Welcome to our beginner-friendly course on the Godot game engine. In this article, we will provide an overview of the course structure and the topics we will cover. Your instructor for this course is Daniel Buckley, and we are excited to help you begin your game development journey with Godot.

About Godot

Godot is a popular, free, and open-source game engine that has gained significant traction in recent years. It offers a wide range of features and receives regular updates, making it a robust choice for both beginners and experienced developers.

Course Overview

Throughout this course, we will explore various aspects of the Godot engine. Here is a breakdown of the topics we will cover:

Getting Started with Godot

- Creating a new project
- Installing the Godot editor
- Overview of the Godot application interface

We will familiarize ourselves with the different windows, buttons, and panels within the Godot editor to gain a well-rounded understanding of how to use the engine effectively.

Fundamentals of Godot

- Introduction to 2D game development
- Exploring key concepts such as nodes, scenes, tools, parenting, and cameras

3D Game Development

- Creating 3D environments
- Working with 3D models, materials, and lighting

Scripting in Godot

In the second half of the course, we will focus on scripting, which involves writing code to define gameplay behaviors. No prior experience is required, as we will start from the very beginning.

Practical Application

Towards the end of the course, we will apply all the knowledge we have gained by creating a simple coin collector mini-game. This hands-on project will help reinforce the concepts learned throughout the course.

About Zenva

Zenva is an online learning academy with over a million students. We offer a wide range of courses suitable for beginners and those looking to learn something new. Our courses are versatile, allowing you to follow along with videos, read lesson summaries, or use the included course files to learn at your own pace.

Getting Started

Now that you have an overview of what to expect, let’s dive into the first lesson and begin our

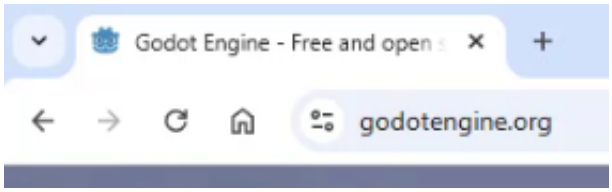
journey with the Godot game engine.

We look forward to seeing the amazing games you will create!

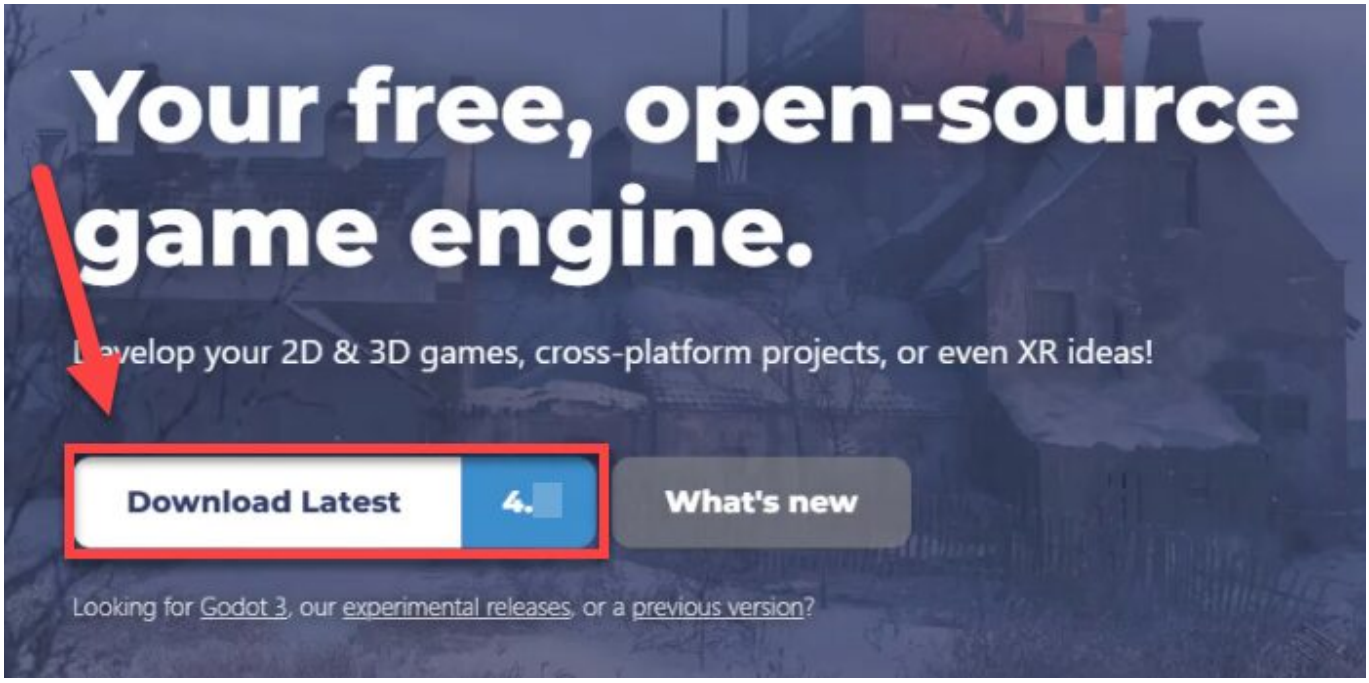
Welcome to this beginner-friendly guide on installing the Godot Engine. By the end of this article, you will have Godot up and running on your computer, ready for you to start creating your own games. Let's get started.

Downloading Godot

To begin, you need to visit the official Godot website. Open your web browser and, in the address bar, type **GodotEngine.org** and press Enter.



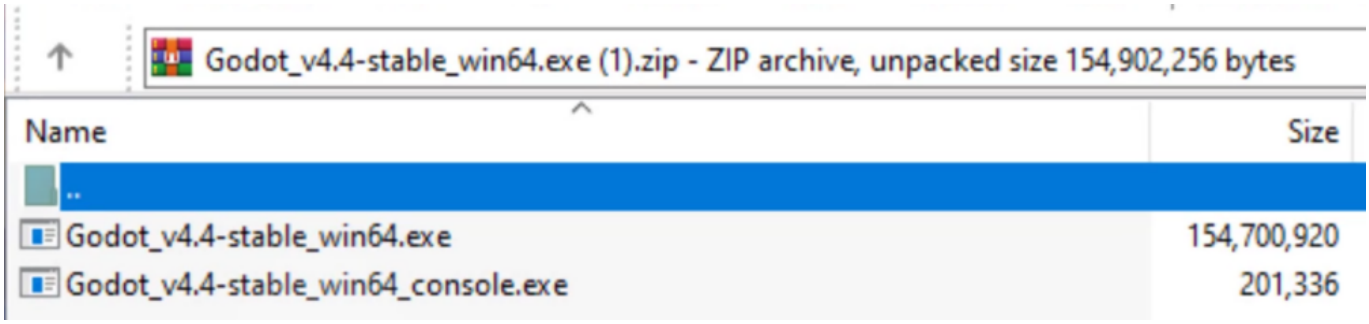
Once you are on the homepage, look for and click the **Download Latest** button at the top of the screen. It should be a big white button with a version number next to it.



The website will automatically detect your operating system and direct you to the appropriate download page. From here, scroll down to find the blue **Godot Engine** button. Click on the button to start the download process for the standard version.

Godot is a very lightweight engine, with the download size being only around 60 megabytes. This makes the download process quick and efficient!

After the download is complete, open the downloaded zip file. You will see two files: the main engine executable and a console executable.



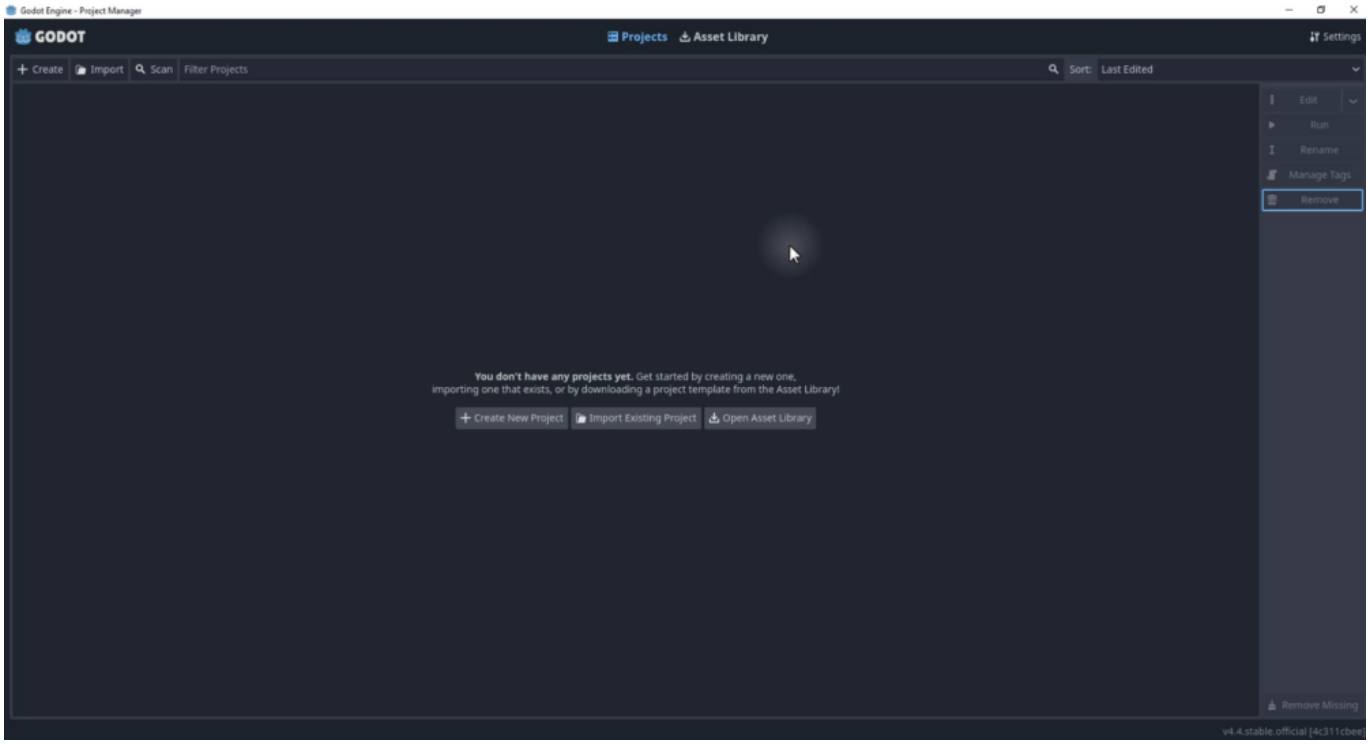
Extract the main engine executable to a folder on your desktop or any other preferred location. Godot is unique in that the entire engine is contained within a single executable file, making it easy to manage and use (with no extra installation steps required).

Launching Godot

To open Godot, simply double-click on the executable file. This will launch the Godot Project Manager. The Project Manager is not the engine itself but a launcher where you can:

- Create new projects.
- View and manage existing projects.

The Project Manager allows you to easily switch between multiple projects, making it convenient for developers working on several games simultaneously.



Next Steps

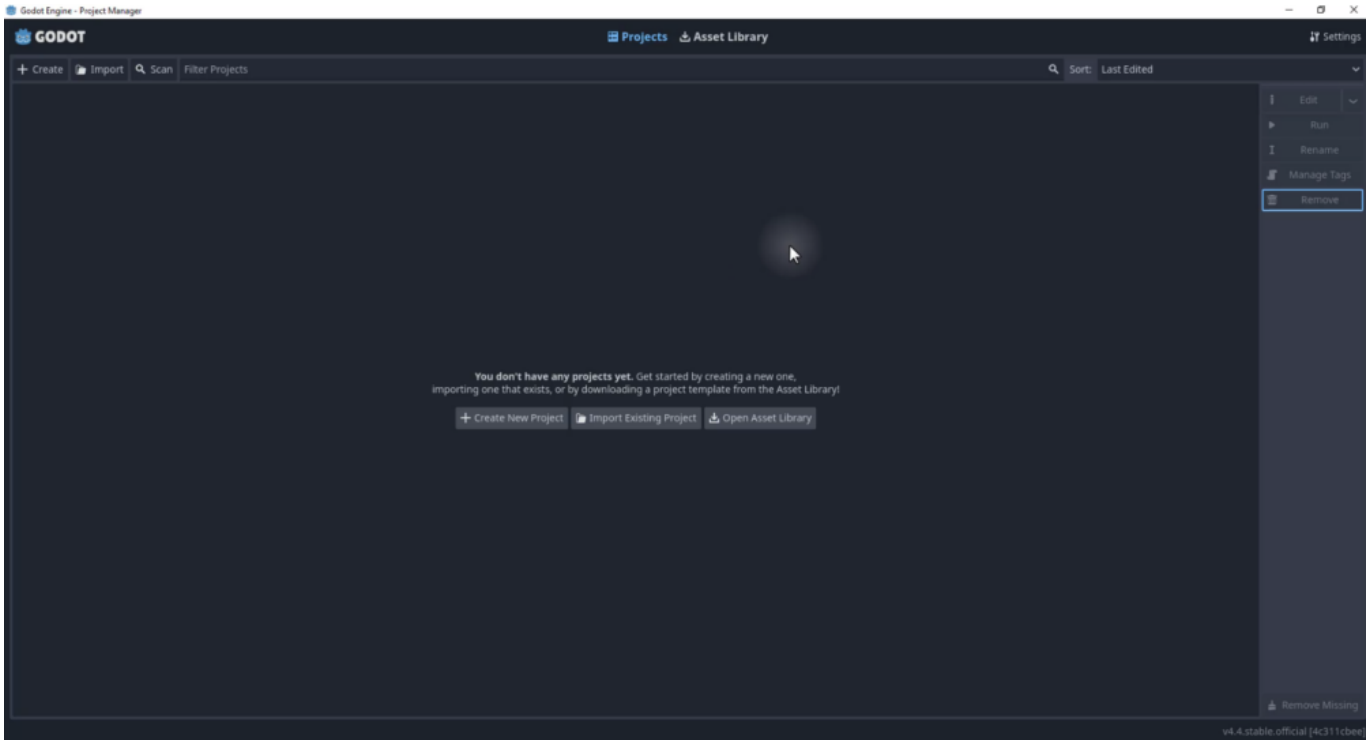
In the next lesson, we will explore the Project Manager in more detail and learn how to create a new project. Stay tuned for more insights and tutorials on using Godot.

Thank you for reading, and we look forward to seeing you in the next lesson.

In this lesson, we will explore how to set up a project in Godot and go over the Project Manager. Let's get started.

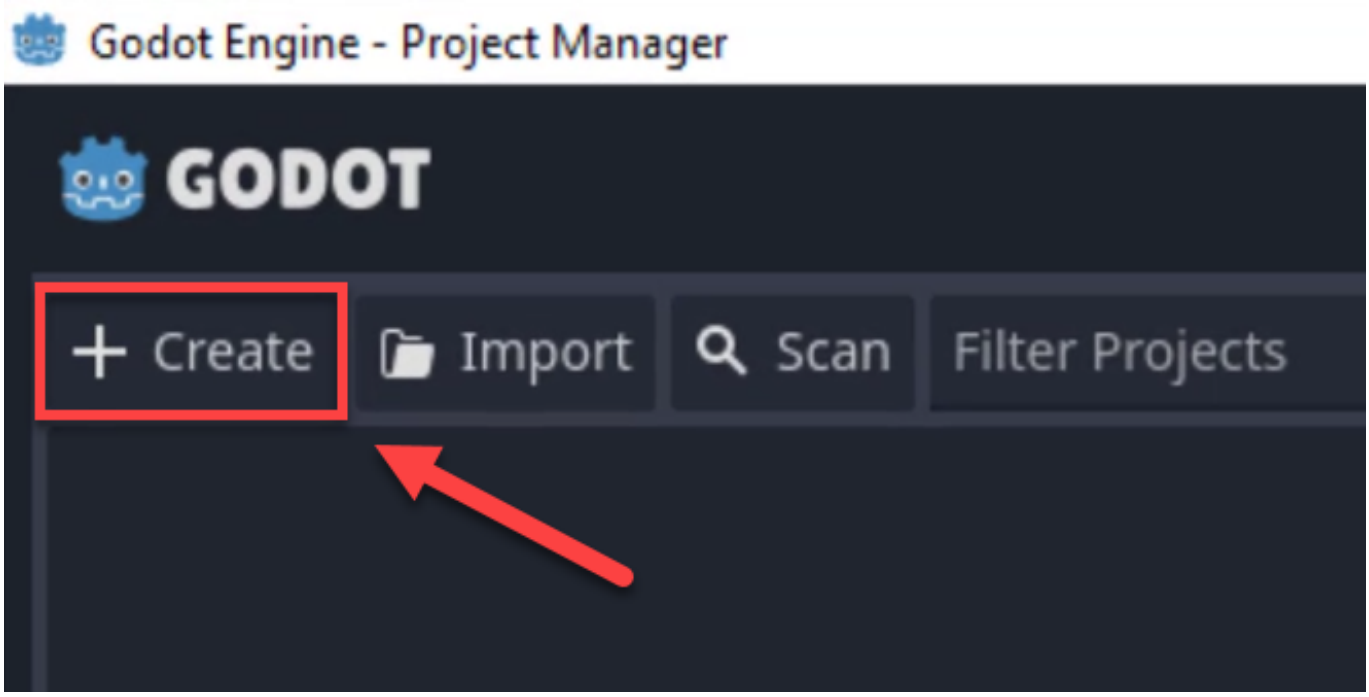
Accessing the Godot Project Manager

To begin, ensure you have the Godot Project Manager open on your screen. If not, navigate to the location where you extracted the Godot engine executable and double-click it. The Godot Project Manager serves as the launcher for Godot and is where you will manage all your game projects.

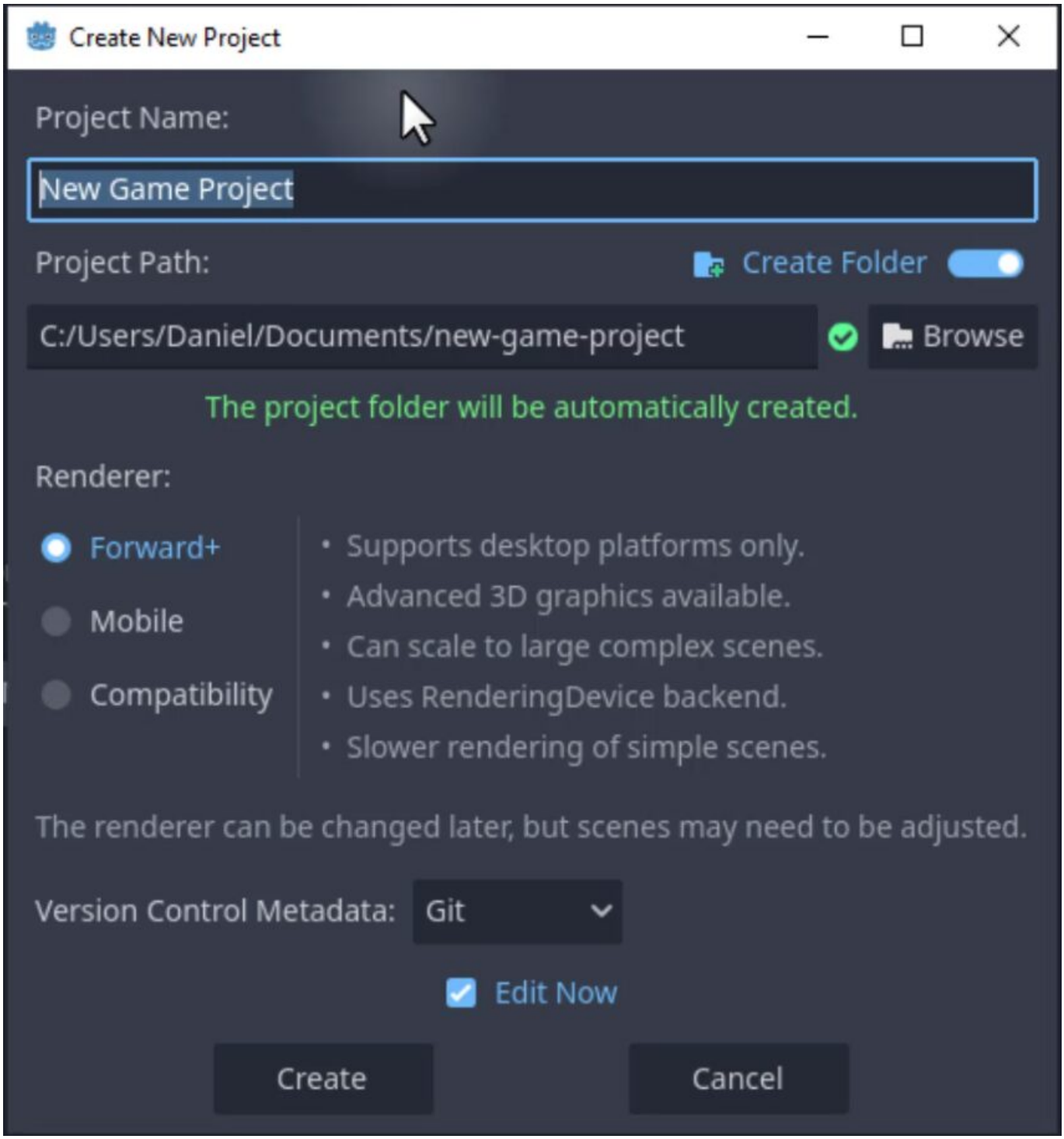


Creating a New Project

Since you are starting fresh, you will not see any projects listed. To create a new project, click the **Create** button located in the top left corner of the screen.



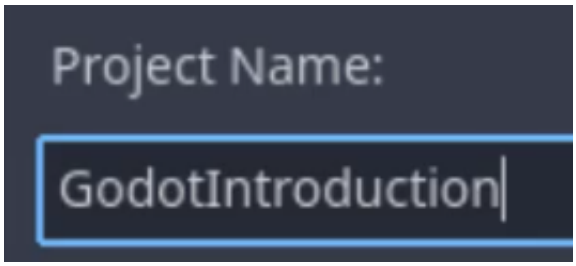
A new window will appear, allowing you to configure your project settings.



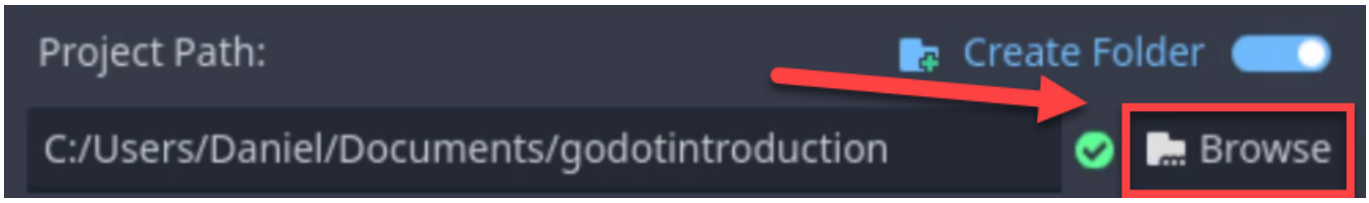
Configuring Project Settings

In the project configuration window, you can customize several options:

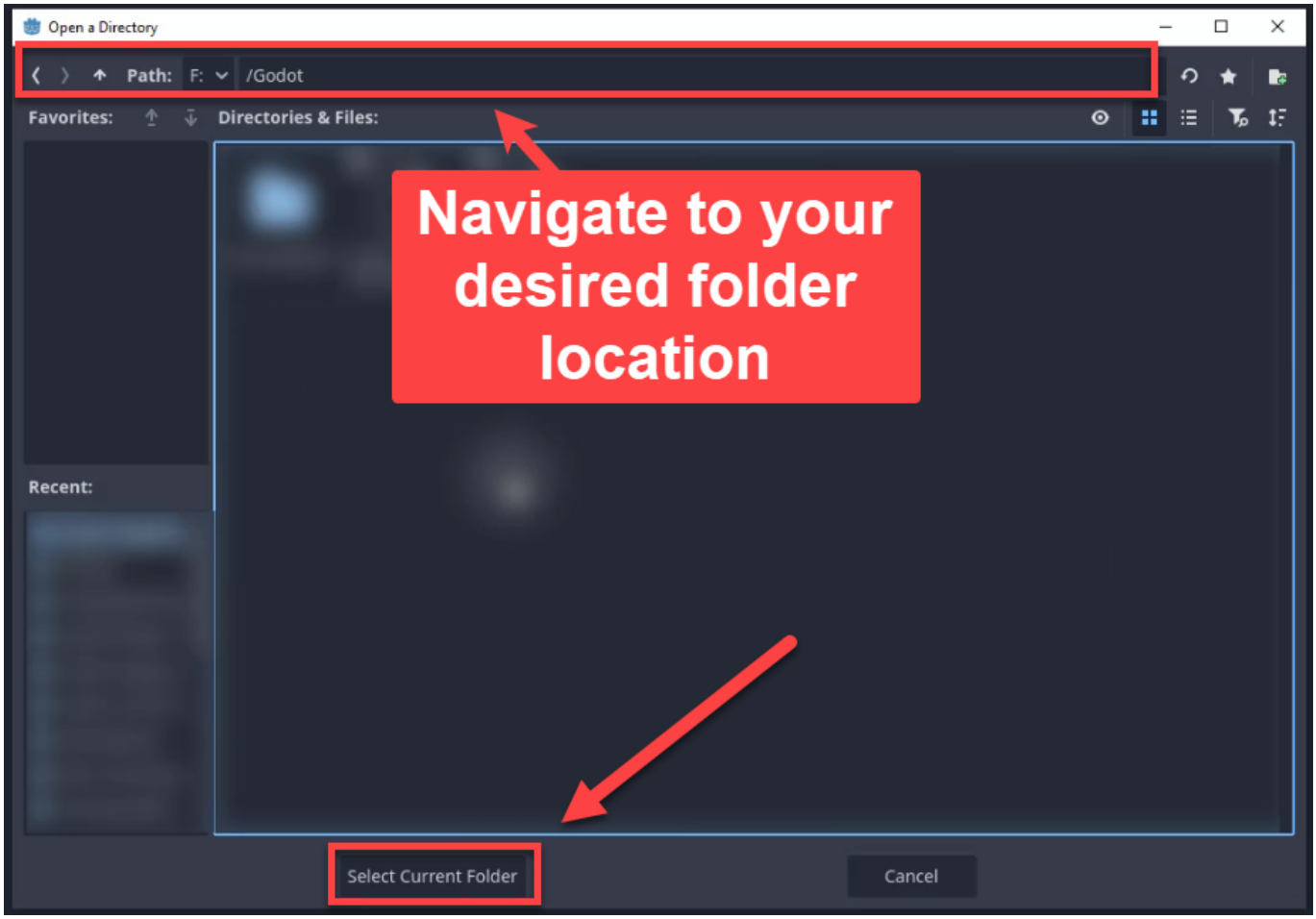
1) **Project Name:** Enter the name of your project. For this example, let's use "GodotIntroduction".



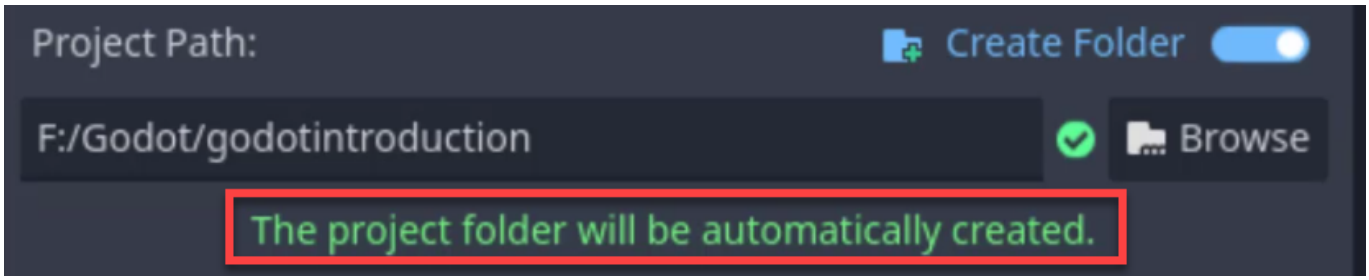
2) **Project Path:** Specify the location where your game project folder will be stored. This folder will contain all your scripts, scenes, textures, sprites, and 3D models. It is recommended to store this in a dedicated folder for all your Godot projects. To select a location, click the **Browse** button to navigate to open the file explorer.



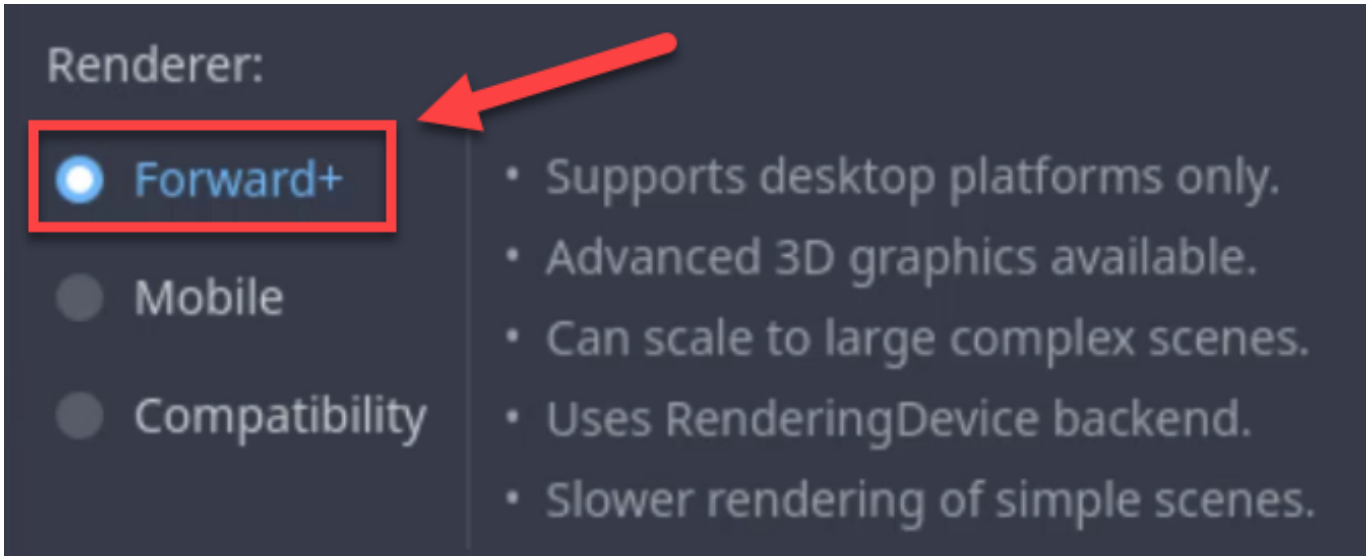
Select the folder and click **Select Current Folder**.



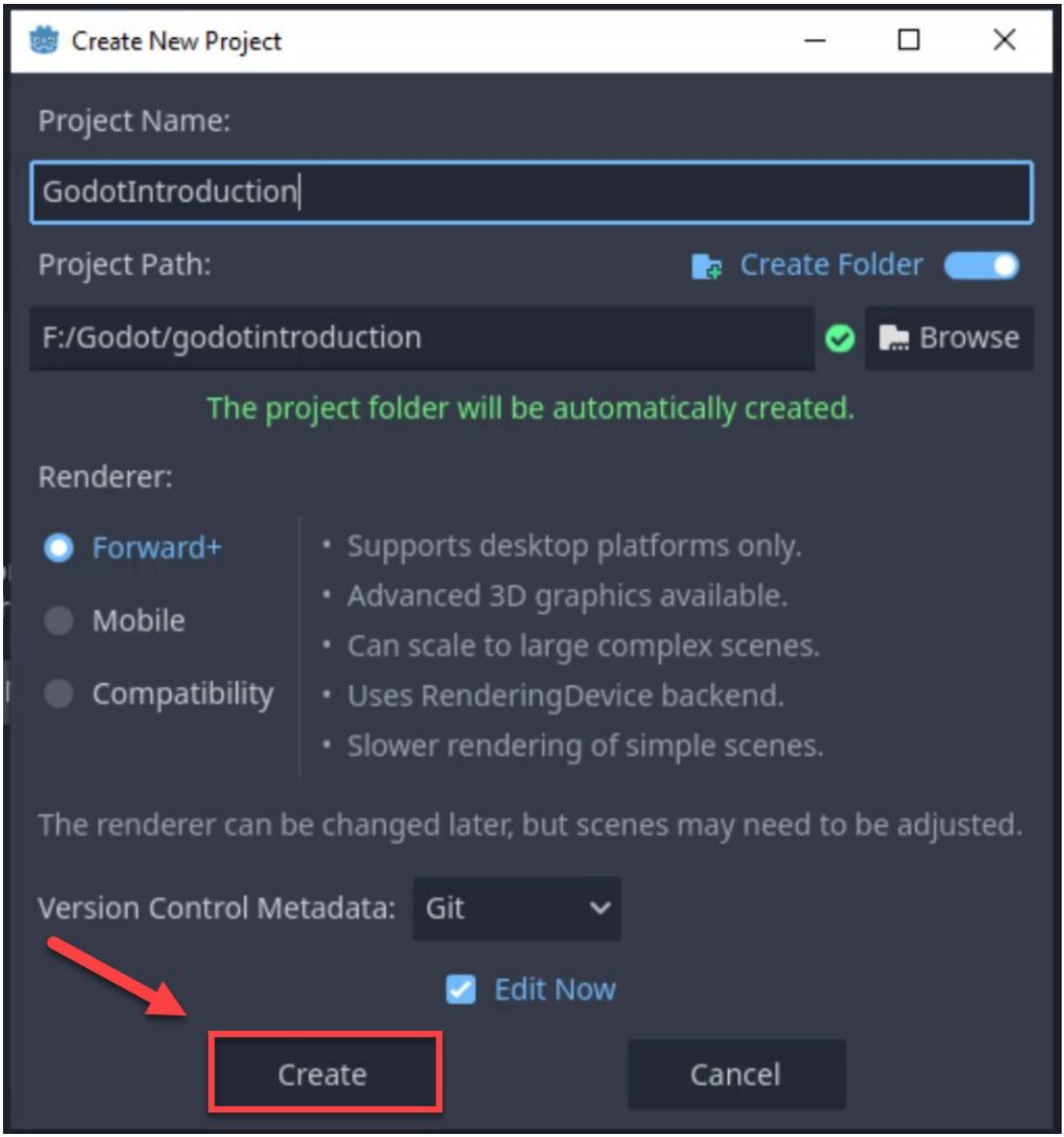
Ensure the green text at the bottom indicates that everything is okay.



3) **Renderer:** You can choose the Rendererer that bests suits the requirements for your project. In our case, verify that the **Forward Plus** renderer is enabled (usually by default, as it is the standard renderer for Godot).

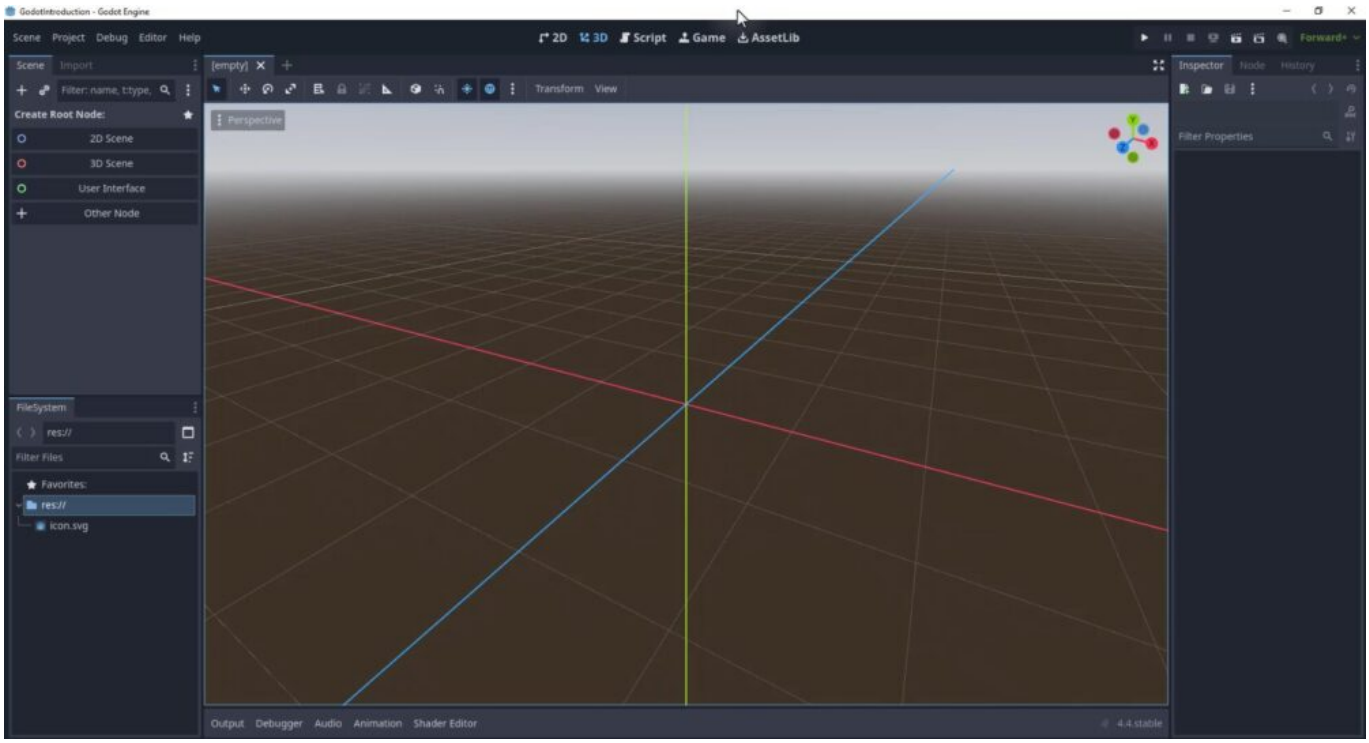


4) Once you finish configuring your project, click the **Create** button to finalize your project setup.



Exploring the Godot Editor

Once you click create, the Godot Editor will open.



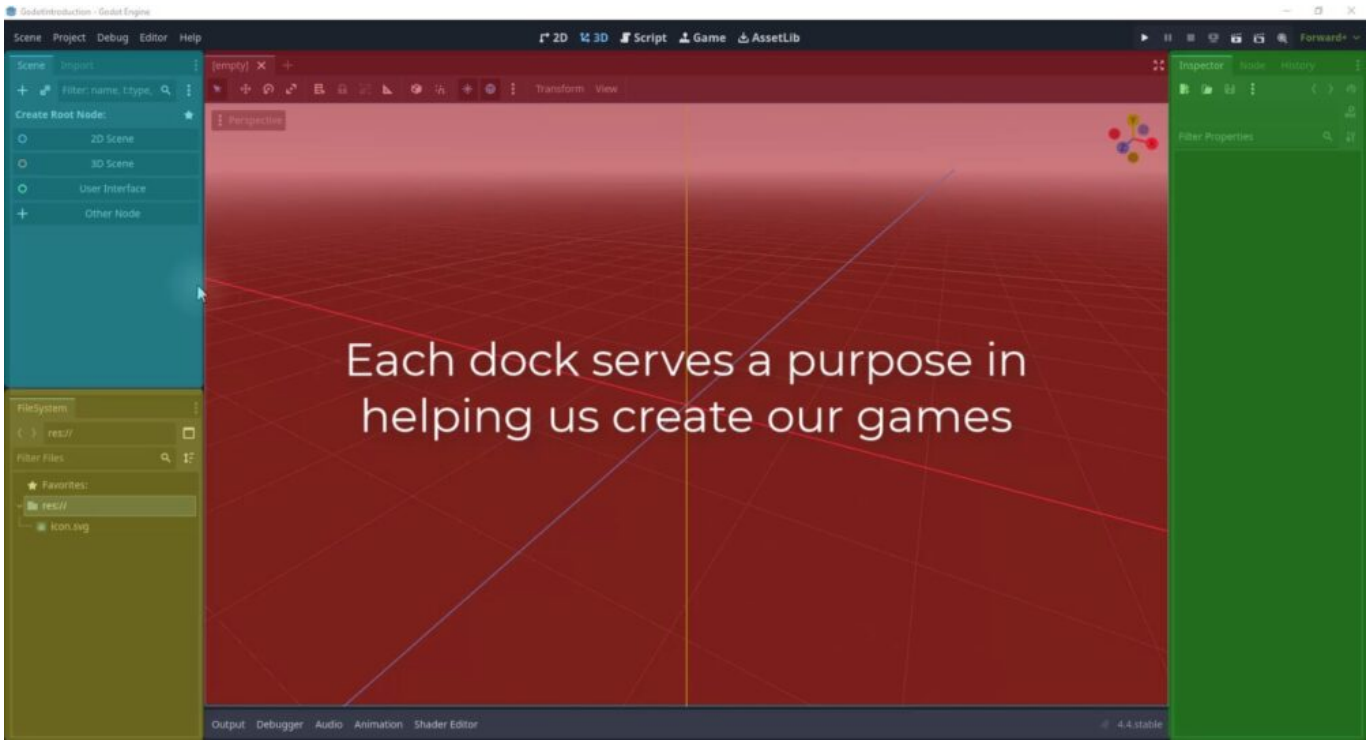
This is the application where you will create your games, write code, and test your project. At first glance, the editor might seem overwhelming with its various panels, buttons, and icons. However, do not be discouraged. Throughout this course, we will guide you step-by-step through all the important tools and aspects you need to understand.

In the next lesson, we will provide an overview of the Godot Editor, helping you understand the general functionality of its components. Stay tuned and happy learning!

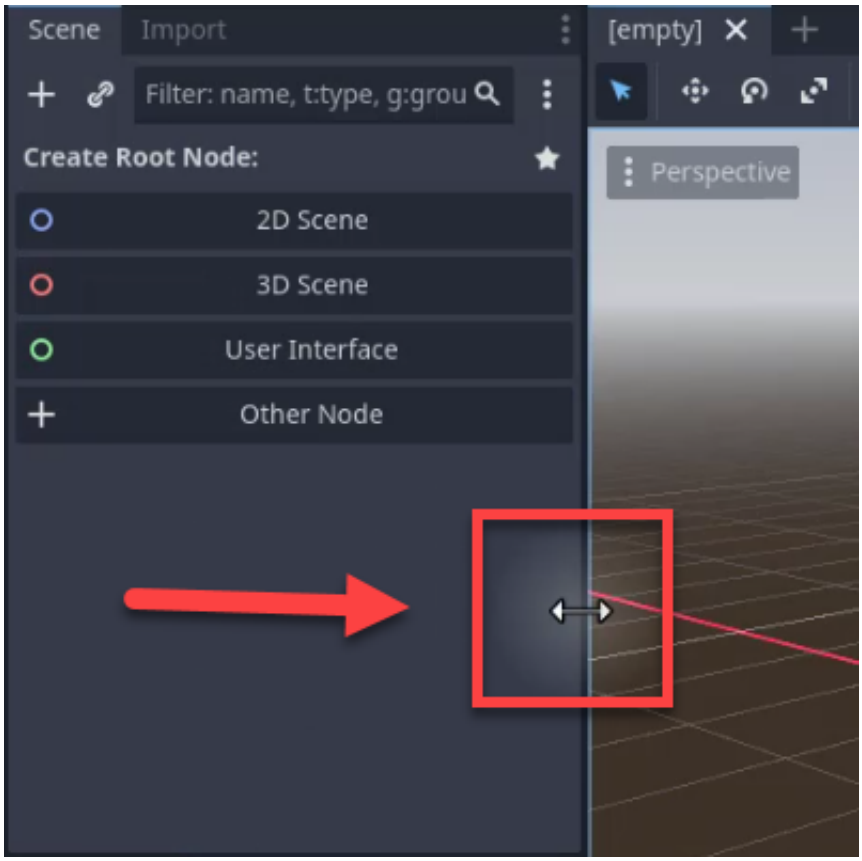
Welcome to your first lesson on exploring the Godot Editor. In this article, we will guide you through the Godot Editor interface, its functionalities, and how to navigate through it. The Godot Editor is where you will spend most of your time designing levels, importing assets, writing code, and testing your game.

Understanding the Godot Editor

The Godot Editor is divided into several **docks**. These docks are self-contained rectangles, each with its own functionality.



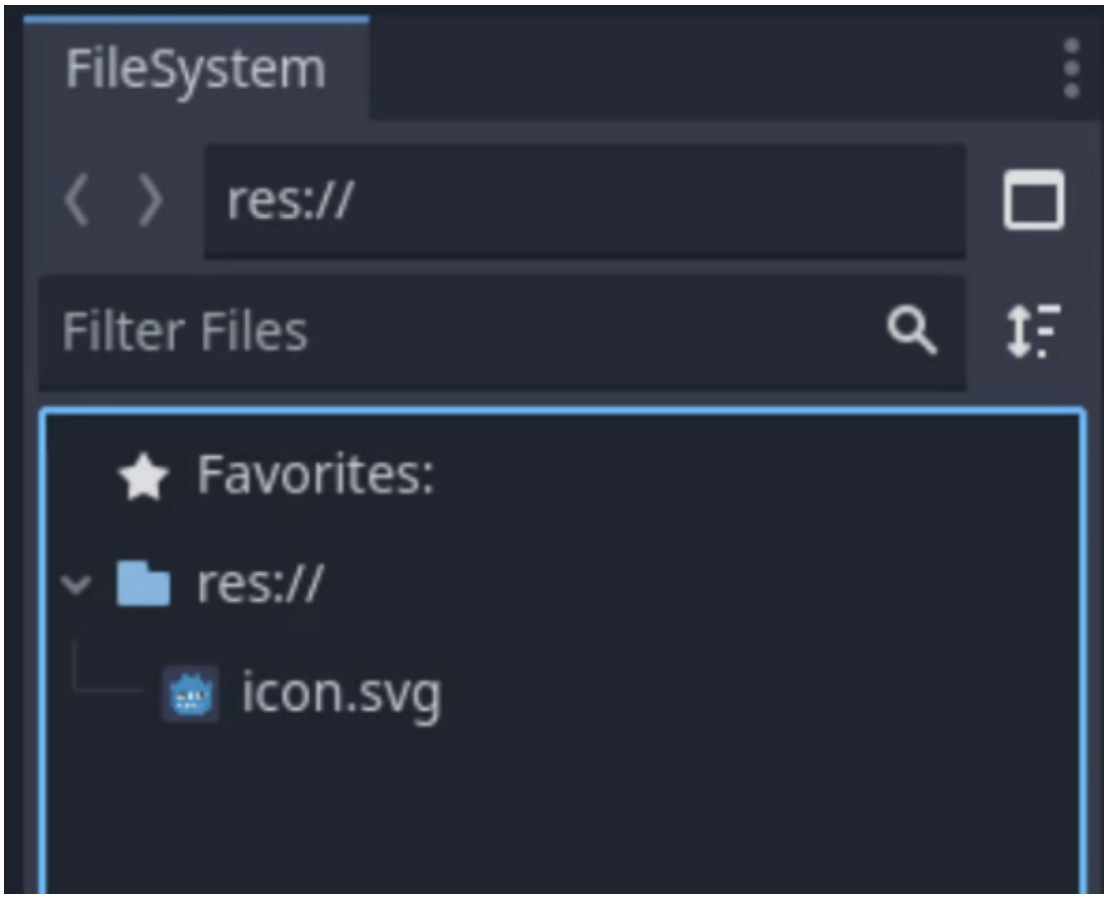
Docks serve specific purposes to help you develop your games. You can also resize these docks by clicking and dragging their edges.



File System Dock

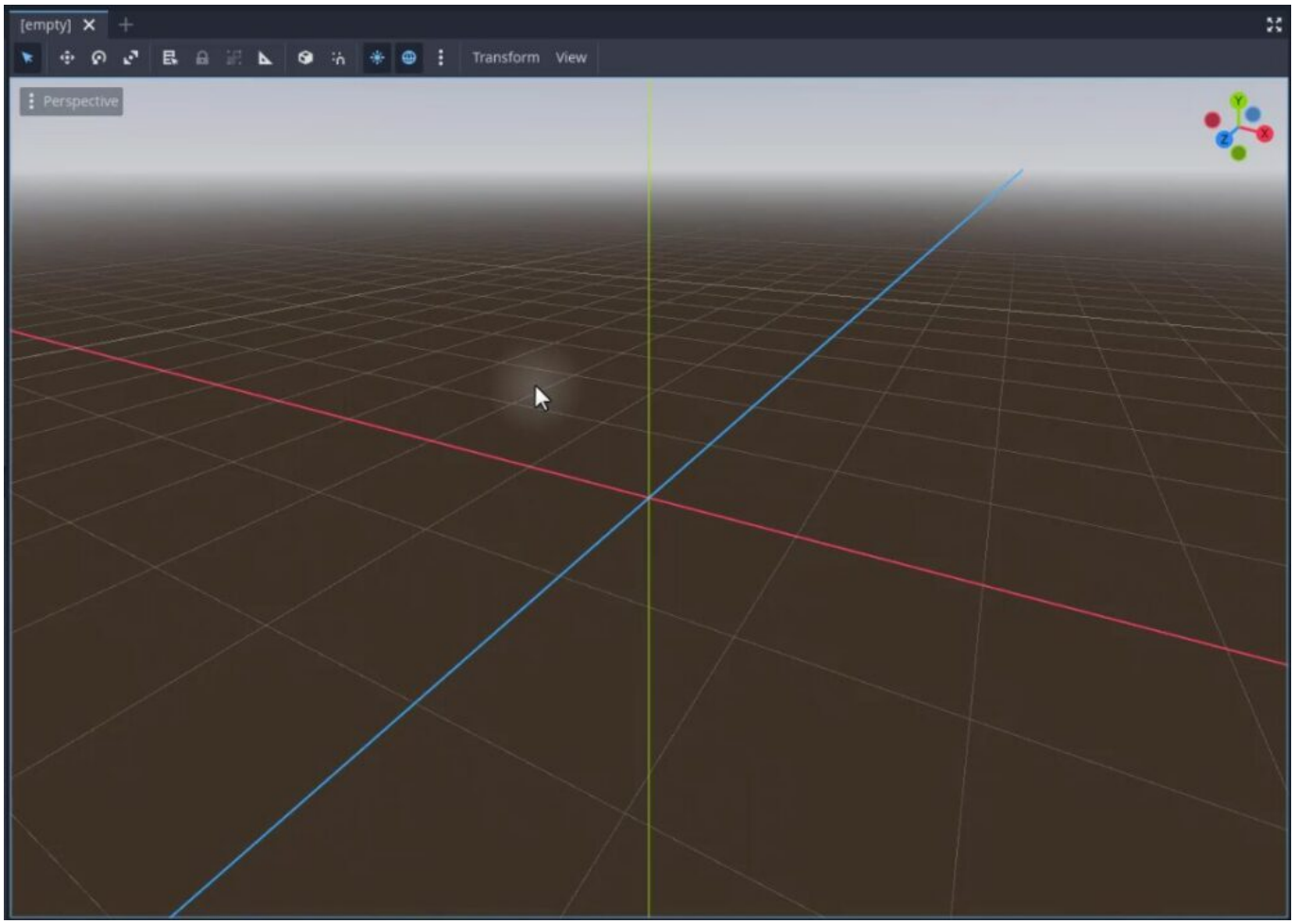
Located at the bottom left, the File System dock acts as a file explorer for your game project. Here you can:

- Navigate through folders and assets.
- Open and close folders by clicking on the little arrows.
- View and manage your game’s assets, such as scenes, scripts, textures, sound effects, and 3D models.



Central Viewport Dock

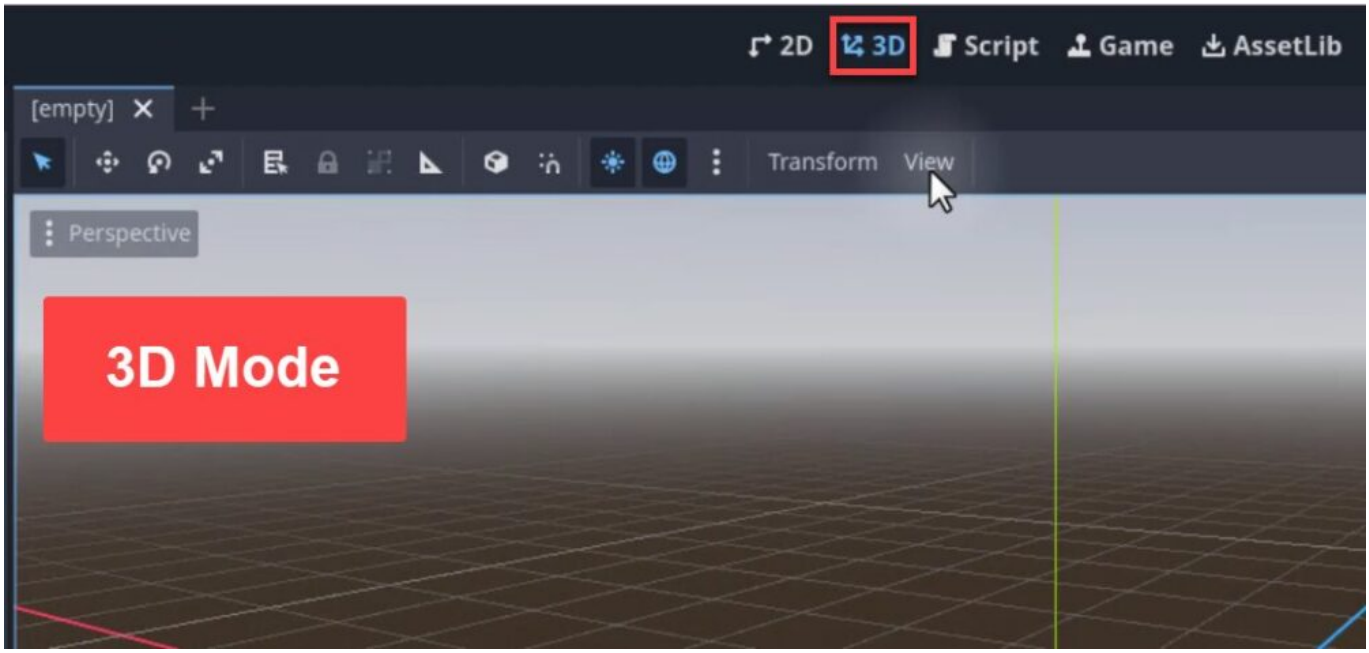
The central dock is the largest and most important area in the editor.

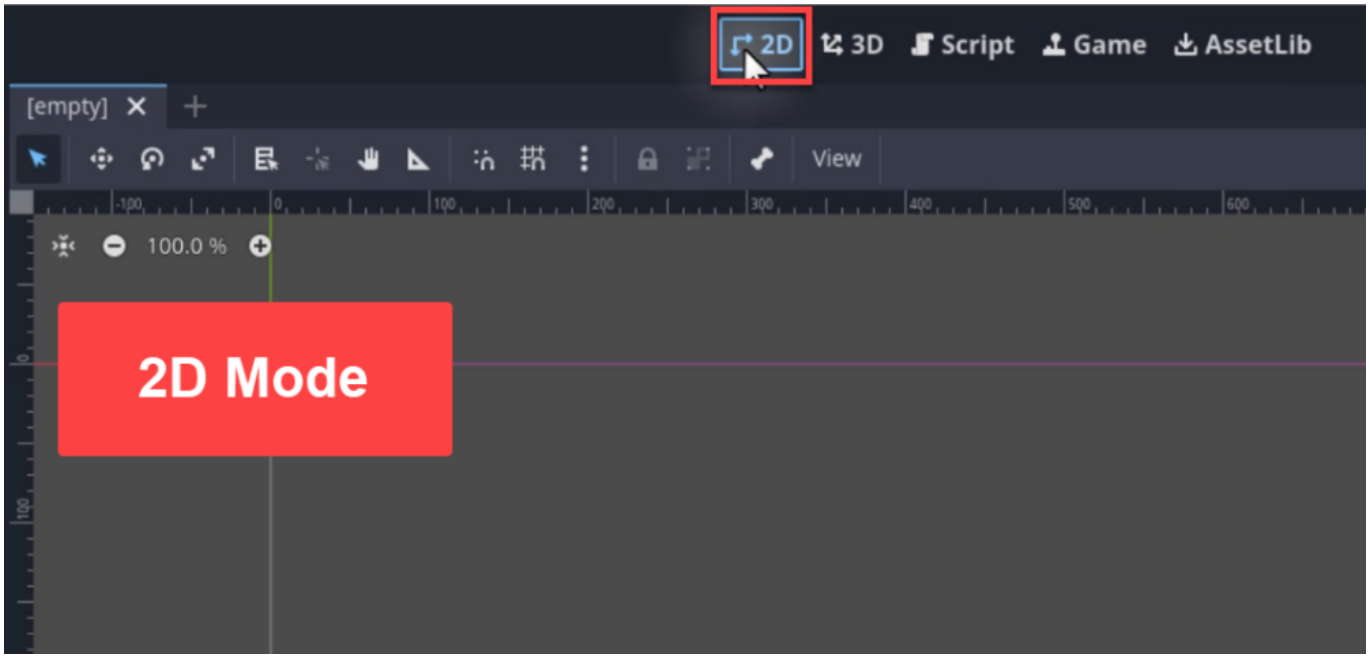


This is where you will develop your levels and design your game world. Key features include:

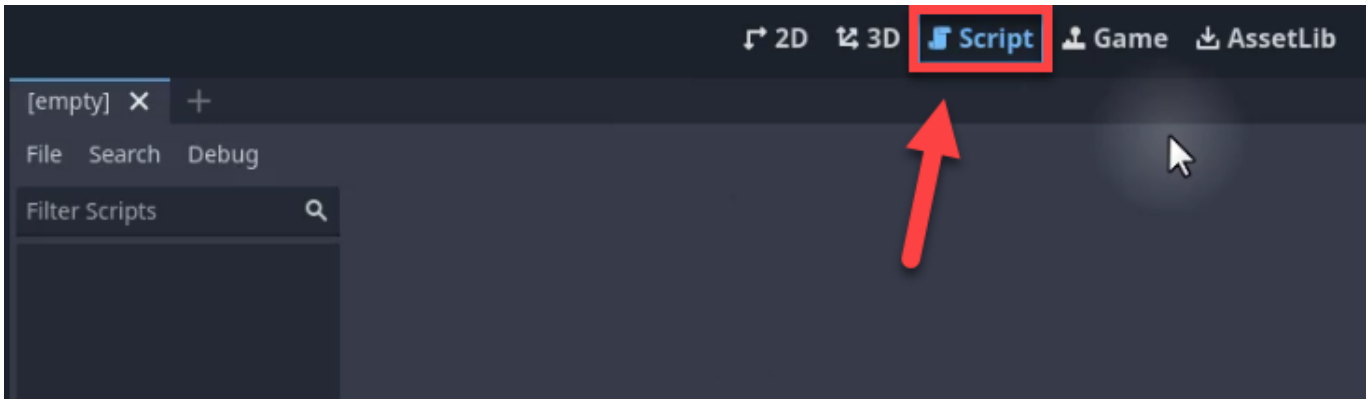
- Switching between 2D and 3D modes.
- Accessing the script editor for coding.
- Designing and positioning game elements.

To switch between 2D and 3D modes, use the buttons at the top of the screen.



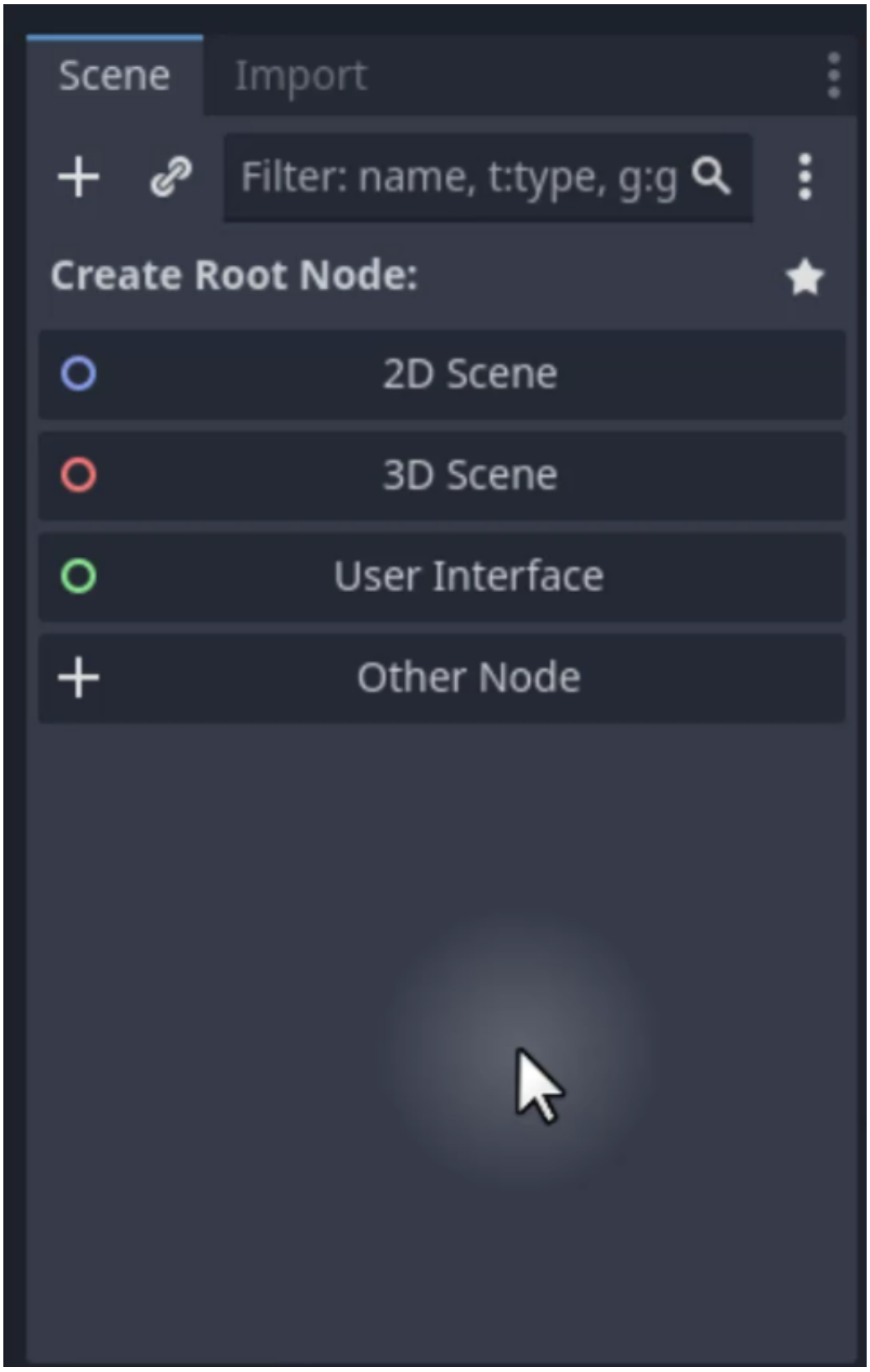


The script button will take you to Godot's built-in scripting editor. At the moment, this screen will be blank, however, as we don't have any scripts in our project.



Scene Dock

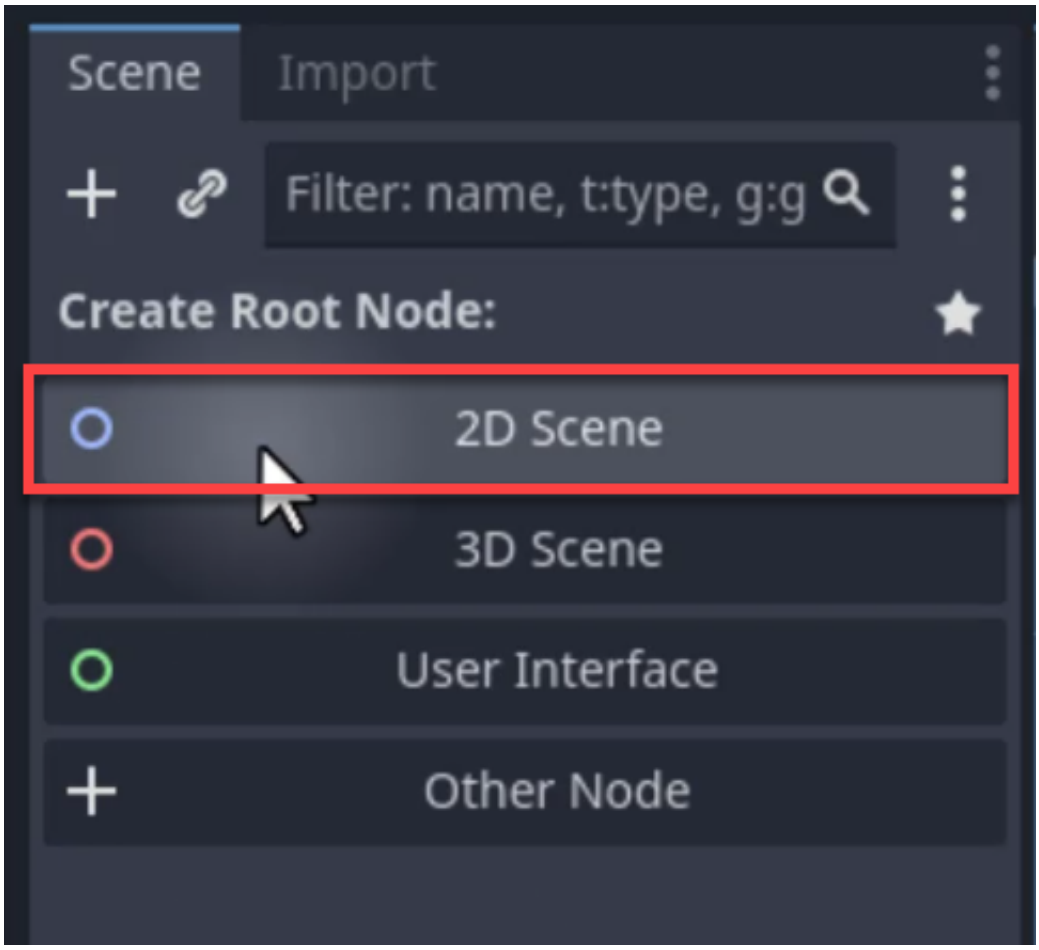
Located at the top left, the Scene dock displays all the nodes in your current scene.



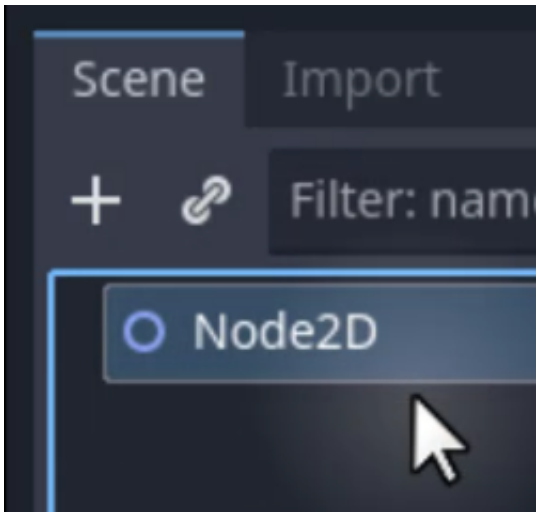
Key concepts include:

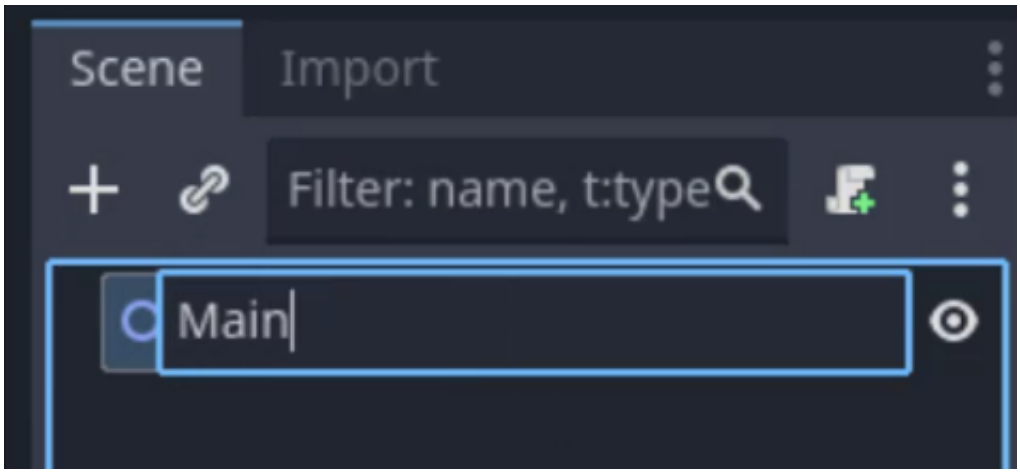
- **Nodes**: Building blocks for any element in your game, such as the camera, player, or particle effects.
- **Scenes**: Collections of nodes, similar to a level in your game.

To create a scene, you need to have a root node that acts as a container for all other nodes.



You can rename nodes by double-clicking on them. For example, we'll rename our "Node2D" node to "Main".



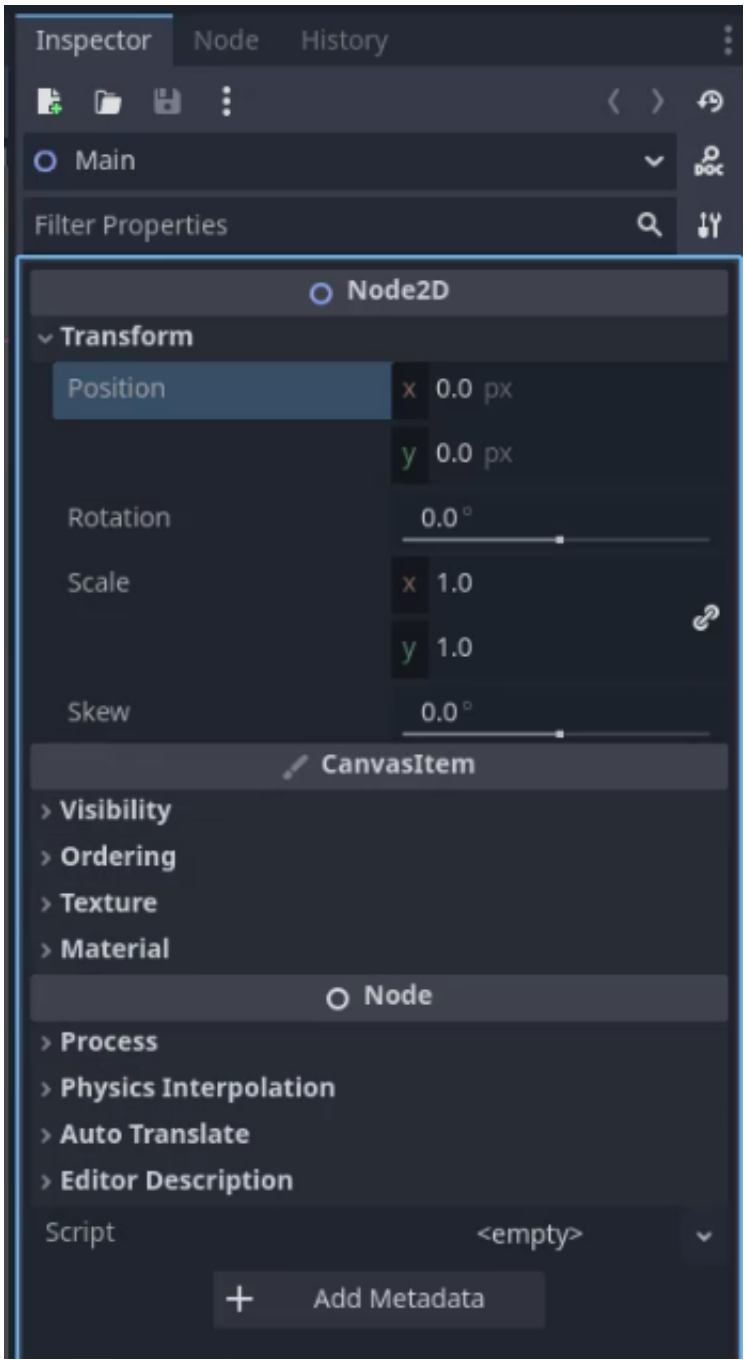


Inspector Dock

The Inspector dock, located on the right, allows you to view and modify the properties of the selected node. Key features include:

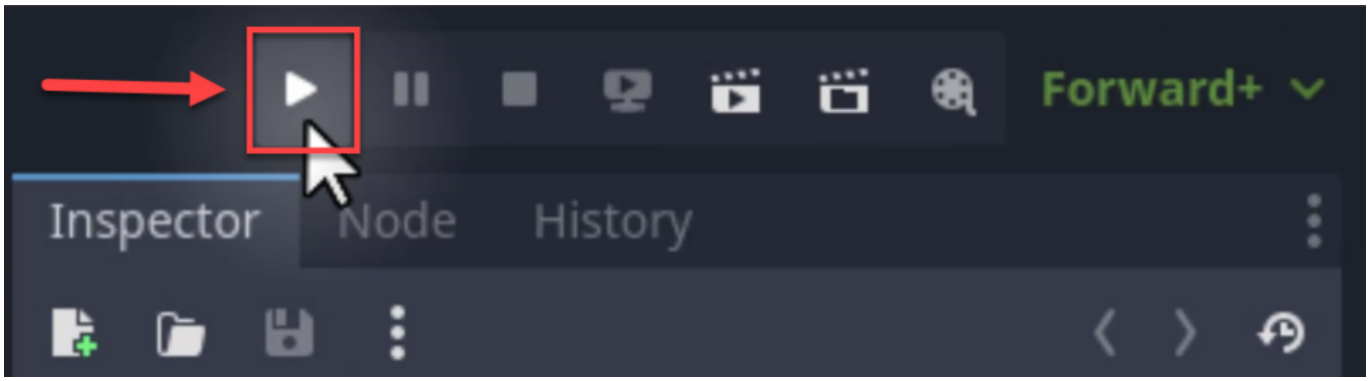
- Viewing and editing node properties like position, rotation, scale, and more.
- Customizing nodes to fit your game’s design.

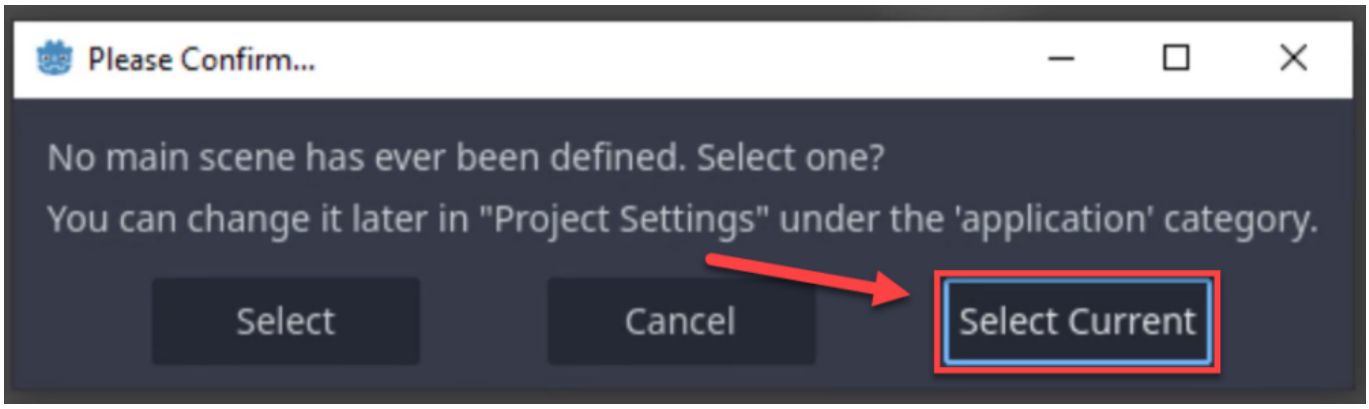
The Inspector dock has various categories of properties you can edit as needed for your project – though most you don’t have to worry about starting out.



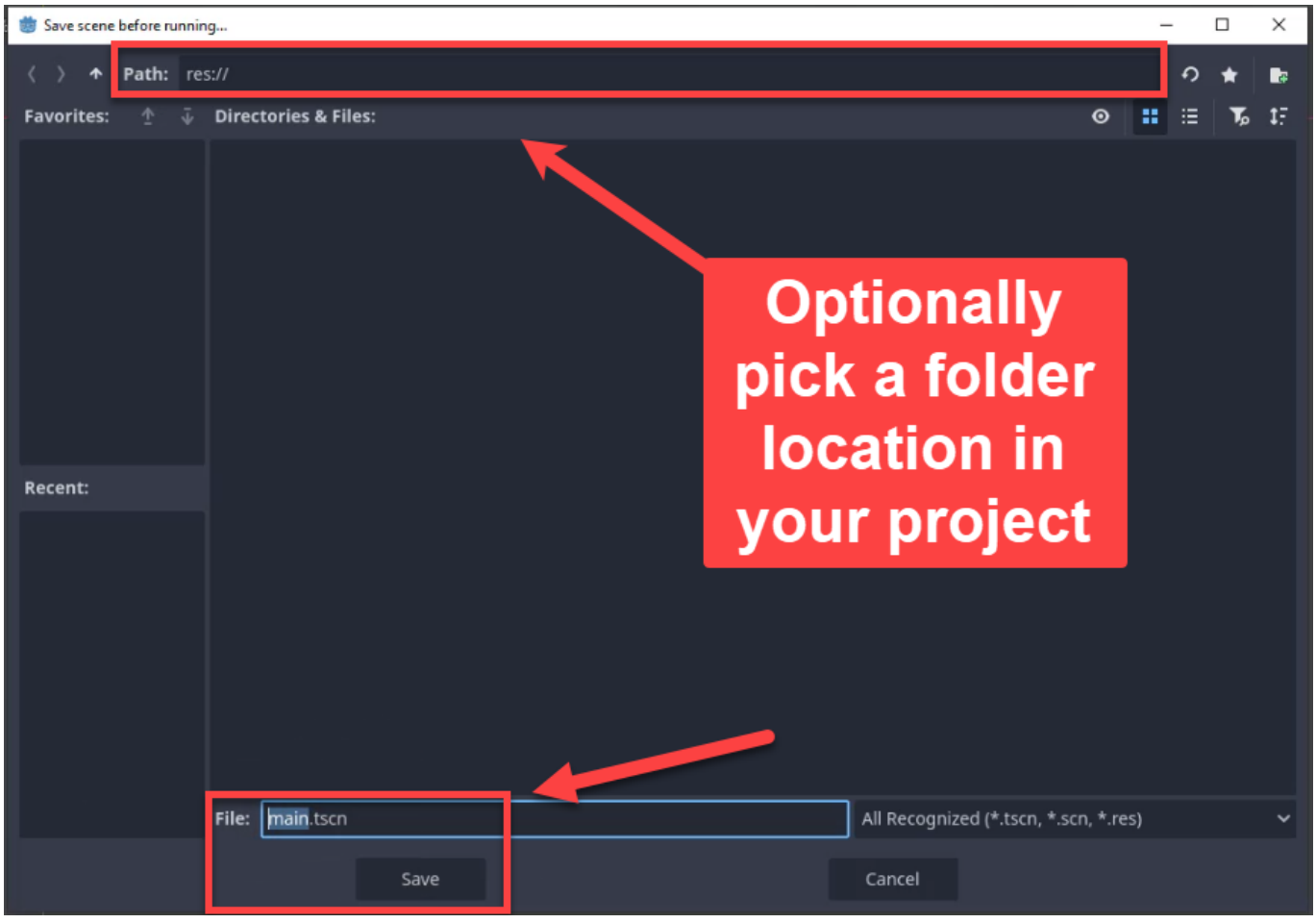
Running Your Game

To play your game, use the buttons in the top right corner of the screen. Click on **Run Project** and select a main scene.

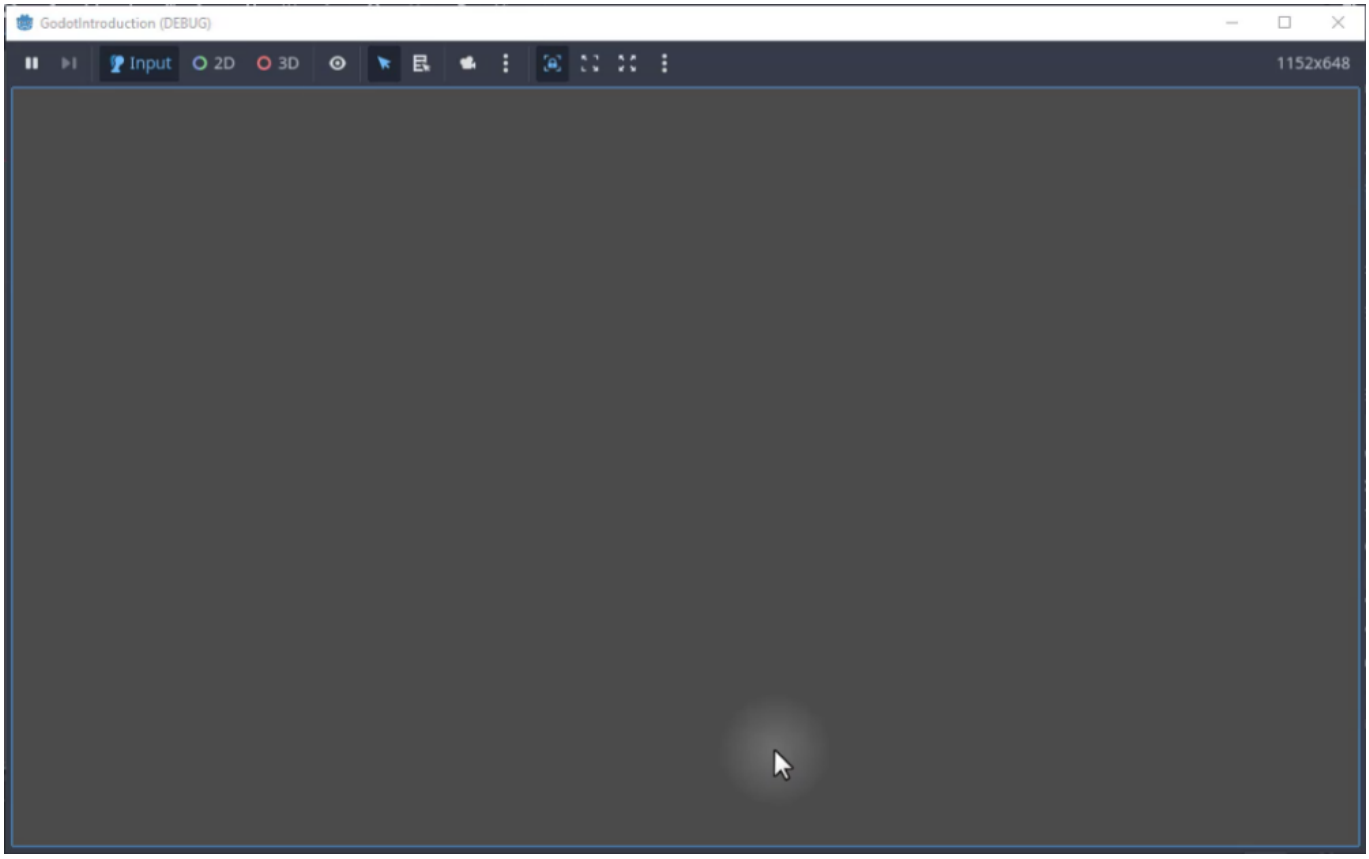




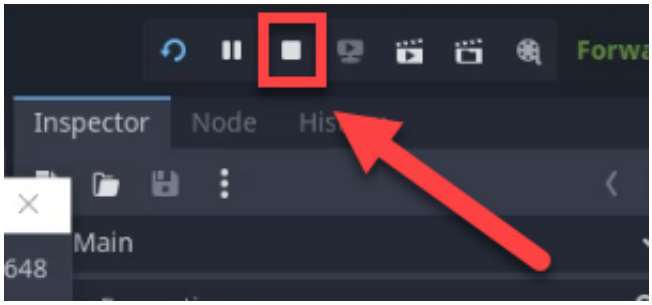
Save your scene when prompted.



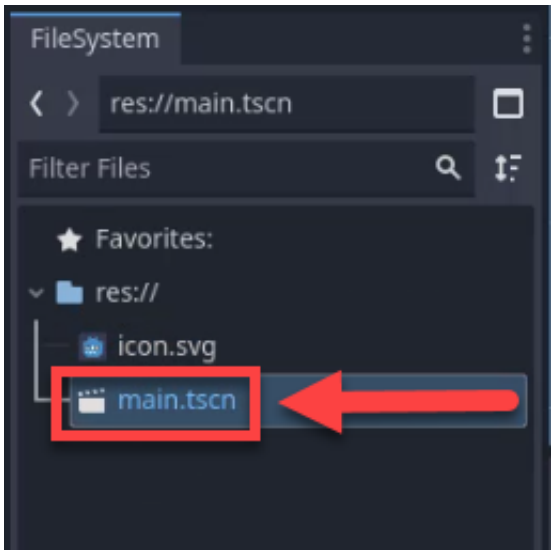
The game window will appear, rendering your game. As we have nothing in our scene, this game window will mostly be blank.



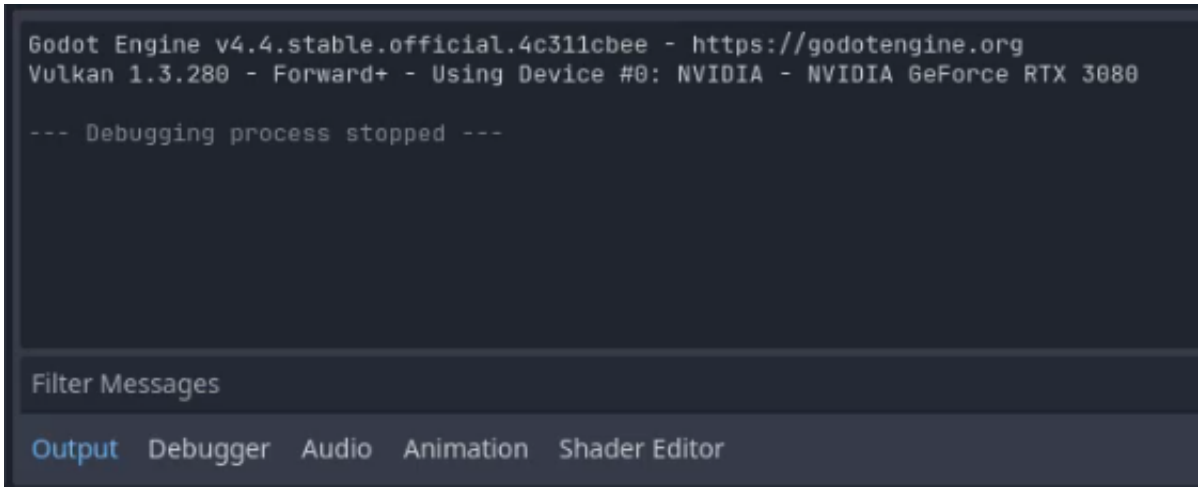
Click on the **Stop** button to stop playing your game.



The saved scene will be added to the File System dock. Scene files use the file extension **.tscn**.



Meanwhile, the Output dock at the bottom will display messages and errors. At the moment for us, it's mostly blank as there are no messages to display.



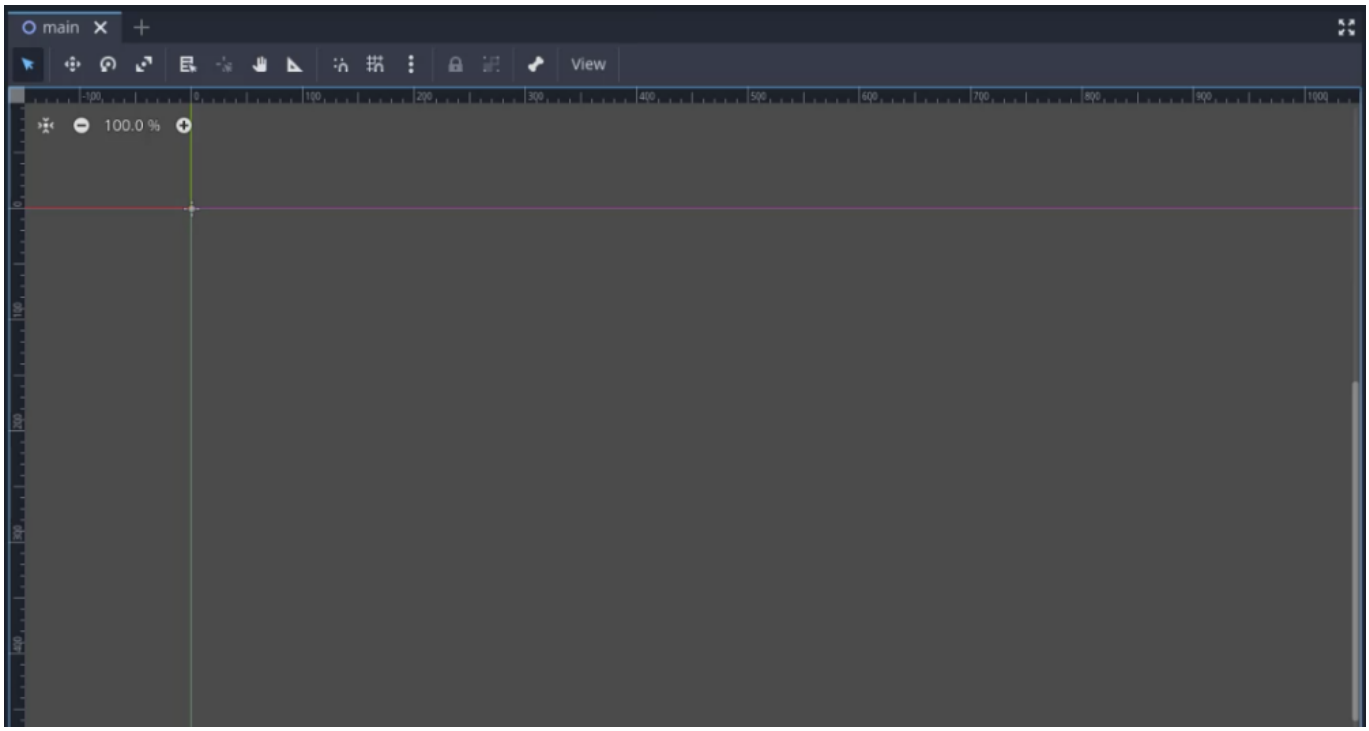
Conclusion

The Godot Editor is a powerful tool with various docks and functionalities to help you develop your games. By understanding and utilizing these docks, you can efficiently design, code, and test your game projects. Happy game development!

Welcome to this beginner-friendly guide on navigating scenes within the Godot engine. This article will walk you through the essential controls for both 2D and 3D viewports, enabling you to efficiently manage and build your game levels. Let’s dive right in.

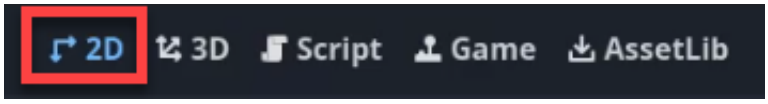
Understanding the Viewport

The main viewport in the Godot editor serves as your window into the game world. This is where you will place nodes, assets, and construct your levels. Effective navigation within this viewport is crucial for game development.



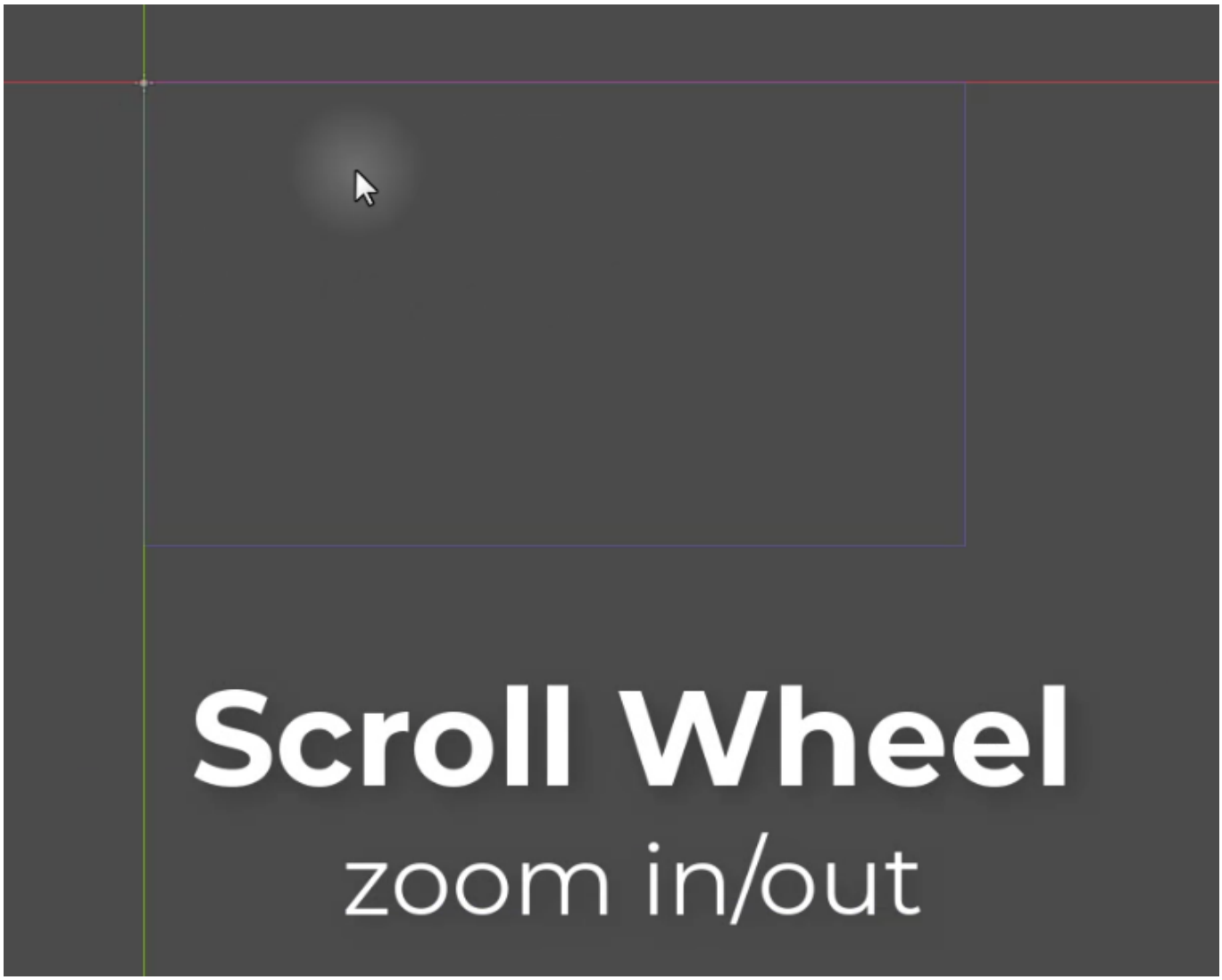
2D Navigation Controls

When working in 2D mode, you have several straightforward controls at your disposal.



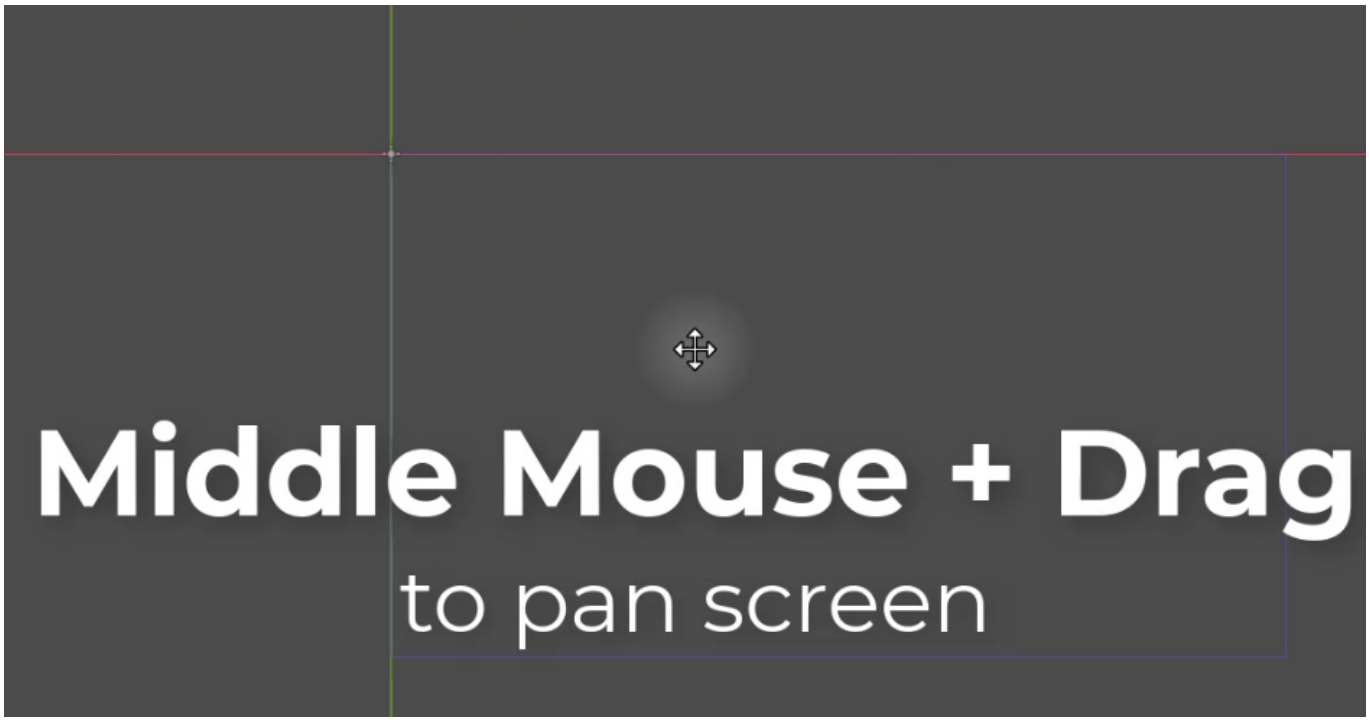
Zooming In and Out

Use the mouse scroll wheel to zoom in and out of your scene.



Panning

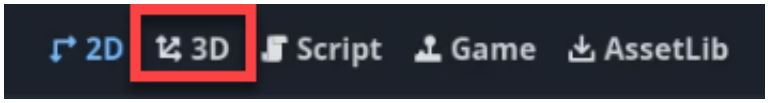
Hold down the middle mouse button and drag to pan across the scene.



Combining these controls allows you to navigate seamlessly around your 2D scene.

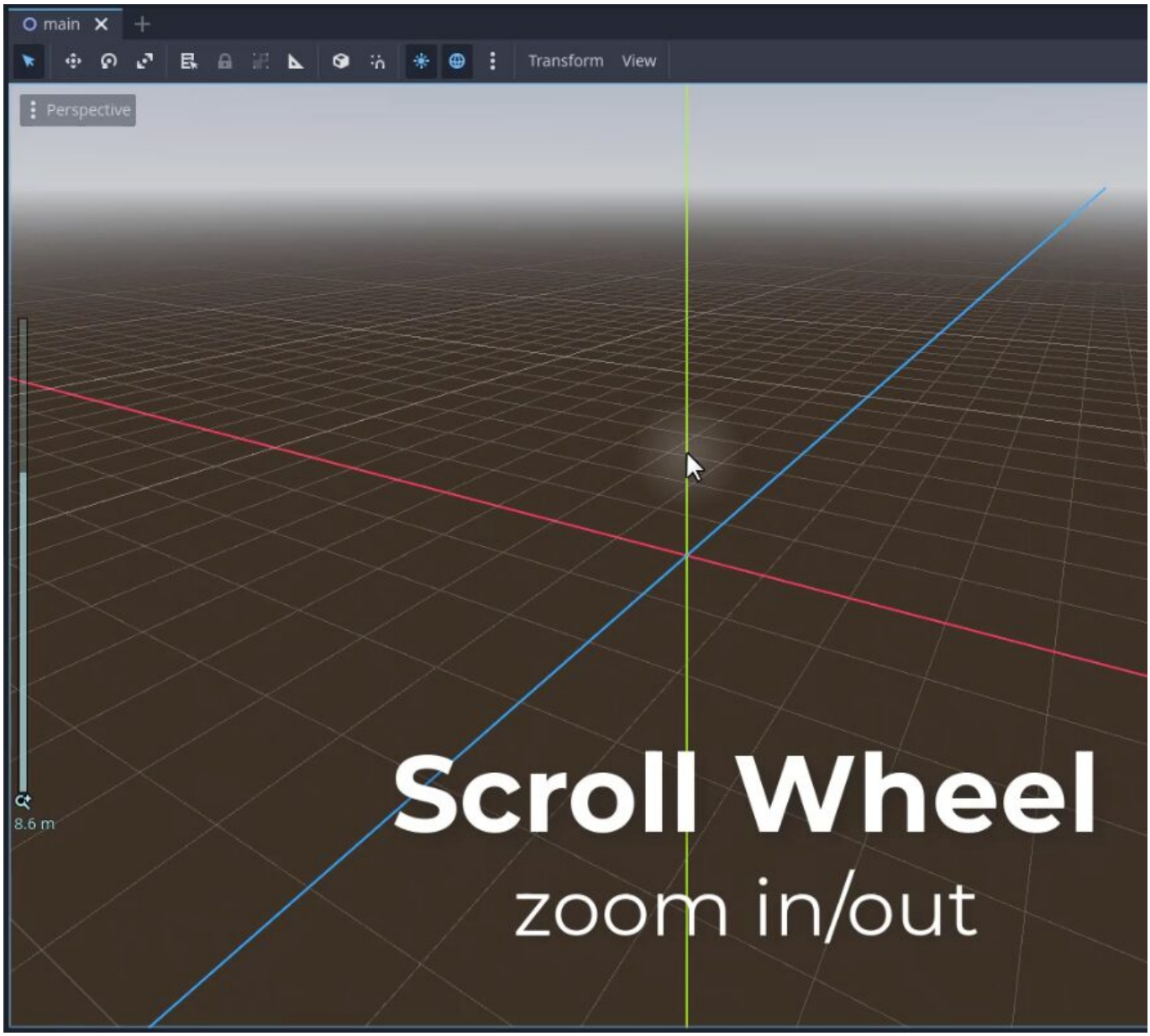
3D Navigation Controls

For 3D game development, the Godot engine offers additional controls to help you navigate the three-dimensional space effectively.



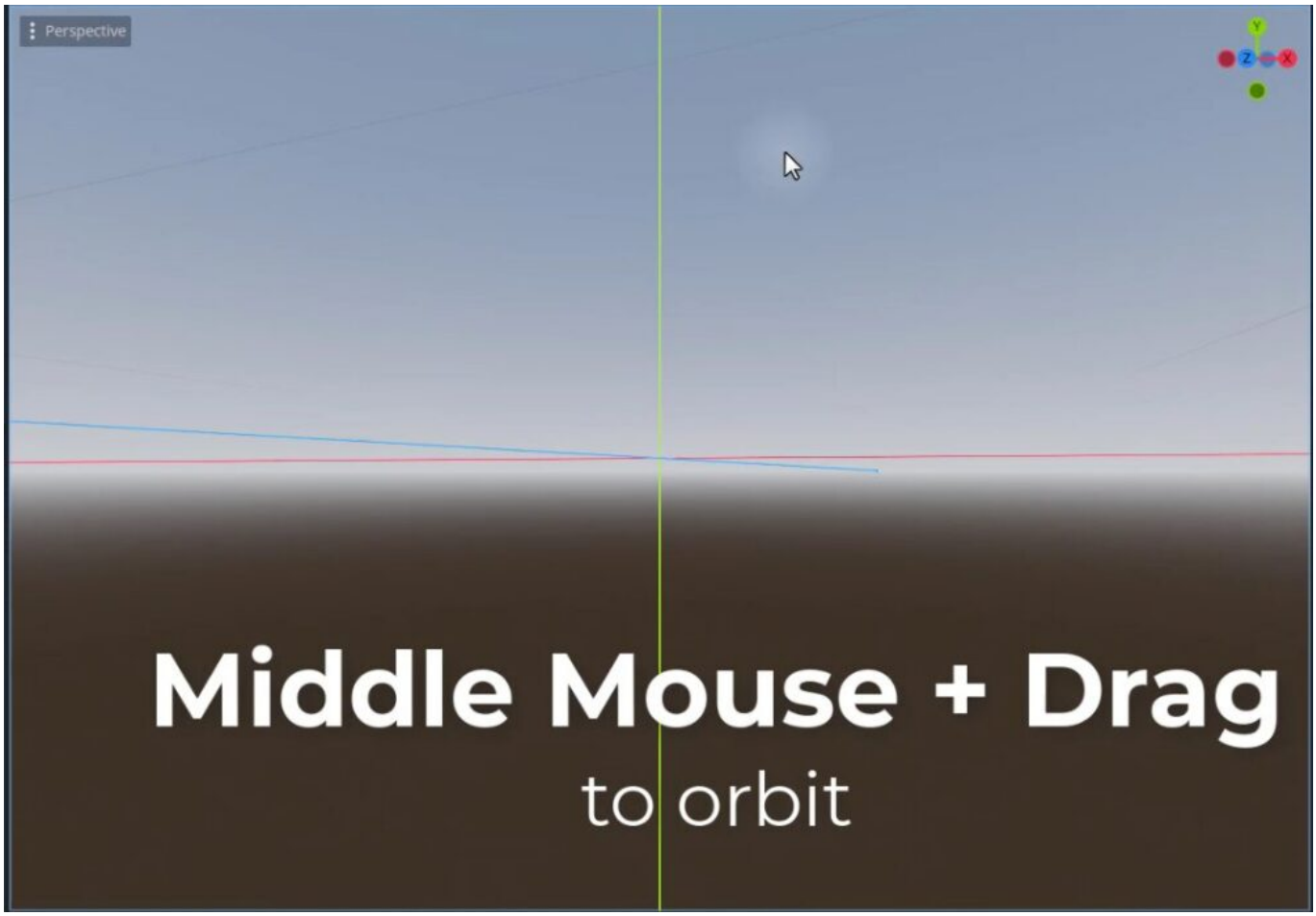
Zooming In and Out

Similar to 2D mode, use the mouse scroll wheel to zoom in and out.



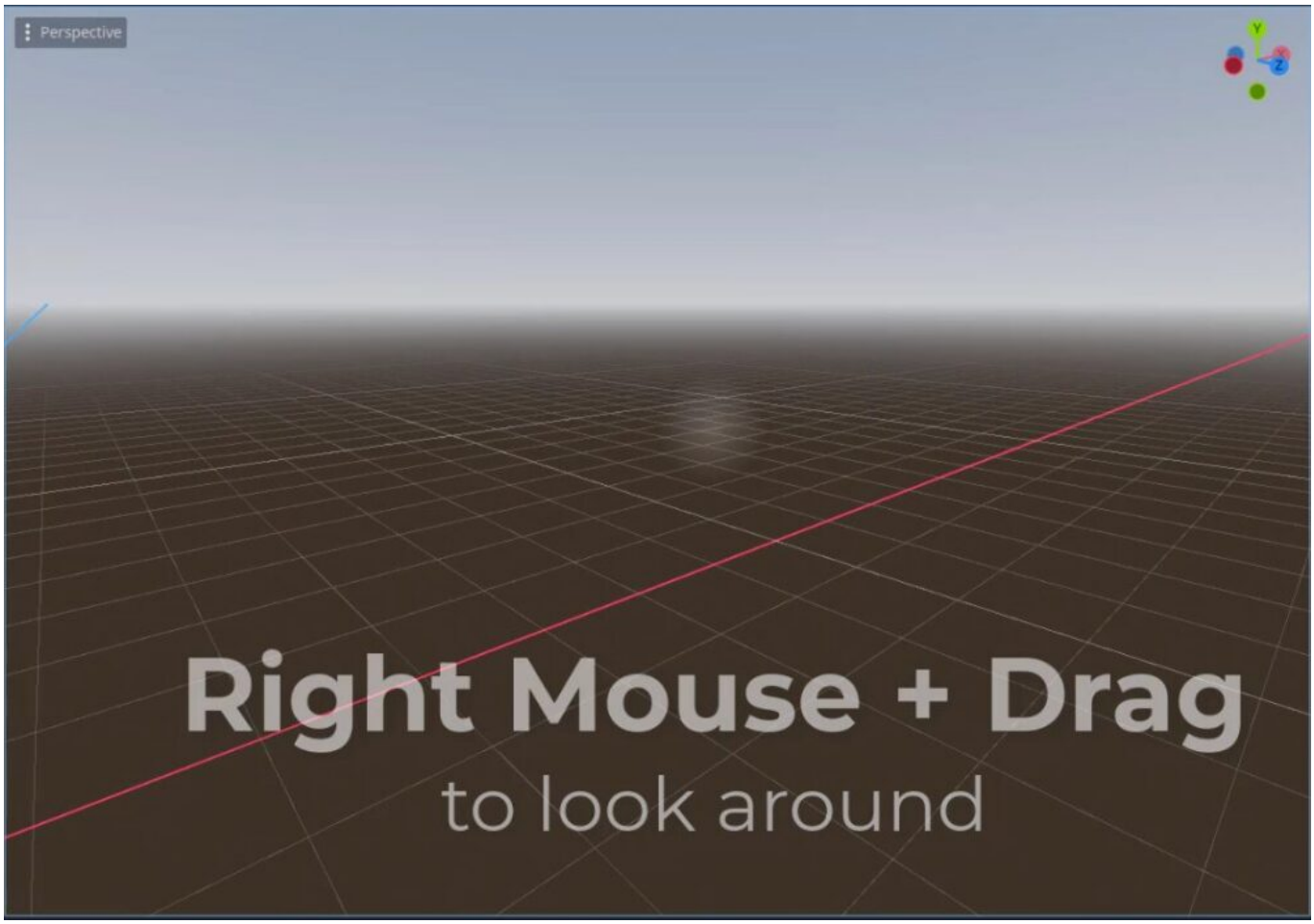
Orbiting

Hold down the middle mouse button and drag to orbit the camera around a point in 3D space.

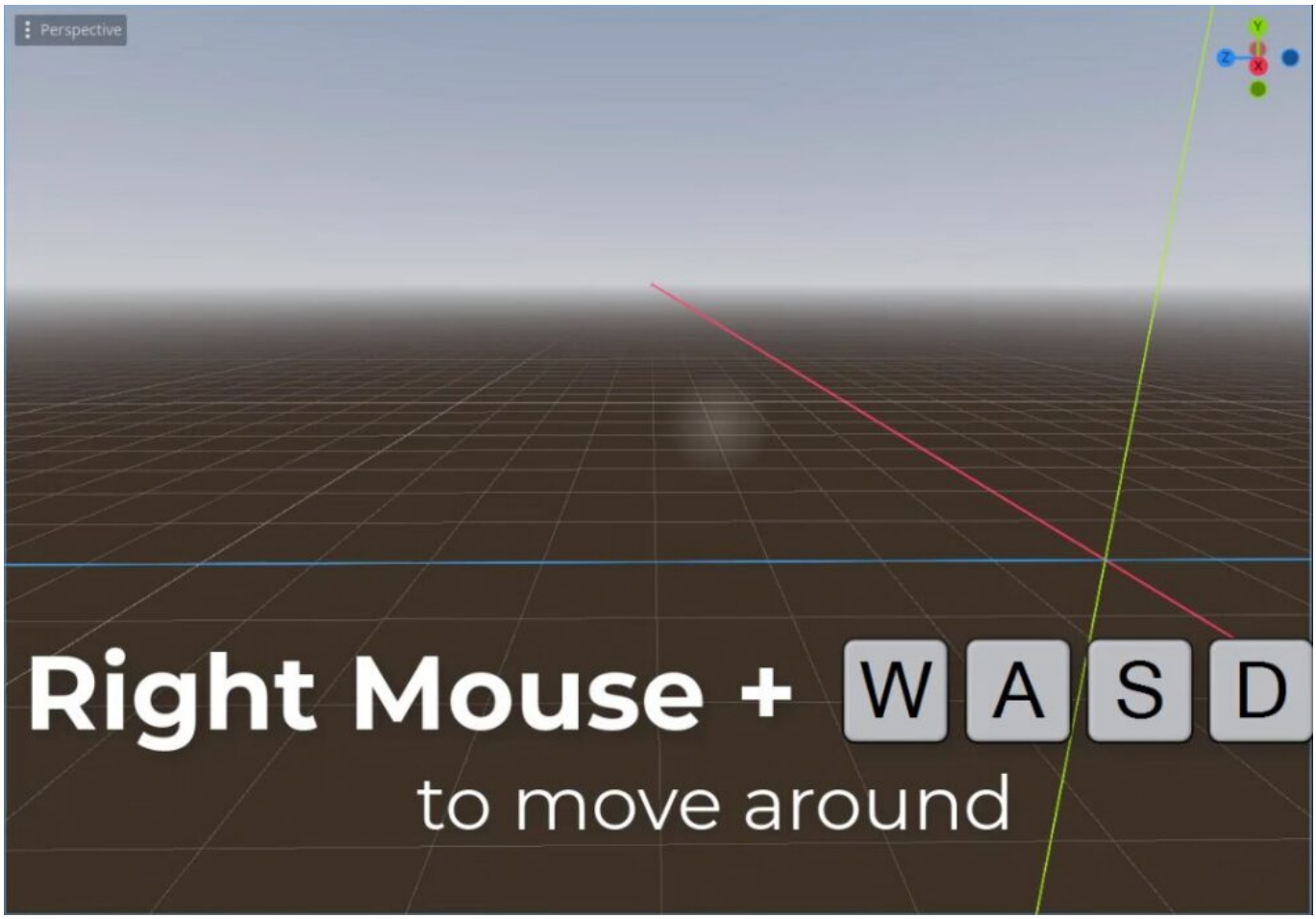


First-Person View

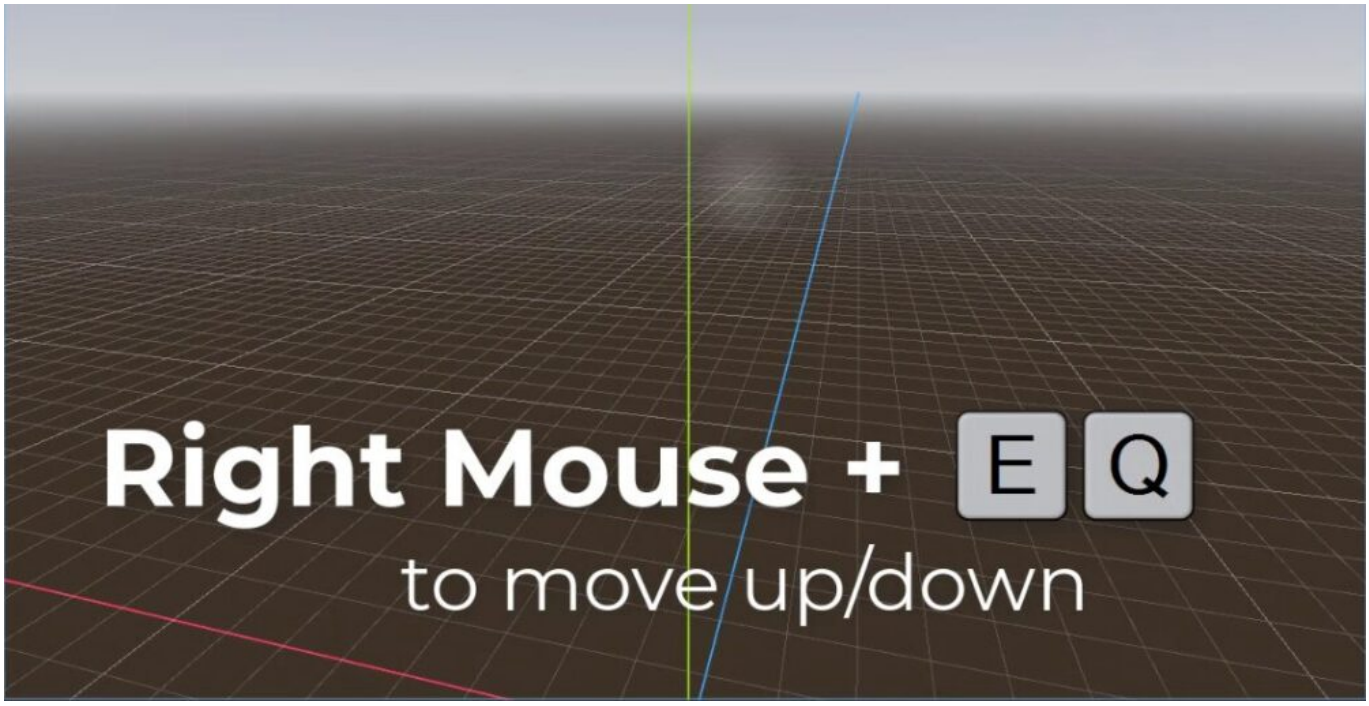
Hold down the right mouse button to enter a first-person view mode. In this mode, you can drag around the mouse to look around as if you were in a first-person video game.



