Documentation for TML’s Engine  
Tutorial

The tutorial will help you learn how to use this engine quickly.

* Read First
* GameMaker Studio 2 Basics
* Overworld
* Battle

Helpful links:

* [GameJolt](https://gamejolt.com/games/undertale_engine/378055)
* [Github Repository](https://github.com/TML233/UndertaleEngine/tree/modernize)
* [Discord server](https://discord.gg/njvUdff)

## NOTE: In 2022, Gamemaker has been making changes that will make older versions of TML’s UNDERTALE engine obsolete. For a quick fix, download the [**modernized version of the engine.**](https://github.com/TML233/UndertaleEngine/tree/modernize)

*Also please note when you see something in [brackets], this is intended to link you to the page the link is referencing. However, because this is a google doc, this feature is unavailable. Jevil is supposed to move this to Github once this documentation is completed and he has the time to do so.*

Read First

# What is GameMaker?

Gamemaker (Formerly known as GameMaker Studio 2) is a video game engine, run by YoYo games. As it is right now, [GameMaker has both a free and subscription option](https://gamemaker.io/en/get). (There is also a single time fee of 99$ option on steam.) The game goes on sale twice a year, at Christmas and then again at the program’s anniversary.

We have a strong stance against priracy/cracking GM. The program can’t be cracked, and you will likely get a virus. For your safety, **don't crack GM.** If you must use the program, use the free monthly trials.

Please note that GameMaker owner, YoYo Games was bought by Opera browser in 2021 and is undergoing major changes. Please stick to the [modern version of the engine](https://github.com/TML233/UndertaleEngine/tree/modernize). To learn GameMaker Studio 2, please watch videos from this:

[Coding Fundamentals in GML (English)](https://youtube.com/playlist?list=PLwgH1hDD0q1Eq2xXKhkiJmtt7ml599CSt) *(Still updates)*

[Let’s Learn GameMaker (English)](https://www.youtube.com/c/letslearngamemaker/playlists) *(up to 2017, but most of it is still applicable to 2022)*

Please note that GameMaker Studio 1 is a different program than Gamemaker, but most of the code logic is the same between programs.

*(GameMaker Studio 2 is what Gamemaker used to be called. Most online resources will refer to it by the GMS 2 title, as the change is recent. This documentation will us GM)*

# Alternatives to UT Engine

There are a few alternatives to this engine, all of which are free. [**Scratch**](https://scratch.mit.edu/projects/224896574/#player), [**Unitale/Create Your Frisk/Create Your Kris**](https://www.reddit.com/r/Unitale/comments/akhrcs/download_links_guides_and_help_for_unitale_and/), and [**Kristal**](https://nyako.gay/kristal/).

*(Clickteam to be properly described later as Sam, the one writing this, does not know much about this program)*

We understand the [[pricing]](https://gamemaker.io/en/get) of gamemaker is inconvenient towards fangame creations.Fangames are intended to be a way to learn game development while also showing your appreciation to the source material. It is not a way to make money, that’s illegal. Paying to make a free fan game feels unfair, especially when many fan game creators are young and don’t have spending money.

We recommend that if you must use the subscription model, wait for your game to finish developing and pay for it once. **Do not forget to unsubscribe,** or you’ll be charged twice.

**Please do *not* crack GM.** If you are unable to pay for the subscription fee to export your game, please use a different program. **There are free alternatives to making your fangame. Don’t risk viruses by pirating GM.**

### Please note the alternatives mentioned here are unaffiliated with us, and are just our own recommendations.

[**Scratch:**](https://scratch.mit.edu/projects/224896574/#player)

Extremely beginner friendly for all game developers. A great tool for someone wanting to learn to code. For Undertale fangames however, it is very limited.

**Price: Free**

**Undertale: Battle**

[**Unitale/Create Your Frisk/Create Your Kris:**](https://www.reddit.com/r/Unitale/comments/akhrcs/download_links_guides_and_help_for_unitale_and/)

This is both an Undertale Engine and Deltarune Engine. While it uses LUA, a beginner's programming language, learning how to use CYF/CYK is a learning curve and can be confusing to newcomers. If this is your first time using code, we recommend starting with beginner’s courses with scratch.

CYF is a fork of Unitale. You do not need a copy of Unity to run or make your own Undertale battle, but it will be necessary for the overworld. You can get around this with a mod of CYF that has a scripting version of the overworld.

Unity, however, has some limitations and was built originally for 3D. It is not as easy to learn as scratch and uses scripts instead of visual programing blocks.

**Price: Free**

**Undertale: Battle/Overworld**

[**Kristal (LOVE2D)**](https://nyako.gay/kristal/)

Kristal is built for Deltarune fangames only. *(Theoretically someone could patch it to become like Undertale)*

Like CYF/CYK, it uses LUA as its programming language. However, it uses [LOVE](https://love2d.org/) as its program. It’s important to know LOVE uses scripts instead of a visual editor. We recommend writing the code using [Visual Studio Code](https://code.visualstudio.com/) or [Notepad ++.](https://notepad-plus-plus.org/) For the visual component, [find a pixel art program](https://www.youtube.com/watch?v=90BghUX7SD0) you like and build the maps using [TileD](https://www.mapeditor.org/).

**Price: Free**

**Undertale: Battle/Overworld**

**Click team Fusion**

**(link)**

A Drag-and-Drop-based 2D game engine. Used by multiple Undertale fan-game developers. Built-in with some shader programs for the screen effects. However, it is not recommended due to the difficulty of maintaining your own game project.

I don’t know enough to talk about it. Jevil, help me out? XD

# Coding Conventions

Here are some tips from TML.

**Naming**

Use *“snake\_case”* for the resources except for [*“scripts”*] and for all [*“variable”*] names. Add *“\_”* for [*“internal variables”*] that shouldn't be accessed outside the [*“object,”*]

Example: *“ \_timer”*

Use *“PascalCase”* for naming [*“scripts”*] and [*“functions.”*]

**Variables of the engine built-in objects**

These do NOT directly control the [*“internal variables”*] (which start with \_) of the engine built-in objects

Example: *“battle.\_state;”*

Always use the [*“functions.”*] provided by the engine

Example: *“Battle\_SetState();”*

Public variables (which don't start with \_) are free to write.

Example: *“battle\_board.angle = value;”*

**Turn preparations**

You should use the *“battle\_turn”* [*“objects”*] in the *“Turn Preparation Start”* [*“object event”*] within the “battle\_enemy” [*“object”*].

Then use the [*“function”*] *“Battle\_SetTurnInfo(info, value)”* in *“Preparation Start”* [*“object event”*] within the *“battle\_turn”* [*“object”*] to set the turn’s properties

Example: *“Battle\_SetTurnInfo(BATTLE\_TURN\_INFO.BOARD\_LEFT, 200);*

*Battle\_SetTurnInfo(BATTLE\_TURN\_INFO.TIME, 500);”*

# 

# Examples And Free Resources

**(Be sure to download the most recent version of the engine. Before the newest release 1.0, examples must be downloaded as a separate project.)**

# [Examples](https://github.com/TML233/undertale_engine_legacy/tree/examples)

Within is a *“gaster-blaster attack”* and a “*simple battle*.” Please download this version instead if you intend to have battles in your game and are still learning.

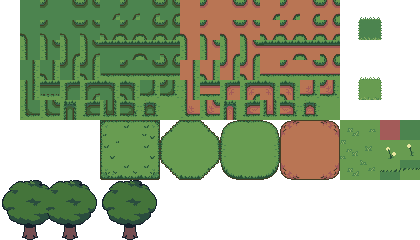
# [Ruins Example by TML](https://github.com/starlightshore/UTE-Ruins-Demo-by-TML)

The Ruins Example is a project that TML (UNDERTALE ENGINE’s original creator) had made but had lost the source code. However, an .exe file had been found. Using UndertaleModTool, starlightshore then decompiled the .exe and remade the whole demo within [GM.]

Demo contains the first 5 rooms of undertale, minus the Flowey fight. This Demo is for studying how to use the UNDERTALE ENGINE only.

# Downloadable Tileset Example

Head to [tileset] for more information on how to use a tileset.



*[IMG ID: Free to use Tileset of some grass and dirt. End ID]*

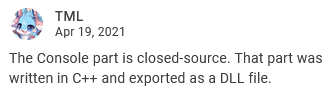
[Download it here.](https://imgur.com/a/vqKOdFl)

# 

# Licensing

UNDERTALE ENGINE - By TML.

This is a fangame engine for the game UNDERTALE by Toby Fox. It is built for GameMaker, (formerly known as Gamemaker Studio 2) and is a free project to learn and create your own fangames within. To use, you **must** credit **TML** as the creator of this engine. It was released under the MIT license, with some being open source.\*

**

*(TML: The Console part is closed-source. That part was written in C++ and exported as a DLL file.)*

The UNDERTALE Engine had been abandoned in May 2019. The project had been resurrected with its new owner, Jevilhumor. New updates and examples are planned and in the process of being released. Documentation has been made from TML, Jevilhumor, Starlightshore, and Hatty.

All fangames must be published free and not for profit. We strongly recommend others to support UNDERTALE and Toby Fox by buying the game or his merch to show appreciation. We do not intend to be a replacement for the original game, and are all here to create fanworks for fun and for educational purposes.

Always credit your sources and ask for permission when using other’s work. You may use The Spriters Resource ([UT](https://www.spriters-resource.com/pc_computer/undertale/) - [DR](https://www.spriters-resource.com/pc_computer/deltarunechapter1/)) to use canon sprite assets.

Non-UNDERTALE Engine Youtube tutorial videos that are linked within these pages are not in any way affiliated with this Engine or its creators.

# Backing Up Your Work

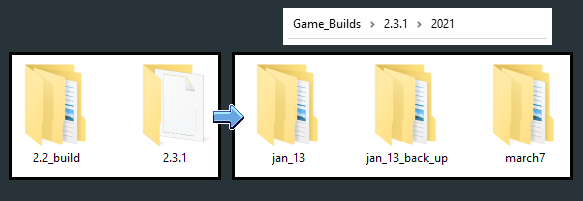
We recommend not upgrading GameMaker Studio 2.2 to 2.3. 2.3 Is still in its beta stage, and therefore is subject to corruption and changes.

To prevent loss of work, either do not upgrade the program or follow this advice: Back up your work.

You may either use GitHub and/or back up your work by copying your project into a new folder.

However, with the complexity of corruption of game files, It is also recommended you make physical backups. You can copy the folder of the game engine and paste it into a new folder, labeled “back up (current date).” Regardless of use of 2.2 or 2.3, it is good practice to back up your work frequently, like whenever you attempt to do a new task.

An example is pictured below.



*[IMG ID: 5 folders are shown. The first two are labeled with the GameMaker Studio 2 type of build. A blue arrow then points to what is inside them, 3 more folders which are labeled with the date of the project’s backup. End ID]*

When moving to GameMaker Studio 2.3, please note that [scripts](https://youtu.be/9nwlgfzyNzA) are done differently.

# GM vs. GMS 2.2. vs GMS 2.3

As of 2022, Gamemaker Studio 2 has changed to Gamemaker. It’ll be referred to as such, or as GM for short. It is the same as GMS 2, the only difference is the name and branding. As GM changes, the engine (which was made in GMS 2.2) will become further outdated. Please use the “modernized” version of the engine as the game will break otherwise.

If your project was made in 2.2 and you wish to move to 2.3, please do so with caution. There are a few game breaking bugs in upgrading, as well as 2.3’s general state of change. It’s glitchy (and able to be corrupted) and not as fine-tuned as 2.2.

To prevent loss of projects, either do not upgrade the program or follow this advice: Back up your work. When creating a 2.3 version of your project, GameMaker Studio 2 will create a new project file that can only be opened within 2.3. You may have both 2.2 and 2.3 versions of GameMaker Studio 2 installed at once.

The conversion should run perfectly fine, however if your project doesn’t work as intended then you may need to redownload it from Github. This is a rare case but it does happen from time to time.

There are a few issues from moving from 2.2 to 2.3.

To learn the IDE changes for Gamemaker Studio 2, [watch this video.](https://youtu.be/SwQWQYVJQmY)

### Console Break

*(Written by TML)*

New versions of GameMaker seemed to break the console. Goto the script Console\_Custom, and set the macro GMU\_CONSOLE\_ENABLED to false to disable the console.

### Text Scaling Issues

*(Written by TML, updated by Sam)*  
In late 2021, Yoyogames' GM new version breaks the text\_typer because it adds a true bool type. Fortunately, It's very easy to fix:

1. Go to text\_typer User Event 2 At **lines: 126** command shadow **132 command outline 196** command instant **202** command skippable **412** command gui **559** command skip\_space
2. Change is\_real to is\_bool

This concludes the fix.

Note: Using those commands, you must use true or false to make the command work, e.g., {instant true} and {instant false} will work, but {instant 1} or {instant 0} won't work.

Attacking Enemy visual error

1. Open the object “*battle\_menu\_fight\_knife*”
2. Open the step event.
3. Head to line 9: “*Anim\_Destroy(self,"\_aim\_x");*”
4. Change “*self*” to “*id*”

# Credits for Documentation

*(listed in alphabetical order)*

**Jevil:**

Current owner of the engine. Helped Starlightshore finish the documentation.

