Build Your First 2D Game with Godot

Brandon Lewis

Download Art Assets and Slides at:

https://github.com/brandon-lewis/CodeMash2023-PixelDash

Download the Standard version of Godot 3.5 (not Mono)

About Me

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

Consultant, hobbyist, gamer, general geek

"IoT from Sensing to Insights Workshop"
- CodeMash 2021

brandonmlewis

brandon-lewis

What is Godot Engine?

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

- A completely free and open-source (MIT license) general-purpose 2D and 3D game engine.
- Build games or applications for desktop (Linux, MacOS, Windows), mobile (Android, iOS), and game consoles.
- Offers multiple languages for development – GDScript, C#, and GDNative which supports C++ and C.

What is Godot Engine?

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

BENEFITS

- Uses widely-supported OpenGL renderer.
- GDScript is a high-level, dynamically-typed language.
- Version-control friendly.
- Completely free and open source.
- Editor plugins are first-class citizens.

DRAWBACKS

- No Vulkan or Direct3D support until Godot 4.
- GDScript is a high-level, dynamically-typed language.
- Not as mature as the more popular engines.
- Primarily used by solo and small-team indie developers.

Notable games/apps made with Godot

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

Dungeondraft

Brotato

(20.2k Steam reviews, 97% positive)

Dome Keeper

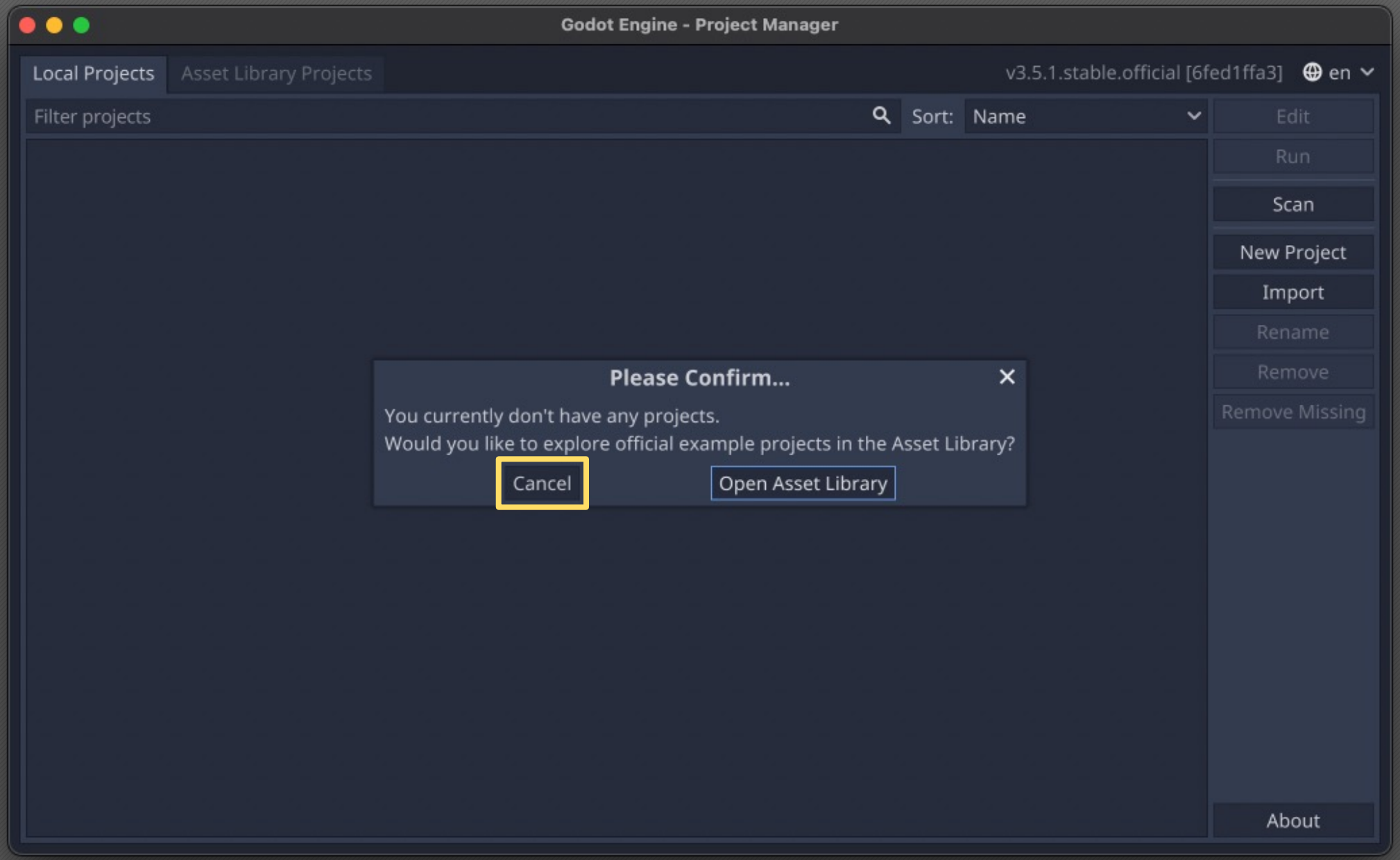
(4.8k Steam reviews, 90% positive)

Session Goals

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

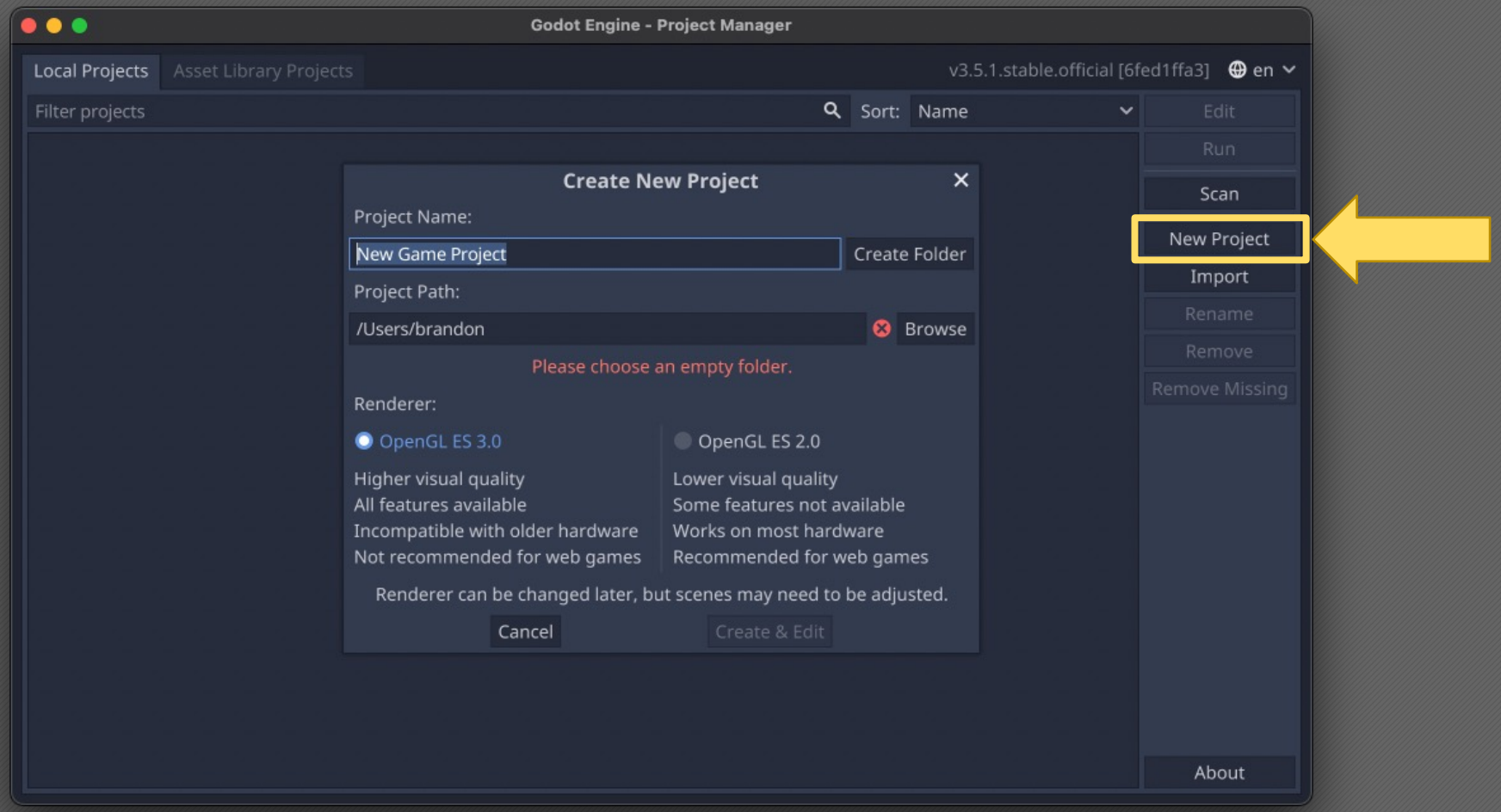
1. Controllable Player
2. Reusable Base Level
3. Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. Level Checkpoints
5. Hazards & Player Respawn
6. Collectibles & HUD
7. A Basic Enemy
8. Main Menu & Changing Scenes
9. Open Workshop, Tinker Time

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



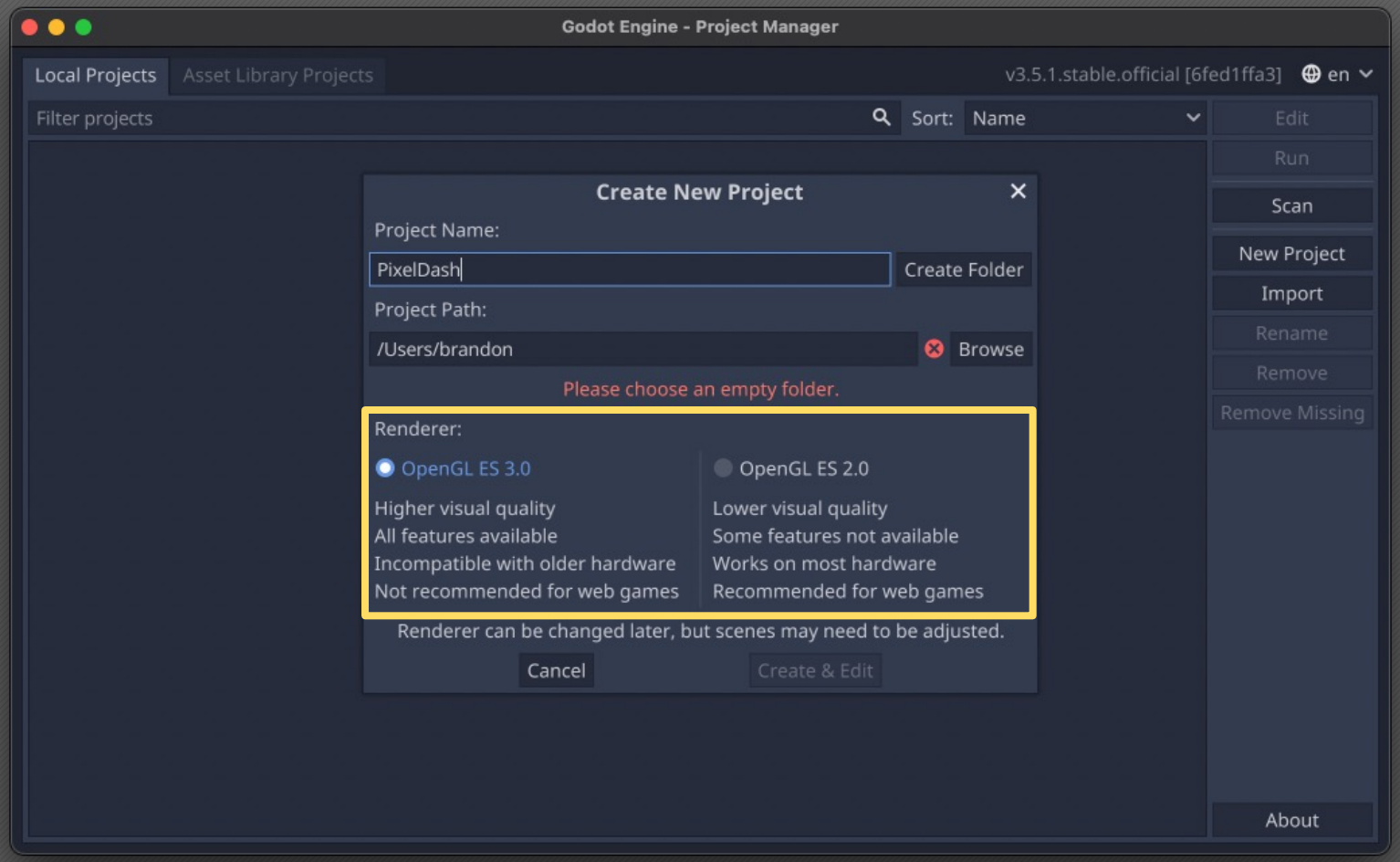
Create a New Project

Session Resources: *https://github.com/brandon-lewis/CodeMash2023-PixelDash*



Create a New Project

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



OpenGL ES 3.0

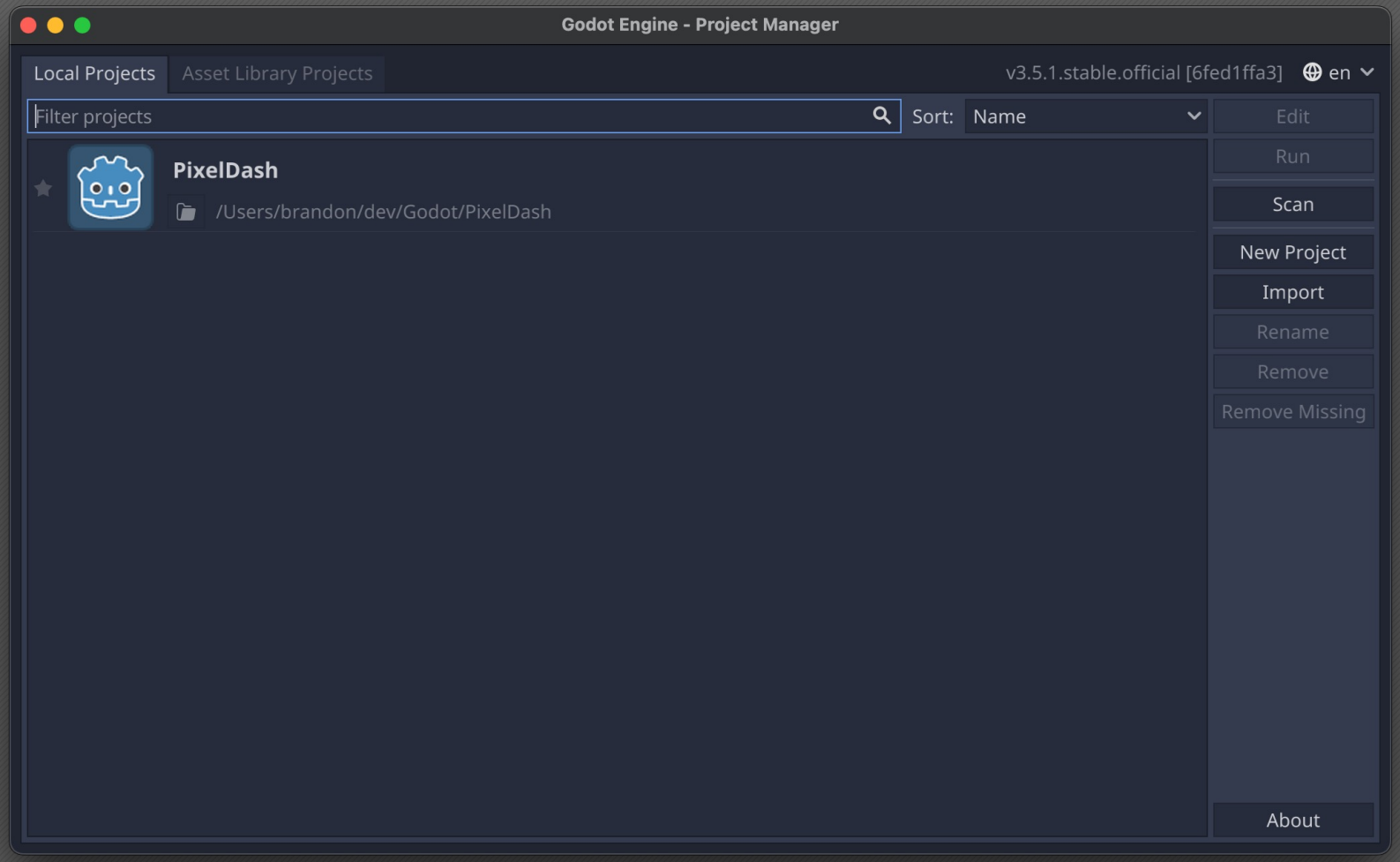
- Offers better shader features and performance.
- Risky compatibility for games on mobile devices.

OpenGL ES 2.0

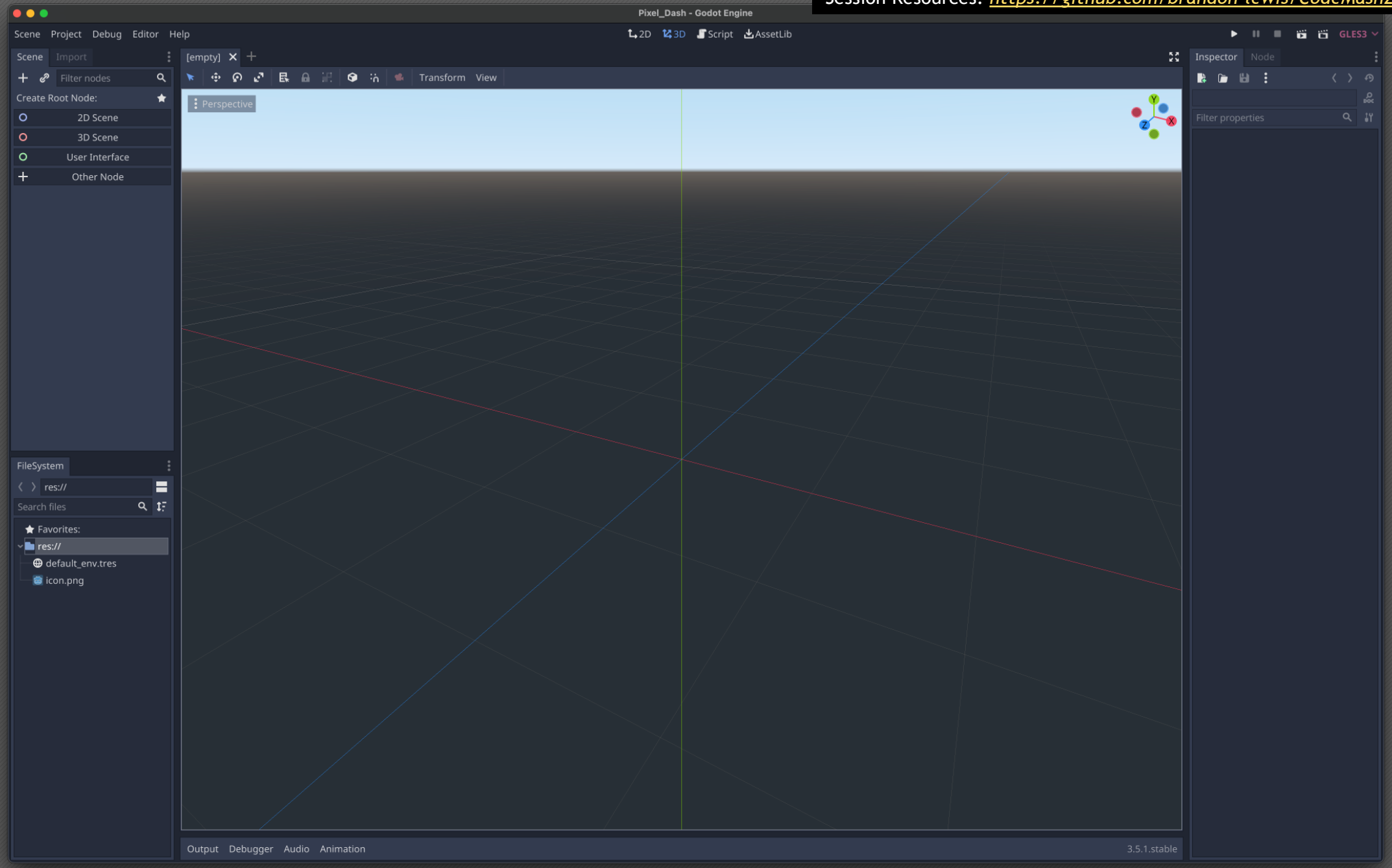
- No GPU particles (can use CPUParticles instead).
- Incapable of HDR rendering.
- Lack of post-processing features.

Exploring the Editor

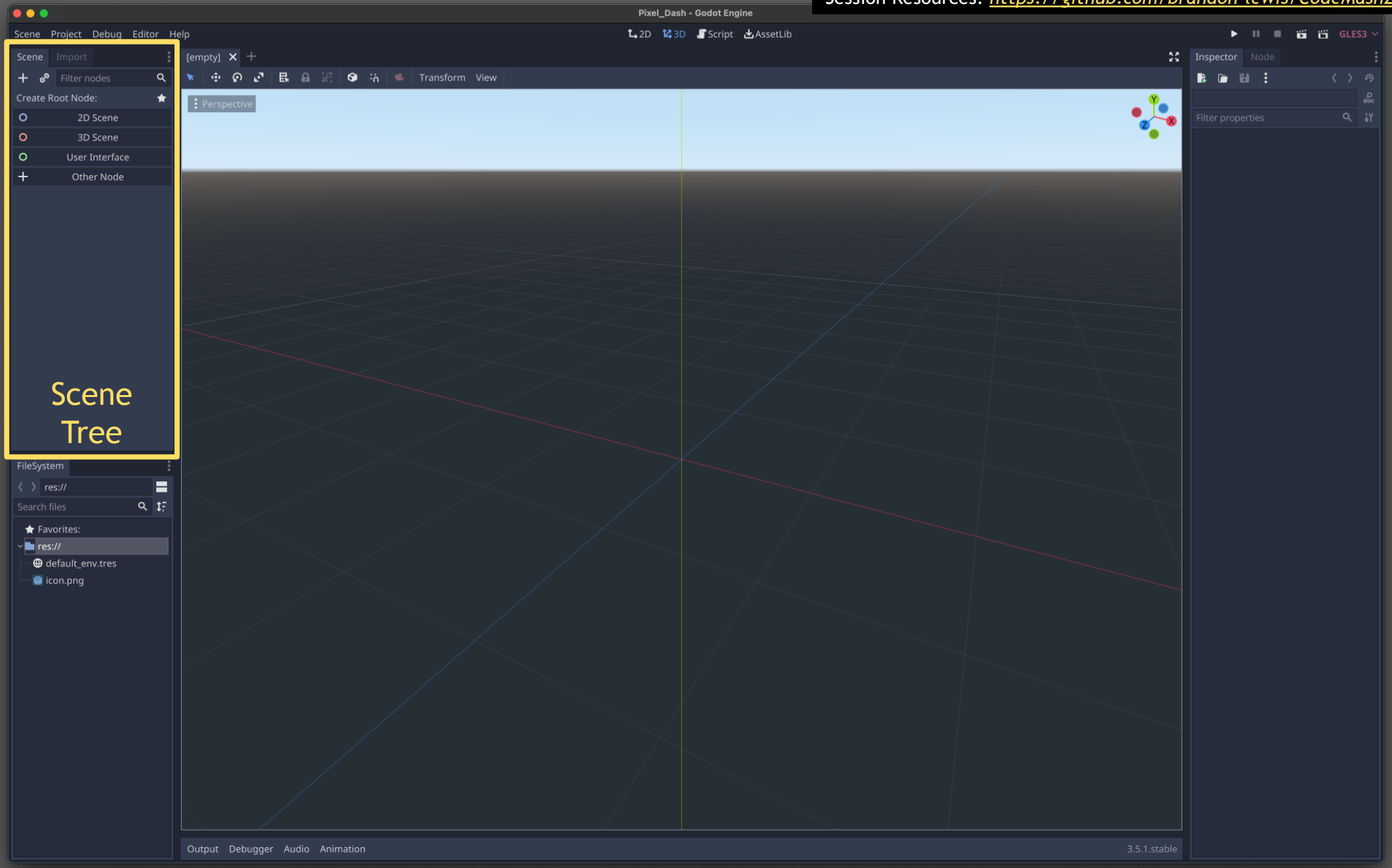
Session Resources: *https://github.com/brandon-lewis/CodeMash2023-PixelDash*



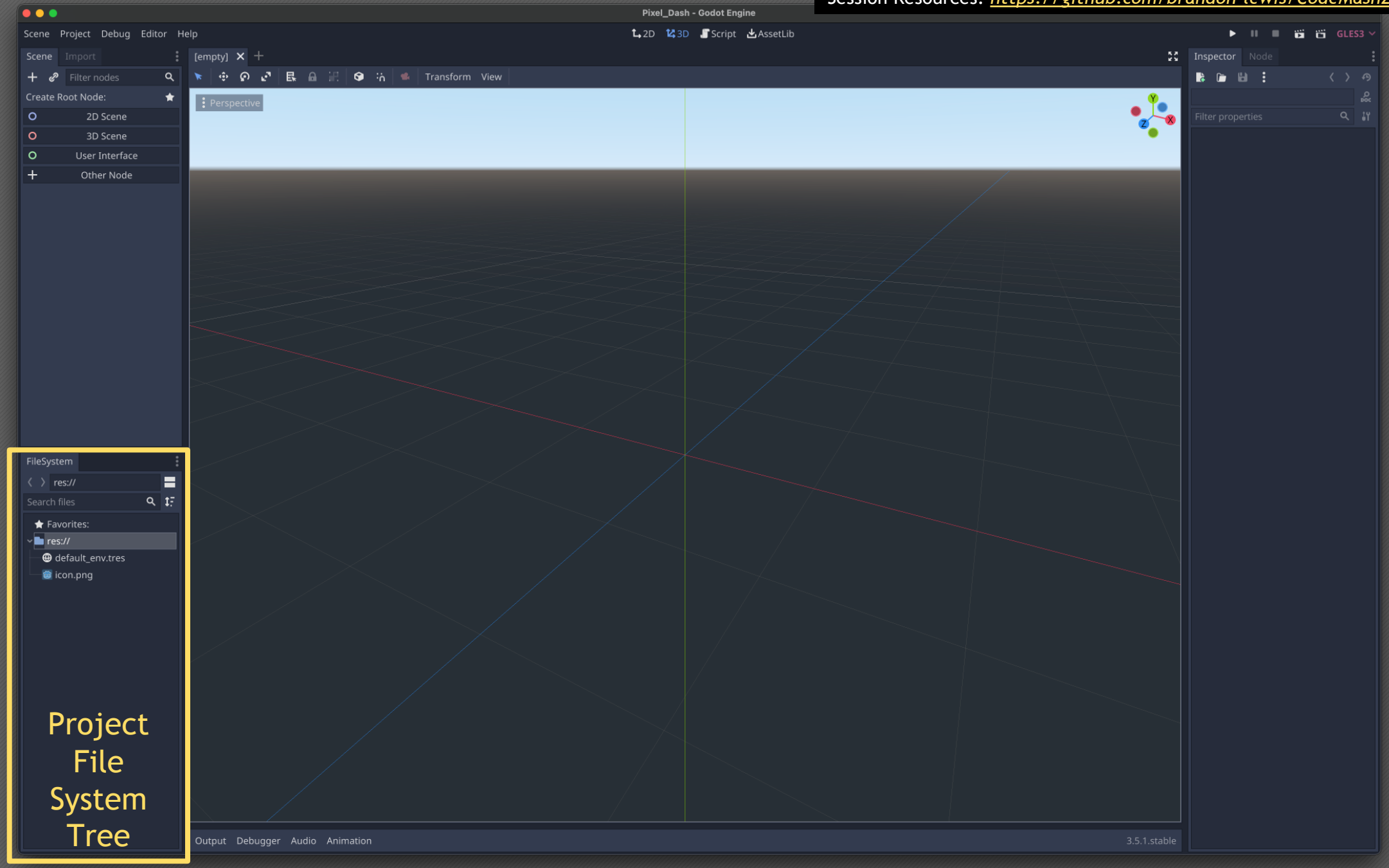
Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



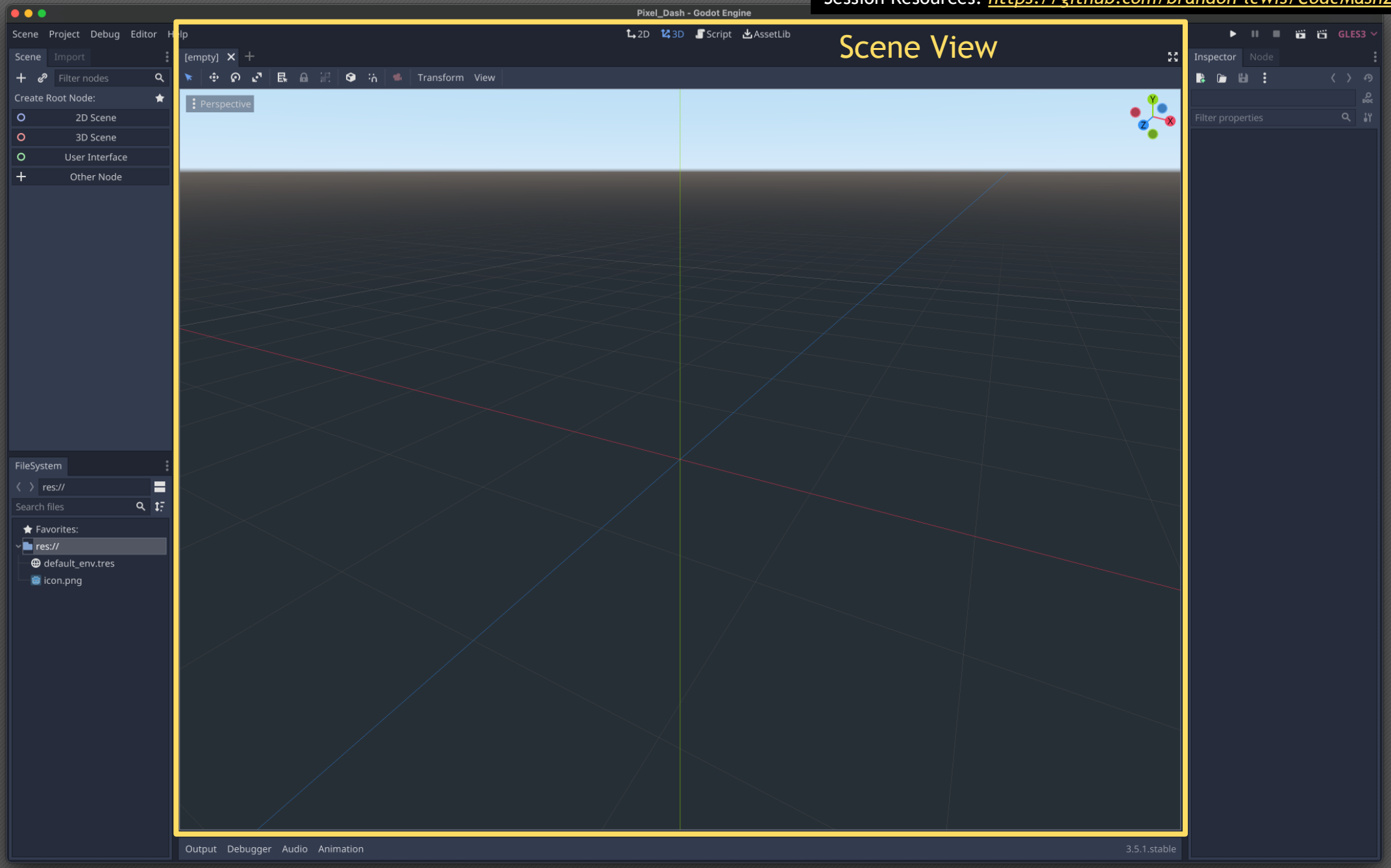
Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



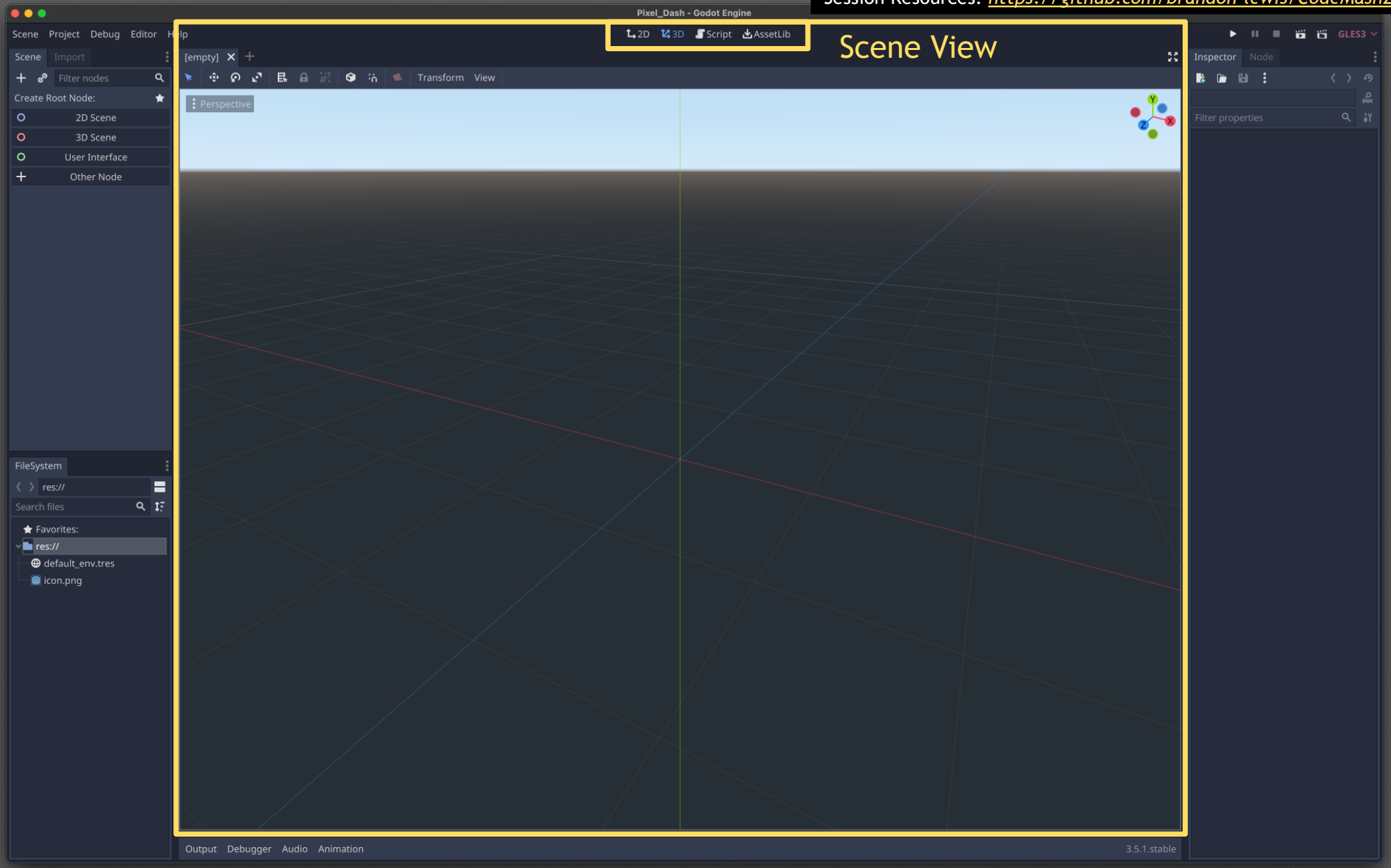
Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

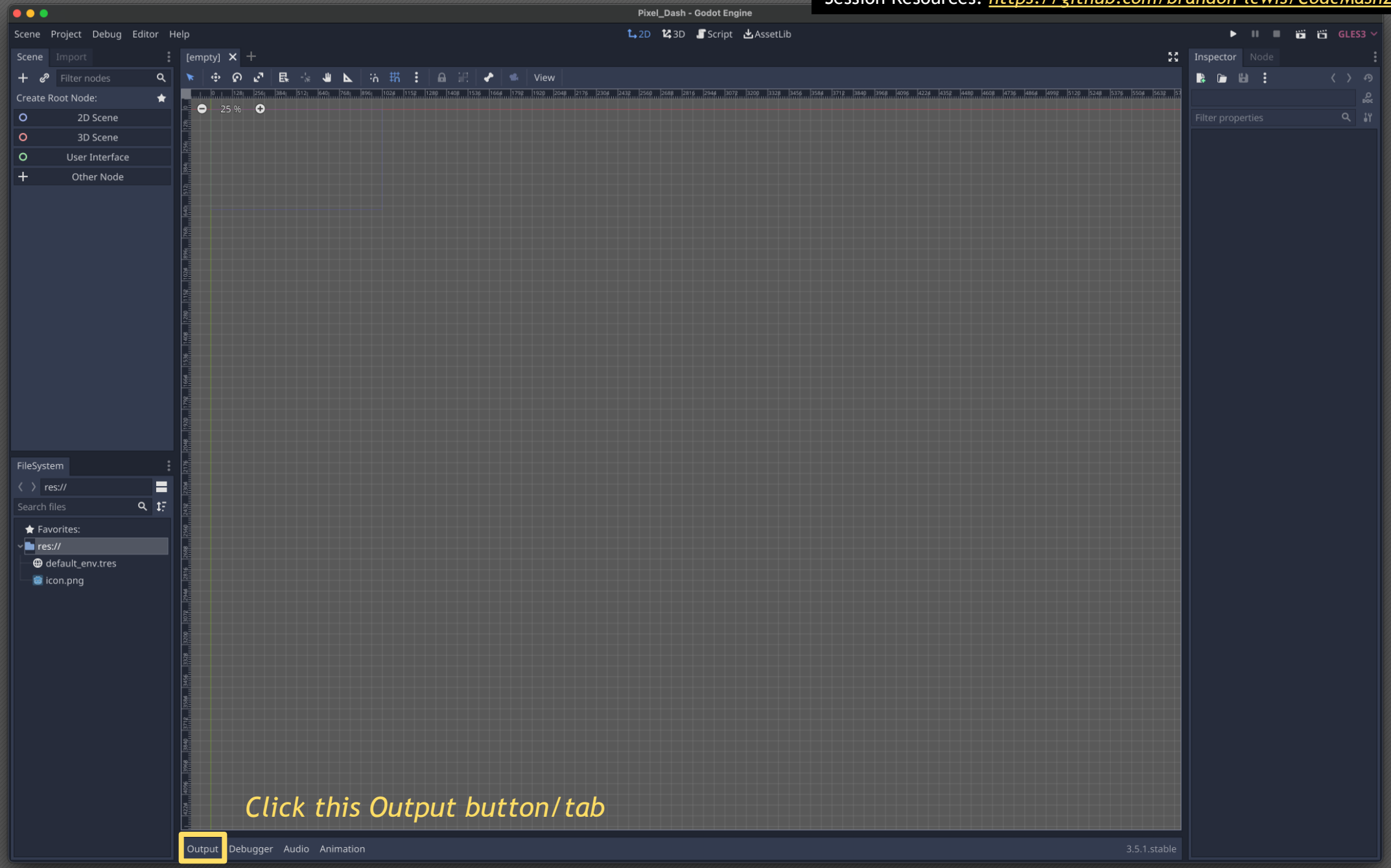
Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

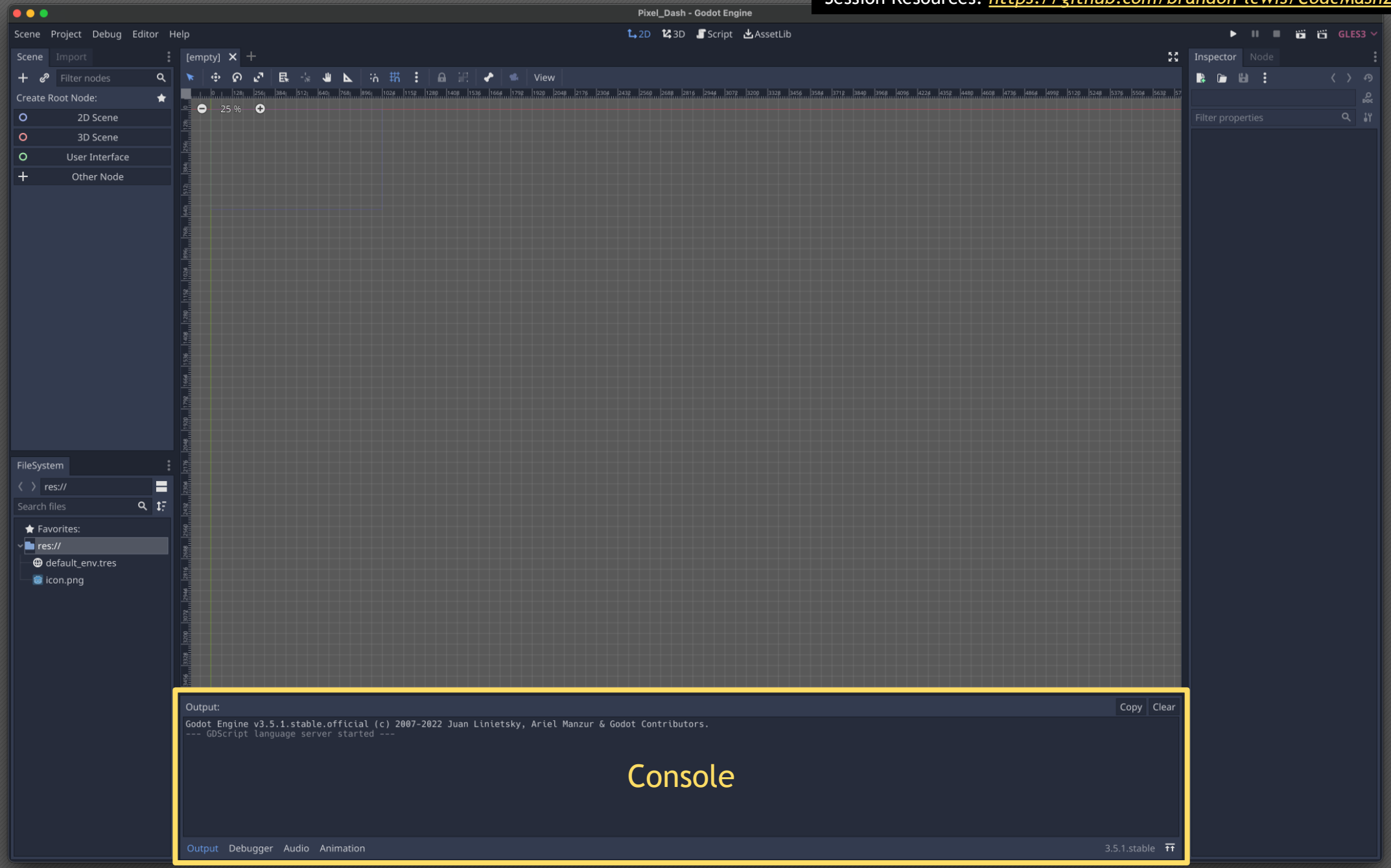
Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

Scene Zoom

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

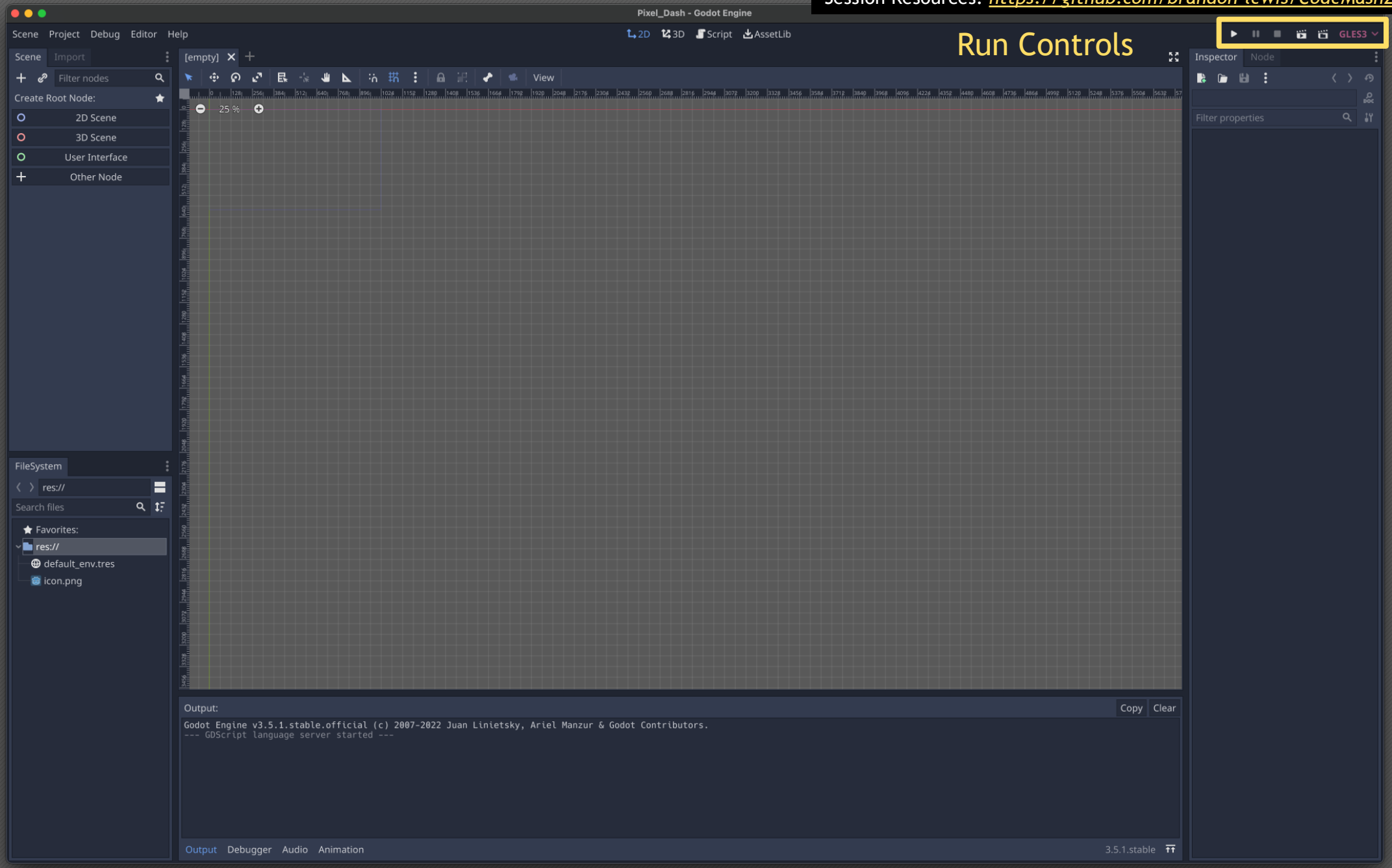


Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



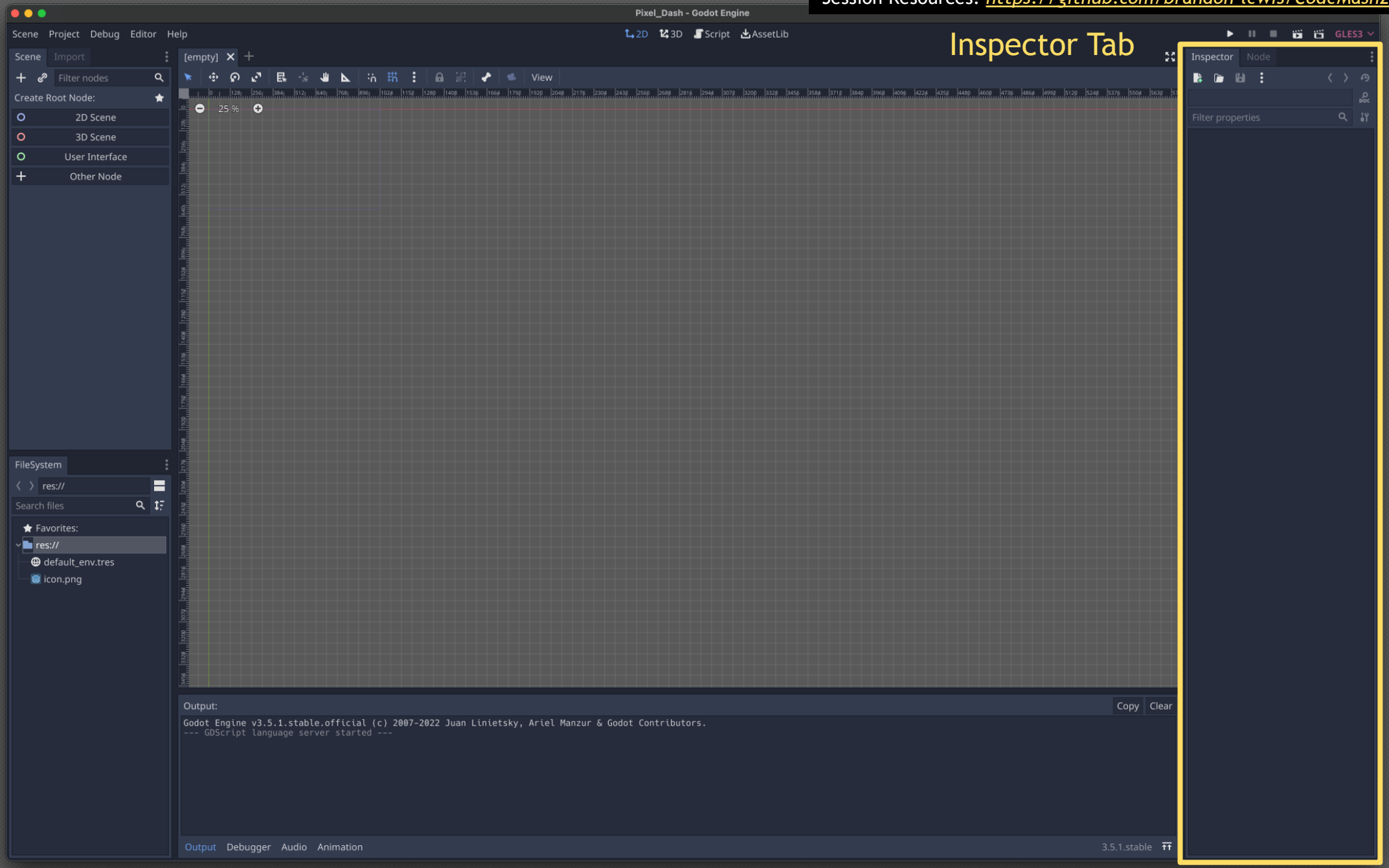
Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

