Welcome to the Course

Thank you for joining this course!

Before we begin, these are the basic requirements for this course:

Prior knowledge

In this course, we will be building a 2D real-time strategy game using the Godot game engine. This game will include the ability to select and move groups of units, along with separate enemy units, and the ability for these units to attack each other.

Course Requirements

In order to easily follow along with this course, you should already have:

- A basic understanding of the Godot editor.
- A basic understanding of the GDScript language.

However, you don’t need to be overly proficient in either one. We will be going through each process of creating the game, step by step, especially when it comes to the scripting side of things. Of course, the more you know before starting, the easier to follow along, but only basic knowledge is required.

Course Features

In this course, we are going to cover a number of Godot features that you may not have used before. These include:

- **Pathfinding** – This will allow our units to navigate around obstacles in the scene. This is a very common feature of AIs in video games.
- **Mouse Interactions** – Here we will handle the selection of units and moving them around, all done with mouse clicks on the screen.
- **Extending Classes** – Inheritance and extending classes will allow us to reuse and expand upon scripts to create new functionality.

All of these features should be easy to pick up and use, given that you have a basic understanding of Godot and GDScript already.

Project Features

This course can be broken down into a few main sections:

1. **Player Units** – We will create the ability to select and command units, along with an AI navigation system based on pathfinding.
2. **Enemy Units** – We will create enemy units that can chase and attack nearby player units. These will be very similar to our player units, however, we won’t be directly controlling them.
3. **Game Manager** – This will handle the gameplay.

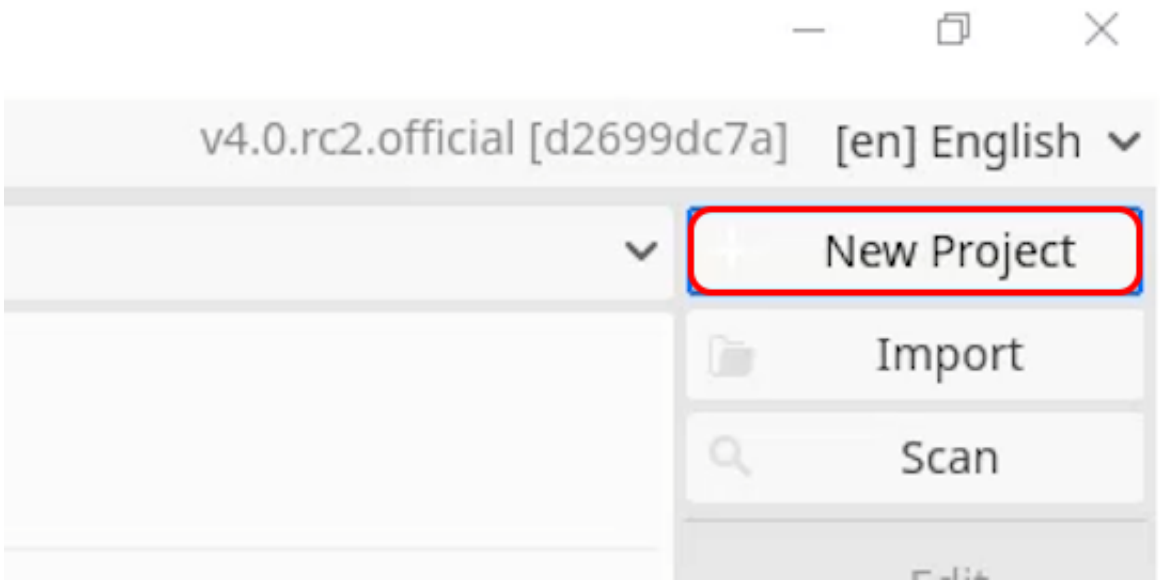
About Zenva

Zenva is an online learning academy with over 1 million students. We offer a wide range of courses for people who are just starting out or for people who want to learn something new. The courses are also very versatile, allowing you to learn in whatever way you want. You can choose to view the online video tutorials, read the included summaries, or follow along with the instructor using the included course files.

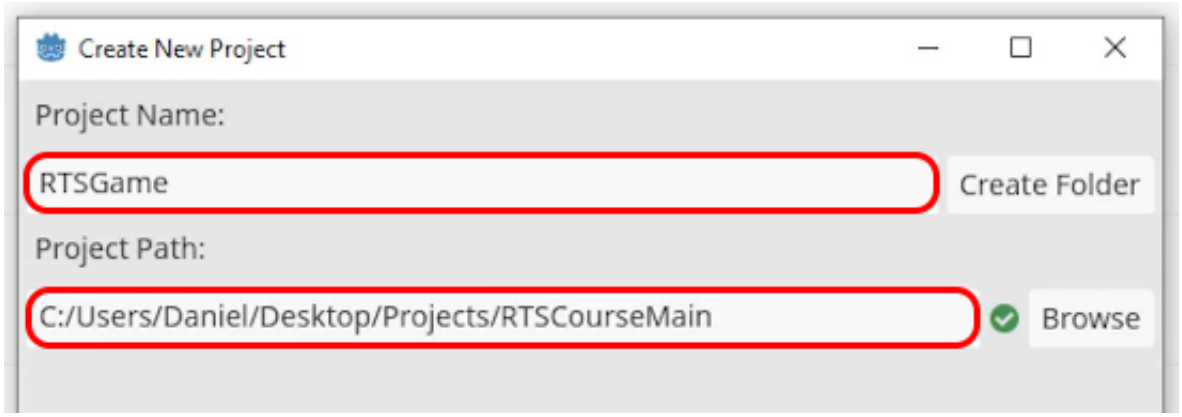
Godot 4.0

This course was made using version 4.0 of the Godot engine. If you are encountering any issues with the course material, that may be due to your version of the engine. Many of the features explored in this course are different or missing from previous Godot versions and may also be different in future versions.

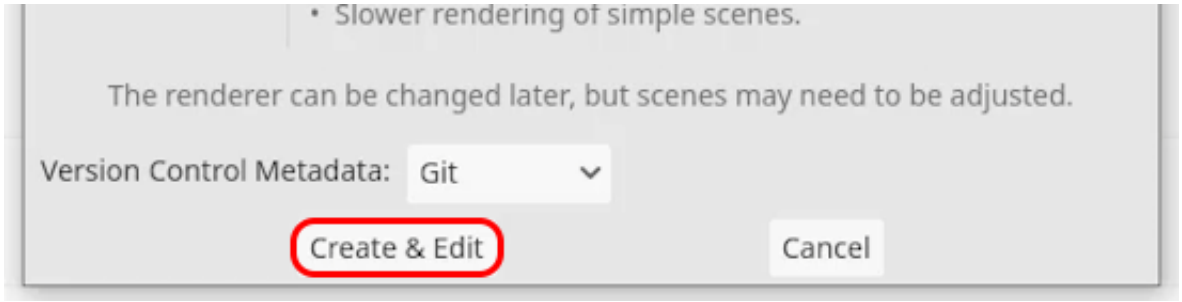
In this lesson, we will set up the project in Godot. We will begin in the *Project Manager* window that appears when you open the Godot executable. Press the **New Project** button to begin the project creation.



We can **name** this project something like *"RTSGame"* and choose a **Project Path** on your computer to save it in. If your project isn't empty, you will be prompted to choose the **Create Folder** button, which is highly recommended otherwise you may have your Godot project scattered everywhere.



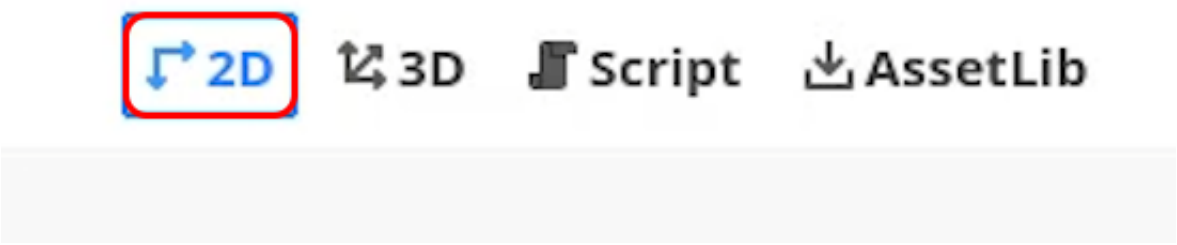
Finally, choose the **Create & Edit** button.



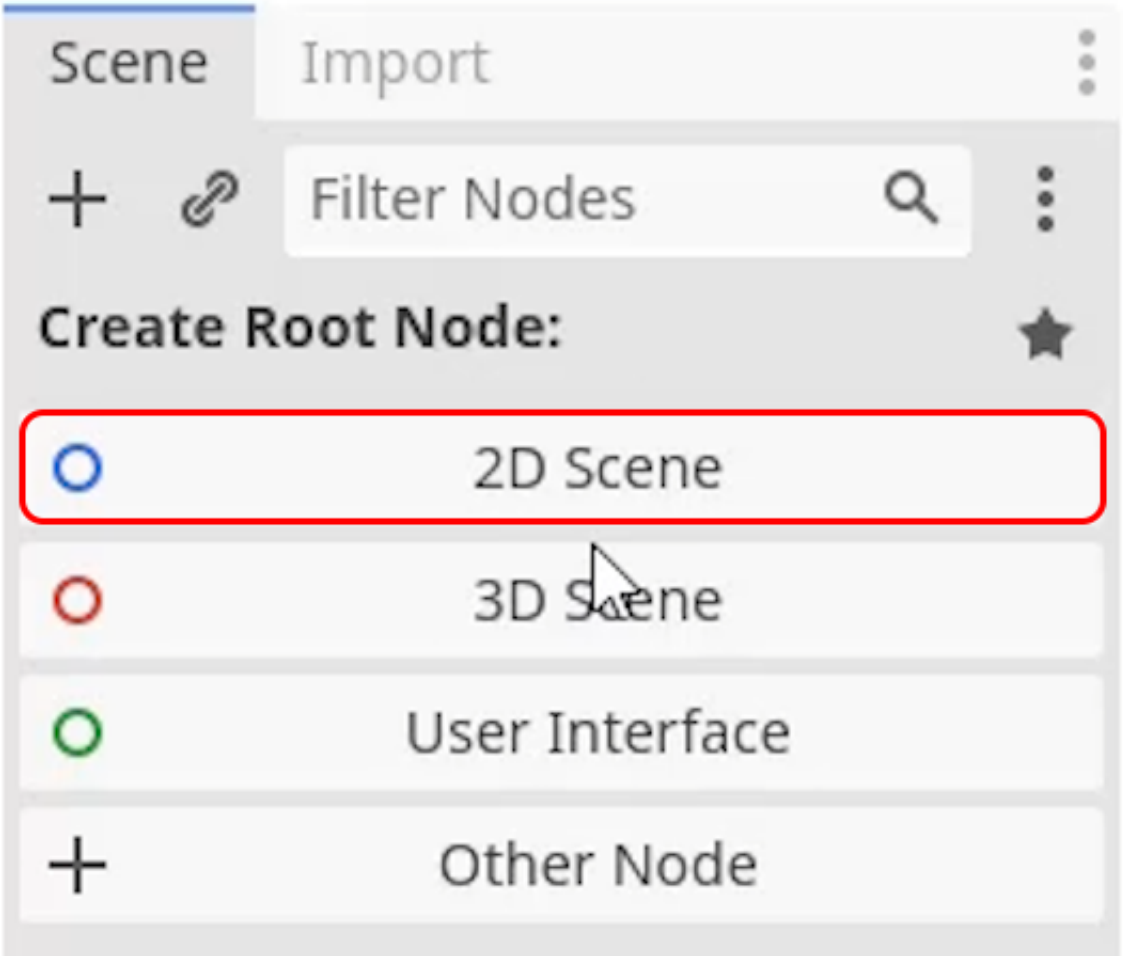
Creating the Scene

The first thing we will do is switch over to the **2D** environment by selecting the **2D** button on the

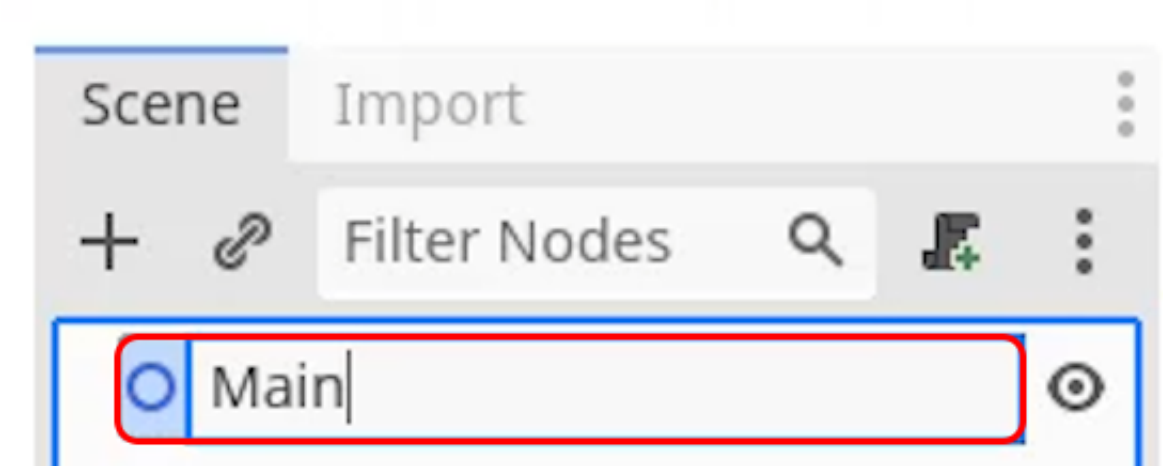
menu.



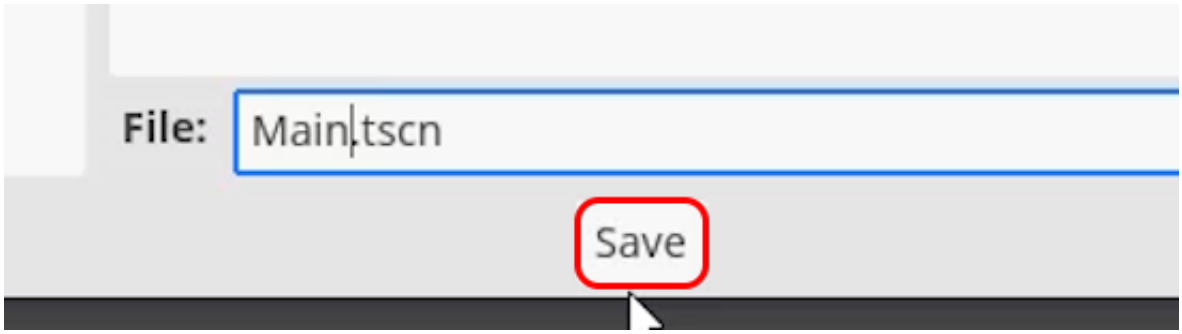
We will choose **2D Scene** as the root node for our main scene.



We can **rename** this root node to *"Main"* so that we can easily access it later.

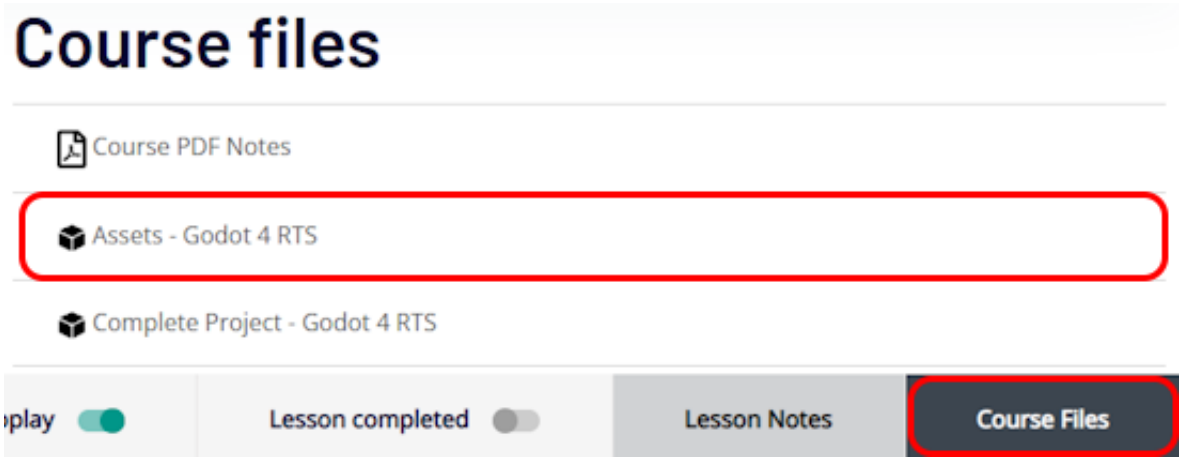


Save this scene (**CTRL+S**) as *"Main.tscn"*.

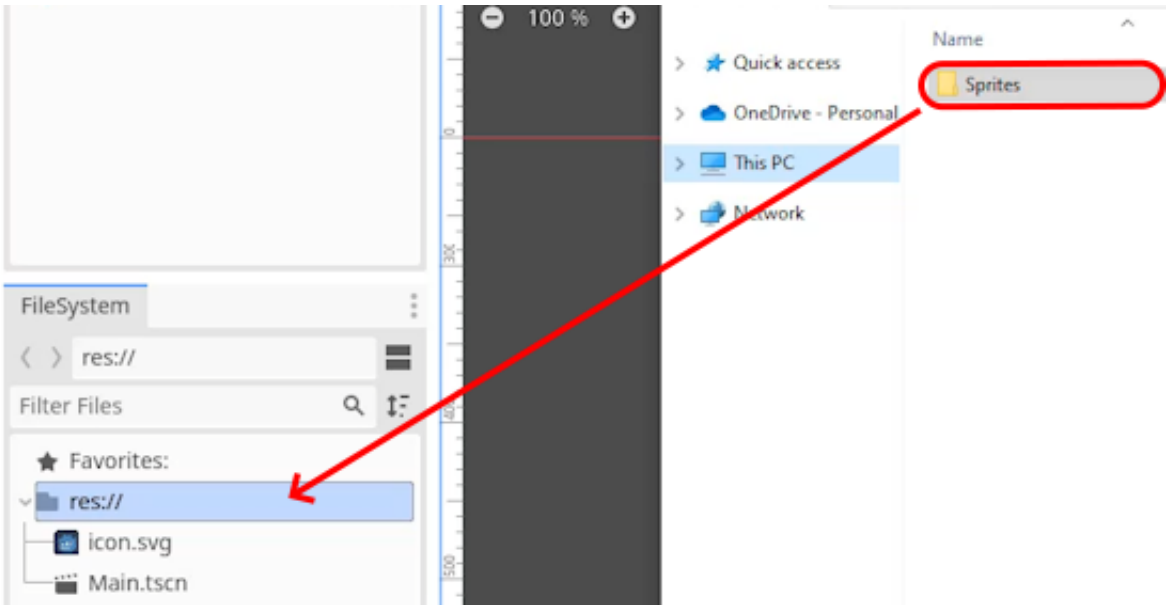


Adding Sprites

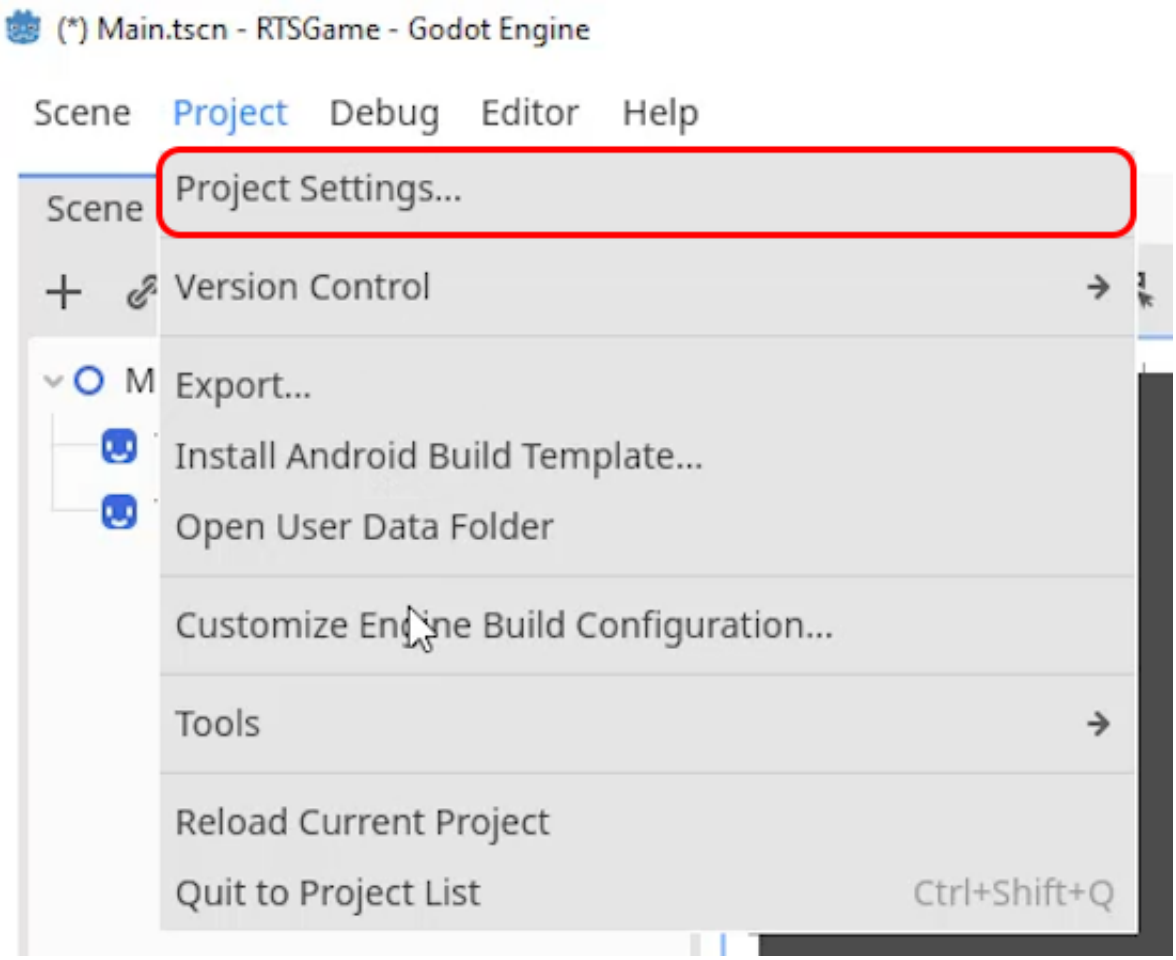
In this course, we will be using a few sprites to represent our units and the environment. You can use your own, however, if you prefer we have supplied a *zip* file containing all the sprites you will need in the **Lesson Notes** section on Zenva.



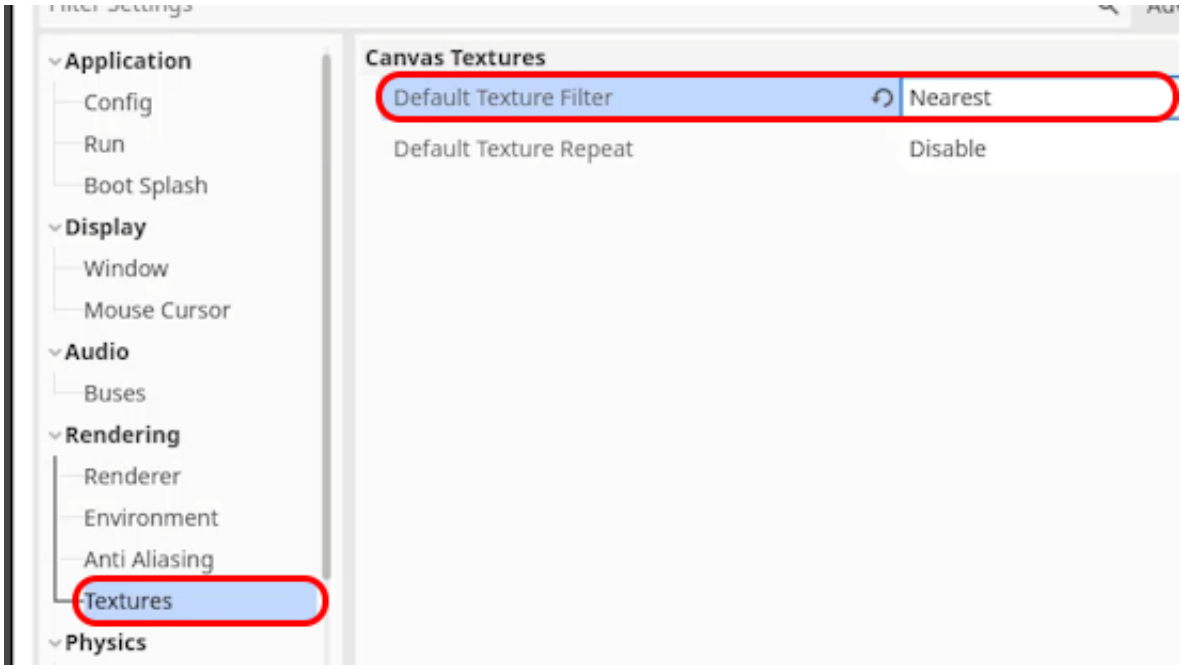
To include these files in the Godot project, unzip the file and drag the **Sprites** folder into the *FileSystem* in Godot.



Because these are pixel art assets, we need to make a few adjustments to avoid our sprites appearing blurry. To fix this we will open the **Project Settings** window.



Under the **Textures** tab, select **Default Texture Filter** and set it to *Nearest*, which will make the pixel art scale properly.

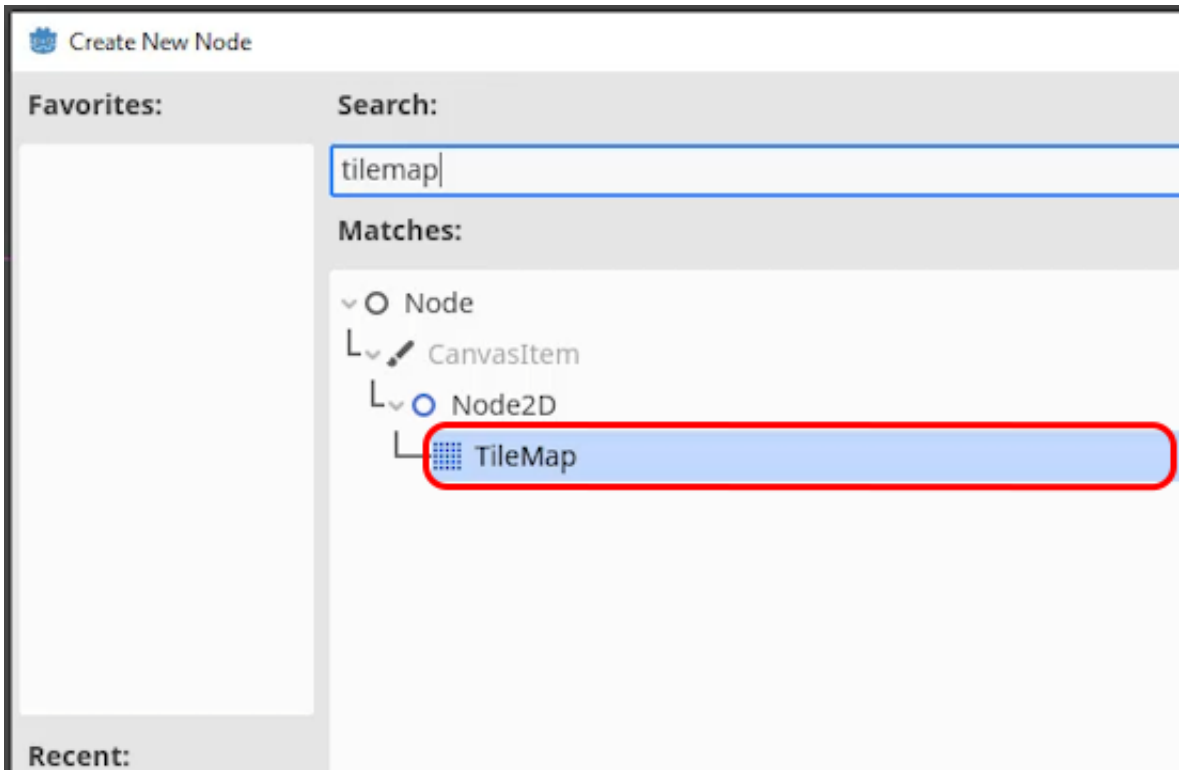


If you are using your own high-resolution assets, this won’t be necessary. In the next lesson, we will be setting up our environment for our units to move around in.

