）から０（Low）に変化した時（＝VBLANK開始）に、PPUはCPUにNMI割り込みを発生させます。

Vブランク中はグラフィックをいじり放題

PPUは描画中、PPU内部のステータスやカセットやグラフィックメモリ、スプライトメモリの内容を

1 ブラウン管 – Wikipedia http://ja.wikipedia.org/wiki/ブラウン管

2 電子回路の豆知識 垂直同期信号 http://www.nahitech.com/nahitafu/mame/mame6/sync.html

見ながら、１行ずつ、１ピクセルずつ描画していきます。この描画を行なっている最中に、それらの描画のための情報を変更するのは、非常に危険な行為です。表示される絵がバグってしまうかもしれません。しかし、Vブランク中はこれらの情報を一切参照しないため、自由に画面の表示内容を変更することができます。NMIが発生するのは、このVブランクの直前なので、NMIのハンドラの中では自由に描画のための情報を書き換えることができるというわけです。

0x0803 PPUから見たカートリッジ＝グラフィックデータ置き場

PPUは、CPUと独立して上記のVSYNC信号などと協同しつつ、テレビにゲームの画面を送り続けます。この点において、完全にCPUから制御されるだけのRAMなどとは違い、能動的に動くパーツであるということができます。

では、この能動的に動作するPPUからは、他のパーツはどのように見ることができるのでしょうか？PPUと接続されているパーツは、CPUの他にはカートリッジだけです。他のパーツとは独立しています。しかも、カートリッジのうちの、半分としか接続されていません。どういうことでしょうか？

スプライトメモリやグラフィックメモリはCPUから直接アクセスできず、PPUを経由しなければならない、と書きましたね。カートリッジの中のデータもグラフィックとプログラムに分かれており図3、CPUからはプログラムのデータは直接アクセスできます（そのデータをプログラムとして実行してる）が、グラフィックのデータは直接アクセスすることができません。

カートリッジ内のグラフィックのデータは、PPUの「グラフィックメモリ」の一部として現れます。

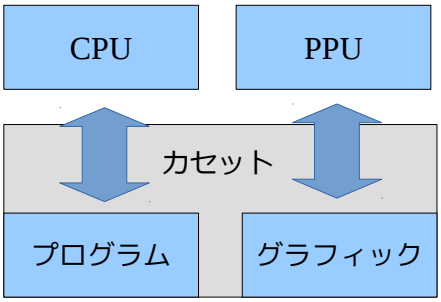

図 *3:* カートリッジの中身も分かれてる

0x0803 バックグラウンドとスプライト

さらにPPUの「スプライトメモリ」と「グラフィックメモリ」について解説する前に、PPUでの描画について説明しなければなりません。もう少しだけ、待ってくださいね。

この時代の2Dグラフィックは、強力なCPUを使って大量のピクセルを操作できる現代とは全然違う考え方をしています。ちょっと驚くかもしれません。

ファミコンのグラフィックは、「バックグラウンド」と「スプライト」の２つに分かれていますので、順当に２つに分けて説明します。

バックグラウンド - タイルグラフィック

「バックグラウンド」は、その名の通り、背景を表すためのグラフィックです。スーパーマリオで例えるなら、ドカンや地面、空の雲などは、すべてこの「バックグラウンド」のシステムで描画されています。

このバックグラウンドで特徴的なのは、バックグラウンドはピクセルの集合ではなく、**タイルを二次元的に敷き詰めたもの**である、と言うことです図 4。このタイルは8x8のピクセルの塊で、ゲーム画面はタイルとピクセルの２段構えで構成されているということになります。

