

LESSON 3

In this lesson we're going to refine our NMI script a bit, learn how to work with tables, make a simple loop function in ASM, and take a cursory glimpse at comparing different number systems.

You'll notice that as we do this, we're jettisoning the NESmaker GUI for generating palettes and doing it all by hand. The NESmaker GUI was intended to simplify a lot of the more cumbersome or rather nebulous things, like what we're doing right now, so that a user could jump straight to the creative part of creating a game. However, learning the inner workings can help advanced users find ways to bend the tool to their needs and make it a valuable part of their workflow, even if they don't want to start with templates. An advanced user can set up their own template that makes use of any of the parts of the software that makes their workflow easier, find creative ways to integrate with exports from other software in their tool chain, and utilize whichever parts of the GUI that aid in their development and speed up the more cumbersome parts of the process.

We'll start this lesson by refining our NMI. Right now, we're wasting a lot of unnecessary cycles, and as we discussed in the last lesson, our time is very precious during vBlank time when the NMI fires, so we need to save every cycle that we can!

Step 1: Writing to 2007 through incrementing rather than direct writes.

In the last lesson, we wrote our palette data directly to vRam addresses. We did this one at a time. We wrote the high byte of the address, then the low byte of the address, then the value. And this is a great concept to understand, because it will be quite common for us to need to write to 16bit addresses this way. However, every time we do this it takes up a certain number of cycles, and in this case, it takes up needless cycles.

The reason I say needless is because when we do a write to \$2007, the address to which we're pointing to automatically increments to the next address. So after we have established the address by writing the high byte to \$2006 and the low byte to \$2007 and the value to \$2007, it automatically increases that low byte to the next incremental value.

Basically, it's the mailman standing at the mailboxes. He puts the contents in mailbox 1 and then closes the mailbox door. Then, he automatically moves to mailbox 2. He doesn't explicitly need to be told to move on to mailbox 2. He moves to mailbox 2 automatically. He's done delivering the mail to box 1, so he moves to the next box without needing instruction to do so, unless for some reason we tell him to behave otherwise.

So with this in mind, let's refine our code and write our four values, but only point to the address before the first write. From there, we can just crank on writes to \$2007 without having to invoke the address again.

```
*C:\Users\NES pc\Desktop\NESmaker_4_5_9\NESmaker_4_5_x\GameEngineData\Routines\BASE_4_5\System\NMI_Tutorial.asm - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
NMI_Blank.asm NMI_Tutorial.asm pong2.asm
31          C_GOLD = #$28
32          C_PEACH = #$26
33          C_ORANGE = #$17
34          C_OLIVE = #$28
35          C_PINK = #$24
36          C_LIGHT_BLUE = #$31
37          C_LIGHT_GREEN = #$39
38          C_LIGHT_GRAY = #$10
39
40          ;;; push a color value to vRam
41          ;;; TWO WRITES TO $2006 SET THE ADDRESS
42          LDA #$3f ;; the high byte of the destination
43          STA $2006 ;; gets written to $2006 first
44          LDA #$00 ;; the low byte of the destination
45          STA $2006 ;; gets written to $2006 second
46          ;; WE ARE NOW ABOUT TO WRITE TO $3f00
47          ;;; WRITE TO $2007 SETS THE VALUE OF THAT ADDRESS
48          LDA #C_PURPLE ;; the color value
49          STA $2007 ;; gets written to $2007
50          ;;; AFTER THE WRITE, it automatically increments the low byte of the destination
51          ;; WE ARE NOW ABOUT TO WRITE TO $3F01
52          LDA #C_GREEN ;; the color value
53          STA $2007 ;; gets written to $2007
54          ;; WE ARE NOW ABOUT TO WRITE TO $3F01
55          LDA #C_GOLD ;; the color value
56          STA $2007 ;; gets written to $2007
57          ;; WE ARE NOW ABOUT TO WRITE TO $3F01
58          LDA #C_BROWN ;; the color value
59          STA $2007 ;; gets written to $2007
60
61
62
63
```

Here, I've commented the code changes. You can see at line 42 (it's ok if your lines of code don't line up exactly, they may be slightly different depending how you spaced it differently than me - it has no bearing on the end result) we write `#$3F`, store it to `$2006`. This sets the high byte of the address. Then we write `#$00`, store it to `$2006`, which sets the low byte of

the address. Whatever we write to \$2007 next will be what gets put into that PPU address.

But then, instead of just repeating the process to set the address one higher, the write to \$2007 automatically writes to one address higher, which would be \$3f01.

