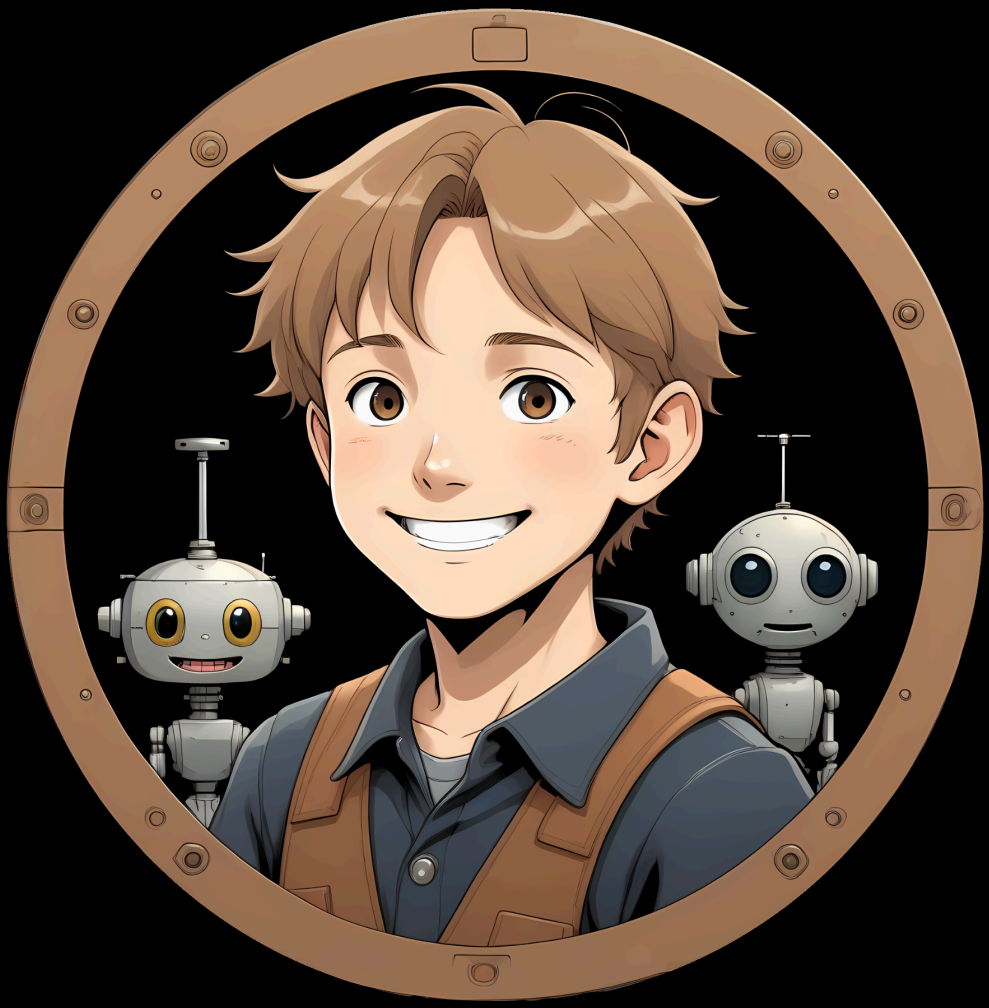
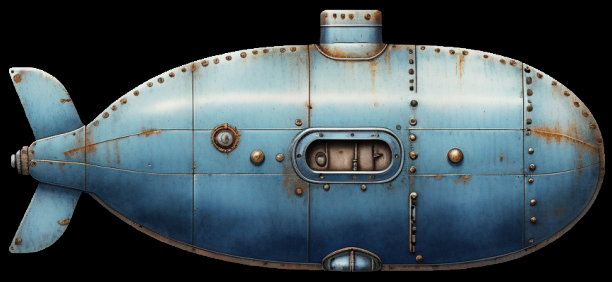


# From Past to Present: Ensuring Saved Game Compatibility

a Godotneers publication

# Contents





# Introduction

When we save games, we want to be able to load them again later. But what if our game changes? How can we make sure that we can still load old save games that were created with an older version of our game?

We're going to explore how changing the mechanics of a game can affect our ability to load saved games that were created before. We'll see which problems can arise and how to approach these problems in our loading code so we can successfully load back older saved games in newer versions of our game.

