* **Godot Engine 4.3 documentation in English**

* + [Physics](https://docs.godotengine.org/en/stable/tutorials/physics/index.html)

* + Using CharacterBody2D/3D
  + [Edit on GitHub](https://github.com/godotengine/godot-docs/blob/4.3/tutorials/physics/using_character_body_2d.rst)

[Learn how to contribute!](https://docs.godotengine.org/en/latest/contributing/documentation/index.html#writing-documentation)

**Up to date**

This page is **up to date** for Godot **4.3**. If you still find outdated information, please [open an issue](https://github.com/godotengine/godot-docs).

**Using CharacterBody2D/3D**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#using-characterbody2d-3d)

**Introduction**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#introduction)

Godot offers several collision objects to provide both collision detection and response. Trying to decide which one to use for your project can be confusing. You can avoid problems and simplify development if you understand how each of them works and what their pros and cons are. In this tutorial, we'll look at the [CharacterBody2D](https://docs.godotengine.org/en/stable/classes/class_characterbody2d.html#class-characterbody2d) node and show some examples of how to use it.

**Note**

While this document uses **CharacterBody2D** in its examples, the same concepts apply in 3D as well.

**What is a character body?**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#what-is-a-character-body)

**CharacterBody2D** is for implementing bodies that are controlled via code. Character bodies detect collisions with other bodies when moving, but are not affected by engine physics properties, like gravity or friction. While this means that you have to write some code to create their behavior, it also means you have more precise control over how they move and react.

**Note**

This document assumes you're familiar with Godot's various physics bodies. Please read [Physics introduction](https://docs.godotengine.org/en/stable/tutorials/physics/physics_introduction.html#doc-physics-introduction) first, for an overview of the physics options.

**Tip**

A *CharacterBody2D* can be affected by gravity and other forces, but you must calculate the movement in code. The physics engine will not move a *CharacterBody2D*.

**Movement and collision**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#movement-and-collision)

When moving a **CharacterBody2D**, you should not set its **position** property directly. Instead, you use the **move\_and\_collide()** or **move\_and\_slide()** methods. These methods move the body along a given vector and detect collisions.

**Warning**

You should handle physics body movement in the **\_physics\_process()** callback.

The two movement methods serve different purposes, and later in this tutorial, you'll see examples of how they work.

**move\_and\_collide**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#move-and-collide)

This method takes one required parameter: a [Vector2](https://docs.godotengine.org/en/stable/classes/class_vector2.html#class-vector2) indicating the body's relative movement. Typically, this is your velocity vector multiplied by the frame timestep (**delta**). If the engine detects a collision anywhere along this vector, the body will immediately stop moving. If this happens, the method will return a [KinematicCollision2D](https://docs.godotengine.org/en/stable/classes/class_kinematiccollision2d.html#class-kinematiccollision2d) object.

**KinematicCollision2D** is an object containing data about the collision and the colliding object. Using this data, you can calculate your collision response.

**move\_and\_collide** is most useful when you just want to move the body and detect collision, but don't need any automatic collision response. For example, if you need a bullet that ricochets off a wall, you can directly change the angle of the velocity when you detect a collision. See below for an example.

**move\_and\_slide**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#move-and-slide)

The **move\_and\_slide()** method is intended to simplify the collision response in the common case where you want one body to slide along the other. It is especially useful in platformers or top-down games, for example.

When calling **move\_and\_slide()**, the function uses a number of node properties to calculate its slide behavior. These properties can be found in the Inspector, or set in code.

* **velocity** - *default value:* **Vector2( 0, 0 )**

This property represents the body's velocity vector in pixels per second. **move\_and\_slide()** will modify this value automatically when colliding.

* **motion\_mode** - *default value:* **MOTION\_MODE\_GROUNDED**

This property is typically used to distinguish between side-scrolling and top-down movement. When using the default value, you can use the **is\_on\_floor()**, **is\_on\_wall()**, and **is\_on\_ceiling()** methods to detect what type of surface the body is in contact with, and the body will interact with slopes. When using **MOTION\_MODE\_FLOATING**, all collisions will be considered "walls".

* **up\_direction** - *default value:* **Vector2( 0, -1 )**

This property allows you to define what surfaces the engine should consider being the floor. Its value lets you use the **is\_on\_floor()**, **is\_on\_wall()**, and **is\_on\_ceiling()** methods to detect what type of surface the body is in contact with. The default value means that the top side of horizontal surfaces will be considered "ground".

* **floor\_stop\_on\_slope** - *default value:* **true**

This parameter prevents a body from sliding down slopes when standing still.

* **wall\_min\_slide\_angle** - *default value:* **0.261799** (in radians, equivalent to **15** degrees)

This is the minimum angle where the body is allowed to slide when it hits a slope.

* **floor\_max\_angle** - *default value:* **0.785398** (in radians, equivalent to **45** degrees)

This parameter is the maximum angle before a surface is no longer considered a "floor."

