# Contents



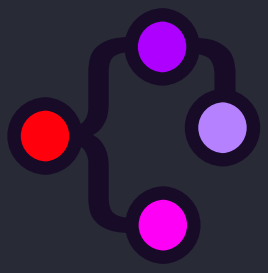


Figure 1: Resource Wrangler Logo

# Resource Wrangler (RW)

**Updated December 2023**

## What is a Resource?

The short answer is, “An Object”. That really says it all. It’s some data and some code. Done.

I like to think of them as little boxes of stuff. When a Sprite2D needs a texture, you fetch a texture-box and plug it in. When your mesh needs a Material, you give it a material box.

In code, stuff starts as classes. I think of these as little ghostly boxes with dotted outlines. There’s only one ghost material box, one Texture2D box and so on.

When you want to make a *specific* Texture, say a picture of a cat, then you go through some ui steps in Godot to make a new Texture2D resource and you point its texture property to the cat picture. Now you have a “cat” box ready for whatever Node (or bit of code) that needs it later.

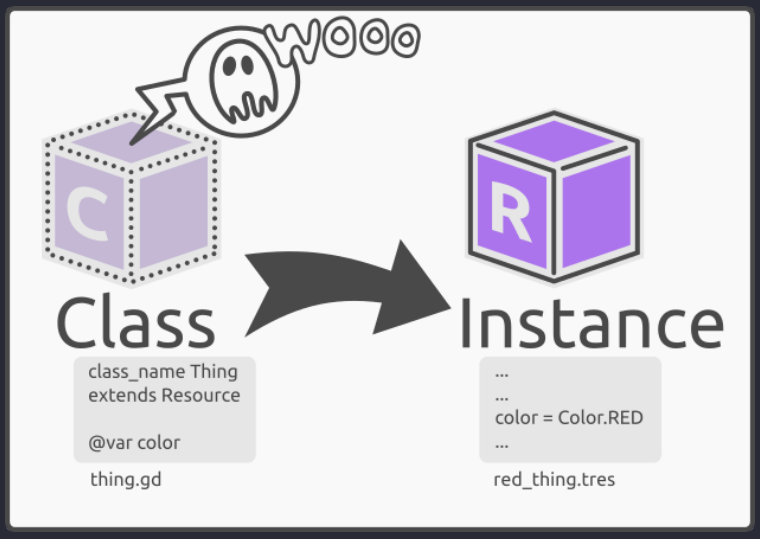



Figure 2: Ghosts and Boxlins!

So, these resource-boxes contain pictures, meshes, frames for animations; the uses are up to you.

You can make your own boxes too! Say you need one to hold a description of a room in a text adventure game, go ahead just write a short script that extends Resource and you’re done.

Here’s a quick example in gdscript:

“room.gd”

```
class_name AdventureRoom
extends Resource

@export var room_name:String
@export var room_description:String

# You can also use code to do whatever you need:
func describe():
  print(room_description)
```

Once you save this, Godot will have a new class (a ghost box!) called “AdventureRoom”. Try the usual way of making a real box of it:

1. Go somewhere in the file explorer. Right click and choose Create New and then Resource.
2. In the dialogue that opens, search for “adventure” and you will see your new class!

3. Double click it and give the box a new name like “room1.tres”

You have just made an actual instance of an “AdventureRoom” and it exists as a file (tres just means text-resource). You can even right-click on it and choose “Open in External Program” to see the contents. That’s worth doing to get a feel for what is happening and lose the fear of these files.

So, ghost-boxes are *classes* (gdscript files) and real-boxes are *instances* of the ghost.

You may wonder how you can actually use room1.tres? Well, for a start double-click it to see it in the inspector. You will see the two properties you made; enter a name and a bit of text for the room. Then Ctrl+s to save it. (Once again, look at the file in an external text-editor to see what’s in the tres file. It’s pretty neat.)

