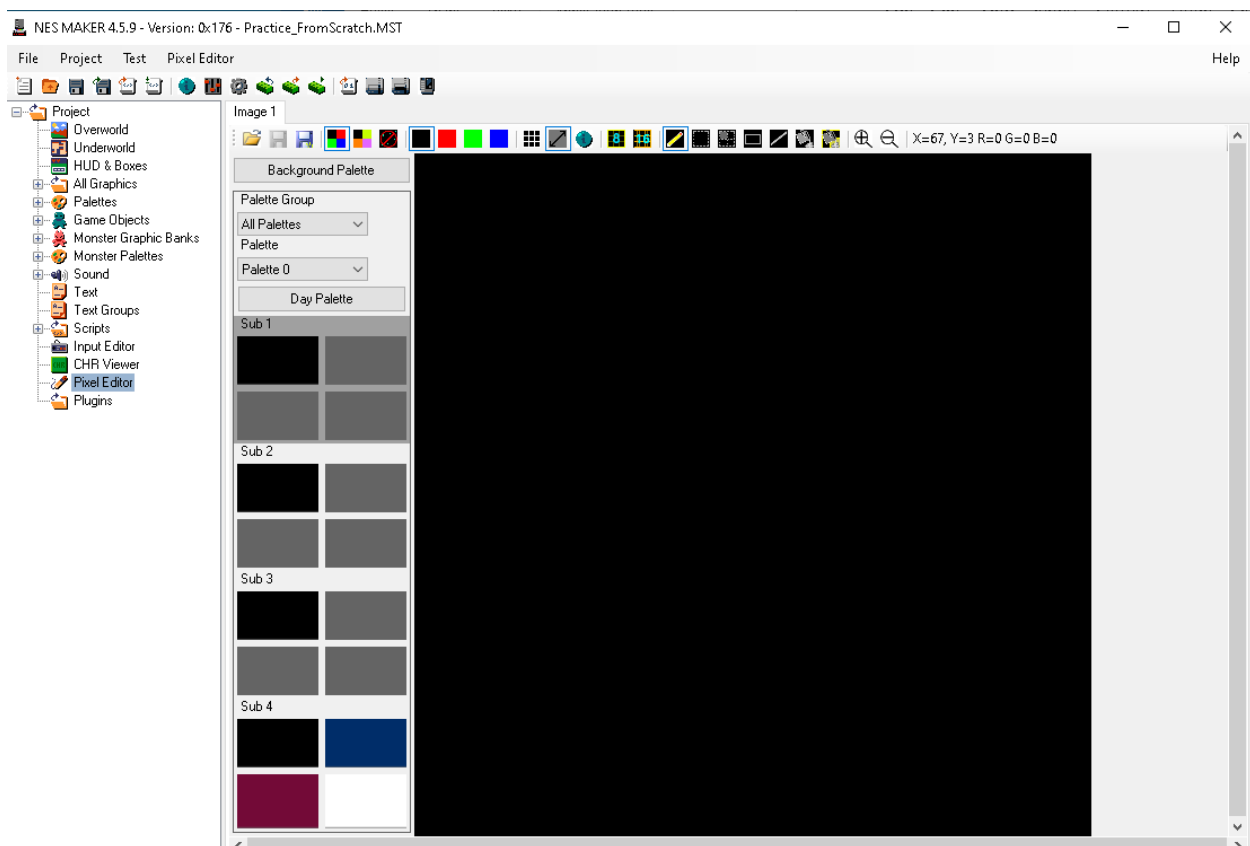


LESSON 4

In this lesson, we're going to start adding graphics to our game. We'll use the NESmaker pixel editor to create and modify graphics and export them as CHR files, however, we will create our own logic systems, functions, and methods to add those to our game rather than using NESmaker's default tileset system. What this will mean is that some of the GUI interfaces will no longer function correctly, since we're deviating from how they are set up. By doing it this longhand way, we'll get a better understanding of how those tileset systems work and why they are set up the way that they are.

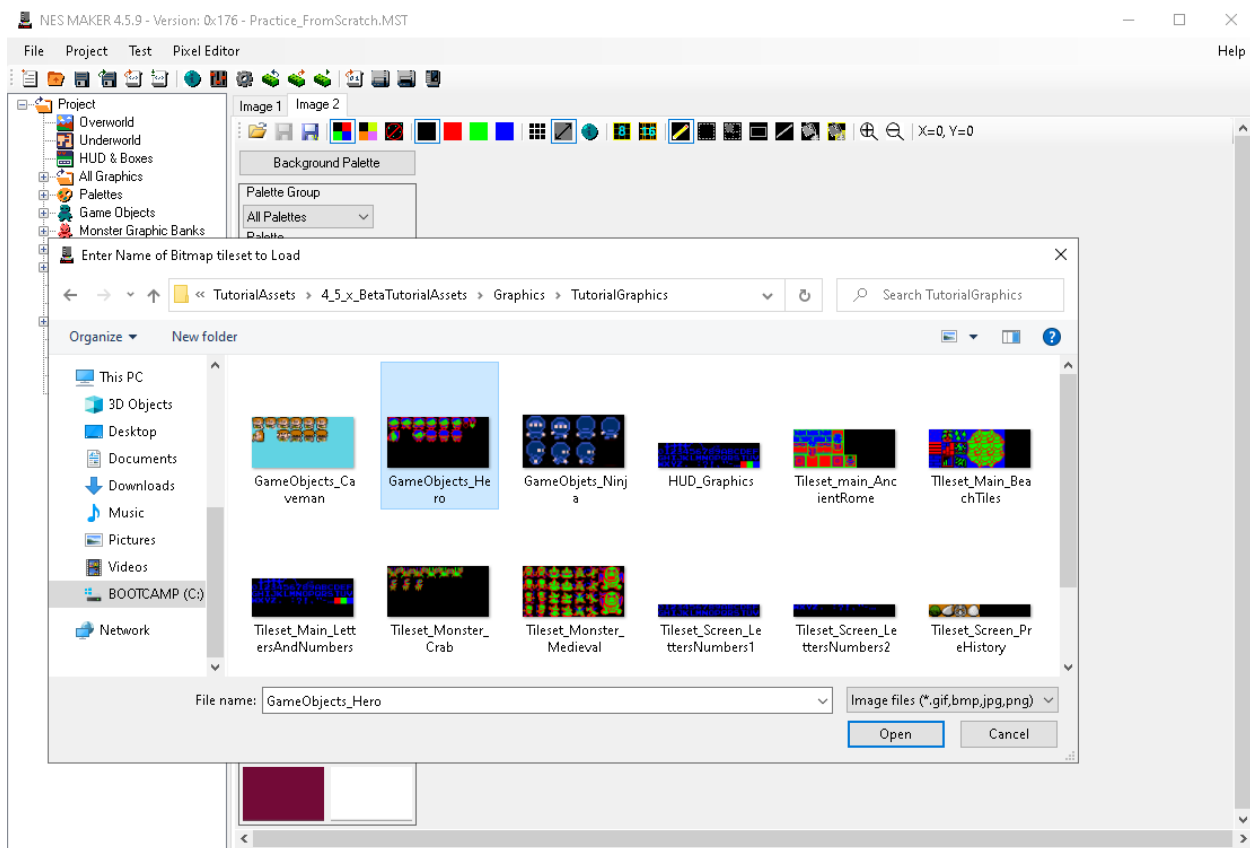
Step 1: Open a new canvas in the Pixel Editor.

Click on the Pixel Editor node in the project hierarchy to open up the pixel editor tool. From the Pixel Editor window that appears at the top of the screen, choose New BMP ->128x128 (Full Tileset). This will create a canvas the exact size of a vRam page for graphics. We'll start by creating our sprite objects.



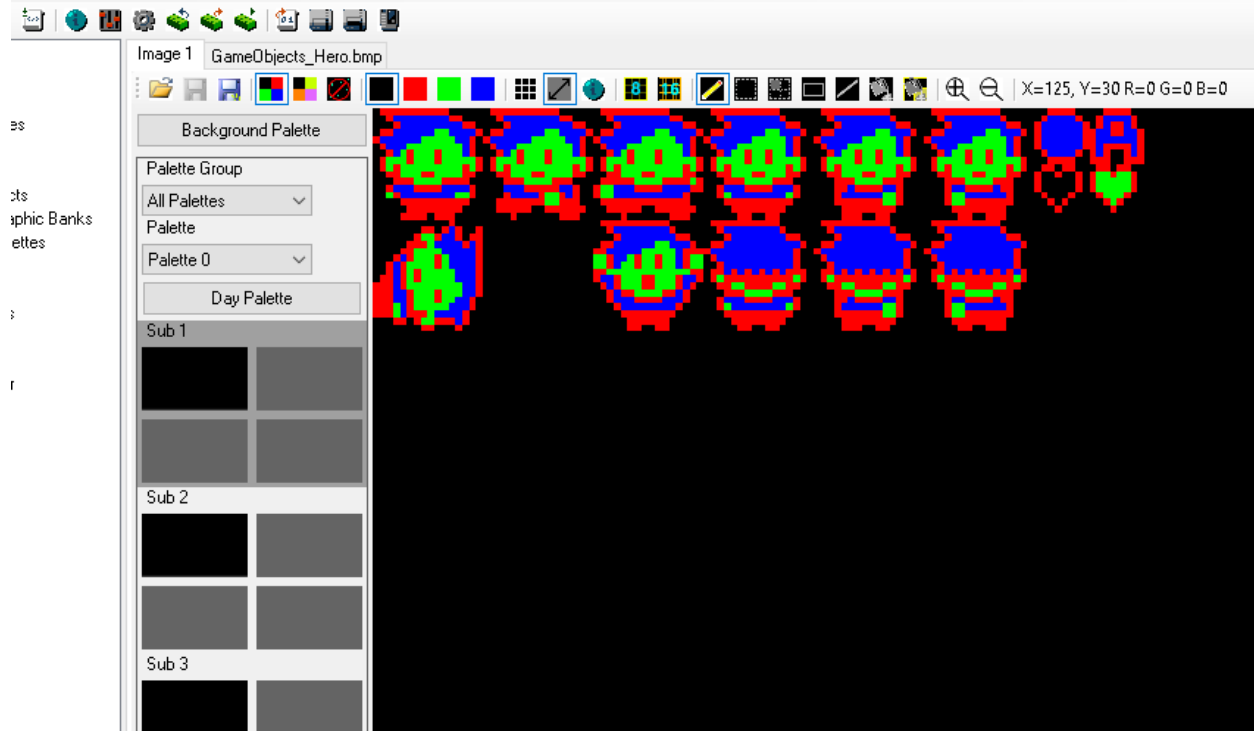
Step 2: Open a tab for existing hero graphics.

