

Table Of Contents

Downloading NESmaker.....	1
Making a Plan.....	4
Starting a New Project.....	8
Navigating the Interface.....	12
Creating the Game States.....	20
NESmaker Graphics Primer.....	22
The Pixel Editor.....	41
Working With Graphics.....	44
The Screen Painter.....	52
The Player Game Object.....	66
Designing Input.....	86
Tile Collisions.....	100
Animation Tools.....	123
Monster Objects.....	139
HUD Basics.....	162
Creating a Start Screen.....	177
Creating a Projectile Object.....	203
Creating a Pickup Object.....	213
Win Screen.....	218
Adding Music and Sound.....	225
Working with Assets.....	230
Saving a New Module.....	235
Flashing to a Cartridge.....	239

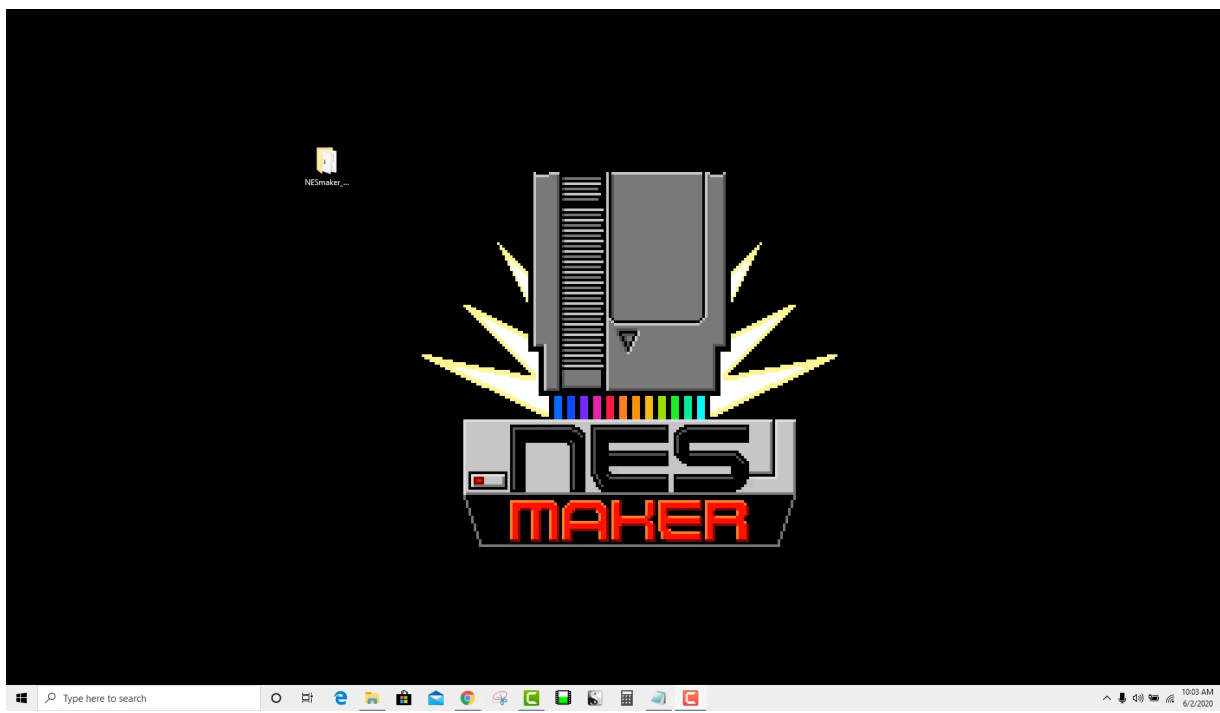
Downloading NESmaker

WELCOME TO NESMAKER 4.5

Welcome to NESmaker 4.5. This is the first official introduction to this version of the software. This instructional will help you create a brand new game that will be playable on the Nintendo Entertainment System, on NES-compatible hardware, and on most emulators. This series is for users of all skill levels, even people with zero experience in game design.

As we make our simple game, we are going to examine each part of NESmaker's tool chain, discuss the technical limitations of the 8-bit system, start glimpsing a bit under the hood and the code that actually powers the game, and take a rudimentary examination of game design in general. By the end of this instructional, you should have a solid understanding of how to use the tool, and how to synthesize various skills learned to bring infinite potential ideas to life.

This instructional presumes that you have a working copy of NESmaker 4.5.x, that you have downloaded the zip file of the newest version of the software, and that you have unzipped the NESmaker folder onto your desktop as seen in the image below.



If you experience trouble opening NESmaker, there are three common causes.

- If you're using a 32-bit operating system, but are trying to open a 64-bit version of the software, it will not open. If the software won't open, check your operating system. If it is, in fact, 32-bit, simply download the 32-bit version of NESmaker instead.
- It is possible that overzealous virus scanners on your system blocked or quarantined some of the small binary files that make up code required for NESmaker and games created with it. Try temporarily disabling your virus protection, re-download the software, and extract the contents of the zip file to your desktop.
- Make sure that wherever you place NESmaker on your computer has administrator level rights. The most common place to put the NESmaker folder is on the Desktop, and this should solve most problems in this regard.

Requirements for This Instructional Series

You will need a copy of NESmaker 4.5.x (any NM4.5 version) and a particular module specifically designed for the steps to follow. The module that you need is called `Tutorials_4_5_6`. This module may not be included with your NESmaker download, but it is very easy to add.

Simply go to www.TheNew8bitHeroes.com/downloads, which is the same location where you likely downloaded the software. Navigate to the `MODULES` section of the download page, and you should see a download for `Tutorial_4_5_x`.

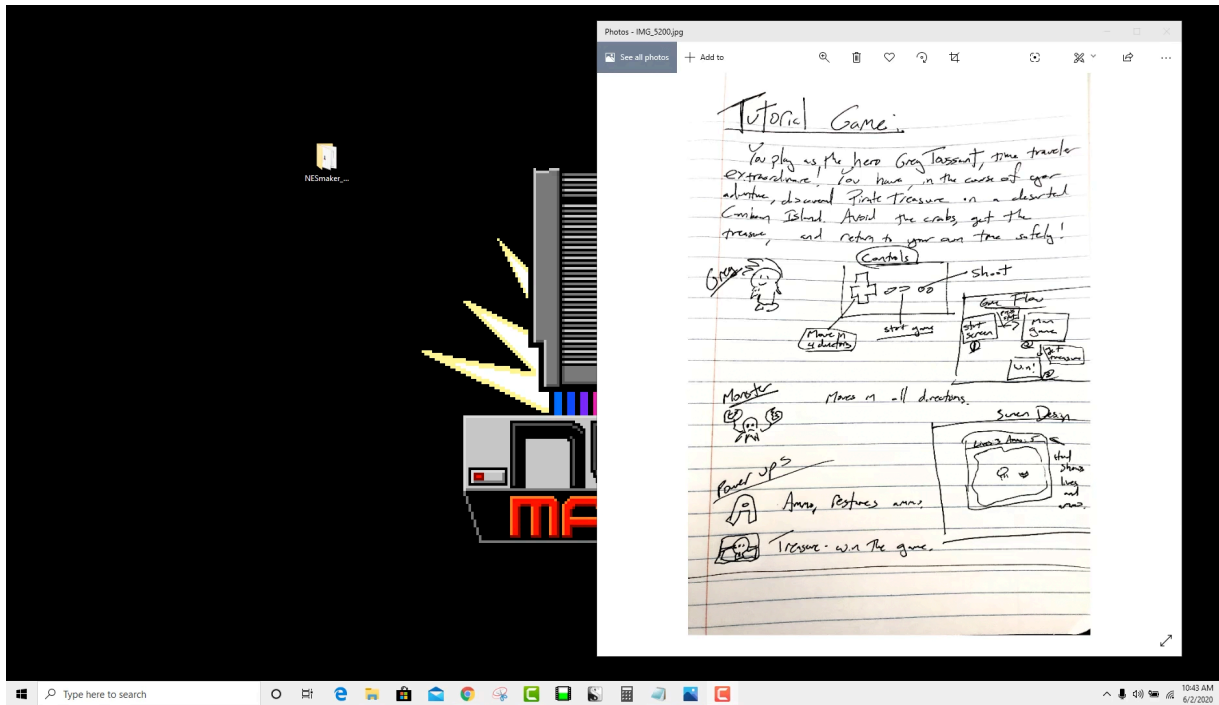
Download this file to your computer. Unzip the folder. Copy the `Tutorial_4_5_x.mod` file from inside that unzipped folder into the `Modules` folder, which is inside your NESmaker folder. Now, when you open the software, this module will be available to you.

Making a Plan

Making the Plan

Before even opening the software to explore, let's conceptualize a simple game that we're going to create. It's often a great idea to even just break out a pen and paper to sketch out a few ideas about what you want to make happen. For a very simple game like we're about to create, we can squeeze most of the ideas onto a single page. For more complicated projects, it's a good idea to have a solid design document built. Now, these documents are often living documents. They are subject to change as you play and experiment and find what works and what doesn't, but there is definitely an advantage to knowing what it is you're trying to accomplish before starting.

The following is a hilariously brief concept document that will provide a roadmap for what we will create over the course of this tutorial.



Summary of the game experience:

You play as the hero, Greg Toussaint, time traveler extraordinaire! You have, in the course of your adventure, discovered pirate treasure on a deserted island. Avoid the crabs, get the treasure, and return safely to your own time.

Even this short, quick summary gives some base context about the type of game we want to create, which allows us to envision it on a conceptual level in our head. Also, thinking about and plotting out the controls helps. This game will have a top-down perspective and so our d-pad will move our player in all four directions. The start button will start the game. The b-button will shoot.

Plot the controls:

- D-pad will move the player in four directions
- Start button will start the game
- B-button will shoot

It's also a very good idea to think out the game flow and get the thoughts on paper. When you first power the game up, what will you see? How do you progress to the next phase of the game, and what does progression even mean in your game? Is there an end to the game? What does the screen show in a win state? What does *winning* mean in the case of this game? What happens when the player loses? What does *losing* mean in the case of this game? For this project, we'll keep it incredibly simple.

Game Flow

- When the game starts up, we will see a start screen.
- On the start screen, pressing the START button on the controller will take us to the first playable game screen.
- When we collect all of the treasures, we will advance to a next level.
- If we run into a monster object, the game will reset and return us to the start screen to try again.
- If we collect all of the treasures on the last level, a WIN screen will load. When on the win screen, if we press the START button, the game will reset and return to the start screen.

Next, let's conceptualize all of the objects in the game. For this project in keeping it very simple, we want a player object, an obstacle object, a prize object, and a win object. Learning how to work with these basic object types opens up infinite combinations of possibilities.

Objects

- **PLAYER OBJECT** - the player will be able to control this object with the d-pad and controller buttons. The d-pad will move the character in all four directions, while the b-button will create a

projectile at the user's position, shooting in the direction the player is facing.

- **PROJECTILE** - this is the player's weapon. When it hits a solid object or screen edge, it will destroy itself. When it hits a monster, it will destroy the monster and itself.
- **MONSTER** - this monster will aimlessly wander around the screen, changing directions when it runs into an obstacle or a screen edge. A collision between the player object and the monster object will trigger the LOSE game state.
- **AMMO** - the player will be able to use the b-button to fire projectiles only if he has ammo. Collision with the ammo object will re-fill the ammo meter, which will appear at the top of the screen. Shooting a projectile by hitting the b-button will subtract one from ammo.
- **TREASURE** - When the player collides with the treasure object, it will trigger the win state.

Lastly, we want to consider basic screen design and what information we want exposed and available to the player. For this game, we will use a basic HUD at the top of the screen.

Screen Design

- There will be a large playable area with a top-down perspective.
- There will be a basic HUD at the top of the screen, which will show the following information:
 - **LIVES**
 - **AMMO**

While all of this seems very standard and logical, it is still always a great idea to think it out first. Now we know that we need to leave room at the top for a HUD. We know that we need to be keeping track of variable lives and variable ammo.

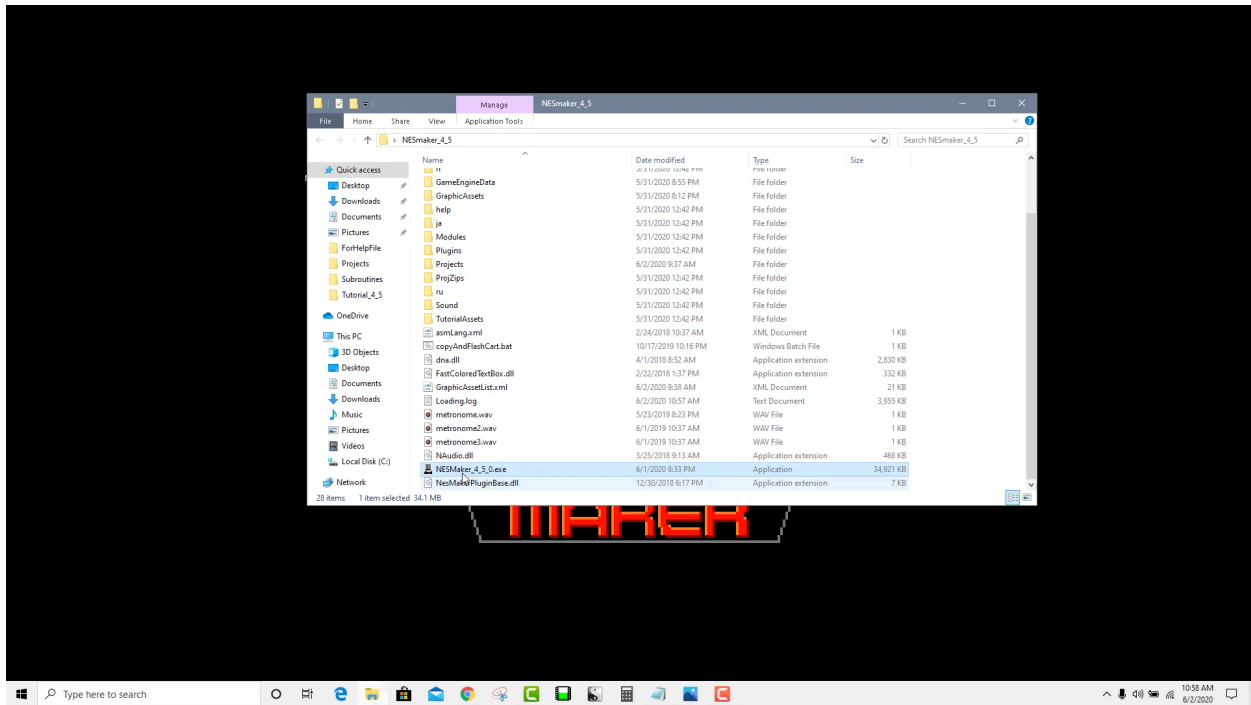
We know that we're going to need text as part of our tileset if we want to draw text to the screen. These types of details are basic creative decisions best established in advance. Even this simple chicken scratch design document mocked up in just a minute or two will be immensely helpful to reference while setting up our first game.

Starting a New Project

Starting a New Project

Step 1: Open NESmaker

Open the NESmaker folder. Scroll down to find the NESmaker_4_5_x application. Depending on your exact version, this may simply say NESmaker, or it may have a variable suffix at the end of NESmaker_4_5_. The last number that follows the underscore would indicate a version. For instance, at the time of writing this, the latest version of NESmaker is NESmaker_4_5_9. Whatever the suffix after NESmaker, this .exe file is the NESmaker application.



Step 2: Registering Your Software

If this is your first time opening NESmaker, you'll see that your only real option is to activate the software. To do this, you'll need the email that is associated with your license and your license number. Enter them into the appropriate fields, and create a password in the last field.

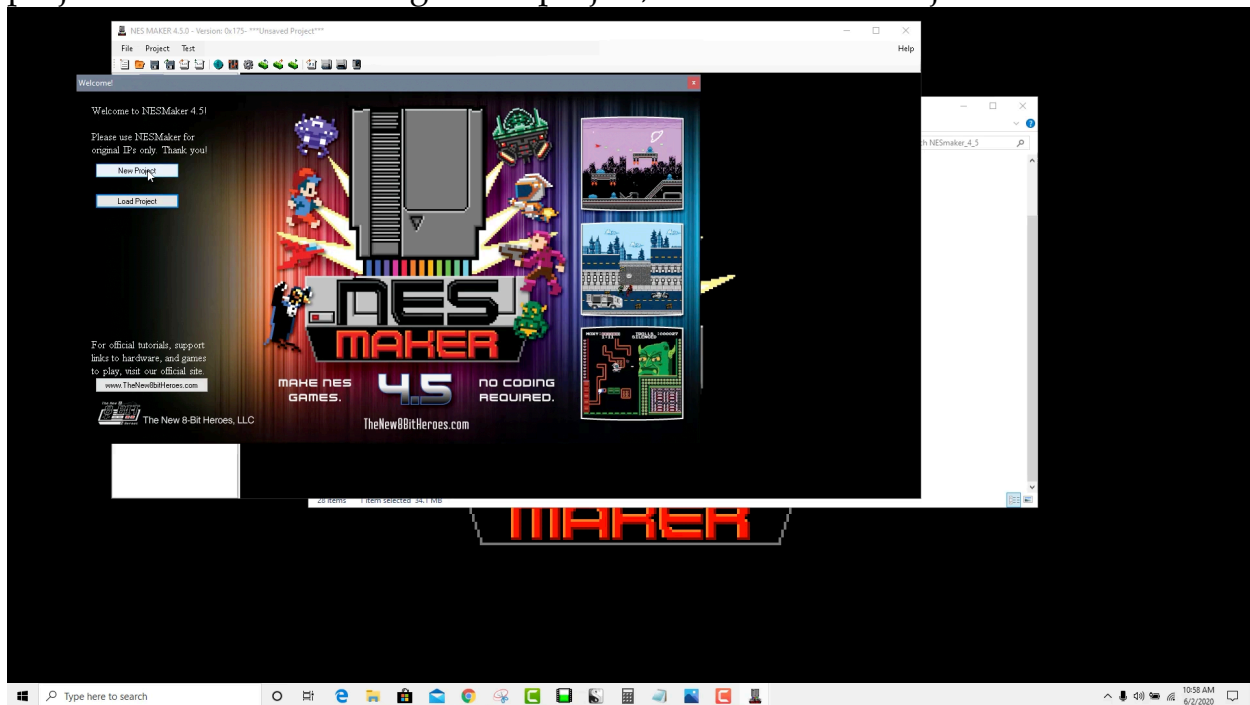
One common problem with activation is use of the wrong email address. Make sure that you are using the same email with which your activation code was purchased.

If you are having any activation troubles, including if you're an existing user who sees an error or lockout message when opening the software, you can contact the team via a form on the CONTACT page of The New 8-bit Heroes website. We try to respond within 48 hours, and most times it is much quicker than that.

Step 3: Creating A New Project

The first thing you'll see is a splash screen. Here, you can see a link to The New 8-bit Heroes website where you can find support and tutorials, you can

load an existing project that you've already started, or you can start a new project. Since we're starting a new project, click the New Project button.

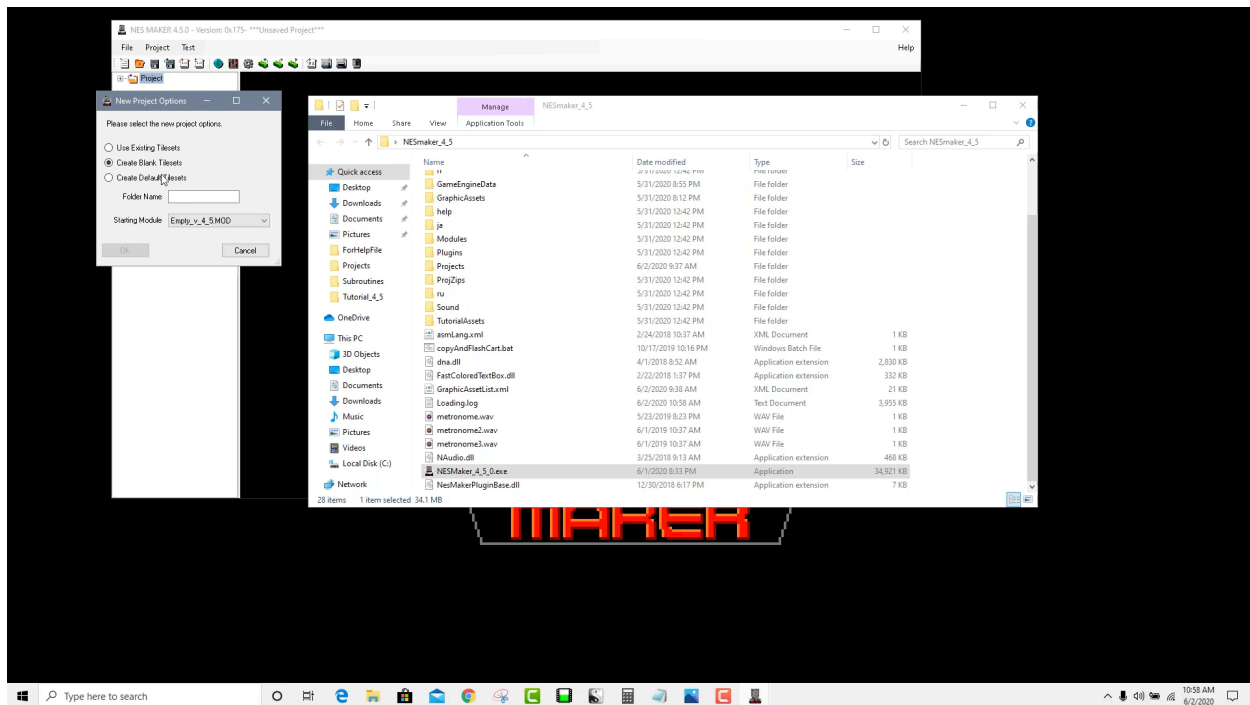


Step 4: Setting Up A Graphics Folder and Choosing A Module

Once we've opted to create a new project, we get a small pop up window. Here, we need to set up two things. The first is the graphics folder and the second is the starting module.

In almost every case, you'll probably want to choose Create Blank Tilesets for the graphics folder. Select this option. This will create a new folder of blank graphic assets specific for this project.

The other options are Use Existing Tilesets and Create Default Tilesets. If you choose Use Existing Tilesets, your project will be working with the template tileset folder rather than a unique folder for this specific project. If you wanted to generate a template of graphics that might apply to all future games, you might choose this method for your first project, and then you could use Create Default Tilesets to make a copy of the template tileset folder as your project folder. Those two options are really reserved for a more advanced or specific use case, so for now, we'll just continue with the Create Blank Tilesets option.



Step 5: Give it a Name

Enter a logical name for your graphics folder where it says Folder Name using only alphanumeric characters. Avoid special characters, and it is also advisable to use camel case rather than spaces. For instance, instead of My Game Graphics, call the folder MyGameGraphics. Instead of New Game 2, call the older NewGame2. We'll call this project TutorialGame.

Step 6: Select a Module

NESmaker 4.5 comes stocked with a handful of modules. Modules are basic code packs that can help you get started with different genres of games. One of the challenges in creating a tool to create games for a system that has such limited memory is that there isn't the same room for all possible options to be available all the time. For instance, a scrolling platformer has dramatically different needs than a top down adventure game, and each of these game types may need to utilize the memory in different ways. Modules are a way to generally optimize the code that is running under the hood to provide basic functionality for a specific genre of game.

However, modules should be thought of as starting points. They are rough templates that give a starting code base with common mechanics that a game of that genre may require. That does not mean that each module is limited to those things. Each can be customized and edited extensively. The more complex the genre of game, the more editing and customizing may be required. Editing and customizing a module will eventually be part of this instructional.

You'll even see a module called EMPTY, which is a bare bones skeleton that can be used by developers who want to create game engines from complete scratch rather than use one of the provided modules. That is obviously for advanced users, but it is good for users to know that that level of extension of the tool is possible.

For this project, we're going to select the module **Tutorial_4_5_x**. If you do not see Tutorial_4_5_x, please make sure that you follow the instructions in the Downloading NESmaker section.

Once you've selected the graphics folder option, given it a name, and selected the desired module, press OK.

Navigating the Interface

Navigating the Interface

The NESmaker work area is split into a standard software menu bar, a quick access toolbar, a project hierarchy, and a general workspace. Here, I name and give very brief description of all of the touch points in each. We will be going over the most commonly used and important ones over the course of creating our game, but you can use this for quick reference. To skip the reference, look ahead to Understanding The Module.

The Menu Bar

The menu bar is a fairly standard desktop application menubar. Depending on the interface, you may notice that additional menus are available that pertain directly to that interface, but there is a set of menu options present regardless of currently exposed interface. There are some items that serve legacy functions, but for clarity, here is a list of the basic main menu items and their functions.

File

New - creates a new NESmaker Project

Load - loads an existing NESmaker project.

Save - saves the current NESmaker project.

Save As - saves the current NESmaker project.

Export - mostly a legacy function, this allows you to export the fundamental components for your game to a custom location outside of the project folder.

Backup Scripts Folder - this allows a user to backup scripts as a new module

Project

Info - mostly a legacy function, this set some initialization states from a GUI for earlier builds of the software. However, the triggered screen group tab is still used to allow a user to toggle screen triggers for testing.

Screen Groups - mostly a legacy function, as there are more advanced methods for grouping screens in the map editor.

Project Settings - - the project settings dialog has a multitude of tabs. Inside these tabs range

everything from setting up global game variables to assigning and working with module scripts to exporting a meta file with all game information.

Import Project Module - this allows a user to overwrite existing project

module information

with a new project module. It is for advanced use.

Export Project Module - this allows a user to export project module information for import

into another project. It is for advanced use.

Run Project Script - this is part of importing a new module over the top of the settings for a project. It is for advanced use.

Test

Export - this exports the data from the current project without trying to assemble it or open it in an emulator.

Export & Test - this exports the data from the current project, runs it through an assembler, and opens the resulting ROM in an emulator.

Retest Last Export - this simply opens the last export in an emulator.

Make Cart - this exports the data from the current project, runs it through an assembler, and opens the flashing software. If a cartridge flasher is connected via USB and a blank

cartridge is inserted, it will then flash the game onto a cartridge.

The Tool Bar

The toolbar contains quick access to commonly used functions in NESmaker. You'll notice that some of the interfaces within the software offer additional toolbars that directly pertain to that interface, but there is a set of icons that is present regardless of the currently exposed interface. Hovering over each icon will generate a tooltip if you need a reminder of what each icon does. Here is a brief description of what each icon does. These explanations will be further explored as we create our game.

New Project - creates a new NESmaker project.

Load Project - loads an existing NESmaker project.

Save Project - saves the current NESmaker project.

Save Project As - saves a copy of the current NESmaker project.

Export ASM to Alternate Folder - mostly a legacy function, this allows you to export the fundamental components for your game to a custom location outside of the project folder.

Backup Scripts - this allows a user to backup scripts as a new module

Project Info - mostly a legacy function, this set some initialization states from a GUI for earlier builds of the software. However, the triggered screen group tab is still used to allow a user to toggle screen triggers for testing.

Screen Groups - mostly a legacy function, as there are more advanced methods for grouping screens in the map editor.

Project Settings - the project settings dialog has a multitude of tabs. Inside these tabs range everything from setting up global game variables to assigning and working with module scripts to exporting a meta file with all game information.

Import Project Module - this allows a user to overwrite existing project module information with a new project module. It is for advanced use.

Export Project Module - this allows a user to export project module information for import into another project. It is for advanced use.

Run Project Module - this is part of importing a new module over the top of the settings for a project. It is for advanced use.

Export ASM - this exports the data from the current project without trying to assemble it or open it in an emulator.

Export and Test - this exports the data from the current project, runs it through an assembler, and opens the resulting ROM in an emulator.

Retest Last Export - this simply opens the last export in an emulator.

Make Cart - this exports the data from the current project, runs it through an

assembler, and opens the flashing software. If a cartridge flasher is connected via USB and a blank cartridge is inserted, it will then flash the game onto a cartridge.

The Hierarchy

You can expand nodes in the hierarchy by clicking on the small plus symbol next to an element of the tree, or collapse nodes in the hierarchy by clicking on the small minus symbol next to an element of the tree.

Expanding the project node will show you all of the elements that make up your game that are exposed to NESmaker's front end, as well as the tools you'll need to create elements. When you click on a terminal object in the hierarchy, it opens the tool or resource either in the general workspace or as a pop-up menu.

Here is a list of the icons for each node in the hierarchy. We will explore them in more detail in the process of making our game.

Overworld - opens the overworld map in the workspace area. This does not have to denote an overworld, but can instead be thought of as map1, a collection of 256 screens.

Underworld - opens the underworld map in the workspace area. This does not have to denote an underworld, but can instead be thought of as map2, a collection of 256 screens.

HUD & Boxes - this tool allows you to make use of some included functions for handling your hud and various boxes, such as dialog box area or gameplay area. Not all modules make use of all box types, but this tool can spit out data that can be used in many different ways even beyond the default use.

All Graphics - here is a repository for all of your background tile sets and tools related to dealing with graphic tiles.

Palettes - here is all of your palette information for background tile color schemes you can load to screens.

Game Objects - This node contains game objects, a special class of object that is designed to be permanently loaded for access on any screen. For instance, the player, power ups, and screen effects will likely be the same no matter the screen, so they would likely be game objects.

Monster Graphic Banks - this node contains monsters, which are game objects that are more malleable. They don't have to actually be antagonists of the game. You might find NPCs here, or any other object that appears with more variability throughout a game.

Monster Palettes - this node contains all of your palette information for object color schemes that you can load to screens.

Sound - this node contains music and sound effects.

Text - This node contains NPC text and game dialog.

Text Groups - this node contains a tool that allows you to batch your text in groups, which will be used as screen data to know what strings can be shown on a particular screen.

Scripts - this node contains two different types of scripts; any scripts that are assigned by the module, and input scripts that can be used with the input editor.

Input Editor - here you can set up input schemes and attach scripts to activate when certain controller settings occur, for instance triggering a movement script when a dpad button is being held down, or triggering creation of a bullet projectile when the b-button is pressed.

CHR Viewer - this node is a simple utility that allows you to view CHR files, which are the graphic files used by NES games.

Pixel Editor - this is a basic pixel editor that allows you to create, import, and work with graphics for your game.

Plugins - it is also possible to create your own plug-ins for NESmaker to help customize your experience. Plugins that are placed in the plugins folder will appear here.

Understanding The Module

Right now, our game does not contain any elements at all. It is effectively a blank game. You could hit the export and test button in the top toolbar, it would compile, and it would open in an emulator, but it would look blank. There are no graphics, there is no music, there is no control scheme. It seems that there is nothing at all here. That is a bit deceptive, though, as loading the module at the beginning of this process did the most important part - it tied dozens of scripts that will make this type of game work to our project. That's nothing we see, but it's all hooked up in the background, allowing us to focus on the creative part, tweaking the underlying engine where needed or desired.

But for a moment, let's take a look at those script tethers. You can find the scripts for this project by going to the Project Settings, which you can access by clicking on the small gear icon in the menu toolbar.

Click on the icon and navigate to the Script Settings tab. Don't worry, we're not going to concern ourselves with the scripts at this point, but it's important to be aware and understand what setting the module actual did. By choosing the maze game module, it linked the scripts that you see to the game. For instance, for this maze game module, it attached a top down physics engine as this game's physics script. If you instead choose a platformer module, it would link a physics script that also uses gravity. There are many small scripts that help define each game. You can assign alternate scripts to each of these. While in some places, choosing the right scripts can be daunting or confusing, in others it is very simple. For instance, it's very easy to assign a new script to an unused collision type, or an unused AI behavior. It's also very easy to iterate a script so that you can play around with dipping your toes in learning 6502 ASM and getting real time feedback on your changes. Even learning how to make very small modifications in the right scripts can lead to the ability for major customization. But fortunately, most of these modules come with enough to get you started without even thinking about the scripts.

Saving and testing our Game

Like with any other software, it's prudent to save often. Since this is a brand new project, you'll see that it is currently called "unsaved project" in the top left of the screen. To save it for the first time, go to File -> Save As. By default, it will save this NESmaker project file to the Projects folder inside of your primary NESmaker folder. Name your project something logical, like TutorialProject, and hit save. You should see that the name of the project in the top left corner of the NESmaker window has changed to reflect that it has been saved.

Even though we haven't created anything yet, we can now export and test our game. To do this, click on the first NES icon in the toolbar. This will run the current project through an assembler, create a ROM file, and open the ROM file in an emulator. If there were any errors in the code, the window that pops up would tell you the name of the code that had the problem, and the line of code where that problem can be found. If no errors were found, it will let you know that the game was successfully written and prompt you to press any key to continue. When you do, it will open our blank game in the MESEN emulator. You should see a blank screen. If you do, that means we are ready to start creating our game.

If a bug was found in your code that caused the assembly to fail, you may get a NESmaker messages saying that it could not find your ROM file. This is expected, because if there was an error, assembling the game could not be completed, meaning there is no file for the emulator to open. The NESmaker warning that pops up is a simple warning to let you know that this occurred, which prevented the emulator from opening a non-existent file.

Creating the Game States

Creating the Game States

Sometimes looking at an empty canvas can be daunting. Where do we even start? Do we start with trying to make some background graphics? Do we start with trying to get controls working with a hero object? It can be a bit nebulous.

This is where that simple plan we made at the beginning really becomes beneficial. Before I worry about any functionality or graphics or sound (because all of these things may be different from game state to game state), I like to start with creating a skeleton of the game flow.

For the game we'll be creating here, we've defined three distinct game states: The Start Screen, the Main Game, and the Win screen. Each of these have controls that behave a bit differently. For instance, pressing the start button on the start screen goes to the main game, but we don't want the start screen to jump to the main game if we're already in the main game. The b-button creates a projectile at the player's position in the main game, but we don't want to try to do that on the start screen. These game states have different rules on how they deal with drawing, with objects, with with input handling. Once you're feeling confident, you might want to create a death screen, cut scenes, inventory screens, a map screen, or any other type of unique, independent state the game could be in, and you'll want to account for these states first so that they're easy to populate later. For our game, we're just going to stick to our Start Screen, our Main Game, and our Win Screen.

Since our main game is going to be the main state that the game can be in, we'll make that the default; game state zero. This is not a requirement. You could just as easily make the Start Screen state zero and the Main Game state one with the justification that that's how they appear on screen, but for the purpose of this project, we'll use 0 for our default Main Game state. The Start Screen will be game state 1 and the win screen will be game state 2. To do this, we're going to

define some labels.

Working with Labels

NESmaker is really just a GUI representation of large swaths of hex data, that then is spit out and run through an assembler. The goal with the tool is to make the dozens if not hundreds of pages of number tables look like something tangible. And because every type of game is different, and in fact games within a type of game can be dramatically different, there are a lot of places in NESmaker that are drive by user defined labels.

For instance, for a platform game, tile type 2 might be a horizontal moving platform, where for a maze game, tile type 2 might be a door. A collision between the player and tile type 2 in the platformer would have one effect, where the collision between a player and tile type 2 in the maze game would have a completely different effect. This is driven by the scripts that we already looked at in the game settings.

But how do you, the user, know which tile is what for any given module? More important, when you start straying from the path and want to do something that isn't even included in the modules, how can you define it and expose it to the tool so you can remember "when I'm placing tile 2, it's this"?

Or in the current case - how can we visualize in the tool that game state 0 is Main Game, game state 1 is Start Screen, and game state 2 is Win? As I said, these things are not hard coded, they're completely variable to meet the broad needs of a broad array of games?

For this, we use labels. Right now, anywhere in the tool that invokes game states, game state 0 is simply called Game State 0. Anywhere in the tool that invokes game state 1 is simply called Game State 1. We're going to change the labels for these game states so that whenever we see them in the tool, rather than saying these generic names, it will show the specific name for them that we want.

Step 1: Open the Project Settings icon in the toolbar. The first tab is called Project labels. You'll see that last item in labels is called GameStates. Click on Game States and you'll see 8 possible Game States show up in a list.

Step 2: Click on GameState 0. With it selected, type Main Game in the Edit Value field. Click on GameState 1, and type in Start Screen. Click on GameState 2, and type in Win Screen.

These names do not need to be exact. They have no bearing on the code. As long as they are recognizable for you and demonstrative of what the labels will represent, that will be fine. Once you have the labels created, you can exit the Project Settings.

NESmaker Graphics Primer

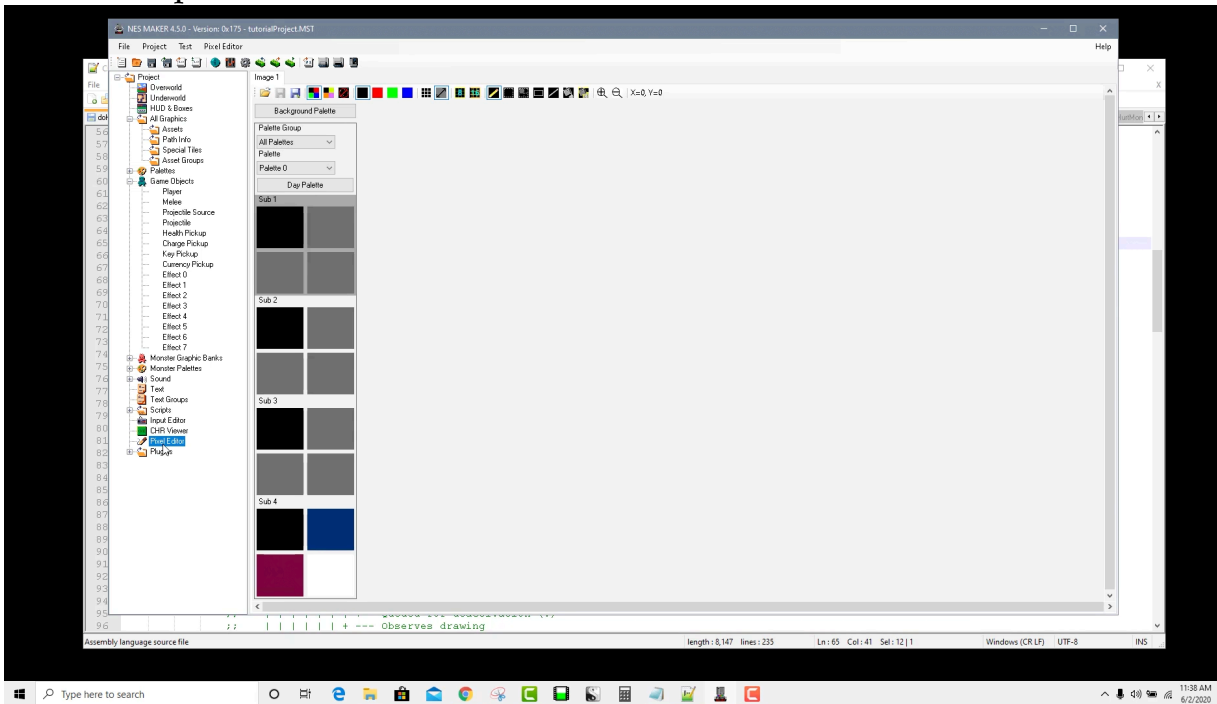
NESmaker Graphics Primer

The next thing to do is to start getting some visual feedback by working on the graphics for our game. We know that we'll need a deserted island tileset for our background graphics, we'll have an animated player, a monster, some power ups, some graphics for text, and potentially some designs for start and win screens. We can either create new graphics using the tools inside NESmaker itself, or we can import graphics from other graphics applications, optimizing them for the NES where necessary. To do this, we're going to use the Pixel Editor. But before we dive too far into the interface of the pixel editor, we should first examine at a very high level how graphics work with NESmaker and the NES in general.

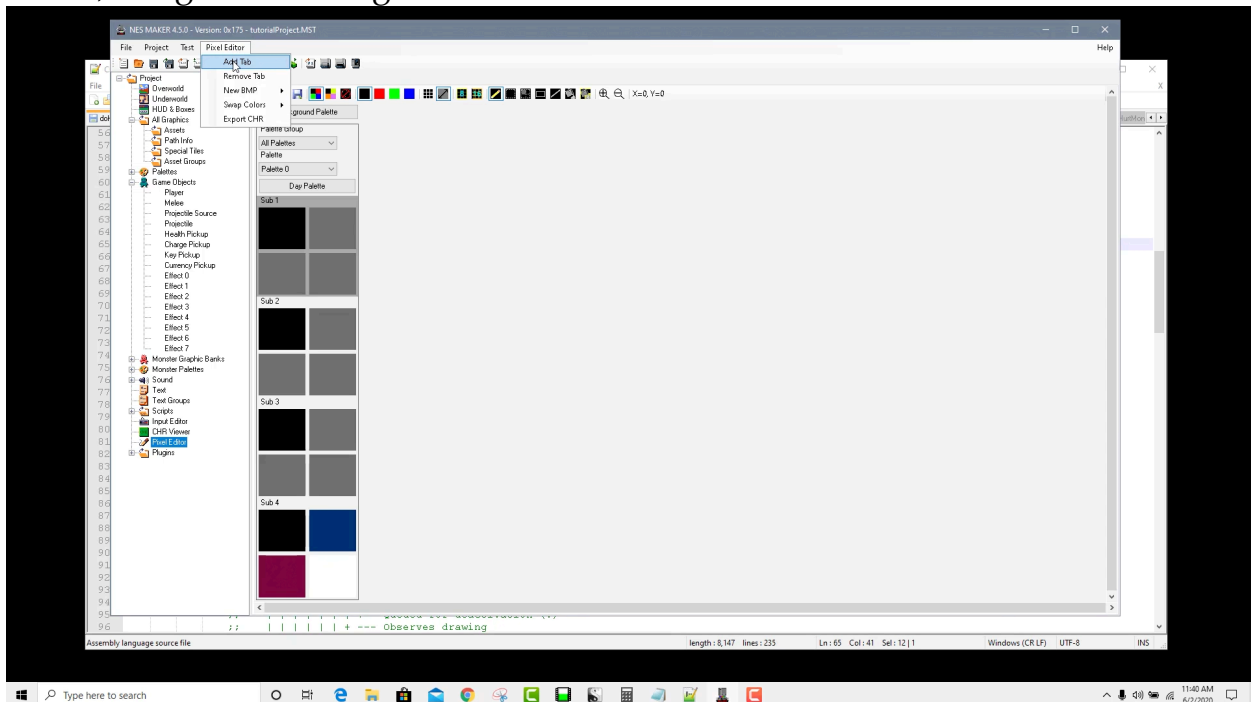
In order to get a feel for how NES graphics work, we're going to create a dummy canvas that won't actually be used for our game. This will just be for practice use to better understand how the NES deals with things like pixels and colors.

Step 1: Click on the pixel editor in the hierarchy to bring up that tool in the

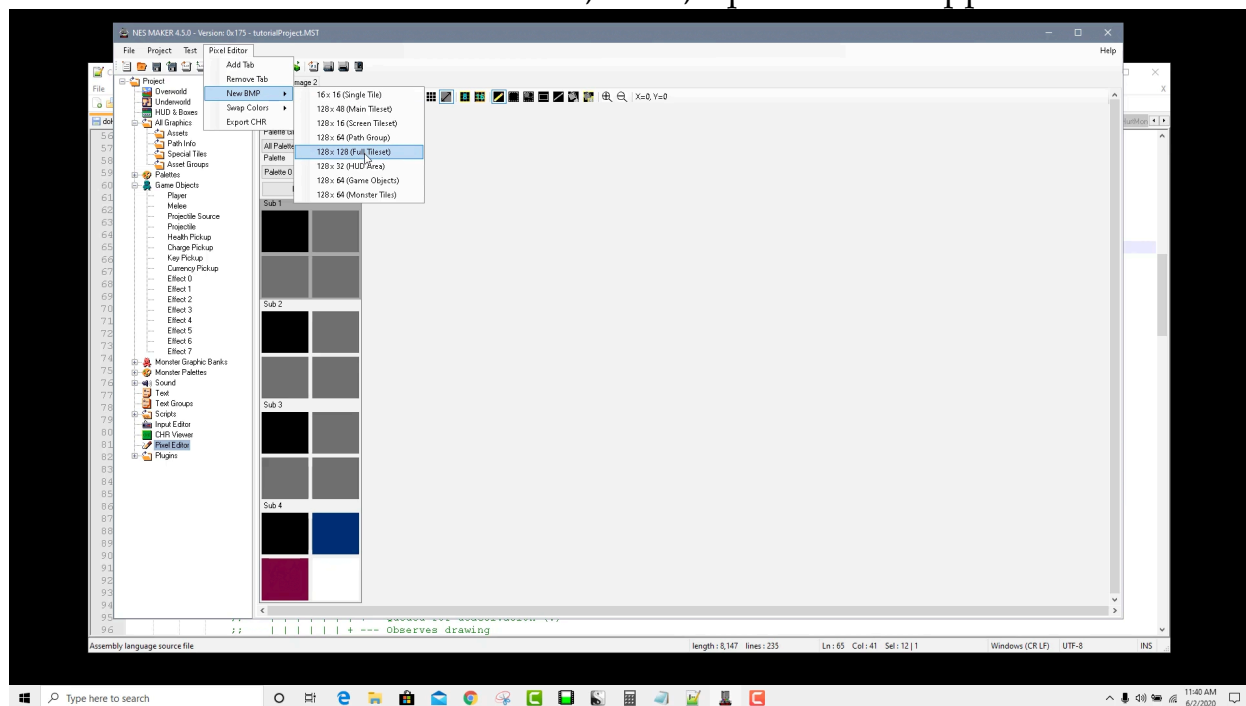
general workspace area.



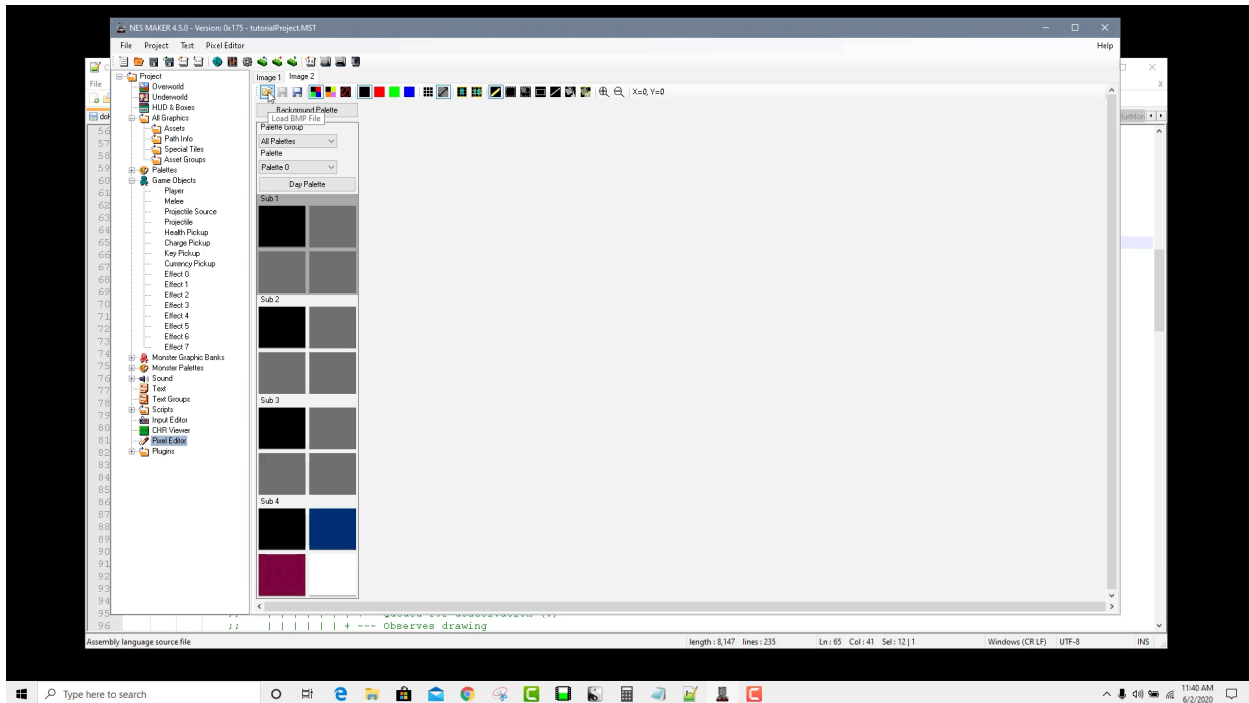
Step 2: You'll see that now that we're in the pixel editor, a Pixel Editor menu has appeared in the menu bar at the top of the screen. Go to the pixel editor menu and select Add Tab. You'll notice you now have two empty tabs in your editor; Image 1 and Image 2.



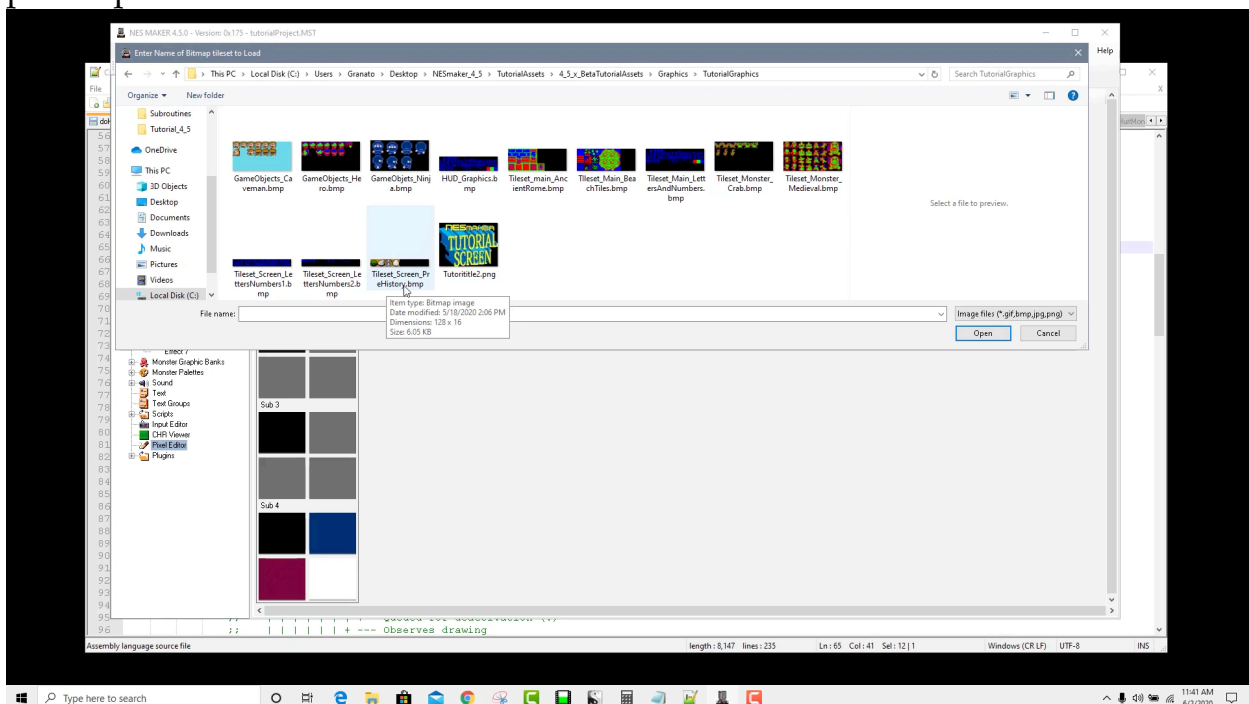
Step 3: While in the Image 1 tab, go to the Pixel Editor menu, select New BMP -> Full Tileset. You should see a blank, black, square canvas appear.



Step 4: Click on the Image 2 tab. Click on Load BMP file, which is the small open icon in the top left of the Pixel Editor specific toolbar. In the explorer window that opens, you see the default graphics for your current project. Look at the file location, and you'll notice that you're inside NESmaker, inside GraphicsAssets, and inside the graphics folder we created at the beginning of the project. This is the default, templated graphic structure for NESmaker games. You'll see reserved tile sets for background tilesets, monster tilesets, Hud tilesets, and more. Part of loading a module was code that loads these specific tilesets in particular places in memory, making making a game's graphics easy to work with. They're blank right now, but as you begin to draw to them or past into them and save them, the NESmaker tool and the game that results will automatically be populated with those corresponding graphics. We'll look a bit more at that in the next step.

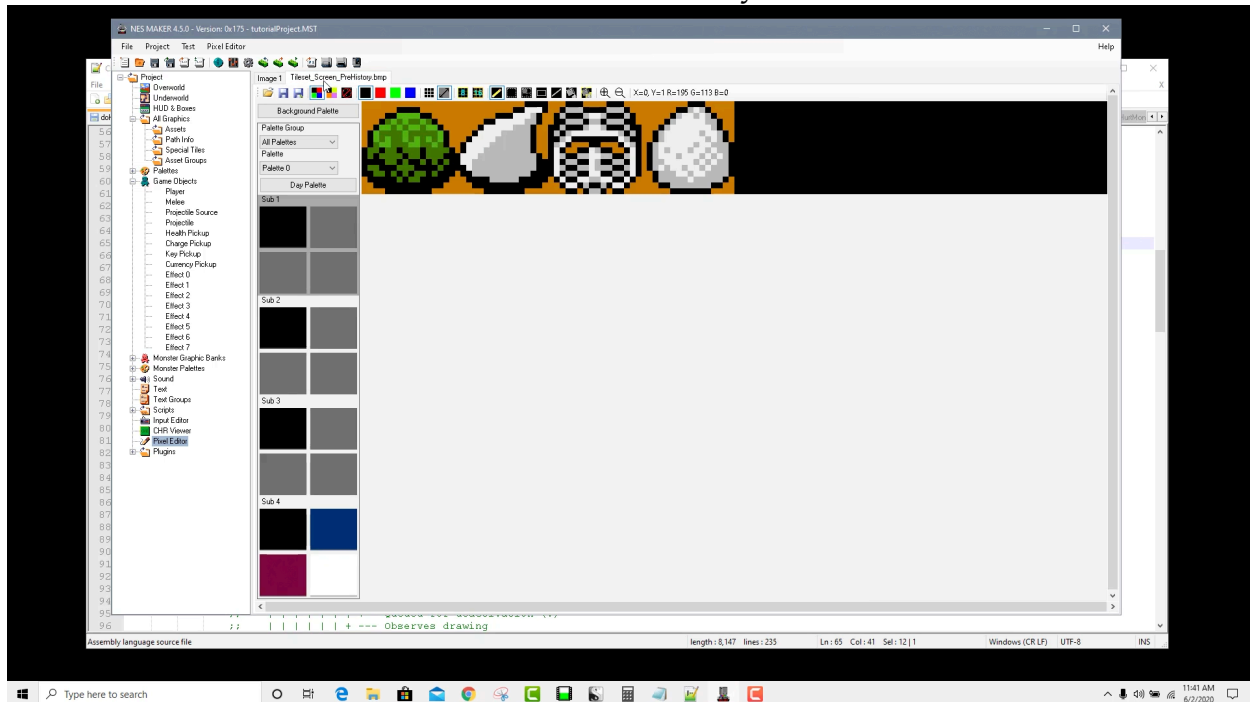


Step 5: Click on the root NESmaker folder in the explorer, scroll down to Tutorial Assets, open the Beta Tutorial Assets, open Graphics, open Tutorial Graphics, and we're just going to pull in the tileset called Tileset_Screen_PreHistory.bmp. Double click on this file, or click on the file and press open.



Now we have two tabs with content. A full, blank tileset in the Image 1 tab,

and a tab that is called `Tileset_Screen_PreHistory` in the second tab.



The PreHistory tileset is actually showing graphics that were created in an external tool, like Photoshop, Gimp, or even a simple graphics program such as Paint. The problem is, these graphics are not conformed to NES specifications. It's just a standard bmp file, which is very different from the type of data that the NES's graphics engine needs.

In order to get these graphics to work with the NES, we need to get them to a four-value format. The NES does not work with colors in the way that we think of them in a modern graphics tool. It utilizes CHR files, which are strings of data that effectively give the hardware instructions as to which currently loaded palette information to make each pixel. Instead of thinking about it like a finished image, think about it like a paint by number. Every pixel is assigned a zero, a one, a two, or a three. Meanwhile, separate from that, four markers are selected and placed in slots 0-3. The NES then colors in each pixel with the marker that was selected for the correlating spot. The graphics file just gives the paint-by-number values for the pixels, so the graphics file doesn't actually carry color

information. It works with the selected proverbial markers in the palette to determine the color to draw everything.

An easy way to understand this might be to consider Mario and Luigi. In a modern game engine, you would likely have separate graphics for Mario and Luigi, even if the only discrepancy was color. On the NES, the actual CHR graphics file is exactly the same for Mario and Luigi. What changes is different colors are loaded into the color slots depending on whether the game is in the player 1 or the player 2 state. So when the player.1 state is loaded, the game knows to grab the Mario colored markers to paint the character with. When the player 2 state is loaded, the game knows to grab the Luigi colored markers to paint the character with. But the actual graphics data is the same either way.

All of this leads to having to think about graphics a bit abstractly. Whatever is painted to a NESmaker tool's pixel editor will only look that way if the screen that is drawing it has the same palettes loaded. Really, you're not painting with color at all. You're painting with value 0, value 1, value 2, and value 3.

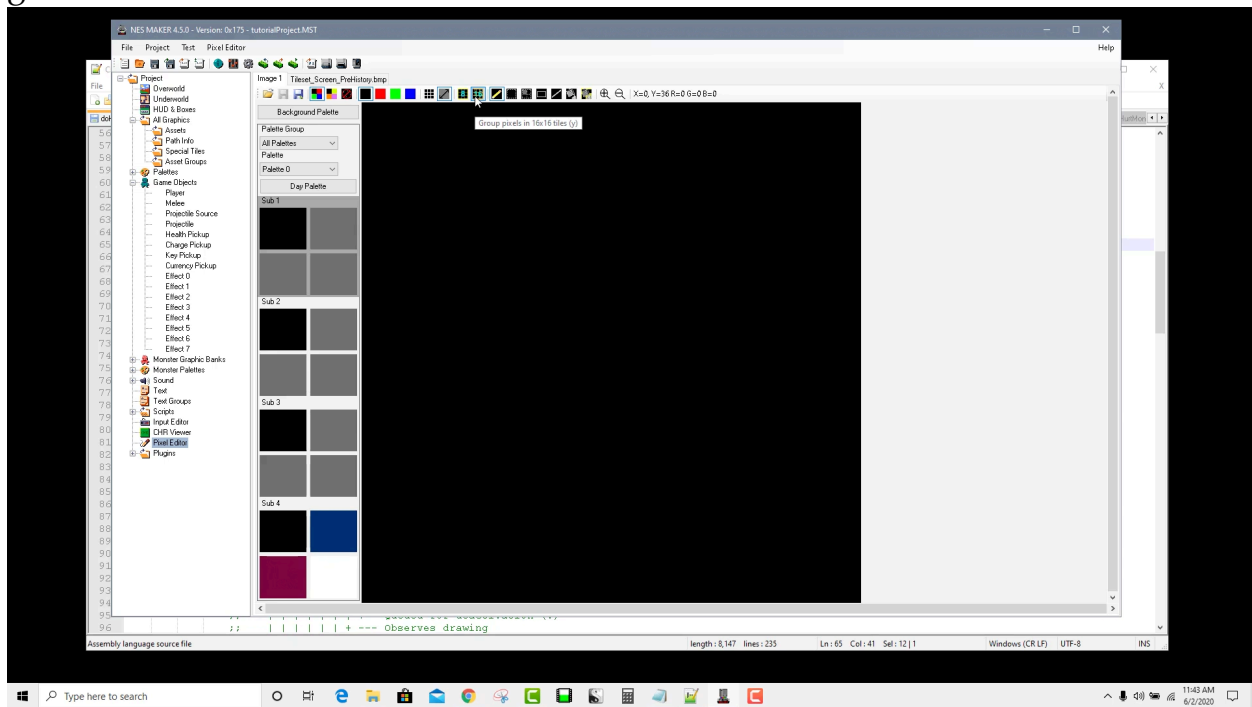
How NESmaker handles this is to truly be creating RGB+A bitmap files that are converted to CHR values when the game is compiled. The term RGB+A means that the graphics are Red-Green-Blue Plus Alpha. Four values. This gives a logical and consistent way to design graphics using only four values. The pixel editor then has the ability to push those values through a palette and view the graphics as they would appear if that palette was loaded to the screen. Again, imagine making the Super Mario Brothers hero using Red, Green, Blue, and Alpha pixels. It wouldn't look like Mario or Luigi. It would be a graphic that was in the exact shape of both, but the colors would look strange. Now imagine seeing it through a Mario colored palette, where red (value 0) is red, green (value 1) is skin tone, and blue (value 3) is white, versus a Luigi colored palette where red (value 0) is green, green (value 1) is skin tone, and blue (value 2) is white. Depending on which palette I loaded, the same graphic data would either look like Mario or Luigi. This more or less illustrates how NES graphics work in

terms of pixels and palettes, and how NESmaker works with graphics at a high level to conform to those constraints.

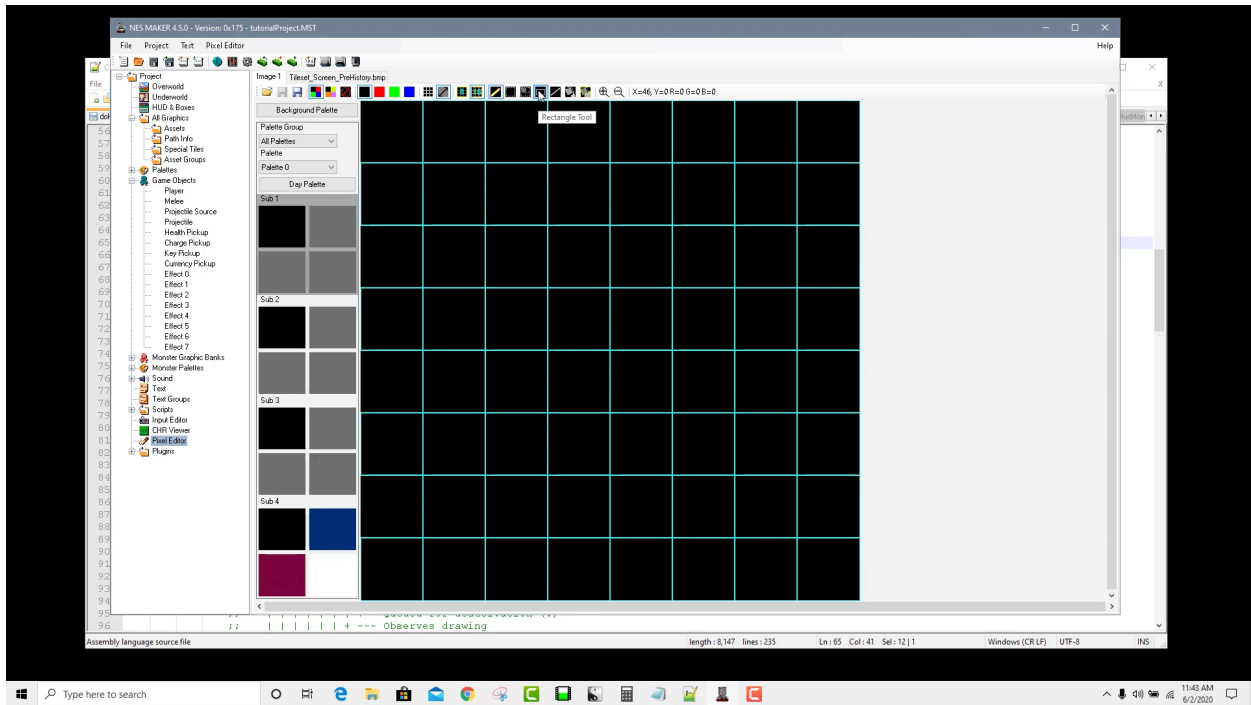
To better understand, we can do some basic exercises in our dummy tileset in the Image 1 tab.

Step 6: Click on the Image 1 tab.

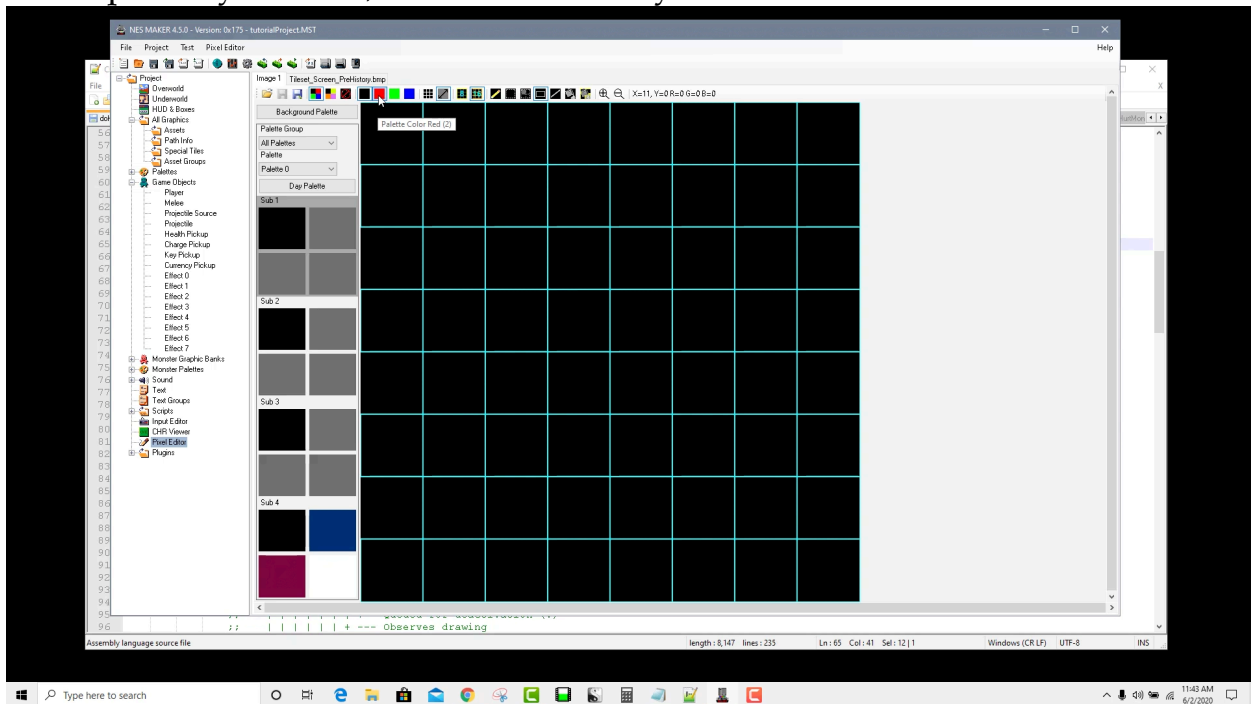
Step 7: Using the menubar at the top of the pixel editor, turn on the 16x16px grid



Step 8: Select the RECTANGLE tool.

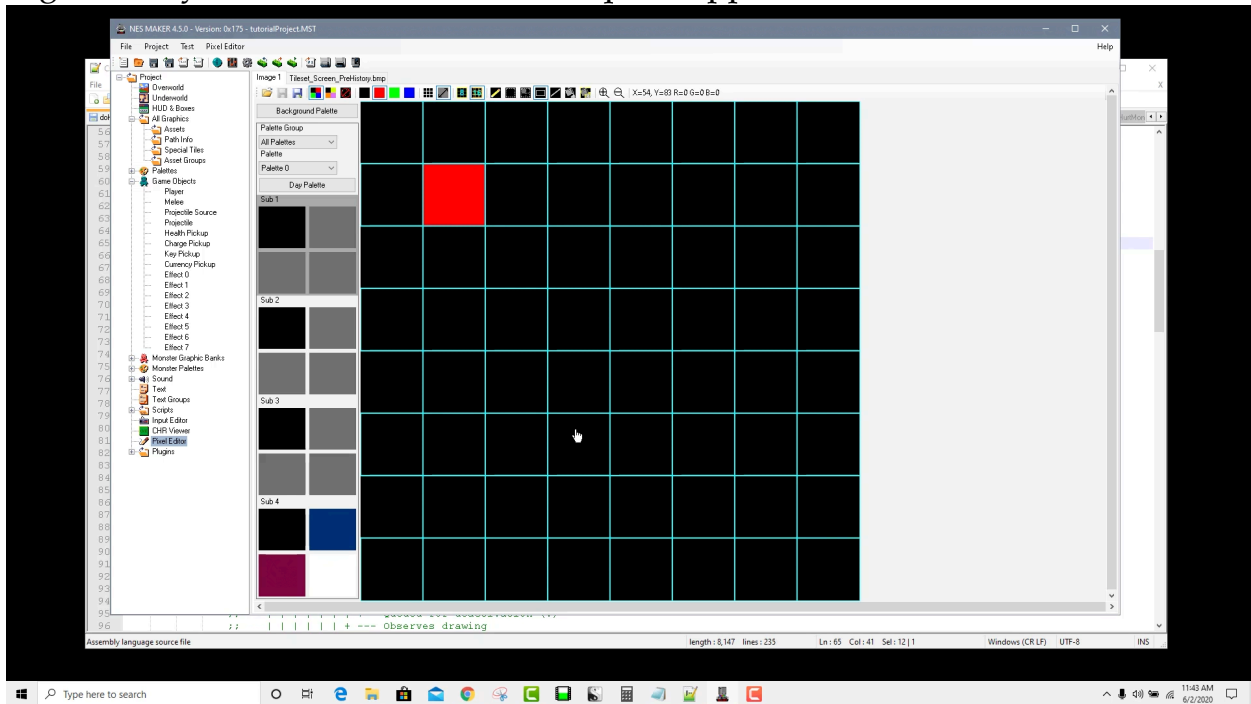


Step 9: Choose the color RED in the top pixel editor menubar. Remember in advance, functionally, this is not drawing the color red. It is drawing the color 1 of our paint-by-number, whatever that may be.

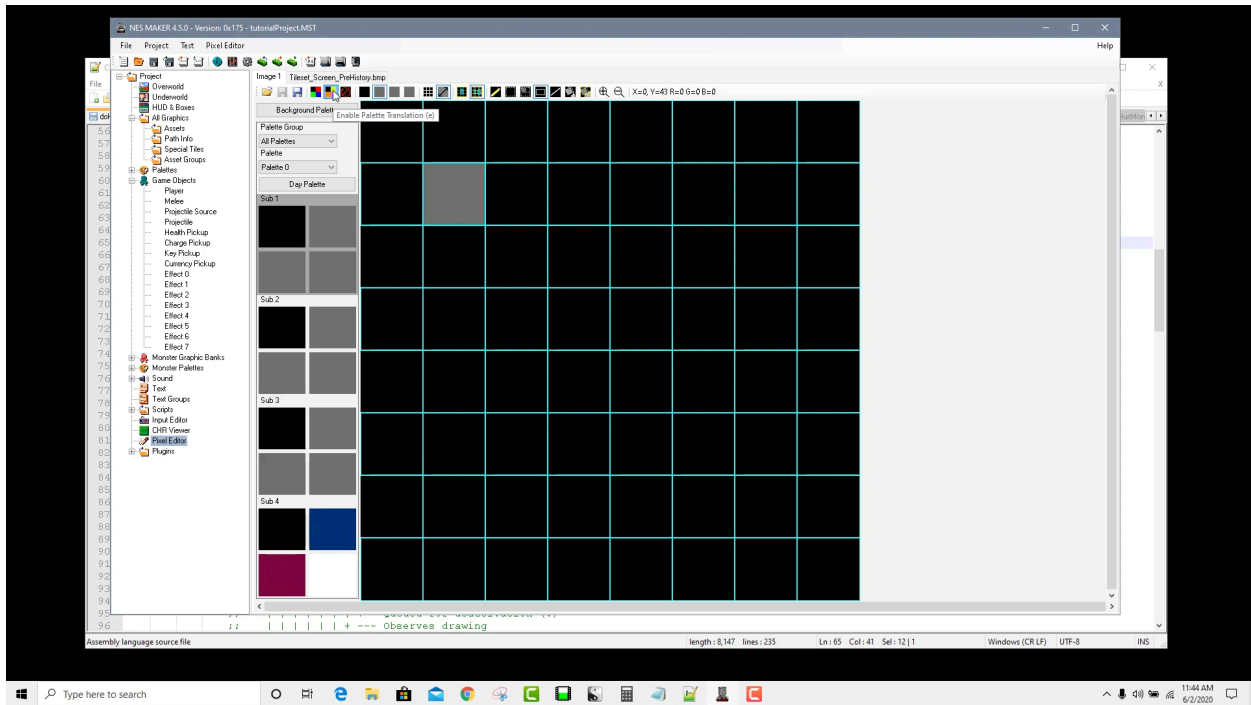


Step 10: Draw a rectangle by dragging across one grid square on the canvas.

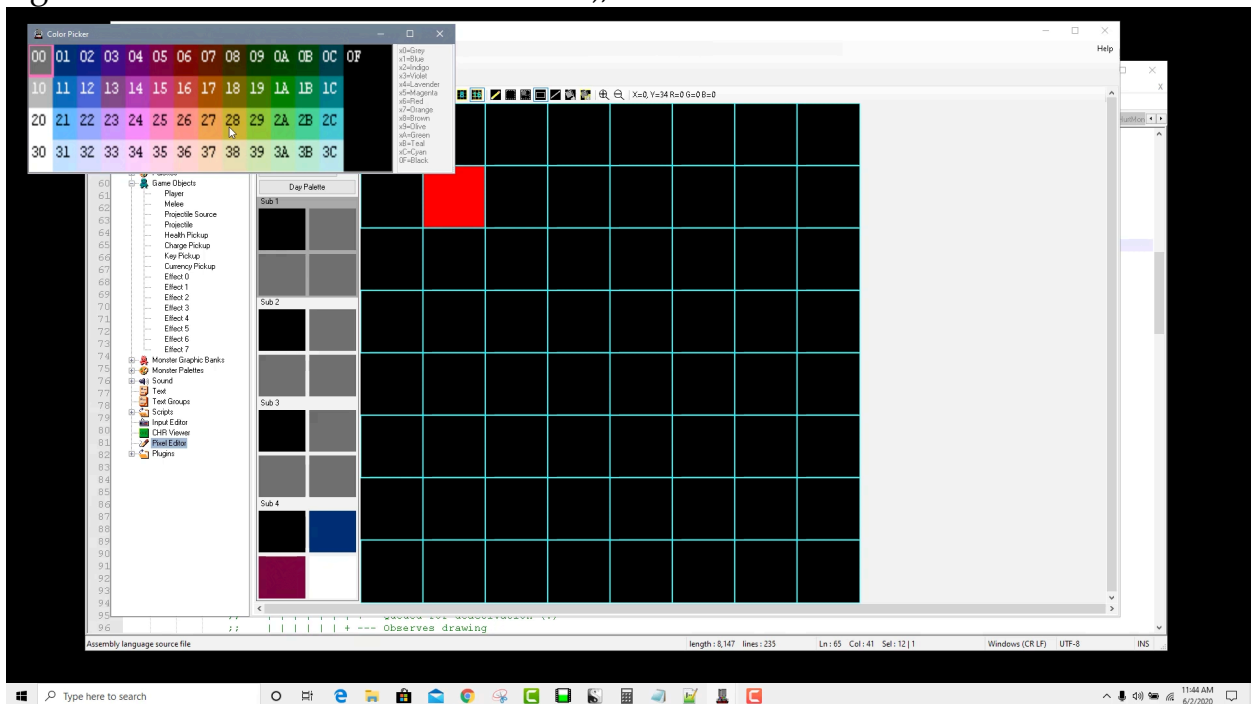
Right now, you will, in fact, see a red square appear.



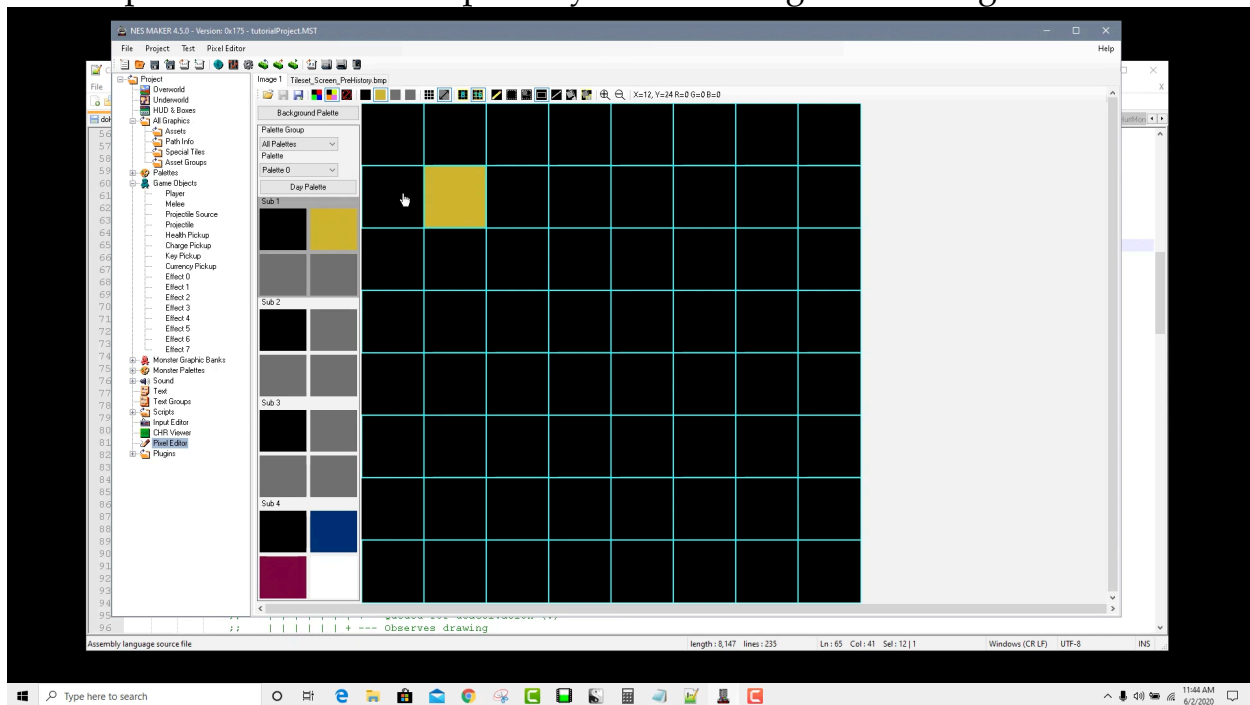
Step 11: In the menubar, click on the Enable Palette Translation. What this button will do is stop showing the RGB+A files that are demonstrative of those paint-by-number values, and instead show what this graphic would look like through the currently loaded palette. If you look to the palettes on the left of the pixel editor, you'll see black in the value 0 slots for each 4 color sub palette, but gray for every other slot. Since this is value 1, and in the selected four color sub palette, value one is gray, clicking on enable palette translation will turn our square gray.



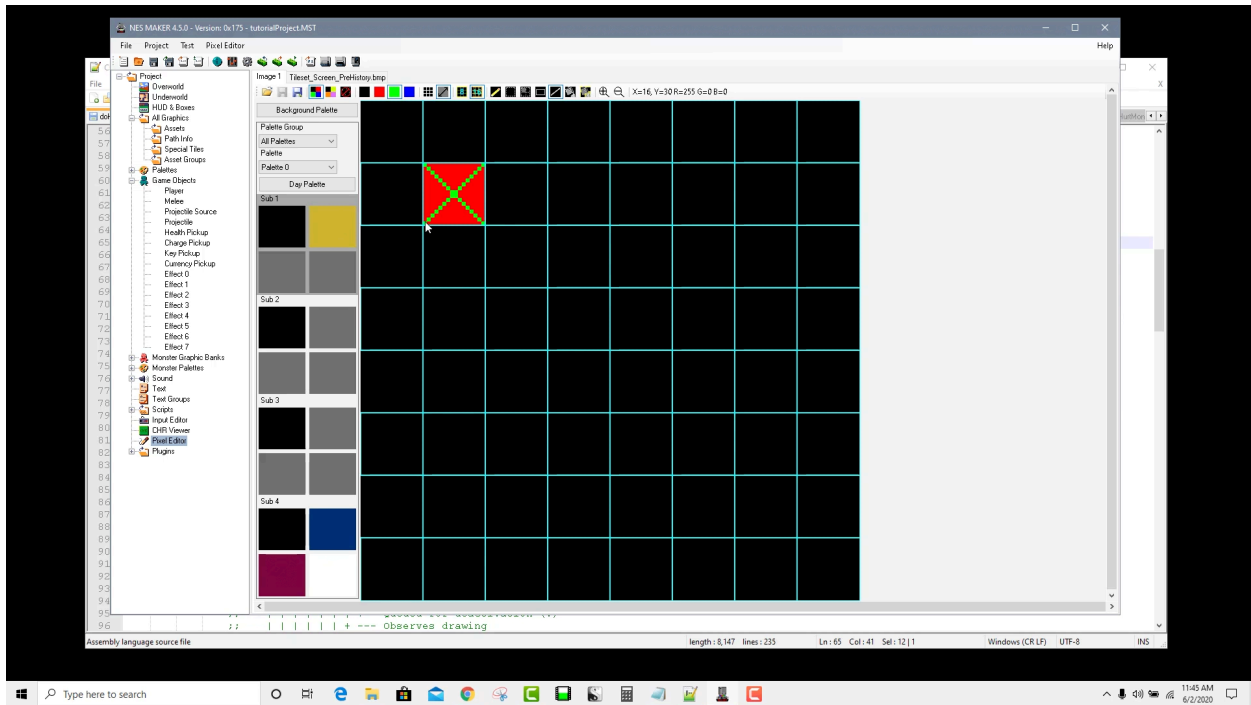
Step 12: Go back up to the menubar at the top of the pixel editor and change back to disable palette translation and the square will turn red again to denote it is showing color value 1. Now, go to the palette, click on color number 1 (top right corner of the four color swatches), and choose a different color.



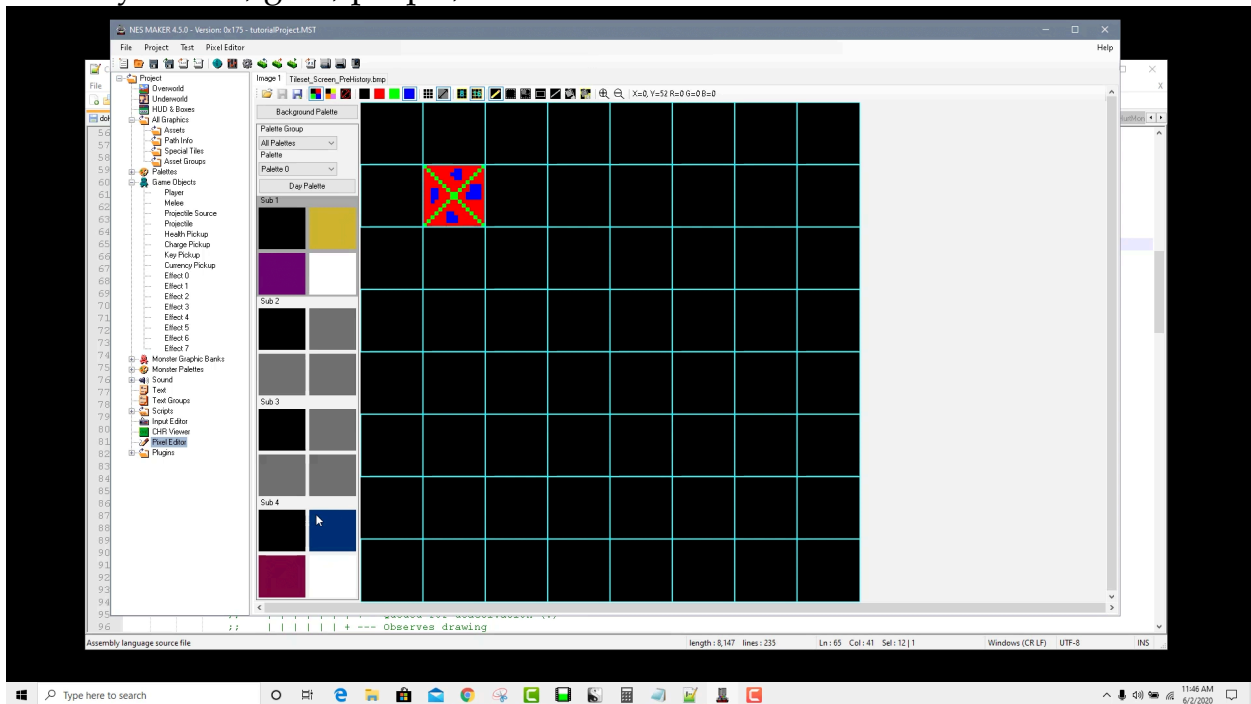
Step 13: Now, enabling palette translation mode again will show what this graphic would look like if the currently loaded palette had gold in its first slot. You can see that the graphic is gold. We have grabbed a gold marker and colored all the spots marked 1 on our paint by number image with that gold marker.



Step 14: Change back to disable palette translation so that you see things in RGB-A mode. Choose the line tool from the menubar at the top of the pixel editor. Choose value 2, "green", from the color swatches in the menu bar, and draw a green x across the box. This doesn't have to be perfect, just as long as you see red and green on the canvas.

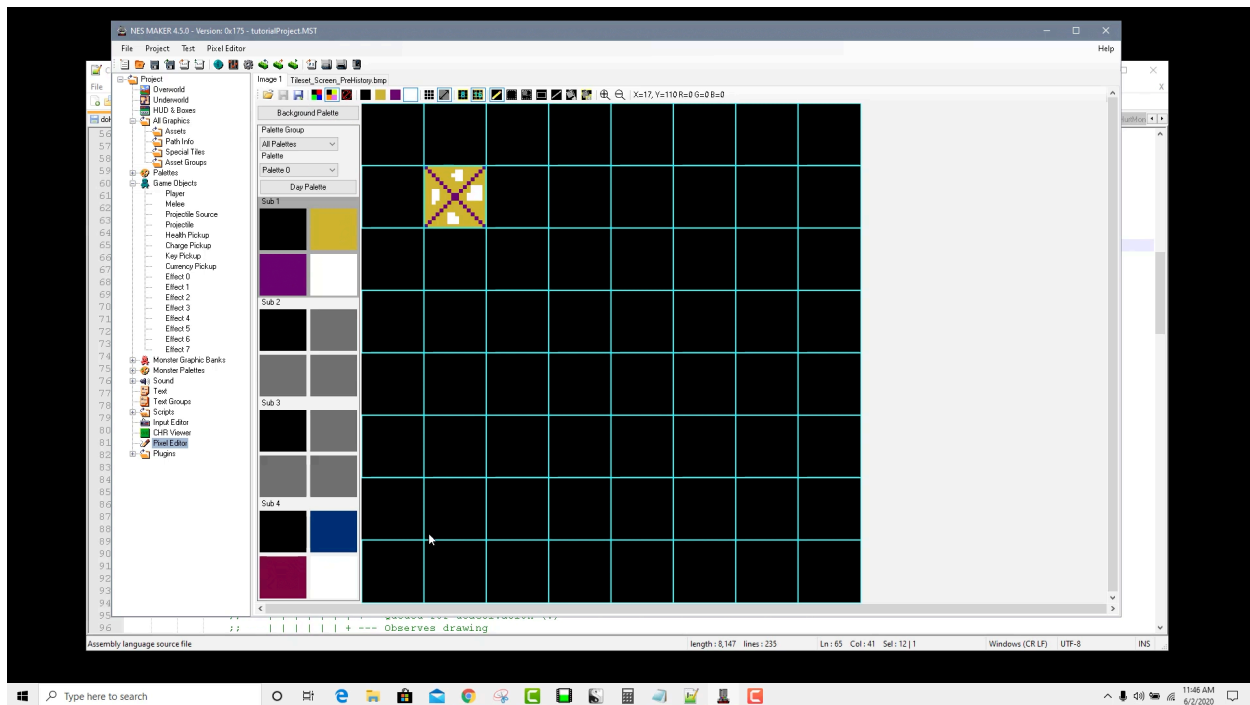


Step 15: Choose the pencil tool from the menubar, choose value 3, “blue”, from the color swatches. Color a few blue dots. In your palette, set the colors up this way - black, gold, purple, and white.



Step 16: Now, if you enable palette translation, color 0 (alpha) will be black,

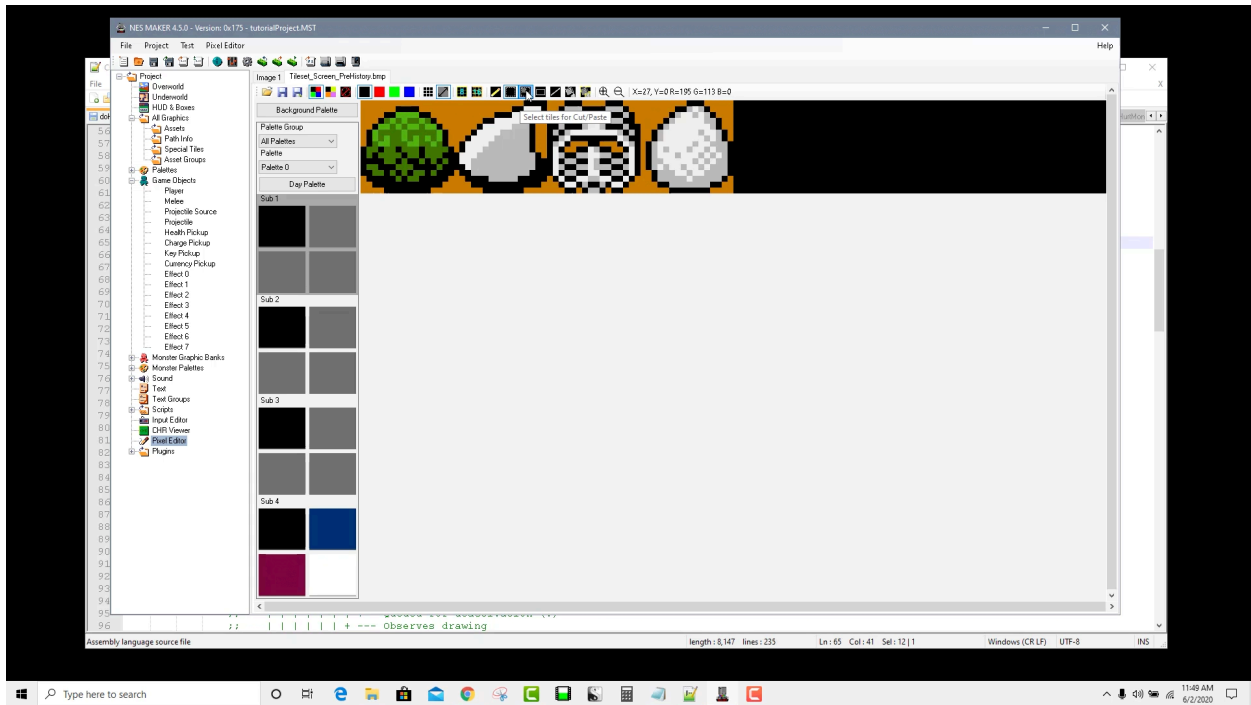
color 1 (red) will be gold, color 2 (green) will be purple, and color 3 (blue) will be white. What you're seeing now is what our funny little square would look like in game if this palette was loaded.



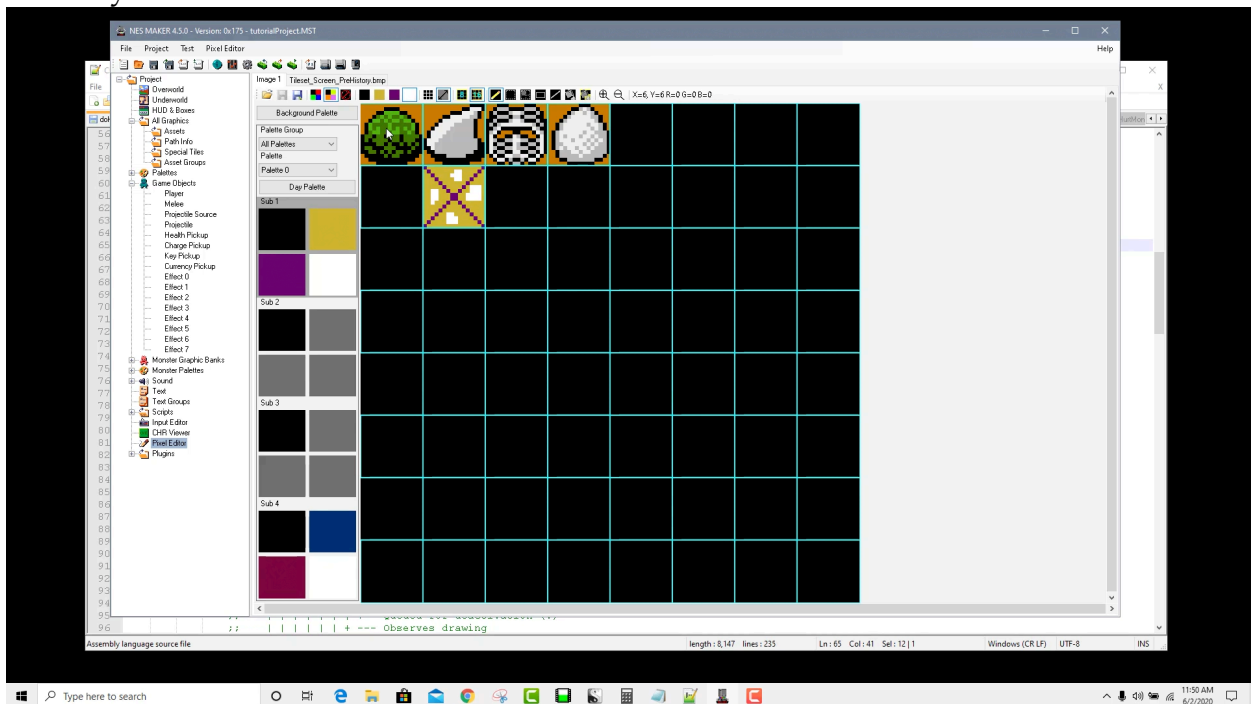
It is also important to understand that this graphic will only look this way if these colors are loaded during gameplay. If a different palette is loaded to those slots when the game is being played, this same graphic will appear a different color. We'll talk a bit more about that when we are designing screens and objects.

So back to second tabs with the bmp graphics on the screen. These graphics are not in RGB-A mode. This is a full color bitmap image, and can not work as such on the NES despite looking simple enough to exist as NES graphics. In these next steps, we'll learn how to copy these graphics into a tileset and conform them to those NES constraints.

Step 17: Navigate to the second tab. Choose the Select Tiles tool from the menubar, turn on the 16px grid lines, and click and drag over all four graphics on screen. Press Control-C to copy.



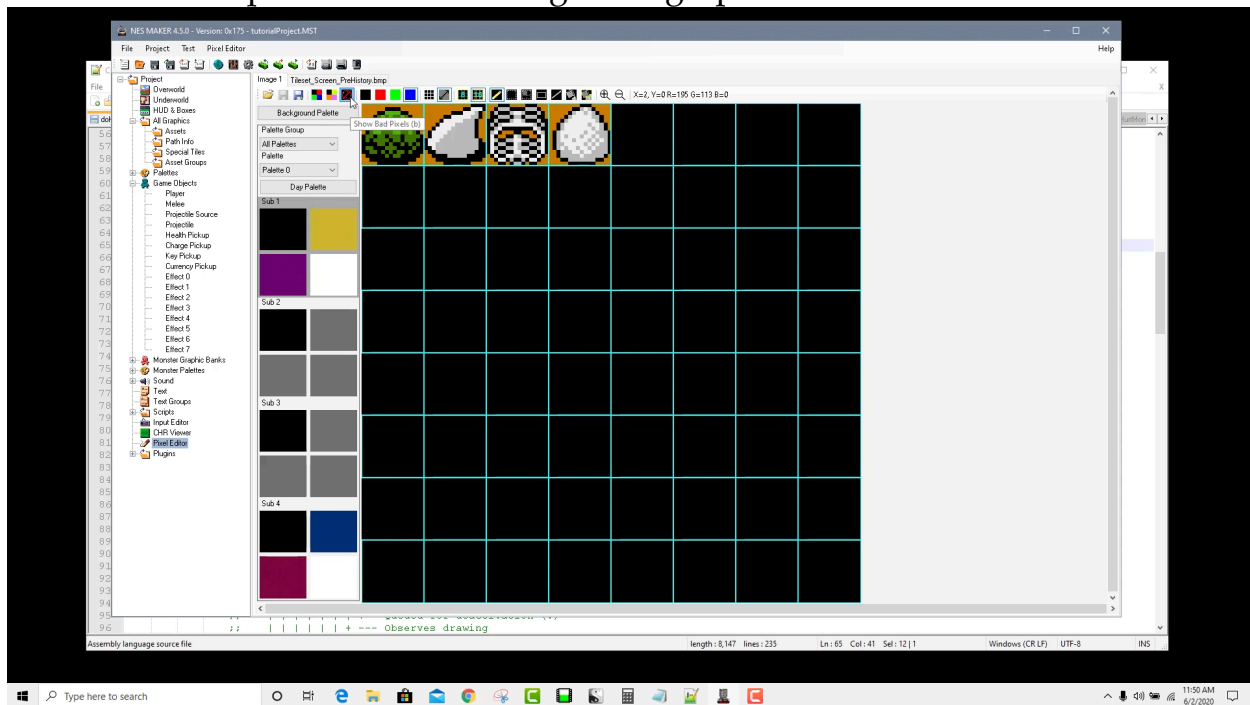
Step 18: Return to the Image 1 tab, move your mouse to a tile where there are four consecutive free squares, and hit control-shift-V. Holding shift while you paste will clamp what you are pasting to the top left corner of the grid space where your mouse is.



Right now, if we were to use this as a tileset on the NES, those newly placed

graphics would not show up at all. When this is converted into a CHR file, since those values are not strict RGB+A, which is how NESmaker determines the correct pixel value. If you click on disable palette translation, you'll see the square we drew turn to RGB, but the newly pasted graphics remain the same. That's a good sign that they won't be recognized.

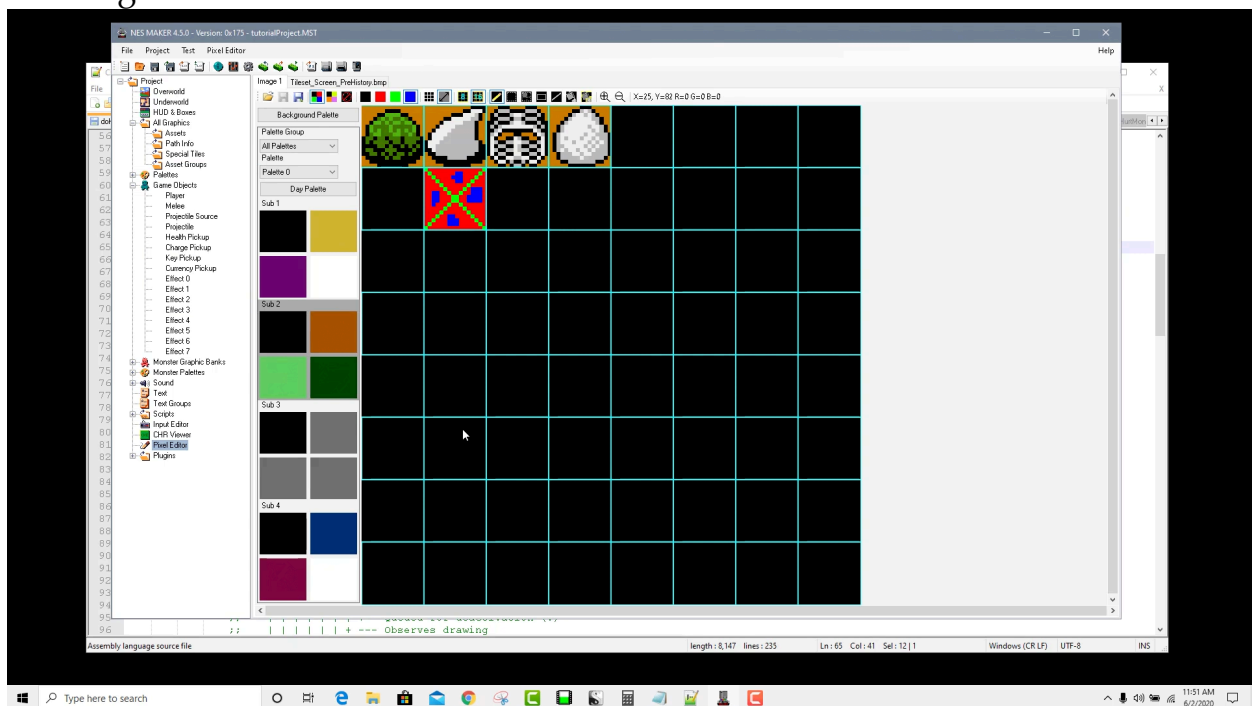
Step 19: Another way to tell what pixels are in the right format and which ones aren't is to press the show bad pixels icon in the pixel editor menubar. This will hide all of the good pixels, leaving only the bad ones. Pressing this causes our drawn graphic to disappear, but the pasted graphics remain. This means that they will all be seen as null on export, and thus all use the zero, or alpha color. So the next few steps will be converting these graphics in to RGB+A.



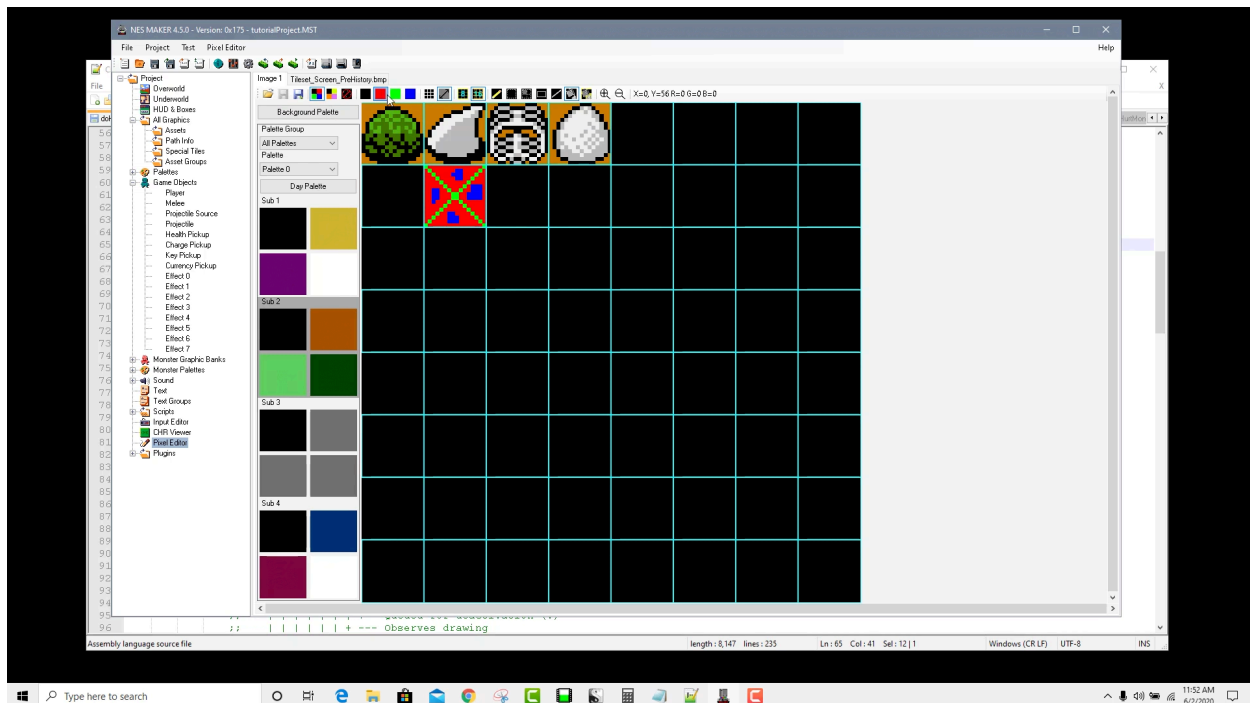
Step 20: In order to conform the graphics, a little creative imagination must be used in order to determine what color should end up utilizing what value number. A good practice might be that red (color 1) is your shadows, green (color 2) is your main color, and blue (color 3) is your highlights. Or red could indicate your main color, and the other two could be used respectively as accent colors. The important thing is consistency, which will make working with palettes easier.