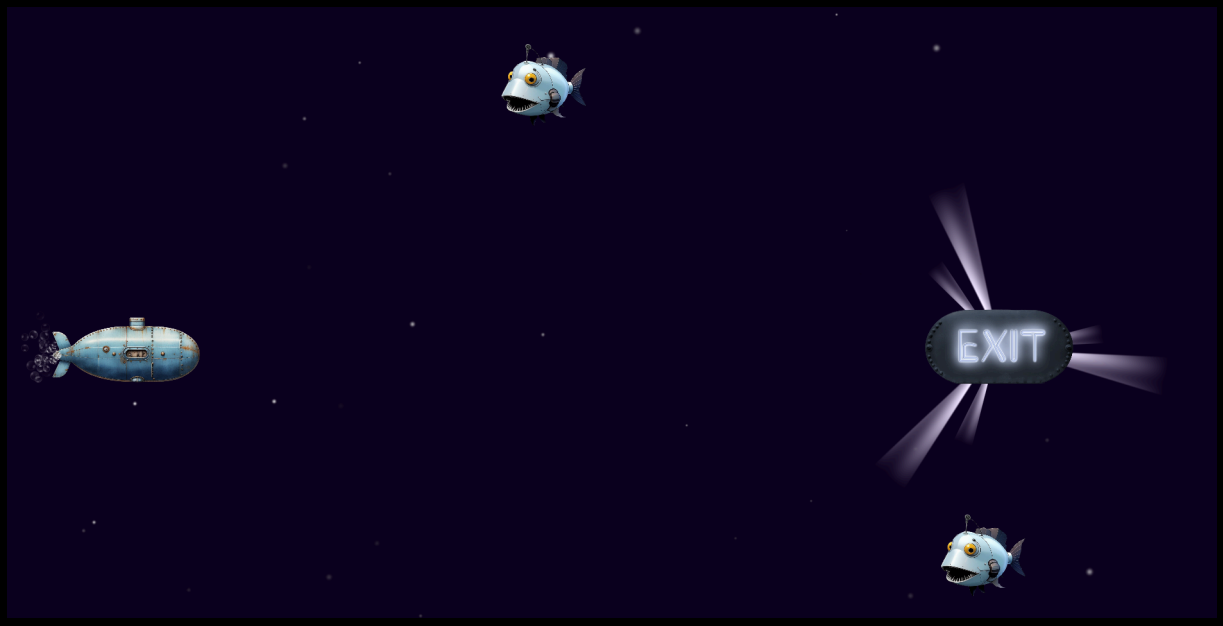
This article builds on the saving and loading tutorial video so it might be a good idea to watch this first.

# The setup

We're going to be using the same setup that was used in the tutorial video, so if you have watched the video, you can probably skip this section. You can also download the full project from GitHub.

## The game

The game is an underwater side-scroller. The player controls a submarine and can shoot torpedoes. There are fish in the sea, which will try to ram the submarine when it gets near them. The goal is to reach the end of the level without getting destroyed by the fish.




## Saving file format

There are many ways to save games in the Godot engine. We can use binary files, JSON or Godot's built-in Resources. Which file format we use, doesn't matter when it comes to compatibility. The formats rather differ in how easy they are to read and write within the Godot engine, how much space they take up and how easy they are to edit by hand.

Here, we will use the Godot resource format, because it is easy to read and write. If you prefer to use a different format, this article should still be useful to you, even though the solutions might look a bit different for other file formats.

## The saved game resource

We save our game in a `SavedGame` resource that looks like this:

```gdscript
class_name SavedGame
extends Resource

## Where is the player (in global coordinates)?
@export var player_position:Vector2
## What is the player's health?
@export var player_health:float
## What additional objects are in the scene?
@export var saved_data:Array[SavedData] = []
```

saved_game.gd

This holds data about the player and a list of data about all the other objects we want to store. Each other object is represented by a `SavedData` resource that looks like this:

```gdscript
class_name SavedData
extends Resource

## Where will the saved object be (in global coordinates)?
@export var position:Vector2
## Which scene will we use to instantiate the saved object?
@export var scene_path:String
```

