

[Introduction](#)
[Who Made This](#)
[How to use the Editor](#)
[Level Browser](#)
[Edit Game Settings](#)
[Testing the game](#)
[Editing Maps](#)
[Drawing Tools](#)
[Map Collision Editor](#)
[Collision Values](#)
[Selecting Collision Tiles from the Map](#)
[Map Layers](#)
[Selecting Tiles From the Map](#)
[The Chipset](#)
[Zooming-in and Panning the Map](#)
[Sprite Editor](#)
[Visibility Flags](#)
[LoadSwf](#)
[RPG Sprites](#)
[About Making Scripts](#)
[Sprite Appearance](#)
[Copy Sprite](#)
[Script Commands](#)
[Comment Command](#)
[Condition Command](#)
[Fade Music Command](#)
[Loop Command](#)
[Move Command](#)
[Music Command](#)
[Pause Command](#)
[Set Variable Command](#)
[SWF Command](#)
[Animate](#)
[Appearance](#)
[Look](#)
[Look At](#)
[Absolute Movement](#)
[Relative Movement](#)
[Wait](#)
[File](#)
[Help](#)
[Convert Text to ActionScript](#)
[Copy Level Actionscript](#)
[Copy Level XML](#)
[List Folder Actionscript](#)
[List Resources](#)
[Convert Map Format](#)
[Copy Map Actionscript](#)
[Copy Map XML](#)
[Importing Maps](#)
[Resizing Maps](#)
[How the editor was made](#)
[Required Folders](#)
[Sprite Image Settings](#)
[About defaultSprite.exe](#)
[Level Data Structure](#)
[Map Data Structure](#)
[Old Map-Data Structure](#)
[Making a Game That Uses This Editor](#)
[Making Sprite Editors](#)
[Making Your Game Self-Contained](#)

Introduction

What is this program?

This is a general-purpose level editor for flash games programmed in AS2. It can theoretically edit any 2D game that uses maps and sprites, such as RPG's and Platformers. Of course, the game needs to be designed to use the editor's data.

Why was this made?

The game editor was originally envisioned to be an RPG Maker for Flash. I created this RPG game-engine in order to test the editor.

Why use this program?

I created this game editor when it started taking too long for one of my flash games to compile. This editor allows you to instantly test a level in the game! While designing it, I realized that I could make it seamlessly import other editors on-the-fly, making it very flexible! It's not perfect, but it's a hell of a lot faster than waiting 5 minutes for a game to recompile!

How was this done?

Turning the game editor into an RPG maker only required creating two sprites. The "player" and the "rpgSprite". The "player" sprite only exists in the game engine. It can walk around, and talk to stuff. (by calling the `talk()` function within `movieClips` that are in front of it.)

The "rpgSprite" can run scripts and look like anything. It's just like an "event" in RPG Maker 2000, and it's probably one of the most complicated sprites imaginable. Creating the sprite in the game engine only took a few days, but creating the editor for it took a full month!

Where's the battle system?

There isn't one.

Most of the games I create don't have battles, so I didn't bother making one. But you could program a battle system as a SWF file or a movieClip and have an `rpgSprite` place it into the game somewhere using a "swf" command whenever you want a fight to occur.

What? No undo!?

There is no "undo" command. So if you make a mistake, double-click the level in the file-browser to reload it from the last time it was saved.

Who Made This

Who do I blame?



[Humbird0](#)

<- This guy!

I'm a cashier... (seriously!)

... but I also play with Flash in my spare time.

Background

This isn't in perfect chronological order, because some things overlap.

Learning How to Program

When I was 8 years old (growing up in the early 90's), my dad gave me a Commodore VIC-20 computer. I taught myself how to program in BASIC by reading its excellent instruction manual.

A few years later, my dad gave me an Atari 800 XL, which could do many things the Commodore couldn't do. It also had tons of games. For many years, I loved that computer... until a lightning storm started loving it. After the power surge, I was amazed when its diagnostic mode told me, in precise detail, exactly how fried its circuits were!

When I was about 13, I was given a Tandy computer that ran DOS, GW Basic, and later QBASIC. I used to plow through my homework in study hall, then ask if I could visit the technology class so I could copy commands for GW Basic from their instruction manual... I was not a normal kid.

Then my mother got a Macintosh and I took a hiatus from programming for a few years until I discovered Flash.

Many years later, when I was going to Collins College, I had a few classes that taught the basics of programming in C++. It came fairly easily to me, but I haven't had any reason to pursue it further.

Rpg Maker 2000

When I was around 17, I discovered RPG Maker 95. I was in heaven! I had never been able to make a real game before!

I later acquired RPG Maker 2000, which I loved even more! It was fun to push it to its limits, making it do things it was never designed to do. But I eventually got fed-up with its limitations and moved onto using Flash full-time.

Flash

One day in high school, I got jealous when one of the kids used Flash to create an impressive class project. So I started teaching myself how to use it.

Later on, I discovered RPG Maker 2000, and sort of ignored Flash for awhile. Then a few years ago, I was creating a game similar to Zelda with real-time combat. But the things I had to do just to make it work were getting ridiculous. So I finally got fed up and switched over to Flash from that point on.

Ever since then, I have wanted to create an RPG Maker in Flash, so that I could have the best of both worlds. I've only just now gotten around to it.

College

I went to 2 colleges:

MVCC & Collins College

After high school, I wanted to create video games, but the closest thing offered at the local college was 3D animation, so I studied traditional art and 3D Studio Max. The art classes completely reinvented my approach to drawing. MVCC was really worth it!

Then, I arbitrarily chose another college that did offer courses in game design. Compared to MVCC, Collins College was a joke, and much more expensive. Most of the things they taught were things I had already experimented with during high school. But I did learn a few new things.

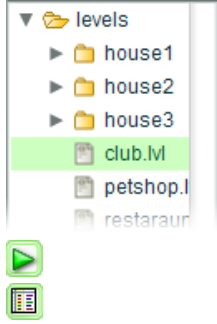
Call me a fool, but after that, I decided to stop chasing things I was supposed to want (that whole "career" thing), and focus on the things that actually mattered to me.

(a steady paycheck and a hobby)

So I moved back near my family, got a job at a nearby store, and have pretty much everything I really want in life. My mother still thinks I'm crazy.

How to use the Editor

Using the Editor



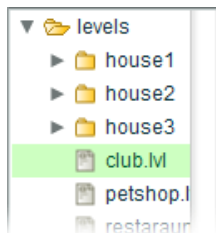
Everything starts with the file browser. Once you create or open a level, the map editor will open, allowing you to draw tiles. After that, use the collision editor to adjust where the walls are. Finally, use the sprite editor to place people, objects, and enemies.

From here, you can test the level by pressing the play button.

You can edit the game's overall settings by pressing the database button.

Level Browser

File Browser

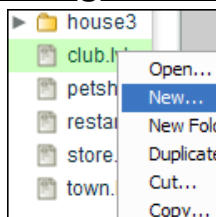


This menu allows you to create, edit, and move levels.

All level files exist in a folder called “levels”

This menu allows you to browse the contents from within the editor.

Making a new level

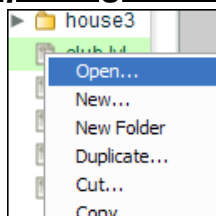


Right-click a folder and select “New...”

Alternatively, you can also press CTRL+N.

After giving the level a name, the editor will open up into map-edit mode.

Opening a level



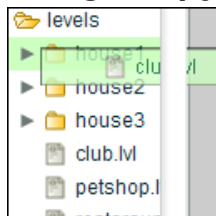
Double-click a level.

Alternatively, you can right-click and select “Open...”

The editor will then open the map-editor.

NOTE: If you open another level while one is already open, the previous level will automatically be saved. To revert any changes made to the current level since its last save, simply re-open it.

Moving / Copying a level



~~You can drag and drop a level or folder to another folder.~~ (Disabled to improve performance)

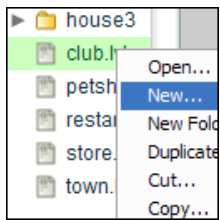
You can also cut, copy, and paste level files.

You can duplicate a level or folder by copying and pasting to the same location. The editor will ask you to enter a name for the duplicate file.

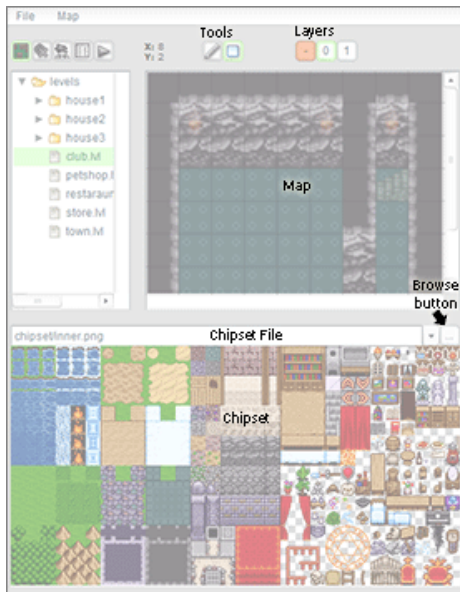
You can also duplicate a level by dragging a level file from Windows onto a folder in the Level Browser.

NOTE: If you move a level, the game will need to be updated to use the new location. For instance, the game’s starting location may need to be updated, and teleport commands in RPG’s will also need to be updated.

Editing Maps

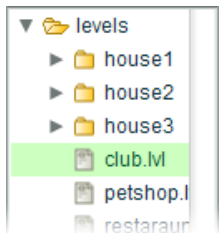


First, create a new map or open an existing one by double clicking a file. That will display the map editor.



You edit maps by [selecting tiles from the chipset](#) and drawing them on the map. The selected [tool](#) determines how the tiles will be drawn. You can draw tiles on different [layers](#) to control how they overlap. It's also possible to [select existing tiles](#) from the map.

Remember: You can't undo



If you make a big mistake, double-click the level in the file browser to reload it from the last time it was saved.

Drawing Tools



Pencil tool

This draws tiles, one at a time.

If many tiles are selected, they'll repeat over and over as they're being drawn.



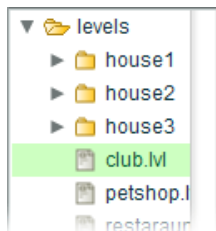
Rectangle tool

This draws tiles over a large area, repeating the selection over and over.

Where's the fill tool?

I didn't add a flood fill, because I couldn't figure out how to implement an "undo" function. A fill tool can make some very big mistakes.

What? No Undo!?



There is no "undo" command. If you make a big mistake, double-click the same level in the file-browser to revert back to the last time it was saved.

The Chipset

About the chipset

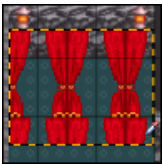


The chipset is a picture containing all the tiles that a map uses. Think of it as your palette. Each map can use a different chipset.

Using the chipset



When you click a tile in the chipset, it'll be highlighted. That's the tile that will be drawn when you edit the map.



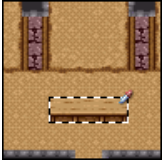
It's also possible to select a whole group of tiles from the chipset by clicking and dragging. When you draw them on the map, the selection will repeat over and over.

Picking a different chipset

To change which chipset the map uses, double-click one of the items in the list to the right. This will allow you to replace that item with a different chipset image file. A map can use multiple chipsets at the same time. To add another one, click on the + symbol at the end of that list. You can remove a chipset by hovering over one of the list items and clicking on the - symbol to the left of that item to remove it. You can switch between current chipset images by clicking different items in the list. Then you can select tiles from the left area and draw them on the map.

Select the chipset to display. (+) Adds a new Chipset. (-) Removes that chipset. Double-click to choose a different file for that chipset. Hover to see which tiles use that chipset on the current layer.

Selecting Tiles From the Map



It is possible to select existing tiles from the map.

Hold the SHIFT key, then click and drag to select tiles from the map.

You can then draw the tiles that you selected.

Not only is this convenient, it's also useful for moving lots of tiles at once. It even allows you to transfer tiles from one layer to another. Just select some tiles, switch layers, then draw.

Zooming-in and Panning the Map

Every map in the editor can be panned and zoomed using the mouse wheel. This doesn't affect the game at all, it's just a convenient way to navigate around the level in the editor.

Keep in mind that computer mice vary. Some mouse wheels cannot be pressed like buttons, and some mice don't have wheels at all. None of these features are necessary to edit a level.

Panning

To scroll around the map easily, click and hold down the mouse wheel, and then drag. (yes, that scroll-wheel is also a clickable button on many mice)

Zooming

To zoom in or out on a map, roll the mouse wheel up or down. The maximum zoom is 200%. If a map is really small, it won't be possible to zoom in or out.

Games will always display maps at 100%

Map Layers

Using Layers



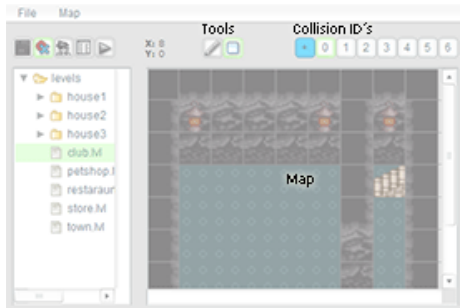
Map tiles can overlap. Use this interface to select which layer to draw tiles on. You can also press a number on the keyboard to select layers quickly. The ~ key selects layer 0.

To increase the number of layers, click the + button. To reduce the number of layers, hold SHIFT and click the – button. Be careful when removing layers. It'll delete all tiles on that layer.

Most games place the player in the map between layer 1 and 2. So tiles on layer 2 and above will overlap the player.

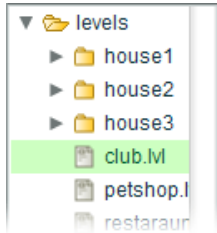
You're not likely to need more than 3 layers, but it is possible to have up to 7. (It's a completely arbitrary limit)

Map Collision Editor



First, you select a [collision ID](#), then you draw the collision tiles on the map. The selected [tool](#) determines how the tiles will be drawn. It's also possible to [select existing tiles](#) from the map.

Remember: You can't undo



If you make a big mistake, double-click the level in the file browser to reload it from the last time it was saved.

Collision Values

Collision Values



Use this interface to select which collision ID to draw.
You can also press a number on the keyboard to select ID's quickly.
The ~ key selects ID zero.

In most games, collision ID 0 is passible, and collision ID 1 represents walls. All the other collision values represent whatever the game wants them to. It'll vary from game to game. But most values go unused.

You could program a game that uses one ID as water and another for harmful objects like lava. A platformer game could use an ID to define one-way jump-ledges.

Selecting Tiles from the Map

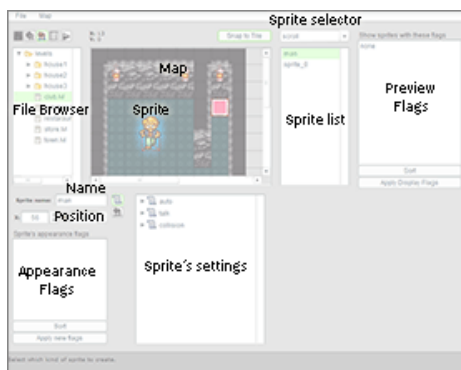
Selecting existing id's



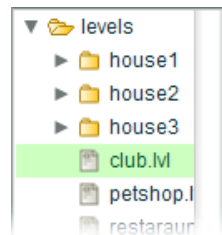
It is possible to select existing collision id's from the map.
Hold the SHIFT key, then click and drag to select id's from the map.
You can then draw the id's that you selected.
This allows you to place collision id's in the same pattern as the selection.



Sprite Editor



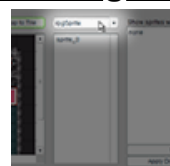
Remember: You can't undo



Be careful when deleting sprites.

If you make a mistake, double-click the level in the file browser to reload it from the last time it was saved.

Creating a sprite



First, select what kind of sprite you want to create from the drop-list above the sprite list. Then, double-click the map where you want the sprite to be placed. Every game has different sprites.

Selecting sprites



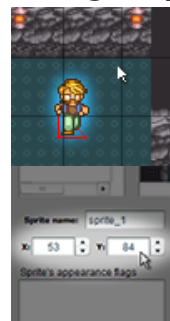
To select a sprite, click its red rectangle, or select its name in the sprite list on the right.

Sprite list



To select a sprite, click its red rectangle, or select its name in the sprite list on the right.

Moving a sprite



Just click and drag a sprite to move it.

If you have overlapping sprites, you can use the sprite list to select a sprite. To move it, click & hold on an empty area of the map and drag.

You can also use the position steppers in the sprite's options to adjust its position precisely.

NOTE: The editor doesn't adjust how sprites overlap, because it's intended to be used for many different types of games, and different games handle overlapping in different ways. For example, RPG's adjust the overlapping based on each sprite's vertical location. Platforming games on the other hand, generally don't care how the sprites overlap.

Deleting sprites



To delete a sprite, select it from the map or sprite list, then press the BACKSPACE or DELETE key. Be careful, there's no "undo" command.

Copying sprites

You can use the clipboard to copy and paste sprites, even between levels. To do it, select a sprite and press CTRL+C. Then move the mouse cursor to where you want the copy and press CTRL+V.

If snapping is on, the new sprite will appear on the nearest tile. Otherwise, it'll be centered on the mouse cursor.

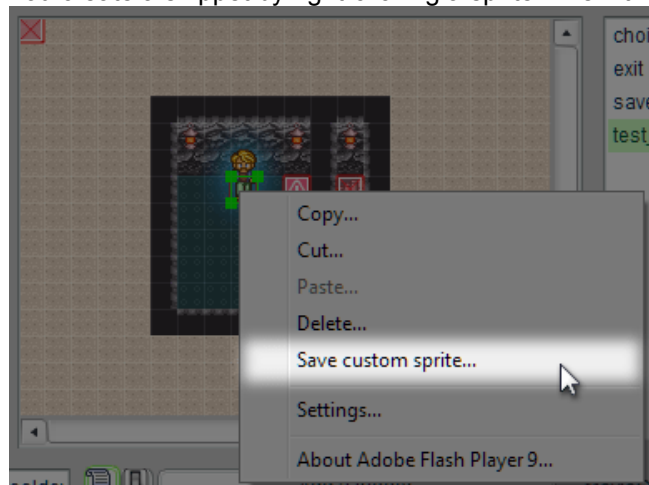
Snippets

You can store a copy of a sprite as a snippet and paste it later on in other levels. The snippet will store the appearance and a copy of all of that sprite's scripts. Modifying the original sprite will NOT update the snippet or any copies of it that you placed in other maps. A snippet is an independent copy.

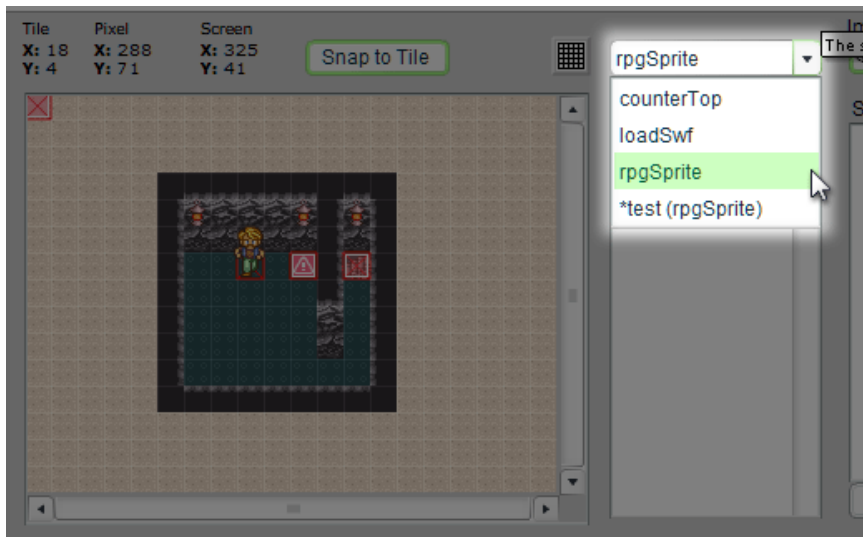
Be careful because you can get yourself into trouble with this. If you place a lot of copies of a sprite and then want to change all of them later, it won't be easy because you would have to manually update all the other copies yourself. For this reason, snippets are best used in combination with the [Copy Sprite](#) feature of an rpgSprite. Before creating a snippet, first create a sprite and tell it to imitate a prototype in the "common" level, and then make a snippet out of this imitation sprite. That way, every copy you place is preconfigured to imitate the prototype. In truth, I really only added the snippet feature as a convenient way to place imitation sprites.