From the Pixel Editor menu, click on Add Tab. With that new tab selected, click on the open icon in the Pixel Editor tool bar. Navigate to Root \ TutorialAssets \ 4_5_x_BetaTutorialAssets \ Graphics \ TutorialGraphics and choose GameObjects_Hero.



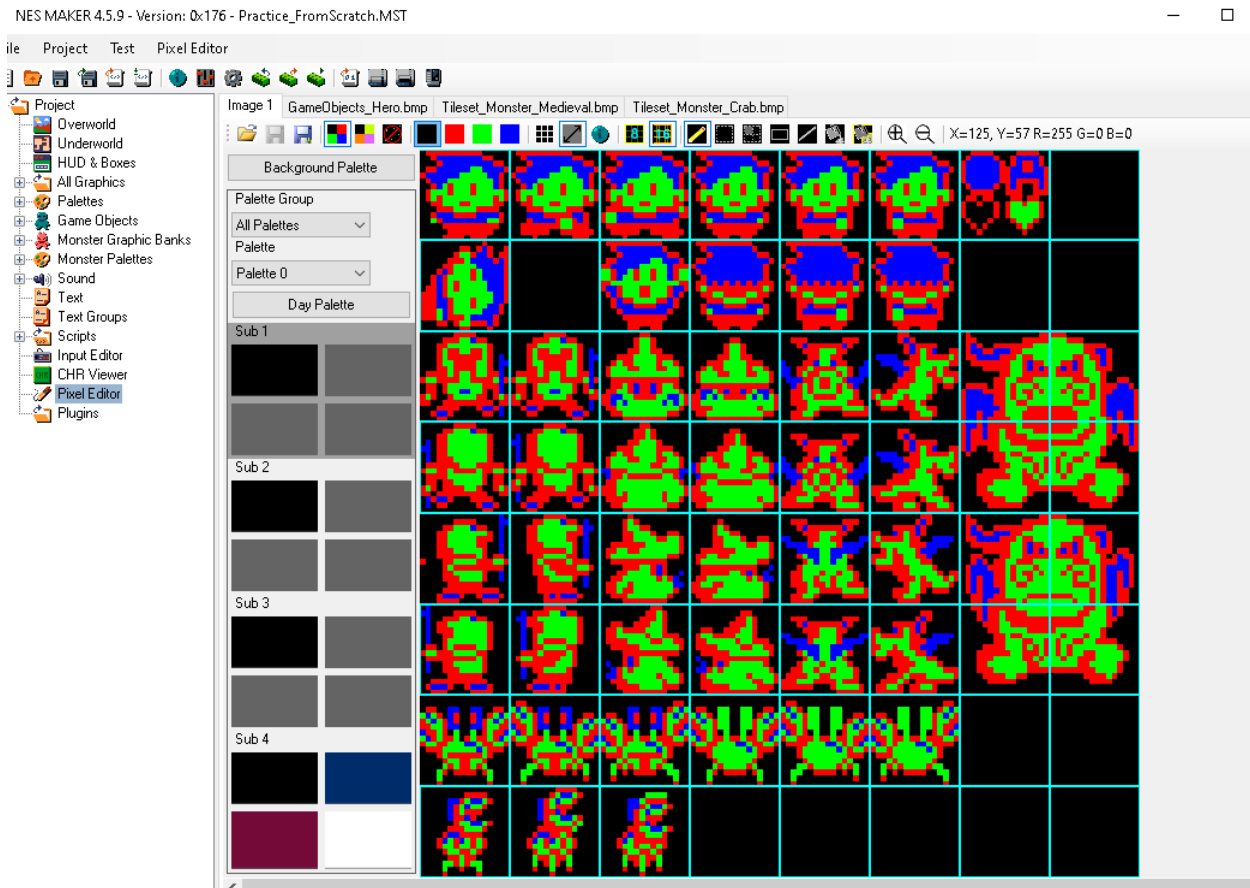
Step 3: Copy the player into the blank canvas.

Use the tile selection tool to select the player graphics, press control-C to copy, then move to your first tab, move your mouse to the top left corner, and press control-shift-V to paste and lock to the grid.



Step 4: Add more graphics to this tileset.

Repeat the process, adding the Medieval Monster tileset and the crab tileset to help fill up this full page. I have turned the 16x16 grid on to be able to more clearly paste the different files into position.



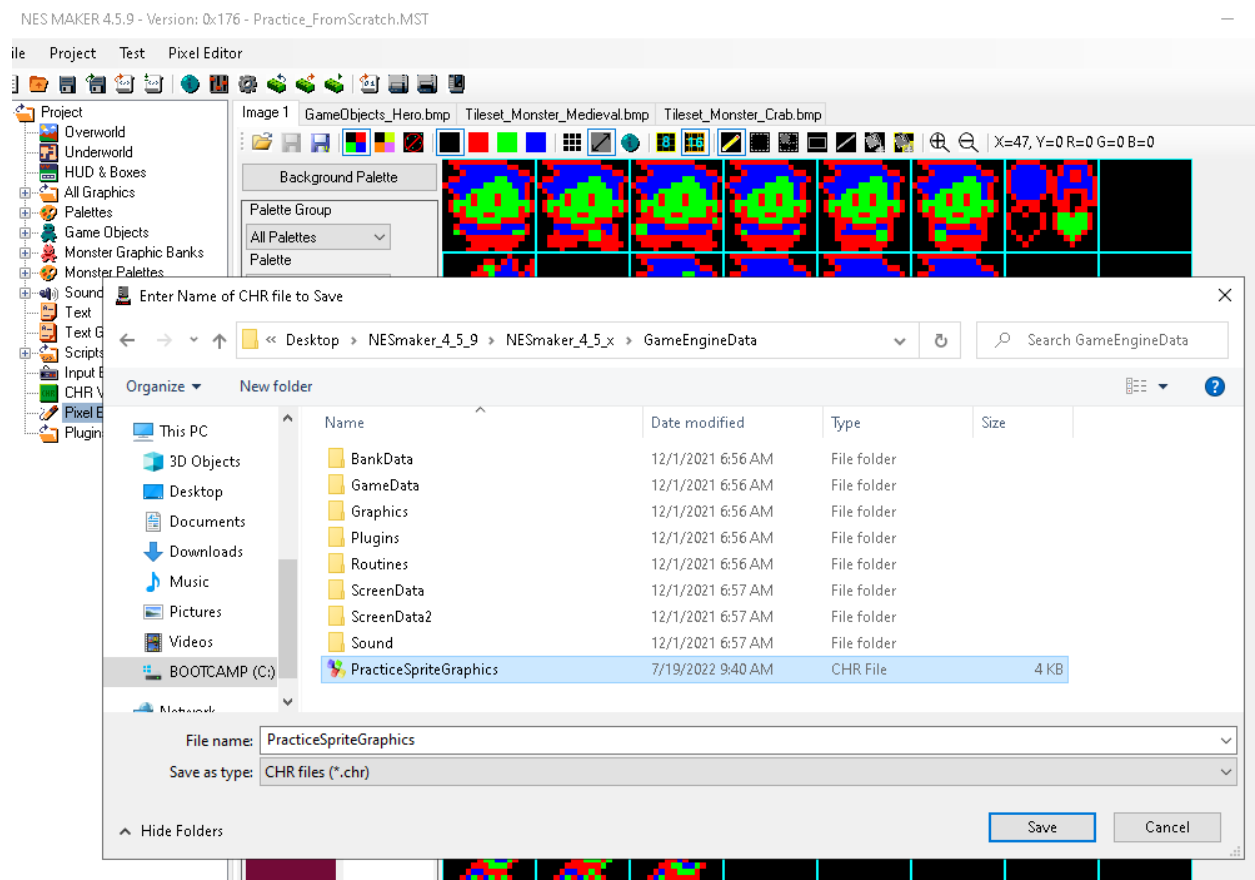
What we have created is a proxy file. When we save this, it will become a four color bitmap file. Working with bitmap files is an outgrowth of early attempts to work with files from external sources such as Photoshop. Upon assembling the NES game with the default modules, NESmaker creates CHR files out of the default tilesets and includes them into memory slots that become available to our GUI. The drop down lists you see in some of the NESmaker tools when you're determining which tilesets to use are the result of this. However, we are not creating a file that fits in that paradigm. Similarly, we are not going to place this bmp file in a location to where it will be converted to a CHR or included in the project. We will have to do this manually.

If we wanted to save this graphics file for easy editing later, we could Save-As and store the bitmap to recall easily. For this project, we'll skip that step and go right to turning it into a CHR file.

Step 5: Export a CHR.

From the Pixel Editor menu, choose Export CHR. For this project, I'm just going to navigate to my GameEngineData folder and place the CHR right there. This is not a great practice, and you should probably make a dedicated folder for any extraneous graphic needs just to keep organized, but for the limited work we're going to do with this project, that will be fine.