saved_data.gd

## Save and restoration process

We use the `SaverLoader` class to save and load games. When we save the game, we collect the data from the player and all other objects in the scene and write it into a `SavedGame` resource.

```gdscript
func save_game():
  # make a new saved game
  var saved_game := SavedGame.new()

  # save player data
  saved_game.player_health = _player.health
  saved_game.player_position = _player.global_position

  # create an array to store the data of all other objects
  var saved_data:Array[SavedData] = []

  # use a group call to collect the data from all other objects
  # all objects that are interested in saving their data
  # will react to this call by adding their data to the array
  get_tree().call_group("game_events", \
      "on_save_game", saved_data)

  # store the array in the saved game
  saved_game.saved_data = saved_data

  # and store it into a file
  ResourceSaver.save(saved_game, "user://savegame.tres")
```

saver_loader.gd

When we load the game, we read the data from the `SavedGame` resource. If this has worked, we run a group call to notify all interested game elements that we are about to load a game. All non-static game elements will react to this call by destroying themselves. This way we can clean up the world before we restore the contents of the saved game.

Then we restore the player position and health by just updating the existing player object. Finally, we walk over the `SavedData` array and instantiate the saved objects. We use the `scene_path` property to load the scene and the `position` property to place the object in the scene. Then we call the `on_load_game` function on the object we just instantiated and this function restores the state of the object.

```
func load_game():
  # load the saved game file
  var saved_game:SavedGame = \
    load("user://savegame.tres") as SavedGame

  # bail out if we can't load the file
  if saved_game == null:
    return

  # clean up all non-static objects from the scene
  # by running a group call
  get_tree().call_group("game_events", "on_before_load_game")

  # restore player data
  _player.global_position = saved_game.player_position
  _player.health = min(saved_game.player_health, 200)

  # restore the saved items
  for item in saved_game.saved_data:
    # instantiate the scene for the item
    var scene := load(item.scene_path) as PackedScene
    var restored_node = scene.instantiate()

    # add the item to the world
    _world_root.add_child(restored_node)

    # run a callback to restore the item's state
    if restored_node.has_method("on_load_game"):
      restored_node.on_load_game(item)
```

saver_loader.gd

This is a very generic approach that works for all kinds of objects. We don't need to know anything about the objects we are saving and loading. We just need to make sure that they have an `on_save_game` and an `on_load_game` function that can be called to save and restore their state. With this approach, it is very easy to add new objects to the game and have them automatically saved and restored.

Now that we have established the framework, we can start to look at some of the things that can go wrong when we change our game and how to deal with them.

# Handling changes that break compatibility

## Replacing resource classes

The `SavedData` class only works for very simple objects. It stores a scene and a position and that's it. If we need to save and restore additional data, we will need to create subclasses to hold this additional data. In the video, we gave the fish a new skill that allowed it to reduce its size when it was hit.

Because of this, we need to store the scale of the fish in the saved game. We can't modify the `SavedData` class because the scale is only relevant to the fish. So we create a new class called `SavedFishData` that inherits from `SavedData` and adds a `scale` property.

```
class_name SavedFishData
extends SavedData

@export var scale:Vector2
```

saved_fish_data.gd

Newer saved games will store the fish data in a `SavedFishData` object, but our previously created saved games stored the fish data in a `SavedData`. So when we load, we cannot simply assume that the `SavedData` object is a `SavedFishData` object.

So our loading code needs to be aware of this and check if the `SavedData` object is a `SavedFishData` object. If it is, we can restore the additional data. If it isn't, we will assume the fish is at its default size.

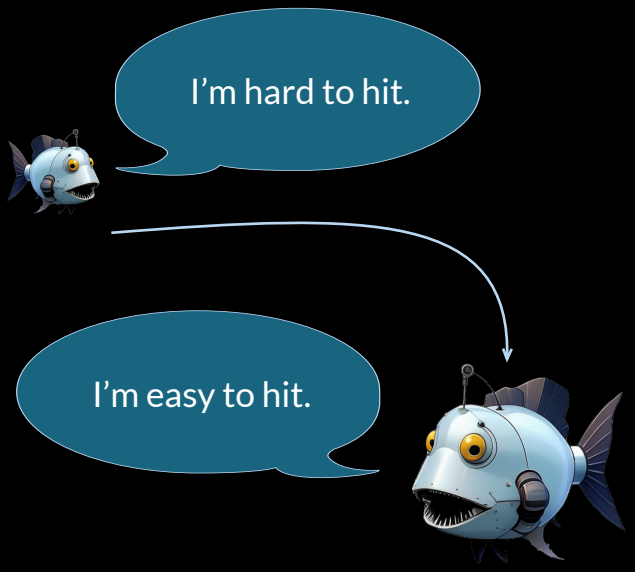
```
func on_load_game(data:SavedData):
    global_position = data.position
    # only restore additional data if data is SavedFishData
    if data is SavedFishData:
      var fish_data = instance as SavedFishData
      scale = saved_data.scale
```

fish.gd

**Summary:** *We can replace resource classes with other resource classes. Godot will automatically load the class that is referenced in the saved resource file. We can use an `is` check to see if the object is of the type we expect and handle different classes accordingly.*