**Hatty:**

Created many google documents/text files to document this engine. Also contributed to code for the engine. Adapted from discord to Documents.

**TML:**

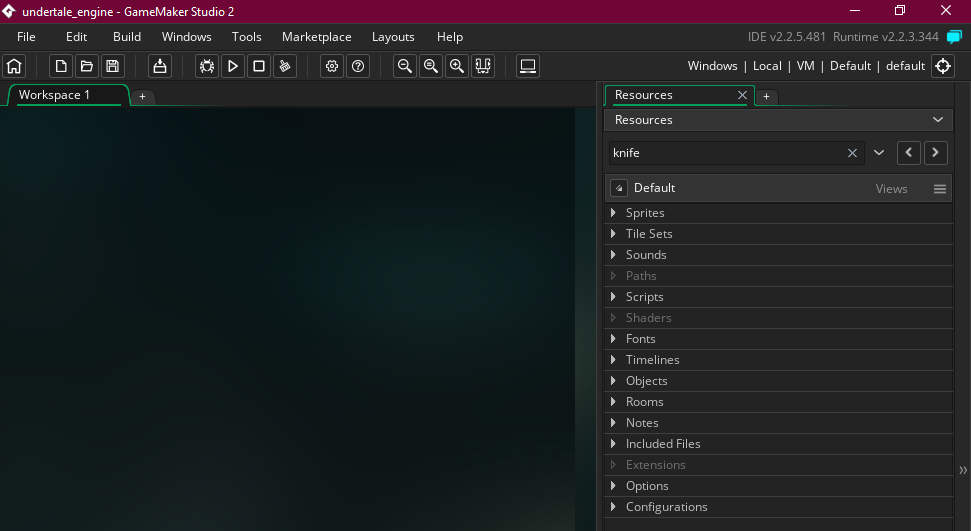
Original owner of the engine. Had abandoned the project in 2019. Wrote some of the pages on the documentation. However, TML is still active on discord and some of the new text has been adapted in the documentation since.

**Starlightshore:**

Written the bulk of the documentation in 2021.

GM Basics

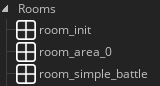
Depending on your experience with code and other game engines, the journey to becoming accustomed to GM can be daunting. However, it’s a steep learning curve. Once you learn the basics of code and how to navigate GM, you will find it freeing and easy. For video learning, watch videos from this [playlist.](https://www.youtube.com/c/letslearngamemaker/playlists) Gamemaker also has an in-depth manual. [(2.2)](http://docs2.yoyogames.com/source/_build/index.html) [(2.3)](https://manual-en.yoyogames.com/#t=Content.htm)



*[IMG ID: A screenshot of GameMaker Studio 2.2, edited to be smaller and fit better. End ID]*

Here, we have our work space. Here is the [manual page.](http://docs2.yoyogames.com/source/_build/1_overview/2_quick_start/3_workspaces.html) The main difference between 2.2 and [2.3’s workspaces,](https://manual-en.yoyogames.com/#t=Introduction%2FWorkspaces.htm) is that room order is now controlled through the room manager. Otherwise, using either one will be a similar experience.

For the UNDERTALE Engine specifically, you will see these rooms\*:



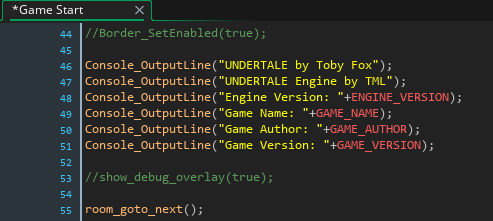
*[IMD ID: List of rooms. room\_init, room\_area\_0, and room\_simple\_battle End ID]*

(\* Do note the “room\_simple\_battle” must be downloaded from the “examples” branch. This is explained [here.])

# 

# room\_init:

This room must always be the **first room of the game.** In 2.2, it has to be the first room listed. In 2.3, in the room order it must be first with the House icon. This game initializes important information like variables and scripts to be running properly. Without it, you will get an error. This room contains the persistent [*“object”*] *“world.”* We recommend avoiding messing with this object unless you’re more well versed with code.

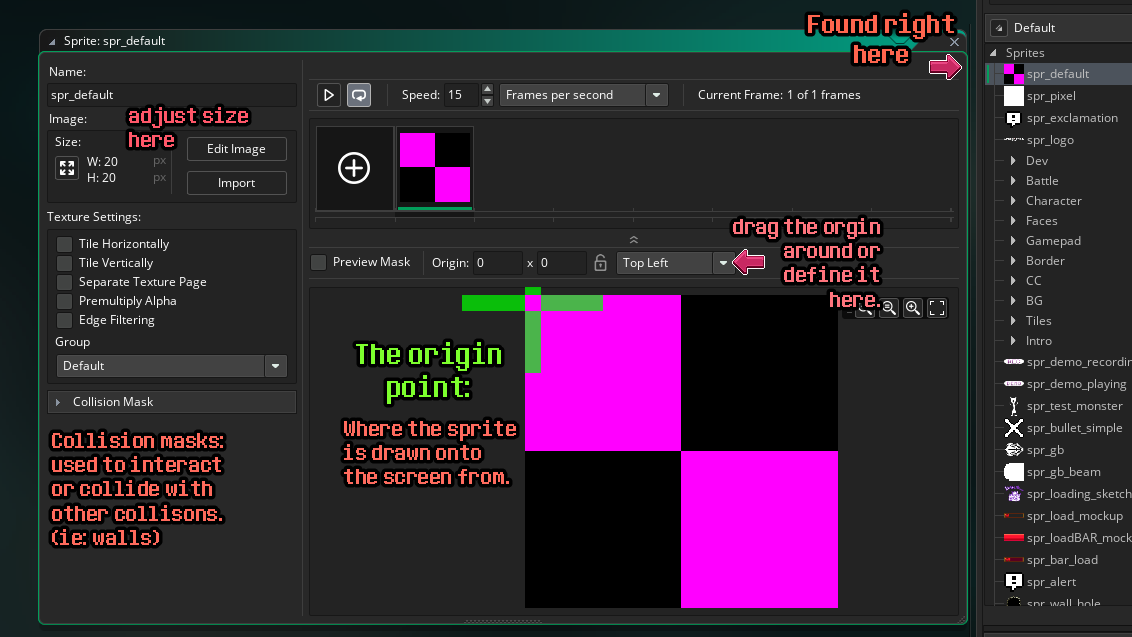


*[IMG ID: A screenshot of the World object’s Game Start event. It reads both the //Border\_SetEnabled(true); and //show\_debug\_overlay(true); functions at line 44 and 55 respectively. End ID]*

In the Game Start event, you may turn on the [*“functions”*]: *“Border\_SetEnabled”* and *“show\_debug\_overlay”* if you wish. However, in the soon-to-be-released latest version of UNDERTALE Engine, this will be togglable within the [Settings.] The *“//”* characters comment out code, making it so the program does not run. To simply uncomment, remove them.

If you’d like to comment out code to leave notes for yourself, you may use *“//”* or *“/\** Commented out \*/” to comment blocks of code between the *“/\*”* and *“\*/”.*

# Sprites:

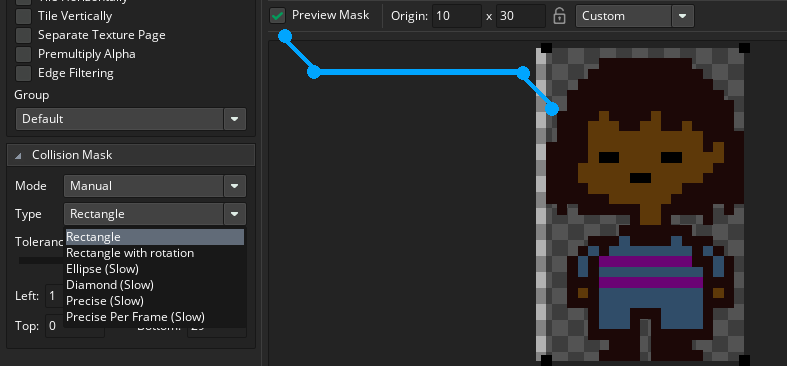
[**[manual]**](https://manual-en.yoyogames.com/#t=Quick_Start_Guide%2FCreating_Sprites.htm)[**[video]**](https://youtu.be/fA3tJo0vXjU)

*[IMG ID: A screenshot of GameMaker Studio 2 that shows where Sprites are located, and details how to move the Origin point, where Collison Masks are located and where to adjust the size and scaling of the sprite. End ID]*

Sprites are the images we use to add visuals to the game. It is recommended you name your files with “*spr\_*” at the start. GM relies on the names of sprites and objects, so you can not have both named the same thing. Differentiating your sprite with “*spr\_”*, and your objects with “*obj\_* “ is an easy naming convention to prevent this error.

Sprites are drawn onto the screen by their *Origin Points.* (Think of how you hold a piece of paper, the origin is wherever your hand would hold the paper.) This can be adjusted by manually dragging the origin, or by putting in new numbers or selecting from 9 options. Ie: Top left -> center -> Bottom right and the inbetweens of each.

Collisions are also defined here within the sprite.



*[IMG ID: A screenshot of UNDERTALE Frisk, placed underneath a collision mask. To the left, we see the options for the Collision mask, ranging from rectangle, ellipse, diamond and precise. End ID]*

By selecting “Preview Mask” you can see the mask as depicted by a transparent black rectangle. You may choose from different types of collision shapes, such as “rectangle,” “Ellipse,” “Diamond,” and “Precise (AKA pixel perfect).”

Rectangle is the best to use by default, as it’s a simple shape that runs against others easily as well as being great for computer performance. While the others will run slower on your PC, unless you have a *lot* of collisions running at once this will not be an issue. Despite that, unless for specific circumstances where you need specifically different shaped collisions, use a rectangle.

One example where you’d need specifics would be for bullets within battle.

You may adjust the size of the canvas or the sprite itself on the left side of the sprite window. Alternatively, you can import a new sprite to replace the current one.

Sprites use *frames* to animate.



*[IMG ID: 4 frames of UNDERTALE Frisk running downwards, facing the player. Each frame is labed from 0-3. There are 4 frames total. End ID]*

GameMakerStudio keeps track of frames starting at 0, this will be important to know when referring to code relating to sprites.

To easily import multiple frames, you can take the frames and create a horizontal strip with each sprite frame equally placed next to each other. Name the file in the format of *“spr\_spritename\_strip#”* (number being the amount of frames in the animation) before being imported to GM. Once inside the engine, you may remove the “*\_strip#*”.



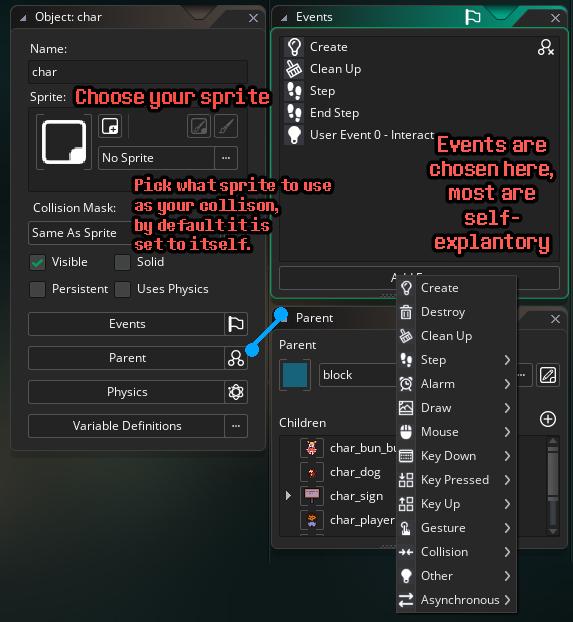
*[IMG ID: 4 sprites of Alphys walking to the right, from the BIG example project.]*

# 

# Objects:

[**[manual]**](https://manual-en.yoyogames.com/#t=Quick_Start_Guide%2FObjects_And_Instances.htm)[**[video]**](https://youtu.be/kvy03HYyJGM)

Objects are what we use to run code within GMS 2. They can be found in the Object folder, located between Timelines and Rooms. To use an event, you have [“Object Events.”] Events are where we write code onto the object, and each event has a different property unique to itself.



*[IMG ID: Screenshot of an object. We see its family tree, where to choose sprites for itself and its collisions, and the object’s events. End ID]*

Objects can have sprites, but it is not necessary to have one. You may also choose a sprite to become its collison mask. Collisions are how we interact with the game’s world. Either by talking or by being blocked by. Collisions are defined within the sprite chosen here, and can be any sprite.

For objects that use multiple sprites (like a NPC who turns around) we recommend using a specific sprite, to prevent issues with the changing collison box.

Notably, you may also [inherent] objects as children or parents. There is also a [variable] window for that specific object. Each of these are explained further in the links provided above.

# 

# Object Events:

[**[manual]**](https://manual-en.yoyogames.com/#t=The_Asset_Editors%2FObject_Properties%2FObject_Events.htm)[**[video]**](https://youtu.be/bL3qFDWLFX8)

There are many events but don’t be discouraged, most are self-explanatory. These are places to run code, and the event is telling the game engine when and how to run it.