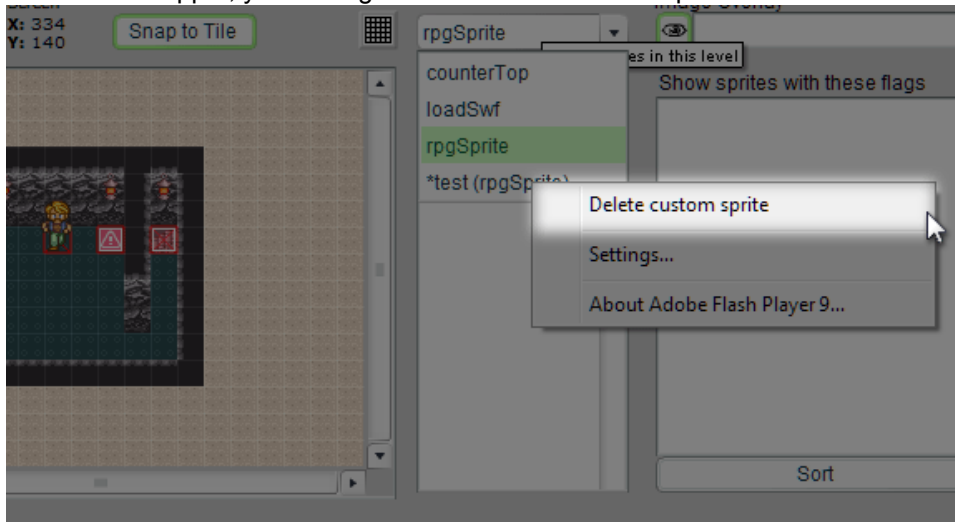
You create a snippet by right-clicking a sprite. Then it will ask you to name the new snippet.



Afterwards, it will appear in the sprite drop-down list as a sprite you can place like any other sprite. These snippets are marked with a * symbol next to their name. And the (parenthesis) indicates what kind of sprite it really is.



To delete a snippet, you can right-click the item in the drop-list and choose to delete it.



Flags

[More info...](#)

RPG Sprites




What is an rpgSprite?

This sprite is used for people and objects in RPG's. It can look like anything, and it can run scripts when certain conditions occur. It single-handedly makes the game an RPG. (Except, you know... without a battle system) Normally, all rpgSprite's have collision. To disable a sprite's collision and allow the player to walk through it, you set its noCollide variable to equal true, by using a setVariable command in the script editor.

[Here's how you turn on noCollide](#)


Edit Modes

There are three things you can edit:

-  this sprite's behavior
-  it's starting appearance
-  The third option allows you to make this sprite imitate another sprite

About Making Scripts

About Scripting

 When you select an existing rpgSprite, it'll display the script editor. A script is a list of things to do. In the list, you'll see 3 triggers. These are reactions that contain things for this sprite to do.

First, you open one of the triggers, such as talk, then you click the <> inside of it, which will make a bunch of buttons appear on the left. Each of those buttons is something you can make the sprite do.

[About script commands](#)

auto

This script will immediately run when the sprite appears, usually when the player enters the map. If this sprite has a flag condition, this script will run when the sprite appears.

This script does not pause the player, so you could have this sprite doing something while the player is walking around.

talk

This script runs when the player talks to this sprite. The player will remain paused until the script is done. This script is also used to make objects react, such as switches.

collision

This script runs when the player bumps into this sprite. This is often used to send the player to another map, such as when they enter a doorway or walk to the edge of a map. You'd simply put an invisible sprite there with a collision script, containing a teleport command.

If this sprite's **noCollide** setting is on, the player will need to walk on top of this sprite to trigger a collision reaction. So this trigger can also be used for things like floor switches too.

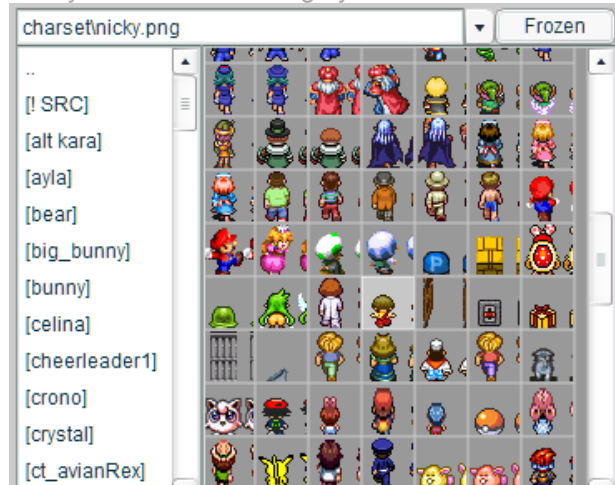
[Here's how you turn on noCollide](#)

Sprite Appearance

Appearance

When you first create an rpgSprite, you'll immediately be able to change what it looks like.

First you click on the image you want to use



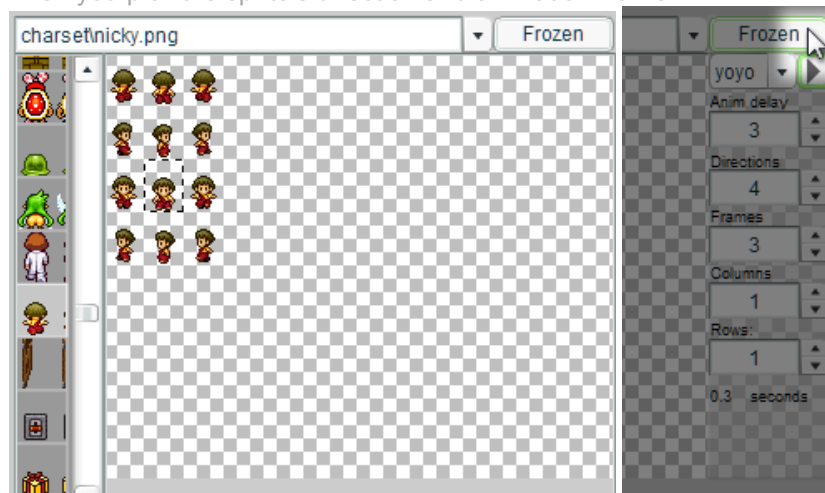
At first, you'll see a preview of every image file within the **charset** folder.

The text at the top shows the path to the currently selected file or folder.

The list on the left side shows all the folders inside of the charset folder. Double-clicking on one of these folders will go inside of it, showing you all the image files it contains.

The thumbnails on the right represent each of the image files. Clicking on one of them will select the file and display the full image.

Then you pick the sprite's direction and animation-frame



When viewing the full image, you can choose the exact animation frame you want to use.

If you hover over the "Frozen" button on the right, this sprite's settings will appear. These control how many animation frames and directions this sprite has, as well as how fast it should animate. The sprite's image on the map will update in real-time to show you what it looks like while you modify its settings. When you select a new image file, you'll need to adjust these settings to tell the editor how this sprite should be displayed. The editor will remember this information the next time you choose this image file.


Clicking on the "Frozen" button changes it to "Animating"

Clicking it again will toggle it back to "Frozen"

This allows you to make a sprite animate as soon as it appears in-game. It's also a useful way to preview a sprite's animation while adjusting its settings on the right.

You can also choose whether to play the animation normally, or in reverse.

 This sprite will use the animation frames from left to right.

 This sprite will use the animation frames from right to left. (backwards)

If you hover over the thumbnails on the left side, the browser will go back to showing you all the files in the current folder.

Copy Sprite

Copy Sprite

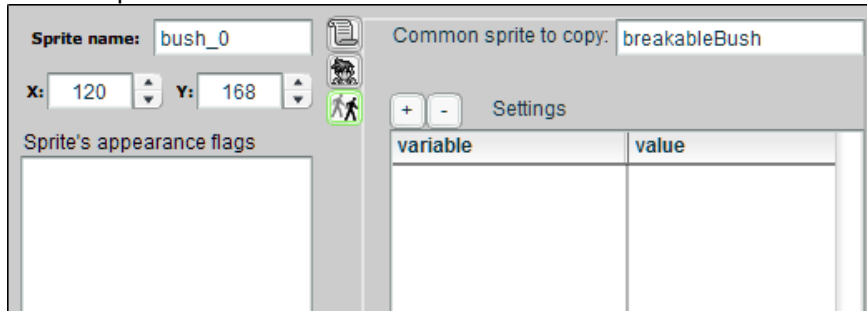
The third option in the sprite's editor tells this sprite to copy the starting appearance and scripts from another rpgSprite when the game is running. This sprite won't exactly mirror what the other sprite does. But it will start off looking the same way, and it will react the same way.

When this is used, this sprite will ignore its own appearance and scripts.

If you don't want to use this feature, just erase the text in the box next to "Common sprite to copy"

This copy feature is useful if you want to place many instances of a common sprite throughout a game (such as breakable bushes) without having to modify ALL of them later on every time you want to change what they do. Just create one master copy in the COMMON level, and then tell each of the in-game sprites to copy that sprite by typing that sprite's name in here.

Normally, this will copy a sprite located in the COMMON level, but you can tell it to copy a sprite within the current level by putting SPRITES. before the sprite's name. SPRITES is the name of a container that the current level uses to store all of its sprites.



Script Commands

Regular Commands



[Pause](#)

This command allows you to pause or un-pause parts of the game engine, such as the player. You can use this to halt the player during **auto** scripts, or allow the player to move during **talk** scripts.

If the game is programmed for it, you can also pause other things such as enemies and or disable menus with this.



[Move](#)

This command can contain a list of movements, that can tell a sprite to walk somewhere. It's also used to change the appearance of that sprite.

This does not pause the script, so it's possible to make multiple characters move at the same time, by using multiple move commands.



[Wait](#)

This tells the script to wait a moment. You can use this for dramatic pauses after a character says something weird.

It can also be used to wait until a sprite finishes moving before continuing the script.

... [Comment](#)

This is used to add notes to your script. It doesn't affect the game, so just use it to plan out your scenes.



[Textbox](#)

This displays a textbox on the screen, which is used for when characters are talking.



[Teleport](#)

This sends the player to another level.



[SWF](#)

This places a SWF file on the screen, or removes an existing one. It's a useful way to add features to your game without reprogramming it.

This can also place picture files, videos (*contained within SWF files*), and internal movieClips.



[Sound](#)

This plays a sound effect.



[Music](#)

This plays a music file. Music continues between levels.



[Fade Music](#)

This makes the current song fade out or fade in.



[Set Variable](#)

This adds or changes variables in the game. Use it to set flags, or make the game remember things for later. This command can also be used to call functions and optionally read values from them. For example...

[Math.random()]

If you put this in the "value" box, The variable will store a random number between 0 and 0.999, which can be stored in a variable. You can also call any function in the game.



[Condition](#)

This is an **if** statement. This checks to see if a variable has a certain value, and then does stuff if it does.



[Loop](#)

This does stuff over and over, as long as a variable has a certain value. To make the loop stop, use a Set Variable command to change the variable's value to something else.

Comment Command

What this does

... This command places a comment into the script. it doesn't do anything. But it helps you plan out what happens.

The screenshot displays the RPG Maker interface. At the top, the 'File' and 'Map' menus are visible. The main window shows a map with a grid and a character sprite named 'thisChar' at coordinates (168, 136). The 'Sprite name' field is set to 'thisChar'. The 'Add a trigger' list includes several commands: 'auto', '<>', 'Disable player', 'move: this', 'wait for music fade', 'text:', 'teleport: test.lvl (10, 8)', 'swf: swf_500', 'sound: sound', and 'music: music'. The 'Comment' command is highlighted in green. The 'message:' field is empty. The interface also shows a 'Sprite's appearance flags' section with 'Sort' and 'Apply new flags' buttons, and a 'Show sprites with these flags' section with 'Sort' and 'Apply Display Flags' buttons.

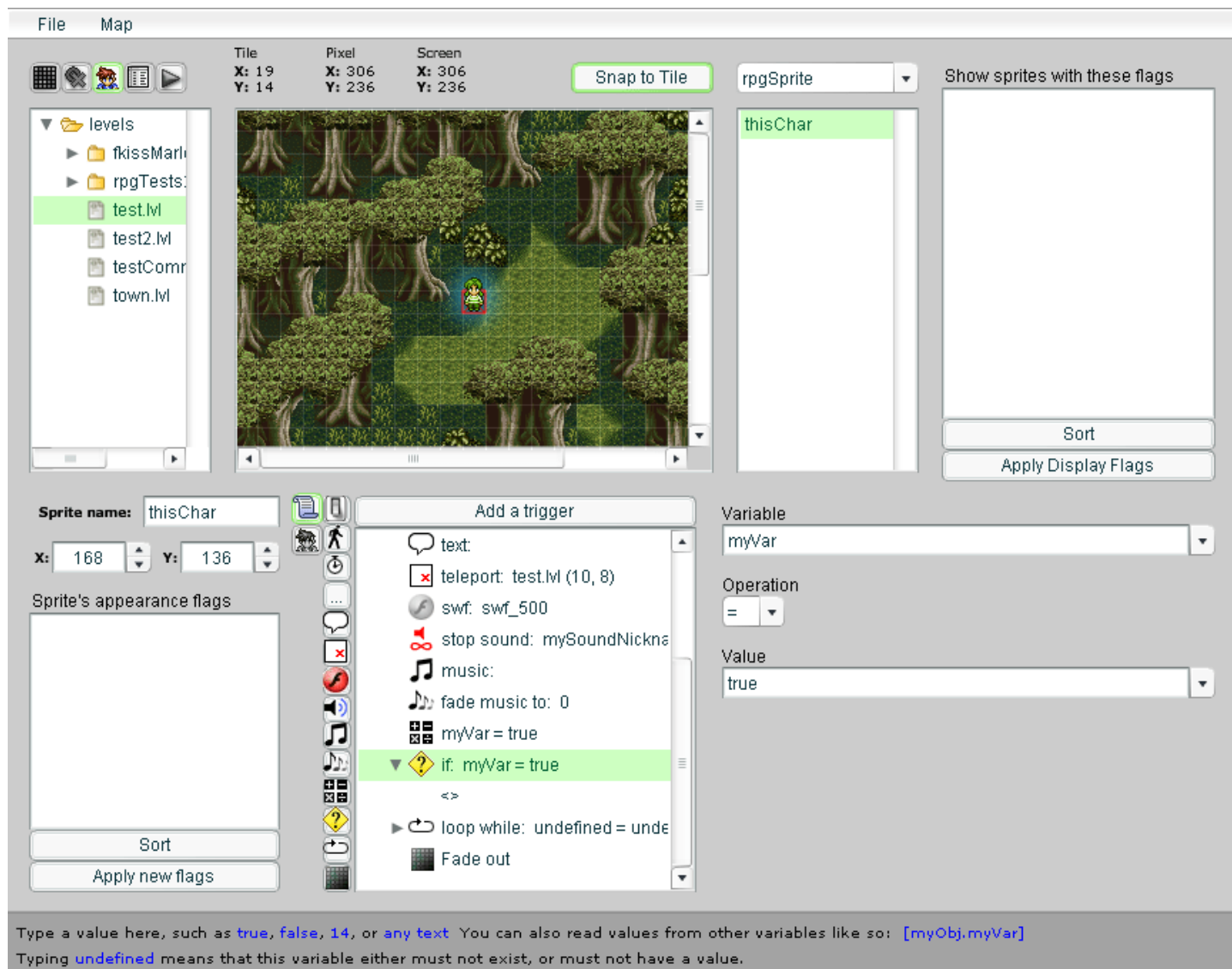
Double-click to create sprites. Click to select one. Drag to move it. Backspace to delete it. Zoom by rolling the mouse-wheel. Pan by pressing the mouse-wheel and dragging.

Condition Command

What this does

This command will optionally run the script inside of it if a variable equals a certain value. You can also use:

= (equal to)
 != (not equal to)
 < (less than)
 <= (less than or equal to)
 > (greater than)
 >= (greater than or equal to)



The screenshot shows the RPG Maker MV interface. The main map area displays a character on a forest floor. The 'Add a trigger' list includes an 'if: myVar = true' command. The configuration panel on the right shows 'Variable: myVar', 'Operation: =', and 'Value: true'.

Type a value here, such as `true`, `false`, `14`, or *any text*. You can also read values from other variables like so: `[myObj.myVar]`
 Typing `undefined` means that this variable either must not exist, or must not have a value.

Comparison tricks

If you're creative, you can do a lot of things with condition commands.

`myVar != true`

This will run a script if `myVar` doesn't exist, if it's false, or if it equals anything other than the word "true". It encompasses much more than `myVar = false`.

`myVar != undefined`

This detects whether `myVar` exists. The script will run if it does exist.

if `myNumber > 1`, set it to 0

if `myNumber = 0`, do one thing

if `myNumber = 1`, do another

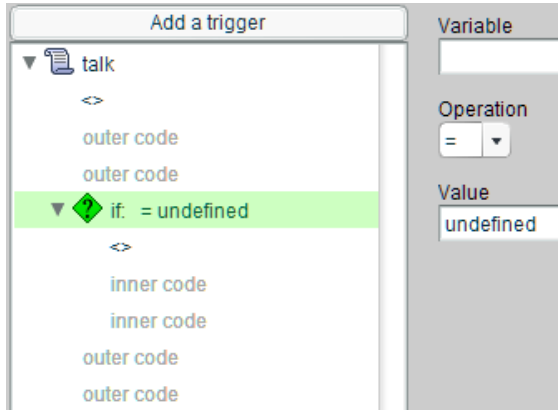
A series of condition commands like this will allow you to toggle a value between 0 and 1 just by adding 1 to it. This is useful for ON/OFF switches

Organizing code

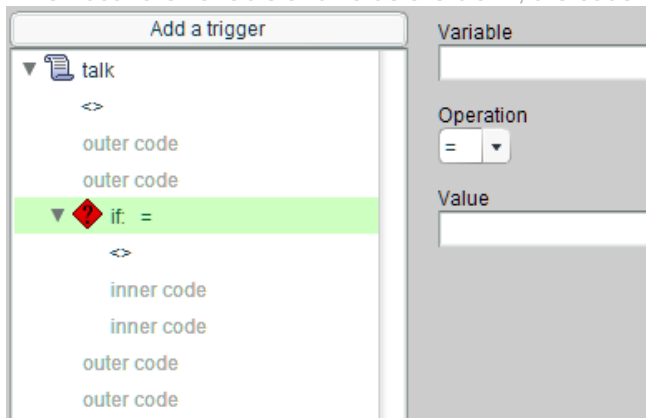
You can also have comparisons that will always be true or always be false. True comparisons always run, and false comparisons never run.

