Physics

Physics introduction

In game development, you often need to know when two objects in the game intersect or come into contact. This is known as **collision detection**. When a collision is detected, you typically want something to happen. This is known as **collision response**.

Godot offers a number of collision objects in 2D and 3D 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 works and what their pros and cons are.

In this guide, you will learn:

- Godot's four collision object types
- How each collision object works
- When and why to choose one type over another

Note

This document's examples will use 2D objects. Every 2D physics object and collision shape has a direct equivalent in 3D and in most cases they work in much the same way.

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.

- 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 behavior.

- RigidBody2D
  This is the node that implements simulated 2D physics. You do not control a `RigidBody2D` directly, but instead you apply forces to it (gravity, impulses, etc.) and the physics engine calculates the resulting movement. Read more about using rigid bodies.

- CharacterBody2D
  A body that provides collision detection, but no physics. All movement and collision response must be implemented in code.

Physics material

Static bodies and rigid bodies can be configured to use a PhysicsMaterial. This allows adjusting the friction and bounce of an object, and set if it's absorbent and/or rough.

Collision shapes

A physics body can hold any number of Shape2D objects as children. These shapes are used to define the object's collision bounds and to detect contact with other objects.

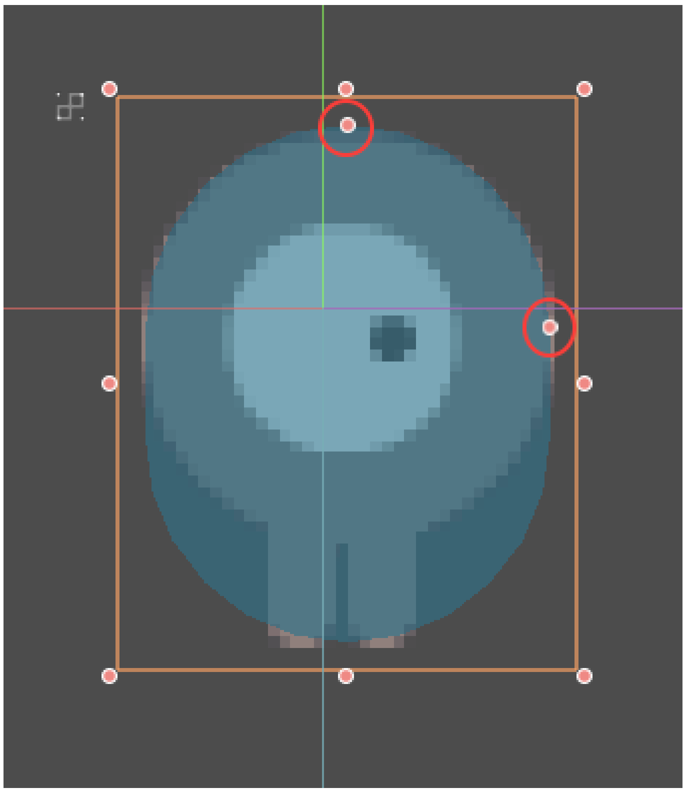
Note

In order to detect collisions, at least one `Shape2D` must be assigned to the object.

The most common way to assign a shape is by adding a CollisionShape2D or CollisionPolygon2D as a child of the object. These nodes allow you to draw the shape directly in the editor workspace.

Important

Be careful to never scale your collision shapes in the editor. The "Scale" property in the Inspector should remain `(1, 1)`. When changing the size of the collision shape, you should always use the size handles, **not** the `Node2D` scale handles. Scaling a shape can result in unexpected collision behavior.



Physics process callback

The physics engine runs at a fixed rate (a default of 60 iterations per second). This rate is typically different from the frame rate which fluctuates based on what is rendered and available resources.

It is important that all physics related code runs at this fixed rate. Therefore Godot differentiates between physics and idle processing. Code that runs each frame is called idle processing and code that runs on each physics tick is called physics processing. Godot provides two different callbacks, one for each of those processing rates.

The physics callback, Node._physics_process(), is called before each physics step. Any code that needs to access a body's properties should be run in here. This method will be passed a `delta` parameter, which is a floating-point number equal to the time passed in *seconds* since the last step. When using the default 60 Hz physics update rate, it will typically be equal to `0.01666...` (but not always, see below).

Note

It's recommended to always use the `delta` parameter when relevant in your physics calculations, so that the game behaves correctly if you change the physics update rate or if the player's device can't keep up.

Collision layers and masks

One of the most powerful, but frequently misunderstood, collision features is the collision layer system. This system allows you to build up complex interactions between a variety of objects. The key concepts are **layers** and **masks**. Each `CollisionObject2D` has 32 different physics layers it can interact with.