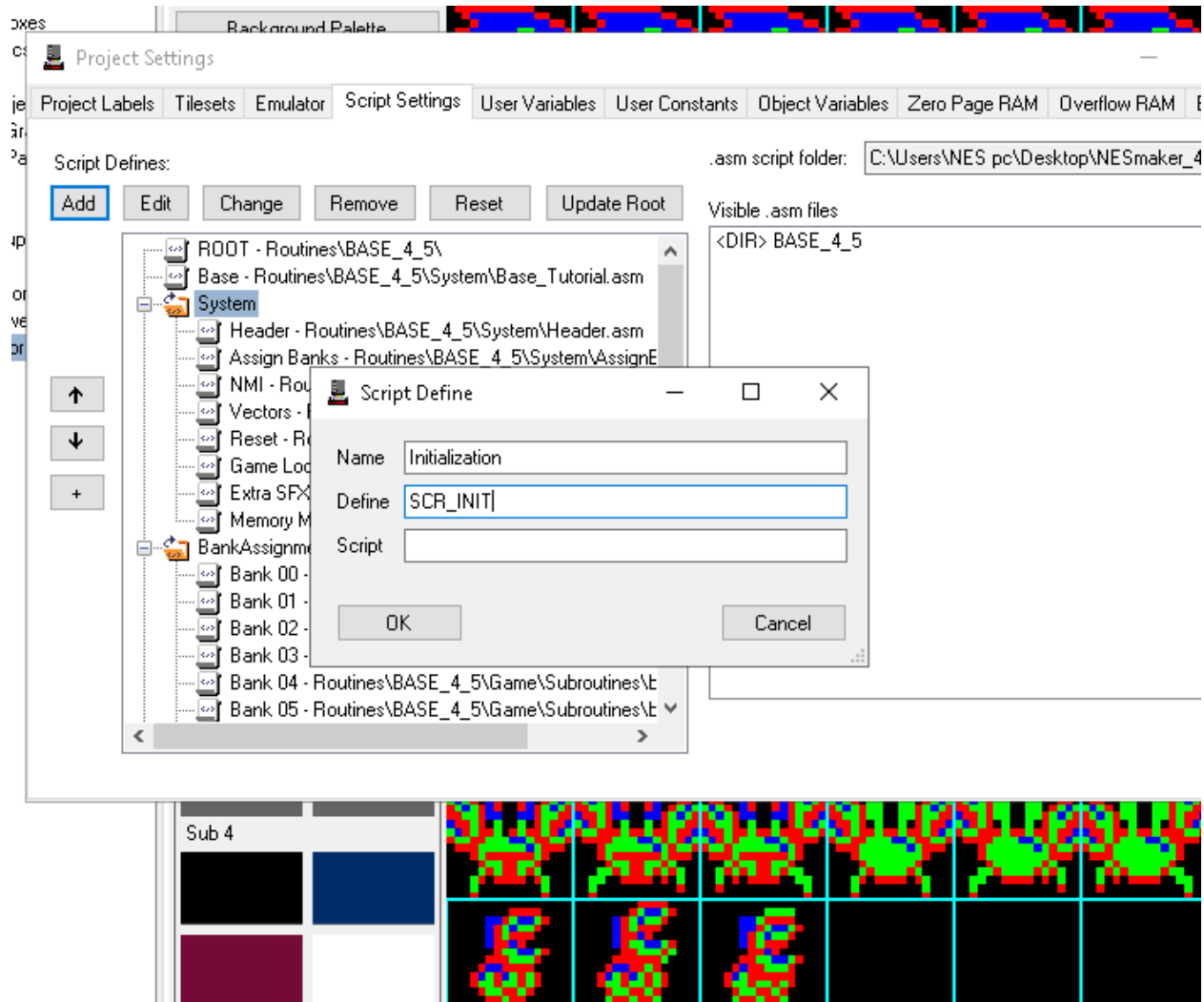
I am going to call mine PracticeSpriteGraphics.



To actually get these graphics into our game, we'll need to do two separate things. The first is load them into our ROM somewhere. The second is to actually load these tiles into the proper place in video memory. We'll create an initialization CHR script to handle loading things into the ROM.

Step 6: Add an Initialization script define.

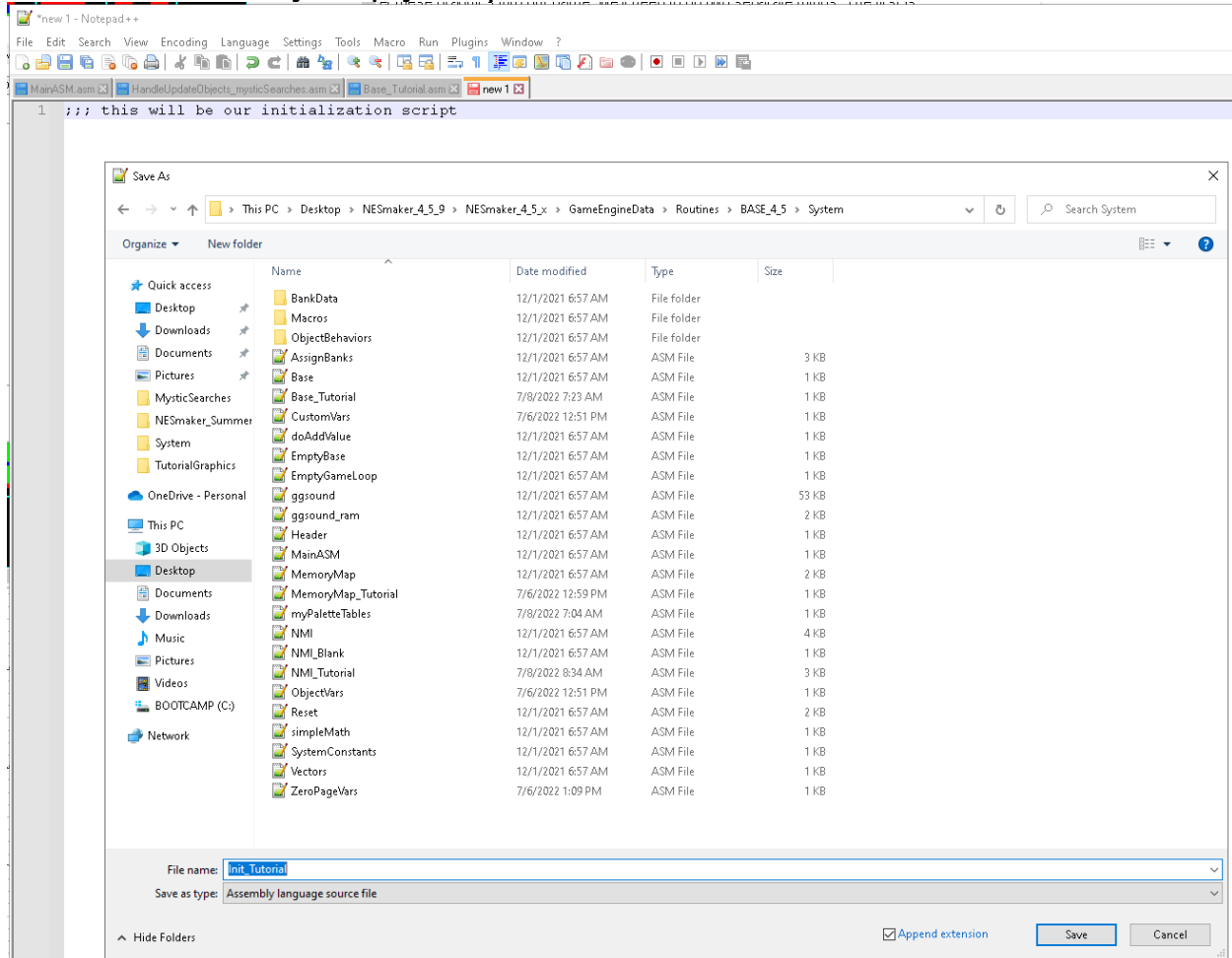
In our Script Settings, in System, add a script define for SCR_INIT.



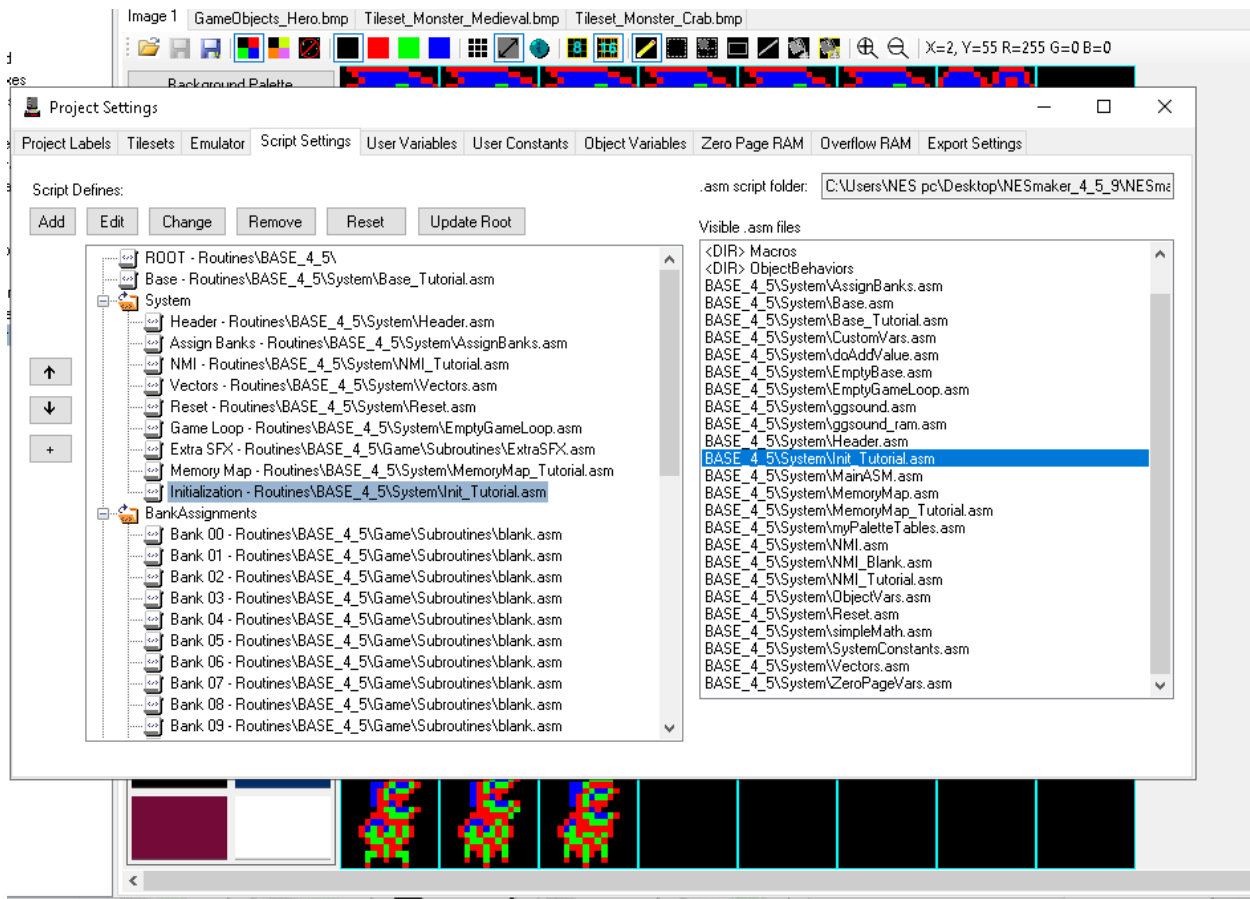
Step 7: Create a blank initialization script.