That might seem pointless, but it can be a useful way to organize code since you're basically treating a condition command like a box of code. This also allows you to easily disable the entire block of code if you're testing something or trying to decide between different versions of a script.

When the variable is blank and the value is undefined, the code inside will always run



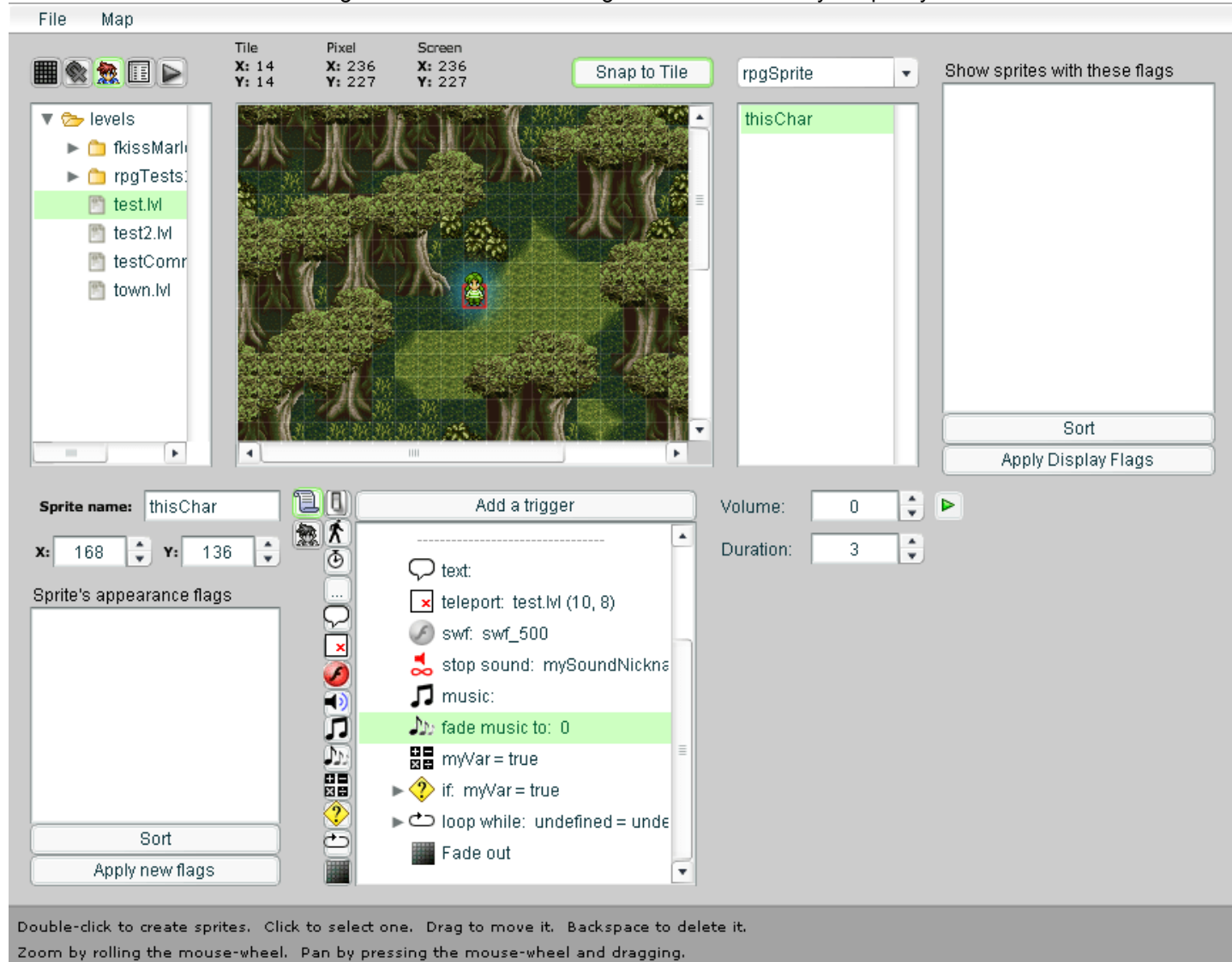
When both the variable and value are blank, the code inside will NEVER run



Fade Music Command

What this does

This command allows you to make music fade in or out. It'll affect whatever song is currently playing. It doesn't have to be all-or-nothing. It'll fade the current song to the volume level you specify.



The screenshot shows the RPG Maker interface with the 'Fade music to: 0' command selected in the event command list. The interface includes a file browser on the left, a map view in the center, and a command list on the right. The 'Fade music to: 0' command is highlighted in green. The volume is set to 0 and the duration is set to 3 seconds. A play button is visible next to the duration field.

Double-click to create sprites. Click to select one. Drag to move it. Backspace to delete it. Zoom by rolling the mouse-wheel. Pan by pressing the mouse-wheel and dragging.

More info

The duration setting tells it how many seconds the fade should take. To preview the effect, click the play button. Previewing only works if a [music](#) command exists in the script somewhere above this command. In the game itself, this won't matter.

Loop Command

↪ What this does

This command allows you to run a set of commands over and over. It will continue to run them as long as the variable condition in this command is true. It's sort of like a [condition](#) command except that it'll repeat over and over.

When the loop command is running, the game will repeatedly run the commands inside of it. When the loop command stops running, the game will run the script commands that come AFTER it.

To make the darn thing stop, you need to change the value of the variable it's looking at so that it's no longer equal to the value this command is looking for.

For example, if you tell it to repeat as long as a variable named myVar = 7, it'll run until you change the contents of myVar to something else.

The screenshot shows the RPG Maker engine interface. At the top, there are menu options for 'File' and 'Map'. Below that, there are status indicators for 'Tile', 'Pixel', and 'Screen' with their respective X and Y coordinates. A 'Snap to Tile' button is visible. The main area is a grid-based map editor showing a forest scene with a character sprite named 'thisChar' at coordinates (168, 136). The 'Sprite name' field is set to 'thisChar'. The 'Add a trigger' list contains several commands: 'teleport: test.lv (10, 8)', 'swf: swf_500', 'stop sound: mySoundNickna', 'music: fade music to: 0', 'myVar = true', 'if: myVar = true', 'loop while: undefined = unde', 'wait for 0.03 sec', and 'Fade out'. The 'loop while' command is currently selected. The 'Variable' and 'Value' fields are both set to 'undefined'. The 'Operation' field is set to '='. The 'Show sprites with these flags' section is empty. At the bottom, there is a prompt: 'Select a level to edit, or create a new one.'

Tricks

Making an infinite loop.

If you actually WANT the loop to run forever (as long as the player is on the map) then type the word undefined into BOTH the variable and value boxes. This works because there's normally no variable by the name of "undefined", so its value will be undefined. If that's what you tell the loop command to look for, then it'll run as long as that variable doesn't exist. This is how loop commands are set when you first create them.

Looping a few times.

To do this, you need to first create a variable and put a number into it. Then, you create the loop command and tell it to run as long as that variable's value is > 0. Finally, you need to put a variable command INSIDE the loop's script that decreases the variable's value by 1. What will happen is that the variable will repeatedly count down until it reaches 0. Then the loop will stop.

Moving movieClips.

This is an advanced trick, but you can use a loop command to make a movieClip move until it reaches a certain

location. To do this, you would type in the movieClip's `_x` or `_y` property into the variable box like so:

```
OVERLAY.myClip._x
```

Then you would type in the destination coordinate into the value box, and have the loop run until it passes that location. Finally, within the loop, you need to create a variable command that increases or decreases that movieClip's `_x` position. The loop will run until the movieClip passes the desired location. I say "passes" because, unless you're moving the movieClip by only 1 pixel each time, it'll skip ahead a few pixels each time the loop runs and may very well skip over the exact location you specify. If you want the movieClip to stop at a specific location, then put another variable command **AFTER** the loop and have it set the movieClip's location to an exact value.

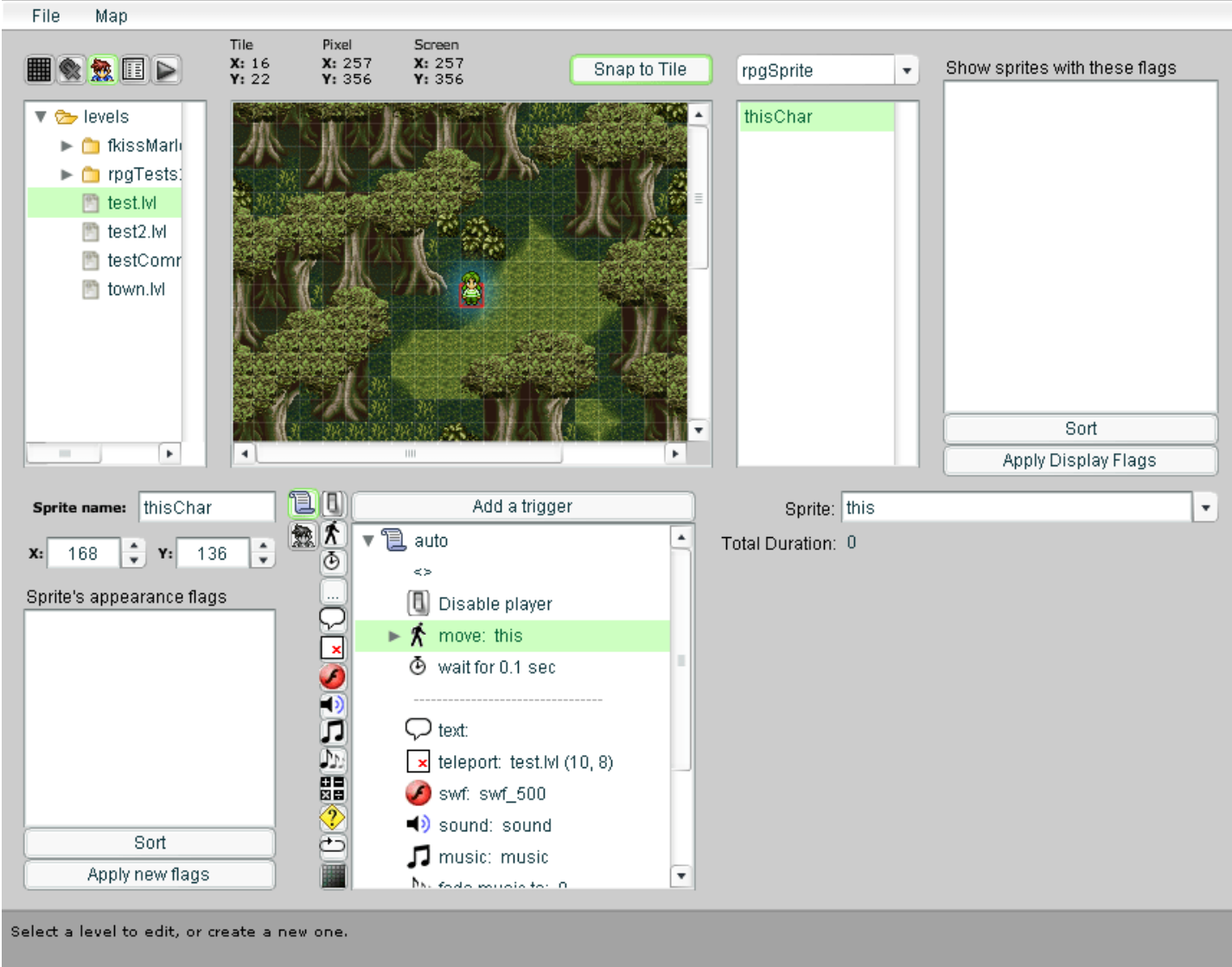
Move Command

What this does

This command tells the game to move a character on the map. The first thing you do is tell this command which character or object you're moving. Then you click the little arrow next to this command to open it up and it'll display some <>.

When you click on the <>, the buttons on the left will change and you'll be able to place a bunch of movement commands inside of this.

If you type the name of a character that doesn't exist on the map, the editor will pretend it does and display it as a transparent sprite on the map. This is helpful when you're moving the player or characters generated by the game.



The screenshot displays the RPG Maker editor interface. At the top, there are menu options for 'File' and 'Map'. Below the menu is a toolbar with icons for grid, tile, character, and movement. The main map area shows a forest scene with a character sprite. To the left is a file tree with levels like 'test.lv' and 'town.lv'. On the right, there are settings for 'Snap to Tile', 'rpgSprite' (set to 'thisChar'), and 'Show sprites with these flags'. Below the map, there are fields for 'Sprite name: thisChar', 'X: 168', and 'Y: 136'. A 'Sprite's appearance flags' section is also present. The 'Add a trigger' list shows various actions, with 'move: this' highlighted. Other triggers include 'wait for 0.1 sec', 'text', 'teleport: test.lv (10, 8)', 'swf: swf_500', 'sound: sound', and 'music: music'. At the bottom, there is a 'Sprite: this' dropdown and 'Total Duration: 0'.

Waiting for movements

Normally, the script will continue running while the movements inside this command are occurring. To make it wait, place a [wait command](#) after this (outside of this move command) and tell it to wait for the sprite that's being moved.

What's good about this is that you can have multiple characters move at the same time, and then wait for each of them. Just make sure they finish moving before telling them to move again, or things might go wrong.

Movement Commands

[relative movement](#)

This command makes the character walk a certain number of steps in any direction. Diagonal movement is also possible.

[absolute movement](#)

This command makes the character walk to an exact spot on the map.

 [animate](#)

This command makes a character start or stop animating.

 [appearance](#)

This command changes what the character looks like. It can also be used to change how fast they animate.

 [look](#)

This command makes the character look in a certain direction.

 [look at](#)

This command makes the character look at another character or object.

 [wait](#)

This command will pause the script for a certain amount of time.

Music Command

🎵 What this does

This command changes the background music. If you clear the Song File box completely and press ENTER, then this command will stop the music.

The screenshot shows the RPG Maker MV interface with the Music Command configuration panel open. The interface includes a top menu bar with 'File' and 'Map', a toolbar with various icons, and a central map view showing a character on a forest floor. The left sidebar displays a file tree with 'levels' containing 'test.lvl', 'test2.lvl', 'testComr', and 'town.lvl'. The right sidebar shows a list of sprites with 'thisChar' selected. The bottom panel is divided into several sections: 'Sprite name' (thisChar), 'X' (168) and 'Y' (136) coordinates, 'Sprite's appearance flags' (empty), 'Add a trigger' (a list of commands including 'music: music\CaveDig_mair'), 'Song file' (music\CaveDig_main.mp3), and 'Volume' (100). The 'Add a trigger' list includes: text, teleport: test.lvl (10, 8), swf: swf_500, stop sound: mySoundNickne, **music: music\CaveDig_mair**, fade music to: 0, myVar = true, if: myVar = true, loop while: undefined = unde, and Fade out. The bottom status bar reads 'Select a level to edit, or create a new one.'

Song intro files.

All songs will play in a loop. It's possible to also give a song an intro which will play before the main loop of the song. To designate which file is the intro, you need to add `_intro` to the end of the filename. So in the music folder, you'll have something like this: `mySong.mp3` `mySong_intro.mp3`

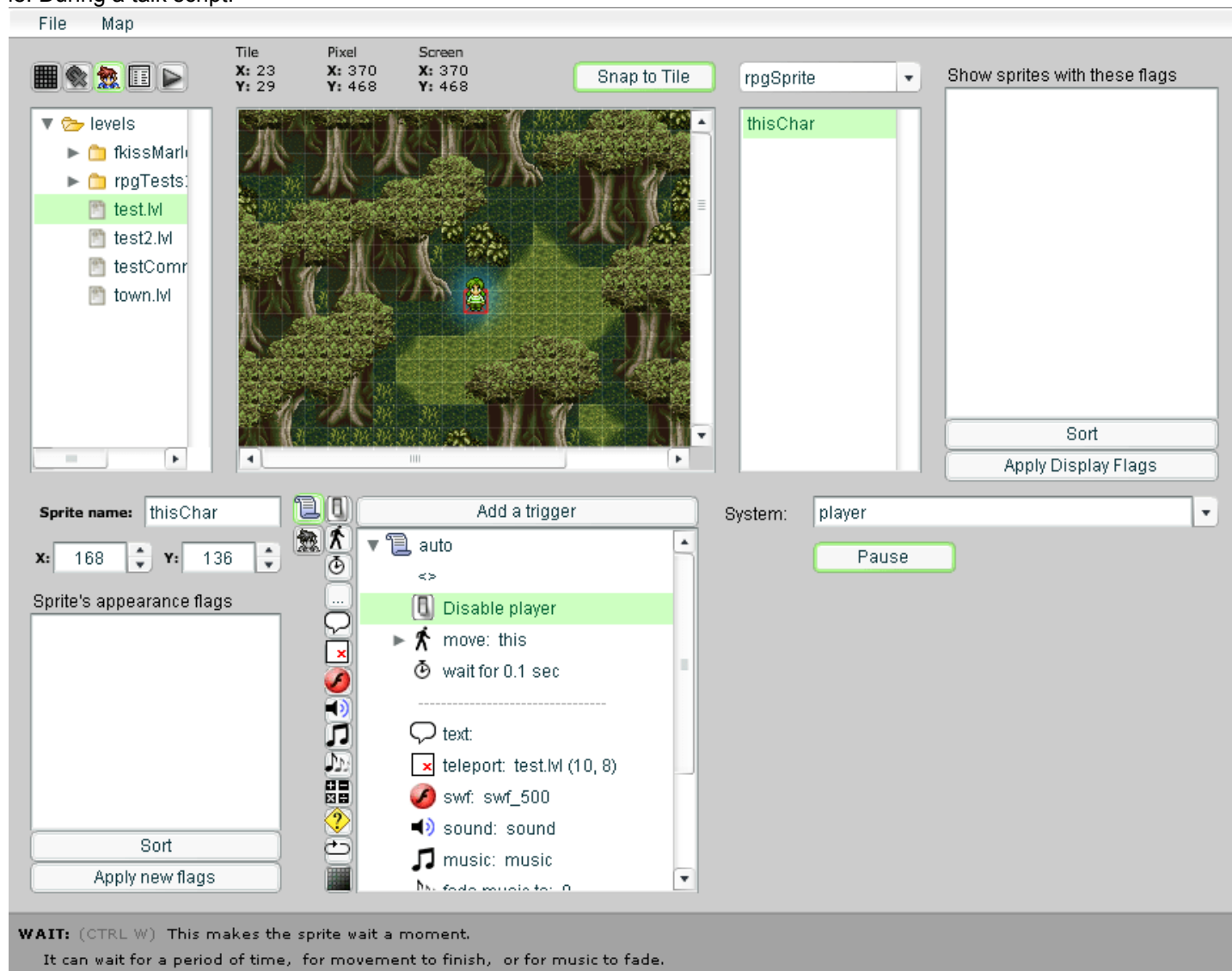
Aside from `_intro`, the file names need to exactly match. The game will automatically detect intros and play them.

Pause Command

What this does

This command allows you to pause or un-pause parts of the game engine, such as the player, other sprites, or disable menus. This is useful when you want to disable the player during cutscenes, which typically occur in auto or collide scripts, or allow the player to move while talking to someone.

ie: During a talk script.