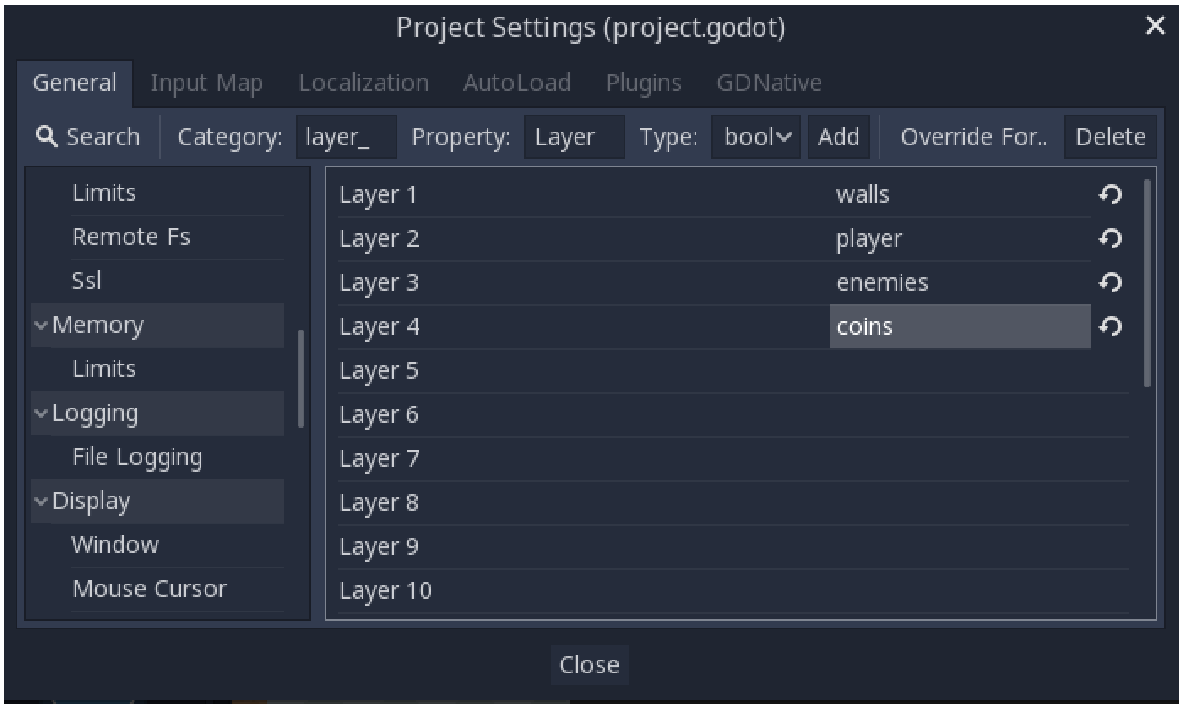
Let's look at each of the properties in turn:

- collision_layer
  This describes the layers that the object appears **in**. By default, all bodies are on layer `1`.
- collision_mask

> This describes what layers the body will **scan** for collisions. If an object isn't in one of the mask layers, the body will ignore it. By default, all bodies scan layer 1.

These properties can be configured via code, or by editing them in the Inspector.

Keeping track of what you're using each layer for can be difficult, so you may find it useful to assign names to the layers you're using. Names can be assigned in Project Settings -> Layer Names.

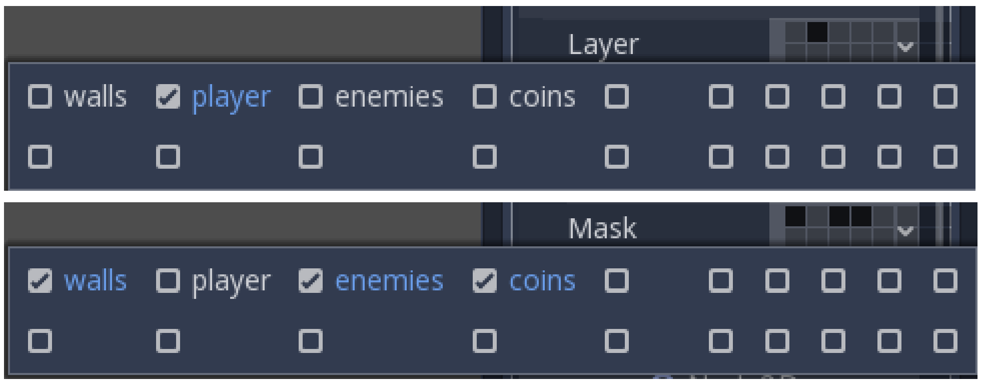


Layer 1	walls
Layer 2	player
Layer 3	enemies
Layer 4	coins
Layer 5	
Layer 6	
Layer 7	
Layer 8	
Layer 9	
Layer 10	

GUI example

You have four node types in your game: Walls, Player, Enemy, and Coin. Both Player and Enemy should collide with Walls. The Player node should detect collisions with both Enemy and Coin, but Enemy and Coin should ignore each other.

Start by naming layers 1-4 "walls", "player", "enemies", and "coins" and place each node type in its respective layer using the "Layer" property. Then set each node's "Mask" property by selecting the layers it should interact with. For example, the Player's settings would look like this:


Layer

☐ walls ☑ player ☐ enemies ☐ coins ☐ ☐ ☐ ☐ ☐ ☐

☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐

Mask

☑ walls ☐ player ☑ enemies ☑ coins ☐ ☐ ☐ ☐ ☐ ☐

☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐

Code example

In function calls, layers are specified as a bitmask. Where a function enables all layers by default, the layer mask will be given as `0xffffffff`. Your code can use binary, hexadecimal, or decimal notation for layer masks, depending on your preference.

