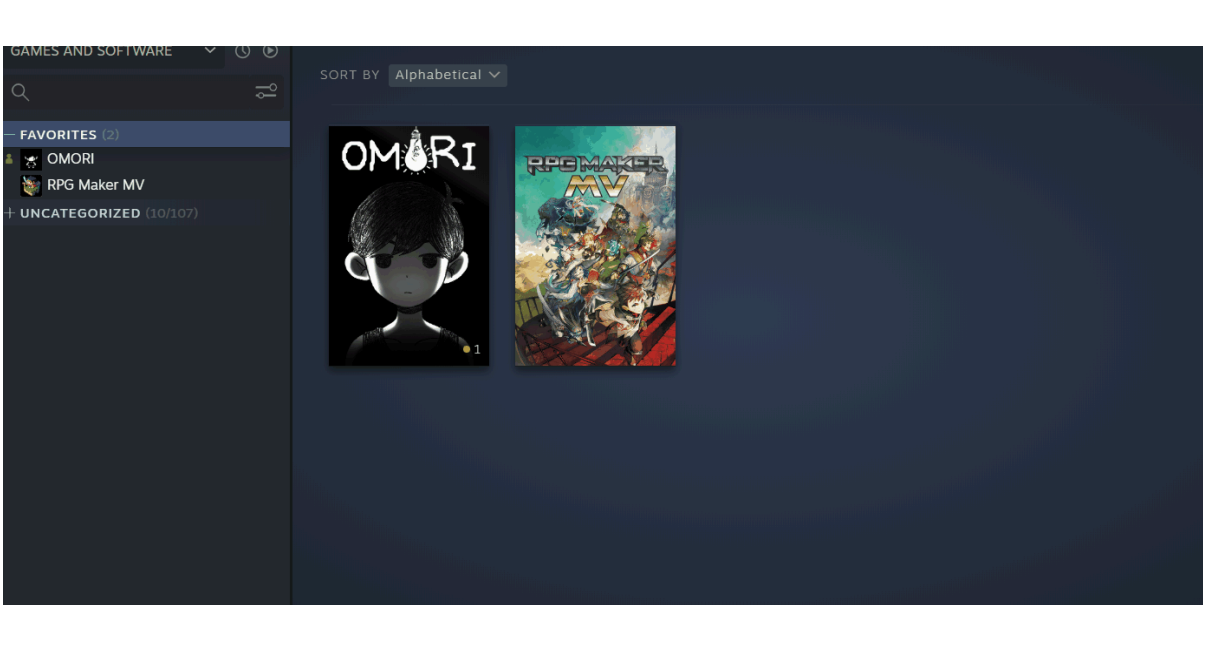
This sheet details many things you may do when making an OMORI mod.

Document Credits

june#0779
Labrallie#2333
nru#0514
ayaan#9403
Cooldry77-T1-RTD#3894
Rph#9999
Stahl#1845

Getting Vanilla Assets

To get vanilla assets to edit/use for reference, first make sure you have OneLoader, made by Rph#9999, downloaded. If you already have GOMORI installed, and would like to switch to OneLoader for more capabilities, such as better compatibility and faster loading, then you can also choose to download the OneLoader Installer Mod. Otherwise, download from the Github repo, extract the "www" folder included in the mod loader zip, into your OMORI folder. To find your game folder, right click on OMORI in steam, click manage, and then local files, as shown in the gif below.



When asked to overwrite files, click yes. You'll notice you have a couple of new folders inside the www folder, namely one labeled "mods". That folder is where you will drop zipped up mods that you've downloaded, or made, to play in-game. These mods can be downloaded from the OMORI mods website hosted by Rph#9999.

Now, you can open OMORI, click on options, mods, and then decrypt! If you want the game to be able to be opened within RPGMaker, make sure you choose that setting! The decrypt *will* take a while, so make sure your settings are correct before starting. Once it's done, you can reopen that OMORI folder, and you'll see "www_playtest" (or "www_decrypt"), along with a string of letters. Congrats, you're now based :)

What Mods Can I Make?

This really depends on your skill level and resources.

- Replacing/adding art, music, dialogue: You need nothing but your own talent! Everything can be put in correctly. For text, you'll need a good text editor like Notepad++ or Visual Studio Code which are both free. For art, you can use anything that supports transparent pngs. For music, anything that exports audio as ogg is good (but even then you can use Audacity to convert it to ogg).
- Adding or editing events, skills, enemies, characters, etc.: You will need a copy of RPG Maker MV. There is a free trial for it, and it goes on sale to around $20 pretty often.
- Adding or editing new maps, tilesets, etc.: You will need only Tiled (1.0.3) for editing maps, and both Tiled and RPGMAKER for making new ones.
- Adding new scripts: You gotta know javascript.

Setting up Mods For Use

In order to set up mods for use, you first need to make a folder with the name of the mod id you want, then create a mod.json file in it. The mod id and folder name MUST be the EXACT SAME. A template to the mod.json can be found here. Set up your files the same way they are set up in the vanilla game.
-Plugins go in the plugins tab
-Text/dialogue goes into the text tab
-Map data goes into the data tab
-In-game maps from Tiled go in the maps tab,
-Assets such as images and audio go into the assets tab. Then, put the path of these folders or files in the mod.json under its respective category. Your mod should now run as intended. Below is a fully functioning image of a mod.json.

```
{
  "id": "Ratatouille_Mod",
  "name": "Ratatouille Mod",
  "description": "Replaces Doughie with Remy the Rat, and Biscuit with Linguini",
  "version": "1.0.0",
  "files": {
    "plugins": [],
    "text": ["languages/en/"],
    "data": ["data/"],
    "maps": [],
    "assets": ["img/enemies/" , "img/sv_actors/" , "img/characters/" , "img/battlebacks1/" , "audio/bgm/"],
    "exec": []
  }
}
```

Image taken from the mod.json used in the Ratatouille Mod.

(Text) Proper .YAML File Formatting

Important thing to note: battle text works a little differently from overworld text! Scroll a little farther down to check out the battle text alternative, however, you should still read the overworld section, as most things are the same!

- **Overworld Text**

To make a proper .YAML file and get all that fun dialogue into your game, you'll need the face sprite sheet open (file name: MainCharacters_DreamWorld.png). After that, know that a yaml follows this order:

```
message_0:
  faceset: MainCharacters_DreamWorld
  faceindex:
  text:

message_1:
  faceset: MainCharacters_DreamWorld
  faceindex:
  text:

message_2:
  faceset: MainCharacters_DreamWorld
  faceindex:
  text:
```

And keeps going up by numbers from there. If a character that's talking doesn't have a face sprite for it, or a little flavor text for that refrigerator you want to be able to interact with, you would use:

```
message_0:
  text:
```

And keep going up by numbers from there. The reason you need the face sprite sheet open (called MainCharacters_DreamWorld) is to display the correct face. to do this, you would just count. top left starts at 0, and it goes from right to left, and then top to bottom.

0 1 2
3 4 5

If you don't understand this, please move to the "Making Custom Art" section, which goes into a little more detail.

Whichever face you count to, let's say we want 5, we put that number in the faceindex column, like so:

```
message_0:
  faceset: MainCharacters_DreamWorld
  faceindex: 5
  text:
```

Finally, we need the actual text, which, obviously goes in the text row. start off by writing \n<CHARACTER NAME HERE> where it says "character name here", fill that in with whatever character is talking, such as:

```
message_0:
  faceset: MainCharacters_DreamWorld
  faceindex: 5
  text: \n<AUBREY>
```

Then, you write the text as normal:

```
message_0:
  faceset: MainCharacters_DreamWorld
  faceindex: 5
  text: \n<AUBREY>OMORI sucks, and it's a low tier game. Trash all-around.
```

Now, text has certain formatting rules on it's own, I will only be mentioning the important one and wont go through each and every single one, because there's a lot, but if you want to take a look (which i suggest doing), all of them are detailed here:

https://github.com/Gilbert142/gomori/wiki/Text-Formatting

The one you should be using a lot is:

\!

This makes it so the player has to click Z to progress the text input. To use it, you should just place it where the text you want to progress from starts, like so:

```
message_0:
  faceset: MainCharacters_DreamWorld
  faceindex: 5
  text: \n<AUBREY>OMORI sucks, and it's a low tier game. \!Trash all-around.
```

The flavor text alternative would look like:

```
message_0:
  text: OMORI sucks, and it's a low tier game. \!Trash all-around.
```

And now, you've successfully made a dialogue box! Make sure there is no extra space between the characters, otherwise there will be a double space, which is ugly.

Here is an image of a working .YAML file:

```
message_0:
  faceset: MainCharacters_DreamWorld
  faceindex: 2
  text: \n<AUBREY>Hey...\! What is this place?

message_1:
  faceset: MainCharacters_DreamWorld
  faceindex: 28
  text: \n<KEL>Yeah.. What is this?\! And what are those kitchen appliances doing here?

message_2:
  faceset: MainCharacters_DreamWorld
  faceindex: 3
  text: \n<AUBREY>Maybe I should hit one with my bat!

message_3:
  faceset: MainCharacters_DreamWorld
  faceindex: 39
  text: \n<HERO>No Aubrey!\! Don't mess with those.. I have a better idea..

message_4:
  faceset: MainCharacters_DreamWorld
  faceindex: 11
  text: \n<AUBREY>Fine fine...\! What's your brilliant idea HERO?

message_5:
  faceset: MainCharacters_DreamWorld
  faceindex: 11
  text: \n<AUBREY>Hey HERO, what are you doing?
```

Taken from The HERO Cooking Show.

- **Battle Text**

The key difference between battle text is the way things are formatted. In battle text, things are formatted like so:

```
message_0:
  text: "CHARACTER NAME HERE: Actually, I disagree, OMORI is a top-tier game!"
```

This is shown below:

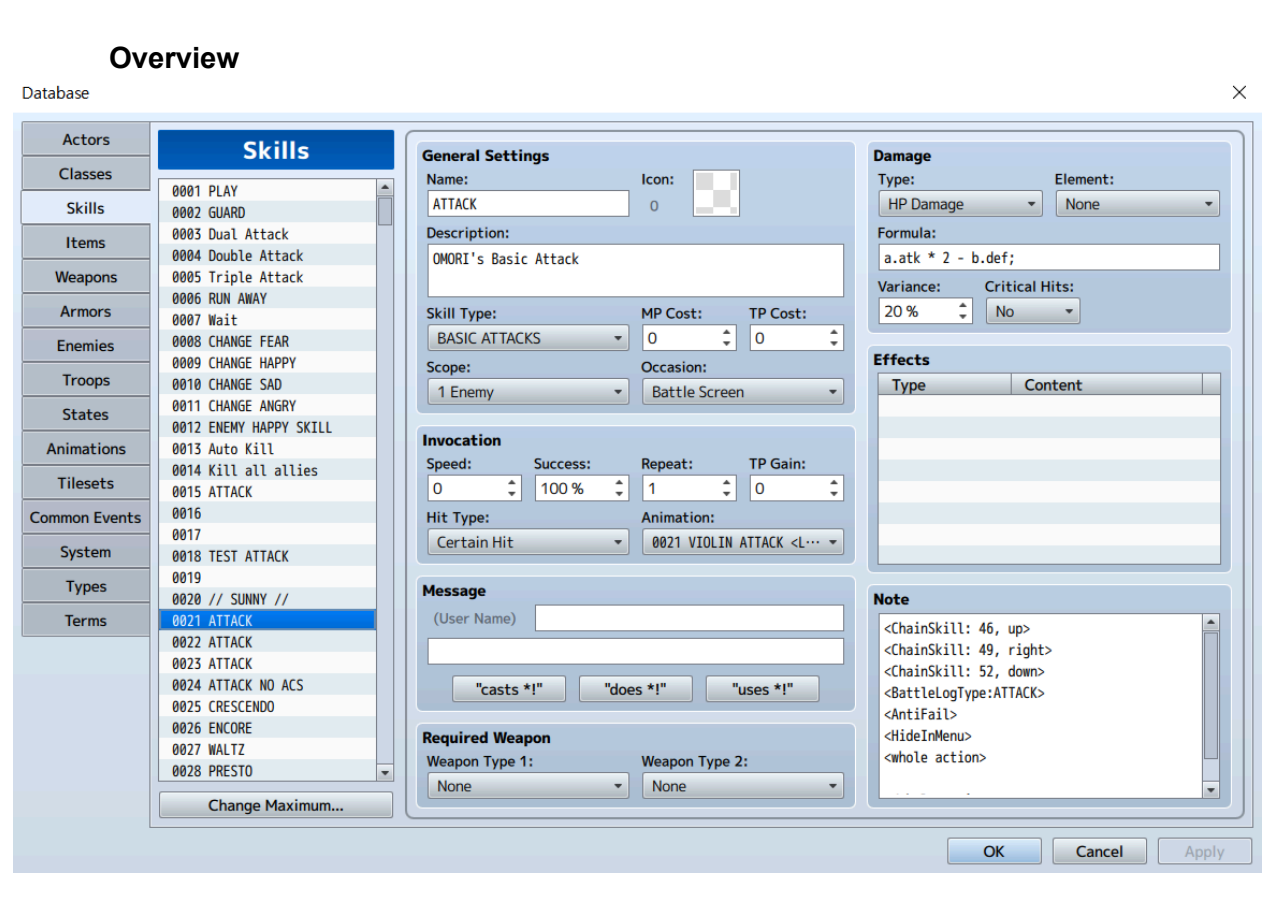
```
message_0:
  text: "HERO: It seems like you finished picking the ingredients!"

message_1:
  text: "HERO: It's time for the taste test!"

message_2:
  text: "HERO: Here goes nothing!"

message_3:
  text: "HERO: Oh lord, this is dog shit. I feel like I'm gonna keel over and die..."
```

Also taken from The HERO Cooking Show.

Making Custom Skills

Skills are essentially where most gameplay magic happens, aside from troop events. Because of this, you might want to learn how to make various things happen using skills, in order to achieve desirable results. Skills can do a number of things, from dealing damage, applying buffs and debuffs, change emotion, etc.

Overview



Let's look at the default UI that RPGMAKER MV has in the skills tab.We'll be going over each item in this screen, one by one.

- **General Settings**

Here is where you can set some basic parameters for your skill, such as the juice cost and target.

a. **Skill Type**

Skill Type has a very limited use in OMORI. Three types exist: BASIC ATTACK, POWER OF FRIENDSHIP, and CALM DOWN. The vast majority of skills you will deal with, and probably make, are in the POWER OF FRIENDSHIP category. Basically all dreamworld, and most real world skills, go in here. BASIC ATTACKS is just as the name suggests: normal attacks for party members go here, and nothing else. CALM DOWN is for the skill Calm Down only.

You might be asking the question: why go through the trouble of assigning skill types, if almost all of them are going to be in one? The answer to that is in the AFRAID emotion. AFRAID prohibits the inflicted party members from using any skills, *with the exception* of normal attacks, and in the case of SOMETHING battles, Calm Down. The game achieves this by making the AFRAID state seal POWER OF FRIENDSHIP skills.

b. **MP cost**

This one's a no brainer. Simply write down the juice cost for your skill. Remember to reference this in the skill description, just like the base game does.

c. **Scope**

Here is where you can specify what your skill affects. RPGMV has these options by default:
- The User
- 1 Enemy
- All Enemies
- 1 ~ 4 Enemies
- 1 Ally
- All Allies
- 1 Dead Ally
- All Dead Allies

You can choose out of these scopes as you see fit. Unfortunately RPGMV does not innately allow more choices: however, if you want, you can obtain Yanfly's Target Core plugin to make custom targets yourself.

d. **Occasion**

This determines where the skill can be used. If it's set as Battle Screen (this will be the case for almost all playable character skills), it will only be usable in battle; vice versa in Menu Screen. Skills will always be usable with Always.

-**Invocation**

a. **Speed**

Speed is a stat that sets turn order in battle. The higher the speed, the faster you will go. Most skills will have 0 as their speed stat: in that case, its speed will only be determined by the user's speed stat, and related buffs. However, if you'd like a skill that always goes first, like Guard, you can set the speed accordingly.

b. **Success**

You likely won't be messing around with this one too much, unless you discard hit/evade rate and use this as the primary hit rate factor. Like the name implies, this sets a percentage of the odds the skill will succeed. In most circumstances, you'll want this to be 100%, the default.

c. **Repeat**

This number specifies how many times the skill's action will repeat itself. This can also be done with notes coding.

d. **Hit Type**

There are three types of hit types: Certain Hit, Physical, and Magical. Magical is not used in OMORI. The two remaining has a simple difference: certain hit has a 100% hit rate, and physical is affected by other hit rate factors. Most buff/debuff and emotion change skills use certain hit, while most skills that deal damage use physical.

e. **Animation**

Here you can choose which animation the skill uses. This can also be done with notes coding.

-**Message**

We'll look into this in the Notes section, along with other methods of showing messages.

-**Required Weapon**

I've personally never used this one, but presumably it makes it so that the skill can't be used without a certain weapon. OMORI doesn't use this, or at least I haven't seen it yet.

-**Damage**

NOTE: there's a really good rpgmv forum post on damage formulas here, aside from my explanation of it. This post goes into a lot of detail that I don't go into. My post focuses more on how it relates to OMORI's gameplay, so I recommend reading both.

a. **Type**
There's a lot of different ways skills can affect the target's HP. As a default, RPGMV allows HP/MP damage, heal, and drain. You can also make additional HP/MP effects using notes coding.

b. **Element**

Elements are a very important part of OMORI, especially with the emotion system. The core mechanism in emotion advantages and damage multipliers uses elements. For that, skills and their elements are a must-know for modders.

Most of the skills you will see have 'None' as their element. This means that states can change that skill's element. You can see where this comes in: emotion states can change the skill's element into that of the element: e.g. the HAPPY element for players that are happy.

Alternatively, setting a skill's element into something that is not 'None' means that its element is now set. States cannot change this skill's element. This is the case for skills like the Snow Angel's 'Exploit Emotion'. Its element is set as 'EMOTION', which is an element that is effective on all emotions. Giving the angel an emotion does not change the element of the skill.

c. **Formula**

This is one of the most important parts of the skill itself. This will dictate the damage that the recipient of the skill will receive. RPGMV's damage formula follows a lexicon that you must know for it to work.

The first thing one should know is prefixes. Every stat in the formula is preceded by a prefix, usually 'a.' or 'b.' The former denotes that the stat is that of the skill's *user*; the latter means that the stat is that of the *recipient*.

These two are the prefixes that you will see for the vast majority of the time. However, there may appear a time where a skill may need the stats of a third party. In OMORI, this happens with follow-up skills: Kel's follow-up with Hero has a damage formula where Kel and Hero's ATTACK stats are added together. To achieve this, the damage formula uses a javascript expression of Hero as the prefix:

$gameActors.actor(4)

Where Hero's actor ID is 4, so the formula uses 4. This form actually appears quite frequently in Battle Action Text, Notes coding, and the like: it's useful to remember it. Aside from Hero, the actor IDs 1, 2, and 3 correspond to OMORI, Aubrey, and Kel respectively.

After the prefixes are the stats themselves. There are various stats in the game, but the one's most frequently used are:

atk - ATTACK, def - DEFENSE, agi - SPEED, luk - LUCK

Many, many other parameters exist, however. I recommend looking at the forum post mentioned above for a full list of them.

Now that we know about prefixes and stats, we can properly take a look into some skills in the game itself. Take a look at Kel's follow-up with Aubrey, for instance:

$gameActors.actor(2).atk * 2 + a.atk * 1 - b.def;

We can see that this skill adds Aubrey's ATTACK, times 2, and Kel's ATTACK, then subtracts it with the recipient's DEFENSE. This type of formula is the standard for most damage-inflicting skills in OMORI.

The final major thing in damage formulas to note is something I call 'State Checks'. For some skills in OMORI, damage dealt differs based on the emotion of the skill's user. This is possible because the game checks to see if the user is under a certain state, and if so, changes the damage formula accordingly.

State Checks are done with a simple javascript if - else if script. However, something called 'ternary operators' can be used as well, which makes the formula simpler. OMORI uses the latter method.

It's simpler to show than to tell here, so let's see an example.

(a.isStateAffected(10)) ? a.atk * 2.5 - b.def : (a.isStateAffected(11)) ? a.atk * 4 - b.def : (a.isStateAffected(12)) ? a.atk * 6 - b.def : a.atk * 1.5 - b.def;

This is the formula of one of my skills, called Punchdown. It's designed to be an attack that gets stronger as the user becomes sadder. For context, states 10, 11, and 12 denote sad, depressed, and miserable respectively. The formula can be summarized as:

(state check) ? (damage formula) : (state check) ? (damage formula) : …

':' divides between damage formulas, and ';' states that the formula has ended. The final item in the above formula, the one without the state check prior to it, is of course the formula to be used when none of the state checks beforehand are true.

You may sometimes find that even with these assets, it may be difficult for you to fully realize your skill idea: in that case, it is possible to create a custom damage formula using notes coding.

d. **Variance**

The percentage marked here denotes how much the damage the skill does varies as it is used. It'd be unnatural for skills to do exactly the same amount of damage every time, thus why this feature exists.

e. **Critical Hits**

This is a simple yes/no choice, which makes it so that the skill has the ability to land a critical hit, or is unable to.

-**Effects**

Effects are quite straightforward. These are the various things that can happen when the skill is used. **However**, all of these effects can be achieved by notes coding, and with greater control. I highly advise using notes coding for your intended effects, seeing as OMORI does the same for its skills as well.

-**Notes**

Despite its name, this is by far **the most important part in this whole section**. If you want to do anything interesting with your skills, being at least acquainted with notes coding is a MUST. In normal RPGMV, this section is nothing but a place to jot down any spare thoughts and to-do lists about the skill. However, with Yanfly's various script plugins (which pretty much every respectable rpgmv game uses), Notes is transformed into a sort of small programmable area, where much more custom effects are allowed to happen. Here we'll look into just about everything important you can, and probably should, do with the Notes section.

a. **Notetags**

The first thing to know are the various notetags one can utilize in this section. These have various effects, from specifying battle text, to controlling when and how animations and whatnot appear.

1. **<xxx action>**

Notetags such as <setup action>, <whole action>, etc. Are a family of notetags that determine the timings for the various effects a skill can have. For example, you might want a skill to show two different animations: one appearing before damage is dealt, and one after it. With these notetags, you can have precise control over them.

One important thing to note is that the various effects to be held within these notetags **must** be wrapped around by a corresponding 'end of notetag' tag. The format for these are:

</xxx action>

So effects under <setup action> must be closed by </setup action>, and so forth. Now for each of these notetags.

<setup action> is used for effects that happen before the primary effect of the skill (i.e. the damage dealing part, etc.) happens. If you need a state to be added, or an animation to be played, before the skill deals damage, this is the place to put those designations.

<whole action> is, paraphrased from the official descriptions, for actions that affect all targets simultaneously. You'll probably be using this one the most, as its timing is just about equal to that of the main action of the skill. Be careful, though - effects (especially AOE) that are applied by <whole action> notetags work a bit funky. Basically, my best guess is that it applies the effect into one target, then uses that result on all the other targets.

For example, let's say that an effect raises the defense of two targets, A and B. If A is at +1 def and B is at 0 def (in terms of buffs), using a <whole action> notetag to raise the target defense makes the game administer the effect on A, making it +2 def, then use the same result on B, making it +2 as well. That's really bad. In these cases, I suggest using the next notetag.

<target action> applies effects on each target, one by one, solving the above problem. It also happens to be where the main effect of the skill is located. If <target action> is not specified in the notes section, the game uses it anyway, with the default of:

<target action>
action effect
</target action>

Where 'action effect' denotes the main effect of the skill. We'll look into this a bit later.

<finish action> is for effects that are supposed to happen after the main effect has been executed. I personally don't use this one too much, but it does have its specific uses: mostly animations, or states that shouldn't be applied before damage is dealt.

These are the effect notetags that you'll be using for the majority of your modding. There are, however, many more, like <follow action>: If you want to know more, I recommend looking up the official descriptions, linked above.

2. **<BattleLogType:xxx>**

This notetag is used in most skills in OMORI. What it does is call on a special js file called **Custom Action Battle Text.js**, and search for a battle log message with the corresponding id (the xxx part). If one is found, then the game displays it. This is elaborated later in the Messages section.

3. **<AntiFail>**

You may want to make a skill that does not directly deal damage, but maybe applies a skill or initiates a common event. In that case, if the skill is used, the game may display a **'It had no effect on (target)!'** message, or a 'no effect' message of similar meaning. In case you don't want that happening, simply make an **<AntiFail>** tag: it will prevent it from occurring.

4. **<HideInMenu>**

Some skills, like normal attack skills, are not supposed to appear in the skill equip menu, nor should it be able to unequip them. In this case, this notetag will effectively hide the skill from the skill menu, and still treat it as being 'equipped'.