Run Controls

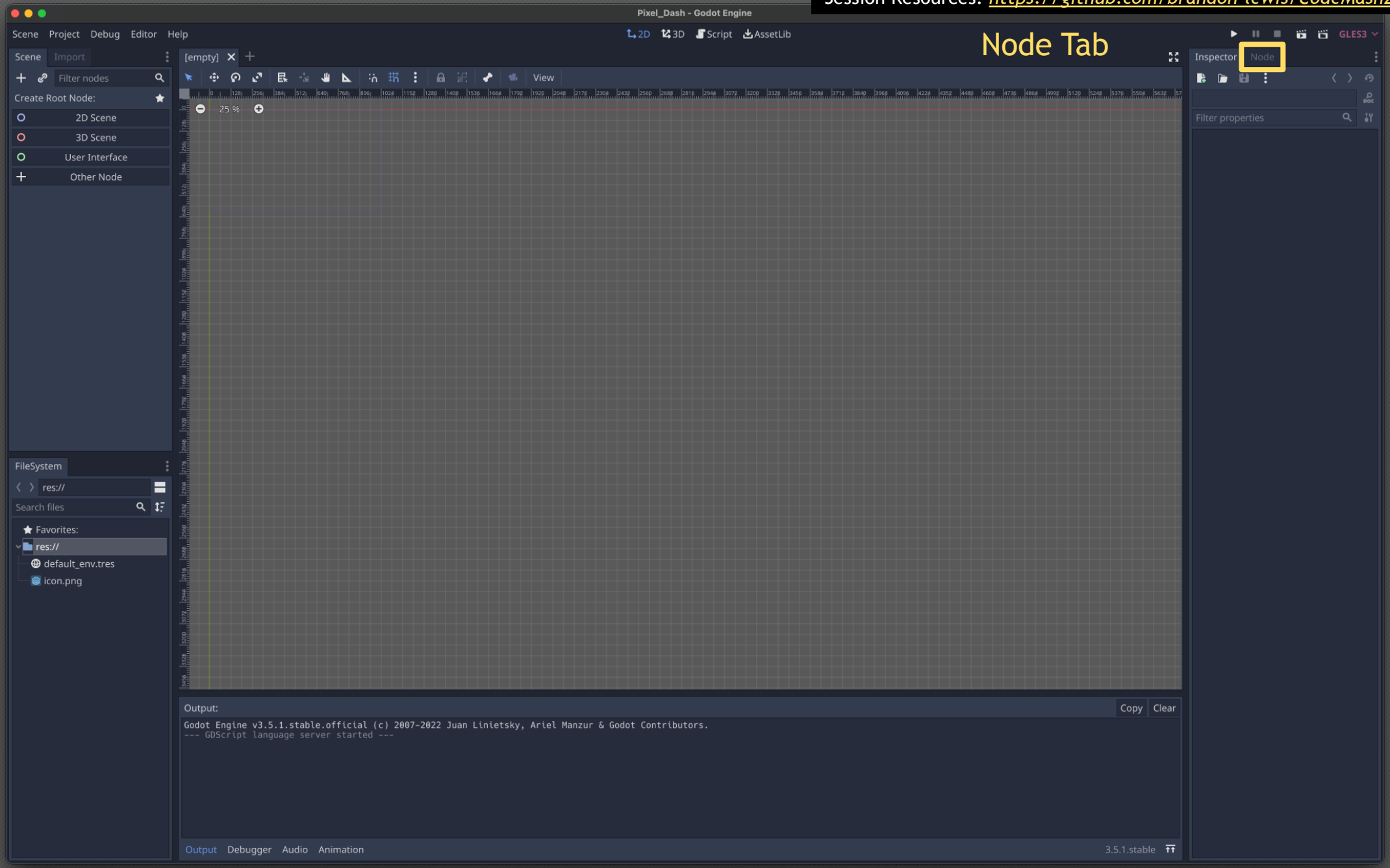


Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash

Inspector Tab

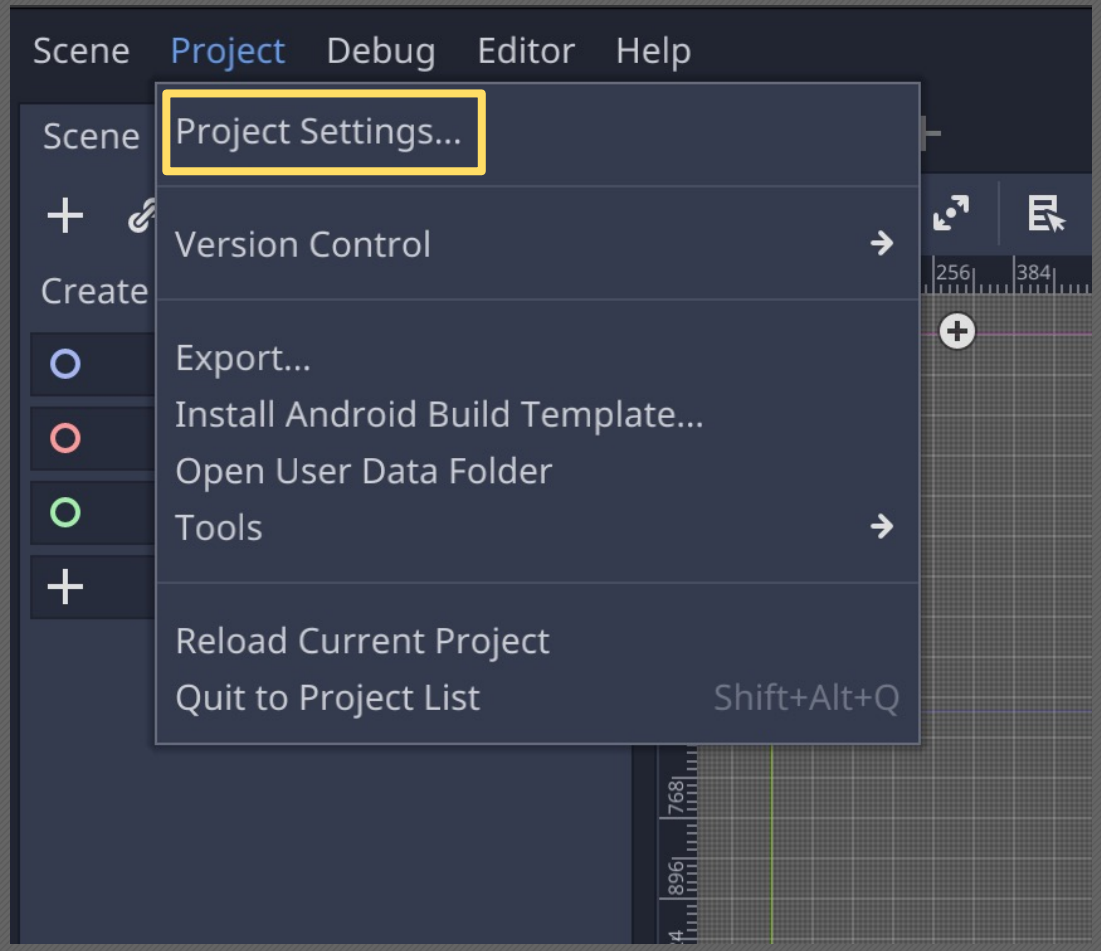


Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



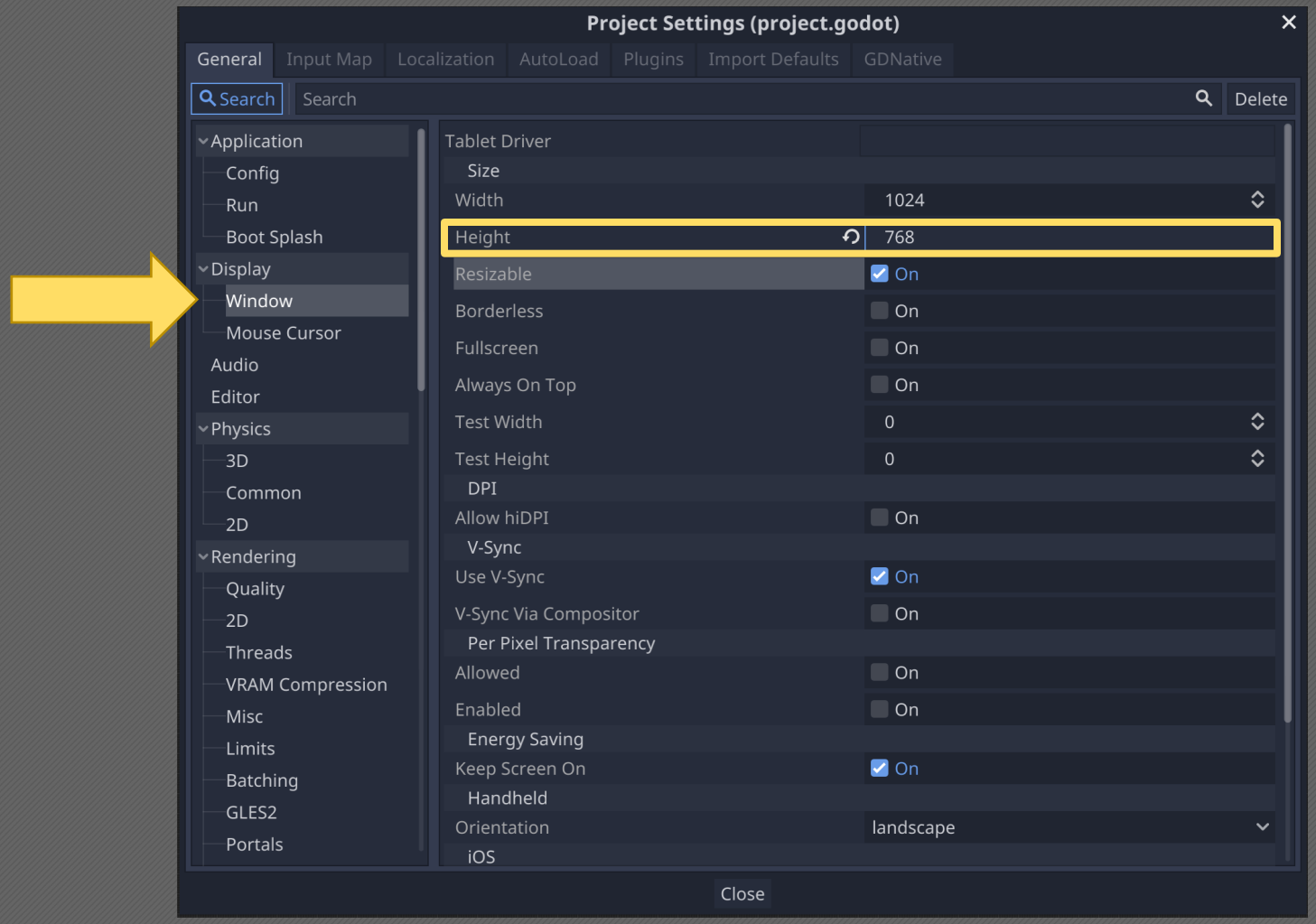
Updating Basic Project Settings

Session Resources: *https://github.com/brandon-lewis/CodeMash2023-PixelDash*



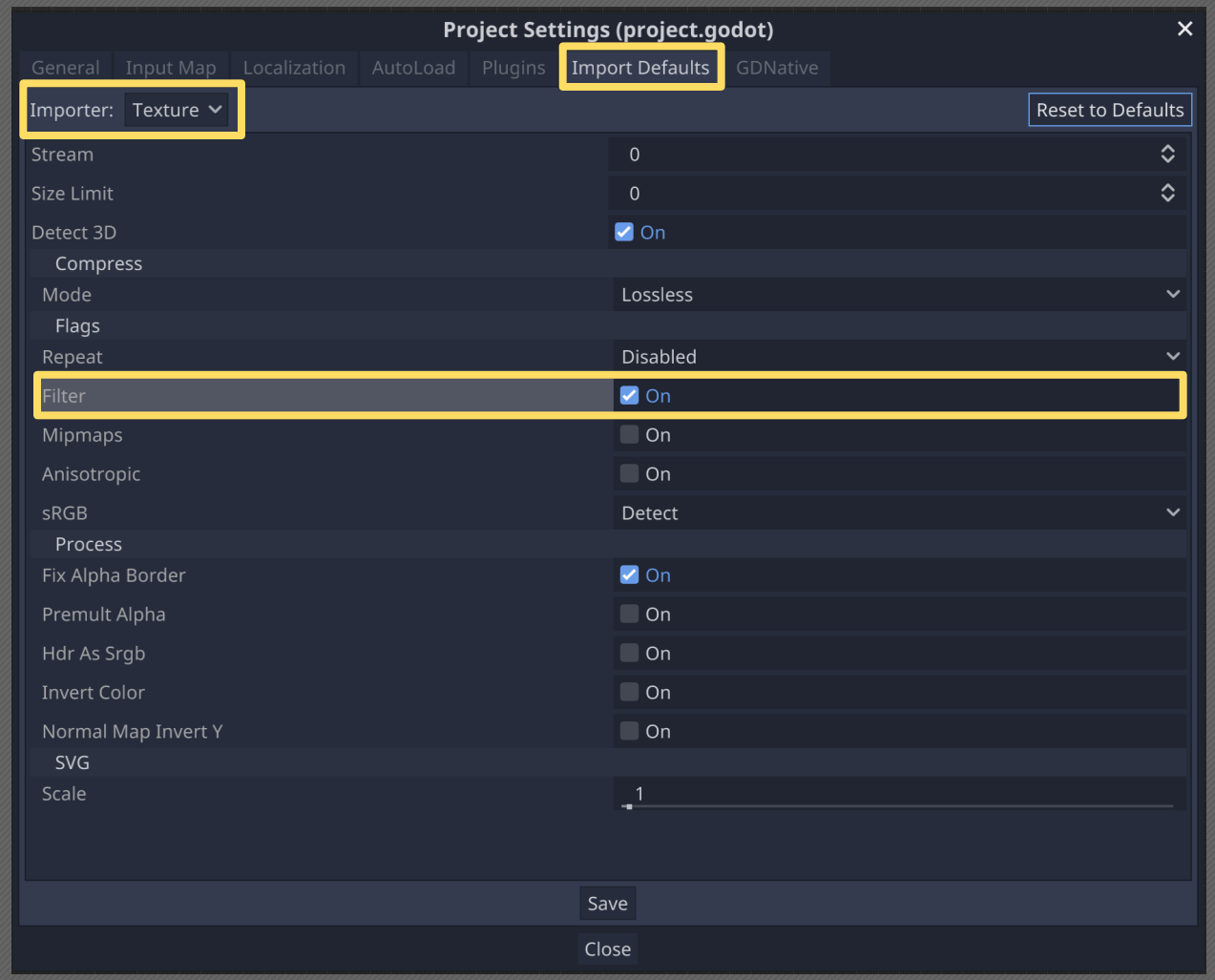
Set Game Window Size to 1024x768

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



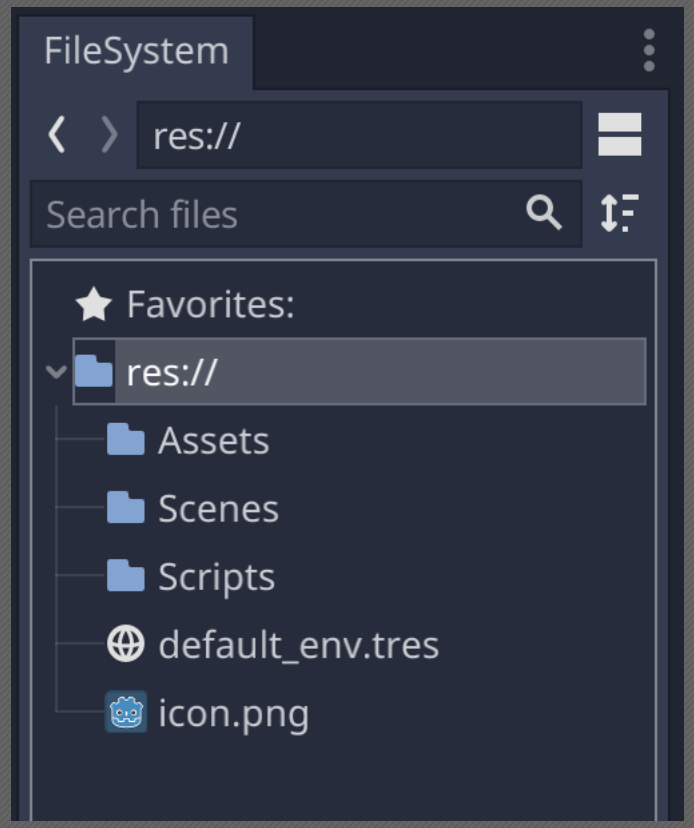
Turn Texture Filtering “Off”

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



Project Folder Structure

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



res:// is a relative "Resource" path representing the root file system path of your project.

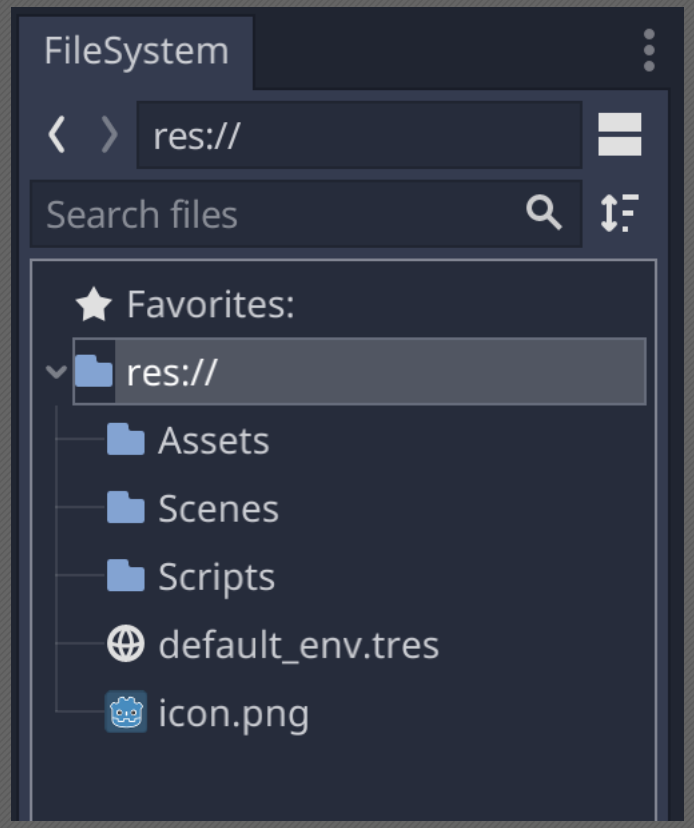
All project files exist in res://

Create the following folders:

- Assets
- Scenes
- Scripts

Importing Files & Assets

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



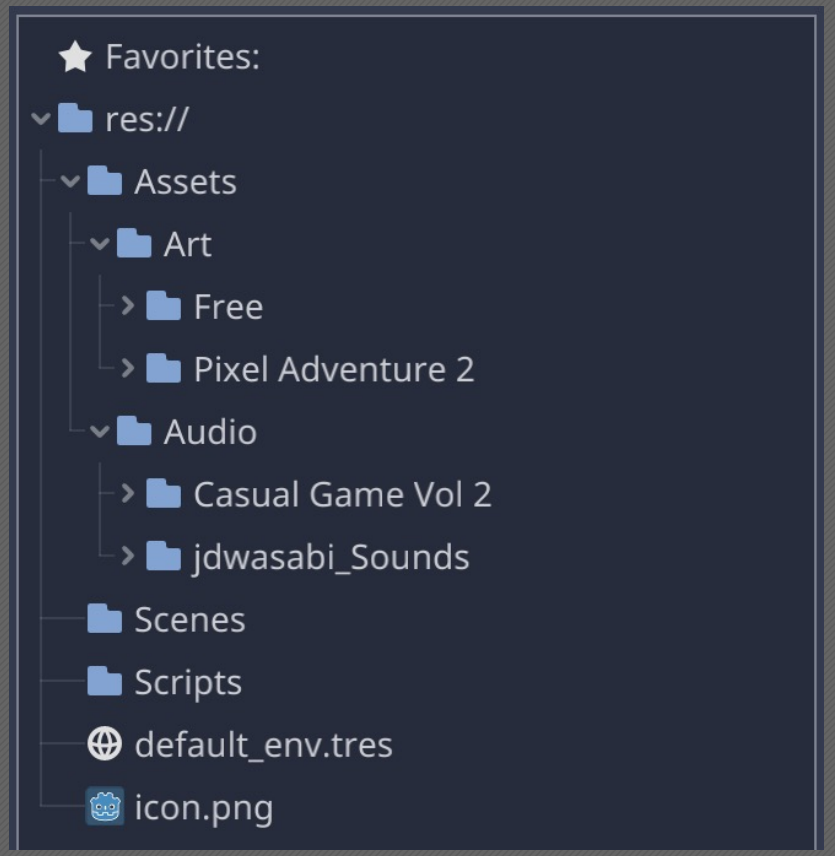
To import, you can simply drag-and-drop files into the FileSystem tab from your OS file explorer.

Note: Files imported via drag-and-drop into the FileSystem tab will import the files into the currently selected folder.

WARNING: If you end up moving project files around, it's best to move those files using the FileSystem tab.

Import the Art & Audio Assets

Session Resources: https://github.com/brandon-lewis/CodeMash2023-PixelDash



Import the assets downloaded from the GitHub page into the project.

Your project structure does not have to match perfectly, as we will not be accessing these assets via resource paths.

Note: Godot will create a *.import* file for every imported file to keep track of that file's settings.

Lab Time (~5 Minutes)

https://github.com/brandon-lewis/CodeMash2023-PixelDash

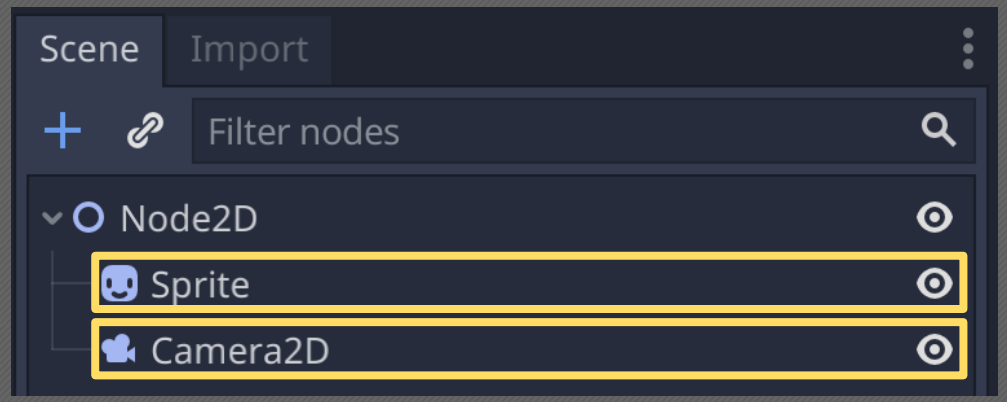
- Download workshop dependencies.
- Create the new project.
- Update the project settings.
- Create base folders and import assets.
- Click around and get a feel for the editor.

Session Goal #1: Controllable Player

1. Controllable Player
2. Reusable Base Level
3. Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. Level Checkpoints
5. Hazards & Player Respawn
6. Collectibles & HUD
7. A Basic Enemy
8. Main Menu & Changing Scenes
9. Open Workshop, Tinker Time

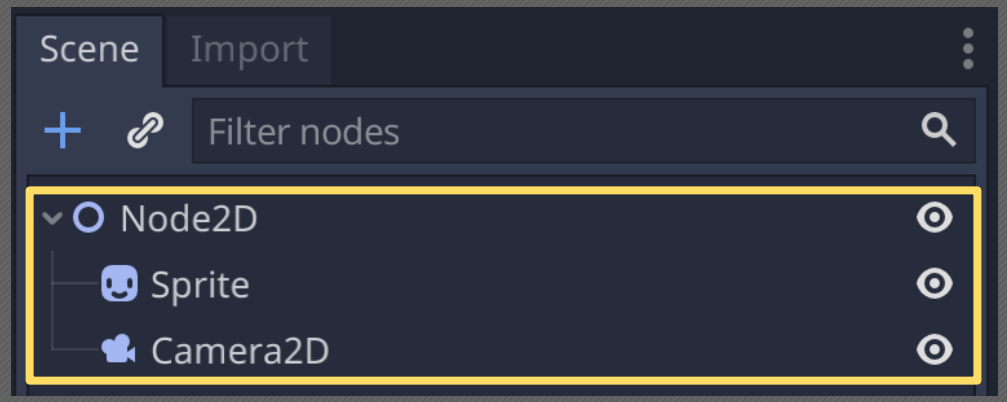
Defining Nodes and Scenes

- A **Node** is a fundamental building block of a scene (akin to a single LEGO brick).
- A **Scene** is a tree of nodes. Every scene has exactly one root node.
- When we run the **Game**, we are always executing at least one top-level Scene.
- A scene can be imported as a node in any other scene.



Defining Nodes and Scenes

- A **Node** is a fundamental building block of a scene (akin to a single LEGO brick).
- A **Scene** is a tree of nodes. Every scene has exactly one root node.
- When we run the **Game**, we are always executing at least one top-level Scene.
- A scene can be imported as a node in any other scene.



Pixel_Dash - Godot Engine

Scene Project Debug Editor Help

2D 3D Script AssetLib

GLES3

Scene Import

Filter nodes

Create Root Node:

2D Scene

3D Scene

User Interface

Other Node

[empty]

View

25 %

Inspector Node

Filter properties

FileSystem

res://

Search files

Favorites:

res://

default_env.tres

icon.png

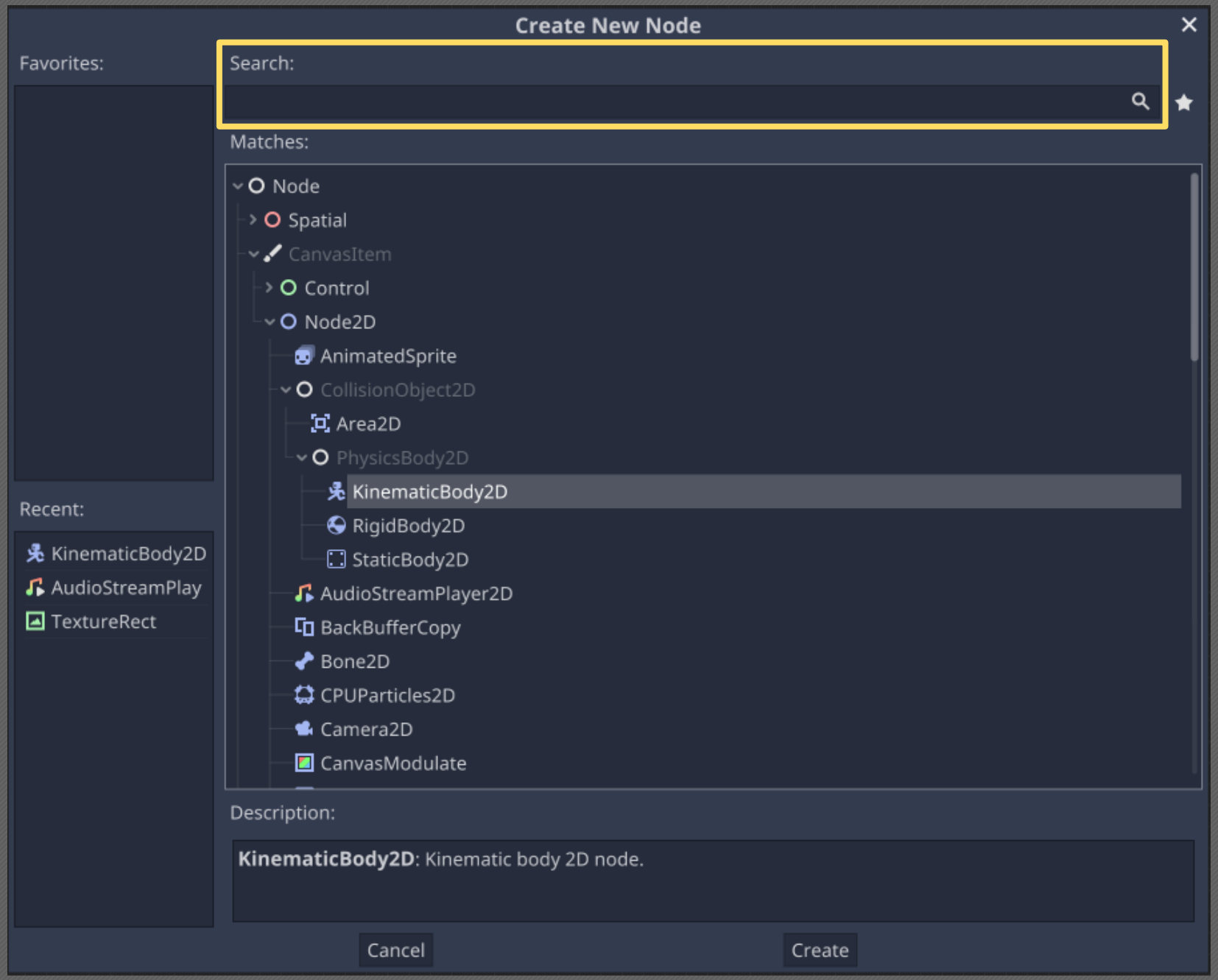
Output:

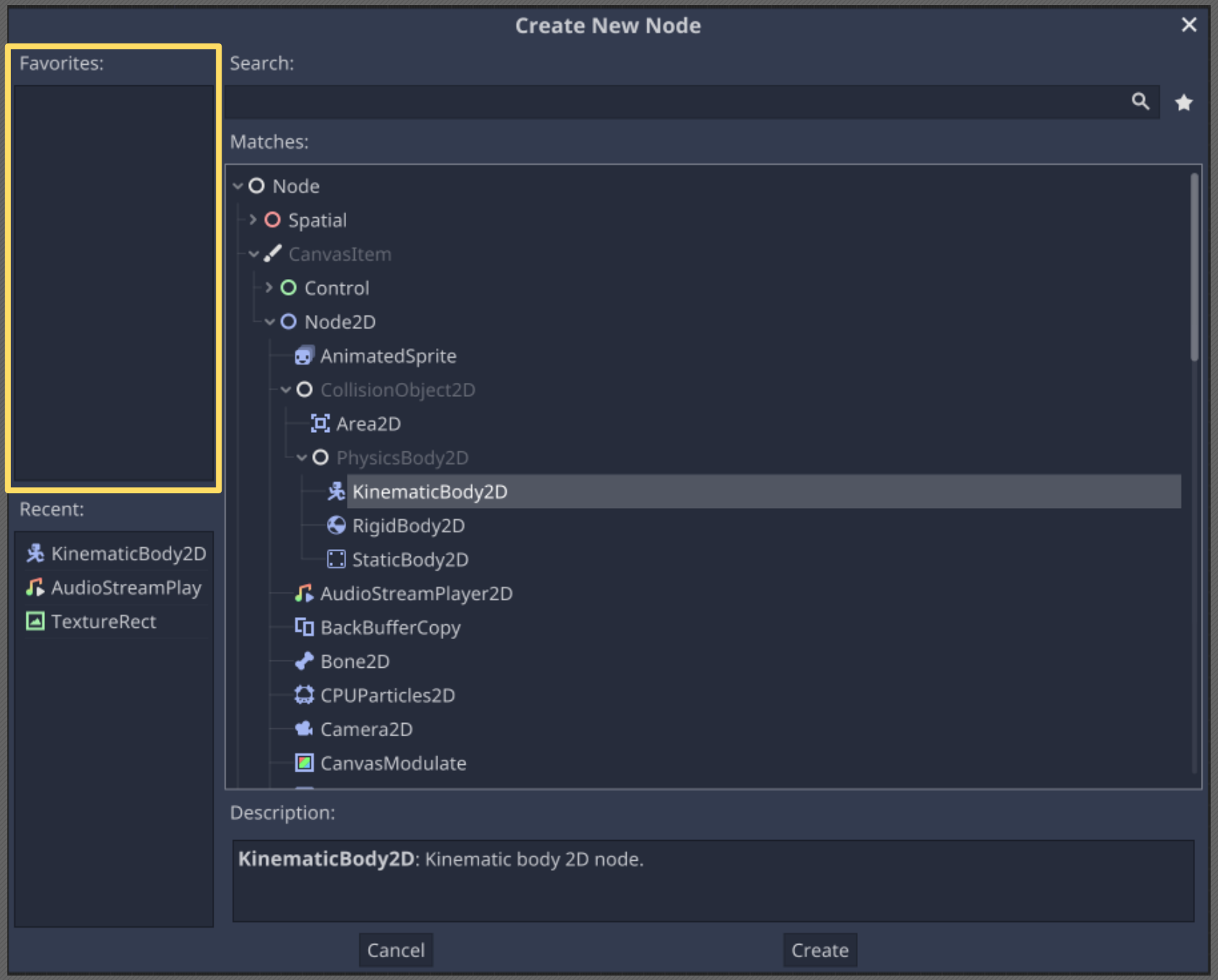
Copy Clear

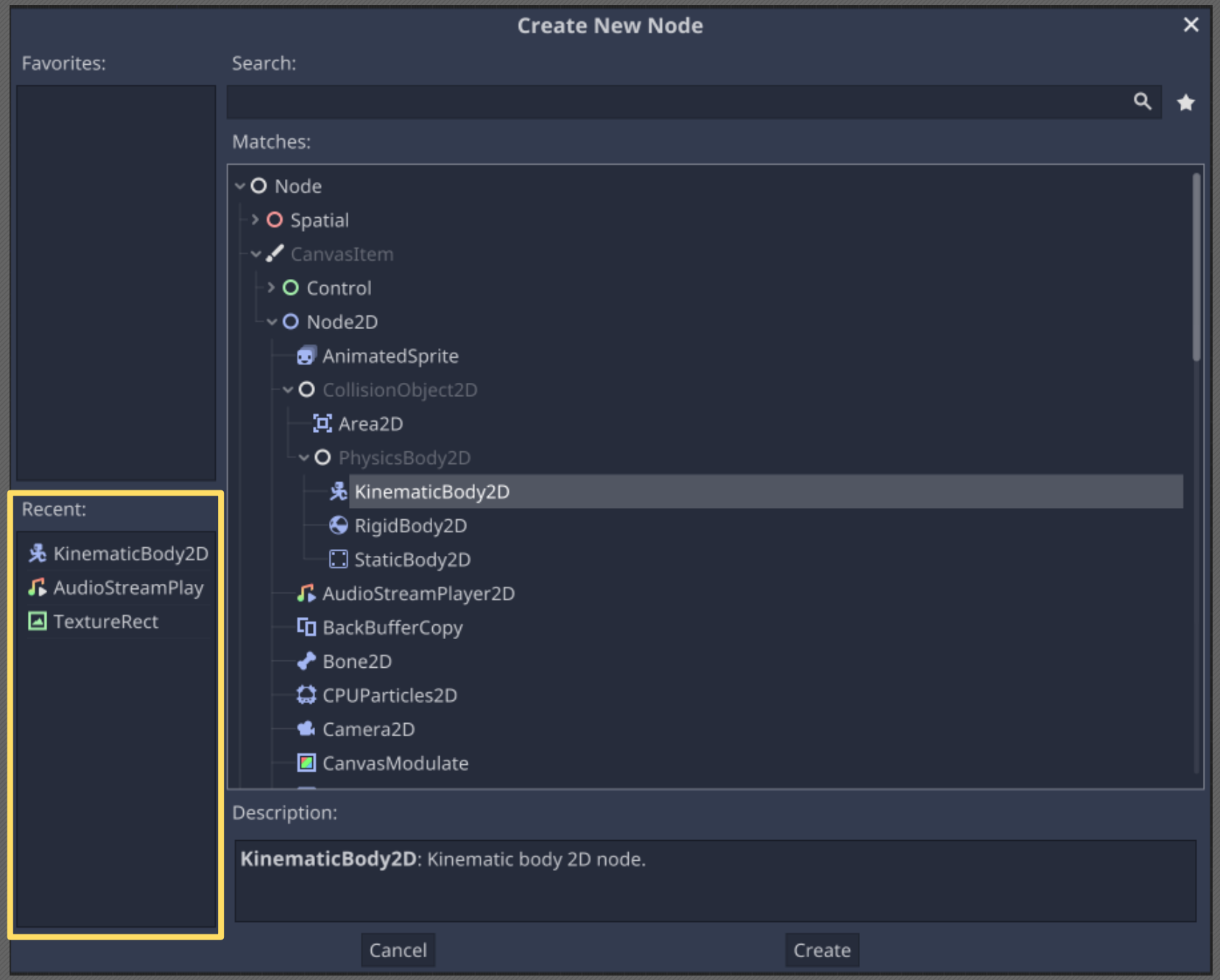
```
Godot Engine v3.5.1.stable.official (c) 2007-2022 Juan Linietsky, Ariel Manzur & Godot Contributors.
--- GDScript language server started ---
```

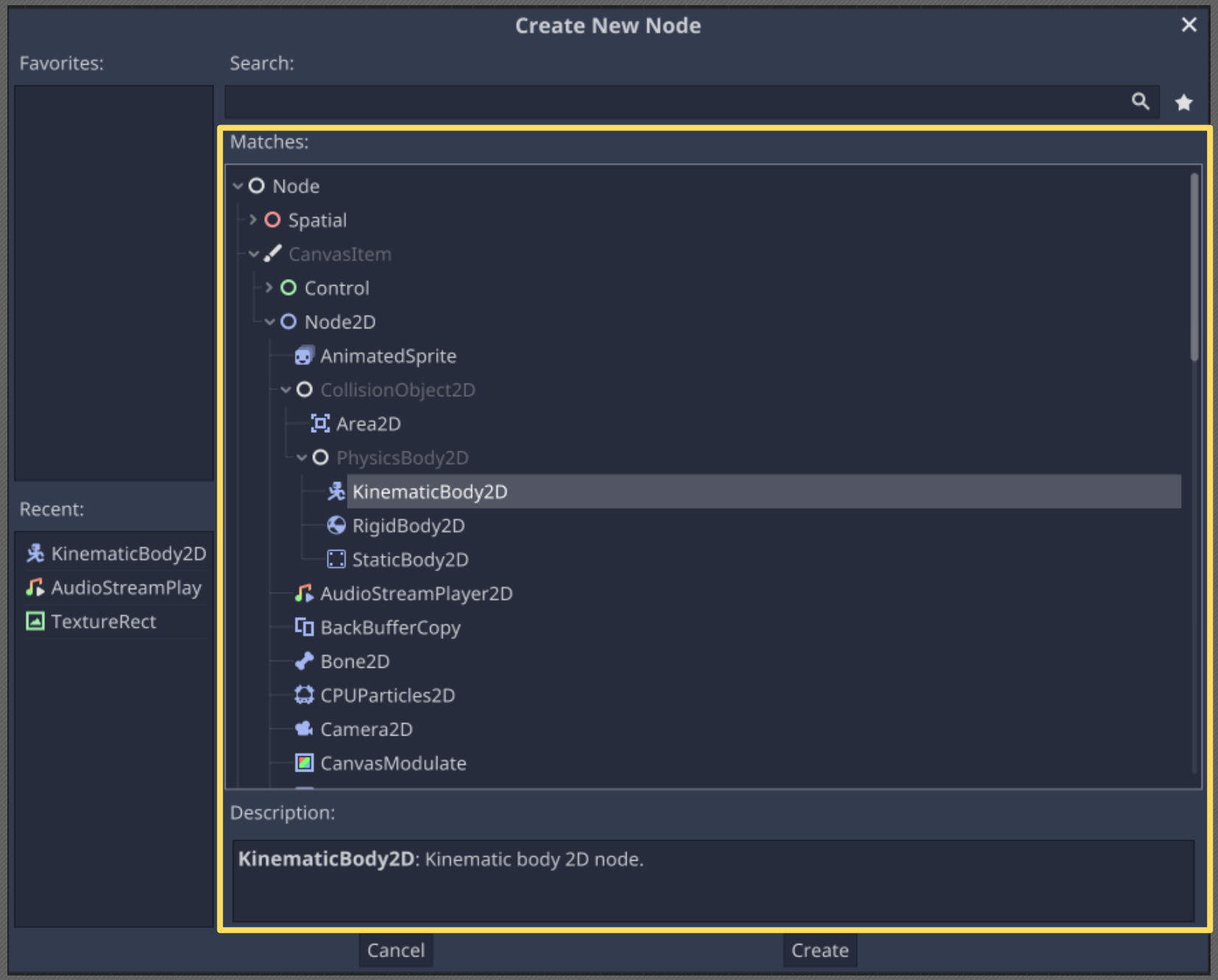
Output Debugger Audio Animation

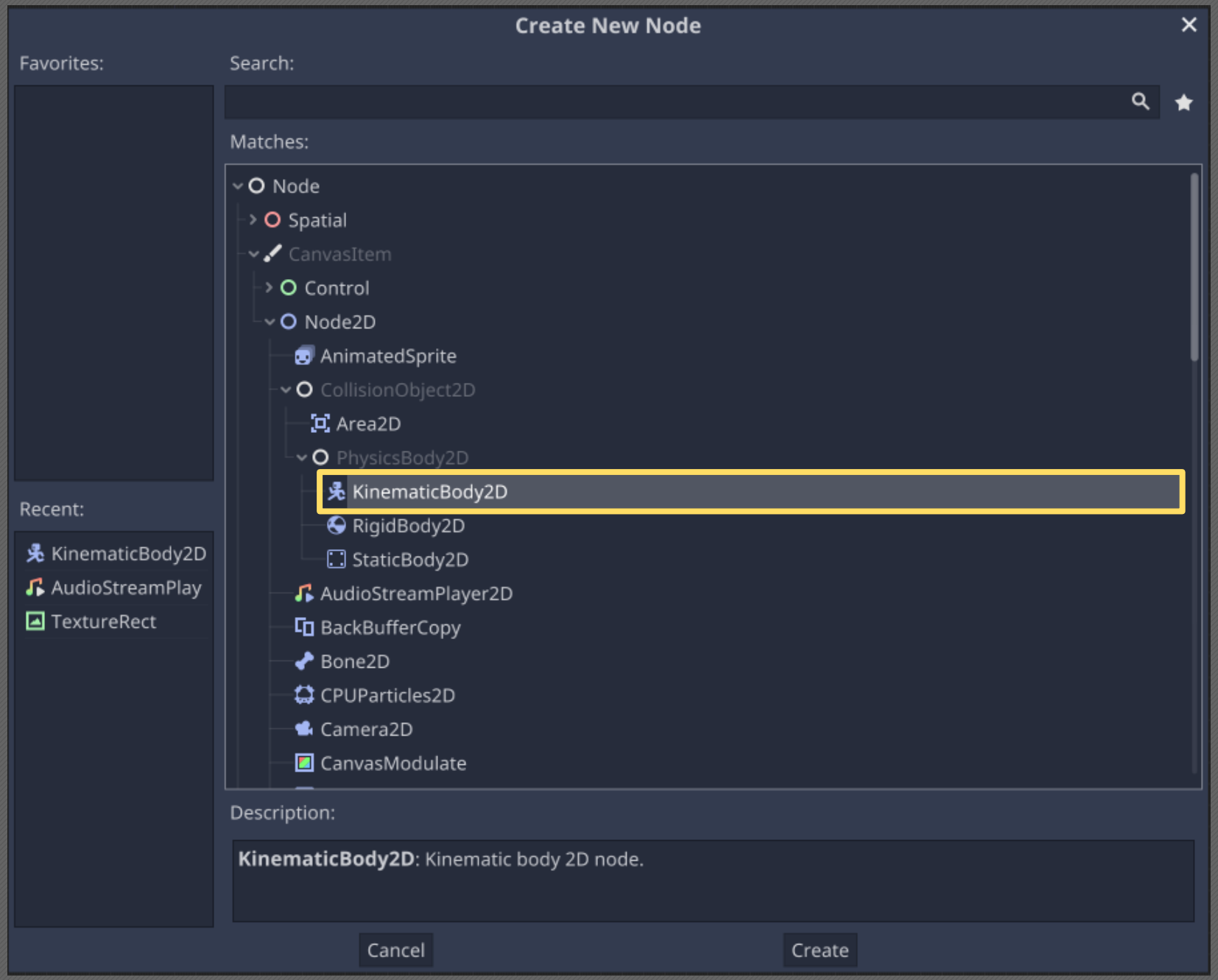
3.5.1.stable


Create New Node
Favorites:
Search:
Matches:
Node
Spatial
CanvasItem
Control
Node2D
AnimatedSprite
CollisionObject2D
Area2D
PhysicsBody2D
KinematicBody2D
RigidBody2D
StaticBody2D
AudioStreamPlayer2D
BackBufferCopy
Bone2D
CPUParticles2D
Camera2D
CanvasModulate
Recent:
KinematicBody2D
AudioStreamPlay
TextureRect
Description:
KinematicBody2D: Kinematic body 2D node.
Cancel
Create

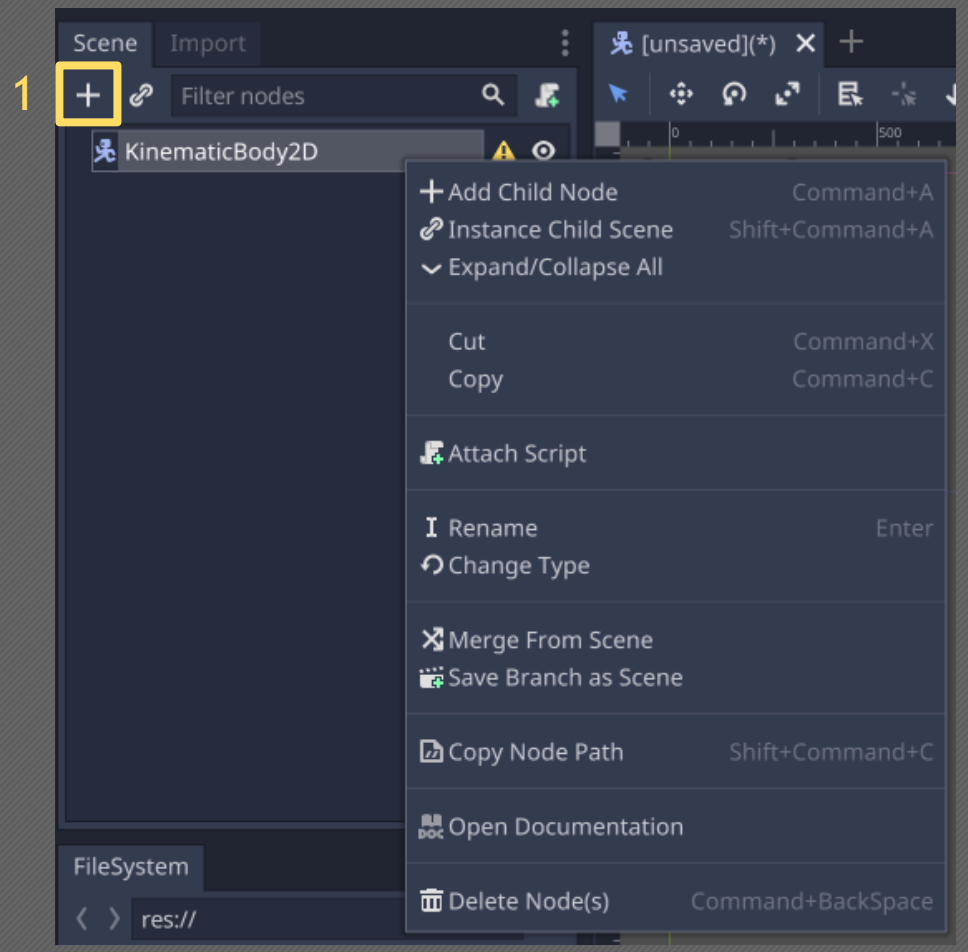

Create New Node
Favorites:
Search:
Matches:
Node
Spatial
CanvasItem
Control
Node2D
AnimatedSprite
CollisionObject2D
Area2D
PhysicsBody2D
KinematicBody2D
RigidBody2D
StaticBody2D
AudioStreamPlayer2D
BackBufferCopy
Bone2D
CPUParticles2D
Camera2D
CanvasModulate
Recent:
KinematicBody2D
AudioStreamPlay
TextureRect
Description:
KinematicBody2D: Kinematic body 2D node.
Cancel
Create


Create New Node
Favorites:
Search:
Matches:
Node
Spatial
CanvasItem
Control
Node2D
AnimatedSprite
CollisionObject2D
Area2D
PhysicsBody2D
KinematicBody2D
RigidBody2D
StaticBody2D
AudioStreamPlayer2D
BackBufferCopy
Bone2D
CPUParticles2D
Camera2D
CanvasModulate
Recent:
KinematicBody2D
AudioStreamPlay
TextureRect
Description:
KinematicBody2D: Kinematic body 2D node.
Cancel
Create


Create New Node
Favorites:
Search:
Matches:
Node
Spatial
CanvasItem
Control
Node2D
AnimatedSprite
CollisionObject2D
Area2D
PhysicsBody2D
KinematicBody2D
RigidBody2D
StaticBody2D
AudioStreamPlayer2D
BackBufferCopy
Bone2D
CPUParticles2D
Camera2D
CanvasModulate
Recent:
KinematicBody2D
AudioStreamPlay
TextureRect
Description:
KinematicBody2D: Kinematic body 2D node.
Cancel
Create


Create New Node
Favorites:
Search:
Matches:
Node
Spatial
CanvasItem
Control
Node2D
AnimatedSprite
CollisionObject2D
Area2D
PhysicsBody2D
KinematicBody2D
RigidBody2D
StaticBody2D
AudioStreamPlayer2D
BackBufferCopy
Bone2D
CPUParticles2D
Camera2D
CanvasModulate
Recent:
KinematicBody2D
AudioStreamPlay
TextureRect
Description:
KinematicBody2D: Kinematic body 2D node.
Cancel
Create

Adding Child Nodes to a Scene



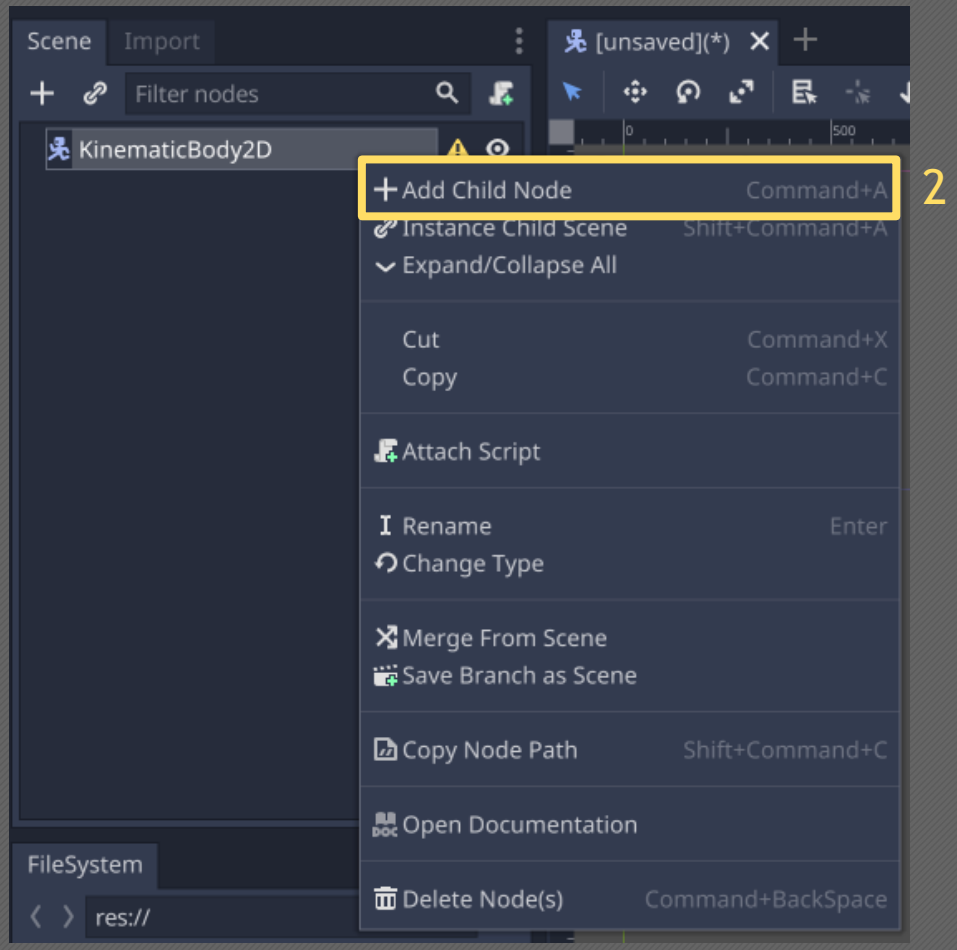
To add child node:

1. Click this "Add Child Node" button above the Scene Tree.

2. Right-click the root node and add select "Add Child Node".

Note: There is only one root node per scene. Any node (besides the root) can be nested as a child of any other node.

Adding Child Nodes to a Scene



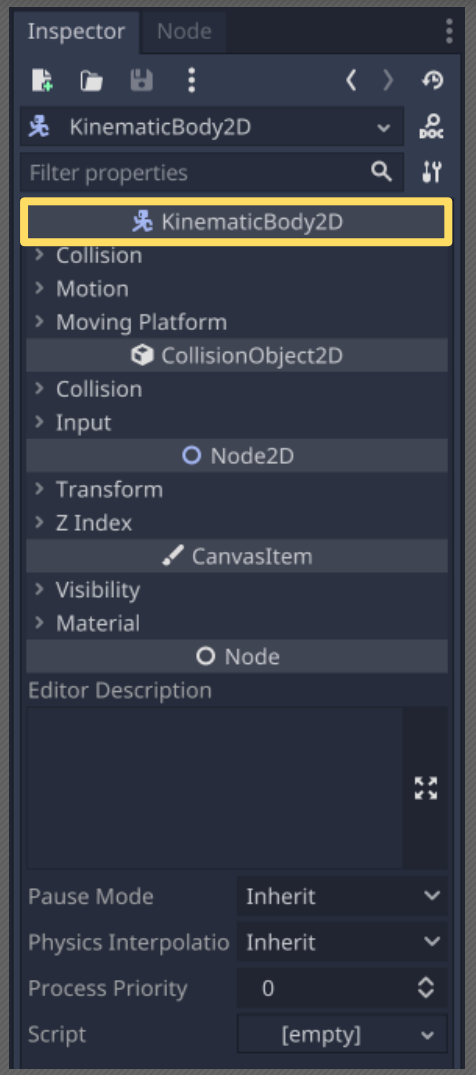
To add child node:

1. Click this “Add Child Node” button above the Scene Tree.

2. Right-click the root node and add select “Add Child Node”.

Note: There is only one root node per scene. Any node (besides the root) can be nested as a child of any other node.

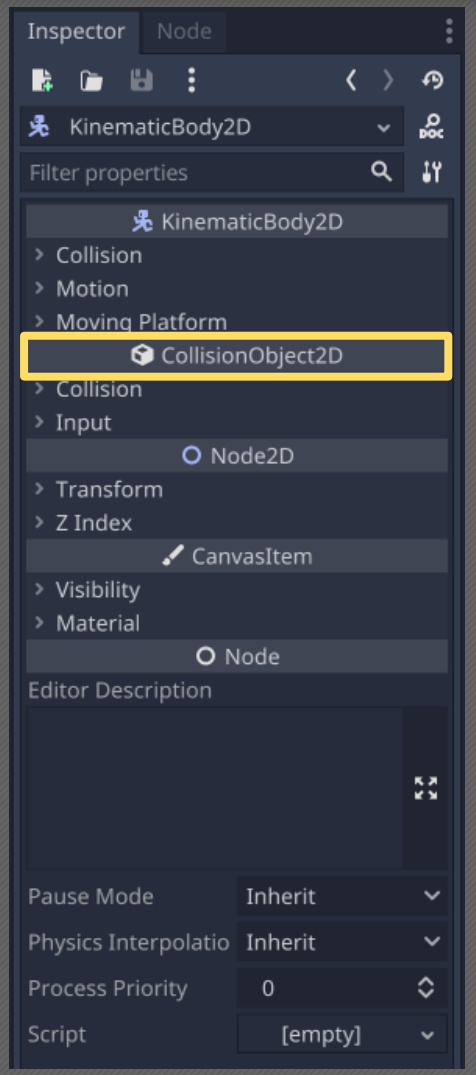
Understanding the Inspector Tab



The Inspector Tab updates whenever you select a node in the Scene Tree or in the central Scene Editor.

Each node has its own properties based on the type of the node and that node's class inheritance hierarchy.

Understanding the Inspector Tab



The Inspector Tab updates whenever you select a node in the Scene Tree or in the central Scene Editor.

Each node has its own properties based on the type of the node and that node's class inheritance hierarchy.

Creating the Player Scene

1. In the current Scene, create the root node as a KinematicBody2D.

2. Add a Sprite node as a child of the KinematicBody2D.

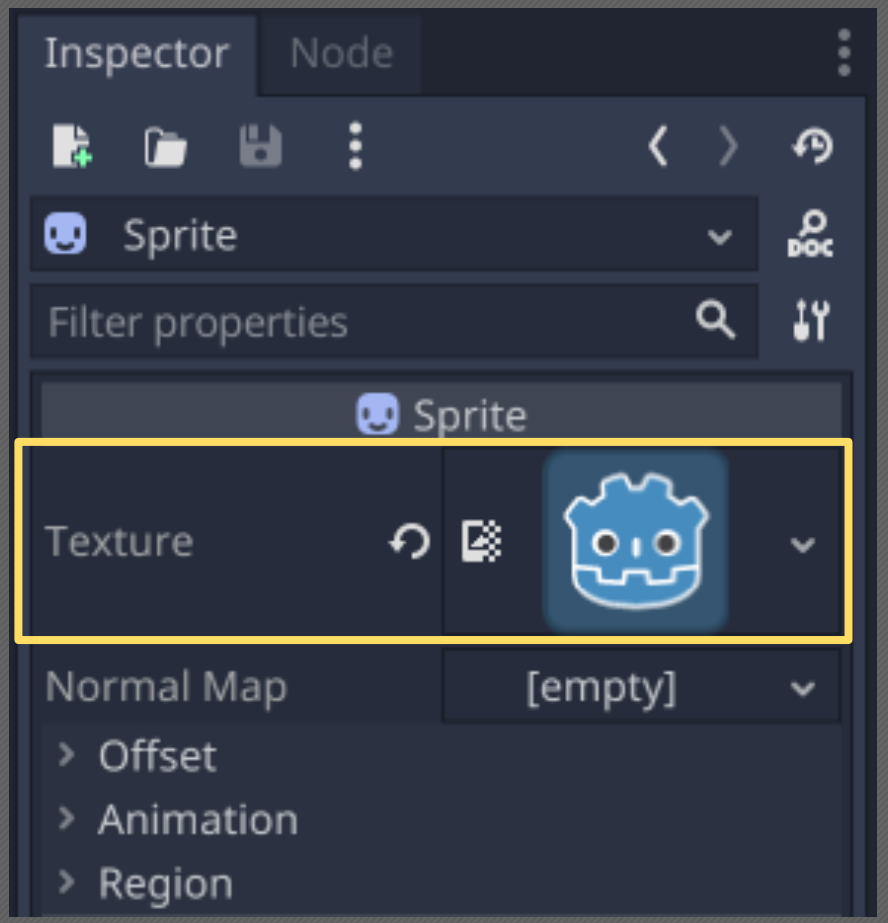
3. Add a CollisionShape2D node as a child of the KinematicBody2D.

Creating the Player Scene

With the Sprite node selected in your Scene tree:

1. Find the Texture property with an [empty] value in the Inspector.

2. Drag the icon.png file in your FileSystem tab into the Texture's [empty] value.

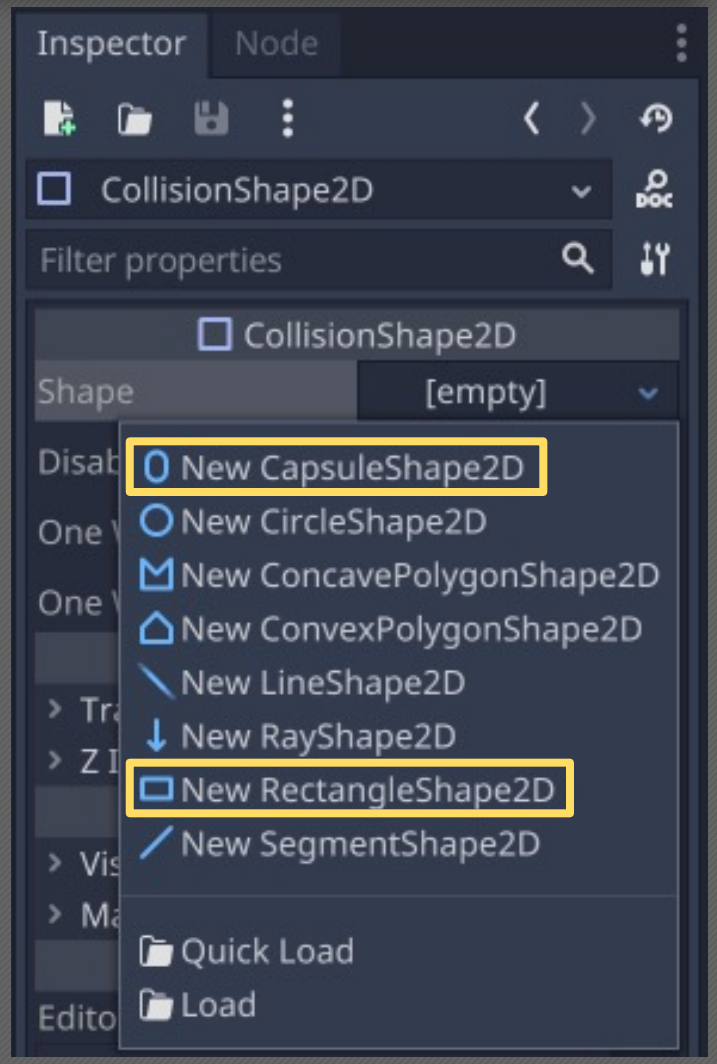


Setting Collision on a CollisionShape2D

Select the CollisionShape2D in your Scene tree:

- Click the Shape property dropdown and choose one of the following collision shape options:
  - **Capsule**: A good option if you want forgiving corners & some “slippery” edges.
  - **Rectangle**: Good for strict edges and larger wall/floor detection.

Stretch the size of your collision shape to be about the same shape as the Sprite.