The code equivalent of the above example where layers 1, 3 and 4 were enabled would be as follows:

```
# Example: Setting mask value for enabling layers 1, 3 and 4

# Binary - set the bit corresponding to the layers you want to enable (1, 3, and 4) to 1, set all other bits to 0.
# Note: Layer 32 is the first bit, layer 1 is the last. The mask for layers 4,3 and 1 is therefore
0b00000000_00000000_00000000_00001101
# (This can be shortened to 0b1101)

# Hexadecimal equivalent (1101 binary converted to hexadecimal)
0x000d
# (This value can be shortened to 0xd)

# Decimal - Add the results of 2 to the power of (layer to be enabled - 1).
# (2^(1-1)) + (2^(3-1)) + (2^(4-1)) = 1 + 4 + 8 = 13
pow(2, 1-1) + pow(2, 3-1) + pow(2, 4-1)
```

Area2D

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

There are three main uses for Area2D:

- Overriding physics parameters (such as gravity) in a given region.
- Detecting when other bodies enter or exit a region or what bodies are currently in a region.
- Checking other areas for overlap.

By default, areas also receive mouse and touchscreen input.

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. However, it can impart motion or rotation to a colliding body **as if** it were moving, using its `constant_linear_velocity` and `constant_angular_velocity` properties.

`StaticBody2D` nodes are most often used for objects that are part of the environment or that do not need to have any dynamic behavior.

Example uses for `StaticBody2D`:

- Platforms (including moving platforms)
- Conveyor belts
- Walls and other obstacles

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 behavior via properties such as "Mass", "Friction", or "Bounce", which can be set in the Inspector.

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.

Using RigidBody2D

One of the benefits of using a rigid body is that a lot of behavior can be had "for free" without writing any code. For example, if you were making an "Angry Birds"-style game with falling blocks, you would only need to create RigidBody2Ds and adjust their properties. Stacking, falling, and bouncing would automatically be calculated by the physics engine.

However, if you do wish to have some control over the body, you should take care - altering the `position`, `linear_velocity`, or other physics properties of a rigid body can result in unexpected behavior. If you need to alter any of the physics-related properties, you should use the _integrate_forces() callback instead of `_physics_process()`. In this callback, you have access to the body's PhysicsDirectBodyState2D, which allows for safely changing properties and synchronizing them with the physics engine.

For example, here is the code for an "Asteroids" style spaceship:

GDScript

```
extends RigidBody2D

var thrust = Vector2(0, -250)
var torque = 20000

func _integrate_forces(state):
    if Input.is_action_pressed("ui_up"):
        state.apply_force(thrust.rotated(rotation))
    else:
        state.apply_force(Vector2())
    var rotation_direction = 0
    if Input.is_action_pressed("ui_right"):
        rotation_direction += 1
    if Input.is_action_pressed("ui_left"):
        rotation_direction -= 1
    state.apply_torque(rotation_direction * torque)
```

C#

```
using Godot;

public partial class Spaceship : RigidBody2D
{
    private Vector2 _thrust = new Vector2(0, -250);
    private float _torque = 20000;

    public override void
_IntegrateForces(PhysicsDirectBodyState2D state)
    {
        if (Input.IsActionPressed("ui_up"))
            state.ApplyForce(_thrust.Rotated(Rotation));
        else
            state.ApplyForce(new Vector2());

        var rotationDir = 0;
        if (Input.IsActionPressed("ui_right"))
            rotationDir += 1;
        if (Input.IsActionPressed("ui_left"))
            rotationDir -= 1;
        state.ApplyTorque(rotationDir * _torque);
    }
}
```

Note that we are not setting the `linear_velocity` or `angular_velocity` properties directly, but rather applying forces (`thrust` and `torque`) to the body and letting the physics engine calculate the resulting movement.

Note

When a rigid body goes to sleep, the `_integrate_forces()` function will not be called. To override this behavior, you will need to keep the body awake by creating a collision, applying a force to it, or by disabling the can_sleep property. Be aware that this can have a negative effect on performance.

Contact reporting

By default, rigid bodies do not keep track of contacts, because this can require a huge amount of memory if many bodies are in the scene. To enable contact reporting, set the max_contacts_reported property to a non-zero value. The contacts can then be obtained via PhysicsDirectBodyState2D.get_contact_count() and related functions.

Contact monitoring via signals can be enabled via the contact_monitor property. See RigidBody2D for the list of available signals.

CharacterBody2D

CharacterBody2D bodies detect collisions with other bodies, but are not affected by physics properties like gravity or friction. Instead, they must be controlled by the user via code. The physics engine will not move a character body.

When moving a character body, you should not set its `position` directly. Instead, you use the `move_and_collide()` or `move_and_slide()` methods. These methods move the body along a given vector, and it will instantly stop if a collision is detected with another body. After the body has collided, any collision response must be coded manually.

Character collision response

After a collision, you may want the body to bounce, to slide along a wall, or to alter the properties of the object it hit. The way you handle collision response depends on which method you used to move the CharacterBody2D.

move_and_collide

When using `move_and_collide()`, the function returns a KinematicCollision2D object, which contains information about the collision and the colliding body. You can use this information to determine the response.

