

Welcome to Godotzilla, an open-source Godzilla: Monster of Monsters framework!

Contents

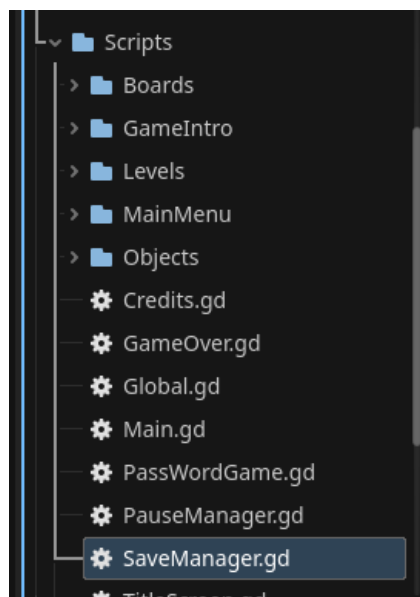
First setup	1
How the framework works: the “Main” scene	2
How the framework works: object components.....	5
How the framework works: state machine.....	7
How to create a new Pass Word Game password	7

First setup

You might be wondering, what should we do when we first setup the framework? I mean, it opens up and runs perfectly.

But here’s what I mean: since there’s a save system built-in and since it uses encryption, it’s better to change the default encryption key so it’s harder for people to mess with the save files (like changing the planet or score).

The encryption key is stored in “Scripts/SaveManager.gd” file:



When you open it up and scroll to the very top, you will see a comment suggesting changing the encryption password:

```
1  extends Node
2
3  const SETTINGS_PATH = "user://settings.cfg"
4  const SAVE_FILE_PATH = "user://save.cfg"
5  const SAVE_FILE_PASS = "Godotzilla" # Change this in your game!!!
6
```

You can change it to whatever you want!

```

1  extends Node
2
3  const SETTINGS_PATH = "user://settings.cfg"
4  const SAVE_FILE_PATH = "user://save.cfg"
5  const SAVE_FILE_PASS = "My awesome game" # Change this in your game!!!

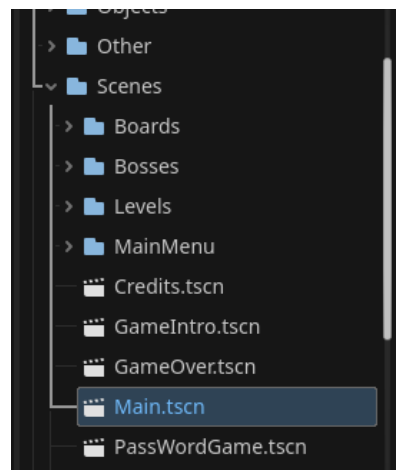
```

But don't panic when you change the password and all existing saves stop working. The existing save file uses an older password, and since it's now different, Godot will not be able to decrypt it.

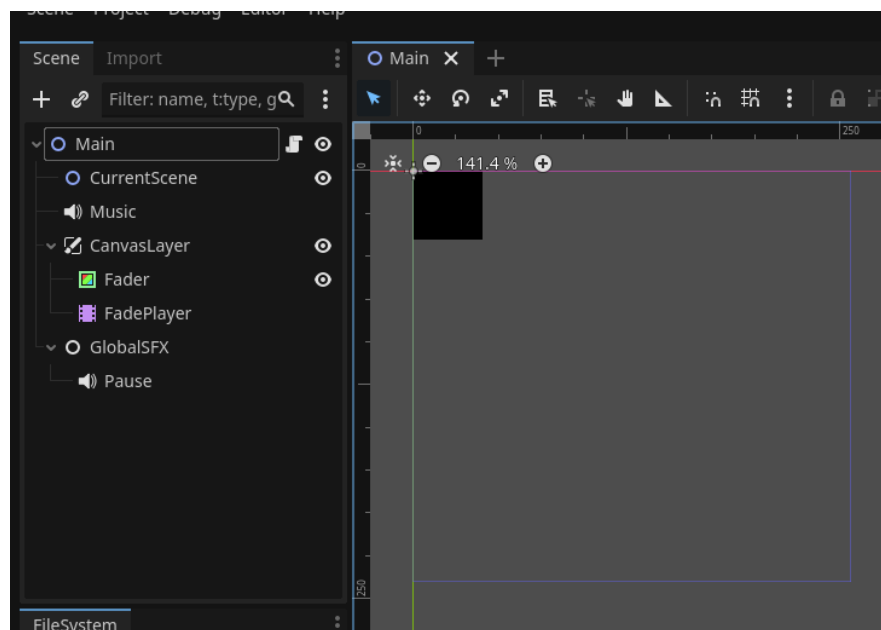
How the framework works: the “Main” scene

This section describes the basics on how the framework works. If you know how it works, you will better understand why this thing was chosen or how to better utilize that another feature, etc.