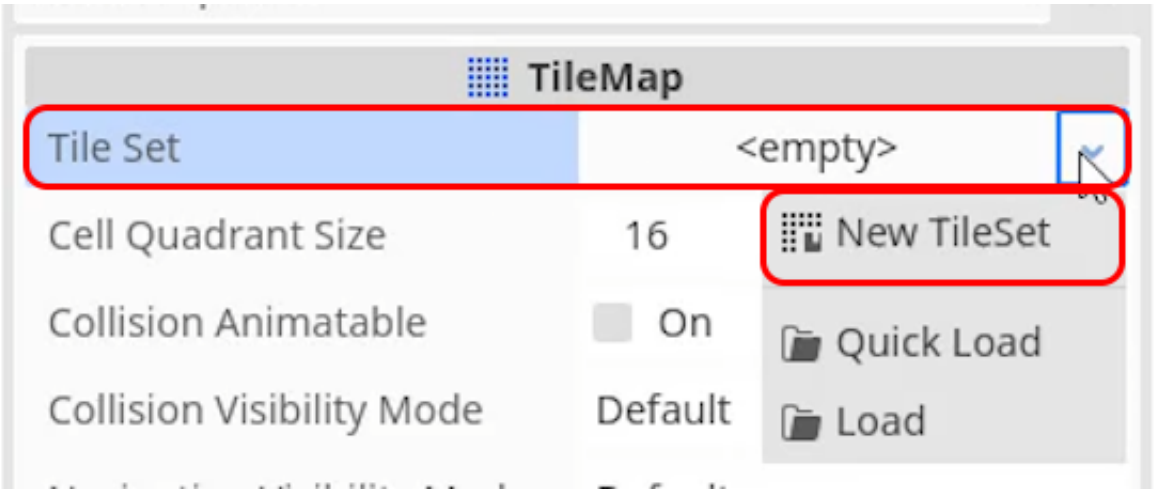
In this lesson, we will be creating an environment for our game to take place in. The environment will contain obstacles that our units will have to navigate around. For this, we will be using a **tilemap** which is a grid that can have sprites (known as tiles) placed on it, allowing us to build levels very quickly.

Using a Tilemap

The first step is to add a new **TileMap** node to the scene. This node will be in charge of handling and rendering our tilemap to the screen.

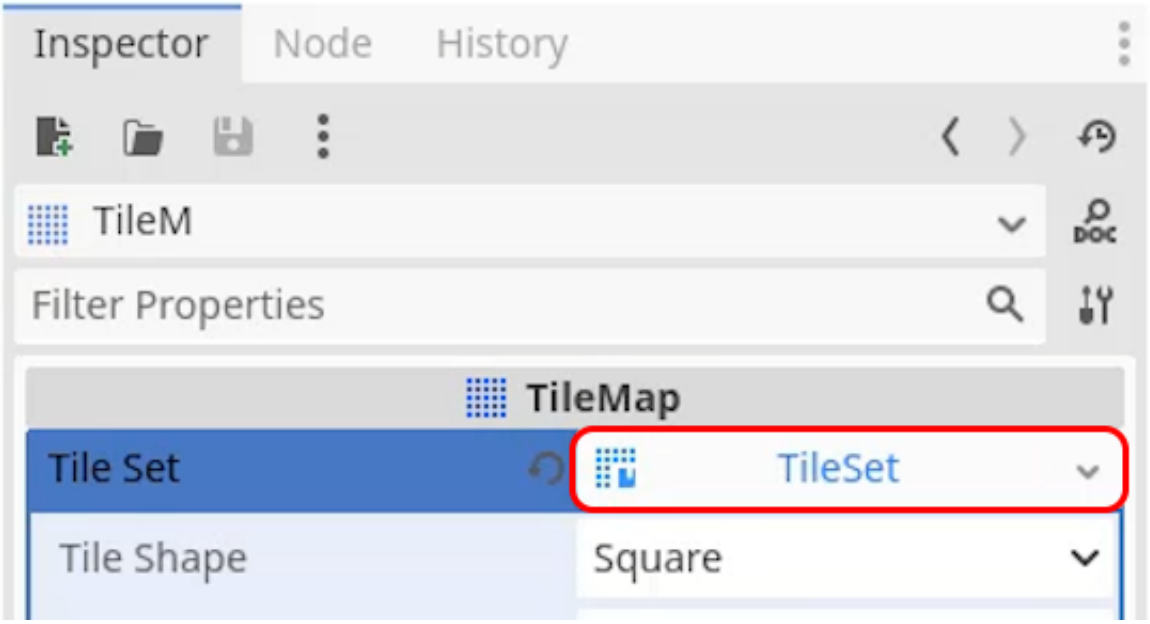


We also need a **TileSet** to draw the sprites to our *TileMap* node. This can be done in the **Tile Set** property of the *TileMap* node.

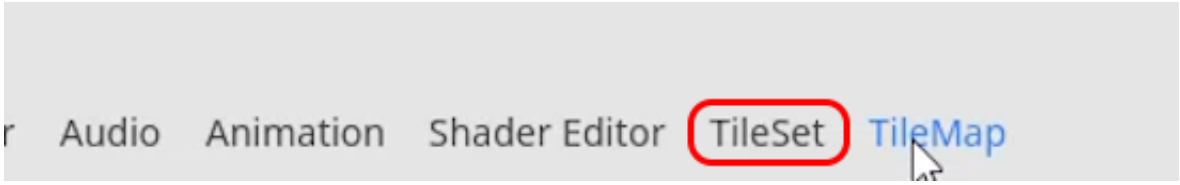


This will make an orange grid appear in the scene editor, this grid represents how big each of our sprites will be when placed on the tilemap. Before we can begin placing our tiles, we first need to tell

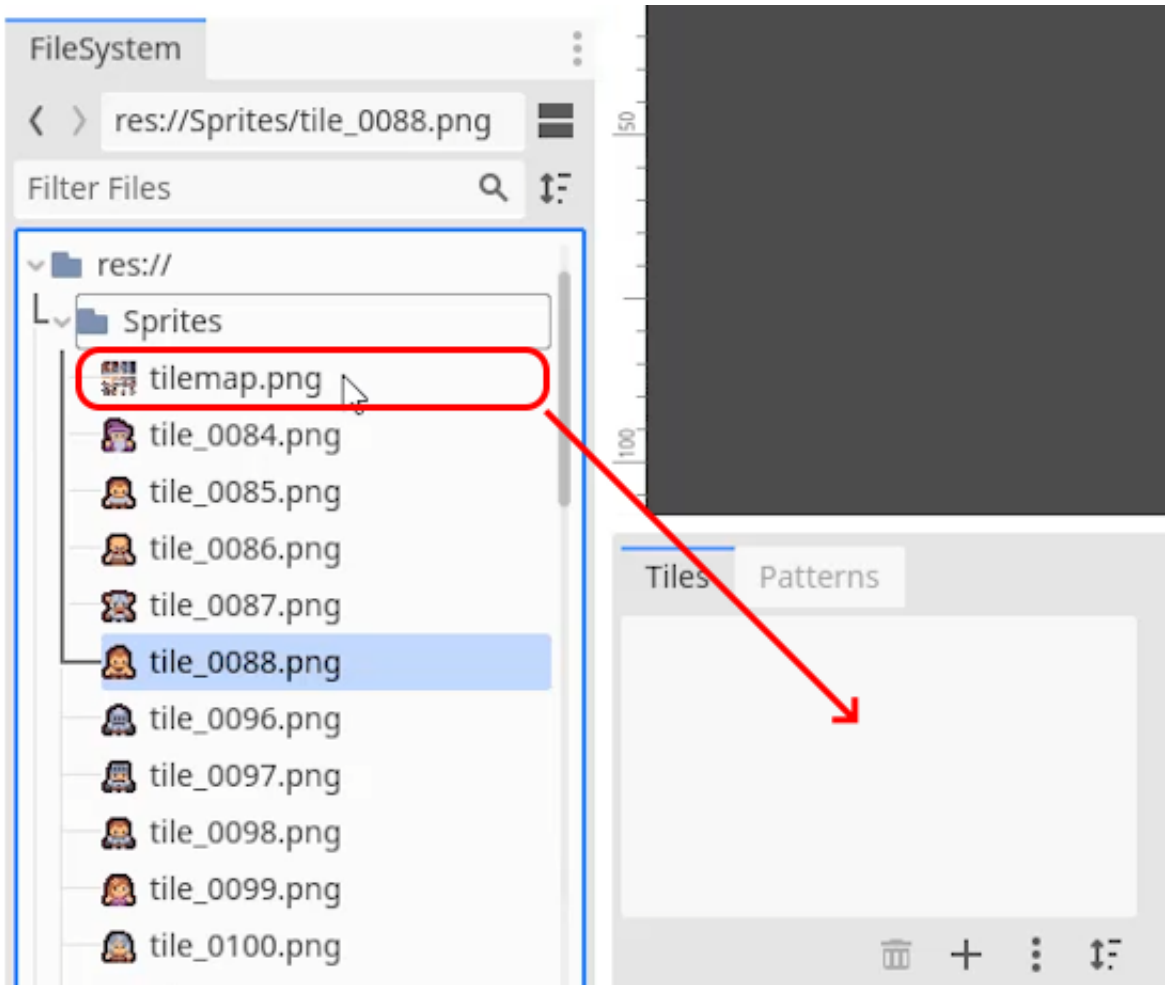
Godot what sprites to use. We will do this by opening the **TileMap** editor window, which is accessible by clicking the **TileSet** value that we just created.



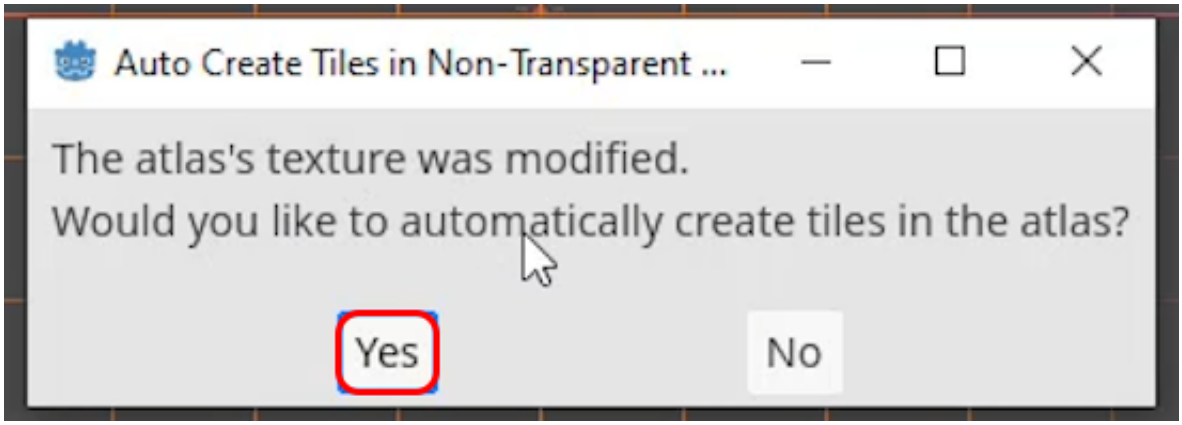
You will notice that by default the window says *"This TileMap's TileSet has no source configured. Edit the TileSet resource to add one."* This means that we need to define our tileset's source file, which can be done in the **TileSet** window.



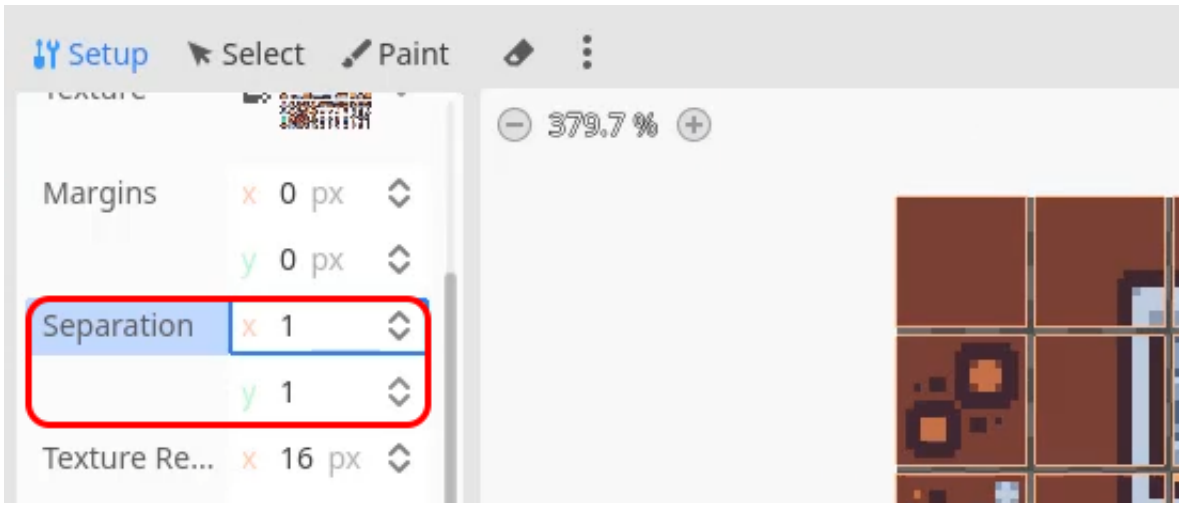
To set the tileset's source file, we will drag the **tilemap.png** file into the empty space of the **TileSet** window.



A pop-up box will appear asking if you want Godot to automatically create tiles using the sprite, we will click **yes** for this.

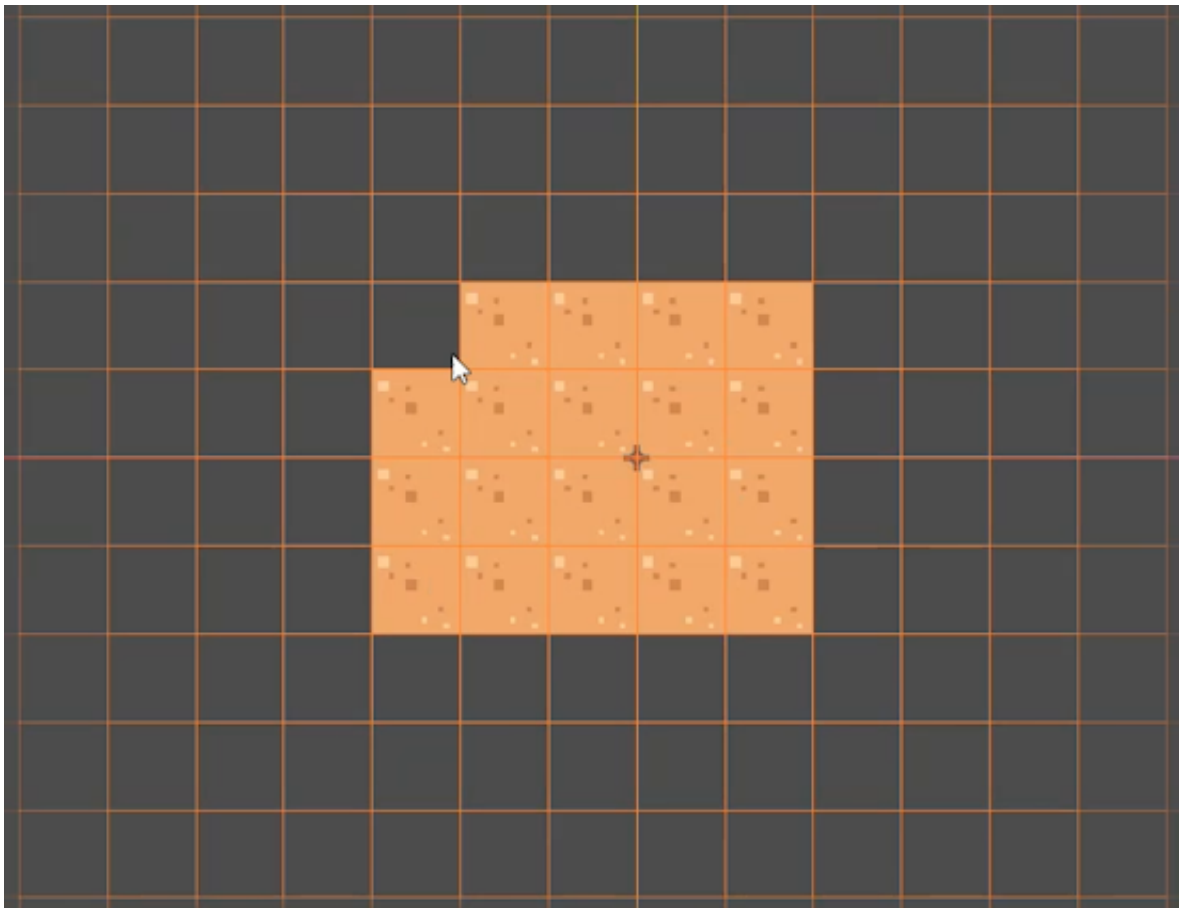


This will generate an orange grid on the sprite in the **TileSet** editor, representing the individual tiles. However, the *tilemap.png* source file contains transparent gaps between each tile sprite. To compensate for this we will adjust the **Separation** property of the tileset, as there is **1 pixel** of separation between each tile sprite. If you are using your own sprite sheet sprite, you may also need to adjust the margins, separation, or the region size as necessary.



With this setup, we can now return to the **TileMap** window to begin selecting the tiles for us to paint.

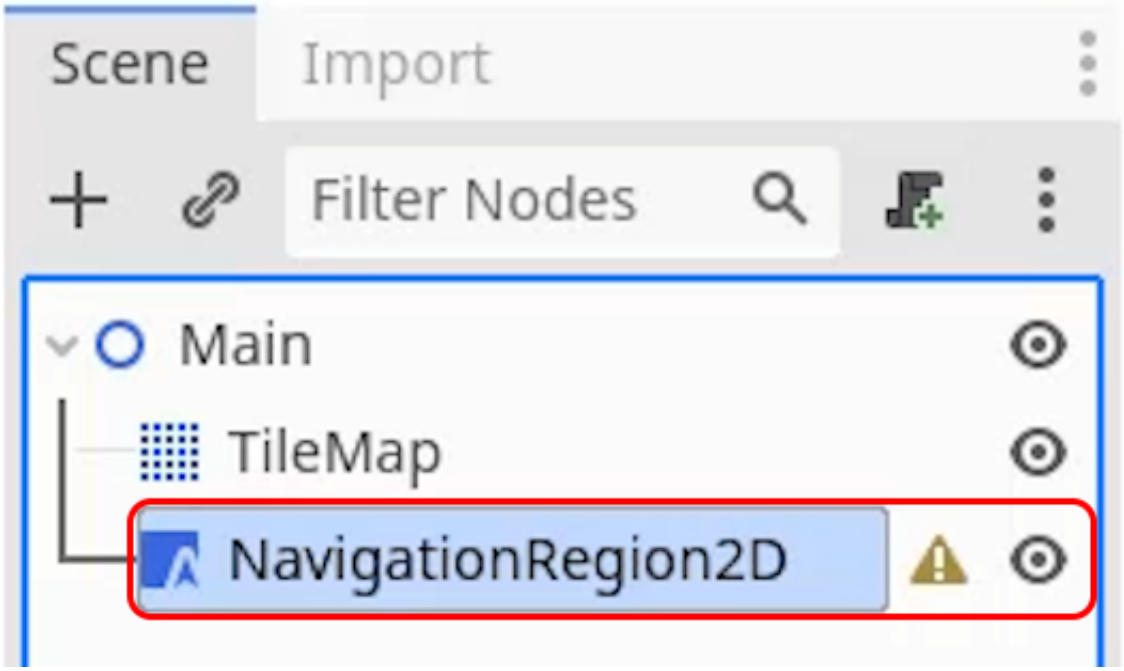
To select a tile, simply click on it in the *TileMap* window, which will show a blue border on the selected tile, as shown above. You can then click and drag in the scene editor to draw the tiles to the **TileMap** node.



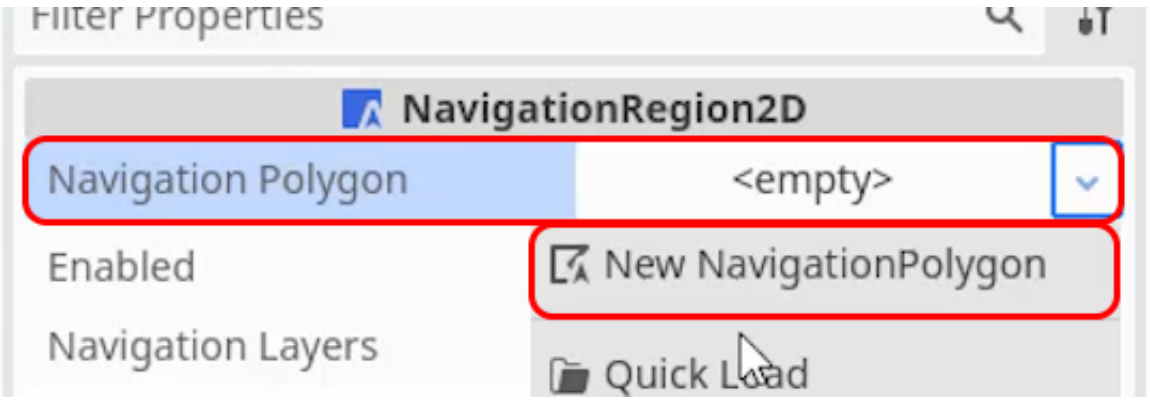
You are now all set to begin creating an environment for your game using the sprites in the tileset that we have created. You can select any of the tile sprites and place them using the *left mouse button*, and you can use the *right mouse button* to remove any tiles that you have placed but don't want. Here is the environment we have created for the course:

Adding Navigation

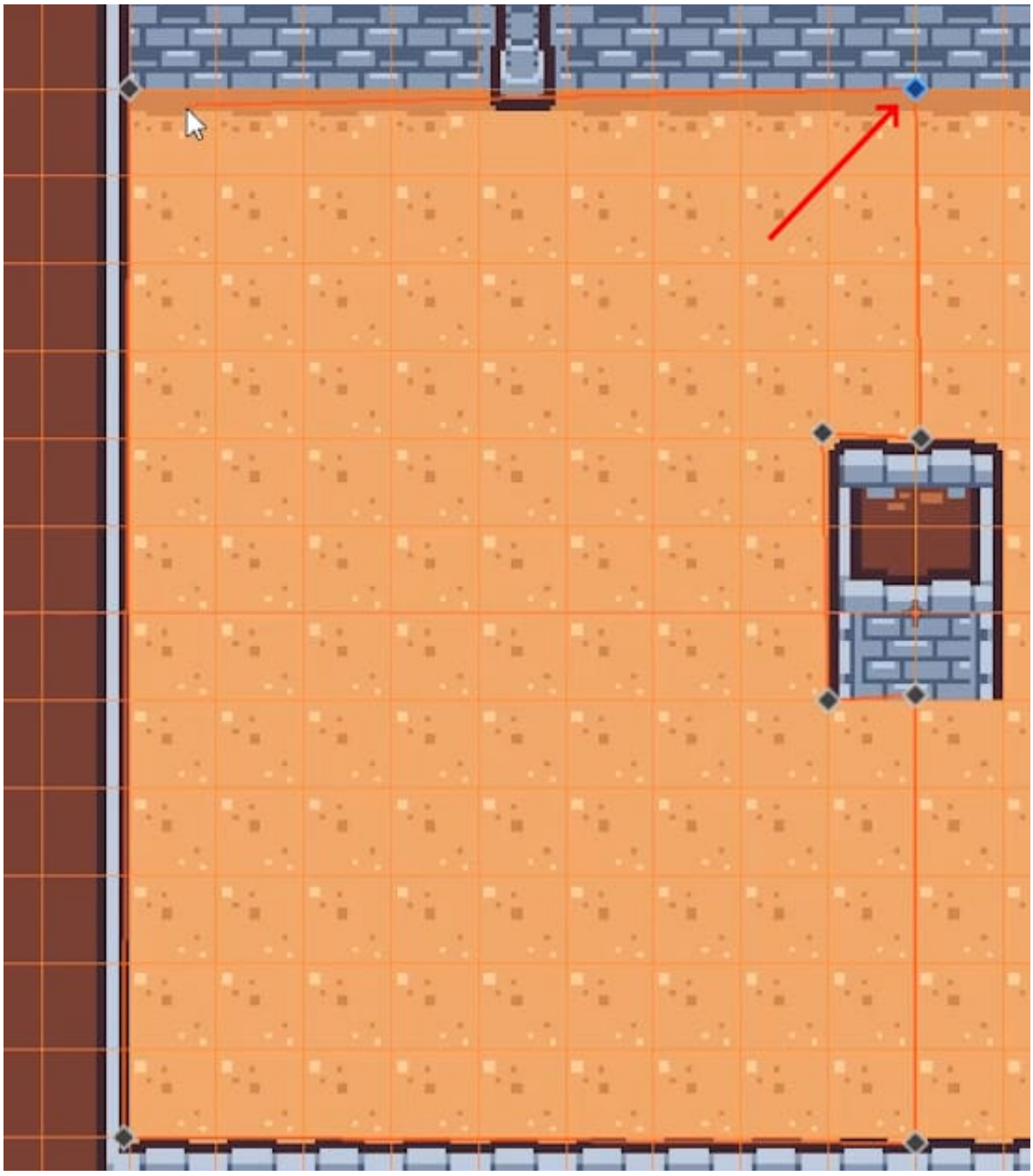
Our tileset doesn’t contain any colliders, so currently, the units would have no way of knowing what is a wall and what is the floor. To fix this we will set up a *Navigation Region*. This will allow our units to navigate around the environment without walking over the obstacles. To do this we will create a new **NavigationRegion2D** node.