For this graphics sheet, all four of those elements have a ground color, a highlight color, and a shadow color. Even though the colors are different from element to element (the bush has greens while the others are white and gray). That's ok, we'll handle that with our palettes.

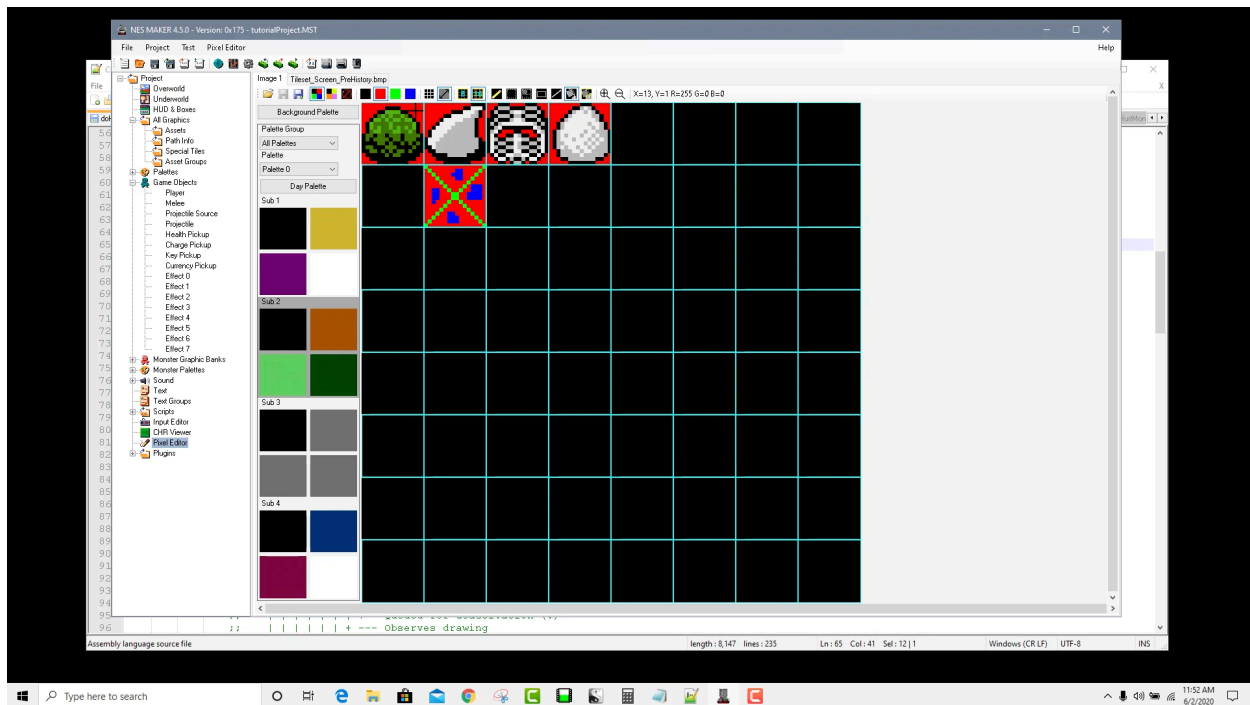
Let's use the second sub-palette (group of four colors) to try to emulate the bush. We'll use our first color as our ground color, so we'll make it an oranges color. Then we'll use our second color as our highlight color, so we'll make it light green. Then we'll use our third color as our shadow color, so we'll make it a darker green.



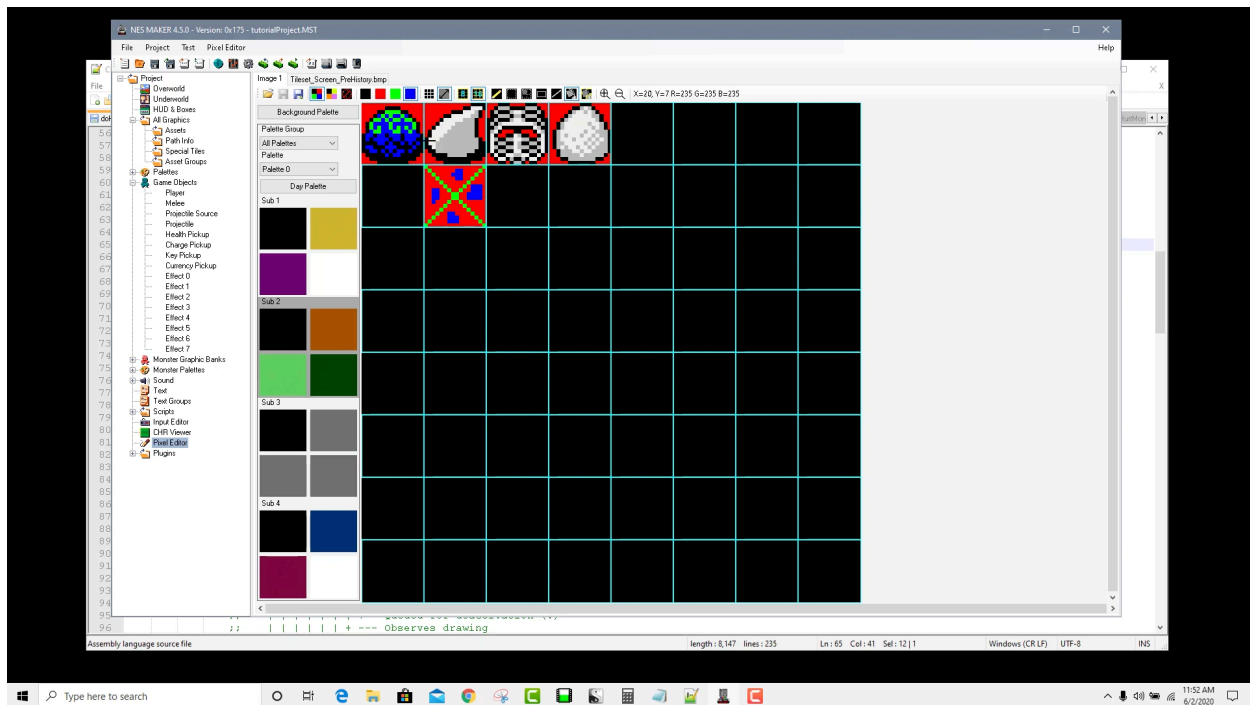
Step 21: Now, just to see the effect, click on disable palette translation if it's not already disabled so that we're seeing the RGB-A values. Choose the tool Global Color Change, which as its name suggests will change all of a given color to a new value across an entire tileset.



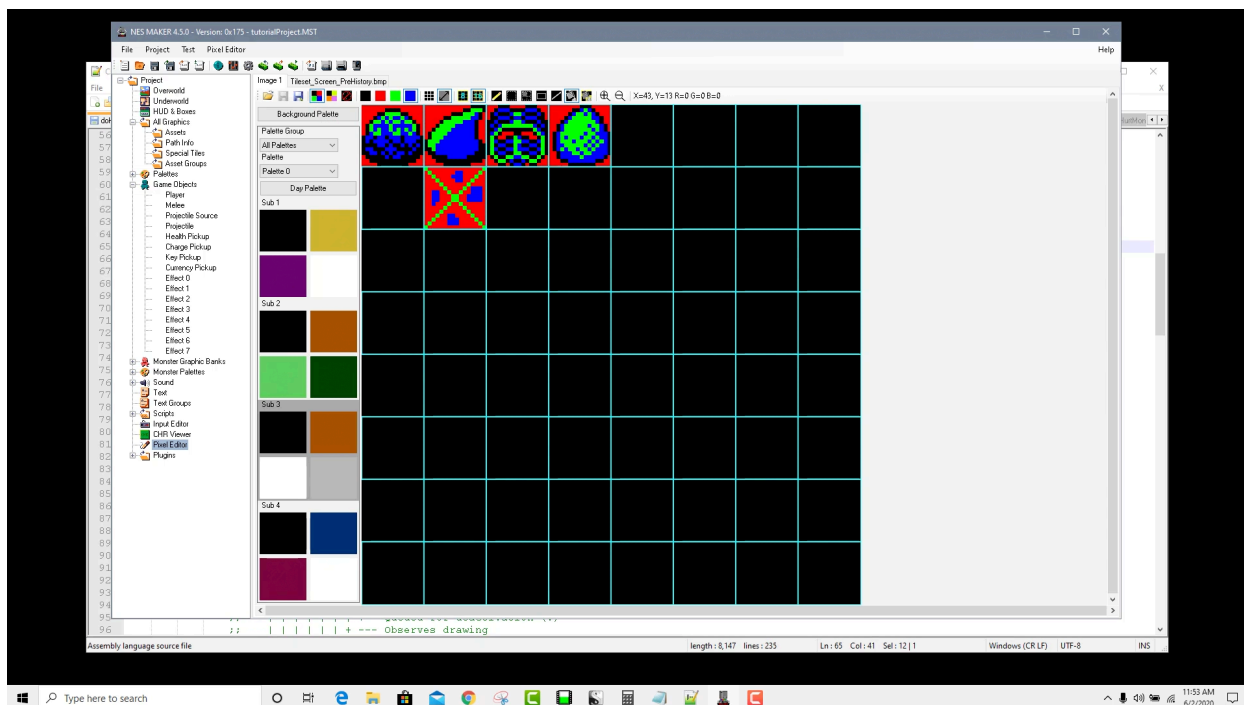
Step 22: Select either the orange color from the palettes on the left, or the red color from the swatches above. Remember, these will tell the file the same information. They're both just saying "fill these pixels with value 1". Then, with the Global Color Change tool and the first color swatch selected, click on the ground area of the pasted tiles. If you have disable palette translation clicked, you'll see that all of the ground turns red (color value 1). If you enable color translation and make sure that your sub palette 2 is highlighted on the left of the pixel editor, you'll see that the ground turns to the orange color we chose for the color 1 of this palette. If you were to click on the first sub palette, you'd see the ground area would turn gold, because that is the color that is in that sub-palettes slot number 1.



Step 23: Repeat the process for the light green color (or color swatch 2) and the dark green color (or color swatch 3). The bush is now completely conformed. Disabling palette translation will show this as a RGB-A tile. If you click on the show bad pixels icon in the menubar, you'll notice that the entire bush disappears, letting you know that the bush is now conformed. But we still have to deal with the white and gray of the rest of the tiles.



Step 24: Change the colors for sub palette 3. Just like with sub palette 2, change the ground color to the same orange. But change color 2 (green) to a white for highlights, and the color 3 (blue) to a gray for shadows. Use the Global Color Change tool and the color 2 swatch to paint the highlight areas and the color 3 swatch to paint the shadow areas. If you disable palette translation, you will see that now the entire tileset is conformed to RGB-A. If you click show bad pixels, everything will disappear, letting you know that there are no bad pixels. Now, this graphic sheet is conformed, and if exported, will create the proper CHR files that can appear in game. Their actual color will depend on what is in the corresponding palette slots when they are loaded.



Hopefully this helps a bit in understanding the relationship between pixel values and palette colors, and how NESmaker can conform graphics to work with the NES constraints. To make it even easier, if you design graphics in NESmaker's pixel editor, they will always be properly conformed.

The Pixel Editor

The Pixel Editor

The pixel editor in NESmaker is designed for creating, importing, and editing simple pixel art in a way that conforms to the needs of the NES hardware. When you open the pixel editor, you'll see that a new Pixel Editor menu appears at the top of the screen, and that a new menubar with has appeared that includes tools specifically for creating and editing pixel art.

Pixel Editor Menu

- Add Tab - this adds a tab to the workspace. You can have multiple

- tabs open simultaneously and work back and forth between them.
- Remove Tab - this will remove the currently selected tab from the workspace.
 - New BMP - This creates a new blank BMP in the canvas. Various common sizes for NESmaker use are there with descriptive names for what you may want to be working on.
 - Swap Colors - occasionally, you want to swap the colors of a current BMP. With this, you can trade any value with any other value. So all pixels that are using value 1 can be swapped with all pixels using value 2 on a canvas.
 - Export CHR - in the event that you need the direct CHR from what you're building in the pixel editor, you can export it using this menu item.

Pixel Editor Menubar

- Load BMP file - this opens a file explorer. You can load BMPs, JPGs and PNG files to the current tab.
- Save BMP - this saves changes to the currently loaded file.
- Save BMP As - this allows a user to save a copy of the currently loaded file.
- Disable Palette Translation - This shows the current graphic by value, using an RGB-A representation where Alpha = value 0, Red = value 1, Green = value 2, and Blue = value 3.
- Enable Palette Translation - this shows the current graphics with the colors of the currently loaded palette applied.
- Show Bad Pixels - Clicking on this will show any pixel that is not conforming to the NES needs, meaning that it exists outside of the RGB-A in the raw file.
- Palette Color Swatches - these show either the RGB-A or Palette translated colors, depending on which mode you are in. You can click on these to paint with that particular color.
- Show Grid - this shows or hides a pixel grid.

- Force Square Aspect Ratio - sometimes when navigating the different NESmaker menus or dealing with bmp that are strange sizes, the canvas can become stretched. The Force Square Ratio fixes that.
- Show Border With Actual Colors - when engaged, all of the swatches in the sub palettes will show an outline denoting their true RGB-A value. This is helpful to see the RGB-A reference when using Enable Palette Translation.
- Group Pixels in 8x8 Tiles - this turns on a grid and also makes selectable areas in divisions of 8 pixels.
- Group pixels in 16x16 Tiles - this turns on a grid and also makes selectable areas in divisions of 16 pixels
- Pencil Tool - good for free drawing or dot-by-dot drawing.
- Select a Region - allows you to select a region of arbitrary size
- Select Tiles - selects entire tiles at a time.
- Rectangle Tool - allows you to draw a rectangle
- Line Tool - allows you to draw a straight line
- Global Color Change - allows you to change all pixels of a certain color that are on the canvas to the selected color
- Tile Fill - allows you to change all pixels of a certain color that are in a given tile to the selected color
- Zoom in / out - allow for zooming in and out of the canvas.

On the left of the pixel editor, there are also some unique controls. Their function is as follows.

- Background Palette / Monster Palette - this toggle allows you to see the loaded graphics through a background palette or a monster palette.
- Palette Group - allows you to filter palettes by group.
- Palette - denotes which palette you are currently looking at.
- Day / Night Palette - this is a toggle that shows the day or the night palettes. Effectively, each palette has two sets of values, originally

designed for day / night systems. Without coding its use, this data is exported but not used.

- Color Swatches - these color swatches represent the colors in the currently selected palette.

Keyboard Shortcuts:

- D - Disable Palette Translation
- E - Enable Palette Translation
- B - Show Bad Pixels
- 1 - Change to RGB-A color value, ALPHA
- 2 - Change to RGB-A color value RED
- 3 - Change to RGB-A color value GREEN
- 4 - Change to RGB-A color value BLUE
- G - turns on and off the pixel grid
- A - forces square aspect ration
- T - groups pixels in 8x8 areas
- Y - groups pixels in 16x16 areas
- H - flips a selected tile horizontally
- V - flips a selected tile vertically
- Control-C - copies a selected tile area
- Control-V - pastes a selected tile area
- Control-Z - Undo
- Control-y - Redo

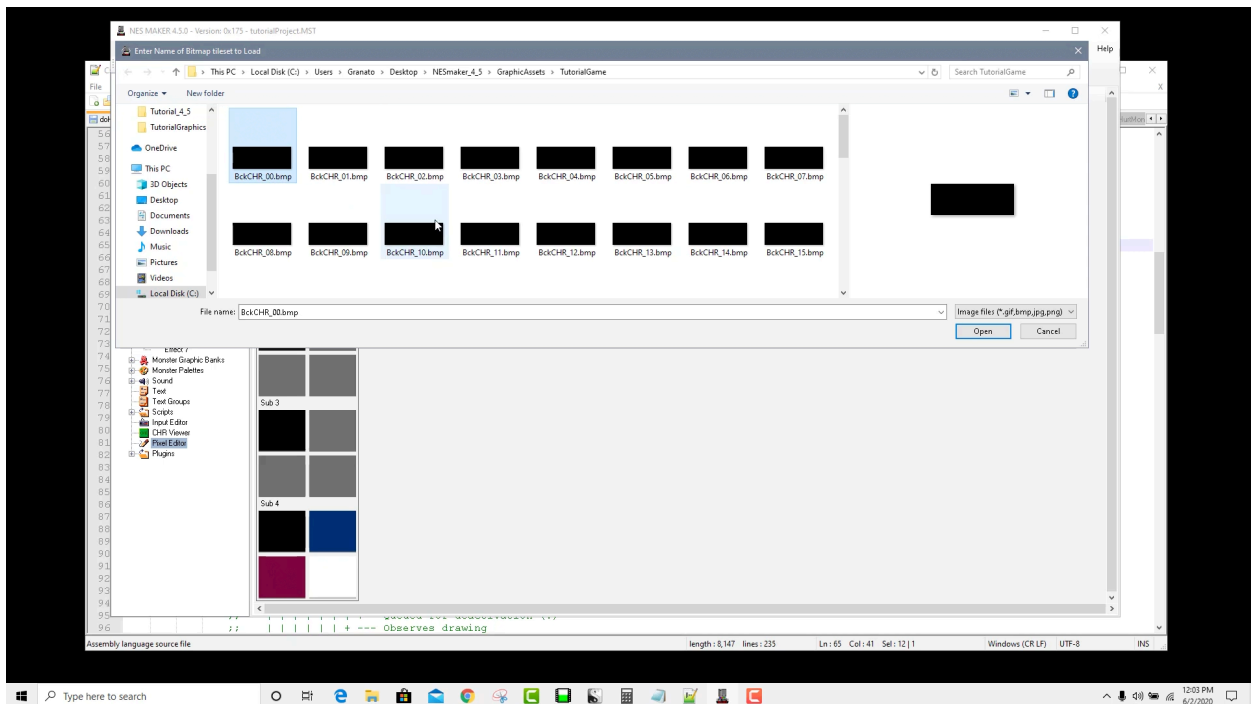
Working With Graphics

Working With Graphics

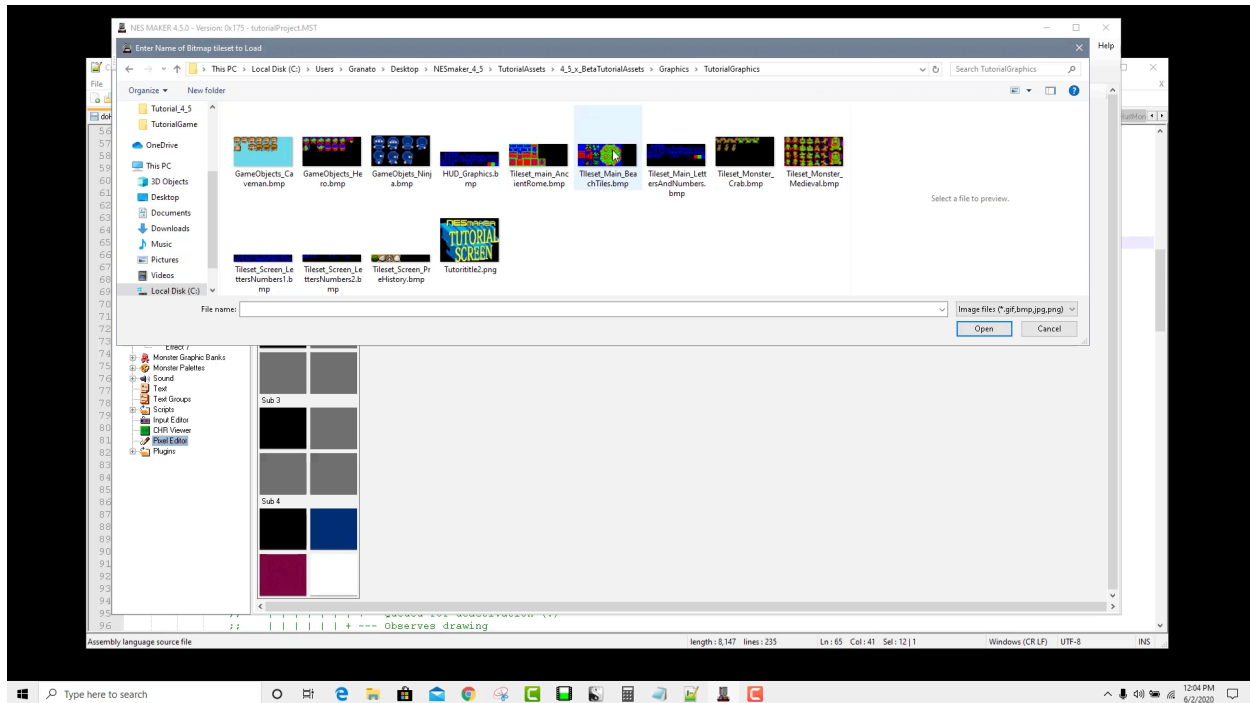
For this project, the first actual graphics we're going to load are the beach

graphics, the background environment for our game. If you followed along with the previous graphics primer, you can close any of the tabs you don't need by going to Pixel Editor -> Close tabs. We want a fresh tab. It will force you to leave one tab open, and that's ok. We'll start from there.

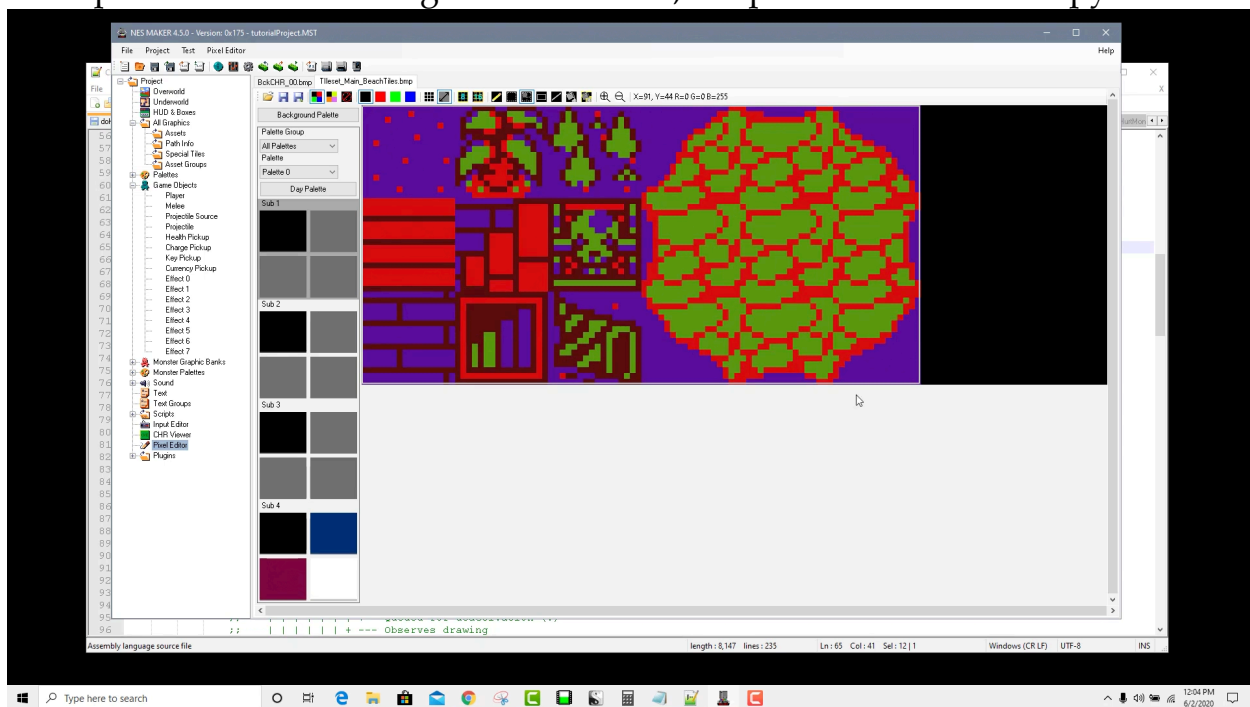
Step 1: In the menubar at the top of the pixel editor, click on the icon to Load BMP File. This will open up a finder window that shows the tilesets that are attached to your current project. Double click on BckCHR_00.bmp, or click and press open. This will open up the first main background tileset in your project, which is currently blank.



Step 2: Go to the Pixel Editor menu and click Add Tab so that you have the BckCHR_00.bmp tab and now a second tab. Click on that second tab. Again, load BMP file, but this time, navigate to the folder NESmaker -> TutorialAssets -> BetaTutorial Assets -> TutorialGraphics. Select Tileset_Main_BeachTiles.bmp. You'll notice these graphics are already NES friendly, conformed into RGB-A mode.



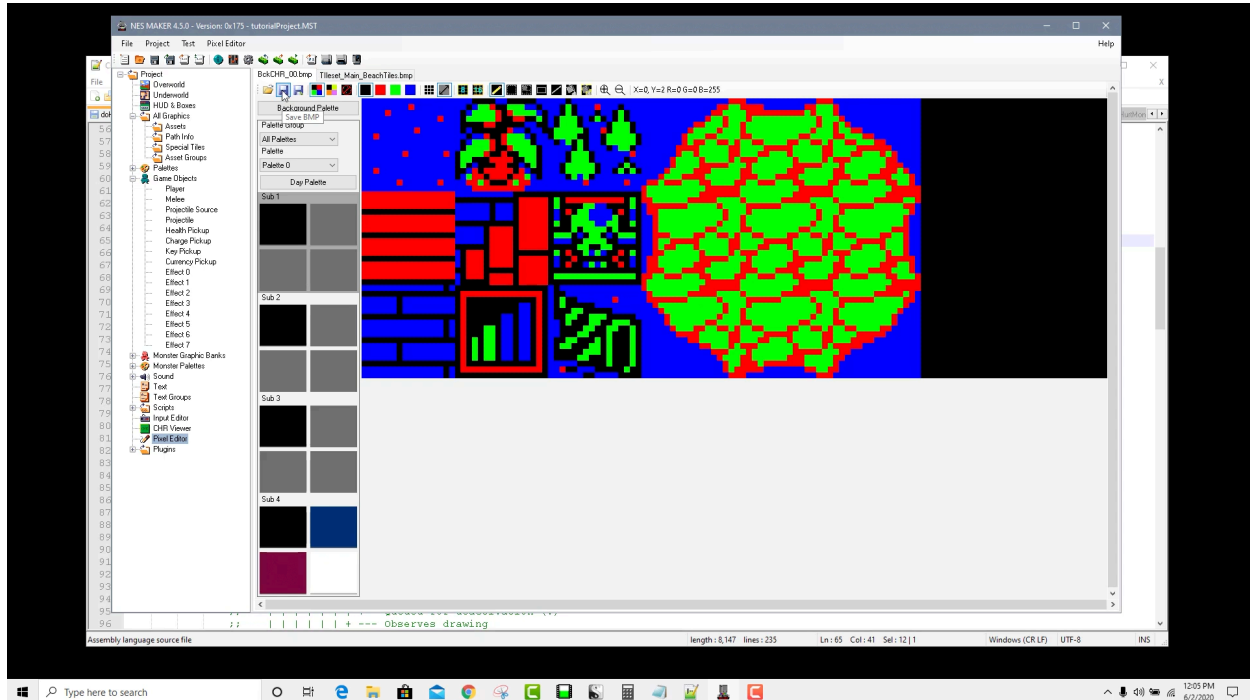
Step 3: Choose the Select Tiles tool from the menubar and click and drag from the top left to the bottom right of this tileset, the press Control C to copy.



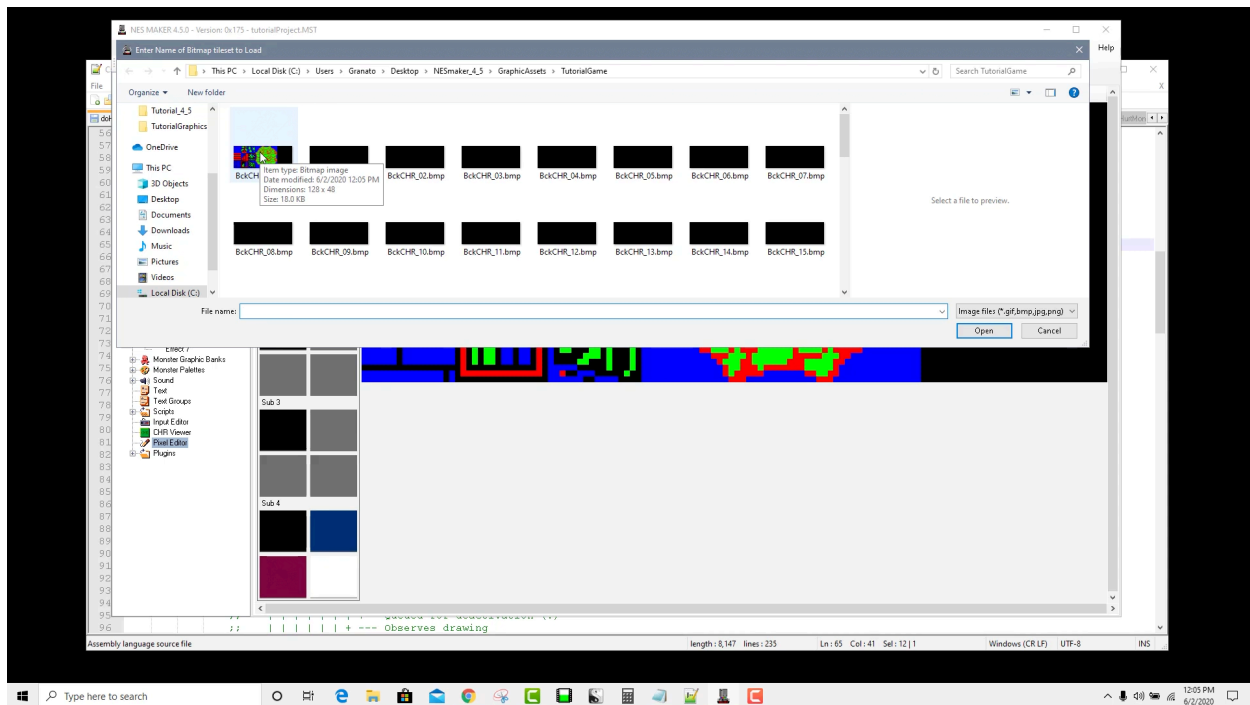
Step 4: Return to your BckCHR_00.bmp tab. Move your mouse to the top left

corner of the tileset and hit control-shift-v, which will paste your selection lined up with the top left of the grid (even though it's visibly turned off right now).

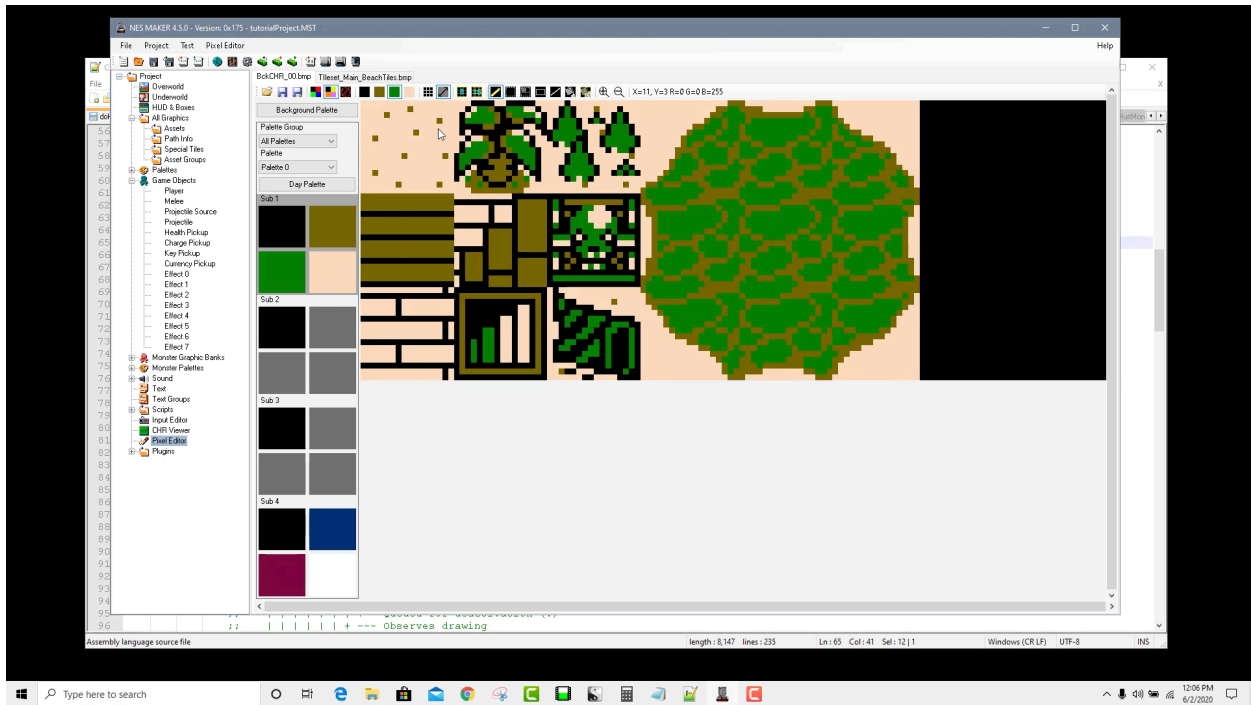
Step 5: Press the save button. This will overwrite the blank tileset for BckCHR_00.



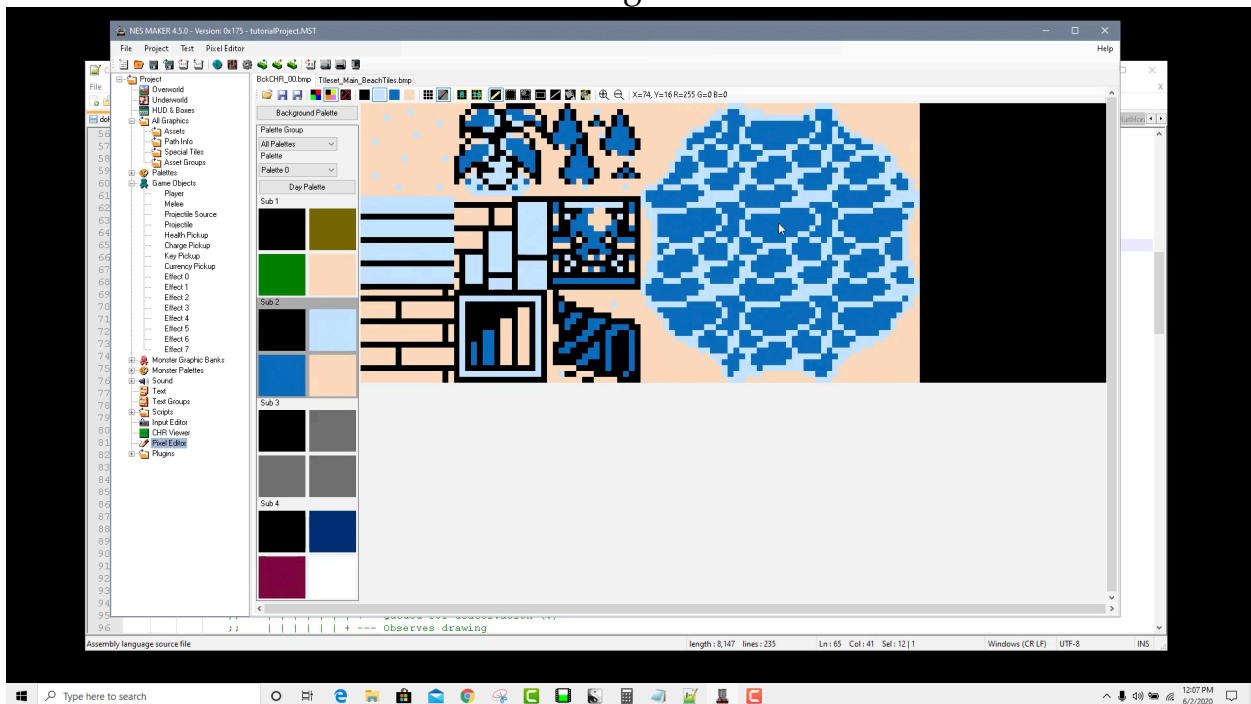
Step 6: Just to confirm that this worked, press the open button. You will be able to see that now your first tileset shows those graphics as a thumbnail (or will if you have thumbnail view turned on for this folder). We're not going to do anything from here and can just hit cancel, this is just a spot check to make sure that the right graphics are in the right place.



Step7: Now we'll work on the palette. If we go into disable palette translation mode, we will see that blue (value 3) is our ground. So click on the last swatch of the first sub palette and choose a sand color. Meanwhile, red (value 1) is used as an accent color. It is dots in the same, shadows under the tree, darkest brown on a treasure chest, ripples in the water. So for this sub palette, let's make color value 1 a darker brown color. The second color is used for the leaves, so we'll make color value 2 a green color. Enabling palette translation will show that the ground and palm tree look pretty good with these colors.



Step 8: For our second sub palette, we'll work on the water. This will have the same color sand in color value 3, but its value 2 areas are dark colored water, while its value 1 is a lighter colored ripple. Enabling palette translation will show that the water now looks like water along a shoreline.

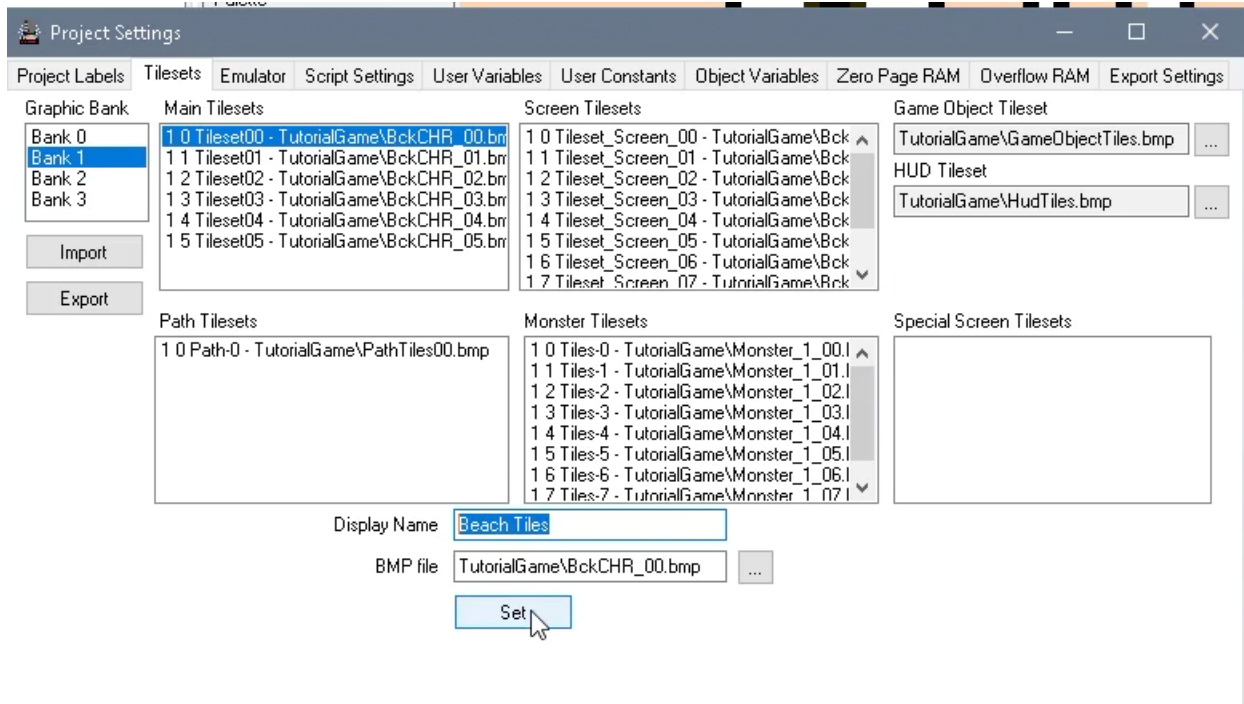


Step 9: For the third sub palette, we'll put it in palette translation mode to start, so we can see what works best for things like the shell and the treasure chest and the spikes. Making color value 1 a darker brown, color value 2 a white, and color value 3 a sand color ends up working for the spikes, chest, and shell.

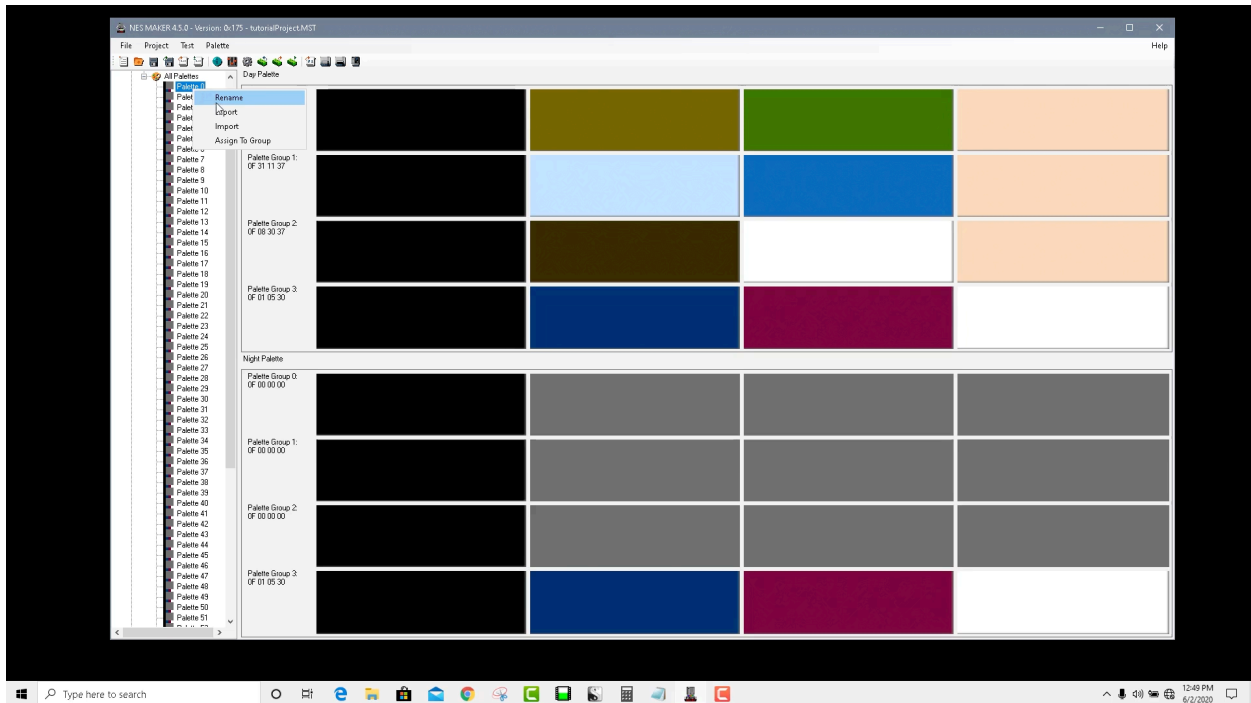
You might be asking what's with the fourth sub palette. Generally, the fourth sub-palette is reserved for HUD data. Giving a dedicated place for HUD colors means that as the game screens change colors, the HUD will stay consistent. That said, this is a default and you can change that to suit your needs. It is recommended, though, that you at least leave white as the last color in the last sub palette for any text that you might want drawn.

Step 10: Just like with Game States, we can give this tileset a proper name. It has no bearing on the code, it is just for organizational purposes to make finding this tileset easier from dropdown menus. A proper name like Beach Set is much more descriptive than Tileset00.

Again, go to Project Settings by clicking the gear icon in the tool bar. Then go to the Tileset tab. This tileset is in Bank 1, it is the first tileset in the Main Tileset group. With those things selected, you can enter a Display Name at the bottom. Here, we've written Beach Tiles. Then hit the set button. This will ensure that when selectable, the display name for this tileset will be Beach Tiles in stead of Tileset00. This is wholly optional, but can help a lot with organization.



Step 11: Also, we're going to name the palette. Right now, we know that this is Palette 0, but we want to know that this is the palette associated with the beach tiles. In our hierarchy we can expand the Palettes node, expand the All Palettes node, and find Palette 0. Right click on the palette, choose rename, and call it Beach Pal. Now in your dropdown menus when looking at palettes, it will say Beach Pal. Naming these sorts of elements as you go with descriptive names makes things much easier to find later.



The Screen Painter

The Screen Painter

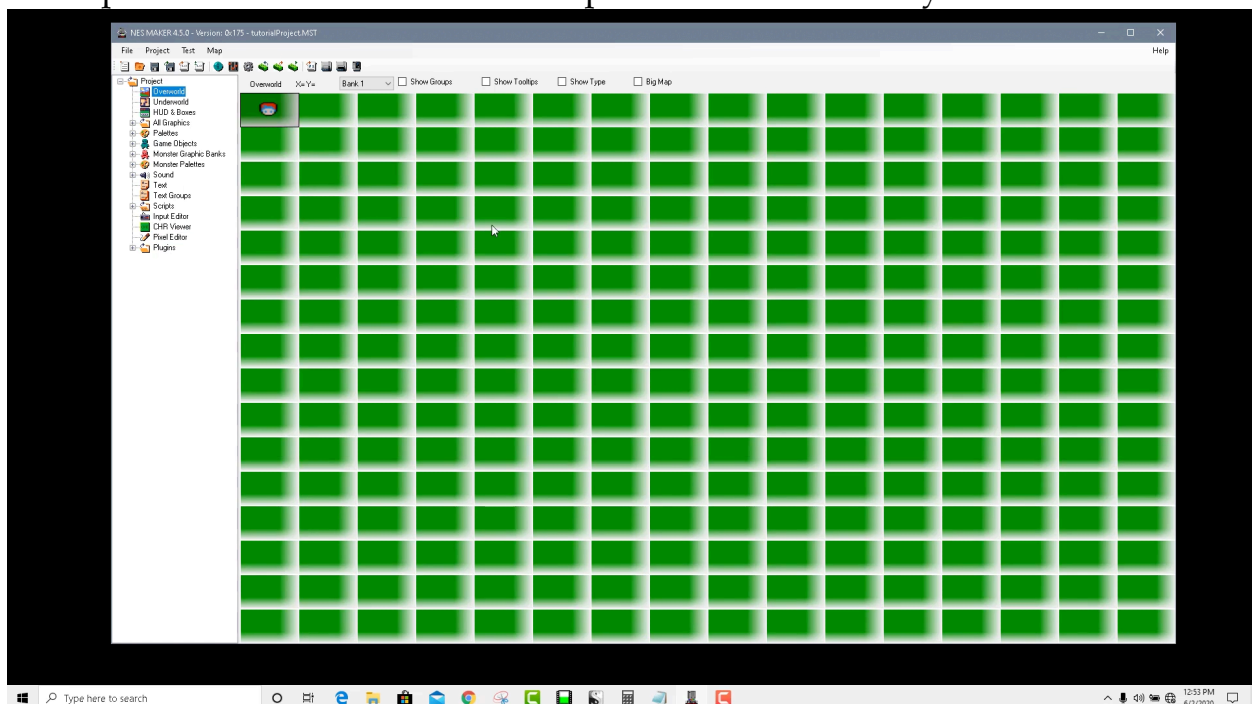
The screen painter is where you'll actually be designing the environments for your player. It is your level designer, where you will make decisions about which game state a screen exists in, the tilesets associated with a screen, objects to place on a screen, songs to play on a screen, and several other parameters.

By default, the data required by NESmaker for each screen is regimented, which makes that screen data easier to organize and work with. It is certainly not the most optimized method of creating screens for the NES, but considering that even with this lack of optimization this method allows for 512 game screens, the worlds that can be created rival even the largest NES games.

To access the screen painter, you can click on either of the map nodes in the hierarchy, Overworld or Underworld. These don't actually denote Overworld or

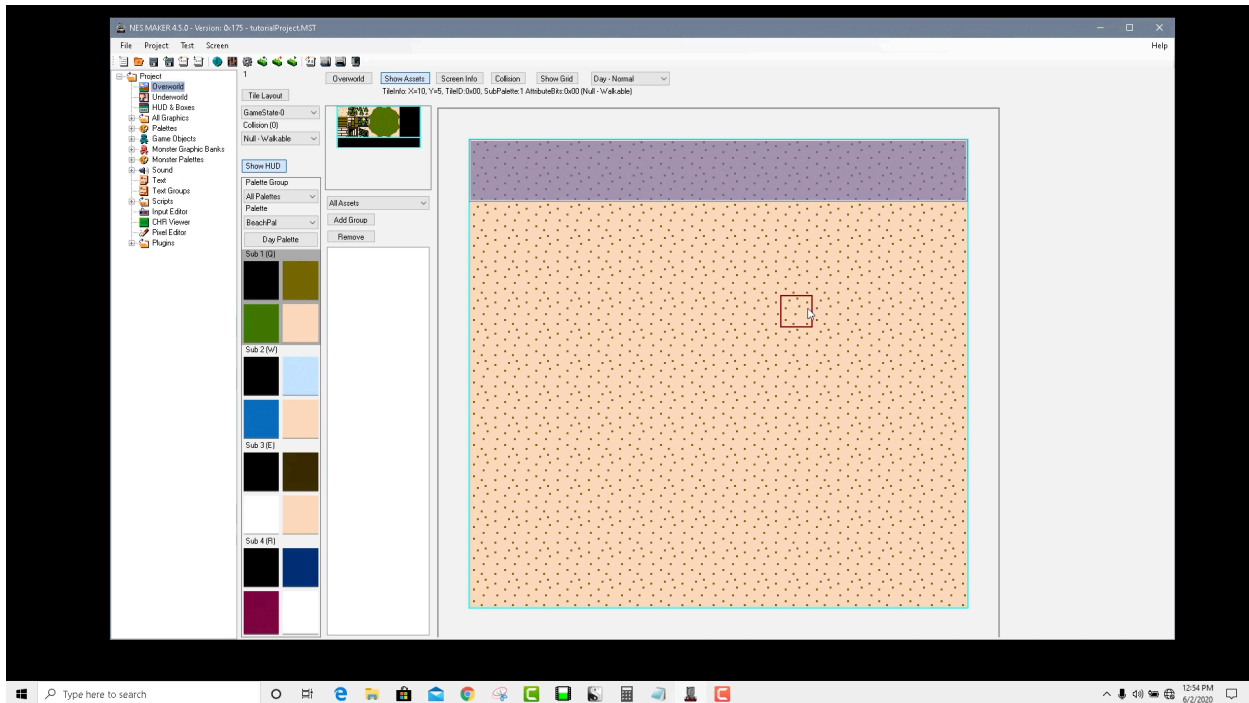
Underworld, it is just a common use of the two respective maps. In reality, these are just two maps worth of 256 screens each that can be used however you see fit or that make sense for your specific game; screens 0-255 on Map 1, and screens 0-255 on map 2. Each of these screens stores its tile layout, color information, collision layout, and general screen info to a particular part of memory that is already wired up to the provided modules, so all you have to worry about is the fun part of painting the screens.

Step 1: Click on the Overworld map node in the hierarchy.



There are a lot of interesting things that can be done in the map area that can help with development, but they're not required to understand in order to build our first screen, so we're going to save looking at them for a later step.

Step 2: Double click on any of the screens to bring up the screen painter. When you do, you are effectively creating a new screen and populating all of its parameters with null values.

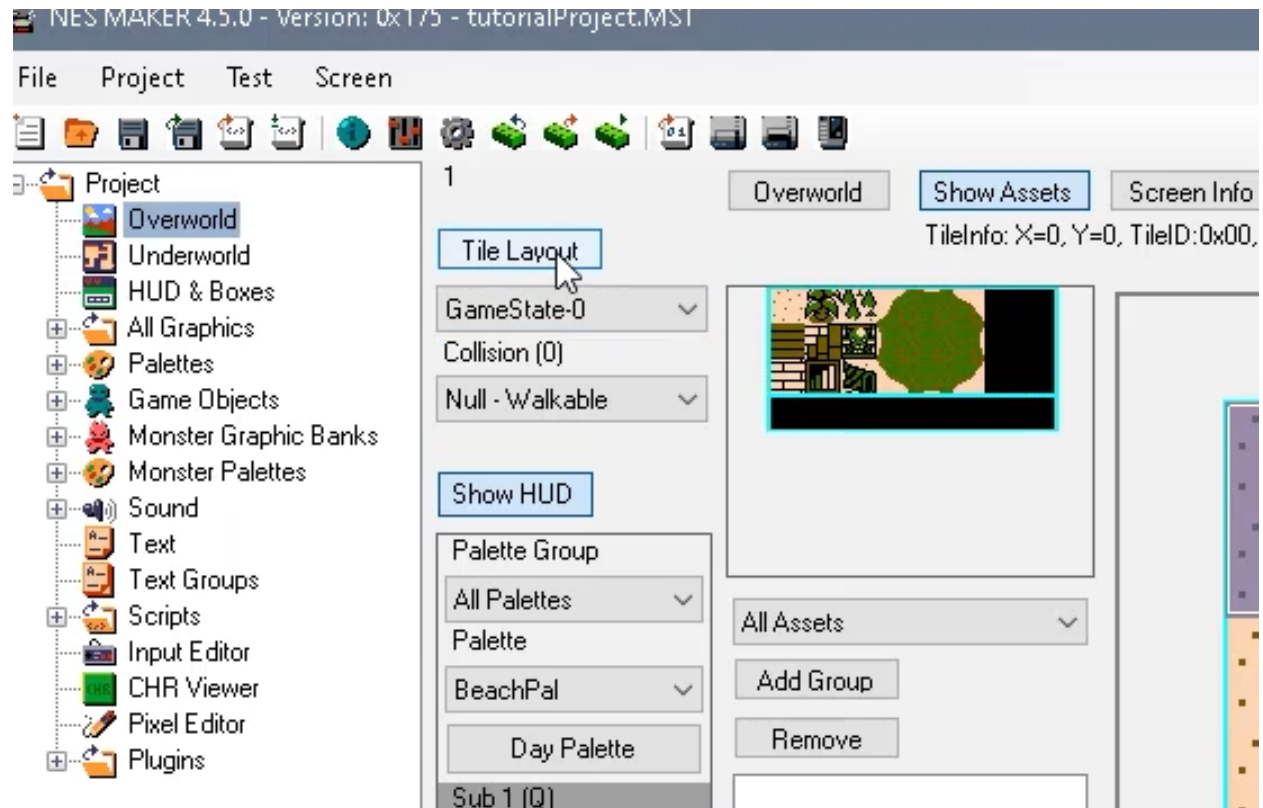


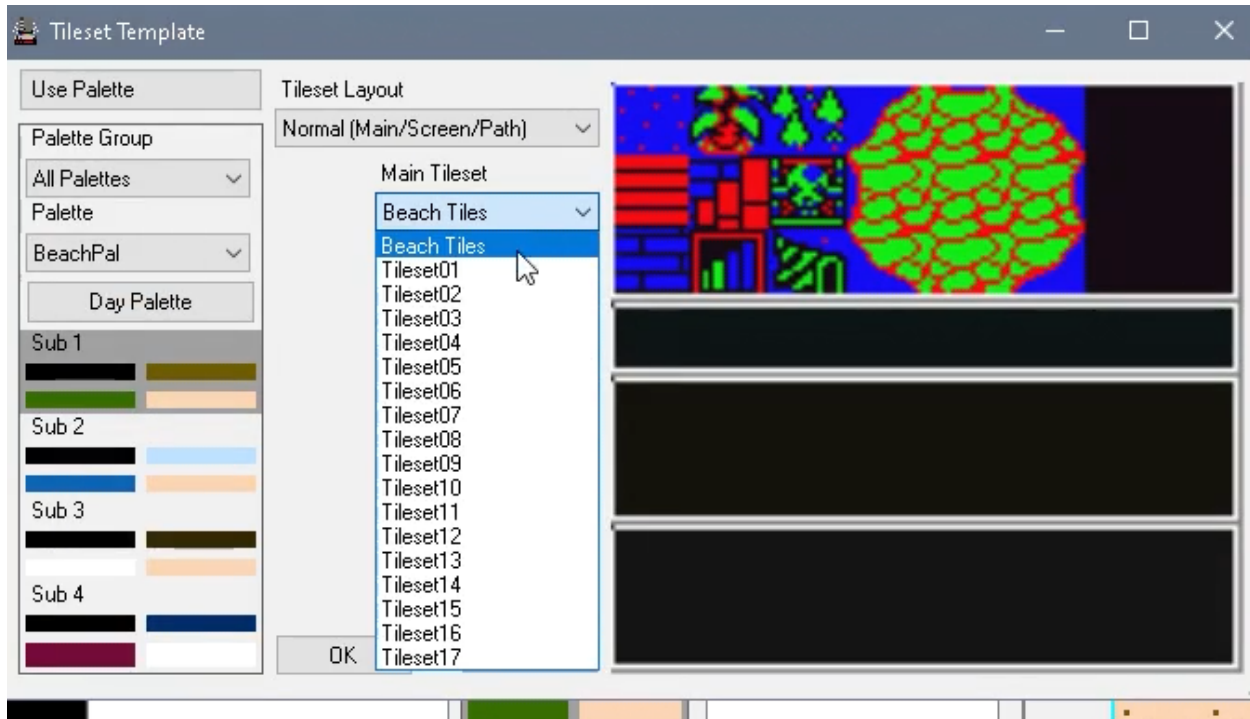
In the screen painter tool, you'll see that a new menu item has been added called Screen. It has two options, Save Screen and Load Screen. This saves and loads parameters of a screen. It is important to understand that it is saving and loading all of the referenced, not actual graphics. So, for instance, if you save a screen from a fantasy game where you have a forest tileset as your tileset 0 with a tree in tile slot 0, and then you try to load that screen into a platform game project where you have a side scrolling city tileset as tileset 0 with bricks from a building in tile slot 0, you would see bricks where in the fantasy game you saw trees. The screen painter backs up and restores the values. What those values point to may be different from game to game. However, this is quite handy when working on screens from the same game, where all of the references would be the same.

You'll notice that our beach sand is covering the screen. This is because by default, in creating a new screen, it loaded null values (or zero references) for everything. It loaded background tileset 0, which is the beach tileset we just created. It loaded palette 0, which we renamed to Beach Pal. It filled the screen canvas with 0 values, which means the tile in the top left corner of the tileset, using the first sub palette. Not that you can see it visually at this moment, but it also loaded 0 for all of the collision types for each tile, and for this module, zero means walkable.

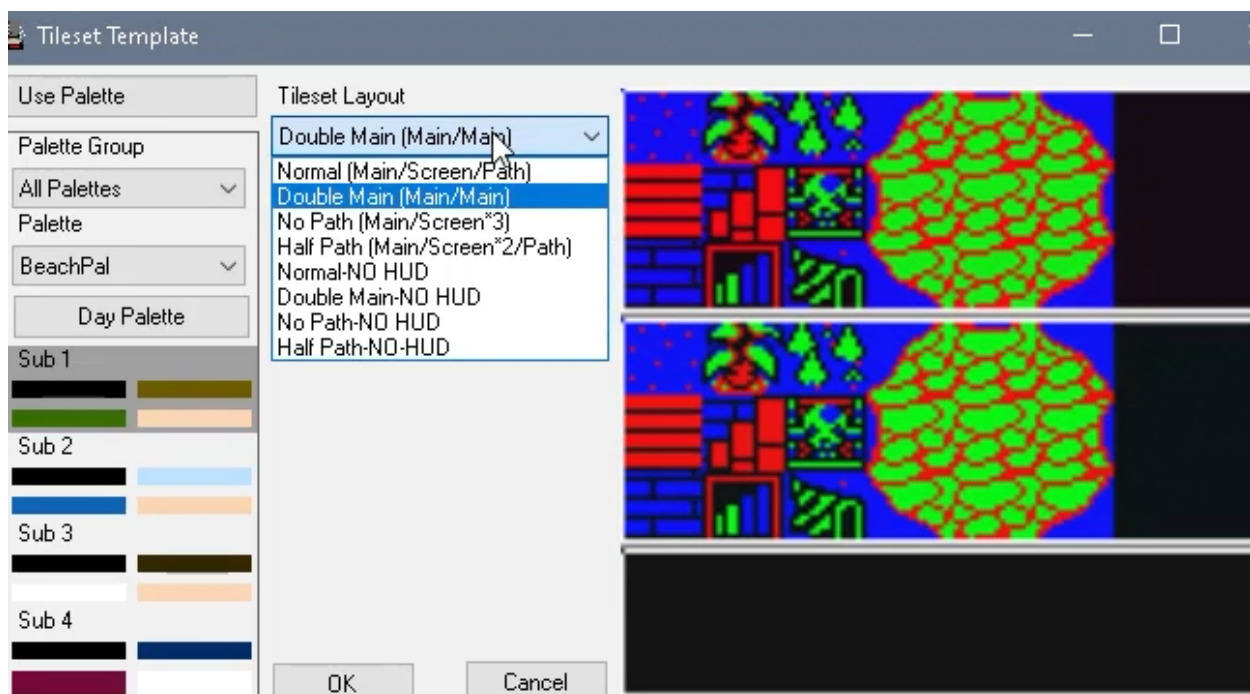
That leads to a good tip. Always make the top left corner of a background tileset a ground tile. If you do, when you create a new screen, you will start with full ground coverage.

If you wanted to change tilesets, you could go to Tile Layout and choose from the available tilesets for each subsection of graphics that will be included for use for tiles for this screen.





Also, you can change the layout of those subsections. The default layout is one Main Background Tileset, one Screen Specific tileset, an area for paths, and an area for hud graphics. But there are multiple ways to mix and match your various sized tilesets depending on the screen's needs.



Main tilesets are the ones in your project's graphic assets folder that begin with the prefix BckCHR. They are 128px x 48px. This generally equates to the space for 24 default background tiles, which are 16x16px in size (three rows of 8). Screen specific tilesets are the ones with the prefix BckSSCHR. These are 128x16, having space for 8 default background tiles. Path tilesets are a block of four single 8px rows and are a bit special, procedurally generated when you draw them to the screen to form paths or water or roads that can wind and turn in a predictable way based on a single row of graphics. The HUD area is a 128x32 block that gives enough space for all alphanumeric characters, basic punctuation, and some room left over for HUD features.

The reason there are different layouts is to facilitate different types of games. For instance, if you have a scrolling platformer, you probably are not using the traditional HUD graphics, because you'll likely be drawing the HUD with sprite graphics rather than using a background. You could get a whole 128x32 px chunk of tile space for backgrounds by using a layout that does not use a HUD. Or, you may have a game with very clearly defined geographical areas, where using double main works well, because all screens within an area use the exact same grouping of tiles. Chances are, though, most users will create games that have some repetitive tiles in an area but some variable tiles that may span across various areas. For instance, you may have a top down adventure game that has both a forest and a mountain, but you want to have the same type of rocks in each. You could make a main tileset for the forest, and a main tileset for the mountain, but use a screen tileset that could be shared that has rocks in it. That way, you don't have redundantly use up space on the main tilesets and can share common elements between screens with a screen specific tileset.

These are just a few examples. But basically, having the ability to change the tileset layout to include different sized groups of tiles allows for a bit more flexibility and memory conservation instead of loading full tilesets for each screen. The types of layouts that are available are as follows.

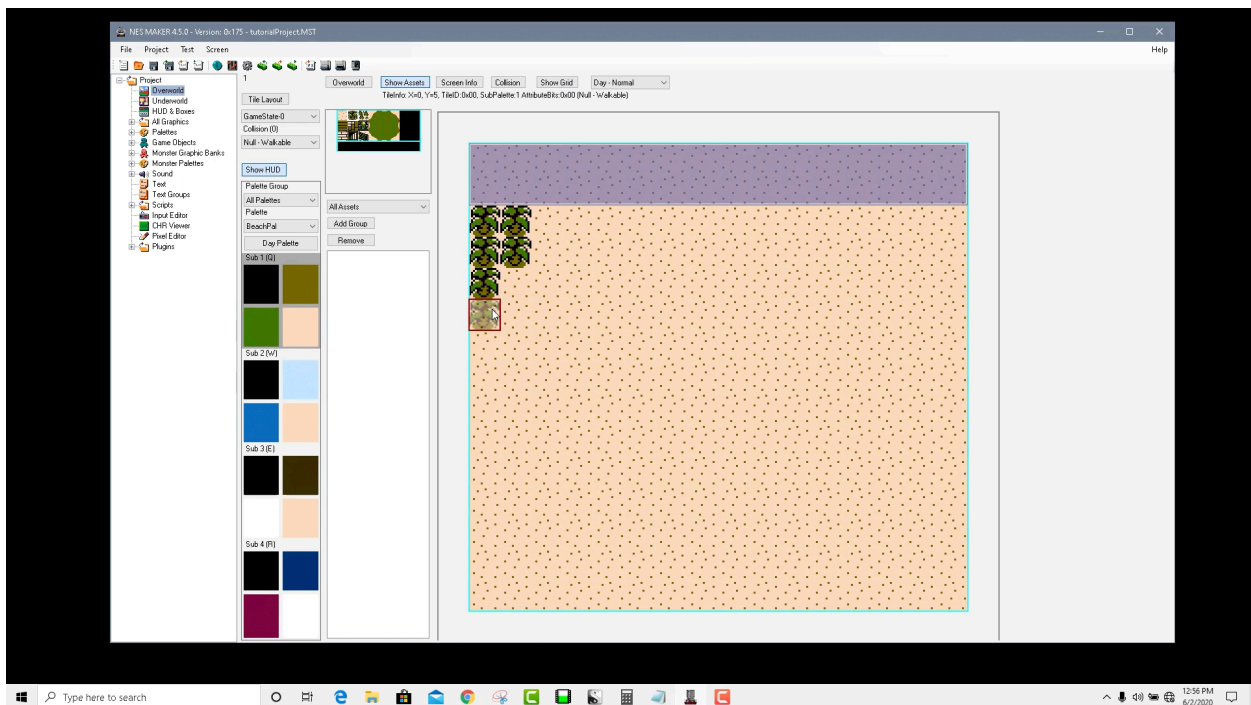
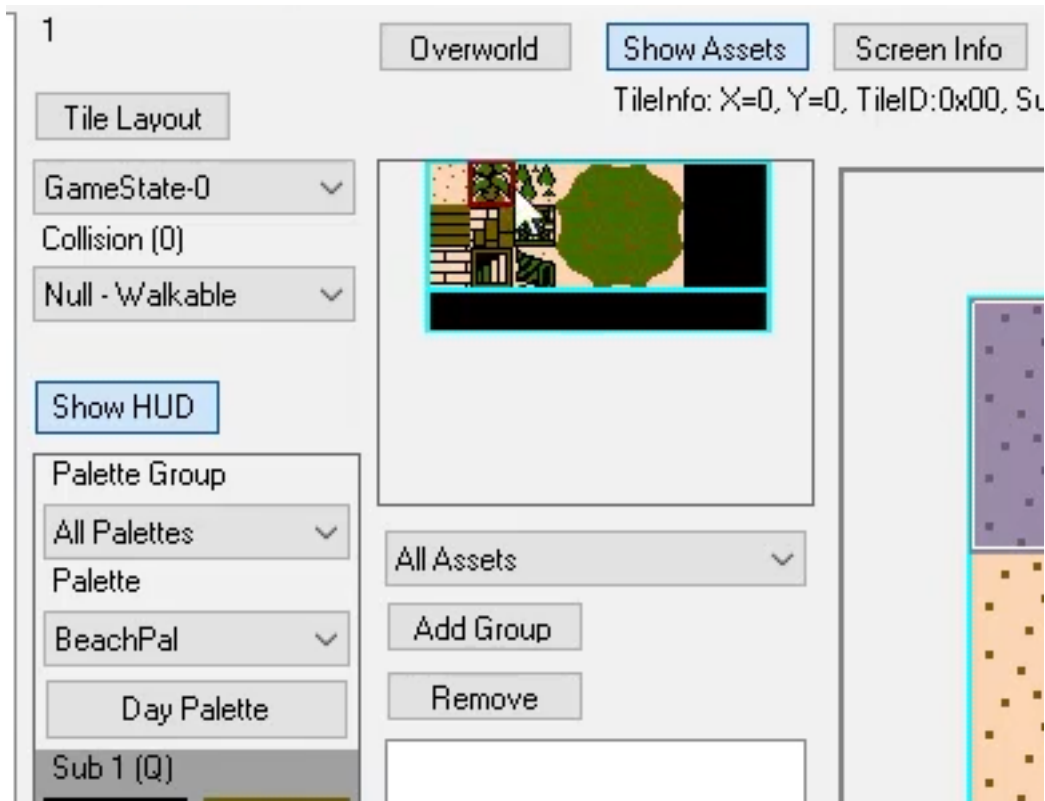
- NORMAL (Main/Screen/Path) - this allows you to load one main background tileset, one screen tileset, one full path area, and the HUD graphics.

- DOUBLE MAIN (Main/Main) - this allows you to load two different Main Background tilesets plus the HUD graphics.
- NO PATH (Main/Screen*3) - this allows you to load one Main Background tileset, three screen tilesets, and the HUD graphics.
- HALF PATH (Main/Screen*2/Path) - this allows you to load one Main Background tileset, two screen tilesets, half a path tileset, and the HUD graphics.
- NORMAL NO HUD - this allows for a Main tileset and a path tilesets and three screen tilesets.
- DOUBLE MAIN NO HUD - this allows for two Main tilesets and two screen tilesets.
- NO PATH NO HUD - this allows for one Main tileset and five screen tilesets
- HALF PATH NO HUD - this allows for one Main tileset, half path tileset, and four screen tilesets.

For this project, we will use the default, NORMAL tile layout already selected.

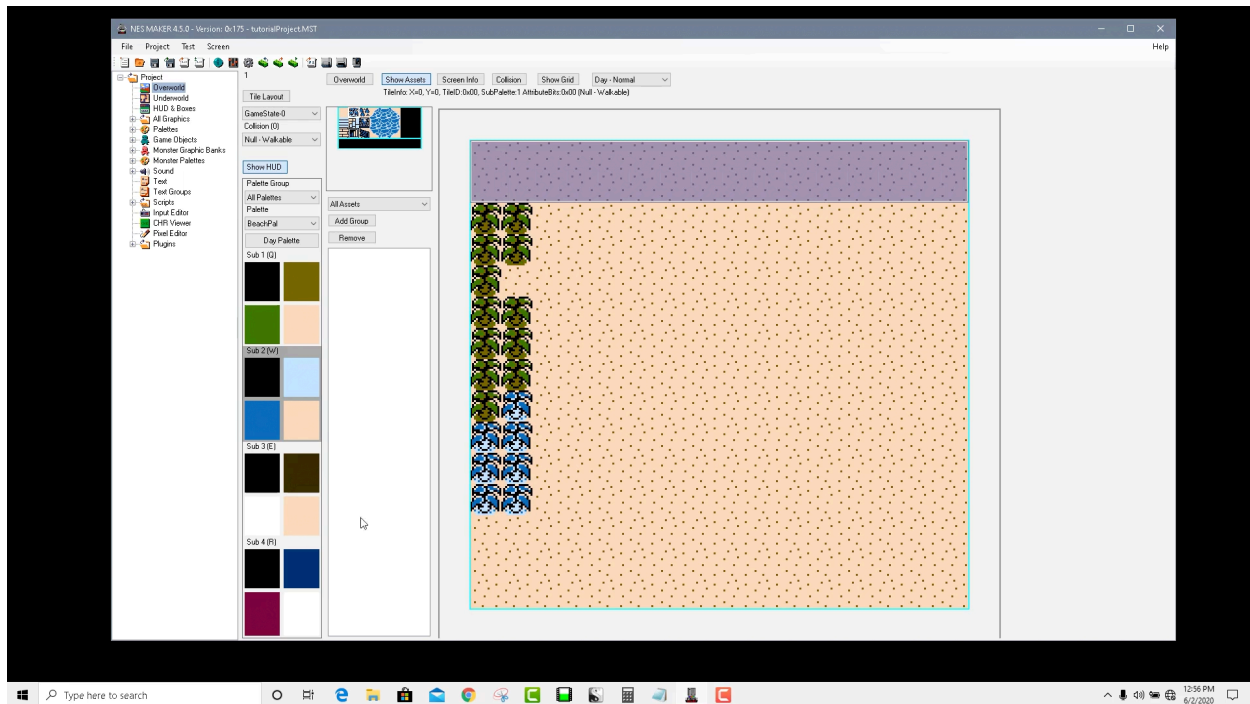
Step 3: There are two many ways to create the look and flow of your screen. One is an asset driven approach, where you use your tileset to create assets of any size that you can select and then stamp onto the screen canvas. These assets include all of the collision data, tile data, and color data. This may be familiar to many people who have used other game development tools. It's very convenient if you're making complex, multi-tiled assets. However, it takes time to set up each asset for each tileset. For some projects, using assets is absolutely the preferred approach, and we will look at how to work with assets in the Working With Assets section of this instructional.

For some projects, though, it is much easier and faster to directly paint data to a screen. To paint graphics from your tileset to the screen, just click on one of the tiles in the tileset viewer. Your cursor is now a brush that will paint that tile graphic using the chosen sub palette colors. By clicking and dragging around the screen, you will paint on the 16x16px grid.



By changing the sub palette, you'll notice that the tileset changes color. The reason for this was explained in the NESmaker Graphics Primer. The trees that

you see in the tileset aren't actually those colors - each pixel is a value that is pushed through the selected four-color palette. So when you choose a different set of four colors (pick a different sub palette, or change the palette altogether), you can paint the same asset with those colors. If you ever noticed that the clouds and the bushes in the original Super Mario Brothers looked exactly the same but were different colors, this is why. The same tiles were being drawn to the screen through two different four color sub palettes.

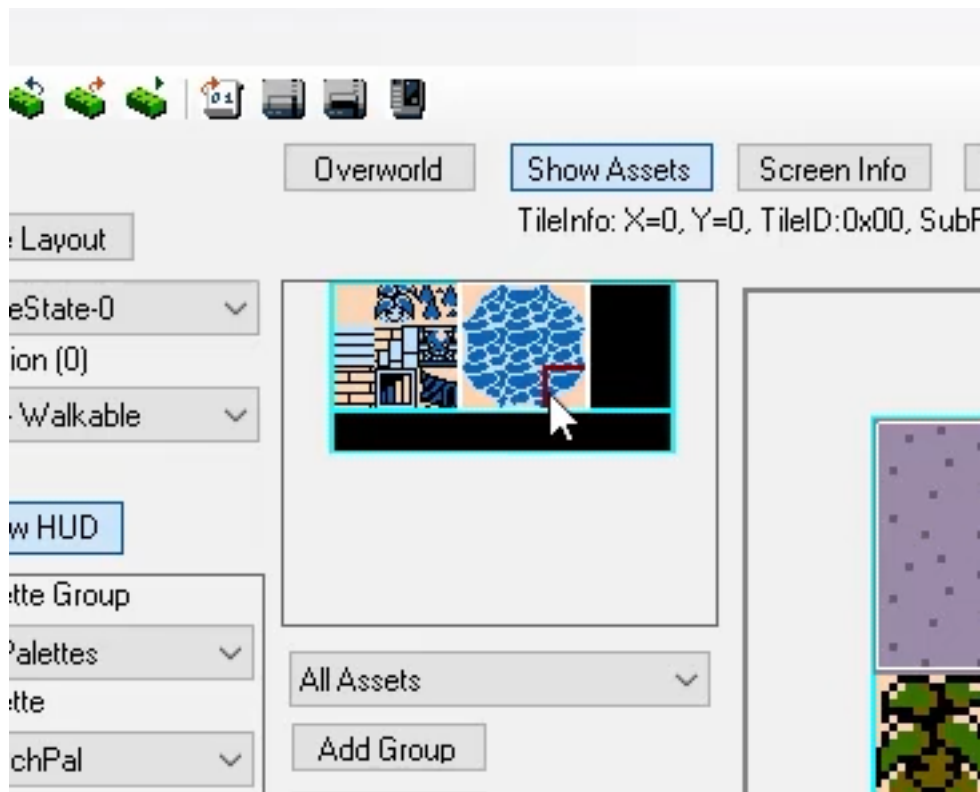


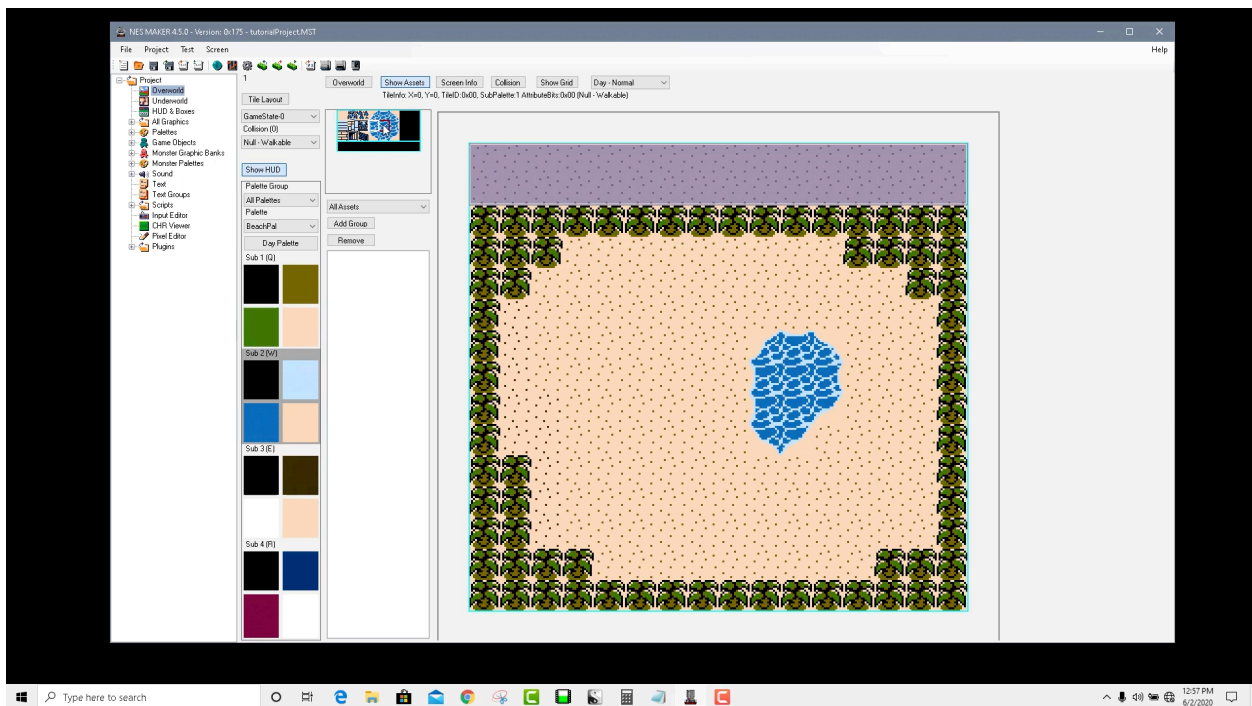
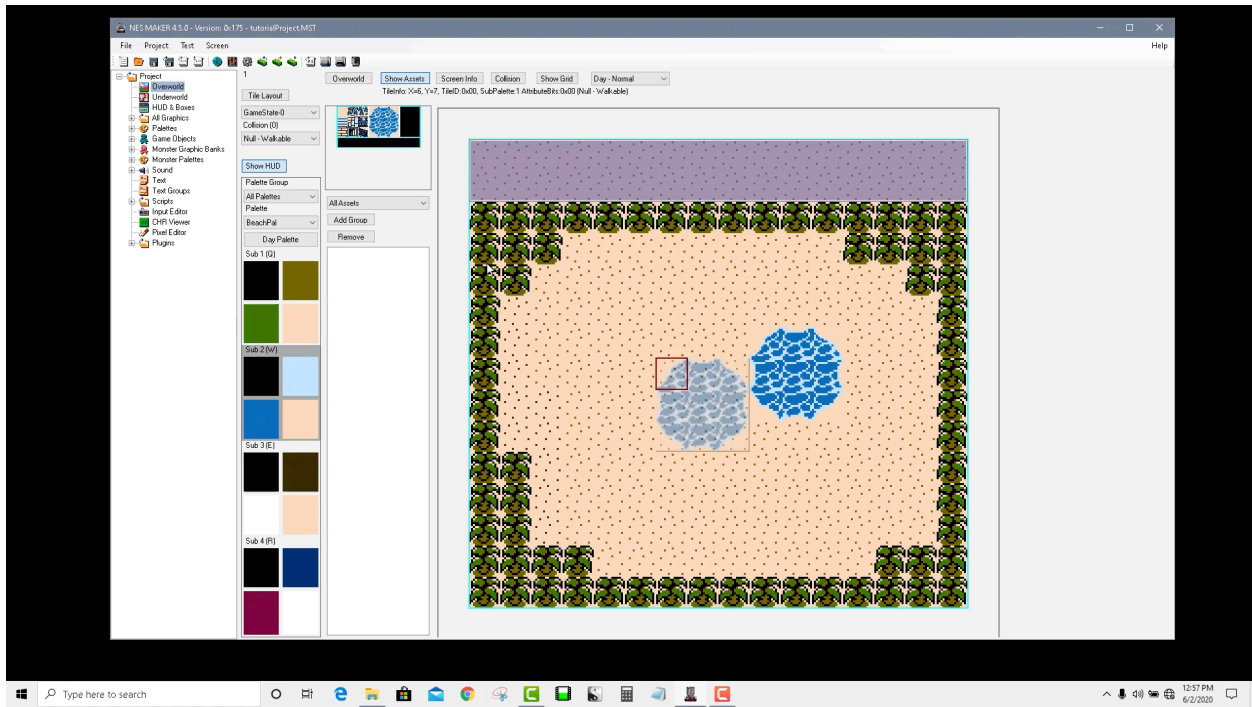
There really is no such thing as erasing a tile. Every grid space will have some value, even if that value is zero. So to erase tiles, it's as easy as selecting the ground tile, choosing the first sub palette, and painting the same way you'd paint any other tile. This is another reason why it's a good idea to always put your default or null background tile in position zero on a tileset.

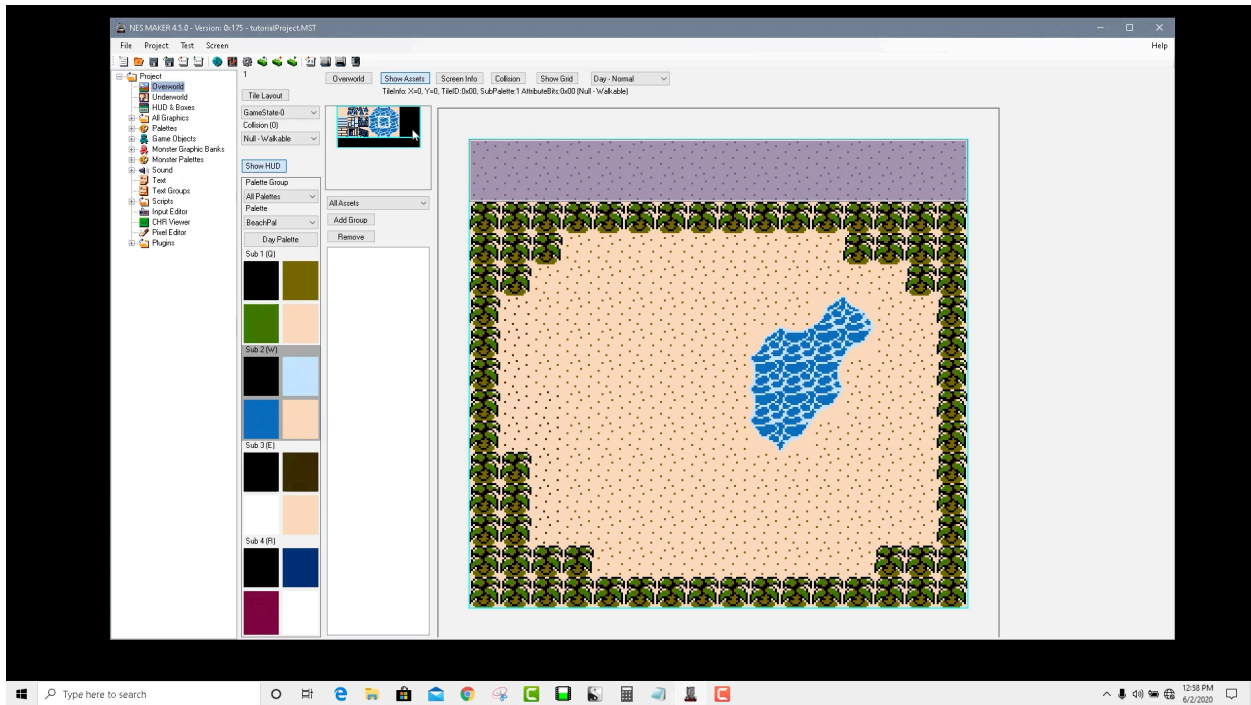
Step 4: Use the tile painter to create a basic screen with the borders blocked off. When it comes to the top of the screen, the shaded area is currently denoting where your HUD will be placed. The size and position in your HUD can be changed, as discussed in the HUD Basics section of this instructional, but for now, we'll just use the default HUD area. You do not have to place anything in the HUD area, as when you run your game, it will be overwritten by the HUD

anyway.

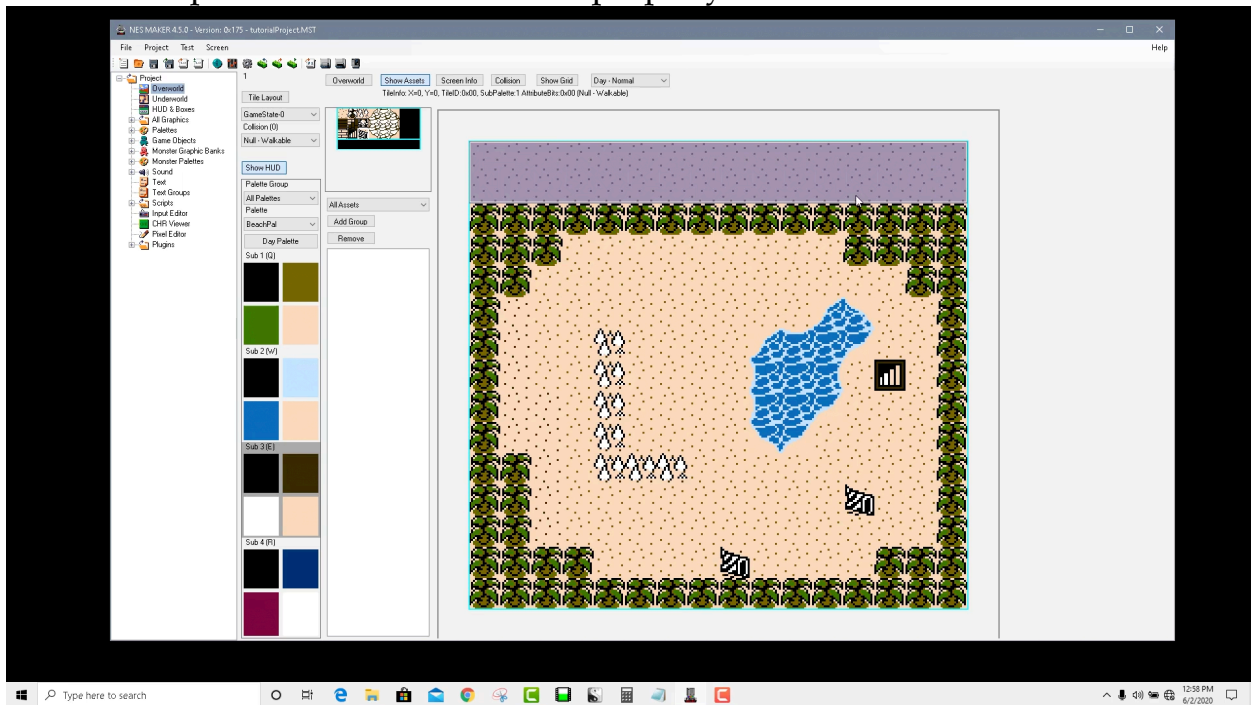
Step 5: Choose the water sub palette. Go up to the tileset view. Rather than grabbing a single tile like you did for the tree, shift-click and drag across the water pool to select multiple tiles at once. You'll now be able to place that entire selection at a time rather than having to construct the water pool one tile at a time. You can still edit each tile as you'd like, but this is a way to paint with a tile brush that might be larger than the standard tile size of 16x16px.





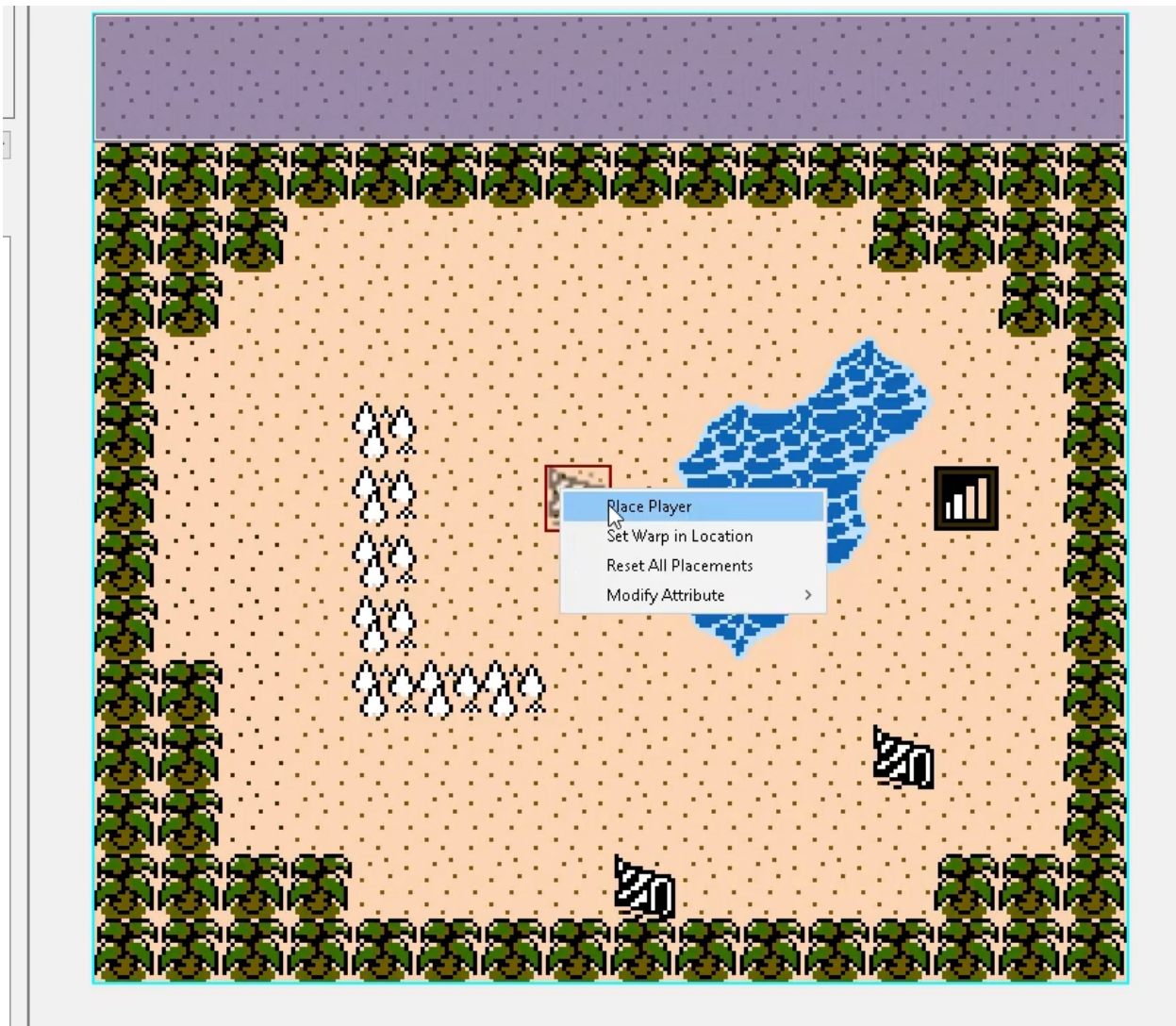


Step 6: Choose sub palette 3, add a few shells, some spikes, and your staircase. It's important to understand that as of right now, there is no collision data assigned to any of these things. Right now, all collision data for each tile still reads as tile 0. All we have done is paint the screen using tiles from our tileset. The next step will be to make the tiles properly functional.

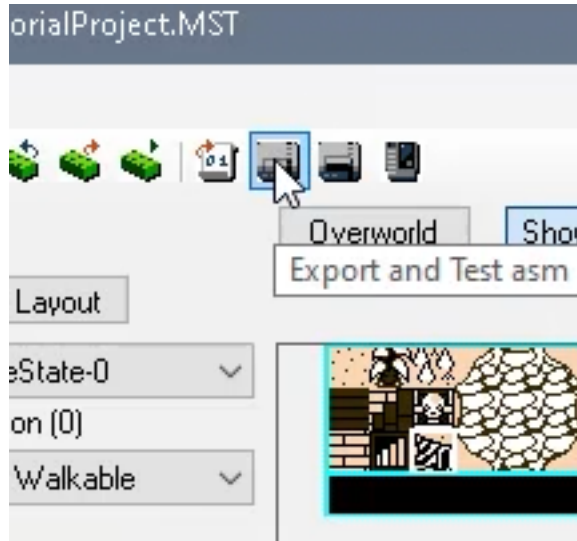


We now are in a place where we can test and see some of the things that we created in the emulator. The game won't do anything yet, but we can verify that our correct tileset loads, and the tile map that we've created for this screen properly loads.

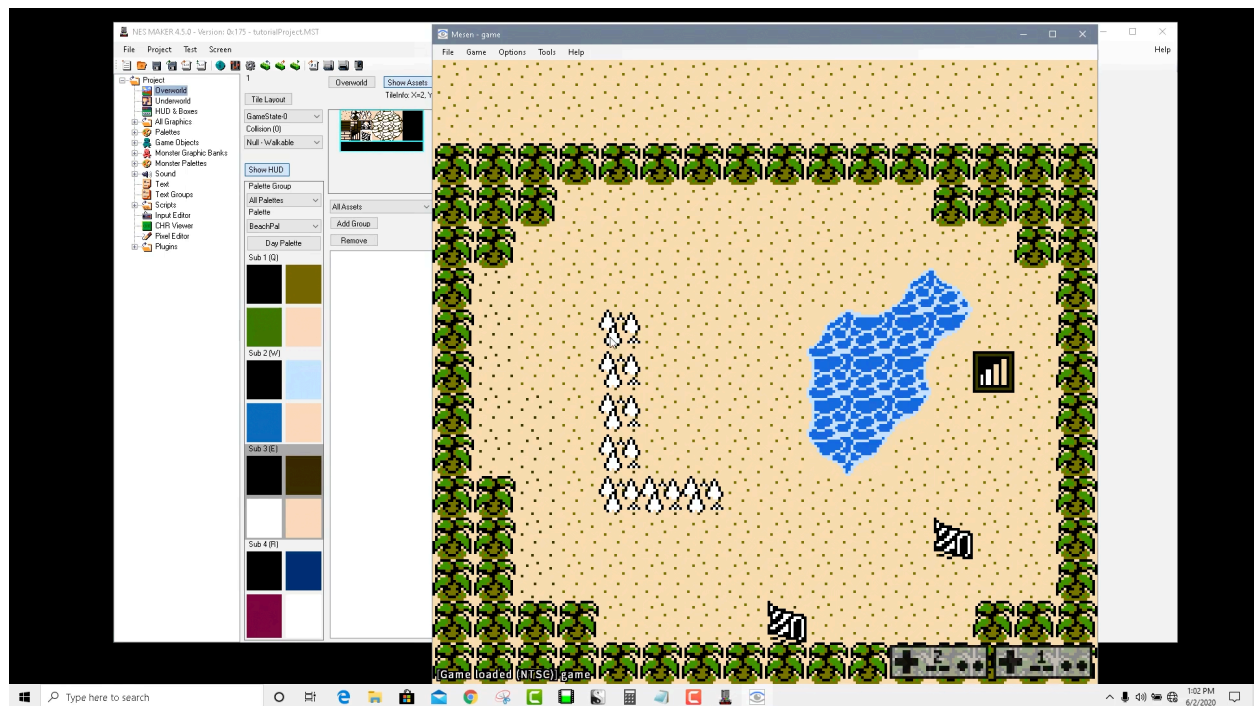
Step 7: Right click somewhere on the screen and choose Place Player. If we had already built our Player Game Object, we would see him in that spot, and as far as the game is concerned, the player now exists in that spot, but since we haven't created a player yet, he is invisible. Still, this lets the game know that this is the screen we want our game to start on.



Step 8: Go to the NESmaker menu bar and click on Export And Test. As we saw before, this assembles the game. Except this time, instead of null values everywhere, it will assemble using the data being generated by what we've created in the tool thus far.



When you press any key to continue after assembly is finished, it will launch your game in its current state in the MESEN emulator, provided with NESmaker courtesy of its developers.



You'll notice that the top HUD area is empty. That's because we have not yet created HUD, so all of the tile values are set to zero for that area, and as we

discussed earlier, a value of zero will default to the tile in the top left corner of the currently loaded tileset. Don't worry, when we set up the HUD this will be taken care of. As for right now, just don't worry to much about that area.

The Player Game Object

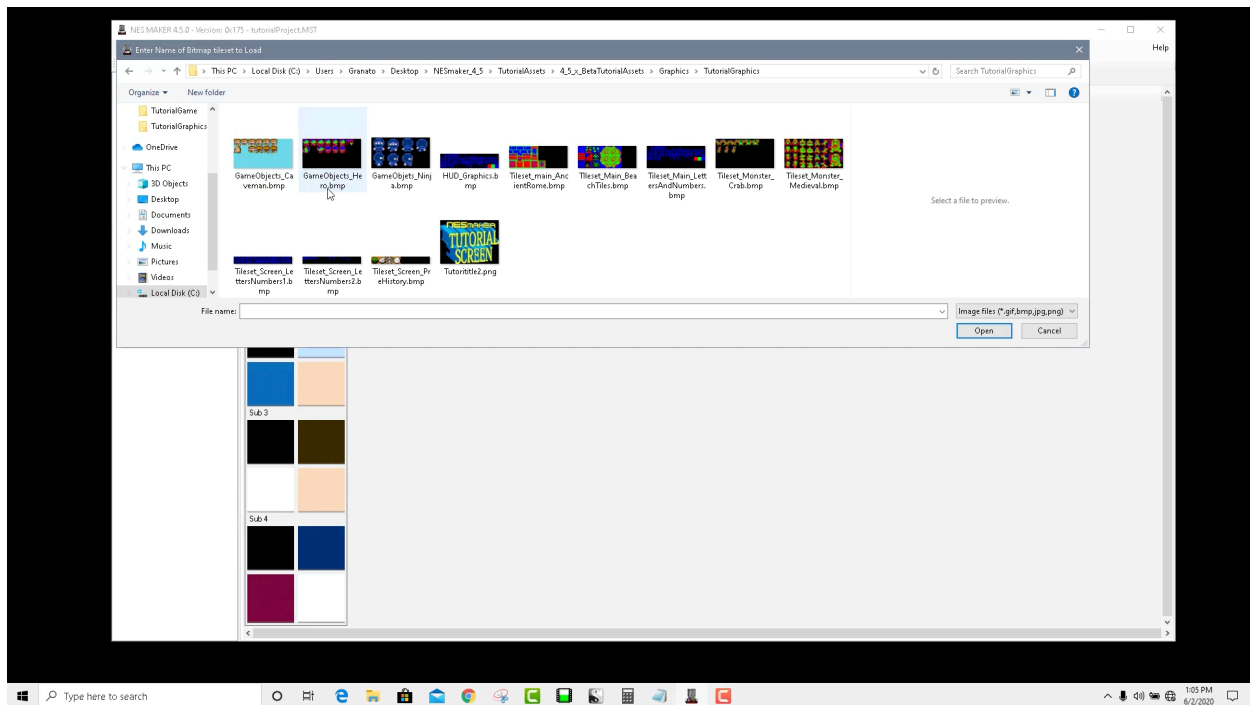
The Player Game Object

Now that we have a stage set up, we need an actor. In order to create a player object, we need to do a few things. Of course, we need to bring in the player character graphics. We also need to set up an object that will hold all of the parameters for a player.

The Game Object Tileset

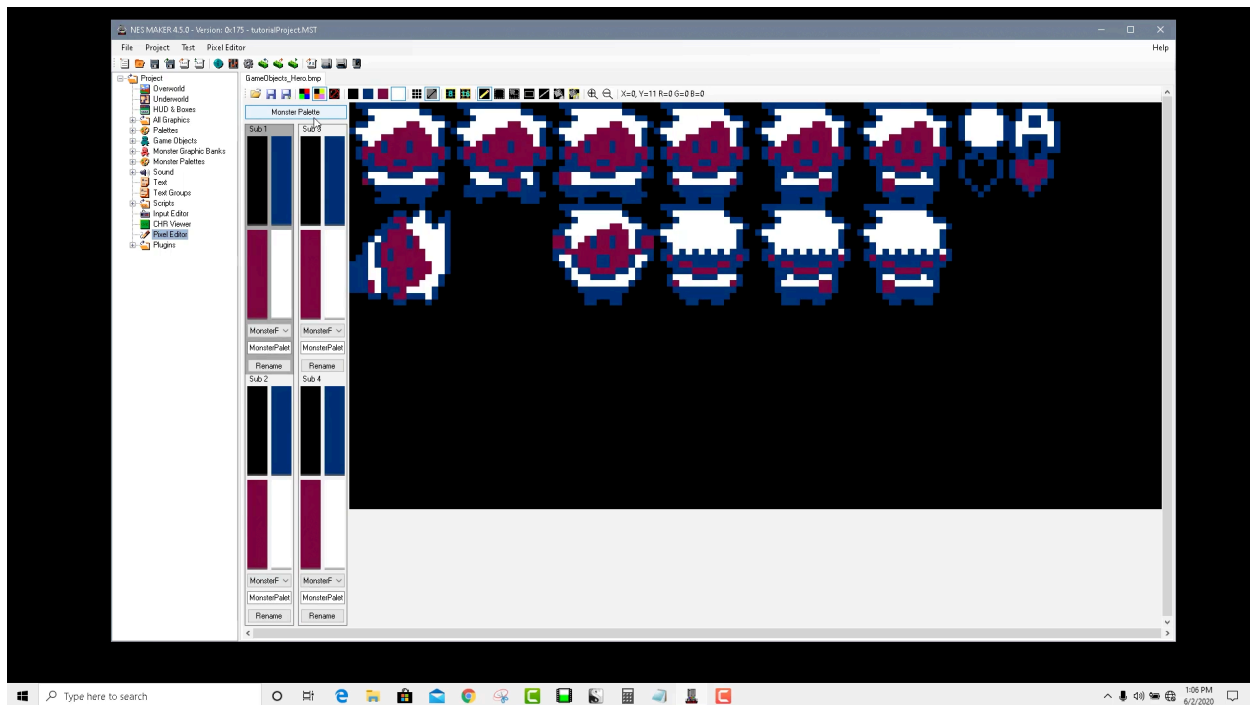
Step 1: Open the pixel editor. If you've previously been working on any other tilesets, they'll likely still be present, and you won't be able to close a tab if it's the last tab. That's ok though. Close all except for one tab. With that tab selected, press the icon to load a new bmp at the top left of the pixel editor menubar.