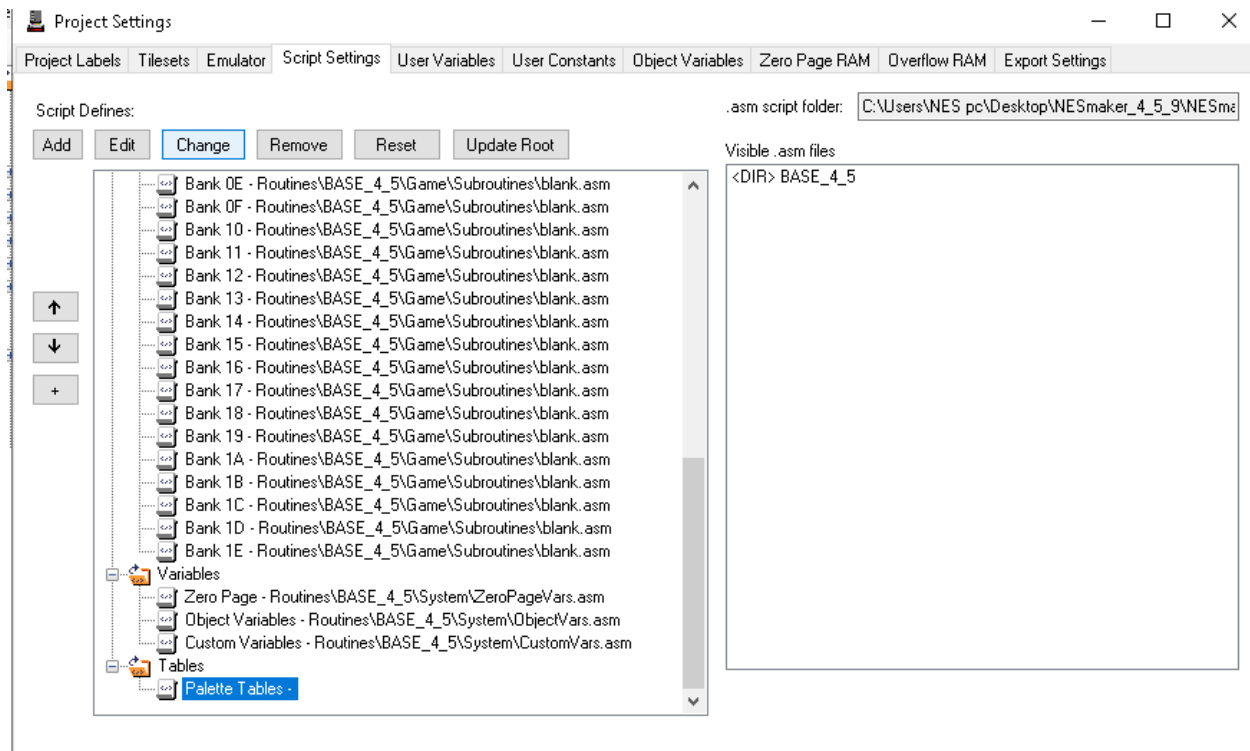
If you test your game and take a look at the PPU, you'll notice that functionally, this is no different in the resulting ROM. However, it is a highly optimized way of writing it, and gets rid of unnecessary code. Our \$2006 address pointer will continue to increment after writes to \$2007 until we point it somewhere else.

Knowing that we have 16 background palette slots and 16 sprite palette slots, this code could get unruly. Especially considering that our pointer address will increment with every write to \$2007, we could create a table with our colors laid out in order, and then use a logical loop to just blast through putting them in their place.

STEP 2: Make a script define for a color table

Just to get practice doing it, we're going to make a script group for tables, and a script define for palettes, which is where we'll place our table. This is not a necessary step in getting the game to function how we want it to, but it will help us with organization so we can find what we need quickly when our code starts to grow and become overwhelming to hunt through.

In the Script Settings tab of our Project Settings, click the + symbol to add a new group node at the bottom of the list. Right click and rename this group node Tables. Then, with that selected, click add at the top of the dialog box to add a new script define. Call it Palette Tables and set the define to SCR_PALETTE_TABLES. We don't yet have a script to attach.