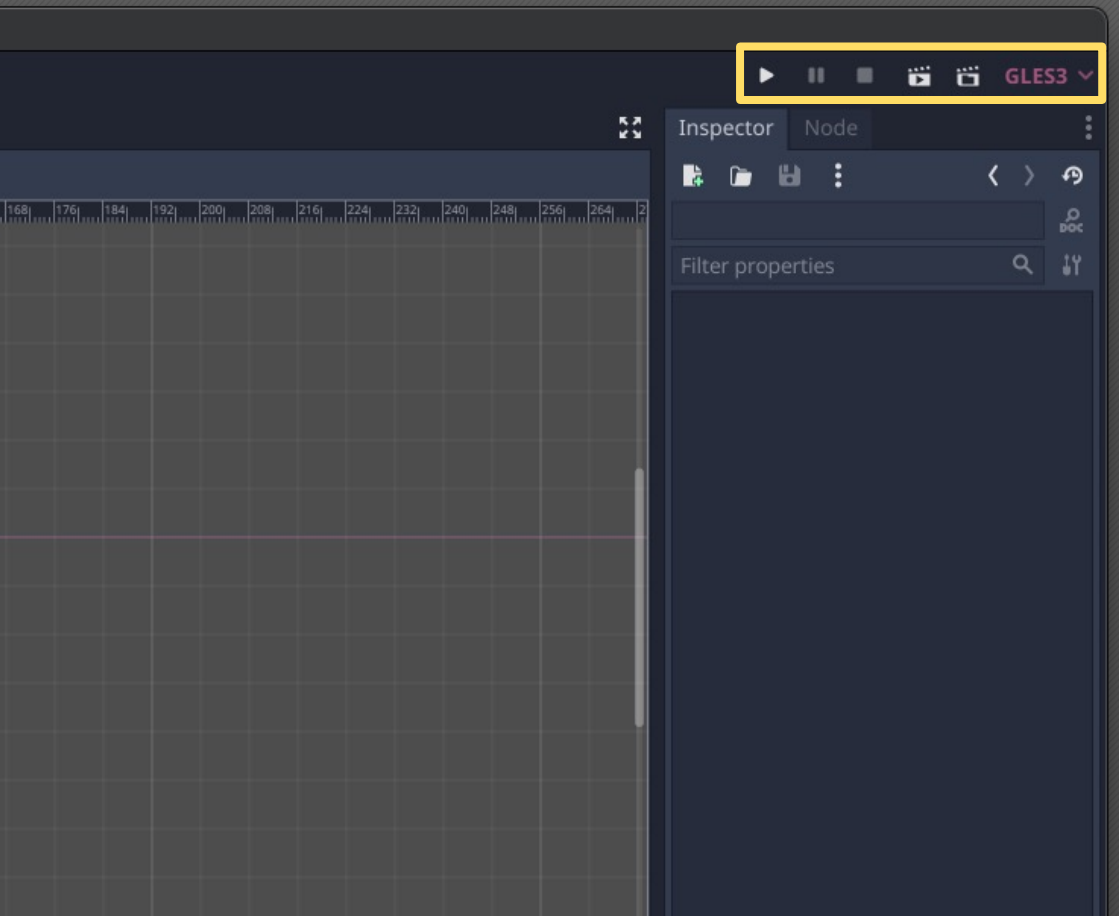
Save the Player Scene!

- Create a new Player subfolder in your Scenes folder.

- Save the current [unsaved] scene as Player.tscn in the new Player folder.

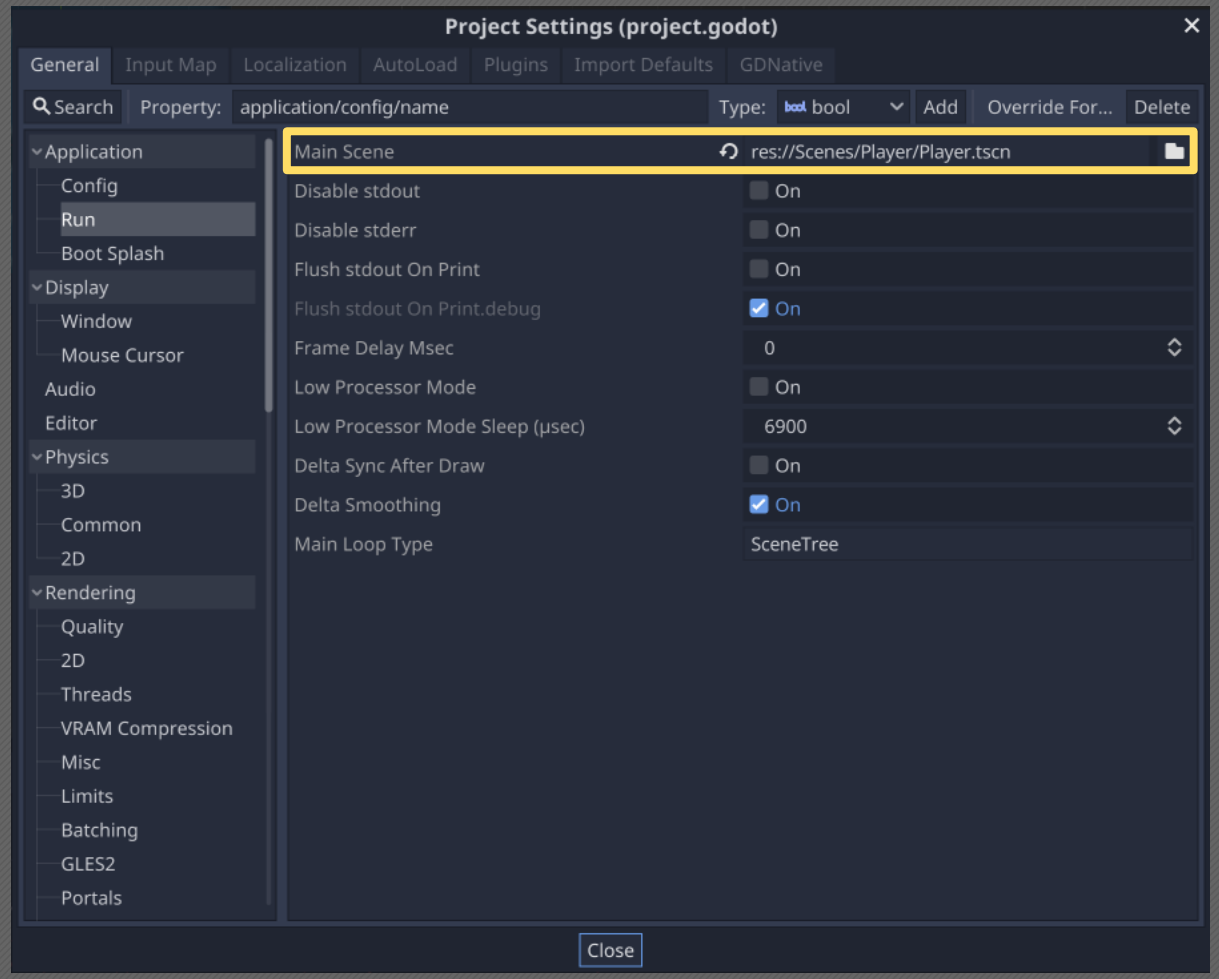
Note: The *.tscn* file format stands for "text scene". This structure is part of what makes Godot so version-control friendly!

Running the Scene

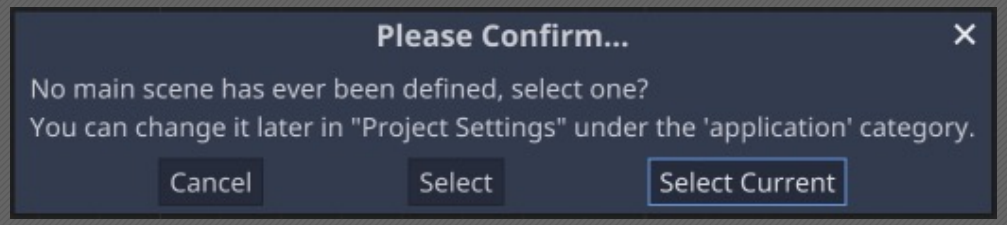


Run the current scene

Running the Scene



Run the project's configured "Main Scene"

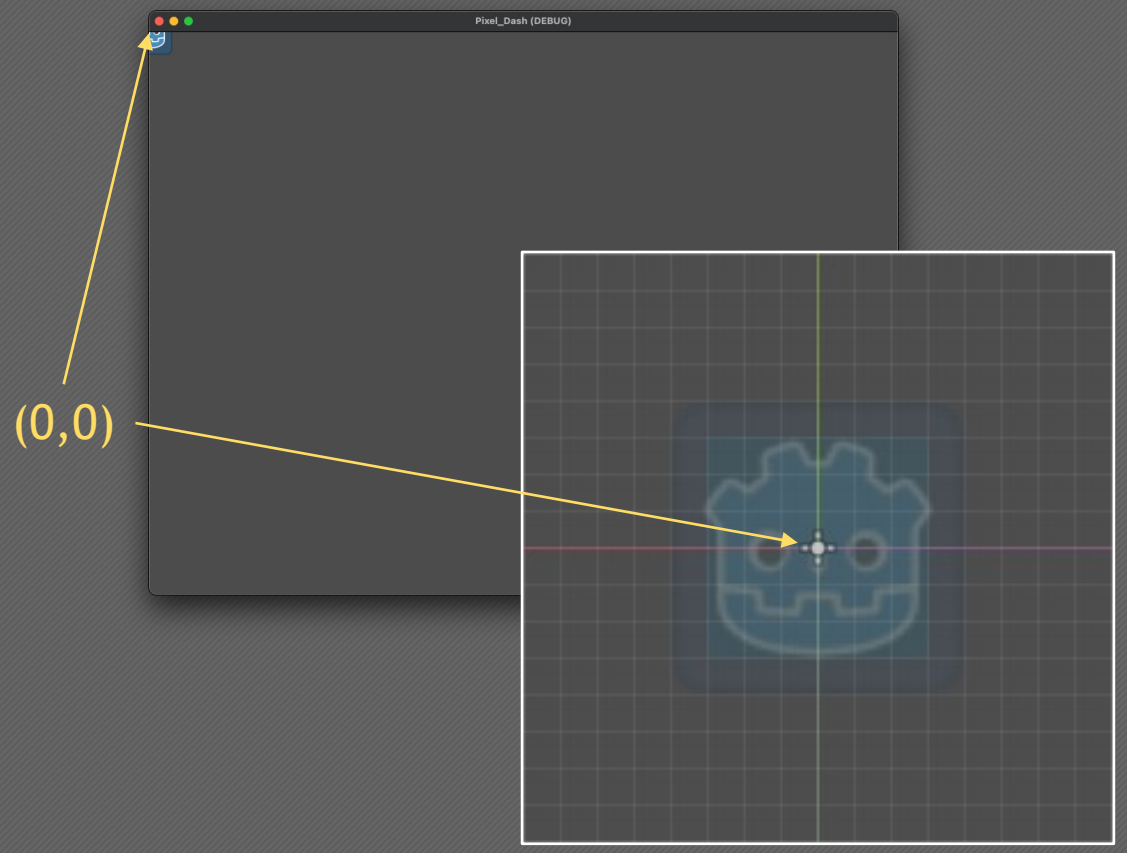


Scene Positions and “Cameras”

- When running a scene without a camera, that scene’s default viewport is positioned with the scene’s central position (0,0) in the top left of the window.

- Our Sprite & CollisionShape2D nodes are positioned at (0,0).

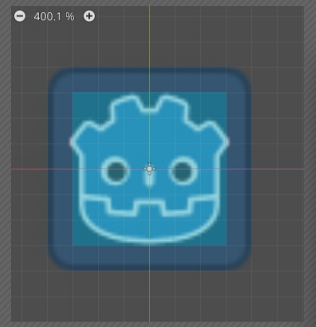
- Given these details, the result is that we end up only seeing the bottom-right corner of our Sprite.



Lab Time (~10 Minutes)

- Create a new Player scene.
- Assign the Sprite node the logo.png as its texture property.
- Give the CollisionShape2D a shape.
- Run the Player scene.

Nothing will happen when you run it, but you should see the Sprite in the corner.



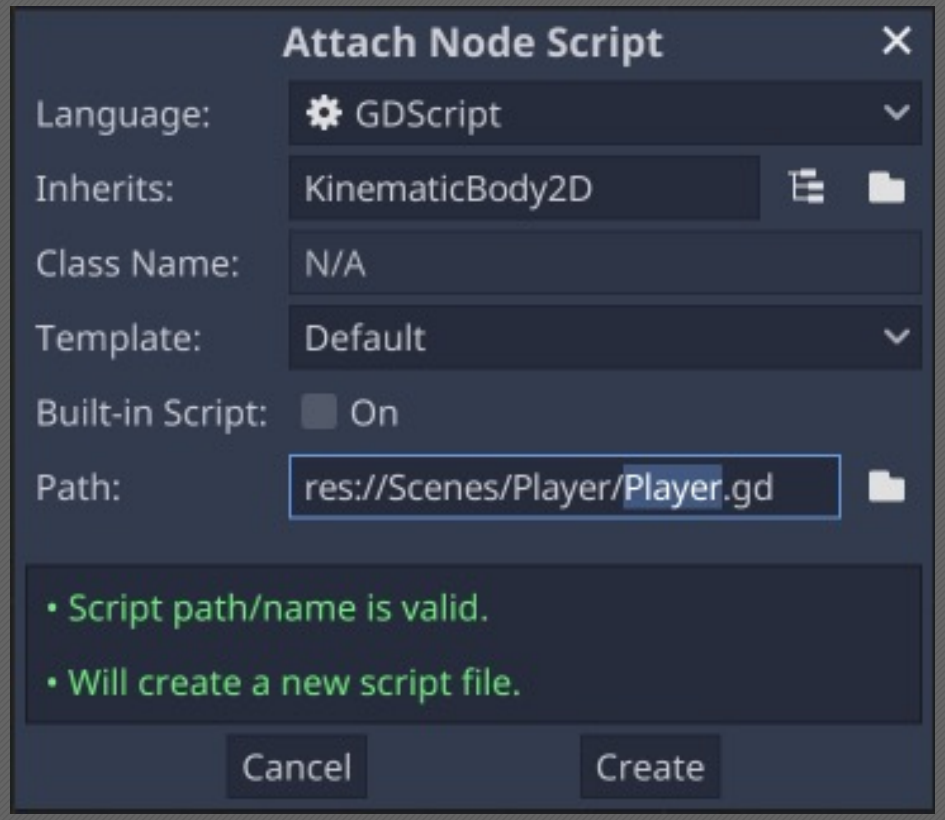
Well, this is boring… we need behavior!

- Most of the basic nodes that we can add to our scene do not inherently provide behavior based on game state or player interaction.
- We can attach a single Script to any node(s) in our scene tree to give it some behavior.
- Once a Script is attached to a node, we then add behavior by writing GDScript code in the attached Script.

Attaching a Script to the Player

- Select the root node (KinematicBody2D) and click the “Attach Script” button.

- Accept all the default prompt values and click Create.



2D 3D Script AssetLib

Player(*)

File Search Edit Go To Debug

Online Docs Search Help

Filter scripts

Player.gd

Player.gd

Filter methods

_ready

```
extends KinematicBody2D


# Declare member variables here. Examples:
# var a = 2
# var b = "text"


# Called when the node enters the scene tree for the first time.
func _ready():
	pass # Replace with function body.


# Called every frame. 'delta' is the elapsed time since the previous frame.
#func _process(delta):
#	pass

```

1 : 24

GDScript: A Short Primer

- An optimized, high-level, dynamic language.
- Highly resembles Python; the syntax is indentation-based and has many similar keywords.

```
var a = 5
var s = "Hello"
var arr = [1, 2, 3]
var dict = {"key": "value", 2: 3}
var typed_var: int
var inferred_type := "String"
```

```
func some_function(param1, param2):
    var local_var = 5

    if param1 < local_var:
        print(param1)
    elif param2 > 5:
        print(param2)
    else:
        print("Fail!")

    for i in range(20):
        print(i)

    while param2 != 0:
        param2 -= 1

    var local_var2 = param1 + 3
    return local_var2
```

```
# Constants

const ANSWER = 42
const THE_NAME = "Charly"

# Enums

enum {UNIT_NEUTRAL, UNIT_ENEMY, UNIT_ALLY}
enum Named {THING_1, THING_2, ANOTHER_THING = -1}

# Built-in vector types

var v2 = Vector2(1, 2)
var v3 = Vector3(1, 2, 3)
```

GDScript: Object Notification Methods

Function Name	Use Case(s)
_ready()	Called when the node is added to the active scene tree. This is executed downwards from the root.
_process(delta)	When you need a *framerate-dependent* delta between frames. Use when you need updates as often as possible. >> *Try to avoid doing physics calculations here*.
_physics_process(delta)	When you need a *framerate-independent* delta between frames. Use when you need consistent updates over time. >> *Try to avoid doing input processing here*.
_input(event)	Called whenever an input event occurs. The input event propagates up through the node tree until a node consumes it.

The delta is the elapsed time since the previous frame was executed/processed.

Player Behavior: Movement & Handling Input

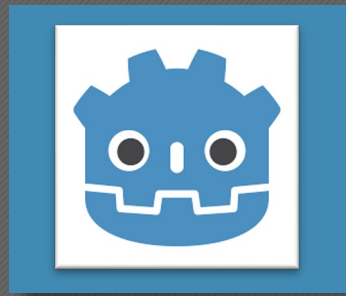
```
extends KinematicBody2D

var velocity = Vector2.ZERO
onready var move_speed : float = 175 * scale.x

func _physics_process(delta):
    var move_left = Input.get_action_strength("ui_left")
    var move_right = Input.get_action_strength("ui_right")
    velocity.x = (move_right - move_left) * move_speed

    var move_up = Input.get_action_strength("ui_up")
    var move_down = Input.get_action_strength("ui_down")
    velocity.y = (move_down - move_up) * move_speed

    velocity = move_and_slide(velocity, Vector2.UP)
```

Player Behavior: Movement & Handling Input

```
extends KinematicBody2D

var velocity = Vector2.ZERO
onready var move_speed : float = 175 * scale.x

func _physics_process(delta):
    var move_left = Input.get_action_strength("ui_left")
    var move_right = Input.get_action_strength("ui_right")
    velocity.x = (move_right - move_left) * move_speed

    var move_up = Input.get_action_strength("ui_up")
    var move_down = Input.get_action_strength("ui_down")
    velocity.y = (move_down - move_up) * move_speed

    velocity = move_and_slide(velocity, Vector2.UP)
```

The velocity variable will be constantly updated and used to determine where the Player is moving to.

Player Behavior: Movement & Handling Input



```
extends KinematicBody2D

var velocity = Vector2.ZERO
onready var move_speed : float = 175 * scale.x

func _physics_process(delta):
    var move_left = Input.get_action_strength("ui_left")
    var move_right = Input.get_action_strength("ui_right")
    velocity.x = (move_right - move_left) * move_speed

    var move_up = Input.get_action_strength("ui_up")
    var move_down = Input.get_action_strength("ui_down")
    velocity.y = (move_down - move_up) * move_speed

    velocity = move_and_slide(velocity, Vector2.UP)
```

onready - initializes a variable once the Node the script is attached to (and its children) are part of the scene tree

Player Behavior: Movement & Handling Input

```
extends KinematicBody2D

var velocity = Vector2.ZERO
onready var move_speed : float = 175 * scale.x

func _physics_process(delta):
    var move_left = Input.get_action_strength("ui_left")
    var move_right = Input.get_action_strength("ui_right")
    velocity.x = (move_right - move_left) * move_speed

    var move_up = Input.get_action_strength("ui_up")
    var move_down = Input.get_action_strength("ui_down")
    velocity.y = (move_down - move_up) * move_speed

    velocity = move_and_slide(velocity, Vector2.UP)
```

The get_action_strength function return a value between 0 (no signal) and 1 (full strength) of the named Input.

Player Behavior: Movement & Handling Input

```
extends KinematicBody2D

var velocity = Vector2.ZERO
onready var move_speed : float = 175 * scale.x

func _physics_process(delta):
    var move_left = Input.get_action_strength("ui_left")
    var move_right = Input.get_action_strength("ui_right")
    velocity.x = (move_right - move_left) * move_speed

    var move_up = Input.get_action_strength("ui_up")
    var move_down = Input.get_action_strength("ui_down")
    velocity.y = (move_down - move_up) * move_speed

    velocity = move_and_slide(velocity, Vector2.UP)
```

The move_and_slide function recalculates/translates the Player's velocity (speed) in the specified direction.

Lazy Debugging

The print() function will print into the Console whatever object arg you give it.

You can perform String interpolation via %s (for String) or %d (for a number) in the first arg, followed by a **% [array]** operation to interpolate the values by index as they're found.

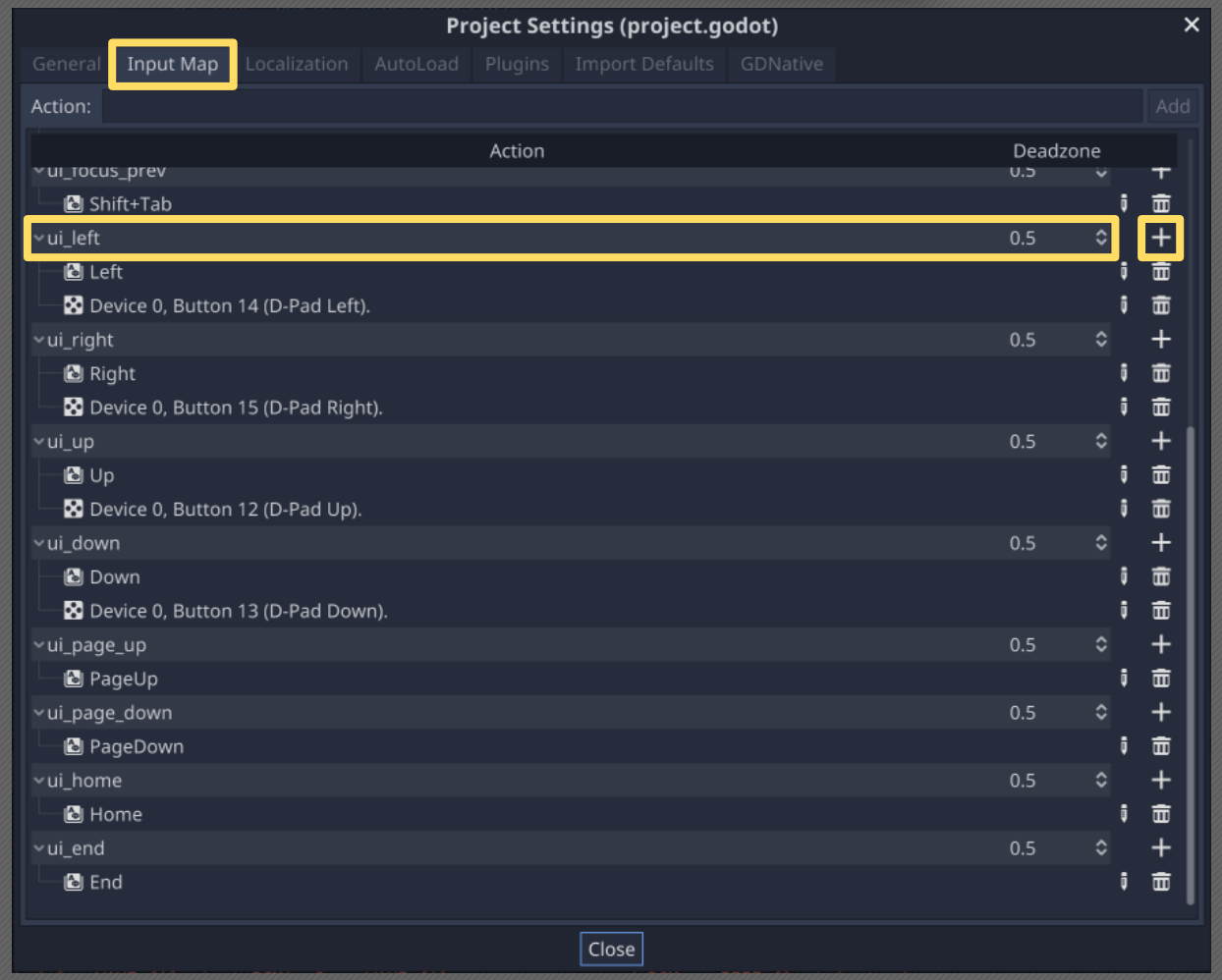
```
print("Some Value %s and Some Value %s" % ["one", "two"])
```

Binding Input Actions

In Project Settings:

- Select the Input Map tab.
- Find ”ui_left” and click the Plus sign.
- Select the ”Key” option, press the “A” key to bind it, and click Ok.
- Do this with the W, S, and D keys for ui_up, ui_down, and ui_right respectively.

Note: The “Key” option represents the logical input key, where “Physical Key” is the physical key location on the keyboard.



Lab Time (~10 Minutes)

- Create a new Player script attached to the Player scene.
- Update Input bindings in Project Settings.
- Implement basic movement.

```
extends KinematicBody2D

var velocity = Vector2.ZERO
onready var move_speed : float = 175 * scale.x

func _physics_process(delta):
    var move_left = Input.get_action_strength("ui_left")
    var move_right = Input.get_action_strength("ui_right")
    velocity.x = (move_right - move_left) * move_speed

    var move_up = Input.get_action_strength("ui_up")
    var move_down = Input.get_action_strength("ui_down")
    velocity.y = (move_down - move_up) * move_speed

    velocity = move_and_slide(velocity, Vector2.UP)
```

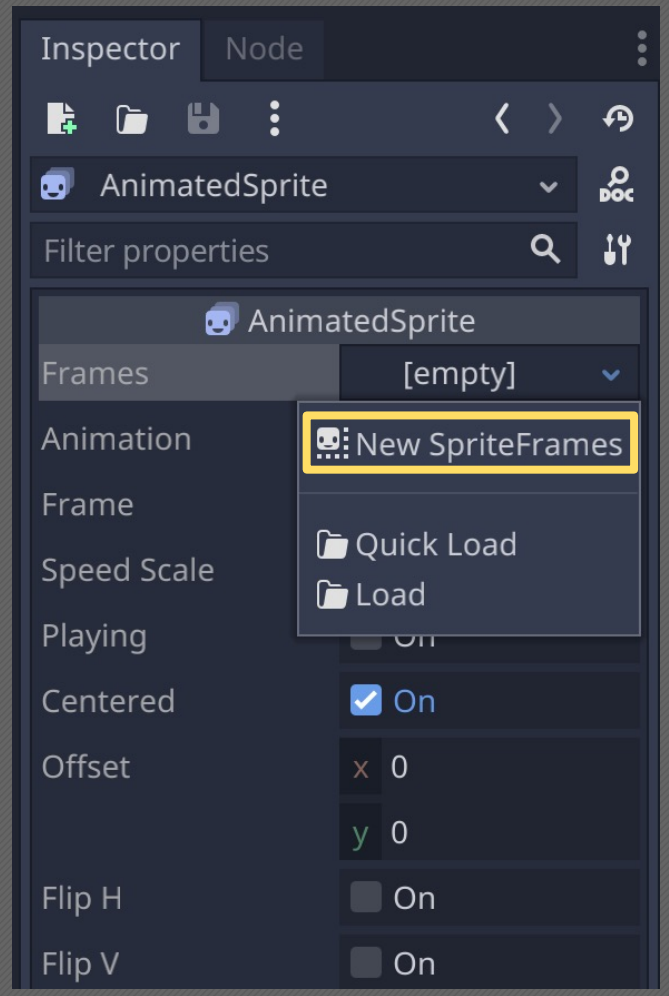
Progress Check

Where we are.

Where we want to be.

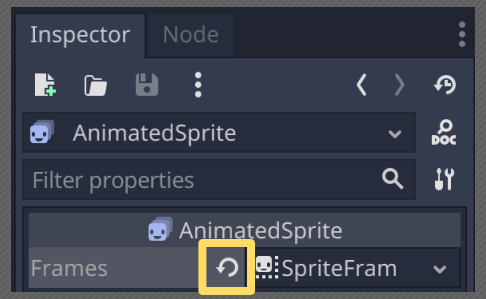
Adding Player Animations

- Rename the root KinematicBody2D as “Player” (Right-Click it and find Rename).
- Add a new AnimatedSprite node to the Scene (the root node).
- Create a new SpriteFrames for that new AnimatedSprite node’s Frames property.

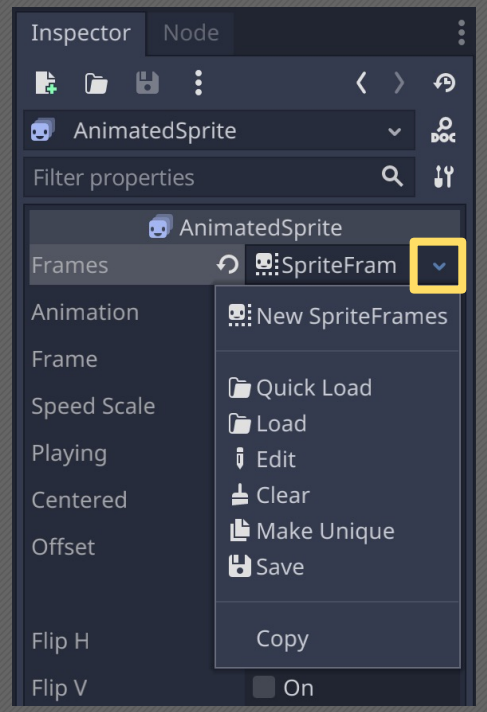


Be careful where you click!

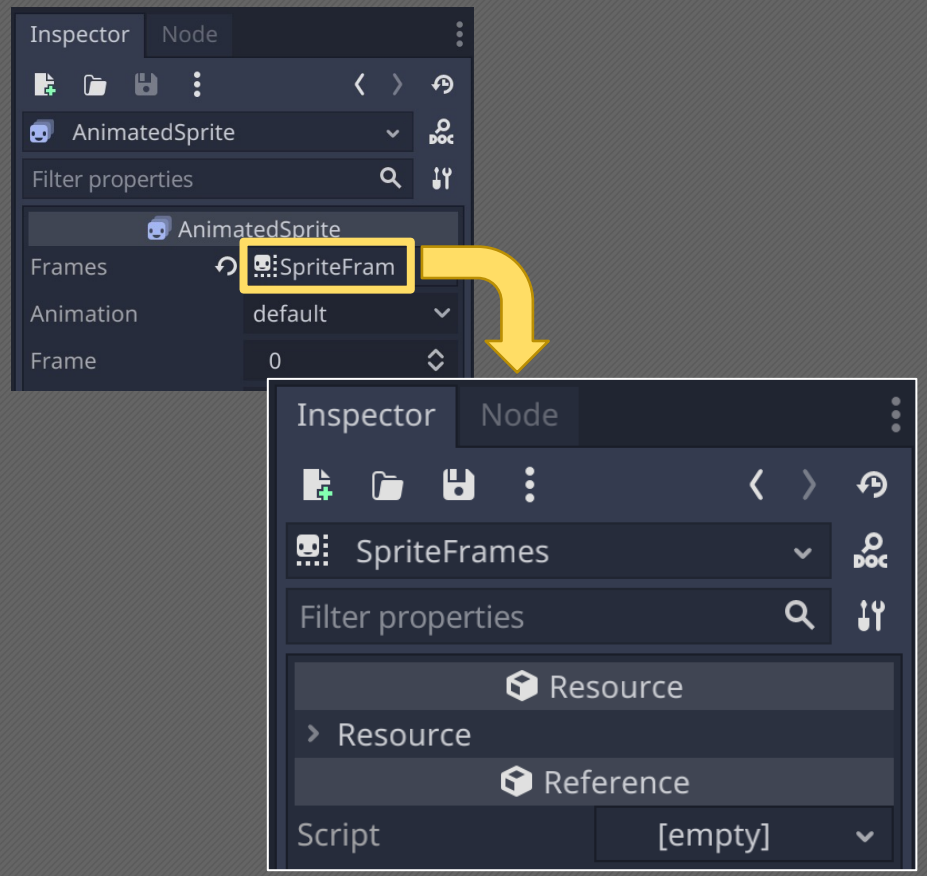
Reset to Default



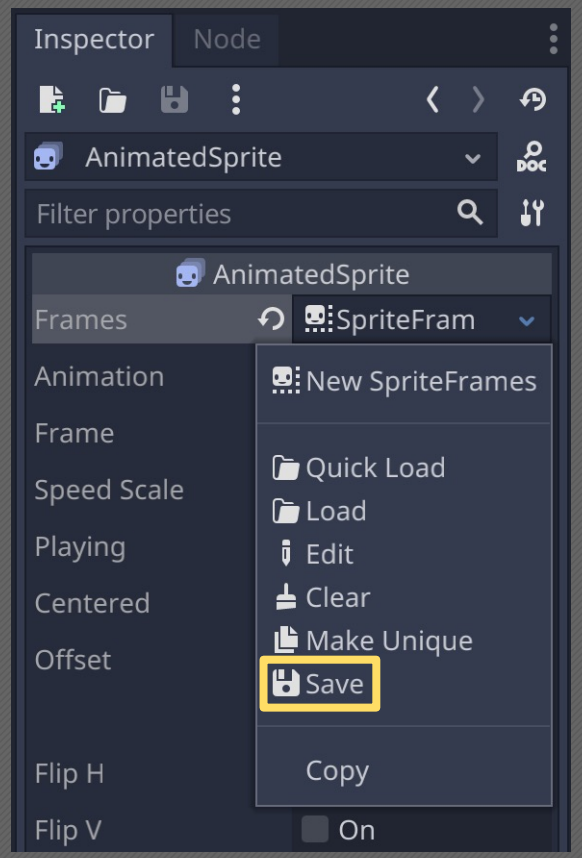
Resource Options

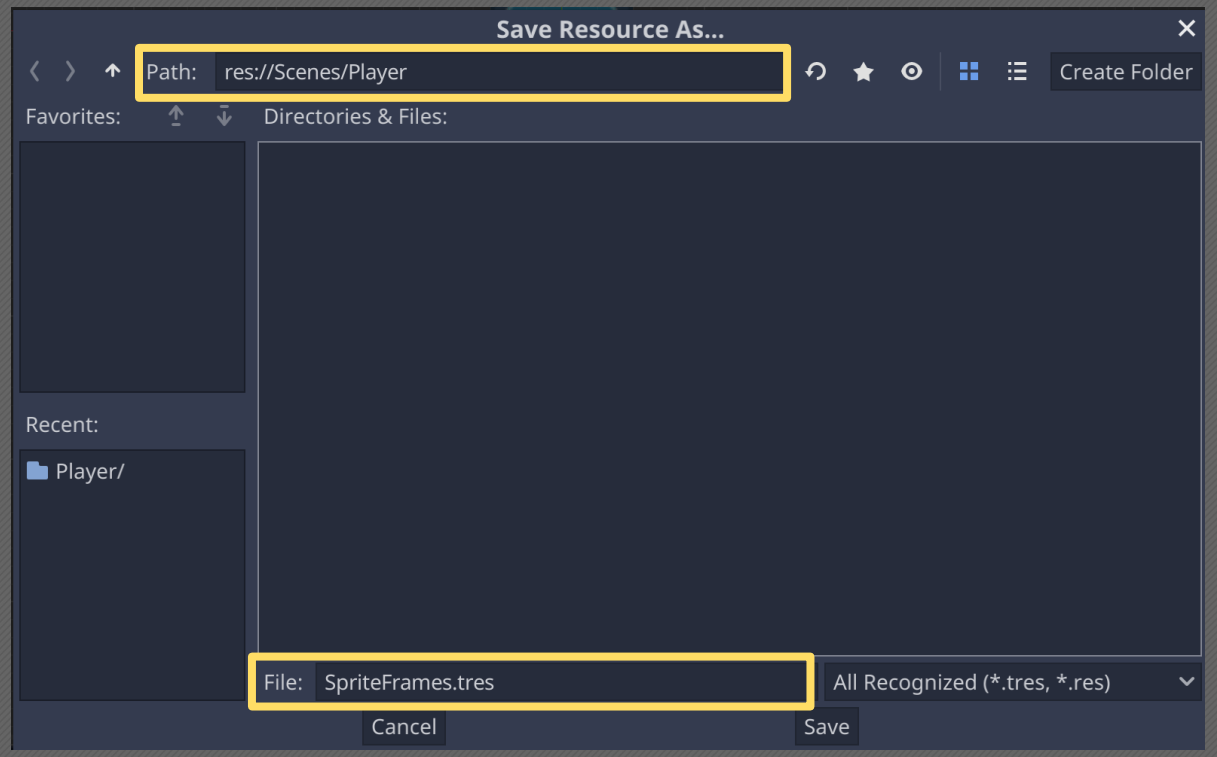


Change Inspector Tab Context



Just to be safe, save the Player SpriteFrames



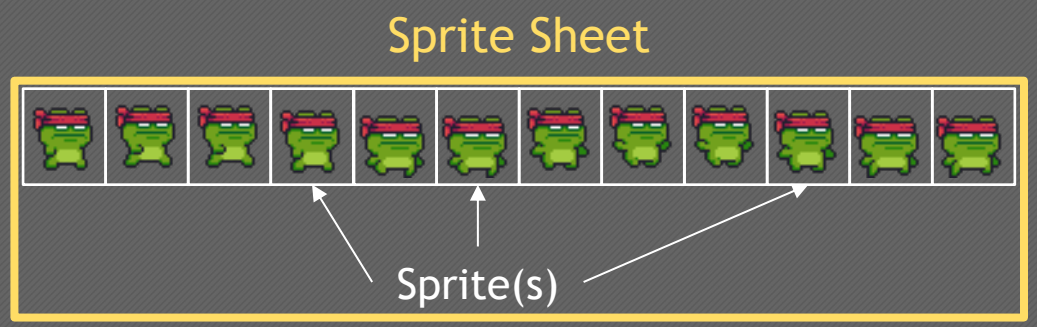


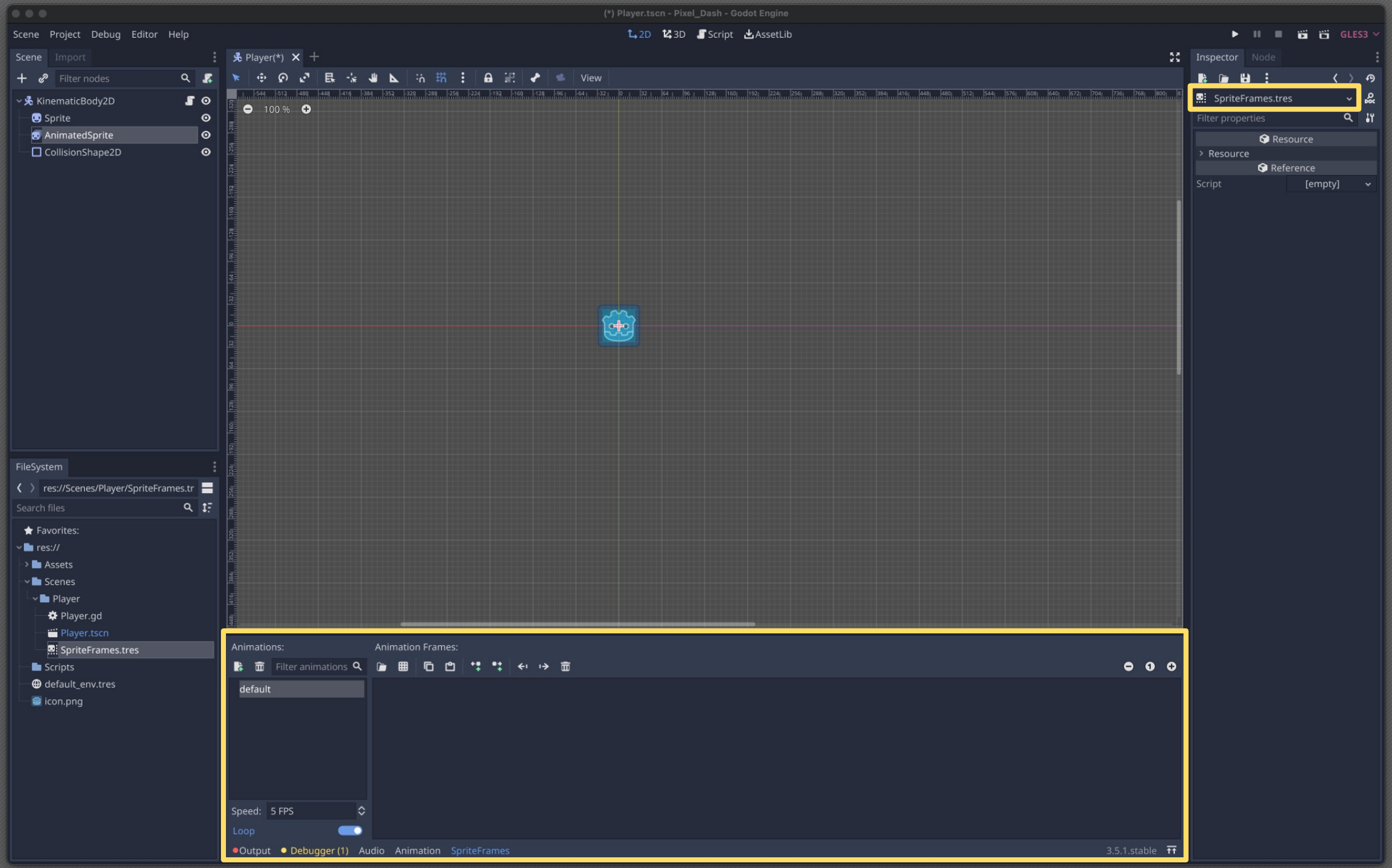
If at some point you accidentally reset or wipe your Frames property, you can drag-and-drop your saved SpriteFrames resource into the Frames property value.

Sprites & Sprite Sheets

- A Sprite Sheet is an image containing a collection (or multiple collections) of logically-grouped, equal-sized images; each individual image slice is called a Sprite.

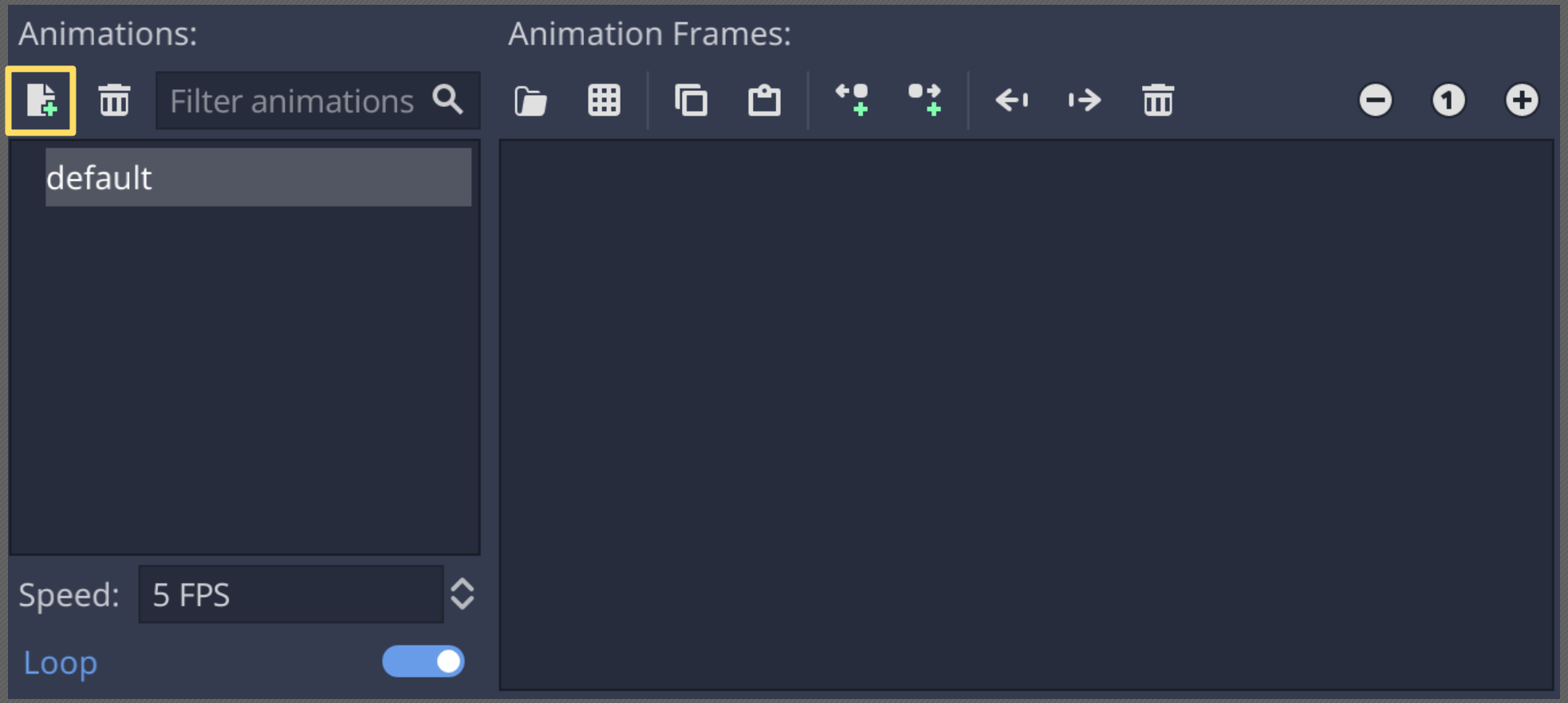
- Sprite Sheets are useful for creating 2D tilesets or animations.




(*) Player.tscn - Pixel_Dash - Godot Engine
Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
GLES3
Scene
Import
Filter nodes
KinematicBody2D
Sprite
AnimatedSprite
CollisionShape2D
Player(*)
View
100 %
Inspector
Node
SpriteFrames.tres
Filter properties
Resource
Resource
Reference
Script
[empty]
FileSystem
res://Scenes/Player/SpriteFrames.tr
Search files
Favorites:
res://
Assets
Scenes
Player
Player.gd
Player.tscn
SpriteFrames.tres
Scripts
default_env.tres
icon.png
Animations:
Filter animations
default
Speed:
5 FPS
Loop
Animation Frames:
Output
Debugger (1)
Audio
Animation
SpriteFrames
3.5.1.stable

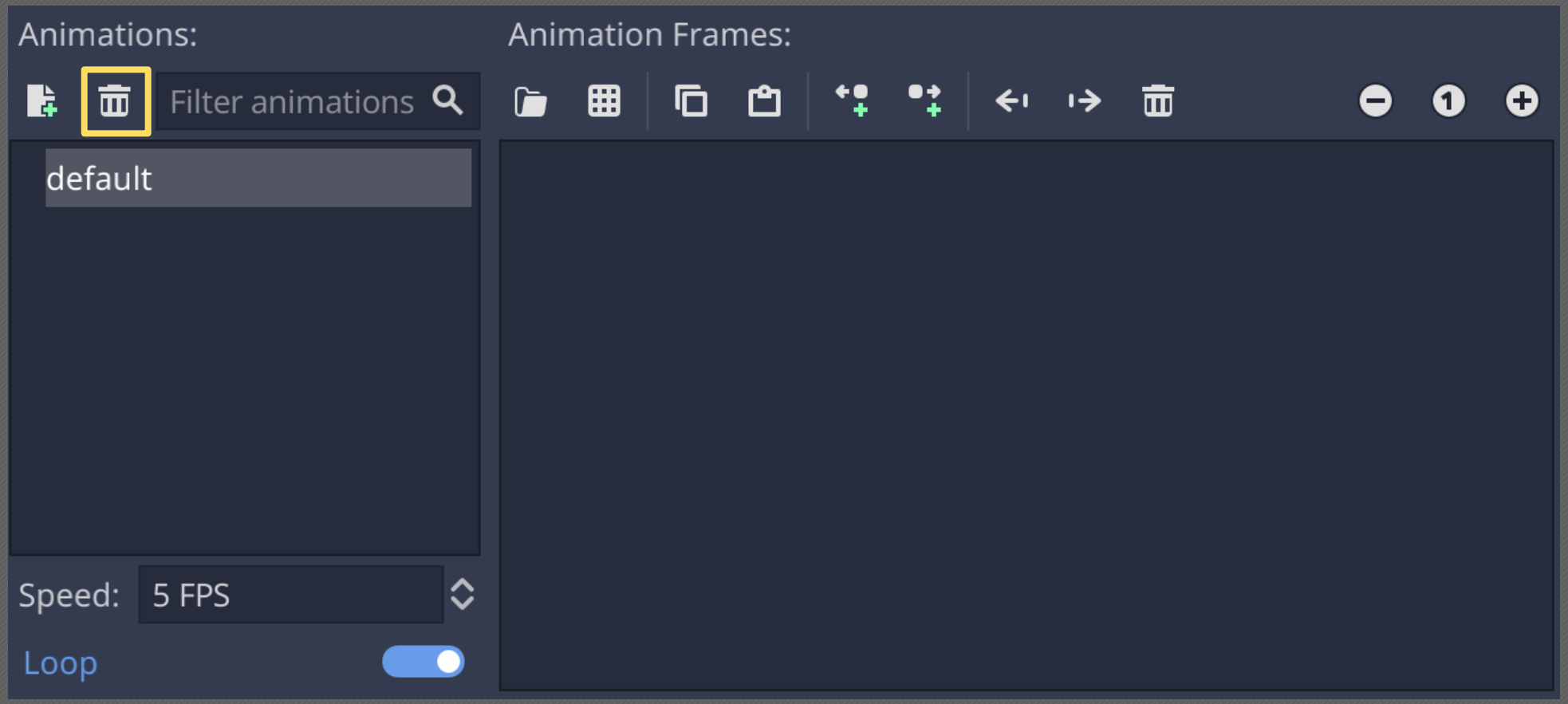
Navigating SpriteFrames

New Animation



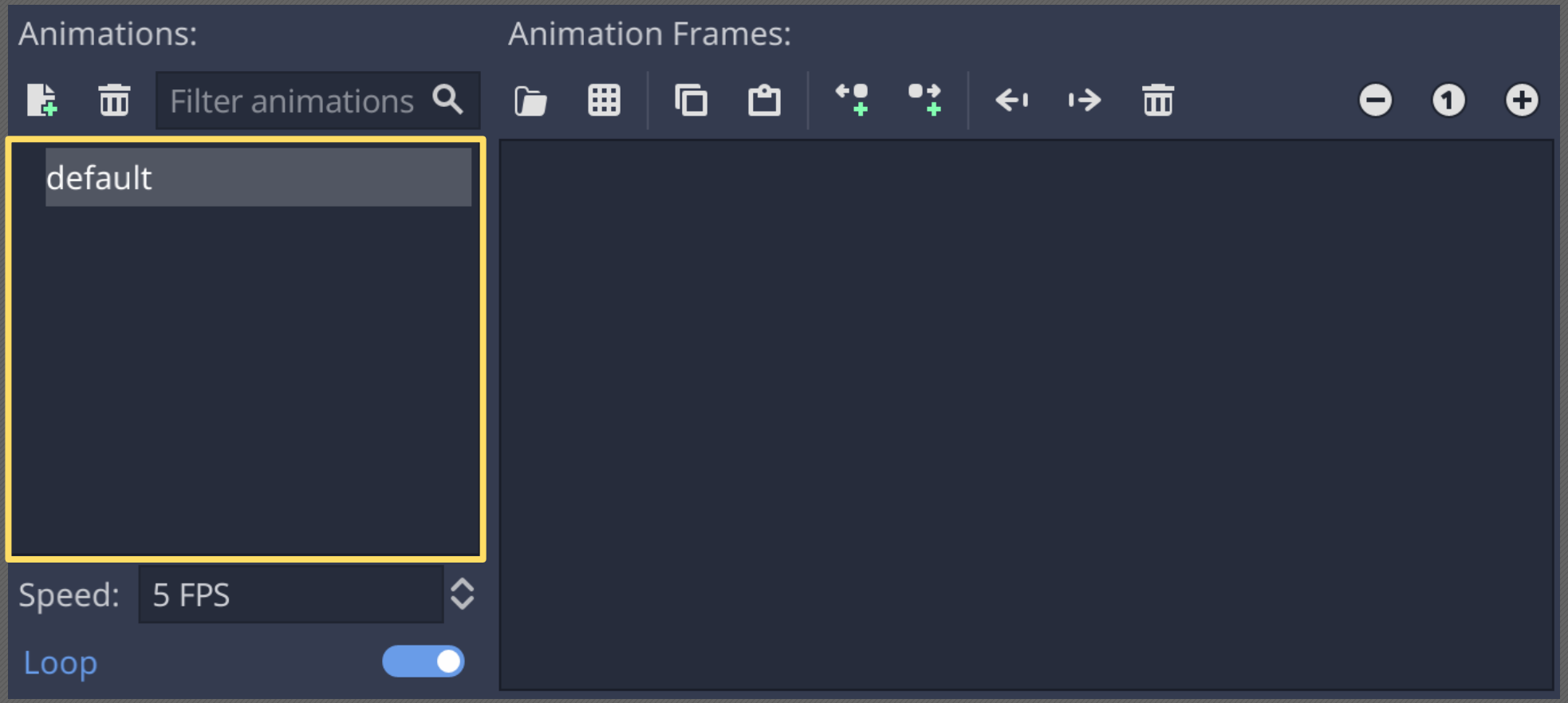
Navigating SpriteFrames

Delete Selected Animation



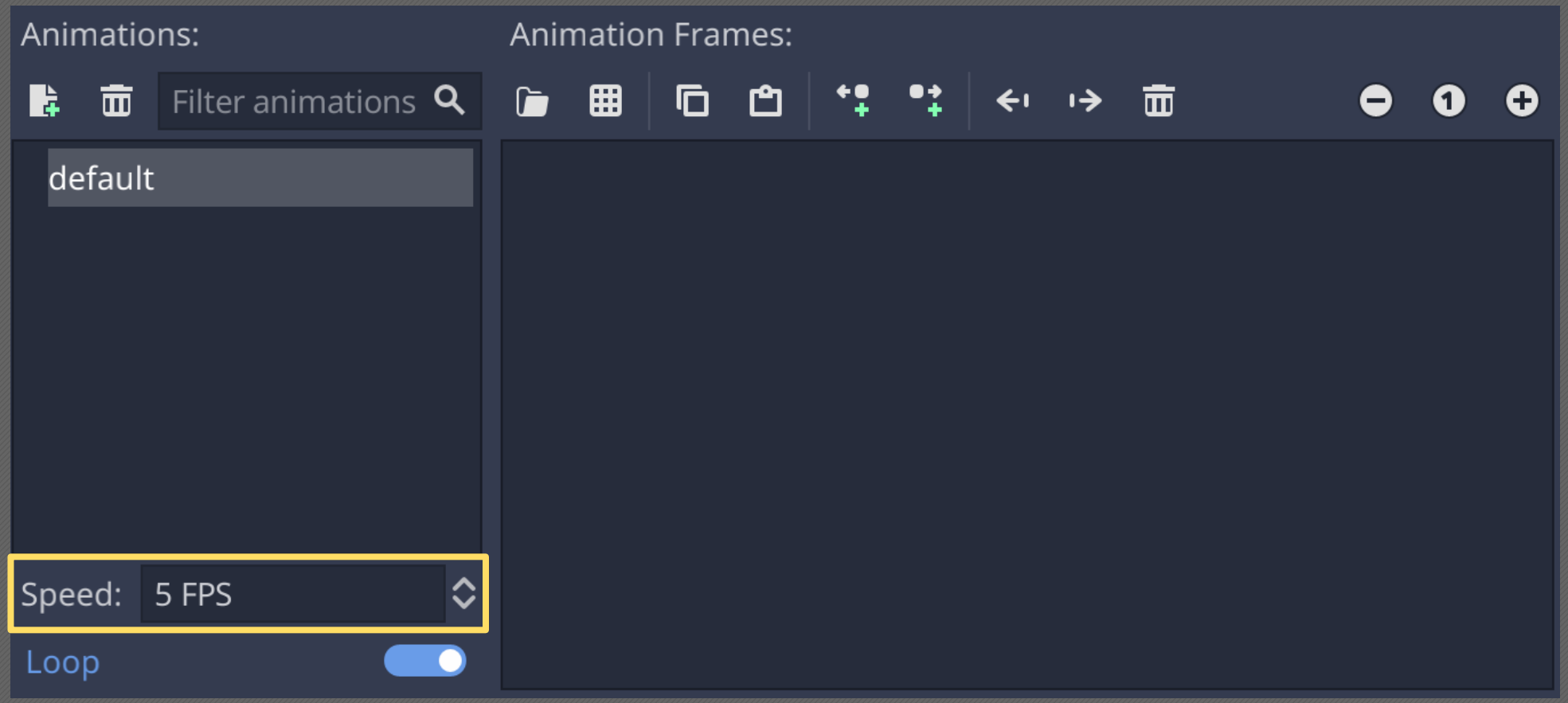
Navigating SpriteFrames

Animations list



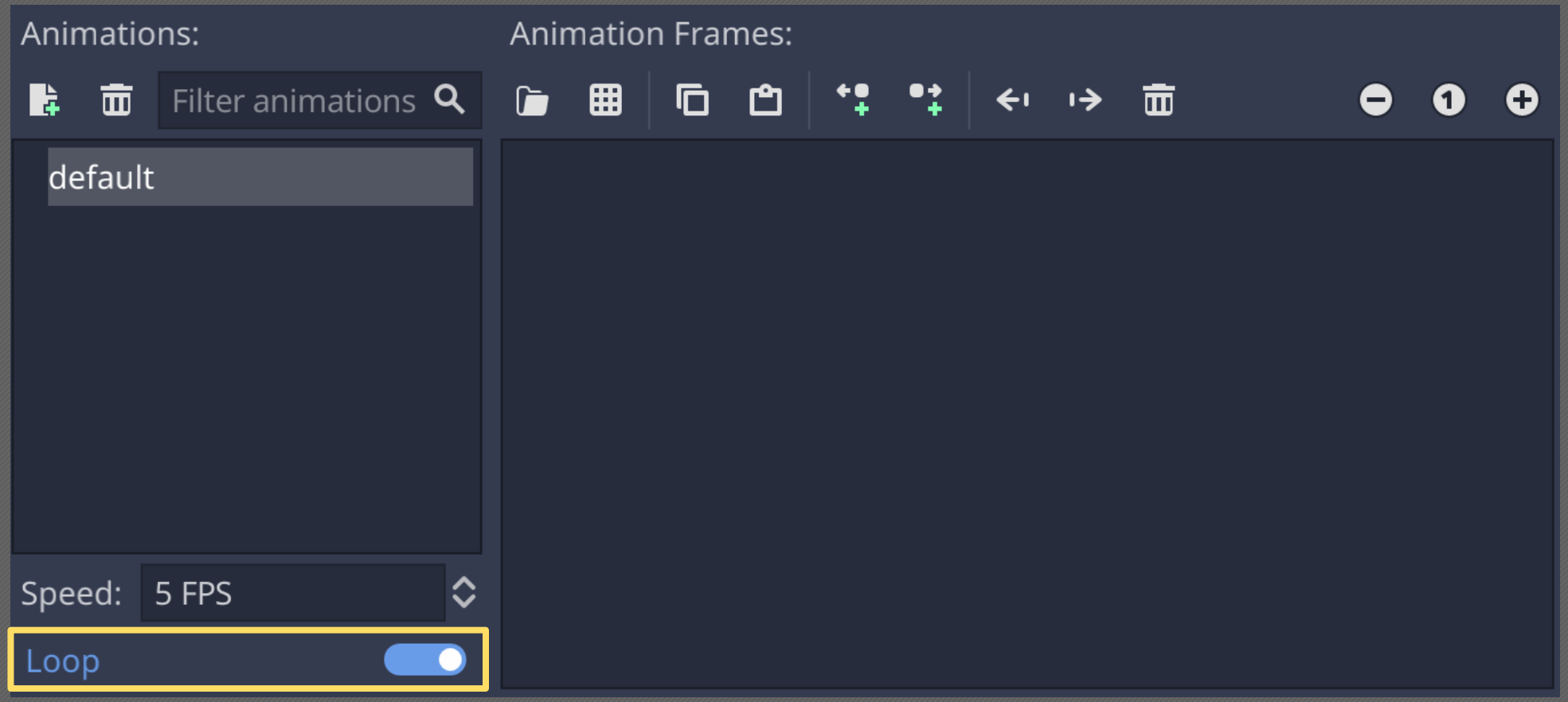
Navigating SpriteFrames

The run speed (frame rate) of the selected animation



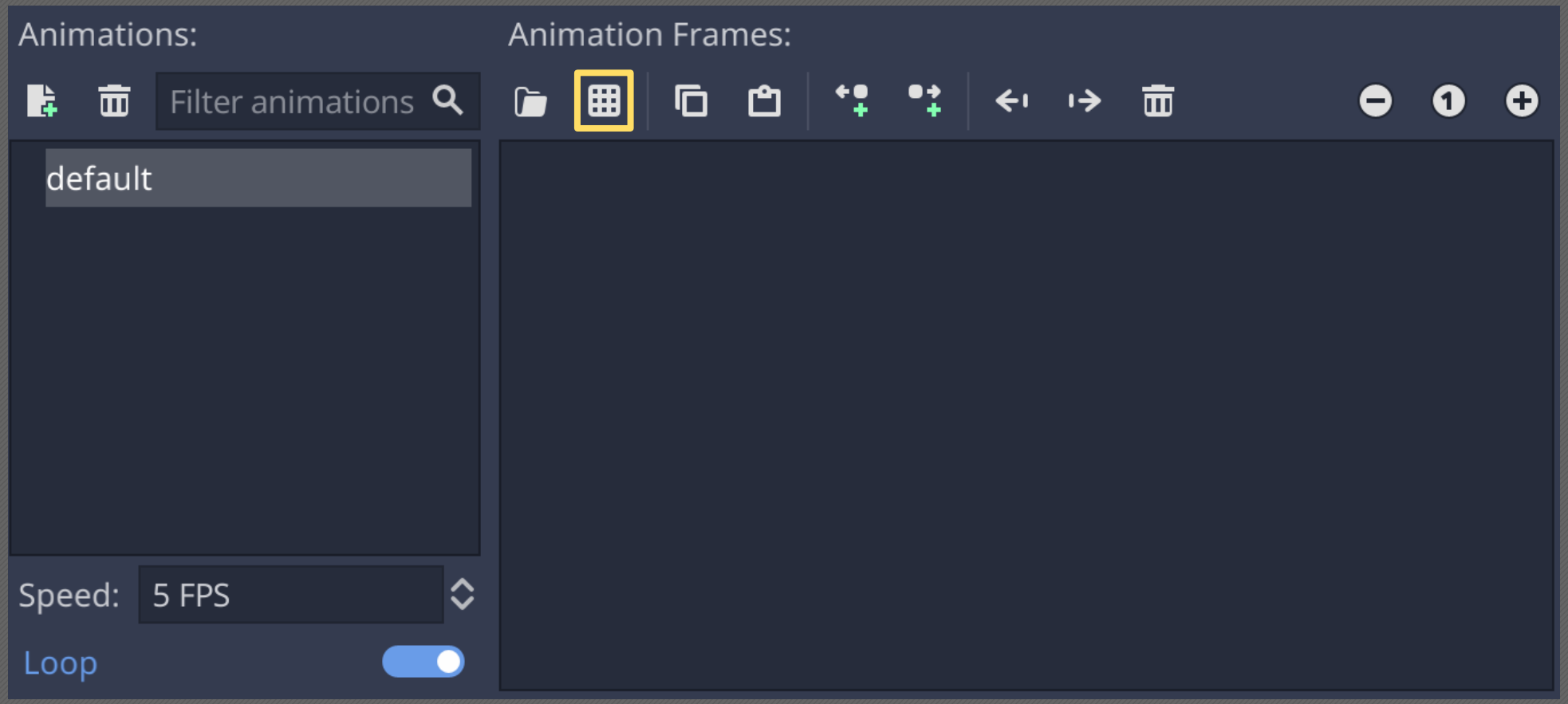
Navigating SpriteFrames

Loop toggle (if toggled On, the animation will continuously replay once it's finished)