Navigate to your root NESmaker folder / TutorialAssets / 4_5_xBetaTutorialAssets / Graphics / TutorialGraphics and double click on the GameObject_Hero.bmp file. You'll notice that this file is already in RGB-A format.



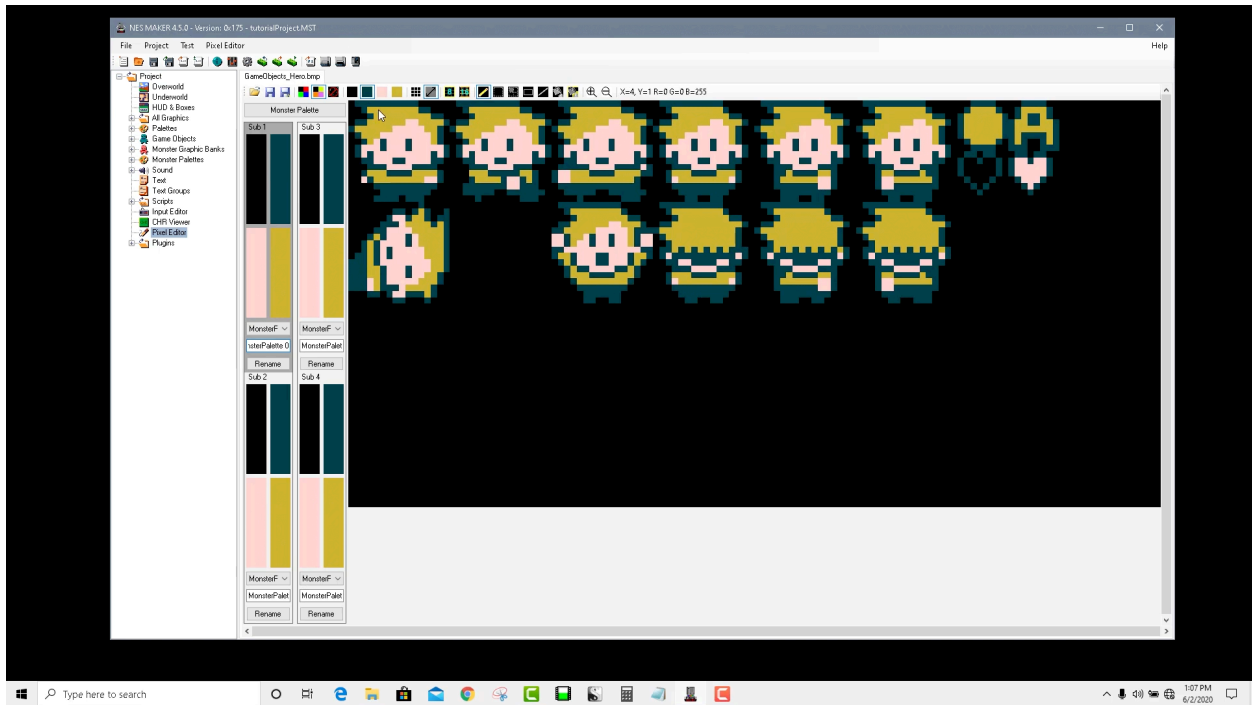
Step 2: If you have it in Disable Palette Translation mode, you'll see the graphics in RGB-A format. However, if you are in Enable Palette Translation and you're looking at the hero graphics through the current palette, it will look pretty ugly. That's because right now, you're looking at this through the background palette we designed for the beach. You do not want to start changing colors of the Beach Palette, otherwise you will be making changes anywhere the beach palette is applied. For instance, if you change the ugly green color in this beach palette to a skin tone color to work better with the hero graphics, that means on the screen we just created, the leaves of our palm trees will appear flesh colored instead of green, and that's obviously not what we want.

Instead, we want to use a different palette. By default, NESmaker has two different palette types, background palettes and object (or monster) palettes. Click on the small button in the top left corner of the pixel editor where it says Background Palette to toggle the view so you're working with object palettes.

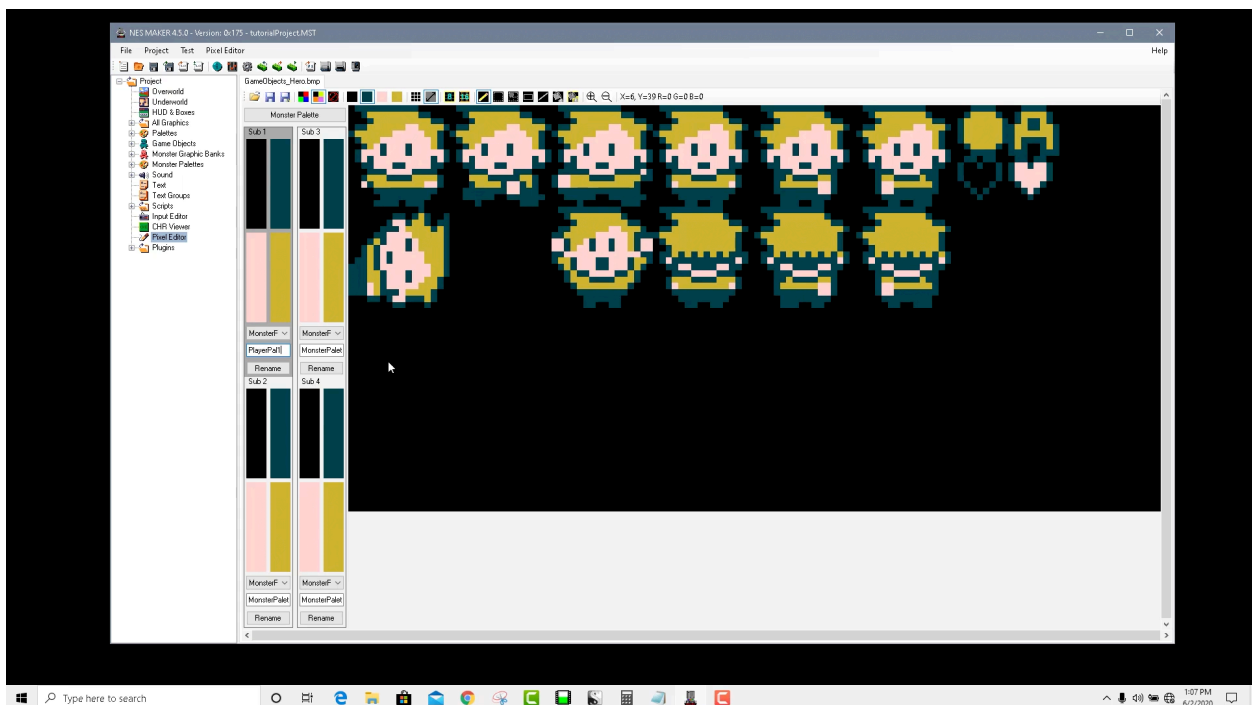


Just like with backgrounds, when you're working in the pixel editor, you're picking your palette with which to view your object's graphics. This is not actually coloring the object, but rather, letting you see what the object would look like if the selected palette was called by the game screen. You're not actually setting any colors here for the pixels at all, just building a palette that can be, if you choose, used by a screen, and if it is, will look like what you're seeing here.

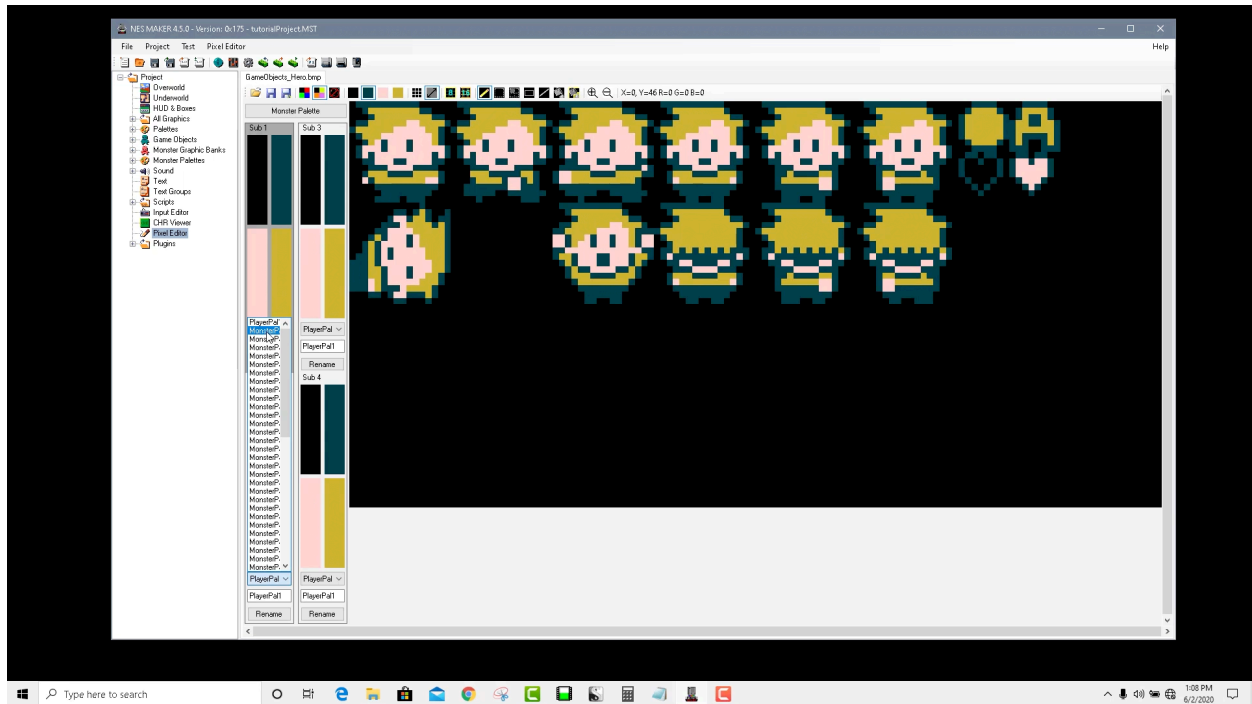
Step 3: Set some logical colors for our hero in the first object sub palette.



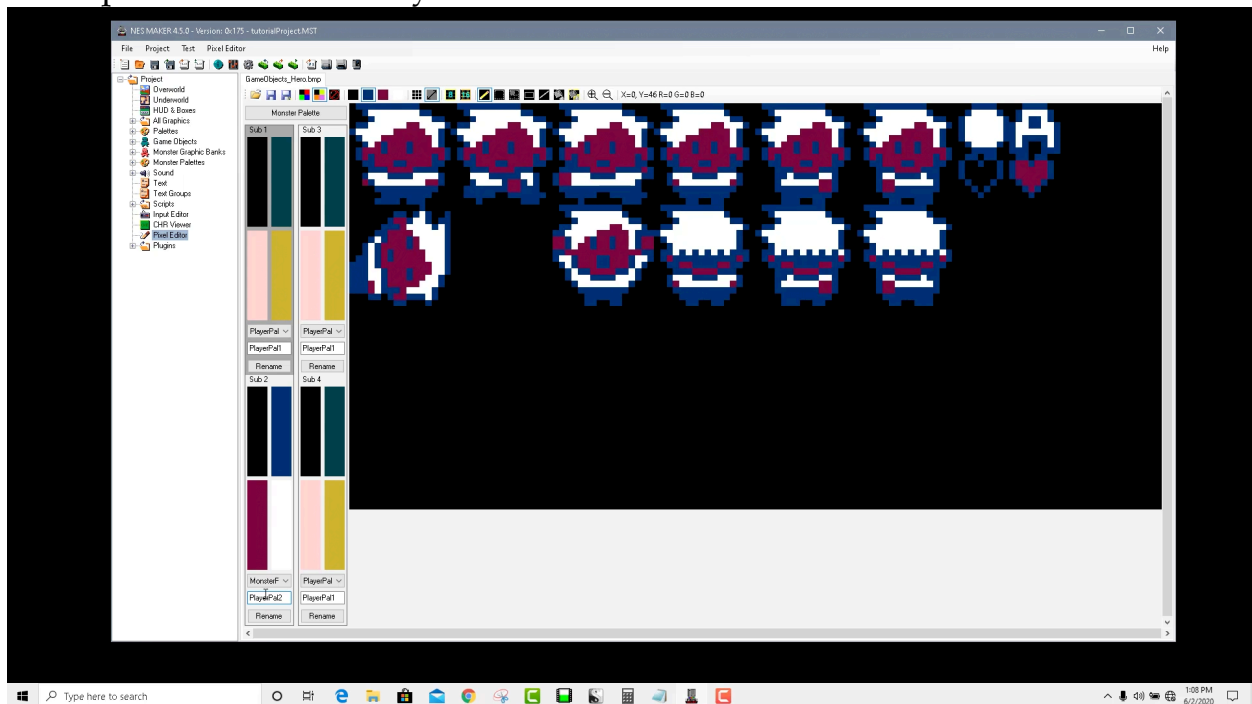
Step 4: In the text field underneath the first object sub palette, enter PlayerPal1 and hit Rename.



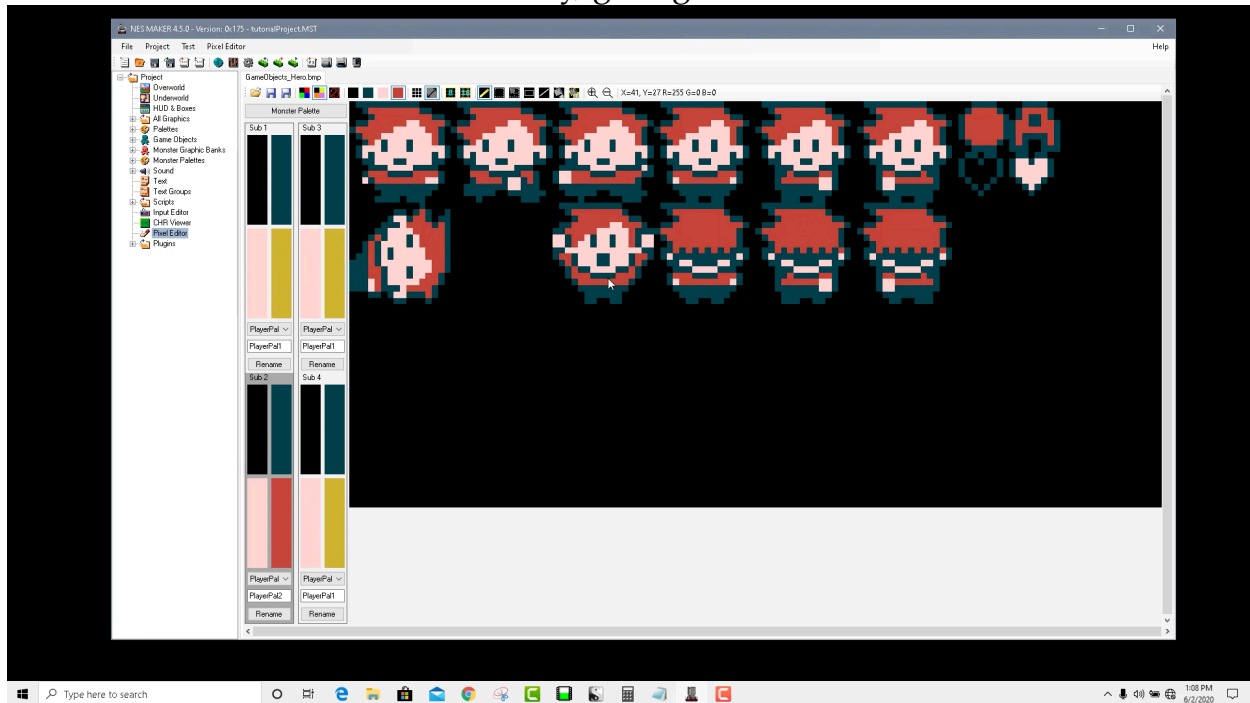
Step 5: In the palette that says Sub 2, choose a new palette from the dropdown menu.



Step 6: Rename this Player Pal 2



Step 7: Double the outline color and the skin color, but change the last color to an orangish red. We are going to use the top sub palette for the top half of the character's body, giving him blond hair, while using the second sub palette for the lower half of the character's body, giving him a reddish shirt.

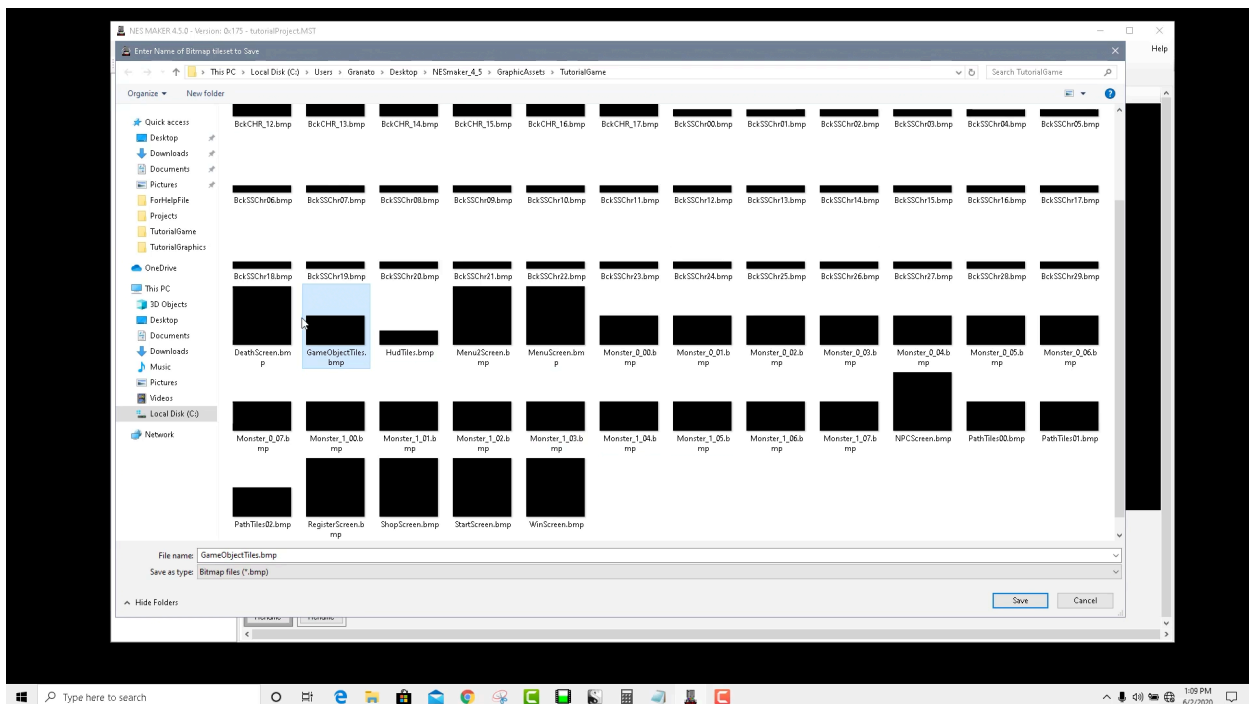
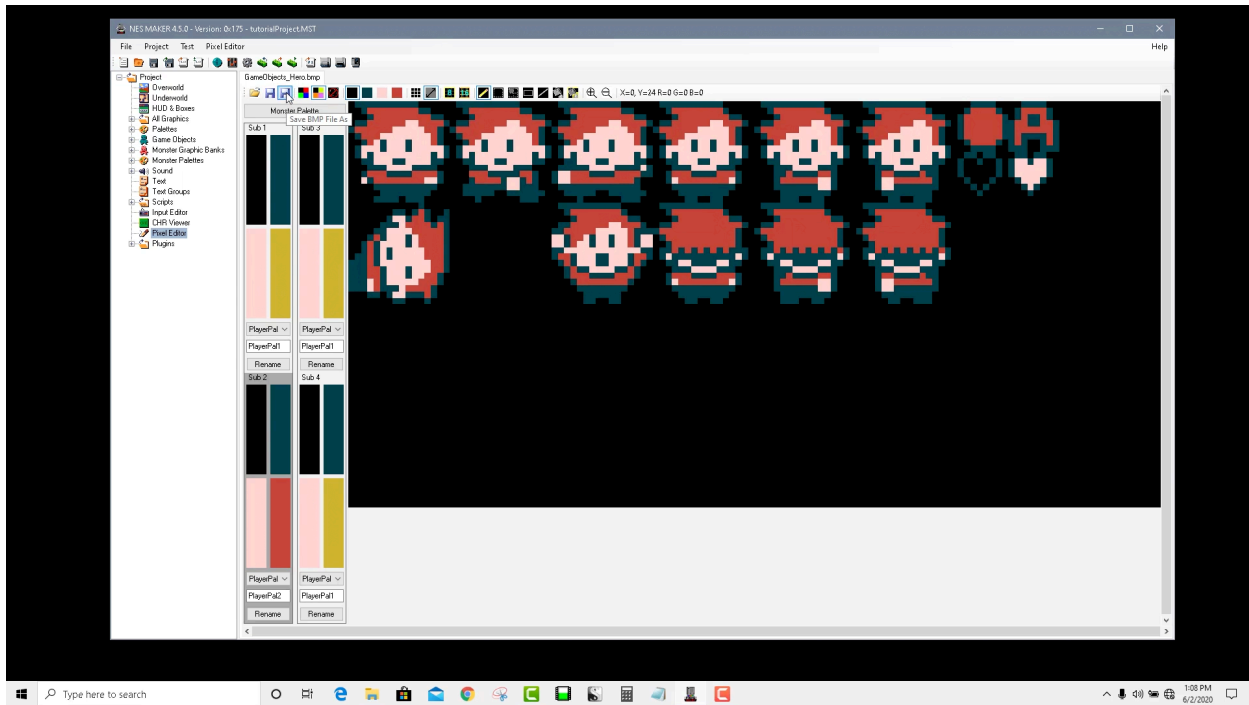


So far, this all has really just been an exercise in setting up object palettes. We haven't made a player object yet or even created anything.

Step 8: Go to Save BMP File As. This BMP file is already the size and shape of what is expected for a GameObject tileset. So in the explorer window that pops up, we can scroll to GameObjectTiles.bmp and double click to overwrite the blank file that currently exists with our hero graphics. It will ask if you want to replace this file. Hit yes.

Important to note, by default with these modules, tilesets are of an expected size. Saving the wrong size graphic file over the top of one of the template files may cause an error. Sometimes it's best to do what we did with the background tilesets. To be sure, you can open the empty tileset in a tab, open the source of graphics you want to add in another, copy and paste into the existing tileset, and

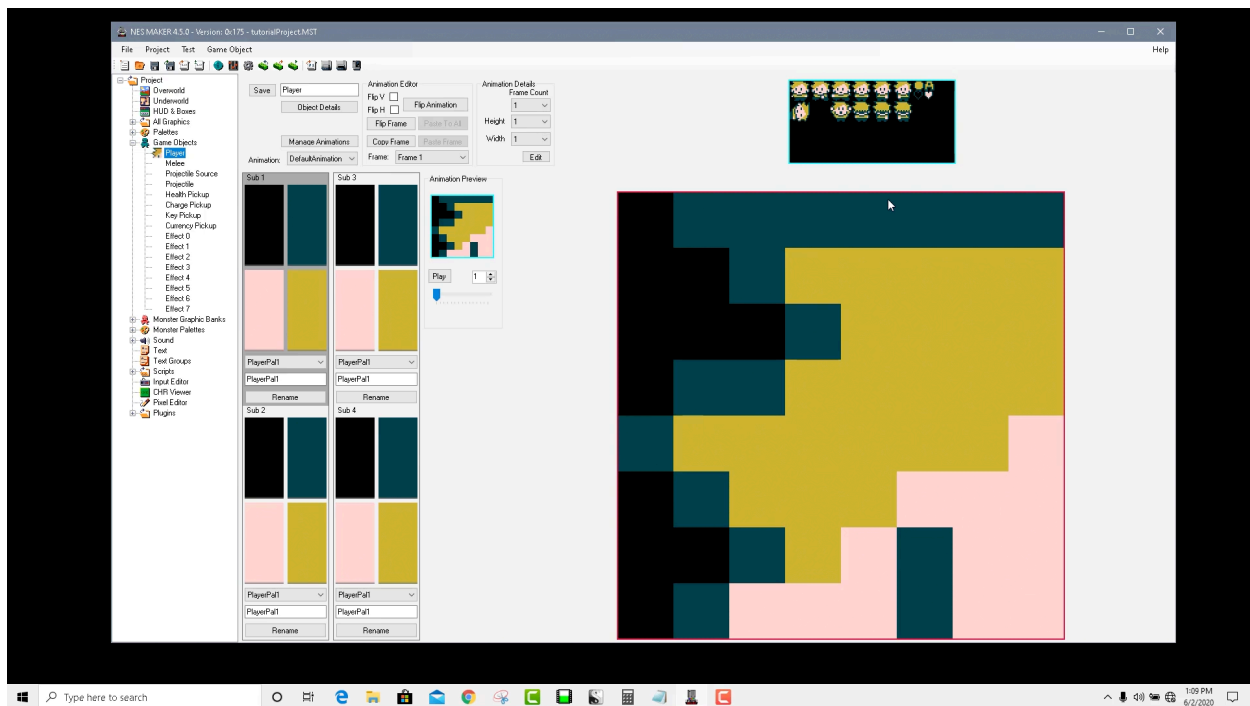
hitting save. This method will ensure that the file is the right size.



Step 9: Now that we have the graphics for our player game object, expand the GameObjects node in the hierarchy and click on the first one called Player. Keep

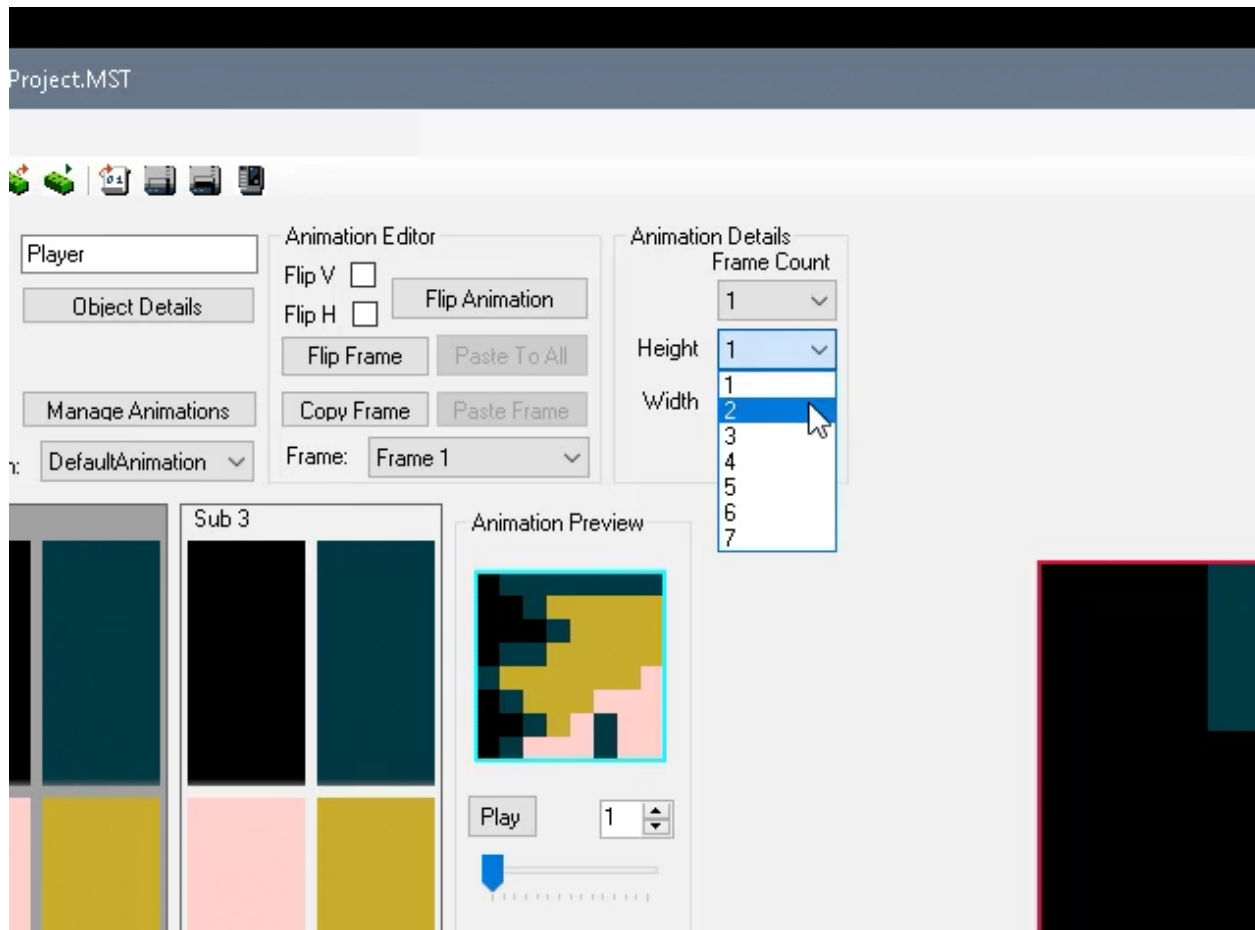
in mind, like many things in this software, these references are just names and as of right now, don't have any particular attributes that make them behave like their label. They may as well all be labeled with generic names, like GameObject0 - GameObject15. They're named by default to give an implicit understanding of the difference between Game Objects and Monsters. Game Objects are objects whose graphics are ever-present throughout a game, whereas monster graphics may change from screen to screen. In most cases, your melee weapon or your key item or your health pickup will look the same no matter what screen that you're on, and they could appear anywhere that you might also find the player. This is what makes them good for the Game Object classification. Also, second players or ubiquitous traps or magical effects that may show up all over the game may make good Game Objects. The important thing is not to get caught up on the names of these Game Objects. As of now, they have no inherent behaviors or mechanics, and all 16 of them can be used to be whatever we need them to be for our particular game.

The Game Object Interface

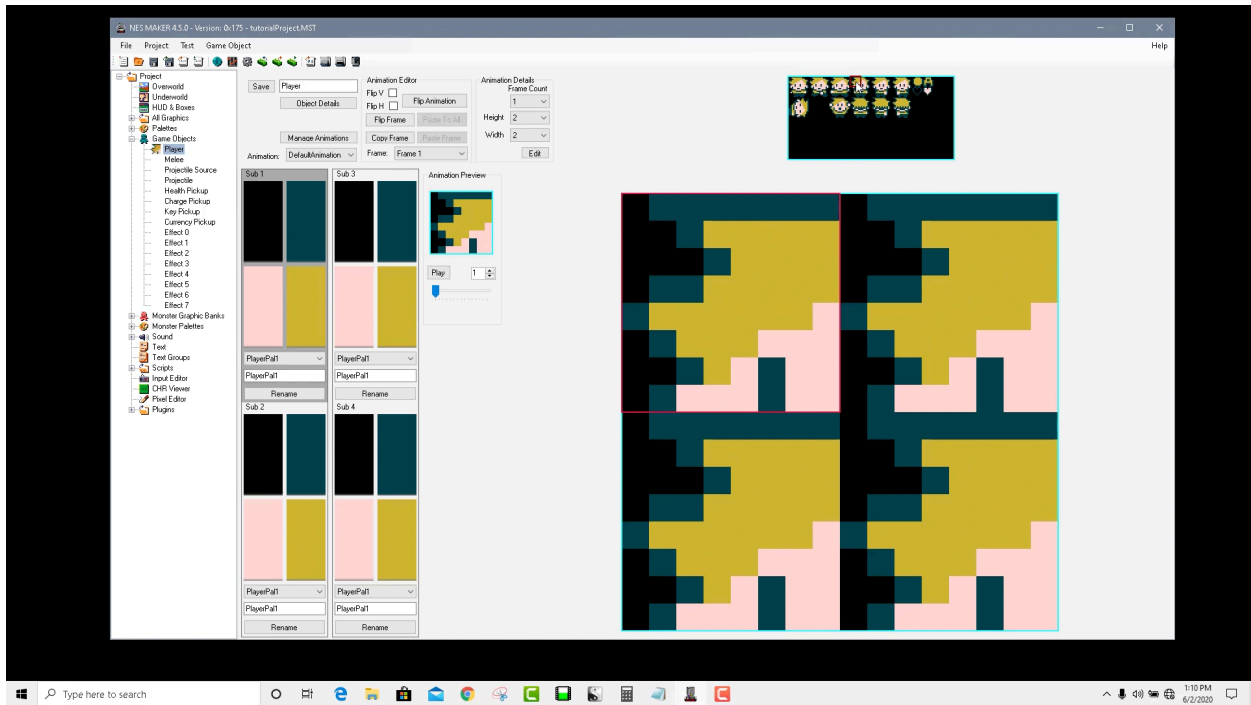


There is a lot involved with the Game Object interface and its submenus. The easiest way to understand it all is go through making an object step by step. What's great is learning how to build our player is almost exactly how we'll build our other objects such as monsters, power ups, and weapons.

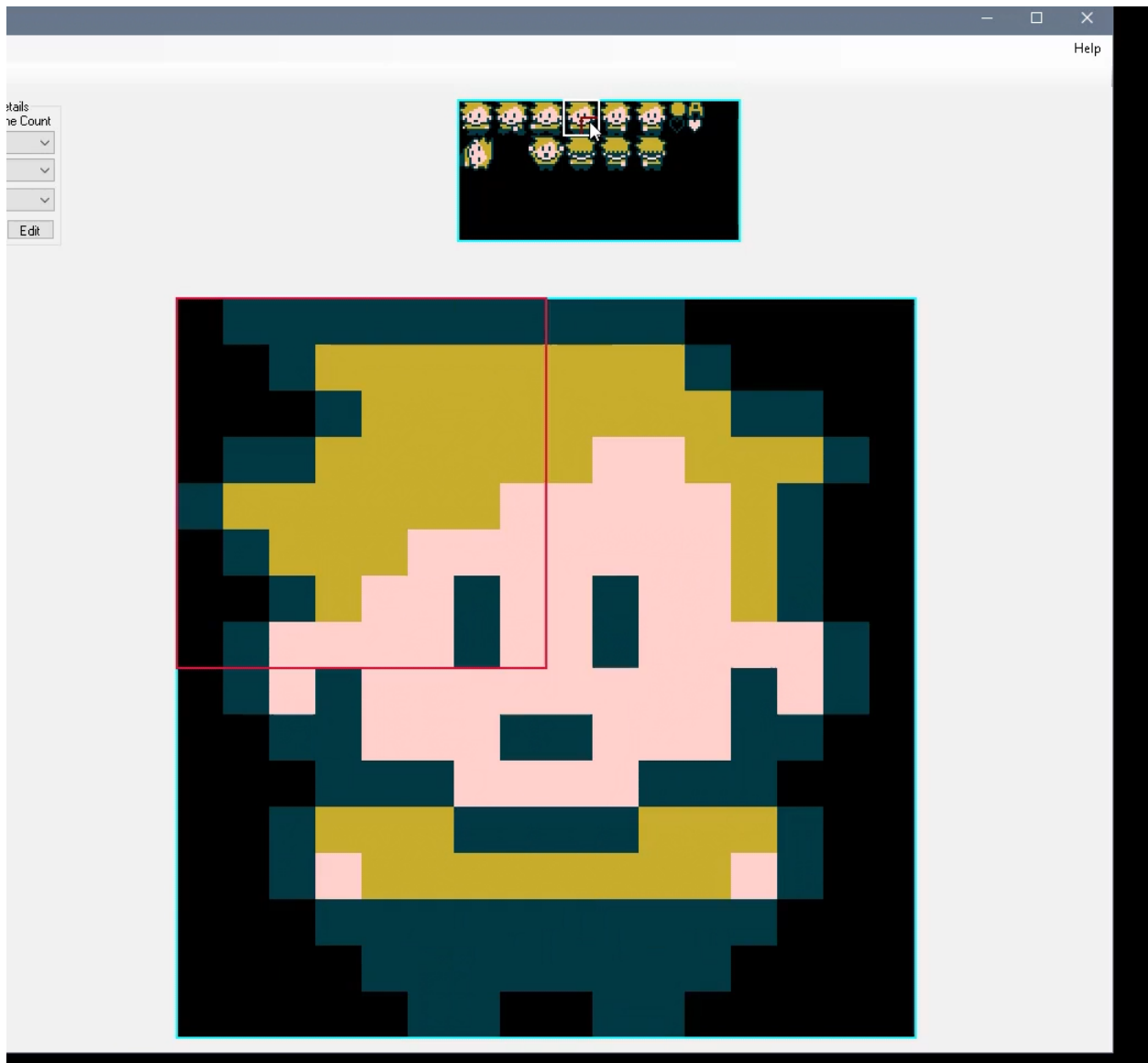
Step 1: Adjust the size of your object. Our player is not a one x one tile character, he is two x two tiles. So from a dropdown, we can change both the height and width to 2. When designing your own characters, it is recommended that you keep your NES characters relatively small (especially in the horizontal direction), as the NES hardware has restrictions as to how many sprite tiles it can draw at a time on screen at one time (64) and how many it can draw in a horizontal row at one time (8). This is the reason many games had flicker. Once 8 sprite tiles are drawn in a horizontal row, no more will appear. However, some games used tricks to vary the order as to which tile gets drawn first, resulting in sprite flickering.




Step 2: Click on the top left tile. You'll know it is selected because it will have a red outline around it. With it selected, choose the tile from the tileset that you'd like to place in that position. We are going to create the character facing forward for a default pose, so choose the top left tile of the fourth pose in the tileset for this.



Step 3: If you hold the sift key and drag across the entire pose, it will place all four tiles in their proper places.



Step 4: Change sub palette 2 to PlayerPal 2.

- Game Objects
-  Player
- Melee
- Projectile Source
- Projectile
- Health Pickup
- Charge Pickup
- Key Pickup
- Currency Pickup
- Effect 0
- Effect 1
- Effect 2
- Effect 3
- Effect 4
- Effect 5
- Effect 6
- Effect 7
- Monster Graphic Banks
- Monster Palettes
- Sound
- Text
- Text Groups
- Scripts
- Input Editor
- CHR Viewer
- Pixel Editor
- Plugins

Manage Animations
Copy Frame
Paste Frame

Width 2

Animation: DefaultAnimation
Frame: Frame 1

Sub 1

PlayerPal1
▼

PlayerPal1

Rename

Sub 3

PlayerPal1
▼

PlayerPal1

Rename

Sub 2

PlayerPal2
▼

PlayerPal2

Rename


Sub 4

PlayerPal1
▼

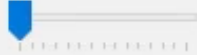
PlayerPal1

Rename

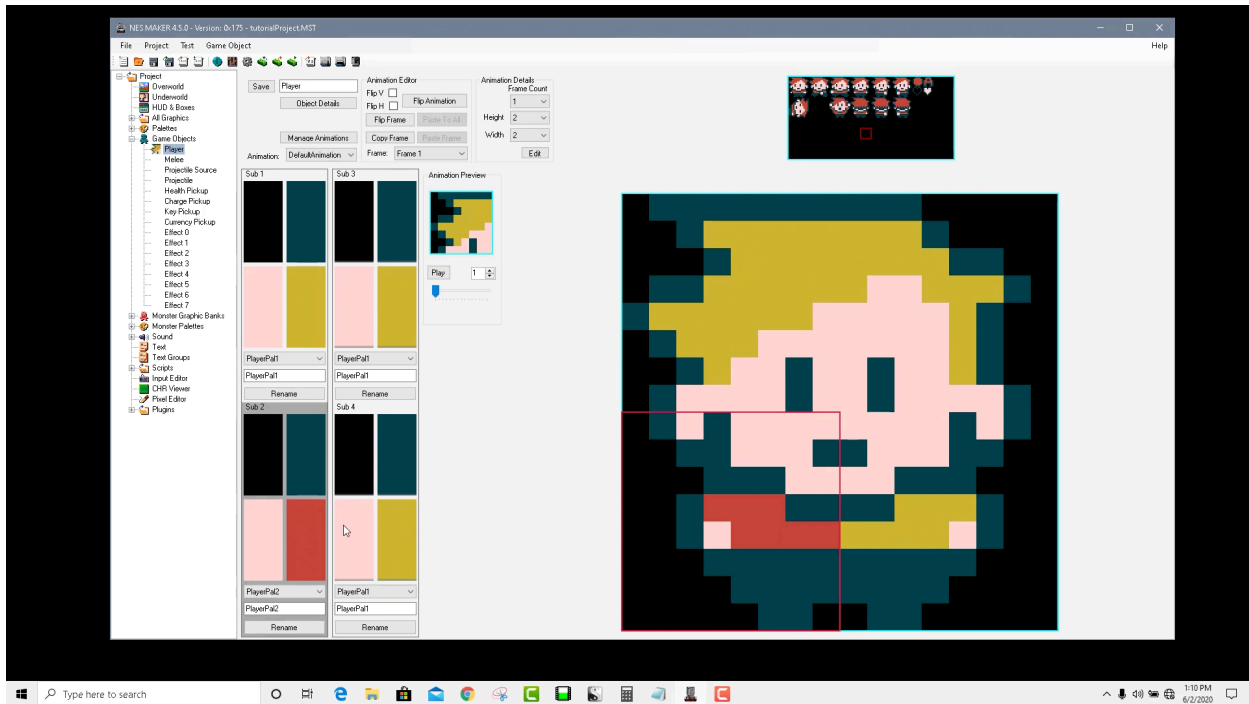
Animation Preview



Play
1



Step 5: Click on the tile you want to change in the canvas. You'll know it's selected because it has a red rectangle around it. With it selected, click anywhere on the sub palette 2, and you'll see the colors of the shirt change. Do the same with the bottom right tile.

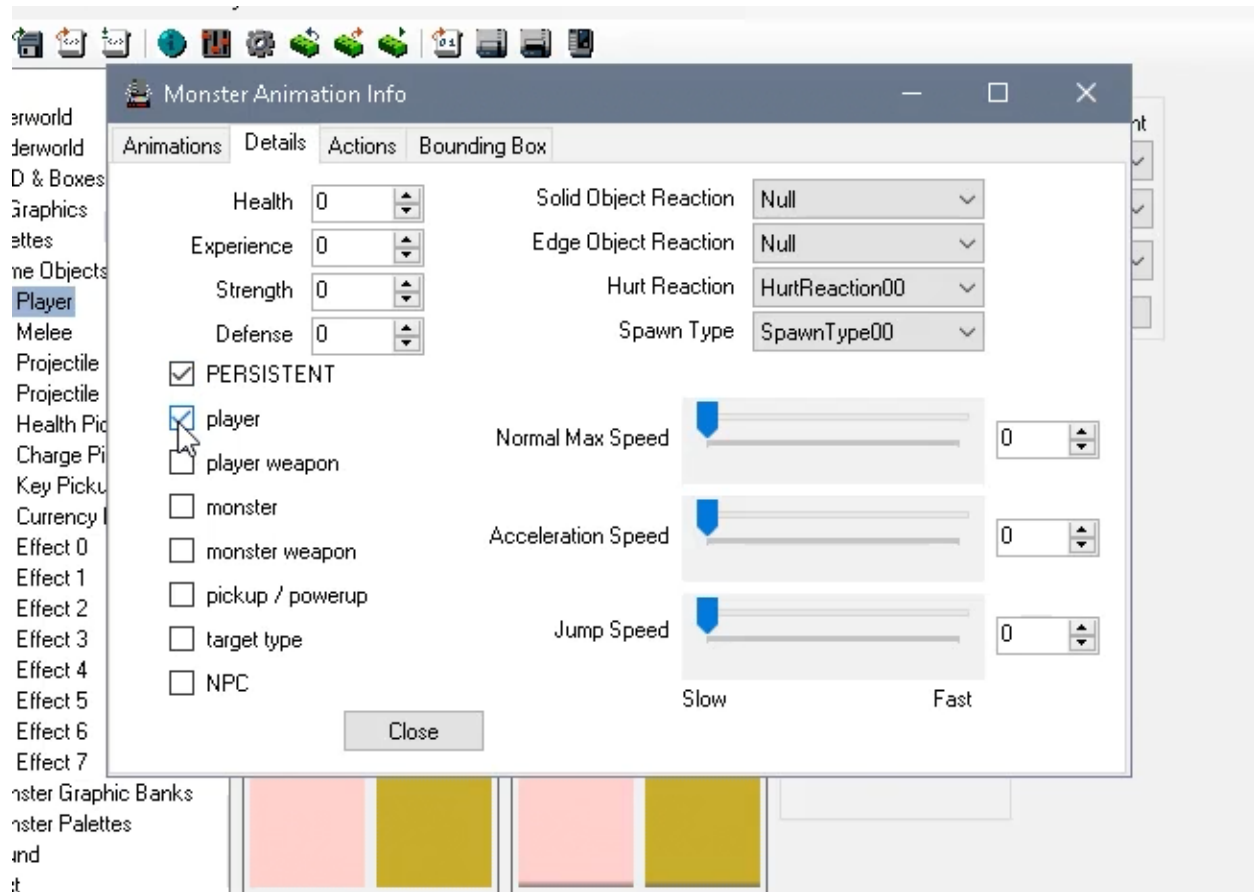


Again, as repeated throughout working with graphics, this does not necessarily denote what color our player will be on a screen. The pixel editor and this animation tool do not control the color of the object. They control the palette reference. What is being exported by this tool is the fact that the top to tiles will use subpalette 1, whatever happens to be loaded into that slot at the time, and the bottom tiles will use subpalette 2, whatever happens to be loaded into that slot at the time. If, on a screen, we load PlayerPal1 into slot 1 and PlayerPal2 into slot 2, the player will look the way he does here. If, on the other hand, we load PlayerPal2 into slot 1 and PlayerPal1 into slot 2, our Greg hero will have fiery red hair and a gold shirt instead. If both slots are loaded with PlayerPal1, which would be the default since that is the null value, then he will have a gold hair and a gold shirt.

In short, this interface does not control the colors of the objects, only which

sub palette will be referenced. The screens themselves control what those sub palettes actually are. We will take a look at that momentarily.

Step 6: Click on the Object Details button and click on the details tab. Ignore just about everything for now. But one thing we do need to worry about are the object bits Persistent and Player. Check both of those boxes.



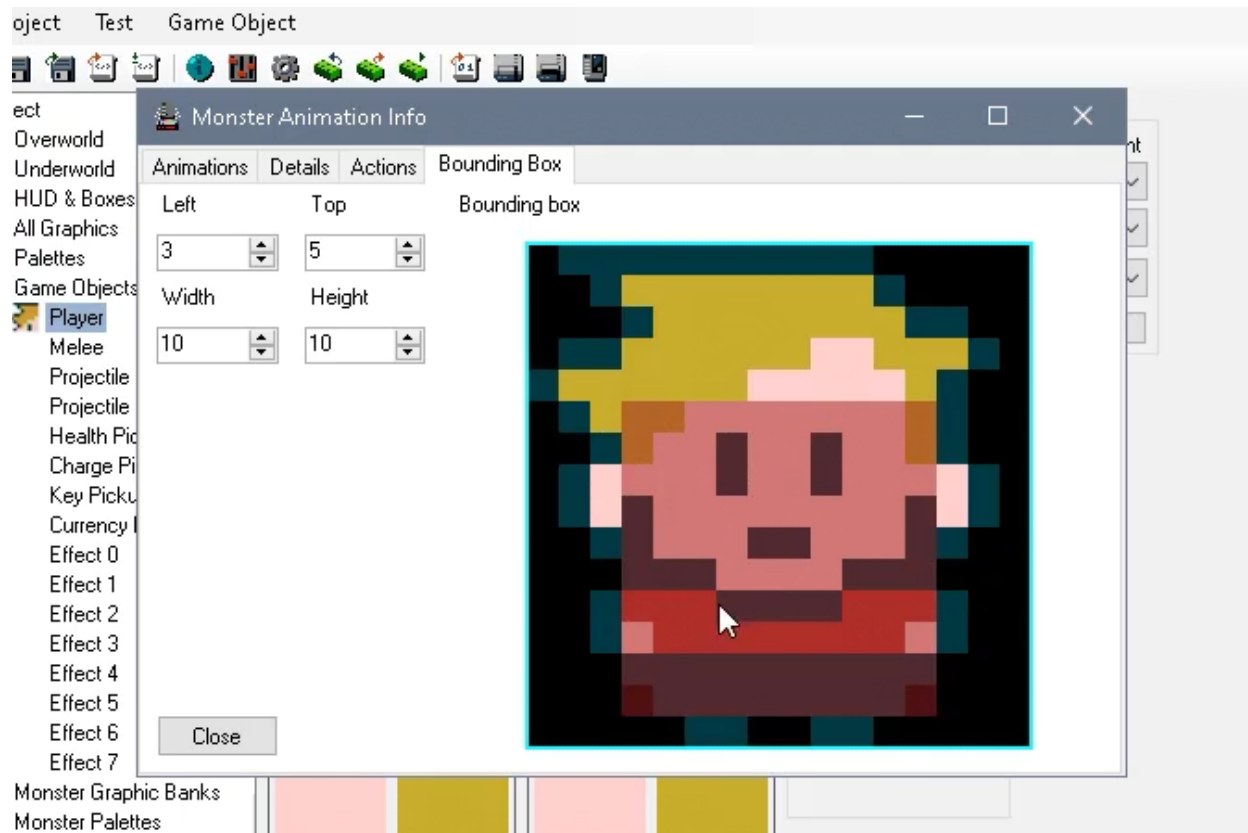
Setting an object to Persistent means that it won't be destroyed during a new screen load, that all of his information will stay in tact in memory and carry over into the next screen. Generally, there are slots designated in memory for objects. When a new screen is loaded, those slots are cleared and then filled with the new screen's object data. Any easy way to think about this is that on screen 1 you might have three bat monsters, while on screen 2 you might have one snake monster. The bat monster slots were cleared out when leaving the first screen. Their position, their hit points, what graphics those objects pointed to, the behaviors those objects called upon, everything about those objects gets cleared.

Then new information for the snakes fill the appropriate slot on the next screen. However, the player object maintains all that information as he transitions from one screen to the next. He persists over the screen change. This is why monsters would not be persistent, while the player would be. In most cases, only the player will be persistent. If you forget to make him persistent, the player will get cleared when the first screen loads, which is why this step is important, otherwise you'll never see him.

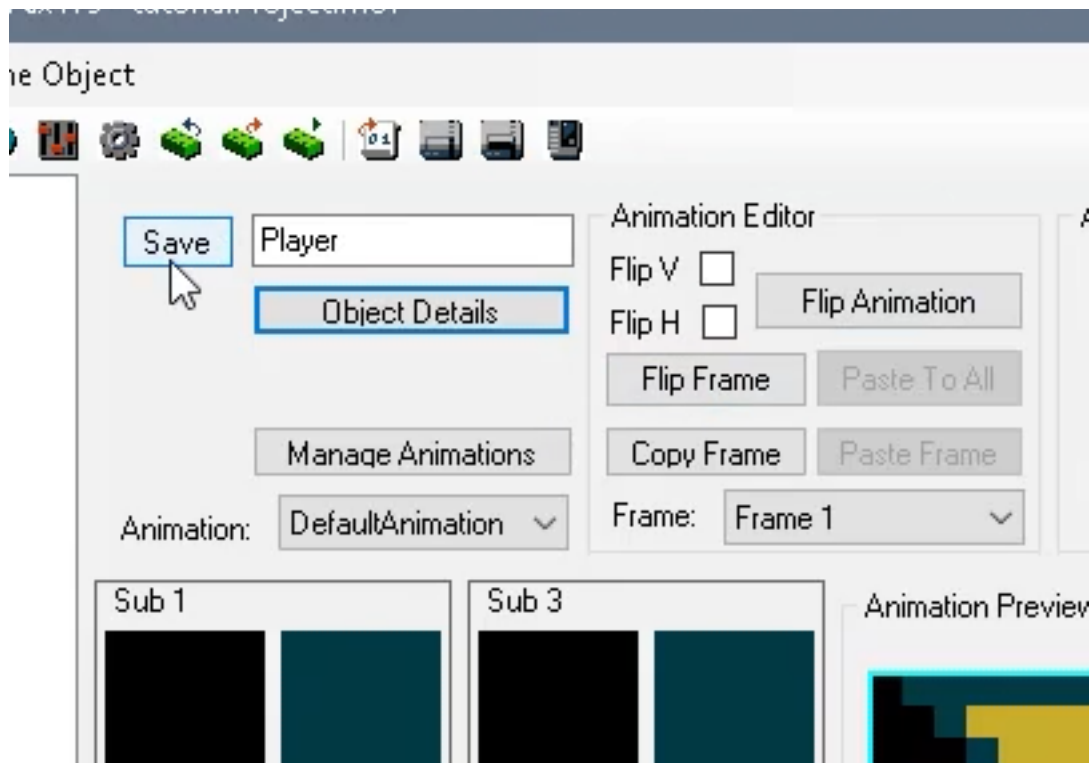
Clicking on the player checkbox will come into play later. It mostly has to do with how the NESmaker modules handle object to object collision.

Step 7: Go to the Bounding Box tab, and on the image of the player, click and drag a small rectangle over the center of the graphic. I usually do not use the entire rectangle for collision, for a few reasons. The most prominent, though, is if there is ever a one tile passage between solid objects and your player's bounding box takes up a full tile of space, the player has to be pixel perfect in his movement to fit through the space, which is often frustrating. Also, you'll notice that there are parts of the player object that have no pixels. This means that if one of those blank pixel spots brushed up against a monster, and the bounding box covered those blank pixels, it would register as a collision even though it visually looked like the two never overlapped. For this reason, I usually cheat just a bit for the player and make a box that looks something like below, but this is also wholly dependent on the type of game being created.

Once the bounding box is set up, hit the close button on this Object Details popup.

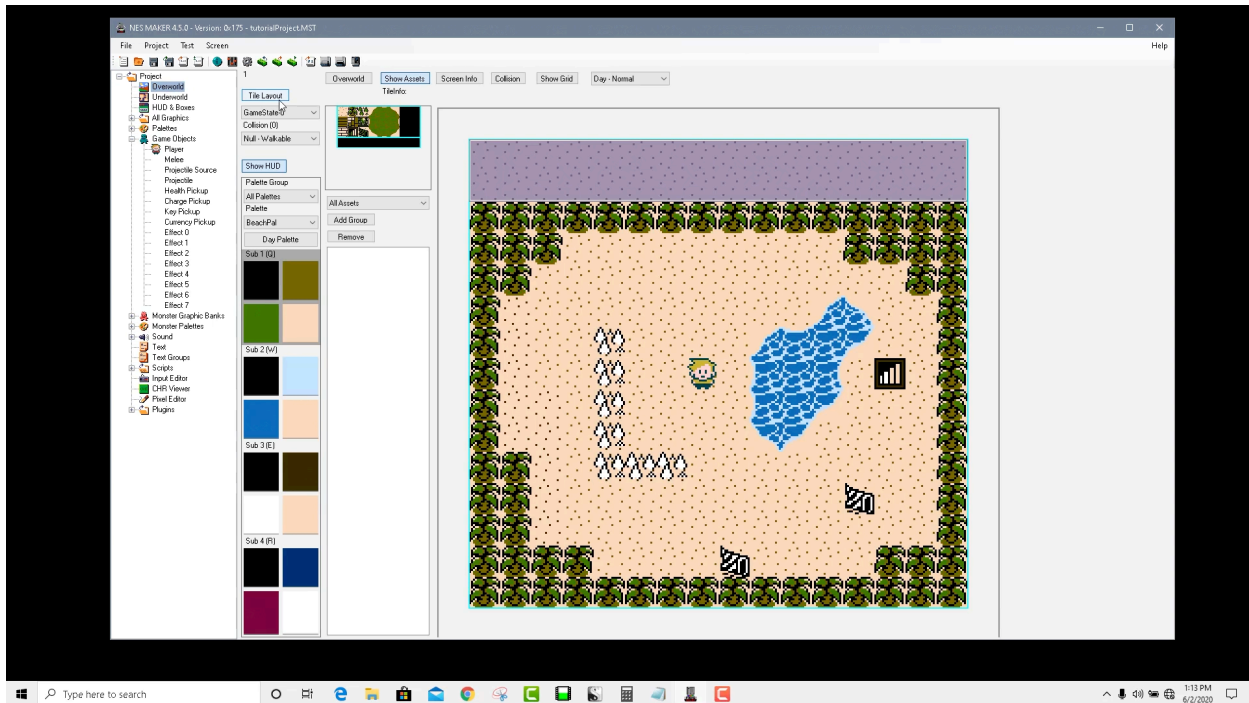


Step 8: Make sure that you hit save. When you do, you'll see that in your hierarchy, the icon of your Player object changes and now resembles a thumbnail of the character.



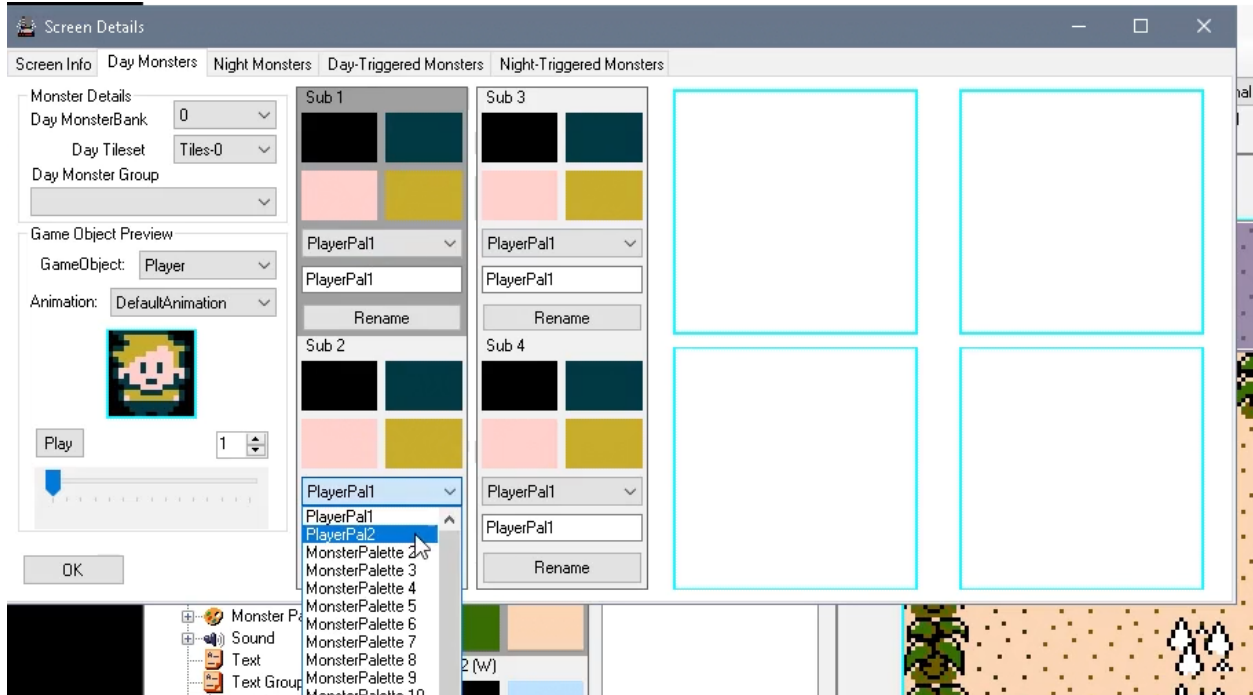
Step 9: Now, if you click on Overworld in the hierarchy and double click on the screen that you created, you will see that your player appears at the position you previously placed the player. Remember, he didn't appear before because he had null graphics, all transparent. But now, we've given him graphics, so that data was filled in.

If you want to move the character to a different position, simply right click where you would like to place him and select Place Player.



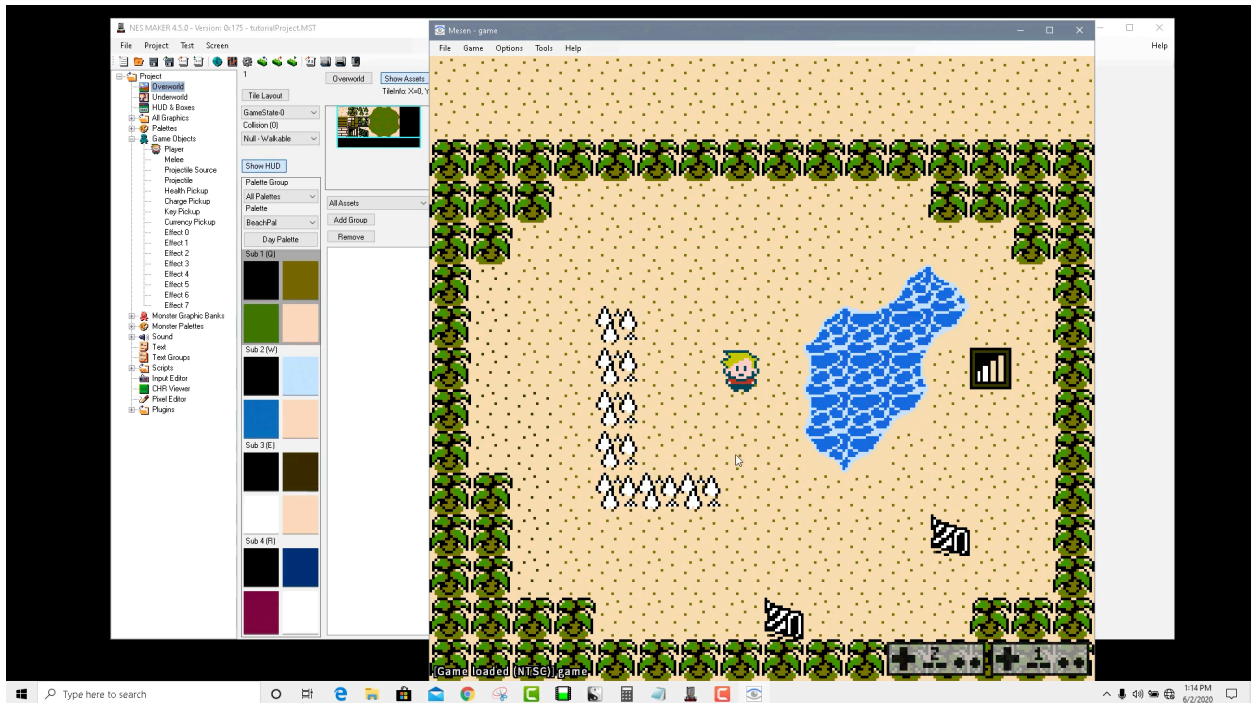
You'll notice, though, that his shirt is not red. As explained before, setting up your character is not what decides the color on the screen, the screen determines the color. The object setup just determines which object sub palettes are being used. Right now, the screen's object sub palettes are both null, or zero, and we know that that first Player Pal 1 is in the zero slot, which is why both top and bottom show that color scheme. The next thing to do is to set up the object palettes for the screen.

Step 10: Click on Screen Info and choose the Day Monsters tab from the pop up dialog. There, you will see the current sub palettes for the game when it is in its normal state. For now, don't worry too much about night or triggered modes, as they are for advanced use. All we have to do is focus on getting the palettes here to match what we expect to see which is Player Pal 1 in the sub 1 slot, and Player Pal 2 in the sub 2 slot. So choose Player Pal 2 from the drop down menu in the Sub 2 slot.



Hit ok, and on the screen painter, your player will now show the appropriate colors.

Step 11: Test your game by going to Export and Test from the NESmaker menubar. You'll see the game open up in the emulator, and now your player object will be in the middle of the screen.



Notice that still, nothing happens. We have not told this object to receive input, we have not told it to animate, we have not told it to interact with anything. But we have placed an object on our screen, and confirmed that it runs in an emulator.

Designing Input

Designing Input

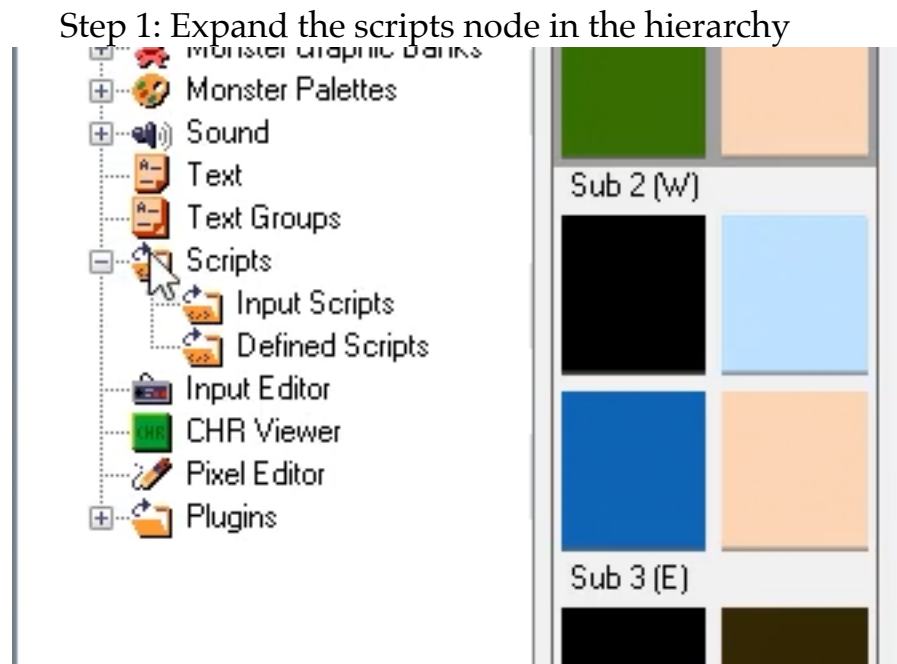
Before we begin designing input, it might be a good idea to revisit our design document. In that document, we determined that we want the player to be able to move in four directions when we press the d-pad buttons, to start the game when we press start, and to shoot when we press the b button. That's a user manual level explanation of things, but we might have to go a bit deeper to get it to function the way we want.

Let's take a single button, the left d-pad button. We press the left d-pad button, the character moves left. Ok, what happens when we release the button?

Does he continue moving left until he runs into an obstacle like PacMan? Does he stop moving? Is there any deceleration time or trot-after frames? Does he turn to face that direction, and does he animate? What happens when he hits the edge of the screen? An obstacle? If a vertical and horizontal button are simultaneously pressed, does he move only in the direction of the last button that was pressed, or does he then move diagonal? And if diagonal, does he move in both directions at full speed (which would make him move faster in diagonal direction than in horizontal and vertical directions) or do we have to account for that in our movement speed?

When designing our simple idea, we glossed over some of these particulars. That's ok, that's why design documents are usually considered living documents, and why they change and are amended during the production process. But those are just a few questions we have to figure out answers to before we set out to design our input scheme.

But let's start simple. Here's how to add input scripts and apply them to the controller.



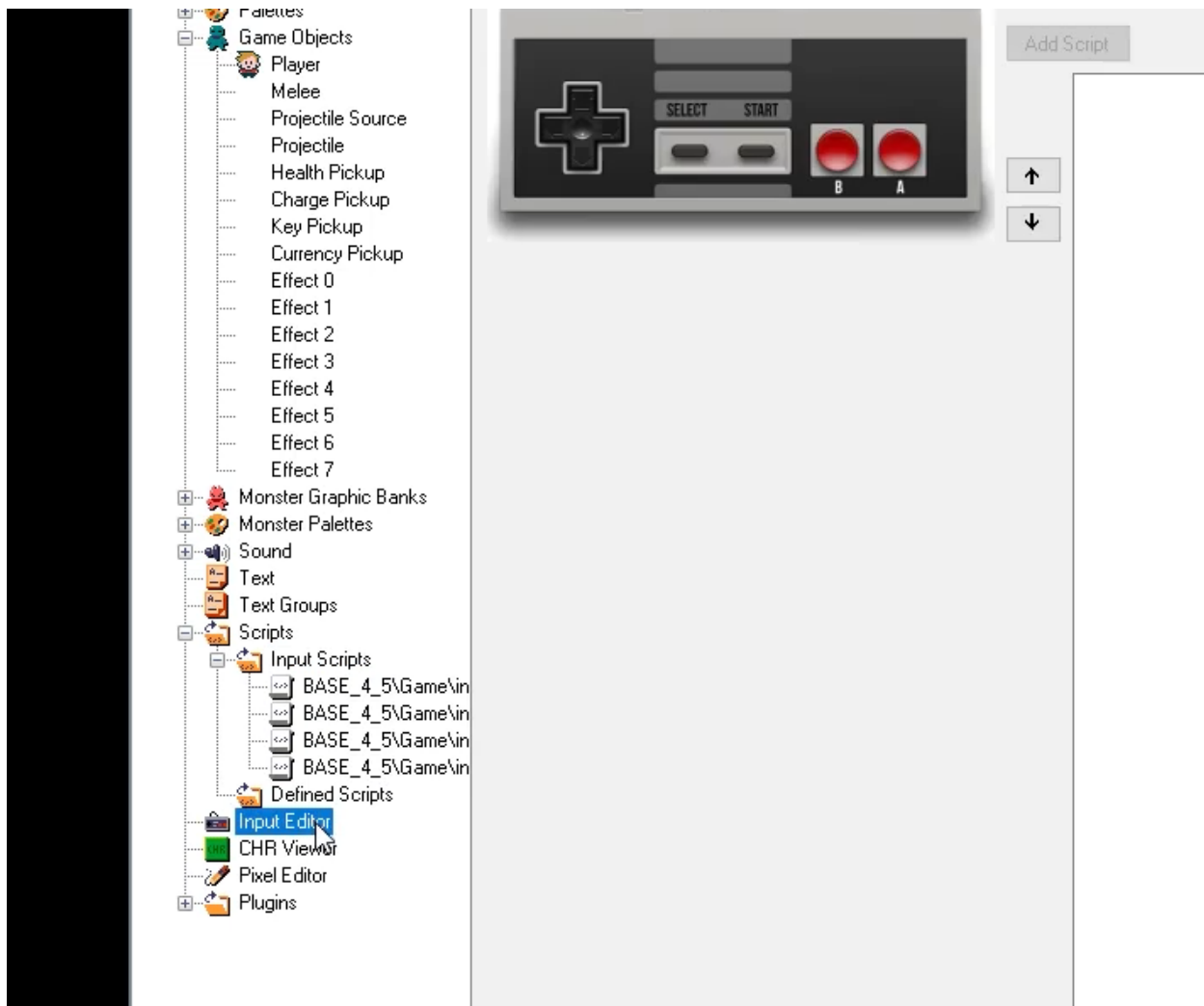
You can see that there are two nodes that appear, one for Input Scripts and one for Defined Scripts. For right now, we just want to worry about Input Scripts.

Input scripts are handled a bit differently than other defined scripts, which we'll talk about later. Click on Input Scripts.

Step 2: In the finder window that shows up in your work area, navigate by double clicking to your root folder /Game / Input Scripts. There, you will see all of the default input scripts that are bundled with this tutorial set. Keep in mind, you can add others, easily edit these that do exist, or even write your own input scripts if you want. These are just part of the template.

Step 3: To add the move scripts to your project, double click on Move Down, Move Left, Move Right, and Move Up. You'll see them disappear from the finder window, and you'll notice that the Input Scripts node in your hierarchy now can be expanded. The input scripts that you are adding get placed into the Input Scripts node, and will now be available to the Input Designer.

Step 4: click on Input Editor in the hierarchy.



The input editor gives you full GUI control of how to activate your input scripts.

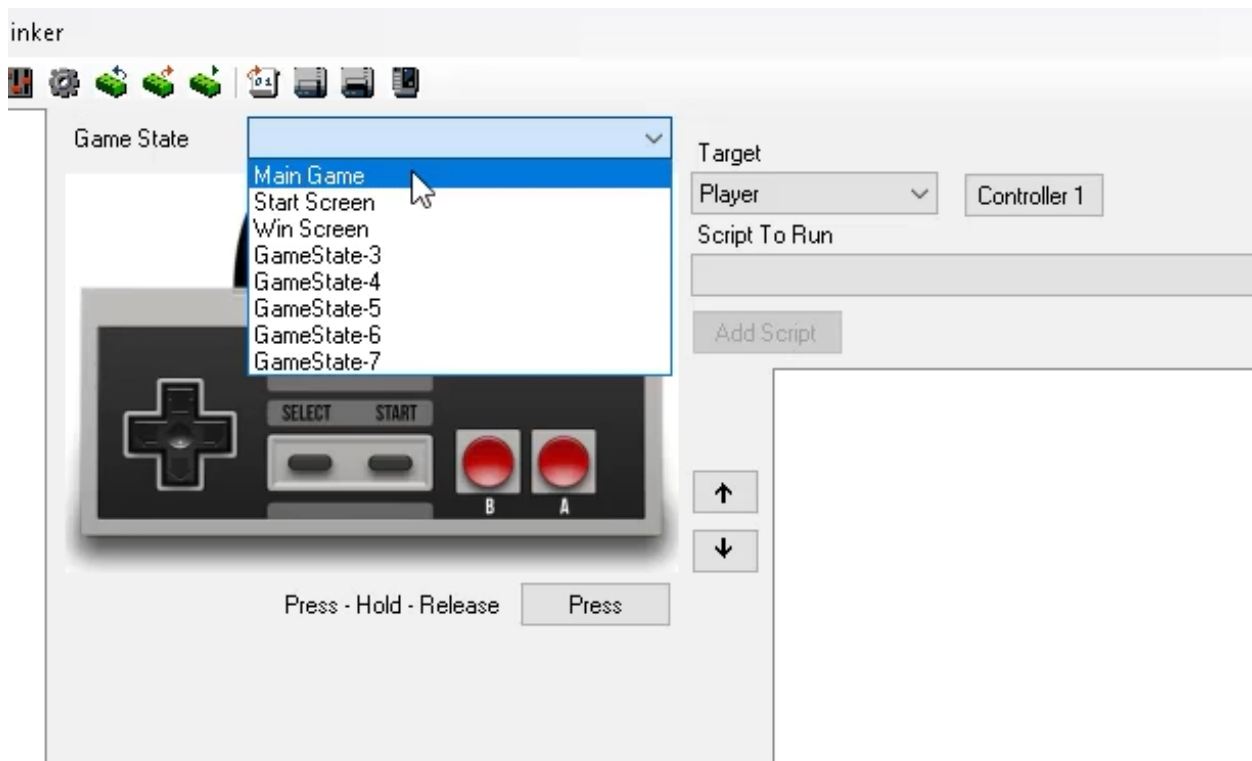
In order to set up an Input to function in our game, we need to:

- Determine what game state should invoke this. If we are in a different game state, it will ignore the controls. Remember, we want the arrows to move our player in the main game state, and the start button to start the game in the start screen state. We don't want these inputs to have the same effect when in other states. So the first important thing to consider is what game state we're working in.

- What target object, if any, will this input effect. The arrow keys definitely should effect the player game object. If we had a two player game, we might have the second controller affect the second game object.

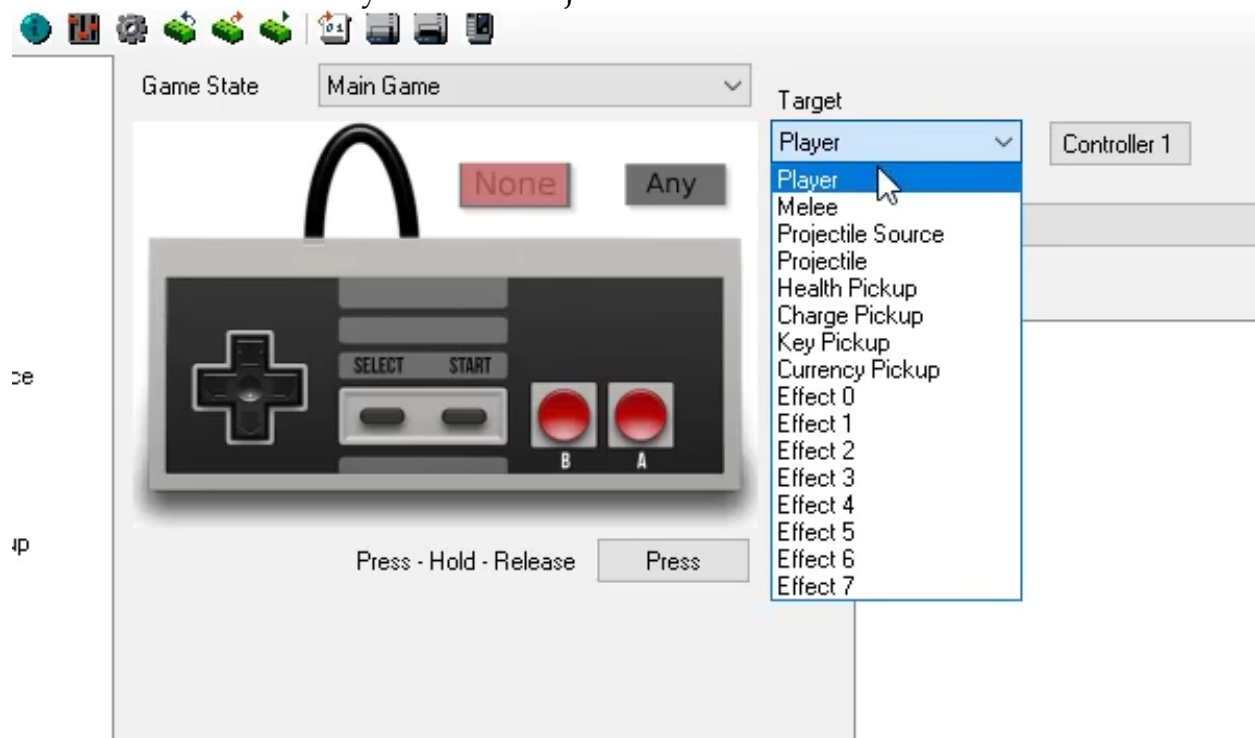
- What controller, one or two, are being read to call this script
- What buttons do will activate this script.
- What button condition will call this script; when the button is first pressed, when it is being held, or when it is suddenly released.
- And then finally, when those conditions are met, what script should be called.

Step 5: Choose Main Game from the Game State drop down. Here's where those labels we created at the beginning come into play. You can see that the three game states we set up for our game are present here. Again, this means nothing to the actual game. All the software knows is that it's reading a variable called game state and checking to see if its value is 0-7. But those labels we created in the tool make it much easier for us to keep track of here while we're building our game.



So now, whatever input information we're about to add will only happen if our screen type is Main Game type (or, type 0). We'll look at where to set a screen's info in just a moment.

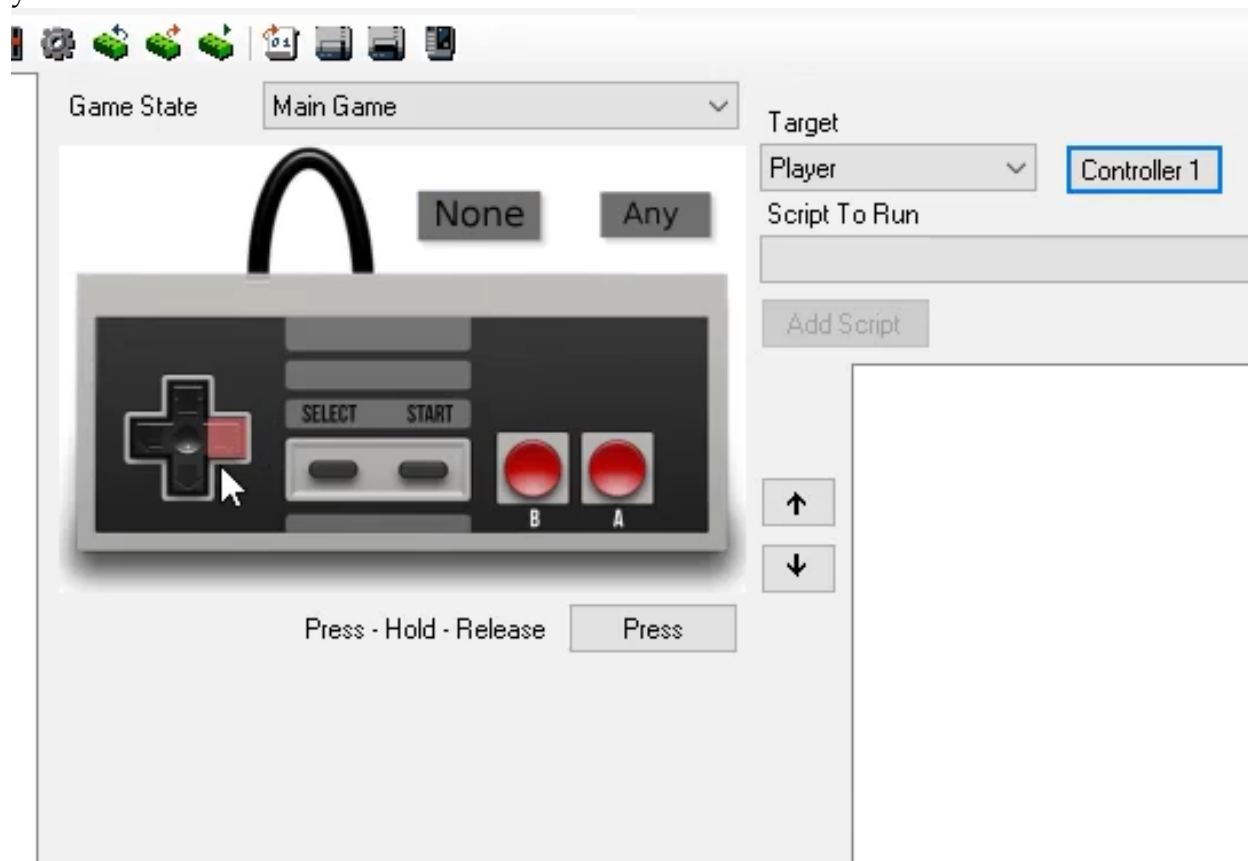
Step 6: To what game object should our control apply? We already have Player selected, and that's what we want for this input. We want the script that we'll invoke to affect the Player Game Object.



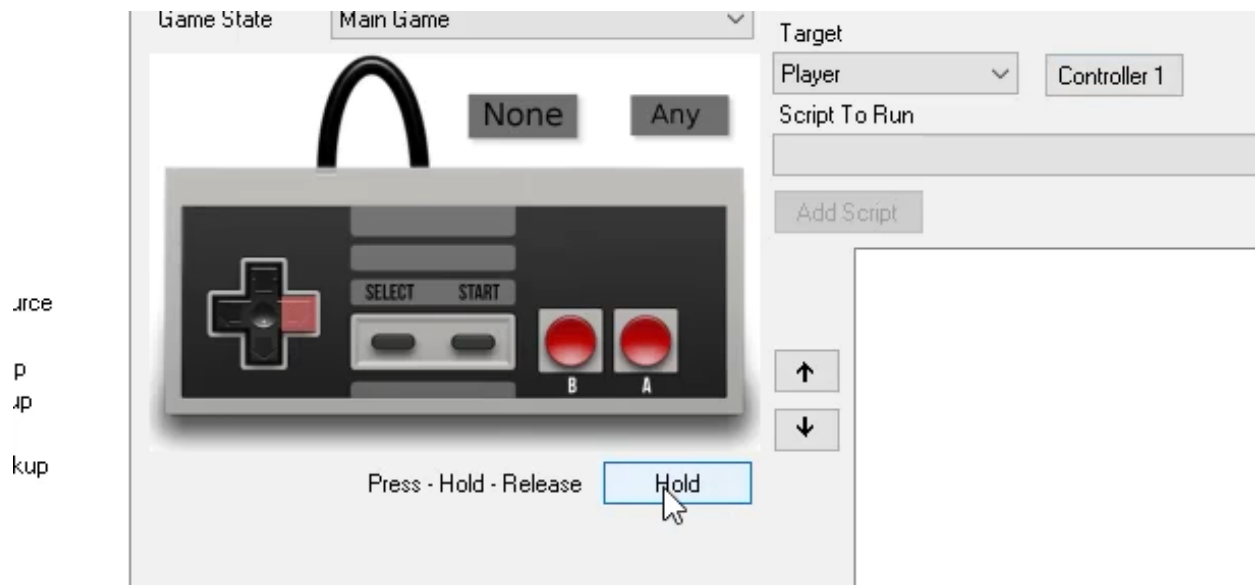
Step 7: What controller are we reading to check to see if we should carry out this input? It is already set to Controller 1. Pressing this button will toggle to Controller 2, but we want to stay on Controller 1 for now. You don't have to click anything, it is already set to Controller 1.



Step 8: Click on the Right D Pad Button. You will see that it turns red to let you know it is selected.



Step 9: What we want is to run this movement script if the right arrow key is being held down. Right now, there is a small state button that says PRESS. This would make it so that you have to PRESS the button to activate the script. This works great for something like jumping or shooting, because for those sorts of actions, you want the player to have to release the button in order to carry out the intended action again. However for movement, most games will continue to call the movement script as long as the button is being held. So to change PRESS to HOLD, click the state button underneath the controller until it says HOLD.

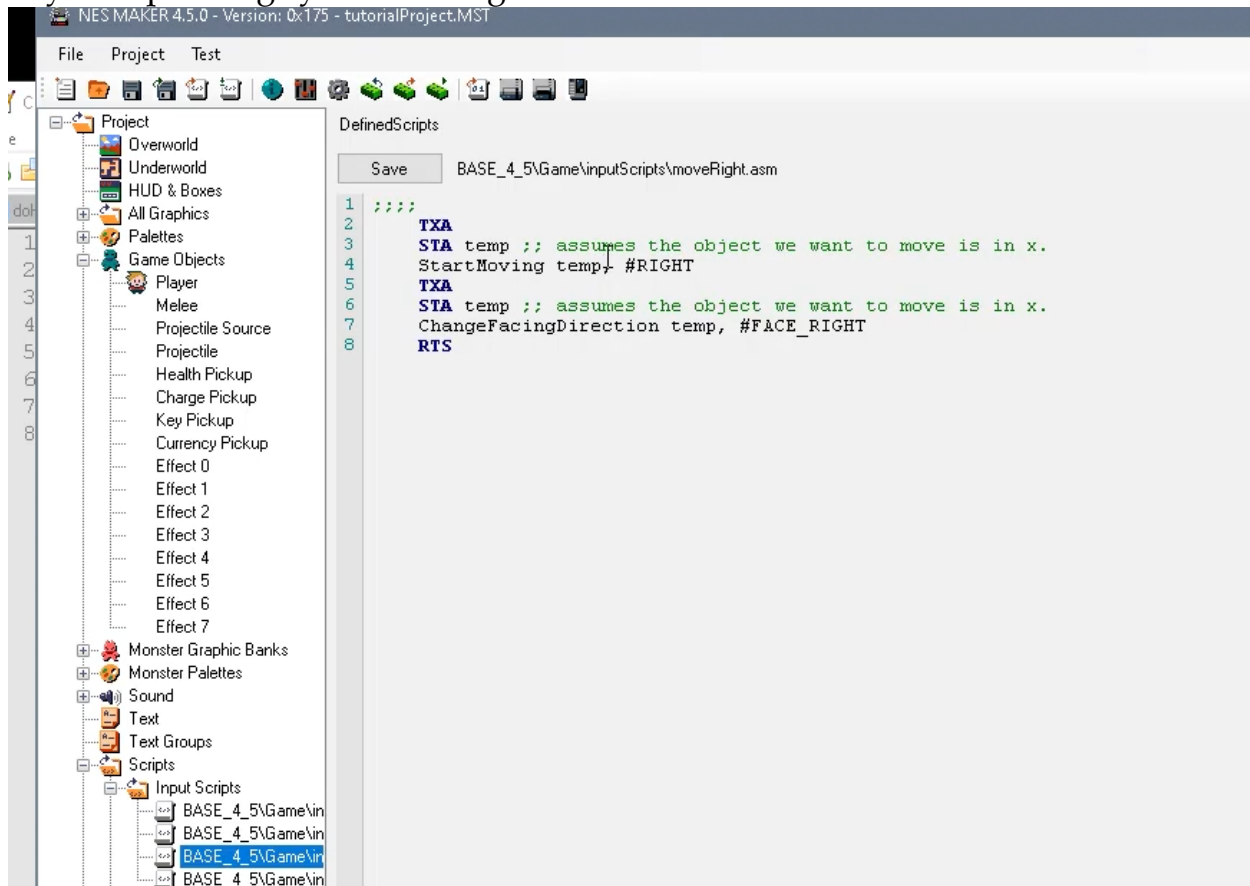


Step 10: Now you can choose your script, which will happen if all of the current conditions are met. From the dropdown, you'll see that the scripts we added to the Input Scripts are exposed here. Pick moveRight. With it selected, click on Add Script.

Now you can read the input editor left to right like a sentence. With controller 1, in the Main Game state, if I hold the right dpad button, the player object will call the moveRight script.

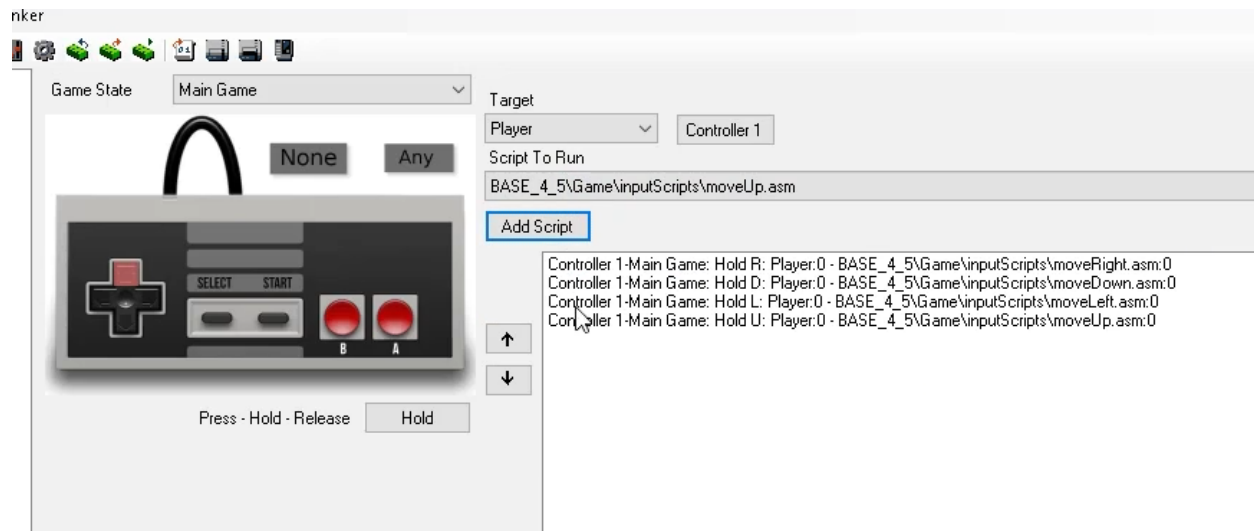
Without diving into a discussion about 6502 ASM code, it is great to understand what is happening here so we can get an appreciation for the flexibility. What is the moveRight script? What does it actually do? If wanted to

view the script, we could click on it in our hierarchy. Clicking on it once will open it in the built in spot-check coding editor. Double-clicking on it will open it in your operating system's designated text editor.

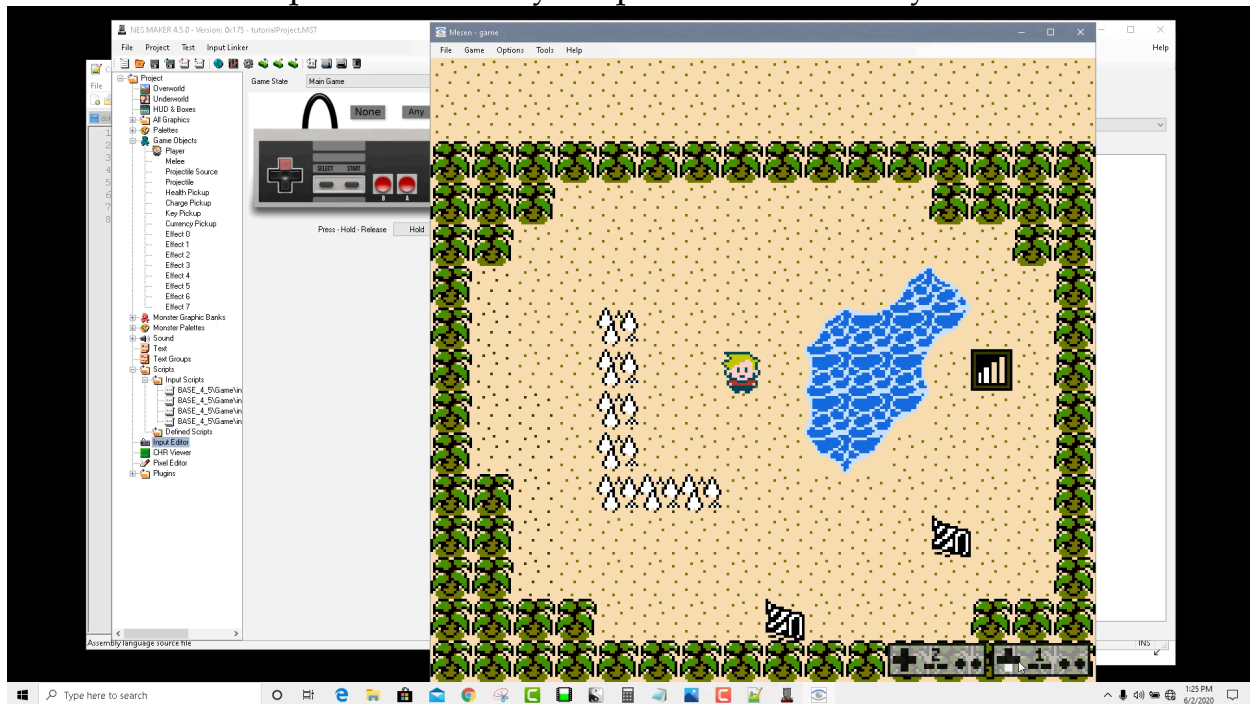


This is what the moveRight script is actually doing. And while we don't even really want to think about code at this point, it's great to know that you can get in and tweak and edit exactly what these scripts do and tailor them specifically to your game's needs. Rather than coding from scratch and trying to find the part of the script for right movement of the player during the main game's controller press amidst 100,000 lines of 6502 ASM code, that small, manageable chunk is right here, linked in a logical place and easy to tweak.

Step 11: Return to the Input Editor and repeat the process to add all four directions. They should all be in the Main Game state, they should all be for controller 1, they should all use your player as the target, and they should all activate when holding their respective d-pad button.



We are going to test the game, but full warning, the player will not move. We have done everything right, the inputs are set up, the proper scripts are called at the right time, and yet the player will still remain stationary. We'll take care of that in the next step. There is a very simple reason as to why.



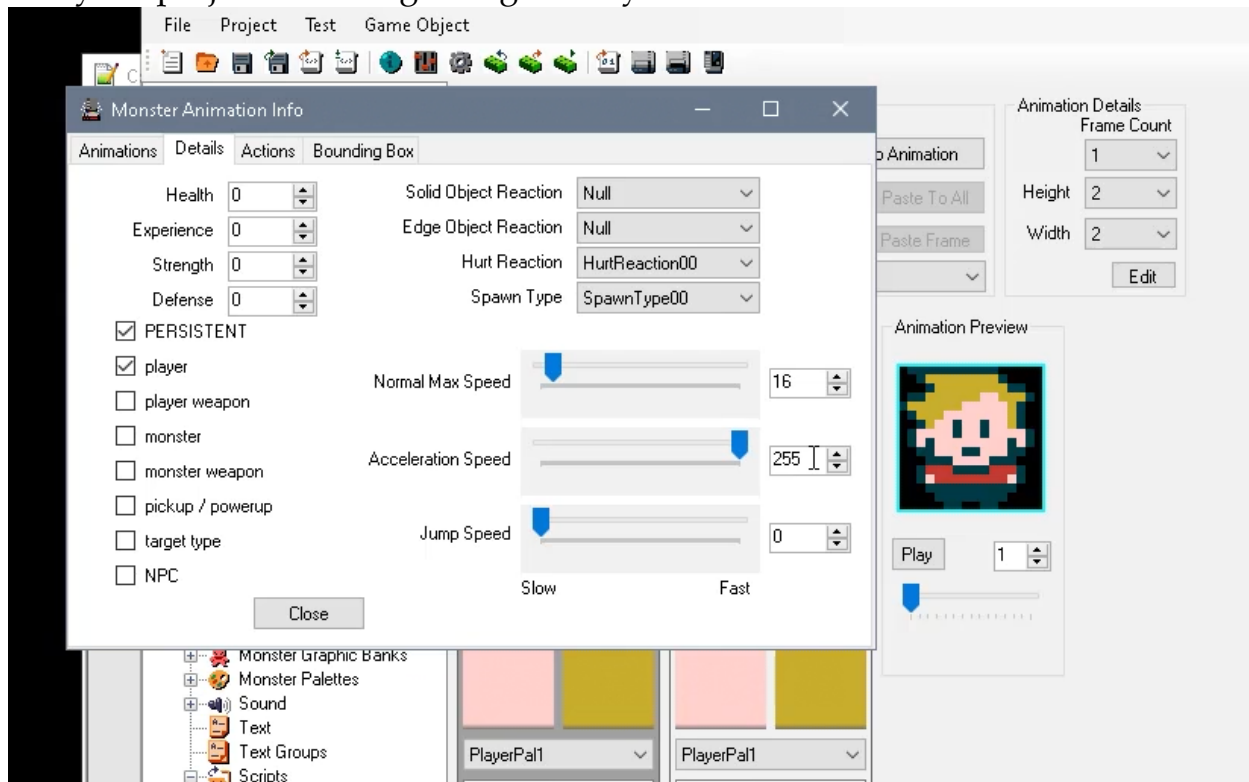
You can see an overlay of my controller in the bottom right corner of this screen. I am pressing down and nothing is happening. This is a common problem, and if you run into it, it may have you wondering if you have set up

your inputs incorrectly. However, there is an extremely simple reason he is still standing still, despite everything being set up correctly. We never gave the Player game object a speed or acceleration speed.

Step 12: Open the player game object from the hierarchy. Click on Object Details and navigate to the Details tab.

Step 13: For now, give the Player object a speed of 16, and turn the acceleration all the way to the right.

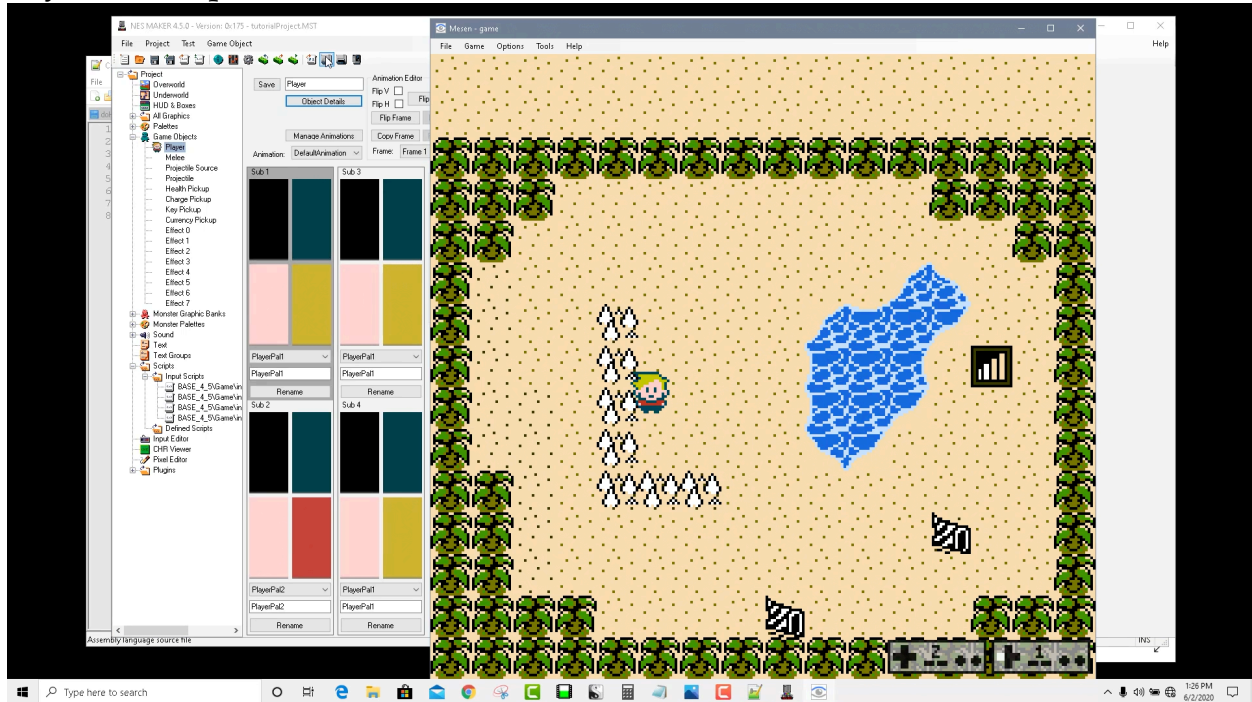
These controls work in conjunction with the physics scripts that were pulled into your project at the beginning when you chose the module.