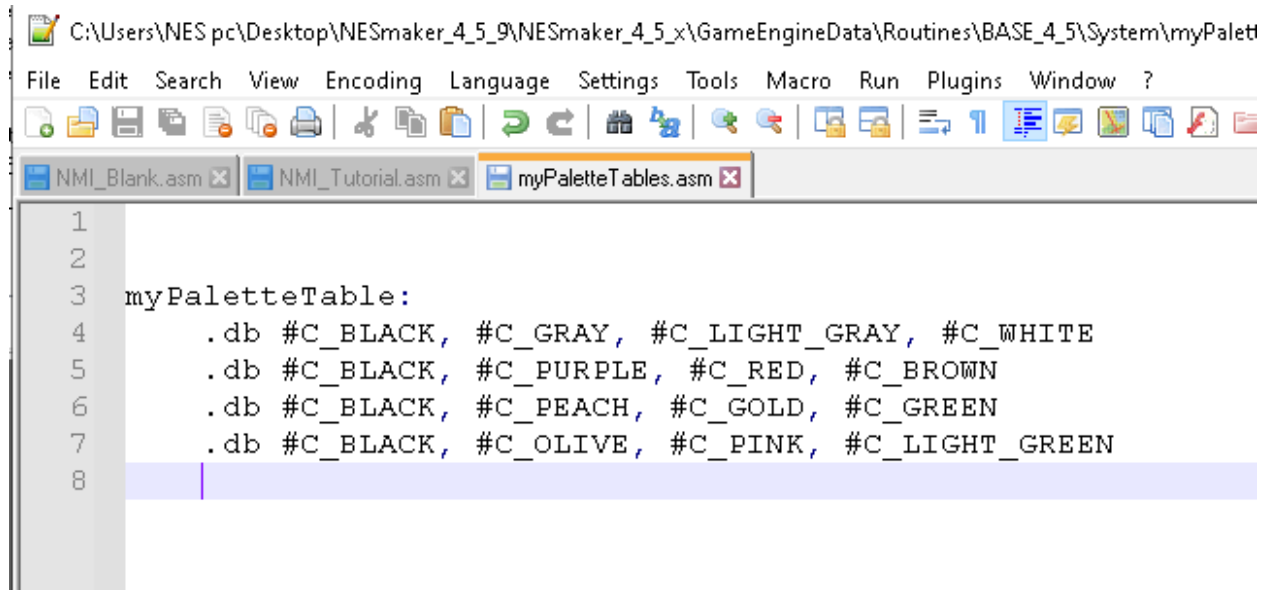


Step 3: Make a color table script.

In Notepad++ (or your script editor of choice), make a new file. This will contain our palette tables. We are going to create a label for our table and fill the table with 16 values, split into four groups of four.

Use the label `myPaletteTable`. Make sure that every row starts with a `.db`, make sure that every value has a comma between it except at the end of lines where no comma is needed. If written like this, the program will see this as 16 consecutive values in `myPaletteTable`.

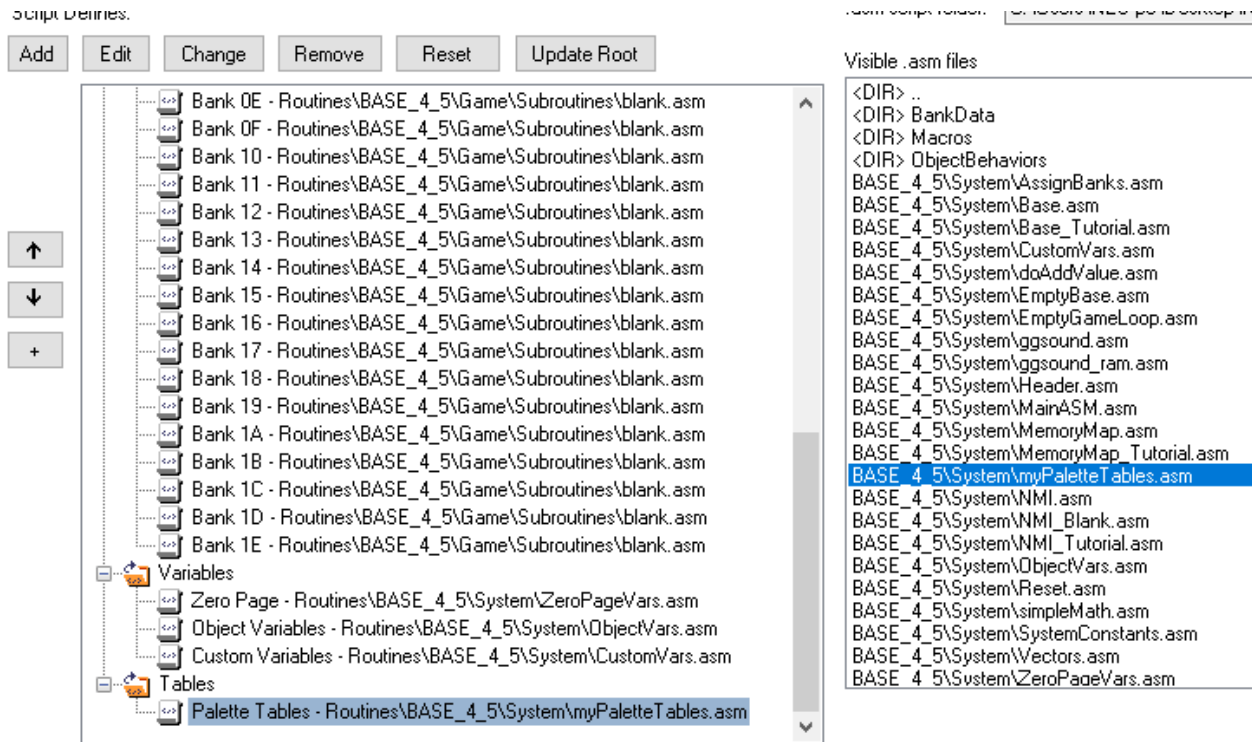
Save the file as `myPaletteTables.asm` to your project's System folder - make sure that it saving as type Assembly Language File. Saving it to the System folder is arbitrary, and it may be possibly it would be better served in the Game folder, but for now we're going to keep all these scripts we're creating for this instructional in one place.