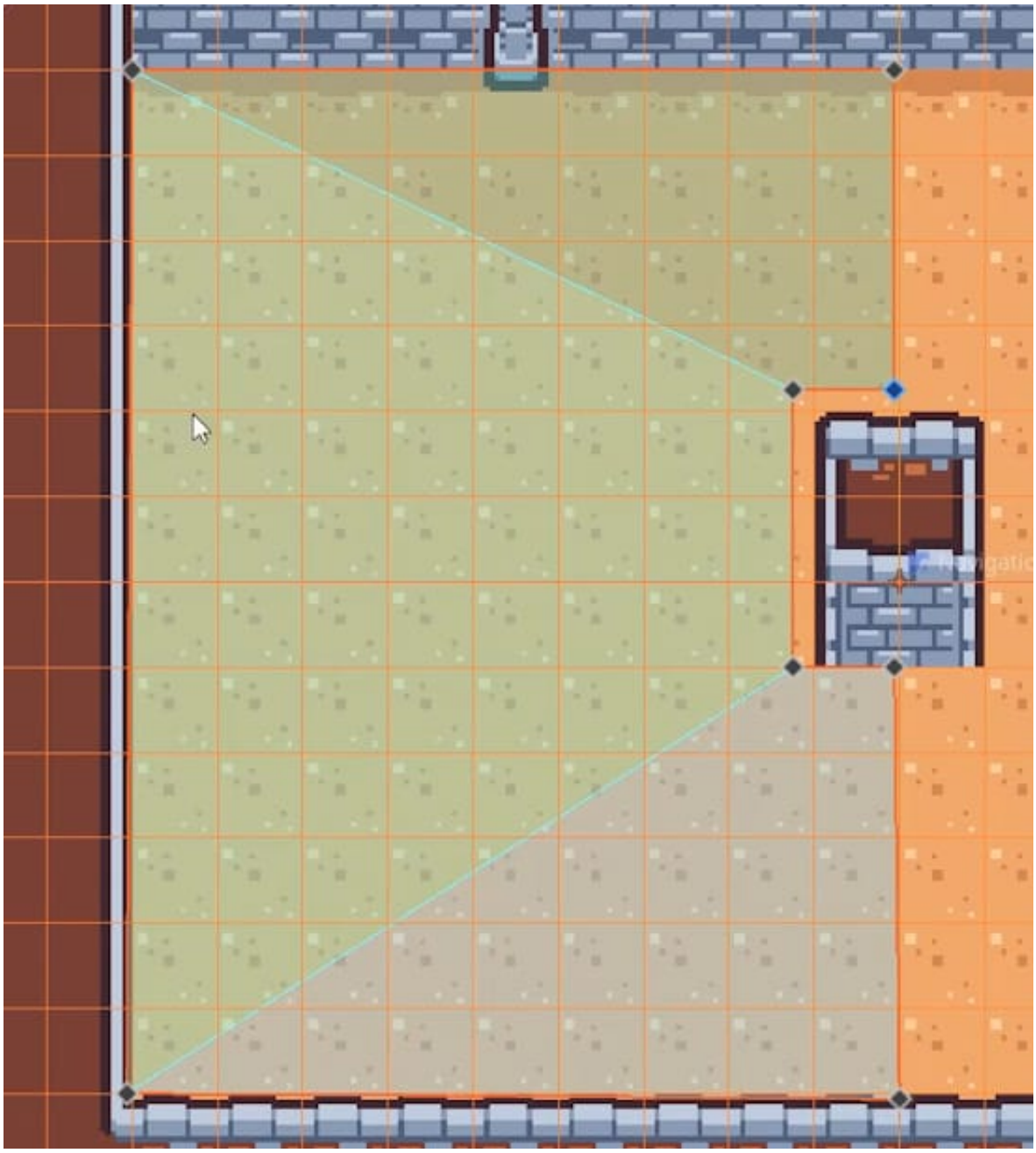
This node needs a **Navigation Polygon** property to function.



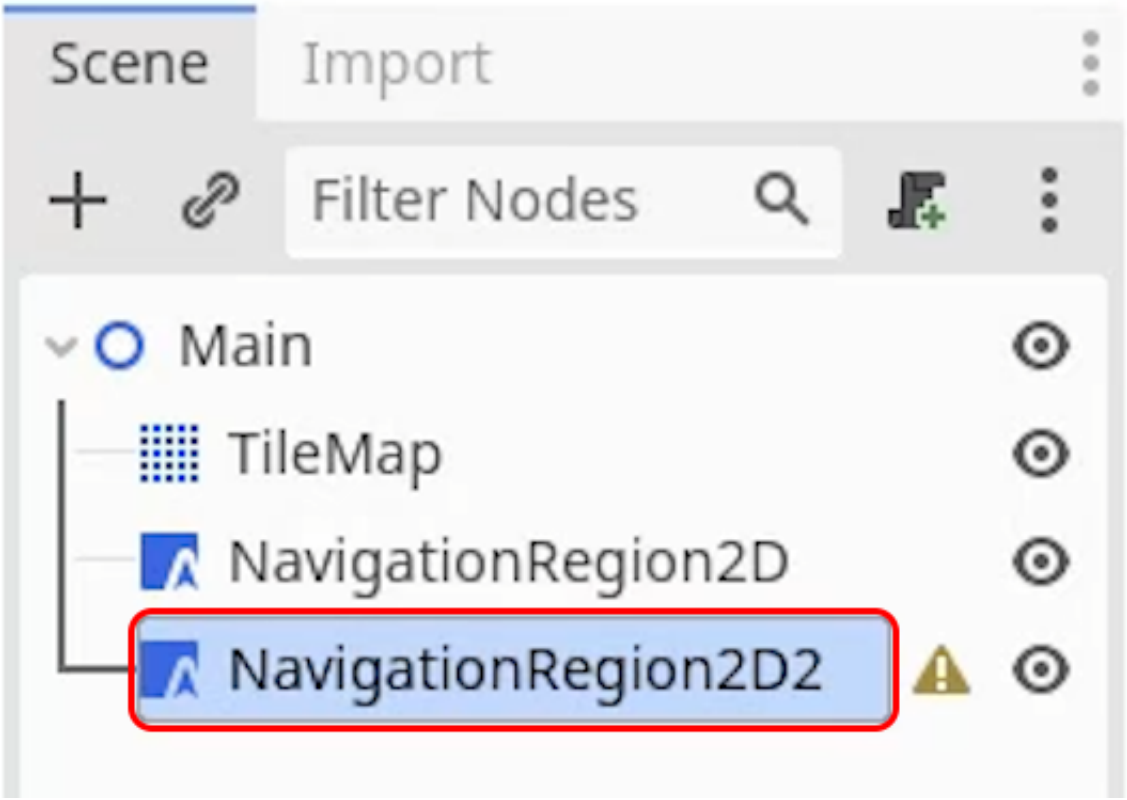
With this *NavigationPolygon* selected, we can click on the corners of the environment to create points in the scene. These points will connect to each other until you click on the first one to create a navigation region. If you place any nodes in the wrong place, you can quickly fix this by clicking and dragging them.



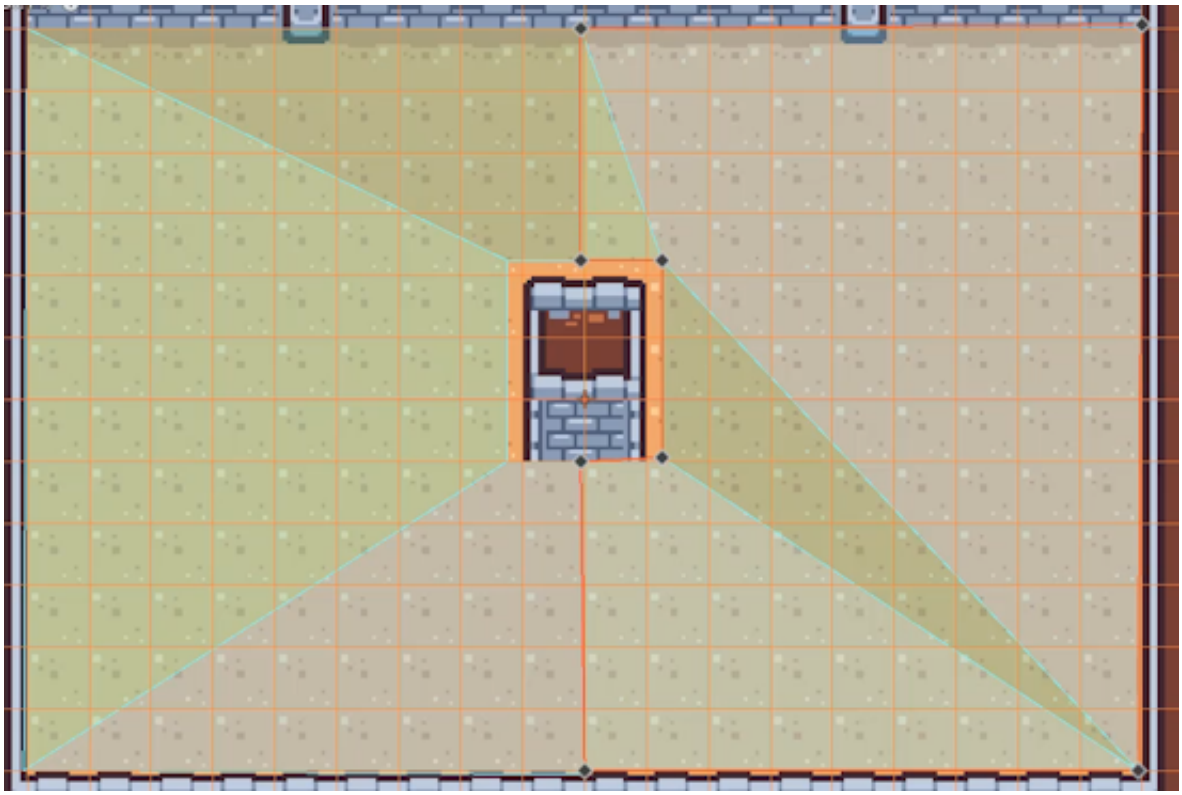
The goal is to map out the area that should be walkable for our units.



Unfortunately, as we can't cut out areas using the *NavigationRegion2D* node, we need to create a second **Navigation2DRegion** node for the second half of the environment.



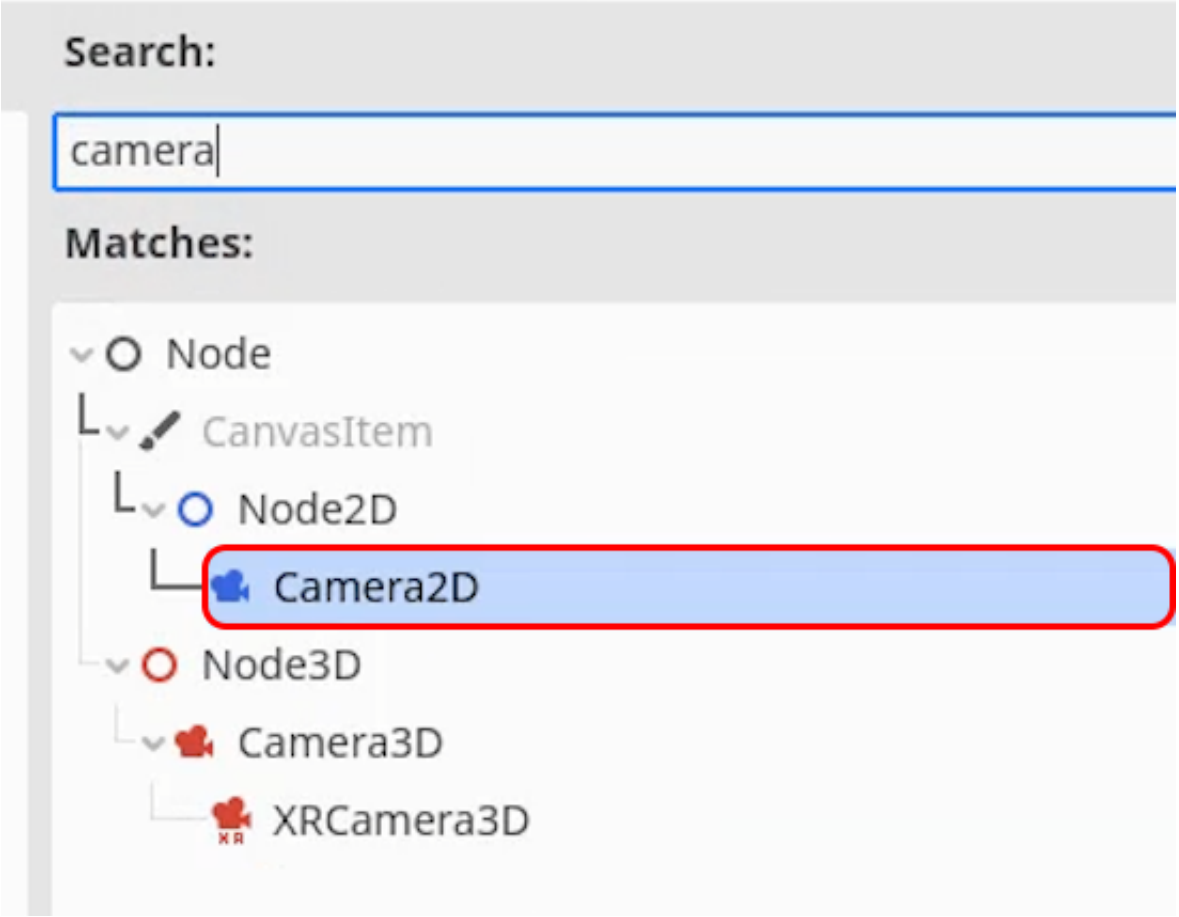
In this node, we can repeat the same navigation mapping for another region of the environment. Depending on how many obstacles are in your scene, you may need to repeat this multiple times. However, in the end, you should have all walkable sections of the environment covered by at least one *Navigation2DRegion* node.



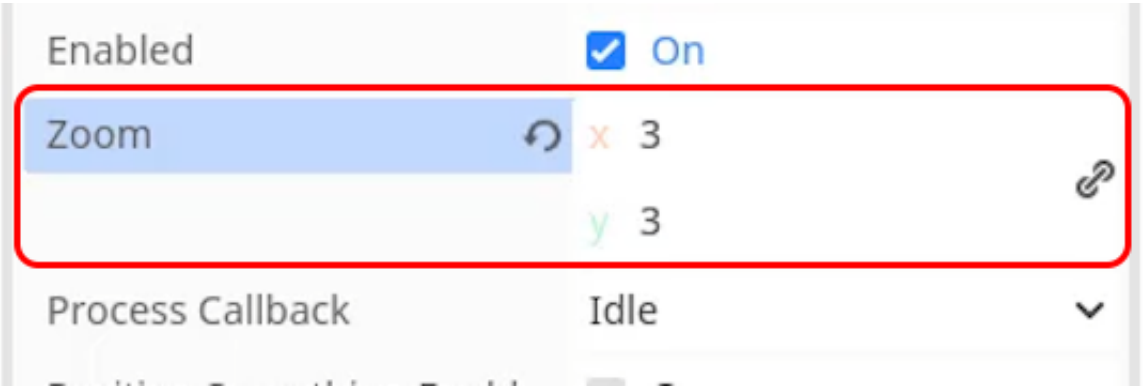
This will allow us to easily program our units to move to any point within these regions.

Adding a Camera

To finish creating our environment, we will add a **Camera2D** node to the scene, which will allow us to view the tilemap environment in the game.



If you press *play* currently, you will see the camera is quite zoomed out from the environment. To fix this, we can change the **Zoom** property to around **(3, 3)**, although you can of course try changing this yourself to make sure the environment fits the screen well.



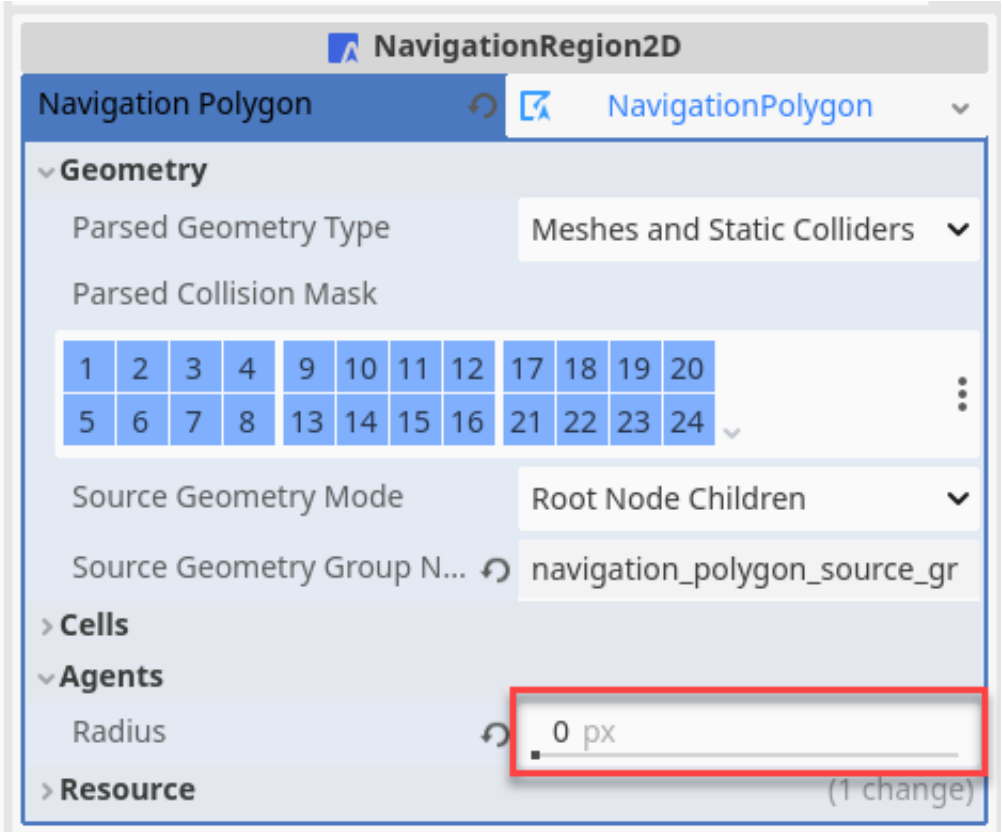
This sets us up perfectly to begin creating the units in the next lesson.

Baking the Navigation Region

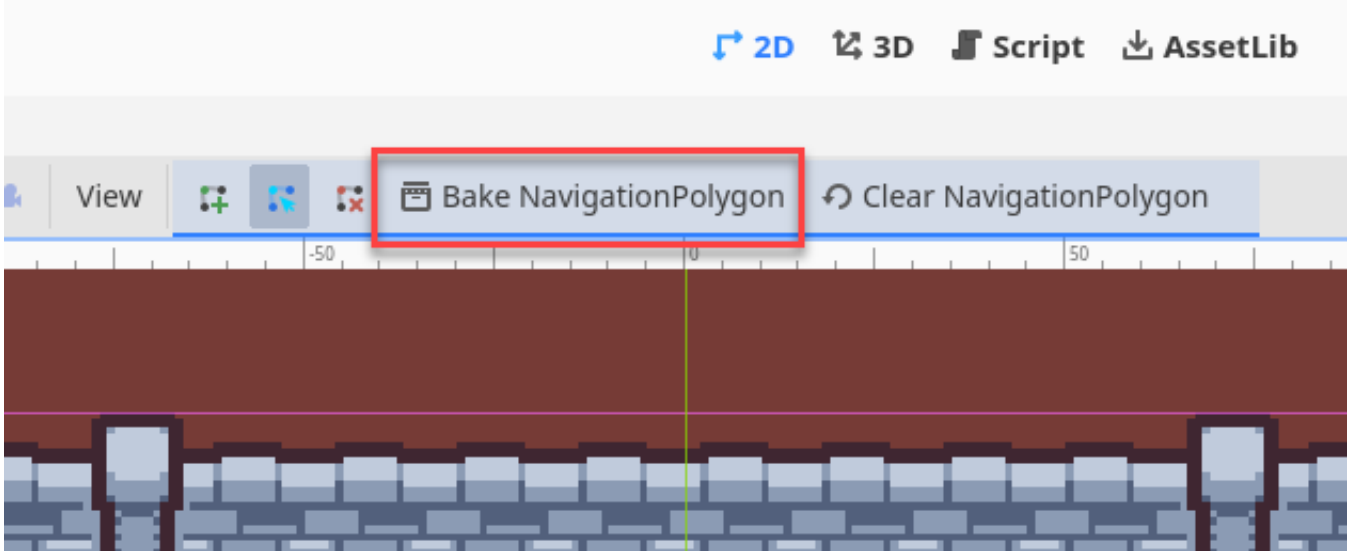
There were some changes made to how NaivgationRegions work in Godot 4.2, so let's go over how we can get our existing regions baked and ready to go.

Follow these steps for *both* of the NavigationRegion2D nodes we created:

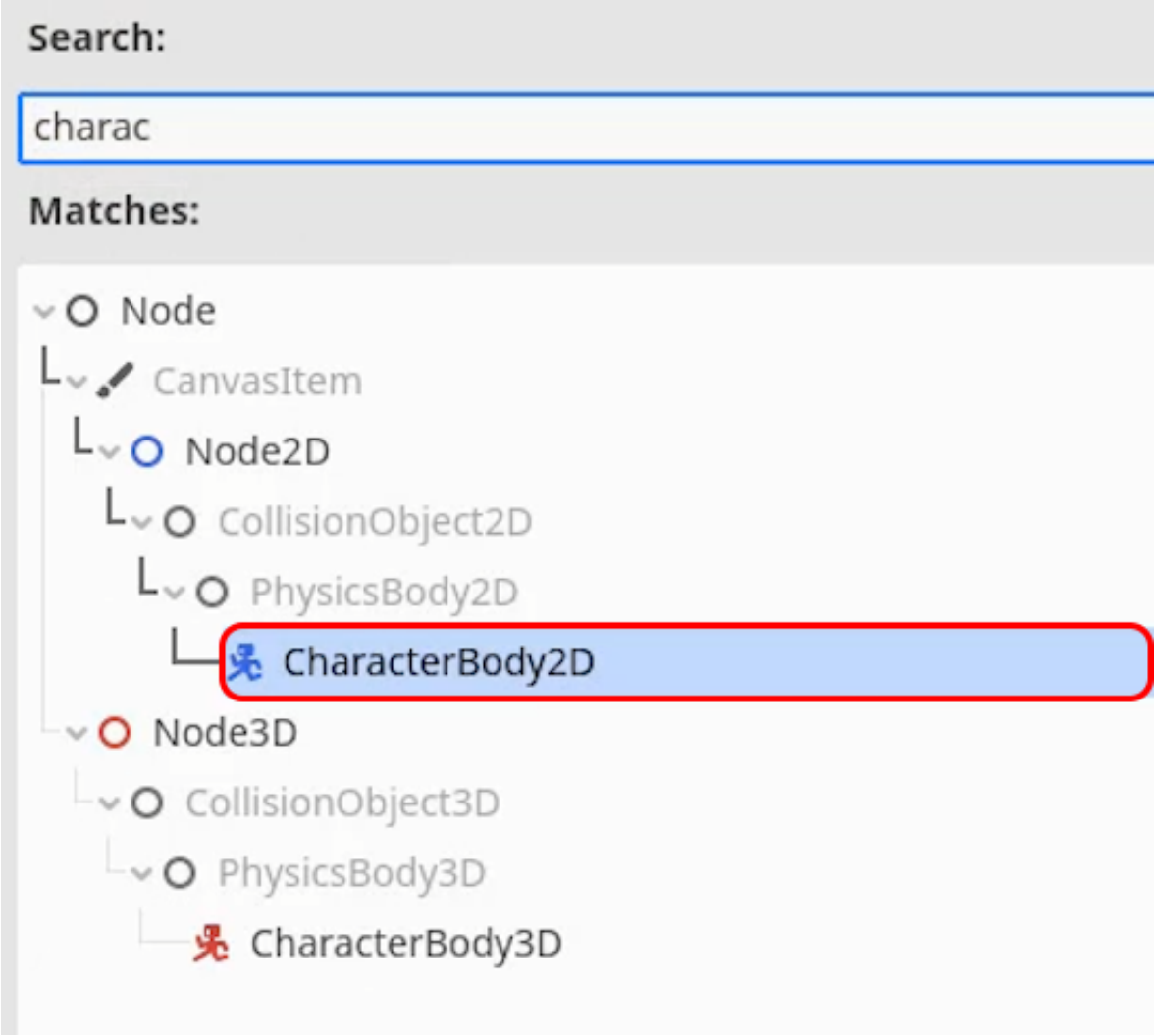
- In the Inspector, open the *Navigation Polygon* property and go down to *Agents* > *Radius* and set that to 0.



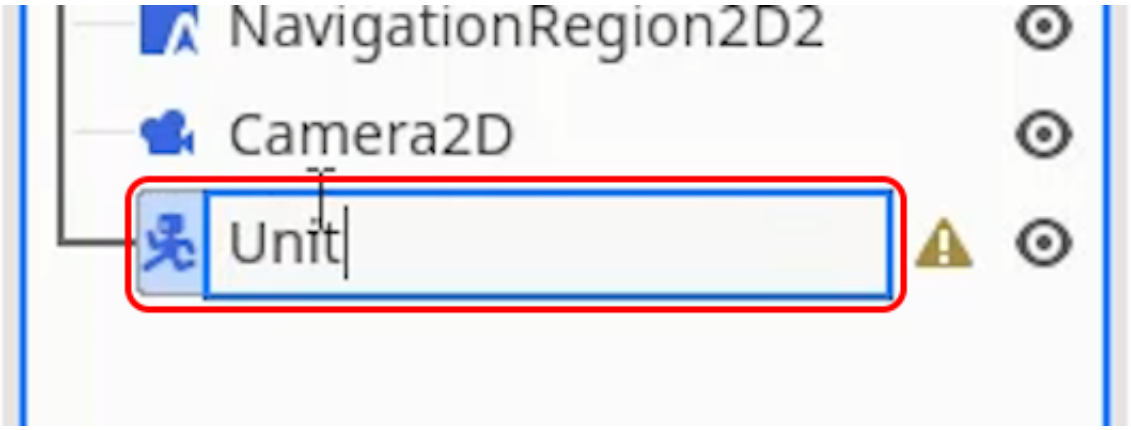
- Then towards the top of the Godot editor, you should see a button called **Bake NavigationPolygon** when you have the node selected. Click on that and you should see the navigation region appear!



In this lesson, we will be creating the scene for our *Units*. This scene will allow us to easily create many instances of both the player and enemy units across the game. These units will need to be able to move around the *Navigation Regions* that we set up previously, along with being able to attack other enemy units when close enough. The root node for our *Unit* scene will be a new **CharacterBody2D** which we can temporarily add to the *Main* scene.

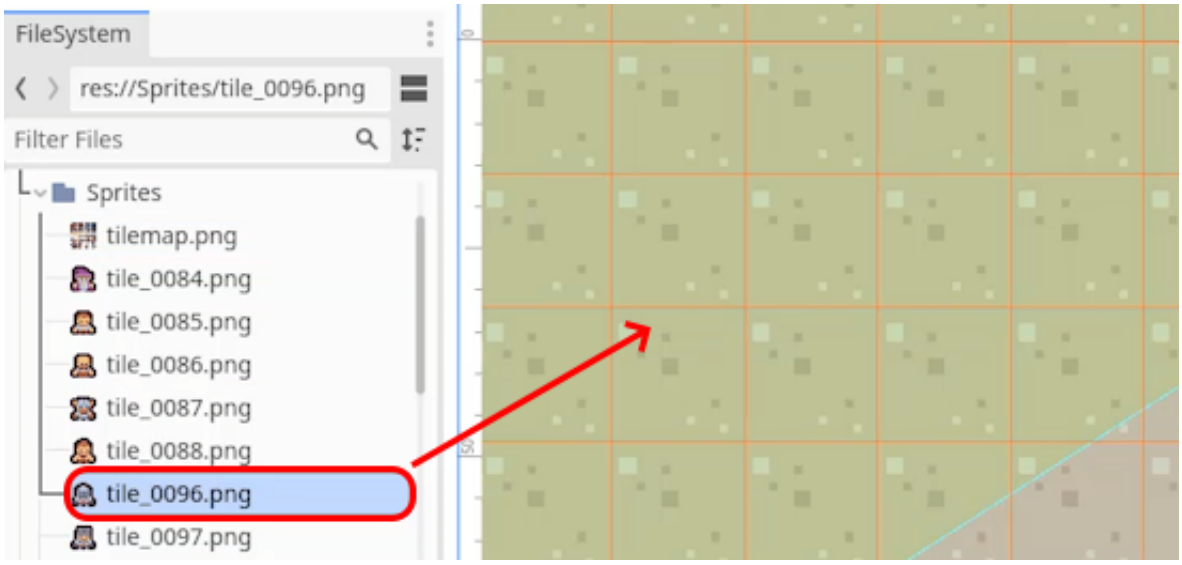


We can **rename** this node to **“Unit”**.

