SILVER OAK UNIVERSITY EDUCATION TO INNOVATION	Course Code: 3040243338 Course Name: Mobile Game Development
	SEMESTER: 5

Unit - 2

Game Development Fundamentals

2.1 2D Game Physics: Gravity, Collisions, RigidBody2D, KinematicBody2D

Physics introduction: **Understanding 2D Physics Engine**

When you are developing a game, you may need the following features to craft the gameplay:

- When two game objects collide with others, for example, when a marble hits a score trigger.
- Control the movement of an object after applying a force or a collision has happened. For example, how the marble bounces off the pegs.

A physics engine is ideal for this; it provides two major functions: **Collision Detection** and **Object Movement Simulation**.

1. **Collision Detection**
   The engine continuously checks for overlaps between the physics objects in the scene and sends signals when they collide.
2. **Object Movement Simulation**
   The engine simulates the movement of objects based on their physics attributes.

These are the physics attributes in Godot:

- **Mass**: Determines how forces affect an object.
- **Gravity**: Applies a downward force to simulate falling.
- **Friction**: Resists the motion between two surfaces in contact.
- **Bounce**: Determines how much velocity is reserved when two bodies collide.

Now you know the basic concepts of the physics engine, you can move on to learn more about the different physics nodes offered by Godot.

Understanding Different Physics Nodes

- Godot offers Four main physics nodes: **CharacterBody2D,RigidBody2D**, **StaticBody2D** and **Area2D**. Each node has different purposes and behaviours and you should choose the right one for each game element.

Collision objects:

Godot offers four kinds of collision objects which all extend CollisionObject2D. The last three listed below are physics bodies and additionally extend PhysicsBody2D**.**

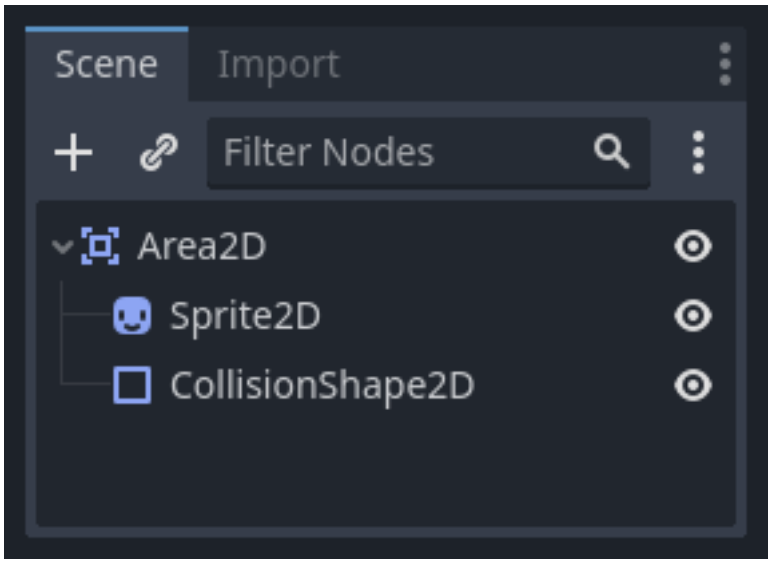
- **Area2D**

Area2D nodes provide detection and influence. They can detect when objects overlap and can emit signals when bodies enter or exit. An Area2D can also be used to override physics properties, such as gravity or damping, in a defined area.

Steps to Add:

1. **Create a new scene and add an Area2D node.**
2. **Add a CollisionShape2D or CollisionPolygon2D as a child.**
3. **Assign a shape (e.g., rectangle or circle).**
4. **Connect signals like body_entered or area_entered to trigger events.**

Use Case: Detect when a player enters a danger zone or picks up an item.



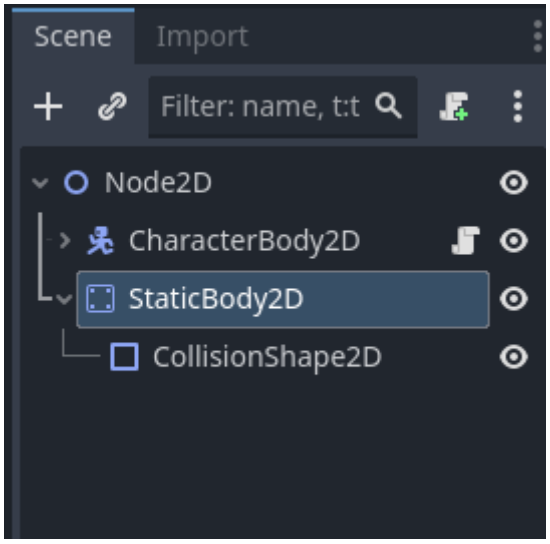
- **StaticBody2D**

A static body is one that is not moved by the physics engine. It participates in collision detection, but does not move in response to the collision. They are most often used for objects that are part of the environment or that do not need to have any dynamic behaviour.

Steps to Add:

1. **Add a StaticBody2D node.**
2. **Add a CollisionShape2D as a child.**
3. **Assign a shape to define its collision area.**
4. **Optionally add a Sprite2D to visualize it.**

Use Case: Create a solid wall or ground that other bodies can collide with.



- **RigidBody2D**

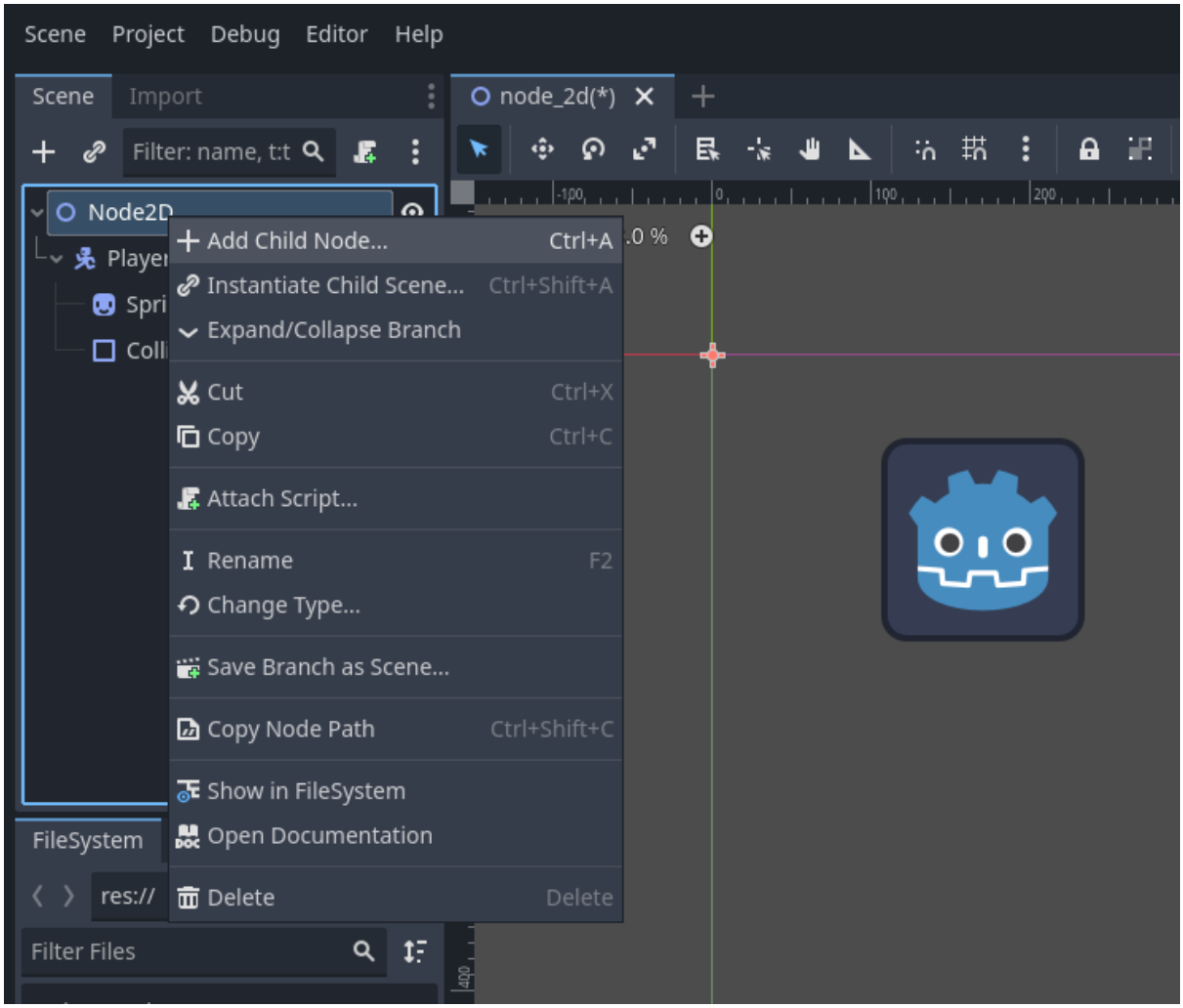
This is the node that implements simulated 2D physics. You do not control a RigidBody2D directly. Instead, you apply forces to it and the physics engine calculates the resulting movement, including collisions with other bodies, and collision responses, such as bouncing, rotating, etc.