**create:**

Code that is run when the object is created. In other words, it’s run only when the object is placed into the scene, so it’s great for initializing [variables.]

**destroy/clean up:**

These two events are similar. Destroy is code that is run when the object is destroyed. Clean up is code run once the object is gone, and is good for preventing memory leakage.

**step:**

This is an event that runs code on every frame per second. By default, it’ll run on 60 frames per second. In other words, that’s running code 60 times in a second, and 3,600 per minute. Because it is constantly running, this is great for checking for player input.

**alarm:**

Code that is run after a certain number of frames. You have up to 11 alarms to use within one object. To use, simply put *“alarm[#] = # of Frames desired”* So, if I wanted this alarm code to run after a second, i’d simply say *“alarm[1] = 60;”*

**draw:**

This is code that affects how the image is drawn. It’s a special event where, when selected, you need to type “*draw\_self()*;” to draw your sprite. Otherwise, when this event is open, the game won’t be told to draw it. If you don’t have this event, this piece of code will be run by default.

You can add sprites with “draw\_sprite();” or “draw\_sprite\_ext();”. Depending on which is run first, the sprite will be drawn over the other unless stated otherwise within its depth. [see functions]

**mouse/key down/key pressed/key up/gesture:**

Code that is run when certain inputs are called. These inputs can also be used with [functions] such as *“if(keyboard\_check(vk\_enter)){ //code}”* which will check if the player has pressed the enter button.

**collison:**

Code that is run once this object interacts with a different specific object.

**other:**

There are multiple events under the Other category. To read about them individually, go [here.](https://manual-en.yoyogames.com/#t=The_Asset_Editors%2FObject_Properties%2FOther_Events.htm) For this engine, we don’t need to cover this for now.(?)

# Object hierarchy:

Objects can be within a hierarchy. When giving an object a parent, the object inherits the parent’s events and code. Think of how children can inherit the eye color of their parents. The events will be grayed out, showing they’re the same. You may then *“override”* it or *“inherit”* the event. *“Override”* will make it so that specific events aren't inherited, while *“inherit”* makes it so the code *“event\_inherited;”* is placed. This code represents all the code of the parent, and as long as the event has it, it will keep the previous object’s code.

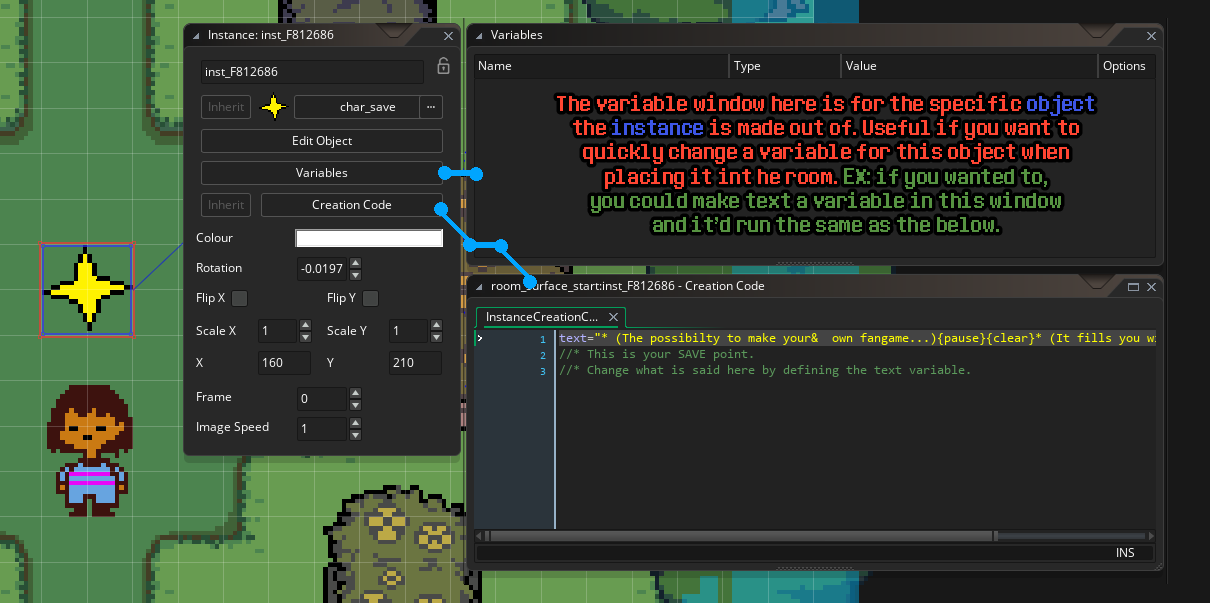
The inherited code will be variables, therefore if you inherit *“block”* then *“block\_enabled=true;”* will be kept true for this child.

Certain objects will be required to be children of certain objects. Most notable ones are: [*“block,”*][*“char,”*] and [*“face.”*]

# 

# Instances:

[**[manual]**](https://manual-en.yoyogames.com/#t=Quick_Start_Guide%2FObjects_And_Instances.htm)[**[video]**](https://youtu.be/kvy03HYyJGM)



[IMG ID: Screenshot Of Frisk standing by a SAVE point instance. Its instance window is opened, showing both the variable and creation code windows attached. End ID]

Instances are objects that are placed inside a room. Instances are not the actual object, but rather, a copy of the object. For each instance of the instance, you can set up specific code to run for itself that won’t affect the original object. This way, you can have multiples of the same object, and have each run or look very different from each other.

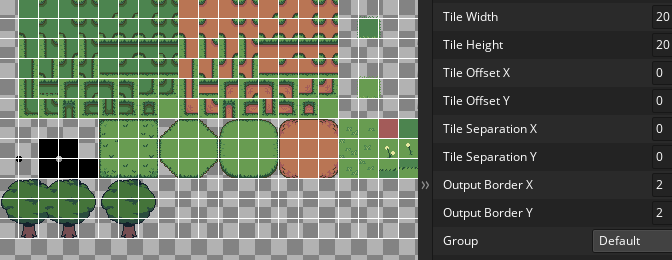
You can manipulate the instance I'm its visuals by: x and y positions, if it's flipped, it's color, it's starting frame and animation speed.

In the drop down windows you can open up the variable window and create code. The variable window is connected to the object, and within the instance of it you can change the values relating to that specific instance. Do note you have to manually set up the variables in the object first, and the variables will only relate to said object. You set up a default value, then with the instance you can change it if you so wish.

The creation code is code only run once that instance is created. Like the variable window, it uses the Instance's object's variables. However, you can refer to other object's variables with *"obj\_name.var\_name."*

# Tilesets:

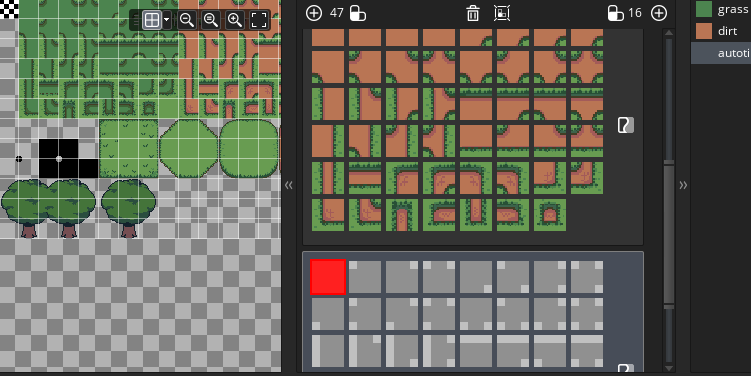
[**[manual]**](https://manual-en.yoyogames.com/#t=The_Asset_Editors%2FTile_Sets.htm&rhsearch=tilesets&rhhlterm=tileset)[**[video]**](https://youtu.be/KKq8Opxi0y4)



*[IMG ID: Tileset for UT engine. It’s a modified deltarune tileset, making for a spring grassy look. It’s built for both auto-tile and manual placement. The tileset width and height are shown to be 20x20. End ID]*

Tilesets are how overworld sprites are laid out to create the environment. It uses a tile- based grid (in the case of Undertale, it’s 20 pixels wide and high) You place a single tile of that size onto a tileset layer, and it will draw it onto the screen like a stamp. You may use as many layers as you like, each set to a different tileset. **Please note** the first tile of the set is left blank. This is because GameMaker Studio 2 uses the first tile to store important information, therefore it will be invisible.

To have your tiles animated, you must have all the frames of the animation within the same tileset and click the *“tile animation”* button. From there, pick the frame rate (how fast the animation is played per frame) and the amount of frames. Then simply click each tile for your animation. It can be played both in the editor by hitting the *“play animation”* button at the top right of the room’s mapping area or by playtesting your game.



*[IMG ID: A screenshot of the auto-tile window. It shows a gray and white grid made of squares and rectangles, then a path of dirt among some grass that fits the pattern of the grid. End ID]*

To easily paint tiles that are adaptive (like the image above) you must use the “auto-tile.” There is a grid pattern you must follow, and in the example tileset, you will see that both the grass and dirt auto-tiles follow the grid perfectly.

# 

# On Music and Sounds

[[manual]](https://manual-en.yoyogames.com/#t=The_Asset_Editors%2FSounds.htm&rhsearch=sound&rhhlterm=sounds%20sound%20audio_sound_is_playable)[[video]](https://youtu.be/XkJbX6lNz0g)

Audio Playbacks: Use the “*BGM\_XXX”* functions to control music playback. *“BGM\_Play(slot, sound)”* slot is the music channel.

**Note** that *“BGM\_Step”* is **only for internal use**, to update the BGM system. **Do not use this function yourself.** This remains the same for other functions like *“Anim\_Step”*.

## The BGM Slot System (How does UT Engine handle background music (BGM)?)

There are 6 slots in the engine (0~5).

* **Slot 0 is for general overworld music.**
  + The object *“hint\_bgm”* plays music at slot 0.
  + An example of using the “hint\_bgm” would be to place an [instance] of the object in *“room\_area\_0.”* In the instance’s [creation code] write *“bgm = audio\_name;”*
* **Slot 5 is for battle music.** Encounters and Battles uses slot 5 to play and control battle music.
* **Other slots (Slots 1-4) are free to use.**

## How to Play SFX

*“audio\_play\_sound(soundid,priority,loops);”* is only for sound effect playbacks.

* ***soundid*** *-*  name of the sound file
* **priority** - For if you have multiple sounds and want one to be played instead of another, otherwise it should be equal to 0.
* **loops** - If you want the SFX to repeat indefinitely, loops = true; otherwise = false.
* e.g. *audio\_play\_sound(snd\_break\_1,0,false);* 
  + *//Plays the snd\_break\_1 sound once.*
* **Please note:** Take caution when using *audio\_play\_sound* in a **Step** event! Because the step event will happen 60 times per second, the sound will also play 60 times per second, which is often unintended and sounds bad.
  + **To prevent this**, use an **alarm** or toggle a [variable] within a [if statement.] Here is an example:

**[create] event:** ready = true; //code run at the start of this object’s creation

**[step] event:** if ready= true{alarm[0]=1;}//this checks if the ready variable has changed to true or false at any time.

**[alarm] 0 event:** audio\_play\_sound(snd\_dununnn,1,0); ready = false; //plays the audio only once, then turns ready off so it doesn’t repeat in the step event.

## Level 1: How to add BGM to an Overworld Room

**0. Add your sound file (typically .ogg) to the list of assets.**

e.g. Add "ruins.ogg", the Ruins overworld theme.

Optional: rename the sound file to have "bgm\_" as a prefix to help separate your music tracks from other sound files (e.g. "bgm\_ruins")

**1. Add the object "hint\_bgm" to the room.**

**2. hint\_bgm > Creation Code**

| **Creation Code** |
| --- |
| bgm = [filename];  //e.g. bgm = bgm\_ruins;  //This theme will play when you load into the room.  pitch = 1;  //This lets you adjust the pitch of the BGM.  //1=normal pitch,  //0.5=lower pitch,  //2=higher pitch. |
|  |

## Level 2: How to add silence (no BGM) to an Overworld Room

**WHY?** If you start in one room with a hint\_bgm loaded with BGM (e.g. bgm\_ruins) and then leave to another room without hint\_bgm, the BGM from the first room will continue to play!

0. Create a new sound, but don’t add any sound to it.

* Optional: rename file to have “bgm\_” as prefix (e.g. bgm\_nobgm)

**1. Add the object hint\_bgm to the room where you want silence.**

**2. hint\_bgm > Creation Code**

| **Creation Code** |
| --- |
| bgm = [filename of sound created in Step 0];  //e.g. bgm = bgm\_nobgm  //When the player enters this room from another room which had BGM, that music will stop! |

## Level 3: How to Manipulate BGM in an Overworld Room with Dialogue (or, the Power of Text Typer)

**WHY?** Perhaps you’re in a room. The Ruins theme is playing, and you speak to a Froggit. The Froggit says something very sad, and the **BGM suddenly cuts out** before being replaced with a **lower pitched version** of the Ruins theme. Then the Froggit asks if ANIME IS REAL, RIGHT?? and **the pitch of the BGM goes up** **super high**! Then the conversation finishes, and **the BGM goes back to normal**.

