LESSON 5

We're finally set up to start making our game actually do something. In this lesson, we're going to set up a very basic drawing system and draw our player to the screen.

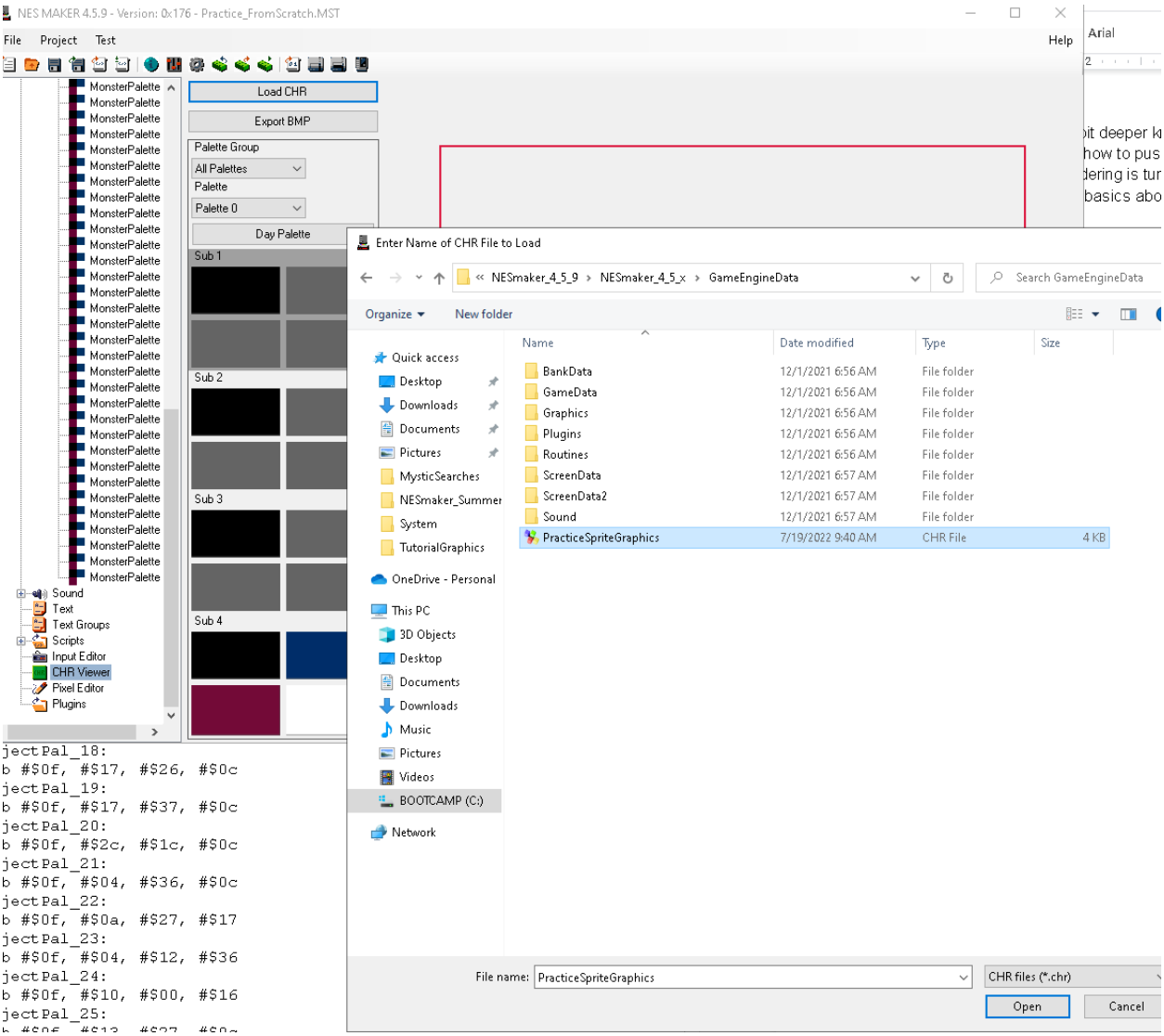
But first, we need to set up some palettes for our sprites. We already learned how to set up palettes for our background, but we never intentionally set up any palettes for our sprites. This is a great opportunity to see the benefits of using NESmaker even when working from scratch.

For our background palette, we manually created a table of colors using constants, and we wrote those values to PPU address $3F00 - 3F0F. Those are the sixteen values that make up our background palette. Immediately following that, in the 16 PPU addresses at $3F10-3F1F, are where our sprite palettes live. We could go through the exact same process of creating a table for these palettes manually, but instead we're going to use NESmaker's GUI controls in the Pixel Editor to define palettes so we can actually see what an object would look like through those palettes, and then make use of the exported data file.

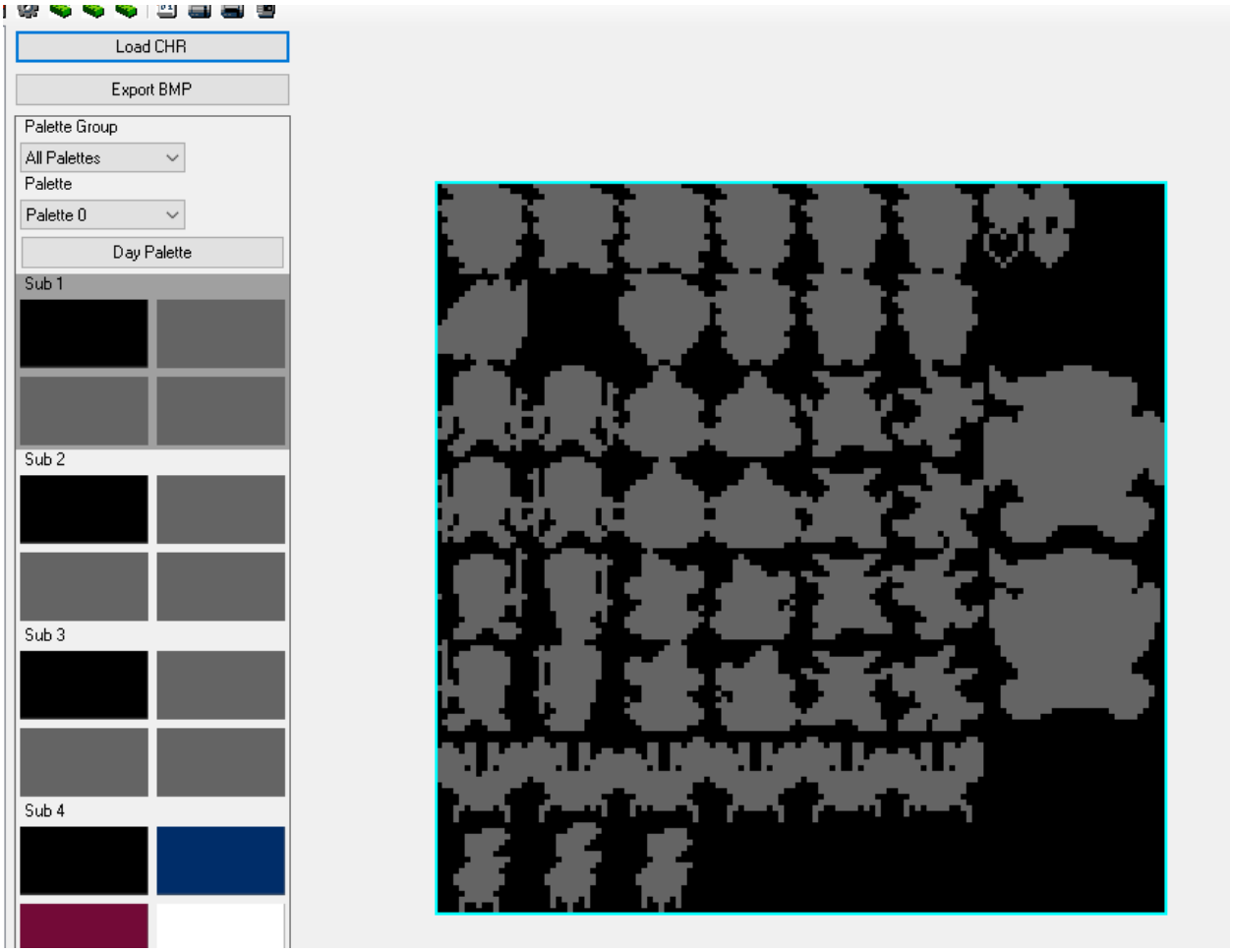
If you have not closed your project, the pixel editor should still show your tileset. But if you have, we are in a bit of trouble. We exported a CHR file, but the NESmaker pixel editor doesn't work directly with CHR files, it deals with bitmap files. You may have closed without saving that bitmap file. Fortunately, NESmaker does have a CHR viewer capable of exporting a BMP file. It's a but of a detour, but let's use the CHR viewer to make a bitmap file of our sprite tileset.

Step 1: Save the CHR as a BMP.

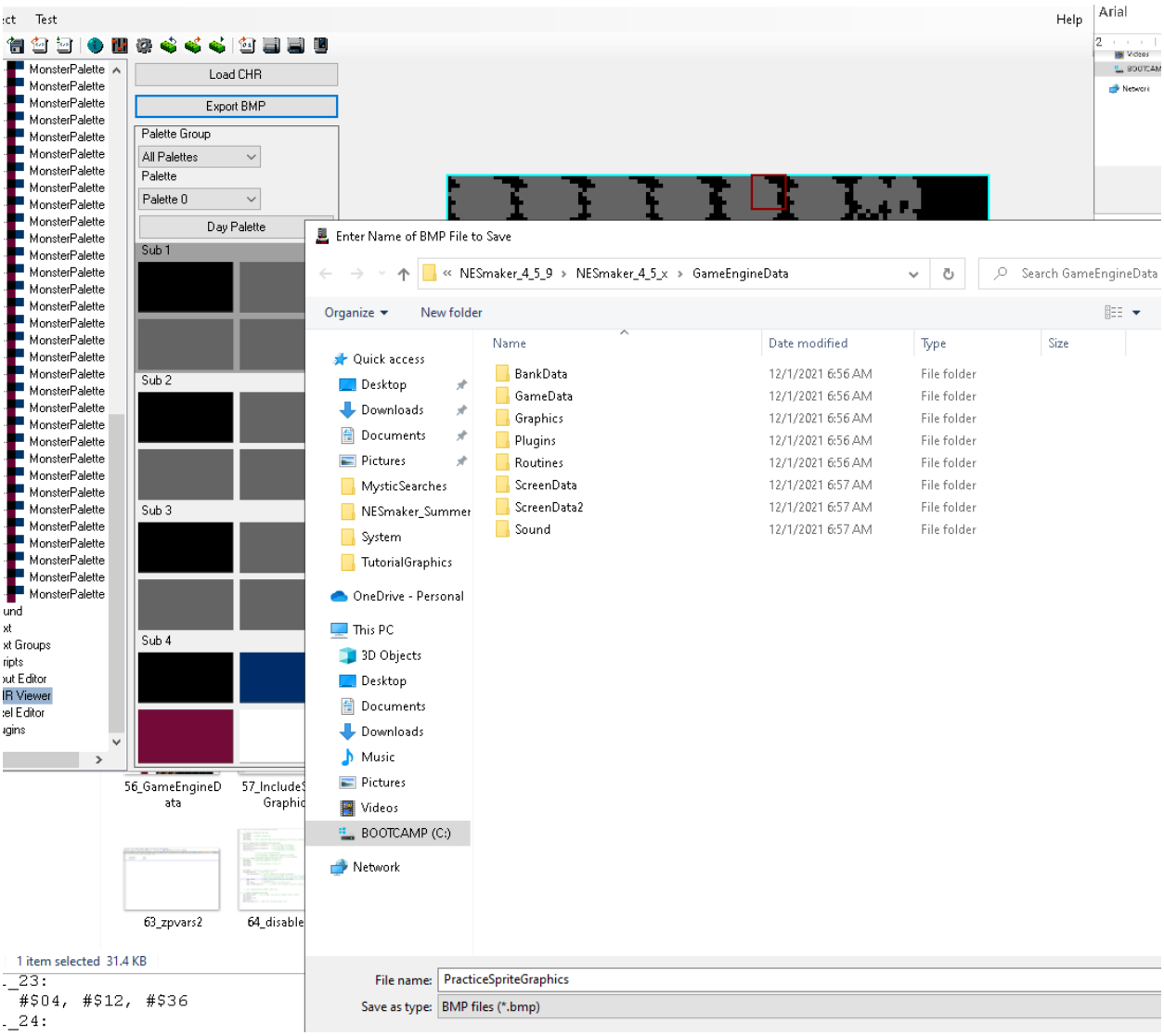
In the hierarchy, click on the CHR Viewer tool. Load the CHR file that we created. It should be in your GameEngineData folder.