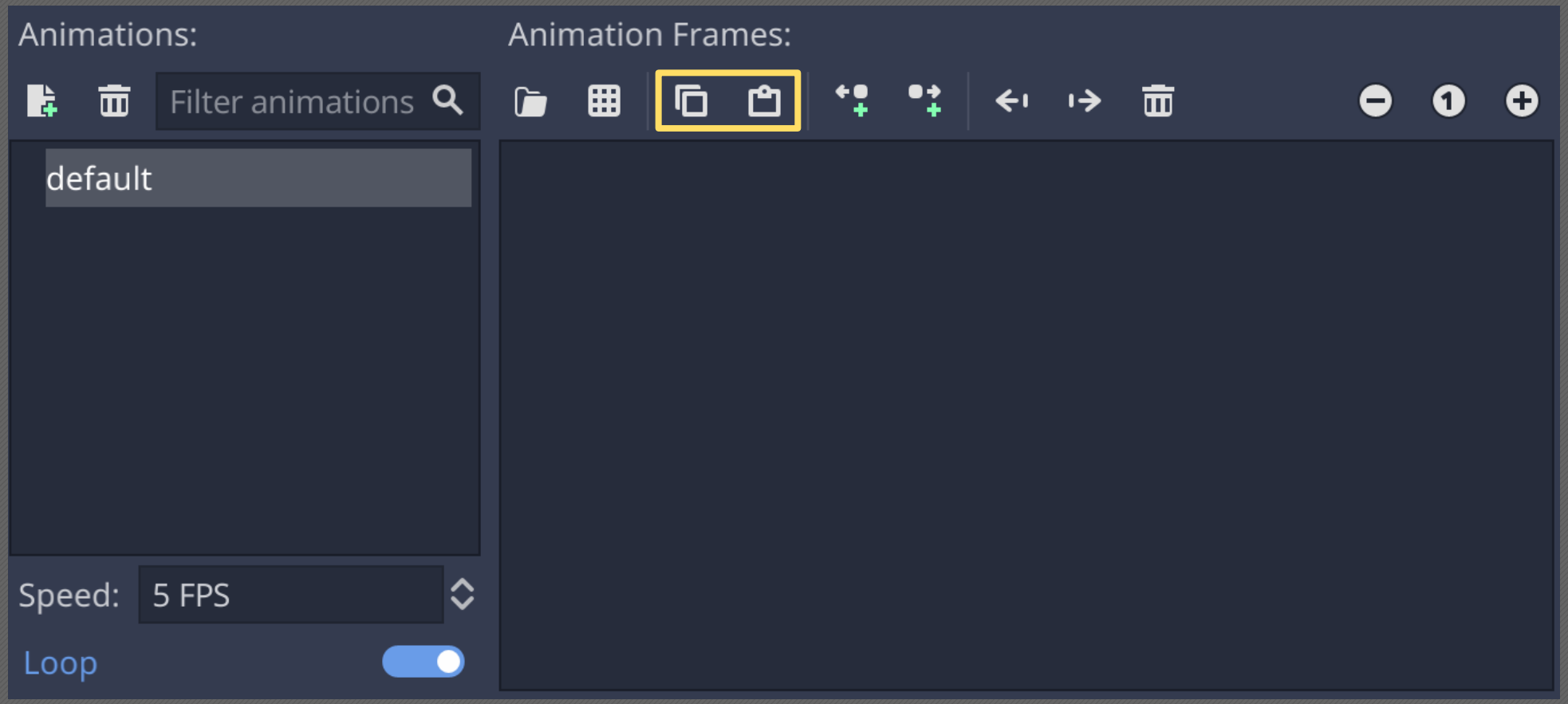
Navigating SpriteFrames

Add Frames from a Sprite Sheet file



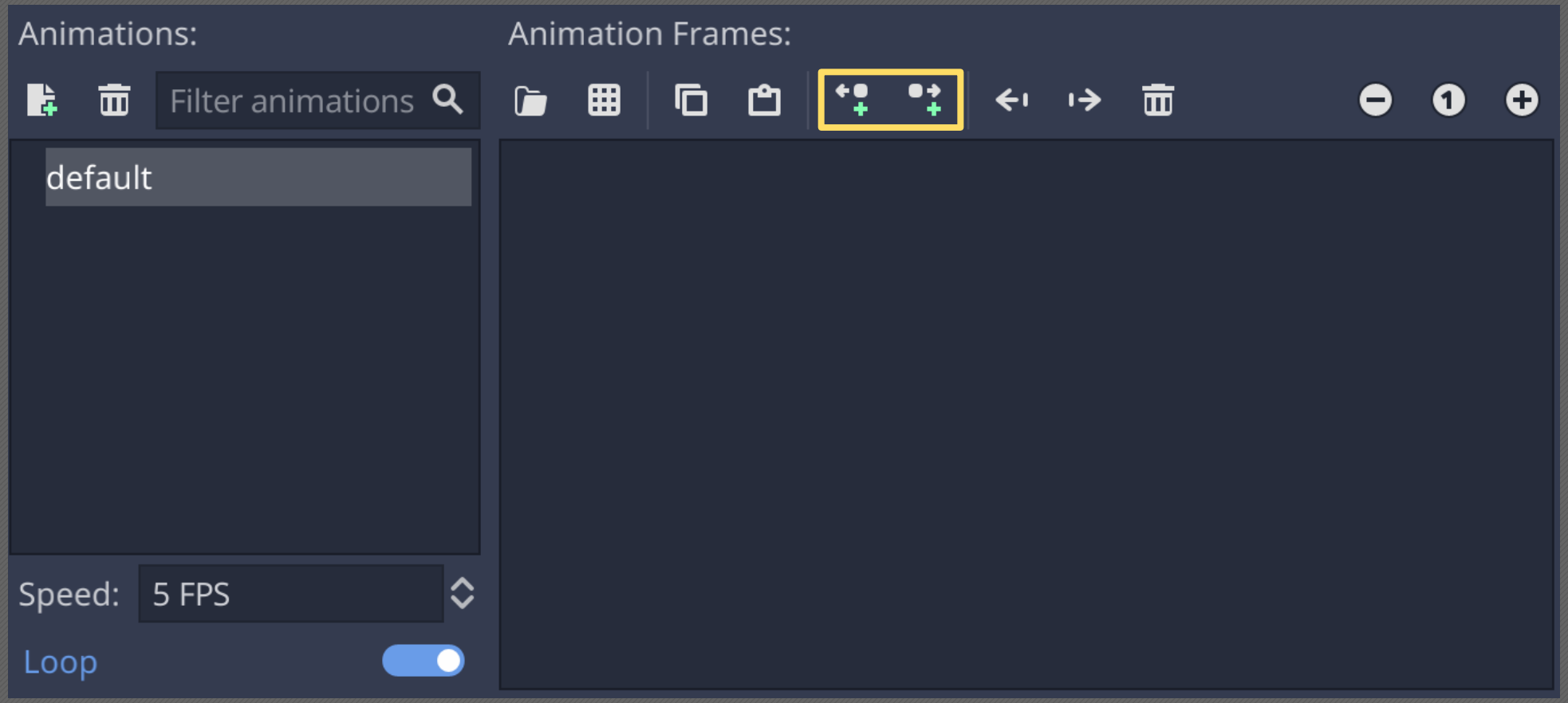
Navigating SpriteFrames

Copy/Paste for animation frames



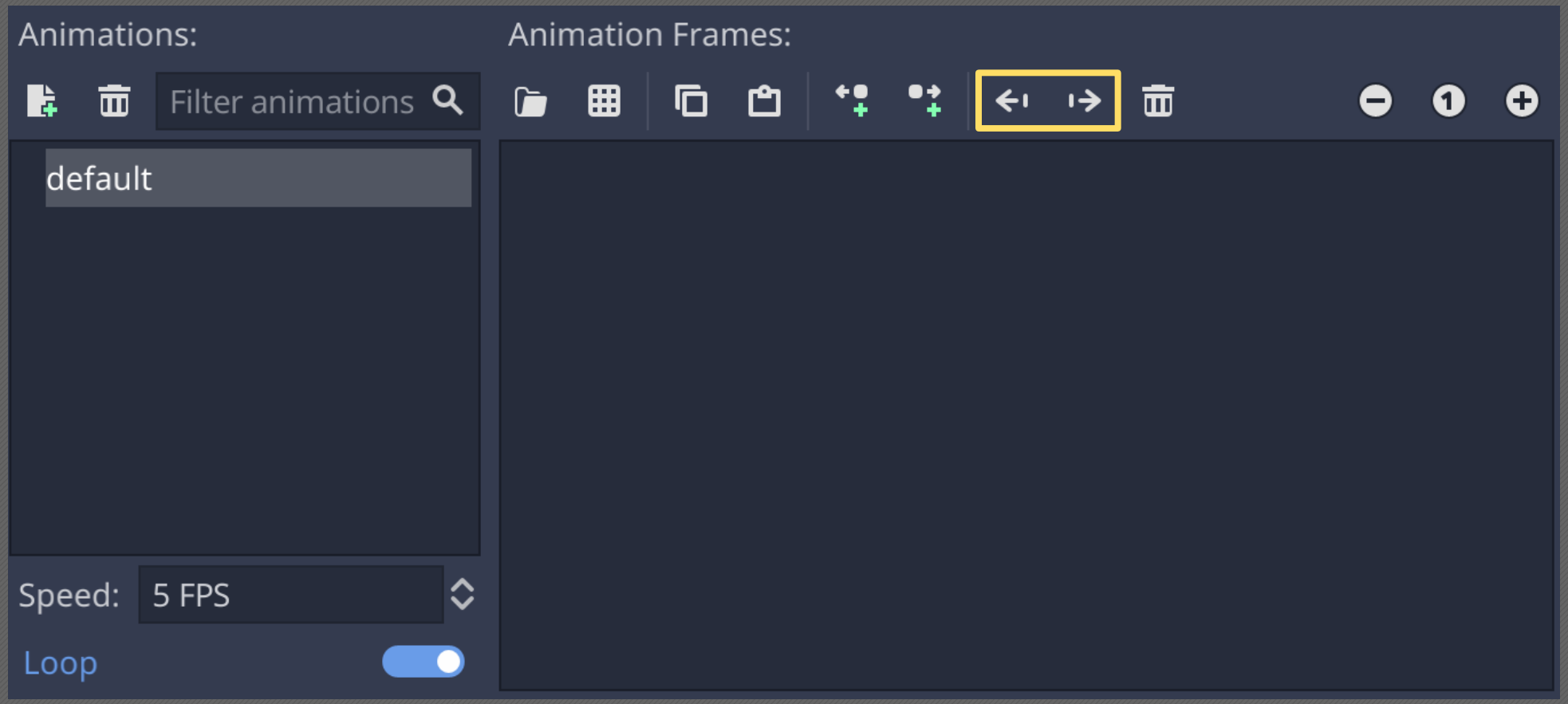
Navigating SpriteFrames

Insert empty (blank) frame before/after the currently-selected frame



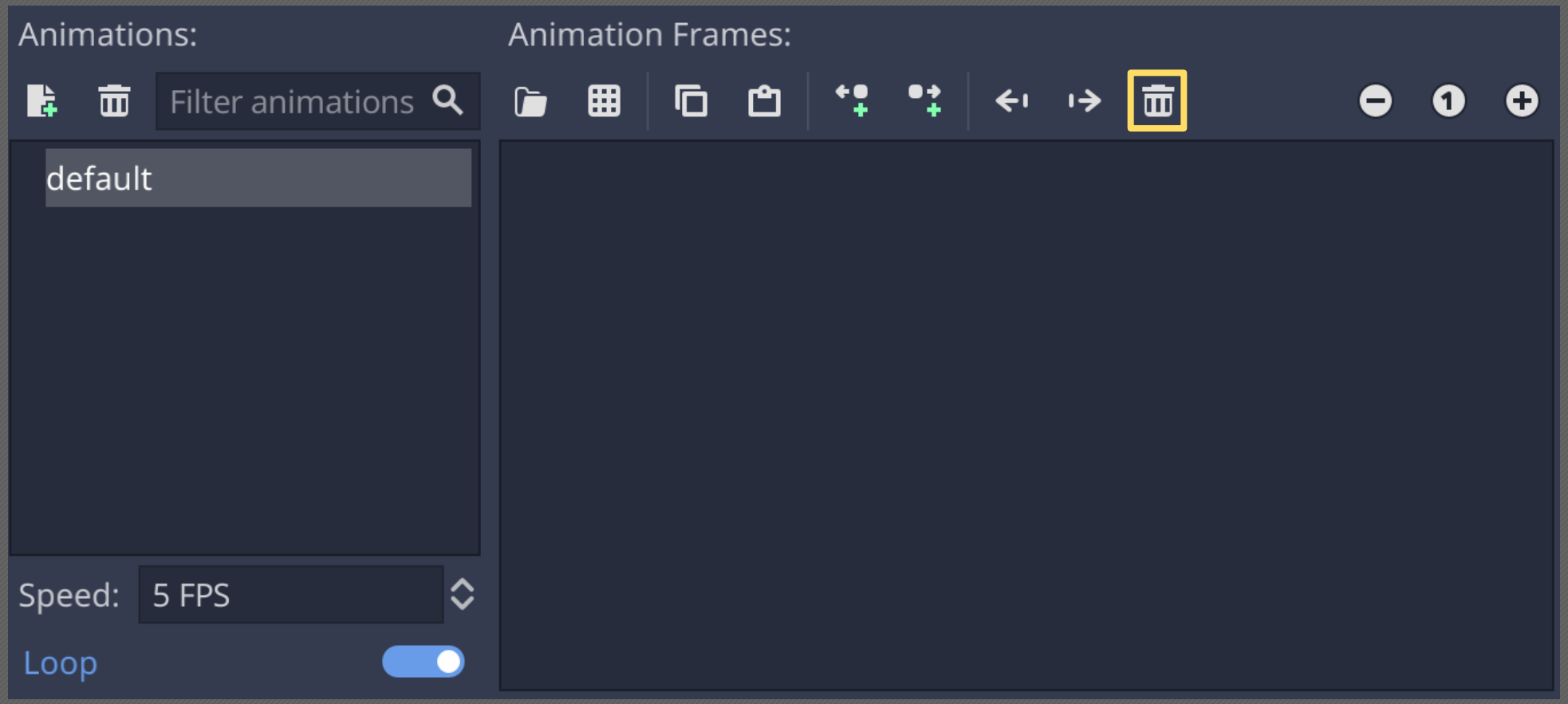
Navigating SpriteFrames

Move the currently-selected frame 1 frame to the left/right



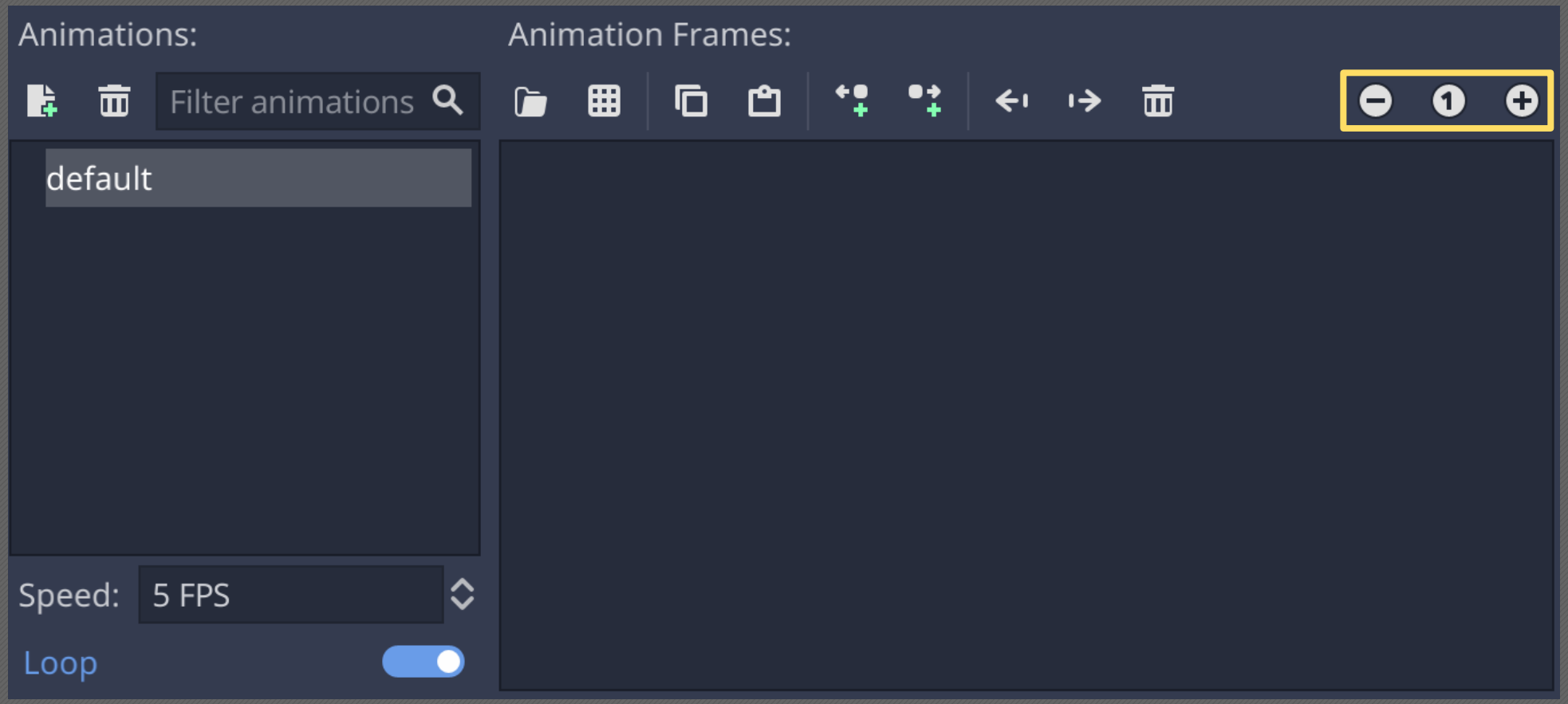
Navigating SpriteFrames

Delete the currently selected frame



Navigating SpriteFrames

Frame view operations (Zoom Out, Reset, Zoom In)

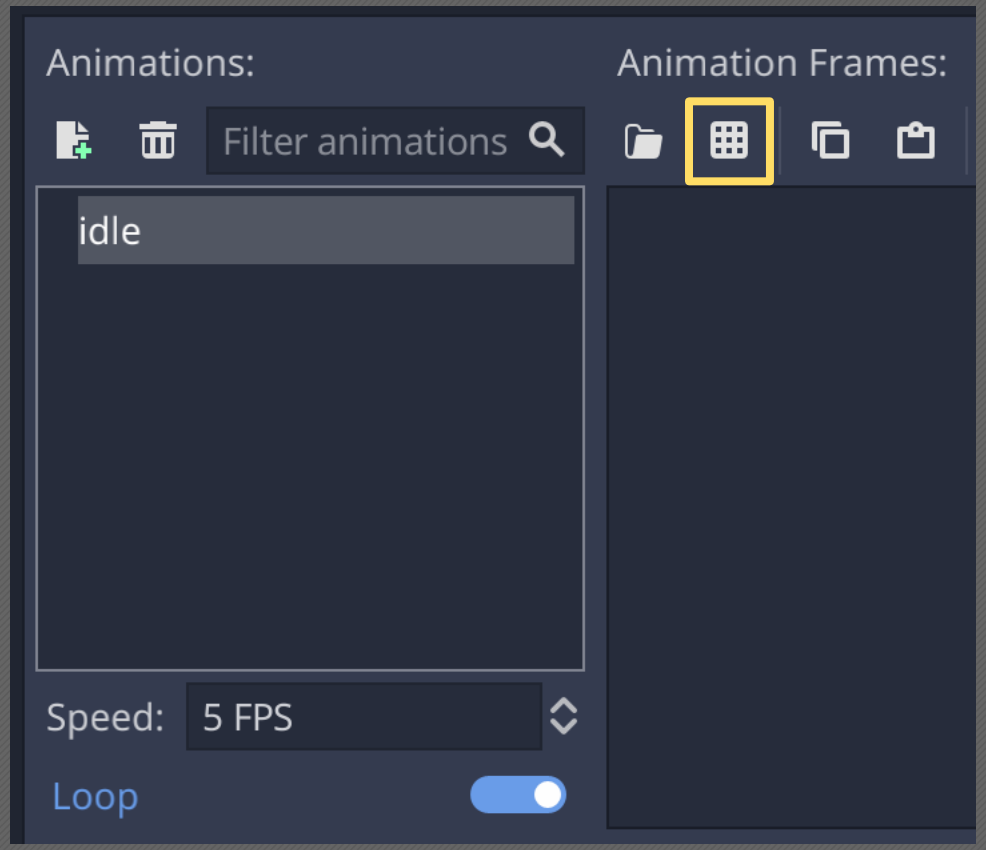


Creating the Idle Animation

- Rename the “default” animation to “idle”.

- Click the “Add Frames from a Sprite Sheet” button to add frames the *currently selected animation*.

Note: You can rename an animation by clicking an already-selected animation in the animation list.



Creating the Idle Animation

Open the “Idle (32x32).png” sprite sheet image from your choice of one of the four character directories.
res://Assets/Art/Free/Main Characters/*{CHARACTER}*/Idle (32x32).png

Mask Dude

Ninja Frog

Pink Man

Virtual Guy

Creating the Idle Animation

Set the Frames/Sprite size to the proper dimensions.

- If you don't know the dimensions, the Horizontal & Vertical arrows are a good option for eyeballing.
- If you know the size, you can set it directly in the Size fields. In our case, the width (first field) and height (second field) are both *32px*.

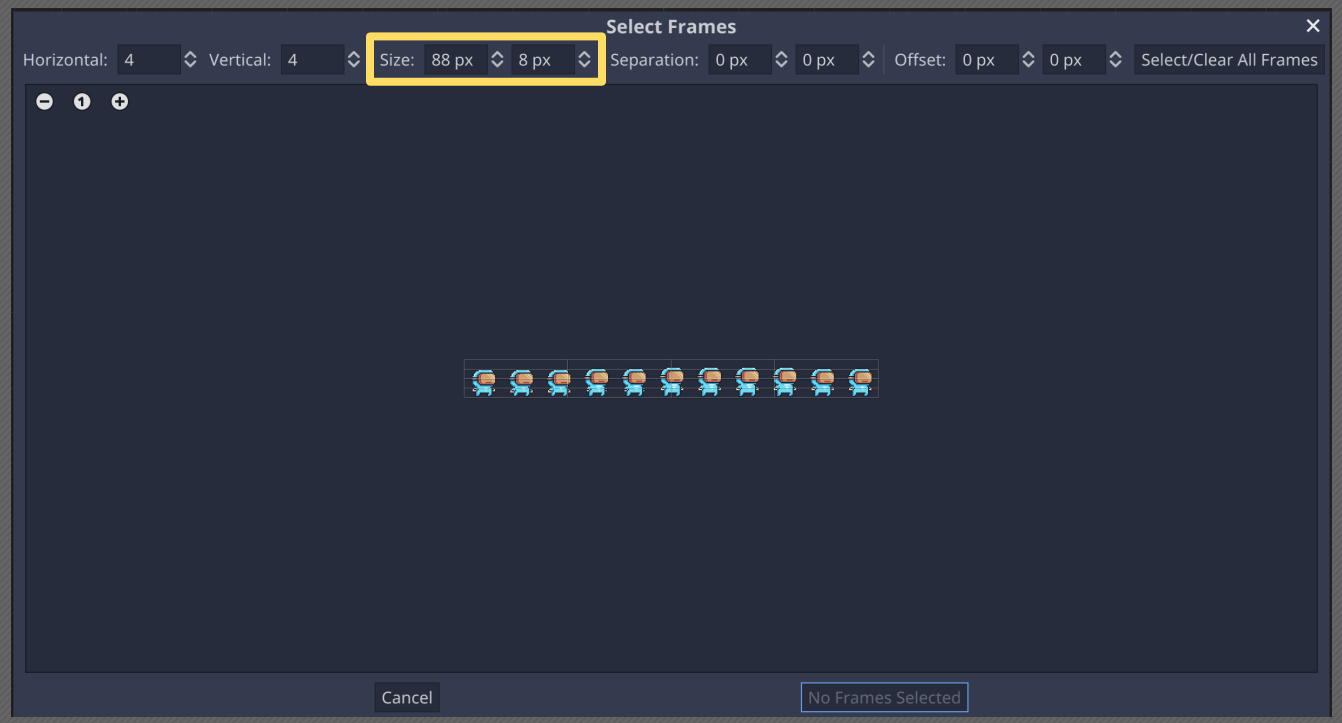
Your sprite sheet & frames should look like this once your dimensions are set properly.

Creating the Idle Animation

Set the Frames/Sprite size to the proper dimensions.

- If you don't know the dimensions, the Horizontal & Vertical arrows are a good option for eyeballing.

- If you know the size of your sprites, you can set it directly in the Size fields. In our case, the width (first field) and height (second field) are both *32px*.



Your sprite sheet & frames should look like this once your dimensions are set properly.

Creating the Idle Animation

- Once your dimensions are set, you then select which frames you want to add to the animation.
- You can select and selected frames via left-click.
- Select all frames and click the "Add 11 Frame(s)" button.

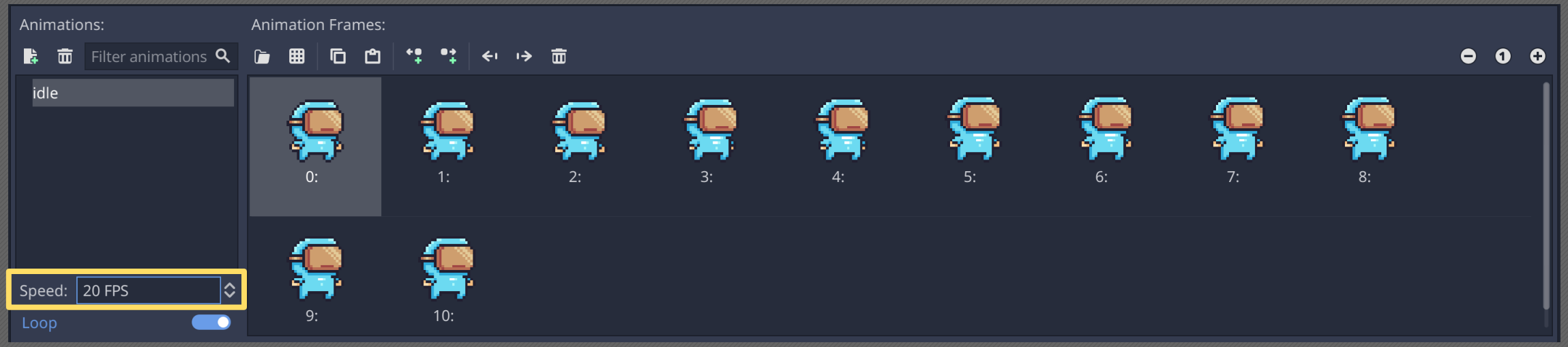
Tip: Use the top right "Select/Clear All Frames" to save on clicks!

Notice the dark background and thin blue border on each frame indicating that it is selected.

Creating the Idle Animation

Your animation should now have the 11 frames we selected in the previous step.

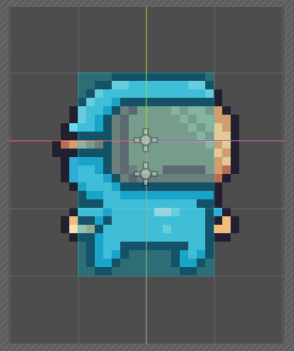
Update the Speed of the “idle” animation to be 20 FPS and keep Loop toggled “On”.



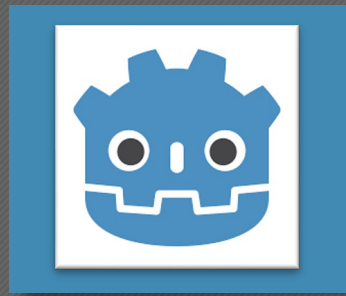
Test the Idle Animation

In the scene tree of your Player scene:

- Delete the Sprite node.
- Set the AnimatedSprite node's "Playing" property to be toggled On.
- Update the CollisionShape2D to roughly match the size of the character.
- Run the scene!



Triggering Player Animations



In the Player script, add this code after the *move_and_slide(...)* call in the _physics_process func.

You can reference scene tree nodes by using a $ symbol followed by the *full* node path in the scene tree.

On the AnimatedSprite node, we can call the play method to play a specific animation that exists on the SpriteFrames resource.

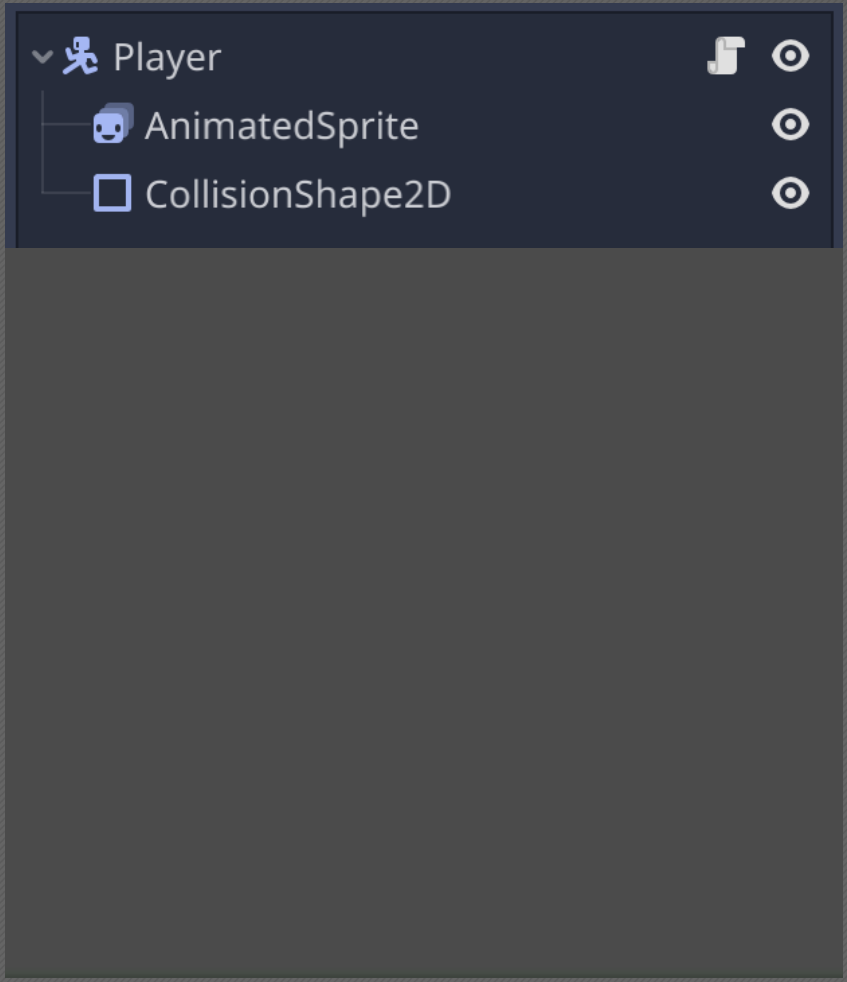
```
if move_left > 0:
    $AnimatedSprite.flip_h = true
elif move_right > 0:
    $AnimatedSprite.flip_h = false

if velocity.y < 0:
    $AnimatedSprite.play("jump")
elif velocity.y > 0:
    $AnimatedSprite.play("fall")
elif velocity.x != 0:
    $AnimatedSprite.play("run")
else:
    $AnimatedSprite.play("idle")
```

Lab Time & Break (~15 Minutes)

- Add the following 4 animations to your character:
  - fall
  - jump
  - idle
  - run

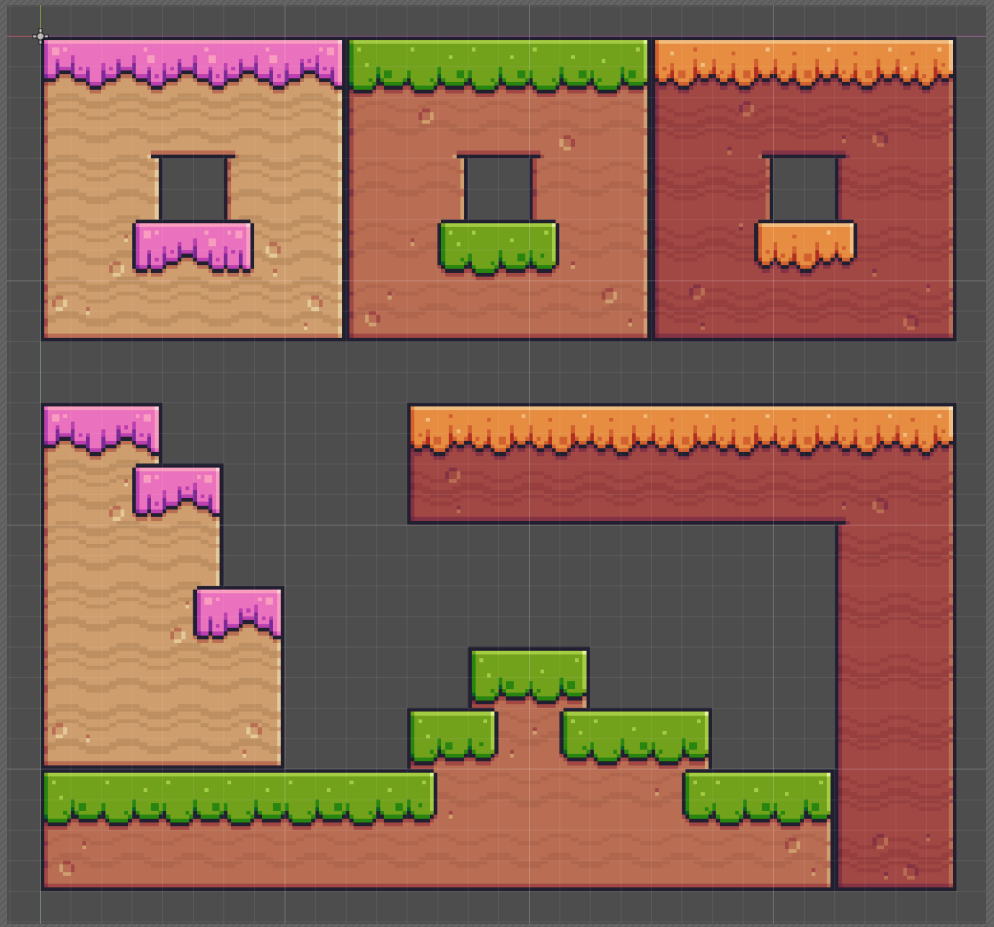
Ensure that the idle and run animations are set to Loop and have a Speed value of 20 FPS!

Update the Player script with the code from the previous slide.



Session Goal #2: Reusable Base Level

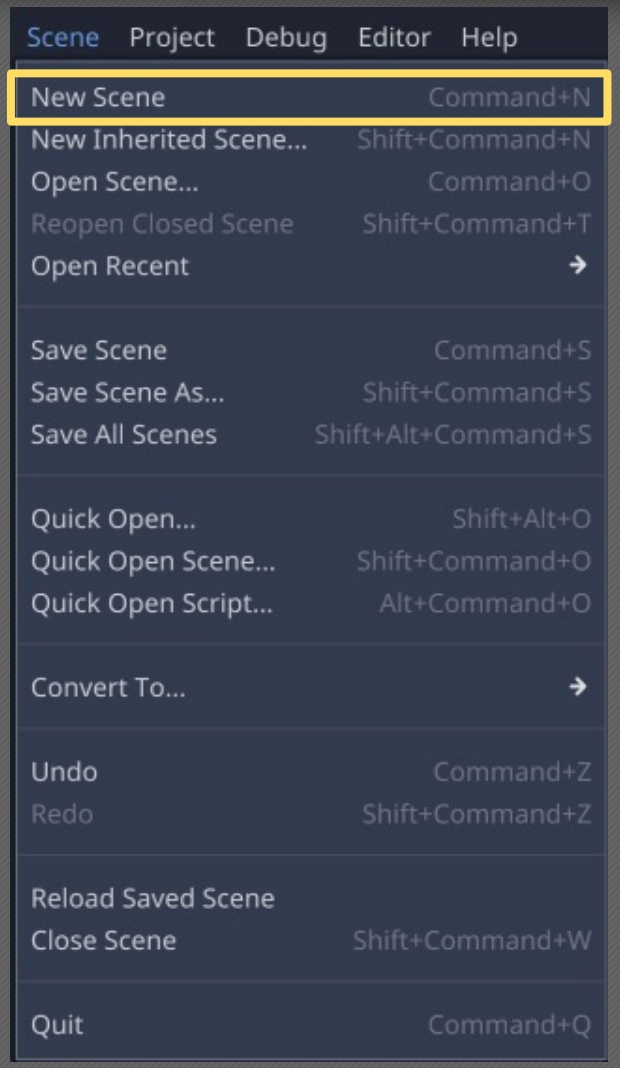
1. ☑ Controllable Player
2. Reusable Base Level
3. Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. Level Checkpoints
5. Hazards & Player Respawn
6. Collectibles & HUD
7. A Basic Enemy
8. Main Menu & Changing Scenes
9. Open Workshop, Tinker Time

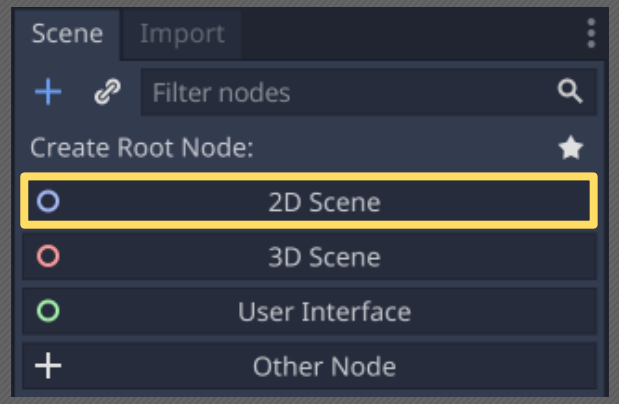


Creating a base Level Scene

We can't easily add ground or platforms to the Player, since they would move *with* the Player.

- Create a New Scene via the Scene menu.
- Use Node2D as the root node for our new Scene.
- Rename this new Scene's Node2D root node as "Level".





Creating a base Level Scene

- Add some new nodes mimicking the structure shown to the right and rename them as shown.

- Make sure the *World* and *Ground* nodes are nested child nodes, not siblings!

- On the Ground TileMap node, set the Cell Size property to be 16x16 instead of 64x64 (in the Inspector window).

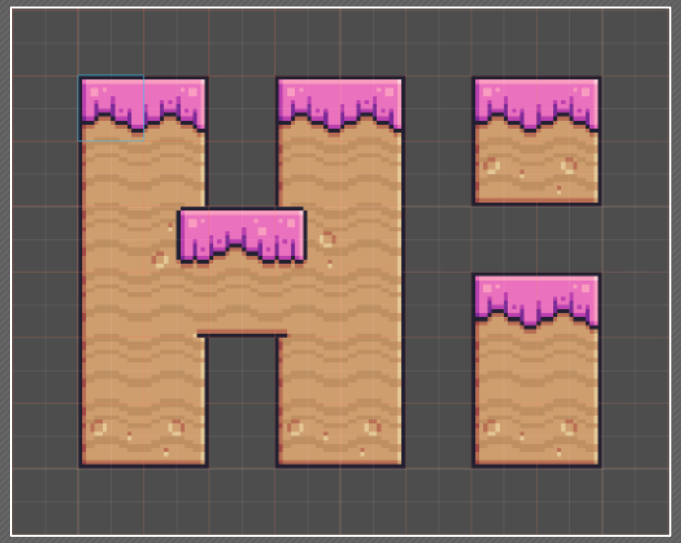
- Set the Tile Set property to be a New Tileset. Once created, click that new value and save the TileSet (similar to what we did for the AnimatedSprite).

Tiles, TileSets, & TileMaps

- A Tile is a single image (a sprite) representing something in the game.
- A TileSet is a library of Tiles (like a logical sprite sheet) to be used in a TileMap.
- You draw the Tiles from one or more TileSets onto a TileMap.

TileSet

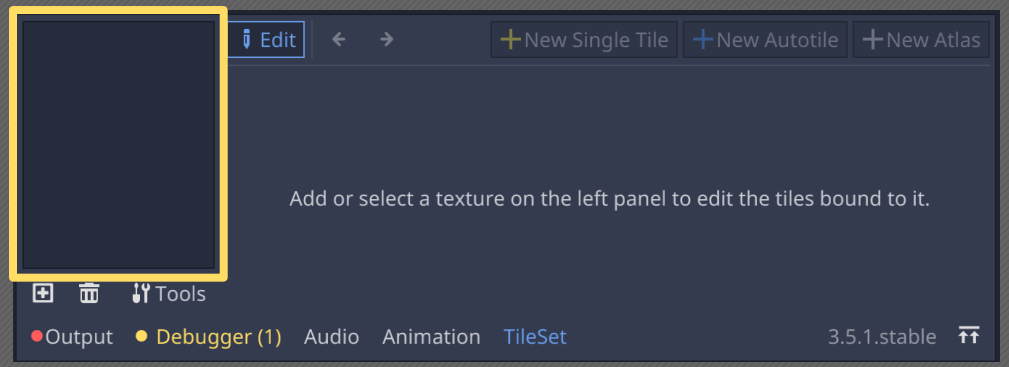
TileMap

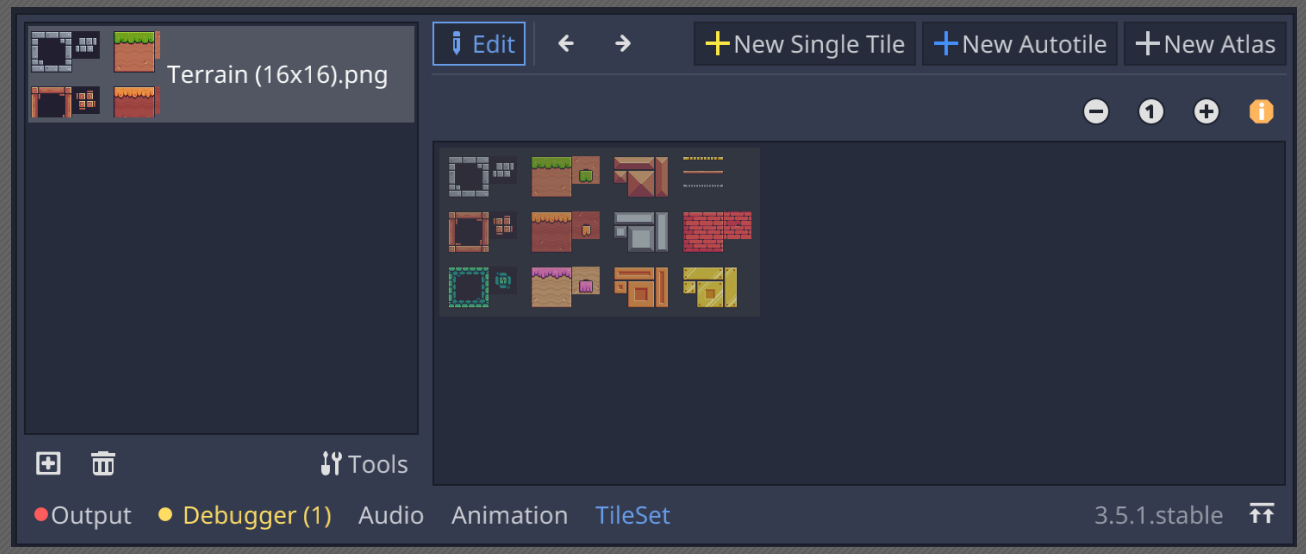


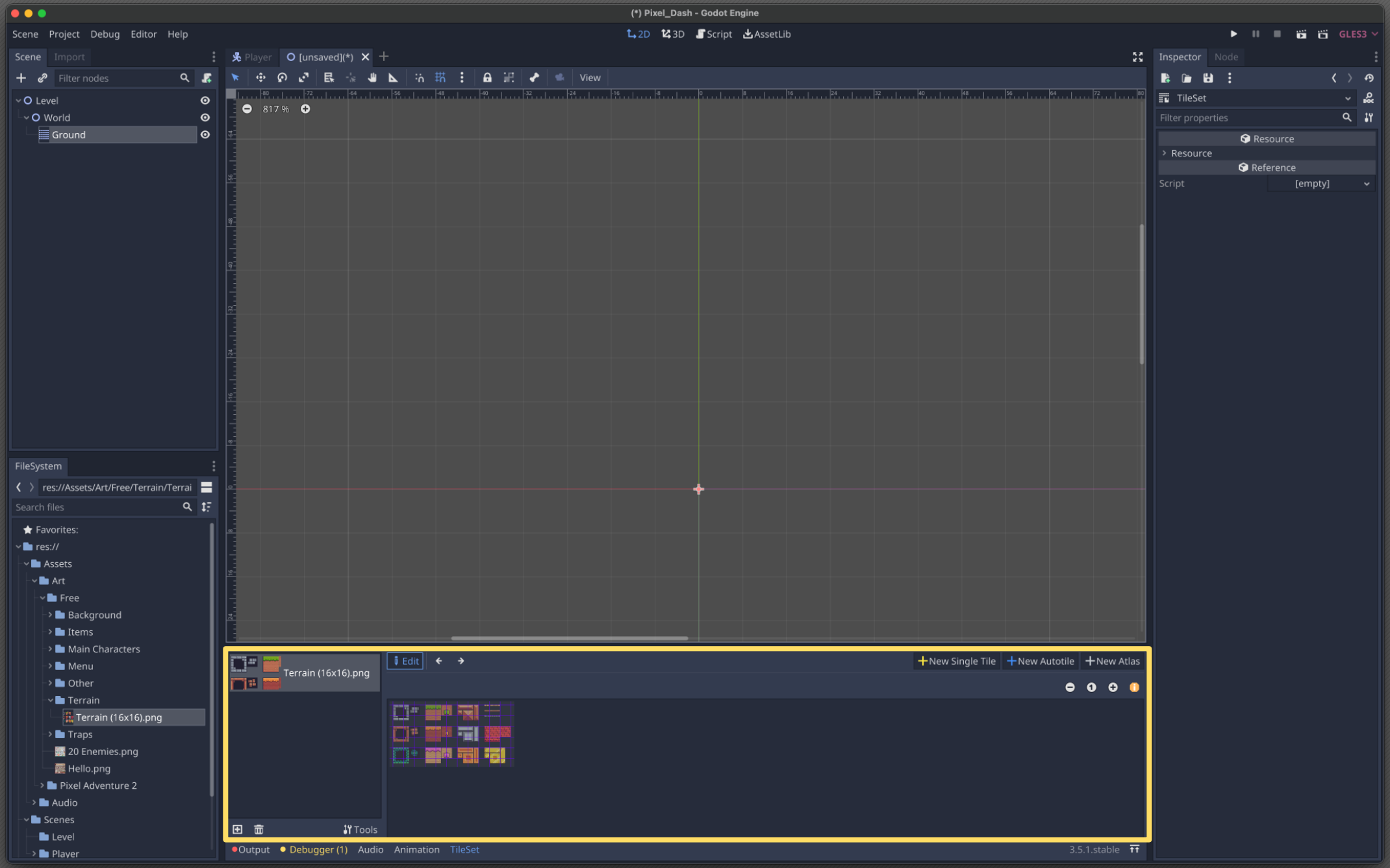
Creating the Level's Borders TileSet

- **Drag** the following file (from the FileSystem tab) into the dark blue box at the bottom of the screen:

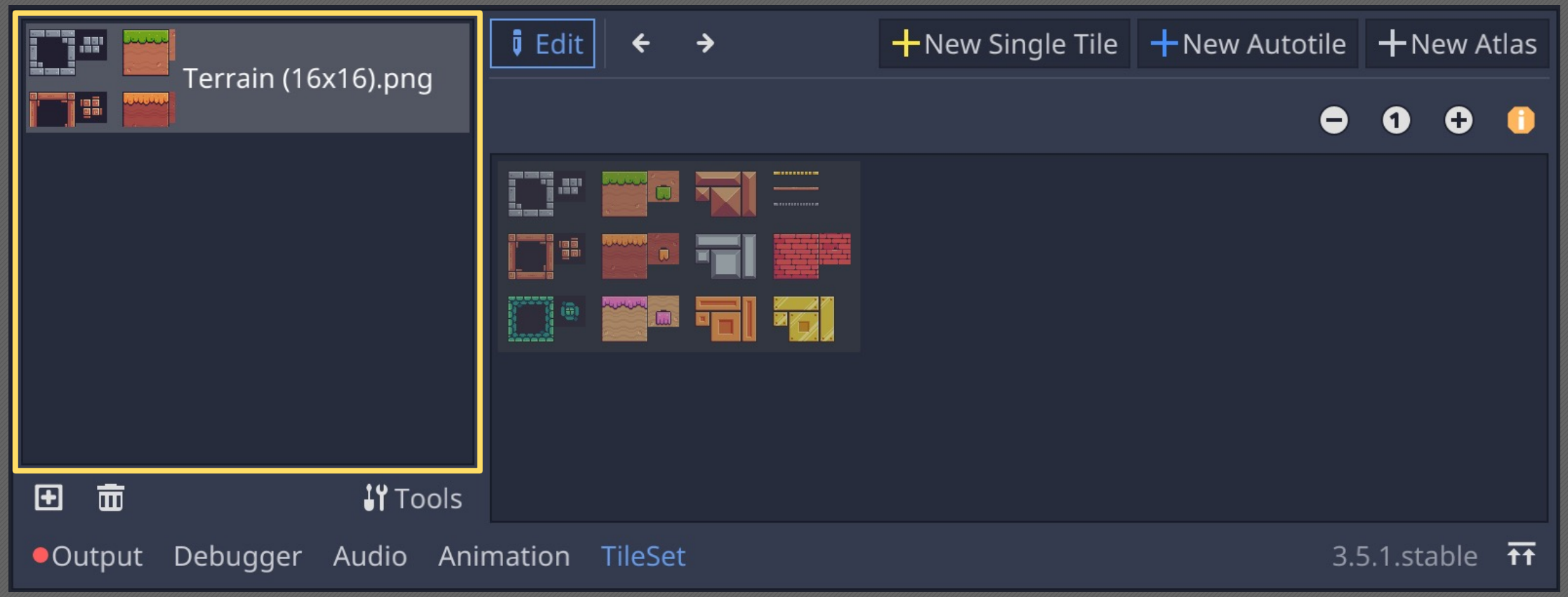
res://Assets/Art/Free/Terrain/Terrain (16x16).png