1. Go to Objects > Character > **char\_sign**.
2. Duplicate char\_sign, rename to “char\_sign\_sequence\_changebgm”.
3. Place char\_sign\_sequence\_changebgm in a Room where the player may interact with it.
4. Select it, and check its Events and Creation Code.

| **Create** - Four(4) new variables will be set: |
| --- |
| event\_inherited();  dir\_locked=true;  text0=""; //renamed text to text0 for name consistency with text1 and text2.  text1="";  text2="";  bgm\_slot=0;  bgm=-1; |

| **User Event 0** - Loads text0, text1, and text2 into the Text Typer, to be printed for the player to read when they interact with the sign. But wait, where’s the code for the BGM/music? You’ll see very soon..! |
| --- |
| event\_inherited();  Dialog\_Add(text0); //Loads text0 to be printed onto the textbox by Dialog\_Start().  Dialog\_Add(text1); //Ditto text1.  Dialog\_Add(text2); //Ditto text2.  Dialog\_Start(); //When the player interacts with this object, text0 will be printed first, followed by text1 and text2. |

| **Creation Code -** Where the real magic happens! |
| --- |
| **//Let’s break down the following line:**  **// text0=”{script *“+string(Music\_SetPitch)+”* “+string(bgm\_slot)+” 2}BGM at double pitch.”;**  //**{}** - The curly brackets allow for **Commands** to be run as text is being typed on screen!  //**{script**} - a Command that lets you call a Script like Music\_SetPitch (aka BGM\_SetPitch) when text reaches a specific location.  //This is useful because **you don’t need to create new objects as often**  for different scripted dialogue: you can just have the magic in **Creation**  **Code**, which can be set for each individual instance!  // **string(variable)** - a function that takes a variable of any type (e.g. Music\_SetPitch, bgm\_slot), and **converts it to string**. This concatenating/attaches the variable (e.g. bgm\_slot=0) to the command.  **//Must be surrounded by +, e.g. +string(variable)+ for this to work!**  //text0=”{script *Music\_SetPitch* bgm\_slot 2}BGM at double pitch.”;  //The above line will cause an error.  //**Music\_SetPitch(bgm\_slot, pitch, …)** - A function that lets you change the pitch of the music currently playing in a certain bgm\_slot.  // A pitch value of **1** is the normal pitch for music. **Higher numbers increase pitch, lower**  **numbers decrease it.**  //e.g. if bgm\_slot = 0 and bgm = bgm\_ruins (ruins.ogg),  // Music\_SetPitch (0, 2) would result in a higher-pitched Ruins theme.  **//Considering all of the above, text0 will be set to “{script Music\_SetPitch 0 2}BGM at double pitch.”**  **//Intended Sequence of Events:**  //Player interacts with an instance of char\_sign\_sequence\_changebgm in a Room.  //The text **“BGM at double pitch.”** will be printed on the textbox, and the current BGM’s **pitch** will suddenly be **high**.  //Player presses CONFIRM(Z). The textbox clears and a new line **“BGM at half pitch.”** will be printed on the textbox, all while the BGM’s **pitch** is suddenly **low**.  //Player presses CONFIRM(Z) again. The textbox clears and a new line **“BGM at normal pitch.”** will be printed on the textbox, and the BGM **pitch** is set **back to normal.**  //Player presses CONFIRM(Z) once more. With no more text to type and BGM pitch back to 1, things are back to normal and the player can move freely again. |
|  |

## How BGM Works in Undertale Engine:

**Scripts > Music > BGM\_Play**

| ///@arg bgm\_slot  ///@arg audio  ///@arg loop\*  ///@arg loop\_start\*  ///@arg loop\_end\*  function BGM\_Play() {  var SLOT=argument[0];  var AUDIO=argument[1];  var LOOP=true;  var LOOP\_START=-1;  var LOOP\_END=-1;  if(argument\_count>=3){  LOOP=argument[2];  }  if(argument\_count>=4){  LOOP\_START=argument[3];  }  if(argument\_count>=5){  LOOP\_END=argument[4];  }  ***//***  **if(Music\_IsSlotValid(SLOT)&&audio\_exists(AUDIO)){**  **Music\_Stop(SLOT);**  **global.\_bgm\_audio[SLOT]=AUDIO;**  **global.\_bgm\_id[SLOT]=audio\_play\_sound(AUDIO,1,LOOP);**  ***// This part plays the actual music!***  global.\_bgm\_loop\_start[SLOT]=LOOP\_START;  global.\_bgm\_loop\_end[SLOT]=LOOP\_END;  return true;  }else{  return false;  } |
| --- |

**Objects > Hint > hint\_bgm > Events**

| **Create** | bgm\_slot=0;  bgm=-1;  pitch=1;  alarm[0]=1; |
| --- | --- |
| **Alarm 0** | if(audio\_exists(bgm)){  if(!BGM\_IsPlaying(bgm\_slot)||(BGM\_IsPlaying(bgm\_slot)&&BGM\_GetAudio(bgm\_slot)!=bgm)){  BGM\_Play(bgm\_slot,**bgm**);  BGM\_SetPitch(bgm\_slot,**pitch**);  }  } |

**Room > Add object hint\_bgm > Creation Code**

| **bgm** = bgm\_ruins;  **pitch** = 1; |
| --- |

# 

# Scripts:

[**[2.3 manual]**](https://manual-en.yoyogames.com/#t=The_Asset_Editors%2FScripts.htm&rhsearch=scripts&rhhlterm=scripts%20script%20scripting)[**[video]**](https://www.youtube.com/watch?v=i1xnHjbbURk&list=PLhwAMKTBx5mU0TPiYkeZVx5uPfz4KrHTH&index=12)

Scripts are code that are able to run anywhere and at any time within the game. These are great for setting up [functions], [marcos] and [FLAGS].

[Please note that Functions and Scripts have changed in GameMaker Studio 2.3.](https://youtu.be/9nwlgfzyNzA)

# 

# Variables:

[**[manual]**](https://manual-en.yoyogames.com/#t=GameMaker_Language%2FGML_Overview%2FVariables_And_Variable_Scope.htm&rhsearch=variable)[**[video]**](https://youtu.be/UxPG9--Ggts)

Variables are code that contain a value. We need variables to be defined before usage, as an undefined variable will crash the game as the computer won’t know what to make of it.



*[IMG ID: Screenshot of a yellow sign warning that Gamemaker warns against within the code. End ID]*

If a variable hasn’t been used before, the game helpfully alerts you with a yellow “!” sign. Useful if you make a typo, or if you forgot to define it beforehand.

Variables can be checked through [*“if statements”*] and [*“switch statements”*].

Below are some values variables may have. For more information go [[here.]](https://manual-en.yoyogames.com/#t=GameMaker_Language%2FGML_Reference%2FVariable_Functions%2FVariable_Functions.htm&rhsearch=variable)

Variable = “” //a variable that contains a string, “text within parentheses.” Great for dialogue.

Variable = # //any numerical value, like 1,25,87 ect.

Variable = -1 //a special value. -1 is used as a placeholder, and not an actual value. Think of it as a “skip” but works to prevent errors as it’s not undefined.

Variable = true/false //A boolean value (data that only has 2 outcomes).

There are 3 types of variables. (Constants aren't’ variables, but are important to know here) [**[video]**](https://youtu.be/9TUpOHMtBzQ)

**Local Variables:** [**[manual]**](https://manual-en.yoyogames.com/GameMaker_Language/GML_Overview/Variables/Local_Variables.htm)A variable that only runs within the [function] or [object event] it’s written in. Once run, it will discard itself. Example: *“var local\_variable = value;”*

**Instance Variables:** [**[manual]**](https://manual-en.yoyogames.com/GameMaker_Language/GML_Overview/Variables/Instance_Variables.htm)A variable that only runs within the [instance] of an object. Different instances of the same object can contain this variable without affecting each other.

Example: *“variable = value;”*

If you would like to modify an instance variable from a different object, you may use the following code: *“object\_name.variable\_name.”*

***Warning****!*

Be sure not to use a [*“built-in variable”*] on accident. Just use a *“\_”* to prevent it from becoming green.

**Global Variables:** [**[manual]**](https://manual-en.yoyogames.com/GameMaker_Language/GML_Overview/Variables/Global_Variables.htm)A variable that can be used anywhere. EX: *“global.variable = value."*

Warning: These variables are not saved between reload/resets. For variables that can be remembered between game sessions, look into *[“FLAGS.”]*

**Constants:** [**[manual]**](https://manual-en.yoyogames.com/GameMaker_Language/GML_Overview/Variables/Constants.htm)A value that is hard coded into a constant value that can never be changed. There are two types of constants: *[“Marcos”]* and *[“Enumerators.”]*

# 

# Marco:

[**[manual]**](https://manual-en.yoyogames.com/GameMaker_Language/GML_Overview/Variables/Constants.htm)[**[video]**](https://youtu.be/FOnXqCPIma0)

A Marco is a piece of code used within a script that can be called anywhere. Useful for things that are static, like the tileset size.

EX: *“#marco TILESET\_SIZE 20”*

Note that macros don't exist in the exported games. They will be inline after compiling.

# 

# Enumerator (Enum):

[**[manual]**](https://manual-en.yoyogames.com/GameMaker_Language/GML_Overview/Variables/Constants.htm)[**[video]**](https://youtu.be/HSTPv-wWIV0)

An *“Enumerator”* (or Enum, for short) is a list of constant values. Example:

enum PIE\_FLAVORS {

CHERRY,

BUTTERSCOTCH,

BLUEBERRY

}

What makes enums special is that when you edit an enum, as long as the enum field name doesn’t change, the values of every other enum field with the same name will sync together.

For example, you have this enum at the beginning:  
enum PIE\_FLAVORS {   
 CHERRY, // This will get the value of 0  
 STRAWBERRY // This will get 1  
}  
And at one time, you decided to add a flavor between CHERRY and STRAWBERRY:  
Enum PIE\_FLAVORS {  
 CHEERY, // This remains the value of 0  
 BLUEBERRY, // This will get 1.  
 STRAWBERRY // This will get 2 instead of 1, because it’s no longer the second field of the enum.  
}  
But you don’t need to search for every code piece you mentioned for the STRAWBERRY flavor. They automatically get the new value of 2. This makes maintenance much easier.

These are useful because we’re looking at **names** and **words** instead of **numbers**, and therefore are easier to remember and call onto.

To use an Enum, use the Enum name (dot) value.

EX: PIE\_FLAVORS.CHERRY

You use the same logic for referencing [*“variables”*] in different [*“objects”*] than the one you’re writing in.

# 

# Built In Variables:

(See [Coding Conventions])

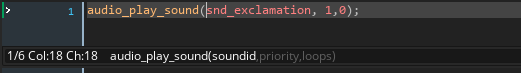
Variables when used, will turn green and refer to engine-specific variables. Can only be used for in-engine purposes, and often contains specific information for that value. An example would be “x”, as the x value will define the coordinate of the sprite or object. If you would like to have a custom variable named x, add a “\_” to it. Making it a “\_x = custom variable;” and will no longer be connected to the built-in variable.

# 

# Functions:

(See [Coding Conventions])

[**[manual]**](https://manual-en.yoyogames.com/#t=GameMaker_Language%2FGML_Overview%2FScript_Functions.htm&rhsearch=functions&rhhlterm=functions%20function%20%40function)[**[video]**](https://youtu.be/BH4kX57w8aM)

****

*[IMG ID: A screenshot of the audio\_play\_sound function within the code editor window. on the bottom left of the window is the function with its parameters listed inside End ID]*

Functions are the way GameMaker Studio 2 runs code within specific conditions. These are usually defined in the [“scripts,”] though in GameMaker Studio 2.3 [scripts and how Functions are called have changed.](https://youtu.be/9nwlgfzyNzA)

The image above is an example of a built-in function. These are functions GameMaker Studio 2 has stored by default, though you can create your own, just as TML has. Functions have *“parameters*” which are defined within the *“().”* Within the parameters, there will be one or multiple *“arguments”* (the term parameter and argument are often referred to as the same thing) which the function uses to ask for certain values.

To find out what the parameters are asking for, there are two methods to do so.

1)middle click (on most mice it is the spinning wheel.) to go to the original script where the function is made (and if it’s a built in function, then it will take you to the GM manual page)

2) look to the bottom left hand corner of the code window, as seen in the image above.

The example above *“audio\_play\_sound(soundid,priority,loops)”* is asking for 3 values to be defined. The Sound Id (the audio file name), the priority (a numerical value that will compare to other sounds, deeming which is more important to be heard), and loops. (A boolean value that asks if it repeats constantly or not).

# If Statements:

[**[manual]**](https://manual-en.yoyogames.com/#t=GameMaker_Language%2FGML_Overview%2FLanguage_Features%2FIf_Else_and_Conditional_Operators.htm&rhsearch=statement&rhhlterm=statement%20statements%20statement1%20statement2)[**[video]**](https://www.youtube.com/watch?v=3ys8IIOi4V4)

A *“If statement”* is an alternative method to the *[“switch statement.”]*

*“If statements”* are a way of checking a variable’s value. Example:

“if variable = 1 {

//code is run here

}”

Be sure to have both ends of the *“{}”* brackets to run the if statement correctly.