You'll notice that the entire tileset probably looks like gray silhouettes. That's because the currently loaded palette through which to see it is all gray, so every pixel is being pushed through its paint-by-number gray. Remember, NES graphics files don't contain any color information, only reference numbers. Those numbers reference the currently loaded palette, whatever it happens to be. In this case, it's all gray, so the graphics will show as gray. This is not important right now. The data is still present.

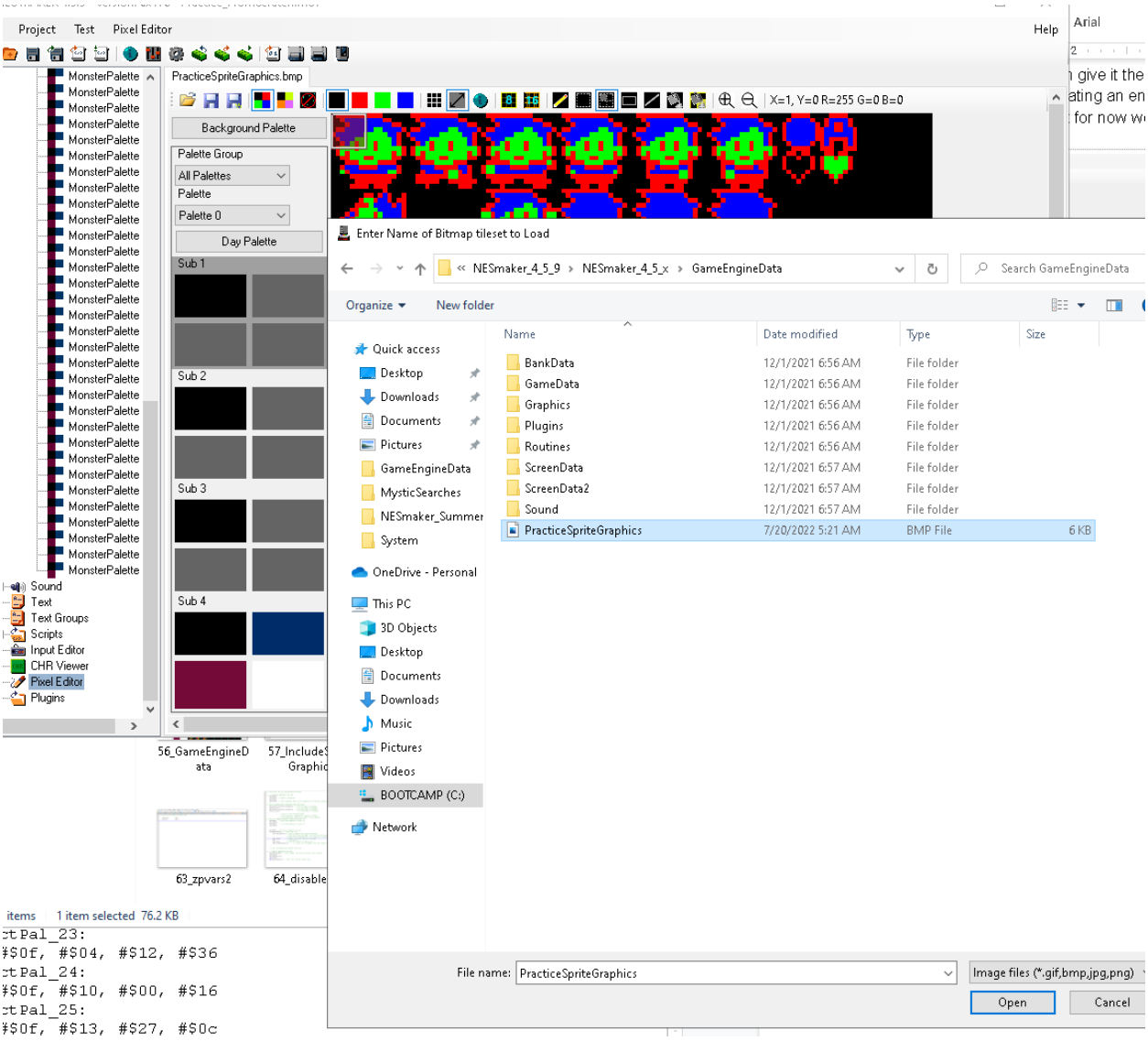


Click on Export BMP. You can give it the same name and save it right next to your CHR file. As stated earlier, if you were creating an entire complex game, you would likely get very strategic with your folder structure, but for now we're just making everything easily accessible.



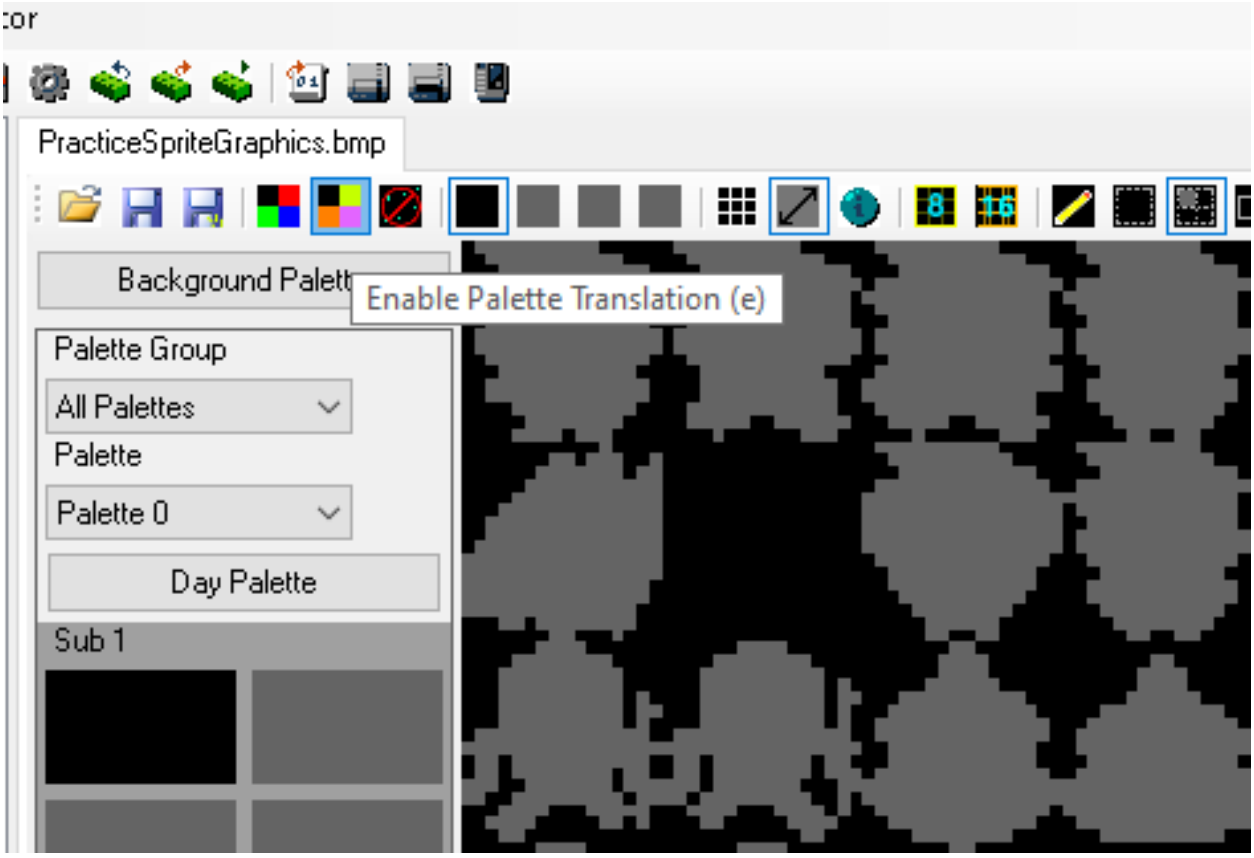
Now we have a CHR version of our graphics, and a BMP version that we can easily bring into other programs such as Photoshop, Gimp, or even Microsoft Paint to make changes. Or, of course, we can bring the file into our own pixel editor as well, which is what we'll do now.

In the pixel editor, open the BMP we just created. By default, the finder window will be in your GraphicsAssets folder because by default, that's where NESmaker modules store their tilesets, But we strayed from default use, so you'll have to back up a few levels to access the GameEngineData folder where we placed our bmp.



Step 2: Setting up Sprite Sub Palettes

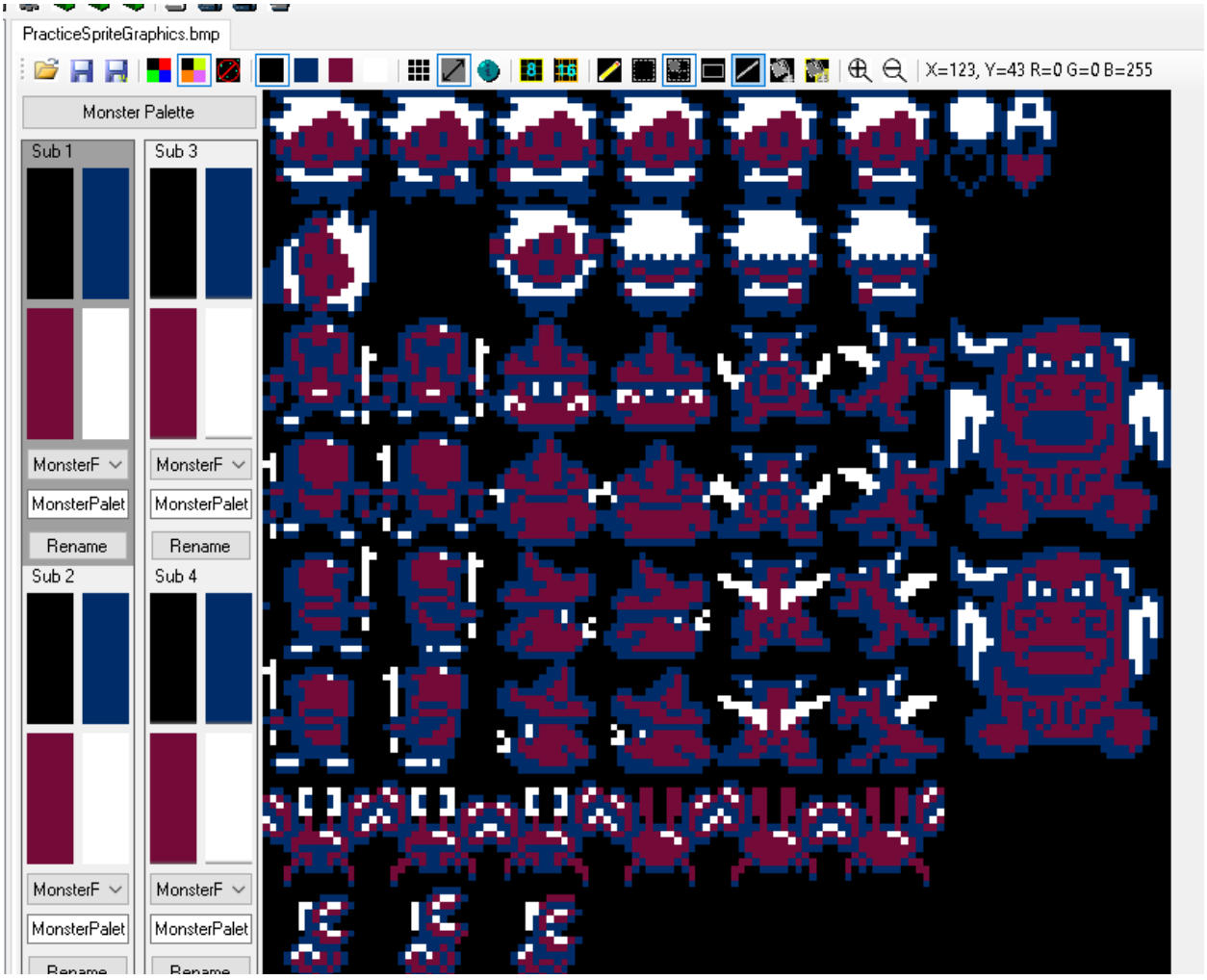
Right now, you probably see your graphics in RGB-A mode. That is to say, you're seeing Black, Red, Green, and Blue to denote color values 0,1,2, or 3. What we want to do instead is enable palette translation, so that color zero (black) is pushed through the first subpalette color, color 1 (red) is pushed through the second subpalette color, color 2 (green) is pushed through the third subpalette color, and color 3 (blue) is pushed through the fourth subpalette color. There is a button in the top menu bar that is called Enable Palette Translation. Turn it on.



Right now, though, we are looking at this through a background palette. The change of any palette we make here would be pushed out to the background palettes, which have a few peculiar characteristics in NESmaker that help with fades and day/night systems. We just want a sprite palette that is a straight run of 16 values that can be pushed the PPU in the sprite palette slots.

