2021

Introduction to GoDot

Ehsan Ullah

Go Chameleons

4/5/2021

Contents

Introduction to Godot's editor 3
Project manager 3
Create or import a project 5
Your first look at Godot's editor 8
The workspaces 12
Modify the interface 16
Move and resize docks 16
Scenes and nodes 17
Introduction 17
Nodes 17
Scenes 18
Editor 19
Configuring the project 25
Instancing 28
Introduction 28
Instancing by example 29
Multiple instances 32
Editing instances 33
Recap 34
Design language 35
Information overload! 36
Scripting 37
Introduction 37
GDScript 37
VisualScript 37
.NET / C# 38
GDNative / C++ 38
Scripting a scene 38
Scene setup 39
Adding a script 39
The role of the script 42
Handling a signal 42
Processing 46
Groups 47


Notifications 48
Overridable functions 48
Creating nodes 49
Instancing scenes 49
Register scripts as classes 50
Signals 52
Introduction 52
Timer example 52
Connecting signals in code 54
Custom signals 55
Conclusion 56

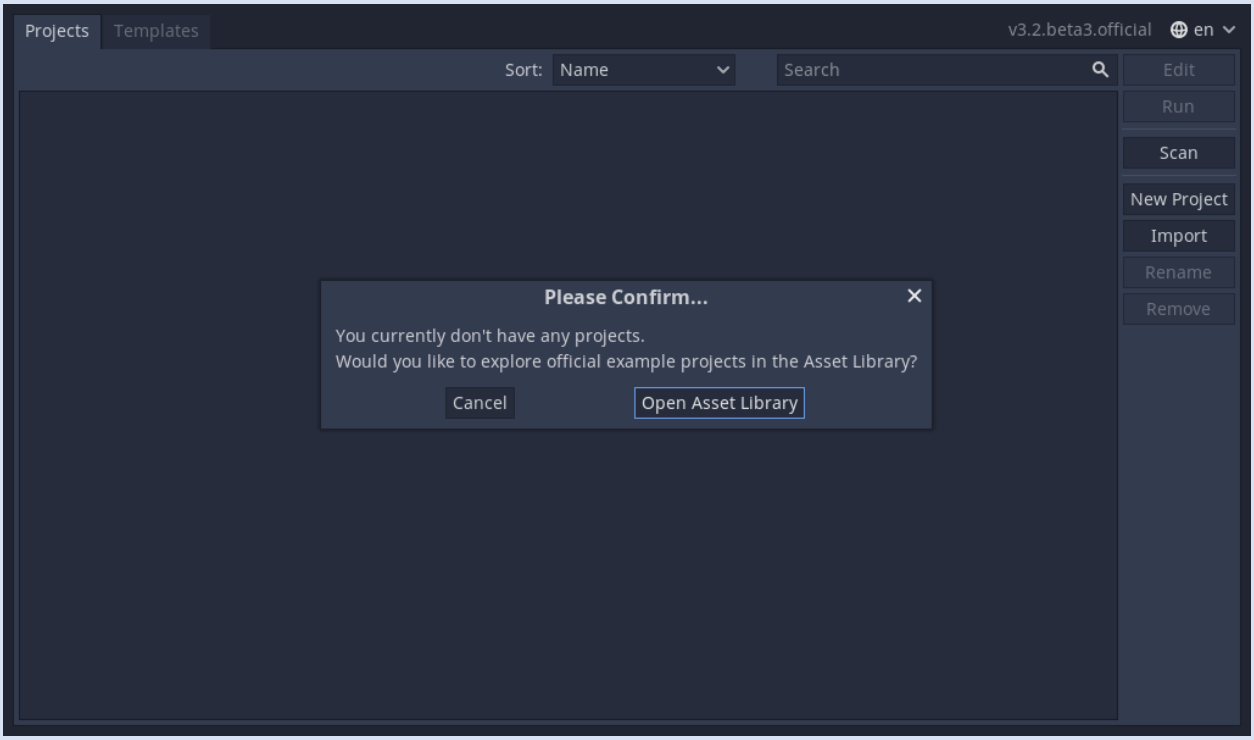
Introduction to Godot's editor

This tutorial will run you through Godot's interface. We're going to look at the Project Manager, docks, workspaces and everything you need to know to get started with the engine.

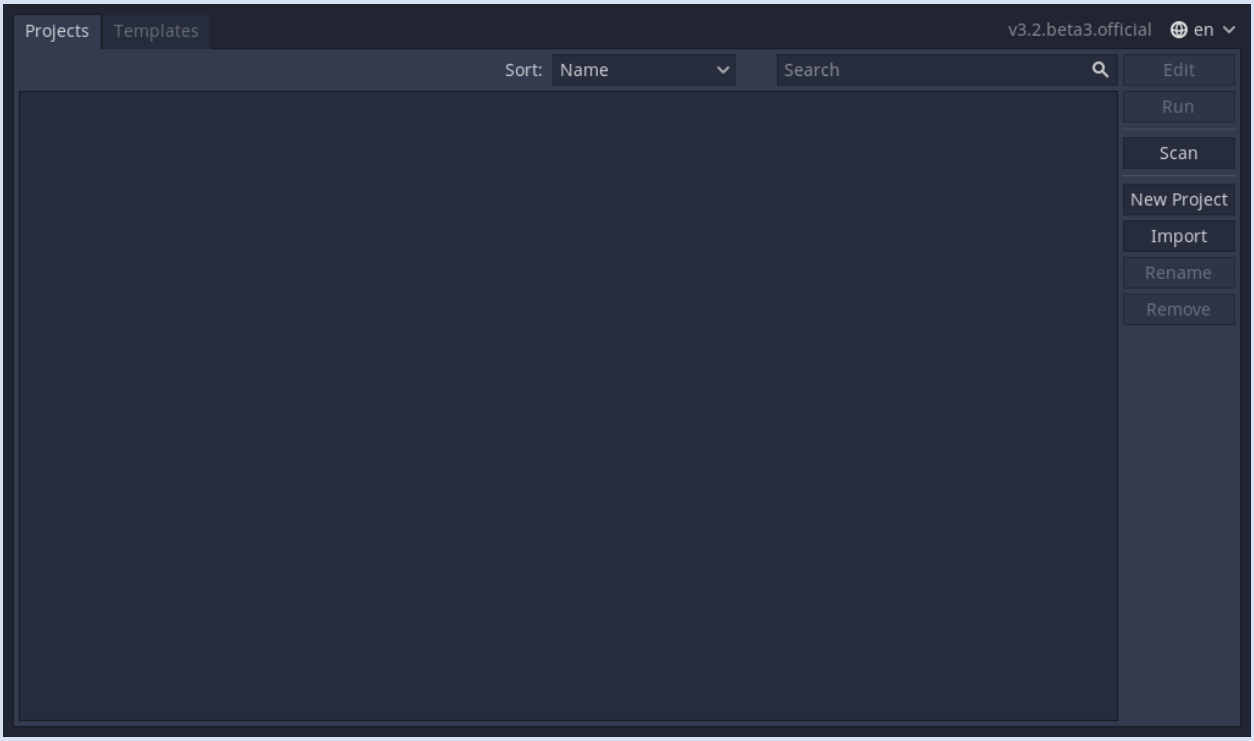
You can download Godot Engine here.

Project manager

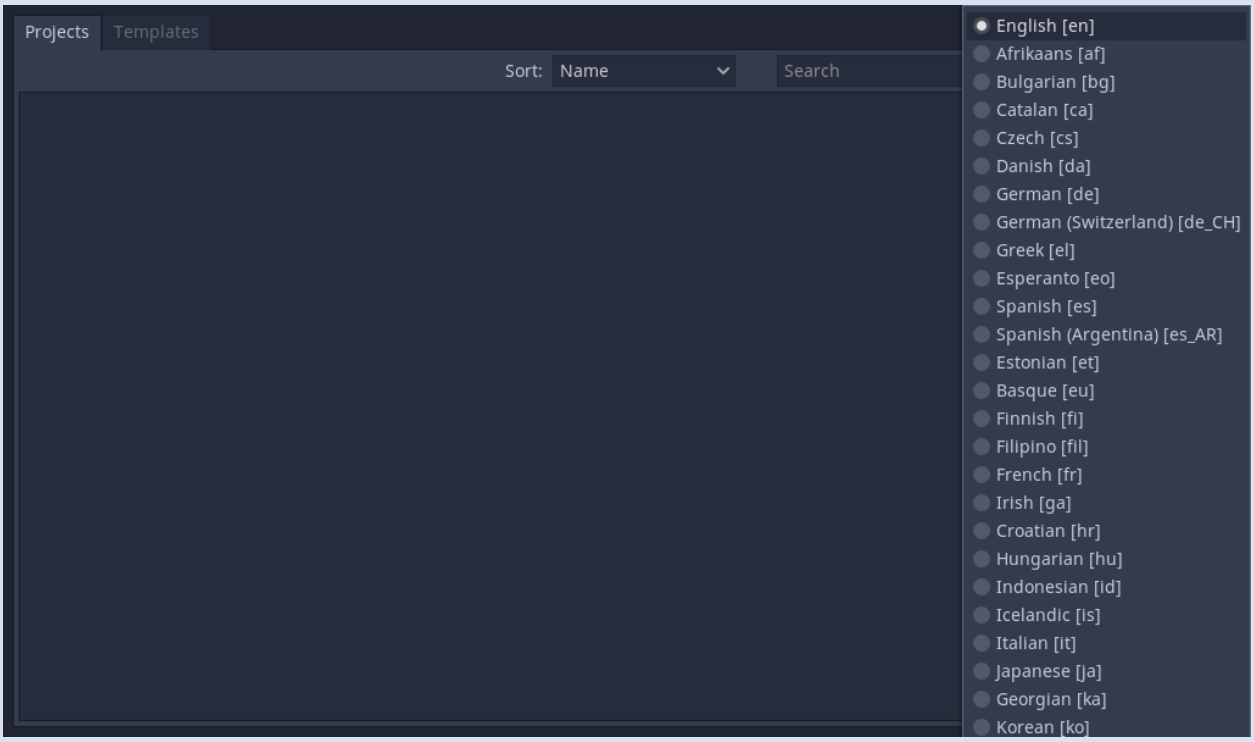
When you launch Godot, the first window you'll see is the Project Manager. Since you have no projects there will be a popup asking if you want to open the asset library, just click cancel, we'll look at it later.



Now you should see the project manager. It lets you create, remove, import or play game projects.

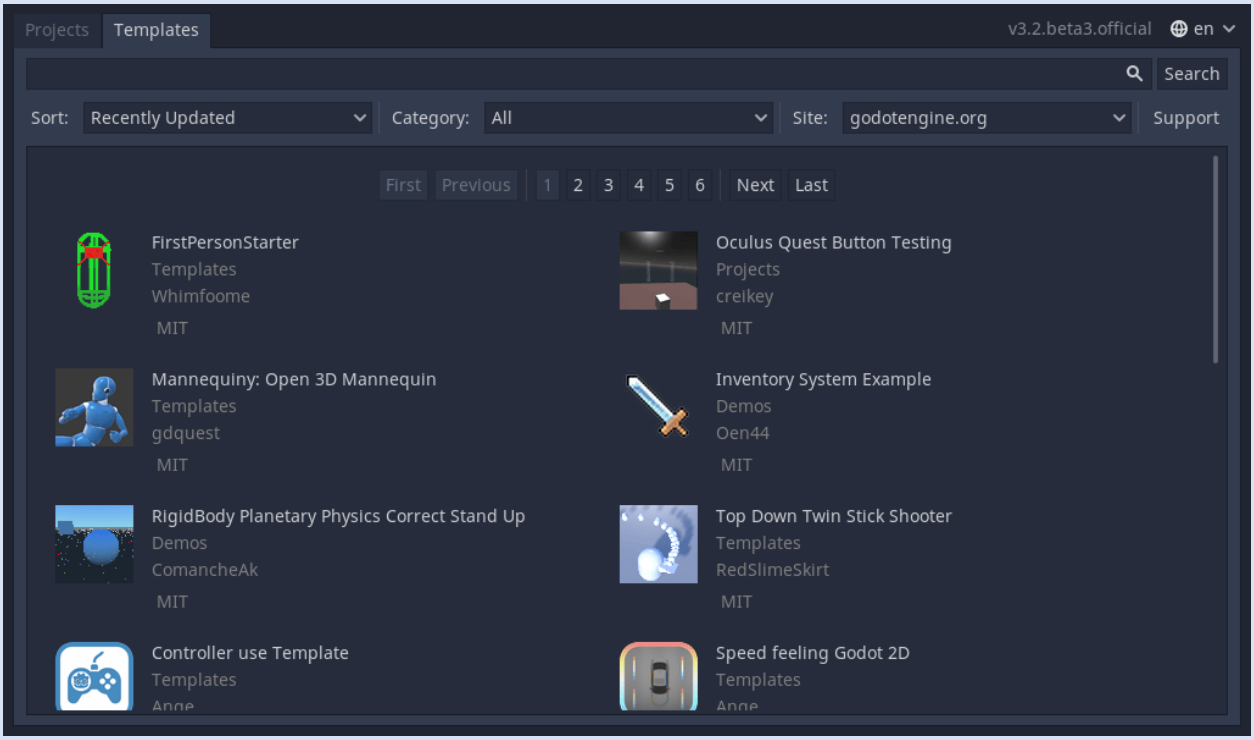


In the top-right corner you'll find a drop-down menu to change the editor's language.



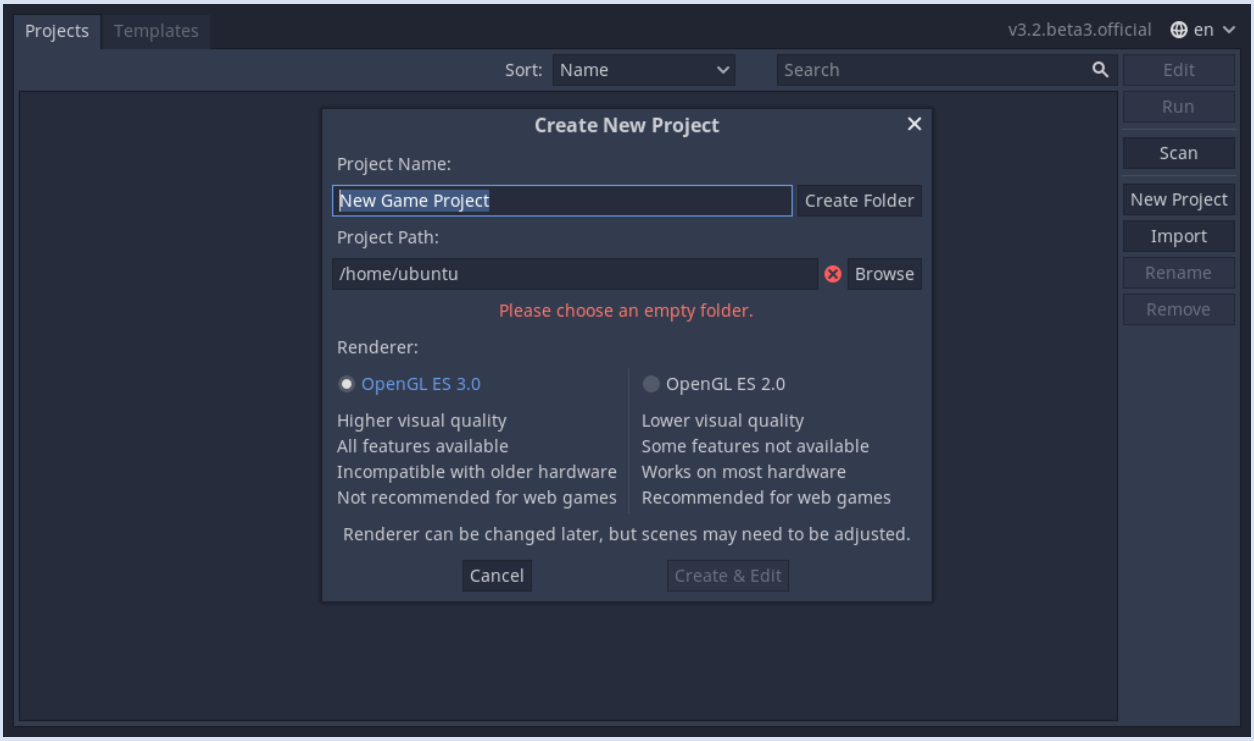
From the Templates tab you can download open source project templates and demos from the Asset Library to help you get started faster. Just select the template or demo you want, click

download, once it's finished downloading click install and choose where you want the project to go. You can learn more about it in About the Asset Library.

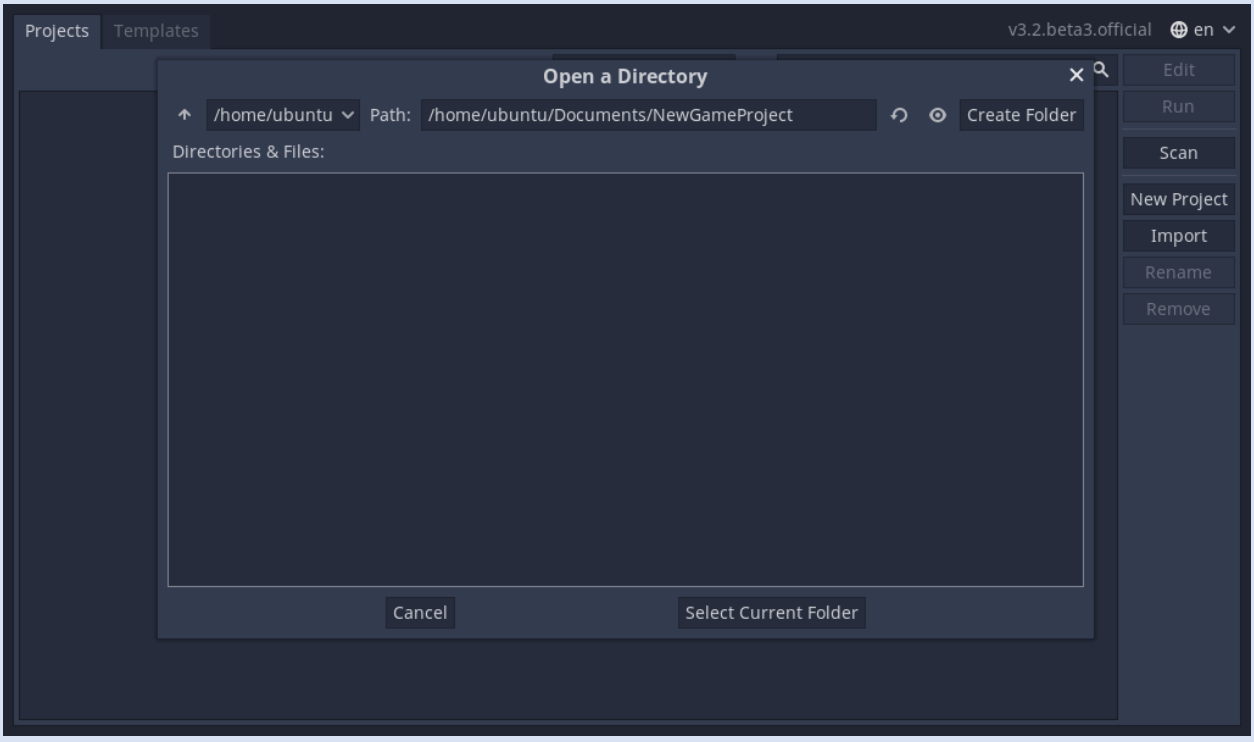


Create or import a project

To create a new project, click the New Project button on the right. Here you give it a name, choose an empty folder on your computer to save it to, and choose a renderer.



Click the Browse button to open Godot's file browser and pick a location or type the folder's path in the Project Path field.