5. **<Enemy or Actor Select>**

In OMORI, emotion skills such as PEP TALK can change the emotion of an ally *or* enemy. This normally isn't possible in vanilla RPGMV, but the game circumvents this by use of this particular notetag. Use this one on skills whose targets can be any enemy or any ally.

6. **<Damage Formula>**

I mentioned before that damage formulas can be done with notes coding. Well, this is the place to do it. The final damage formula is denoted as the variable **value**, so keep that in mind and don't declare/use other variables with the same ID.

Let's look at the notes tag damage formula for Aubrey's WIND-UP THROW as an example.

<Damage Formula>
// Check if the damage has already been calculated

```
if (this._calculatedBaseDmg) {
// Set the damage to the cached amount
value = this._calculatedBaseDmg;
// Otherwise
} else {
// Get the total number of enemies
var totalEnemies = target.friendsUnit().aliveMembers().length;
// If there is 1 enemy total
if (totalEnemies === 1) {
// Use this formula
this._calculatedBaseDmg = user.atk * 3 - b.def;
// If there are 2 enemies in total
} else if (totalEnemies === 2) {
// Use this formula
this._calculatedBaseDmg = user.atk * 2.5 - b.def;
// If there are 3 enemies in total
} else if (totalEnemies === 3) {
// Use this formula
this._calculatedBaseDmg = user.atk * 2 - b.def;
// if there are 4 or more
} else {
// Use this formula
this._calculatedBaseDmg = user.atk * 2 - b.def;
}
// Set the value equal to the calculated amount
value = this._calculatedBaseDmg;
}
</Damage Formula>
```

The comments do a pretty good job of explaining what's going on. Depending on the number of enemies, the game changes the value of **calculatedBaseDmg**, then equates the variable **value** to that, which is the final damage formula. You can see that it is possible to formulate much more complicated formulas using this method.

7. **Follow up notetags**

There are a couple of notetags regarding follow-ups in OMORI. However, their roles are extremely niche and specific, so I'll only skim through them.

<EnergyCost: xxx> how much energy the follow up consumes. Usually 3, with the exception of release energy, which is 10.

<ChainSkillIcon: xxx> specifies the bubble that appears for this follow up attack. See img/system/ACS_Bubble.png for order.

You probably won't be needing these, unless you're working on a character replacement mod or the like. In that case, kudos! Best of luck.

There are a lot more notetags you can use, such as **<Custom HP cost>, <Post-Damage Eval>, <After Eval>,** etc. There are way too many to list them all, so I've explained the only most commonly used ones. I don't know if a full list of these exists: there used to be one, but as far as I can tell it's been deleted. I recommend looking through the game's skill notetags, or giving a google search, if you want to know more. In any case, you'll be able to achieve most of what you want to happen with the notetags described above.

b. Messages

There are three main ways to show messages on the battle log as a skill is executed.

- **Message Tab**

The first is not using the notes at all, but instead using the Messages tab in the UI. This is certainly possible, and easier than the other two methods. However, with this you can only write two lines of words, and the message cannot change dynamically as effects of the skill differ based on situations. I suggest using this method only on extremely simple normal attack skills. Also note that this method will always display hp damage text when the skill deals damage. ("A took X damage.")

- **Custom Action Battle Text**

This is, in my opinion, the most elegant way to display text of the three. This method utilizes a special js file in the plugins folder called 'Custom Action Battle Text'. Let's open it and look at one of the items.

```
case 'ATTACK': // ATTACK
  text = user.name() + ' attacks ' + target.name() + '!\r\n';
  text += hpDamageText;
  break;
```

This is the most commonly used battle text in the game, the one for normal attacks. When making your own items in this file, make sure to follow the standards set by other items.

Each skill in OMORI calls on its specific battle text using the notetag **<BattleLogText: >** . The case name for the skill goes after the colon, no spaces. So in the case above, the notetag would be <BattleLogText:ATTACK> . Try looking in the notes section of an already existing skill: this notetag can be found at the very top.

Below the skill's case name is the text that should be displayed. **user.name()** and **target.name()** functions are self-explanatory enough. Remember that the strings these functions return do not automatically have a space before and after them, so when connecting them with strings of your own, make sure to insert spaces, like ' attacks ' above.

You might be wondering what the '**\r\n**' at the end of the second line does. This indicates that the following text goes on a different line. Think of it like an 'enter' key. If you don't put this in between lines, they'll all go together in one jumbled up line, so keep that in mind. Also important is the fact that the **\r\n** goes *inside* the string, not outside.

Any new line other than the first one is declared with **text += ( )** , since you're adding onto an existing line with a new one. In the third line, the variable **hpDamageText** calls for - you guessed it - hp damage text. This means that you have the choice to not display them as well, which can be useful if your skill repeatedly deals damage. Each line of hp damage text takes about 2 to 3 seconds to load in and then out, which can make them especially tedious in gameplay in the above scenario.

Finally, at the end of all the text that should be displayed MUST be a '**break;**' statement. You might be familiar with this, it ends the process of reading the js file. If you don't put this in, the game will read the text of the *next* skill in the js list instead of the one you want, upon skill execution, so be careful.

With the basics done, let's look at some advanced battle text items, namely for skills that change emotions / apply buffs and debuffs.

```
case 'KNEAD': // KNEAD
  text = user.name() + ' prepares ' + target.name() + '\ for a\r\n';
  text += 'good baking.\r\n';

  if(!target._noStateMessage) {text += target.name() + '\'s DEFENSE fell.\r\n';}
  else {text += parseNoStateChange(target.name(), "DEFENSE", "lower!\r\n")}

  text += user.name() + ' is tired of bread...\r\n';

  if(!user._noEffectMessage) {
    if(user.isStateAffected(10)) {text += user.name() + ' feels SAD.';}
    else if(user.isStateAffected(11)) {text += user.name() + ' feels DEPRESSED..';}
    else if(user.isStateAffected(12)) {text += user.name() + ' feels MISERABLE...';}
  }
  else {text += parseNoEffectEmotion(user.name(), "SADDER!")}
break;
```

This is the text for another one of my skills, called Knead. It's a skill that lowers the defense of the recipient, while making the user sad. The second and fourth parts of this text, as you can see, are the parts that deal with defense lowering and emotion change respectively. I won't go into too much detail for this bit, because a) it's way more complicated once you go into depth, having to look into function declarations in multiple core plugin js files, and b) these aren't very customizable, instead acting as a chunk of code, so you won't have to play around with it (unlike the basic functions and variables above)

Although this big chunk of text may seem intimidating at first, it's quite intuitive once you get a good look into it. For the second part, it checks if a state change in the enemy has been detected, and if positive, displays the 'defense lowered' message. If not, it instead shows the 'defense cannot go lower' message. For the fourth part, it does a similar thing: it checks for changes in the user this time, and displays an emotion change message according to the emotion the user is feeling.

To completely understand how the game does this, you'll have to dig through some more advanced game files, namely **rpg_objects.js** and **GTP_OmoriFixes.js** . This is completely optional, however: unless you aren't planning to make some serious changes to how emotions and buffs work in OMORI, you'll be perfectly fine just copy-pasting from existing items in the js. Only a little bit of common sense is necessary: you might have to change around the numbers in the **isStateAffected()** functions, for example.

Ok, one last thing to look at, then we'll be finished with Custom Action Battle Text.

```
case 'PASS HERO 2':  // KEL PASS HERO
  if(target.index() <= unitLowestIndex) {
    text = user.name() + '  dunks on the foes with style!\r\n';
    text += "All foes' ATTACK fell!\r\n";
  }
  text += hpDamageText;
  break;
```

This is the battle action text for Kel's follow-up with Hero. Notice anything different? The lines regarding text are wrapped around by an if statement. This statement, **if(target.index() <= unitLowestIndex) {}** , is a device used for skills that affect more than one target. What it does is only apply the battle text to one target, the one with the lowest index, when the skill is executed. If you don't use this while making AOE skills, the text will repeat itself for the number of targets. So you'll want to remember this, it's quite important.

Of course, the entries above are not the only things you can do with your battle text. There's a couple things in the js file itself that aren't described here (they appear only a few times and are pretty gimmicky), and if you have sufficient knowledge of javascript, obviously the world is your oyster. These are the main building blocks you'll have to know, however.

- **Eval: text**

This is a bit of an unorthodox method: and coincidentally, my favorite. Basically, it's possible to write the battle text in the notes section itself, without having to bother with the Custom Action Battle Text file. It's a bit cluttered, but this allows you to time the text with exact precision, something that you can't do with the other two. It's used, albeit rarely, in the base game, and I suggest at least looking into it.

One of the cases where eval statements are used in OMORI is in Aubrey's twirl.

```
if user.isStateAffected(7)
eval: BattleManager.addText(user.name() + " feels ECSTATIC!!")
else if user.isStateAffected(6)
eval: BattleManager.addText(user.name() + " feels HAPPY!")
end
```

As you can see, adding text via notes is very simple. You start off with the base of:

eval: BattleManager.addText(*text here***)**

(or alternatively,)

eval: SceneManager._scene._logWindow.push(*text here***)**

And use operators to control how those texts show up. Hp damage text is displayed automatically. It's actually a bit funky how that works, so we'll discuss it some more just a bit later.

Buff state change, and more advanced things, can be done with eval as well. Taken from the notes section of OMORI's Mock:

```
if target.isEmotionAffected("angry") && target.result().isHit()
add state 94: target
eval: $gameTemp._mockCheck = true
end
if !target._noStateMessage && $gameTemp._mockCheck
animation 219: target
wait: 30
eval: SceneManager._scene._logWindow.push("addText", target.name() + "'s
ATTACK fell.")
else if $gameTemp._mockCheck
eval: Gamefall.OmoriFixes.parseNoStateChange(target.name(), 'ATTACK',
"lower!")
end
eval: $gameTemp._mockCheck = undefined
```

This is a bit complicated to explain: I'm sure those experienced in js can understand it well enough, but it makes use of a lot of pre-declared functions and variables, making it difficult to understand for someone who's just started out on modding. This is why I recommend that you use the second method for emotion and buff state changes.

Now, I'll explain just what makes this method very very useful. I'll go into this a bit later as well, but it's possible to make various effects happen, at certain moments during the skill execution itself, using notetags like **<setup action>** and **<whole action>**. This means that, depending on where you place the eval statement, you have *absolute control* over when and how the text appears as the skill is used.

Take an attack + debuff skill, for instance: you want a 'x does y!' text to appear as the main damage animation plays out, and then after that, 'z's STAT fell!' to appear along with the debuff animation. With the first two methods, the two lines of text go alongside each other, and it's difficult to line the animations along with them. However, by using notes, you can make everything come out in a clean order. Something like this:

```
<setup action>
//first line of text
eval: BattleManager.addText("x does y!")
</setup action>

<target action>
//initial damage animation
animation xxx: target
//deals damage
action effect
//the number of frames to wait can change depending on the length of the animation
wait: 30

//in this case, the skill lowers the target's attack
if target.isStateAffected(93)
  add state 94: target
  remove state 93: target
 else if target.isStateAffected(92)
  add state 93: target
  remove state 92: target
 else if target.isStateAffected(89)
  remove state 89: target
 else if target.isStateAffected(90)
  add state 89: target
  remove state 90: target
 else if target.isStateAffected(91)
  add state 90: target
  remove state 91: target
 else
  add state 92: target
end

//animation 219 is the 'enemy stat down' animation
animation 219: target
//second line of text ('stat fell!')
if !target._noStateMessage
eval: SceneManager._scene._logWindow.push("addText", target.name() + "'s ATTACK
fell.")
else
eval: Gamefall.OmoriFixes.parseNoStateChange(target.name(), 'ATTACK', "lower!")
end
</target action>
```

(Note that I just wrote this on the fly, so the timing might not be perfect, but you get the idea. With this method and slight trial and error, you can customize the timing on your text so it looks very satisfying.)

c. Mid-Skill Shenanigans

Now, here is the juicy bit. I'll be describing the effects you can make your skill apply in this section, ranging from multi-hit attacks, buffs and debuffs, emotion changes, animations, and much more. You'll see that Notes is a very powerful section, capable of doing things you wouldn't expect it to be able to do. Note that all of the code described here only works when they are placed within action notetags (<whole action>, etc.)

- **action effect**

This simple code denotes the main action of the skill, mostly damage infliction as followed by the damage formula. This piece of code, as mentioned before, belongs by default to the <target action> notetag: This means that if you make use of said notetag, you *must* add in an **action effect** within it to make the main effect be executed properly. This also means that you can choose *not* to do it if you wanted to, and have the damage execution happen somewhere else.

I said before that multi-hit attacks can be done with notes coding, as well as the 'Repeat' section of the skill's Invocation. This is simply done by ***repeating*** '**action effect'** as many times as you want the skill to hit. Of course, in this case the number of repetition in the 'Repeat' section should be set to 1. A good example of this is the skill RED HANDS:

```
<target action>
action effect
action effect
action effect
action effect
</target action>
```

You can see that damage is dealt 4 times, just as the skill description says.

Be careful of using **action effect** within notetags that aren't <target action>, however: unless you purposefully make a <target action> notetag ***without* 'action effect'**, the game will understand the code as having *two* instances of **action effect**, one in your specified notetag and another hidden within <target action>, and execute the effect twice. Keep this in mind.

- **animations**

I also said that animations can be done with notes coding. In fact, this method of displaying animations is far superior to the one intended by RPGMV, because of the fact that notes coding allows for you to display multiple animations for one skill, as well as specify targets(origins) for them. The basic form of displaying animation is:

animation xxx: (target)

Here, xxx is the animation's ID (a number). Simply check the animations tab in the database: the ID is right next to the name, on the left. The target is denoted by a set of predetermined lexicon, such as:

user - the user of the skill
target - the target
enemies - the enemy screen
actors - all allies
character x - one ally, with their actor ID as x

In my experience, *target* is the one most frequently used. *Enemies* is used in cases where you need to display an animation on the enemy, but not on each one. (Although this can also be done by setting the animation's position to *screen.*)

Sometimes, when displaying multiple animations back to back, you may find that the later one starts appearing too soon, overlapping with the previous one. In that case, it may be useful to halt the game's reading of the code until the first animation has ended, by following up its code with

wait for animation

This will simply stop execution of the skill's effects until the animation has ended. To be honest, this particular code doesn't seem to be very consistent in its effect: sometimes the skill execution will resume before the animation has ended, and sometimes far too late after it. If such a scenario happens, you can also set a custom delay time using the next piece of code: '**wait'**.

- **wait**

You might have noticed, but I find that timing is very important in the execution of skills. Very often, you can find that the various effects for your skills happen all at once, somewhat ruining that. You may want to create certain delays between each effect, so that a good tempo can be maintained to keep the flow of battle going. In this case, the code that is used is as follows.

wait: xx

Where 'xx' denotes the number of *frames* to wait before moving on with the execution. Notice that this is *frames*, not *seconds*: since RPGMV runs at 60 fps, you should write in **wait: 30** in order to create a half-second delay.

- **States & Common events**

Whether it be buffs/debuffs, emotions, or other more technical things, states play an important role in OMORI. The primary code for applying a state on a target is:

add state xxx: (target)

Again, 'xxx' is the ID of the skill, and the target is denoted by terms such as **user, target, character x**, etc. Some follow-up skills utilize **character x** to apply states to allies that are neither the user nor the target: see Aubrey's follow-up with Kel.

Common events work similarly, with the format being

- **If statements**

Thanks to Yanfly's plugins, the notes section (within action notetags) can be treated as a javascript coding platform. As such, if-else if statements are used frequently. Anyone with experience in coding should know how these work: just remember that this is javascript, so if statements must be followed by a set of '{}' . Also make sure to follow up a full if statement with '**end**': if not, the game will treat everything after it as part of the statement as well.

- **Applying buffs/debuffs**

The application of buffs and debuffs usually follow a pretty strict format, which makes it easy to just copy and paste from existing entries. In fact, I just made a reference sheet for buffs and debuffs a couple months into modding. I'll link to it here. **However**, be warned: the format I use is a bit different from OMORI's. For one, OMORI doesn't have hit rate/evasion as a buff parameter. More importantly, in the base game, buffs and debuffs are treated as separate and they do not stack: even if they're for the same parameter. I personally don't like it, so instead I use a format that allows for the two to stack properly.

Simply copy-paste an existing entry for the buff/debuff of your choice within an action notetag. I recommend using <target action>, for reasons supplied above in the notetag section.

- **Applying emotions**

This works similarly to the above: unless you aren't doing anything crazy, like instantly give someone a +3 emotion, you'll be fine just copy-pasting an existing code for emotion changes, which are fairly abundant among the skills in this game. Here's an example:

```
 if target.isStateAffected(15)
add state 16: target
else if target.isStateAffected(14)
add state 15: target
else
add state 14: target
end
```

This particular bit of code makes the target angrier.

Obviously, there are a **LOT** more things you can do, since this all runs on javascript. The above are only the basics. You can still achieve quite a lot of things (even with these limited tools) with a bit of creativity, however. If you want to make a really custom, special effect for your skill, one that is a bit difficult with the devices described here, then chances are that there are skills in OMORI itself that work similarly. I recommend looking for those yourself, and trying to reverse engineer how they work. If I can do it, I have the utmost certainty that you can, as well.

This marks the end of the Skills section.
If you find any errors within this section, or if there is anything else you'd like to know, I would be very grateful if you could dm me about it. Thank you.
-nru#0514

Making Custom Art

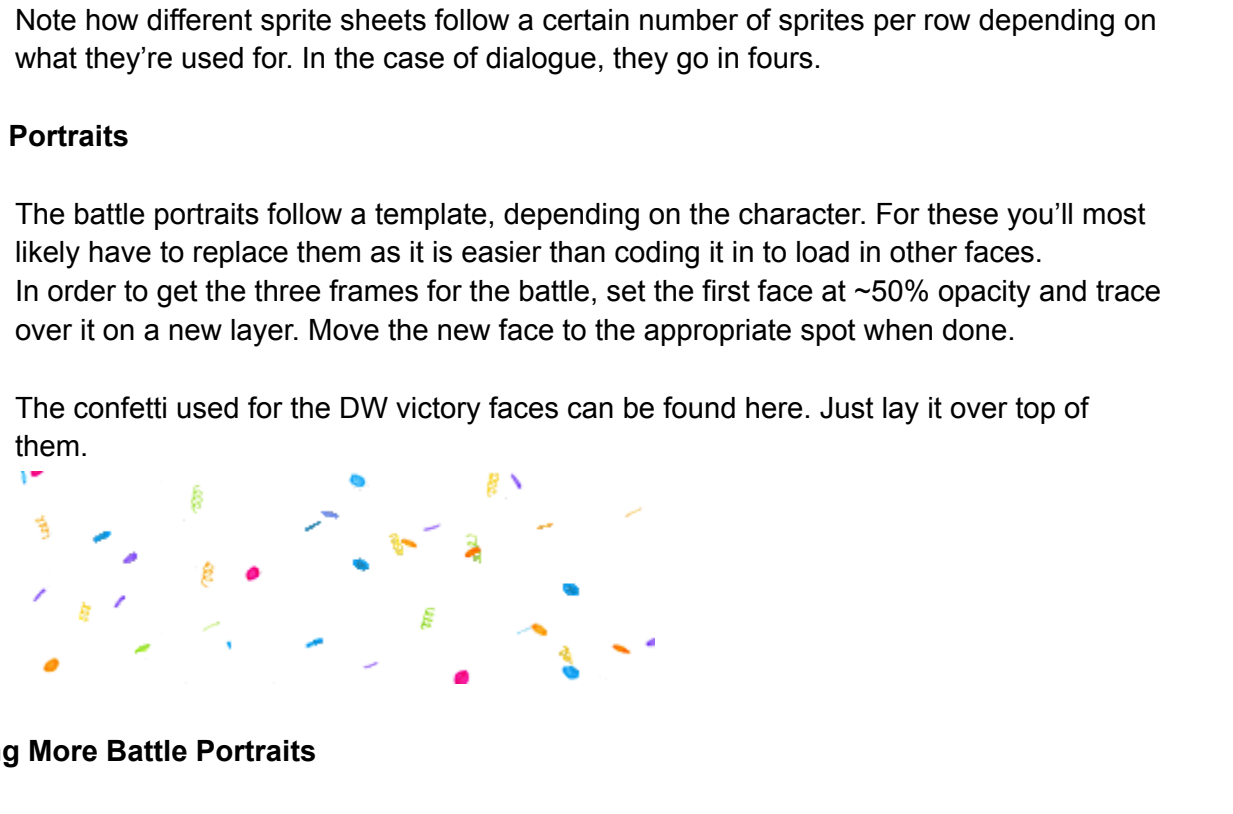
OMORI has a very distinct and unique art style. As such it can be tricky to replicate it to perfection. The process the original team used was to draw it digitally using a pencil-like brush, print it out, scan it to a computer, and make any finishing touches digitally. Understandably not everyone has access to those materials, so simply drawing it digitally can get you close enough. You don't even have to get the style completely right lol.

As for implementing the custom art into your game, it is one of the easiest things you can do. So let's get started!

Making portraits

All portraits have the dimensions of 106 x 106 pixels. For dialogue, they are loaded individually through a .yml file, so you are free to make your own sprite sheets and call on them through there (in fact, it's better in every way possible so do it)

The faceindex values start from 0, and go left to right, top to bottom.



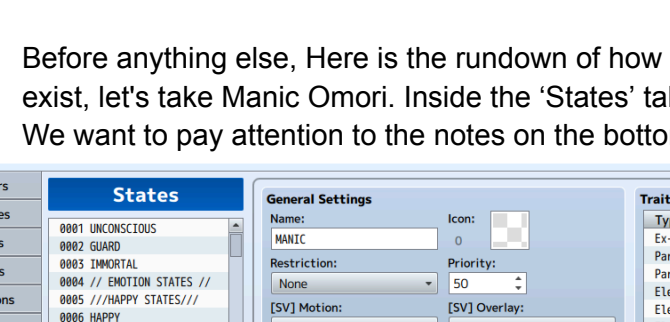
Credit: Cooldry77-T1-RTD

Note how different sprite sheets follow a certain number of sprites per row depending on what they're used for. In the case of dialogue, they go in fours.

Battle Portraits

The battle portraits follow a template, depending on the character. For these you'll most likely have to replace them as it is easier than coding it in to load in other faces.
In order to get the three frames for the battle, set the first face at ~50% opacity and trace over it on a new layer. Move the new face to the appropriate spot when done.

The confetti used for the DW victory faces can be found here. Just lay it over top of them.

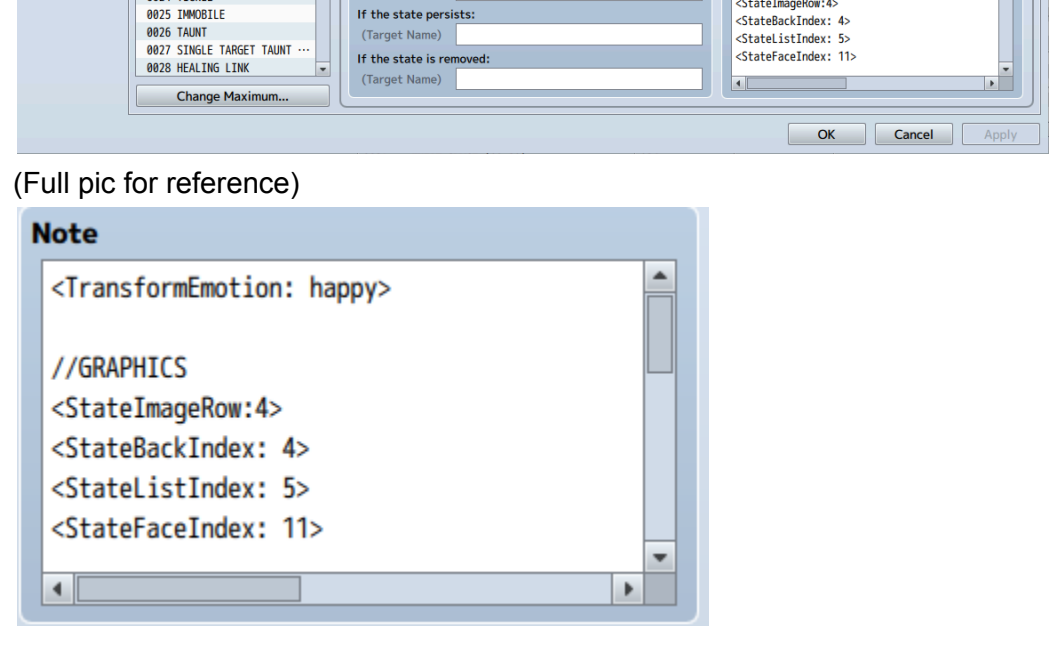


Making More Battle Portraits

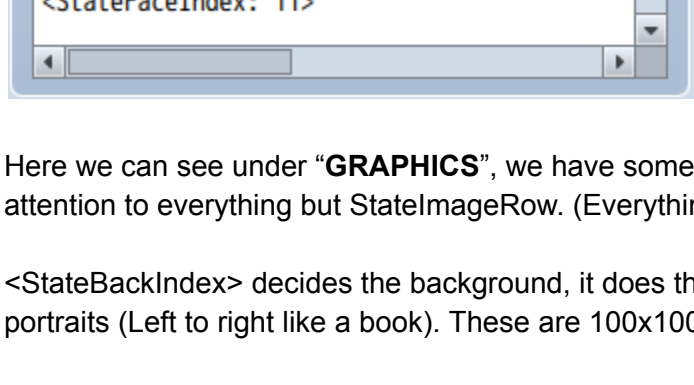
So you know how to make a portrait and replace battle portraits, but what if you wanted To make an extra face completely? Maybe for a new emotion or status effect? This Is how you do it. In this example, I will be making a new status for Hero, this will use Visual part of the process.