For this, click on the big Background Palette button and you'll see it toggles to "Monster" Palettes. You'll already see the colors change, and you'll see that instead of having a string of 16 connected colors, we have four sets of four subpalettes. This is a creative NESmaker decision. Usually, when designing backgrounds, you will utilize a full 16 color palette for a screen. However, you may use your sprite subpalettes in varied combinations. For instance, you'll probably always have one subpalette reserved for your player, but you might have another reserved for his spell or weapon, while reserving two others for monsters on the screen. Screen by screen, the monsters might change, so their colors might change. Selecting a different magic type or weapon might change. But your player will stay the same.

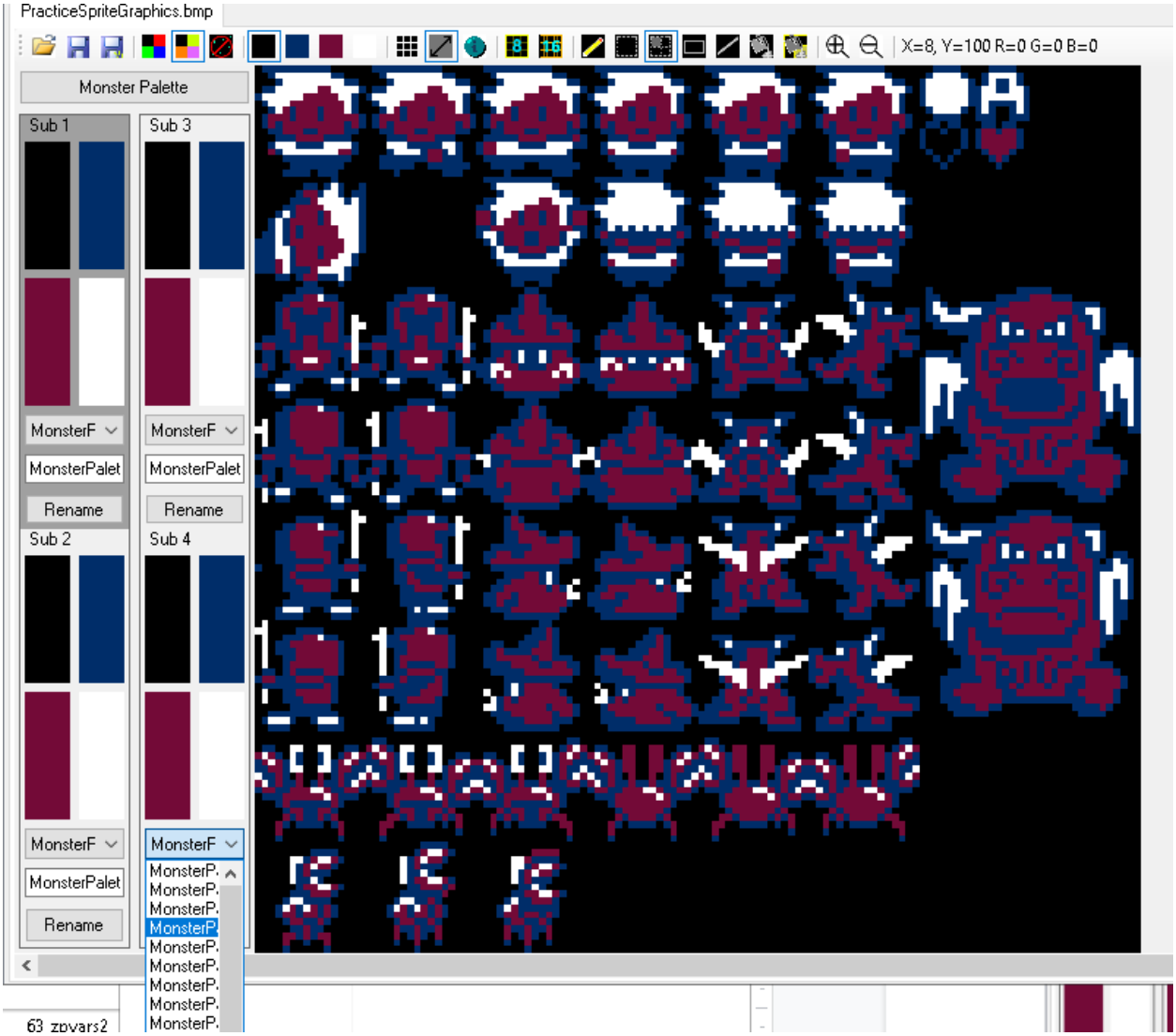
Or, perhaps you have multiple playable characters, and a subpalette for your player needs to change depending on the choice, whereas the monsters always have a static color. There are a lot of reasons why you'd want to be able to mix and match subpalettes for sprites, those are just a few. That is why sprite, or "monster" palettes are treated a little bit differently than background subpalettes in this GUI.



Right now, all subpalettes are assigned to the same group of four colors (called monsterPalette0). We want to make sure that they're all set to different color sets. Keep "sub1" as MonsterPalette0, but change Sub 2 to MonsterPalette1, change sub 3 to MonsterPalette2, and change Sub 3 to MonsterPalette 3.

Due to a slight glitch in NESmaker 4.5.x, it is hard to see the names of these, but you can always rename them, and for these purposes, just make

sure that the four sub palettes of the GUI are assigned to the first four items from the drop down list.

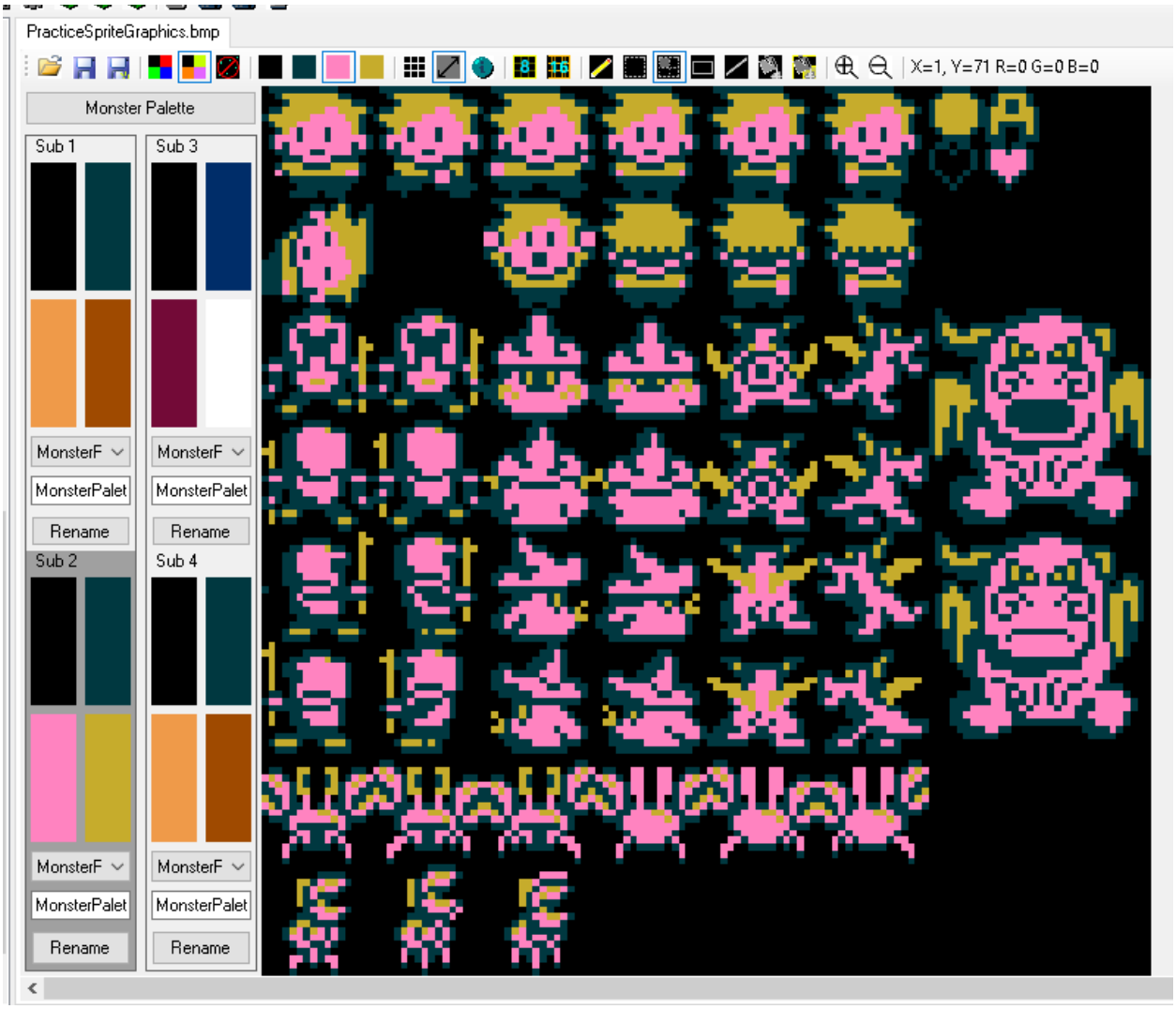


Step 3: Setting up different sub palettes

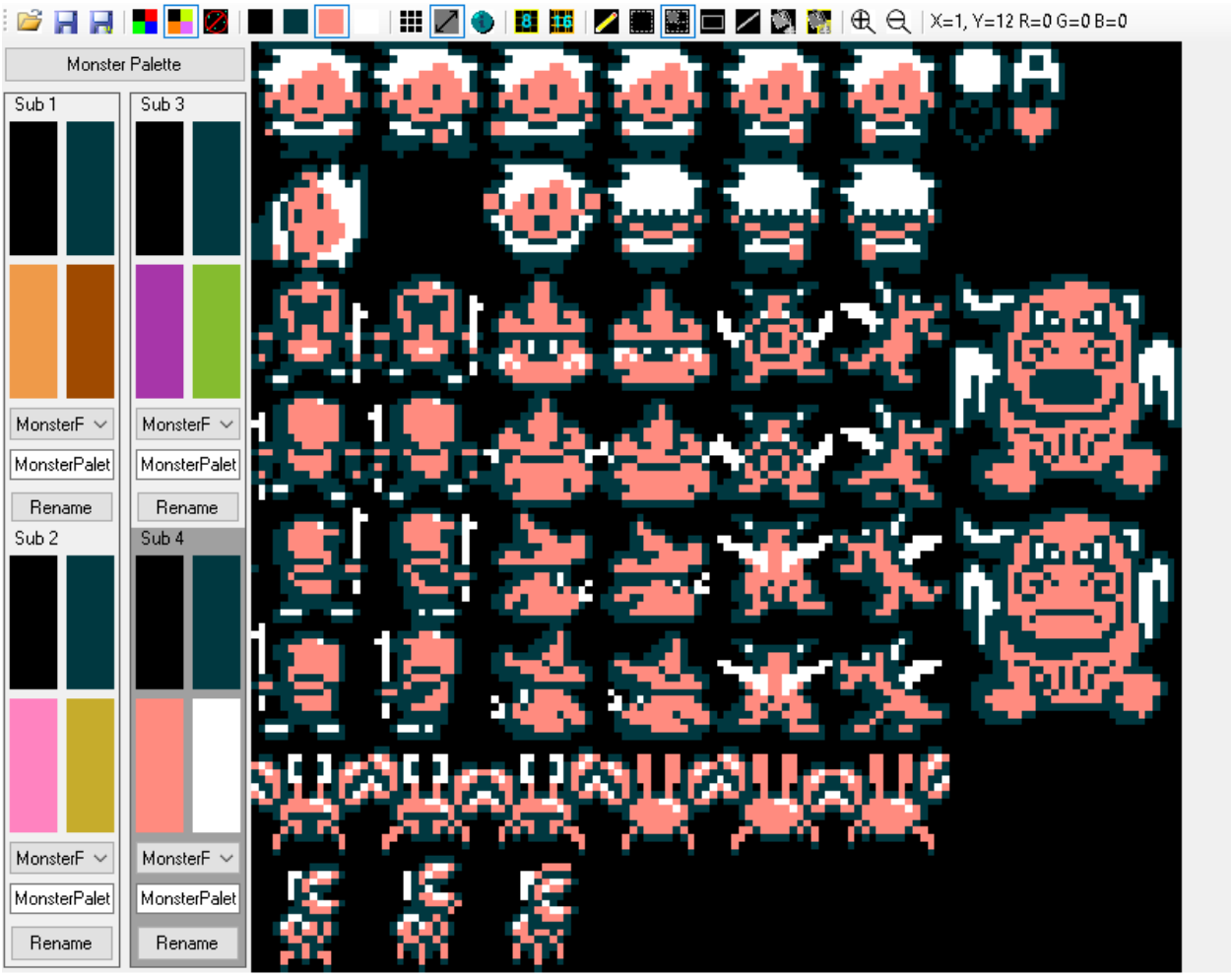
You will see all graphics for all sprites being pushed through whatever subpalette you have selected. Again, this is not actually determining the color. We will determine the color when we're drawing these sprites to the screen. What we want to focus on is creating a sub palette for the player, a sub palette for our power ups, and two sub palettes for our monsters. Right click on any colors in the subpalette will bring up a color picker.