For example, if you want to find the point in space where the collision occurred:

GDScript

```
extends PhysicsBody2D

var velocity = Vector2(250, 250)

func _physics_process(delta):
    var collision_info = move_and_collide(velocity * delta)
    if collision_info:
        var collision_point = collision_info.get_position()
```

C#

```
using Godot;

public partial class Body : PhysicsBody2D
{
    private Vector2 _velocity = new Vector2(250, 250);

    public override void _PhysicsProcess(double delta)
    {
        var collisionInfo = MoveAndCollide(_velocity *
(float)delta);
        if (collisionInfo != null)
        {
            var collisionPoint = collisionInfo.GetPosition();
        }
    }
}
```

Or to bounce off of the colliding object:

GDScript

```
extends PhysicsBody2D

var velocity = Vector2(250, 250)

func _physics_process(delta):
    var collision_info = move_and_collide(velocity * delta)
    if collision_info:
        velocity =
velocity.bounce(collision_info.get_normal())
```

C#

```
using Godot;

public partial class Body : PhysicsBody2D
{
    private Vector2 _velocity = new Vector2(250, 250);

    public override void _PhysicsProcess(double delta)
    {
        var collisionInfo = MoveAndCollide(_velocity *
(float)delta);
        if (collisionInfo != null)
            _velocity =
_velocity.Bounce(collisionInfo.GetNormal());
    }
}
```

move_and_slide

Sliding is a common collision response; imagine a player moving along walls in a top-down game or running up and down slopes in a platformer. While it's possible to code this response yourself after using `move_and_collide()`, `move_and_slide()` provides a convenient way to implement sliding movement without writing much code.

Warning

`move_and_slide()` automatically includes the timestep in its calculation, so you should **not** multiply the velocity vector by `delta`.

For example, use the following code to make a character that can walk along the ground (including slopes) and jump when standing on the ground:

GDScript

```
extends CharacterBody2D

var run_speed = 350
var jump_speed = -1000
var gravity = 2500

func get_input():
    velocity.x = 0
    var right = Input.is_action_pressed('ui_right')
    var left = Input.is_action_pressed('ui_left')
    var jump = Input.is_action_just_pressed('ui_select')

    if is_on_floor() and jump:
        velocity.y = jump_speed
    if right:
        velocity.x += run_speed
    if left:
        velocity.x -= run_speed

func _physics_process(delta):
    velocity.y += gravity * delta
    get_input()
    move_and_slide()
```

C#

```csharp
using Godot;

public partial class Body : CharacterBody2D
{
    private float _runSpeed = 350;
    private float _jumpSpeed = -1000;
    private float _gravity = 2500;

    private void GetInput()
    {
        var velocity = Velocity;
        velocity.X = 0;

        var right = Input.IsActionPressed("ui_right");
        var left = Input.IsActionPressed("ui_left");
        var jump = Input.IsActionPressed("ui_select");

        if (IsOnFloor() && jump)
            velocity.Y = _jumpSpeed;
        if (right)
            velocity.X += _runSpeed;
        if (left)
            velocity.X -= _runSpeed;

        Velocity = velocity;
    }

    public override void _PhysicsProcess(double delta)
    {
        var velocity = Velocity;
        velocity.Y += _gravity * (float)delta;
        Velocity = velocity;
        GetInput();
        MoveAndSlide();
    }
}
```

See Kinematic character (2D) for more details on using `move_and_slide()`, including a demo project with detailed code.

Using RigidBody

What is a rigid body?

A rigid body is one that is directly controlled by the physics engine in order to simulate the behavior of physical objects. In order to define the shape of the body, it must have one or more Shape3D objects assigned. Note that setting the position of these shapes will affect the body's center of mass.

How to control a rigid body

A rigid body's behavior can be altered by setting its properties, such as mass and weight. A physics material needs to be added to the rigid body to adjust its friction and bounce, and set if it's absorbent and/or rough. These properties can be set in the Inspector or via code. See RigidBody3D and PhysicsMaterial for the full list of properties and their effects.

There are several ways to control a rigid body's movement, depending on your desired application.

If you only need to place a rigid body once, for example to set its initial location, you can use the methods provided by the Node3D node, such as `set_global_transform()` or `look_at()`. However, these methods cannot be called every frame or the physics engine will not be able to correctly simulate the body's state. As an example, consider a rigid body that you want to rotate so that it points towards another object. A common mistake when implementing this kind of behavior is to use `look_at()` every frame, which breaks the physics simulation. Below, we'll demonstrate how to implement this correctly.

The fact that you can't use `set_global_transform()` or `look_at()` methods doesn't mean that you can't have full control of a rigid body. Instead, you can control it by using the `_integrate_forces()` callback. In this method, you can add *forces*, apply *impulses*, or set the *velocity* in order to achieve any movement you desire.

The "look at" method

As described above, using the Node3D's `look_at()` method can't be used each frame to follow a target. Here is a custom `look_at()` method called `look_follow()` that will work with rigid bodies:

GDScript

```
extends RigidBody3D

var speed: float = 0.1

func look_follow(state: PhysicsDirectBodyState3D,
current_transform: Transform3D, target_position: Vector3) ->
void:
    var forward_local_axis: Vector3 = Vector3(1, 0, 0)
    var forward_dir: Vector3 = (current_transform.basis *
forward_local_axis).normalized()
    var target_dir: Vector3 = (target_position -
current_transform.origin).normalized()
    var local_speed: float = clampf(speed, 0,
acos(forward_dir.dot(target_dir)))
    if forward_dir.dot(target_dir) > 1e-4:
        state.angular_velocity = local_speed *
forward_dir.cross(target_dir) / state.step

func _integrate_forces(state):
    var target_position =
$my_target_node3d_node.global_transform.origin
    look_follow(state, global_transform, target_position)
```