A screenshot of an IDE window titled 'myPaletteTables.asm'. The window shows a list of files: 'NMI_Blank.asm', 'NMI_Tutorial.asm', and 'myPaletteTables.asm'. The main editor area contains assembly code for a palette table. The code is as follows:

```
1  
2  
3 myPaletteTable:  
4     .db #C_BLACK, #C_GRAY, #C_LIGHT_GRAY, #C_WHITE  
5     .db #C_BLACK, #C_PURPLE, #C_RED, #C_BROWN  
6     .db #C_BLACK, #C_PEACH, #C_GOLD, #C_GREEN  
7     .db #C_BLACK, #C_OLIVE, #C_PINK, #C_LIGHT_GREEN  
8
```

The colors that you use for this are arbitrary. You can use any colors from the list of constants that you created, but it's important to keep the first color of each of these rows of four the same. This is just a part of how the system operates. The first color is a common color that all sub palettes will share. If you were to change the first value in the last row to `#C_RED`, it would change all of the first row values to `#C_RED`. The best thing to do is just keep them all the same, and write your palettes like this so that's very easy to see.

Step 4: Attach the tables script to our script define.



Step 5: Add this new table to our code.

Now we have to figure out where to put this script in our body of code. And admittedly, there isn't much in our body of code right now. To understand where we should (and maybe more importantly, should not) put it, we'll have to consider our current existing code.

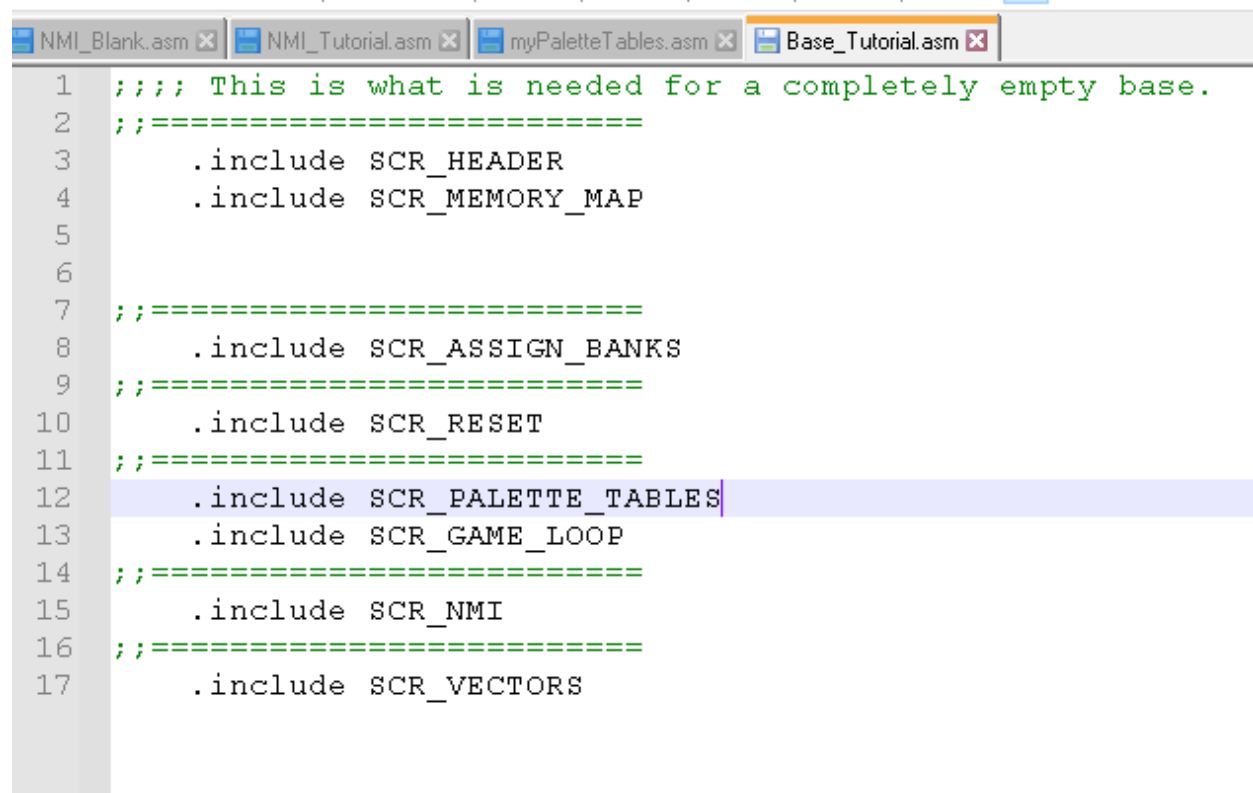
Go to your script settings and edit the Base script to open it in your code editor. This is the backbone of our entire game. We can see our memory map at the top. We can see where our NMI sits. We can imagine if we just replaced all of these includes with the script that they define, that would be what our game looks like in code. These are just references to those files, broken out through NESmaker's script editor so we can just focus on what we want to focus on at a time rather than have a complicated long string of endless code to sort through.