Make sure that the first color in each subpalette remains black. Beyond that, play with the colors in SUB 1 (which is generating four values for monsterPalette0) that look good for the player. Disregard the other graphics and just focus on creating what looks good for those sprites.

Next, focus on the powerups. Use Sub 2 for this. Disregard the player and the monsters and just focus on the powerups.



Next, focus on the monster set. We only have to subpalettes left. I'm going to focus on the wizard and the crab. I'll use Sub 3 for the wizard and Sub 4 for the crab. In the end I'll have something that looks like this.



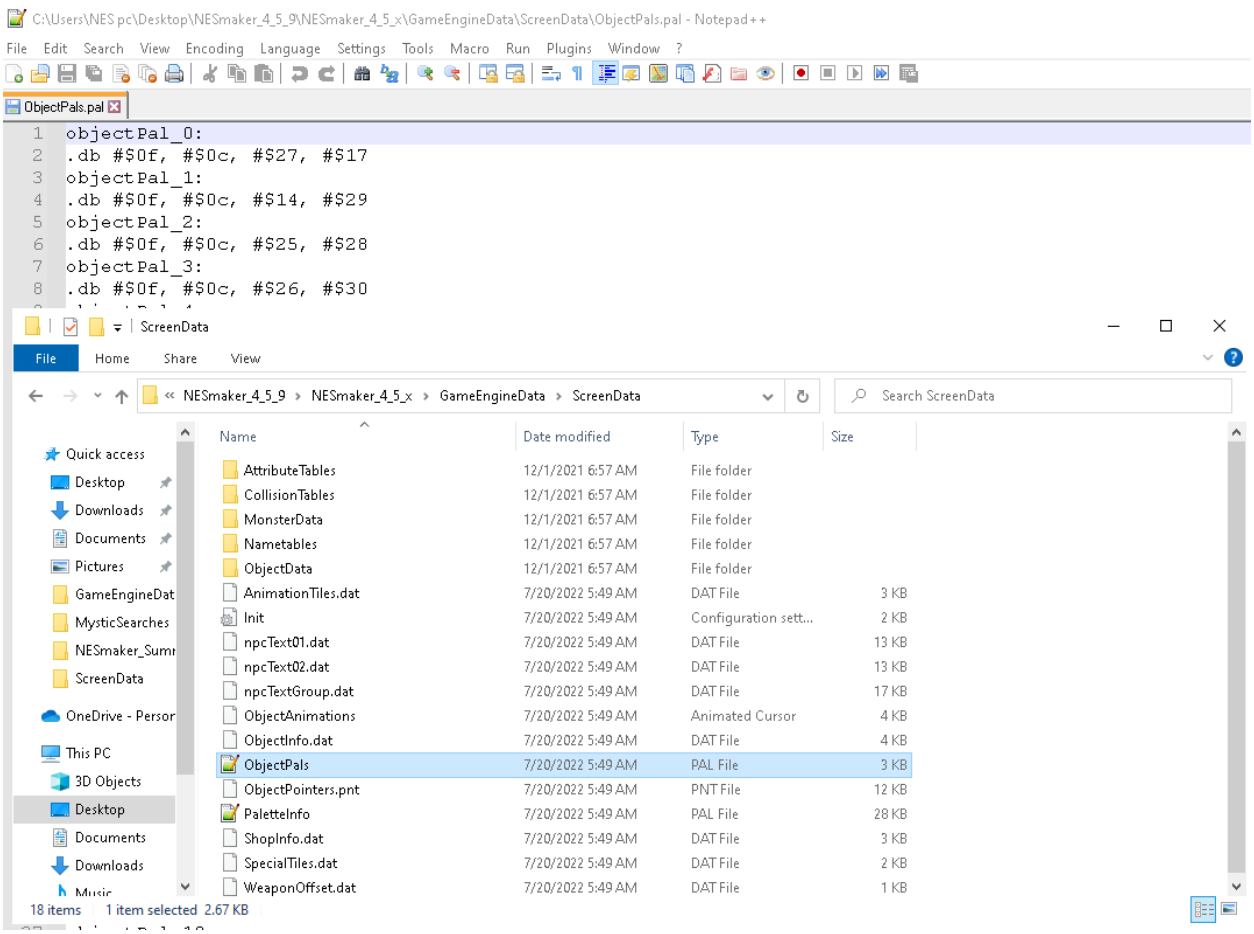
What we've done is we've created four subpalettes of four colors each. If we were to load these consecutively, we'd have 16 values with which to fill the slots of the sprite palette.

Step 4: Looking at the export data.

Export and test your game. Nothing will have changed since the last time we exported, except for the fact that there is an object palette file that contains something different than the default values.

If we navigate to GameEngineData \ ScreenData, you will see a file called ObjectPals. If you right click and open this with your text editor (if it's not already set to open this file type with a text editor), you will see a file with

labels, and under each label, four values. Those first four labels are what we just established in our tool. If we were to set a pointer to objectPal_0 and write 16 consecutive values to PPU addresses $3F10-3F1F, the four sub palettes from our GUI would be set up in order in our PPU. We could then use those colors to draw our respective sprites.

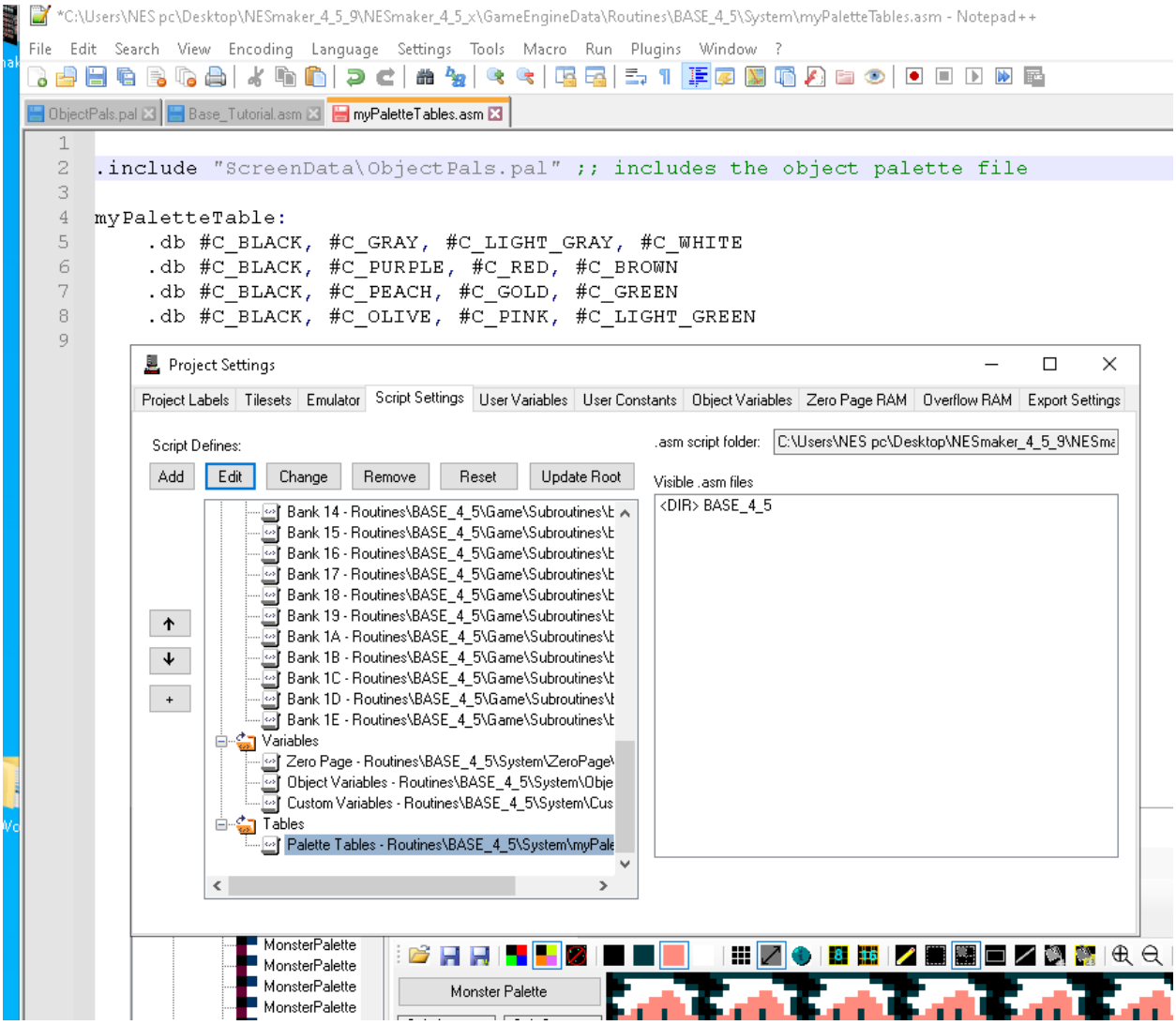


Now, we could just copy this data and put it in a table in our program, and then do exactly what we did for the background palettes except write to the sprite palette address instead, but instead we're going to make all of this object palette data available to us by using this export as intended. This is a great and easy to understand opportunity to demonstrate how the GUI works with the underlying code in the modules.

We're going to do exactly what we did with the CHR file. We're going to load this table to our ROM, then we're going to reference the table when we want to write to the PPU.

Step 5: Including the new tool-generated palette tables.

Open your script settings and scroll down to your palette tables file. This script, if you don't remember, is referenced in your base scripts, and right now contains the myPaletteTable data which populates the background palette slots. In order to keep our palette information together, we're going to include our palette file here, which will load into ROM all 64 object sub palettes, four values each. In fact, if we go back and make changes in the GUI or alter other sub palettes, the data will automatically correct on export. Rather than having to change a number value or a constant in code in order to see the change, we can just go into the GUI, right click and pick a new color that looks better, and re-export. The end result is the same - a file of HEX data is being created and referenced in the code we created that pushes values in certain places to the PPU. The thing that's different about the approaches is having to track down the right color to change and writing it manually versus seeing it visually and having the actual correlating value be exported into the right place.



Step 6: Pushing the Sprite Palette data during the NMI.

Keeping this as simple to understand as possible, albeit with the potential for better optimization, we are simply going to copy what we did for the background palettes and apply the exact same logic for the sprite palettes. Since we have set up the four colors or each subpalette to be successive on our big object palette table, we can start at the first palette, blast out 16 values, and have the exact same result as if we were going to each subpalette and blasting out four values into each subpalette slot.

From your Script Settings, open up the NMI. I added a comment before the loop that loads the background colors so I could easily see where the

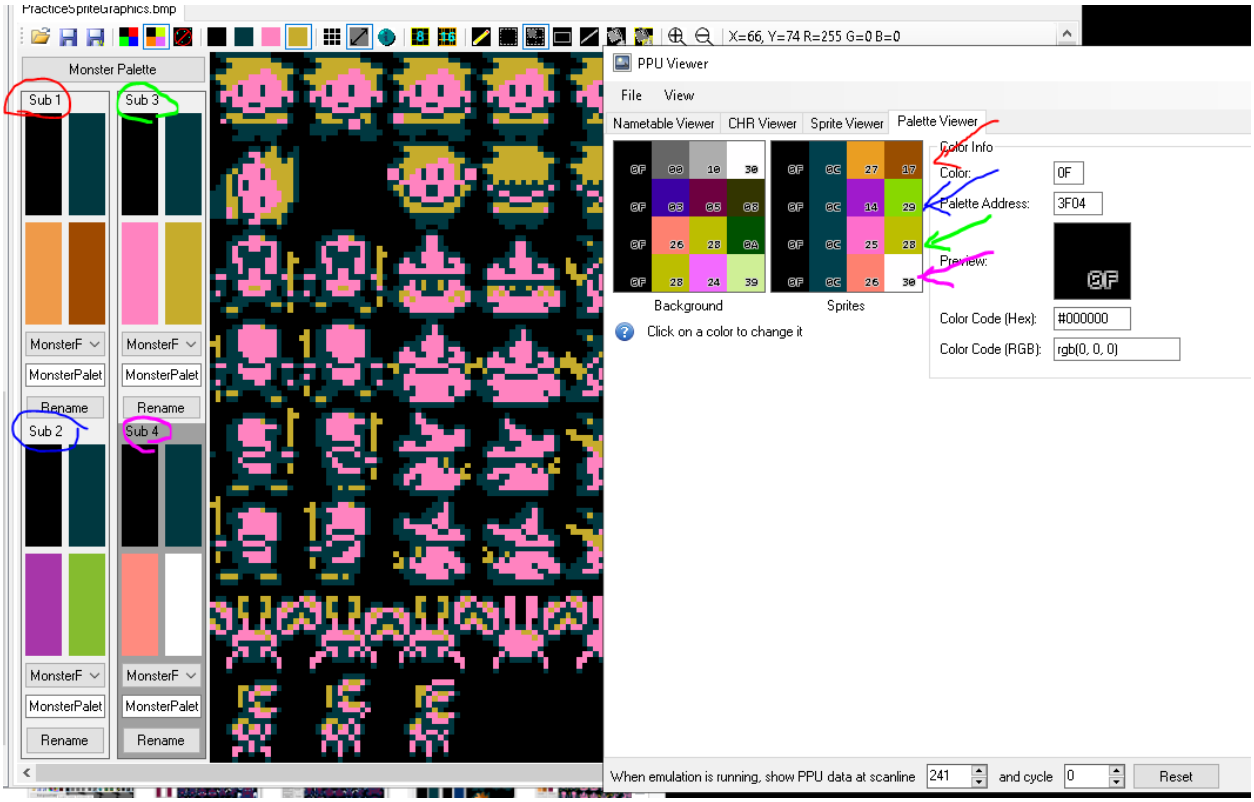
routine started. I then selected down to the end of the loop, copied, and pasted.