Use the **W, A, S, D** keys to move around, similar to controlling a character in a video game.



Use the **E** key to move up and the **Q** key to move down.



Hold down the **Shift** key to move faster, akin to running.

Practical Applications

Mastering these navigation controls is essential for efficient game development:

- In 2D, zoom in to adjust fine details, zoom out to see the bigger picture, and pan to move between different parts of your level.
- In 3D, use the first-person view and orbiting controls to fly around your level, placing objects and inspecting different areas with ease.

Understanding and utilizing these navigation tools will significantly enhance your workflow within the Godot engine. Stay tuned for our next lesson, where we will explore importing assets into your game.

Thank you for reading, and happy game developing!

Welcome to our Godot shortcuts guide! Below, you will find the basic keyboard and mouse shortcuts that we use throughout our course. These are the main shortcuts you should know when starting out with Godot, helping you navigate and utilize the Godot Editor efficiently.

General Navigation

Action	Key
Pan Screen (2D)	Middle Mouse
Move Around	Right Mouse + WASD
Fast Move Around	Right Mouse + Shift + WASD
Focus on Node	F
Zoom In/Out	Mouse Scroll

Scene Navigation

Action	Key
Pan in 2D mode	Middle Mouse
Zoom in/out in 2D mode	Scroll Wheel
Rotate around the center	Middle Mouse (Hold in 3D mode)
Pan in 3D mode	Shift + Middle Mouse
Fly around in 3D mode	Right Mouse + WASD
Move upward in 3D mode	Right Mouse + E
Move downward in 3D mode	Right Mouse + Q

Editor Tools

Action	Key
Select Mode	Q
Move Mode	W
Rotate Mode	E
Scale Mode	R
Toggle Use Local Space	T

File Management

Action	Key
Save Scene/Script	Ctrl + S or CMD + S
New Scene/Script	Ctrl + N or CMD + N

Welcome to this beginner-friendly guide on importing assets into your Godot project. In this lesson, we will walk you through the process of understanding what assets are and how to import them into your game project using the Godot engine. By the end of this article, you will be able to confidently manage and utilize various types of assets within your Godot projects.

Understanding Assets

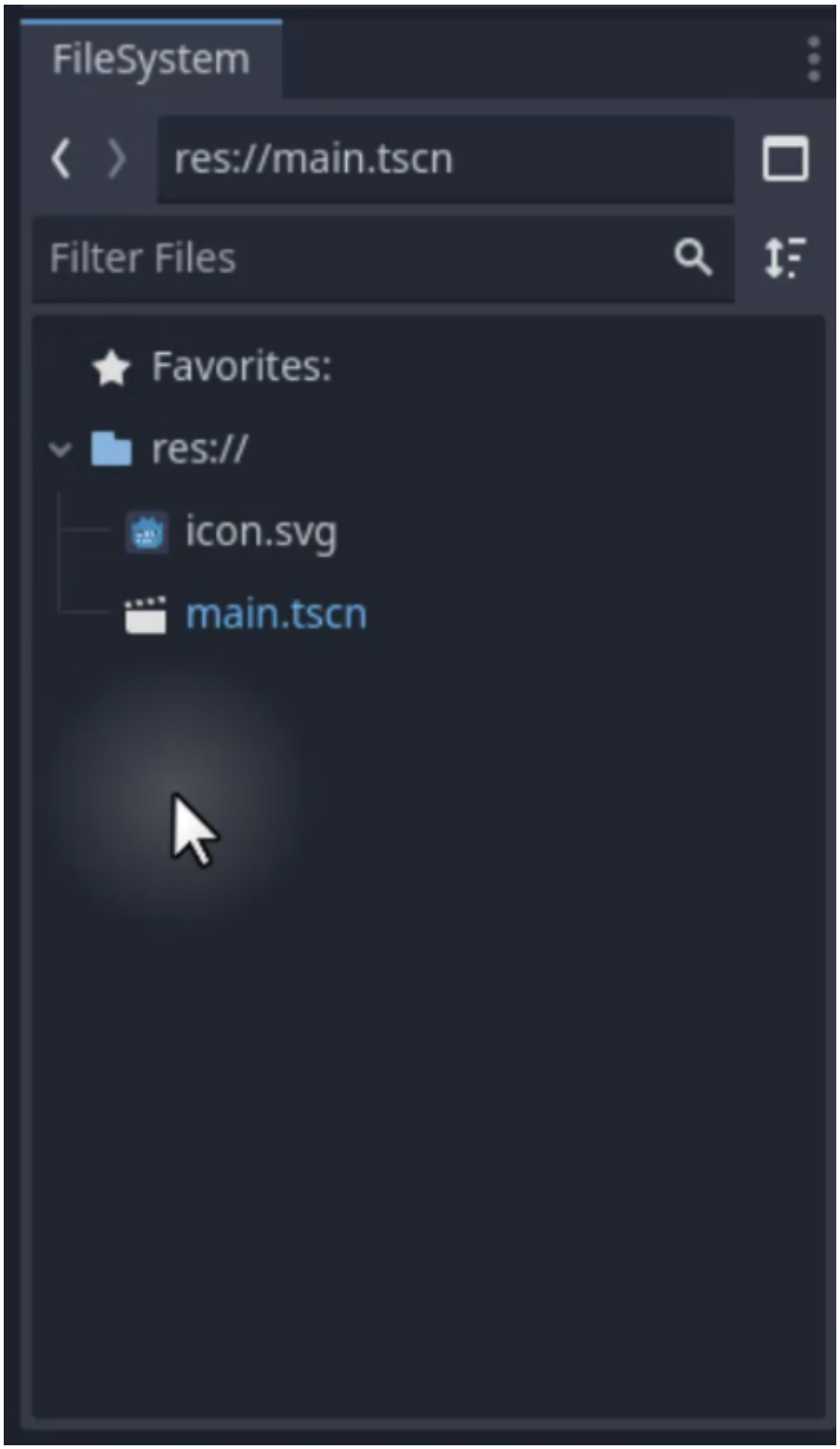
Before we dive into the importing process, let’s clarify what an asset is. An asset refers to any file that you intend to use in your game project. This includes, but is not limited to:

- Textures
- Sprites
- Sound effects
- Music
- 3D models
- Materials

Essentially, any file that contributes to the visual, auditory, or interactive elements of your game is considered an asset.

Navigating the FileSystem

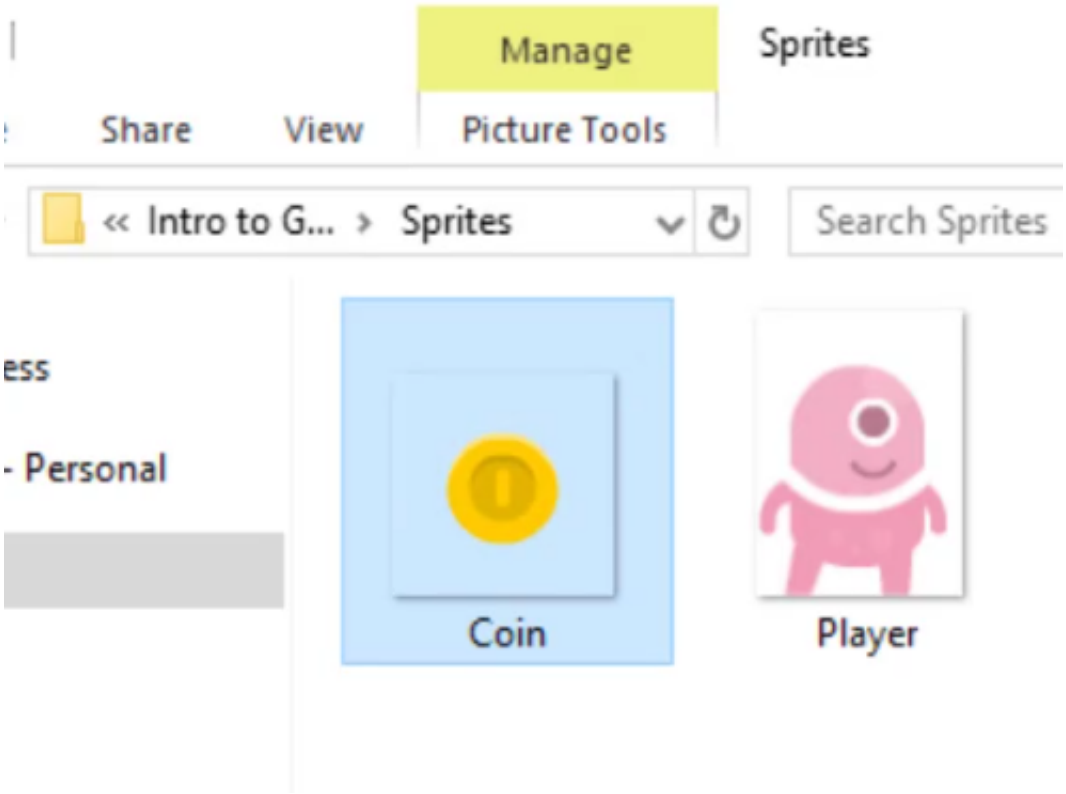
The FileSystem in Godot acts as a file explorer, displaying all the folders and assets within your project. This makes it easy to organize and access your assets as needed.



Downloading and Extracting Assets

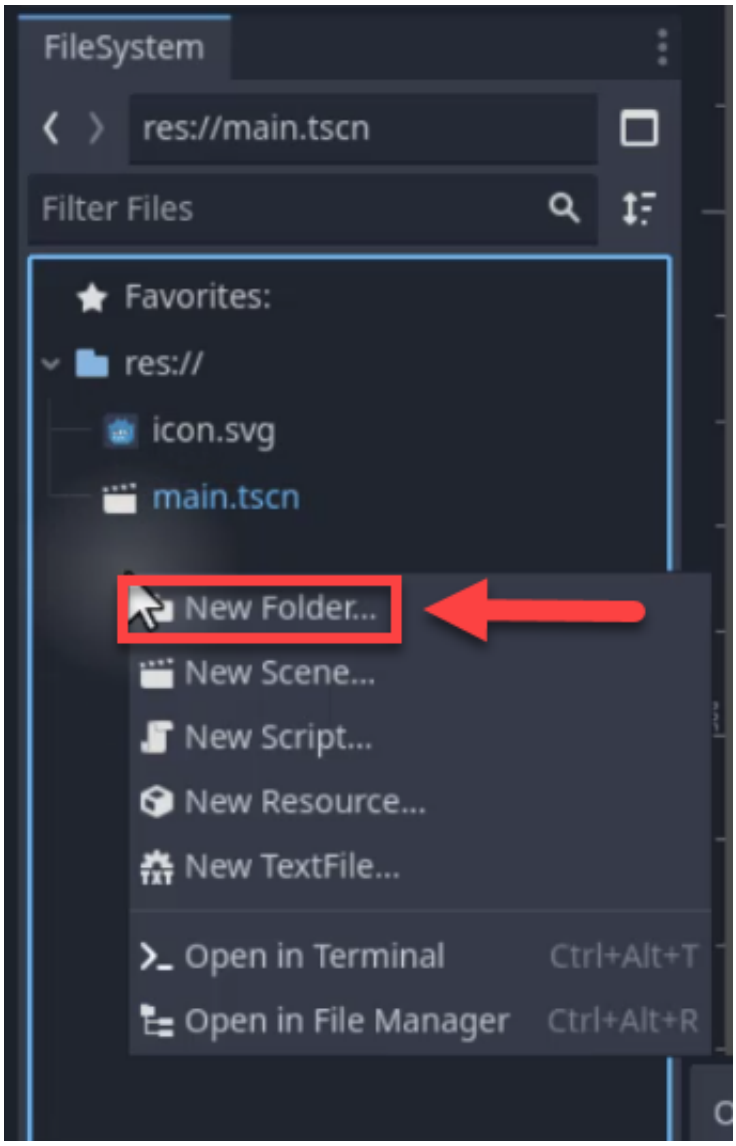
Let’s get some assets to import. Navigate to the course files tab of this course and download the provided ZIP file containing the assets for this course. Extract the contents of the ZIP file to your computer.

Upon extraction, you will find a folder named ‘Sprites’ containing two images: a coin and a player image. These images will be used as sprites in our game.

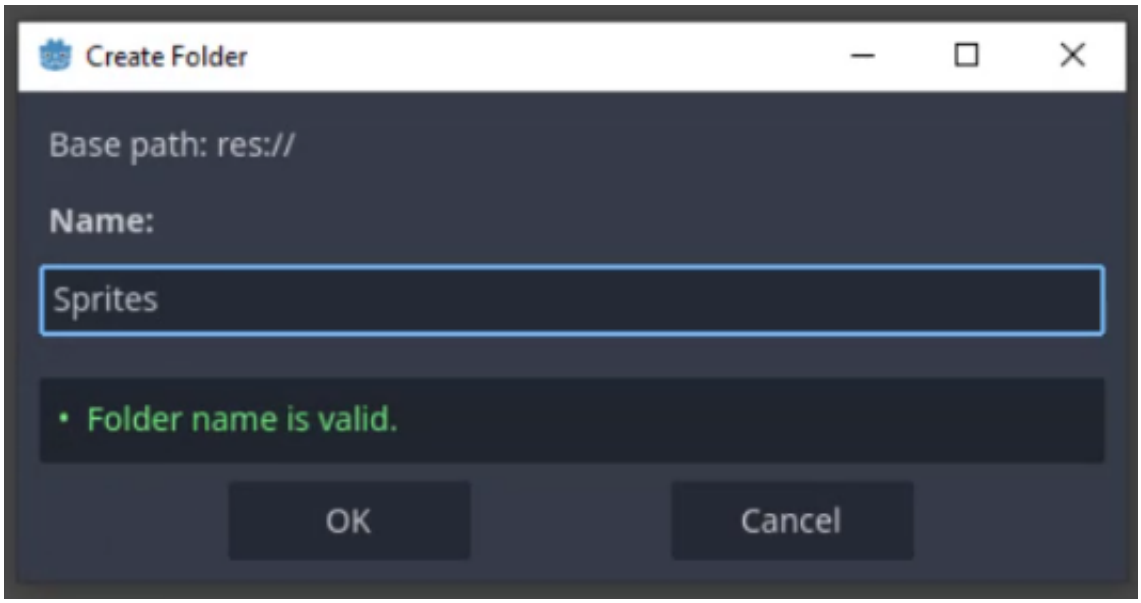


Importing Assets into Your Project

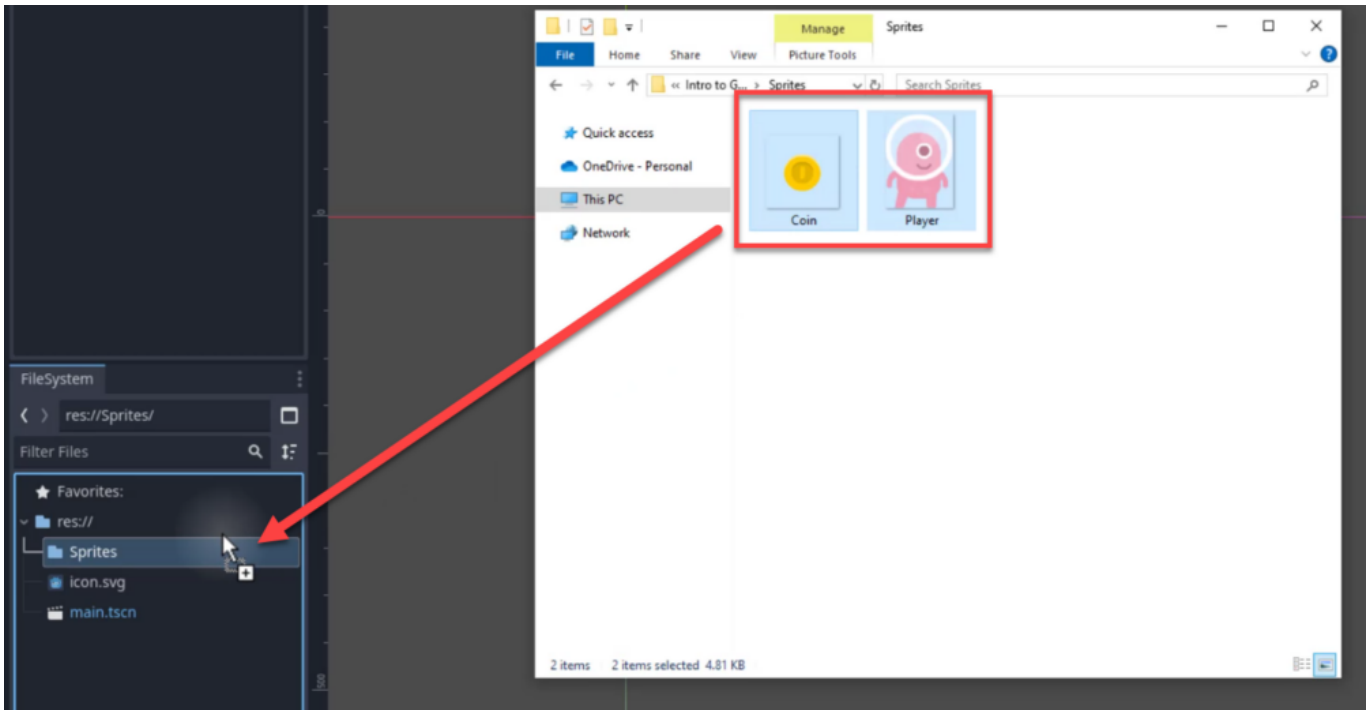
Let’s import our sprites now. First, in the FileSystem panel of your Godot project, we need to create a new folder to store your sprites. Right-click in the FileSystem panel and select ‘New Folder’.



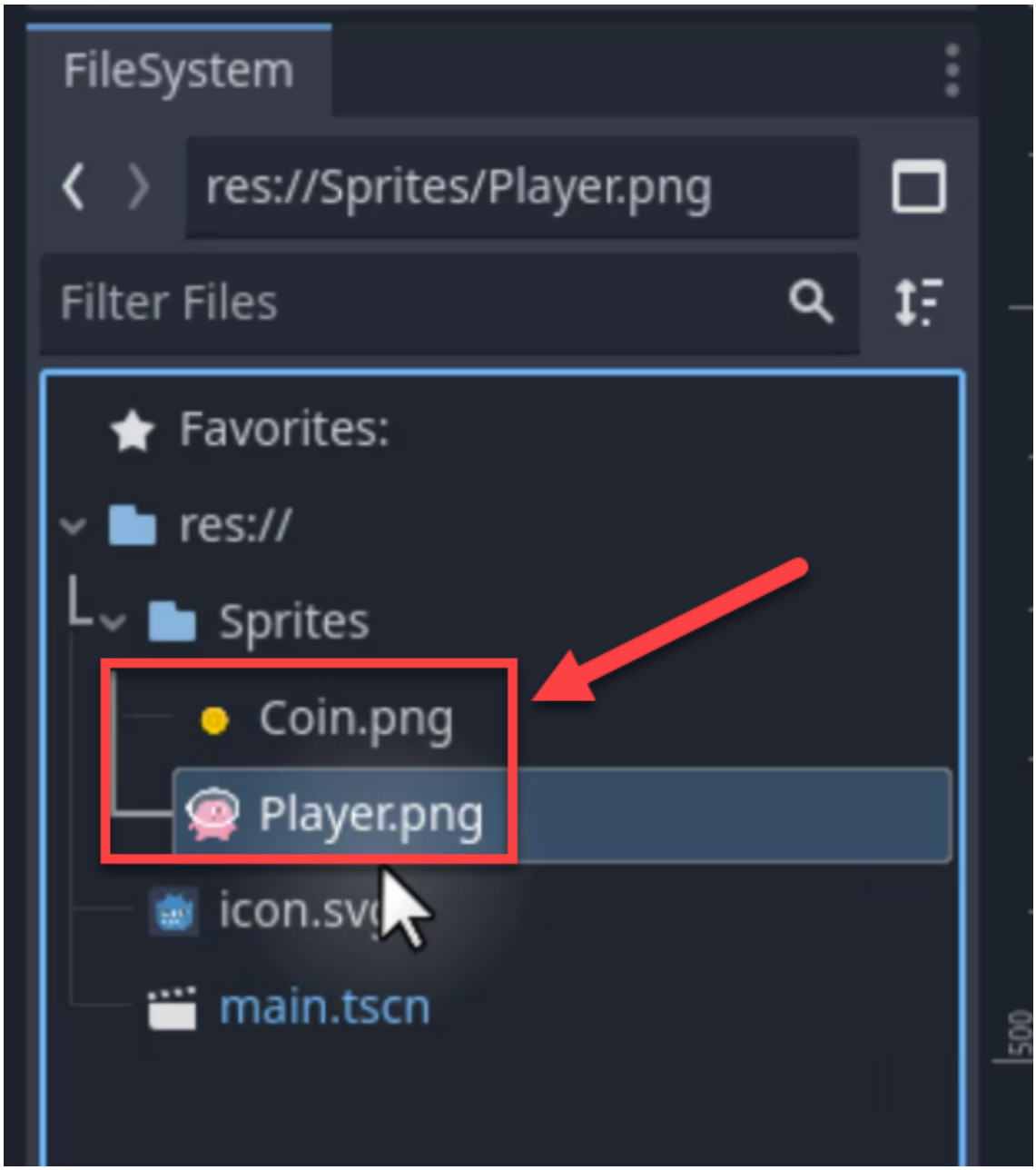
Name the folder ‘Sprites’ and click OK.



Next, open your computer's file explorer and navigate to the location where you extracted the sprites. Select both the coin and player images, and drag the selected images into the 'Sprites' folder in the Godot FileSystem panel. You will notice the mouse cursor changes to an import icon.

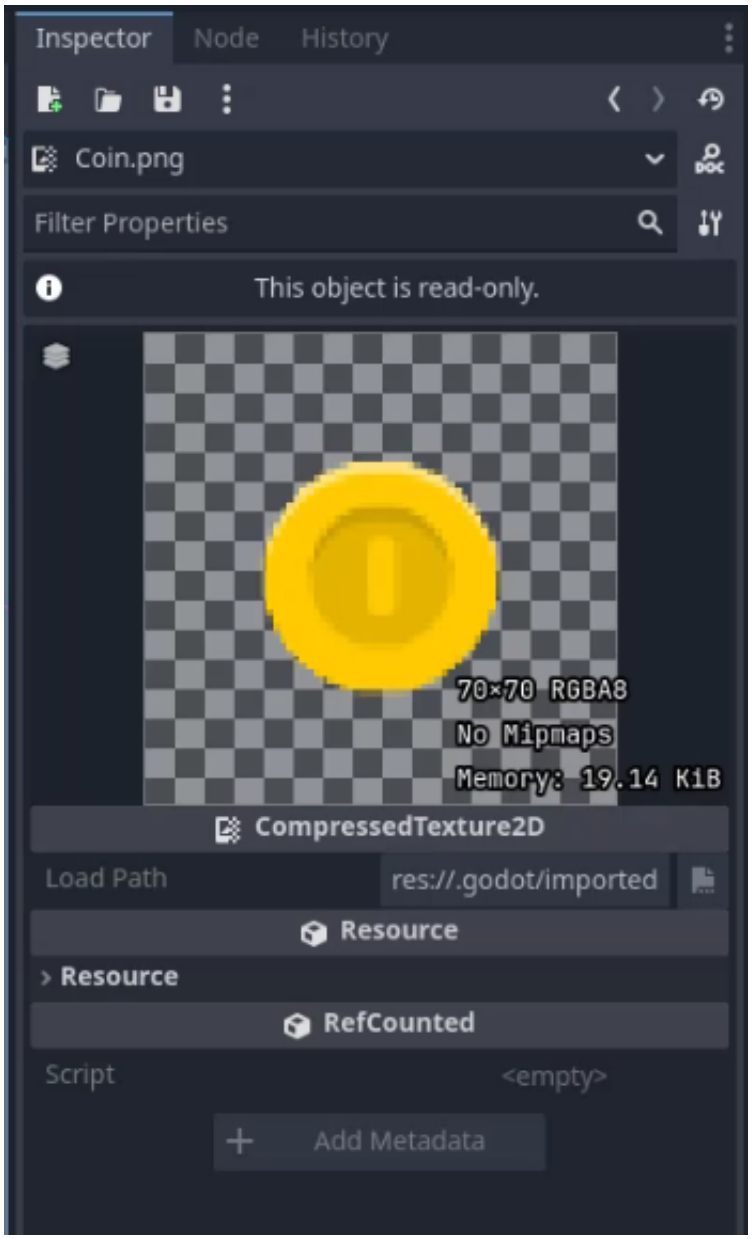


Release the mouse button to initiate the import process. Once the import is complete, you will see a progress bar indicating the status. Upon completion, the 'sprites' folder will display the imported coin and player PNG files.



Inspecting Imported Assets

To view more details about an imported asset, double-click on the asset in the FileSystem panel. The asset will open in the inspector, displaying the image and its properties, such as resolution and other settings.



Importing Other Types of Assets

The process of importing other types of assets, such as audio clips or 3D models, follows a similar procedure. Simply drag the desired files into the appropriate folder within the FileSystem panel to import them into your project.

Congratulations! You have successfully learned how to import assets into your Godot project. This skill is fundamental to game development and will enable you to create rich and engaging experiences. Happy game developing!

Welcome to this beginner-friendly guide on understanding nodes and scenes within the Godot engine. In this article, we will explore the fundamental concepts of nodes and scenes, which are the building blocks of any Godot game. By the end of this guide, you will have a clear understanding of what nodes and scenes are, how to create them, and how to manipulate their properties.

Understanding Nodes

Nodes are the basic units that make up a Godot game. Similar to how atoms form the building blocks of our world, nodes constitute everything within the Godot engine. A node can represent various elements in your game, such as:

- The player character
- Items or weapons
- Environmental objects like rocks or trees
- Enemies or non-player characters (NPCs)
- Terrain and landscape features

Nodes are highly versatile and can be used to create virtually any distinct element in your game. Depending on the complexity of your game, you might have anywhere from a few dozen to thousands of nodes.

Understanding Scenes

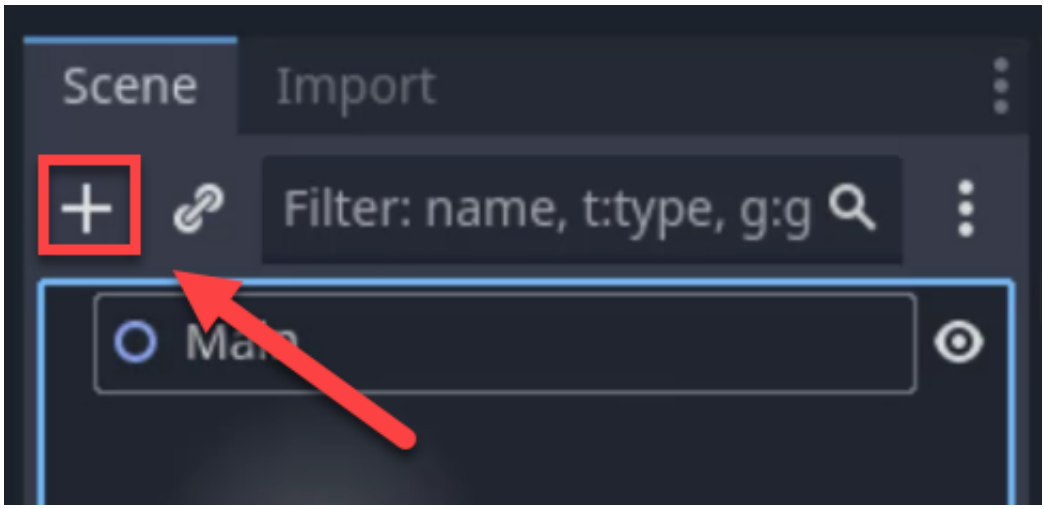
Scenes are collections of nodes. You can think of a scene as a level or a specific part of your game. For example, a scene might include:

- The player character
- The environment
- Enemies or NPCs
- An end gate or portal to the next level
- The root node which contains all other nodes of your level

All these nodes are collected into one scene, which can then be loaded into your game. Scenes can also act as nodes themselves, making Godot a highly modular engine. This means you can create reusable scenes, such as a player scene, which can be instantiated as a node whenever needed.

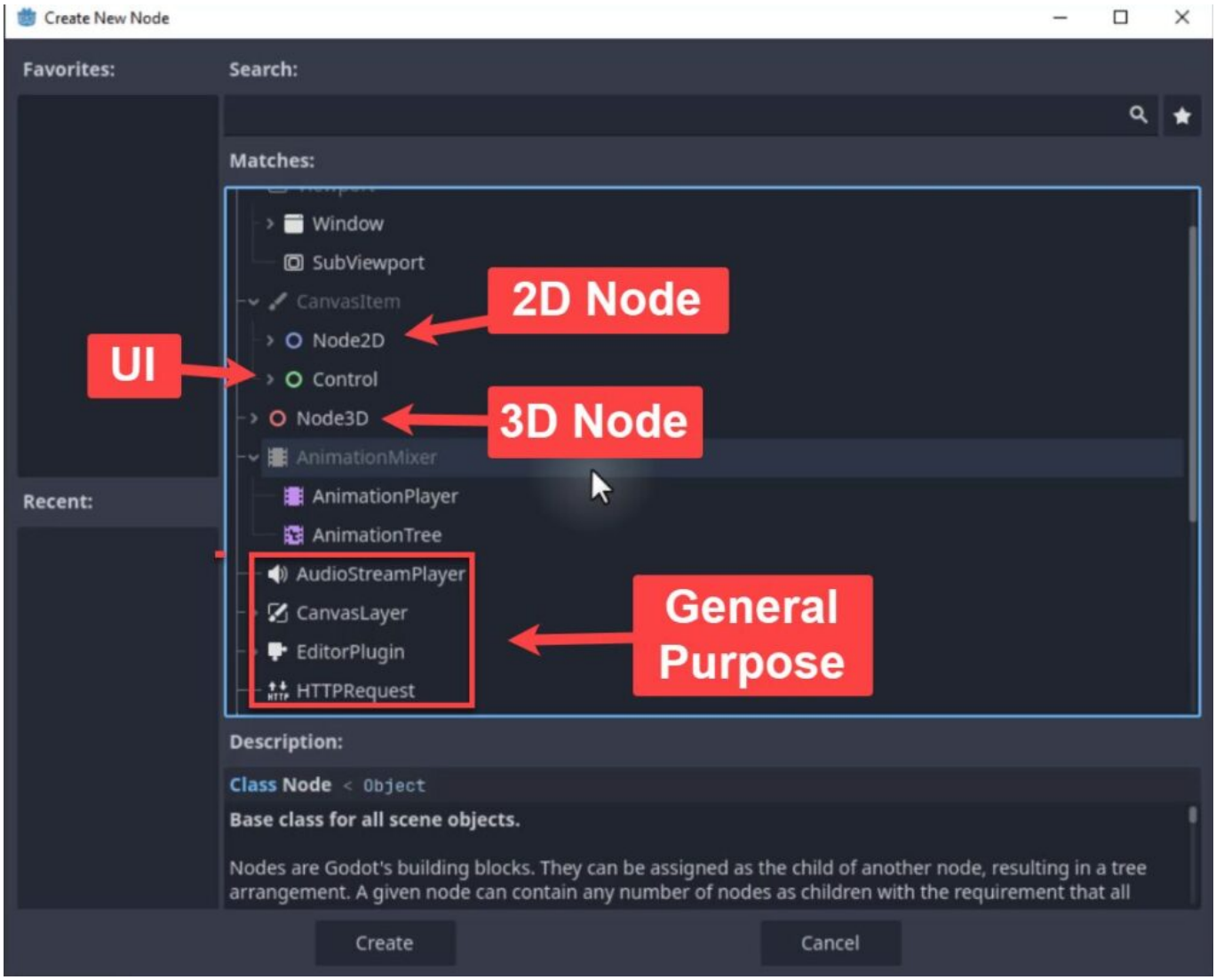
Creating and Managing Nodes

To create a new node, click on the plus icon in the Scene dock to open the ‘Create New Node’ window.

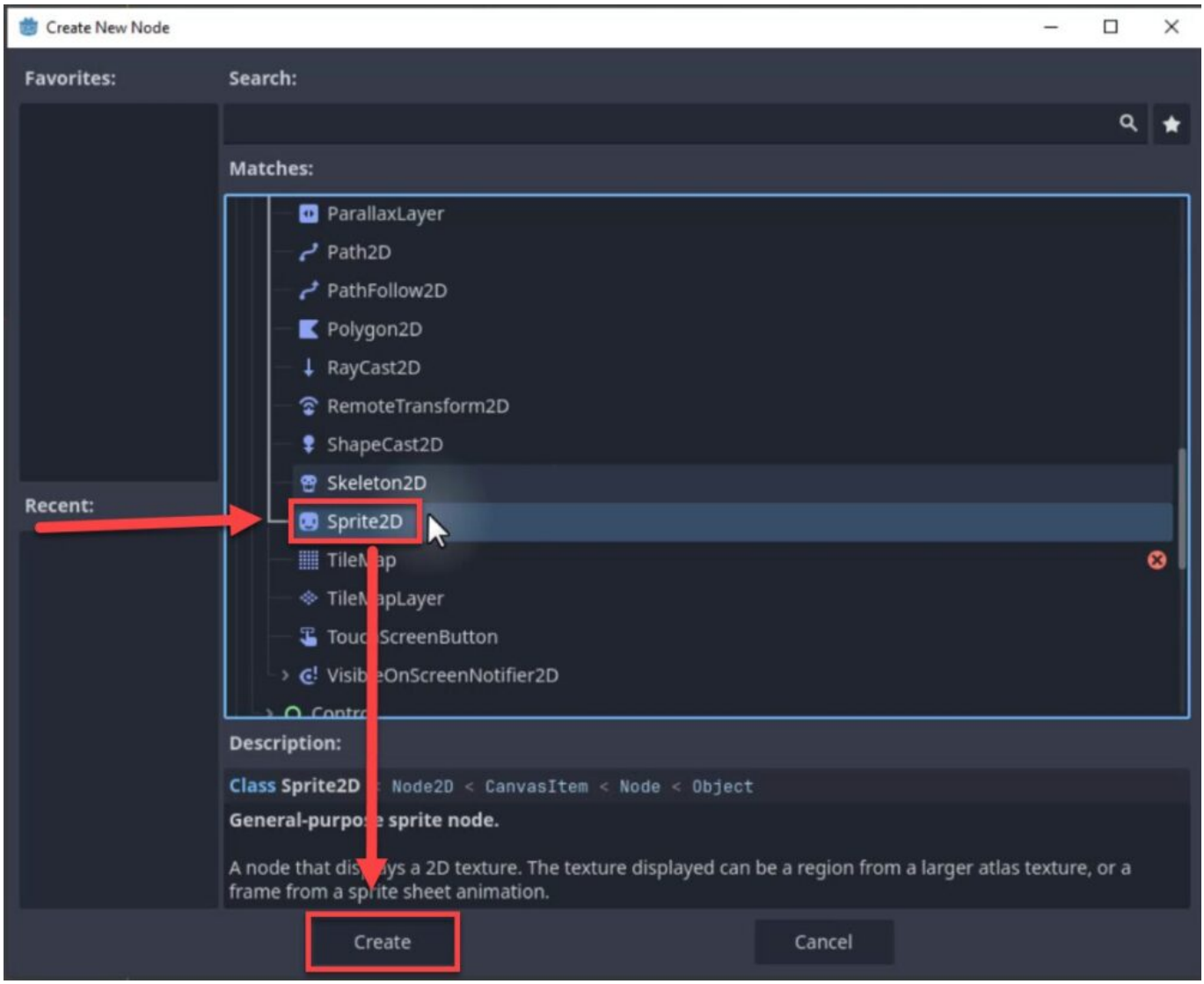


For now, you may find it helpful to browse through the available nodes. Nodes are color-coded:

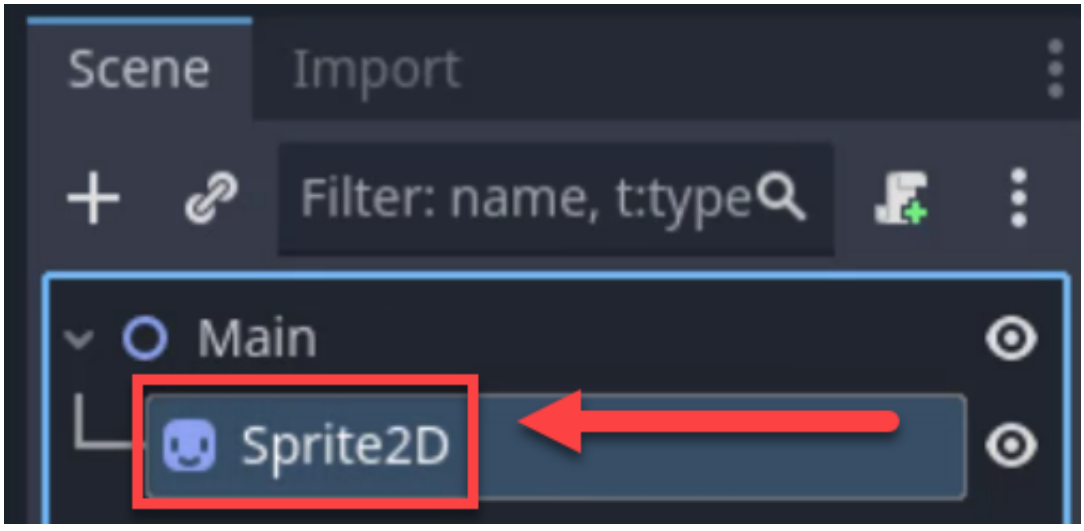
- **Blue**: 2D nodes
- **Red**: 3D nodes
- **Green**: UI elements
- **White or other colors**: General-purpose nodes



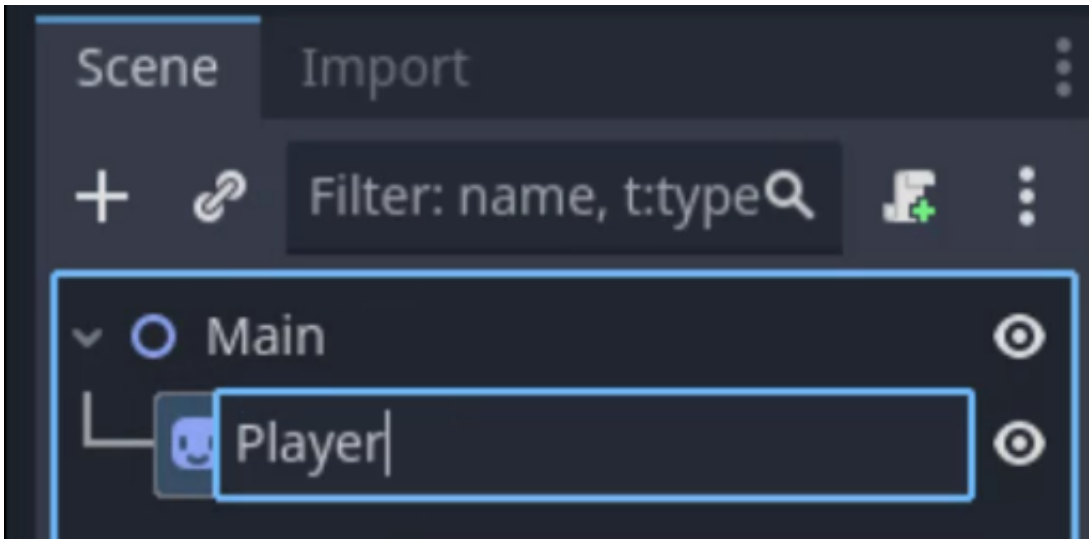
To create a node, select the desired node type. For example, to render a texture, select the **Sprite2D** node. Once you've picked your node, click 'Create' to add the new node to your scene.



The new node will appear indented under the root node, indicating that it is a child of the root node (we'll talk more about parenting later on in the course).

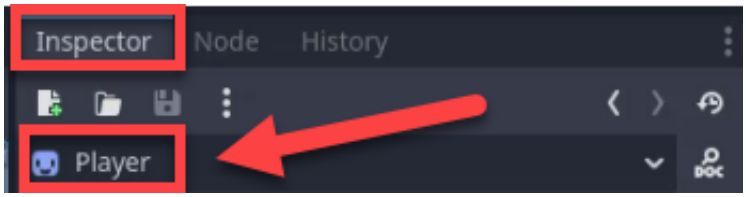


You can rename the node by double-clicking its name and entering a new one, such as 'Player'.

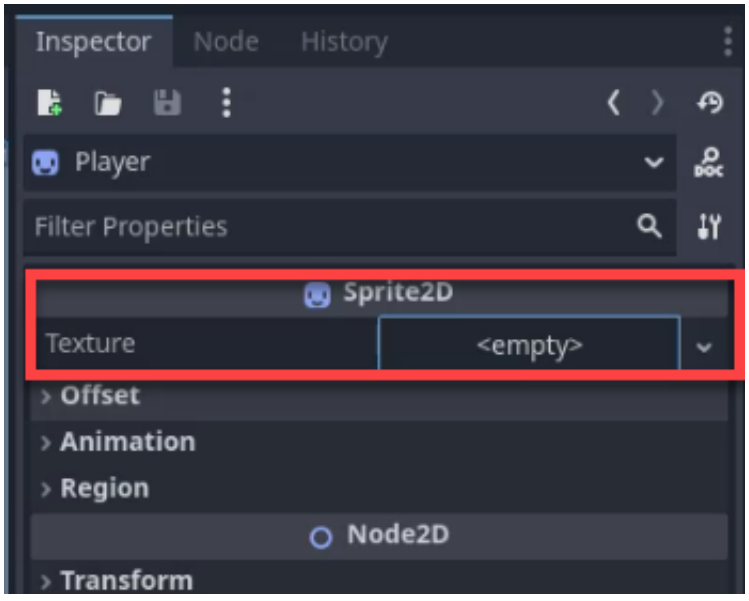


Assigning Properties to Nodes

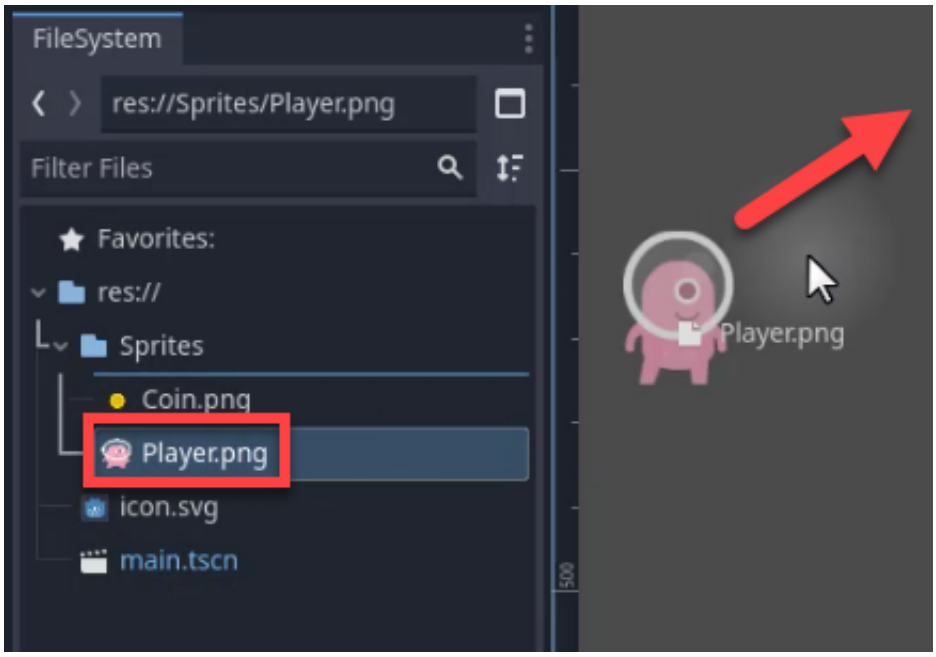
With a node selected, you can access and modify its properties in the inspector.

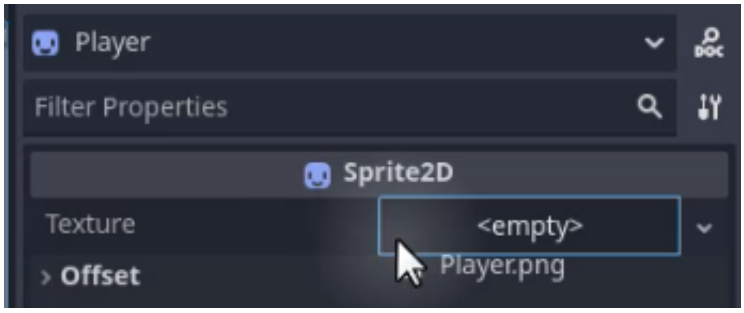


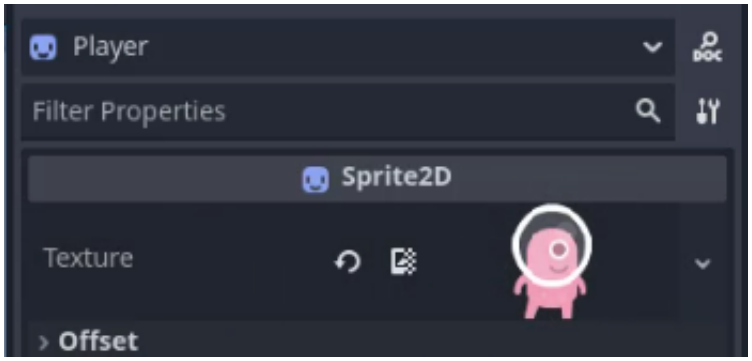
For a Sprite2D node, the main property is the texture.



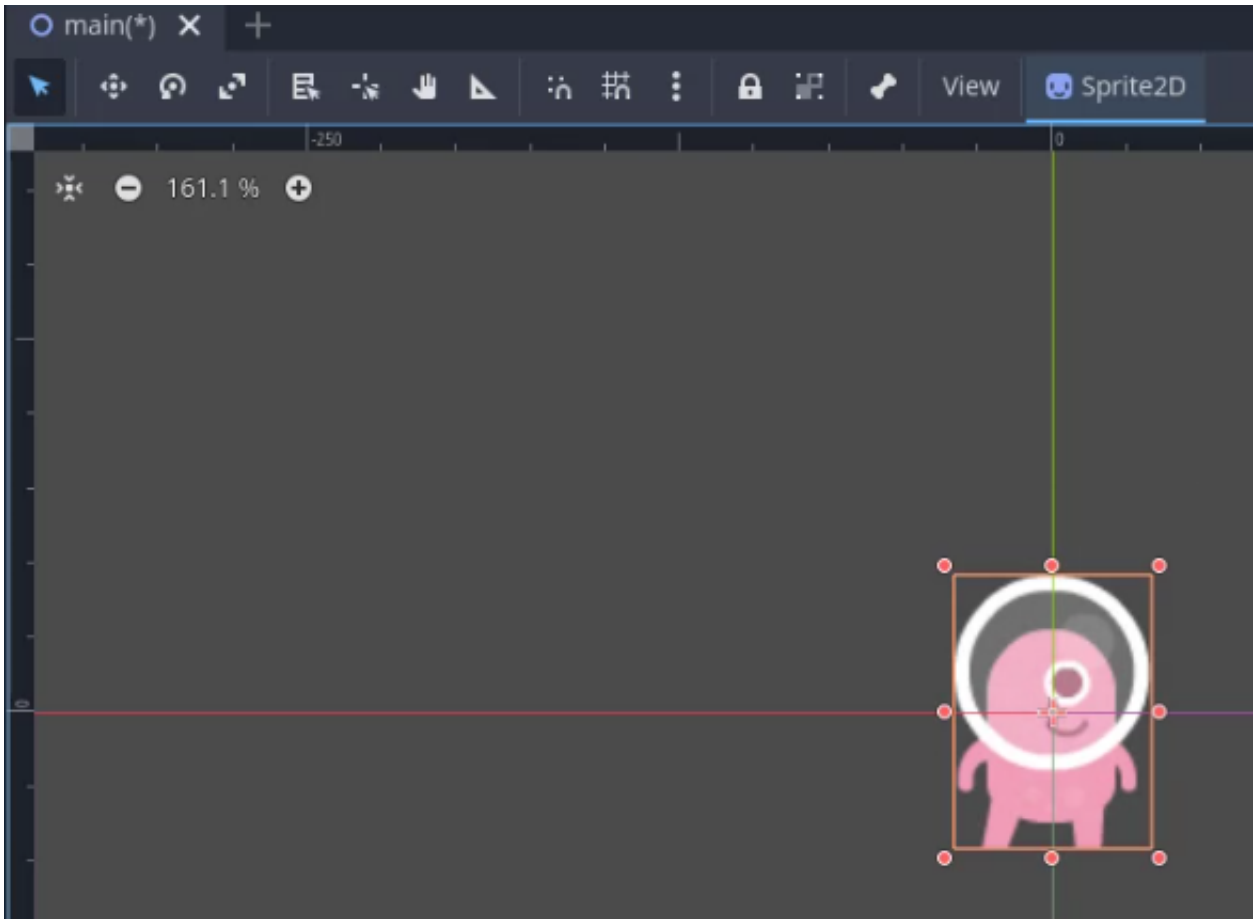
To assign a texture, drag and drop the desired image file (e.g., player.png) into the texture field.







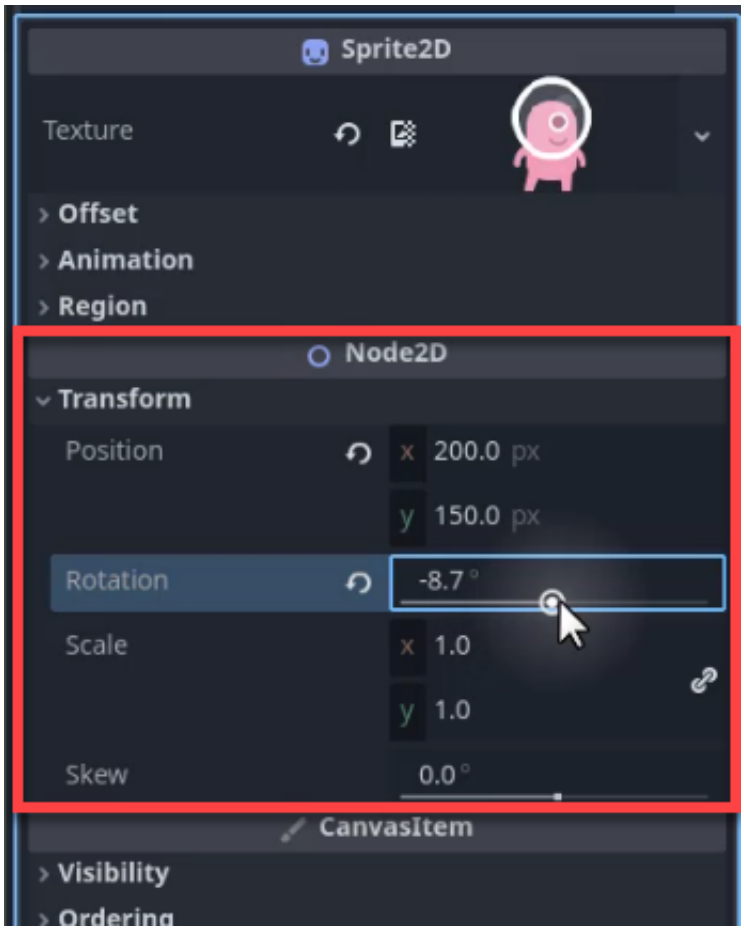
The texture will now be displayed in the scene view. You can zoom in, pan around, and focus on the node by selecting it and pressing the ‘F’ key.

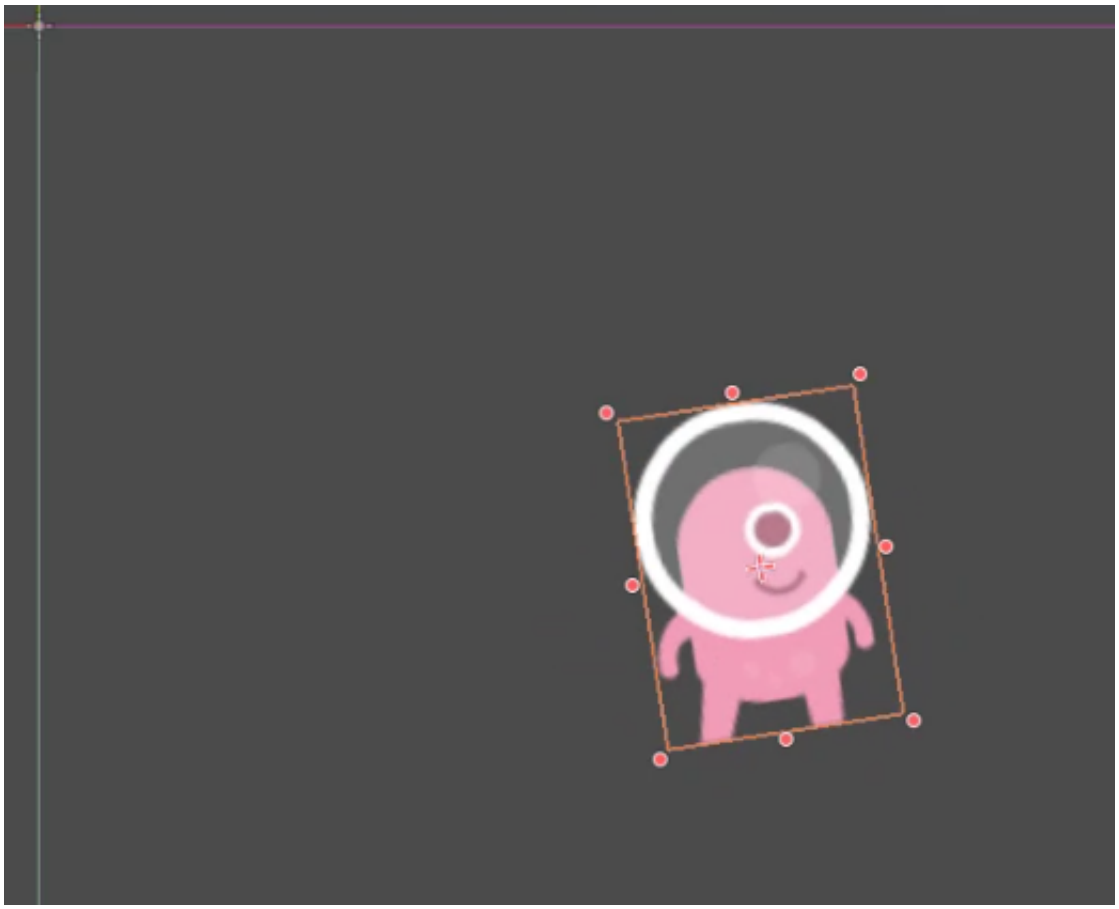


Manipulating Node Properties

To move, rotate, or scale a node, use the transform properties in the inspector:

- **Position**: Adjust the X (horizontal) and Y (vertical) coordinates to move the node. Note that the coordinates are in pixels.
- **Rotation**: Click and drag the rotation bar to change the node's orientation. In this case, the numbers are listed in degrees.
- **Scale**: Adjust the scale property to change the node's size. The scale acts is in a times scale where a scale of 2 would make the object twice as big and a scale of 0.5 would make it 50% of its original size.





Conclusion

Congratulations! You have now learned the basics of nodes and scenes in the Godot engine. In the next lesson, we will delve deeper into working with scenes and exploring more advanced features. Stay tuned and happy game development!

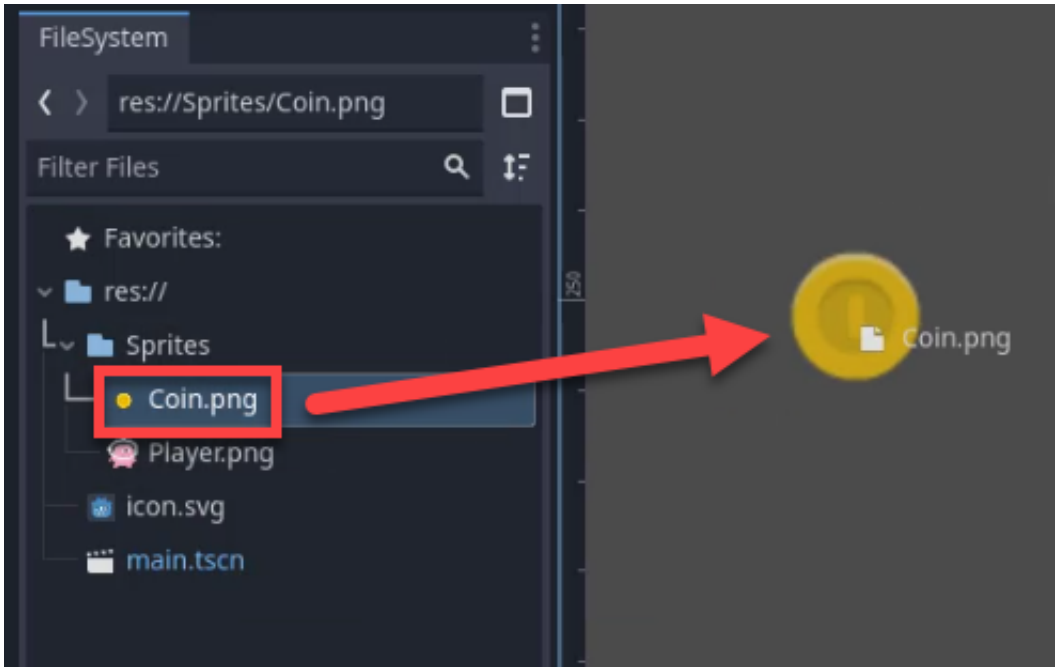
In this lesson, we will explore how to efficiently manage and manipulate multiple similar objects within your game project. By the end of this tutorial, you will understand the power of scenes and how they can streamline your game development process.

Introduction to Scenes in Godot

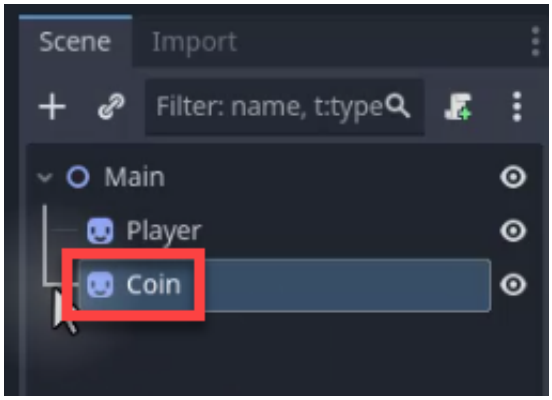
Scenes in Godot are collections of nodes that can be saved and instanced multiple times. This means that any changes made to the scene will be applied to all instances of that scene, making it a powerful tool for managing multiple similar objects.

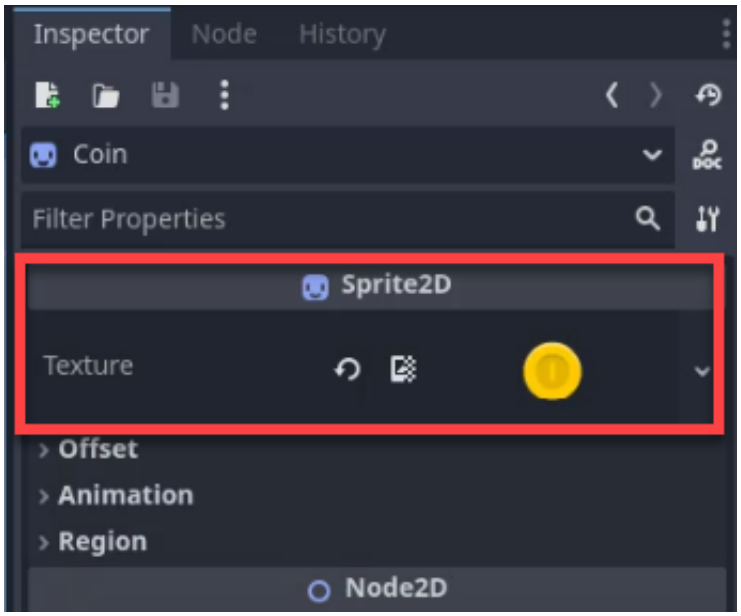
Creating a Coin Scene

Let’s start by creating a simple coin scene. Drag your coin image from the file system into your scene.

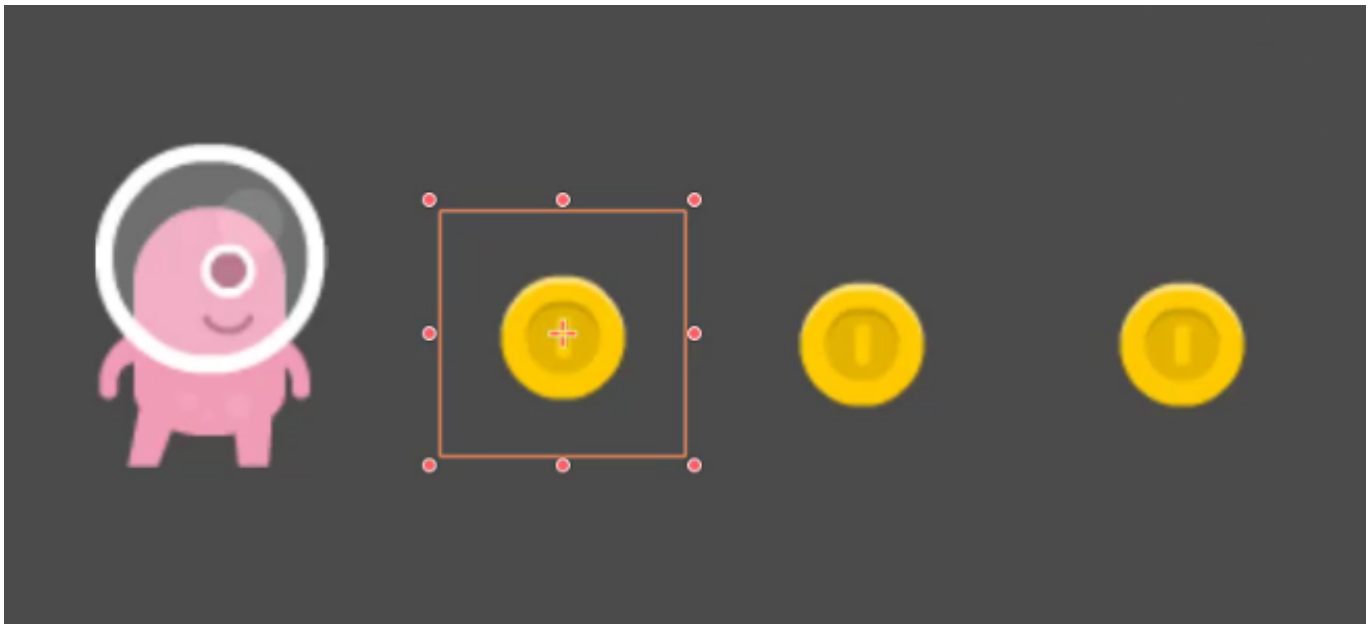


This will create a new Sprite2D node with the texture assigned.



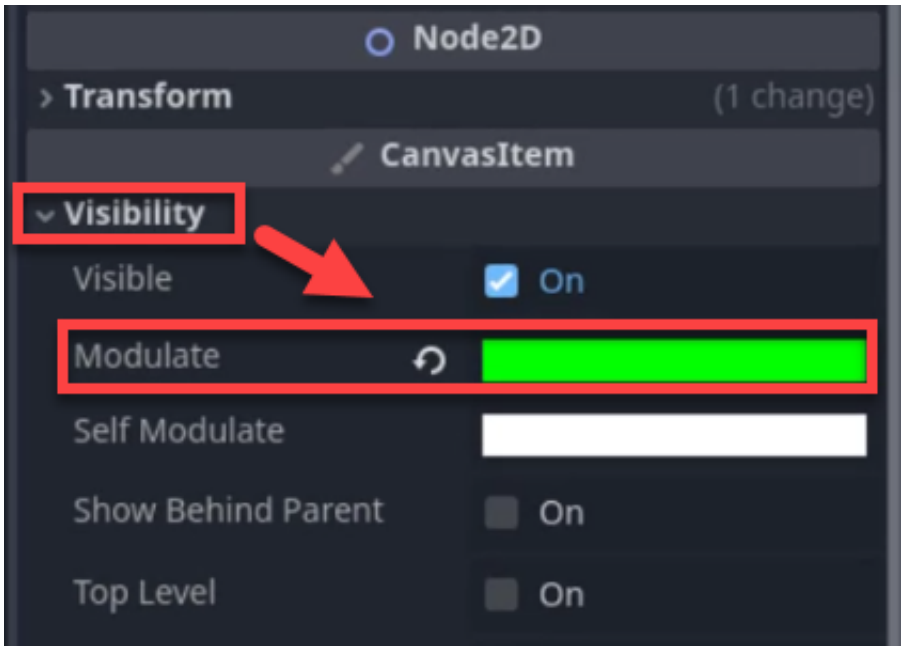


Repeat the process to create multiple coin nodes in your scene.



Modifying Individual Nodes

If you want to change the properties of individual nodes, such as the color of the coins, you can do so through the Inspector. Select a coin node, go to the Visibility section, and use the Modulate property to change the color of the coin.

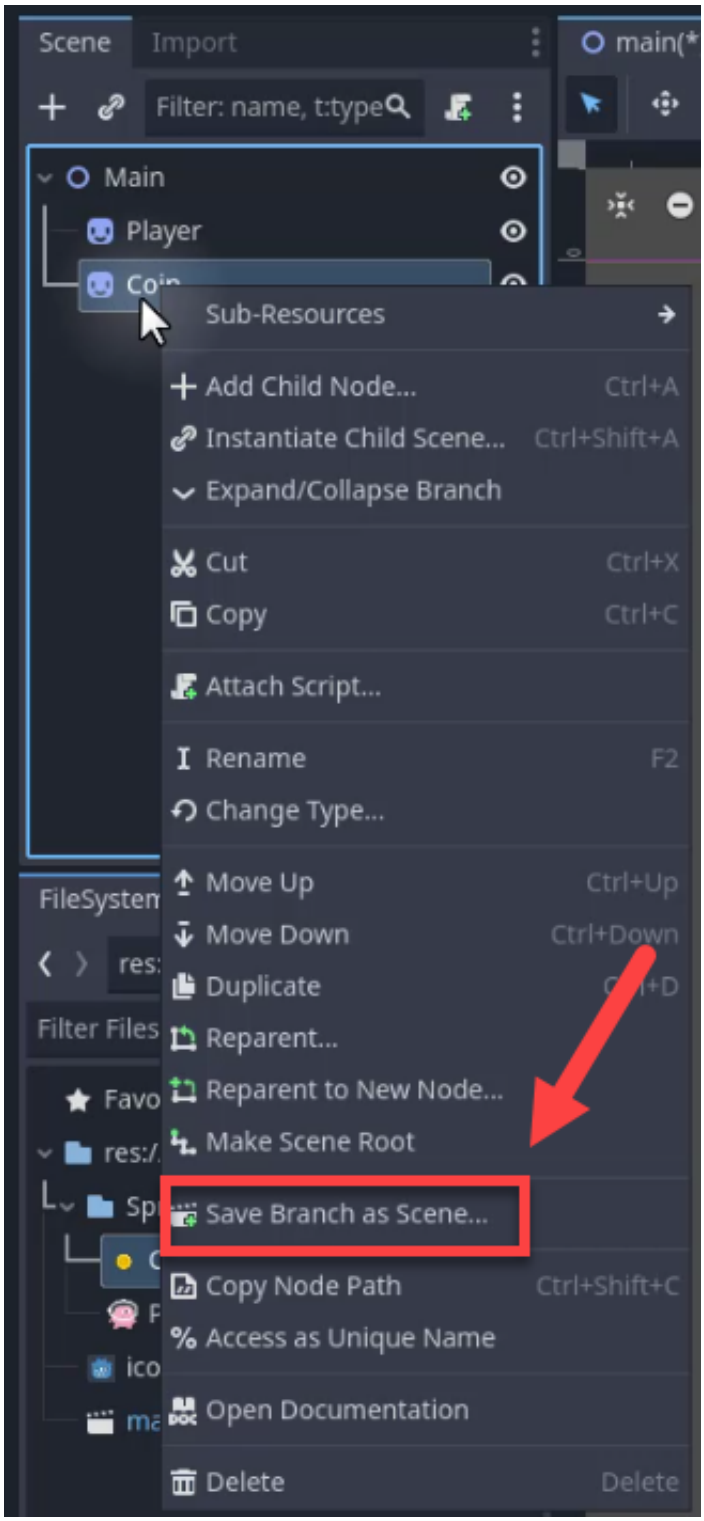


However, modifying each node individually can be time-consuming, especially if you have many nodes.

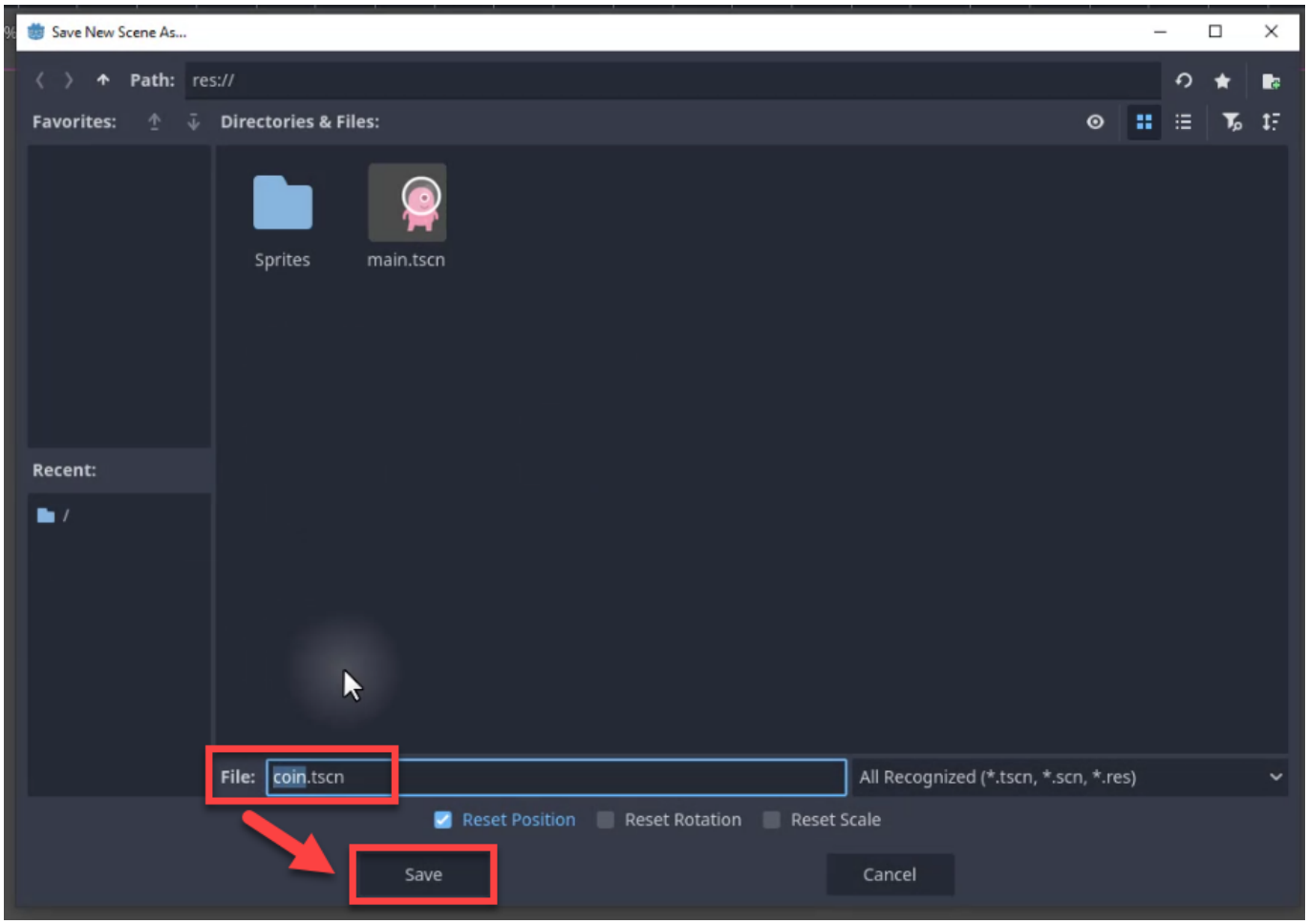
This is where scenes come in handy.

Saving a Node as a Scene

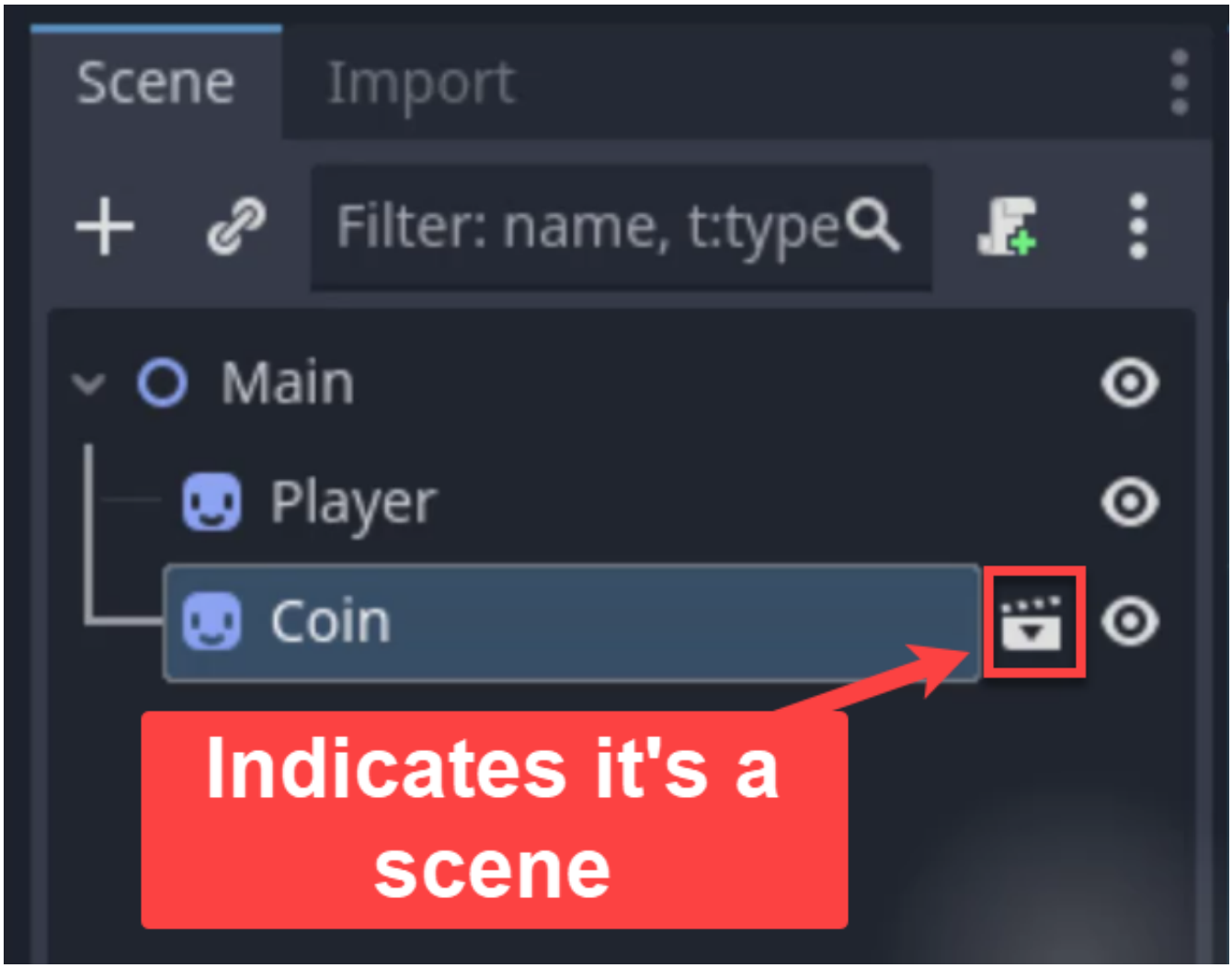
To save a node as a scene, first right-click on the node you want to save as a scene. In the menu that popups, select Save Branch as Scene.



Give your scene a name, such as coin.tscn, and click Save.

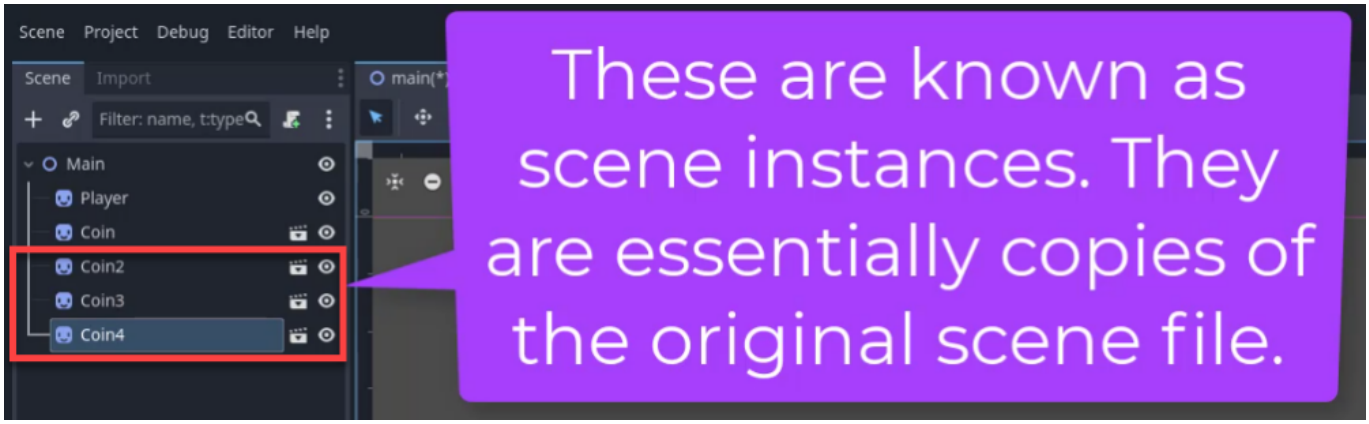


Now, the node is saved as a scene, and any changes made to this scene will be applied to all instances of it.

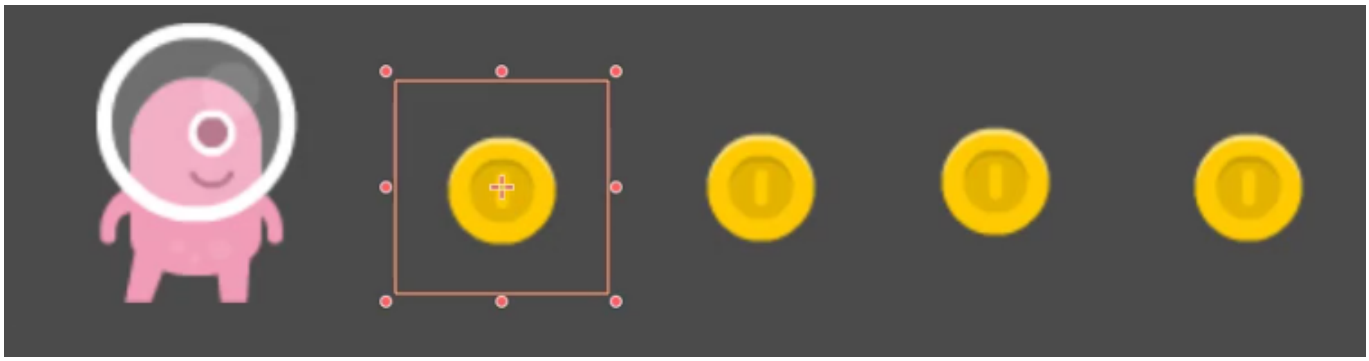


Duplicating and Managing Scene Instances

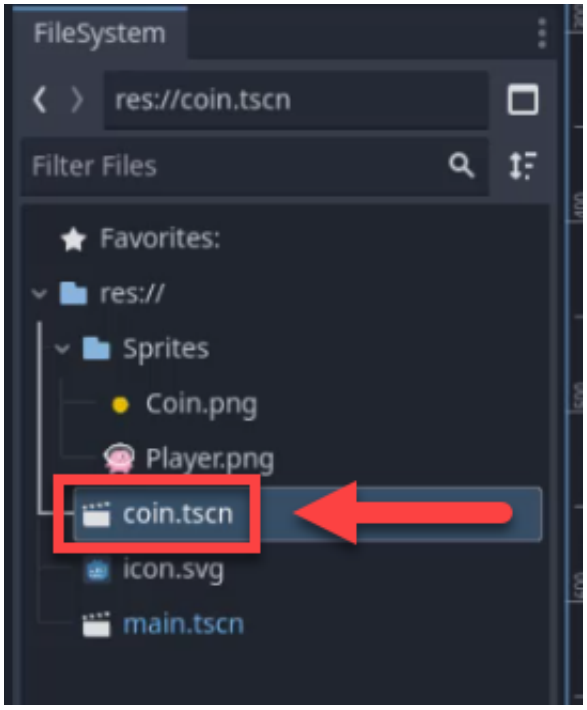
To create multiple instances of your coin scene, select the coin node in your scene and press Ctrl+D to duplicate the node.



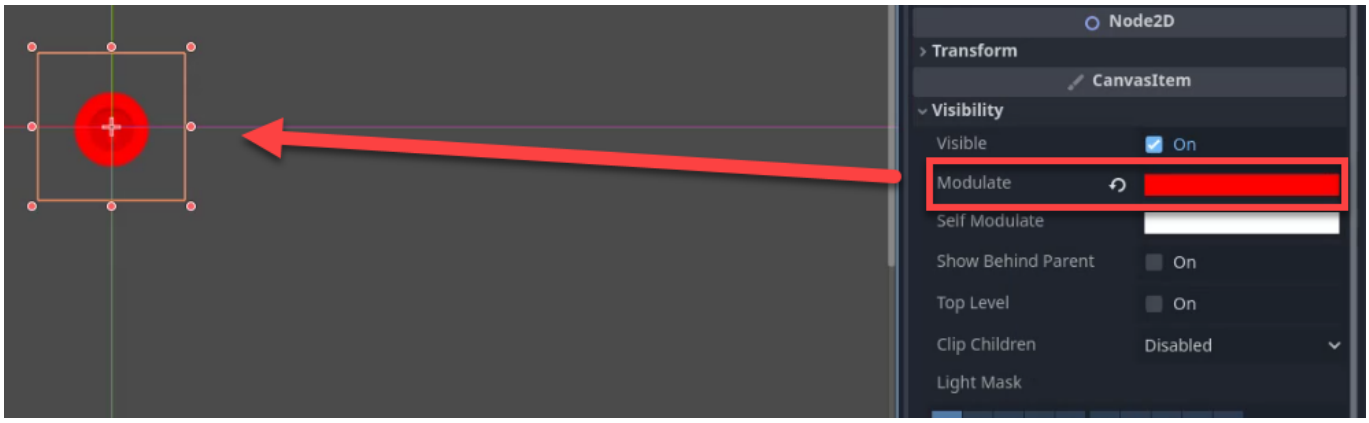
Move the duplicated nodes to their desired positions.



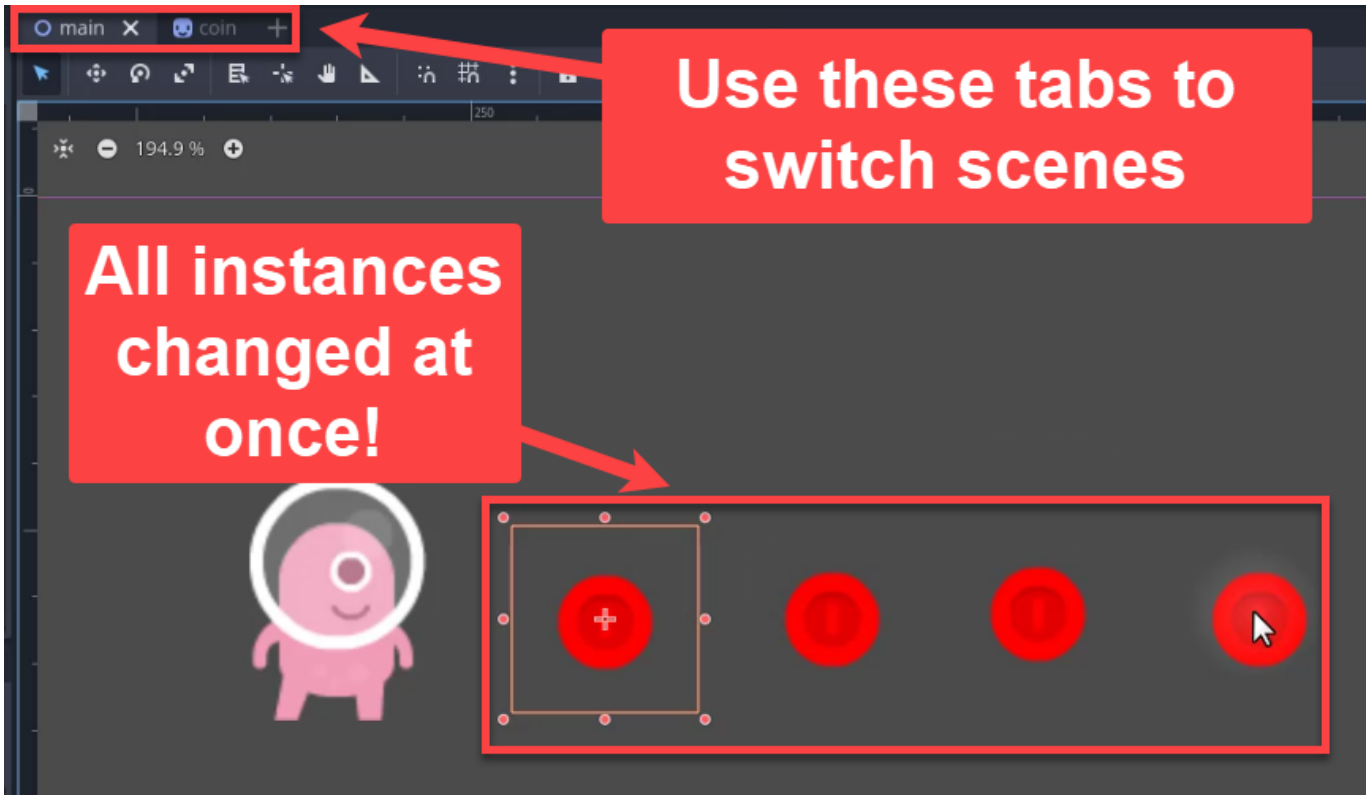
If you want to change the properties of all coin instances, we now simply modify the original scene. Double-click on the coin scene in the file system to open it.



Make the desired changes, such as changing the color or scale. Once done, save the scene with Ctrl+S.

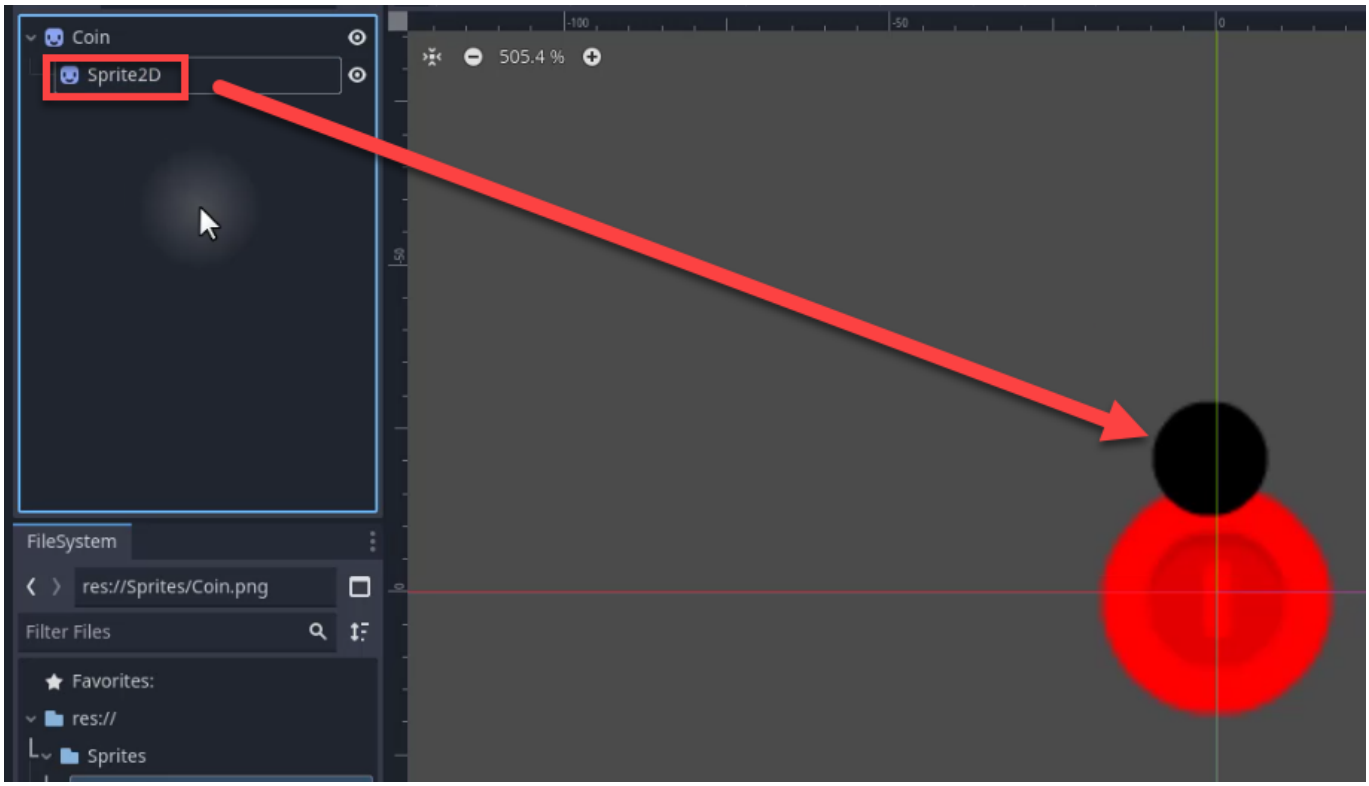


Return to your main scene to see the changes applied to all instances.

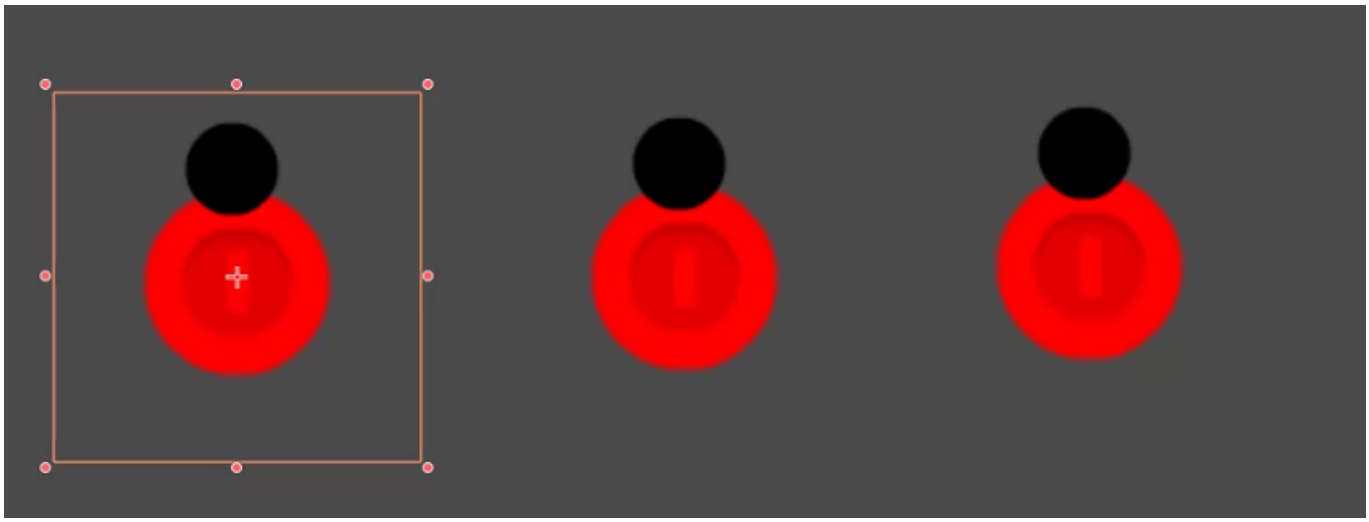


Adding Nodes to a Scene

You can also add new nodes to your scene to enhance its functionality. Open your coin scene and add a new Sprite2D node (or any other node you desire). Customize the new node's properties as needed.



Save the scene and return to your main scene to see the changes applied to all instances again!



Experimenting with Scenes

Feel free to experiment with setting up nodes and scenes, saving a scene, and duplicating it multiple times to see how it works and to get familiar with the process. This will help you understand the power of utilizing scenes and nodes in Godot.

Conclusion

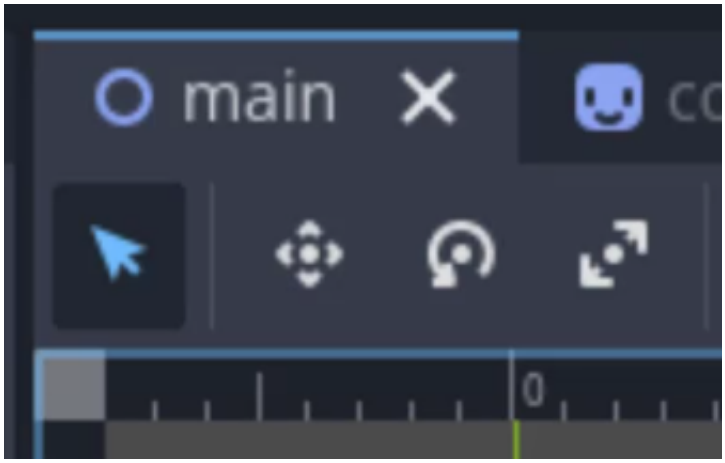
In the next lesson, we will be going over node tools, which will allow us to rotate, move, and scale our nodes. Stay tuned for more insights into game development with Godot. Thank you for watching, and see you all then!

In this lesson, we will explore various tools that allow us to modify the position, rotation, and scale of nodes within our game. These tools are essential for designing and arranging elements in your game world.

Introduction to Node Tools

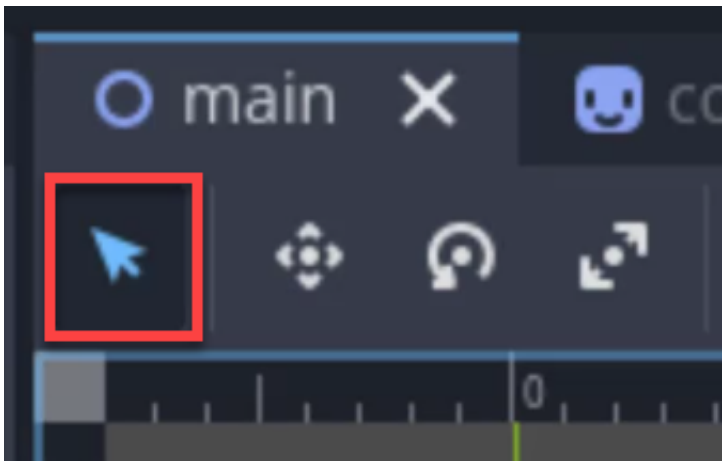
Node tools in Godot enable us to manipulate the properties of nodes directly in the viewport. These tools include:

- Select Mode
- Move Mode
- Rotate Mode
- Scale Mode

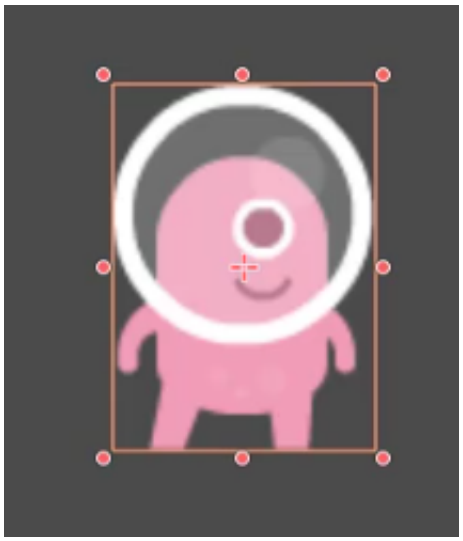


Select Mode

The Select Mode is the default tool that allows you to select nodes in the viewport. When a node is selected, it is highlighted, and its properties are displayed in the inspector.



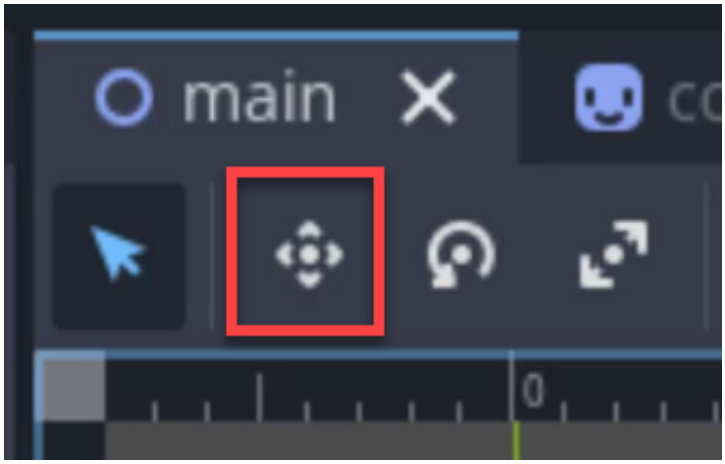
To select a node, simply click on it in the viewport. The selected node will be highlighted with an orange box.

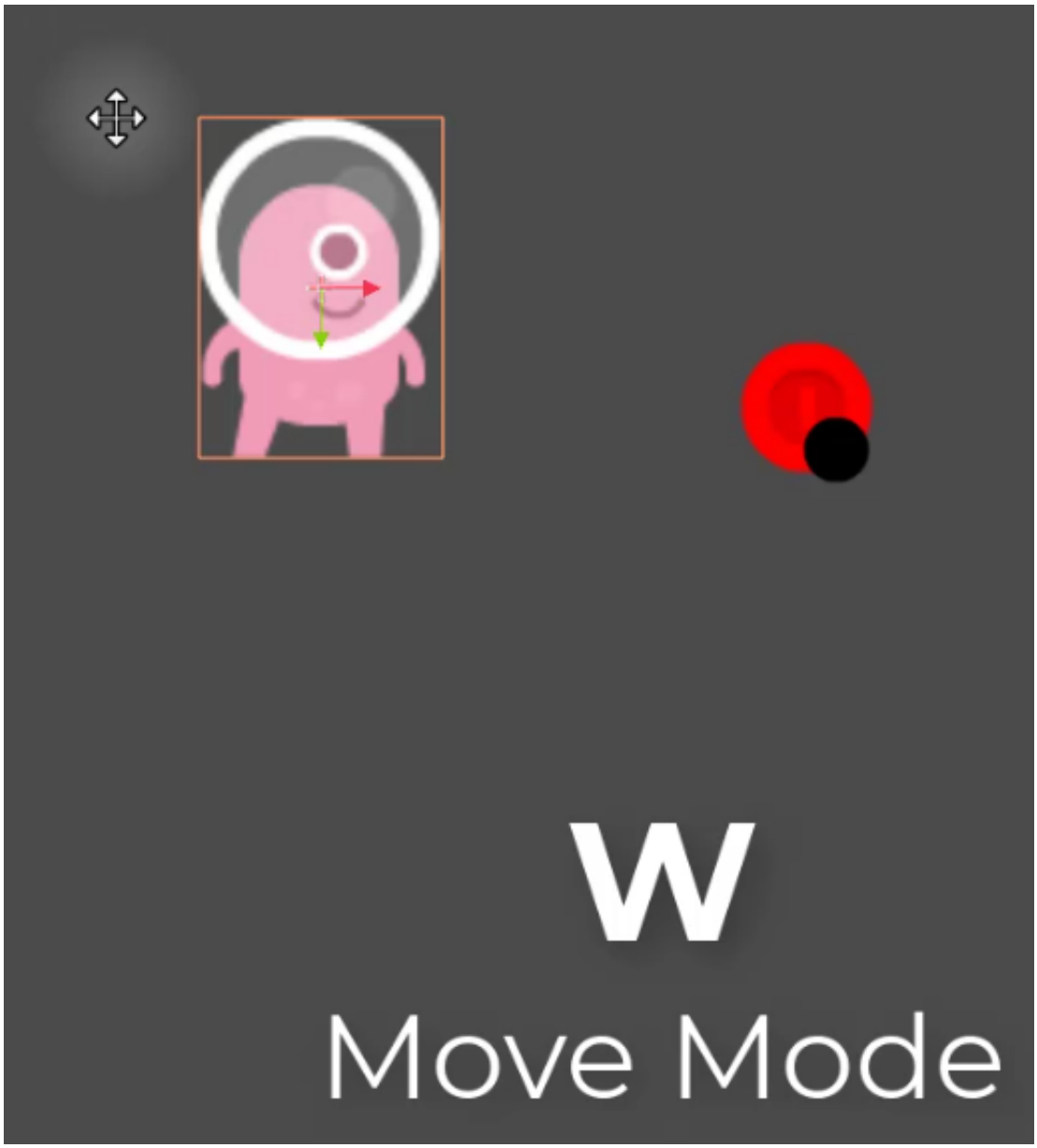


You can deselect a node by clicking elsewhere in the viewport.

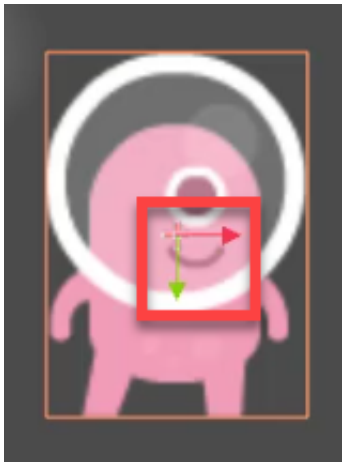
Move Mode

The Move Mode allows you to reposition nodes in the game world. You can access this mode by clicking the move tool icon or using the keyboard shortcut W.



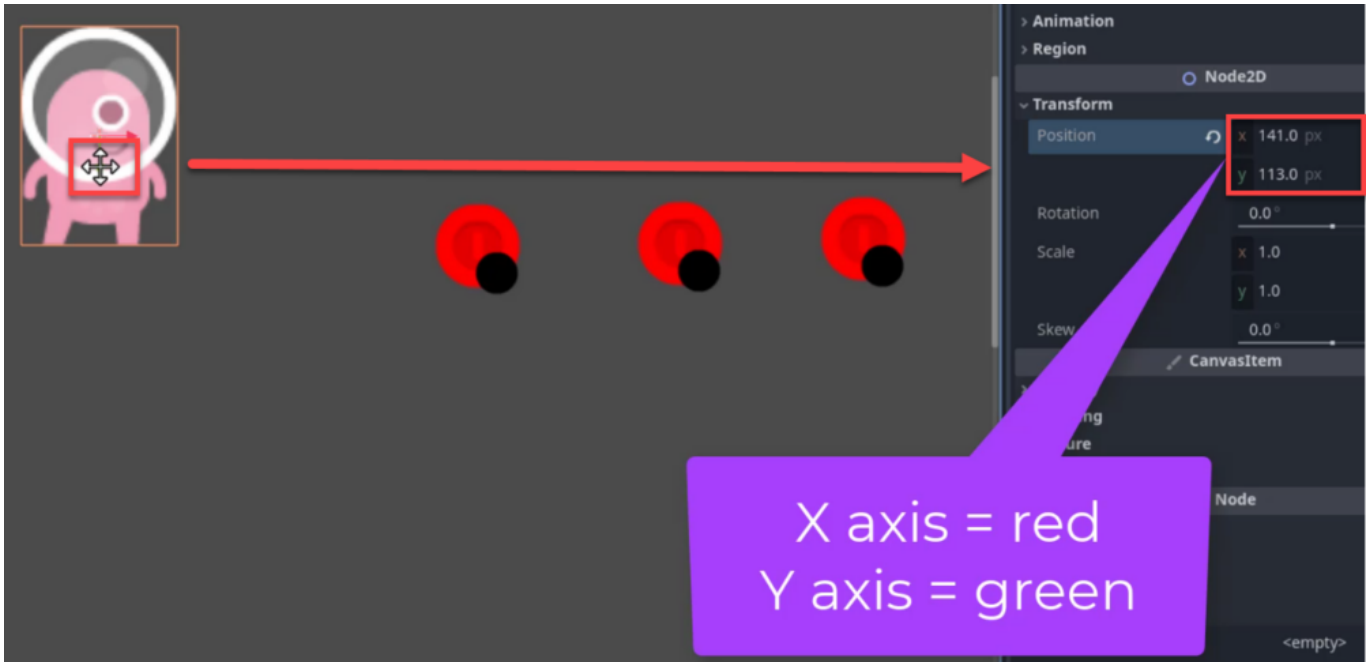


In Move Mode, you will see red and green arrows around the selected node.



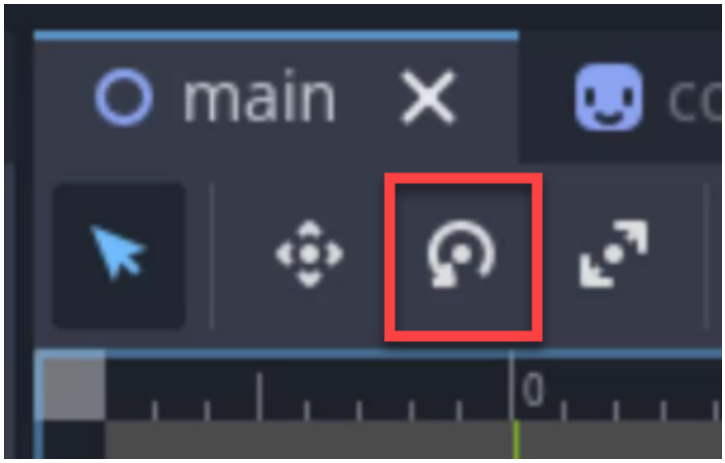
You can click and drag the red arrow to move the node along the X-axis (left and right) or the green

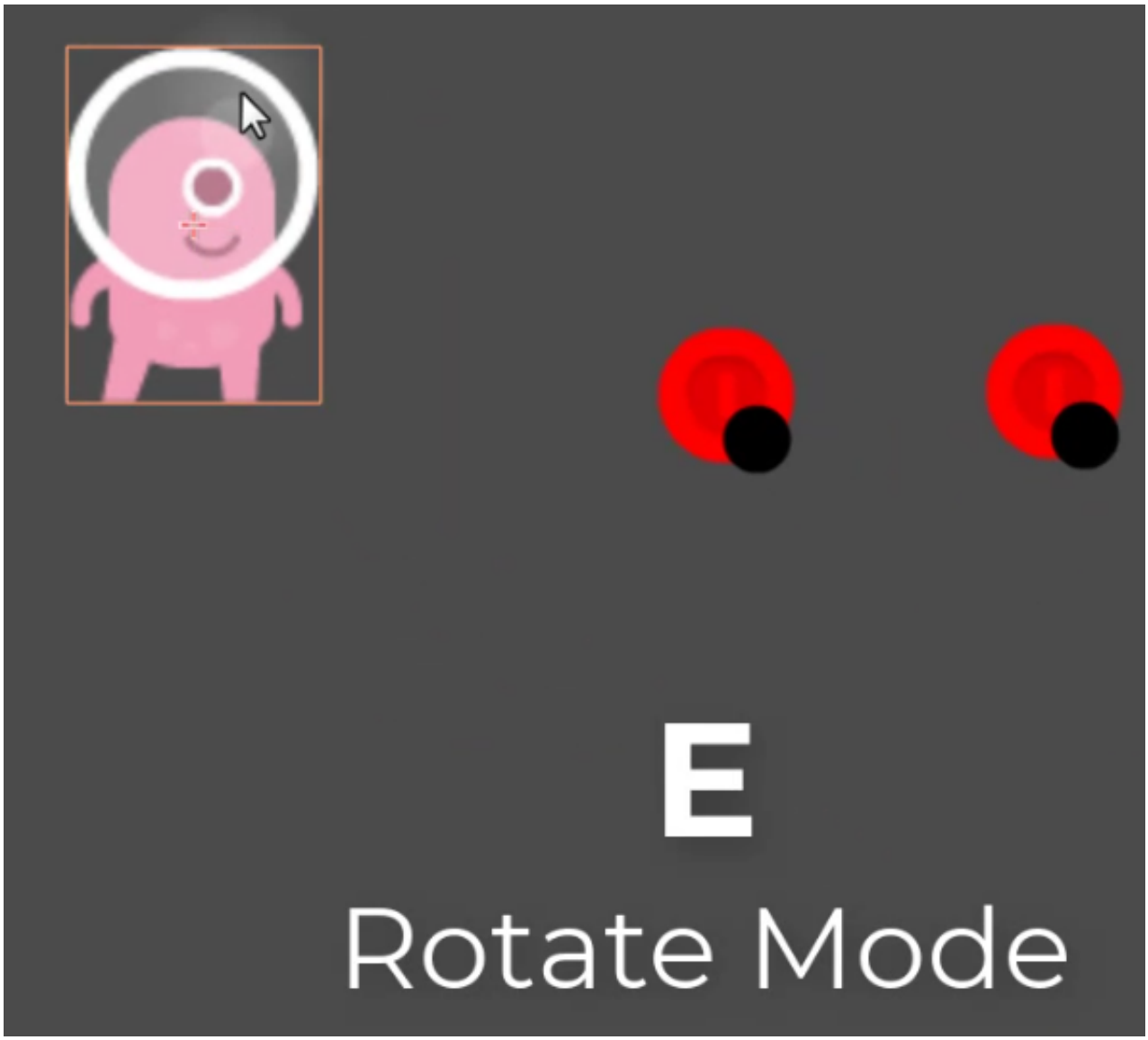
arrow to move the node along the Y-axis (up and down). You can also click and drag anywhere on the node to move it freely in any direction.



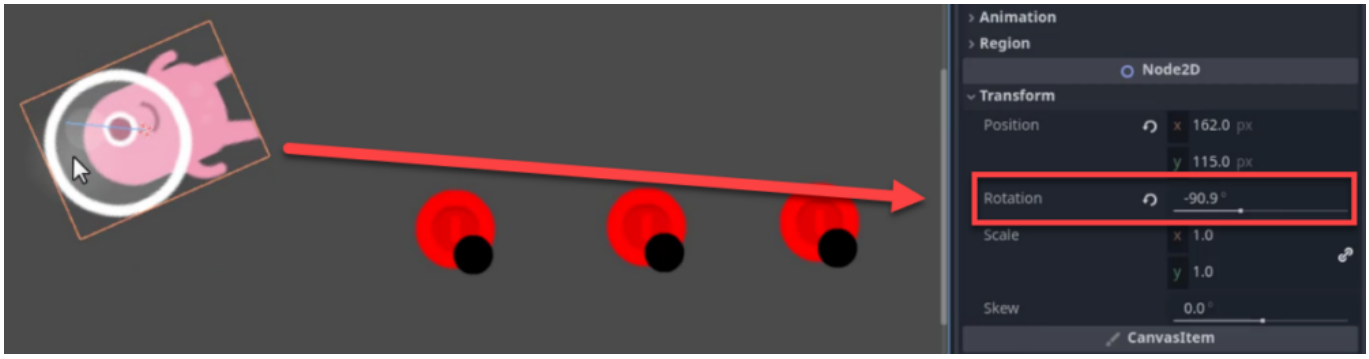
Rotate Mode

The Rotate Mode enables you to change the orientation of nodes. You can access this mode by clicking the rotate tool icon or using the keyboard shortcut E.

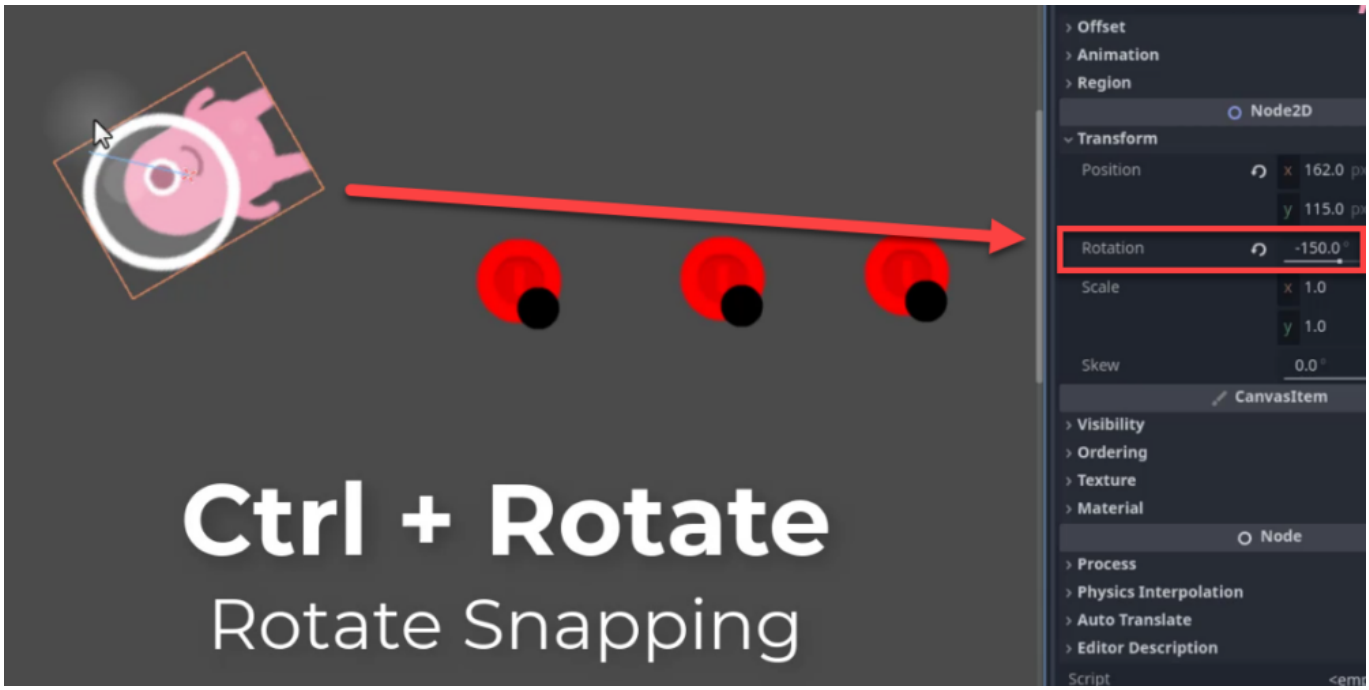




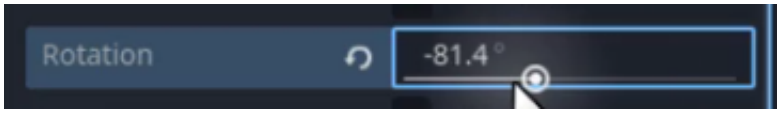
In Rotate Mode, click and drag around the node to rotate it. The rotation is measured in degrees, and you can see the rotation property changing in the Inspector.



Hold down the Ctrl key while rotating to snap the rotation in increments of 15 degrees.

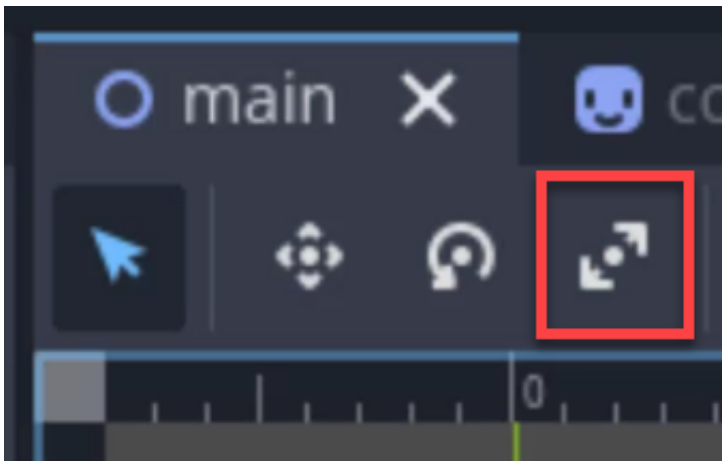


You can also directly input the desired rotation degree in the inspector or use the slider to adjust it.