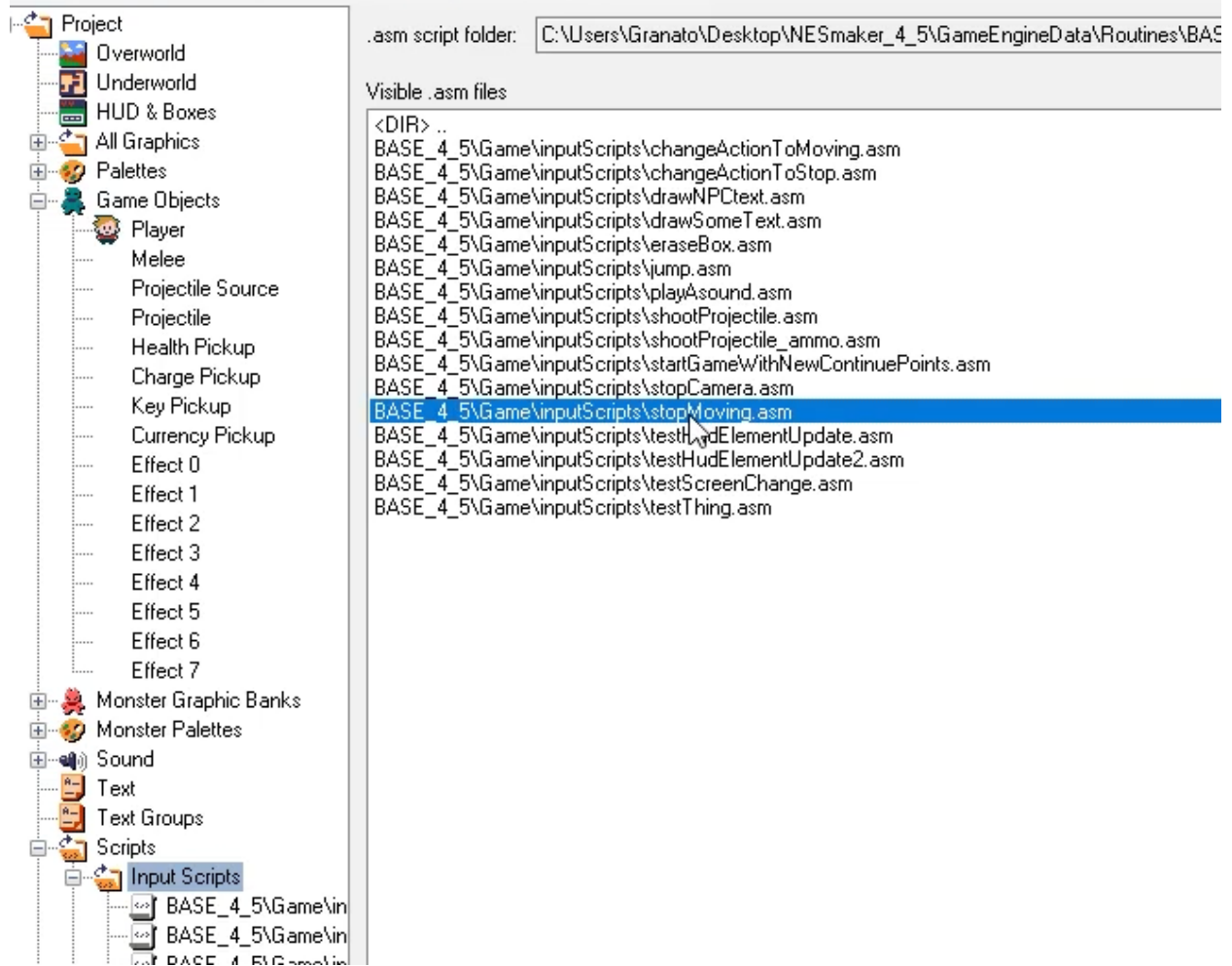
Hit the close button.

Step 14: Test your game. The d-pad buttons now make the player move, just as expected. Obviously, we haven't talked about things like tile collision or what

to do at screen boundaries yet, but there is an even more immediately problem that we anticipated at the beginning of setting up the inputs. We've never given the player object any instruction to stop. We set up inputs that told him to begin moving any time the button is held. We need the player object to stop moving any time a d-pad button is released.

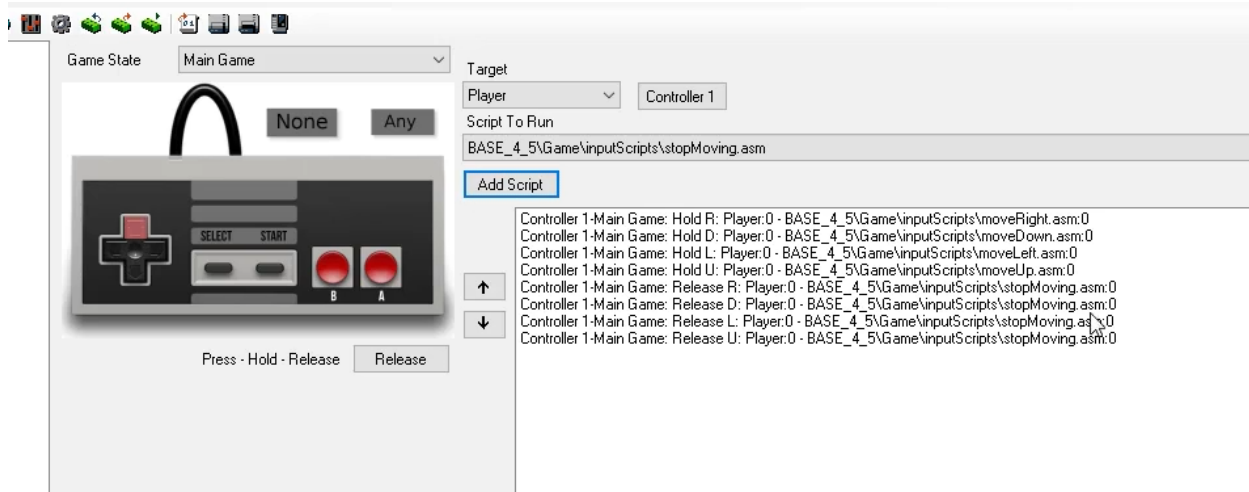


Step 15: Click on Input Scripts in your hierarchy, navigate to Game/ Input Scripts, and double click on stopMoving, which will add this to your hierarchy.



Step 16: In the input editor, set up the following scripts, using the RELEASE button state instead of the PRESS or HOLD button states.

- When I release the Up Button, call the stopMoving script.
- When I release the Down Button, call the stopMoving script
- When I release the Left Button, call the stopMoving script
- When I release the Right Button, call the stopMoving script.



Step 17: Test your game. Now you have a very basic input scheme to handle movement of our hero Greg in his time traveling adventure.

Tile Collisions

Tile Collisions

One of the challenges of creating this sort of tool for a system with extremely limited memory is how to handle the wide breadth of things like tile collisions, object collisions, and physics. These major components to a game can be dramatically different depending on genre, and even within the same genre can vary greatly game to game. This resulted in the module system for NESmaker, which allowed us to make various simple templates that would call the proper scripts that would work together for the desired genre.

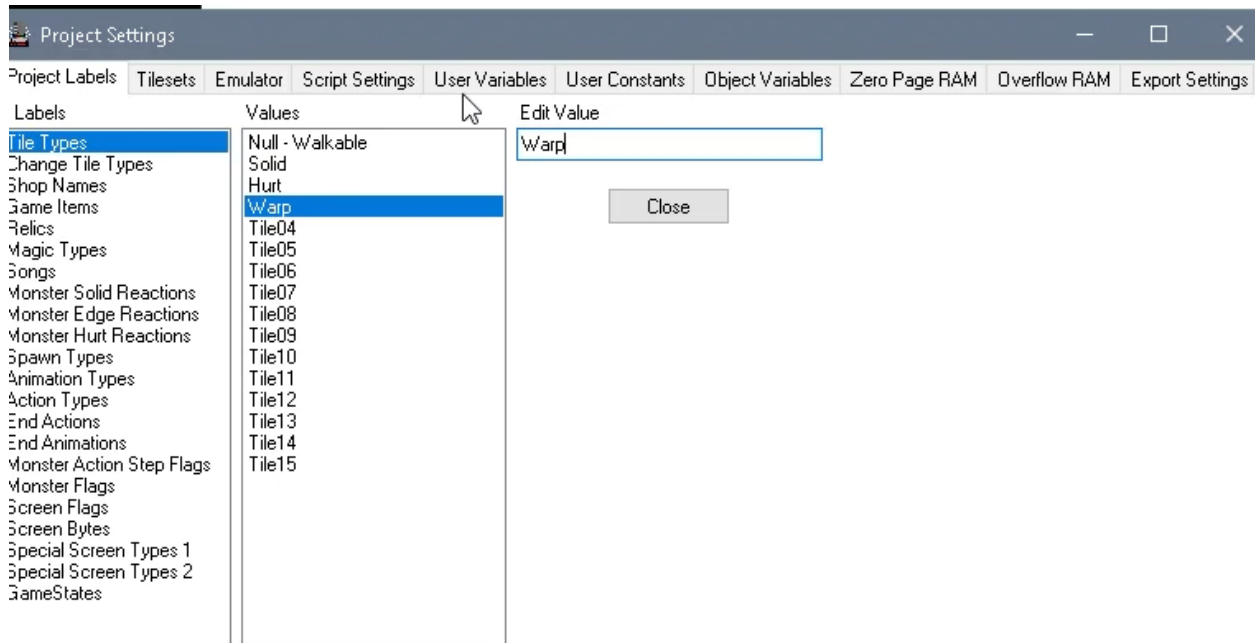
But inevitably, you're going to want to stray from clinging so close to the tutorial projects. You're going to have your own ideas and your own visions that improve upon or go beyond the default templates. For this reason, it's important to know how these things are set up by the tool.

For our current game, we have conceptualized a few tile types.

- SOLID: A solid object I can not pass.
- HURT: Spikes, which should hurt me, even though we have not declared what hurt should mean for this game yet.
- WARP: A staircase that could take me to a different screen

Step 1: Go to Project Settings using the gear icon at the top of the screen. Again, we're going to set up some labels for different types of collision types we want our game to have. The first type of label is for tile types.

For this game, we want a walkable tile, a solid tile, a hurt tile, and a warp tile, so we will name the first for tile types as such.

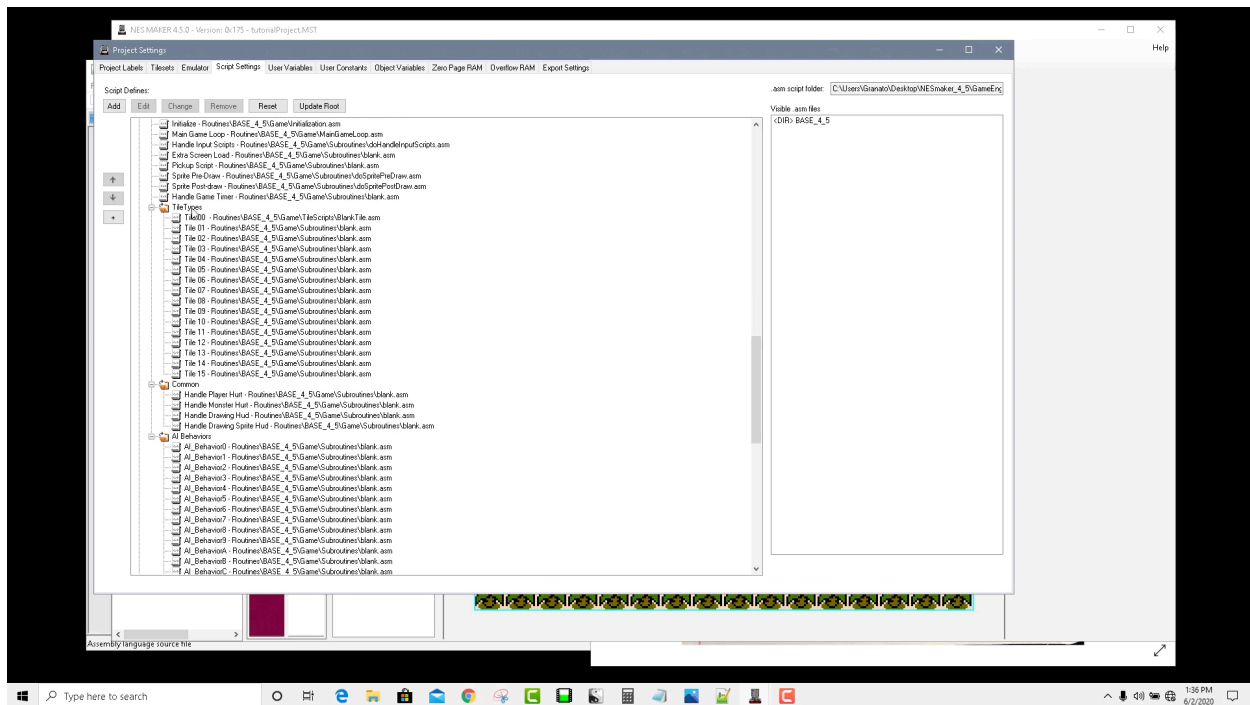


Now, it is important to understand, doing this has no bearing on what the tiles actually do. This will not suddenly make tile type 1 be a solid tile or tile type 2 be a hurt tile. Just like with Game States, this is a tool reference. When we see a reference to tile type 2 in the tool now, it will say Warp instead of Tile Type 2, which will help us easily remember which tile is which.

Just to check to see what this does, close project settings, open your overworld and your screen. If you go to the screen painter's collision drop down, you'll see that there are now four types outlined. Instead of just saying tile0 - tile 15, they are given proper names.

The next step will be actually invoking the scripts that cause the tiles to behave in these expected ways.

Step 2: Open Project settings and navigate to the Script Settings tab. Scroll down and find the scripts that are referencing tile types.

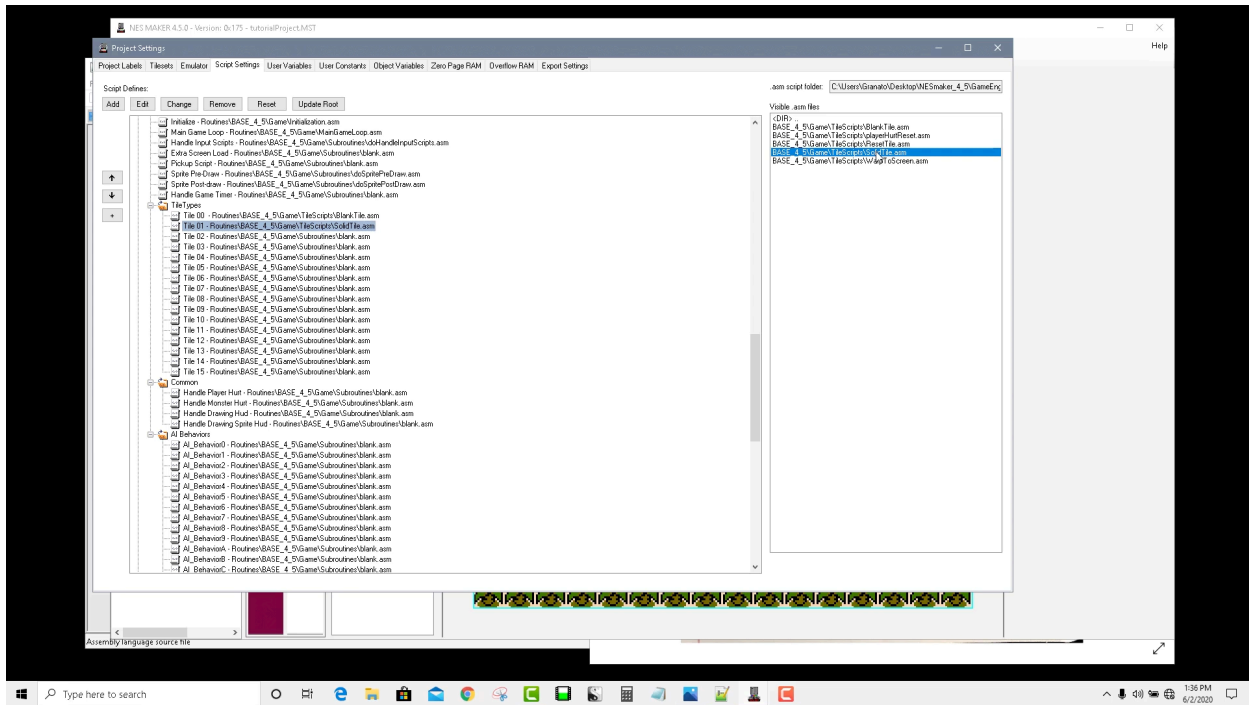


Step 3:

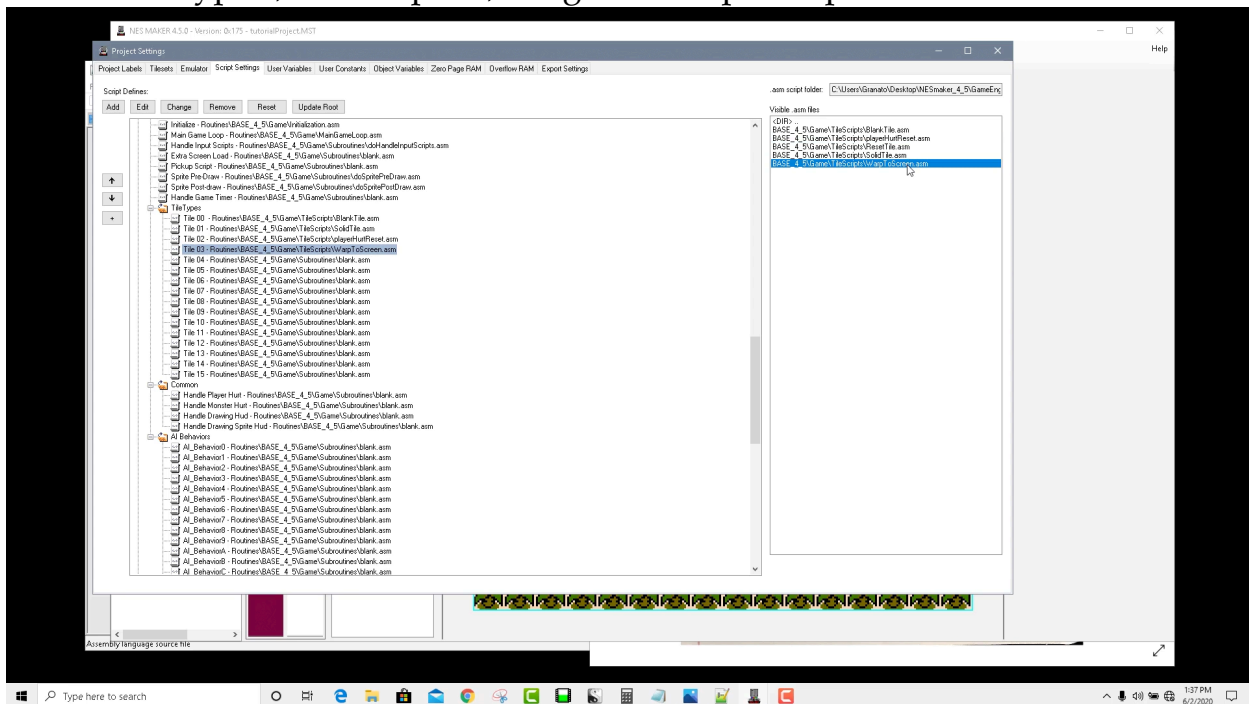
We can see that Tile 00 is set to a Blank Tile. This is just an empty script. That's exactly what we want to have happen. When an object runs into Tile 00, nothing happens.

But for the others, we need to assign them. Click on Tile 01, and then use the finder on the right of this window to navigate to root / Game / Tile Scripts, and then choose SolidTile. Double click on it, and you'll see that in your script list, it now says that the Tile01 reference now points to the SolidTile.asm script. This is how modules are built and customized - by assigning the appropriate scripts to the right code tethers.

Now, because of the physics engine that is part of this module, its tile collision section will check to see what should happens when an object runs into Tile01, and when the game is in that condition, it will run this particular script. This is similar to what we did with the input editor, checking to see a condition and then running a selected script.



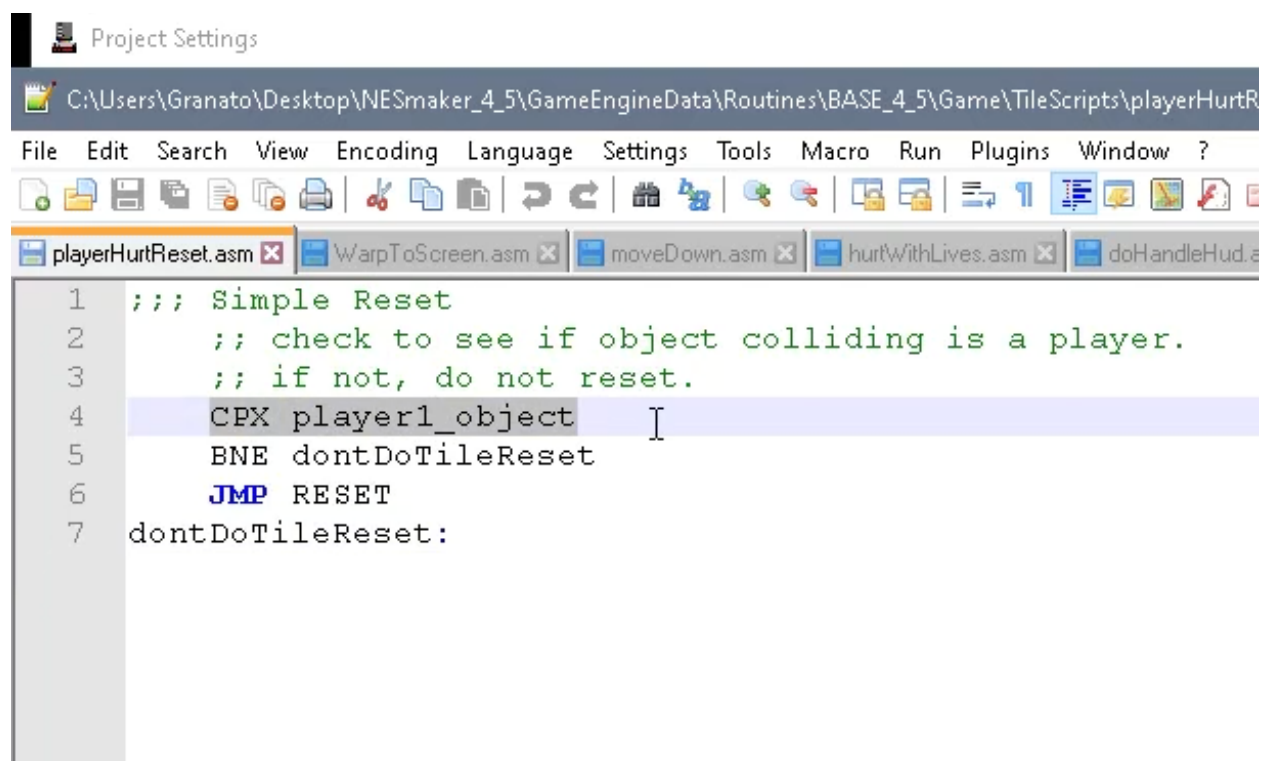
Step 4: For tile type 2, the hurt tile, assign the script called playerHurtReset, and for tile type 3, the warp tile, assign the script WarpToScreen.



If you ever want to look at one of these scripts, you can click on it, press edit, and it will open that script for viewing and editing in your selected code editor.

Another way to see it in NESmaker's code viewer and editor is to look at the Assigned Scripts node in the hierarchy. Every assignment by the current module shows up there and can be viewed and edited right inside the tool.

Again, just to take a quick look at the code that's running underneath, try editing the playerHurtReset script. You'll see it pop up in a text editor. Even though we don't know any code at this point, we can pretty easily walk through what happens when an object collides with this tile.



The screenshot shows a text editor window titled "playerHurtReset.asm" with the following assembly code:

```
1  ;;; Simple Reset
2      ;; check to see if object colliding is a player.
3      ;; if not, do not reset.
4  CPX player1_object
5  BNE dontDoTileReset
6  JMP RESET
7  dontDoTileReset:
```

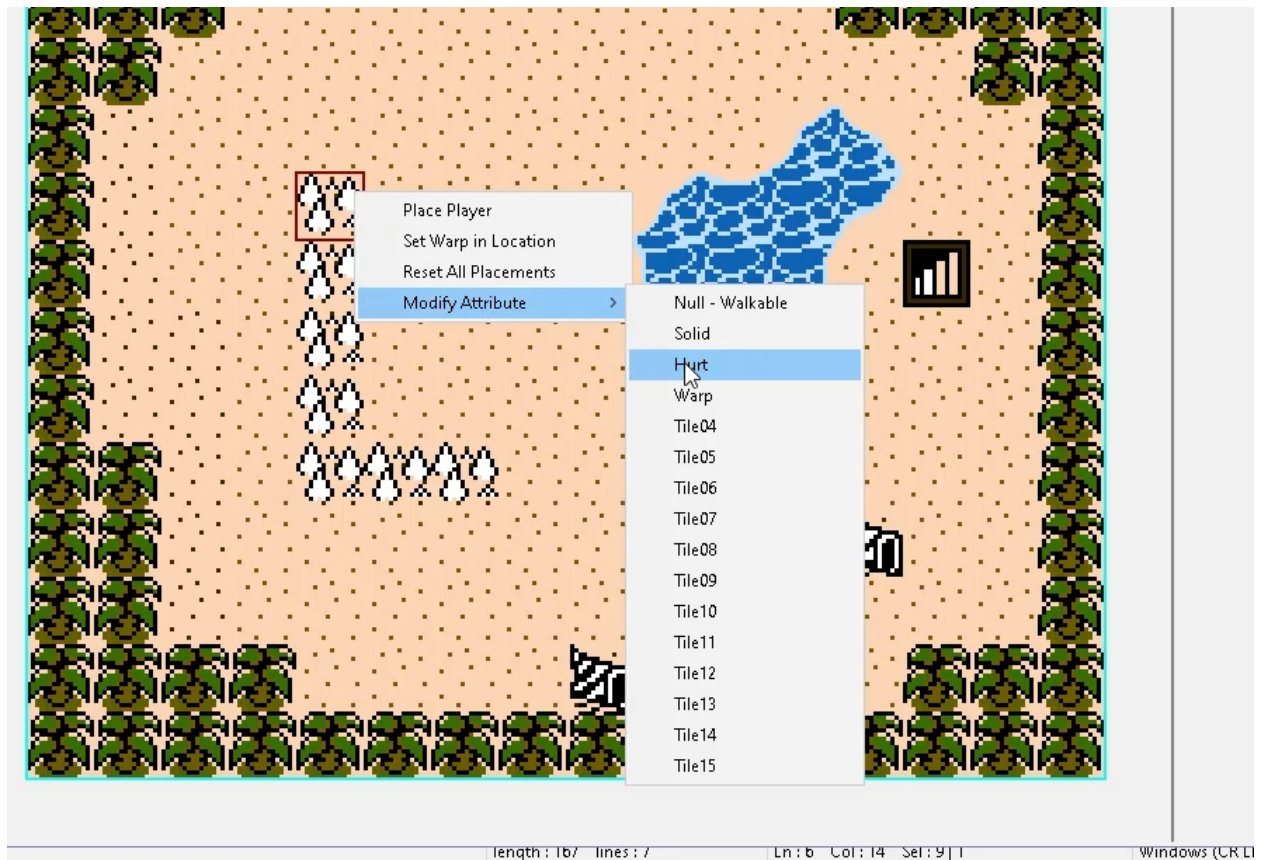
The first line, check to see if the object is a player object. If it is not the player, skip passed the reset. This means that only a player object will cause the reset to occur. So if your monster or projectile runs into a hurt object, it will skip over the reset function, and only do anything if object colliding is a player type.

Now maybe eventually you want to edit this for your own game. Maybe you do want certain objects to interact with this. Maybe projectiles get destroyed.

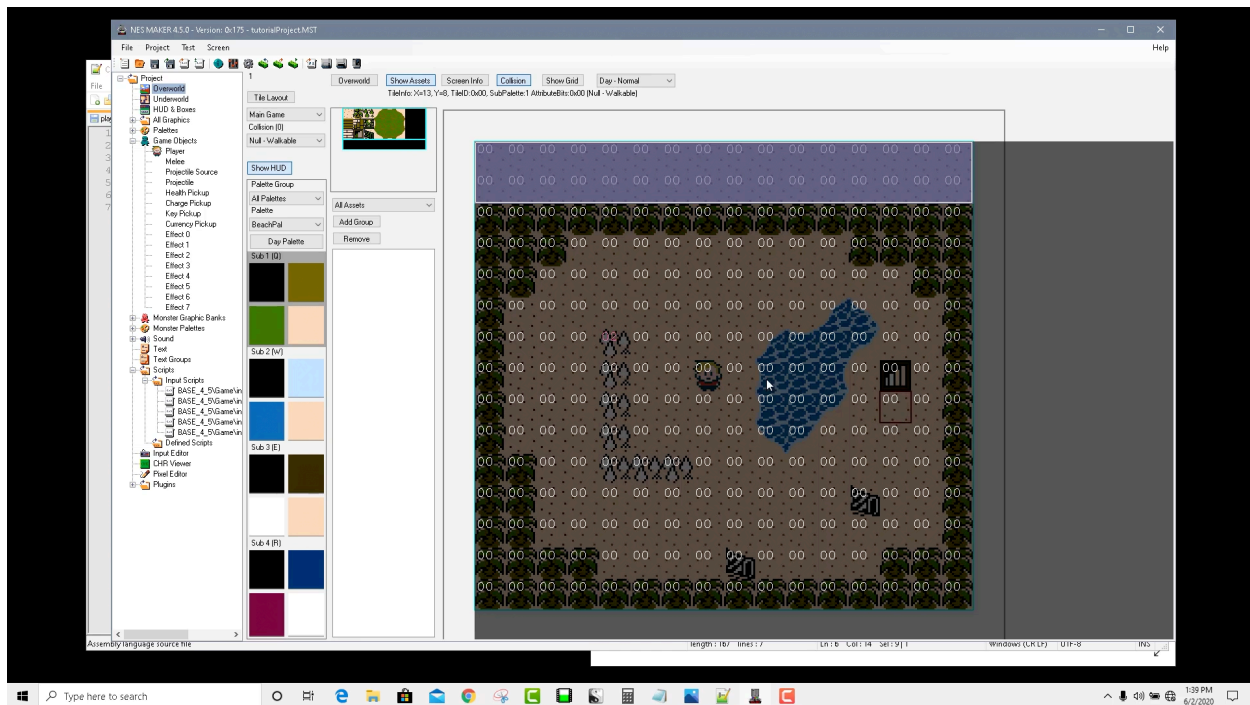
Maybe monsters change direction. Writing a line or two of code beyond what exists would allow you to create another case for what should happen to a non-player object if it ran into this. Or, you could find another NESmaker users who uses the same physics scripts who has developed a more interesting hurt collision script, and you could replace this script reference with that user's custom script with just a few clicks.

The main reason to look at this is to understand the relationship between what is in the script defines in the project info and the scripts running underneath.

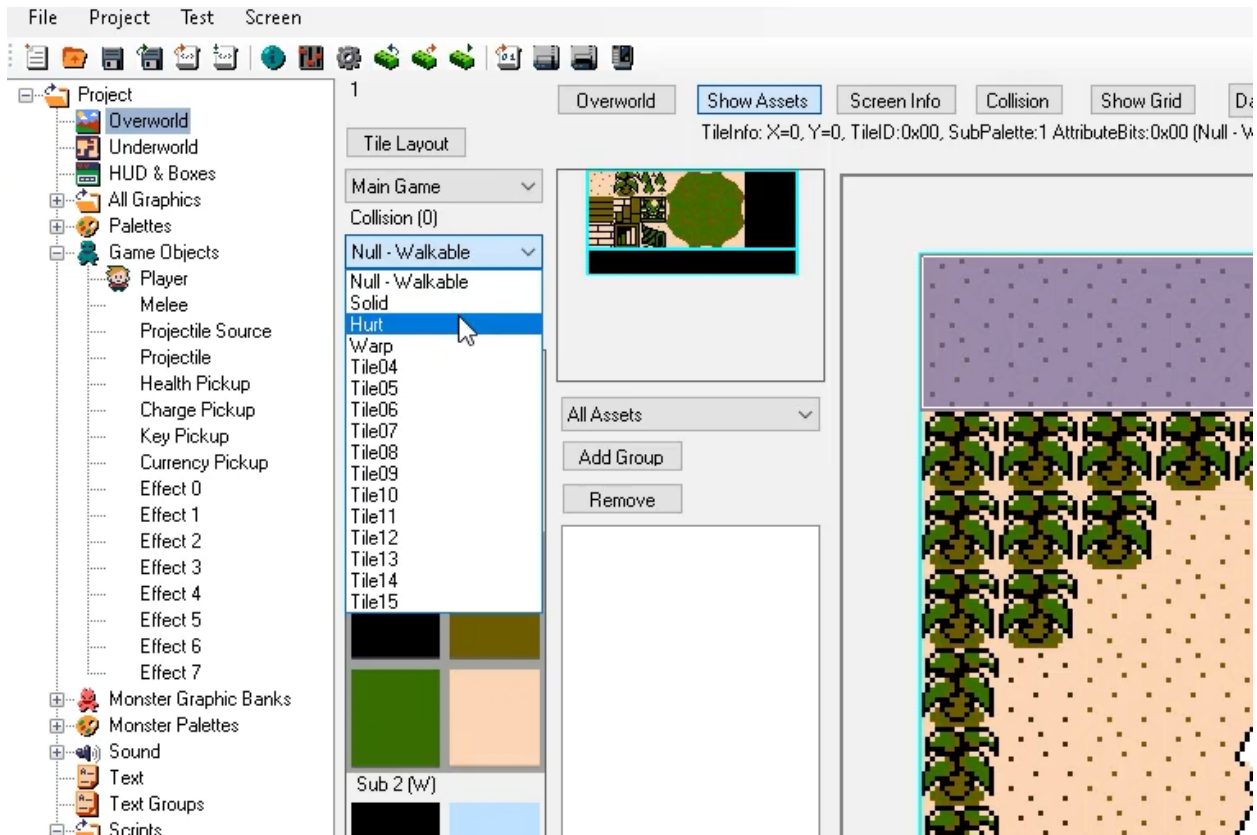
Step 5: Navigate to your screen in the screen painter. There are a few different ways to modify the collision data on your screen. The first way is find a tile whose collision data you want to change, right click on it, select change attributes, and change it to the desired collision data. This is probably not how you will do screen painting with collision info, but it's good to understand that it's an option, especially dealing with spot changing single tiles.



You can also press the collision button in the screen painter menubar at the top of the screen to toggle an overlay that shows the numeric collision data for each tile. You can see that all tiles are currently null except for the one spike that I just changed, which now has a small 2, denoting it is tile type 2.



We are going to hold off on painting the solid tiles for just a moment and focus on the hurt tiles. The easier and more common way to paint collision data to screens is to go to the collision dropdown list and choose your desired collision type.



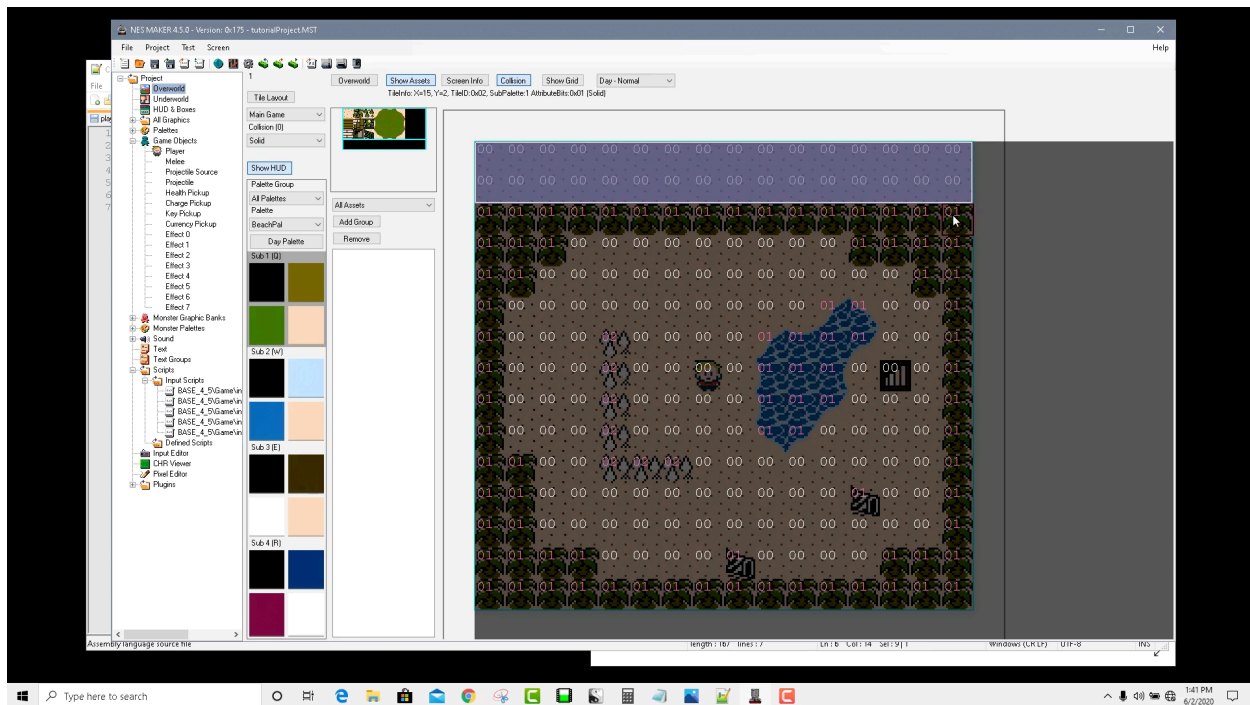
Now, place your mouse over any tile you wish to change and press the zero key on your keyboard. This will paint the currently selected collision to that tile. While holding zero key, your mouse is a paintbrush for tile collisions.

Sometimes it's easier to paint collisions while having the collisions overlay turned on, since you have visual feedback for each tile's collision value.



Test the game. Try running into your hurt tile. What should happen is when you run into the hurt tile, the game should reset.

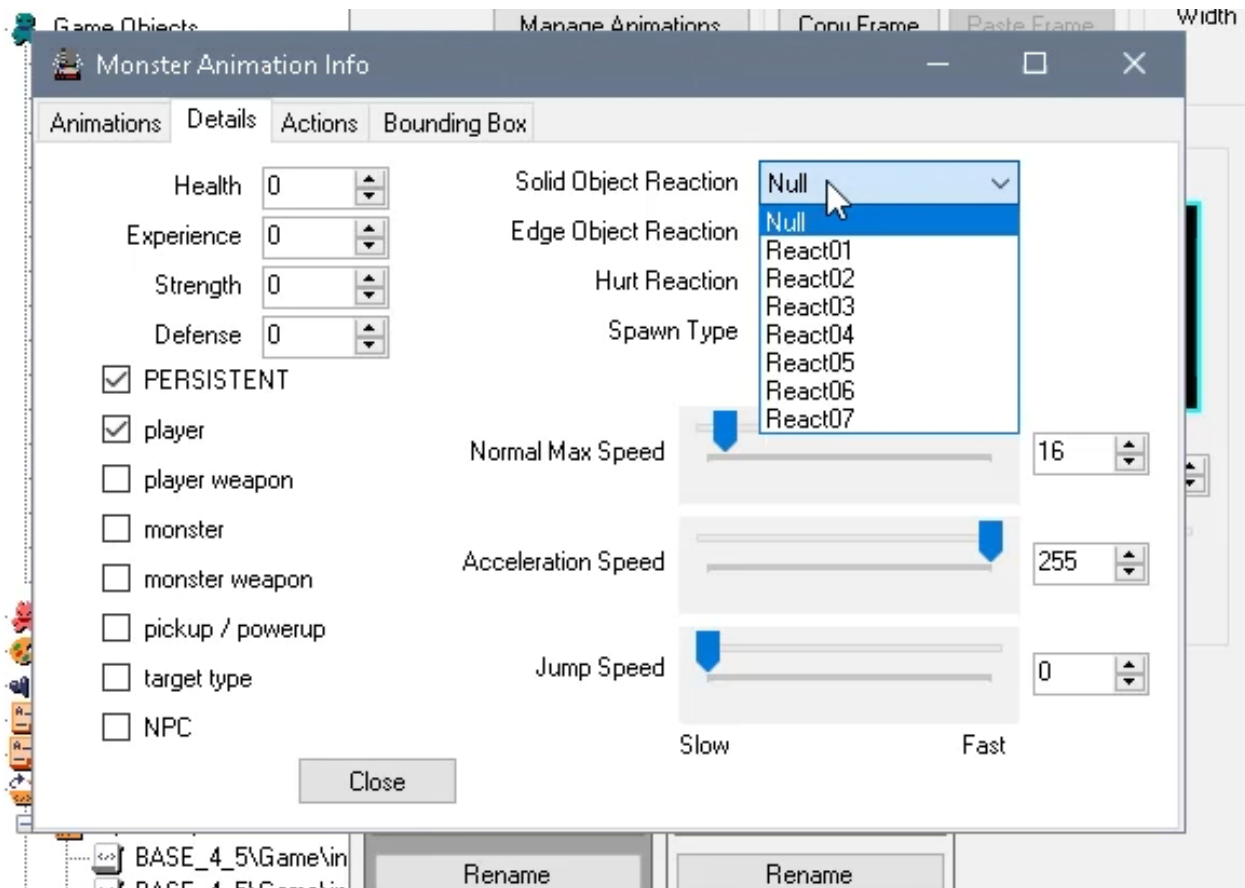
Step 6: Paint all of the solid tiles with their appropriate value.



If you were to test your game now, you'd notice that your player walks right through the walls even though they're supposed to be solid. At this point, that's expected, because we haven't told the player object what his very particular behavior should be when he interacts with a solid. Because in most games, there are multiple ways that objects interact with solid tiles (players stop, projectiles explode, monster turn around and move the other direction, balls bounce, players hit their head and stop moving down, players land from a jump, etc), every object has a specific definition for what should happen when it hits a solid tile. Let's set up what should happen for a player in this game.

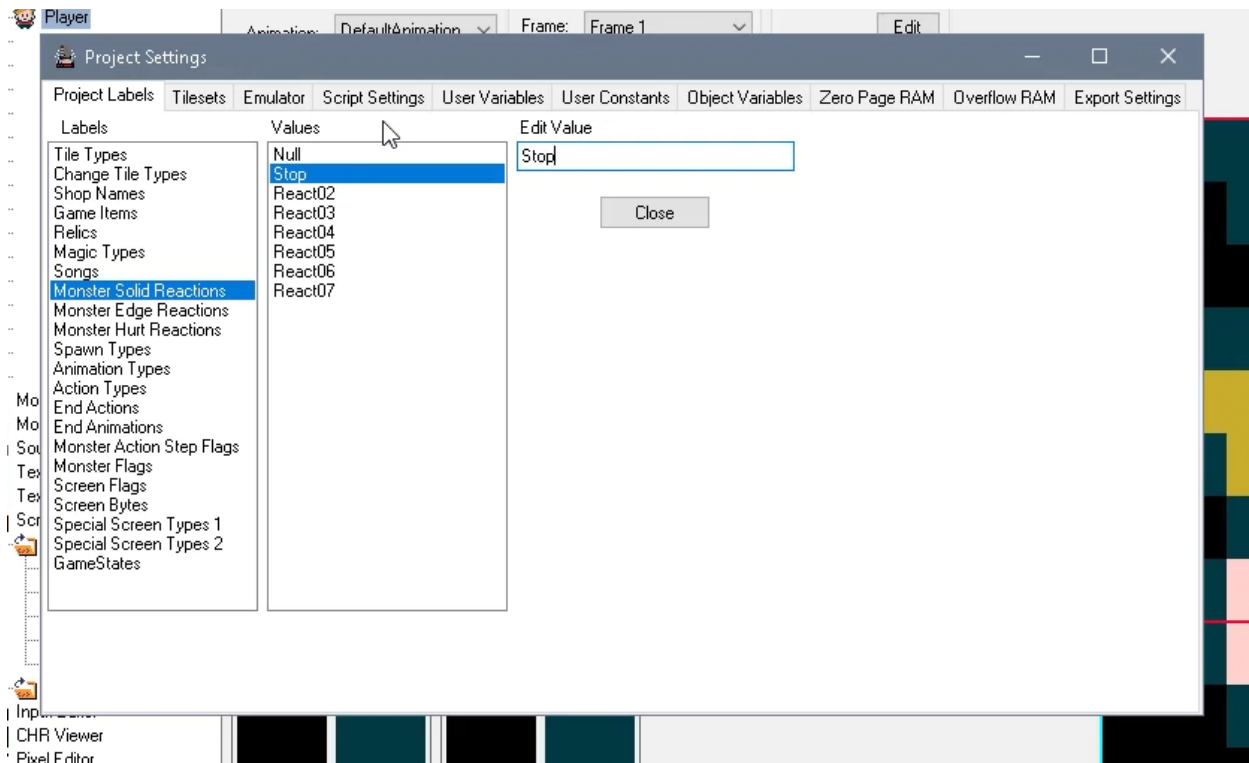
Step 7: Open the player object from the hierarchy. Click on Object Details and navigate to the details tab. You'll see a dropdown for Solid Object Reaction. Right now, NULL is selected. That means that when this particular object runs into a solid wall, it does nothing, which is why our player is currently walking right through it.

Clicking on the dropdown shows you your current options.



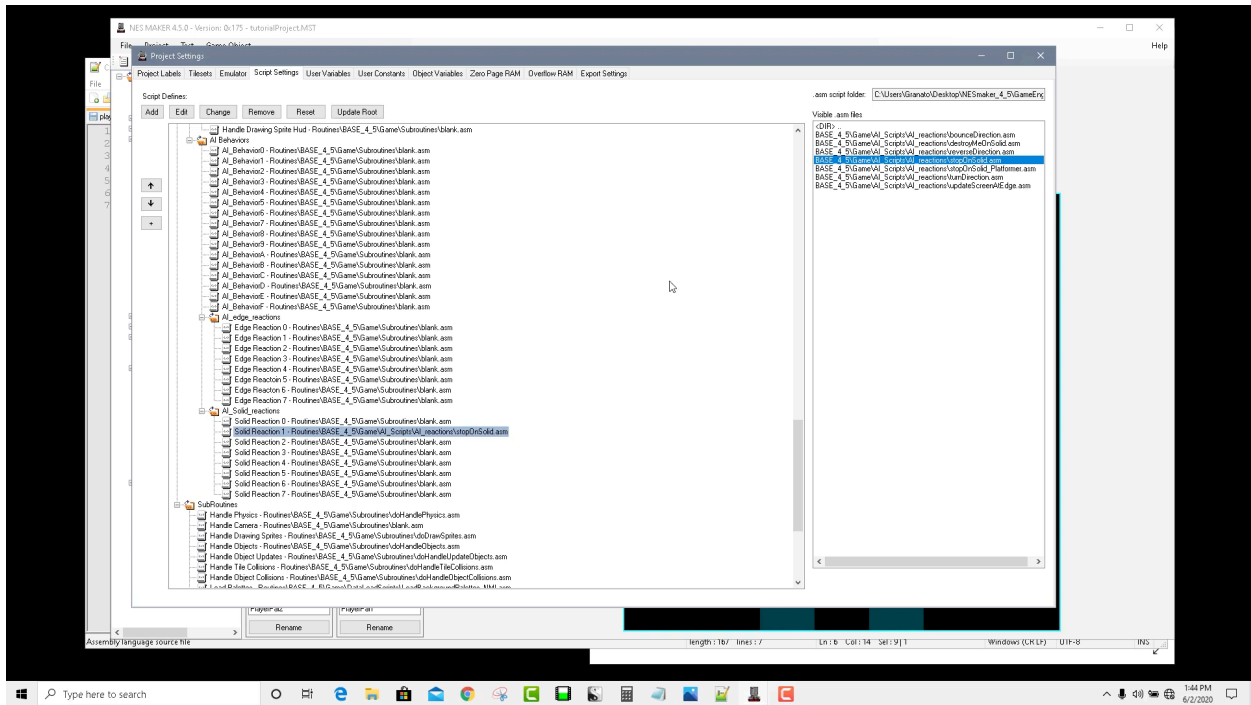
Currently, these are undefined. We need to make a label and assign one of these solid reaction types to a stop behavior.

Step 8: Close object info, go to project settings, and in the labels tab, find Monster Solid Reactions. Keep the first choice called Null, but change the second choice to Stop.

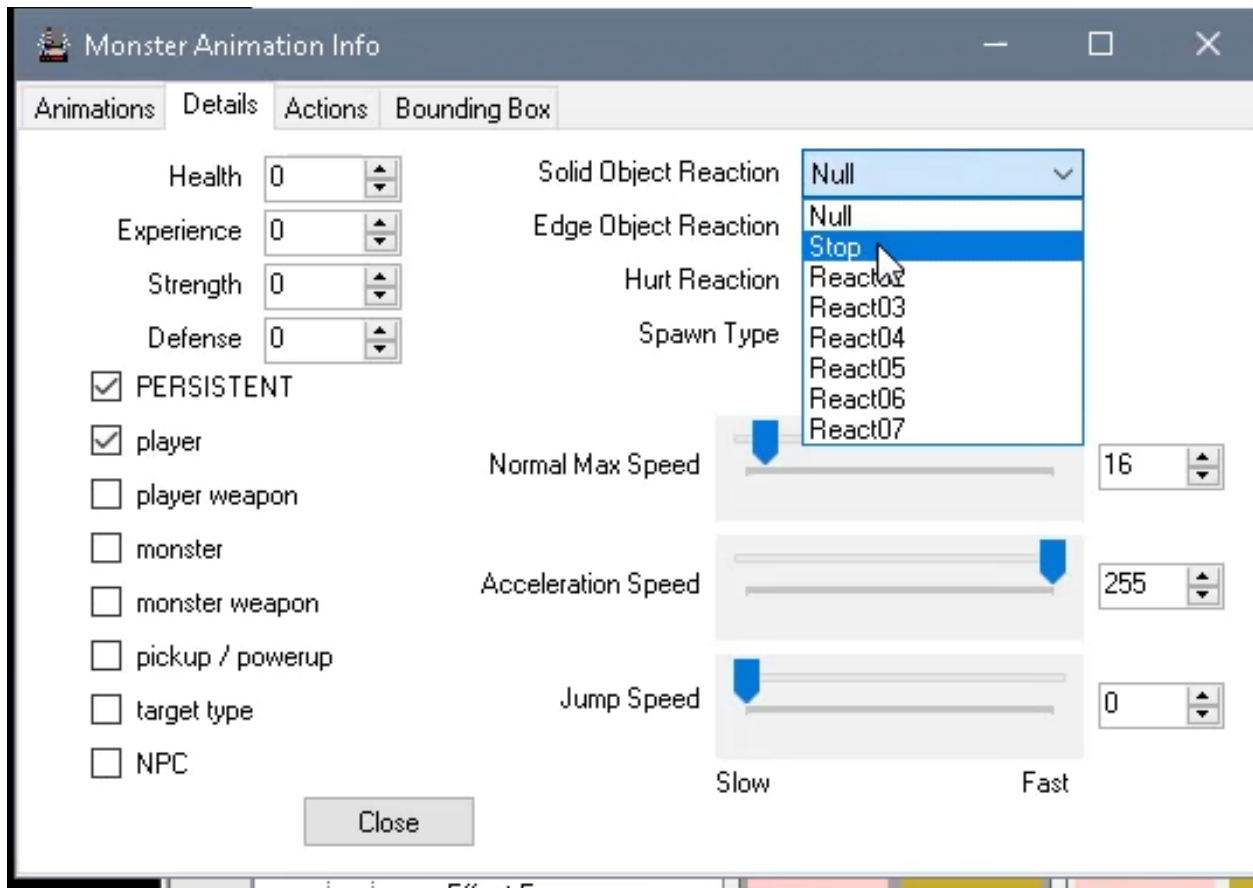


Again, just like with changing the names for tile types, this doesn't do anything. It is just for our tool-side organization, so that we can remember reaction type 0 is null and reaction type 1 is stop. Now we have to set up the stop behavior.

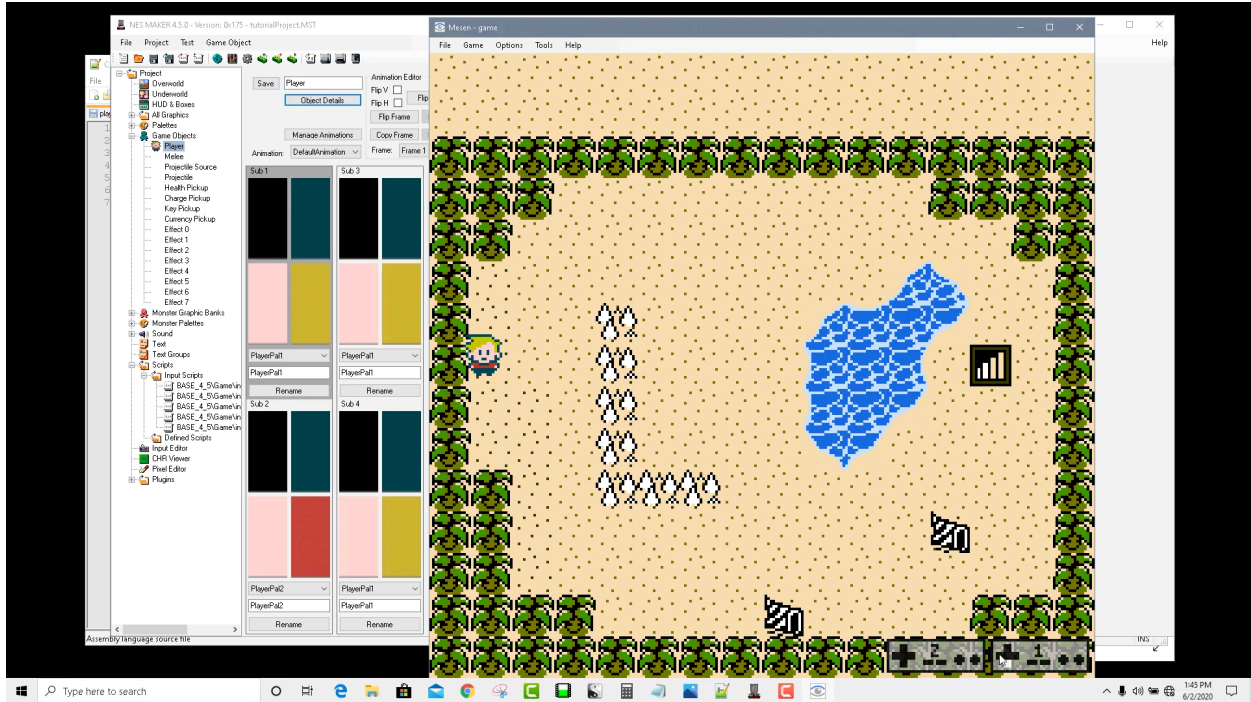
Step 9: Go to Script Settings, scroll down to AI Behaviors and you'll see a subfolder called AI Solid Reactions. Click on Solid Reaction 1, and assign the script from Base / Game / AI Scripts / AI Reactions / StopOnSolid.



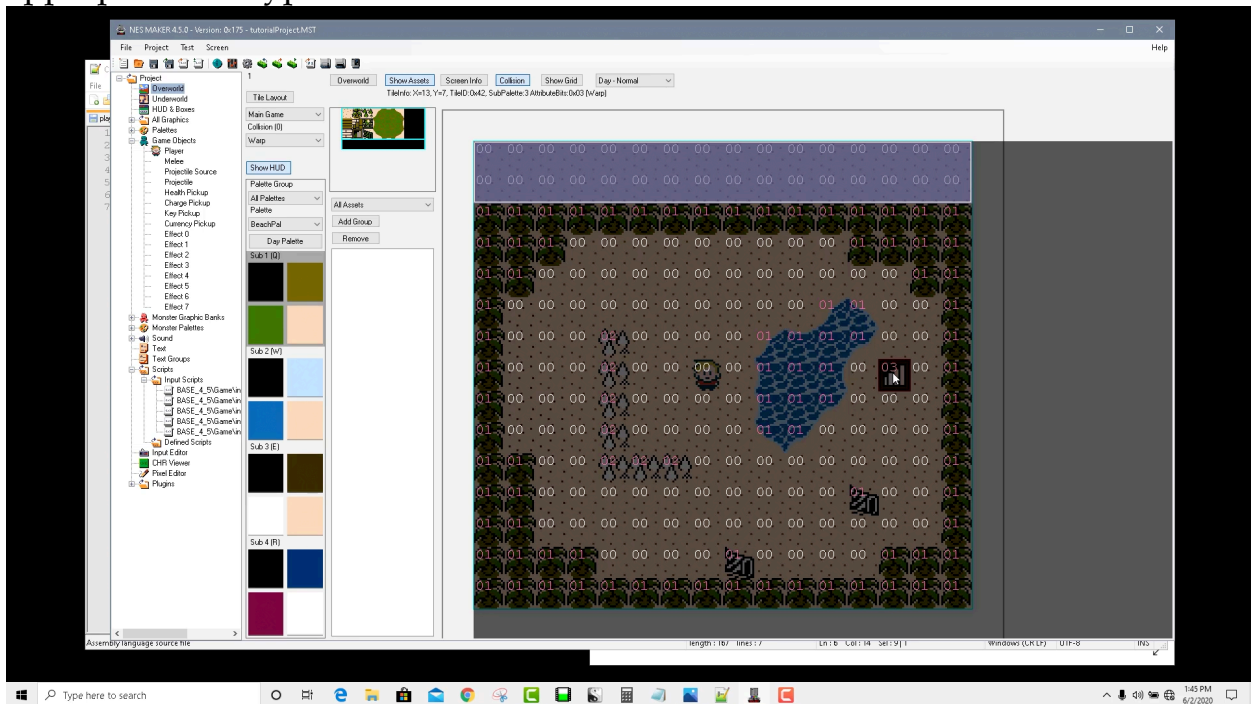
Step 10: Open the player object again, click on Object Details and go to the details tab again. If you click on the Solid Object Reaction dropdown, you'll now see that STOP is in the place where Behavior 1 used to be. Choose stop. Now this object knows that when it hits a solid object, it should do the Stop On Solid script.



Step 11: Press the close button. Test your game and you'll notice that the solid tiles will keep you contained in the room.



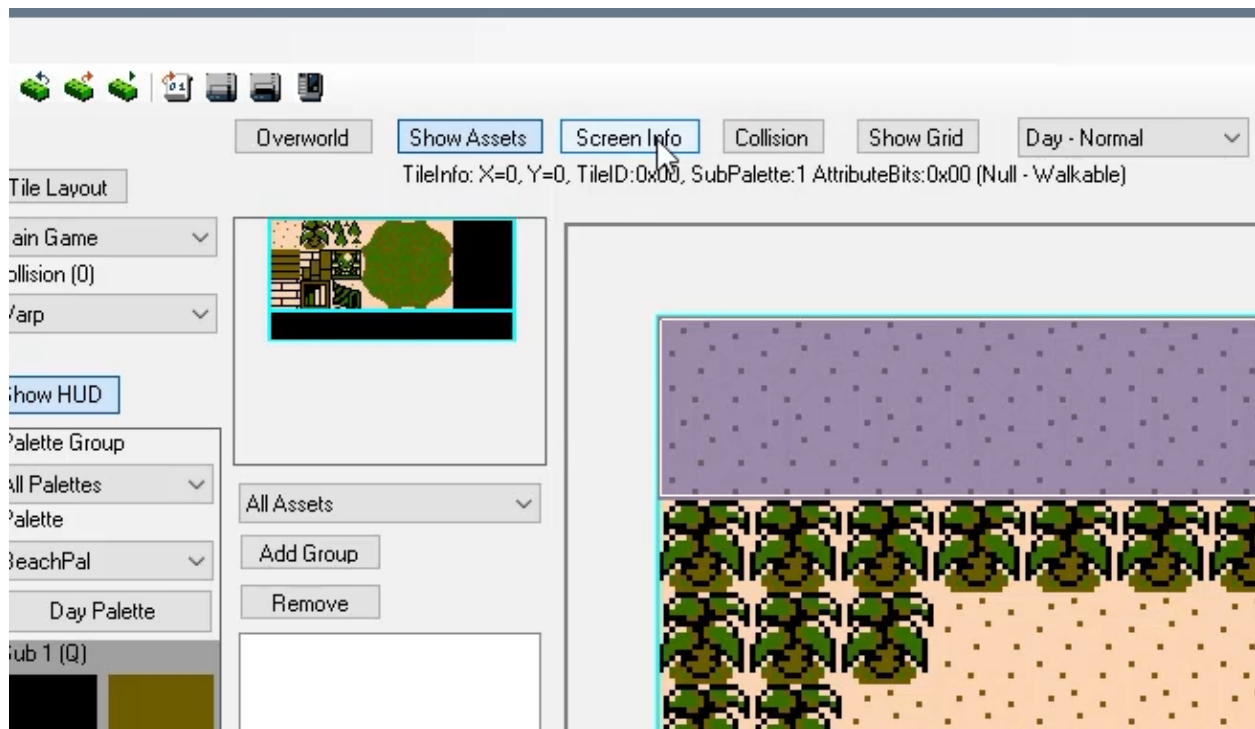
Step 12: Return to your screen painter and change your warp tile to the appropriate tile type.



Step13: Test your game and walk your character over to the warp tile. It will

work, but it might do something strange. It will warp your player to screen zero, and placed your player at tile zero. The reason for this should be pretty obvious by now. We haven't yet set up any data on where the player should warp, or what position he should be in when he gets to that new screen. Since no data was entered, it's null, or zero. Thus, he's moving the player to screen 0, and placing the player at position 0.

Step 14: Click on Screen Info.

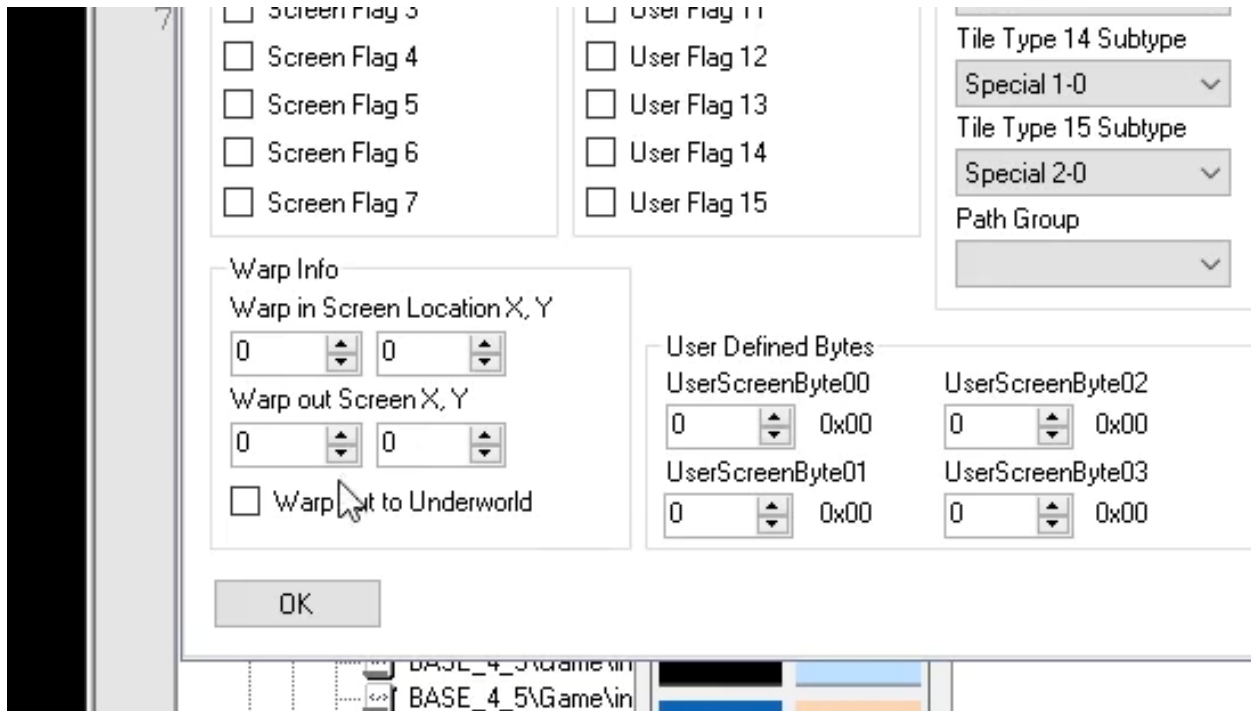


In the bottom left corner, you'll see a few boxes called Warp Info. The first two values tell the engine "If I warp INTO this screen from a warp, I should arrive at position x,y. The second two values say "If I hit a warp tile on this screen, I should go to the screen on the map at coordinates x,y."

So to set up a warp properly, we'll need to make a new screen, set these warp out coordinates to that new screen, and on that new screen, set the warp in position where we want a warped character to appear.

Right now, if I hit a warp on this screen, it will go to screen 0,0. If I warp from

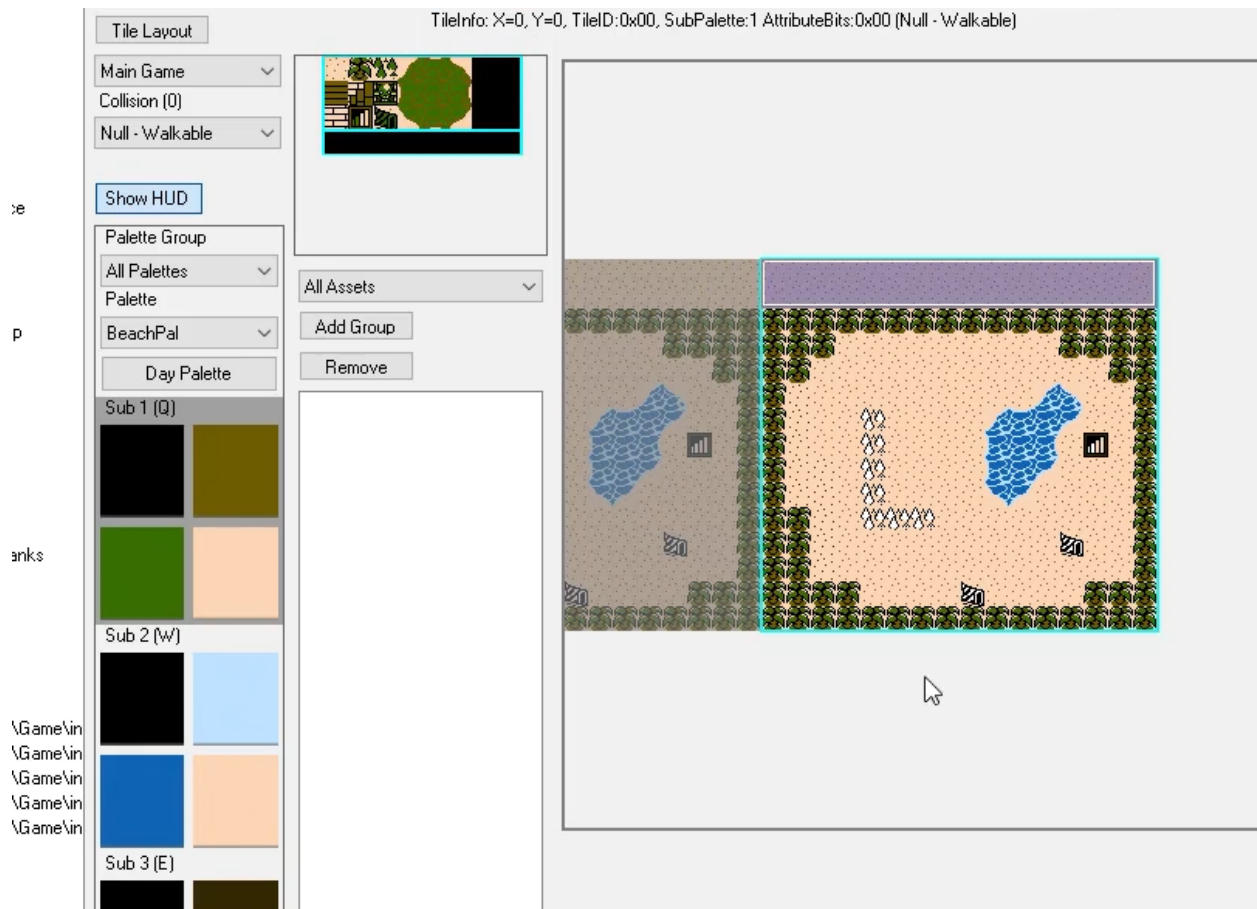
a different screen into this screen, it will place me at position 0,0.



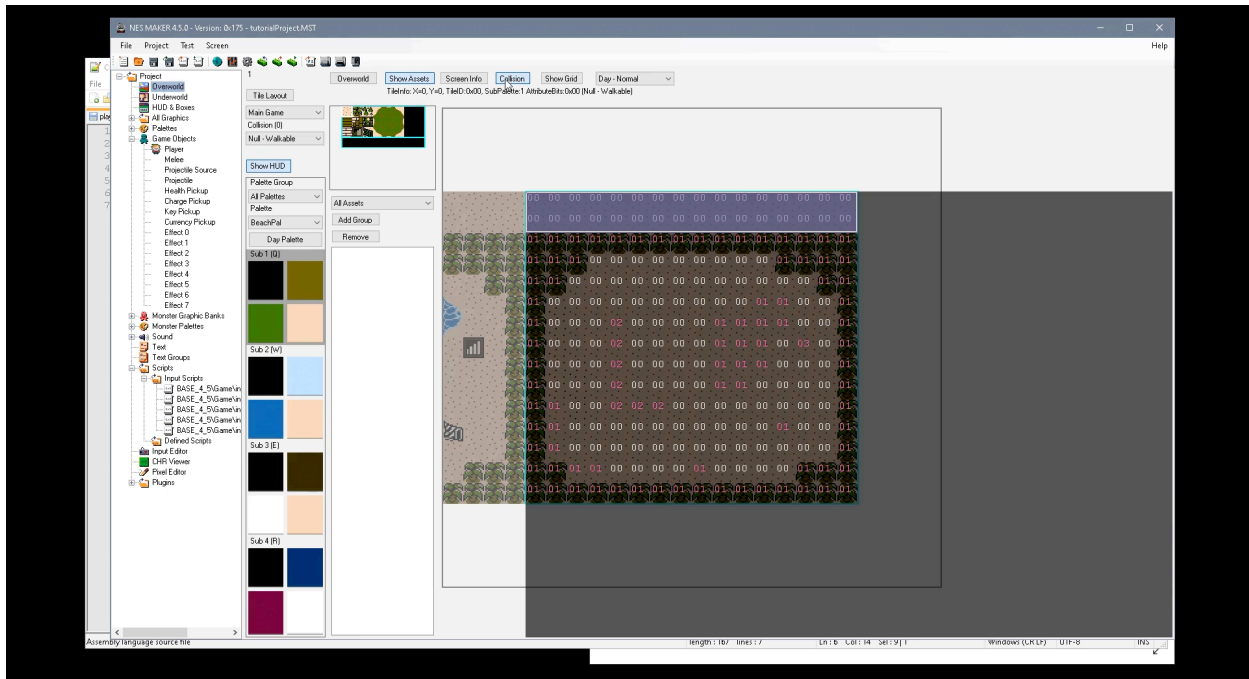
Step 15: Click on Overworld in your hierarchy to open your map screen. We're going to make a second screen. It can be anywhere on the map, but for our purposes, we'll just put it right next to the first screen. There are a few different ways we can make a new screen.

- If you double click on an empty screen, it will create a new screen with all of the values set to null, just like they were when we created our first screen.
- If you hold shift and control on the keyboard, you can drag your existing screen onto a new cell to copy all of its data into the new screen position. This is a great option if you want to create a new screen while maintaining the tilesets and other settings. All you'll have to do is clear out the tiles.
- You can also click on an existing screen and press control C, then move over an empty cell on the map and hit control V. This effect has the same result as shift-control and drag, copying all of the contents of the one screen to the new screen.

While you're looking at a screen, you'll notice that you can see the bordered tile of the adjacent screens. This is to help you line up collisions. You don't want to go from one screen to another and get stuck in a wall or instantly run into a death tile! If you need to see more of the screens around you, you can hold your mouse over the screen and hit the plus and minus keys on your keyboard to zoom in and out a bit.



Step 16: Use the blank tile to clear out the screen's contents, and maybe try to make it look slightly different. Keep in mind that by using the screen painter rather than assets (which we haven't learned about quite yet), you're only painting the tile data, and the collision data is still existing underneath. So after you're done clearing out the tile data, make sure to choose null collision from the collision drop down and paint your wrong collision data using the zero key.

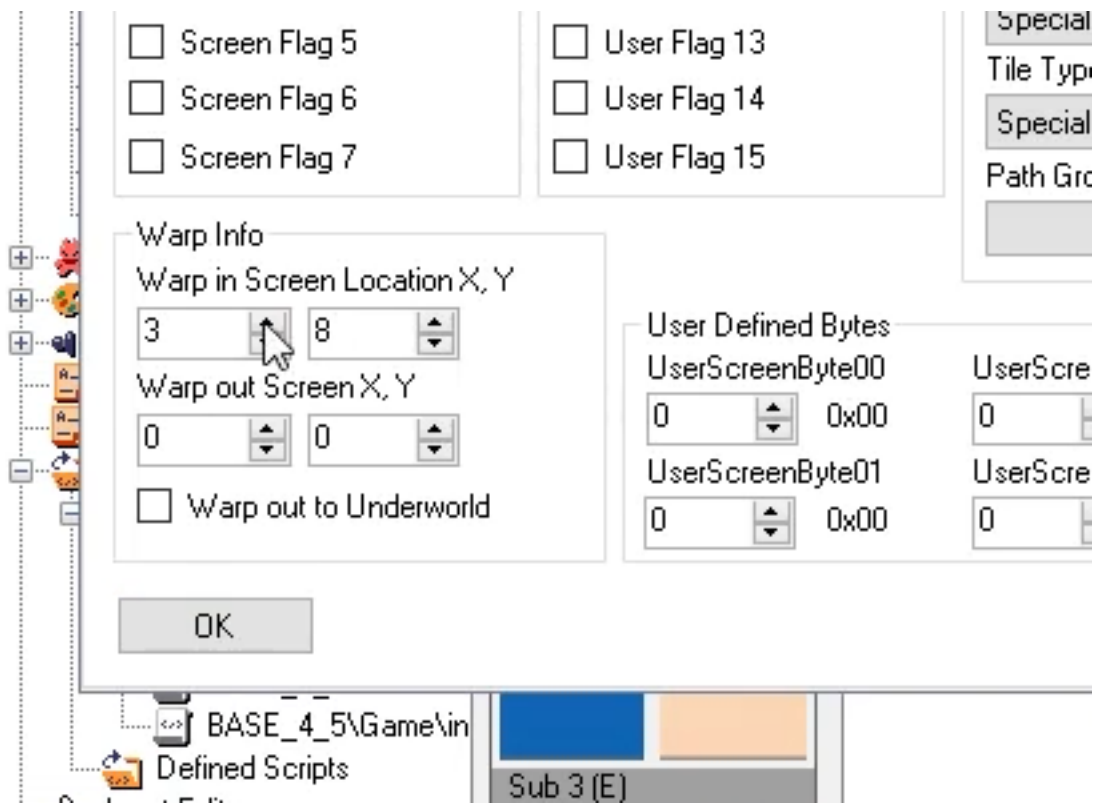


If you want, add some more tiles to give the screen a bit of character.

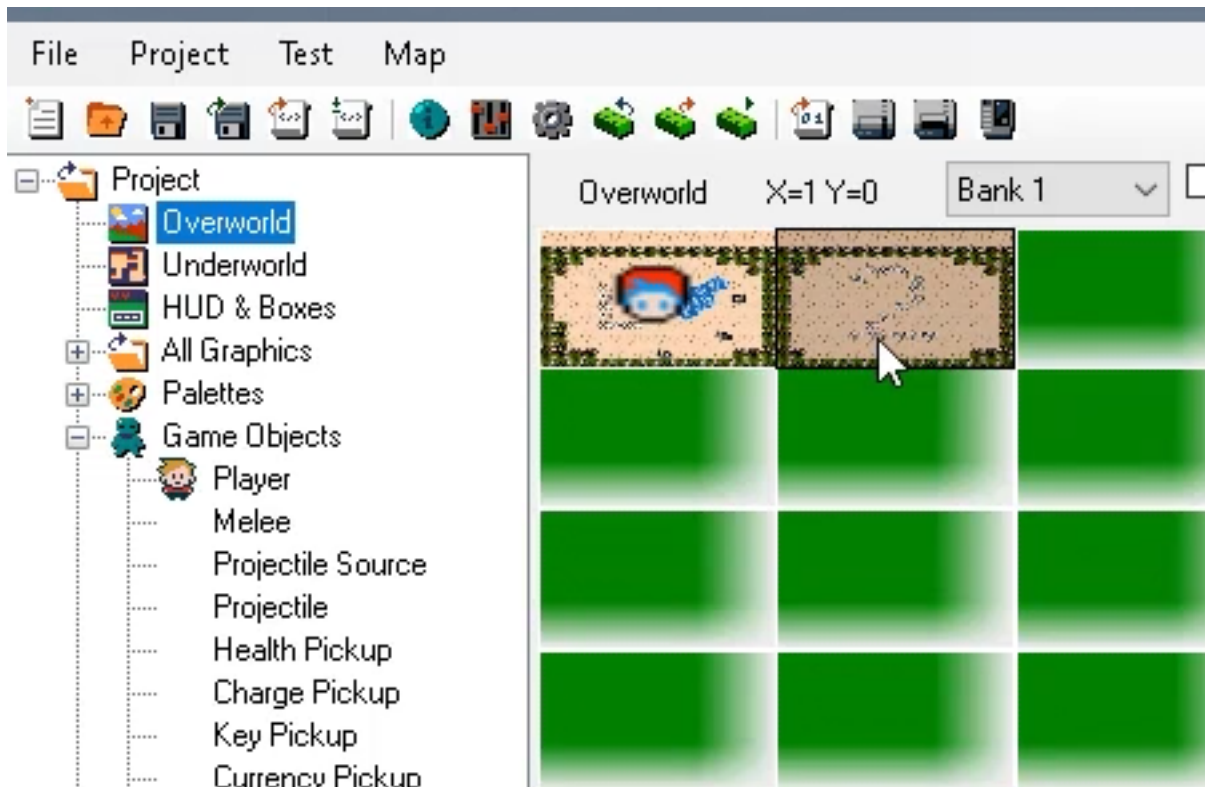
Step 17: Right click somewhere on the screen and select “Set Warp In Location”. This will update the screen info so that when the player warps in to this screen, it will warp to the set coordinates of the tile you right clicked on. An important note - don’t make this another warp. Think about what happens if you do. You would warp into this screen. Your player would immediately have a collision with a warp tile, and run the warp code, warping out to wherever this screen sends them. There are some clever advanced tricks that you could use to handle this, but the easiest thing is just have the player warp in next to a warp if you really want to set up a warping back and forth type scenario between two screens.



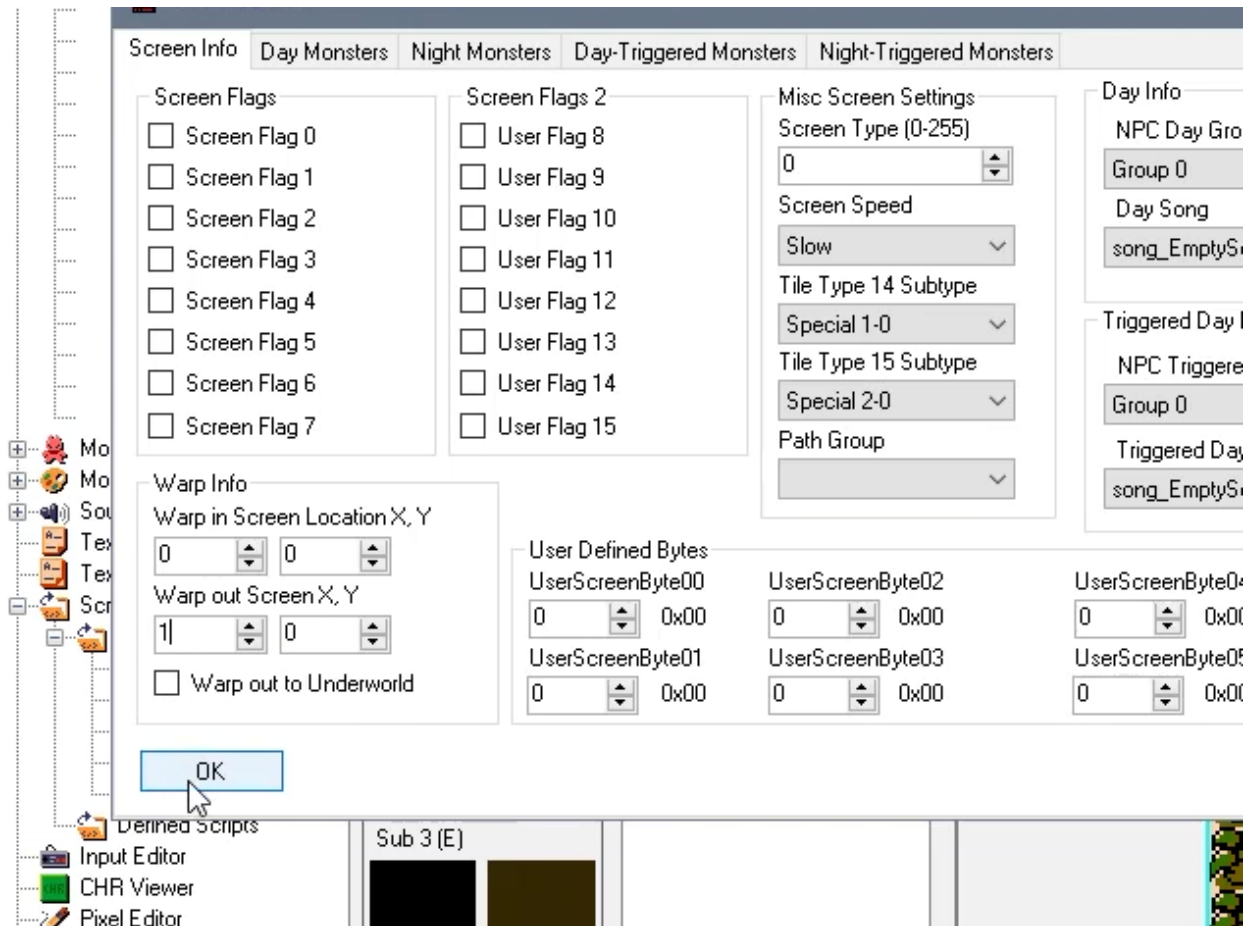
If you go to screen info, you can see that your Warp In screen info has changed to reflect the coordinates on screen you just selected.



Step 18: Now we need to return to the first screen to connect the warp. You can actually navigate between map screens using the arrow keys, or you can return to your overworld map and click back on your original screen. On my map, I need to connect screen 0,0 to screen 1,0. If you have placed your screens in a different location on the map, you can easily see the x,y map coordinates listed at the top of the screen when you mouseover one of the cells if you need a quick reference.



Step 19: In the screen info for your first screen, change the Warp Out Screen coordinates to 1,0, or to the coordinates of your second screen if you did not place it in the same location as I did.



Test the game, and now the warp should take you to the new screen in the right place.

Animation Tools

Creating Animations

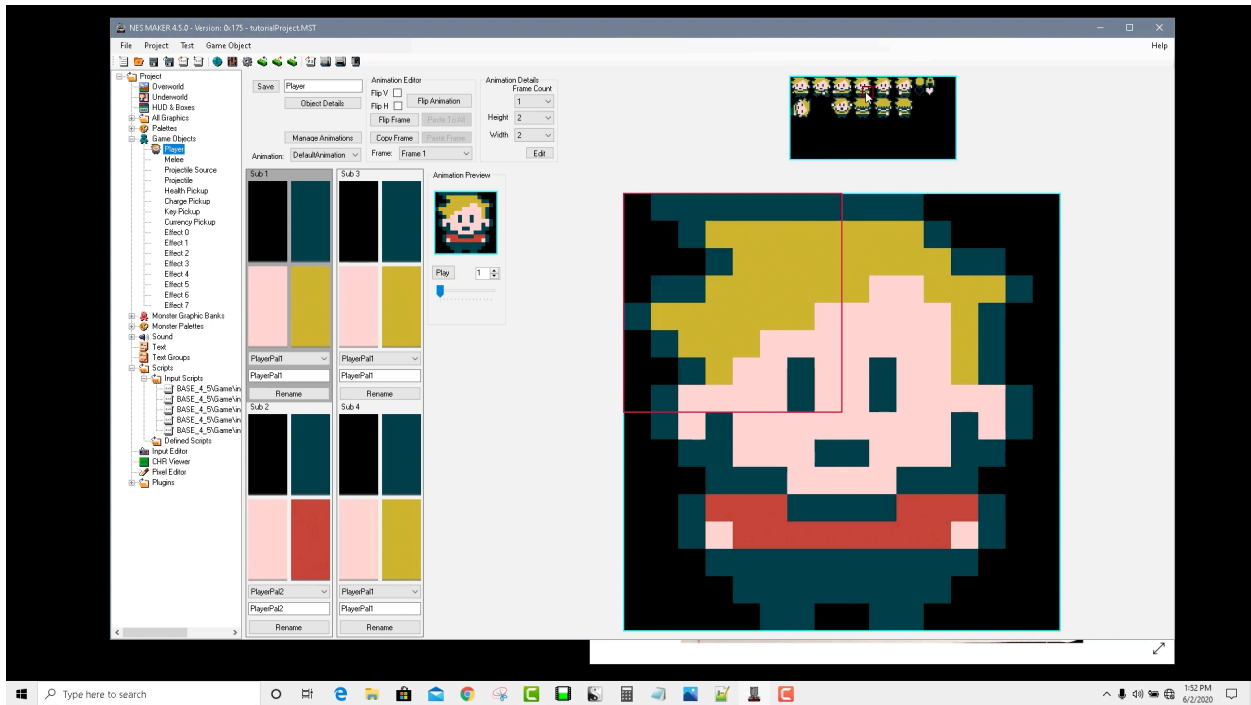
At this point, we now do have some things working. The player does move based on input. He does observe collisions with background tiles. He can warp between screens. But he's always facing the same direction, and always in a static

pose. For some games, this would work fine. Maybe you're creating a point and click adventure game, and this object you're moving around the screen is a pointer of some sort rather than a time traveling adventurer. But for Greg, we already have a tileset with different animations and different directions. We should definitely make use of them.

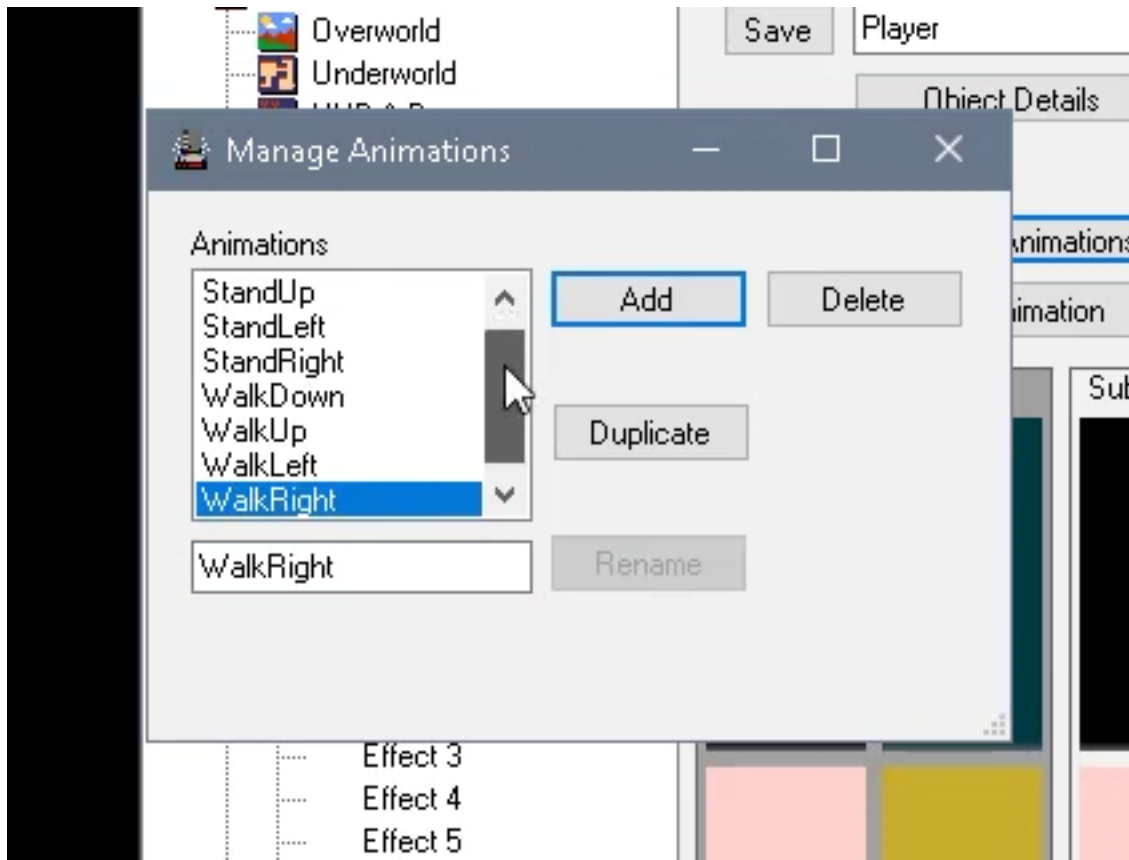
Step 1: Open the player Game Object. At the top right, you can see the current Game Object tileset and get a feel for the animations that are already created. For this game, we want an idle frame if the player is just standing there, and a walk animation if he is moving around. For this, we have two animation types; standing and walking. Each of those animation types will have four animations, one for each direction.

So effectively, we need to create animations for:

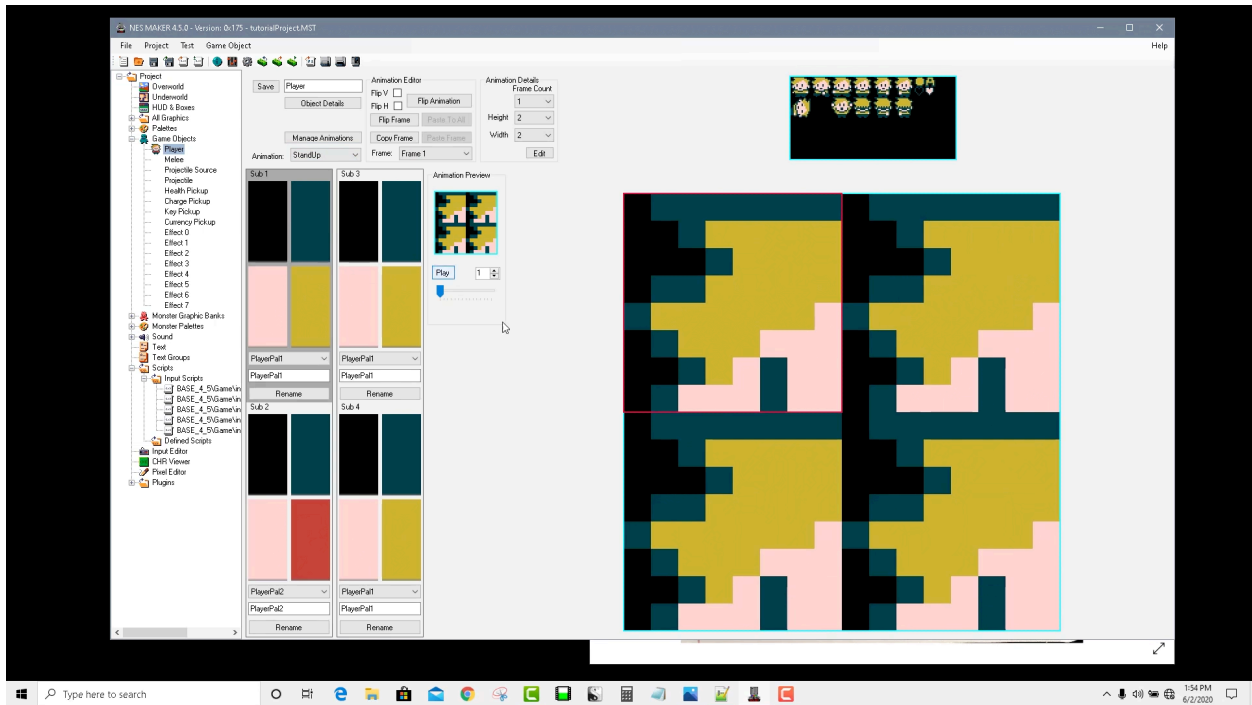
- Stand Up
- Stand Down
- Stand Left
- Stand Right
- Walk Up
- Walk Down
- Walk Left
- Walk Right



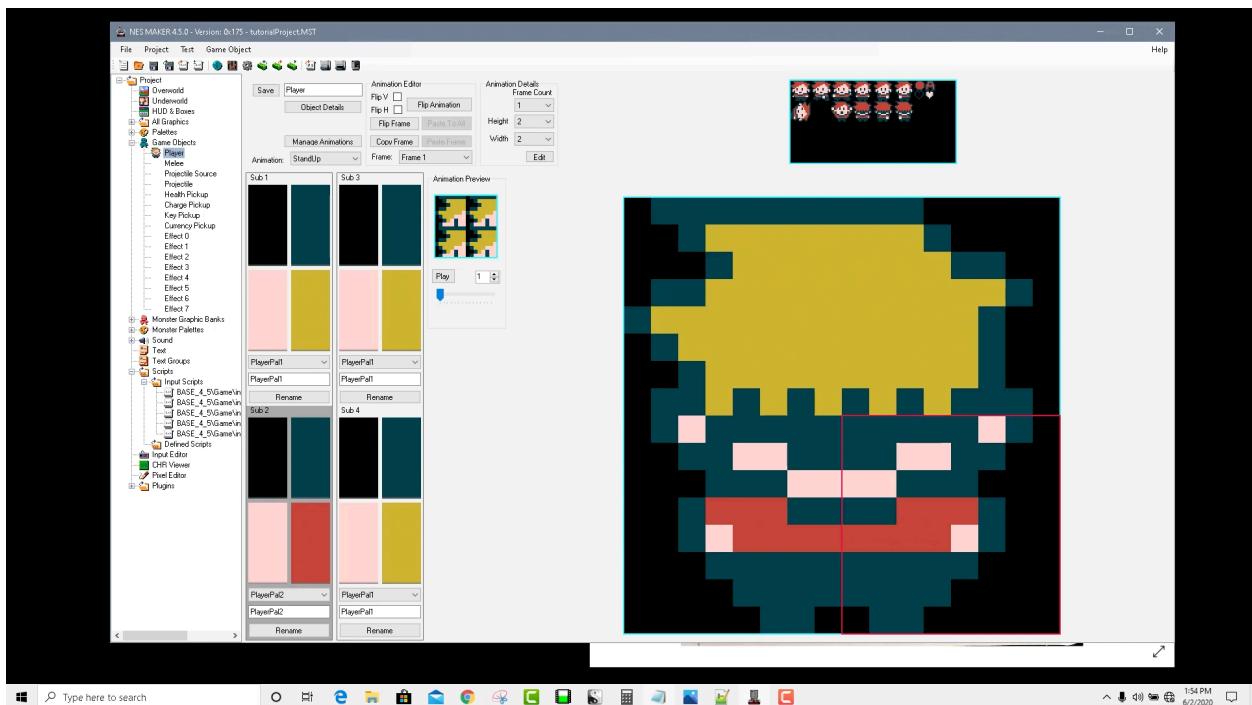
Step 2: Click manage animations. Rename the Default Animation to Stand Down, since that's what we're looking at on the screen. Then, proceed to create the other seven animations listed above.



Step 3: The Stand Down animation is what is loaded, and it is already finished. It's a one frame, 2x2 tile animation. The tiles and colors are already correct. But if you go to the Animation trio down and select Stand Up, you'll see that it's all images of the top left corner of his head. Why the top left corner of his head? Again, this is uninitiated, so all of the references are null, or zero. Zero would be the first tile in the tileset, which are the top left corner of his head.

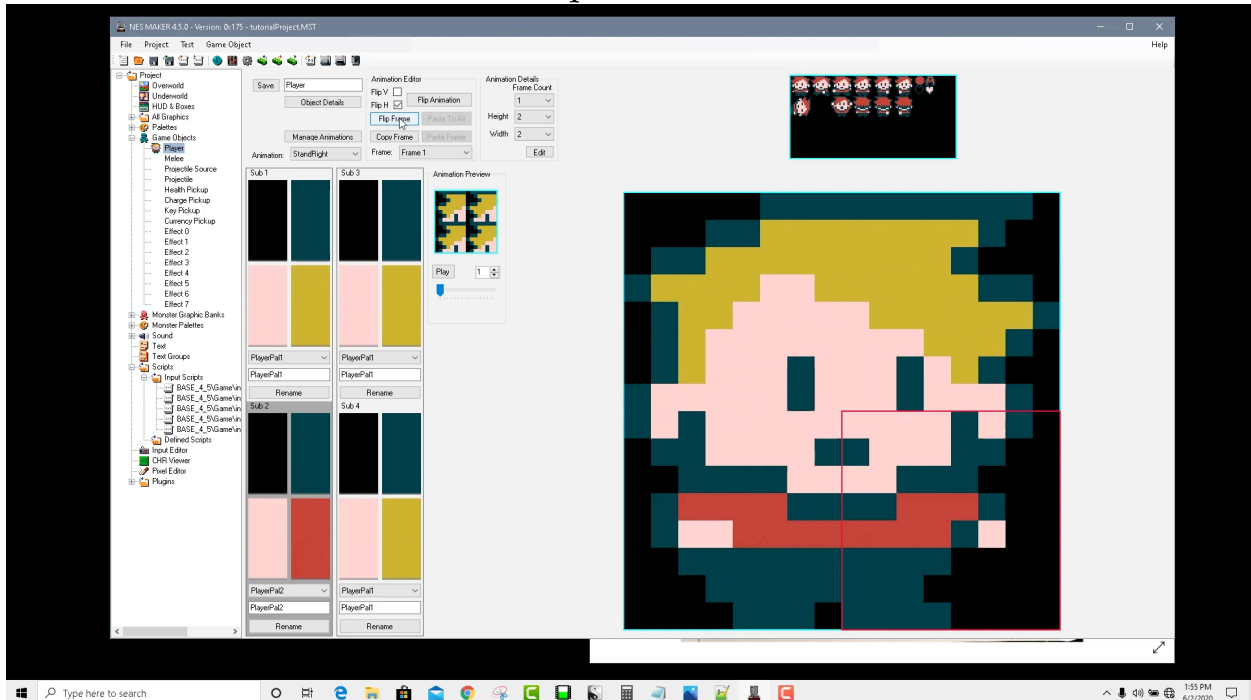


Step 4: Create the animation for Stand Up, and don't forget to change the palette for the bottom two tiles.

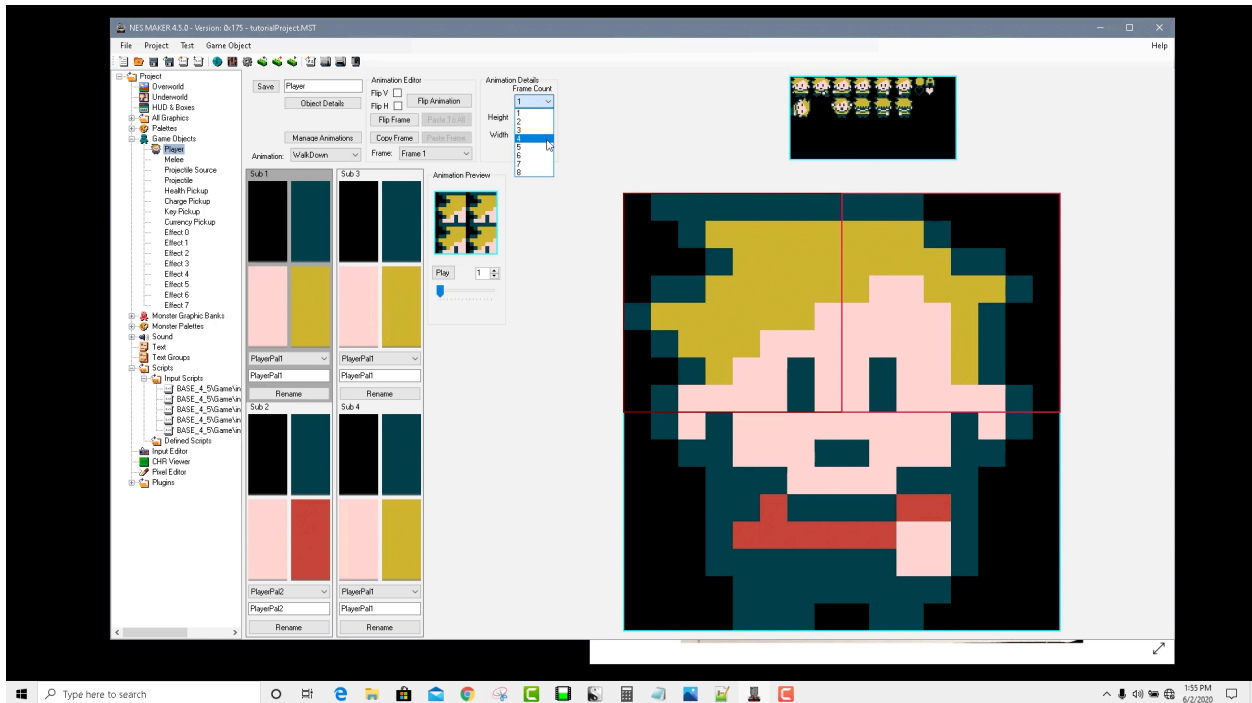


Step 5: Do the same for Stand Left. Select Stand Left from the dropdown, choose the correct tiles, update the sub palette information.

Step 6: Notice, there is no right facing sprite. What we'll do is exactly what we did for the left facing sprite, however when we are finished, click on Flip Frame. This flips the current frame of animation horizontally. Since this is a one frame animation, we're good. If we had a more complicated multi-frame animation that we wanted to mirror, we'd hit the Flip Animation button instead.



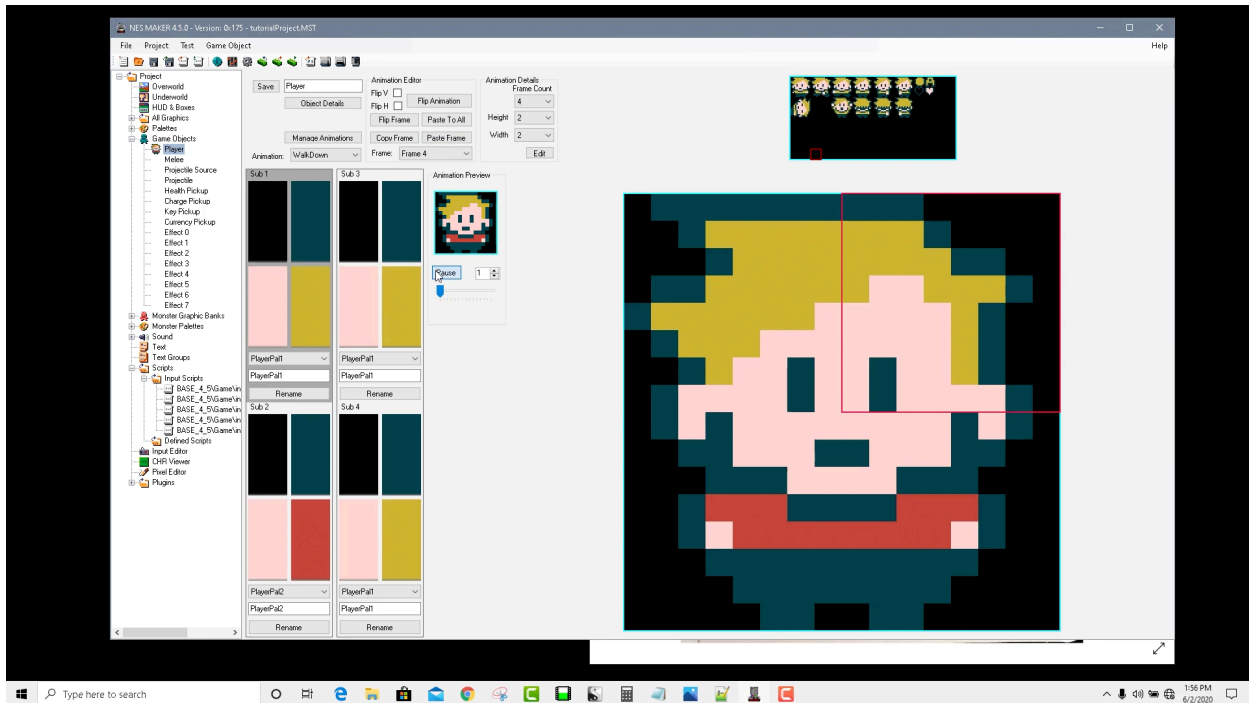
Step 7: Begin setting up a four frame walk cycle the same way. Use the dropdown to select WalkDown. Set up the tiles for your first walk frame. Since we want this to be a four frame animation, we need to go to the Frame Count in the Animation Details section and choose 4.