Then I made the following changes:

- The PPU address I'm going to write to - rather than writing to $3f00, I want to start writing to $3f10, which is where my sprite palettes start. *If you've got a good grasp on what's happening, you might understand this is a redundant step, since every time we write to $2007, the position updates by one, and our last position would have been $3F0F. Without even setting the address values, this would automatically write the next write to $2007 at the address $3f10. The only reason I'm doing this is to make it very clear what is happening for those who are still getting used to addressing.*
- I changed the name of the loop - I added "sprite" to it, and to it again at the bottom where it sets to cycle. If the names were the same, the assembler would yell at me and tell me that the labels were already defined.
- Instead of reading values from myPaletteTable, we want to read values starting at objectPal_0. This is only 4 values, but then objectPal_1 is four values, objectPal_2 is four values, and objectPal_3 is four values. Writing these 16 values will give us the exact setup that we created in our GUI.

```
38              C_LIGHT_GRAY = #$10
39  ;;;BACKGROUND COLORS
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 offest 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  ;;; SPRITE COLORS
63      LDA #$3f ;; the high byte of the destination
64      STA $2006 ;; gets written to $2006 first
65      LDA #$10 ;; the low byte of the destination
66      STA $2006 ;; gets written to $2006 second
67                  ;; WE ARE NOW ABOUT TO WRITE TO $3f00
68
69
70      LDX #$00
71      doLoadSpritePaletteLoop:
72          LDA objectPal_0,x           ;; load the value from
73                                      ;; objectPal_0, but
74                                      ;; with the offest of whatever
75                                      ;; value is in the X register
76          STA $2007                   ;; Write it to vRam
77          INX                         ;; increase the x register
78          CPX #$10                    ;; did X hit 16 yet?
79          BNE doLoadSpritePaletteLoop     ;; if it did not, do the loop
80                                          ;; again. If so, flow forward.
81
82
83
```

Now, if you run your game, open the PPU by clicking control-P, and click on the palette viewer, yo should see that the colors on the right side of the viewer match the colors we set up in our NESmaker palette GUI.



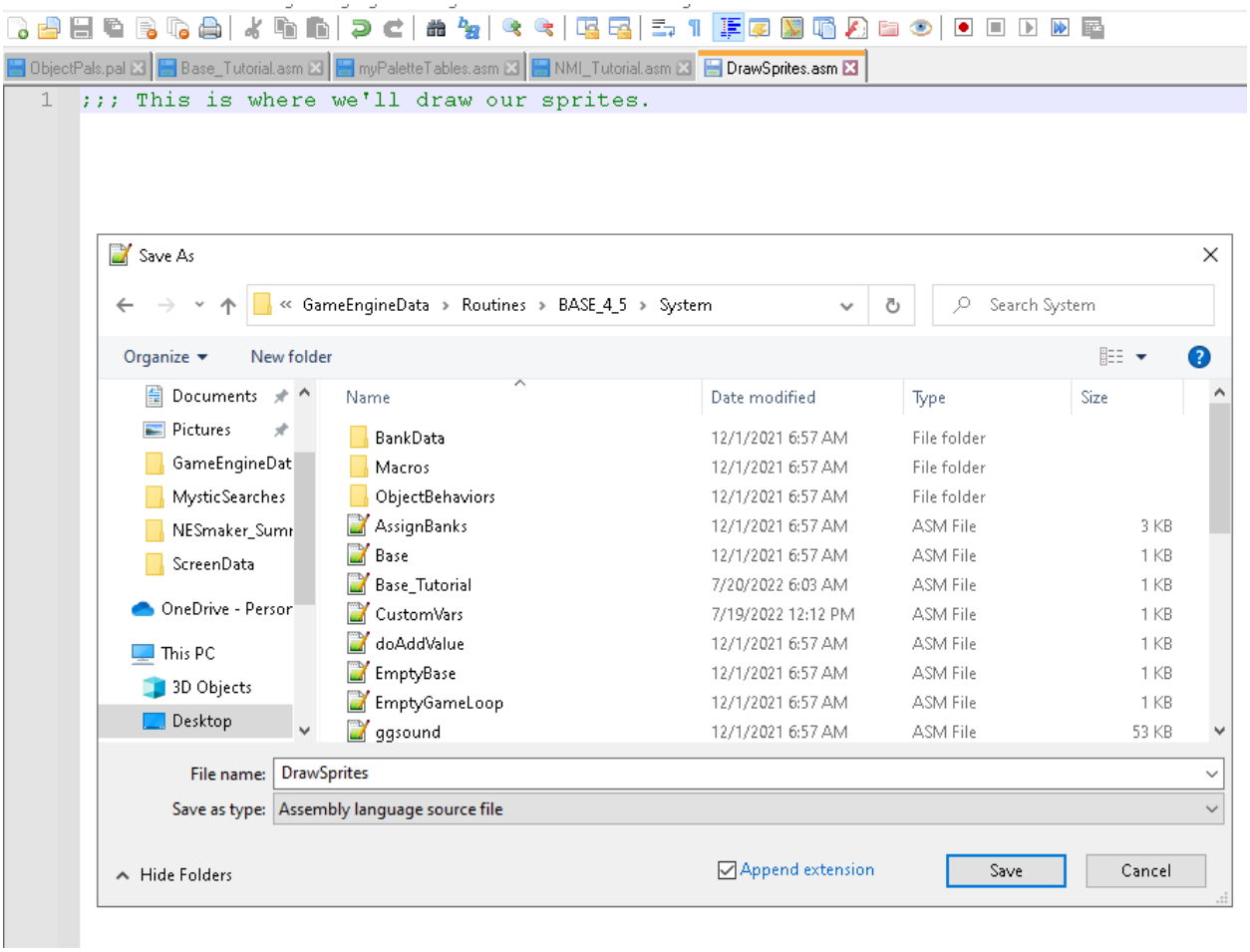
We can now finally begin drawing our sprites to the screen.

Step 7: Setting up a simple draw routine in our Main Game Loop.

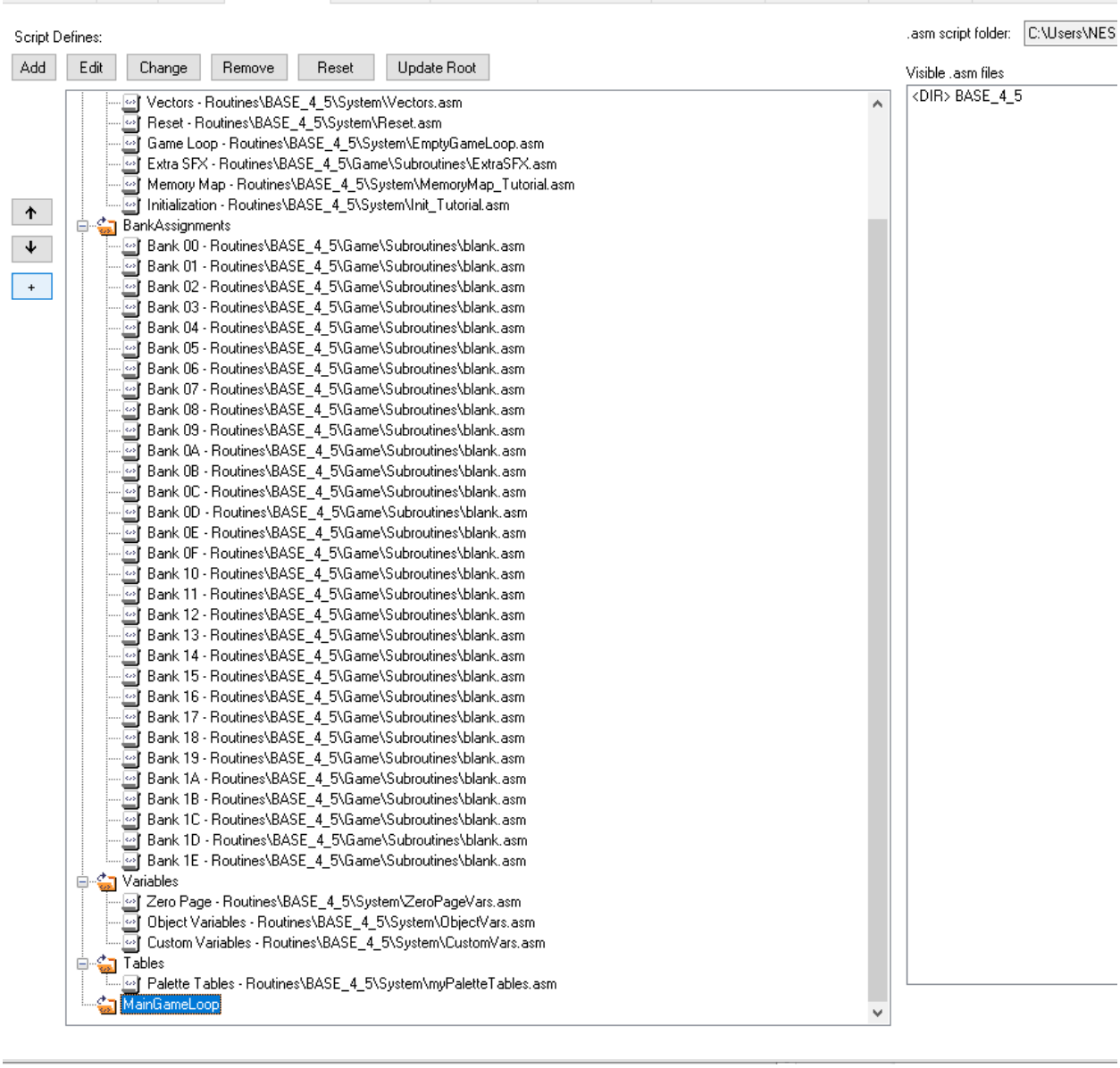
Our drawing system is something we're going to continue to revise and change as it grows more complex. This is a great example of how our Script Settings interface can be very beneficial. We're going to make a new Script Definition that handles drawing and then include it into our main game loop. Then, any time we need to make changes, rather than scan through our entire code or go hunting deep in a folder for the right file, we can just find it on our Script Settings list, click edit, and start making modifications. Alternatively, we can also make alternate drawing scripts if we want to test out some new features without breaking our game. That way, we can always go back to the previous version very easily without interrupting the rest of what we've already set up.

In our text editor, create a new script and call it DrawSprites.asm. Put a comment in it somewhere so that it can save, and make sure it is set to Assembly Source File when you save it.