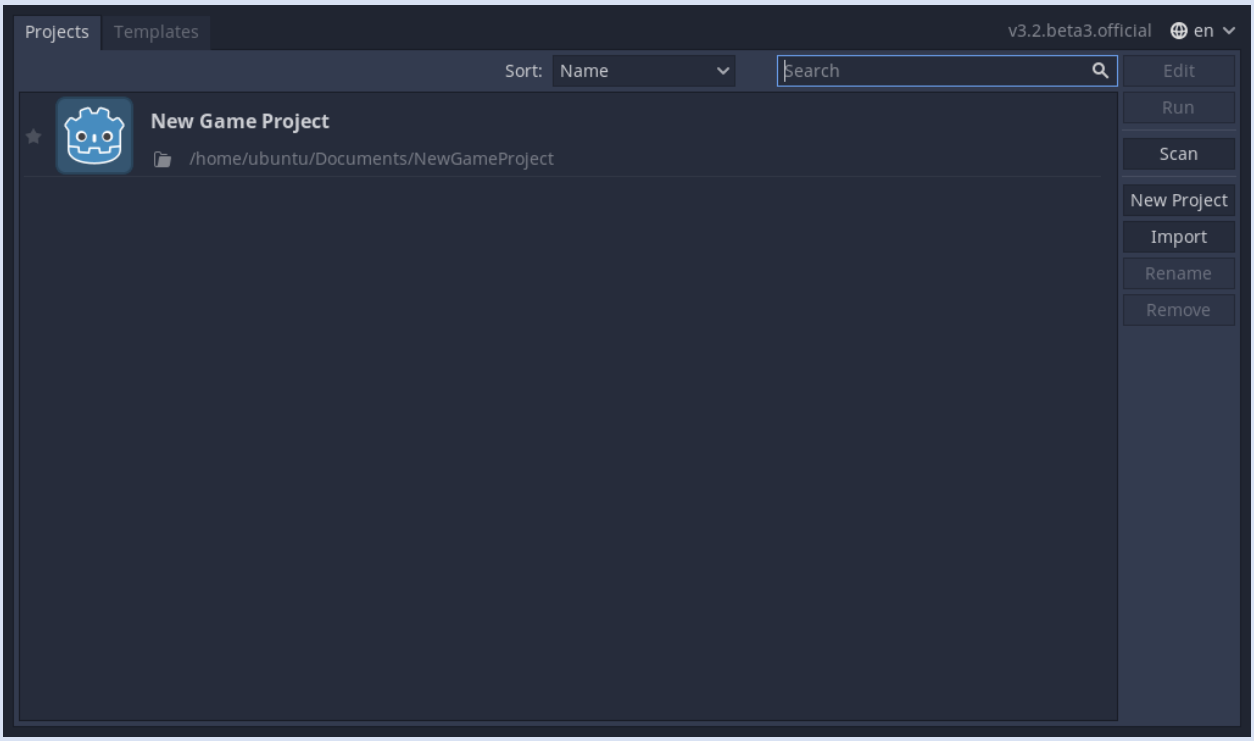


When you see the green tick on the right, it means the engine detects an empty folder. You can also click the Create Folder button next to your project name and an empty folder will be created with that name for the project.

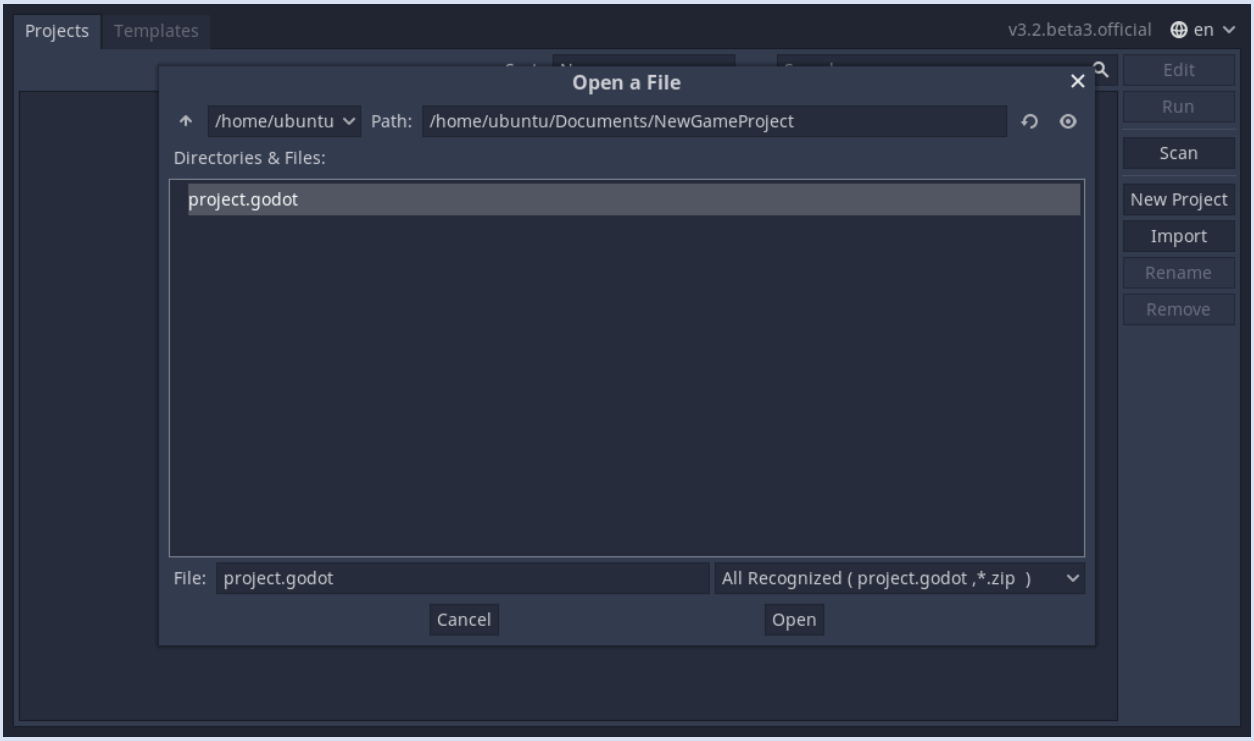
Finally, you need to choose which renderer to use (OpenGL ES 3.0 or OpenGL ES 2.0). The advantages and disadvantages of each are listed to help you choose, and you can refer to Differences between GLES2 and GLES3 for more details. Note that you can change the backend from the project settings if you change your mind later on. For this tutorial either backend is fine.

Once you are done click Create & Edit. Godot will create the project for you and open it in the editor.

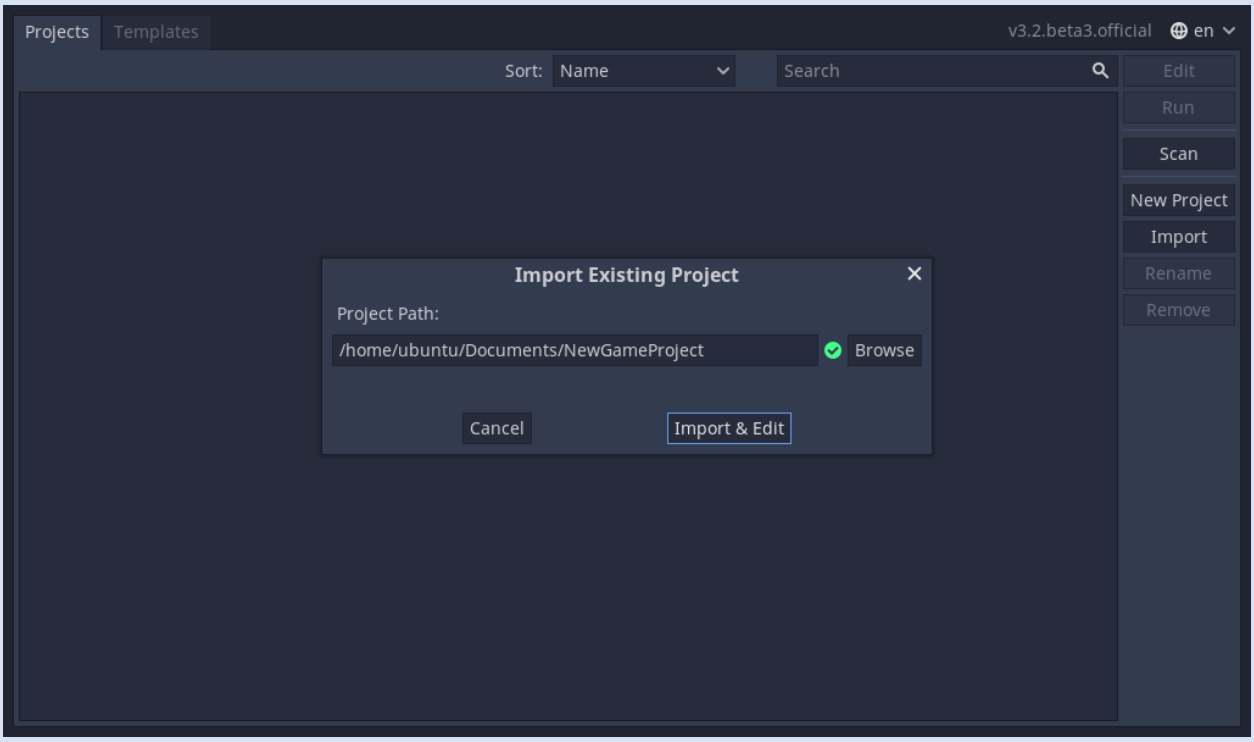
The next time you open the project manager, you'll see your new project in the list. Double click on it to open it in the editor.



You can import existing projects in a similar way, using the Import button. Locate the folder that contains the project or the project.godot file to import and edit it.

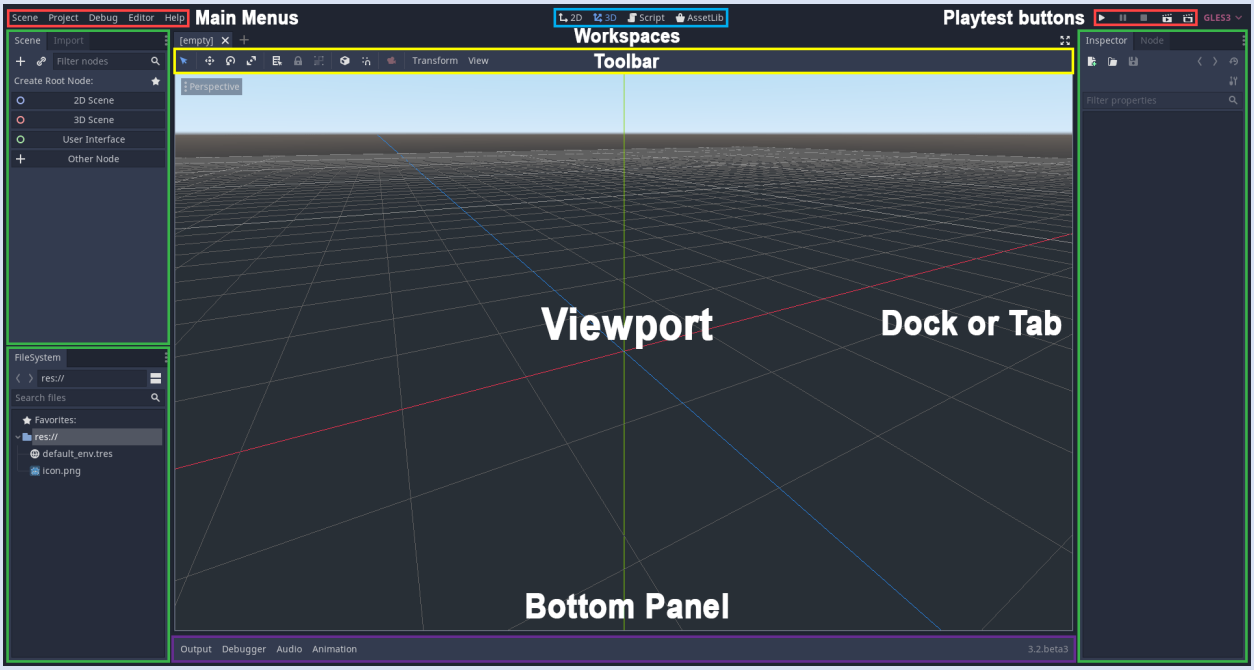


When the folder path is correct, you'll see a green checkmark.



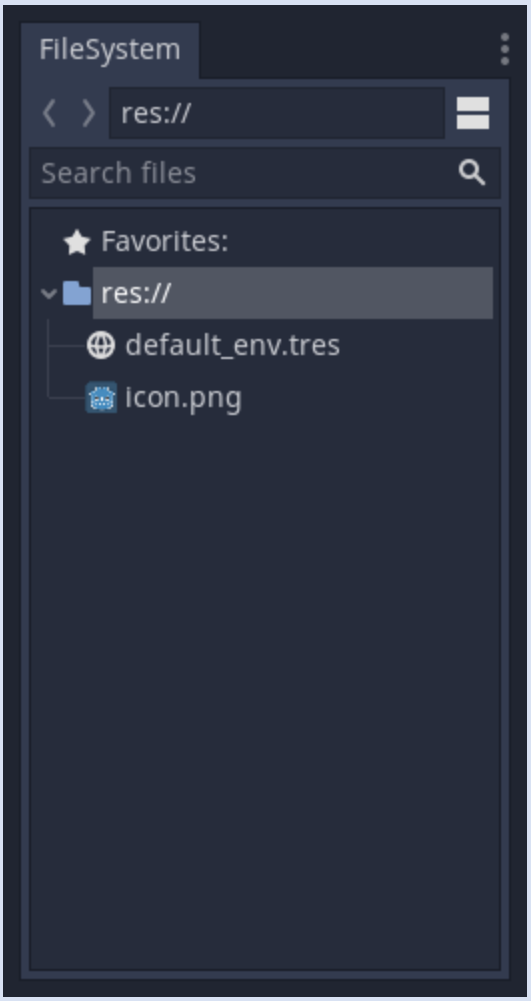
Your first look at Godot's editor

Welcome to Godot! With your project open, you should see the editor's interface with menus along the top of the interface and docks along the far extremes of the interface on either side of the viewport.

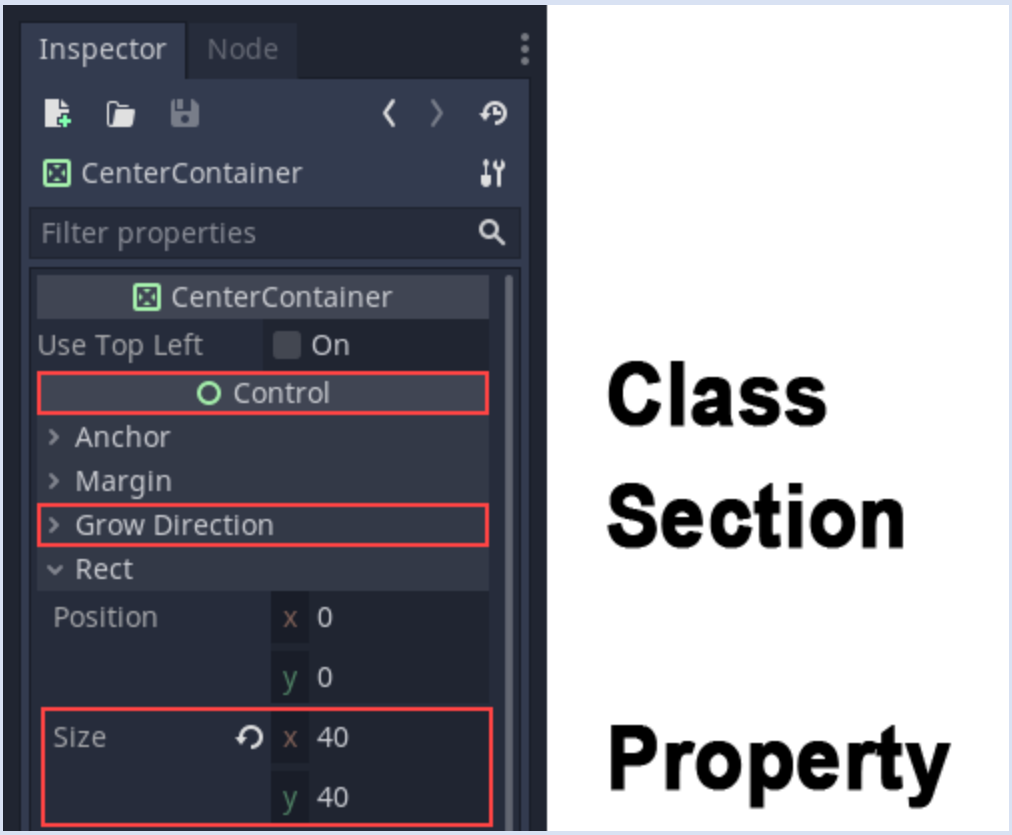


At the top, from left to right, you can see the main menus, the workspaces, and the playtest buttons.

The FileSystem dock is where you'll manage your project files and assets.

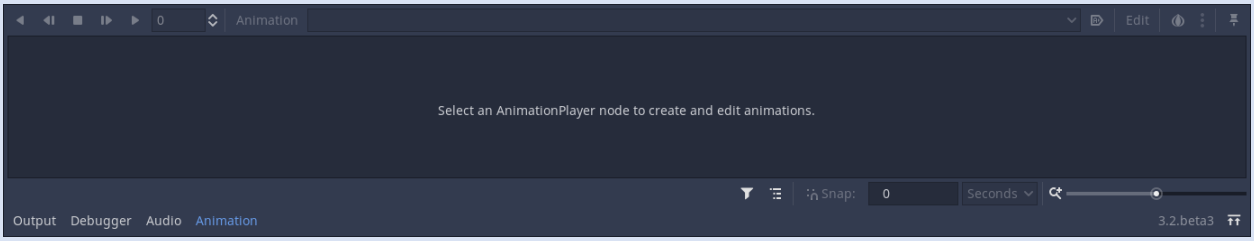


The Scene dock lists the active scene's content and the Inspector allows for the management of the properties of a scene's content.



In the center, you have the Toolbar at the top, where you'll find tools to move, scale or lock your scene's objects. It changes as you jump to different workspaces.

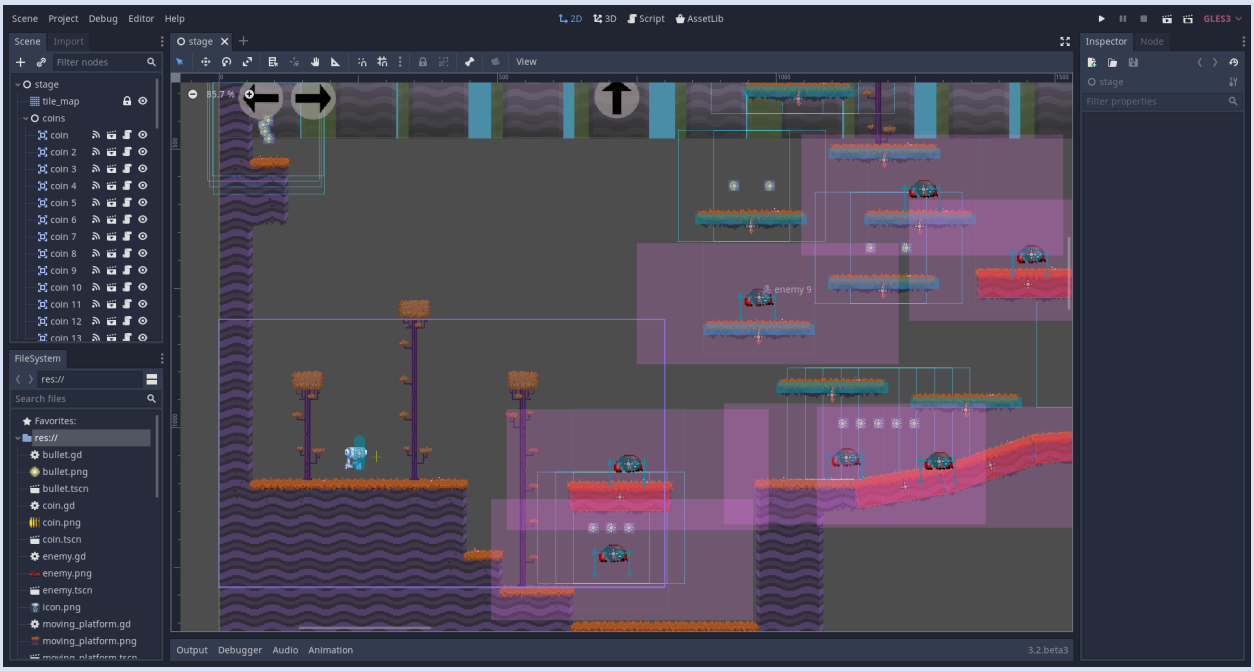
The Bottom Panel is the host for the debug console, the animation editor, the audio mixer… They are wide and can take precious space. That's why they're folded by default.