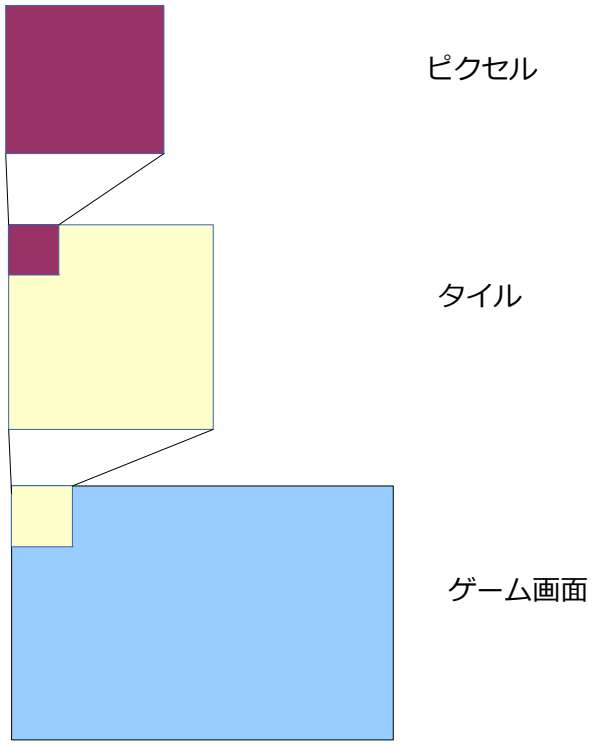

図 *4:* 画面、タイル、ピクセル

タイルが敷き詰められる順番にそって、どのタイルを表示するかの情報だけがRAM上に格納され、タイルそれぞれの実際のピクセルデータはカセット内に保持されています。

なぜこのようにするのかというと、まずひとつに処理速度の問題があります。画面の内容を書き換えるときに、ピクセルをすべて変更しようとするより、8x8のタイル単位で書き換えたほうが、一度にたくさんの部分を書き換えることができて、速度的に有利なのです。

メモリー量の問題もあります。ファミコンの解像度は256x240なので、1ピクセル１バイトで持ったりすると60KBとかトンデモない量になってしまい、とてもじゃないけど、メモリが足りなくなってしまいます（グラフィックではないですが、CPU用のRAMの大きさは2048バイトしか無かったのを思い出してください）。

一方、タイルグラフィックなら、タイルの指定をするための領域（これを、「ネームテーブル」と呼びます。タイルの番号を名前として見立ててるのでしょうか）は、ピクセルを直接保持したときの1/64、つまり960バイトだけですみます。ピクセルデータ自体はカセット内に入っているデータをそのまま直接読み込むので、余分なRAMを消費したりしません。

スプライト – 背景に浮かぶ妖精さん

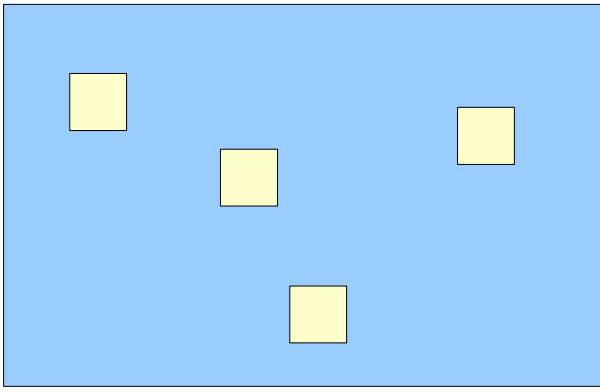

図 5: 画面上を飛ぶスプライト

バックグラウンドは、マリオでは雲や地面のような背景を担っていると書きました。それでは、マリオやクリボーはどのように表示されているのでしょうか？

それが、「スプライト」というシステムです。原義としては「妖精」という意味で[3]、その名の通り、スプライトを使うと、画面上を動き回る「妖精さん」を登場させることができます図 5。

どういう事か、もう少し詳しく説明いたしましょう。「バックグラウンド」ではタイルが8x8のピクセル格子にそって32行x30列＝960個のタイルが敷き詰められているのですが、このせいで、中途半端な場所にタイル画像を表示させることができません。つまり、タイルが置かれる座標は縦横ともにかならず8の倍数になってしまいます。後述する「スクロール機能」を使うと、バックグラウンド全体を数ドット上下左右に動かすことは可能なのですが、移動するのは「全体」なので、マリオの画像だけ１ドット右に動かそう、みたいなことはできません。

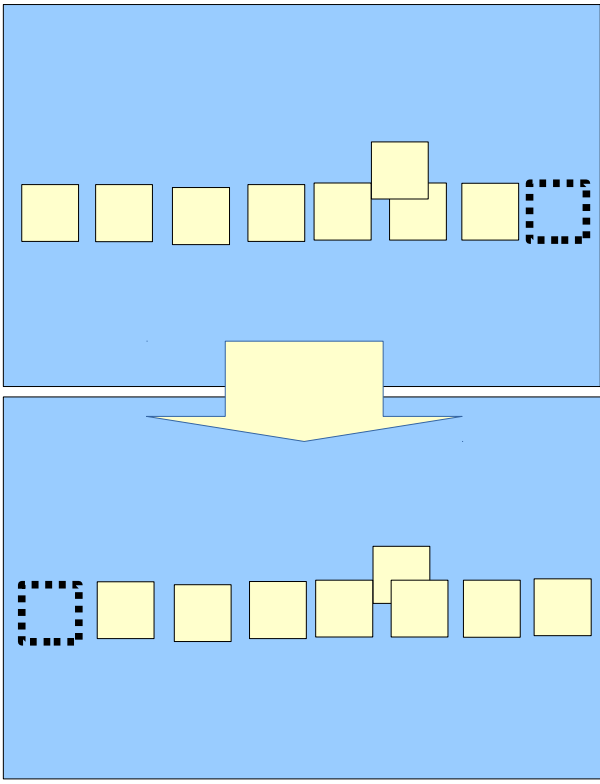

図 6: スプライトは同時に横8個

そもそも、「敷き詰めている」ので、画像（タイル）同士を重ね合わせるように配置することもできません。これでは、土管の中に隠れていくマリオなども表現できないことになります。