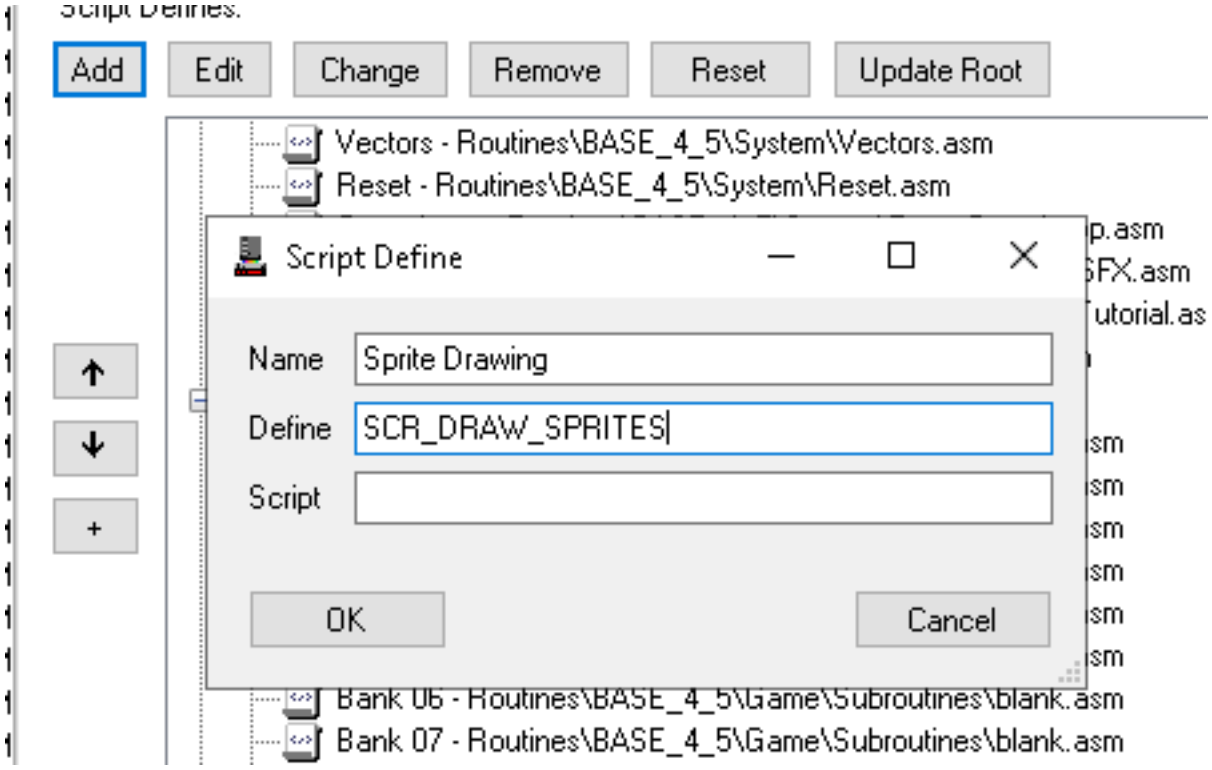
The best place to save it for this project's purposes would be to Root \ System, though by now you should have a pretty good idea of how to place it wherever you want inside the root folder and be able to attach it properly.



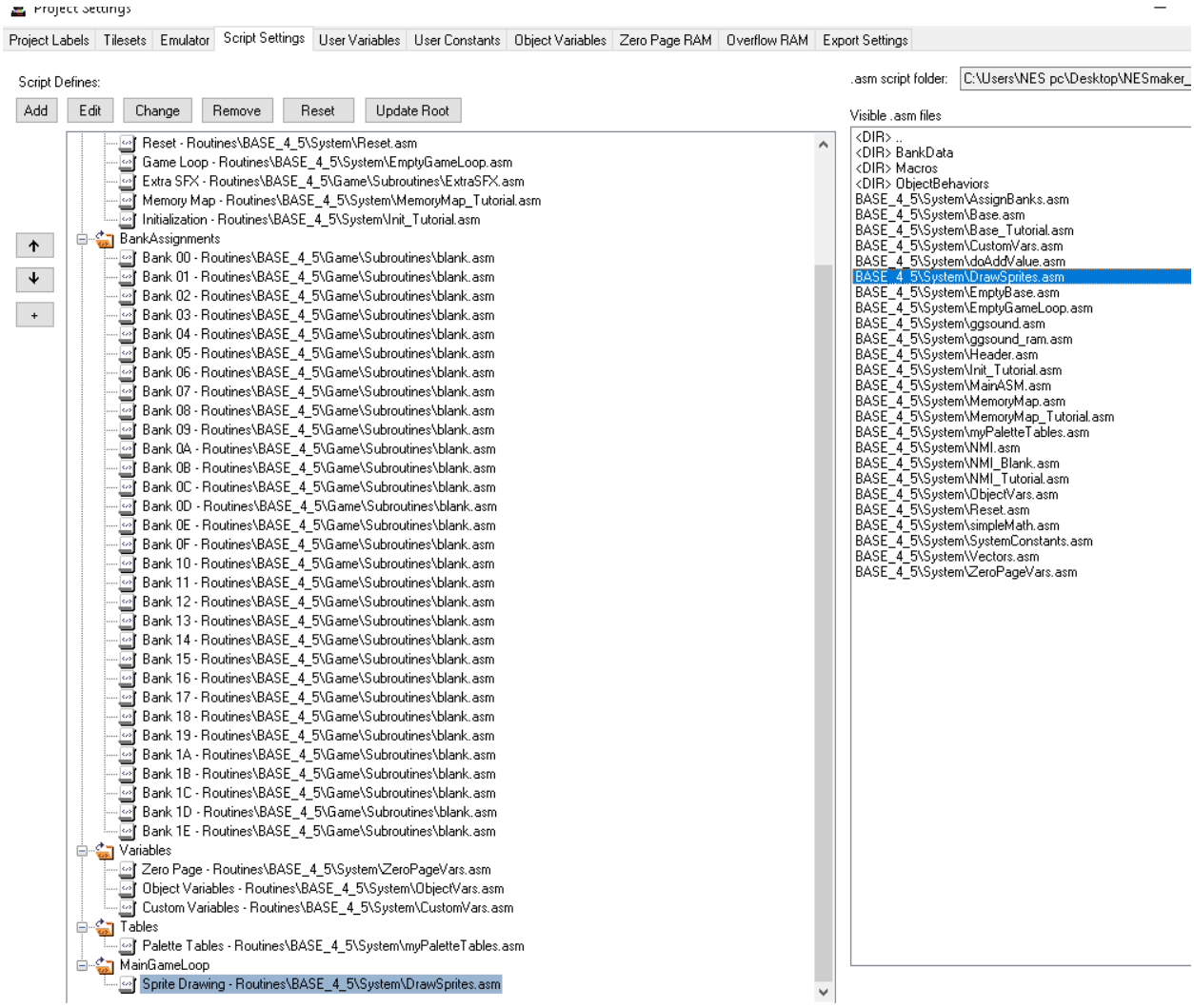
In your script settings, click the plus sign on the left of the script defines to create a new group. It will appear at the bottom. Right click on this group to rename it to MainGameLoop. This is simply an organizational step, not a functional one. We will put any high level scripts (sprite drawing, object management, physics, collisions) that relate to our MainGameLoop functionality inside this folder. Again, this folder structure is simply a management system for the GUI and does not relate to where scripts actually are included in the game.



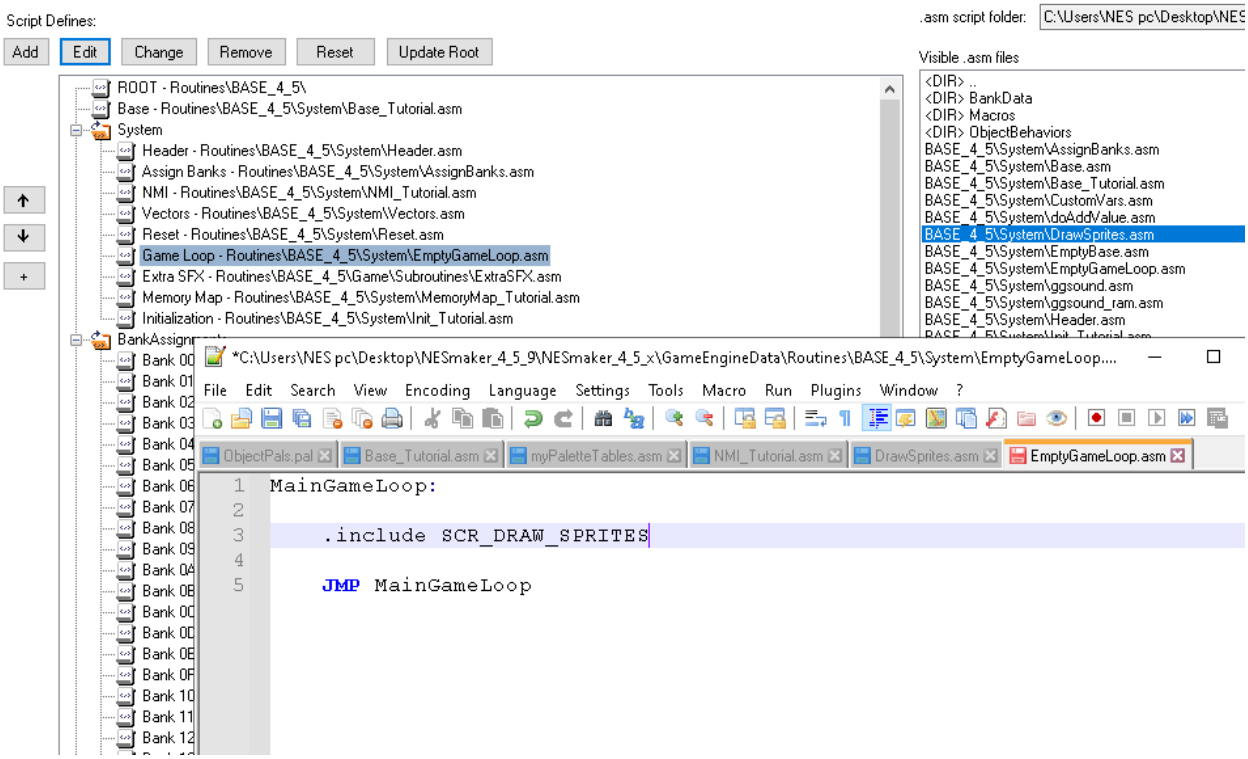
With MainGameLoop folder selected, press the Add button at the top of the Script Defines list. We will call this Sprite Drawing, and the script definition will be SCR_DRAW_SPRITES.



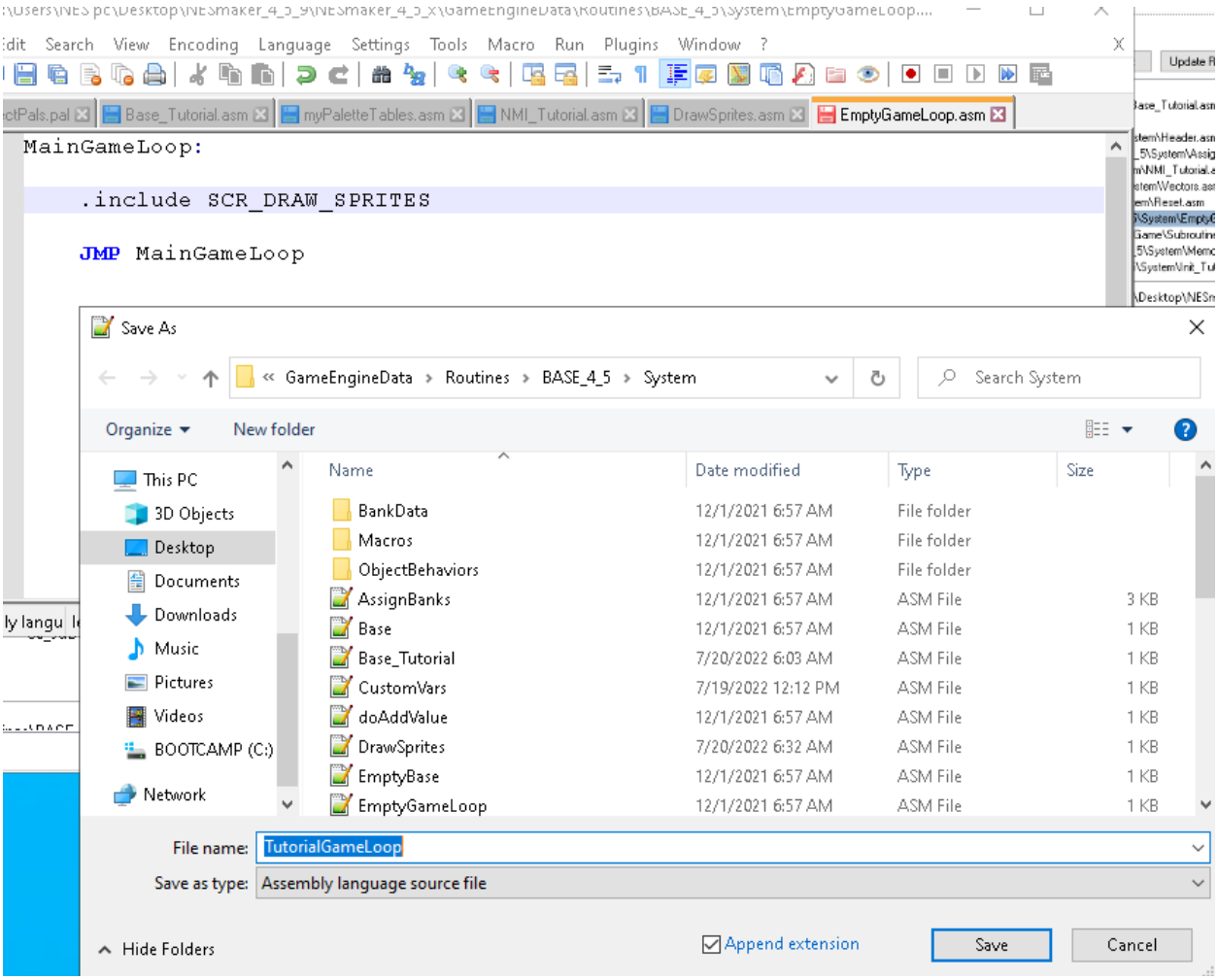
Then, attach our newly created sprite drawing script to this using the script finder window.