The screenshot shows the RPG Maker engine interface. The top menu bar includes 'File' and 'Map'. Below the menu bar, there are several toolbars and panels. The 'Tile' panel shows coordinates: X: 23, Y: 29. The 'Pixel' panel shows coordinates: X: 370, Y: 468. The 'Screen' panel shows coordinates: X: 370, Y: 468. A 'Snap to Tile' button is visible. The 'rpgSprite' dropdown menu is set to 'thisChar'. The 'Show sprites with these flags' panel is empty. The 'Sprite name' field is set to 'thisChar'. The 'X' and 'Y' coordinates are 168 and 136 respectively. The 'Sprite's appearance flags' panel is empty. The 'Add a trigger' panel shows a list of triggers: 'auto', 'Disable player', 'move: this', 'wait for 0.1 sec', 'text:', 'teleport: test.lvl (10, 8)', 'swf: swf_500', 'sound: sound', and 'music: music'. The 'System' dropdown menu is set to 'player'. A 'Pause' button is highlighted in green. At the bottom, there is a 'WAIT:' section with the text: '(CTRL W) This makes the sprite wait a moment. It can wait for a period of time, for movement to finish, or for music to fade.'

More info

This command can pause other parts of a game too, if the game is programmed to allow it. Internally, any pausing in a game is handled by the loop system. Any movieClips handled by that system can be paused. When you use this command, you give it the name of the loop-set to pause or un-pause.

Set Variable Command

What this does

This command stores some text in a **VARIABLE**. It can contain any word, phrase, or number you want. It can also be set to true or false to turn things on or off.

This command also has the ability to create objects and run code.

The screenshot shows the RPG Maker interface with the 'Set Variable' command selected in the 'Add a trigger' list. The command is configured as follows:

- Sprite name:** thisChar
- X:** 168, **Y:** 136
- Variable:** myVar
- Operation:** =
- Value:** true

The 'Add a trigger' list includes the following items:

- text:
- teleport: test.lvl (10, 8)
- swf: swf_500
- stop sound: mySoundNickna
- music:
- fade music to: 0
- myVar = true** (highlighted)
- if: myVar = true
- loop while: undefined = unde
- Fade out

The interface also shows a map view with a character on a forest floor, a file explorer on the left, and various toolbars and buttons like 'Sort' and 'Apply Display Flags'.

What is variable?

When you want the game to remember something, you give that information a name. Then you can use that name to recall what was stored.

What are variables for?

Variables are used to remember stuff, such as a character's name, their HP, or whether a story event has occurred. Variables can even change the game's settings and behavior, such as how fast the player walks.

For example?


The screenshot shows the 'Set Variable' command configuration for a character's name:

- Variable:** name1
- Operation:** =
- Value:** Marie

The 'Add a trigger' list includes the following items:

- auto
- talk
- name1 = Marie** (highlighted)
- text: Hi, my name is [name1]
- collision

What if you want the game to remember the player's name, and re-use it later. This is useful in case you ever want to change the name later, without having to change all the text in the game.

To do this, you click on the button that looks like this. 

Then on the right under variable, you type a name for this information. And under value, you type in something for it to remember. From that point on, you'll be able to use the variable name to recall what you typed in.

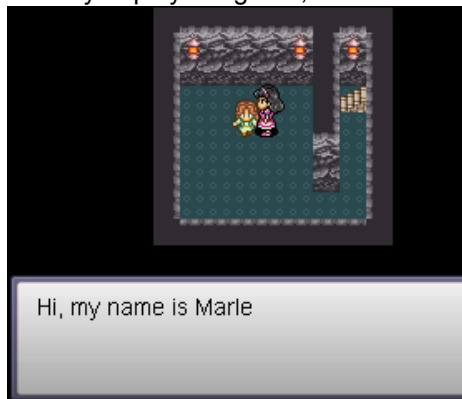
You'll still be able to change the variable's contents later or even erase the variable itself if you want.

To recall the word you typed, you put the variable's name in brackets. So if you want to display a character's name in a text box, you write this:

Hi, my name is [name1]

Assuming that "name1" is what you named the variable.

When you play the game, it'll look like this:



Just make sure that you type the variable's name EXACTLY the same way as it was written. Variable names are case-sensitive, so it's generally a good idea to always make them lowercase. The contents of the variable, on the other hand, can be whatever you want.

Rules for a variable's name

A variable's name should never contain any spaces. Also, the name should not start with a number. Finally, don't use weird characters or punctuation.

Where can I use variables?

Most script commands allow you to use variables if you type the variable's name with [] brackets around it. Basically, anywhere you can type text.

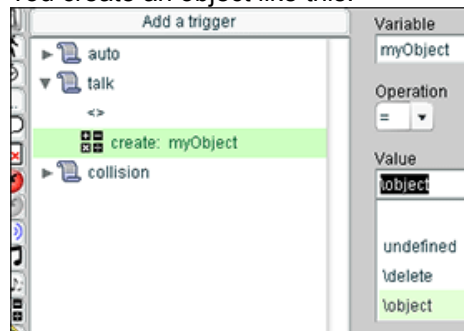
Advanced usage:

Using objects:

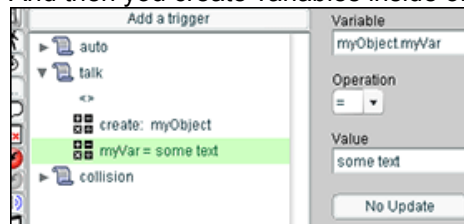
The set-variable command can also create objects, which can help you organize variables by grouping related ones together.

An object is like a box that can contain a bunch of variables. You need to create an object before you can create variables inside of it.

You create an object like this:

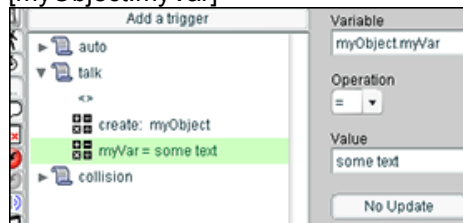


And then you create variables inside of it like this:



To read the value of a variable inside of an object, you type the object's name, a period, and end it with the name of a variable inside it:

[myObject.myVar]

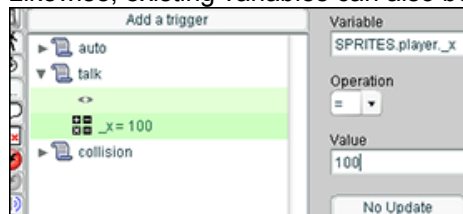


You probably won't need to use objects very much, but accessing a setting or sprite in the game usually requires digging into a pre-existing object.

For example:

[SPRITES.player._x]

Likewise, existing variables can also be altered like so:



This example would make the player instantly jump to a spot that's 100 pixels away from the left side of the map. And as you can see, objects can contain other objects.

Pre-existing objects:

The RPG game engine already contains many variables and objects. I won't go over all of them, but the more you know about the game's code, the more you can mess with it. These are the main objects in the game engine:

Some of these objects are actually movieClips. So you can place SWF files into them using the [swf](#) script command. For example, you would typically put a menu system into the HUD movieClip this way.

The contents of HUD will remain even when you teleport to another level. The PANORAMA, UNDERLAY, and MAP_OVERLAY movieClips are automatically cleared between levels.

[RAM](#)
[ROM](#)
[MUSIC](#)
[SOUND](#)
[PANORAMA](#)
[UNDERLAY](#)
[SPRITES](#)
[MAP_OVERLAY](#)
[SCREEN_OVERLAY](#)
[HUD](#)
[LEVEL](#)

Deleting variables or objects:

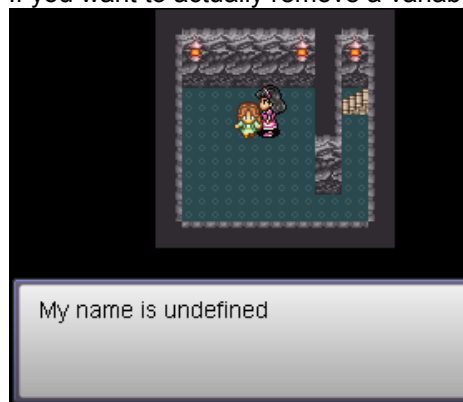
The set-variable command can delete both variables or objects. You specify the thing to delete, and set the value to \delete, which can be selected from the list.

Missing variables:

If the variable you're trying to access doesn't exist, its value will be the undefined. This special value is not treated like text. It's sort of like false, but it typically means that something is missing.

If you set an existing variable to the word undefined, the variable itself will still exist, but the game will pretend it doesn't.

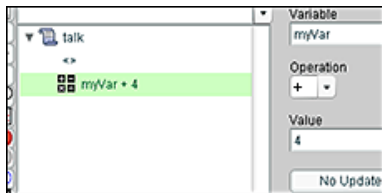
If you want to actually remove a variable you need to use /delete.



Changing numbers within variables:

If a variable contains a number, you can increase or decrease its value without knowing what the value is.

To do this, you create a setVariable command in the script, type the name of the



variable, then select the + or – symbol from the box in its settings. The number you type into the value box, will be added to the number currently stored in the variable. Also, adding a negative number is the same as subtracting.

Calling functions:

Functions are pieces of programming that can be triggered. They can only be created in Actionsript, but the rpgSprite can trigger pre-existing functions, including many that are built into flash itself. For example, you can generate a random number by putting this in the value field of the set-variable command:

```
[Math.random()]
```

The resulting number will get stored in the variable you specified. This also works in text boxes. Not all functions spit out a result, so you can leave the variable box empty if you want.

A function is accessed by name the same way a variable is. The difference is that you put paranthesis after the name to trigger its code. If the name you specified isn't actually a function, adding paranthesis won't do anything.

Furthermore, it is possible to send parameters to a function like this:

```
[Math.round(1.6)]
```

This would spit out the number 2.

And finally, it's possible to use a variable as a parameter like this:

```
[Math.round([myVar])]
```

In this case, the value of myVar will get rounded by the function, so if myVar contains the number 2.6 then Math.round will turn that number into 3.

Built-in sprite variables:

Characters and objects in the game have some built-in variables that control their behavior. You access these variables using the word this followed by a period, and then the variable name.

this.noCollide

If you set this variable to **true**, the player will be able to walk through this character or object. This variable is normally set to false.

this.overAll

If you set this variable to **true**, then this character will always overlap all other characters, no matter what its position is. This variable is normally set to **false**. Normally, characters overlap each other based on their vertical position on the map. This variable overrides this.

this.underAll

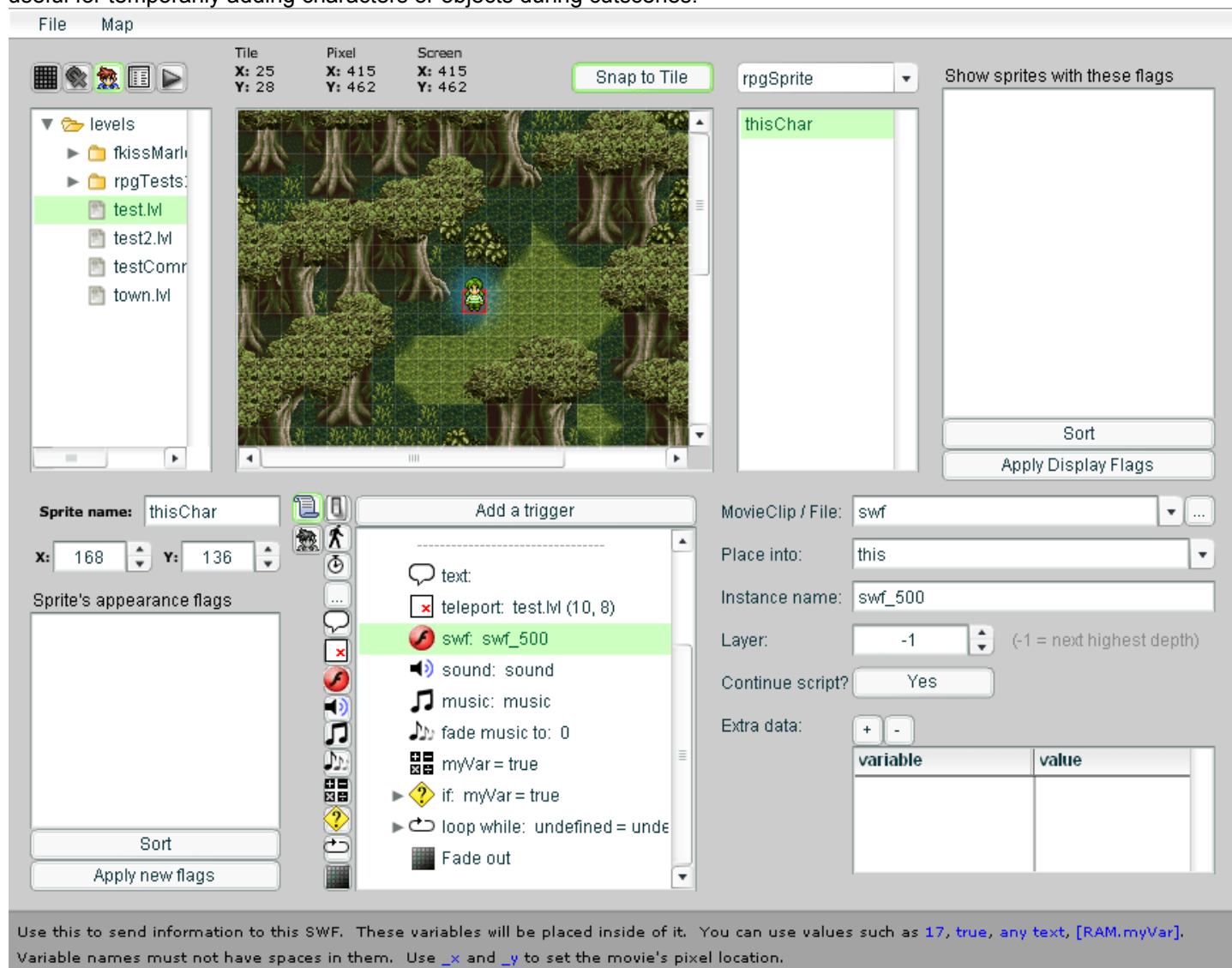
If you set this variable to **true**, then this character will always get overlapped by all other characters, no matter what its position is. This variable is normally set to **false**. Normally, characters overlap each other based on their vertical position on the map. This variable overrides this.

SWF Command

What this does

Flash creates animations and games. These are stored as SWF files. This command can place an external SWF file inside of the game, or remove one from the game that you have already placed. The game itself is a SWF file, and other SWF files can be placed inside of it at any time. These files can contain pretty much anything, but are typically animations or code. This is useful for displaying animations and videos, or for temporarily adding menus and features to the game.

In addition to SWF files, this command can also load image files like JPG's and PNG's and place them inside the game. It can also place internal movieClips, which are like SWF files that are stored inside of the game itself that can be placed at any time without loading an external file. To place one of these, you don't need to type a file-path, you just need to type the name of it. For example you have "black" (without the quotes) which places a black rectangle that you can use to cover up parts of the map. You have "empty" which places an invisible object that you can place other things inside of and move around. And you can also place "rpgSprite" which basically adds a new character to your map that you can pose and move around like any other rpgSprite. However it won't have any "talk" scripts or anything. It's mostly just useful for temporarily adding characters or objects during cutscenes.



The screenshot shows the RPG Maker interface for the SWF Command. The top bar includes 'File' and 'Map' menus. Below the menu bar, there are icons for various actions and a 'Snap to Tile' button. The main map area shows a forest scene with a character. On the left, a file explorer shows a folder named 'levels' containing several level files. The central panel is titled 'Add a trigger' and lists various triggers, with 'swf: swf_500' selected. To the right of the trigger list, there are settings for the SWF file, including 'MovieClip / File', 'Place into', 'Instance name', 'Layer', 'Continue script?', and 'Extra data'. The 'Extra data' section has a table with columns 'variable' and 'value'.

Use this to send information to this SWF. These variables will be placed inside of it. You can use values such as **17**, **true**, **any text**, **[RAM.myVar]**. Variable names must not have spaces in them. Use **_x** and **_y** to set the movie's pixel location.

How to use this

MovieClip / File:

First, you select the SWF file you want to place into the game. If you set this to a blank value, then this command will remove an existing SWF file from the game instead.

Place into:

Then you tell it where you to put the loaded file...
or where the existing file is located

Instance name: swf

...and what to call it.

or what the existing file was named when it was placed within the game

Layer: -1

You can optionally specify a layer to put it on to control overlapping with other SWF files. Typing -1 tells it to put it on top of everything else.

Continue script? Yes

And finally you choose whether you want the script to continue running after placing this, or you want it to stop and wait for this SWF animation to finish. (AKA: Wait until it reaches its last frame of animation)

Extra Data: +

You can also pass settings to the imported SWF file or movieClip by adding variables to the box at the bottom. Most of the time, you'll use this to set the file's position on the screen by setting its `_x` and `_y` variables. Some SWF files have default settings which will show up here when the file is selected.

Details

MovieClip / File: swf

You can place SWF files, image files, or internal movieClips.

SWF swf\myFile.swf

IMAGE swf\myFile.jpg

MOVIECLIP enemySprite

When you place internal movieClips, you type in a linkage name instead of a file path.

Place into: HUD

There are a number of pre-defined movieClips in the game where you can place SWF files. You can almost think of them as graphics layers.

[LEVEL](#)

[HUD](#)

[SCREEN_OVERLAY](#)

[MAP_OVERLAY](#)

[this](#)

[SPRITES](#)

[UNDERLAY](#)

[PANORAMA](#)

Instance name: swf

This is the internal name of the SWF file you're placing. If you give it the same name as something else, it will usually replace it. You'll use this name to target or remove this SWF animation later. You can also access stuff inside of the placed SWF animation using the name you give it, the same way you can with sprites. For example, the [variable](#) command would access variables inside of a placed SWF file like so:

HUD.swfName.myVariable

If you leave the name blank, then you won't be able to remove the SWF later because it'll be given a random name.

Layer: -1

Layers control which order things will overlap. If you place something into the game's SPRITES movieClip, the layering will be handled automatically by the game. Be careful not to give this SWF file the same layer number as another SWF or sprite within that movieClip, or it'll get replaced by this one. If you're not sure what layer to use, just type -1 and it'll safely be placed above everything else in that movieClip. This is usually what you want unless you have a very specific layout in mind where you care about which items overlap which other items.

Continue script? Yes

This basically just waits until the SWF file reaches its last frame. If you're placing a special effect such as an explosion, you might want the script to continue while it animates. On the other hand, if you're placing a menu or dialog box that the player interacts with, then you'll probably want the script to stop and wait until it's finished. All the textboxes in the game are actually SWF commands that load a textbox.swf file into the game and then wait for it to close. When they close, they move to their last animation frame to tell the game that they're "done"

Extra Data: +

If you were placing your own custom-made textbox. This is where you'd tell it what to say. I've already included a bunch of SWF files in the "swf" folder. Most of them have default settings that are automatically filled-in for you when you select that file. In addition, you'll usually see some descriptive text displayed under the settings. If you hover your mouse over this, it will expand so you can read it.

Animate

► What this does

This command makes a character start or stop animating.

The screenshot shows the RPG Maker interface with the 'Animate' command selected. The 'Sprite name' is 'thisChar'. The 'Add a trigger' list includes: Disable player, move: this, move: 2, 0, moveTo: 14, 8, wait: 0.1 sec, pose: charsetcrown1.png, **start animating**, look: up, lookAt: player, and wait for music fade. The 'Moving' status is active. The map view shows a character on a forest map. The 'Sprite's appearance flags' section is empty.