C#

```
using Godot;

public partial class MyRigidBody3D : RigidBody3D
{
    private float _speed = 0.1f;
    private void LookFollow(PhysicsDirectBodyState3D state,
Transform3D currentTransform, Vector3 targetPosition)
    {
        Vector3 forwardLocalAxis = new Vector3(1, 0, 0);
        Vector3 forwardDir = (currentTransform.Basis *
forwardLocalAxis).Normalized();
        Vector3 targetDir = (targetPosition -
currentTransform.Origin).Normalized();
        float localSpeed = Mathf.Clamp(_speed, 0.0f,
Mathf.Acos(forwardDir.Dot(targetDir)));
        if (forwardDir.Dot(targetDir) > 1e-4)
        {
            state.AngularVelocity =
forwardDir.Cross(targetDir) * localSpeed / state.Step;
        }
    }

    public override void
_IntegrateForces(PhysicsDirectBodyState3D state)
    {
        Vector3 targetPosition = GetNode<Node3D>
("MyTargetNode3DNode").GlobalTransform.Origin;
        LookFollow(state, GlobalTransform, targetPosition);
    }
}
```

This method uses the rigid body's `angular_velocity` property to rotate the body. The axis to rotate around is given by the cross product between the current forward direction and the direction one wants to look in. The `clamp` is a simple method used to prevent the amount of rotation from going past the direction which is wanted to be looked in, as the total amount of rotation needed is given by the arccosine of the dot product. This method can be used with `axis_lock_angular_*` as well. If more precise control is needed, solutions such as ones relying on Quaternion may be required, as discussed in Using 3D transforms.

Using Area2D

Introduction

Godot offers a number of 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 Area2D node and show some examples of how it can be used.

Note

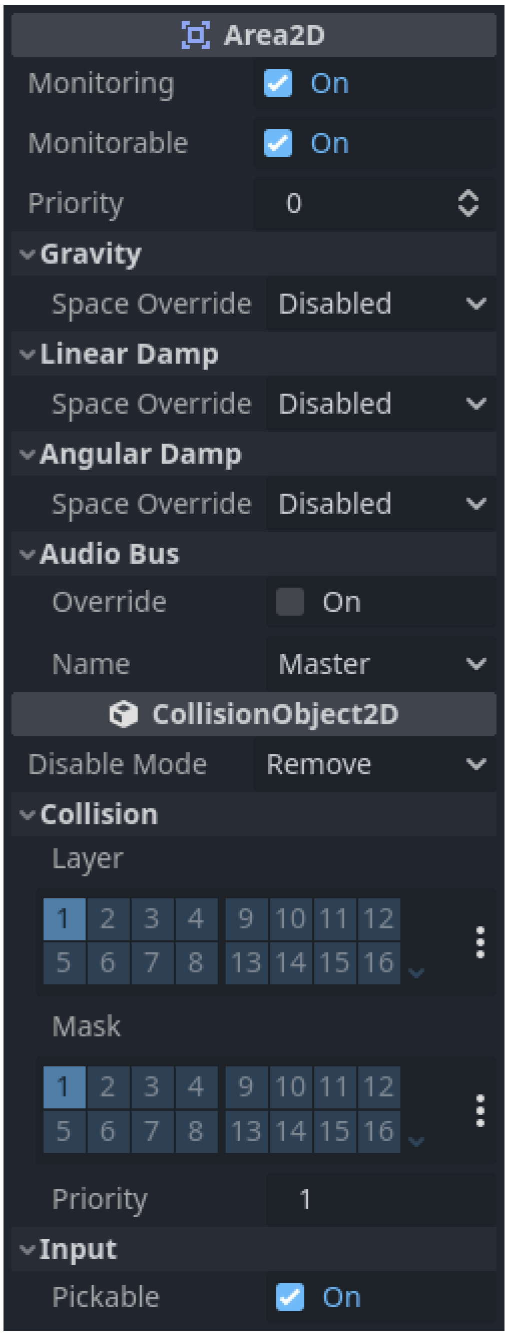
This document assumes you're familiar with Godot's various physics bodies. Please read Physics introduction first.

What is an area?

An Area2D defines a region of 2D space. In this space you can detect other CollisionObject2D nodes overlapping, entering, and exiting. Areas also allow for overriding local physics properties. We'll explore each of these functions below.

Area properties

Areas have many properties you can use to customize their behavior.



Area2D

Monitoring	✓ On
Monitorable	✓ On
Priority	0

Gravity

Space Override	Disabled

Linear Damp

Space Override	Disabled

Angular Damp

Space Override	Disabled

Audio Bus

Override	☐ On
Name	Master

CollisionObject2D

Disable Mode	Remove

Collision

Layer: 1 (selected)

Mask: 1 (selected)

Priority	1

Input

Pickable	✓ On

The `Gravity`, `Linear Damp`, and `Angular Damp` sections are used to configure the area's physics override behavior. We'll look at how to use those in the *Area influence* section below.

`Monitoring` and `Monitorable` are used to enable and disable the area.

The `Audio Bus` section allows you to override audio in the area, for example to apply an audio effect when the player moves through.