そこで使われるのが、「スプライト」です。スプライトを使うと、バックグラウンドのタイルと同じもの（か、それを縦に２つ並べた、8x16の画像）を、画面上の任意の位置に配置することができます図 5。これなら、放物線にそってジャンプするマリオの画像を表現できますし、１ドットずつ移動する弾の画像を表現することができます。

しかしながら、もちろん良い事ずくめではありません。まず、

3 ちなみに炭酸飲料とはもちろん別物ですが、名前の由来は同じ

スプライトは画面上に最大で64個までしか表示することができません。スプライトだけを使ってすべてを表現するには少なすぎる数で、バックグラウンドとの使い分けを考えていく必要があります。

さらに、画面の横方向には**最大で8つ**しか並べることができません。8つを超えてしまった場合、9つ目より右側のスプライトは表示されなくなってしまいます。

この制限はとても困るので、よくあるゲームでは、9つ以上スプライトがある場合、１フレームごとに交代で消して、同時には「8つ」までしか表示されないようにしています図 6。この結果、キャラクタが点滅しながら表示されるようになります。「ロックマン２」とかだとキャラクタがたくさん出るので、主人公や敵キャラが点滅しちゃう現象がよく発生しますよネ！

0x0805 スクロールレジスタでなめらかスクロール

さて、マリオの背景は「バックグラウンド」に描かれており、これはタイルを敷き詰めたものだと説明しました。しかし、これだけではマリオのように、スムーズに背景を移動させることができません。

タイルの敷き詰めだけで背景の移動を表現しようとする場合、すべてのタイルを一列ずつ移動させるしかありません図 7。これでは、ガクッ、ガクッ、っと不自然に８ドットずつ移動してしまうことになります。MSXなどの機種では、タイルを配置し直すしか背景の移動を表現する方法がなかったそうで、このようにガクガクと背景が移動していました。これを一部では「８ドットスクロール」と呼んでいます。

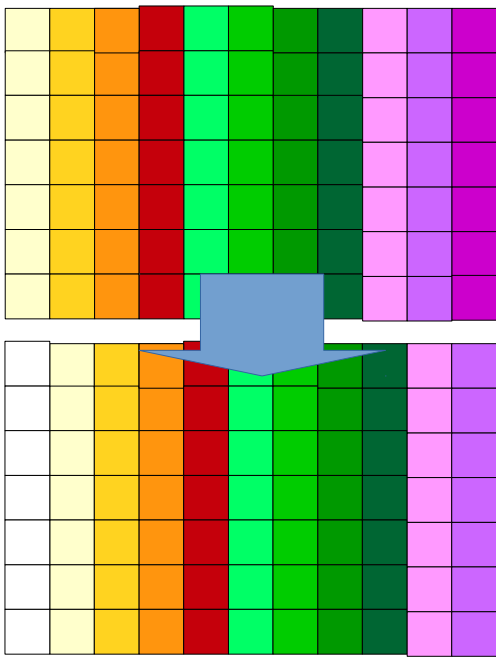

図 *7: 8*ドットスクロール

さて、もちろん、ファミコンゲームを遊んだ方なら誰でも知っているとおり、マリオは8ドットスクロールせず、綺麗に背景が移動します。このために存在するのが「スクロール機能」です。

PPUのハードウェアの機能を使うことで、背景を上下左右に１ドットだけずらして描画させることが可能になります図 8。

スクロール機能ではCPU空間アドレス0x2005の「ハードウェアスクロール」に二度（X軸とY軸）値を書き込むことで、制御することができます。

0x0806 多画面を使って自然にスクロール

さて、スクロール機能で綺麗にスクロールすることができる、とわかりました。スクロールする場合、右にスクロールしたら、右側が少し隠れ、左側が少し空いてしまいます。この部分は、どうするのでしょうか？マリオのゲームでは、ちゃんとその部分にも絵が入ってたはずです。

そのための機能が、「多画面」と「ミラーリング」です。多画面というのは、表示するための画面が、実際に表示される一面だけでなく、いくつもあることです。

例えば、横方向に並ぶ２つの画面を使い、スクロールレジスタを右にズラすように設定すると、左側にはもともとの画面Aの内容が、右側には新しい画面Bの内容が表示されます。このようにすることで、スクロールしても画面の一部を欠けたりさせることなく、綺麗にスクロールすることができるのですエラー: 参照先が見つかりません。。