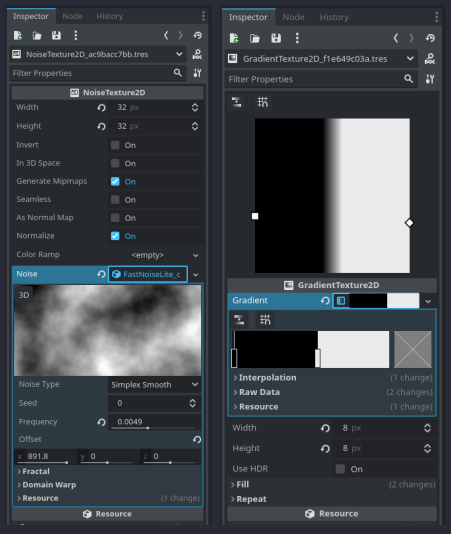
Now you can imagine making many room-files like “room2.tres”, “mainhall.tres”, “gallery.tres” and so on. You could even create a new resource called “AdventureMap” which could hold a dictionary of all the rooms, or something. It’s open to your imagination and needs.

You can make other resources, like “AdventureDoor” which in-turn can hold info about each door; like, if it’s locked, what key it needs, what noise it makes when it opens and so on. Each door could also have an array of which rooms it connects-to.

In your own code, you can grab resources and instance them as variables to use their contents as you require. Resources lend themselves to this kind of code-only use where what you need is a way to organize data. They’re very flexible.

## Boxes in boxes

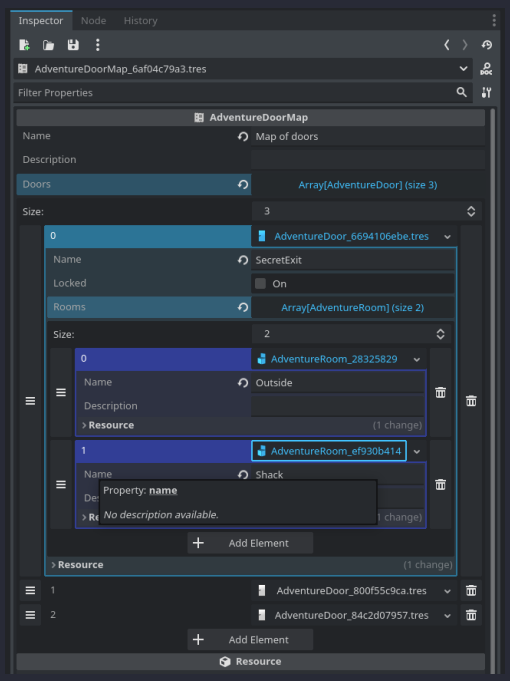
Some of the built-in resources that come with Godot need other resources. For example a NoiseTexture2D resource needs a “noise” and what fits is a FastNoiseLite resource.




*A NoiseTexture2D resource (left) with a FastNoiseLite sub-resource and a GradientTexture2D resource (right) with a Gradient sub-resource.*

You will also find many *Nodes* (which are just Objects too) that want resources as inputs. All those gradients and noise textures have to go somewhere!

Stuffing boxes into other boxes is nothing new in coding. You’ll do this all the time with arrays and dictionaries. Take the “AdventureDoor” example; you could have an *array* of rooms in it. Now you can have any number of room-boxes (instances) *inside* your door-box.

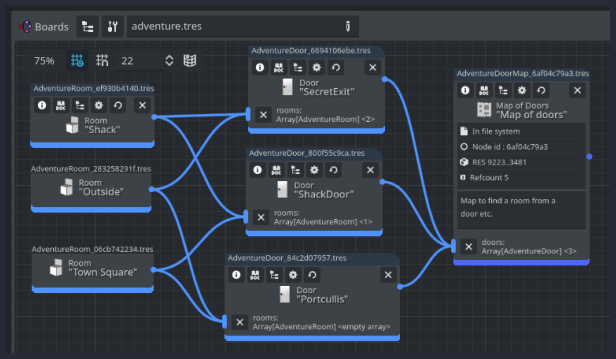



*Boxes all the way in.*

However, in the current Godot ui, it might be a little hard to keep track of what's inside what and where stuff is. It's also rather tricky to assign resources to their slots and even to make new resources in the first place.

## Enter the Wrangler (Dah dah dah dah!)

For some reason I really like node-graphs like the visual shader. I find them easier to understand. Having the nodes before me gives a better overview of a situation than some small section of code that I have to scroll through.