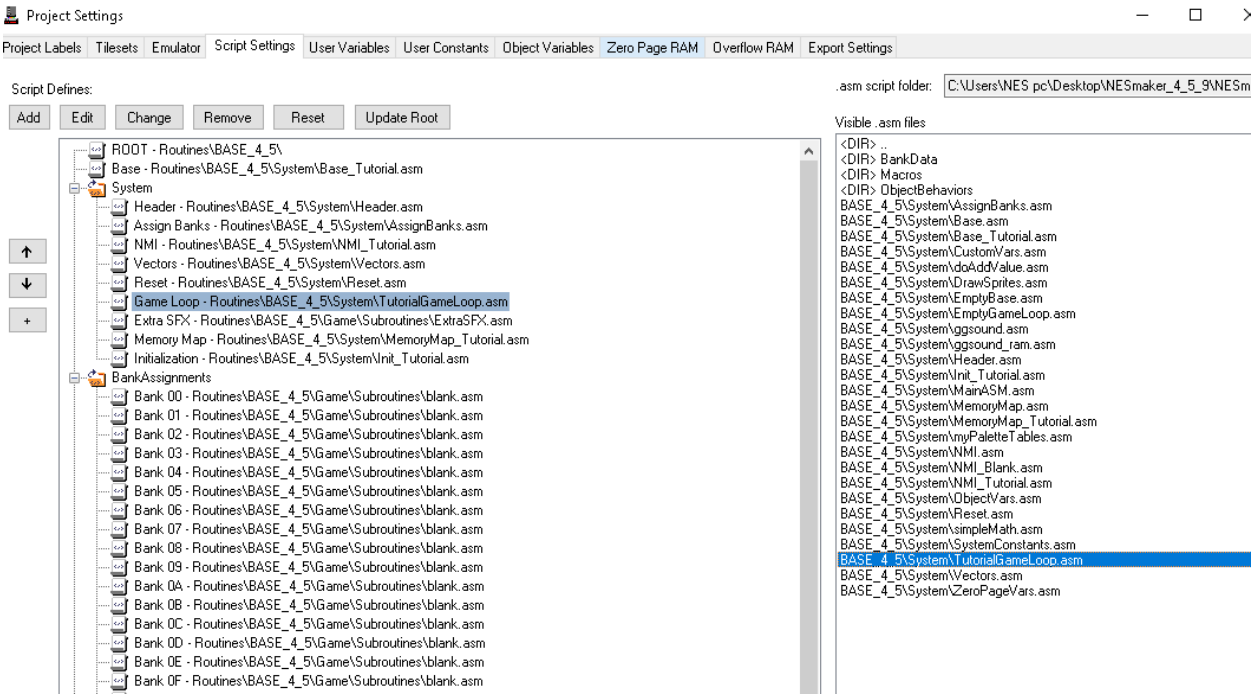
Next, we need to include this in our Main Game Loop. Scroll up the list and find



Notice, this is no longer an “empty game loop”. We may want to keep our empty game loop in tact, so rather than just saving this file in our text editor, let’s go to File->SaveAs. We’ll rename it TutorialGameLoop.



Now, we need our program to reference our new TutorialGameLoop rather than the EmptyGameLoop file for its game loop. In the Script Settings window, click on GameLoop and navigate to System. Double click on TutorialGameLoop to set this script definition to that new script.



We have preserved the Empty Game Loop, but our program will no longer read from that file. Instead, now it will read from the Tutorial Game Loop. We can make changes to the TutorialGameLoop file, and if we ever want to A/B compare or return to this empty state, we can always re-define Game Loop as Empty Game Loop instead.

This new TutorialGameLoop does one thing, and that's run the drawing code, which is currently blank. Next, we will start manually drawing sprites to the screen.

Step 8: Drawing our first sprite to the screen.

In the Script Settings, edit the sprite drawing script. We now have to jump back conceptually to a few things we talked about all the way back when we set up our memory map. When we set up our memory map, we reserved some RAM space for sprite data. We even called it SpriteRam. SpriteRam was a full page of ram we reserved for sprite related data; from address $0200-$02FF. During the NMI when rendering is not happening, all data from $0200-$02FF is transferred to the PPU so that it can properly

render the sprites during the next frame. These are things we already set up.

So what goes in those addresses? Well, it's actually pretty simple. We actually already touched on it briefly, but now after having gone through all that we have, it might make a bit more sense. Every sprite requires four bytes in order to be drawn. The reason that the NES has a 64 sprite limit is because 4 byte values x 64 sprites = 256 byte values, or one full page of data.

A sprite's byte anatomy is this:

```
Byte 0 = The Y value of the sprite, 0-255
Byte 1 = Which of your 256 sprites is being drawn from the sprite table?
Byte 2 = the attribute data, including what sub palette this sprite will use
Byte 3 = the X value of the sprite, 0-255.
```

So the first sprite to be drawn uses:

- 0200 to determine the vertical position
- 0201 to determine which sprite is being drawn
- 0202 to determine what colors and other attributes affect the sprite
- 0203 to determine the horizontal position

The second sprite to be drawn uses:

- 0204 to determine the vertical position
- 0205 to determine which sprite is being drawn
- 0206 to determine what colors and other attributes affect the sprite
- 0207 to determine the horizontal position

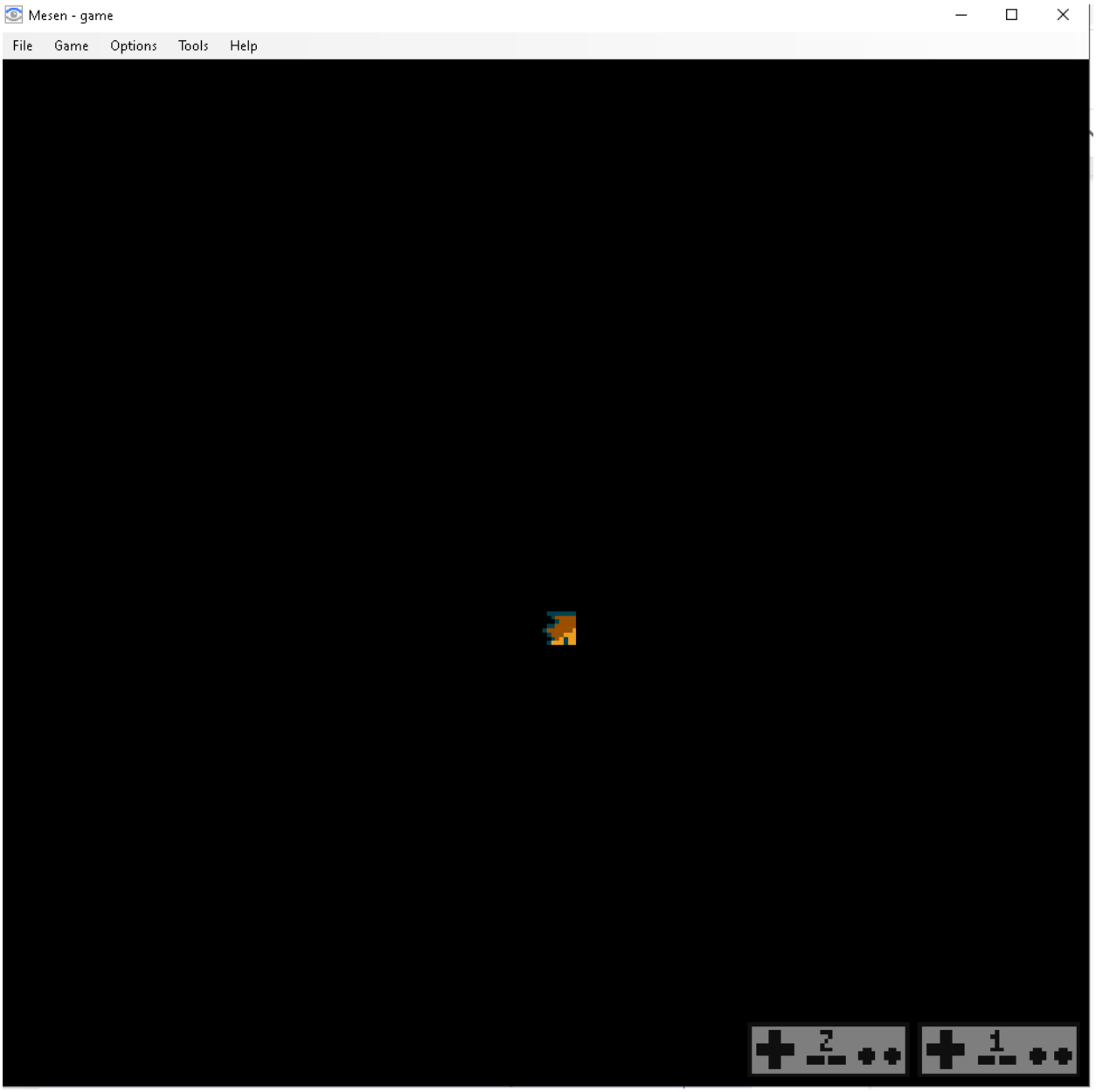
And so on and so forth. The sixty-fourth sprite to be drawn will determine its horizontal position at 02FF. And then, during the NMI, all of those RAM values, 0200-02FF, will be pushed to the PPU. That is the data it will use to render the sprites during the next frame.

To draw our first sprite in a very longhand way, we can simply fill in those blanks, like this:

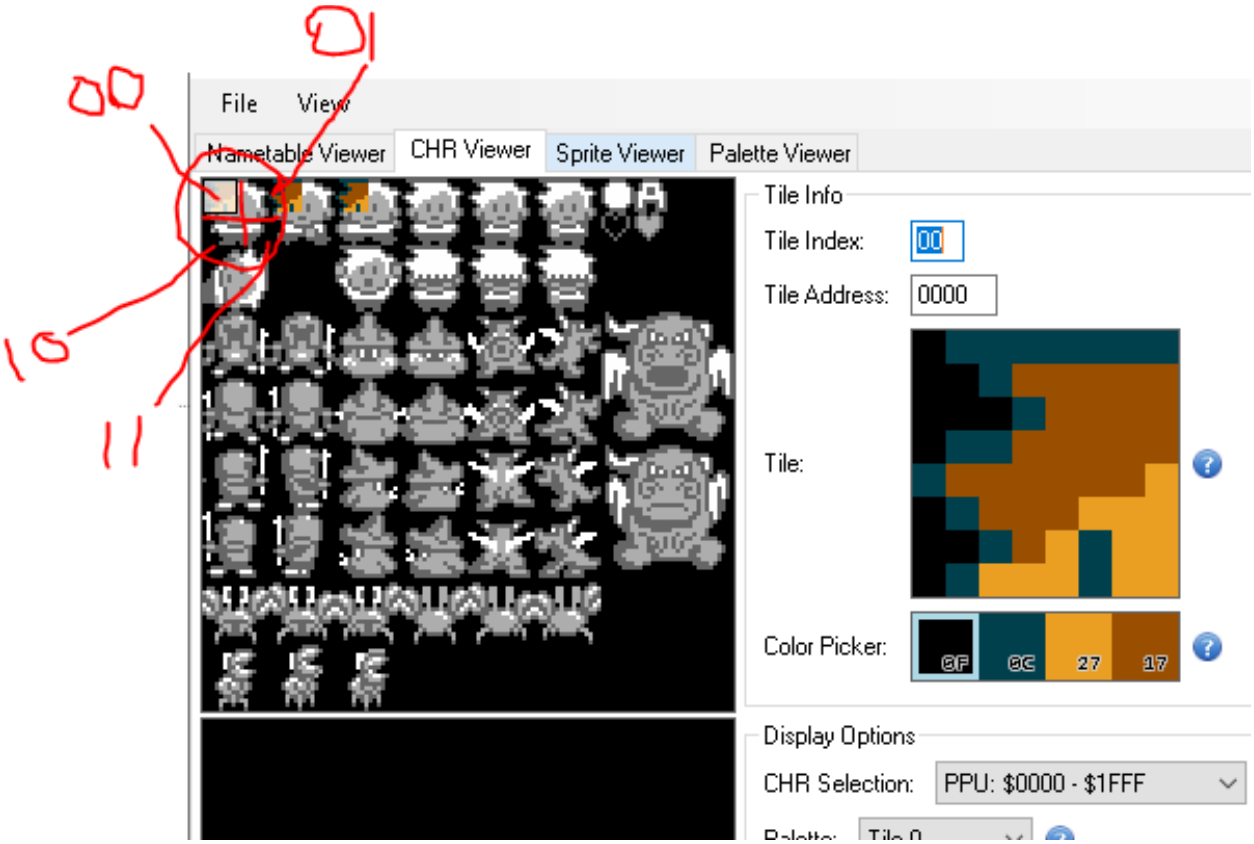
ObjectPals.pal | Base_Tutorial.asm | myPaletteTables.asm | NMI_Tutorial.asm | DrawSprites.asm | TutorialGameLoop.asm

```
1  ;;; This is where we'll draw our sprites.
2
3  LDA #128 ;; decimal value 128
4  STA $0200 ;; $0200 is y value of first sprite to be drawn
5  LDA #$00 ;; Load zero
6  STA $0201 ;; $0201 is the value to pick from the sprite pattern table
7            ;; zero will pick the first 8x8px area in the top left
8            ;; corner of the currently loaded table.
9  LDA #$00 ;; Load zero
10 STA $0202 ;; $0202 is the attribute data for this sprite.
11           ;; the last two bits are sub palette. Since
12           ;; this would result in 0 and 0 for the last two
13           ;; bits, this would be sub palette zero
14 LDA #128 ;; decimal value 128
15 STA $0203 ;; $0203 is the x value of the first sprite to be drawn.
```

If you run and test your game, you will see that the top left corner of your player's head shows up more or less in the middle of the screen.