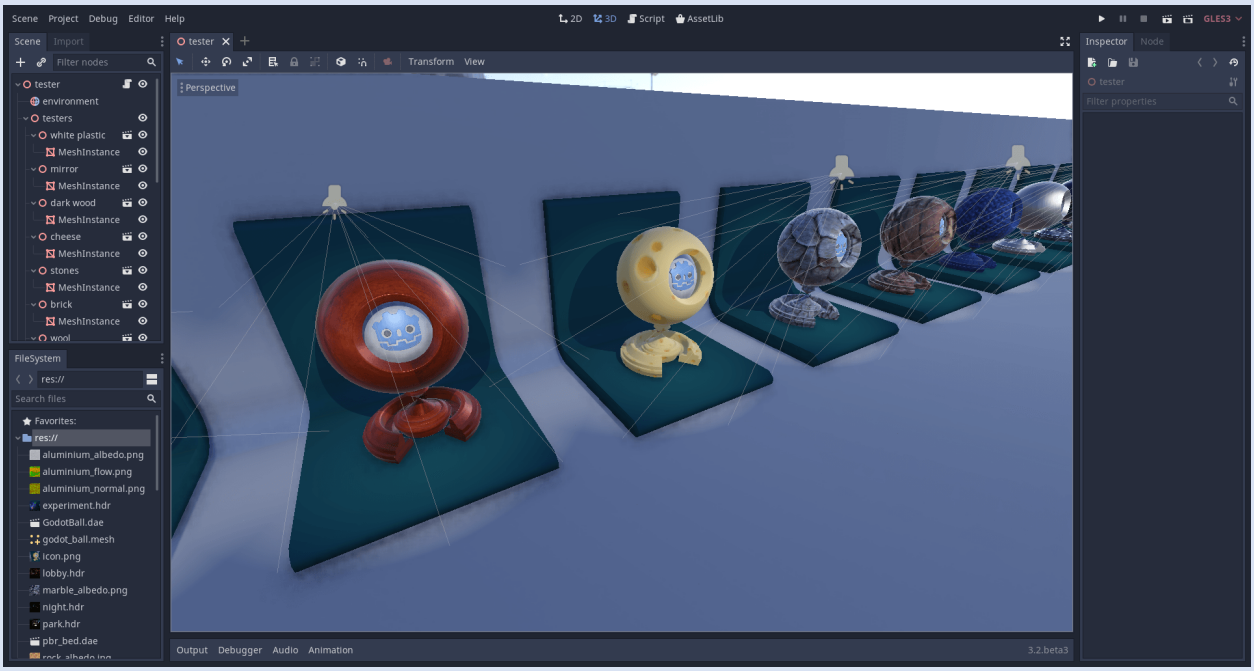
The workspaces

You can see four workspace buttons at the top: 2D, 3D, Script and AssetLib.

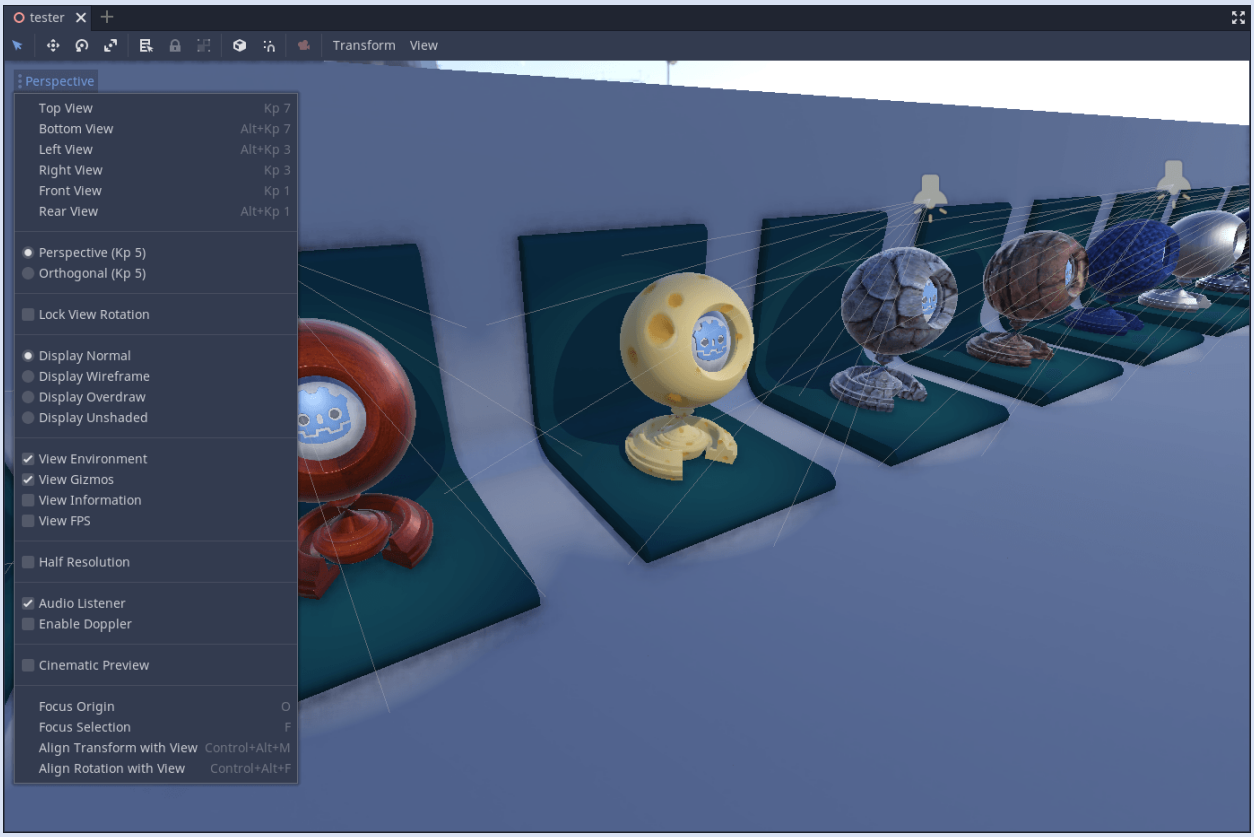
You'll use the 2D workspace for all types of games. In addition to 2D games, the 2D workspace is where you'll build your interfaces. Press F1 (or Alt + 1 on macOS) to access it.



In the 3D workspace, you can work with meshes, lights, and design levels for 3D games. Press F2 (or Alt + 2 on macOS) to access it.



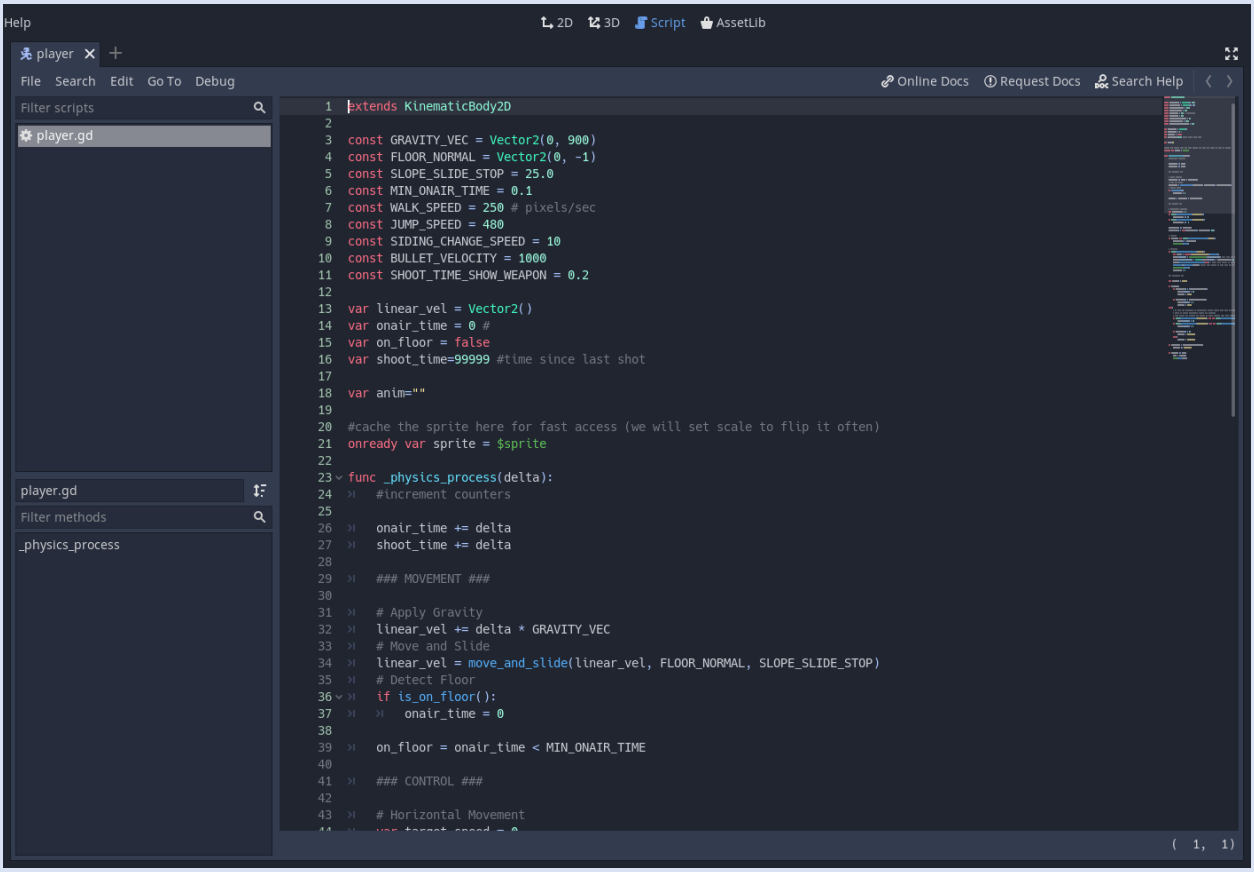
Notice the perspective button under the toolbar, it opens a list of options related to the 3D viewport.



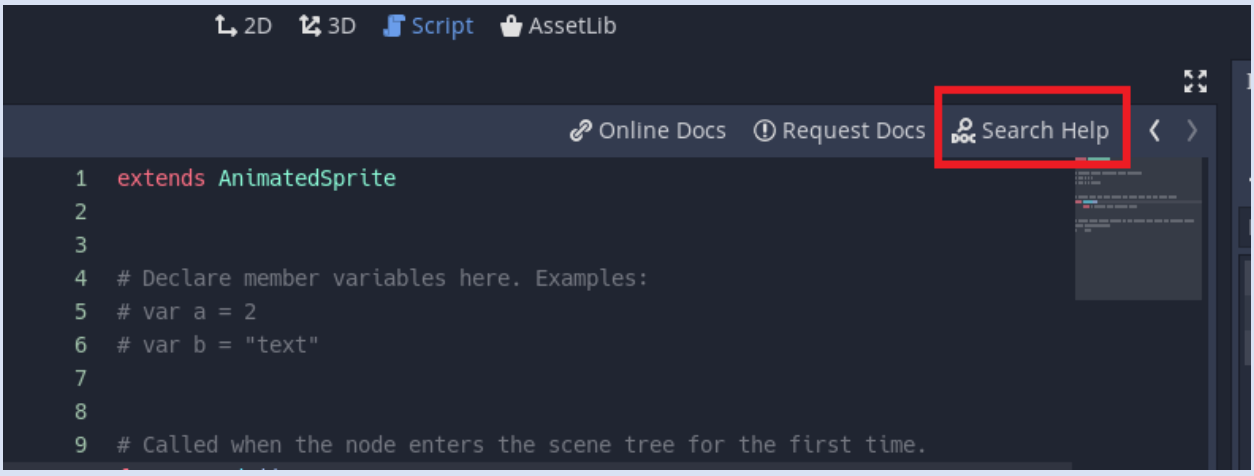
Note

Read Introduction to 3D for more detail about 3D workspace.

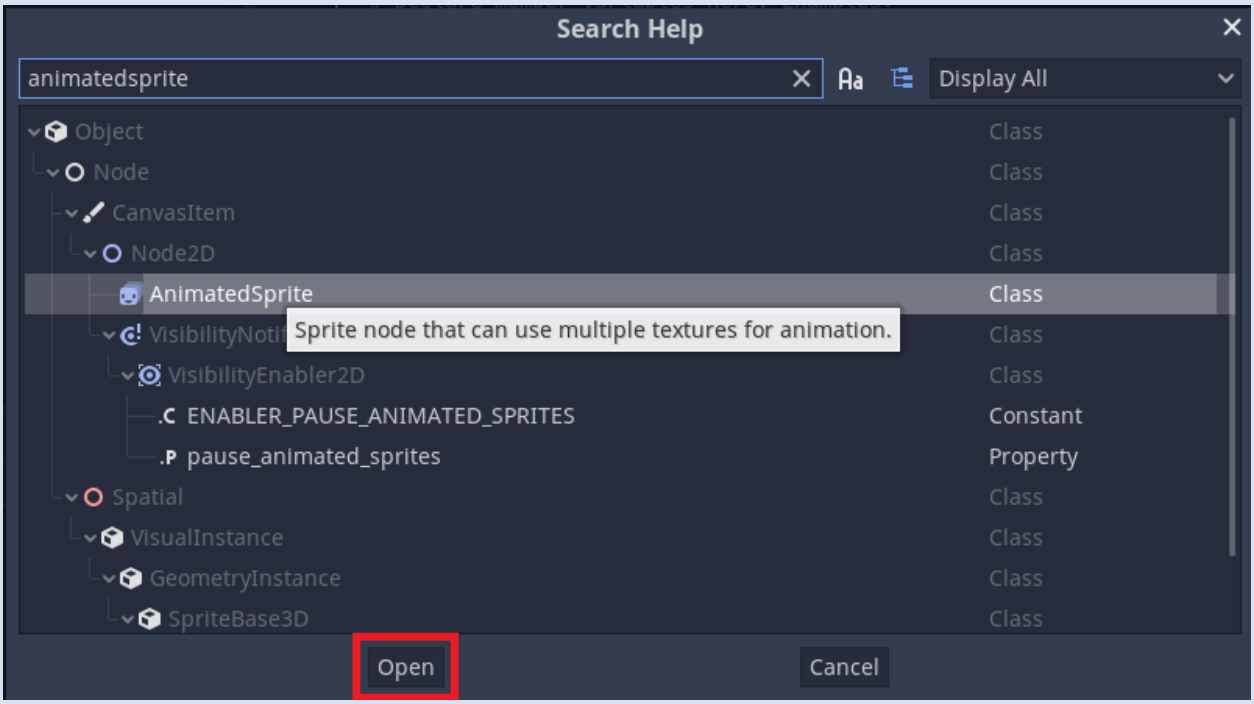
The Script workspace is a complete code editor with a debugger, rich auto-completion, and built-in code reference. Press F3 (or Alt + 3 on macOS) to access it, and Shift + F1 to search the reference.



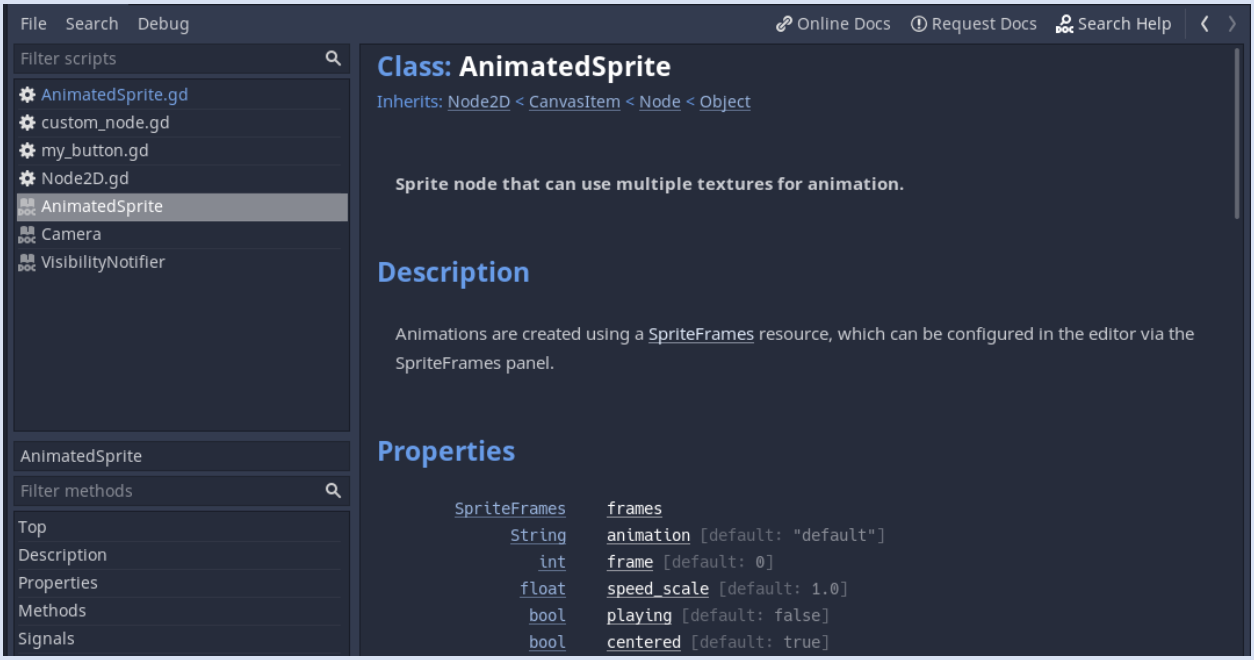
To search for information about a class, method, property, constant, or signal in the engine while you are writing a script, press the "Search Help" button at the top right of the Script workspace.



A new window will pop up. Search for the item that you want to find information about.



Click on the item you are looking for and press open. The documentation for the item will be displayed in the script workspace.



Finally, the AssetLib is a library of free and open source add-ons, scripts and assets to use in your projects.

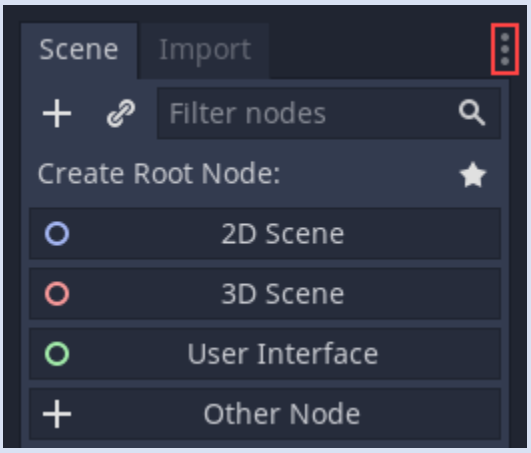
Modify the interface

Godot's interface lives in a single window. You cannot split it across multiple screens although you can work with an external code editor like Atom or Visual Studio Code for instance.

Move and resize docks

Click and drag on the edge of any dock or panel to resize it horizontally or vertically.

Click the three-dotted icon at the top of any dock to change its location.