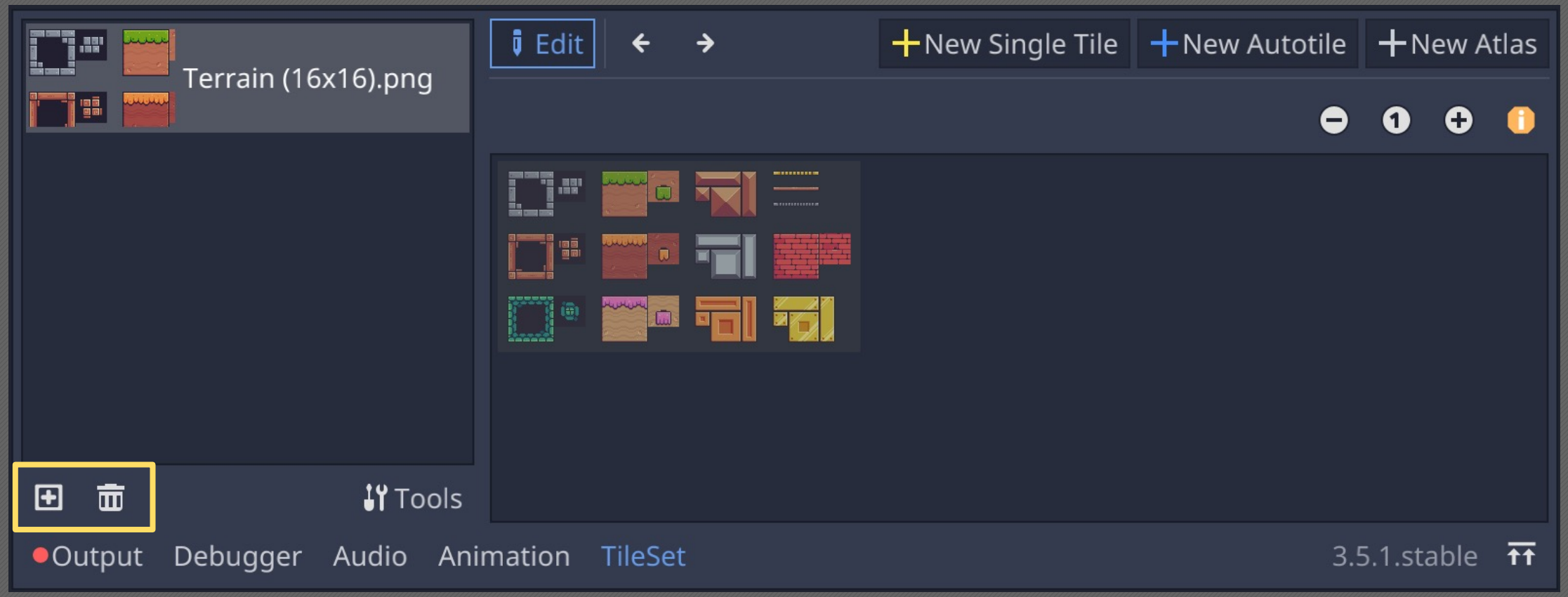



(*) Pixel_Dash - Godot Engine
Scene Project Debug Editor Help
2D 3D Script AssetLib
GLES3
Scene Import
Filter nodes
Level
World
Ground
Player [unsaved](*)
View
817 %
Inspector Node
TileSet
Filter properties
Resource
Resource
Reference
Script [empty]
FileSystem
res://Assets/Art/Free/Terrain/Terrai
Search files
Favorites:
res://
Assets
Art
Free
Background
Items
Main Characters
Menu
Other
Terrain
Terrain (16x16).png
Traps
20 Enemies.png
Hello.png
Pixel Adventure 2
Audio
Scenes
Level
Player
Terrain (16x16).png
Edit
New Single Tile
New Autotile
New Atlas
Tools
Output
Debugger (1)
Audio
Animation
TileSet
3.5.1.stable

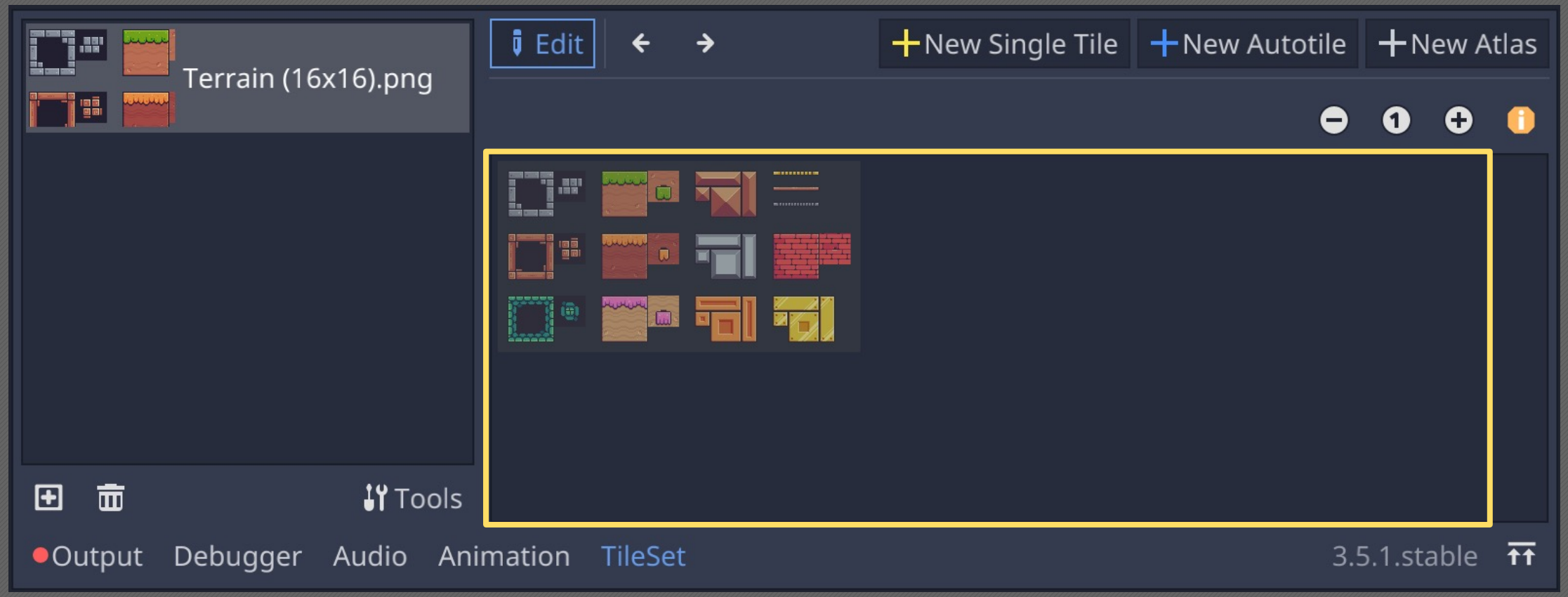
TileSet Textures (sprite sheets) List



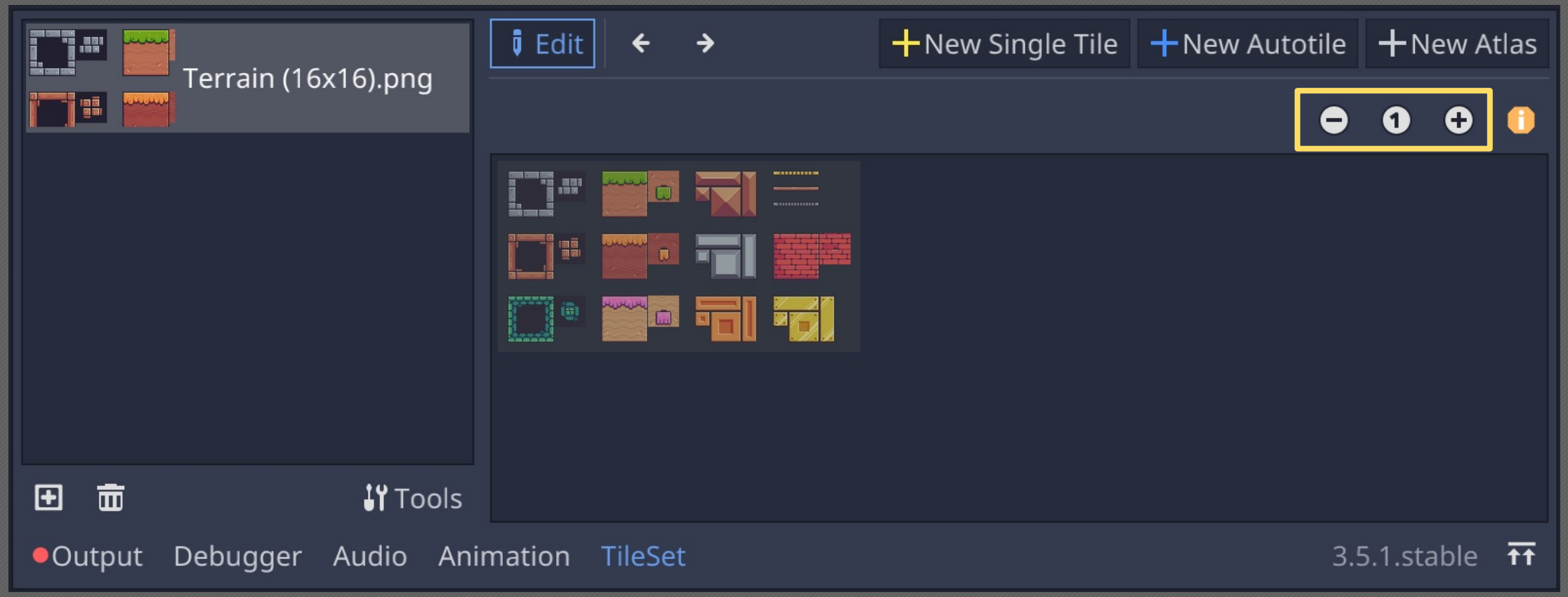
Add/Delete Texture



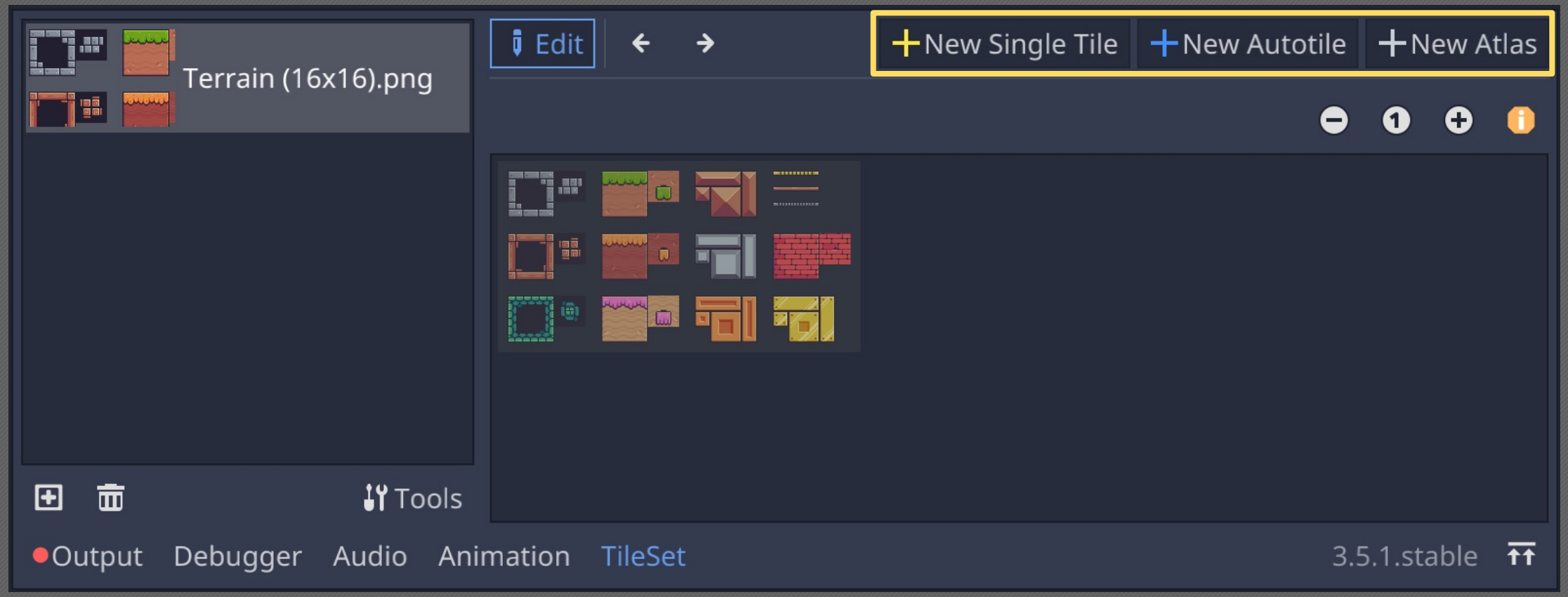
TileSet Editor Area



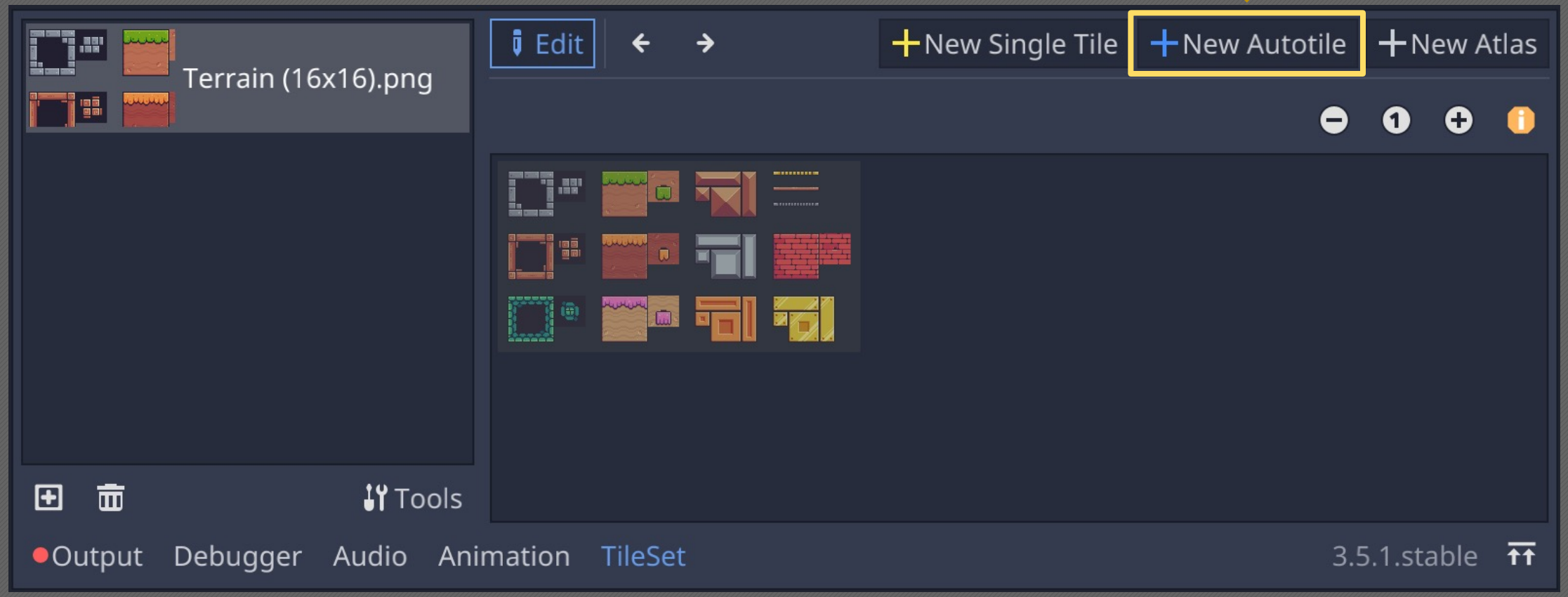
Zoom Out, Reset, Zoom In Buttons



“New Tile” Buttons



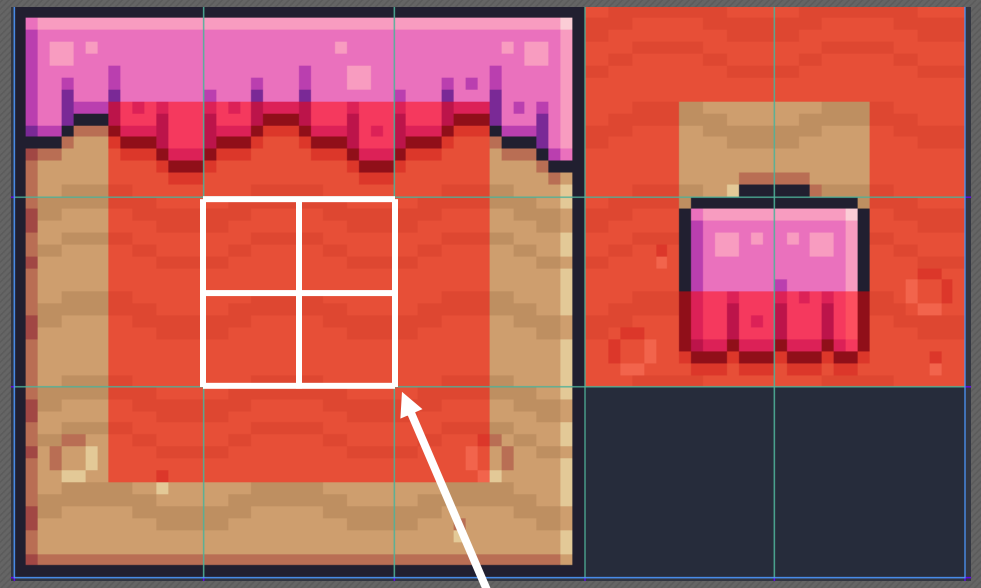
We will only be creating Autotiles today!



AutoTiles and Bitmasks

- An Autotile is a tile that is configured to style itself based on the presence of neighbor tiles.

- A Bitmask (in the context of a TileSet) is a grid-based “on/off” map identifying which tile should display in each scenario.

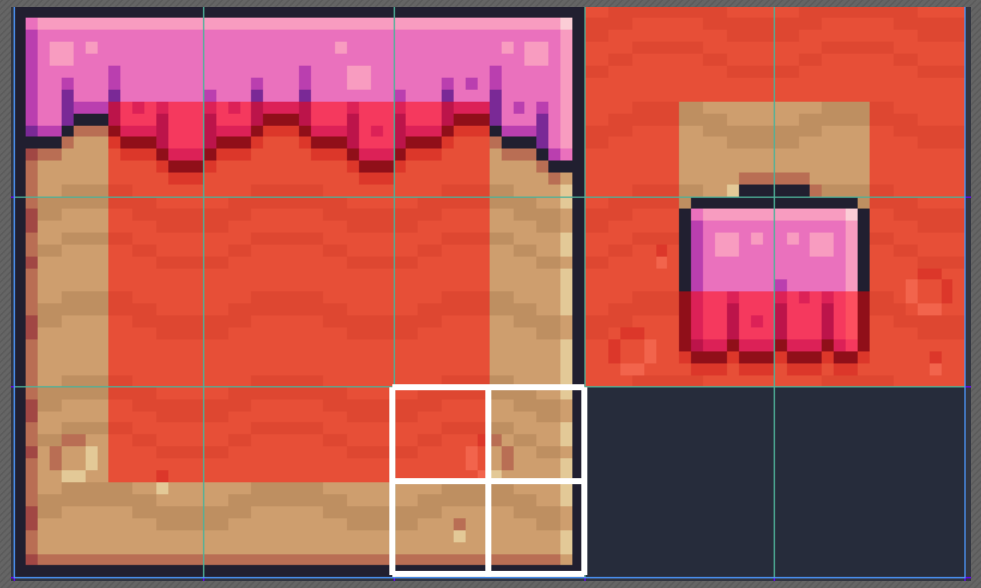


Use this tile when it’s surrounded by neighbors.

AutoTiles and Bitmasks

- An Autotile is a tile that is configured to style itself based on the presence of neighbor tiles.

- A Bitmask (in the context of a TileSet) is a grid-based “on/off” map identifying which tile should display in each scenario.

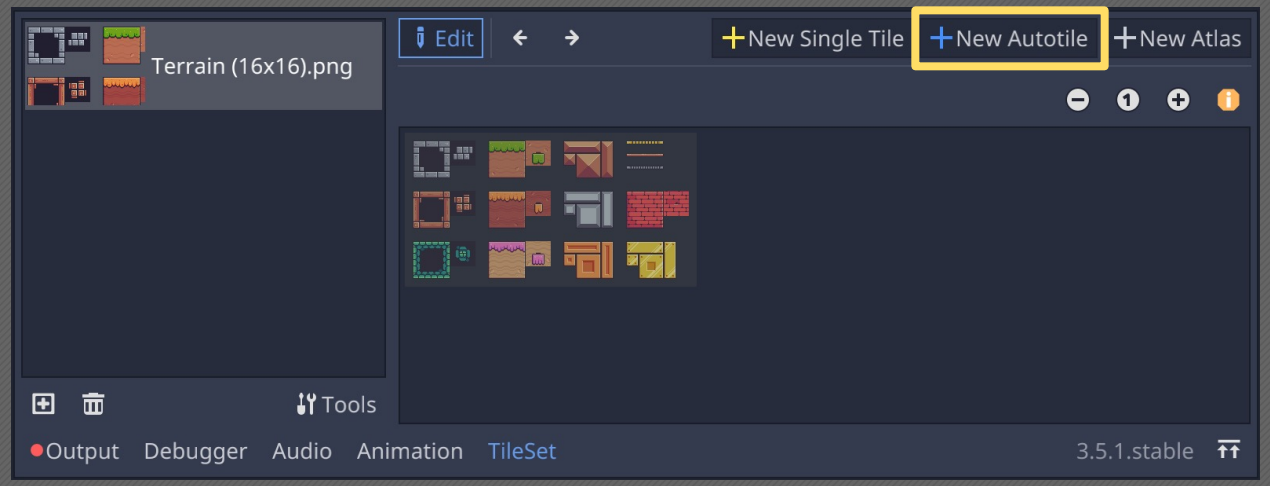


Use this tile when it has all 3 neighbors in its top-left corner (top, top-left, and left).

Setting up an Autotile can be tricky!

To create a new AutoTile:

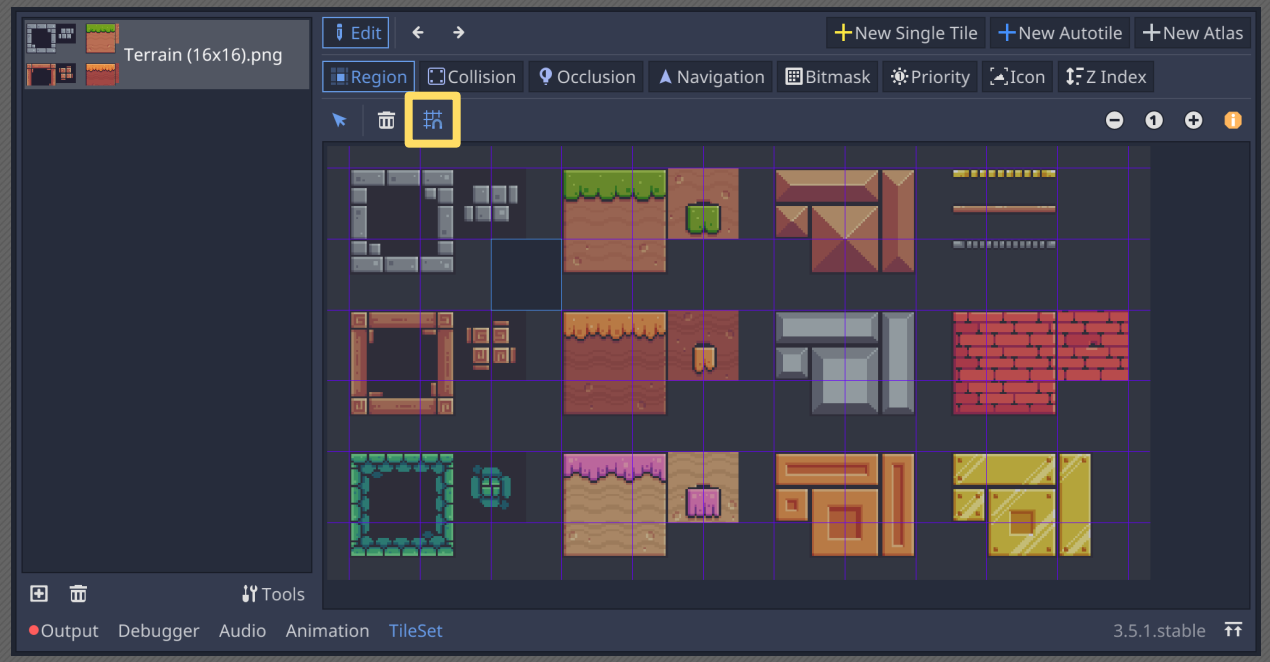
1. Click “New Autotile”.
2. Toggle on “Snap to Grid” and select any square in the sprite sheet.
3. In the TileSet properties (in the right Inspector tab), update the Step and Subtile Size properties (x and y) to all be 16.
4. In the TileSet editor area, reselect an entire AutoTile “section” by dragging left-click.



Setting up an Autotile can be tricky!

To create a new AutoTile:

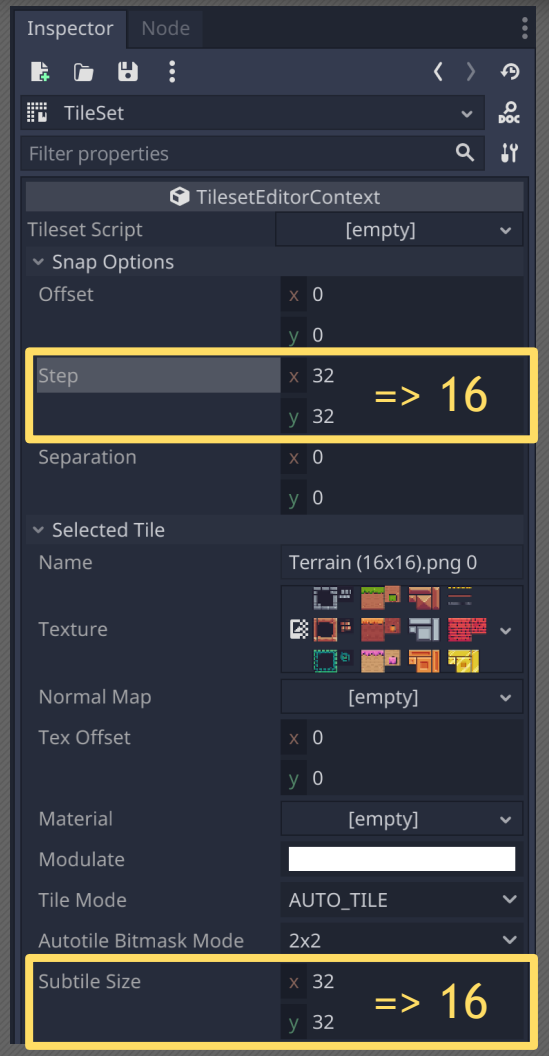
1. Click “New Autotile”.
2. Toggle on “Snap to Grid” and select any square in the sprite sheet.
3. In the TileSet properties (in the right Inspector tab), update the Step and Subtile Size properties (x and y) to all be 16.
4. In the TileSet editor area, reselect an entire AutoTile “section” by dragging left-click.



Setting up an Autotile can be tricky!

To create a new AutoTile:

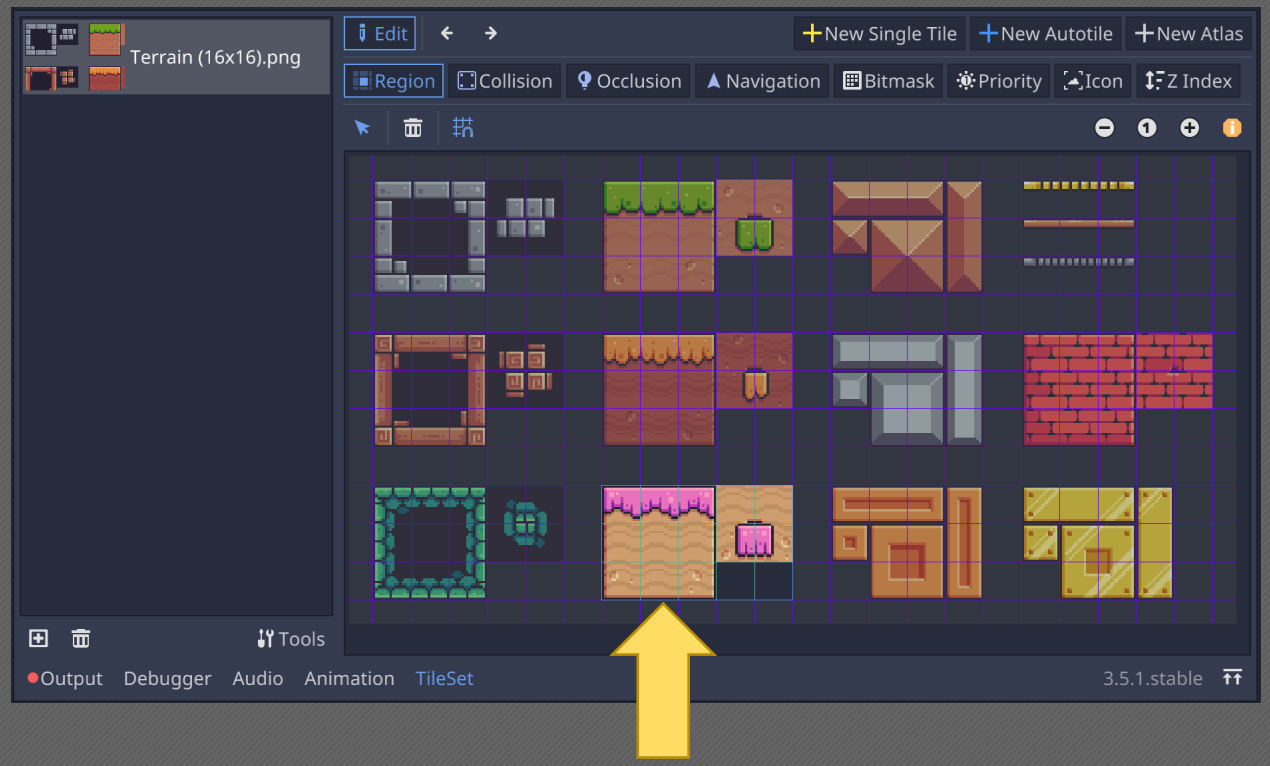
1. Click “New Autotile”.
2. Toggle on “Snap to Grid” and select any square in the sprite sheet.
3. In the TileSet properties (in the right Inspector tab), update the Step and Subtile Size properties (x and y) to all be 16.
4. In the TileSet editor area, reselect an entire AutoTile “section” by dragging left-click.



Setting up an Autotile can be tricky!

To create a new AutoTile:

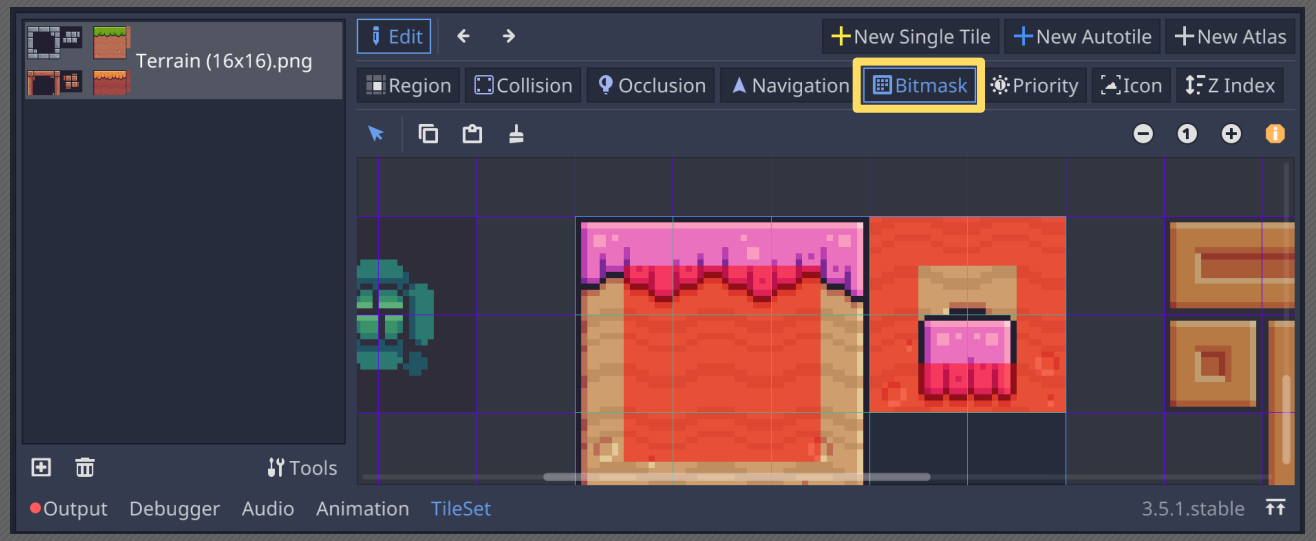
1. Click “New Autotile”.
2. Toggle on “Snap to Grid” and select any square in the sprite sheet.
3. In the TileSet properties (in the right Inspector tab), update the Step and Subtile Size properties (x and y) to all be 16.
4. In the TileSet editor area, reselect an entire AutoTile “section” by dragging left-click.



The subtle green outline highlights your current selection.

Defining the Autotile Bitmask

- Select the Bitmask tab.
- Using left/right click to toggle the masks on/off respectively, create the Bitmask as seen to the right.
- Create a new Autotile for all 3 of the “wall” sprites in the left column (a total of 3 autotiles)

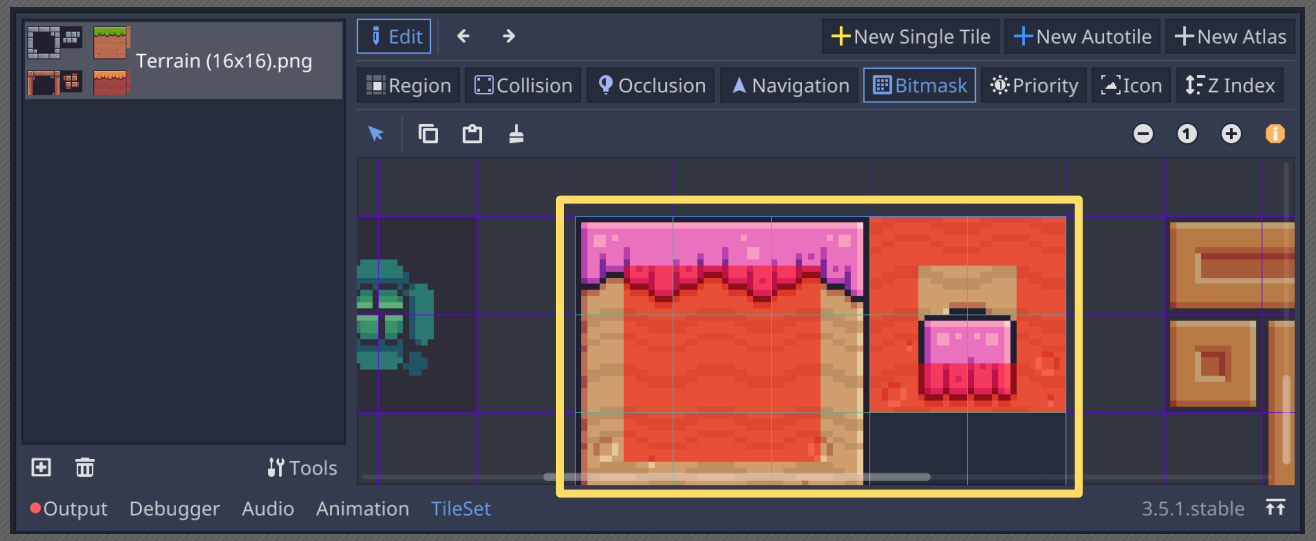


Warning: You can redraw the Region at any time while the Region tab is active - be careful where you click!

(Mac Users: Press *Shift + Cmd* if you’re seeing a blue checker pattern instead of red squares)

Defining the Autotile Bitmask

- Select the Bitmask tab.
- Using left/right click to toggle the masks on/off respectively, create the Bitmask as seen to the right.
- Create a new Autotile for all 3 of the “wall” sprites in the left column (a total of 3 autotiles)



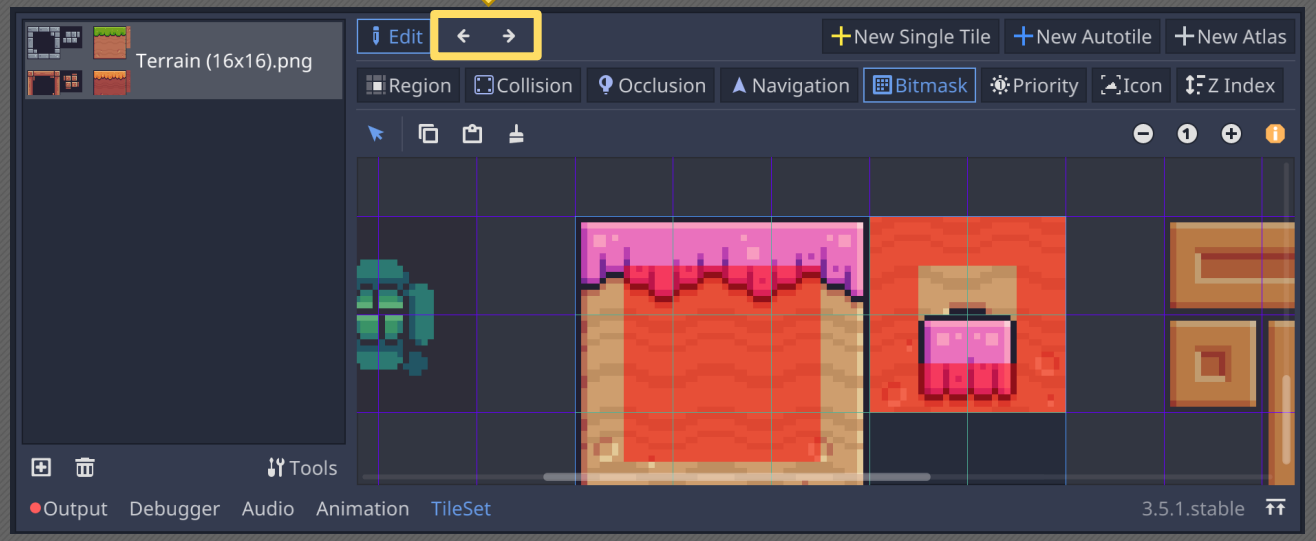
Warning: You can redraw the Region at any time while the Region tab is active - be careful where you click!

(Mac Users: Press *Shift + Cmd* if you’re seeing a blue checker pattern instead of red squares)

Defining the Autotile Bitmask

- Select the Bitmask tab.
- Using left/right click to toggle the masks on/off respectively, create the Bitmask as seen to the right.
- Create a new Autotile for all 3 of the “wall” sprites in the left column (a total of 3 autotiles)

Use these arrows to navigate between tiles and autotiles.

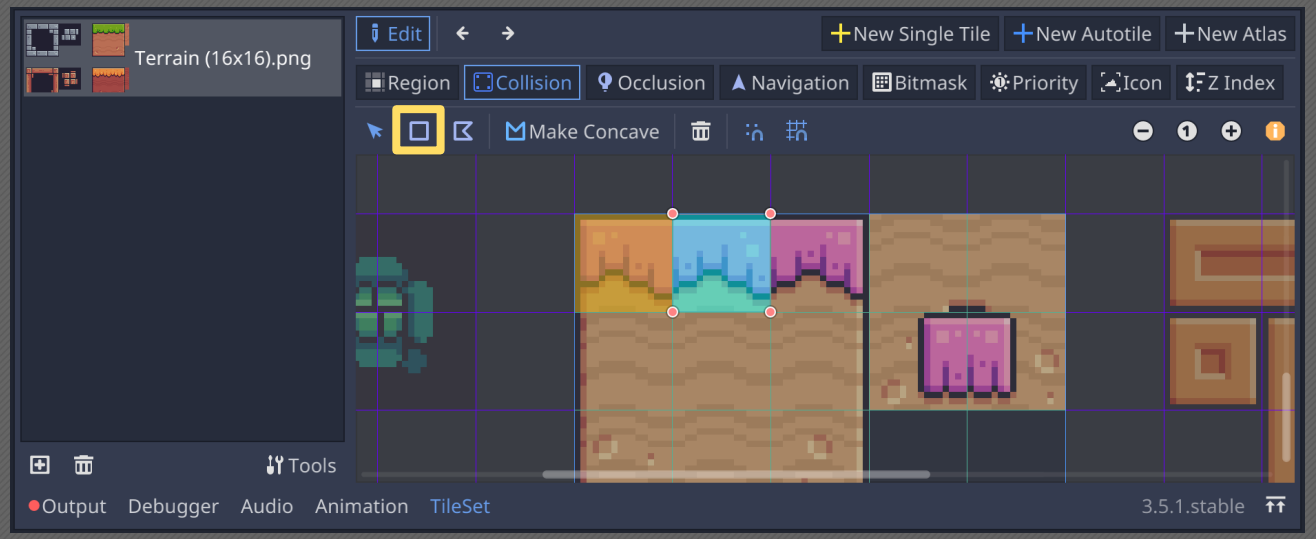


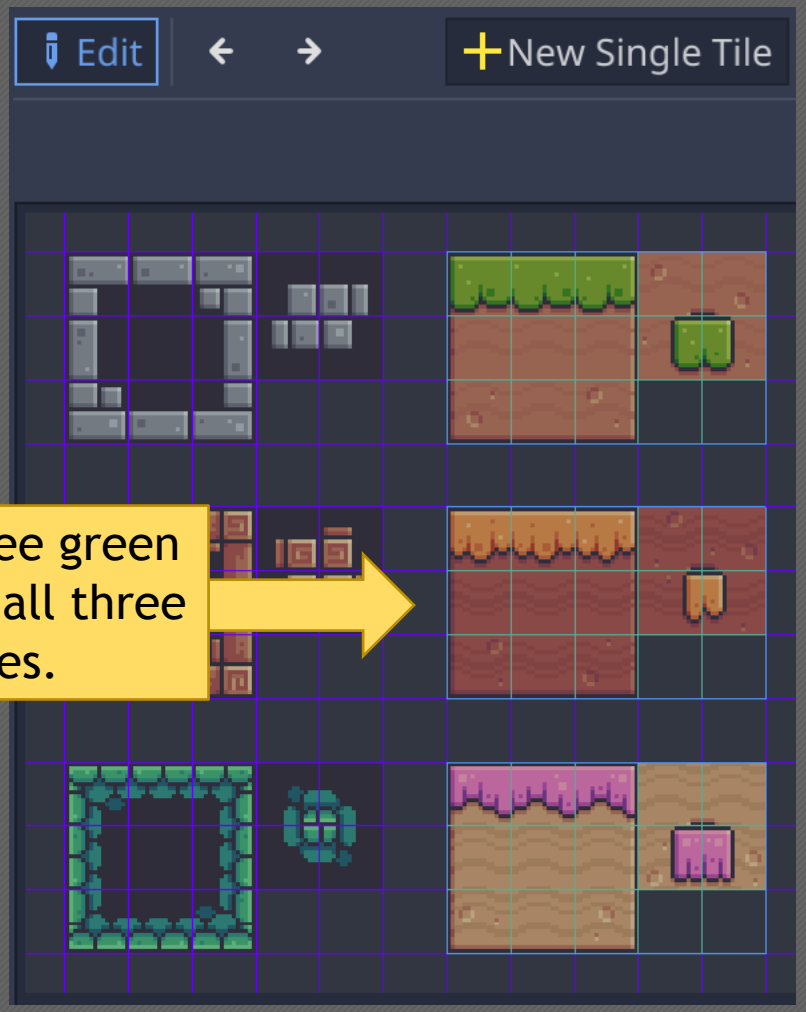
Warning: You can redraw the Region at any time while the Region tab is active - be careful where you click!

(Mac Users: Press *Shift + Cmd* if you’re seeing a blue checker pattern instead of red squares)

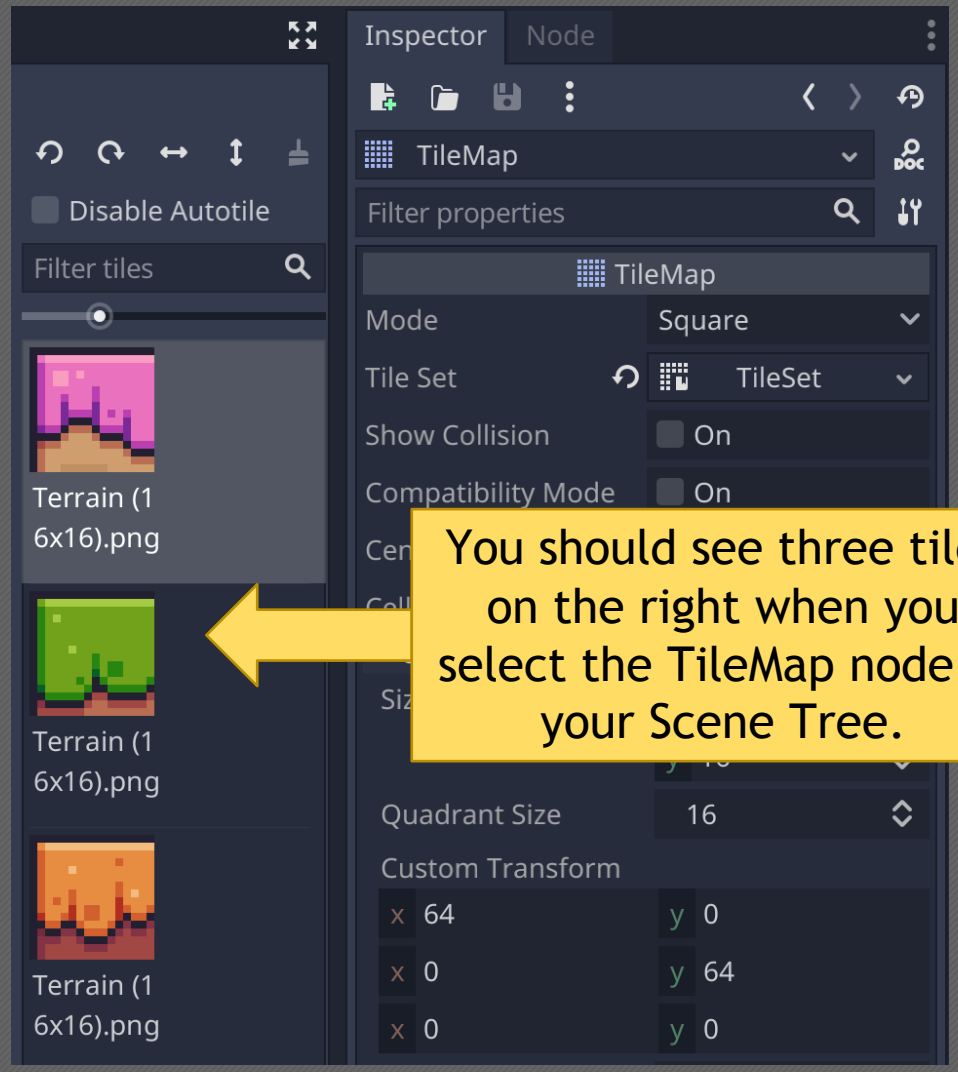
Adding Collision to the autotiles

- Select the Collision tab.
- Select a tile you want to add collision to.
- Select which collision shape you want to add (use the rectangle shape).
- Click in the selected shape again to create the full shape.
- Do this for each individual tile on all three autotiles.



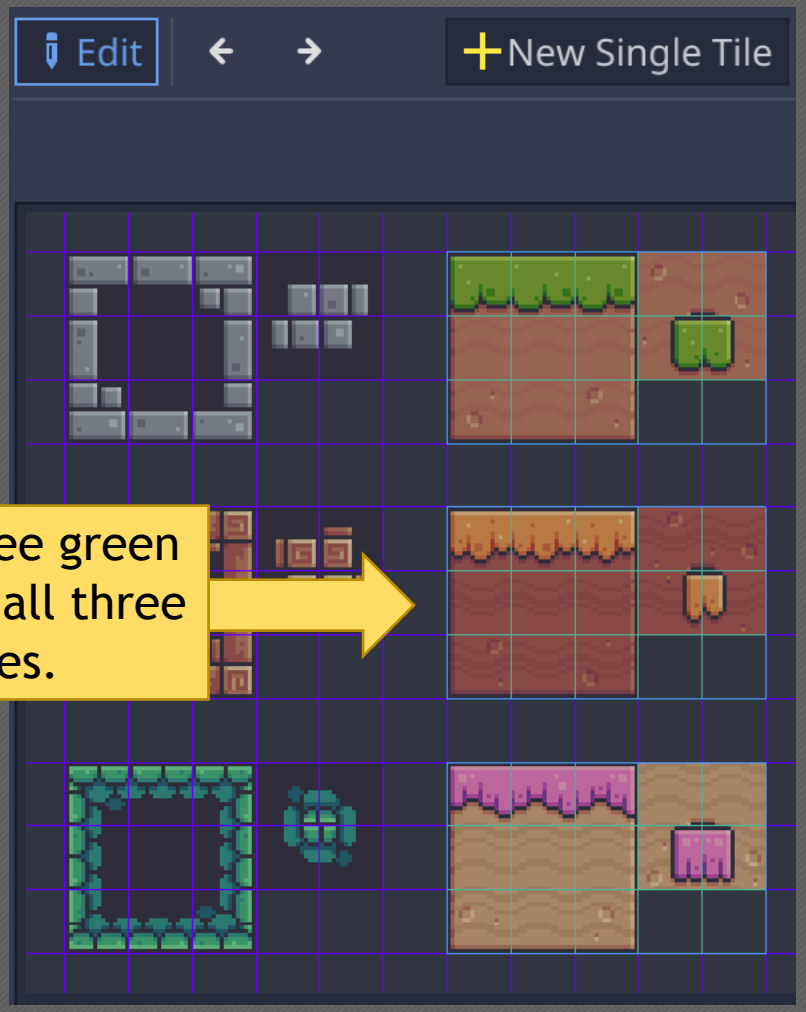


You should see green lines around all three autotiles.

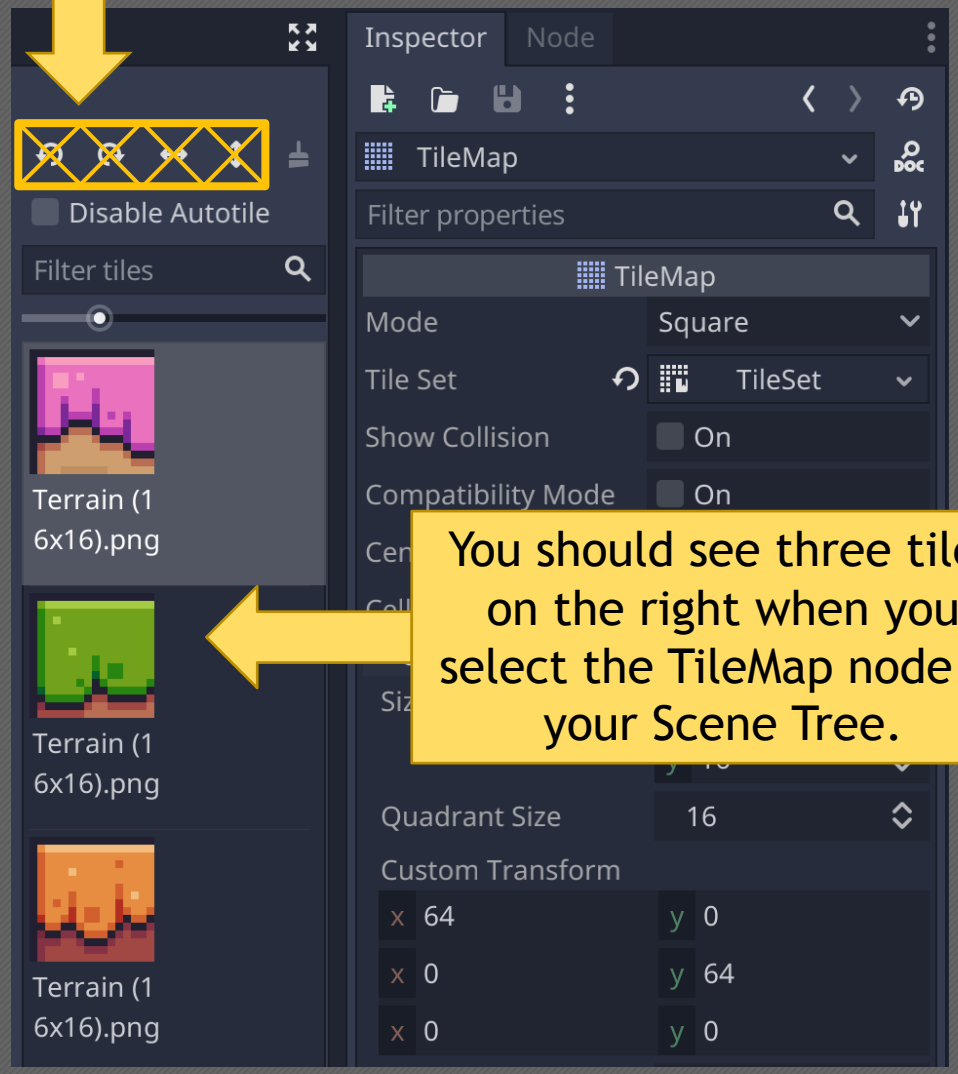


You should see three tiles on the right when you select the TileMap node in your Scene Tree.

Your tiles will automatically draw themselves; DO NOT rotate or shift your tiles!



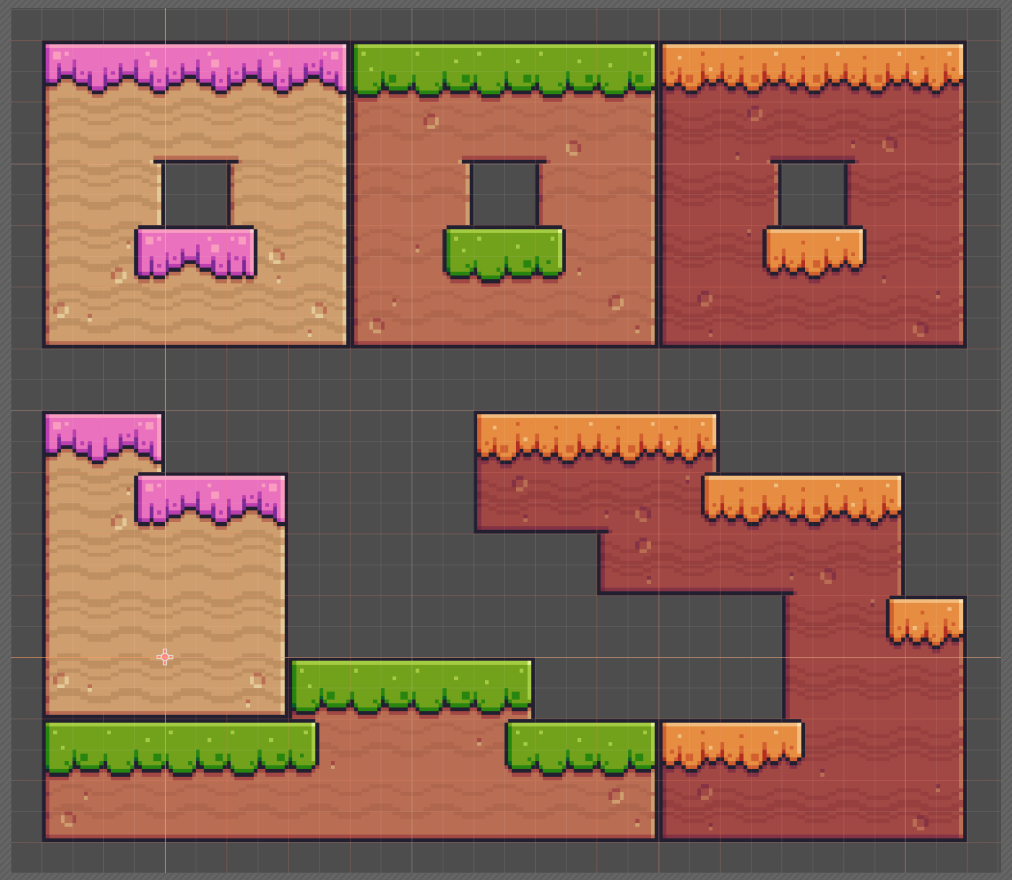
You should see green lines around all three autotiles.



You should see three tiles on the right when you select the TileMap node in your Scene Tree.

Drawing Your Tiles

- With your TileMap selected:
  - Select one of the tiles on the right of the screen.
  - Left-click to draw your tiles in the scene.
  - Right-click to remove a tile.
  - Your tiles should automatically update as they're drawn!
  - All tilemap tiles are built on the same "layer".



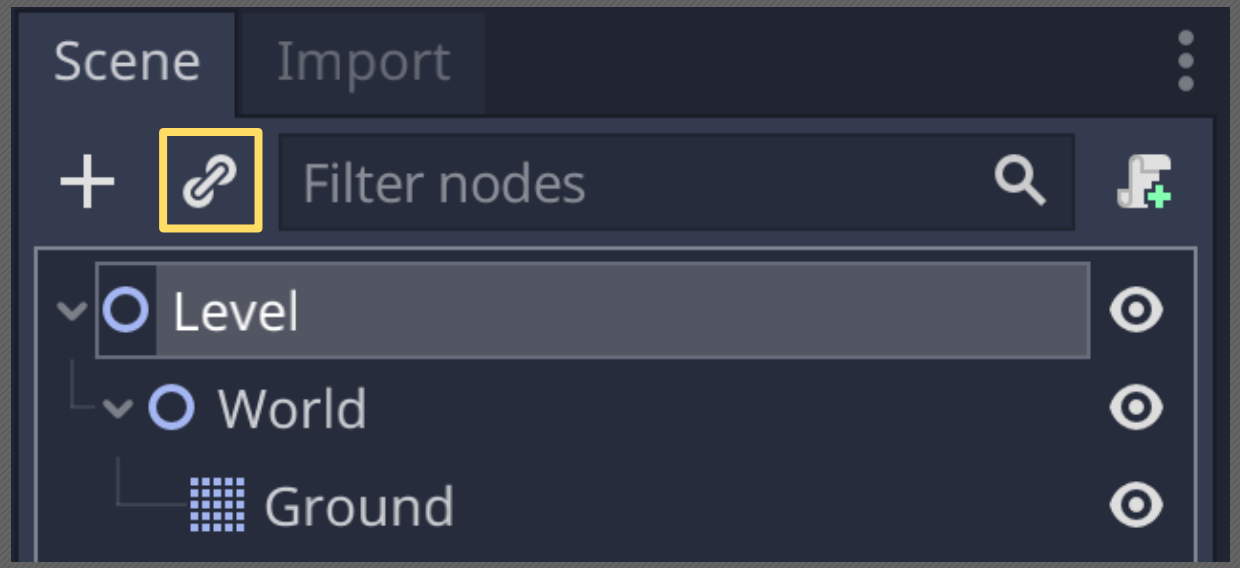
Pro Tip: Shift+Click to draw straight lines!