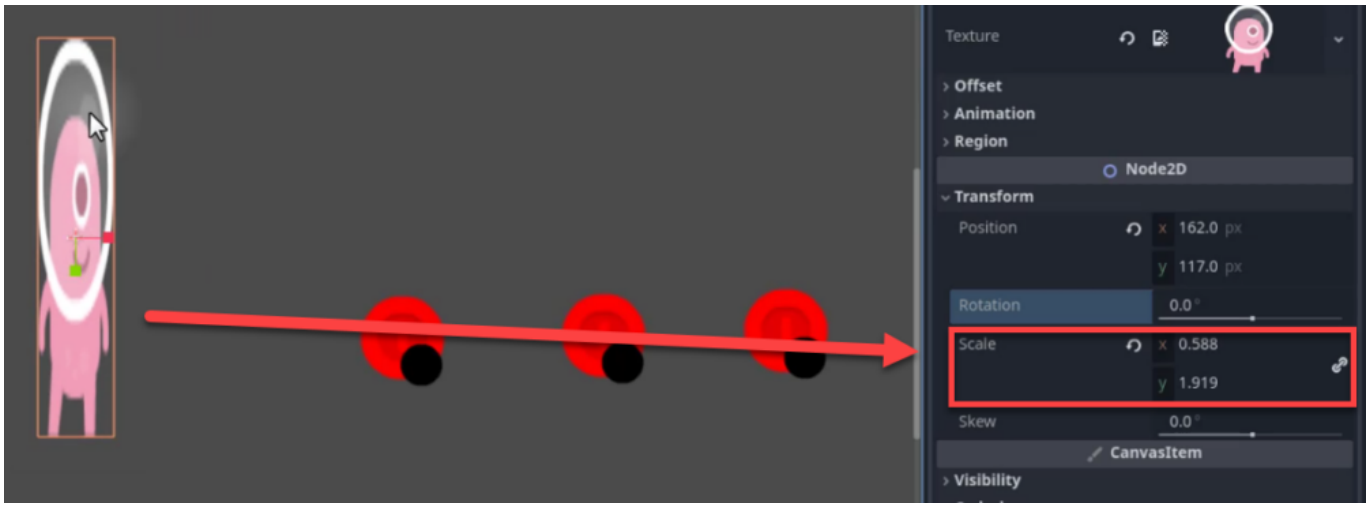


Scale Mode

The Scale Mode allows you to change the size of nodes. You can access this mode by clicking the scale tool icon or using the keyboard shortcut S.

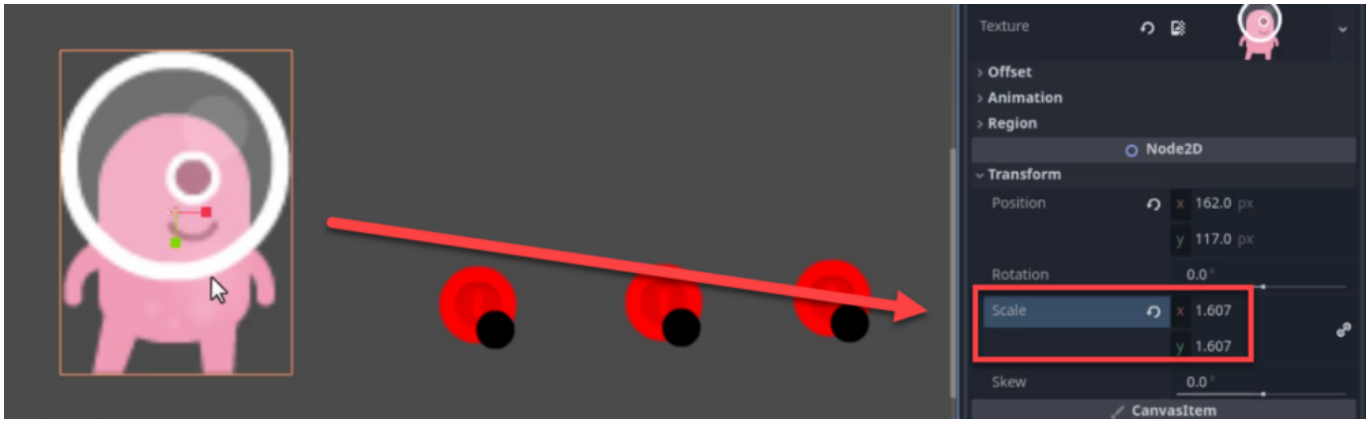


In Scale Mode, click and drag on the node to change its size.

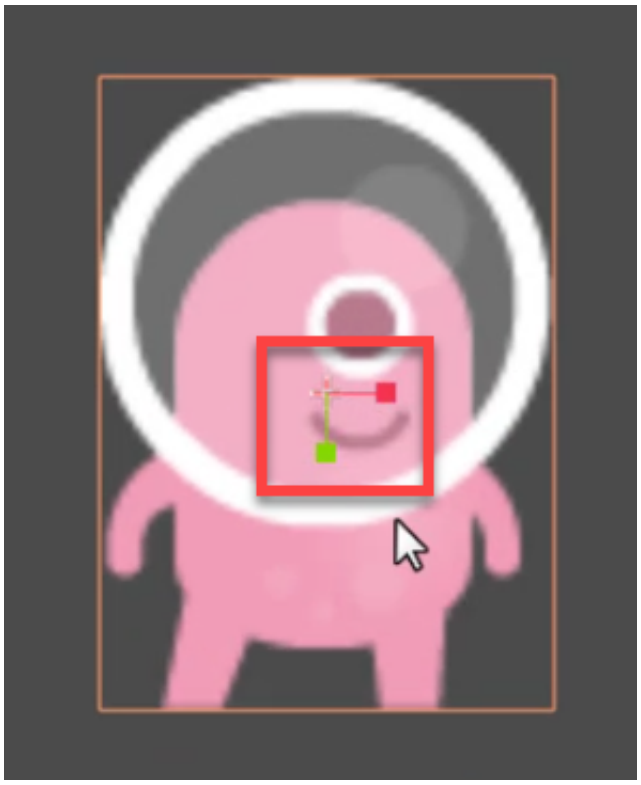


The scale is represented as a multiplier, where 1 is the original size. Setting the scale to 0 makes the node infinitely small.

Hold down the Shift key while scaling to scale the node uniformly along both axes.



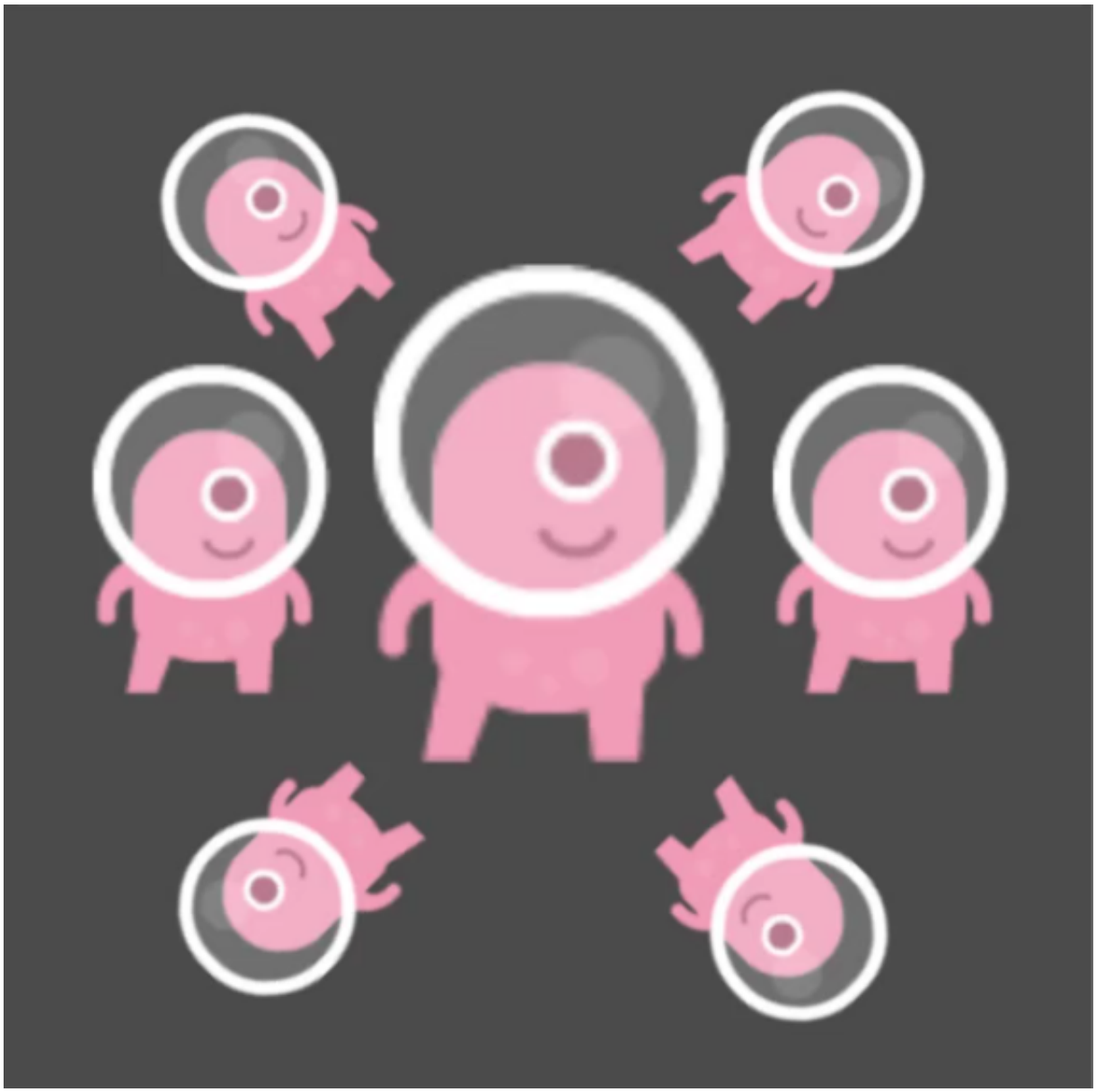
Like moving, you can click and drag along the red line to scale along the X-axis, or click and drag along the green line to scale along the Y-axis.



Challenge

To reinforce what you've learned, here's a challenge:

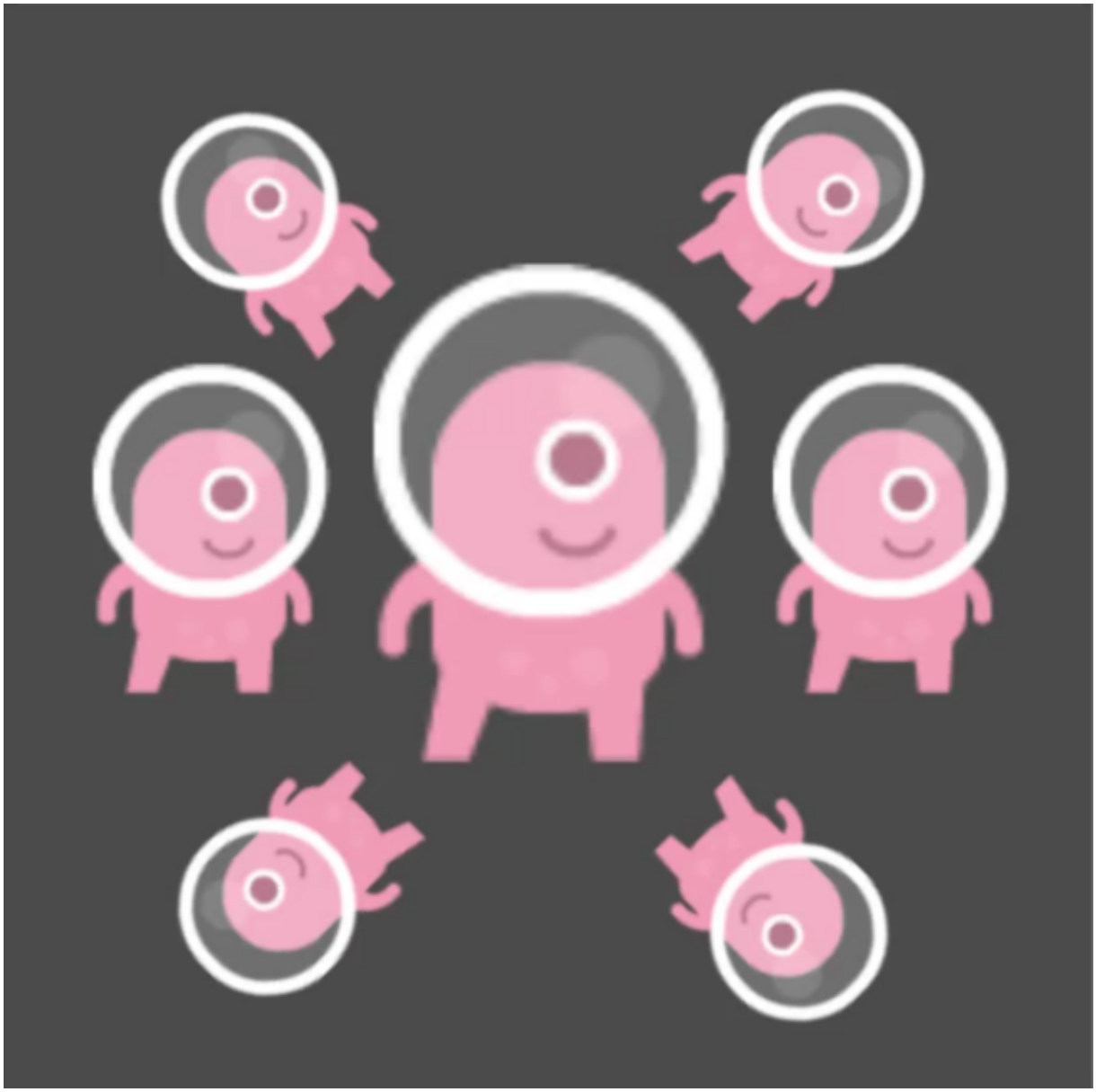
1. Create a new Sprite 2D node with the player texture.
2. Duplicate the node and use the move, rotate, and scale tools to position, rotate, and scale the nodes to match the given image.



This challenge will help you practice using the node tools before moving on to the next lesson, where we will review the solution.

Thank you for reading, and happy game development with Godot and Zenva!

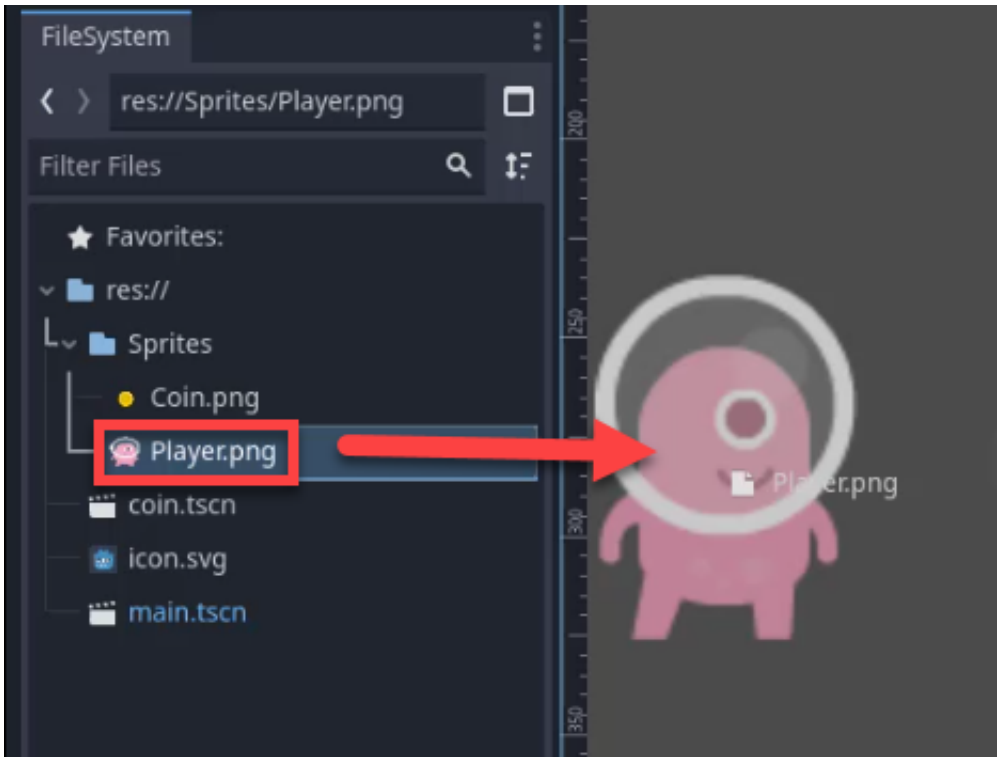
In this lesson, we'll solve the challenge provided earlier by creating and manipulating **Sprite2D** nodes to match a specific scene layout (seen below).



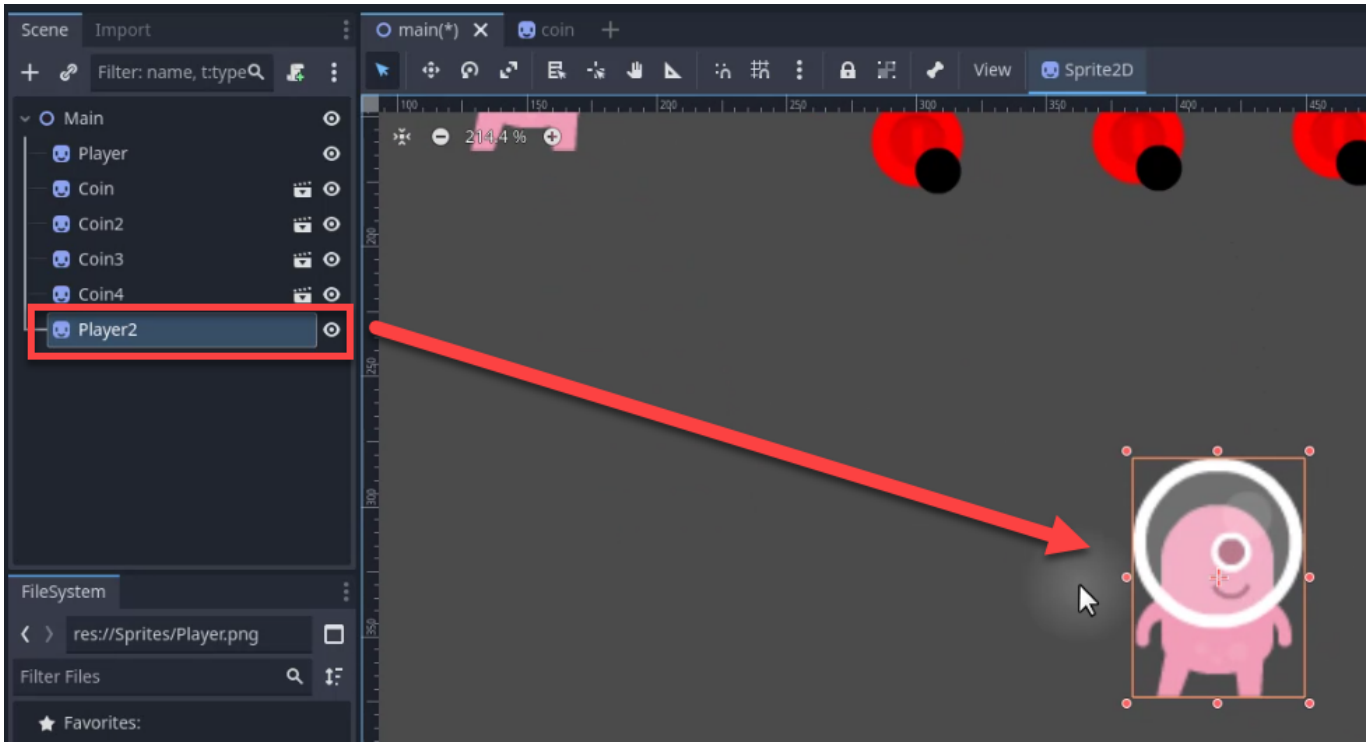
This is a great way to practice using the **Select**, **Move**, **Rotate**, and **Scale** tools in Godot. Let's begin!

Create the Initial Sprite2D Node

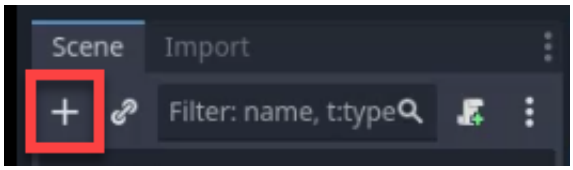
First, we need to create a **Sprite2D** node with the player texture. Drag your player texture directly into the viewport.

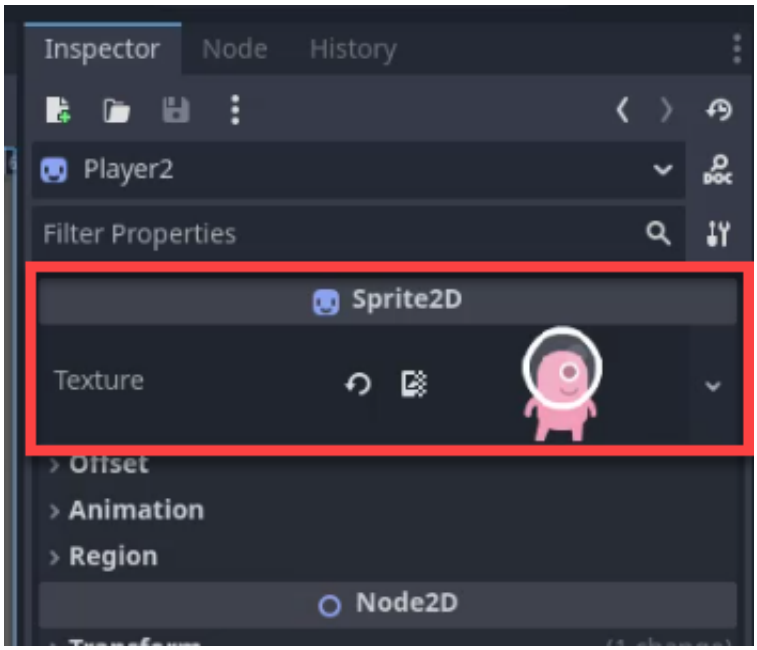


This automatically creates a Sprite2D node and assigns the texture.



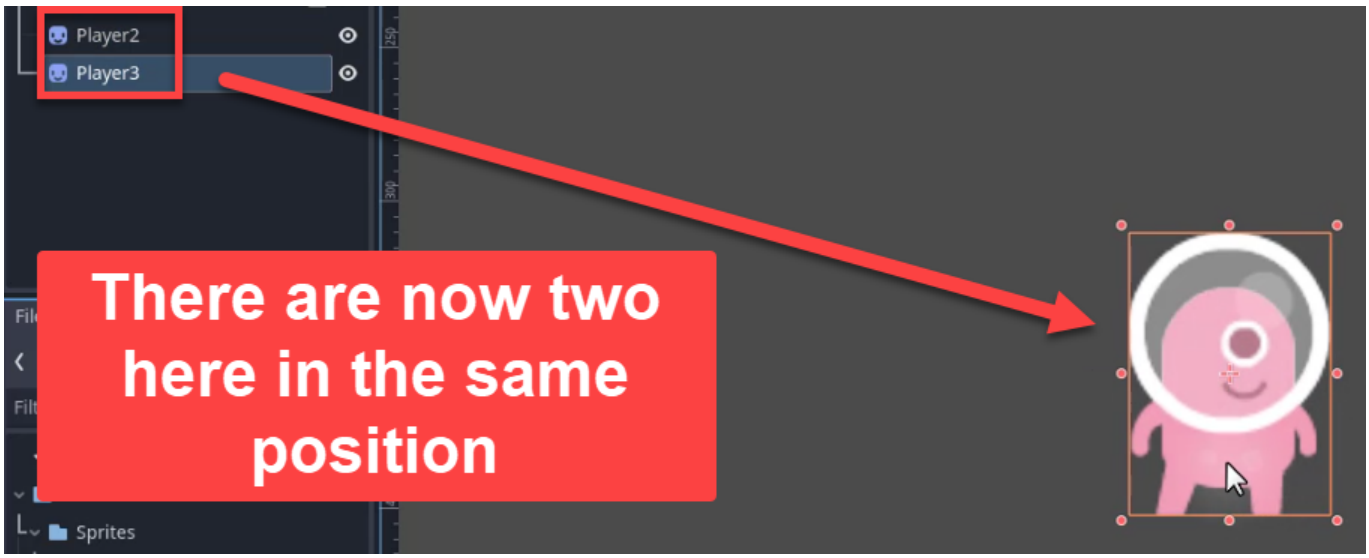
Alternatively, click the *Add Child Node* button, search for Sprite2D, then assign the texture through the *Inspector*.



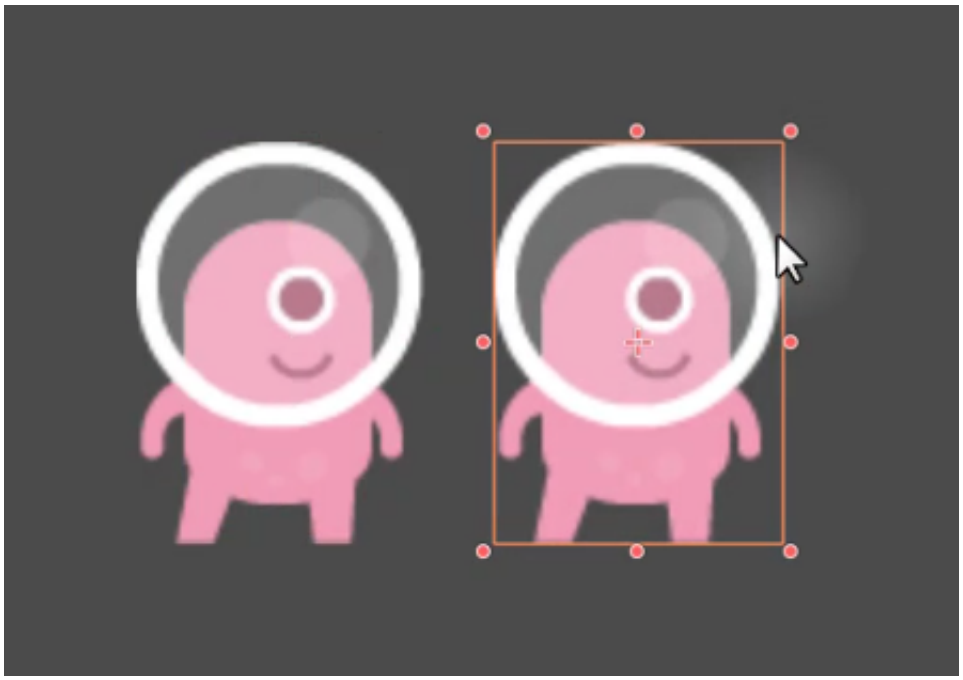


Duplicate and Move the Node

With the first sprite in place, let's create additional sprites and position them. Select the sprite and press Ctrl + D to duplicate it.

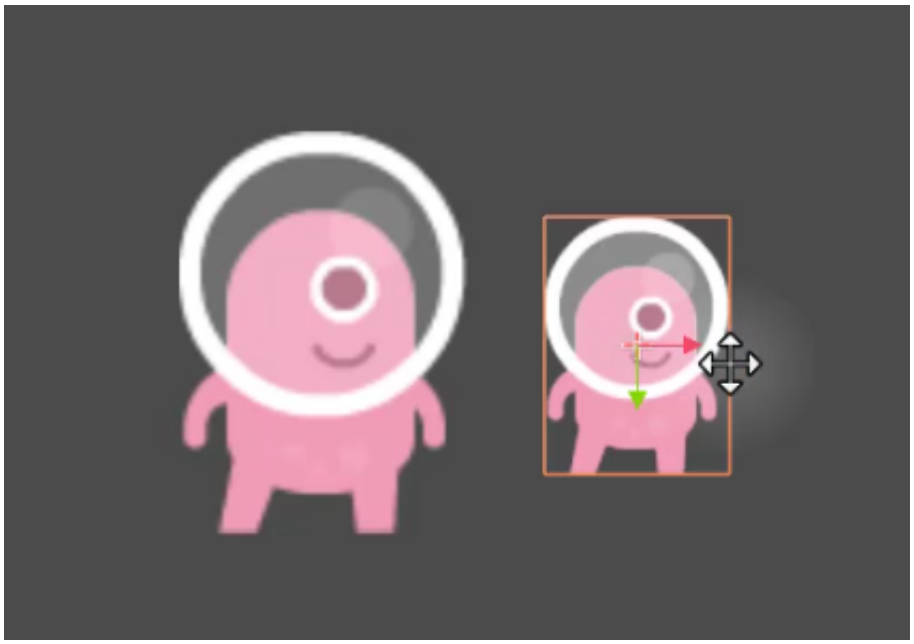


Press W to activate the **Move Tool**. Then, drag the duplicate to the right along the X-axis. Hold Shift while dragging to ensure it stays aligned to the axis.

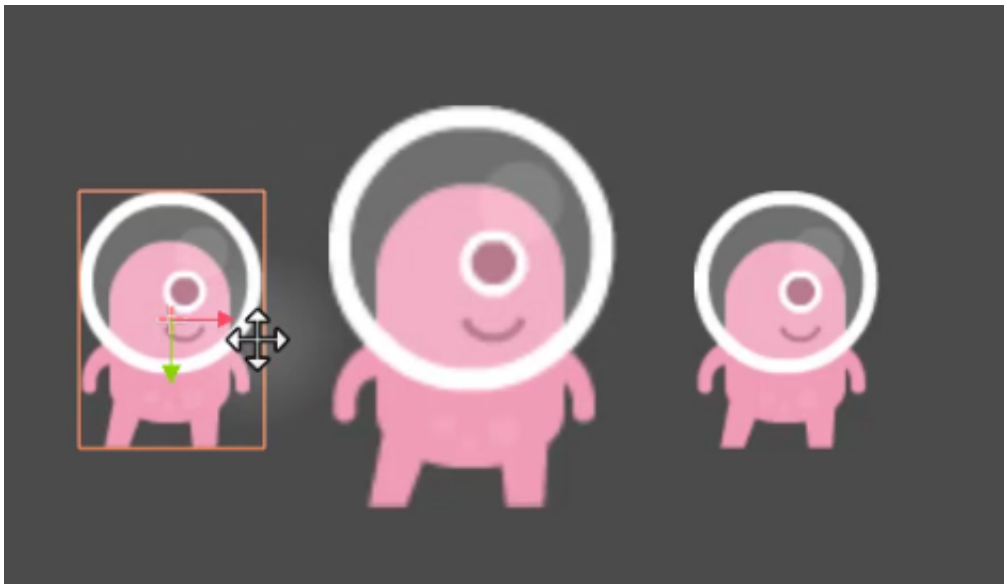


Scale the Node

To resize the new sprite, press S to activate the **Scale Tool**. Click and drag while holding Shift to scale the sprite uniformly along both axes. Adjust the size as needed to match our target image, and feel free to move the image again if necessary.

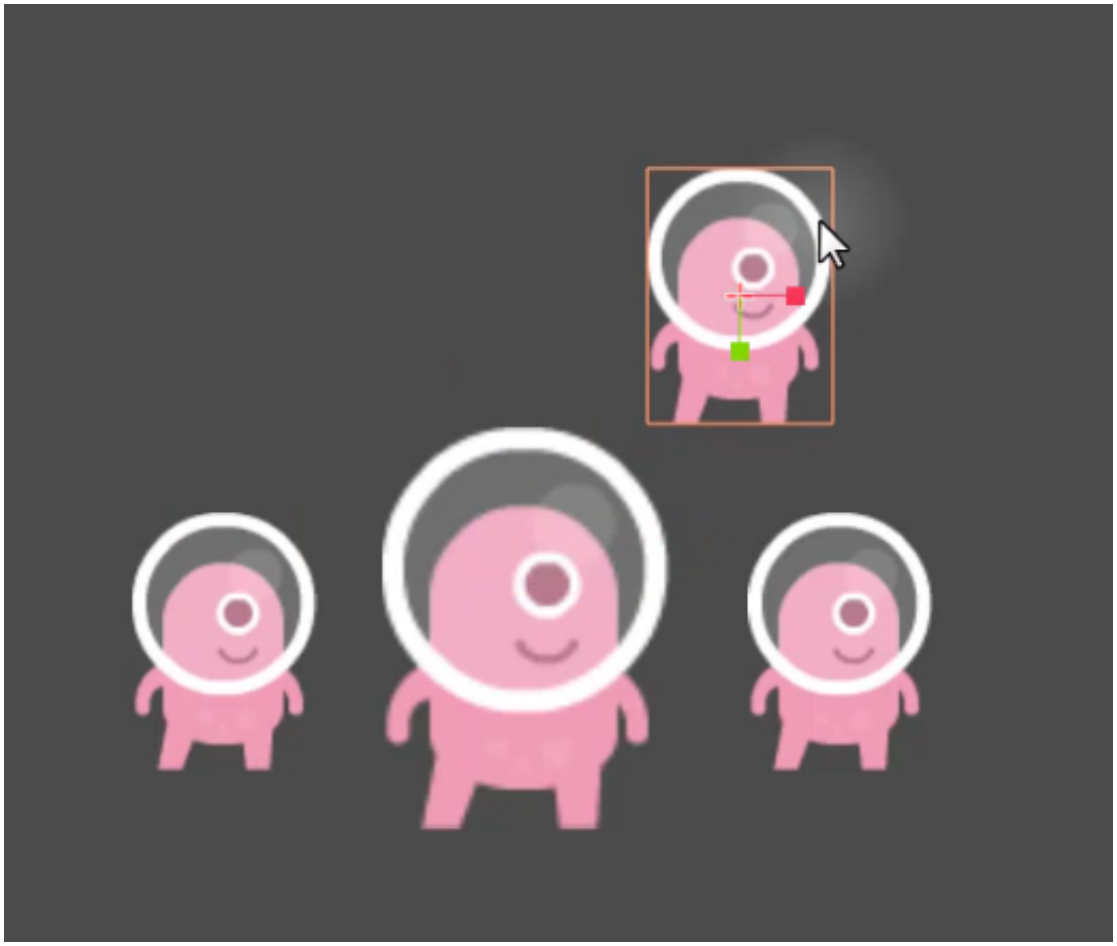


After this, duplicate the scaled sprite and drag it over to the other side of the larger sprite on the X-axis.

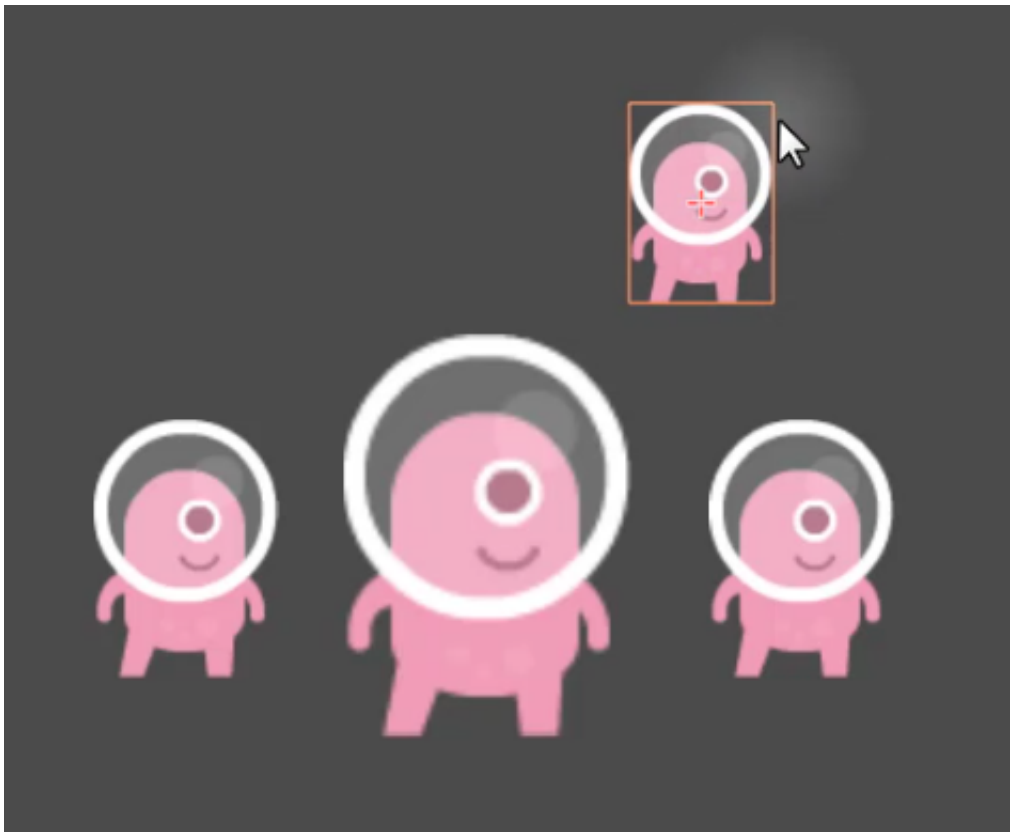


Add Rotated Sprites

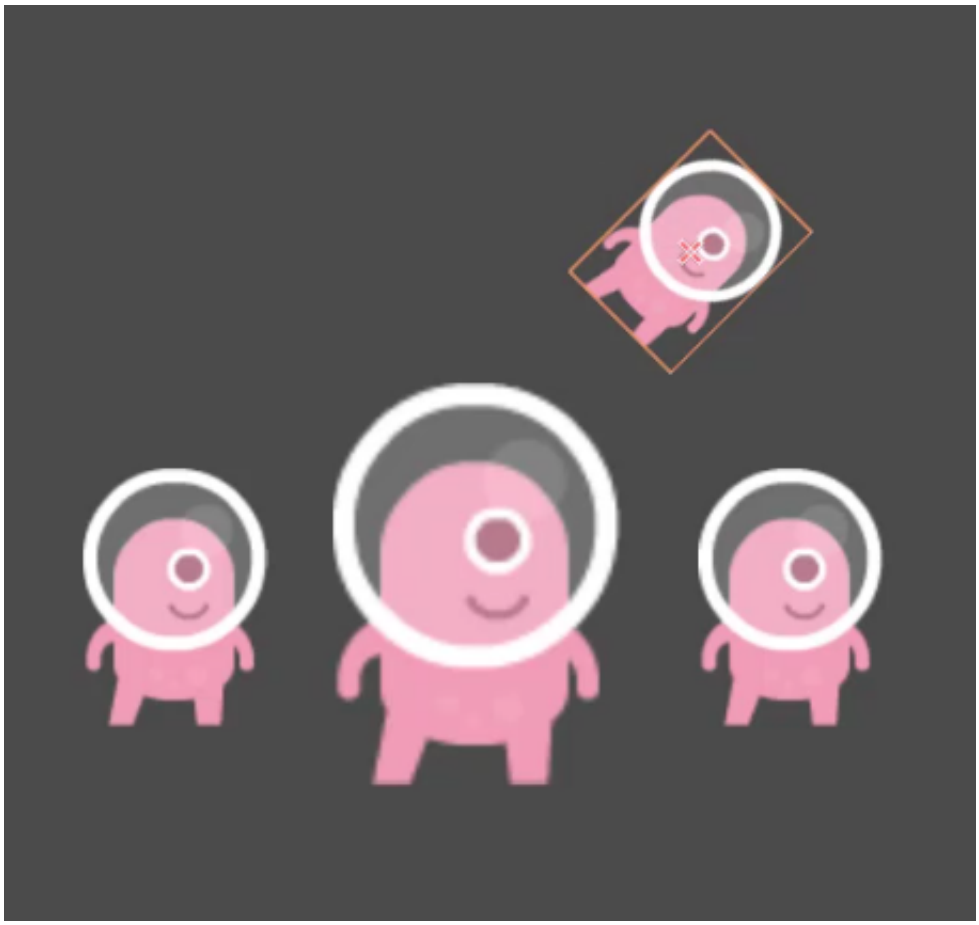
Now, let’s add some rotated sprites to the layout. Duplicate another node and move it into position.



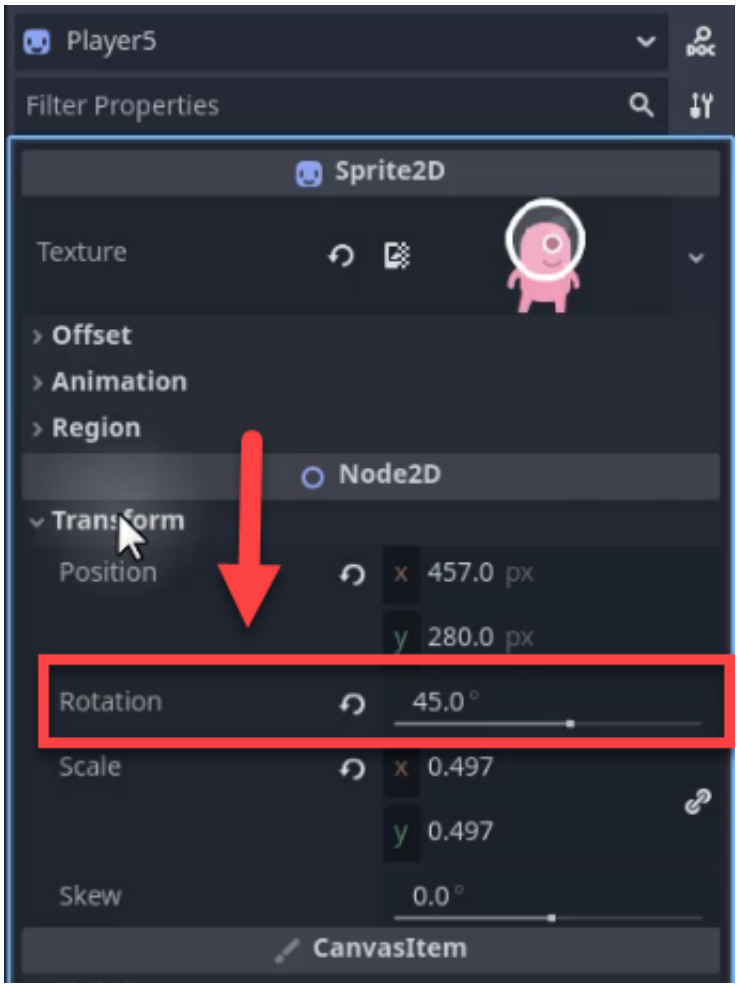
Use the **Scale Tool** to scale this sprite slightly smaller than the two to the side.



Press E to activate the **Rotate Tool**, then click and drag around the sprite to rotate it. To snap the rotation to increments (e.g., 45 degrees), hold Ctrl while rotating.



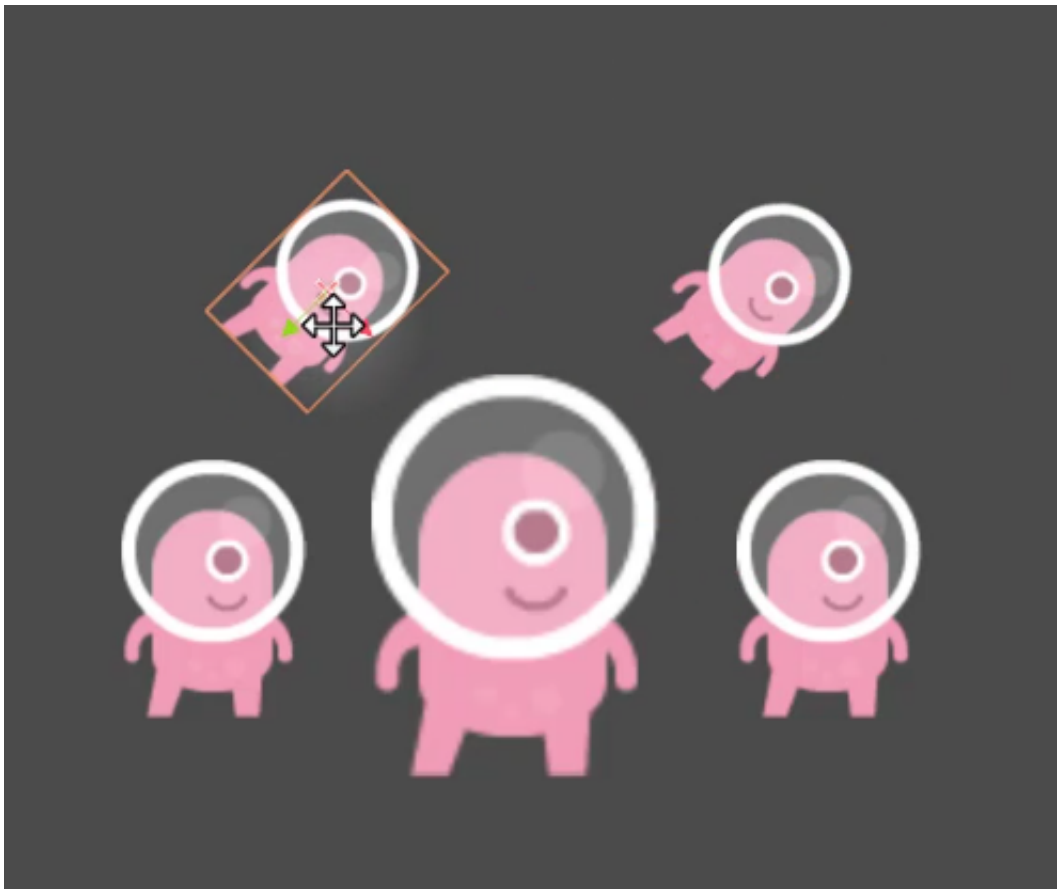
Alternatively, you can manually adjust the rotation in the **Transform** properties in the *Inspector*.



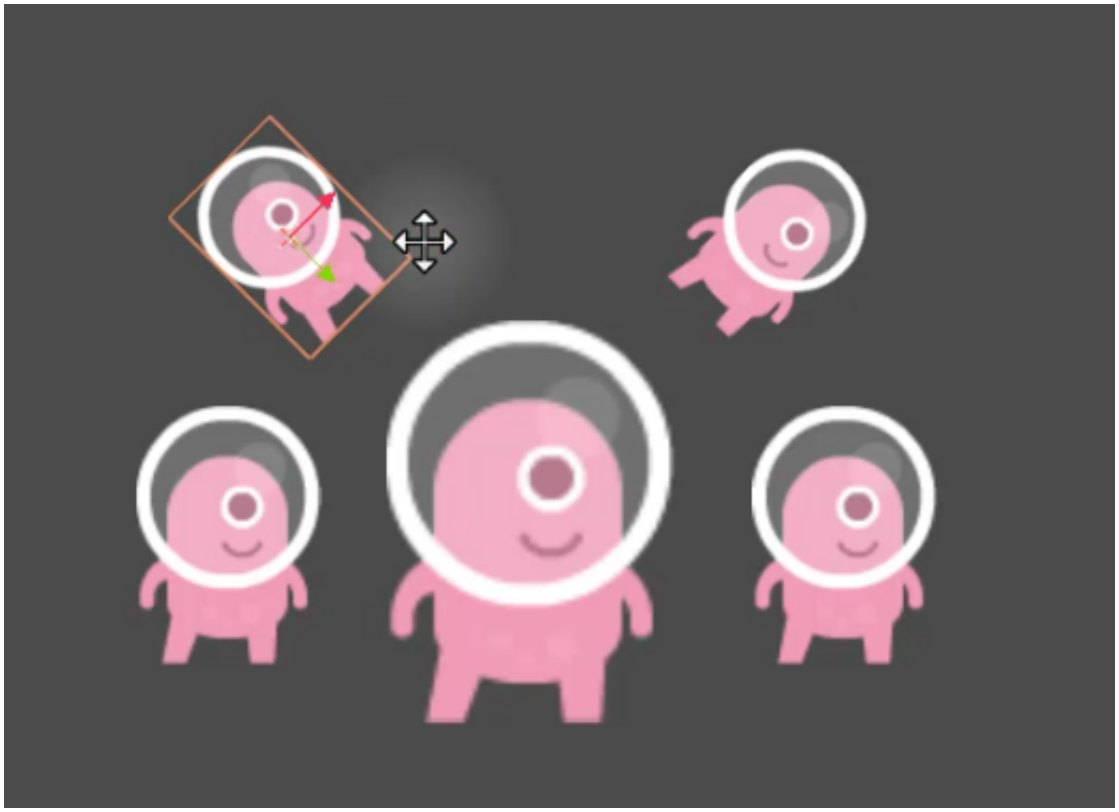
Align and Adjust Nodes

To finish the scene, follow these steps to duplicate and align more nodes:

1) Duplicate and position additional nodes on the opposite side, ensuring they are aligned along a specific axis. Hold Shift while dragging to snap to a single axis.



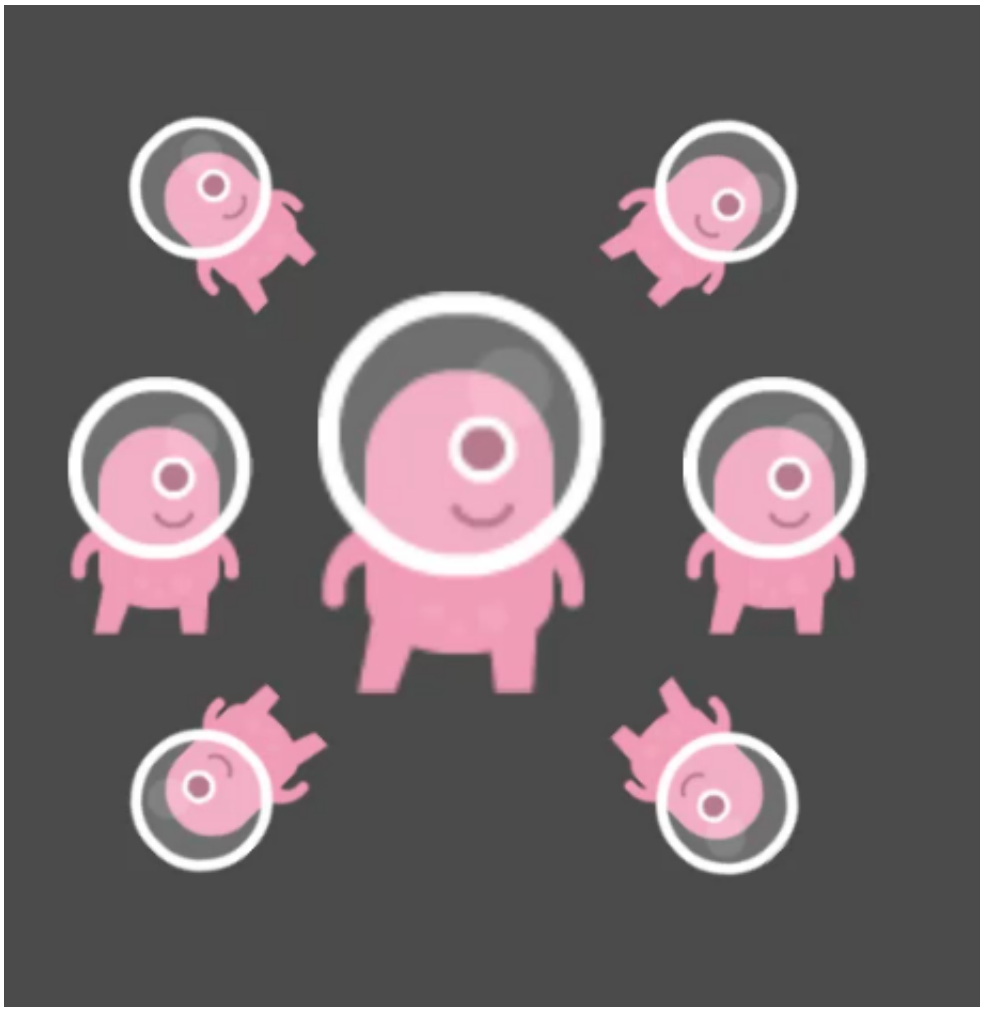
2) To create mirrored or flipped versions, use the **Rotate Tool** or flip the scale values (e.g., setting the X-scale to -1).



You can also select multiple nodes at once (hold Shift and click each), then move or rotate them together for efficiency.

Finalizing the Scene

After rearranging and adjusting the nodes, you should have a setup that resembles the challenge goal. Be sure to double-check alignment and rotation for accuracy.



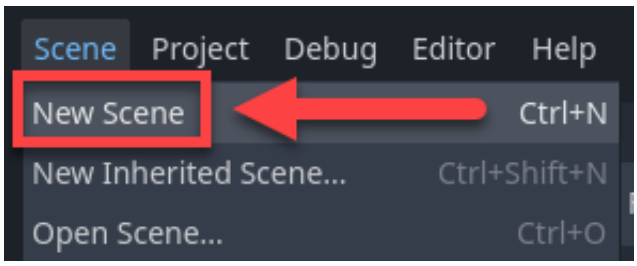
What’s Next?

Now that you’ve completed the challenge, we encourage you to experiment further with the **Move**, **Scale**, and **Rotate** tools. These tools are essential for designing levels and arranging elements in your game world. Practicing these skills will help you gain confidence in setting up your scenes effectively.

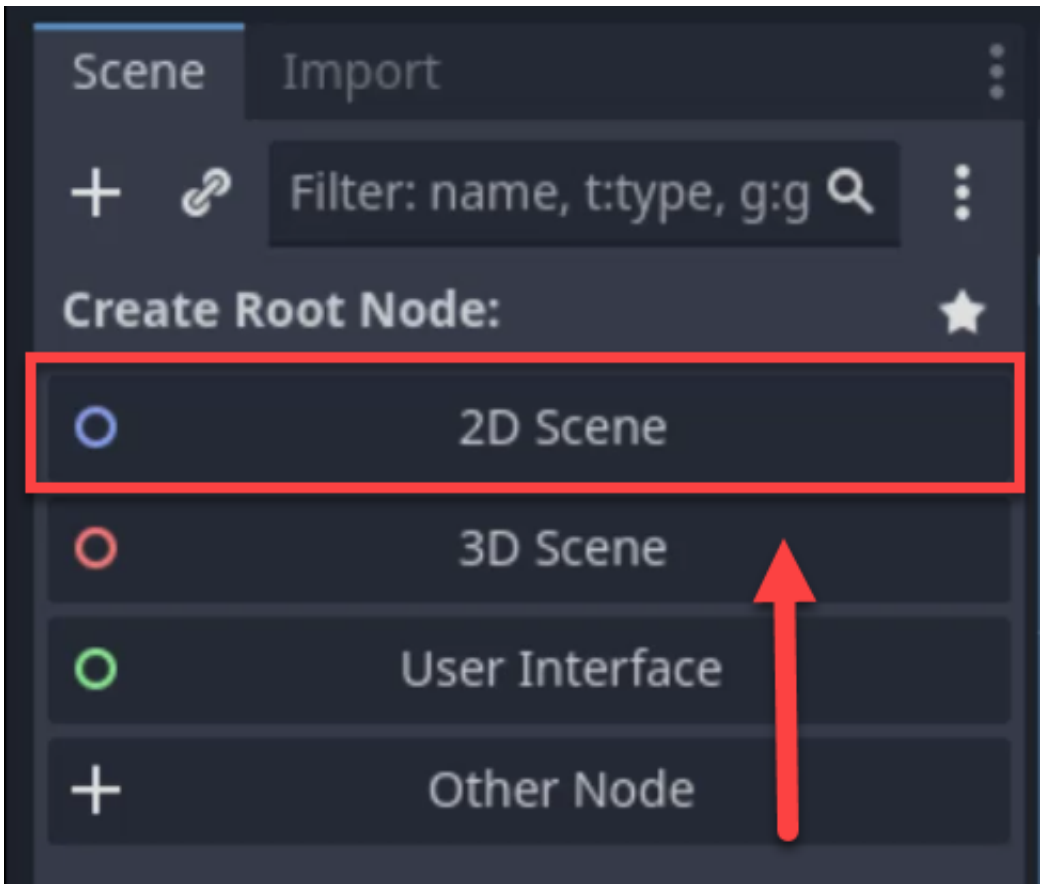
In this tutorial, we will explore the concept of parenting and child nodes, which is fundamental to creating complex behaviors in your games.

Creating a New Scene

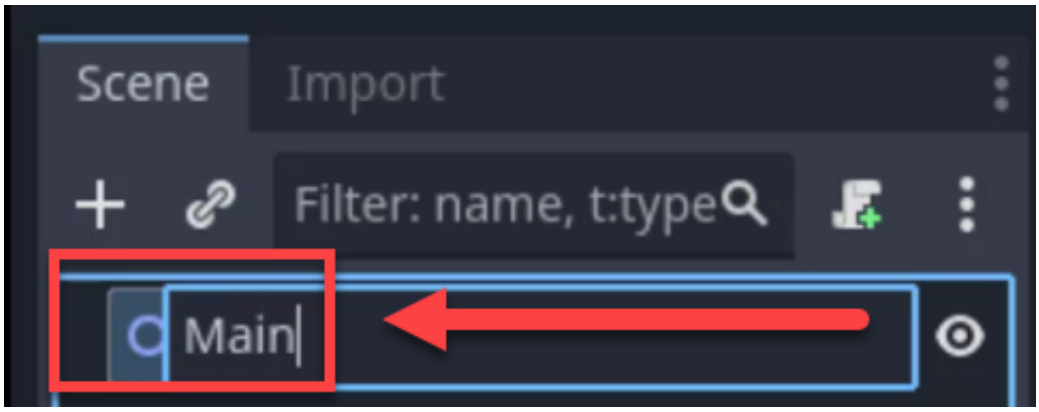
Let's start by creating a new scene to have a clean slate. Click on the **Scene** button at the top left corner of your screen and select **New Scene** (or use the shortcut Ctrl + N.)



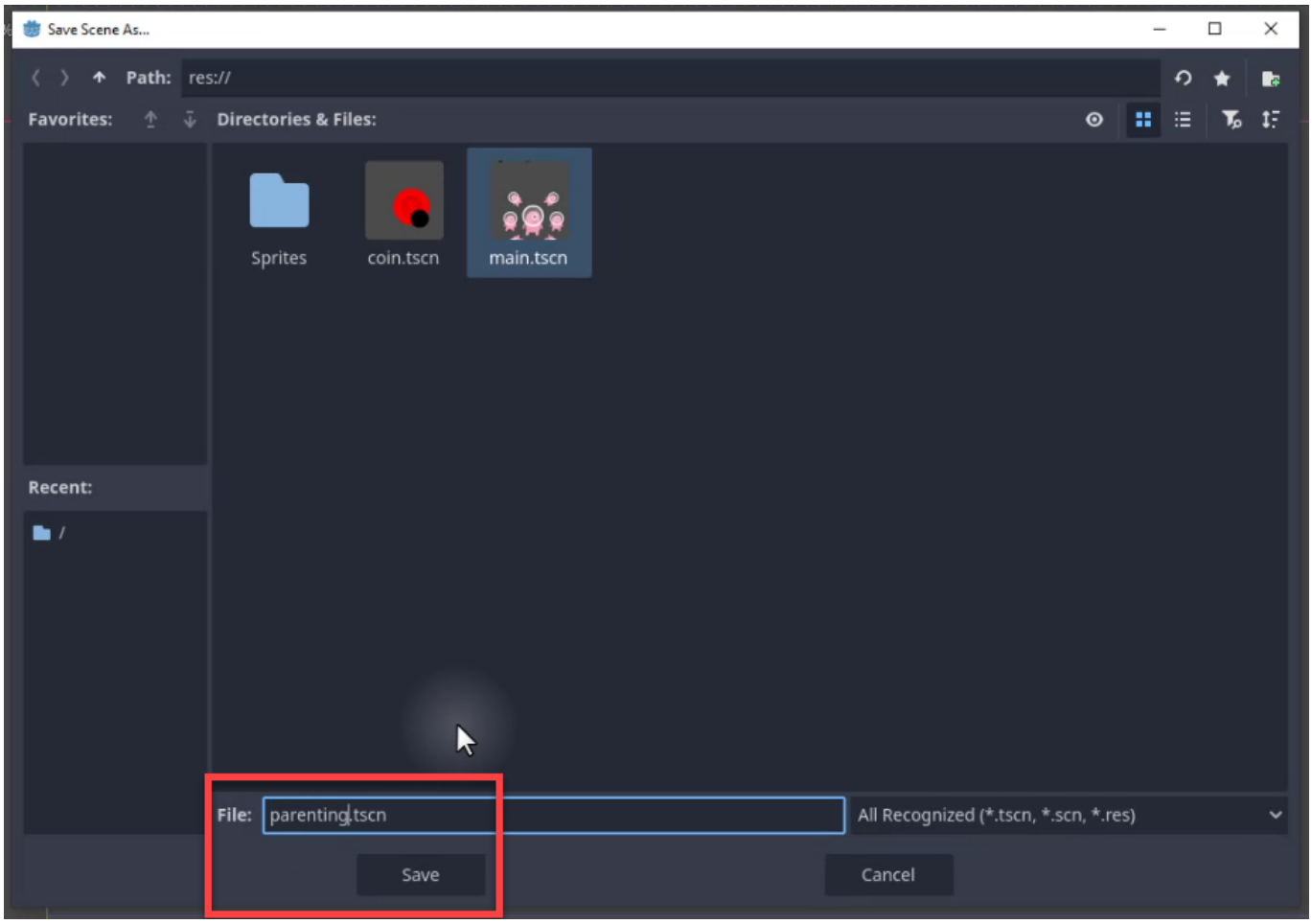
Choose **2D Scene** as the root node.



Rename the root node to something like **Main**.



Save the scene using Ctrl + S and name it **parenting.tscn**.

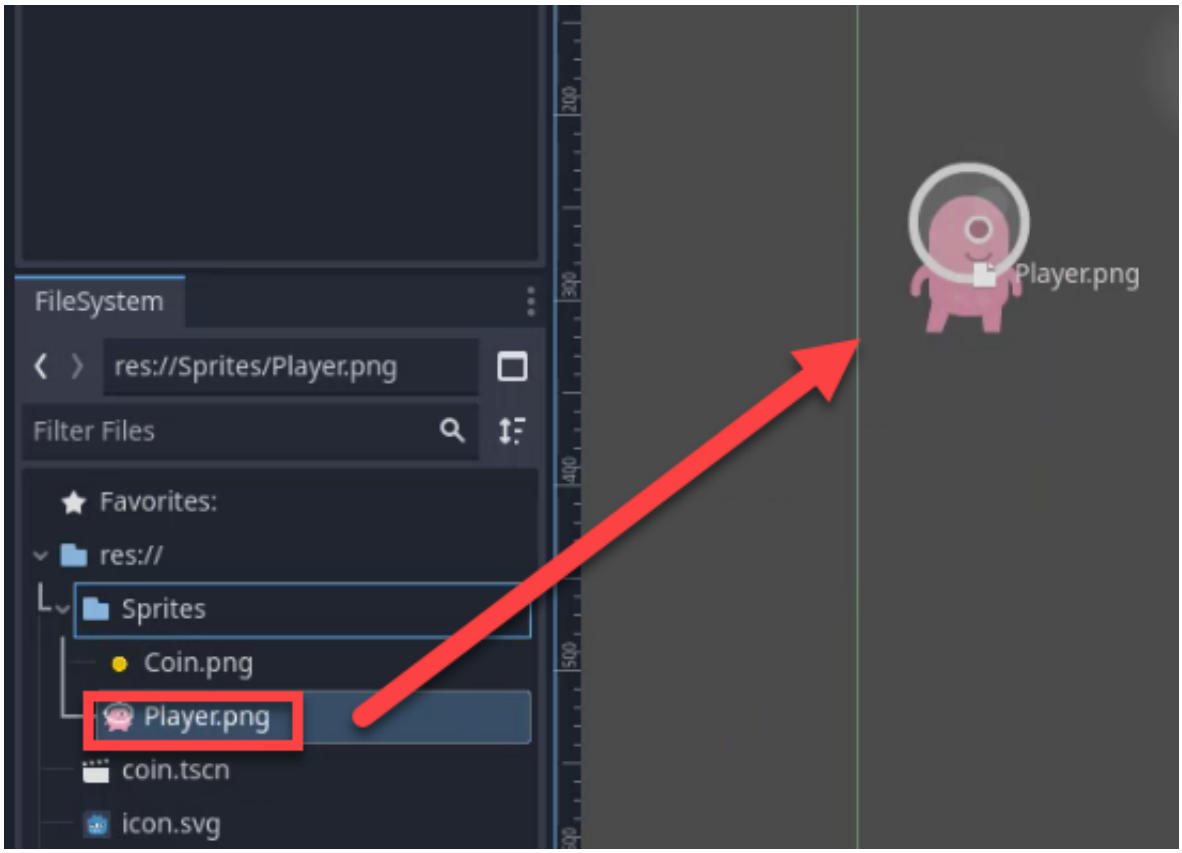


Understanding Parenting and Child Nodes

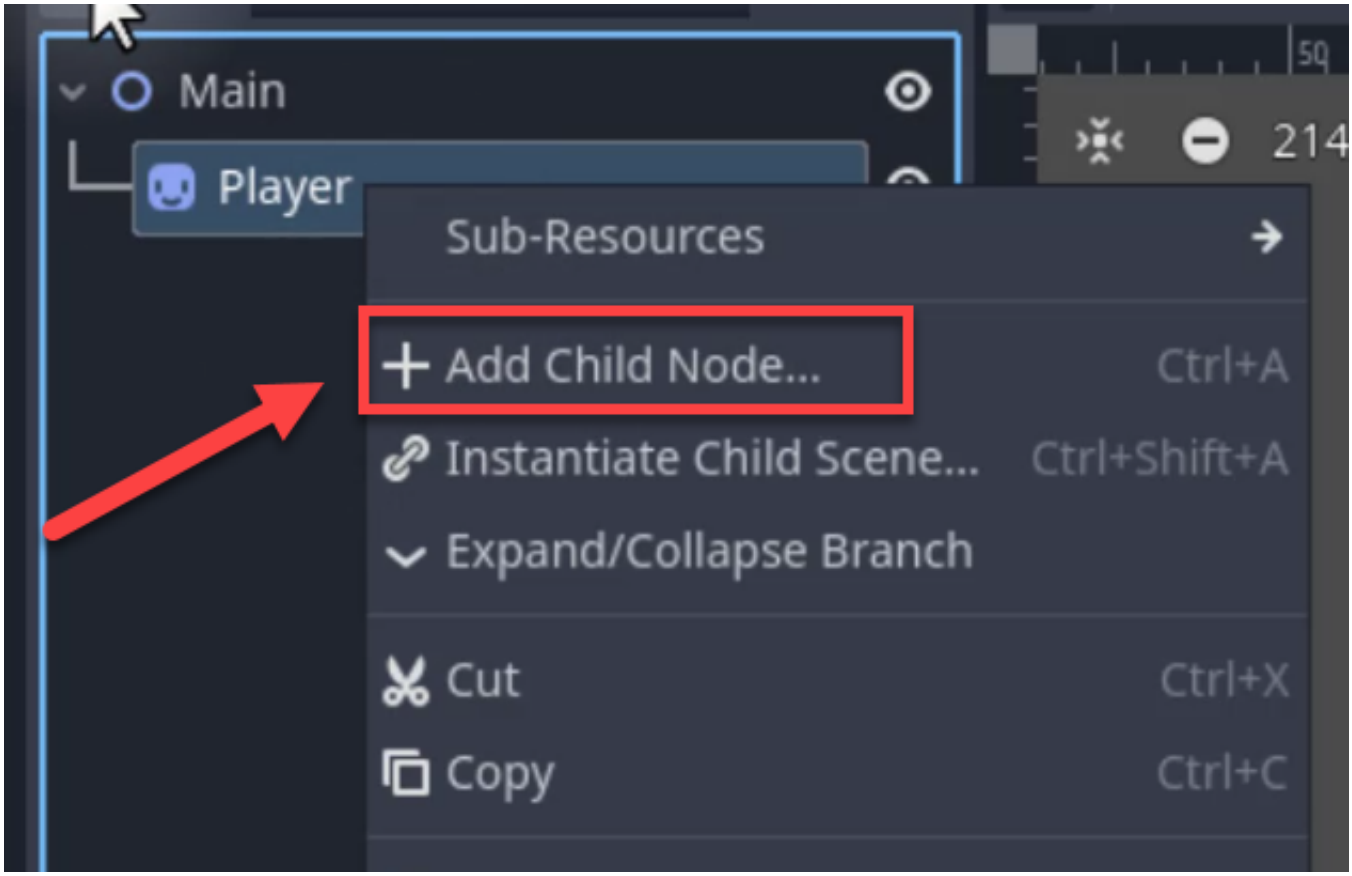
In Godot, nodes can have parent-child relationships. This means that a child node will follow the transformations (movement, rotation, scaling) of its parent node. This is crucial for creating complex objects that have multiple components.

Example: Creating a Player Node with Child Nodes

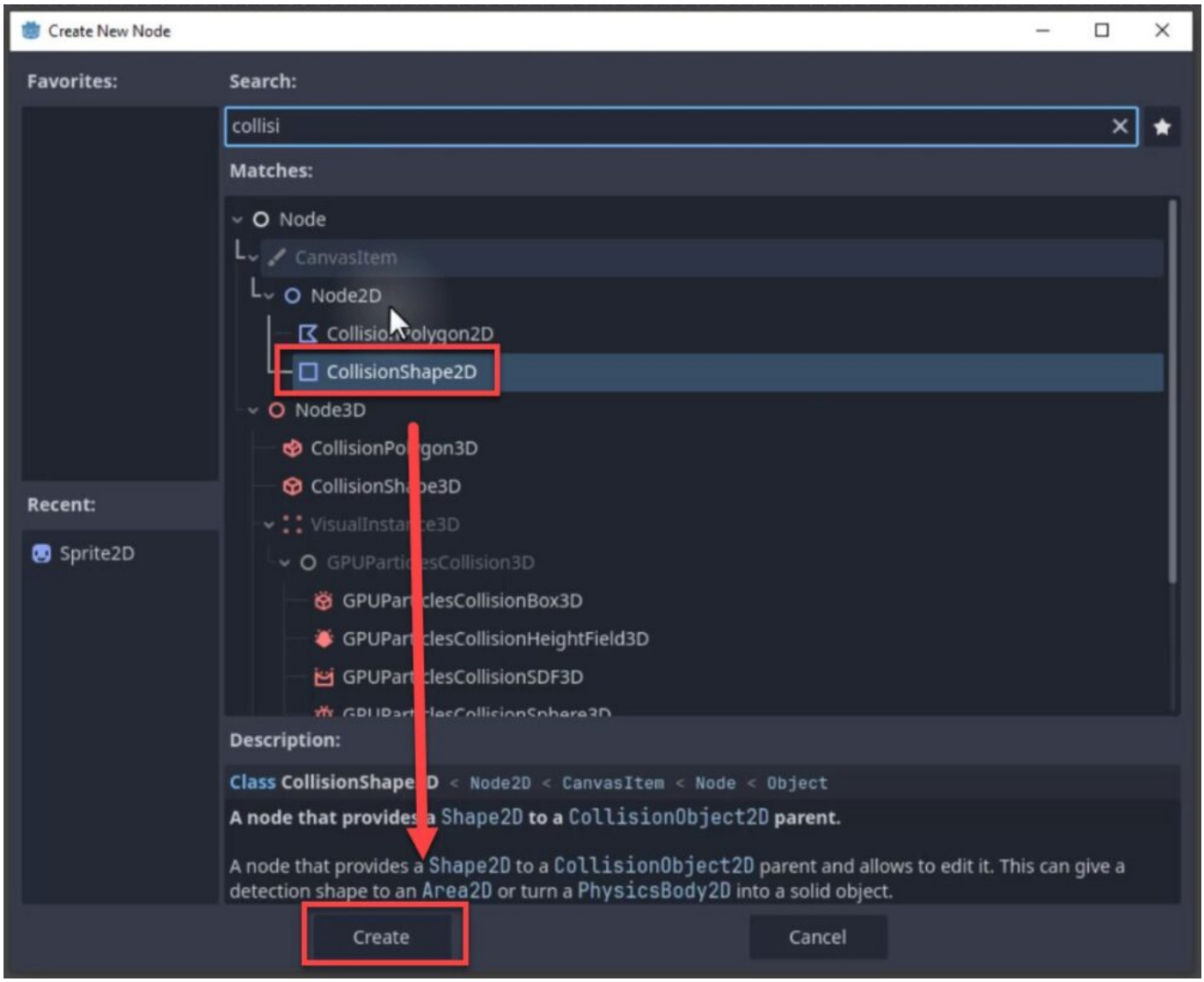
Let’s showcase this idea with an example. Drag your player PNG into the scene to create a **Sprite2D** node.



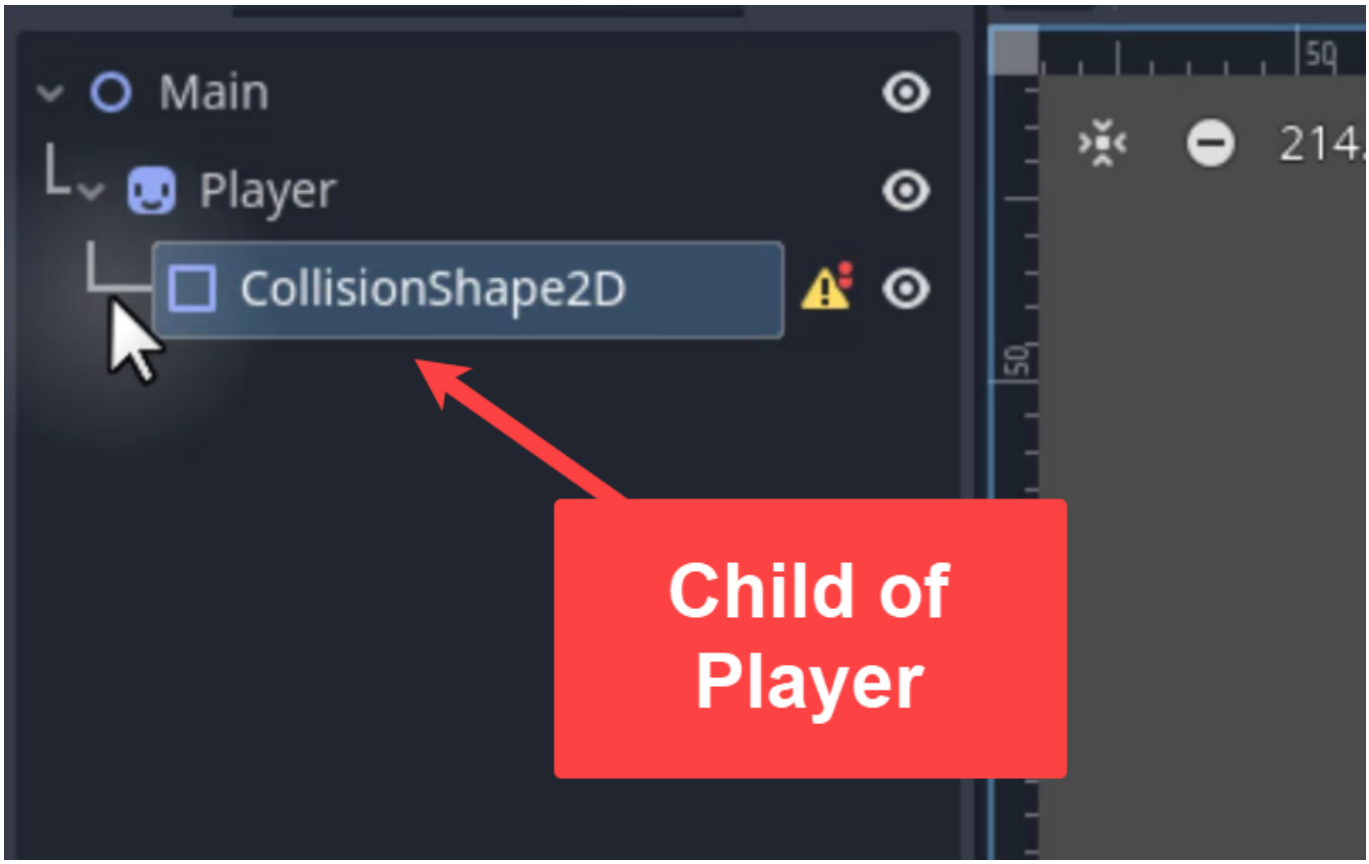
Right-click on the player node and select **Add Child Node**.



Search for and add a **CollisionShape2D** node.

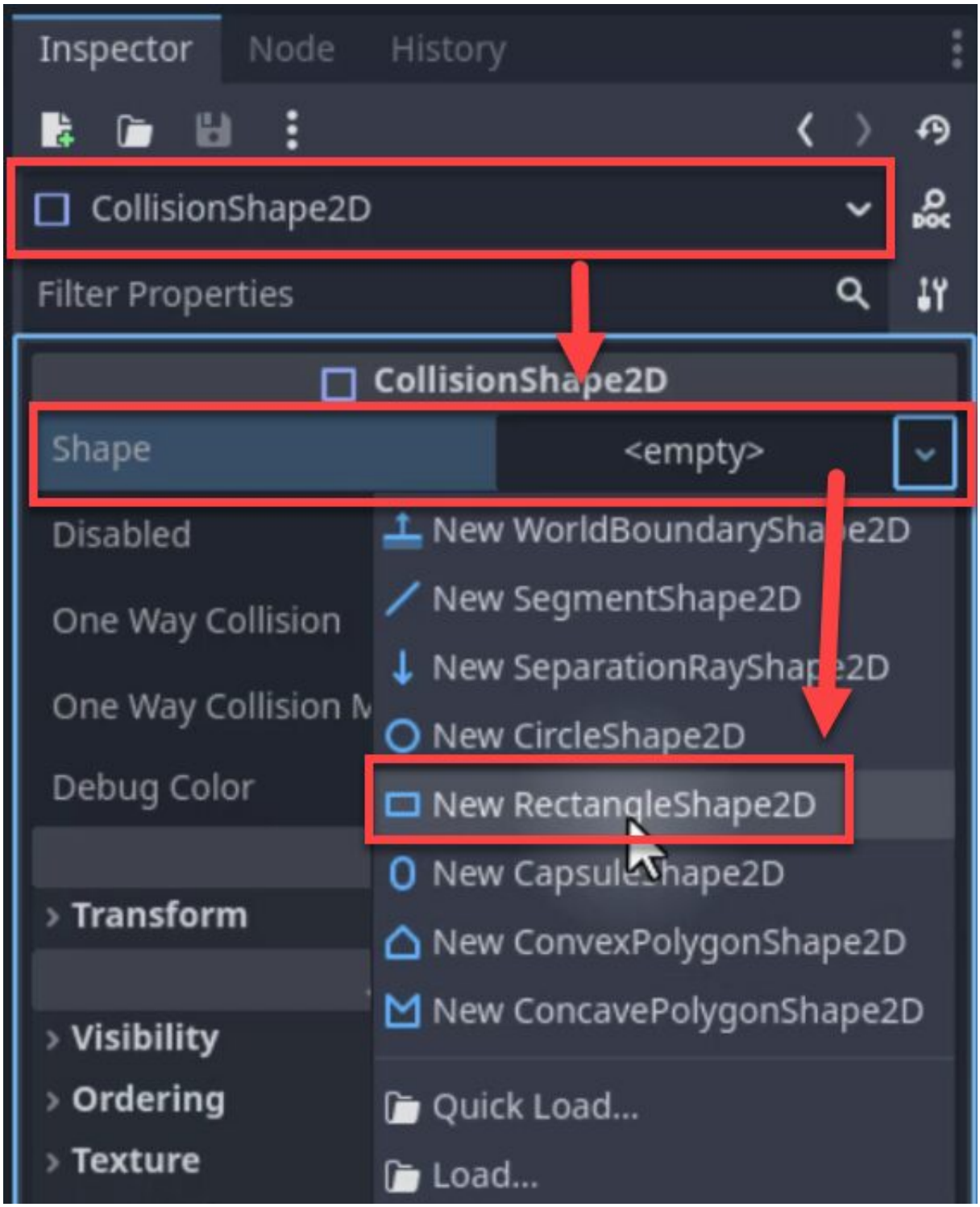


Notice how the **CollisionShape2D** node is indented under the player node, indicating it is a child of the player node.

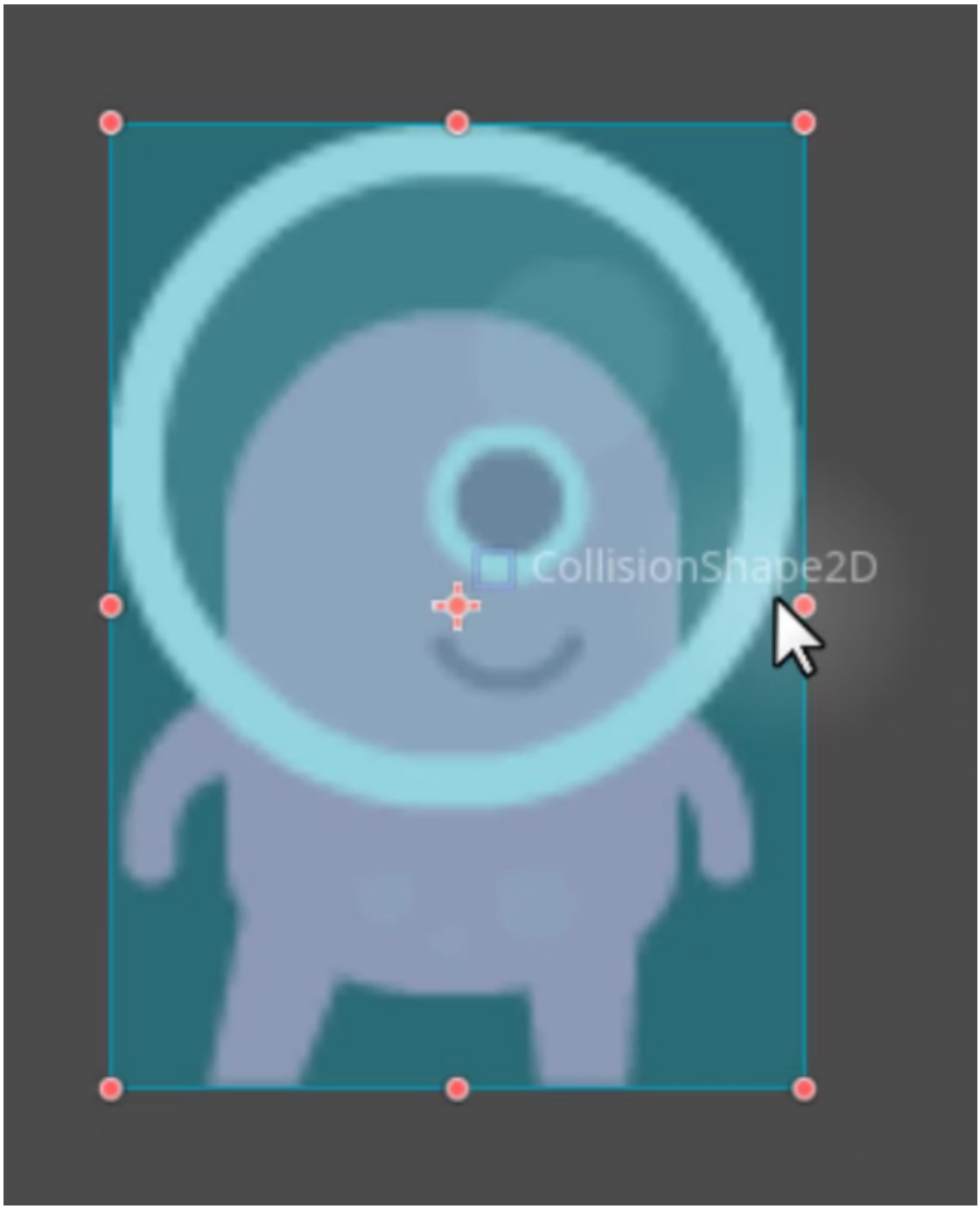


Configuring the Collision Shape

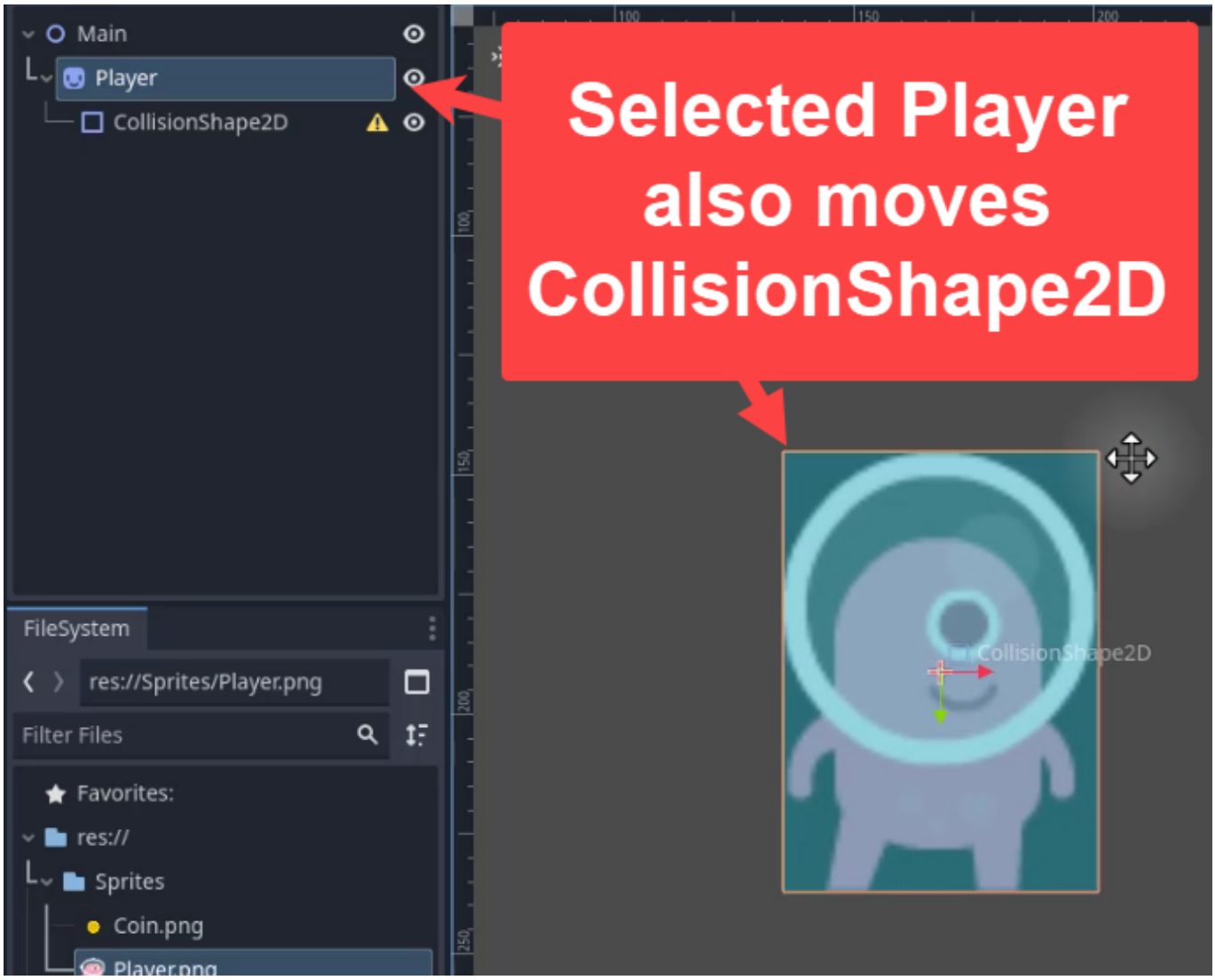
To continue our example, let’s set up a collision shape. Select the **CollisionShape2D** node. In the Inspector, find the **Shape** property and select **New RectangleShape2D**.



Adjust the size of the collision shape to fit the player sprite.

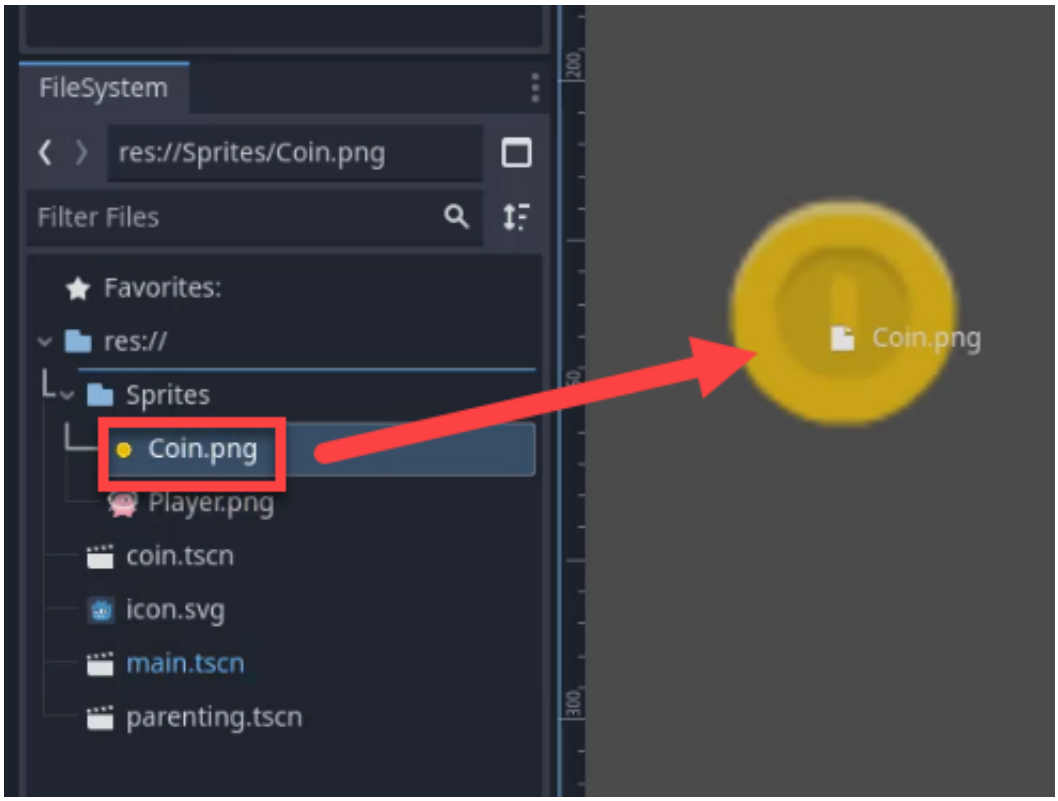


Now, when you move, rotate, or scale the player node, the collision shape will follow along.

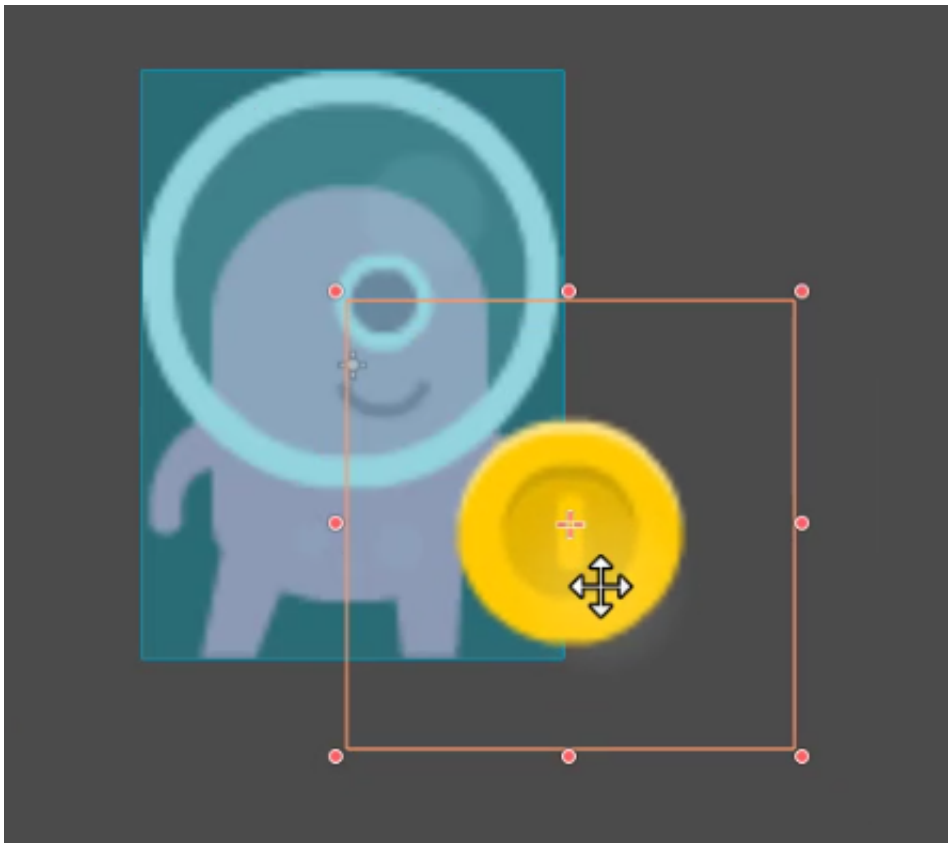


Adding More Child Nodes

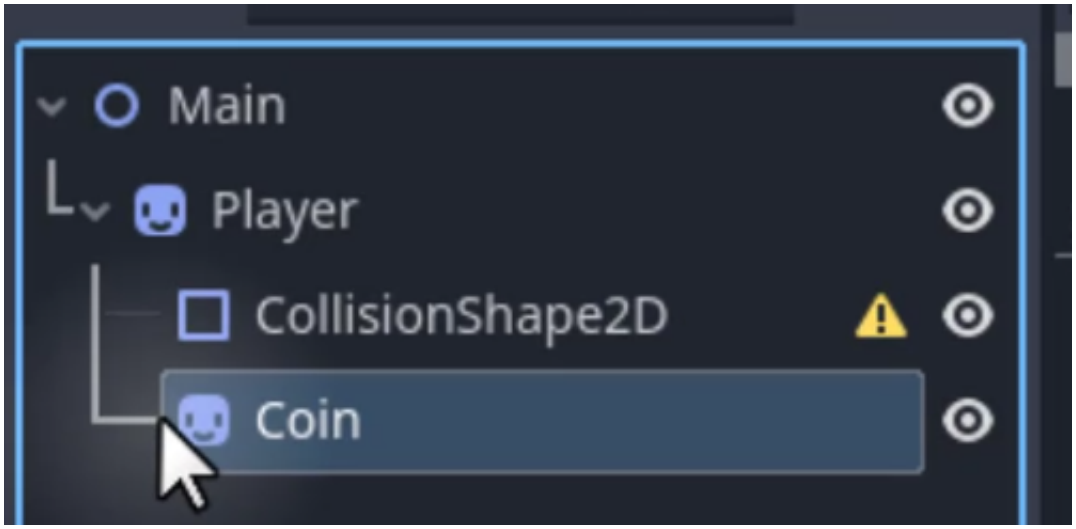
Let’s add another child node to the player, such as a coin that the player is holding. Drag the coin PNG into the scene.



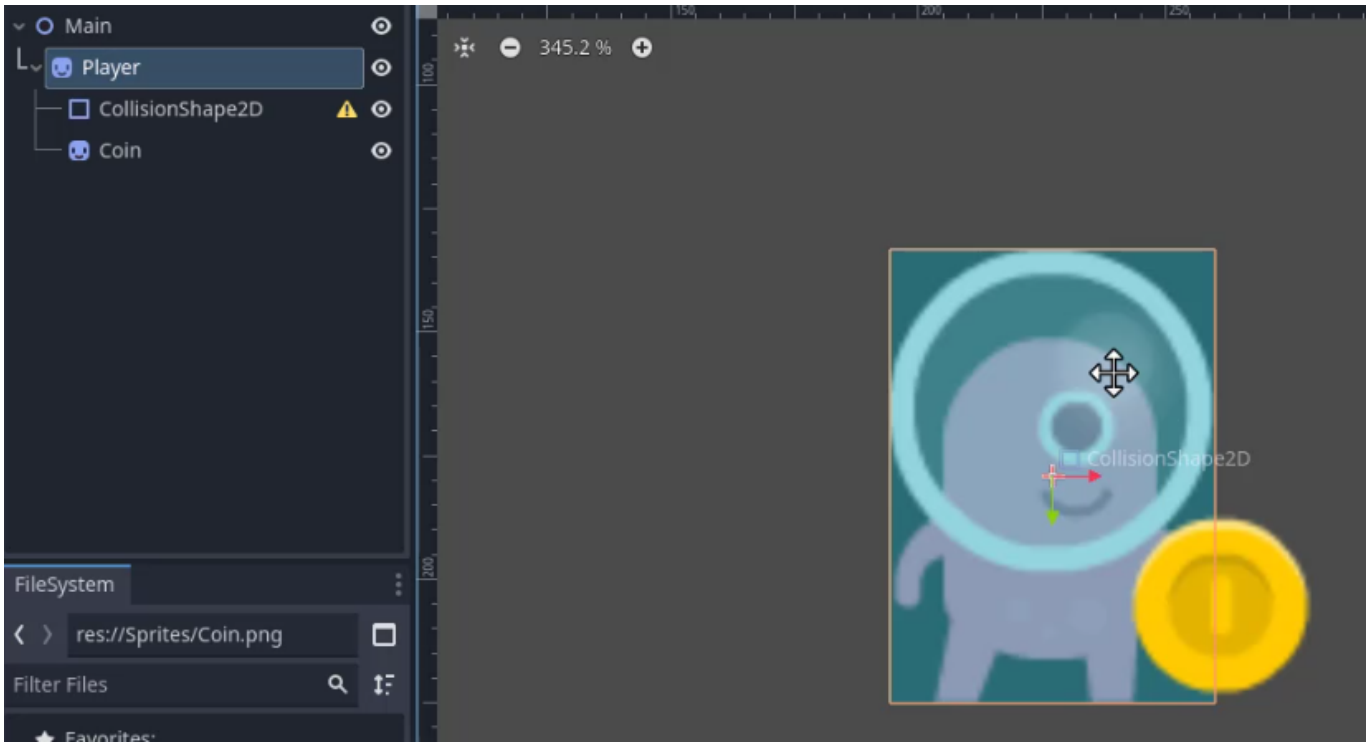
Position it over the player's hand.



Click and drag the coin node onto the player node to make it a child of the player node.



Now, when you move the player node, both the collision shape and the coin will follow along.



Experimenting with Parenting

To fully understand the concept of parenting and child nodes, it’s important to experiment:

- Create different nodes and make them children of other nodes.
- Observe how they react when the parent node is moved, rotated, or scaled.
- Try creating a hierarchy of nodes, where a node is a child of another child node.

Conclusion

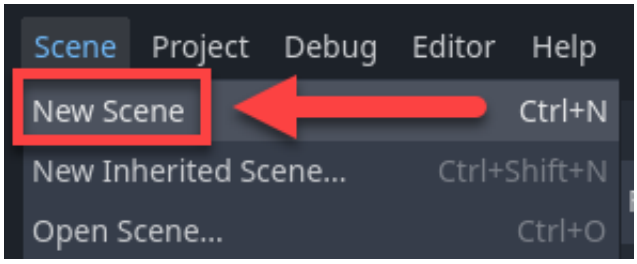
Parenting and child nodes are a powerful concept in Godot that allows for complex behaviors in your games. Don’t worry if you don’t fully understand it yet. As we continue through this course, we will explore more ways to implement these relationships. For now, experiment with different nodes and see how they interact.

In the next lesson, we will take a break from 2D and start having a look at 3D capabilities in Godot!

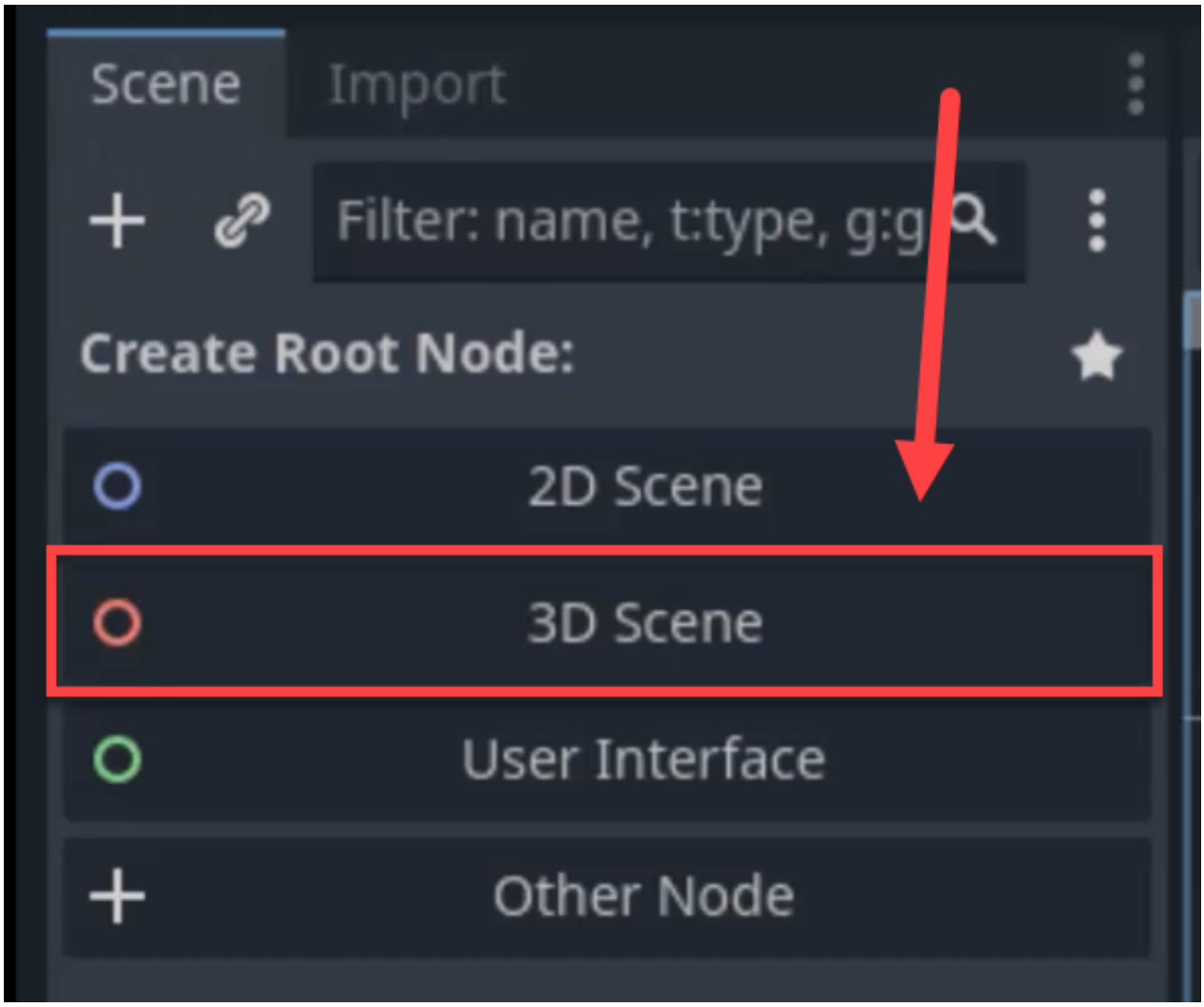
In this lesson, we will delve into the fascinating world of 3D in the Godot engine. Up until now, we have been focusing on 2D projects, but it's time to add that third dimension to our skillset. Let's get started!

Creating a 3D Scene

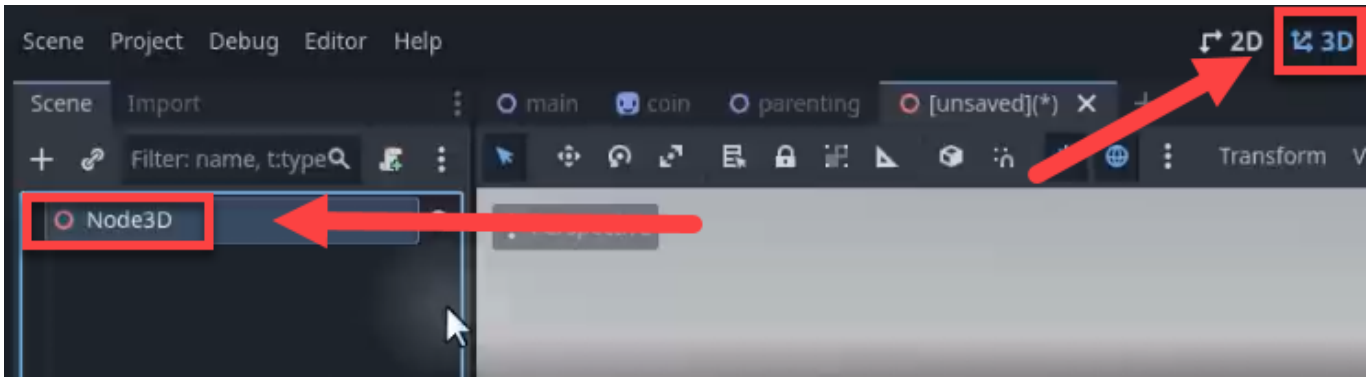
To begin, we need to create a new 3D scene. Click on **Scene** in the top menu and select **New Scene**.



For the new scene, choose **3D Scene**.



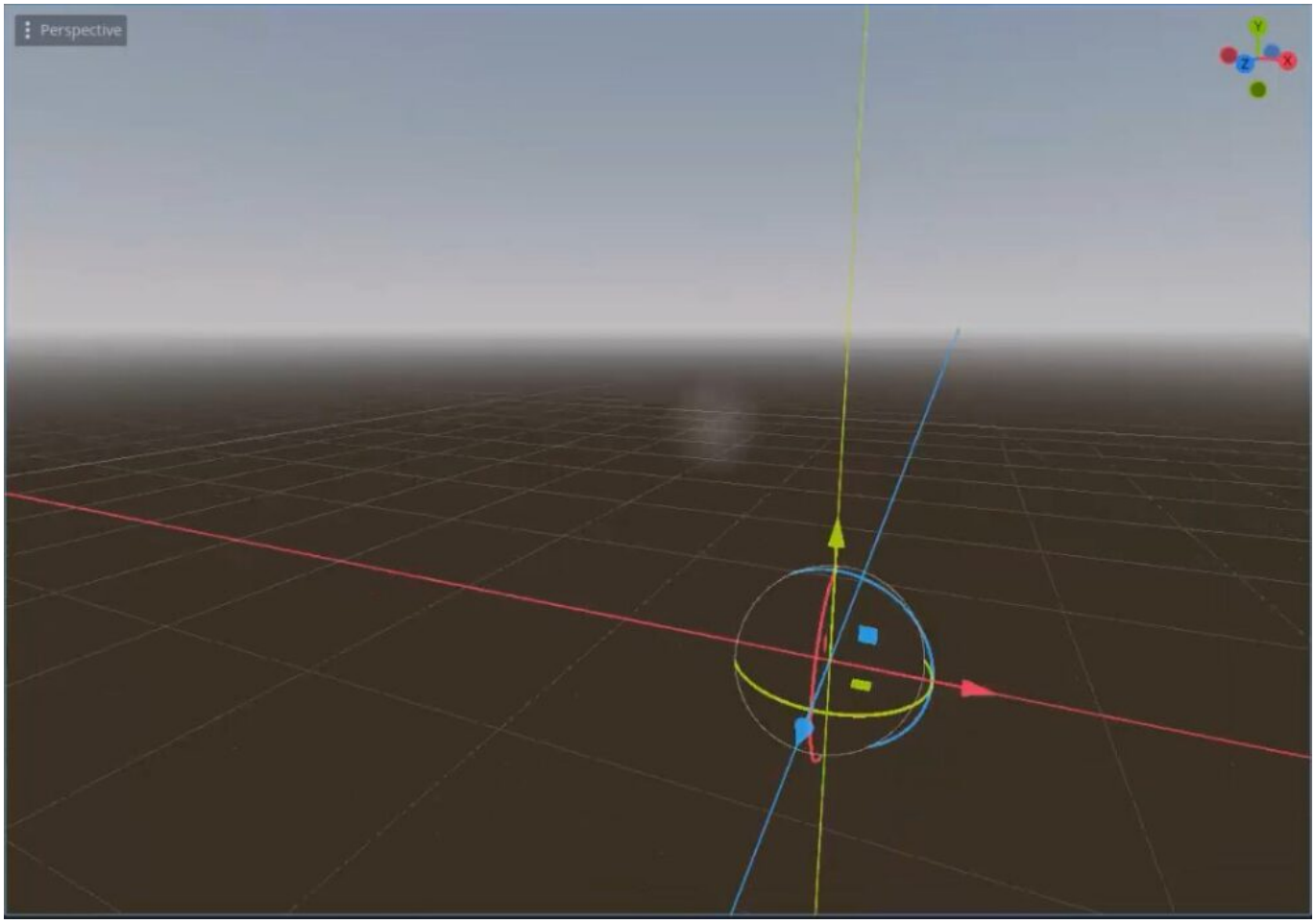
You will notice that a root node for 3D is created, indicated by a red icon. This will switch your viewport to 3D mode.



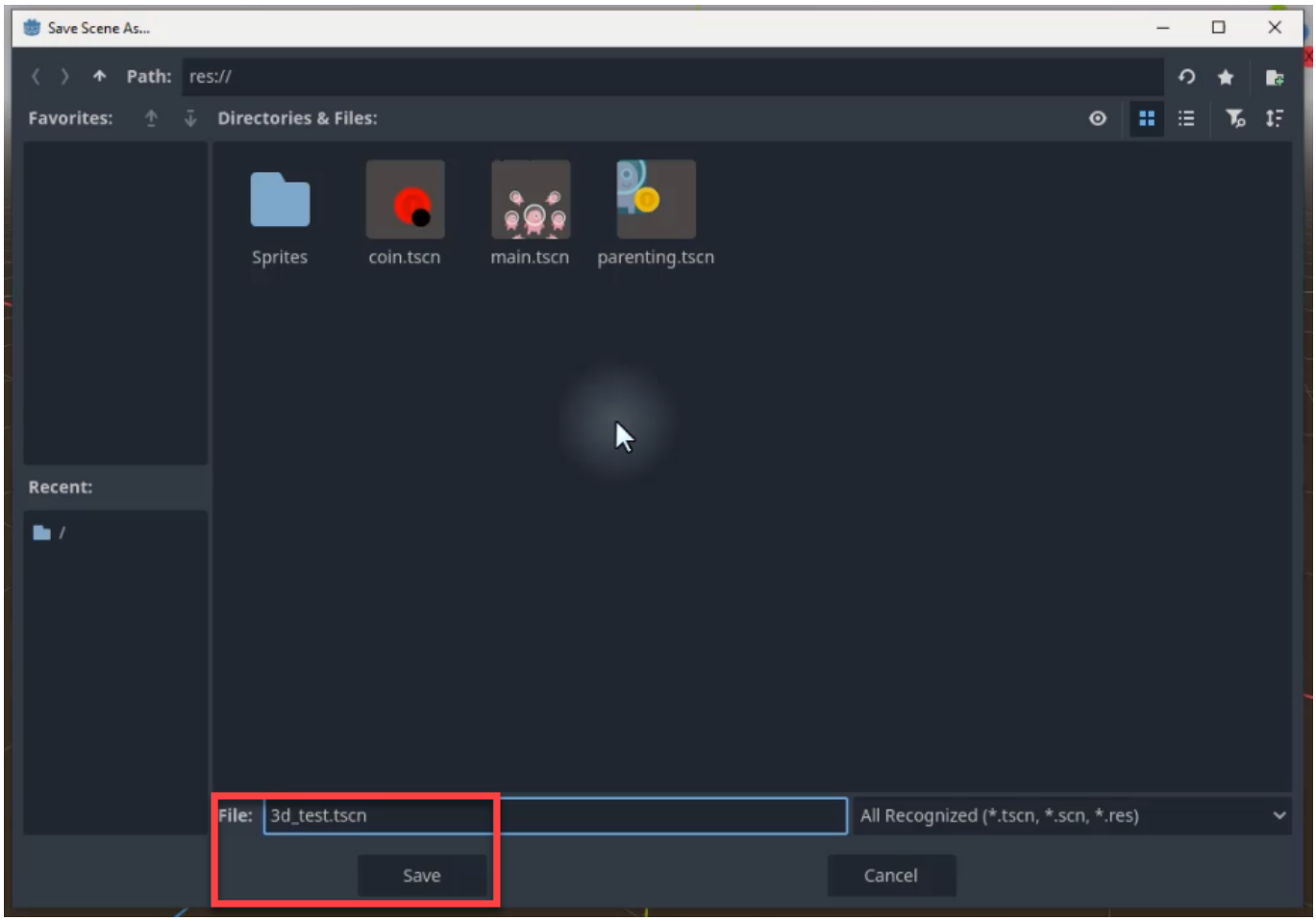
Navigating the 3D Environment

Navigating in 3D is a bit different from 2D. Here are the controls you need to know:

- Hold down the **right mouse button** to look around the camera.
- Use the **WASD keys** to fly around.
- Press **E** to go up and **Q** to go down.



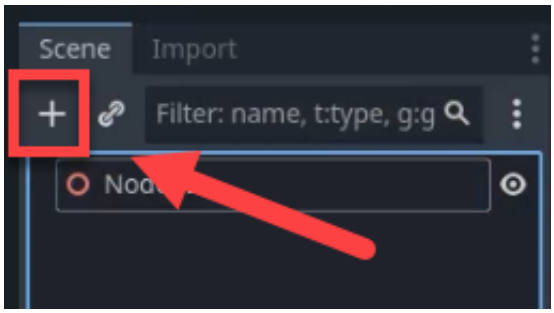
Save your scene by pressing **CTRL + S** and name it something like “3D_test”.



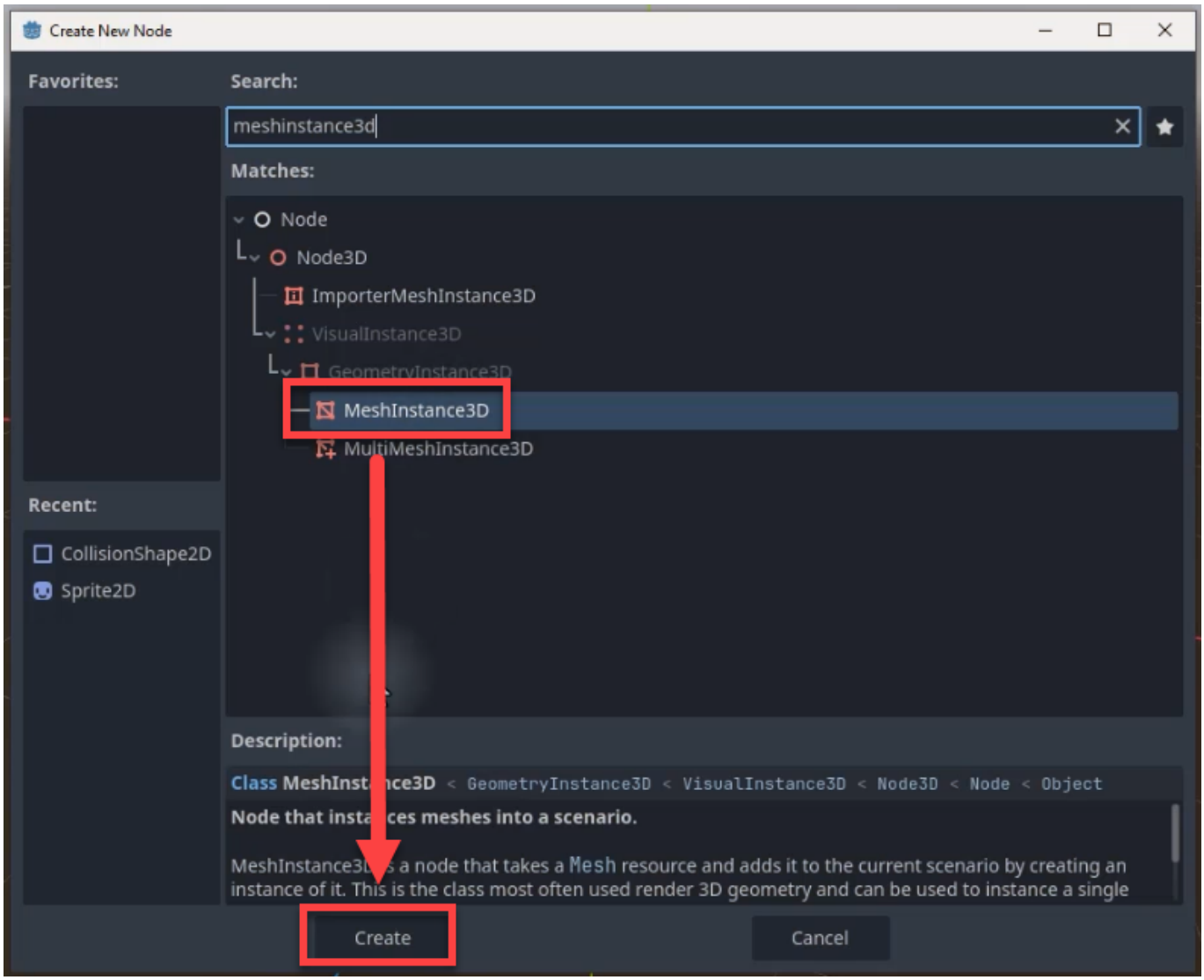
Working with 3D Models

In 2D, we used sprites to represent objects. In 3D, we use 3D models. Let's see how we can create a simple 3D object right in the engine!