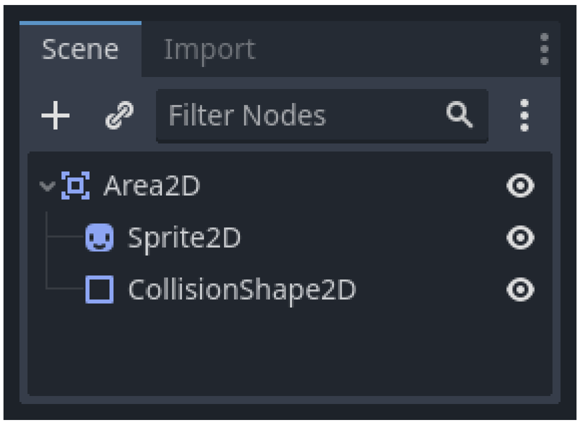
Note that Area2D extends [CollisionObject2D](), so it also provides properties inherited from that class. The `Collision` section of `CollisionObject2D` is where you configure the area's collision layer(s) and mask(s).

Overlap detection

Perhaps the most common use of Area2D nodes is for contact and overlap detection. When you need to know that two objects have touched, but don't need physical collision, you can use an area to notify you of the contact.

For example, let's say we're making a coin for the player to pick up. The coin is not a solid object - the player can't stand on it or push it - we just want it to disappear when the player touches it.

Here's the node setup for the coin:



To detect the overlap, we'll connect the appropriate signal on the Area2D. Which signal to use depends on the player's node type. If the player is another area, use `area_entered`. However, let's assume our player is a `CharacterBody2D` (and therefore a `CollisionObject2D` type), so we'll connect the `body_entered` signal.

Note

If you're not familiar with using signals, see Using signals for an introduction.

GDScript

```
extends Area2D

func _on_coin_body_entered(body):
    queue_free()
```

C#

```
using Godot;

public partial class Coin : Area2D
{
    private void OnCoinBodyEntered(PhysicsBody2D body)
    {
        QueueFree();
    }
}
```

Now our player can collect the coins!

Some other usage examples:

- Areas are great for bullets and other projectiles that hit and deal damage, but don't need any other physics such as bouncing.
- Use a large circular area around an enemy to define its "detect" radius. When the player is outside the area, the enemy can't "see" it.
- "Security cameras" - In a large level with multiple cameras, attach areas to each camera and activate them when the player enters.

See the Your first 2D game for an example of using Area2D in a game.

Area influence

The second major use for area nodes is to alter physics. By default, the area won't do this, but you can enable this with the `Space Override` property. When areas overlap, they are processed in `Priority` order (higher priority areas are processed first). There are four options for override:

- *Combine* - The area adds its values to what has been calculated so far.
- *Replace* - The area replaces physics properties, and lower priority areas are ignored.
- *Combine-Replace* - The area adds its gravity/damping values to whatever has been calculated so far (in priority order), ignoring any lower priority areas.
- *Replace-Combine* - The area replaces any gravity/damping calculated so far, but keeps calculating the rest of the areas.

Using these properties, you can create very complex behavior with multiple overlapping areas.

The physics properties that can be overridden are:

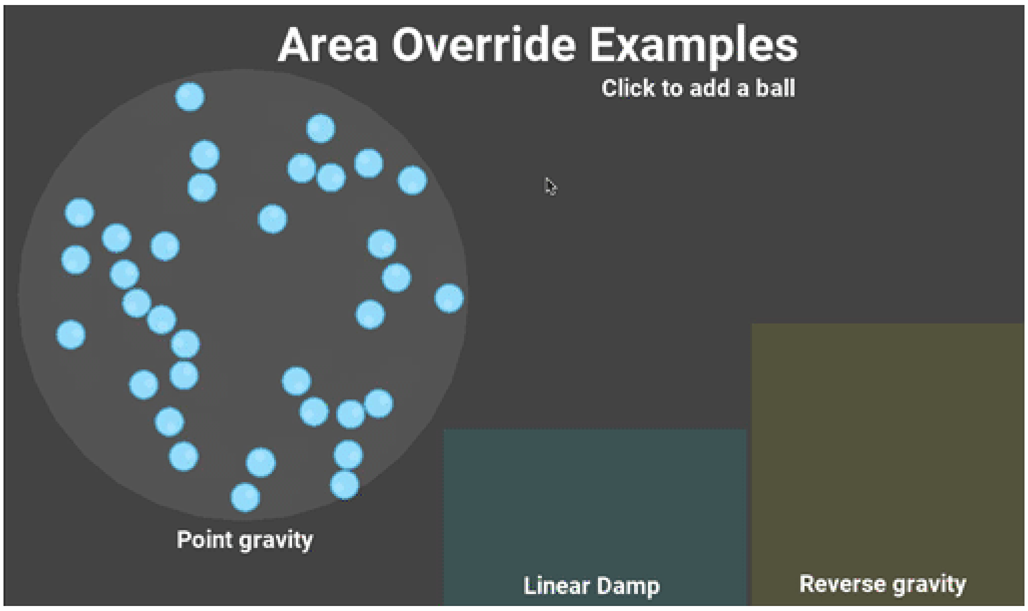
- *Gravity* - Gravity's strength inside the area.
- *Gravity Direction* - This vector does not need to be normalized.
- *Linear Damp* - How quickly objects stop moving - linear velocity lost per second.
- *Angular Damp* - How quickly objects stop spinning - angular velocity lost per second.