*"Adventure" nodes connecting rooms to doors.*

When it comes to resource boxes (and boxes inside those boxes) I also struggle with the way Godot's Inspector handles them. It's not bad and has its advantages, but it suffers from the problem of ever-shrinking space when you have many boxes nested.

"Why not pull all that out into a node-graph?", I asked myself about a year ago. And so it began.. and it's been difficult.

Resource Wrangler shows Resources as *nodes* on a graph. When you add a new node, you instance a ghost as a real box. Each box is saved for you as a tres file. You work on what I call

a “Board” which lets you connect the nodes and build-out whatever structures you need.

It becomes quite nice to develop custom resources this way too. It’s a nice flow of a bit of scripting, some noding, some noodling, then more scripting. Repeat.

As you go along, you can quickly see where things are not making sense and you can go back and forth from code to nodes. (There’s a little refresh button that updates all a resource’s properties). One can also code new classes and drop the gd files onto the board to spawn instances and so continue the process.

It’s a fluid way to visually dance with your data.

## Short intro

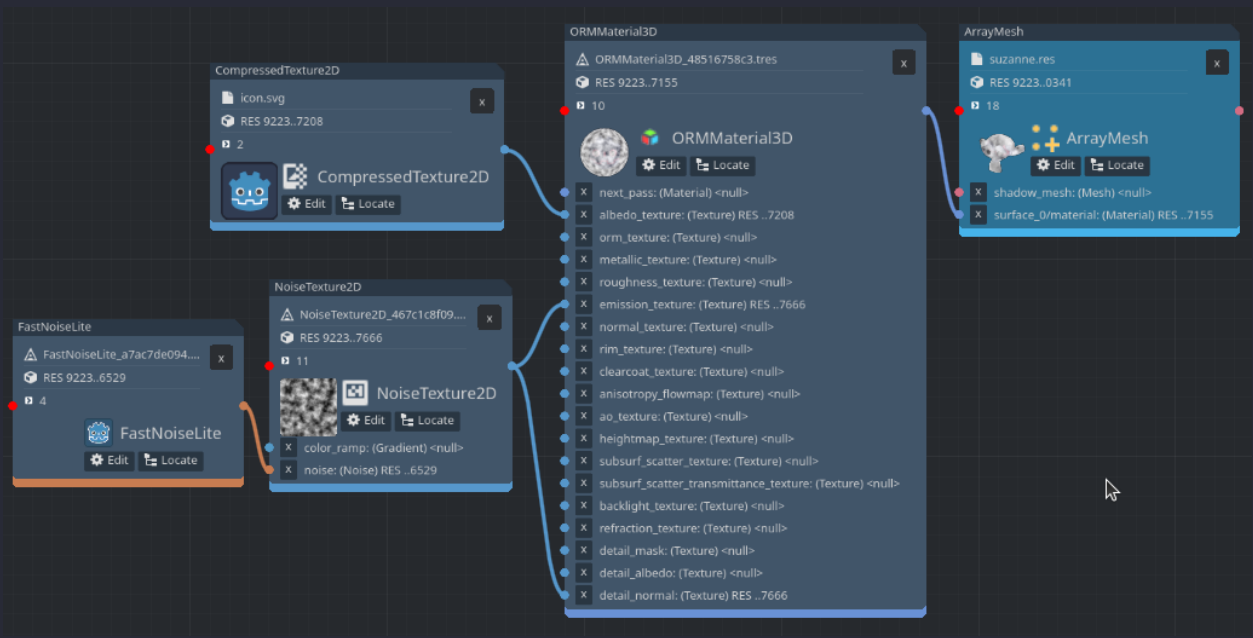



Figure 3: An ArrayMesh Resource with a material and its various textures.

*An ArrayMesh Resource with a material and its various textures.*

Once the Addon is activated (in Project Settings) you should see a new button named “Resource Wrangler” on the bottom menu area, alongside Shader Editor.