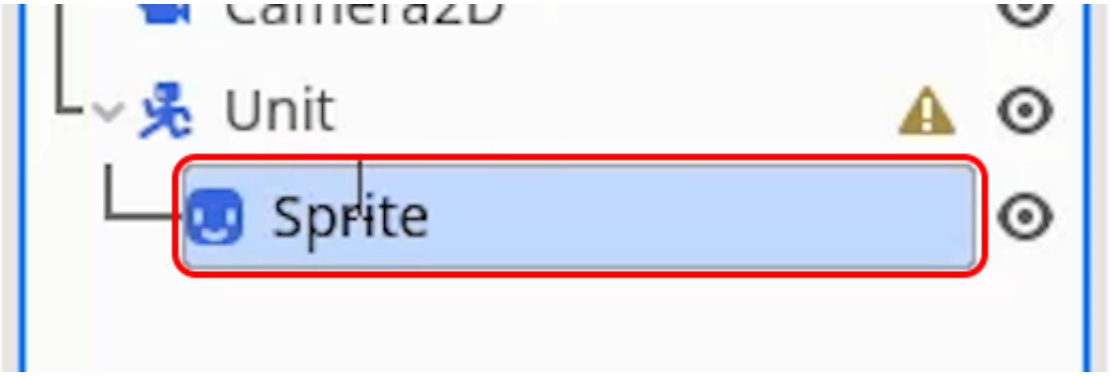


Adding a Sprite

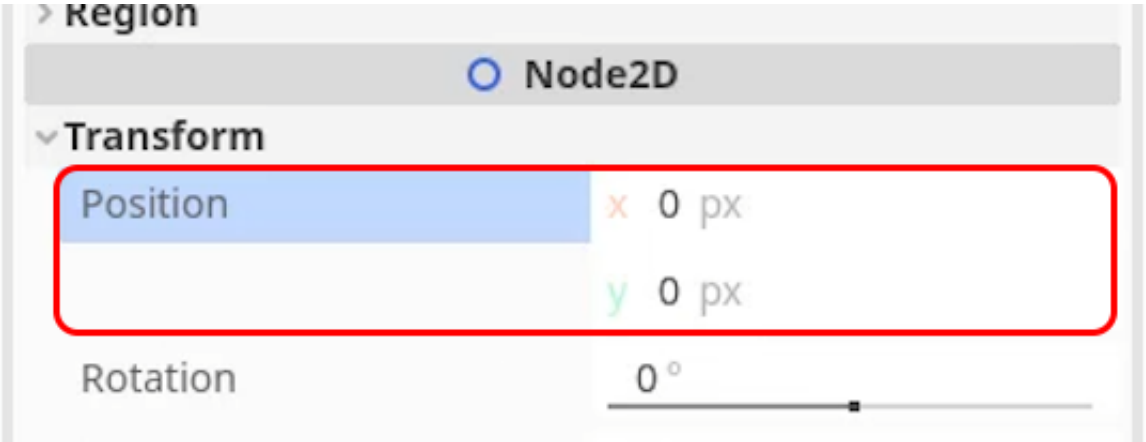
Next, drag in a sprite from the *FileSystem* to act as our *Unit's* visual representation. We will be using the *tile_0096.png* asset, although you can choose anything you want!



This sprite can be made a child of the *Unit* node, and we can **rename** it to **"Sprite"** to make it easily identifiable.



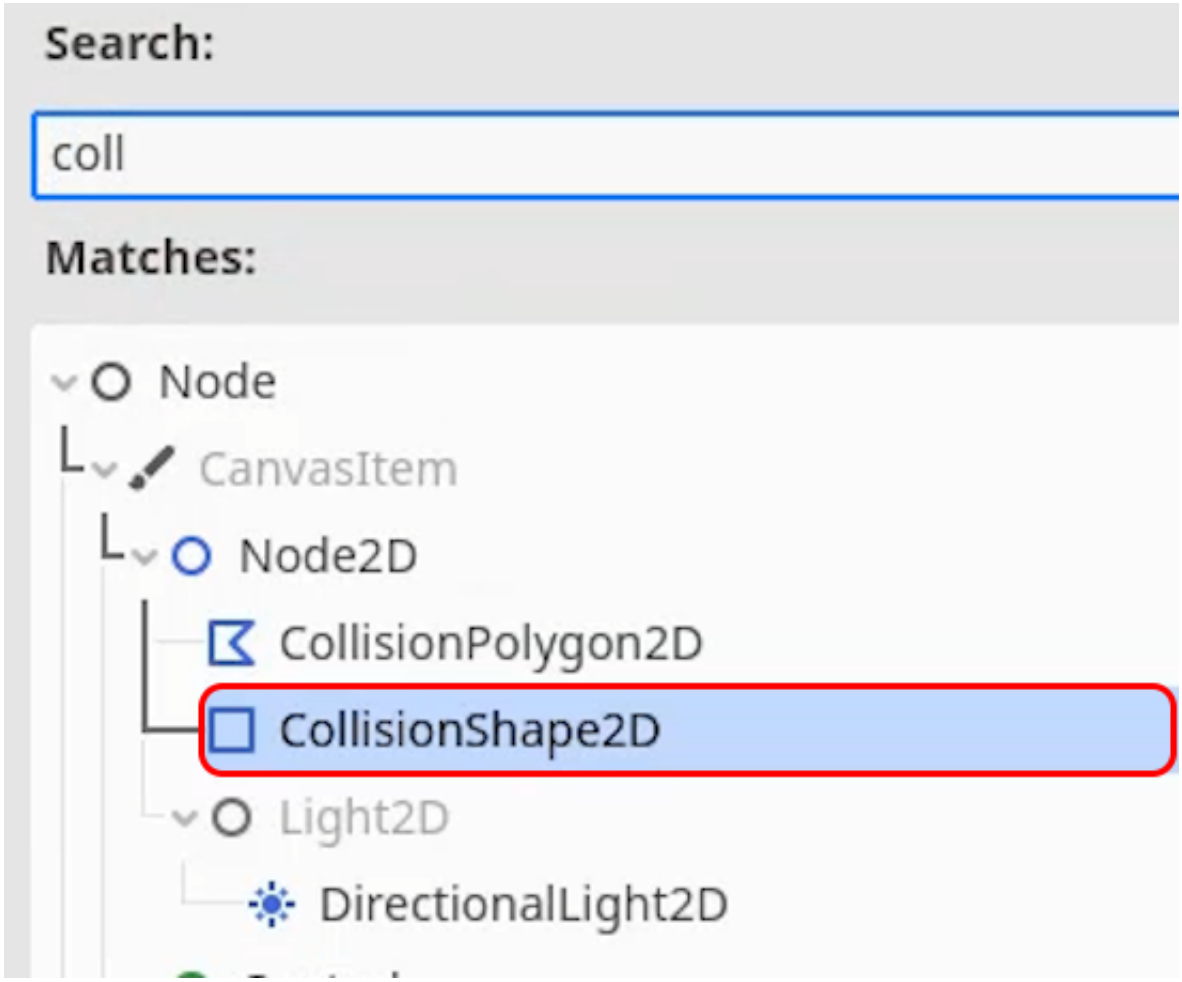
Finally, we will set the **Position** of the *Sprite* to **(0,0)** to make it centered on the *Unit* parent node.



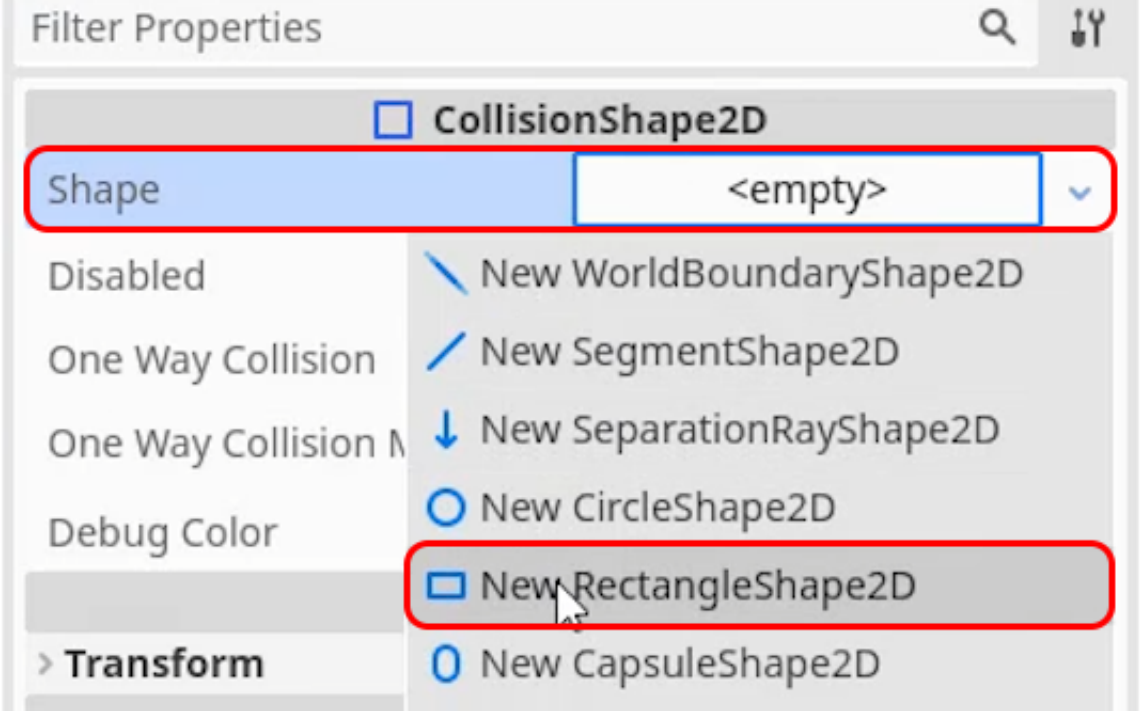
Adding a Collider

The next step is to add a collider, as needed by the *CharacterBody2D Unit* node. This will be done

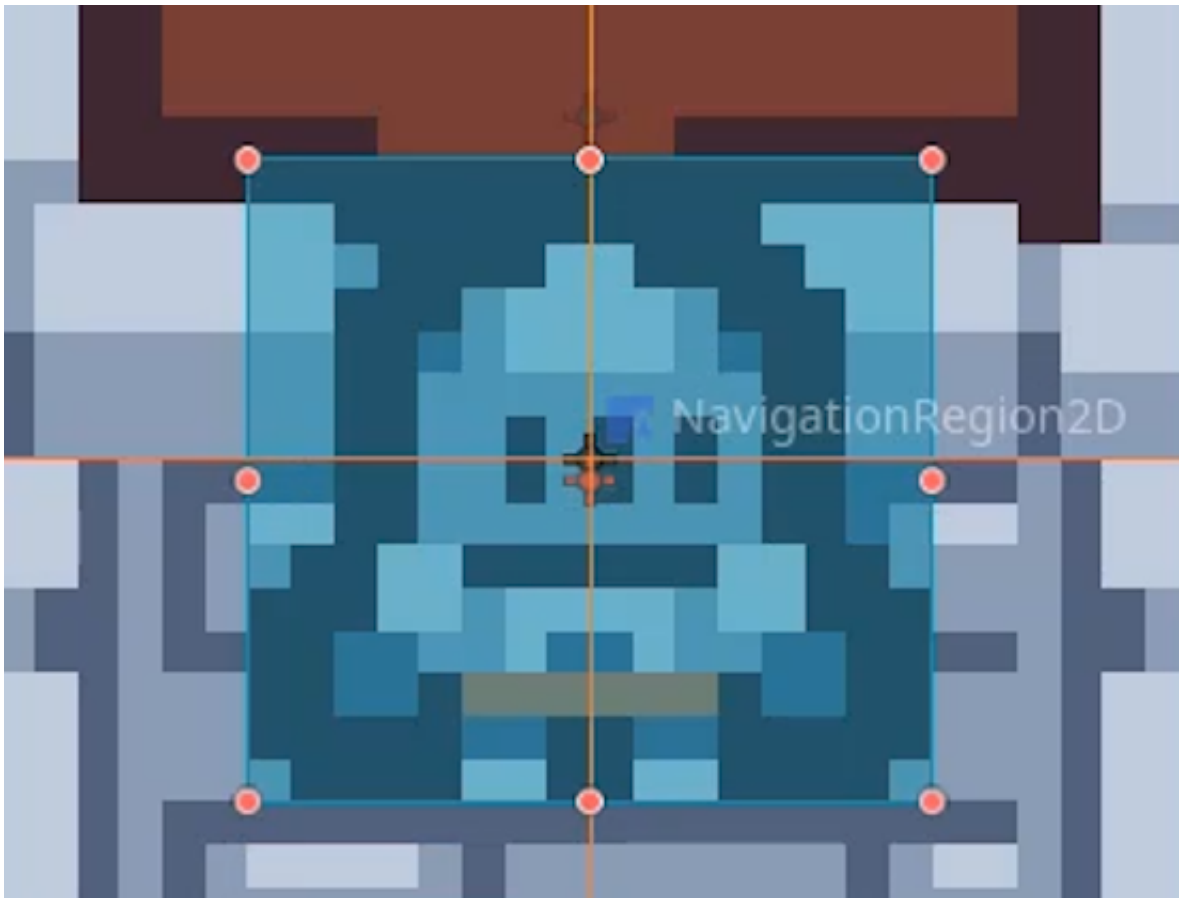
using a new **CollisionShape2D** child node.



We need to assign a **Shape** property to this collision shape. For this sprite, we will be using a **New RectangleShape2D**.



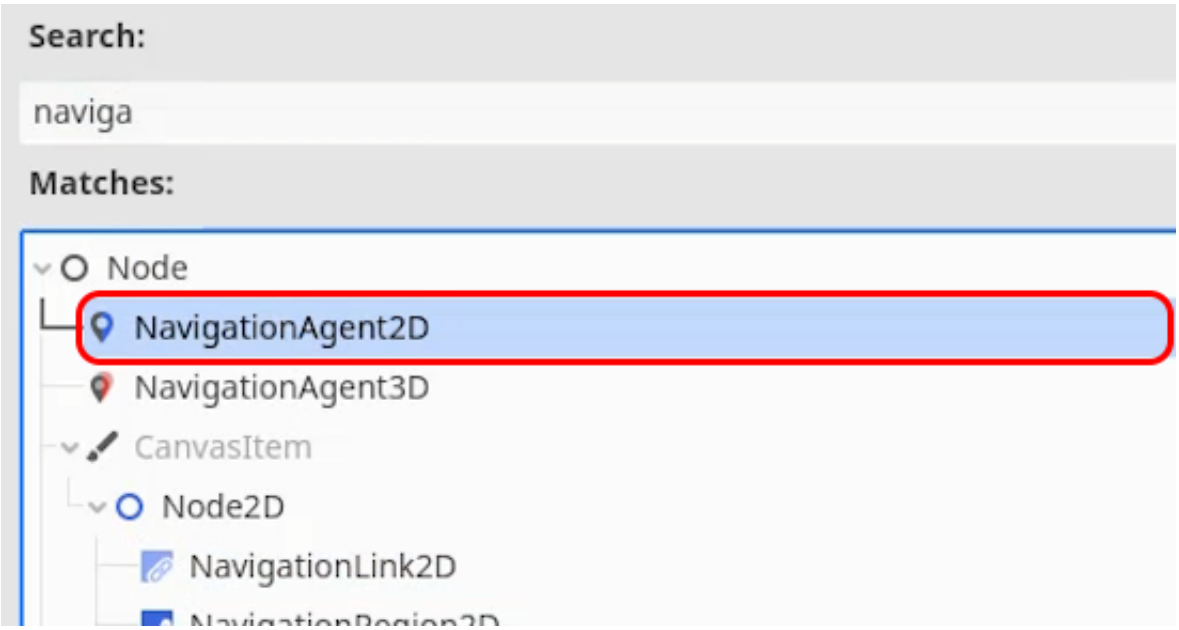
This collision shape then needs to be resized to fit the *Sprite* node.



Using Navigations

A key feature of our *Unit* scene is its ability to move around the *Navigation Regions* that we have set

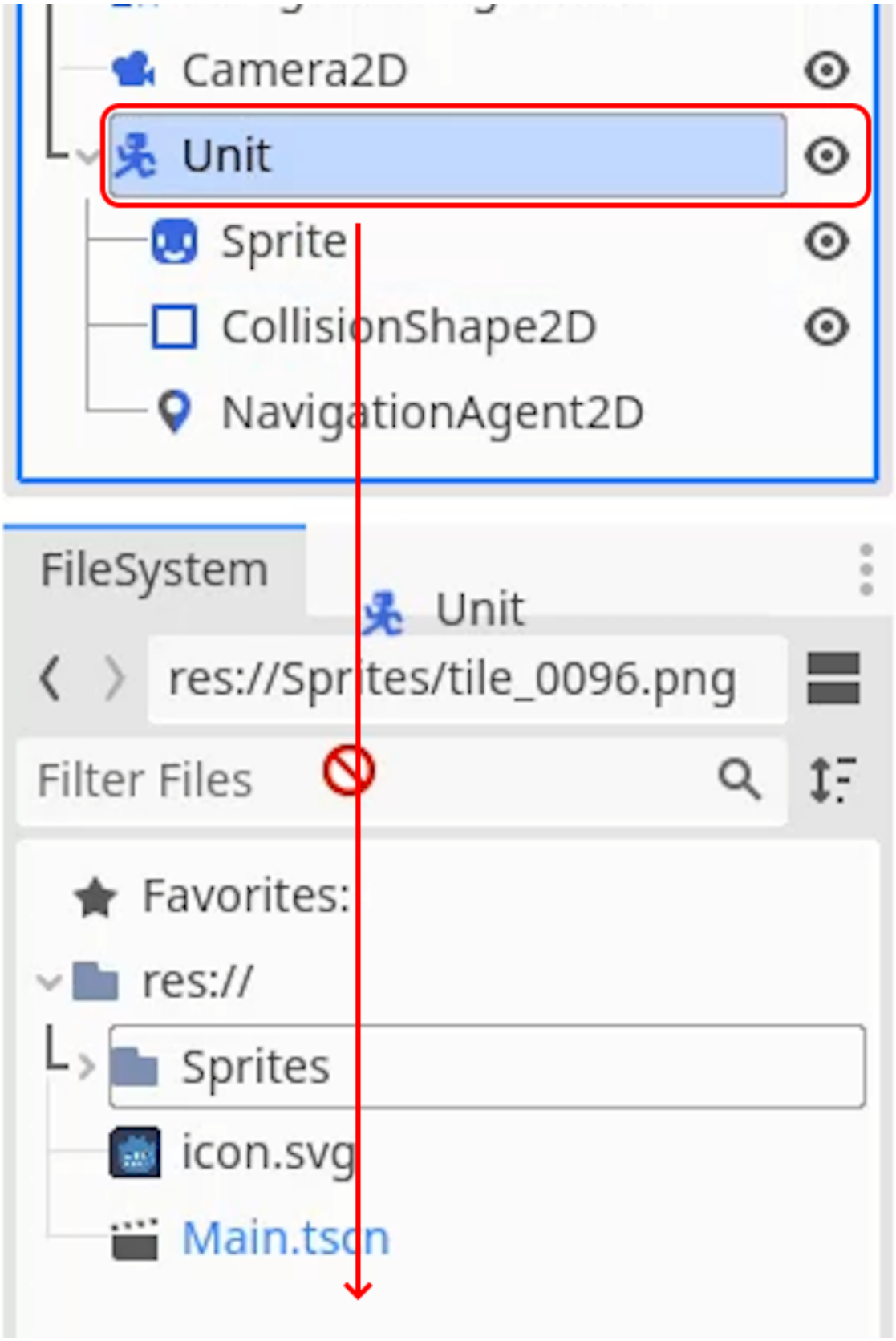
up on the environment. To make use of these we need to add a **NavigationAgent2D** node to the scene.



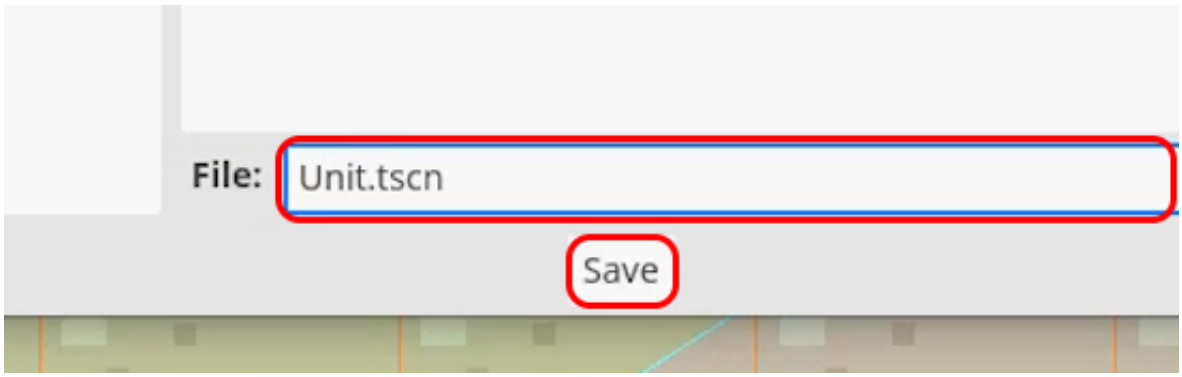
This node will allow us to easily calculate a path between the unit and a point on the tilemap when we implement the *Unit's* script.

Turning the Unit Node Tree into a Scene

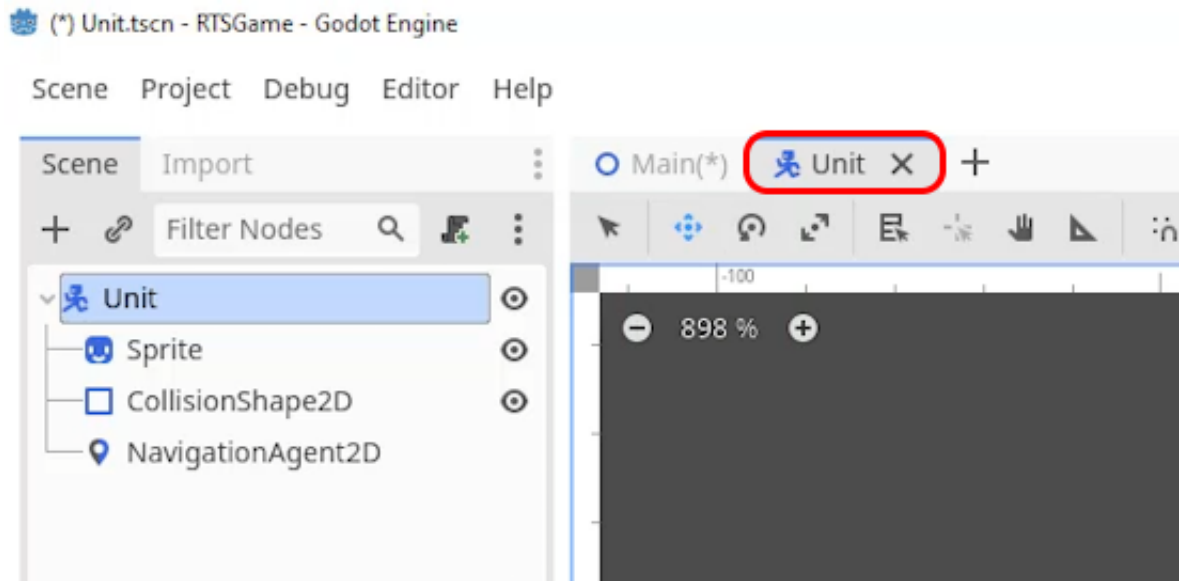
Later down the line we may add more nodes to handle unit selection or for other functionality, but for now this will be a good starting point for our *Unit*. This means we are ready to drag the **Unit** parent node into the **FileSystem** to turn the node tree into it's own scene.



This can be saved as *Unit.tscn*.

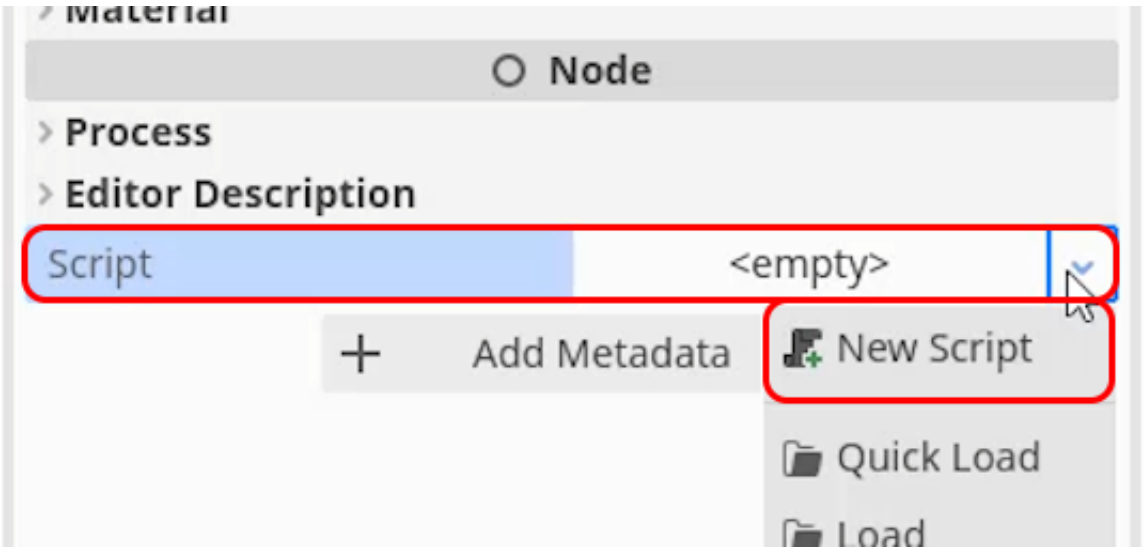


We can now **double-click** the **Unit.tscn** file to open it in the scene editor. This will allow us to edit every instance of the *Unit* scene across the game, without needing to edit them individually.