But where in this list should we include our new palette table? There is no single answer that is right, but there are answers that are wrong. Generally, we want to place our tables in the same section as our game loop. To understand why, it's important to understand what the

SCR_ASSIGN_BANKS did for us. In simplest terms, it sets up the content for all of our possible swappable memory banks to be put at address \$8000 when they are swapped in, and then sets a directive to flow to the beginning of our static bank, which is located at \$C000. Here is where our game code starts, at memory address \$C000. If you were to look at the code associated with the assign banks script, you'd see that the last line of it is .org \$C000. This tells our program that subsequent data will be placed starting at \$C000 unless we tell it otherwise.

By including our code after this .org directive, we are ensuring that it is placed in our static bank so that we can easily reference it.

For the time being, place an include to this script just before the Main Game Loop. As this project grows, we will be doing an entire initialization step here, but for now we can just include the palette. This will make sure that it's included in our static bank, and we'll be able to reference it at all times from the NMI. Make sure to save the file.



```
1  ;;;; This is what is needed for a completely empty base.
2  ;=====
3      .include SCR_HEADER
4      .include SCR_MEMORY_MAP
5
6
7  ;=====
8      .include SCR_ASSIGN_BANKS
9  ;=====
10     .include SCR_RESET
11 ;=====
12     .include SCR_PALETTE_TABLES
13     .include SCR_GAME_LOOP
14 ;=====
15     .include SCR_NMI
16 ;=====
17     .include SCR_VECTORS
```

Step 6: Make a loop to load colors during the NMI.

So now that we know our palette table is accessible to us, we are going to write a loop to load values in the NMI. To test it, we're just going to loop through the first four values (the remaining will be arbitrary garbage). When we're finished, if you set yours up to be the same colors as mine, you should see a gradient - black, gray, light gray, and white in the first four color slots of the palette viewer in the emulator.

In our NMI script, we're going to keep those initial writes to \$2006, because we still want that to be the starting address of our first write. However, as far as what to actually write, we want to use the first table value. Then we want to get the next table value and write it, then the next and write it.

So here's the logic of what we want to do.

- ACTION A: Get the starting address of our first mailbox we want to deliver a letter to.
- ACTION B: Hold up zero fingers (in NES speak, zero will count as a *first* of something. Our counting of things will almost always start at zero, not 1).

START A LOOPING ACTION...

- ACTION C: Our fingers being held up will determine what mail we should grab. Right now, my hand has zero fingers, so look for the letter addressed to mailbox zero.
- ACTION D: Place it in the zero mailbox. Automatically after closing this mailbox, I move to the next mailbox.
- ACTION E: Increase how many fingers are being held up. Is it four yet? If so, jump to END LOOPING ACTION. But if I don't see four fingers yet, jump to ACTION C, the beginning of the loop and repeat, with that one more finger raised than last time

around.

END LOOPING ACTION

Looking at that in assembly code looks like this.

```
35          C_PINK = #$24
36          C_LIGHT_BLUE = #$31
37          C_LIGHT_GREEN = #$39
38          C_LIGHT_GRAY = #$10
39
40          ;;; push a color value to vRam
41          ;;; TWO WRITES TO $2006 SET THE ADDRESS
42          LDA #$3f ;; the high byte of the destination
43          STA $2006 ;; gets written to $2006 first
44          LDA #$00 ;; the low byte of the destination
45          STA $2006 ;; gets written to $2006 second
46                  ;; WE ARE NOW ABOUT TO WRITE TO $3F00
47
48
49          LDX #$00
50          doLoadPaletteLoop:
51              LDA myPaletteTable,x ;; load the value from
52                                  ;; myPaletteTable, but
53                                  ;; with the offset of whatever
54                                  ;; value is in the X register
55          STA $2007 ;; Write it to vRam
56          INX ;; increase the x register
57          CPX #$04 ;; did X hit 4 yet?
58          BNE doLoadPaletteLoop ;; if it did not, do the loop
59                                  ;; again. If so, flow forward.
60
61
62
```

- The writes to 2006 set up our first address that we want to deliver a letter to.
- LDX #\$00 tells the program to load zero into the x register. This is telling the system we are holding up zero proverbial fingers.