# 

# Switch Statements:

[**[manual]**](https://manual-en.yoyogames.com/#t=GameMaker_Language%2FGML_Overview%2FLanguage_Features%2Fswitch.htm&rhsearch=switch%20statement&rhhlterm=switch%20switches%20statements%20statement%20statement1%20statement2)[**[video]**](https://youtu.be/40eMcfPOTVo)

A *“switch statement”* is an alternative method to the *“[If statement.]”*

*“Switch statements”* are a way of checking a [“*variable’s”*] value. They are more clear looking than if statements, and are better for performance. *“Switch statements”* will only look at the correct value you are asking, and ignore the rest. In comparison, *“If statements”* will still read the rest of the statements, even if they are not true.

Example:

switch(variable\_we\_are\_checking){

case value\_possibilty 0: variable = value; break;

case value\_possibilty 1: variable = different\_value; break;

}

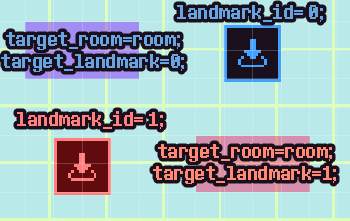
A switch statement checks a [“*variable”*] (can be any variable) and then uses *“cases”* to check the value of that variable. Each case equals a different outcome, they can be numbers, strings, or boolean (true/false).

All cases must be within the *“{}”* brackets that start at the start of the switch. ALL cases **must end with “break;”** otherwise it will not stop reading the code or end in an error.

Overworld

*“Overworld”* refers to the part of the game that happens outside of battle. It is recommended that you understand the basics of [GameMaker Studio 2] first.

# Room Warps



*[IMG ID: 2 room\_warp objects are shown, one in blue and the other red. Their trigger\_warps are also shown in matching colors across from each other. End ID]*

*“Trigger\_warp”* and *“hint\_landmark”* are two *“objects”* that are used together to move the [*“Player”*] from room to room.

To start, we need two rooms. Room A will be the original room the *“Player”* is in. Room B is the destination room.

In Room B, simply place a *“hint\_landmark”* to where you wish the *“Player”* to land. In the [*“creation code”*] in the *“instance,”* type the following code:

*“landmark\_id = #;”*

*“#”* can be replaced with any number, just be sure to remember what number you used.

In Room A, place a *“trigger\_warp.”* Just like with the *“hint\_landmark”* instance, head onto the *“creation code”* to add the following:

“target\_room = room\_name;

target\_landmark = #;”

Set the *“target\_room”* to Room B, then set the *“target\_landmark”* to the same number that was placed in the *“hint\_landmark”* in Room A.

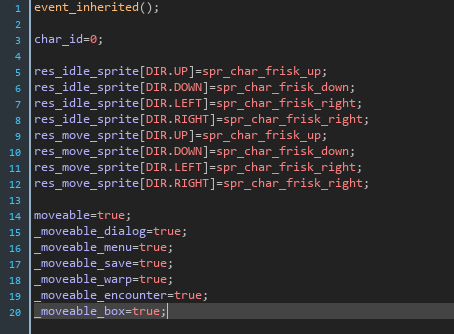
In the above image, note how each corresponding color matches the numbers displayed in each. You can have as many landmarks and warps as you like in any room, just be sure to match them together with the right landmark.

# Player

The *“Player”* is the audience playing the game, and the *“Player Character”* is the ingame character you control. For the default *“Player Character”* in the UNDERTALE Engine, you play as *“Frisk.”* However, you do not need to play as them.

*“char\_player”* is the [*“object”*] we use to control. It is a [*“inherited”*] child *“object”* of *“char.”* We will use *“char”* as the parent for both the player and all NPCs and interactable objects that speak.

In the [*“create”*] event:



*[IMG ID: A screenshot of the char\_player object. Code will be referenced below. End ID]*

The new code displayed in the *“char\_player”* is the following, explanations are coded out:

Create Event:

event\_inherited(); //read [“Object Inheritance”] for explanation

char\_id=0; // This is code used to refer to the “character.” You may have many characters in a scene, but only one specific number per each.

res\_idle\_sprite[DIR.UP]=spr\_char\_frisk\_up;

res\_idle\_sprite[DIR.DOWN]=spr\_char\_frisk\_down;

res\_idle\_sprite[DIR.LEFT]=spr\_char\_frisk\_right;

res\_idle\_sprite[DIR.RIGHT]=spr\_char\_frisk\_right;

//this is used to set up the “idle” [“sprites.”] You may change the name of the sprite to any sprite you wish.

**//IMPORTANT TO KNOW:**

res\_idle\_flip\_x[DIR.XX] indicates whether to horizontally flip the sprite, which is usually used for using the right sprite as the left sprite by simply flipping it, saving tons of resources. **Set it to false to disable this feature.**

res\_move\_sprite[DIR.UP]=spr\_char\_frisk\_up;

res\_move\_sprite[DIR.DOWN]=spr\_char\_frisk\_down;

res\_move\_sprite[DIR.LEFT]=spr\_char\_frisk\_right;

res\_move\_sprite[DIR.RIGHT]=spr\_char\_frisk\_right;

//this is used to set up the “move” [“sprites.”]

moveable=true;

\_moveable\_dialog=true;

\_moveable\_menu=true;

\_moveable\_save=true;

\_moveable\_warp=true;

\_moveable\_encounter=true;

\_moveable\_box=true;

//movable are [“variables”] that are here to set up the “player” to move around. Important for turning off during cutscenes.

Step Event:

This is not necessary to cover. Only thing needed to understand is that this is a [“step”] event. It checks for two things: “player input” and if [“dialogue”] is being spoken. If it is being spoken, the “player” can not move.

Alarm 0 Event:

camera.scale\_x=2;

camera.scale\_y=2;

camera.target=char\_player;

//the [*“camera”*] is an [*“object”*] that is being called. Whenever the *“player object”* is within the room, the scale of the camera will be set to 2. Otherwise, you need to use [*“hint\_halfsize"*] to keep the camera scale at 2.

# NPCs

Similarly to [*“char\_player,”*] NPCS are a child (see [*“object\_inherited”*] for more information) of the [*“char”*] [*“object.”*] Because they are so similar, the following will be copy/pasted from the *“char\_player”* object’s create event and only slightly modified.

Create Event:

event\_inherited(); //read [“Object Inheritance”] for explanation

char\_id=0; // This is code used to refer to the “character.” You may have many characters in a scene, but only one specific number per each.

res\_idle\_sprite[DIR.UP]=spr\_char\_alphys\_up;;

res\_idle\_sprite[DIR.DOWN]=spr\_char\_alphys\_down;

res\_idle\_sprite[DIR.LEFT]=spr\_char\_alphys\_right;

res\_idle\_sprite[DIR.RIGHT]=spr\_char\_alphys\_right;

//this is used to set up the “idle” [“sprites.”] You may change the name of the sprite to any sprite you wish.

**//IMPORTANT TO KNOW:**

res\_idle\_flip\_x[DIR.XX] indicates whether to horizontally flip the sprite, which is usually used for using the right sprite as the left sprite by simply flipping it, saving tons of resources. **Set it to false to disable this feature.**

res\_move\_sprite[DIR.UP]=spr\_char\_alphys\_up;

res\_move\_sprite[DIR.DOWN]=spr\_char\_alphys\_down;

res\_move\_sprite[DIR.LEFT]=spr\_char\_alphys\_right;

res\_move\_sprite[DIR.RIGHT]=spr\_char\_alphys\_right;

//this is used to set up the “move” [“sprites.”]

res\_talk\_sprite[DIR.UP]=spr\_char\_alphys\_up;

res\_talk\_sprite[DIR.DOWN]=spr\_char\_alphys\_down;

res\_talk\_sprite[DIR.LEFT]=spr\_char\_alphys\_right;

res\_talk\_sprite[DIR.RIGHT]=spr\_char\_alphys\_right;

//this is used to set up the “move” [“sprites.”]

moveable=true;

\_moveable\_dialog=true;

\_moveable\_menu=true;

\_moveable\_save=true;

\_moveable\_warp=true;

\_moveable\_encounter=true;

\_moveable\_box=true;

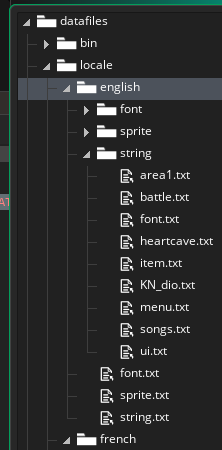
dir\_locked=true;

//movable are [*“variables”*] that are here to set up the *“player”* to move around. Important for turning off during cutscenes.

//direction locked is used specifically for NPCs, in case it’s a still object that can’t move around like the *“char\_sign”* object.

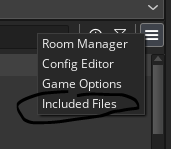
# Multi-Language Support

Multi-language is supported within TML’s engine. Looking into the project’s **datafiles/local/** you will find an **English folder**.



*[IMG ID: A list of folders containing the Multi-language system. The string text files are inside the string folder, which is in the language folder, inside the local folder. End ID]*

You can find this either by going to the project folder or by clicking on the included files from the asset manager tab.

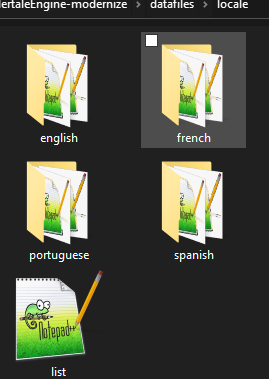


*[IMG ID: A screenshot of the asset manager’s menu, where “included files” is circled. End ID]*

Create a new folder with the title being the language you wish to write for. From there, copy the files in the English folder and rewrite them in the new language to be translated.

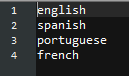
Add additional files for new dialogue following the same format. Keep things organized, such as keeping NPCs in their own document while battle narration can be on its own.

You can do this for both strings and sprites, just have them in their correct folder.



*[IMG ID: A screenshot of several folders, each named for a language. There is one text file named “list.” End ID]*

To add a new language for the engine to read, add it to the list file, found alongside the language folders.



*[IMG ID: A screenshot of the list file, where several languages are listed. Each list entry is named in the next paragraph. End ID]*

**The order of languages is important**. They’ll start at 0 (as many code lists start at 0). For me, English = 0, Spanish = 1. Portuguese = 2, French = 4.

To change which language is being used, create a settings [object with a menu.]

With this menu, change the following functions to match the number on the list.

Lang\_LoadString(0);

Lang\_LoadSprite(0);

Lang\_LoadFont(0);

^ will display English, as English = 0 on my list.

Now that we know how to change the language, let’s add them to our game.

Open the object that will be translated.

Instead of:

text = “Text I want to change.”

use this instead:

Lang\_GetString(“stringkey.stringID")

Lang\_GetString("area.example\_narration\_line1")

*(Same logic with Sprites and Fonts, just with their own Lang\_Get function.)*

# Dialogue

*(Written by Hatty. Moved to this documentation without any edits. The original* [*document by Hatty*](https://docs.google.com/document/d/1eIDB8LFYsK-j0JCeP2gmYXSIcglY_nb-X9gM06WF3ZY/edit?usp=sharing) *can still be viewed. The Dialogue document was originally researched by Hatty with the help of Sam & IIias)*

**text="\* Hello world!";**

**&** - splits text to the next line.

e.g. text="\* Hello world! \* Hello world! \* Hello world!"; will output:

\* Hello world! \* Hello world! \* Hello world! (This will go off screen.)

e.g. text="\* Hello world! &\* Hello world! &\* Hello world!"; will output...

\* Hello world!

\* Hello world!

\* Hello world!

**{choice X )`}** - adds interactive choices for the player to select!

e.g. text="\* Hello world!&& {choice 0}choice 1 {choice 1}choice 2{choice 'TMP'}{pause}”;

IMPORTANT TIP: {choice 'TMP'} must be added near the end of the text for the choice system to work. Think of it like the engine of a car!

See tutorial

**{clear}** - clears the text box. When combined with {pause}, enables additional dialogue to be printed when the player presses a key!

e.g. text="\* Hello world! {pause}{clear}\* How are you today?"; will first print “\* Hello world!”, then the player presses a key, clearing the text box and printing “\* How are you today?” prints.

**{color `(color name here)`}** - colors the whole text your color of choice, with a slight gradient

e.g. text="{color `yellow`}\* Hello world!"; will turn "Hello world!" a yellow with a gradient!

Note: color and color\_text will not work with " or ', you MUST use `!

**{color\_text `(color name here)`}** - colors the whole text your color of choice.

e.g. text="{color\_text `yellow`}\* Hello world!"; will turn "Hello world!" a solid yellow!

IMPORTANT TIP: If you don't use {color\_text 'white'}, most white text (esp. dialogue) will have a dark blue shadow, which is only found in Deltarune.

To get that authentic Undertale white text (with no shadows), use {color\_text 'white'} in all your regular text!

**{effect X}** - adds a cool effect to the text.

X=-1, no effect

X=0, shakey text!

X=1, wavy text (Napstablook style)

**{face X}** - adds a face sprite to the text. X is the face id of the face sprite as defined in

Objects>Text>text\_typer>User Event 5 - Group & Macro.

See tutorial by Hatty on the UNDERTALE Engine Discord server, "Adding faces/talksprites to NPCs"!