You can modify a rigid body's behaviour via properties such as "Mass", "Friction", or "Bounce", which can be set in the Inspector.

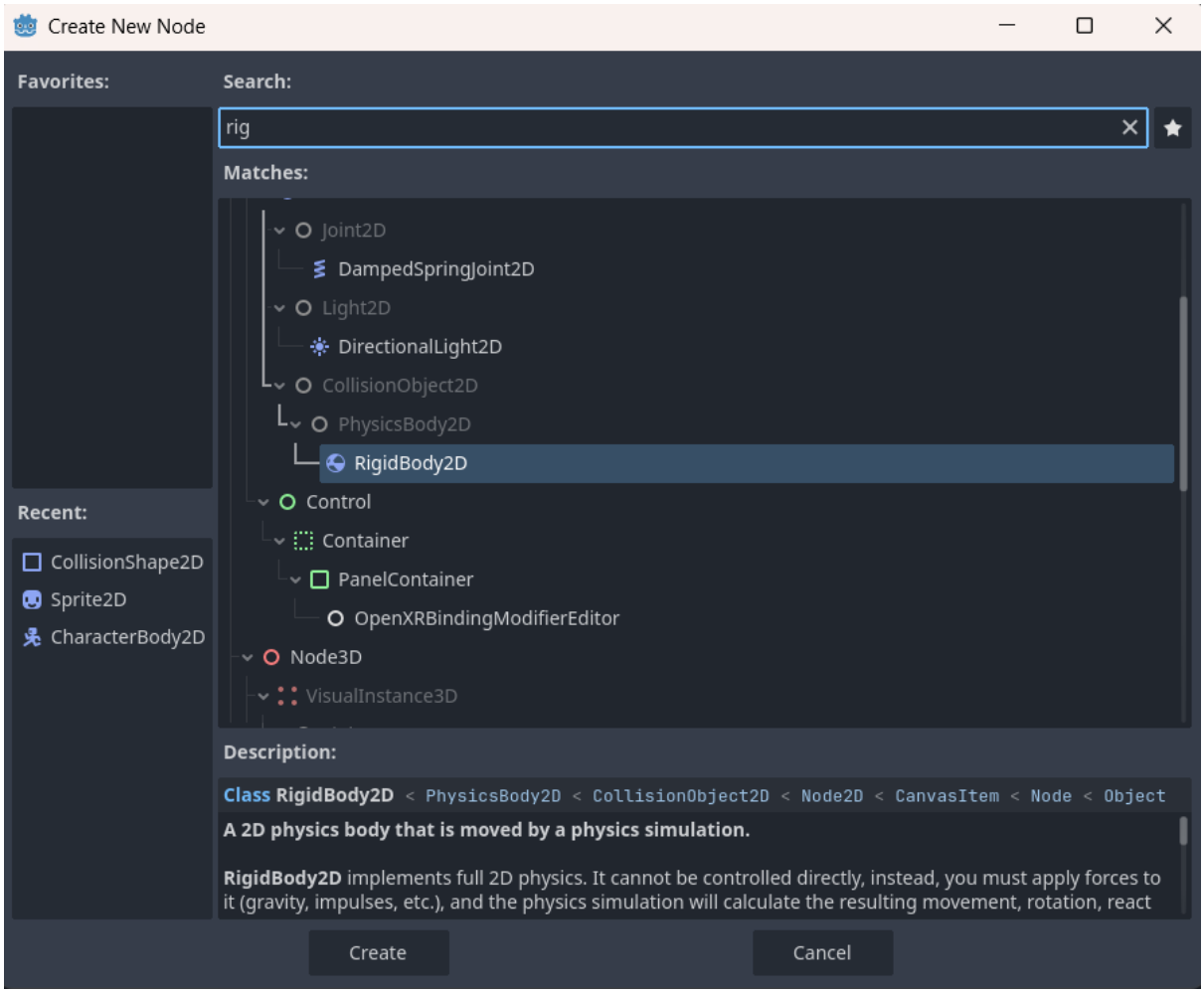
Steps to Add:

1. **Add a RigidBody2D node.**
2. **Add a CollisionShape2D and a Sprite2D.**
3. **Use methods like apply_impulse() or add_force() to move it.**
4. **Avoid manually setting position—let physics handle it.**

Use Case: Physics-based objects like falling crates or bouncing balls.



The body's behavior is also affected by the world's properties, as set in **Project Settings > Physics**, or by entering an Area2D that is overriding the global physics properties.



When a rigid body is at rest and hasn't moved for a while, it goes to sleep. A sleeping body acts like a static body, and its forces are not calculated by the physics engine. The body will wake up when forces are applied, either by a collision or via code