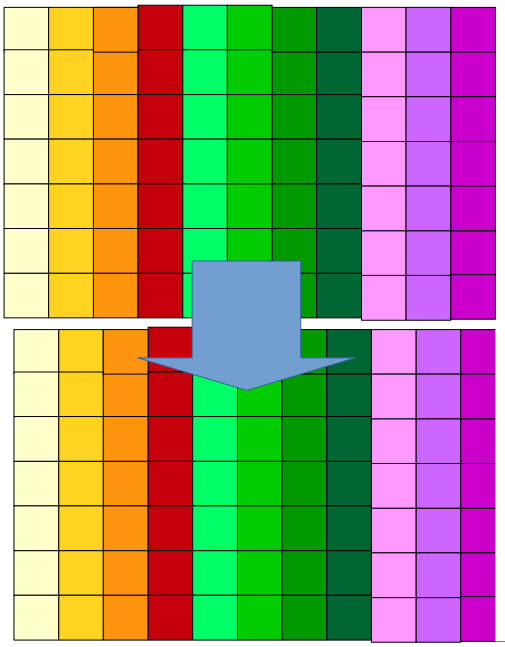

図 *8: 1*ドットだけスクロール

ファミコンは、このような画面を４つもち、大きな2x2の画面を形作っています図 10。書いてあるアドレスは、「グラフィックメモリ」上のどこに書けばバックグラウンドの内容が変わるかのアドレスです。

…が、実は本当は２つしかありません。これが「ミラーリング」で、「ホライゾンタル・ミラーリング」と「バーティカル・ミラーリング」の二種類があり、RAMの話でしたミラーリングと同様に、内容がコピーされます。

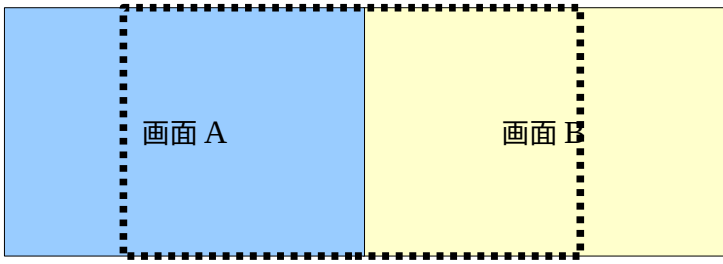

図 *9:* 画面とスクロールの組み合わせ

- ホライゾンタル・ミラーリングでは、AとBが同じ内容に、CとDが同じ内容になり、水平方向にスクロールするのにちょうどよい設定（CとDはたいてい使わない）
- バーティカル・ミラーリングでは、AとCが同じ内容に、BとDが同じ内容になり、垂直方向にスクロールさせるのにちょうどよい設定（BとDはたいてい使わない）

なんでこんな手の込んだことを？と思われるとおもいますが、たいていのゲームでは上下か左右のスクロールだけをすれば済むので（例えば、マリオは左右だけ）、その分搭載するメモリを節約するためにこのような設定になってるのだと思われます。

この設定は動的に変更することは普通できず、カートリッジごとに固定になっています。

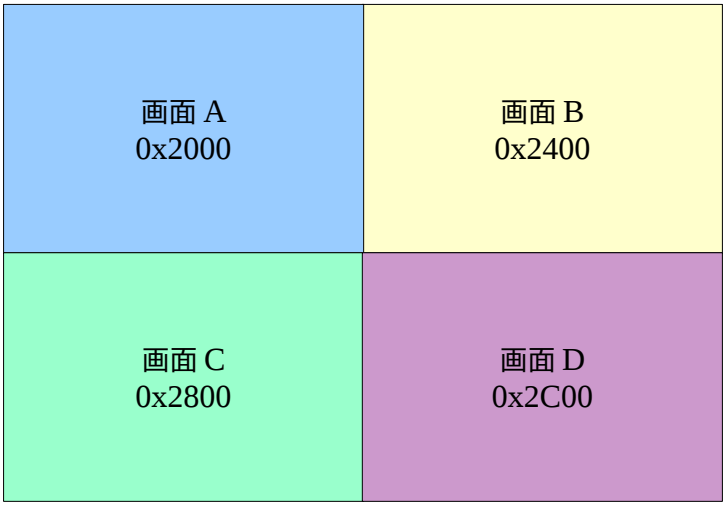

図 *10:* ファミコンの持つ４つの画面

一部のカセットはカセット内にグラフィックRAMが余分に搭載されており、これを使うことで、４つの画面に「ミラーリング」を起こさずにすべて違う画面として使うことができます。一部のRPGなどで、勇者を上下左右になめらかに冒険させたい場合などに使われていたようです。