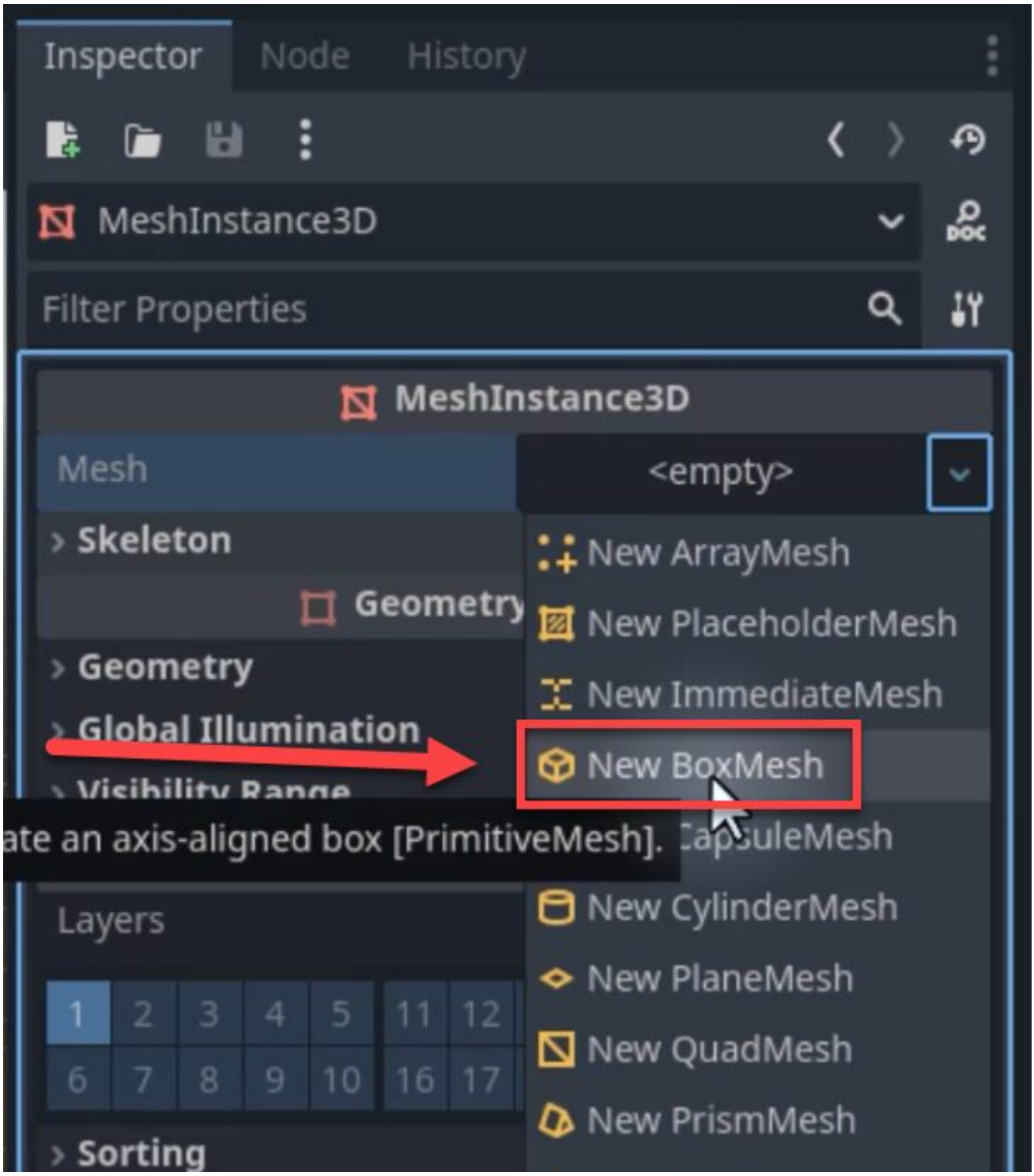
First, go to the **Scene** dock and click on the **plus icon**.



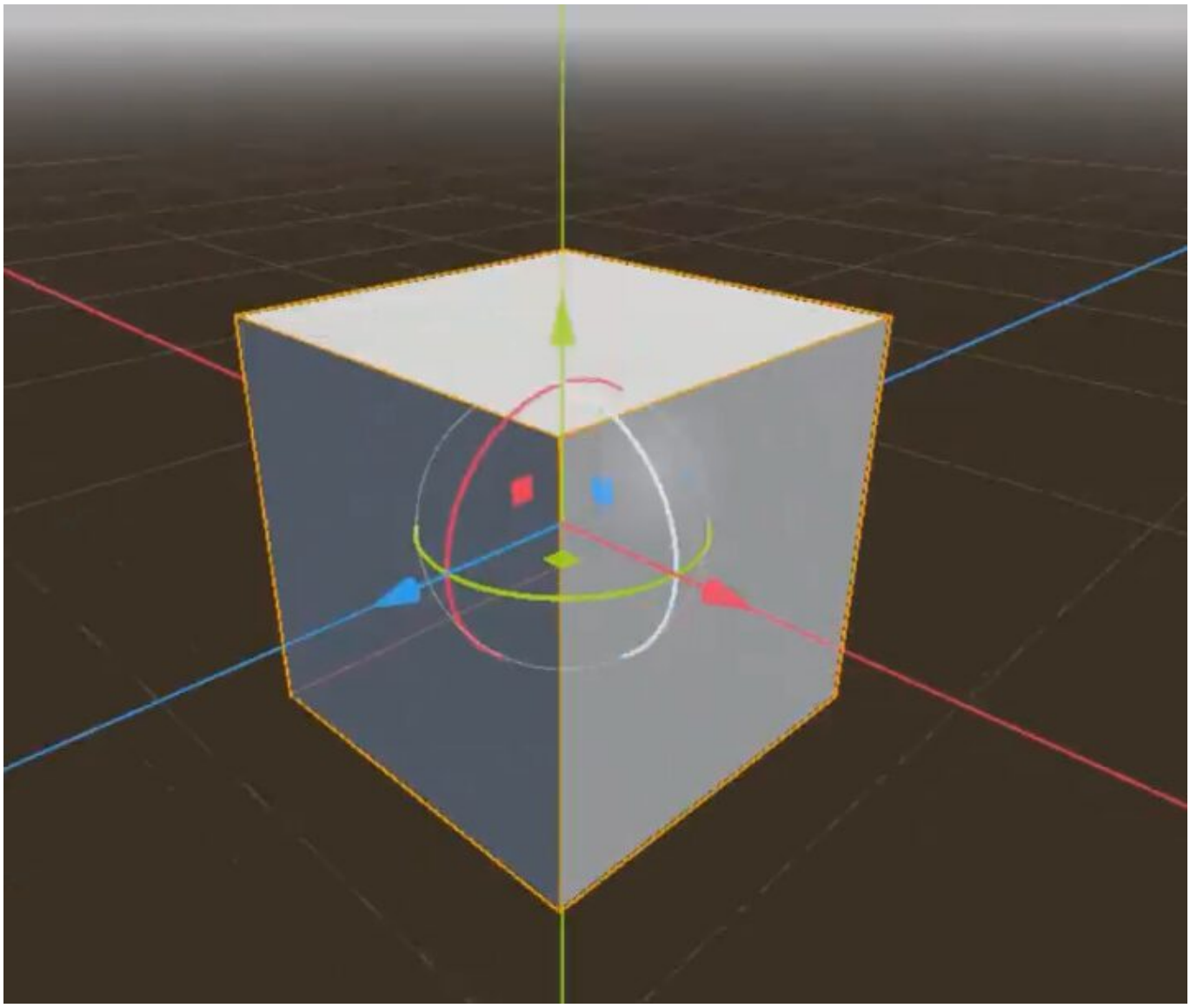
Search for and select **MeshInstance3D**.



In the **Inspector**, you will see a **Mesh** property. Click on the dropdown and select **New BoxMesh** to create a cube.

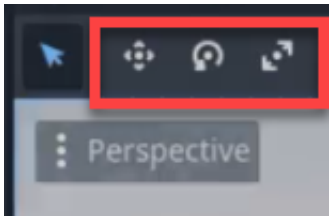


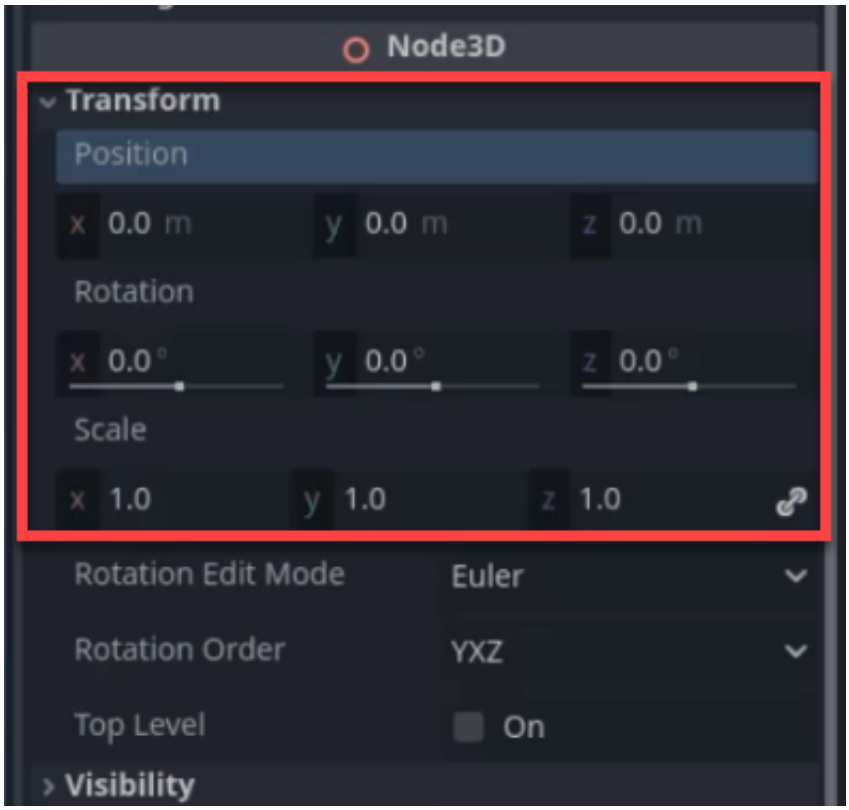
In the main scene editor, you should see a 3D cube!



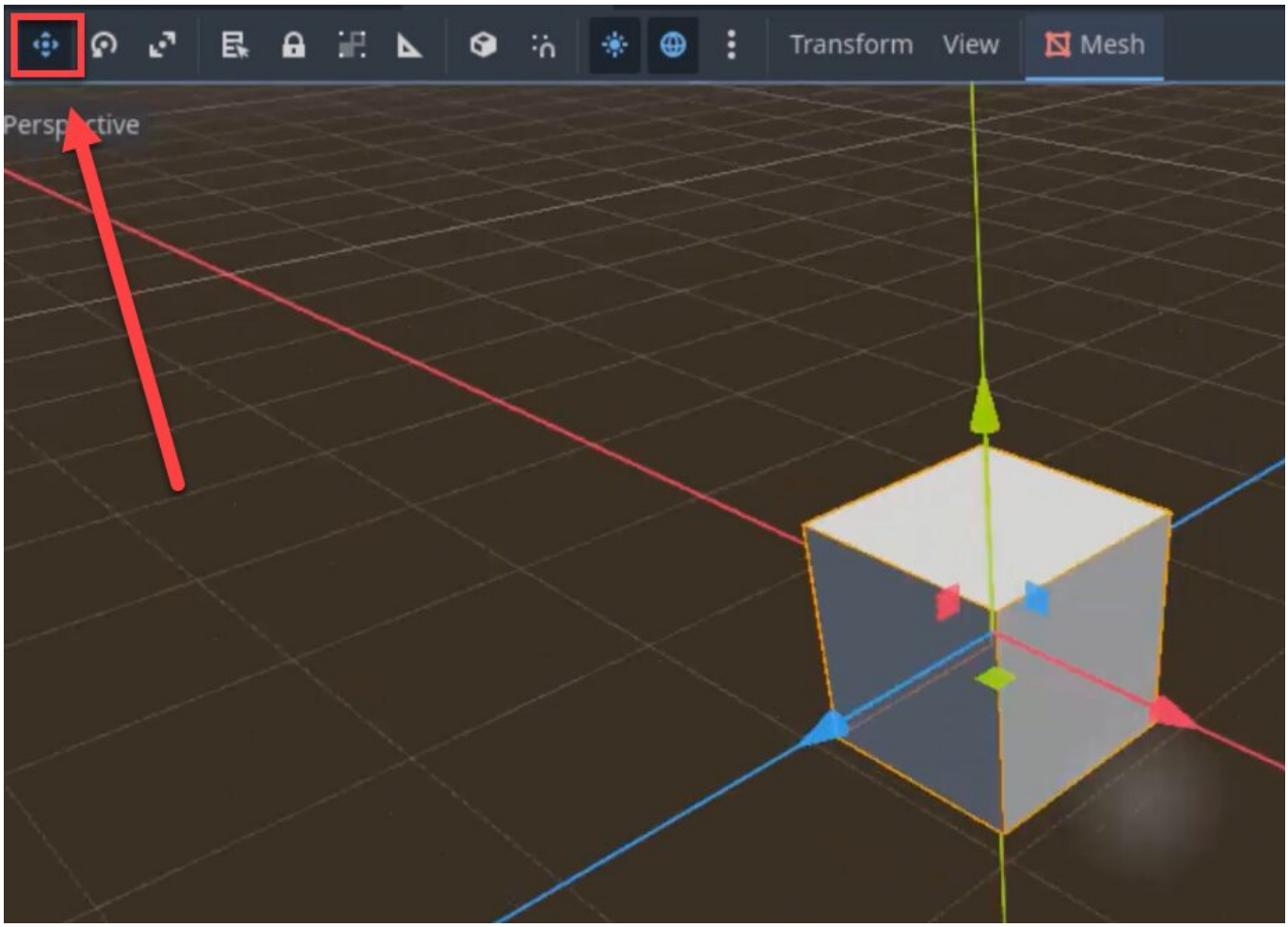
Manipulating 3D Objects

Just like in 2D, you can move, rotate, and scale your 3D objects using the node tools and using the Transform values in the Inspector.

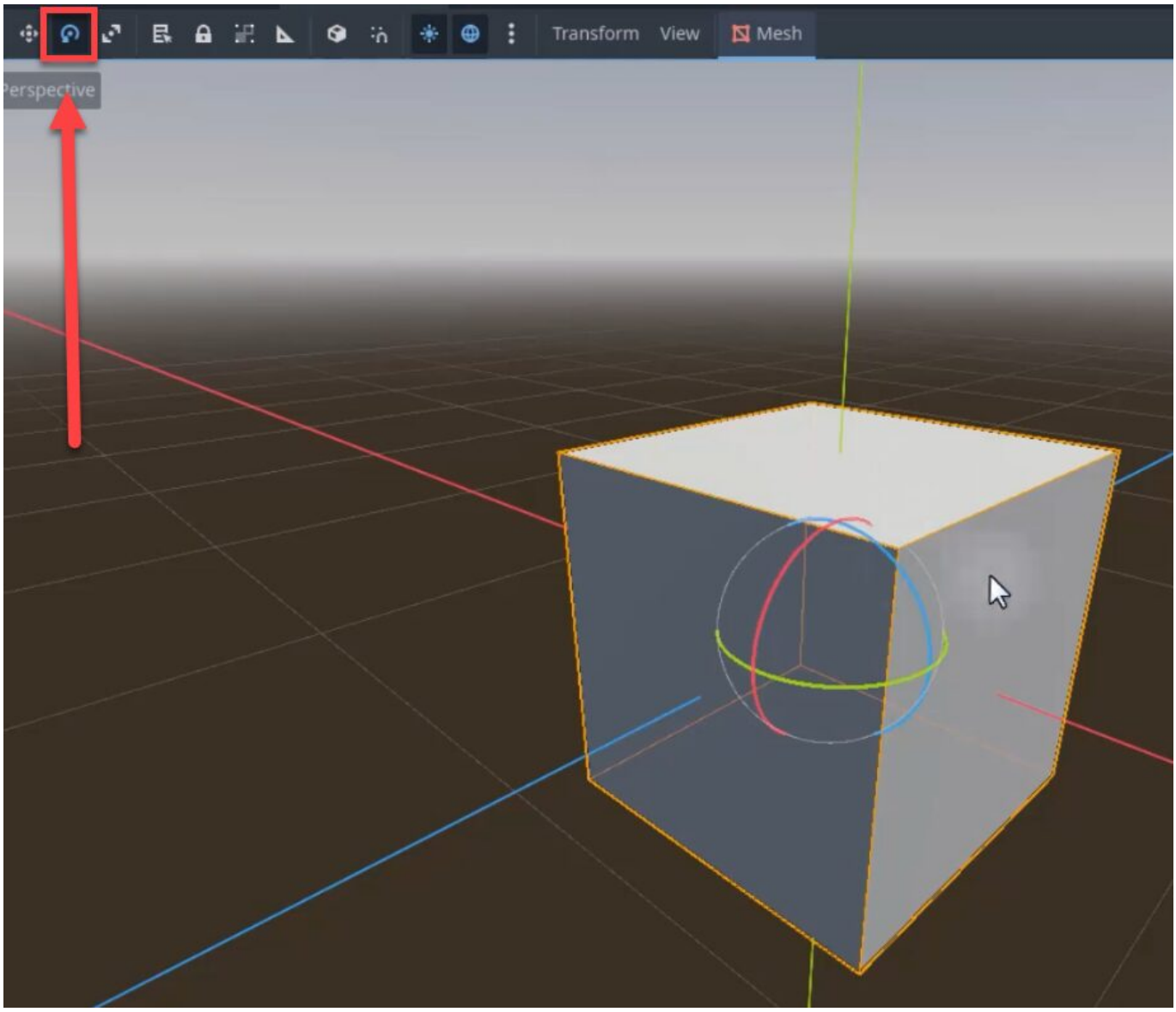




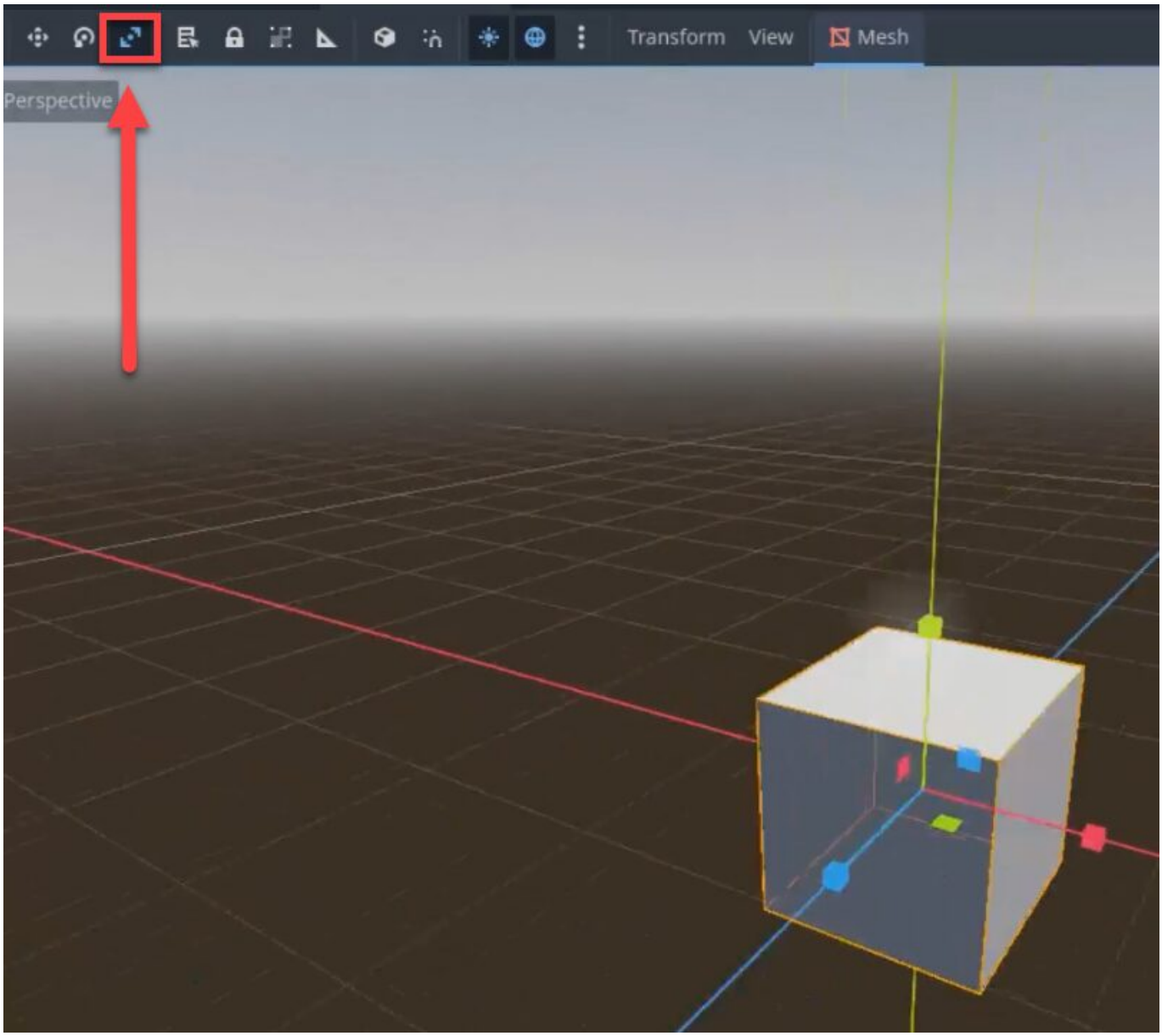
- **Move Tool**: The Move Tool allows you to move the object along the X, Y, and Z axes. In the case of Godot, X and Z represent the horizontal axes, and Y represents the vertical axis.



- **Rotate Tool**: The rotate tool enables rotation along the three axes.



- **Scale Tool**: Lets you change the size of the object uniformly or along specific axes.

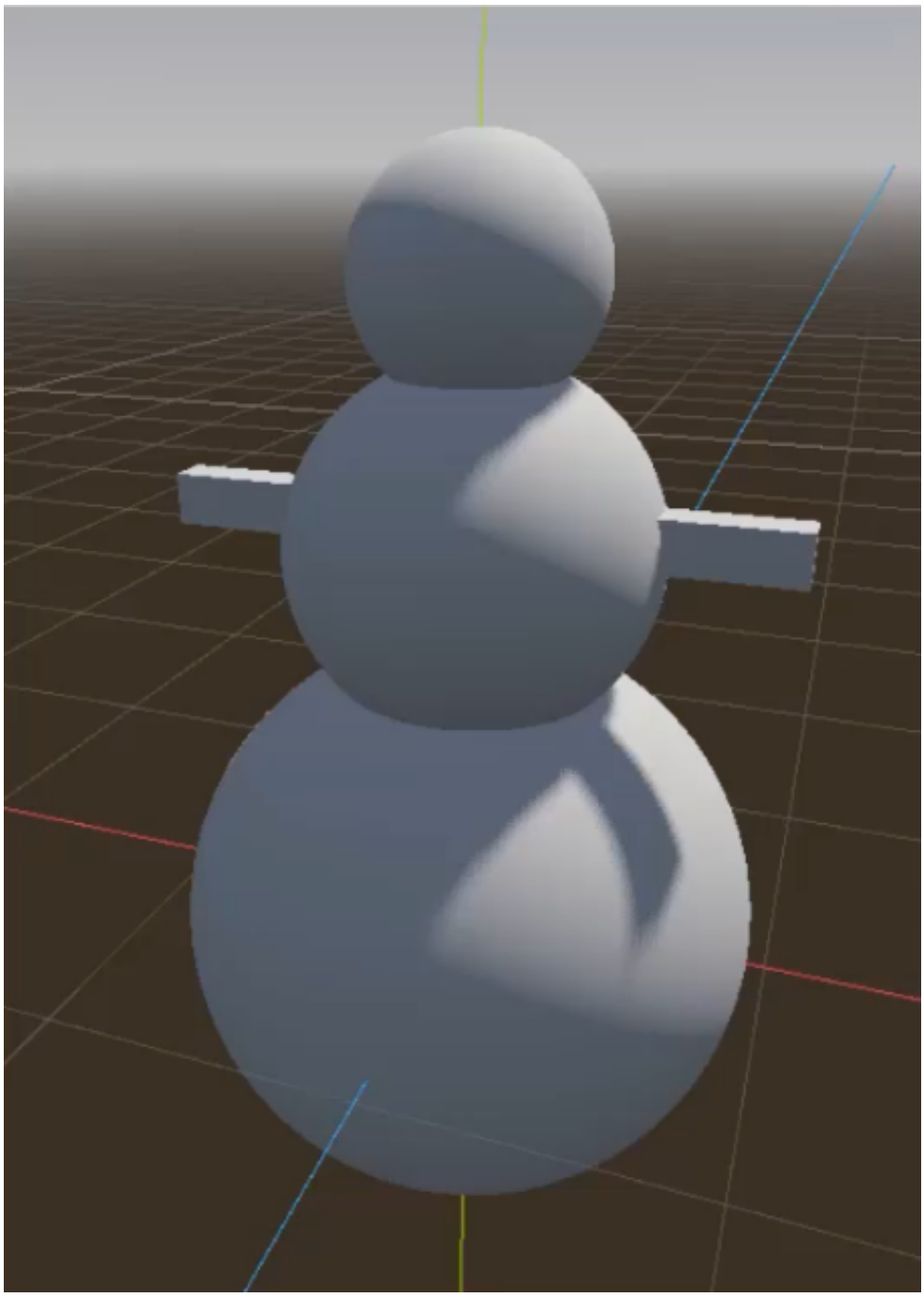


As mentioned, you can also manually enter values in the **Inspector** for precise positioning.

Challenge: Create a Snowman

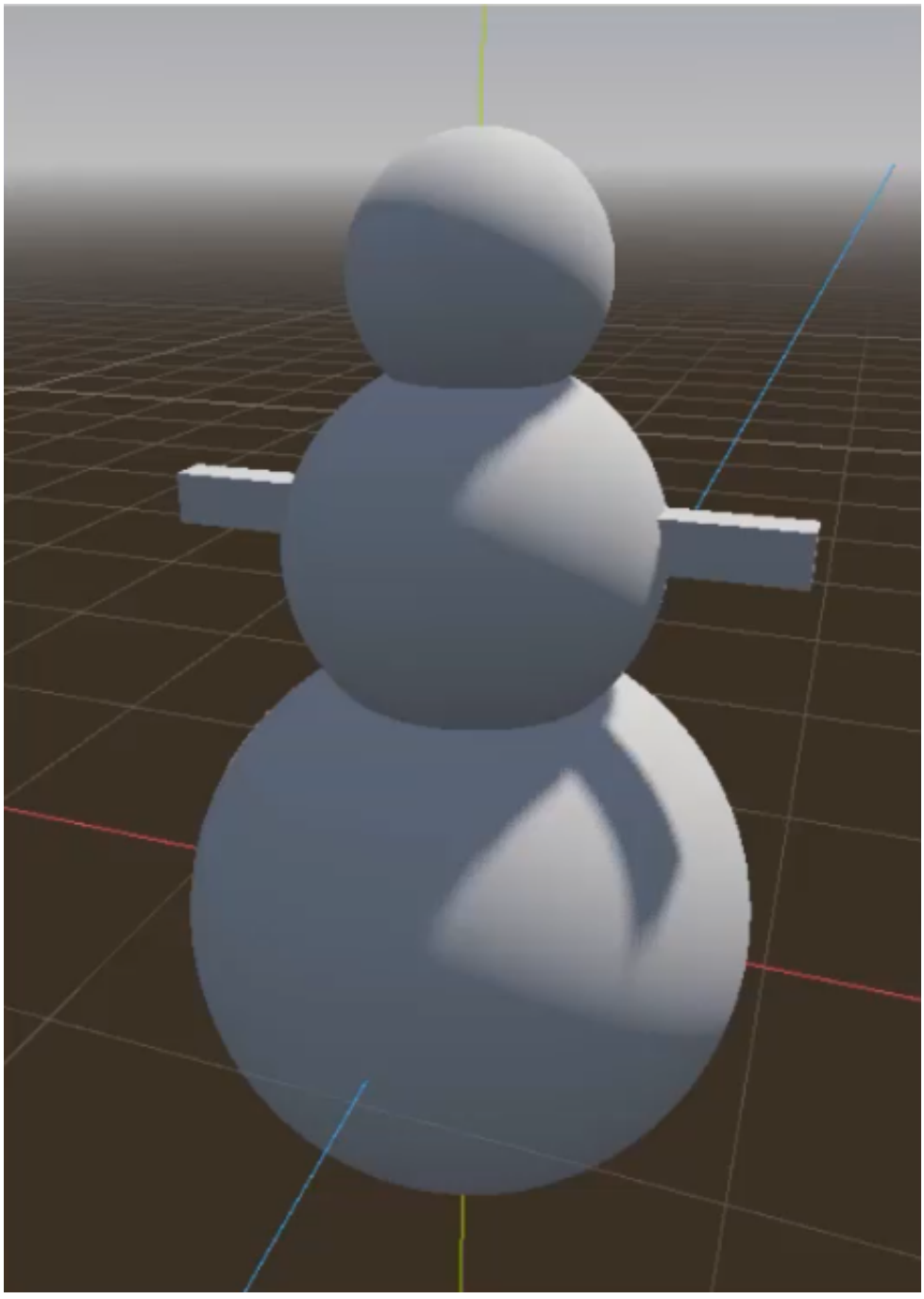
Before the next lesson, try creating a snowman using multiple **MeshInstance3D** nodes. Use the move, rotate, and scale tools to position and size the spheres and box to match the example. Here's a hint:

- Use three spheres for the body and head.
- Use one box for the hat.



Good luck, and we'll see you in the next lesson!

In this article, we will walk through the process of assembling and manipulating basic shapes to form a snowman.

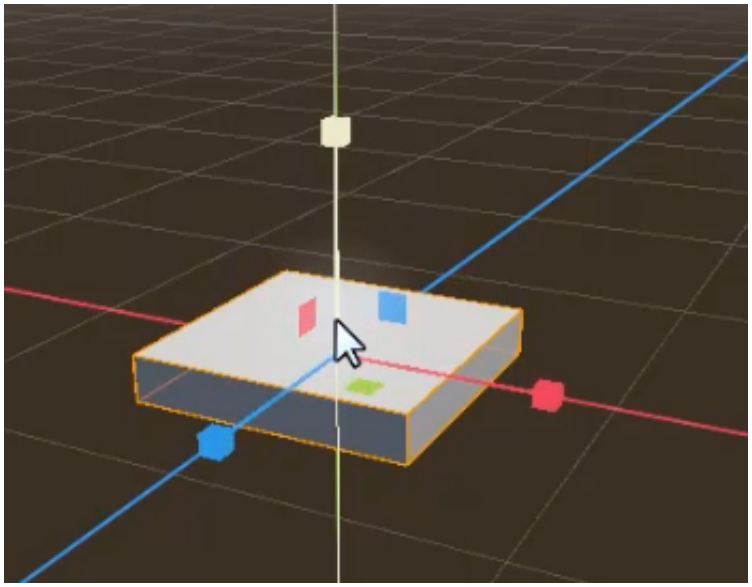


By the end of this tutorial, you will have a solid understanding of how to create and manage 3D objects within Godot.

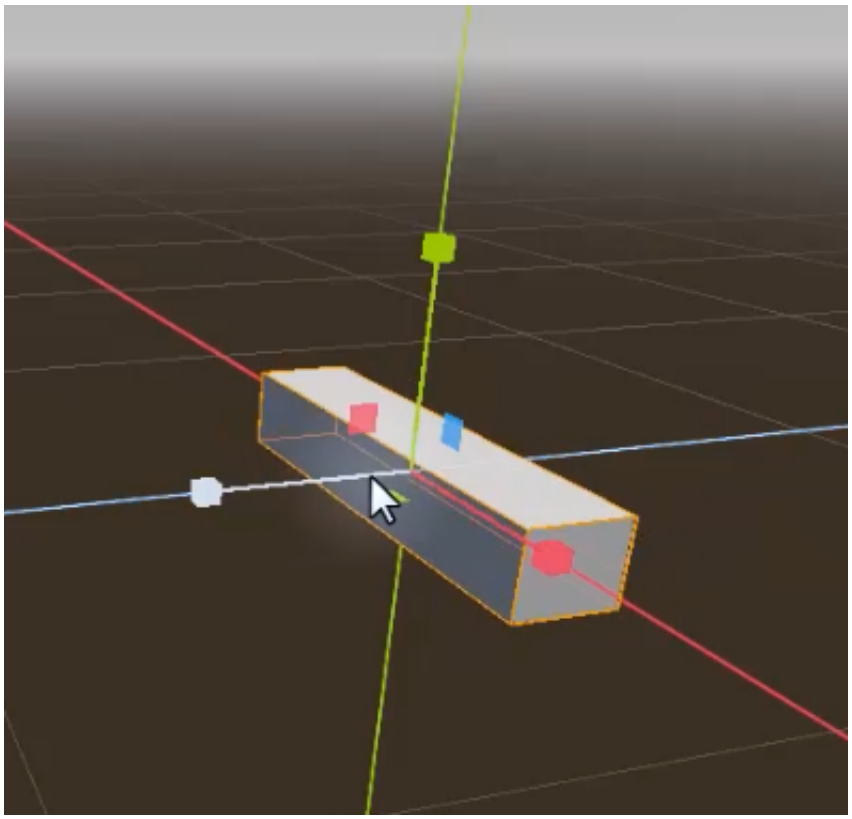
Renaming and Resizing the Arms

Let’s start by focusing on the arms of the snowman. First, select the mesh instance that will represent the arms and rename the node to “Arms” by double-clicking on it.

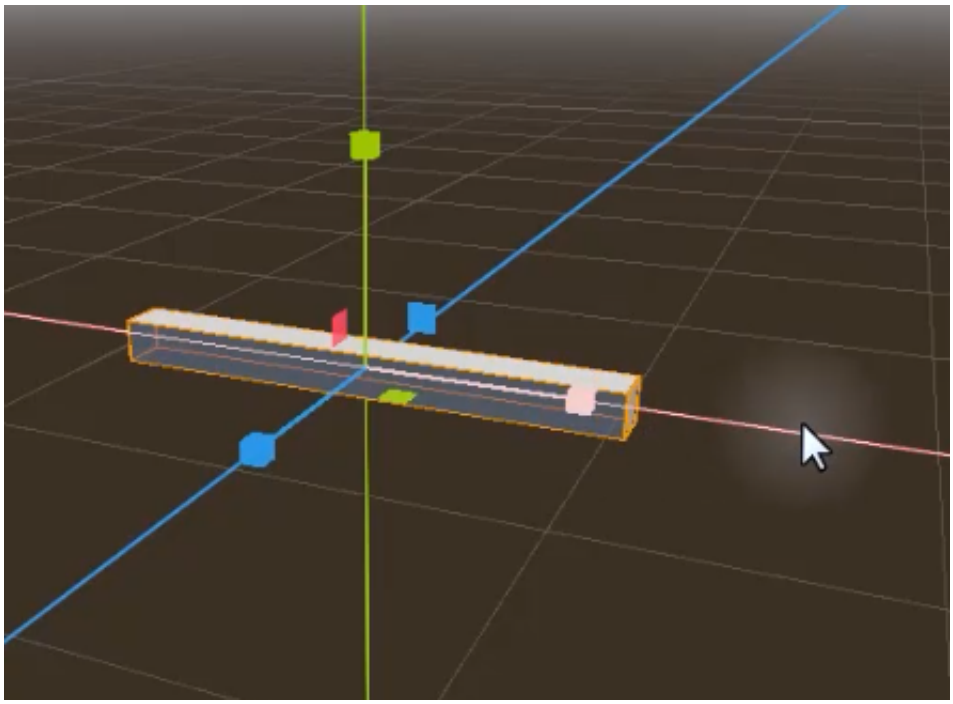
Let’s adjust the size of the arms. Shrink the mesh vertically.



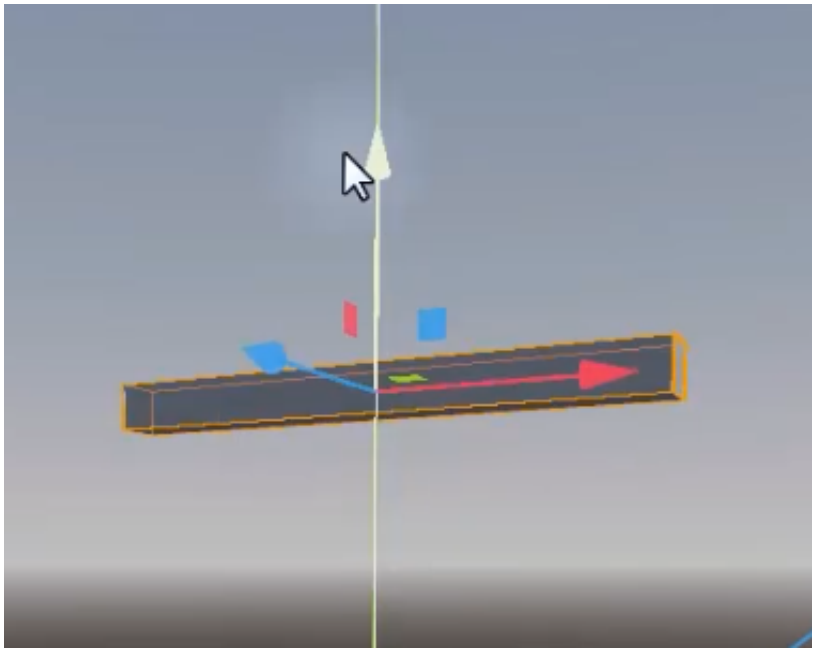
Then, decrease the width.



Extend the length slightly to give it more of an arm shape.

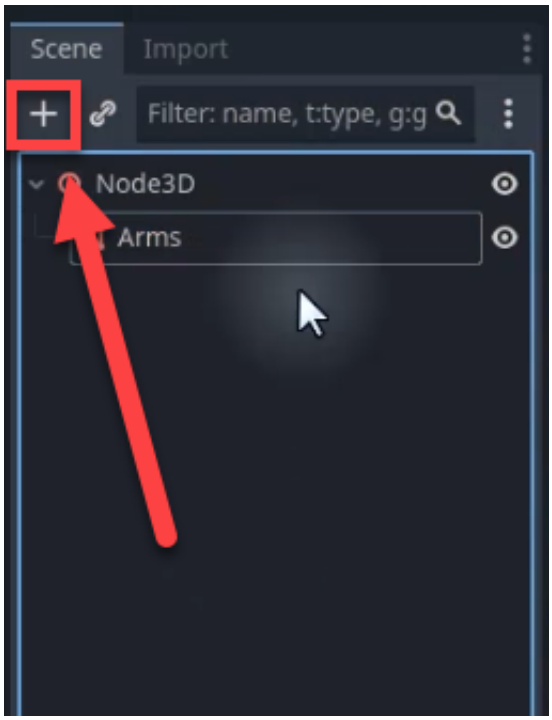


Finally, position the arms appropriately on the snowman.

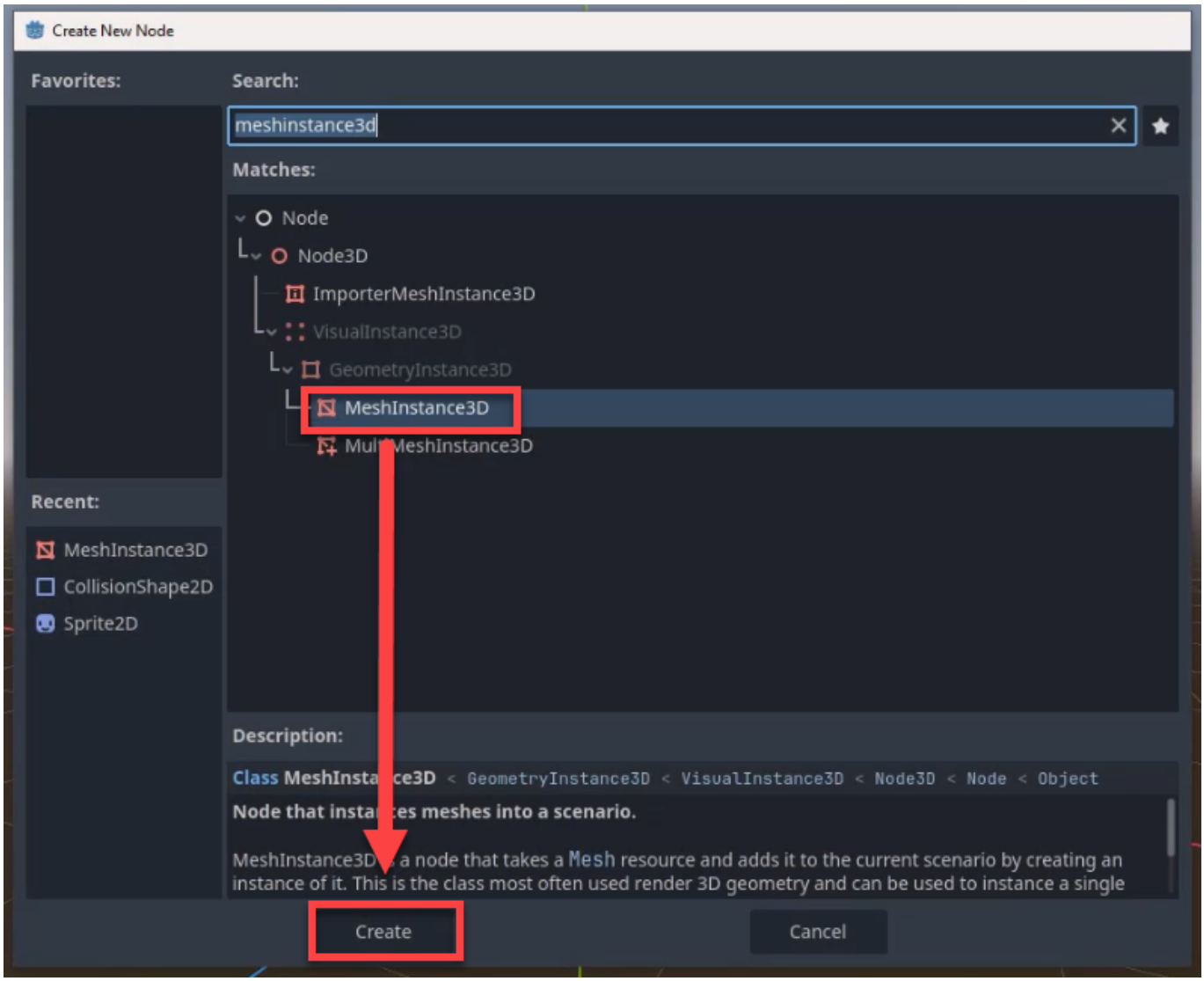


Creating the Legs, Torso, and Head

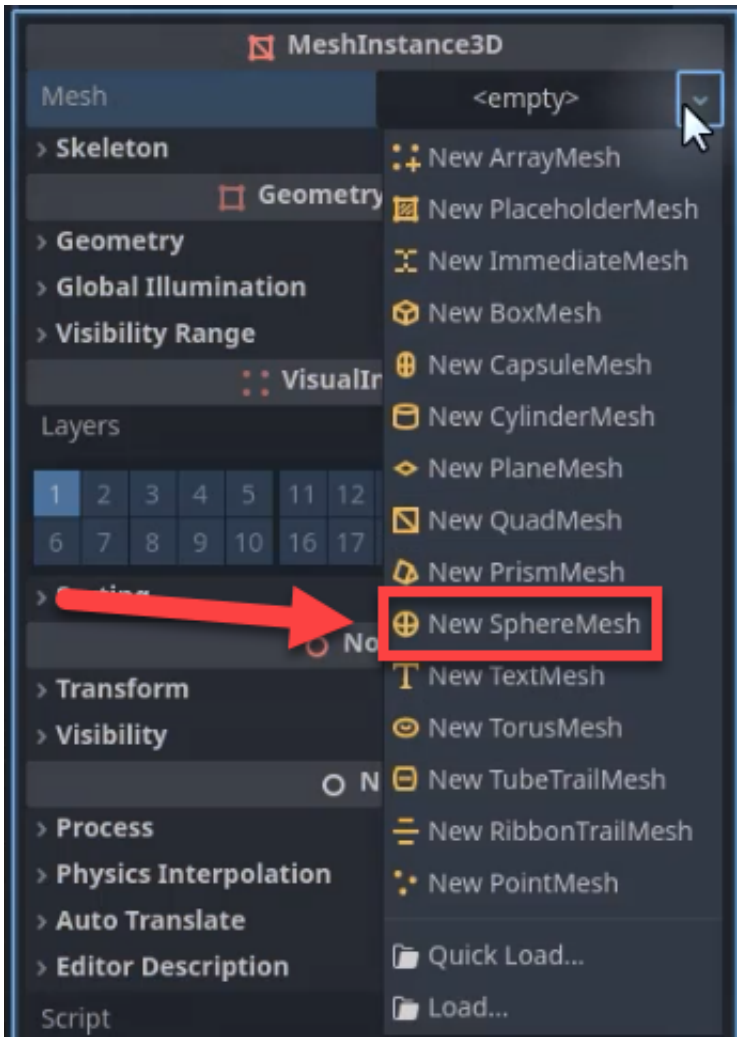
Next, we will create the spheres for the legs, torso, and head. To begin, click the plus sign in the Scene dock to create a new Child node of our root node.



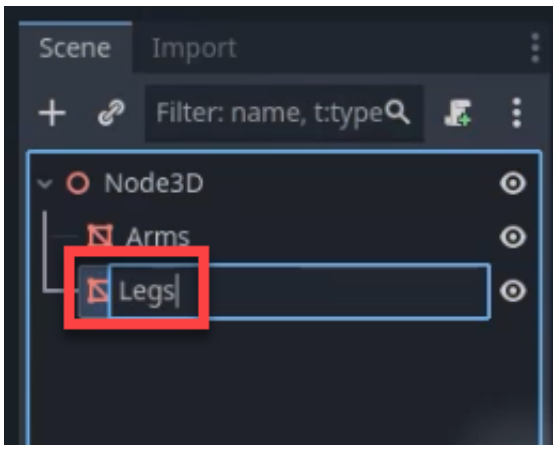
Create a new **MeshInstance3D** node.



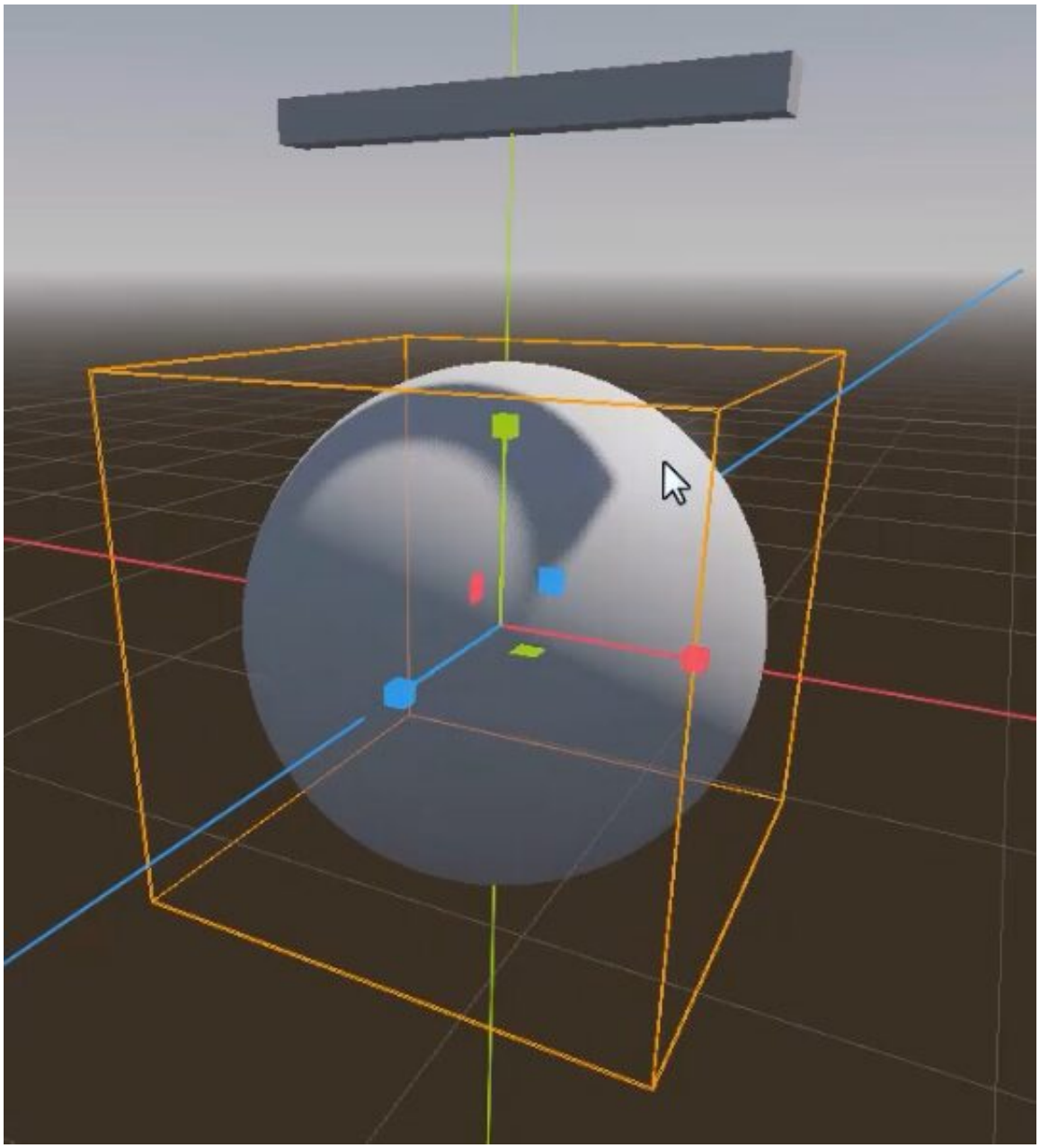
In the Inspector, set the mesh property to a new sphere mesh.



Rename the node to “Legs”.



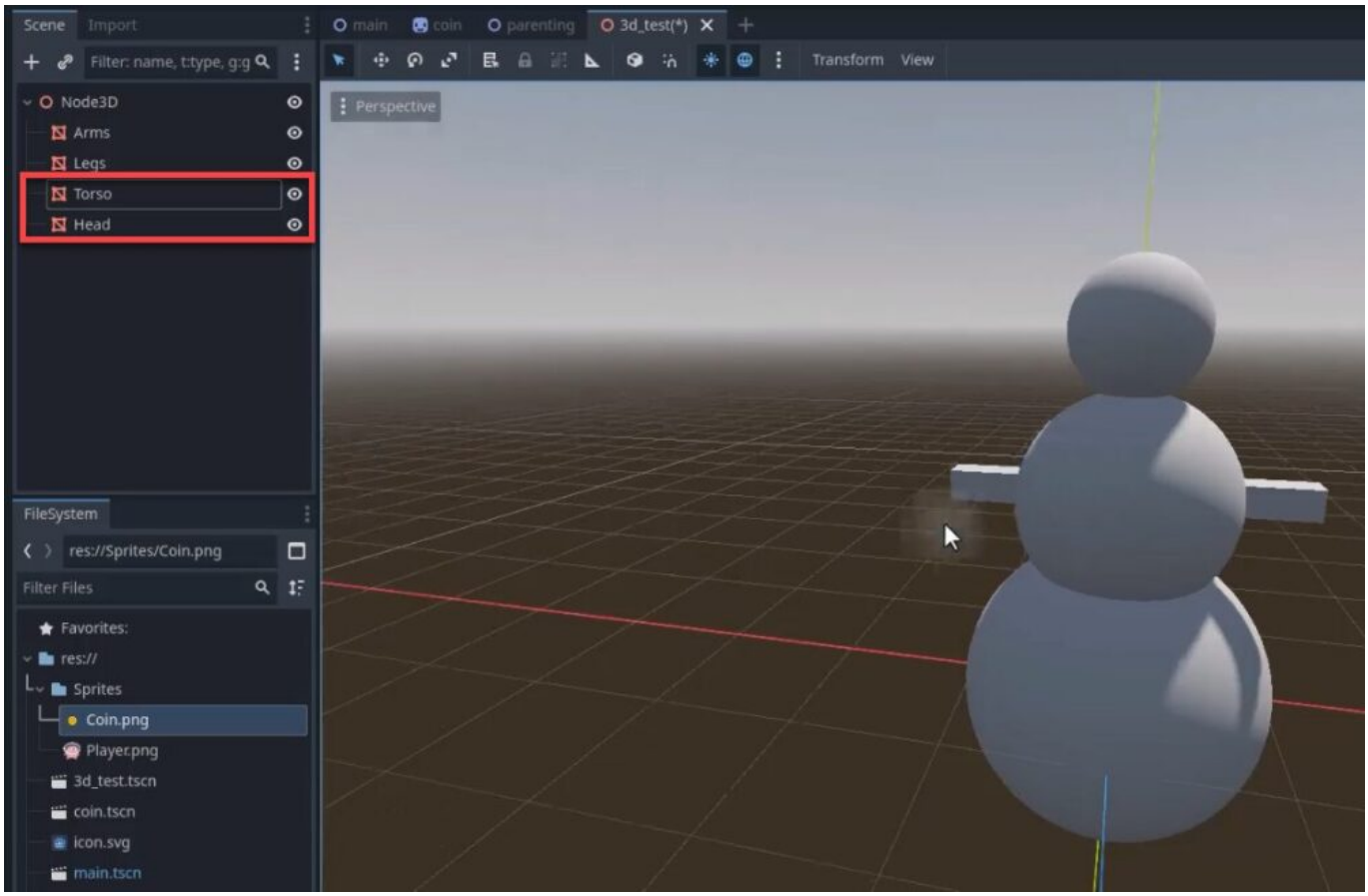
Use the scale tool (hotkey R) to increase the size of the legs.



Repeat this process for the other two spheres!

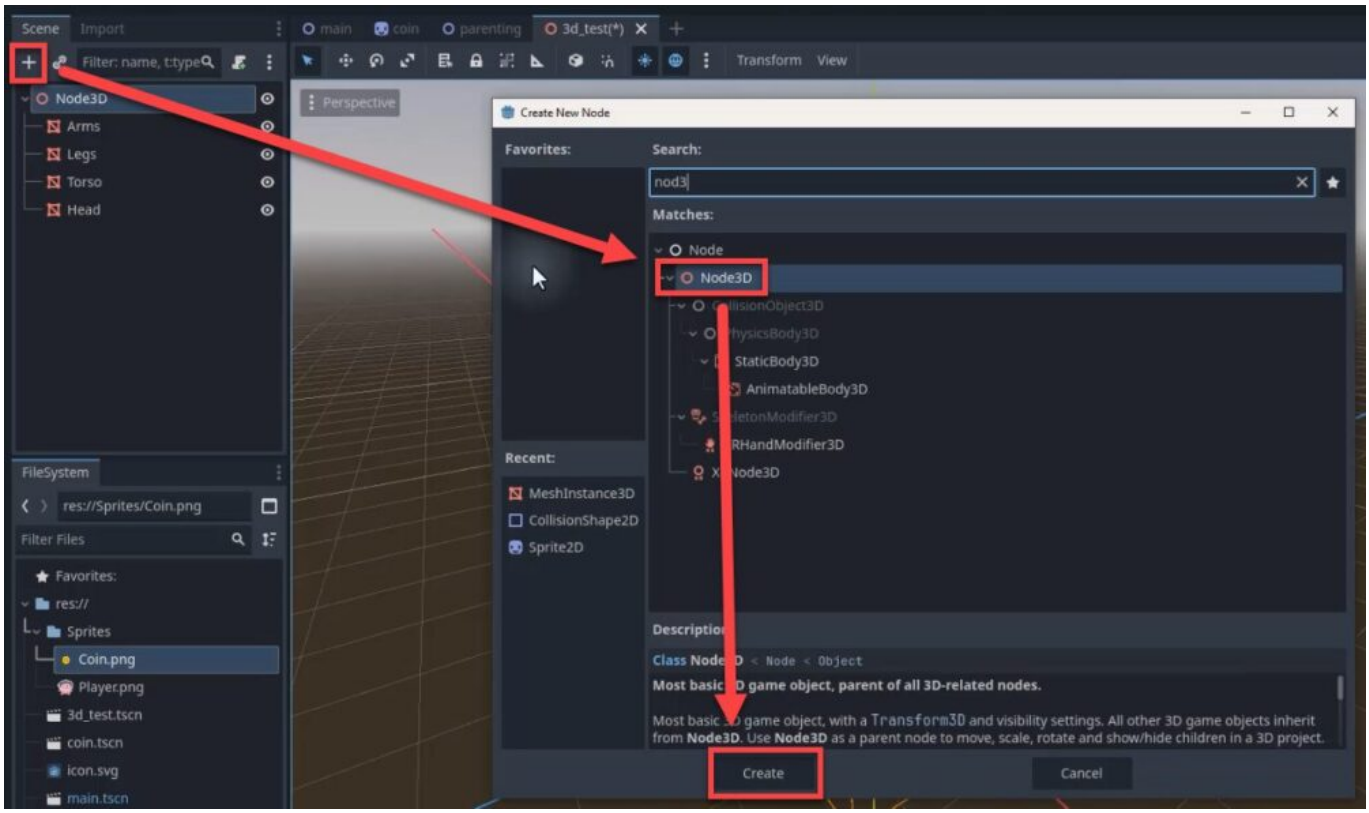
- Duplicate the legs sphere (Ctrl + D), adjust its size, and position for the torso. Then, rename the middle sphere to “Torso”.
- Duplicate the torso sphere, move it up, shrink it down, and rename it to “Head”.

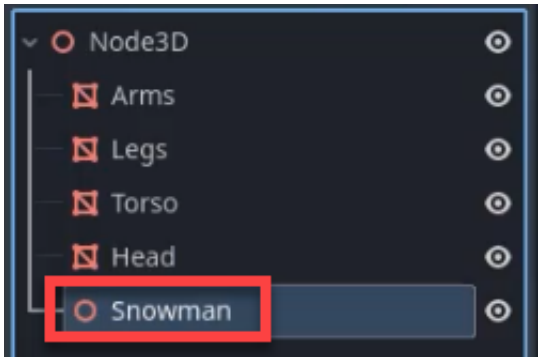
Once everything is created, move everything into place with the arms. Feel free to scale objects further until you get something you like.



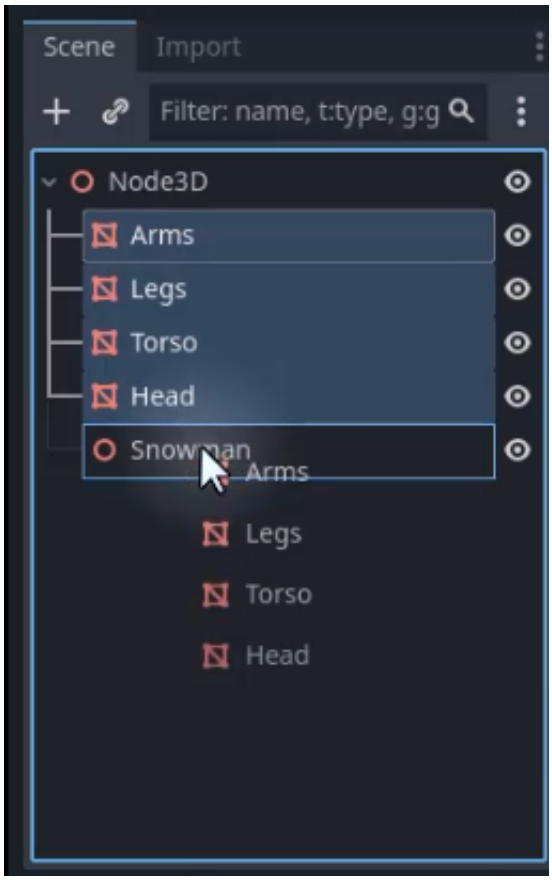
Assembling the Snowman

Now that we have all the parts, let’s assemble them into a single movable object. Create a new **Node3D** and rename it to “Snowman”.

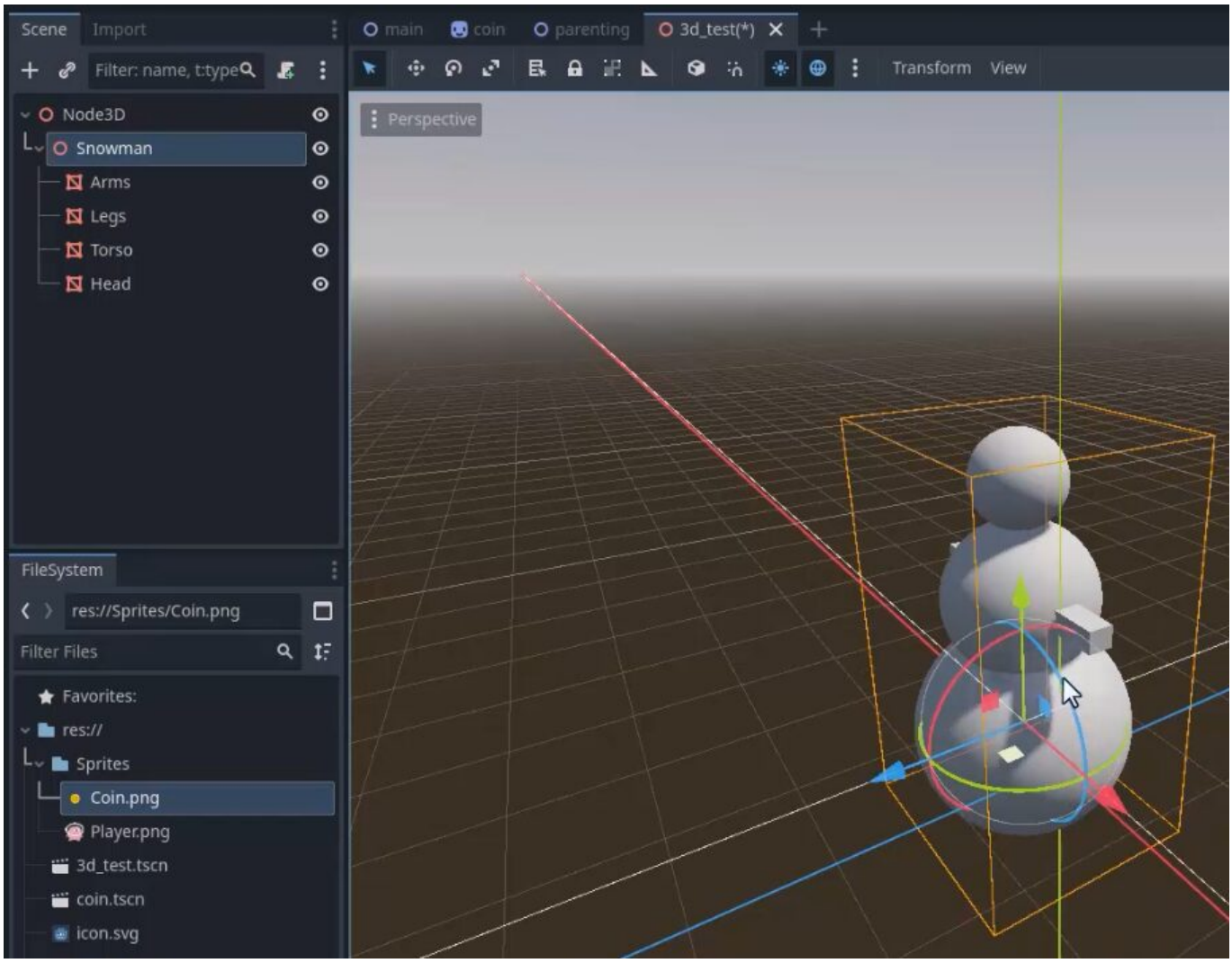




Select the head, torso, legs, and arms nodes and drag them to be children of the “snowman” node.



By selecting the “Snowman” node, you can now move, rotate, and scale the entire snowman as a single entity.



Understanding Node3D

The **Node3D** is a fundamental node in Godot that has basic properties like position, rotation, and scale. It does not render anything on the screen or have specific behaviors, making it ideal for grouping and managing other nodes.

Next Steps

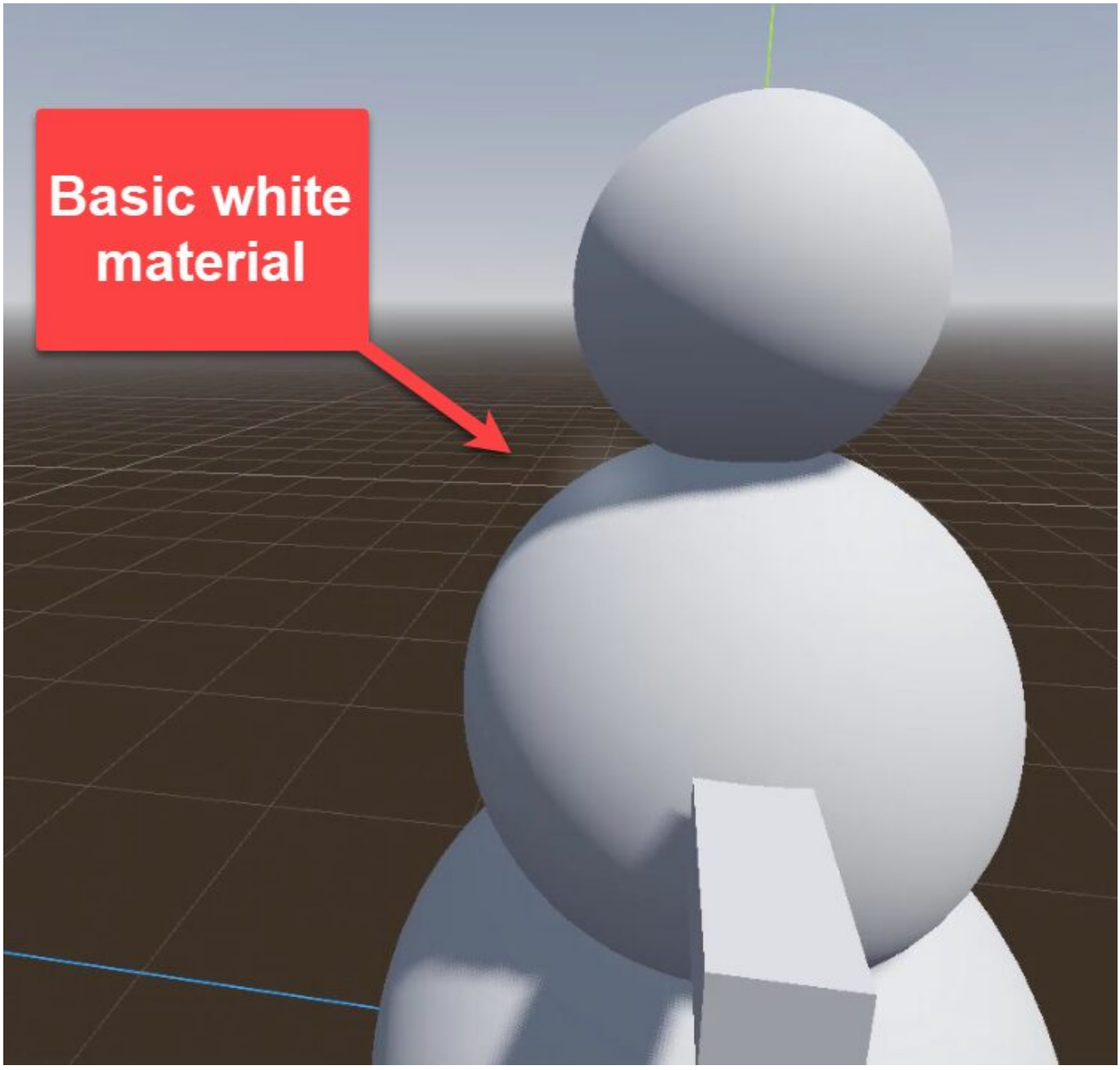
In the next lesson, we will explore the concept of materials, which allow us to change the visual appearance of our 3D models. This will help us move beyond plain white spheres and cubes, adding more depth and realism to our snowman.

Thank you for following along. We look forward to seeing you in the next lesson!

In this tutorial, we will delve into the concept of materials, their significance, and how to create and apply them to 3D models within Godot. By the end of this article, you will have a solid understanding of how to manipulate materials to enhance the visual appeal of your 3D objects.

Understanding Materials

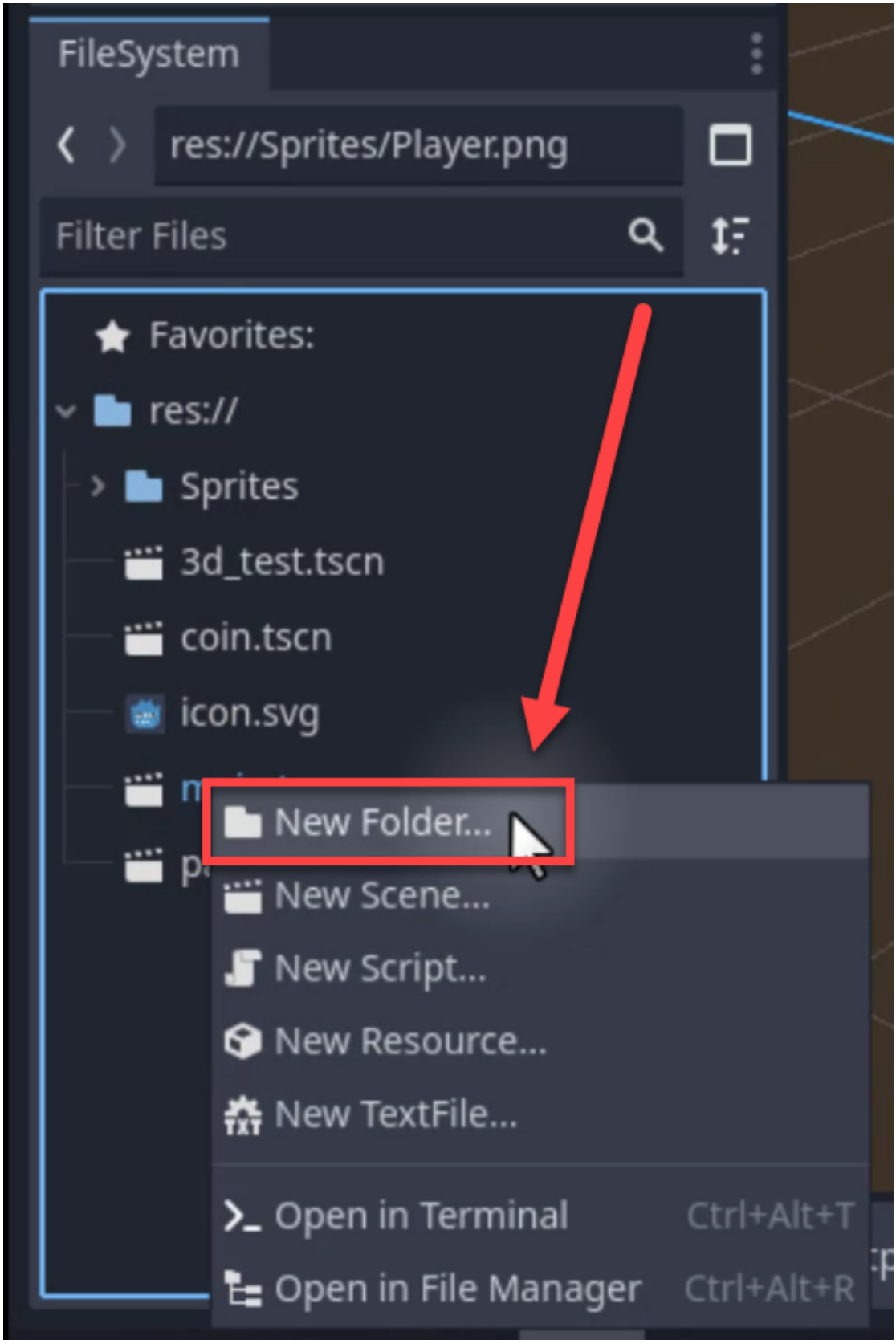
A material in Godot is a resource that defines the visual appearance of a 3D model. By default, 3D models in Godot have a simple white material.

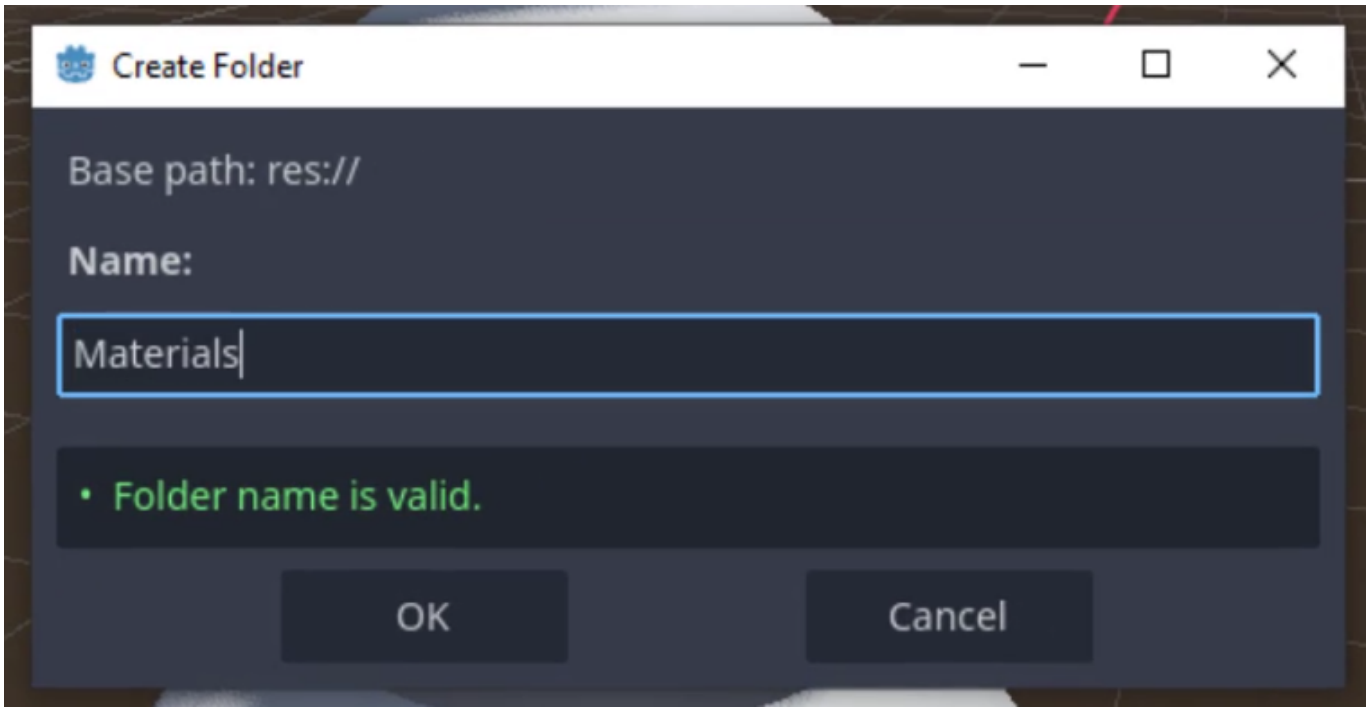


However, materials can be customized to change various visual properties such as color, reflectivity, transparency, and more. This allows for a high degree of creative control over the look and feel of your 3D objects.

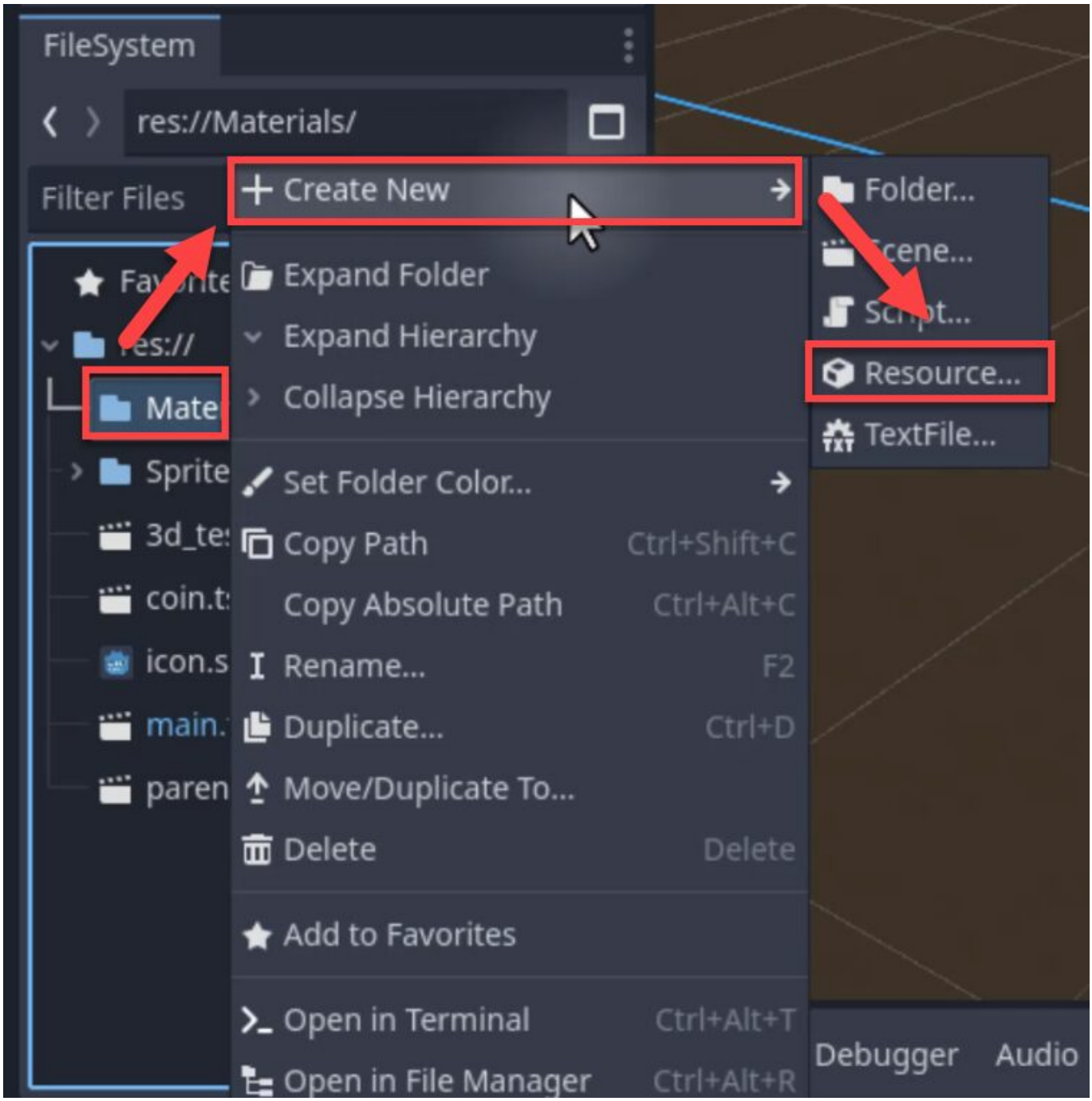
Creating a New Material

Let’s create a new material. First, navigate to your FileSystem within Godot, right-click, and choose to create a new folder. We’ll name this folder “Materials”.

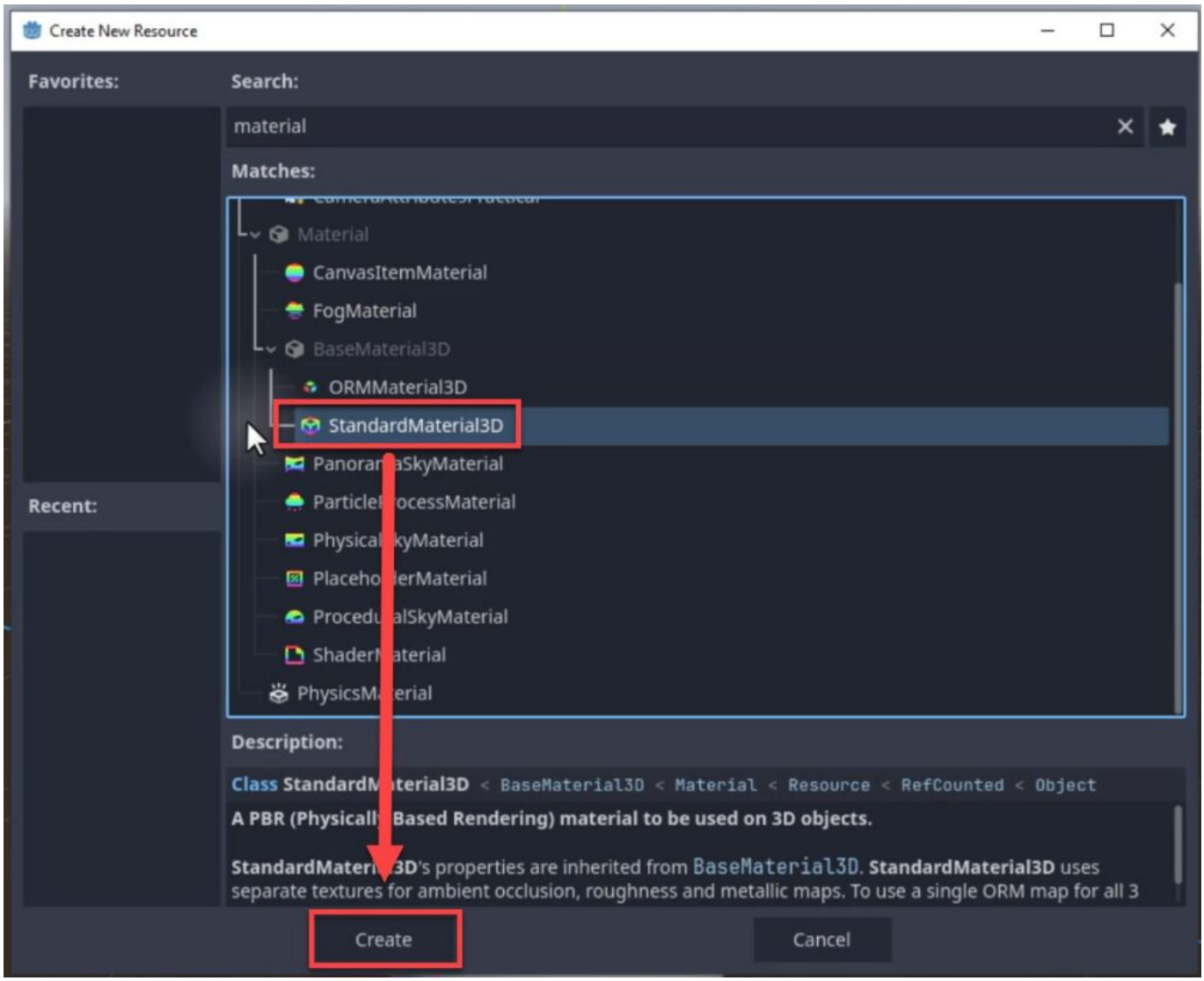




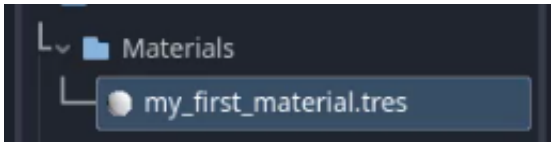
Right-click on the “Materials” folder, select “Create New,” and then choose “Resource.”



Search for “Material” and select “Standard Material 3D.”

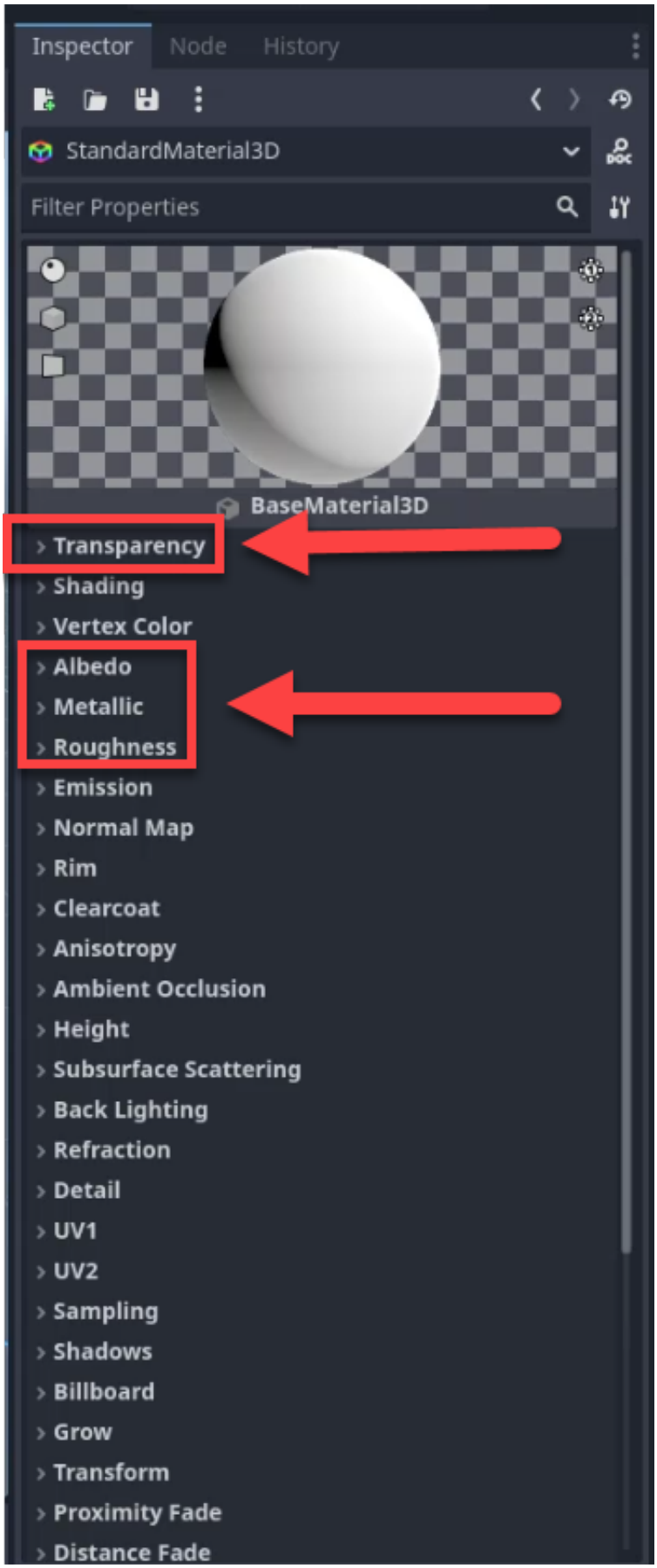


Name your new material (e.g., “my_first_material”).



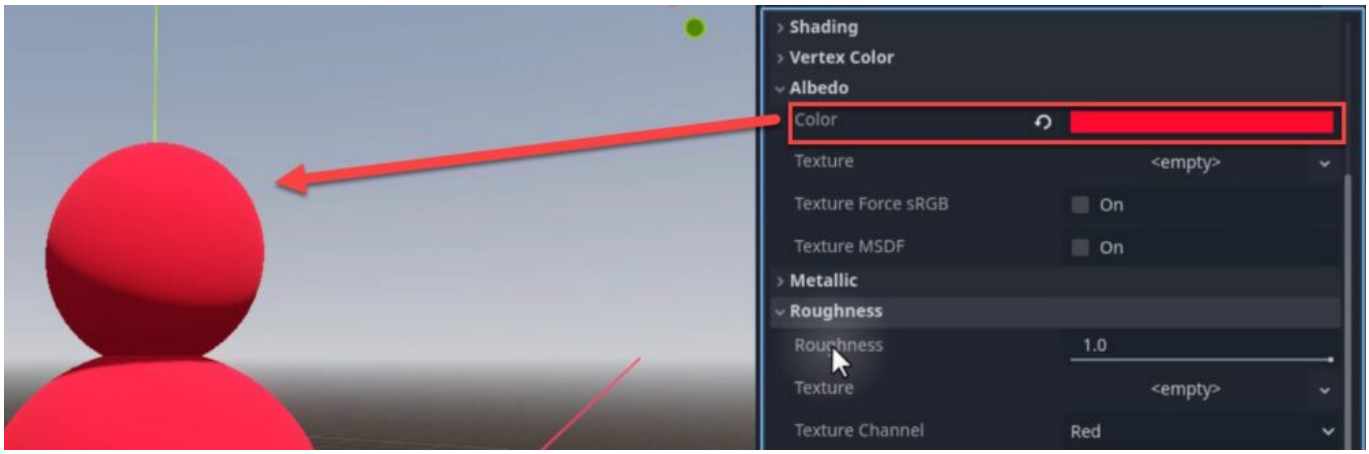
Customizing Material Properties

Once you have created a material, you can customize its properties in the Inspector. Here are some key properties you can modify.



Albedo

This defines the base color of the material. You can use the color picker to change the color.

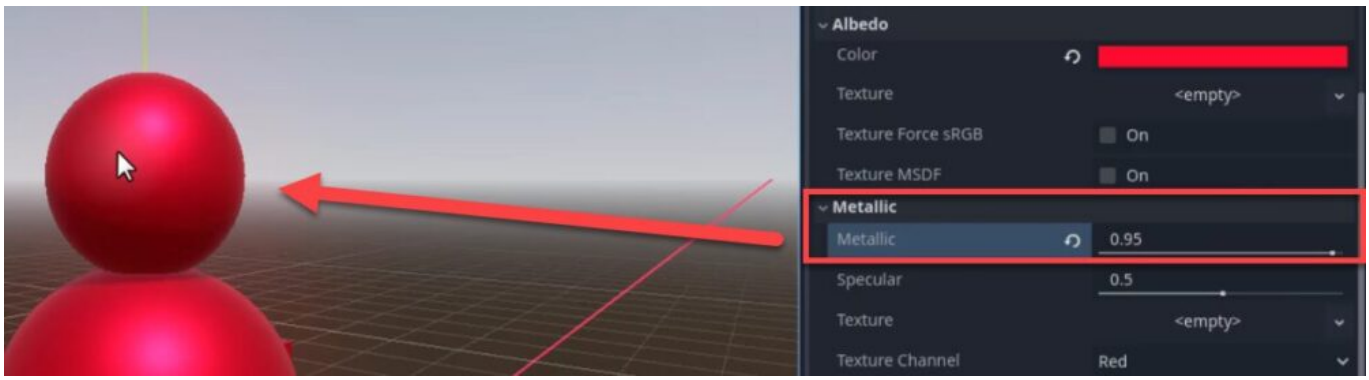


Roughness

This property determines how reflective the material is. A roughness of 1 means the material is not reflective at all (such as above), while a roughness of 0 makes it highly reflective, like glass or a mirror.

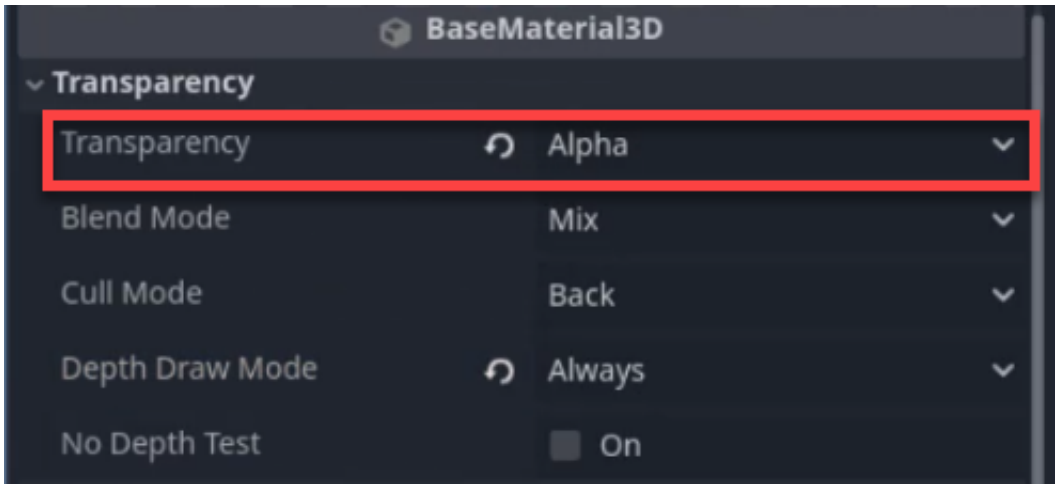
Metallic

This property defines how metallic the material looks. Increasing this value makes the material appear more like metal.

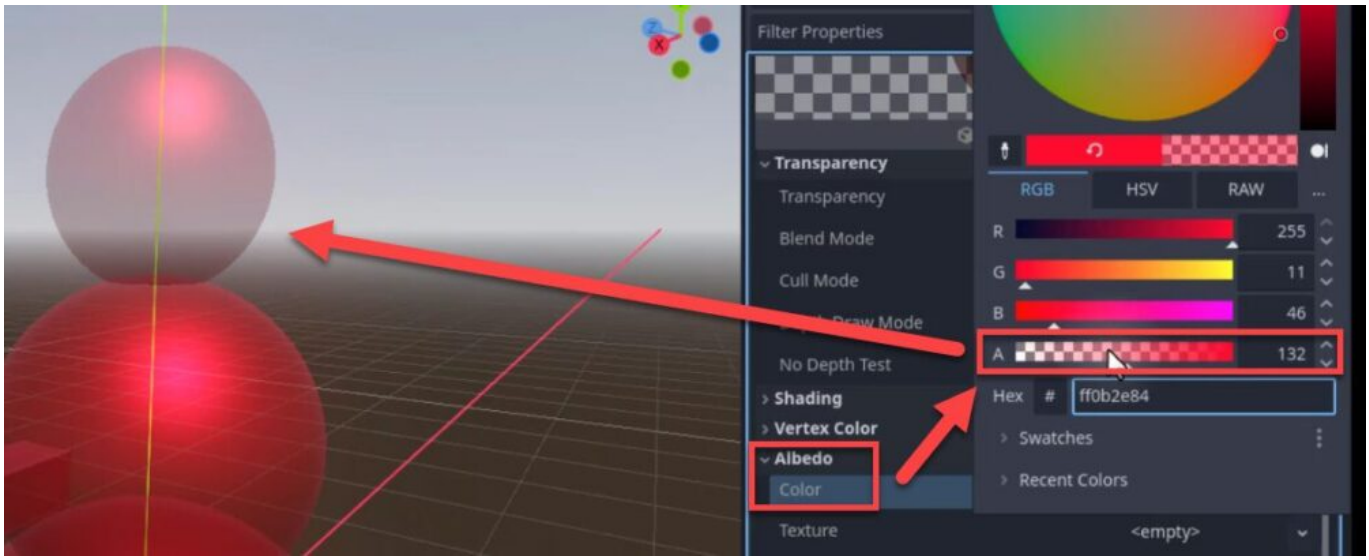


Transparency

To make a material transparent, change the transparency mode from “Disabled” to “Alpha.”



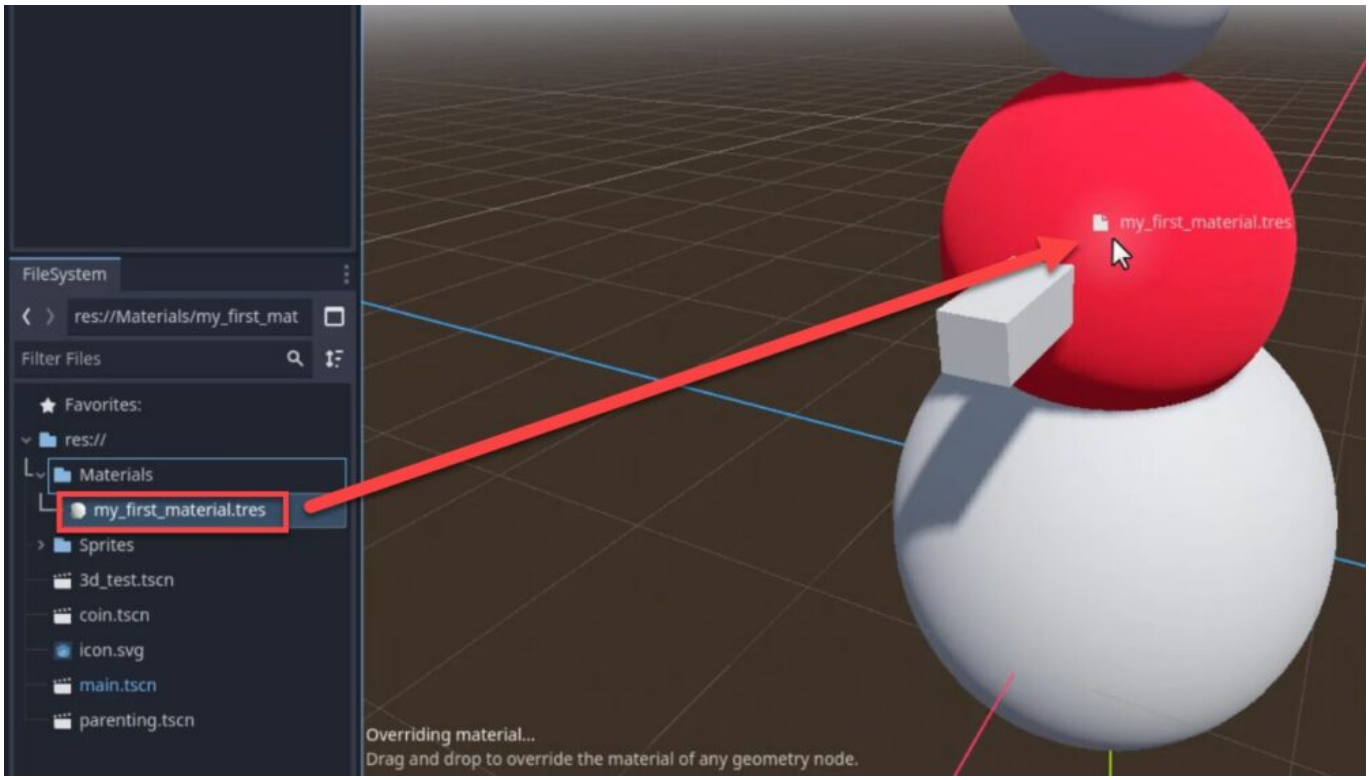
Adjust the alpha value in the color property to control the level of transparency.



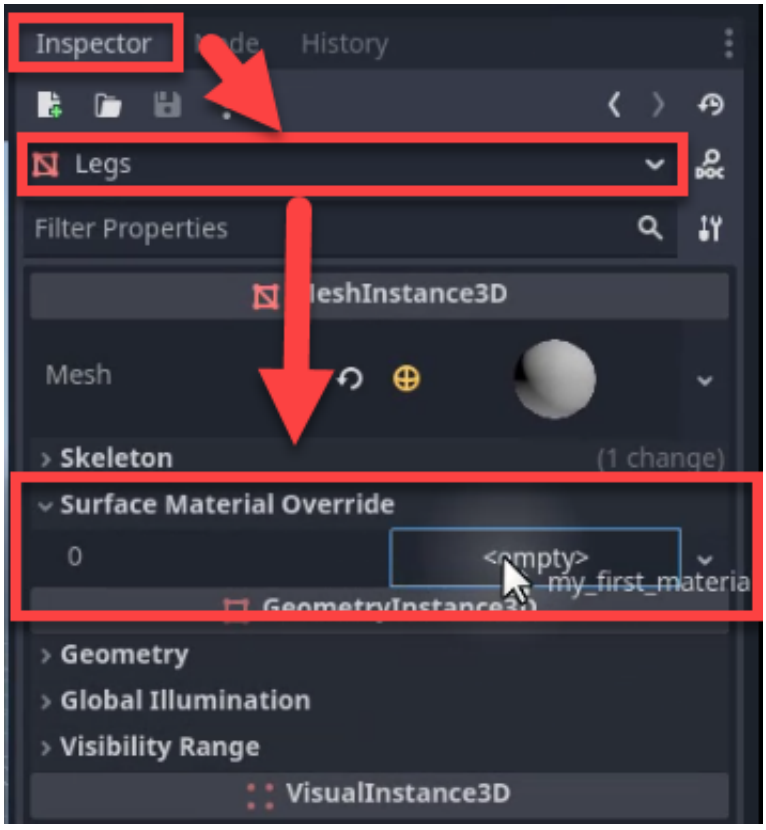
Applying a Material to a 3D Model

To apply a material to a 3D model, you can either:

- Drag the material onto the 3D model directly.

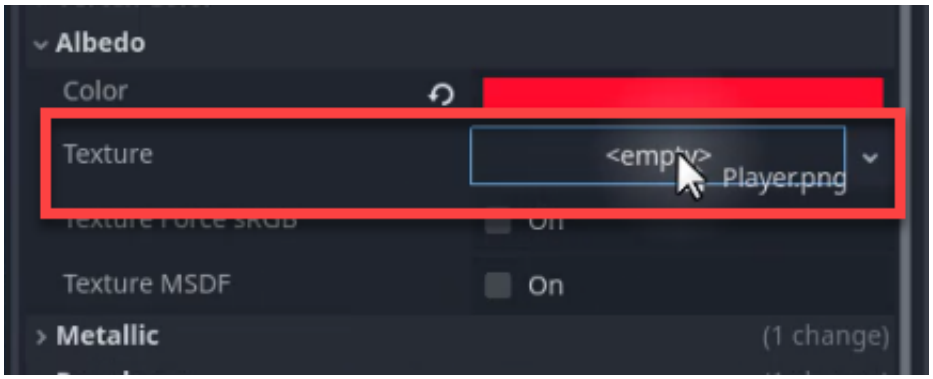


- Select the 3D model, go to the Inspector, find the “Surface Material Override” property, and drag the material into this field.

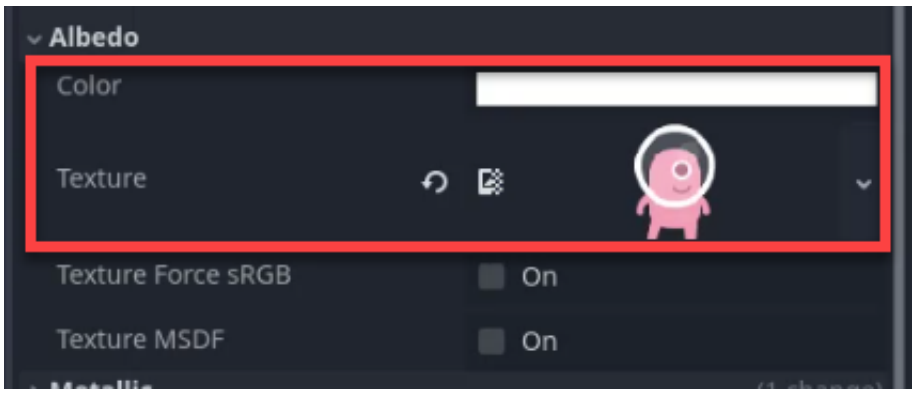


Using Textures

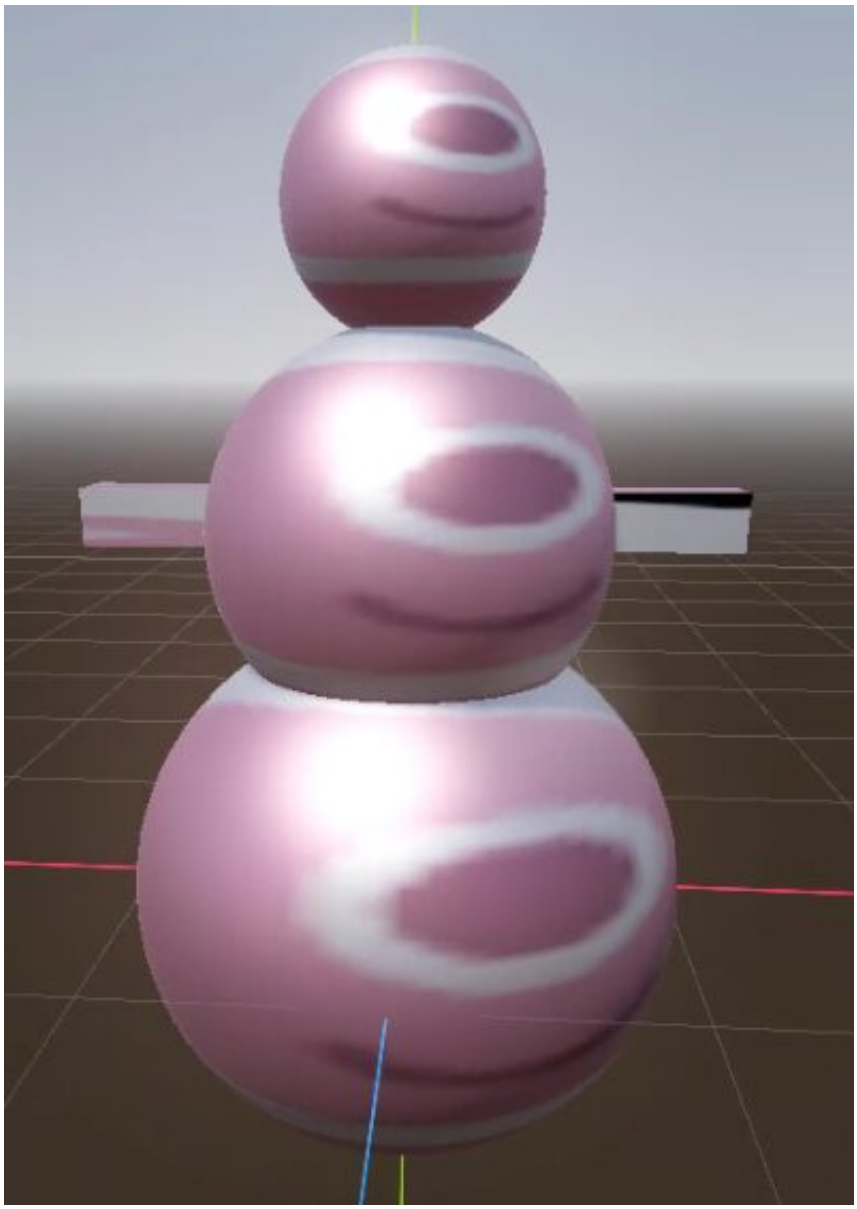
Textures can be applied to materials to add detailed visuals to your 3D models. To apply a texture, go to the Albedo property in the Inspector and find the “Texture” field at the bottom. Drag and drop your desired texture (e.g., a PNG file) into this field.



Ensure the Albedo color is set to white to see the raw texture color.



You can then see the texture applied to any 3D object sporting your material.



Challenge

To reinforce your learning, try the following challenge:

1. Create four new spheres in your Godot project.
2. Apply the following materials to each sphere:
    - A standard purple material.
    - A metallic green material.
    - A transparent or translucent red material.
    - A material with a texture.

This challenge will help you practice creating and applying different types of materials to 3D models. Good luck, and have fun experimenting with materials in Godot!

In this lesson, we'll cover the solution for creating the four materials we challenged you to create last lesson. Those materials were:

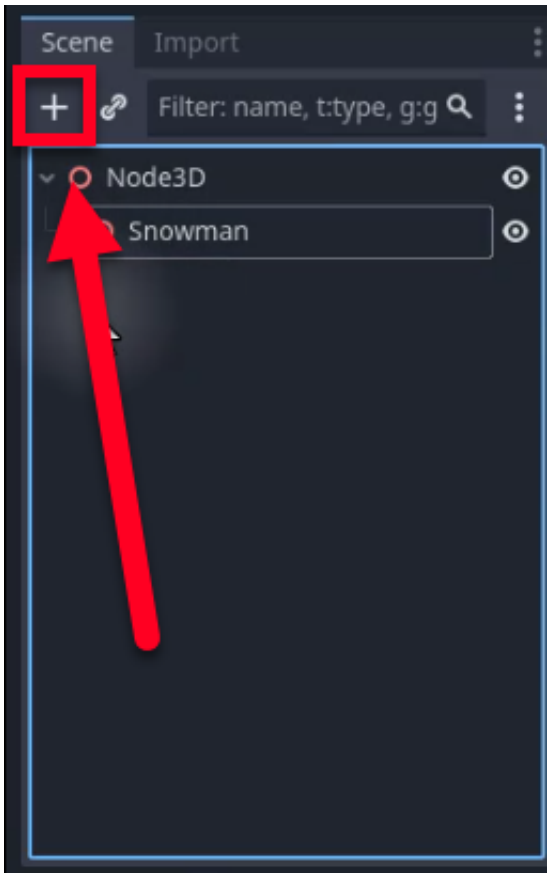
- A standard purple material.
- A metallic green material.
- A transparent or translucent red material.
- A material with a texture.

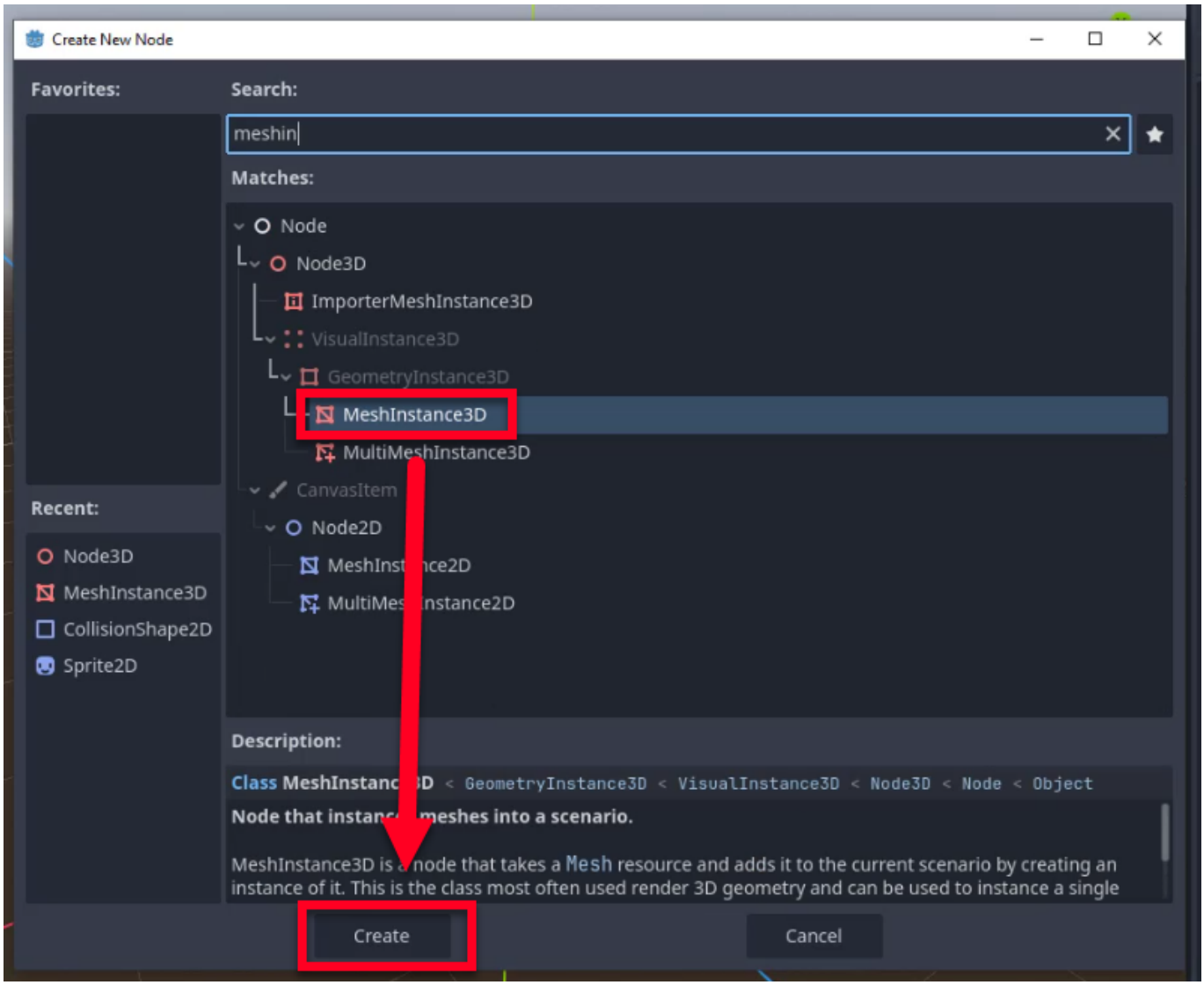
Let's dive in!

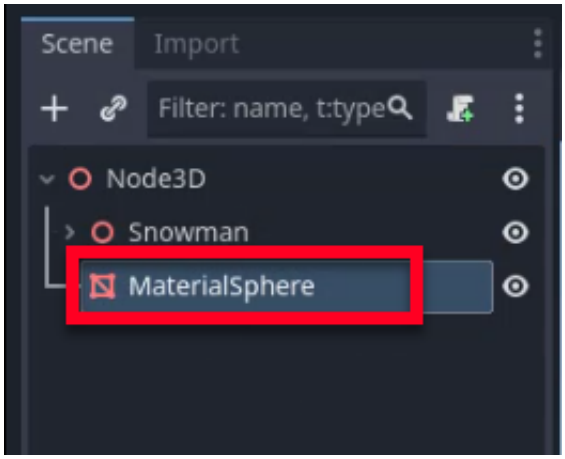
Creating 3D Spheres

Let's start by creating four new spheres in our Godot project to showcase our materials.

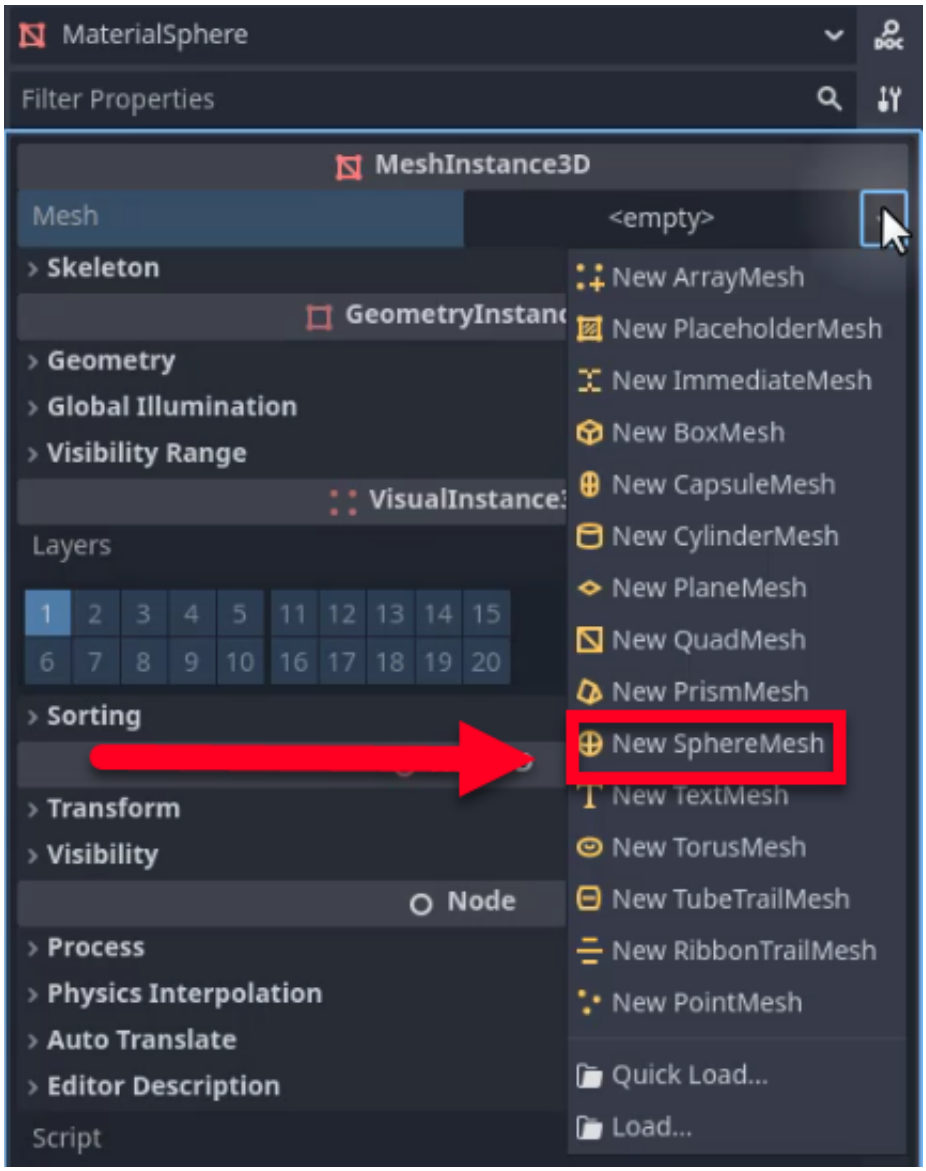
Create a new **MeshInstance3D** node and name it MaterialSphere.