If you look in the “Scenes” folder you will see that there's a file called “Main.tscn”



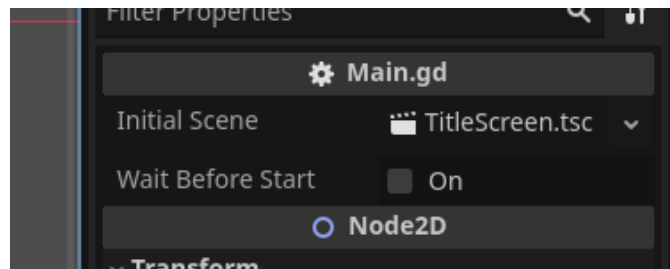
If you open it up you will see this:



It has “Music”, “Fader”, “CurrentScene” nodes. And the whole game relies on music and fades, so how did we make it so that the objects of this scene are then included into every other scene?

We didn’t. This main scene includes other games scenes. Let me show what I mean.

If you open the properties of the root node (“Main”) you will see a property called “Initial Scene”

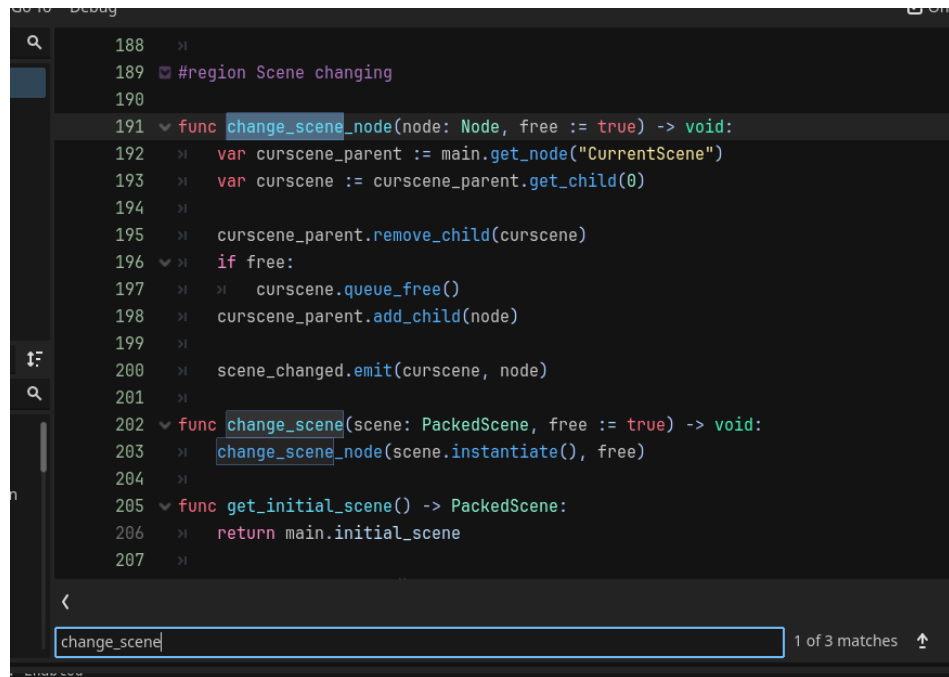


And if open the script tab, you will see the script for this “Main” scene. There’s a “start” function that is called when the main node is ready (“_ready” function) and it instantiates the initial scene that was passed as a property and adds it as a child of “CurrentScene” node.

```
21 >| >| >| >| >| )
22 >| else:
23 >| >| start()
24 >|
25 >| func start() -> void:
26 >| >| $CurrentScene.add_child(initial_scene.instantiate())
27 >| >| Global.widescreen_changed.connect(_on_widescreen_change)
28 >| >| Global.scene_changed.connect(func(_from: Node, _to: Node) -> voi
29 >| >| _on_widescreen_change()
30 >| >| )
```

Okay, but that’s just the initial scene. How do we handle scene transitions?