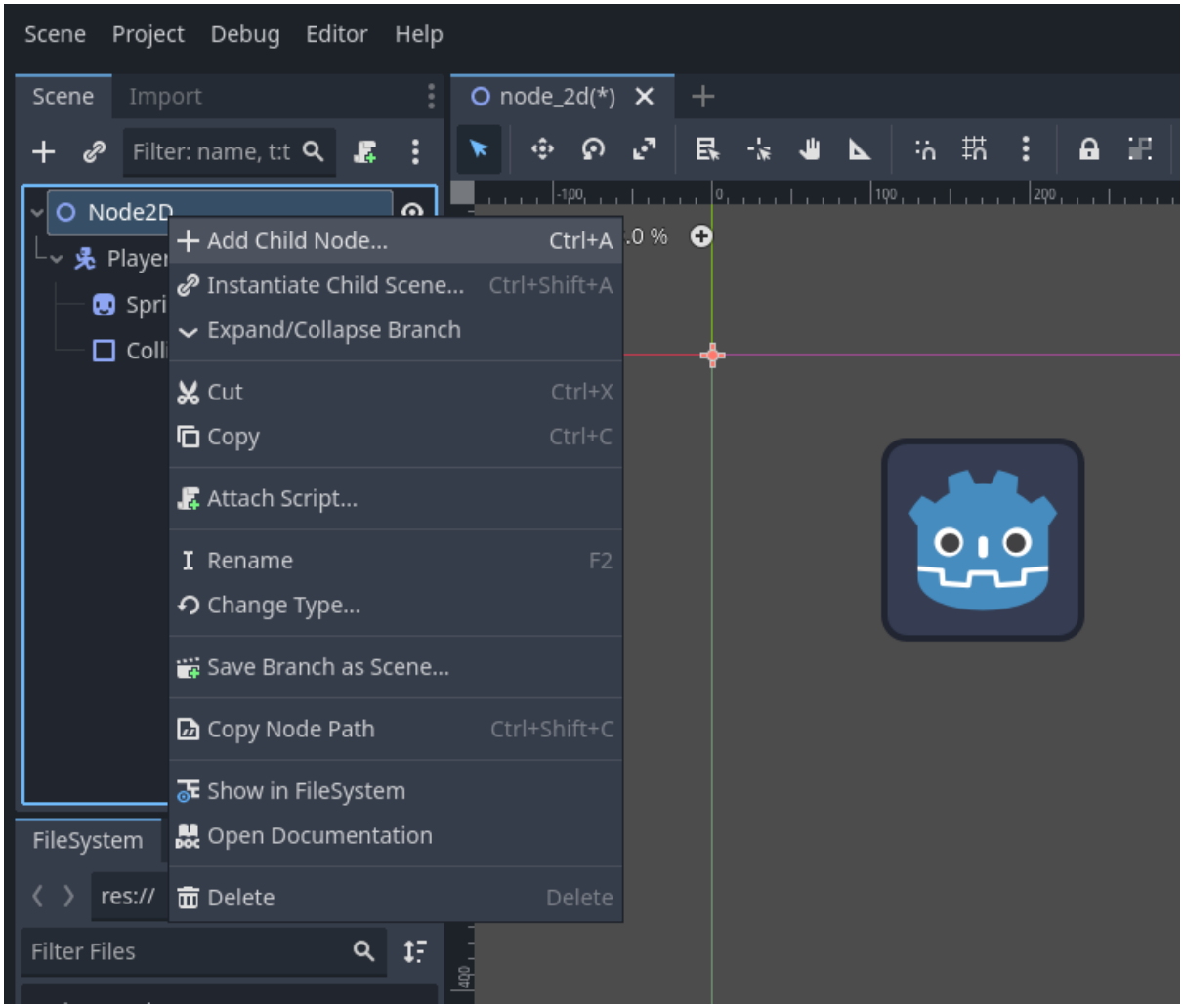
- **CharacterBody2D**

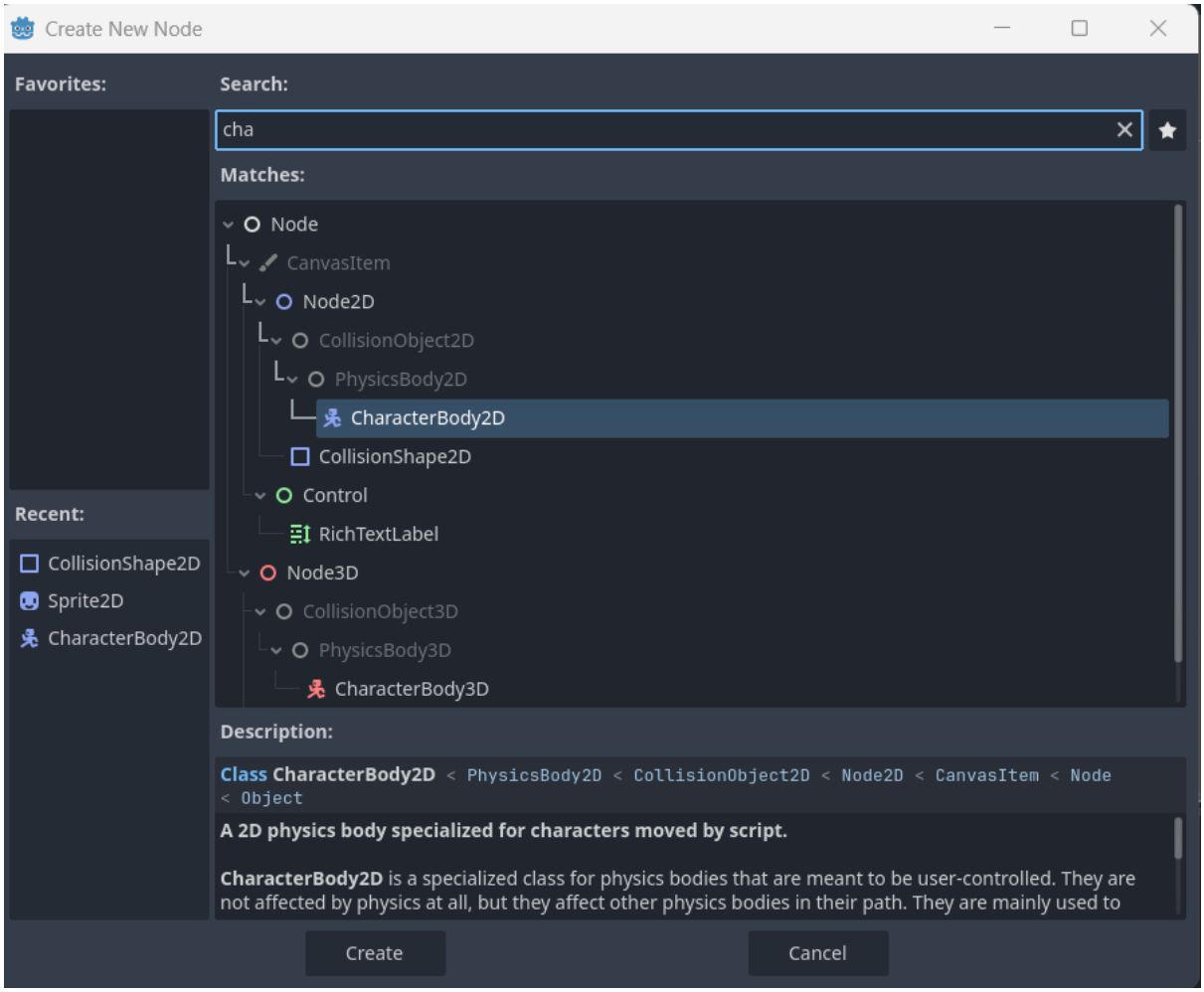
A body that provides collision detection, but no physics. All movement and collision response must be implemented in code.

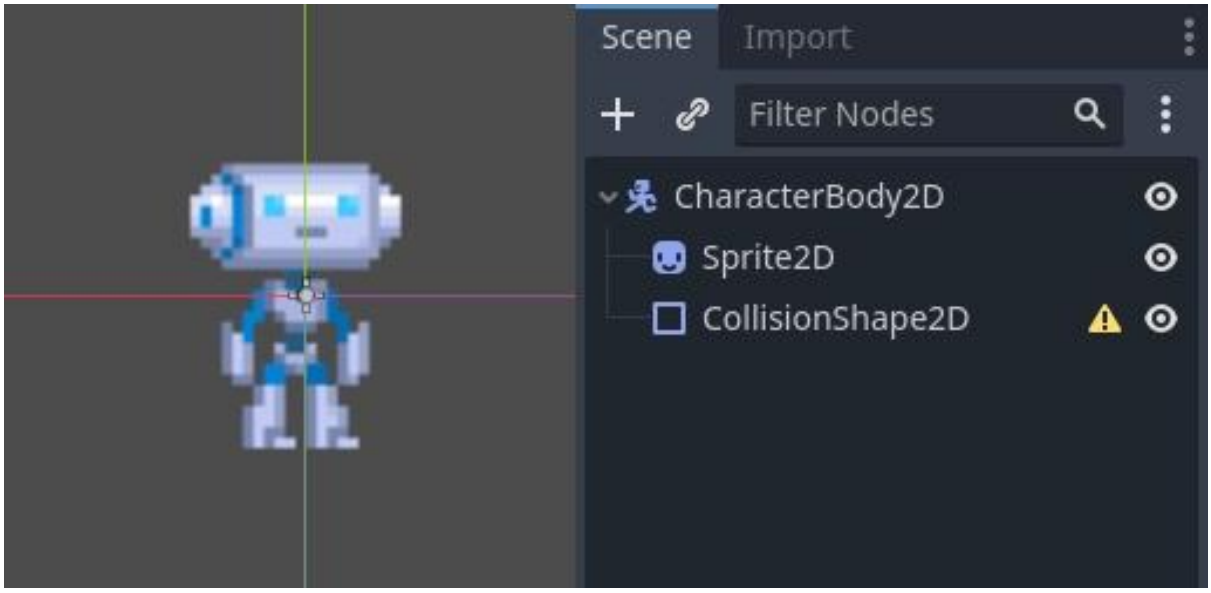
Steps to Add:

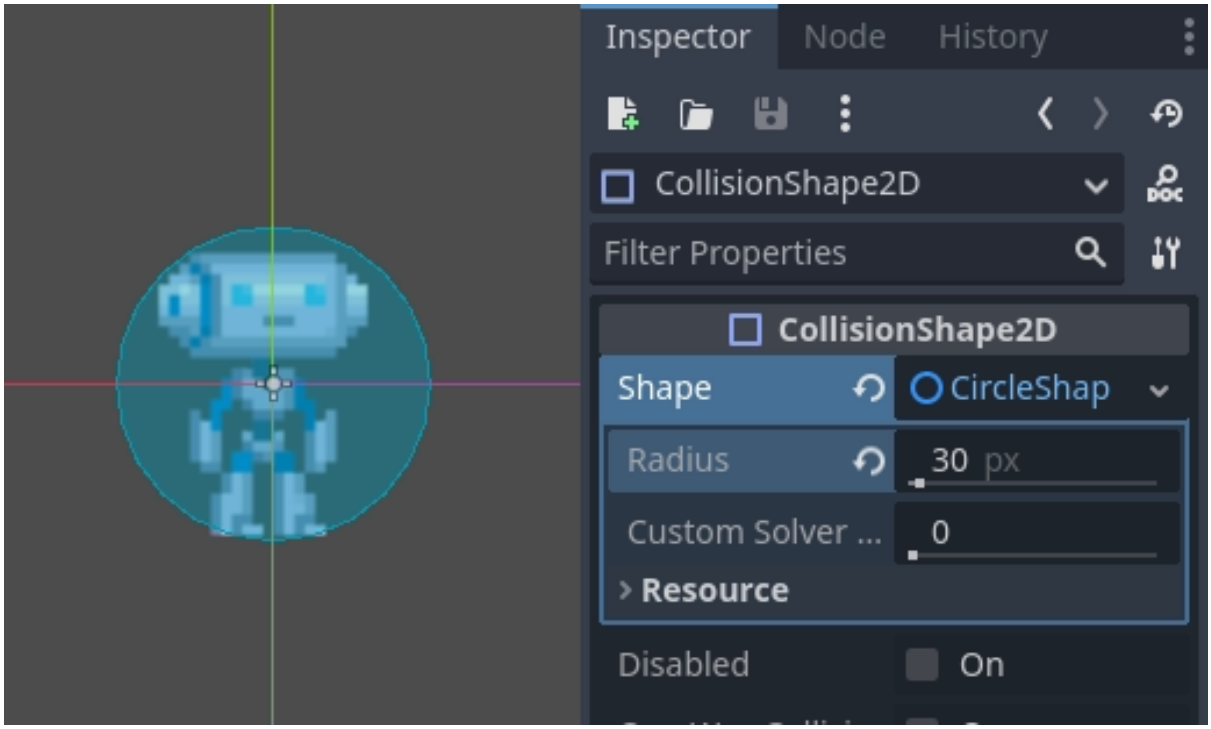
1. Add a CharacterBody2D node.
2. Add a CollisionShape2D and a Sprite2D.
3. Use move_and_slide() or move_and_collide() in _physics_process() for movement.
4. Customize gravity, floor detection, and slope handling.

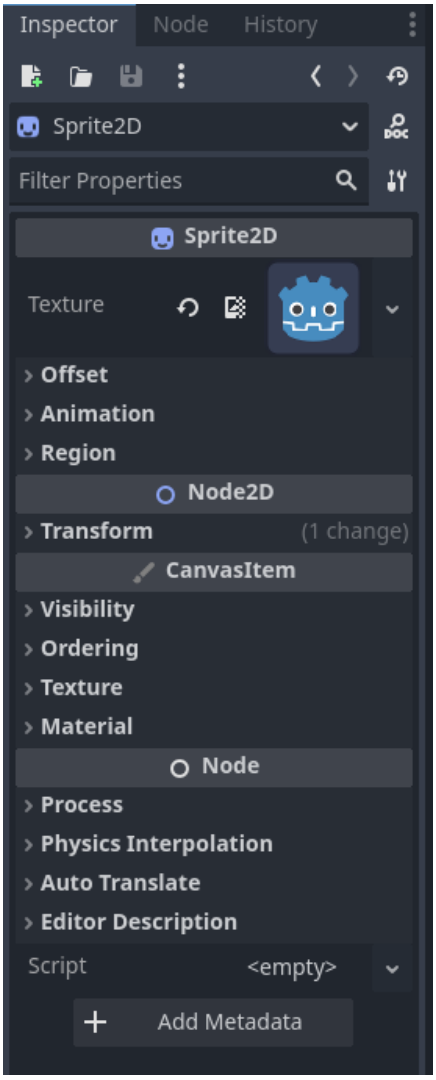
Use Case: Platformer characters, top-down movement, or custom controls.


Scene
Project
Debug
Editor
Help
Scene
Import
node_2d(*)
Filter: name, t:t
Node2D
Player
Spri
Coll
Add Child Node... Ctrl+A
Instantiate Child Scene... Ctrl+Shift+A
Expand/Collapse Branch
Cut Ctrl+X
Copy Ctrl+C
Attach Script...
Rename F2
Change Type...
Save Branch as Scene...
Copy Node Path Ctrl+Shift+C
Show in FileSystem
Open Documentation
Delete Delete
FileSystem
res://
Filter Files


Create New Node
Favorites:
Search:
cha
Matches:
Node
CanvasItem
Node2D
CollisionObject2D
PhysicsBody2D
CharacterBody2D
CollisionShape2D
Control
RichTextLabel
Node3D
CollisionObject3D
PhysicsBody3D
CharacterBody3D
Recent:
CollisionShape2D
Sprite2D
CharacterBody2D
Description:
Class CharacterBody2D < PhysicsBody2D < CollisionObject2D < Node2D < CanvasItem < Node < Object
A 2D physics body specialized for characters moved by script.
CharacterBody2D is a specialized class for physics bodies that are meant to be user-controlled. They are not affected by physics at all, but they affect other physics bodies in their path. They are mainly used to
Create
Cancel


Scene
Import
Filter Nodes
CharacterBody2D
Sprite2D
CollisionShape2D


Inspector
Node
History
CollisionShape2D
Filter Properties
CollisionShape2D
Shape
CircleShap
Radius
30 px
Custom Solver ...
0
Resource
Disabled
On


Inspector
Node
History
Sprite2D
Filter Properties
Sprite2D
Texture
Offset
Animation
Region
Node2D
Transform
(1 change)
CanvasItem
Visibility
Ordering
Texture
Material
Node
Process
Physics Interpolation
Auto Translate
Editor Description
Script
<empty>
Add Metadata

Here’s a quick comparison between the physics nodes:

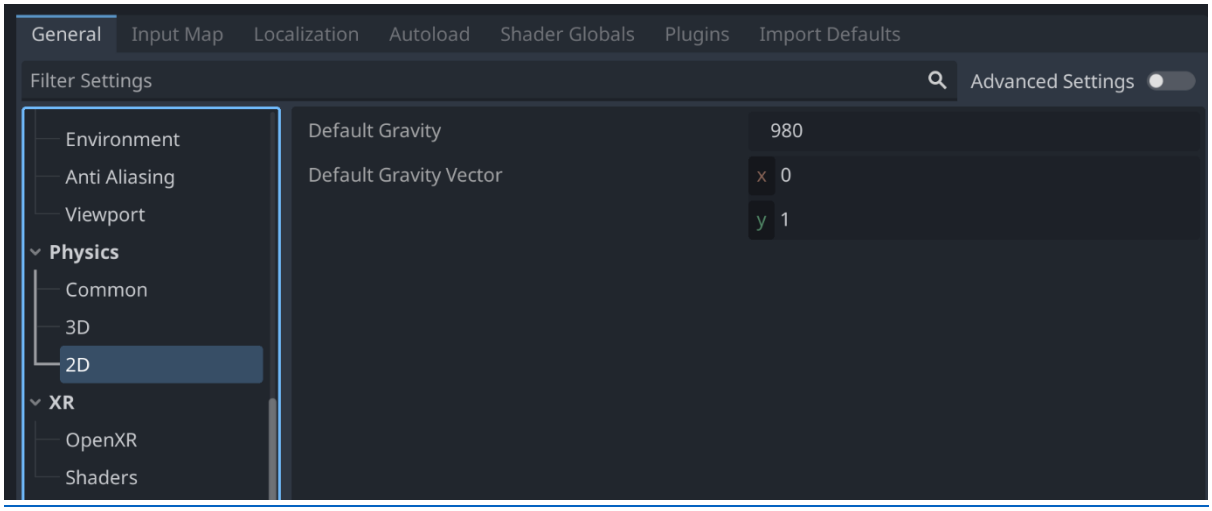
	RigidBody2D	StaticBody2D	Area2D
Collision Detection	Yes	Yes	Yes
Physics Reaction	Yes	Yes	No
Movement simulation	Yes	No	No

Gravity Setting:

Gravity plays an important role in the physics simulation. It controls how fast the node falls and in which direction.

There are two places to set the gravity. The global setting is in the project setting and the individual setting is in the **RigidBody2D** node.

First, take a look of the global setting. Open the **Project Setting** dialog by selecting **Project ▸ Project Settings** in the menu. Then, select the **General** tab. You’ll see the **Default Gravity** and **Default Gravity Vector** properties at the **Physics ▸ 2D** section.

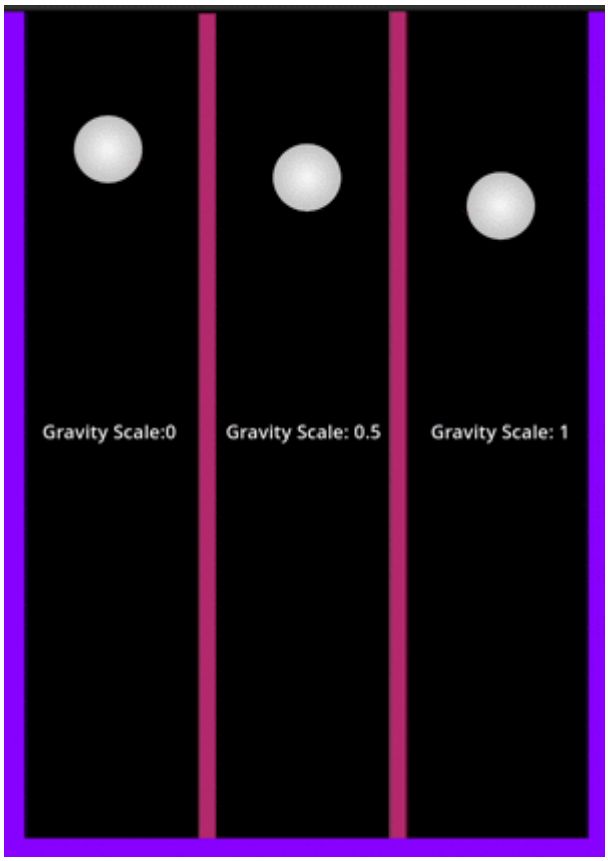


The **Default Gravity** defines how many pixels move per second. The **Default Gravity Vector** defines the direction of the gravity.

In the **RigidBody2D** node, you’ll find the **Gravity Scale** located directly below the **Physics Material Override** property.

The **Gravity Scale** controls the percentage of the global gravity affecting the node. With a value of 1, the node experiences the full force of gravity, while a value of 0 means it is completely unaffected by gravity.

The following demonstration illustrates how different **Gravity Scale** values impact the behavior of the physics node:



Collision shapes (2D)

This guide explains:

- The types of collision shapes available in 2D in Godot.
- Using an image converted to a polygon as a collision shape.
- Performance considerations regarding 2D collisions.

Godot provides many kinds of collision shapes, with different performance and accuracy tradeoffs.

You can define the shape of a PhysicsBody2D by adding one or more CollisionShape2Ds or CollisionPolygon2Ds as *direct* child nodes. Indirect child nodes (i.e. children of child nodes) will be ignored and won't be used as collision shapes. Also, note that you must add a Shape2D *resource* to collision shape nodes in the Inspector dock.

Primitive collision shapes

Godot provides the following primitive collision shape types:

- RectangleShape2D
- CircleShape2D
- CapsuleShape2D
- SegmentShape2D
- SeparationRayShape2D (designed for characters)
- WorldBoundaryShape2D (infinite plane)

You can represent the collision of most smaller objects using one or more primitive shapes. However, for more complex objects, such as a large ship or a whole level, you may need convex or concave shapes instead. More on that below.

We recommend favoring primitive shapes for dynamic objects such as RigidBodies and CharacterBodies as their behavior is the most reliable. They often provide better performance as well.

Kinematic character (2D)

Introduction

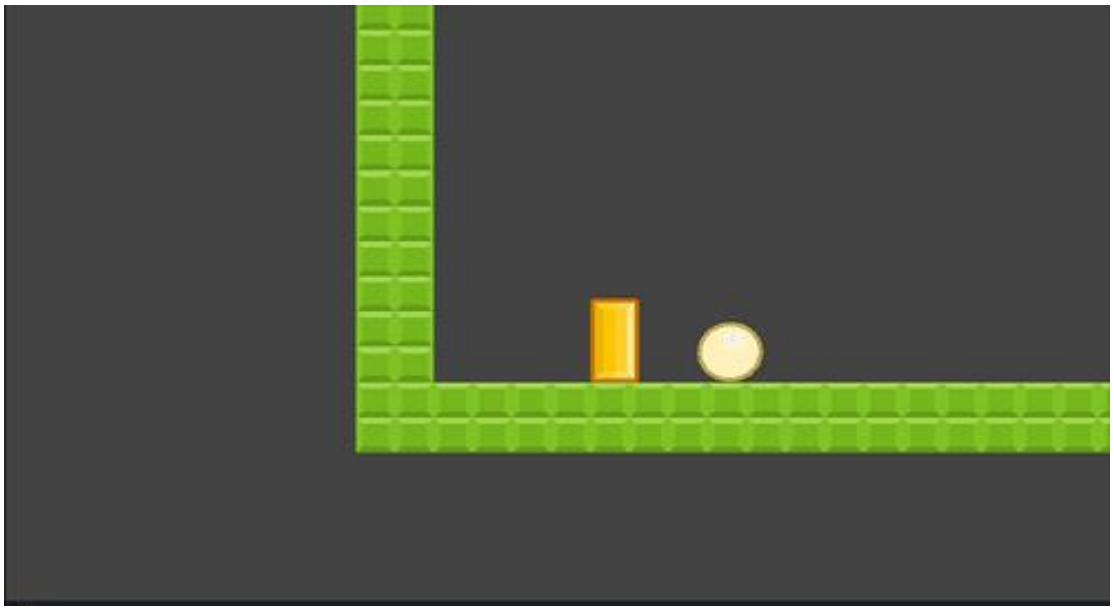
Yes, the name sounds strange. "Kinematic Character". What is that? The reason for the name is that, when physics engines came out, they were called "Dynamics" engines (because they dealt mainly with collision responses). Many attempts were made to create a character controller using the dynamics engines, but it wasn't as easy as it seemed. Godot has one of the best implementations of dynamic character controller you can find (as it can be seen in the 2d/platformer demo), but using it requires a considerable level of skill and understanding of physics engines (or a lot of patience with trial and error).