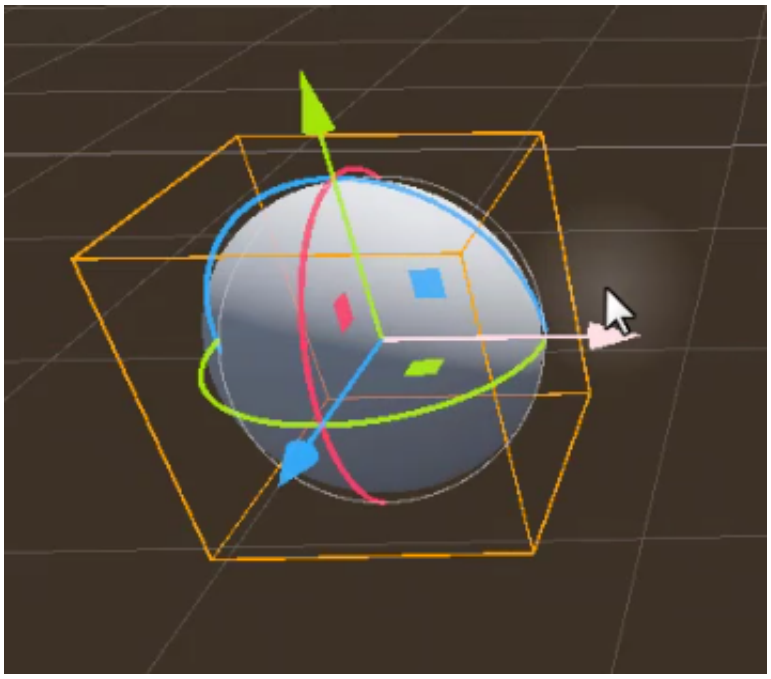




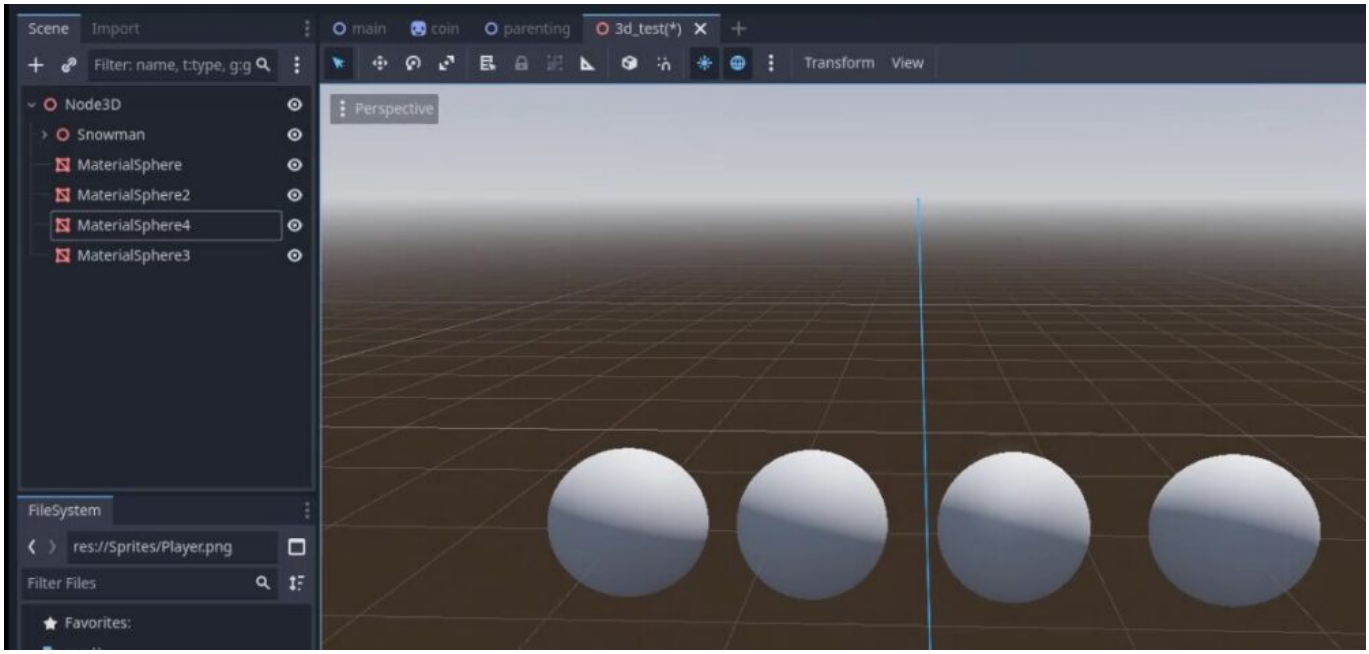
In the Inspector, set the **Mesh** property to a new **SphereMesh**.



Position the sphere in your 3D space. You can hold down the control key to snap the sphere to one-meter increments.

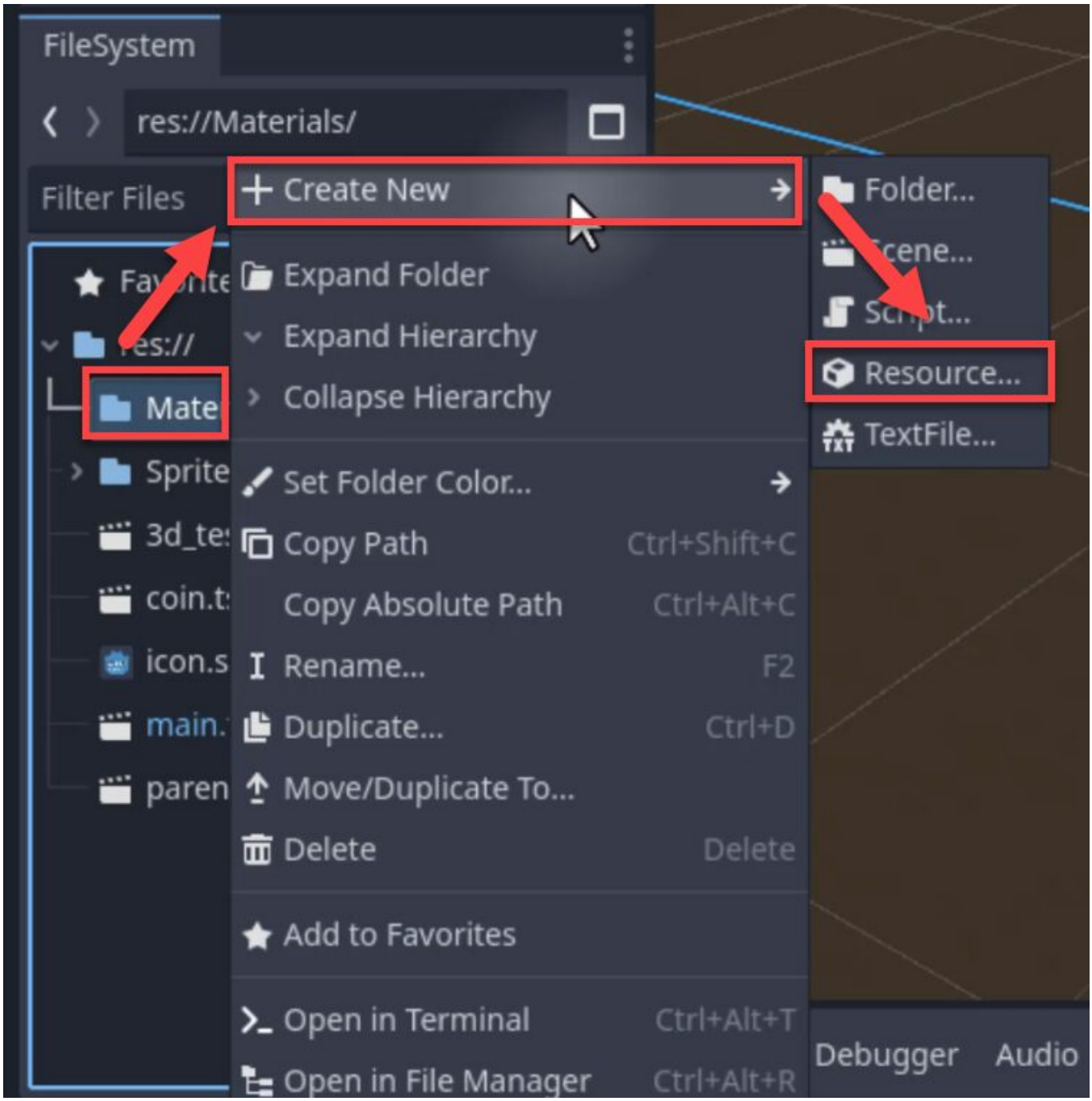


Duplicate the sphere and position the duplicates to have a total of four spheres.

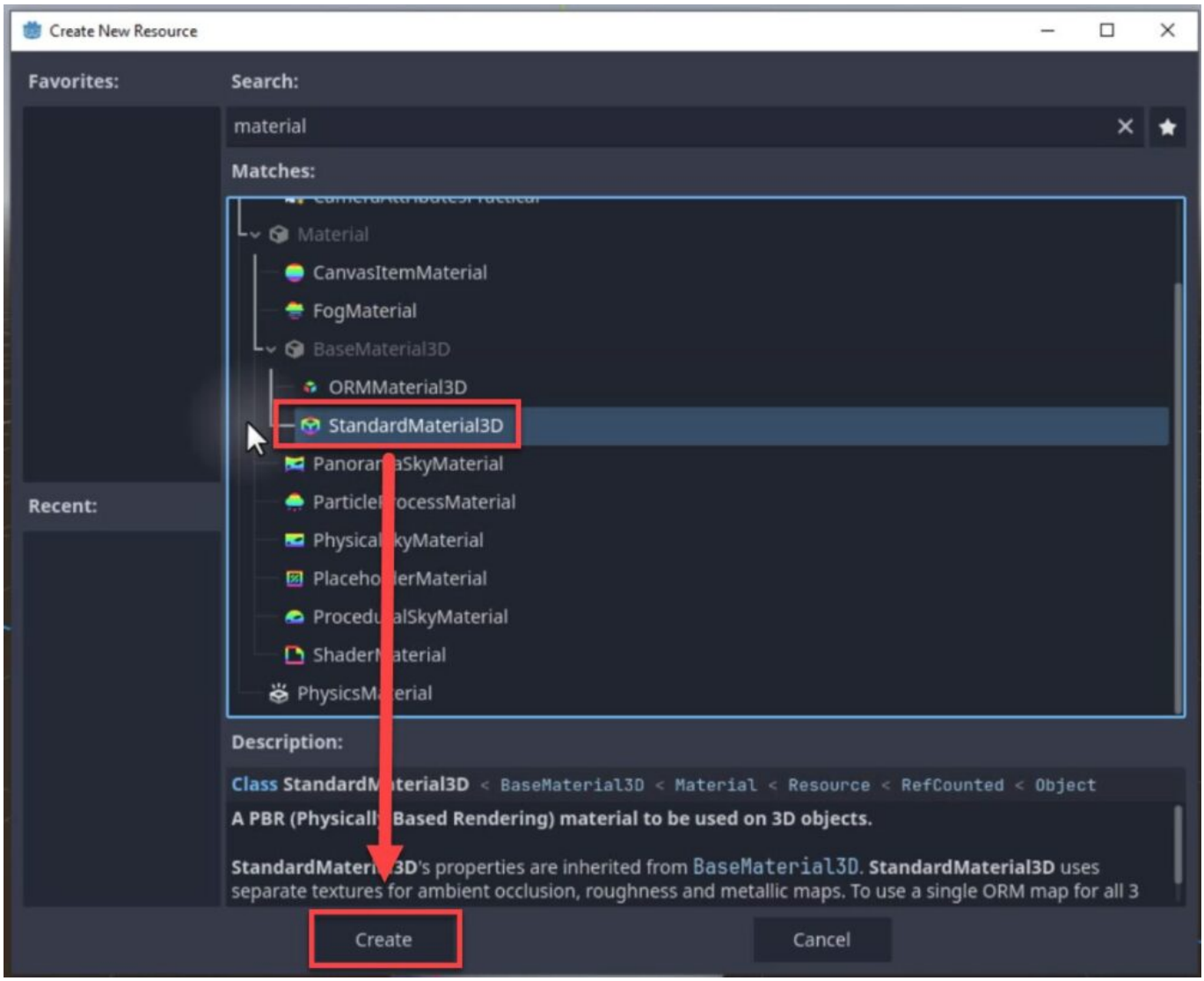


Creating Reusable Materials

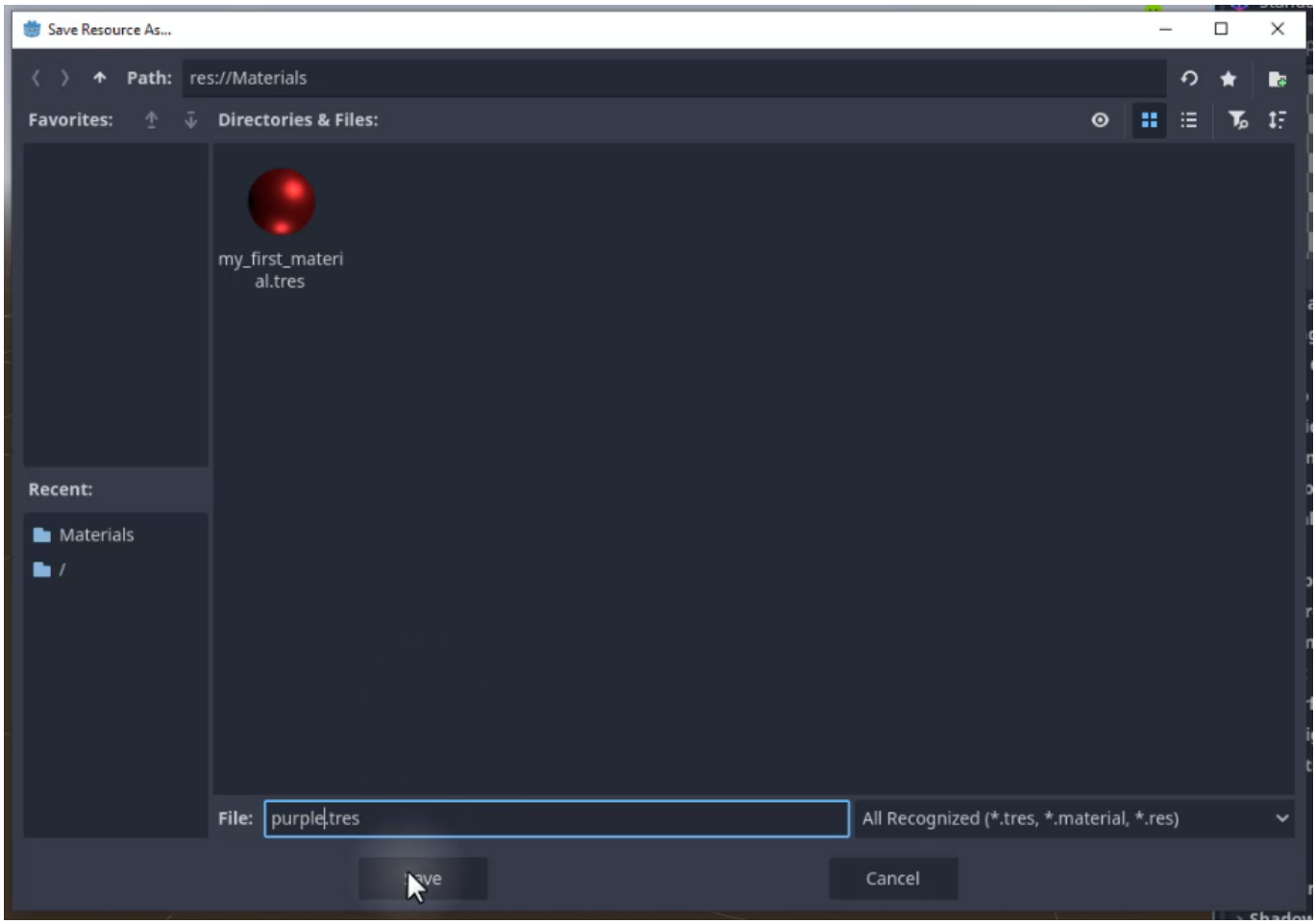
Next, let’s create reusable materials that can be saved to your FileSystem. In your “Materials” folder, right-click and select **Create New > Resource**.



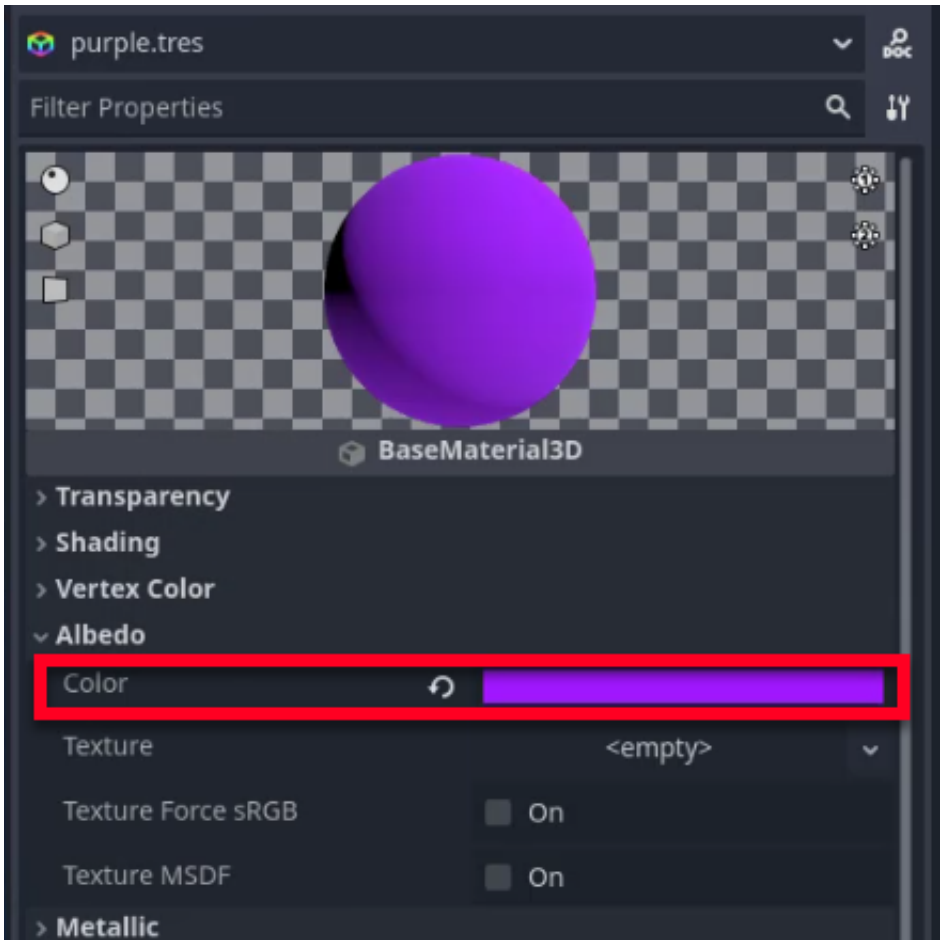
Search for and create a **StandardMaterial3D**.



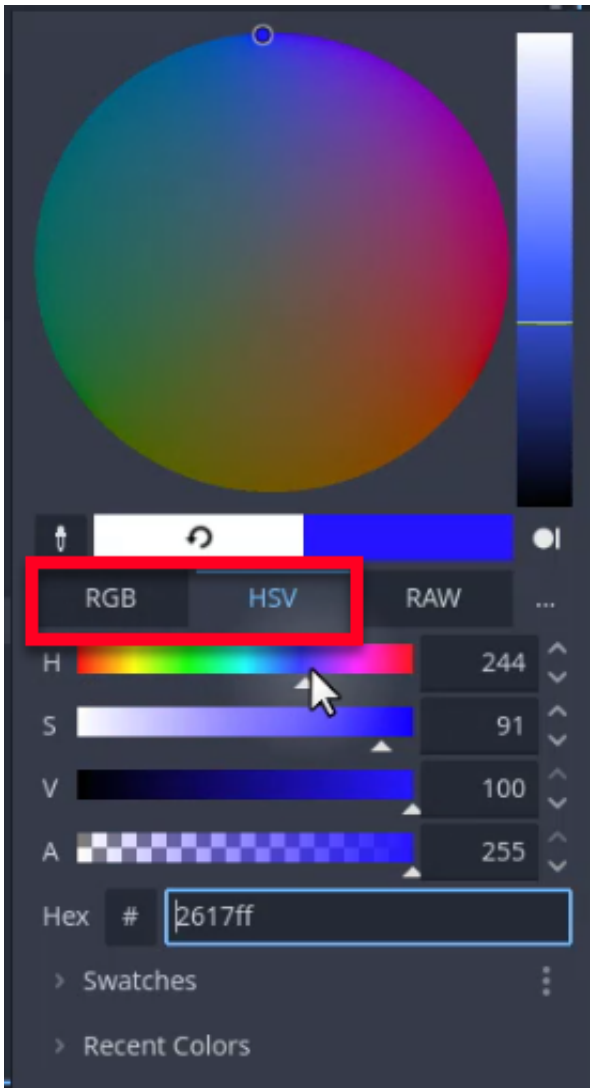
Name the material (e.g., Purple) and save it.



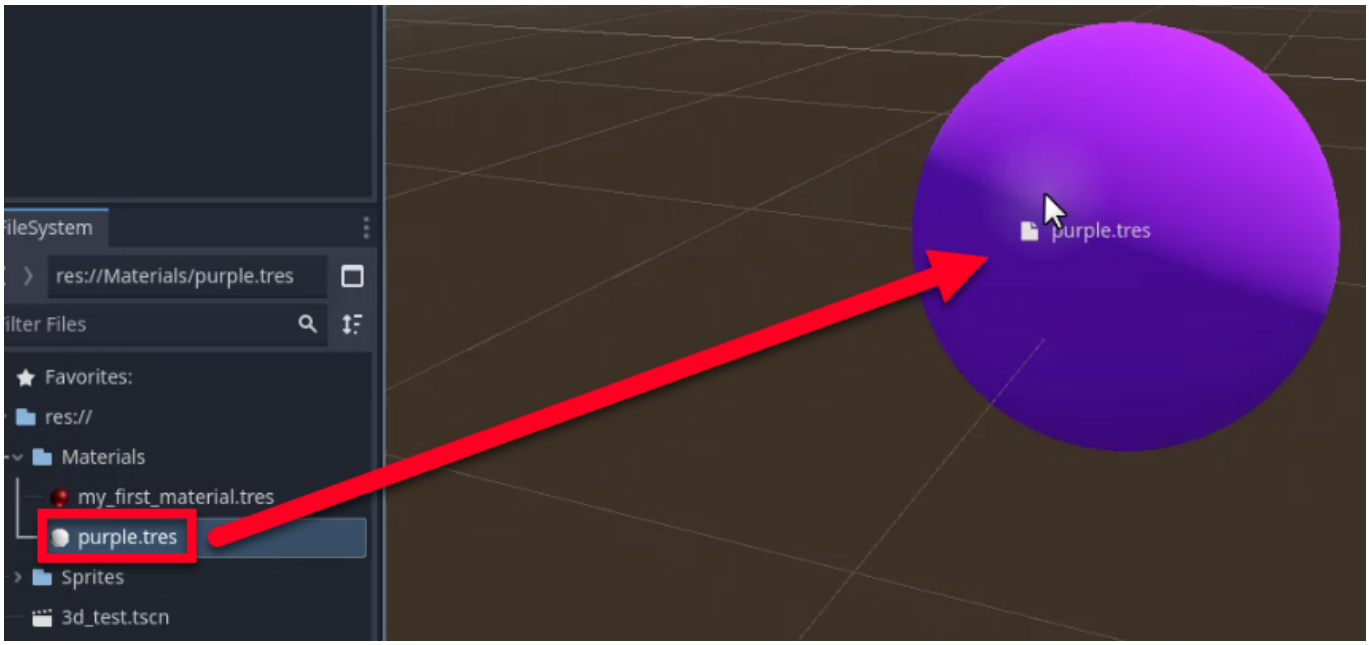
Select the material and modify its **Albedo > Color** property to your desired color.



You can use RGB or HSV values.

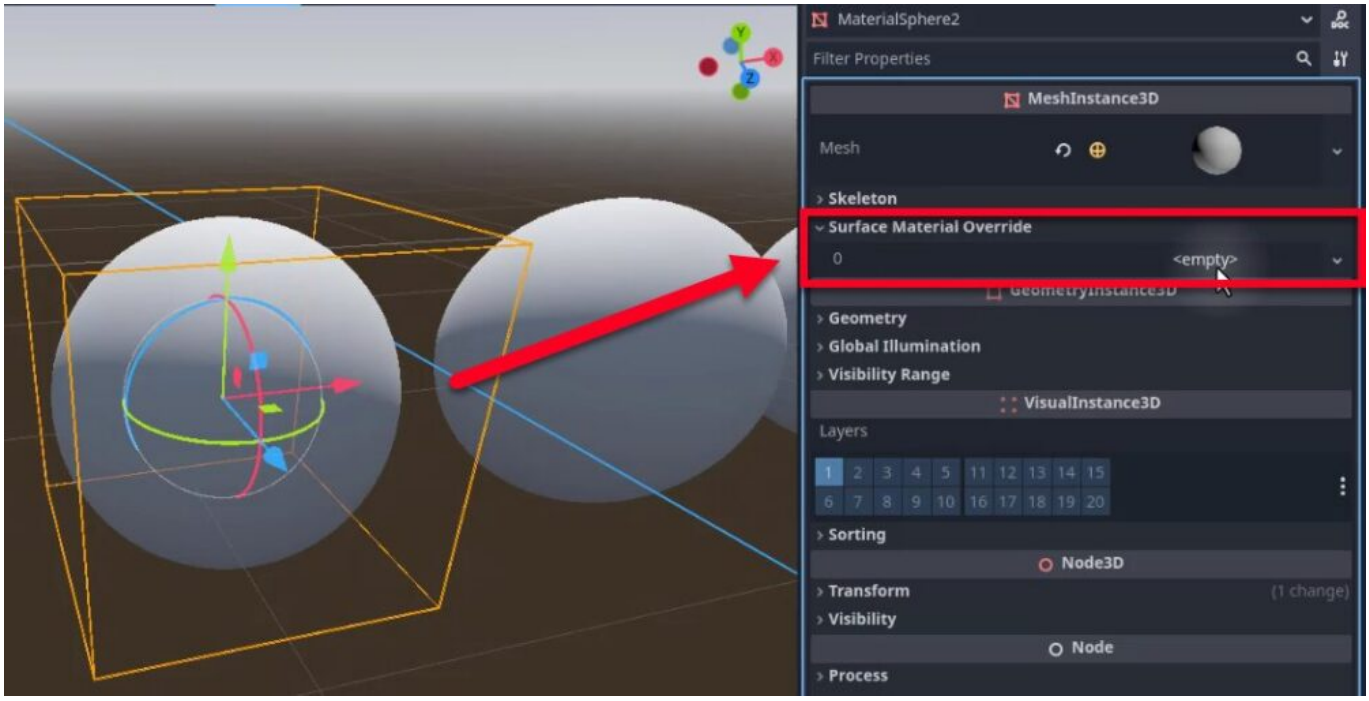


Drag and drop the material onto one of the spheres to apply it.

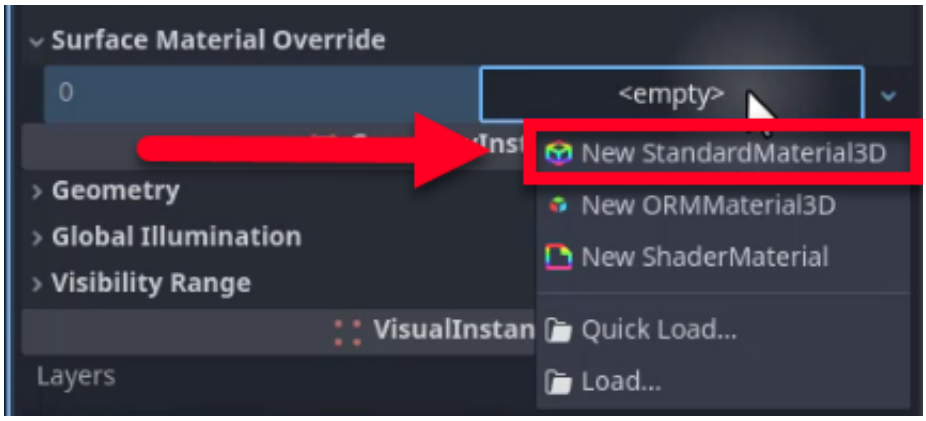


Creating Sub-Resources

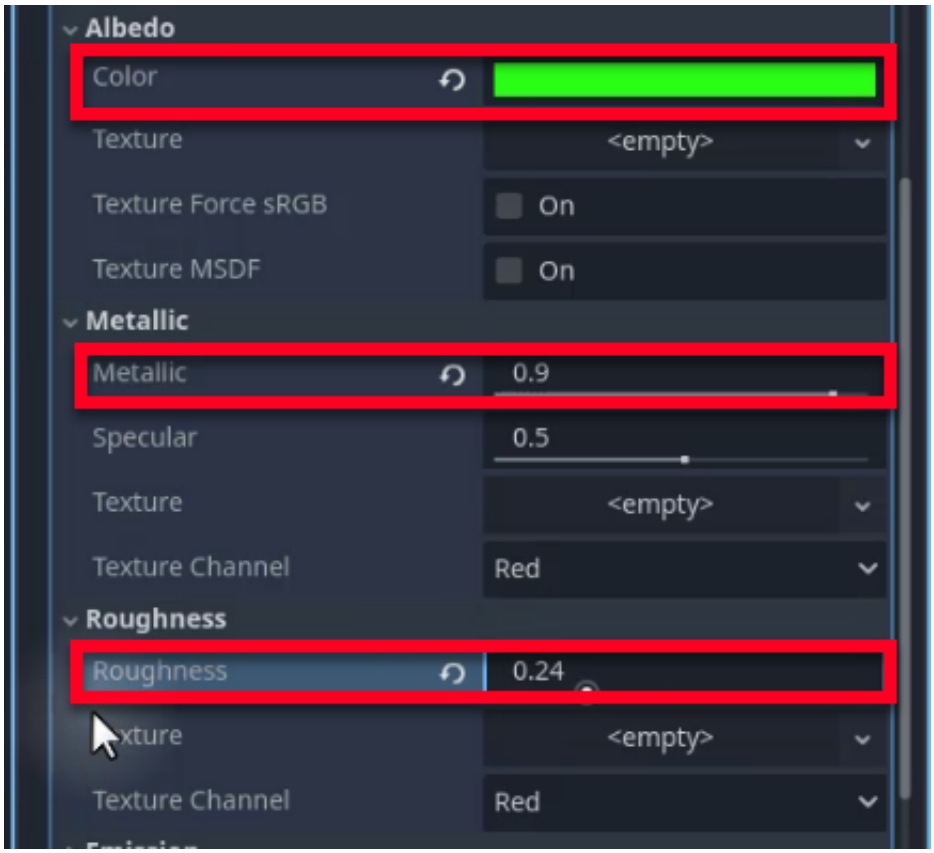
For materials that you do not plan to reuse, you can create sub-resources. To do this, first select a sphere and open the **Surface Material Override** section in the Inspector.



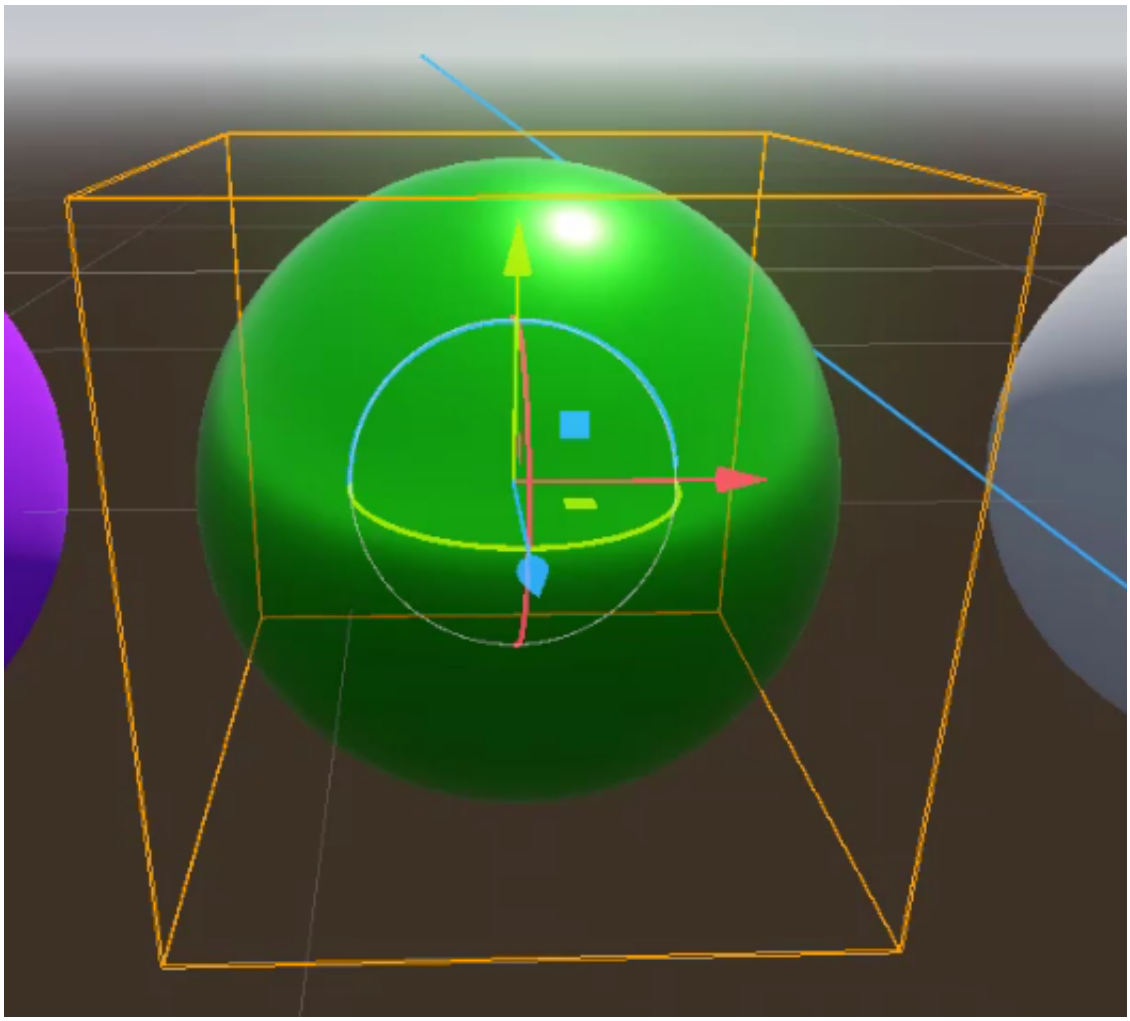
Click on the empty property and select **New StandardMaterial3D**.



Edit the material directly in the Inspector. For example, create a metallic green material by adjusting the **Albedo > Color** to green, **Metallic** to something around 0.9, and **Roughness** to something around 0.24.

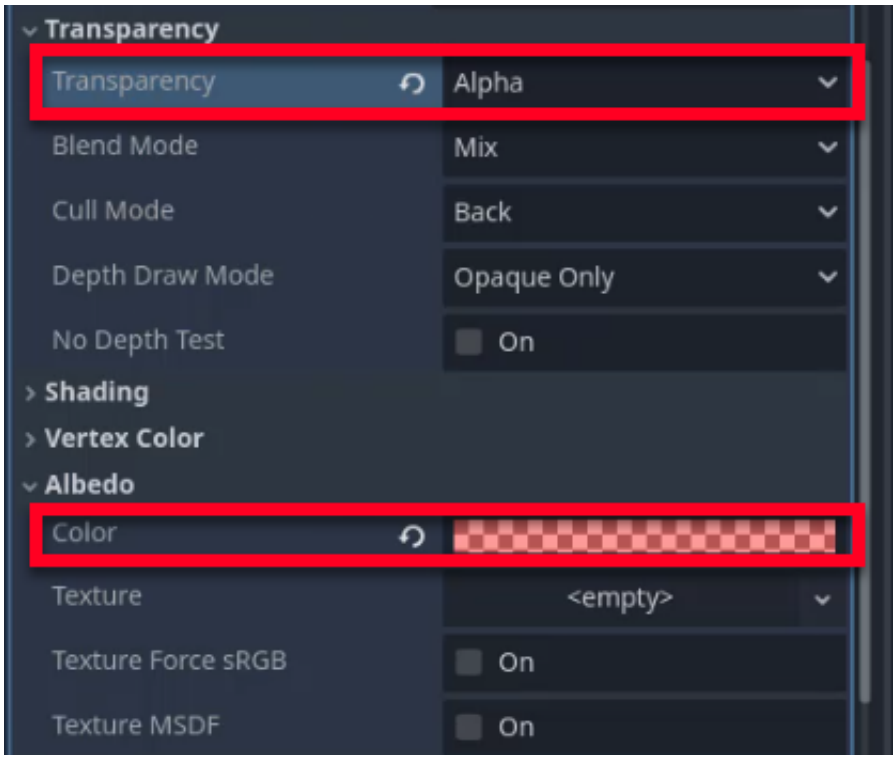


You'll see your changes automatically reflected in the editor.

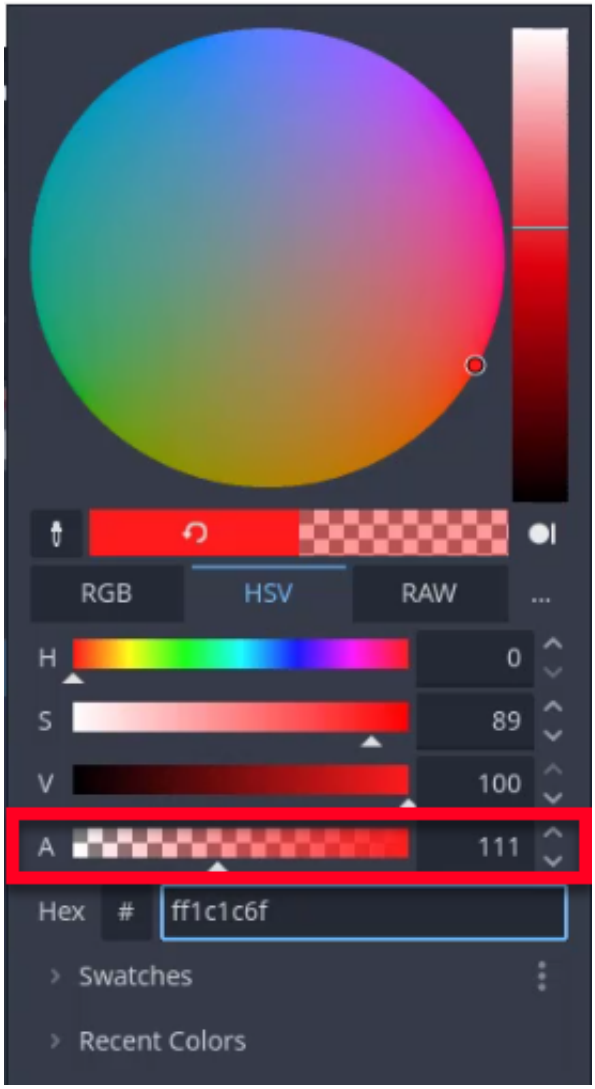


Creating Transparent Materials

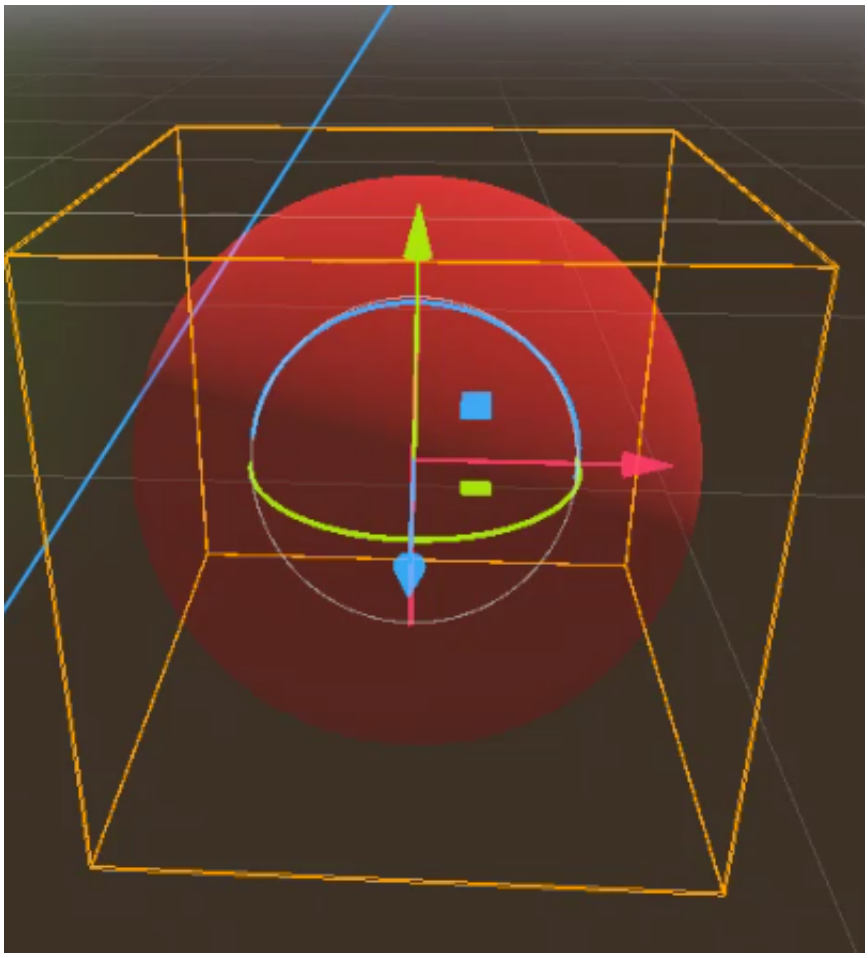
Let’s create our transparent material. First, repeat the steps above and create a new sub-resource for one of the spheres. In the Inspector, set the **Transparency** property to “Alpha” and set the **Albedo** color to a transparent red.



Remember, the color picker itself will have an alpha field to adjust, which will allow you to make the color more or less opaque.

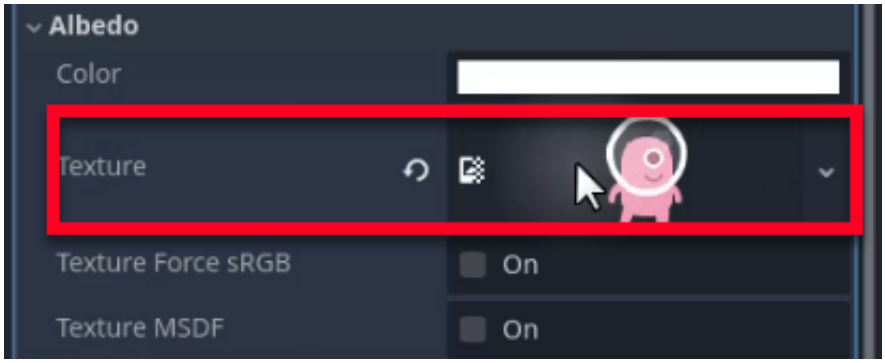


Once done, you'll have your transparent material.

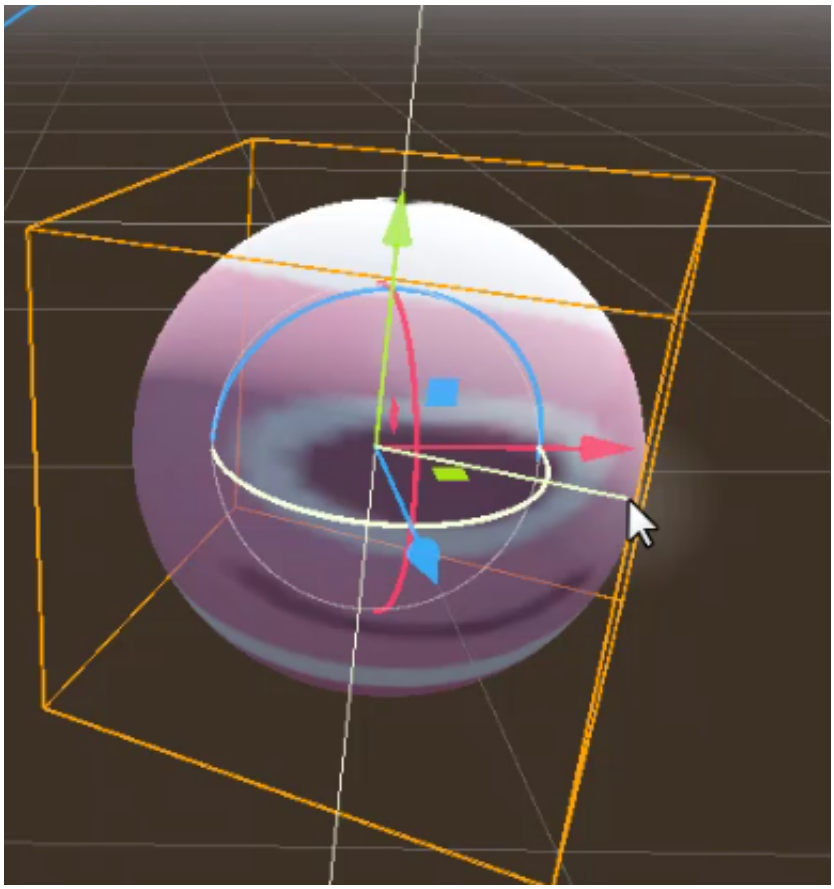


Creating Textured Materials

Finally, let's create a textured materials. Select the remain sphere and repeat the steps again to create a sub-resource. Then, in the Inspector, drag over the **Player.png** sprite from our FileSystem to the **Texture** property.



Feel free to rotate your sphere if you'd like a better look at your applied texture.

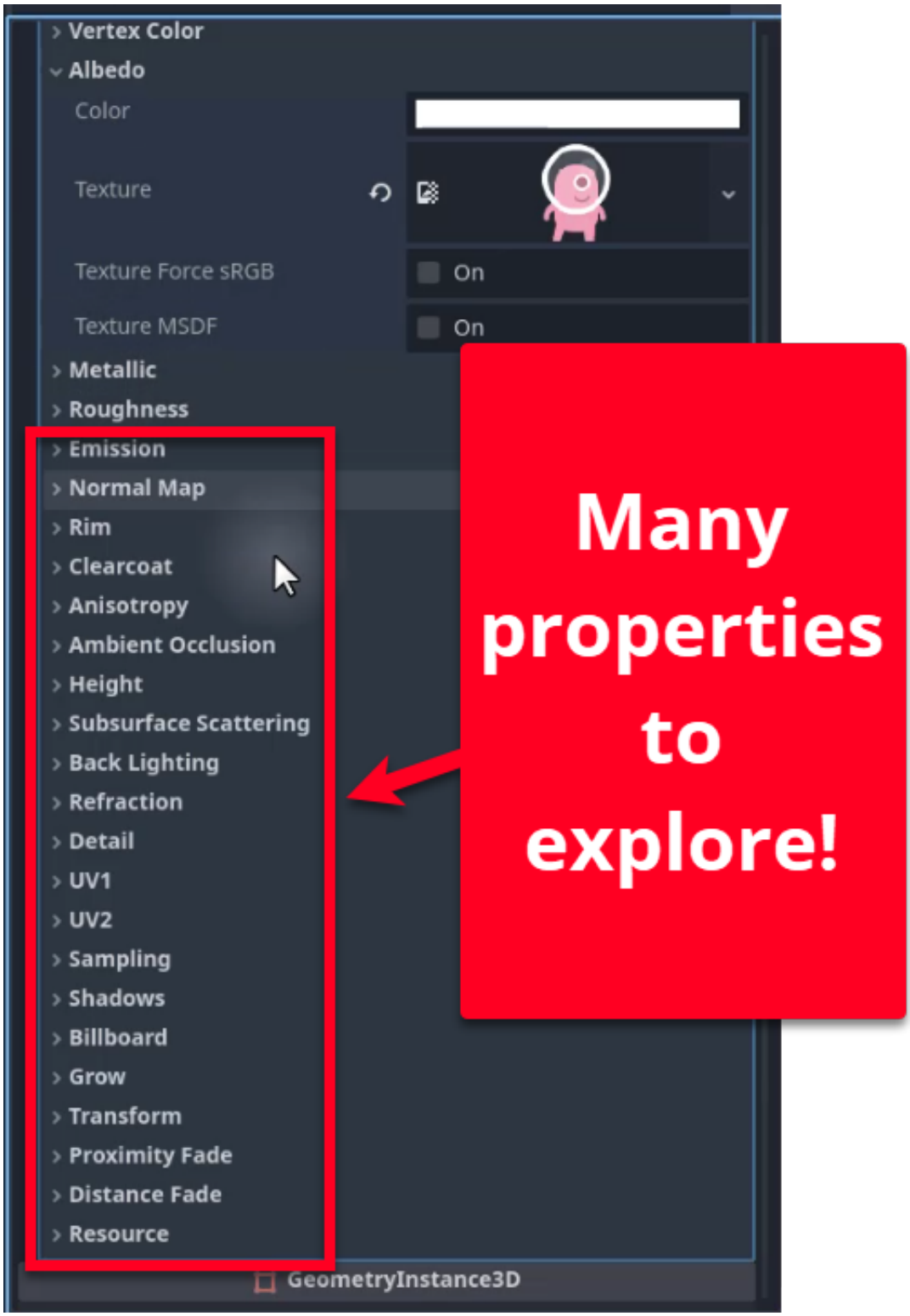


Experimenting with Material Properties

Godot offers a wide range of material properties to explore, such as:

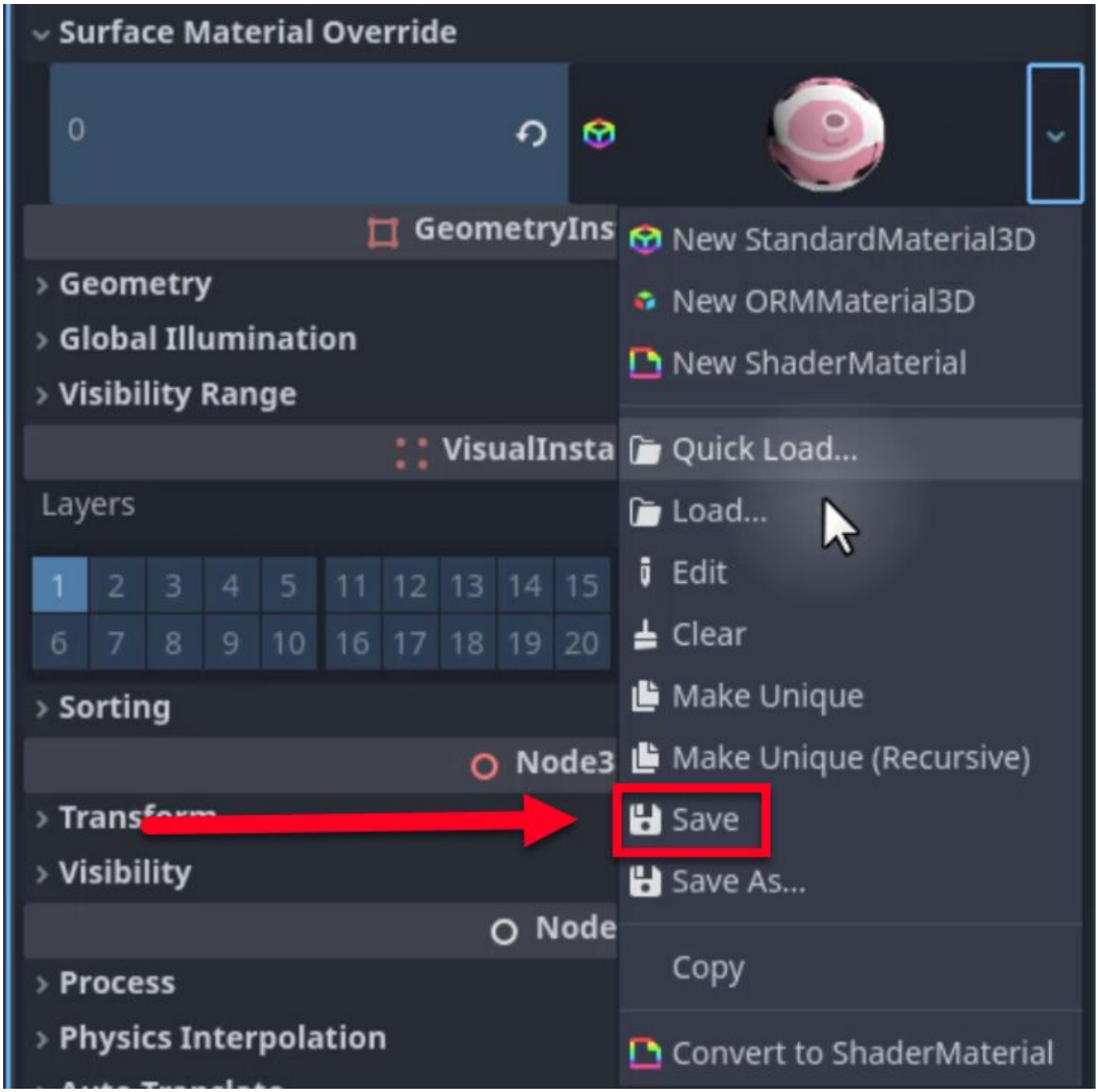
- **Emission:** Allows the material to emit light.
- **Rim:** Creates a rim lighting effect.
- **Clear Coat:** Adds a clear coat layer to the material.

Experiment with these properties to understand their effects and possibilities.

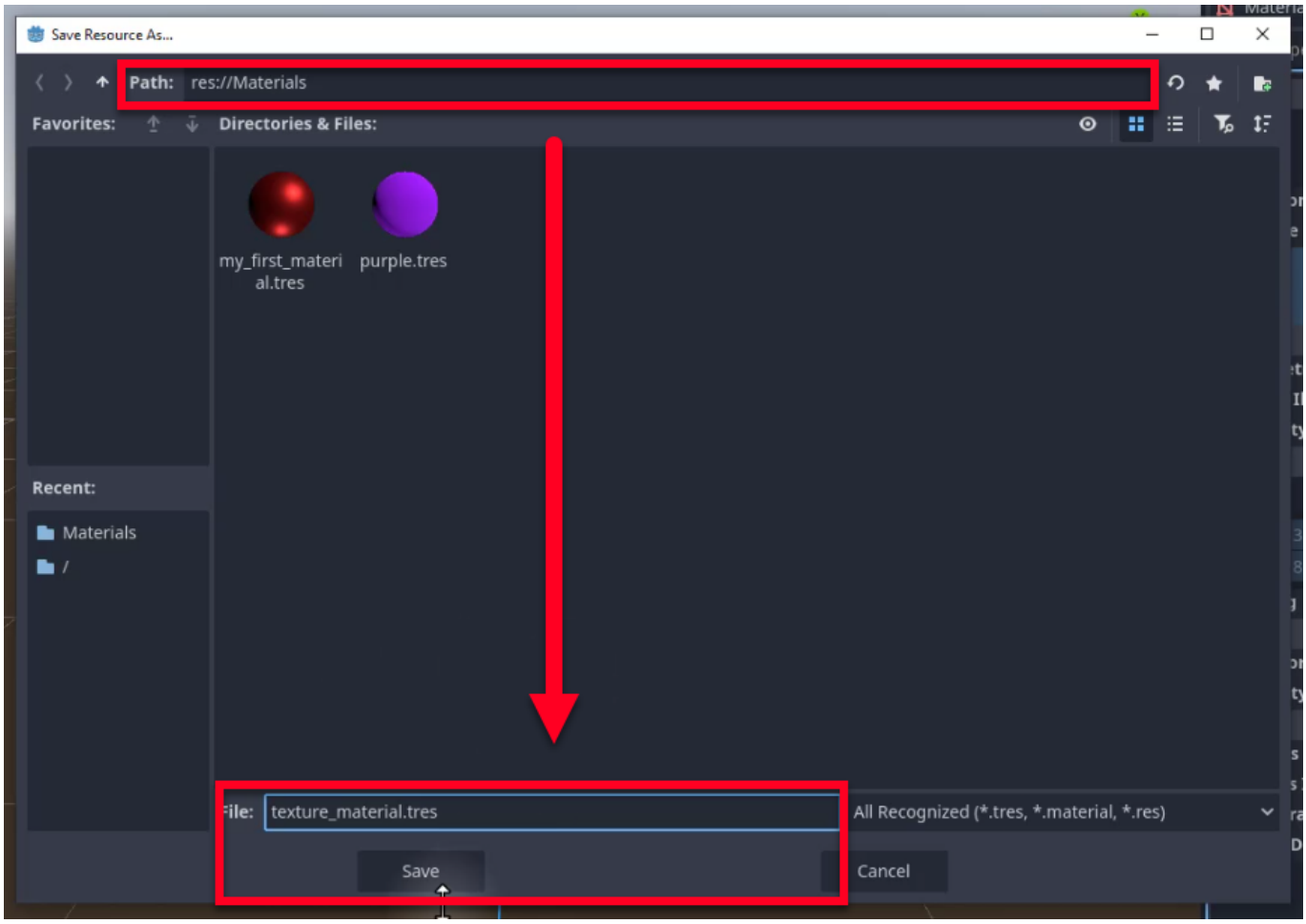


Saving Sub-Resources as Reusable Materials

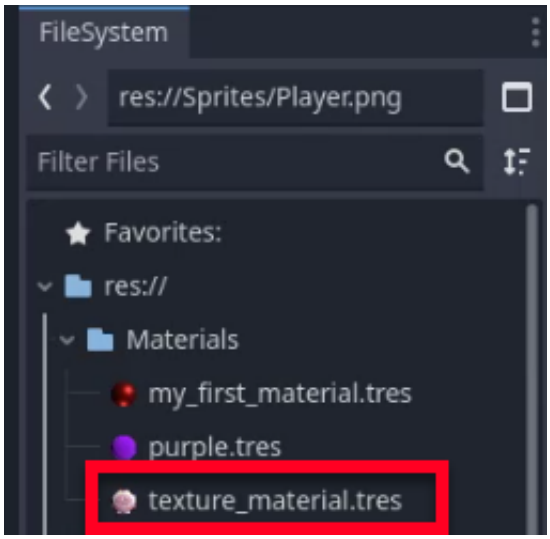
If you decide to save a sub-resource material for reuse, you can save it as a regular material in your FileSystem. First, click on the down arrow next to the material in the Inspector and select **Save**.



Choose the folder where you want to save the material, name the material, and click **Save**.



Voila! The material is now available in your FileSystem for re-use!

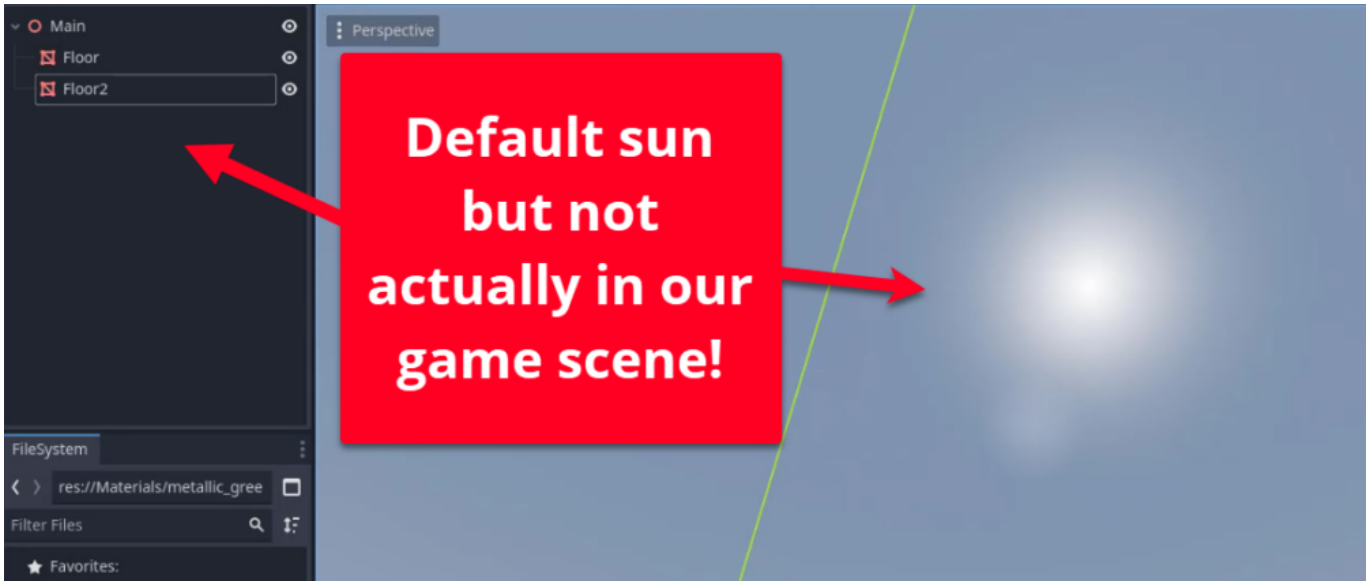


Congratulations! You have now learned how to create, apply, and manage materials in Godot. Continue experimenting with different material properties to enhance your 3D projects.

Lighting is a crucial aspect of 3D game development, as it sets the mood, changes the atmosphere, and adds realism through shadows. In this article, we will explore the main types of lighting in Godot and how to use them effectively.

Introduction to Lighting in Godot

When you start a new scene in Godot, you might notice that there is a default light source acting as the sun. However, this light is not active during gameplay unless manually added.

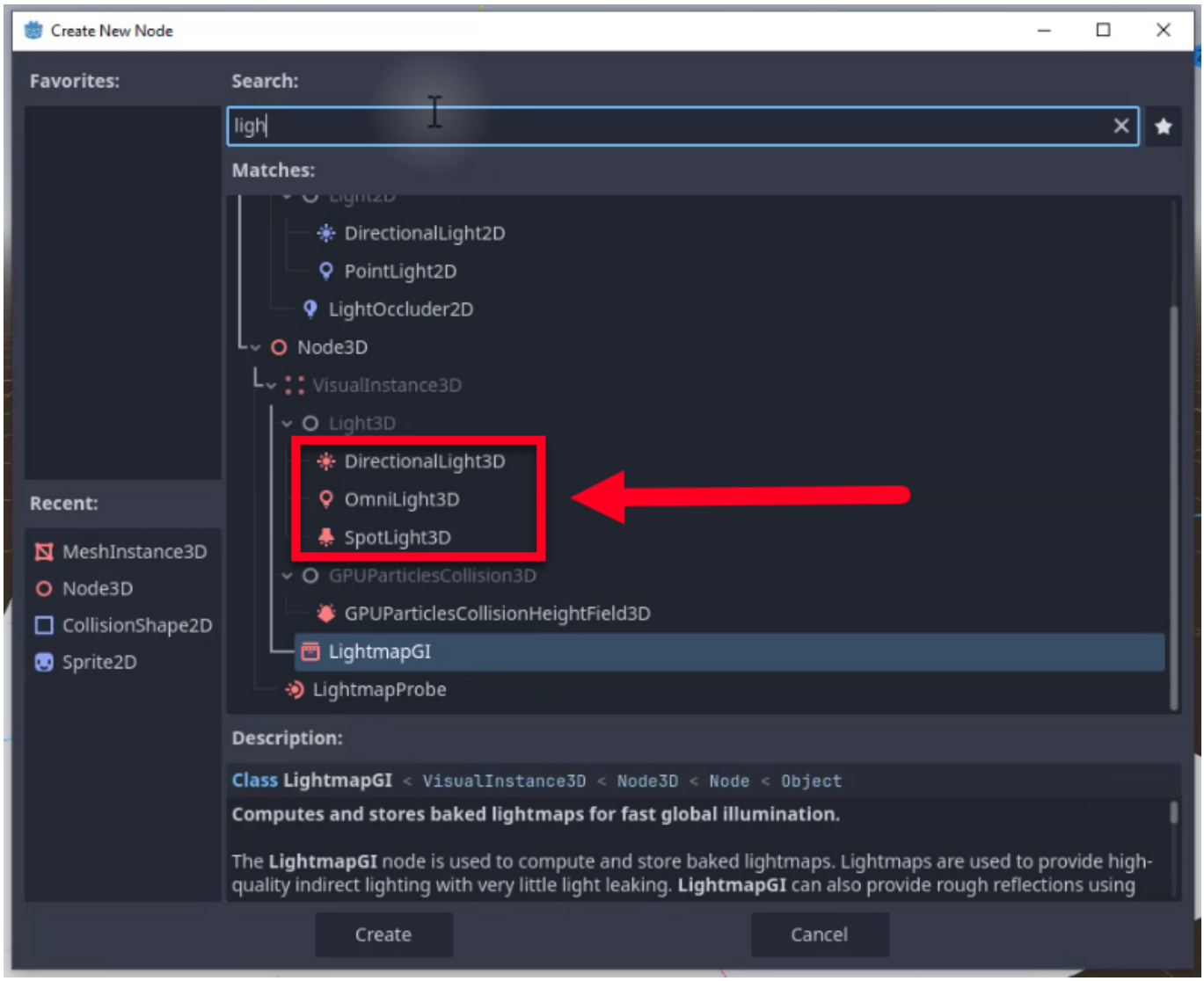


Let’s dive into the different types of lights and their properties.

Types of Lights in Godot

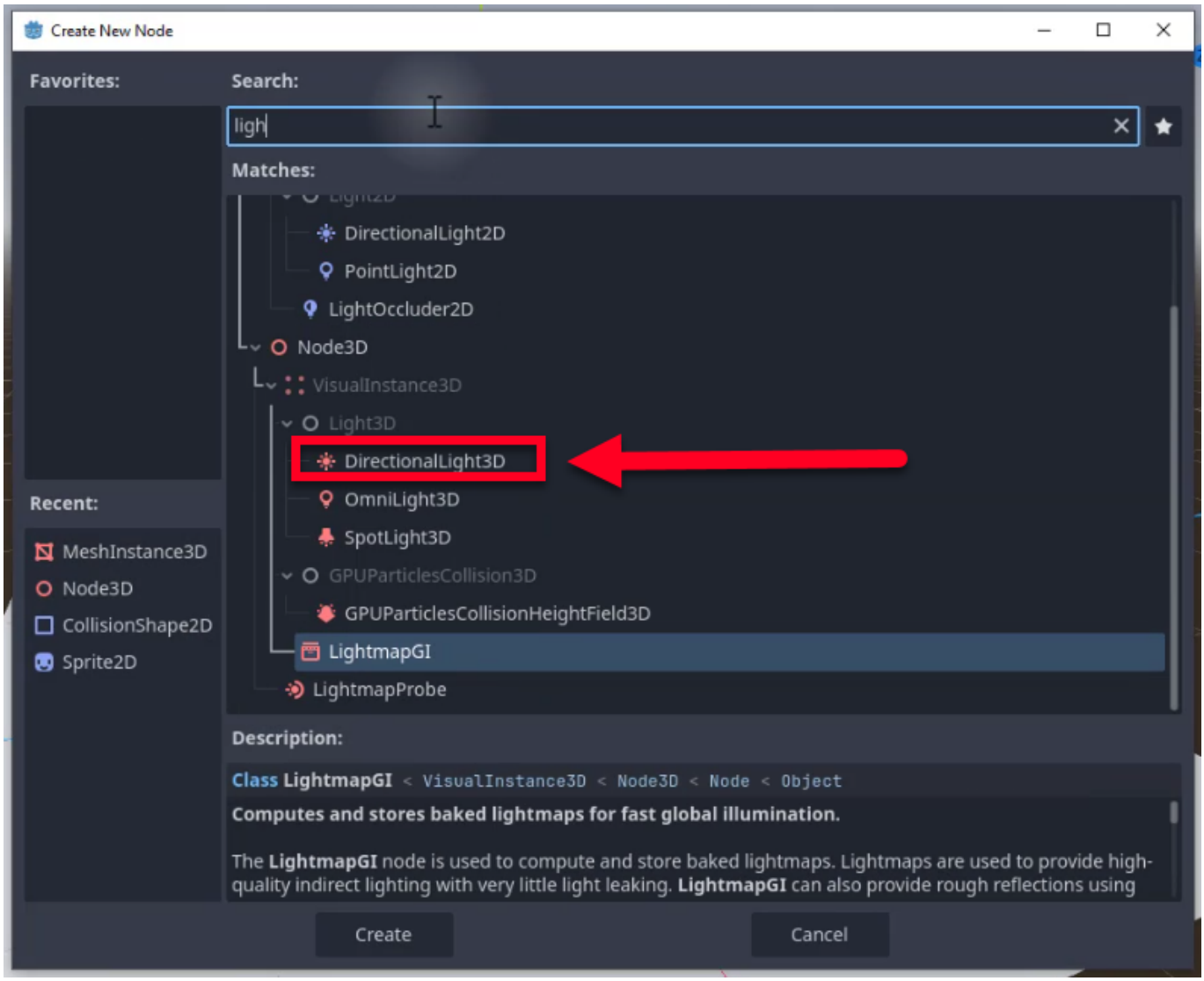
Godot offers three primary types of lights for 3D environments:

- **Directional Light**: Acts as a global light source like the sun or moon.
- **Omni Light**: Emits light in all directions, similar to a candle or light bulb.
- **Spotlight**: Projects light in a cone shape, resembling a flashlight or street lamp.

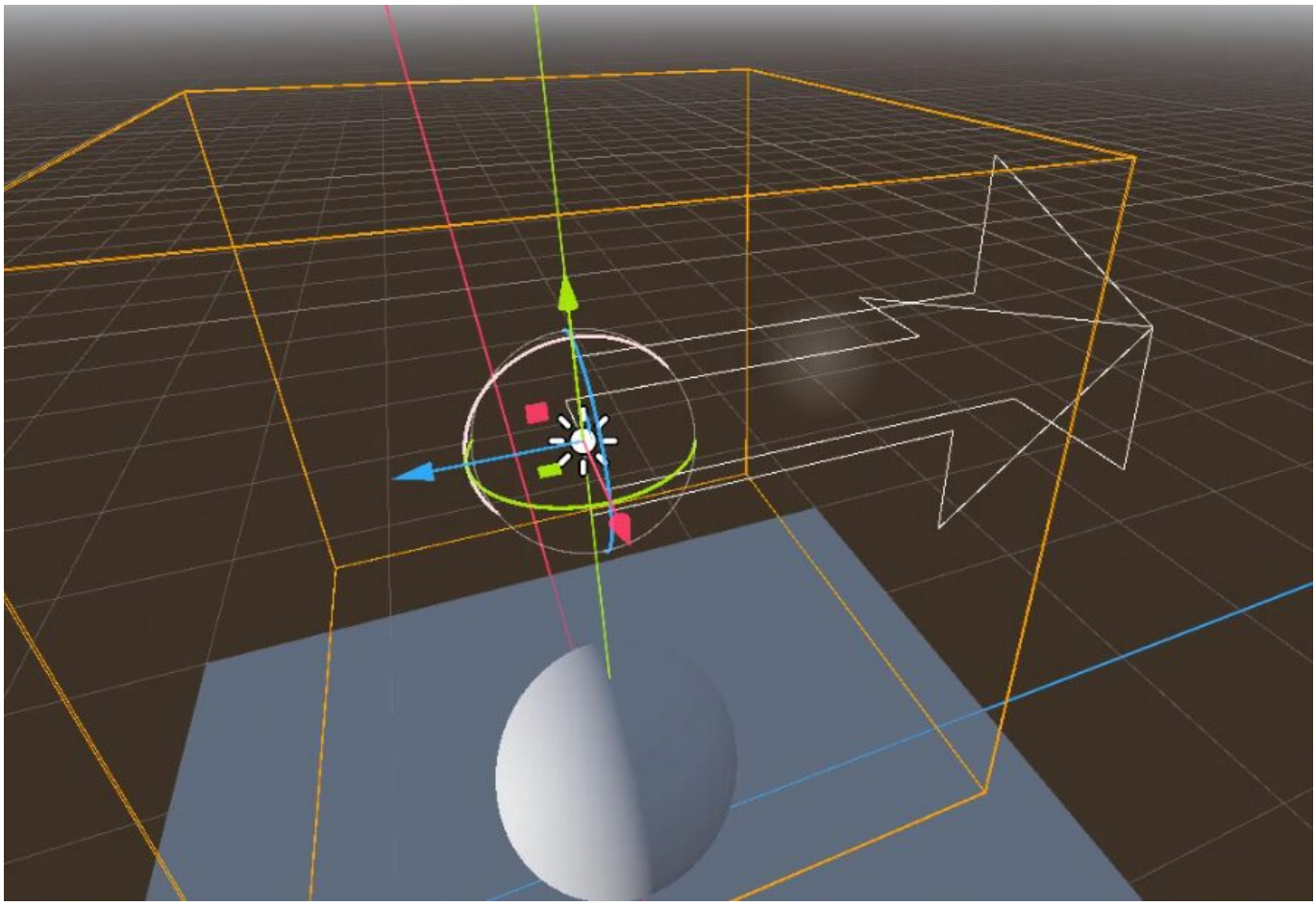


Directional Light

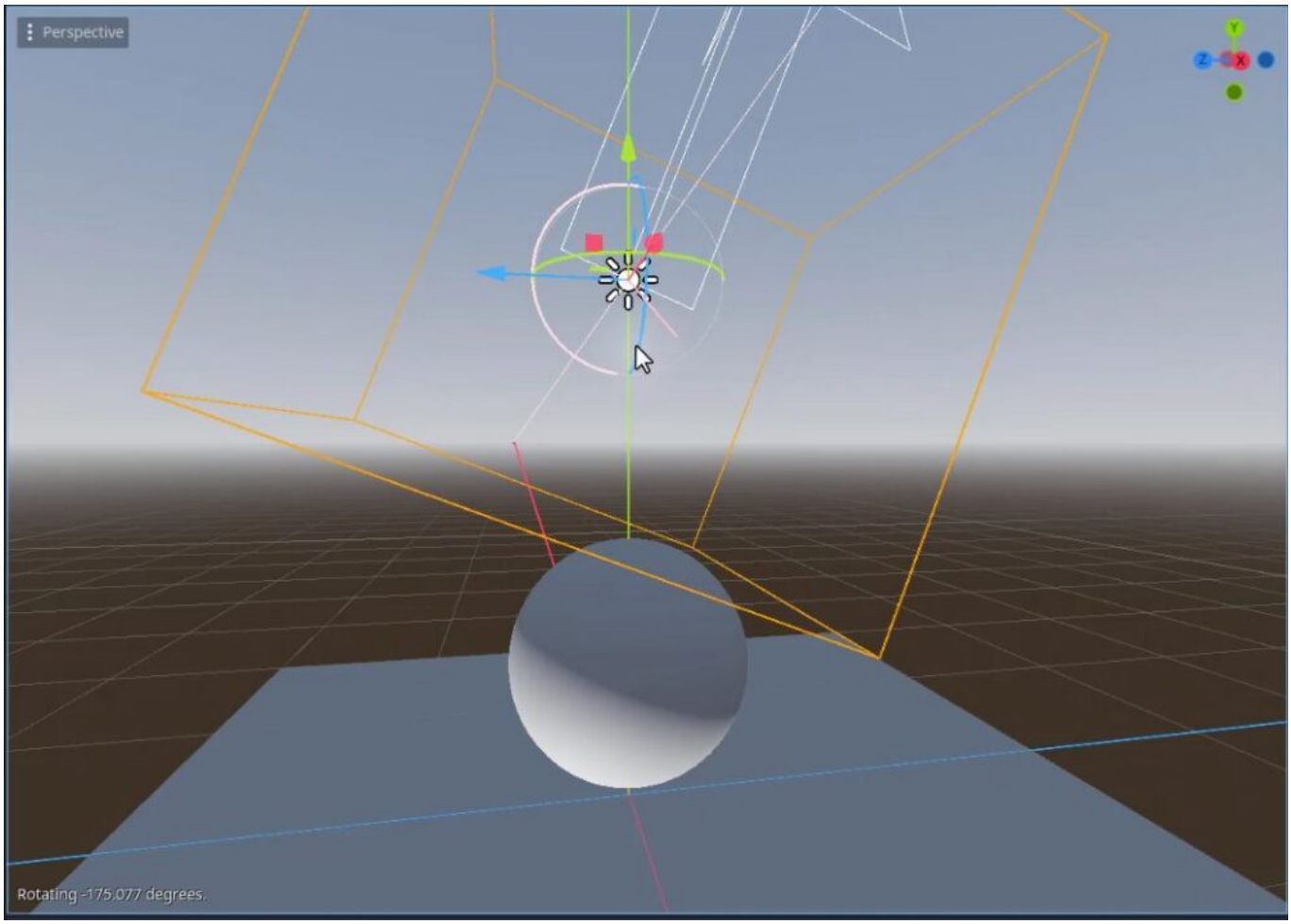
A Directional Light is ideal for creating overall lighting in your scene. To test it out, add a new node and search for “DirectionalLight3D”.

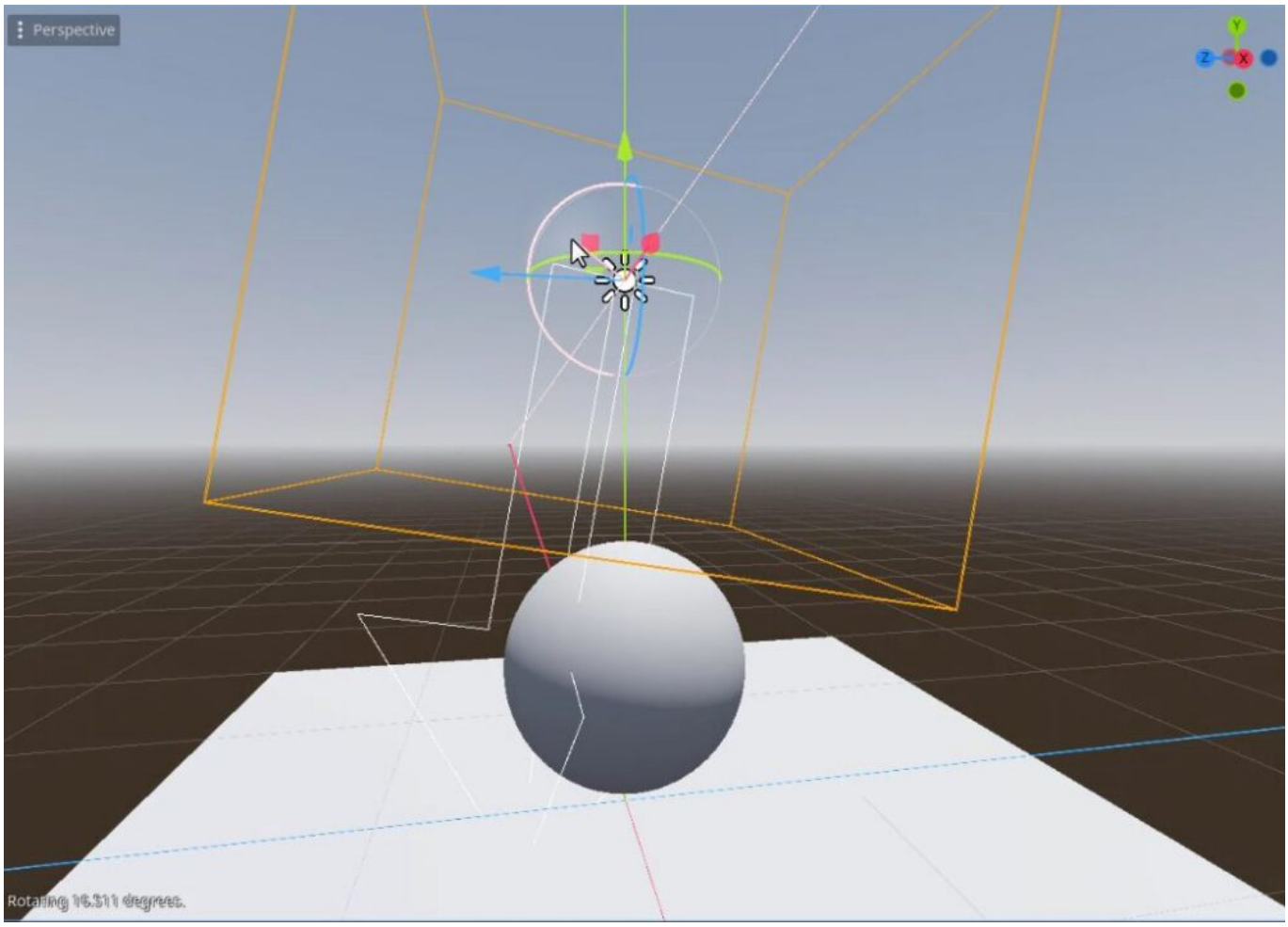


Once created, you can position your light anywhere in your scene, as a **Directional Light's position does not affect its lighting**.

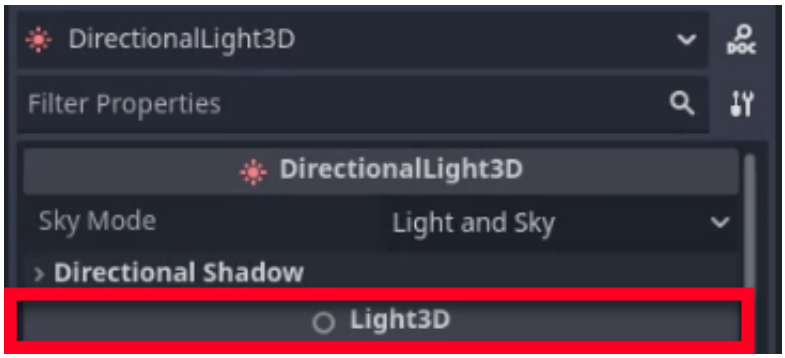


As you position the light, you'll notice the arrow. This arrow indicates the direction of the light. You can rotate the node to change the lighting direction until you get something you like.

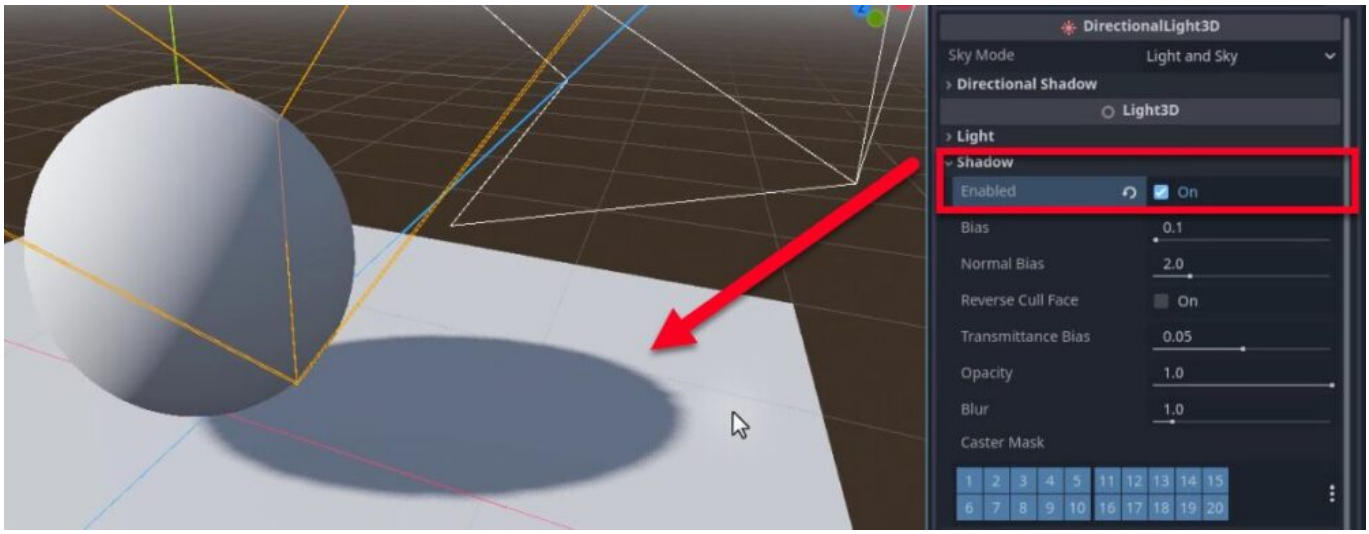




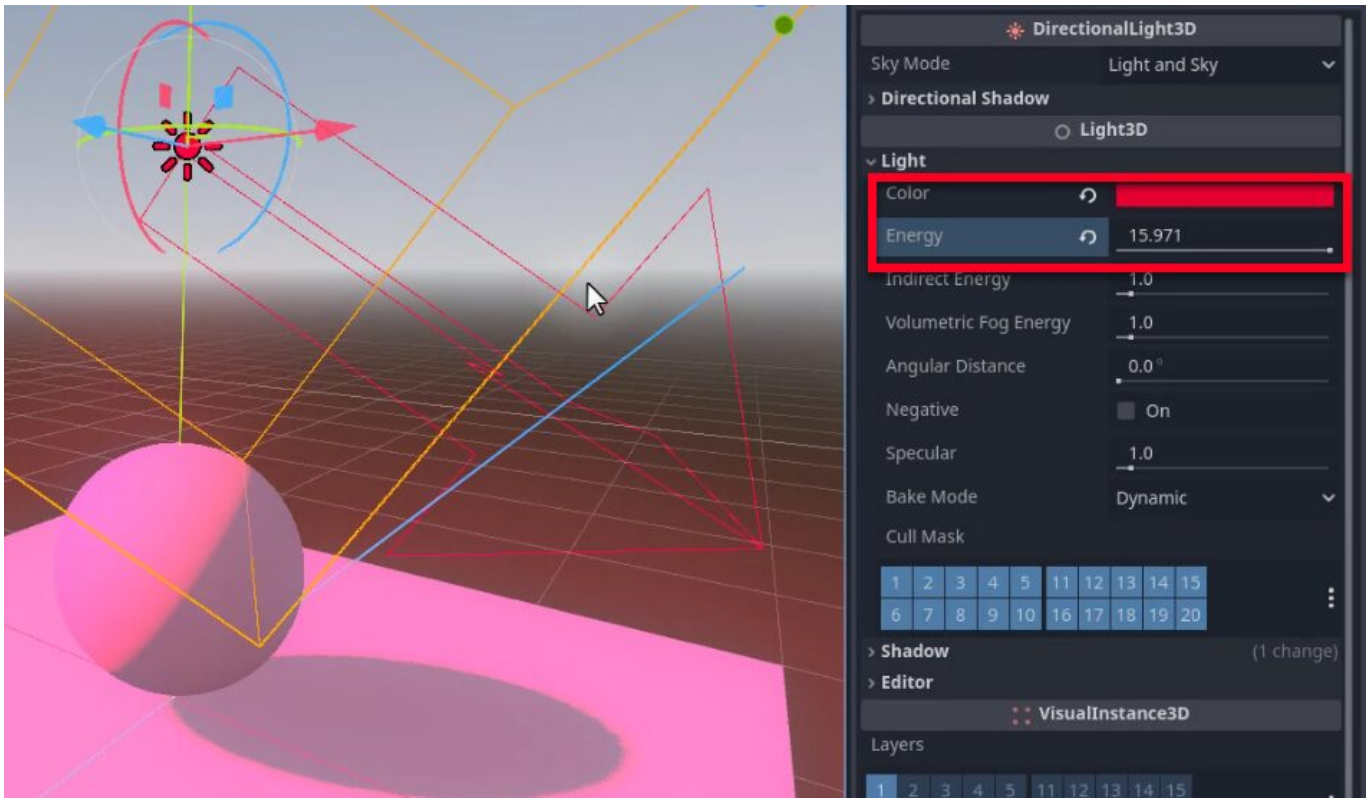
You can also change a light's properties from the Inspector in the Light3D node area (with an applicable Light node selected, of course).



For example, you can enable shadows in the Inspector to add realism.



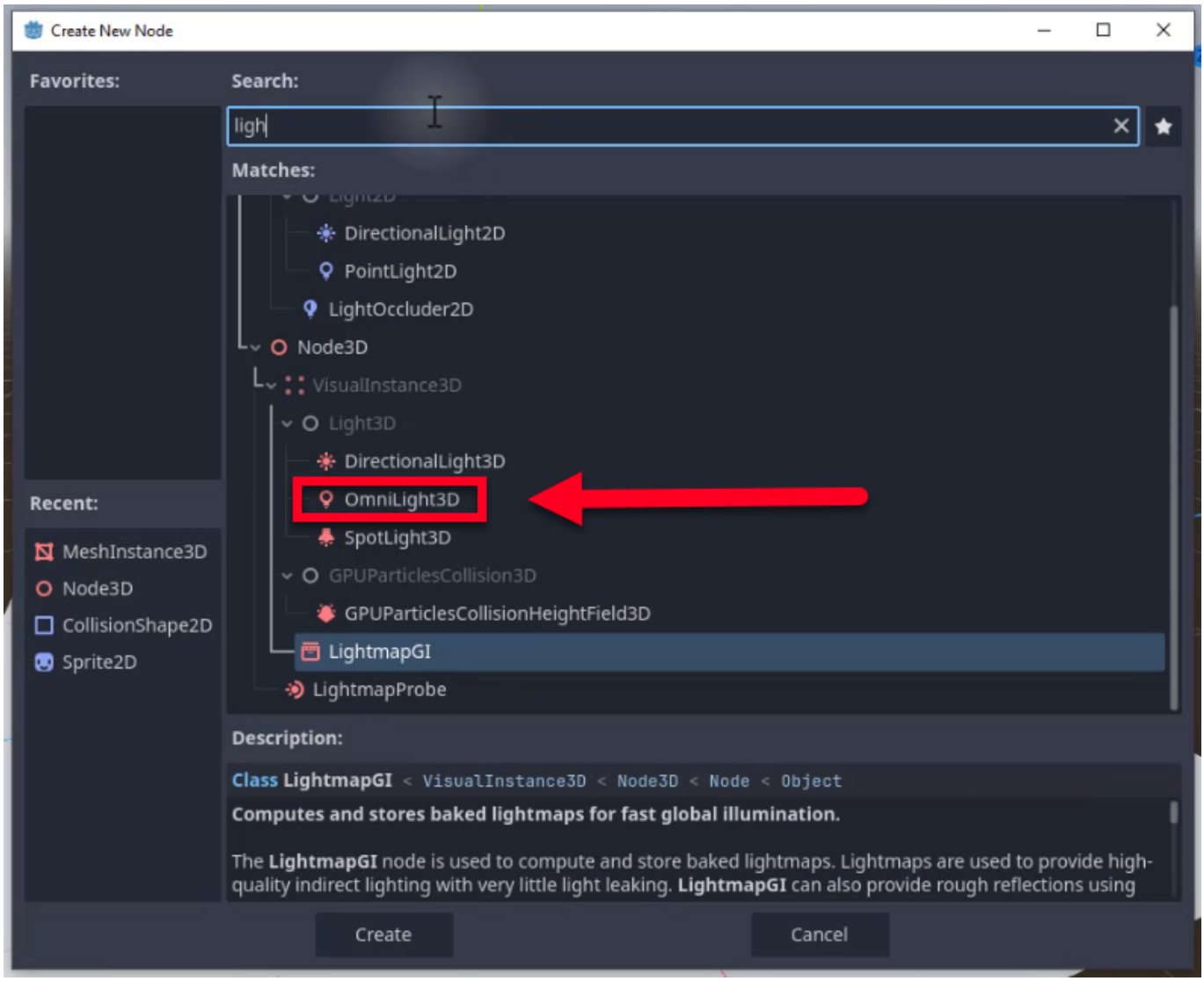
Alternatively, you can adjust aspects like the color and energy (i.e. how bright the light is) properties to customize the light.



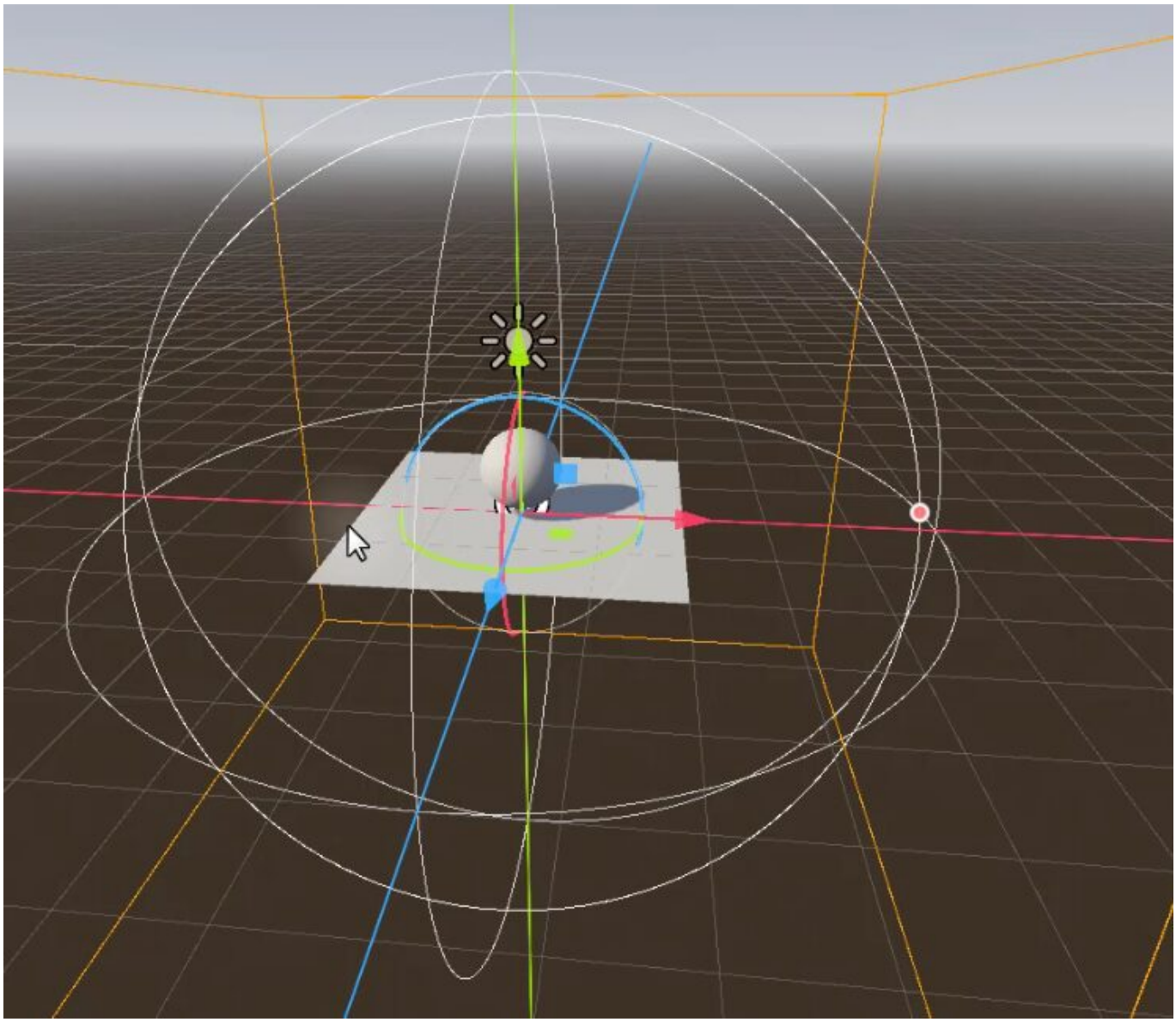
There are many settings to play around with to get the light you want for your game!

Omni Light

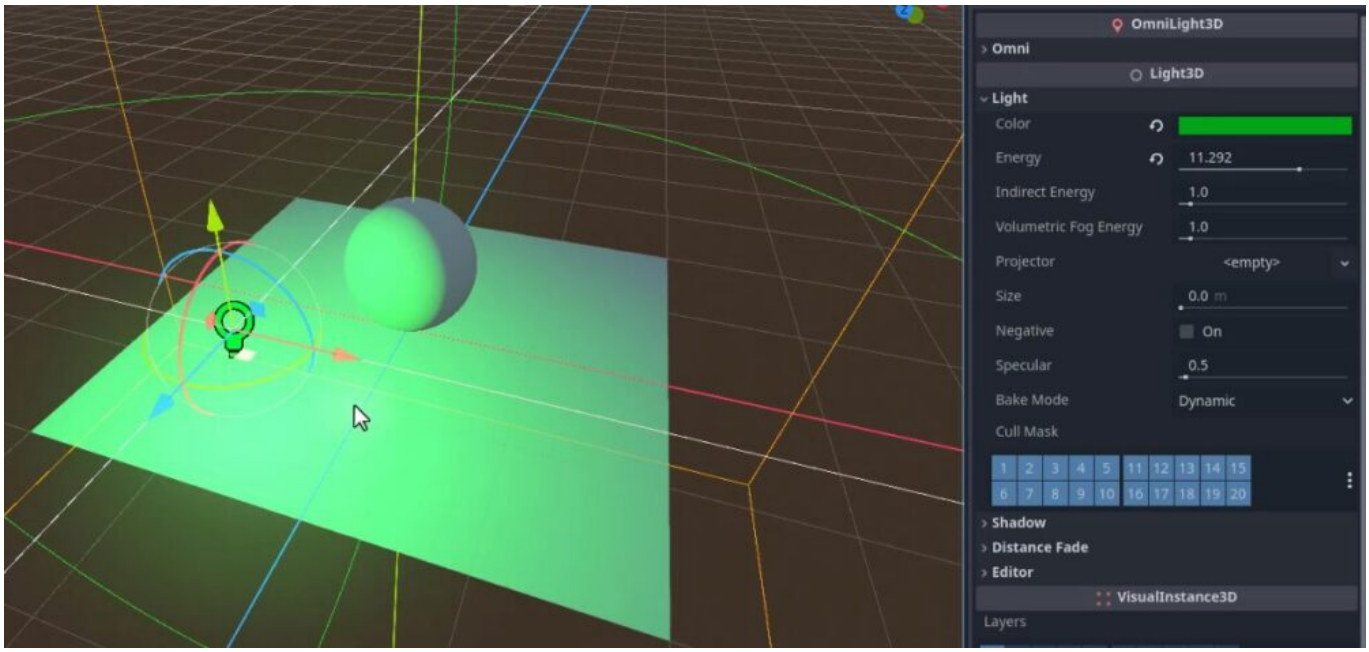
An Omni Light emits light in all directions, making it perfect for localized lighting effects. To test it out, add a new node and search for “OmniLight3D”.



In this light's case, the position in your scene does matter, as the sphere around it defines the light's range. You can click and drag the sphere to increase or shrink the light's range.

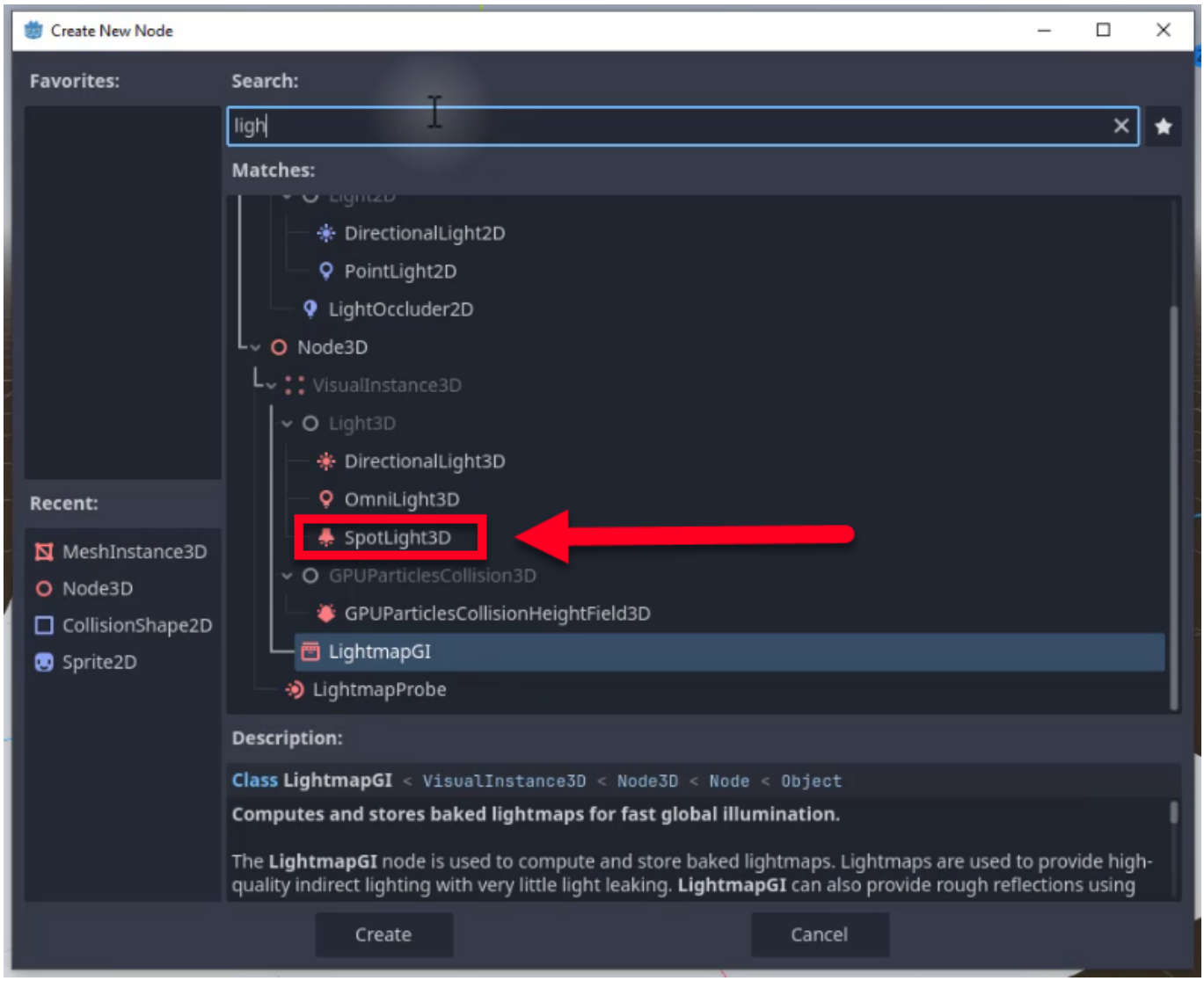


Like the Directional Light, you can adjust the OmniLight3D’s properties in the Inspector to create a custom light suited to your own tastes.

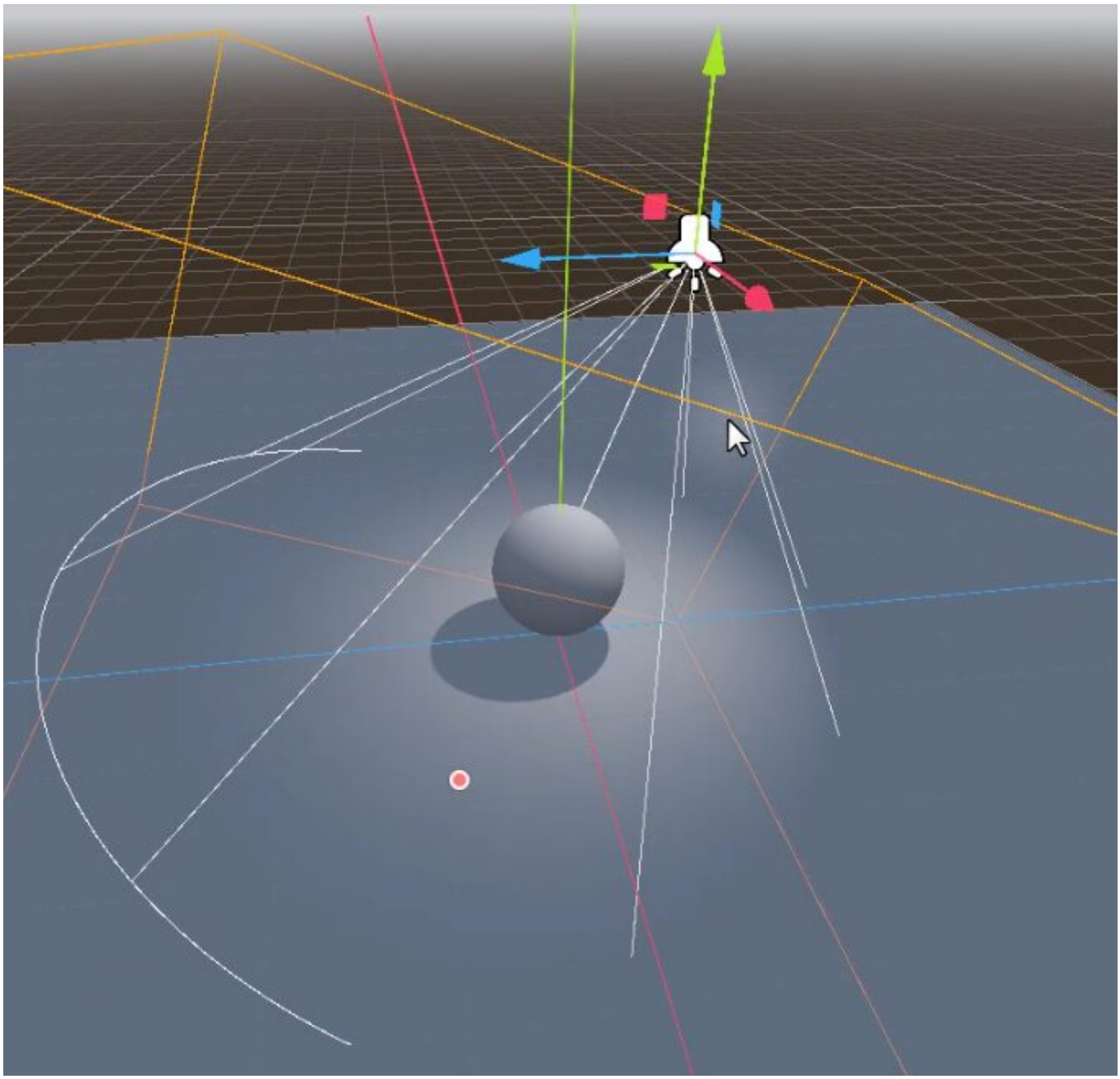


Spotlight

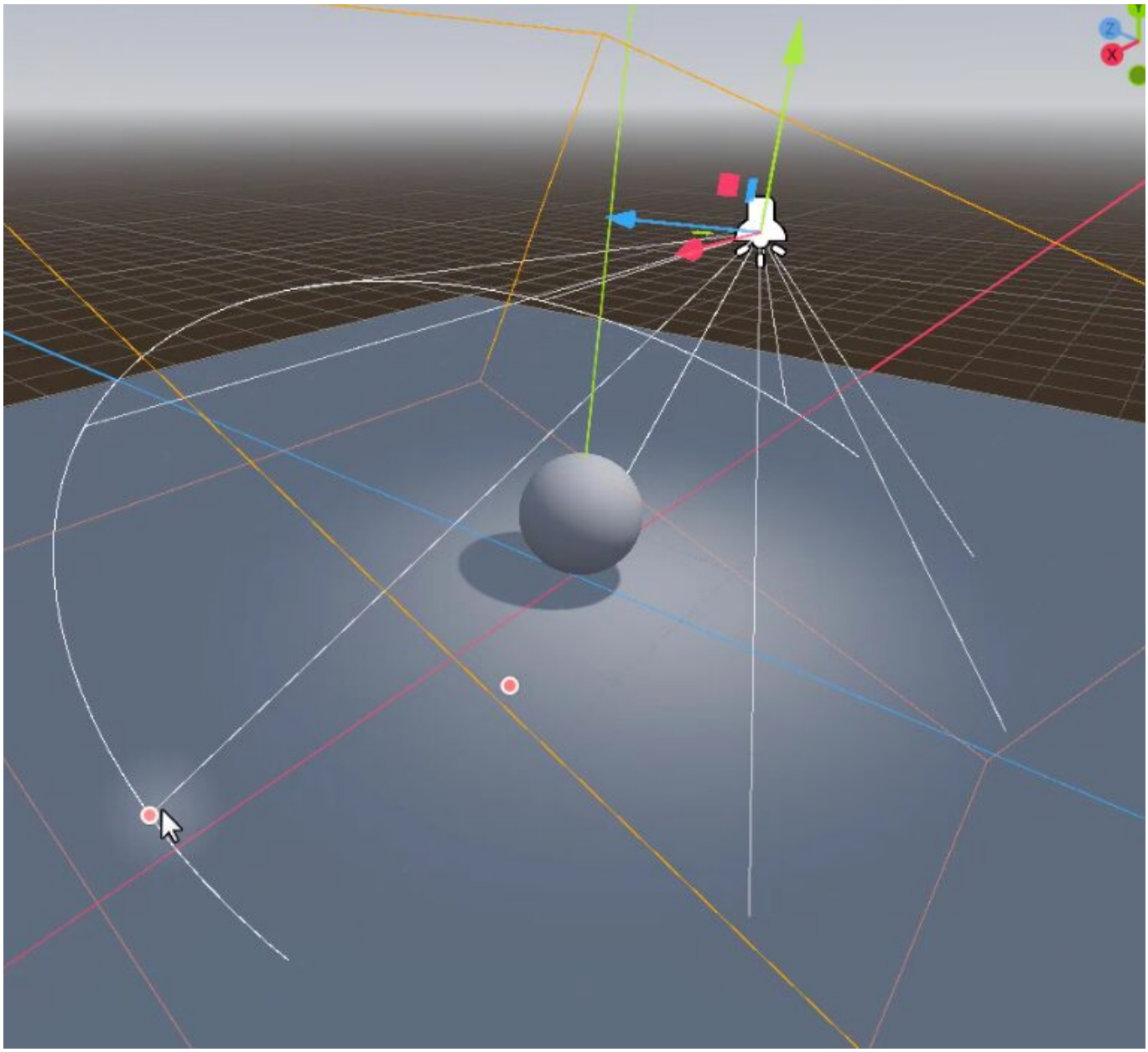
A Spotlight projects light in a cone shape, ideal for focused lighting effects. To test it out, add a new node and search for “SpotLight3D”.



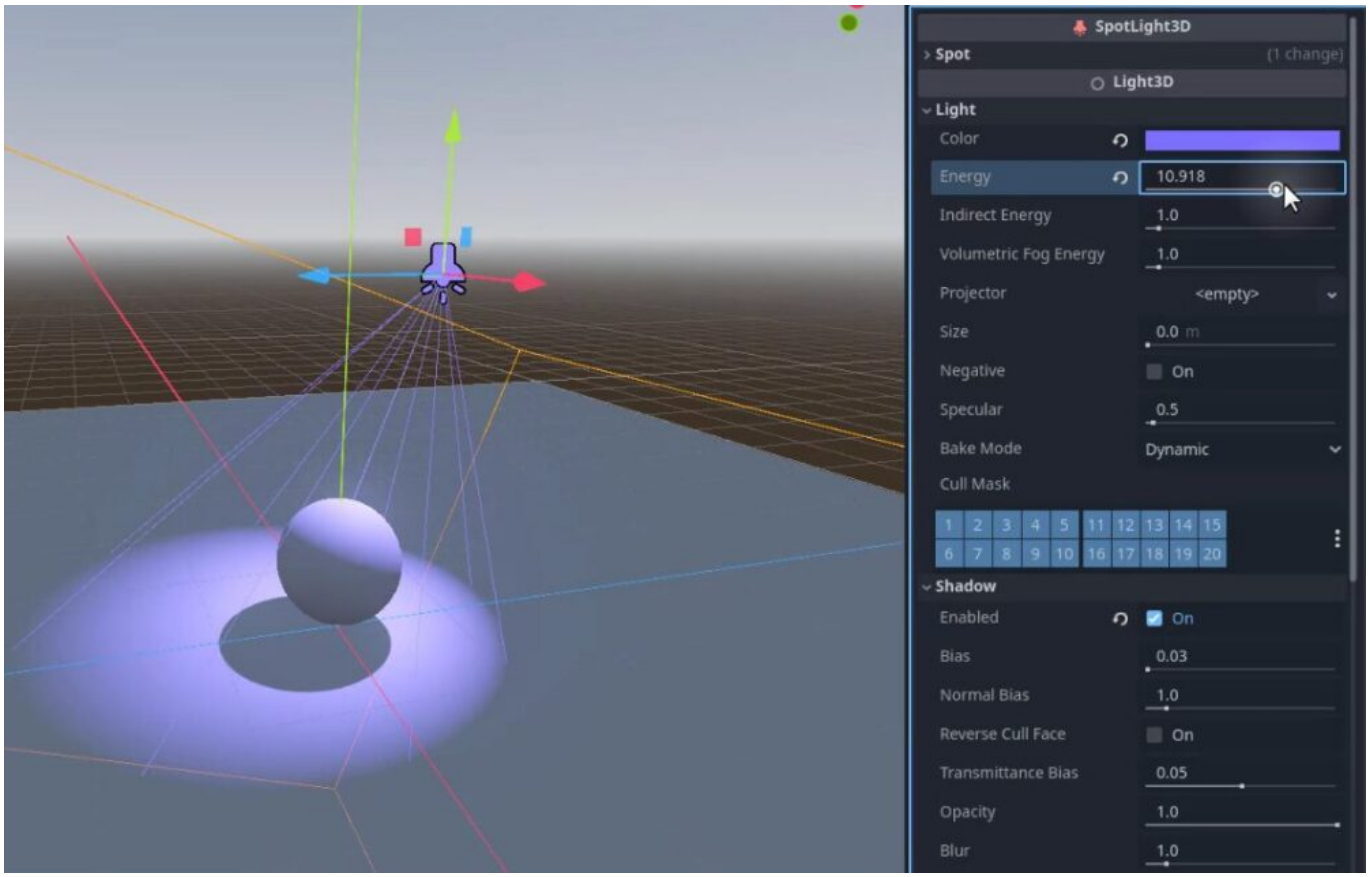
In the case of the SpotLight3D node, both position and rotation matter. As such, you should position the light where you want its origin in the scene to be, and rotate it so that the cone (which indicates the light's direction and reach) faces the orientation you need the light to go.



Somewhat similar to the OmniLight3D, you can adjust the angle and range of the light by clicking and dragging on the cone.



Like the other lights, you can adjust various properties in the Inspector to customize the light.