## Adding properties

Our fish can now shrink when it is hit. But during playtesting, we find out that the fish is very hard to hit when it is small. This makes the game a good deal harder. To alleviate this, we now decide that the fish should return to its normal size after 3 seconds.




For this, we can add a `time_until_big` property to the fish that stores how much time is left before the fish returns to its normal size.

```
var time_until_big:float = 0
```

fish.gd

When we save the game, we want to save the current value of `time_until_big` so we can restore it when we load the game. So we need to add a property for this into our `SavedFishData` class.

```
# time left until the fish grows big again
@export var time_until_big:float = 0
```

saved_fish_data.gd

In the `on_save_game` function of the fish, we can now store the value of `time_until_big` in the `SavedFishData` object:

```
func on_save_game(saved_data:Array[SavedData]):
    # ...

    my_data.time_until_big = time_until_big

    # ...
```

fish.gd

This is pretty easy. But what happens when we load an old saved game that doesn't have this property? Godot will handle this for us automatically. When it loads a resource, it tries to restore all the properties that are stored in the resource file. If a property is missing, it will simply leave the property at its default value.

So when we load an old saved game, the `time_until_big` property will be left at its default value of `0` and the fish will immediately grow back to its normal size. So when we introduce a new property, we can simply set a sensible default value and Godot will handle the rest for us.

**Summary:** *When we add a new property to a saved game resource, we can simply set a sensible default value. When we load an old saved game, the property will be left at its default value if it is not present in the saved game file.*

## Renaming properties

After a while, we decide that `time_until_big` is not a very good name. We rename it to `time_until_normal_size`. This is a purely cosmetic change. The property still does the same thing, it just has a different name. There are two ways we can handle this when it comes to saving. We can decide to just rename the property in our `fish.gd` script and leave the `SavedFishData` class as it is. In this case, all we need to do is to update our saving and loading code in the fish to use the new property name:

```gdscript
func on_save_game(saved_data:Array[SavedData]):
    # ...

    my_data.time_until_big = time_until_normal_size

    # ...

func on_load_game(data:SavedData):
    #...

    time_until_normal_size = fish_data.time_until_big

    #...
```

fish.gd

This works fine, but it can get confusing over time when we have a lot of properties that are named differently in the saved game resource than in the actual game element. To alleviate this, we can create a *computed* property in the `SavedFishData` class that is named `time_until_normal_size` but just accesses the `time_until_big` property. This way we can keep the property names in the `SavedFishData` class in sync with the property names in the `fish.gd` script, but we can still load old saved games that use the old property name.

```gdscript
# our original property, we keep this so we can
# load old saved games
@export var time_until_big:float = 0

# a computed property that just accesses the original property,
# so our code stays consistent. Note that this is not
# marked as @export so it won't be saved in the resource file
var time_until_normal_size:float:
  set(value):
    time_until_big = value
  get:
    return time_until_big
```

saved_fish_data.gd

This computed property allows us to keep the knowledge about the old property name in the `SavedFishData` class and the rest of the code can just use the new property name.

**Summary:** *When we rename a property, we can create a* computed *property in the saved game resource that just accesses the old property. This way we can keep the property names in the saved game resource in sync with the property names in the game object but we can still load old saved games that use the old property name.*

## Removing properties

Playtesting goes on and it turns out the whole fish-shrinking mechanic is not a lot of fun. So we decide to remove it. We remove the `time_until_normal_size` property from the fish and the `SavedFishData` class.

What happens when we load an old saved game that still has this property? Godot will handle this for us automatically as well. When it loads a resource and finds a property that is not present in the matching resource class, it will simply ignore it. So when we load an old saved game, the `time_until_normal_size` property will be ignored and the fish will behave as if it never had this property.