If you open the Global.gd file again, press Ctrl+F, enter “change_scene” you will be met with this:

A screenshot of the Godot code editor showing a search for the function 'change_scene_node'. The search results show three matches. The first match is at line 191, which is the function definition:

```
191 func change_scene_node(node: Node, free := true) -> void:
192     var curscene_parent := main.get_node("CurrentScene")
193     var curscene := curscene_parent.get_child(0)
194
195     curscene_parent.remove_child(curscene)
196     if free:
197         curscene.queue_free()
198     curscene_parent.add_child(node)
199
200     scene_changed.emit(curscene, node)
201
202 func change_scene(scene: PackedScene, free := true) -> void:
203     change_scene_node(scene.instantiate(), free)
204
205 func get_initial_scene() -> PackedScene:
206     return main.initial_scene
207
```

 The search bar at the bottom contains 'change_scene' and indicates '1 of 3 matches'.

As you can see, the “change_scene_node” gets the “CurrentScene” node in the main scene, gets its child (the initial scene that was instantiated earlier, for example), removes it and adds another node as its child. Basically the same method as the one in the main scene, but this one removes the already existing scene node that is currently being shown on screen, replaces it with another one (that was already instantiated, now it’s a node) and calls it a day.

You might be wondering, why wouldn’t we just use Godot’s built-in system for changing scenes? Like this

```
get_tree().change_scene_to_file("res://Scenes/Levels/TransitionLevel.tscn")
```

The answer is simple: it doesn’t provide enough flexibility for our needs. For example, we would manually need to instantiate our music and fade nodes. And also, Godot’s built-in system only supports changing the scene to a file, not a node.

Huh? Why would we change the scene to a node?

Here’s an example: the game created the level scene; the player is already playing the level. But then they want to pause the game and go to the pause menu. Simple enough, we just change the scene to this pause menu file.

But how would we handle exiting from the pause menu? The level was already cleared from the computer’s memory, and there was no way for us to save it anywhere.

That’s where our system comes in. If you look at the above screenshot with “change_scene_node” function you can see it also takes another parameter, “free”. If it’s false, it won’t remove the previous scene node from the computer’s memory.

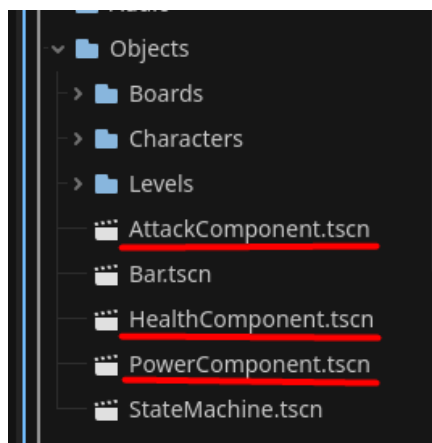
But this parameter should only be used if you intend to save a reference to the current scene node and then get back to it some time later. Otherwise, you can just ignore this parameter.

(Don't worry, we weren't the first ones to introduce a system like this in place of Godot's built-in system. You can find it in other frameworks too, such as "Sonic Worlds Next")

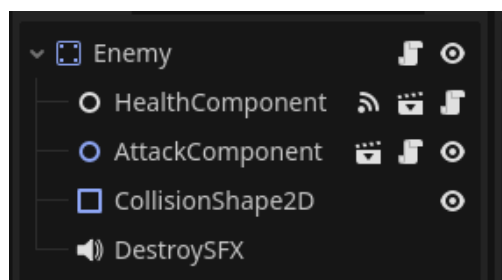
How the framework works: object components

When you create an enemy, you probably expect to every time recode the following parts: health points, attack system, or sometimes the power points for more complex enemies.

Well, you don't have to! That's where reusable object components come in!