Go to the Editor menu and Editor Settings to fine-tune the look and feel of the editor.

Scenes and nodes

Introduction

Imagine for a second that you are not a game developer anymore. Instead, you're a chef! Change your hipster outfit for a toque and a double breasted jacket. Now, instead of making games, you create new and delicious recipes for your guests.

So, how does a chef create a recipe? Recipes are divided into two sections: the first is the ingredients and the second is the instructions to prepare it. This way, anyone can follow the recipe and savor your magnificent creation.

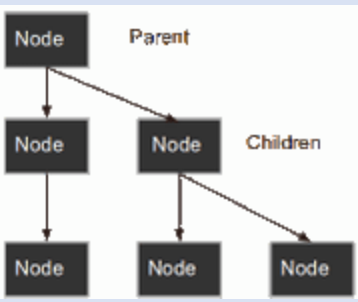
Making games in Godot feels pretty much the same way. Using the engine feels like being in a kitchen. In this kitchen, *nodes* are like a refrigerator full of fresh ingredients with which to cook.

There are many types of nodes. Some show images, others play sound, other nodes display 3D models, etc. There are dozens of them.

Nodes

But let's start with the basics. Nodes are fundamental building blocks for creating a game. As mentioned above, a node can perform a variety of specialized functions. However, any given node always has the following attributes:

- It has a name.
- It has editable properties.
- It can receive a callback to process every frame.
- It can be extended (to have more functions).
- It can be added to another node as a child.

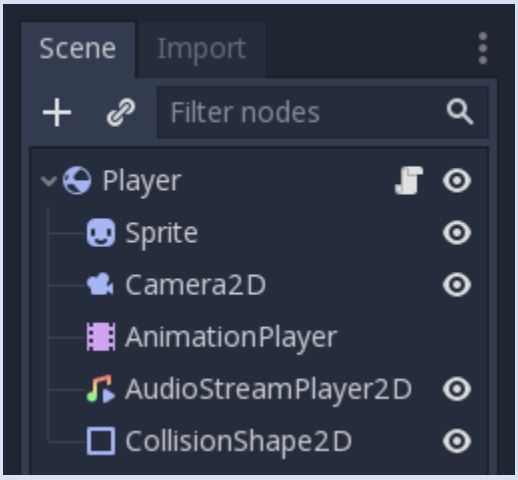


The last one is important. Nodes can have other nodes as children. When arranged in this way, the nodes become a tree.

In Godot, the ability to arrange nodes in this way creates a powerful tool for organizing projects. Since different nodes have different functions, combining them allows for the creation of more complex functions.

Don't worry if this doesn't click yet. We will continue to explore this over the next few sections. The most important fact to remember for now is that nodes exist and can be arranged this way.

Scenes



Now that the concept of nodes has been defined, the next logical step is to explain what a Scene is.

A scene is composed of a group of nodes organized hierarchically (in tree fashion). Furthermore, a scene:

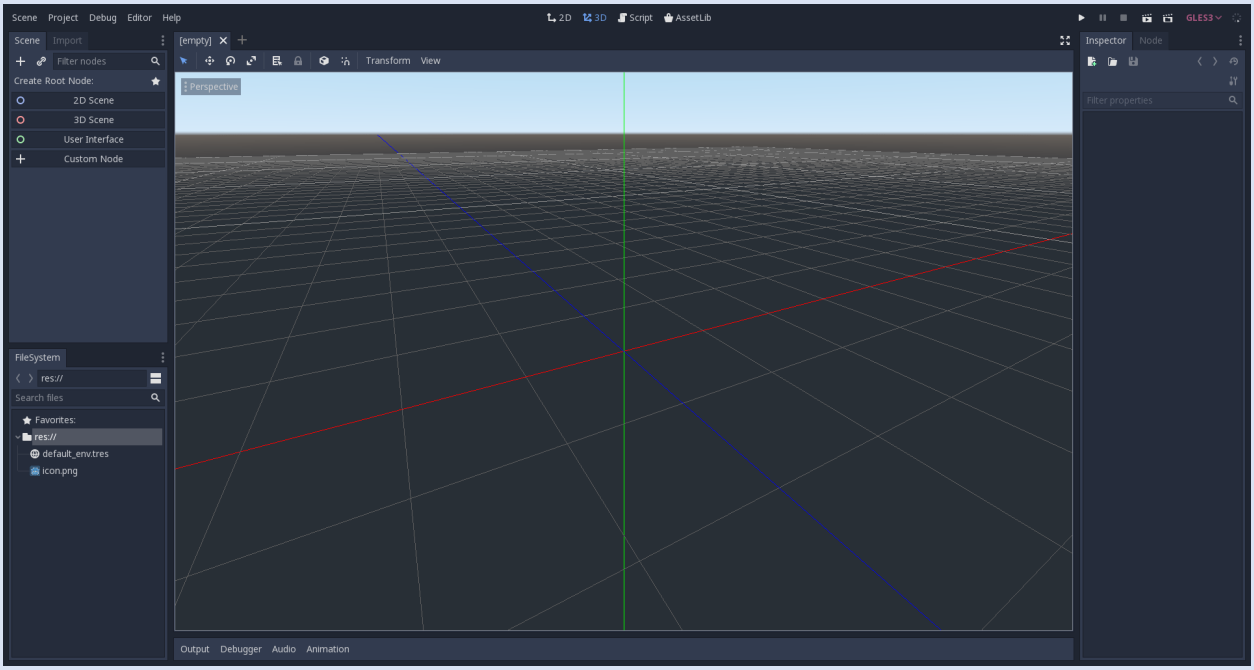
- always has one root node.
- can be saved to disk and loaded back.
- can be *instanced* (more on that later).

Running a game means running a scene. A project can contain several scenes, but for the game to start, one of them must be selected as the main scene.

Basically, the Godot editor is a scene editor. It has plenty of tools for editing 2D and 3D scenes as well as user interfaces, but the editor is based on the concept of editing a scene and the nodes that compose it.

Editor

Open the project you made in Introduction to Godot's editor, or create a new one. This will open the Godot editor:



As mentioned before, making games in Godot feels like being in a kitchen, so let's open the refrigerator and add some fresh nodes to the project. We'll begin with a "Hello World" message that we'll put on the screen.

To do this we need to add a Label node. Press the "Add Child Node" button at the top left of the scene dock (the icon represents a plus symbol). This button is the main way to add new nodes to a scene, and will always add the chosen node as a child of the currently selected node (or, in an empty scene, as the "root" node).

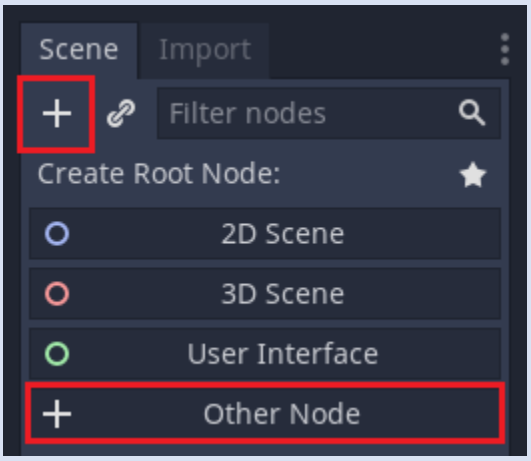
Note

In an empty scene (without root node), the scene dock shows several options to quickly add a root node to the scene. "2D Scene" adds a Node2D node, "3D Scene" adds a Spatial node, "User Interface" adds a Control node, and "Other Node" which lets you select any node (so it is

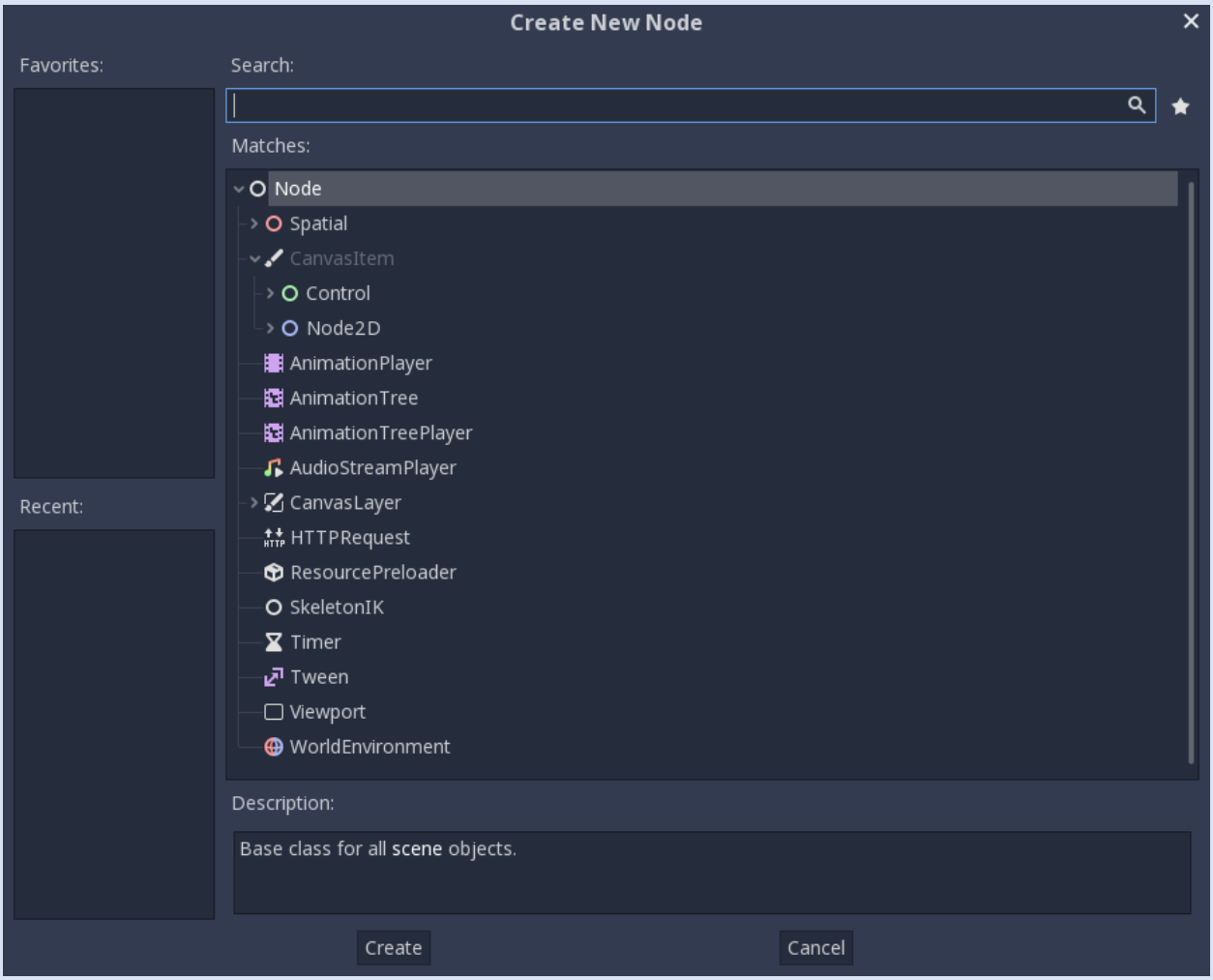
equivalent to pressing the "Add Child Node" button). You can also press the star-shaped icon to toggle the display of your favorited nodes.

Note that these presets are here for convenience and are not mandatory for the different types of scenes. Not every 3D scene needs a Spatial node as its root node, likewise not every GUI or 2D scene needs a Control node or Node2D as their root node.

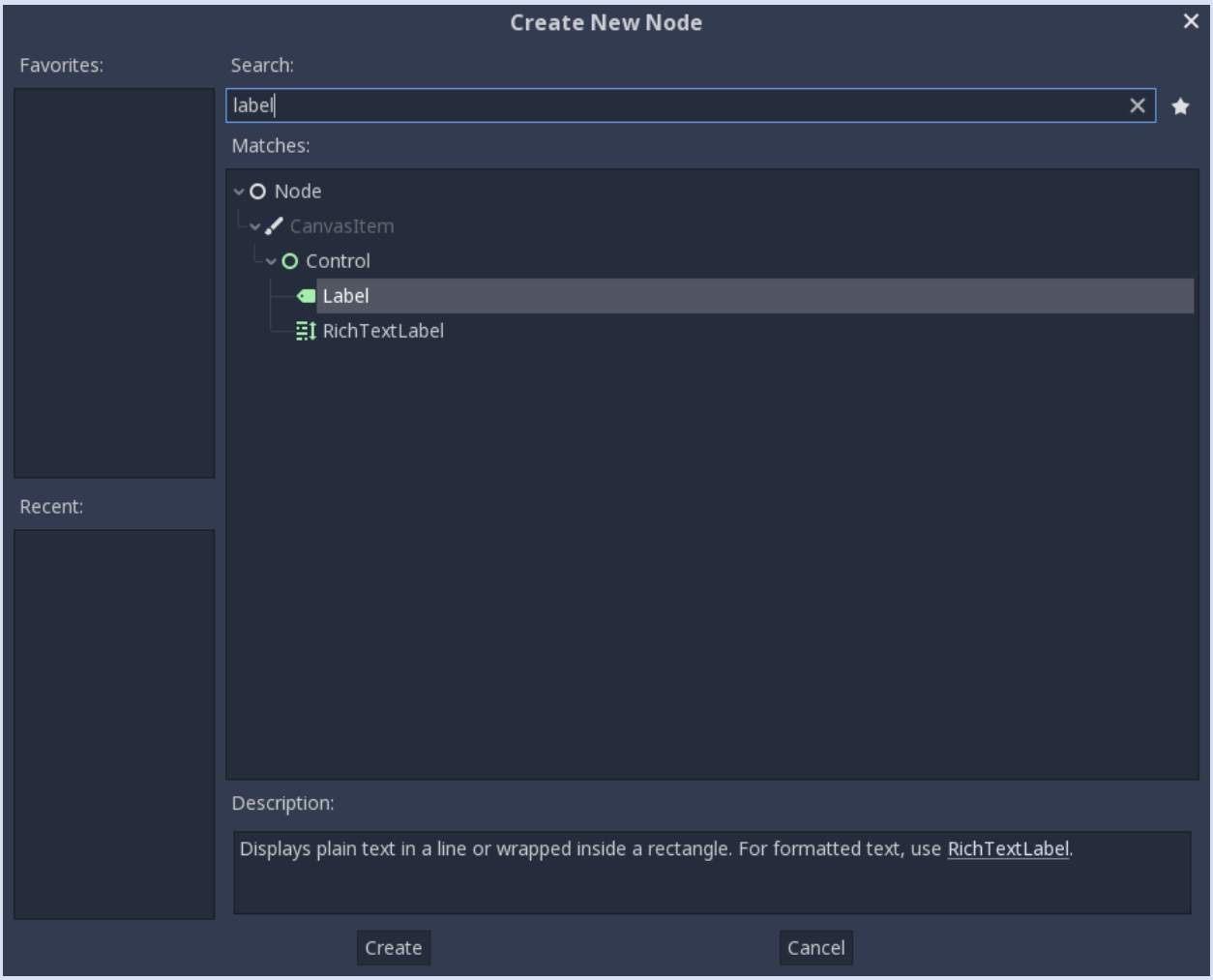
Now, to add a label node to this scene you can click on the Other Node button or the Add Node button at the top. In scenes that aren't empty you use the add node button to create every child node.



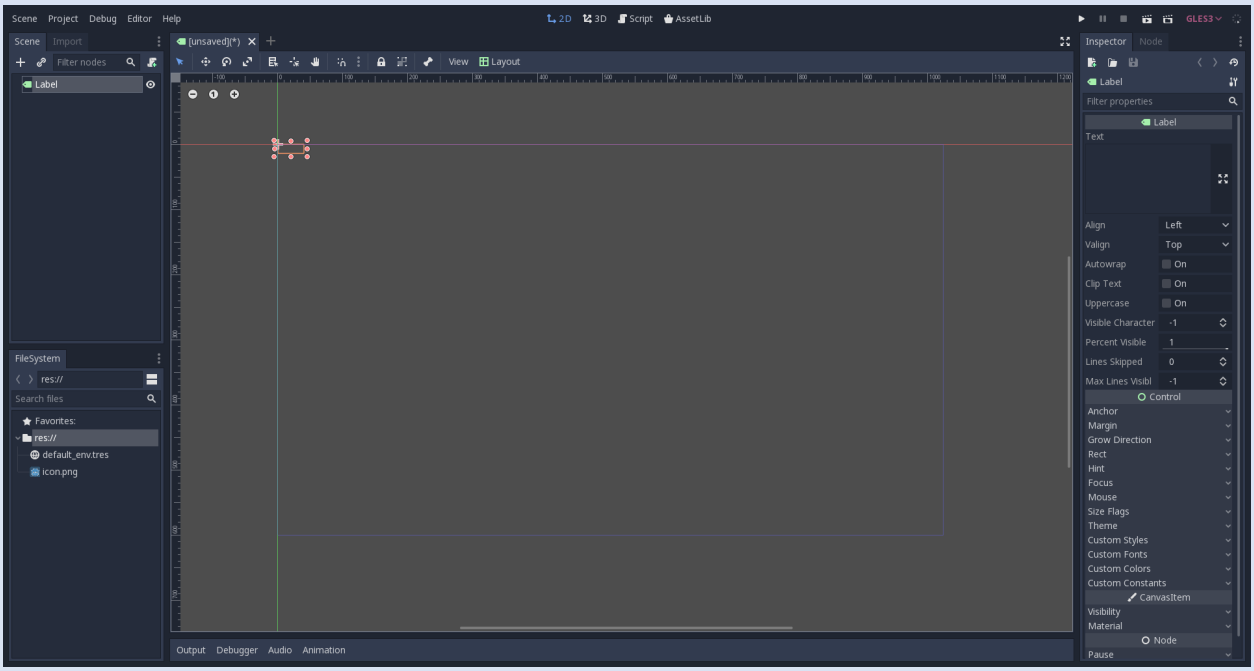
This will open the Create Node dialog, showing the long list of nodes that can be created:



From there, select the "Label" node first. Searching for it is probably the fastest way:



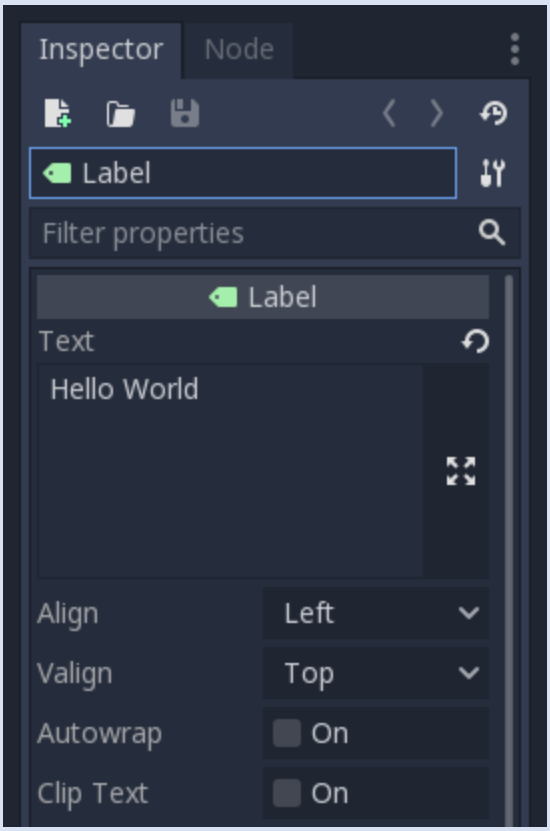
And finally, create the Label! A lot happens when Create is pressed:



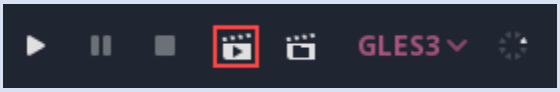
First of all, the scene changes to the 2D editor (because Label is a 2D Node type), and the Label appears, selected, at the top left corner of the viewport.