Now you can click the Frame drop down to see your various frames of animation.

Step 8: For frame 1, have one hand forward. For frame 2, have his idle stance. For frame 3, have his opposite hand forward. For frame 4, return to his idle stance.

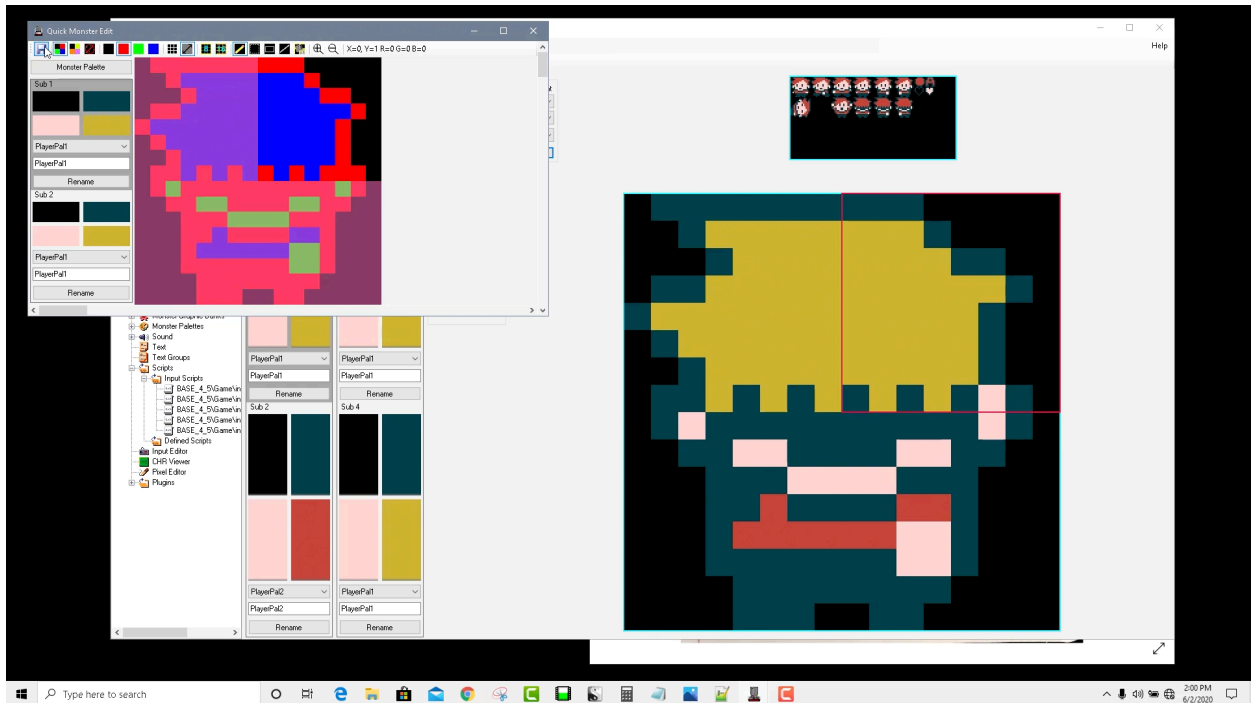
To preview this animation, you can click on the play button in the Animation Preview.



The speed of the preview has nothing to do with the speed that this player will animate in game. This is just for quick visual reference so you can see how this animation looks at various speeds.

Step 9: Repeat this process for walk up, using facing up tiles.

Also, a quick trick while you're in the animation tool. If you find that you want to make edits to the graphics in a frame of animation, you can actually make those edits without closing your object editor. By hitting on the EDIT button in the Animation Details area, you can bring up the current frame of animation in a breakout version of the pixel editor. When you make changes to these tiles, they will update the pixel editor accordingly. Now, this is changing the tile itself on the tileset, so if this tile is used elsewhere, it will also change in all places where it is used, but it is a great way to spot check animations if something doesn't quite look right or you need to make quick adjustments to the pixel art once you see it in the context of its animation.



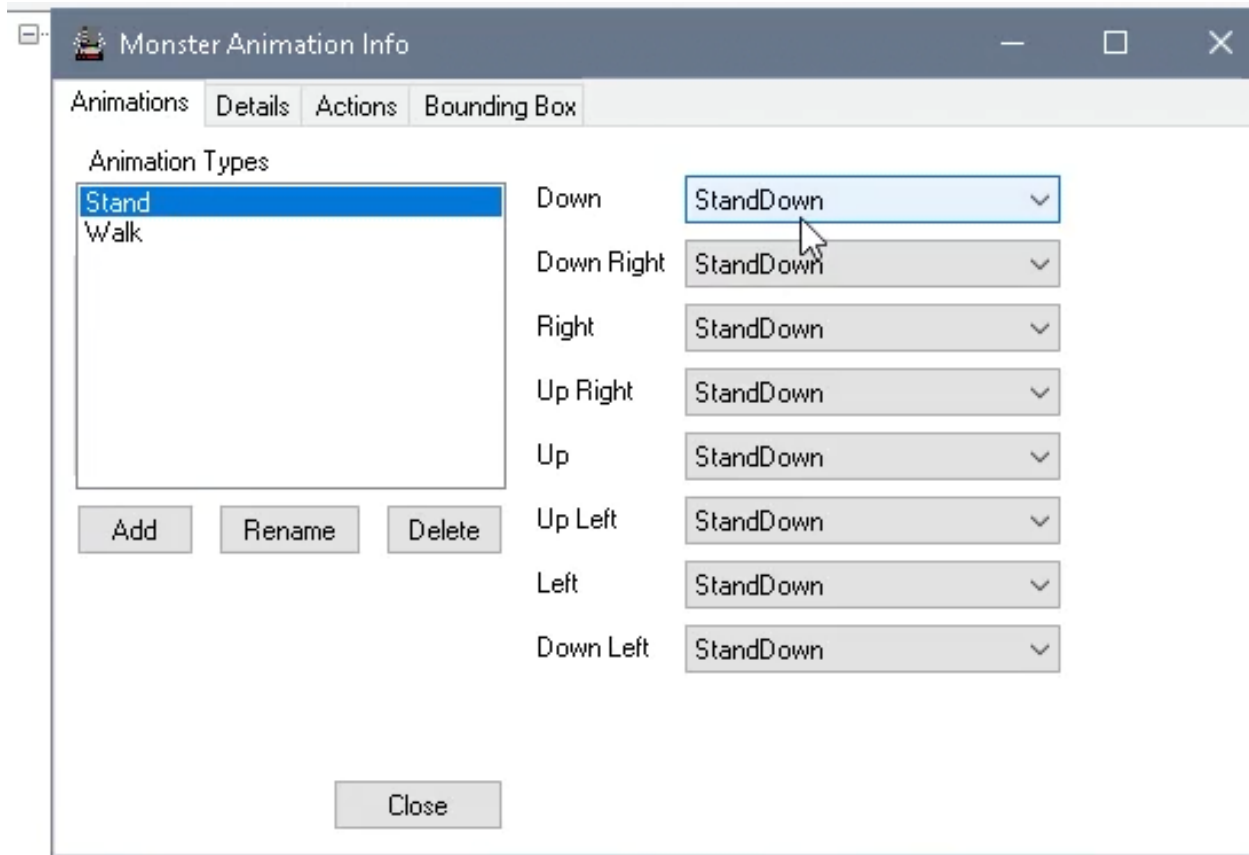
Step 10: Repeat the process for walk left, using the facing left tiles.

Step 11: For walk right, do exactly the same thing as walk left, however when you are finished with all four frames, you can hit the Flip Animation button to mirror all frames to facing right.

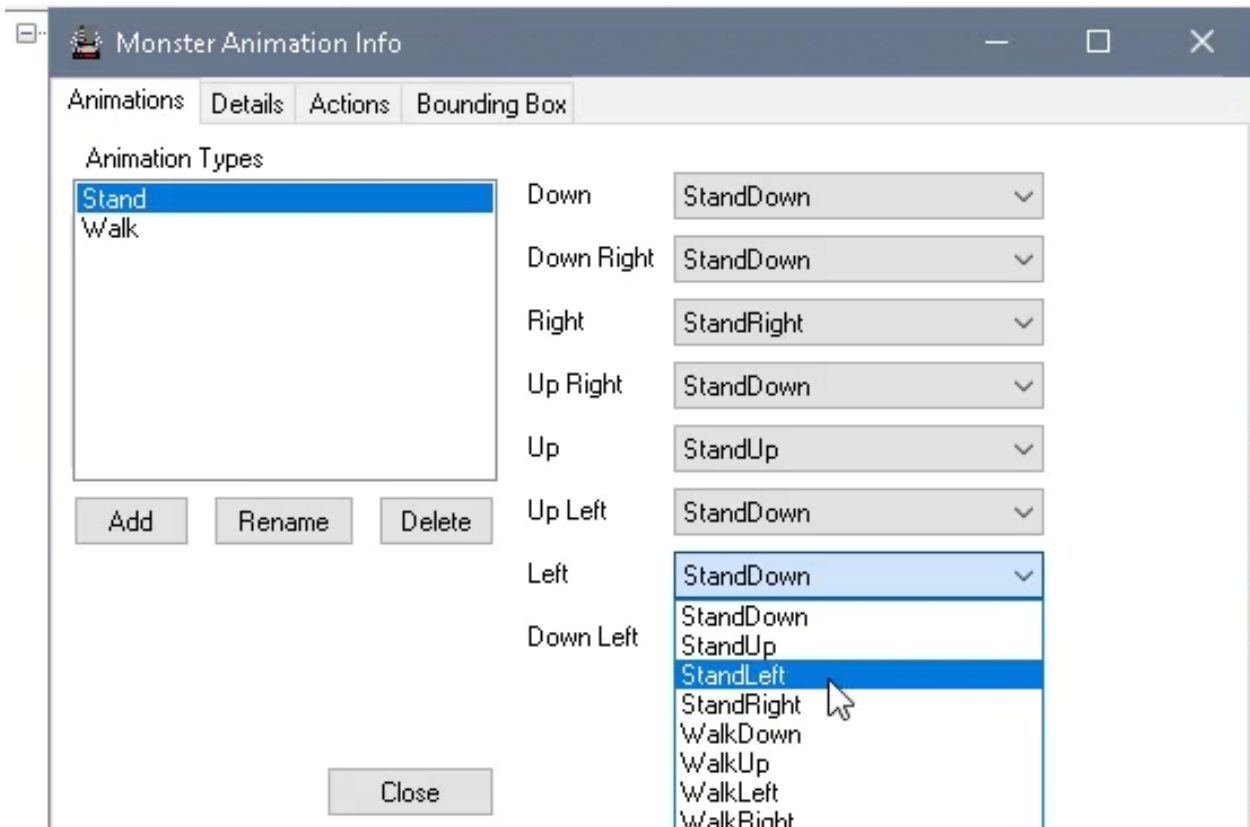
Now we should have two different animation types with four directions in each.

Step 12: Open Object Details. The first tab you see is the Animation tab, and it is showing you your animation types. We have allotted for two animation types, stand and walk, and we've created the various directional animations that go into each of these types, but now we need to actually create the types and assign the proper animations to each.

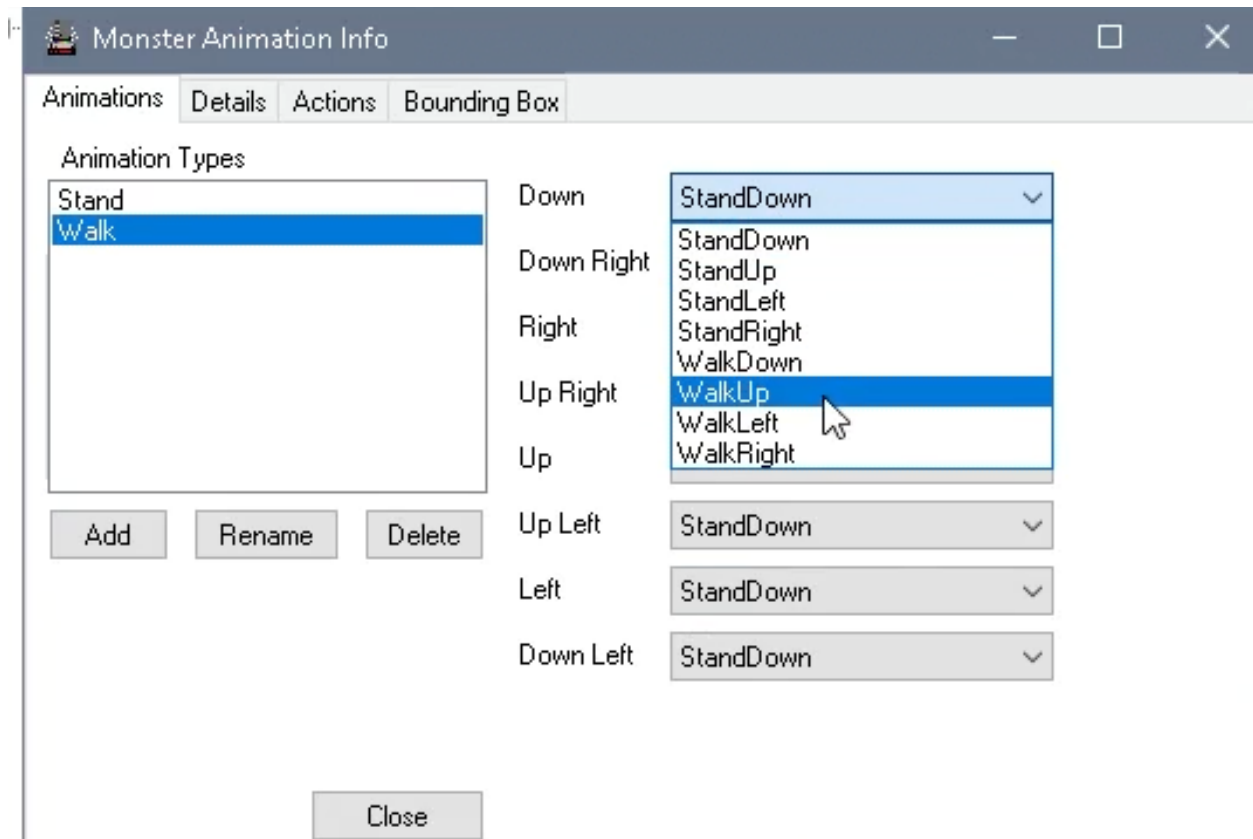
Rename Default to Stand, and Add Walk.



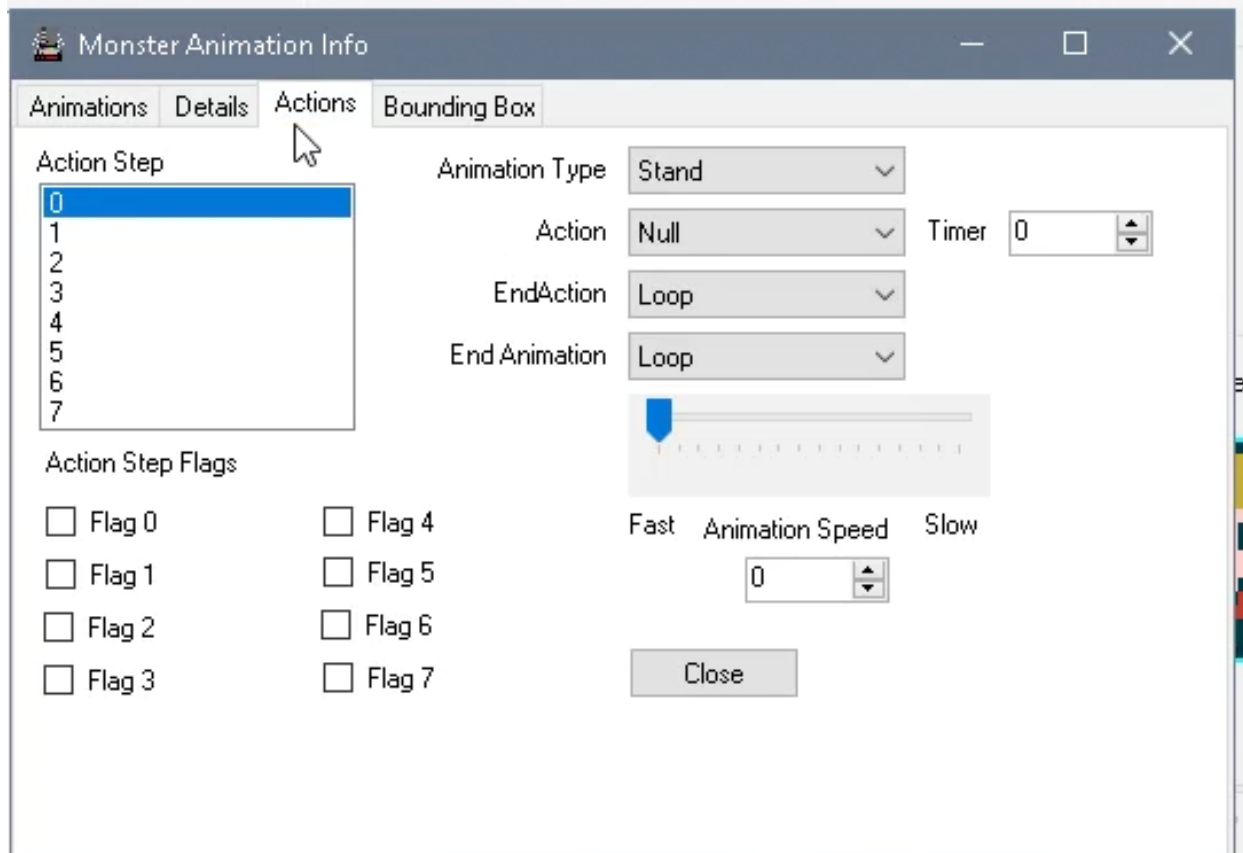
Step 13: Make sure that you have Stand selected. Then, in the dropdown to the right, select the proper animations for each direction (StandDown, StandUp, etc). What should we do for diagonals? Well, since our game doesn't have diagonals, these are not used, but it is still good to pick something logical for all directions.



Step 14. Make sure that you have Walk selected. Then repeat the process, only this time, choose the animations for Walk rather than the animations for Stand from the dropdowns.

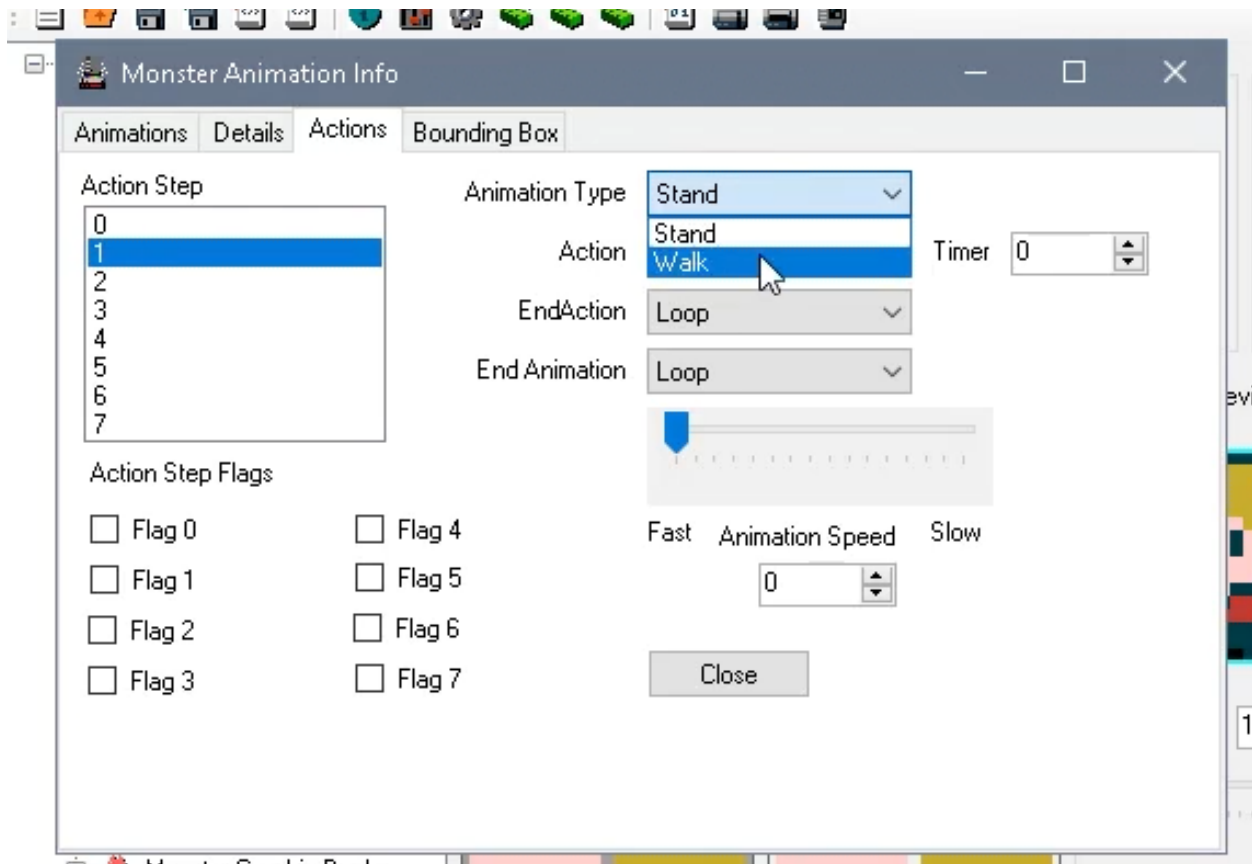


Step 15: Click on the Actions tab. This is the dialog that shows which animation type I am showing based on the behavior state that an object is in. When the object starts the game, he should just be standing. He should stay in this idle stand state unless input happens.

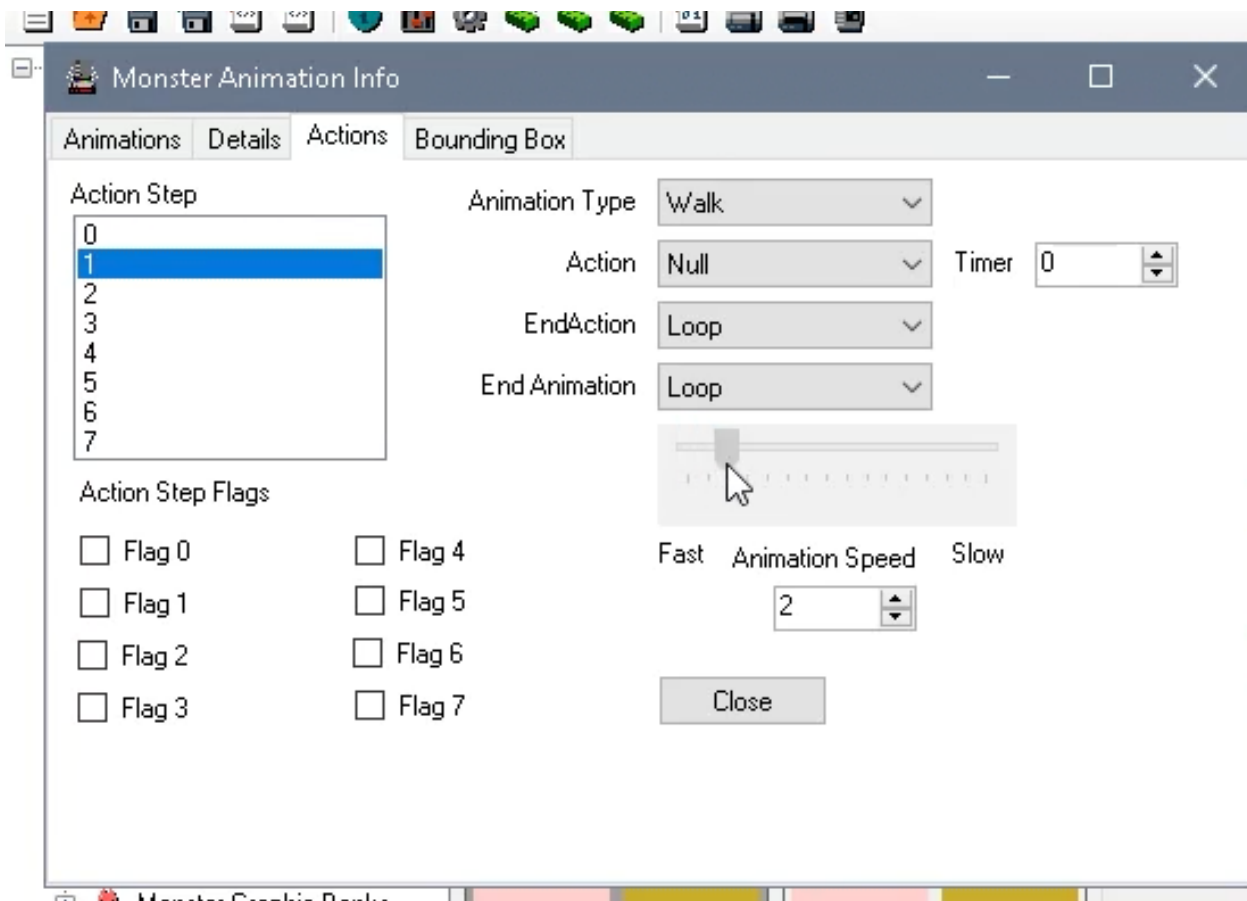


Step 16: For this project, we are going to use action step 0 for stand animation, and action step 1 for walk animation. This is arbitrary, as different games will have different needs for animation, however it's logical for our purpose here. Object zero is already set up to null values, which means that the stand animation, which was animation type 0, is already selected by default.

Make sure Action Step 1 is selected and then select Walk from the Animation Type.



Step 17: If we want to see the object animate, we need to make sure that there is a non-zero animation speed.



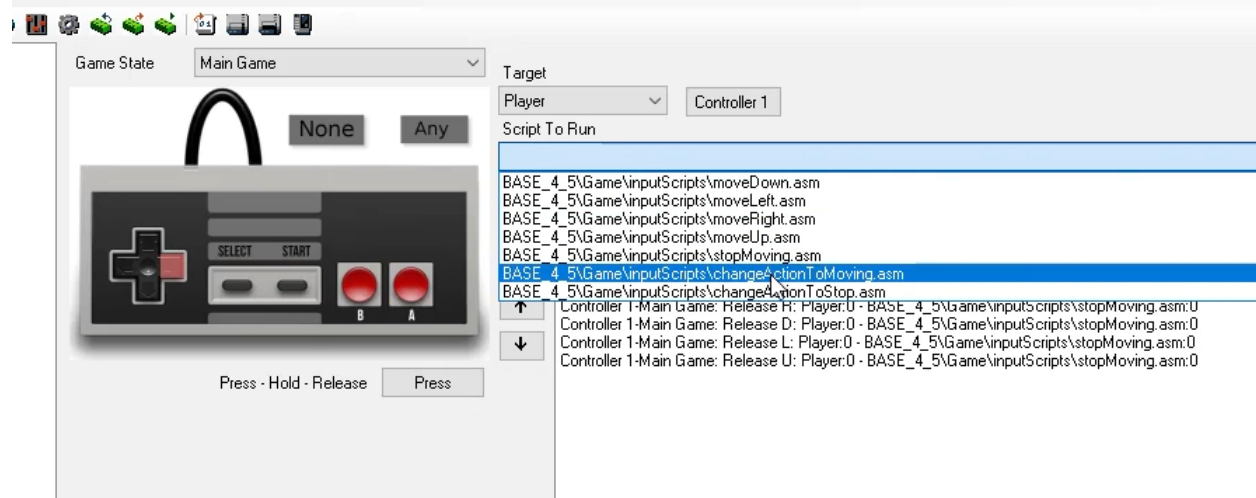
Press close and test your game. You'll notice he does change direction, but he's not animating yet. The reason is that we never told the object when to change action states. Some games will include this in their move scripts. Some have complicated checks to determine what action state should be showing. For our purpose, we need to conceive of when we want the action step to change to 1, and when we want the action step to change to 0.

We know it will have something to do with the input. If we were to think about it like a player, we would know that we want him to start walking as soon as the d-pad is engaged, and change to standing when we let go of the d-pad. However, if we change to walking animation during the d-pad button hold input, that means every step that the button is being held, it will try changing to walking, which means the animation will stay perpetually stuck on frame one because every frame it's resetting its animation. A few fun logical ways around that might be to check first to see if it's already action step 1, and if it is, skip changing it to step 1. Another even easier way is simply to add some d-pad press and release events. So

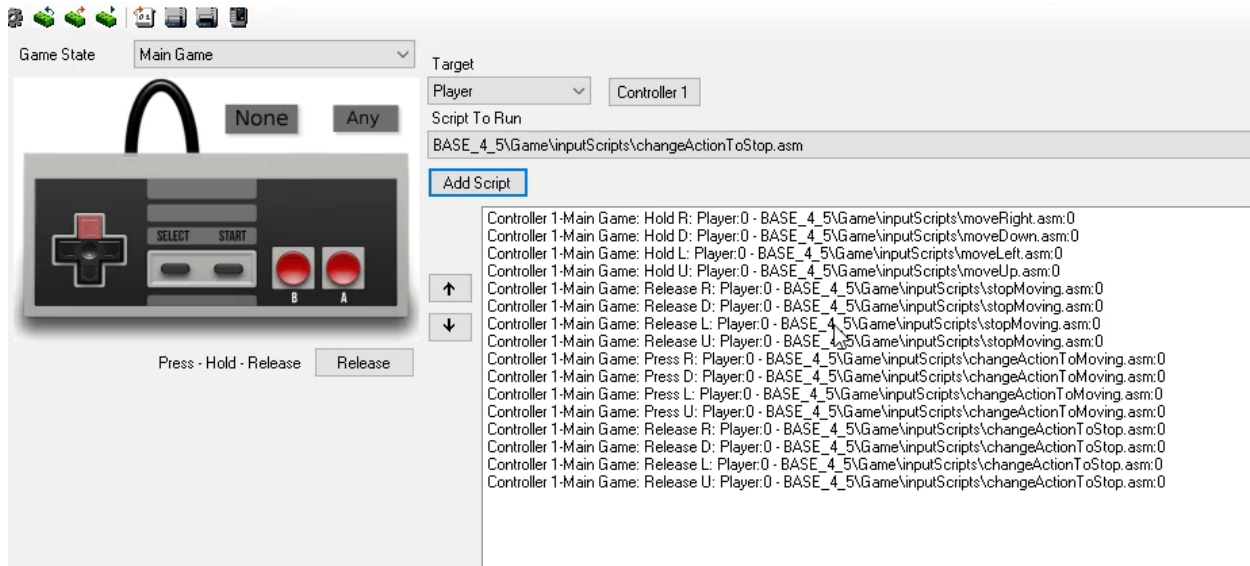
while the movement will be handled by the hold and release, the actual state change will be handled by press and release.

Step 18: Click on Input Scripts in the hierarchy. Navigate to root / Game / Input Scripts. You'll see ChangeActionToMoving and ChangeActionToStop. These are very specifically designed for this module, meant to change the action step to one and zero respectively. Double click on each of these to add it to the project's input scripts.

Step 19: Go to the Input Editor in your hierarchy. Add a new input: If you're in the main game, for your player object, with controller 1, when you PRESS the right arrow key, ChangeToMoving. Repeat this for all four directions.



Step 20: Add when you RELEASE each arrow key, ChangeToStop to all directions.



Test your game, and you should now see your player animate while he moves and properly changes direction.

Monster Objects

Monster Objects

We've learned how to add graphics, how to paint screens, how to set up objects and give them input. We've added tile collision and animations. We can test our work so far in our emulator. But so far, it doesn't feel like a game yet, and the reason is that there are no obstacles and no clear goal. Right now, getting to the staircase seems an implicit goal based on a collective appreciation of decades of gaming vernacular, but avoiding static spikes to get there certainly doesn't feel

very game-like. A game, especially a game in this style, really needs antagonists to avoid.

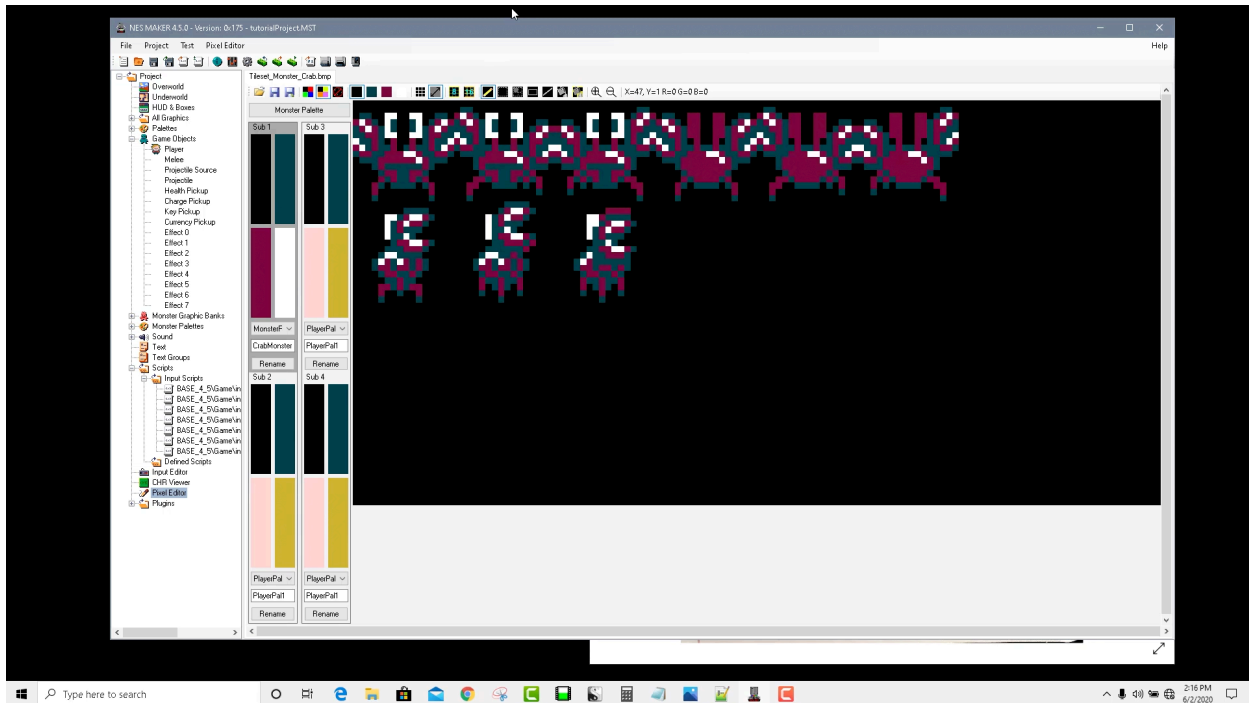
The good news is creating monsters utilizes many of the techniques we've already learned. We obviously need to create or import graphics for them. The monster object tool is almost identical to the game object tool. The real primary difference between the player object and the monster object is that the player object is controlled with input, where a monster object needs to utilize AI.

Setup the Monster's Graphics

I will quickly brush through some of the things that we've already covered. Feel free to reference previous sections of this instructional if you need a reminder of how to do particular tasks.

Step 1: Open the pixel editor tool and click open. Navigate to the root folder / TutorialAssets / BetaTutorialAssets / Graphics / TutorialGraphics, and open the file called Tilset_Monster_Crab.bmp.

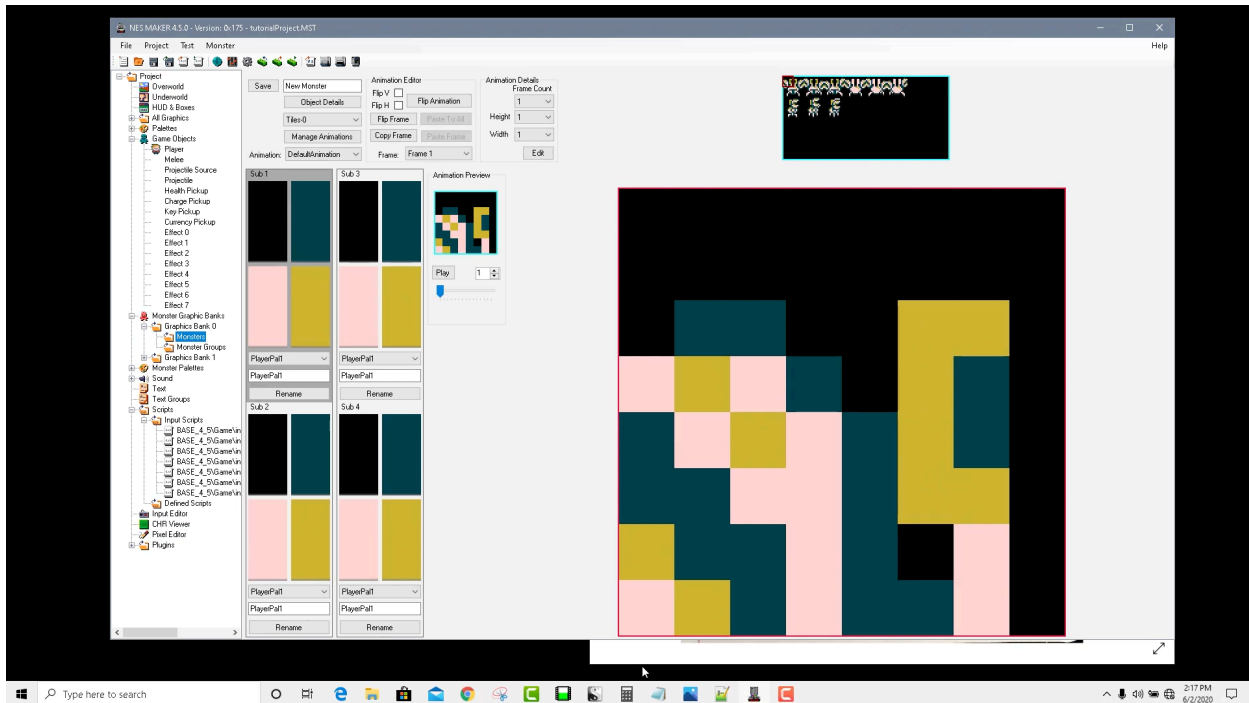
Change to object palettes, and from one of the palette dropdown, choose a new palette that we haven't used or given name. Name it CrabMonsterPal and hit rename. With enable palette translation selected, pick a color scheme you like for this monster.



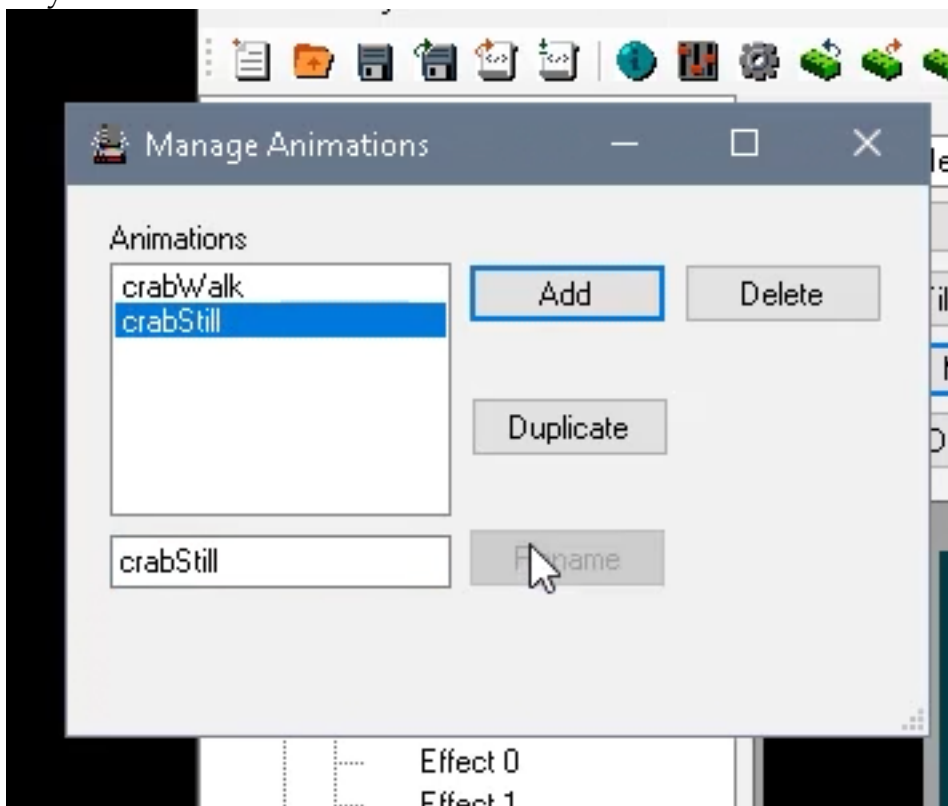
Step 2: Since this canvas happens to be the exact right size for a monster tileset, press Save-As and save over the top of Monster_0_00.bmp. Alternatively, you can open Monster_0_00.bmp in a separate tab, then copy and paste the crab graphics from the first tileset into the Monster_0_00 tileset. This would be necessary if the crab monster graphics were on a canvas that was the wrong size, but since it's a perfect match in terms of size, it will have the exact same effect.

Step 3:

On the hierarchy, expand Monster Graphic Banks, expand Graphics Bank 0, and click on monsters. This will bring up an object (monster) editor that looks identical to the one we used to create our player, except by default it is showing our first monster tileset instead of our game object tileset.



Step 4: Using the same method as with the player, give him two animations, one called CrabWalk and one called CrabStill. The names are incidental, as long as you remember one is for his movement and one is for his idle state.

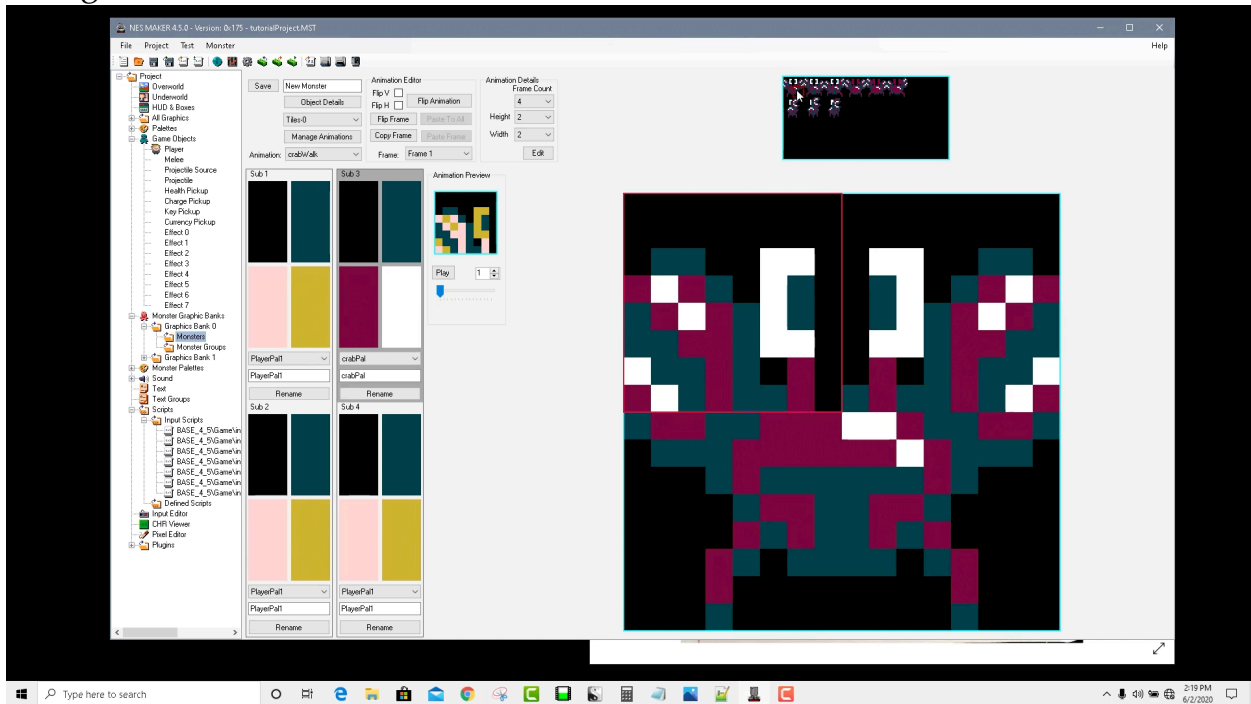


Step 5: Make this monster 2x2 tiles, and give this crab walk animation four frames.

Considering that when we designed our screen, we used sub palette 1 for the top half of our player, and we used sub palette 2 for the bottom half of our player, the logical thing is to use sub palettes 3 and 4 for our monsters. So load our crab palette that we created into sub palette 3 and make sure it's selected as we started editing our crab monster object.

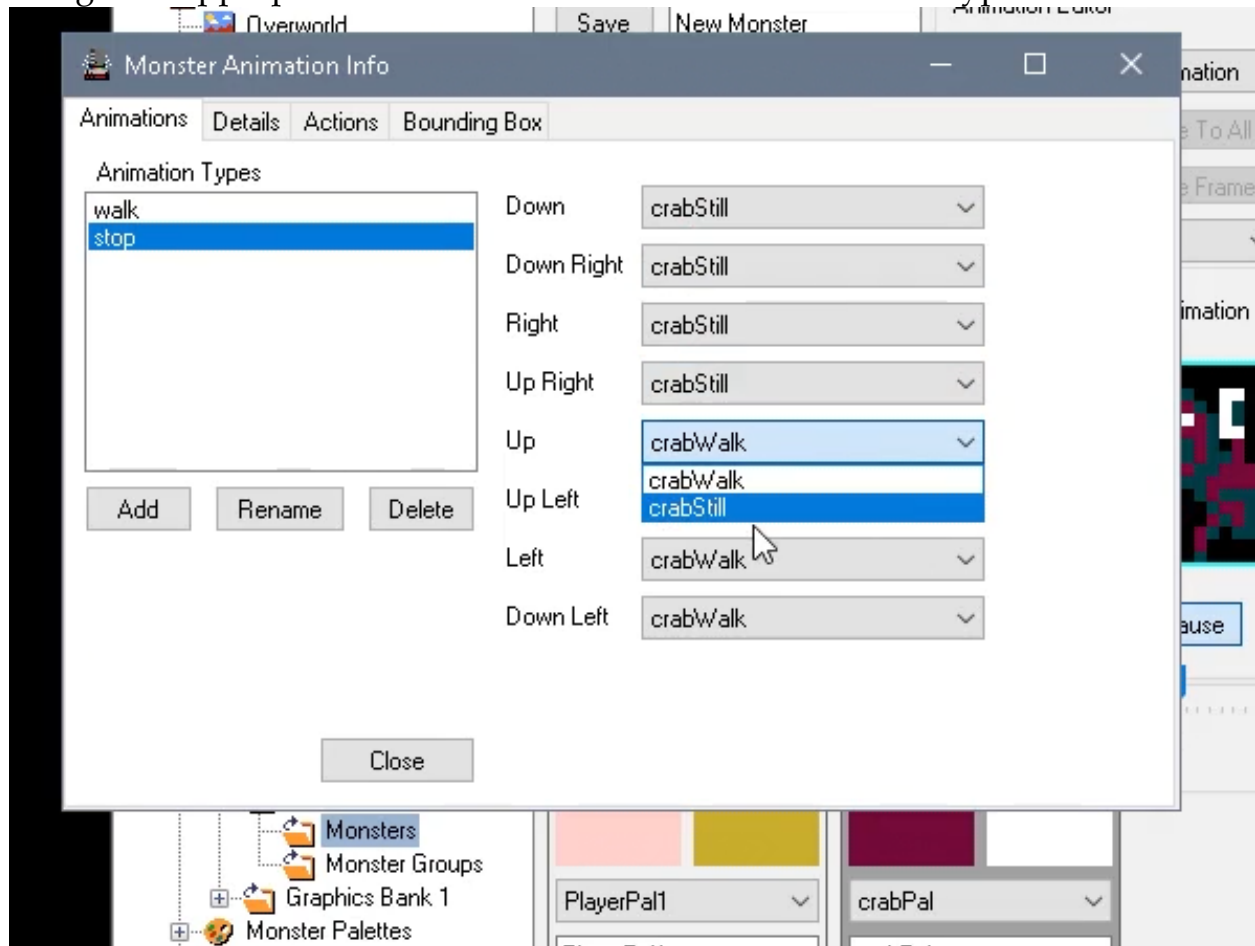
Make four frames of crab animation with sub palette 3 selected.

We could make animations for all four directions, but considering how real life crabs scatter in a ll directions, this monster works pretty well with only one facing direction.

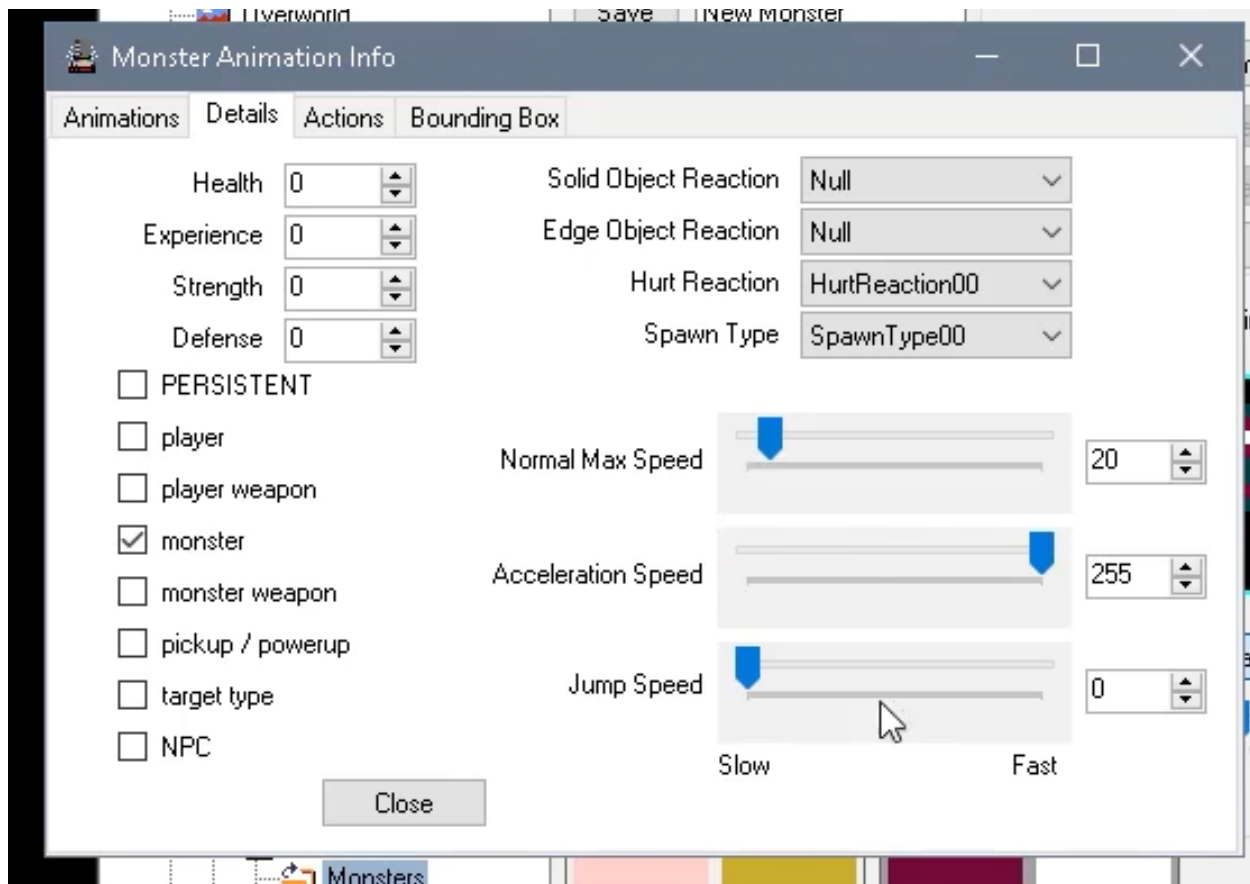


Step 6: Choose the CrabStill animation type from the drop down list, and with sub palette 3 selected, make the still frame for the crab. This is just one frame of animation.

Step 7: Open object details and make two animation types, walk and stop. Assign the appropriate animations to the different animation types.

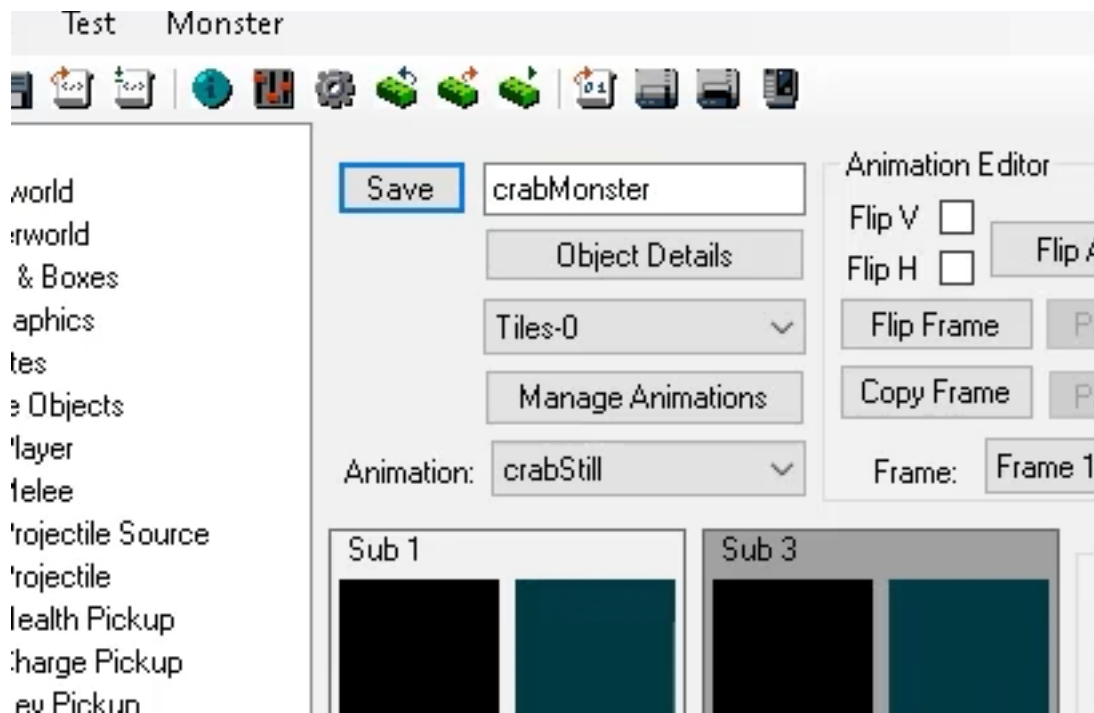


Step 8: In the Details tab, set the object type to Monster, and give him a speed and an acceleration speed.

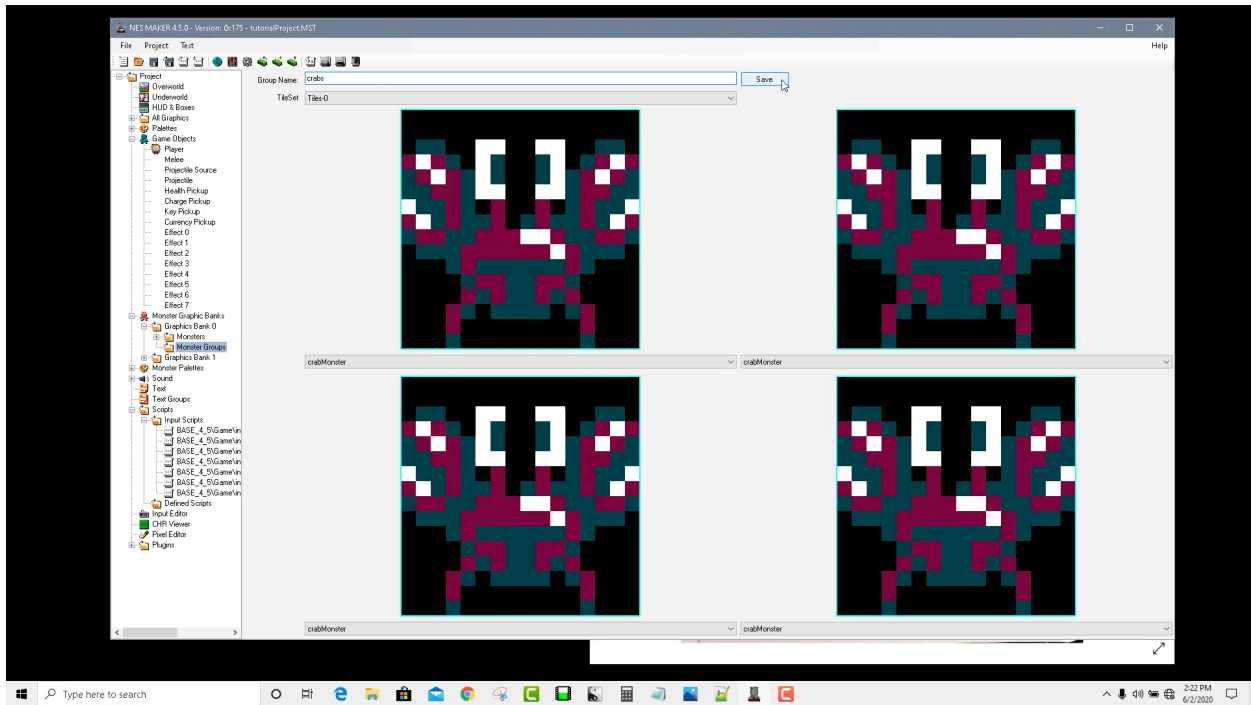


Step 9: In the bounding box tab, give him a proper bounding box.

Step 10: Very important, before leaving this editor screen, give him a name and hit save.

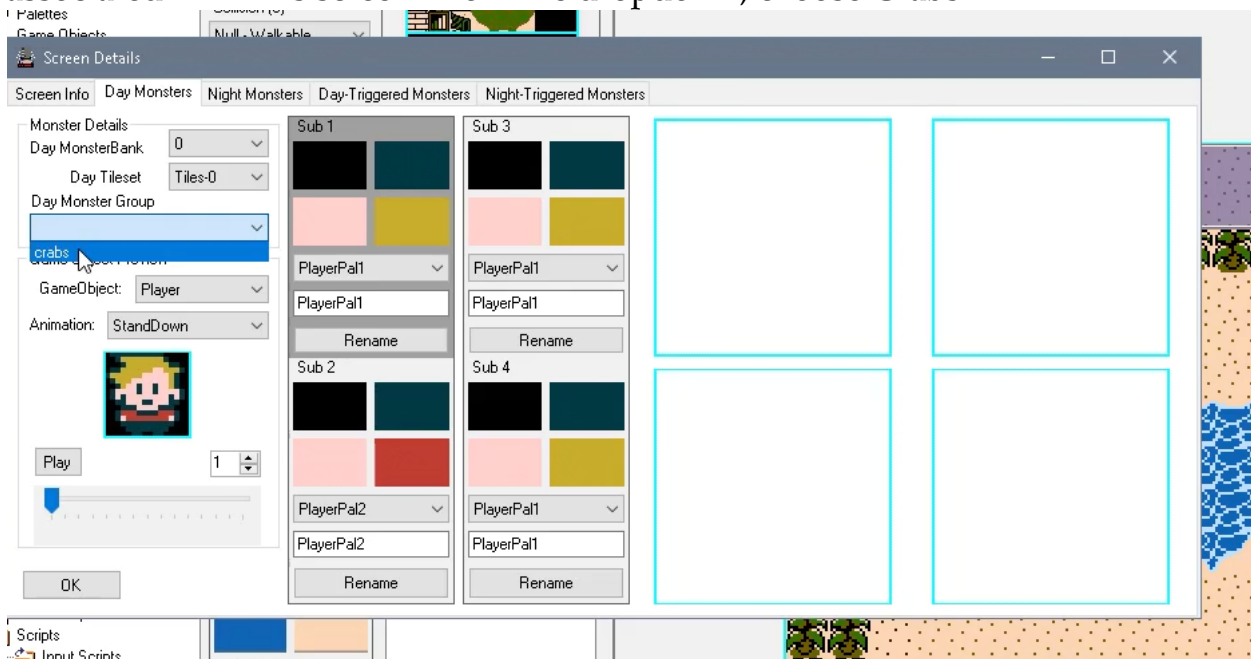


Step 11: After you've saved the monster, in the hierarchy hit Monster Groups. For each of the available slots, pick the only choice available (which will be whatever you named this monster object). Give the group a name and hit save. This particular group, when loaded to a screen, will let the screen know that it will be all crabs. Maybe later you create different monsters, and have different monster groups that mix and match objects that use the same tileset.

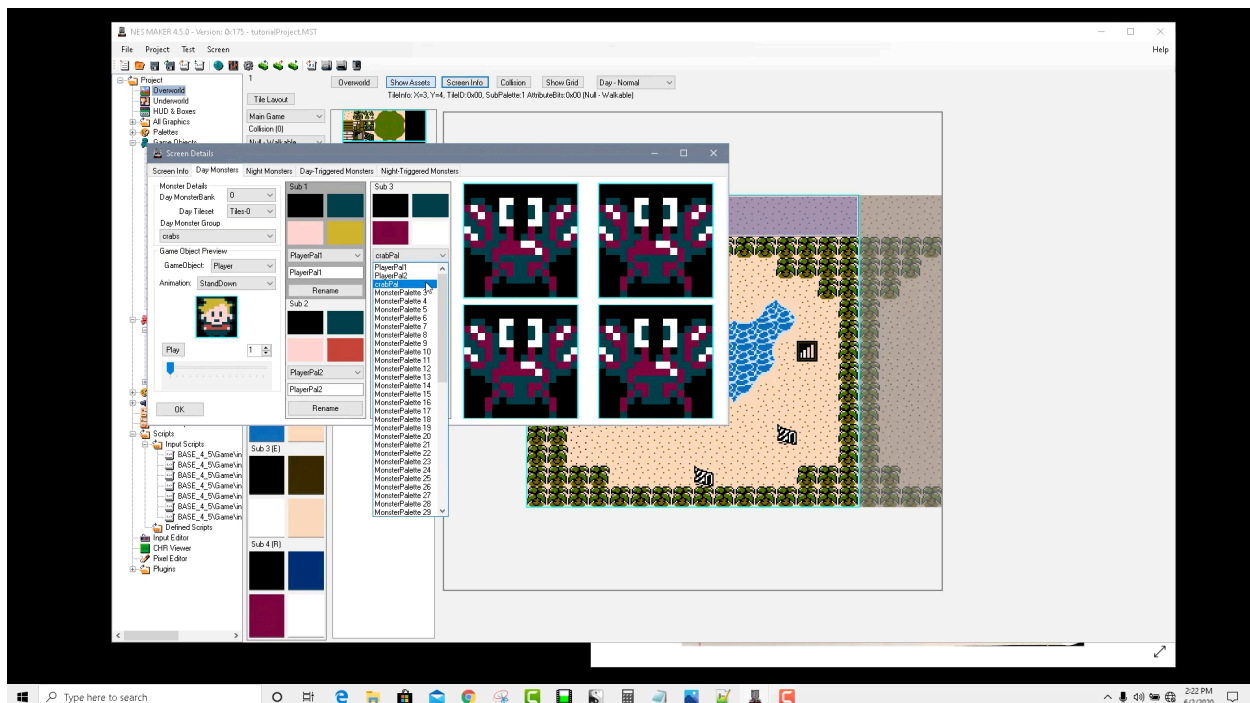


Step 12: In the hierarchy, click on overworld and open the screen where you'd like to place the monsters.

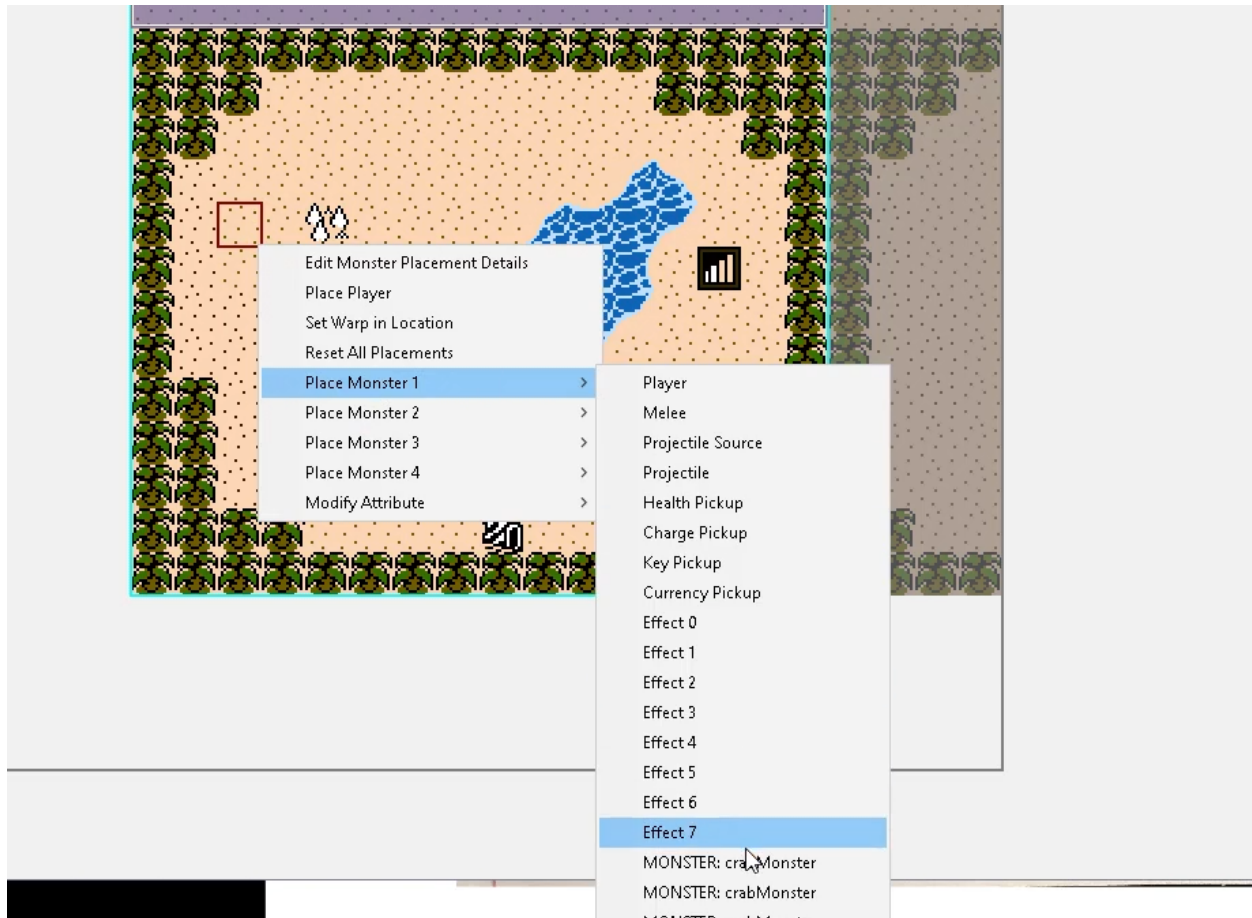
Step 13: On that screen, open the Screen Info and navigate to the Day Monsters tab. This is where you will tell the screen which monster group will be associated with this screen. From the dropdown, choose Crabs.



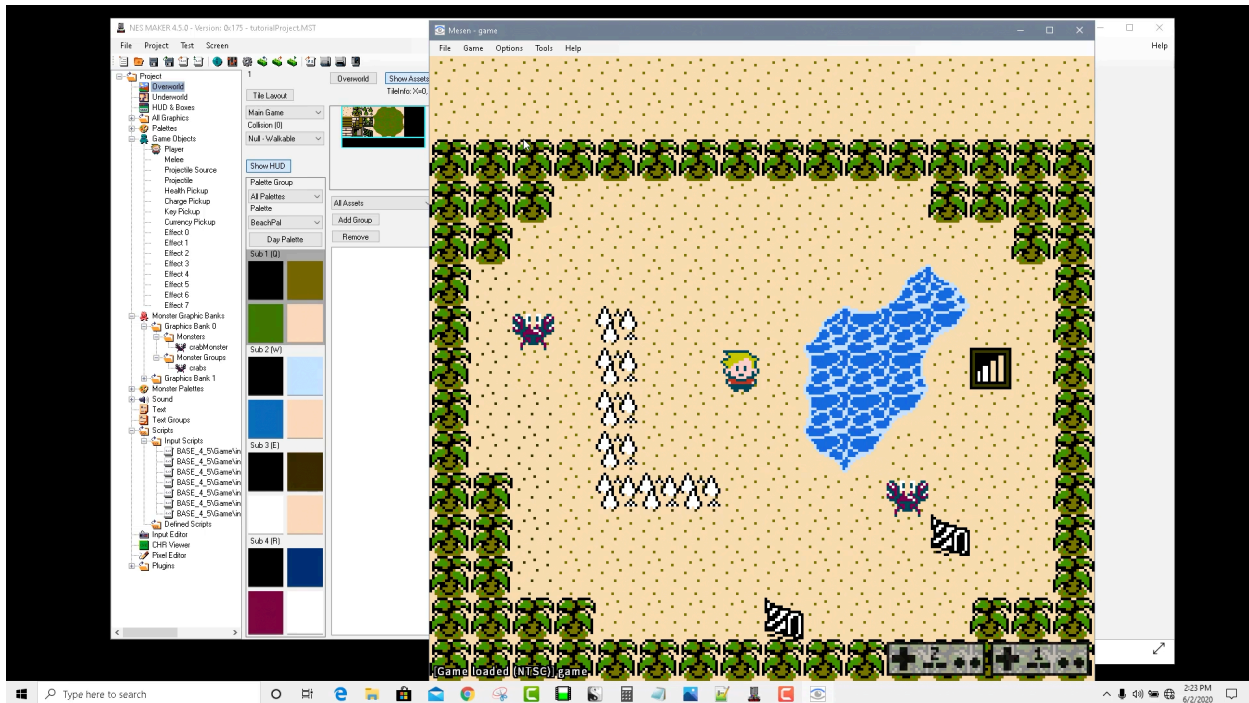
Step 14: Remember, these monsters use sub palette 3, and they have a palette called crab palette. So make sure to change sub palette 3 to crab palette. Because we used sub palette 3 for them when we were creating their animations, they will use whatever values are in sub palette 3 on each screen.



Step 15: Right click on the screen where you'd like to place a monster. Choose Place Monster 1, and you'll see a list of all of the objects you can place on this screen, which includes any game object and all monsters that use the currently loaded tileset. For Place Monster 1, choose crab monster.



Test your game and you should see the monster standing still at the correct coordinates.



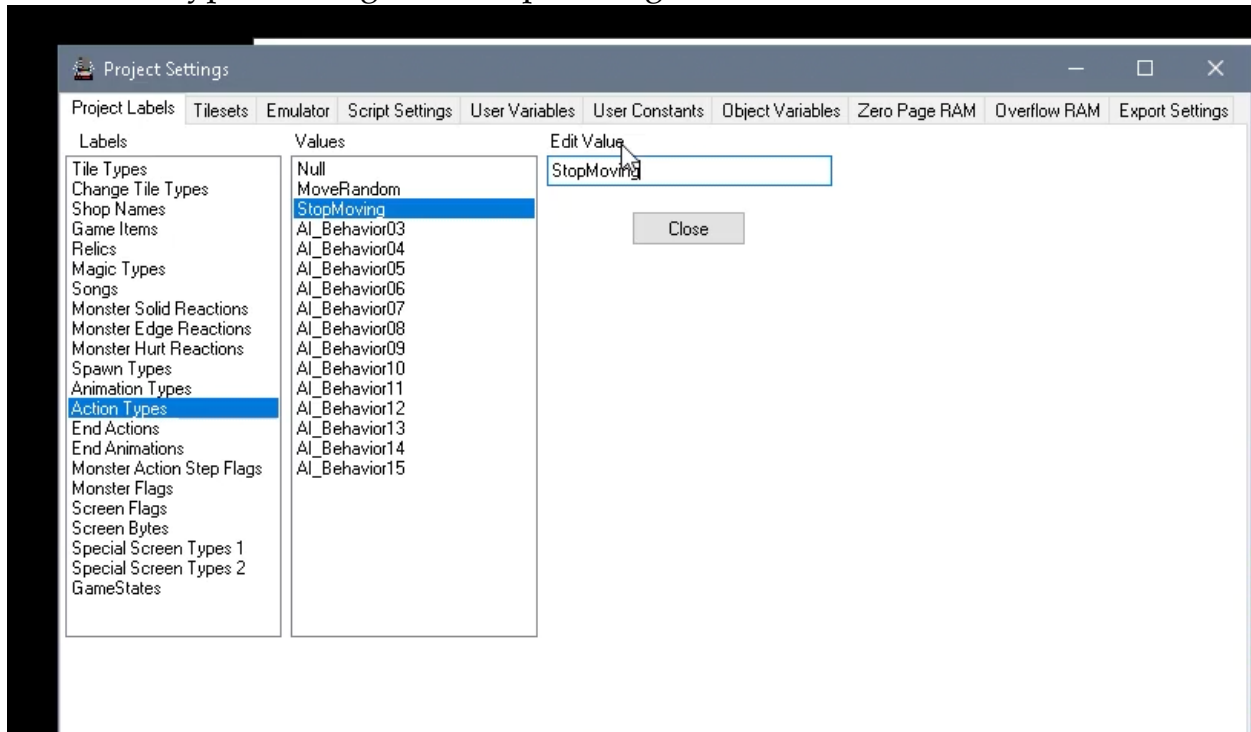
We have not determined what happens when the player runs into the monster. We have not determined the monster's behavior. We have not determined what happens when the monster runs into a solid object or screen edge. We have not given the crab a way to go through its action states. These are the things that are left to get this monster feeling like a true game monster.

Setting up Behaviors

We want these crabs to have a simple yet multi-layered behavior. We're going to make the crabs walk in a random direction, then stop for a second or so, then start moving in a random direction again. They will also have to reverse their direction when they hit a solid object or a wall.

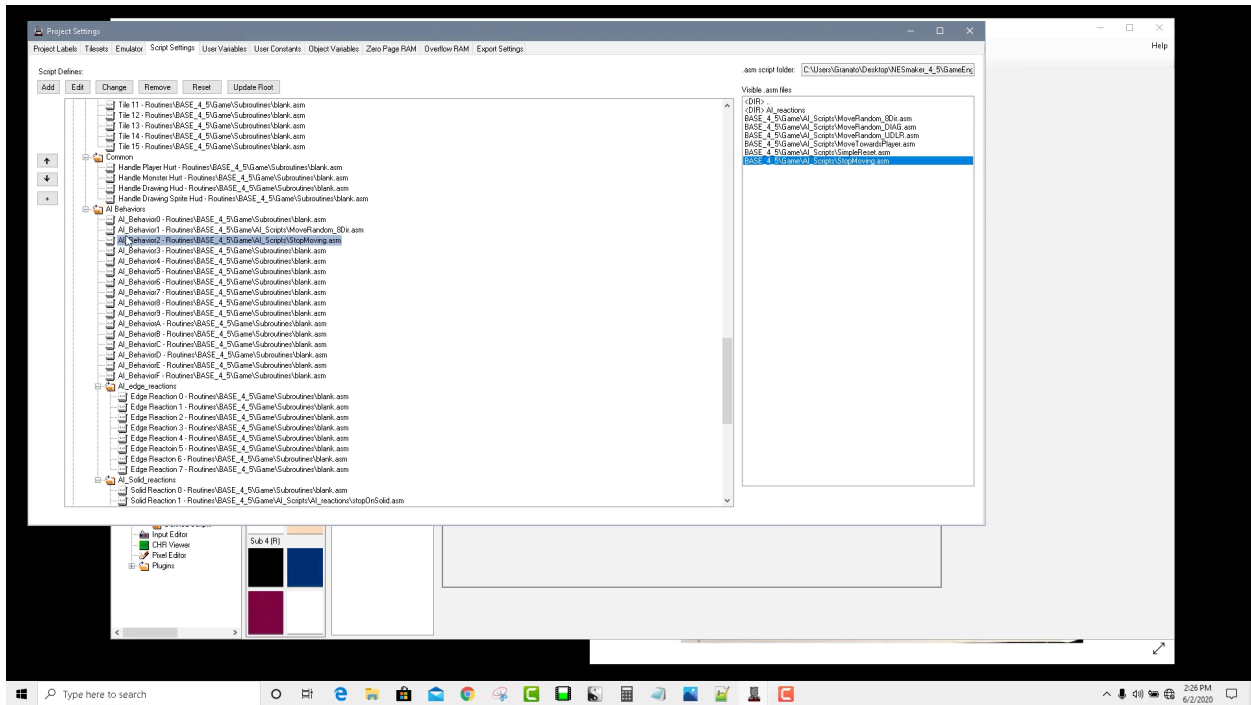
Step 1: Just like with tile types, we are going to establish a few labels for AI behaviors. Go to project settings from the gear icon at the top of the page. On the labels tab, click on action types. These are your AI controlled behaviors. We'll keep Action Type 0 as null. For Action Type 1, change it to Move Random, and

for Action Type 2, change it to StopMoving.



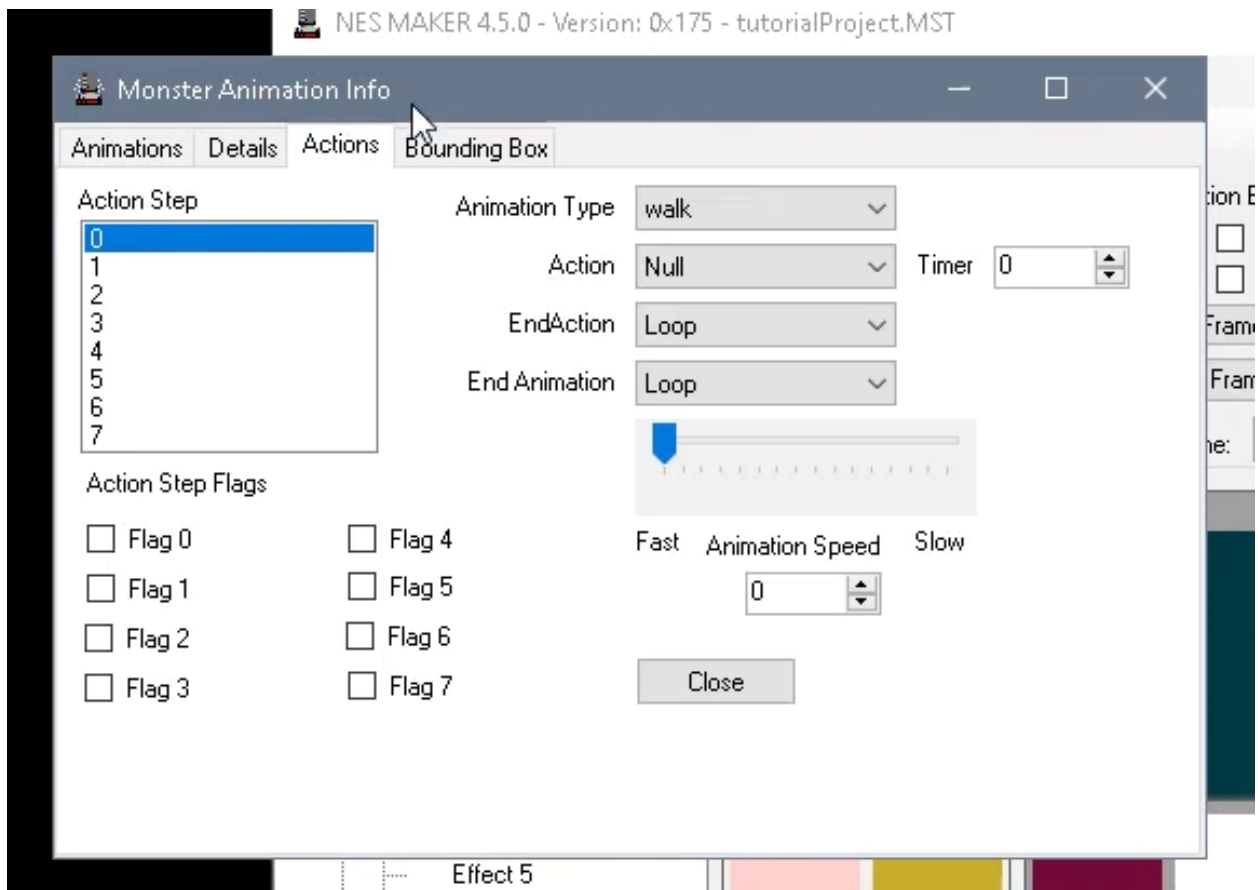
Just like with tile types, these aren't set up to do anything yet, these are just exposed labels. In the next step, we will attach scripts that will make these Behaviors behave in the expected way based on those label descriptors.

Step 2: Click on the Script Settings tab. Scroll down to the section for AI Behaviors. For zero, we will keep null as a blank script. That's what we want to happen for AI behavior 0 - nothing at all. Click on AI Behavior 1, then from the script finder window on the right, go to root / Game / AI Scripts, and double click on MoveRandom_8Dir. This will tell our engine that when an object invokes AI Behavior 1, it will pick a random direction and begin moving. For AI Behavior 2, double click on StopMoving.

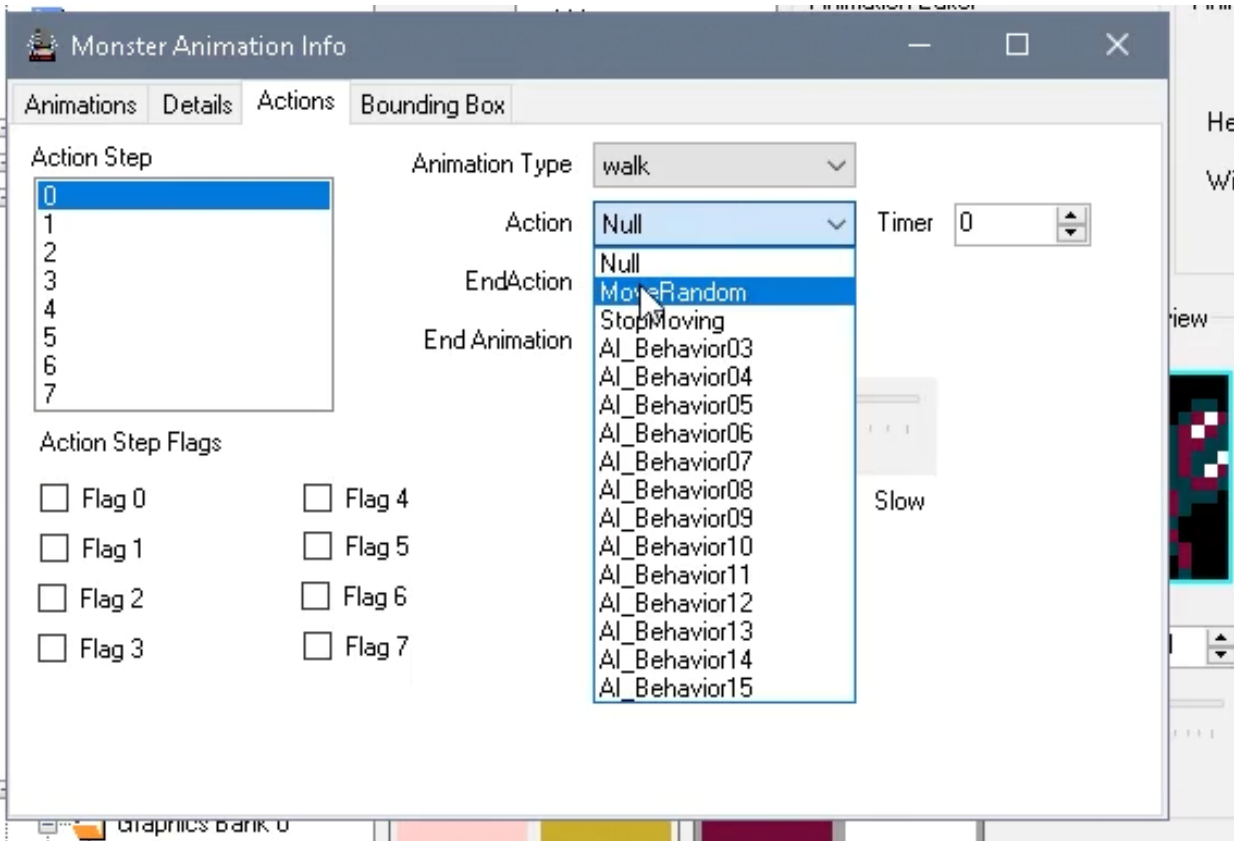


Now we have scripts in the right places that match the descriptive labels that we set up.

Step 3: Click on the crab monster in your hierarchy. Click on the Object Details button, and click on the actions tab.



Step 4: For action step 1, we will use the animation type walk, and we will give it a non-zero animation speed. Also, though, we want to give it an action. We want to set it up so that when the monster first appears on the screen, his first step will be to start moving. So we will choose for Action Step 1 the Move Random label from the drop down. Really, this is just telling the game “Do Behavior Type 1”, which we set up with a script to the Move Random script. The label helps us remember which is which in our custom shorthand.

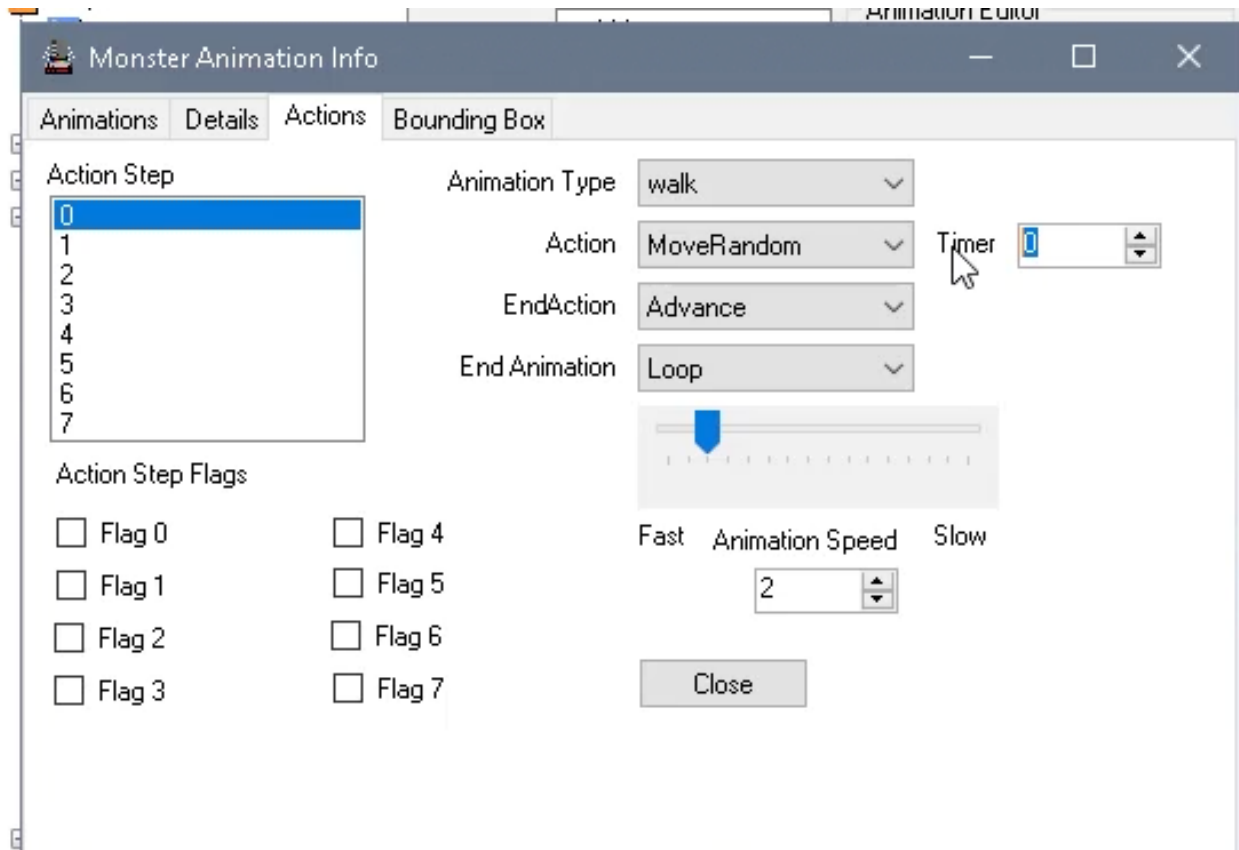


Step 5: Unlock with the player, we don't want controller input to inform the change to the next action step. There are a few ways we can tell it to push to our next desired action step. We can either check for the end of an animation cycle, or we can set a timer.

Setting an action timer to zero means that the timer will be set to a random number. If you want more regular timing, you can set the action timer to a non-zero number. The higher the number, the longer the timer, but non-zero numbers are each a strict number of frames. When the timer gets to zero, it will do whatever is in the End Action drop down. It could repeat the same action, it could move to the next action, it could jump to the first action, it could destroy this object. There are a lot of options here, and again, all of these can be set up through scripts for advanced users who want their own outcomes. Most times for AI, I just use zero.

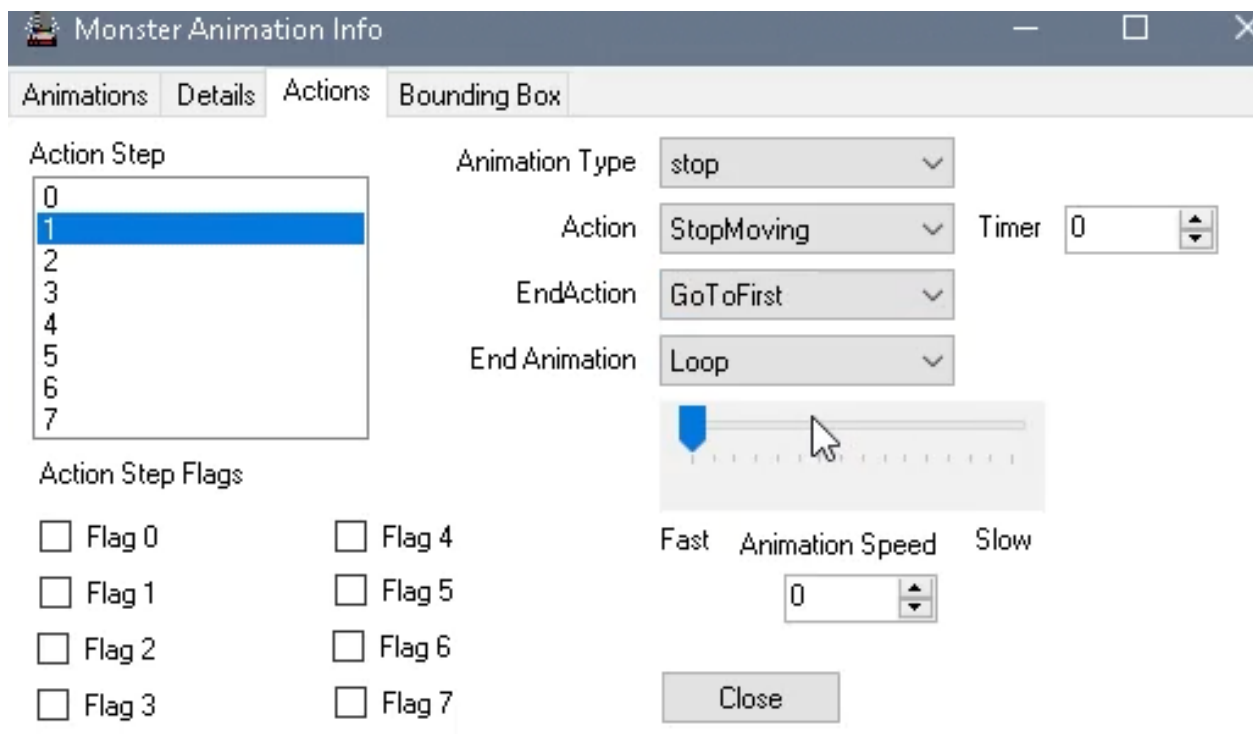
At the end of the action, I want it to Advance.

So now, when the crab shows up on the screen, he'll start moving in a random direction based on the speed we have set for this object. He will set a timer to a random number. When the timer reaches zero, his action step will jump from zero to one.



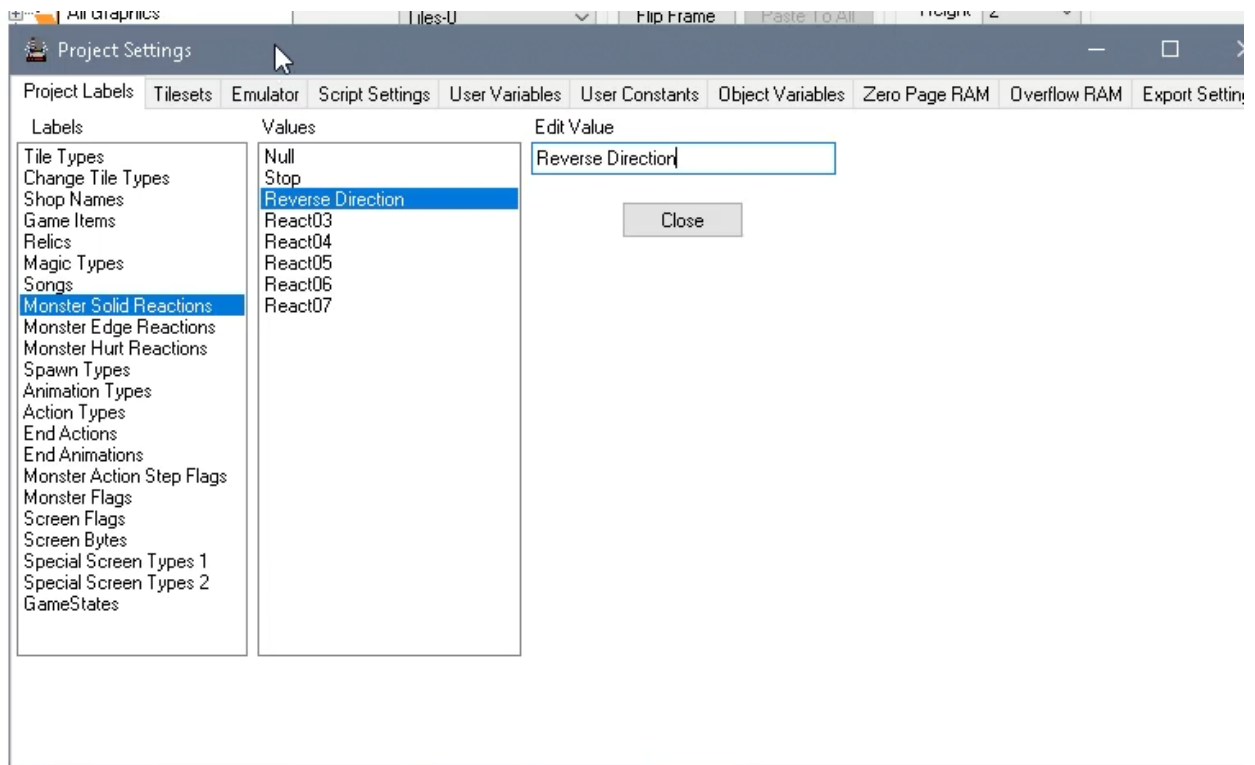
Step 6: When our crab monster hits this action step, I want him to stop. What does that mean? It means I want him to show the Stop animation type, and it means I want him to do the Stop Moving behavior. Also, though, I need it to set a timer so that it can continue, otherwise it will remain in that stopped state indefinitely. I want the character to go to the first action again when this action step timer goes off. This will create a loop where it will do action 0 for a random amount of time, then do action 1 for a random amount of time, then loop and repeat.

So for Action Step 1, set your Animation Type to Stop. Set your action to Stop Moving with an action timer set to 0 (random). Set the end action to GoToFirst.

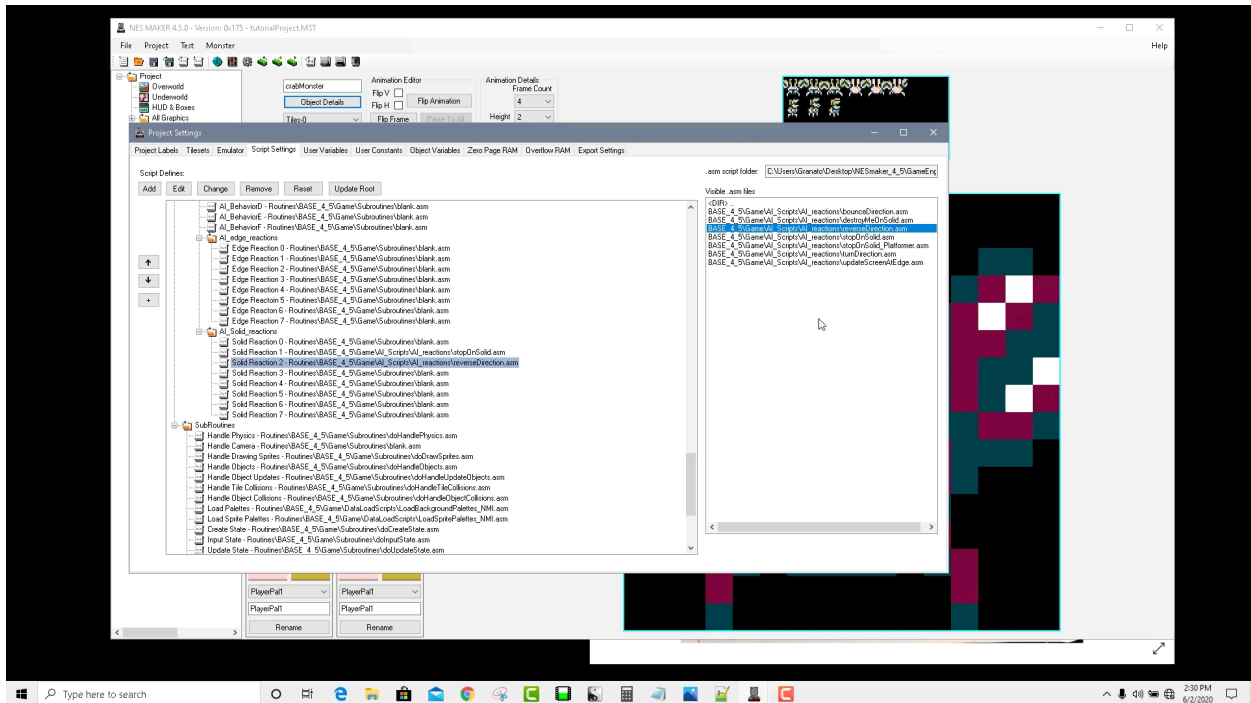


Step 7: Right now, if the monster runs into a solid object, he'll walk right through it. We need him to reverse direction when he hits a solid object. For the player, we set up a solid reaction type for STOP, because we want the player to stop when he hits a solid object. But with a monster, that's not exactly what we want him to do. We want him to keep moving, just reverse his direction. So we need a new AI reaction Type.

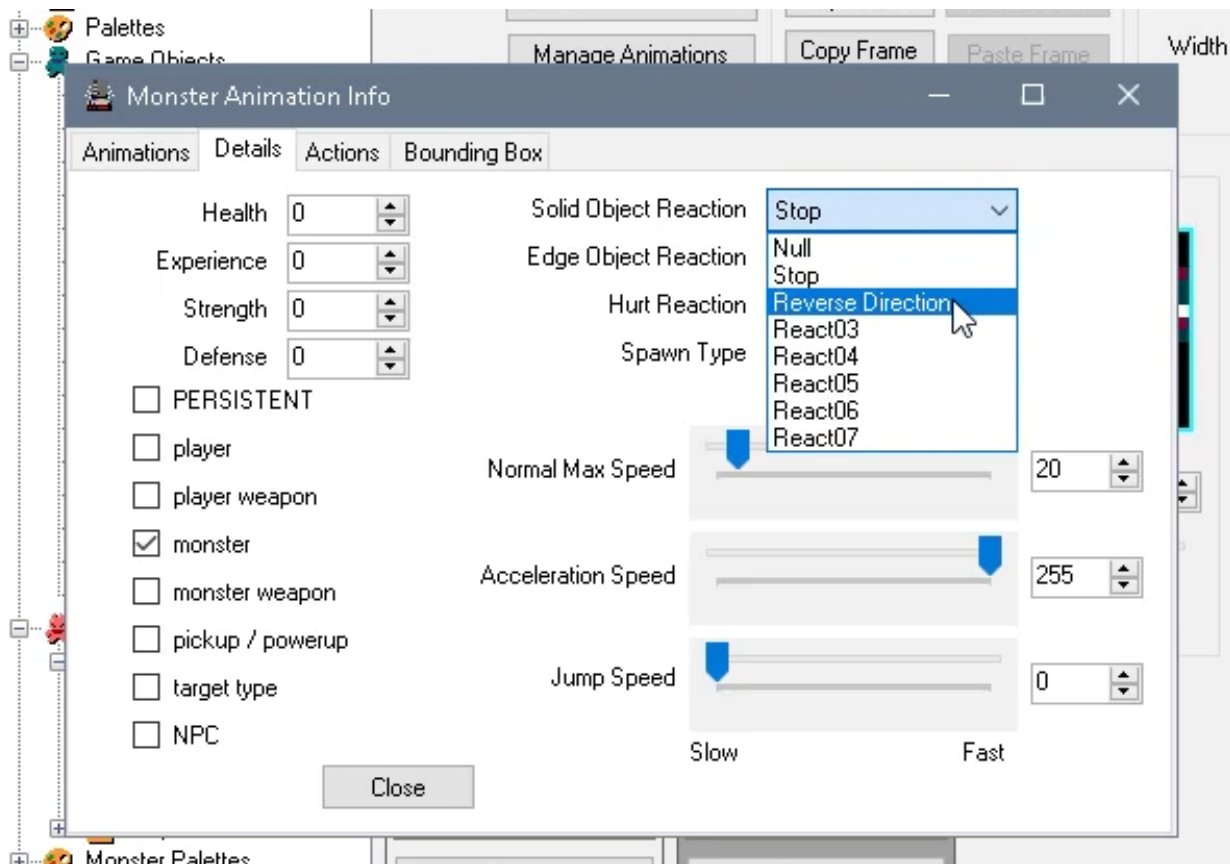
Open Project Settings, and on the labels tab, click on Monster Solid Reactions. Change React2 to Reverse Direction. Again, this is just setting up a label. This is not actually doing anything. We still have to assign a piece of code to this reaction that will cause it to have the intended effect.