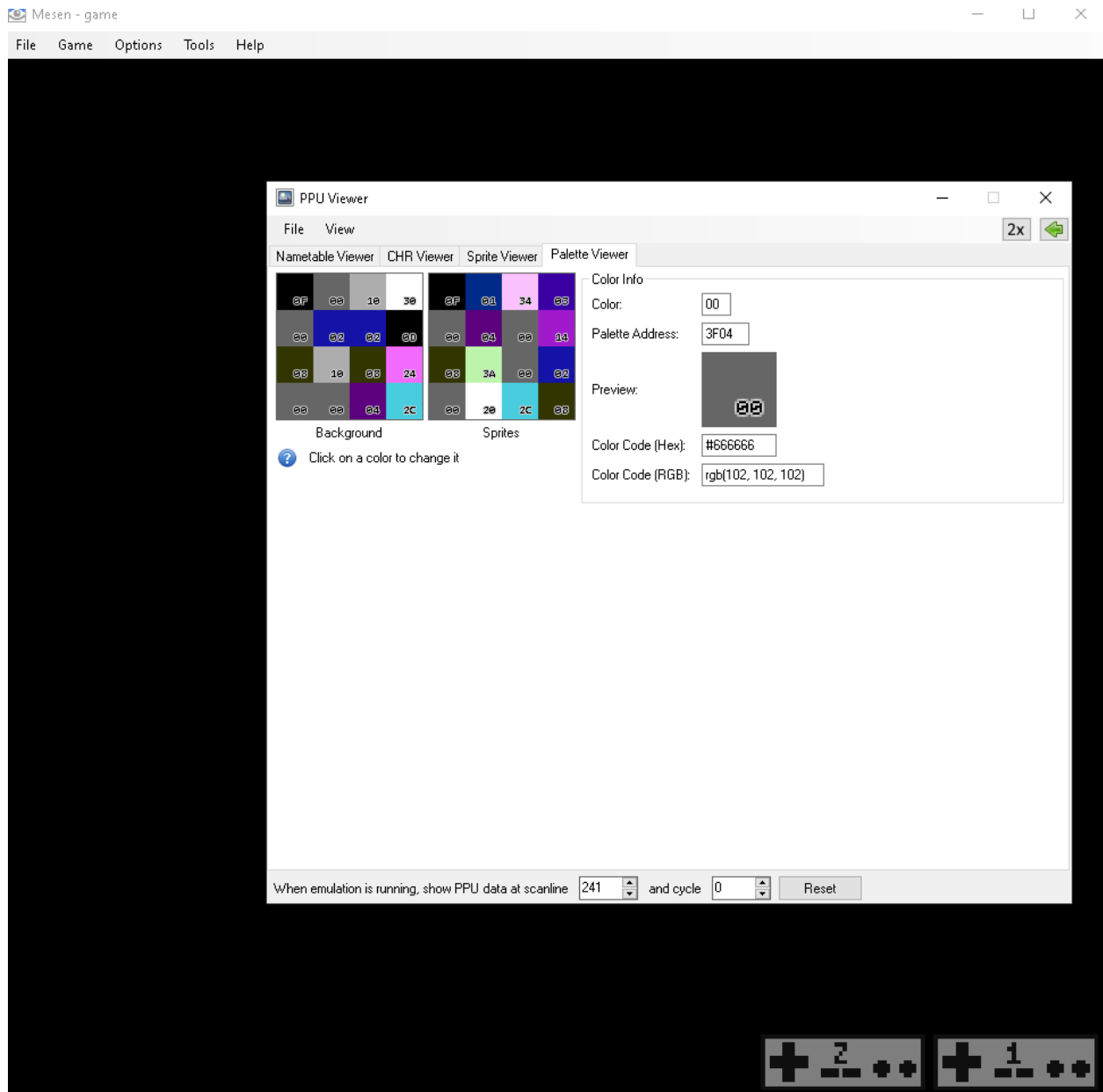
- doLoadPaletteLoop is a label that will represent the start of our loop. When we hit the point where we want to start the loop over, this is where we will jump back to.
 - Look at the table myPaletteTable, but increased from the start of that table by an offset of X. So if x is 0, it would be the first value in that table. If x is 1, it would be the second value in that table. If x is 2, it would be the third value in that table. The X register is keeping track of how many times we have gone through the loop so far...how many metaphorical fingers we are holding up. Take the number at that spot, and put it in the accumulator (grab it so we're ready to deliver it to the mailbox).
 - We store what is in the accumulator. We deliver it to the proper mailbox. Then, automatically by doing so, our program knows to shut that mailbox and move to the next one.
 - Increase the value in the X register by 1 (raise one more finger). Now, compare X to the number four (look at your fingers - are four of them held up yet?).