The node appears in the scene tree editor in the Scene dock, and the label properties appear in the Inspector dock.

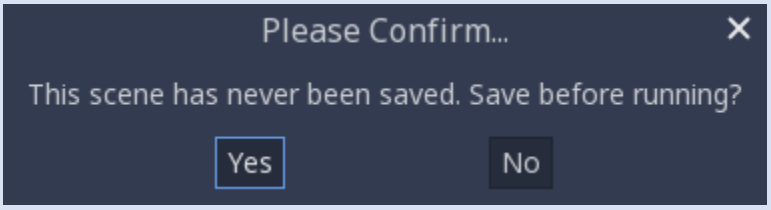
The next step will be to change the "Text" Property of the label. Let's change it to "Hello World":



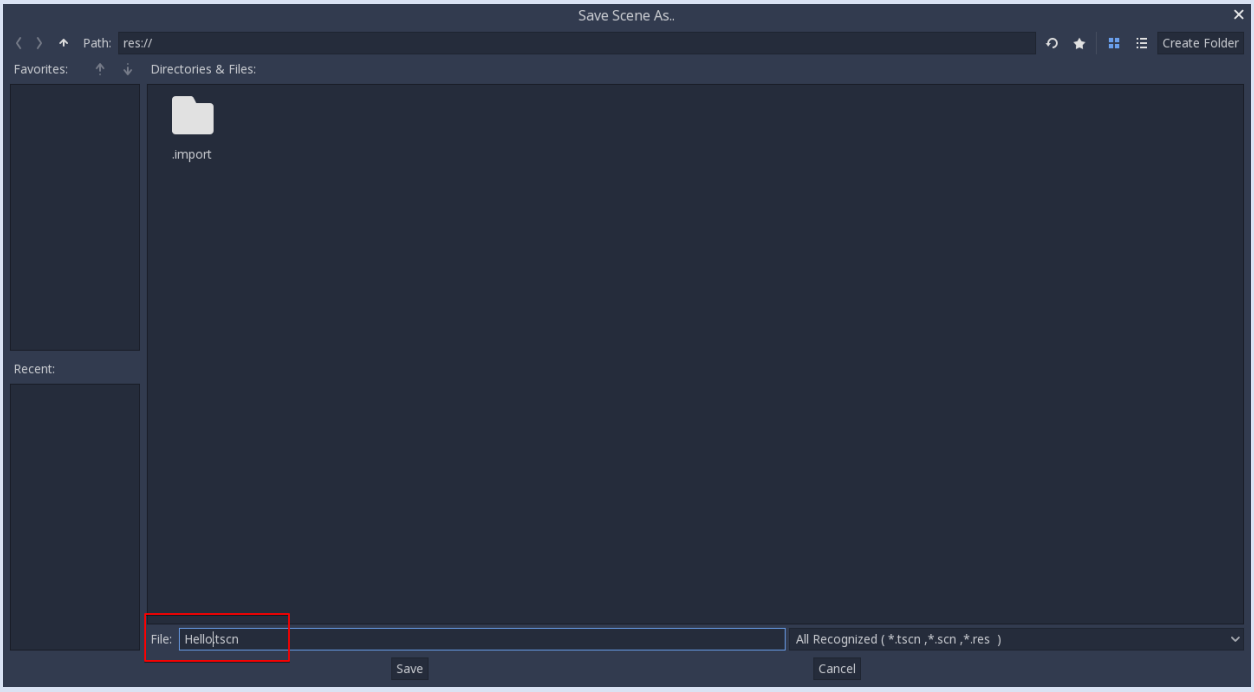
Ok, everything's ready to run the scene! Press the PLAY SCENE Button on the top bar (or hit F6):



Aaaand... Oops.

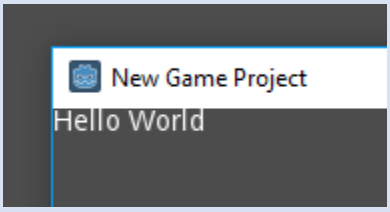


Scenes need to be saved to be run, so save the scene to something like Hello.tscn in Scene -> Save:



And here's when something funny happens. The file dialog is a special file dialog, and only allows you to save inside the project. The project root is res:// which means "resource path". This means that files can only be saved inside the project. For the future, when doing file operations in Godot, remember that res:// is the resource path, and no matter the platform or install location, it is the way to locate where resource files are from inside the game.

After saving the scene and pressing run scene again, the "Hello World" demo should finally execute:



Success!

Note

If this doesn't immediately work and you have a hiDPI display on at least one of your monitors, go to Project → Project Settings → Display → Window then enable Allow Hidpi under Dpi.

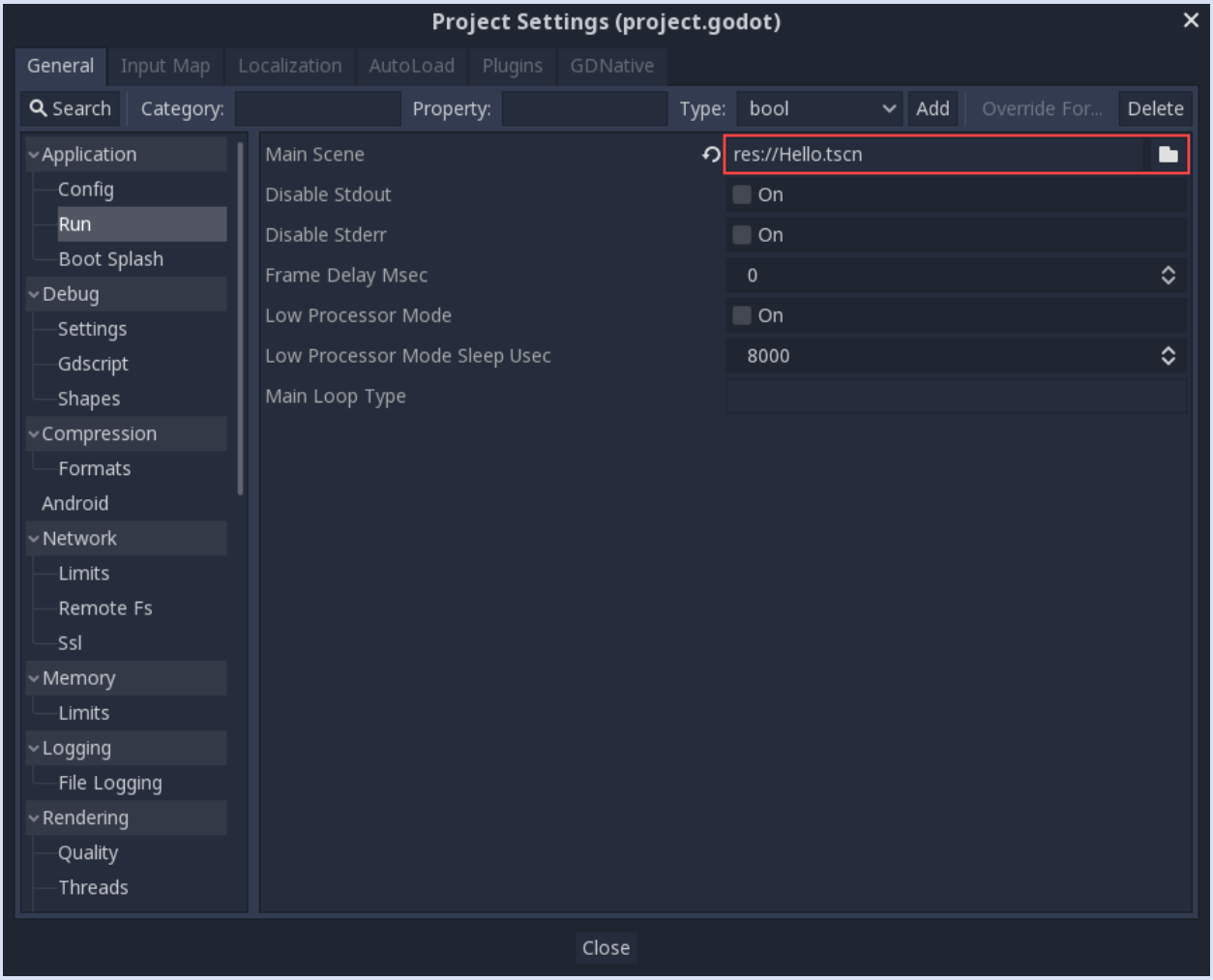
Configuring the project

Ok, it's time to configure the project. Right now, the only way to run something is to execute the current scene. Projects, however, may have several scenes, so one of them must be set as the main scene. This is the scene that will be loaded any time the project is run.

These settings are all stored in a project.godot file, which is a plaintext file in win.ini format (for easy editing). There are dozens of settings that you can change in this file to alter how a project executes. To simplify this process, Godot provides a project settings dialog, which acts as a sort of frontend to editing a project.godot file.

To access that dialog, select Project -> Project Settings. Try it now.

Once the window opens, let's select a main scene. Locate the *Application/Run/Main Scene* property and click on it to select 'Hello.tscn'.



Now, with this change, when you press the regular Play button (or F5), this scene will run, no matter which scene is actively being edited.


The project settings dialog provides a lot of options that can be saved to a project.godot file and shows their default values. If you change a value, a tick is marked to the left of its name. This means that the property will be saved to the project.godot file and remembered.

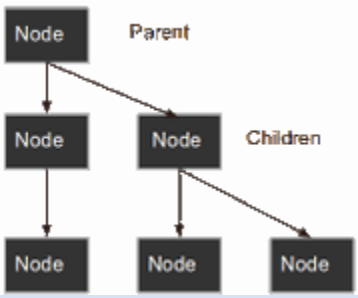
As a side note, it is also possible to add custom configuration options and read them in at run-time using the ProjectSettings singleton.

Instancing

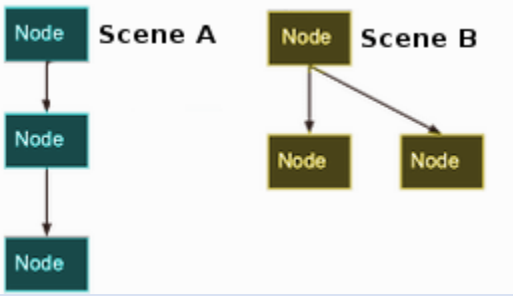
Introduction

Creating a single scene and adding nodes into it might work for small projects, but as a project grows in size and complexity, the number of nodes can quickly become unmanageable. To address this, Godot allows a project to be separated into any number of scenes. This provides you with a powerful tool that helps you organize the different components of your game.

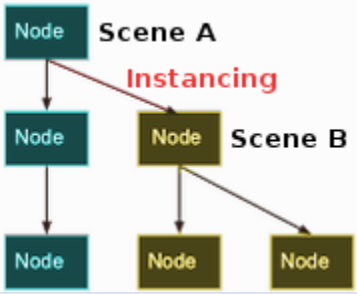
In Scenes and nodes you learned that a scene is a collection of nodes organized in a tree structure, with a single node as the tree root.



You can create as many scenes as you like and save them to disk. Scenes saved in this manner are called "Packed Scenes" and have a .tscn filename extension.



Once a scene has been saved, it can be instanced into another scene as if it were any other node.

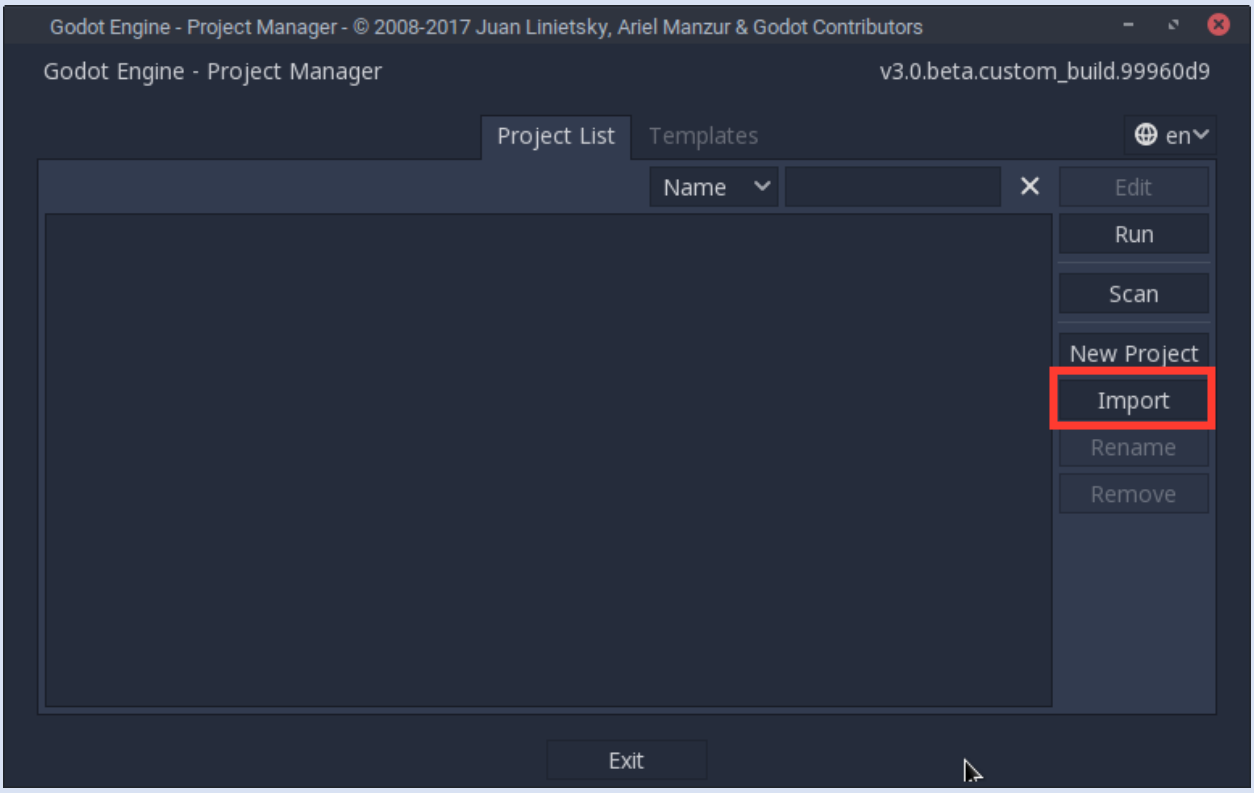


In the above picture, Scene B was added to Scene A as an instance.

Instancing by example

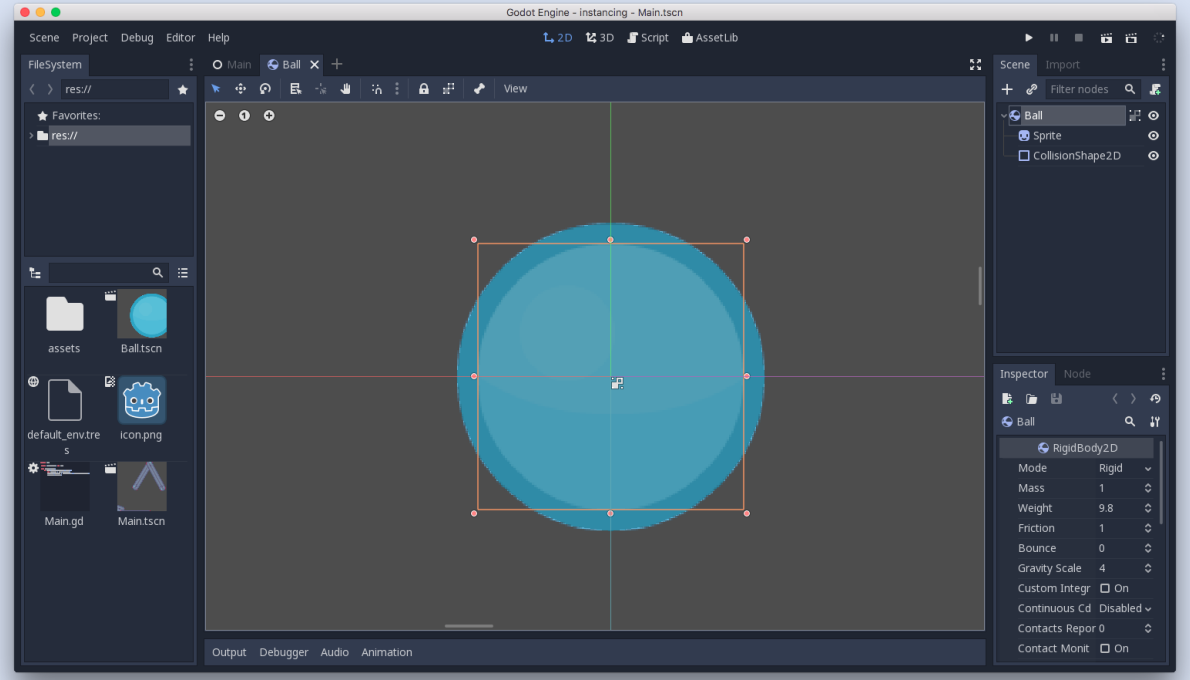
To learn how instancing works, let's start by downloading a sample project: instancing.zip.

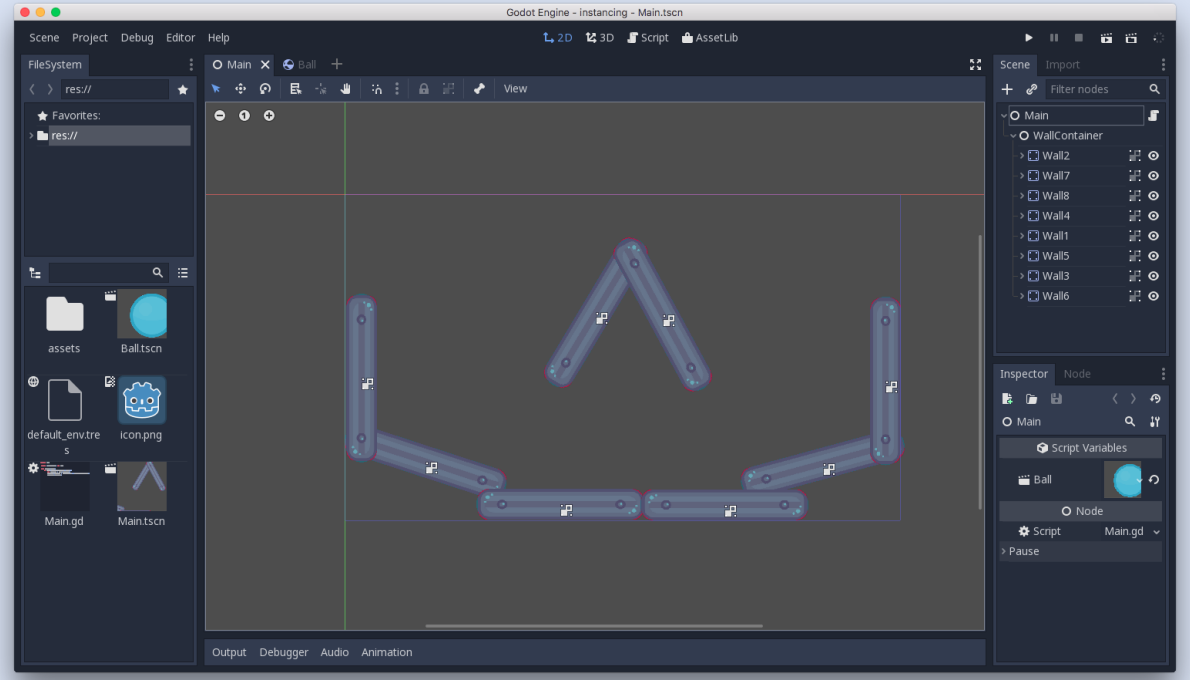
Unzip this project anywhere you like. Then open Godot and add this project to the project manager using the 'Import' button:



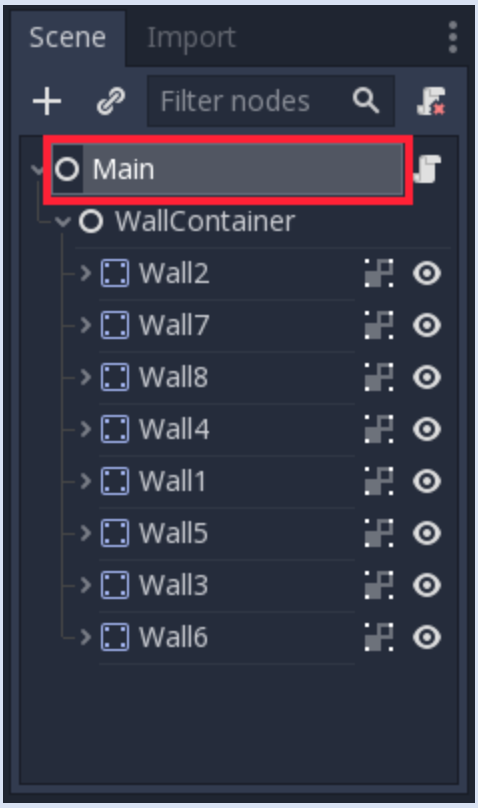
Browse to the folder you extracted and open the "project.godot" file you can find inside it. After doing this, the new project will appear on the list of projects. Edit the project by pressing the 'Edit' button.