Instancing Child Scenes

The Instance Child Scene button allows you to instantiate a saved scene into the current scene tree.

Any changes made to a source scene will have a ripple effect in other scenes it is instanced into.

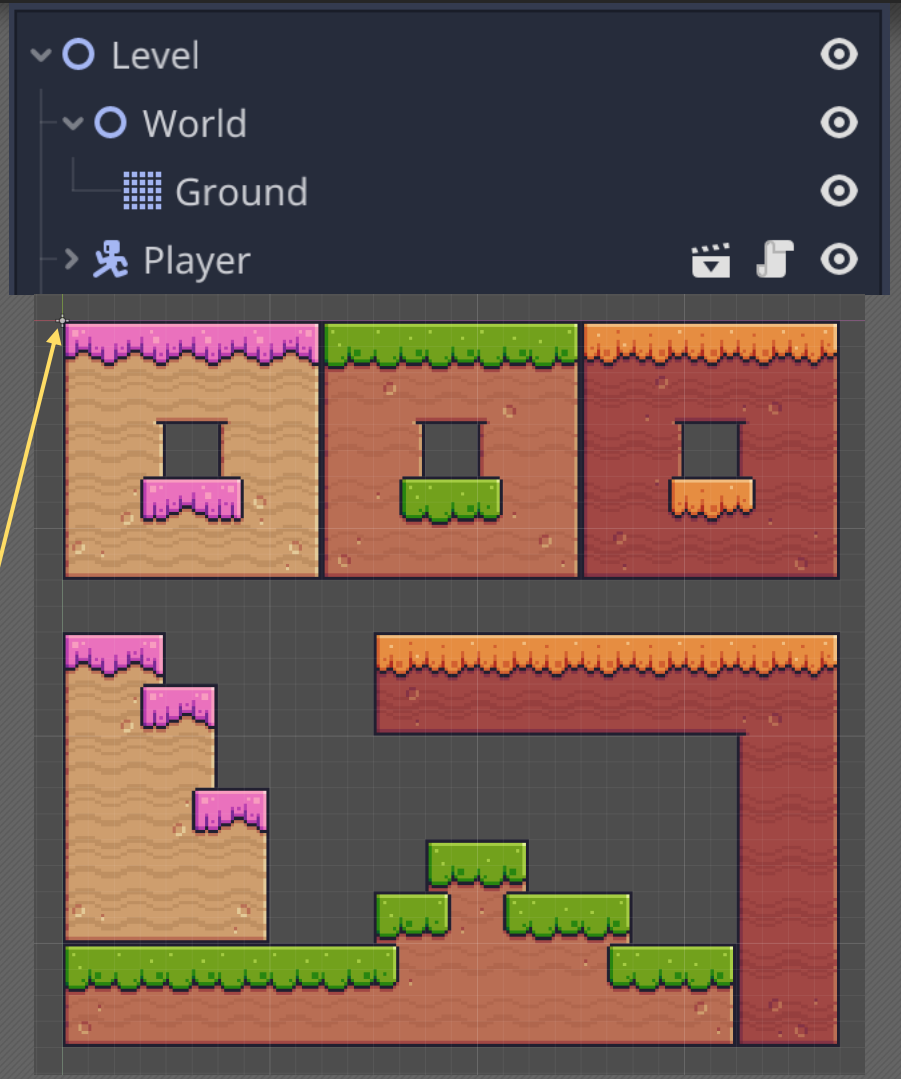
- Select the root Level node.
- Click the Instance Child Scene button and select the saved Player.tscn scene file.



Lab Time (~15 Minutes)

- Create a new Level scene.
- Create the three Ground autotiles, all with proper bitmasks and collision.
- Instance in the Player scene as a child node of the Level scene.
- Draw a small level & run!

Draw all tiles to the bottom-right of the center-point (0,0) of the Level scene!



Session Goal #3: Player & Level Polish

1. ✅ Controllable Player
2. ✅ Reusable Base Level
3. Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. Level Checkpoints
5. Hazards & Player Respawn
6. Collectibles & HUD
7. A Basic Enemy
8. Main Menu & Changing Scenes
9. Open Workshop, Tinker Time

Create a Camera that Follows the Player

- Add a Camera2D node as child of your instanced Player scene node in your Level scene.
- A Camera Zoom value of 1 or less will zoom *in*, where a value above 1 will zoom *out*.
- Limit sets the screen boundaries that your camera will not extend beyond.

Update the Camera2D Properties

Camera2D:

Current: ✅ On

Zoom: x = 0.5, y = 0.5

Limit:

Left: 0 Top: 0

Right: 1024 Bottom: 768

Editor:

Draw Limits: ✅ On

Add Gravity & Jump to the Player Scene

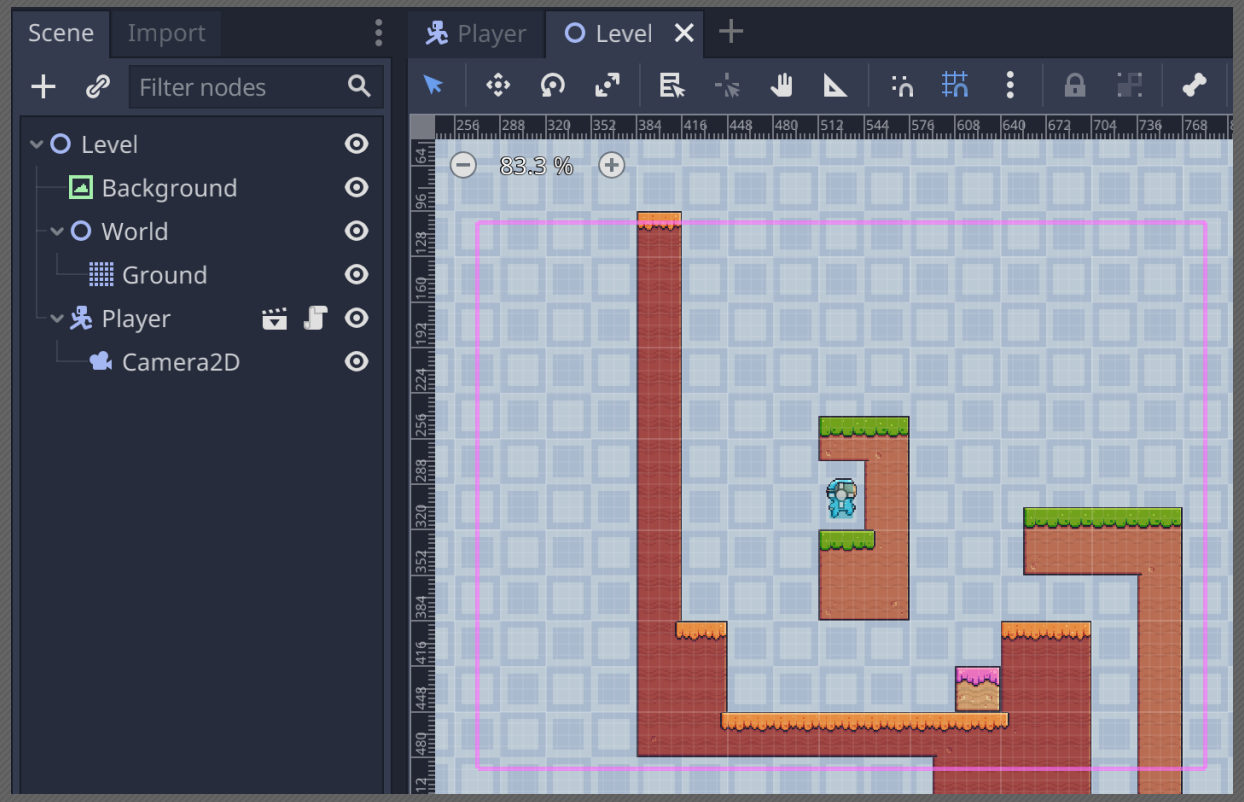
```
const GRAVITY = 25
const JUMP_FORCE := 400

func _physics_process(delta):
    # ...
    # Remove move_up and move_down code, add GRAVITY
    velocity.y += GRAVITY
    velocity = move_and_slide(velocity, Vector2.UP)


func _input(event):
    if event.is_action_pressed("ui_accept") and is_on_floor():
        velocity.y -= JUMP_FORCE
```

Add a Background to the Level Scene

- Add a TextureRect node as an immediate child of Level and name it “Background”.
- Set the Background’s Stretch Mode to ”Tile” & Rect Size to 1024x768.
- Set the Texture property to any of the images from the Background assets folder.

res://Assets/Art/Free/Background

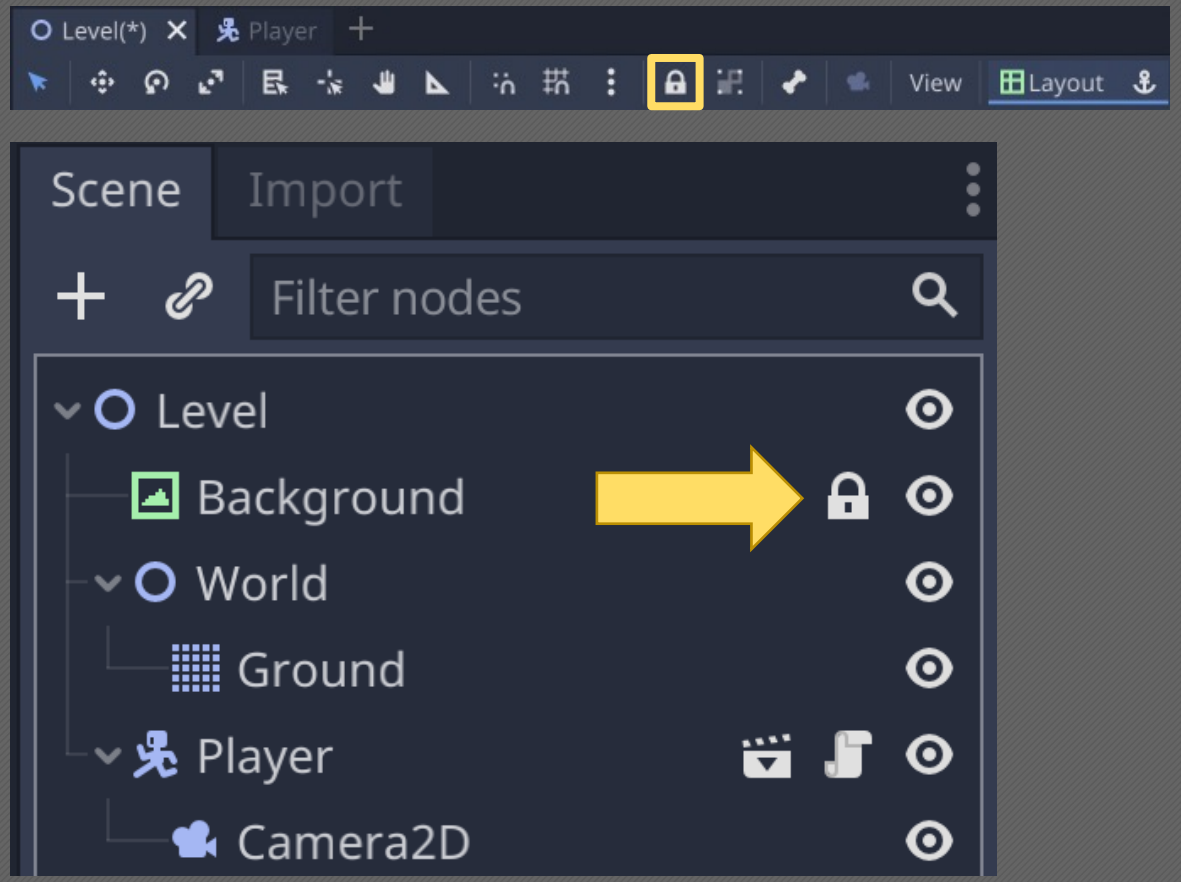
The ”Background” texture node must be the first child, because nodes are drawn into the viewport from top to bottom.



Locking Nodes

You can lock a selected node by clicking the lock icon seen to the right.

Locking the node prevents the node from being selectable in the 2D scene view.



Adding a Jump Sound Effect

In your Player Scene:

- Add an AudioStreamPlayer node named “JumpSound” as a child of the root node.
- Set the JumpSound’s Stream property to be the Jump 1.wav file (drag and drop).

  res://Assets/Audio/jdwasabi_Sounds/Jump 1.wav

- Update the Player script to call `$JumpSound.play()` where we trigger our jump in the _input(event) function.

Note: An AudioStreamPlayer2D node can be used to play 2D-spatial audio.

Lab Time (~10 Minutes)

- Add a Level background.
- Add a following camera to the Player in the Level scene.
- Add Gravity & Jump behavior to the Player script.
- Add a Jump sound to the Player.

[Optional] Add music to the Level.

Session Goal #4: Level Checkpoints

1. ✅ Controllable Player
2. ✅ Reusable Base Level
3. ✅ Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. Level Checkpoints
5. Hazards & Player Respawn
6. Collectibles & HUD
7. A Basic Enemy
8. Main Menu & Changing Scenes
9. Open Workshop, Tinker Time

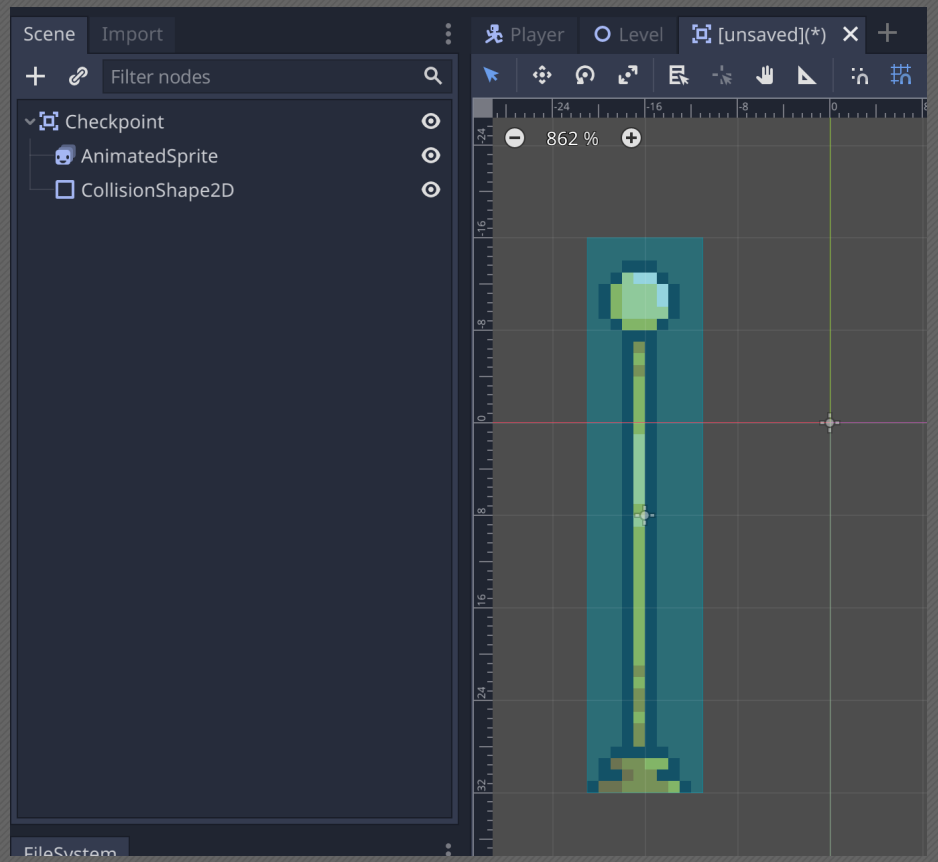
Adding Level Checkpoints

- Create a new Area2D scene and name the root node “Checkpoint”.
- Add an AnimatedSprite child node and create three new 20 FPS sprite animations (idle, activating, active) using the (No Flag, Flag Out, Flag Idle) sprite sheets from following directory:

  Assets/Art/Free/Items/Checkpoints/Checkpoint
- Position the sprite so the center of the scene sits to the middle-right of the pole.
- Save the Checkpoint scene. Also save the AnimatedSprite as a resource for safety.

Adding Level Checkpoints

- Set the AnimatedSprite's Playing property to "On" and set the Animation property to the "idle" animation.
- Set all animations to be 20 FPS and set the activating animation to not loop.
- Add a CollisionShape2D child to the root Checkpoint node and give it a rectangle shape that encompasses the shape of the flagpole.



Create a Checkpoint Script

```
extends Area2D

# optional “setget” keyword = set_func[, get_func]
export (bool) var is_active := false setget set_active

func set_active(new_value):
    is_active = new_value

    if is_active:
        $AnimatedSprite.play("activating")
    else:
        # second arg is an optional “play backwards” flag
        $AnimatedSprite.play("activating", true)
```

export – makes the value accessible for editing in the Inspector tab

Intro to Signals

"Call down, Signal up"

- Signals are messages that nodes can emit when something specific has happened (such as a button being pressed, or an area being entered).
- Other nodes or scripts can connect to and listen for specific signals, allowing those components to react when the signals are emitted.
- You can connect signals via either the Node tab (to the right of the Inspector tab) or directly through code.

Order of Events

1. The Player scene's KinematicBody2D will collide with (or enter) the Checkpoint scene's Area2D.

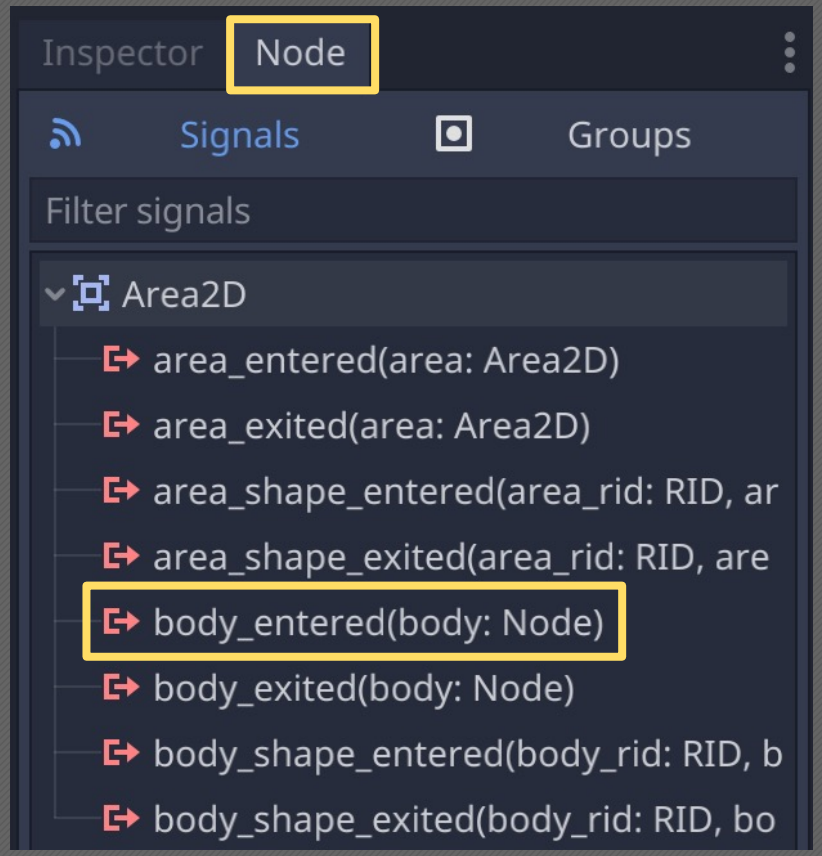
2. The Checkpoint's Area2D node (the root node type) will emit a signal that a collision has occurred.

3. The Checkpoint script will respond to the event by attempting to call the triggering colliding body's update_checkpoint function if the colliding body has that function (intended for the Player).

4. The Player will update its own checkpoint references for future use in respawn behavior via its own update_checkpoint function.

Adding Level Checkpoints

- Select your Checkpoint scene's Area2D root node in the Scene Tree.
- Switch from the Inspector tab to the Node tab.
- Connect the Area2D body_entered signal to the Checkpoint script by double-clicking that signal name.

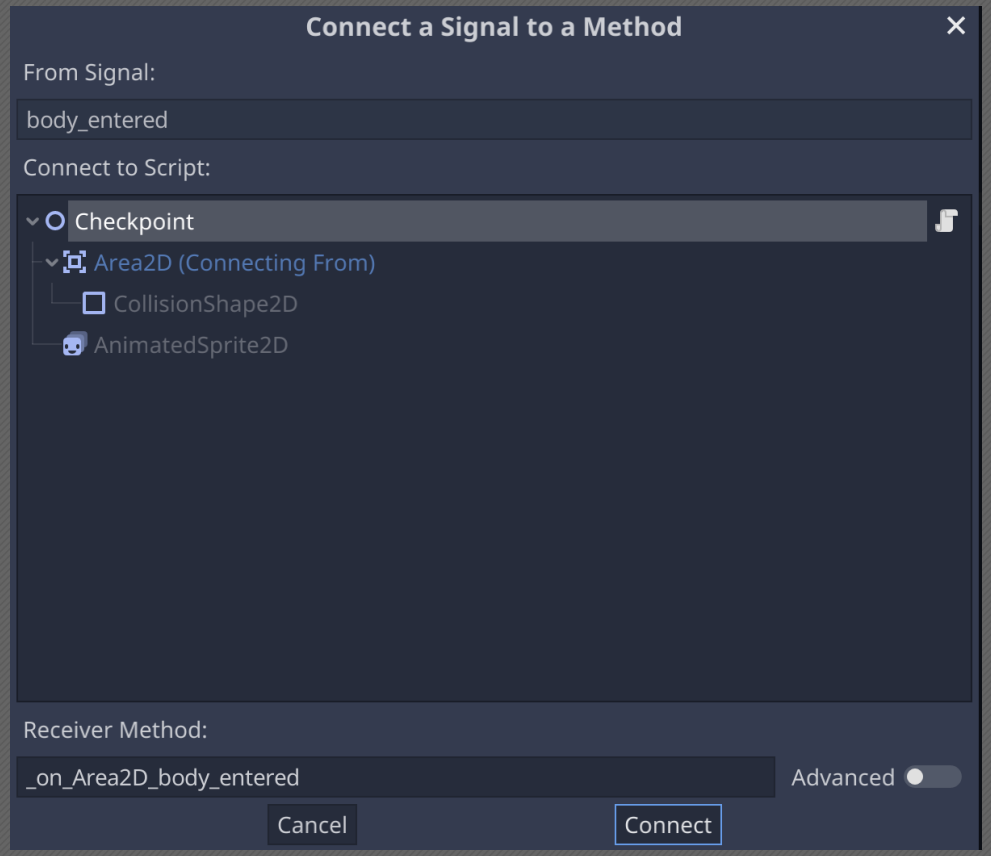


Adding Level Checkpoints

This dialog is prompting for which node/script you want to connect to listen for event emissions from the Area2D's body_entered signal.

The result of this dialog is a generated event handler callback function for the selected event.

Keep the defaults and click the connect button.



Adding Level Checkpoints

- Add this basic code to the empty generated callback function.

- Notice the status icon in the left gutter of the same line as the signal callback function. This only happens on signals connected through the Node tab.

```
21
22 func _on_Area2D_body_entered(body):
23     if body.has_method("update_checkpoint"):
24         body.update_checkpoint(self)
25
```

- The “update_checkpoint” function will be added to the Player script soon.

Adding Level Checkpoints

Do the same kind of signal wiring for the Checkpoint scene's AnimatedSprite's animation_finished signal.

```
func _on_AnimatedSprite_animation_finished():
    if is_active:
        $AnimatedSprite.play("active")
    else:
        $AnimatedSprite.play("idle")
```

Update the Player to Use Checkpoints

```
var respawn_checkpoint_node_ref : Node2D


func update_checkpoint(new_checkpoint: Node2D):
    if respawn_checkpoint_node_ref == new_checkpoint:
        return
    if respawn_checkpoint_node_ref != null:
        respawn_checkpoint_node_ref.set_active(false)

    respawn_checkpoint_node_ref = new_checkpoint
    respawn_checkpoint_node_ref.set_active(true)
```

Add Checkpoints to the Level Scene

- In your Level scene, add a new Node child named Checkpoints. Make sure this new node is *between* the World and Player scenes in the scene tree.
- With the Checkpoints node selected, instance in a Checkpoint scene.
- Put the Checkpoint somewhere other than the Player's starting point.
- Duplicate the Checkpoint (right-click the scene node and select the "Duplicate" option from the context menu) and move it to a different location in the level.
- Run the game and walk to the flags!

Lab Time & Break (~15 Minutes)

- Create the Checkpoint scene.
- Create the Checkpoint script and set up signal handlers.
- Update the Player script to have a basic checkpoint implementation.
- Instance some Checkpoint scenes into the Level scene.

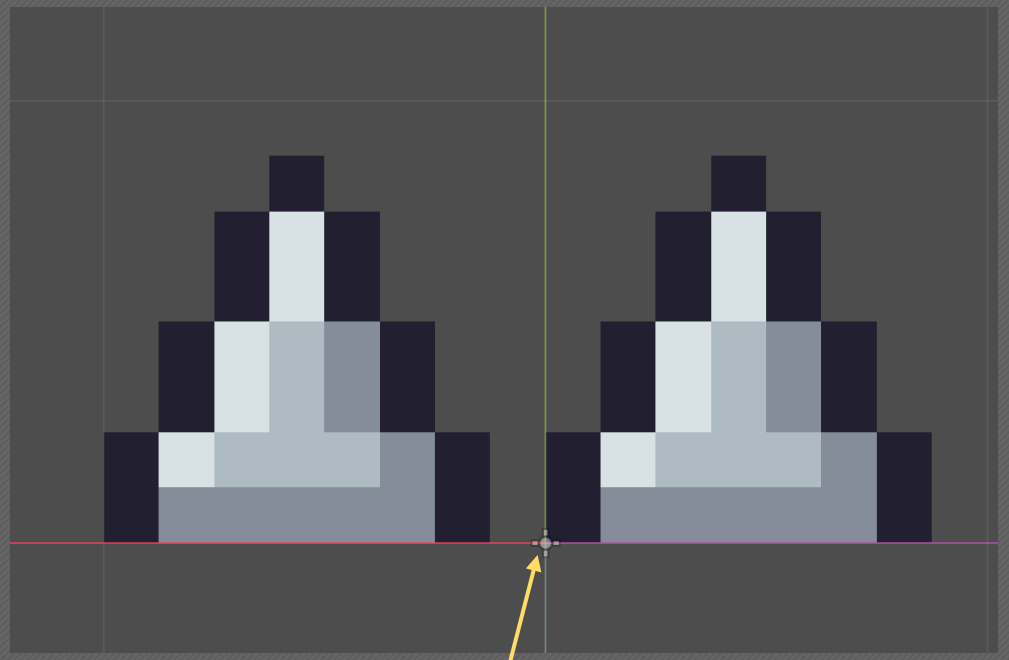
Session Goal #5: Hazards & Player Respawn

1. ✅ Controllable Player
2. ✅ Reusable Base Level
3. ✅ Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. ✅ Level Checkpoints
5. Hazards & Player Respawn
6. Collectibles & HUD
7. A Basic Enemy
8. Main Menu & Changing Scenes
9. Open Workshop, Tinker Time

Creating a Spikes Scene

- Create a new Area2D scene and rename it “Spikes”.
- Add a child Sprite node and set its Texture property to:

  Assets/Art/*Free/Traps/Spikes/Idle.png*
- Align the Sprite so the spikes appear above the center (0,0) of the scene.

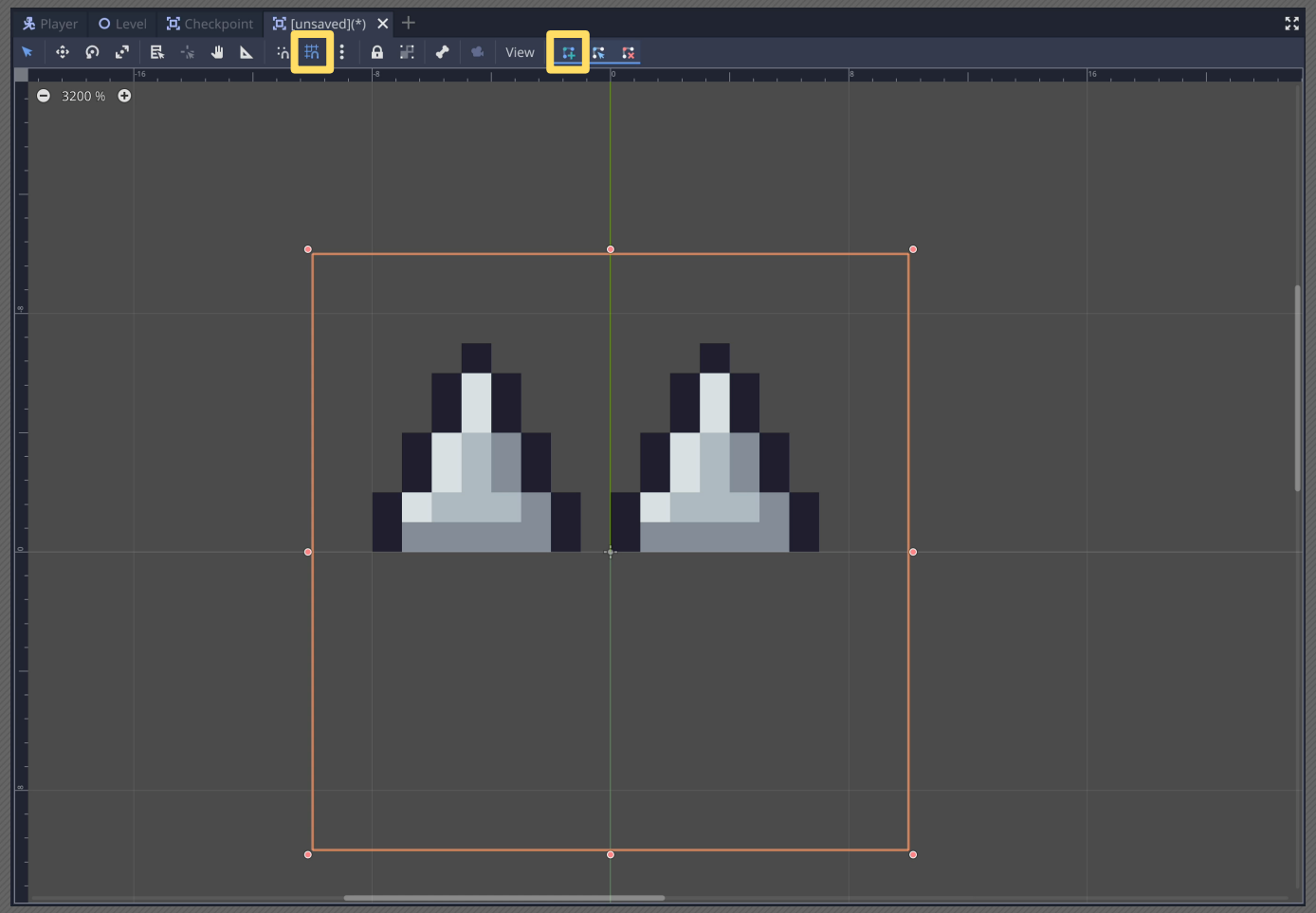


Notice the Spike scene’s center (0,0) position

Creating a Spikes Scene

- Add a CollisionPolygon2D node as a child of the Spikes root node.
- Click the “Create Points” button left-click parts of the boundary one-by-one to produce a shape.
- Close the polygon by adding a second point on the starting point.

Tip: Turn off Grid Snapping if you want a pixel-perfect shape.



Emit the Spikes “Damage” Signal

- Create a new Spikes script on the root node of the Spikes scene.
- Connect the Area2D body_entered signal to the new Spikes script using the same has_method strategy used in the Checkpoint scene.

```
extends Area2D

func _on_Spikes_body_entered(body):
    if body.has_method("take_spike_damage"):
        body.take_spike_damage()
```

Update the Player to Respond to Spikes

Add the following code to the Player script.

```
onready var START_POSITION = global_position

func take_spike_damage() -> void:
    respawn()

func respawn():
    if respawn_checkpoint_node_ref != null:
        global_position = START_POSITION
    else:
        global_position = respawn_checkpoint_node_ref.global_position
```

Update the Player to Respond to Spikes

Add the following code to the Player script.

The START_POSITION var is a default position to avoid null checkpoint refs.

```
onready var START_POSITION = global_position

func take_spike_damage() -> void:
    respawn()

func respawn():
    if respawn_checkpoint_node_ref != null:
        global_position = START_POSITION
    else:
        global_position = respawn_checkpoint_node_ref.global_position
```

Update the Player to Respond to Spikes

Add the following code to the Player script.

The take_spike_damage func is called by the Spikes to trigger a respawn.

```
onready var START_POSITION = global_position

func take_spike_damage() -> void:
    respawn()

func respawn():
    if respawn_checkpoint_node_ref != null:
        global_position = START_POSITION
    else:
        global_position = respawn_checkpoint_node_ref.global_position
```

Update the Player to Respond to Spikes

Add the following code to the Player script.

The respawn func safely moves our player back to the current “checkpoint”.

```
onready var START_POSITION = global_position

func take_spike_damage() -> void:
    respawn()
```

```
func respawn():
    if respawn_checkpoint_node_ref != null:
        global_position = START_POSITION
    else:
        global_position = respawn_checkpoint_node_ref.global_position
```

Lab Time & Break (~10 Minutes)

- Create the Spikes scene and script, with the Area2D signal callbacks.
- Create a new “Hazards” Node as an immediate sibling of the Checkpoints node.
- Instantiate some Spikes scenes into the new Hazards node.
- Verify the Damage & Respawn behavior works by jumping on Spikes.

Session Goal #6: Collectibles & HUD

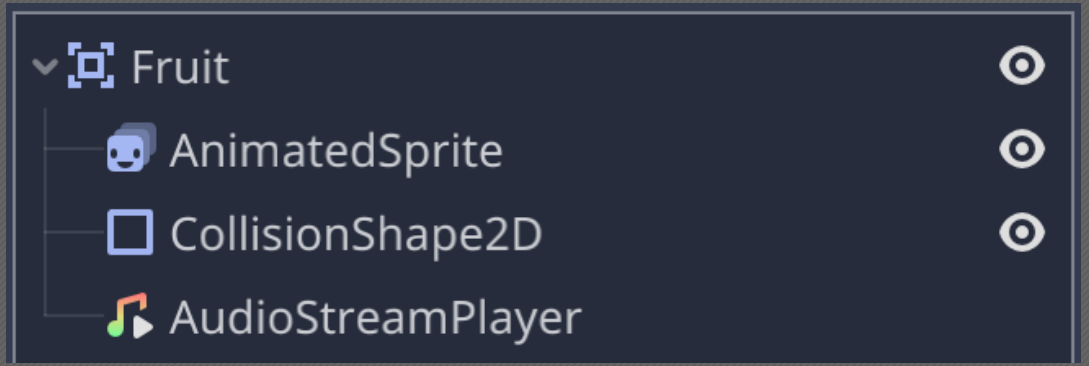
1. ✅ Controllable Player
2. ✅ Reusable Base Level
3. ✅ Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. ✅ Level Checkpoints
5. ✅ Hazards & Player Respawn
6. Collectibles & HUD
7. A Basic Enemy
8. Main Menu & Changing Scenes
9. Open Workshop, Tinker Time

Creating a Fruit Scene

- Create a new Area2D scene named “Fruit” with the same scene tree structure seen to the right.

- Create a new SpriteFrames resource for the AnimatedSprite’s Frames property.

- Create two new animations using any two of the sprite sheets at the following path:
  Assets/Art/Free/Items/Fruits/

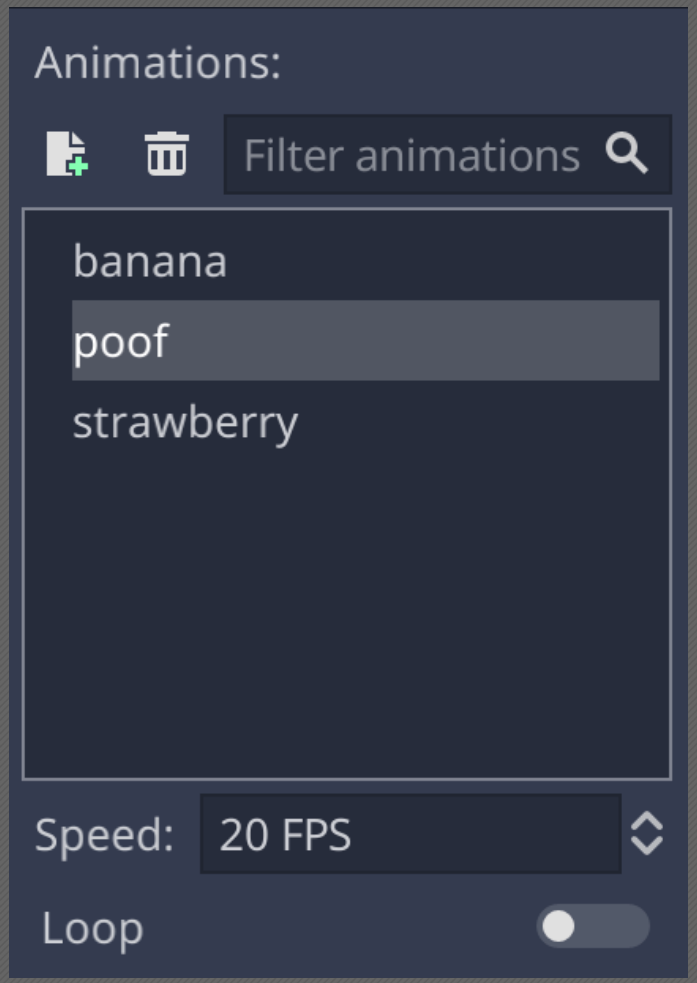


Creating some animated Fruits

- Create two new appropriately-named animations using any two of the sprite sheets at the following path:
  Assets/Art/Free/Items/Fruits/

- Create a third animation called "poof" on the same SpriteFrames resource using the Collected.png file from the same directory as above.

- Make sure all animations are 20 FPS, and that "poof" is the only animation with Loop turned off.

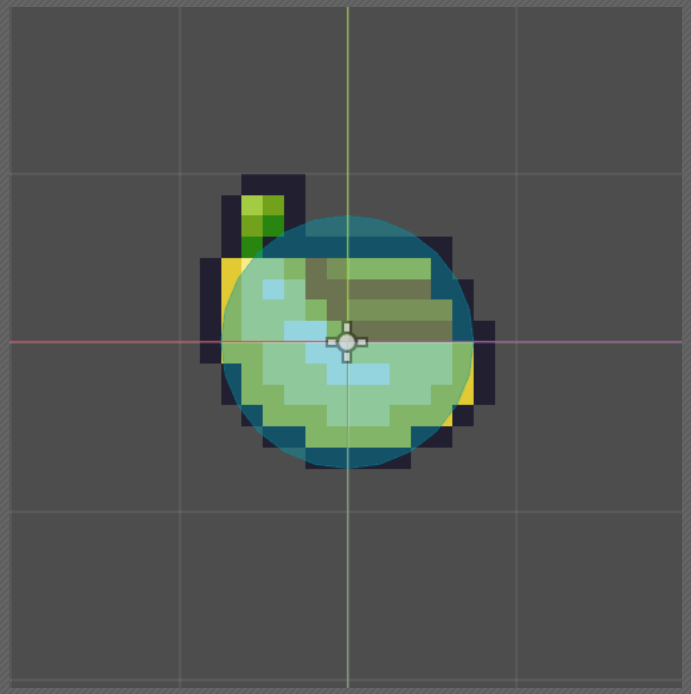


Fruit Collision and Collection Sound

- Apply a Circle shape to the CollisionShape2D and set the size of the circle to encompass the sprite.

- Update the AudioStreamPlayer Stream property to be the “Fruit collect 1” wav sound from the sounds folder:

  Assets/Audio/jdwasabi_Sounds/



Create a Fruit Script and Connect Signals

```
extends Area2D

export (String, "Banana", "Strawberry") var fruit = "Banana" setget set_fruit
onready var ANIMATED_SPRITE := $AnimatedSprite

func set_fruit(new_value:String):
    fruit = new_value
    $AnimatedSprite.play(fruit.to_lower())

func _on_Fruit_body_entered(body):
    if visible and body.has_method("collect_fruit"):
        body.collect_fruit(fruit.to_lower())
        ANIMATED_SPRITE.play("poof")
        $AudioStreamPlayer.play()

func _on_AnimatedSprite_animation_finished():
    if $AnimatedSprite.animation == "poof": visible = false
```

Create a Fruit Script and Connect Signals

The ***export*** keyword makes the variable available for editing in instanced child nodes in the Inspector window.

```
extends Area2D

export (String, "Banana", "Strawberry") var fruit = "Banana" setget set_fruit
onready var ANIMATED_SPRITE := $AnimatedSprite

func set_fruit(new_value:String):
    fruit = new_value
    $AnimatedSprite.play(fruit.to_lower())

func _on_Fruit_body_entered(body):
    if visible and body.has_method("collect_fruit"):
        body.collect_fruit(fruit.to_lower())
        ANIMATED_SPRITE.play("poof")
        $AudioStreamPlayer.play()

func _on_AnimatedSprite_animation_finished():
    if $AnimatedSprite.animation == "poof": visible = false
```

Create a Fruit Script and Connect Signals

Creating an *onready* variable that references a named node in the Scene Tree ($ syntax) makes maintenance easier.

```
extends Area2D

export (String, "Banana", "Strawberry") var fruit = "Banana" setget set_fruit
onready var ANIMATED_SPRITE := $AnimatedSprite

func set_fruit(new_value:String):
    fruit = new_value
    $AnimatedSprite.play(fruit.to_lower())

func _on_Fruit_body_entered(body):
    if visible and body.has_method("collect_fruit"):
        body.collect_fruit(fruit.to_lower())
        ANIMATED_SPRITE.play("poof")
        $AudioStreamPlayer.play()

func _on_AnimatedSprite_animation_finished():
    if $AnimatedSprite.animation == "poof": visible = false
```

Create a Fruit Script and Connect Signals

Update the internal value of the fruit variable and run the animation the same name when the setter is called.

```
extends Area2D

export (String, "Banana", "Strawberry") var fruit = "Banana" setget set_fruit
onready var ANIMATED_SPRITE := $AnimatedSprite

func set_fruit(new_value:String):
    fruit = new_value
    $AnimatedSprite.play(fruit.to_lower())

func _on_Fruit_body_entered(body):
    if visible and body.has_method("collect_fruit"):
        body.collect_fruit(fruit.to_lower())
        ANIMATED_SPRITE.play("poof")
        $AudioStreamPlayer.play()

func _on_AnimatedSprite_animation_finished():
    if $AnimatedSprite.animation == "poof": visible = false
```

Create a Fruit Script and Connect Signals

Play an animation, sound, and call a method against a valid triggering/colliding body against the Area2D collision body.

```
extends Area2D

export (String, "Banana", "Strawberry") var fruit = "Banana" setget set_fruit
onready var ANIMATED_SPRITE := $AnimatedSprite

func set_fruit(new_value:String):
    fruit = new_value
    $AnimatedSprite.play(fruit.to_lower())

func _on_Fruit_body_entered(body):
    if visible and body.has_method("collect_fruit"):
        body.collect_fruit(fruit.to_lower())
        ANIMATED_SPRITE.play("poof")
        $AudioStreamPlayer.play()

func _on_AnimatedSprite_animation_finished():
    if $AnimatedSprite.animation == "poof": visible = false
```

Create a Fruit Script and Connect Signals

When the AnimatedSprite finishes playing the “poof” animation, make this Fruit scene invisible.

```
extends Area2D

export (String, "Banana", "Strawberry") var fruit = "Banana" setget set_fruit
onready var ANIMATED_SPRITE := $AnimatedSprite

func set_fruit(new_value:String):
    fruit = new_value
    $AnimatedSprite.play(fruit.to_lower())

func _on_Fruit_body_entered(body):
    if visible and body.has_method("collect_fruit"):
        body.collect_fruit(fruit.to_lower())
        ANIMATED_SPRITE.play("poof")
        $AudioStreamPlayer.play()

func _on_AnimatedSprite_animation_finished():
    if $AnimatedSprite.animation == "poof": visible = false
```

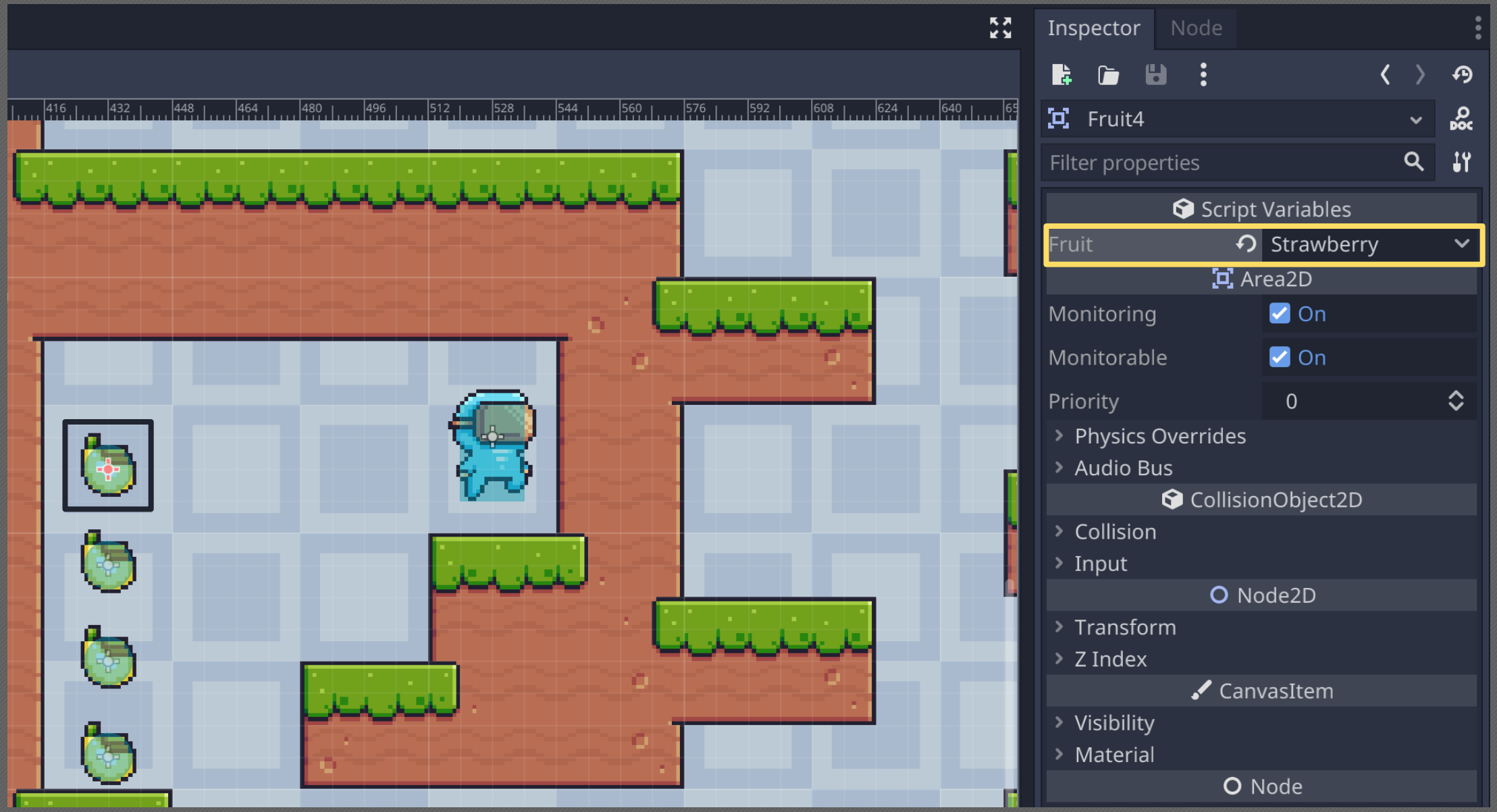
Create a Fruit Script and Connect Signals

Note the use of $AnimatedSprite instead of ANIMATED_SPRITE (to avoid impending Nil reference errors).

```
extends Area2D

export (String, "Banana", "Strawberry") var fruit = "Banana" setget set_fruit
onready var ANIMATED_SPRITE := $AnimatedSprite

func set_fruit(new_value:String):
    fruit = new_value
    $AnimatedSprite.play(fruit.to_lower())

func _on_Fruit_body_entered(body):
    if visible and body.has_method("collect_fruit"):
        body.collect_fruit(fruit.to_lower())
        ANIMATED_SPRITE.play("poof")
        $AudioStreamPlayer.play()

func _on_AnimatedSprite_animation_finished():
    if $AnimatedSprite.animation == "poof": visible = false
```

The ***export*** keyword makes the variable available for editing in instanced child nodes in the Inspector window.

However, changing the value in the Inspector does not update the animation of the Fruit scene(s) in the editor.

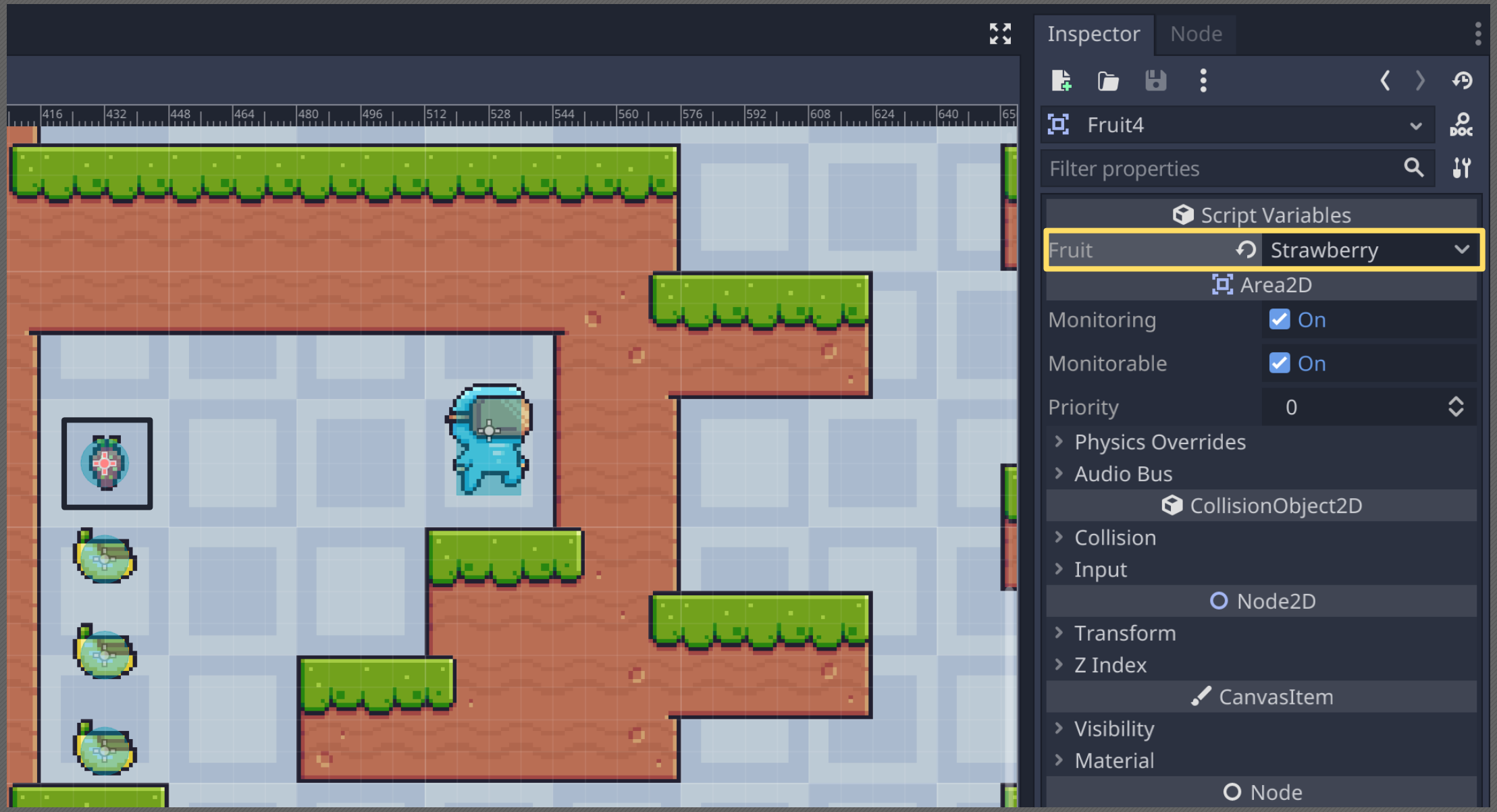


Update the Fruit script to be a “tool”

Add the tool keyword to the very beginning of a Script to have the script/scene run in the editor the same way it would run in-game.

```
tool extends Area2D

export (String, "Banana", "Strawberry") var fruit = "Banana" setget set_fruit
onready var ANIMATED_SPRITE := $AnimatedSprite

func set_fruit(new_value:String):
    fruit = new_value
    $AnimatedSprite.play(fruit.to_lower())

# ...

func _on_AnimatedSprite_animation_finished():
    if $AnimatedSprite.animation == "poof":
        visible = false
```

With the tool keyword on our Fruit script and the new setter function, we can see the Fruit animation update in the editor whenever we change the value of the Fruit variable. This will also work if you change the fruit value in-game!



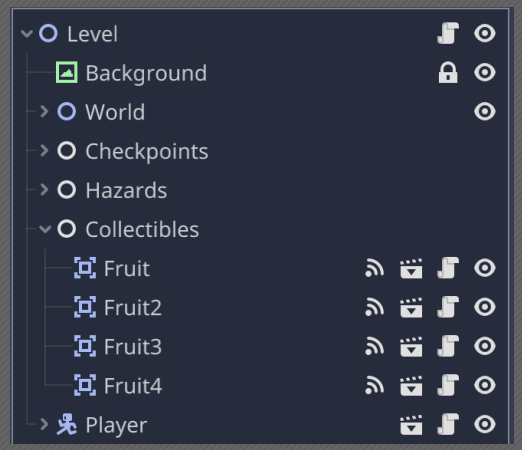
Update the Player to Handle Fruit Collection

- The collected_fruit variable is a Dictionary type, which very much resembles and behaves like a Map or JSON object.

```
var collected_fruit := {}

func collect_fruit(fruit:String) -> void:
    if not collected_fruit.has(fruit):
        collected_fruit[fruit] = 0
    collected_fruit[fruit] += 1
```

Lab Time (~15 Minutes)

- Create the Fruit scene with at least two different types of fruit available.
- Create a new Node named “Collectibles” in your Level scene tree as a sibling of the Hazards node.
- Instance in some Fruit nodes as children of the Collectibles node, and scatter them around.



Creating a HUD/UI using a CanvasLayer

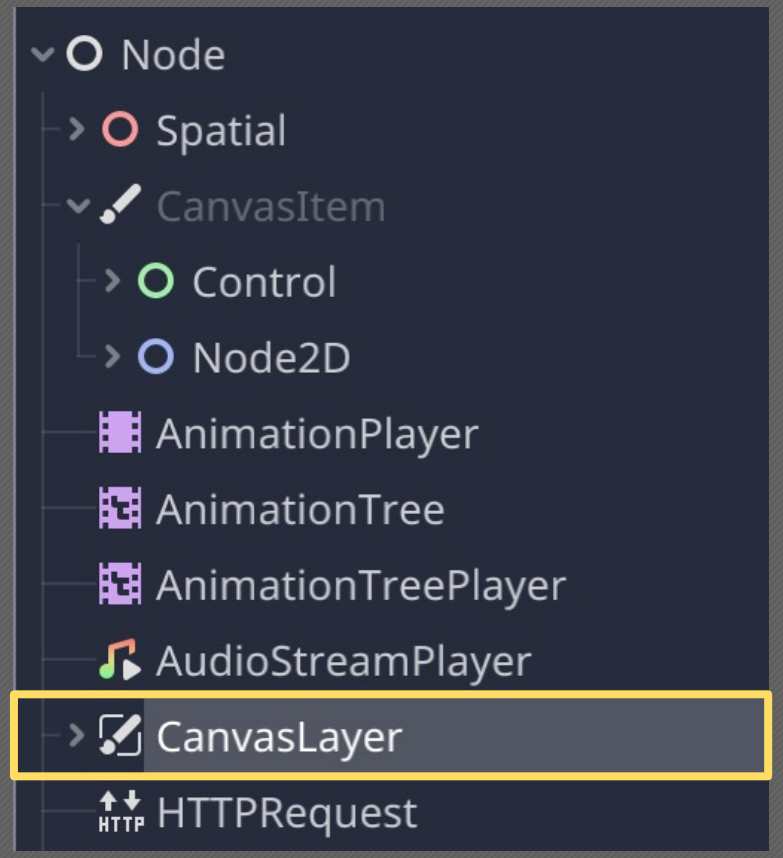
A HUD (heads-up display) is a UI tool used for providing useful game information to the player.