**{pause}** - Pauses text or dialogue until the player presses a key.

e.g. text="\* Hello {pause} world!"; Only “hello” will be printed when first interacting with the NPC, pressing any key will print “world!” afterward!

Different from {clear} because the text box is not cleared.

Essential if {choice} is used, to pause the progression of text until the player makes a choice!

**{sleep X}** - pauses the text typing by X amount of steps.

e.g. text="\* Hello {sleep 10} world!"; will delay the typing of "world" by 10 steps.

**{speed X}** - speeds up or slows down text.

-10 -4 -3 -2 -1 0 1 2 3 4 10

<---------------|--------------->

The larger the number (positive), the slower it goes

e.g. text="{speed 10}\* Hello world!"; will type very slowly!

**{voice X}** - adds a voice blip that plays as text is typed. X is the voice id of the voice file as defined in

Objects>Text>text\_typer>User Event 5 - Group & Macro.

# Adding Dialogue to your Room

Before we start, we recommend adding the Dialogue Array Add-on by EternalShine. (This will later be updated).

**THIS IS OPTIONAL.** Using arrays for dialogue makes things easier, but you can skip this step if you’d like.

It can be found on the Discord or can be added manually. Add this function to a new script named “Scr\_Dialog\_Array''. We will be making a function called Dialog\_Add\_Array:

///@arg text\_arr  
///@arg prefix

function Dialog\_Add\_Array() {

var prefix = ""  
 if (argument\_count > 1) prefix = argument[1];  
var text\_arr = argument[0];

var text\_full = prefix;

for (var i = 0; i < array\_length\_1d(text\_arr); i++) text\_full += text\_arr[i];

Dialog\_Add(string\_replace\_all(text\_full, "{end\_ar}", "{pause}{clear}"));

return true;

}

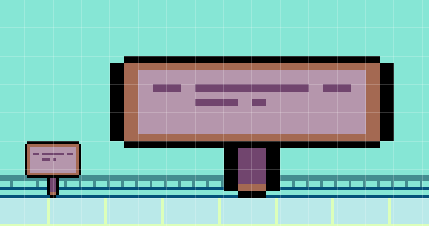
When using this function, you can now add dialogue as an **array** and have **multiple lines added at once.** An **array** is a **list** of things. In this case, a list of (*“strings.”*)

Adding Dialogue to your scenes:

There are several ways you can go about adding dialogue, this is my preferred method.

### 

### The Sign Method:

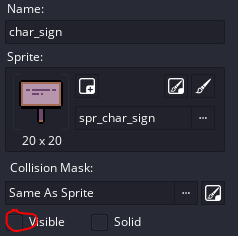


*[IMG ID: Two sign instances are placed in a room. One is normally sized, while the other is stretched into a large instance, scaled to 5 x 2.5]*

One way to save making a new object for every interaction, is to add a new sign to the room.

The sign will be what we are interacting with to get the dialogue.

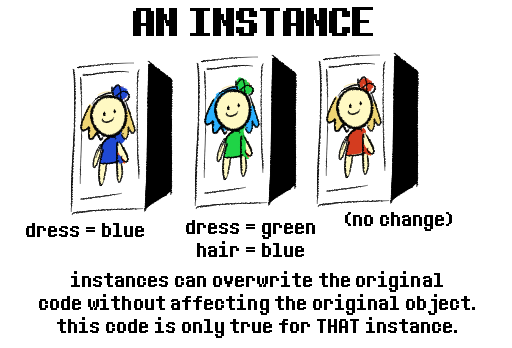
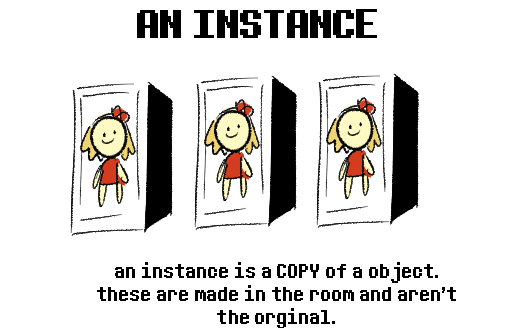
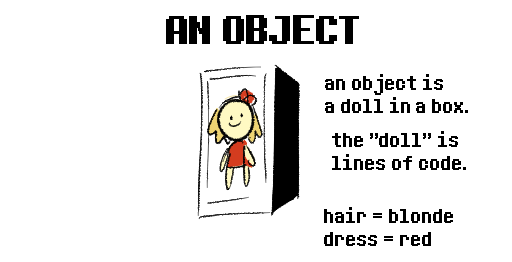
We will be making these signs and they will be set to **invisible,** so you can have the same object used over and over to interact with instead of making a custom object each time. To make the object invisible, simply uncheck the box labeled “visible” on the sign object.



[img id: the sign object is shown to have it’s Visible box unchecked. Id end.]

You can place an invisible sign over a [*“sprite”*] (placed into the room on top the **[“asset layer”]** below).The sprite doesn’t contain code, it’s just a static image. The object is what we interact with that contains code.

[*“Objects”*]are how gamemaker store code. An [*“instance”*] is a copy of an object. You can have multiple instances of the same object in the room. Each instance can be altered without affecting the other instances nor the original object.



Inside the sign [*“object”*], go to the [*“create event”*]. You will see a [*“variable”*] called “text” that is equal to “”. This will be a blank **string,** which can hold text that can later be drawn.

Next go to the [*“user event 0”*], the event that is run when the player interacts with the object. You will then see two [*“functions”*]:

“Dialog\_Add(text)”

“Dialog\_Start()”

Dialog\_Add() will take the variable “text” and will store it in a list that will be run with Dialog\_Start.

If you use Dialog\_Add\_Array, you can run an **array** instead of a single string.

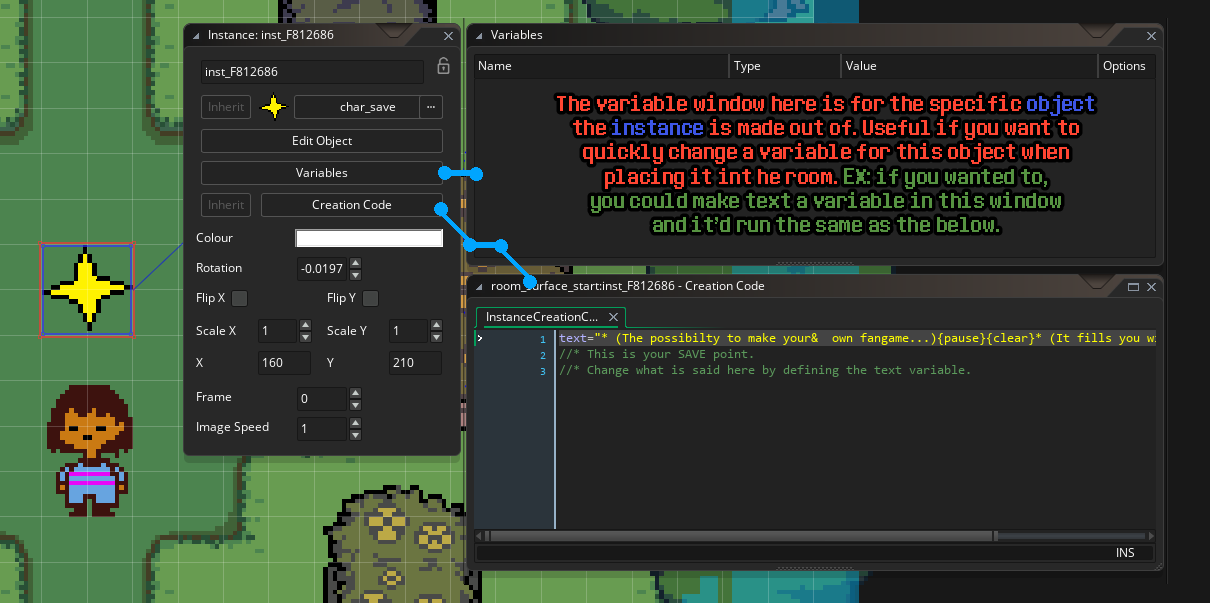
| Dialog\_Add() | text = “\* I am a single string of text that is stored in a string.”  Dialog\_Add(text)  Dialog\_Start(); |
| --- | --- |
| Dialog\_Add\_Array() | text = [  “\* I am line 1{pause}{clear}”,  “\* I am line 2.& I will play after Line 1 runs.”  ]  Dialog\_Add\_Array(text)  Dialog\_Start(); |

The sign, depending on which Dialog\_Add function you use, will take that text variable and run it with Dialog\_Start();

In summary:

Sign is an object. When created, it makes a variable [text = “”;]. Then when the player interacts with the sign instance, it runs the event user 0 code that uses [Dialog\_Add(text)] and [Dialog\_Start()]

To make each instance of the sign say something, go to its [*“creation code”*].



[IMG ID: Screenshot Of Frisk standing by a SAVE point instance. Its instance window is opened, showing both the variable and creation code windows attached. End ID]

(^ Above image was taken from the page **[Instances].)**

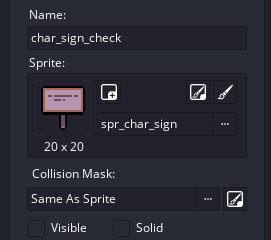
In the sign’s creation code, you can then update the [text = “”] variable and change it to a new line of text. [text = “\* I am a new line of text!”]

Summary in order of what to do:

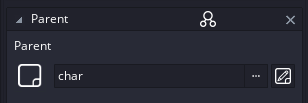
1. Uncheck “visible” on the sign object
2. Place a sign into a room.
3. Put the sign on top of a sprite on the asset layer.
4. Then, in the sign’s creation code, write [text = “\* New line of text.”]

### Sign Multiple Check Method:

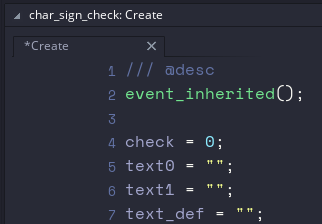
Now, what if you want to have the text update upon being interacted with multiple times?

You don’t need to make a new object for each time you want to update the text. We are going to do something slightly more advanced with the Sign Method.

1. Make a blank new object
2. Name it “char\_sign\_check”
3. Add the sprite image “spr\_char\_sign”
4. Uncheck the “visible” box



1. Make the parent of this object be “char”

What we’ve done is essentially remake the sign object. But now, it’s independent of the sign object and can be changed without affecting the original.

First, we’re going to add some new [*“variables”*]. We’re adding multiple texts. **Add as many as you’d think you’ll ever need.** We’re going for the **max** number of interactions possible. The example on the left uses **3 interactions total,** counting from 0 to 2. This means the labels are: 0 - 1 - default.

0 will be the first, 1 will be the second, and default will be whatever you get from then on out.

All these new variables will be set to [“”] blank. This means if run, you’d get an empty text box.

We do this because we are going to later define the text variables in the [*“creation code”*] like with the sign. As said before, this uses the same logic as the original sign method. We’re just accounting for multiple lines of text.

Now in the [*“user event 0”*] we’re going to change how we use the Dialog\_Add() [*“function”*].

| if check = 0 {  if text0 != "" {  Dialog\_Add(text0);  }  }else if check = 1{  if text1 != "" {  Dialog\_Add(text1);  }  }else{  if text\_def != "" {  Dialog\_Add(text\_def);  }  }  Dialog\_Start();  check ++; |
| --- |

We use an [*“if statement”*] to check what the variable “check” equals to. Remember, we made [check = 0;] in the [*“create event”*]. This means when check is being looked for in the [if check = 0], it will equal zero and therefore run.

Inside the if check = 0{}, there is another if statement. This one is [if text0 != “”{}]

The ! means “does not”. This is saying: if text0 does not equal “” (aka blank) then perform what’s in the following {} (curly brackets.)

Because we set **[check] to [= 0]** the if statement of **[if check = 0] is run**. It then runs a new if statement, to see if **[text0 = “”] is blank**. If you **define text0 to equal something in the instance creation code**, then this will **run**. Otherwise it’s not and it continues to the next if statement.

If the text0 is not “” (blank) that means it contains a filled out string. EX: [text = “\* I am the first dialogue interaction!”] Dialog\_Add(text0) is then run to be run.

**Note**: You can make text0 to be an array text0=[] if you want to use Dialog\_Add\_Array();

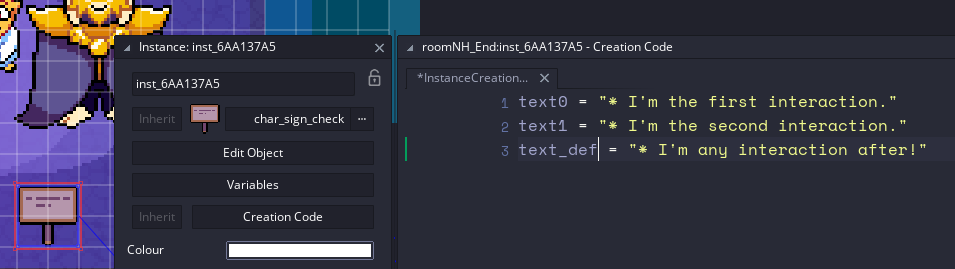
| Dialog\_Add(text0) | (instance creation code)  text0 = “\* I am the first dialogue interaction!” |
| --- | --- |
| Dialog\_Add\_Array(text0) | (instance creation code)  text0 = [  “\* I am the first dialogue interaction!{pause}{clear}”,  “\* I am a second line of dialogue within the first interaction!” ] |

After the Dialog\_Add(); (*or if you’re using Dialog\_Add\_Array();*) are defined using the if statements, the line [**check++**] is used. The **++** means **check += 1;** It adds one to the total of [check].