BNE is an ASM instruction that means B(ran)ch if N(ot) E(qual). This tells the program to branch to the label doLoadPaletteLoop (Jump to the beginning of the loop) if the previous condition (did x equal 4?) was not met yet.

Otherwise, it just ignores doing anything and continues on down the code.

So if you work the logic through, this will load the first value in the myPaletteTable into the initial palette slot (0), loop back to load the second value of the table into the second palette slot (1), loop back to load the third in the third slot (2), loop back to load the fourth in the fourth slot (3), and then when the x register increases one more time it will be 4, so the comparison to 4 will cause it to leave the loop.

If we run the game and hit control P to enter the PPU viewer in MESEN, we should see our black to white gradient in the first four color slot (the rest is all garbage values so just ignore for a moment).



That's great, but we made a table that is 16 slots long. We should be able to fill up all four rows of all four columns by just changing the number of times we cycle through the variable. We don't want to stop if we have four

fingers up. We want to stop if we have 16 fingers up (or I suppose ten fingers and six toes for most of us).

Step 7: Extend the loop to run 16 times.

Here, we're going to change the line `CPX #$04` to `CPX #$10`. For those who have never worked with hex values, `#$10` is a hex value with the decimal equivalent of 16. The first time working with hex values can be a bit confusing, but it's conceptually easy to understand even if it takes some getting used to to master.

When you're using hexadecimal math, you're working with a base 16 number system, whereas decimal is a base 10 number system. That means that in the regular decimal math that you're used to, numbers go from 0-9 (ten values), and when they increase again, we add one to the higher place value and start over again (10-19; 20-29; 30-39, etc).

With hexadecimal, numbers go from 0-f (sixteen values), and when they increase again, we add one to the higher place value and start over again (10-1f; 20-2f, 30-3f, etc).

Translating back and forth can seem weird at first, but as you work with hexadecimal numbers, certain ones become very obvious. But here's a quick chart to show how it works:

```
DEC 00 - 01 - 02 - 03 - 04 - 05 - 06 - 07 - 08 - 09 - 10 - 11 - 12 - 13 - 14 - 15
HEX 00 - 01 - 02 - 03 - 04 - 05 - 06 - 07 - 08 - 09 - 0A - 0B - 0C - 0D - 0E - 0F
```

After 0F, the hex values would start over as 10. You can see the next value on the decimal chart would be 16. This is why `DEC 16 = HEX 10`.

Hexadecimal numbers are written `#$xx`. If you ever are confused about hexadecimal numbers, though, you can always just write `#xx` to work in decimal. If you were to write `#16`, that would mean decimal value sixteen. If you were to write `#$10`, that would be the equivalent, also meaning sixteen.

So why work in HEX at all? Well, for one thing, it is really handy in terms of working with a system like the NES, which works entirely in 8-bit. An 8-bit

number is a number that is made up of 8 individual bits, each of which can be flipped to a zero or a 1. You can also write numbers in binary, which reflects the bits. In binary, the numbers one through fifteen would be written like this:

DECIMAL	HEXADECIMAL	BINARY
0	00	00000000
1	01	00000001
2	02	00000010
3	03	00000011
4	04	00000100
5	05	00000101
6	06	00000110
7	07	00000111
8	08	00001000
9	09	00001001
10	0A	00001010
11	0B	00001011
12	0C	00001100
13	0D	00001101
14	0E	00001110
15	0F	00001111

The total number of possible combinations for a single byte, which is made up of 8 bits, is 255, which is expressed in binary as `#%11111111`, which is expressed in hexadecimal as `#$FF`.

Now, with binary, there are no other possible combinations. Each of those places can only be occupied by a one or a zero. With decimal numbers, though, it doesn't quite translate. The next value would be 256, then 257, then 258, etc...and we wouldn't be thinking about the next place value until 999.

But HEX is an interesting middle ground. Since each place value can go from 0-F, with F being the 15th and final value possible for each place value, `#$FF` represents the highest 8-bit number. Adding one to FF sets it back to 00 (and adds one to the upper place value). It is easier to read than binary (for most people), but it also more quickly conveys a relationship to the 8 bit number than decimal.