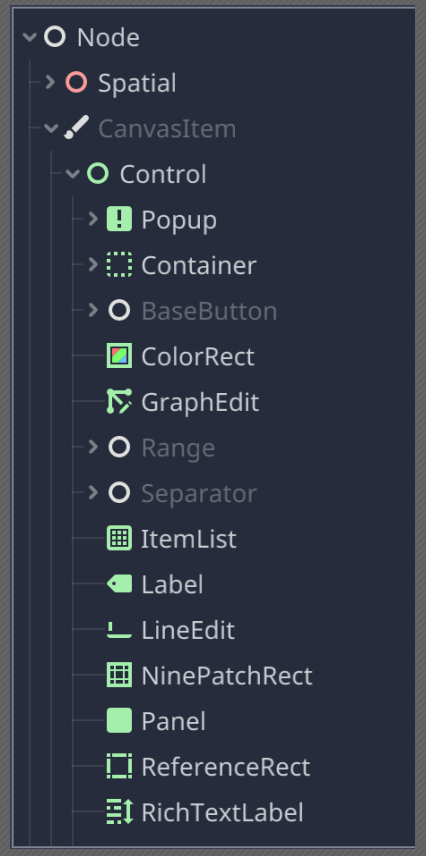
Most HUD elements are overlays that are not intended to be directly interacted with.

Each Godot CanvasLayer is a separate 2D rendering layer. We can use CanvasLayers to draw these *static* UI elements on the screen.

Godot CanvasLayer & Control Nodes

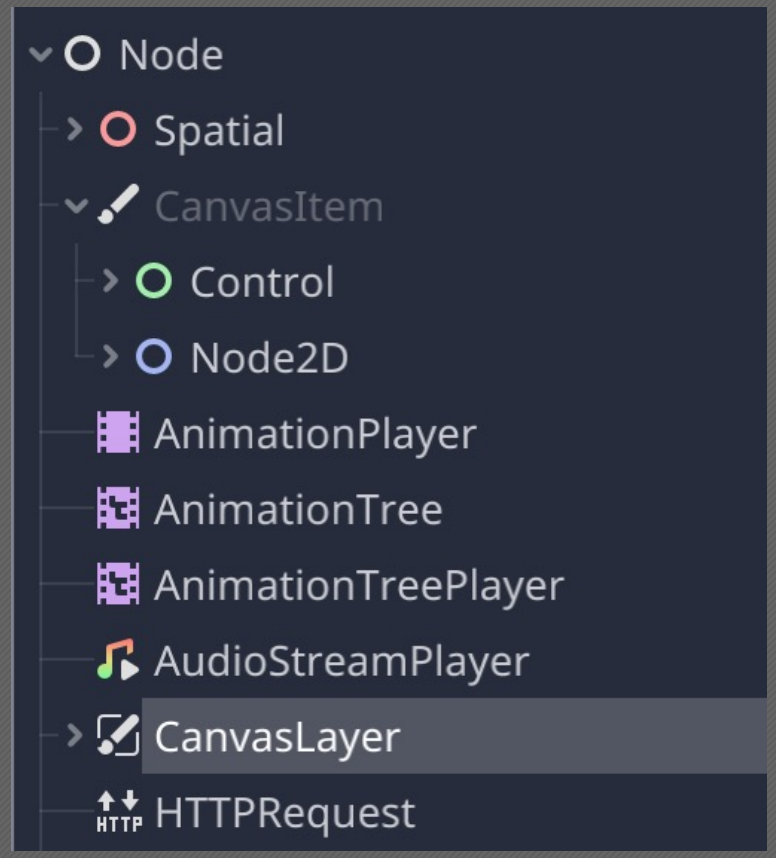
CanvasLayer nodes are UI/drawing layers that are independent of scene tree order. These nodes are drawn depending only on their layer number, so they can be instantiated whenever/however needed.

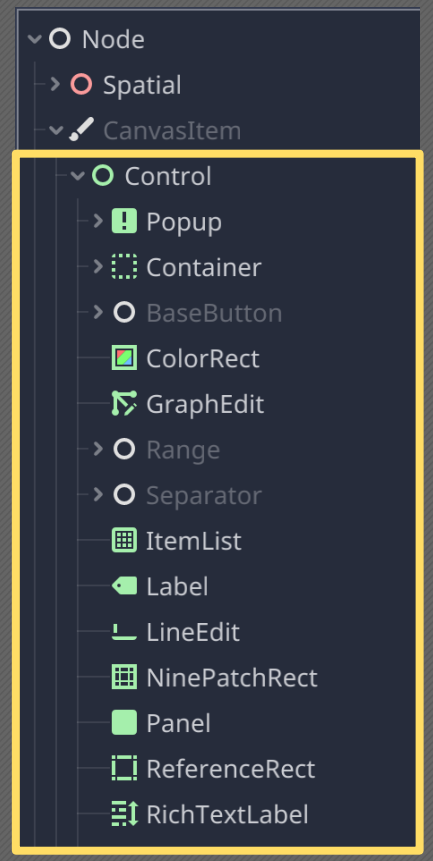




Godot CanvasLayer & Control Nodes

The Control node is the base class/node for all UI-related nodes. It has features allowing it to be positioned relative to its parent control or the current viewport (including screen size changes).

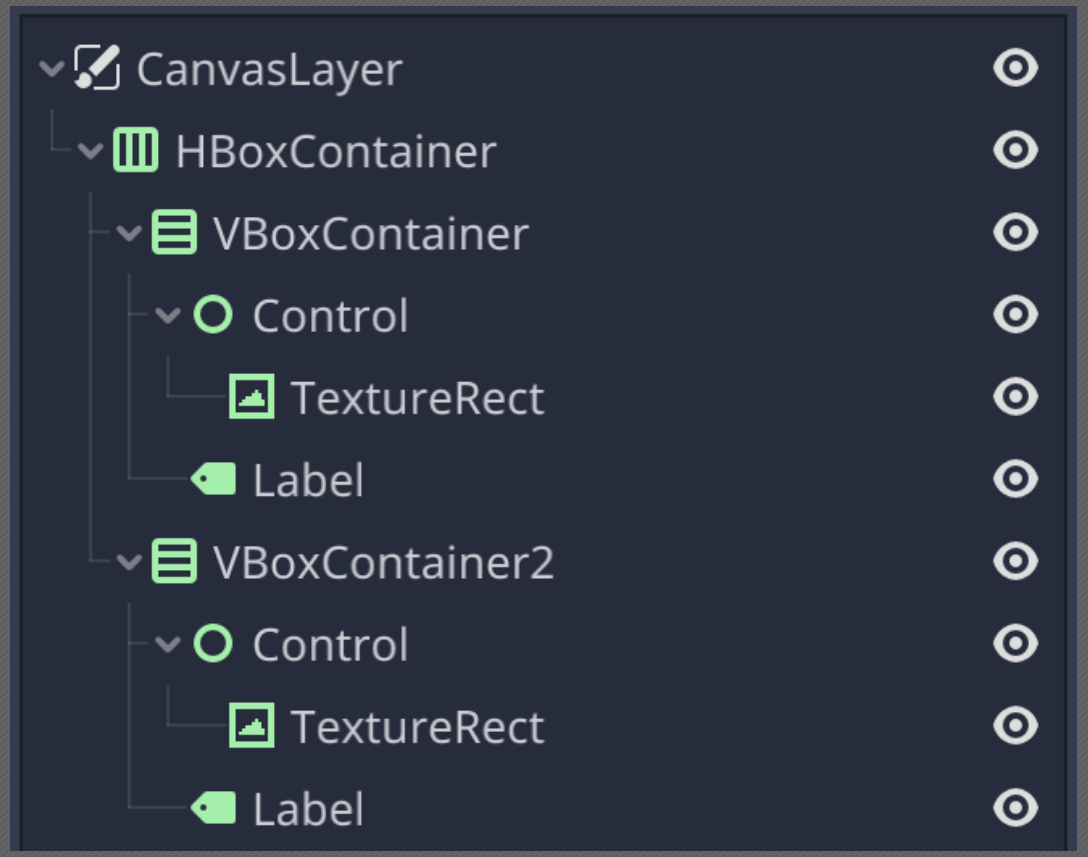




Creating a HUD

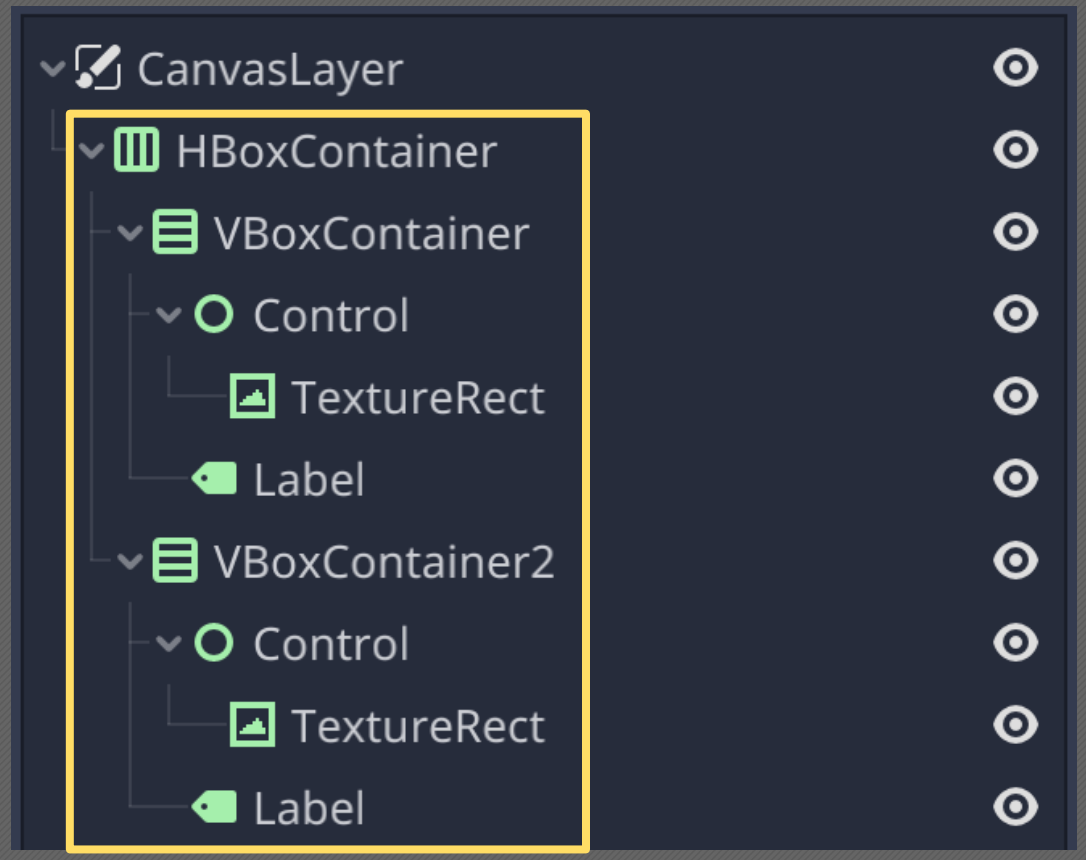
- Create a new scene using a CanvasLayer as the root node.
- Add a bunch of new nodes mimicking the scene tree structure seen to the right.
- Rename the CanvasLayer root node to “HUD” and save it with the same name.
- Update the Transform > Scale property to “2” for x and y (to make the HUD twice as large).

Intended HUD Structure



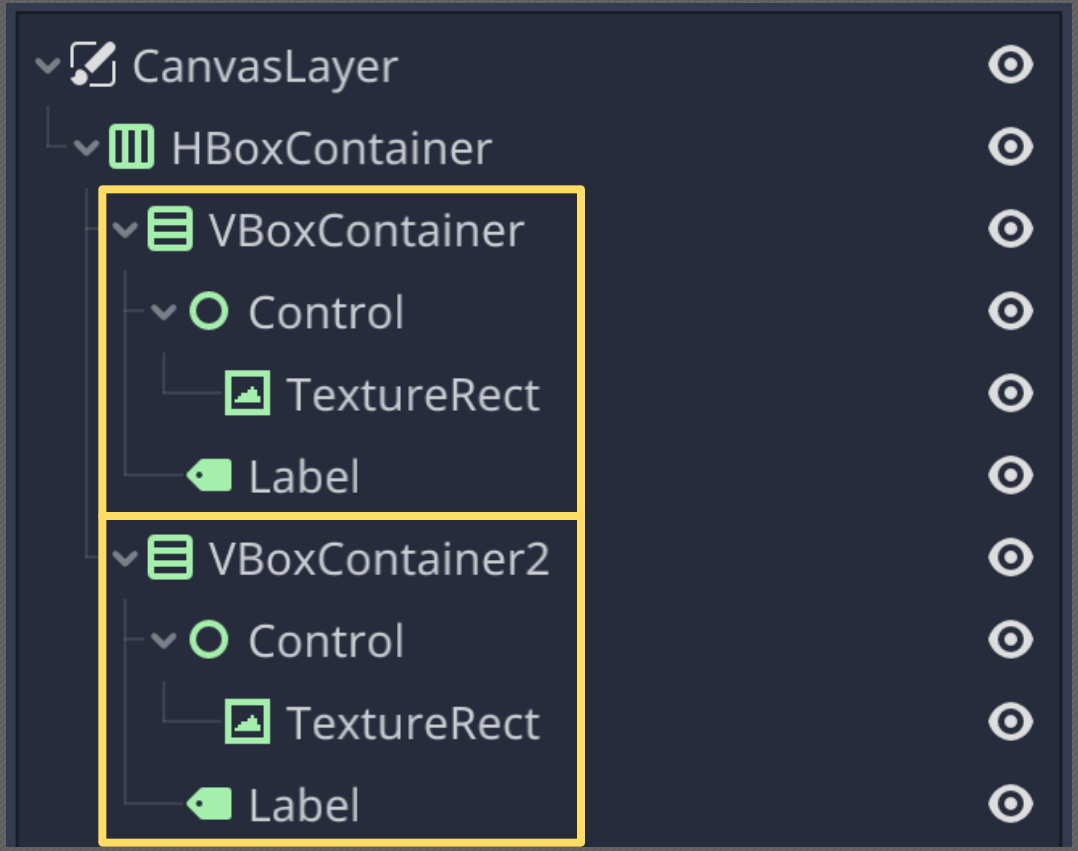
A Quick Look at the Control Nodes

- An HBoxContainer (Horizontal) is a container that allows us to simply align items horizontally.
- A VBoxContainer (Vertical) is like an HBoxContainer, except it behaves vertically.
- A Control lets us group related components.
- A TextureRect is a rectangle that can have a texture applied to it.
- A Label displays plain text.



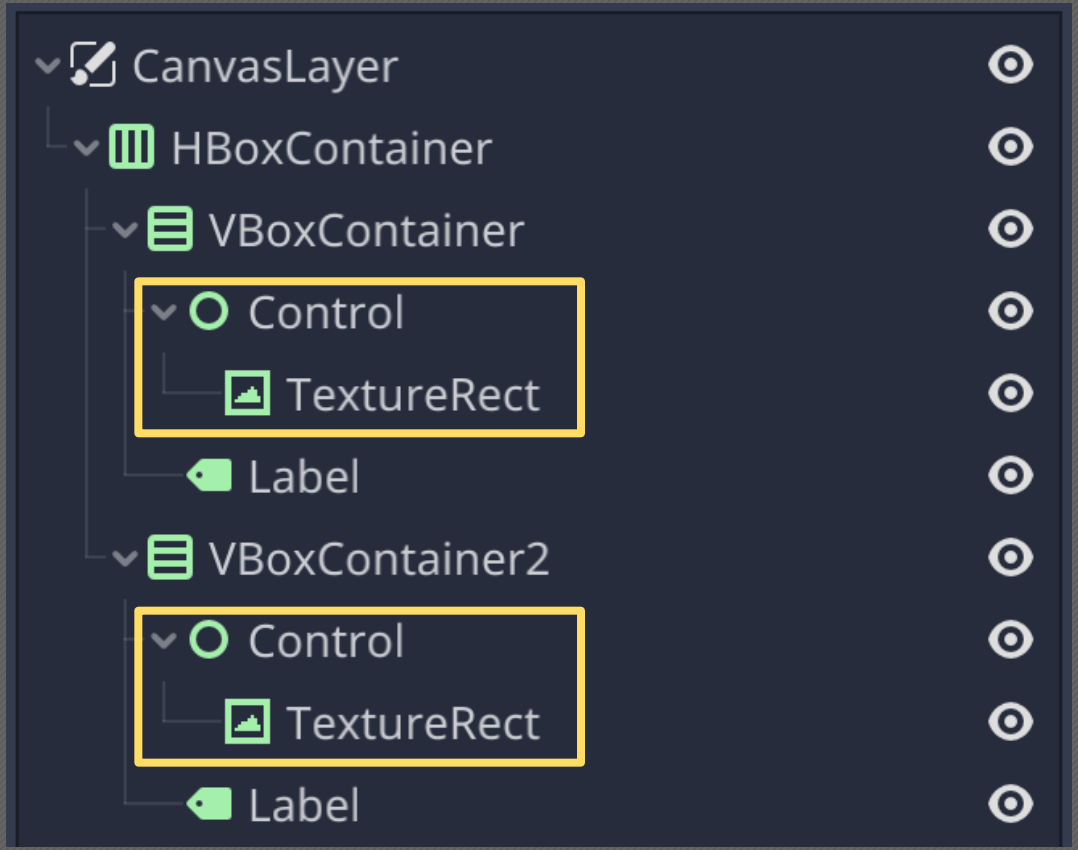
A Quick Look at the Control Nodes

- An HBoxContainer (Horizontal) is a container that allows us to simply align items horizontally.
- A VBoxContainer (Vertical) is like an HBoxContainer, except it behaves vertically.
- A Control lets us group related components.
- A TextureRect is a rectangle that can have a texture applied to it.
- A Label displays plain text.



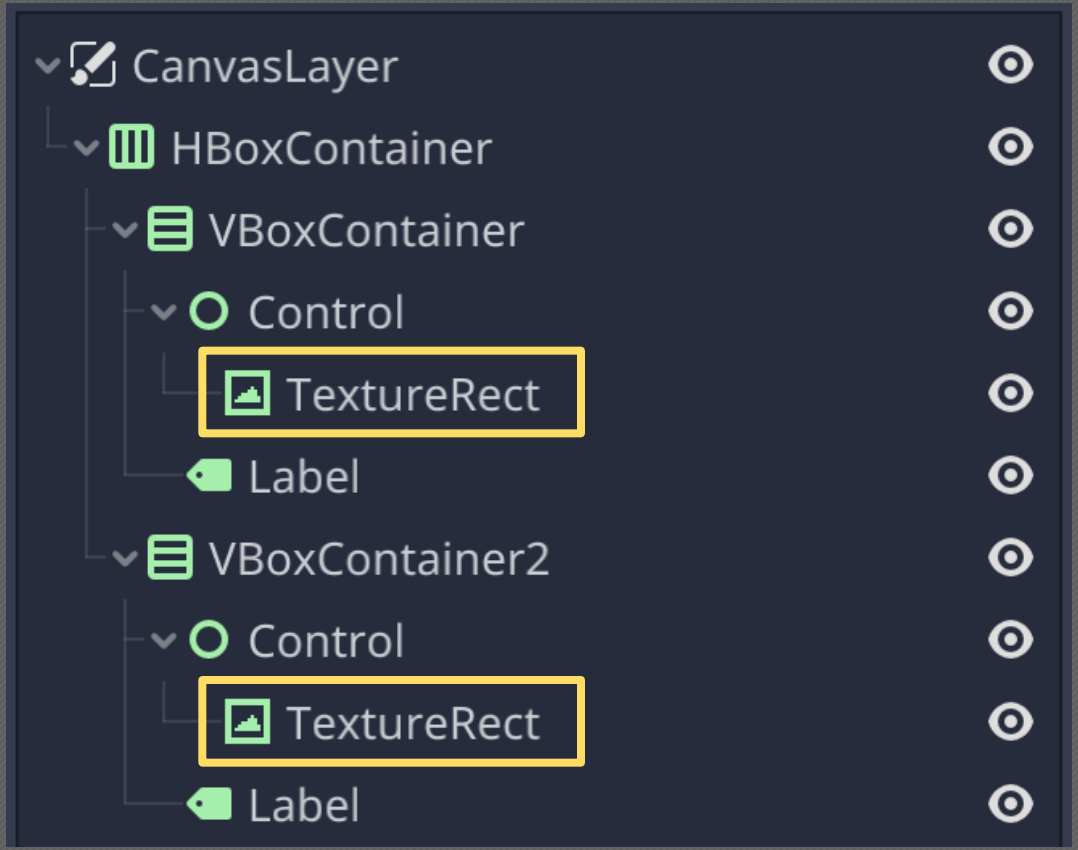
A Quick Look at the Control Nodes

- An HBoxContainer (Horizontal) is a container that allows us to simply align items horizontally.
- A VBoxContainer (Vertical) is like an HBoxContainer, except it behaves vertically.
- A Control lets us group related components.
- A TextureRect is a rectangle that can have a texture applied to it.
- A Label displays plain text.



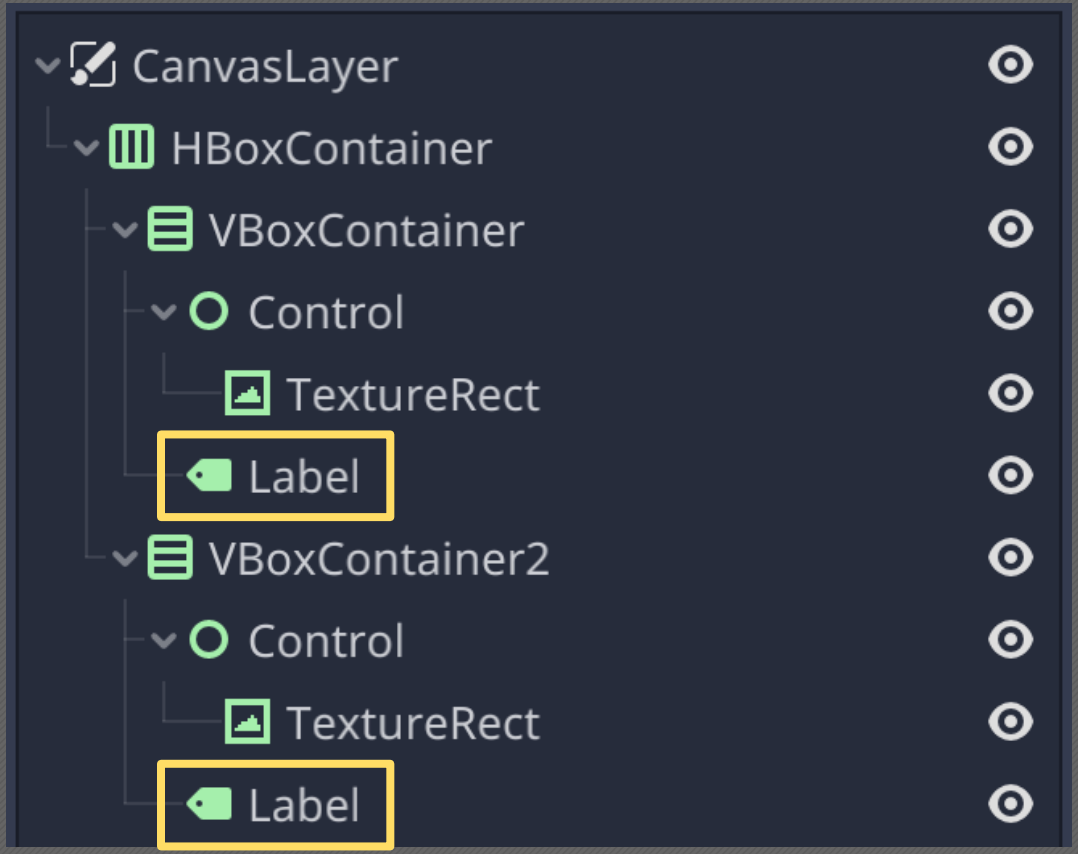
A Quick Look at the Control Nodes

- An HBoxContainer (Horizontal) is a container that allows us to simply align items horizontally.
- A VBoxContainer (Vertical) is like an HBoxContainer, except it behaves vertically.
- A Control lets us group related components.
- A TextureRect is a rectangle that can have a texture applied to it.
- A Label displays plain text.



A Quick Look at the Control Nodes

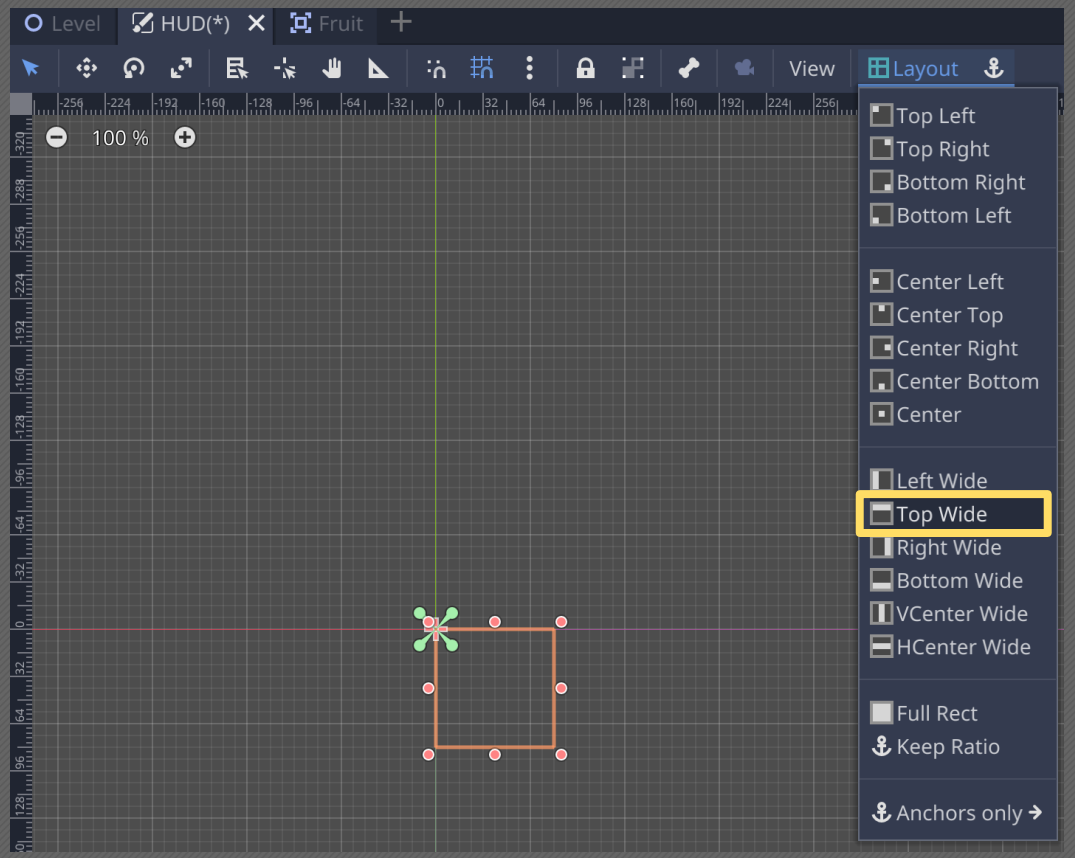
- An HBoxContainer (Horizontal) is a container that allows us to simply align items horizontally.
- A VBoxContainer (Vertical) is like an HBoxContainer, except it behaves vertically.
- A Control lets us group related components.
- A TextureRect is a rectangle that can have a texture applied to it.
- A Label displays plain text.



Creating a HUD: HBoxContainer

With the CanvasLayer “HUD” node selected in your scene tree:

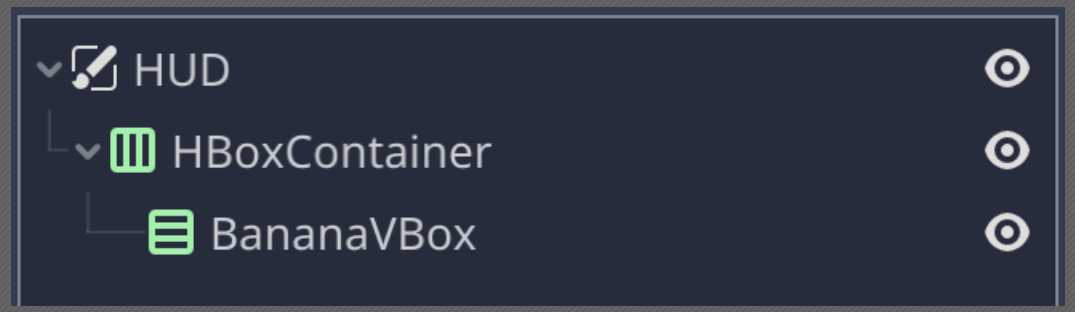
- Set the Layout to Top Wide, which will cause the HUD to stretch across the the width of the screen/viewport.
- In the Inspector, update the Margin > Left to be 8 and the Rect > Size > y to be 48.
- Set the Theme Overrides > Constants > Separation to 16 for gap spacing.



Creating a HUD: VBoxContainer

With the first VBoxContainer selected in the HUD scene tree:

- Update its name to be {Fruit}VBox for one of your chosen fruits (e.g., BananaVBox if you chose Banana as one of your collectibles).



Creating a HUD: Control

With the Control of the first FruitVBox selected in the HUD scene tree:

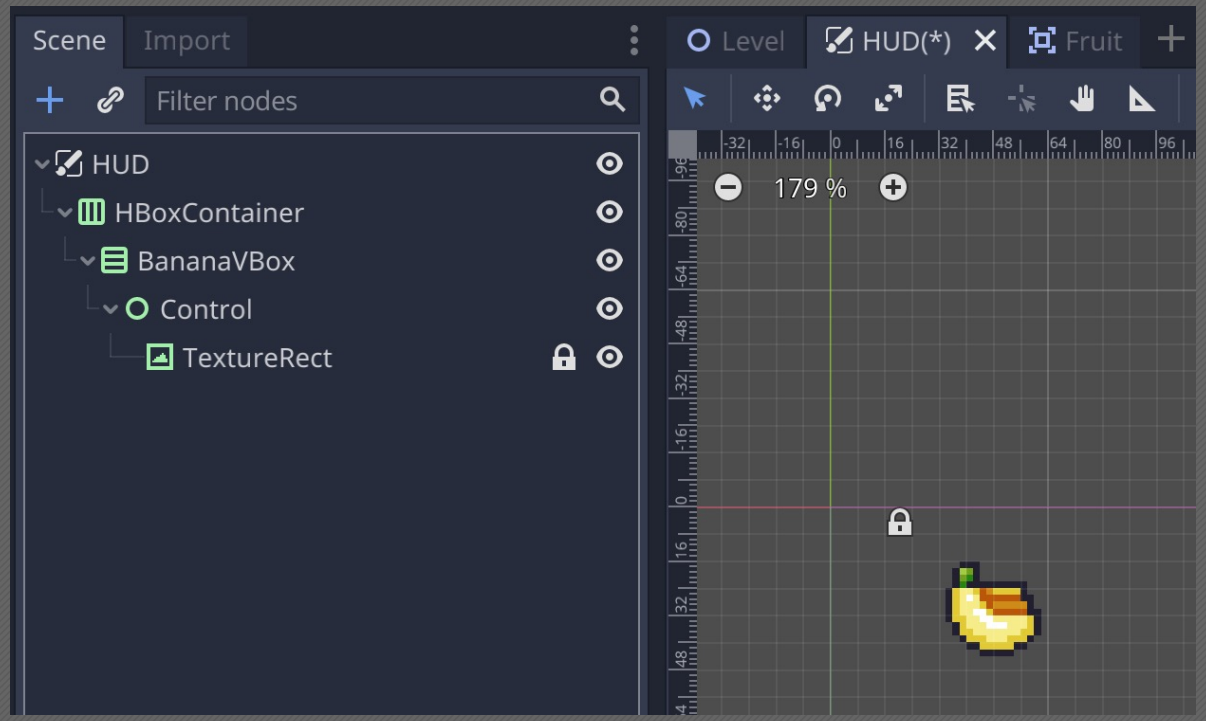
- Set its Rect > Min Size to be 32 for the x and y properties.
- Set the Rect > Clip Content to On; this will prevent overflow content from displaying.

Creating a HUD: TextureRect

With the TextureRect of the Control selected in the HUD scene tree:

- Set its Texture to be the same *Fruit*.png file used for the AnimatedSprite of the fruit being represented.

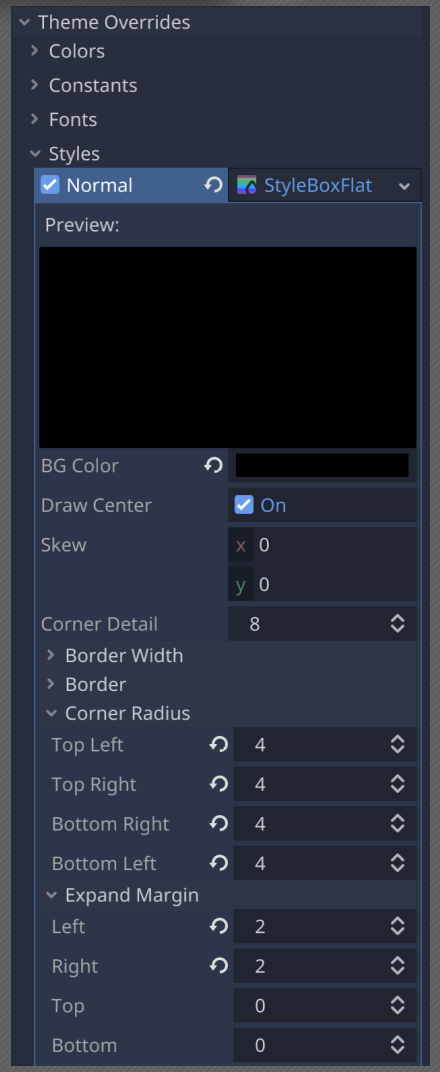
- Since it's such a wide image, lock the TextureRect in the editor window out of caution.



Creating a HUD: Label

With the Label of the first FruitVBox selected in the HUD scene tree:

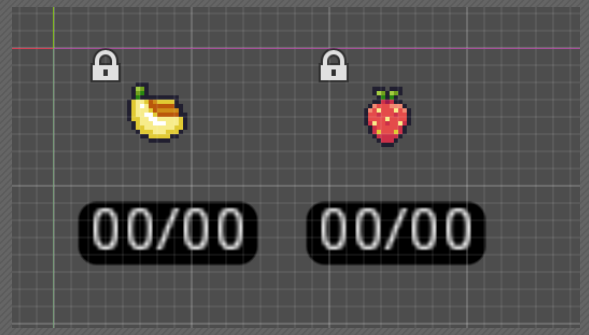
- Set its Text to “00/00” for some simple testing.
- Set Align and V Align to “Center”.
- Under the Theme Overrides section, create a new Normal StyleBoxFlat value and give it some basic styling as seen to the far right.

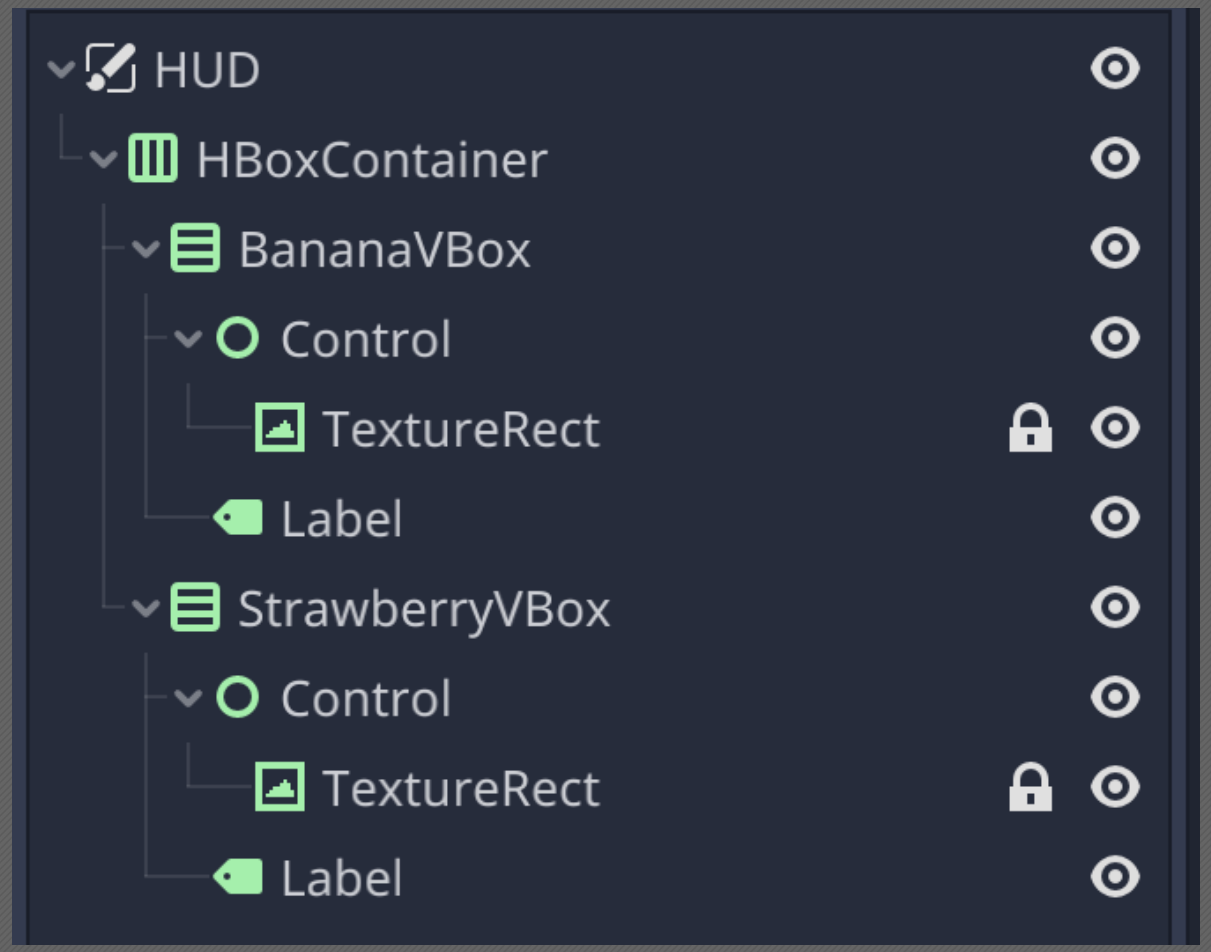


Creating a HUD: Repeat for Fruit #2

Rinse-and-repeat the same process for your second fruit.

Note: You should have two VBoxContainer nodes and only one HBoxContainer.



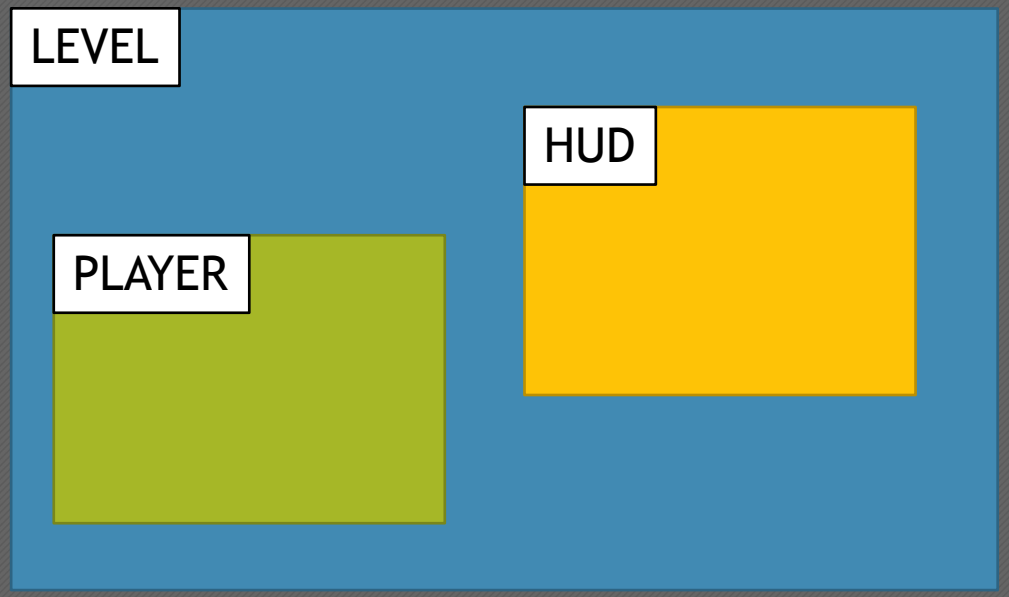


Lab Time (~15 Minutes)

- Create the HUD scene.
- Instance in the HUD scene as a child node in the Level scene.

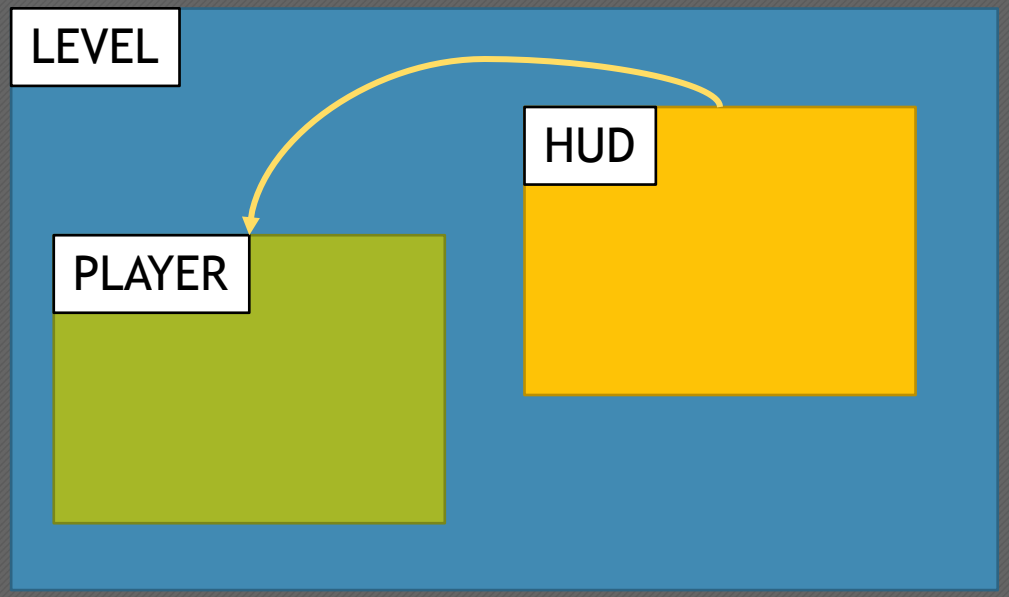
Creating a Signal Chain from Player to HUD

The Player & HUD exist within the Level, but all have inter-communication needs. We need to cater to those needs without breaking the appropriate abstractions.



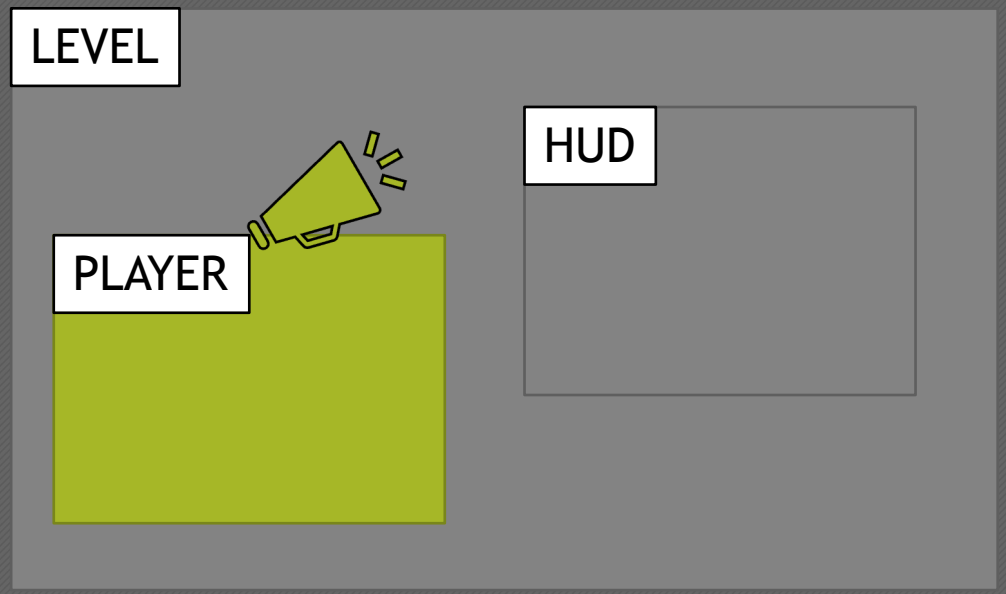
Creating a Signal Chain from Player to HUD

1) When the Level finishes loading, the Level tells its HUD child to listen for specific signals emitted by its Player child.



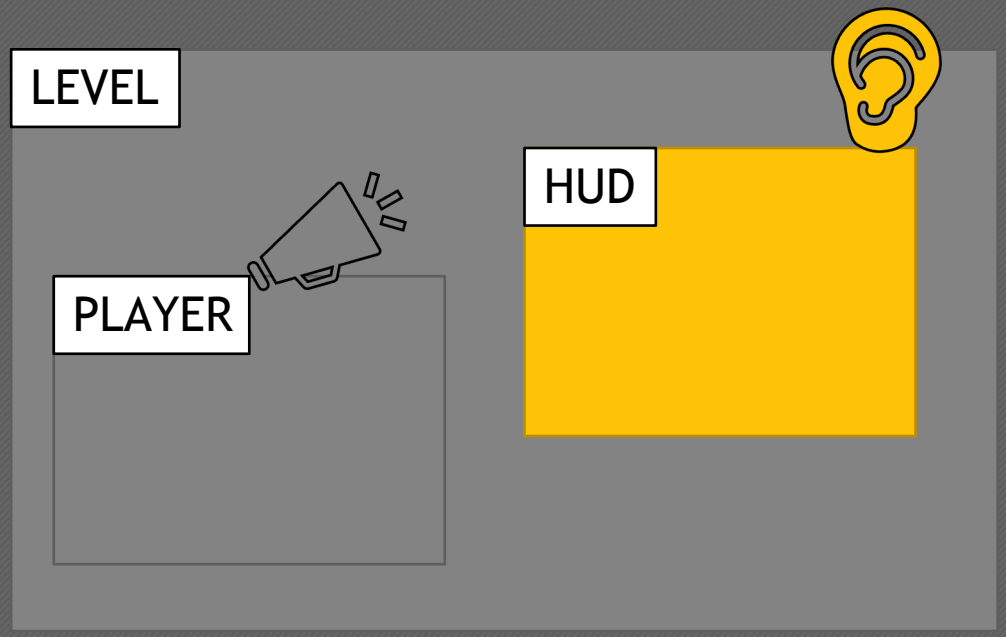
Creating a Signal Chain from Player to HUD

2) When the Player collects a Fruit, the Player will emit a specific signal indicating that event has occurred.



Creating a Signal Chain from Player to HUD

3) The HUD reacts to the observed signal by updating its fruit labels with the info provided with the signal.



Creating a HUD: Emit a Signal from Player

Since the Player is trigging the event, the Player script needs to be updated to emit an appropriate Signal.

```
extends KinematicBody2D

signal fruit_collected

# ...

func collect_fruit(fruit:String) -> void:
    if not collected_fruit.has(fruit):
        collected_fruit[fruit] = 0
    collected_fruit[fruit] += 1
    emit_signal("fruit_collected", collected_fruit)
```

Creating a HUD: Emit a Signal from Player

This line defines a new “fruit_collected” signal that can be emitted from/by the Player.

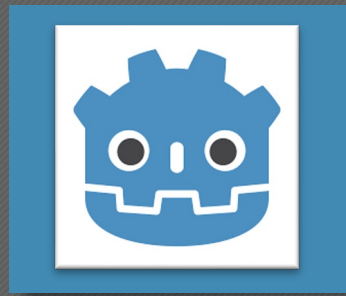
```
extends KinematicBody2D

signal fruit_collected

# ...

func collect_fruit(fruit:String) -> void:
    if not collected_fruit.has(fruit):
        collected_fruit[fruit] = 0
    collected_fruit[fruit] += 1
    emit_signal("fruit_collected", collected_fruit)
```

Creating a HUD: Emit a Signal from Player

Update the Player's *collect_fruit* function to emit the new signal, providing the collected_fruit dictionary variable as extra data for listeners/observers to process.

```
extends KinematicBody2D

signal fruit_collected

# ...

func collect_fruit(fruit:String) -> void:
    if not collected_fruit.has(fruit):
        collected_fruit[fruit] = 0
    collected_fruit[fruit] += 1
    emit_signal("fruit_collected", collected_fruit)
```

Creating a HUD: Add HUD Update Behavior



We need a way to tell the HUD to update the Fruit count labels, so we need to create a HUD script.

Tip: You can drag-and-drop a node from the scene tree into a Script to use its relative path.

```
extends CanvasLayer

onready var bananaLabel = $HBoxContainer/BananaVBox/Label
onready var strawberryLabel = $HBoxContainer/StrawberryVBox/Label

onready var fruit_totals := {
    "banana": { "current": 0, "total": 0, "labelNode": bananaLabel },
    "strawberry": { "current": 0, "total": 0, "labelNode": strawberryLabel }
}
```

Creating a HUD: Add HUD Update Behavior

We first need a reference to the label node in the Scene tree for each fruit.

Tip: You can drag-and-drop a node from the scene tree into a Script for easy access to the node's relative path.

```
extends CanvasLayer

onready var bananaLabel = $HBoxContainer/BananaVBox/Label
onready var strawberryLabel = $HBoxContainer/StrawberryVBox/Label

onready var fruit_totals := {
    "banana": { "current": 0, "total": 0, "labelNode": bananaLabel },
    "strawberry": { "current": 0, "total": 0, "labelNode": strawberryLabel }
}
```

Creating a HUD: Add HUD Update Behavior

We then initialize our fruit data with a current and total value.

We also store each fruit's node reference for easy updating (to avoid dynamic node lookups in the scene tree).

```
extends CanvasLayer

onready var bananaLabel = $HBoxContainer/BananaVBox/Label
onready var strawberryLabel = $HBoxContainer/StrawberryVBox/Label

onready var fruit_totals := {
    "banana": { "current": 0, "total": 0, "labelNode": bananaLabel },
    "strawberry": { "current": 0, "total": 0, "labelNode": strawberryLabel }
}
```

Creating a HUD: Add HUD Update Behavior

Then we create some functions to facilitate updating the current values (the fruit dictionary provided by the Player signal) as well as increasing the total fruit count (called by the Level on start/ready).

```
func update_current_fruit(fruit_data:Dictionary) -> void:
    for fruit in fruit_data.keys():
        fruit_totals.get(fruit).current = fruit_data.get(fruit)
    update_labels()

func increase_fruit_max(fruit:String) -> void:
    fruit_totals.get(fruit.to_lower()).total += 1
    update_labels()
```

```
func update_labels():
    for fruit_value in fruit_totals.values():
        var fruit_label = "%d/%d" % [fruit_value.current, fruit_value.total]
        fruit_value.labelNode.text = fruit_label
```

Creating a HUD: Add HUD Update Behavior

And finally, a function to update our fruit labels when any of the current or total fruit counts change.

```
func update_current_fruit(fruit_data:Dictionary) -> void:
    for fruit in fruit_data.keys():
        fruit_totals.get(fruit).current = fruit_data.get(fruit)
    update_labels()

func increase_fruit_max(fruit:String) -> void:
    fruit_totals.get(fruit.to_lower()).total += 1
    update_labels()
```

```
func update_labels():
    for fruit_value in fruit_totals.values():
        var fruit_label = "%d/%d" % [fruit_value.current, fruit_value.total]
        fruit_value.labelNode.text = fruit_label
```

Creating a HUD: Connect Signals via the Level

With the Player and the HUD set up to do their respective parts, it is up to the Level to set up the communication between itself and its child nodes (the HUD and the Player). This happens on the _ready() function of a new Level script.

```
extends Node2D

func _ready():
    $Player.connect("fruit_collected", $HUD, "update_current_fruit")
    for fruit_child in $Collectibles.get_children():
        if fruit_child.get("fruit") != null:
            $HUD.increase_fruit_max(fruit_child.fruit)
```

Creating a HUD: Make the Level a Conductor

Using the SOURCE of the signal (the Player node), we connect its “fruit_collected” signal to a target object intended to react to the signal (the HUD node) via a specific function on the target object.

“HUD, please update your labels when the Player says it collected fruit.”

```
extends Node2D

func _ready():
    $Player.connect("fruit_collected", $HUD, "update_current_fruit")
    for fruit_child in $Collectibles.get_children():
        if fruit_child.get("fruit") != null:
            $HUD.increase_fruit_max(fruit_child.fruit)
```

```
connect(signal: String, target: Object, method: String)
```

Creating a HUD: Make the Level a Conductor

Finally, we loop through all of the child nodes in the Collectibles node (which should only be Fruit nodes) to safely update the total fruit counts in the HUD.

```
extends Node2D

func _ready():
    $Player.connect("fruit_collected", $HUD, "update_current_fruit")
    for fruit_child in $Collectibles.get_children():
        if fruit_child.get("fruit") != null:
            $HUD.increase_fruit_max(fruit_child.fruit)
```

Lab Time & Break (~15 Minutes)

- Update the Player script to emit a fruit_collected signal when the Player collects fruit.
- Create a HUD script with functions to update the fruit labels.
- Create a Level script to connect the Player signal to an appropriate HUD function to update fruit labels when fruit is collected.
- Have the Level script set/increase the max fruit counts on startup.

Session Goal #7: A Basic Enemy

1. ✅ Controllable Player
2. ✅ Reusable Base Level
3. ✅ Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. ✅ Level Checkpoints
5. ✅ Hazards & Player Respawn
6. ✅ Collectibles & HUD
7. A Basic Enemy
8. Main Menu & Changing Scenes
9. Open Workshop, Tinker Time

Create a new SnailEnemy Scene

- A Scene new KinematicBody2D scene named SnailEnemy.

- Mimic the scene tree structure seen to the right.

Note: You can hide the RayCast2D for now (or otherwise ignore it in the editor). We will discuss it after going over the properties of familiar nodes.

Add Animations and Hit Sound

- Create two animations for the AnimatedSprite node (20 FPS, looping):

  "defeated"

  Assets/Art/Pixel Adventure 2/Enemies/Snail/Walk (38x24).png

  "walk" *(first frame only)*

  Assets/Art/Pixel Adventure 2/Enemies/Snail/Idle (38x24).png

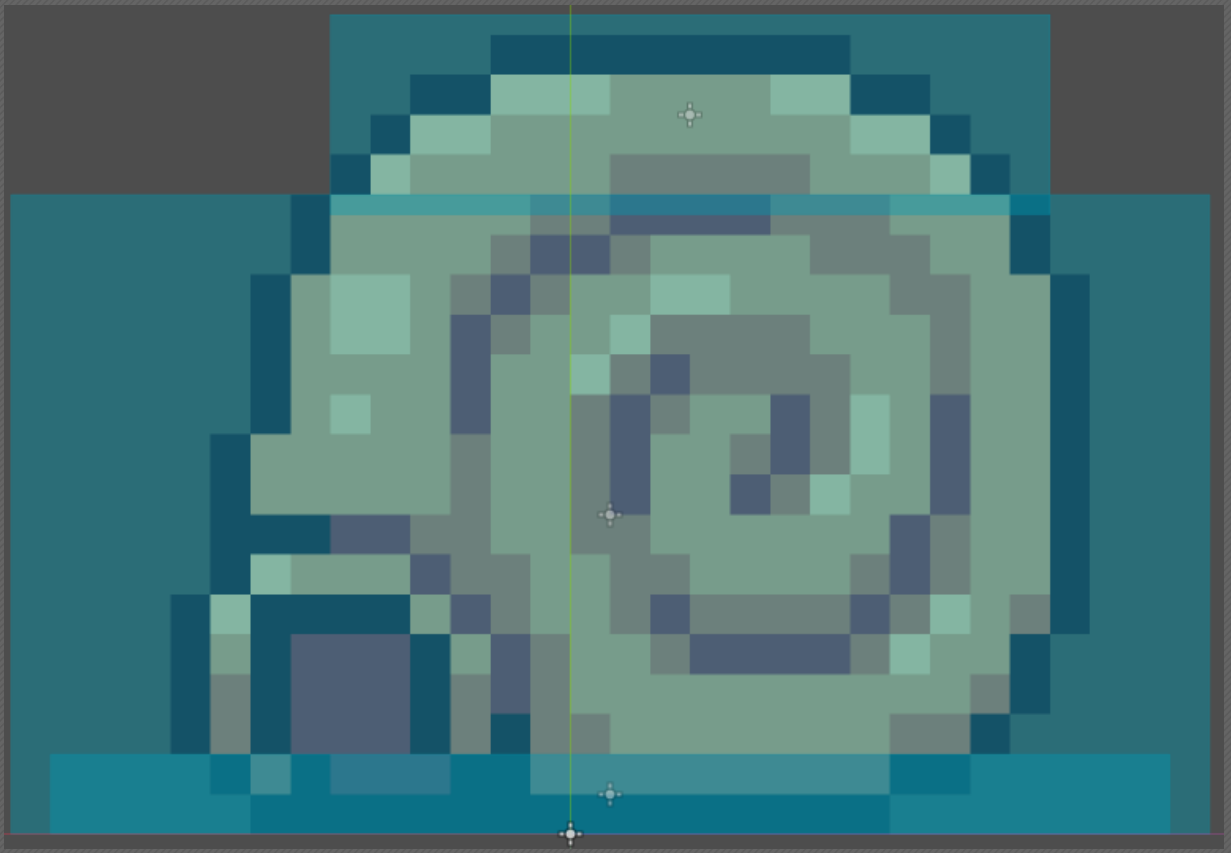
- Use the following sound for the HitAudioStreamPlayer "Stream" sound:

  Assets/Audio/jdwasabi_Sounds/Hit damage 1.wav

Note: The AnimatedSprite should be positioned so that it is sitting on the scene's center position.