First things first, What can you change? Well, you can change the face itself, the Background behind the face, and the label above the face (The thing that says the emotion).

Before anything else, Here is the rundown of how it works with something that already exist, let's take Manic Omori. Inside the 'States' tab is the 'Manic' state. In said state, We want to pay attention to the notes on the bottom right



(Full pic for reference)



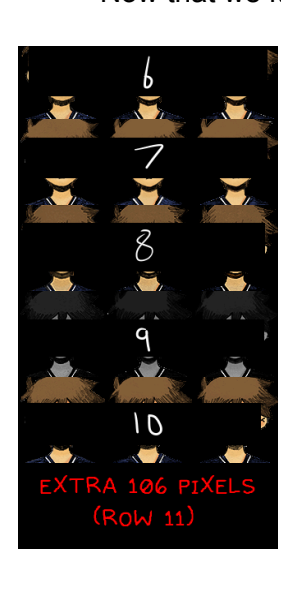
Here we can see under "**GRAPHICS**", we have some statements. We only need to really pay attention to everything but StateImageRow. (Everything counts from 0.)

<StateBackIndex> decides the background, it does this in the exact same manner as talking portraits (Left to right like a book). These are 100x100 pixels each

<StateListIndex> decides the label above the character, in this case "**MANIC**". This sheet just goes down vertically. These are 134x24 pixels each.

<StateFaceIndex> decides the face, these are decided in **ROWS** so this is row 11 that is being used, there are 3 faces per row for animation. These are all in 106x106 pixels.

Now that we know how this works generally, we will make our own. We will make a state Called "**Problem?**"



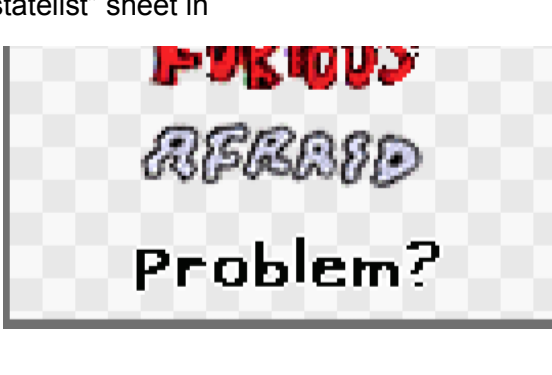
Let us start with the **face**, unlike the process with dialogue portraits, we **DO NOT** make a separate sheet, this will not work. Instead we make the existing sheet longer. So if you Wanted to make one more face, then you extend the sheet downwards by 106-pixels. In Heros' case, this takes us from rows 0-10, to 0-11.

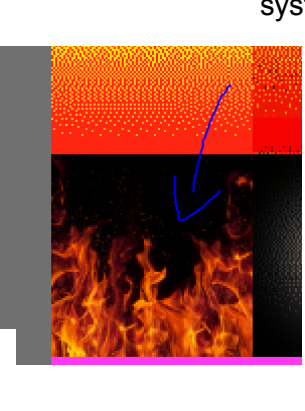
Next is adding in your drawings, make sure to be careful to centre them so they animate properly. We will be using Troll face because it's funny

These go into the new row you added.

In the notes section, we write <StateFaceIndex: 11>

Next we will make a "statelist" sheet in label. We make the system longer by 24-pixels.





Then we make our label. <StateListIndex: 13>

Finally, we have a portrait background. You want to remove the pink from the very next square in the order and put something there.
<StateBackIndex: 12>

We here we see the fruits of our labour

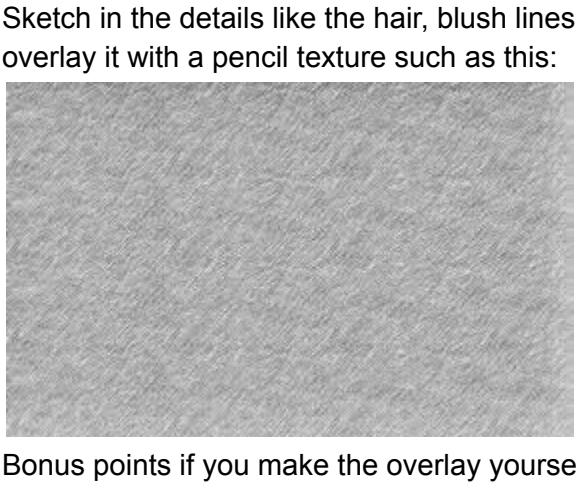
We have just completed making a new portrait!

Art Tips!

Only Use 100% opacity when drawing. Anything below 100 will not show at all.

Start with a base to go off of for your various emotions. If you are inexperienced, you can trace off other characters.

For DW characters, use various shades of blue, as well as a brush like the pencil. Sketch in the details like the hair, blush lines, and the eyes with a darker shade. Then, overlay it with a pencil texture such as this:



Bonus points if you make the overlay yourself :). Here are a couple examples that follow these steps to some capacity:

Credit: Labrallie, june

For RW characters, use the pen tool for cleaner lines and a more "realistic" look. Colors are up to you, but don't make them too bright or vibrant. This is reality after all, and it's gloomy as heck. Also use shading, blush, and shadows. Those can be done by coloring with the overlay layer.

Credit: Gum

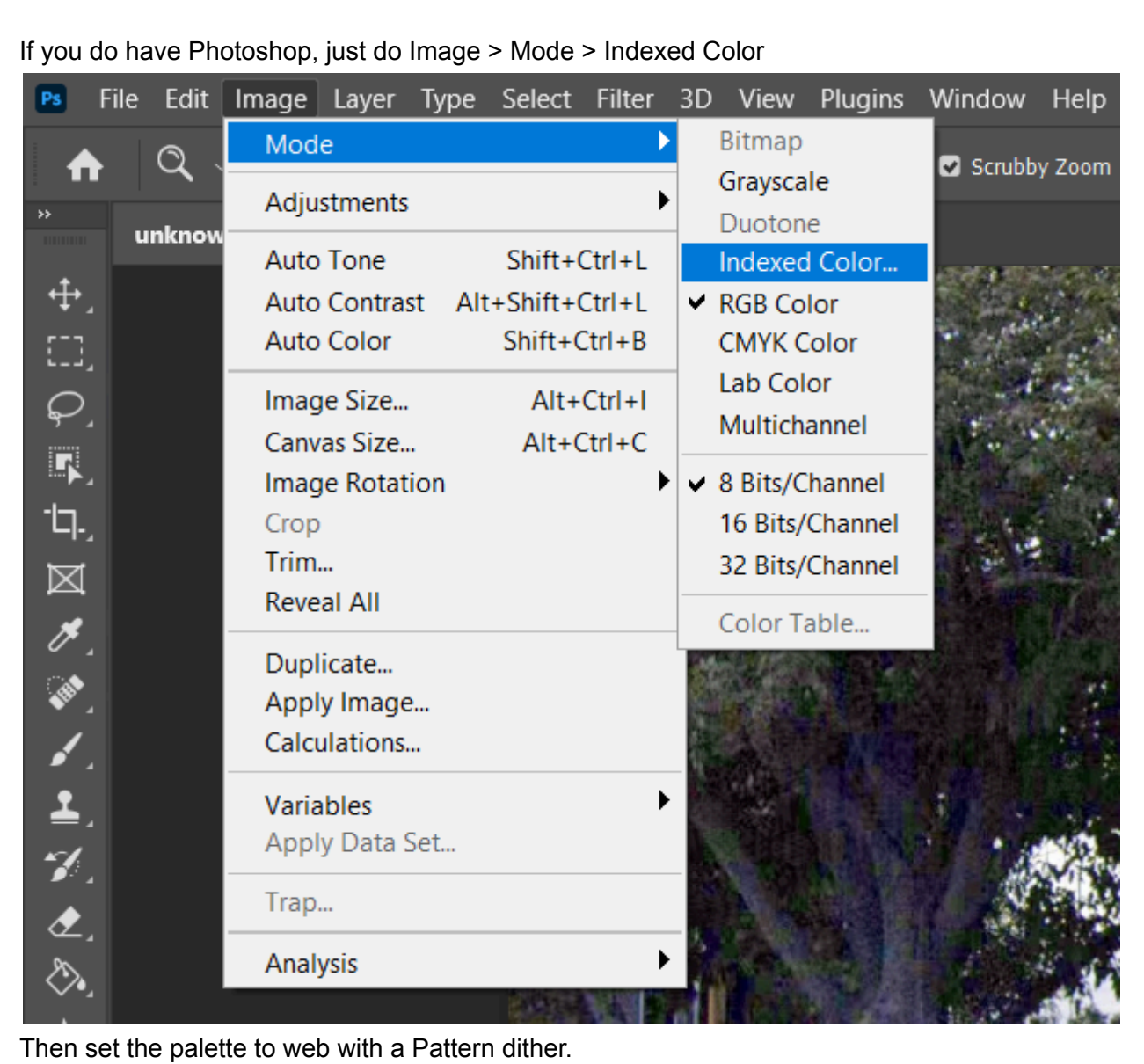
If you want to make enemy sprites, here is a special step-by-step guide on how to do it: https://www.reddit.com/r/OMORI/comments/neb2uo/omori_artstyle_guide/

Custom Battle Backgrounds

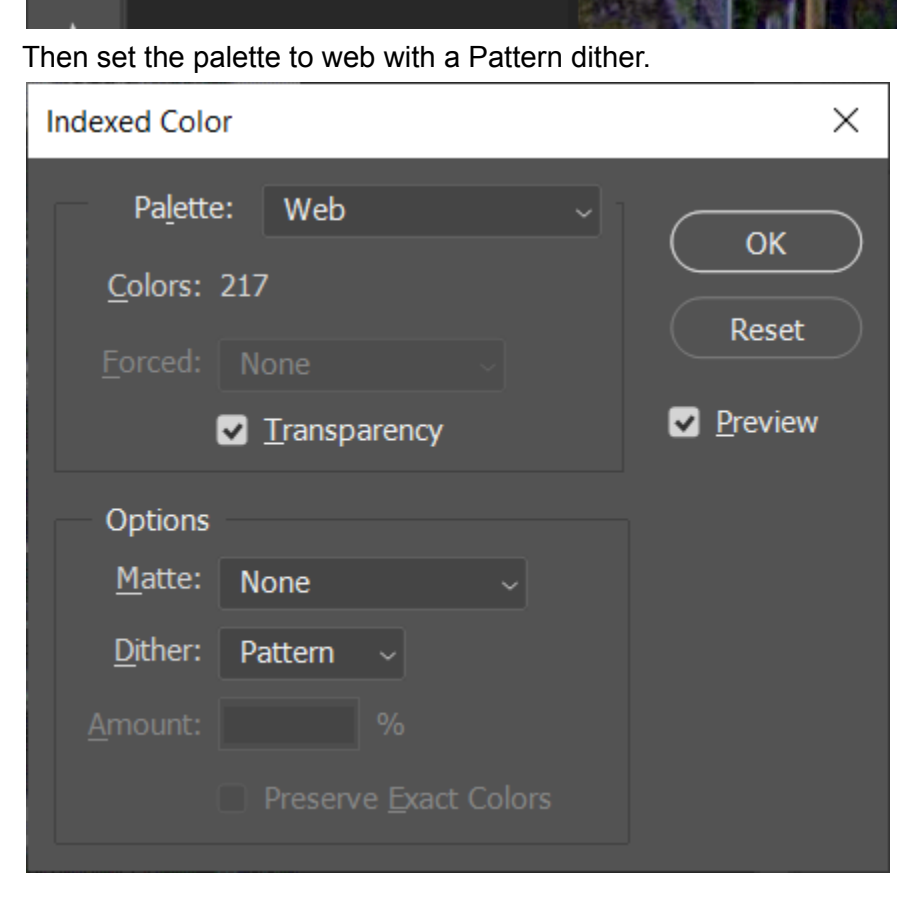
Making custom battle backgrounds is a very easy and fun process! First step is to find an image that you want. The original dev team used Pexels and Pixabay, as they have high quality images that are free to use. Next, resize and crop the image down to 640x480 pixels. Edit it to your liking, and save it as a png. You then have two options. Photoshop will give you the most accurate result, but it's pricey.

If you don't have Photoshop, then put the cropped image through a dithering tool (https://ditherit.com is a good one).

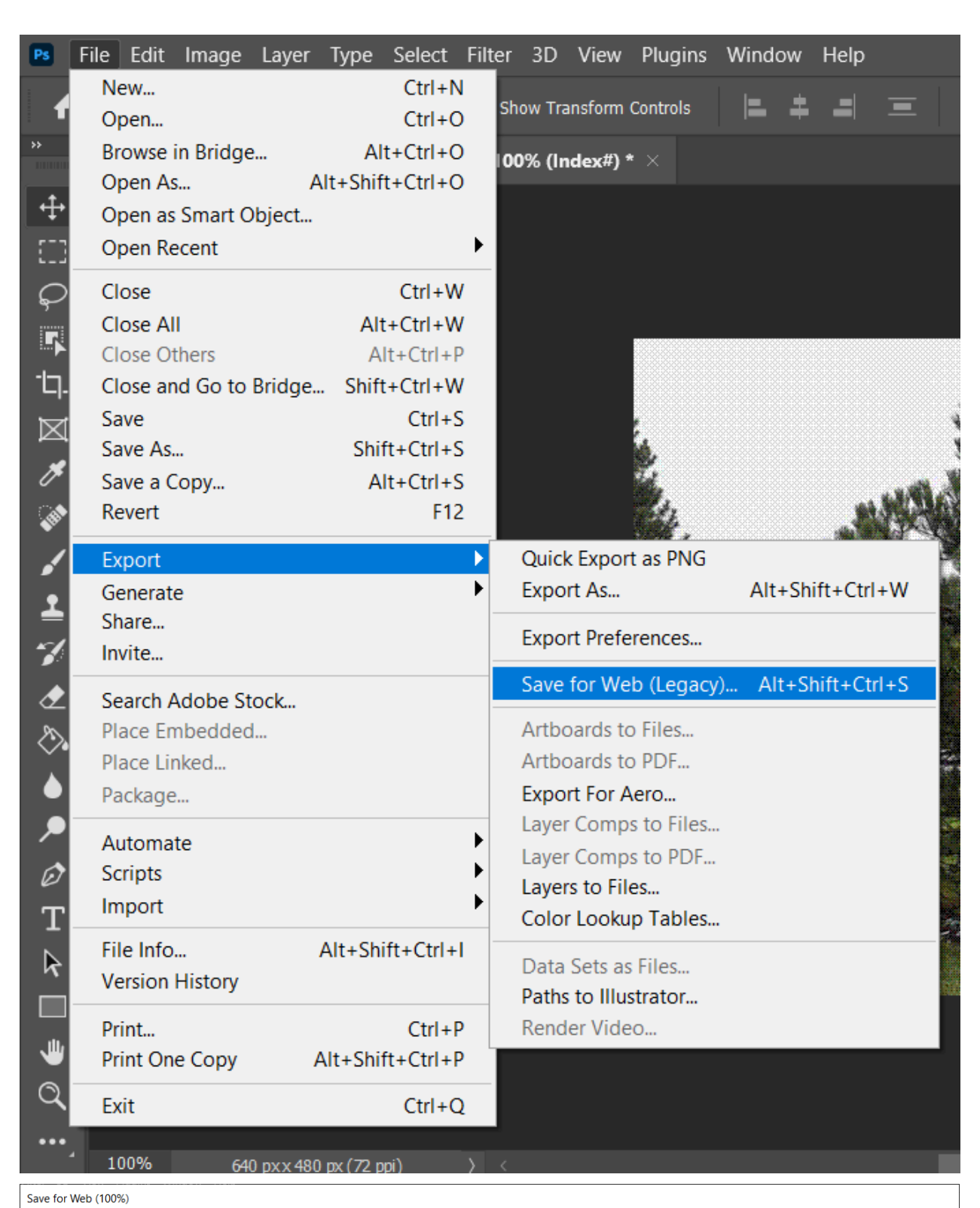
If you do have Photoshop, just do Image > Mode > Indexed Color

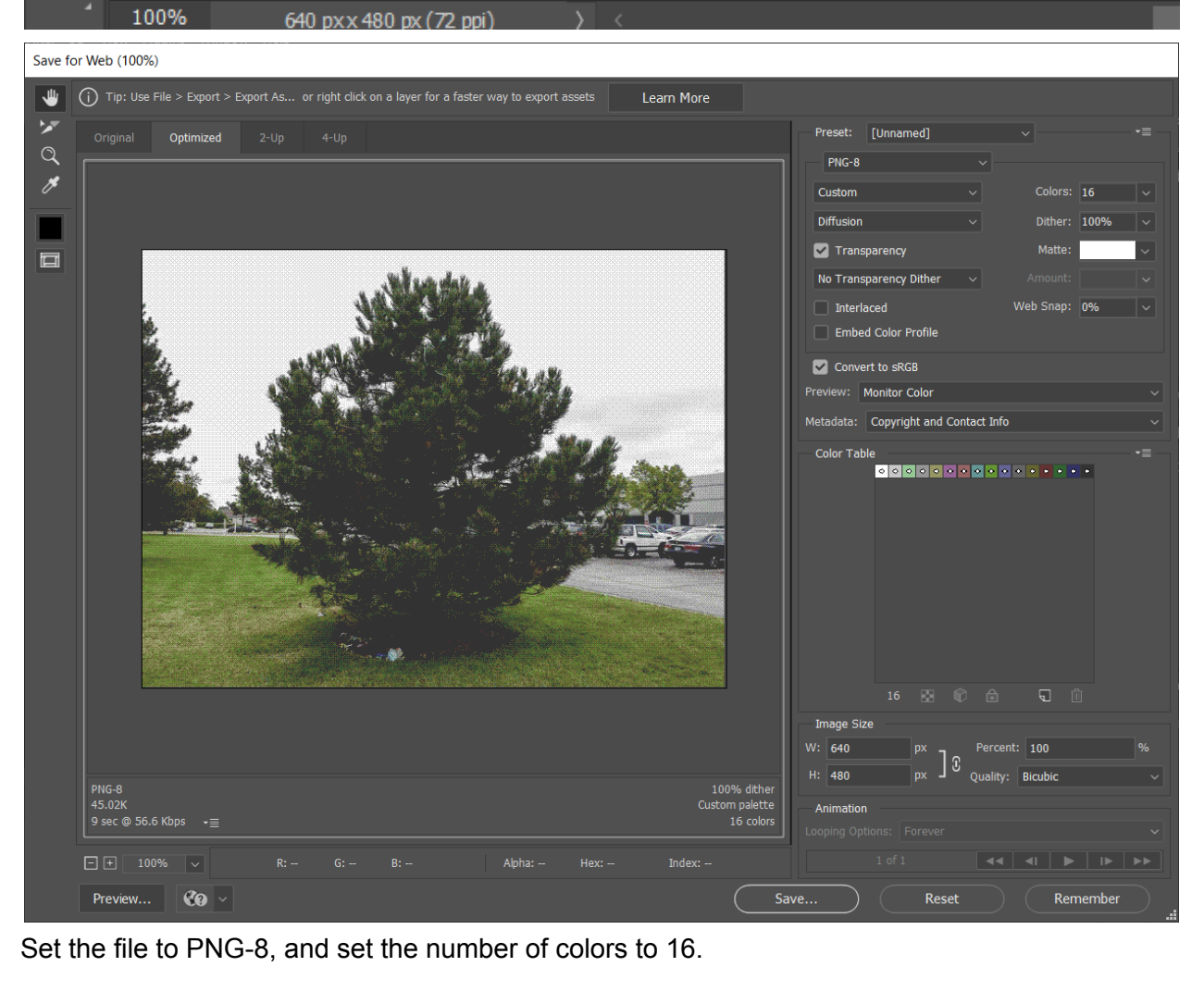


Then set the palette to web with a Pattern dither.



Then, go to File > Export > Save for Web (Legacy)





Set the file to PNG-8, and set the number of colors to 16.

You can then save your result under img/battlebacks1, and voila! You have your new battleback

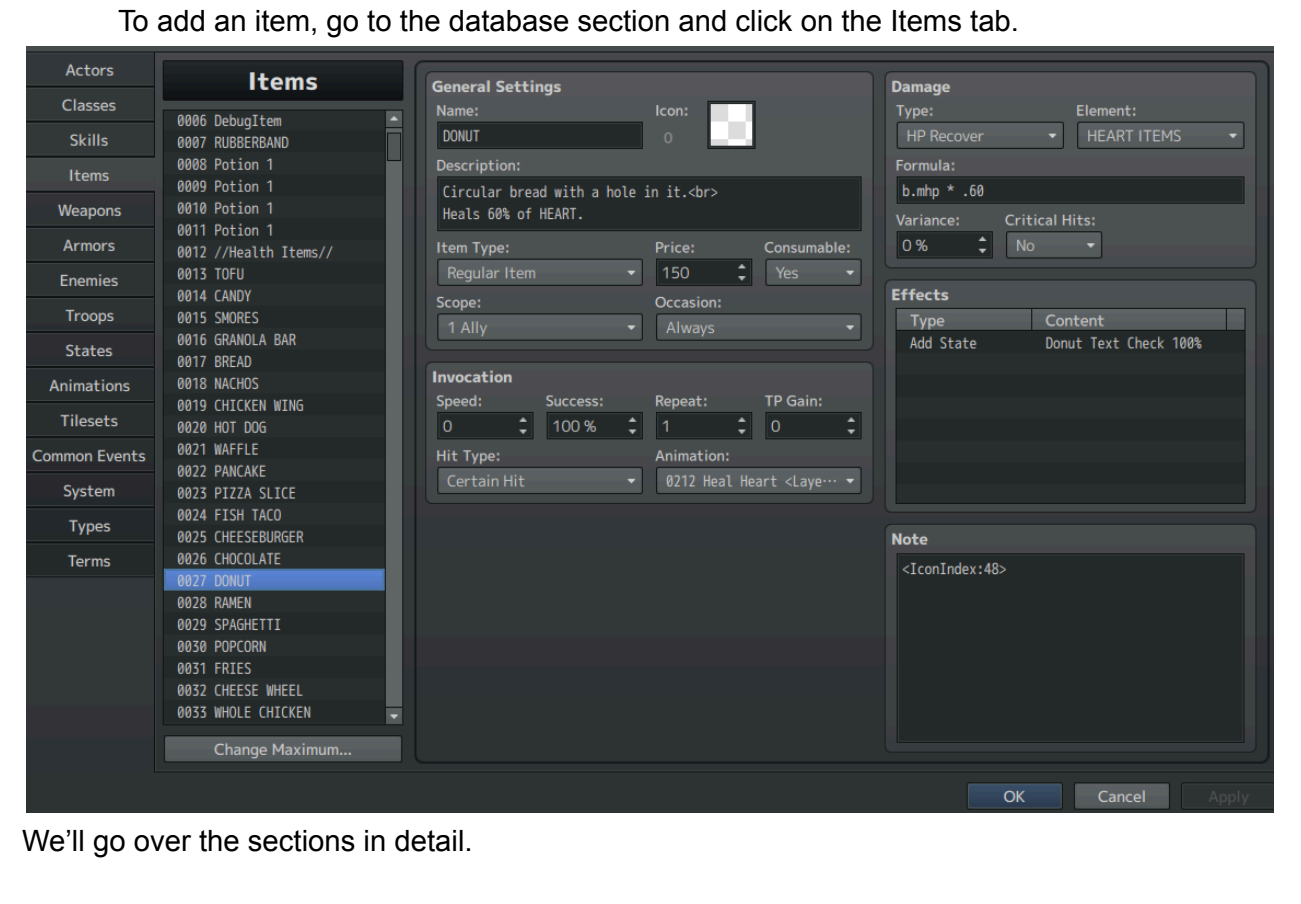
Credit: Labrallie

Credit: dawn

Adding Items

Giving the Item Function

To add an item, go to the database section and click on the Items tab.