This project contains two scenes: "Ball.tscn" and "Main.tscn". The ball scene uses a RigidBody2D to provide physics behavior while the main scene has a set of obstacles for the ball to collide with (using StaticBody2D).

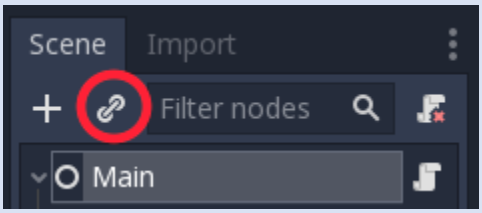




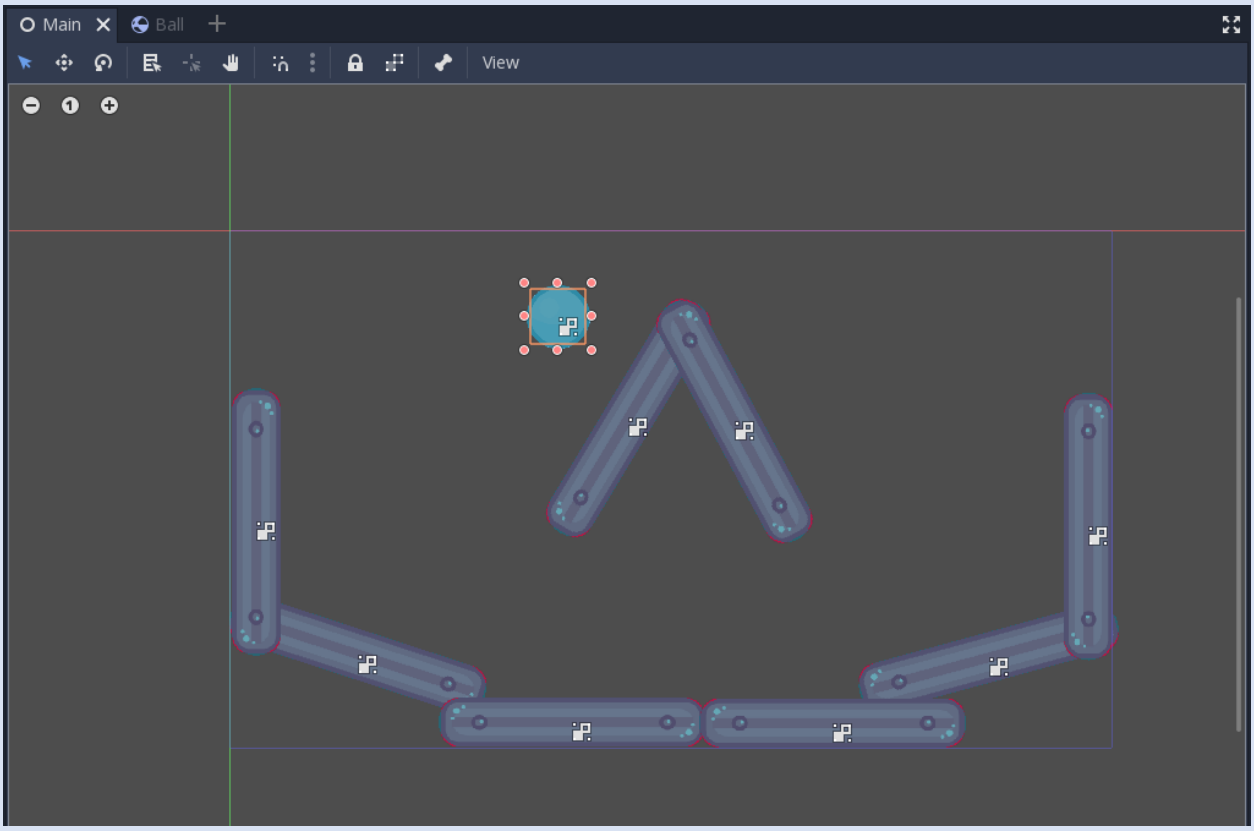
Open the Main scene, and then select the root node:



We want to add an instance of the Ball scene as a child of Main. Click the "link"-shaped button (its hover-text says "Instance a scene file as a Node.") and select the Ball.tscn file.



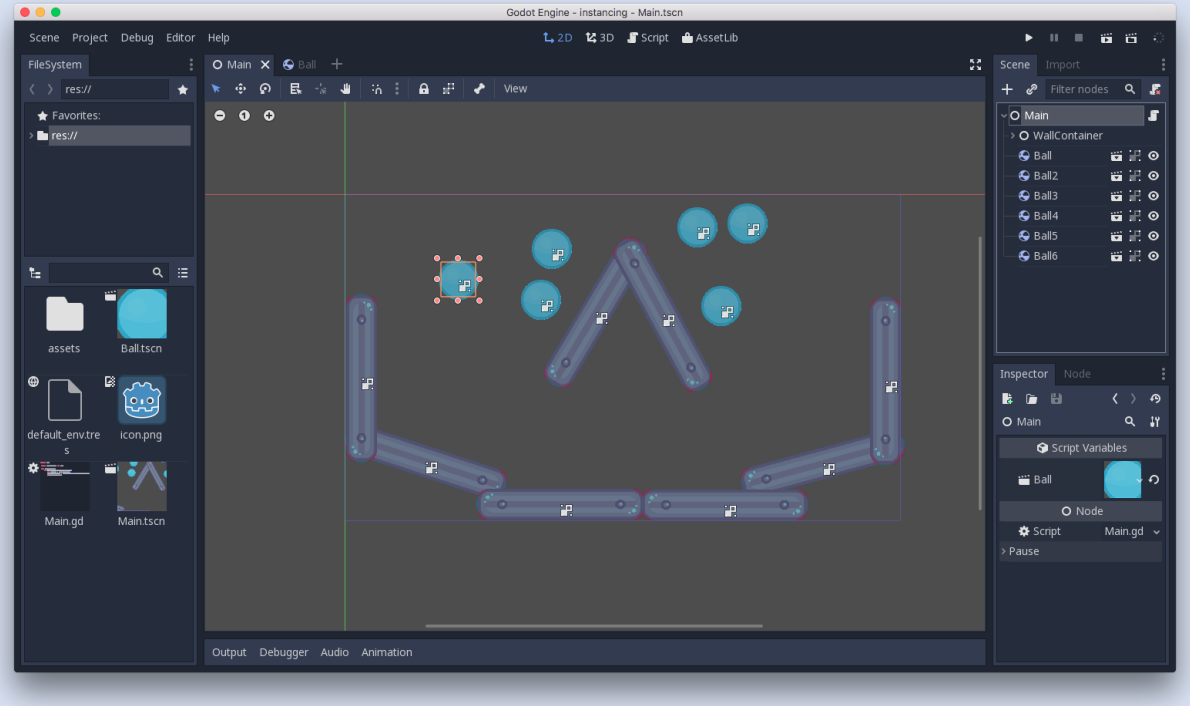
The ball will be placed at the top-left corner of the screen area (this is (0, 0) in screen coordinates). Click and drag the ball somewhere near the top-center of the scene:



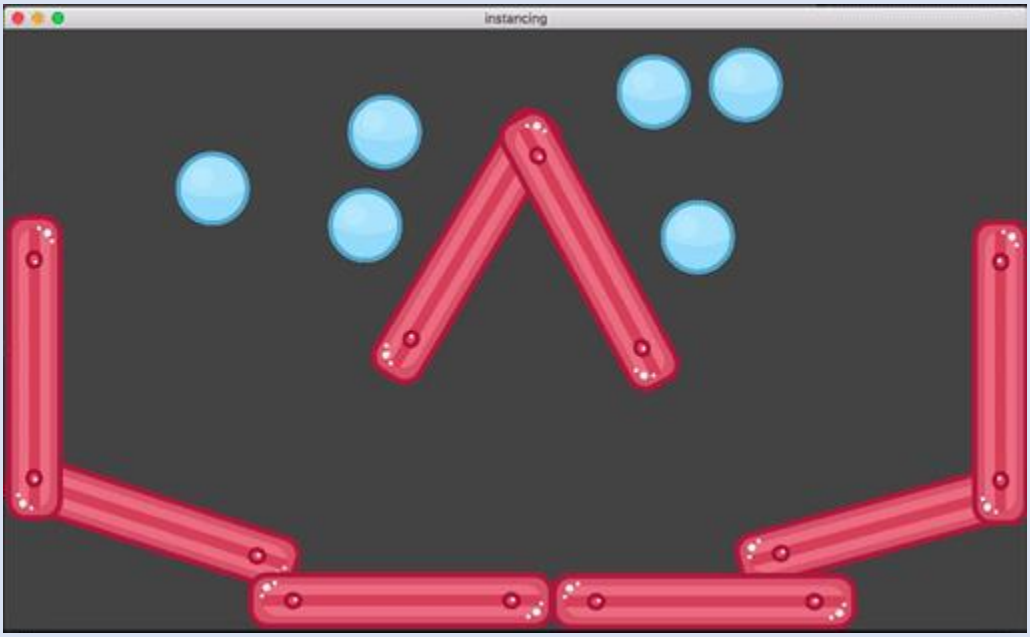
Press "Play" and watch the ball fall to the bottom of the screen:

Multiple instances

You can add as many instances as you like to a scene, either by using the "Instance" button again, or by clicking on the ball instance and pressing Ctrl + D (Cmd + D on macOS) to duplicate it:

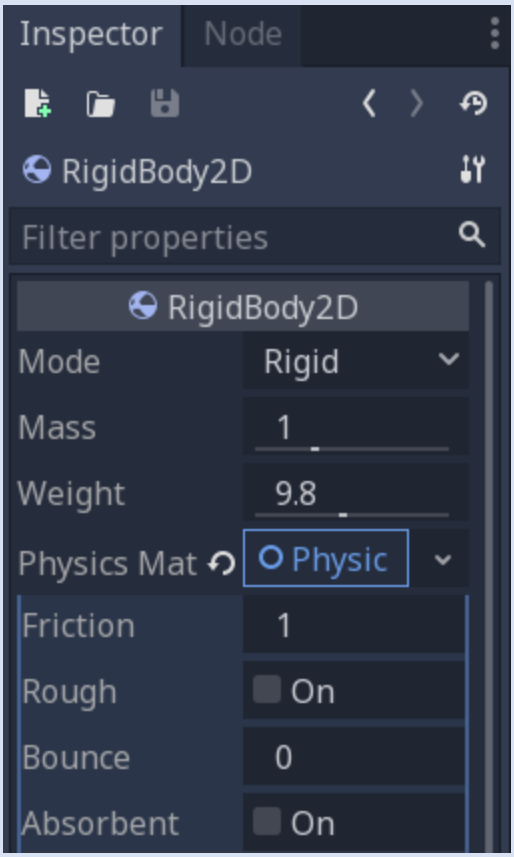


Run the scene again and all of the balls will fall.



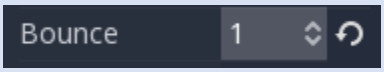
Editing instances

Open the Ball scene, expand the PhysicsMaterial by clicking on it, and set the Bounce property to 1.



Press "Play" and notice that all of the instanced balls are now much more bouncy. Because the instanced balls are based on the saved scene, changes to that scene will affect all instances.

You can also adjust individual instances. Set the bounce value back to 0 and then in the Main scene, select one of the instanced balls. Resources like PhysicsMaterial are shared between instances by default, so we need to make it unique. Click on the tools button in the top-right of the Inspector dock and select "Make Sub-Resources Unique". Set its Bounce to 1 and press "Play".



Notice that a grey "revert" button appears next to the adjusted property. When this button is present, it means you modified a property in the instanced scene to override its value in the saved scene. Even if that property is modified in the original scene, the custom value will remain. Pressing the revert button will restore the property to the value in the saved scene.

Recap

Instancing has many handy uses. At a glance, with instancing you have:

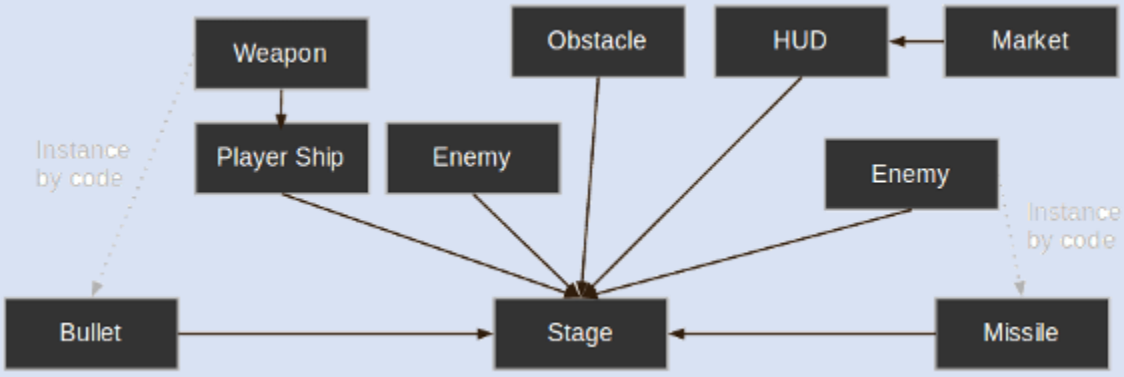
- The ability to subdivide scenes and make them easier to manage.
- A tool to manage and edit multiple node instances at once.
- A way to organize and embed complex game flows or even UIs (in Godot, UI Elements are nodes, too).

Design language

But the greatest strength that comes with instancing scenes is that it works as an excellent design language. This distinguishes Godot from all the other engines out there. Godot was designed from the ground up around this concept.

When making games with Godot, the recommended approach is to dismiss most common design patterns, such as MVC or Entity-Relationship diagrams, and instead think about your scenes in a more natural way. Start by imagining the visible elements in your game, the ones that can be named not just by a programmer, but by anyone.

For example, here's how a simple shooter game could be imagined:

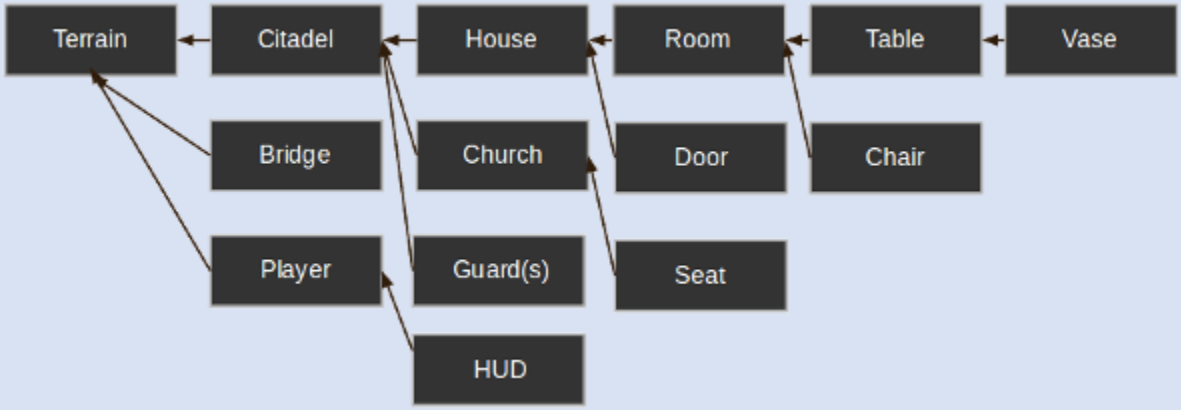


You can come up with a diagram like this for almost any kind of game. Write down the parts of the game that you can visualize, and then add arrows to represent ownership of one component by another.

Once you have a diagram like this, the recommended process for making a game is to create a scene for each element listed in the diagram. You'll use instancing (either by code or directly in the editor) for the ownership relationships.

A lot of time spent in programming games (or software in general) is on designing an architecture and fitting game components to that architecture. Designing based on scenes replaces that approach and makes development much faster and more straightforward, allowing you to concentrate on the game logic itself. Because most game components map directly to a scene, using a design based on scene instantiation means little other architectural code is needed.

Let's take a look at one more, somewhat more complex, example of an open-world type game with lots of assets and nested elements:



Take a look at the room element. Let's say we started there. We could make a couple of different room scenes, with different arrangements of furniture (also scenes) in them. Later, we could make a house scene, connecting rooms to make up its interior.

Then, we could make a citadel scene, which is made out of many instanced houses. Then, we could start working on the world map terrain, adding the citadel onto it.

Later, we could create scenes that represent guards (and other NPCs) and add them to the citadel as well. As a result, they would be indirectly added to the overall game world.

With Godot, it's easy to iterate on your game like this, as all you need to do is create and instance more scenes. Furthermore, the editor UI is designed to be user friendly for programmers and non-programmers alike. A typical team development process can involve 2D or 3D artists, level designers, game designers, and animators, all working with the editor interface.

Information overload!

This has been a lot of high level information dropped on you all at once. However, the important part of this tutorial was to create an awareness of how scenes and instancing are used in real projects.

Scripting

Introduction

Before Godot 3.0, the only choice for scripting a game was to use GDScript. Nowadays, Godot has four (yes, four!) official languages and the ability to add extra scripting languages dynamically!

This is great, mostly due to the large amount of flexibility provided, but it also makes our work supporting languages more difficult.

The "main" languages in Godot, though, are GDScript and VisualScript. The main reason to choose them is their level of integration with Godot, as this makes the experience smoother; both have slick editor integration, while C# and C++ need to be edited in a separate IDE. If you are a big fan of statically typed languages, go with C# and C++ instead.

GDScript

GDScript is, as mentioned above, the main language used in Godot. Using it has some positive points compared to other languages due to its high integration with Godot:

- It's simple, elegant, and designed to be familiar for users of other languages such as Lua, Python, Squirrel, etc.
- Loads and compiles blazingly fast.
- The editor integration is a pleasure to work with, with code completion for nodes, signals, and many other items pertaining to the scene being edited.
- Has vector types built-in (such as Vectors, transforms, etc.), making it efficient for heavy use of linear algebra.
- Supports multiple threads as efficiently as statically typed languages - one of the limitations that made us avoid VMs such as Lua, Squirrel, etc.
- Uses no garbage collector, so it trades a small bit of automation (most objects are reference counted anyway), by determinism.
- Its dynamic nature makes it easy to optimize sections of code in C++ (via GDNative) if more performance is required, all without recompiling the engine.

If you're undecided and have experience with programming, especially dynamically typed languages, go for GDScript!

VisualScript

Beginning with 3.0, Godot offers Visual Scripting. This is a typical implementation of a "blocks and connections" language, but adapted to how Godot works.