Step 8: Click on the Script Settings tab and scroll down to AI Reactions. Click on Solid Reaction 2. In the script picker, navigate to root / Game / AI Scripts / AI Reactions, and double click on reverseDirection. This will associate the reverseDirection script with AI Reaction type 2.



Step 9: Return to your crab object info. Click on Object Details, and navigate to the Details tab. Where it says Solid Object Reaction, choose Reverse Direction from the drop down. Now, when this particular object runs into a solid, it will have the desired effect and turn around.



Test your game, and now you'll see your crabs running around the screen, stopping every second or so, and changing direction when they run into a solid object.

Monster-Player Collision

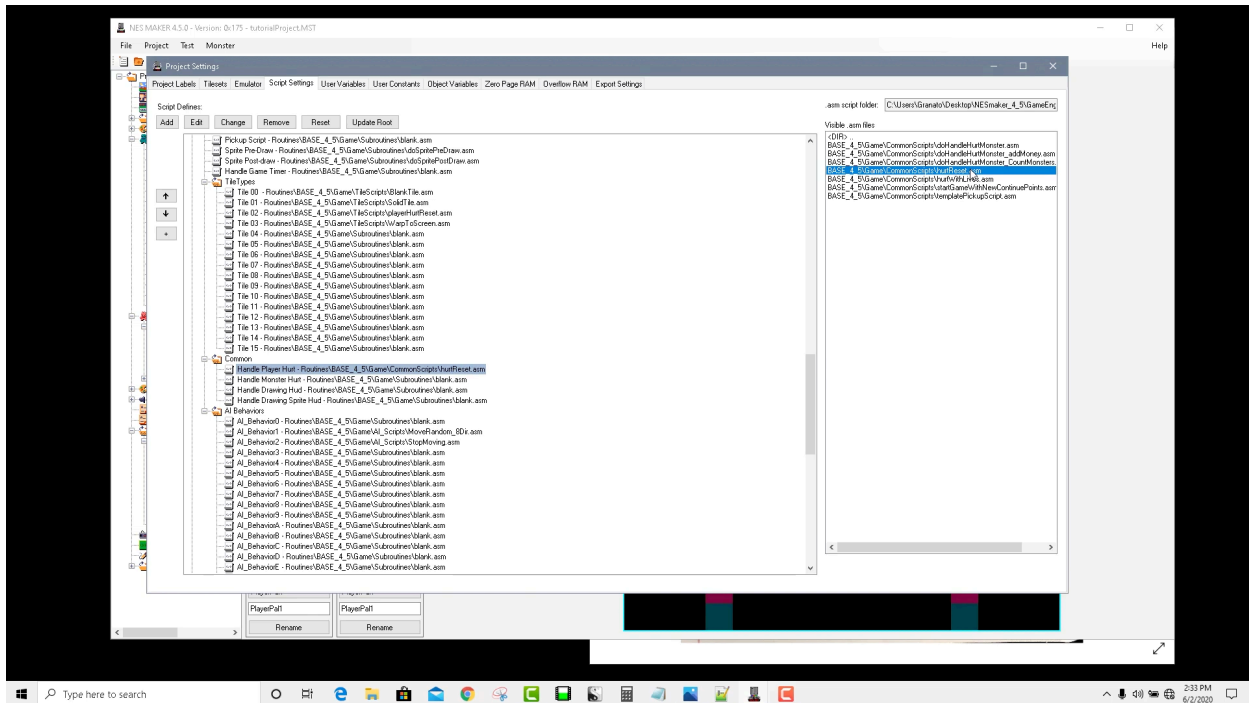
Right now, if your player collides with the crab, they walk right through each other. Obviously, the point of having an antagonist running around the screen is to be a challenge preventing further advancing into the game.

Looking at the game through decades of gaming vernacular, you probably have some ideas as to what could or should happen when the two collide, but what happens when two objects collide varies greatly from game to game. If I said in my design document a high level statement like, "When the player runs into the crab monster, he dies", what does "dies" mean in context? Is it a simple reset of

the game? Does he have a lives system where the collision causes him to lose a life and start the current stage over again? Is there any sort of animation that plays upon death? Sound effect? Does the player have health? If he has health, does he get knocked back? Does he have an invincibility timer to let the player recover? Is he frozen while in a hurt state? Even what seems like an obvious mechanic like this can mean an infinite number of things, and each would have particulars about the method to achieve them. Consider the difference between “dying” in Super Mario Brothers versus “dying” in the Legend of Zelda. In the former, the music stops, a sound effect plays, the hero faces the camera and gets tossed up in the air. Then, after the sound effect is done, it shows a screen that displays the lives left and restarts you at the beginning of that level. In Zelda, when you run into a monster, it takes away a heart. When you lose all hearts, the palettes all change to black except for your player, who shows a spin and explode animation, then you’re taken to a screen with options to save, continue or retry. Each option restarts gameplay in a different way. Before we say the boilerplate “when you run into this monster, the player should die”, we really have to consider what that means for our game.

For our game, for now, we’re just going to set it to restart the game. It’s a simple but clear loss method, and we already have a protocol for it since this is what happens when we run into the spike’s hurt tile type.

Step 1: Go to Project Settings and click on the Script Settings tab. Scroll down to the Common section, and you’ll see that right now, the Player Hurt script has a blank script attached to it. With that script selected, use the script navigator to go to root / Game / CommonScripts, and double click on the script called hurtReset.



Now, because of how this particular module is set up, when an object with the Player bit selected in its object info runs into an object with the Monster bit selected in its object info, it will run this hurtReset script.

Test your game. Now if you run into a monster, you get hurt, and in this case it means a simple reset. We probably want to make something more interesting here, and we will before the end of this tutorial, but now our game has challenges, which makes it feel much more like a game. You can navigate the screen, advance levels using the stair tile, and you get hurt when you run into a monster.

HUD Basics

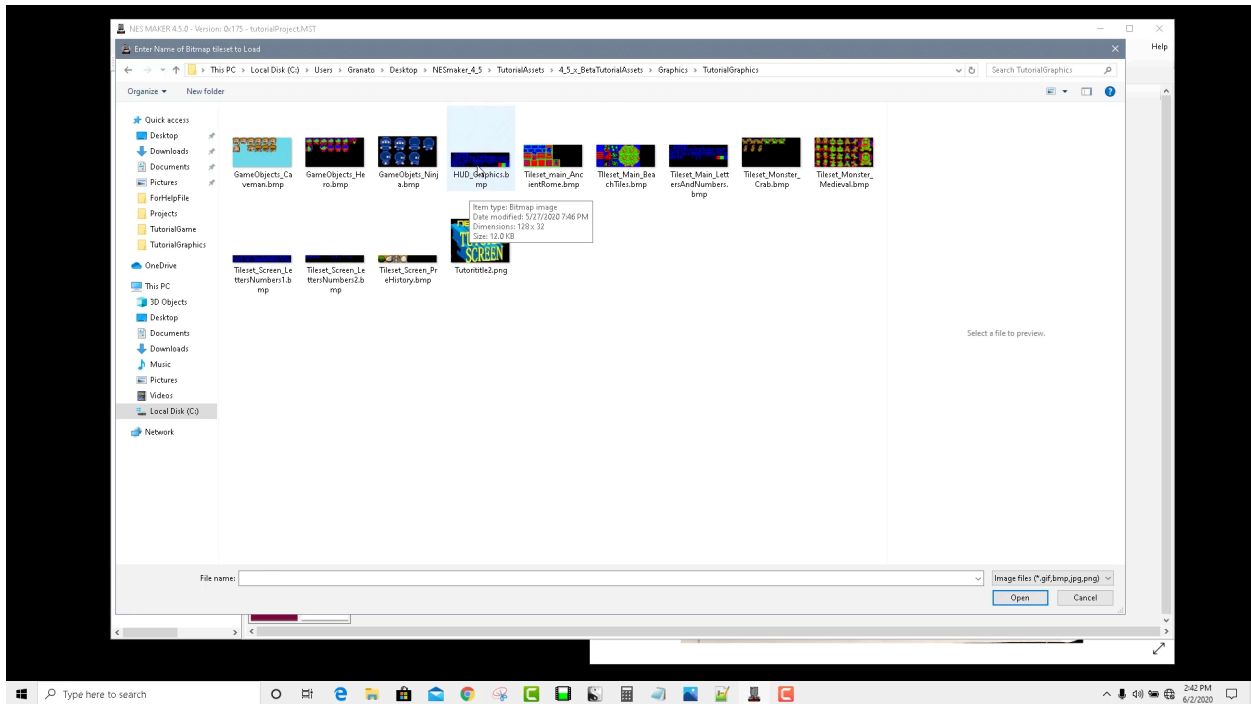
HUD Basics

HUD stands for Heads Up Display. It's an area in a game that gives you, the player, feedback about the state of the game. In NES games, it was often common to find things like lives or magic meters or scores or health or selected weapons in the HUD. Every game has dramatically different needs, and some games don't need a HUD at all. NESmaker's HUD manager works with the preloaded modules to offer a GUI control to HUD design. It's obviously also possible for advanced users to just skip using this method altogether and manager the data in their HUD manually, but since this is an instructional for beginning users, we'll use the built in HUD manager.

If we look back at our design document, our master plan was to have a simple hud that showed us how many lives we have and how much ammo we have left. This was very high level. We never thought out what we would need to do to keep track of that data, how to draw it to the screen, or how and when to invoke updates. We just started a high level as to what experience we want the player to have.

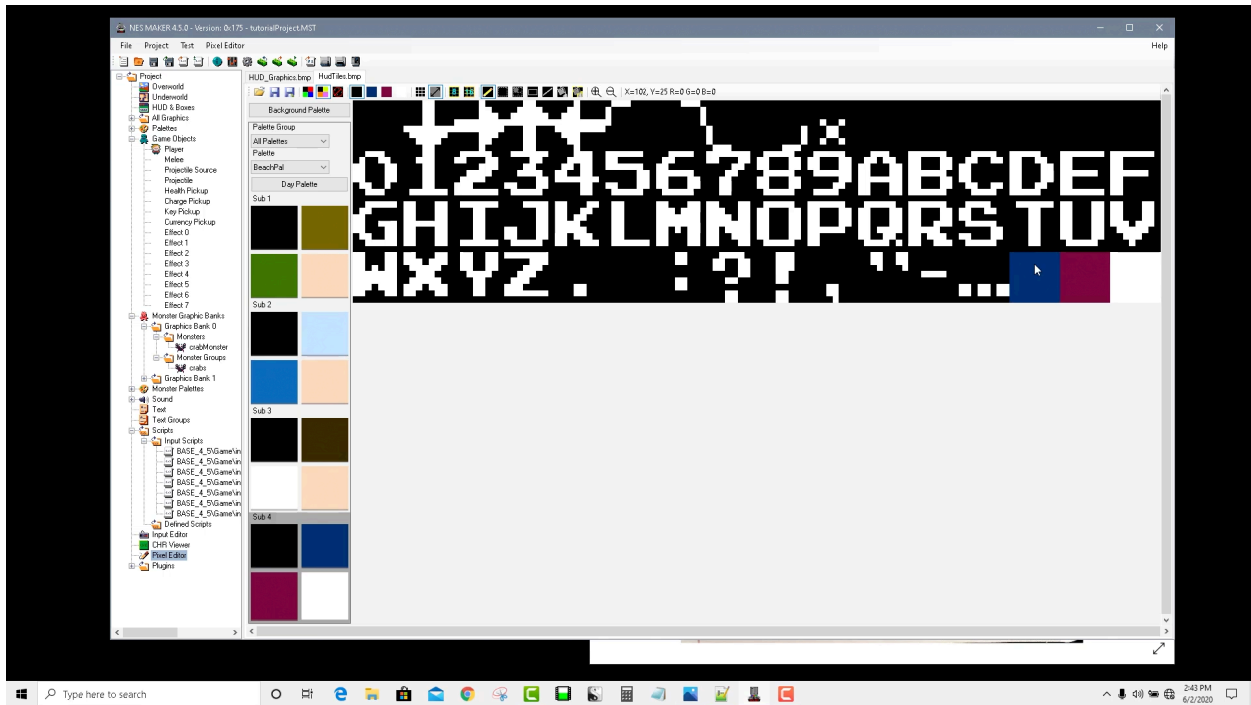
Setting up the HUD

Step 1: The first thing we need are HUD tiles. Right now, we don't have any loaded. Open the pixel editor from the hierarchy. With a new tab, open the root / TutorialAssets / 4_5_x_BetaTutorialAssets / Graphics / TutorialGraphics. There, you'll see HUD_Graphics.bmp.

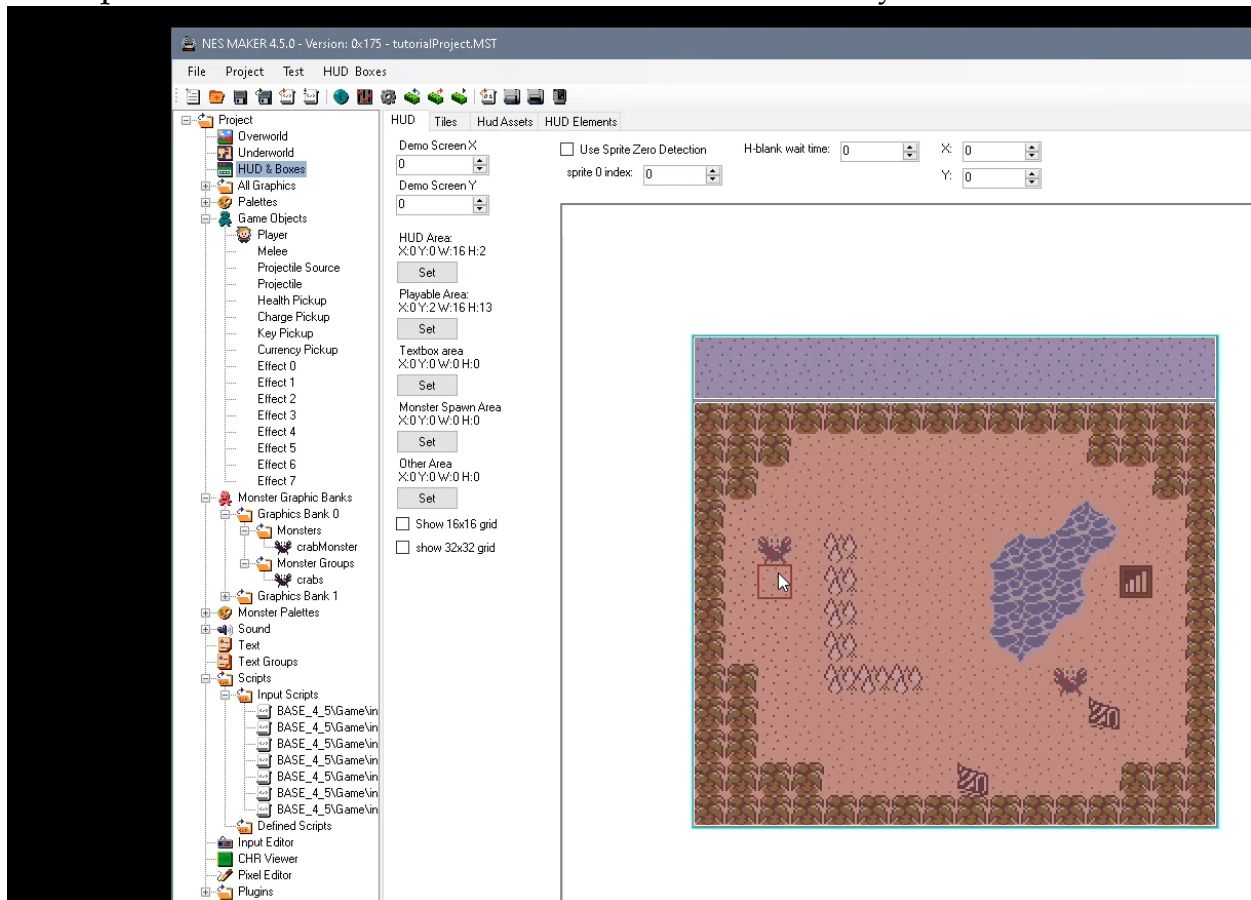


Step 2: This graphic is the size of a HUD bitmap. Just like with other tilesets, you can always use a second tab to open the graphic file where you want these to live and copy / paste from one to the other. But since we know this is the correct size, we can hit Save As and save over the top of the the HudTiles.bmp file.

If you pick the last sub palette and enable palette translation, you can see why we wanted to reserve the last sub palette for the HUD. Now, we can have white for the text, and two explicit other HUD colors should we want them.

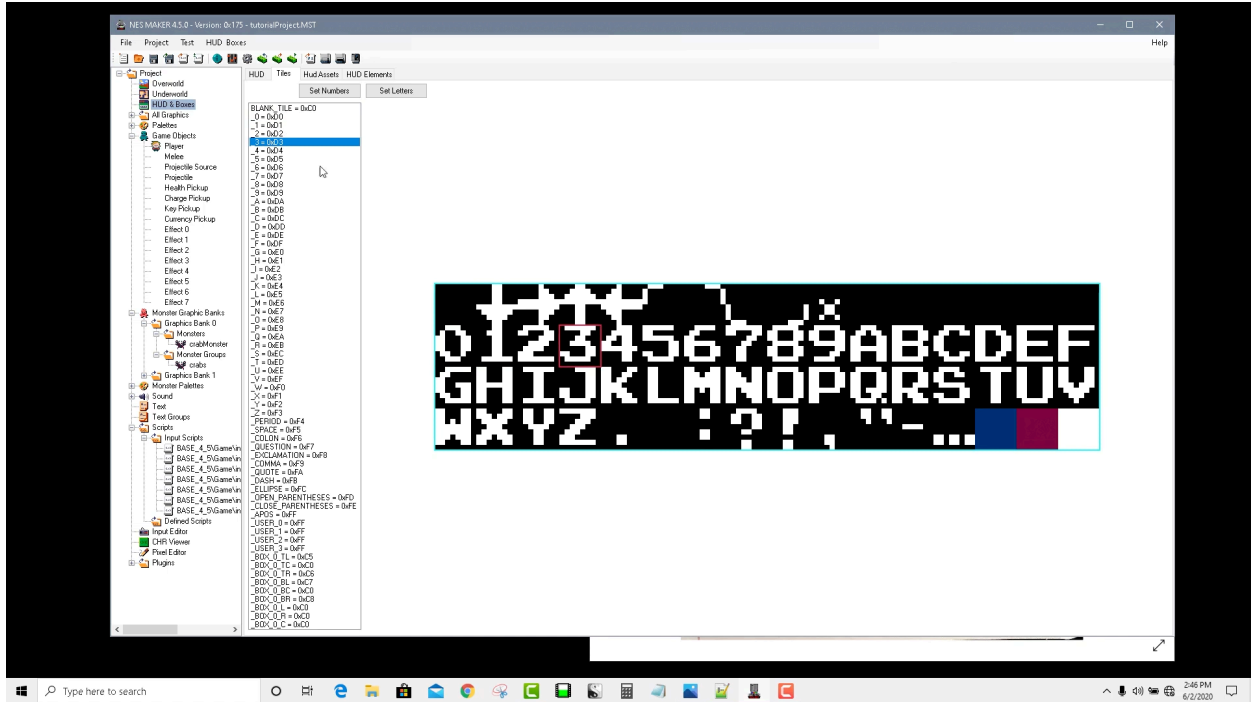


Step 2: Click on the Hud & Boxes node in the hierarchy.

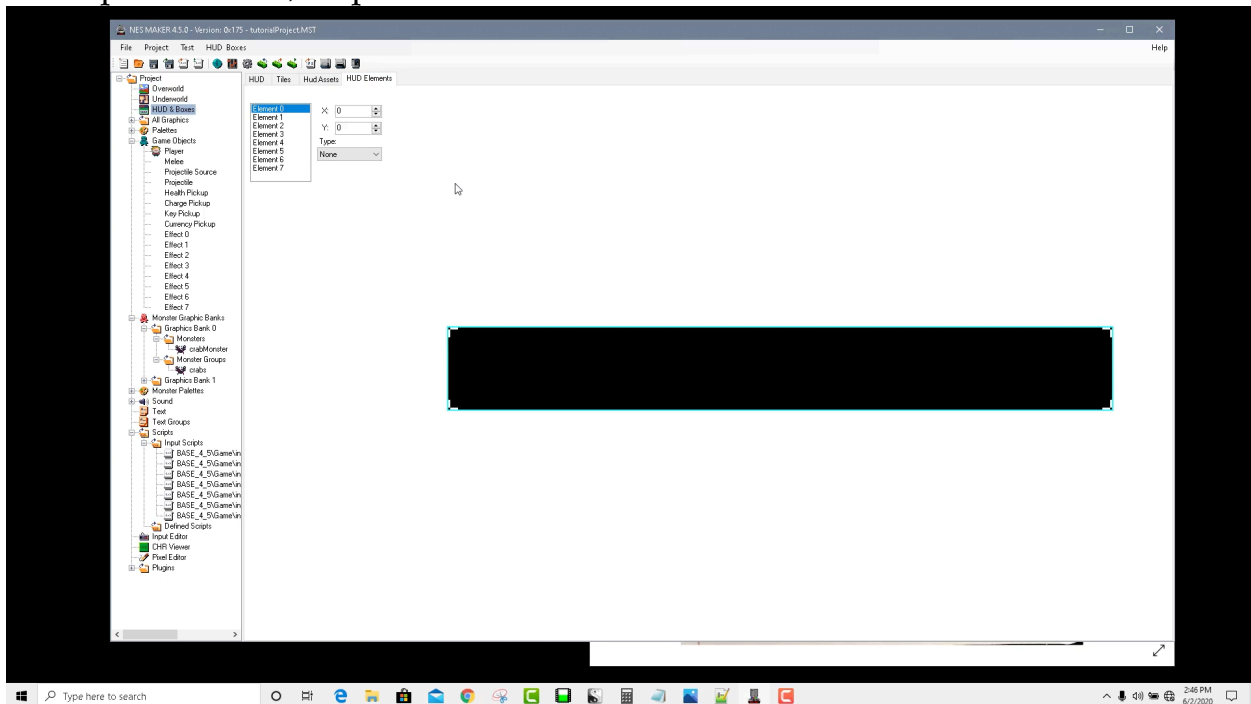


Here we can see that the game is already laid out the way we'd want it. The HUD is already set to the right space, which is why it shows up as an overlay on our screen designer. Our playable area is all the rest of the screen. You could click and drag a rectangle to change the size or shape of the HUD or playable area, too. For this game, this is the appropriate size. If you want to see how the HUD looks on various screens, you can change the Demo Screen X and Y, and that will show you screens based on map coordinates for reference.

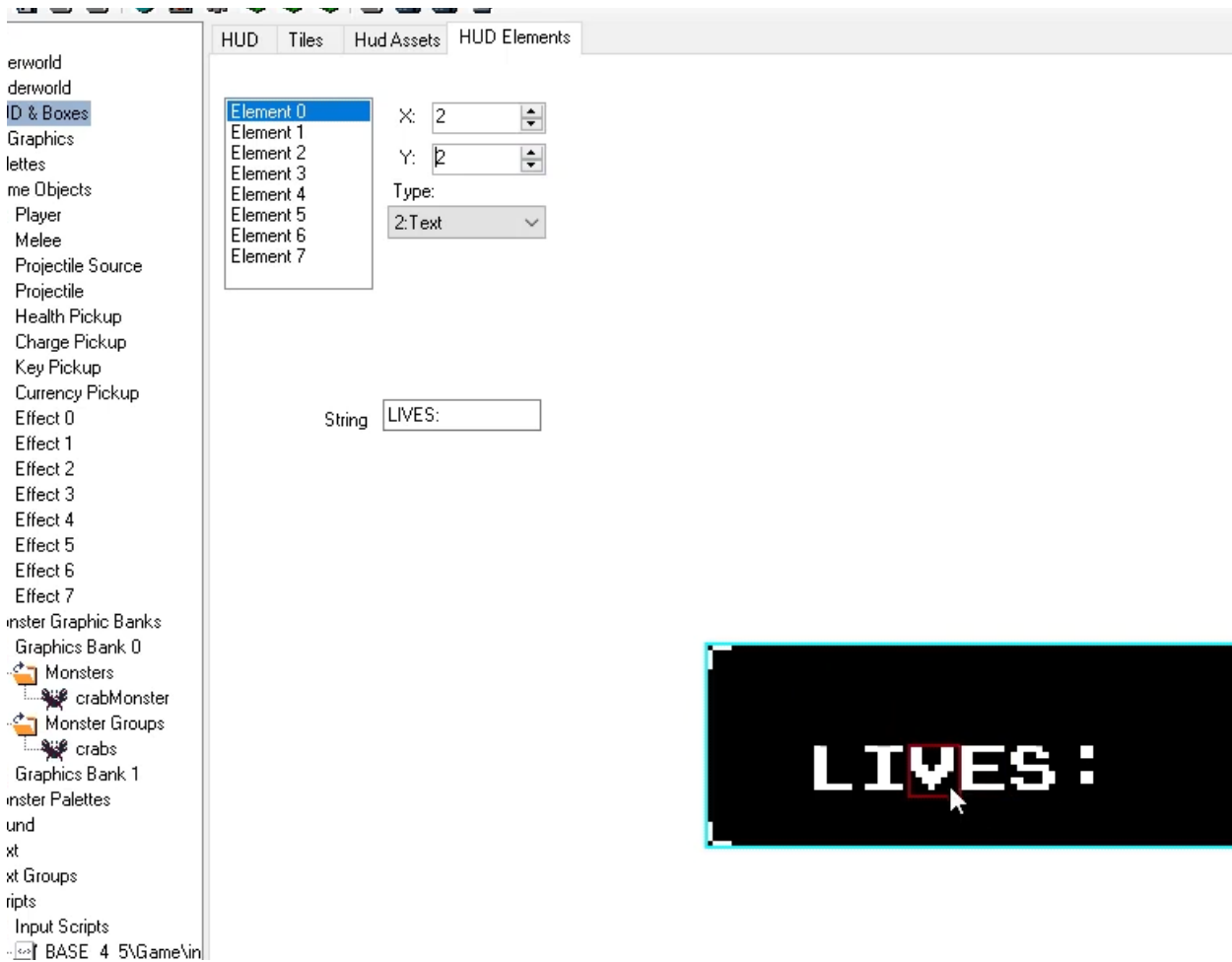
Step 3: In the tiles tab, you can see a list on the left of all of the shorthand that these NESmaker modules' engines understand for text tiles, whether that be in a HUD or an NPC dialog box or a cut scene. These are already defaulted to this tileset that you see, but if you created your own tileset with different needs, you could go through and individual change the tiles to whatever placement you want them. Additionally, if your numbers or letters are sequential, you can click on the first (0 / A respectively) and hit the buttons at the top that say Set Numbers / Set Letters. For the former, it will automatically use the selected +9 tiles to denote numbers. For the latter, it will use the selected +25 for letters. Again, if using this default HUD tileset, or one where the letters and numbers are in the same place, this is unnecessary since they are already set.



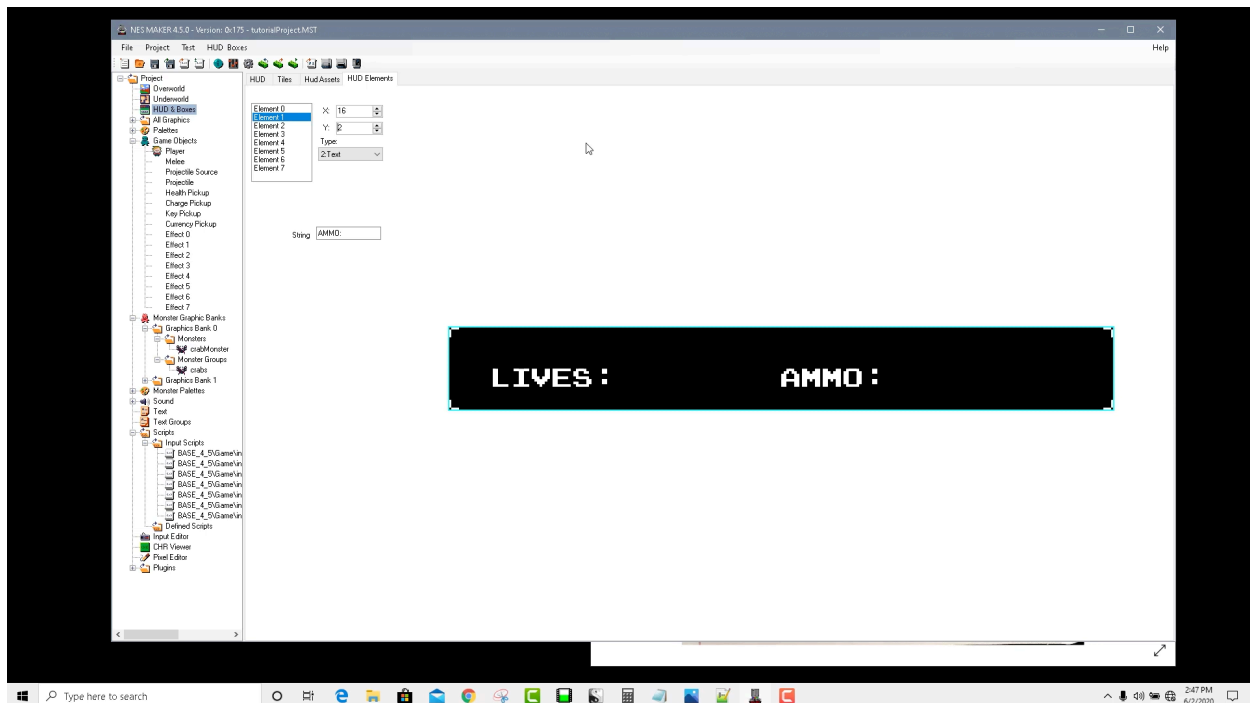
Step 4: For now, skip the HUD Assets tab and click on the HUD Elements tab.



Step 5: For element 0, chose Text from the Type dropdown. Type the word LIVES: in the String field. Use the X and Y number fields to move the word LIVES: to the desired location.



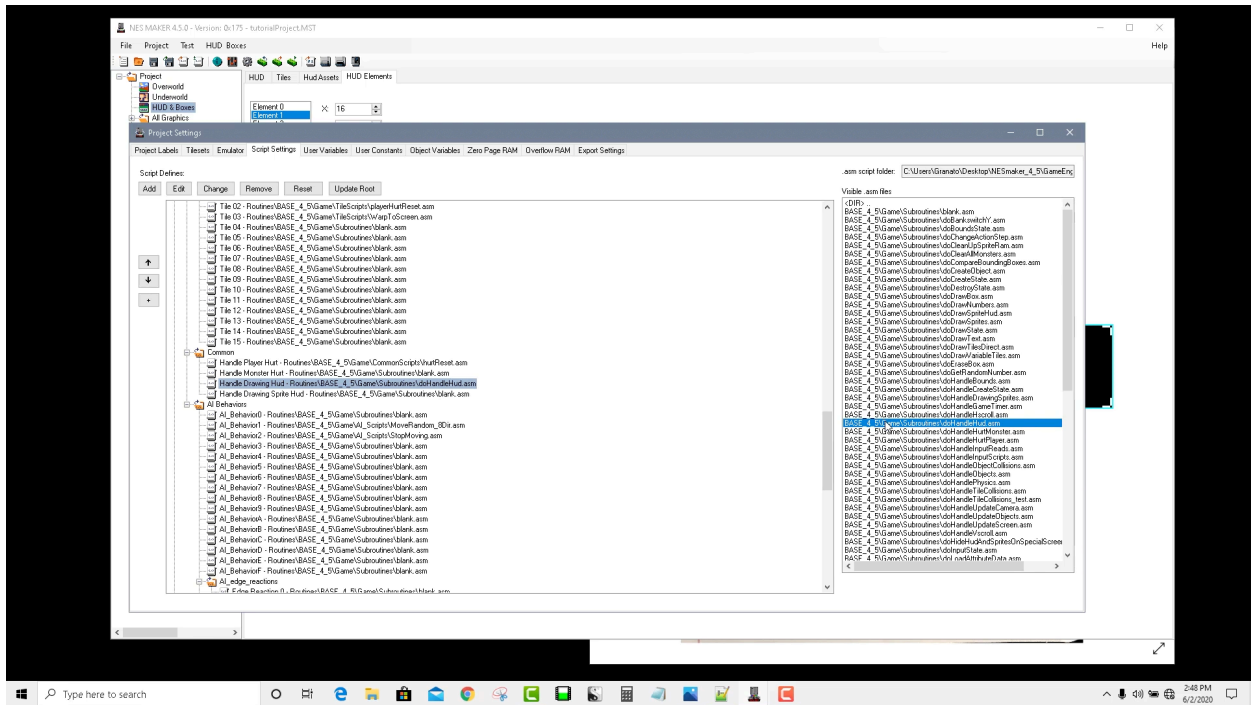
Step 6: For element 1, choose Text from the Type dropdown. Type the word AMMO: in the String field. Use the X and Y number fields to move the word AMMO: to the desired location.



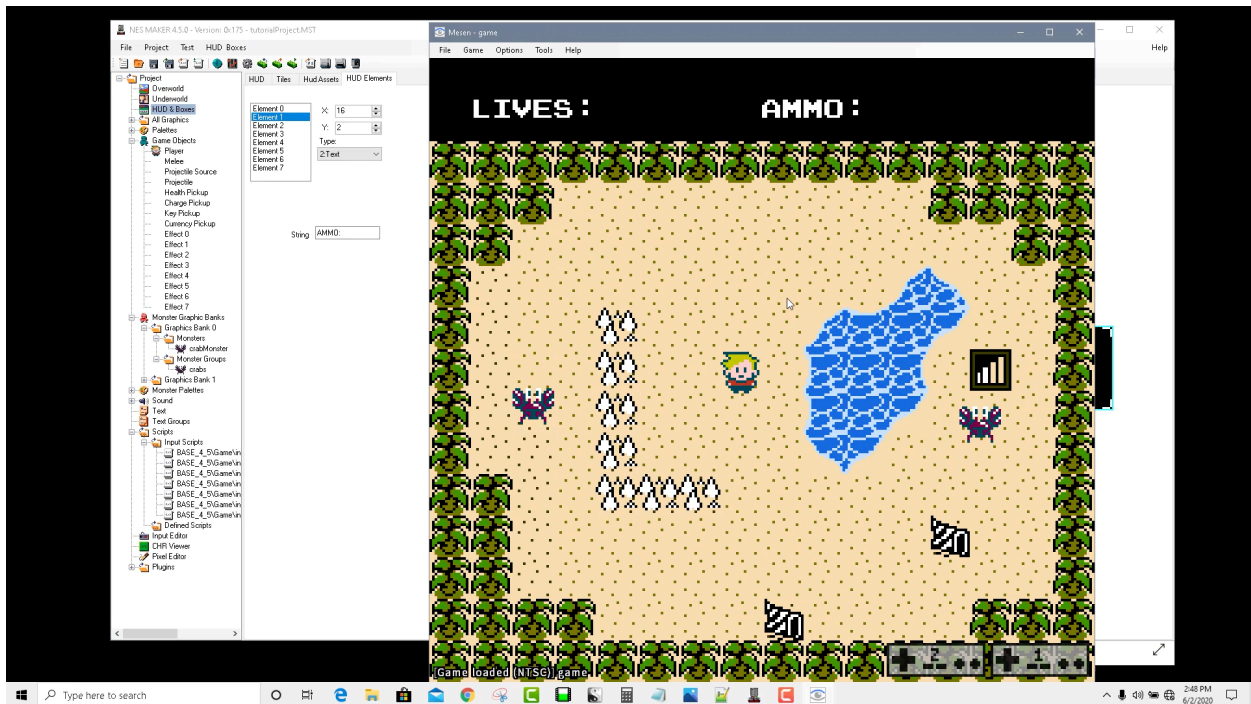
Step 7: Right now, the HUD code that would cause this to draw to the screen is not referenced in our HUD game. When we navigate to the part of the engine that handles the HUD, we'll see it's a blank script. We need to assign a script that will know what to do with the info that this is spitting out. If our game doesn't use this traditional top-bar style HUD, we could leave the HUD script blank. This would give us more room for game logic. Or, perhaps we want to write an optimized HUD handling script that meets the specific needs for our game rather than being needlessly versatile. This is the benefit of breaking the script into these referenced pieces. We can include what we need for our game, leave out what we don't need to save space, or optimize script pieces to fit the very specific needs of our game.

For this game, we will use the normal HUD. Open Project Settings, click on the Script Settings tab, and scroll down to Common. There, you'll see a Handle Drawing Hud script reference that points to a blank script. Click on it.

In the script finder on the right, navigate to Root / Game / Subroutines and double click on doHandleHud.



Test your game and you'll now see that our basic HUD is drawn at the top, with the words LIVES and AMMO in their proper positions.



This is great, but obviously right now these two HUD indicators are relatively

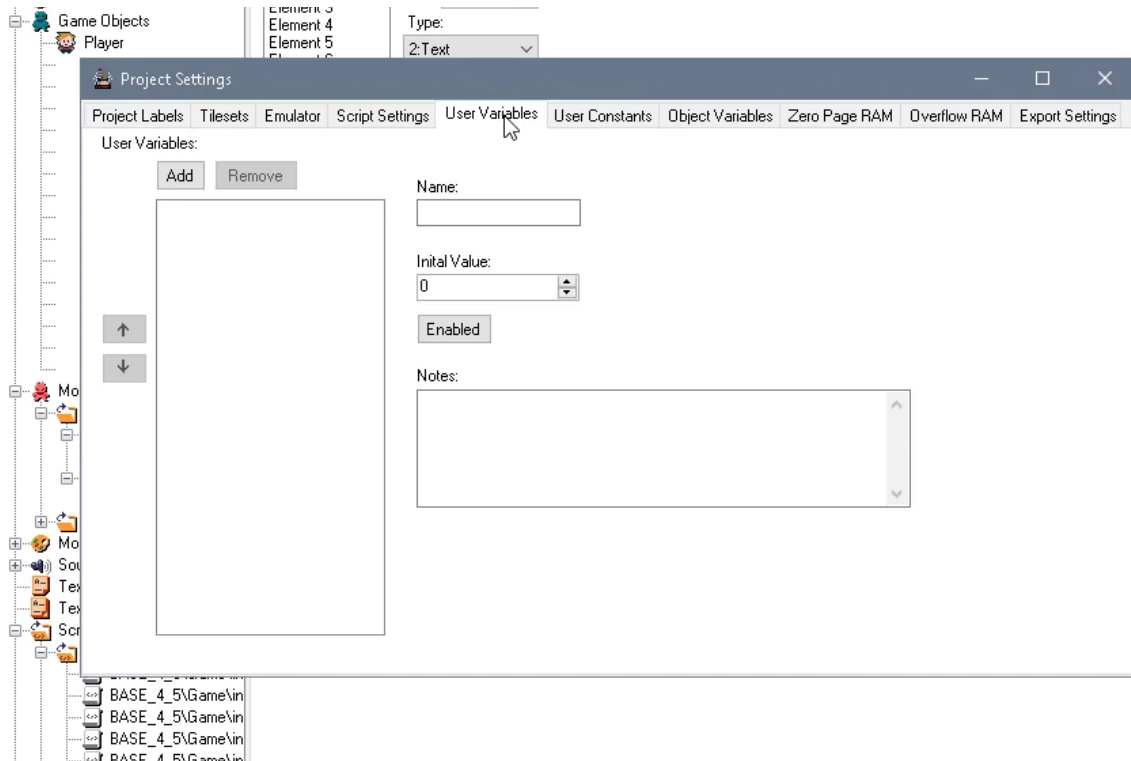
meaningless since they only give us blank information. We haven't given the game any indication of what LIVES or AMMO mean, and we also haven't given the game any indication of how or what to draw. For the game, drawing the word LIVES is no different than drawing a tree or a shell. Our HUD tool just gave it the right tiles to draw in the right places. The next step is to create values to draw .

HUD Variable Basics

Step 8: We need to start working with variables. In game development, variables are essentially anything in a game that doesn't have a static value. In other words, anything that changes over the course of the game. The current game state is variable. The screen we are on is variable. The position of our player is determined by his x and y coordinate variables. What tileset this screen loaded is decided by a variable. All of these are variables that exist as part of the existing module engine already, because all of them are rather game-agnostic. All games of all types need those sorts of things. But now we want to get into our game specific variables, which we will need to declare, give value to, and define how they should work.

For this game, we need to create two variables, one that will show us how many LIVES we have left, and one that will show us how much AMMO we currently have.

Go to Project Settings and click on the User Variables tab.

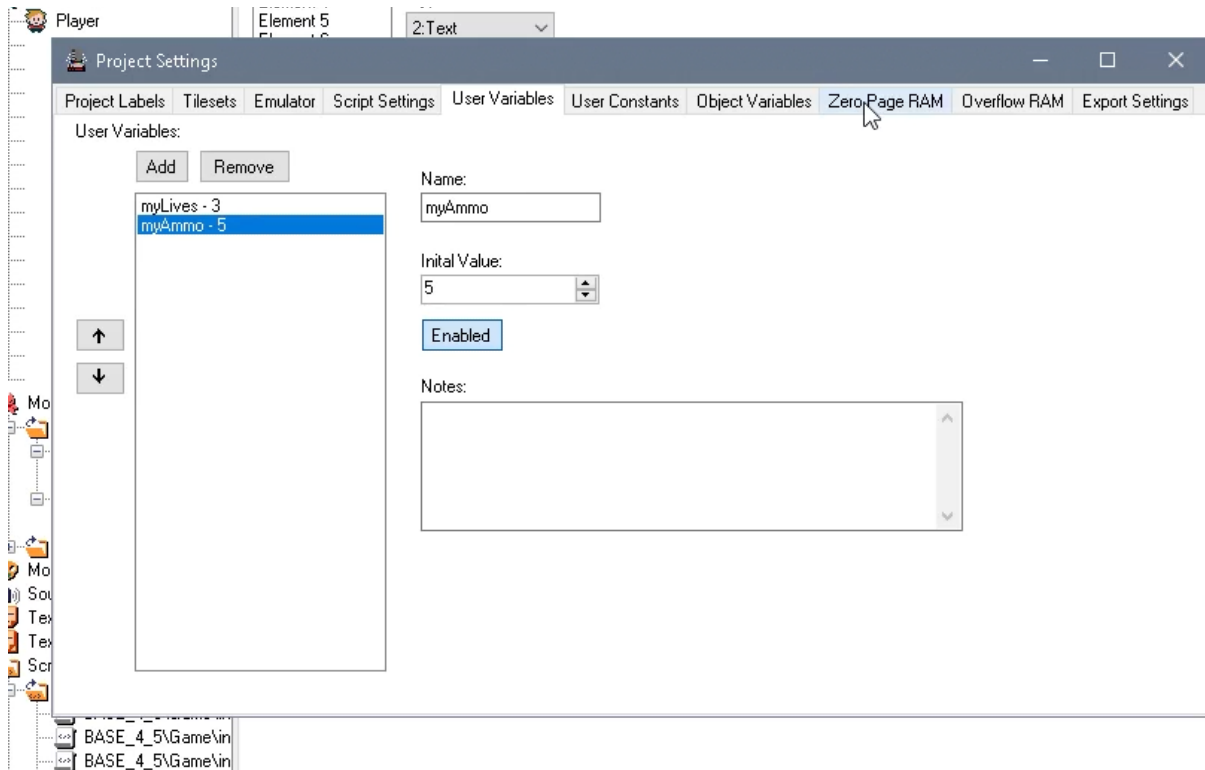


Here you can create a variable and give it an initial value.

Step 9: Click the add button to add a new custom variable and call it myLives. You can really call this anything you want (stick to alphanumeric characters with no spaces, and starting with a letter). However, I suggest for this practice session, you use the variable myLives, spelled exactly that way, because a script we'll be adding to handle your lives system will be looking for that exact variable. It's also important for you to know when you start creating your own variables for your own purposes that if you change their names in the calling scripts, you could change their names to match here.

Set the initial value to 3.

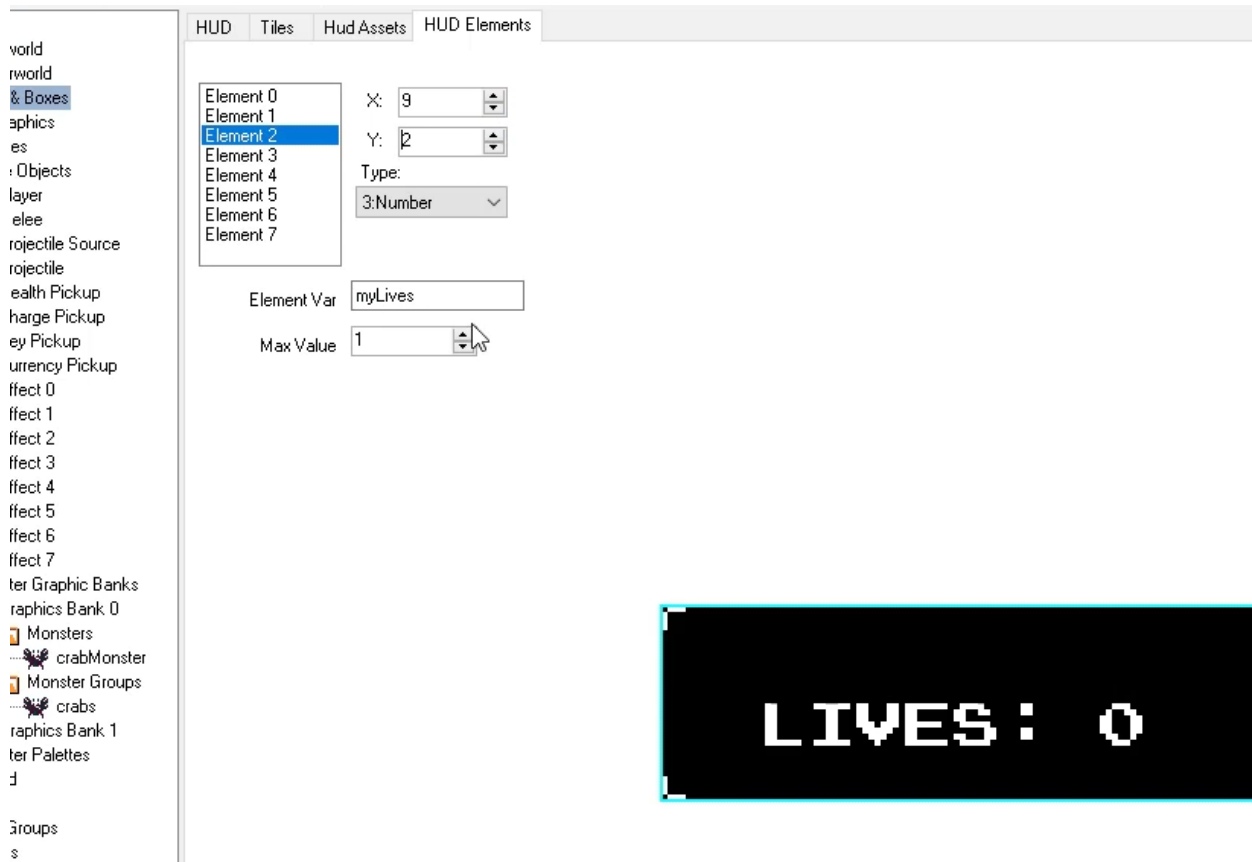
Also add a variable called myAmmo. Set the initial value to 5.



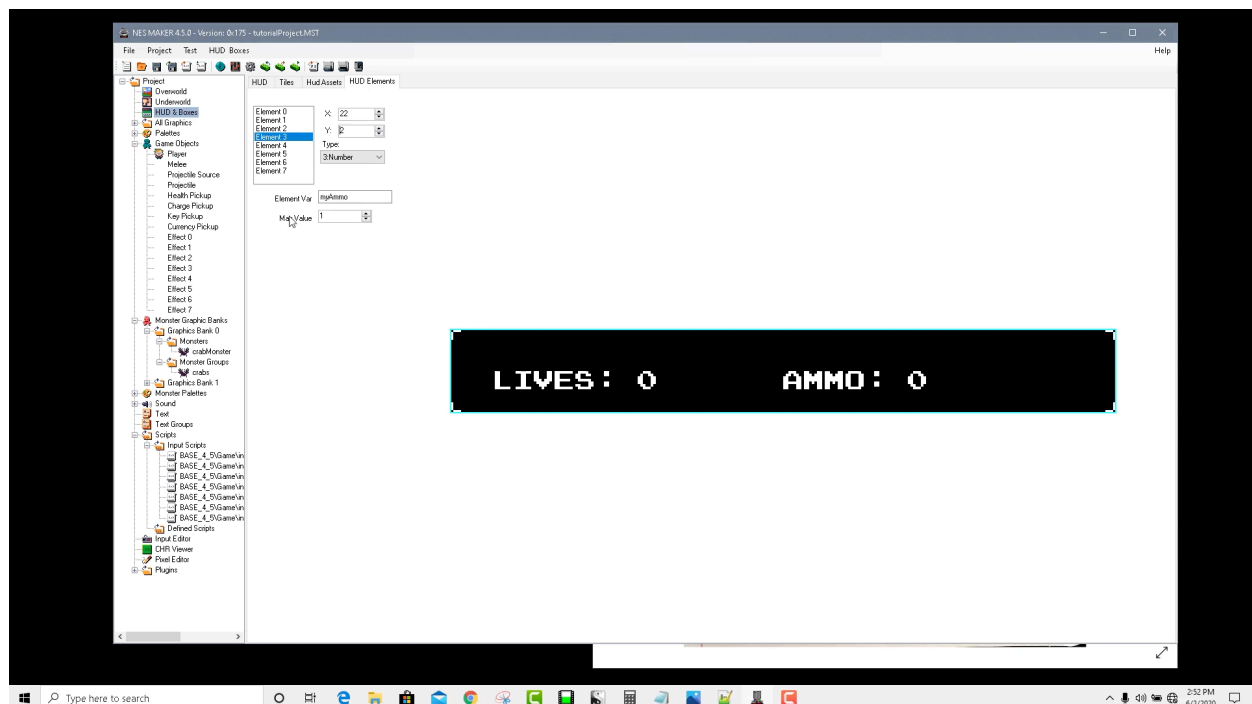
As of right now, these variables are still quite meaningless to the game engine. They are not yet tied to our HUD. They are not yet tied to any function scripts. We won't see the values in our game, and they won't change under any game conditions. But in the background, we now have placeholders for myLives and myAmmo, which are set to 3 and 5 respectively at the start of the game.

Step 10: Open the HUD & Boxes node and click on the HUD Elements Tab. Pick a new element that you haven't used yet (element 2). Set the Element Type to Number from the dropdown list. Change the element Var to the variable we just set up, myLives. Where it says Max Value, for number value, it's asking to how many places should this number extend. We just want a single digit, so we will make this one.

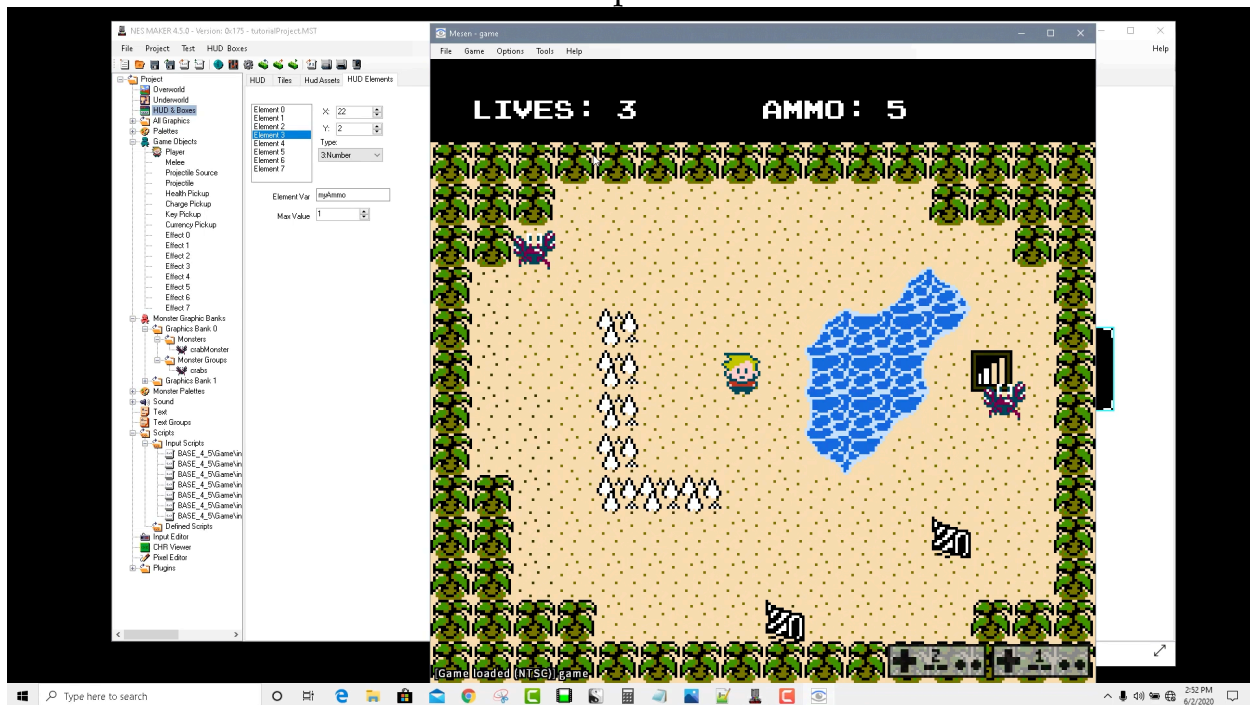
Then, you can use the X and Y fields to move the value of the variable into place.



For element 3, do the same, except use myAmmo for the Element Var, and move it next to the word Ammo.



On the screen, Lives and Ammo will not say zero and zero when you play your game. It will show the value of the variable. Of course, we set the initial values as 3 and 5. So what we would expect is to see 3 lives and 5 ammo.



This is what you should see. Of course, they still functionally mean nothing, as we haven't told the game how to utilize those variables. But now we have a legitimate HUD that is showing us identifier text and the value of variables that we can change. In fact, if you were to go into your Project Settings and change the initial values of these to a number between 0-9, you'd see the value change on screen upon re-running the game.

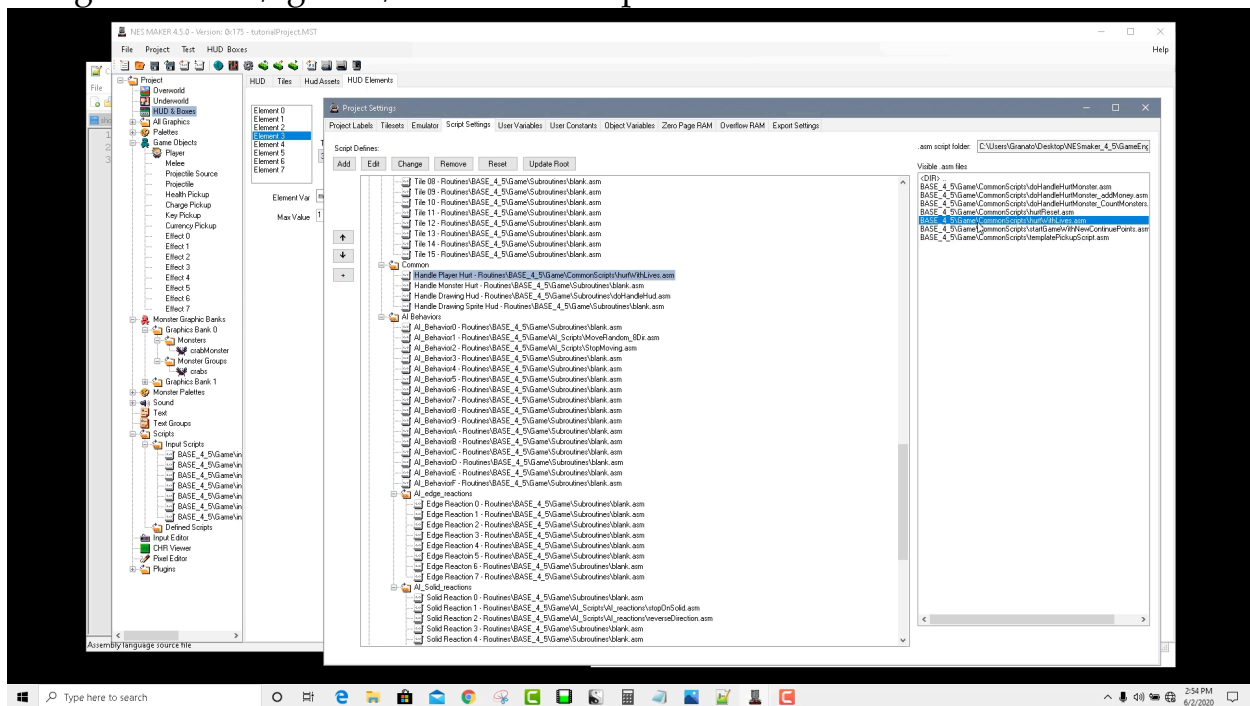
Adding Function to our HUD Variables

With a basic lives system, we would expect to lose a life every time we “die”, which occurs whenever we run into a monster. When we lose all three lives, we would expect the game to completely start over. Right now you'll notice that every time you run into a monster, the game resets. There is no script invoking the myLives variable. Let's make the lives system work the way that it should.

Step 1:

Open Project Settings and click on the Script Settings tab. Scroll down to Common, and you'll see a script define called Handle Player Hurt that we already set. In a previous step, we set this to a script that restarts the game when the player triggers a hurt state. A hurt state is triggered when a player object collides with a monster object. What we're about to do helps demonstrated the further benefits of designing the NESmaker engine this way. We are now going to take the current player hurt script and replace it with a different one. Both work, and both are perfectly sensible scripts to use for this. What exists now may be perfect for one type of game, while the update we're about to give it might be perfect for a different type of game. Similarly, we could download other users' scripts that are shared with the community, or we could even edit the script ourselves, making iterations and A/Bing them against each other to see what works or feels best.

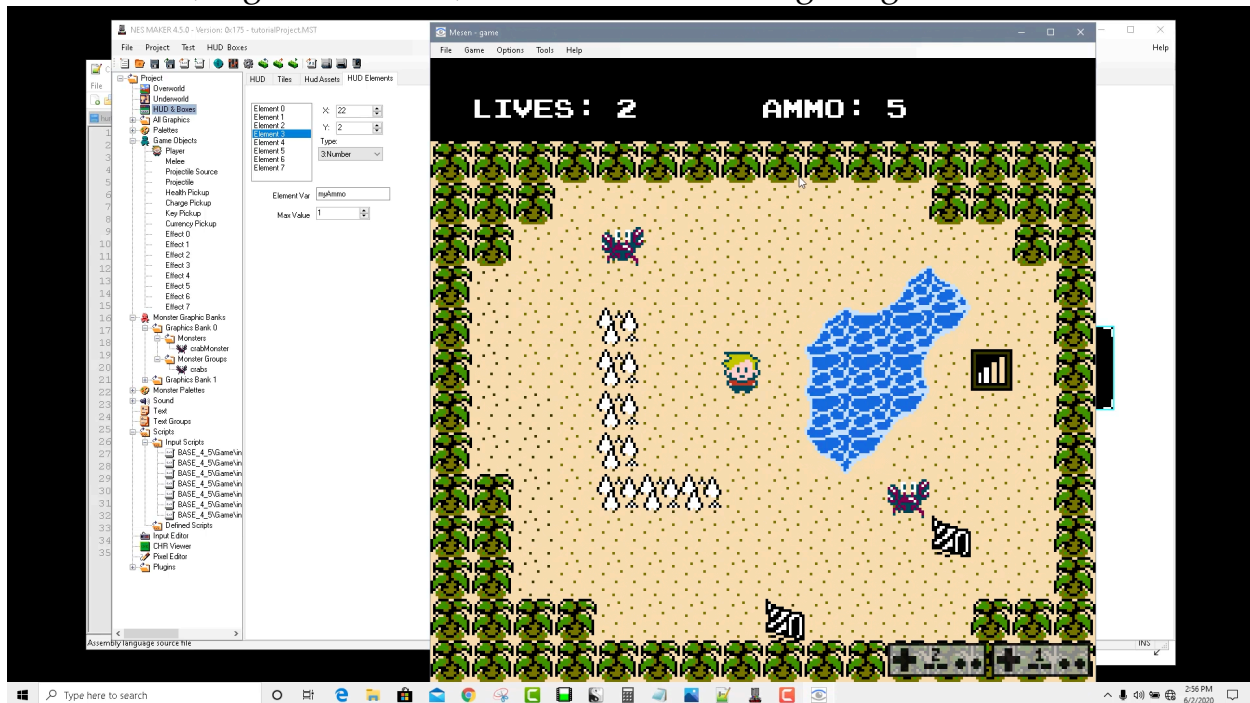
Click on the Handle Player Hurt script define, then use the script finder to navigate to root / game / CommonScripts. Double click on hurtWithLives.



This script works with the myLives variable. It subtracts one from the

myLives variable. If the result is not zero, it resets the current screen. If the result is zero, it resets the game.

Test your game. Try running into your monster. You should see that the LIVES indicator decreases by one each time you get hit until it reaches zero. When it reaches zero, it goes back to 3, because it is restarting the game.



This will be even more clear when our game begins with a start screen instead of a game screen. Then, when the game resets we'll see that start screen instead of the game screen reloading.

Creating a Start Screen

Creating a Start Screen

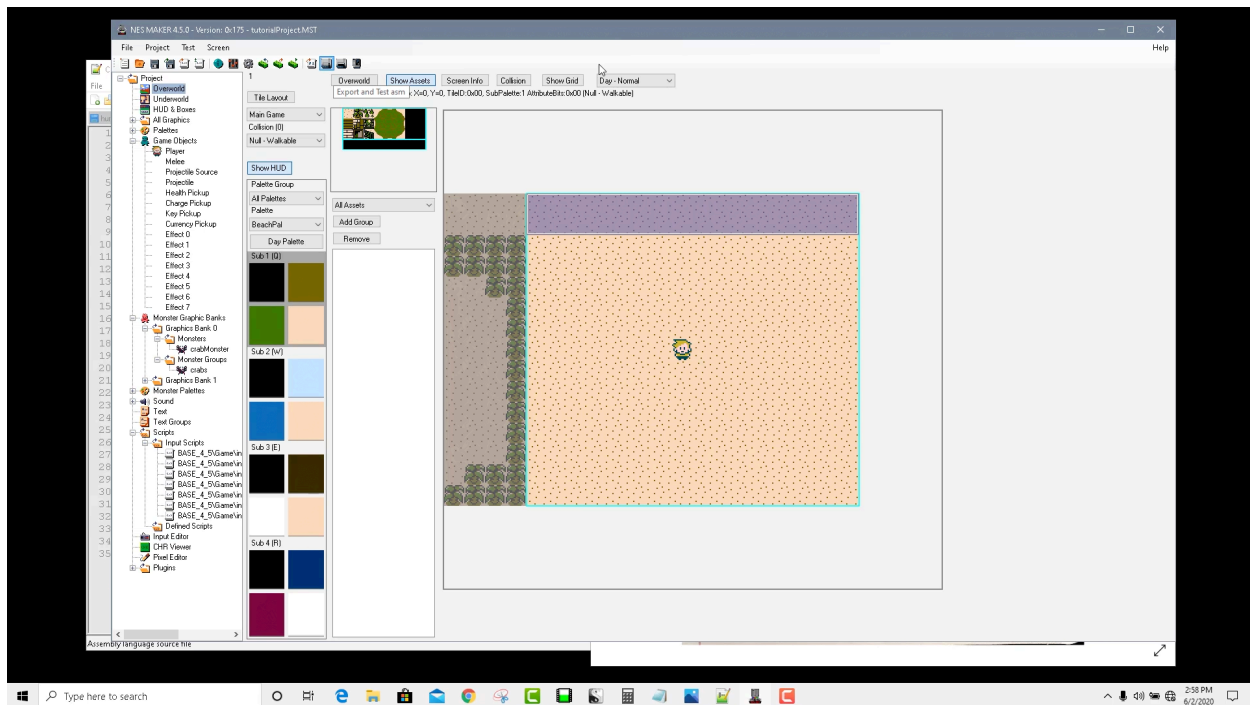
If we look back at the design document we created at the beginning of this project, we can see that we have a problem with our game flow. Right now, the game starts and launches right into the first screen. But that's not right. It's supposed to start with a Start Screen. Then, when we press the start key, it's supposed to go to the Main Game.

In creating a Start Screen, we're going to revisit our game states a bit, and we're going to learn about more complex things we can do with the map editor.

Before we even focus on the graphics, we can just have the start screen be a black screen to get the game flow working properly, and then worry about what that start screen should look like later.

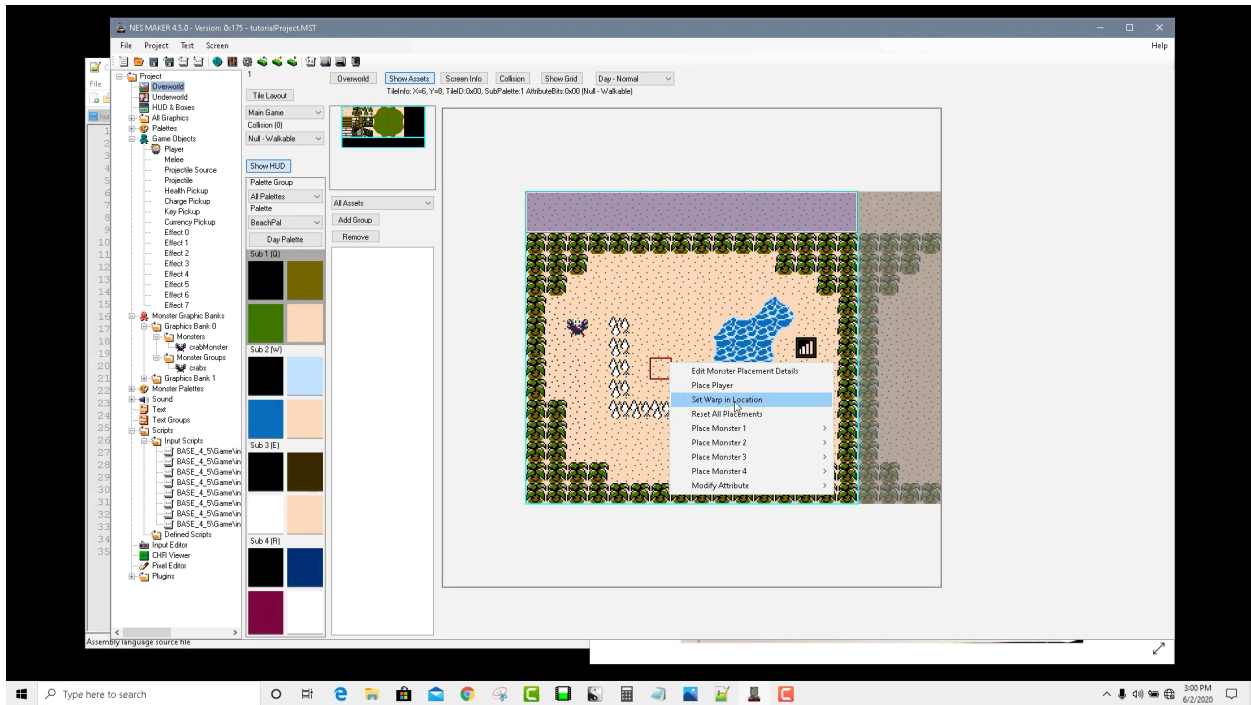
Setting up Start Screen Input

Step 1: Click on overworld from the Hierarchy. For now, just pick an arbitrary screen - any unused screen on the map, and double click on it. In the screen painter, right click anywhere and select Place Player. Now, the game will start on this screen instead of your first gameplay screen.



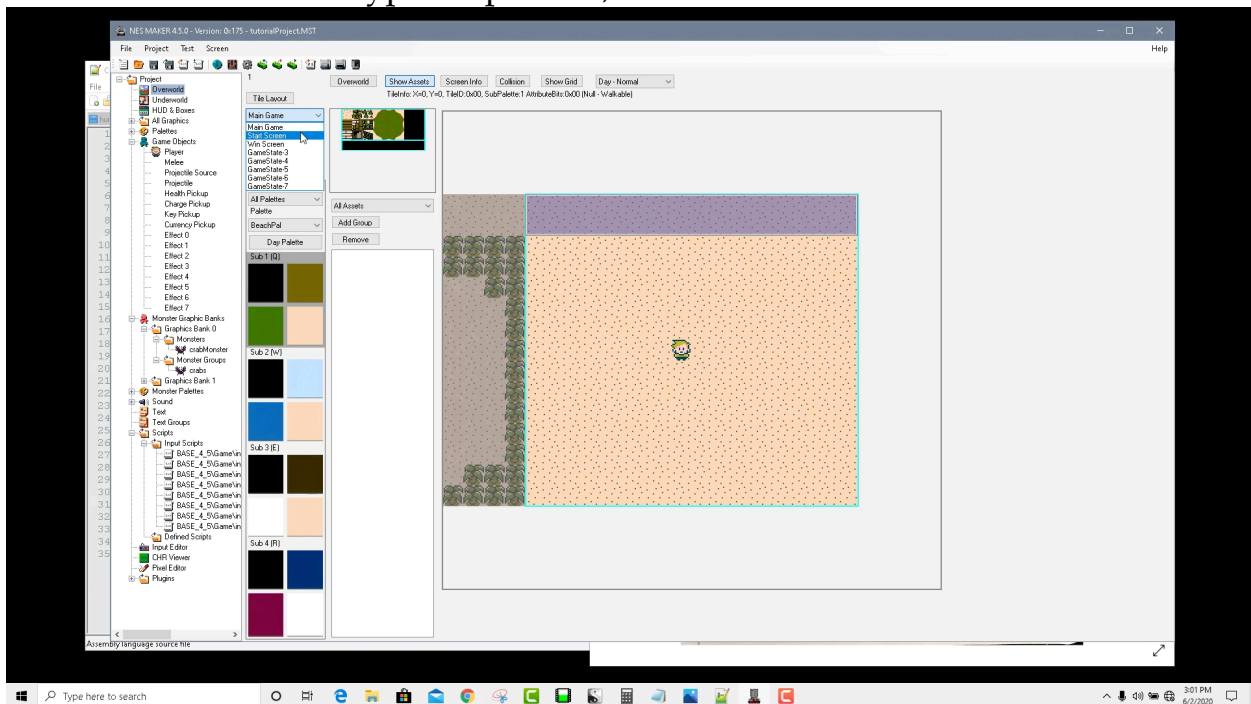
Step 2: In the hierarchy, open the Scripts node and click on Input Scripts. Then navigate in the script finder to root / game / inputScripts, and double click on startGameWithNewContinuePoints. That will add this input function to our game. We still haven't told the game when to use this script, but now we will have access to it in our input editor.

Step 3: This new script will use a warp type screen transition to get from the start screen to the new screen. So return to your overworld map, go to the first gameplay screen, right click where you want your player to appear on this screen, and choose Set Warp In Location. We still want the player's initial placement to be on that start screen, but pressing start will warp him to this place just like a warp tile would.

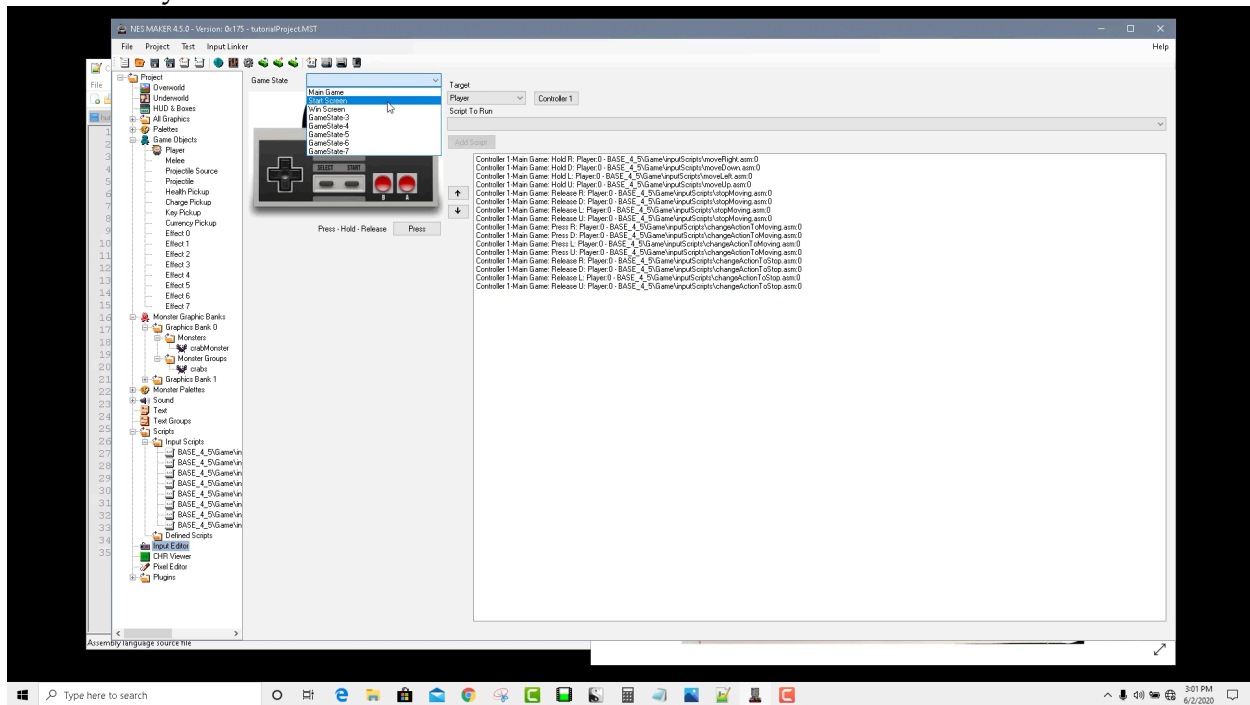


Step 4: We only want to be able to set the start button on the Start Screen screen type. Right now, this screen has defaulted to Main Game screen type, which we established as the zero screen type in earlier steps.

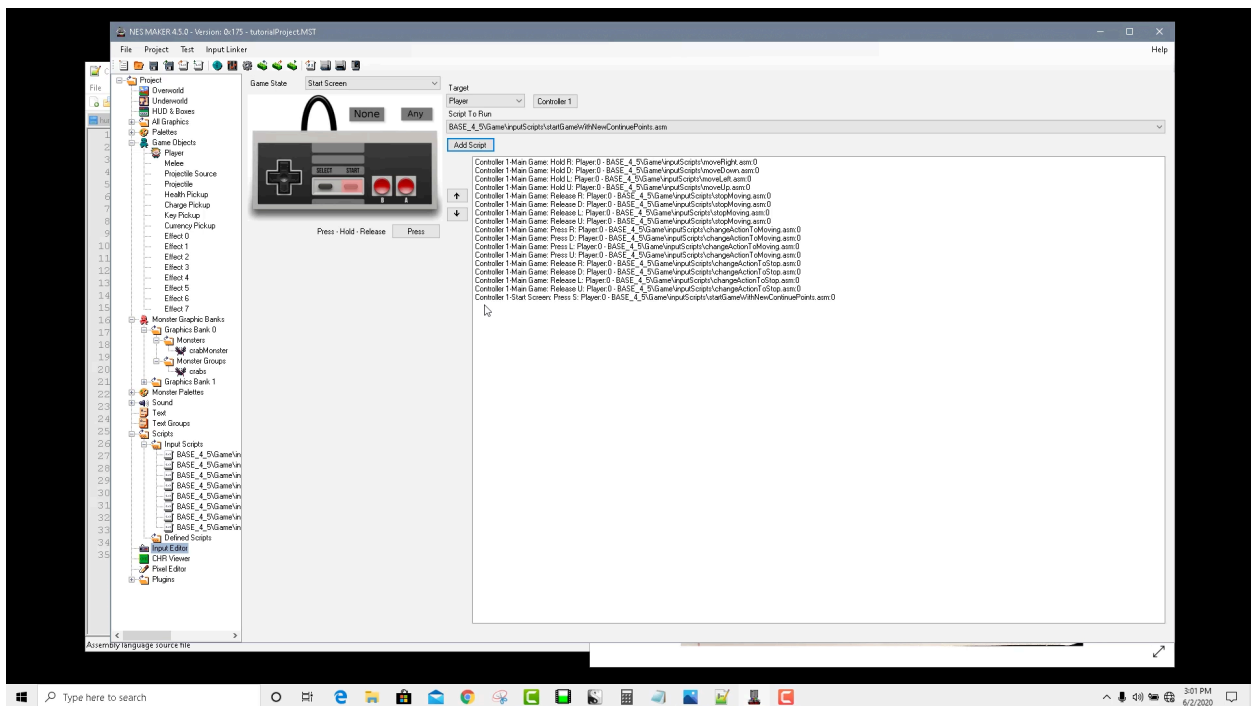
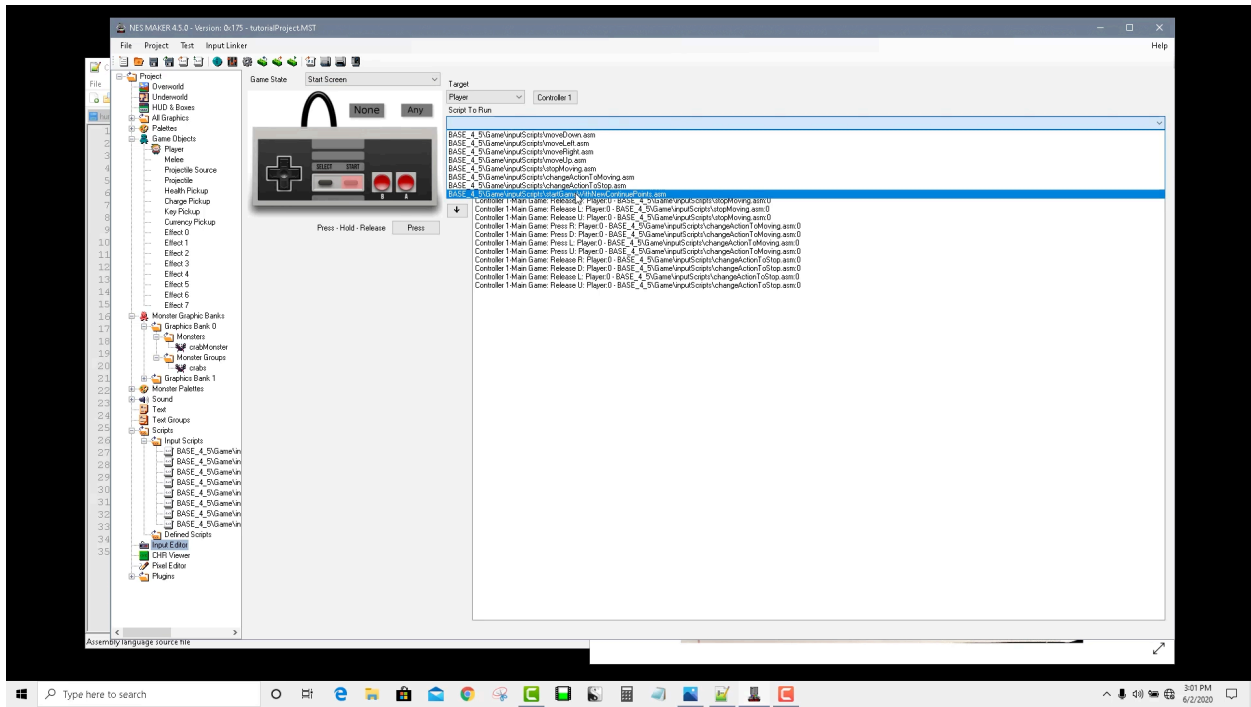
So from the Screen Type drop down, select Start Screen.



Step 5: Click on the Input Editor node in the hierarchy. Change Game State to Start Screen. Now, whatever input command we add will apply to the start screen only.



Step 6: Click on the start button. Make sure the button under the controller says "Press". Choose the script we just added, startGameWithNewContinuePoints. Then press Add Script to add it to your script list.



Since we thought out our game states in the beginning, this is very easy to set up, and to understand. If you read the line for the new input, it says that with controller 1, on the start screen, when you press start, it will run that new start game script. Notice, that it will only do this for screens tagged with the start

screen game state. Similarly, the player movement scripts above it will only work for the Main game game state, so when on a start screen, will have no control over the player.

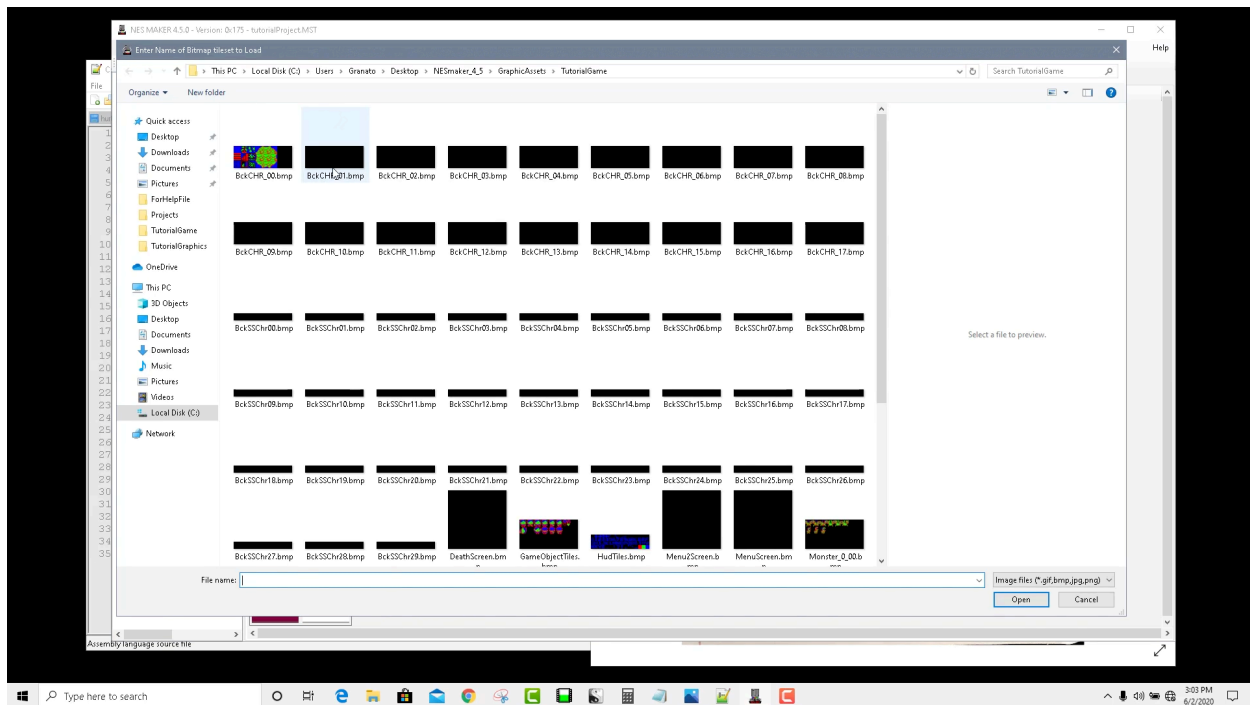
Test your game. When you press start, it should take you to the first screen. When on that first screen, the start button should do nothing.

Just some extended ideas. You could set up a map screen that had certain input controls that differed from the main game or start screen. You could set up an inventory screen that had certain input controls. You could set up cutscenes that had certain controls. Even with this limited knowledge, you can start to conceptualize really creative game flows.

Unfortunately, our start screen is really not that good of a start screen. It's completely functional, and I'm sure we could create a tileset with graphics that would help this look and feel more like a start screen. However, we're going to use this opportunity to check out some of the other map features.

Making the Start Screen Tilesets:

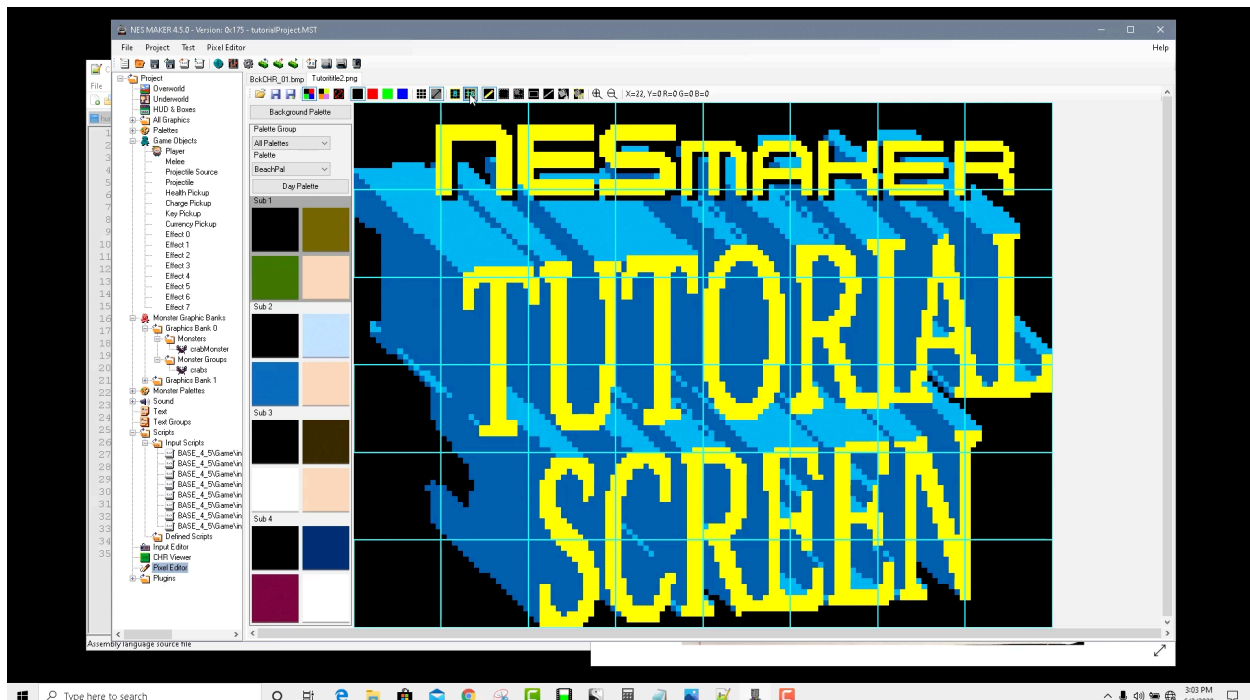
Step 7: Go to the pixel editor. Open a fresh tab. Feel free to close any other open tabs. Open a blank background tileset. I'll use BckChr_01.



Step 8: Make a new tab. Open root / TutorialAssets/ 4_5_x_BetaTutorialAssets / Graphics / tutorial graphics. Double click on NESmaker Tutorial Screen.

The first thing you may notice is that the Tutorial Screen graphics won't fit in a single tileset. Also, of course, they are not in RGB-A format.

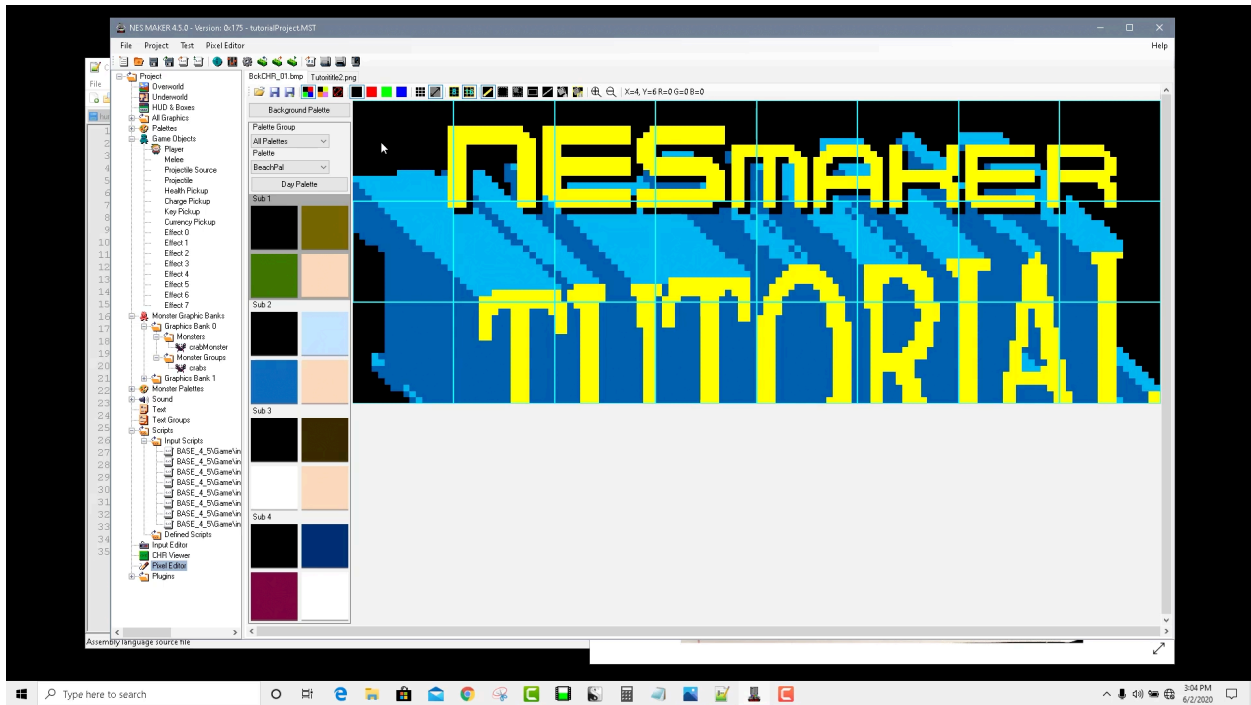
So with the NESmaker Tutorial Screen graphic tab open, turn on the 16px grid.



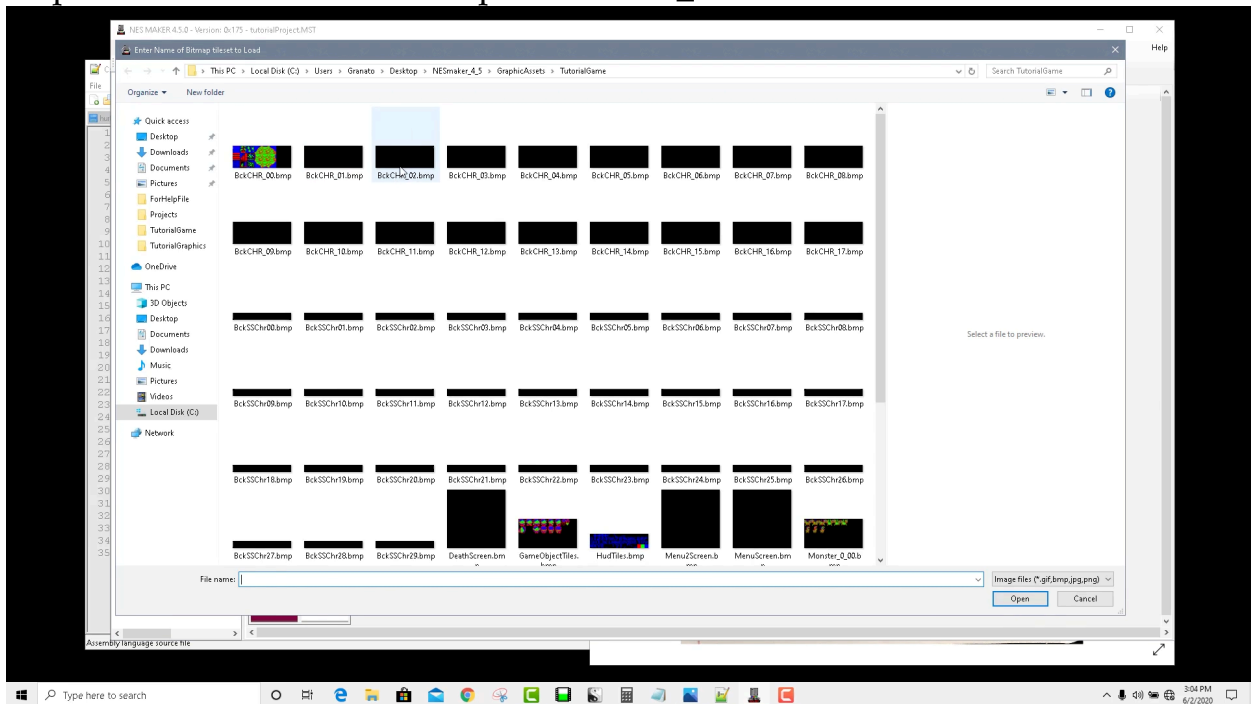
You can see that it is six 16x16px tiles tall. If you click on the tab for your BckChr_01 and turn on its 16px grid, you can see it's only three 16x16px tiles tall. We'll need to make use of two main tilesets to draw this graphic.

Step 9: With the NESmaker Tutorial Screen tab open, use the Tile Selection tool to select the top three rows of tiles. Press control-C to copy.

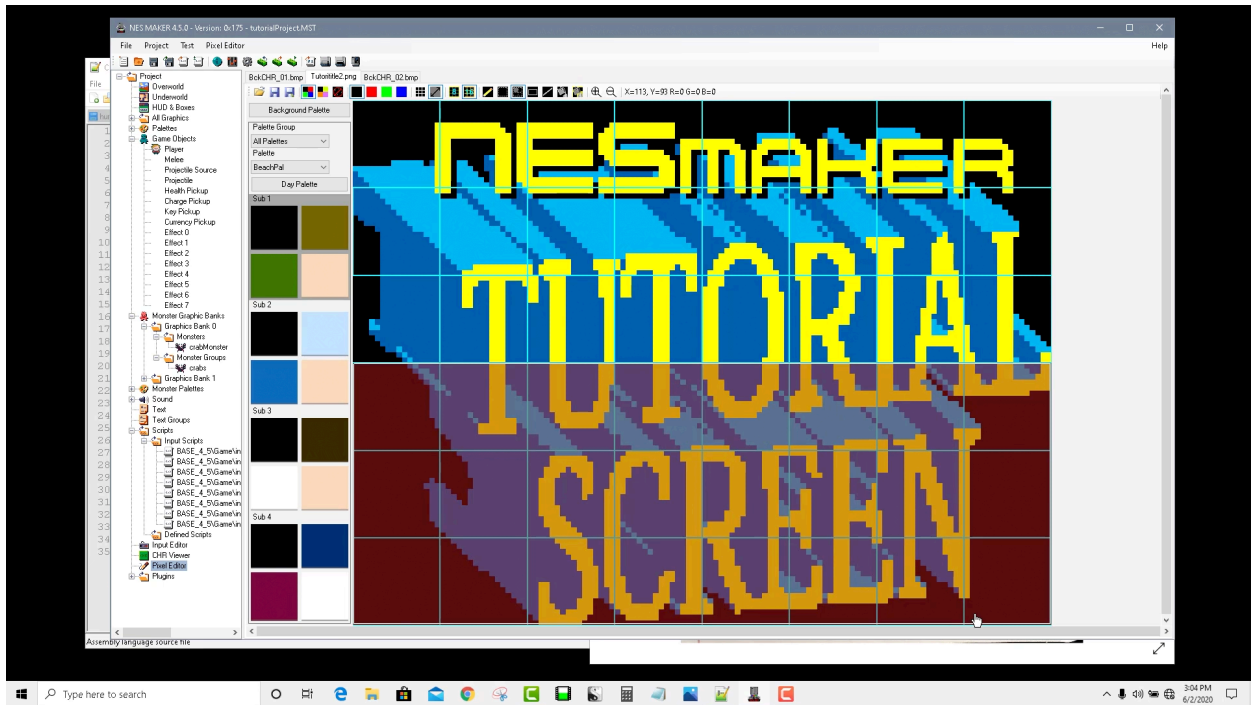
Step 10: In the BckChr_01 tab, move your mouse to the top left grid space and press control-shift-v. This will paste it placed against the grid. Once pasted, make sure to press the save button in the pixel editor menubar.



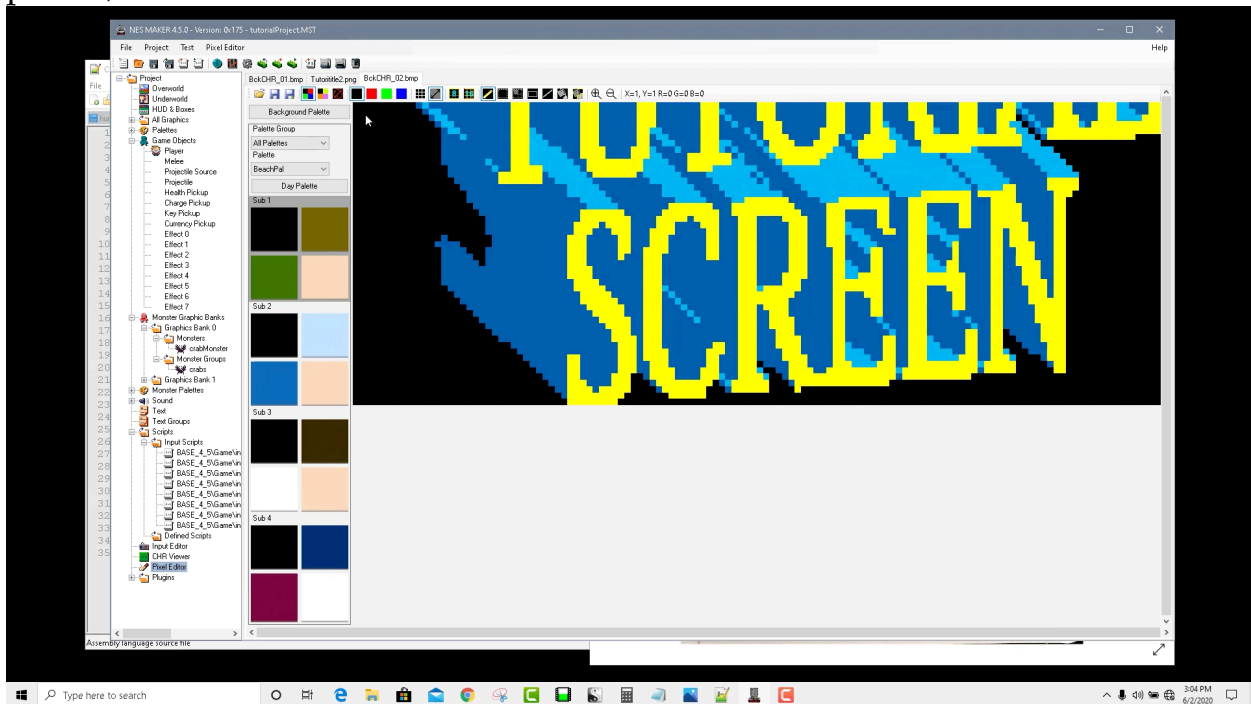
Step 11: Add another tab and open BckCHR_02.



Step 12: Using the Tile Select tool, grab the bottom three rows of the NESmaker Tutorial screen graphic. Use control-C to copy.

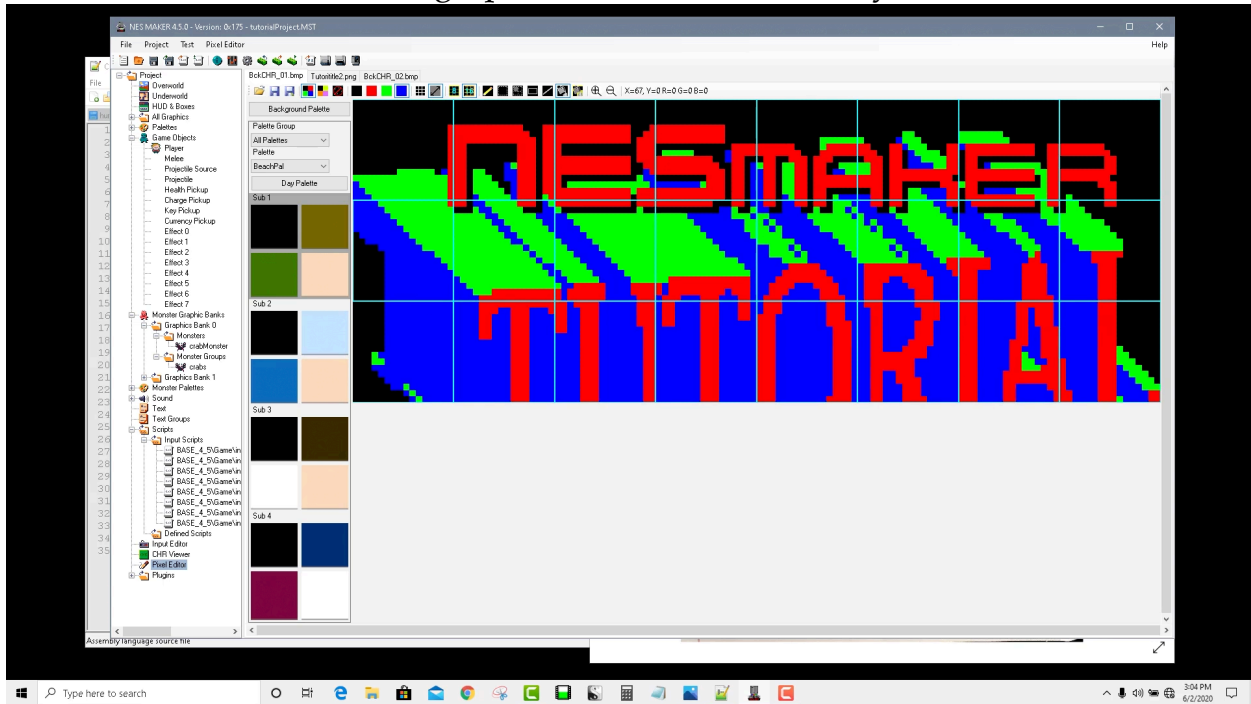


Step 13: In the BckChr_02 tab, move your mouse to the top left corner of the canvas and hit control-shift-V to paste placed against the top corner. Once pasted, make sure to hit save on this as well.