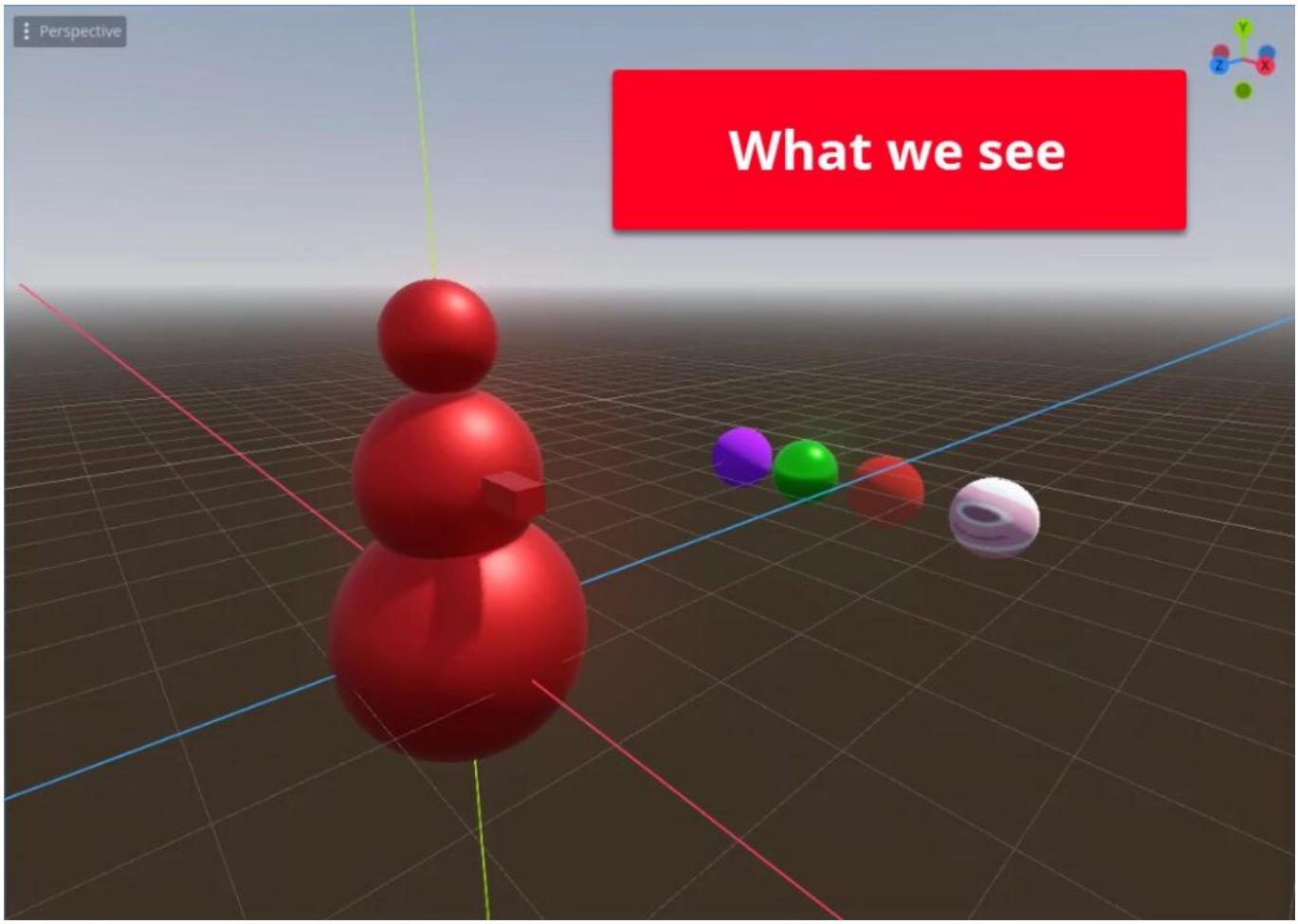
Conclusion

Understanding and utilizing different types of lighting in Godot can significantly enhance the visual appeal of your 3D games. By experimenting with Directional Lights, Omni Lights, and Spotlights, you can create a variety of atmospheres and moods. Happy game developing!

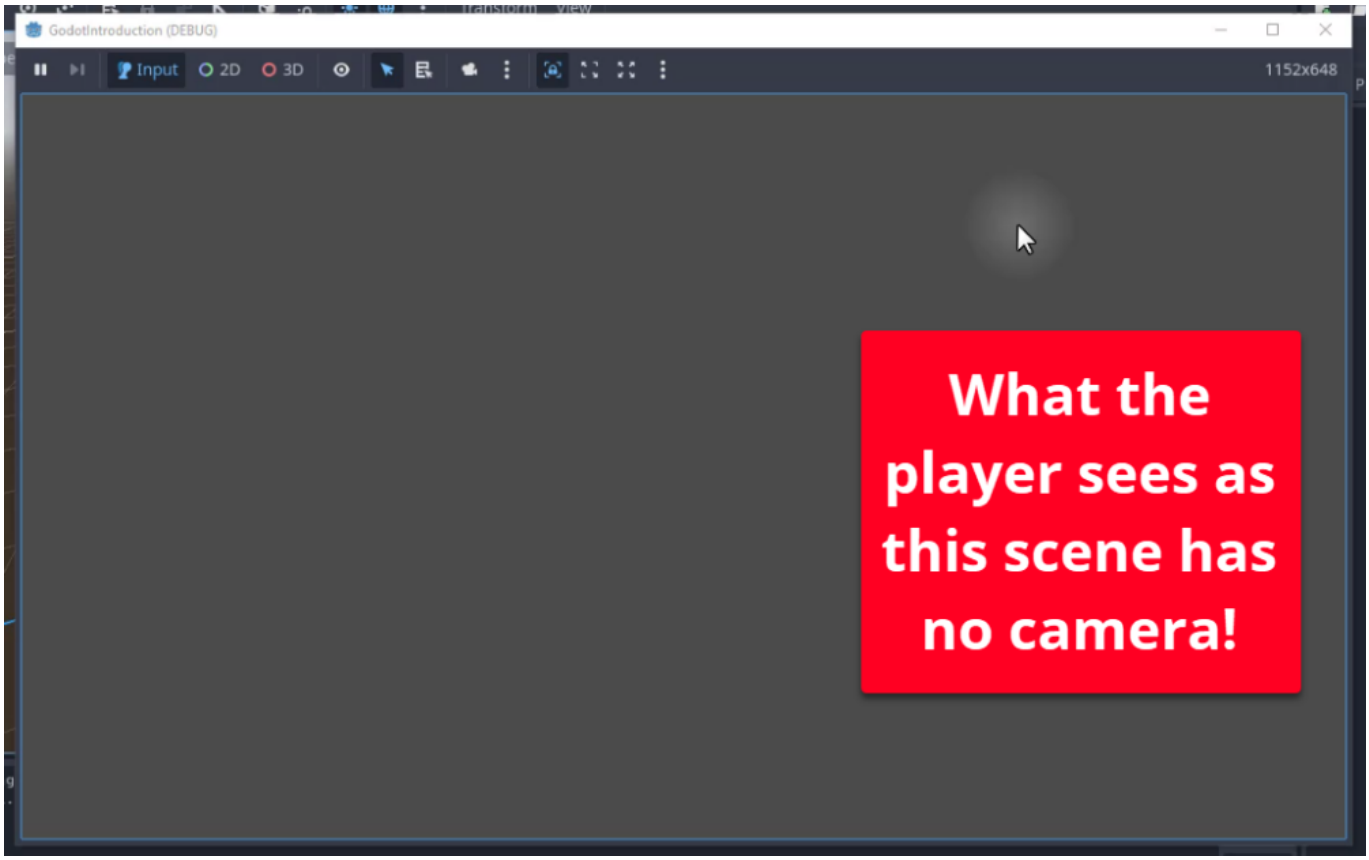
In this lesson, we will cover cameras in Godot. Cameras in Godot are essential for defining the viewport that players see when they play your game. Unlike the editor, where you can freely move around and adjust nodes, the camera node provides a specific viewpoint for the gameplay experience.

Understanding the Camera Node

The camera node is crucial for rendering the scene from a particular perspective. In the Godot editor, you have the flexibility to navigate and design your scenes.

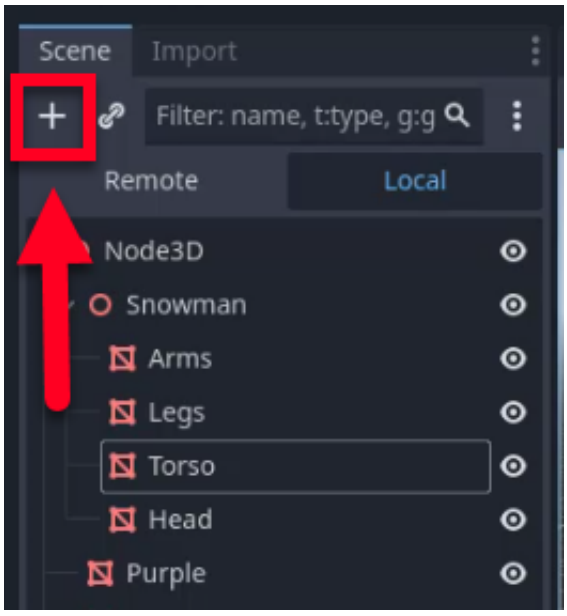


However, during gameplay, the camera node determines what the player sees.

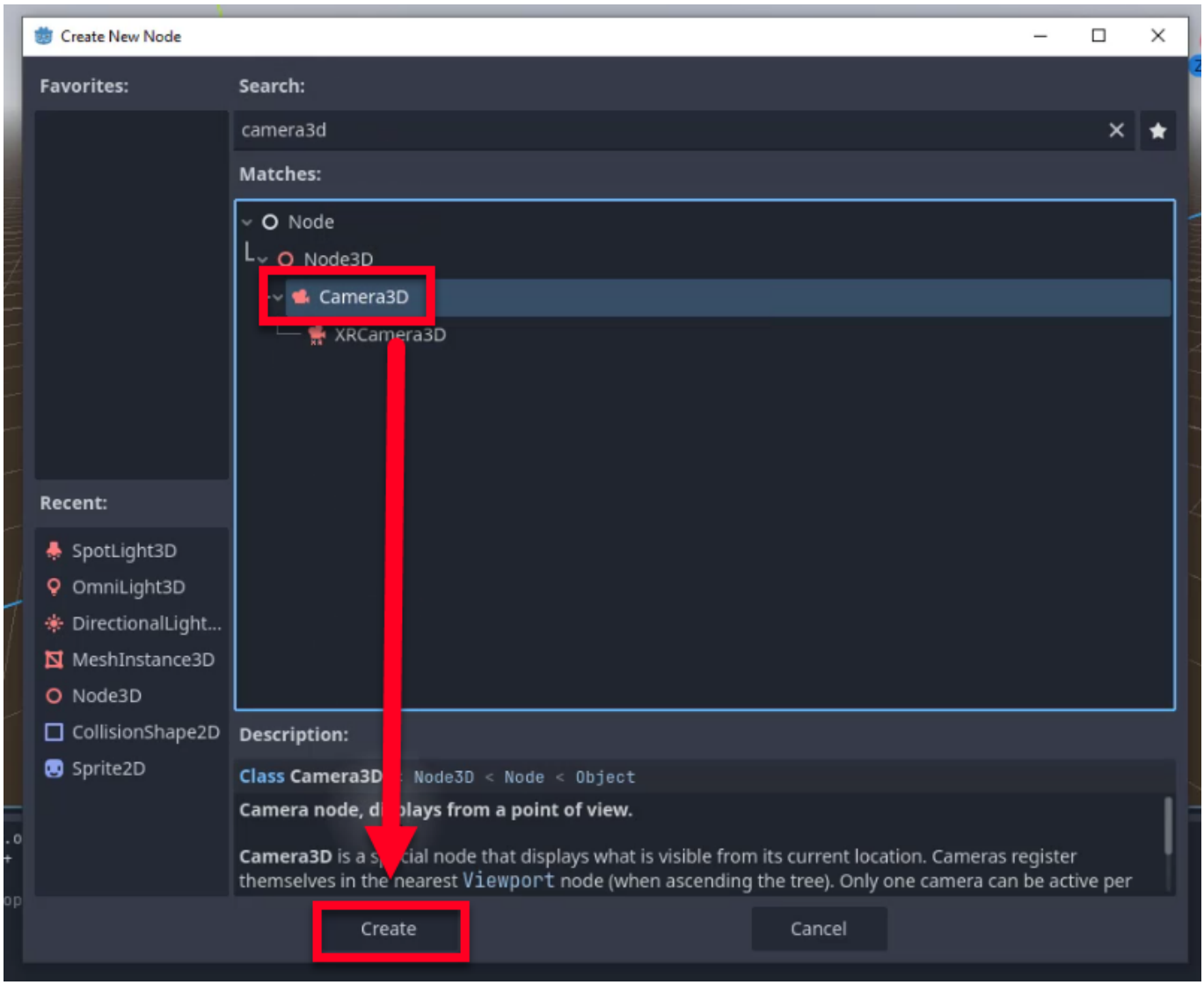


Setting Up a 3D Camera

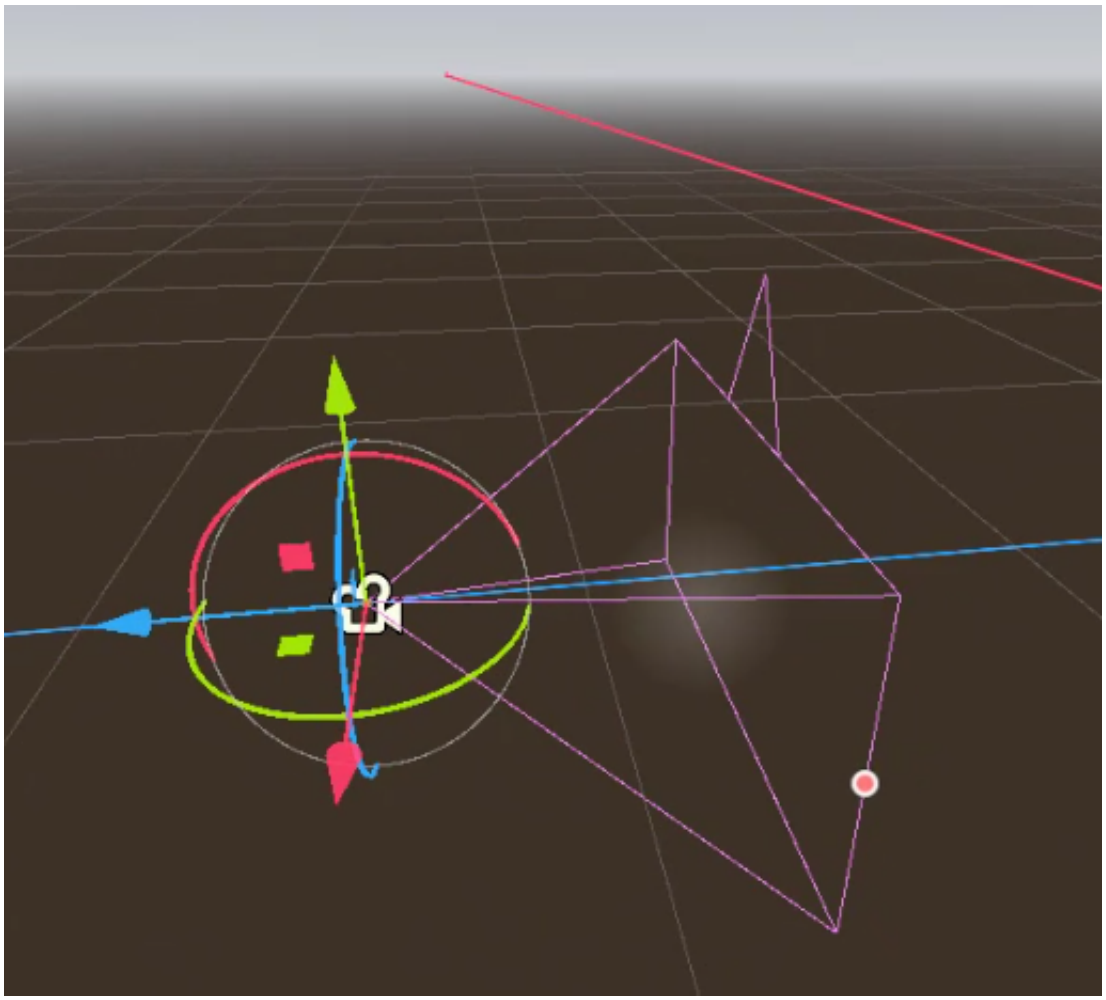
Let’s try creating a 3D camera. Click the plus icon to add a new node.



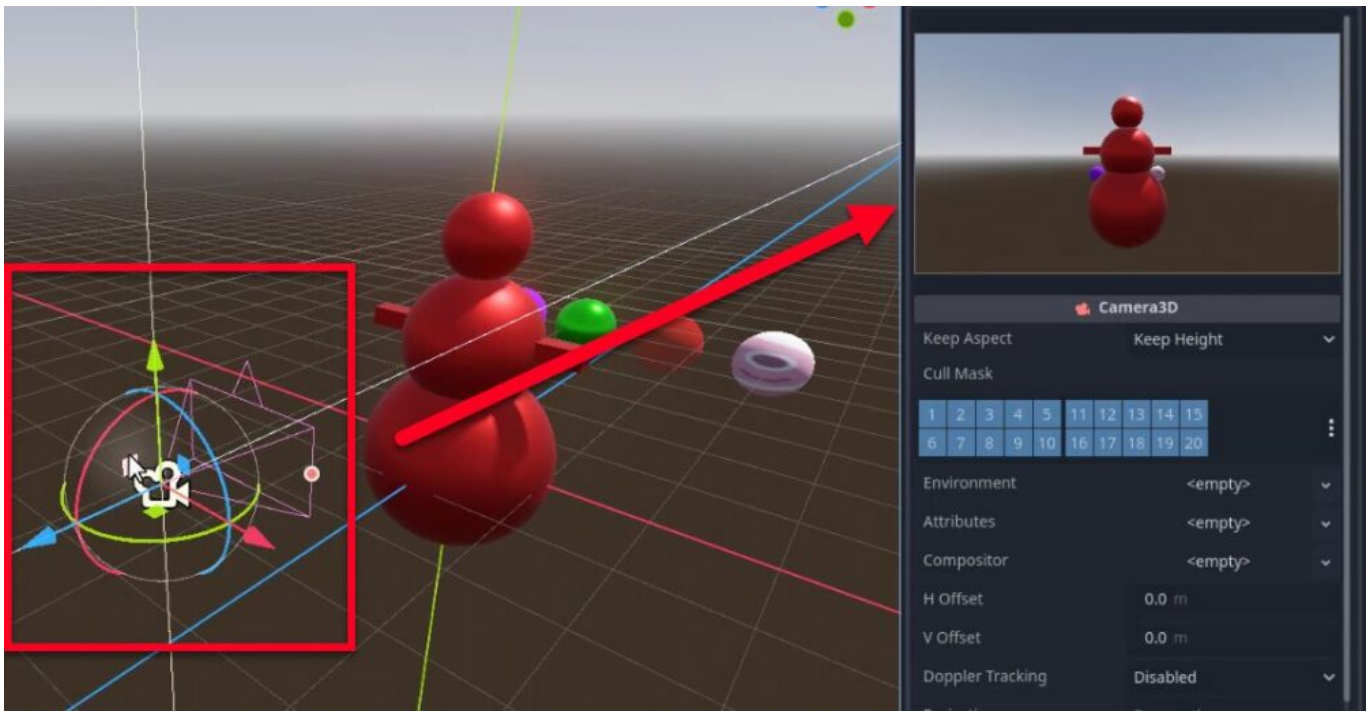
Search for “Camera3D”, select it, and click “Create” to add the Camera3D node to your scene.



Once the camera is added, you will see a purple box representing the camera's angle and direction.

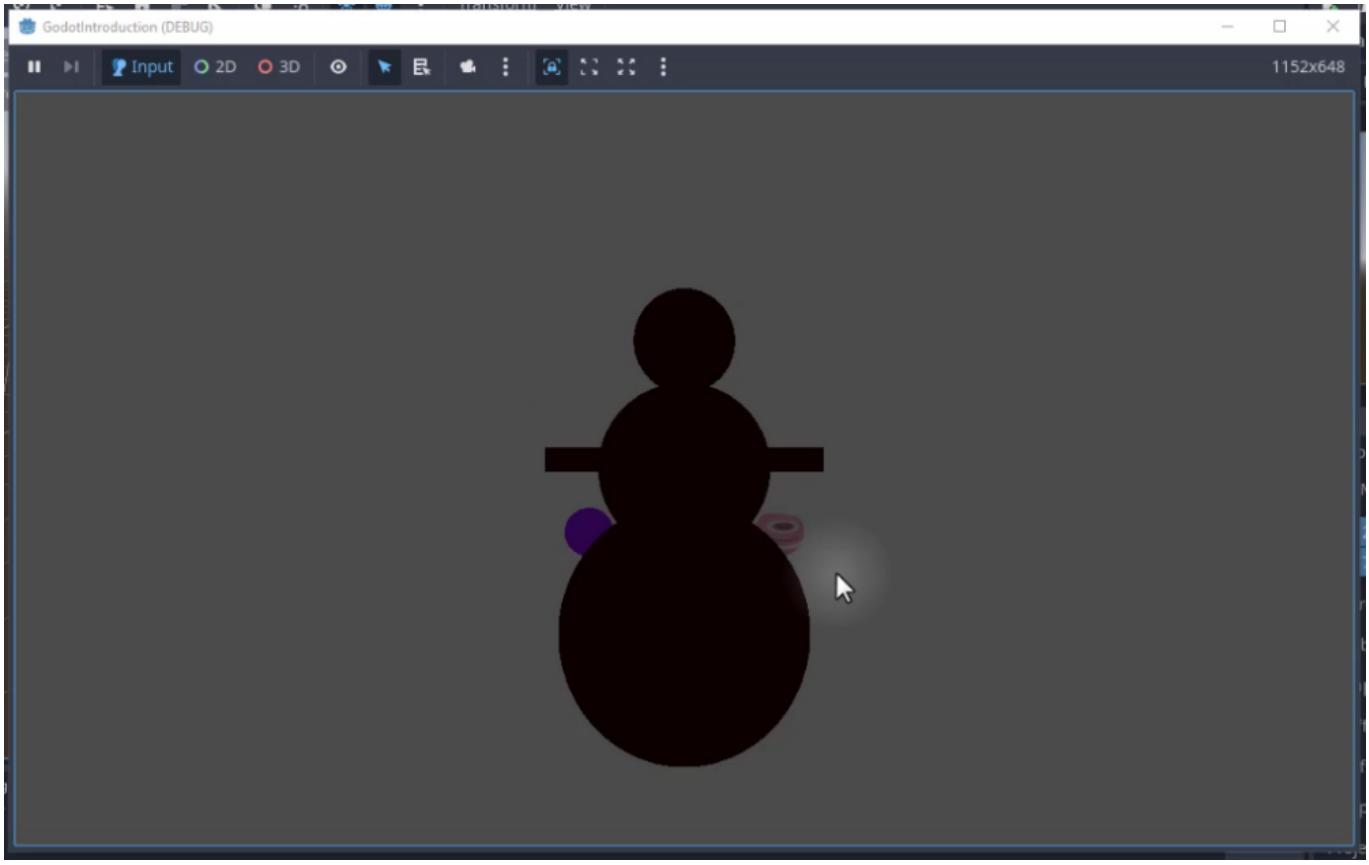


You can adjust the camera’s position and rotation like any object – checking the Inspector to achieve the desired viewport.

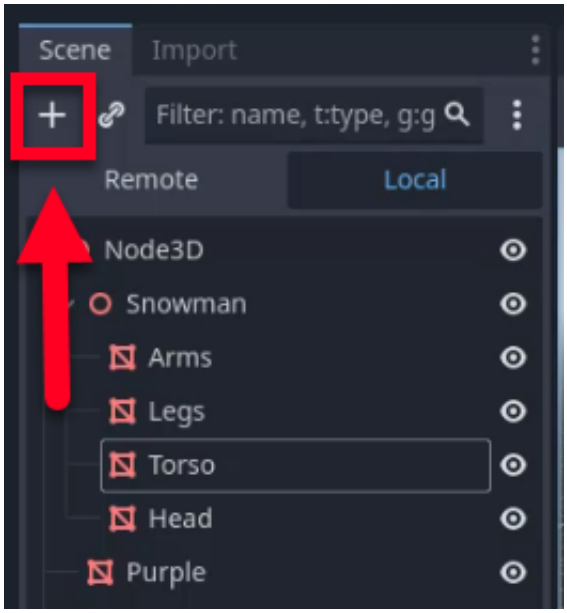


Adding Lighting and Shadows

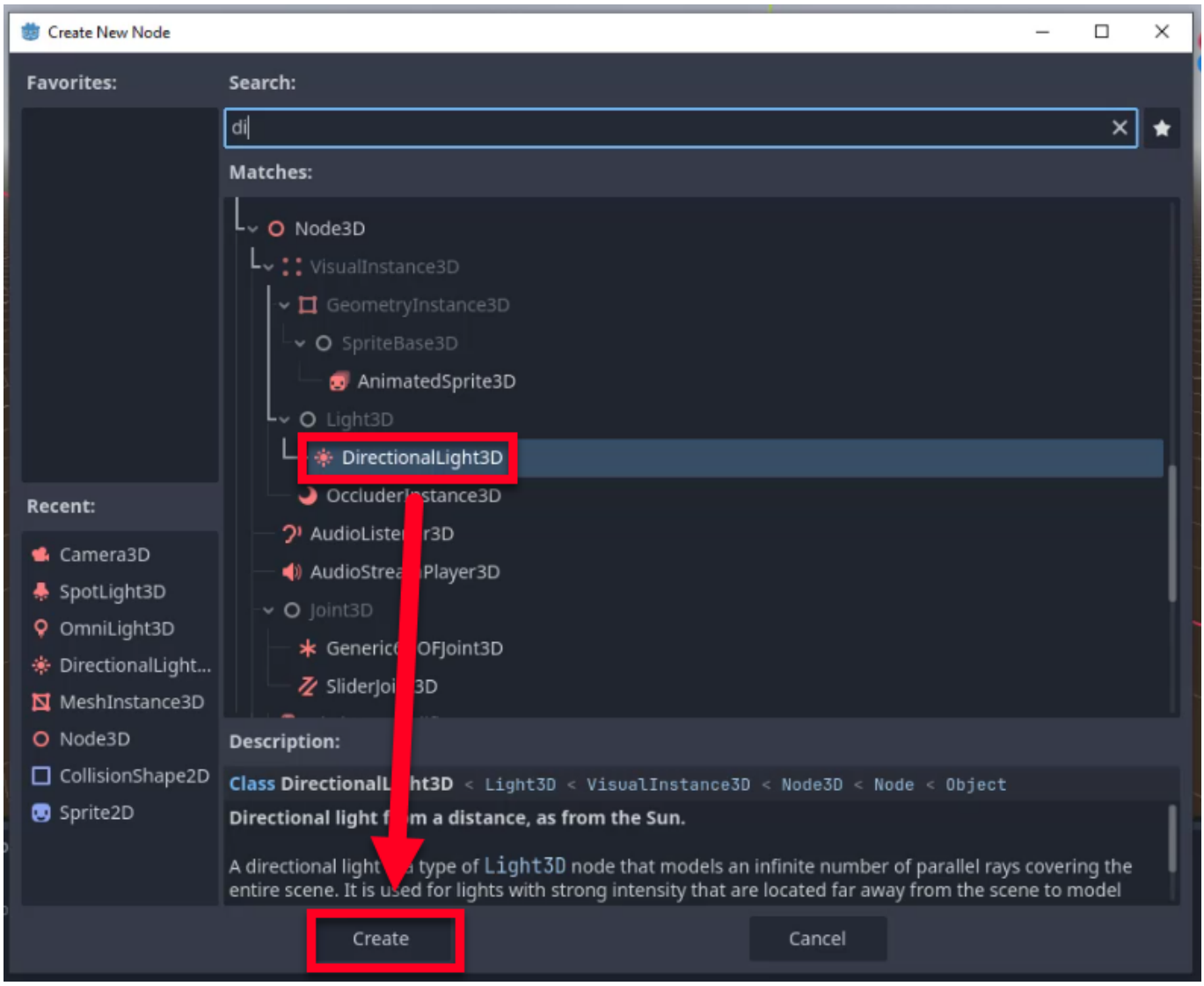
If you were to run this scene right this second, you would see it looks nothing like it does in the editor.



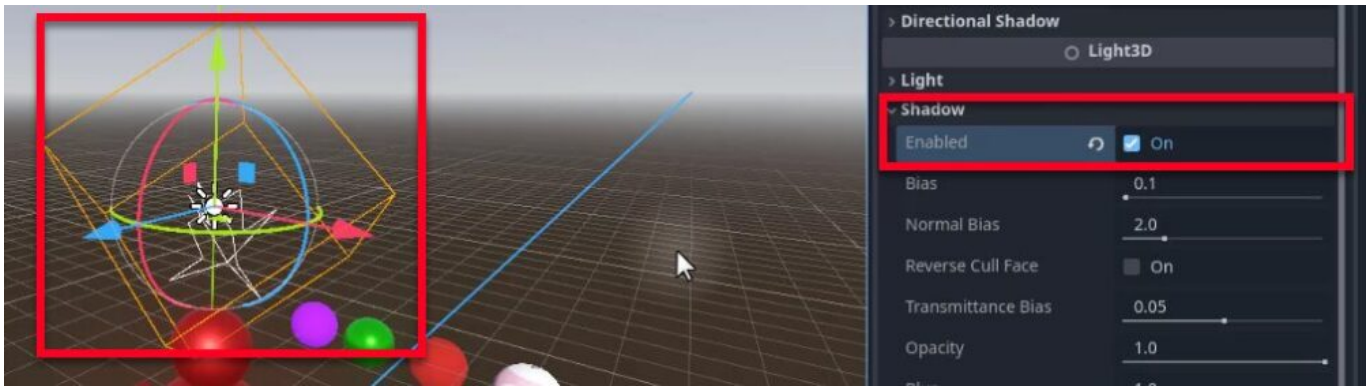
Remember from our previous lessons on lighting that the Godot adds a default light to the editor that is not actually in our game scene. We have to manually add a light in order to light up our scene. Let’s do this by creating a new child node of the scene.



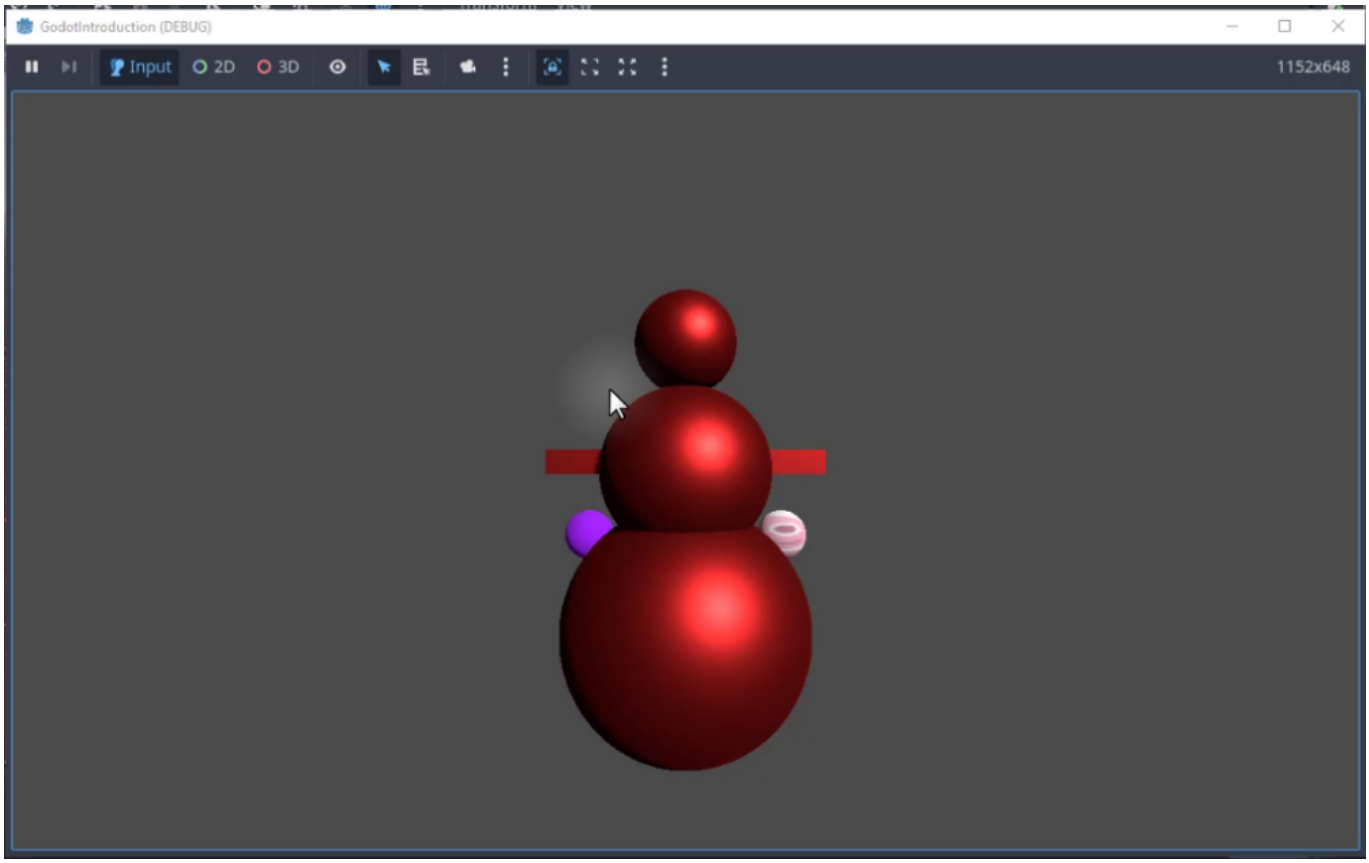
We will then add a DirectionalLight3D node to the scene.



Adjust the light's direction and enable shadows if needed.

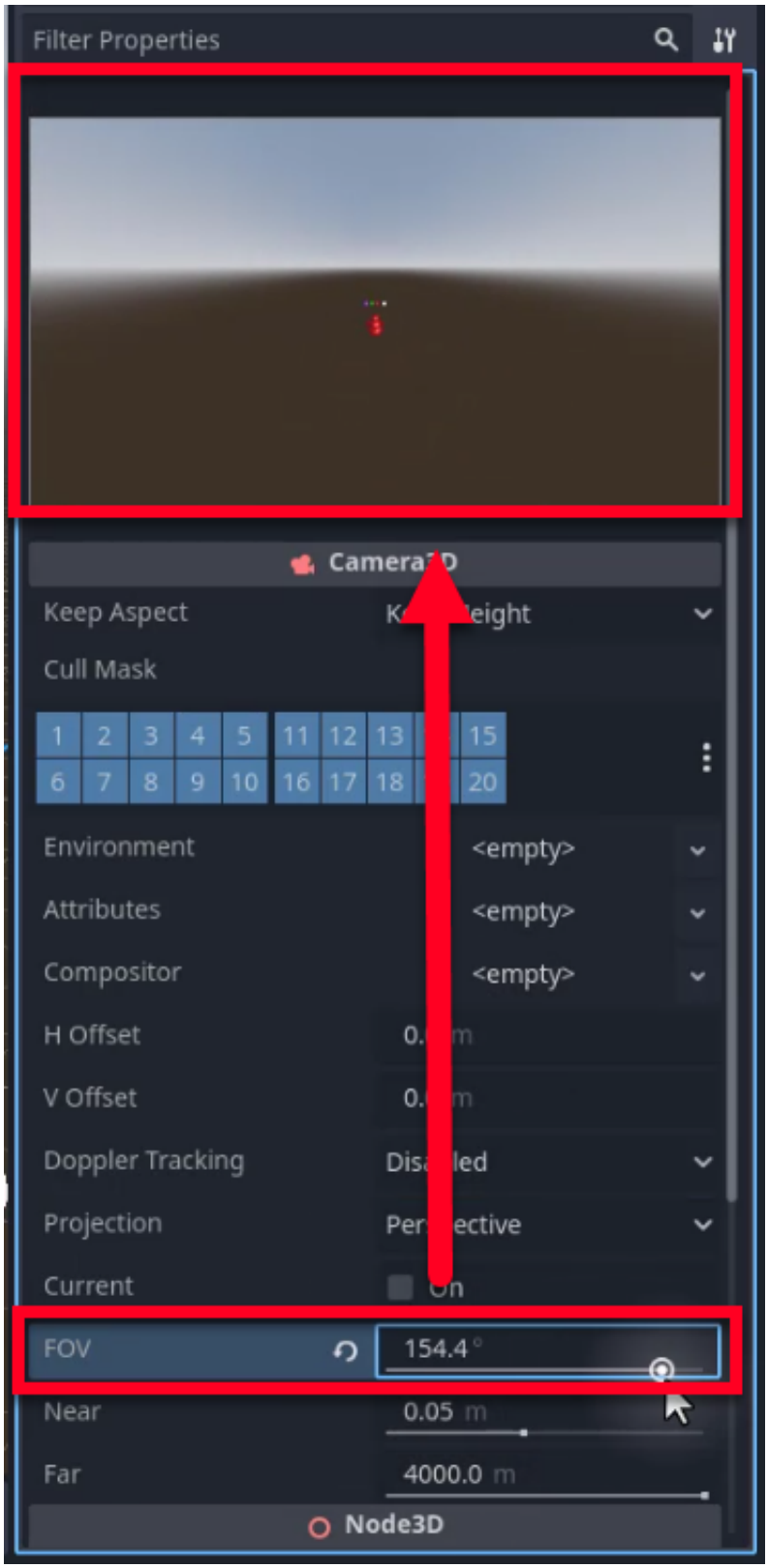


This will provide better visibility and depth to your 3D scene.



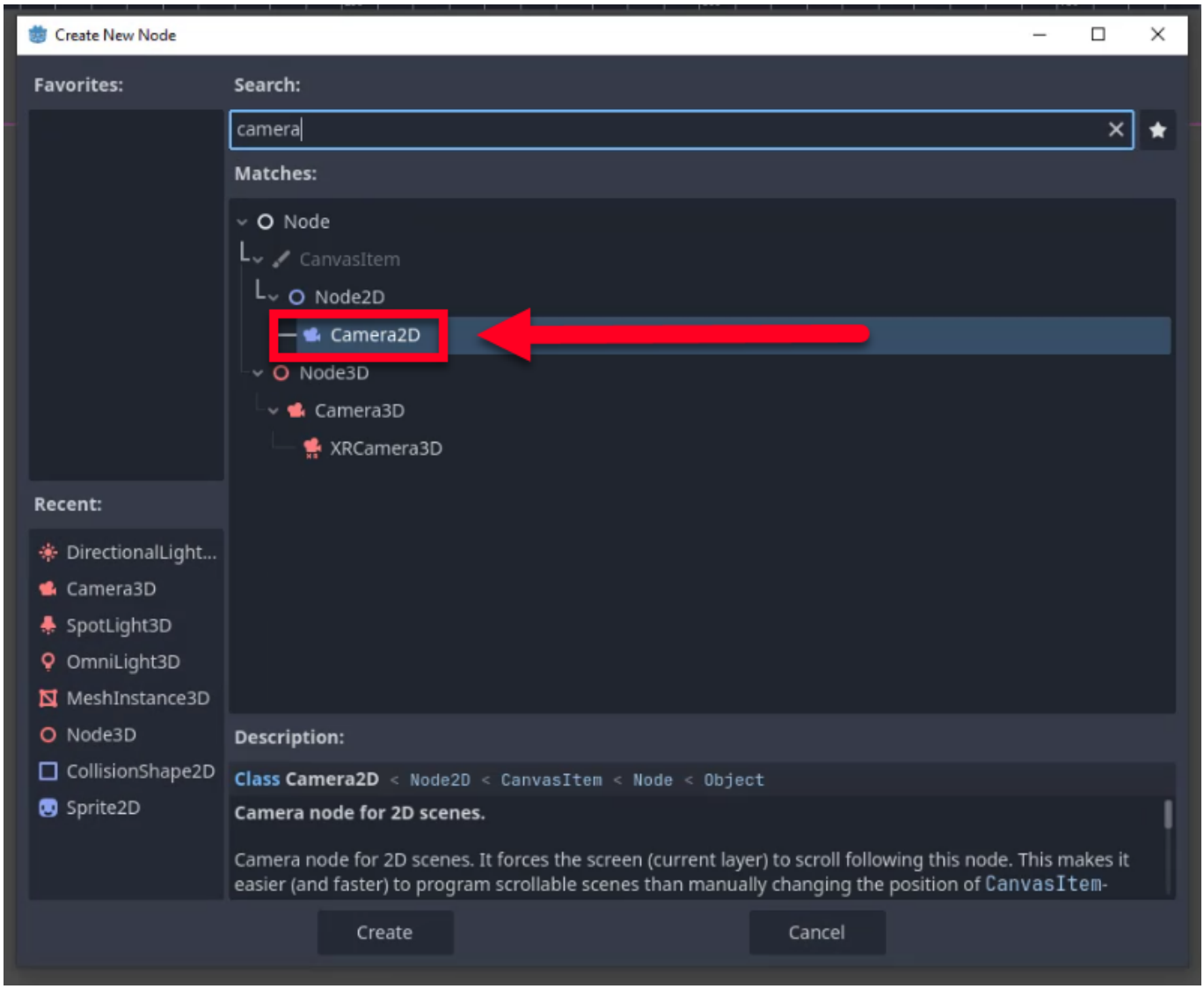
Adjusting the Camera View

You can modify several properties to customize the camera view in the Inspector. One of the most important ones to keep in mind, though, is **Field of View (FOV)**. FOV determines the camera's capture angle. The default is 75 degrees, but you can increase or decrease it to zoom in or out of the scene.

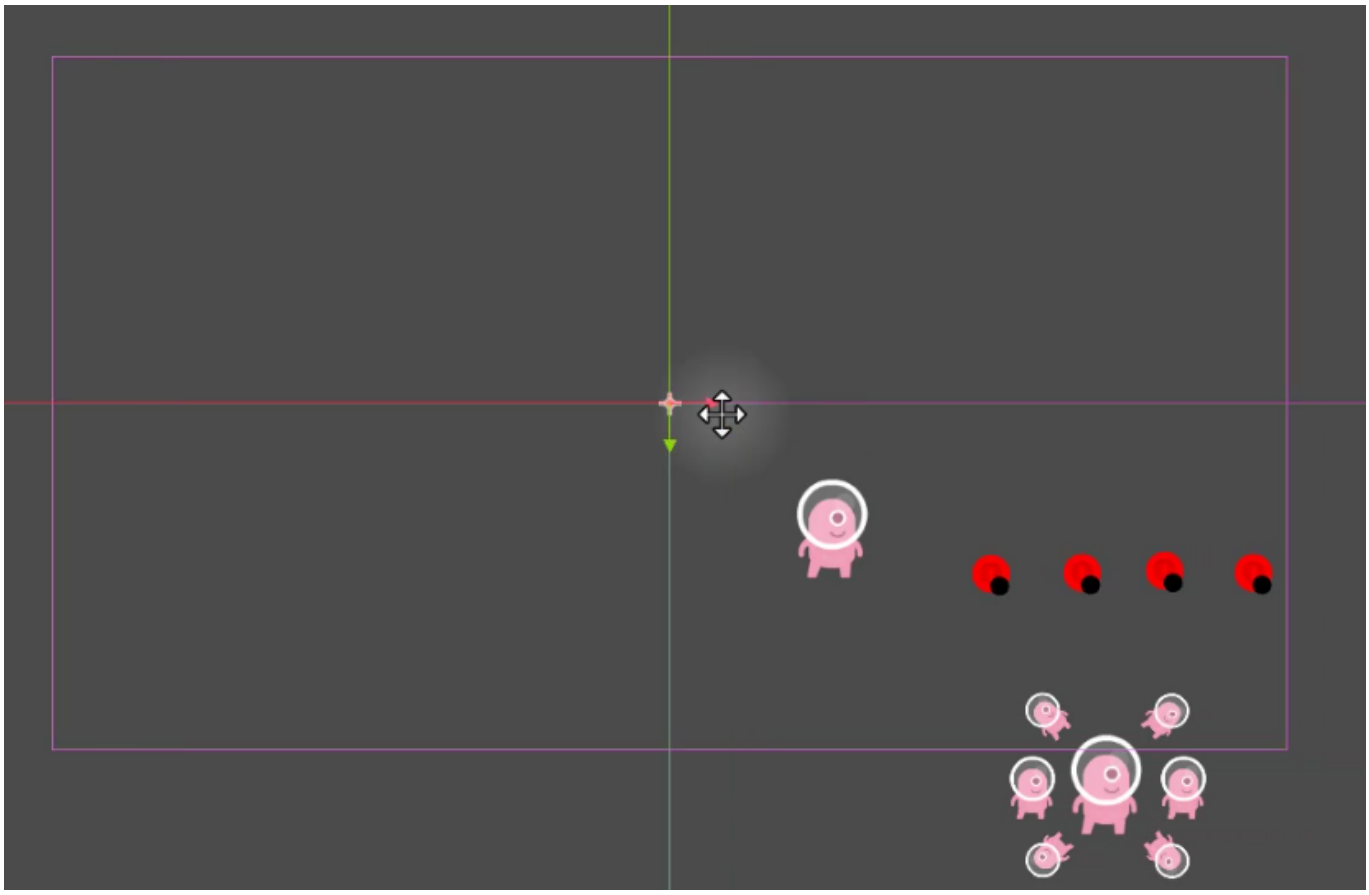


Setting Up a 2D Camera

Up until now, we've been talking strictly about the Camera3D node for 3D projects. For 2D projects, the process to setting up a camera is similar but with a Camera2D node.

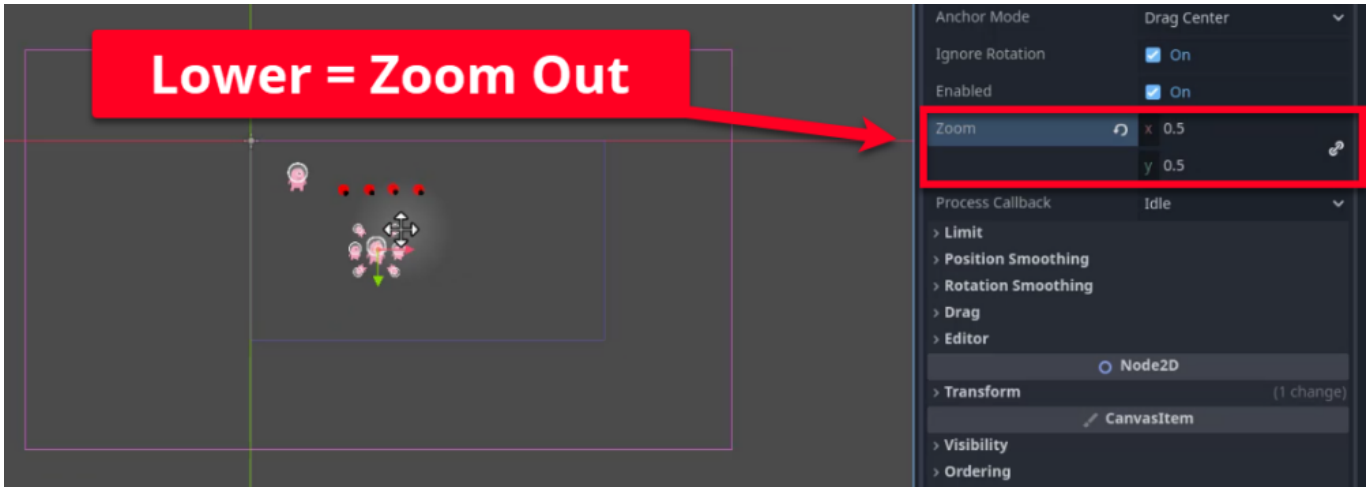


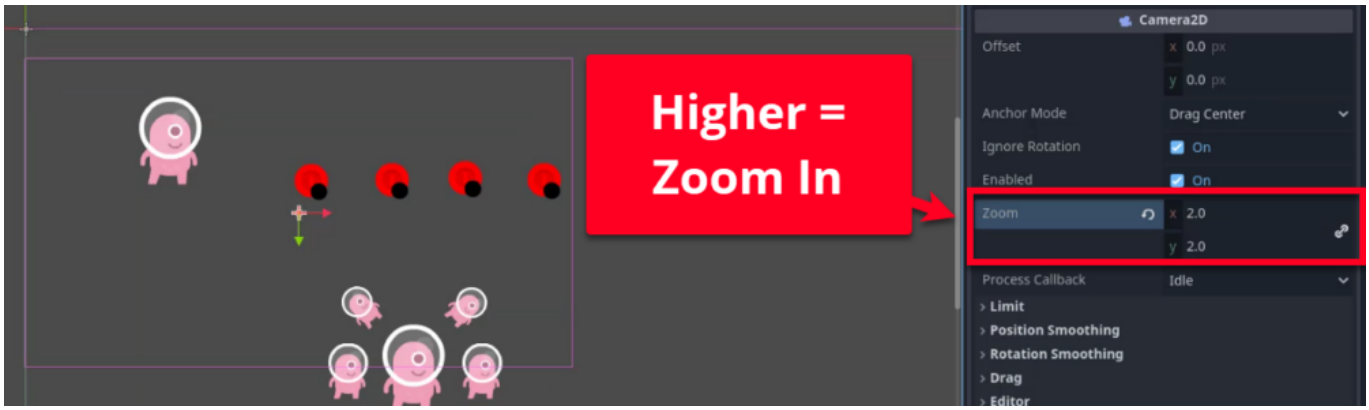
The Camera2D node is represented by a large purple box that you can move and position within your 2D scene.



Adjusting the 2D Camera View

In 2D, you can zoom in and out using the **Zoom Property**. Setting the zoom level controls how close or far the camera view is. A value of 1 means 1 pixel in Godot units represents 1 pixel in the game window. Lower values than 1 zoom out the camera, while higher values zoom in.





Experimenting with Cameras

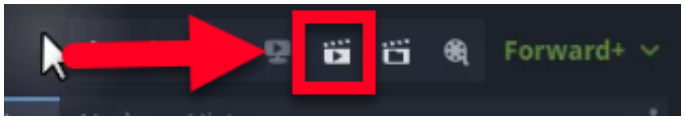
Feel free to experiment with both 3D and 2D cameras to get comfortable with positioning and adjusting them. In 3D, try rotating the camera and changing the field of view to achieve the perfect perspective for your game.

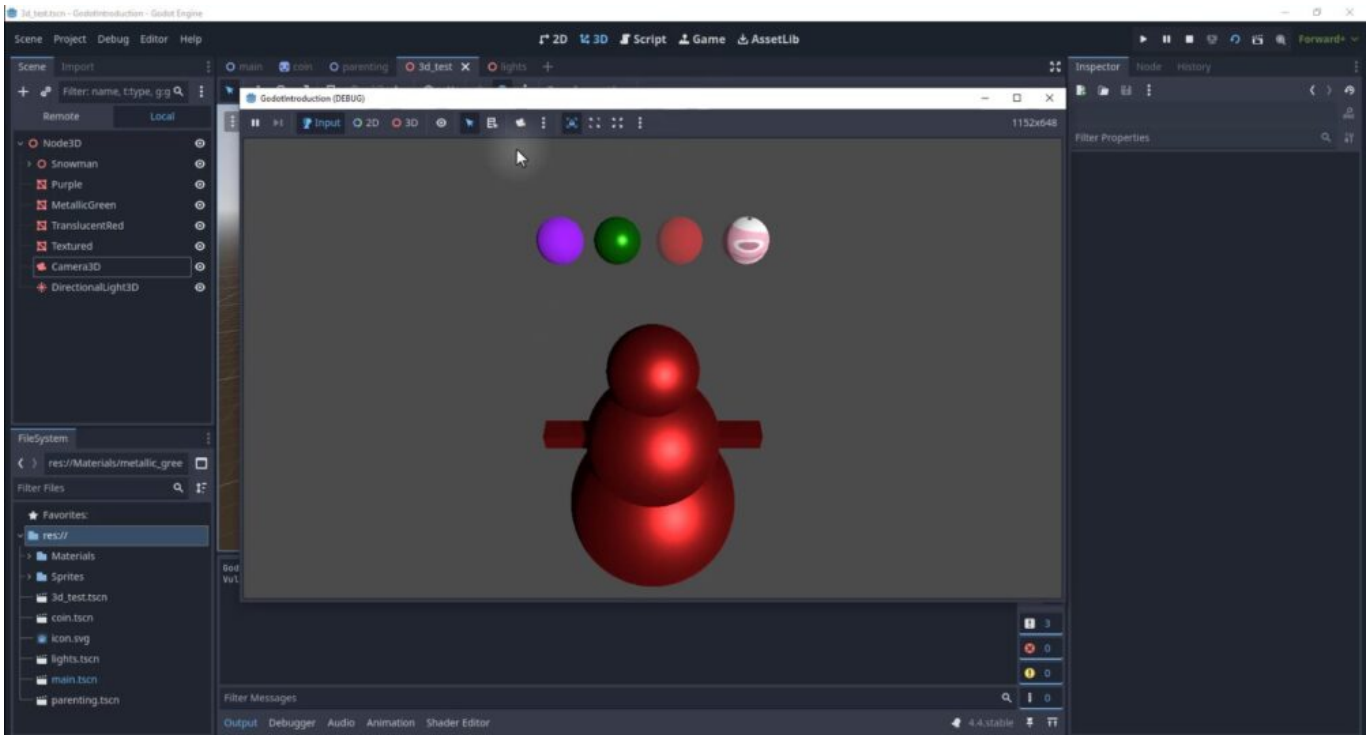
By following these steps and tips, you'll be well on your way to mastering camera setup in Godot. Happy game development!

Welcome to this lesson, where we will explore the game window in Godot and its various features. The game window is a crucial tool for running and testing your game within the Godot editor. By the end of this article, you will understand how to embed the game window, use different modes for input and editing, and utilize debugging tools.

Embedding the Game Window

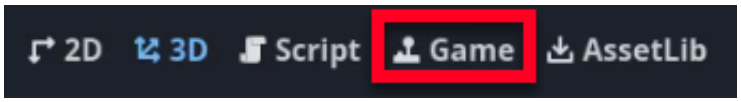
When you run your current scene, Godot opens a separate window to display your game.



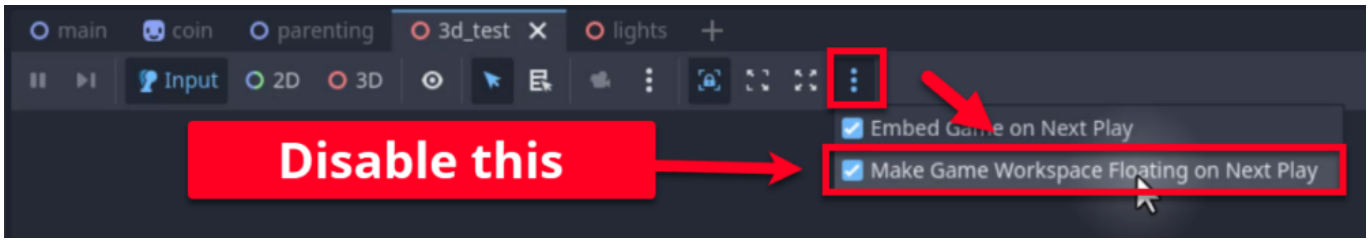


This can be inconvenient as the game window may overlap with the editor, making it difficult to navigate. To resolve this, you can embed the game window into the editor itself. To do this, stop the currently running game.

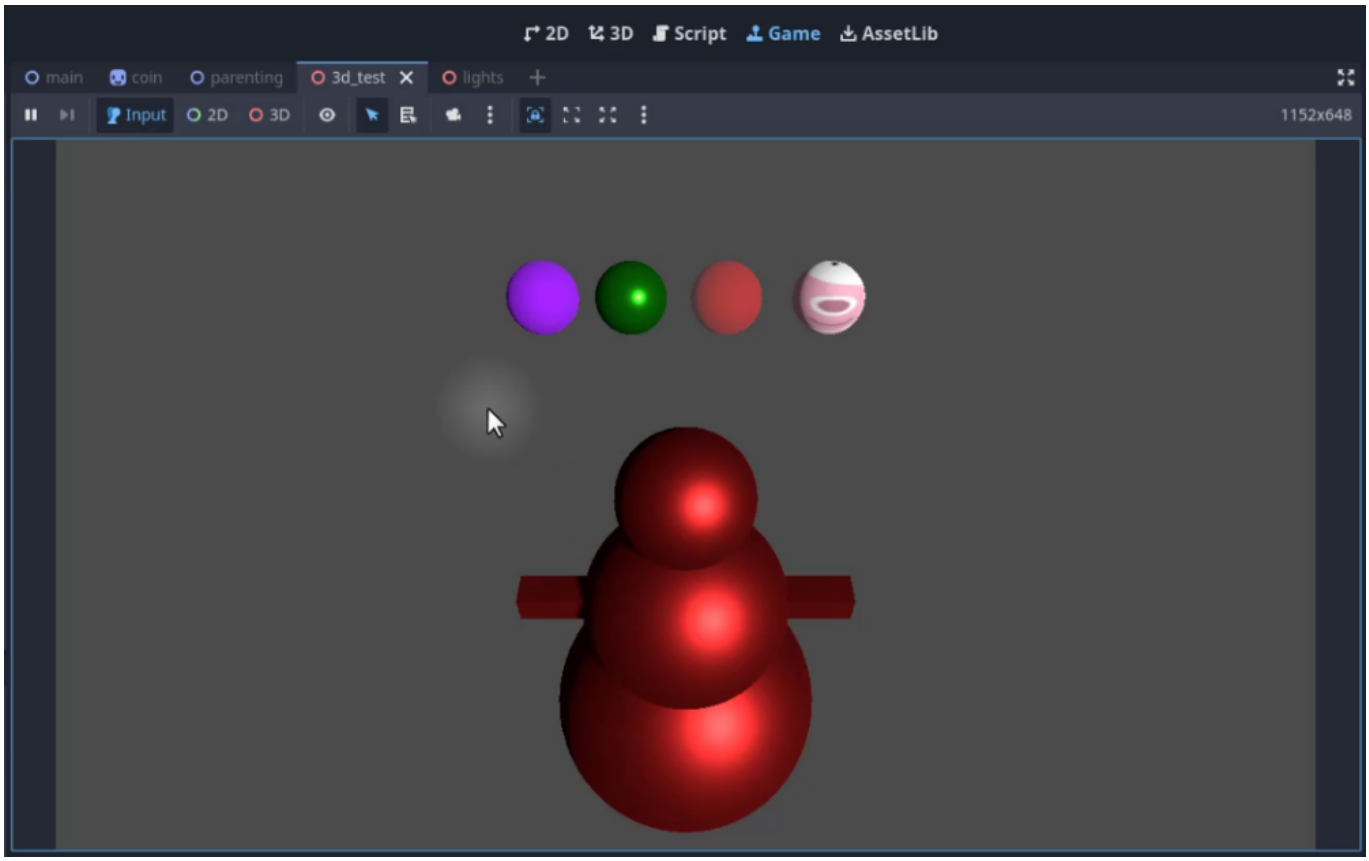
Once stopped, click on the **Game** tab at the top of the editor to open the game dock window.



In the game window, click on the three dots on the far right to open the embedding options. Disable the option **Make Game Workspace Floating on Next Play**.



Now, when you run your game, it will appear within the editor, allowing you to navigate and interact with the editor while your game is running.

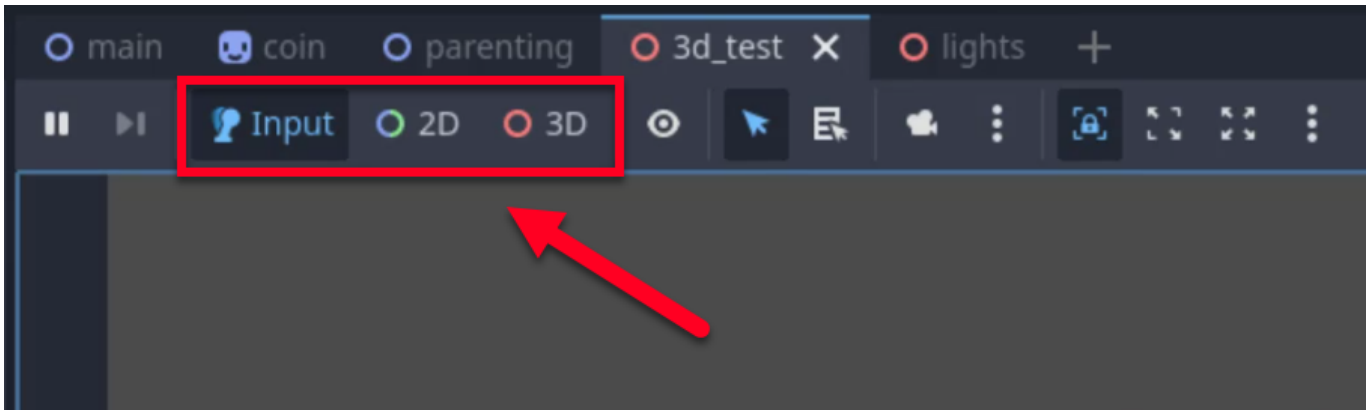


Game Window Modes

The game window offers different modes that allow you to interact with your game in various ways. These modes are:

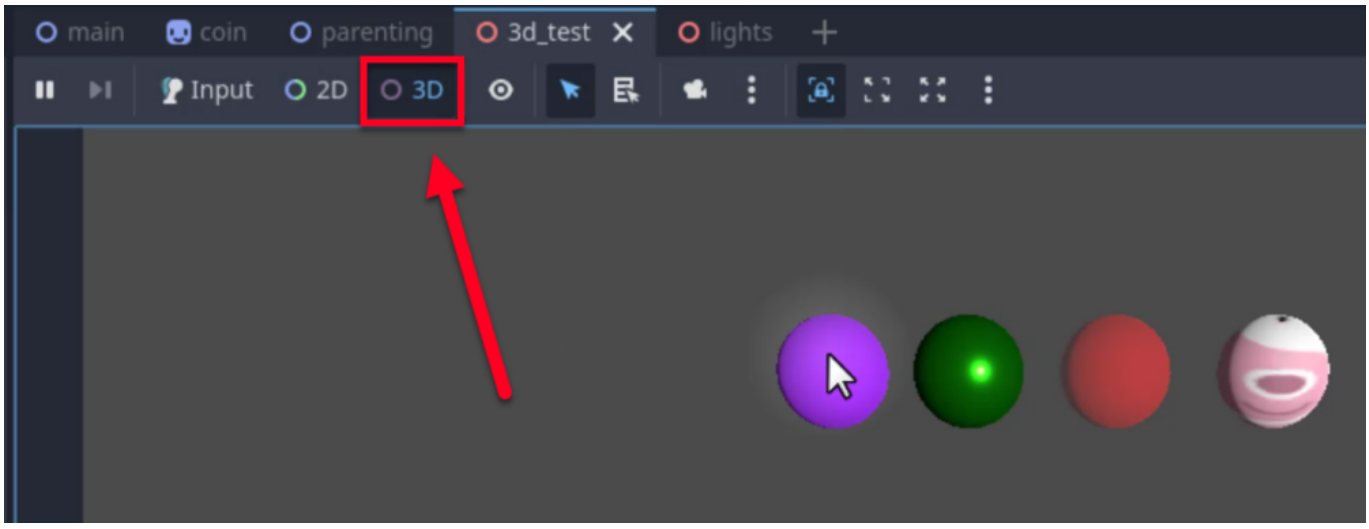
- **Input**: This is the default mode that allows you to use your keyboard and mouse to play the game.
- **2D**: This mode disables game input and allows you to select and manipulate 2D nodes and the 2D camera.
- **3D**: This mode disables game input and allows you to select and manipulate 3D nodes and the 3D camera.

To switch between these modes, click on the respective buttons (Input, 2D, 3D) at the top of the game window.

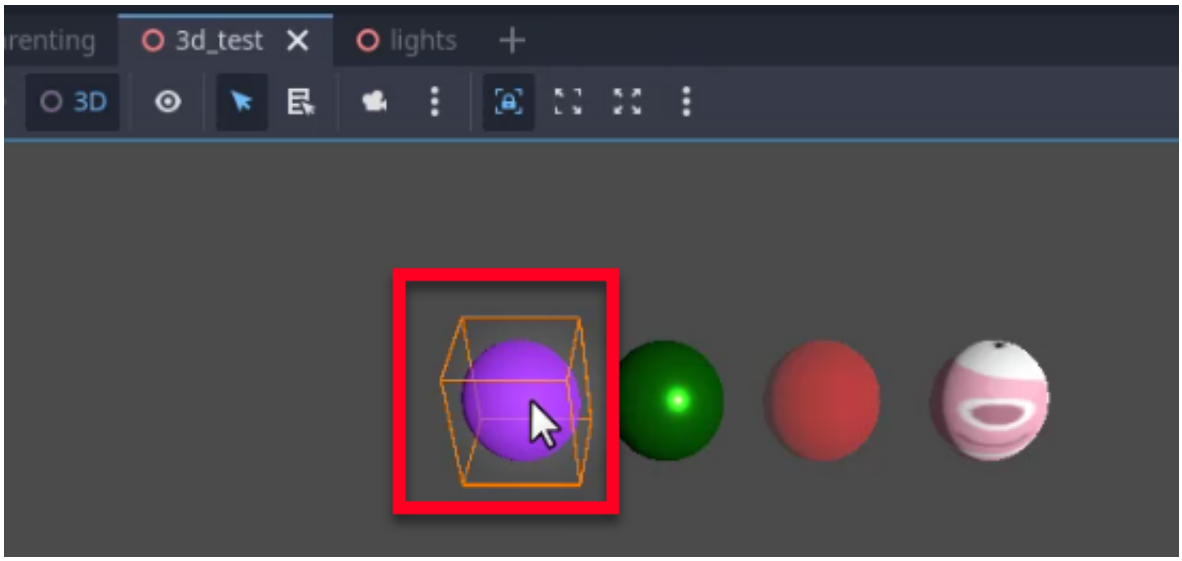


Editing Nodes in Game Window

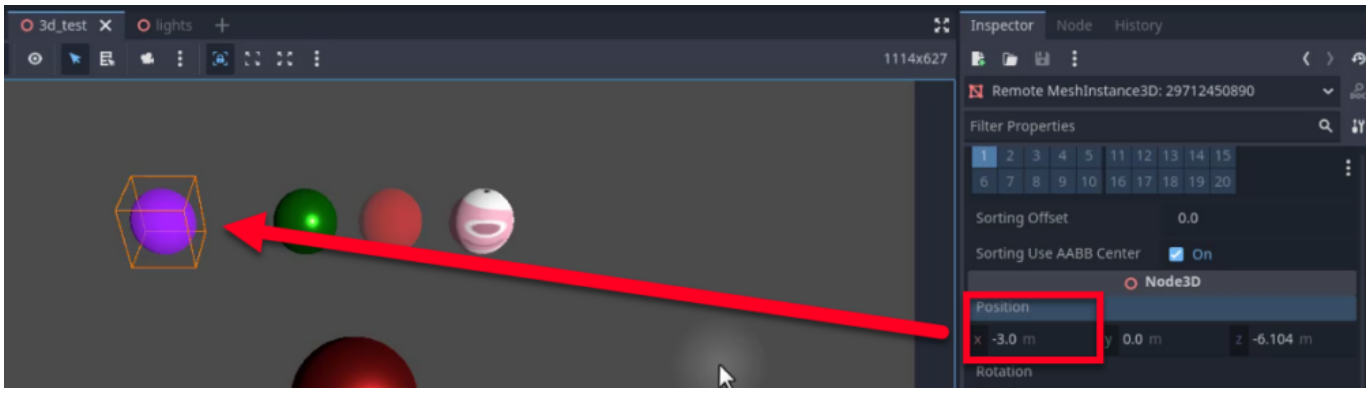
One of the powerful features of the game window is the ability to select and edit nodes while the game is running. To do this, first run your game and switch to the **3D** or **2D** mode depending on your scene. We're in our 3D scene, so we'll select 3D.



Click on the node you want to edit in the game window.



The properties of the selected node will appear in the Inspector, allowing you to modify them.

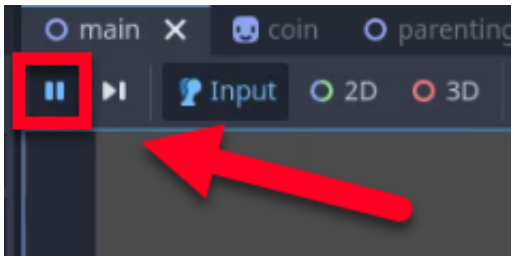


This is particularly useful for checking and adjusting node properties during gameplay without stopping the game. Note, however, that these changes do not save once you stop playing the game. You will need to note them down and reapply them if you want them to stay. This is why it's best to refine values slowly over several iterations!

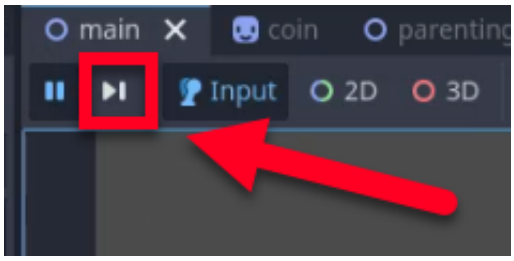
Debugging Tools

The game window also provides debugging tools to help you identify and fix issues in your game:

- **Pause**: Click the pause button at the top left corner to pause the game. This is useful for checking the state of the game at a specific moment.



- **Frame-by-Frame**: Click the button to the right of the pause button to advance the game frame by frame. This can help you pinpoint where something went wrong.



These tools are essential for debugging and ensuring your game runs smoothly.

Conclusion

The game window in Godot is a versatile tool that enhances your development workflow. By embedding the game window, using different modes, editing nodes during gameplay, and utilizing debugging tools, you can efficiently test and refine your game. Happy game development!

Welcome to your introduction to scripting in Godot! In this lesson, we will explore what scripting is and how it can be used to create gameplay behaviors and logic for our games. Scripting is essentially coding, where you write specific instructions to control various aspects of your game.

What is Scripting?

Scripting is the process of writing code to create gameplay behaviors and logic. A script is a file that contains this code, similar to a text document. These scripts can be attached to nodes in Godot to control their functionality.

Understanding Scripts and Nodes

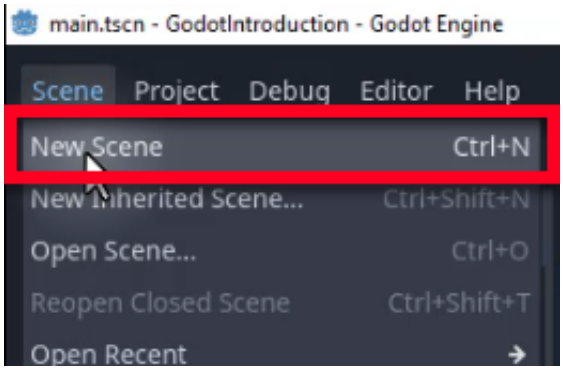
- **Script:** A file containing code that defines behaviors and logic.
- **Node:** An object in Godot that can have a script attached to it.

For example:

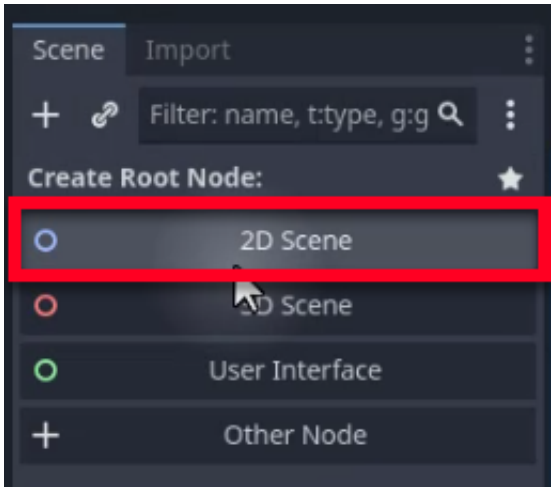
- A player script attached to a player node can control the player's movement.
- A rotate script attached to a windmill node can make the windmill rotate.

Creating a New Scene

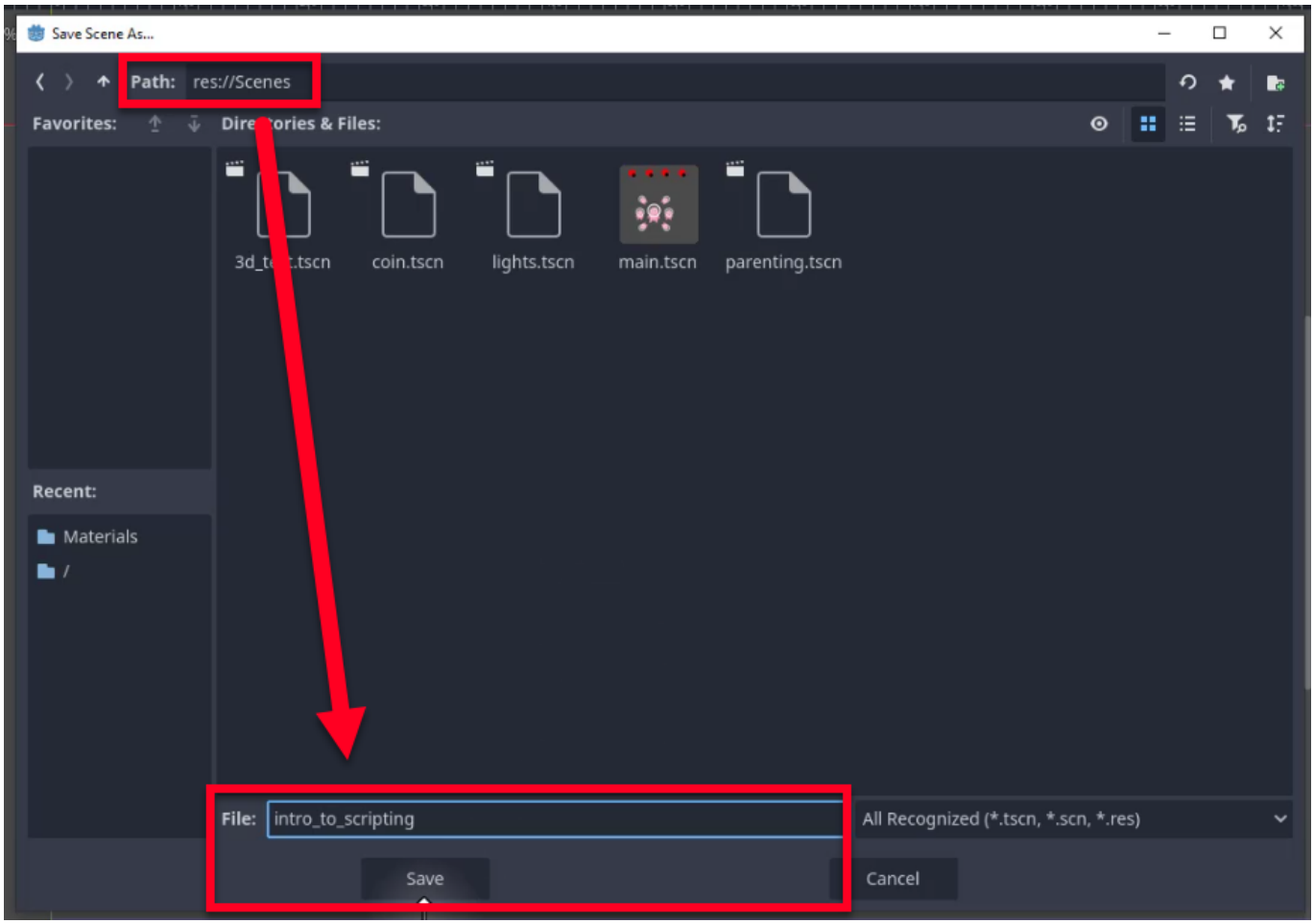
Let's walk through the steps to create a script in Godot. However, we'll first create a new scene so we can have a fresh slate. From the Scene menu, select **New Scene**.



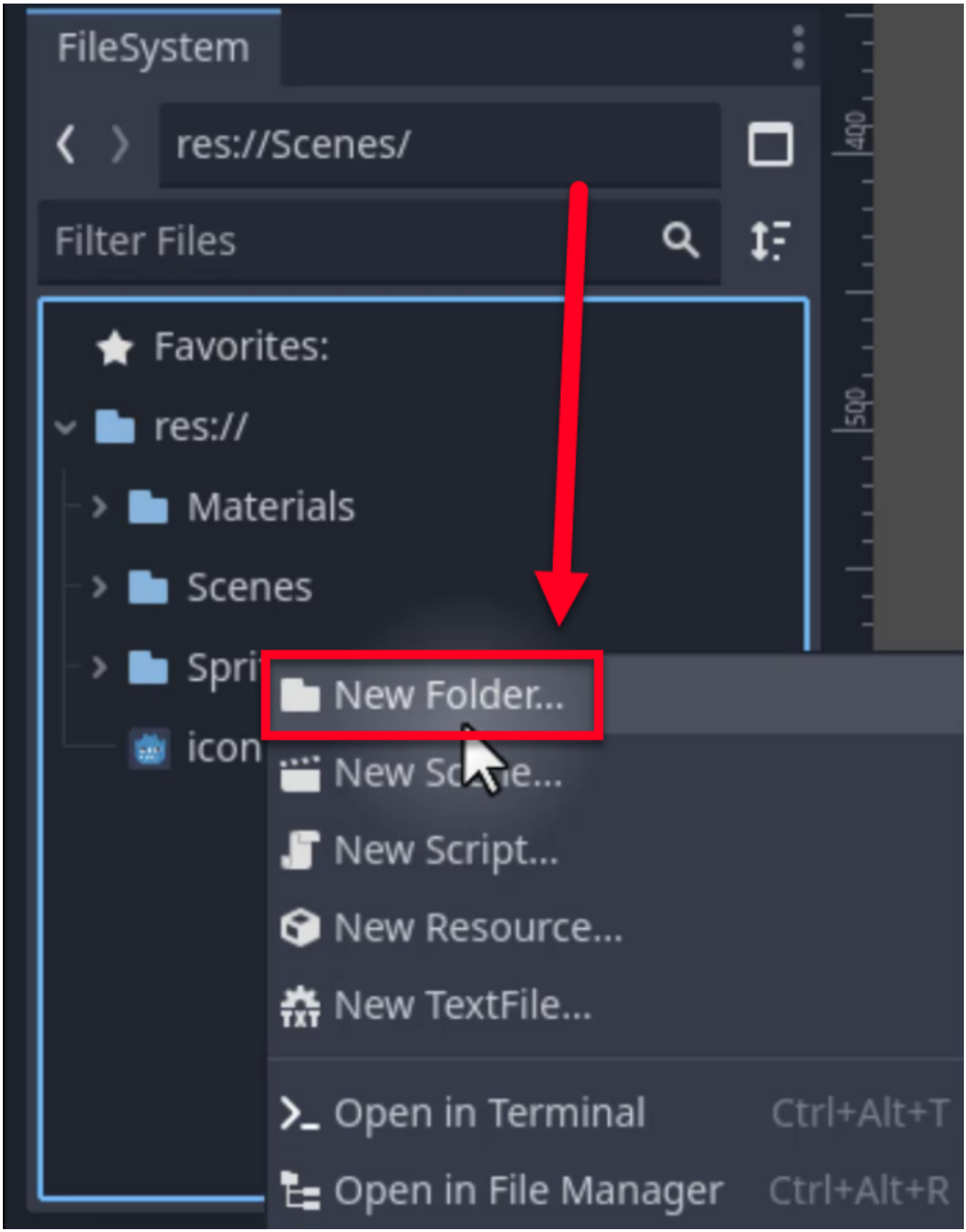
Select 2D Scene as the root node.



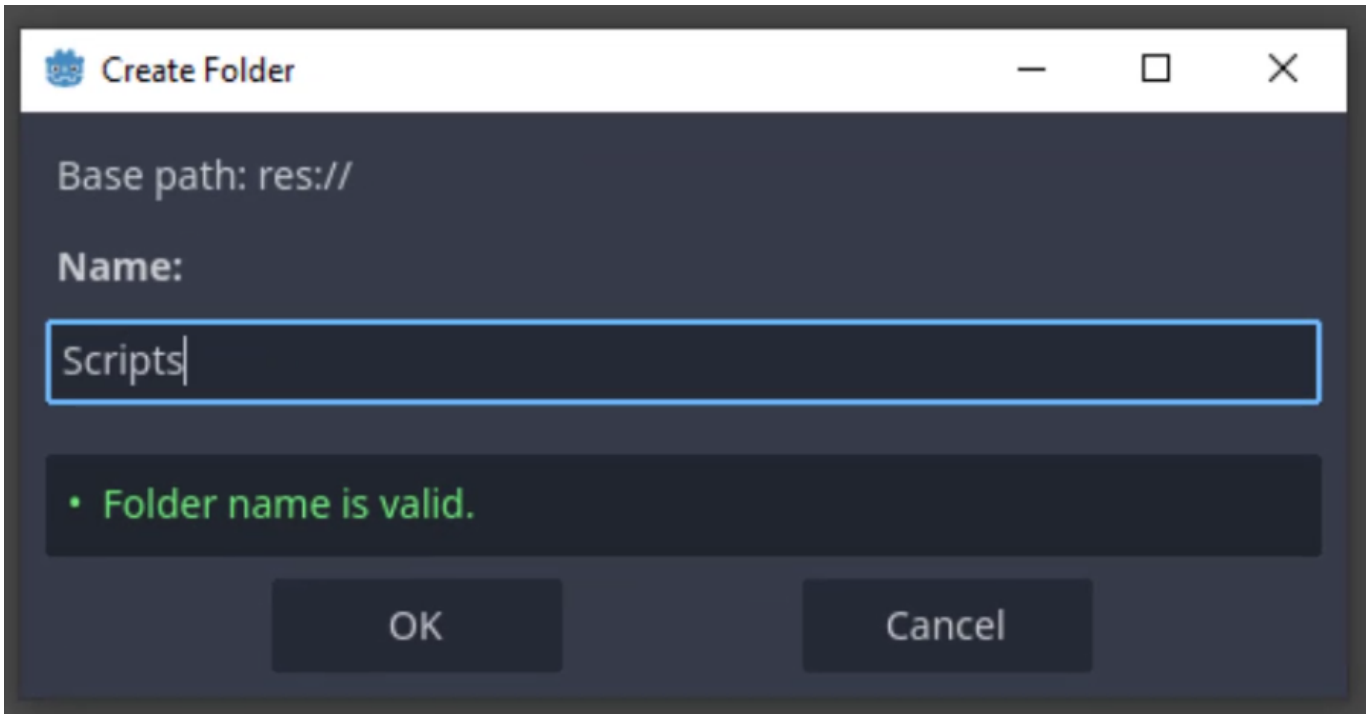
Save the scene in a designated folder, e.g., scenes.



Before we create a script, let's create a new folder to store it. Right-click on your FileSystem and select *New Folder*.

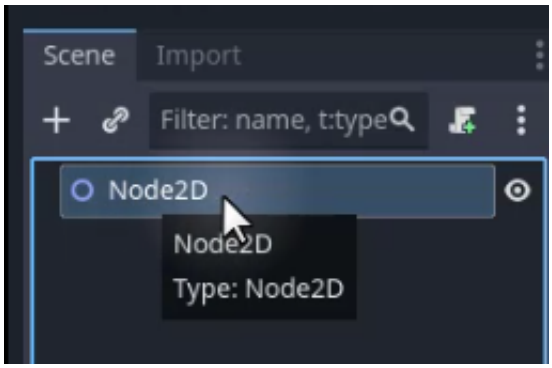


Create the folder, naming it something like “Scripts”.

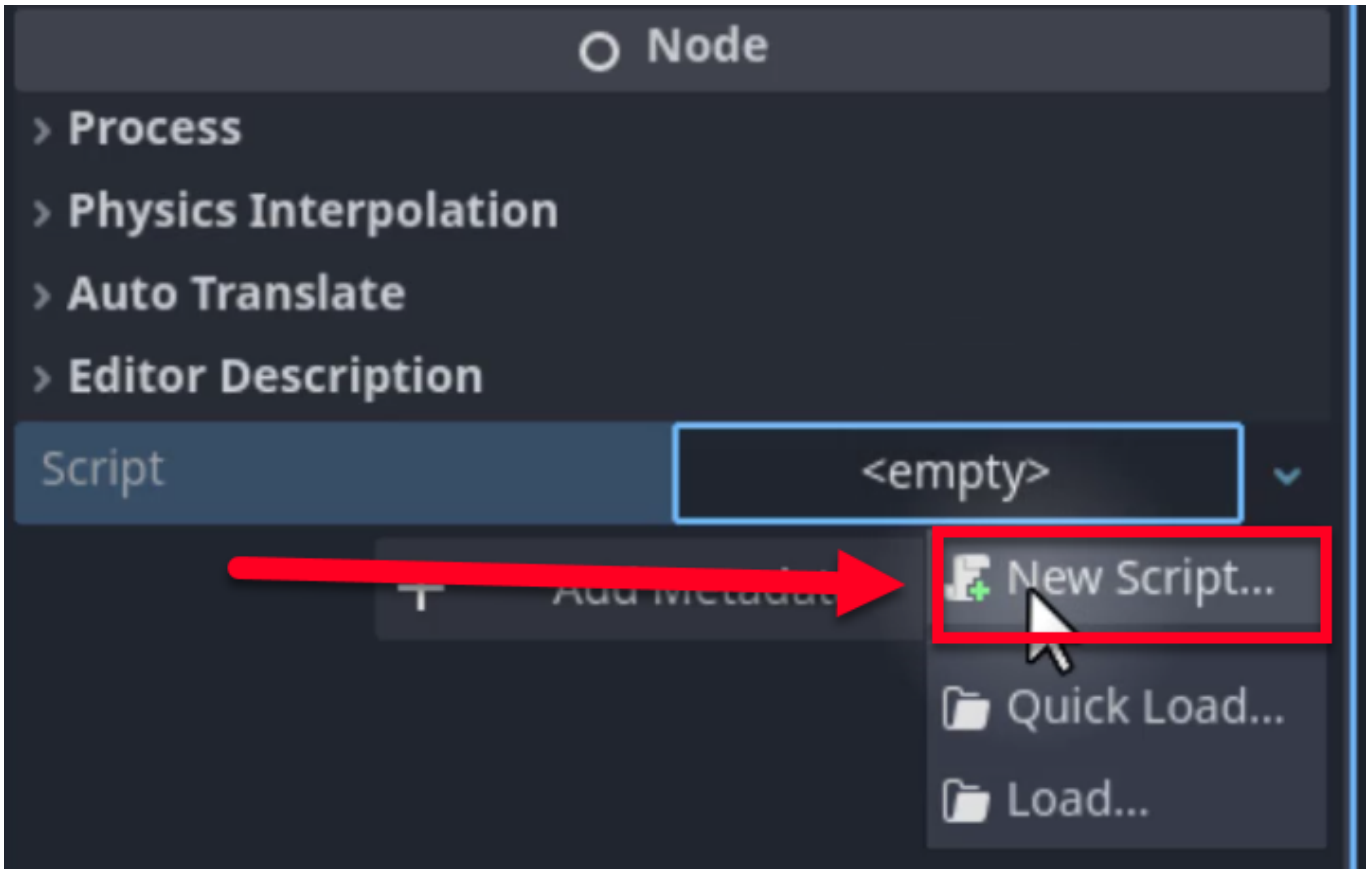


Creating a Script in Godot

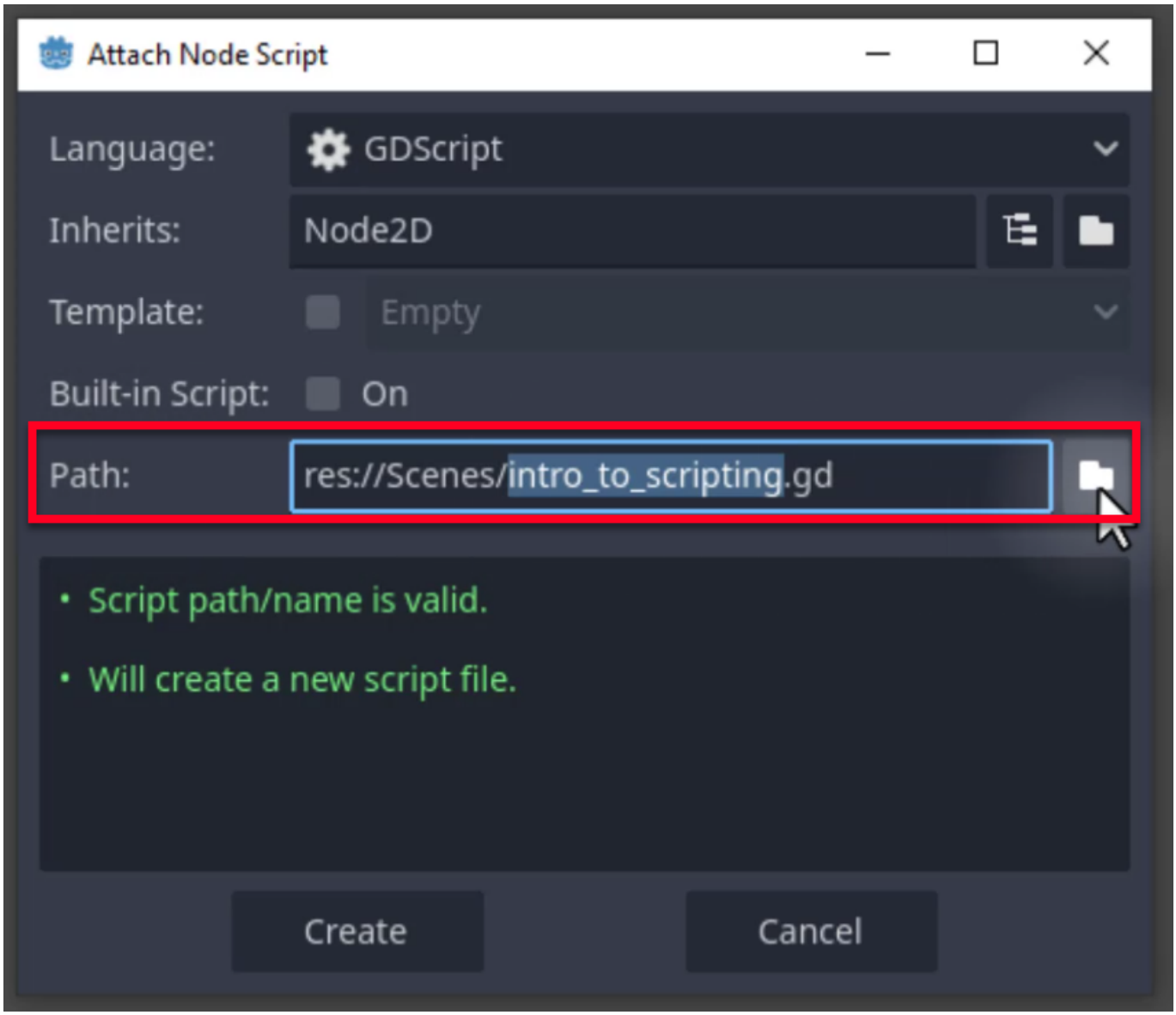
We can now create our first script! First, select the root node in your scene.

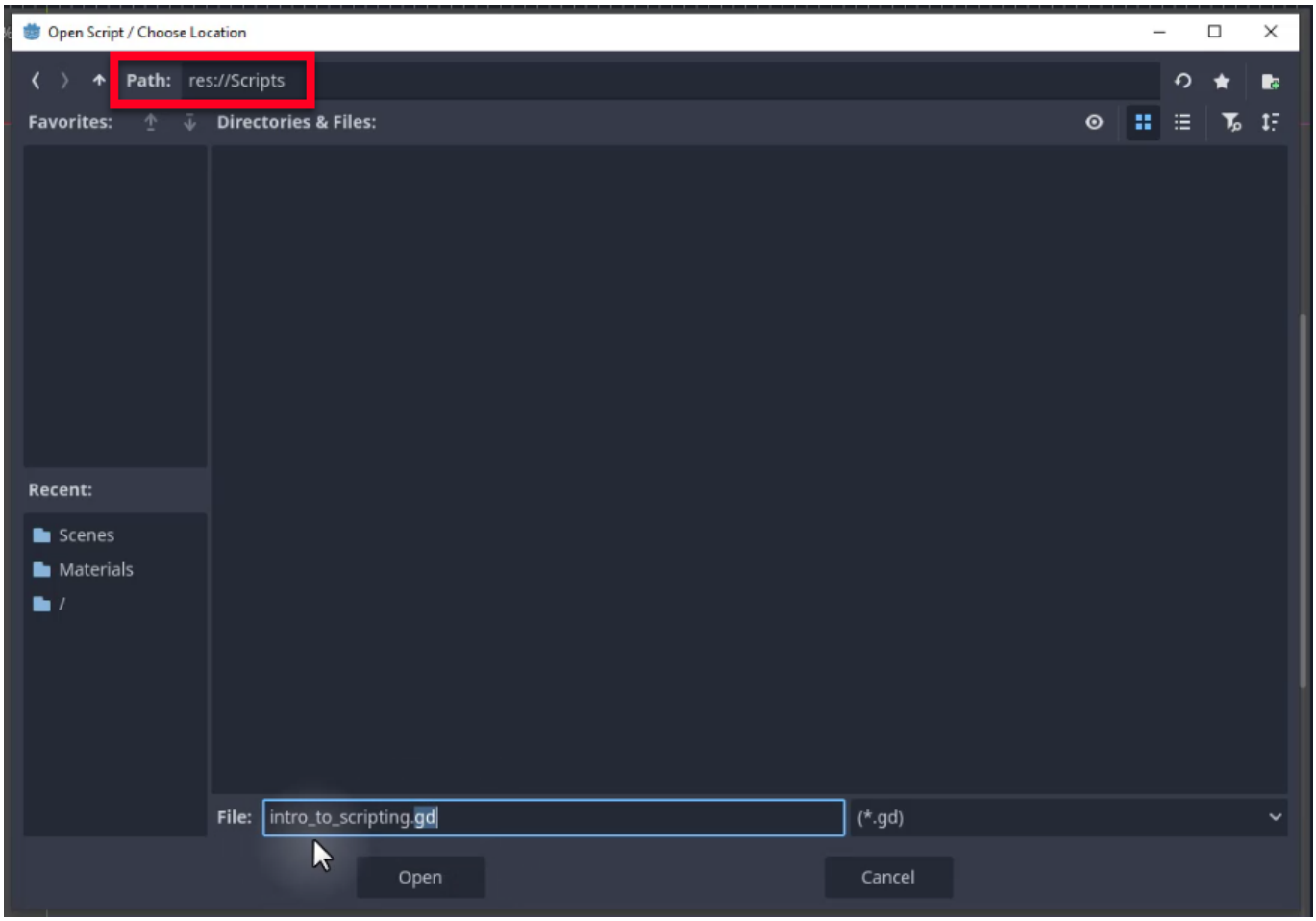


In the Inspector, find the Script property, click on the empty field, and click New Script.



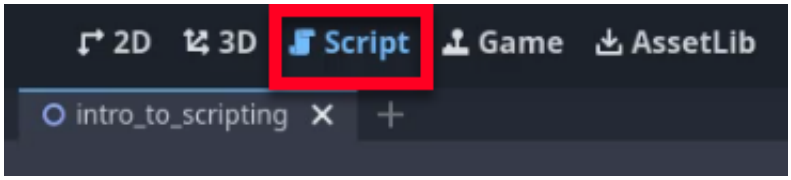
Provide a name and path for your script. In this case, choose the Scripts folder and name your script, e.g., intro_to_scripting.gd. Click Create to generate the script file once ready.





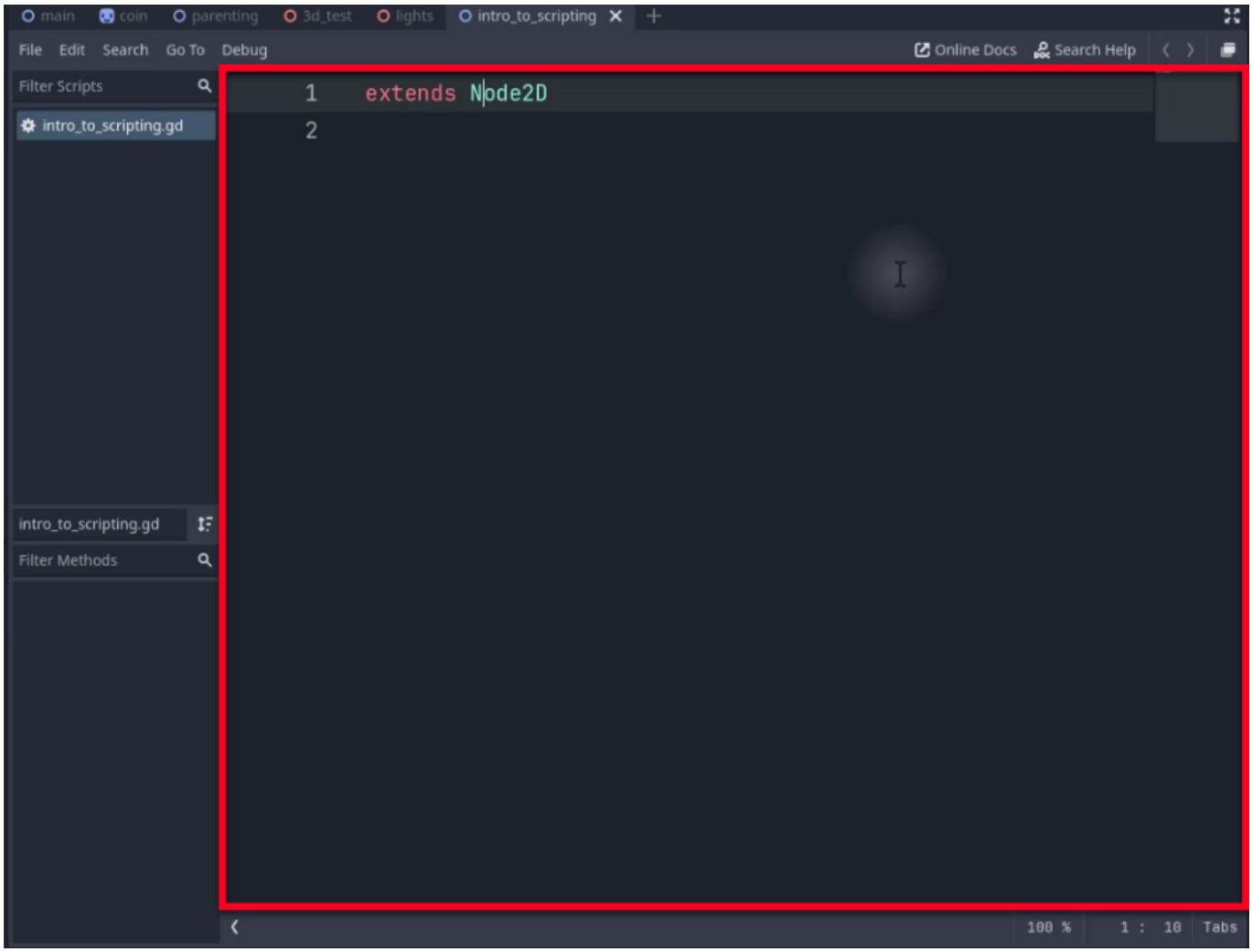
The Scripting Window

The scripting window is where you write and edit your code.

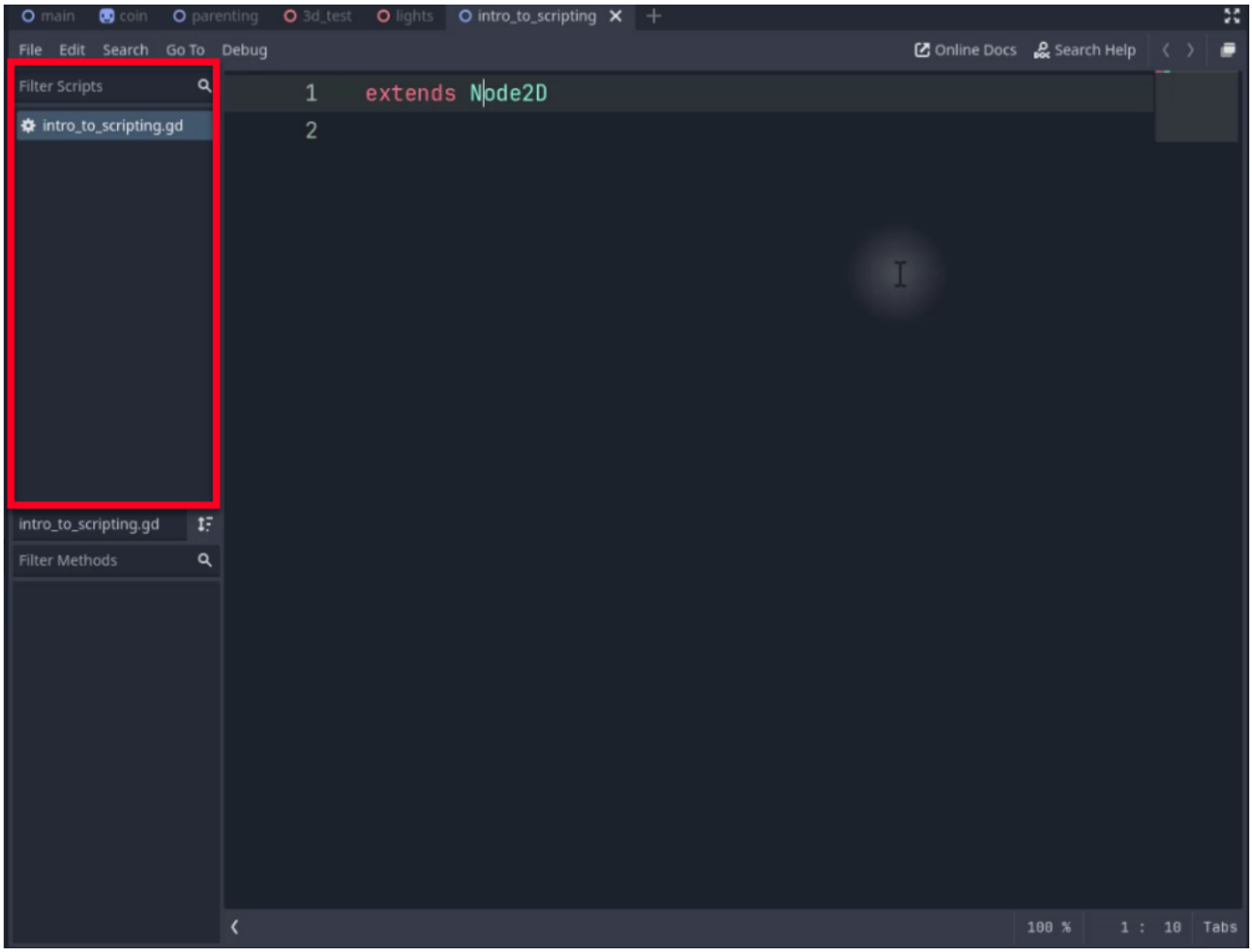


Here are some key features:

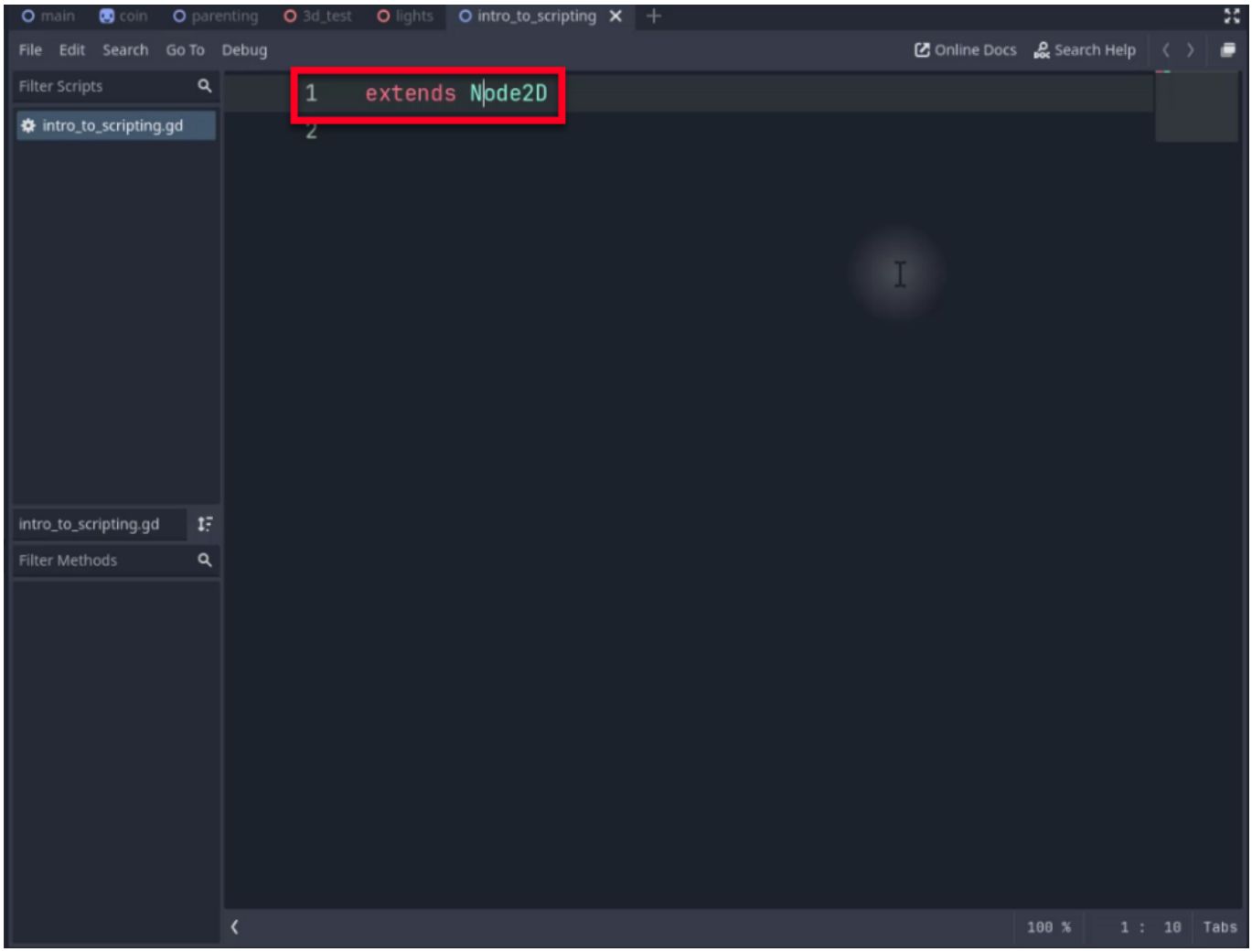
- **Text Editor:** The main area where you write your code.



- **Script List:** A list of all your scripts for easy navigation.



- **Code Highlighting:** Automatic highlighting and formatting of your code for better readability.



- **Autocomplete:** Suggestions for code completion as you type.

GDScript: The Programming Language

GDScript is the programming language used in Godot. It is similar to Python and is designed to be easy to learn. Even if you have no prior coding experience, you can start with the basics and gradually build your understanding.

Next Steps

In the next lesson, we will dive into the concept of variables, which are fundamental to programming. Stay tuned and keep practicing!

Thank you for joining us on this journey into scripting with Godot. See you in the next lesson!

Welcome to this lesson on variables! Variables are a fundamental concept in programming that allow us to store, read, and modify data. Think of a variable as a box where you can store a value, such as a number or text. You can give this box a name and change its contents as needed. Let's dive into how variables work and how to create them in Godot.

Understanding Variables

A variable is a piece of data that you can define, read, and change. Here are some key points about variables:

- Variables store data in the computer's memory.
- You can access and modify the value of a variable at any time.
- Variables are used extensively in programming for tasks like tracking scores, managing game states, and more.

Creating Variables in Godot

Let's create our first variable in Godot. Follow these steps:

1. Open your script in Godot.
2. To create a variable, start by typing var followed by the variable name and its default value.

For example, let's create a variable named score and set its initial value to 0:

```
var score = 0
```

You can also set the variable to a different value, like 5:

```
var score = 5
```

Printing Variables

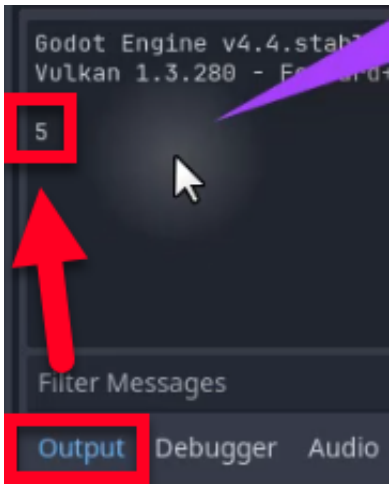
To see the value of a variable, you can print it to the output. Here's how you can do it:

1. Create a function named _ready.
2. Inside the _ready function, use the print function to output the variable's value.

Here is the complete code:

```
extends Node2D

var score = 5

func _ready():
    print(score)
```

When you run the scene, you should see the value 5 printed in the output.



Changing Variable Values

You can change the value of a variable at any time. For example, let's change the value of score to 10 initially, change it to 20 in the function, and then print it:

```
extends Node

var score = 10

func _ready():
    score = 20
    print(score)
```

When you run the scene, you should see the value 20 printed in the output.

Order of Execution

The order in which code is executed is crucial. In a function, code is run line by line from top to bottom. Let's see an example:

```
extends Node

var score = 10

func _ready():
    print(score)
    score = 20
    print(score)
```

When you run the scene, you should see the values 10 and then 20 printed in the output. This demonstrates how the order of execution affects the output.

Statically Typing Variables

Godot allows you to define the data type of a variable, which is known as statically typing. Here are some common data types:

- int: Whole numbers
- float: Decimal numbers
- bool: True or false values
- String: Text

Let’s create variables with different data types:

```
extends Node2D

var score : int = 10
var move_speed : float = 2.53
var game_over : bool = false
var ability : String = "slash"
```

Naming Conventions

In Godot, it’s common to use **snake_case** for naming variables. This means using lowercase letters and underscores to separate words. For example:

```
var move_speed: float = 2.53
```

This convention helps keep your code clean and readable.

Conclusion

In this lesson, you learned about variables, how to create them, print them, and change their values. You also learned about different data types and naming conventions in Godot. In the next lesson, we will explore more about variables and how to use them effectively in your projects.

Thank you for joining this lesson, and see you in the next one!

Welcome back to our course on game development. In this lesson, we will continue exploring variables in GDScript, the scripting language used in the Godot engine. Additionally, we will discuss the importance of proper indentation in GDScript and provide a challenge to reinforce your understanding.

Variables Re-Review

Let’s have a quick re-review. In the last lesson, we discussed how variables are used to store data that can be used and manipulated throughout your game. In GDScript, you can define variables of different data types, including integers (whole numbers), floating-point numbers (decimal numbers), booleans (true or false), and strings (text). Here is an example of how to define variables of each type:

```
var score : int = 10
var move_speed : float = 2.53
var game_over : bool = false
var ability : String = "slash"
```

Printing Variable Values

To print the values of variables to the output, you can use the print function. This is useful for debugging and verifying that your variables hold the expected values. While last lesson we used this for our score, you can also use it for the other data types as well:

```
func _ready():
    print(score)
    print(move_speed)
    print(game_over)
    print(ability)
```

Proper Indentation

Proper indentation is crucial in GDScript. All code that you want to run inside a function must be indented. If the indentation is not correct, it will result in errors as GDScript does not understand the code is meant to be part of the function. Below is an example of improper indentation:

```
func _ready():
print(score)
print(move_speed)
print(game_over)
print(ability)
```

Here is an example of proper indentation:

```
func _ready():
    print(score)
    print(move_speed)
    print(game_over)
    print(ability)
```


Changing Variable Values

Changing the value of a variable is straightforward. You simply assign a new value to the variable using the equals sign (=). Last lesson we showcased how to do this with the score value. Like printing though, we can assing new values to our other variable types as well.

```
func _ready():
    move_speed = 0.1183
    game_over = true
    ability = "attack"
```

Challenge

To reinforce your understanding of variables and data types, here is a challenge for you:

1. Create a variable for the country name
2. Create a variable for the population
3. Create a variable for the highest altitude
4. Create a variable for whether or not the country is landlocked

For each variable above, you want to make sure to assign both the value and an appropriate data type. In the next lesson, we'll go over the solution!

In this lesson, we will focus on solving our challenge issued in the last lesson.

Challenge Recap

As a recap before we dive into the solution, our challenge to you was to:

1. Create a variable for the country name
2. Create a variable for the population
3. Create a variable for the highest altitude
4. Create a variable for whether or not the country is landlocked

For each variable, you were also asked to statically type it with the most appropriate data type.

Let's jump into the solution for this now!

Clearing the Script

To start, let's clear any existing variables and code inside the _ready function. This ensures we have a clean slate to work with. However, leaving the _ready function empty will result in an error. To avoid this, we use the pass keyword, which acts as a placeholder.

```
extends Node2D

func _ready():
    pass
```

Defining Variables

Next, we will define several variables to store different types of data. Each variable will have a specific type and an initial value.

- **String:** For text data, such as the name of a country.
- **Integer (int):** For whole numbers, such as the population of a country.
- **Float:** For decimal numbers, such as the highest altitude of a country.
- **Boolean (bool):** For true/false values, such as whether a country is landlocked.

Let's define these variables:

```
extends Node2D

var country_name : String = "Australia"
var population : int = 25000000
var highest_altitude : float = 2.228
var landlocked : bool = false

func _ready():
    pass
```

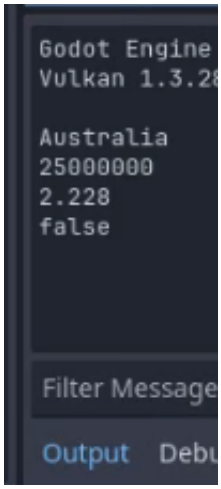
Printing Variables

Now that we have defined our variables, we can print their values to the output. We will use the print function to output the values of our variables. Here is how you can do it:


```
extends Node2D

var country_name : String = "Australia"
var population : int = 25000000
var highest_altitude : float = 2.228
var landlocked : bool = false

func _ready():
    print(country_name)
    print(population)
    print(highest_altitude)
    print(landlocked)
```

Running the Scene

After saving the script, run the scene. You should see the values of your variables printed in the output console. This confirms that our variables are defined and initialized correctly.



Modifying Variable Values

While we can print and change the values of our variables within the _ready function, sometimes we need to modify them without changing their entire value. This is where operators come into play. Operators allow us to perform operations like addition, subtraction, multiplication, and more.

For example, if we have a score variable and we want to increase it by one, we can use the addition operator. Similarly, we can use other operators to manipulate floating-point numbers or combine strings.

We will explore operators in more detail in the next lesson.

Congratulations! You have successfully defined, initialized, and printed variables in Godot. This foundational knowledge will be crucial as you progress through the course and build more complex games and applications.

Variables are containers that store data values, such as numbers, text, or objects. They have names that allow you to access and manipulate the stored data. Variables can change during the game's runtime, enabling dynamic behavior and flexibility in managing and processing information throughout your code.

How to Create a Variable

Variables are defined with the **var** keyword, then a name, then an equals sign, then a value.

Here we have defined a variable called *health* with a value of 100.

```
var health = 100
```

As mentioned, we can modify the value off a variable, like so:

```
health = 90
```

Types of Variables

Godot has a number of different variable types, known as data types.

- **Integer** – Whole numbers: 1, 5, -12, 520, 4325, etc.
- **Float** – Decimal numbers: 0.552, 54.2, -2.99, 42143.1222, etc.
- **String** – Text: "Hello!", "Player Name", etc.
- **Boolean** – True or false.

Here's how we can create a variable of each data type:

```
var health = 100
var speed = 5.25
var name = "Godot"
var can_fly = false
```

Looking at this, it may be hard to know which variable is for which data type. We can define the data type that the variable will use. This is known as static typing in programming – where we are strict in defining what things are and what they can do.

```
var health : int = 100
var speed : float = 5.25
var name : String = "Godot"
var can_fly : bool = false
```

Do be aware that there are many more types of variables you can create. These are just the most common, and the ones you will see across many other languages.

Additional Resources

If you wish to learn more, you can refer to the Godot documentation.

- GDScript Reference

In this lesson, we will explore the fundamental concept of operators in programming. Operators are symbols that instruct the computer to perform specific mathematical operations. They are essential for manipulating numbers and variables. We will focus on four primary operators: addition, subtraction, multiplication, and division. These operators allow us to increase, decrease, multiply, and divide values effectively.

Understanding Operators

Operators are used to perform various mathematical operations on variables and values. Here are the four main operators we will cover:

- **Addition (+)**: Increases the value of a variable.
- **Subtraction (-)**: Decreases the value of a variable.
- **Multiplication (*)**: Multiplies the value of a variable.
- **Division (/)**: Divides the value of a variable.