Some physics engines, such as Havok seem to swear by dynamic character controllers as the best option, while others (PhysX) would rather promote the kinematic one.

So, what is the difference?:

- A **dynamic character controller** uses a rigid body with an infinite inertia tensor. It's a rigid body that can't rotate. Physics engines always let objects move and collide, then solve their collisions all together. This makes dynamic character controllers able to interact with other physics objects seamlessly, as seen in the platformer demo. However, these interactions are not always predictable. Collisions can take more than one frame to be solved, so a few collisions may seem to displace a tiny bit. Those problems can be fixed, but require a certain amount of skill.
- A **kinematic character controller** is assumed to always begin in a non-colliding state, and will always move to a non-colliding state. If it starts in a colliding state, it will try to free itself like rigid bodies do, but this is the exception, not the rule. This makes their control and motion a lot more predictable and easier to program. However, as a downside, they can't directly interact with other physics objects, unless done by hand in code.

This short tutorial focuses on the kinematic character controller. It uses the old-school way of handling collisions, which is not necessarily simpler under the hood, but well hidden and presented as an API.



2.2 Game Assets: Sprites, Textures, Animations, and 3D Models

Getting Started with Godot Engine Assets

When it comes to game development, one thing most professionals agree on is the importance of understanding your tools. The Godot Engine, known for its open-source nature and versatility, is a powerhouse in the indie game development scene. But to really harness its potential, you need to get a grip on Godot Engine assets. Whether you're a seasoned developer or just dipping your toes into game creation, grasping how assets work in Godot can transform your projects. So, let's dive into what Godot Engine assets are all about and how they can take your game development to the next level.

we'll cover the basics of Godot Engine assets, explore different types of assets, and discuss how to manage them effectively. By the end, you'll have a solid understanding of how to leverage these assets to create more efficient and impressive games.

Before we get into the nitty-gritty, let's set the stage. Imagine you're working on a game, and you've got all these amazing ideas for characters, environments, and gameplay mechanics. Godot Engine assets are the building blocks that bring those ideas to life. They include everything from 3D models and textures to scripts and audio files. Understanding how to use these assets effectively can mean the difference between a clunky, unfinished project and a polished, engaging game.

So, what exactly are Godot Engine assets? Basically, they're any files or resources you use in your game. This could be a 3D model of a character, a background music track, or a script that controls gameplay mechanics. Assets are what make your game unique and interactive. But it's not just about having cool stuff in your game; it's about knowing how to organize and use these assets efficiently.

Different Types of Godot Engine Assets

Alright, so we know what assets are. But what kinds of assets are we talking about? Let's break it down.

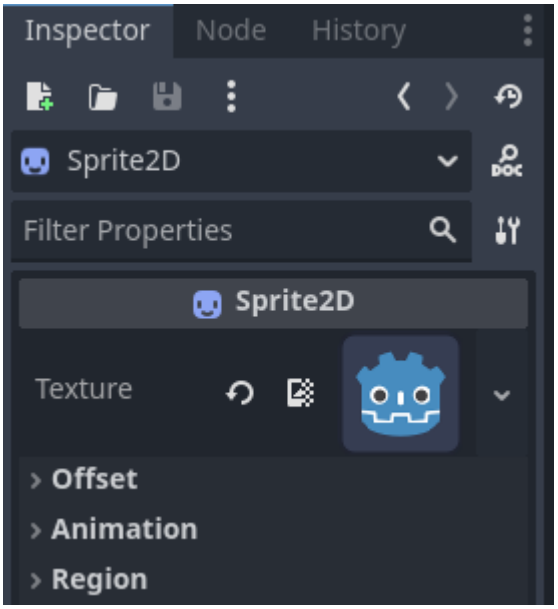
2.2.1 Sprite

A sprite is a 2D graphic object that is integrated into a larger scene. Sprites are a fundamental part of game design. They are the visual elements that the player interacts with and are used to represent characters, items, backgrounds, and more.According to Wikipedia, the term was coined because sprites "float" on top of the background, like spirits or ghosts.Sprites are used in both 2D and 3D games. In 2D games, sprites often make up most of the visual aspect of the game. They can be static images or animated sequences. In 3D games, sprites are often used for certain visual effects, like particles or explosions, or for elements that don't need to be 3D, like the user interface.In Godot, a **Sprite2D** is a Node that displays a 2D texture. You can manipulate its properties to change its position, rotation, scale, etc. You can also change the texture it displays, allowing you to animate the sprite by changing the texture over time. Godot also provides an **AnimatedSprite2D** to display animated textures, and the equivalent **Sprite3D** and **AnimatedSprite3D** nodes for displaying sprites in 3D scenes.

Adding Sprites (2D)

Steps:

1. Open your Godot project and create a new scene.
2. Right click on Node2D and click on Add child Node, Add a Sprite2D node.
3. In the Inspector, click the **Texture** property and load your image file (e.g., PNG).
4. Adjust position, scale, and visibility as needed.



3D Models and Textures

First up, we've got 3D models and textures. These are the visual elements that make up your game world. 3D models are the shapes and forms of objects in your game, like characters, buildings, and vehicles. Textures are the images that give these models their color and detail. Think of it like this: the 3D model is the clay sculpture, and the texture is the paint that brings it to life.

2.2.2. Applying Textures

Steps:

1. For 2D: Use the Sprite2D node and assign a texture.
2. For 3D: Add a MeshInstance3D node.
3. Create a new StandardMaterial3D and assign your texture under the **Albedo** property.

Creating 3D models and textures can be a pretty involved process. You might use software like Blender or Maya to design your models, and then use a program like Photoshop or GIMP to create the textures. Once you've got your models and textures ready, you can import them into Godot and start building your game world.

4. Importing 3D Models

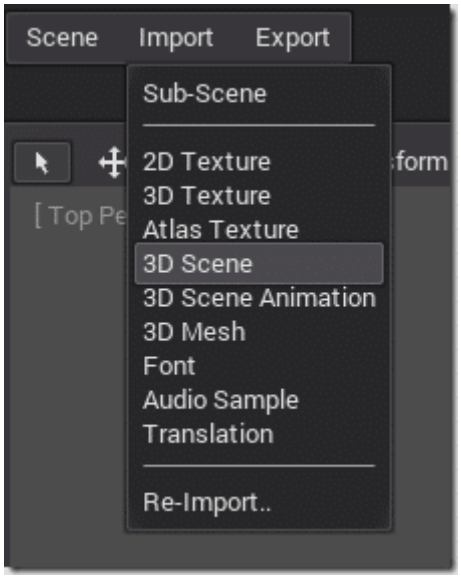
Steps:

1. Drag and drop .glb, .obj, or .dae files into the FileSystem.
2. Add a MeshInstance3D node and assign the imported mesh.
3. Apply materials and textures as needed.
4. Use AnimationPlayer or Skeleton3D for rigged models.