There are many other properties that can be used to modify the body's behavior under specific circumstances. See the [CharacterBody2D](https://docs.godotengine.org/en/stable/classes/class_characterbody2d.html" \l "class-characterbody2d) docs for full details.

**Detecting collisions[ℑ](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html" \l "detecting-collisions" \o "Link to this heading)**

When using **move\_and\_collide()** the function returns a **KinematicCollision2D** directly, and you can use this in your code.

When using **move\_and\_slide()** it's possible to have multiple collisions occur, as the slide response is calculated. To process these collisions, use **get\_slide\_collision\_count()** and **get\_slide\_collision()**:

GDScriptC#

*# Using move\_and\_collide.*

**var** collision = move\_and\_collide(velocity \* delta)

**if** collision:

print("I collided with ", collision.get\_collider().name)

*# Using move\_and\_slide.*

move\_and\_slide()

**for** i **in** get\_slide\_collision\_count():

**var** collision = get\_slide\_collision(i)

print("I collided with ", collision.get\_collider().name)

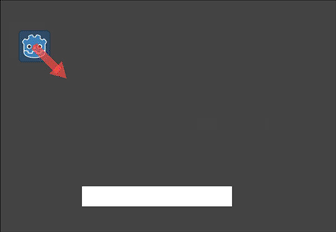
**Note**

*get\_slide\_collision\_count()* only counts times the body has collided and changed direction.

See [KinematicCollision2D](https://docs.godotengine.org/en/stable/classes/class_kinematiccollision2d.html#class-kinematiccollision2d) for details on what collision data is returned.

**Which movement method to use?**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#which-movement-method-to-use)

A common question from new Godot users is: "How do you decide which movement function to use?" Often, the response is to use **move\_and\_slide()** because it seems simpler, but this is not necessarily the case. One way to think of it is that **move\_and\_slide()** is a special case, and **move\_and\_collide()** is more general. For example, the following two code snippets result in the same collision response:



GDScriptC#

*# using move\_and\_collide*

**var** collision = move\_and\_collide(velocity \* delta)

**if** collision:

velocity = velocity.slide(collision.get\_normal())

*# using move\_and\_slide*

move\_and\_slide()

Anything you do with **move\_and\_slide()** can also be done with **move\_and\_collide()**, but it might take a little more code. However, as we'll see in the examples below, there are cases where **move\_and\_slide()** doesn't provide the response you want.

In the example above, **move\_and\_slide()** automatically alters the **velocity** variable. This is because when the character collides with the environment, the function recalculates the speed internally to reflect the slowdown.

For example, if your character fell on the floor, you don't want it to accumulate vertical speed due to the effect of gravity. Instead, you want its vertical speed to reset to zero.

**move\_and\_slide()** may also recalculate the kinematic body's velocity several times in a loop as, to produce a smooth motion, it moves the character and collides up to five times by default. At the end of the process, the character's new velocity is available for use on the next frame.

**Examples**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#examples)

To see these examples in action, download the sample project: [character\_body\_2d\_starter.zip](https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/character_body_2d_starter.zip)

**Movement and walls**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#movement-and-walls)

If you've downloaded the sample project, this example is in "basic\_movement.tscn".

For this example, add a **CharacterBody2D** with two children: a **Sprite2D** and a **CollisionShape2D**. Use the Godot "icon.svg" as the Sprite2D's texture (drag it from the Filesystem dock to the *Texture* property of the **Sprite2D**). In the **CollisionShape2D**'s *Shape* property, select "New RectangleShape2D" and size the rectangle to fit over the sprite image.

**Note**

See [2D movement overview](https://docs.godotengine.org/en/stable/tutorials/2d/2d_movement.html" \l "doc-2d-movement) for examples of implementing 2D movement schemes.

Attach a script to the CharacterBody2D and add the following code:

GDScriptC#

**extends** CharacterBody2D

**var** speed = 300

**func** get\_input():

**var** input\_dir = Input.get\_vector("ui\_left", "ui\_right", "ui\_up", "ui\_down")

velocity = input\_dir \* speed

**func** \_physics\_process(delta):

get\_input()

move\_and\_collide(velocity \* delta)

Run this scene and you'll see that **move\_and\_collide()** works as expected, moving the body along the velocity vector. Now let's see what happens when you add some obstacles. Add a [StaticBody2D](https://docs.godotengine.org/en/stable/classes/class_staticbody2d.html" \l "class-staticbody2d) with a rectangular collision shape. For visibility, you can use a Sprite2D, a Polygon2D, or turn on "Visible Collision Shapes" from the "Debug" menu.

Run the scene again and try moving into the obstacle. You'll see that the **CharacterBody2D** can't penetrate the obstacle. However, try moving into the obstacle at an angle and you'll find that the obstacle acts like glue - it feels like the body gets stuck.