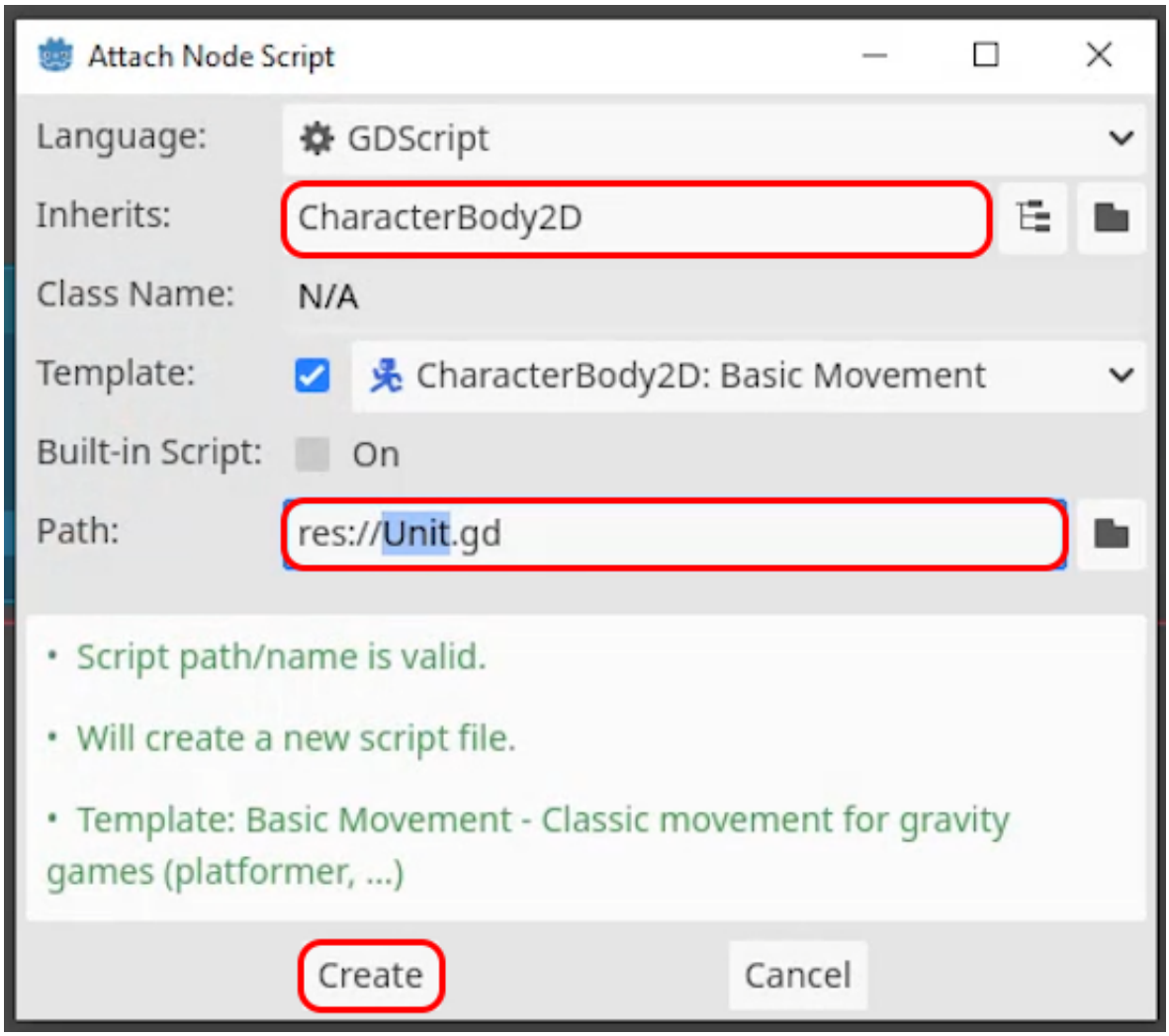


Adding the Script

In the next lesson, we will begin implementing the *Unit* scene's functionality by adding code to a script. To prepare for this, we can add a **new script** to the **Unit** parent node of the *Unit* scene.



This script can be called *“Unit.gd”* and should inherit from *CharacterBody2D*.



In this lesson, we will be adding functionality to the *Unit* script that we created in the previous lesson. Before writing any code, we first need to remove the script Godot automatically generated. This code is useful to get you quickly started when making a game like a platformer, but for our purposes, it won't be very helpful. So we will remove all but the first line. This should leave your script with just:

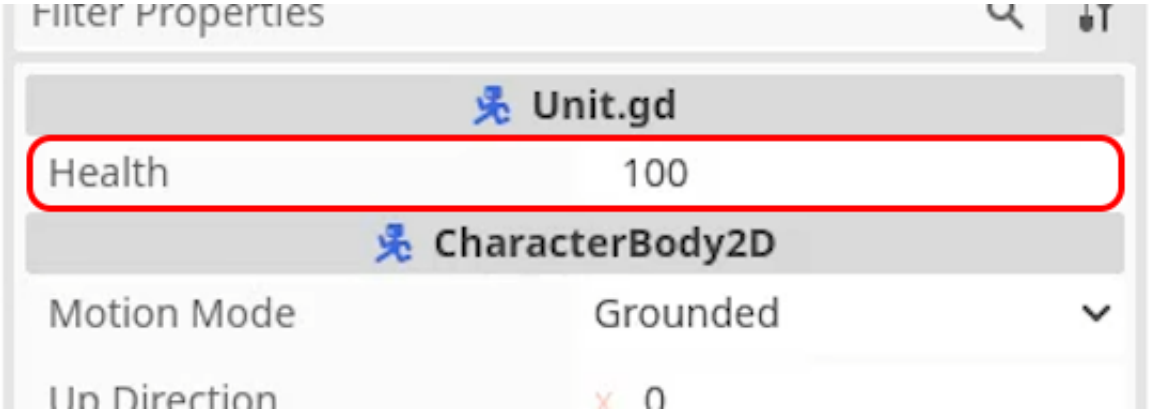
```
extends CharacterBody2D
```

Adding Variables

The first step for our script will be to add a few variables that will be used across the *Unit's* functionality. Our first *Unit* will be called **health** and be of an **int** type. We can give this a default value of **100**. We may want to change this in the *Inspector* window, so to make it editable, we will use the **export** tag.

```
@export var health : int = 100
```

The *export* tag will make our variable editable in the *Inspector* window when the *Unit* parent node is selected.



The next variable is going to be the **damage** variable, this will also be an **int** with a default value of **20**. This variable could also be editable per unit, so we will add the **export** tag to this as well. This variable will, of course, represent the amount of damage our *Unit* instance will do when attacking another *Unit*.

```
@export damage : int = 20
```

The *Unit* will also need a variable called **move_speed**, of type **float**, and a default value of **50.0**. This variable will be **exported**, and here the value represents the number of pixels per second the *Unit* will move across the screen. By default, this is 50 pixels per second.

```
@export var move_speed : float = 50.0
```

The next variable will define the *Unit's* **attack_range**. This will also be **exported**, with a type

of **float**, and a default value of **20.0**. The attack range will define how close to an opponent this *Unit* instance needs to be before it begins attacking the target.

```
@export var attack_range : float = 20.0
```

We don't want the *Units* to attack every frame, so we will define an **attack_rate** variable to limit the time to wait between attacks. This will also be **exported**, have a type of **float**, and a default value of **0.5**.

```
@export var attack_rate : float = 0.5
```

To track the time since the last attack, we will use another variable called **last_attack_time** with a type of **float**. This won't be exported or given a default value, as this variable will only be assigned from the functions in the script and used to keep track of attacks.

```
var last_attack_time : float
```

The next variable will be called **target** and be of type **CharacterBody2D**. This will keep track of an opponent *Unit* instance that this *Unit* is chasing and will attack.

```
var target : CharacterBody2D
```

Additionally, we need a **NavigationAgent2D** variable called **agent** to keep track of the *Unit's NavigationAgent2D* child node. We will need a similar variable called **sprite** which will keep track of the **Sprite2D** node of the *Unit*.

```
var agent : NavigationAgent2D
var sprite : Sprite2D
```

The final variable will keep track of whether the player owns the current *Unit* instance or not. This will be called **is_player**, be of type **bool,** and be **exported** so that it is accessible from the *Inspector* window.

```
@export var is_player : bool
```

Adding Functions

With the variables set up, we are ready to begin defining the functions that will make our *Unit* script operate. The first function will be the **_ready** function, that Godot supplies by default, and will automatically run when the scene is first initialized.

```
func _ready():
```


Because this function runs once at the start, it is a good place to initialize our node tracking variables. To do this we will make use of the **$** character, which allows us to find a child-node with a given name.

```
func _ready():
    agent = $NavigationAgent2D
    sprite = $Sprite2D
```

The next function will be the **take_damage** function which will apply a **damage_to_take** property to our *health* variable.

```
func take_damage (damage_to_take):
    health -= damage_to_take
```

In this function we can also check if the health has reached zero, and if so, call the **queue_free** function, which will delete the *Unit* instance.

```
func take_damage(damage_to_take):
    ...
    if health <= 0:
        queue_free()
```

To complement the *take_damage* function, we also want a function that can handle attacks. This will be called **_try_attack_target** and will be called every frame when we are in range of our *target* unit.

```
func _try_attack_target():
    var cur_time = Time.get_unix_time_from_system()
    if cur_time - last_attack_time > attack_rate:
```

The code we have added above is quite large, so we can break it down into sections.

1. We first created a function called **_try_attack_target**.
2. We then get the *unix_time* using the **get_unix_time_from_system** function and save this to a new variable called **cur_time**. The *unix_time* is a universal time value that constantly goes up.
3. We can then compare the *cur_time* variable against our **last_attack_time** variable to see if more time than our **attack_rate** has passed.

If more time than the *attack_rate* has passed, we can call the **take_damage** function on our target, along with updating the **last_attack_time** variable to the current time.

```
func _try_attack_target():
    ...
    if cur_time - last_attack_time > attack_range:
        target.take_damage(damage)
```



```
        last_attack_time = cur_time
```

Currently, our target variable is always empty. To fix this, we will implement a new function called **set_target** that changes the **target** variable to a property called **new_target**.

```
func set_target(new_target):
    target = new_target
```

Additionally, our *Units* need a way to move between points. This can be placed in its own function called **move_to_location** which will have a property of **location**. The first thing this function will do is set the **target** variable to null.

```
func move_to_location(location):
    target = null
```

We can then update the *target_position* property of our **agent** variable to calculate a path using the *NavigationAgent2D*. This won't move our *Unit* instance, but it will calculate the path to get from the current position to the destination.

```
func move_to_location(location):
    ...
    agent.target_position = location
```

That covers everything we will set up in this lesson, in the next lesson we will continue expanding the functionality of this script.

In this lesson, we will continue working on implementing the functions required by this script. The first one to add will be called **_target_check** which will check to see if the target is in range of the current *Unit* instance.

```
func _target_check():
```

The first thing to check in this function is whether our target variable has a value. If it doesn't, we can't check against it.

```
func _target_check():
    if target != null:
```

From here, we can compare the positions of our target and the *Unit's* **global_position**. We will store this in a variable and then compare it against the **attack_range** variable, which we created in the previous lesson.

```
func _target_check():
    if target != null:
        var dist = global_position.distance_to(target.global_position)
        if dist <= attack_range:
```

The first thing we want to do, when the target is in range, is to stop our *Unit* from moving, by updating the **agent's target_position** property to the current position. We can then call the **_try_attack_target** function to begin the attacking loop.

```
func _target_check():
    if target != null:
        ...
        if dist <= attack_range:
            agent.target_position = global_position
            _try_attack_target()
```

Finally, if the target is not within the *attack_range,* the *Unit* needs to continue chasing the *target*. To do this, we will update the **agent's target_position** to the target's *global_position*.

```
func _target_check():
    if target != null:
        ...
        if dist <= attack_range:
            ...
        else:
            agent.target_position = target.global_position
```

We will be calling this function from the **_process** function. The process function is supplied by Godot and is automatically called once per frame, which is perfect for calling our

_target_check function.

```
func _process(delta):
    _target_check()
```

Moving our Unit

With this in place, most of our *Unit* script is complete. The final thing to do is make our *Unit* move based on the *NavigationAgent2D's* calculated path. This will be done in the **_physics_process** function, which is also supplied by Godot, but instead of running once per frame, *_physics_process* runs a fixed number of times per second. We will return out of the *_physics_process* function if the agent has finished its navigation, as we don't want to move when there is no path.

```
func _phyiscs_process(delta):
    if agent.is_navigation_finished():
        return
```

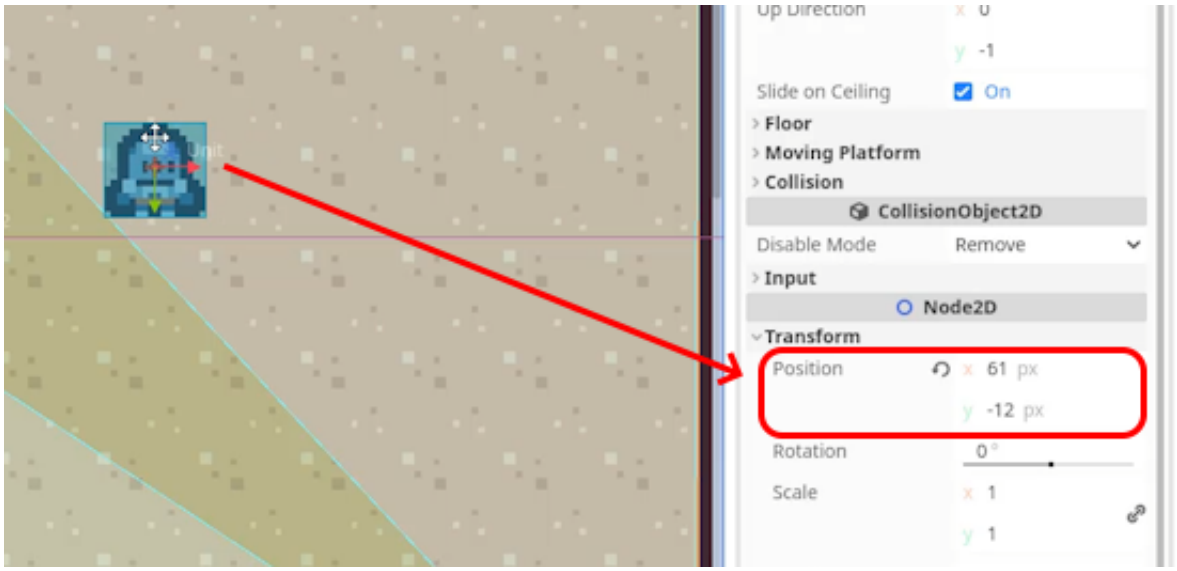
Our *NavigationAgent2D* calculates a list of points to get to the destination we have given, this is known as the path. To move to these points, we first need to calculate a direction to the next point in the path.

```
func _physics_process(delta):
    ...
    var direction = global_position.direction_to(agent.get_next_path_position())
```

With the direction calculated, we can assign the **velocity** of our *CharacterBody2D* node and call the **move_and_slide** function to move the *Unit*.

```
func _physics_process(delta):
    ...
    velocity = direction * move_speed
    move_and_slide()
```

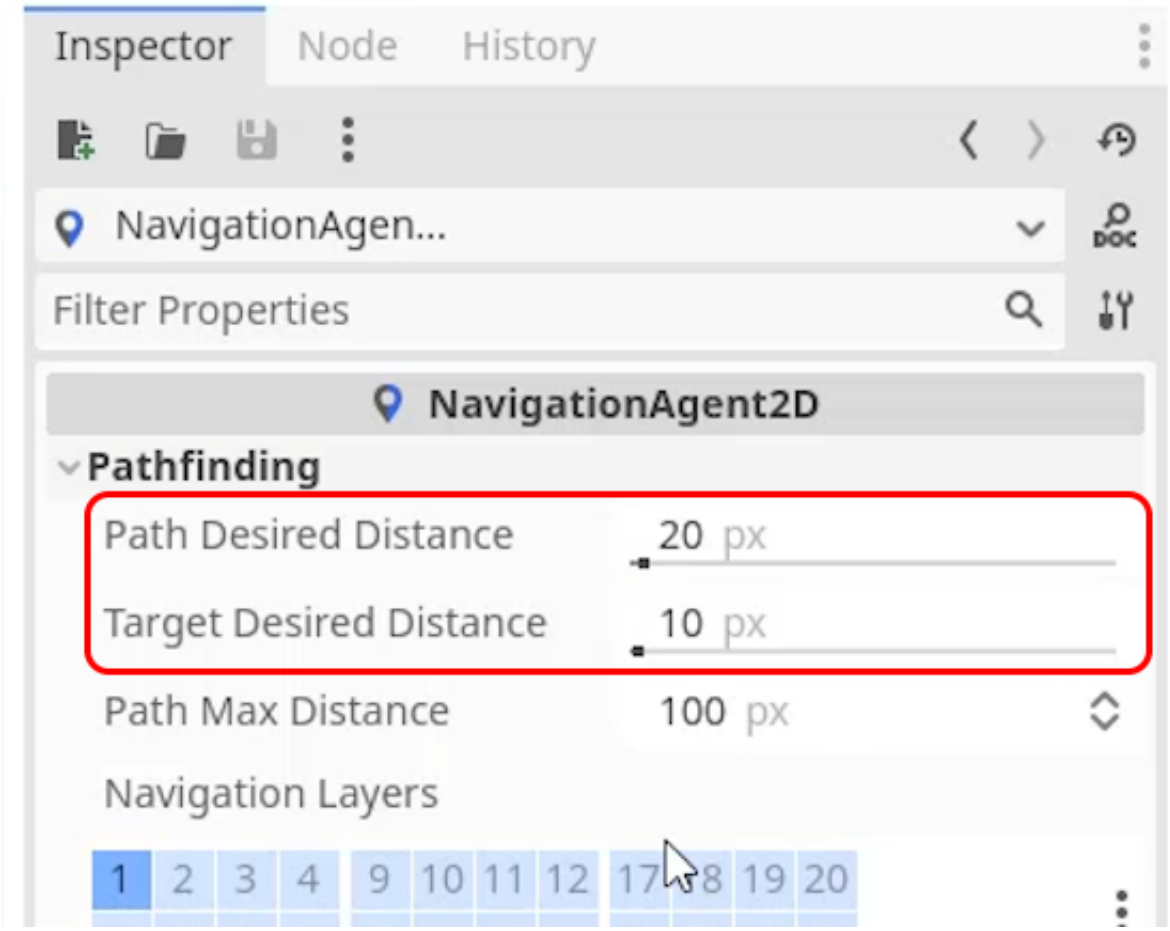
With our movement in place, we are ready to begin testing our *Unit* scene. We don't have any way our commanding our *Units* yet, so instead we will test the *Unit* instance by moving it from the left to the right of the environment. To make this work, we will temporarily code in a position to move to in our *Unit* script. To get the position we want to move to, we can move our *Unit* in the scene editor, and record the position values that we want it to move to. Remember to move the *Unit* back after you have found the position.



In our case, we will be using **(60, 0)**. We can use the **move_to_location** function that we created previously to make our *Unit* move.

```
func _ready():
    ...
    move_to_location(Vector2(60, 0))
```

Now if you press the **play** button, you will see your *Unit* move from its start position to the target position. If you are not happy with the path the *NavigationAgent2D* node generates, there are some options you can change in the *Inspector* window.



These variables affect the following things:

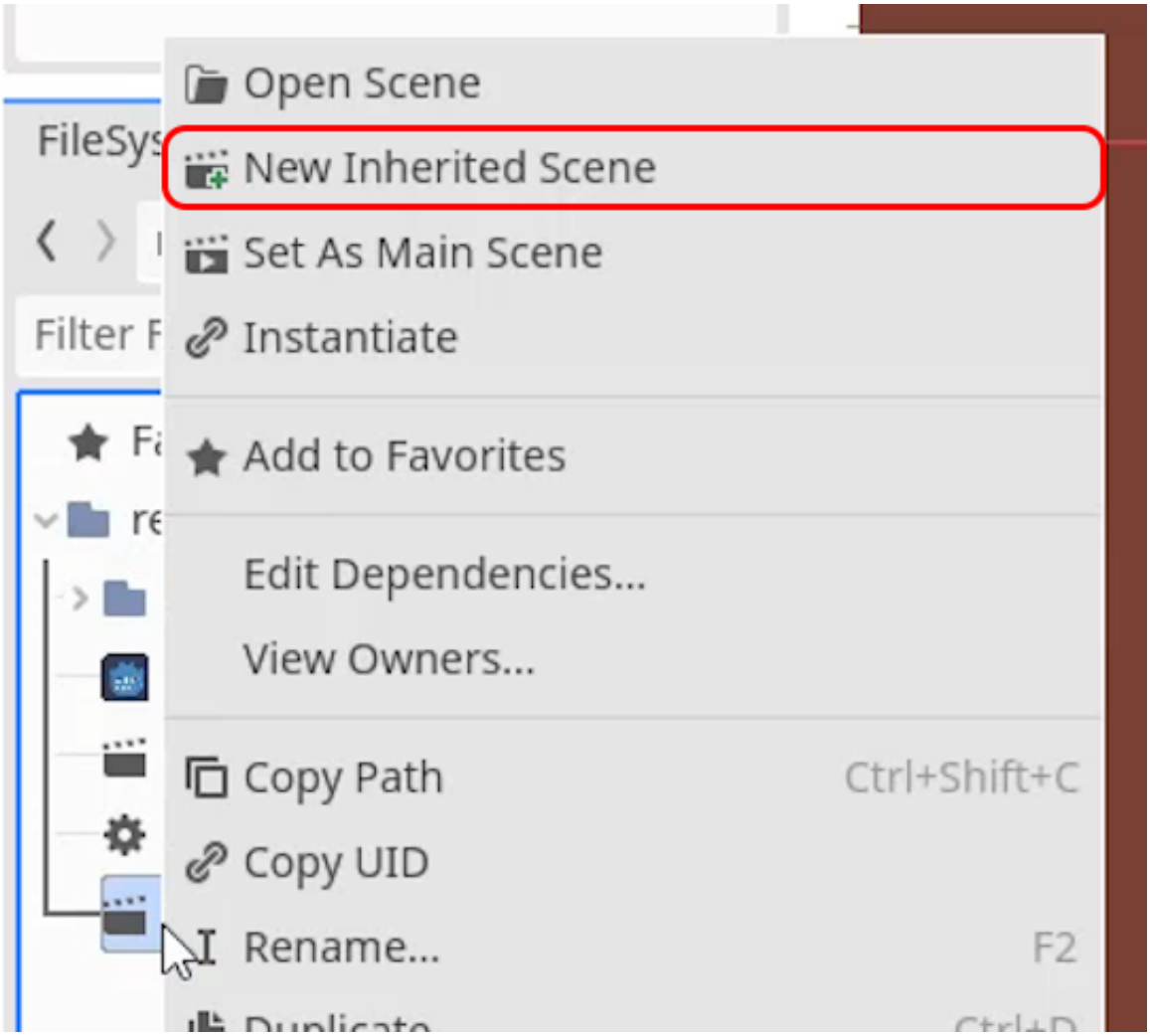
- **Path Desired Distance:** This is the minimum distance to a point that the *Unit* needs to be, before moving on to the next point.
- **Target Desired Distance:** This is the minimum distance to the target destination for the path to be counted as finished.

With this in place, our *Unit* script is ready to go. You can **remove** the **move_to_location** call as we don't want this in the final game.

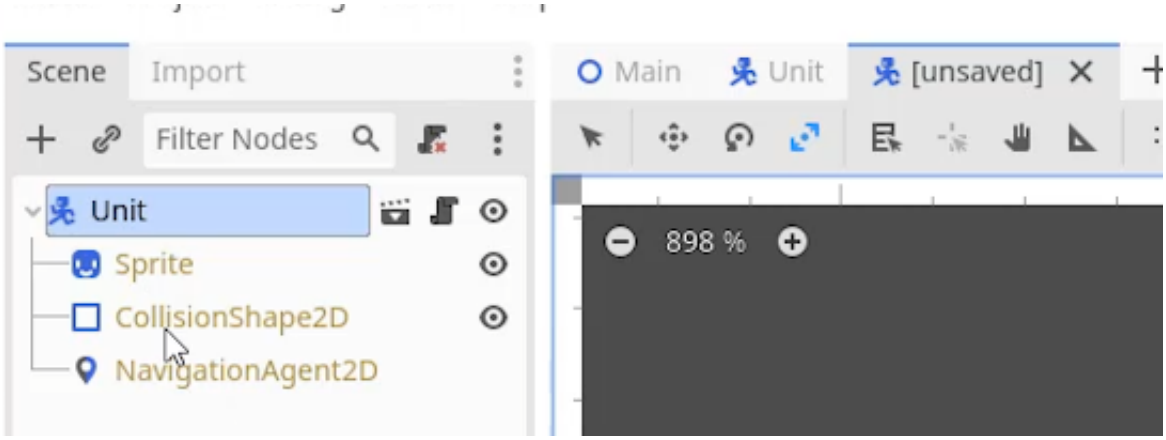
In this lesson, we will be focusing on setting up our player and enemy *Units*. We have already created a *Unit* scene and the *Unit.gd* script. However, for our player and enemy *Unit* instances, we might want additional functionality that the *Unit* scene doesn't offer. For example, our player *Unit* may need the ability to handle selections, and the enemy *Unit* will need some kind of AI to attack player units. It wouldn't be optimal to put all of this in the base *Unit* scene, as our player instances don't need an AI and the enemies don't need a selection visual.

Player Unit Scene

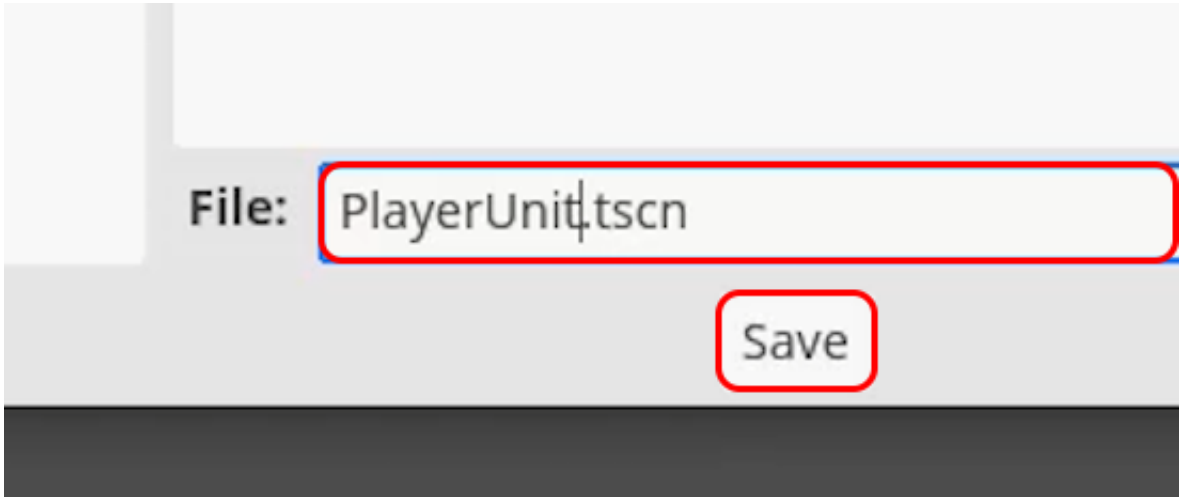
Instead of placing everything in the one *Unit* scene, we can create two new scenes, for our *Player Unit*s and our *Enemy Unit*s which each inherit from the *Unit* base scene. To do this **right-click** the **Unit.tscn** file and choose **New Inherited Scene**.