This means every time you interact with an instance of char\_sign\_array, that instance will increase the check variable. When the if statement is run over the check variable, it will instead run the correct value of text. **It is important to update check ++ *after* the if statement.** This is because the check variable starts at 0, and we will need [if check = 0 {}] to run first. If we update the check to add to itself first thing, then it’ll equal 1 and thus skip over the if 0 statement.

[Dialog\_Start();] of course runs the dialogue depending on what’s been added through the Dialog\_Add(); / Dialog\_Add\_Array() functions;

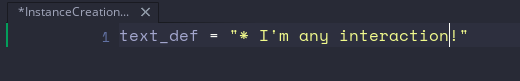
Now, in the creation code:

****

This sign will now update its checked dialogue.



If we only have text0 and text\_def, it’ll still run both. text1 is completely ignored.



^ would also work, but at that point just use a [char\_sign].

### Multiple Check (SWITCH VERSION):

If you’re familiar with a [*“switch statement”*] you can use this for the [*“user event 0”*] instead of [*“if statements”*] within the Sign Multiple Check Method.



| switch(check){  case 0:  if text0 != "" {Dialog\_Add(text0);}  break  case 1:  if text1 != "" {Dialog\_Add(text1);}  break  default:  if text\_def != "" {Dialog\_Add(text\_def);}  break  }  Dialog\_Start();  check ++; |
| --- |

Which will run the same, but is easier to look at and understand.

## TO BE WRITTEN AND FIXED UP LMAO

With the added use of [*“Flags”*] you can have the text change permanently.

text = [  
 text = “Who’s dog is this?”

]  
dog = Flag\_Get(FLAG\_TYPE.STATIC,FLAG\_STATIC.MET\_DOG)  
if dog = true {  
 text = [  
 "\* This is ",  
 Player\_GetName(),  
 “‘s dog.”)  
 ]  
}

The above checks the flag “MET\_DOG”, and if set to true, performs a new sentence.

You can use functions inside the text variable to add names. The above reads “\* This is (Player name)’s dog.”

To use the sign, add it to the room.

But what if you don’t want to use a literal sign but rather want to interact with something else?

Well, you have three options.

1. Create a new object. Inherit the sign, but change the name and sprite to whatever you want to interact with.
2. Use the sign *(you can also stretch it out)* but change the instance color to transparent. *(Alternatively, just add “visible = false” in the* [*“creation code”*]*.)* In an asset layer, place the sprite over the invisible sign.
3. Use the sign object, but add a “sprite = spr\_char\_sign” variable in the create event. For new interactables, change this variable to equal whatever the new sprite’s name is.

By adding a sign, you can add the dialogue in the text variable in the creation code of the [*“instance”*] inside the [*“room”*].

text = “”

To define the text variable.

For [*“Multi language”*] support, use the [*“function”*] “Lang\_GetString()” instead of using a blank string (“”).

Example:

Lang\_GetString(“textID.key”)

Lang\_GetString(“narration.ExampleLine1”)

# Dialogue Portraits

Face Portraits can be used either within the dialogue box or by attaching the face onto a body sprite (as seen with [Shops.])

To create a face [*“object”*], create a blank new object and make it a [*“child”*] of the object “face.” Name it *face\_char\_name*, replacing *char\_name* with the character’s actual name.

To add in new faces use the following code in the “text\_typer” object, User Event 5: Group and Marco.

At the end of the event, add:

\_group\_face[0]=face\_char\_name;

Now, back at your *face\_char\_name* object. Add:

idle\_sprite[0]=spr\_face;

idle\_image[0]=0;

idle\_speed[0]=0;

talk\_sprite[0]=spr\_face;

talk\_image[0]=0;

talk\_speed[0]=0;

**Idle\_sprite**

This is default sprite that shows when not talking

**idle\_image**

This is the frame number of that sprite

**idle\_speed**

This is the speed of the default sprite, good for if the sprite has a looping action when not speaking. (Think Toriel blinking as an example.)

Leave at 0 otherwise.

**talk\_sprite**

This is the talking sprite. Use a open mouth sprite to give the illusion of speaking

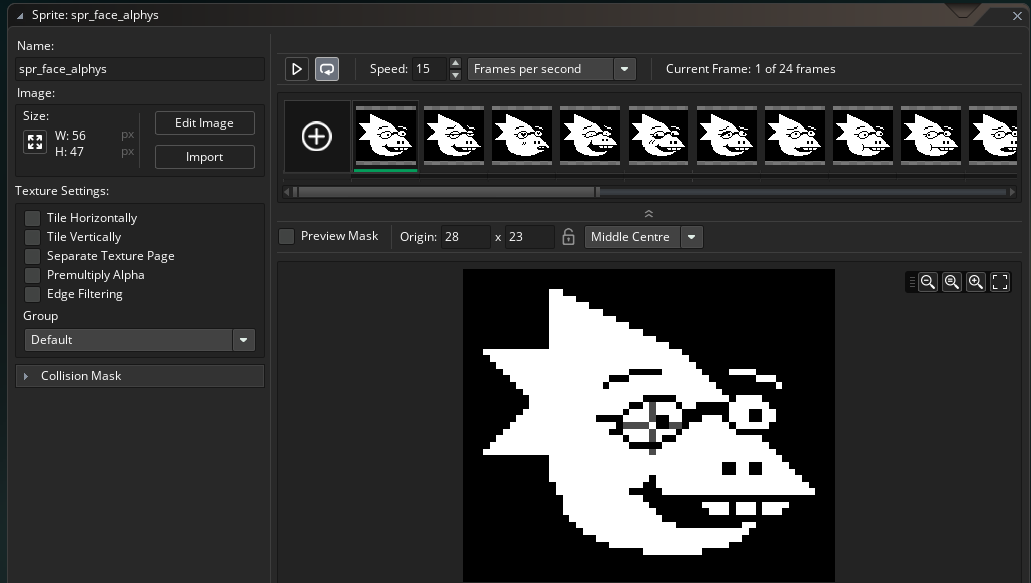
**talk\_image**

This is the frame of the talking sprite we're using

**talk\_speed**

Speed of the talking frame

Example:



[*IMG ID: Screenshot of a sprite window, containing Alphys’ facial portraits from undertale.*]

Add the [*“sprite”*] using the strip method (mentioned in *Sprites*) and set the origin to “middle center.”

For characters with moving eyes, use separate sprites for each expression and mouth movement.



[*IMG ID: Screenshot of a Toriel’s happy sprites as she talks.*]

//4 Talk Joy

idle\_sprite[4]=spr\_face\_toriel\_talk\_joyed;

idle\_image[4]=0;

idle\_speed[4]=0;

talk\_sprite[4]=spr\_face\_toriel\_talk\_joyed;

talk\_image[4]=1;

talk\_speed[4]=0.5;

# Hints

# There are several Hint [*objects*], each with their own purpose.

* hint\_bgm
  + add this to your room to set up the music to play.
  + use “bgm = AudioTrackName;” within the creation code.
  + For battles, look at [*Enemy Set-up*] to set the music when the battle starts.
  + For more information on audio, read [*On Music and Sounds*]
* hint\_landmark
  + Used alongside the trigger\_warp to move the player room to room.
  + For more information on warping, read [*Room Warps*]
* hint\_border
  + Borders, when enabled in the *“world”* [*“object”*] display a border to make the game window display wider with the border image attached. This is handy if you want to replace the black borders with an image and to make the game’s resolution better match for video.
* hint\_half\_size
  + Place hint\_half\_size to adjust your camera’s sizing to two. This will automatically happen when the player is in the room, but this is handy for if you have a room that needs to be scaled up but the player isn’t present.

# 

# Object block (Collisions)

# A block [*“object”*] contains collisions. Collisions are where objects interact with different objects. In the case of the *Block* and *Block Corner,* it prevents the player from passing through.

To turn off the collision for a object’s [*“instance”*], write the code:

block\_enabled=false;

### 

# Cutscenes

There’s a couple of ways to handle cutscenes. Today I will be describing my personal method.

I use a modified version of [Juju Adam’s Coroutines.](https://github.com/JujuAdams/Coroutines) Coroutines have code placed on certain conditions one after another. You can have something happen after a certain amount of milliseconds or under if then statements.

The Coroutines had to be modified to work with the TML engine. The [*“Anim\_Create”*] [*“function”]* also uses the same [*“macros”*] names which then create conflict with each other, crashing the game. The edit can be found on the discord, under the Community-Resources channel.

The JuJu Adam’s Coroutines has its own documentation so I won’t explain it here. The edits only change the macros.

RETURN -> C\_RETURN

REPEAT -> C\_REPEAT

IN -> C\_IN

DELAY -> C\_DELAY

*(C\_ stands for Coroutine\_)*

# Camera

The camera is how we view the game. UNDERTALE uses a [*“tileset”*] of 20x20 pixels. When the game runs and the player is present the pixels are scaled to a doubled size (2x2). If you do not have the player present and still wish to have the scaled up effect, use [*“hint\_half\_size”*] to do so.

Battles run at (1x1) and thus don’t need the camera adjusted.

To manipulate the camera for cutscenes:

* target=noone;
  + (if the player is present, the target will be set to the player. To be able to move the camera, you need to temporarily turn this to “noone” instead.)
* Use [*“Anim\_Create”*] to modify these values to move the camera:
  + x=0;
  + y=0;

# Sequence Editor

[[video](https://youtu.be/-k-tE44Qocg)]

The Sequence Editor is a Gamemaker Studio 2.3 only feature. Before upgrading to 2.3, read up on our page regarding 2.3.

The Sequence Editor is used to manipulate sprites in a set movement in a video-style editor. (Think Adobe After Effects or Premiere, most video software will have a similar timeline editor feature.)

# Anim\_Create

**(NOTE: there is an extension of the Anim\_Create functions in the Community Resources channel in the Undertale Engine Server)**

Anim\_Create is a [*“function”*] that is used to change values within a certain amount of time. Most usefully, it is used to move [*“objects”*] to animate. However, it can be used for any value and works similarly to [[*“lerp and sin”*](https://youtu.be/WxXrgUcj5-Y)] functions.

**Anim\_Create(id,value,ANIM\_TWEEN.---,ANIM\_EASE.---,start,change,duration);**

Example: Anim\_Create(id,x,ANIM\_TWEEN.LINEAR,ANIM\_EASE.IN,0,+80,60);

**Anim\_create:**

The [*“function”*] you call

**():**

Parameters (settings you define)

**id:**

Whatever object you're manipulating, can refer to itself or a different object.

Example: self

**"value":**

The value that you are animating, it can be any [*“variable”*].

Example: “x”

The x value is where the object is placed on the x-axis of the screen.

**ANIM\_TWEEN.---- and ANIM\_EASE.---:**

The tween and ease. Tweens and Eases define how an object moves over the course of time. Both use [*“ENUMERATORS”*] to list the different kinds of tweens/eases. You can middle click each to see the ENUM in the example below, and choose from the list which kind of tween you’d like to use. Read up on tweens with [this link.](https://www.redgiant.com/user-guide/universe/animation-tween-examples/) Alternatively, try out different combinations of the tween and eases to find which one works for the situation needed.

Example: ANIM\_TWEEN.LINEAR and ANIM\_EASE.IN.

**Start:**

The start value can be any number or a variable that contains a number.

Example: 0

It starts at the “x” value of 0.

**Change:**

The amount of change for the value. Can be a positive or negative number or variable that contains a number.

Example: +80

The start would be at value 0, and end at value 80. This means the object will move 80 pixels to the right.

**Duration:**

How many frames it takes for the object to move.

Example: 60

By default, the engine runs at 60 frames per second. It will take 1 second for the object to reach the changed value.

# FLAGS

FLAGS are variables that are stored within an [*ENUM*] that are set to have different behaviors when Saved into the game’s file. Put simply, they are special variables that are saved either by manually saving them or when the player saves.

By default, whatever happens in a room will repeat if you re-enter the room. You pick up a slice of cake, that cake will be there when you return. To prevent this from repeating, use a FLAG.

* enum FLAG\_TYPE{
* STATIC,
* DYNAMIC,
* TEMP,
* INFO,
* SETTINGS,
* DEMO
* };

Are the available flags you can use.

For our purposes, we will only cover STATIC and DYNAMIC.

STATIC FLAGS are remembered when the game saves or by manually placing them. Think of the cake example. It won’t be there when you return, but will be there if you reset.

In contrast, a DYNAMIC FLAG will be remembered even if the player resets. Think of the player killing in a neutral route, only to reset and have Flowey comment on your actions.

To store the value of a FLAG in a [*“variable”*], simply have a variable = the [*“function”*] “Flag\_Get();”

var xp=Flag\_Get(FLAG\_TYPE.STATIC,FLAG\_STATIC.EXP);

This captures the EXP of the player. The parameters of this function are: **(type,slot,default\*)**

**type:** either STATIC or DYNAMIC. To refer to the FLAG within the enum, you have to phrase it as FLAG\_TYPE.STATIC or FLAG\_TYPE.DYNAMIC. “FLAG\_TYPE” is the ENUM, and the STATIC or DYNAMIC are the values.