File Map

Tile X: 31 Y: 26 Pixel X: 496 Y: 431 Screen X: 496 Y: 431 Snap to Tile rpgSprite Show sprites with these flags

levels
fKissMarl
rpgTests:
test.lv
test2.lv
testComr
town.lv

Sprite name: thisChar X: 168 Y: 136

Sprite's appearance flags

Sort Apply new flags

Add a trigger

- <>
- Disable player
- ▼ move: this
 - <>
 - move: 2, 0
 - moveTo: 14, 8
 - wait: 0.1 sec
 - pose: charsetcrown1.png
 - ▶ start animating**
 - look: up
 - lookAt: player
 - wait for music fade

Moving

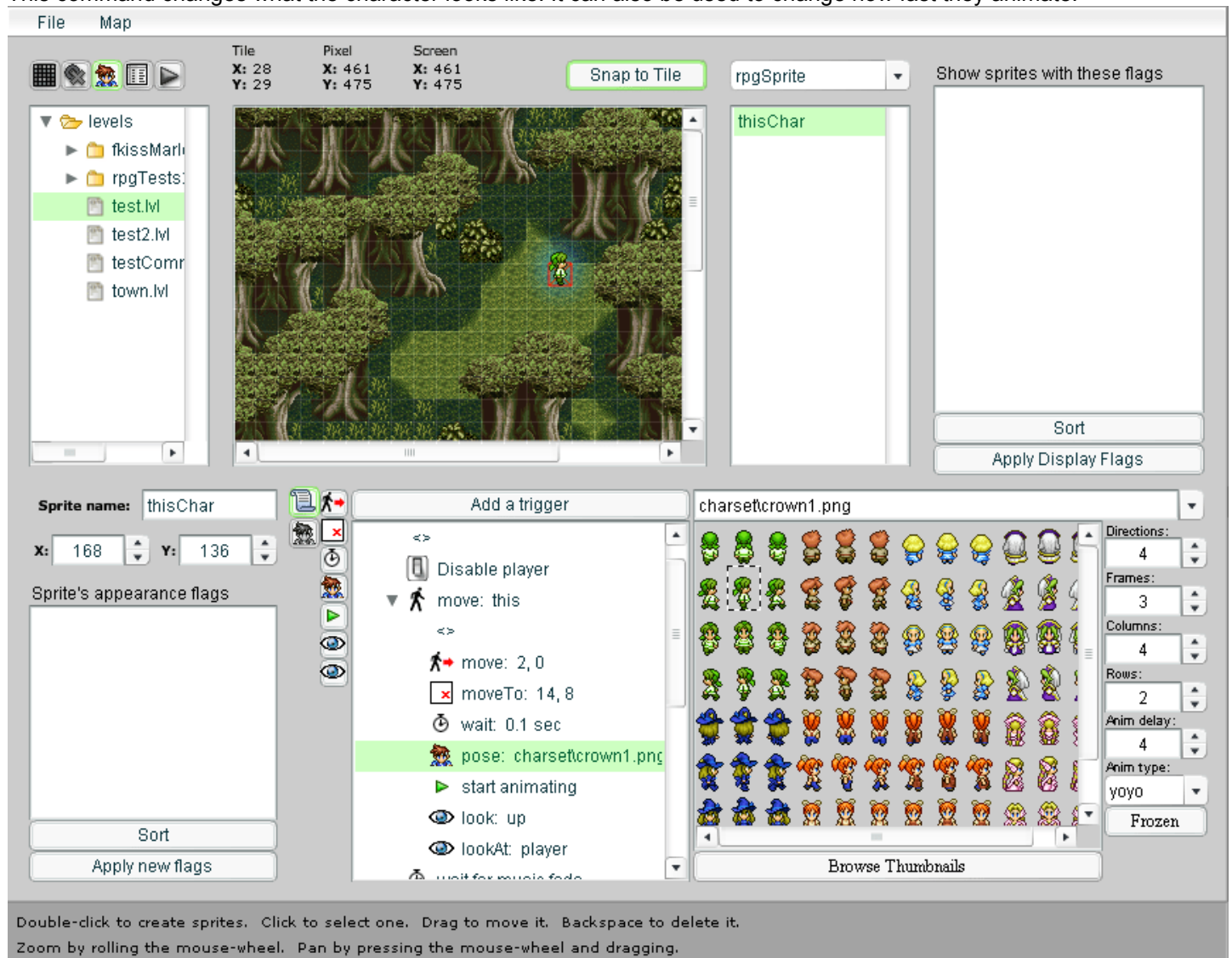
Sort Apply Display Flags

Select a level to edit, or create a new one.

Appearance

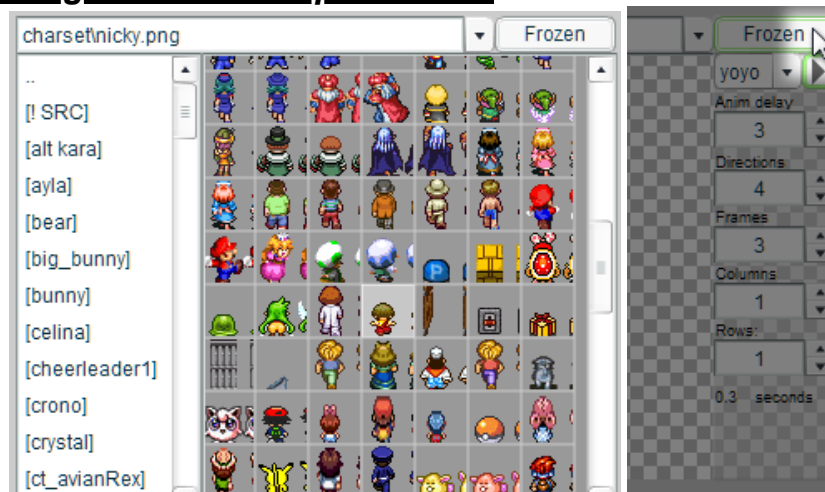
What this does

This command changes what the character looks like. It can also be used to change how fast they animate.



Double-click to create sprites. Click to select one. Drag to move it. Backspace to delete it. Zoom by rolling the mouse-wheel. Pan by pressing the mouse-wheel and dragging.

Using this fucked-up interface

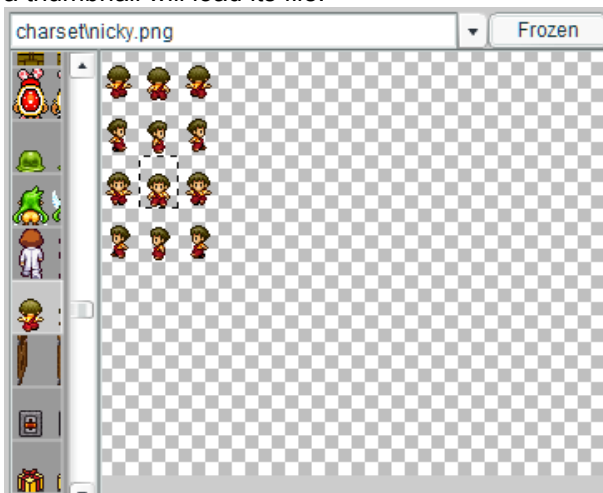


At the top of the settings, you can select a sprite sheet, and on the right, you can adjust the [settings](#) that tell it how many directions and animation frames this sprite sheet has. The sprite's image on the map will update in real-time to show you what it looks like.

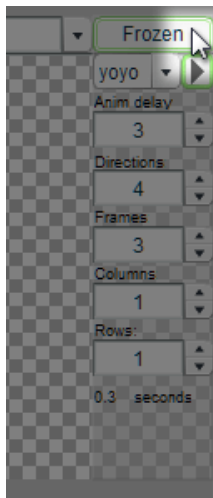
Any changes you make to the settings will be remembered for you, so that when you select that file again, those settings will automatically be used.

The folder browser on the left allows you to easily navigate into and out of folders by double-clicking on them.

To the right of it is the thumbnail browser, which allows you to preview all the sprite sheets in the current folder. Clicking a thumbnail will load its file.



After you click on a sprite file, the interface will change. From here you can choose which direction and animation frame you want to use by clicking on it. An animated rectangle should appear around the animation frame you have selected. If the rectangle appears around the entire image, then you need to adjust the settings to tell the program how many animation frames and facing-directions this image file is supposed to have.

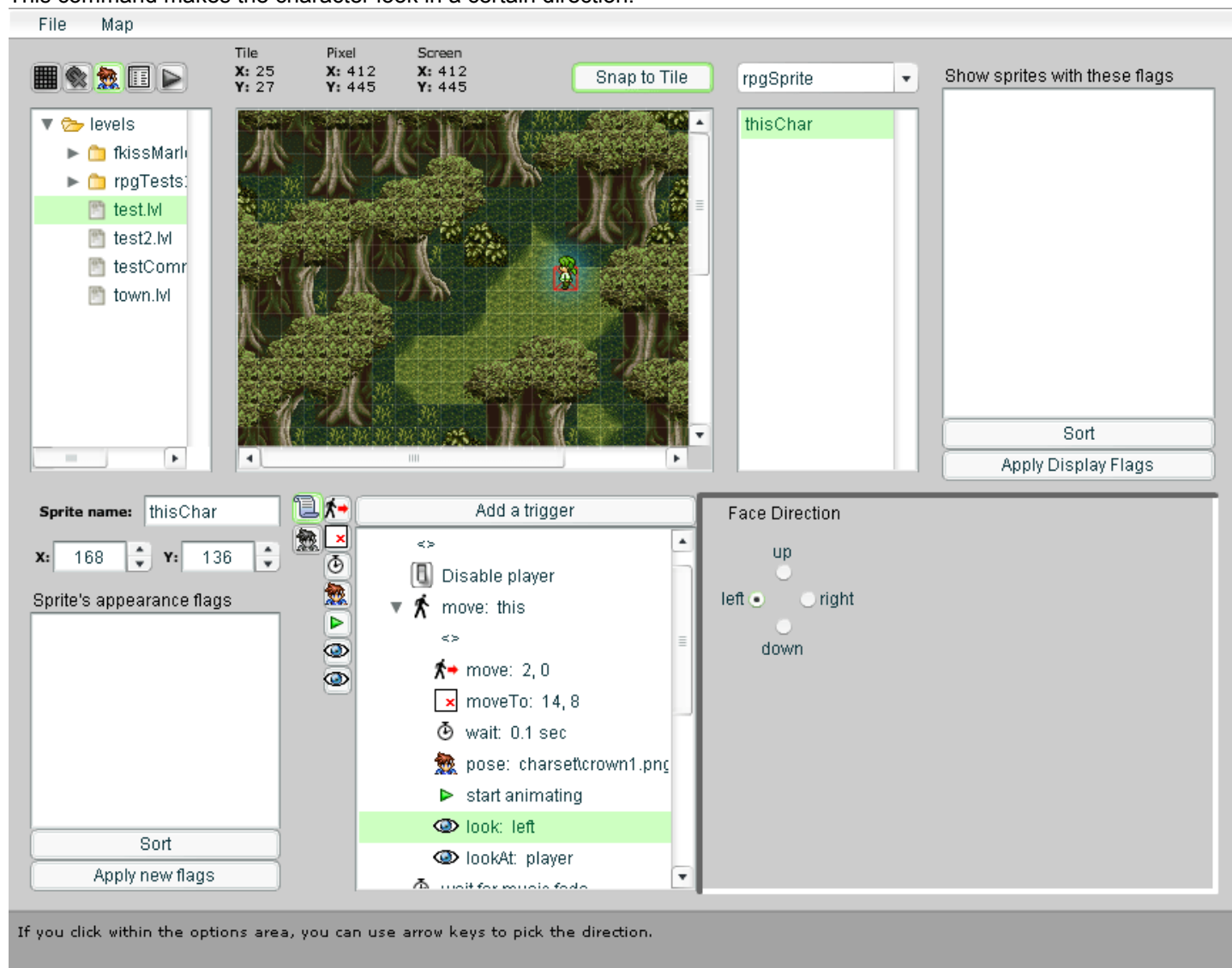


To do that, hover the mouse over the button on the top right to reveal the settings for this file. Each file has its own [settings](#).

Look

What this does

This command makes the character look in a certain direction.



The screenshot displays the RPG Maker interface with the 'Look' command selected in the event command list. The interface is divided into several sections:

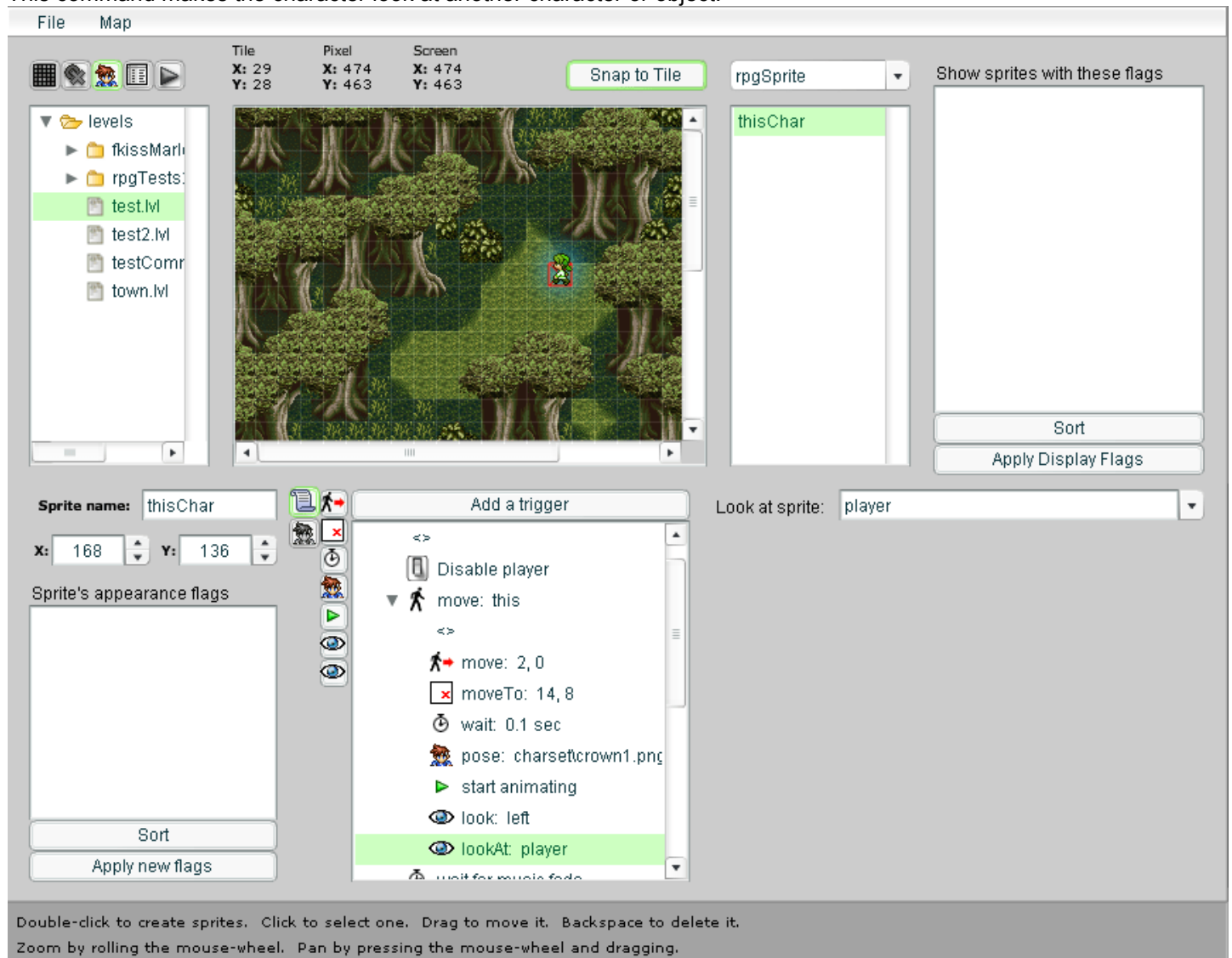
- Top Panel:** Shows the 'File' and 'Map' menus, a toolbar with icons for tile, pixel, and screen selection, and a 'Snap to Tile' button. The current tile is 'rpgSprite' and the selected sprite is 'thisChar'.
- Left Panel:** A file explorer showing a tree structure of levels, with 'test.lvl' selected.
- Center Panel:** A map view showing a character sprite on a grid. The character is positioned at X: 168, Y: 136.
- Right Panel:** A list of display flags for the selected sprite, currently empty.
- Bottom Panel:**
 - Sprite name:** 'thisChar'
 - Coordinates:** X: 168, Y: 136
 - Sprite's appearance flags:** A list of flags for the sprite's appearance, currently empty.
 - Add a trigger:** A list of event commands including 'Disable player', 'move: this', 'move: 2, 0', 'moveTo: 14, 8', 'wait: 0.1 sec', 'pose: charsetcrown1.png', 'start animating', 'look: left' (highlighted), and 'lookAt: player'.
 - Face Direction:** A directional pad with 'left' selected.

At the bottom of the interface, a note states: "If you click within the options area, you can use arrow keys to pick the direction."

Look At

What this does

This command makes the character look at another character or object.



The screenshot shows the RPG Maker engine interface with the 'Look At' command configured for a sprite named 'thisChar'. The interface includes a file browser on the left, a map view in the center, and a command list on the right. The 'Look at sprite' dropdown is set to 'player'. The command list includes 'Disable player', 'move: this', 'move: 2, 0', 'moveTo: 14, 8', 'wait: 0.1 sec', 'pose: charsetcrown1.png', 'start animating', 'look: left', and 'lookAt: player'.

File Map

Tile X: 29 Y: 28 Pixel X: 474 Y: 463 Screen X: 474 Y: 463 Snap to Tile rpgSprite Show sprites with these flags

levels
 ▶ fKissMark
 ▶ rpgTests:
 test.lv
 test2.lv
 testComr
 town.lv

Sprite name: thisChar X: 168 Y: 136 Sprite's appearance flags

Add a trigger

- <>
- Disable player
- ▼ move: this
 - <>
 - move: 2, 0
 - moveTo: 14, 8
 - wait: 0.1 sec
 - pose: charsetcrown1.png
 - start animating
 - look: left
 - lookAt: player
 - wait for music fade

Look at sprite: player

Sort Apply Display Flags

Sort Apply new flags

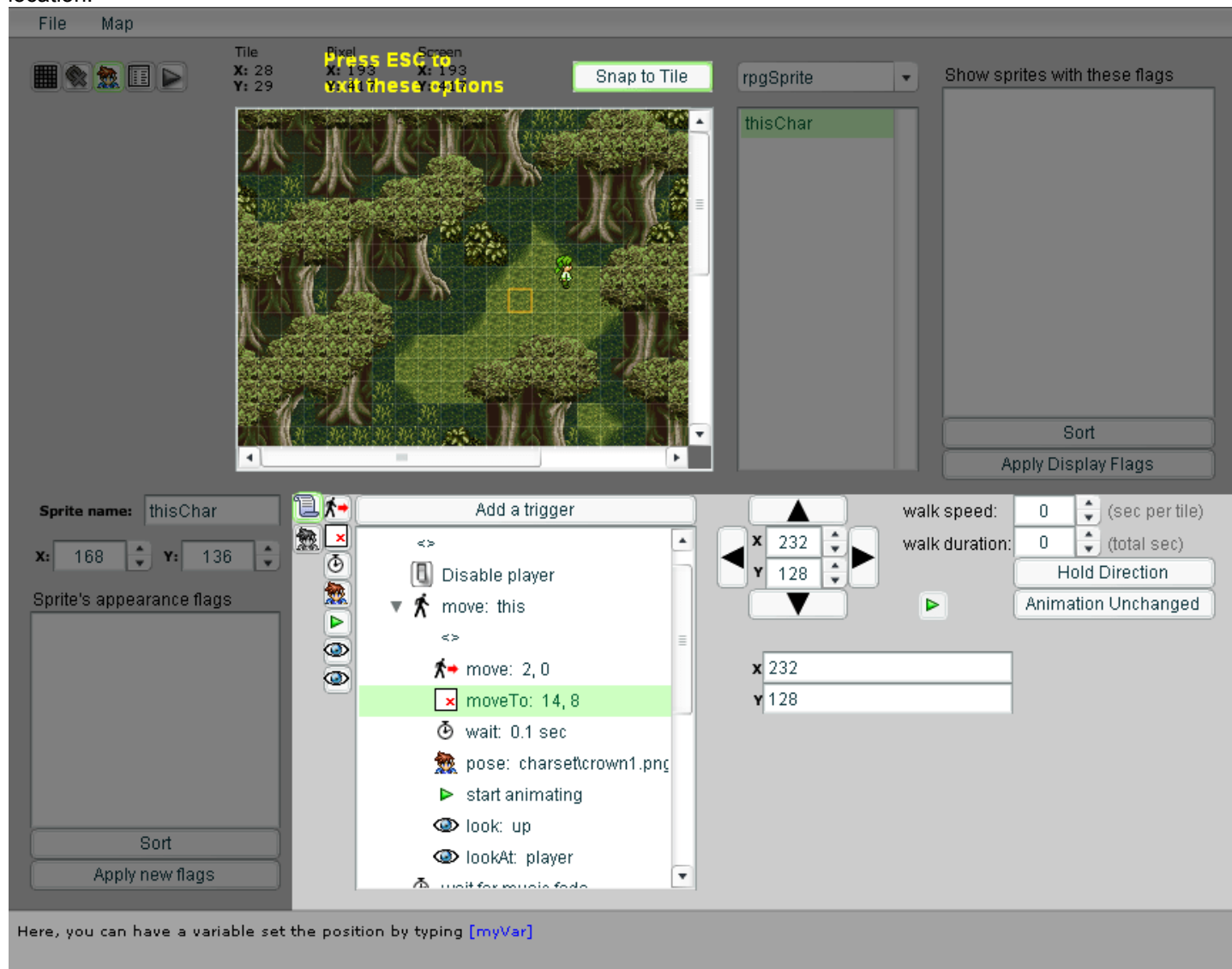
Double-click to create sprites. Click to select one. Drag to move it. Backspace to delete it. Zoom by rolling the mouse-wheel. Pan by pressing the mouse-wheel and dragging.