Visual scripting is a great tool for non-programmers, or even for experienced developers who want to make parts of the code more accessible to others, like game designers or artists.

It can also be used by programmers to build state machines or custom visual node workflows - for example, a dialogue system.

.NET / C#

As Microsoft's C# is a favorite amongst game developers, we have added official support for it. C# is a mature language with tons of code written for it, and support was added thanks to a generous donation from Microsoft.

It has an excellent tradeoff between performance and ease of use, although one must be aware of its garbage collector.

Since Godot uses the Mono .NET runtime, in theory any third-party .NET library or framework can be used for scripting in Godot, as well as any Common Language Infrastructure-compliant programming language, such as F#, Boo or ClojureCLR. In practice however, C# is the only officially supported .NET option.

GDNative / C++

Finally, one of our brightest additions for the 3.0 release: GDNative allows scripting in C++ without needing to recompile (or even restart) Godot.

Any C++ version can be used, and mixing compiler brands and versions for the generated shared libraries works perfectly, thanks to our use of an internal C API Bridge.

This language is the best choice for performance and does not need to be used throughout an entire game, as other parts can be written in GDScript or Visual Script. However, the API is clear and easy to use as it resembles, mostly, Godot's actual C++ API.

More languages can be made available through the GDNative interface, but keep in mind we don't have official support for them.

Scripting a scene

For the rest of this tutorial we'll set up a GUI scene consisting of a button and a label, where pressing the button will update the label. This will demonstrate:

- Writing a script and attaching it to a node.
- Hooking up UI elements via signals.
- Writing a script that can access other nodes in the scene.

Before continuing, make sure to skim and bookmark the GDScript reference. It's a language designed to be simple, and the reference is structured into sections to make it easier to get an overview of the concepts.

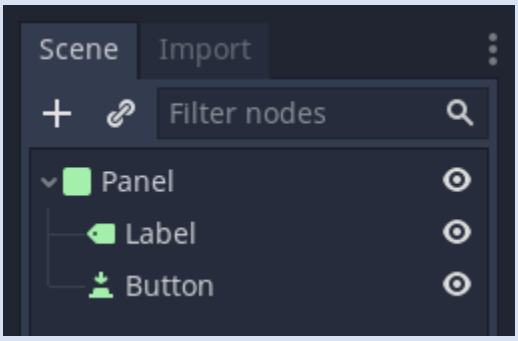
Scene setup

If you still have the "instancing" project open from the previous tutorial, then close that out (Project -> Quit to Project List) and create a New Project.

Use the "Add Child Node" dialogue accessed from the Scene tab (or by pressing Ctrl + A) to create a hierarchy with the following nodes:

- Panel
  - Label
  - Button

The scene tree should look like this:

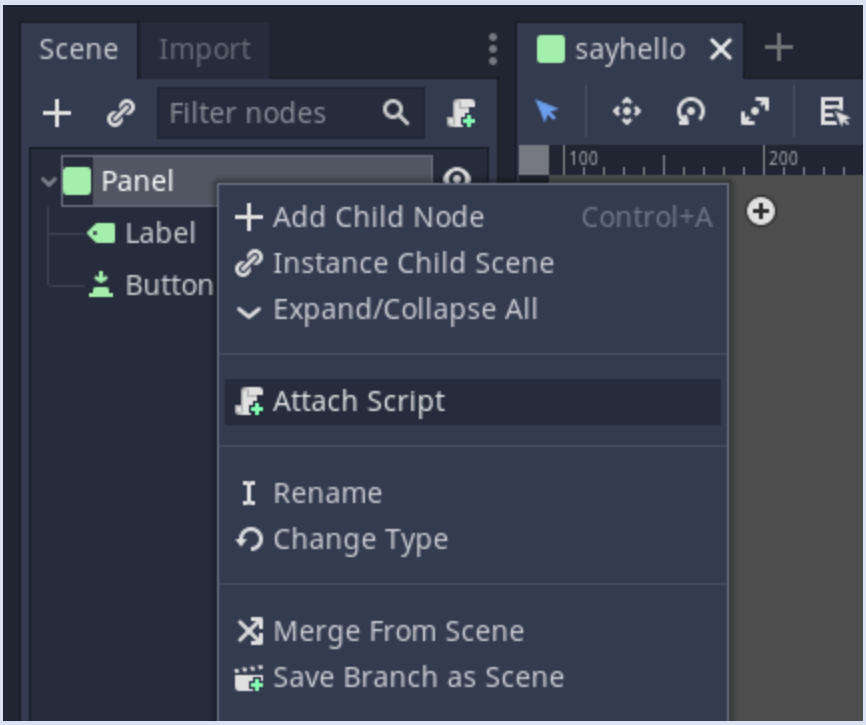


Use the 2D editor to position and resize the Button and Label so that they look like the image below. You can set the text from the Inspector tab.

Finally, save the scene with a name such as sayhello.tscn.

Adding a script

Right click on the Panel node, then select "Attach Script" from the context menu:

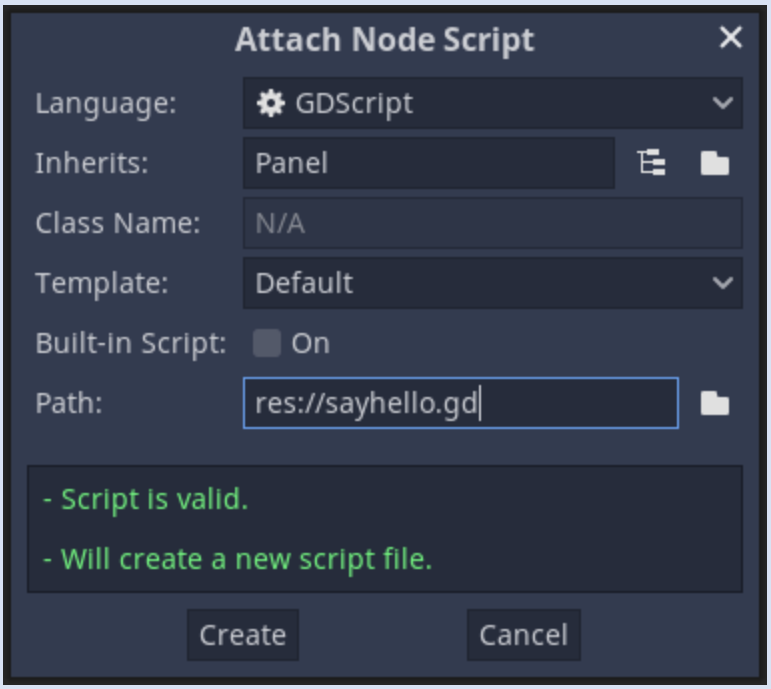


The script creation dialog will pop up. This dialog allows you to set the script's language, class name, and other relevant options.

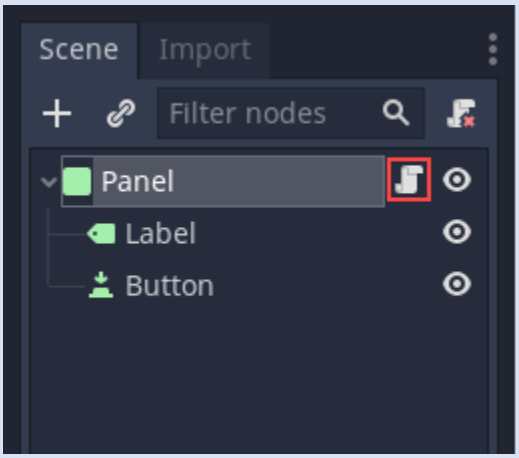
In GDScript, the file itself represents the class, so the class name field is not editable.

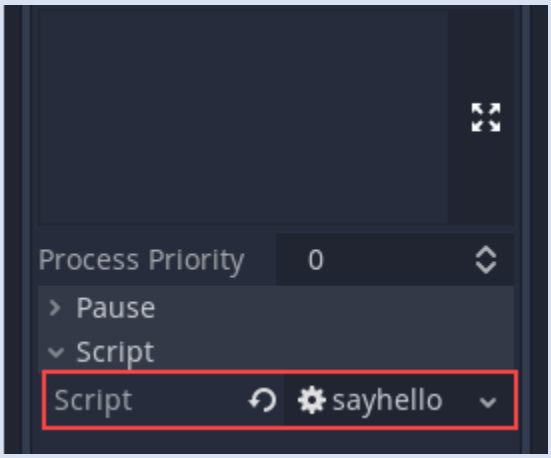
The node we're attaching the script to is a panel, so the Inherits field will automatically be filled in with "Panel". This is what we want, as the script's goal is to extend the functionality of our panel node.

Finally, enter a path name for the script and select Create:

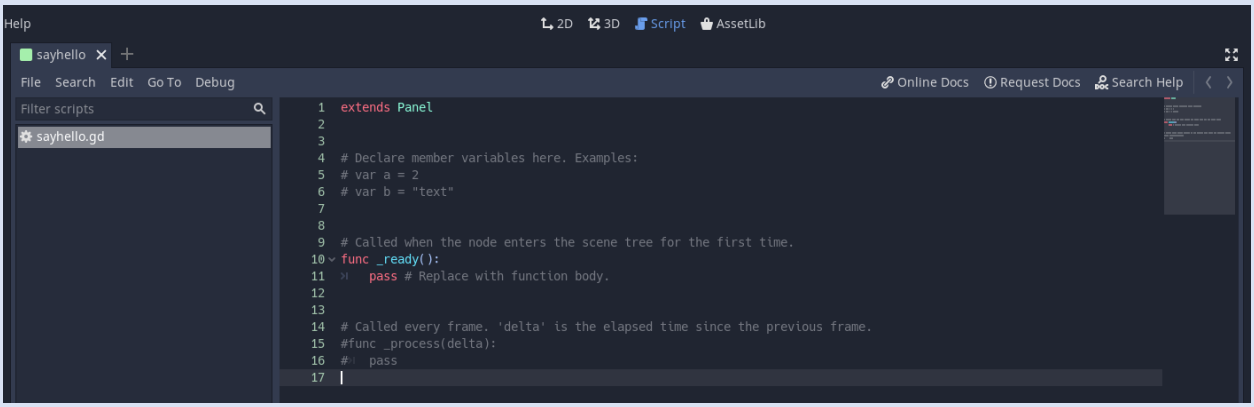


The script will then be created and added to the node. You can see this as an "Open script" icon next to the node in the Scene tab, as well as in the script property under Inspector:





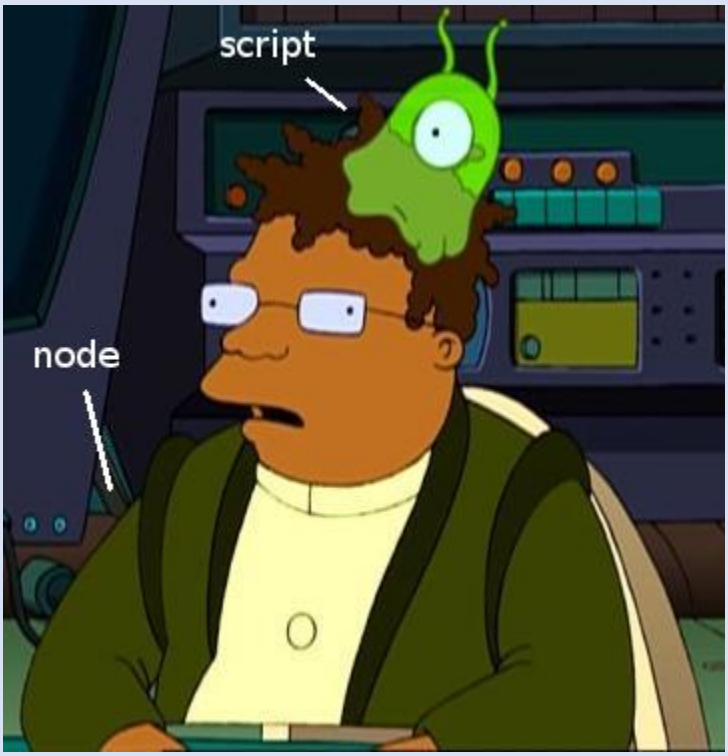
To edit the script, select either of these buttons, both of which are highlighted in the above image. This will bring you to the script editor, where a default template will be included:



There's not much there. The _ready() function is called when the node, and all its children, enters the active scene. Note: _ready() is not the constructor; the constructor is instead _init().

The role of the script

A script adds behavior to a node. It is used to control how the node functions as well as how it interacts with other nodes: children, parent, siblings, and so on. The local scope of the script is the node. In other words, the script inherits the functions provided by that node.



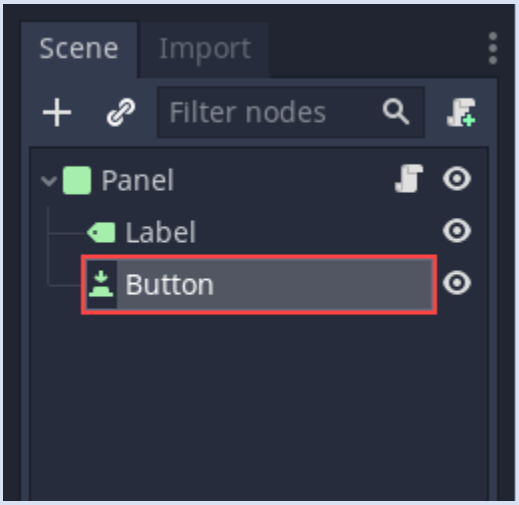
Handling a signal

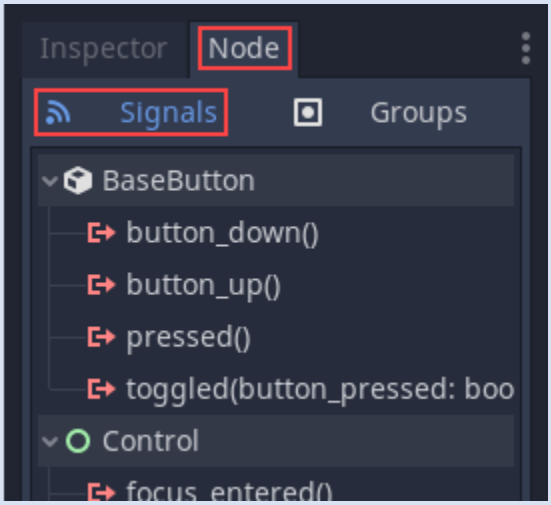
Signals are "emitted" when some specific kind of action happens, and they can be connected to any function of any script instance. Signals are used mostly in GUI nodes, although other nodes have them too, and you can even define custom signals in your own scripts.

In this step, we'll connect the "pressed" signal to a custom function. Forming connections is the first part and defining the custom function is the second part. For the first part, Godot provides two ways to create connections: through a visual interface the editor provides or through code.

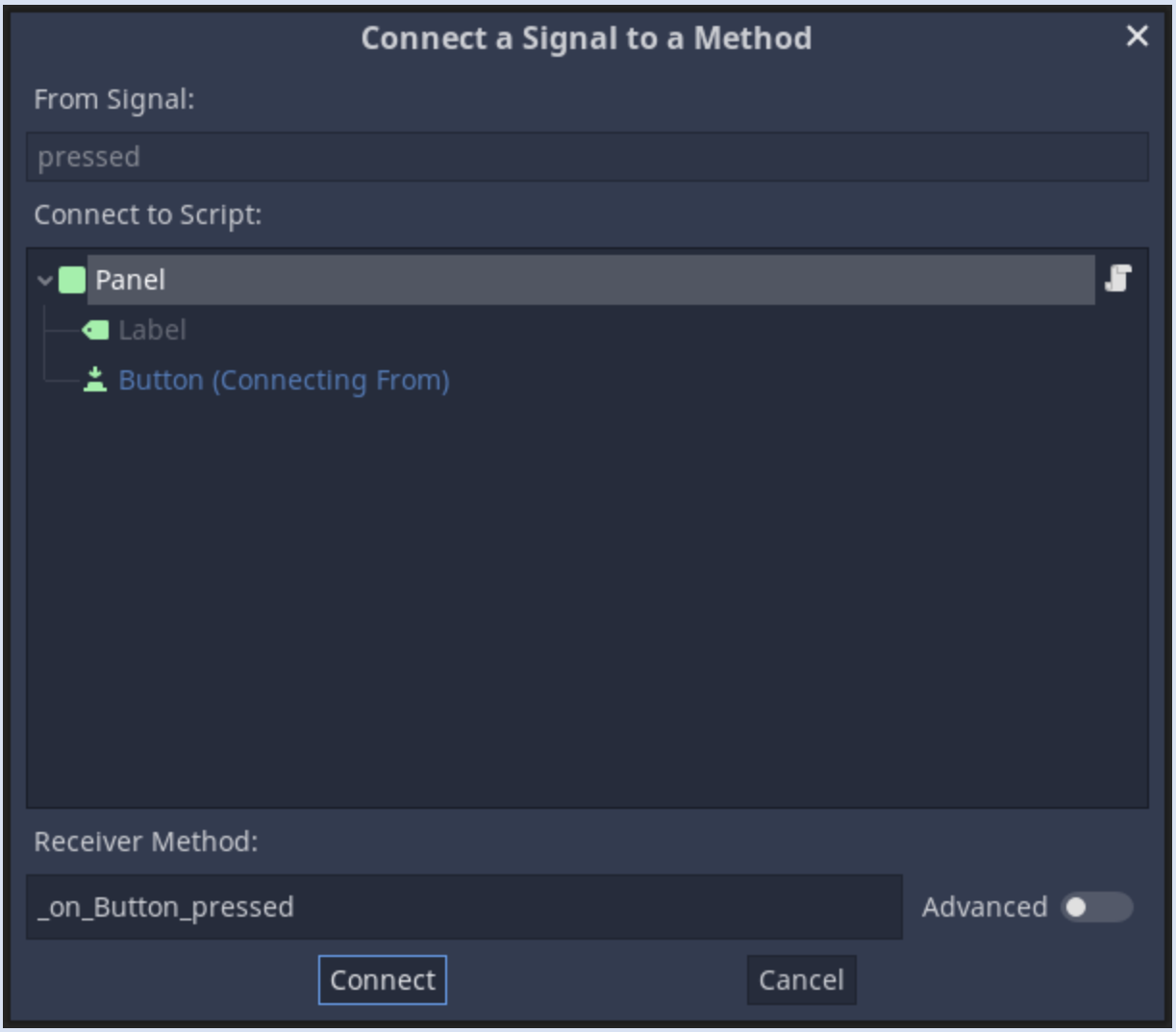
While we will use the code method for the remainder of this tutorial series, let's cover how the editor interface works for future reference.

Select the Button node in the scene tree and then select the "Node" tab. Next, make sure that you have "Signals" selected.





If you then select "pressed()" under "BaseButton" and click the "Connect..." button in the bottom right, you'll open up the connection creation dialogue.



The top of the dialogue displays a list of your scene's nodes with the emitting node's name highlighted in blue. Select the "Panel" node here.

The bottom of the dialogue shows the name of the method that will be created. By default, the method name will contain the emitting node's name ("Button" in this case), resulting in `_on_[EmitterNode]_[signal_name]`.

And that concludes the guide on how to use the visual interface. However, this is a scripting tutorial, so for the sake of learning, let's dive into the manual process!

To accomplish this, we will introduce a function that is probably the most used by Godot programmers: Node.get_node(). This function uses paths to fetch nodes anywhere in the scene, relative to the node that owns the script.

For the sake of convenience, delete everything underneath `extends Panel`. You will fill out the rest of the script manually.

Because the Button and Label are siblings under the Panel where the script is attached, you can fetch the Button by typing the following underneath the _ready() function:

```
func _ready():
    get_node("Button")
```

Next, write a function which will be called when the button is pressed:

```
func _on_Button_pressed():
    get_node("Label").text = "HELLO!"
```

Finally, connect the button's "pressed" signal to _on_Button_pressed() by using Object.connect().

```
func _ready():
    get_node("Button").connect("pressed", self, "_on_Button_pressed")
```

The final script should look like this:

```
extends Panel

func _ready():
    get_node("Button").connect("pressed", self, "_on_Button_pressed")

func _on_Button_pressed():
    get_node("Label").text = "HELLO!"
```

Run the scene and press the button. You should get the following result:

Why, hello there! Congratulations on scripting your first scene.