- When you open RW, the last board will be restored. If there was none, a new one is created with a random name. (You can rename it.)
- Use the ‘Boards’ menu to make new boards, open them and save them (Ctrl+s) also saves.
- To quickly rename a board, use the text-box with the pencil icon. Press enter to rename.
- There’s a button to locate the board in the FileSystem.
- There’s another button to open the Project Settings for the plugin.
- You can move, delete and rename board files from the FileSystem in Godot. (They don’t have to live in the default folder.)
- You can also move and rename resource files (which are what underlie each node) and the board should keep track of them. Ymmv.

## How to recover from weird

Godot can get weird. When it does, the best thing to do is save and exit. Keep an eye on the error messages because they try quite hard to tell you what went wrong. Sometimes you even have to close, open and then close and open again! I kid you not.

A recurring villian is cycles of references between Resources. If A contains B which contains A, bad things happen. RW tries to prevent these, but as you develop and start and stop the plugin, etc., stuff happens. Keep in mind that you can always edit tres files in a text editor; they are not hard to understand.

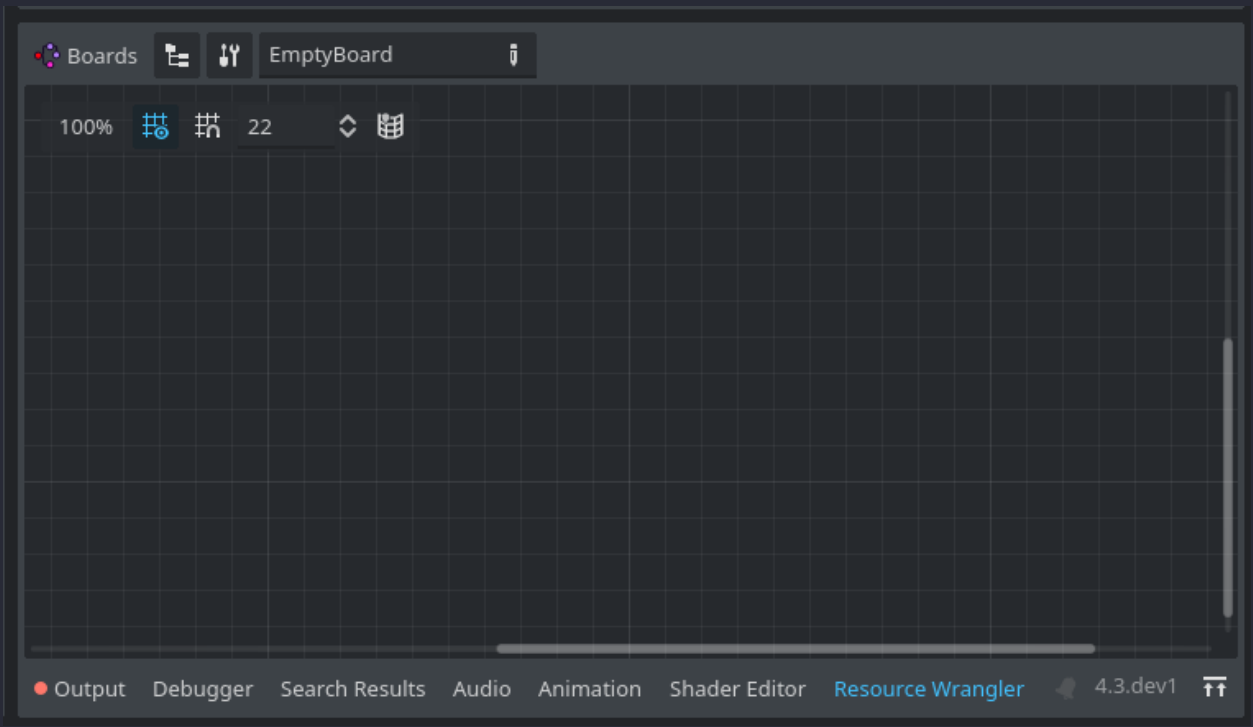
## Settings

Open the Project Settings and look for “Resource Wrangler”. In there you will find the settings where you can change some paths and stuff.

The ‘automade_path’ field is where Resources will be saved when you make them in the editor. You can rename and drag them out of there to other places as you need.

There’s also a button on the menu bar to quickly open the settings.