0x0807 パレットを使って色を指定

さて、今までは「タイルのピクセルデータはカートリッジ上にある」とごまかしていましたが、どうカートリッジ上にあるのでしょうか？

現代のコンピュータでは、ピクセルデータを表すために、RGBの三色を並べるのが主流です。つまり、赤色のピクセルを表すために「0xff 0x00 0x00」という３バイトを使って１ピクセルを表現するわけです。大変わかり易いのですが、ファミコンでは１ピクセルに三バイトも使うほど余裕がありません。

その代わりに、ファミコンでは「パレット」を使います図 11。パレットは現実世界でのパレットと同じで、色を幾つか集めたものです。

タイル上のピクセルは、具体的な色を集めて表現するのではなく、パレット上のどこの色なのかを保持しています。各パレットは3色集めることができ、さらにもう一色パレットと関係なく「透明」が使えます。合計4色を使うことができますので、2ビットあれば1ピクセルの色が何なのかを指定することが

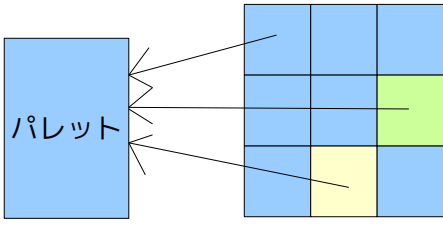

図 *11:* ピクセルはパレットへの参照を持つ

できます。3バイトで表すと24ビットですから、1/12です。大分減らせましたね。

もう一つ副次的な効果があります。それは、同じ画像の色だけ変えたものを簡単に生み出せることです。ピクセルはパレット内のどの色かの情報しか持たないので、パレットを入れ替えれば、キャラクタの色も変更することができます。「スーパーマリオブラザーズ２」までのマリオとルイージは、服の色を変えただけで同じドットで表現されているのですが、これもこのパレットの仕組みを使って実現しています。

ところで、パレットの色の指定の仕方なのですが、これもRGBではなく、「明度と色相」を使って計6ビットで指定します[4]。美術の時間が懐かしいですね。この64個の「色」に、具体的にどのような色を割り当てるのかは趣味で色々別れる所で（テレビの調整一つで変わってしまいますし）、いくつかの割り当て方が存在します[5]。

長かったですが、グラフィックの基本的な概念の説明はこれで終わりです。VBLANK、タイルグラフィック、スプライト、スクロール、ミラーリング、パレット、これぐらいが分かればあとは細かい実装上の話になると思います。

0x0808 スプライトメモリとグラフィックメモリの整理

さて、ここからはもっと細かい話にはいっていきます。まずは、多画面のところでも出てきた、グラフィックメモリや、スプライトメモリのメモリ空間について整理しましょう。

グラフィックメモリ

CPUのRAMに比べれば、あまり複雑ではありません。

「パターン・テーブル」がピクセルが実際に格納されている空間です。カートリッジに格納されているので、ここの部分はカートリッジに接続されています。

「ネームテーブル」が、それぞれの画面に表示されるタイルがどれなのかを決定します。

メモリアドレス	内容
0x0000 → 0x0fff	パターン・テーブル１
0x1000 → 0x1fff	パターン・テーブル２
0x2000 → 0x23ff	画面Aのネームテーブル
0x2400 → 0x27ff	画面Bのネームテーブル
0x2800 → 0x2Bff	画面Cのネームテーブル
0x2c00 → 0x2fff	画面Dのネームテーブル
0x3000 → 0x3eff	ネームテーブル（0x2000 → 0x2fff）のミラー
0x3f00 → 0x3f1f	パレット・テーブル
0x3f20 → 0x3fff	パレット・テーブルのミラー

表 *4:* グラフィックメモリ

「パターンテーブル」と「ネームテーブル」はこれからの説明で沢山出てきて、どっちがどっちかこんがらがると思います…が、実際にプログラムを書いてる時はまぁ区別できるので心配は不要です。

最後に残っているのはパレット・テーブルです。バックグラウンド用のパレットとスプライト用のパレットがそれぞれ４つで計8パレット24色、あと、バックグラウンドとスプライトが「何もない」ところに置かれる「背景色」の一色分が置かれています。

スプライトメモリ

スプライトメモリはもっと簡単です。アドレス空間は１バイトしかなく、目的も全部同じ、「スプ

4 PPU Palettes http://wiki.nesdev.com/w/index.php/PPU_palettes
5 NESエミュレータ用パレット http://hlc6502.web.fc2.com/NesPal2.htm

ライト情報の定義」です。

スプライトは、ひとつ４バイトで定義されます。

- １バイト目：Y座標
- ２バイト目：タイル番号
- ３バイト目：設定
- ４バイト目：X座標

座標の間にタイル番号と設定が挟まってるのがなんだか不思議な感じがするのですが、そうなっているので仕方がありません。

設定では、使用するパレット番号や、上下左右での反転、さらに、バックグラウンドの後ろ側に表示するか、表側に表示するかを設定できます。

これらの情報を使って、CPUから値を読み書きできるようにし、さらにクロックに沿って画面を１ドットずつ描画するモジュールを書けば、PPUの完成です。案外あっけないです。