Absolute Movement

What this does

This command makes the character walk to an exact spot on the map.

Normally, the movement is set to occur instantaneously. But you can set a walk speed to make the character walk to the location.

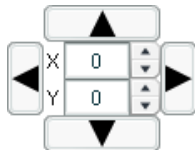


The screenshot displays the RPG Maker interface for configuring the Absolute Movement command. The top panel shows the map view with a character and a yellow square indicating the target location. The command list includes 'moveTo: 14, 8'. The settings panel shows walk speed and duration set to 0, and the character's position set to X: 232, Y: 128. The bottom panel shows the character's appearance flags and the 'Add a trigger' list.

Moving the character

When you create or click on this command, the rest of the screen will go dark to highlight the relevant parts of the interface. When you're done, press ESC.

To move the character, just click on the spot on the map you want them to move to. If you turn off snapping, you can click on exact pixel locations.



You can also click on these arrow buttons to move the character. Or you can type numbers in the X and Y boxes to specify the exact pixel coordinates to move to.

You also see these numbers reflected in a pair of boxes at the bottom. You can type in variable names in these boxes if you want variables to control the character's movement instead of specifying exact numbers.

Settings

Walk speed

This controls how many seconds it takes for the character to walk 1 tile. Usually, it doesn't take a whole second. Changing this will automatically adjust the walk duration.

Walk duration

This shows how long the entire distance will take. As you move the character, this value will be continually updated.

However, you can adjust it afterwards to force the overall movement to be faster or slower. Or set it to 0 to make the movement instantaneous.

Auto direction

This button controls whether or not the character automatically faces the direction they're moving in.

Auto animate

This button controls whether or not the character automatically animates while moving and stops afterwards.

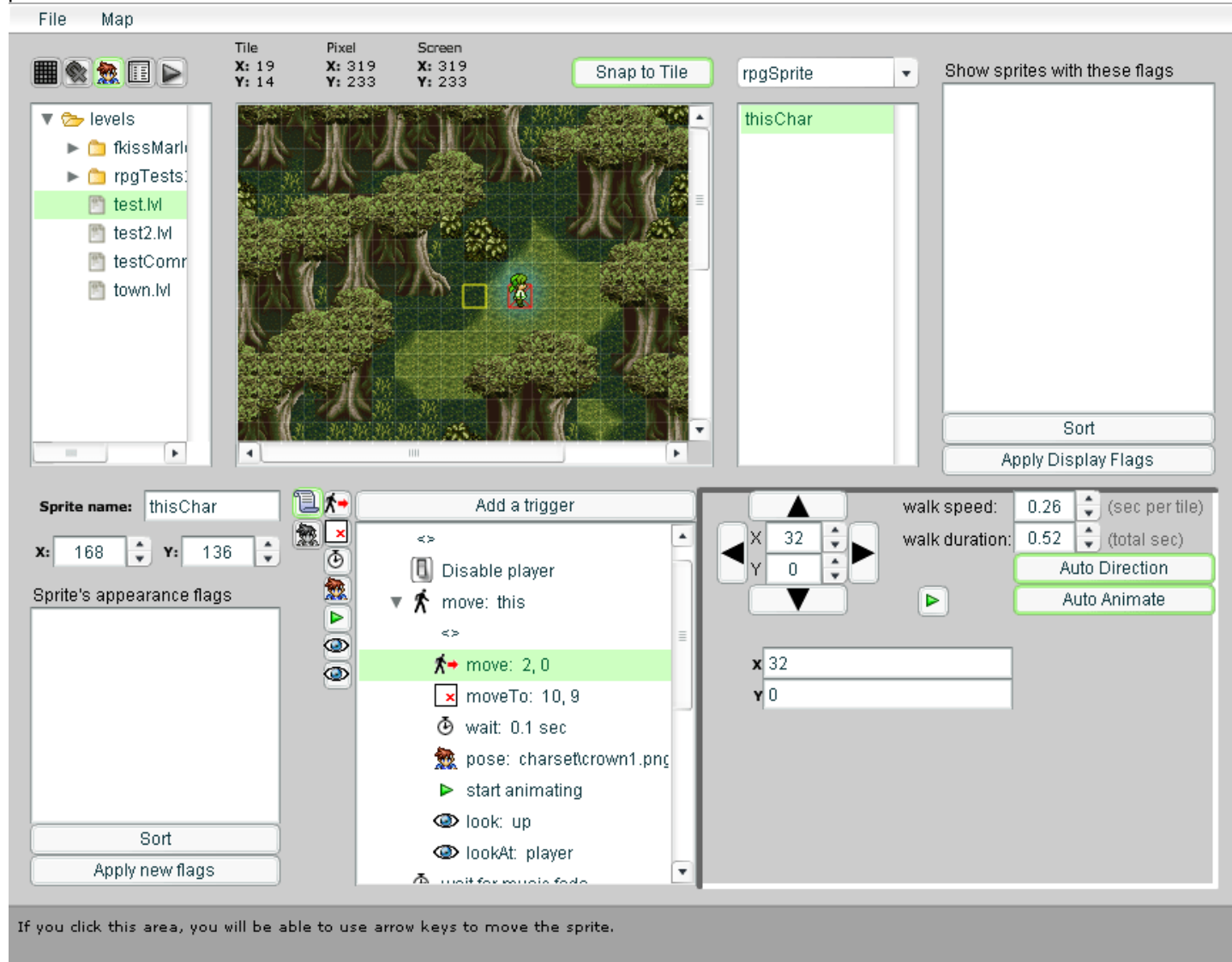
Play button

The play button can preview the character's movement. This is helpful to see how fast they'll move.

Relative Movement

What this does

This command makes the character walk a certain number of steps in any direction. Diagonal movement is also possible.



The screenshot shows the RPG Maker engine interface with the Relative Movement command selected. The command list includes: Disable player, move: this, **move: 2, 0**, moveTo: 10, 9, wait: 0.1 sec, pose: charsetcrown1.png, start animating, look: up, lookAt: player, and wait for music fade.

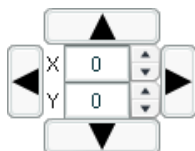
The configuration panel on the right shows:

- walk speed: 0.26 (sec per tile)
- walk duration: 0.52 (total sec)
- Auto Direction (checked)
- Auto Animate (checked)
- X: 32
- Y: 0

At the bottom of the interface, a note reads: "If you click this area, you will be able to use arrow keys to move the sprite."

Moving the character

When you first create this command, it'll respond to the arrow keys on the keyboard and move the character accordingly. If you click on any of the buttons in the editor, it'll stop responding to arrow keys. To make it respond again, click on the background. (you'll notice that its border changes as you mouse over it) The border will turn yellow and you can use arrow keys to move the character again.



You can also click on these arrow buttons to move the character.

Or you can type numbers in the X and Y boxes to manually set exactly how many pixels the character moves horizontally and vertically.

You also see these numbers reflected in a pair of boxes at the bottom. You can type in variable names in these boxes if you want variables to control the character's movement instead of specifying exact numbers.

Settings

Walk speed

This controls how many seconds it takes for the character to walk 1 tile. Usually, it doesn't take a whole second. Changing this will automatically adjust the walk duration.

Walk duration

This shows how long the entire distance will take. As you move the character, this value will be continually updated.

However, you can adjust it afterwards to force the overall movement to be faster or slower. Or set it to 0 to make the movement instantaneous.

Auto direction

This button controls whether or not the character automatically faces the direction they're moving in.

Auto animate

This button controls whether or not the character automatically animates while moving and stops afterwards.

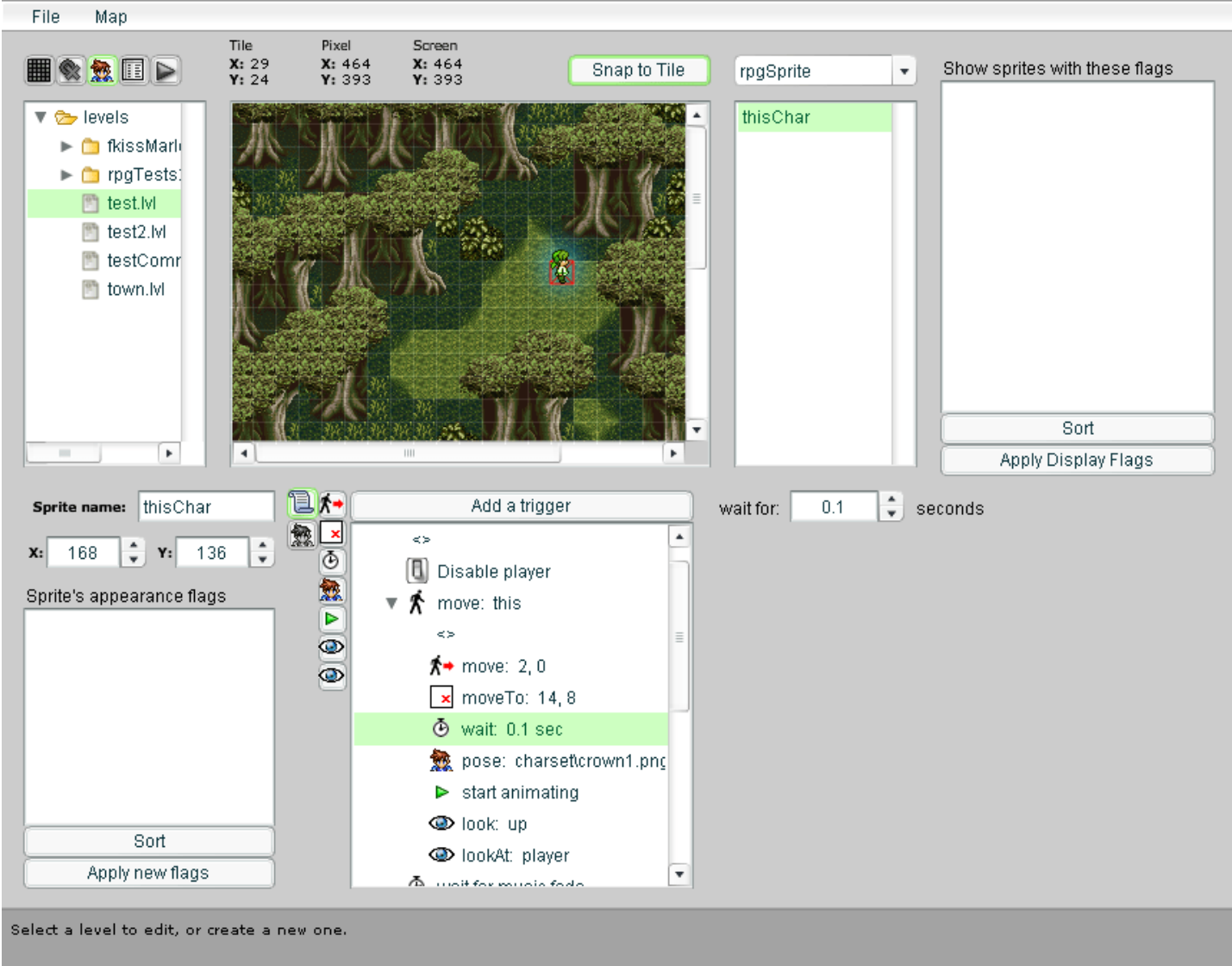
Play button

The play button can preview the character's movement. This is helpful to see how fast they'll move.

Wait

What this does

This command will pause the script for a certain amount of time.



The screenshot displays the RPG Maker interface with the following elements:

- Top Panel:** Shows 'File' and 'Map' menus. A status bar indicates 'Tile X: 29 Y: 24', 'Pixel X: 464 Y: 393', and 'Screen X: 464 Y: 393'. A 'Snap to Tile' button is active. A dropdown menu shows 'rpgSprite' and a list containing 'thisChar'. A 'Show sprites with these flags' section is empty, with 'Sort' and 'Apply Display Flags' buttons below it.
- Map View:** A central window showing a forest map with a character sprite on a path.
- Sprite Properties:** 'Sprite name: thisChar', 'X: 168', 'Y: 136'. A 'Sprite's appearance flags' section is empty, with 'Sort' and 'Apply new flags' buttons below it.
- Script Editor:** An 'Add a trigger' window is open, showing a list of commands: '<>', 'Disable player', 'move: this', '<>', 'move: 2, 0', 'moveTo: 14, 8', 'wait: 0.1 sec' (highlighted), 'pose: charsetcrown1.png', 'start animating', 'look: up', 'lookAt: player', and 'wait for music fade'. To the right of the script editor, a 'wait for: 0.1 seconds' control is visible.
- Footer:** A message reads 'Select a level to edit, or create a new one.'

loadSwf

What is a loadSwf?

This sprite displays a picture or an animation.
You can tell it to load a JPG, a PNG, or a SWF file.

That's it??

That's it.

Visibility Flags

The flag menus are probably the most confusing part of the editor. And I'll probably be answering questions about them until the day I die. But I'll try to explain it here.

What are these menus for?



These menus allow you to control when things appear in the game.

What are "flags"?



Flags are conditions. They typically control whether or not things are present during a given point in a game's story. They can also control whether or not cutscenes occur.

This is not the only way to make sprites disappear however. Sprites can be removed directly during any script using the SWF command, or made invisible using a move command. Flags are mainly used for keeping track of story events and deciding whether or not a sprite should exist during certain moments in a story.

When a variable's value changes in the game, every **SPRITES** "flag" condition is checked to figure out whether or not that sprite should exist. Some sprites will appear, others will disappear. You can also make cutscenes occur by making an invisible sprite appear, containing an automatic script.

Actually, each sprite can store a set of conditions, all of which need to be true for it to appear, otherwise it disappears.

NOTE: Not all games use flags.

Platformer games usually don't need them.

With a game like that, the editor will still be able to create and display flag conditions, but the game will ignore them and simply display all the sprites, all the time. With games like that, you don't even need to worry about these menus. You can ignore them completely.

"Preview" flags

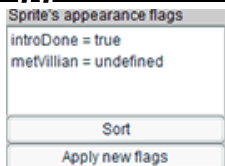


The list on the top-right controls which **SPRITES** you currently see in the editor. They're temporary and are not stored in the game.

This allows you to see what the player will see at a given moment in the game.

When the list is empty, you'll see EVERY sprite in this level, including the ones with flags.

"Appearance" flags



Each **SPRITE** has its own set of conditions that control when it appears in the game. A sprite won't appear unless ALL of its conditions are met.

(there can be more than one condition)

And it'll vanish the moment any one of the conditions is NOT met.

For example:

yVar = true

This sprite will only appear when the variable called myVar is equal to true. And it'll disappear if myVar has any other value.

If there are no conditions, (the list is empty) then this sprite will always appear.

How do I use flags?

When you click on a **SPRITE**, you'll see the flags that control it on the left side of its options. The sprite will only appear in the game when ALL of these conditions are met.

(there can be more than one condition)

If the list is empty, the sprite will always be visible.



NOTE: Clicking the “apply” button updates the editor’s display. If the sprite’s flags are excluded by the current preview flags, it will be hidden.

What are “conditions”?



These are examples of things you can type into a flag list:

```
myVar = tru
myObject.myVar <
myStory = "intro&#8221
myObject = undefine
fooBar != undefine
fooBar
```

Think of it as a bunch of “if” statements.

As you can see, you can also read variables that are inside of objects, or detect whether or not an object or variable even exists.

Also...

myVar

Is the same as writing:

```
myVar != undefined
```

It’s just an easy way of detecting whether it exists.

My sprite disappeared!



If you type something in the display flag list,

(on the upper-right)

then the editor will **only** display the sprites whose flags match that condition. To view all the sprites, including the ones with flags, simply empty the display flag list on the top-right.

To view only sprites that require a certain condition, type that condition into the top-right flag list.

How do I make a sprite vanish?

To deliberately make a sprite disappear under certain conditions, type something like this:

```
myVar != undefined
```

The sprite will be visible until myVar gets a value. In other words:

As long as myVar doesn’t exist, this sprite will be displayed.

NOTE: Some sprites are able to remove themselves. For instance, RPG sprites can remove themselves or turn invisible using their script commands.

Edit Game Settings

Copy Level Actionscript

If you want to make a game self-contained, it'll be necessary to store the level data in the game's SWF somehow. This command converts the level data into XML, and then sets it up to be stored as a really really long string! This command is helpful because Flash has a quirk where it can't define super-long strings all at once. This command automatically splits up the string when necessary to work around that.

It stores something like this in the clipboard:

```
data_txt = '<data>~~really long xml~~';  
data_txt += '~~~ </data>';  
data = new XML( data_txt );  
onLoad( data );
```

Testing the game

Testing the game

Click this button to play the current level in the game engine.

First, you'll need to set your starting place. Or click the "From Title" button.

Test conditions

+	-	RAM
variable		
introDone		

Here, you can set the starting value of flag variables in the game. This allows you to jump right into the middle of the game with certain conditions set.

File

This is mostly just a familiar way to create and save levels. You can also do all of these things by right clicking stuff in the file browser window. Oh, and you can also look up the keyboard shortcuts in here. But they're the same as literally every program out there.

Help

Help Manual

You're lookin' at it.