In your default script editor, make a new script. Save it to the folder that contains your other System Scripts. Where you save it isn't hugely important, but this will make it easier to find. Make sure to save it as an

ASM (Assembly Language Source File) type, and name it Init_Tutorial. Don't forget that you may have to put something in the file for it to save, so using semicolons with some sort of description is always an easy way to save a functionally empty script.



Then, from your script settings, attach this script to initialization.



Step 8: Include the initialization script in your game.

Open up your Base script by clicking on it in the script settings and pressing edit. We're going to include this, but we're also going to start being smart about the order of things in this file. Yes, the header should be at the top. Then the memory map. And then bank assignments. All of that looks good. Once we've moved beyond Assigned Banks, we have put our pen on the top of the page of our static bank. Right now, the first thing we write is the RESET routine. That makes sense. Before we do anything else, we want to clear out all of the existing RAM data. But then we move into a table. That might be problematic. Our game is now just spitting a bunch of random numbers without context, which may be read as opcodes and do all sorts of wacky things!

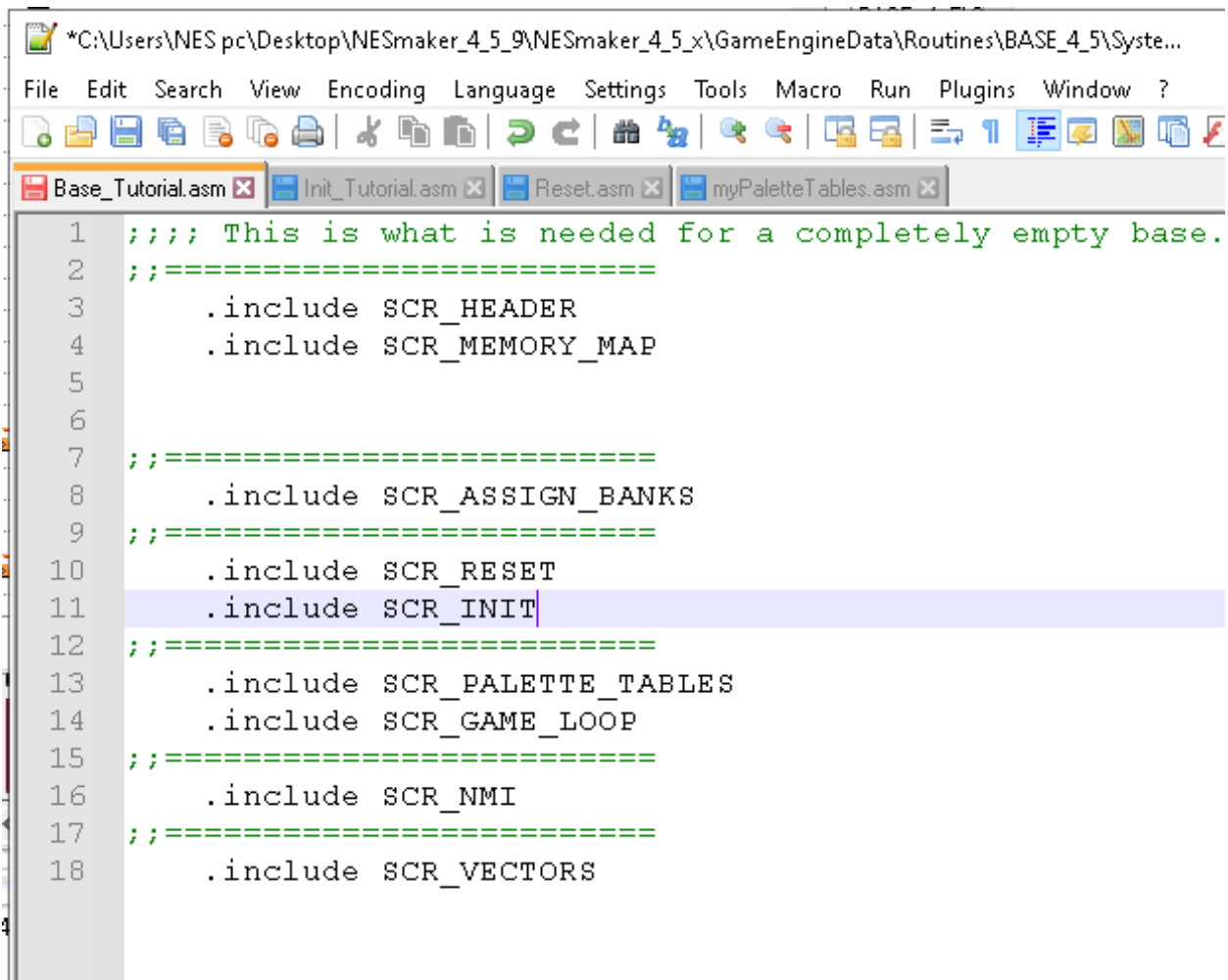
What we really want to do after our RESET is to initialize our startup stuff and then jump to the main game loop. At the end of the main game loop, it

will return to the top of the game loop, and the program will just stay in the main game loop infinitely until there is a command to bounce outside of it. At that point, there is no problem telling it to reference the numbers in our palette tables (like with the NMI) because we'd be giving those numbers context in our main game loop.

So if read like steps of instructions, here's what we want to do:

1. Let the emulator know what is happening (the header).
2. Set up the memory map so we know what ram data goes where. (Memory Map)
3. Assign banks, so that we know what rom data goes where. When we're done doing this, we'll end at the beginning of our static bank. (Assign Banks)
4. The first thing the program should do is reset all the RAM (RESET)
- 5. Initialize any important data we want set up at the start of the game.**
6. Jump to our game loop, and start it looping. (Game Loop)

If we do it like this, we can keep the palette tables right where they are, because at the end of our initialization, we'll jump straight passed them, only ever referencing them when we've told our program to go outside the main loop to fetch them.



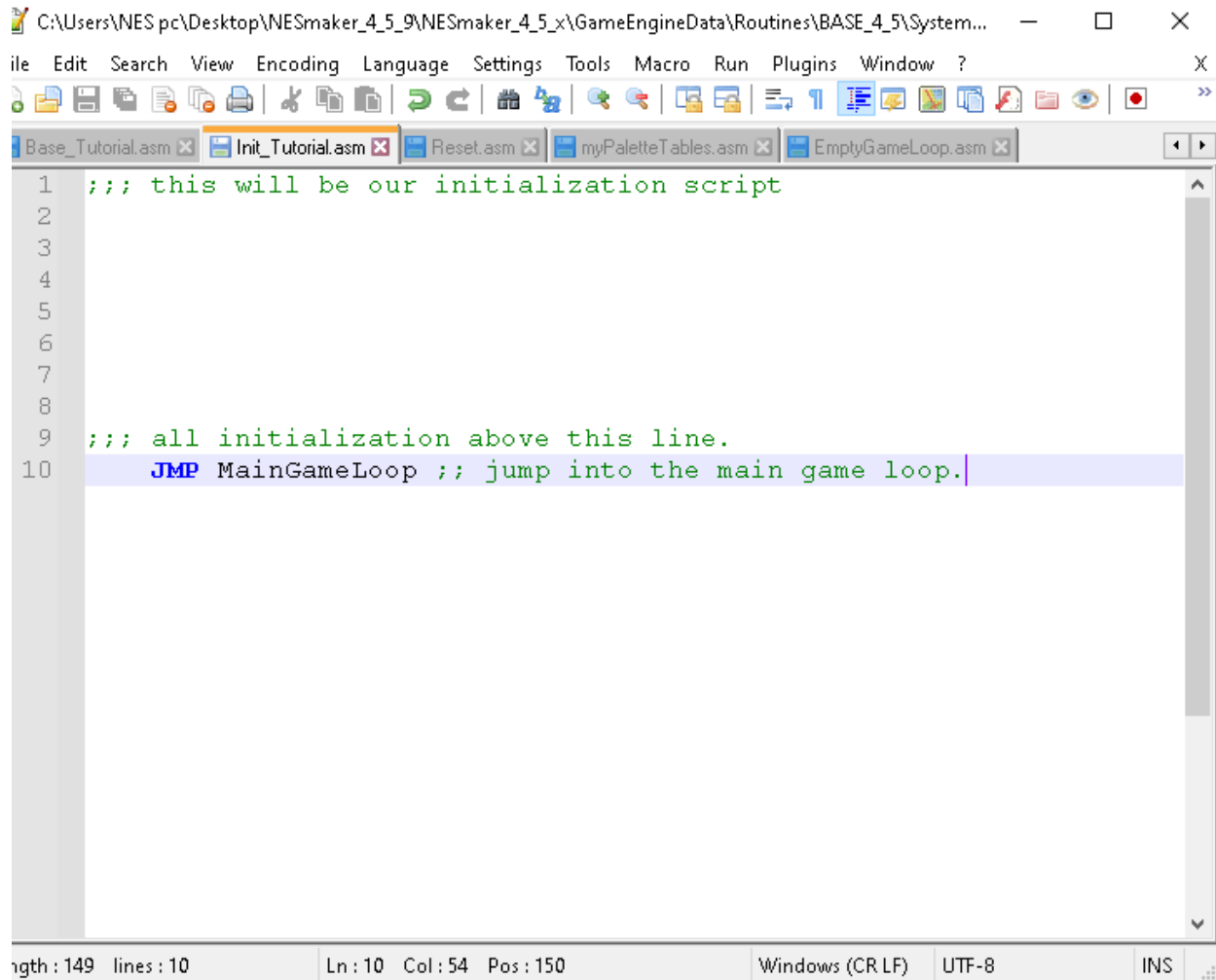
```
*C:\Users\NES pc\Desktop\NESmaker_4_5_9\NESmaker_4_5_x\GameEngineData\Routines\BASE_4_5\System...
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Base_Tutorial.asm x Init_Tutorial.asm x Reset.asm x myPaletteTables.asm x
1      ;;;; This is what is needed for a completely 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     .include SCR_INIT
12     ;=====
13     .include SCR_PALETTE_TABLES
14     .include SCR_GAME_LOOP
15     ;=====
16     .include SCR_NMI
17     ;=====
18     .include SCR_VECTORS
```