0x0809 CPUと接続し、設定の読み書きとメモリ空間の受け渡しを行おう

さて、ここからはソースコードを見て行きましょう。

CPUの読み書き命令は、VirtualMachineクラスが仲介しているのでした。今まで何も書かれて痛かったCPU側メモリ空間0x2000のところに、こんな感じでVideoクラスに処理を仲介するコードを書きました。

```
inline uint8_t read(uint16_t addr) //from processor to subsystems.
{
	switch(addr & 0xE000){
		case 0x0000:
			return ram.read(addr);
		case 0x2000:
			return video.readReg(addr);
		case 0x4000:
			//略
	}
}
inline void write(uint16_t addr, uint8_t value) // from processor to subsystems.
{
	switch(addr & 0xE000){
	    case 0x0000:
	        ram.write(addr, value);
	        break;
	    case 0x2000:
	        video.writeReg(addr, value);
	        break;
	     case 0x4000:
	          //略
	}
}
```

0x2000->0x3fffのすべての空間をVideoクラスに一旦丸投げます。そのほうがパーツごとの分担ができている感じがするので、いいのではないかと思います。

さらに、Videoクラス内でここから割り振ります。[src/emulator/Video.cpp]

```
uint8_t Video::readReg(uint16_t addr)
{
  switch(addr & 0x07) {
      /* PPU Control and Status Registers */
      //case 0x00: //2000h - PPU Control Register 1 (W)
      //case 0x01: //2001h - PPU Control Register 2 (W)
      case 0x02: //2002h - PPU Status Register ®
          return buildPPUStatusRegister();
      /* PPU SPR-RAM Access Registers */
      //case 0x03: //2003h - SPR-RAM Address Register (W)
      case 0x04: //2004h - SPR-RAM Data Register (Read/Write)
          return readSpriteDataRegister();
      /* PPU VRAM Access Registers */
      //case 0x05: //PPU Background Scrolling Offset (W2)
      //case 0x06: //VRAM Address Register (W2)
      case 0x07: //VRAM Read/Write Data Register (RW)
          return readVramDataRegister();
      default:
```

```
            return 0;
        //throw EmulatorException() << "Invalid addr: 0x" << std::hex << addr;
    }
}
void Video::writeReg(uint16_t addr, uint8_t value)
{
   switch(addr & 0x07)
     {
              /* PPU Control and Status Registers */
          case 0x00: //2000h - PPU Control Register 1 (W)
                analyzePPUControlRegister1(value);
                break;
          case 0x01: //2001h - PPU Control Register 2 (W)
                analyzePPUControlRegister2(value);
                break;
          //case 0x02: //2002h - PPU Status Register ®
          /* PPU SPR-RAM Access Registers */
          case 0x03: //2003h - SPR-RAM Address Register (W)
                analyzeSpriteAddrRegister(value);
                break;
          case 0x04: //2004h - SPR-RAM Data Register (Read/Write)
                 writeSpriteDataRegister(value);
                 break;
          /* PPU VRAM Access Registers */
          case 0x05: //PPU Background Scrolling Offset (W2)
                analyzePPUBackgroundScrollingOffset(value);
                break;
          case 0x06: //VRAM Address Register (W2)
                 analyzeVramAddrRegister(value);
                 break;
          case 0x07: //VRAM Read/Write Data Register (RW)
                 writeVramDataRegister(value);
                 break;
          default:
                 throw EmulatorException() << "Invalid addr: 0x" << std::hex << addr;
    }
}
```

読み込みのみ、書き込みのみのアドレスに、正しくない方のアクセスが発生したら例外を投げるようにしていますが、たまに普通のゲームでもそういった正しくないアクセスをするゲームが散見されるので、仕方なく0を返すようにしてるところがあります。

それぞれの設定などは、Nesdevの資料[6]を参考に、ビット演算を使っていくことで処理することができます。例えば、0x2000でPPUの設定を行うところを見てみましょう。

```
inline void Video::analyzePPUControlRegister1(uint8_t value)
{
   executeNMIonVBlank = ((value & 0x80) == 0x80) ? true : false;
   spriteHeight = ((value & 0x20) == 0x20) ? 16 : 8;
   patternTableAddressBackground = (value & 0x10) << 8;
   patternTableAddress8x8Sprites = (value & 0x8) << 9;
   vramIncrementSize = ((value & 0x4) == 0x4) ? 32 : 1;
   vramAddrReloadRegister = (vramAddrReloadRegister & 0x73ff) | ((value & 0x3) << 10);
}
```