**slot:** refer from the chosen STATIC or DYNAMIC from their own ENUM. You can middle click over the type or search for the “MARCO\_FLAG” script. You will find several ENUMs, look for the “STATIC” and “DYNAMIC” ENUMs.

* #region dynamic
* enum FLAG\_DYNAMIC{
* };
* #endregion

Create your own new FLAG (always use CAPS for writing within ENUMs) within the ENUM. In this example, we’ll call it “NEW.”

var NewFlag = Flag\_Get(FLAG\_TYPE.DYNAMIC,FLAG\_DYNAMIC.**NEW**);

You can now treat NewFlag as a regular variable.

To change the value of the Flag, use the function *“Flag\_Set.”*

Flag\_Set(FLAG\_TYPE.STATIC,FLAG\_STATIC.CAKE,1);

The parameters are the following: (type, slot, value)

**type and slot:** refer to the above.

**value:** The new value given to the FLAG. You can add, subtract, or set the value to whatever you like.

There is also an additional “PLOT” flag within the script “Marco\_Plot.”

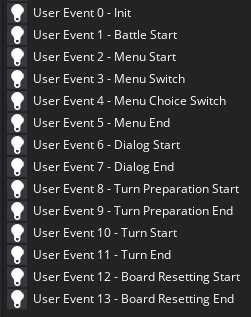
* enum PLOT{
* START,  
   MORE\_PLOT\_POINTS,  
   MORE\_HERE
* };

The PLOT Flag will be equal to one value at a time, so this is great for setting up unique events that happen within your plot that happen once. An example would be to keep track of where Toriel is in a room. She demonstrates a puzzle, then leaves the room. Change the flag from “PUZZLEROOM” to “AFTERPUZZLEROOM,” when the char\_toriel\_puzzle\_room is set to only appear in that room.

Battle

# Battle Object Custom

First, you must make a new [*“object.”*] Make it a child of the parent *“battle\_enemy”* object. (See [*“Object inheritance”*] for more details.) Name it *“battle\_enemy\_custom”* with custom being the name of your enemy.

The events of the object work within a loop. Each event works as a state of battle.

*[IMG ID: User Events 0-13, all of which are covered below]*

* **User Event 0 - Init**
  + Init is short for Initialize. This is where you set up the variables, functions, and stats of the enemy.
* **User Event 1 - Battle Start**
  + Self explanatory.
* **User Event 2 - Menu Start**
  + This is when the player can now use the Menu choice buttons.
* **User Event 3 - Menu Switch**
  + This is run when you click the buttons.
* **User Event 4 - Menu Choice Switch**
  + Runs when you switch through the choices.
* **User Event 5 - Menu End**
  + Once you have chosen your button, you can check the player input for what button they pressed. This is where you would check if the enemy is dead or damaged, spared, or a specific ACT choice.
* **User Event 6 - Dialog Start**
  + For if the Enemy speaks before the attack.
* **User Event 7- Dialog End**
  + If you want something to happen after the Enemy speaks.
* **User Event 8 - Turn Preparation Start**
  + The Enemy’s attack is set up here.
* **User Event 9 - Turn Preparation End**
  + The Enemy’s attack is run here.
* **User Event 10 - Turn Start**
  + The enemy’s turn starts.
* **User Event 11 - Turn End**
  + Once the enemy’s turn is over, be it through code, the bullets are gone or the time of the attack is over.
* **User Event 12 - Board Resetting Start**
  + Once the enemy’s attack is over, the board then animates to a dialogue box for narration.
* **User Event 13 - Board Resetting End**
  + This is when that narration is over.

Battles are a loop of these events, simply checking player input and creating variables to be added, subtracted or checked to progress the battle. Check [*“Examples”*] to see this demonstrated within the code.

# Enemy set-up

*(Please download the [Example] project or import from a local package within the discord server.)*

In the *“Encounter\_Custom”* [*“script”*] you will see the following:

Encounter\_Set(0,-1,battle\_enemy,-1,"\* You encountered nothing!",-1);

//Simple Battle

Encounter\_Set(1,-1,battle\_enemy\_simple,-1,"\* You encountered the test& monster!",-1);

//Gaster Blaster

Encounter\_Set(2,-1,battle\_enemy\_gb,-1,"\* Gaster Blaster example.",-1);

*“Encounter\_Set(id,enemy\_0,enemy\_1,enemy\_2,menu\_dialog,bgm\*)”* is the [*“function”*] that handles setting up the enemy.

**id:** place a number to define which enemy you will encounter. You must use this number *once* and not repeat it, as it will refer to only that specific battle.

**enemy\_0 - enemy\_2:** You have up to 3 enemies you may add to the scene. Simply write down your [*“Battle Object Custom’s”*] name to define that *“enemy\_#”* to. If you wish to have one or two enemies, use *“-1”* (see [*“variables”*] for more information) to skip the spacing.

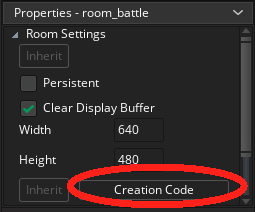
example: *“-1,battle\_enemy\_simple,-1,”* enemy\_0: skipped, enemy\_1: enemy\_simple, enemy\_2: skipped.

**menu\_dialog:** The initial narration upon meeting the enemy.

**bgm:** Short for background music. Simply place the Music audio track’s name.

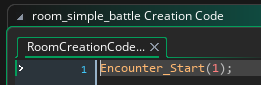
Duplicate one of the battle rooms, it doesn’t matter which one.

Example: room\_battle



*[IMG ID: Screenshot of the Room’s Properties panel. The Creation Code is circled in red.]*

Then head onto the room’s *“creation code.”* (Found at the bottom left of the room.)



*[IMG ID: Screenshot of the room’s creation code, revealing the Encounter\_Start(1); function. ID end.]*

You will see the above. The [*“function”*] *“Encounter\_Start(1);”* is seen. The number here is what the *“Id”* in the *“Encounter\_set();”* function refers to.

You may call this code anywhere in the game, not just the battle room. Having a battle room set up with this function is an easy way of playtesting, as you’re sent directly to the battle upon the game starting. (Provided the battle room is under *“room\_init.”*)

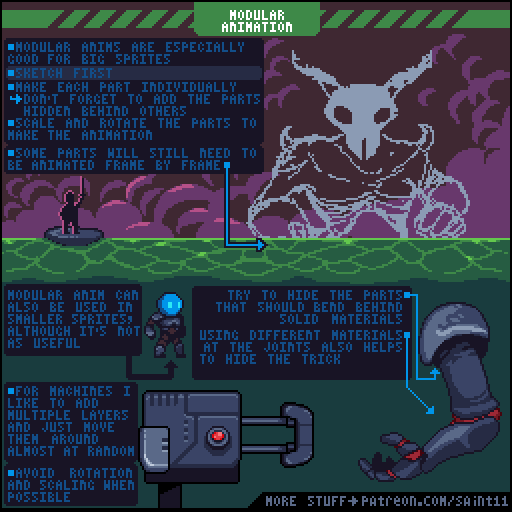
To add a sprite to the enemy, set your new [*“Battle Object Custom’s”*] sprite to be the sprite. However, if you wish for your enemy to have moving body parts separate from the enemy itself, just have the bottom half of your enemy be the sprite. You can then add body parts as objects with the [*“function”*] *“instance\_create();”* and manipulate them within a [*“anim\_create();”*] function. See [*“Enemy Animation”*] for more details.

# 

# Enemy Animation

For animating your enemy, you can either use a single sprite for your enemy or use the technique *“modular animation”* to do so. Modular uses multiple sprites set to move in a path determined by code. In the case of the UNDERTALE Engine, we use [*“Anim Create.”*]

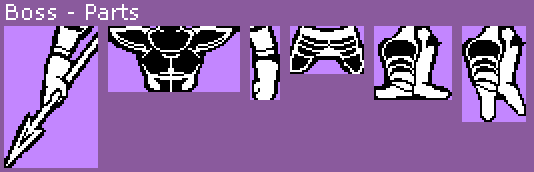
*(****NOTE:*** *there is the added extension of “Anim Create” from Eternal Shine that allows for playback modes like “ping pong” and “repeat” to loop the animation for your convenience.)*



*[IMG ID: An animated infographic from Saint11. It covers the detailed process of Modular Animation, showcasing how to do so for video game sprite art. ID end.]*

([Above image taken from Saint11.](https://saint11.org/blog/pixel-art-tutorials/) Used for educational purposes.)

For example:



*[IMG ID: Multiple body part sprites of Undyne from UNDERTALE. ID end.]*

You would then create an object for each body part. Then, within the [*“battle\_enemy\_custom”*]’s create event or init event, use the [*“function”*] *“instance\_create();”* to add these objects to the scene.

To line them up properly in a layered fashion you’d need to be aware of how you’re placing them on the screen. Keep in mind their origin points and how they’ll relate to the “battle\_enemy\_custom” object.

Have the battle\_enemy\_custom be the feet while the other body parts are the new objects we’re creating with the function “instance\_create();” as mentioned before. Reference the x/y axis of the battle\_enemy\_custom object when placing down the body part objects.

Example:

//legs are battle\_enemy\_custom

//head

//(note: code is spaced out for easier reading)

variable\_head = instance\_create(

battle\_enemy\_custom.x-40,

battle\_enemy\_custom.y-40,

battle\_enemy\_custom\_head);

*(The x and y modifiers will be different, depending on the sprite size. Just playtest and experiment with the values until they line up properly.)*

To make the sprites change within the battle, there’s a couple of things you’ll need to do.

First, in the battle\_enemy object (the [*“parent”*] object of battle\_enemy\_custom) in the create or init event create both a [*“variable”*] and [*“ENUM”*] for the states of the enemy.

* enum BATTLE\_ENEMY\_STATE{
* IDLE,
* HURT,
* DEATH,
* ATTACK,
* };
* state = BATTLE\_ENEMY\_STATE.IDLE

Change the value of the state variable in the different turns depending on the actions of the player. For example, in the code that runs when the enemy is hurt, you’d put state = BATTLE\_ENEMY\_STATE.HURT. Do the same for every state you plan on the sprite changing for.

Then in the draw event of the child, use a [*“switch statement”*] to check which state the enemy is in.

example:

switch(state){

case 0:

sprite\_index = sprite;

obj\_head.sprite\_index = sprite; break;

Anim\_Create(self,"y",ANIM\_TWEEN.CUBIC,ANIM\_EASE.IN\_OUT,y,+20,60,false,ANIM\_DATA.ARG\_0,ANIM\_DATA.ARG\_1,ANIM\_MODE.PINGPONG);\*

***NOTE:*** *the added functionality of ANIM\_MODE is from the Eternal shine add-on for anim\_create. This is found in the community\_resources channel on discord and will be added to the main github soon.*

*If you are not using this, remove the ,false,ANIM\_DATA.ARG\_0,ANIM\_DATA.ARG\_1,ANIM\_MODE.PINGPONG*

*from the list and use checks of the Y value to use multiple anim\_creates to loop the animation.*

case 1: //Whatever state next etc

break;

}

For each case, write what the state means, then use [*“Anim Create”*] in a repeating loop. Repeat for each state.

For facial expressions, use the [*“face\_link”*] variables and [*“face”*] object as the parent for the enemy head object. (documentation for the face link has not been written yet. put simply, have the char\_id = # in the face object and add {char\_link #} in the dialogue to change the portraits.)

# 

# Enemy dialogue

The following code adds a dialogue speech box for the enemy.

# var inst=instance\_create\_depth(x+100,y-150,0,battle\_dialog\_enemy);

inst.text="1";

inst.template=0;

***“var inst”*** (see [*“variables”* for more information] creates a variable that will equal the [*“instance”*] of the [*“object”*] *“battle\_dialog\_enemy.”* Put simply: an instance of the *“battle\_dialog\_enemy”* is created and the variable refers to that instance. We can now manipulate that instance’s variables from inside the *“battle\_enemy”* object.

The ***“x+100”*** and ***“y-150”*** and ***“0”*** refer to the *“x”*,*”y”*, and *“depth”* of where this new instance is placed compared to the *“battle\_enemy\_custom’s”* x and y values. In other words, the speech bubble will be placed relatively close to where the enemy is. (See [*“Enemy Set-Up”*] for more details.)

**“inst.text = “1”;”** the “inst” is the variable we created in the line above. “text” is a variable within the *“battle\_dialog\_enemy”* object. The variable “text” is used the same way as in [*“Dialogue,”*] simply write the speech you’d like the enemy to say in the bubble in the string. (Between the “” characters.) In the above example, the enemy will say *“1.”*

# **“inst.template=0;”** The template refers to the *“battle\_dialog\_enemy”* object’s alarm 0 event. Within it, there is a [*“switch statement”*] that defines the sizing and direction the speech bubble is used. You may create or modify the templates to your desire.Turns/Attacks

Fill in later

# Enemy Turns and Bullets

# Fill in later

# Sans

Fill in later