Using Operators in Godot

Let’s dive into Godot and see how we can use these operators in practice. We will start by creating a new variable and performing various operations on it.

Addition and Subtraction

First, let’s declare a variable called score and initialize it to 0:

```
var score : int = 0
```

Next, we will use the addition operator to increase the value of score by 1:

```
score += 1
```

The += operator adds the value on the right to the value on the left. Similarly, we can use the subtraction operator to decrease the value of score by 5:

```
score -= 5
```

Multiplication and Division

For multiplication, we use the asterisk (*) symbol. Let’s multiply score by 2:

```
score *= 2
```

For division, we use the forward slash (/) symbol. Let’s divide score by 4:

```
score /= 4
```

Working with Floating Point Numbers

Operators can also be used with floating-point numbers (decimals). Let's declare a variable called speed and initialize it to 5.5:

```
var speed : float = 5.5
```

We can multiply speed by a decimal value, such as 2.548:

```
speed *= 2.548
```

Similarly, we can divide speed by a decimal value, such as 2.09:

```
speed /= 2.09
```

Concatenating Strings

Operators can also be used with strings to concatenate (combine) them. Let's declare a variable called text and initialize it to "first":

```
var text : String = "first"
```

We can add another string to text using the addition operator:

```
text += "second"
```

This will result in text containing the value "firstsecond".

Challenge

To reinforce what you've learned, let's try a challenge. Create a variable called money and initialize it to 10. Perform the following operations using the appropriate operators:

1. Add 5 to money.
2. Double the value of money.
3. Subtract 3 from money.
4. Halve the value of money.

In the next lesson, we'll go over the solution!

In this lesson, we will continue our exploration of operators in GDScript. We'll first go over the solution to our challenge, and then explore some additional ways we can use them to manipulate variables. Let's dive in.

Challenge Solution

We'll start by going over the solution to our challenge – where you were tasked with creating a money variable, initializing it to 10, and performing the following on it:

1. Add 5 to money.
2. Double the value of money.
3. Subtract 3 from money.
4. Halve the value of money.

Below you'll find the solution where we perform this series of operations on this variable within the _ready function.

```
extends Node2D

var money : int = 10

func _ready():
    money += 5  # Add 5 to money
    money *= 2  # Multiply money by 2
    money -= 3  # Subtract 3 from money
    money /= 2  # Divide money by 2
    print(money)  # Print the final value of money
```

In the code above:

- money += 5 adds 5 to the current value of money.
- money *= 2 multiplies the current value of money by 2.
- money -= 3 subtracts 3 from the current value of money.
- money /= 2 divides the current value of money by 2.
- print(money) prints the final value of money to the output.

Understanding Variable Scope

Next, let's discuss the concept of variable scope. Scope refers to the accessibility of variables in different parts of your code. Variables declared outside of a function have a global scope, meaning they can be accessed by any function within the script. Variables declared inside a function have a local scope, meaning they can only be accessed within that function.

This has vast implications for how we manipulate variables, so keep your declarations in mind when creating them!

```
var x : int = 2 # Global variable available in any function

func _ready():
    x = 3 # As a global variable, X can be accessed and changed anywhere
    var y : float = 4.2 # Local variable I can only access and change in _ready().
```


Complex Equations and Variable Manipulation

Up until this point, we've only been performing simple calculations where we manipulate a variable with a single number. However, we are not limited to this. For example, we can assign a variable to the result of an equation like "2 + 2". We can also chain our equations, making them as complex as we require.

```
func _ready():
    var a : int = 5  # Local variable
    a = 10 + 2  # Assign the result of 10 + 2 to a
    a = 5 * 10 + 2 + 81 / 2  # Perform complex arithmetic and assign the result to a
```

In the code above:

- var a : int = 5 declares a local variable a and initializes it to 5.
- a = 10 + 2 assigns the result of 10 + 2 to a.
- a = 5 * 10 + 2 + 81 / 2 performs complex arithmetic and assigns the result to a.

As variables themselves can store numerical values, we can also use the variables directly in our equations. This is frequently used in game development to dynamically manipulate aspects like movement speed, attack values, and so forth – giving you immense control over the way your games work!

```
func _ready():
    var a : int = 5  # Local variable
    a = 10 + 2  # Assign the result of 10 + 2 to a
    a = 5 * 10 + 2 + 81 / 2  # Perform complex arithmetic and assign the result to a

    var b : int = a * 2  # Create a new variable b and assign it the value of a * 2
    var c : int = 0  # Create a new variable c and initialize it to 0
    c = a / b  # Assign the result of a / b to c
```

In the new parts of the code above:

- var b : int = a * 2 declares a new variable b and assigns it the value of a * 2.
- var c : int = 0 declares a new variable c and initializes it to 0.
- c = a / b assigns the result of a / b to c.

Summary

In this lesson, we learned how to use operators to modify variables and understood the concept of variable scope. By practicing these concepts, you will become more comfortable with manipulating variables and performing arithmetic operations in Godot.

In the next lesson, we will explore the concept of conditions, which allow us to branch out our code based on certain criteria.

Operators are symbols or keywords that perform operations on variables and values. They can be used to manipulate data, perform calculations, compare values, or combine logical conditions.

Arithmetic Operators

Arithmetic operators are symbols used in programming to perform basic mathematical operations on numbers.

- **Addition** – Adds the value on the left to the value on the right.
- **Subtraction** – Subtracts the value on the right from the value on the left.
- **Multiplication** – Multiplies the value on the left with the value on the right.
- **Division** – Divides the value on the left by the value on the right.

```
var a = 10

# addition
var add = a + 5

# subtraction
var sub = a - 2

# multiplication
var mult = a * 10

# division
var div = a / 2
```

However, it's worth noting the addition operator (+) can also be used on strings. In this case, rather than performing a mathematical calculation, the operator will combine the strings together in a process called concatenation.

```
var name = "John " + "Smith"
```

Assignment Operators

Assignment operators in programming are used to assign or update the value of a variable.

For example, if we wanted to increase the value of a variable we could do it like this:

```
health = health + 10
```

But a more streamlined way would be to use an assignment operator:

```
health += 10
```

These two methods work in the exact same way, but assignment operators allow us to shrink down the code! Let's now look at the different assignment operators.

```
var height = 140.5

# addition
height += 10

# subtraction
height -= 20

# multiplication
height *= 2

# division
height /= 4
```

Additional Resources

If you wish to learn more, you can refer to the Godot documentation.

- GDScript Reference

Welcome to this lesson on conditions in coding with Godot! In this session, we will explore what conditions are, how to use them, and how they work within if statements. By the end of this lesson, you will be able to write basic conditional statements to control the flow of your code.

What is a Condition?

A condition is a statement that evaluates to either true or false. Conditions are essential in programming as they allow us to make decisions and control the flow of our code. For example, we can use the equality operator (==) to check if two values are equal:

```
5 == 5  # This condition is true


3 == 7  # This condition is false
```

If Statements

If statements allow us to execute a block of code only if a certain condition is true. If the condition is false, the code block is skipped. Here is a basic example:

```
var a = 5
if a == 5:
    print("Condition is true!")
```

In this example, the message “Condition is true” will be printed because the condition a == 5 is true.

Setting Up Conditions in Godot

Let’s set up some conditions in Godot. We will start by creating a new variable called score and assigning it a value of 10. Then, we will use an if statement to check if the score is equal to 10.

```
var score = 10

func _ready():
    if score == 10:
        print("score is 10")
```

If you run this code, you will see the message “score is 10” printed to the output. However, if you change the value of score to 5, the message will not be printed because the condition is no longer true.

Comparison Operators

In addition to the equality operator, there are several other comparison operators you can use:

- > (greater than)
- < (less than)
- >= (greater than or equal to)

- <= (less than or equal to)
- != (not equal to)

Example: Greater Than

Let's check if the score is greater than 5:

```
var score : int = 6

func _ready():
    if score > 5:
        print("Score is greater than 5")
```

If you run this code with score set to 6, you will see the message "Score is greater than 5" printed to the output. Likewise, if you set score to something like 4, the code block will be skipped since 4 is not greater than 5.

Example: Less Than

Now, let's create two variables, a and b, and check if a is less than b:

```
var a = 50
var b = 100

func _ready():
    if a < b:
        print("a is less than b")
```

Since a is 50 and b is 100, the message "a is less than b" will be printed.

Example: Not Equal To

Finally, let's check if a is not equal to b using the not equal to operator (!=):

```
var a = 50
var b = 100

func _ready():
    if a != b:
        print("a is not b")
```

Since a is not equal to b, the message "a is not b" will be printed.

Summary

In this lesson, we covered the basics of conditions and if statements in Godot. We learned how to use various comparison operators to check different conditions and control the flow of our code. In the next lesson, we will explore more advanced uses of conditions and expand on different ways to utilize them.

Thank you for joining us, and we look forward to seeing you in the next lesson!

In this lesson, we will delve deeper into the concept of conditions and if statements. Specifically, we will explore the use of elif (else if) and else statements. These constructs allow us to create more complex decision-making structures in our code. Let's get started.

Understanding If, Elif, and Else Statements

To illustrate the use of if, elif, and else statements, let's consider a scenario where we have a variable called score. We want to grade our user based on their score, which ranges from 0 to 100.

```
var score : int = 100
```

The grading criteria are as follows:

- Score above 80: Grade A
- Score above 60: Grade B
- Score above 30: Grade C
- Any other score: Grade D

Implementing the Grading System

Let's start by implementing the grading system using if statements. We will begin with the condition for Grade A:

```
if score > 80:
    print("A")
```

If we run this code with a score of 100, it will print "A" because 100 is greater than 80. However, if we add another condition for Grade B:

```
if score > 80:
    print("A")
if score > 60:
    print("B")
```

Running this code with a score of 100 will print both "A" and "B", which is not what we want. We want to print only one grade based on the score. This is where the elif statement comes in.

Using Elif Statements

The elif statement allows us to check multiple conditions in a sequence. If one condition is true, the subsequent elif statements are not checked. Here's how we can use elif to implement our grading system:

```
if score > 80:
    print("A")
elif score > 60:
    print("B")
```


With this code, if the score is greater than 80, it will print “A” and skip the elif statement. If the score is not greater than 80 but is greater than 60, it will print “B”.

Adding More Elif Statements

We can add more elif statements to cover all the grading criteria:

```
if score > 80:
    print("A")
elif score > 60:
    print("B")
elif score > 30:
    print("C")
```

Using Else Statements

The else statement is used to cover any conditions that are not met by the if or elif statements. In our case, if the score is not greater than 80, 60, or 30, we will print “D”:

```
if score > 80:
    print("A")
elif score > 60:
    print("B")
elif score > 30:
    print("C")
else:
    print("D")
```

With this code, if the score is 20, it will print “D” because none of the previous conditions are met.

Nesting If Statements

It’s also possible to nest if statements within other if statements. For example, we can check if the score is exactly 100 and print “Perfect Score”:

```
if score > 80:
    print("A")
    if score == 100:
        print("Perfect Score")
```

If the score is 100, this code will print “A” and “Perfect Score”.

Challenge

Now it’s your turn to practice. Create a boolean variable called game_over. If the game is over, print “Go to menu” to the output; otherwise, print “Keep playing”. Give it a try and see how you do! We’ll cover the solution in the next lesson.

In this lesson, we will go over the solution to our challenge. As a recap, you were tasked with creating a game_over variable and using a conditional statement to either print “Go to menu” or “Keep playing”. Let’s dive into the answer.

Creating a Boolean Variable

First, let’s create a new boolean variable called game_over. In Godot, the boolean data type is abbreviated as bool. We will initialize this variable to true.

```
var game_over : bool = true
```

Using If-Else Statements

Next, we will use an if-else statement to check the value of game_over and print different messages based on its value.

- If game_over is true, we will print “Go to menu”.
- If game_over is false, we will print “Keep playing”.

Here is the code snippet for the if-else statement:

```
func _ready():
 if game_over == true:
  print("Go to menu")
 else:
  print("Keep playing")
```

Testing the Condition

Let’s test the condition by running the game:

- Since game_over is initially set to true, the output should display “Go to menu”.
- If we change the value of game_over to false, the output should display “Keep playing”.

Using Other Data Types in Conditions

Conditional statements can be used with various data types, not just booleans or numbers. For example, we can use strings to check if a password matches a specific value.

Let’s create a string variable called password and set it to “123”. We will then use an if statement to check if the password matches “123” and print “Enter” if it does.

```
var password : String = "123"

func _ready():
 if password == "123":
  print("Enter")
```

Testing the String Condition

Let’s test the string condition by running the game:

- Since password is set to “123”, the output should display “Enter”.
- If we change the value of password to something other than “123”, the output will not display “Enter”.

Summary

In this lesson, we learned how to use conditional statements in Godot to control the flow of your game based on certain conditions. We created a boolean variable and used it within an if-else statement to print different messages to the output. We also explored how to use other data types, such as strings, in conditional statements.

Remember, conditional operators like == (equals) and != (not equals) are used for rigid values like texts or booleans, while operators like > (greater than) and < (less than) are mainly used for numbers that have a spectrum-based value.

In the next lesson, we will wrap up our learning on conditions before moving onto our next programming concept.

As of now, our code has executed in sequential order, line by line, every line. But what if we want to run one bit of code only if something else happens? Well that's where conditions come into play. Conditions allow us to branch our code, executing certain lines only if a condition has been met.

How Conditions Work

At its core, a condition's job is to return either **true** or **false**. It does this by checking if the defined condition is true or not.

For example, this condition is *TRUE*, because 10 is equal to 10.

```
10 == 10
```

This condition on the other hand, is *FALSE*, because 5 is not greater than 20.

```
5 > 20
```

Comparison Operators

Comparison operators compare two values then returns either true or false.

```
var a = 5
var b = 10

# equal to (return false)
a == b

# not equal to (return true)
a != b

# less than (return true)
a < b

# greater than (return false)
a > b

# less than or equal to (return true)
a <= b

# greater than or equal to (return false)
a >= b
```

If Statements

Now that we understand conditions, what is an if statement? Well, it allows us to run a certain bit of code ONLY if a condition is true.

```
var health = 50
```



```
if health <= 0:
    print("Game Over!")
```

The above example would NOT run the print function, because the condition it is checking (if health, which is 50, is less than or equal to 0) returns false.

This condition on the other hand, would return true, so the if statement would run the code inside:

```
var has_key = true

if has_key == true:
    print("Open Door")
```

Things to note!

1. Remember to add a colon at the end of the if statement (:)
2. Remember to indent the code you wish to be inside the if statement by 1 tab's width (this should automatically be done for you when you hit enter)

Else If

An else if statement allows us to check another condition if the first one returned false.

```
var a = 100

if a < 50:
    print("a is less than 50")
elif a > 50:
    print("a is more than 50")
```

In the above example, the first if statement will be false, so the second (elif) statement will be checked. That condition is true, so the print line will be ran.

You can also stack else if statements as long as you wish!

```
var keys = 2

if keys == 0:
    # open no doors
elif keys == 1:
    # open red door
elif keys == 2:
    # open green door
elif keys == 3:
    # open blue door
```

Else

Finally, we have the else statement. This defines code which runs if the above if statements all return false.

```
var health = 20

if health <= 0:
    print("Game Over!")
else:
    print("Player is still alive")
```

In the example above, the second print line would run, since the first condition would return false.

Additional Resources

If you wish to learn more, you can refer to the Godot documentation:

- GDScript Reference

In this lesson, we will explore functions within GDScript, the scripting language used in the Godot Engine. Functions are reusable blocks of code that perform specific tasks. They are essential for organizing and managing your game's logic efficiently. We will look at built-in functions provided by Godot and also create our own custom functions.

Understanding Functions

Functions are blocks of code that can be called to perform specific actions. They help in modularizing your code, making it more readable and maintainable. In Godot, functions are defined using the func keyword.

Built-in Functions

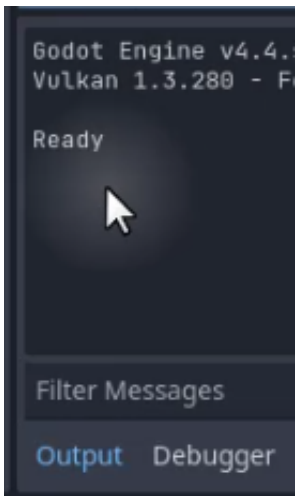
Godot provides several built-in functions that are commonly used in game development. Two of the most important ones are the _ready and _process functions.

The _ready Function

The _ready function is called once when the node to which the script is attached is added to the scene tree. It is typically used for initialization tasks.

```
func _ready():
    print("ready")
```

When you run the game, the message "ready" will be printed to the output once.



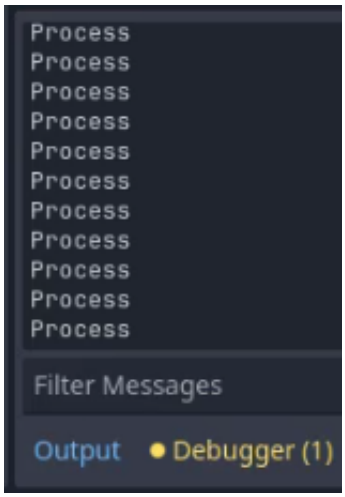
The _process Function

The _process function is called every frame. It is useful for tasks that need to be performed continuously, such as updating the player's position.

```
func _process(delta):
    print("process")
```

When you run the game, the message "process" will be printed to the output continuously,

indicating that the function is called every frame.



Creating Custom Functions

In addition to built-in functions, you can create your own custom functions to perform specific tasks. Functions consist of a specific syntax:

```
func [name]([paramas]):
```

- func is the keyword that lets GDScript know we're about to declare a function
- [name] is where you should give the function a **unique** name. We will use this name to call the function and tell our program to run its contents.
- [params] (short for parameters) is where we can pass data to our function – though this is entirely optional. We will discuss parameters in a moment.

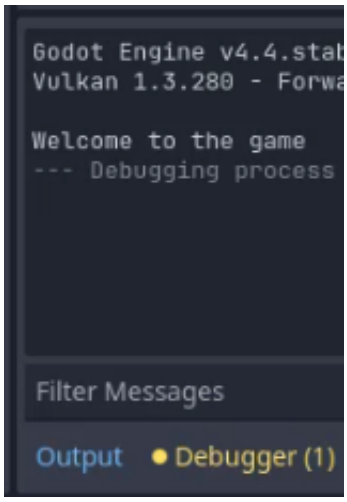
Let's create a simple function that prints a welcome message.

```
func _welcome_message():
    print("Welcome to the game")
```

To call this function, you need to use its name followed by parentheses:

```
func _ready():
    _welcome_message()
```

When you run the game, the welcome message will be printed to the output.



Functions with Parameters

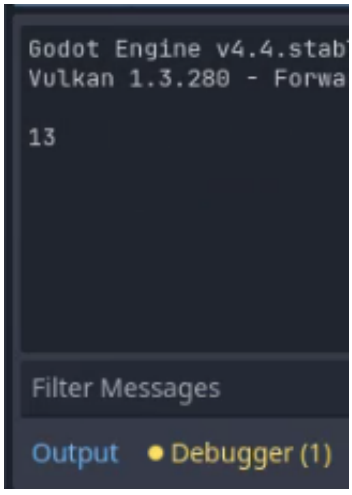
Functions can accept parameters, which are values passed to the function when it is called. Let's create a function that adds two numbers together.

```
func _add(a, b):
    var sum = a + b
    print(sum)
```

To call this function and pass parameters to it, you need to provide the values within the parentheses:

```
func _ready():
    _add(5, 8)
```

When you run the game, the sum of the two numbers (13) will be printed to the output.



Summary

In this lesson, we explored the concept of functions in GDScript. We looked at built-in functions like

_ready and _process, and we created our own custom functions. We also learned how to pass parameters to functions. In the next lesson, we will continue with functions and look at returning values from them.

Thank you for your attention, and see you in the next lesson!

In this lesson, we will delve deeper into the concept of functions in Godot. We will explore how to return values from functions, understand the significance of parameters, and learn how to statically type our functions. By the end of this lesson, you will have a solid understanding of how to create and use functions effectively in your Godot projects.

Parameters Recap

Parameters in functions act like temporary variables that are used within the function. Once the function has finished executing, these variables are essentially deleted. Let's consider a simple function called add that takes two parameters, a and b.

```
func _add(a, b):
    var sum = a + b
    print(sum)
```

In the above example, the function _add takes two parameters, a and b, adds them together, and prints the result. However, what if we want to use the result of this addition elsewhere in our code? This is where returning values comes into play.

Returning Values from Functions

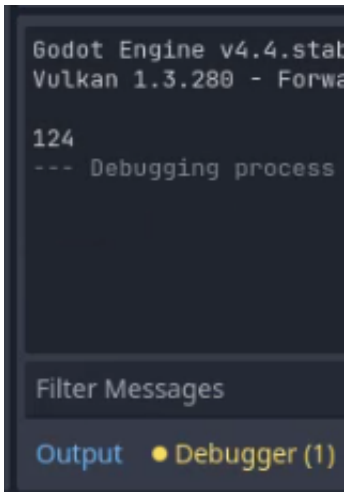
Returning a value from a function allows us to use the result of the function's computation elsewhere in our code. Let's modify the _add function to return the sum instead of printing it:

```
func _add(a, b):
    var sum = a + b
    return sum
```

Now, we can call the _add function and use its return value:

```
func _ready():
    var result = _add(42, 23)
    print(result)
```

In this example, the _add function returns the sum of 42 and 23, which is then stored in the variable result and printed to the output.



Statically Typing Functions

Statically typing our functions involves defining the data types of the parameters and the return value. This practice makes our code more readable and helps prevent errors, especially when working in a team. Let's statically type the _add function:

```
func _add(a : float, b : float) -> float:
    var sum : float = a + b
    return sum
```

In this example, we have defined the parameters a and b as floats, and the return type of the function is also a float. This makes it clear that the function expects floating-point numbers as inputs and will return a floating-point number as the output.

Challenge: Creating a Function with Conditions

To reinforce your understanding, create a function called _has_won. This function will take an integer parameter called score and return a boolean value indicating whether the score is above 100.

Here are the steps to create the function:

1. Define the function _has_won with a parameter score of type int.
2. Check if the score is greater than 100.
3. Return true if the score is greater than 100, otherwise return false.

In the next lesson, we will go over the solution and wrap up our unit on functions!

In this lesson, we will go over the solution to our challenge. Remember that our challenge was to create a function called _has_won. This function would take an integer parameter called score and return a boolean value indicating whether the score was above 100.

Creating the Function

To start, we need to create a new function called _has_won. As mentioned, this function will take an integer parameter called score and return a boolean value (true or false).

Here is the structure of the function:

```
func _has_won(score : int) -> bool:
```

Implementing the Logic

Inside the function, we will check if the score is greater than or equal to 100. If it is, the function will return true; otherwise, it will return false.

The complete function looks like this:

```
func _has_won(score: int) -> bool:
    if score >= 100:
        return true
    else:
        return false
```

Using the Function

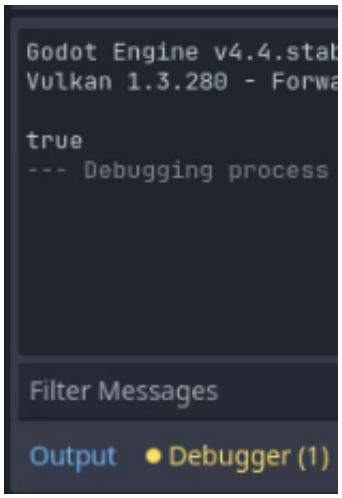
Now that we have our function, let's use it within the _ready function. We will create a variable called game_over and assign it the result of the _has_won function with a score of 120.

Here is how you can do it:

```
func _ready():
    var game_over = _has_won(120)
    print(game_over)
```

Testing the Function

To test our function, we will run the game. If the score is 120, the output should be true, indicating that the player has won the game.

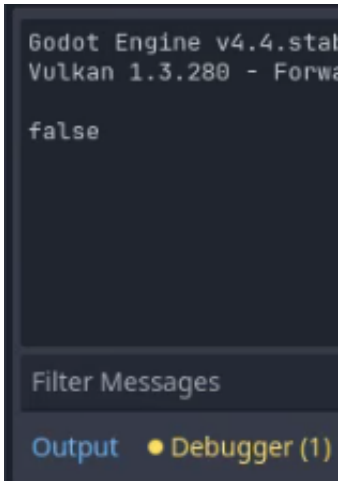


If we change the score to a value less than 100, the output should be false.

Let's test with a score of 5:

```
func _ready():
    var game_over = _has_won(5)
    print(game_over)
```

Running the game with this score should print false.



Summary

In this lesson, we learned how to create and use a function in Godot to determine if a player has won the game based on their score. Functions are powerful tools that can be used for various purposes, and you will encounter them frequently in your programming journey.

For the next lesson, we will do some wrap-up work for functions to cement your knowledge!

Functions are how we create reusable blocks of code. With Godot, you have two core functions already built-in to the engine for you: **_ready**, which gets called once when the node is first initialized, and **_process**, which gets called every frame.

Structure of a Function

Let's look at a function that simply prints something to the output:

```
func print_hello ():
    print("Hello")
```

So here we have defined a function, but if we were to run our game, nothing would happen. That is because functions on their own don't do anything. We need to **call** the function.

```
print_hello()
```

That is how we essentially run the function.

Parameters

Think of functions like a **factory**. You have stuff going in, it gets processed, then a result comes out. The same can be said for functions. **Parameters** are values we can feed into a function.

```
func add_then_print (a, b):
    var sum = a + b
    print(sum)
```

The above function takes in two parameters (a and b). These are essentially temporary variables that only this function can use.

To call this function, we would write it like so:

```
add_then_print(5, 20)
```

The resulting print message would be "25".

You can stack as many parameters as you wish into a function, just make sure to separate them by a comma.

Returning

I mentioned before how a function is like a factory. We have inputs, then a process inside, then finally an output. Functions, likewise, can output (or return) a result. We do this by writing the keyword **return** followed by a value.

```
func add (a, b):
```



```
    var sum = a + b
    return sum
```

The above function will add a and b together, then return the sum. So how can this be useful? Well since the *add* function is outputting a value, we can simply assign a variable to it like so:

```
var number = add(15, 55)
```

The *number* variable is equal to that function call, which returns a number. In this case, number would be equal to 70.

Additional Resources

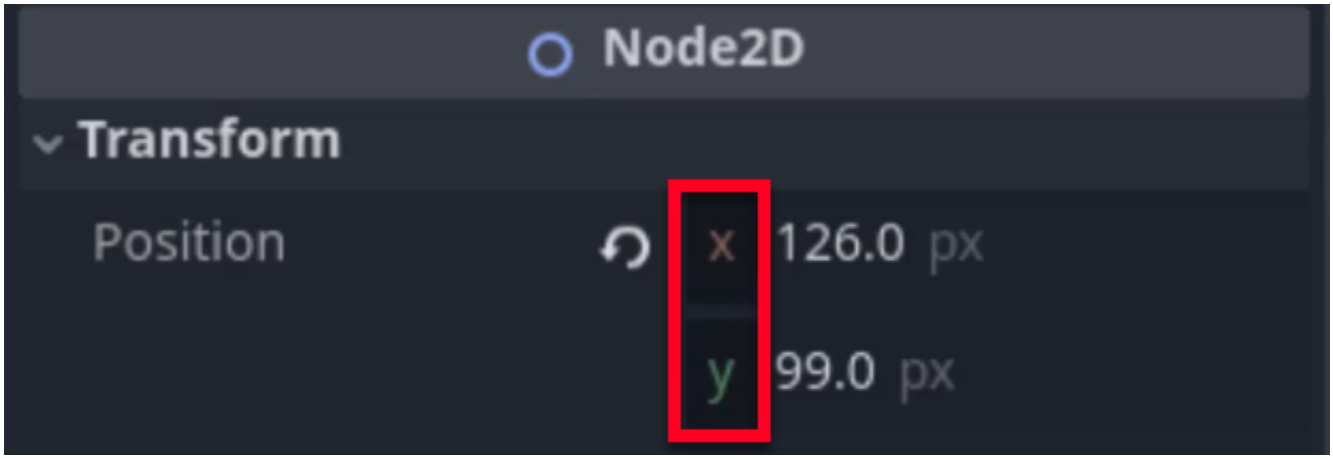
If you wish to learn more, you can refer to the Godot documentation.

- GDScript Reference

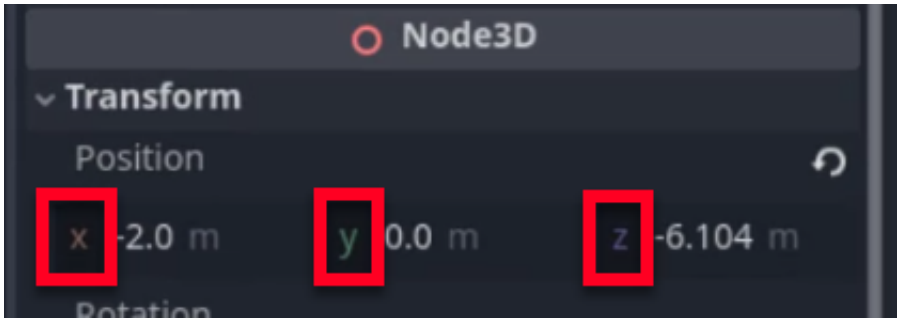
In this lesson, we will explore the concept of vectors in Godot. Vectors are essential for representing positions and directions in your game. We will learn how to access and modify vectors to move a player character around the screen.

Understanding Vectors

Vectors are used to represent positions and directions in 2D and 3D space. In 2D, a vector has two components: X (horizontal position) and Y (vertical position).

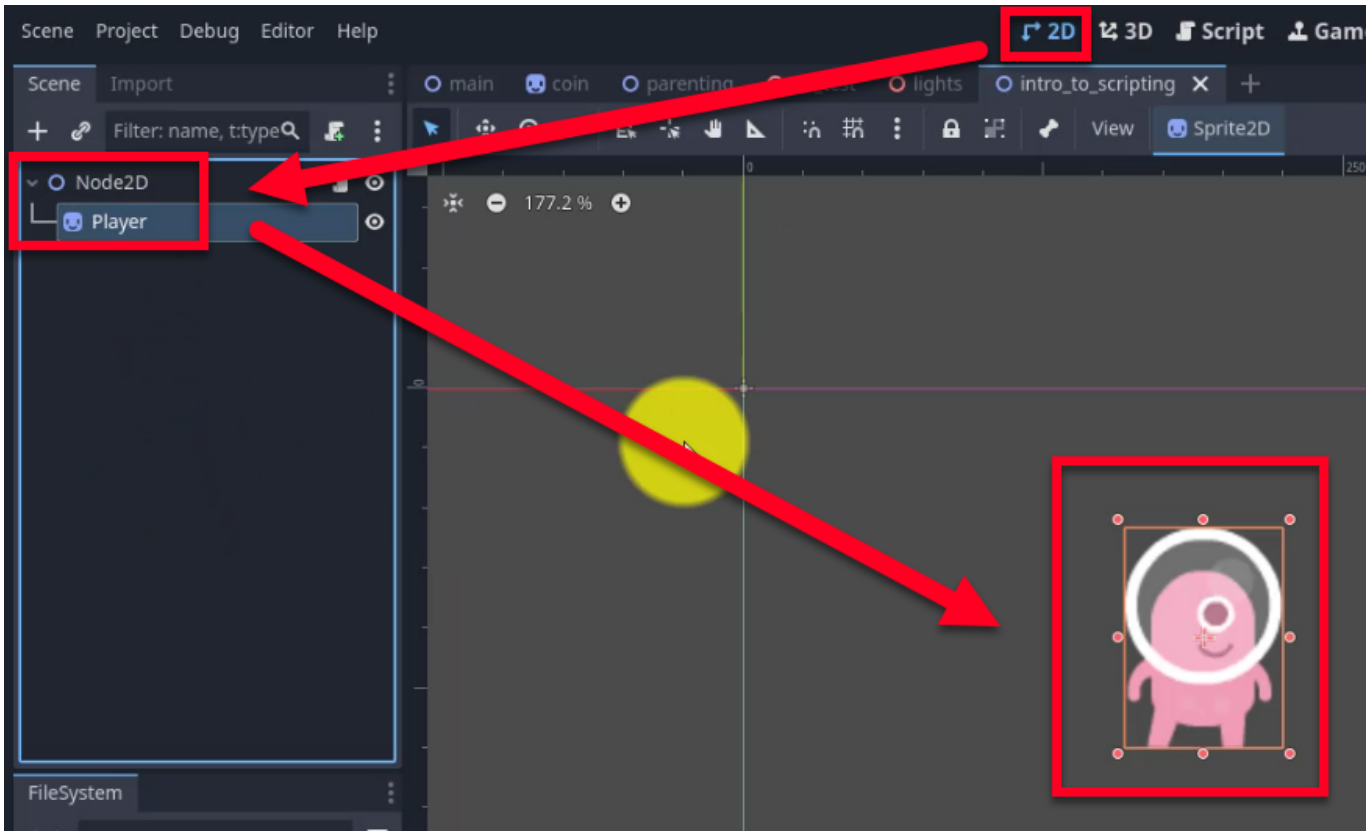


In 3D, a vector has three components: X, Y, and Z (depth).

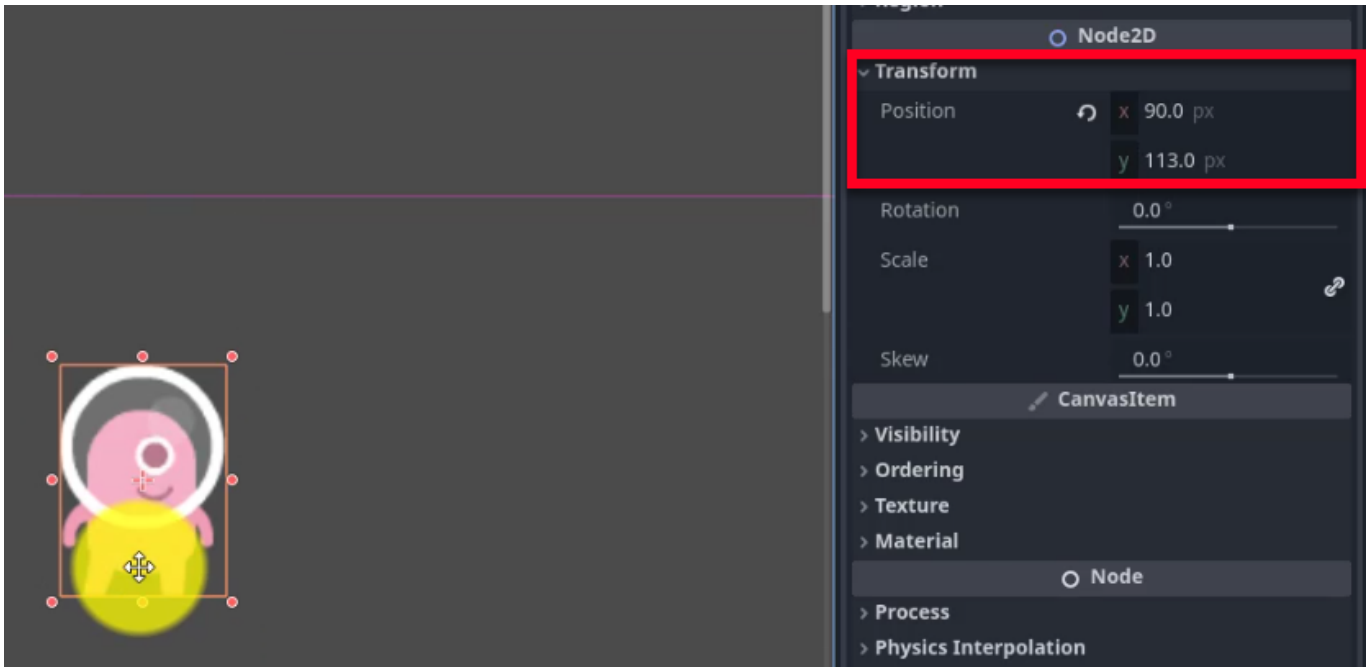


Accessing and Modifying Vectors

Let’s start by adding a player sprite to our scene and moving it around using vectors. Switch to 2D mode in Godot, and add a new Sprite2D node. Assign your player.png to it.



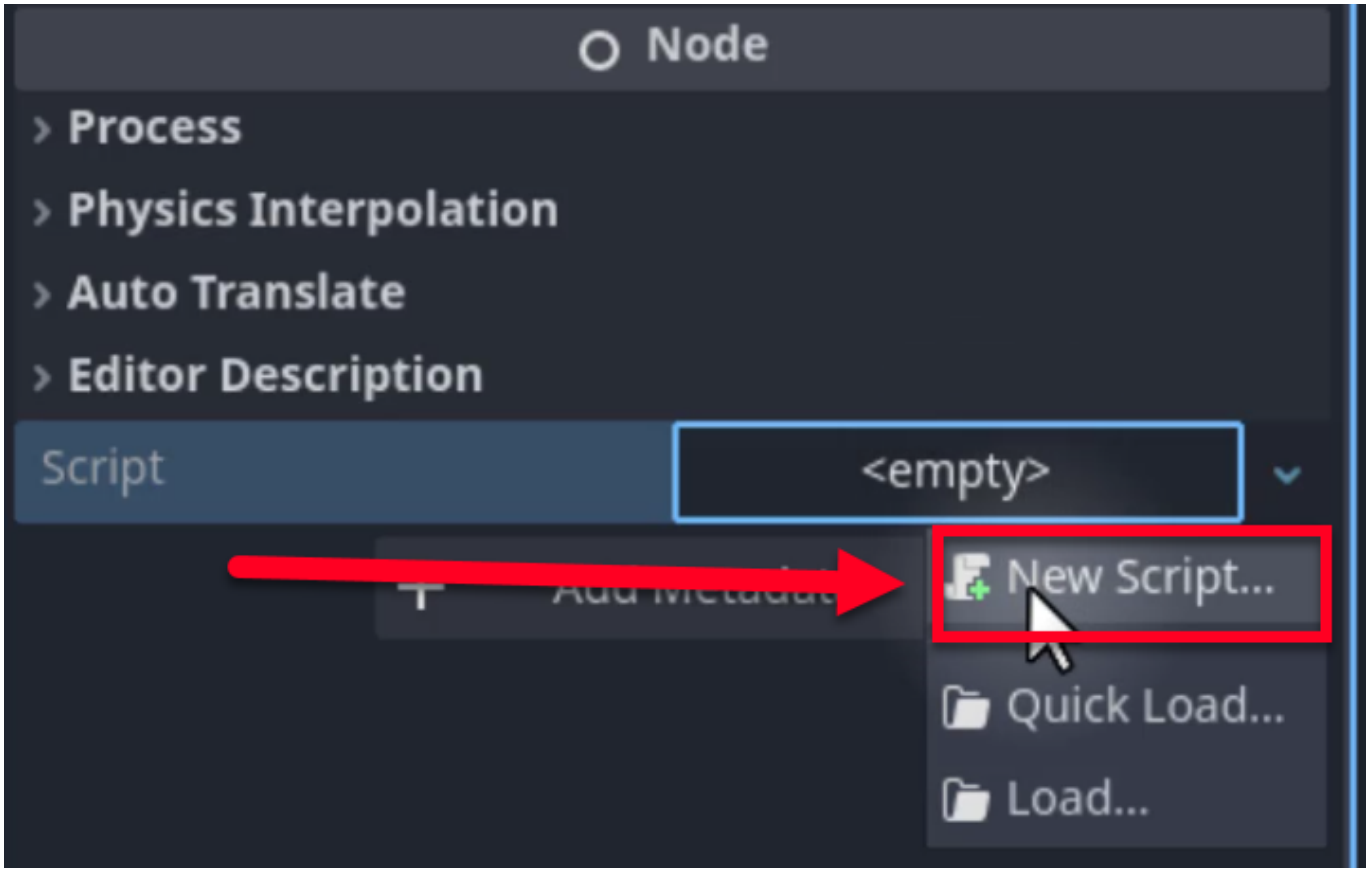
In the Inspector, locate the Transform section to see the position vector (X and Y coordinates). You can move your sprite around the scene and see how these values change as you move the node horizontally and vertically.



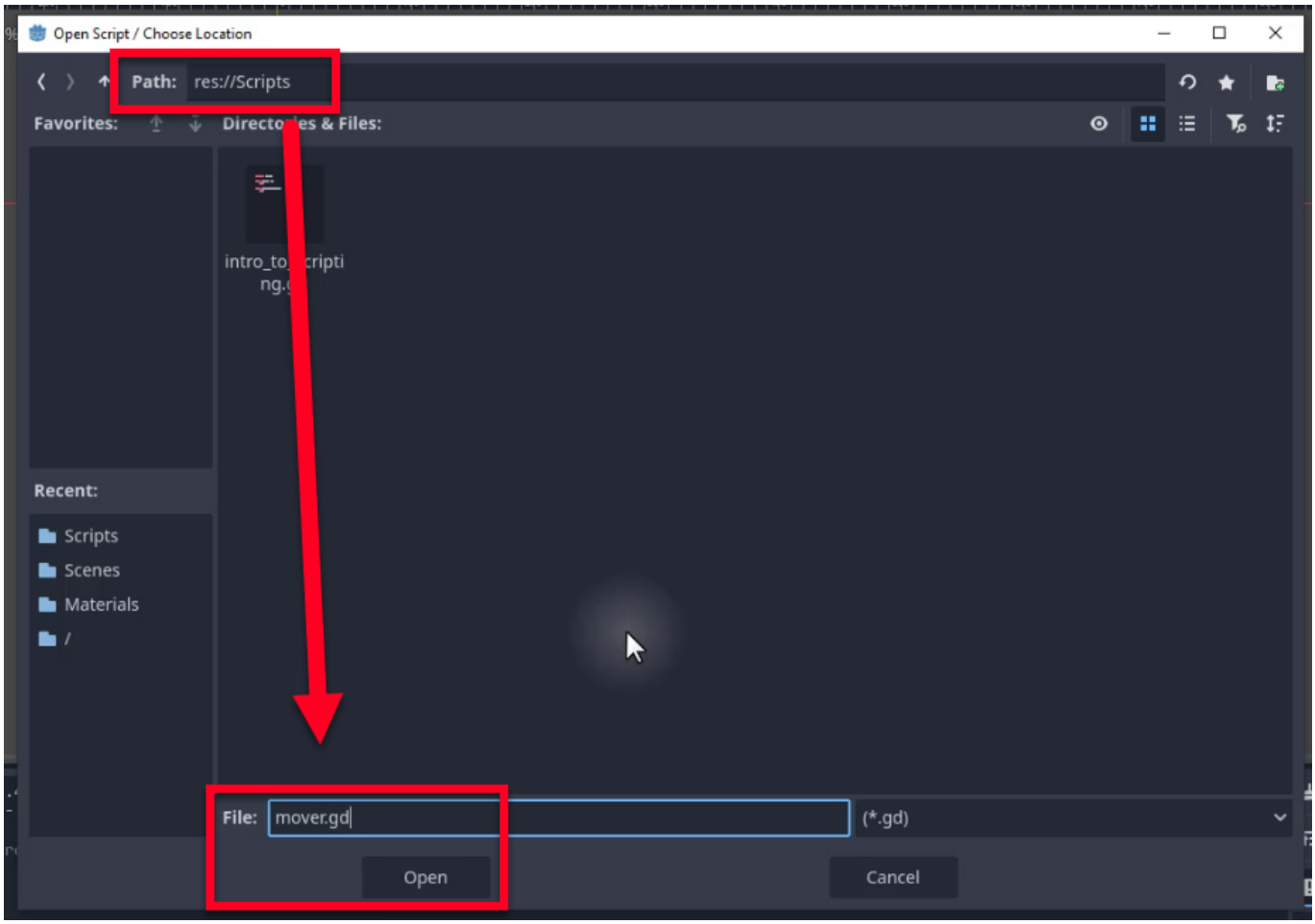
The position vector can be accessed and modified in your script to move the player, which is where the power of Vectors truly comes in! Let's cover that next.

Creating the Script

Select the player node and scroll down to the bottom of the Inspector. Click on the “Attach Script” icon (it looks like a scroll with a small arrow) to create a new script.



Save the script as mover.gd in your Scripts folder.



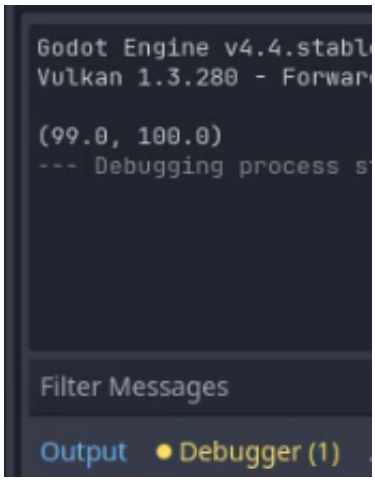
Inside the mover.gd script, we will modify the player's position vector.

Printing the Position

First, let's print the player's initial position:

```
extends Sprite2D

func _ready():
    print(position)
```

In the output, you will see a set of coordinates printed based on where your Player is positioned in the world.

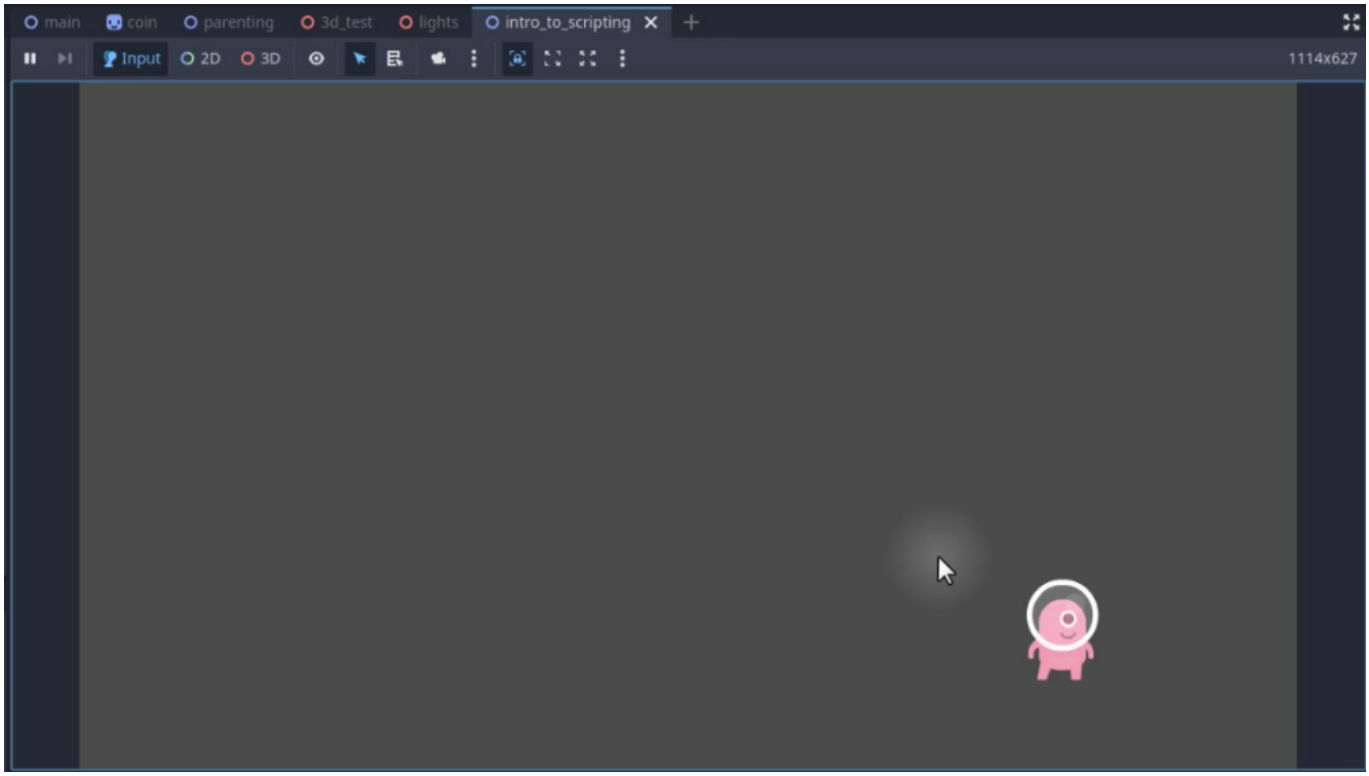


Modifying Individual Axes

Next, let's modify the X and Y coordinates individually:

```
extends Sprite2D

func _ready():
    position.x = 400
    position.y = 500
```

If we press play now, we will see the player has moved to those coordinates on the screen!



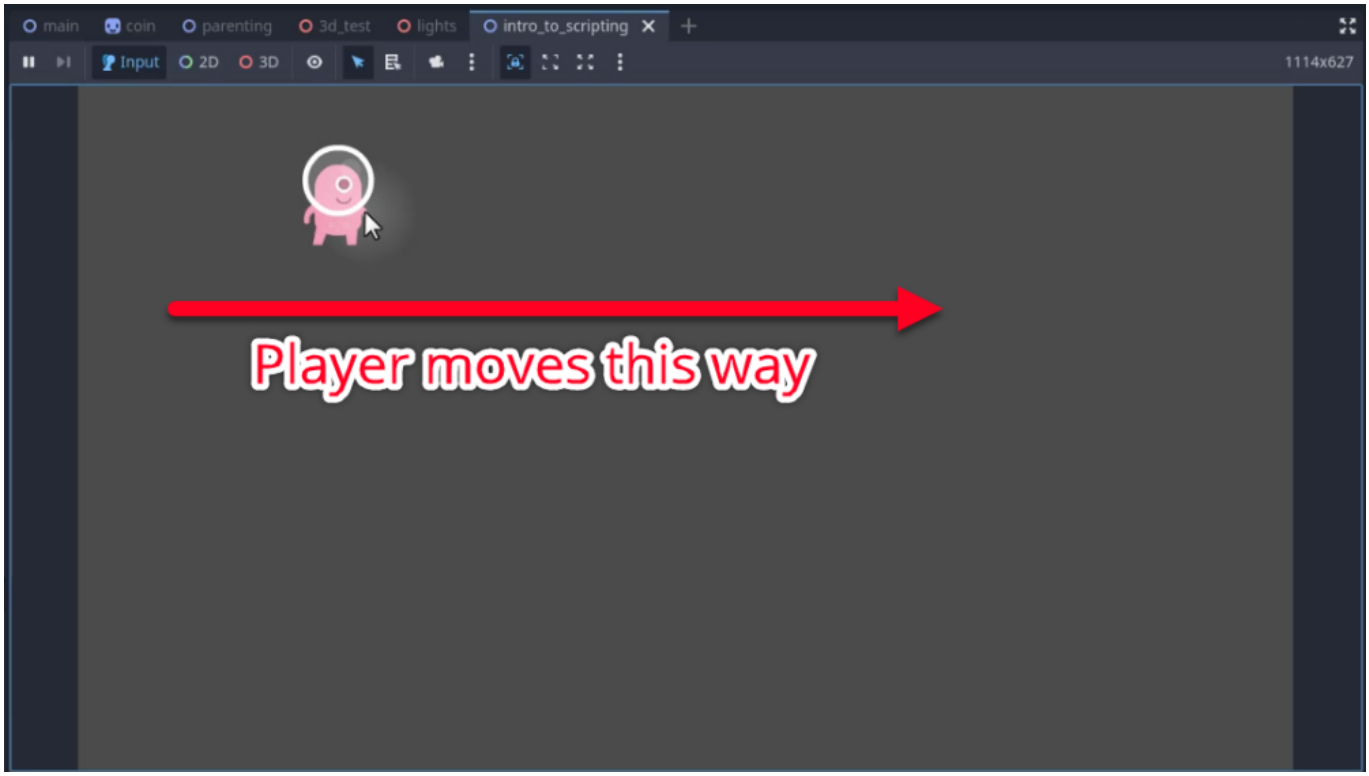
We can also modify the entire position vector at once, which is useful if we're changing both values:

```
extends Sprite2D

func _ready():
    position = Vector2(500, 200)
```

Moving the Player Over Time

To move the player continuously, we use the _process function, which runs every frame. For example, we can make the player move horizontally on the x-axis with the following, which moves it 1 pixel to the right per frame:

```
extends Sprite2D

func _process(delta):
    position.x += 1
```



Adding the Speed Variable

However, using a fixed value isn't ideal for actual games. Instead, we can create a speed variable and use that to modify the vector (allowing us to later quickly change the speed variable itself).

```
extends Sprite2D

var speed : float = 10.0

func _process(delta):
```



```
    position.x += speed
```

However, insofar, our implementation has a slight issue. Using a flat value as we've done here moves the node at a rate of **pixels per frame**. For a game, this means that a player who can run the game at a higher FPS will have the object move faster. Instead, to create a consistent experience, we need to make sure the object moves at a rate of **pixels per second**.

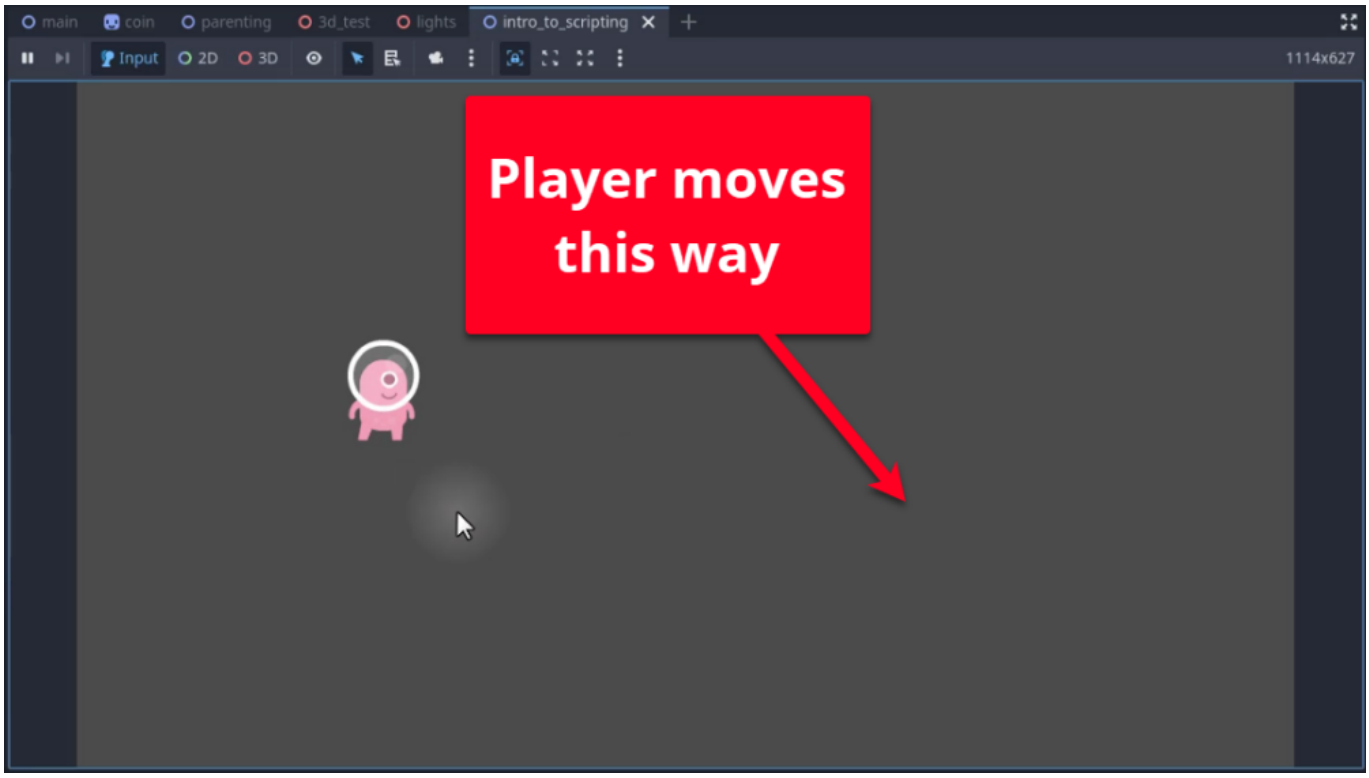
Fortunately, this can be achieved very easily by simply multiplying our speed variable by the **delta parameter**. The delta parameter is a dynamic parameter that contains the time between frames for your computer. The more FPS your computer can achieve, the smaller the number will be. This allows the code to dynamically adjust the final position value in a way that is consistent across computers.

```
extends Sprite2D

var speed : float = 10.0

func _process(delta):
    position.x += speed * delta
```

Moving in a Direction

While we have been using vectors to represent position, vectors can also be used to represent direction. For example, let's say we want our sprite to move to the right and down. We can use a vector to represent this direction.

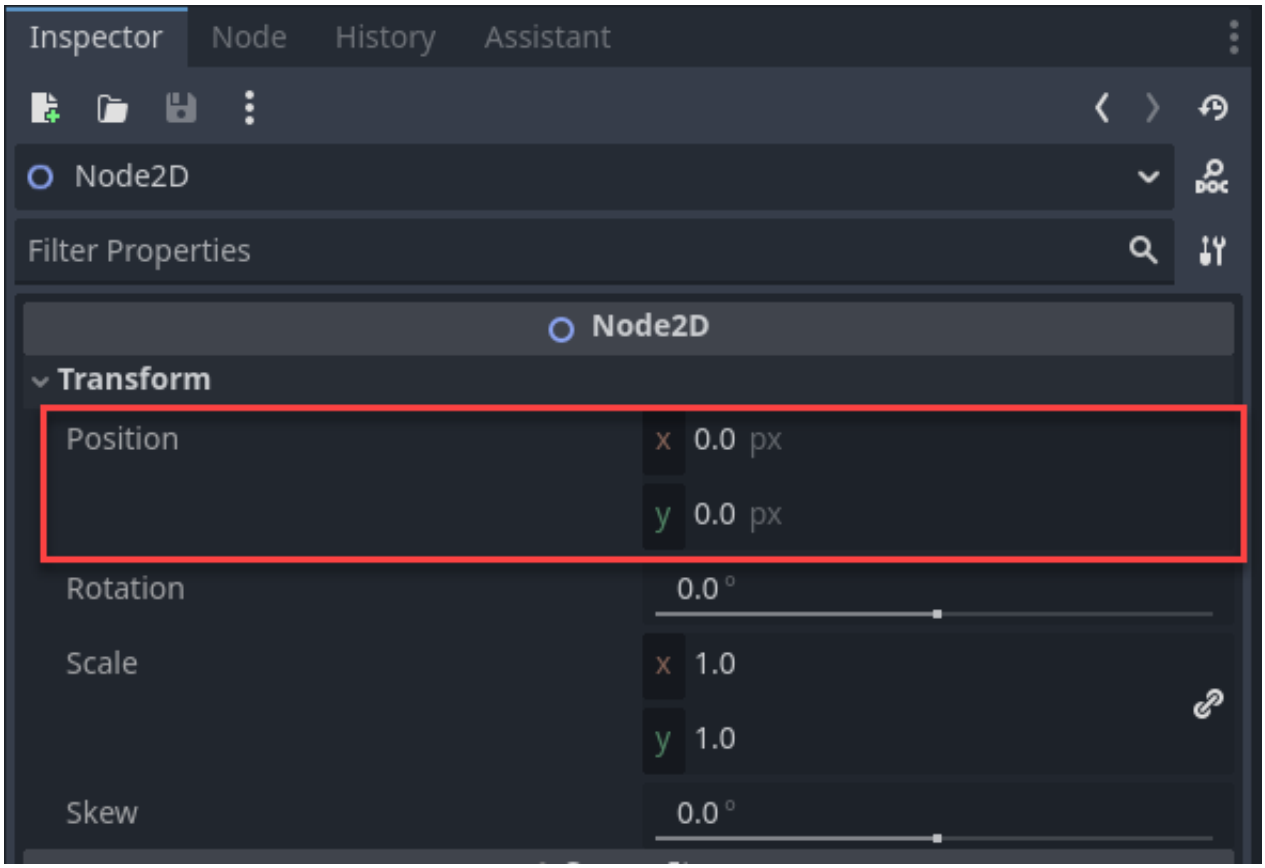
```
extends Sprite2D

var speed : float = 100

func _process(delta):
    var direction = Vector2(1, 1)
    position.x += direction * delta * speed
```

By following these steps, you've learned how to access and modify vectors to move a player character in Godot. Vectors are a powerful tool for handling positions and directions in your game, and understanding them is crucial for more advanced game development tasks.

Vectors are a data structure which are used to represent position, direction, velocity, etc.

Node2D's have a *Position* property that contains an X and Y value – this is a vector!



Using Vectors in Scripts

To access the global position vector of the node which the script is attached to, we can do it like this:

```
global_position
```

This will give us access to our global position vector, which contains an X and Y value. We can modify these values individually. If for example, we wanted to move our player 50 pixels to the right, we could write this:

```
global_position.x += 50
```

We can even assign the vector as a whole with the **Vector2** structure: *Vector2(x, y)*

```
global_position = Vector2(60, 5)
```

We can create a variable that stores a vector.


```
var pos = Vector2(10, 10)
```

And then add that as a whole to our position.

```
global_position += pos
```

We can add, subtract, multiply and divide vectors.

```
var dir = Vector2(1, 0)

global_position += dir * 100
```

In the above example, we would move to the right 100 pixels.

Delta

The **delta** parameter is one we find sent over with the **_process** function. We used it in the previous lesson, but what does it actually do?

Well, delta is basically the time between frames. We use this to convert measurements of pixels per frame, to pixels per second. This allows us to move objects around at the same speed independent of frame rate.

- At 60 fps, delta = 0.0166
- at 150 fps, delta = 0.006

This number is continuously changing based on the frame rate, so no matter what, our player will always move at a consistent rate.

Additional Resources

If you wish to learn more, you can refer to the Godot documentation.

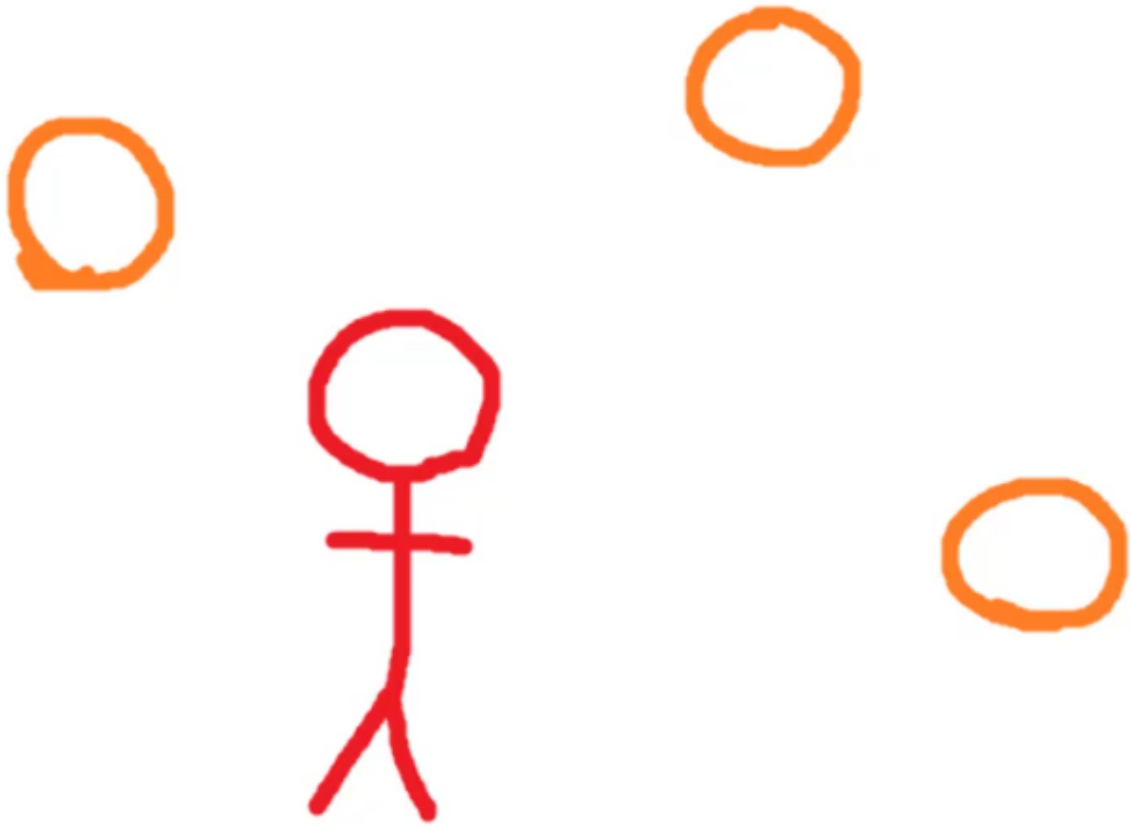
- Vector2
- Idle and Physics Processing

Welcome to our first complete project using the Godot game engine. In this tutorial, we will create a fun and engaging coin collector game combining all the skills we've previously learned. Let's dive into the details of our game and understand its mechanics.

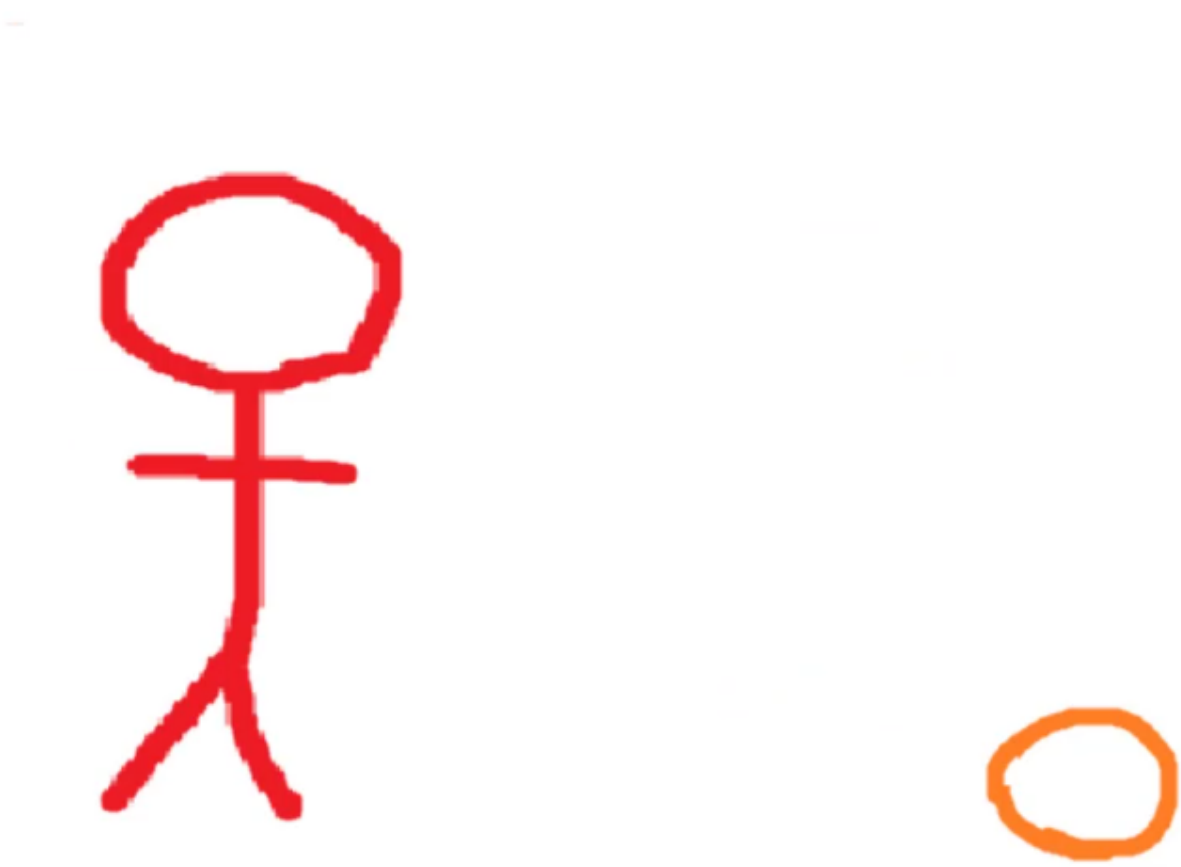
Game Overview

Our coin collector game will feature two primary components:

- **Player:** A character that can move around the level using arrow keys.
- **Coins:** Collectible items scattered throughout the level.



The objective of the game is for the player to collect coins. Each time a coin is collected, the player will increase in size, and the coin will be destroyed.



Game Mechanics

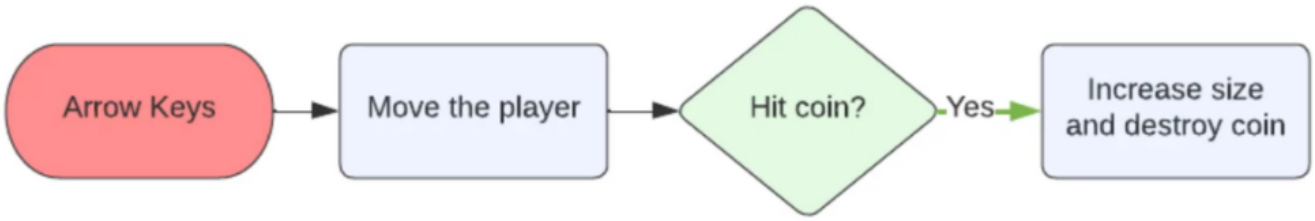
To break down the game mechanics more definitively, we will focus on the following key parts:

1. **Player Movement:** Create a player that can move around the level using arrow keys.
2. **Collectible Coins:** Design a coin scene that the player can collect.
3. **Collision Detection:** Implement a system where the player increases in size and the coin is destroyed upon collision.

Game Flow

Let’s outline the game flow using a simple flowchart:

1. The player presses the arrow keys to move.
2. The player’s position is updated based on the input.
3. Check if the player collides with a coin.
4. If a collision is detected:
    - Increase the player’s size.
    - Destroy the coin.



Next Steps

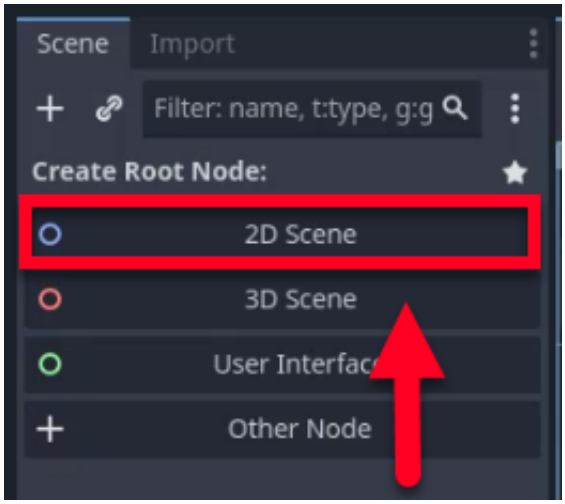
In the upcoming lesson, we will begin setting up our game. This will include creating the player character, designing the coin scene, and implementing the collision detection system.

Thank you for joining us on this exciting journey. Stay tuned for the next lesson where we will dive deeper into the development process. See you there!

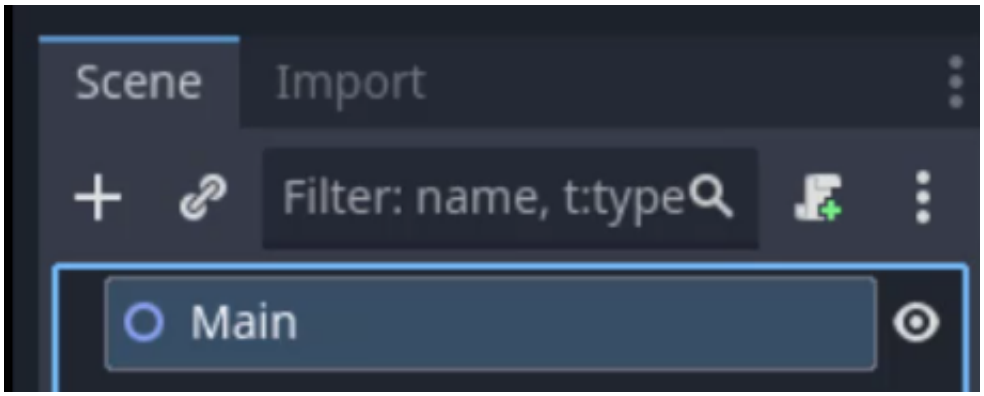
In this lesson, we will start creating a coin collector game. By the end of this lesson, you will have a player character that can move around the screen using the arrow keys. Let's get started!

Setting Up the Scene

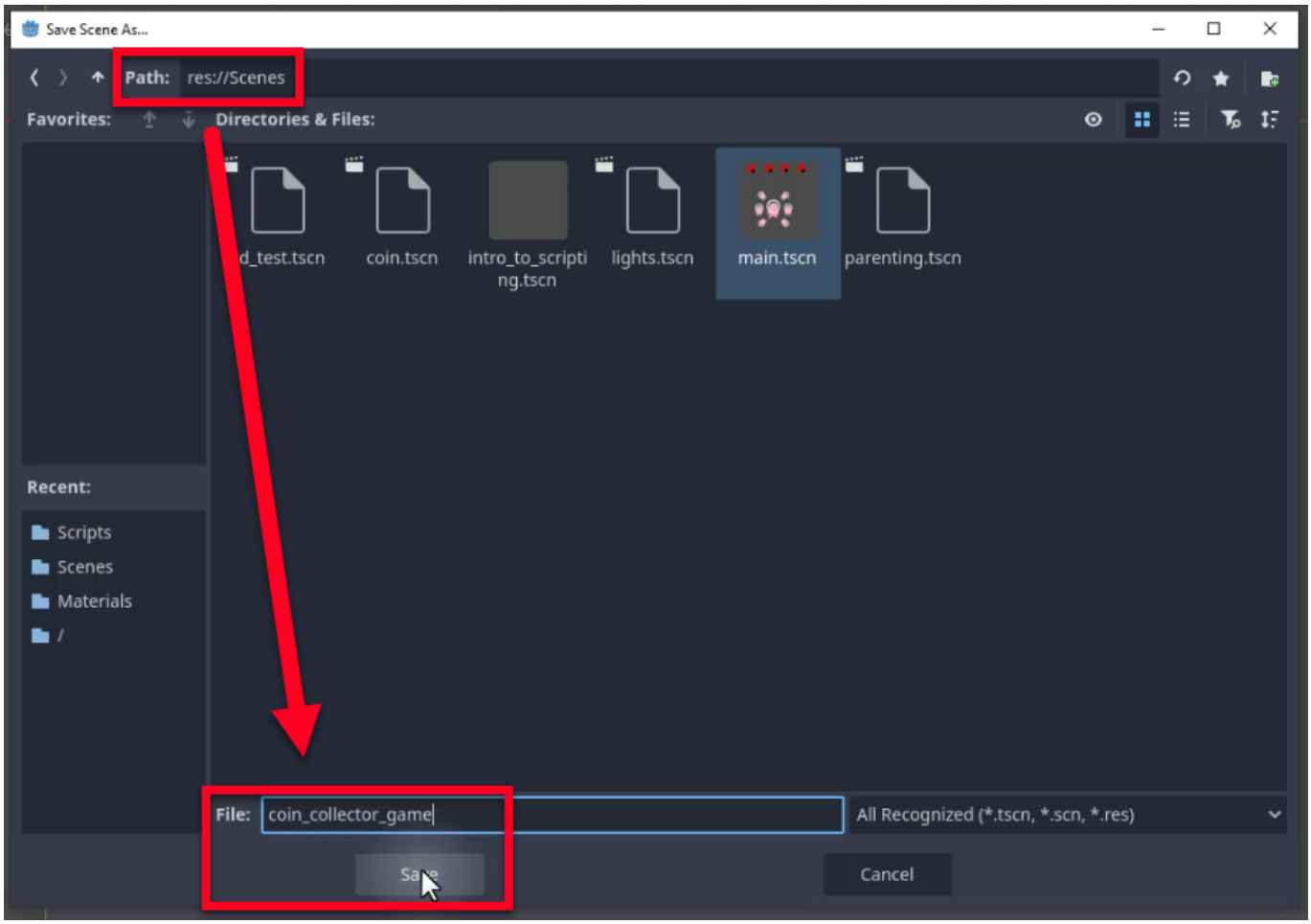
First, let's create a new scene for our game. Create a new scene and choose 2D Scene as the root.



Rename the root node to Main. This helps keep our scene organized.

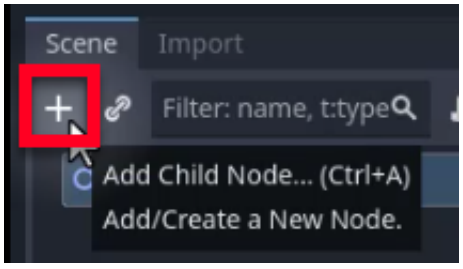


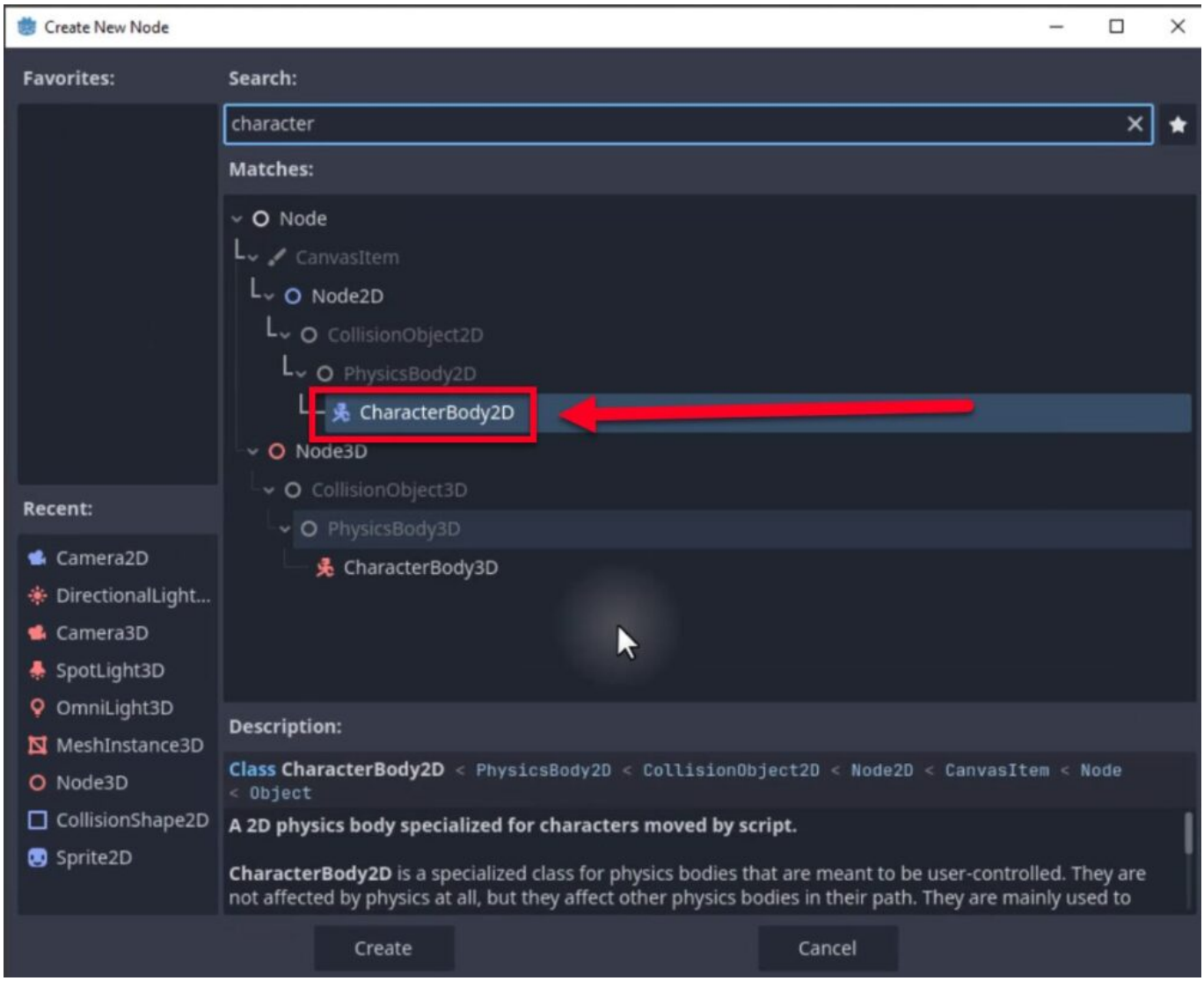
Save the scene in the Scenes folder as coin_collector_game.tscn.



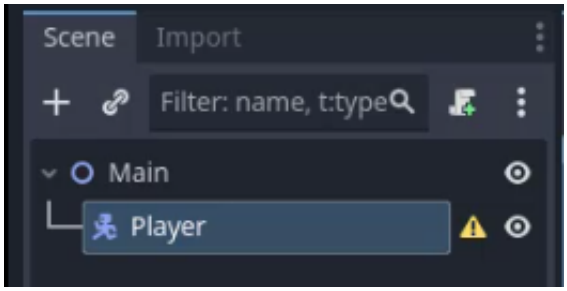
Adding the Player

Next, we'll add a player character that we can move around the level. Add a CharacterBody2D node as a child of the Main node. This node is specialized for user-controlled characters in a 2D physics environment.

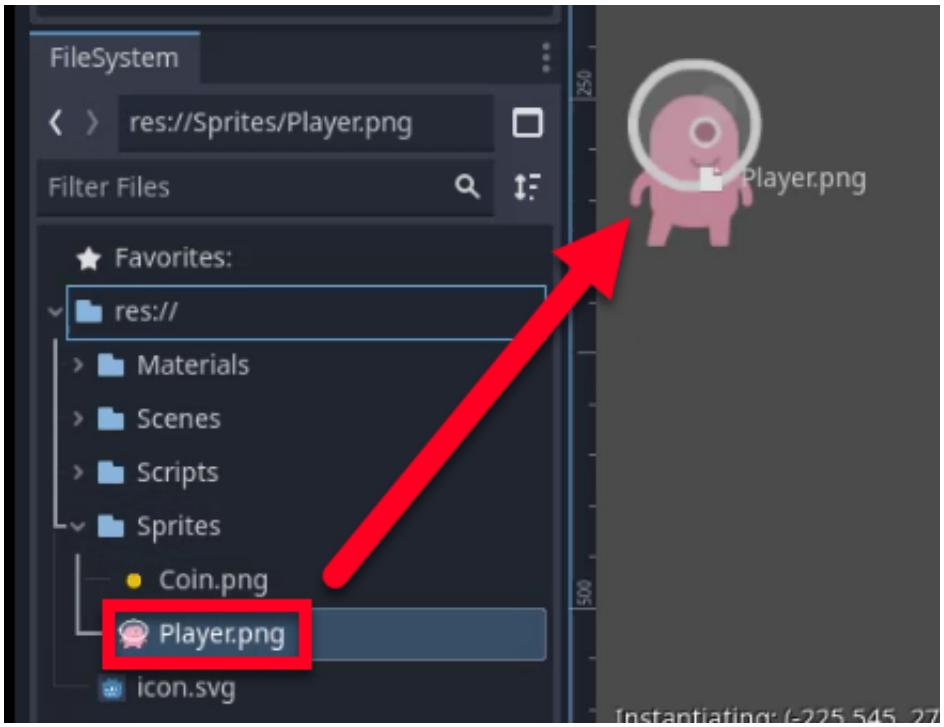




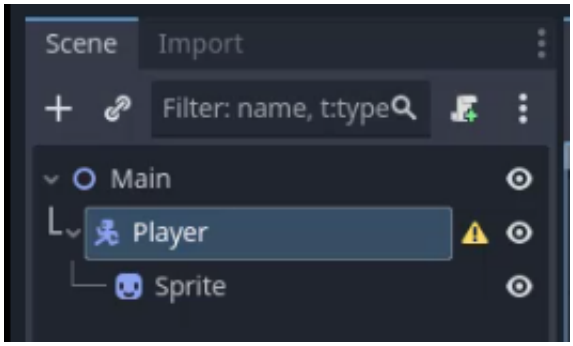
Rename the CharacterBody2D node to Player.



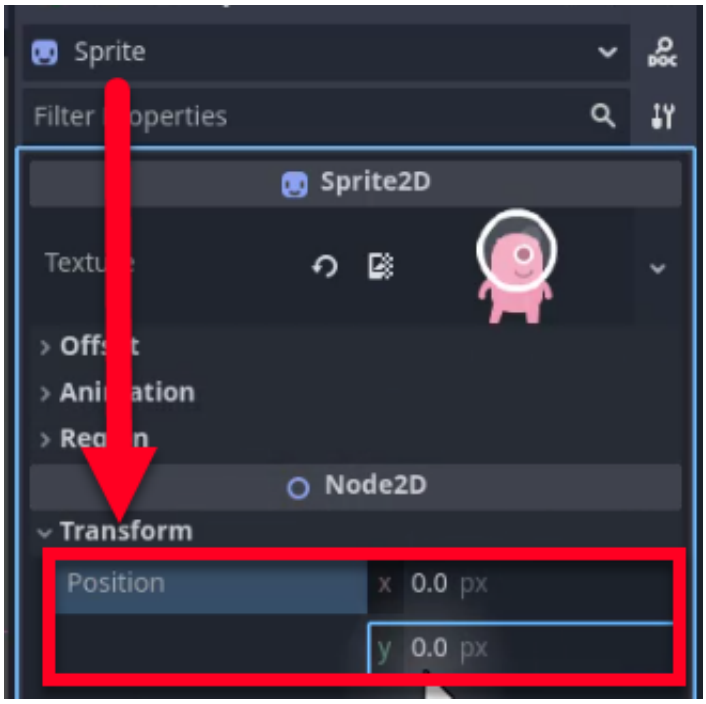
Add a Sprite2D node as a child of the Player node for the player's visuals. You can drag your player sprite from the Sprites folder onto the editor to have this node created automatically.



Rename the Sprite2D node to Sprite and make sure it is a child of the Player node.

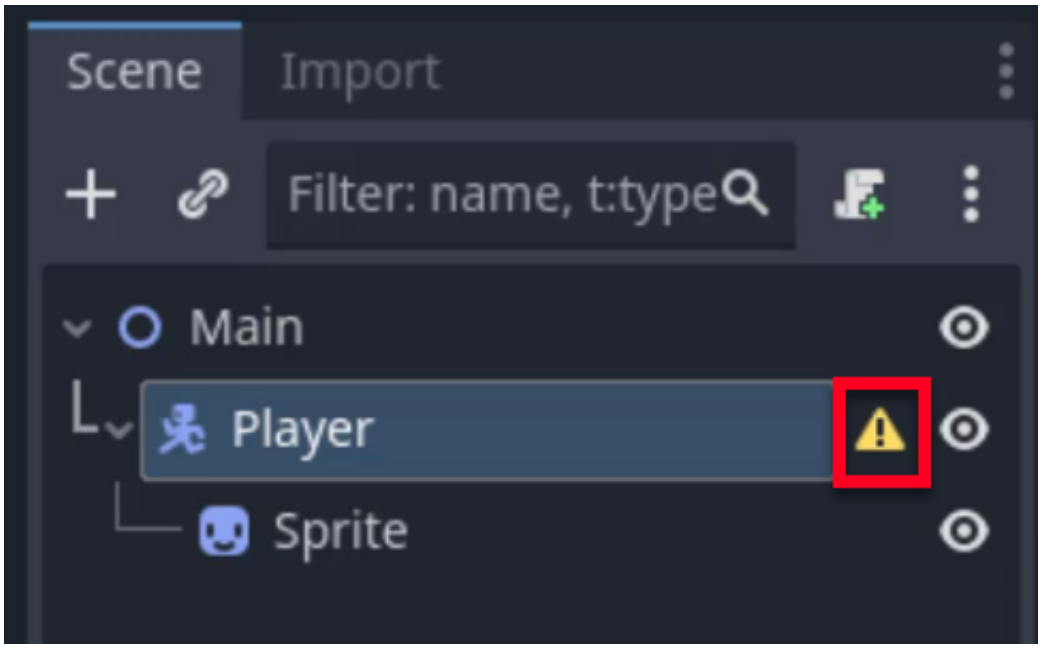


Ensure the sprite is centered on the player's origin by setting its position to (0, 0) in the Transform properties.

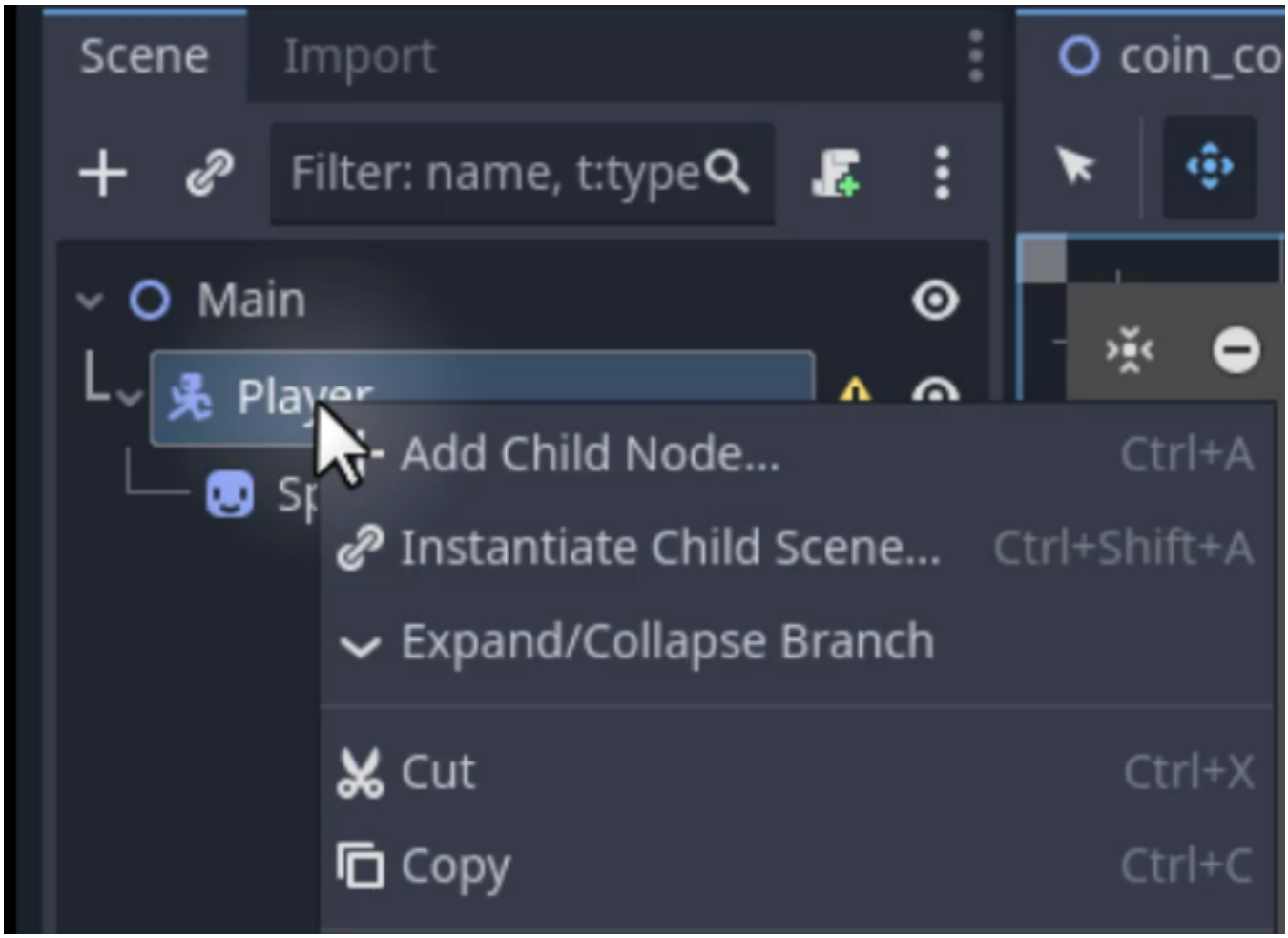


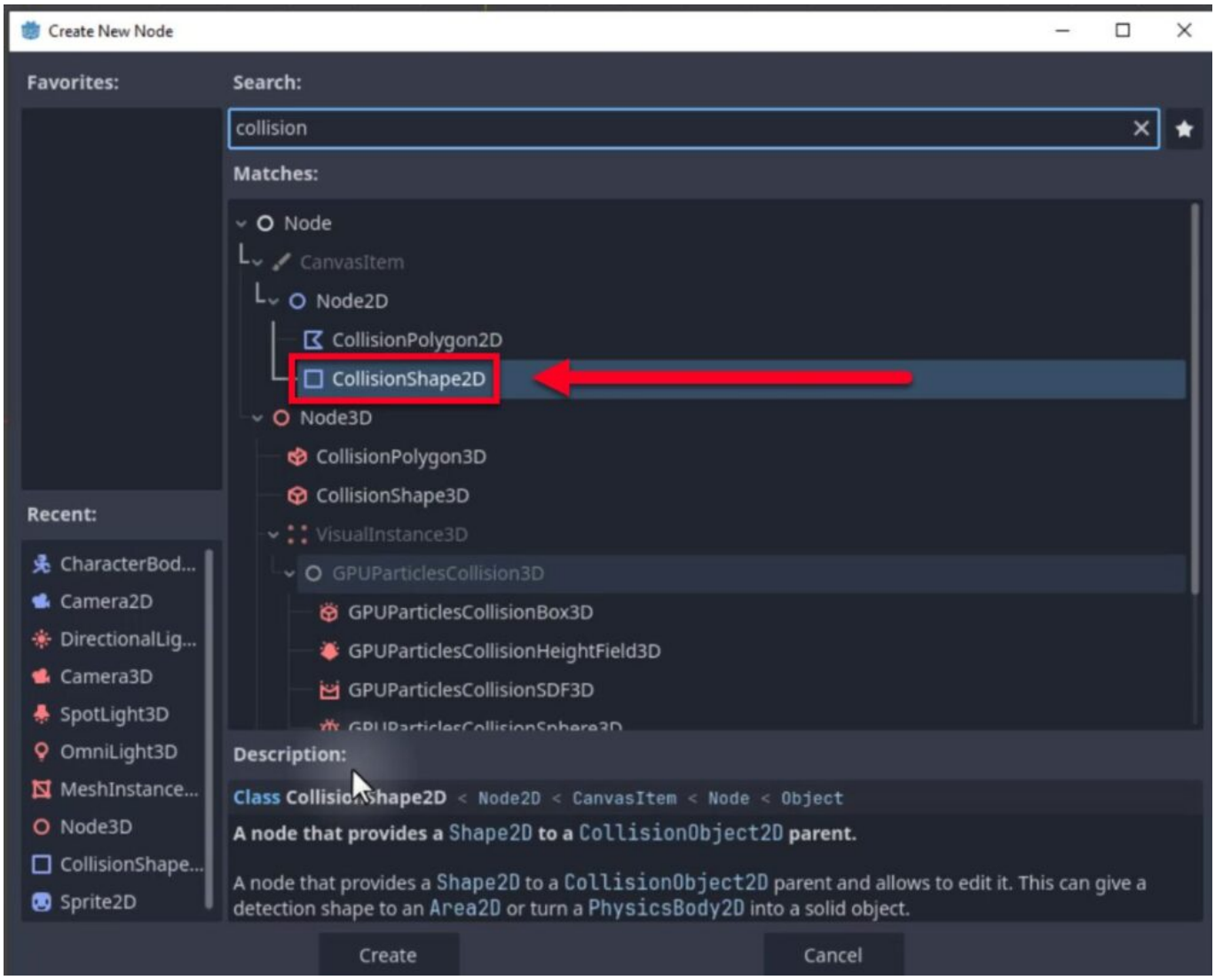
Adding a Collision Shape

Since we are using a CharacterBody2D, we need to add a collision shape (which Godot warns us about with the yellow icon next to the node in the scene tree.

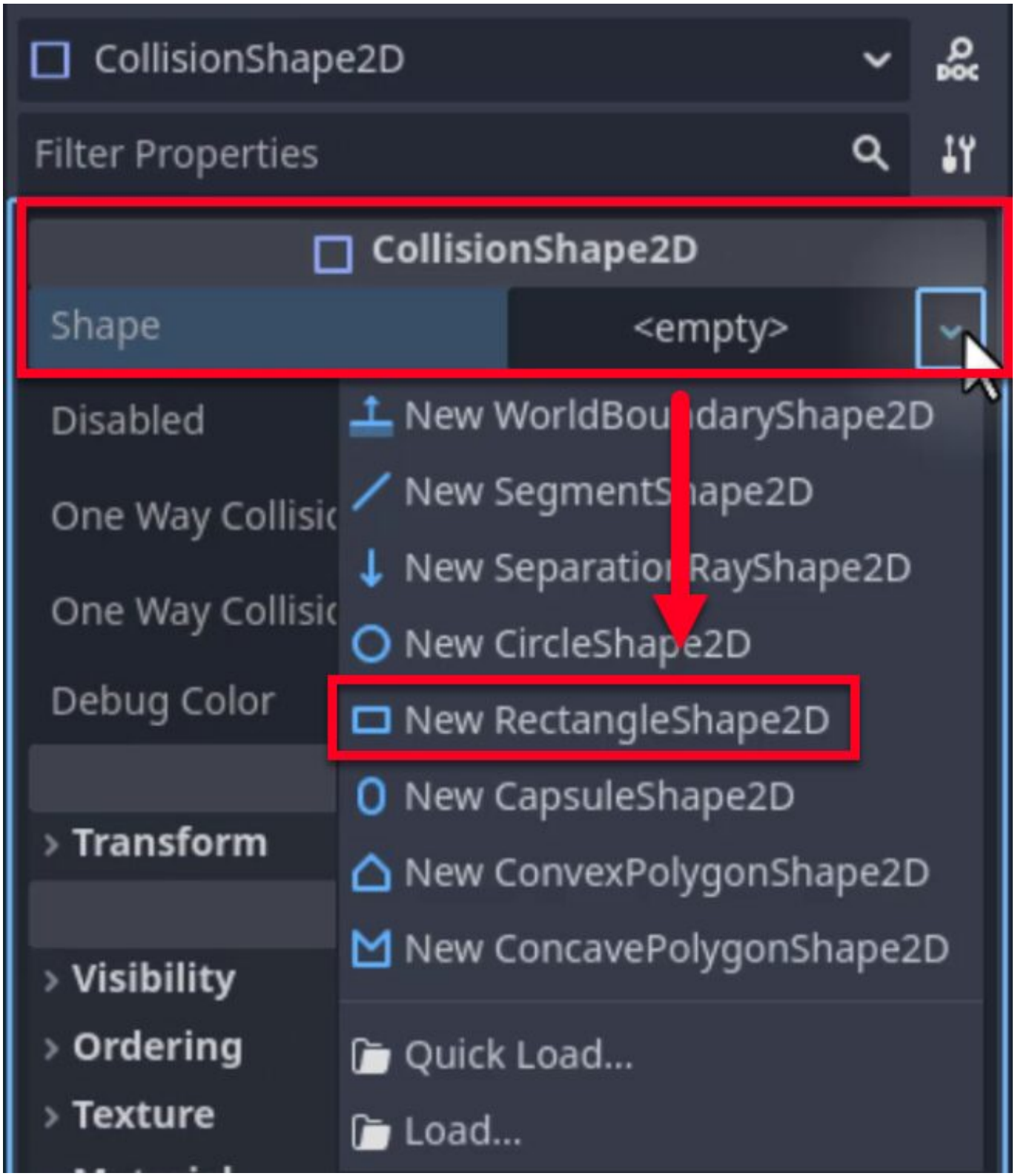


Add a CollisionShape2D node as a child of the Player node. Remember you can right-click a specific node and select the “Add Child Node” option to make the child directly on that node.

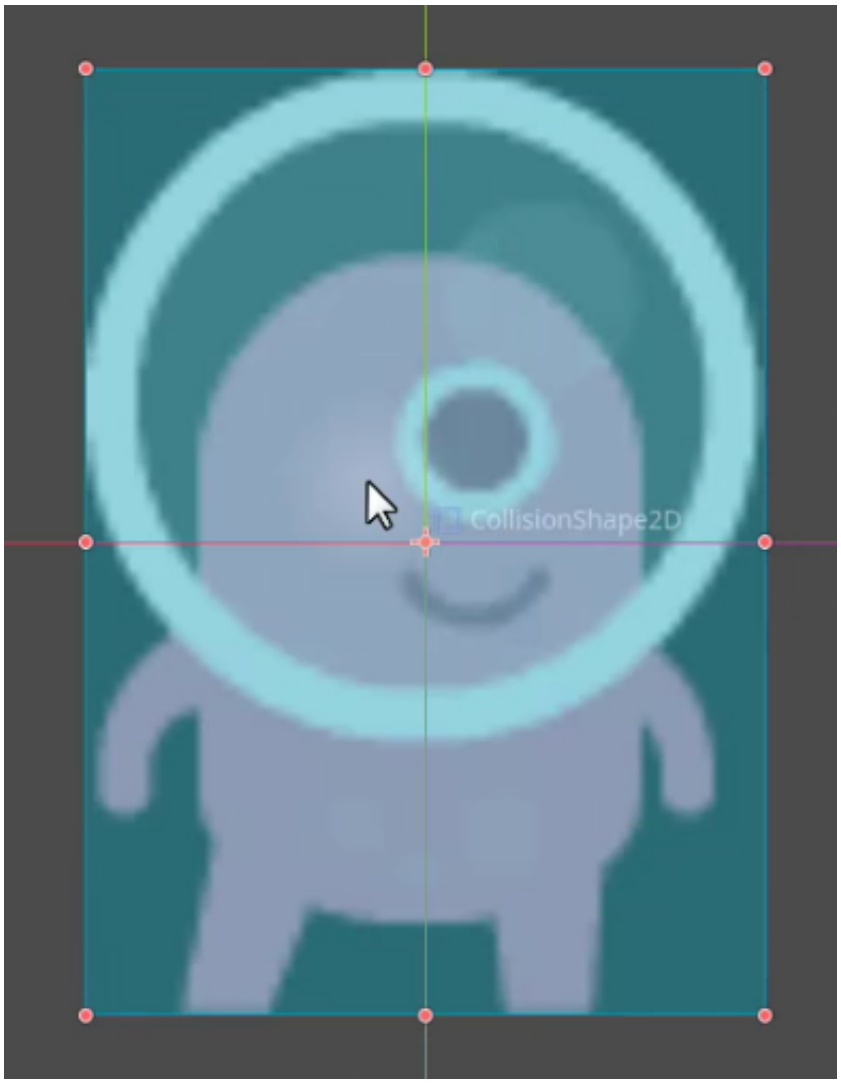




With the CollisionShape2D node selected, in the Inspector, set the Shape property to a new RectangleShape2D.

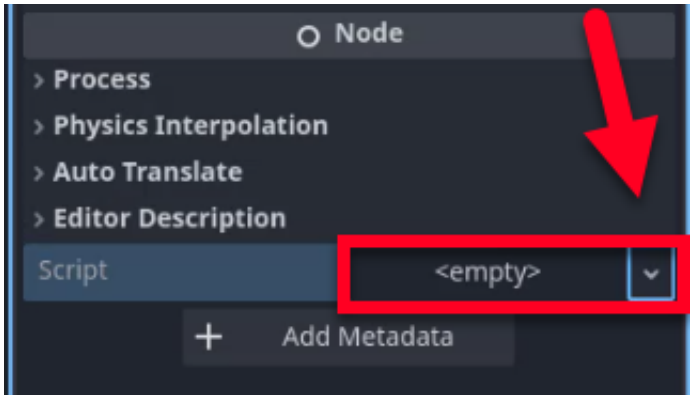


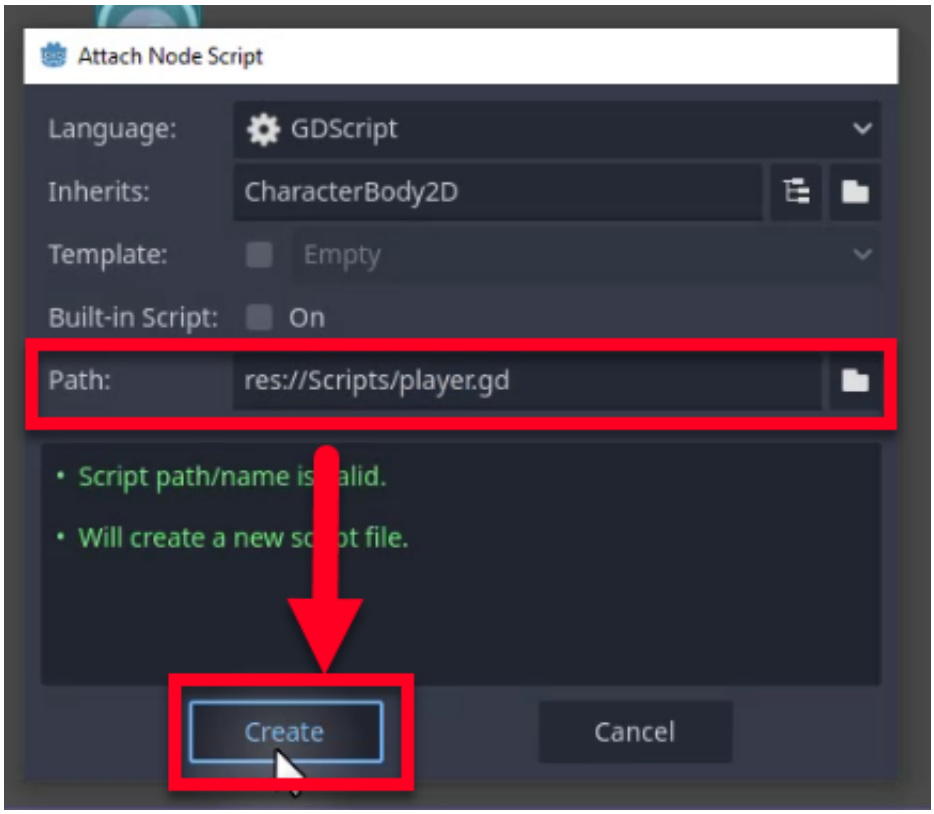
Adjust the rectangle shape to fit the size of your player sprite in the editor. You can hold down Alt while clicking and dragging the orange circles to resize from the center.



Creating the Player Script

Now, let’s create a script to control the player’s movement. Attach a new script to the Player node and save it in the Scripts folder as player.gd.





When opening the script for the first time, you'll see it extends the CharacterBody2D class.

```
extends CharacterBody2D
```

This is essential because CharacterBody2D provides built-in properties and methods like velocity and move_and_slide() that make movement and collision handling much easier. You'll see below how we use them.

Define a Speed Variable

The first step in our script is to define a variable to control how fast the player moves.

```
var speed: float = 100
```

This variable stores the player's movement speed in pixels per second. You can adjust this value later to make the player move faster or slower. Using the float type ensures we're working with a decimal number, which is ideal for precise control over speed.

Use the _physics_process() Function

Movement in Godot is typically handled in either the _process() or _physics_process() function. While both run every frame, _physics_process() is specifically designed for physics-related updates. It runs at a fixed rate of 60 times per second, regardless of your frame rate, ensuring smooth and consistent movement.

Add the following function to your script:


```
func _physics_process(delta):
    pass
```

We'll build the function step by step in the following sections.

Reset the Velocity

The first thing we need to do in the _physics_process() function is reset the player's velocity at the start of every frame:

```
velocity.x = 0
velocity.y = 0
```

The velocity property is a Vector2 (a two-dimensional vector) that defines the player's speed along the X and Y axes. Resetting it ensures that movement doesn't "stack up" and cause the player to move uncontrollably if keys are pressed repeatedly.

Handle Arrow Key Input

Once we've reset the velocity, we can modify it based on the player's input. Godot provides the Input.is_key_pressed() method to check if a specific key is being pressed. For each arrow key, we adjust the velocity accordingly:

- **Right Arrow Key (KEY_RIGHT)**: Increase the X velocity to move right.
- **Left Arrow Key (KEY_LEFT)**: Decrease the X velocity to move left.
- **Up Arrow Key (KEY_UP)**: Decrease the Y velocity to move up (in 2D, moving up is a negative Y direction).
- **Down Arrow Key (KEY_DOWN)**: Increase the Y velocity to move down.

Here's how the input-handling code looks:

```
if Input.is_key_pressed(KEY_RIGHT):
    velocity.x += speed
if Input.is_key_pressed(KEY_LEFT):
    velocity.x -= speed
if Input.is_key_pressed(KEY_UP):
    velocity.y -= speed
if Input.is_key_pressed(KEY_DOWN):
    velocity.y += speed
```

Move the Player

To make the player move based on the velocity we've defined, call the move_and_slide() function at the end of _physics_process(). This function updates the player's position and handles collisions automatically.

```
move_and_slide()
```


The move_and_slide() function ensures that the character interacts properly with walls and other objects in the scene. It relies on the current value of velocity to determine movement, making it the final step in building our movement logic.

Complete Script

Now that we've walked through each step, here's how the complete player.gd script should look:

```
extends CharacterBody2D

var speed: float = 100  # Speed of the player in pixels per second

func _physics_process(delta):
    # Reset velocity at the start of every frame
    velocity.x = 0
    velocity.y = 0

    # Handle arrow key input
    if Input.is_key_pressed(KEY_RIGHT):
        velocity.x += speed
    if Input.is_key_pressed(KEY_LEFT):
        velocity.x -= speed
    if Input.is_key_pressed(KEY_UP):
        velocity.y -= speed
    if Input.is_key_pressed(KEY_DOWN):
        velocity.y += speed

    # Apply velocity to move the player and handle collisions
    move_and_slide()
```

Why No Multiplication by delta?

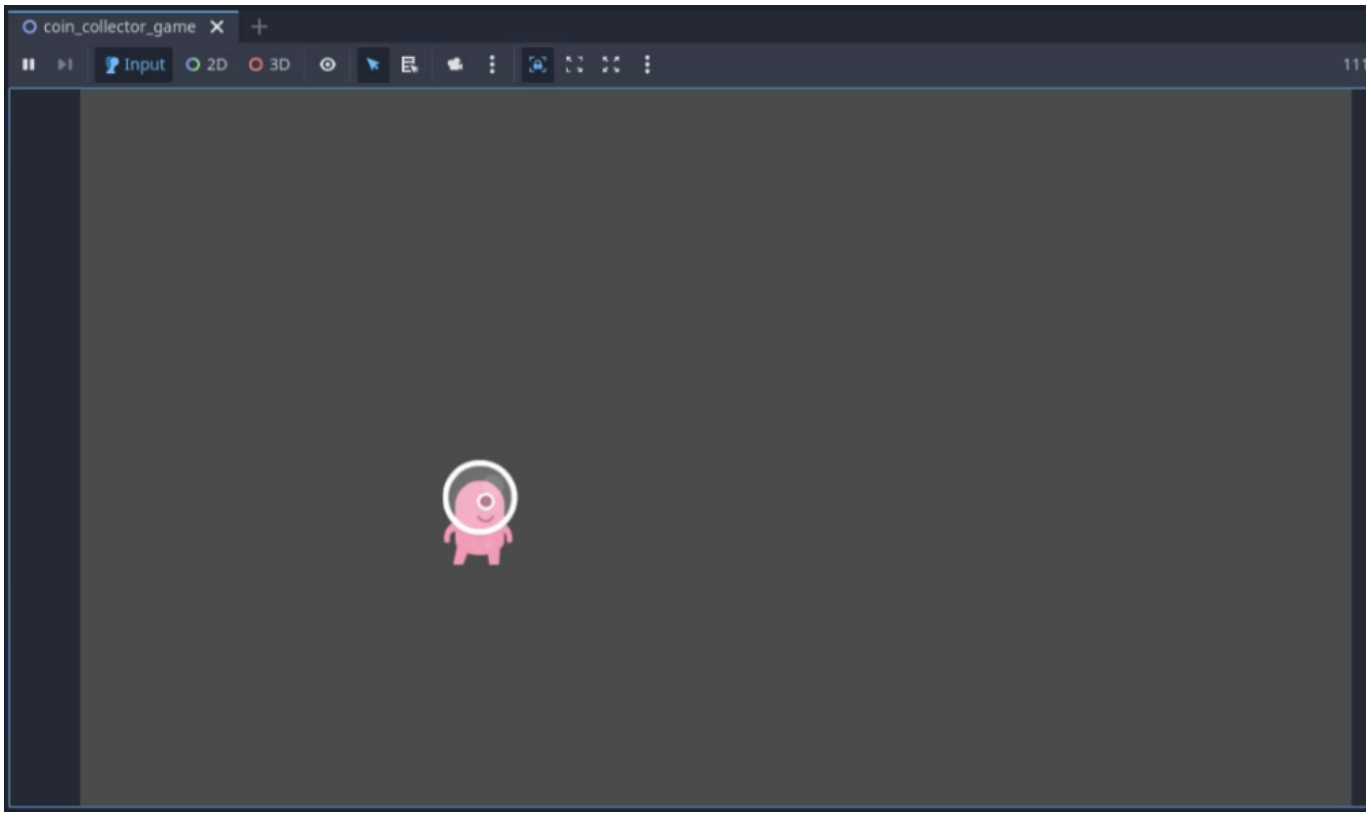
In many physics-based scripts, we multiply by delta to make movement consistent across different frame rates. However, in this case, the CharacterBody2D node already interprets velocity as pixels per second. This means it internally handles the time calculations for consistent movement, so no manual delta adjustment is needed.

Testing Your Player Movement

Save your script and press **Play**. Use the arrow keys to move your player character around the screen. If everything is working correctly:

- Pressing the **Right** arrow makes the player move right.
- Pressing the **Up** arrow makes the player move up.
- You should be able to move in all four directions smoothly.

If the movement feels too fast or slow, try adjusting the speed variable.

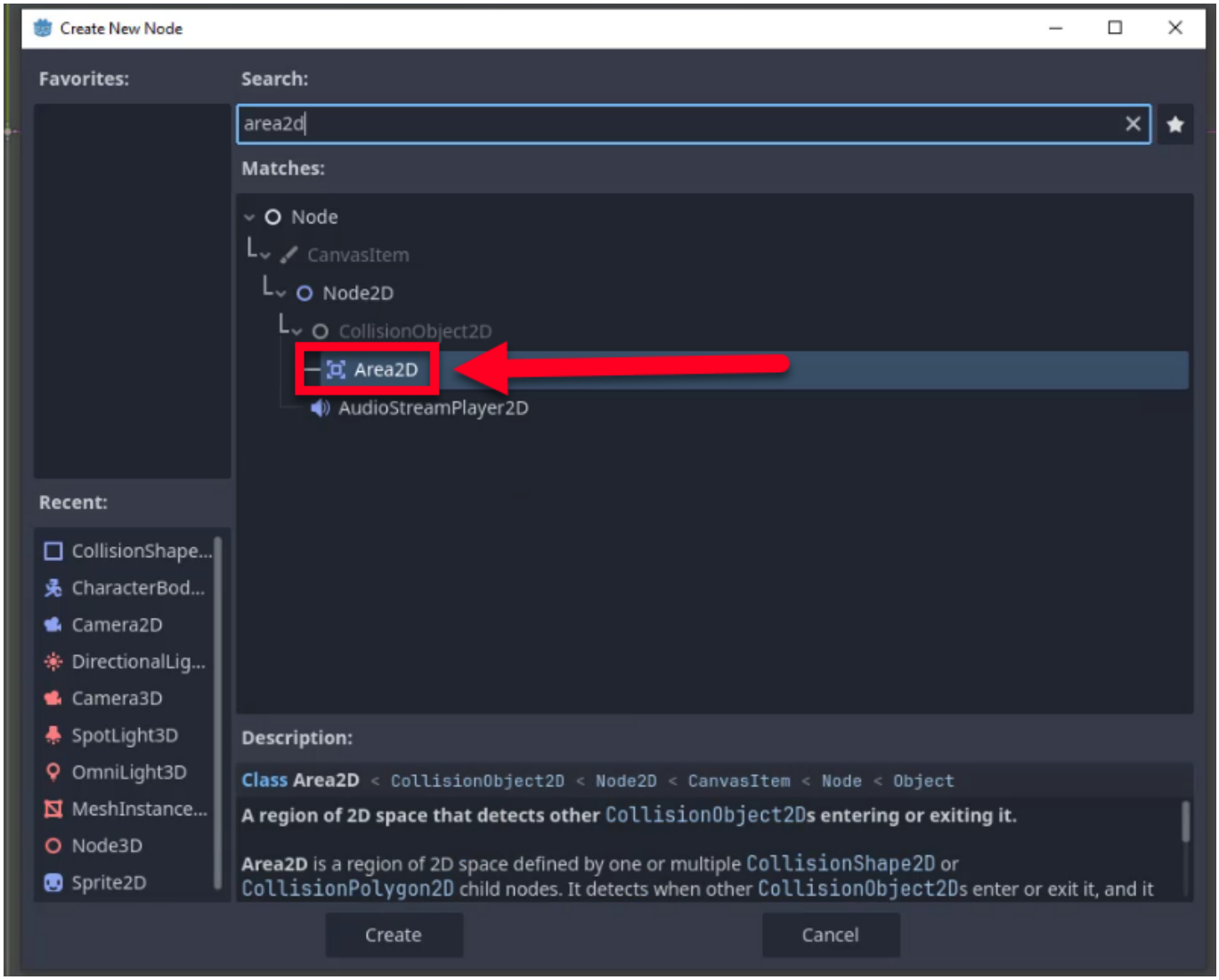


With this basic movement script complete, your player character can now move around using the arrow keys. In the next lesson, we'll add collectibles to our game to make it more interactive – stay tuned!