We'll go over the sections in detail.

General Settings:

Name and description are straightforward. Just type in what you want, but make sure to include line breaks so the text doesn't get hidden behind the item icon. The Item Type will always either be Regular Item or Key Item. Use Regular Item for toys and snacks, and use Key Items for quests. The price will determine how much the item will cost in shops, and Consumable should be set to yes for toys and snacks, and no for quest items.

Invocation:

The only thing that you should input yourself is the animation that plays for it, depending on what you are making. Leave everything else as-is.

Damage:

Here you can determine what exactly the item will do. If they are quest items, don't worry about this. HP Recover type items will always be a HEART ITEMS element, and MP Recover type items will always be a JUICE ITEMS element. HP and MP damage will always be Physical. You should have no variance, and no critical hits. You can then determine the formula. "**b.mhp**" is the target's health, and "**b.mmp**" is the target's juice.

Effects:

Typically this isn't used unless it's for scenarios like your Stats being raised by a Sno-Cone (see the states tab for how that works), or a donut text check apparently. Most of it is done in the Notes section.

Notes:

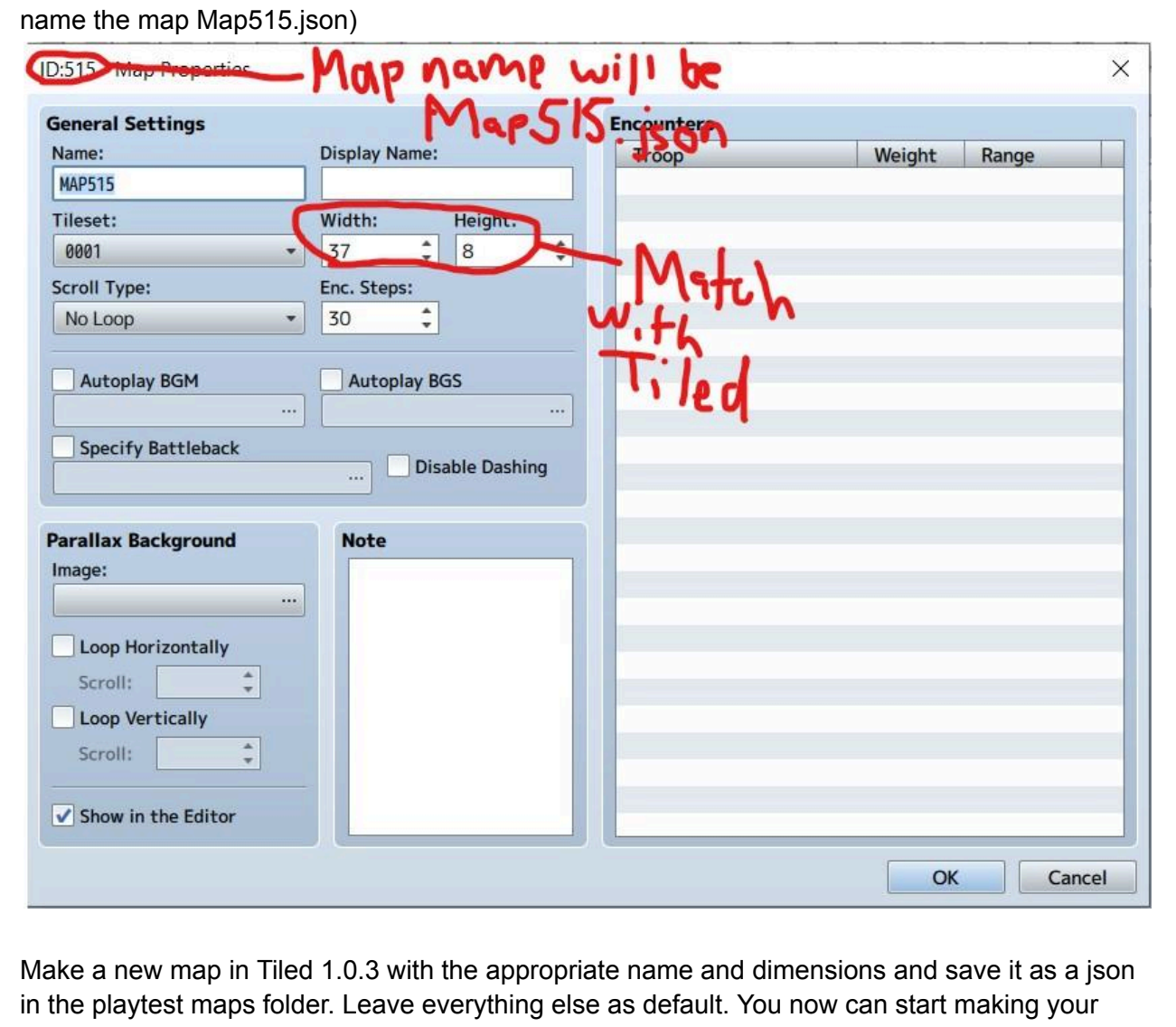
Hey, hey. Thought you could run from it, but JavaScript runs faster. Here you can set the icon shown in the menu, through <IconIndex:xx>. The sprite sheets are under img/system/, and they are labeled as itemImportant for quest items, and itemConsumables for toys and snacks. If the item is a toy, put <is Toy> above the Icon Index, and <AntiFail> above that if said toy can't be used in battle. Then, you can do a variety of things, such as play animations, set damage formulas, and display battle text. If you're like me and don't know JS, I'd just recommend copying code from a similar item and changing it to your needs.

Making Maps

Video Tutorial: https://youtu.be/pstj8qSbM0g

Setting Up the Map

Open up the playtest folder and click the RPG Project file. Make a new map and take a note of its Map ID and size. The Map ID will determine the Map name (e.g. Map ID is 515, so name the map Map515.json)



Make a new map in Tiled 1.0.3 with the appropriate name and dimensions and save it as a json in the playtest maps folder. Leave everything else as default. You now can start making your map!

Layering

In order to make your maps look less like cave paintings, you'll need to have multiple layers. For OMORI, each of these layers save for collision and region layers (which we'll get into) will have two custom properties: zIndex and priority. Priority tells RPG Maker which layer to load in first. The bottom layer will have priority of 1, and it will continue upwards until the zIndex changes, which it will then go back to 1.

Custom Properties	
priority	4
zIndex	1

OMORI typically uses 3 different zIndexes:
1 - GROUND
3 - SAME AS PLAYER
5 - ABOVE ALL

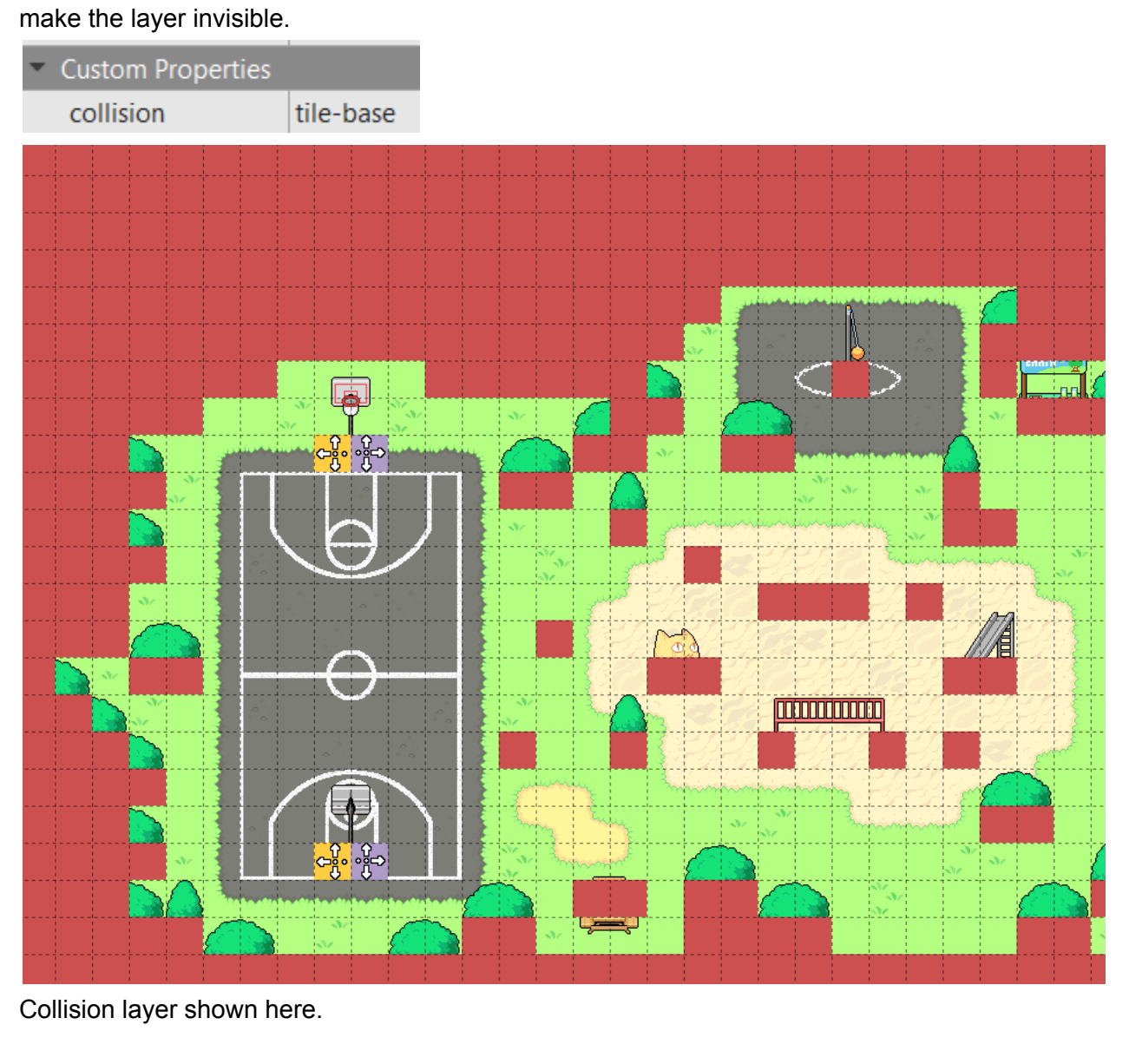
There is also this quick guide on zIndexes should you choose to have more than these:

```
* [Z coordinate]
*  0 : Lower tiles
*  1 : Lower characters
*  3 : Normal characters
*  4 : Upper tiles
*  5 : Upper characters
*  6 : Airship shadow
*  7 : Balloon
*  8 : Animation
*  9 : Destination
```

OMORI also occasionally uses object layers. These have the same custom properties as the tile layers, but can also support object movement and scaling. These are used for objects that require more than one tile and are used more than once.

You'll also have to set up a Collision Layer. This will always be directly above the Tile and Object Layers. Set up a custom property called collision, and set the value to "tile-base". You can then use the special collision tileset for setting up collisions in your map. I'd suggest setting the opacity to 0.5 so you can join see what you are covering up with the tiles. Once you are done, make the layer invisible.

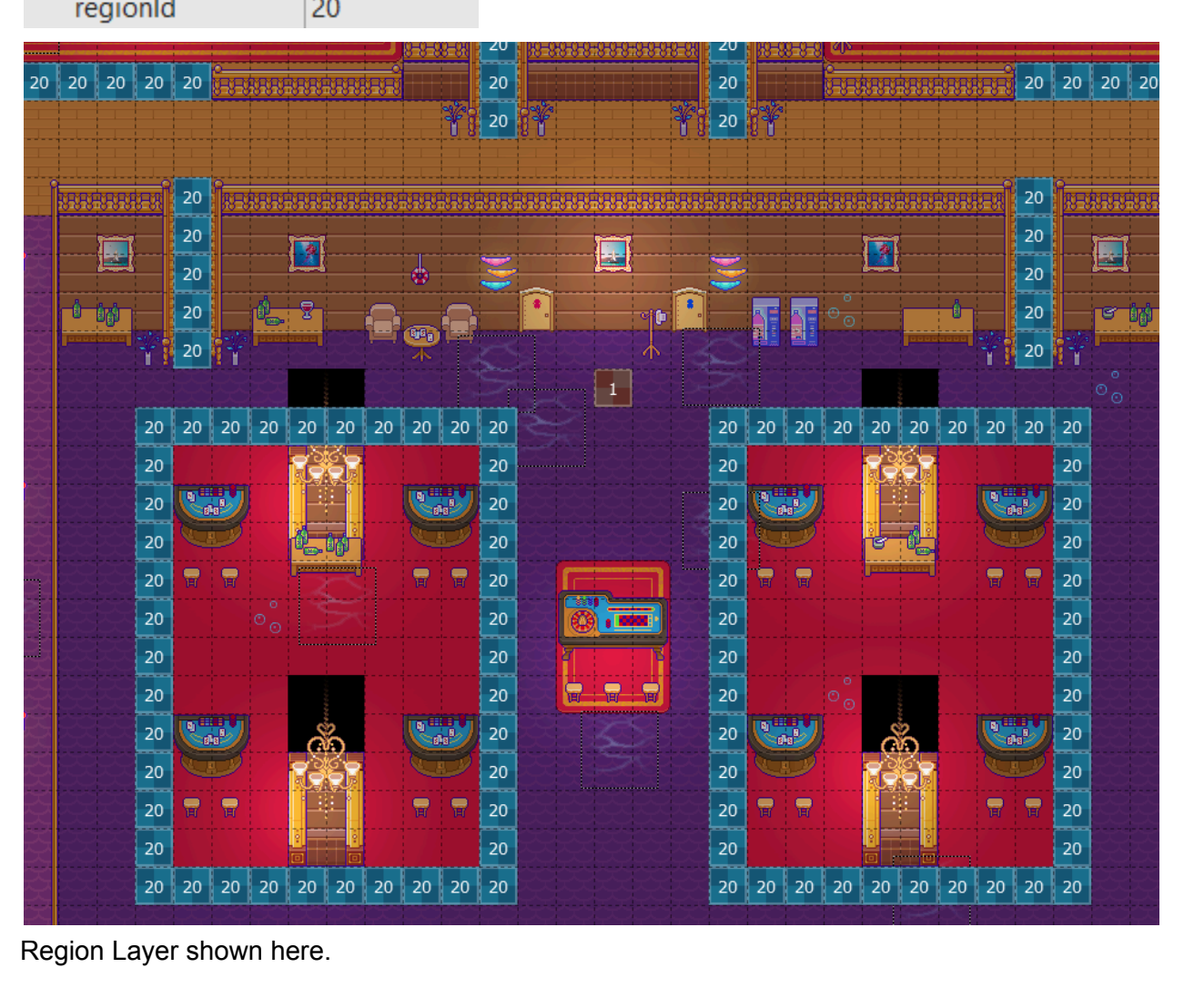
Custom Properties	
collision	tile-base



Collision layer shown here.

Region Layers can be used as boundaries for enemies. They require a custom property called "regionId", and use the special region tile set. Set the value of that property to whatever region tile you are using. Do that for each Region Layer. These will always be the top layer and will be invisible as well.

Custom Properties	
regionId	20



Region Layer shown here.

Creating a Tileset to Use

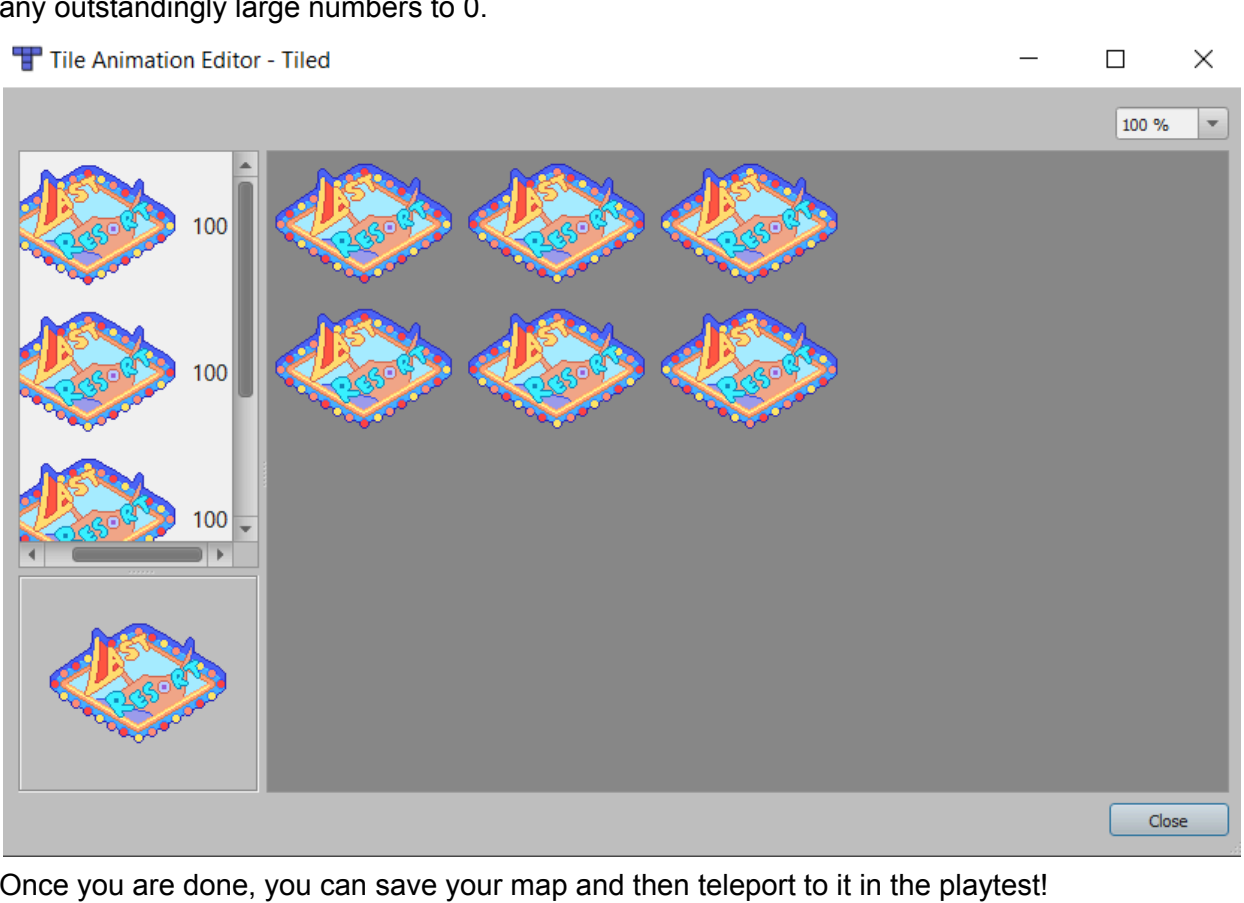
Tilesets are also loaded as json files. In order to import them into the game, you must first create a tileset. The tiles must be 32x32 each, with no spacing in between them. Then, save your tileset in the playtest folder as a png, under img/tilesets. Create a new tileset through Tiled based on Tileset Image, and save it as a json file in the maps folder.

Animating Tiles

Animating tiles is a great way to liven up your maps, and OMORI has no shortage of them! To animate tiles, open up your tileset json file in Tiled, then click on the tile you want to animate and go down to the Tileset dropdown menu. Open up the Tile Animation Editor. Through there you can select the frames of said tile through double clicking the frames and set how long each frame will last in milliseconds. Then, save the json file when you're done!

THIS IS IMPORTANT, DO NOT ROTATE YOUR TILES AND PLACE THEM. IT WILL NOT LOAD IN THE GAME.

If you believe you have rotated tiles in the map, open the json file in a regular text editor and set any outstandingly large numbers to 0.



Once you are done, you can save your map and then teleport to it in the playtest!

Making Brand New Battles

All examples used in this section are from THE BASEMENT

Special thanks to sunny#0007 and DevaliousL#3317, they taught me some of the stuff here!

Now that you have your custom skills, you need to apply them somewhere! What better place than your very own battle? In order to do this, you will need a copy of RPG Maker MV!

-**Prerequisites**

These are things you will need before opening RPG Maker.

a. **Battle Background [img/battlebacks1/]**

You will need your battle background ready and at hand. Refer back to earlier in this document to see how exactly you should be making your battle backgrounds fit in with OMORI.

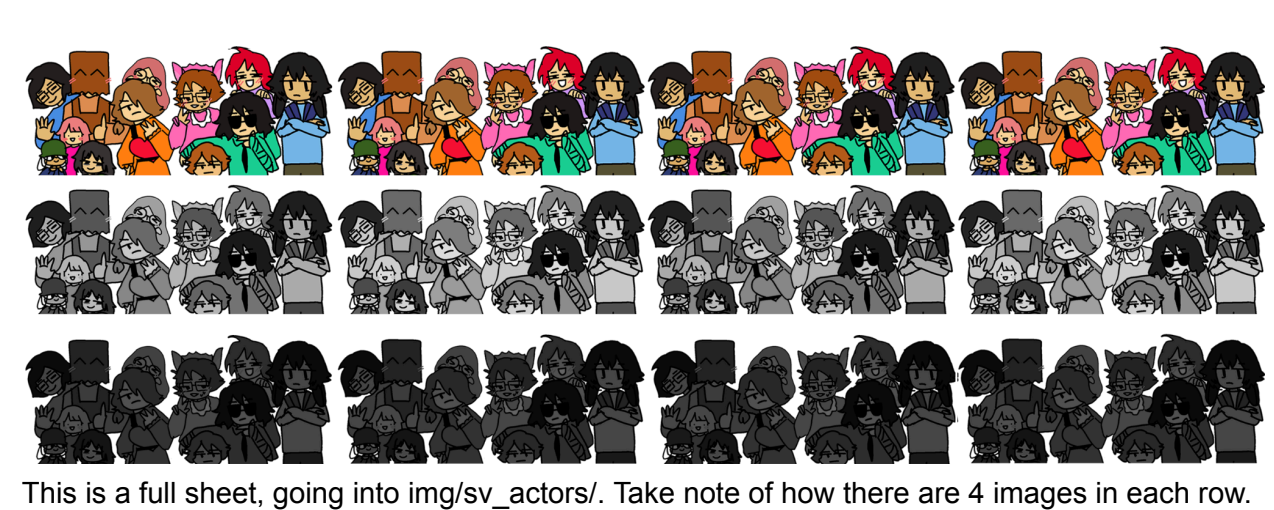
b. **Character(s)/Sprite(s) [img/characters/]**

Your battle needs to be tied to some sort of character or sprite. You'll need to have created your sprite sheet by this point. A tip with these are copying another sprite sheet straight from the game and drawing over it. A great program for this is Aseprite.

c. **In-Battle Character(s) [img/enemies/ , img/sv_actors/]**

This is the drawing that you'll see in-battle. Within img/enemies/ should be a singular picture. In img/sv_actors/, there should be the full sheet, both of these are pictured below.

This is a singular image, which would be put into img/enemies/. Art by chronostat#0007.



This is a full sheet, going into img/sv_actors/. Take note of how there are 4 images in each row.

Both of these images should be named the same, starting with an exclamation point. For both of these, I named them !battle_the_basement, fitting with the rest of the battle image formats.

d. Game Text

You'll need the in and outside of battle text ready and in .yml files. I suggest making 2 brand new files, one for overworld text, and one for in-battle text. Name them something you'll remember, such as "the_basement_battle_text.yml"

All text must be in the following format!

message_0:
    text:

If you want someone's name to be there, you would put:

message_0:
    text: \n<Insert Name Here>

Here's an example:

```
message_0:
    text: \n<AYAAN>Hello OMORI Modders!

message_1:
    text: \n<AYAAN>This is AYAAN here, showing you how text files work!

message_2:
    text: \n<AYAAN>Make sure to follow this format!
```

Once you have all of this, compile it into a mod and put it into your mods folder, along with Rph's Playtester Maker mod, and those new files will now be in the decrypted www_playtest folder. Now you are ready to open up RPG Maker.