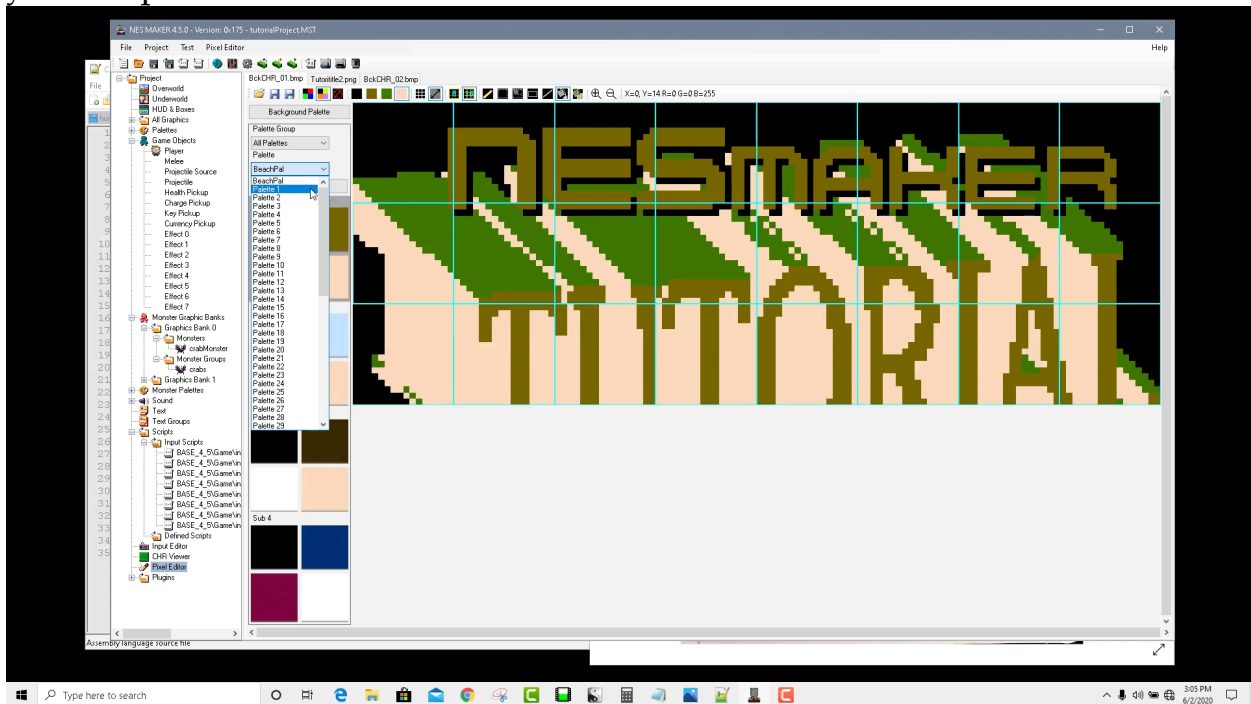


Step 14: Use Disable Palette Translation mode. Use the Global Color Change fill

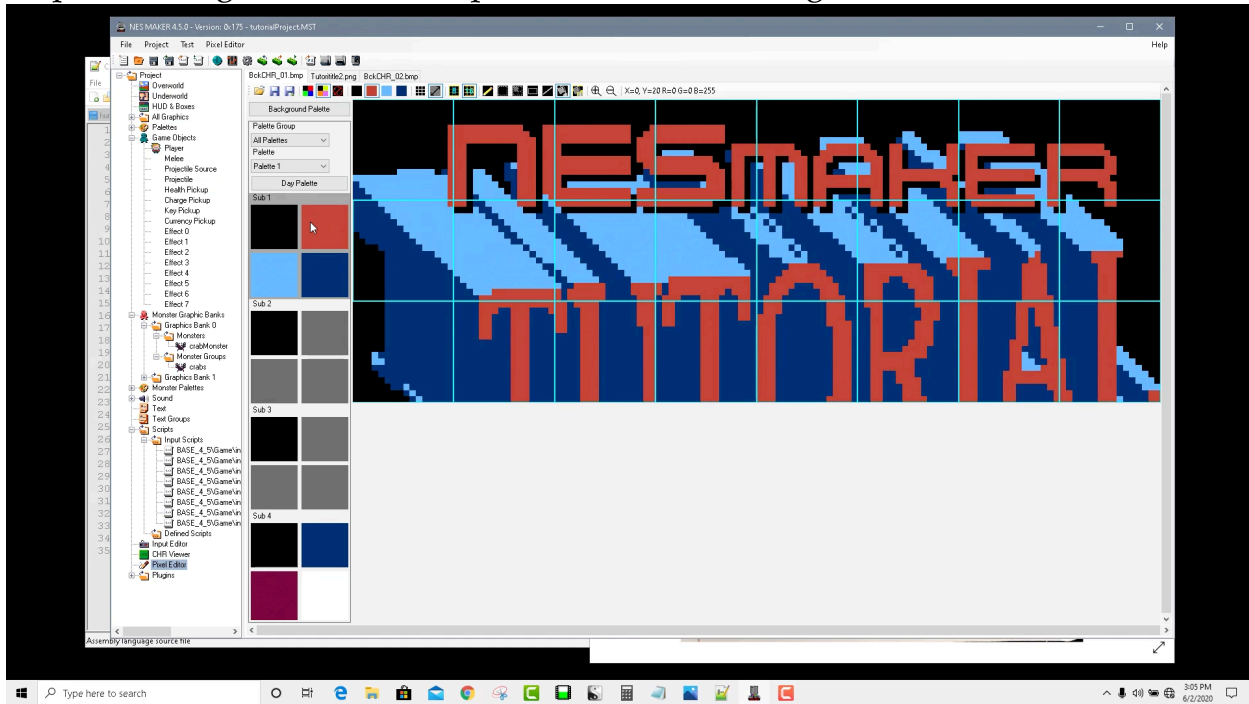
tool to make the three color graphics true RGB-A. When you're done, hit save.



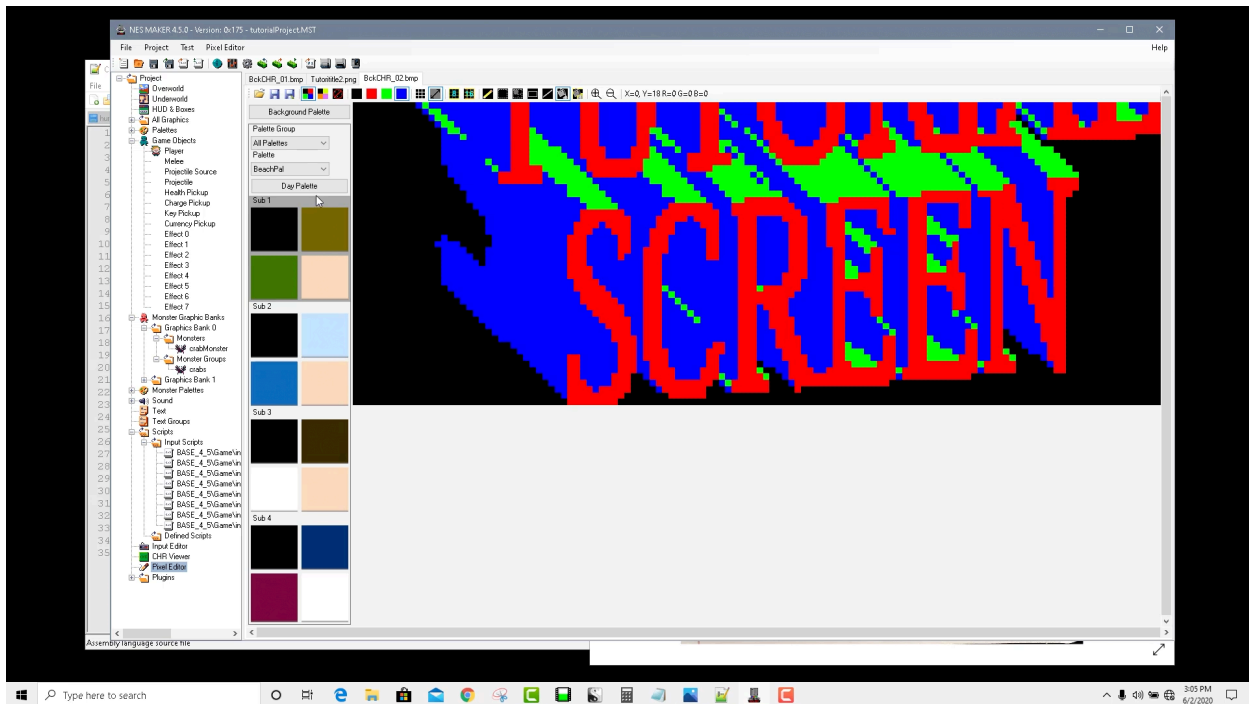
Step 15: Use Enable Palette Translation. It will likely look horrible because it is showing through your beach palette. Find an unused background palette from your dropdown list.



Step 16. Change the colors for palette 1 so that the logo looks as desired.



Step 16: Also make sure to change the bottom half to an RGB-A file using the same method as above and click save.



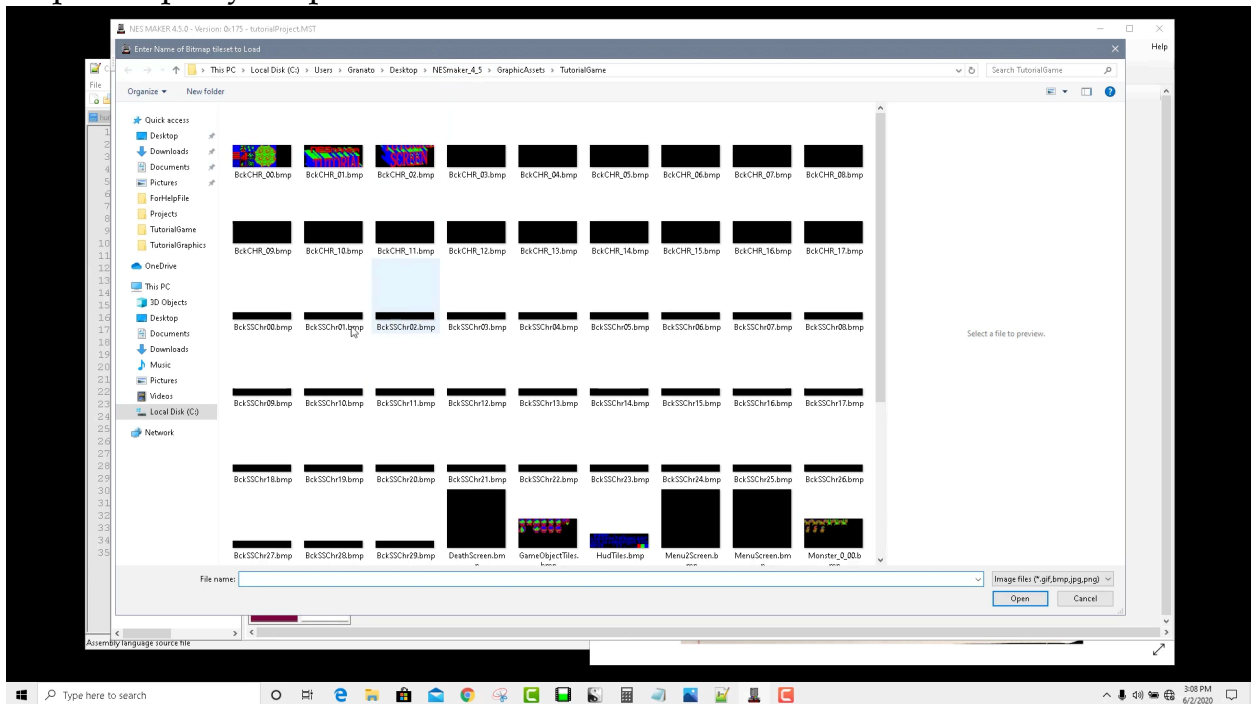
These are both main tilesets. If you remember when looking at the screen painter,

there were various tile layouts that could be used for a screen. A few of them were Double Main. We could make use of a Double Main screen layout, choose BckChr_01 for the first main, BckChr_02 for the second main, and be able to reconstruct these graphics on screen pretty easily. That would work, and in many cases, that might be the best way to do it.

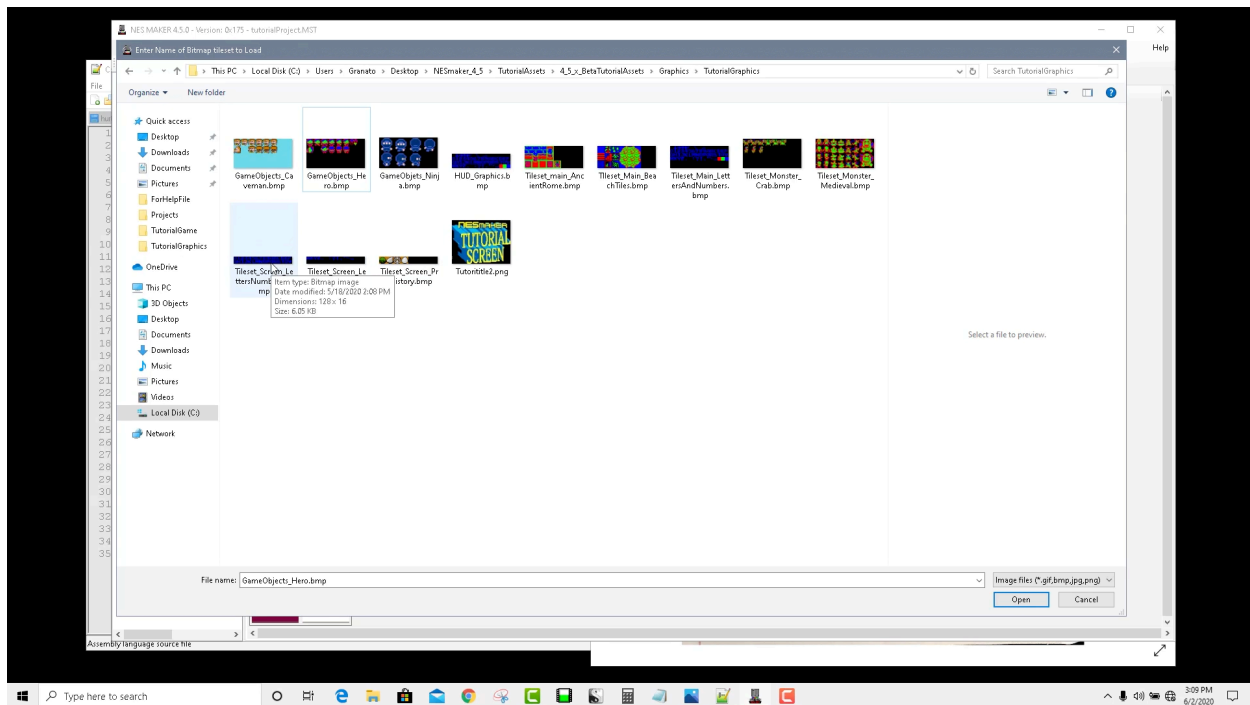
But we're also going to want to add some text at the bottom that says Press Start, and have freedom to use the full alphabet to add copyright or studio information that was common for games to have at the bottom back then.

For this, we'll need to make a screen specific tileset.

Step 17: Open your pixel editor and choose BckSSChr00.

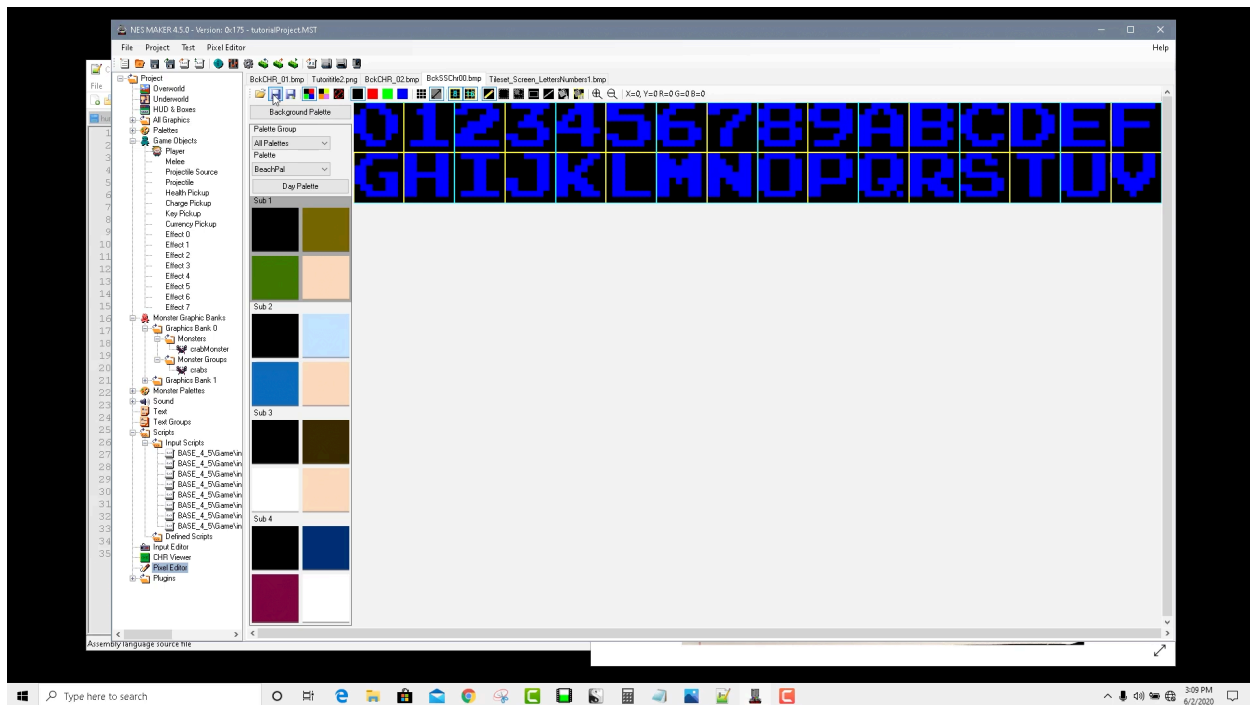


Step 18: Add a new tab in the pixel editor. Use it to open the bmp at the location root / TutorialAssets / 4_5_x_BetaTutorialAssets / Graphics / TutorialGraphics called Tileset_Screen_LettersAndNumbers.



Step 19: It's not the full alphabet, but it's all we need for this project. We could always use a second screen specific tileset to get the second set of letters and numbers, too.

For right now, though, select all of the tiles, hit control-C, open the BckSSChr00 tab, move your mouse to the top left corner, and hit shift-control-V to past them locked to the corner of the canvas. Make sure to hit save.



Now, in the current 16x16px tile mode, it would be impossible to add individual letters or numbers to the screen. We'd end up adding 2x2 letter / number blocks. But there is a screen mode that allows us to draw tiles more granularly. We can set it up in the Map Editor.

The Map Editor

Step 20: Open the overworld map. Let's talk a bit about some of the deeper features of this interface.



You'll notice that a new menu item is added at the top. It has the following options.

- Export Both Maps - this exports all data for both maps.
- Import Both Maps - this imports exported map data and applies it to both maps.
- Export Map Image - this allows you to take a picture of the map and save it as an image file to your computer.
- Erase Map - this clears the map and fills every available space in memory with a null value.
- Advanced Map Properties - this allows you to select sets of screens for more granular tile control. Instead of 16x16px tiles, you can work with 8x8px tiles. The trade off of this is 4x the amount of memory used, so 1/4 the screens for that selected area. Effectively, it turns a 32 screen section (two rows) into an 8 screen section that can be edited at great detail at the tile level. This can sometimes be great for special screen types such as start screens.
- Merge Maps - allows you to import selected data from a map file and place it over the top of selected map data. For instance, if your friend worked on the south part of a map and you worked on the

north part of a map, you could merge his map into yours. The stipulation here is that it is merging the data, and presumes that everything else is set up the same in your projects (tilesets, collision scripts, labels, palettes, etc).

- Show Screen Info - shows some basic information about the selected screen, and allows for spot check editing of some of the parameters.
- Set Map Focus - mostly a legacied function allowing you to quickly view screens with like parameters.

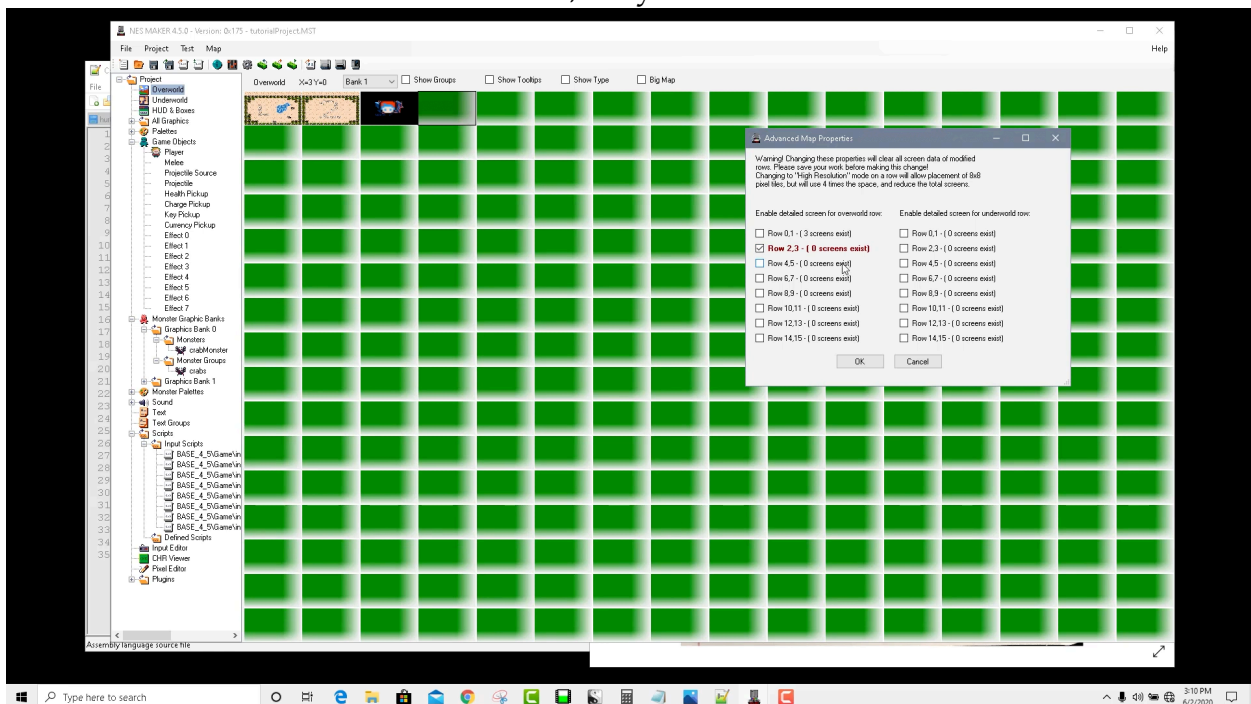
You'll also see a few checkboxes at the top of the screen. They provide the following information:

- Show Groups - while mostly legacies, some game types may benefit from this. You can set up groups with colors and names in the Project menu. With those groups in mind, you can right click on a screen and choose set comment. In that dialog, you can assign a screen a group. When you do, if you have Show Groups selected, you will see an outline around the screen that helps easily identify screens that belong to the same group. This has no bearing on the game, and is only for organizational purposes.
- Show Tooltips - if you have assigned a group or set a comment for a screen, you will see a tooltip popup with that information upon hover.
- Show Type - this will show a numeric display in the top left corner of each screen that denotes the screen types. This can be especially helpful with dealing with triggers and keeping track of what triggers will affect what screen types.
- Big Map - this just expands the map in a way that makes each screen a bit easier to see.

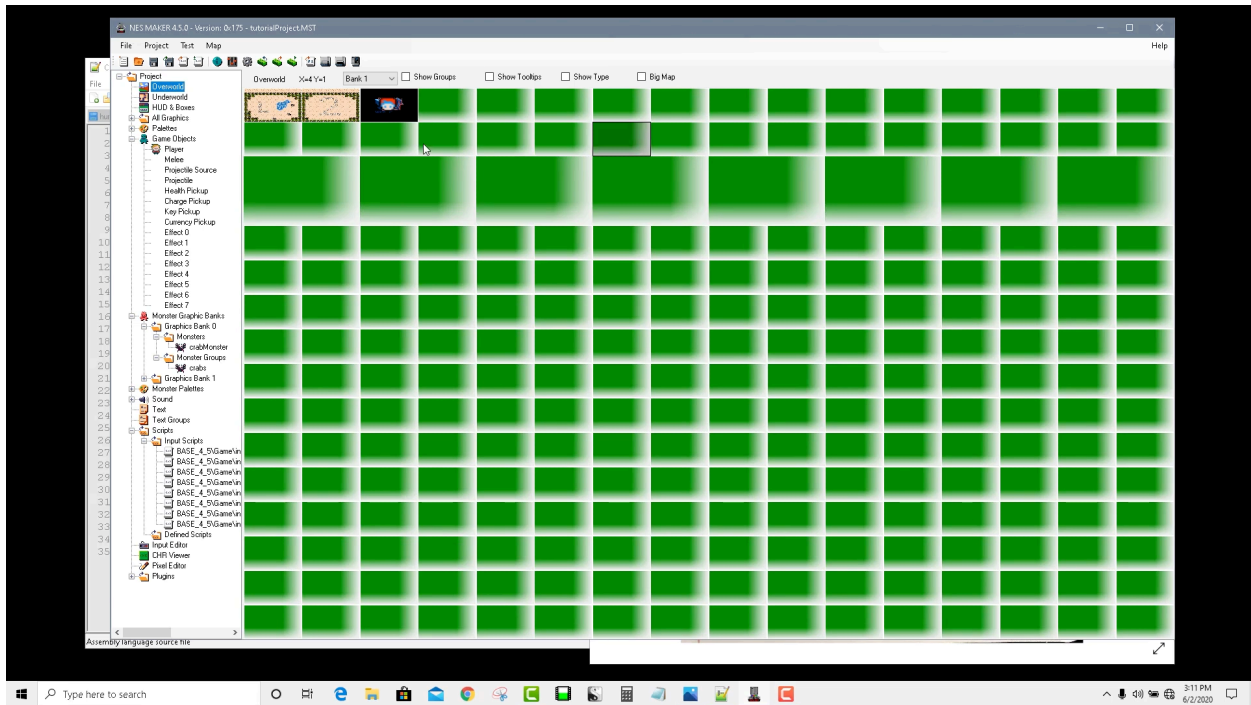
Step 21: We are going to convert two rows of thirty two 16x16px tile screens to one row of eight 8x8px tile screens using the Advanced Map Properties. We can work in greater detail on these screens graphically, with the trade off of taking up 4x the amount of memory (so having 1/4th the screen count for that area). Effectively, you could do an entire project in this mode if your game could

function with 1/4 the number of screens (32 screens per map, 64 screens total). This might be perfectly fine for some games. For our purposes, we're going to use this mode for special screens only.

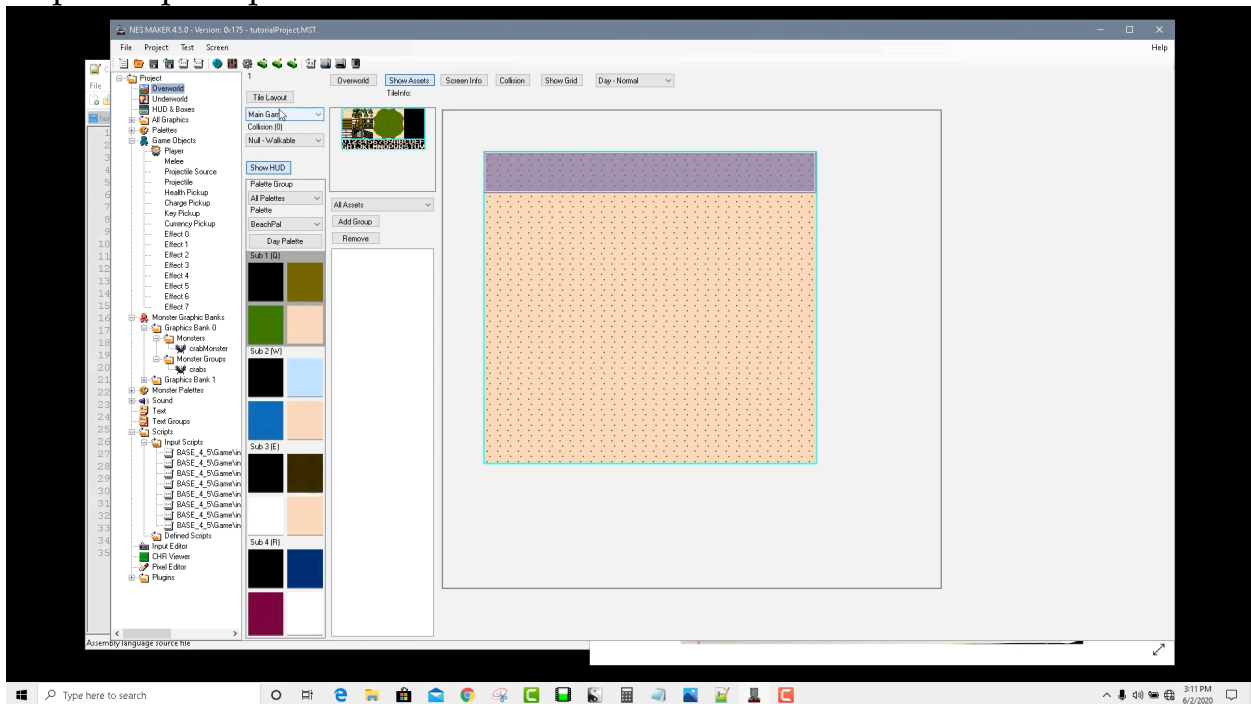
Open Advanced Map Properties. For Row 2-3, check the box. This will take the 32 screens that span rows 2 and 3 and convert them into a single row of 8 screens that can utilize this feature. The small parenthetical lets you know that you currently don't have any screens that this will erase. If you had chosen Rows 0-1, it would've given you a warning that existing screens will be overwritten, but since there are no screens in rows 2-3, they're safe.



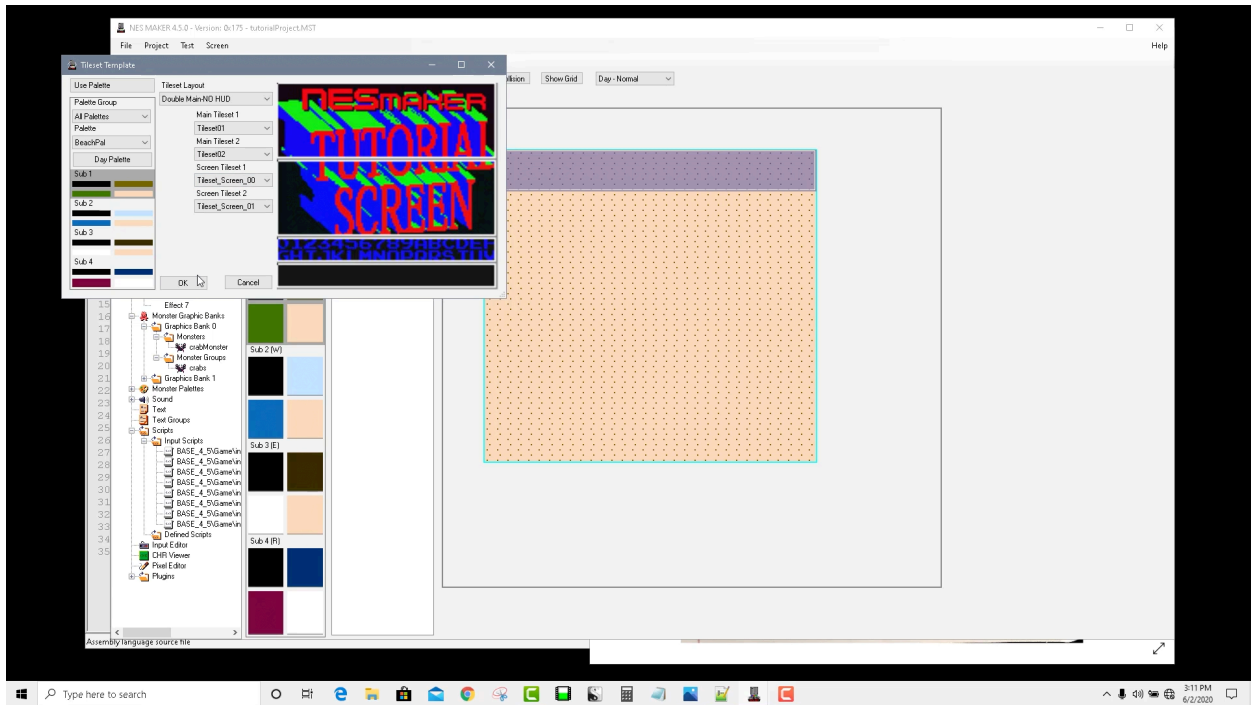
Press ok. You'll see that the area for those rows has changed. You now only have access to the first four screens in that map space, but they can contain 4x the amount of tile data.



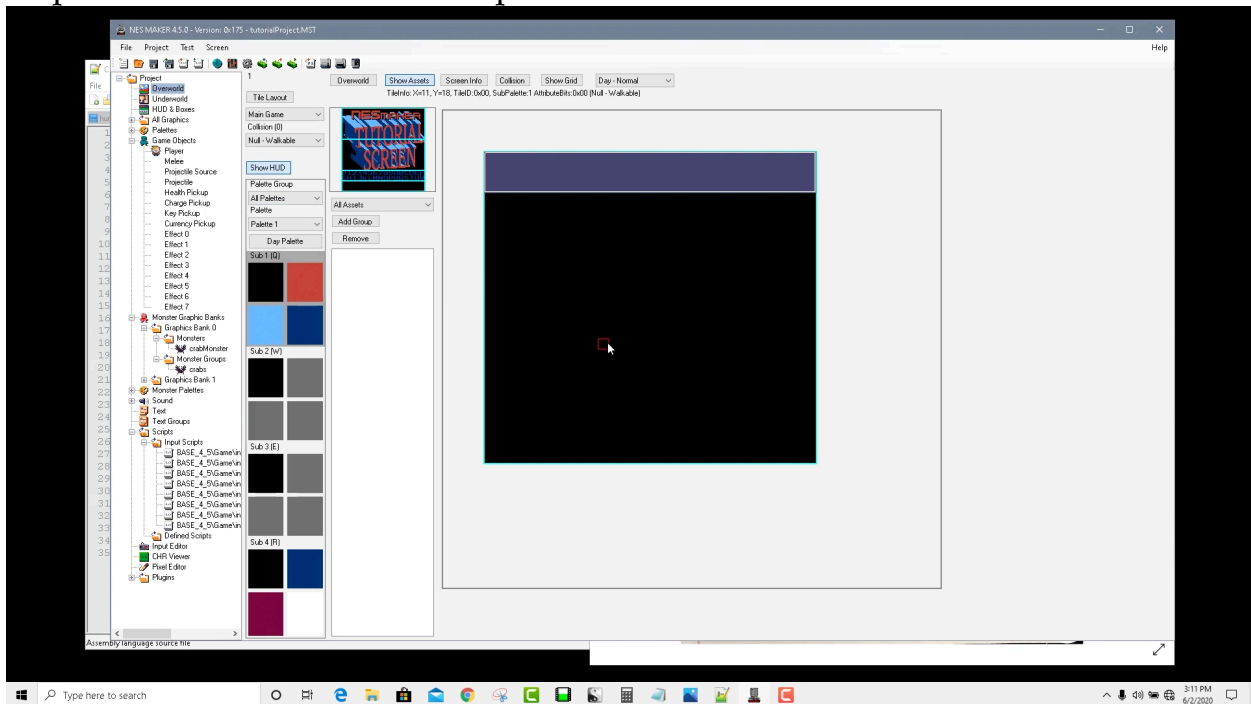
Step 22: Open up the first screen in that row.



Step 23: Go to Tile Layout. Choose Double Main - No Hud. Choose Tileset 01 and Tileset02, and it should automatically select the first screen specific tileset at the bottom.

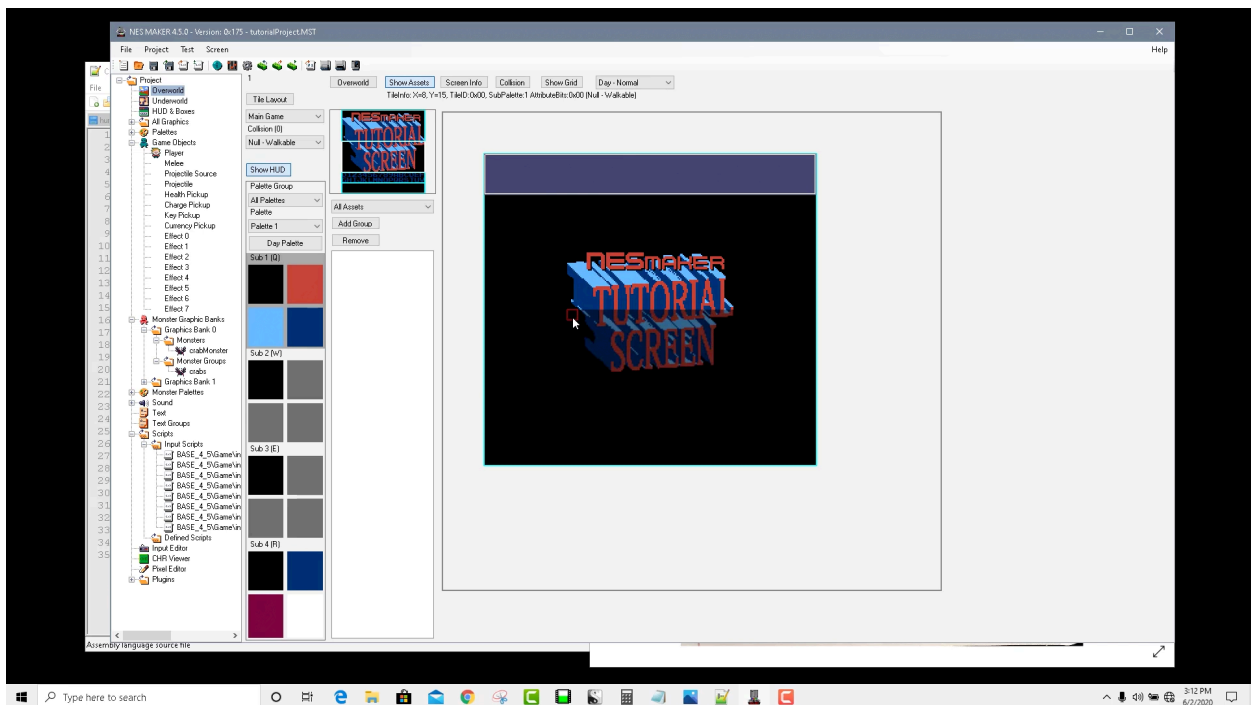
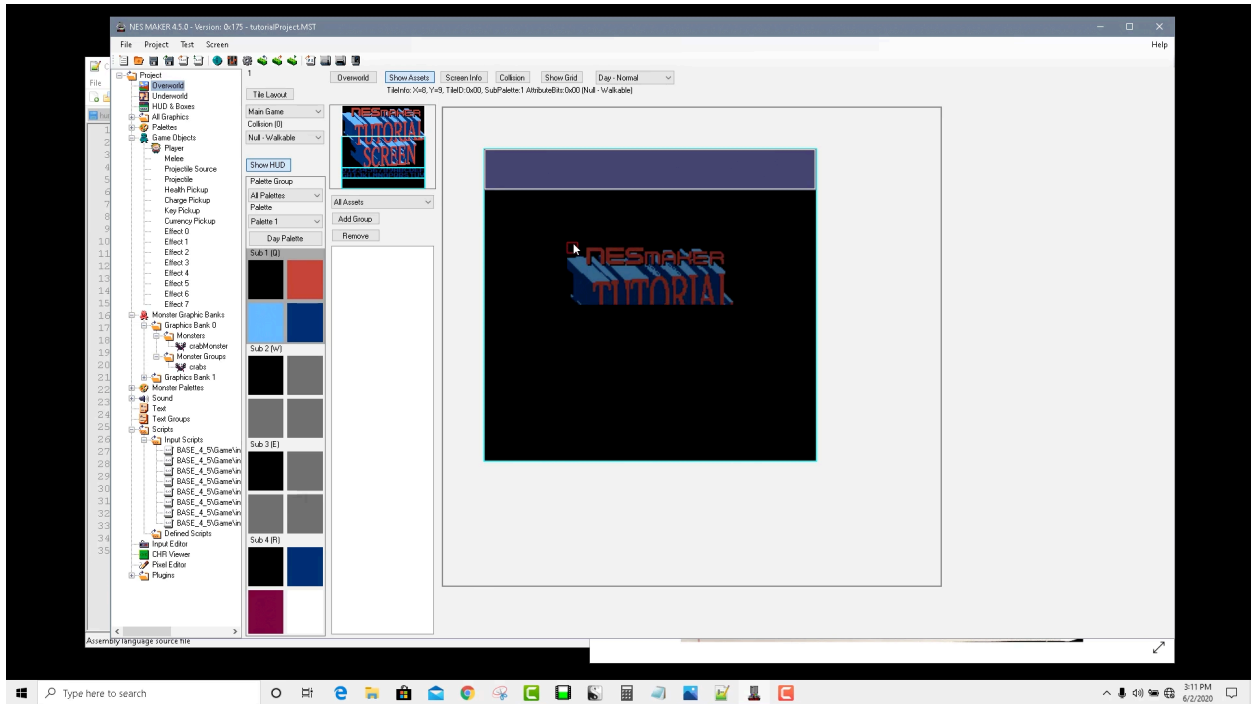


Step 24: Choose the start screen palette that we made.



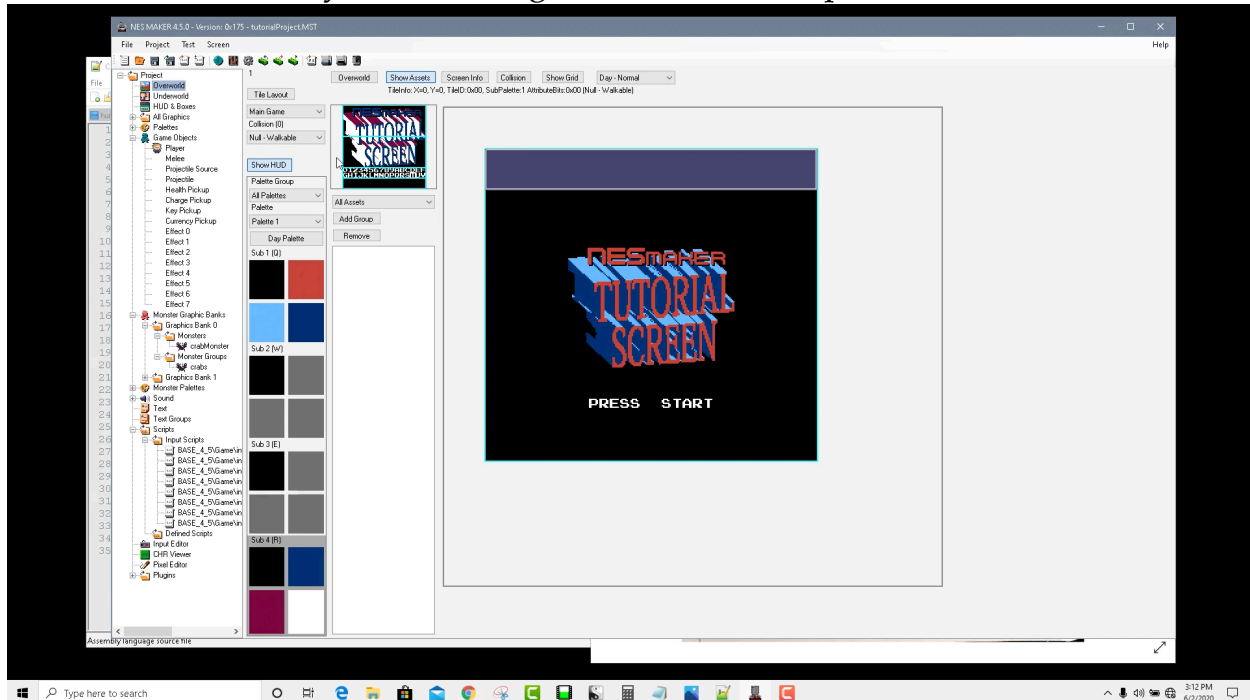
When you go to select tiles from your tileset, you'll notice you no longer have a 16x16px tile grabber, but instead, it's allowing you to select 8x8 tiles. Proceed the same way as you did before. Shift-click and drag over the top tileset and paint

that onto your screen's canvas, then do the same with the bottom, lining it up.

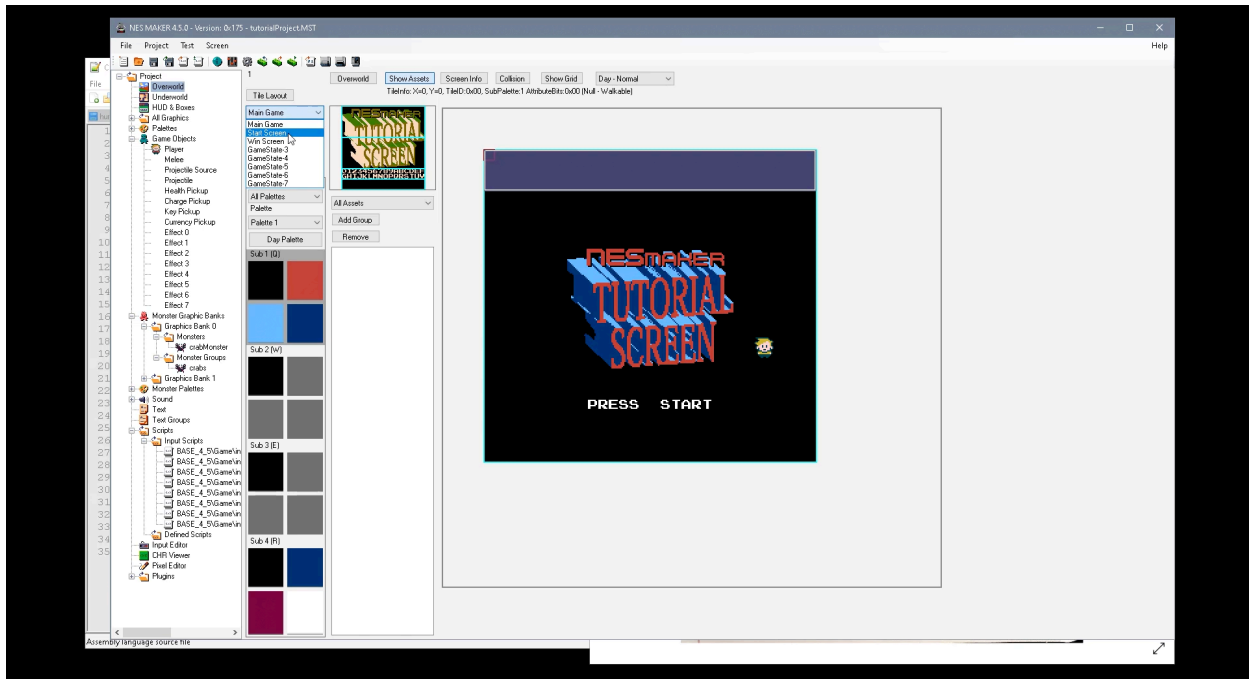


Step 25: Choose the last sub palette and character by character, add the words Press Start at the bottom. If we had been in 16x16px mode, it would've been

impossible to grab individual letters like this. You could continue on to add copyright information or your studio name at the bottom. Since you have free access to 8px tile drawing in this mode, you can add letter by letter whatever you need the screen to say. This is one great use of the 8x8px screens.



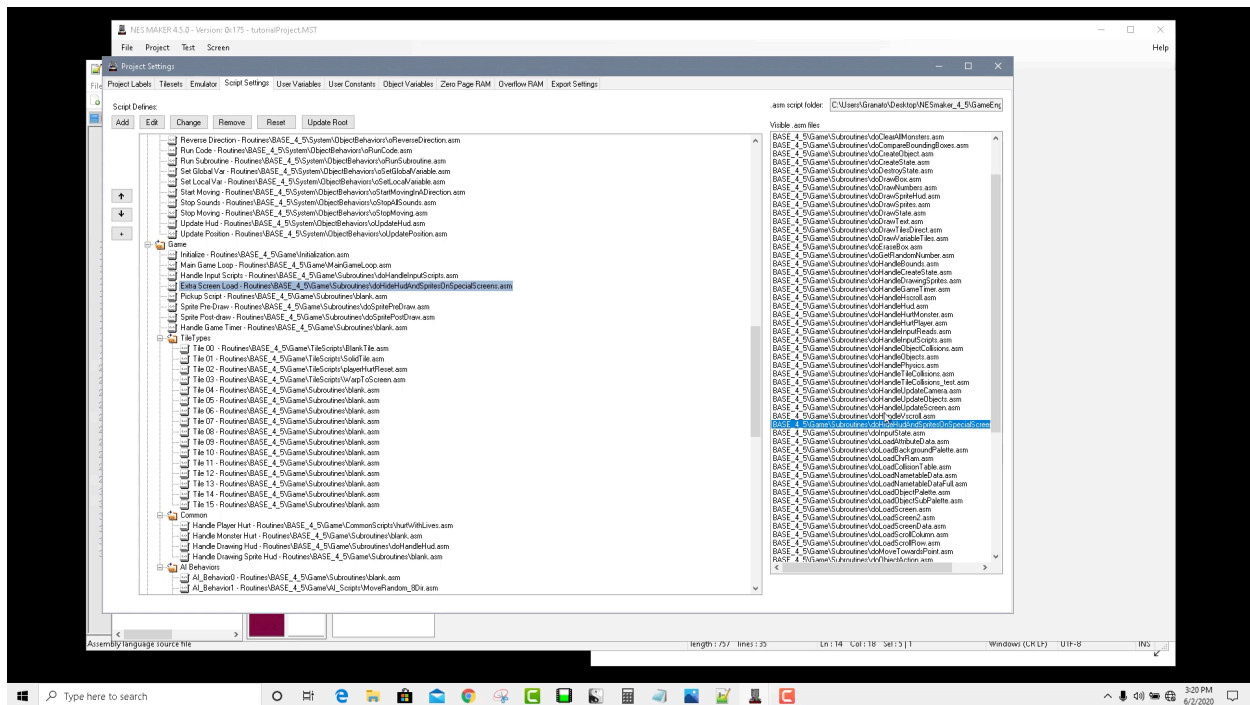
Step 26: Somewhere on the screen, right click and select place player. Also, make sure that we set the Game State to Start Screen. There will still be a few problems - for instance, we don't want to see the player on this screen, and we don't want to see the HUD on this screen. But we'll get to that in a moment.



First, test the game. You should now see the start screen. When you press start, it should take you to the game. If you run into the monster, it should re-start your level with one less life. If you run out of lives, it should take you back to the start screen to denote that it has restarted the game.

Step 27: We want to create a method for complete control on a screen by screen basis as to whether or not our sprites show up and if our hud shows up. On special screens like this, we probably don't want the sprite and we don't want the hud. Maybe on a cut scene, we do see our sprite, but not our hud. Maybe on a win screen, we see the hud but not the sprite. We want to run a little extra code during the screen load that can check to see what to do.

Go to your script settings and scroll down to game. There, you'll see a script define called Extra Screen Load. It is currently blank. If you go to the script browser and go to root / game / subroutines, you can find one called doHideHudAndSpritesOnSpecialScreens. Double click to add it.

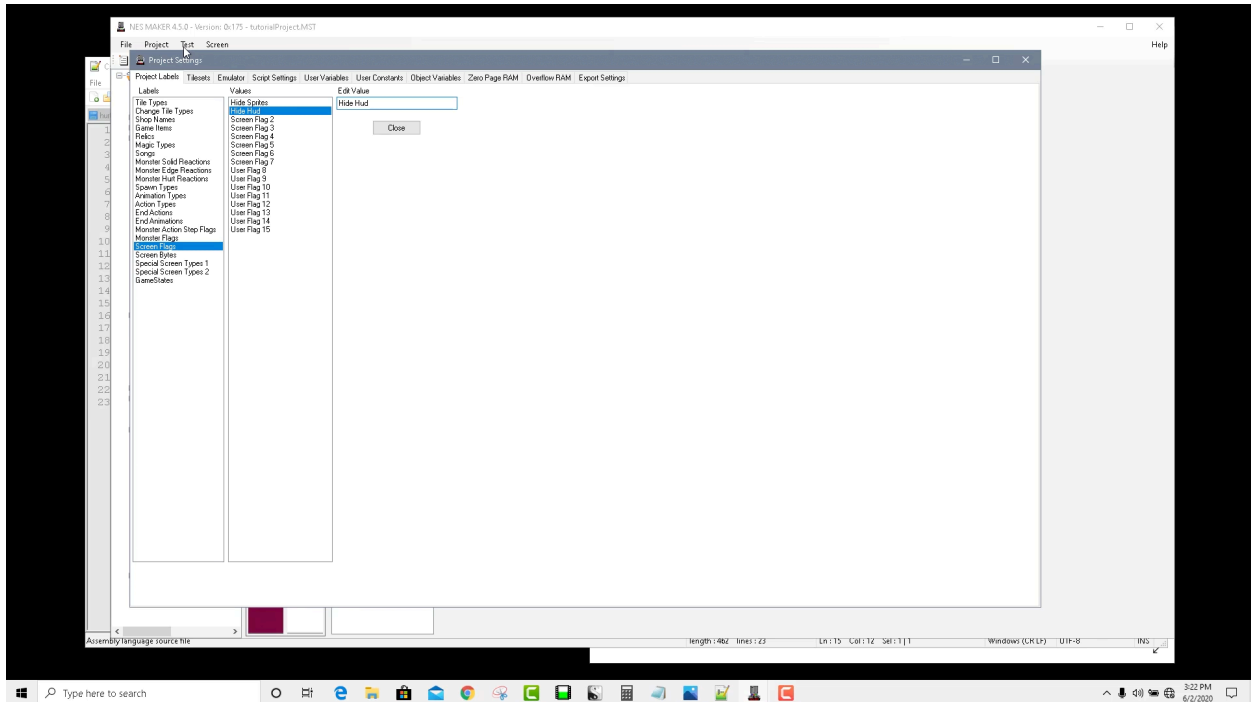


What this script does is that it looks at a variable that already exist in the module called ScreenFlags. This variable is a byte of data, meaning its a series of ones and zeroes. How each value, called a bit, for that byte are set will determine what happens in the script. For this particular script, it checks that variable to see the state of these bits - one of the bits controls whether or not we see sprites. Another controls whether or not we load a HUD. If it sees a zero, it ignores that part of the script. If it sees a one, it does that part of the script.

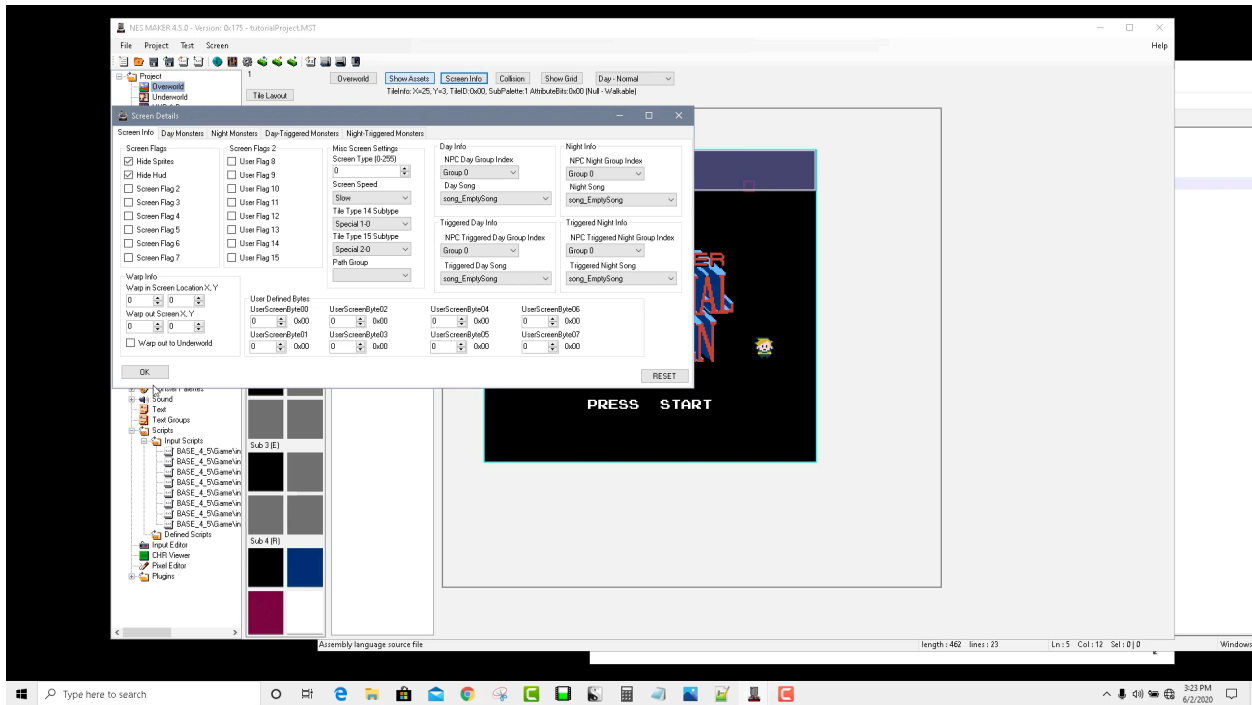
But those are specific functions of these flags to this module. Every game might make use of these differently, just like every game might use different tile collision scripts, just like every game might use different AI Behaviors. So just like in those cases, we will create labels for these Screen Flags so that when we see them in our screen info, it's easy to remember what each does. Again, don't forget, the only reason ticking these boxes on the screen info will have the effect is because of the script we just attached to the Extra Screen Load define. Without that script, they would not have this effect. And with a different script, they may have a completely different effect. That's why it's important to be able to re-label them as needed.

Step 28: Go to Project Labels. Click on Screen Flags. Name the first screen flag

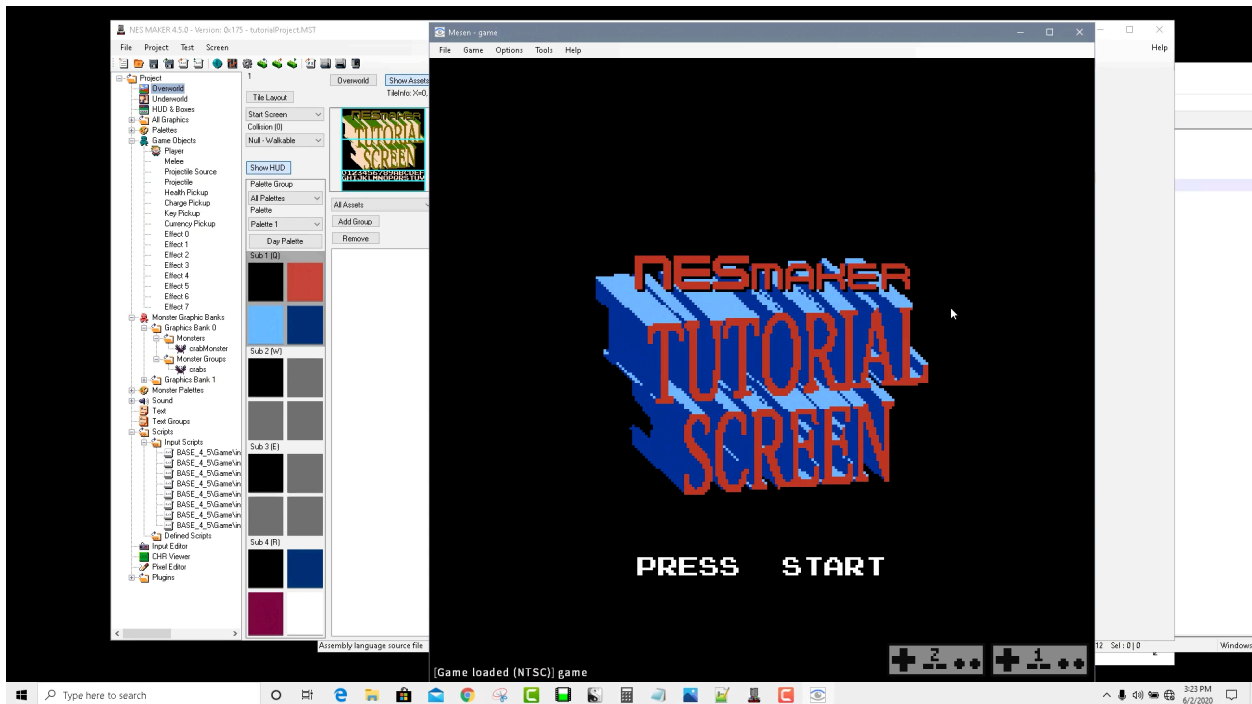
Hide Sprites and the second screen flag Hide Hud.



Step 29: Open your start screen and go to screen info. You'll now see that the first two screen flags are named as we just named them. Now, ticking Hide Sprites will cause this screen, upon loading, to hide the sprites. Ticking Hide Hide will cause this screen to ignore the HUD load. Select them both for this start screen.



When you test your game, the HUD and the sprites will be invisible.



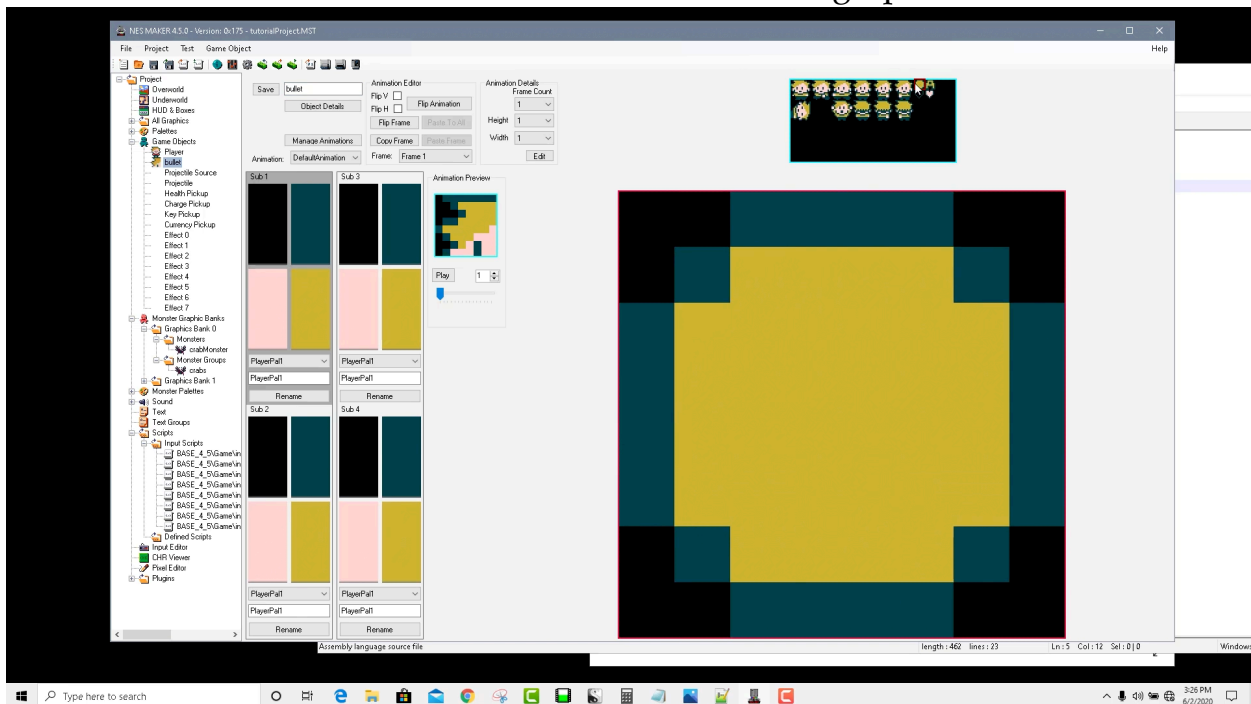
Creating a Projectile Object

Creating a Projectile Object

As fun as it is to play keep away from the monsters, it was always in our design to be able to attack them.

Step 1: Click on the Game Object that currently says melee. As we talked about before, the name of these things is rather incidental. This being called melee does not mean that it is a melee object. It's a suggestion for the types of things you might find as game objects. For us, we're going to use this as our bullet or fireball or whatever it is that Greg the Time Traveler creates to exterminate the crabs.

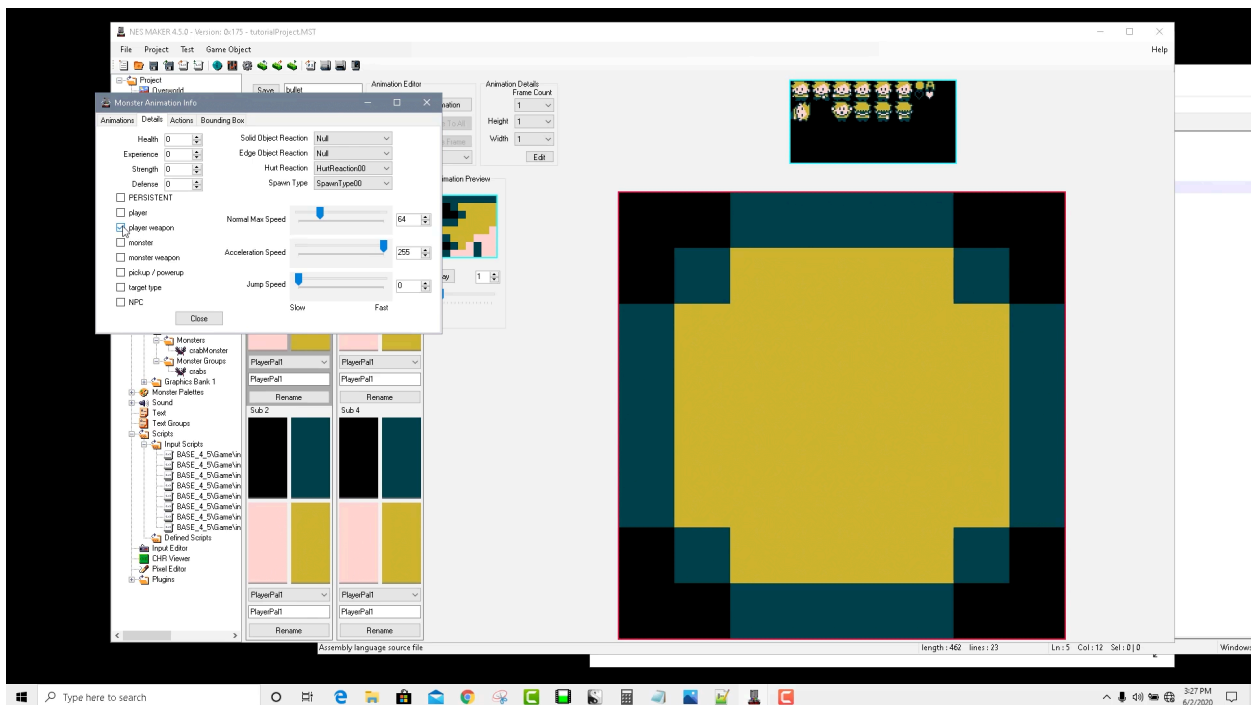
Rename it Bullet and choose the circular tile for its graphic.



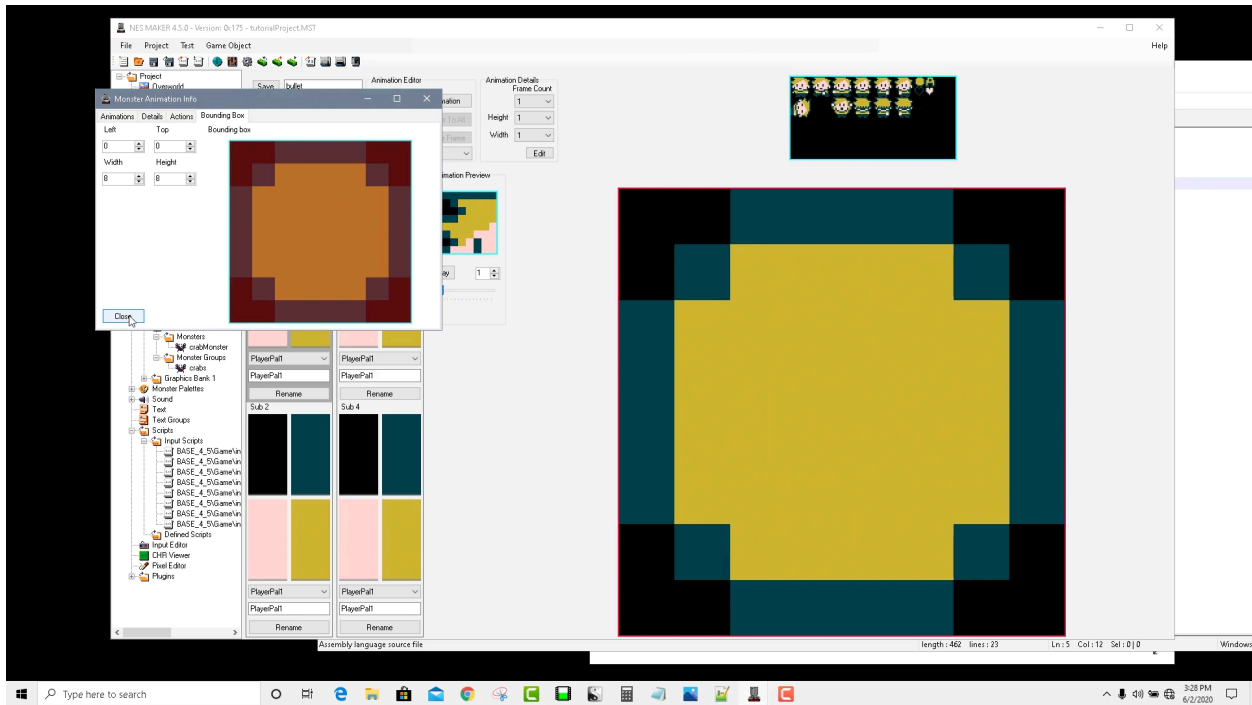
Step 2: In Object Details, click on the Details tab. Set this to a Player Weapon object type. What this means is that it will observe collision with the monster using this module, but it will not trigger the player hurt script. Instead, it will trigger the monster hurt script.

Also, give it a speed of around 64, and turn the acceleration all the way up.

As far as its Edge and Solid collisions, neither really express what we want to happen. We don't want it to just stop, like the player does when he hits a solid wall. We also don't want it to reverse direction and start bouncing around the screen like a super ball. While the latter could make for an interesting mechanic, it's also problematic considering that this engine can only draw so many objects on the screen at a time. If you add too many, you risk slow down or full on crashing of the game. So we'll come back to this, as we're going to need to add another reaction type.



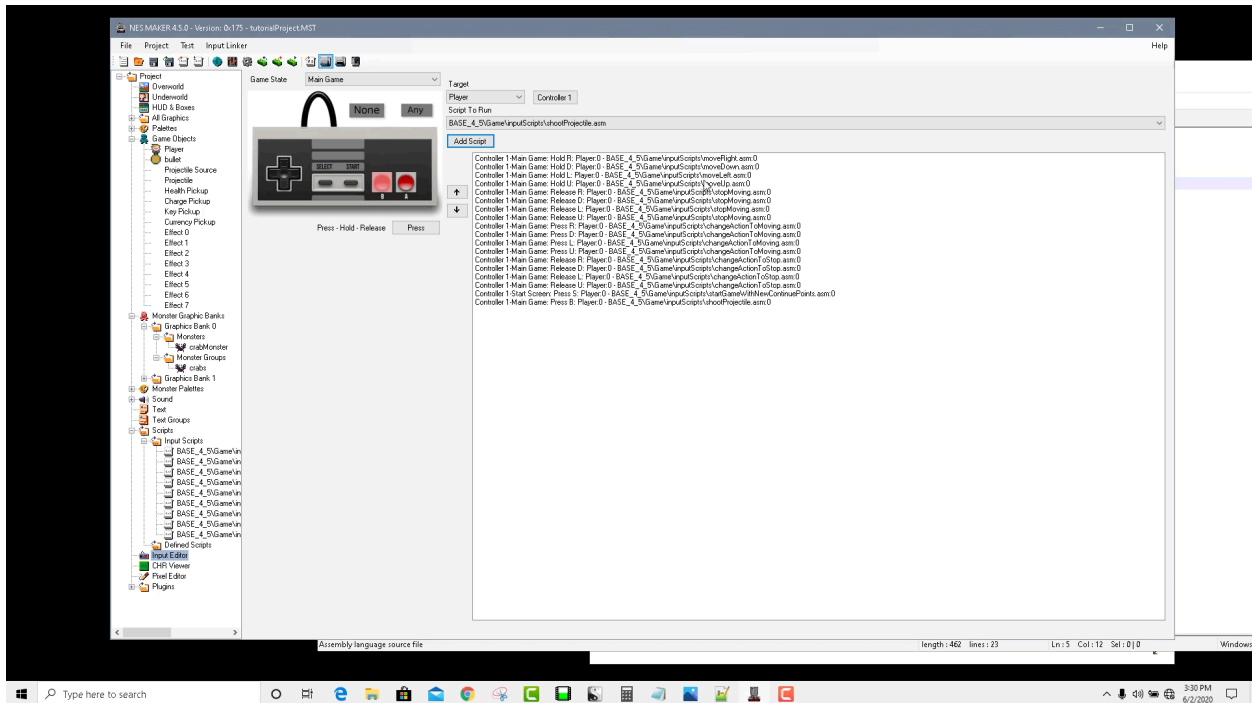
Step 3: Click on the Bounding Box tab and select the whole area.



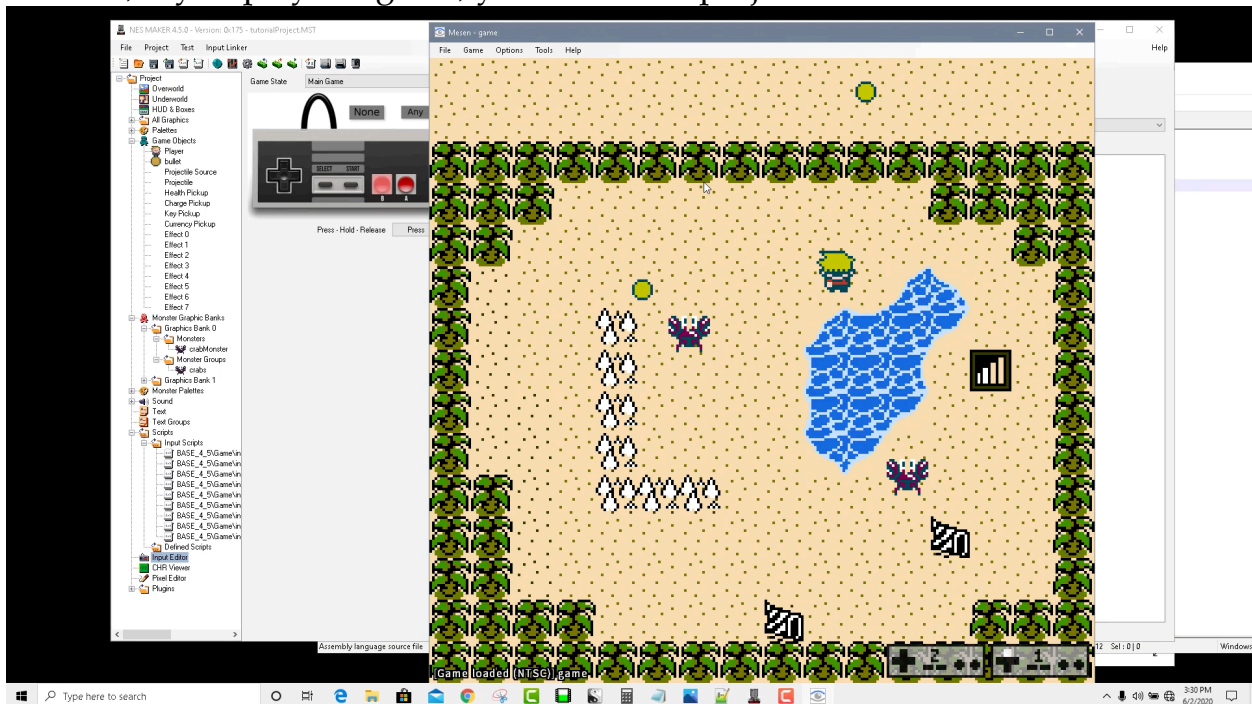
Make sure to hit save on the object, and you'll see that the graphic for it updates in your Game Objects folder in the hierarchy.

Step 4: Click on the Input Scripts node inside the Scripts node in the hierarchy. Navigate to root / game / InputScripts and double click on shootProjectile to add it to your project and make it accessible to the input editor. This script gets the player's position on the screen, saves his direction, creates this new object (game object 1) at that player's position, and sets it to move in that saved direction.

Step 5: In the Input Editor, we want to set up a new line. We want to say that for the Main Game (game state), when the B button is pressed, for the Player object, run the shootProjectile script. Set up this input and press Add script.



Now, if you play the game, you can shoot projectiles around the screen.

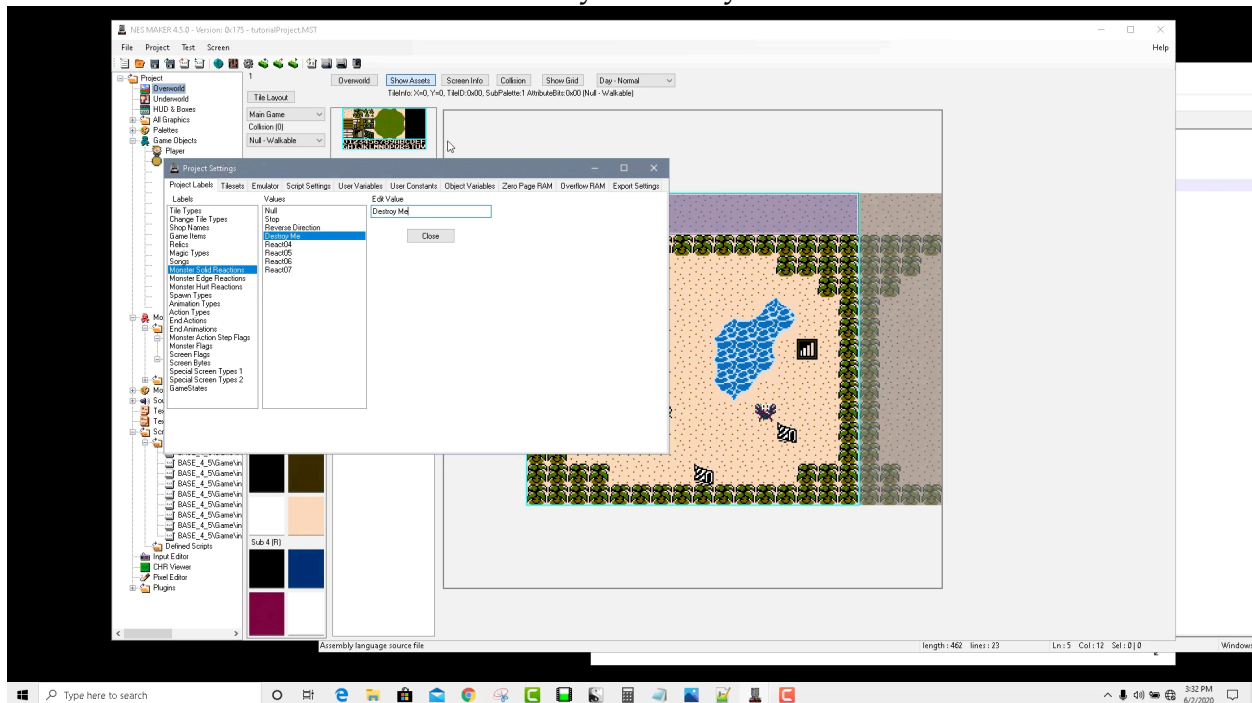


There are still lots of problems. Nothing happens when the hit the monster, because we haven't told the game what to do when a player weapon hits a monster yet. Nothing happens when they get to a solid object or the screen edge

because we haven't made those reactions yet. And it doesn't utilize our ammo system.

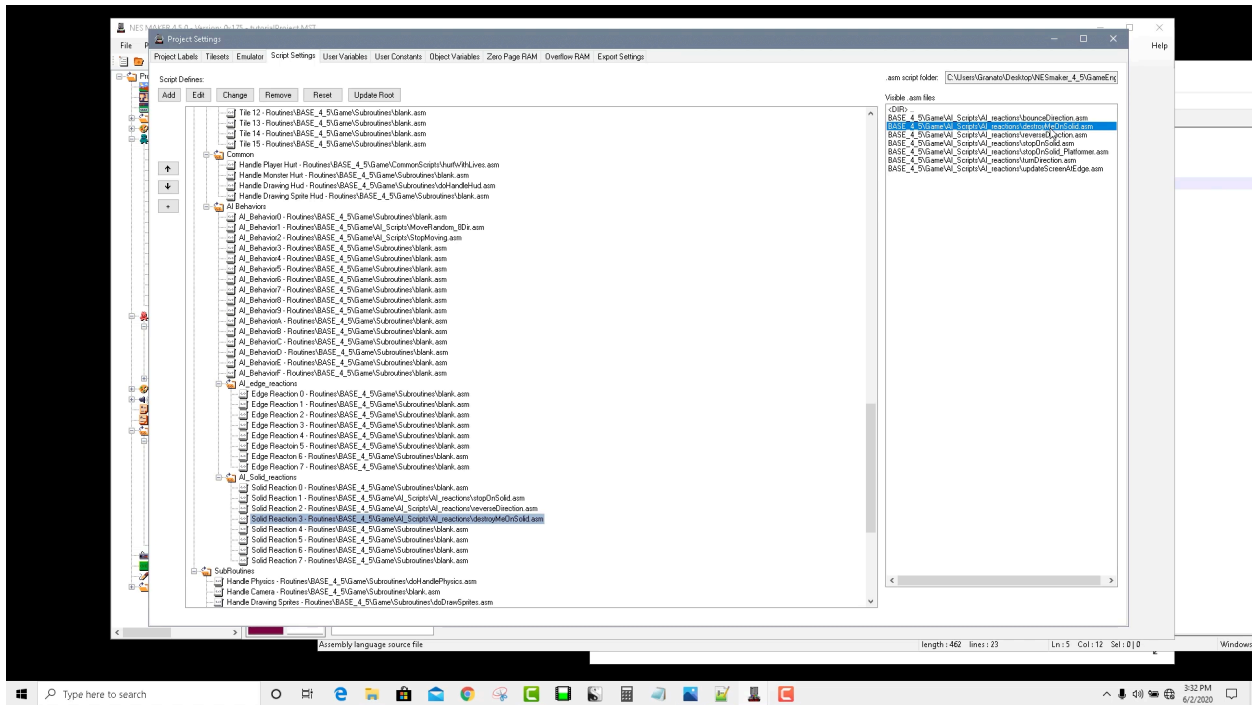
Step 6:

Open up Project Settings, then on labels, click on Monster Solid Reactions, add a label to the first free slot that says Destroy Me.



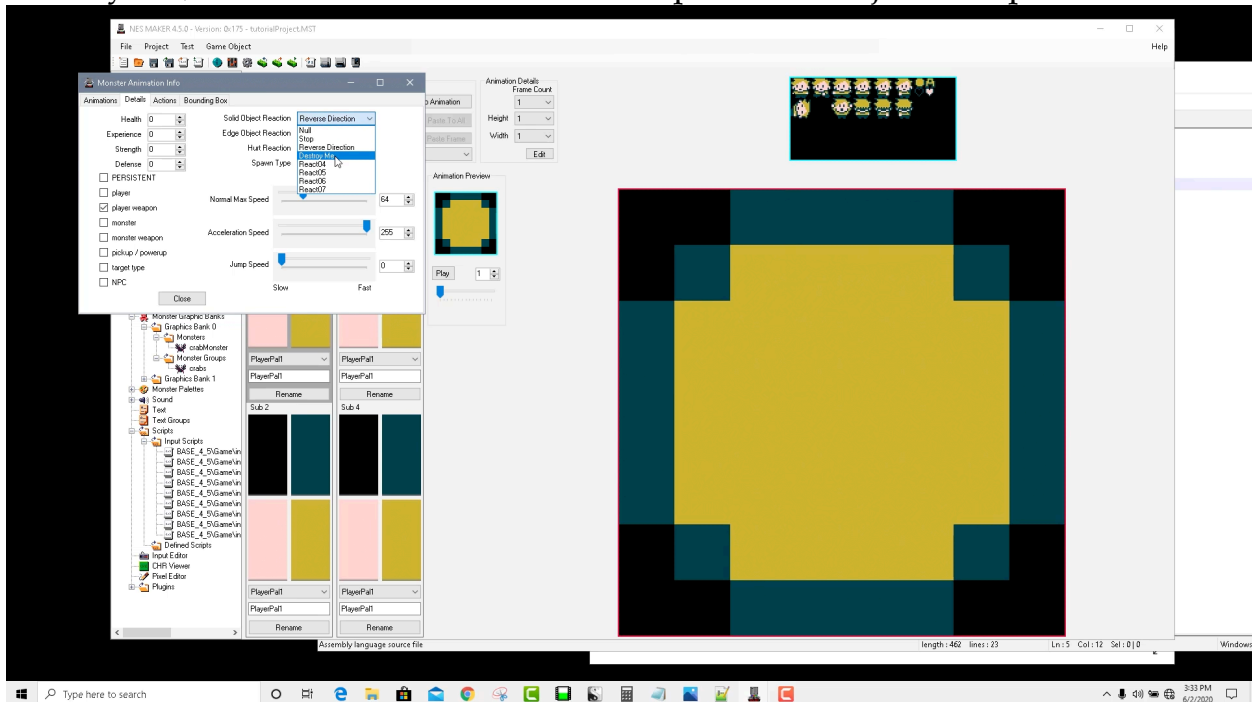
Step 7:

Go to Script Settings. Scroll down to AI_Behaviors, inside AI_Solid_Reactions. Click on the corresponding Solid Reaction, which is Solid Reaction 3. Then, in your script finder, go to root / game / AI_Scripts / AI_Reactions and double click on DestroyMeOnSolid.



Step 8:

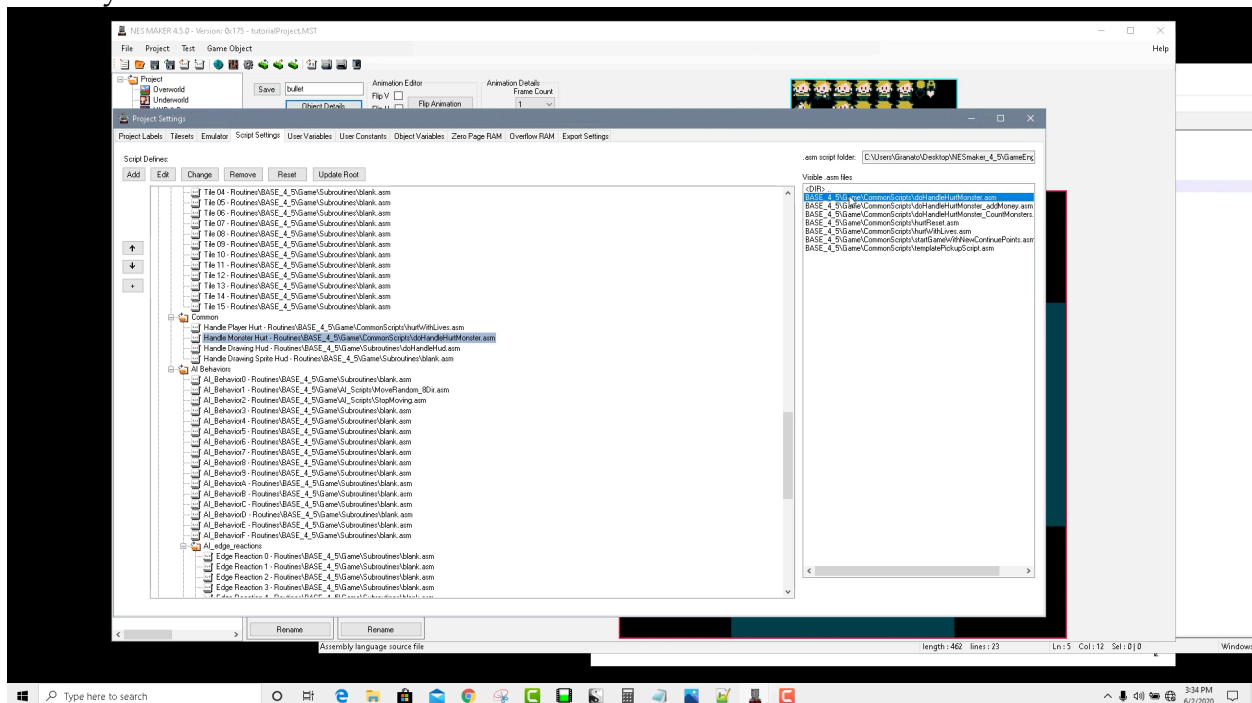
Click on your Bullet object in your game objects folder. Open Object Details. On the Details tab, click on the drop down for Solid Object Reaction and choose Destroy Me, which is the new label and script define we just set up.



Run your game, and now solid objects should destroy your bullets.

Step 9: Now, we want to make the monster go away when you hit him with the projectile. Open Project Settings and click on the tab for Script Settings. Scroll down to Common and you'll see a script called Handle Monster Hurt. In this module, this is what happens to the monster type object when it has a collision with a player weapon type object. Right now, you'll see that it's blank, which is why nothing happens to the monster.

Use the script finder to go to root / game / commonScripts, and double click on doHandleHurtMonster. This particular script simply causes the monster to be destroyed.



Test your game. You should now be able to successfully shoot your monsters.

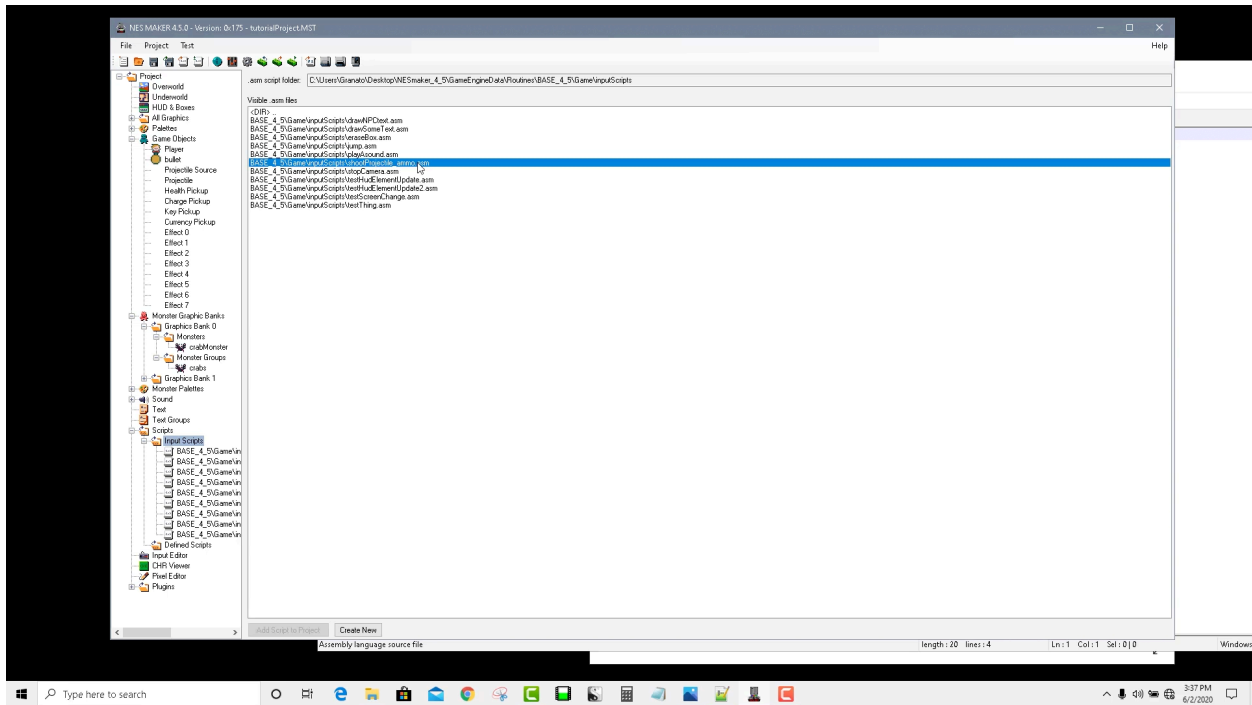
Step 10: This works great, but it would be a lot more interesting if we had to be a bit more conservative with our shots. Our design document calls for us to make use of the ammo. We should only be able to create that bullet projectile if we have ammo, and reduce one value from the “myAmmo” variable. If we're out

of ammo, the b-button should no longer have any effect on the game.

We do have a script for that, and we'll have to replace our current b-button script with it. Click on the Input Scripts node in the hierarchy. Navigate to root / game / InputScripts, and double click on shootProjectile_ammo. It will show up in your list, and now you'll be able to access it in the input editor.

This script does three important things. First, it does the same create object function as the normal create projectile script, but only if the variable myAmmo is greater than zero. It also subtracts one value from the variable myAmmo. Lastly, it updates the tiles drawn to element 3 of the HUD to reflect that ammo change. It's easy to forget that is happening, because for us as the game player, we understand it implicitly - shoot, lose ammo, and the HUD shows us how much is left. But with our game engine, we must refresh the tile to reflect the new number that corresponds with the variable myAmmo, otherwise even though the variable itself will change (when it gets to zero, we can't shoot anymore), we won't see its value change in the HUD.

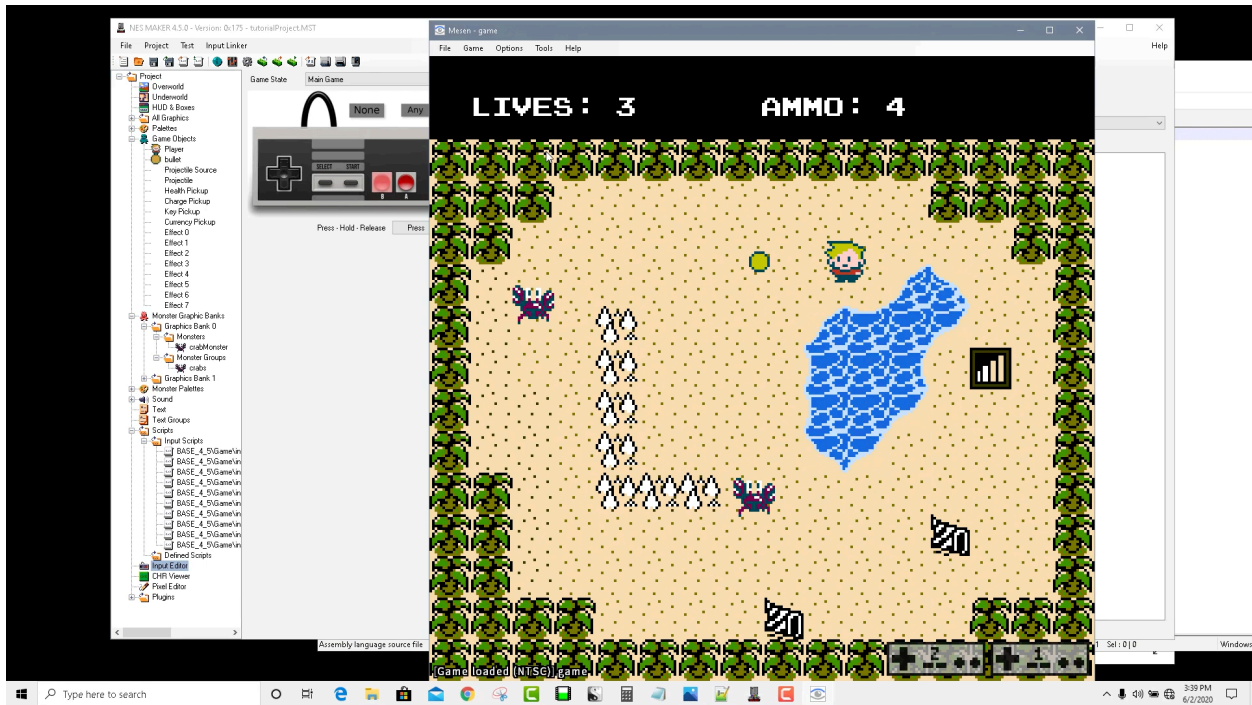
This script does all those things for you, but it's good to understand. We might later want to create a script that redraws a different variable, and for that we would have to look at the script to change which element to update. That's nothing we have to worry about for this instructional, though.



Step 11:

In the input editor, delete the current createProjectile script for your b-button by right clicking and removing the script. Then, set up all the same settings - Main Game, B-Button Press, Player object, except this time choose the new shootProjectile_ammo from your dropdown.

Run your game and you'll see that ammo is observed.

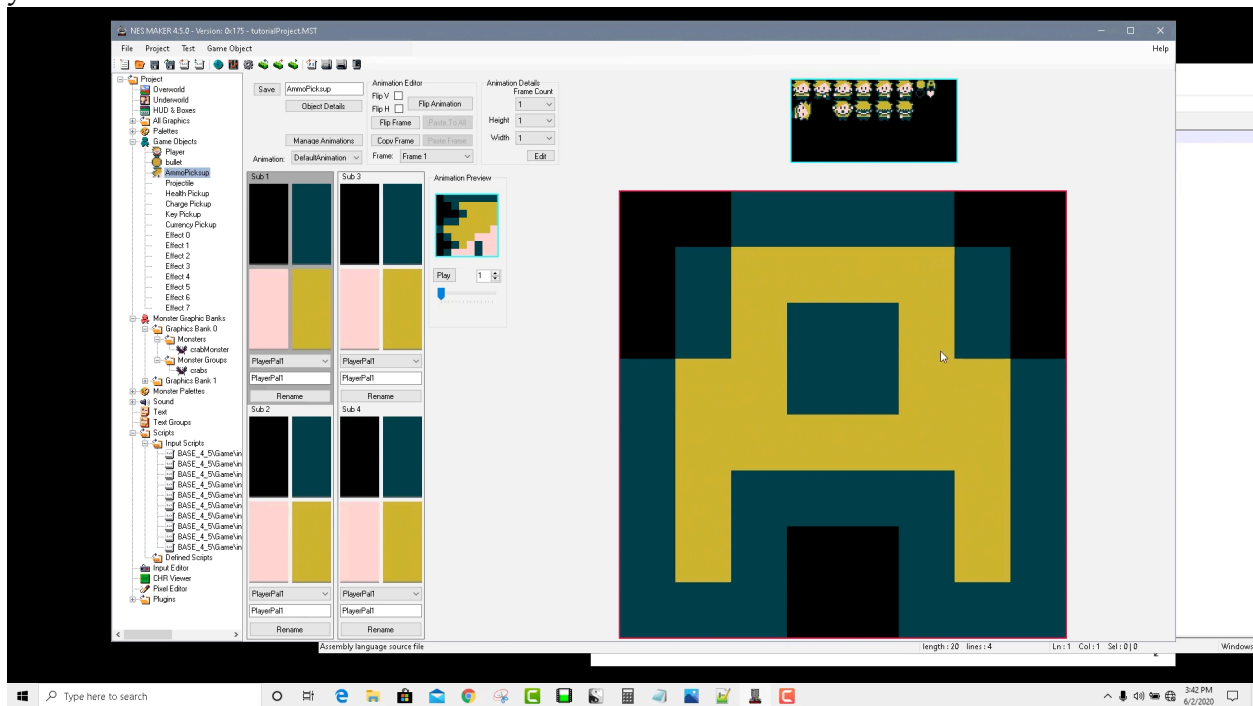


Creating a Pickup Object

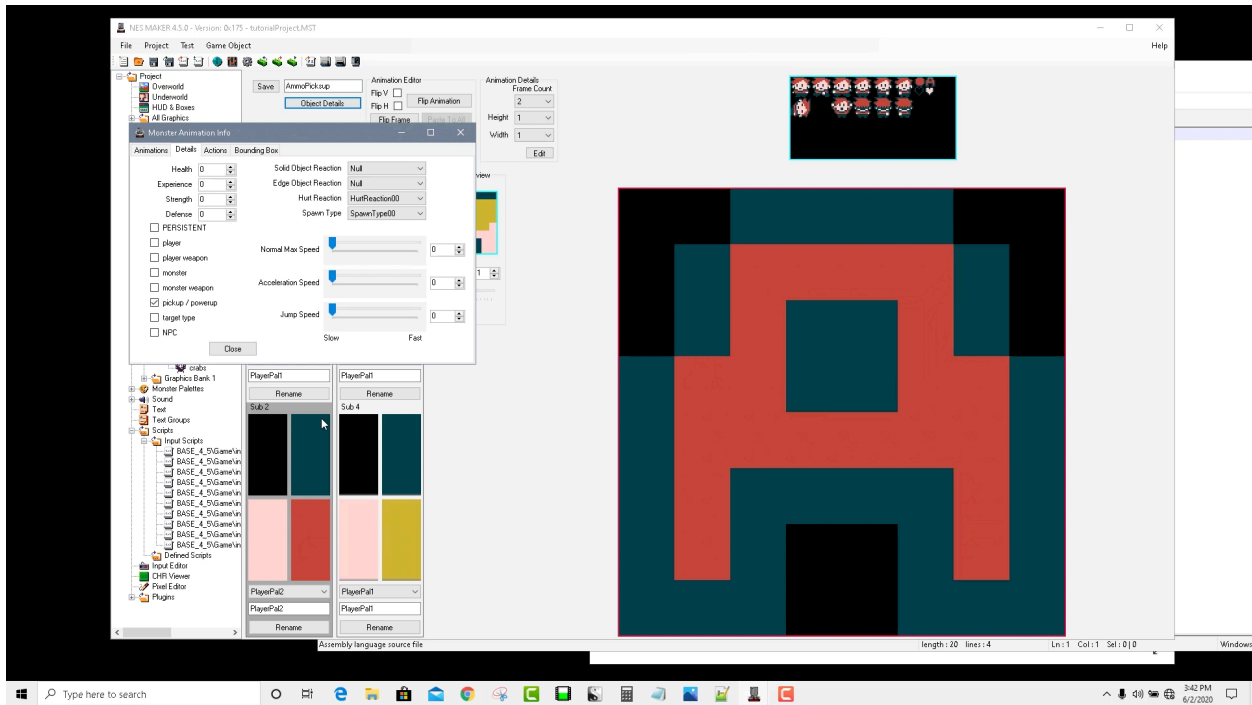
Creating Pickup Object

In our game, we can run out of ammo quite quickly. We definitely need a way to replenish this. In our game, the player will be able to find pickup objects, one of which will do exactly this. Understanding how this pickup object works will open the door to many possibilities involving pickups, power ups, and game objectives.