Make sure to save this file.

In order for the above plan to work, the very last instruction for our INIT routine has to be to jump to our main game loop. If you were to edit your Game Loop script, you'd find that it has a label called MainGameLoop. The only thing under it is a jump back to MainGameLoop. In between those two lines is effectively where your entire game will live. But first we have to get there.

Step 9: Jump to MainGameLoop.

Open up our currently empty initialization script and at the end of it, write `JMP MainGameLoop`. This will cause our program to correctly jump into our main game loop at the end of initialization.



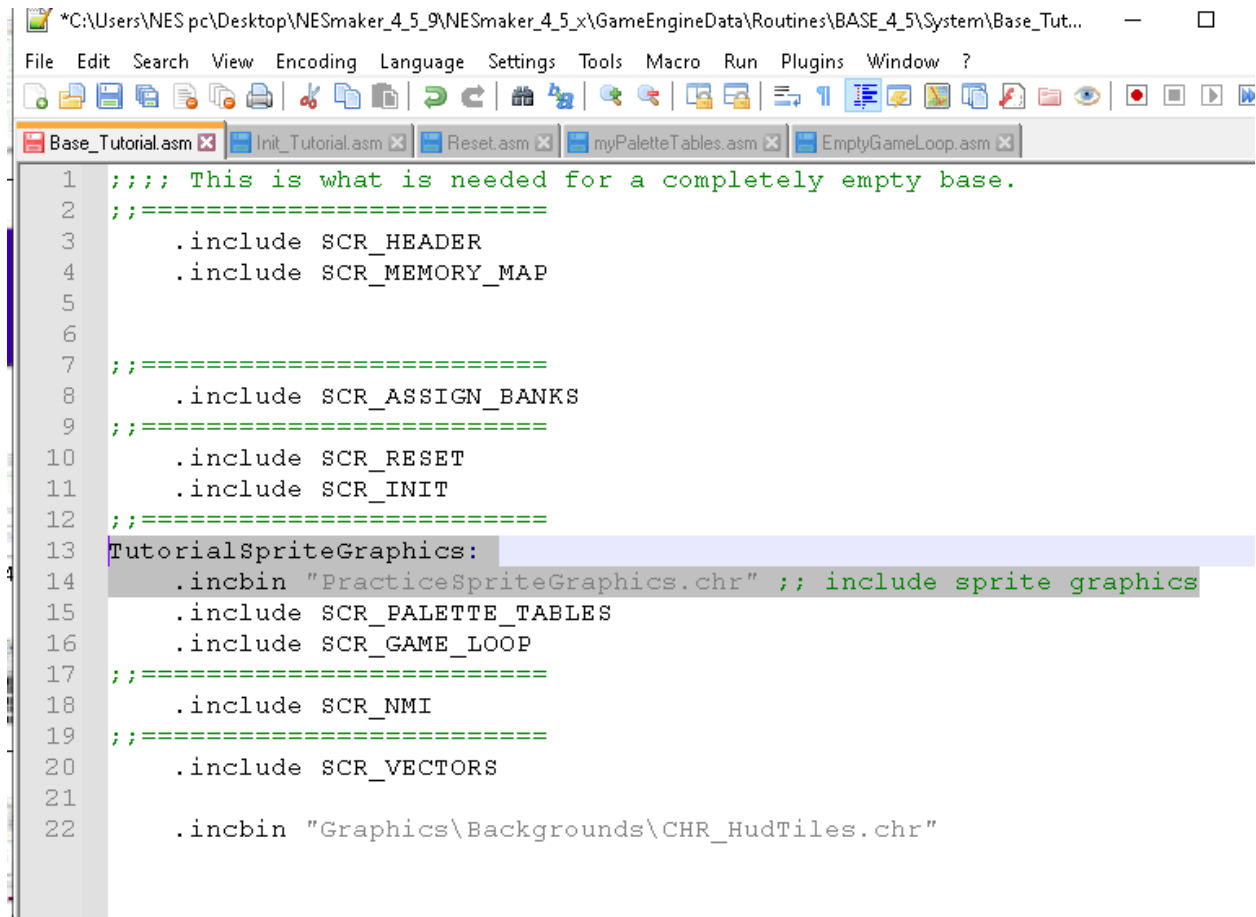
```
C:\Users\NES pc\Desktop\NESmaker_4_5_9\NESmaker_4_5_x\GameEngineData\Routines\BASE_4_5\System...
file Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Base_Tutorial.asm x Init_Tutorial.asm x Reset.asm x myPaletteTables.asm x EmptyGameLoop.asm x
1  ;;; this will be our initialization script
2
3
4
5
6
7
8
9  ;;; all initialization above this line.
10 JMP MainGameLoop ;; jump into the main game loop.
length : 149 lines : 10 Ln : 10 Col : 54 Pos : 150 Windows (CR LF) UTF-8 INS
```

Now we know that at the end of our Initialization, we'll jump right to the game loop. So we can put tables or other data in between the INIT and the Game Loop if we need to.

Step 10: Include the graphics file in our ROM.

Now we have to make a space in the ROM to actually include the binary data for the CHR file we created. This won't load it to our video memory

yet, but will include it in our program so that it's accessible to video memory. Ordinarily, we'd almost certainly spread our graphics data to one of our other memory banks. For right now, though, we're going to try to fit something functional all inside of one bank.



```
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 .include SCR_INIT
12 ;=====
13 TutorialSpriteGraphics:
14 .incbin "PracticeSpriteGraphics.chr" ;; include sprite graphics
15 .include SCR_PALETTE_TABLES
16 .include SCR_GAME_LOOP
17 ;=====
18 .include SCR_NMI
19 ;=====
20 .include SCR_VECTORS
21
22 .incbin "Graphics\Backgrounds\CHR_HudTiles.chr"
```

Incbin includes a binary file, like a CHR file, to our project. Since our codebase sees the GameEngineData as our root folder, we can just incbin the name of the file without adding any folder delineation.

Notice I added a label of TutorialSpriteGraphics before the incbin. We will use this label as a reference point when we go to fetch these graphics to load them into video memory. In normal NESmaker modules, there are labels for each of the default tilesets, and routines to call the right one based on the choice selected in the GUI for a given screen or situation. Here, in doing it by hand, we're just going to crudely send this whole CHR page to the video memory during our initialization.