**Summary:** *When we remove a property from a saved game resource, this will not break existing saved games.*

## Moving classes

Sooner or later we will want to reorganize our code. We might want to move classes into different folders or rename them. Let's say we want to reorganize our code and move all saving and loading related resources into the `saved_game` folder, where we already have the `SavedGame` and `SavedData` classes.

So the `SavedFishData` class will now be located in `saved_game/saved_fish_data.gd`. Let's have a look at an old saved game:

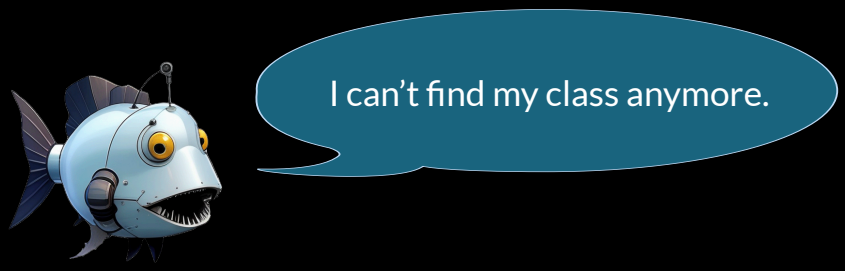
```
[gd_resource type="Resource" script_class="SavedGame"
      load_steps=5 format=3]
;...

[ext_resource type="Script"
    path="res://fish/saved_fish_data.gd" id="1_yjbm1"]

;...
```

saved_game.tres

We can see that the `SavedFishData` class is referenced by its path. So renaming the class is not a problem, but changing the path of the script file will break the saved game. When we load the saved game, Godot will try to load the `SavedFishData` class from the old path (in `res://fish/saved_fish_data.gd`) and fail.




Because loading a resource is fully done by Godot, we have no way of intercepting this. We can solve this by patching our saved game before we load it. A very simple way to do this is to just open the saved game file and replace the old path with the new path. With a little helper class, this is pretty easy:

```gdscript
class_name PathFixer

## This holds a dictionary of paths to replace
## in the input file
const Mappings = {
  "res://fish/saved_fish_data.gd" : \
    "res://saved_game/saved_fish_data.gd",
}

## Fixes paths referenced in the input file. Writes
## a new file to a temporary location and returns the
## path to that file.
static func fix_paths(inputPath:String) -> String:
  var text = FileAccess.get_file_as_string(inputPath)
  if text == "":
    push_error("Failed to read file: " + inputPath)
    return ""

  var result = text
  for search in Mappings:
    var replace = Mappings[search]
    result = result.replace(search, replace)

  # make a folder to store temporary files
  DirAccess.make_dir_recursive_absolute("user://_res_fixer")

  # generate a random file name
  var name = (str(Time.get_ticks_msec()) + \
      str(randi())).md5_text() + ".tres"

  # write patched resource back to temp file
  var final_name = "user://_res_fixer/" + name
  var file = FileAccess.open(final_name, FileAccess.WRITE)
  file.store_string(result)
  file.close()

  return final_name
```

path_fixer.gd

Now we can adapt our loading code in `SaverLoader` to use this helper class to fix the paths in the saved game before we load it:

```gdscript
func load_game():
  var fixed_path = PathFixer.fix_paths("user://savegame.tres")

  var saved_game:SavedGame = load(fixed_path) as SavedGame

  # ...
```

saver_loader.gd

Because we have fixed the path in the saved game, Godot will now be able to load the `SavedFishData` class from the new location and everything will work as expected.

**Summary:** *When we move classes, we can fix the paths in the saved game before we load it. This way we can move classes around without breaking our saved games.*

## Moving scenes or other resources

Our saved game resource stores the path to the scene that we want to instantiate when we load the game. So similar to moving classes, moving scenes can break our saved games. We already have a helper class that can fix paths in saved game resources, so we can use this to fix the paths to the scenes as well.

Say we move the `fish.tscn` from `res://fish/fish.tscn` to `res://enemies/fish/fish.tscn` because we added a bunch of other enemies and want to keep them organized in a separate folder.