This happens because there is no *collision response*. **move\_and\_collide()** stops the body's movement when a collision occurs. We need to code whatever response we want from the collision.

Try changing the function to **move\_and\_slide()** and running again.

**move\_and\_slide()** provides a default collision response of sliding the body along the collision object. This is useful for a great many game types, and may be all you need to get the behavior you want.

**Bouncing/reflecting**[**ℑ**](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html#bouncing-reflecting)

What if you don't want a sliding collision response? For this example ("bounce\_and\_collide.tscn" in the sample project), we have a character shooting bullets and we want the bullets to bounce off the walls.

This example uses three scenes. The main scene contains the Player and Walls. The Bullet and Wall are separate scenes so that they can be instanced.

The Player is controlled by the **w** and **s** keys for forward and back. Aiming uses the mouse pointer. Here is the code for the Player, using **move\_and\_slide()**:

GDScriptC#

**extends** CharacterBody2D

**var** Bullet = preload("res://bullet.tscn")

**var** speed = 200

**func** get\_input():

*# Add these actions in Project Settings -> Input Map.*

**var** input\_dir = Input.get\_axis("backward", "forward")

velocity = transform.x \* input\_dir \* speed

**if** Input.is\_action\_just\_pressed("shoot"):

shoot()

**func** shoot():

*# "Muzzle" is a Marker2D placed at the barrel of the gun.*

**var** b = Bullet.instantiate()

b.start($Muzzle.global\_position, rotation)

get\_tree().root.add\_child(b)

**func** \_physics\_process(delta):

get\_input()

**var** dir = get\_global\_mouse\_position() - global\_position

*# Don't move if too close to the mouse pointer.*

**if** dir.length() > 5:

rotation = dir.angle()

move\_and\_slide()

And the code for the Bullet:

GDScriptC#

**extends** CharacterBody2D

**var** speed = 750

**func** start(\_position, \_direction):

rotation = \_direction

position = \_position

velocity = Vector2(speed, 0).rotated(rotation)

**func** \_physics\_process(delta):

**var** collision = move\_and\_collide(velocity \* delta)

**if** collision:

velocity = velocity.bounce(collision.get\_normal())

**if** collision.get\_collider().has\_method("hit"):

collision.get\_collider().hit()

**func** \_on\_VisibilityNotifier2D\_screen\_exited():

*# Deletes the bullet when it exits the screen.*

queue\_free()

The action happens in **\_physics\_process()**. After using **move\_and\_collide()**, if a collision occurs, a **KinematicCollision2D** object is returned (otherwise, the return is **null**).

If there is a returned collision, we use the **normal** of the collision to reflect the bullet's **velocity** with the **Vector2.bounce()** method.

If the colliding object (**collider**) has a **hit** method, we also call it. In the example project, we've added a flashing color effect to the Wall to demonstrate this.



**Platformer movement[ℑ](https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html" \l "platformer-movement" \o "Link to this heading)**

Let's try one more popular example: the 2D platformer. **move\_and\_slide()** is ideal for quickly getting a functional character controller up and running. If you've downloaded the sample project, you can find this in "platformer.tscn".

For this example, we'll assume you have a level made of one or more **StaticBody2D** objects. They can be any shape and size. In the sample project, we're using [Polygon2D](https://docs.godotengine.org/en/stable/classes/class_polygon2d.html" \l "class-polygon2d) to create the platform shapes.

Here's the code for the player body:

GDScriptC#

**extends** CharacterBody2D

**var** speed = 300.0

**var** jump\_speed = -400.0

*# Get the gravity from the project settings so you can sync with rigid body nodes.*

**var** gravity = ProjectSettings.get\_setting("physics/2d/default\_gravity")

**func** \_physics\_process(delta):

*# Add the gravity.*

velocity.y += gravity \* delta

*# Handle Jump.*

**if** Input.is\_action\_just\_pressed("jump") **and** is\_on\_floor():

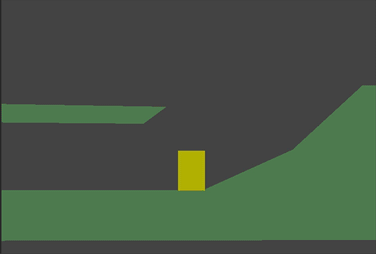
velocity.y = jump\_speed

*# Get the input direction.*

**var** direction = Input.get\_axis("ui\_left", "ui\_right")

velocity.x = direction \* speed

move\_and\_slide()



In this code we're using **move\_and\_slide()** as described above - to move the body along its velocity vector, sliding along any collision surfaces such as the ground or a platform. We're also using **is\_on\_floor()** to check if a jump should be allowed. Without this, you'd be able to "jump" in midair; great if you're making Flappy Bird, but not for a platformer game.

There is a lot more that goes into a complete platformer character: acceleration, double-jumps, coyote-time, and many more. The code above is just a starting point. You can use it as a base to expand into whatever movement behavior you need for your own projects.