6 PPU registers - Nesdev Wiki http://wiki.nesdev.com/w/index.php/PPU_registers

基本的には、対応するビットが1になってる値とandを取り、その値と同じ結果になるかで判定できます。最後のビット演算は、描画する際に、どのネームテーブルから値を取ってくるかを決めるレジスタで、この値を変更することで表示するメインのバックグラウンド画面をどれにするのか設定しています。こんな奇妙なことになっているのは、スクロール機能との兼ね合いです。

そうそう、グラフィックメモリのデータの読みこみには、一つ注意があります。PPUとカセットやネームテーブル用RAMの間にはバッファ（一時的なデータを貯めこむ場所）が挟まっており、この影響で、二回読み出し処理を行わないと内容が読めなくなっています。

```
inline uint8_t Video::readVramDataRegister()
{
   if((vramAddrRegister & 0x3f00) == 0x3f00){
        const uint8_t ret = readPalette(vramAddrRegister);
        //ミラーされてるVRAMにも同時にアクセスしなければならない。
        vramBuffer = readVramExternal(vramAddrRegister);
        vramAddrRegister = (vramAddrRegister + vramIncrementSize) & 0x3fff;
        return ret;
    }else{
        const uint8_t ret = vramBuffer;
        vramBuffer = readVramExternal(vramAddrRegister);
        vramAddrRegister = (vramAddrRegister + vramIncrementSize) & 0x3fff;
        return ret;
    }
}
```

一度読んだだけでは、vramBufferという変数が変化するだけで、戻ってくる値は以前のvramBufferの値になっています。パレットにアクセスしたときだけはPPUの内部にあるメモリなのでvramBufferの影響を受けませんが、vramBufferにはネームテーブルの中身が入るようです。

0x080A　１行１行、１ピクセル１ピクセル丁寧に出力

これでやっとついに、画像を出すことができます。前節で設定した値に沿って、画像を出力していきます。Video#run(uint16_t clockDelta) @ src/emulator/Video.cppにその詳しい処理が記述されています。

CPUのところで、CPU命令のクロック数を正確に記録しましたね？PPUでは、その記録されたクロック数に応じて、その分だけレンダリングしていきます。1CPUクロックは3PPUクロックで、1PPUクロックで基本的には１ピクセル描画します。横のピクセル数を管理するのがthis->nowX、縦を管理するのがthis->nowYです。それを踏まえた上で、runを見てみましょう。

```
void Video::run(uint16_t clockDelta)
{
   this->nowX += clockDelta;
   while(this->nowX >= 341){
       this->nowY++;
       this->nowX -= 341;
       if(this->nowY <= 240){
           uint8_t* const lineBuff = screenBuffer[nowY-1];
           memset(lineBuff, EmptyBit | this->palette[8][0], screenWidth); //0x00: 空
           spriteEval();
           if(this->backgroundVisibility || this->spriteVisibility){
               // from http://nocash.emubase.de/everynes.htm#pictureprocessingunitppu
               vramAddrRegister =
                    (vramAddrRegister & 0x7BE0) | (vramAddrReloadRegister & 0x041F);
```

```
                buildBgLine();
                buildSpriteLine();
                vramAddrRegister += (1 << 12);
                vramAddrRegister += (vramAddrRegister & 0x8000) >> 10;
                vramAddrRegister &= 0x7fff;
                if((vramAddrRegister & 0x03e0) == 0x3c0){
                    vramAddrRegister &= 0xFC1F;
                    vramAddrRegister ^= 0x800;
                }
             }
        }else if(this->nowY == 241){
            //241: The PPU just idles during this scanline.
            //Despite this, this scanline still occurs before the VBlank flag is set.
            this->videoFairy.dispatchRendering(screenBuffer, this->paletteMask);
            this->nowOnVBnank = true;
            spriteAddr = 0;//and typically contains 00h
                           //at the begin of the VBlank periods
        }else if(this->nowY == 242){
            // NESDEV: These occur during VBlank. The VBlank flag of the PPU
            //       is pulled low during scanline 241, so the VBlank NMI occurs here.
            // EVERYNES: http://nocash.emubase.de/everynes.htm#ppudimensionstimings
            // とあるものの…BeNesの実装だともっと後に発生すると記述されてる。詳しくは以下。
            // なお、$2002のレジスタがHIGHになった後に
            //VBLANKを起こさないと「ソロモンの鍵」にてゲームが始まらない。
            // (NMI割り込みがレジスタを読み込みフラグをリセットしてしまう上、
            // NMI割り込みが非常に長く、
              //クリアしなくてもすでにVBLANKが終わった後に返ってくる)
            //nowOnVBlankフラグの立ち上がり後、数クロックでNMIが発生。
            if(executeNMIonVBlank){
                this->VM.sendNMI();
            }
            this->VM.sendVBlank();
         }else if(this->nowY <= 261){
           //nowVBlank.
        }else if(this->nowY == 262){
            this->nowOnVBnank = false;
            this->sprite0Hit = false;
            this->nowY = 0;
                if(!this->isEven){
                    this->nowX++;
                }
            this->isEven = !this->isEven;
            // the reload value is automatically loaded into the Pointer
            //                    at the end of the vblank period (vertical reload bits)
            // from http://nocash.emubase.de/everynes.htm#pictureprocessingunitppu
            if(this->backgroundVisibility || this->spriteVisibility){
                this->vramAddrRegister =
                          (vramAddrRegister & 0x041F) | (vramAddrReloadRegister & 0x7BE0);
            }
        }else{
            throw EmulatorException("Invalid scanline") << this->nowY;
        }
    }
}
```