Reset Image Cache

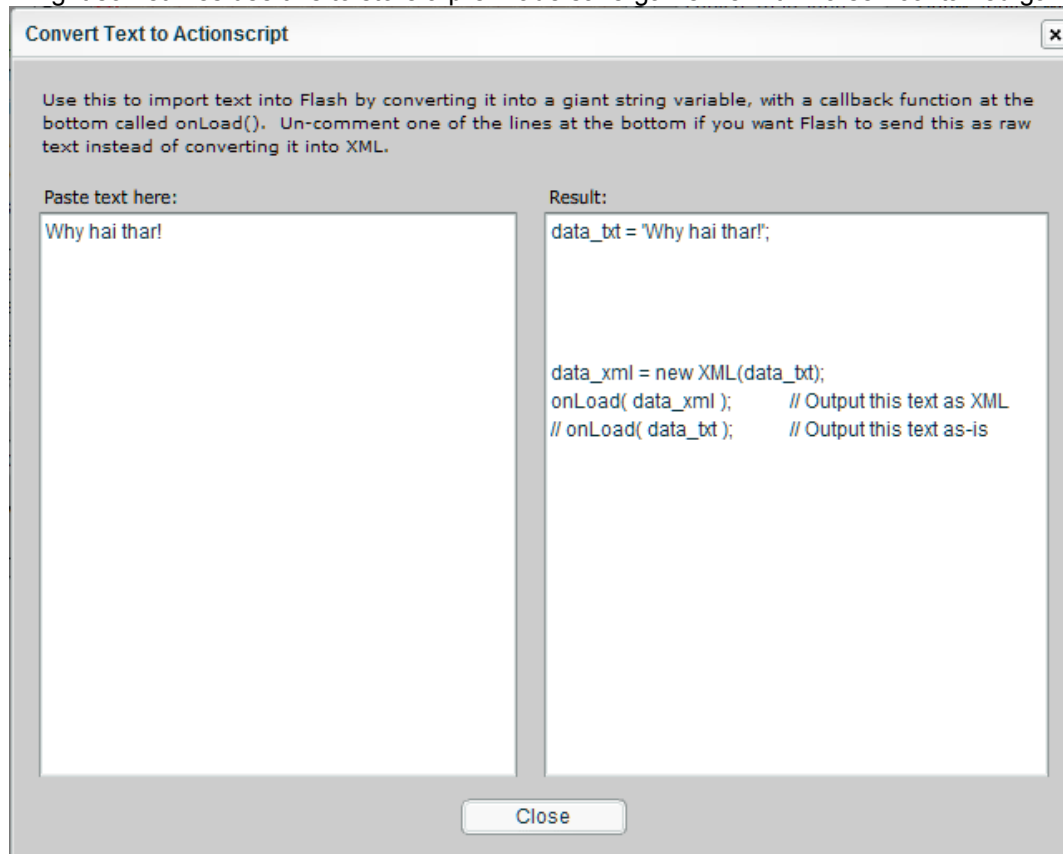
This is kind of a fail-safe in case something goes wrong. The level editor has a cache that it uses to speed up the performance of displaying image files, by remembering every image file of every sprite it loads. In fact, it preemptively starts reading the "charset" folder the moment you open the editor so that by the time you try to change a sprite's appearance, all of the files have already been loaded and (should) display instantly. Basically it loads every image only one time, and then just remembers the result instead of loading it again.

But... if it ever gets confused about which image goes with which file, the wrong image will be displayed when you try to access that file. You can reset the cache to fix this.

A cache reset is also needed if you're editing the image file in another program and need to tell the level editor to reload it.

Convert Text to ActionScript

This is similar to [List Folder Actionscript](#), except that it's used to "import" generic text files instead of levels. For example, I might sometimes use this to store a pre-made save-game file within a self-contained game.



Copy Level Actionsript

If you want to make a game self-contained, it'll be necessary to store the level data in the game's SWF somehow. This command converts the level data into XML, and then sets it up to be stored as a really **really** long string! This command is helpful because Flash has a quirk where it can't define super-long strings all at once. This command automatically splits up the string when necessary to work around that.

It stores something like this in the clipboard:

```
data_txt = '<data>~~really long xml~~';  
data_txt += '~~~ </data>';  
data = new XML( data_txt );  
onLoad( data );
```

Copy Level XML

If you go to Export → Copy Level XML

you can copy the level to the clipboard in XML form, using this command.

This is useful if you need to manually save the level to a text file for some reason. Of course, you can also open a normal level file in a text editor.

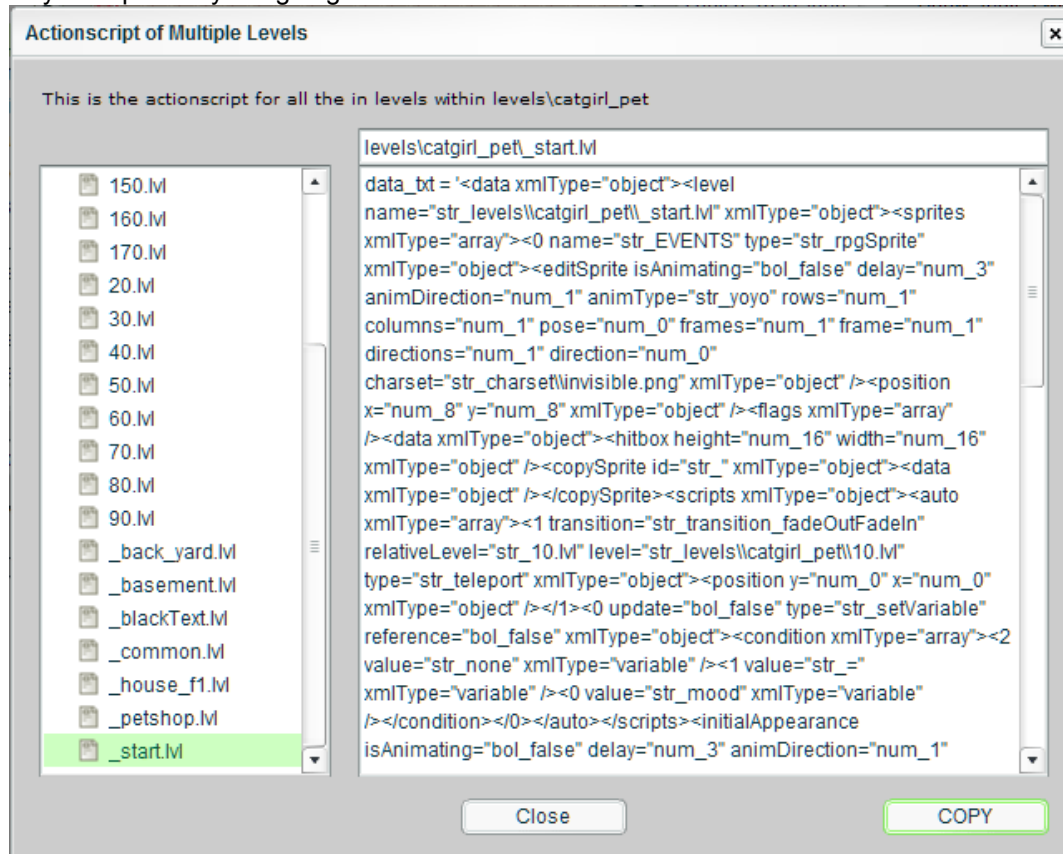
Copying the level might be useful when you're making a game self-contained, since the level's data would need to be stored in the SWF somehow.

However, it's better to use "Copy Level Actionscript" for that purpose.

List Folder Actionscript

I often use this when converting a finished game into a single self-contained file. This displays the XML of all level files within the selected folder and wraps that inside of a **very** large string variable, which I can then paste directly into Flash. I store levels within a self-contained game by creating an internal movieClip for each level and then just pasting this text inside each one. When the game "loads" that level, it places the movieClip, reads this text, and then just treats it the text itself the same as the result of loading an external file.

... you're probably not going to use this.



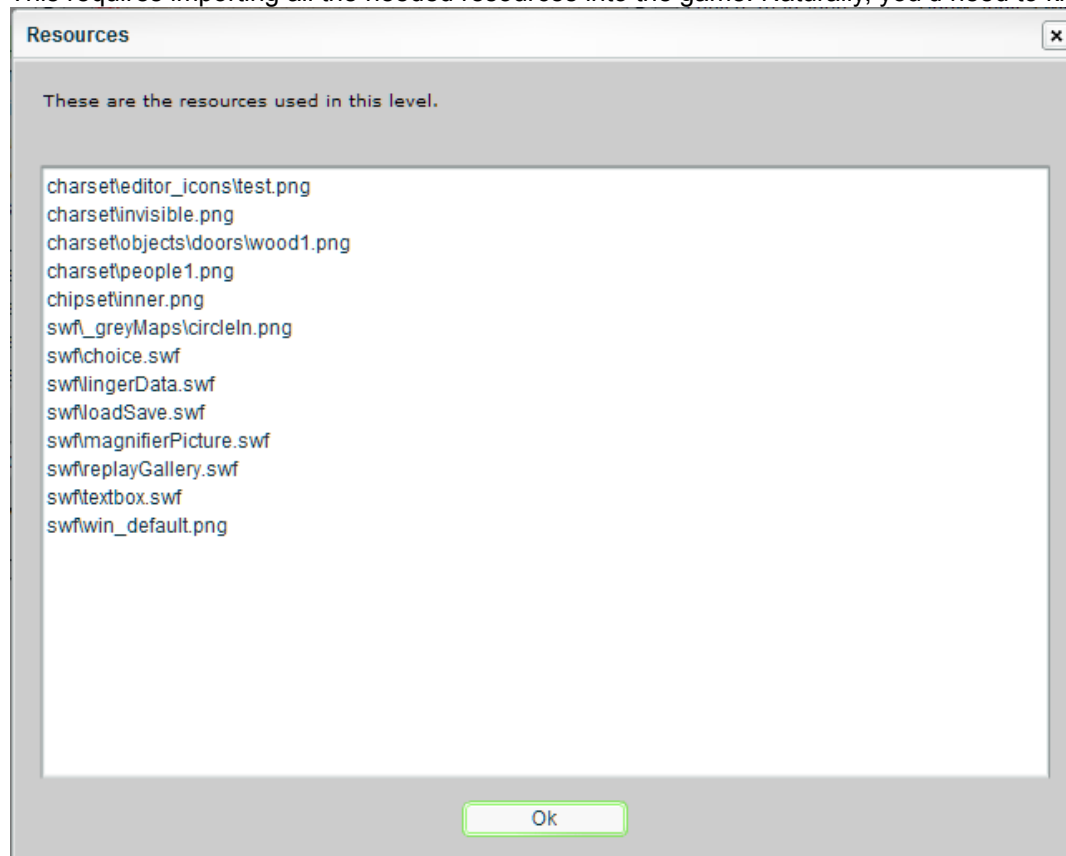
List Resources

List Resources

If you go to “Export” and select “List Resources”, you’ll see a list of every file the current level needs. This is useful for a couple things:

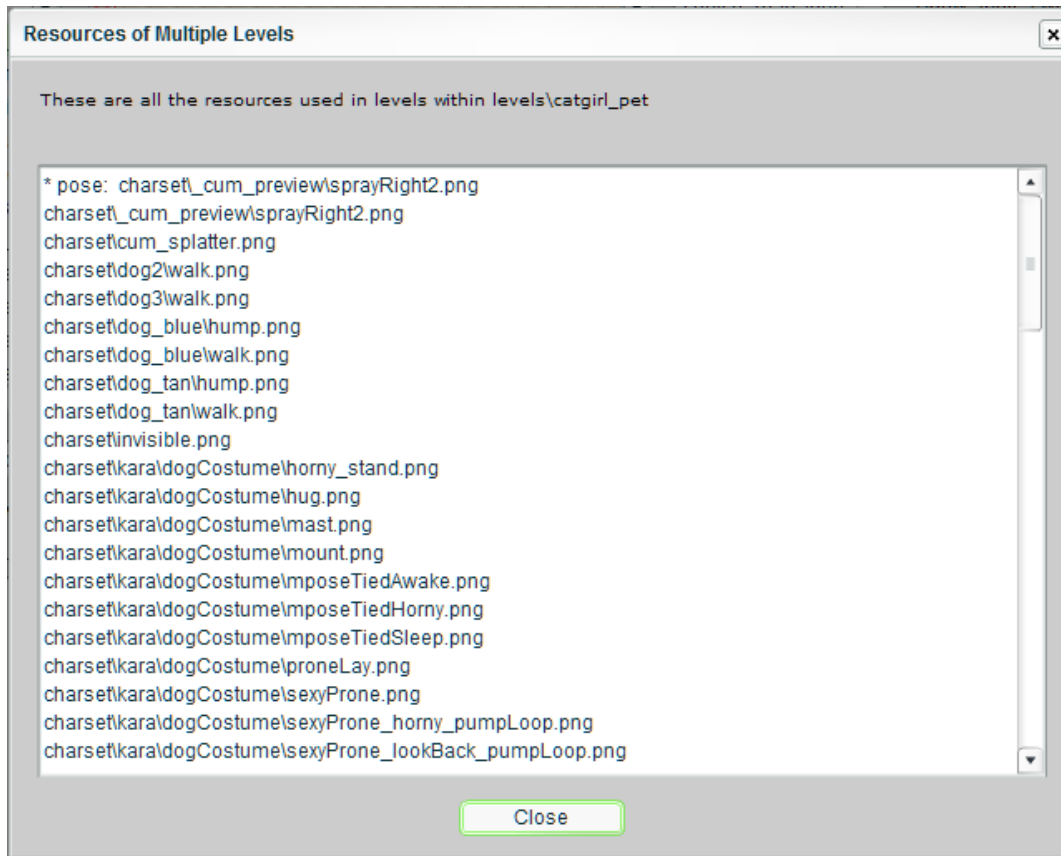
- You may have a completed game and want to reduce its size by removing all unused files.
- You may want to make the game into a self-contained SWF file.

This requires importing all the needed resources into the game. Naturally, you’d need to know which files to import.



List Folder Resources

This is similar to the other option except that it’ll scan every level inside of the selected folder and list the resources used by all of them.



Convert Map Format

At one point, I came up with a new way to store maps that allowed them to use multiple charset files at the same time (among other things) These options are used to convert between the old and new map formats. The game and the editor can use either format.

Copy Map Actionscript

This options converts the current map into XML and then writes actionscript code that will store it as a super-long string. This is useful for making certain games self-contained. My earlier games did not use level data, so each map was stored within them in this way.

It stores something like this in the clipboard:

```
data_txt = '<data>~~really long xml~~';  
data_txt += '~~~ </data>';  
data = new XML( data_txt );  
onLoad( data );
```

Flash has a quirk where it can't define super-long strings all at once. This command automatically splits up the string when necessary to work around that issue.

Copy Map XML

Most people won't need this feature.

This options converts the current map into XML and stores it in the clipboard.

With this, it's possible to manually save a map file via notepad.

It may also be helpful for making certain games self-contained. My earlier games did not use level data, so each map was stored within them as a long string containing the map in XML format.

Importing Maps

Importing & Exporting Maps

It is possible to save a map file without any sprites. This is useful if you want to re-use the map in other games.

To do this, go to the “Map” menu and select “Export Map...”

Map → Export Map...

This will save the map as an XML file with a .map extension.

The tiles and collision will be saved.

If you want to use a saved map in another level, open that level,

then go to the “Map” menu and select “Import Map...”

Map → Import Map...

This will REPLACE the current level’s map. However the level’s sprites will not be touched at all.

Resizing Maps

Resizing

You can change the width and height of the map by going to:

Map → Resize

This will allow you to resize the map and change the position of the existing map within the new area. Any tiles outside of the new map's area will be lost.

Repositioning can be useful when you want to center a room in a larger map, for instance.

Wrapping

You can also shift the map in a way that makes edge tiles wrap-around from one side to the other. To do this, go to:

Map → Reposition & Wrap

It's useful for centering rooms in RPG's.

How the editor was made

How the f*ck did you make this!?

In Flash... with Actionscript 2...
over the course of about 5 months...
in my spare time.

(uphill... both ways...)

I also used a program called MDM Zinc to add features that Flash normally wouldn't have.

I tried to focus on building everything out of very modular components and designing intuitive interfaces. I won't go into more detail since there's so much involved that it'd take a month just to explain it all... also, I completely reprogrammed large parts of it over the years.

Required Folders

The game and game editor load resources from external folders. And these folders need to be present for them to work. The only exception is when a game has been converted to be self-contained. In which case, it'll run as a single SWF file.

Folders needed to play a game

charset

This folder stores all the sprite sheets. These are PNG files.

chipset

This folder stores all the tiles used by maps. These are PNG files.

levels

This folder contains level files created and edited by the game editor. They use a .lvl extension, but are really XML files. Level data has [this structure](#) after being decoded from XML.

music

This folder contains looping MP3 files.

Some songs optionally have intros.

Song intros have _intro appended to their names, and are automatically detected and loaded by the music system.

For example:

mySong.mp3

mySong_intro.mp3

If the game tries to play "mySong.mp3", the music system automatically plays "mySong_intro.mp3" first.

sound

This folder contains short MP3 files, which are used as sound effects.

swf

This is a miscellaneous folder for any Pictures and SWF files you want in the game. These can be used for anything.

Such as:

Title Screens, Animations, and even whole Programs coded as SWF files.

You could even program a battle system as a SWF file and simply place it in any game as needed.

Files needed to play a game

game.swf

This file is the game itself.

database.xml

This optional file stores all non-level data such as the starting place and title screen. Each game has its own unique database.

(Some don't have a database at all)

Folders needed for the editor

In addition to all the files and folders above, the editor also needs these:

sprites

This folder contains SWF files that tell the editor how to edit this game's [SPRITES](#). Each game has its own sprites folder.

[This is how they're setup.](#)

database editors

This folder contains SWF files used for editing database information. Each SWF file corresponds to a specific type of information. For example, the titleScreen.swf file contains the editor that is used to select the title screen of your game.

Files needed for the editor

basis.png

This is needed because it's the default chipset used by new maps in the editor. However, you don't have to use it in any games.

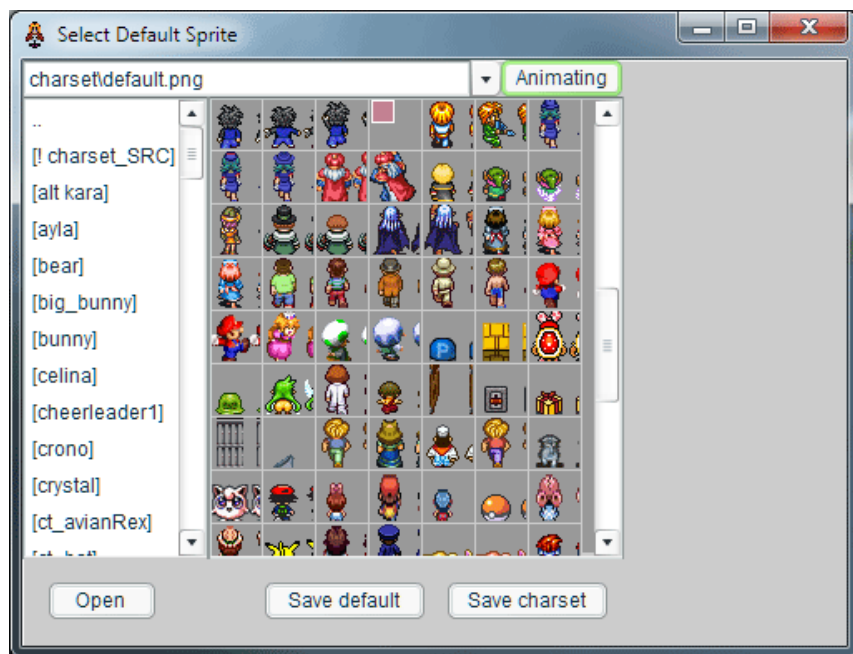
editDatabase.swf

This optional file is loaded by the editor and tells it how to create and edit this game's database.xml file. (Not all games have a database.xml, but most do)

defaultSprite.exe

This tool is used to define how each sprite looks within the editor. [Using DefaultSprite.exe](#)

About defaultSprite.exe



What is it?