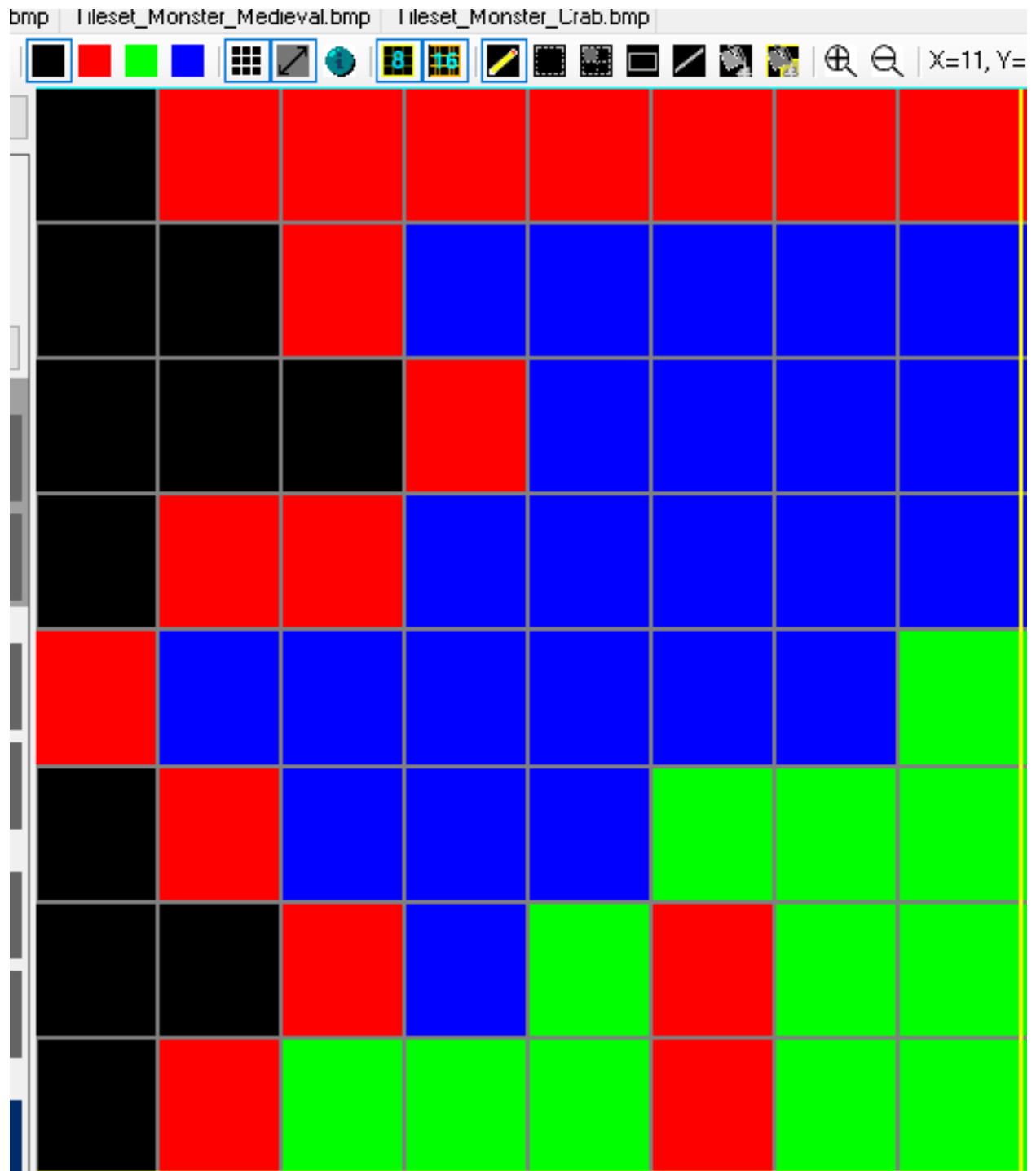
Step 11: Understanding the CHR File

Our Picture Processing Unit has two pages devoted for this graphics data. Based on some things we already set up (bit 3 written to \$2000), our sprite graphics will exist at PPU address \$0000 while our background graphics will exist at PPU address \$1000.

We're going to synthesize some of the things we've already learned and add a few new concepts. First of all, a CHR file is just a long string of bytes that represent pixel information. Every pixel for a tile can be expressed in one of four values, which translate through the loaded palette as colors. If you have colors in your palette slots black, red, blue, and green, any pixel utilizing value 0 will be black, any pixel using value 1 will be red, any pixel using value 2 will be blue, and any pixel using value 3 will be green. If you have an understanding of the NESmaker pixel editor, this should be second nature to you.

But for a moment, let's talk about the actual binary data that makes up a chr file. For the sake of argument, let's focus on a single 8x8px tile. In that tile, we'd expect 64 pixels, since it is 8 across and 8 down.

So where black is 0, red is 1, green is 2, and blue is 3, an eventual tile might look like this:



It could be expressed in a table of colors like this:

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

0	0	1	3	3	3	3	3
0	0	0	1	3	3	3	3
0	1	1	3	3	3	3	3
1	3	3	3	3	3	3	2
0	1	3	3	3	<u>2</u>	<u>2</u>	<u>2</u>
0	0	1	3	2	1	2	2
0	1	2	2	2	1	2	2

Unfortunately, this can not be written in 8 bytes, because bytes can only have 8 bits that can either be flipped to a zero or a one - there is no such thing as a two or a three as a value of a bit. So how a CHR file is written is that each 8x8px tile is two tables made up of 8 bytes that are combined to figure out the final value.

- 0 AND 0 = BLACK
- 1 AND 0 = RED
- 0 AND 1 = GREEN
- 1 AND 1 = BLUE

	BYTE 1								BYTE 2							
0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	
0	0	1	1	1	1	1	1	0	0	0	1	1	1	1	1	
0	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1	
0	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	
1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	
0	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	
0	0	1	1	0	1	0	0	0	0	0	1	1	0	1	1	



The binary equivalent of this tile, which is the top left corner of our character's head, would look something like this in the CHR file:

```
(First set of Bytes)
01111111 00111111 00011111 01111111 11111110 01111000 00110100 01000100

(Second set of Bytes)
00000000 00011111 00001111 00011111 01111111 00111111 00011011 00111011
```

Even though we will never actually write CHR files this way, it's very important to understand how it works in order to properly load the CHR data that we created with our pixel editor. We know that to push a single 8x8px tile's worth of data from our program to the Picture Processing Unit, we will need to write 16 bytes.

The entire page of CHR data consisted of 16 columns and 16 rows of 8x8px tiles, which is a total of 256.

This means that we will need to push 256 x 16 consecutive bytes. That's 4096 bytes of data for that single page of CHR data.

No problem. That's easy enough to conceive of. We write byte one to the proper address, then we write byte 2, then we write byte 3...just keep incrementing until we reach 4096. But unfortunately, we're dealing with an 8 bit system, and as we've discussed already, an 8-bit system only plays nice with numbers from 0-255. For this reason, we'll need to keep track of our copy position using a 16 bit address.

To think about it in terms of mailboxes, every street has 256 houses, each with its own mailbox. I start on First Street. I need to grab the mail from mailbox one on First Street and send it to the PPU, then move along to the next, and then the next, and then the next, until I get to the end of the block.

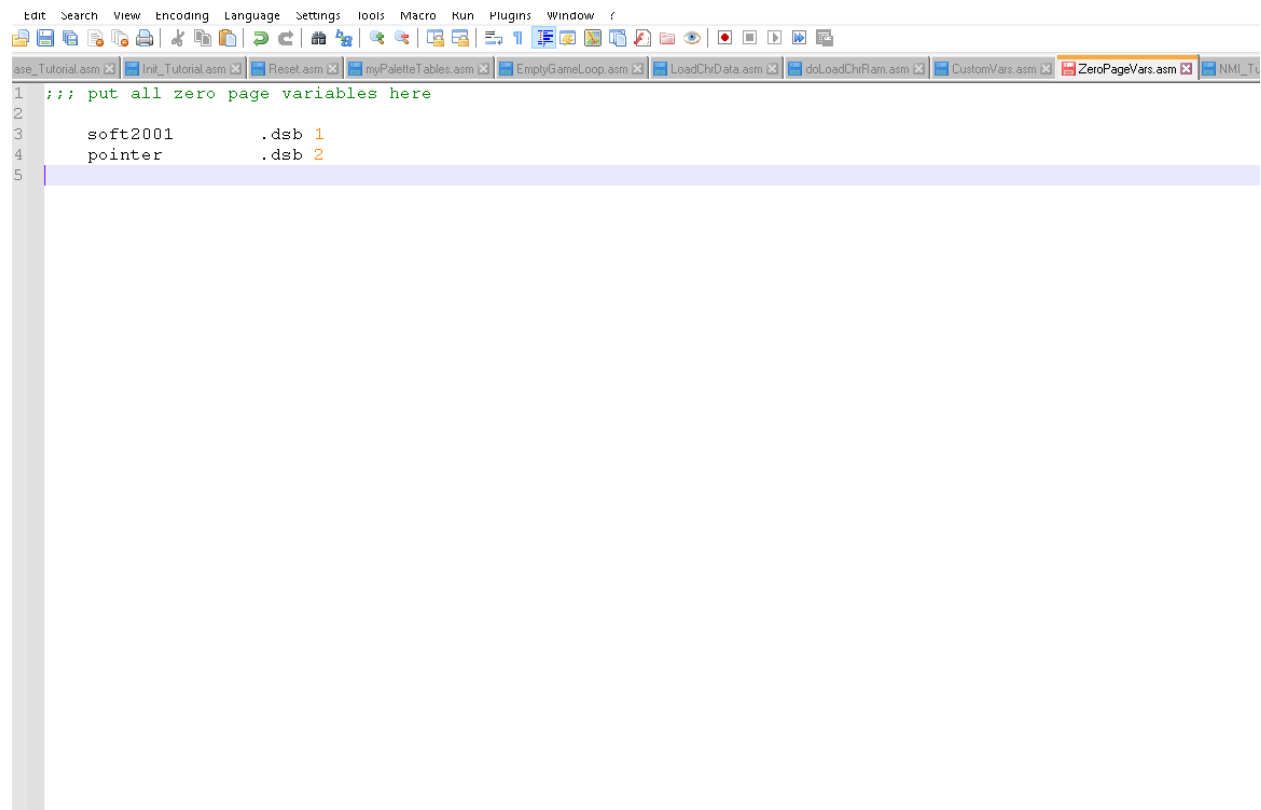
When I get to the last mailbox on First Street, I have not completely finished picking up all the mail. There are sixteen street. When I get done

with First Street, I move to the first mailbox on Second Street, and pick up their mail one by one until the end, before moving on to Third Street, and so on. In this example, the mailbox number is the low byte of the sixteen bit address, while the high byte is the actual street I am on. There are still only ever 256 “mailboxes” on each street, so I’m only ever counting 0-255, but I can then increment through the streets themselves to pick up more than 256 mailboxes full of mail.

So what I’m going to need is a sixteen bit pointer. For this, we’ll create a few variables in our zero page.

Step 12: Creating a 16 bit pointer in our zero page.

Open up your Script Settings and go to the zero page variable file. There, we’ll create a variable called pointer, which will be comprised of two bytes (a high and low byte).



```
1  ;;: put all zero page variables here
2
3  soft2001      .dsb 1
4  pointer       .dsb 2
5
```

Now, in your Initialization file, we're going to set up a loop similar to what we did with the palettes, using our newly created pointer variable, to load in the graphics to the proper place.

Step 13: Writing our logical framework for loading a tile

From here we'll start simple. Here's the logic:

- Load the 16 bit address of our chr file label to the pointer.
- Write the PPU address that we want to write to to \$2006. We're trying to write to \$0000, so we'll write #\$00 to \$2006 twice.
- Use the y register as an offset to iterate through the file byte by byte until we've written 16 bytes (remember, 16 in dec is hex #\$10). Writing 16 bytes from the CHR file will write one full tile, as explained above.