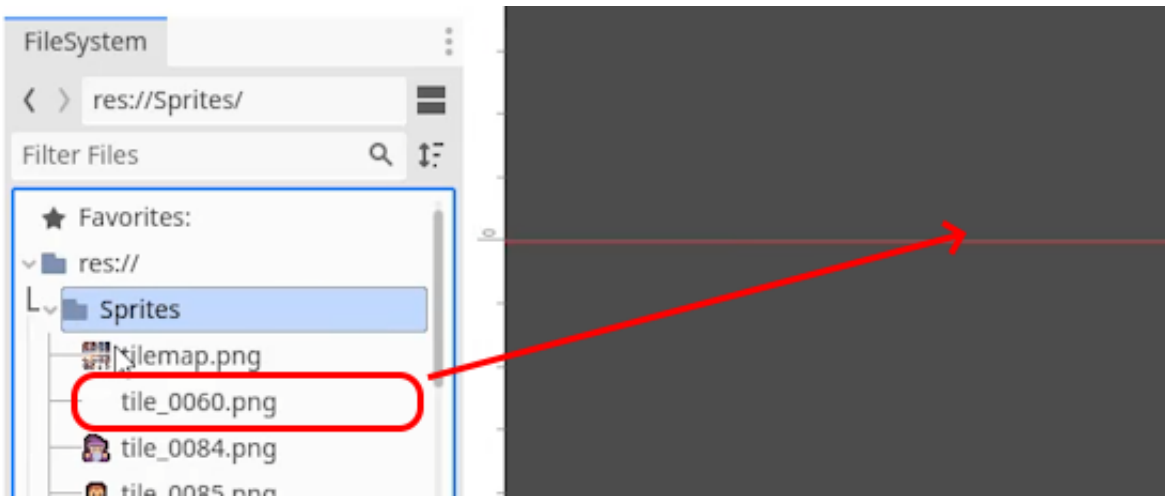
This will create a new scene that contains the nodes and properties from our *Unit* base scene. Some of these nodes appear yellow, these represent the nodes inherited from the *Unit* scene. This means that if any of these nodes are changed in the *Unit* scene, those changes will be replicated in the new scene. However, anything added to the new scene does not affect the *Unit* base scene.



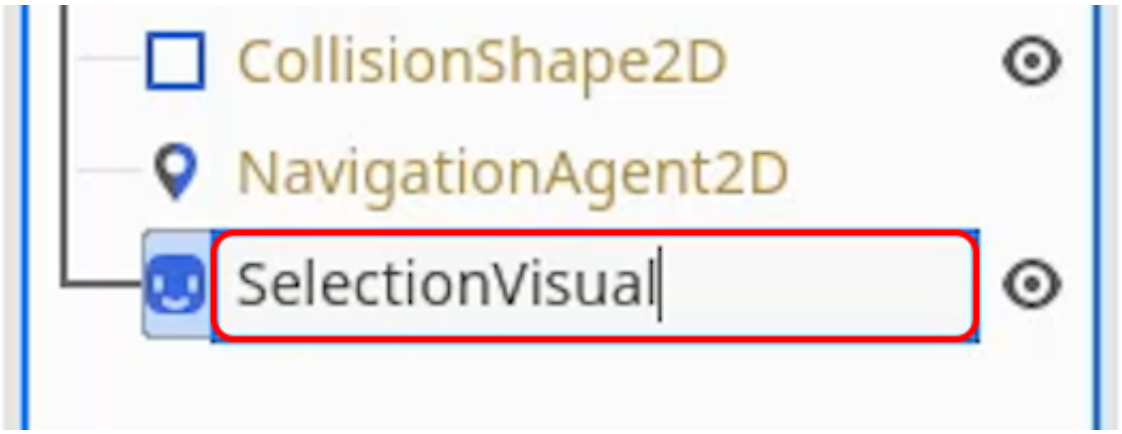
We can then **save** (**CTRL+S**) this scene as its own scene file, called **“PlayerUnit.tscn”.**



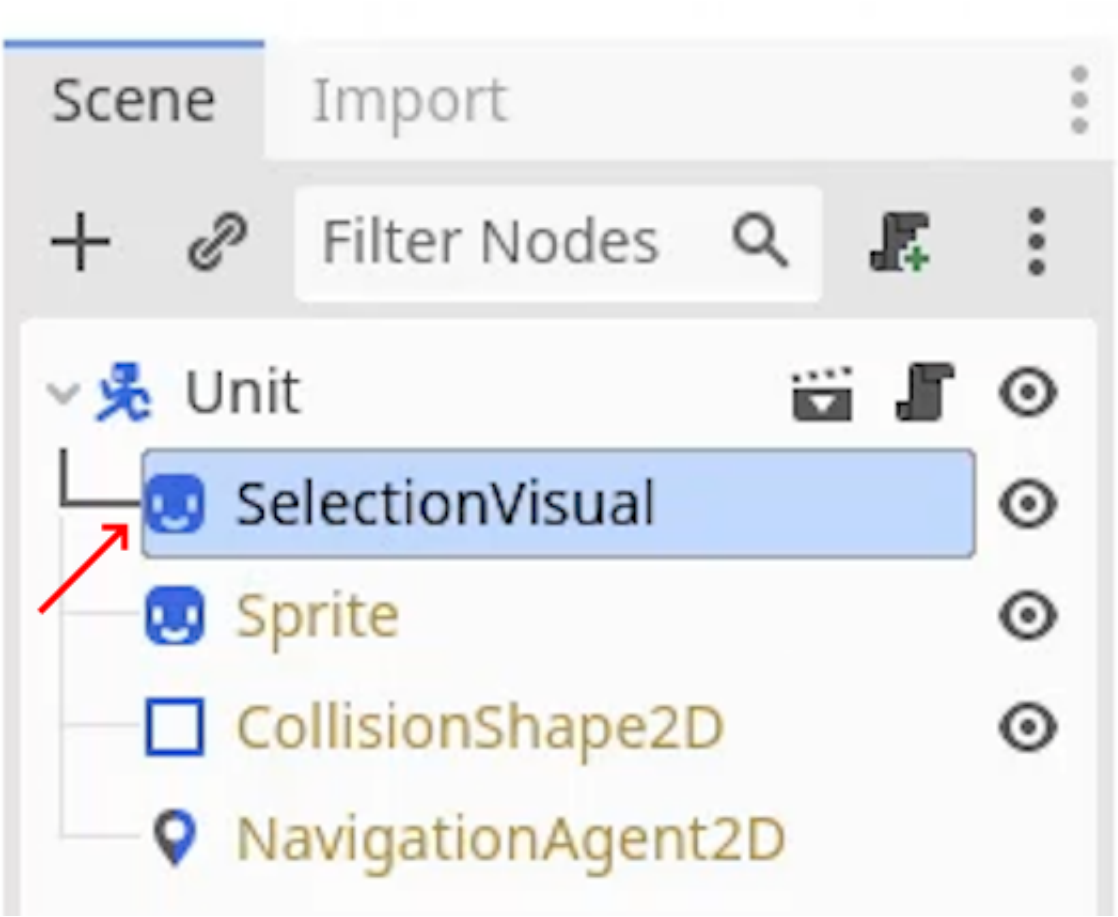
The main difference of our *Player Unit* scene is that it will contain the functionality to let a player select it. The first feature to add for this is the selection visual, which will be a sprite node that appears when the *Player Unit* is selected. For this, we can drag the **tile_0060.png** file into the scene, which will add it as a child node.



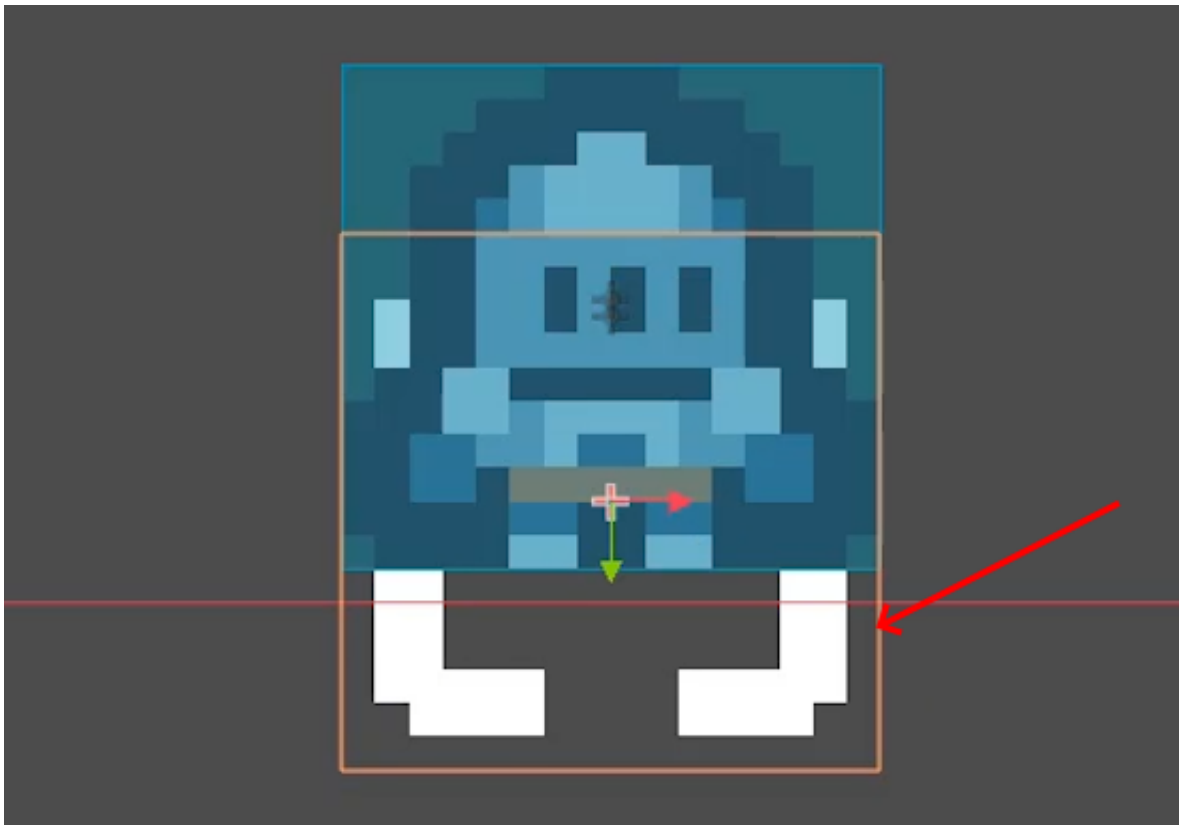
We can **rename** this to **“SelectionVisual”**.



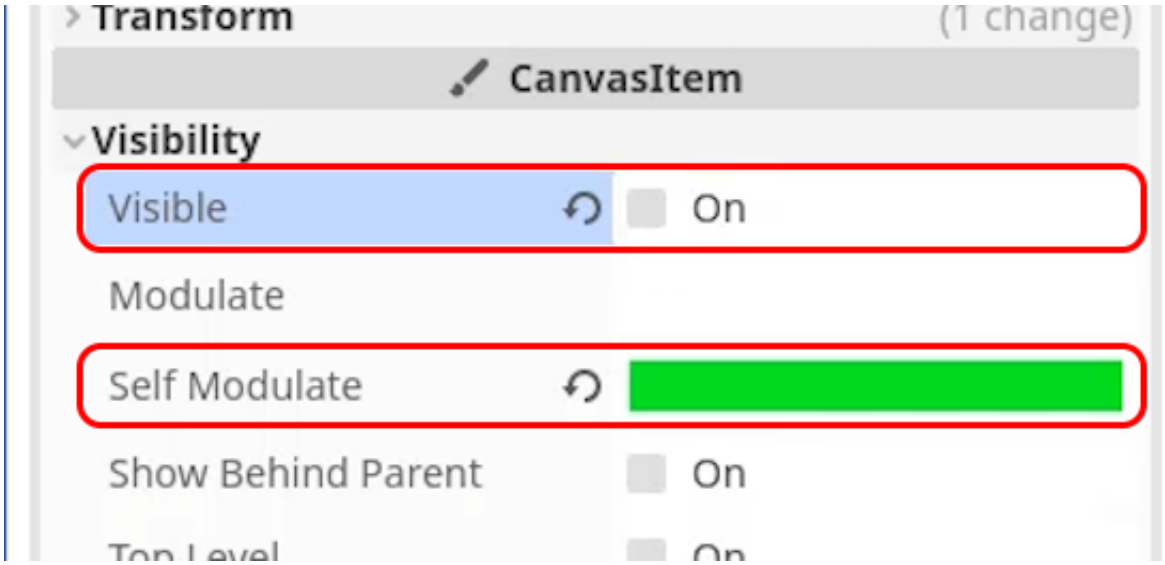
As we want this sprite to appear behind the main *Sprite* node, we will drag the *SelectionVisual* above it in the *Scene* window's hierarchy.



We can also position it just below the *Sprite*.



Finally, we can also change the color of the *SelectionVisual* by changing the **Self Modulate** property. We also want to uncheck the **Visible** property, as we don't want to display the *Player Unit* as selected until it has been selected.

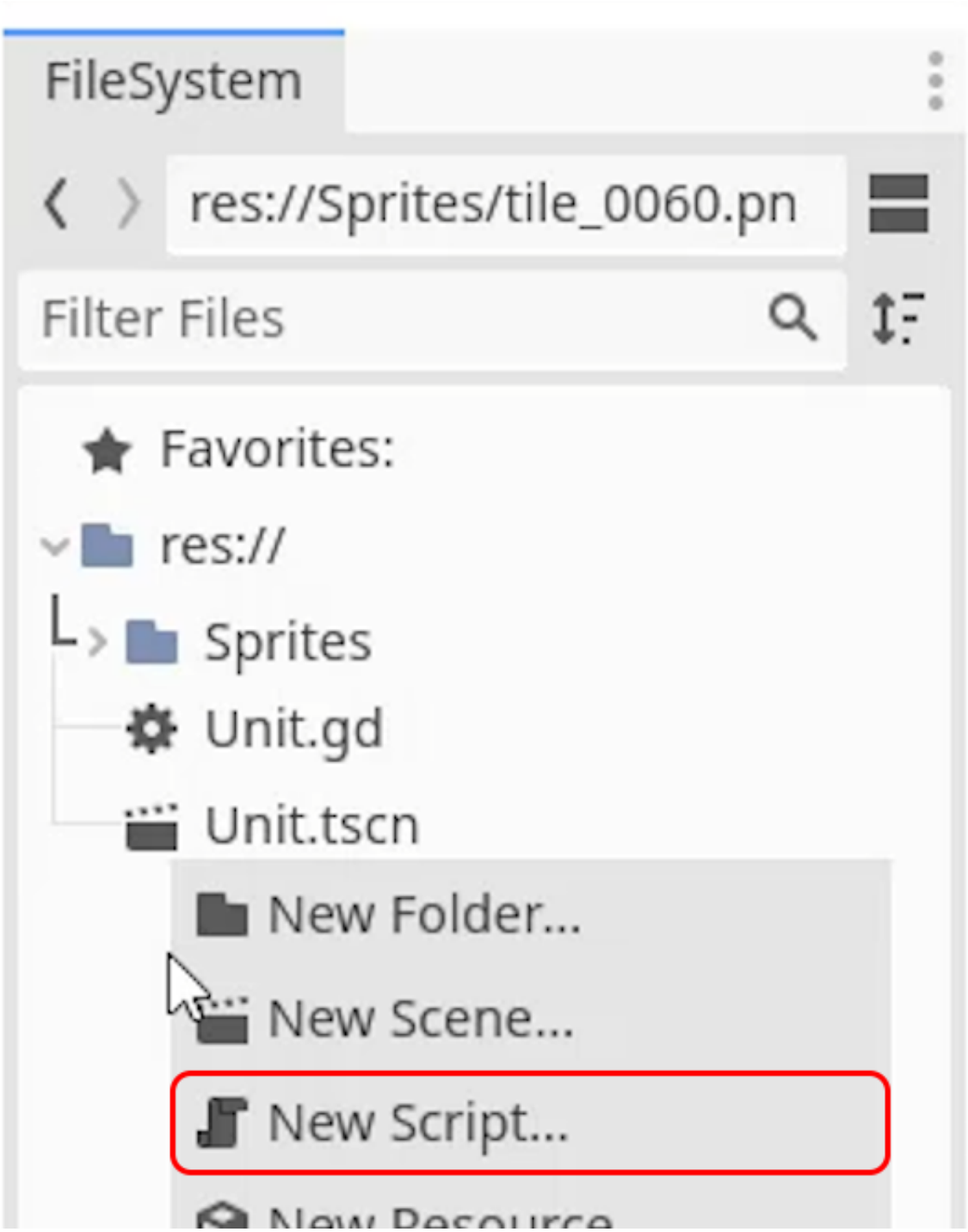


Player Unit Script

We also want to have a separate script for our *Player Unit* as we don't want to add all the functionality to the *Unit* script. To do this, we need to define the *Unit* script as a class, which we can do using the **class_name** keyword.

```
class_name Unit
```

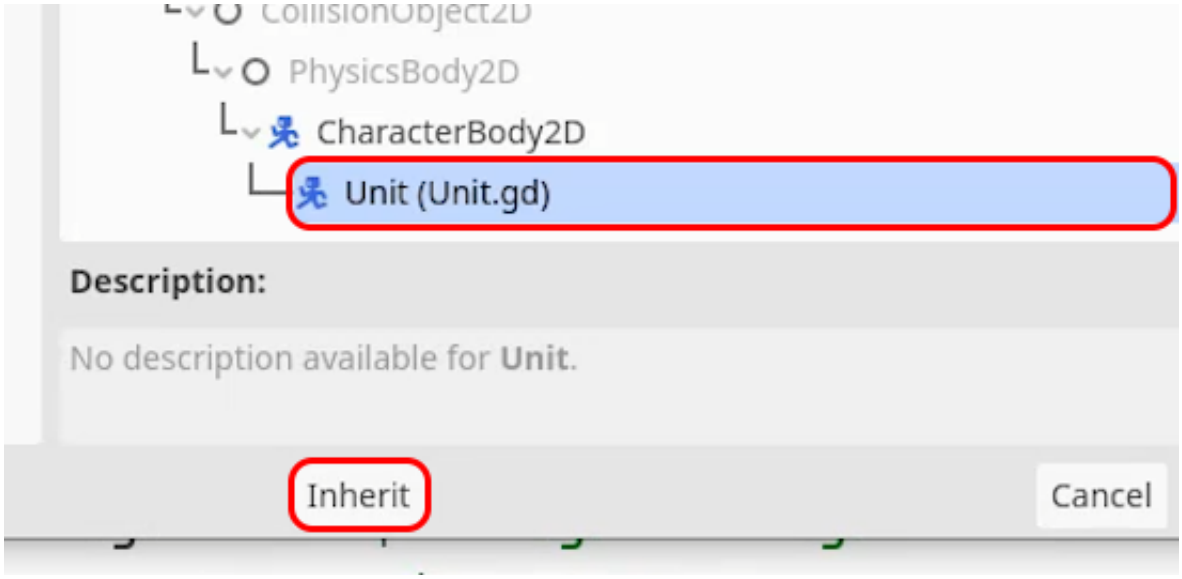

By assigning a class name of *"Unit"* this means we can extend other scripts from it, similarly to how *Unit* extends from *CharacterBody2D*. We can now create a **New Script** by **right-clicking** in the *FileSystem* window.



This script will be called **"PlayerUnit.gd"**.

For the **Inherits** property, we will open the inheritance tree using the **tree** button.

Here we can search for, and select, the **Unit** script.



This will create a brand new script that *extends Unit*. Godot may add some extra code automatically, however, we will only be using this first *extends* line, so anything but this can be removed.

```
extends Unit
```

By extending the *Unit* class, our *PlayerUnit.gd* script will have access and build upon all of the functions and properties of the *Unit.gd* script. To begin our *Player*-specific functionality, we will add a variable to keep track of the selection. The variable will be called **selection_visual** and have a type of **Sprite2D**. We can add an **onready** tag (to make the code run at the start of the game) and we will use the **get_node** function to find the **SelectionVisual** node.

```
@onready select_visual : Sprite2D = get_node("SelectionVisual")
```

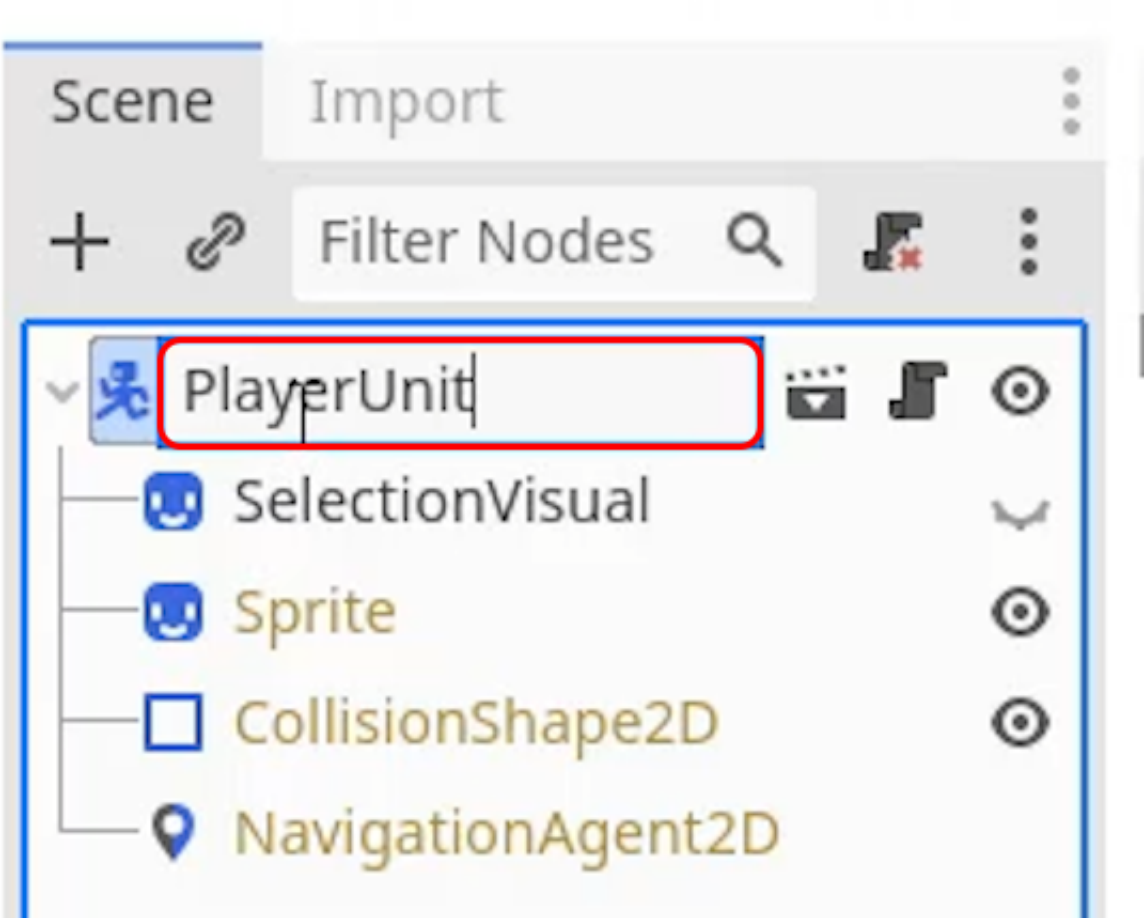

Next, we will add a function called **toggle_selection_visual** with a property of **toggle**.

```
func toggle_selection_visual (toggle):
```

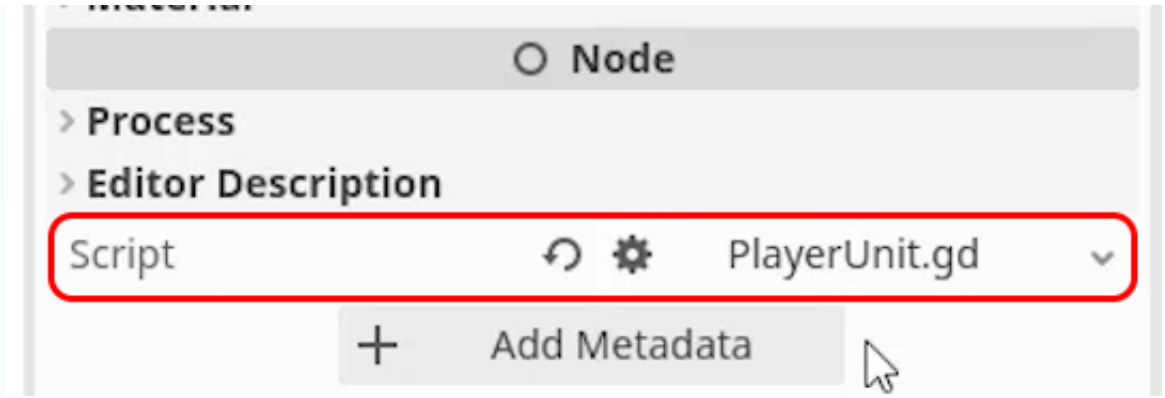
We can then assign the **visible** property of the *select_visual* node to match the toggle property.

```
func toggle_selection_visual (toggle):
    selection_visual.visible = toggle
```

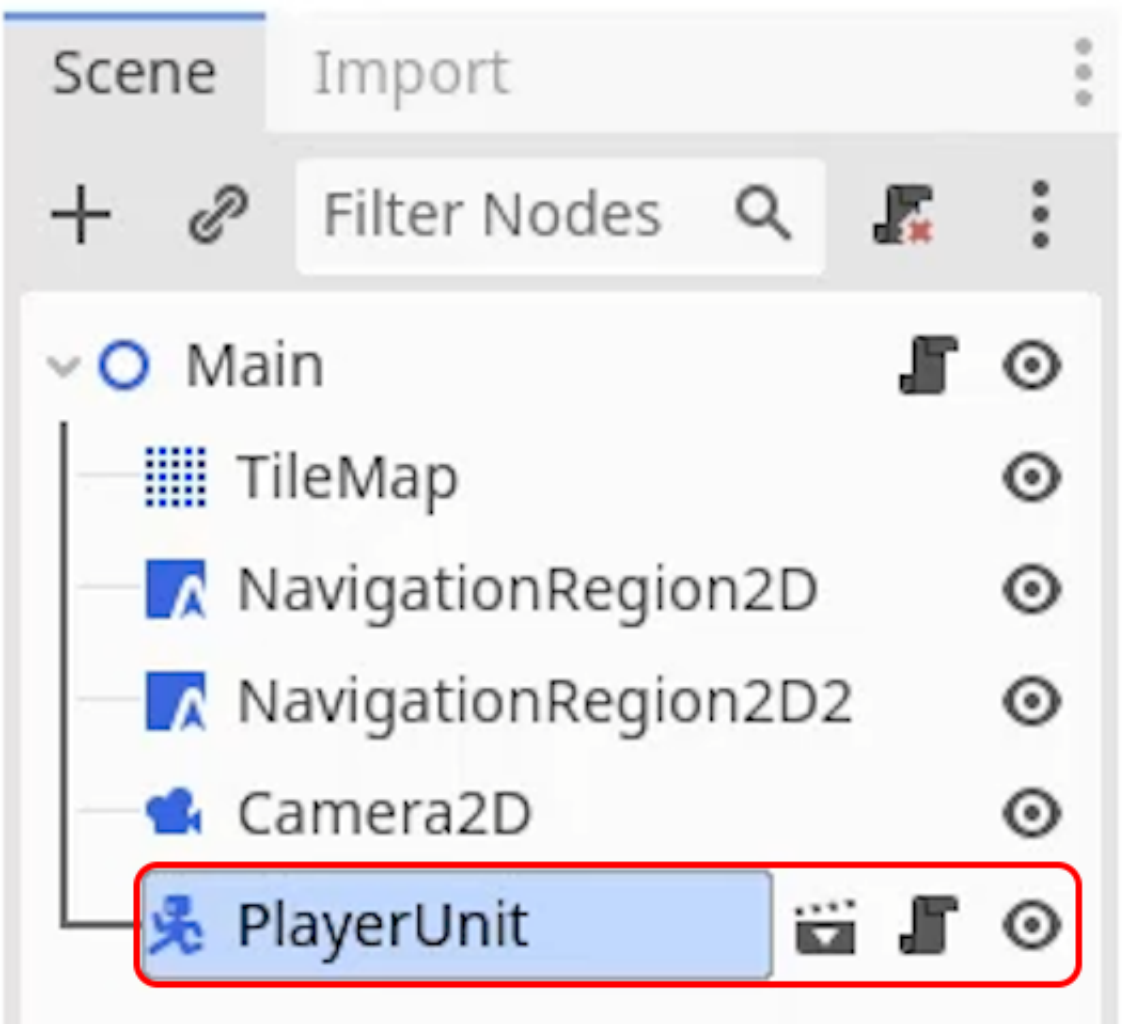
The next stage is to assign our script to the *Unit*. To begin with, we will **rename** the root node to **“PlayerUnit”** to make it more descriptive of the scene.