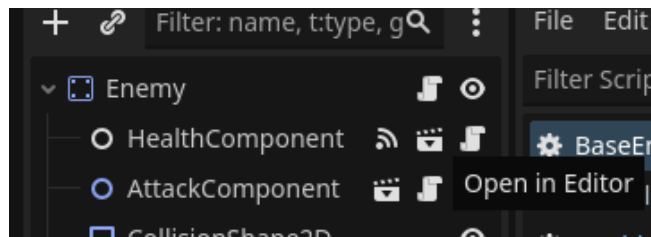


And if you open the base enemy scene ("Objects/Levels/Enemies/BaseEnemy.tscn") you can see it already has health and attack components



So how does it work and why would you want to use that instead of recoding these features every time?

If you open the health component scene (like this:)...



... and look into its script you can see some features that this component provides:

```
## Health points
@export var max_value := 10.0
@export var enemy := false
@export var invincibility_time_seconds := 0.0
## How many health points should be taken each frame
@export var health_speed := 1.0
```

```
signal value_changed(new_value: float)
signal damaged(amount: float, hurt_time: float)
signal dead
signal healed(amount: float)
signal resized(new_amount: float)
```

And here are some other ones: it checks if its parent is hurtable and will not apply damage in this case; it makes the health points gradually go down in case a big damage was applied (for example, Godzilla's heat beam against a breakable mountain in the Wilderness level); and it already handles the invincibility time in case you need it!

```
func damage(amount: float, hurt_time: float = -1) -> void:
    if amount <= 0 or invincible or died \
    or (get_parent().has_method("is_hurttable") and not get_parent().is_hurttable()):
        return
    target_value = clampf(target_value - amount, 0.0, max_value)
    damaged.emit(amount, hurt_time)
    if invincibility_time_seconds > 0.0:
        invincible = true
        await get_tree().create_timer(invincibility_time_seconds, false).timeout
        invincible = false
```

The attack component includes an Area2D node, and if the component in its attacking state and another body that has a health component enters the area2d node, it applies damage to the node and informs the parent that a body was attacked (for example, that's how it's used for the player:)

```
func _on_attack_component_attacked(attacked_body: Node2D, _amount: float) -> void:
    if attacked_body is Enemy:
        add_xp(5)
        Global.add_score(100)
```

You can see examples of how this component is used in “Scripts/Objects/Characters/Attack.gd”.

And the last one, the power component. It handles the power amount, and it provides a check if the current power amount is enough for an object’s attack or any other action.

How the framework works: state machine

When you’re just starting to code a game, your player script might be simple enough to include every action you currently have. But when your game grows, it becomes harder to manage every action if all of them are in one single file.

That’s where state machine comes in. It separates actions into different nodes that are enabled or disabled when the action is changed. You can see some player actions code in “Scripts/Objects/Characters” folder, and in “Scripts/Objects/StateMachine.gd” file you can see how a reusable state machine was made.

How to create a new Pass Word Game password

Here’s a pretty simple tutorial that is too small to be included in a separate file. If you open the “Scripts/PassWordGame.gd” file and scroll up you will see the list of every password that is possible to enter in this mode:

```
21
22  ▾ # Dictionary of all passwords
23    # Passwords should be lowercase and stripped from spaces on the right
24  ▾ var passwords := {
25    >| "test": pw_test,
26    }
27
```

Let’s create a new one.

```
# Passwords should be low
▾ var passwords := {
  >| "test": pw_test,
  >| "g0j1ra": pw_test,
}
```

Remember that not every letter is included in the pass word game alphabet!

Now if you scroll all the way down to the bottom of the script you will find the “pw_test” function:

```

5  ▼ #####
6  # Password actions below #
7  #####
8
9  # Just a test password
10 ▼ func pw_test() -> void:
11   >| get_tree().paused = true
12   >|
13   >| Global.music_fade_out()
14   >| await Global.fade_out(Global.FadeColor.WHITE)
15   >| await get_tree().create_timer(1).timeout
16   >|
17   >| Global.play_music(preload("res://Audio/Soundtrack/PassWordGame.ogg"))
18   >| await Global.fade_in(Global.FadeColor.WHITE)
19   >|
20   >| get_tree().paused = false

```

Let's create a similar one but instead it will just play the credits music:

```

1
2 ▼ func pw_g0j1ra() -> void:
3   >| Global.play_music(preload("res://Audio/Soundtrack/Credits.ogg"))
4

```

Good, now we will reference it in the passwords list

```

▼ var passwords := {
  >| "test": pw_test,
  >| "g0j1ra": pw_g0j1ra,
  >|
}

```

Now you can test the game and verify that it works!