Point gravity

The `Gravity Point` property allows you to create an "attractor". Gravity in the area will be calculated towards a point, given by the `Point Center` property. Values are relative to the Area2D, so for example using `(0, 0)` will attract objects to the center of the area.

Examples

The example project attached below has three areas demonstrating physics override.



You can download this project here: [area_2d_starter.zip](https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/area_2d_starter.zip)
[https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/area_2d_starter.zip]

Using CharacterBody2D/3D

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 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?

`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 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

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

This method takes one required parameter: a 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 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

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 docs for full details.

Detecting collisions

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()`:

GDScript

```
# 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)
```

C#

```
// Using MoveAndCollide.
var collision = MoveAndCollide(Velocity * (float)delta);
if (collision != null)
{
    GD.Print("I collided with ",
((Node)collision.GetCollider()).Name);
}

// Using MoveAndSlide.
MoveAndSlide();
for (int i = 0; i < GetSlideCollisionCount(); i++)
{
    var collision = GetSlideCollision(i);
    GD.Print("I collided with ",
((Node)collision.GetCollider()).Name);
}
```

Note

get_slide_collision_count() only counts times the body has collided and changed direction.

See KinematicCollision2D for details on what collision data is returned.

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:

GDScript

```
# 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()
```

C#

```
// using MoveAndCollide
var collision = MoveAndCollide(Velocity * (float)delta);
if (collision != null)
{
    Velocity = Velocity.Slide(collision.GetNormal());
}

// using MoveAndSlide
MoveAndSlide();
```

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

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

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 for examples of implementing 2D movement schemes.

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

GDScript

```
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)
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private int _speed = 300;

    public void GetInput()
    {
        Vector2 inputDir = Input.GetVector("ui_left",
"ui_right", "ui_up", "ui_down");
        Velocity = inputDir * _speed;
    }

    public override void _PhysicsProcess(double delta)
    {
        GetInput();
        MoveAndCollide(Velocity * (float)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 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

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()`:

GDScript

```
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()
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private PackedScene _bullet = GD.Load<PackedScene>
("res://Bullet.tscn");
    private int _speed = 200;

    public void GetInput()
    {
        // Add these actions in Project Settings -> Input
Map.
        float inputDir = Input.GetAxis("backward",
"forward");
        Velocity = Transform.X * inputDir * _speed;
        if (Input.IsActionPressed("shoot"))
        {
            Shoot();
        }
    }

    public void Shoot()
    {
        // "Muzzle" is a Marker2D placed at the barrel of the
gun.
        var b = (Bullet)_bullet.Instantiate();
        b.Start(GetNode<Node2D>("Muzzle").GlobalPosition,
Rotation);
        GetTree().Root.AddChild(b);
    }

    public override void _PhysicsProcess(double delta)
    {
        GetInput();
        var dir = GetGlobalMousePosition() - GlobalPosition;
        // Don't move if too close to the mouse pointer.
        if (dir.Length() > 5)
        {
            Rotation = dir.Angle();
            MoveAndSlide();
        }
    }
}
```

And the code for the Bullet:

GDScript

```
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()
```

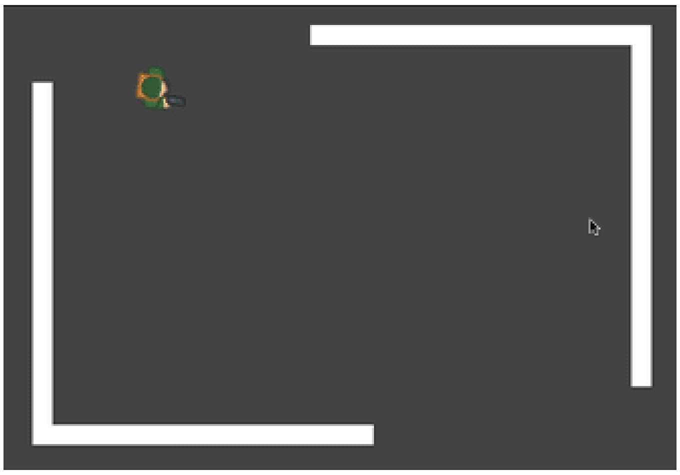
C#

```
using Godot;

public partial class Bullet : CharacterBody2D
{
    public int _speed = 750;

    public void Start(Vector2 position, float direction)
    {
        Rotation = direction;
        Position = position;
        Velocity = new Vector2(speed, 0).Rotated(Rotation);
    }

    public override void _PhysicsProcess(double delta)
    {
        var collision = MoveAndCollide(Velocity *
(float)delta);
        if (collision != null)
        {
            Velocity =
Velocity.Bounce(collision.GetNormal());
            if (collision.GetCollider().HasMethod("Hit"))
            {
                collision.GetCollider().Call("Hit");
            }
        }
    }