## Boards




*An empty board. The menu button is on the left.*

The plugin shows a “board” where the nodes will appear. You can open a board by pressing the “Resource Wrangler” button on the bottom of the screen, alongside “Shader Editor”.

You can save a board by Ctrl+S and it will have a “.tres” extension.

To rename a board, use the text edit control at the top of the board, or rename it in the filesystem. (You may have to reopen it after that.)

You can open board-files (which are saved as .tres files) by double-clicking “board_name_whatever.tres” file (in the filesystem inspector) or by dropping one onto the board.

## Nodes

Each node in RW represent a Resource Object (and file). There are several ways to make them:

1. **Drop** resources onto the board from the File Manager.
2. **Drop** resources from the inspector. (Ymmv.)
3. **Right-click** the board and choose a resource from the Chooser.
4. **Drag + Drop** from the *input* ports of a node (left hand side) to make a new input resource.

- From *empty ports* when you drop, you’ll see a list of suitable choices in a grid of buttons. If not, that resource will be shown as a new node.
- From *array ports* you can only make new entries.

**Please note**: Not all resource types can be made in code. I filter those out, so if you can’t find a resource that’s probably why.

### Newly made resources

When a new resource is made, it is automatically saved into the “Automade Path”. You can change that path in the settings. Each resource is saved as a .tres file with a unique ID.

These nodes (saved in the Automade Path) are flagged as “Automade” to remind you where they are. You can move, rename and delete all resource files from the FileSystem. They don’t have to stay in the automade path.

After using RW a while, the automades will mount-up. I am pondering a way to automatically clean them up (those that have nothing else using them). You can use the built-in Godot methods to clean them up for now. [Project > Project Tools > Orphan Resource Explorer]

## Ports

A port represents a Resource property in the Node. That property can be empty (null) or full (a reference).

The ports along the left of the nodes are **input** ports and the single port on the right is the **output** port.

If you want to place a resource (represented by any particular node) into some slot in the Inspector (for some node or other), then you can *drag and drop* from the output port while holding **SHIFT** to get a resource icon to drop where you want to.

Alternatively, you can press the Show in Filesystem button (on the node) to show the resource file in the filesystem. Then you can drag it from there.

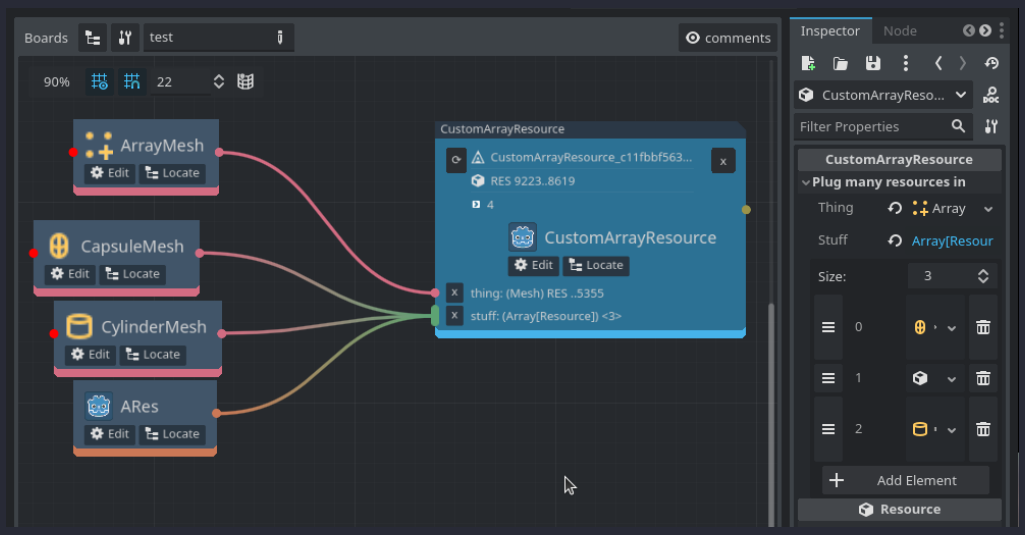
When you delete an incoming node, the property of the port *will not* become empty.