クロック数をカウントし、適切なタイミングに適切な描画（spriteEval、buildBgLine、builsSpriteLine）を行いつつ、CPUに対してsendNMIとしてNMI割り込みを発生させる…といった感じのコードになります。

描画の具体的な内容については、やっぱりNesdevを見ると詳しいタイミング情報付きで載っています[7]。スプライトの表示・非表示を決定するSpriteEvalという処理も大事です[8]。もし実装に困ったら、JavaScriptエミュレータのソースは読みやすかったので、参考にしてみてください（ただし、スプライトの処理は正確でないので気をつけて）。

0x080B PPUのデバッグは？

PPUはやはりグラフィックを扱う以上、CPUのときのようにトレースログを追いかけるだけ、というような機械的なデバッグは難しいです（出力した画像を１ピクセルずつ比較するようなツール作ればよいのかもしれませんが、そこまでやるのは面倒臭いです）。

延々実装して、よくテストされてないコードを積み上げてからデバッグするのは大変です。まずは、スプライトかバックグラウンド、どちらかのシステムだけ作ってデバッグするのがお勧めです。

フリーソフトウェアでも、自分でお持ちの実物のカセットでも良いので、タイトル画面は比較的優しい気がするので、タイトル画面を綺麗に出力するのを目標にしてみるのはどうでしょうか？

その際、いくつかソフトがあるなら、 マリオブラザーズやテニス、サッカーなどのタイトル画面が**スクロールしない**ようなゲームでちゃんと画面が表示されるのを確認した上で、ゼビウスみたいに**微妙にスクロールするゲーム**でスクロール機能の動作を確かめ、最後に「スーパーマリオブラザーズ」のような、複雑（で綺麗）なタイトル画面の表示にチャレンジしてみるとよいと思います。

さすがに画像が出てくると、ちょっと感動できるのではないでしょうか。ぜひチャレンジしてみてください！正月三が日はエミュレータ！！

7 PPU rendering http://wiki.nesdev.com/w/index.php/PPU_rendering
8 PPU sprite evalutation http://wiki.nesdev.com/w/index.php/PPU_sprite_evaluation

0xFF To be continued...

ああ、やっぱり今回も間に合わなかったよ。この本は２年ほど前に書いたものを改定しつつ、PPUの章を追記したものです。出来れば、最後まで書きたかったのですが、残念、時間の壁は、破れない。

そのPPUも、現代とは違う概念の説明が多く、ソースコードの詳しい解説や、実装上のテクニックについてあまり言及できなかったのがとても残念です。次参加するコミケでは今度こそ完成させるんだ…（フラグ建築士）。

残りの章はこんな感じです。

- 0x09 ついにコントローラだ！遊べるんだ！
  - ここまで来ると、音無しですがゲームを遊ぶことができます！感動もひとしお。
  - 結構簡単なので、せめてここまでは書きたかったのですがね…。
- 0x0A APU！矩形波！三角波！ノイズ！ピコピコサウンド！
  - APU（オーディオ処理プロセッサ）です。
  - 音感が無い人間が作ると、デバッグが絶望的です。…頑張りましょう。
  - 綺麗なファミコンサウンドが出ると、ここでもまたちょっと感動しちゃいますよ！
- 特殊マッパー
  - 容量を増やすためのマッパーだけではなく、割り込みを起こすマッパーなども実装してみましょう！
  - 割り込みタイミングのシビアさに泣いて頂きます（ぉ
  - ファミコンで綺麗なラスタースクロールがっ！動く！！

こんな感じです♪ぜひ次こそは完成させたいです（二度目）

エミュレータはソフトなマホウ。

～ハード解析なしで作るファミコンエミュレータ～

written by: ψ（プサイ）

Blog: 「ψ（プサイ）の興味関心空間」http://ledyba.org/

Mail: psi@ledyba.org

Twitter: @tikal