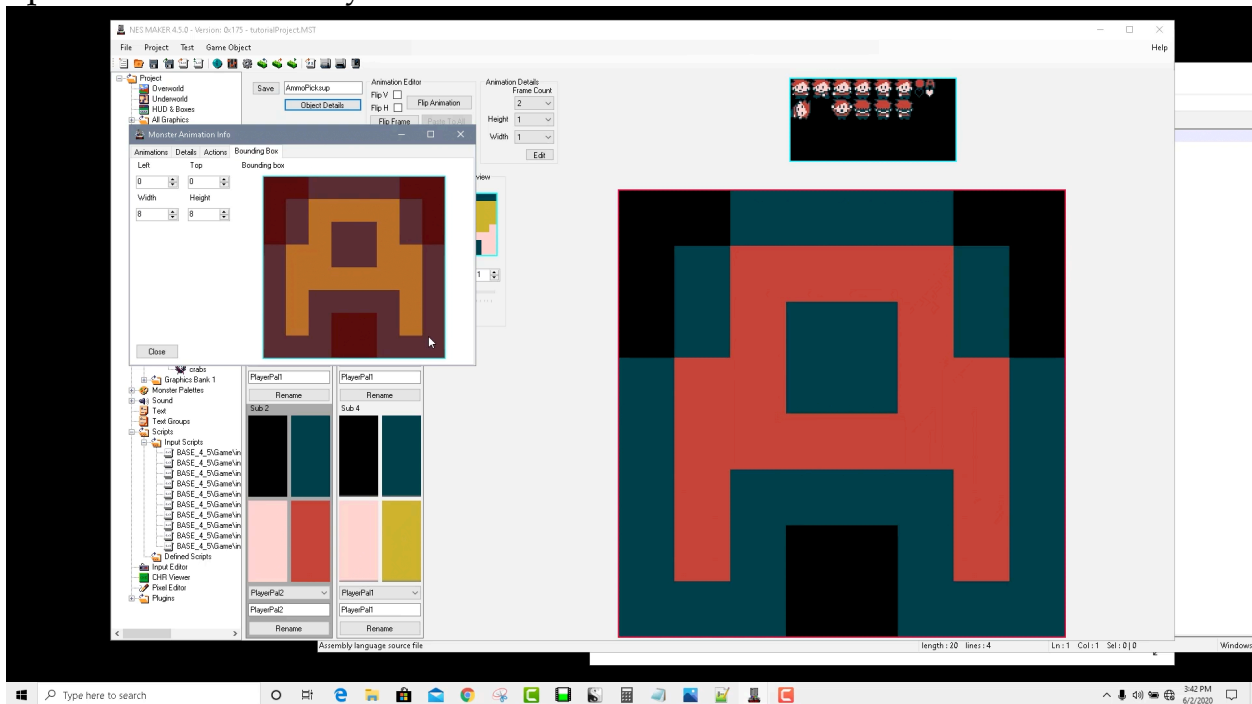
Step 1: Click on Game Object 2, the one that says Projectile Source. As stated a few times, these names are incidental. We'll use this Game Object 2 as our ammo pickup, so rename it AmmoPickup, and make it's 8x8 tile look like the A from your tileset.



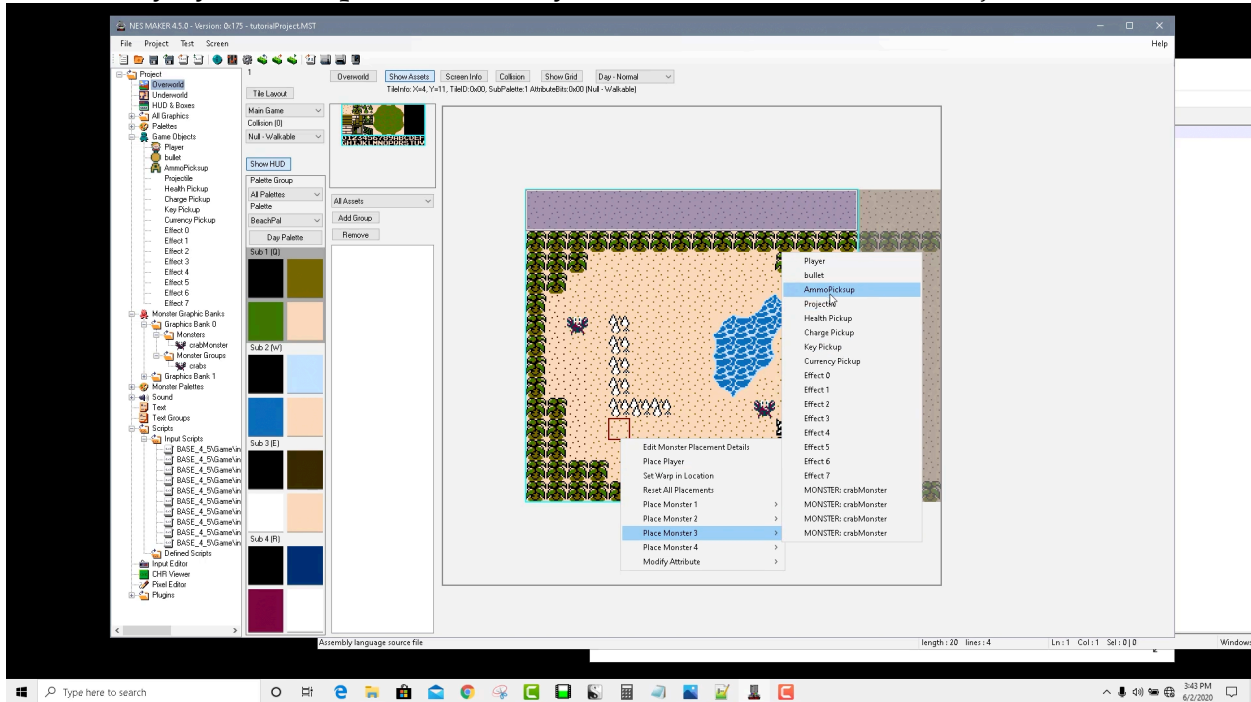
Step 2: In Object Details, make it a pickup / powerup object type.



Step 3: Give it a full bounding box. After this, you can hit close on the bounding box dialog and hit save on the object. You should see it's graphic update in the hierarchy.

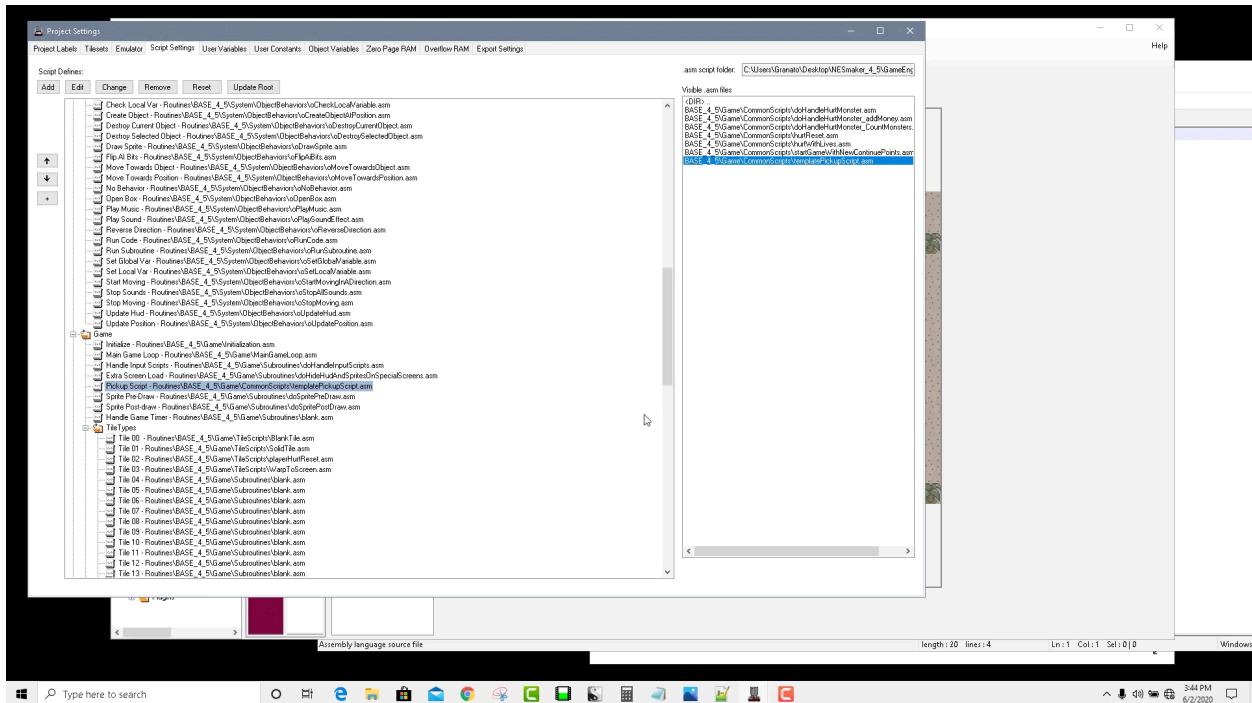


Step 4: Go to one of your game screens. If you've already set the monster group for a screen, it is very easy to add an ammo pickup. To any screen where a monster group is defined, you can place all of your game objects along with your monsters. You can place up to four non-player objects at a time, and there are clever ways you can spawn more if you need more than four objects.



If you test your game, it will half work. It will go away, because in this module's codebase, a player object colliding with a powerup object will cause the powerup object to be destroyed. However, it won't update our ammo, because we have not invoked any script to tell this power up exactly what to do.

Step 5: Go to your Project Settings and your Script Settings tab. Scroll down to Game and click on Pickup Script, which you'll see is currently blank. Use your script finder to navigate to root / game / commonScripts and double click on templatePickupScript.



We're going to look under the hood at the code so that you'll understand better how to easily manipulate this basic pickup script. While PickupScript is selected, hit the Edit button at the top of the defined script list.

This is a pretty simple script, but understanding how to manipulate it will open up a lot of possibilities.

Line 9 says LDA Object_type,x. This line is loading the type of object that you are colliding with to determine what to do next.

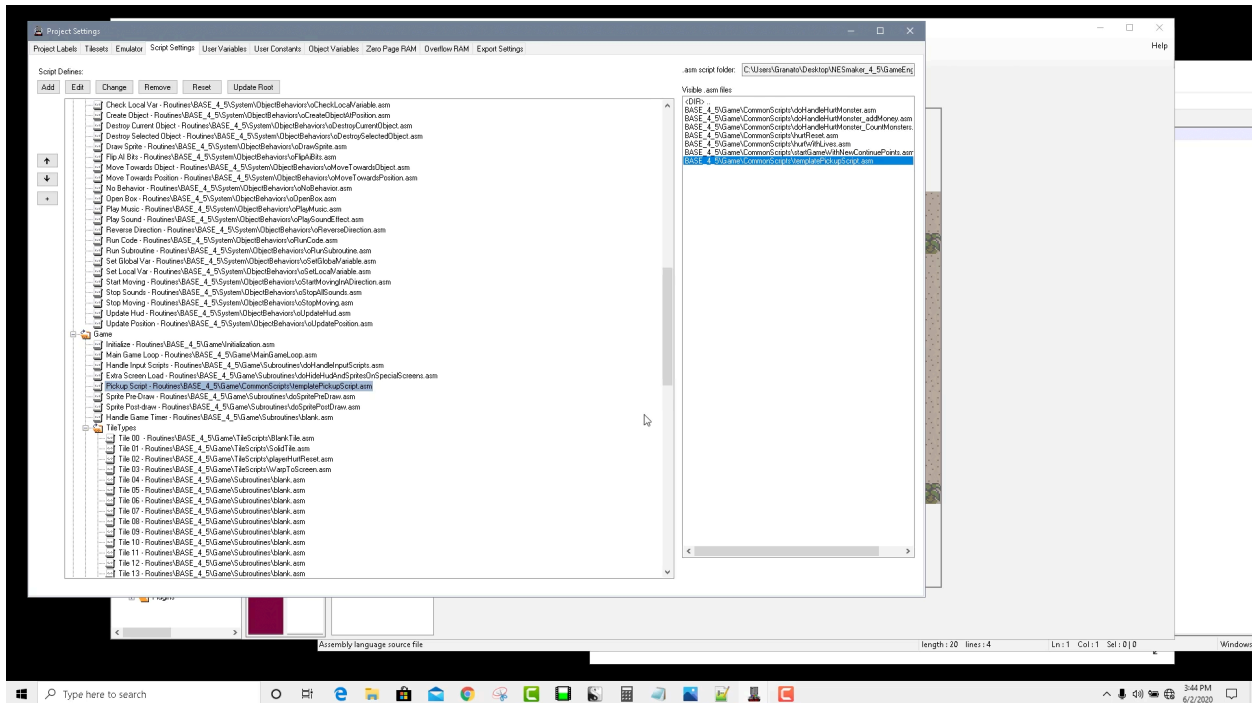
Line 10 says CMP #\$02. That is determine if it is the second game object.

Line 11 says BNE +notThisPickup. That means to branch to the label called +notThisPickup if it is NOT equal to GameObject 2. This means that it will only do the next bit of code if it is equal to two, otherwise it will skip it.

Line 15 says LDA #\$05. This is loading the number five.

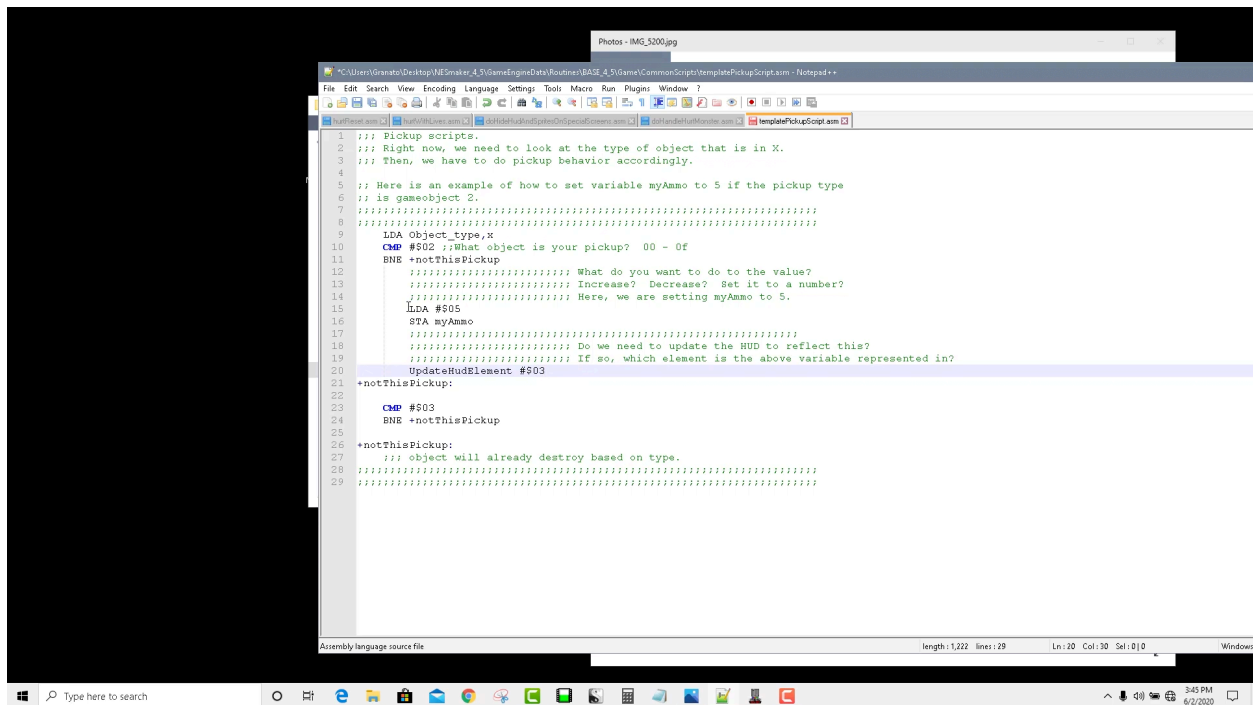
Line 16 says STA myAmmo. This is taking 5 and placing it in the proverbial myAmmo container. Now, myAmmo is 5.

Line 20 says UpdateHudElement #\$07. This will update the graphics for HUD Element 7.



Line 20 might prove problematic for our purposes, though. Our myAmmo variable was not in HUD element 7. Nothing was in HUD element 7. Our myAmmo variable is in HUD element 3. Hud element 0 draws the word LIVES. Hud element 1 draws the word AMMO. Hud element 2 draws the value of the variable myLives. Hud element 3 draws the value of the variable myAmmo. So if we leave this exactly as is, the variable myAmmo will change and we'll be able to shoot again, but we will not see the change reflected in the HUD, because it's updating the wrong element.

Change the 7 to a 3 and save the file.



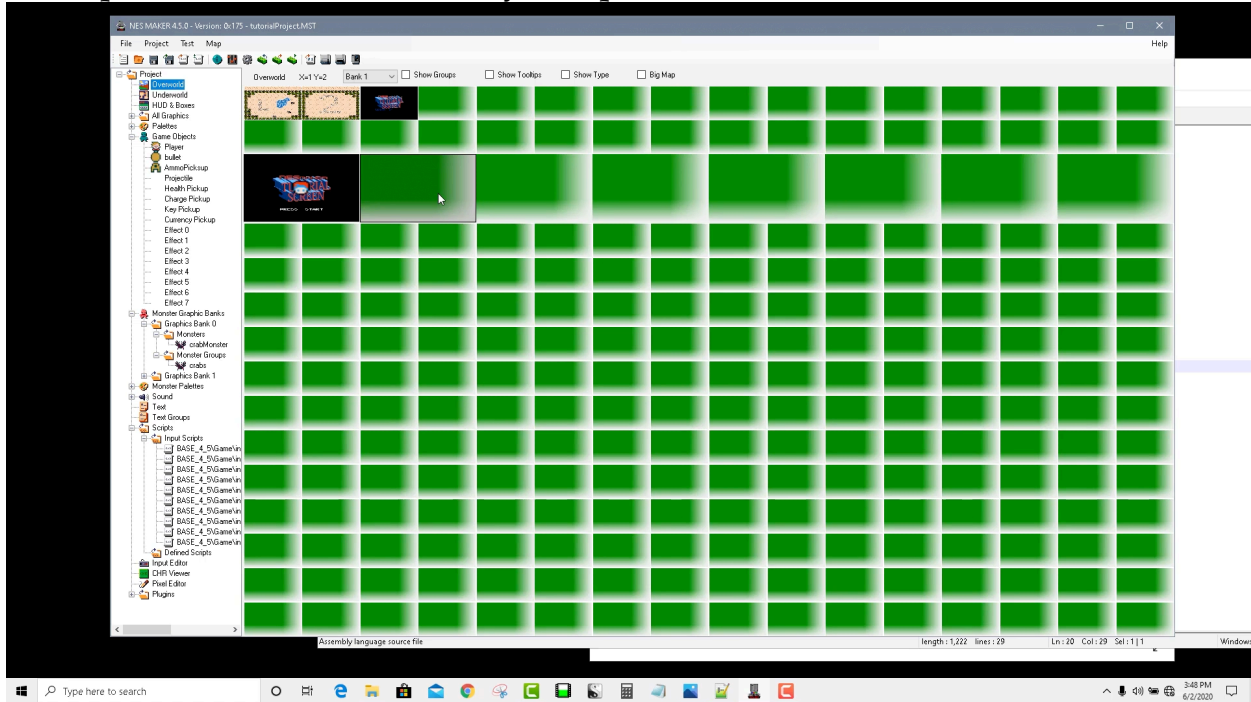
Now, test your game. When you run into ammo, it will destroy the ammo pickup object, and it will run the default pickup script. In the default pickup script it will act based on what game object is involved in the collision. In our case, it's game object 2, which our script says should set the myAmmo variable to 5 and update the hud element 3, which is the one that happens to show the myAmmo variable in the hud.

Win Screen

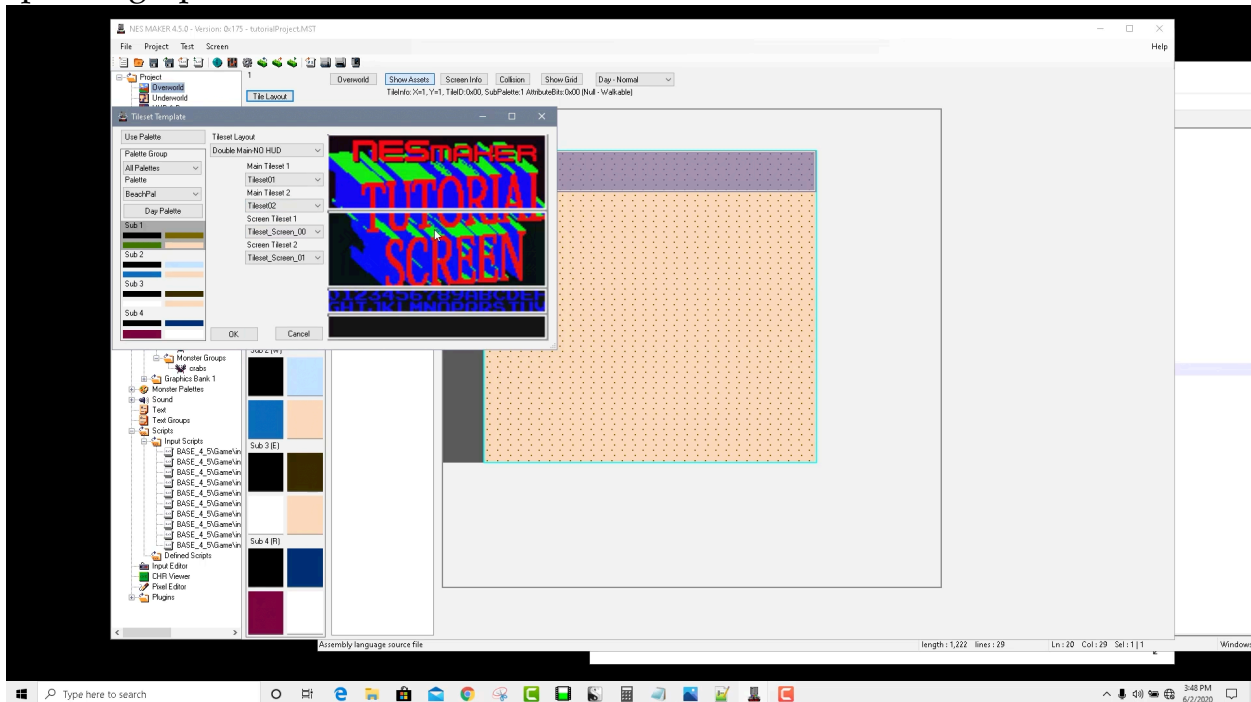
Creating a Win Screen

We could get very ambitious with our win screen, but for the purposes of this basic instructional, we'll keep it nice and simple. We already have a game state reserved for win. We also already have enough graphics established that we can crank out a quick screen that says CONGRATS.

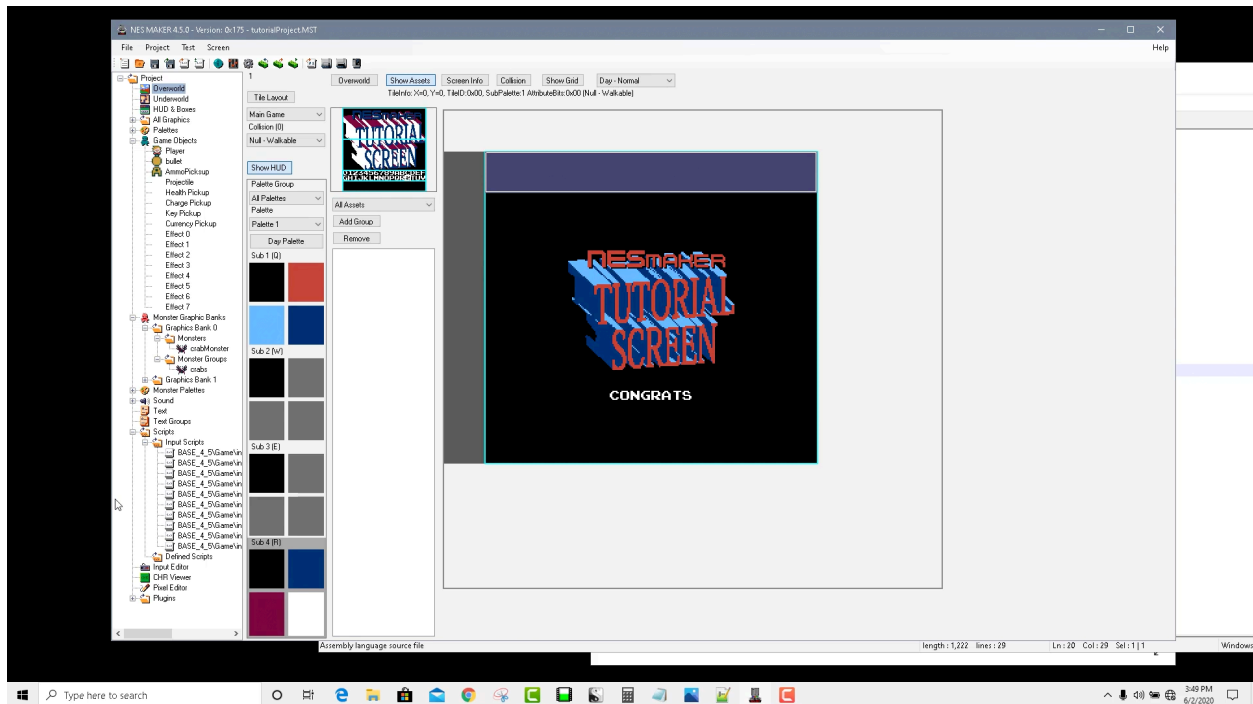
Step 1: Make a new screen in your special screen area.



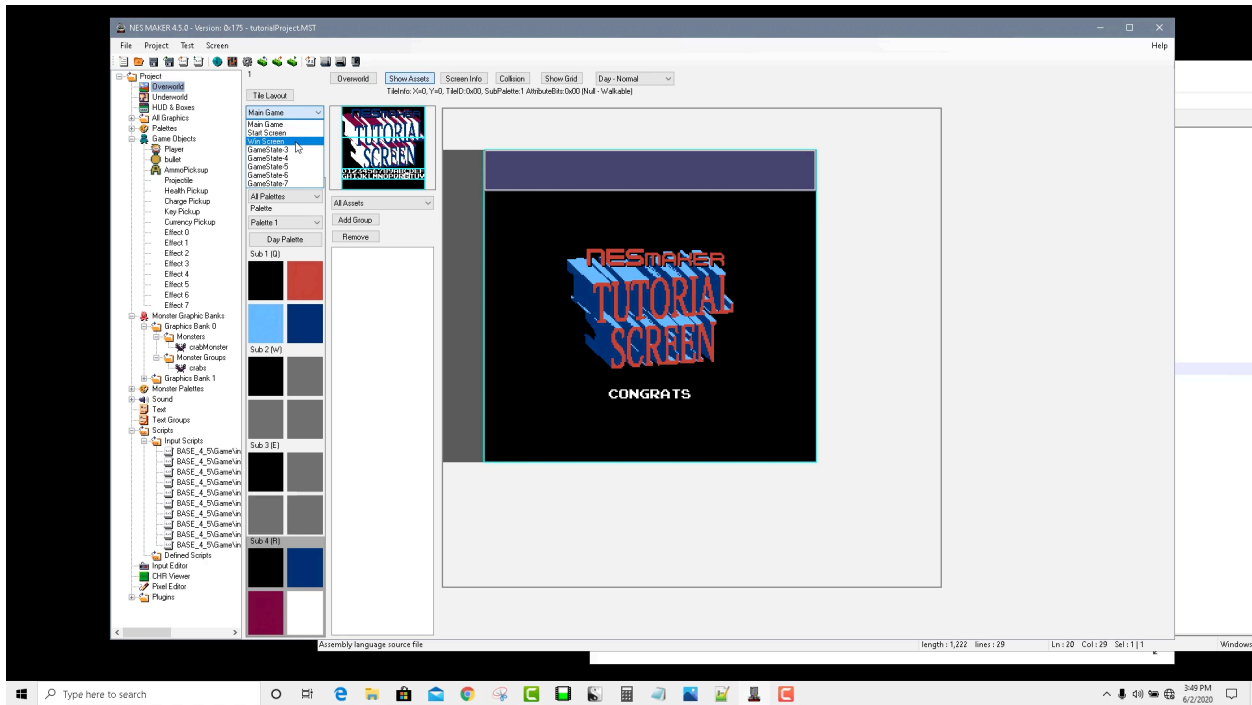
Step 2: Set the tile layout up the same way as we did with the start screen - double main no hud with the tutorial graphics and letters and numbers screen specific graphics.



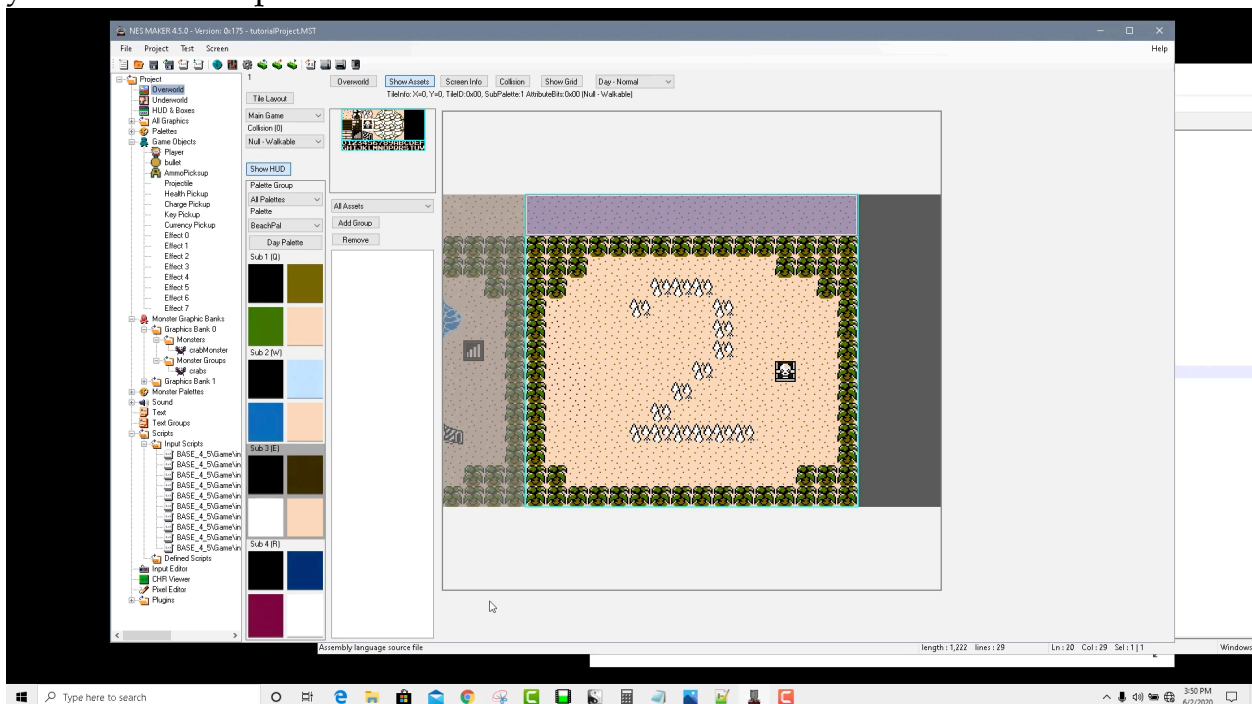
Step 3: Select the start screen palette that we created. Then paint the screen with our tiles. Use shift-click-drag to select the part of the tileset you want then place it into the screen. Choose the last sub palette to add a white CONGRATS to the button of the screen.



Step 4: Select Win screen from the Screen Type dropdown. This is not a main game or start screen type. It is a new type of screen, which means currently none of our inputs will work on this screen, and we could set up different inputs that would only affect this screen.

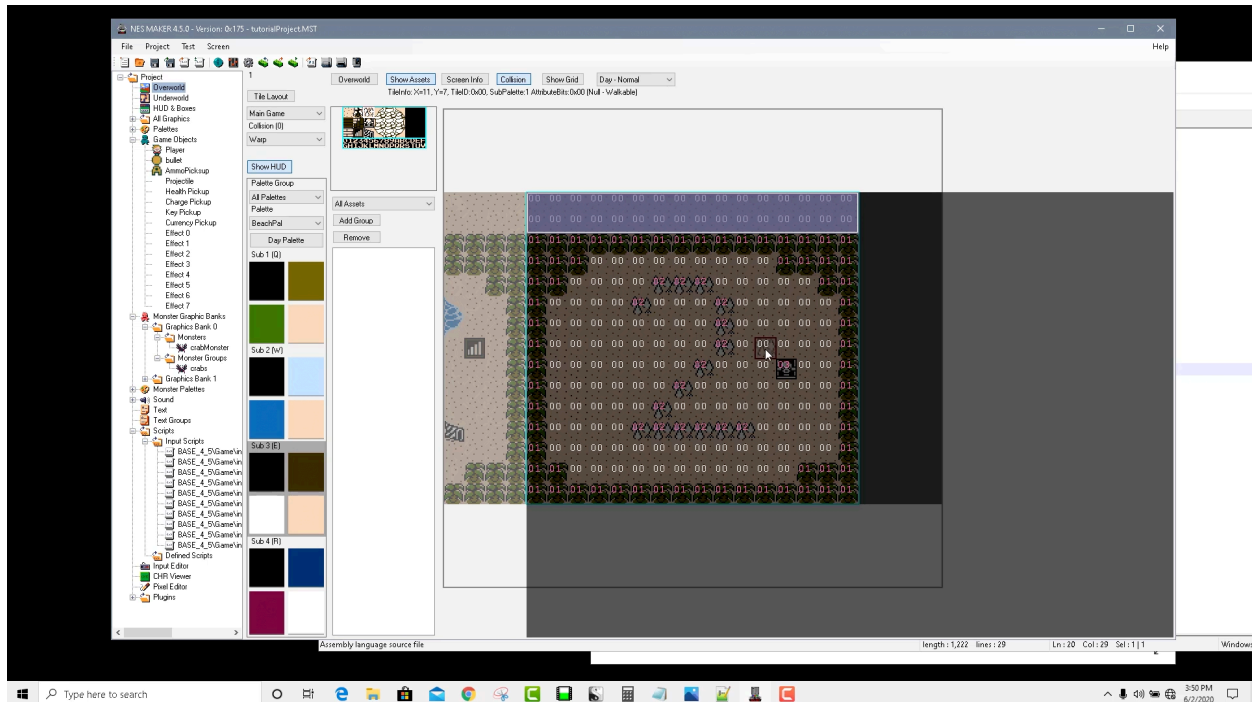


Step 5: Go to your second game screen. Choose the treasure chest tile from your tileset and place it somewhere in the room.

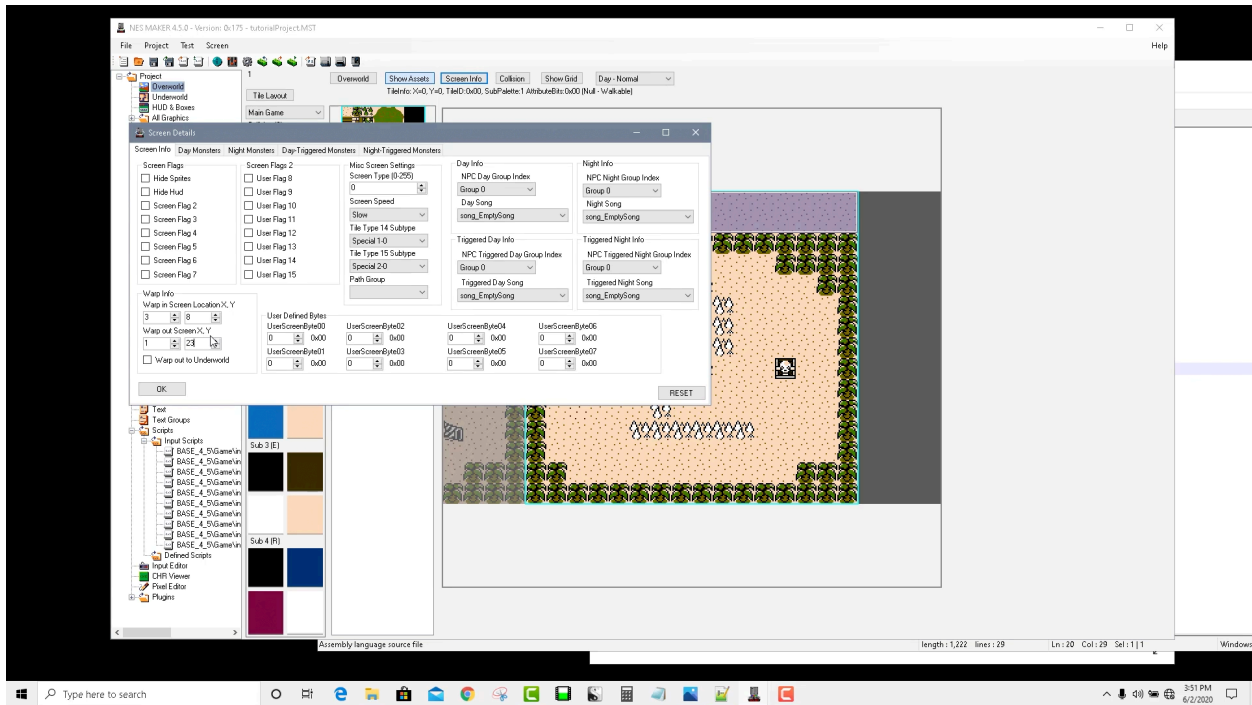


Step 6: Turn on the collision overlay. If you haven't already made the spikes into hurt tiles, select Hurt from the collision drop down and use the zero key to

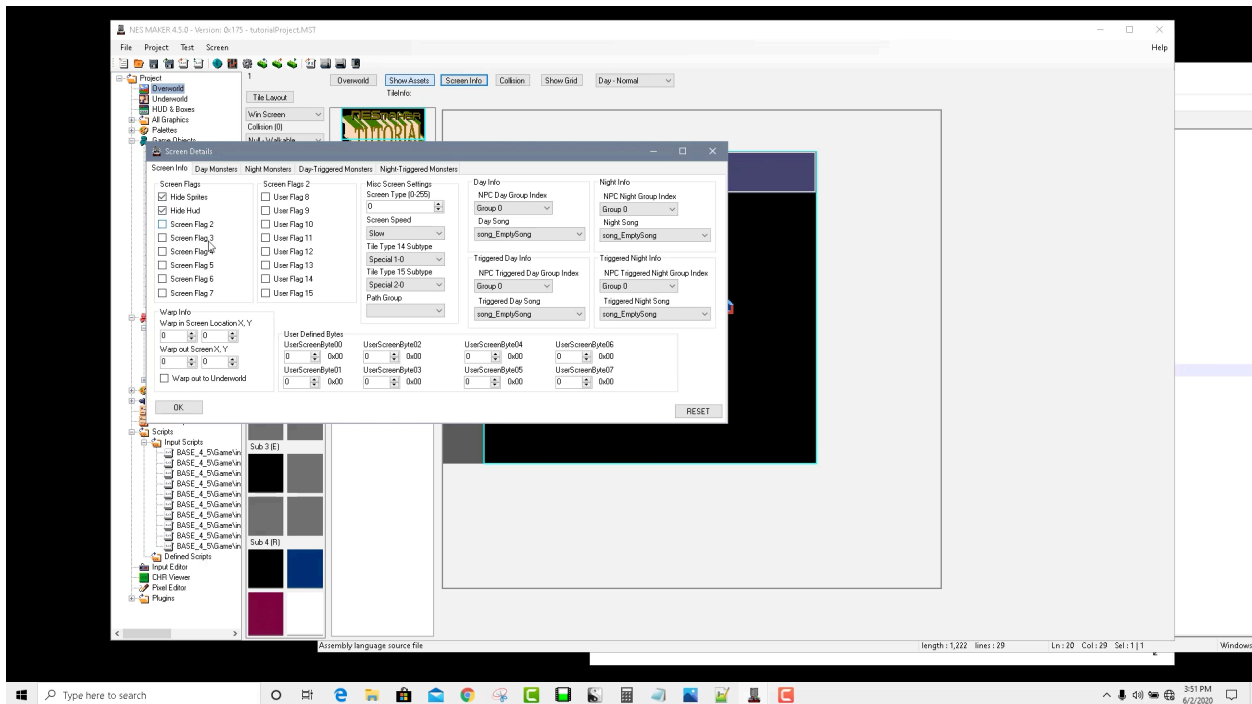
paint them. Then, choose the warp tile from the drop down, and use the zero key to paint the treasure chest with a warp tile type.



Step 7: If you've designed your map just like mine, the screen we need to warp to is in column 1, row 2 (keep in mind, the first row for each is zero). To see this, you can go to your map screen and hover over a screen and it will tell you its coordinates in the top left of the map. We need to tell this screen that upon hitting a warp, it will warp out to screen 1,2. In the screen info, type 1,2 in the Warp Out Screen X,Y boxes.



Step 8: On your CONGRATS win screen, open up screen info and choose Hide Sprites and Hide HUD, just like on the start screen.



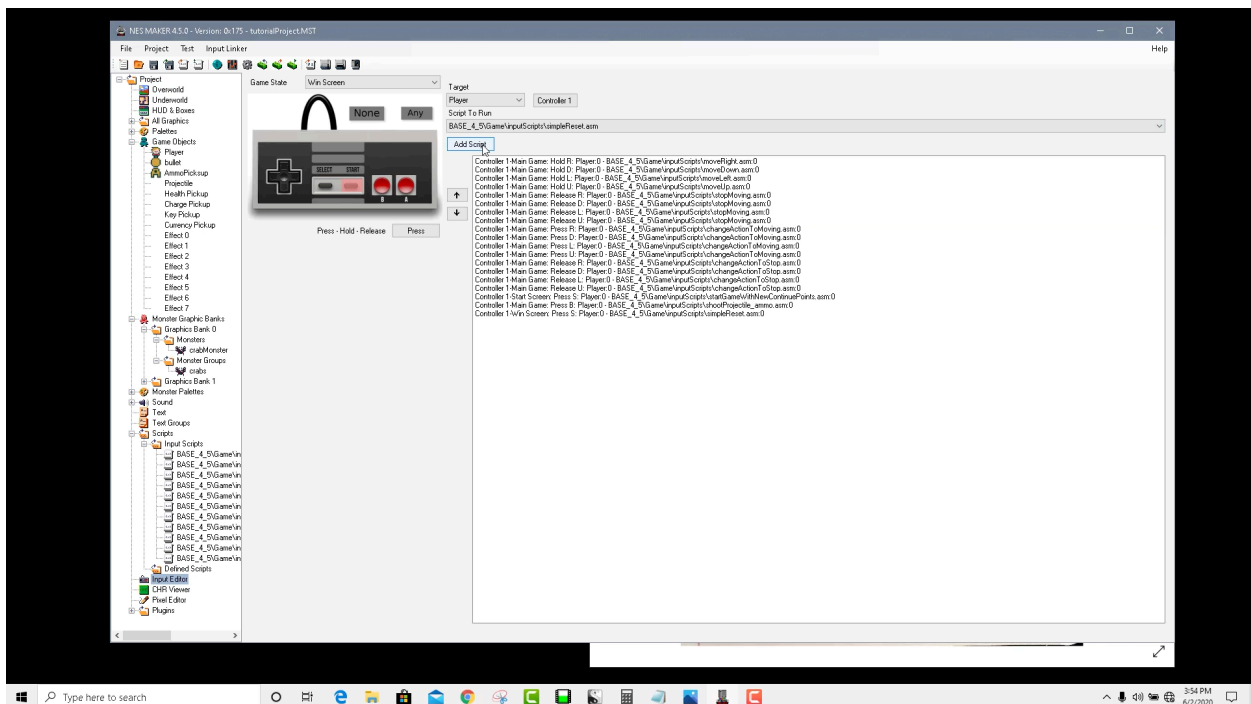
If you test your game, you should now be able to play all the way through to the

win screen. Dodge or shoot the crabs, hit the staircase warp to the second screen, grab the treasure, and it should take you to the win screen.

There is no way to get back out of this screen, though. But setting up a reset upon pressing the start button is very simple.

Step 9: Click on your input scripts node. Navigate to root / game / inputScripts and double click on the script called SimpleReset. This will add a simple reset script that will be exposed to your input editor.

Step 10: In your input editor, make sure that the Game State is set to WIN SCREEN. Click on the Start button. Make sure the button state says PRESS. From the script drop down, choose SimpleReset, and hit Add Script.



Test your game. You should now be able to get to the end of your game and reset by pressing the start button.

Adding Music and Sound

Add Music And Sound Effects

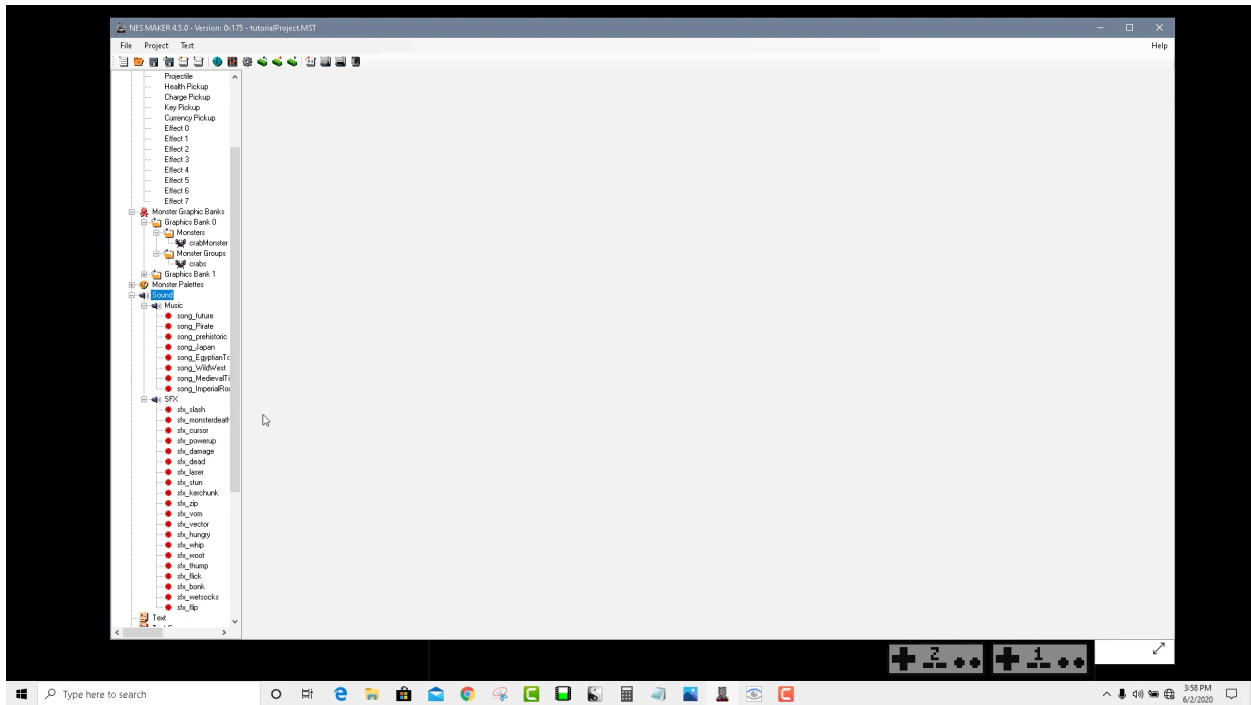
The game is fun, but what really begins to make it feel like a polished game are the details like the music and sound effects. We're not going to into music creation in this tutorial, but we will show how to import files made with the NES music program FamiTracker, how to attach songs to screens, and how to trigger sound effects where we may want them triggered.

Songs

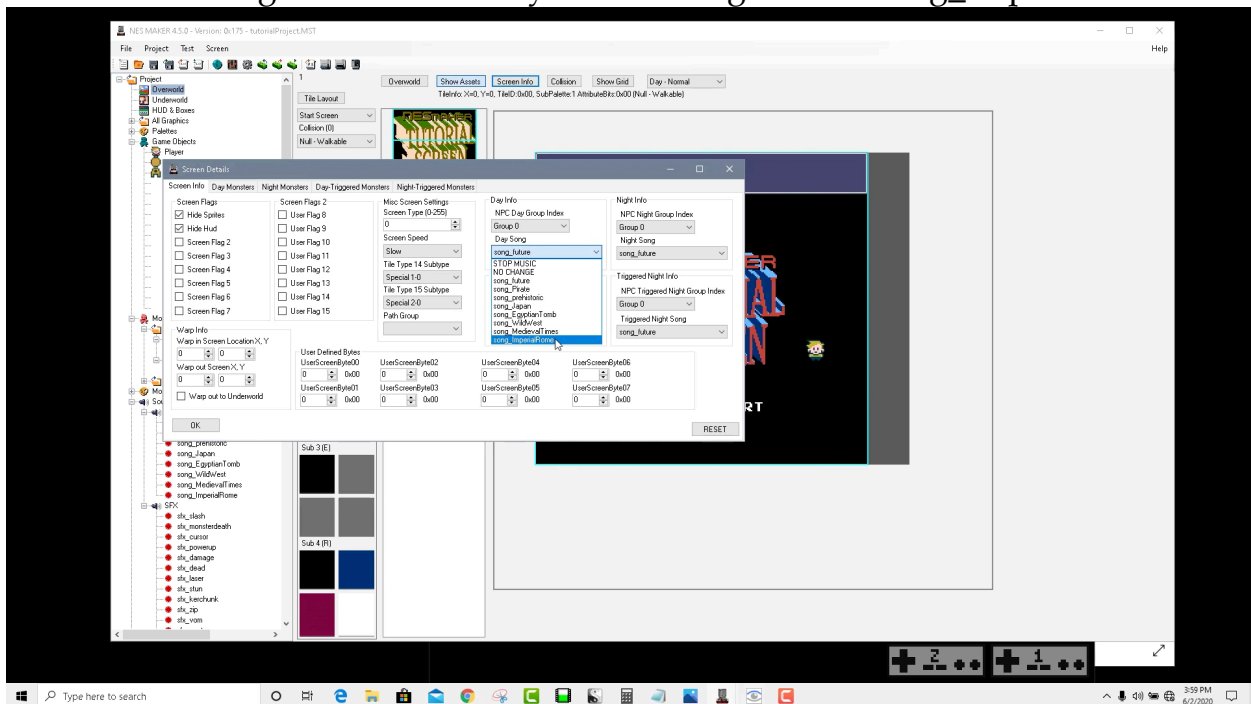
Step 1:

Right click on the sound node and click on Import Famitracker txt

Step 2: Go to root / TutorialAssets / 4_5_x_TutorialAssets / Sound. Double click on the Tutorial_4_5_SoundPack to add the tutorial music and sounds to your game. That will load a bunch of songs and a bunch of sound effects to your project.



Step 3: Go to your overworld and double click on your start screen. Then, click on Screen Info. This will bring up the dialog about this screen. One of the components here is song for the screen in different states. The only state we're worried about right now is the Day state. Change this to song_ImperialRome.



Step 4: Do the same for your other game screens, but use `song_PirateTheme` for all of the main game screens and `song_WildWest` for the win screen.

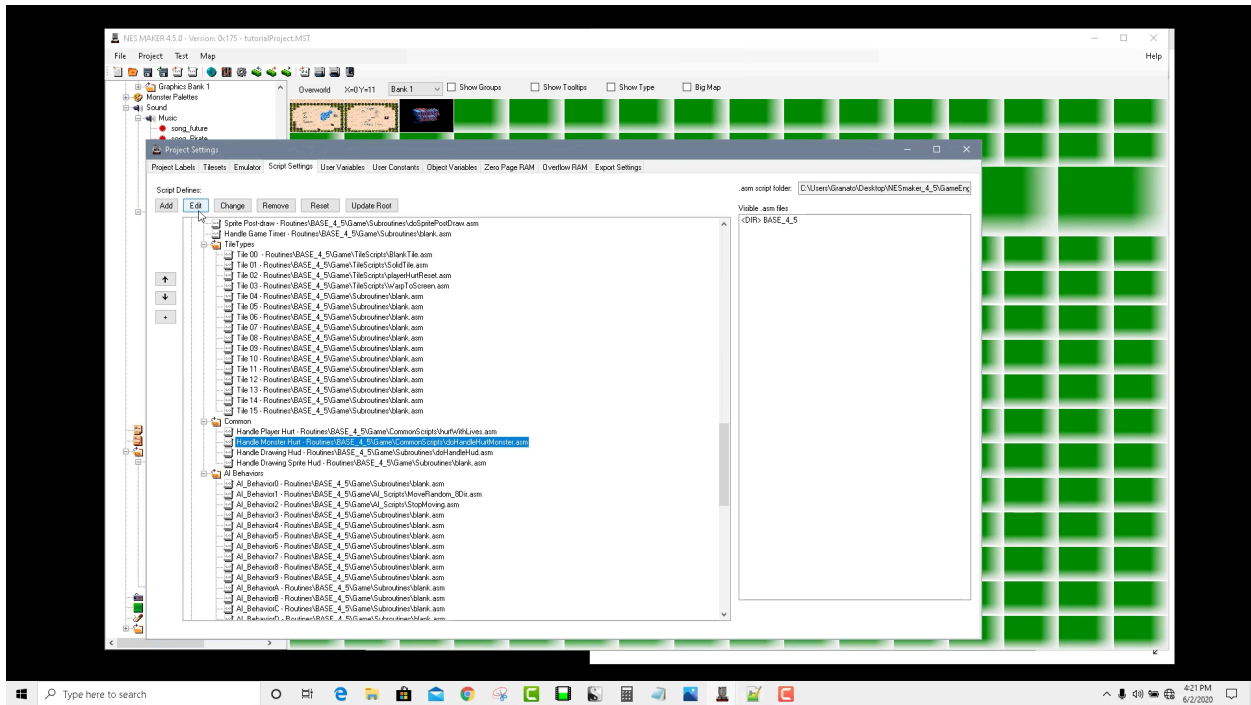
Test your game and your screens will now play the corresponding music.

Sound Effects

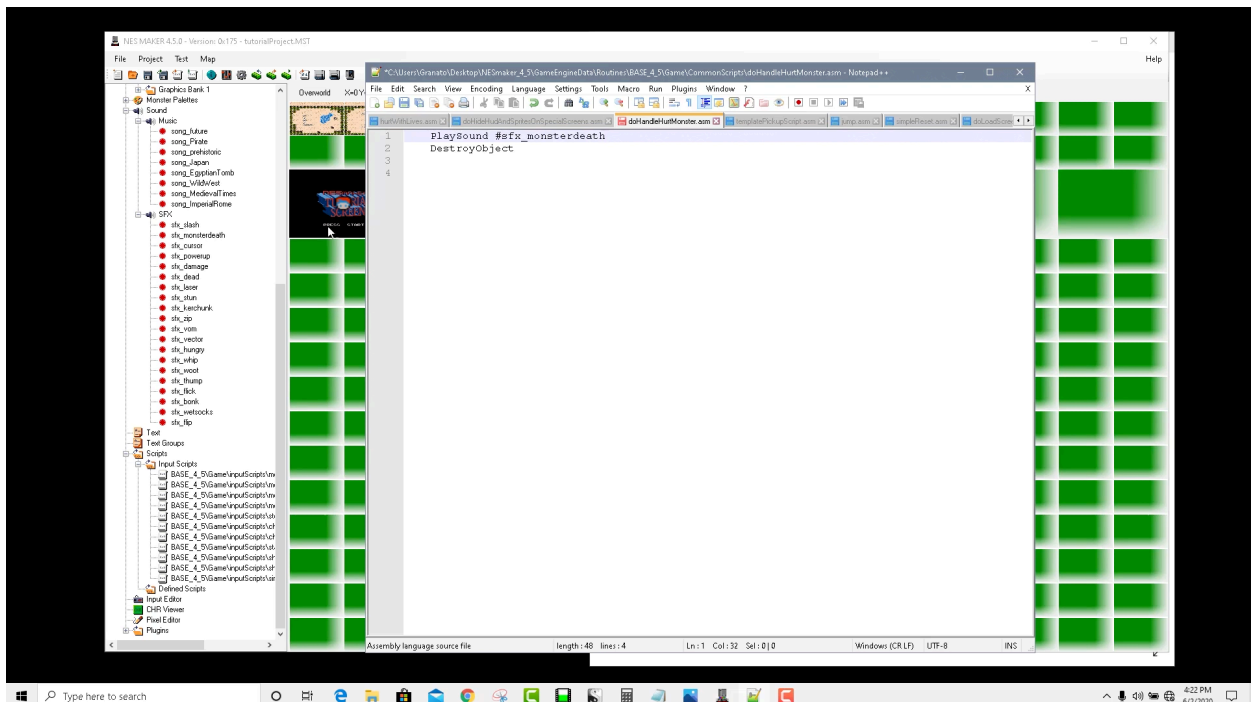
Sound effects are slightly trickier, because exactly what triggers sound effects may differ from game to game. We're going to add one or two in appropriate spots in the code, and you should be able to figure out how to go off on your own from there.

For this tutorial, we are going to add a sound for when the monster is killed, when you shoot your projectile, and for when you get your ammo power up. We need to think out where those things happen in our script settings.

Step 1: In Project Settings, in Script Settings, scroll down to Handle Monster Hurt in your common folder and select it. Then, at the top of the window, click on the Edit button.

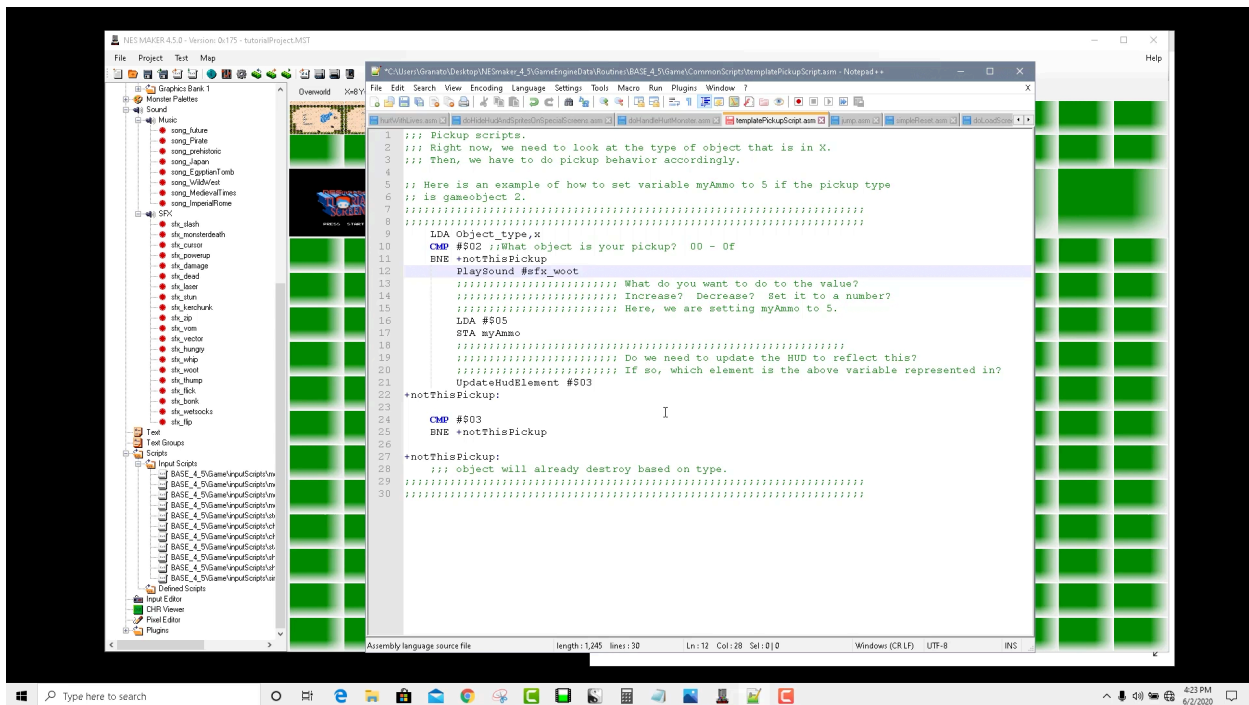


Step 2: Right now, the script has a simple macro function to destroy the object. That's all it does. To add the sound effect, simply add a new line before or after that and write PlaySound followed by the sound you want to play. We're going to add PlaySound #sfx_monsterdeath. Once you've done that, save the file.



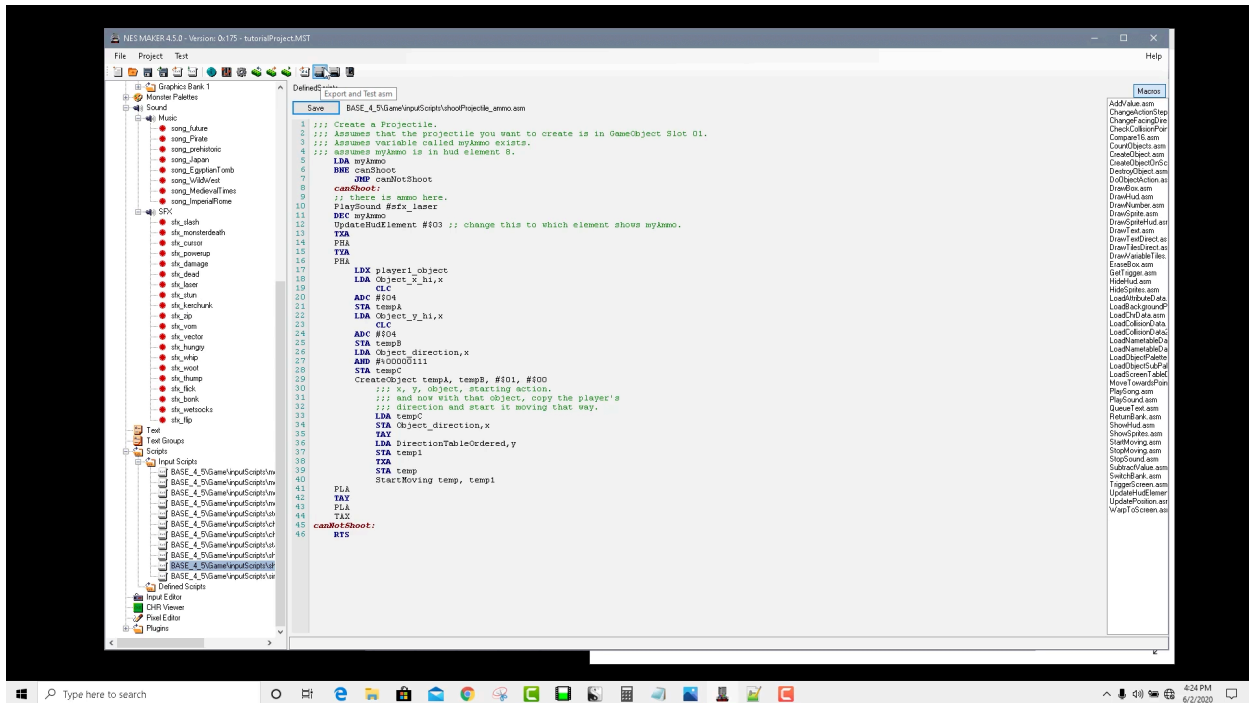
Now, if you test your game, it will make a sound when the monster dies. This one is a fairly quiet sound effect compared to the music, but if you listen carefully you will hear it.

Step 3: Next, we want to add a sound when we collect the ammo. This can be found in Script Settings by scrolling down to game, clicking on Pickup Script, and pressing edit. Somewhere in the code block of things that happen if the pickup object is game object 2, add the code `PlaySound #sfx_woot`. Make sure to save the file.



Test your game and you should hear a sound effect when you pickup more ammo.

Step 4: Lastly, let's add a laser sound when we shoot. For that, we'll go to our input scripts and click on shootProjectile_ammo. As soon as we center the canShoot portion of the script, `PlaySound #sfx_laser`. Make sure to save the file.



Now if you test the game, it will play a sound when you shoot, play a sound when you get your ammo, and play a sound when you defeat a monster.

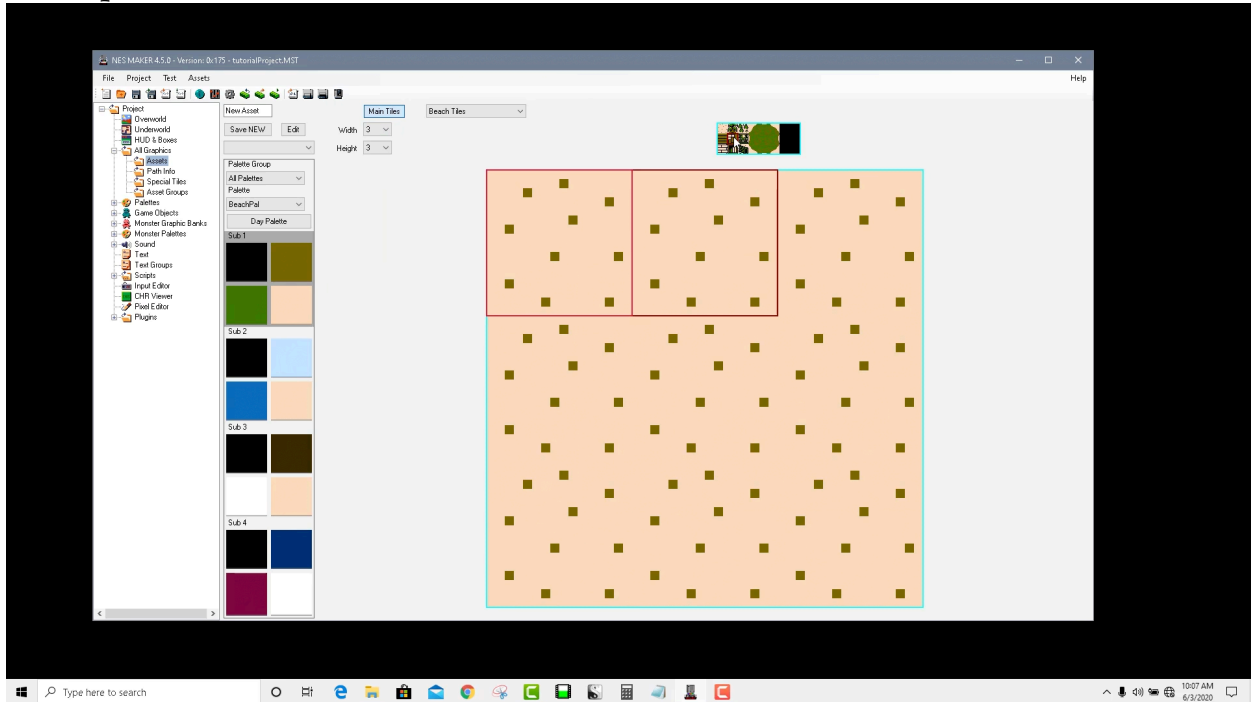
Working with Assets

Screen Painting with Assets

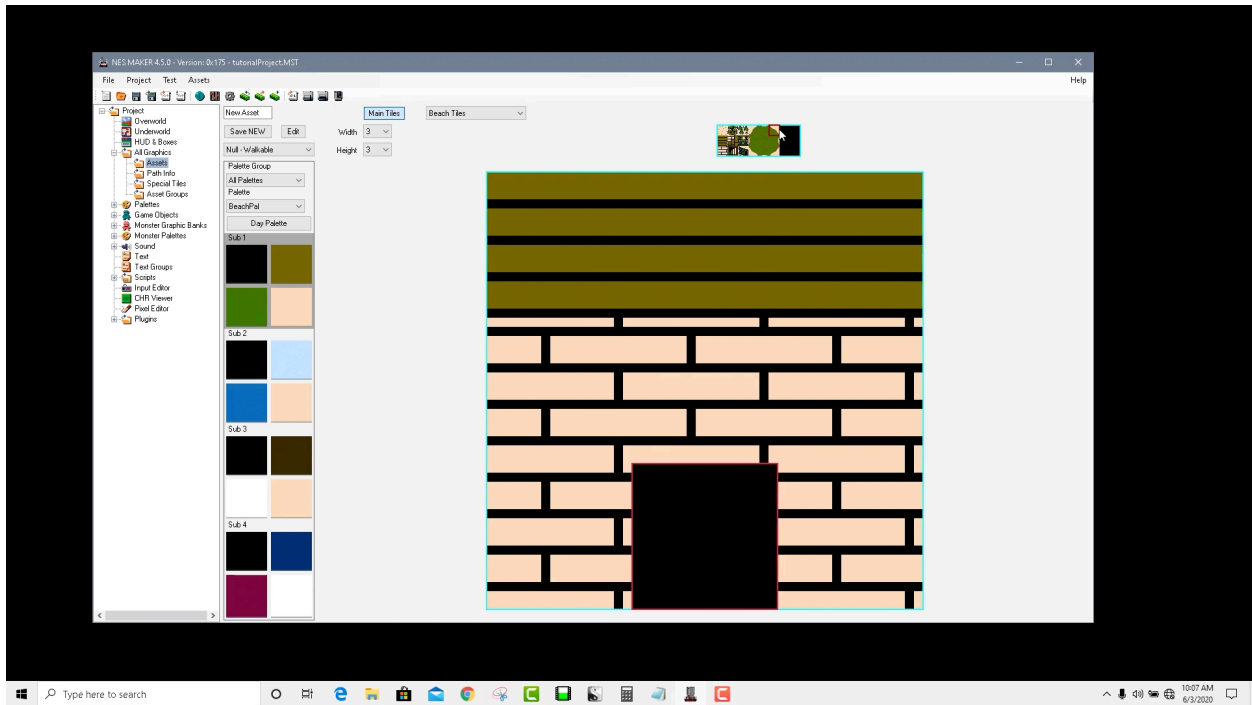
The original way to paint to the screen was with an asset system like what is in this section. It is still a way that many people prefer to work. In this method, rather than grabbing tiles to paint to the screen then grabbing collisions to paint over the top, you create assets which combine color data, tile data, and collision data. You can then pick them from a menu and stamp all the data to the screen at once. Both ways have their merit and different users work differently.

Step 1: Open the all graphics node and click on Assets. What you will see is tileset 0. You can always use the drop down to change tilesets. You can also toggle to see screen specific tileset if you need.

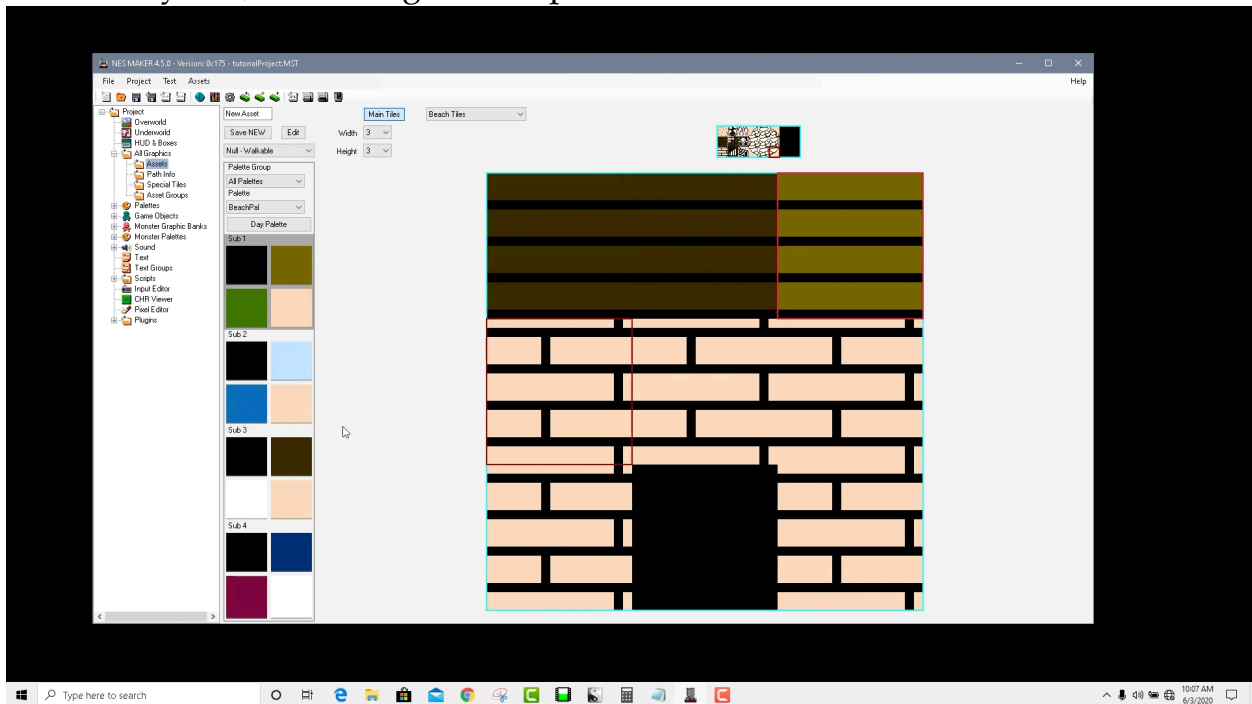
Step 2: Make a 3x3 tile canvas area.



Step 3: Select a tile to change, then click in the tileset to change it. Make a little house, with a roof, brick walls, and a black door.



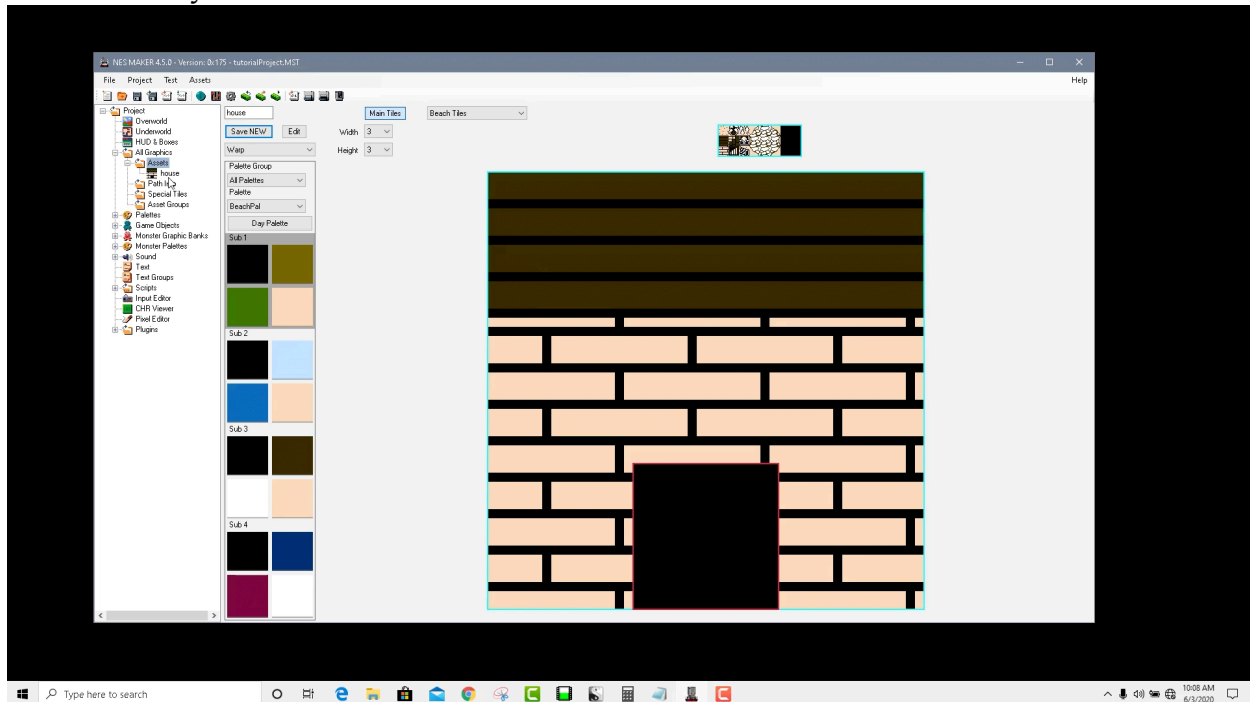
Step 4: We can also change the sub palette of individual tiles. Click on the top tiles one by one, and change to sub palette 3 to make a darker roof.



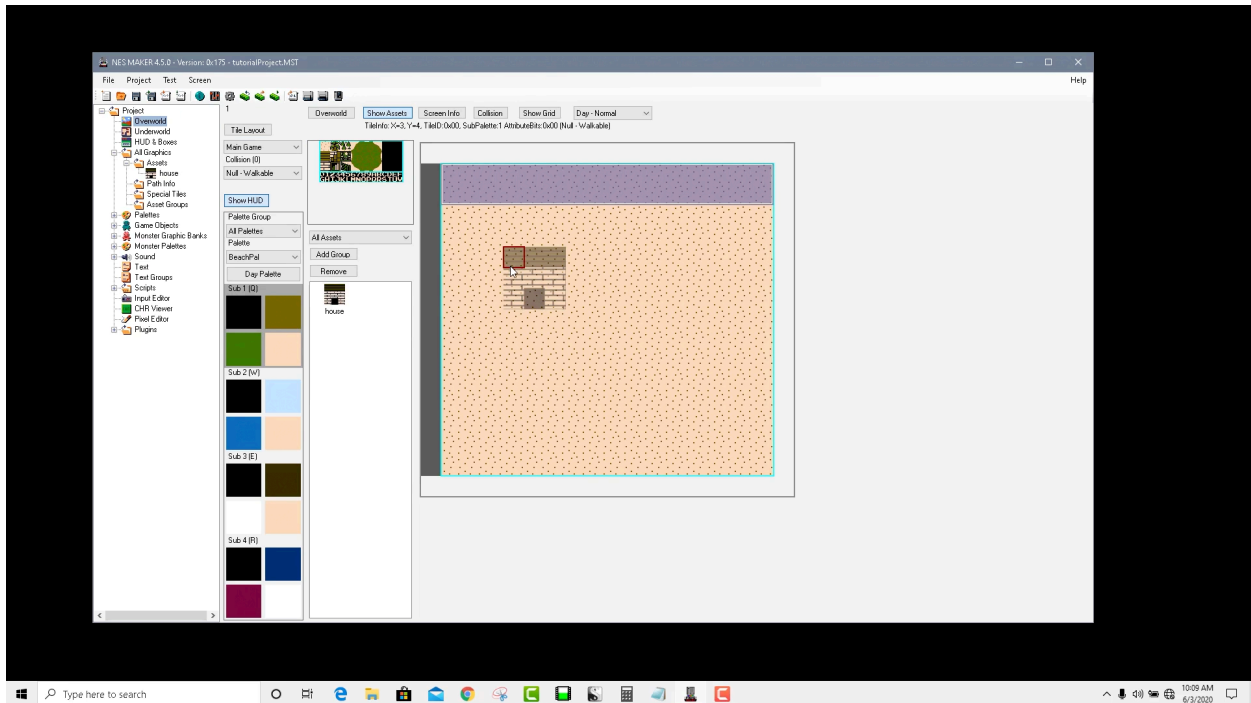
Step 5: Click on each tile. For all of them, click on the tile then choose SOLID from the collision dropdown list. Click on the door and choose WARP from the

dropdown list. We are mixing collisions in one asset. Now we'll have a solid house with a door that acts as a warp, all in one placeable asset.

Name it House and click Save New. You'll see it appear in your Asset list in the hierarchy.



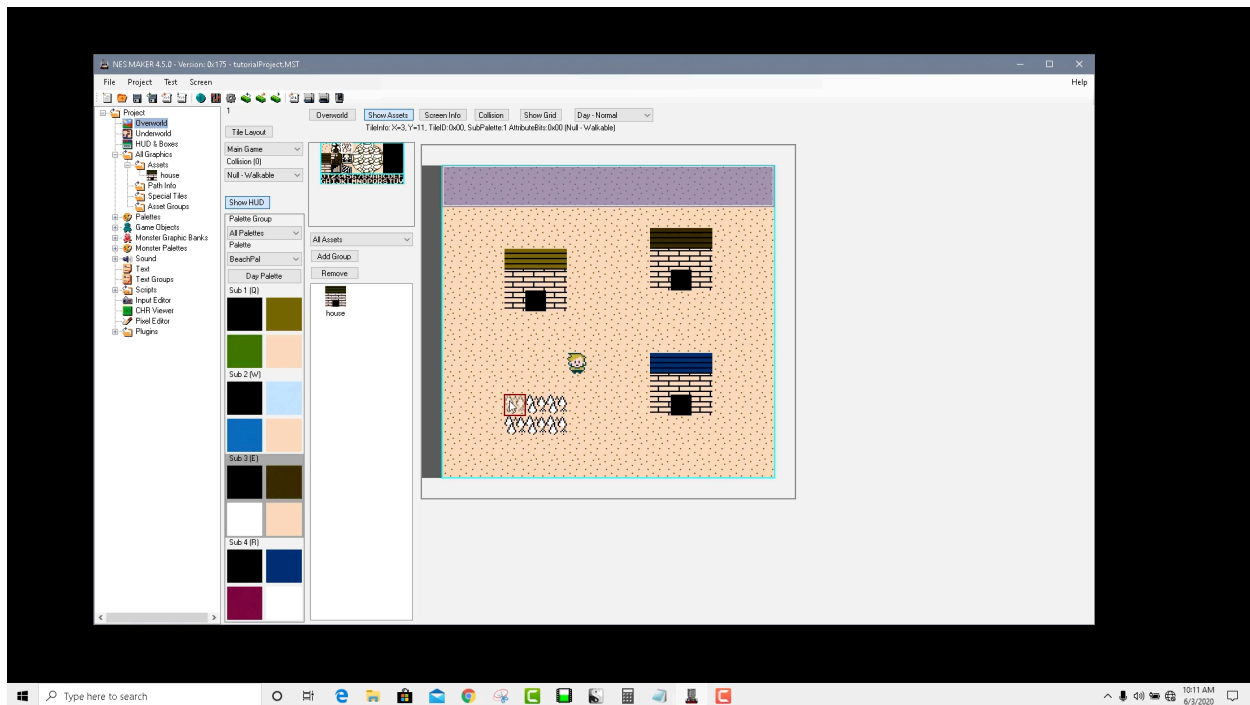
Step 6: Click on a screen - either one of the ones you already created or just click on a new one. If it has your beach tileset loaded, then you'll see the house asset in the list of assets. When you click on it, your cursor will turn into a stamp tool, and you can click anywhere on the screen painter canvas to place the asset. All of its tiles, collision data, and attributes will be set in place with one click.



Step 7: A few keyboard commands to know for this screen.

- Escape will deselect the asset and remove the ghosted image from your cursor
- Shift will change the handle of the selected tile horizontally to make it easier to place along the right side of the screen.
- Control will change the handle of the selected tile vertically to make it easier to place along the bottom of the screen
- QWER keys - this will paint the sub palettes over the top of placed tiles. If you have the tiles you like but want to quickly change the color, mouse over the tile whose color you want to change and hit the Q to change to subpalette 1, W to change to sub palette 2, E to change to sub palette 3 and R to change to sub palette 4.

Using a combination of the assets and QWER keys makes it very easy to place these assets with proper collision data and quickly create variety with multiple sub palettes.



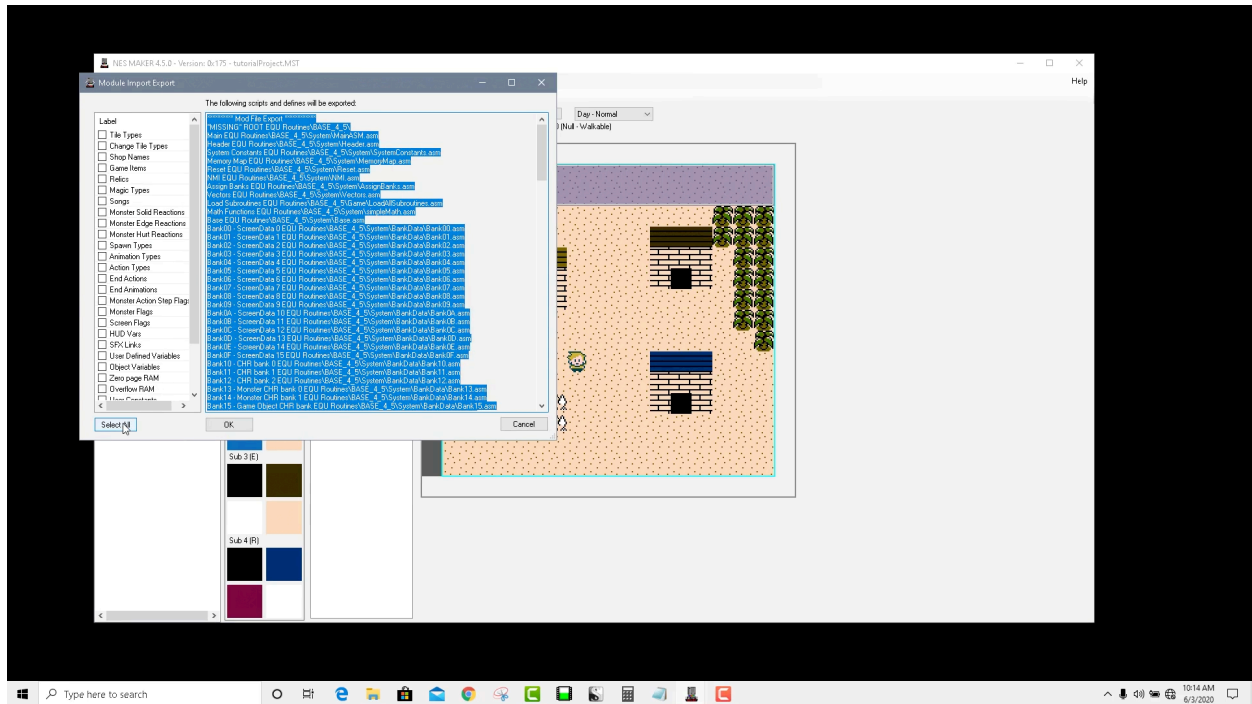
Saving a New Module

Saving a New Module

We started this project with a pretty vanilla module. We then went ahead and added some functionality that gave us the game that we can currently play. Yes, there were very game-specific features that we added such as the graphics and the music, but we also made alterations to the module to make the game behave mechanically in just this way. This mostly involved setting up some labels and defining a bunch of scripts. This is exactly what a module is. We can also save this module so that those things are set up the next time we want to create a game of this type.

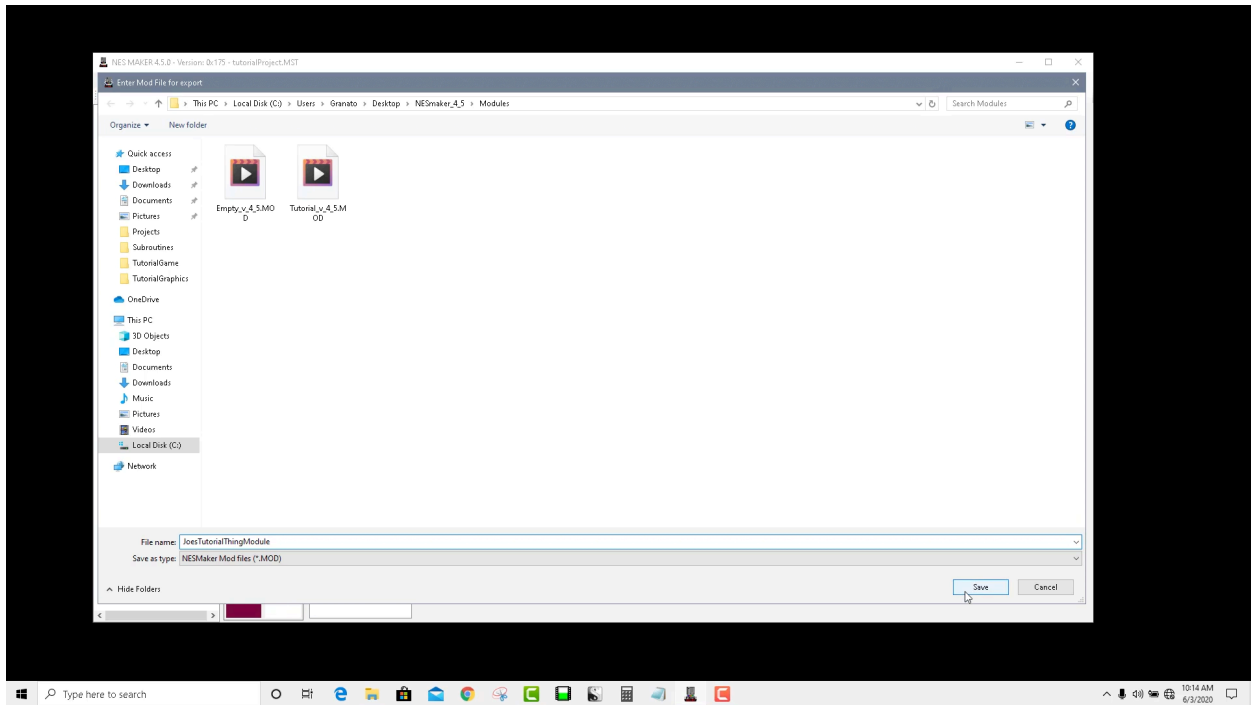
Step 1: At the top of the screen in the menubar, click on Export Project Module. At the bottom of the window, click the Select All button. This will include everything you see selected in the module we're about to build. All of the

labels, all of the variables that we may have added, and all of the script defines that we updated. When you've done this, hit OK.

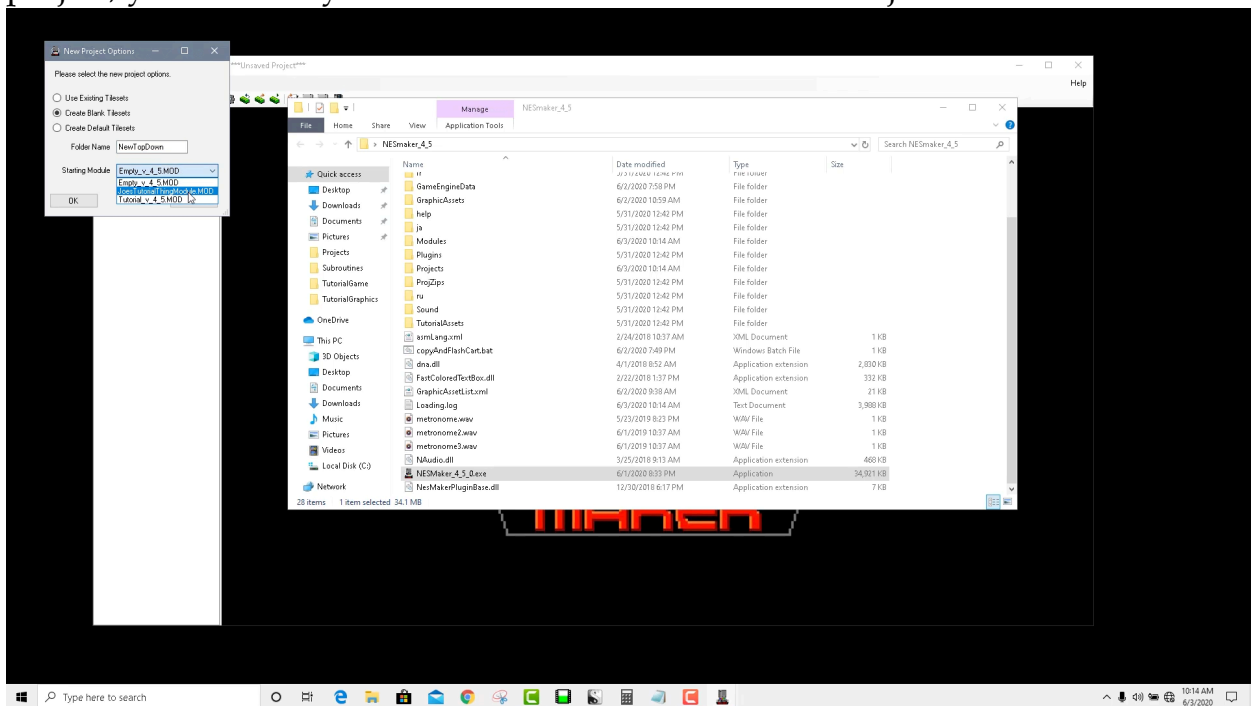


Step 2: It will automatically try to save it into the folder called Modules. When you started the new project and got to choose your module, it was pulling those from this folder. So if you create a new module, it will be available to you upon starting a new project, and it will launch with all of the settings that you are currently saving.

I'll call mine JoesTutorialThingModule and hit save.



If you were to save this project, quit, and re-open NESmaker, then start a new project, you'd see in your list of modules the new one we just created.

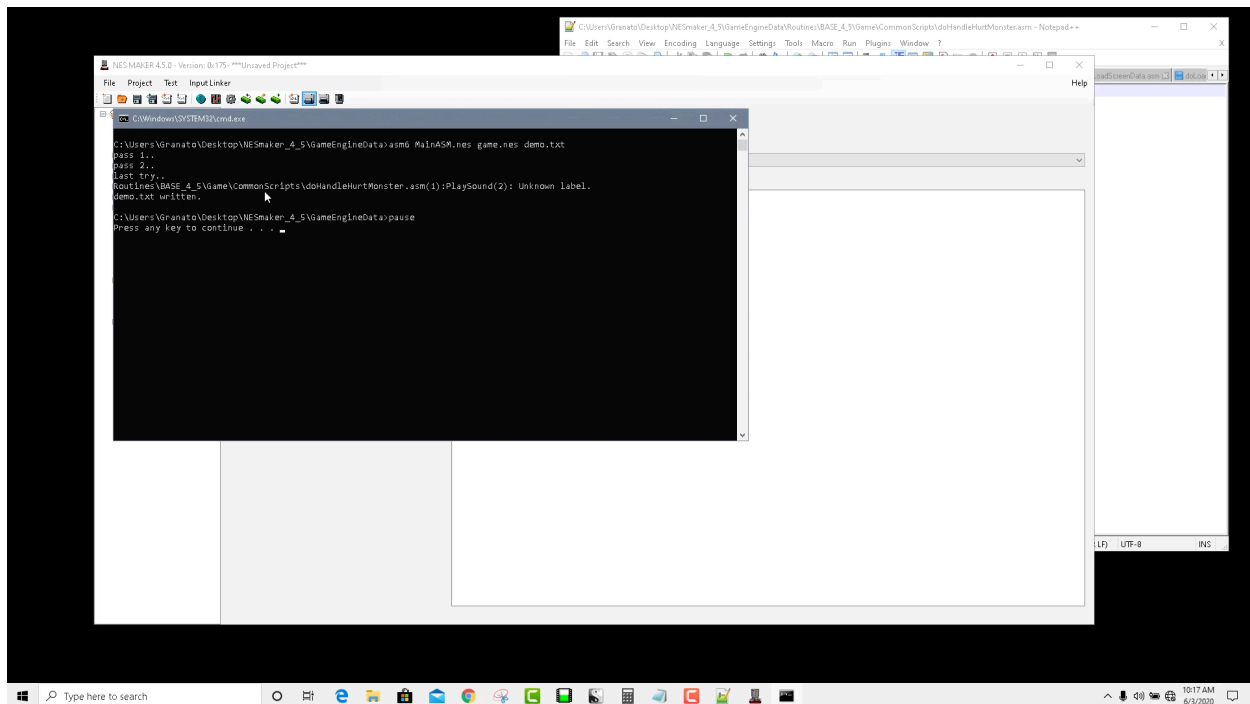


This will not bring in game specific things like the graphics, the sounds, or the screens. But it will set up the script defines, labels, and variables so that all of the

functional things work as they did by the end of this instructional. The AI behaviors you loaded, the tile types you set up, the labels...all of those things will be ready to populate with new game-specific content.

The only problem spots will be, for instance, scripts you added PlaySound to. If you don't have the same sounds loaded to your project, it won't have any reference for those and it will give you an error. But that's easy enough to fix by just deleting those lines of code.

When you try to compile, it will throw an error and not open the game. The error will look like this.



This is telling you that in the doHandleHurMonster script at line 1, there is an unknown label. The labels it means is the #sfx_monsterdeath sound. Since we haven't loaded sounds to this game (or maybe we have, but there is no sound called #sfx_monsterdeath in our custom sound pack or something), we'll get this. All we have to do is go to the script it says and remove the line that was a problem, and we should be able to compile without issue.

Flashing to a Cartridge

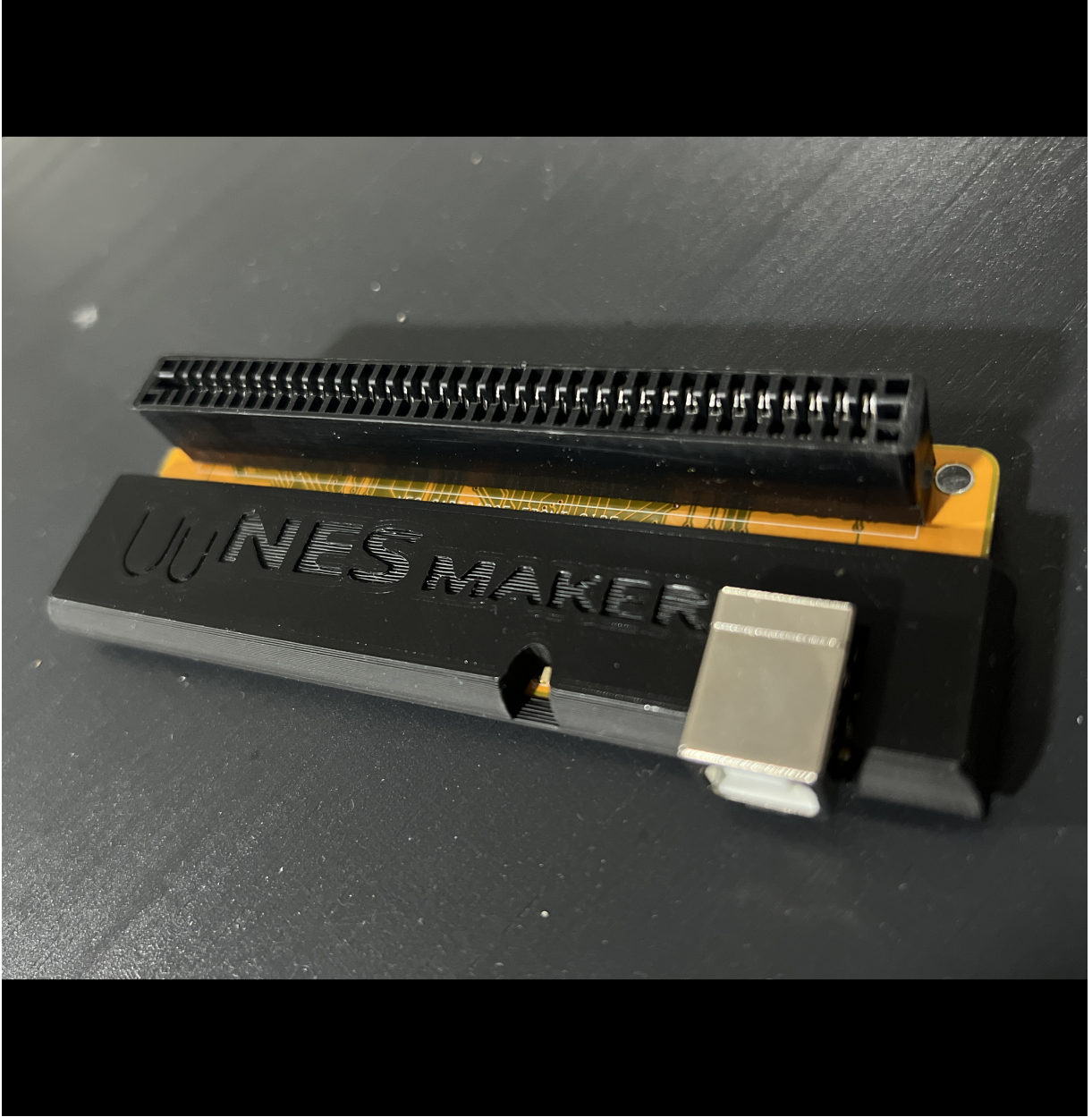
Flashing your Game to a Cartridge

The main reason why most people are interested in NESmaker is not because it has the capacity to make video games. There are a lot of tools available to make video games. There are none at the time of writing this that can flash NES games to a cartridge with the push of a single button. So to end this instructional, let's take a look at how to do that.

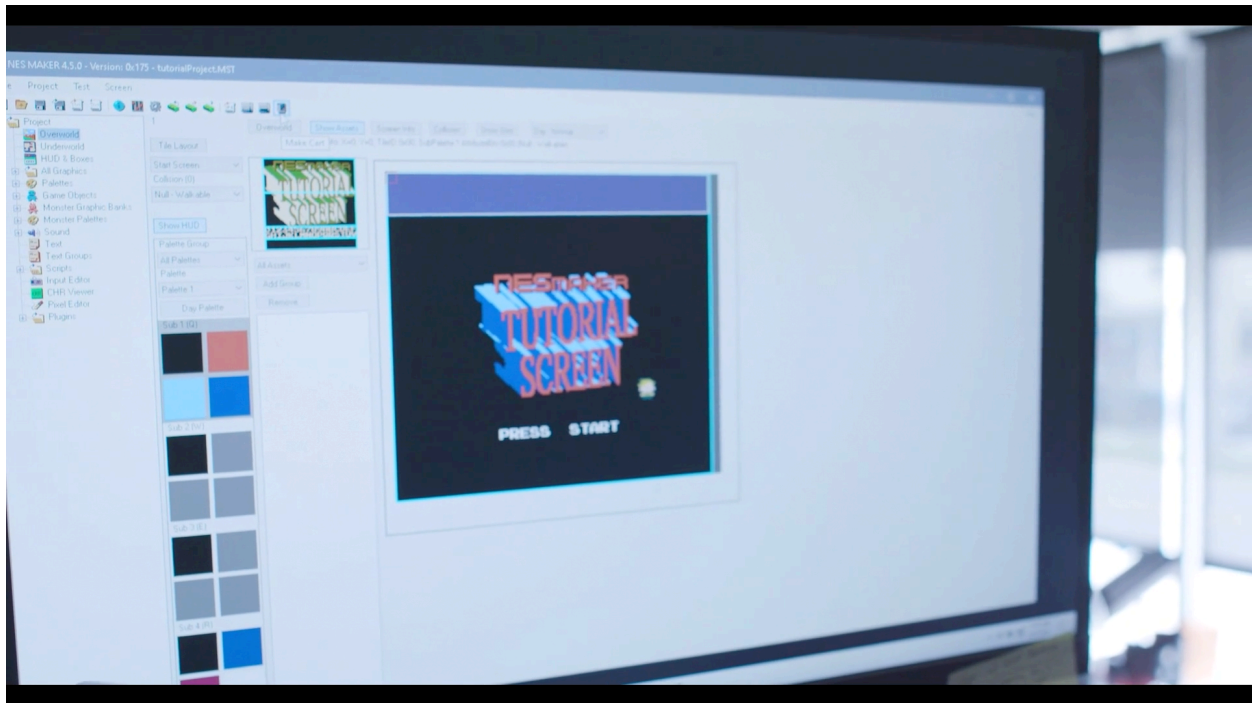
Step 1: First, you'll need a cartridge with a reflashable Mapper30 board. Check The New 8-bit Heroes website to see where to find these. Traditionally, we have used Infinite NES Lives, but we also plan to start stocking these ourselves.



Then, get yourself a cartridge flasher, also from Infinite NES Lives, also which we hope to begin stocking.



Step 2: Hook up the cartridge flasher to your computer and pop the cartridge on the flasher, with the front of the cartridge facing the flasher's NESMaker logo.



Step 3: Click on the Make Cart button at the top of the screen in the menubar. The game will compile, but instead of opening it in the emulator, it will start flashing to the cartridge. It takes about 20 seconds to finish. The resulting cartridge will play in any NES console or hardware based NES clone console. Some consoles that rely on emulation may not recognize mapper30, but that is rare.

Step 4: Put it in your NES and hit the power button.