Welcome back to our course on creating a coin collector game using Godot. In this lesson, we will focus on creating the coin that the player will collect. We will use an Area2D node to detect when the player overlaps with the coin, and then write a script to handle the collection logic.

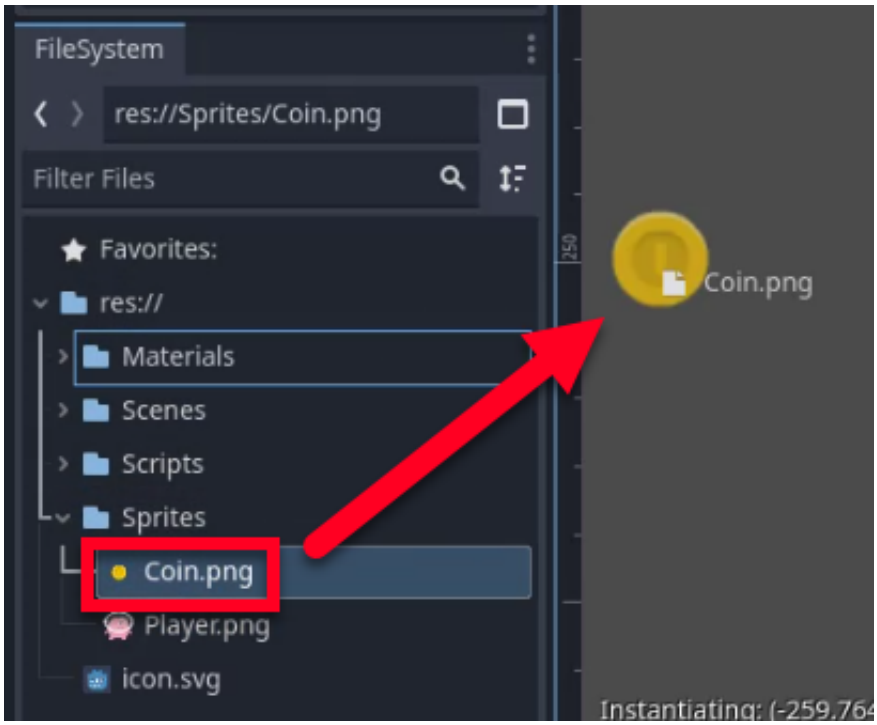
Creating the Coin

Let's first create the coin scene itself. In your scene, create a new node of type Area2D.

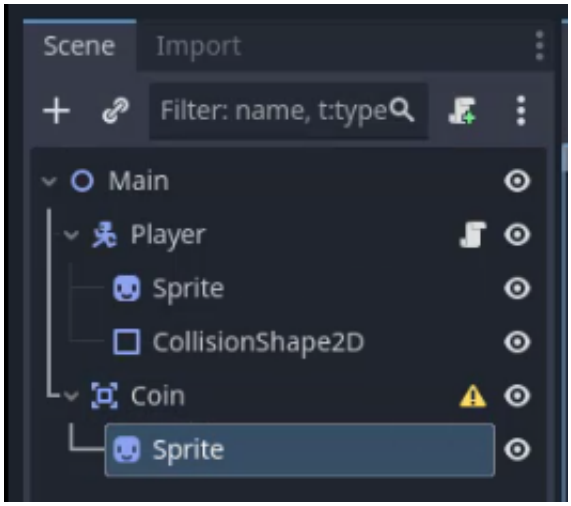


Rename this Area2D node to Coin.

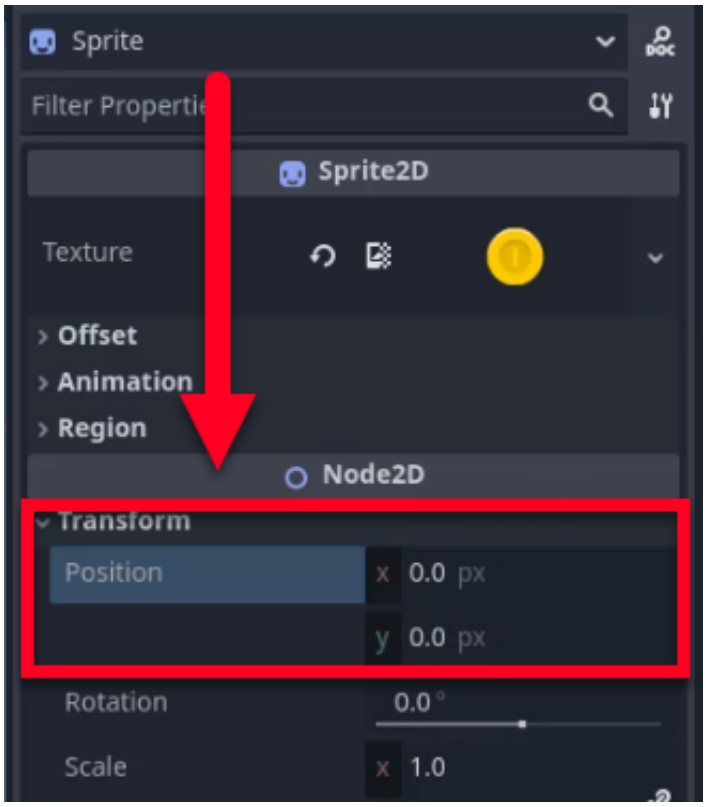
Drag coin.png from the Sprites folder into the scene to automatically create a sprite node.



Make it a child of the Coin node and rename the sprite node to Sprite.

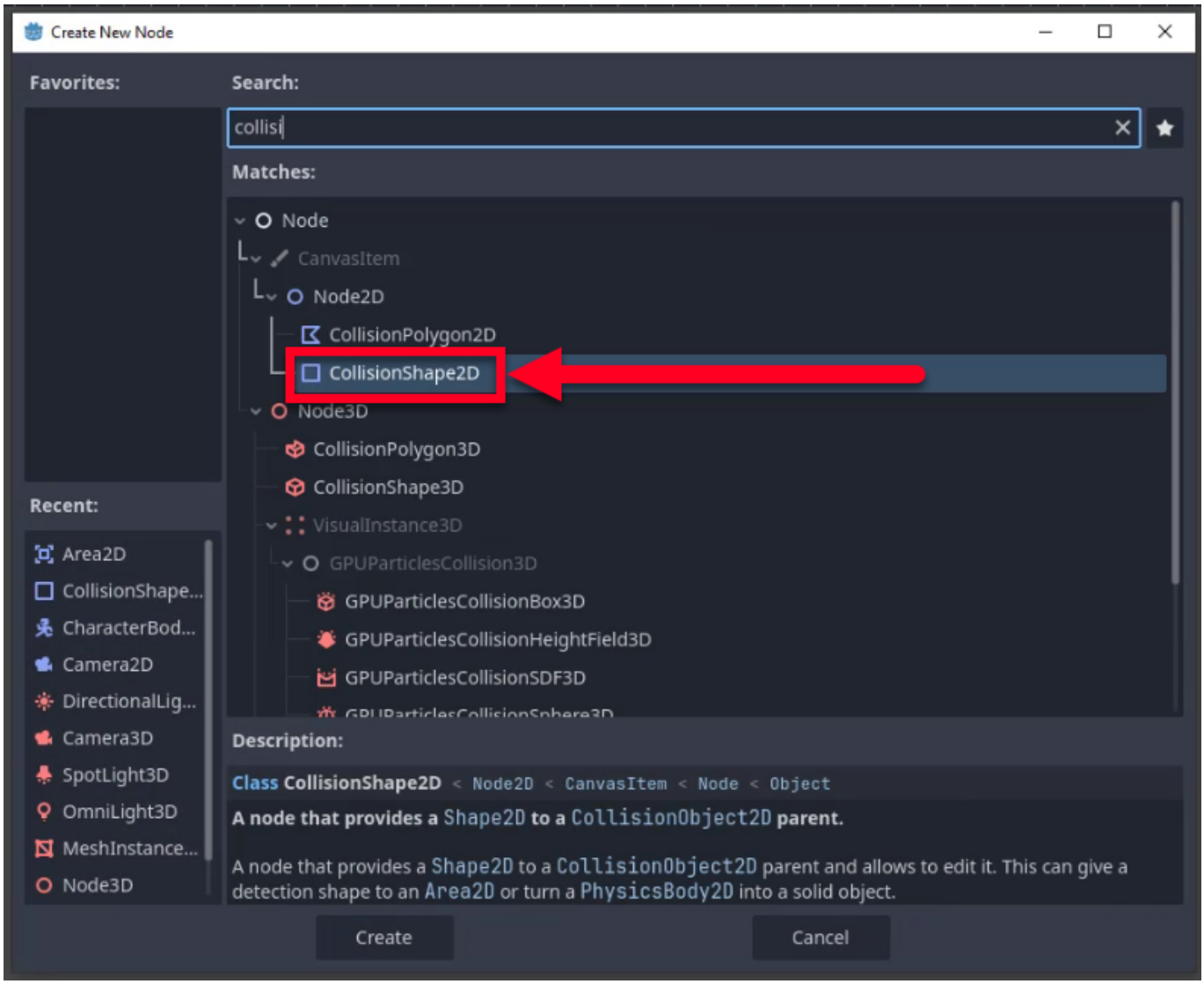


Similar to the Player's sprite, in the inspector, set the position of the Sprite to center it within the Coin node.

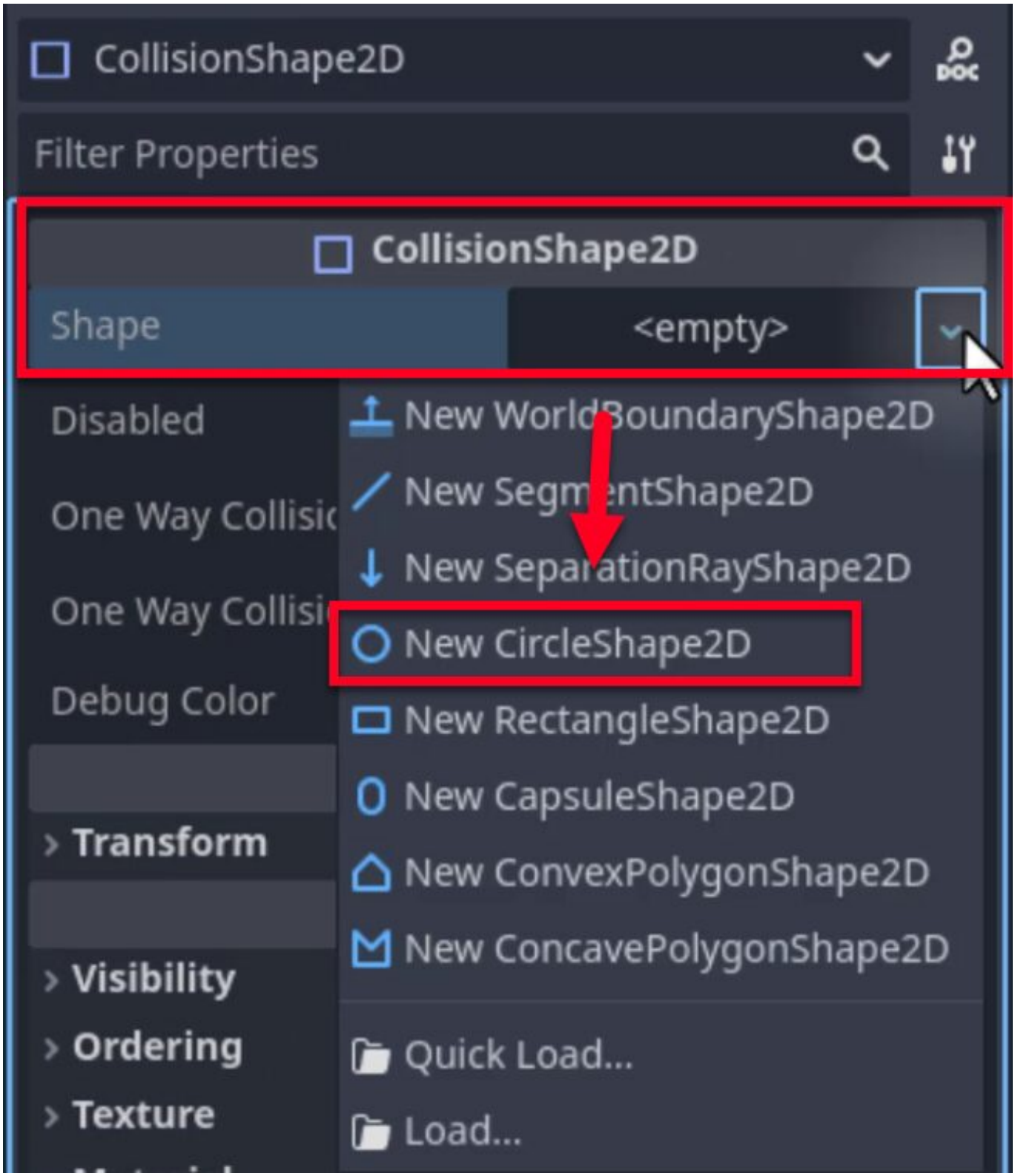


Adding a Collider

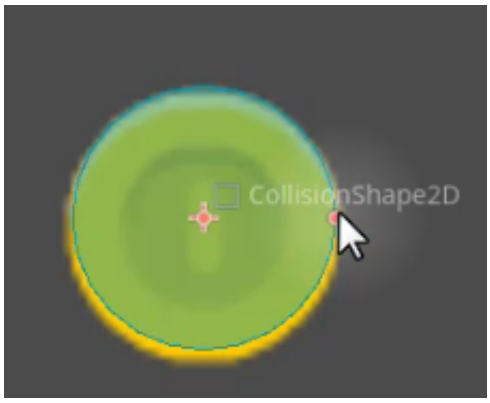
Next, we need to add a collider to the Coin node so that it can detect when the player overlaps with it. Add a CollisionShape2D node as a child of the Coin node.



In the Inspector, change the shape of the collider to a circle to match the shape of the coin.

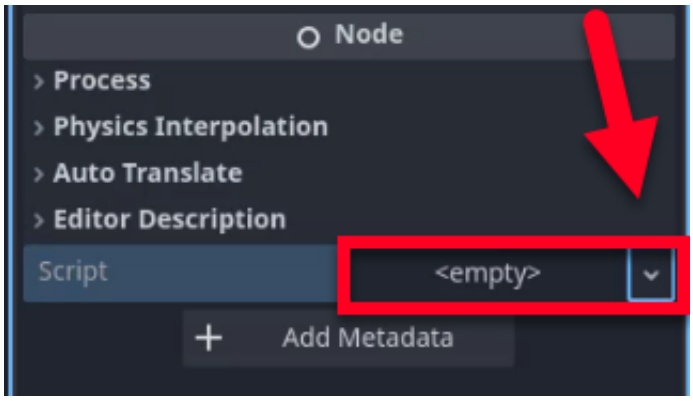


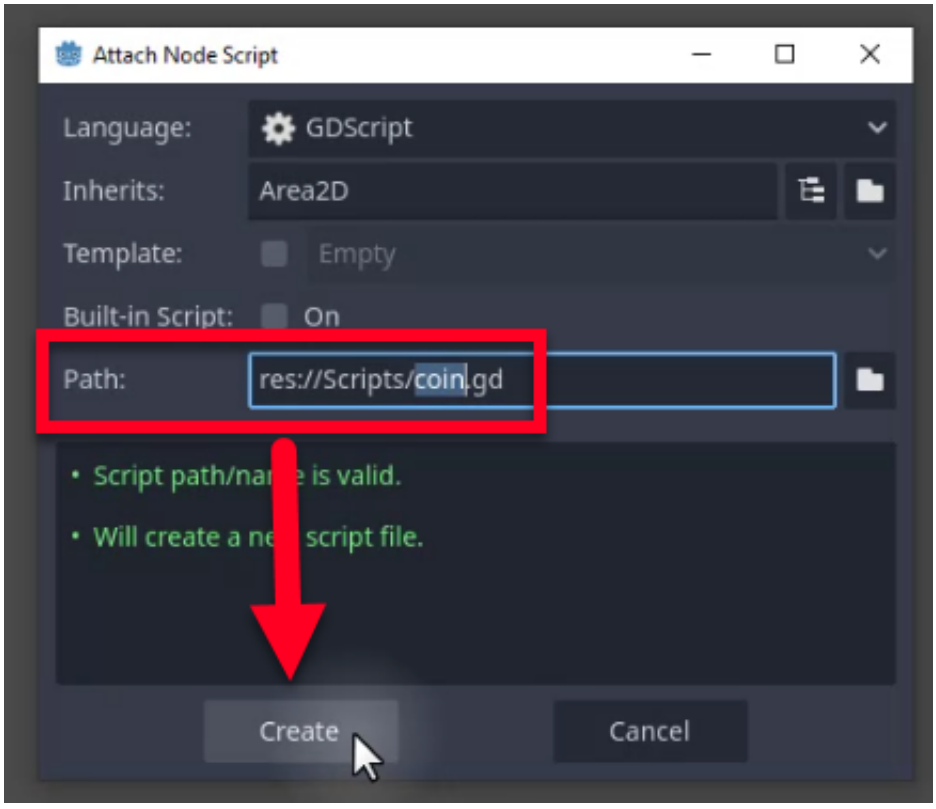
Adjust the size of the circle to fit the coin sprite in the editor.



Creating the Coin Script

Now, we will create a script to handle the logic for when the player collects the coin. Select the Coin node and create a new script named coin.gd in the scripts folder.



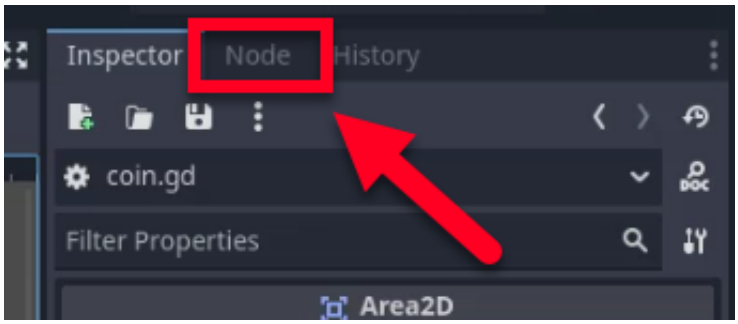


Connect the body_entered Signal

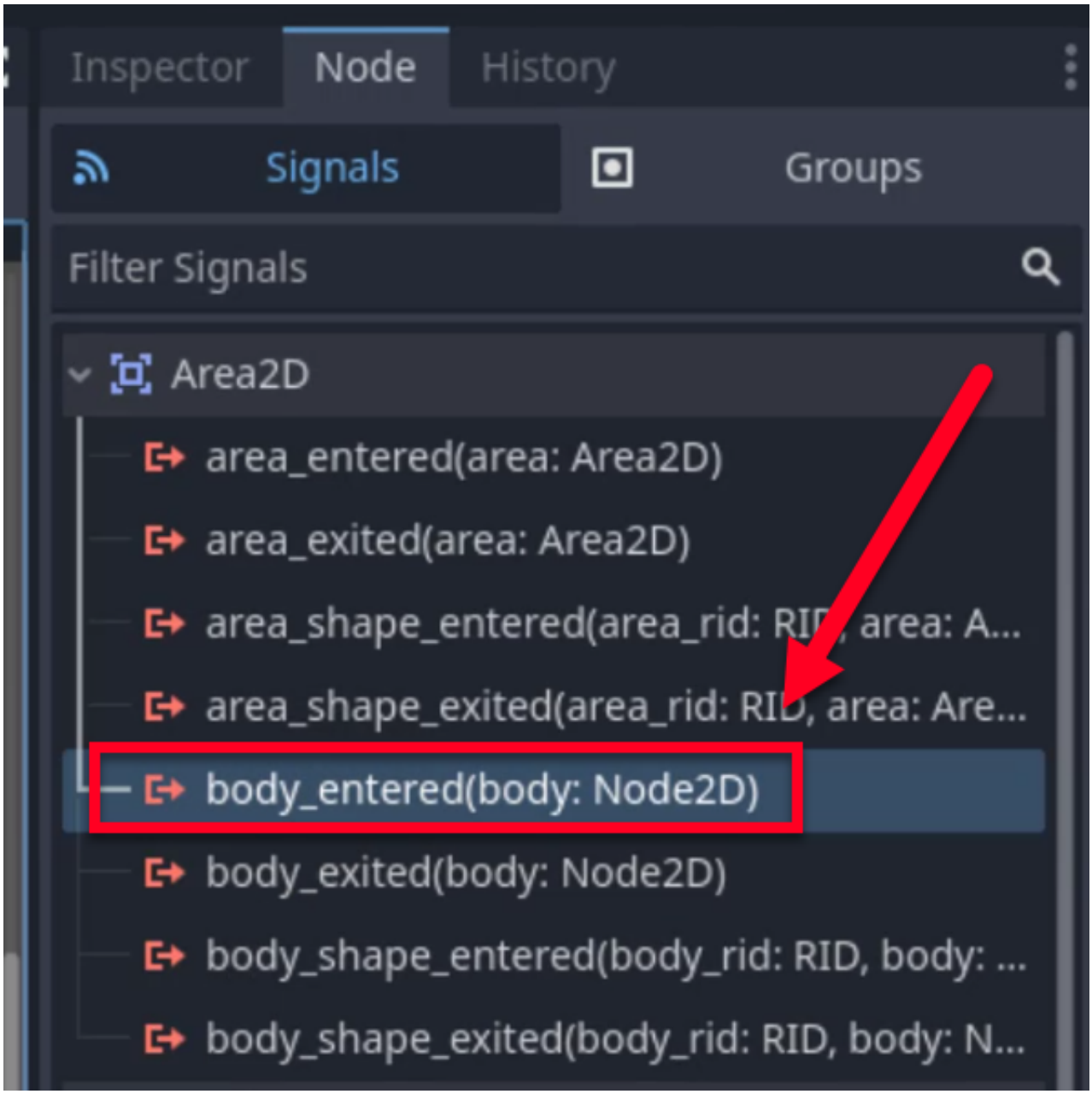
Right now, if you press play, the player can move through the coin, but nothing happens. That's because we haven't told the script to react to the player's presence. To fix this, we'll use **signals**, a powerful feature in Godot.

A signal is like an event that gets emitted when something happens. For example, the body_entered signal is emitted when another physics body, such as the player, enters the coin's collider. We can connect this signals directly from Godot.

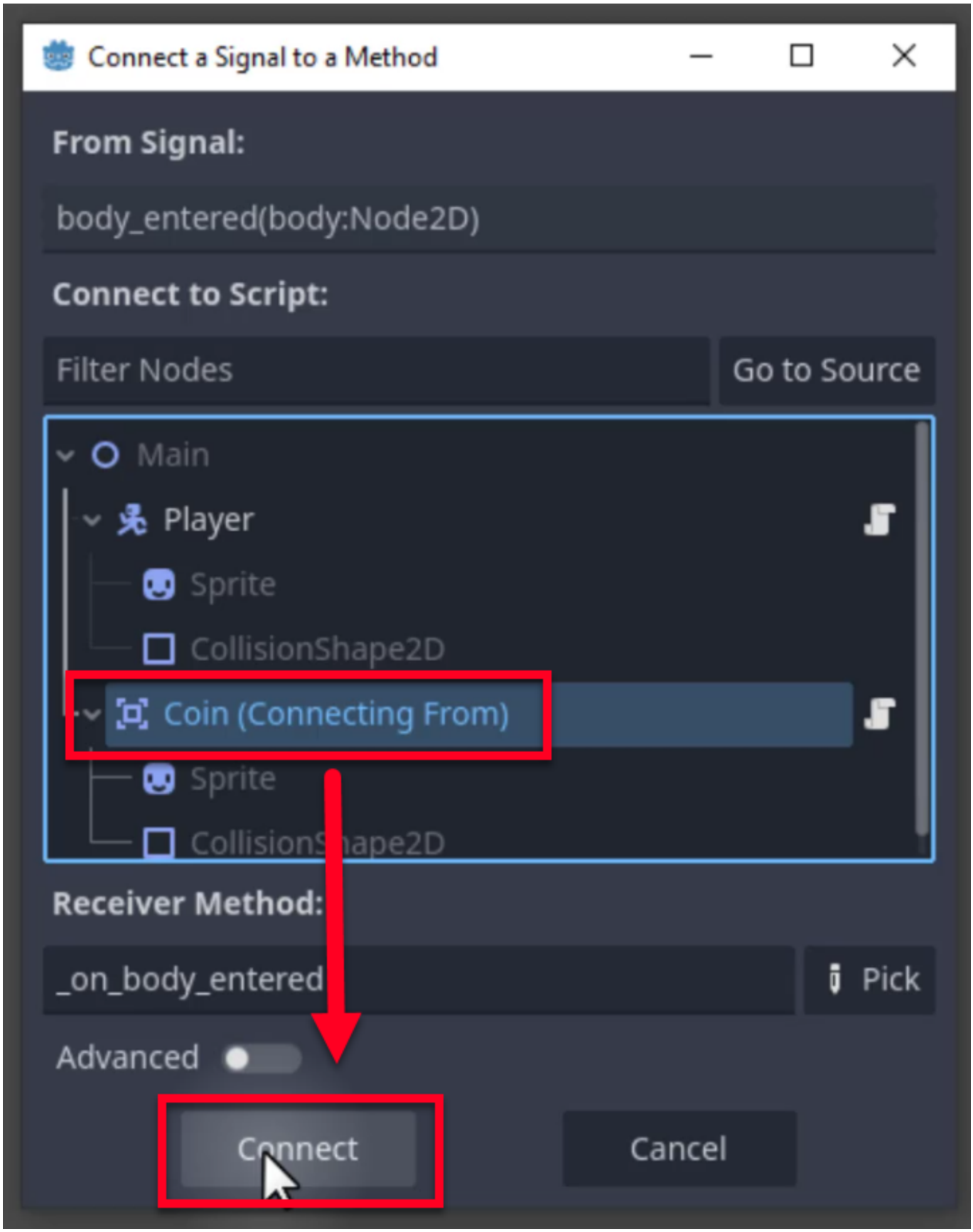
First, click on the Coin node and go to the **Node** tab in the *Inspector* dock area.



Find the body_entered signal in the list and double-click it.



Select the Coin node in the dialog box that appears (this is where we'll handle the signal) and click **Connect**.



This will automatically create a new function in the coin's script called _on_body_entered(body). Now, we'll write the logic for what happens when the player collects the coin.

Implement the Coin's Functionality

In the new _on_body_entered function, we'll do two things:

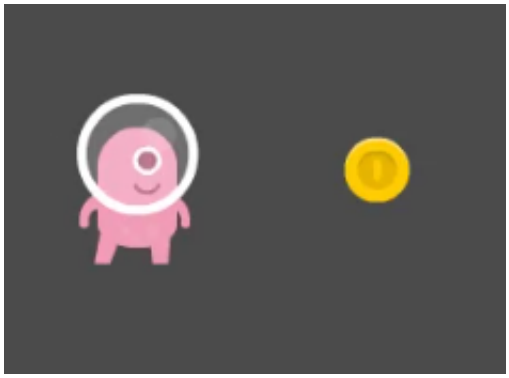
- Increase the size of the player by modifying its scale.
- Remove the coin from the scene using the queue_free() function.

Here's how the function looks:

```
func _on_body_entered(body):
    # Increase the player's size
    body.scale.x += 0.2
    body.scale.y += 0.2

    # Remove the coin
    queue_free()
```

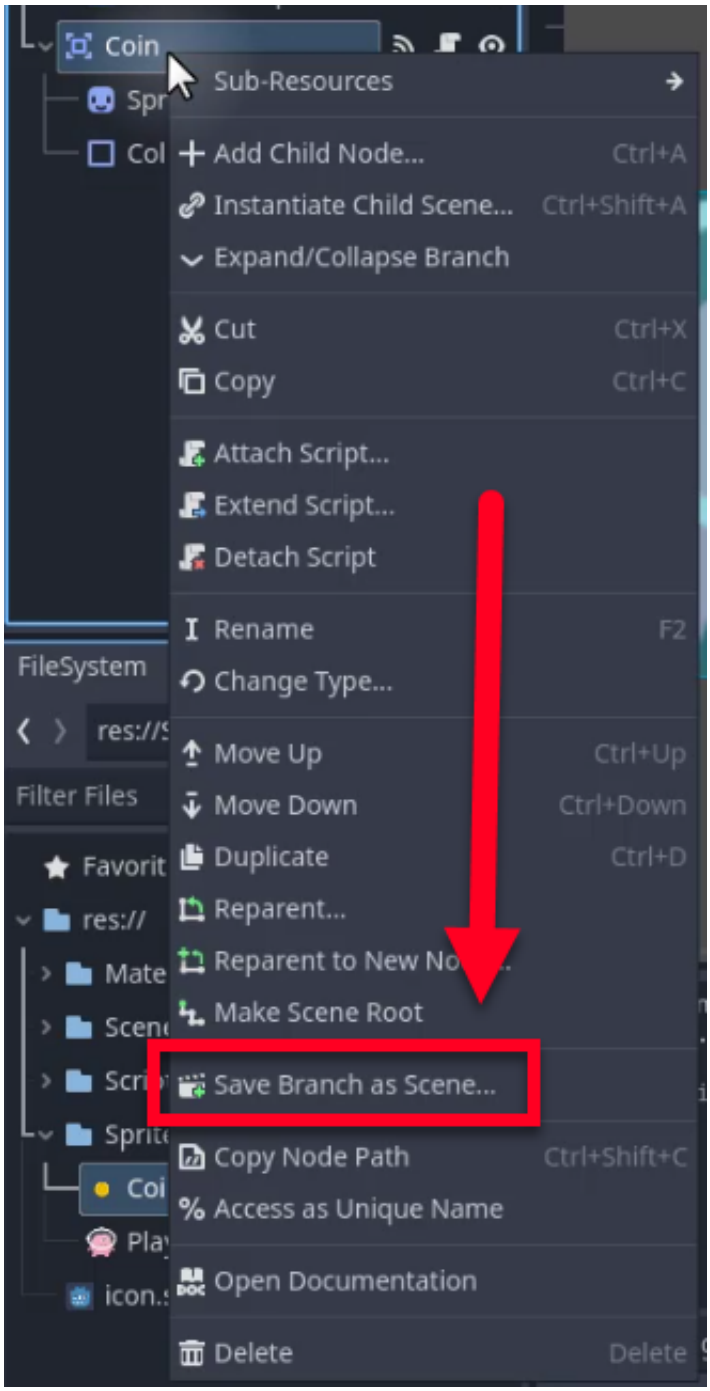
The body parameter represents the node that entered the coin's collider—in this case, the player. By accessing the player's scale property, we increase its size by 0.2 on both the X and Y axes. The queue_free() function then removes the coin from the scene.

Test your game to ensure that when the player touches the coin, the player grows larger and the coin disappears.

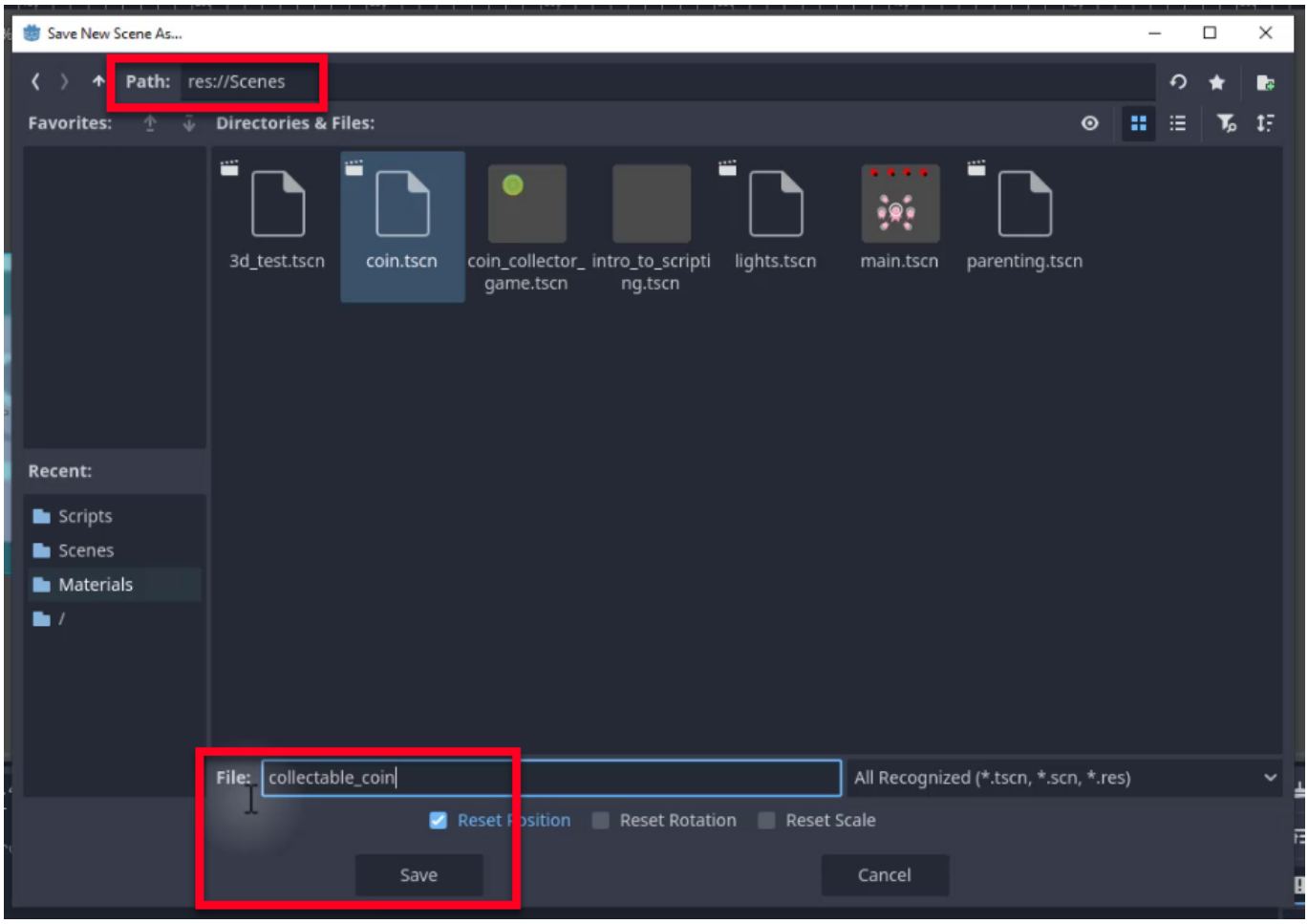


Save the Coin as a Scene

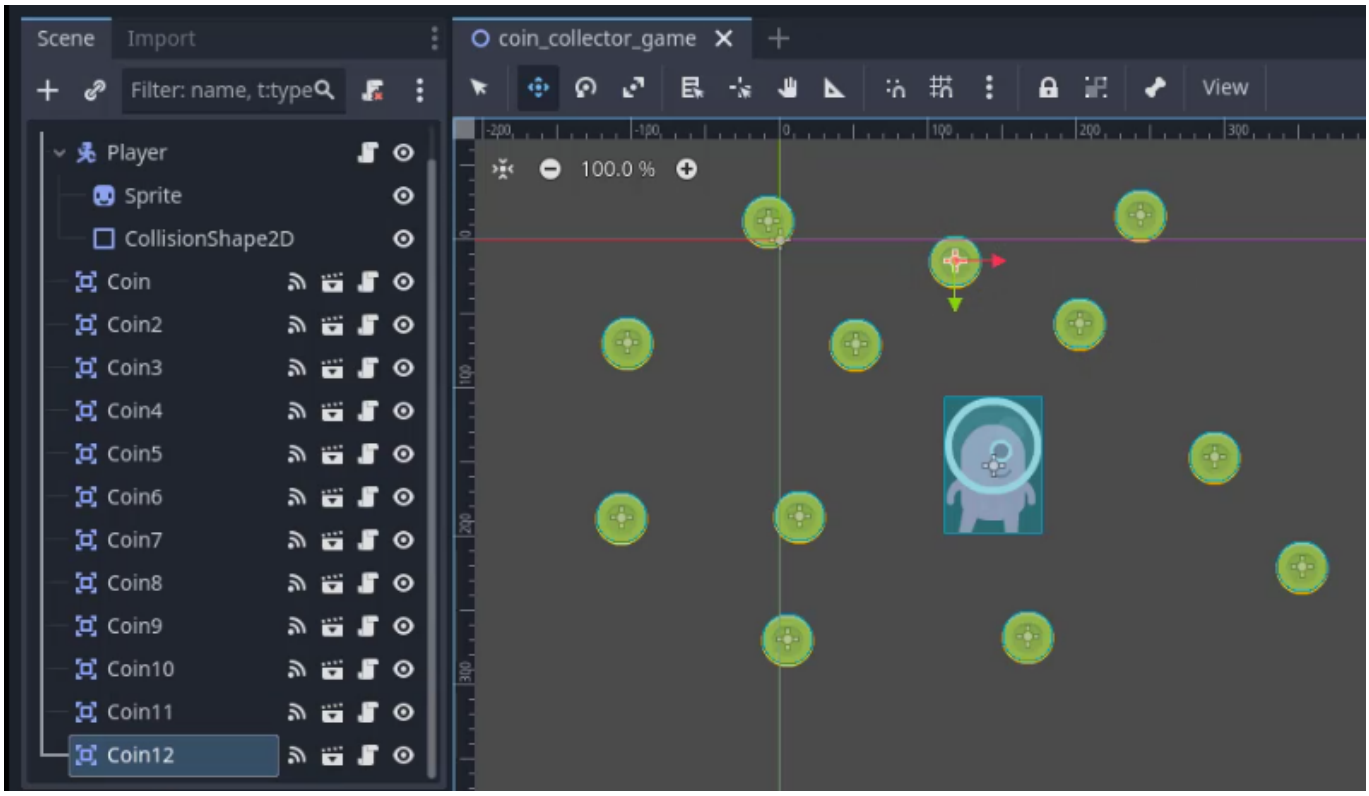
To make it easier to reuse the coin, we'll save it as its own scene. This allows us to duplicate it throughout the game. Right-click on the Coin node in the *Scene* tab and select **"Save Branch as Scene"**.



Save the scene as collectible_coin.tscn in your Scenes folder.

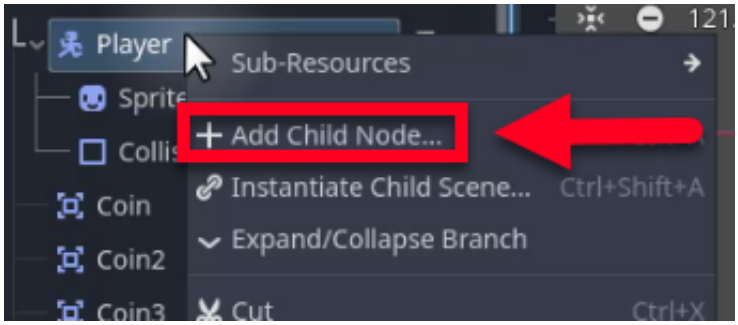


Now, you can drag and drop the coin scene into your game to place additional coins wherever you like. Test again to ensure the player can collect multiple coins.

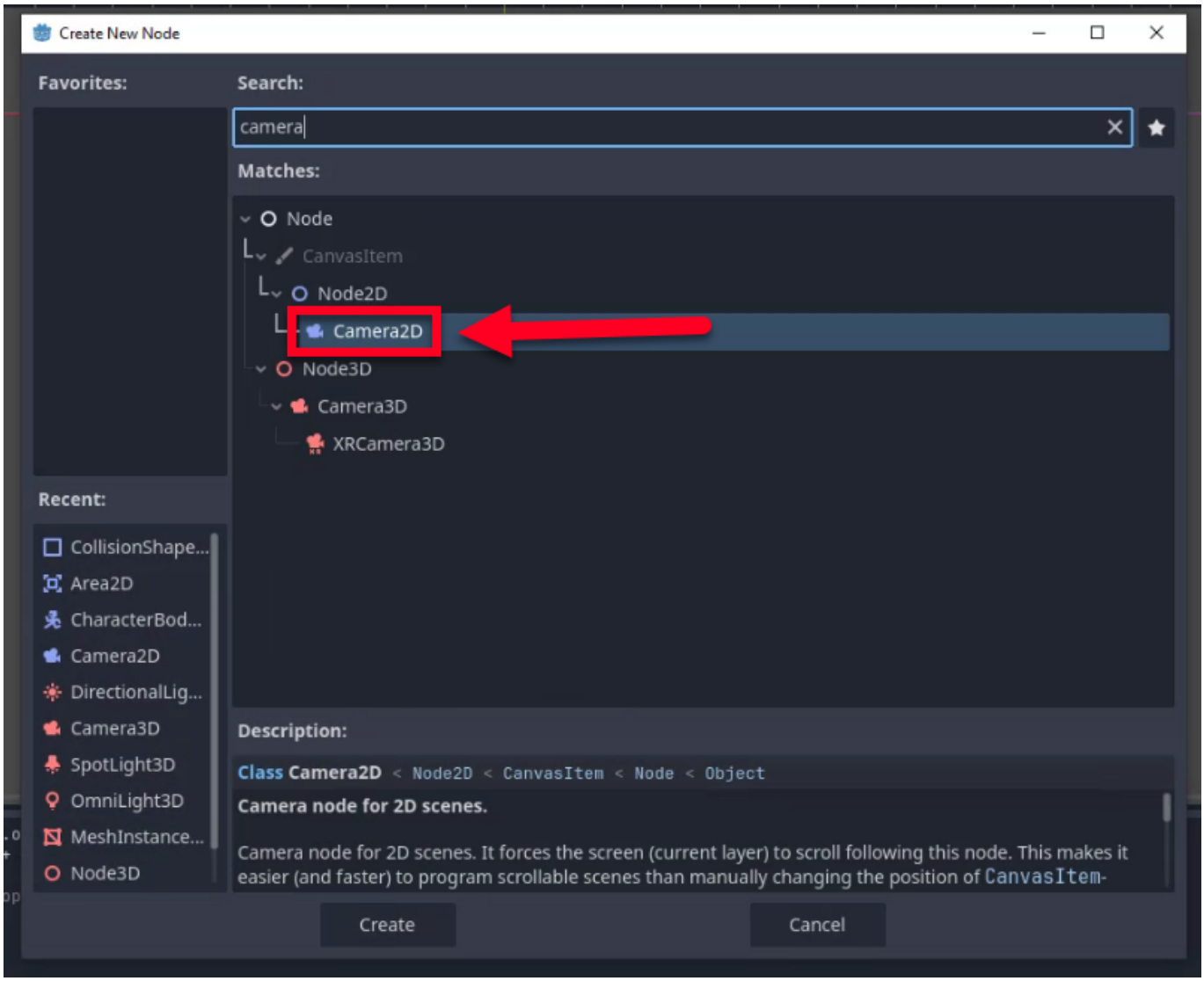


Add a Camera That Follows the Player

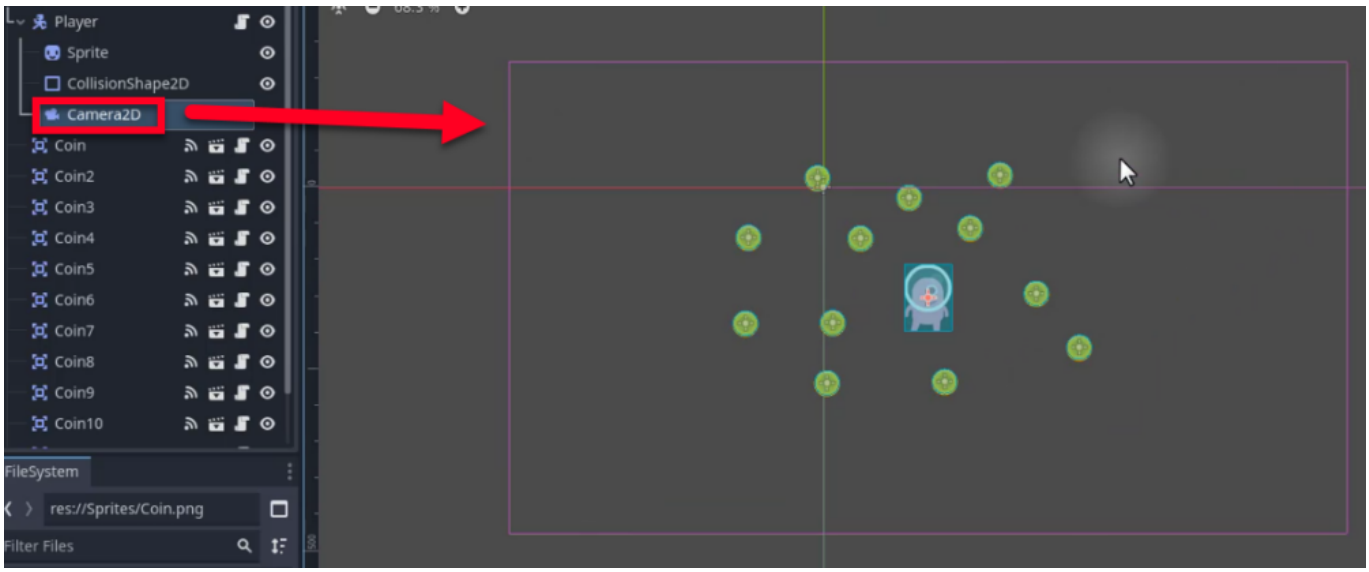
Finally, let's add a camera that follows the player wherever they move. A camera ensures the player always stays in view. Right-click on the Player node in the *Scene* tab and select **"Add Child Node"**.



Search for Camera2D and add it.

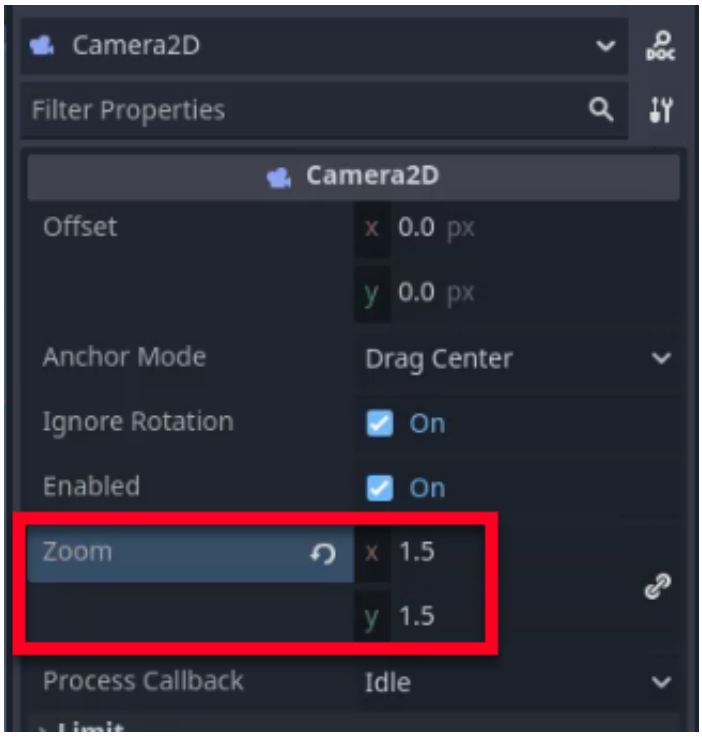


Once added, the camera will automatically follow the player since it’s a child of the Player node.

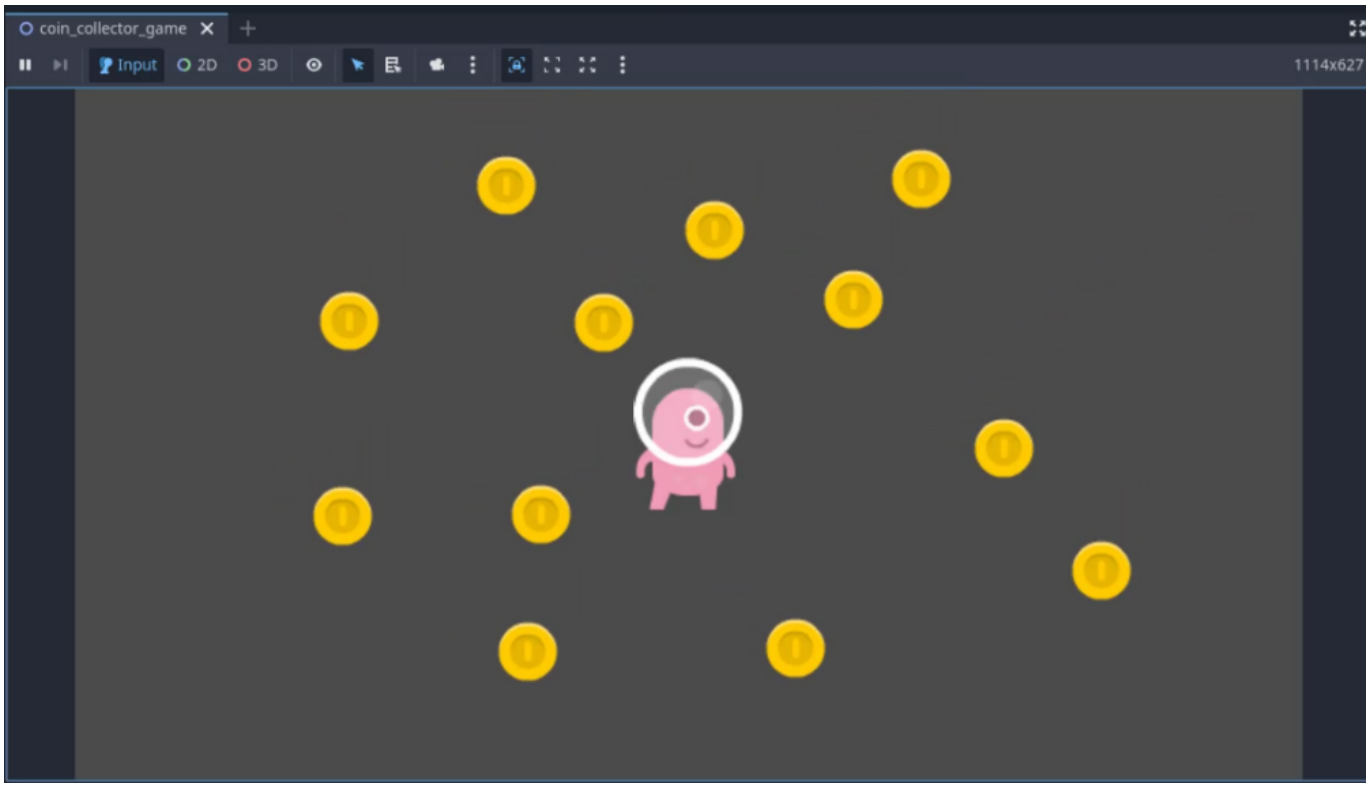


You can adjust the camera’s zoom to change how much of the scene is visible. In the *Inspector*, find

the zoom property and modify it (e.g., set it to (1.5, 1.5)).



This zooms in slightly, giving players a closer view of the game. Save your scene and press play. The camera keeps the player centered while they move, and any coins they collect will still work as expected.



With these features, your game should feel far more interactive!

In this quick article, we will recap our game project to reinforce the key concepts we covered.

Setting Up the Player Script

We began by creating a player script that enables the player to move around. This script was attached to a **CharacterBody2D** node. The CharacterBody2D node is essential for characters in Godot that need to interact with physics objects, such as detecting collisions.

Creating the Collectible Coin

Next, we created a collectible coin using an **Area2D** node. An Area2D node acts as a collider that can detect collisions but allows objects to pass through it, unlike a solid wall. Think of it like a portal. When the player collides with the coin, the player's size increases, and the coin is destroyed.

Key Concepts and Steps

- **Player Movement:** The player script handles the movement of the player character.
- **Collision Detection:** The CharacterBody2D node is used for the player to detect collisions with other objects.
- **Collectible Coin:** The Area2D node is used for the coin to detect when the player collides with it.
- **Player Growth:** Upon collecting a coin, the player's size increases, and the coin is removed from the game.

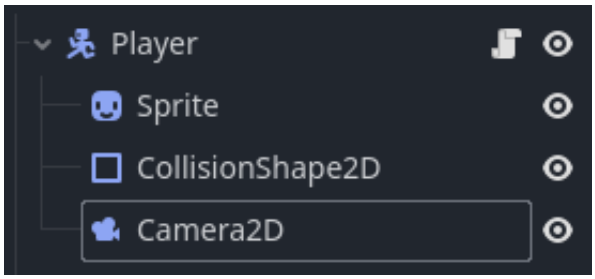
Future Possibilities

With the foundation set, you can now explore creating more complex games. Remember, game development takes time, effort, and practice. As you continue to practice and learn, you will be able to create more intricate and engaging games using Godot.

Keep practicing, and soon you will be able to create any game you can imagine.

Our player is based upon the **CharacterBody2D** node. This is a foundation for characters, providing collision detection, physics, movement, etc.

This node requires a **CollisionShape2D**, which we added and fit the bounds of our **Sprite** node, which provided us with a visual.



We then created a new script called **Player** and attached it to the CharacterBody2D node.

The first thing you will notice is that we are extending from CharacterBody2D. This essentially means our script is building off from the CharacterBody2D node, allowing us to access its functionality.

```
extends CharacterBody2D
```

We are then defining a **move_speed** variable of type float. With a value of 200, this means our player will move at 200 pixels per second.

```
var speed : float = 200.0
```

Next, we have the **_physics_process** function, which acts a lot like the normal **_process** function (called every frame), except this one is called a fixed number of times per second, as physics relies on consistency.

```
func _physics_process(_delta):
    velocity.x = 0
    velocity.y = 0

    if Input.is_key_pressed(KEY_RIGHT):
        velocity.x += speed

    if Input.is_key_pressed(KEY_LEFT):
        velocity.x -= speed

    if Input.is_key_pressed(KEY_UP):
        velocity.y -= speed

    if Input.is_key_pressed(KEY_DOWN):
        velocity.y += speed

    move_and_slide()
```

The first thing we are doing in this function, is setting our **velocity** variable to 0. This variable is managed by the CharacterBody2D. We do this to cancel any movement we might have had from the last frame, as next we will calculate it again based on inputs.

```
velocity.x = 0
velocity.y = 0
```

The next 4 sections, each detect an arrow key, and if pressed, will modify the apropriate axis on our velocity variable.

```
if Input.is_key_pressed(KEY_LEFT):
    velocity.x -= speed
```

In the above example, we are checking the condition: *Input.is_key_pressed(KEY_LEFT)*, which will return true if the left arrow key is held down, and false otherwise. If that condition is true, then we are subtracting our velocity's X axis by our speed, which is the left direction.

We want to do this with all 4 axis (up, down, left and right).

And last but not least, we will call the **move_and_slide** function, which is managed by the CharacterBody2D. This is what we call when we want to apply the velocity to the player, calculate collisions, physics, etc.

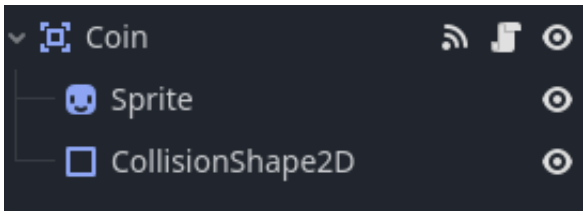
```
move_and_slide()
```

Additional Resources

If you wish to learn more, you can refer to the Godot documentation.

- CharacterBody2D
- Area2D
- Physics Process Function
- Signals
- Input.is_key_pressed

The coins are what the player will move around to collect. Our coin is its own scene with a root node of **Area2D**. An Area2D is a physics based node that can detect collisions. As a child of that, we have a Sprite (to give it a visual), and a CollisionShape2D (to give it a physical being).

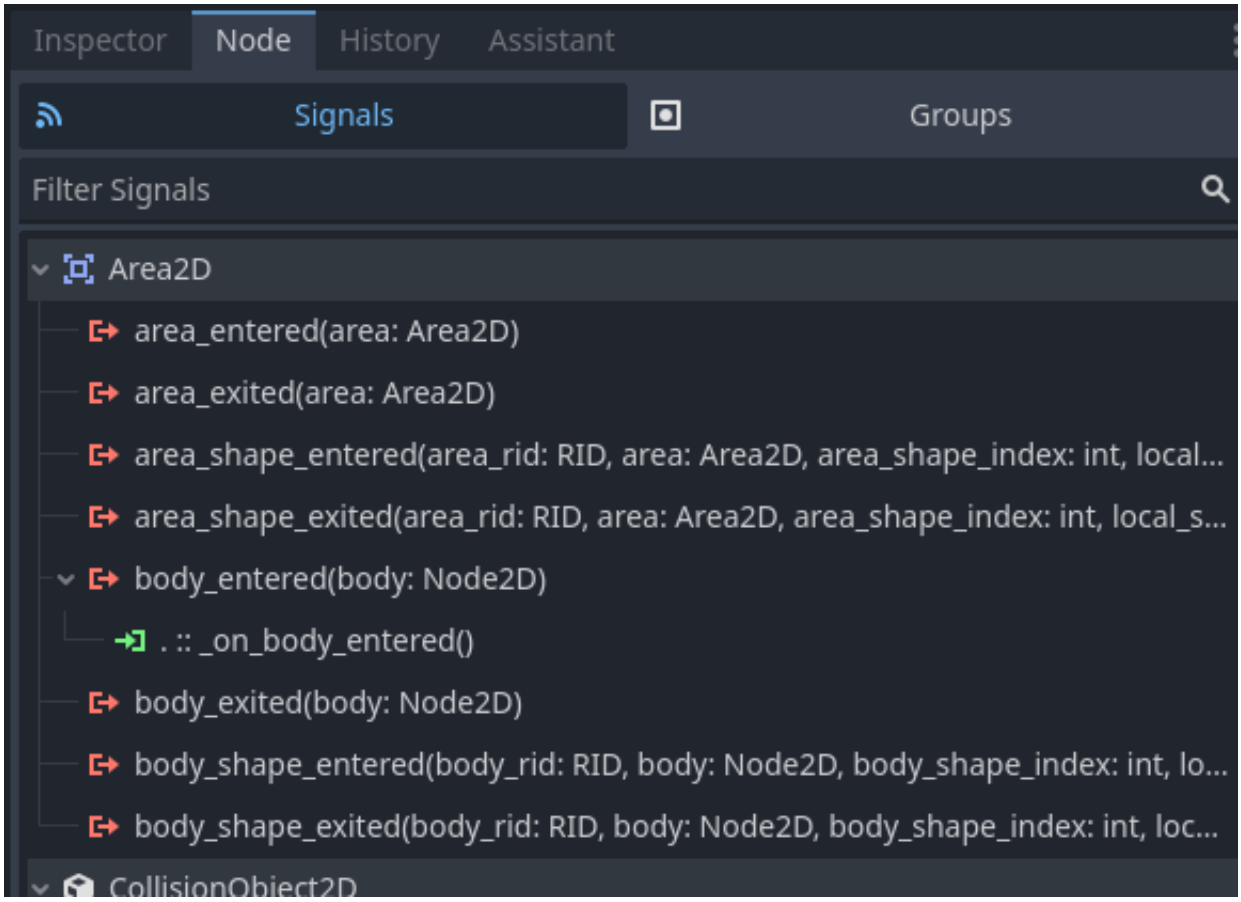


We then attach a new script called **Coin** to the Area2D node.

Inside here, we are extending from Area2D, as that is the node type which the script is attached to.

```
extends Area2D
```

Now before we continue with the code, we need to connect our coin script to the coin's **body_entered signal**. What is a signal? Well basically, a signal is an event that can be invoked when something happens. You can then connect functions to these signals, so when they are emitted, those functions will be called. We are wanting to connect the **body_entered** signal to a function in our script. This signal gets emitted when the coin has entered another body's collider.



Once connected, you should have a new function created in your Coin script. The **body** parameter is the node which has hit us.


```
func _on_body_entered(body):
```

Since the player is the only thing that can hit the coin, we can assume it’s them. So what we’re going to do, is increase the size of that body (the player) by a number.

```
body.scale.x += 0.2
body.scale.y += 0.2
```

Then finally, we want to destroy the coin so it cannot be collected again.

```
queue_free()
```

Additional Resources

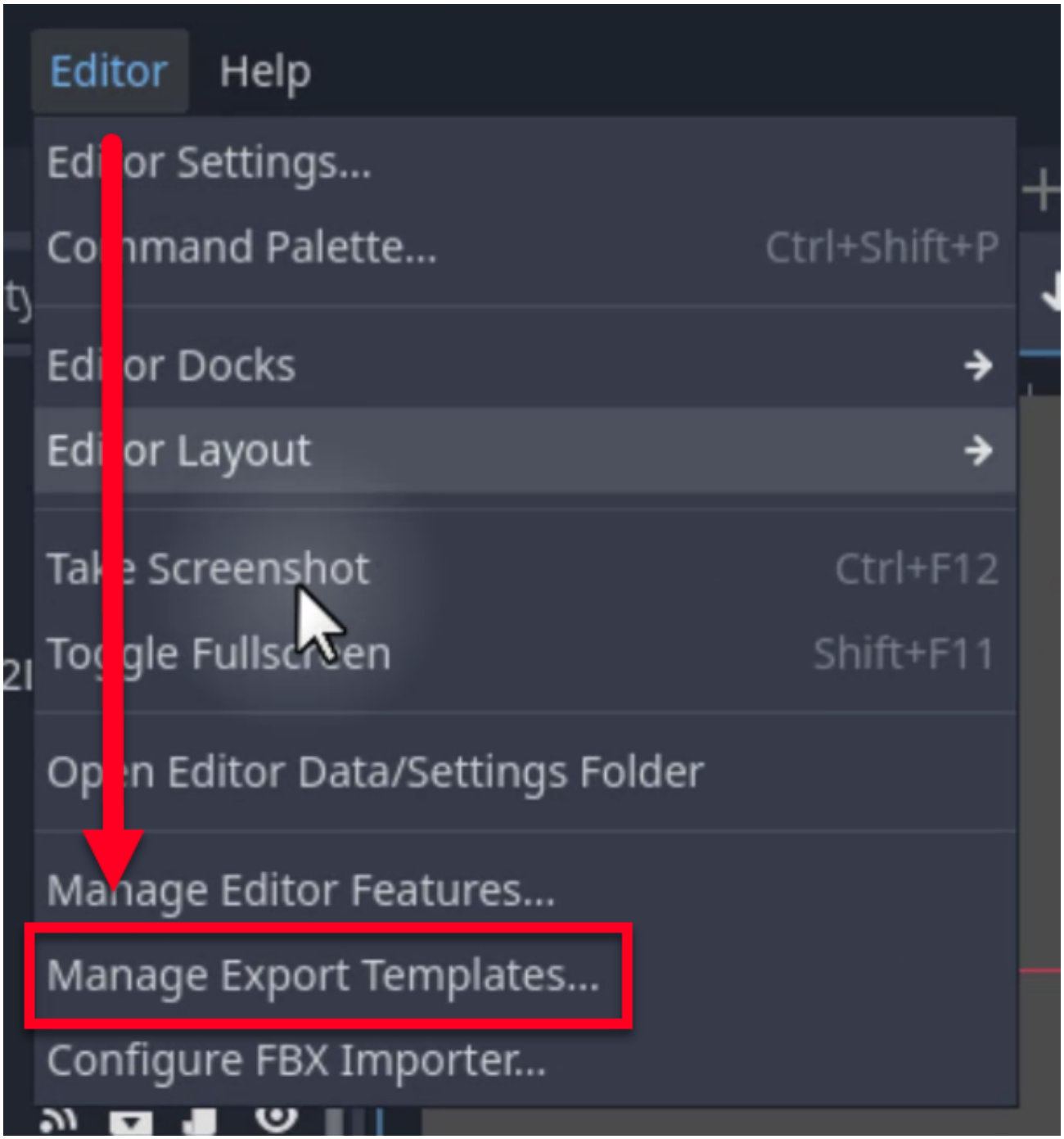
If you wish to learn more, you can refer to the Godot documentation.

- CharacterBody2D
- Area2D
- Physics Process Function
- Signals
- Input.is_key_pressed

In this lesson, we will explore how to export your game in Godot. Exporting your game is a crucial step in game development, as it allows you to package all your assets, scripts, and scenes into an executable file. This file can then be shared with others for testing or uploaded to the internet. Let's dive into the process step by step.

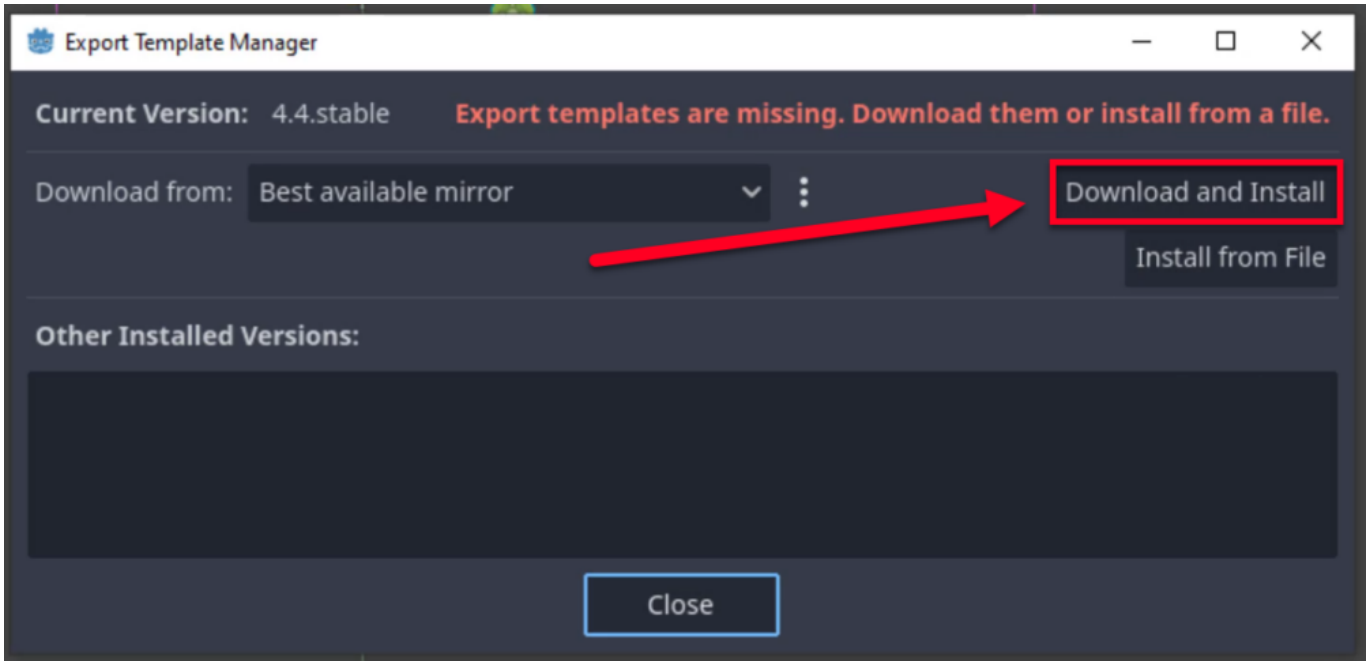
Downloading Export Templates

Before you can export your game, you need to download the export templates. These templates are necessary for building your game to different platforms such as Windows, Mac, or Linux. To start, navigate to the top menu and select **Editor**. From the dropdown menu, choose **Manage Export Templates**.

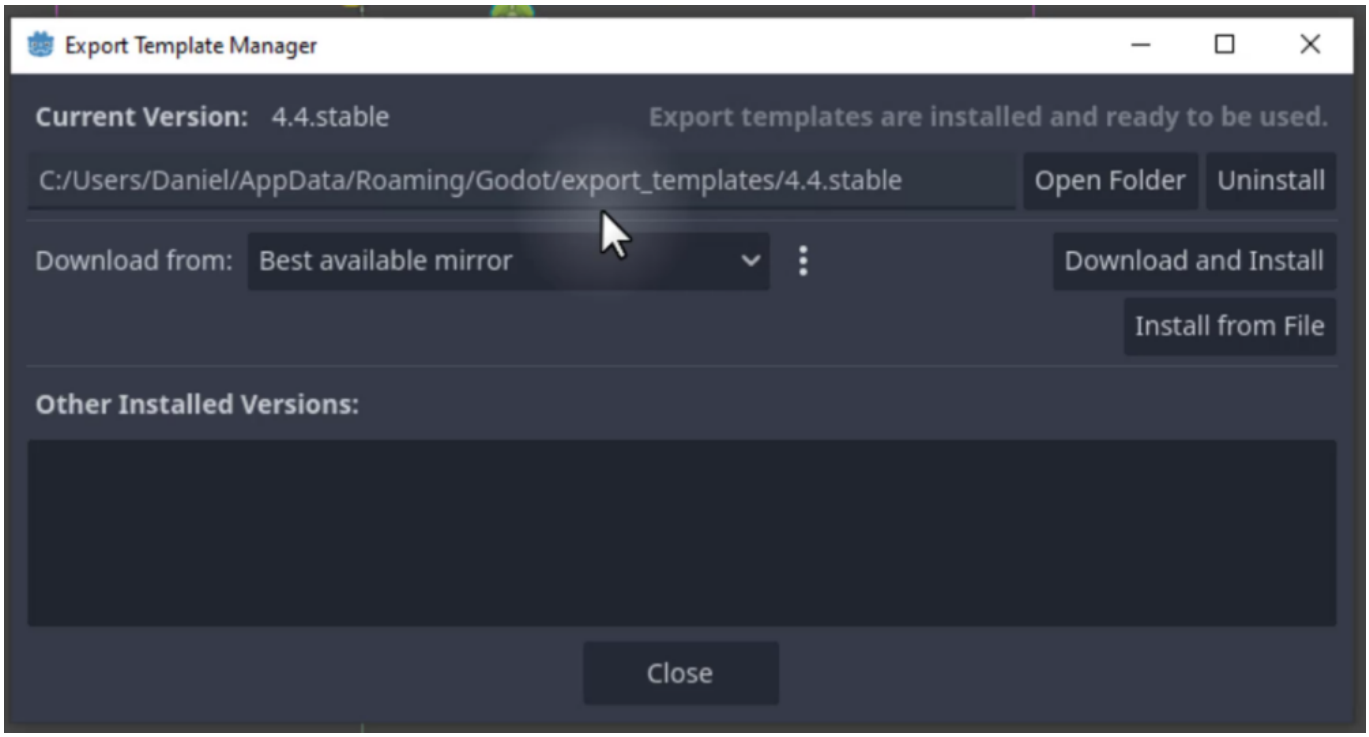


Click on the **Download and Install** button. This will initiate the download process for the export

templates, which can be quite large in size due to the numerous prerequisites and boilerplate files required for different platforms.

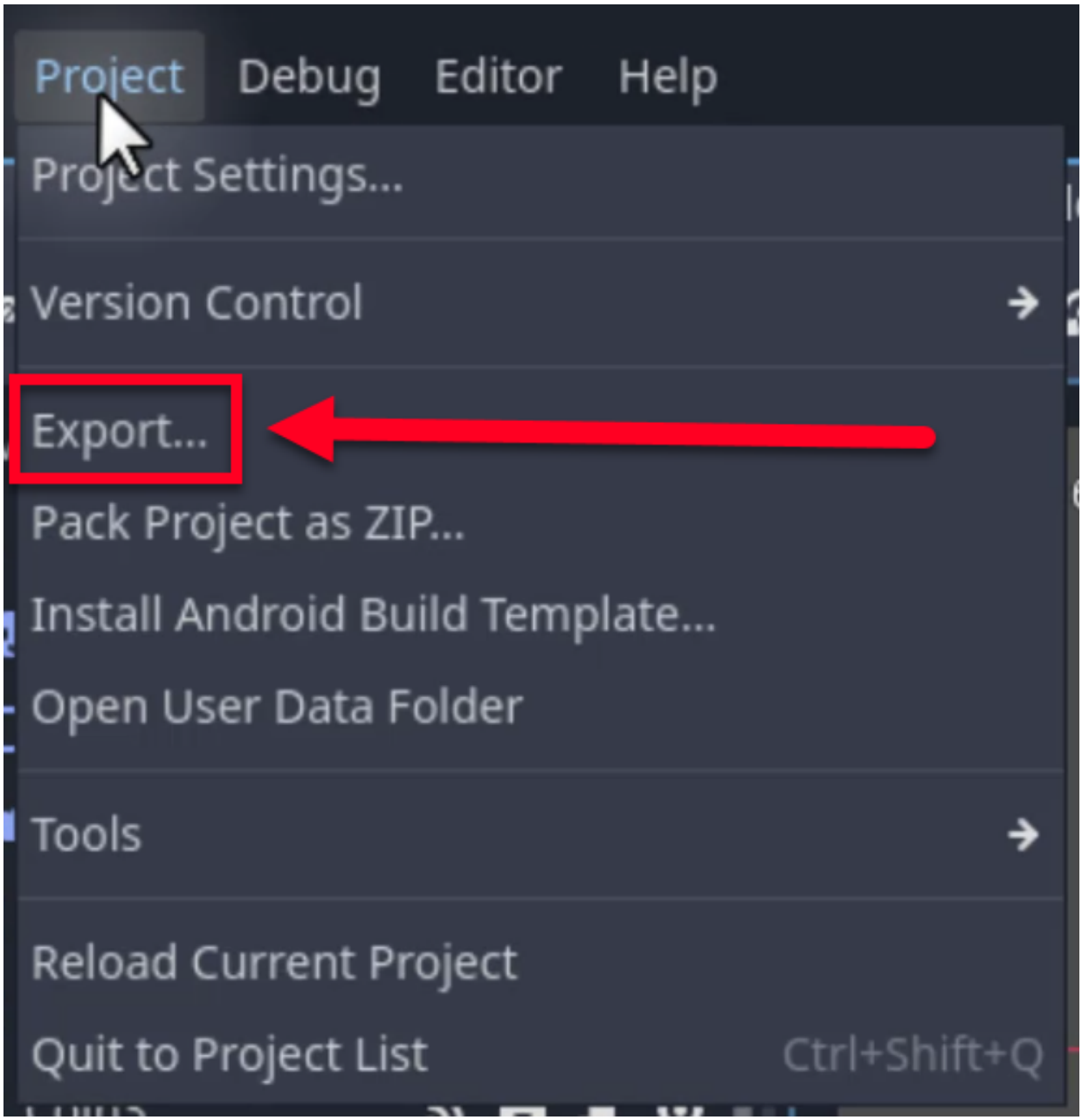


Wait for the installation to complete. You can close the window once the process is finished.

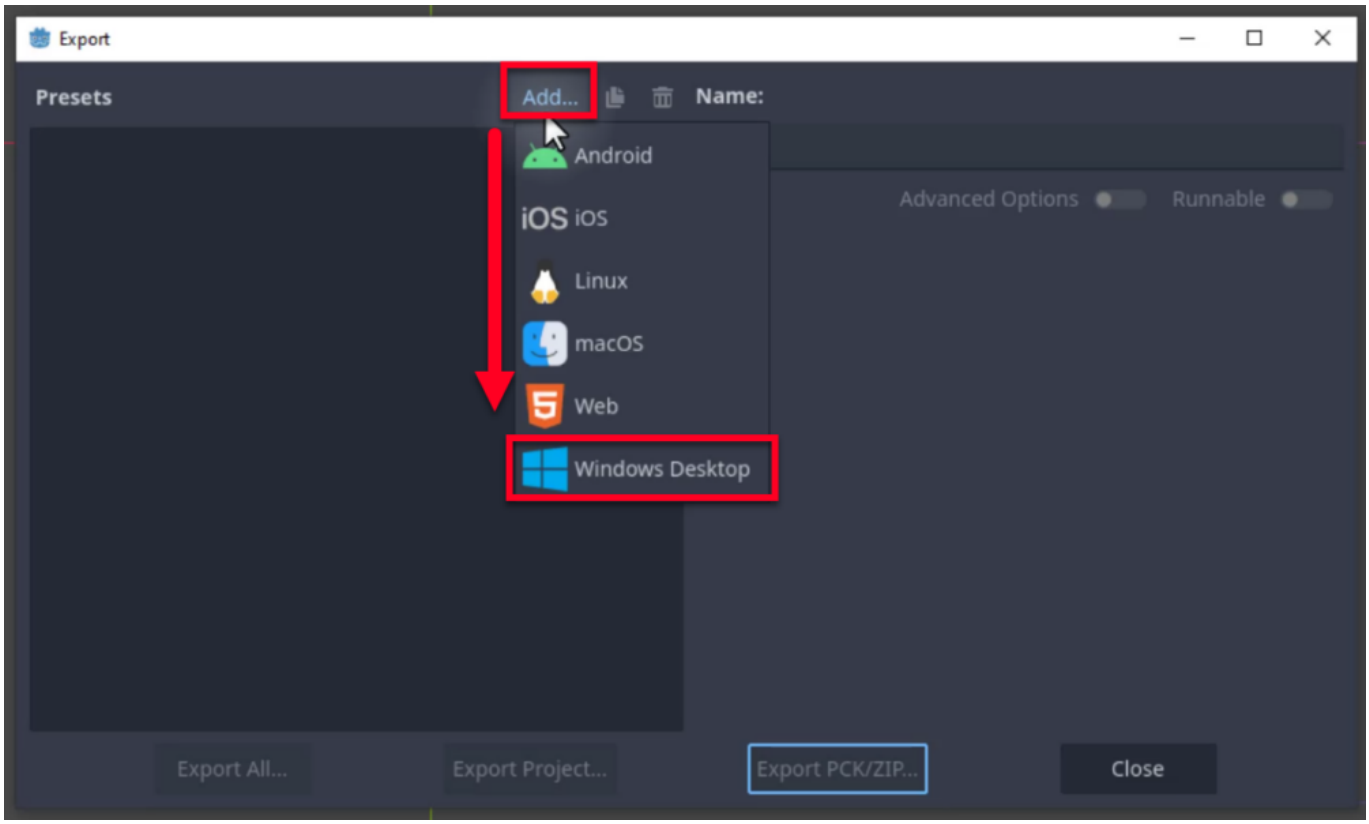


Creating an Export Preset

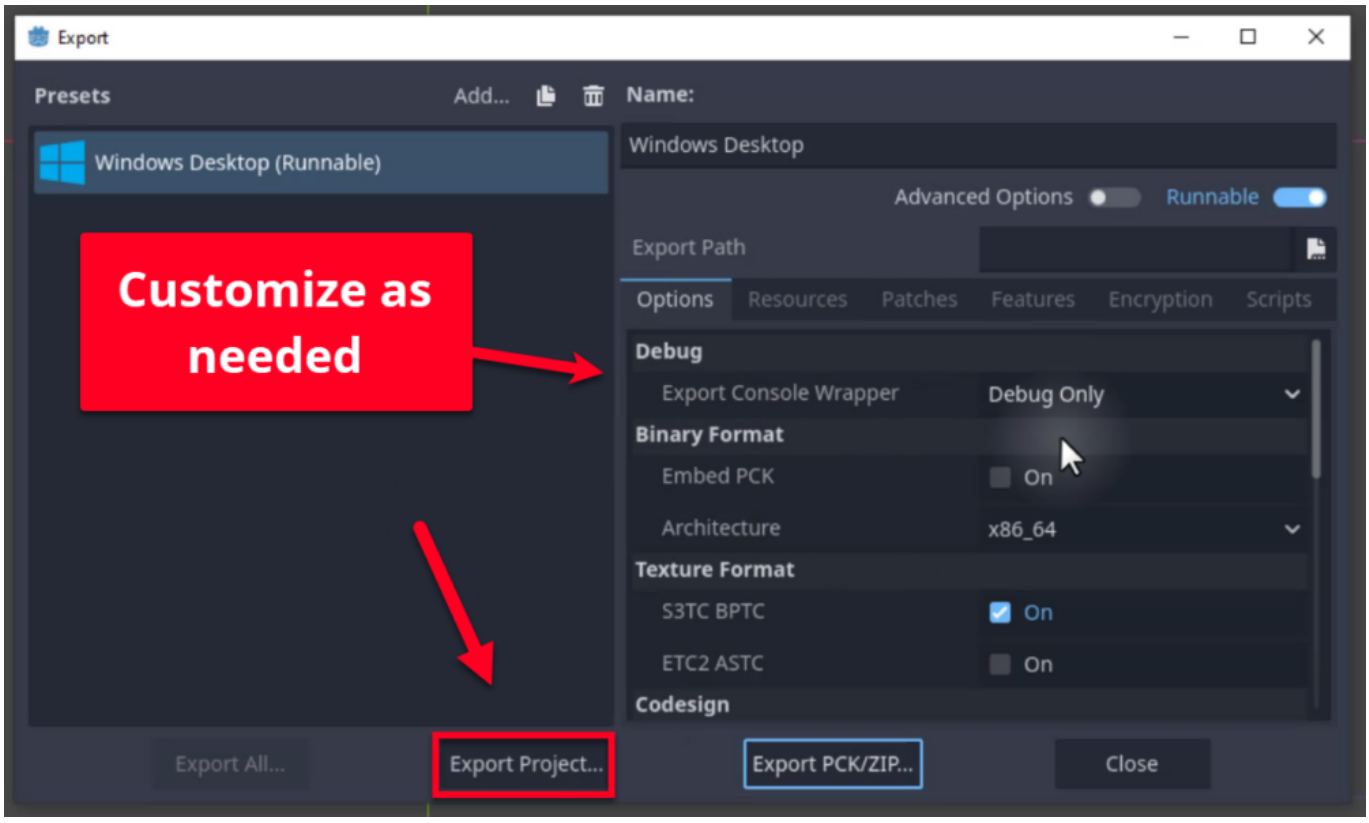
Once the export templates are installed, you can create an export preset for your desired platform. Go to the top menu and select **Project**. From the dropdown menu, choose **Export**.



In the Export window, click on the **Add** button to create a new export preset. Select your target platform. For this example, we will choose **Windows Desktop**.

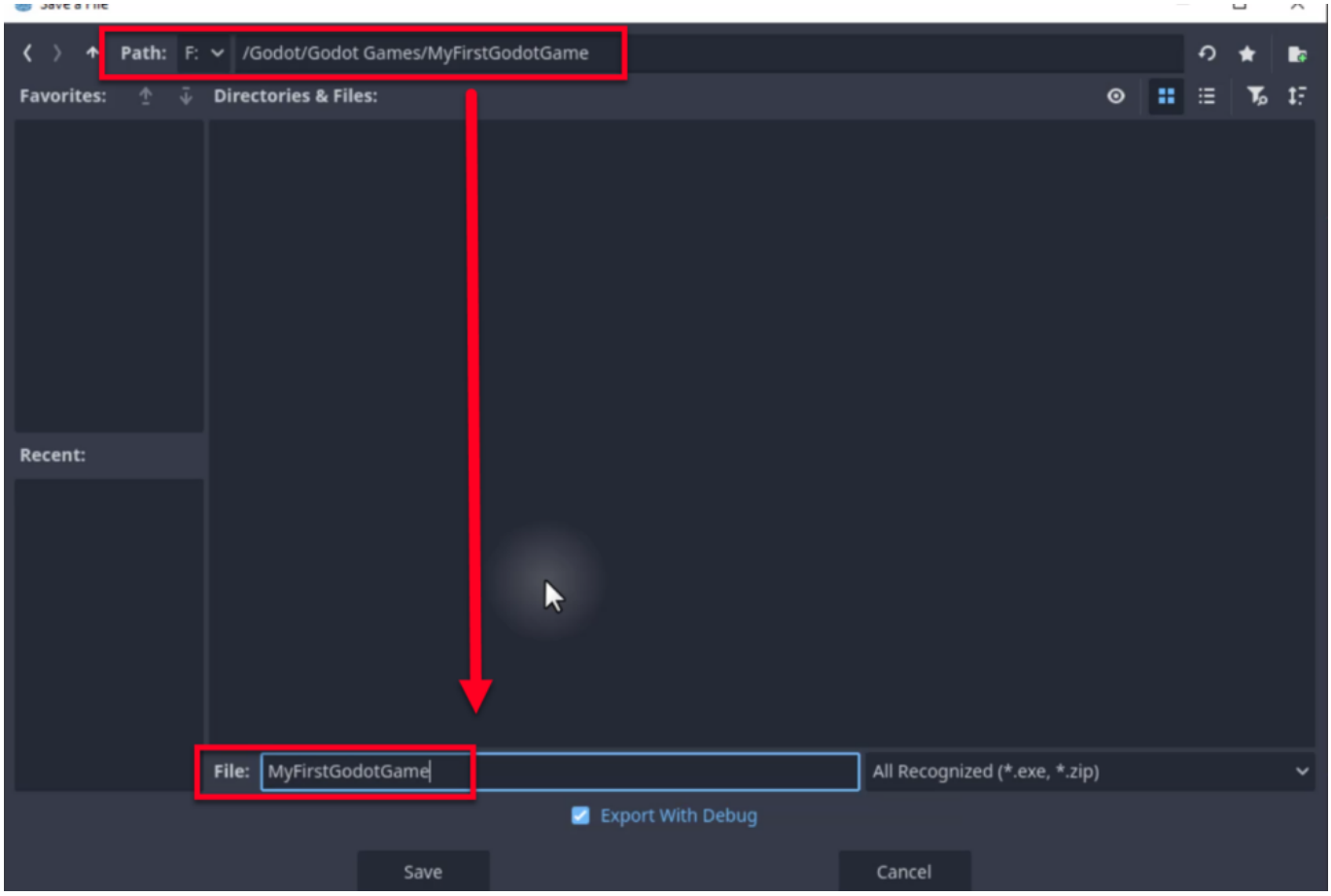


You can customize various options on the right-hand side, but for a basic export, you can leave these settings as they are. Scroll down to the **Export Project** section and click on it.



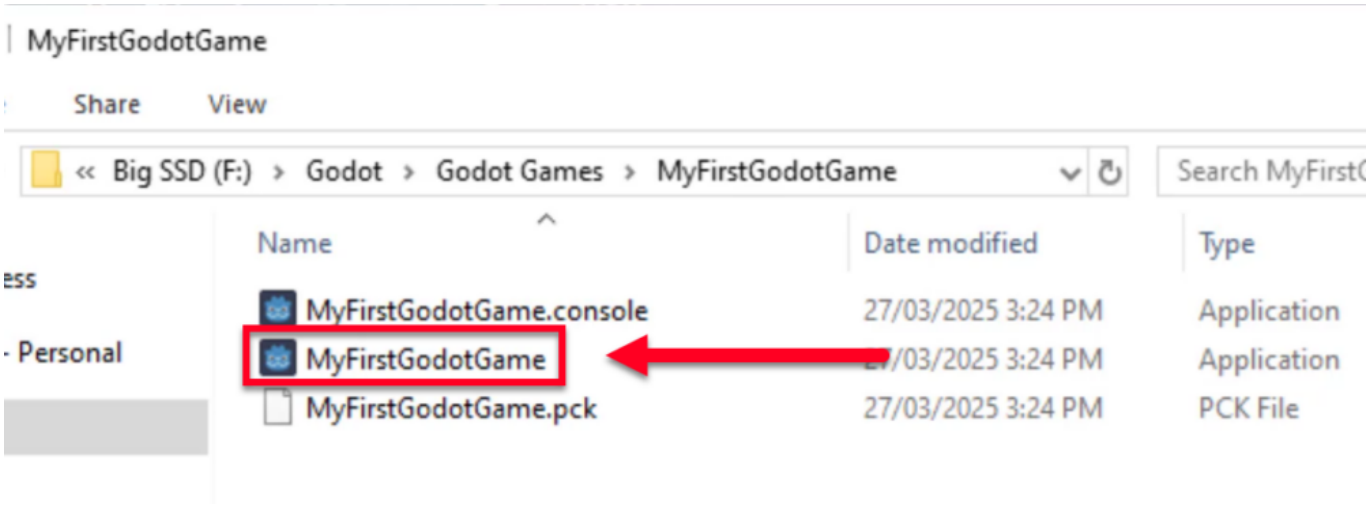
Choose a location for your executable file. Create a new folder for your game, name it appropriately

(e.g., **My First Godot Game**), and name the executable file. Then click Save to trigger the actual export.

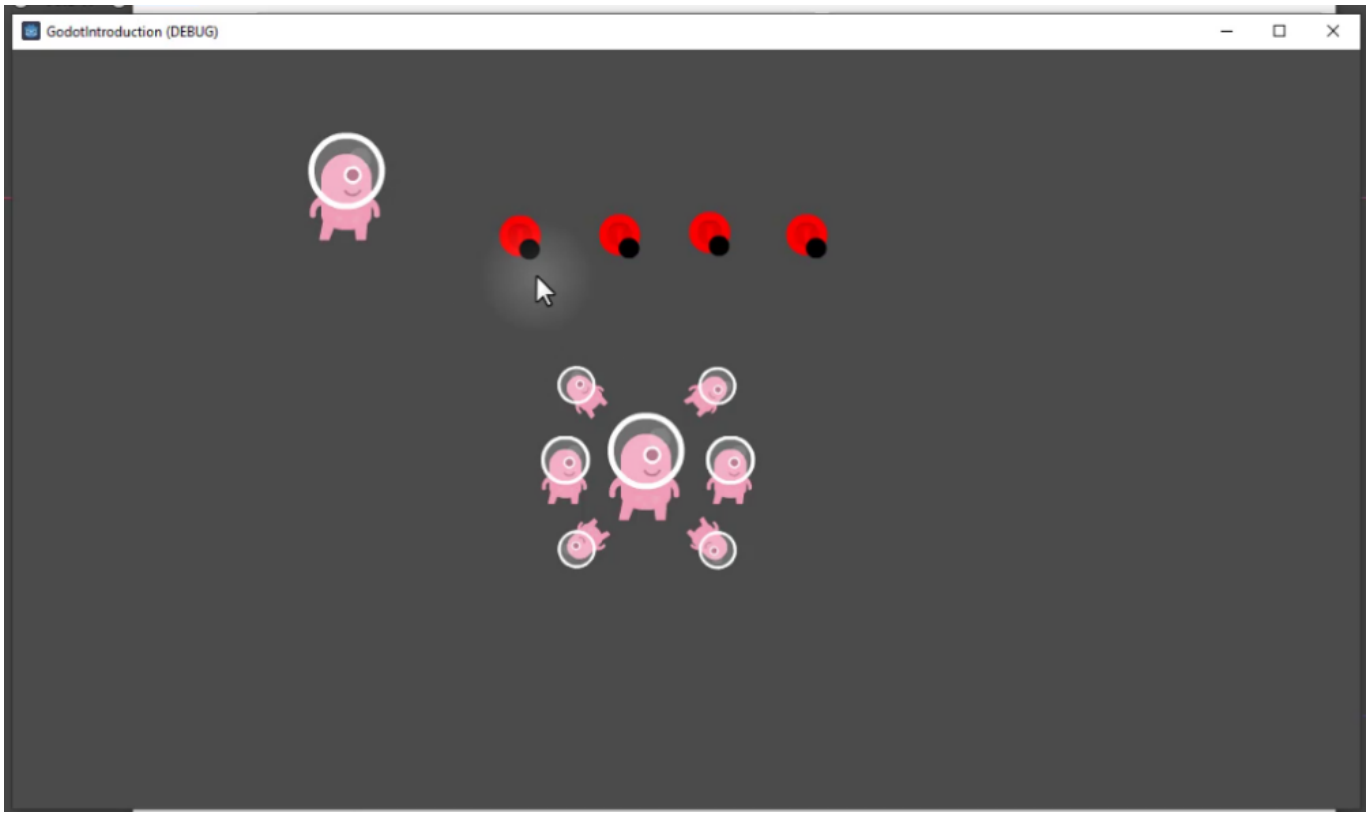


Running Your Executable

After creating the export preset and specifying the location, Godot will generate the executable file. Navigate to the folder where you saved the executable file and double-click on the executable to launch your game.



Your game should now run in its own window.



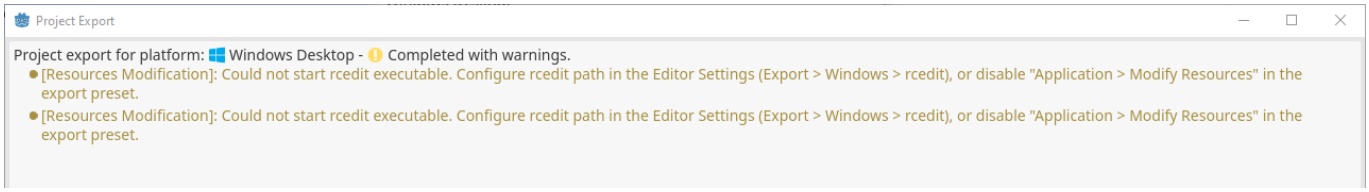
Sharing Your Game

To share your game with others, follow these simple steps:

- Zip the folder containing your executable file and any other necessary files.
- Send the zipped folder to whoever you want to share your game with.

Congratulations! You have successfully exported and shared your first Godot game. This process is essential for testing and distributing your games, and mastering it will greatly benefit your game development journey at Zenva.

You may have noticed that when exporting your game, there is a warning and/or error that occurs relating to the the *rcedit executable*.



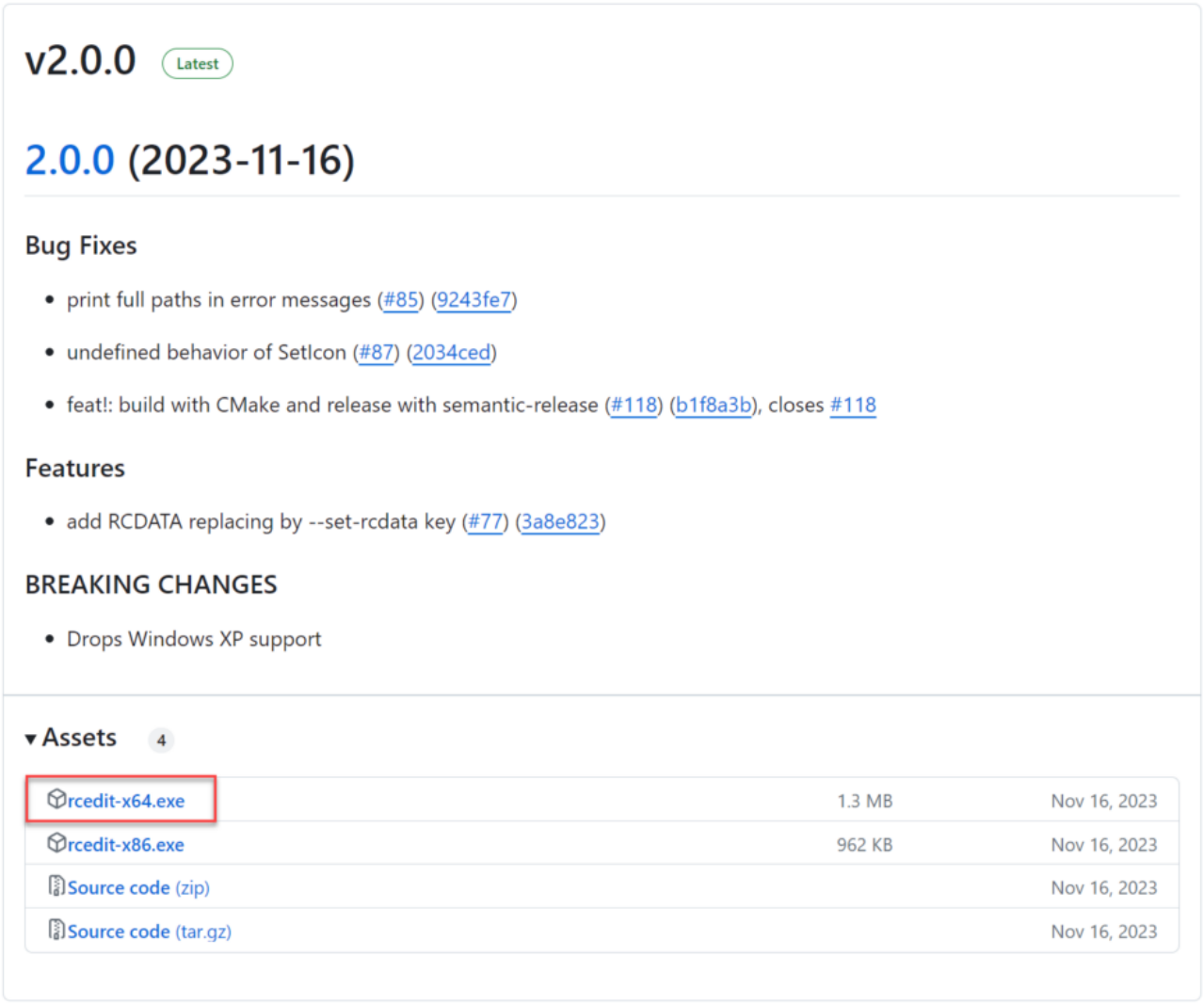
This is totally optional and does not affect your game export if you choose to ignore it. But if you would like to include a custom icon for your game application, then follow these steps.

Creating the Icon

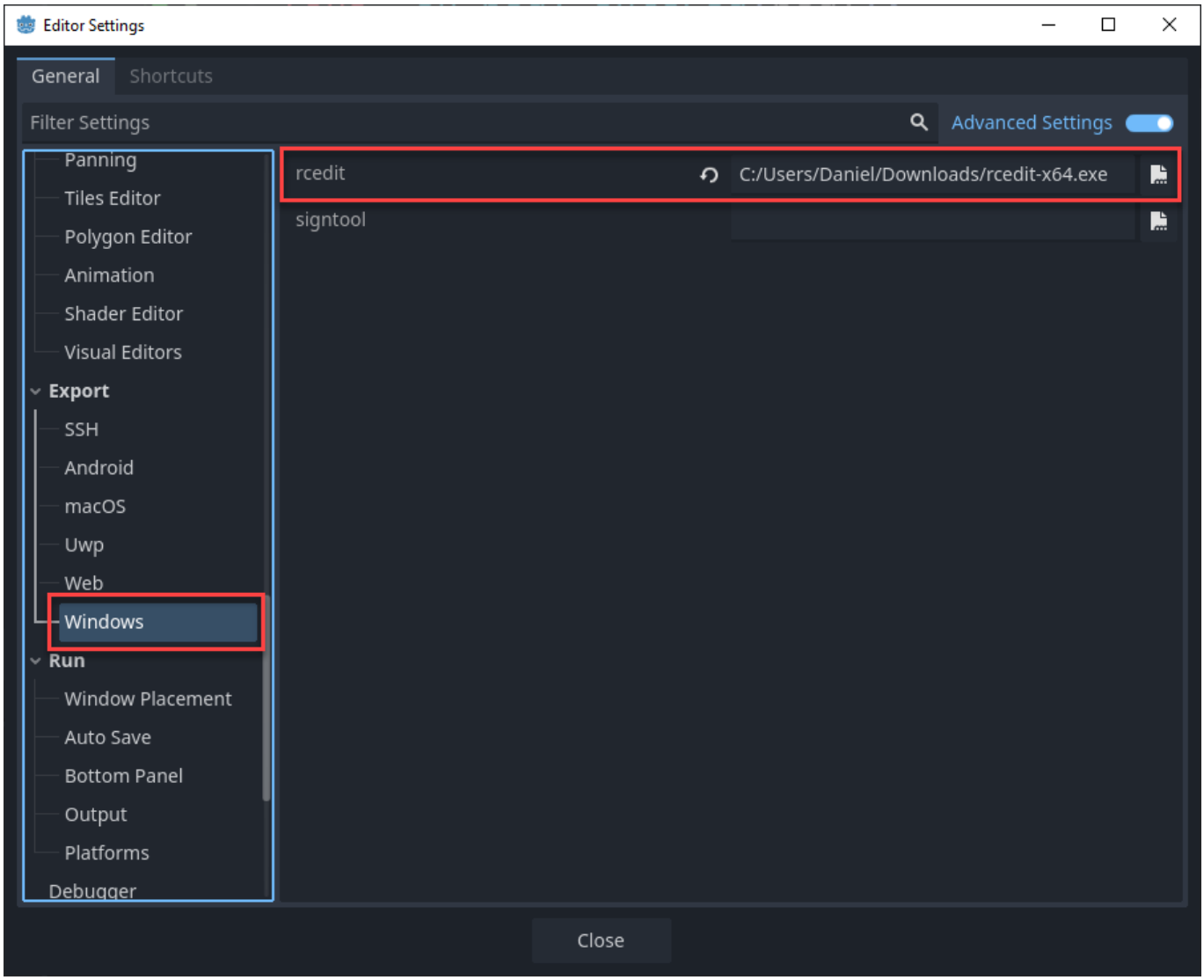
Application icons don't use image formats such as PNG or JPG. They instead use ICO. However, we can use *rcedit* to handle this for us.

Changing the Executable Icon

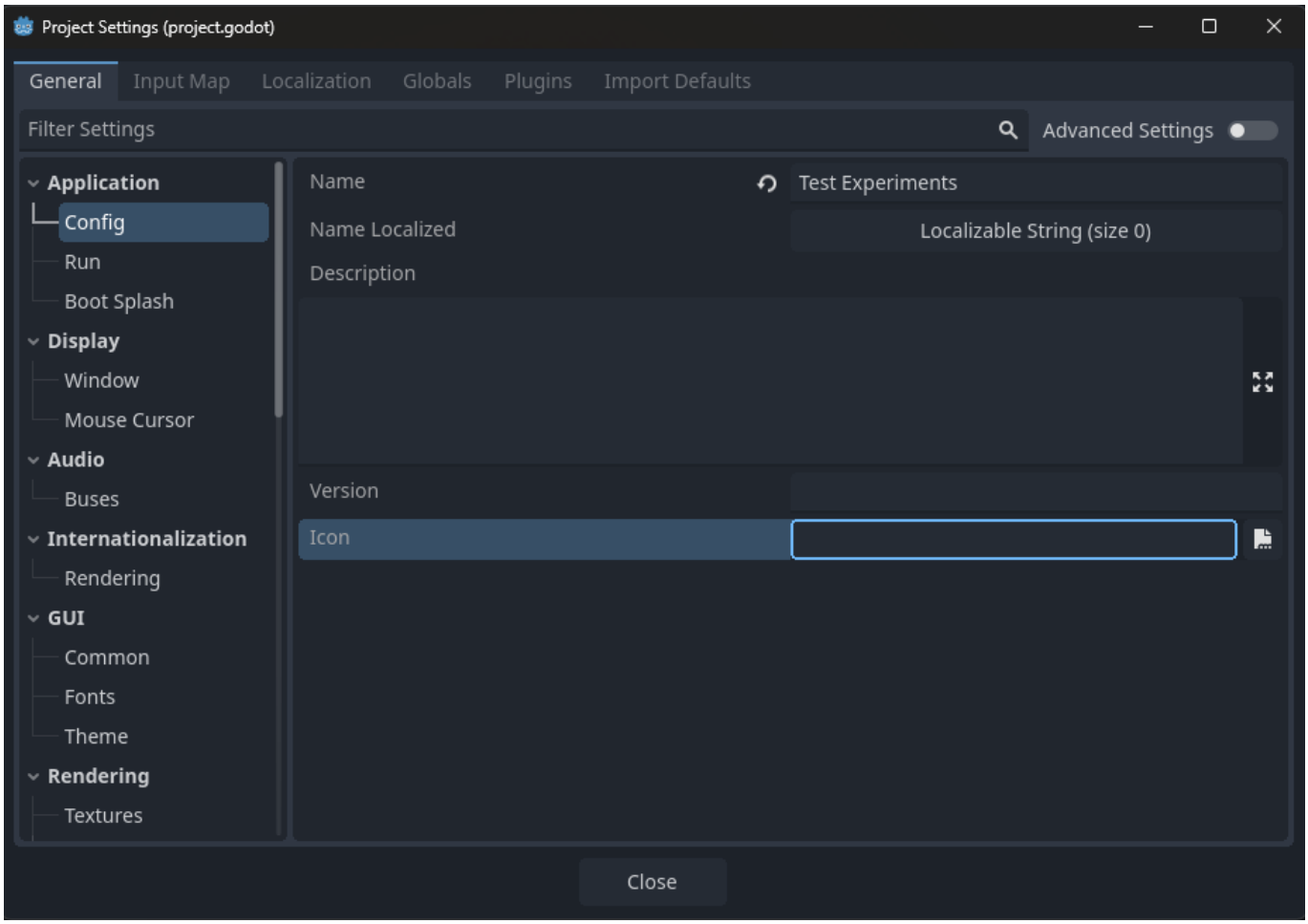
To change the icon that appears on the executable file, we first need to download the *rcedit* tool here. Download whatever the most recent version if for you.



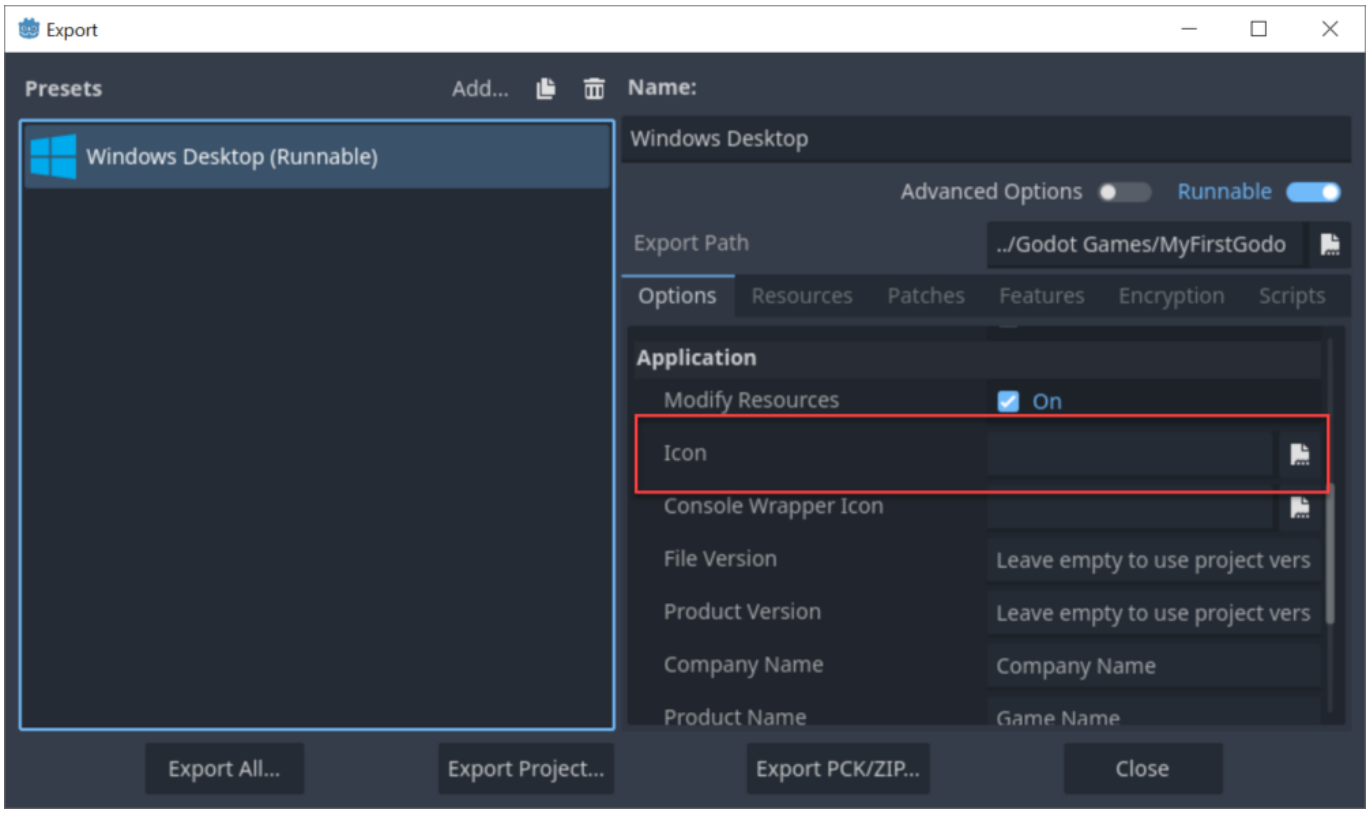
Then back in Godot, go to **Editor > Editor Settings > Export > Windows** and set the *rcedit* path to be that executable we just downloaded.



In the project settings, set the Icon to be your PNG file.



Now we can go to the export window (**Project > Export**) and in the options panel, set the *Icon* to be your PNG file.



Now you can export your project and see the results!

Congratulations on completing your introductory course on the Godot game engine! Throughout this journey, you've covered a wide range of topics that have equipped you with the fundamental skills needed to create your own games. Let's recap what you've learned and explore how you can continue your journey with Zenva.

Getting Started with Godot

At the beginning of the course, you learned how to:

- Install the Godot game engine.
- Create a new project.
- Navigate the Godot editor, which initially might have seemed overwhelming with its various buttons, panels, and input fields.

Fundamentals of Godot and 2D Development

As you progressed, you delved into the core concepts of Godot and 2D game development, including:

- **Nodes and Scenes:** The building blocks of Godot games.
- **Tools:** Various tools within the Godot editor to aid in game development.
- **Parenting:** Organizing nodes in a hierarchical structure.
- **Cameras:** Managing the viewport and rendering.

Transitioning to 3D Development

The course also introduced you to 3D game development, covering essential aspects such as:

- **3D Environments:** Creating immersive 3D worlds.
- **Models:** Working with 3D objects.
- **Materials:** Applying textures and shaders to models.
- **Lighting:** Illuminating your 3D scenes for enhanced realism.

Scripting and Game Logic

The second half of the course focused on scripting using GDScript, Godot's integrated scripting language. You learned about:

- **Variables:** Storing and managing data.
- **Conditions:** Making decisions in your code.
- **Operators:** Performing operations on variables.
- **Functions:** Organizing and reusing code.
- **Vectors:** Working with 2D and 3D coordinates.

Putting It All Together: Creating a Coin Collector Game

To reinforce your learning, you created a coin collector mini-game that allowed you to apply all the concepts you've learned. This hands-on project also introduced you to additional game development techniques.

Continuing Your Learning Journey with Zenva

Zenva is an online learning academy with over a million students, offering a diverse range of courses suitable for beginners and those looking to expand their skills. The versatility of Zenva's courses allows you to learn in various ways:

- Follow along with video tutorials.
- Read included lesson summaries.
- Use the provided course files to follow along with the instructor.

Thank you for completing this course, and we wish you the best of luck with your future Godot game development endeavors!

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

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

coin.gd

Found in folder: **/Scripts**

This code increases the size of an object (in this case, the player) when it enters the coin's collision area. The object's width and height are increased by 20%. After the object's size is increased, the coin itself is removed.

```
extends Area2D

func _on_body_entered(body):
 body.scale.x += 0.2
 body.scale.y += 0.2

 queue_free()
```

conditions.gd

Found in folder: **/Scripts**

This code initializes variables for score, game over status, and password. It then checks various conditions, printing messages based on the score value, comparisons between variables, and the game over status. Finally, it checks the password and prints a message if it matches the set value.

```
extends Node

var score : int = 6
var game_over : bool = false
var password : String = "123"

func _ready ():
 if score == 10:
  print("score is 10")

 if score > 5:
  print("score is greater than 5")

 var a : int = 50
 var b : int = 100

 if a < b:
  print("a is less than b")

 if a != b:
  print("a is not b")
```



```
 if score > 80:
  print("A")
 elif score > 60:
  print("B")
 elif score > 30:
  print("C")
 else:
  print("D")

 # Challenge
 if game_over == true:
  print("Go to menu")
 else:
  print("Keep playing")

 if password == "123":
  print("Enter")
```

functions.gd

Found in folder: **/Scripts**

This code initializes a node in the game scene and prints a welcome message when it's ready. It then performs a simple arithmetic operation, adding 42 and 23, and prints the result. Additionally, it checks if a score of 59 is enough to win the game and prints the outcome.

```
extends Node2D

func _ready():
 _welcome_message()

 var result = _add(42, 23)
 print(result)

 # Challenge
 var game_over = _has_won(59)
 print(game_over)

func _process(delta):
 pass

func _welcome_message ():
 print("Welcome to the game")

func _add (a : float, b : float) -> float:
 var sum : float = a + b
 return sum

# Challenge
func _has_won (score : int) -> bool:
 if score >= 100:
  return true
 else:
```



```
  return false
```

mover.gd

Found in folder: **/Scripts**

This code defines a 2D sprite that moves downwards at a constant speed. The speed of the movement is determined by the `speed` variable, which is set to 100. The sprite's position is updated every frame, with the movement being scaled by the time elapsed since the last frame (`delta`).

```
extends Sprite2D

var speed : float = 100

func _process(delta):
 var direction = Vector2(0, 1)
 position += direction * delta * speed
```

operators.gd

Found in folder: **/Scripts**

This code initializes and modifies several variables, including score, speed, text, and money, performing arithmetic operations such as addition, subtraction, multiplication, and division on them. It then prints the results of these operations to the console. The code also declares and manipulates additional variables a, b, and c, performing further arithmetic operations and assignments.

```
extends Node

var score : int = 0
var speed : float = 5.5
var text : String = "first"

var money : int = 10

func _ready ():
 score = 10
 score += 1
 score -= 5
 score *= 2
 score /= 4
 print(score)

 speed *= 2.548
 speed /= 2.09
 print(speed)

 text += "second"
 print(text)
```



```
# Challenge
money += 5
money *= 2
money -= 3
money /= 2
print(money)

var a : int = 5
a = 10 + 2
a = 5 * 10 + 2 + 81 / 2

var b : int = a * 2
var c : int = 0

c = a / b
```

player.gd

Found in folder: **/Scripts**

This code controls the movement of a character in a 2D game. It sets the character's velocity to zero every frame, then checks for input from the right, left, up, and down arrow keys to adjust the velocity accordingly. The character is then moved based on its velocity using the `move_and_slide` function.

```
extends CharacterBody2D

var speed : float = 100

func _physics_process(delta):
 velocity.x = 0
 velocity.y = 0

 if Input.is_key_pressed(KEY_RIGHT):
  velocity.x += speed

 if Input.is_key_pressed(KEY_LEFT):
  velocity.x -= speed

 if Input.is_key_pressed(KEY_UP):
  velocity.y -= speed

 if Input.is_key_pressed(KEY_DOWN):
  velocity.y += speed

 move_and_slide()
```

variables.gd

Found in folder: **/Scripts**

This code defines a set of variables to store game-related data, such as score, movement speed, and game state, as well as geographical data about a country. When the node is initialized, it changes some of the game-related variables and prints out the values of all variables to the console. The code demonstrates basic variable declaration, assignment, and printing in GDScript.

```
extends Node

var score : int = 10
var move_speed : float = 2.53
var game_over : bool = false
var ability : String = "slash"

# Challenge
var country_name : String = "Australia"
var population : int = 25000000
var highest_altitude: float = 2.228
var landlocked : bool = false

func _ready ():
 move_speed = 0.1183
 game_over = true
 ability = "attack"

 print(score)
 score = 20
 print(score)

 # Challenge
 print(country_name)
 print(population)
 print(highest_altitude)
 print(landlocked)
```

vectors.gd

Found in folder: **/Scripts**

This code moves a node along the X axis at a speed defined by the `speed` variable. It also moves the node in a diagonal direction (down and to the right) at the same speed. The movement is updated every frame, with the distance moved being scaled by the time elapsed since the last frame (`delta`).

```
extends Node2D

var speed : float = 100.0

func _process (delta):
 # Move along X axis
 position.x += speed * delta

 # Move along defined direction
 var direction : Vector2 = Vector2(1, 1)
 position += direction * delta * speed
```