To fix this, we can simply add a new mapping to the `PathFixer` class:

```gdscript
class_name PathFixer

# ...

const Mappings = {
  # our fish resource mapping stays...
  "res://fish/saved_fish_data.gd" : \
    "res://saved_game/saved_fish_data.gd",
  # ... but we add a new mapping for the fish scene
  "res://fish/fish.tscn" : \
    "res://enemies/fish/fish.tscn",
}
```

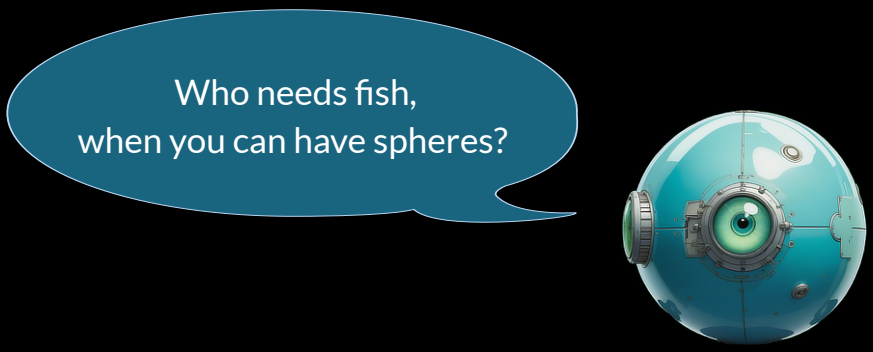
path_fixer.gd

Now when we load an old saved game, the `PathFixer` will fix the path to the fish scene and everything will work as expected.

**Summary:** *When we move scenes or other resources that are referenced in saved games, we can fix the paths in the saved game before we load it.*

## Deleting game elements

We have introduced a bunch of new enemies and we find that the fish is not fun anymore and want to remove it. We remove the fish scene from the game and delete the `SavedFishData` class.



This will naturally break our saved games because they still reference the `SavedFishData` resource and inside of it also to the fish scene:

```
[gd_resource type="Resource" script_class="SavedGame"
      load_steps=5 format=3]

;... this resource references saved_fish_data.gd
[ext_resource type="Script"
    path="res://saved_game/saved_fish_data.gd" id="1_yjbm1"]

; ... this resource references fish.tscn
[sub_resource type="Resource" id="Resource_qvxrh"]
script = ExtResource("1_yjbm1")
scale = Vector2(0.5, 0.5)
position = Vector2(1537, 884)
scene_path = "res://enemies/fish/fish.tscn"
```

saved_game.tres

Now we have a problem. We can't fix the path to the `SavedFishData` class anymore because we have no replacement. Also, we don't want to grow our path fixer into a full-blown resource editor. So what can we do?

We can simply replace the `SavedFishData` class with a dummy resource that does nothing. Let's add a new class named `DeletedData` that inherits from `SavedData` and has no properties:

```
class_name DeletedData
extends SavedData
```

deleted_data.gd

This new class can serve as our replacement, so we can update our `PathFixer` class to use this instead of the `SavedFishData` class:

```
; ...

const Mappings = {
  # ...
  "res://saved_game/saved_fish_data.gd" : \
    "res://saved_game/deleted_data.gd",
  # ...
}
```

path_fixer.gd

Now when Godot loads the old saved game, it will use our dummy class for the saved fish data. This will work because Godot will ignore all properties that are not present in the class. Then after we loaded the saved game, we can simply check if the `SavedData` object is a `DeletedData` object and ignore it instead of instantiating a scene and restoring the state:

```
func load_game():
  # ...

  for item in saved_game.saved_data:
    # ...
    if item is DeletedData:
      continue

    # ...
```

saver_loader.gd

This way the removed game elements will simply not be restored when we load an old saved game.

**Summary:** *When we delete game elements, we can replace their saved data entries in the saved game with a reference to a dummy class and then skip over this loaded dummy class when we encounter it.*

## Final thoughts

We have seen that there are quite a lot of ways to break saved games and looked at ways to address a variety of cases. However, if you do a major rewrite of your game, you may change so many things that it becomes simply impossible to migrate data from an old saved game into the new world.

This is very common - you will find a lot of examples online where game updates broke saved games and the only thing players could do is to start new.

The more changes you make, the more moving parts you need to keep track of. At some point, the amount of effort that you have to put into this, may not be worth it anymore. So in the end you will need to decide how far ahead you want to plan, and what amount of effort you are willing to spend to keep saved games functioning after updates.

**Summary:** *Sometimes it is simply not worth the effort to keep saved games working after major updates. It's okay to just tell your players that they need to start a new game after a major update.*

## Thank you!

You made it to the end of this article. I hope you found it useful. If this has helped you, please consider supporting me on Ko-fi. Thank you very much and happy Godotneering!