*This is a design decision — resources contain other resources that may have multiple instances, removing a resource from one property would alter those. The integrity of resources is important.*

There is an “X” button next to each port to actually empty that property.

### Array Ports

You can use Arrays of typed resources, e.g. `@export var A = Array[Resource]`, in your custom resource classes. When adding such a node, it will draw with slightly larger ports (to indicate where the arrays are). You can add many noodles into one array port.




*A Custom Resource (made from a script) showing a Mesh port and an Array[Resource] port. You can also see the Inspector showing the contents of the Resource.*

### Array ports are a little weird

Make resource nodes and then connect them into the array, or drag out from the Array port and choose from the Chooser. (Dragging out *will not* disconnect because there's no way for me to tell which array element any noodle represents.)

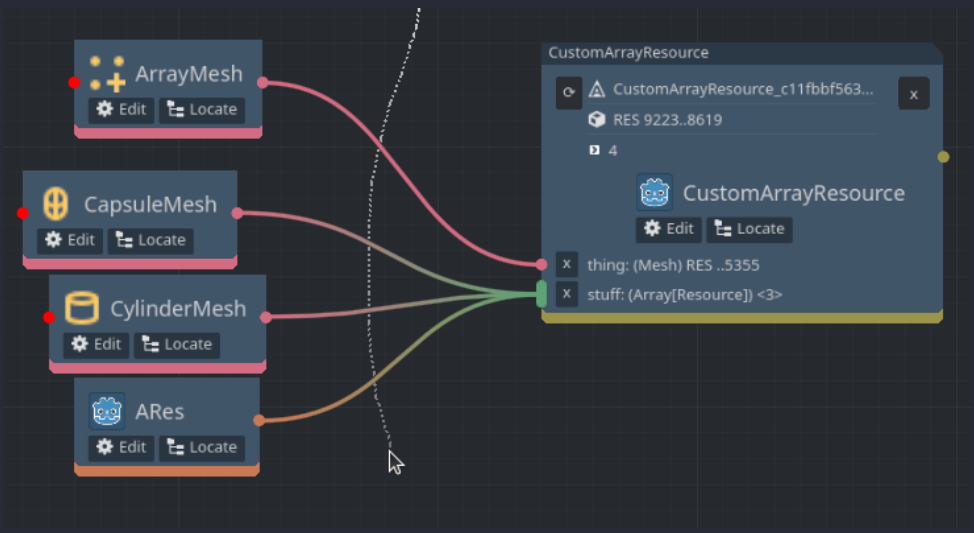
If you want to see individual nodes within an array, look in the *Inspector*. You can *drag and drop* them to the board (from the inspector) to make a node. If you connect that node by noodle to the array port, it *will not append again* because it's *already* in your array, but at least the noodle will be there. :)

## Noodles

Noodles connect ports. When you connect one, a resource is placed into the input port (on the LHS of a node). *When you disconnect one the port's value becomes null.*

To disconnect noodles, either:

1. Press the X next to the port. That will cut it.
2. Use the Ctrl + RMB drag across one or more noodles to cut them all.

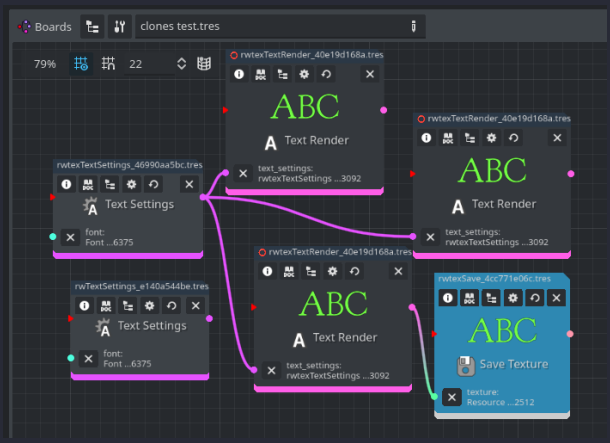



*Another way to disconnect noodles is to hold Ctrl+RMB and to drag across one (or more) noodles.*

*Please note:* Dragging-out from a connected noodle no longer disconnects it.