We'll try that to see if we can write a single tile, then we'll talk about how to write all 256 tiles.

```

;;; this will be our initialization script

;; THIS IS WHERE WE'RE FETCHING THE DATA FROM
LDA #<TutorialSpriteGraphics ;; load low byte of address
STA pointer ;; to low byte of pointer
LDA #>TutorialSpriteGraphics ;; load high byte of address
STA pointer+1 ;; to high byte of pointer

;; write to PPU address $0000
;; by writing to consecutive writes to $2006,
;; the high byte and the low byte of the address.
;;; THIS IS WHERE WE WANT TO WRITE THE DATA TO
LDA #$00 ;; high PPU address to write to
STA $2006
LDA #$00 ;; low PPU address to write to
STA $2006

LDY #$00 ;; set the y register to zero
LoadChrRamLoop: ;; start the loop
LDA (pointer),y ;; read y beyond the pointer address.
;; the first time through, this will be zero beyond pointer.
;; then we'll increase it, so it will be 1 beyond pointer. etc.
STA $2007 ;; Write the value there to the PPU
INY ;; increase y.
CPY #$10 ;; have we written 16 bytes yet?
BNE LoadChrRamLoop ;; if not, do it again, with the higher y value.

;;; all initialization above this line.
JMP MainGameLoop ;; jump into the main game loop.

```

Step 14: Making sure that rendering is turned off when writing to the PPU

There is one additional problem with this though if we trace through our logic. In our base file, we do the reset followed by the init script. This is good practice for sure - if we did them in the reverse order, we'd blank out a lot of the ram variables we set up with our INIT. But the problem is, currently at the end of our RESET routine, we turn rendering back on. If you remember from previous lessons, we can only write to the PPU when we're in the vertical blanking period or the screen is turned off. If we try to write to the PPU while the PPU is trying to actively render the screen, a lot can go wrong and we'll get all sorts of jumbled graphics. And of course trying to write over 4000 bytes to the PPU is far too much to try to do in vBlank during the NMI. So what we'll have to do is make sure that rendering is turned off.

We also want to make sure that no NMI is observed in the middle of doing all this, because at the end of our NMI, we turn back on Rendering again.

Doing these things involves writing to \$2000 and \$2001. One option is to just move our writes to turn on rendering out of the RESET and to the end of this INIT script. But since we don't know what else we might put in between the two, it'll be safest for the time being (and to get used to the process) to simply disable rendering and the NMI at the beginning of writing to the PPU in our INIT script, and then enable it at the end.

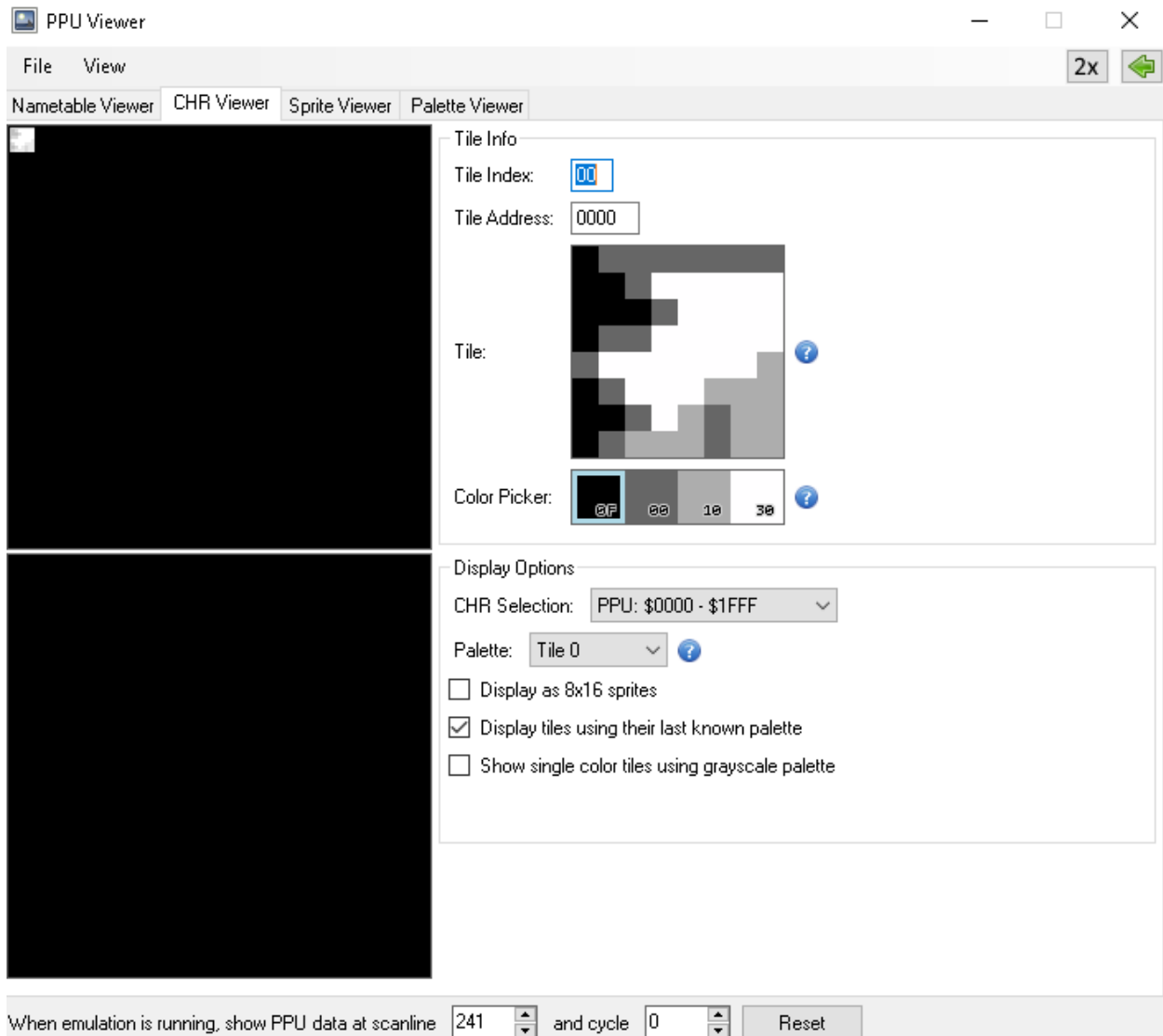
So append the script. At the beginning of the init script, disable rendering and turn off the NMI by writing to \$2000 and \$2001, and then at the end right before jumping to the main game, turn back on rendering and enable the NMI.

```

1  ;;; this will be our initialization script
2
3  ;;; ;;; DISABLE RENDERING AND NMI
4  LDA #$00
5  STA $2001 ;; disable rendering
6  LDA #$00
7  STA $2000 ;; don't generate NMI at the beginning of vertical blanking
8
9  ;; THIS IS WHERE WE'RE FETCHING THE DATA FROM
10 LDA #<TutorialSpriteGraphics ;; load low byte of address
11 STA pointer ;; to low byte of pointer
12 LDA #>TutorialSpriteGraphics ;; load high byte of address
13 STA pointer+1 ;; to high byte of pointer
14
15 ;; write to PPU address $0000
16 ;; by writing to consecutive writes to $2006,
17 ;; the high byte and the low byte of the address.
18 ;;; THIS IS WHERE WE WANT TO WRITE THE DATA TO
19 LDA #$00 ;; high PPU address to write to
20 STA $2006
21 LDA #$00 ;; low PPU address to write to
22 STA $2006
23
24
25 LDY #$00 ;; set the y register to zero
26 LoadChrRamLoop: ;; start the loop
27 LDA (pointer),y ;; read y beyond the pointer address.
28 ;; the first time through, this will be zero beyond pointer.
29 ;; then we'll increase it, so it will be 1 beyond pointer. etc.
30 STA $2007 ;; Write the value there to the PPU
31 INY ;; increase y.
32 CPY #$10 ;; have we written 16 bytes yet?
33 BNE LoadChrRamLoop ;; if not, do it again, with the higher y value.
34
35
36 ;;; all initialization above this line.
37
38 ;;; ;;; ENABLE RENDERING AND NMI
39 LDA #%10010000 ; turn on NMI, set sprites $0000, bkg to $1000
40 STA $2000
41 LDA #00001110
42 STA $2001
43 JMP MainGameLoop ;; jump into the main game loop.
44
45
46
47

```

If you test your game and look at the CHR viewer in the PPU, you should see that you have pulled one tile to the PPU.



Now we need to expand this to load 256 tiles. Since this whole process creates one tile, and we want to create 256 tiles, we can use the X register as a counter to loop through the loop 256 times.

Step 15: Loading our entire tilesheet

Here's the logic. We're going to take the routine that we just wrote, the one that writes 16 consecutive bytes from the position of "pointer" to our PPU, and we're going to loop the whole thing 256 times. In each iteration, we're going to increase the pointer's starting position by 16 so that the position on

the next go-round is at the address that points to the next tile. This will involve 16 bit math.

Let's talk a bit about math in ASM before we put it to work. Addition is actually quite easy.

```
LDA someVariable      ;; load the variable you want to change
ADC #$01              ;; ADC = add, the number is the amount to add
STA someVariable     ;; store the sum back to the variable.
```

So if prior to this operation, 5 was stored in the someVariable mailbox, this routine would take it out, add 1 to it, and put the result back in to the someVariable mailbox.

However, we know that in 8bit math, there are only 256 values before it loops around to zero again. To make this super clear, I'll switch to decimal. What happens if I write this?

```
LDA #250
ADC #10
```

The result of this would be decimal 260. Unfortunately, there is no way to cram more than 255 into an 8-bit mailbox. This would actually end up equating to four. 251, 252, 253, 254, 255, 0, 1, 2, 3, 4. So #250+#10 would equal #4.