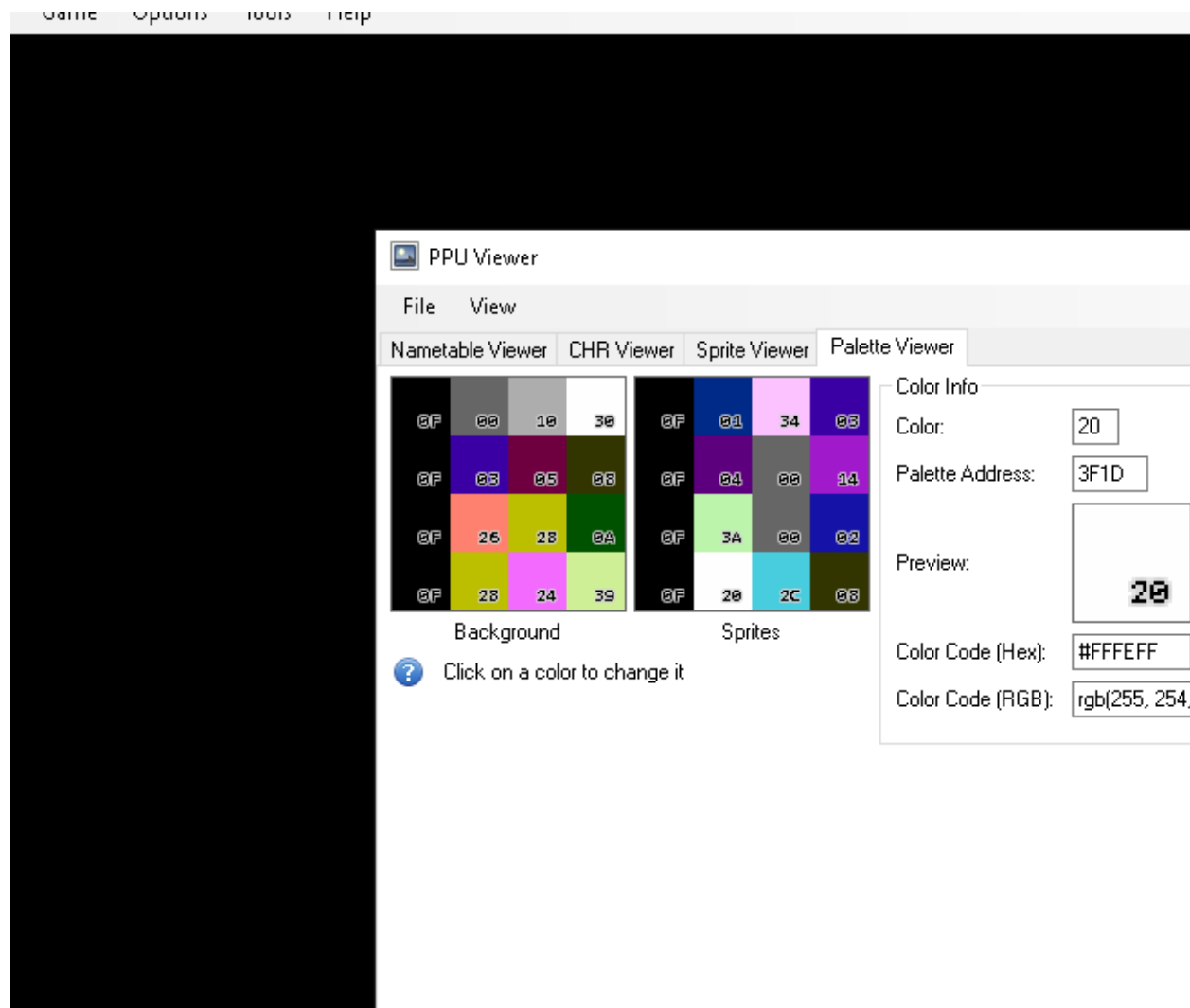
The more you work with it, the more intuitive it becomes, and the more you realize why it's very valuable when working in 8-bit math to understand how to play with hex numbers. But as far as the program understands it, `#16 = #$10 = #%00010000`...they all equal "sixteen". Feel free to put any of them as your comparison for X and they'll work the same.

```

41      ;; TWO WRITES TO $2006 SET THE ADDRESS
42      LDA #$3F ;; the high byte of the destination
43      STA $2006 ;; gets written to $2006 first
44      LDA #$00 ;; the low byte of the destination
45      STA $2006 ;; gets written to $2006 second
46              ;; WE ARE NOW ABOUT TO WRITE TO $3f00
47
48
49      LDX #$00
50      doLoadPaletteLoop:
51          LDA myPaletteTable,x ;; load the value from
52              ;; myPaletteTable, but
53              ;; with the offset of whatever
54              ;; value is in the X register
55          STA $2007 ;; Write it to vRam
56          INX      ;; increase the x register
57          CPX #$10 ;; did X hit 16 yet?
58          BNE doLoadPaletteLoop ;; if it did not, do the loop
59              ;; again. If so, flow forward.
60
61
62
63
64

```

Now if you save and test your game, you should see that your entire slate of 16 colors fill the proper slots in MESEN's palette viewer.



END OF LESSON 3

Make sure to save your project. By this point, you should know a bit more about NES's memory allocation, how to write to vRam during NMI, how to create and read from table data, a little bit about the X register and how it works, and how to construct a simple loop in ASM.