Build CollisionShape2D Shapes

Mimic each of the CollisionShape2D node shapes as seen to the right for the following parent nodes:

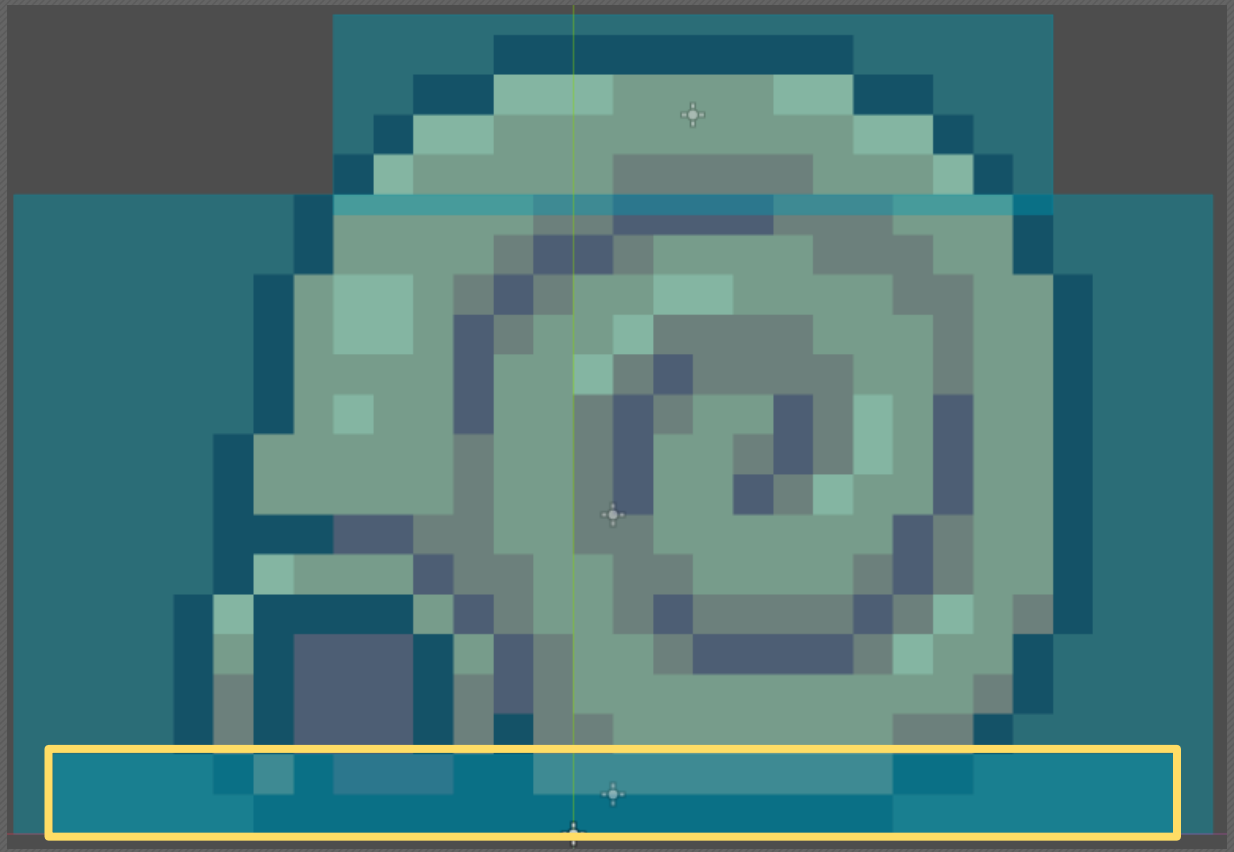


Build CollisionShape2D Shapes

Mimic each of the CollisionShape2D node shapes as seen to the right for the following parent nodes:

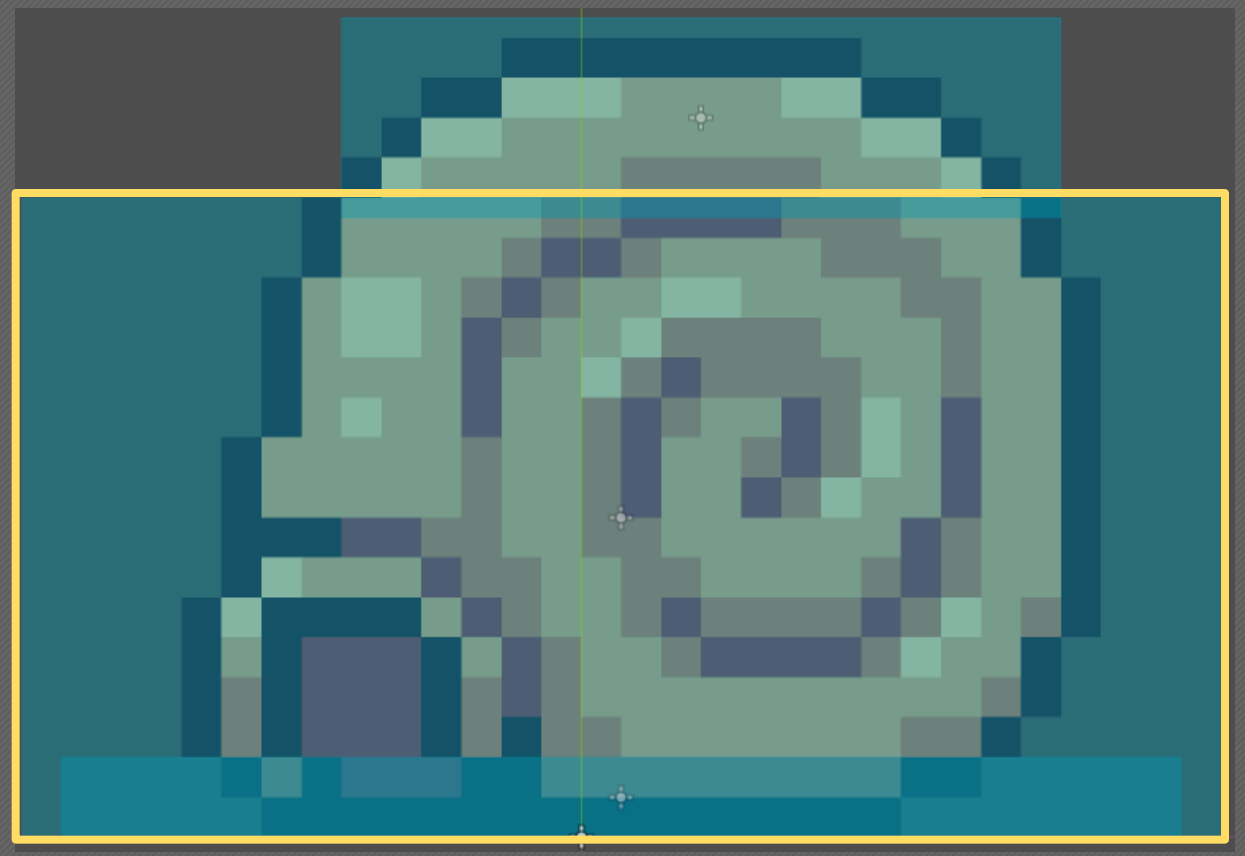
SnailEnemy *(the root node)*

This collision shape will be used to keep the KinematicBody2D on the ground, much like how the Player behaves.



Build CollisionShape2D Shapes

Mimic each of the CollisionShape2D node shapes as seen to the right for the following parent nodes:

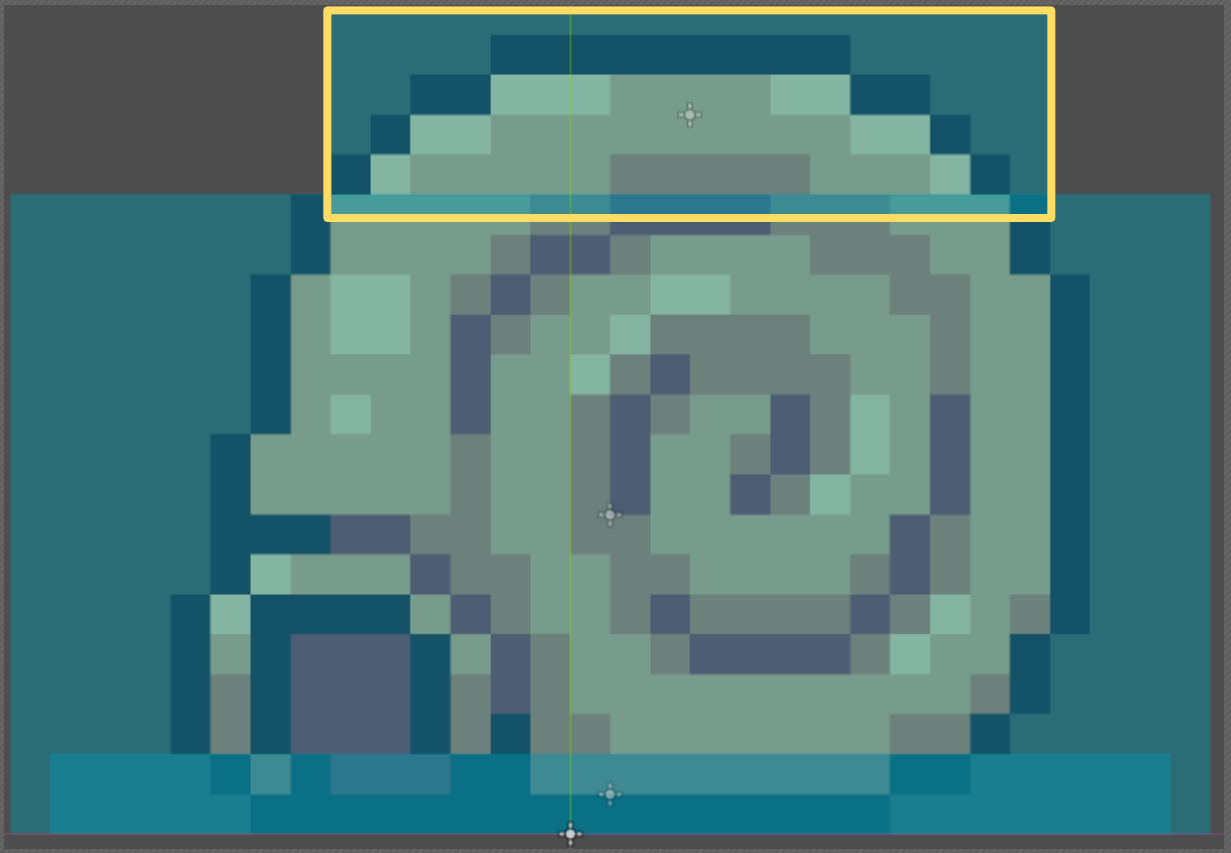


HitArea2D

This HitArea2D shape will be used for collision detection with the Player. Collision with this shape will trigger the Player to respawn.

Build CollisionShape2D Shapes

Mimic each of the CollisionShape2D node shapes as seen to the right for the following parent nodes:



ShellArea2D

This ShellArea2D will be the area that the Player can jump on to defeat the Snail.

Add Basic Movement to the SnailEnemy Script

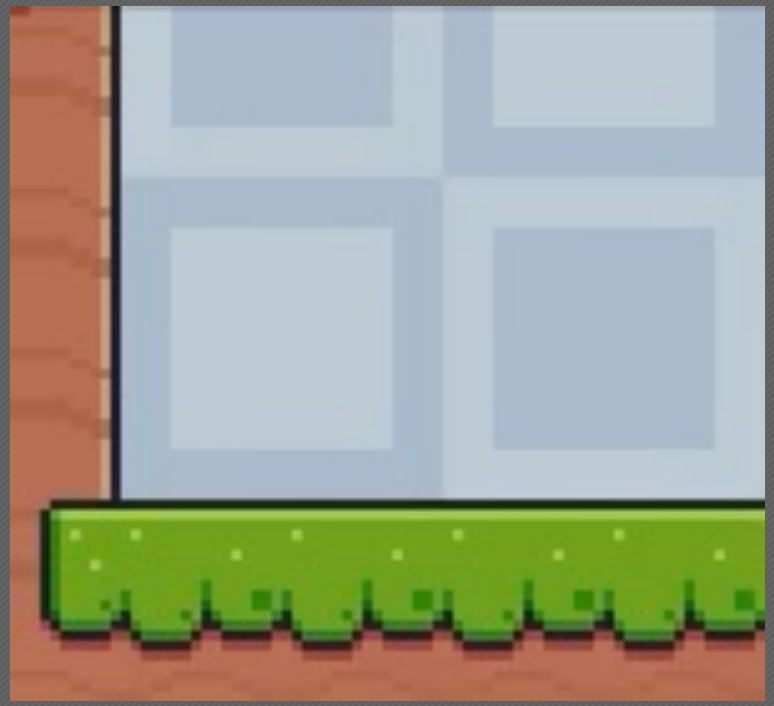
Since the Snail's sprites face left by default, we simply modify the X velocity by the negative move speed to flip it.

```
extends KinematicBody2D

const DEFEATED_BOUNCE_FORCE := 250  # will be used soon
const GRAVITY = 25
onready var move_speed : float = 35 * scale.x

var is_alive := true  # will be used soon
var velocity = Vector2.ZERO

func _physics_process(delta):
    if is_alive: velocity.x = -move_speed
    velocity.y += GRAVITY
    velocity = move_and_slide(velocity, Vector2.UP)

    if is_alive and velocity.x != 0: $AnimatedSprite.play("walk")
```

Turning Around on Collision with a Wall

The KinematicBody2D has a convenient is_on_wall() function to make this a relatively “trivial” implementation.

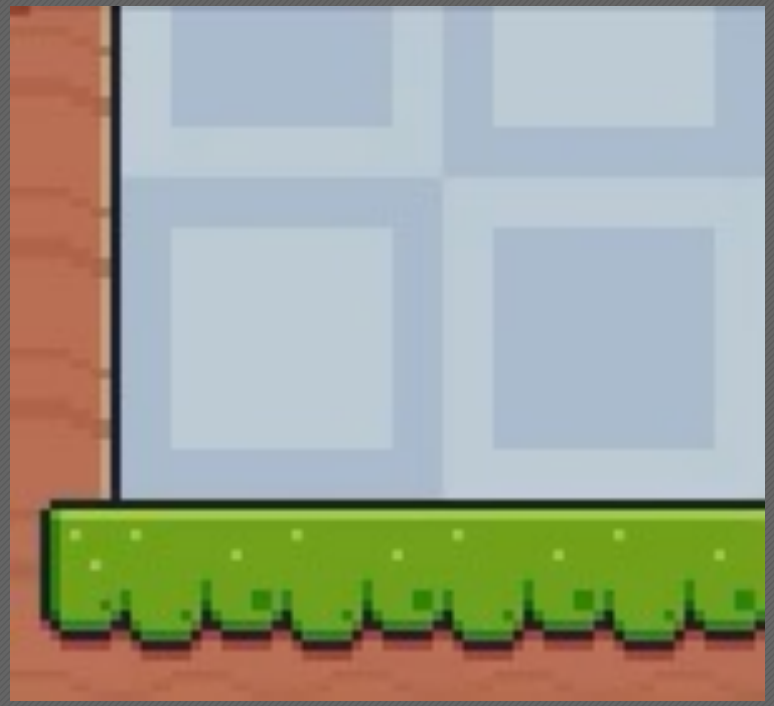
```
func _physics_process(delta):
    # ...
    if is_alive and is_on_wall():
        turn_around()

func turn_around():
    scale.x *= -1
    move_speed *= -1
```



Turning Around on Collision with a Wall

The inverted X scale “horizontally flips” all nodes in the scene.
The inverted move_speed changes its direction.

```
func _physics_process(delta):
    # ...
    if is_alive and is_on_wall():
        turn_around()

func turn_around():
    scale.x *= -1
    move_speed *= -1
```



Connect SnailEnemy HitArea with the Player

SnailEnemy.gd

```
func _on_HitArea2D_body_entered(body):
    if body.has_method("take_enemy_damage"):
        body.take_enemy_damage()
```

Note: *Don't forget to connect the HitArea2D signal to the SnailEnemy script via the Node tab!*

Player.gd

```
func take_enemy_damage() -> void:
    respawn()
```

A simple duplication of take_spike_damage() for easy changes later.

Taking Damage from the SnailEnemy

Ultimately, we have two ways the Player can collide with a snail:

1. The SnailEnemy collides with the Player
2. The Player collides with the SnailEnemy *(e.g., from behind)*

The move_and_slide function stores collisions as a side effect. Collisions are only triggered by the *collider*, and that function will not trigger any collisions that happen outside of the slide (like from behind).

```
func _physics_process(delta):
    # ...
    velocity = move_and_slide(velocity, Vector2.UP)

    for i in get_slide_count():
        if get_slide_collision(i).collider.name.begins_with("SnailEnemy"):
            take_enemy_damage()
```

What is a RayCast2D?

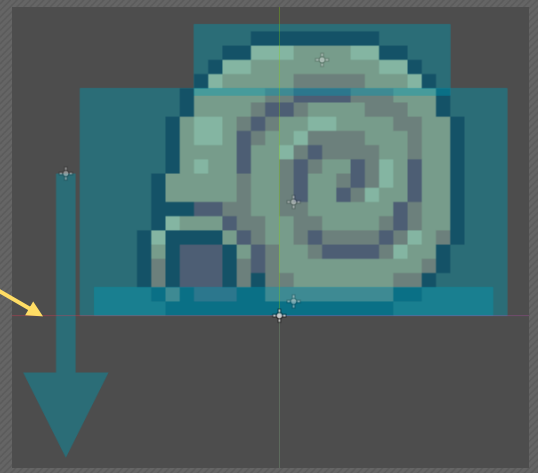
- A RayCast2D represents a line from its origin to a destination point on a 2D plane.
- It is used to query a 2D space in order to find the closest object along the path of the ray (if one exists).
- We can use a RayCast2D to detect a ledge, so the snail can turn around and avoid falling into holes or pits.

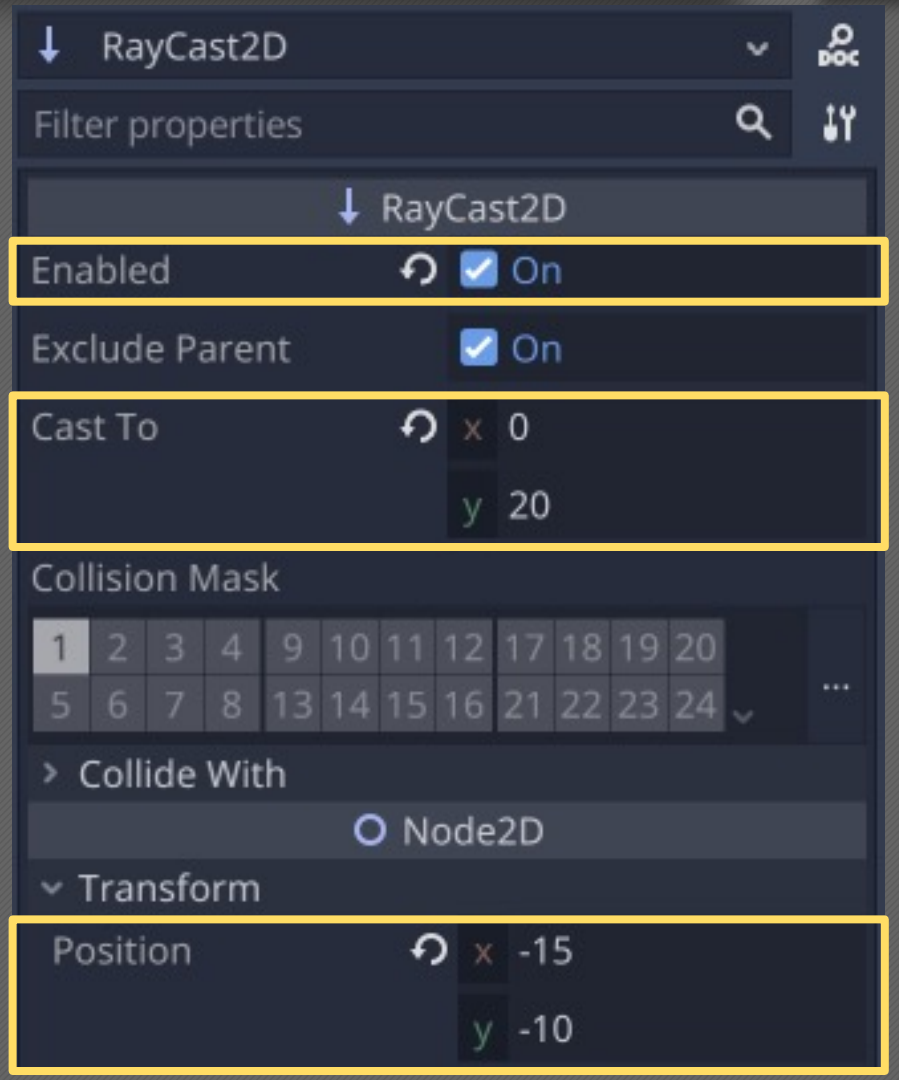
Setting up the RayCast2D

- Remember to make the RayCast2D node visible if you previously disabled or hid it.

- Update the RayCast2D properties as seen to the right.

Notice the position and length of the RayCast2D arrow.





Using the RayCast2D to avoid ledges

Update the SnailEnemy to also check for a missing collision on the RayCast2D ("no collision" means "no ground").

```
func _physics_process(delta):
    # ...
    if is_alive and velocity.x != 0: $AnimatedSprite.play("walk")
    var is_wall_or_ledge = is_on_wall() or not $RayCast2D.is_colliding()
    if is_alive and is_wall_or_ledge: turn_around()
```

Defeating the Snail by Bouncing On It

Remember – you get these function names by connecting to signals in the Node tab!

```
func _on_ShellArea2D_body_entered(body):
    if is_alive:
        is_alive = false
        if body.has_method("bounce"): body.bounce()
        set_collision_layer_bit(0, false)
        set_collision_mask_bit(0, false)
        $HitArea2D.set_collision_layer_bit(0, false)
        $HitArea2D.set_collision_mask_bit(0, false)
        $ShellArea2D.set_collision_layer_bit(0, false)
        $ShellArea2D.set_collision_mask_bit(0, false)
        $HitAudioStreamPlayer.play()
        velocity.y -= DEFEATED_BOUNCE_FORCE
        $AnimatedSprite.play("defeated")
        yield(get_tree().create_timer(1), "timeout")
        queue_free()
```

Defeating the Snail by Bouncing On It

Do some basic checks to determine if the Snail is alive. Immediately set is_alive to false to prevent multiple executions.

```
func _on_ShellArea2D_body_entered(body):
    if is_alive:
        is_alive = false
        if body.has_method("bounce"): body.bounce()
        set_collision_layer_bit(0, false)
        set_collision_mask_bit(0, false)
        $HitArea2D.set_collision_layer_bit(0, false)
        $HitArea2D.set_collision_mask_bit(0, false)
        $ShellArea2D.set_collision_layer_bit(0, false)
        $ShellArea2D.set_collision_mask_bit(0, false)
        $HitAudioStreamPlayer.play()
        velocity.y -= DEFEATED_BOUNCE_FORCE
        $AnimatedSprite.play("defeated")
        yield(get_tree().create_timer(1), "timeout")
        queue_free()
```

Defeating the Snail by Bouncing On It

If the colliding body (e.g., the Player) has a “bounce” method, call it. We will add this soon.

```
func _on_ShellArea2D_body_entered(body):
    if is_alive:
        is_alive = false
        if body.has_method("bounce"): body.bounce()
        set_collision_layer_bit(0, false)
        set_collision_mask_bit(0, false)
        $HitArea2D.set_collision_layer_bit(0, false)
        $HitArea2D.set_collision_mask_bit(0, false)
        $ShellArea2D.set_collision_layer_bit(0, false)
        $ShellArea2D.set_collision_mask_bit(0, false)
        $HitAudioStreamPlayer.play()
        velocity.y -= DEFEATED_BOUNCE_FORCE
        $AnimatedSprite.play("defeated")
        yield(get_tree().create_timer(1), "timeout")
        queue_free()
```

Defeating the Snail by Bouncing On It

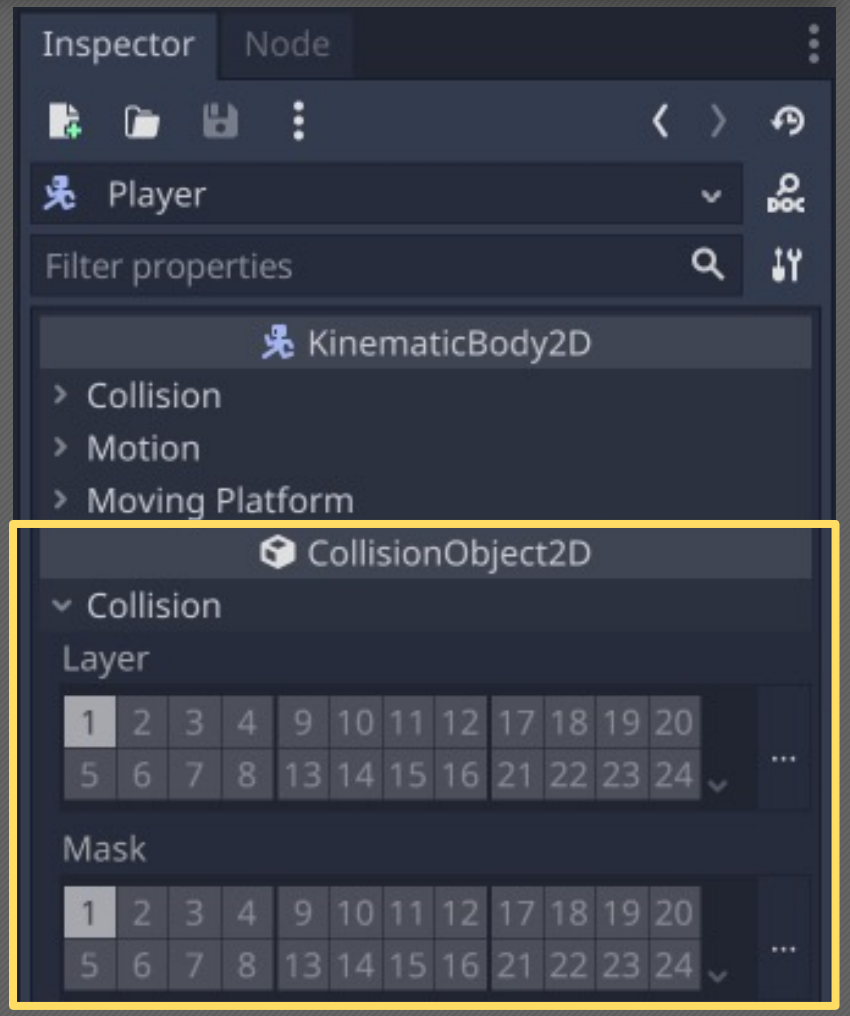
Disable collisions on the KinematicBody2D, HitArea2D, ShellArea2D.

```
func _on_ShellArea2D_body_entered(body):
    if is_alive:
        is_alive = false
        if body.has_method("bounce"): body.bounce()
        set_collision_layer_bit(0, false)
        set_collision_mask_bit(0, false)
        $HitArea2D.set_collision_layer_bit(0, false)
        $HitArea2D.set_collision_mask_bit(0, false)
        $ShellArea2D.set_collision_layer_bit(0, false)
        $ShellArea2D.set_collision_mask_bit(0, false)
        $HitAudioStreamPlayer.play()
        velocity.y -= DEFEATED_BOUNCE_FORCE
        $AnimatedSprite.play("defeated")
        yield(get_tree().create_timer(1), "timeout")
        queue_free()
```

A Quick Note on Collision Layers and Masks

- Each CollisionObject2D object has Collision Layer and Mask values.
- A Collision Layer is the logical layer(s) this node belongs to.
- A Collision Mask indicates what collision Layers this node can collide or interact with.

We do not otherwise use Layers or Masks in this project, but they are very effective and useful for larger projects.



Defeating the Snail by Bouncing On It

Disable collisions on the KinematicBody2D, HitArea2D, ShellArea2D.

```
func _on_ShellArea2D_body_entered(body):
    if is_alive:
        is_alive = false
        if body.has_method("bounce"): body.bounce()
        set_collision_layer_bit(0, false)
        set_collision_mask_bit(0, false)
        $HitArea2D.set_collision_layer_bit(0, false)
        $HitArea2D.set_collision_mask_bit(0, false)
        $ShellArea2D.set_collision_layer_bit(0, false)
        $ShellArea2D.set_collision_mask_bit(0, false)
        $HitAudioStreamPlayer.play()
        velocity.y -= DEFEATED_BOUNCE_FORCE
        $AnimatedSprite.play("defeated")
        yield(get_tree().create_timer(1), "timeout")
        queue_free()
```

Defeating the Snail by Bouncing On It

Play the Hit sound, give it a little bounce effect, and play the empty-shell “defeated” animation.

```
func _on_ShellArea2D_body_entered(body):
    if is_alive:
        is_alive = false
        if body.has_method("bounce"): body.bounce()
        set_collision_layer_bit(0, false)
        set_collision_mask_bit(0, false)
        $HitArea2D.set_collision_layer_bit(0, false)
        $HitArea2D.set_collision_mask_bit(0, false)
        $ShellArea2D.set_collision_layer_bit(0, false)
        $ShellArea2D.set_collision_mask_bit(0, false)
        $HitAudioStreamPlayer.play()
        velocity.y -= DEFEATED_BOUNCE_FORCE
        $AnimatedSprite.play("defeated")
        yield(get_tree().create_timer(1), "timeout")
        queue_free()
```

Defeating the Snail by Bouncing On It

The yield function is a synchronous pause in the current script until the defined signal (timeout) is emitted.

```
func _on_ShellArea2D_body_entered(body):
    if is_alive:
        is_alive = false
        if body.has_method("bounce"): body.bounce()
        set_collision_layer_bit(0, false)
        set_collision_mask_bit(0, false)
        $HitArea2D.set_collision_layer_bit(0, false)
        $HitArea2D.set_collision_mask_bit(0, false)
        $ShellArea2D.set_collision_layer_bit(0, false)
        $ShellArea2D.set_collision_mask_bit(0, false)
        $HitAudioStreamPlayer.play()
        velocity.y -= DEFEATED_BOUNCE_FORCE
        $AnimatedSprite.play("defeated")
        yield(get_tree().create_timer(1), "timeout")
        queue_free()
```

Defeating the Snail by Bouncing On It

queue_free() is a system func that marks the object (e.g., this whole SnailEnemy scene instance) ready for garbage collection

```
func _on_ShellArea2D_body_entered(body):
    if is_alive:
        is_alive = false
        if body.has_method("bounce"): body.bounce()
        set_collision_layer_bit(0, false)
        set_collision_mask_bit(0, false)
        $HitArea2D.set_collision_layer_bit(0, false)
        $HitArea2D.set_collision_mask_bit(0, false)
        $ShellArea2D.set_collision_layer_bit(0, false)
        $ShellArea2D.set_collision_mask_bit(0, false)
        $HitAudioStreamPlayer.play()
        velocity.y -= DEFEATED_BOUNCE_FORCE
        $AnimatedSprite.play("defeated")
        yield(get_tree().create_timer(1), "timeout")
        queue_free()
```

Bounce Player After Jumping on SnailEnemy

Player.gd

```
func bounce() -> void:
    velocity.y = -JUMP_FORCE
```

Update the Player script to have a bounce() function, which is triggered by bouncing on the Snail as seen in the last slides.

Lab Time (~25 Minutes)

- Create the SnailEnemy scene.
- Implement the SnailsEnemy's movement and collision mechanics.
- Add multiple SnailEnemy instances to your Level scene.

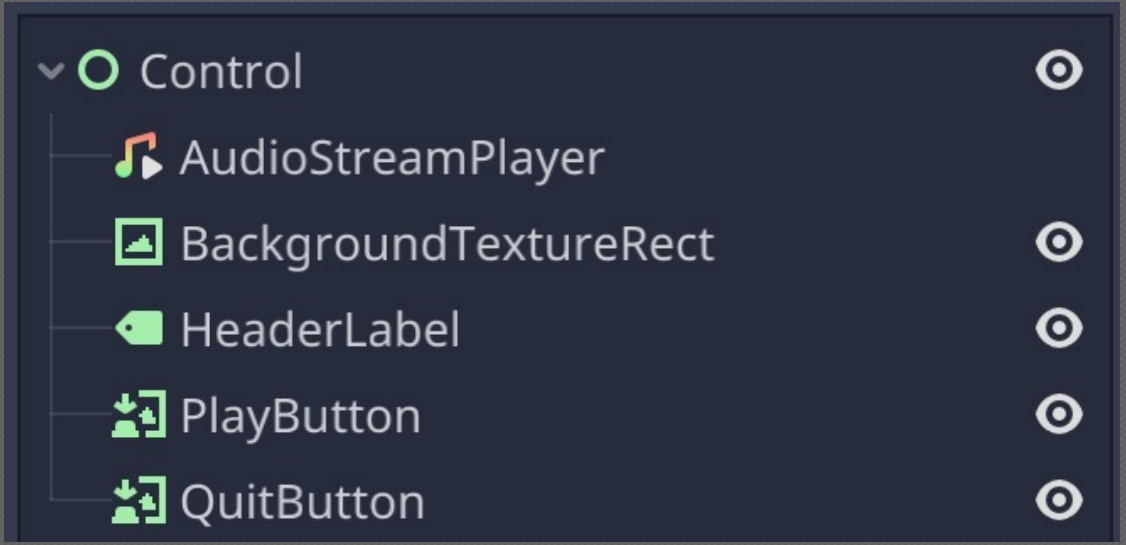
Session Goal #8: Main Menu

1. ✅ Controllable Player
2. ✅ Reusable Base Level
3. ✅ Player & Level Polish
   (Camera, Gravity, Sound, Background)
4. ✅ Level Checkpoints
5. ✅ Hazards & Player Respawn
6. ✅ Collectibles & HUD
7. ✅ A Basic Enemy
8. Main Menu & Scene Transition
9. Open Workshop, Tinker Time

Creating a Main Menu

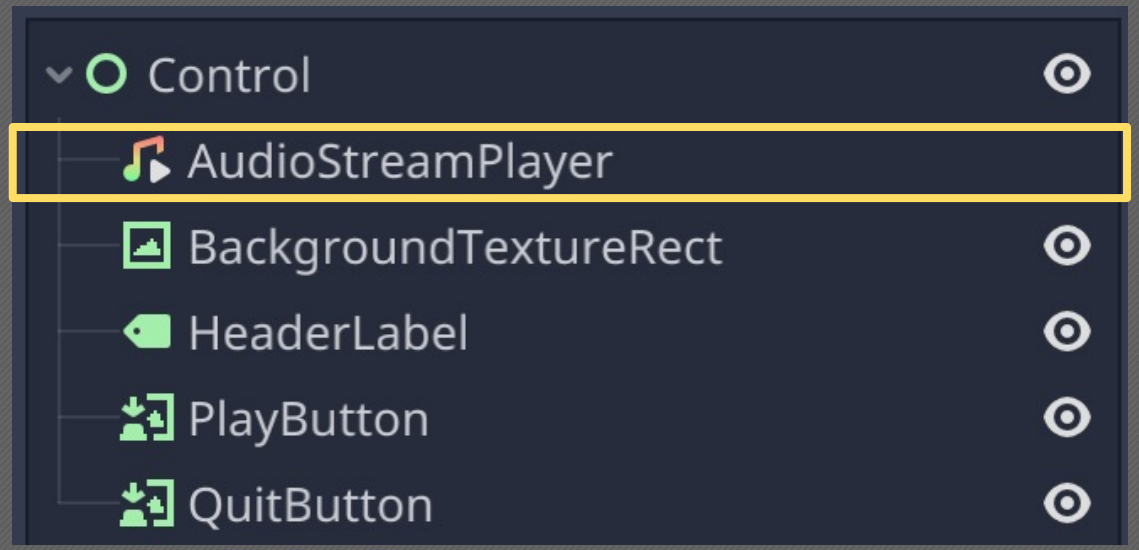
- Create a new Scene using the “User Interface” (the Control type) root node.

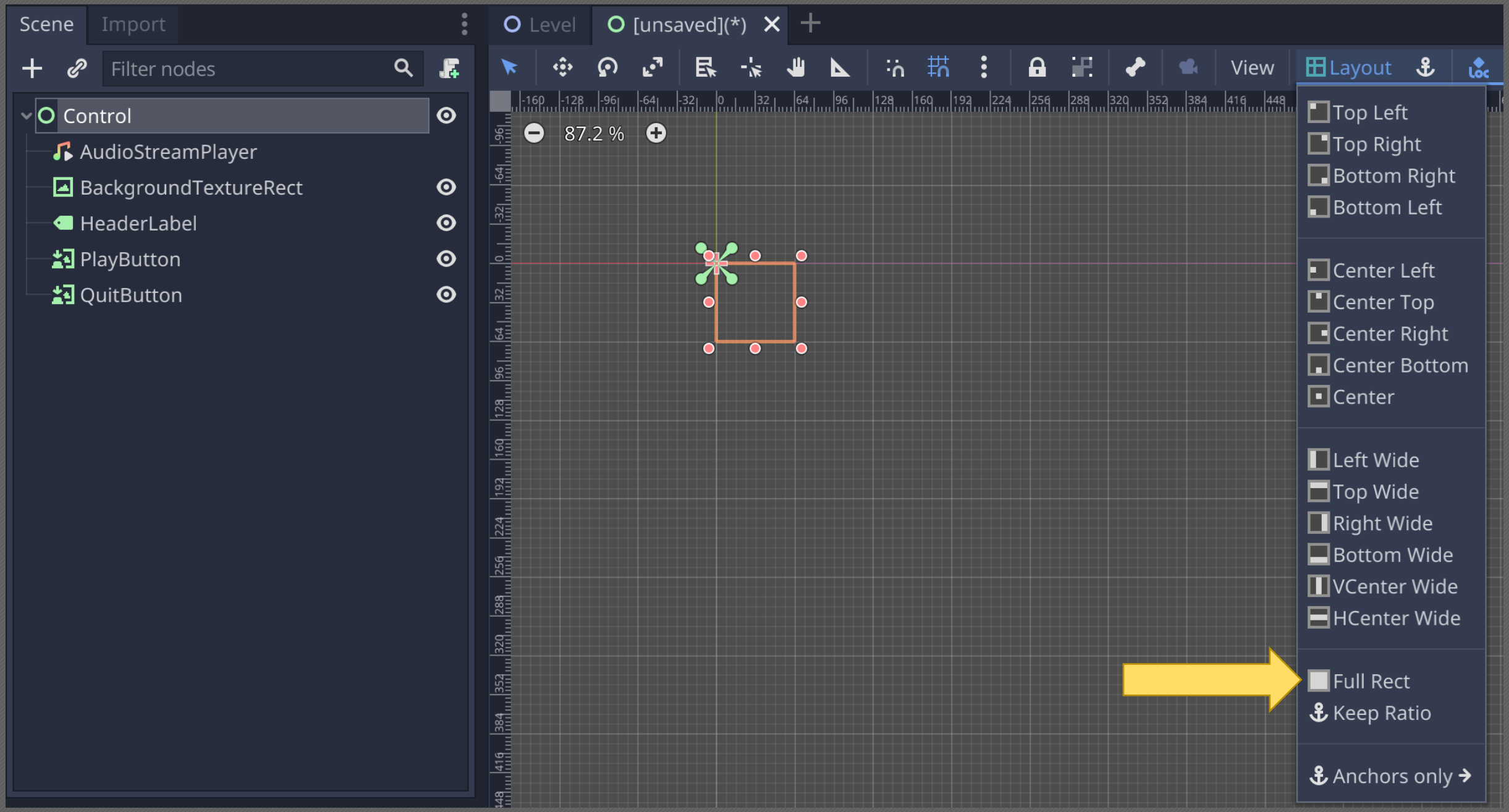
- Create the scene tree structure seen to the right (all nodes end with their node type for clarity).



Creating a Main Menu: Music

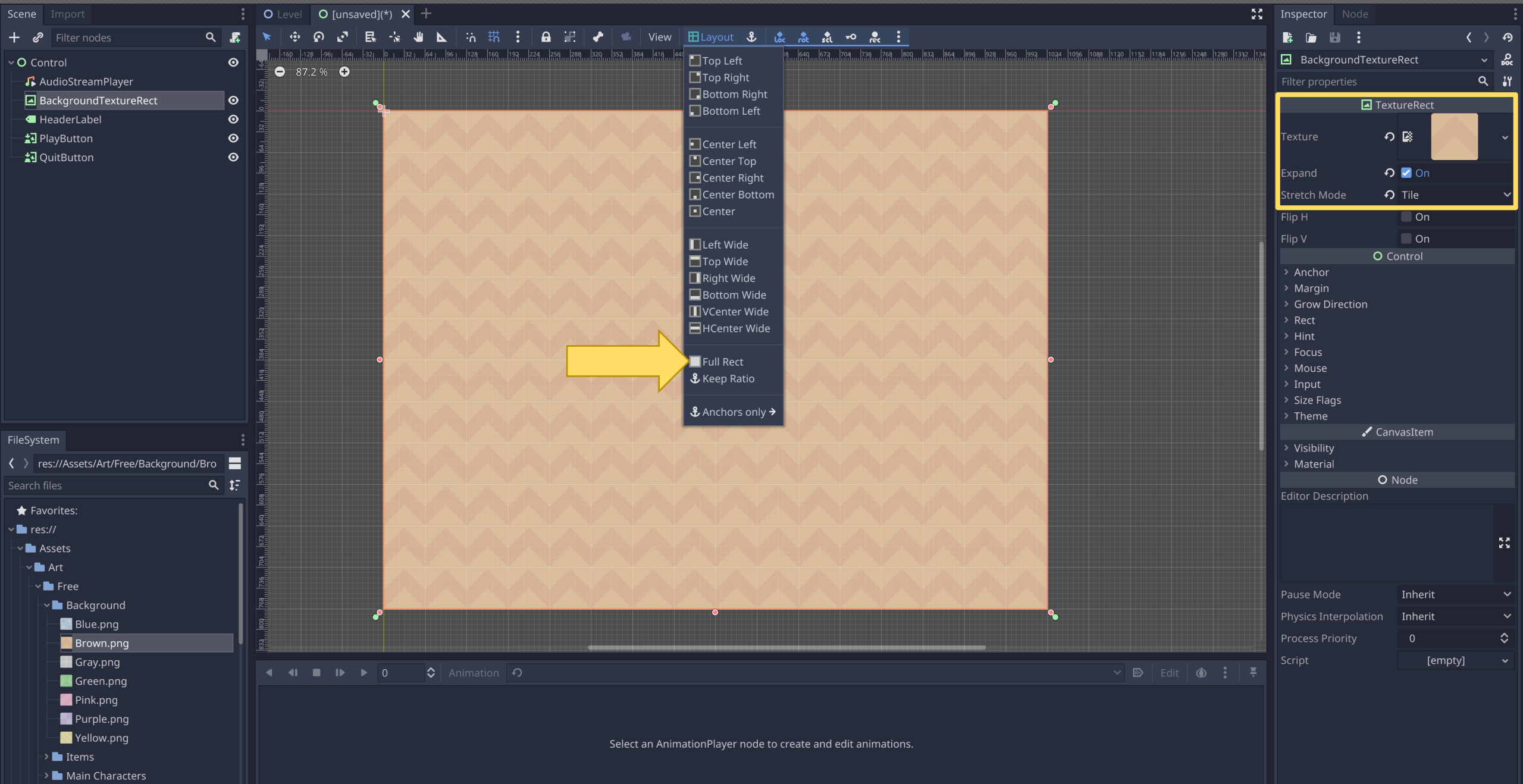
- With AudioStreamPlayer selected, drag and drop any of the audio files (Assets/Free/Sounds/Casual Game Vol 2) into the Stream property.
- Set the AudioStreamPlayer's Autoplay property to true.
- Select the audio file in the FileSystem, select the Import tab next to the Scene tab, set the Loop Mode to Forward, and Reimport the file.



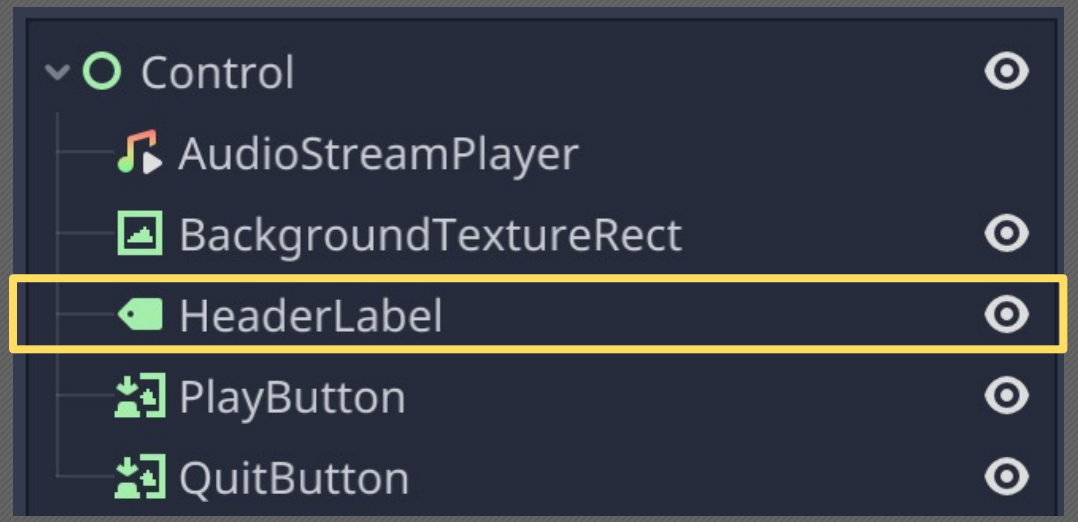

Scene
Import
Filter nodes
Control
AudioStreamPlayer
BackgroundTextureRect
HeaderLabel
PlayButton
QuitButton
Level
[unsaved](*)
View
Layout
87.2 %
Top Left
Top Right
Bottom Right
Bottom Left
Center Left
Center Top
Center Right
Center Bottom
Center
Left Wide
Top Wide
Right Wide
Bottom Wide
VCenter Wide
HCenter Wide
Full Rect
Keep Ratio
Anchors only

Assign a Texture to the BackgroundTextureRect.

Assets/Free/Background



Creating a Main Menu: Title/Header Label



Give the HeaderLabel node a Full Rect Layout, and set these properties:

Text: *any name*

Align & V Align: Center

Theme Overrides:

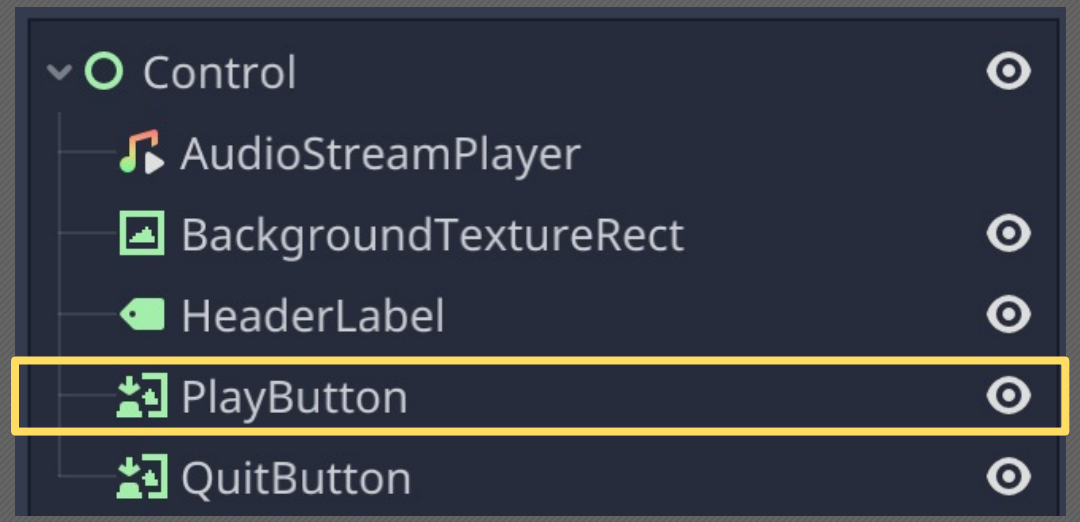
Fonts:

Font: New Dynamic Font

(Font Data = PixelDash.ttf)

Customize the Font and Control Theme Override properties to be styled to your liking.

Creating a Main Menu: Quit Button



With the PlayButton selected, give it a Rect size of about 96x96 and position it somewhere under the title.

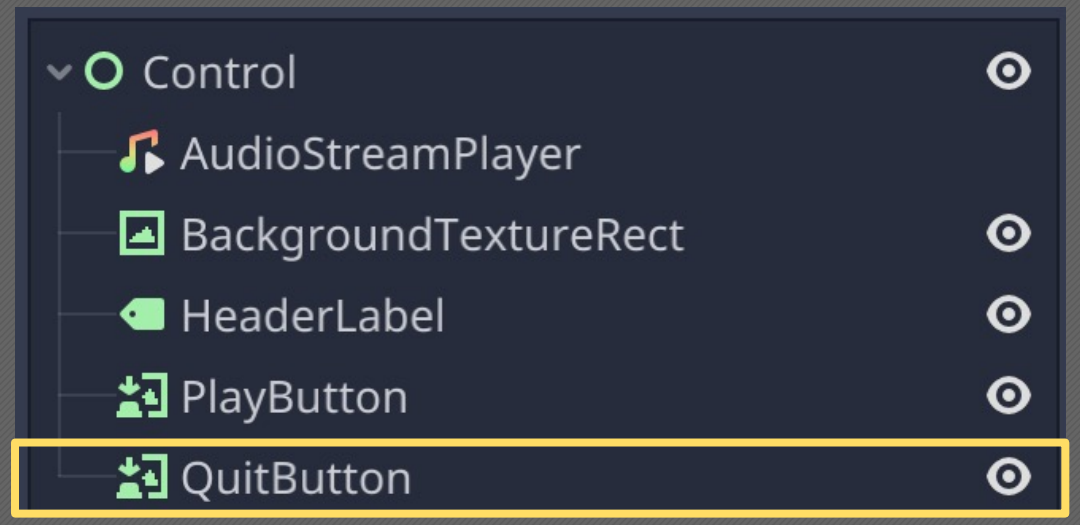
Expand: On

Stretch Mode: Keep Aspect Centered

Texture > Normal:

Assets/Free/Menu/Buttons/Play.png

Creating a Main Menu: Quit Button



With the QuitButton selected, set its Layout to “Top Right”, position it, and set its properties to the following:

Expand:	On
Stretch Mode:	Keep Aspect Centered
Texture > Normal:	
	Assets/Free/Menu/Buttons/Close.png
Rect > Size:	56 x 56

Scene Management

- Godot runs a single “top-level scene” at any point in time.
- Naturally, every project’s scenes are organized and managed differently based on project size and needs.
- Scene Managers are used in many projects to make the process of transitioning between scenes easier, as well as providing loading screens and screen transition animations.

Main Menu Button Signal Callbacks

Create an attach a MainMenu script to the root node of the MainMenu scene. Then connect the Quit and Play buttons' pressed signal to the MainMenu script with with the following implementations.

```
extends Control

func _on_QuitButton_pressed():
    get_tree().quit()

func _on_PlayButton_pressed():
    get_tree().change_scene("res://Scenes/Level/Level.tscn")
```

Main Menu Button Signal Callbacks

The get_tree() function gives us the currently active scene's scene tree. We then call the quit() function on the scene tree to quit the scene and ultimately close our game.

```
extends Control

func _on_QuitButton_pressed():
    get_tree().quit()

func _on_PlayButton_pressed():
    get_tree().change_scene("res://Scenes/Level/Level.tscn")
```

Main Menu Button Signal Callbacks

The change_scene() on the scene tree changes the currently active scene to the scene (the .tscn file) of a scene saved at a specific project path.

```
extends Control

func _on_QuitButton_pressed():
    get_tree().quit()

func _on_PlayButton_pressed():
    get_tree().change_scene("res://Scenes/Level/Level.tscn")
```

Lab Time (~20 Minutes)

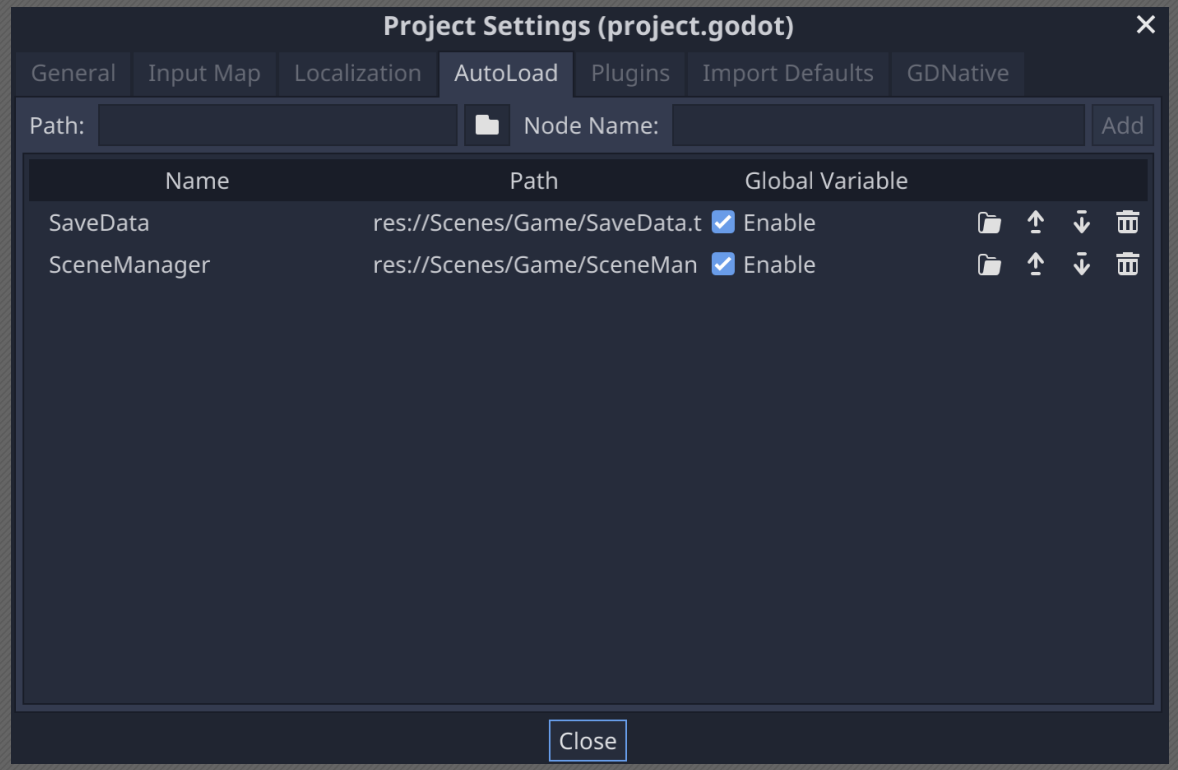
- Create the SnailEnemy scene.
- Implement the SnailsEnemy's movement and collision mechanics.
- Add multiple SnailEnemy instances to your Level scene.

Ideas for Continued Tinkering

- Create multiple levels! The Level script is relatively reusable if you were to duplicate your Level scene.
- Create more enemies! The Art assets folder has over a dozen unique enemies for you to get creative.
- Create more traps, such as falling platforms or a Trampoline that bounces you into the air when you land on it.

AutoLoad (Singletons)

- Any Scenes/Scripts in the AutoLoad tab of your Project Settings will automatically be loaded into the active scene.
- Singletons are loaded top-down as listed in the AutoLoad tab.
- If the Global Variable flag is enabled, the singleton can be referenced in scripts by its Name.



Continued Learning

https://docs.godotengine.org/en/stable/index.html

The Godot docs are *actually* pretty nice, and they're getting better! The Tutorials section is rich with descriptive tutorials for common problems in game development (such as animations, save data, AI navigation, and networking).

YouTube

The HeartBeast and GDQuest are two very popular and high-quality channels dedicated to learning how to develop games in Godot.

https://www.youtube.com/@uheartbeast

https://www.youtube.com/@Gdquest