-Enemies Tab

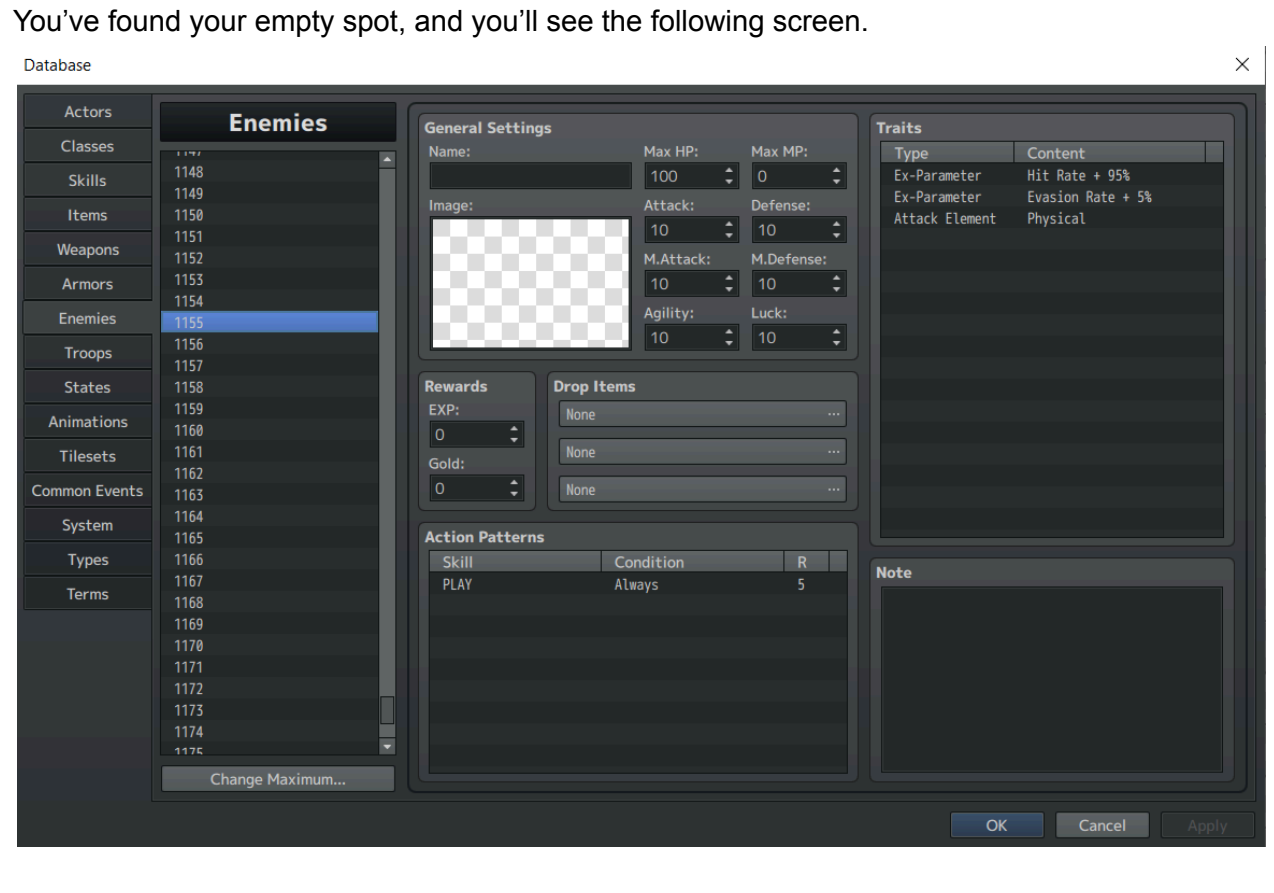
Once you've opened up RPG Maker, click on the gear icon at the top of the screen. You'll be greeted with the following screen:

If you notice, 7-9 on the list of enemies are empty. These empty spaces are where you'd go in order to work on your enemy. However, it'd be much better if you scrolled to the bottom and took one of the many empty spots in there, as, depending on what you're doing with your boss, you'll need 4 spots total.

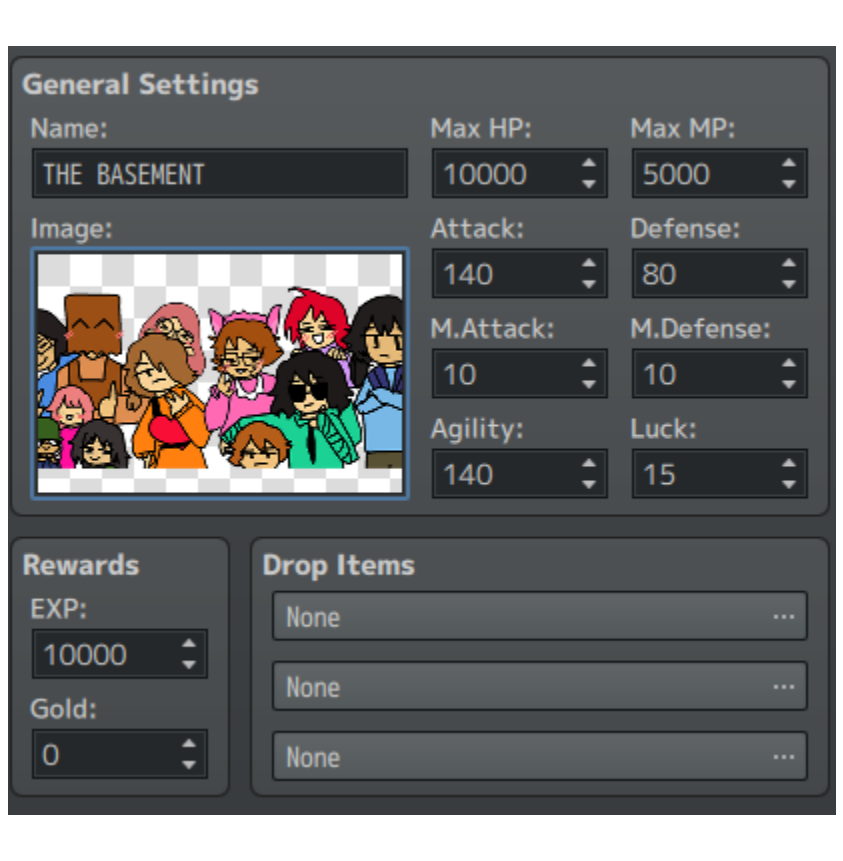
Note: Action Patterns tab isn't normally used, what's done there (skills) is done in the Note tab.

a. General Settings

You've found your empty spot, and you'll see the following screen.



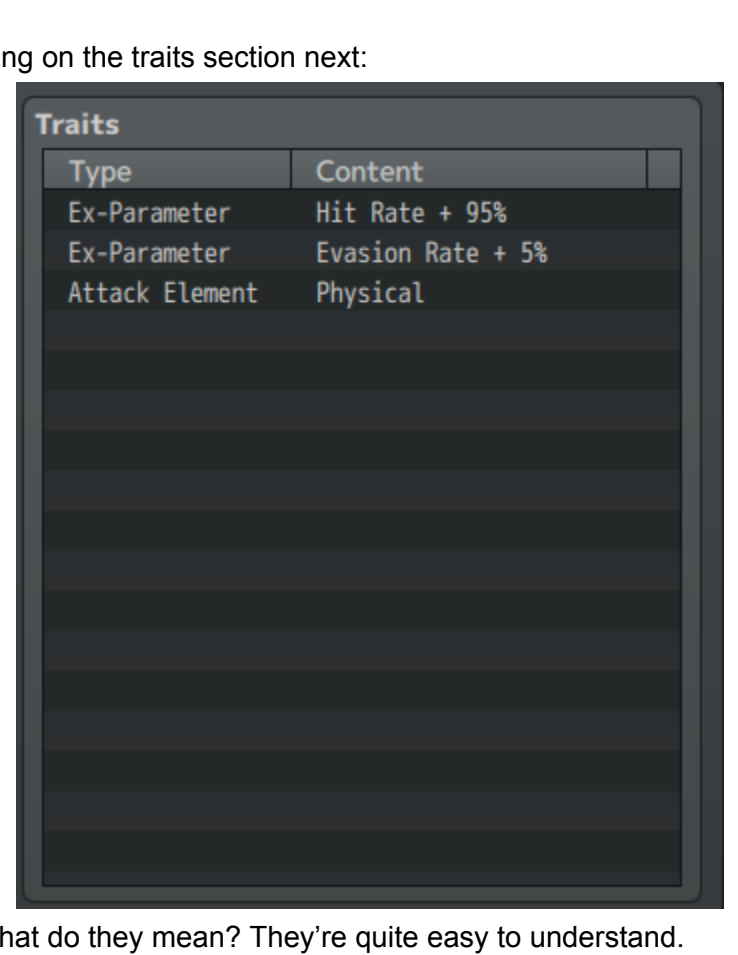
Let's focus on just the middle area, labeled as "General Settings". Here you can change the HP, juice, attack power, defense, etc. You can make these whatever your heart desires. At the image section, you can double click the box and choose one of the enemy images, which are taken from img/enemies/. Find the image you placed before, and select it. After completing all of this, you'll have something like this:



Congrats, you've finished one of the easier portions of this process, give yourself a pat on the back!

b. Traits

We'll be concentrating on the traits section next:



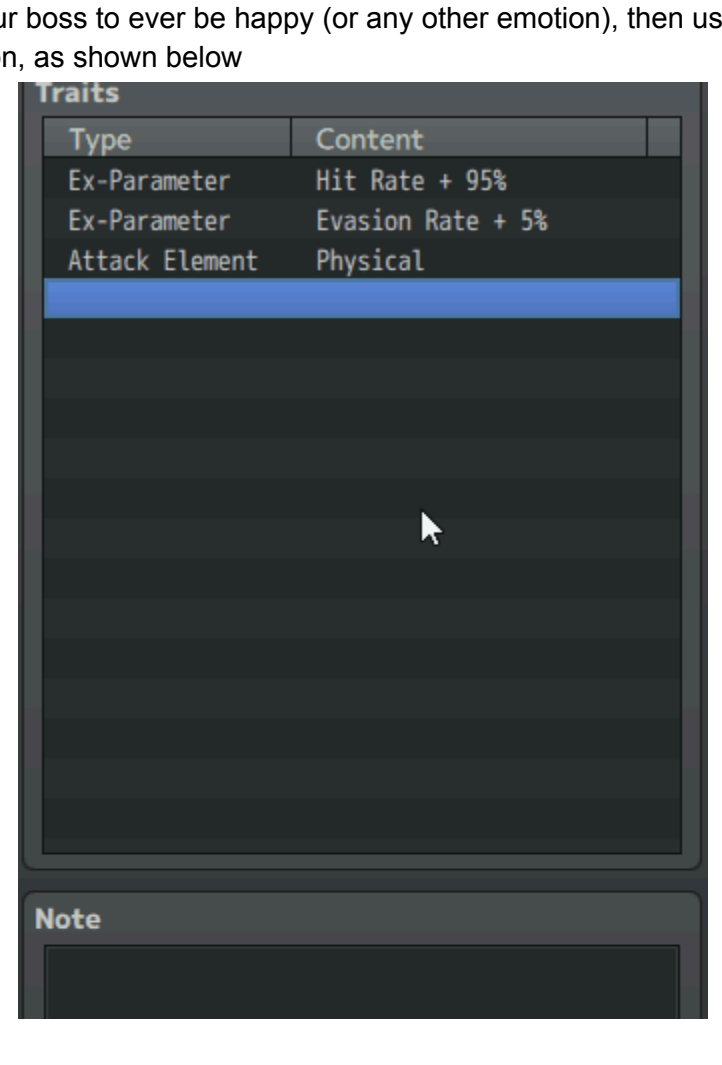
Funky words, but what do they mean? They're quite easy to understand.

Ex-Parameter - Hit Rate, is how often the attacks the boss does will land.
Ex-Parameter - Evasion Rate, is how often you will miss an attack, otherwise known as the boss dodging from it, hence the usage of the word "evasion".
Attack Element - Physical tells you how the boss attacks.

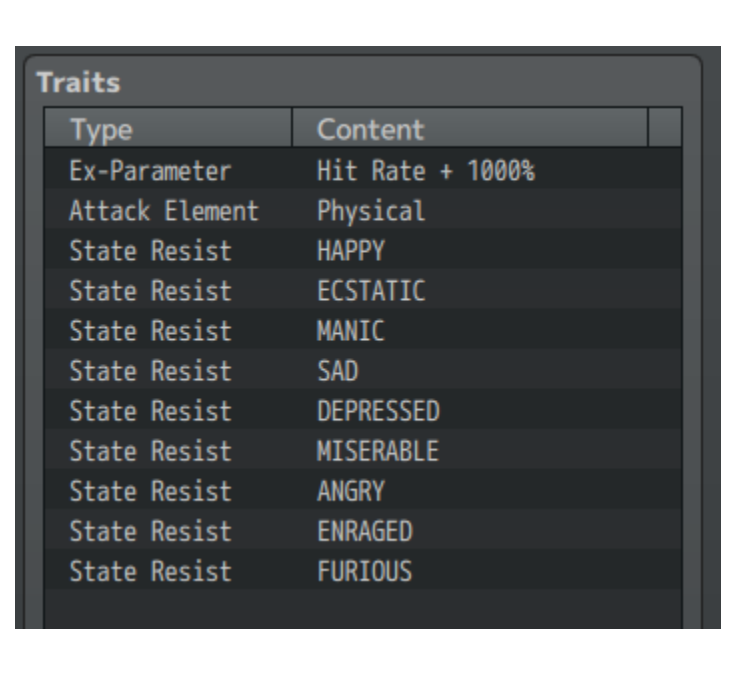
These are the basic things you'll need, and they're already there for you! There's still more for this tab, however.

The final thing you'll need to know is State Resist. Bosses will tend to go down a certain emotion route, and won't be other ones. You can't change their path. For example, SWEETHEART goes down the happy emotion scale. You can't make her angry, and this is done by State Resist.

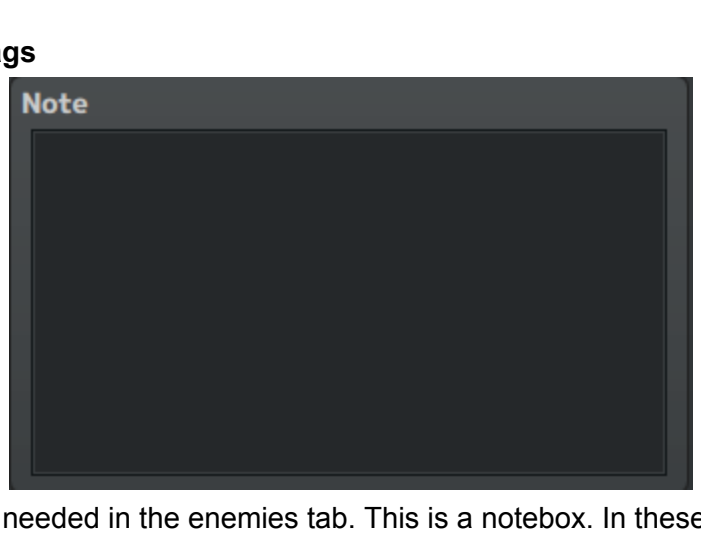
If you don't want your boss to ever be happy (or any other emotion), then use State Resist, and set down the emotion, as shown below



These are all the Traits you'll be needing in your boss fights. Here's a complete list:



-Notes and Notetags



This is the final thing needed in the enemies tab. This is a notebox. In these, you can put in notes about whatever you're making, and you can also <Put Notetags> just like that. Notetags are little bits of commands that tell RPG Maker what the enemy should be doing.

There are a couple of notetags that you'll be seeing around, and we'll be going through each of them.

Important note: if an "x", "y", and/or "z" is used in the notetags in this document, that signals a variable that you yourself would have to put in.

a. <TransformBaseId: x>

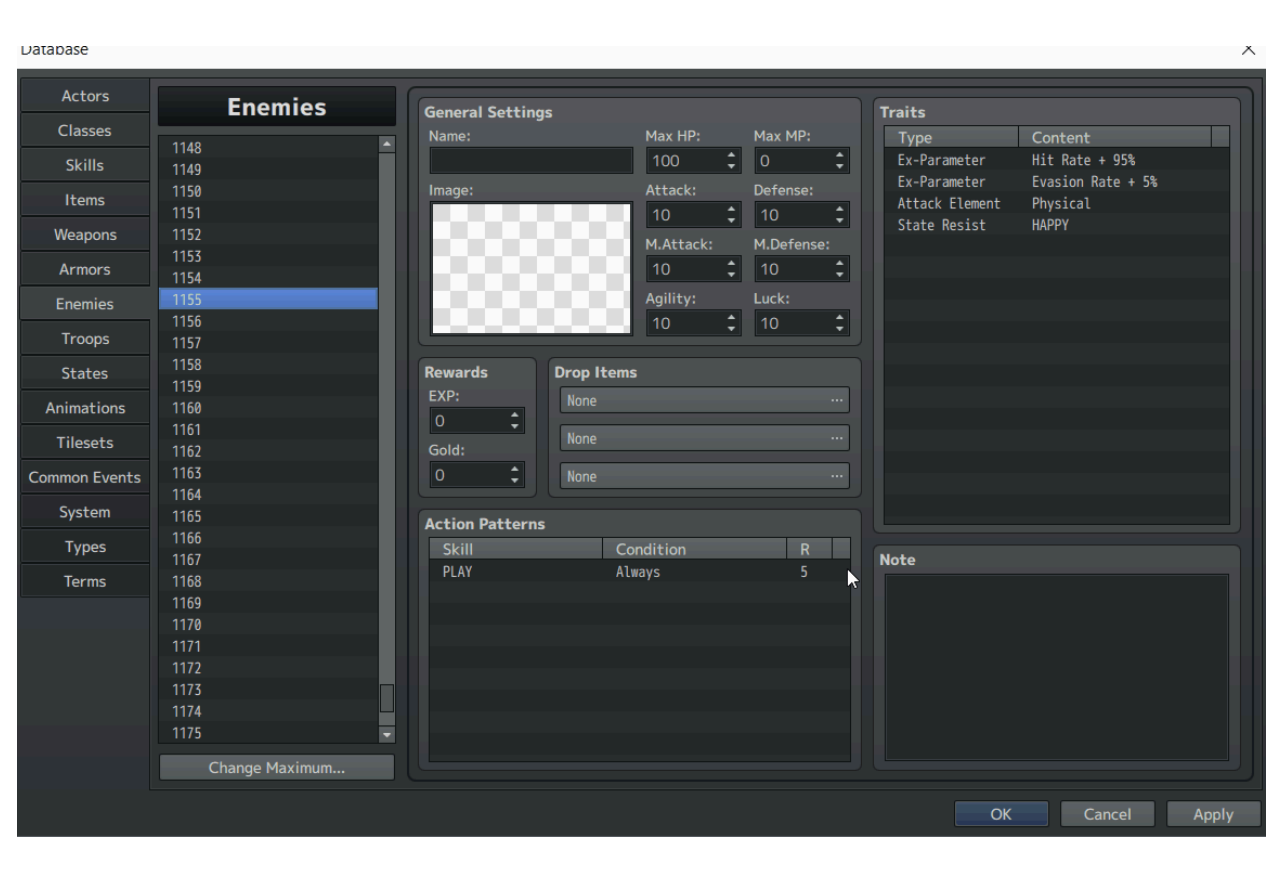
This notetag is used at the top of most notes. The Base ID is what enemy number you chose. Since I chose 1155, my notetag would be:

<TransformBaseId: 1155>

This helps make sure that all 4 enemies that you'll have later on are all tied together into 1.

b. <PassiveState: x>

This controls what state your "neutral" emotion is for your boss. If you want your boss to passively be something in the State's tab on the side, find what you want, remember what number it is, go back to the notetag, and input the number as your passive state, as shown below. Many bosses don't use this, it's not required



c. <Static Level: 1>

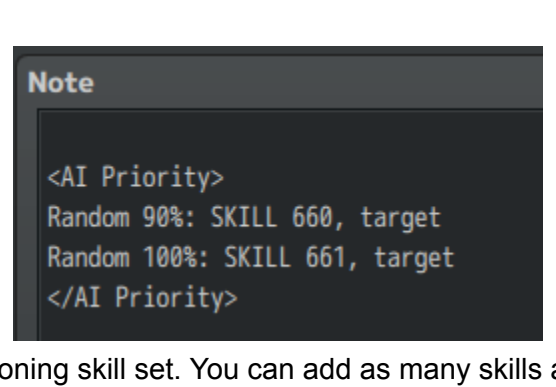
This is the enemy starting level, and should not be anything other than 1.

d. <AI Level: 100>

This is how smart your enemy is, all enemies in OMORI use 100. This is more specifically the chance the enemy would follow the specified skill order used in <AI Priority>. If it doesn't, it casts random skills it's allowed to in the base engine. (Yanfly)

e. <AI Priority>
Random x%: SKILL y, target
</AI Priority>

These are the skills assigned to your boss! If you want to give your boss a skill that you made (or an already made one) then go to the skills section on the side bar, find the skill number, and input that number in place of y. You can add as many skills as you'd like to. Variable x in this case is how likely the attack will happen. If you want your attack to have a 25% chance of being used, then variable x would be 25.



This is a fully functioning skill set. You can add as many skills as you want!

Note: You'll be seeing a lot of the words "Sideview" and "Sideview Battler". These notetags all come from the same plugin, I've split them up into easier, manageable, parts. A complete version of all these notetags together will be farther down.

f. <Sideview Battler: x>
<Sideview Battler Frames: y>
<Sideview Battler Speed: z>

These notetags are VERY important in a battle. This is where your img/sv_actors/ sheet will come in handy. Whatever you named that sheet, that is what you put in x.

For variable y, have you noticed in OMORI when you're fighting a boss, they'll be moving ever so slightly? This is what controls that. Almost all battles, if not all of them, use 4 frames. This is why the sheets will have 4 images in each row.

Finally, for variable z, this is how fast the boss "moves". Most, if not all, battles use 12 as their speed.

g. <Sideview Battler Size: x, y>
<Sideview Width: 0>
<Sideview Height: 0>
<Position Offset Y: 0>
<StatusYOffset: z>

For variables x and y, this determines how large 1 frame is. The easiest way to do this is to look at the dimensions of your enemy. Variable z is where your first frame starts. You can take you sv_actors image and divide the y variable by however many rows there are, and that number will be around what you need.

h. <Sideview Battler Motion>
Name: x
Index: y
Loop
</Sideview Battler Motion>

This notetag is repeated quite a bit of times in a singular note. We'll go through what each name means, as well as what the index is.

For basics, the name refers to what the image is, such as Walk being emotion.
Index is where on your sheet the game should look at. The top of your sheet is Index 0, next is 1, and so on, as shown below.

My index 0 is my resting/neutral image. Index 1 is what they look like when they're taking damage. Index 2 is what they look like when they're successfully killed.

The format that majority of OMORI follows is index 0 as resting/neutral, 1 as taking damage, 2 as the death scene, and 3+ as emotions.

The following is each name and what they are used for.

Name: Walk/Thrust/Guard/Skill/Other/Item/Spell
Purpose: All of these handle emotions. For example, if you want your enemy to be angry, you would have the index at whichever row you need, and you'd put it in all of these categories.

No name
Purpose: Normal image. This should be fully colored, normal art. This is normally at the top of your sv_actors sheet, which is normally index 0.

Name: Damage
Purpose: The taking damage animation that plays whenever an enemy gets hit. This is normally at index 1.

Name: Dead
Purpose: This is what plays when the boss is finally killed. Normally on index 2.

The order these should be in is walk, no name, damage, dead, thrust, guard, victory, spell, skill, item, and other.

This is a complete version of what everything should look like:

<Sideview Battler Motion>
Name: Walk
Index: 3
Loop
</Sideview Battler Motion>

<Sideview Battler Motion>
Index: 0
Loop
</Sideview Battler Motion>

<Sideview Battler Motion>
Name: Thrust
Index: 3
Loop
</Sideview Battler Motion>

<Sideview Battler Motion>
Name: Guard
Index: 3
Loop
</Sideview Battler Motion>

<Sideview Battler Motion>
Name: Victory
Index: 3
Loop
</Sideview Battler Motion>

<Sideview Battler Motion>
Name: Damage
Index: 1
</Sideview Battler Motion>

<Sideview Battler Motion>
Name: Dead
Index: 2
</Sideview Battler Motion>

<Sideview Battler Motion>
Name: Spell
Index: 3
Loop
</Sideview Battler Motion>

<Sideview Battler Motion>
Name: Skill
Index: 3
Loop
</Sideview Battler Motion>

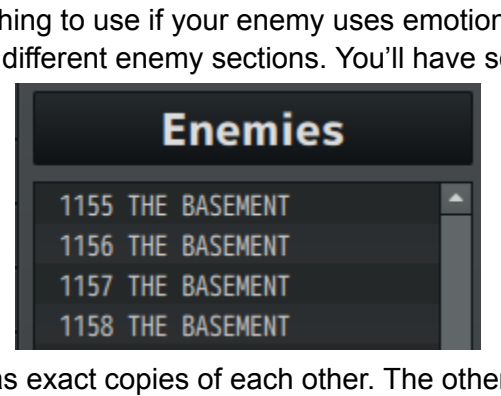
<Sideview Battler Motion>
Name: Item
Index: 3
Loop
</Sideview Battler Motion>

<Sideview Battler Motion>
Name: Other
Index: 3
Loop
</Sideview Battler Motion>

And that's it for notetags!