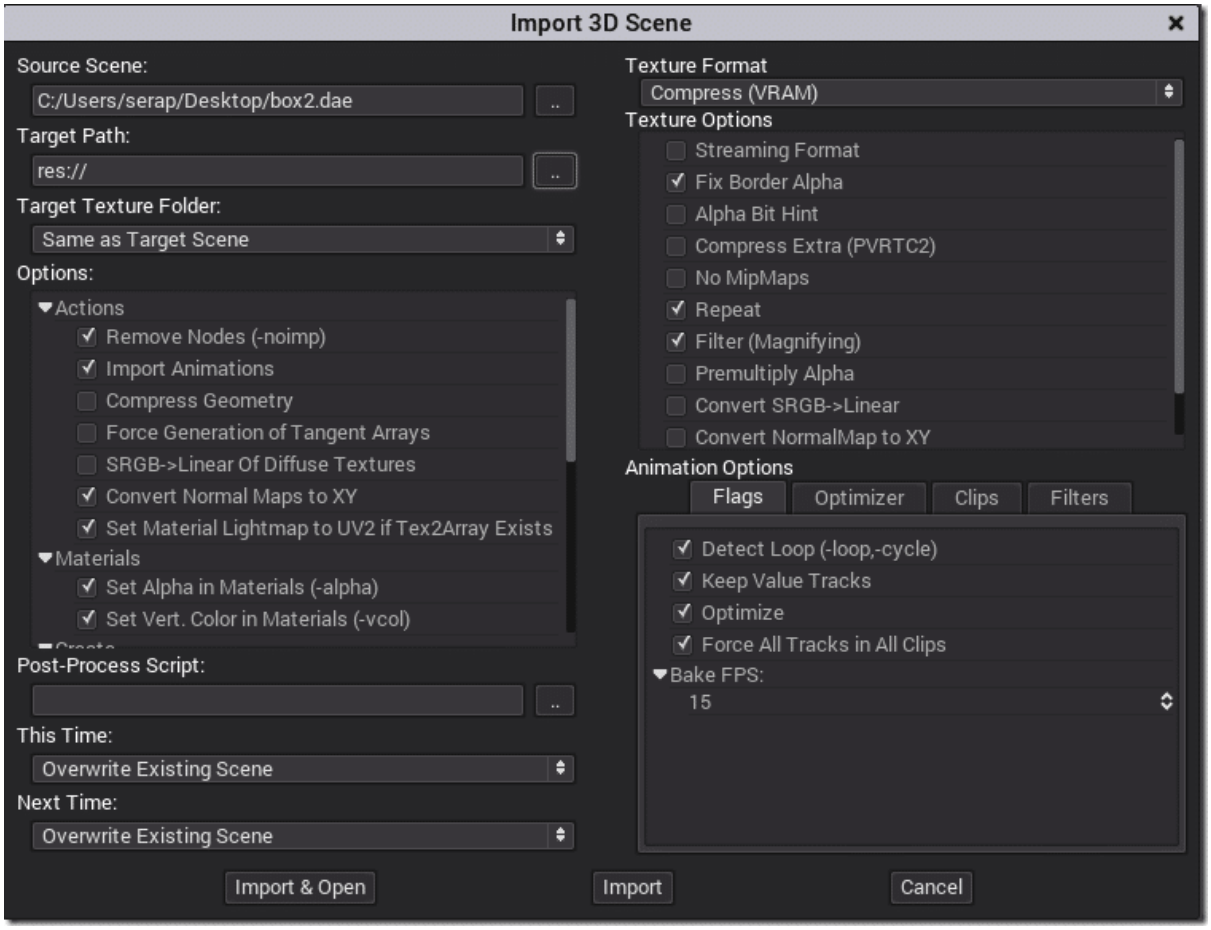
2.2.3 Importing a 3D Model

First let's start with the process of importing a 3D model. In Godot a 3D Model is imported as a scene. This means it will also have all of the miscellaneous things included in your scene, such as lights and cameras. Be sure to remove those or not export them if you do not want them as part of the hierarchy. Also **be sure to save your existing scene** before importing a new model, as unsaved changes will be discarded.

Godot only supports dae (COLLADA format) models. Import by selecting Import->3D Scene.

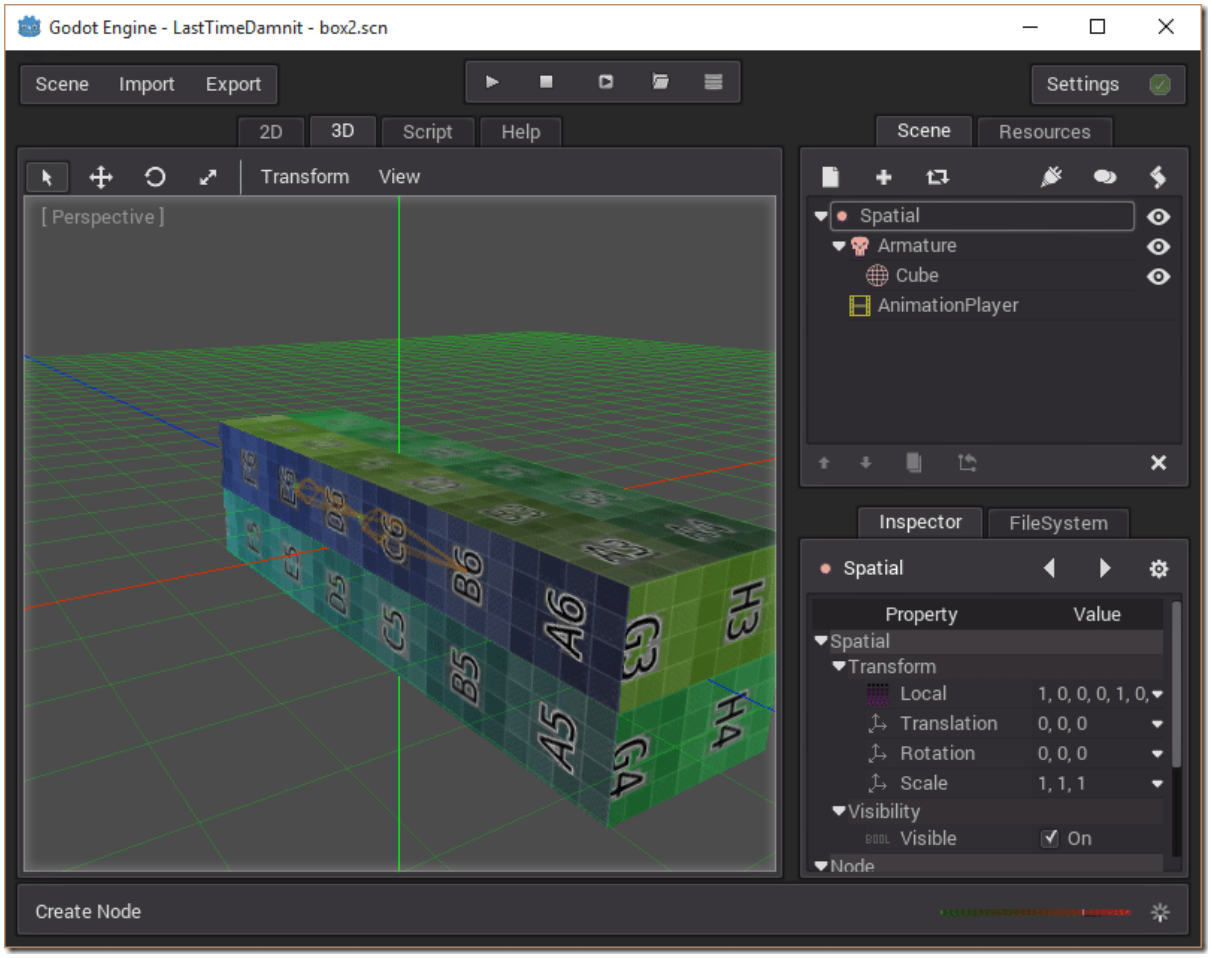


In the resulting dialog, you have to select a file, as well as a location to put it. There are several other things we can set controlling how the mesh is imported, how textures work and even defining animation clips, but the defaults will work for now.



Again, be certain to save your scene before you click Import and Open!