## Clones

Resources are instances of some class (whether your own code or Godot's built-ins). It stands to reason that one can drop a resource instance onto a board twice. There would now be two nodes of the *same* resource. I call these nodes "clones". You can work with them as normal and (hopefully) the graph will update as expected as you go along.




*The clones have a little red circle and should always look identical.*

## Custom Resource Nodes

Making your own resources is one of the superpowers of Godot. RW can display your custom resources as nodes and allow you to work with them in the graph.

There are some some details for you to know:

## Tool Keyword

All your custom Resources **must** have `@tool` at the top. Without it, you will get weird errors and your nodes won't connect properly.

## Samples

Demo boards are in ""res://addons/resource_wrangler/docs/demos/boards/"

(The boards are just .tres files, but you have to first open the plugin and then use the menu ("Boards" button) to open them. You can also drop any board tres file onto a board to open it, in its own board.

"res://addons/resource_wrangler/docs/demos/boards/basic_nodes/"

These scripts show how you can make your own basic Resource nodes. Try drop any of them onto a board to see what happens. Also look at their code.

## Extra name for Resources

If you supply a propery called 'name', it will be be shown on the node.

```
@export var name:String
```

## Icons

Use @icon in your custom script and it will become the icon for the node:

```
@icon("icons/adventure_door.cleaned.svg") # for example
```

The path can be relative or absolute.

## Resource Previews

Supply this code to hint what to use as a preview:

```
@tool # At the very top. And probably reload the project too.

var preview_this : Resource:
  get:
  return some_resource_you_want_to_preview
```

You must call it 'preview_this'; and it should not be an @export var.

## Metadata

These are special vars you can use:

1. *display_class_name_as* : (string) Will show this name rather then the class_name
2. *category* : (string) Will sort this resource under this category in the choosers
3. *noinstance* : (bool) Will prevent this class from appearing in the choosers
4. *partial_save_name* : (string) Will influence the resource filename
5. *deny_list* : (array) of class names (Strings) that a node will reject from all ports

Metadata all goes into a special method:

```
static func rw_metadata():
  return {
  &"display_class_name_as" : &"Door",
  &"category" : &"Adventure Demo",
  &"deny_list" : [&"ArrayMesh", &"StandardMaterial3D"]
  etc..
  }
```

Keep in mind, if you extend this class, you should probably use super() to feed the metadata from the super class:

```
static func rw_metadata():
  var md = super() # gets the dict from the super-class
  md[&"display_class_name_as"] = &"An Alien Nubbin" # Alter a key
  return md
```

### Extending the Node UI

See "…/docs/samples_nodes/extended_nodes/extended_node.gd" for the example.

If you want to 'glue' extra controls onto a node, you can use this process:

```
@tool # <-- make sure this is used
class_name ExtendedResourceNode # Your name here
extends rwExtendedResourceBase # must extend this class

# Your Exports As you please
@export_group("Meshes")
@export var my_meshes:Array[Mesh]

# Supply a Scene and Override show_node

# This is the scene that is your gui
const SCENE_PRELOAD = preload("sample_extended_ui.tscn")

# Implement this func:
# Calls parent which handles adding the gui
# Then does the necc to show the data in the gui
# _nop just means no operation, i.e. ignore it; leave as-is
func show_node_gui(graphnode:GraphNode, data: Dictionary, _nop) -> void:
  super.show_node_gui(graphnode, data, SCENE_PRELOAD)
  # Actual body up to you.. See example file.
```

(You need not use rwExtendedResourceBase if you don't want to, just incorporate its code in your own class and all should work.)

*TIP*: If you get weird errors like *Resource has no method "show_node_gui"* then restart the project. Godot is iffy like that.

## Quick Tutorial

TODO: Split this into sections so it fits better.