Note

A common misunderstanding regarding this tutorial is how get_node(path) works. For a given node, get_node(path) searches its immediate children. In the above code, this means that Button must be a child of Panel. If Button were instead a child of Label, the code to obtain it would be:

```
# Not for this case,
# but just in case.
get_node("Label/Button")
```

Also, remember that nodes are referenced by name, not by type.

Note

The 'advanced' panel of the connect dialogue is for binding specific values to the connected function's parameters. You can add and remove values of different types.

The code approach also enables this with a 4th Array parameter that is empty by default. Feel free to read up on the Object.connect method for more information.

Processing

Several actions in Godot are triggered by callbacks or virtual functions, so there is no need to write code that runs all the time.

However, it is still common to need a script to be processed on every frame. There are two types of processing: idle processing and physics processing.

Idle processing is activated when the method Node._process() is found in a script. It can be turned off and on with the Node.set_process() function.

This method will be called every time a frame is drawn:

```
func _process(delta):
    # Do something...
    pass
```

It's important to bear in mind that the frequency with which _process() will be called depends on how many frames per second (FPS) your application is running at. This rate can vary over time and devices.

To help manage this variability, the delta parameter contains the time elapsed in seconds as a floating-point number since the previous call to _process().

This parameter can be used to make sure things always take the same amount of time, regardless of the game's FPS.

For example, movement is often multiplied with a time delta to make movement speed both constant and independent of the frame rate.

Physics processing with _physics_process() is similar, but it should be used for processes that must happen before each physics step, such as controlling a character. It always runs before a physics

step and it is called at fixed time intervals: 60 times per second by default. You can change the interval from the Project Settings, under Physics -> Common -> Physics Fps.

The function _process(), however, is not synced with physics. Its frame rate is not constant and is dependent on hardware and game optimization. Its execution is done after the physics step on single-threaded games.

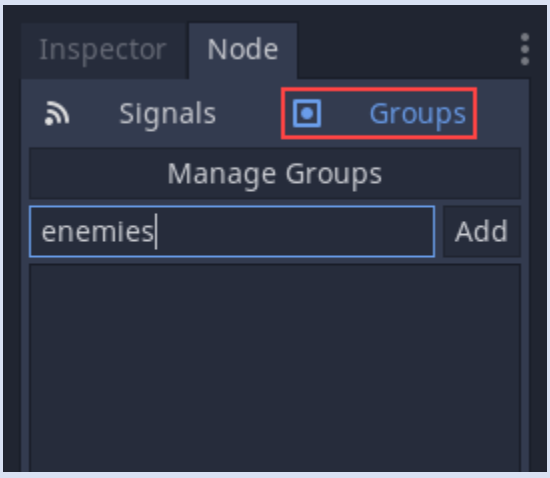
A simple way to see the _process() function at work is to create a scene with a single Label node, with the following script:

```
extends Label

var accum = 0

func _process(delta):
    accum += delta
    text = str(accum) # 'text' is a built-in label property.
```

Which will show a counter increasing each frame.

Groups

Groups in Godot work like tags you might have come across in other software. A node can be added to as many groups as desired. This is a useful feature for organizing large scenes. There are two ways to add nodes to groups. The first is from the UI, using the Groups button under the Node panel:



And the second way is from code. The following script would add the current node to the enemies group as soon as it appeared in the scene tree.

```
func _ready():
    add_to_group("enemies")
```

This way, if the player is discovered sneaking into a secret base, all enemies can be notified about its alarm sounding by using SceneTree.call_group():

```
func _on_discovered(): # This is a purely illustrative function.
    get_tree().call_group("enemies", "player_was_discovered")
```

The above code calls the function player_was_discovered on every member of the group enemies.

It is also possible to get the full list of enemies nodes by calling SceneTree.get_nodes_in_group():

```
var enemies = get_tree().get_nodes_in_group("enemies")
```

The SceneTree class provides many useful methods, like interacting with scenes, their node hierarchy and groups of nodes. It allows you to easily switch scenes or reload them, to quit the game or pause and unpause it. It even comes with interesting signals. So check it out if you have some time!

Notifications

Godot has a system of notifications. These are usually not needed for scripting, as it's too low-level and virtual functions are provided for most of them. It's just good to know they exist. For example, you may add an Object._notification() function in your script:

```
func _notification(what):
    match what:
        NOTIFICATION_READY:
            print("This is the same as overriding _ready()...")
        NOTIFICATION_PROCESS:
            print("This is the same as overriding _process()...")
```

The documentation of each class in the Class Reference shows the notifications it can receive. However, in most cases GDScript provides simpler overridable functions.

Overridable functions

Such overridable functions, which are described as follows, can be applied to nodes:

```
func _enter_tree():
    # When the node enters the Scene Tree, it becomes active
    # and  this function is called. Children nodes have not entered
    # the active scene yet. In general, it's better to use _ready()
    # for most cases.
    pass

func _ready():
    # This function is called after _enter_tree, but it ensures
    # that all children nodes have also entered the Scene Tree,
    # and became active.
    pass
```


```
func _exit_tree():
    # When the node exits the Scene Tree, this function is called.
    # Children nodes have all exited the Scene Tree at this point
    # and all became inactive.
    pass

func _process(delta):
    # This function is called every frame.
    pass

func _physics_process(delta):
    # This is called every physics frame.
    pass
```

As mentioned before, it's better to use these functions instead of the notification system.

Creating nodes

To create a node from code, call the .new() method, like for any other class-based datatype. For example:

```
var s
func _ready():
    s = Sprite.new() # Create a new sprite!
    add_child(s) # Add it as a child of this node.
```

To delete a node, be it inside or outside the scene, free() must be used:

```
func _someaction():
    s.free() # Immediately removes the node from the scene and frees it.
```

When a node is freed, it also frees all its child nodes. Because of this, manually deleting nodes is much simpler than it appears. Free the base node and everything else in the subtree goes away with it.

A situation might occur where we want to delete a node that is currently "blocked", because it is emitting a signal or calling a function. This will crash the game. Running Godot with the debugger will often catch this case and warn you about it.

The safest way to delete a node is by using Node.queue_free(). This erases the node safely during idle.

```
func _someaction():
    s.queue_free() # Removes the node from the scene and frees it when it becomes safe to do so.
```

Instancing scenes


Instancing a scene from code is done in two steps. The first one is to load the scene from your hard drive:

```
var scene = load("res://myscene.tscn") # Will load when the script is instanced.
```

Preloading it can be more convenient, as it happens at parse time (GDScript only):

```
var scene = preload("res://myscene.tscn") # Will load when parsing the script.
```

But `scene` is not yet a node. It's packed in a special resource called PackedScene. To create the actual node, the function PackedScene.instance() must be called. This will return the tree of nodes that can be added to the active scene:

```
var node = scene.instance()
add_child(node)
```

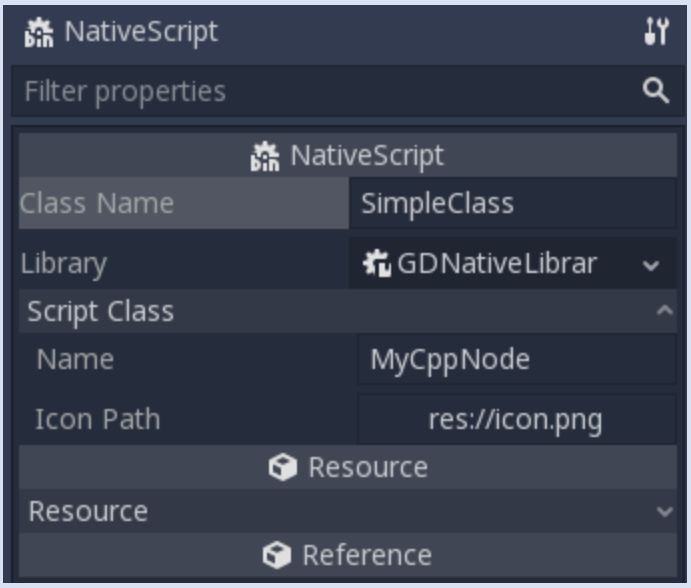
The advantage of this two-step process is that a packed scene may be kept loaded and ready to use so that you can create as many instances as desired. This is especially useful to quickly instance several enemies, bullets, and other entities in the active scene.

Register scripts as classes

Godot has a "Script Class" feature to register individual scripts with the Editor. By default, you can only access unnamed scripts by loading the file directly.

You can name a script and register it as a type in the editor with the `class_name` keyword followed by the class's name. You may add a comma and an optional path to an image to use as an icon. You will then find your new type in the Node or Resource creation dialog.

```
extends Node

# Declare the class name here
class_name ScriptName, "res://path/to/optional/icon.svg"

func _ready():
    var this = ScriptName          # reference to the script
    var cppNode = MyCppNode.new()   # new instance of a class named MyCppNode

    cppNode.queue_free()
```



Warning

In Godot 3.1:

- Only GDScript and NativeScript, i.e., C++ and other GDNative-powered languages, can register scripts.
- Only GDScript creates global variables for each named script.

Signals

Introduction

Signals are Godot's version of the *observer* pattern. They allow a node to send out a message that other nodes can listen for and respond to. For example, rather than continuously checking a button to see if it's being pressed, the button can emit a signal when it's pressed.

Note

You can read more about the observer pattern here:
https://gameprogrammingpatterns.com/observer.html

Signals are a way to *decouple* your game objects, which leads to better organized and more manageable code. Instead of forcing game objects to expect other objects to always be present, they can instead emit signals that all interested objects can subscribe to and respond to.

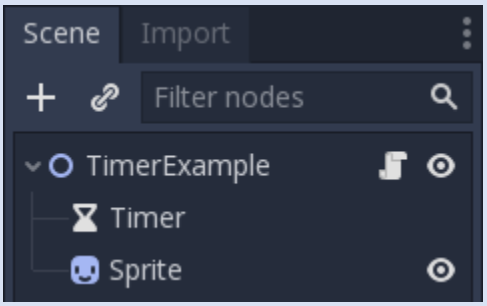
Below you can see some examples of how you can use signals in your own projects.

Timer example

To see how signals work, let's try using a Timer node. Create a new scene with a Node2D and two children: a Timer and a Sprite. In the Scene dock, rename Node2D to TimerExample.

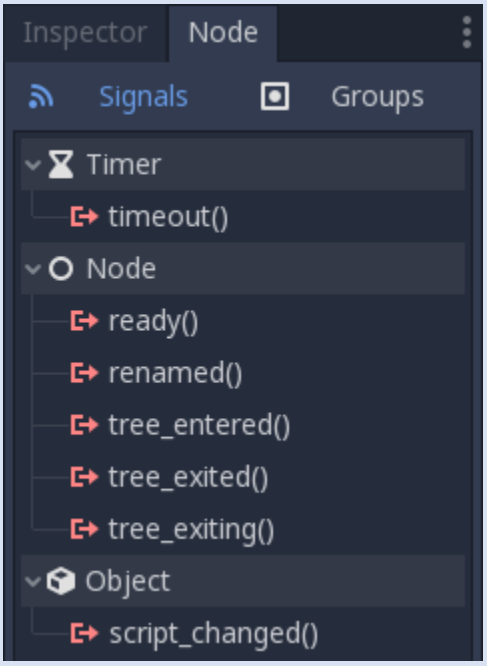
For the Sprite's texture, you can use the Godot icon, or any other image you like. Do so by selecting Load in the Sprite's Texture attribute drop-down menu. Attach a script to the root node, but don't add any code to it yet.

Your scene tree should look like this:

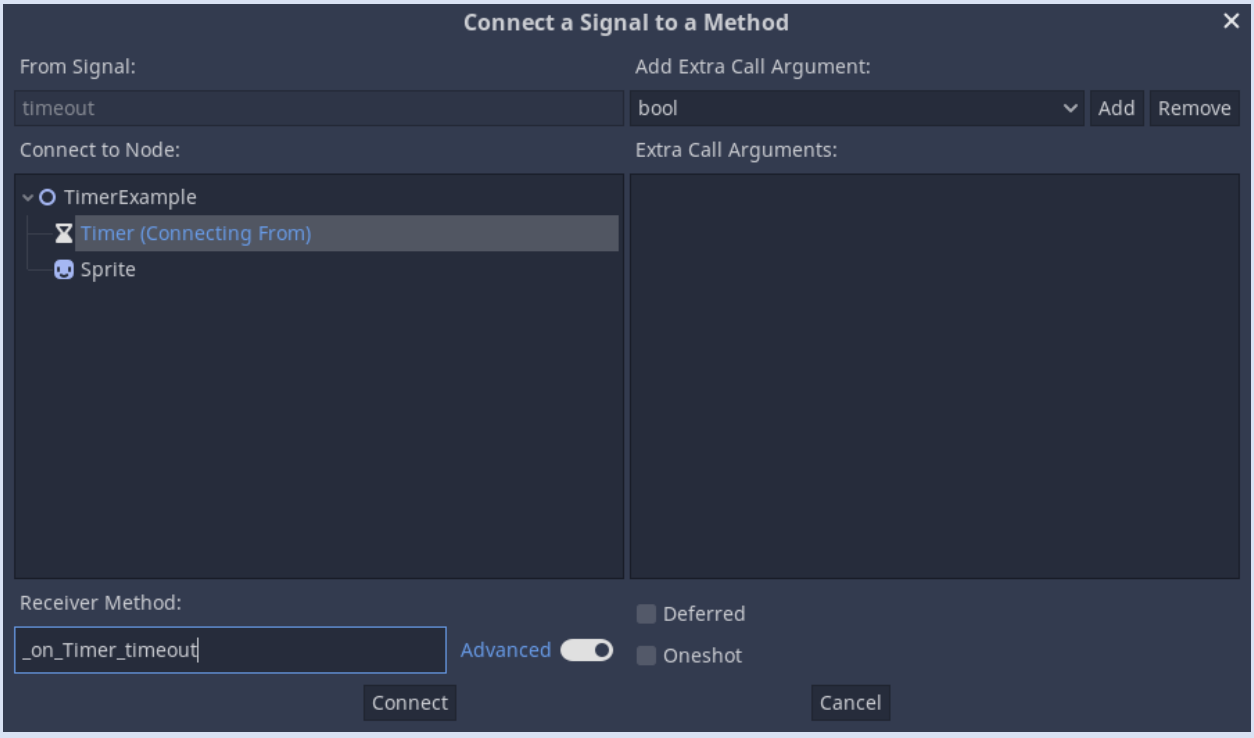


In the Timer node's properties, check the "On" box next to *Autostart*. This will cause the timer to start automatically when you run the scene. You can leave the *Wait Time* at 1 second.

Next to the "Inspector" tab is a tab labeled "Node". Click on this tab and you'll see all of the signals that the selected node can emit. In the case of the Timer node, the one we're concerned with is "timeout". This signal is emitted whenever the Timer reaches 0.



Click on the "timeout()" signal and click "Connect..." at the bottom of the signals panel. You'll see the following window, where you can define how you want to connect the signal:



On the left side, you'll see the nodes in your scene and can select the node that you want to "listen" for the signal. Note that the Timer node is blue, this is a visual indication that it's the node that is emitting the signal. Select the root node.

Warning

The target node *must* have a script attached or you'll receive an error message.

If you toggle the Advanced menu, you'll see on the right side that you can bind an arbitrary number of arguments of (possibly) different types. This can be useful when you have more than one signal connected to the same method, as each signal propagation will result in different values for those extra call arguments.

On the bottom of the window is a field labeled "Receiver Method". This is the name of the function in the target node's script that you want to use. By default, Godot will create this function using the naming convention _on_<node_name>_<signal_name> but you can change it if you wish.

Click "Connect" and you'll see that the function has been created in the script:

```
extends Node2D


func _on_Timer_timeout():
    pass # Replace with function body.
```

Now we can replace the placeholder code with whatever code we want to run when the signal is received. Let's make the Sprite blink:

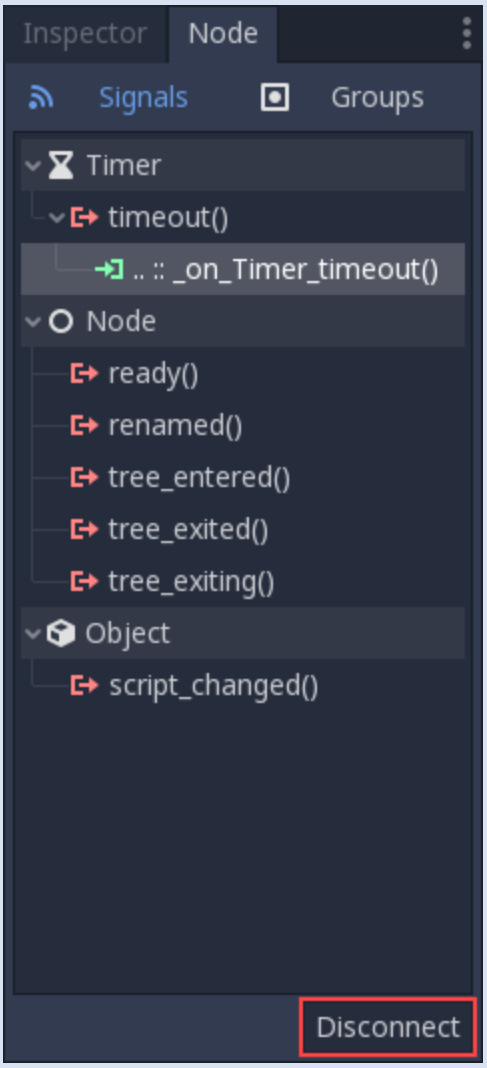
```
extends Node2D


func _on_Timer_timeout():
    # Note: the `$` operator is a shorthand for `get_node()`,
    # so `$Sprite` is equivalent to `get_node("Sprite")`.
    $Sprite.visible = !$Sprite.visible
```

Run the scene and you'll see the Sprite blinking on and off every second. You can change the Timer's *Wait Time* property to alter this.

Connecting signals in code

You can also make the signal connection in code rather than with the editor. This is usually necessary when you're instancing nodes via code and so you can't use the editor to make the connection.

First, disconnect the signal by selecting the connection in the Timer's "Node" tab and clicking disconnect.



To make the connection in code, we can use the connect function. We'll put it in _ready() so that the connection will be made on run. The syntax of the function is <source_node>.connect(<signal_name>, <target_node>, <target_function_name>). Here is the code for our Timer connection:

```
extends Node2D


func _ready():
    $Timer.connect("timeout", self, "_on_Timer_timeout")


func _on_Timer_timeout():
    $Sprite.visible = !$Sprite.visible
```

Custom signals

You can also declare your own custom signals in Godot:

```
extends Node2D


signal my_signal
```

Once declared, your custom signals will appear in the Inspector and can be connected in the same way as a node's built-in signals.

To emit a signal via code, use the emit_signal function:

```
extends Node2D


signal my_signal


func _ready():
    emit_signal("my_signal")
```

A signal can also optionally declare one or more arguments. Specify the argument names between parentheses:

```
extends Node


signal my_signal(value, other_value)
```

Note

The signal arguments show up in the editor's node dock, and Godot can use them to generate callback functions for you. However, you can still emit any number of arguments when you emit signals. So it's up to you to emit the correct values.

To pass values, add them as the second argument to the emit_signal function:

```
extends Node


signal my_signal(value, other_value)


func _ready():
    emit_signal("my_signal", true, 42)
```

Conclusion

Many of Godot's built-in node types provide signals you can use to detect events. For example, an Area2D representing a coin emits a body_entered signal whenever the player's physics body enters its collision shape, allowing you to know when the player collected it.