Now your model will load in a new scene. It will default oriented about the origin and there is a good chance your camera will be inside your model, simply use the scroll wheel to zoom out.



Note that the scene has the same name as your model by default. Notice also the hierarchy of nodes it created. At the top is a Spatial, and if you have a skeleton attached and Armature exists with the Mesh(es) as children. If there are animations, there will also be an AnimationPlayer node created. This is what we are going to focus on next.

2.2.4 Animations

Animations are another key type of asset. These are the sequences of images or movements that bring your characters and objects to life. In Godot, you can create animations using the Animation Player node, which allows you to define keyframes and interpolate between them to create smooth motion.

Animations can be used for all sorts of things, like character movements, environmental effects, and user interface elements. By combining animations with scripts and other assets, you can create complex, dynamic gameplay experiences that keep players engaged.

2D sprite animation

Introduction

In this tutorial, you'll learn how to create 2D animated characters with the AnimatedSprite2D class and the AnimationPlayer. Typically, when you create or download an animated character, it will come in one of two ways: as individual images or as a single sprite sheet containing all the animation's frames. Both can be animated in Godot with the AnimatedSprite2D class.

Creating Animations

Steps:

1. Add an AnimatedSprite2D node for 2D or AnimationPlayer for custom animations.
2. Load a sprite sheet or multiple frames.
3. Define animations in the **AnimationPlayer** timeline.
4. Set playback options like looping and speed.

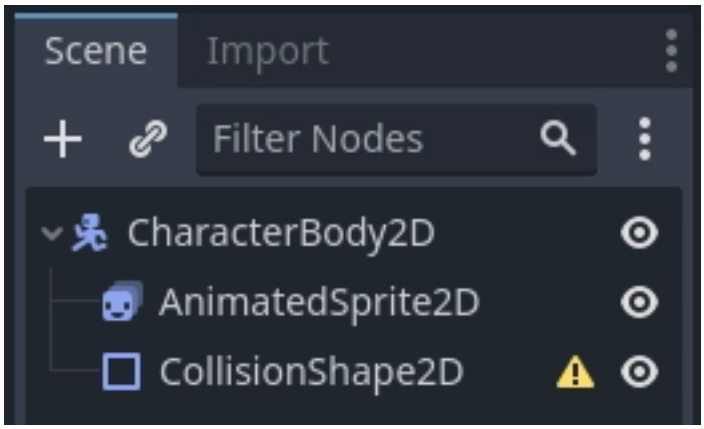
First, we'll use AnimatedSprite2D to animate a collection of individual images. Then we will animate a sprite sheet using this class. Finally, we will learn another way to animate a sprite sheet with AnimationPlayer and the *Animation* property of Sprite2D.

Note

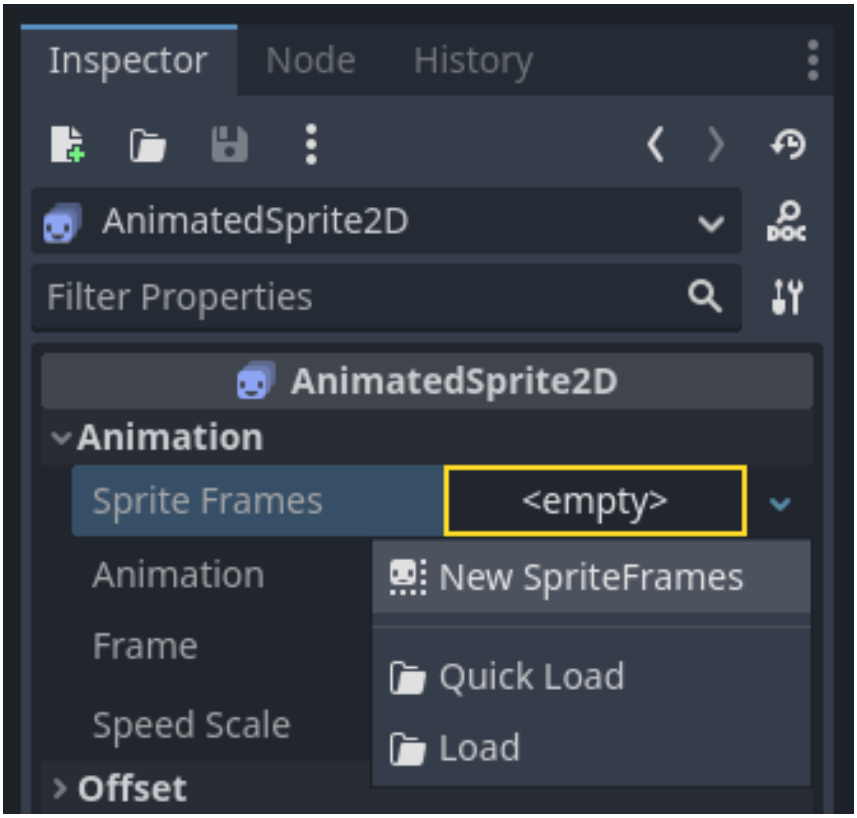
Art for the following examples by https://opengameart.org/users/ansimuz and tgfcoder.

Individual images with AnimatedSprite2D

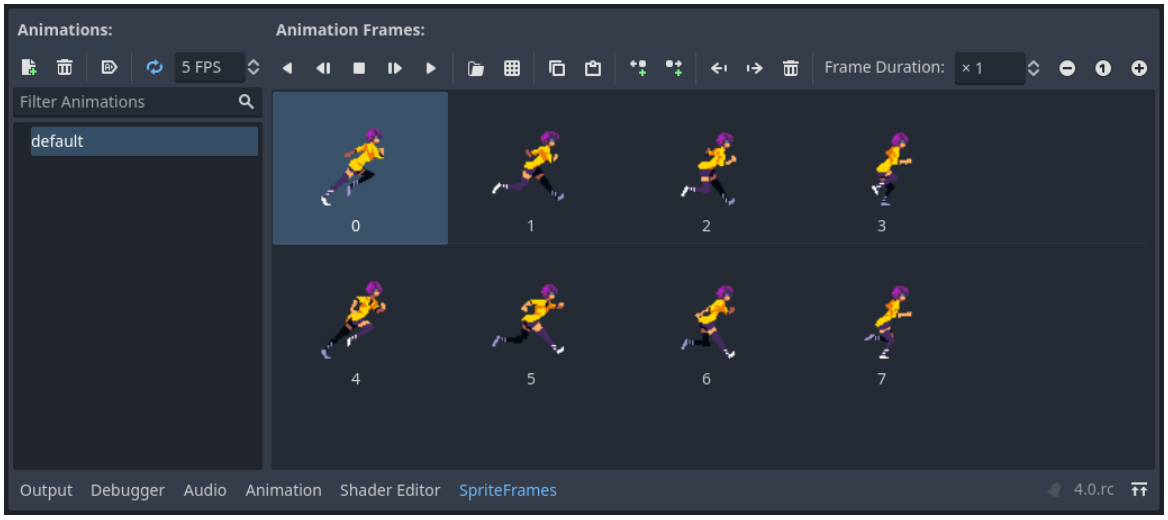
In this scenario, you have a collection of images, each containing one of your character's animation frames. For this example, we'll use the following animation:



Now select the **AnimatedSprite2D** and in its *SpriteFrames* property, select "New SpriteFrames".



Click on the new SpriteFrames resource and you'll see a new panel appear at the bottom of the editor window:



From the FileSystem dock on the left side, drag the 8 individual images into the center part of the SpriteFrames panel. On the left side, change the name of the animation from "default" to "run"

Use the "Play" buttons on the top-right of the *Filter Animations* input to preview the animation. You should now see the animation playing in the viewport. However, it is a bit slow. To fix this, change the *Speed (FPS)* setting in the SpriteFrames panel to 10.

You can add additional animations by clicking the "Add Animation" button and adding additional images.