DefaultSprite.exe is used to define how sprites look within the game editor.

It's used to create a file named:

defaultSprite.xml

... which is stored in each sprite's folder. For example, the one for rpgSprite is located here:

sprites\rpgSprite\defaultSprite.xml

The Game Editor uses these files to figure out which graphics to use when displaying sprites.

[Here's how they're set up.](#)

How do I use it?

First, you run the program.

double-click defaultSprite.exe

Then select an image file from the list at the top. (It doesn't support sub-folders)

Now, adjust the settings on the right:

()

Finally, click the "save" button and save the file to the sprite's folder.

Make sure it's named: defaultSprite.xml

Sprite Image Settings

The level editor uses the sprite image system to display sprites. A compatible game will use this system as well. To use the sprite image system in your game, you would #include the sprite.as file. If you open this file in flash, the comments at the top will explain how to use this system and go over the settings. The settings will also be explained here for convenient reference. The sprite image system uses these settings to control what a sprite looks like.

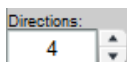


Sprite image settings

charset

This string is the filename of the image that contains the graphics for this sprite.

directions



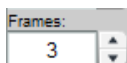
How many directions is this sprite capable of looking? This number is how many directions this sprite can face. Each direction is stacked vertically, one on top of the other, in clock-wise order.

direction



This is stored as a number, representing the direction this sprite is currently facing. The directions are arranged in clockwise order, with Zero at the top, 1 below it, 2 below that, etc. You can also use the strings "up" "right" "down" and "left" When you pass these strings to the sprite system, it'll convert them into the numbers that correspond with those directions

frames



How many frames of animation does this sprite use? Animation frames are arranged next to each other, and usually animate from left to right. What happens after the last frame is up to the [anim type](#). Each direction and pose in an image will have the same number of animation frames.



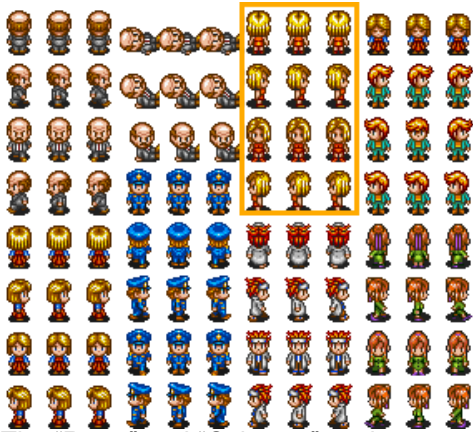
frame

This number is the frame that the animation starts on. The sprite will also jump to this frame when the animation is stopped.

Columns & Rows

Columns: 4
 Rows: 2

Multiple sprites are sometimes stored in a single image file, each sprite has its own directions and animation frames. Within a file, these sets are tiled. Each one has the same amount of animation frames and directions.



The "Rows" and "Columns" settings are used to tell the editor how many sets there are horizontally and vertically. **Columns** refers to how many sets are sitting next to each other, horizontally. **Rows** are how many sets are stacked on top of each other, vertically.

isAnimating

Animating

This boolean controls whether or not this sprite is animating.

animType

Anim type: yoyo

This string controls how this sprite animates.

- LOOP
- YOYO
- ONCE

animDirection

This number controls whether the sprite animates forwards or backwards through its frames.

delay

Anim delay: 3

This number controls how fast the sprite animates. It is the delay between animation frames. Higher numbers make it slower.



When the settings are correct, the selection box should only be surrounding a single frame of the character's animation.

After adjusting the settings, click on the animation frame you want it to start on.

Level Data Structure

It's helpful to know what the level editor is creating. Below, you'll see all the data that it creates and how it's organized. This is basically what's stored in each level file.

Key

{object}
[array]
variable

Level structure

```
{data}
  {level}
    name = levels\test.lvl"
    {map}
      map data
      [sprites]
      {0}
        type = "rpgSprite"
        name = "sprite_1634"
        {data}
          arbitrary settings
        {position}
          x = 160
          y = 120
        [flags]
          0 = myFlag=true
          1 = myNum>4
          2 = myObj.myFlag2="text"
        {editSprite}
          sprite image settings
```

Additional Info

map data

The "map" object stores the map. It contains information such as the map's width, height, and collision_array.

arbitrary settings

Each sprite can have its own unique settings. All of those settings are stored inside of this "data" object.

sprite image settings

Map Data Structure

It's helpful to know how maps are stored in a level. And it's also helpful to know the contents of the map movieClip. Below, I'll show the structure of both.

Key

```
{object}
[array]
  [x]          (Horizontal position)
  y =         (Vertical position)
variable
variable_mc   (movieClip)
variable_pic  (bitmap data)
```

Map data structure

It's helpful to know how maps are stored in a level. This object is what you would send to a map to make it draw something.

Since the map can store multiple layers of tiles, the "layer" array contains multiple 2-dimensional arrays, one for each layer. Each 2-dimensional array stores the tiles for that layer. Each tile is an object that contains a "chipset" "x" and "y" variable. These variables are used to look-up the tile image located at the coordinate specified by "x" and "y". And the "chipset" variable is an index number that refers to one of the files in the [chipsets] array. Therefore each tile can use a completely different chipset.

The zeros and ones in the collision array refer to collision ID's. Zero usually represents open space, and One usually represents walls. Other values are treated differently by different games.

When map and level files are saved, some of the map data is stored in a different way to reduce filesize. The maps in these files will have their "isCompressed" variable set to "true". The level editor de-compresses the data into the structure you see here and sets the "isCompressed" variable to "false". When it saves a file, it stores the map data in compressed format. Games typically leave the data in compressed format and simply send the compressed data to the map system, which can display both formats.

The [old map format](#) handles this differently.

```
{map}
  format = 3          (This map format is always labeled as 3)
  isCompressed = false (This data is un-compressed. Files are compressed)
  width = 20         (the map width, in tiles)
  height = 15        (the map height, in tiles)
  tileSize = 16      (the width & height of each tile, in pixels)
  [chipsets]         (there can be up to 36 chipsets)
    0 = "chipset/basis.png"
    1 = "chipset/pokemon.png"
    2 = "chipset/zelda.png"
  [layers]           (there can be up to 9 layers)
    [0]
      [x]
        [y]
          chipset = 0 (chipset index. This one refers to basis.png)
          x = 4       (this tile's horizontal coordinate in the chipset, in tiles)
          y = 12      (this tile's vertical coordinate in the chipset, in tiles)
    [1]
      [x]
        [y]
          chipset = 2 (chipset index. This one refers to zelda.png)
          x = 35      (coordinates can be between 0 and 35)
          y = 0
  [collision]
    [x]
      y = 0
      y = 0
      y = 1
      y = 0
```

Map movieClip structure

When the map system loads the above data, it'll create a movieClip with the following structure. It's useful to understand the contents in order to access things like collision data. This movieClip also contains some useful functions, which are explained in the comments at the top of map3.as.

```
{map_mc}
  {layer0_mc}
    pic = BitmapData           (image data being displayed by layer0_mc)
  {layer1_mc}
    pic = BitmapData           (image data being displayed by layer1_mc)
  [layers_mc_array]
    [0]                         (reference to layer0_mc movieClip)
    [1]                         (reference to layer1_mc movieClip)
    [2]                         (reference to layer2_mc movieClip)
  [chipset_mc_array]
    [0]
      file = "chipset/basis.png" (path to an image file)
      pic = BitmapData           (image data of basis.png)
    [1]
      file = "chipset/pokemon.png" (path to an image file)
      pic = BitmapData           (image data of pokemon.png)
  [collision_array]
    [x]
      y = 0
      y = 0
      y = 1
      y = 0
```

Old Map-Data Structure

It's helpful to know how maps are stored in a level. And it's also helpful to know the contents of the map movieClip. Below, I'll show the structure of both.

Key

```
{object}
[array]
  [x]          (horizontal position)
  y =         (vertical position)
variable
variable_mc   (movieClip)
variable_pic  (bitmap data)
```

Map data structure

It's helpful to know how maps are stored in a level. This object is what you would send to a map to make it draw something.

Since the map can store multiple layers of tiles, the "layer" array contains multiple 2-dimensional arrays, one for each layer. Each 2-dimensional array stores the tiles for that layer. The numbers assigned to the various "y" positions refer to tile image ID's in the chipset. So the map system looks at this number and knows which tile in the chipset to display. The zeros and ones in the collision array refer to collision ID's. Zero usually represents open space, and One usually represents walls. Other values are treated differently by different games.

When map and level files are saved, some of the map data is stored in a different way to reduce filesize. The maps in these files will have their "format" variable set to 2. The level editor de-compresses the data into the structure you see here and sets the "format" variable to 1. When it saves a file, it stores the map data in compressed format. Games typically leave the data in compressed format and simply send the compressed data to the map system, which can display both formats.

The newer [Map3 format](#) handles this differently.

```
{map}
  format = 1    (1 = un-compressed data. 2 = compressed data)
  width = 20    (the map width, in tiles)
  height = 15   (the map height, in tiles)
  chipset = "chipset/basis.png"
  [layers]      (there can be up to 9 layers)
    [0]
      [x]
        y = 214
        y = 16
        y = 0
    [1]
      [x]
        y = 214
        y = 16
        y = 0
  [collision]
    [x]
      y = 0
      y = 0
      y = 1
      y = 0
```

Map movieClip structure

When the map system loads the above data, it'll create a movieClip with the following structure.

It's useful to understand the contents in order to access things like collision data. This movieClip also contains some useful functions, which are explained in the comments at the top of map.as.

```
{map_mc}
  {layer0_mc}
    layer0_pic
  {layer1_mc}
    layer1_pic
  width = 20    (the map width, in tiles)
```

```
height = 15 (the map height, in tiles)
chipset_pic
[collision_array]
[x]
  y = 0
  y = 0
  y = 1
  y = 0
```

Making a Game That Uses This Editor

Making a compatible game

To make a game that works with the game editor, it needs to be able to display levels and sprites.

These are the things it needs to do:

[systems](#)

[load a level](#)

[load a map](#)

[map collision](#)

[place sprites](#)

And some things you'll probably want:

[editor hook](#)

[read ROM](#)

[read RAM](#)

I also prepared a full [online video tutorial](#), that covers most of these.

Systems

The level editor uses a general-purpose map system to display maps, and a general-purpose sprite system to display sprites.

These "systems" are reusable code stored in .AS files. To use one, create a folder in the same location as your game. (I usually name it "functions") Copy the .AS file into that folder. Use #include to insert the code into your game, and then call the system's setup function. The setup function is usually named after the AS file, but open the AS file and check to make sure.

[Video](#) (at 0:27)

These are the systems you'll need:

- readXml.as
- nextDepth.as
- map.as
- addCollisionMethods.as
- sprite.as
- makeStreamingMusic.as
- makeStreamingSoundSystem.as

(USAGE EXAMPLE)

(REQUIRED SYSTEMS)

Loading a level

Each level file is actually an XML file, so you use flash's XML class to load it. After loading it, you use the readXml.as system to convert the XML into regular flash variables and objects. Finally, you send this data to a function or movieClip that reads it and creates the map and sprites.

[Video](#) (at 2:30)

(CODE GOES HERE)

Loading a map

Each [level](#) stores map information in this location:

data.level.map

To display it, simply tell a map system to draw the contents of this object, like so:

```
MAP.drawObj( data.level.map );
```

[Video](#) (at 2:45)

(USAGE EXAMPLE GOES HERE)

Map collision

Maps store their collision using a 2-dimensional array of numbers.

The array is called collision_array.

Therefore:

```
pre.
```

```
myMap.collision_array[4][3]
```

Would look up the collision value of a tile located 4 blocks over, and 3 down.

I usually use 0 to represent open space, and 1 to represent walls. You can use other numbers to represent other things, such as water.

[Video](#) (at 6:45)

(EXAMPLE GOES HERE)

Place sprites

Each [level](#) stores sprite information in this location:

`data.level.sprites`

“sprites” is an array, where each element is a sprite in the level. To place all the sprites, you simply loop through each one and use the `attachMovie()` function to place the sprite specified by the “type” variable.

[Video](#) (at 7:30)

(SPRITE VARIABLES)

(CODE)

If you want the appearance of sprites to be controlled by flags, use the system: `updateFlaggedSprites.as`

Also, if you open this file, the comments at the top will have example code you can use. (warning, it's a bit long)

Editor hook

This is a function on the first frame that runs before anything else. It allows the game to be play-tested from within the editor. The only thing it does is receive level data and store it in a variable.

[Video](#) (at 0:27)

(CODE)

Read ROM

A game can store game information outside of levels in a file called `database.xml`. This is useful for defining things such as the starting place at the beginning of the game.

To use this file, you load the XML, convert it into flash data, and store it in a global object.

I usually name the object `ROM`, since it won't change during gameplay.

(CODE)

Read RAM

I usually store gameplay variables and flags in a global object named `RAM`. Gameplay variables are things that keep track of your progress through the game, and would be saved if you saved your game. When you test your game in the editor, you can define some `RAM` values to use while testing.

To allow the editor to define the `RAM` variables, you need to detect whether or not `RAM` already exists at the beginning of the game, where you'd normally create it.

(CODE)

Video Tutorial

Still confused? Click the link below to see how most of these are done!

[Online Video tutorial](#)

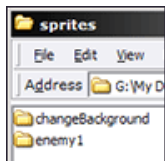
http://www.humbird0.com/#/tutorials/making_a_shooter

Making Sprite Editors

Before the level editor can place a sprite, you need to give it a little information about the sprite.

- Place a folder in the editor's "sprites" folder.
- Use [DefaultSprite.exe](#) to tell the editor what the sprite looks like.
- Create editor.swf, which the editor will use to edit the sprite's settings.

Sprite folder



In the editor's "sprites" folder, create another folder and name it after the sprite's linkage name. This will allow the editor to place the sprite.

editor.swf

The level editor will send a bunch of information to this editor, which will all be available when you get to frame 2. You'll mostly be interested in this object:

editor.sprite.data

This object stores all the settings for this sprite. They can be whatever you want. At first, it'll be empty. So you'll want to define your settings when this sprite is first created and this editor is loaded.

(CODE GOES HERE)

By defining the settings in this way, they'll be created when the editor is opened for the first time. But they won't get overwritten with default values afterwards.

Now all you have to do is use some flash components to allow the user to change these variables, and your editor will be complete.

Available data

The level editor makes lots of information available to sprite editors.

PASSED DATA:

(editor)

(cursor) Cursor component

(menuBar_mc) MenuBar component

(status) Help text object

(clipboard) Clipboard object

(levelChanged) Tracks level changes

(data) current level's data

(sprite) selected sprite's data

(editSprite) sprite image settings

(position) location on the map

 x Horizontal pixel location

 y Vertical pixel location

[flags] array of flag strings

(data) this sprite's settings (These are edited by editor.swf)

type sprite's linkage name

name sprite's instance name

snapping Map snapping: true/false

(map_mc) Map component

(sprites_mc) Sprites component

(overlay_mc) empty movieClip

(fileBrowser_mc) file browser

[sprite image settings](#)

[flag strings](#)

I won't go over all these right now since it would take a very long time to explain them all.

Video Tutorial

Still confused? Click the link below to see a video demonstrating how it's all done!

[Online Video Tutorial](#)

Making Your Game Self-Contained

When you want to upload your game to the internet, you'll usually need it to be a single file. To do this, you need to import all the external files your game uses and store them within game.flc.

It's generally best to only do this after all of your game's levels have been completed.

Making the game self-contained involves these steps:

- Add a preloader.
- Import all the external resources.
- Give them all linkage names.
- Drag them on-stage in the importer.
- Change the level loader.

External resources needed

charset
chipset
levels
music
sound
swf

Preloader

There are many tutorials online about how to make preloaders, but the basic point is to make flash wait until everything has finished loading before playing. This is the basic code you'd put on frame 1:

(CODE GOES HERE)

It's also a good idea to display the progress. To do that, you'll usually create a fill bar, which is basically an empty box and a rectangular movieClip that scales until it fills the box. You'd calculate the scale using a proportion formula like this:
fill._xscale = loaded * 100 / total;

Of course, there are much more creative ways to represent load progress. Just make sure the preloader doesn't take much memory, otherwise, the user would have to wait for the preloader to load, and that would defeat the point.

Importing resources

Charsets and chipsets are imported like regular pictures.

To import a level, open the level file in the game editor and select the menu option: File → Copy Level Actionscript
Then create a new movieClip in your game, and paste the code in the clipboard into frame 1.

Music and sounds are imported normally as well, except you generally want to import WAV versions of your music to make them loop smoothly. (MP3's will pause slightly at the beginning)

SWF files need to be copied as movieClips. You do this by opening the FLA files for each SWF file your game uses, and copying their contents over. (Keep your libraries organized)

Linkage names

The linkage names of all your imported resources must exactly match their relative paths & filenames. For example, if you had a song named titleScreen.mp3, the linkage for its imported version should be:

music/titleScreen.mp3

This applies even if your imported version is a WAV file, it still must end with.mp3.

The map, sprite, sound, and music systems automatically look for internal versions of files when external versions are not found. So you want the linkage names to exactly match the original file names being given to these systems.

Another thing you **must** do in all the linkage options for these resources is un-check:

Export in first frame

This is critical! If you forget to do this, the resources will try to load before the preloader instead of after it! Which would defeat the whole point of having a preloader.

The "importer" movieClip

This movieclip tells Flash to load everything after the pre-loader. It works like this because when you tell Flash to not load things on the first frame, their linkages won't work until flash encounters them on-stage first.

So here's what you do:

Create a new movieClip.

Place stop() on its first frame.

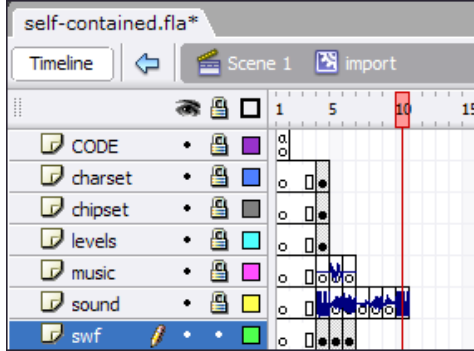
Then **after** the 1st frame, drag EVERY imported resource on-stage.

For sounds and music, create multiple frames and select a sound for each one. (Use the properties panel)

Finally, insert a frame in your main timeline between your preloader and your game's setup code, and drag this "importer" movieClip on-stage on that frame.

When Flash plays past this frame it'll encounter everything "on-stage" for a brief moment, which enables all their linkages to work when the game attempts to use them. The person playing the game won't actually see the importer

because the resources are not on the importer's 1st frame. Its 1st frame is blank.



Level loader

The last thing you need to do is alter your level-loading code so that it loads the level from the internal data if an external version of the level file cannot be found.

It would end up looking something like this:

(CODE GOES HERE)

In the code above, notice the “success” parameter in the onLoad() function. If an XML file fails to load, that variable will be false.

When you place an internal level's movieClip, you'll also want to define a function inside of that movieClip (also named onLoad()) This function will receive the level's XML data as a parameter.

The next thing to do is convert the XML into flash data, using the readXml() function. (imported at the top of the example)

And finally, you send the data (now in the form of regular objects and variables) to whatever function or movieClip you use to construct the level, placing its map and sprites.

Video Tutorial

Still confused? Click the link below to see how all of this is done!

The “self-contained” section demonstrates all of this.

[Online Video Tutorial](#)

http://www.humbird0.com/#tutorials/making_a_shooter