-Splitting Enemy Into 4

I'm sure you noticed while scrolling down the enemies list that almost every enemy has 4 copies! This is an important thing to use if your enemy uses emotions. Using control c + v, copy and paste your enemy into 4 different enemy sections. You'll have something like this:



The first 2 are going to stay as exact copies of each other. The other 2 will have slight changes with the <Sideview Battler Motion> notetags we just finished discussing.

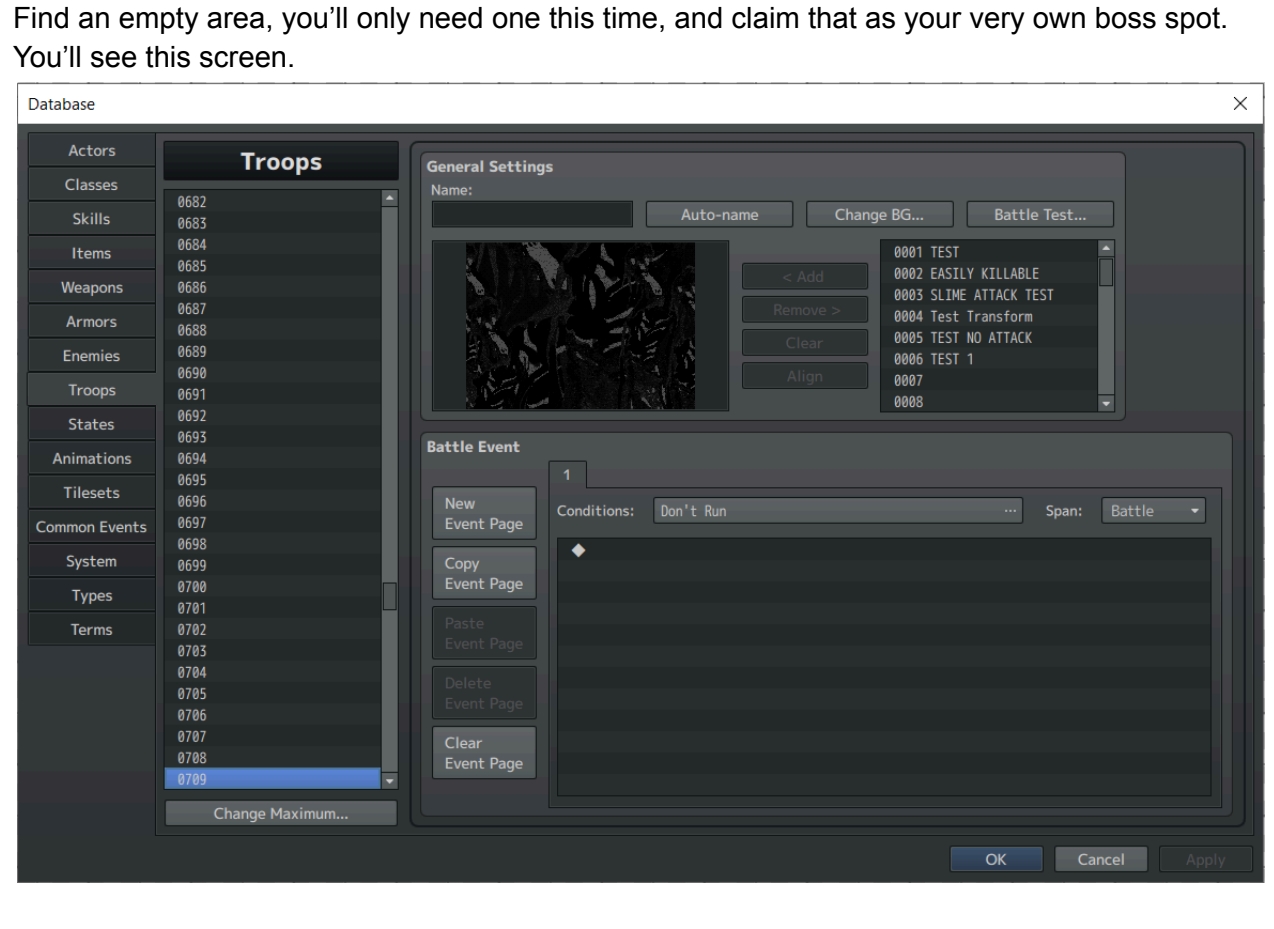
There are 3 levels in the emotion scale that a boss will follow, such as SWEETHEART going down the happy path. For the third copy of the boss, you would change the index of the emotions to be the index of the next level in the emotion scale you are following. The 4th copy will be the final scale. For example, if your happy emotion is index 3, your ecstatic is index 4, and your manic is index 5, your first 2 copies would use index 3 as your emotions, your 3rd copy would be index 4, and your 4th would be index 5.

And finally, after so much work, you've finally completed the hardest step in creating your boss, congratulations!

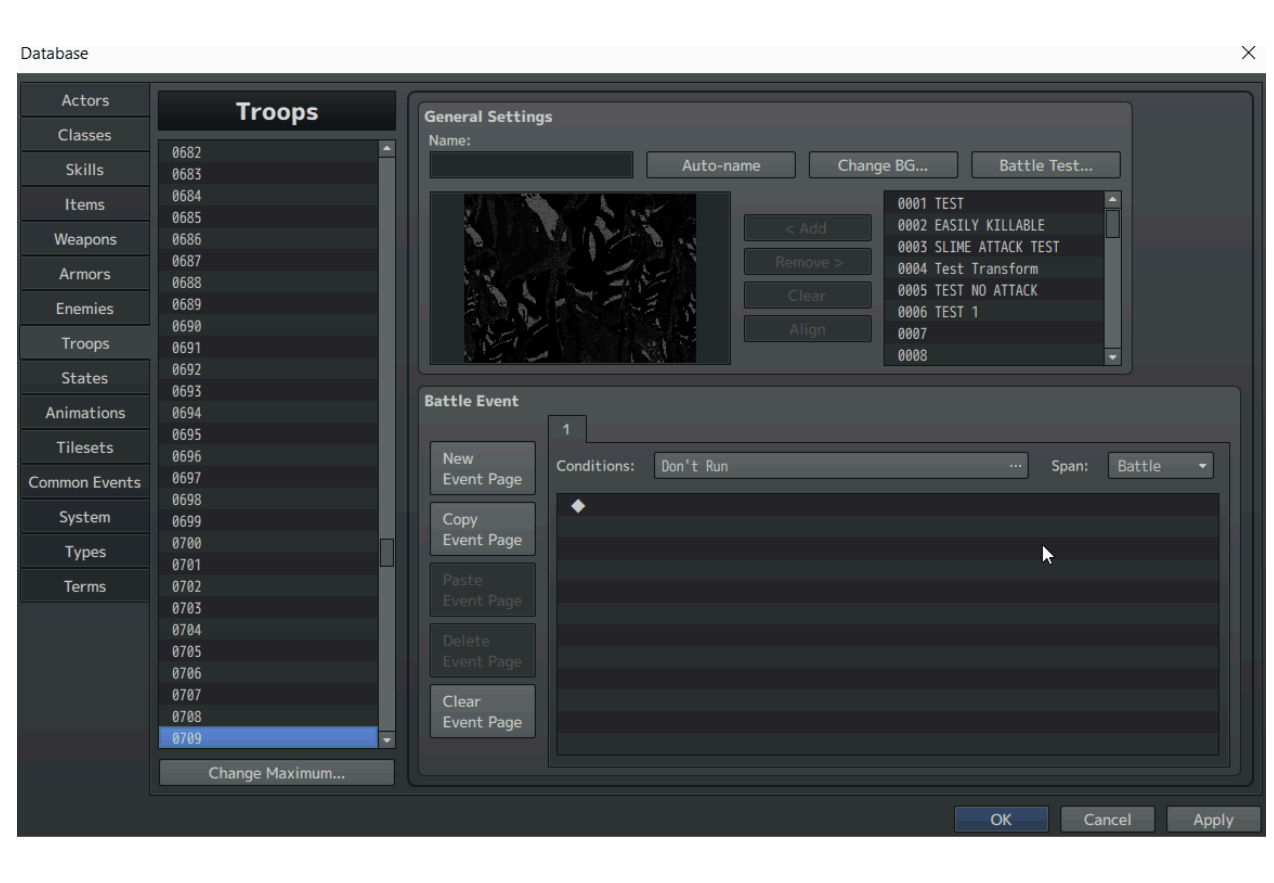
-Troops Tab

Welcome, you've made it to a new tab, good luck!

Find an empty area, you'll only need one this time, and claim that as your very own boss spot. You'll see this screen.



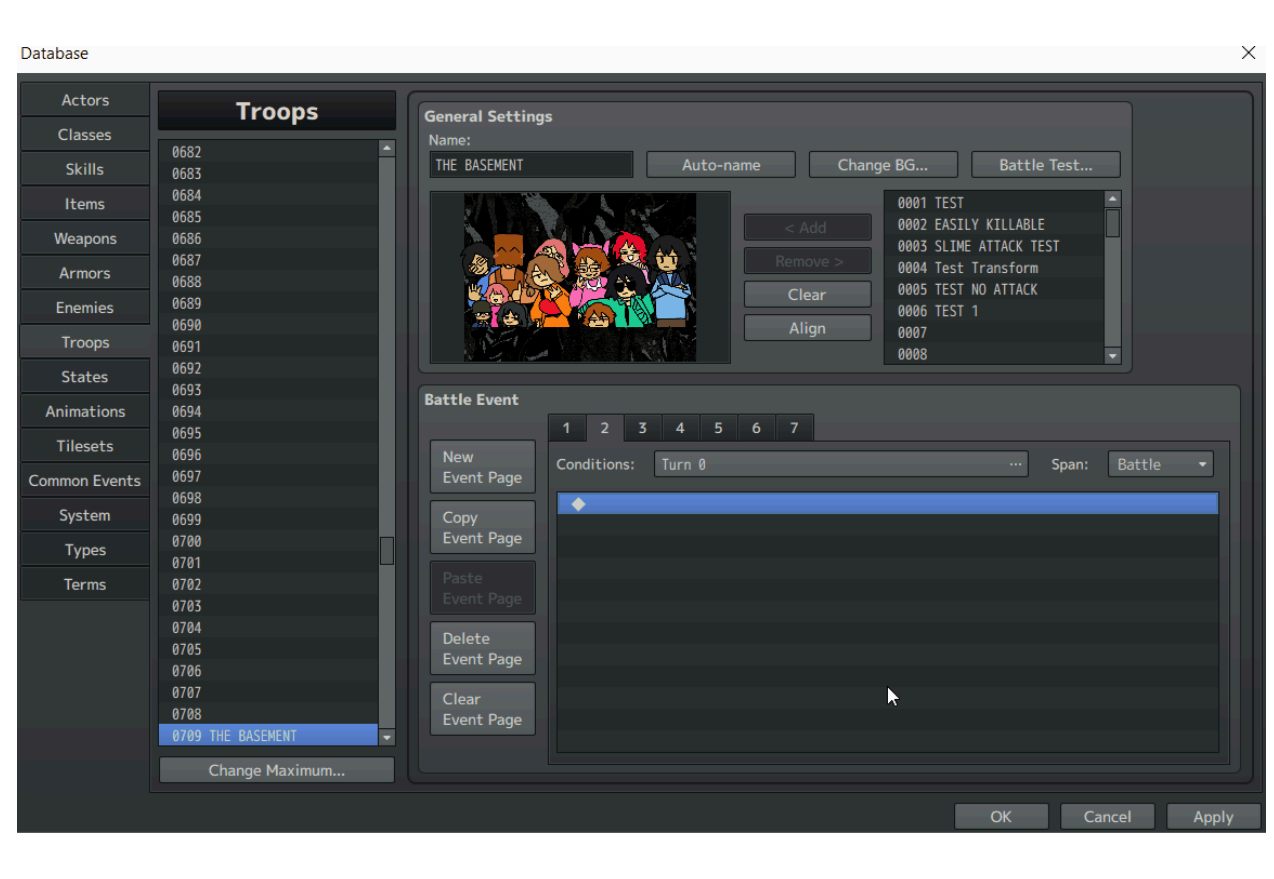
If you notice, on the side you can see a list you can scroll down. That's the enemy tab, where you can add in your very own enemy. You're gonna have to painfully scroll through the hundreds of enemies in search of yours, so it's best if you have your enemy number/id memorized. Here's the whole process shown below!



Once the image is in, you can move it around and position it as high or as low as you want it to be.
Once this is done, all you need to focus on is the Battle Event tab below it.

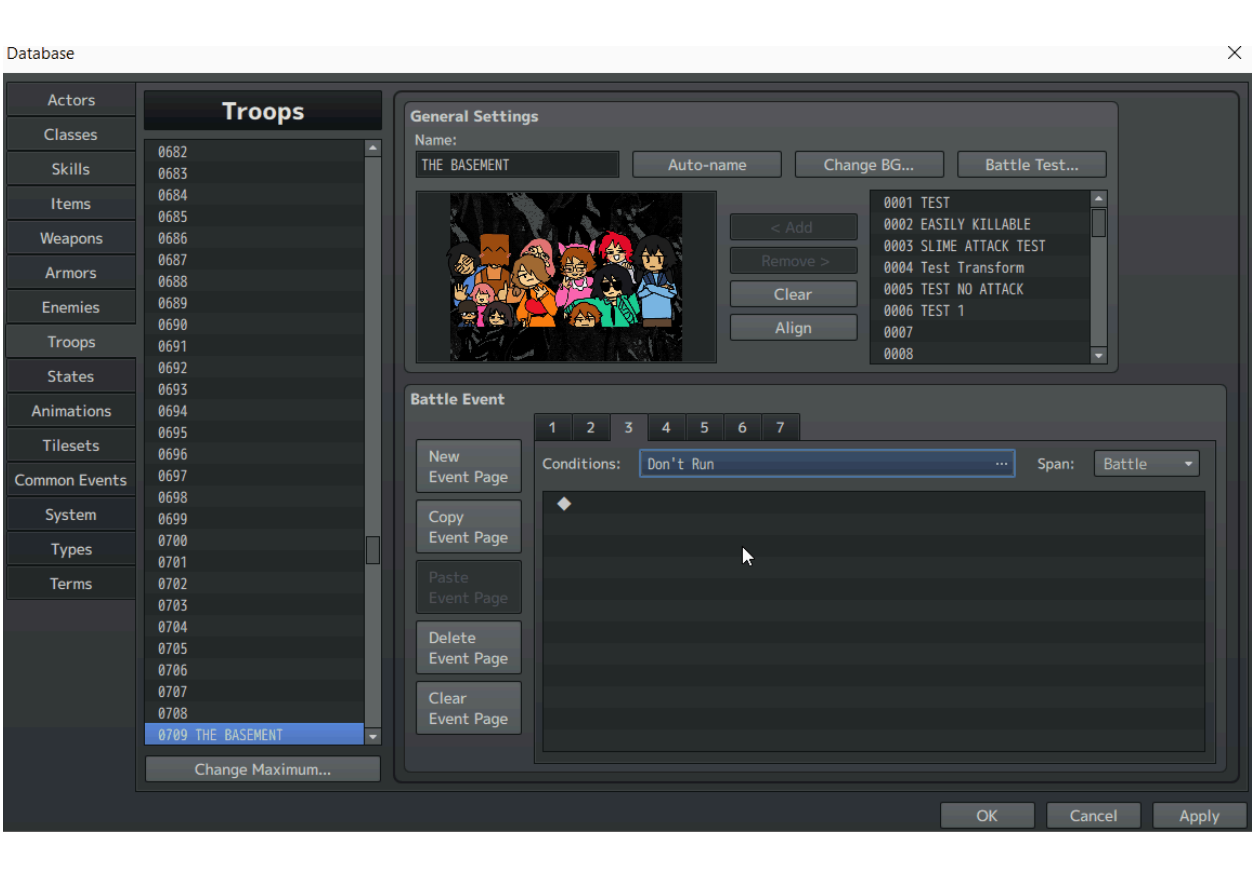
An average OMORI boss battle will have around 7 or 8 event pages, make those now by clicking "New Event Page". Event page 1 will remain empty, the only thing there will be the condition at the top. Make sure it says "Don't Run" if you don't want to allow the player to run away from the boss fight like regular OMORI bosses!

Event page #2 should have "Turn 0" as it's condition. Here are things you can add at the beginning of the battle, before any moves are made. For example, if you wanted to add text, you would double click the lines in the event page, click on plugin command (page 3 of the event commands list), and write "ShowMessage [insert yml file name here of your battle text].message_[insert message number here]. This is shown below.



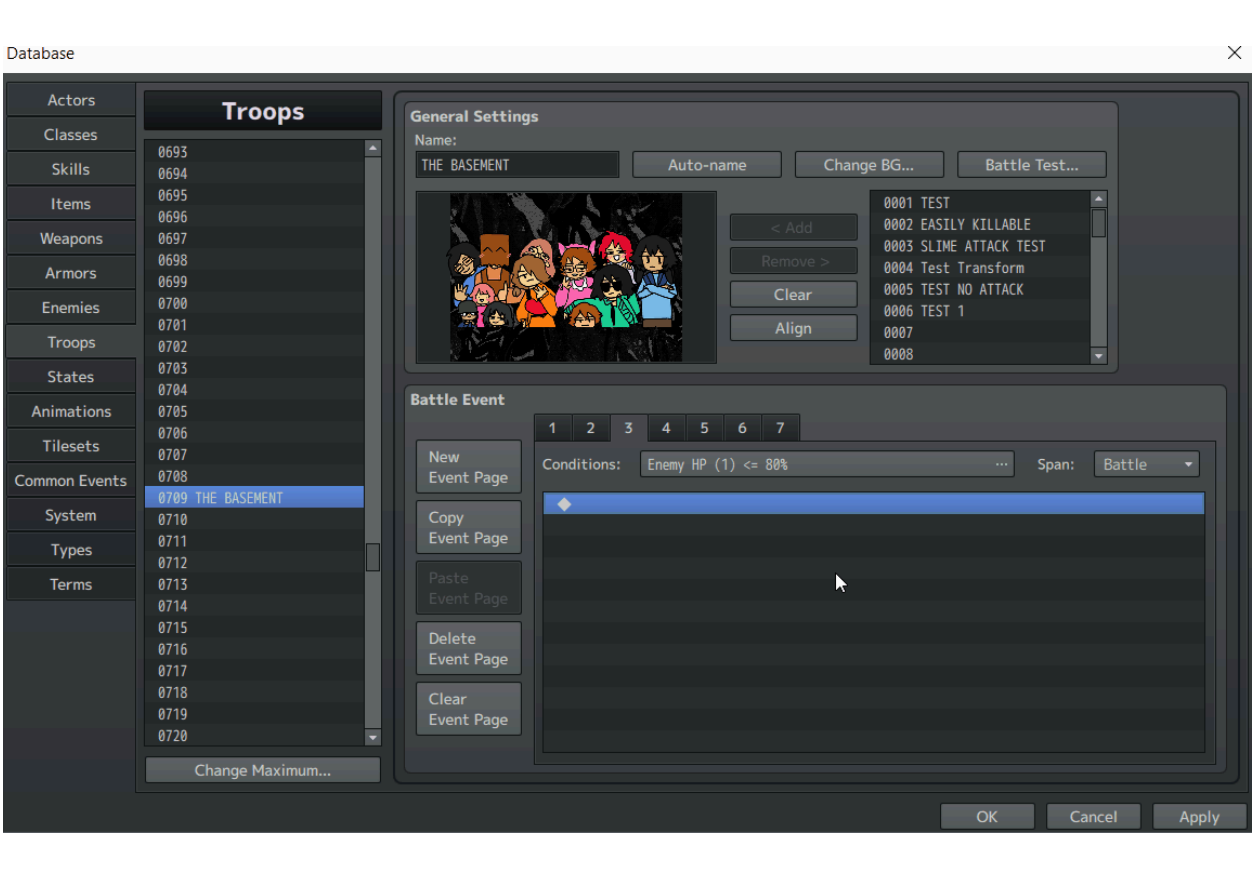
After you finish setting up this page, we can move on to the next few pages. These ones are linked to the boss' health. In OMORI, if a boss gets past a certain amount of health, they'll talk a tiny bit and move on to the next emotion stage. This can be set up with the next few event pages with the conditions at the top.

Let's say, at 80% health remaining, I want my boss to say "I will turn you into a convertible", written as message_7 in my hilolthisisayaan.yml file. I would open event page 3, and click on the conditions. From there, I'd click on the Enemy HP box, and set the percent to be 80% or below. This is shown underneath this.



Now that the condition is set up, another plugin command can be used to display text. Simple!

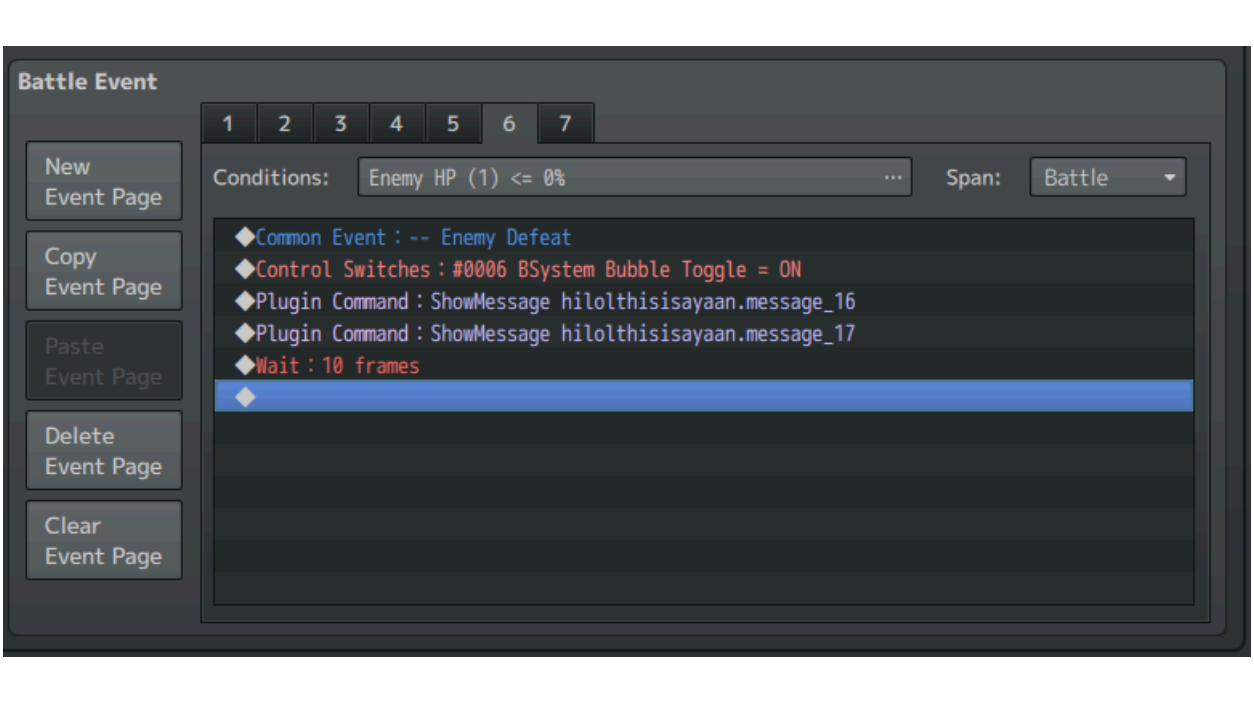
You can use this with the rest of the event pages, increment downwards by however much you want until you reach 0. Add whatever you'd like, text, or a change in emotion. In order to change the emotion of your boss, you would double click a blank line, click on "Change Enemy State" which is on page 3 of the event commands pop-up, change entire troop to the singular enemy name, choose if you want to add the emotion or remove one, and finally, choose the emotion, under the states tab. This is all shown below.