If we look at the pattern tables in the emulator by opening the PPU viewer and looking at the CHR viewer, we can see what graphics we would need to construct the entire character. We would need to draw tiles 00, 01, 10, and 11 in the proper locations.



And if we think out where we want to draw them, if we can consider it relatively:

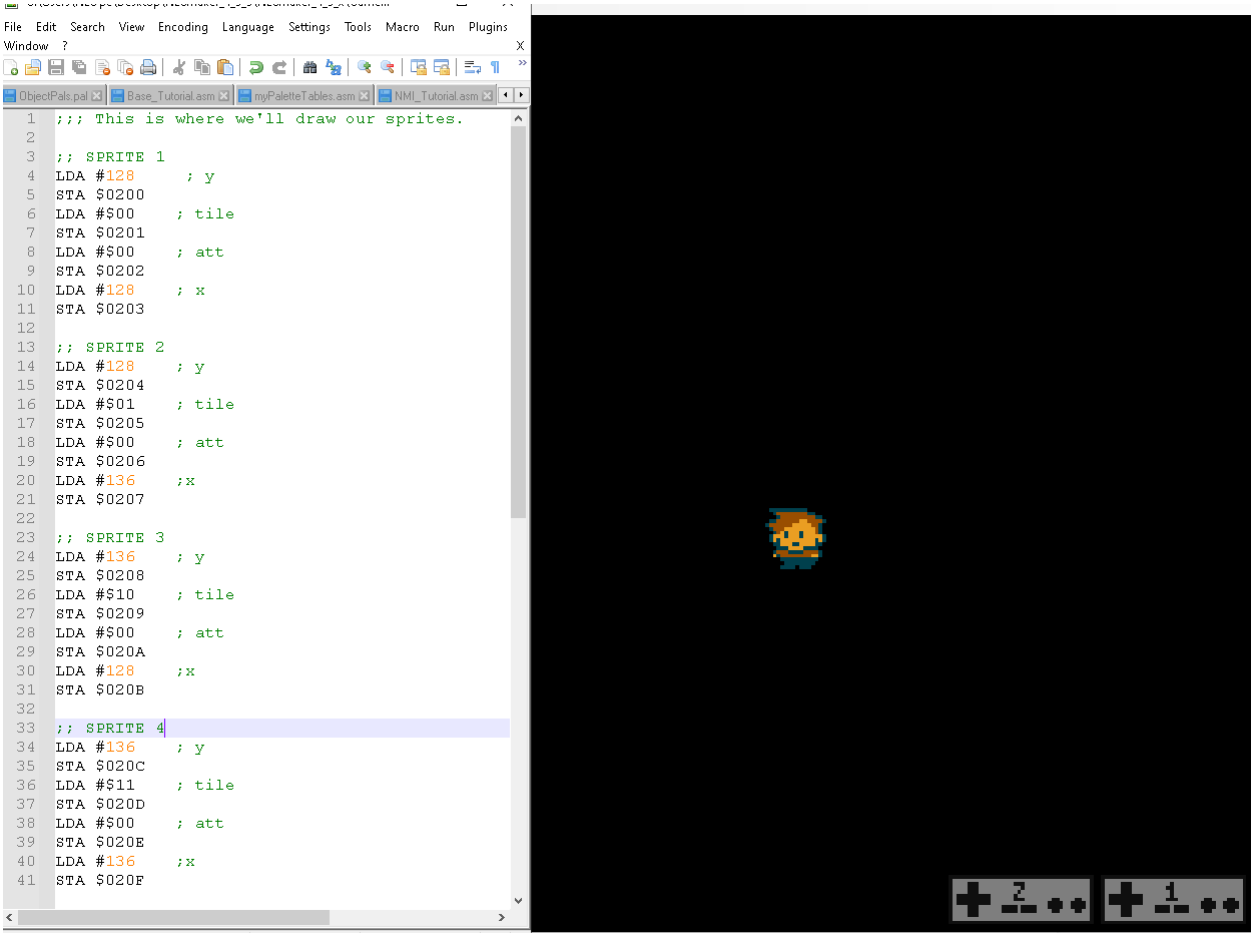
First tile: 0 pixels down, 0 pixels over.
Second tile: 0 pixels down, 8 pixels over.
Third tile: 8 pixels down, 0 pixels over
Fourth tile: 8 pixels down, 8 pixels over.

So, if the first tile is drawn at 128,128, the coordinates of each tile would be:
(x) (y)
128, 128
136, 128
128, 136
136, 136

They'll all use the same attributes. So basically, we want to keep these numbers in mind when trying to draw this character to the screen:

	Y value	Tile #	Attribute	X value
SPRITE 0	128	00	00	128
SPRITE 1	128	01	00	136
SPRITE 2	136	10	00	128
SPRITE 3	136	11	00	136

Here's what the script would look like to manually draw this entire character to the screen (with the comments removed for easy reading)..



```
;;; This is where we'll draw our sprites.

;; SPRITE 1
LDA #128     ; y
STA $0200
LDA #$00     ; tile
STA $0201
LDA #$00     ; att
STA $0202
LDA #128     ; x
STA $0203

;; SPRITE 2
LDA #128     ; y
STA $0204
LDA #$01     ; tile
STA $0205
LDA #$00     ; att
STA $0206
LDA #136     ;x
STA $0207

;; SPRITE 3
LDA #136     ; y
STA $0208
LDA #$10     ; tile
STA $0209
LDA #$00     ; att
STA $020A
LDA #128     ;x
STA $020B

;; SPRITE 4
LDA #136     ; y
STA $020C
LDA #$11     ; tile
STA $020D
LDA #$00     ; att
STA $020E
LDA #136     ;x
STA $020F
```

This long hand way of drawing 64 variable sprites at variable positions would be completely untenable, but learning how to do it this way hopefully gives some good insight as to what exactly is going on and how to manually draw sprites when needed.

Three of the four values that make up a sprite are pretty easy to understand. The x value is its horizontal position. The y value is its vertical position. The tile # is what number tile on the loaded sprite sheet it will draw. The attribute byte has some characteristics that definitely warrant going over.

Step 9: Playing with the attribute bytes

As we've already gone over at length, a byte is made up of 8 bits. Sometimes, this byte represents a complete value, from 0-255. However, sometimes bytes are used for their individual bit information. Attribute bytes are a good example of this.

Each bit can either be on or off, 0 or 1. When evaluating how to draw a sprite, the PPU looks at one bit of the attribute byte at a time, and each of these bits tells the renderer to do something slightly different.

The highest bit, the one all the way to the left, bit 7, determines whether or not a sprite is flipped vertically. If it sees a 1 in this bit slot, the sprite will be flipped vertically. If it sees a zero, it won't.

Bit 6 determines whether or not it will be flipped horizontally.

Bit 5 relates to whether the sprite is drawn in front of or behind background pixels. This is of no consequence to us right now since we only have a fully transparent background, but if we did have a background, any non-transparent(color 0) pixels would cover up the drawing of this sprite if bit 5 was set to a 1. If it was set to a zero, the sprite would be on top of the background (this is the most common).

Bit 4, 3, and 2 are unimplemented and not used for anything.

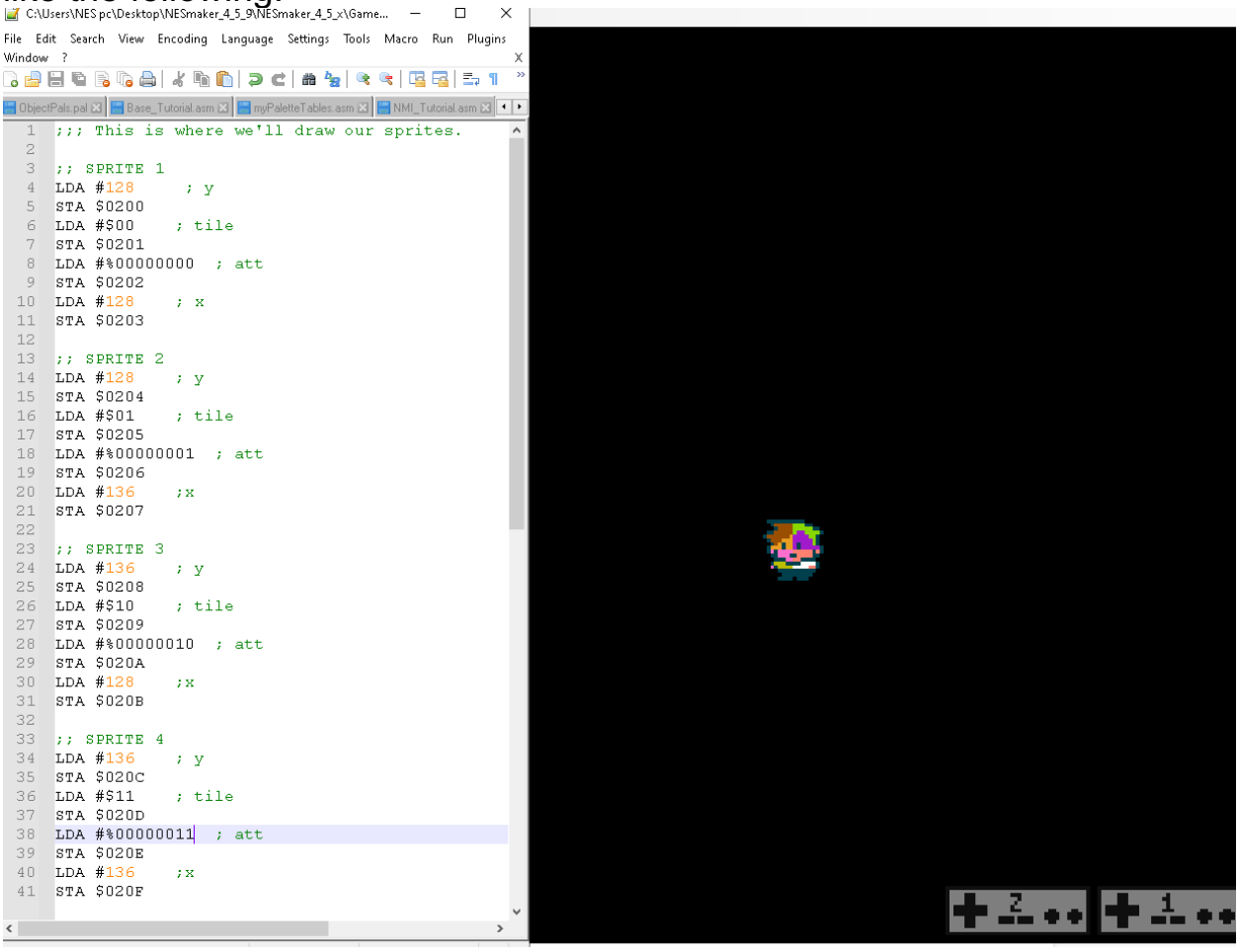
Bit 1 and 0 determine which subpalette a sprite uses. So if those last two bits read 00, it will draw with the four colors in the first subpalette. If the last

two bits read 01, it will draw with the four colors in the second subpalette. If the last two bits read 10, it will draw with the four colors in the third subpalette. If the last two bits read 11, it will draw with the four colors in the fourth subpalette.

For fun, let's make our player's four sprites use the four different subpalettes. Right now, our attributes are all 00, and written as such in hex. We're going to change to write in binary so that we can manually write each bit.

The four attribute bytes will be written:
#%00000000 for the first sprite
#%00000001 for the second sprite
#%00000010 for the third sprite
#%00000011 for the fourth sprite

Changing the attribute bytes in the script will give us something that looks like the following.



You can continue to play with these values - try setting the flip values to 1s instead of 0s to see the sprite flip. Change the x and y values for each sprite to draw them to different parts of the screen. See if you can reconstruct the player upside down and backwards by flipping the sprites and moving the x and y of each into the right place.

End of Lesson 5.

By the end of this lesson, you should have learned a little bit about how to utilize NESmaker's GUI to export data, and then how to work that data into your game. You should have reinforced creating custom scripts and working with script defines. You should have also reinforced writing to the PPU and working with logic loops. You should have learned the basics of drawing sprites to a screen.