We can then change the **Script** property to our new **PlayerUnit.gd** script file.

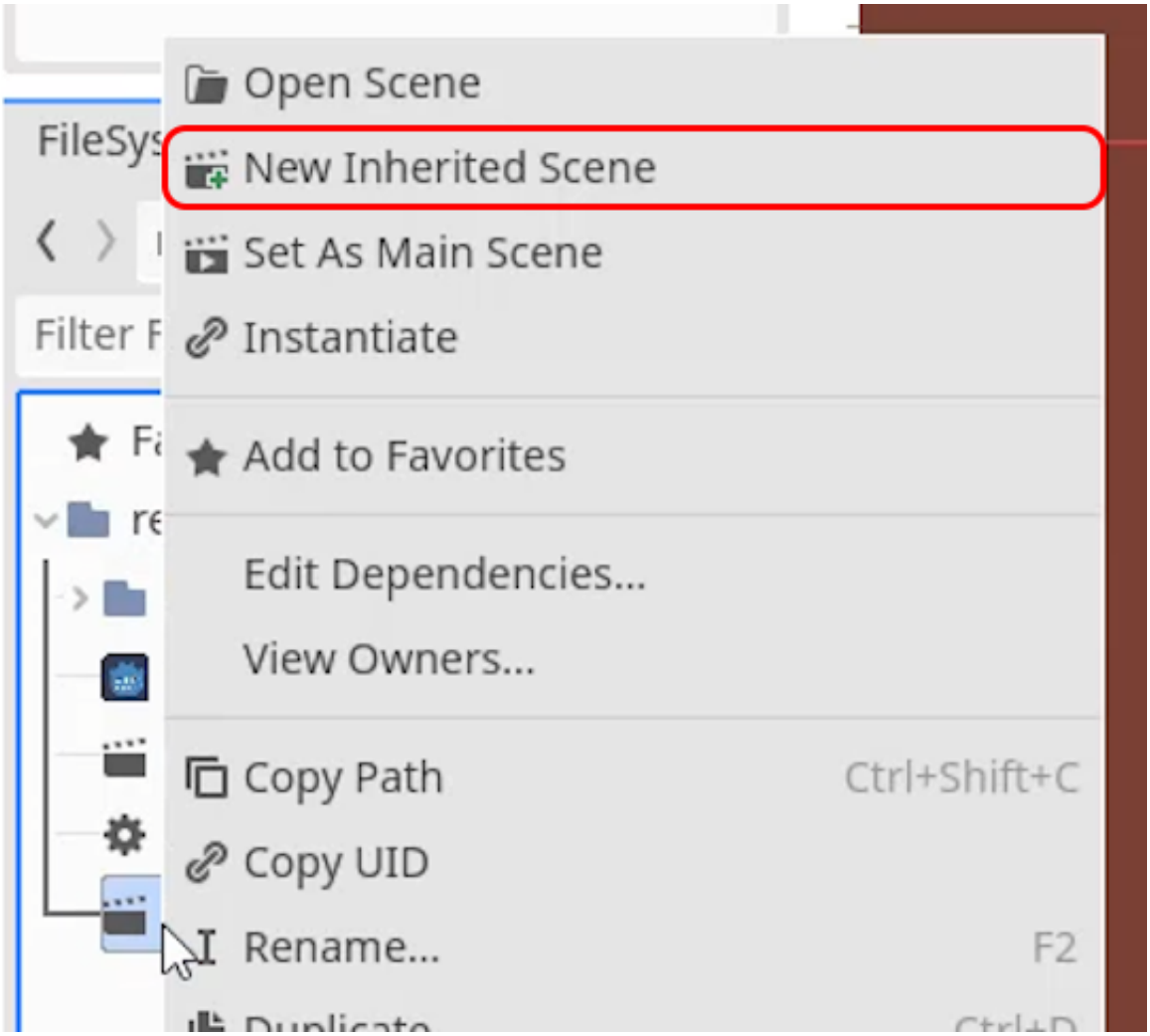


We can now replace the *Unit* instance in our *Main* scene with an instance of the *PlayerUnit*.

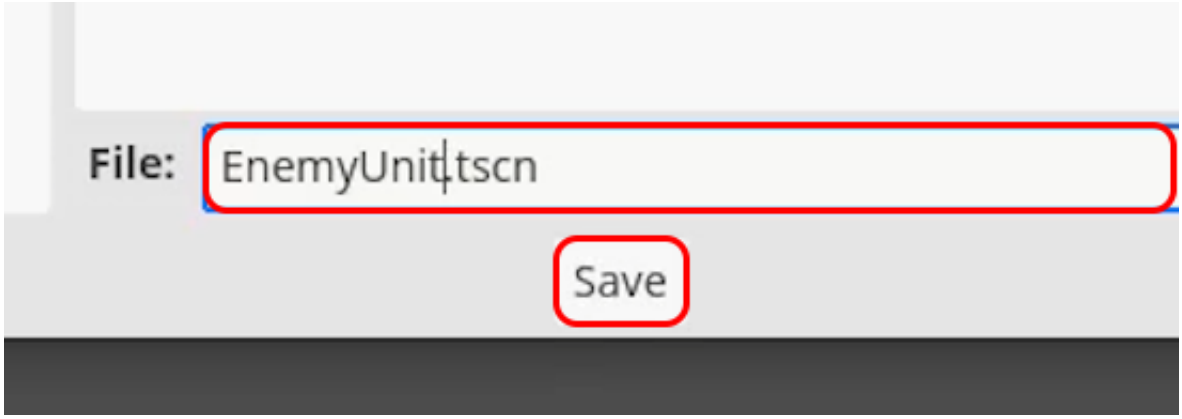


As a challenge, try setting up the *Enemy Unit* scene and script before the next lesson. The enemies can't be selected, so there is no need to add any nodes.

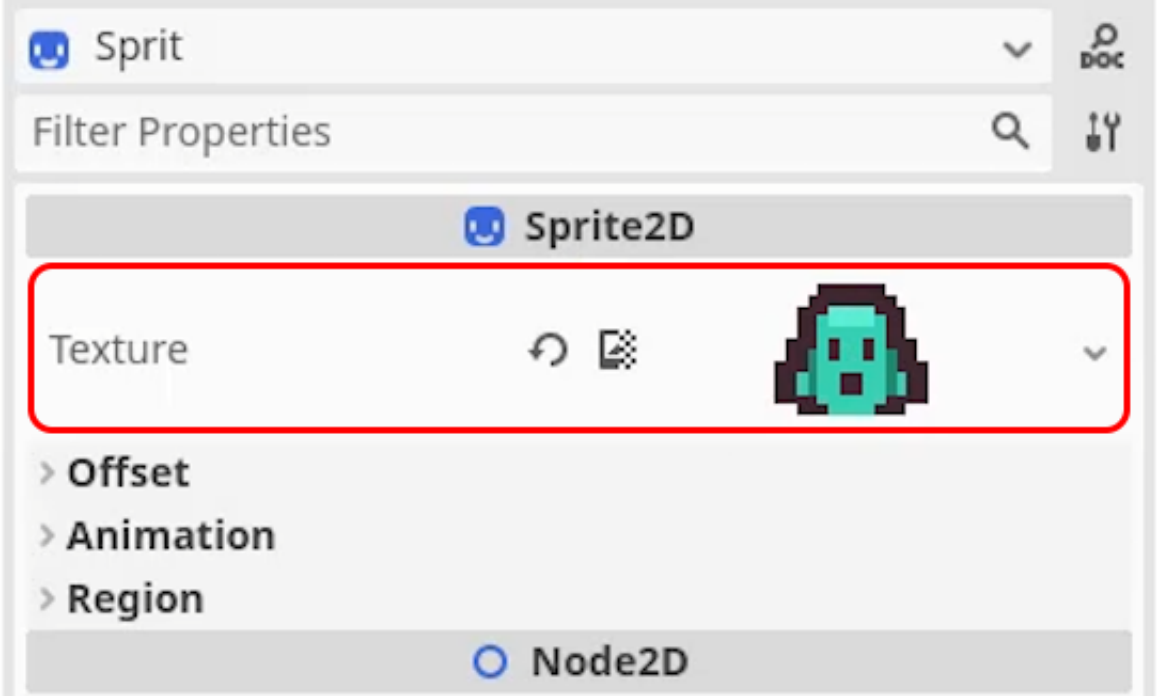
In the previous lesson, we challenged you to try setting up the *Enemy Unit* scene. To start this we will **right-click** the **Unit.tscn** scene file and select **New Inherited Scene**.



This can be **saved** (**CTRL+S**) as *"EnemyUnit.tscn"*.

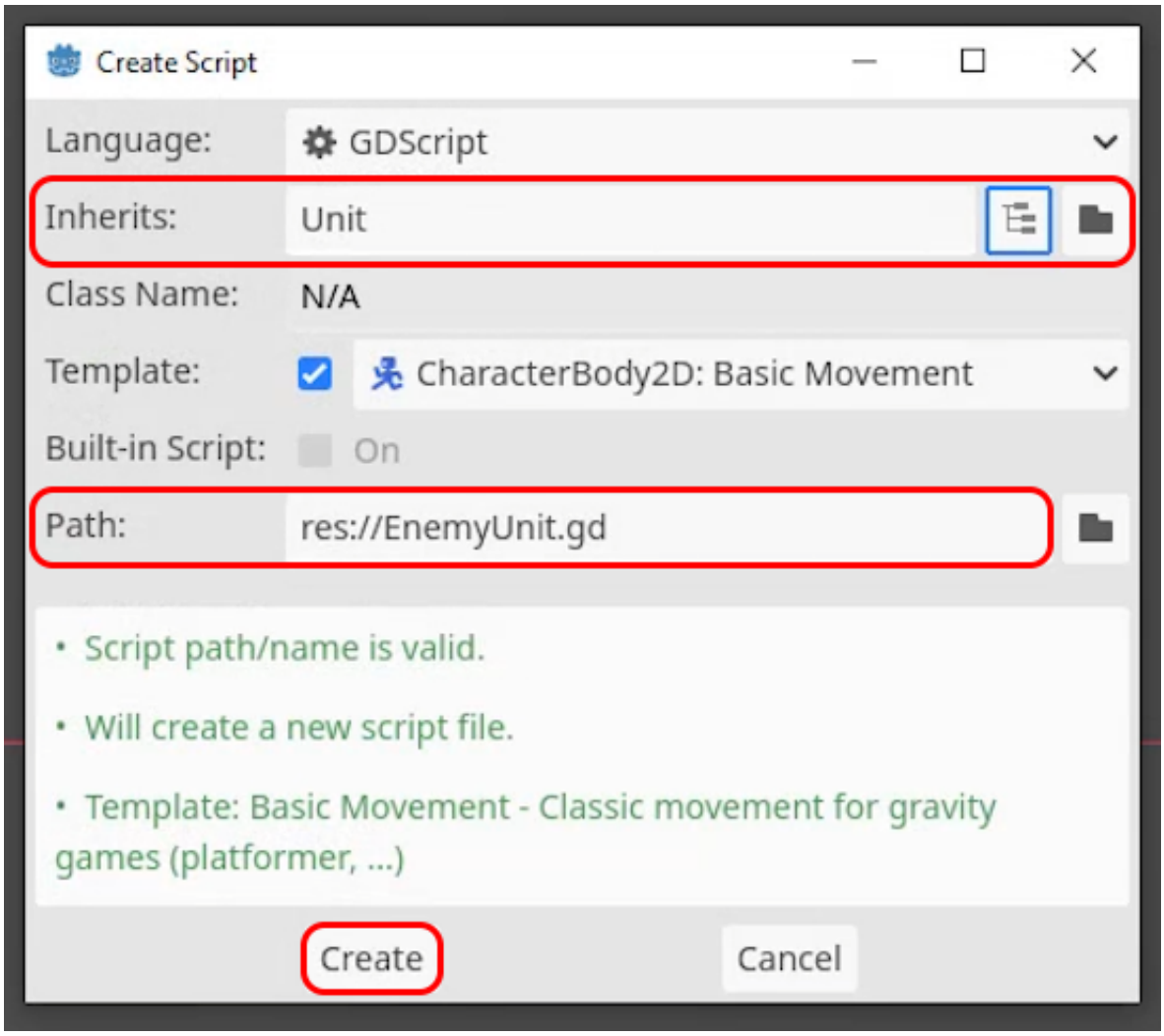


To make this scene easily identifiable as an *Enemy Unit*, we will change the **Texture** property of the **Sprite** node to an enemy asset.

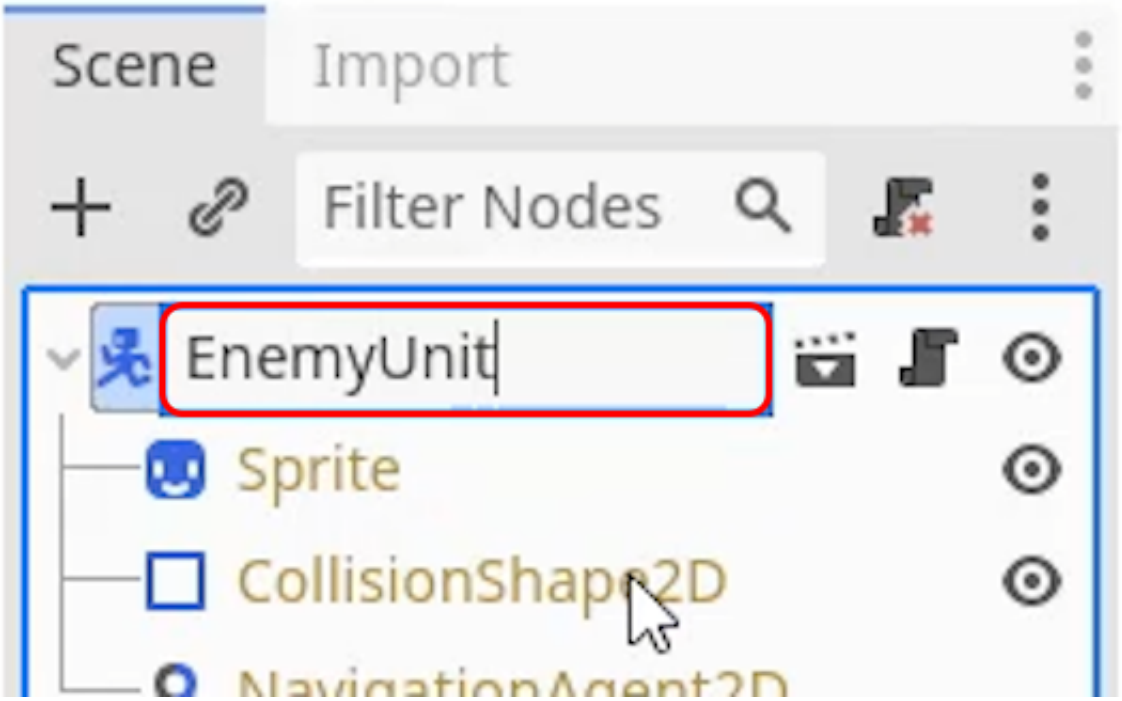


Enemy Script

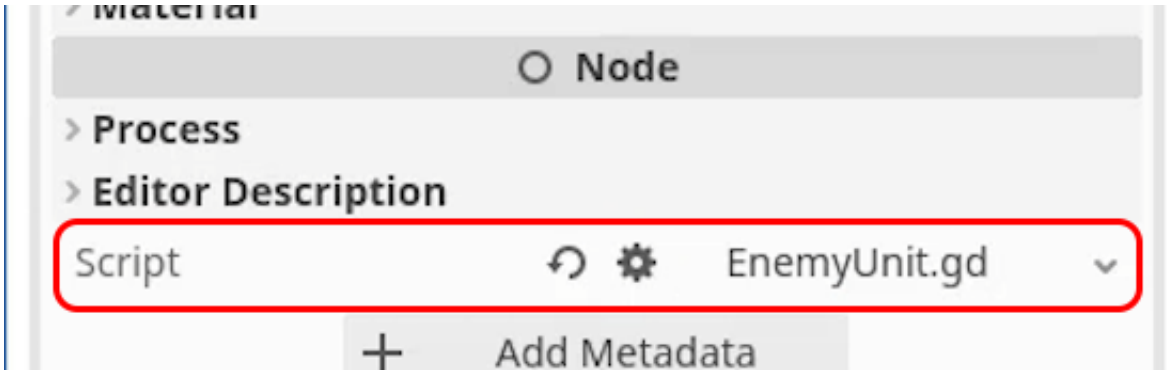
The next step is to set up the *Enemy.gd* script. We will do this by creating a **new script** in the *File System* window, called **“EnemyUnit.gd”**, that inherits from **Unit**.



Before assigning the script, we will **rename** the root node to **“EnemyUnit”**.



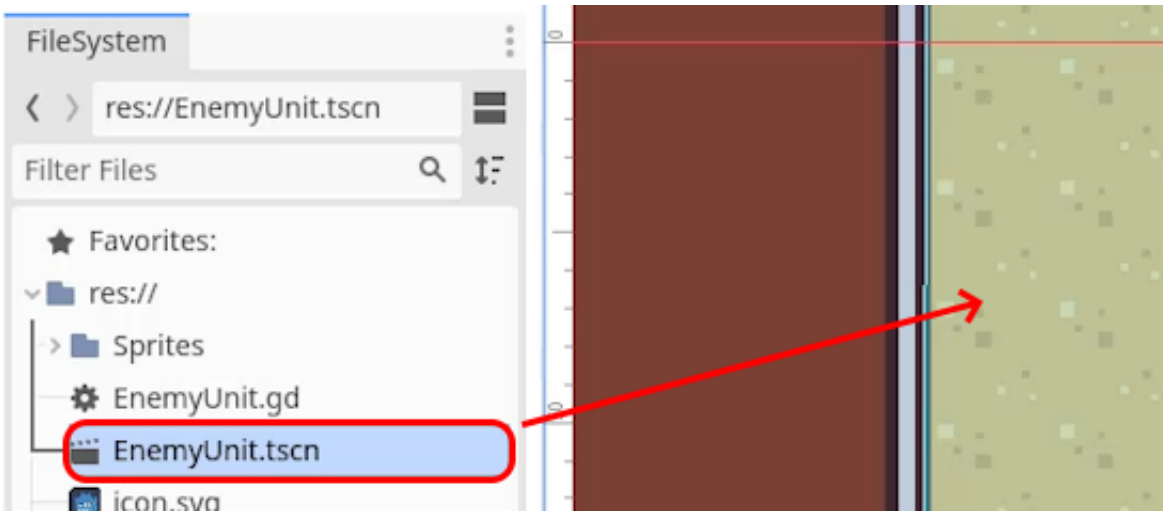
We can then assign the **Script** property of this node to the script we have just created.



Godot may automatically generate some code for this script, like before, we will remove this and just leave the *extends* line.

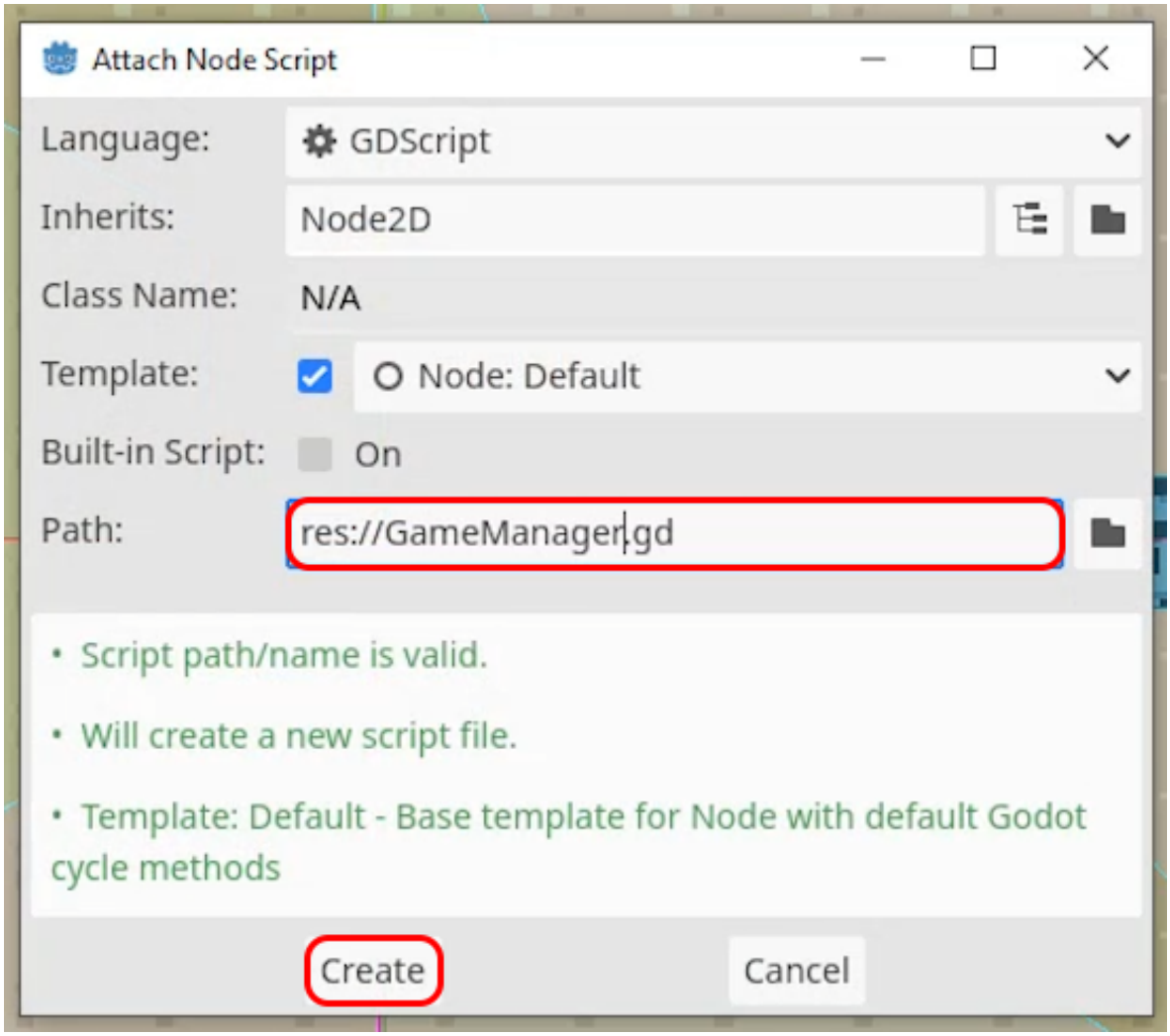
```
extends Unit
```

We won't be filling this script out in this lesson, as this will be its own lesson in the future. We can also instantiate the *Enemy Unit* scene into the *Main* scene.



This will complete our *Enemy Unit* for now, in the next lesson, we will begin setting up our game manager, which will allow us to select and control *Player Units*.

In this lesson, we will be working on the *Game Manager* script. This script will handle everything from detecting units to the player inputs, allowing the player to select and command their *Units*. We will be attaching this script to the root node of the **Main** scene. This script will be called **“GameManager.gd”**.



This script won’t make use of the *_ready* or *_process* functions, so we can remove these, leaving just the *extends* line.

```
extends Node2D
```

Game Manager Variables

This script will only need three variables to keep track of the game. The first will track our **selected_unit**, which will have a type of **CharacterBody2D**. This variable will keep track of the unit the player currently has selected.

```
var selected_unit : CharacterBody2D
```

We also need two variables to keep track of every *Player Unit*, and another to keep track of every

Enemy Unit. For this, we will be using a type of **Array**, which is a variable type that can hold multiple values in a list format. To define the type of data the *Array* will store, we will use square brackets containing **CharacterBody2D**. We will call these variables **players** and **enemies**.

```
var players : Array[CharacterBody2D]
var enemies : Array[CharacterBody2D]
```

Get Selected Unit Function

The first function we will create is the **_get_selected_unit** function. This function will find if a *Unit* is currently under the mouse cursor, and return that *Unit* if one is available. To do this we will be sending a *query* to Godot's *Space State*, telling it we are looking for a *Unit* object and where we are looking for it. This sounds a bit complex at first, but it can be broken down into multiple steps. The first step is to get the *Space State* and store it in a variable, called **space**.

```
func _get_selected_unit():
    var space = get_world_2d().direct_space_state
```

Next, we want to create a **query** variable which will be a **PhysicsPointQueryParameters2D** object. This will allow us to pass the query a **position** value, which means Godot will search the scene for an object at a given position.

```
func _get_selected_unit():
    ...
    var query = PhysicsPointQueryParameters2D.new()
```

For the **position** value, we will supply the **global mouse position**, as we want to check what object is below the mouse cursor.

```
func _get_selected_unit():
    ...
    query.position = get_global_mouse_position()
```

The next step is to create a variable called **intersection**, which we will get using the **space** object we defined above. This function will take our *query* value, therefore giving the mouse position, and a value of how many objects to find, which we will pass as **1** object.

```
func _get_selected_unit():
    ...
    var intersection = space.intersect_point(query, 1)
```

This code will find if an object is below the cursor in the scene. The next step is to find if an object was found, which we can do using the **is_empty** function and an **!** to check that it is *not* empty.

```
func _get_selected_unit():
```



```
    ...
    if !intersection.is_empty():
```

If the intersection value is not empty, we will return the **collider** of the first object found (at the **0th** position in the list).

```
func _get_selected_unit():
    ...
    if !intersection.is_empty():
        return intersection[0].collider
```

Finally, if this code doesn't run, we can return a value of **null** at the end of the function.

```
func _get_selected_unit():
    ...
    return null
```

General Game Manager Functions

With this function complete, we can begin implementing the other functions. To begin with, we will create each function using the **pass** keyword, so that we can come back and add the functionality afterwards. The first function will be called **_try_select_unit**, and will call the *_get_selected_unit* to try and find a *Unit* object to select.

```
func _try_select_unit():
    pass
```

The next function will be called **_select_unit**, which will handle the functionality behind selecting a *Unit* object when one is found. This function will have a parameter of **unit**, which will be the *Unit* object found below the cursor.

```
func _select_unit(unit):
    pass
```

We will need a similar function called **_unselect_unit**, which will unselect the *Unit* stored in the *selected_unit* variable.

```
func _unselect_unit():
    pass
```

_try_command_unit will be a function that is called when the player right-clicks on the ground or an *Enemy Unit*.


```
func _try_command_unit():
    pass
```

Finally, we will use the **_input** function provided by Godot to handle input events. This function requires a parameter of **event** which will give us the information about the player's input.

```
func _input(event):
```

We will start by checking if the **event** property is a **mouse-click** event.

```
func _input(event):
    if event is InputEventMouseButton and event.pressed:
```

This code checks if the *event* property is a mouse button event, and then checks if the event is about a mouse button being pressed. Next, we want to check if the button is the **left mouse button** as this will be used to call the **try_select_unit** function.

```
func _input(event):
    if event is InputEventMouseButton and event.pressed:
        if event.button_index == MOUSE_BUTTON_LEFT:
            _try_select_unit()
```

On the other hand, if the **right mouse button** is pressed, we will call the **_try_command_unit** function.

```
func _input(event):
    if event is InputEventMouseButton and event.pressed:
        ...
        elif event.button_index == MOUSE_BUTTON_RIGHT:
            _try_command_unit()
```

We can now implement the **_try_select_unit** function, as this will be the first to be called. Inside this function, we will check if the mouse is over a unit, which we will do using the **_get_selected_unit** function.

```
func _try_select_unit():
    var unit = _get_selected_unit()
```

This function will either return a collider object or return *null,* depending on if an object is found or not. The first thing to check is whether an object is found (**!= null**) and then check if the object is actually a *Player Unit* object. We want to check the **is_player** variable here, as we don't want to try and select an *Enemy Unit*.


```
func _try_select_unit():
    ...
    if unit != null and unit.is_player:
```

If this is the case, we will call the **_select_unit** function, while sending over the **unit** variable.

```
func _try_select_unit():
    ...
    if unit != null and unit.is_player:
        _select_unit(unit)
```

In the case a *Unit* object isn't found, we want to call the **_unselect_unit** function, as the player is clicking in empty space.

```
func _try_select_unit():
    ...
    else:
        _unselect_unit()
```

Continuing the *selection* code, we first want to call the **_unselect_unit** function in the **_select_unit** function, so that before a new unit is selected, the last one is unselected.

```
func _select_unit(unit):
    _unselect_unit()
```

We can then update the **selected_unit** variable with the new **unit** parameter. We can also call the **toggle_selected_visual** function on the unit, to make it visible to the player that the *Unit* has been selected.

```
func _select_unit(unit):
    ...
    selected_unit = unit
    selected_unit.toggle_selected_visual(true)
```

In the next lesson, we will continue implementing the other functions in this script.