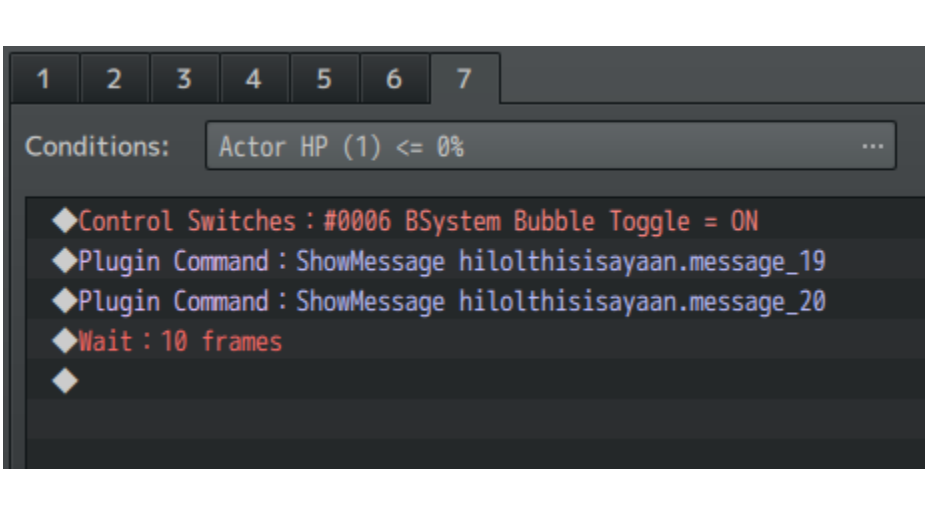
You can do this with the rest of the tabs, keep the last 2 blank. If you want to add more, then add more! Once you're done customizing these HP tabs, we can move on to the final 2 tabs.

These tabs are for the end of your boss fight. Second to last being you've successfully killed the boss, and the last tab would be when you die to the boss instead. In order to do this, we'd set the condition of one tab to be "Enemy HP (insert enemy name here) → 0%" along with whatever cool text you want to show up, and the final tab would be "Actor HP (OMORI) → 0%" with some cool text from the boss that's like "get owned nerd" or something.

Let's focus on the tab that has the Enemy HP 0% condition. Within the lines of the tab, you need a couple of things. Namely, the common event labeled "Enemy Defeat". Double click a line, go to event page 1, click common event, and find enemy defeat. Right after that should be Control Switch #6 turned on, which you can do by double clicking the line, going to page 1 of the event commands, and then clicking on control switches. Choose #6, and turn it on. From here you can add the text you like, and when you're done, add a "Wait: 10 frames" command, which can be found in page 2 of the event commands. Your page should look something like this when it's done.



Now, go to the final page, with the actor hp condition set. Add the same Control Switch (#6), then you can add your battle text, and finally, a Wait: 10 frames command.

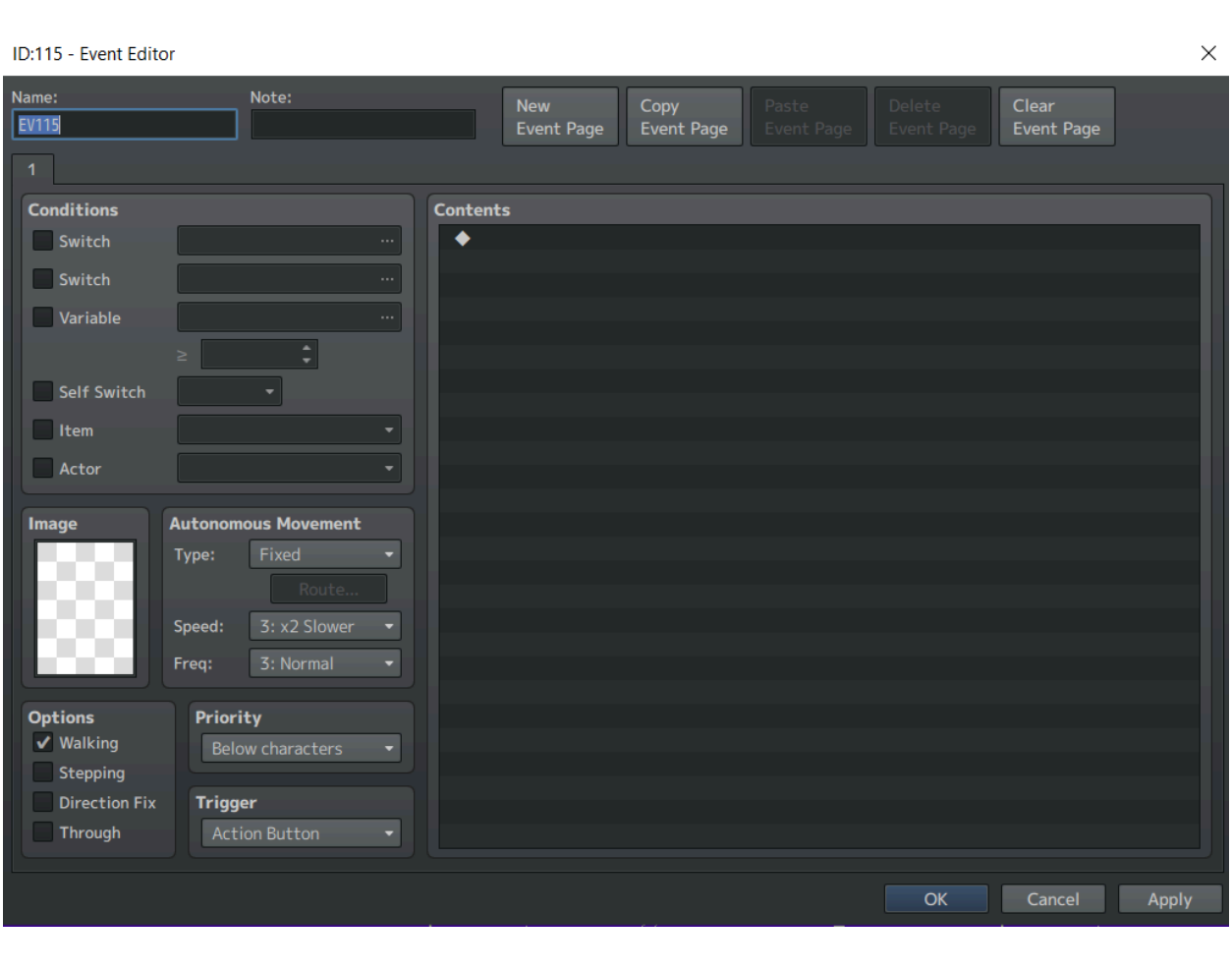


And that's it with the Troops tab! We're almost done. There is just one more thing to do.

-Implementing the Battle Into the Game
This is probably the most simple piece of the puzzle. Find the map you'd like to place the sprite of your characters. Once you're there, find any tile, and double click it to open the very cool event editor!

Tip: If you want to have your sprite be in a certain place on your map, instead of guessing where you are on the map, you can use this tool, which is made by Rph yet again! To use this tool, just drop the .exe file into the www_playtest folder, double click it, and wait up to 10 minutes for the program to finish doing the job. Once that is done, all you have to do is drag and drop the right image from the scaled folder into img/parallaxes/ and set that image as the background parallax on the map. Do this by right clicking on the map, click edit, and click on image under the Parallax Background section.

If you followed the steps correctly, you should be looking at something like this:



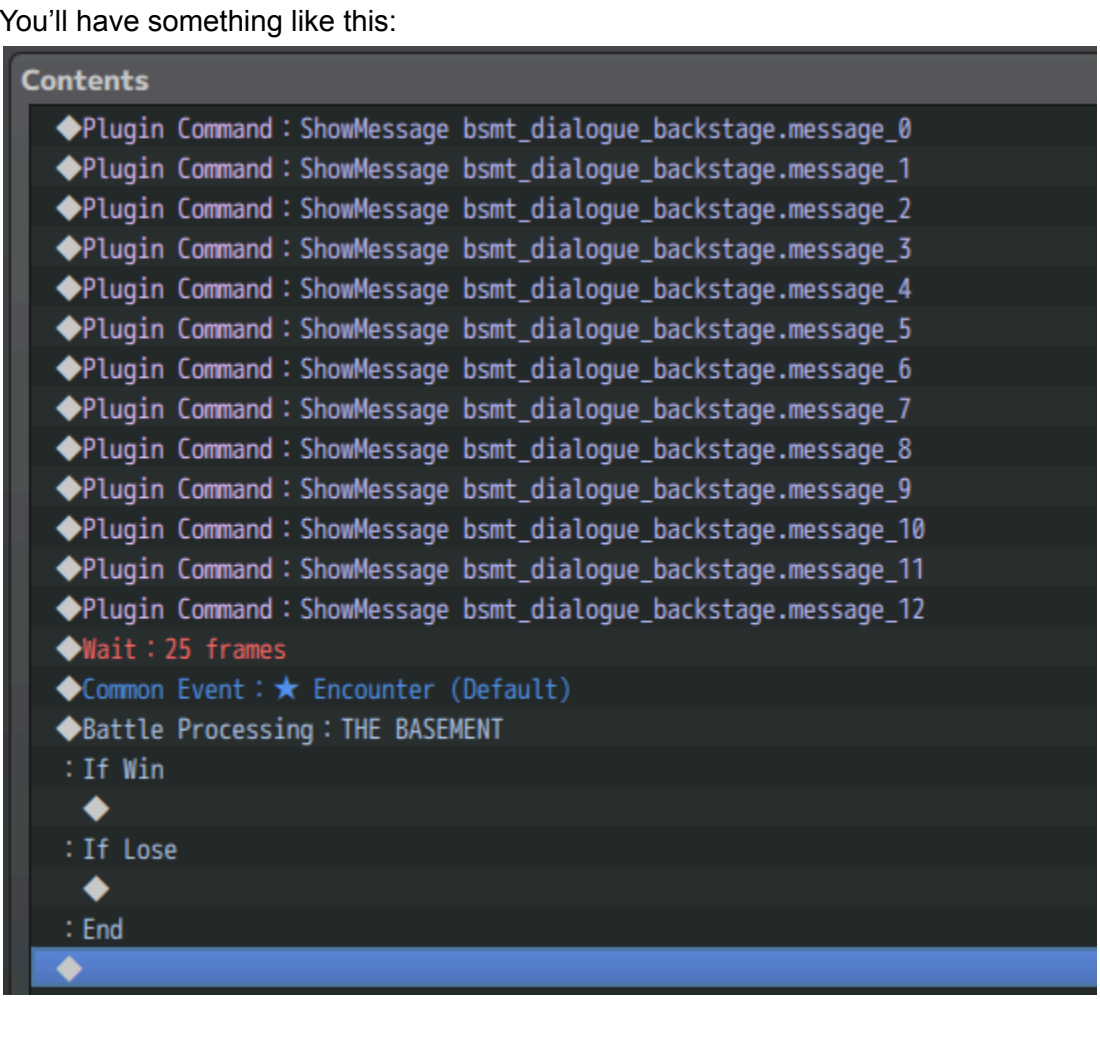
Give your event a name. If it's a person, I'd suggest writing their name as the event name as well.

Now, notice the image tab on the left side, towards the bottom? Click on that, and find the sprite you made previously. This way, this sprite will appear on the map as an interactable.

Moving on, let's focus on the contents section. Here you can input things like pre-battle text. After inputting anything you want pre-battle, click on the net empty line, click on common event, and click on common event 86. This will give the cool animation that OMORI uses when going into battle.

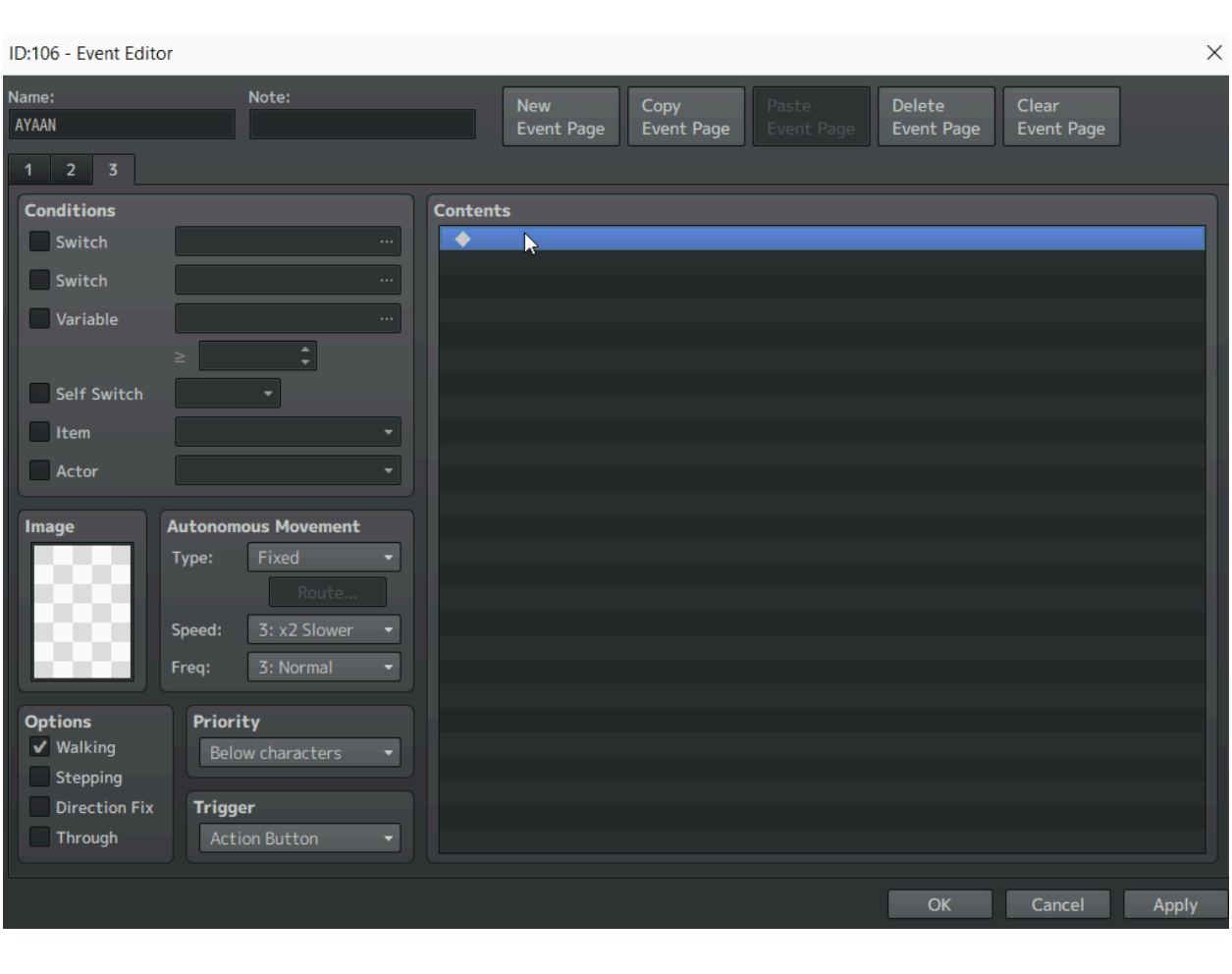
Right below this line, click on battle processing (page 3 of the event commands tab) and select your battle from the list that comes up. Make sure to check on can lose!

You'll have something like this:

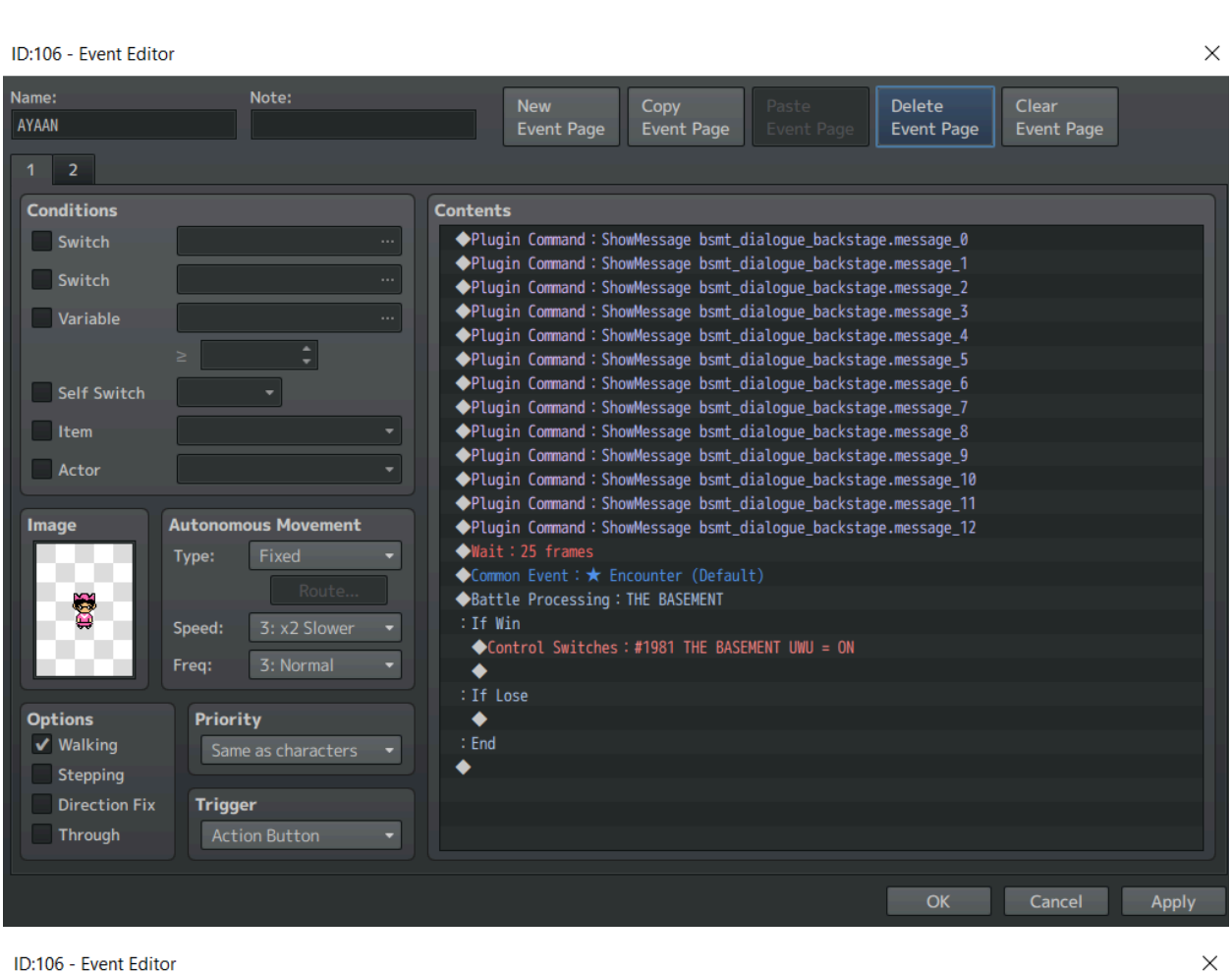


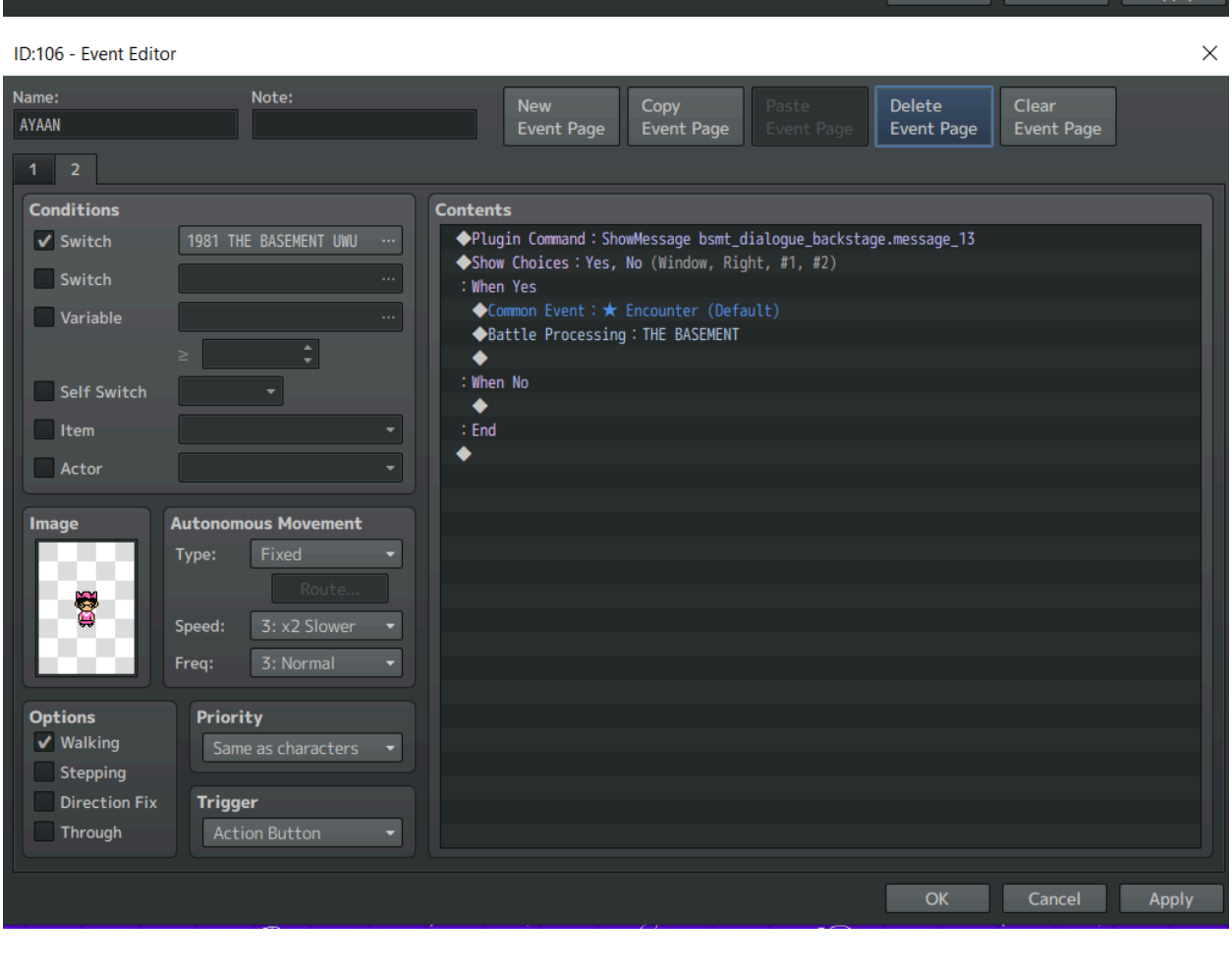
Now, it's time to introduce Switches and Self-Switches!

Let's say when you beat your custom battle, you want it to unlock something, such as a new room. This is where you'd use a switch. Double click on the line under "If win" and go to "Control Switches", which is on page 1 of the event commands tab. Click on the number shown, and find any empty switch spot, and give it a name you'll remember. This is all shown in the gif below.



Now that the switch is set, when someone beats that battle, it'll be turned ON. This can be used on the side bar, at the condition window. Select the switch you want, and that event tab will be used in the event that the boss is defeated. THE BASEMENT uses this by allowing you to replay the battle again, which is shown below.





Make sure this is in a new event page!