But at the same time, an internal carry flag would be ticked. Our program would understand that these numbers overflowed, the same way that in base 10, if we were to say 95+10, we would get 105, which is effectively 05 with a carry into the hundreds place. 1-05.

By default, this carry is set and will be applied to the next mathematical operation until I clear it using the operation CLC, which literally means clear the carry flag.

So if I were to write:

```
LDA someVariable
ADC #$01
STA someVariable
```

...but the carry flag was set from a previous operation, it would carry into this mathematical equation and it would actually add *two* to someVariable instead of one. For this reason, when working in 8 bit math, it is always important to clear the carry if you can't be sure that the state that it's in. So really, trying to add 1 to someVariable would usually look like:

```
LDA someVariable ;; load the var
CLC                ;; clear the carry, just in case it is set
ADC #$01          ;; add one
STA someVariable  ;; store it back to the var
```

But now we're in a peculiar case. We know that there are 4096 bytes we have to load. We need a 16 bit pointer to continue to iterate through those 16 at a time. If we were just using 8 bit math, we would get to 240, try to add 16, and we'd start loading from the beginning again, when we want to continue to move in 16 bit space through the addresses. We want to start loading graphics from the beginning of the label (so, plus zero), and load 16 bytes. That's tile 0. Then we want to jump 16 spaces, which is where the second tile starts, so +16. That's tile 1. Then we want to jump 16 spaces, which is where the third tile starts, so +32. That's tile 2. Each time, loading 16 bytes from the CHR file that make up a single tile, like this.

- Address of Tiles +0
- Address of Tiles +16
- Address of Tiles +32
- Address of Tiles + 48
- Address of Tiles +64
- Address of Tiles +80
- Address of Tiles + 96
- Address of Tiles + 112
- Address of Tiles +128
- Address of Tiles +144
- Address of Tiles + 160
- Address of Tiles + 176
- Address of Tiles + 192
- Address of Tiles + 208
- Address of Tiles +224
- Address of Tiles + 240

But now, on this 17th value, we have a problem. We need to jump another 16, except another 16 would take us to 256. That would effectively lead us

back to +0, which would re-write Address of Tiles + 0 again. We can't have that. This is why we're going to use a carry for the high byte. Like this:

High byte of address +0	Low byte of address +0
High byte of address +0	Low byte of address +16
High byte of address +0	Low byte of address +32
High byte of address +0	Low byte of address +48
High byte of address +0	Low byte of address +64
High byte of address +0	Low byte of address +80
High byte of address +0	Low byte of address +96
High byte of address +0	Low byte of address +112
High byte of address +0	Low byte of address +128
High byte of address +0	Low byte of address +144
High byte of address +0	Low byte of address +160
High byte of address +0	Low byte of address +176
High byte of address +0	Low byte of address +192
High byte of address +0	Low byte of address +208
High byte of address +0	Low byte of address +224
High byte of address +0	Low byte of address +240
High byte of address +1	Low byte of address +0
High byte of address +1	Low byte of address +16
High byte of address +1	Low byte of address +32
...etc	...etc

You can see that when we loop around again, the high byte of the address will increase. Every time we get to that carry again, it will increase again, allowing us to reference offsets through the code as far as we need to without worrying about an 8 bit limit of math.

So how do we do that? Simple - we clear the carry when we add the low byte, but we DON'T clear the carry when we add the high byte. Like this:

```
LDA someVariable
CLC
ADC #16
STA someVariable
```

```
LDA someVariable+1
ADC #0
STA someVariable+1
```

Now, some variable will get regular 8 bit math applied. The second variable, here called someVariable+1, will get zero applied to UNLESS the carry was set during the 8 bit math. If the carry was set, it will add one to someVariable+1. This is a quick and dirty explanation of 16 bit math. We're going to use it for our pointer.

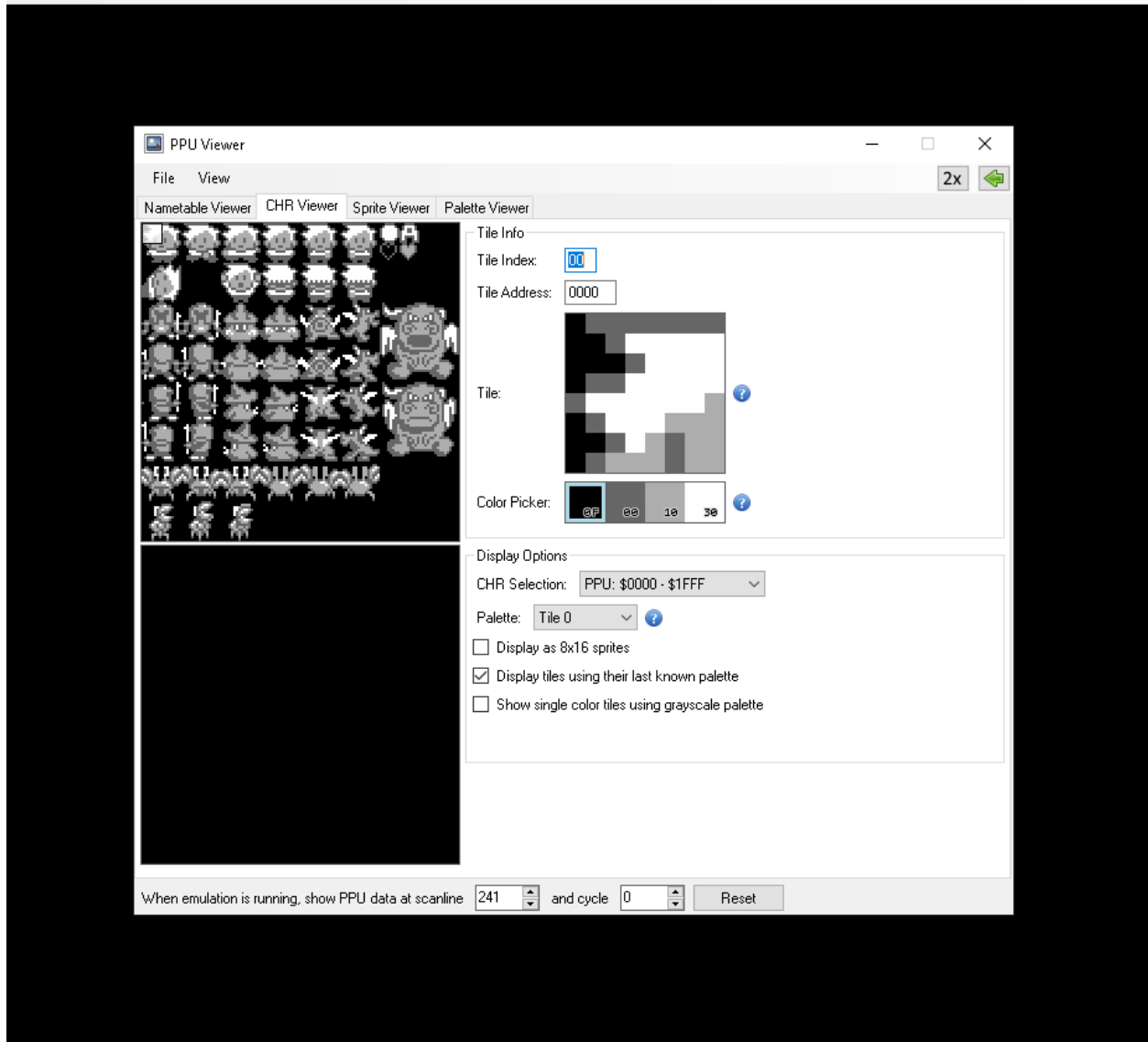
Here, we'll use the X register to count how many tiles we've loaded. If X starts at zero and we take one away, it will become 255. At that point, it is no longer zero, and we will continue our loop. It will continue to do this, decreasing X each time through the loop until X = 0, which is 255 times.

```

8  ;; THIS IS WHERE WE WANT TO WRITE THE DATA TO
9  LDA #$00  ;; high PPU address to write to
0  STA $2006
1  LDA #$00  ;; low PPU address to write to
2  STA $2006
3
4  LDX #$00  ;; set x to zero
5  ;; we'll use this to count down the number of times
6  ;; we've done the internal loop of setting up a single tile.
7  ;; to set up a whole page, we'll need to do it 255 times.
8  ;; If we count down from zero, it will take 256
9  ;; times through the loop to reach zero again
0  doLoadFullCHRpageLoop:
1  LDY #$00  ;; set the y register to zero
2  LoadChrRamLoop: ;; start the loop
3  LDA (pointer),y  ;; read y beyond the pointer address.
4  ;; the first time through, this will be zero beyond pointer.
5  ;; then we'll increase it, so it will be 1 beyond pointer. etc.
6  STA $2007  ;; Write the value there to the PPU
7  INY  ;; increase y.
8  CPY #$10  ;; have we written 16 bytes yet?
9  BNE LoadChrRamLoop  ;; if not, do it again, with the higher y value.
0  LDA pointer  ;; Load the low byte of the pointer address
1  CLC  ;; clear the carry
2  ADC #$10  ;; add hex 10 (16)
3  STA pointer  ;; store it to the lower byte of the pointer address
4  LDA pointer+1  ;; Load the high byte of the pointer address
5  ADC #$00  ;; add zero, but if there was a carry, this will add 1
6  STA pointer+1  ;; store the new value to the high byte of the pointer address
7  DEX  ;; subtract one from x.
8  BNE doLoadFullCHRpageLoop  ;; if it hasn't counted down to zero yet,
9  ;; we're not done yet, so loop and do it all again.
0
1  ;; all initialization above this line.
2

```

Test your game and check out the PPU. You should now see your entire graphics sheet loaded in the top table, being pushed through whatever you have set up for your sprite palette.



End of Lesson 4.

You should now have a little bit deeper knowledge of the structure of CHR files, how to load a CHR file into your ROM, and how to push it to the PPU. You should be aware that you can only work with the PPU when rendering is turned off. You should also have a decent grasp on how 8 and 16 bit addition work, and basics about using 16-bit pointers.