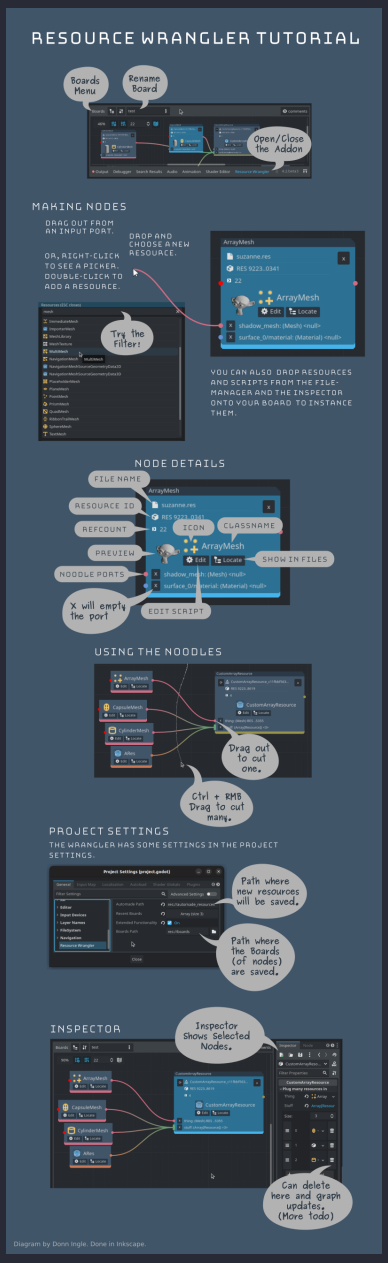



Figure 4: Quick tutorial diagram

## Cleaning Up Resource Files

```
for f in *.tres; do git grep $f; [[ $? == 1 ]] && echo $f >> failed; done
```

That will make a file named ‘failed’ which contains paths to resource files which are not mentioned anywhere in the project: not in other resources, scenes or scripts. Combine that with the built-in orphan tool (in Godot) and your common sense to get rid of unused resource files.

## TODO’s

- Fix _func names that are not actually overrides.
- Better README
- i18n etc.
- Copy/Cut and Paste (also between boards)
- Still some troubles when board files are deleted and still showing in the Boards recent files menu.

## Technical Notes

### Addon Lessons

For some reason one can’t use class_name in addons. I could be wrong, but I could not get it right. So, you end-up using this pattern:

1. `const ChooserThingScene = preload("res://addons/resource_wrangler/components/chooser_thing.tscn")`
2. `const ChooserThing = preload("res://addons/resource_wrangler/components/chooser_thing.gd")`
3. `var chooser_thing: ChooserThing = ChooserThingScene.instantiate()`

- You assign the SCENE to a const
- You assign the SCRIPT to another const (and you also have the same script added to the scene’s root node)
- You make a new object by assigning a var of the TYPE of the SCRIPT to the SCENE’s const dot instantiate()

### Script instancing issues

In the case of a resource which is based on a script, like in the rwNode, the loaded res seems not to have access to its script vars for some reason. (If i make the script a @tool, then it works.)

BUT over in chooser_thing.gd, when I new the scripts, I can reach those vars *without* @tool being there. *Shrug*

```
## in chooser_thing.gd I was doing this:
var a = ProjectSettings.get_global_class_list()
for dict in a:
  if dict.get("path", false):
  var _tmp = load(dict.path)
  var _has_rw_node_data = _tmp.get_script_property_list().filter(
  func(d): return d.name == &"rw_node_data")
  if _has_rw_node_data:
  if Hacks.can_we_instantiate(dict.class):
  var _new = _tmp.new()
```

```
        #Bam! I can reach rw_node_data
    var _d = _new.get(&"rw_node_data")
    # etc.
```

So, if I want to access variables within, I must new() scripts which raises several issues:

1. _init will run and that might not be cool.  You can use an Engine.is_editor_hint() test to stop that.
2. If you extend a class that has that var rw_node_data={} in it then you will get the name and category of the class you are extending. This requires you to do:

```
func _init() -> void:
rw_node_data = {
display_name = "My name!", #<-- to override super
category = "Sample"
}
```

Both of these are a thing you have to remember and I must document...

I am going to use the ProjectSettings as a way of passing this info between plugins.

## Credits

Thanks to these ultra-cool peeps who helped me along the way:

1. https://mastodon.gamedev.place/@jdbaudi
2. https://mastodon.gamedev.place/@exoticorn
3. https://mastodon.gamedev.place/@efi@chitter.xyz