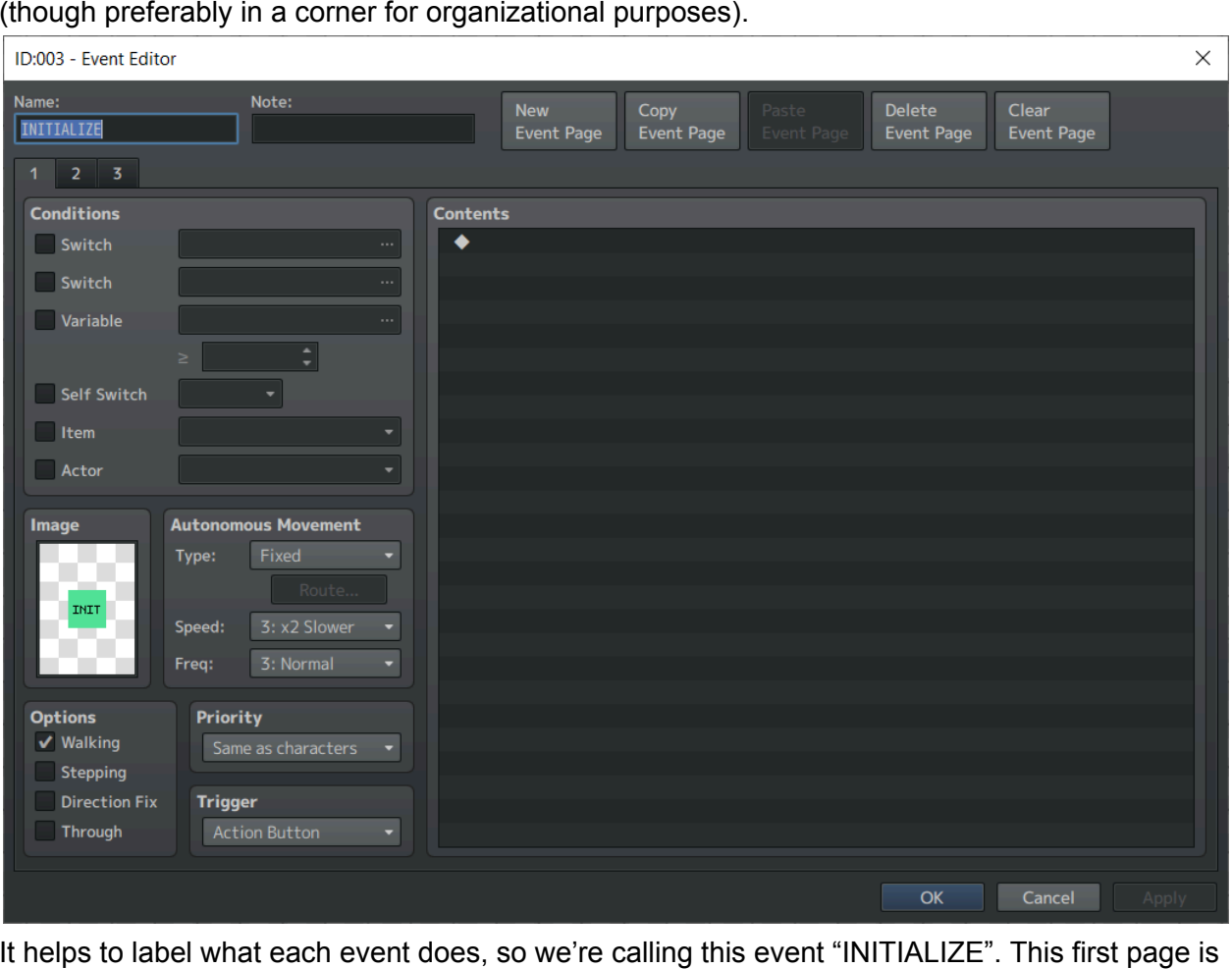
A self-switch is similar, but only used for that one event. It goes from A-D, and will only apply to that event. Honestly, I could've used a self-switch instead of a normal one when making THE BASEMENT, however, I didn't really know what they did, so…

Well, you did it! You have successfully created your very own boss fight! Pat yourself on the back!

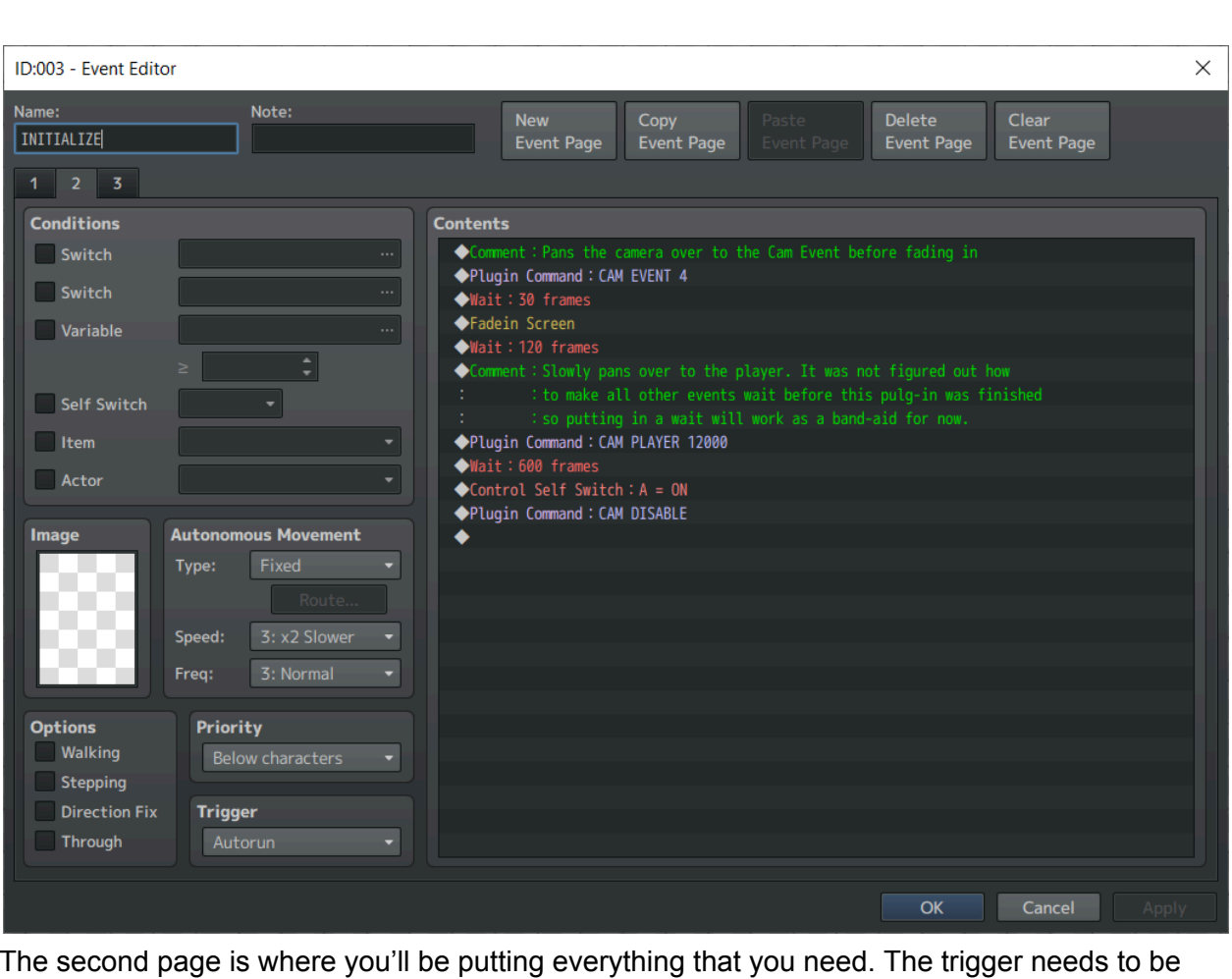
Getting Started With RPG Maker Events

As you may know, OMORI has a ton of pre-scripted events. Here, you will learn about how to make them yourself. Because there is a wide variety of what you can do, this tutorial will only cover the basic set-up for an in-game cutscene, as well as interactable objects.

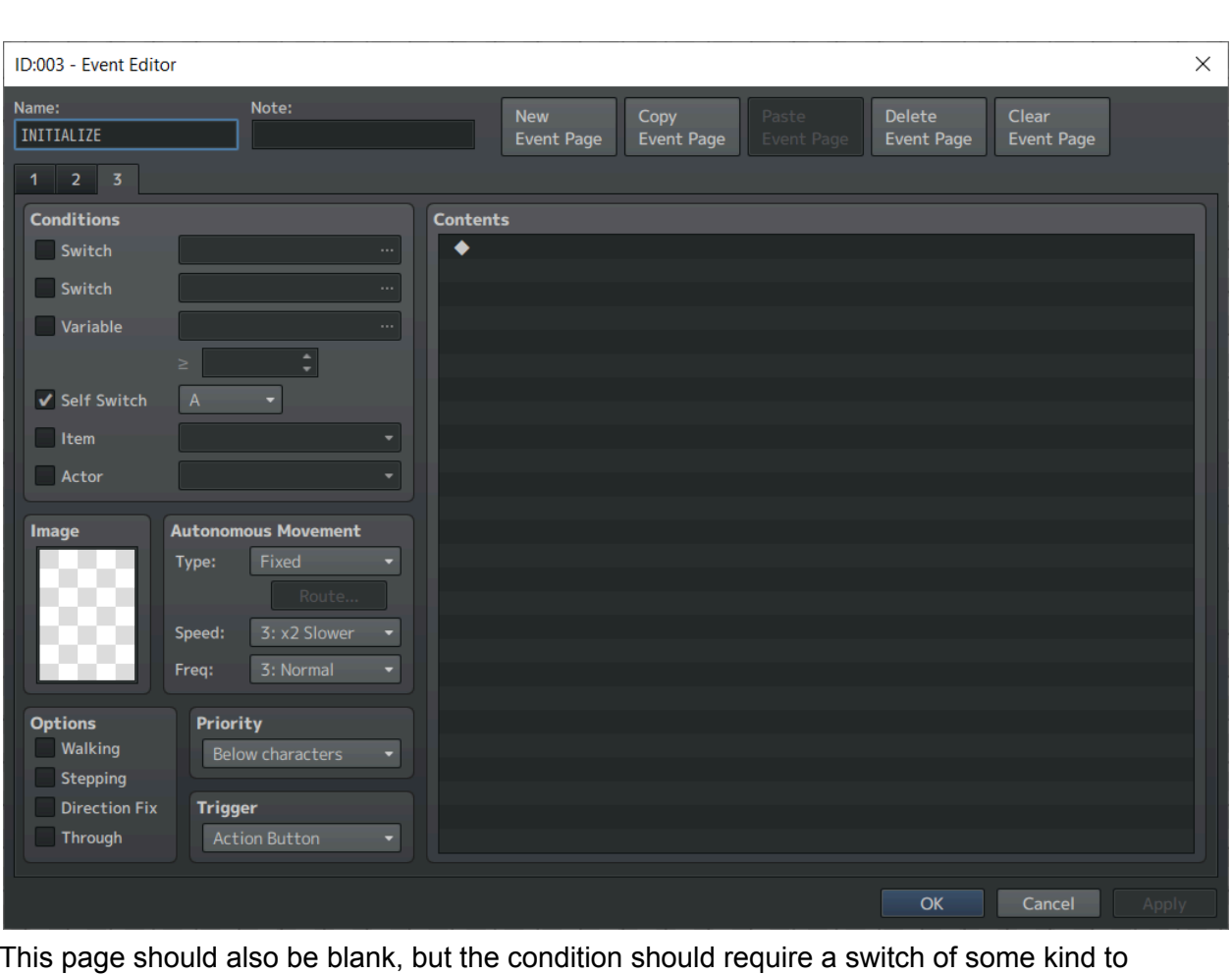
The first thing you need to do is to make a new event in your map. For this tutorial, this event will be triggered by a map change. Double click anywhere in the map to create a new event (though preferably in a corner for organizational purposes).



It helps to label what each event does, so we're calling this event "INITIALIZE". This first page is left empty so the developer texture never shows in-game (which is also a good habit to pick up. For dev textures the file name is DEV_TEST).



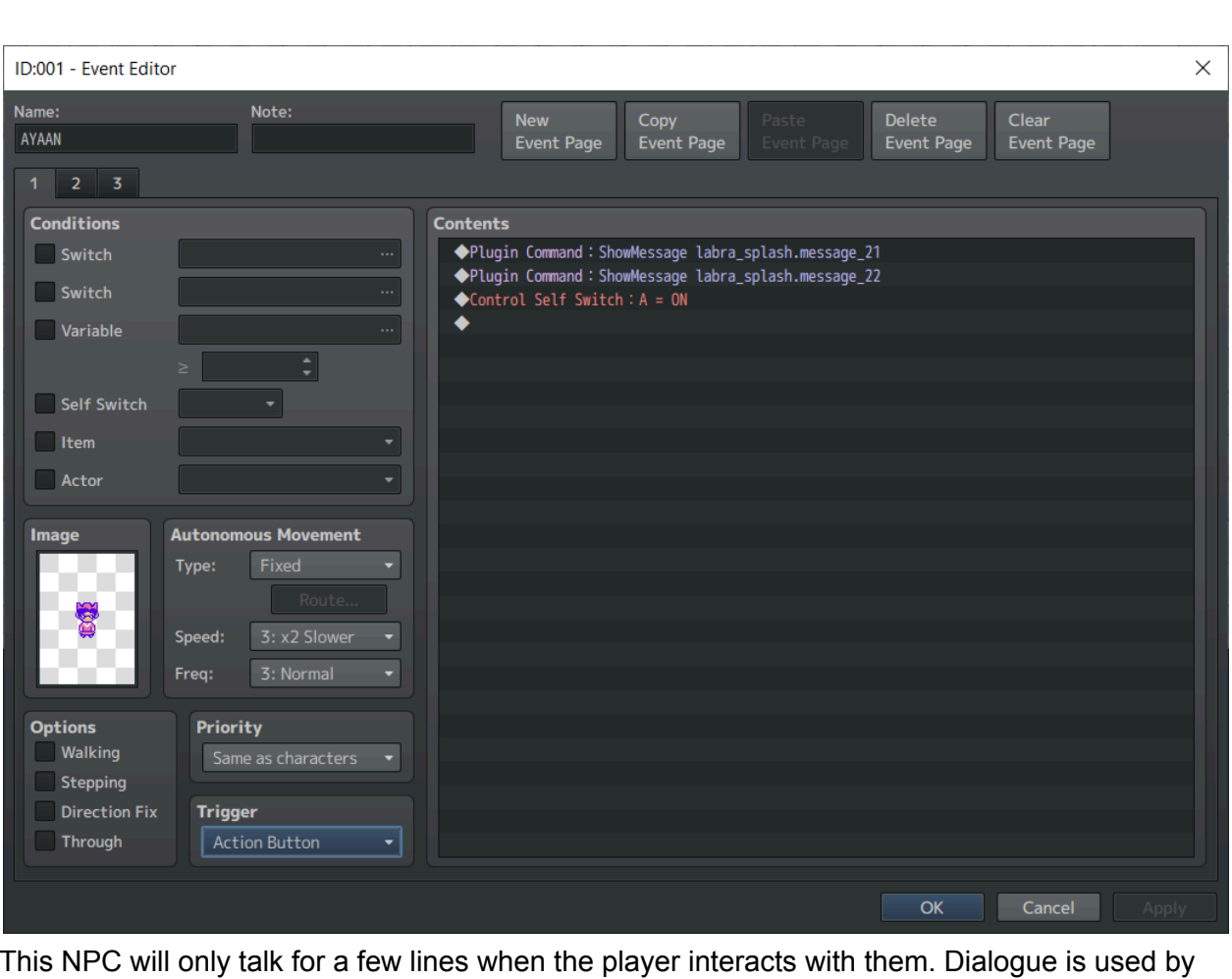
The second page is where you'll be putting everything that you need. The trigger needs to be set to "Autorun," that way the event runs when the player teleports into the map. Script out everything that you need to do (comment on what it does!), then at the end have some sort of self switch to turn on.



This page should also be blank, but the condition should require a switch of some kind to ensure the cutscene only plays once. You don't have to limit yourself to just self-switches. In fact, for more complex events that involve phases of the story it's recommended to have global switches.

And that's it for the basics! Feel free to experiment with RPG Maker and see what each script does! If you need to do a specific thing but don't know how, take a look at a similar event to see how that's achieved!

Making NPC's and objects is even easier. Double click in the spot in the map where you want to have it.



This NPC will only talk for a few lines when the player interacts with them. Dialogue is used by loading in YAMLs using the above plug-in command. ShowMessage (filename).message_#. See above for making proper dialogue files. If you want them to say something different, use the self switch method I mentioned earlier! Remember that later pages take priority. Again, don't be afraid to experiment!

Here are some common script/plug-in commands that might come in handy! For more info, take a look at the plug-in list and see what all commands they do!
ShowMessage <filename>.message_xx, displays dialogue

Script: // MapID
$gameVariables.setValue(9, x);
// X
$gameVariables.setValue(10, x);
// Y
$gameVariables.setValue(11, x);
// Direction
$gameVariables.setValue(12, x);
// Fade Type (0 - Black, 1 - White, 2 - None)
$gameVariables.setValue(13, x);
// Fade Duration
$gameVariables.setValue(14, x);
Teleports a player

AddChoice <filename>.message_xx (Labelx or END), Label adds message as an option, END is the final option.

ShowChoices x, shows choices

Script Calls

Here, you can find an extensive list of script calls to use in RPG Maker in order to give yourself some more wiggle room, maintained by none other than the lead programmer of OMORI, Archeia! I'd recommend learning some of it, as it includes manual self switching which is what some big mods like Cecilyspace do!
https://docs.google.com/spreadsheets/d/1-Oa0cRGpjC8L5JO8vdMwOaYMKO75dtfKDOetnvh7OHs/htmlview?pli=1#gid=0

JavaScript Documentation

https://kinoar.github.io/rmmv-doc-web/globals.html this document provides some bare-bones explanation on setting up plug-ins for yourself.

Delta Patches

Delta patches are the key to making sure your mod is as compatible as possible with others. Instead of replacing the whole file, they replace only part of the file. This means that two mods that modify the same file can work together! They'll also reduce your mod folder size by a considerable amount, so use them when you can!

Generating Delta Files

Delta patches are notoriously hard to do by hand when replacing text, and impossible if you're replacing data. Luckily, our savior Rph has a tool that can generate deltas. Heck, it even generates whole mod folders! You can find it here: https://github.com/rphsoftware/gmkr/releases

The tool is a bit picky, so you need to follow these steps in order to successfully generate what you need.

Video tutorial:
https://cdn.discordapp.com/attachments/813311727503671306/846464970636263454/gaming.mp4

1. Use Rph's playtest mod to make a playtest file, or generate one with OneLoader..
2. Mod the files to your liking
3. Obtain a completely clean copy of OMORI. That means uninstalling the game through Steam and deleting the local files from your computer (backup your saves and mod configurations!)
4. Save that clean copy somewhere else for next time
5. Run the gmkr tool in Powershell
6. Give the file path to your clean copy, which is normally C:\ProgramFiles(x86)\Steam\steamapps\common\OMORI
7. Select your mode of operation
8. Give the file path to your playtest folder
9. Follow the instructions in the prompt of gmkr.

And that should give you what you need! Now you just need to indicate in your mod.json file that what you are putting in are deltas. That can be done by just replacing "text" with "text_delta" and so on.

```
{
  "id": "smilemore",
  "name": "Smile More",
  "description": "by Labraman, Cooldry77-T1-RTD, and Pixelguin",
  "version": "2.1.0",
  "files": {
    "plugins": [],
    "text_delta": ["text_delta/"],
    "data": [],
    "maps": [],
    "assets": ["img/faces/"],
    "exec": []
  }
}
```

Example shown is the mod.json for Smile More, which uses text deltas.

Generating Image Deltas
Image deltas are an interesting case, as they are not always beneficial due to the way images are compared to text-based files. They are also OneLoader-exclusive. However, compact packaging and optimization is instrumental to a career in software engineering. So give it a try!

Use this link to generate .olid files, which are image deltas: https://rph.space/olid/generator.html

Compare the file size of the .olid to the modified image file. If the .olid is larger in size, don't include it. Next, put them into a folder of your liking within the mod folder. OneLoader has two methods of image delta patching.

First, in your mod.json file, add this flag after the file brackets like so:

```
{
  "id": "sunburn",
  "name": "Sunburn Scene",
  "description": "adds in a short aubrey x sunny scene in aubrey's room & a new item.",
  "version": "0.0.1",
  "files": {
    "text_delta": [
      "text_delta/"
    ],
    "data_delta": [
      "data_delta/"
    ],
    "image_deltas": [
      {"patch":"img/system/", "with":"delta/system/", "dir":true}
    ]
  },
  "_flags": [
    "do_olid"
  ]
}
```

Example shown from the Sunburn Remaster.

Here are the two methods you can use for image delta patching:

```
  "image_deltas": [
    {"patch":"img/faces/01_something_i_forgot.png",
"with":"deltas/faces/01_goodshit.olid"}
  ],
```

This method is good for single files.

```
  "image_deltas": [
    {"patch":"img/characters/", "with":"delta/characters/", "dir":true},
    {"patch":"img/animations/", "with":"delta/animations/", "dir":true},
    {"patch":"img/atlases/", "with":"delta/atlases/", "dir":true},
    {"patch":"img/faces/", "with":"delta/faces/", "dir":true},
    {"patch":"img/pictures/", "with":"delta/pictures/", "dir":true}
  ],
```

This method is better for multiple files in folders.
Either way, you'll need to put in the "image_deltas" argument in like you would with any other delta file.

RPG MAKER Sprite Sheets

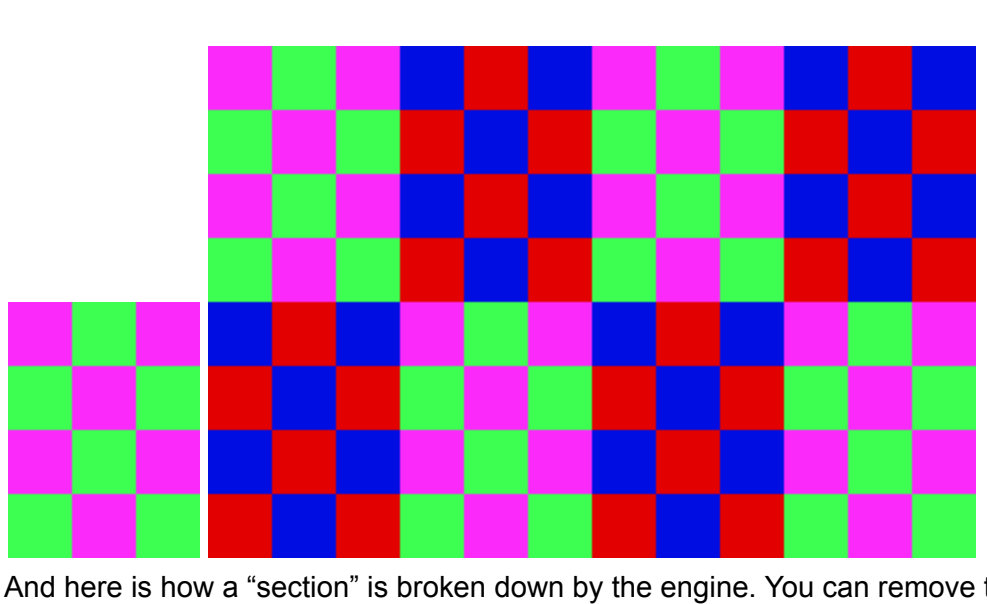
If you have even bothered to touch Rpgmaker MV once for a mod, you should know how weird the sprites seem to work. The sheets for events actually have to be aligned in a certain way:

A normal spritesheet will have to be 12 x 8 tiles, technically containing 8 sections for an up, down, left, and right direction of the sprites. The size of these tiles can be just about anything, however.

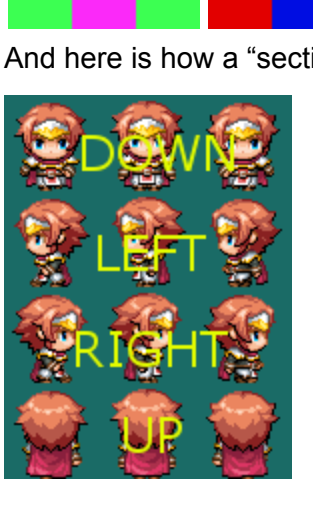
You can format a smaller spritesheet that's 3 x 4 tiles (1 section) by adding a "$" to the beginning of the sheet's name.

Adding an "!" to the beginning of the sheet's name will make the sprites perfectly aligned, as opposed to the default which is y-shifted up by 6 pixels.

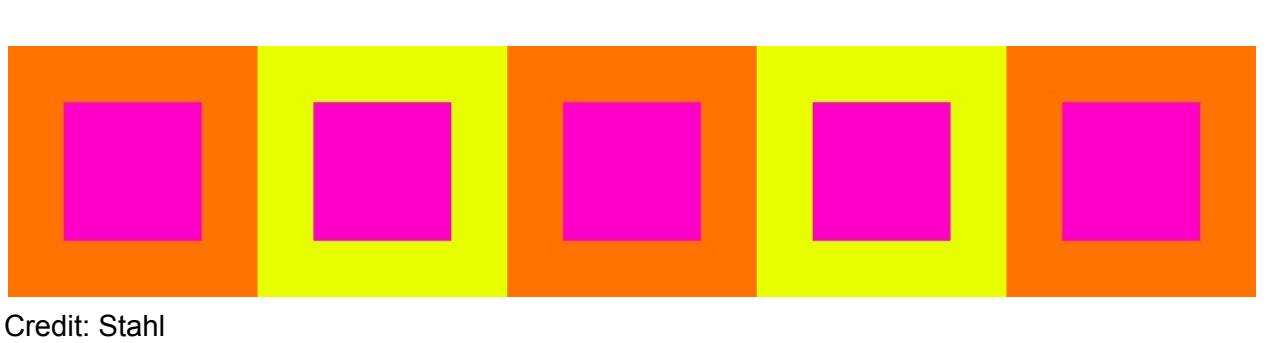
Here are template images for both formats in 32x32 (OMORI''s usual tile size):



And here is how a “section” is broken down by the engine. You can remove this by adding “Direction fix” to the event.



Animation Frames



Credit: Stahl

For battle animations, there is a very particular frame to follow. The alternating box is 192x192, and is the typical boundary for animations. However, if you want it inside the portrait, then you would follow the purple boxes.