To continue on from the previous lesson, we will begin by implementing the **_unselect_unit** function. The first thing to do here is to check if there is currently a selected unit, as we don't want to try and unselect nothing.

```
func _unselect_unit():
    if selected_unit != null:
```

In the case that there is a *selected_unit* value, we will toggle the selection visual on this *Unit*.

```
func _unselect_unit():
    if selected_unit != null:
        selected_unit.toggle_selection_visual(false)
```

We will then update the *selected_unit* value to **null** to show there is no currently selected *Unit* object.

```
func _unselect_unit():
    ...
    selected_unit = null
```

The final function to implement is the **_try_command_unit** function. We don't want this function to run if no **selected_unit** exists, so this is the first thing to check.

```
func _try_command_unit():
    if selected_unit == null:
        return
```

The next thing this function needs to do is find out whether the mouse is currently over an *Enemy Unit*. This means we can reuse the **_get_selected_unit()** and store it in a variable called **target**.

```
func _try_command_unit():
    ...
    var target = _get_selected_unit()
```

To check if this is an *Enemy Unit* we first need to check the **target** value is not **null**, and then check the **is_player** property is false.

```
func _try_command_unit():
    ...
    if target != null and target.is_player == false:
```

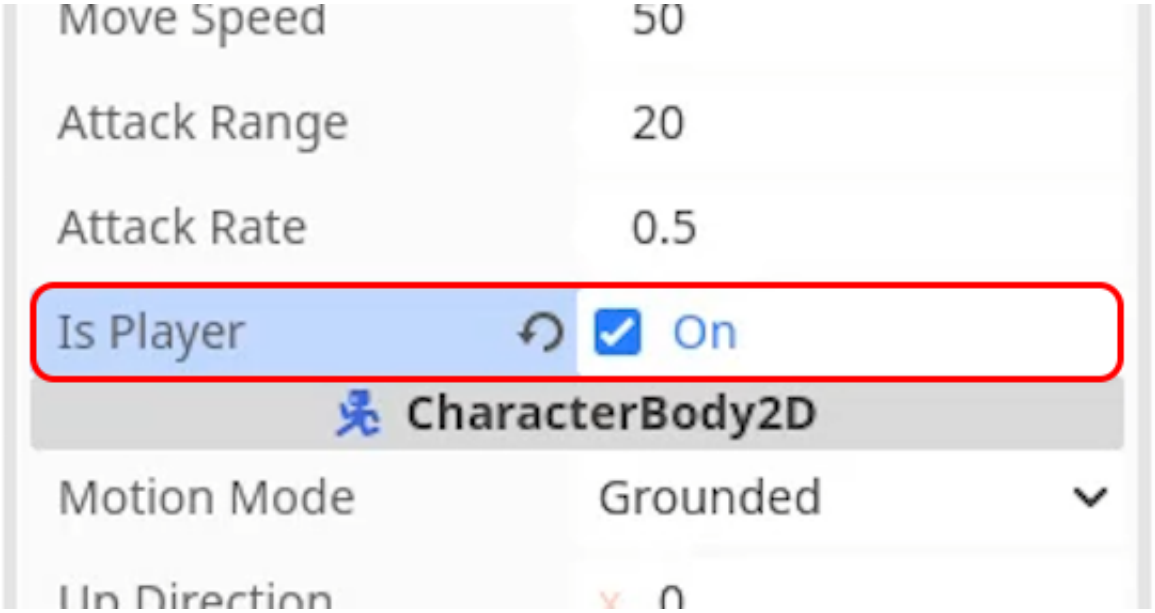
If the player has clicked on an *Enemy Unit*, we can direct the **selected_unit** towards it, using the **set_target** function.


```
func _try_command_unit():
    ...
    if target != null and target.is_player == false:
        selected_unit.set_target(target)
```

In the case that an *Enemy Unit* isn't found, we can call the **move_to_location** function to make the **selected_unit** move towards the global mouse position.

```
func _try_command_unit():
    ...
    else:
        selected_unit.move_to_location(get_global_mouse_position())
```

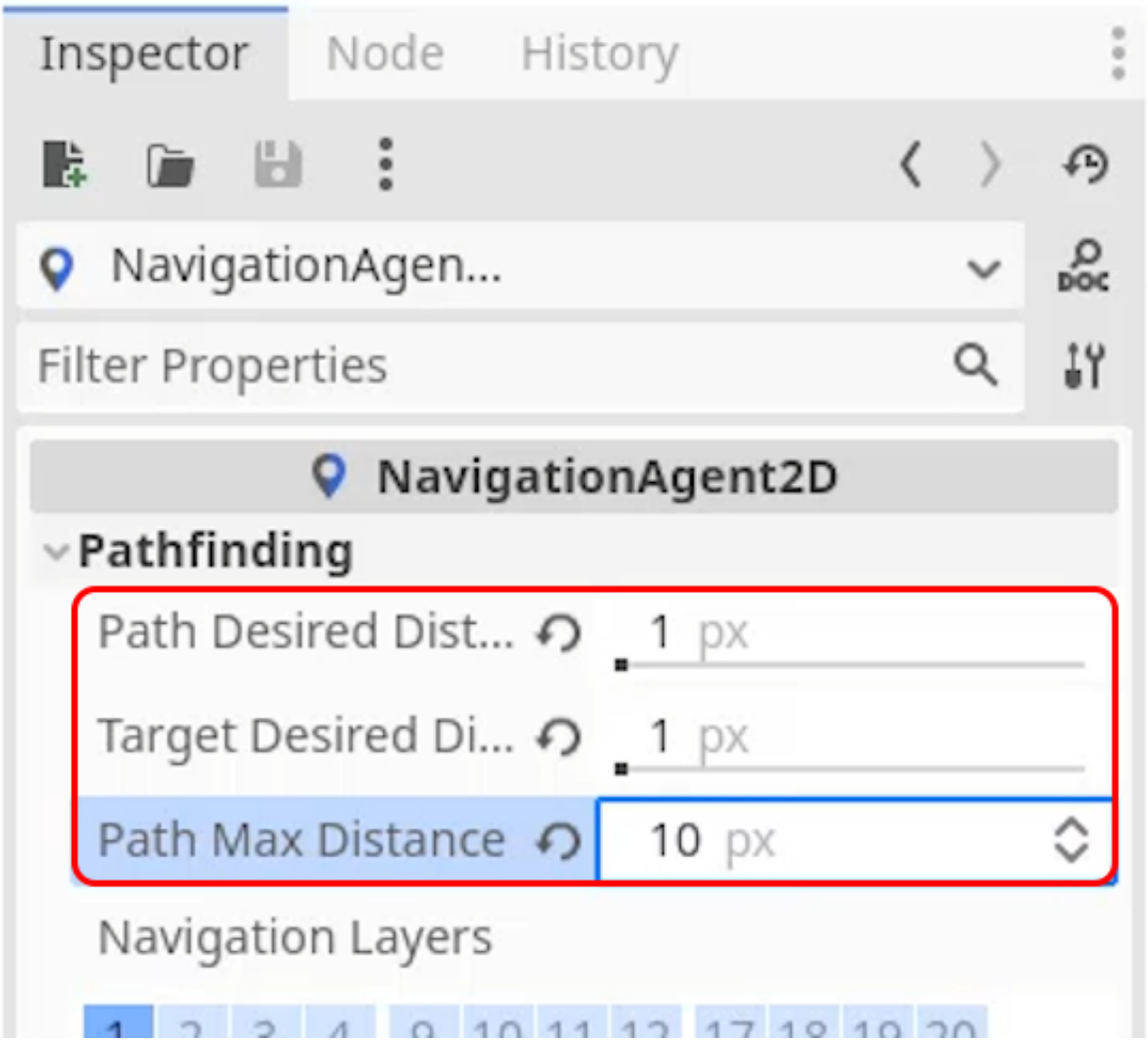
The final thing to do is make sure **Is Player** is checked on the *Player Unit* root node of the *PlayerUnit.tscn* scene. Make sure to **save** (**CTRL+S**) this to update the scene. This should be unchecked on the *Enemy Unit* scene.



Testing the Functions

We are now ready to begin testing the functions we have set up. To do this, press **play** and try **left-clicking** on the *Player Unit* object, this should make the selection visual appear.

In case this isn't working, you can try changing *"$Sprite2D"* to *"$Sprite"* in the *_ready* function of the *Unit.gd* script. You should be able to **right-click** anywhere on the ground and the *Player Unit* will move to that location. If you **left-click** off the agent, it will deselect it. You will notice that the *Unit* often stops short of where you click to move, to fix that we will set the **Path Desired Distance** and **Target Desired Distance** properties of the **NavigationAgent2D** in the *Unit* scene to **1**. We will also change the **Path Max Distance** property to **10**.



The final thing to test is **right-clicking** on the enemy, this will make the *Player Unit* move towards the *Enemy Unit* and attack the enemy. This is a bit difficult to see, as there is no visual representation of the attacks, so in the next lesson, we will set up some damage flashes.

In the previous lesson, we implemented the functions to command our *Player Units,* allowing them to attack the *Enemy Units.* This works well, but we currently have no way of telling when the *Units* are attacking each other, they just disappear when one loses. One easy method to show damage is by making the *Unit's Sprite node* flash when it takes damage. We will do this by modifying the **Modulate** property of the sprite when a damage event occurs. We can easily implement this within the **Unit.gd** script.

We will add the code in the **take_damage** function, below the existing functionality. The first step is to set the **modulate** property of the **sprite** variable to the *red* color. This will apply a red tint on top of the *Sprite* texture.

```
func take_damage(damage_to_take):
    ...
    sprite.modulate = Color.RED
```

Before changing the *modulate* value back, we want to add a small delay. This is made easy in Godot using timers.

```
func take_damage(damage_to_take):
    ...
    await get_tree().create_timer(0.1).timeout
```

This code will create a *Timer* node on the node tree with a wait time of **0.1**, and then wait for the timer to run the **timeout** event. Here the **await** keyword means the code will pause until the *timeout* event occurs. Finally, we can switch the **modulate** value back to **white**. This won't set the sprite to white but will return the texture to default.

```
func take_damage(damage_to_take):
    ...
    sprite.modulate = Color.WHITE
```

We can test this code by **playing** the game and attacking the *Enemy Unit*, this will make the *Enemy Unit*'s sprite flash red each time it is damaged. In the next lesson, we will implement the code to allow our *Enemy Units* to attack the *Player Units*.

The final element to implement into our game is the AI for our *Enemy Units*, which is what we will cover in this lesson. We will be giving the enemies the ability to chase after our player units and deal damage to them. The first step will be to assign our *Units* to either the *players* or *enemies* variables in the *GameManager* script. To do this, we will add some lines to the **_ready** function. The first step will be to get a reference to the *GameManager* script on the **Main** node using the **get_node** function.

```
func _ready():
    ...
    var gm = get_node("/root/Main")
```

Next, depending on the value of **is_player** we will add this *Unit* instance to either the *players* array variable or the *enemies* array variable.

```
if is_player:
    gm.players.append(self)
else:
    gm.enemies.append(self)
```

This will allow us to search the *players* array variable from the **EnemyUnit** script to find if any *Player Units* are nearby. The first step for this is to create a variable called **detect_range**, which will have a type of **float** and a default value of **100.0**. We will make this variable **exported** and this will represent the furtherest away a *Player Unit* can be for an *Enemy Unit* to start tracking it.

```
@export var detect_range : float = 100.0
```

The next variable will be used to track the *GameManager* script. This will be called **game_manager** and will be assigned when the script first runs, using the **get_node** function and the **onready** tag.

```
@onready var game_manager = get_node("/root/Main")
```

This script will be making use of the **_process** function to handle the enemy AI, however, the *Unit* base script already uses *_process*. As we will be overwriting *Unit's* default *_process* functionality, we will call the **_target_check** function, just like the *_process* function does in *Unit*.

```
func _process(delta):
    _target_check()
```

Before this call, we want to check if our *Unit* already has a target, and if not, loop through every available *Player Unit* in the *players* array variable.

```
func _process(delta):
    if target == null:
        for player in game_manager.players:
```



```
    ...
```

We then want to check if the *Player Unit* is dead by checking if it is **null**. If it is, we will use the **continue** keyword to move on to the next iteration of the loop.

```
func _process(delta):
    if target == null:
        for player in game_manager.players:
            if player == null:
                continue
    ...
```

Next, if we have found a *Player Unit*, we want to calculate the distance to this unit and compare it against our **detect_range** variable.

```
func _process(delta):
    if target == null:
        for player in game_manager.players:
            ...
            var dist = global_position.distance_to(player.global_position)
            if dist <= detect_range:
    ...
```

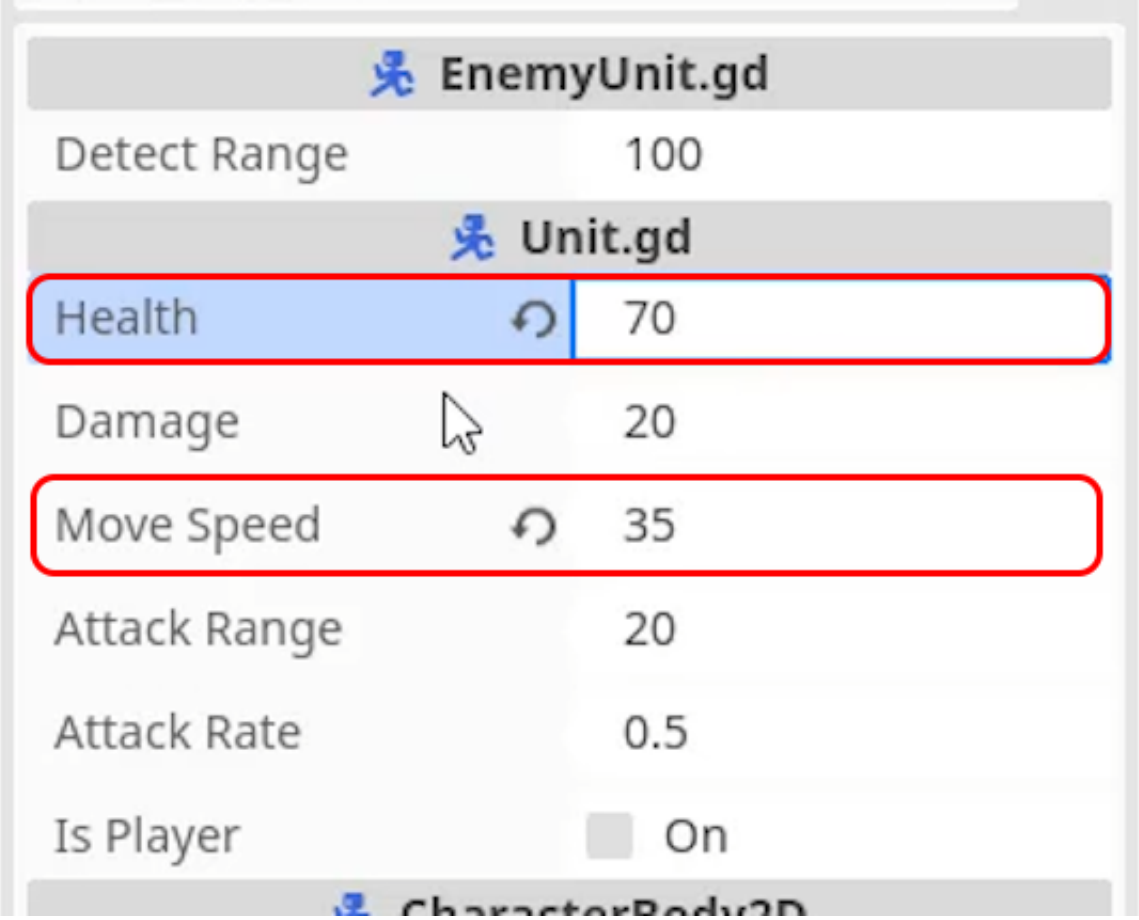
Finally, if the *distance* is less than the *detect_range*, we will call the **set_target** function and pass the **player** parameter through.

```
func _process(delta):
    if target == null:
        for player in game_manager.players:
            ...
            var dist = global_position.distance_to(player.global_position)
            if dist <= detect_range:
                set_target(player)
    ...
```

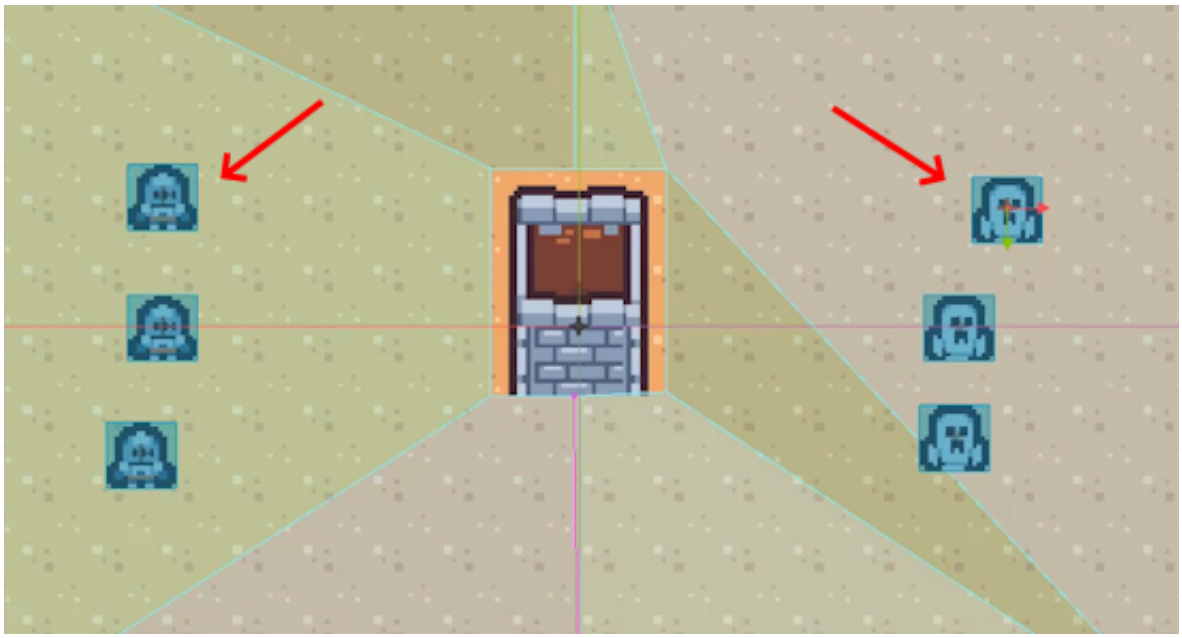
Using the Enemy AI

That completes everything for our basic enemy AI. If you press **play** to test the game and walk your *Player Unit* within the *Enemy Unit's* range, you will see it chase the *Player Unit* before it is in the attack range, and then the *Units* will fight.

To make the game more fun, we can adjust the **Move Speed** property of the **Enemy Unit** to make it slightly slower than our *Player Unit*. A value around **35** will make sure the player can move their *Units* away from the enemies if they are careful. We can also change the **Health** value of the *Enemy* to **70**.



Because our *Units* are set up as scenes, we can also duplicate both our *Player Units* and *Enemy Units* around the environment.



That completes our implementation of the enemy AI. Feel free to expand upon this to create more complex AIs or multiple-unit selections, the options are endless!

Congratulations! You have completed our course on creating a real-time strategy game in Godot. The systems that we have created here can act as a pretty good foundation for creating the game you want to create. You can take the lessons learned in this project to expand into new projects or to continue to create more features for this game. You may want to expand upon the enemy AI or add the ability to select multiple units. The options are limitless.

Course Overview

We began this course by focussing on the *Units*, and specifically the *Player Units*. These had the functionality to move around using AI navigation along with the ability to handle selections using the mouse buttons.

We then moved on to our *Enemy Units*, implementing a basic AI that detected whether a *Player Unit* was within a certain range, and targeting that unit if it is.

About Zenva

Zenva is an online learning academy with over 1 million students. We offer a wide range of courses for people who are just starting out or for people who want to learn something new. The courses are also very versatile, allowing you to learn in whatever way you want. You can choose to view the online video tutorials, read the included summaries, or follow along with the instructor using the included course files.

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

Unit.gd

Found in the Project root folder

This script defines a 'Unit' type in a 2D game in Godot, with customizable properties (like health or speed). It allows Units to navigate, attack other units, and be attacked. When a Unit's health reaches 0, it disappears from the game scene. It also includes a visual flourish: the Unit flashes red when it takes damage.

```
extends CharacterBody2D
class_name Unit

@export var health : int = 100
@export var damage : int = 20

@export var move_speed : float = 50.0
@export var attack_range : float = 20.0
@export var attack_rate : float = 0.5
var last_attack_time : float

var target : CharacterBody2D

var agent : NavigationAgent2D
var sprite : Sprite2D

@export var is_player : bool

func _ready ():
  agent = $NavigationAgent2D
  sprite = $Sprite

  var gm = get_node("/root/Main")

  # add our unit to either the player or enemies array
  if is_player:
    gm.players.append(self)
  else:
    gm.enemies.append(self)

func _physics_process(delta):
  if agent.is_navigation_finished():
    return

  var direction = global_position.direction_to(agent.get_next_path_position())
  velocity = direction * move_speed

  move_and_slide()
```



```
func _process(delta):
  _target_check()

# called every frame
# stop moving when in attack range and begin attacks
# otherwise chase after target
func _target_check ():
  if target != null:
    var dist = global_position.distance_to(target.global_position)

    if dist <= attack_range:
      agent.target_position = global_position
      _try_attack_target()
    else:
      agent.target_position = target.global_position

# called when we want the unit to move to a position
func move_to_location (location):
  target = null
  agent.target_position = location

# called when we want the unit to chase after a target
func set_target (new_target):
  target = new_target

# if we are able to attack then deal damage to target
func _try_attack_target ():
  var cur_time = Time.get_unix_time_from_system()

  if cur_time - last_attack_time > attack_rate:
    target.take_damage(damage)
    last_attack_time = cur_time

# called when an enemy deals damage to us
func take_damage (damage_to_take):
  health -= damage_to_take

  if health <= 0:
    queue_free()

  # flash the sprite red for 0.1 seconds
  sprite.modulate = Color.RED
  await get_tree().create_timer(0.1).timeout
  sprite.modulate = Color.WHITE
```

PlayerUnit.gd

Found in the Project root folder

This script manages a ‘SelectionVisual’ for a ‘Unit’ in Godot, which marks if the unit is selected or not. It includes a function ‘toggle_selection_visual’ to make the selection visual visible or invisible based on the variable ‘toggle’.


```
extends Unit

@onready var selection_visual : Sprite2D = get_node("SelectionVisual")

# enable/disable the selection visual
func toggle_selection_visual (toggle):
  selection_visual.visible = toggle
```

EnemyUnit.gd

Found in the Project root folder

This script extends a 'Unit' class with enemy behavior in Godot. It scans for player units within a certain detection range. If a player unit is found within this range, it gets set as the target for this enemy unit, who will then attempt to close the range.

```
extends Unit

@export var detect_range : float = 100.0
@onready var game_manager = get_node("/root/Main")

# loop through each player unit
# if a player is within the detect range - set them as our target
func _process(delta):
  if target == null:
    for player in game_manager.players:
      if player == null:
        continue

      var dist = global_position.distance_to(player.global_position)

      if dist <= detect_range:
        set_target(player)

  _target_check()
```

GameManager.gd

Found in the Project root folder

This script, for a "Main" node in a Godot game, manages the player's interactions with units. If a unit is left clicked, it gets selected. If an enemy is right clicked while a unit is selected, that enemy is targeted. If the right click happens anywhere else, the selected unit moves to that position.

```
extends Node2D

var selected_unit : CharacterBody2D
var players : Array[CharacterBody2D]
var enemies : Array[CharacterBody2D]

# called when an input is detected
```



```
func _input(event):
  if event is InputEventMouseButton and event.pressed:
    if event.button_index == MOUSE_BUTTON_LEFT:
      _try_select_unit()
    elif event.button_index == MOUSE_BUTTON_RIGHT:
      _try_command_unit()

# returns what we're currently hovering over
func _get_selected_unit ():
  var space = get_world_2d().direct_space_state
  var query = PhysicsPointQueryParameters2D.new()
  query.position = get_global_mouse_position()
  var intersection = space.intersect_point(query, 1)

  if !intersection.is_empty():
    return intersection[0].collider

  return null

# called when we left click
func _try_select_unit():
  var unit = _get_selected_unit()

  if unit != null and unit.is_player:
    _select_unit(unit)
  else:
    _unselect_unit()

# selects the given unit
func _select_unit (unit):
  _unselect_unit()
  selected_unit = unit
  selected_unit.toggle_selection_visual(true)

# unselects the current unit
func _unselect_unit():
  if selected_unit != null:
    selected_unit.toggle_selection_visual(false)

  selected_unit = null

# called when we right click
func _try_command_unit ():
  # if we have no selected unit - return
  if selected_unit == null:
    return

  # get the unit we're hovering over, if any
  var target = _get_selected_unit()

  # if we're right clicking an enemy, set them as our target
  if target != null and target.is_player == false:
    selected_unit.set_target(target)
  # otherwise move to our mouse position
  else:
```

```
    selected_unit.move_to_location(get_global_mouse_position())
```