    private void OnVisibilityNotifier2DScreenExited()
    {
        // Deletes the bullet when it exits the screen.
        QueueFree();
    }
}
```

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

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]() to create the platform shapes.

Here's the code for the player body:

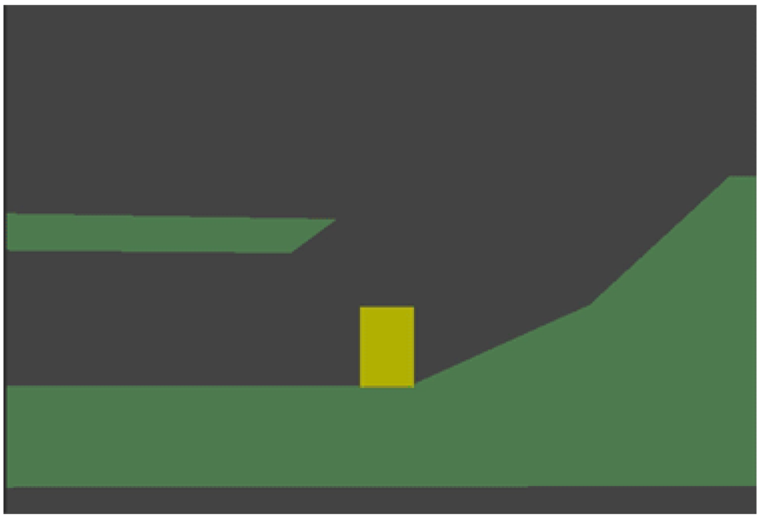
GDScript

```
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()
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private float _speed = 100.0f;
    private float _jumpSpeed = -400.0f;

    // Get the gravity from the project settings so you can
sync with rigid body nodes.
    public float Gravity =
ProjectSettings.GetSetting("physics/2d/default_gravity").AsSi
ngle();

    public override void _PhysicsProcess(double delta)
    {
        Vector2 velocity = Velocity;

        // Add the gravity.
        velocity.Y += Gravity * (float)delta;

        // Handle jump.
        if (Input.IsActionJustPressed("jump") && IsOnFloor())
            velocity.Y = _jumpSpeed;
        // Get the input direction.
        float direction = Input.GetAxis("ui_left",
"ui_right");
        velocity.X = direction * _speed;

        Velocity = velocity;
        MoveAndSlide();
    }
}
```



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.

Ray-casting

Introduction

One of the most common tasks in game development is casting a ray (or custom shaped object) and checking what it hits. This enables complex behaviors, AI, etc. to take place. This tutorial will explain how to do this in 2D and 3D.

Godot stores all the low level game information in servers, while the scene is only a frontend. As such, ray casting is generally a lower-level task. For simple raycasts, nodes like RayCast3D and RayCast2D will work, as they return every frame what the result of a raycast is.

Many times, though, ray-casting needs to be a more interactive process so a way to do this by code must exist.

Space

In the physics world, Godot stores all the low level collision and physics information in a *space*. The current 2d space (for 2D Physics) can be obtained by accessing CanvasItem.get_world_2d().space. For 3D, it's Node3D.get_world_3d().space.

The resulting space RID can be used in PhysicsServer3D and PhysicsServer2D respectively for 3D and 2D.

Accessing space

Godot physics runs by default in the same thread as game logic, but may be set to run on a separate thread to work more efficiently. Due to this, the only time accessing space is safe is during the

Node._physics_process() callback. Accessing it from outside this function may result in an error due to space being *locked*.

To perform queries into physics space, the PhysicsDirectSpaceState2D and PhysicsDirectSpaceState3D must be used.

Use the following code in 2D:

GDScript

```
func _physics_process(delta):
    var space_rid = get_world_2d().space
    var space_state =
PhysicsServer2D.space_get_direct_state(space_rid)
```

C#

```
public override void _PhysicsProcess(double delta)
{
    var spaceRid = GetWorld2D().Space;
    var spaceState =
Physics2DServer.SpaceGetDirectState(spaceRid);
}
```

Or more directly:

GDScript

```
func _physics_process(delta):
    var space_state = get_world_2d().direct_space_state
```

C#

```
public override void _PhysicsProcess(double delta)
{
    var spaceState = GetWorld2D().DirectSpaceState;
}
```

And in 3D:

GDScript

```
func _physics_process(delta):
    var space_state = get_world_3d().direct_space_state
```

C#

```
public override void _PhysicsProcess(double delta)
{
    var spaceState = GetWorld3D().DirectSpaceState;
}
```

Raycast query

For performing a 2D raycast query, the method PhysicsDirectSpaceState2D.intersect_ray() may be used. For example:

GDScript

```
func _physics_process(delta):
    var space_state = get_world_2d().direct_space_state
    # use global coordinates, not local to node
    var query = PhysicsRayQueryParameters2D.create(Vector2(0,
0), Vector2(50, 100))
    var result = space_state.intersect_ray(query)
```

C#

```
public override void _PhysicsProcess(double delta)
{
    var spaceState = GetWorld2D().DirectSpaceState;
    // use global coordinates, not local to node
    var query =
PhysicsRayQueryParameters2D.Create(Vector2.Zero, new
Vector2(50, 100));
    var result = spaceState.IntersectRay(query);
}
```

The result is a dictionary. If the ray didn't hit anything, the dictionary will be empty. If it did hit something, it will contain collision information:

GDScript

```
if result:
    print("Hit at point: ", result.position)
```

C#

```
if (result.Count > 0)
    GD.Print("Hit at point: ", result["position"]);
```

The `result` dictionary when a collision occurs contains the following data:

```
{
   position: Vector2 # point in world space for collision
   normal: Vector2 # normal in world space for collision
   collider: Object # Object collided or null (if
unassociated)
   collider_id: ObjectID # Object it collided against
   rid: RID # RID it collided against
   shape: int # shape index of collider
   metadata: Variant() # metadata of collider
}
```

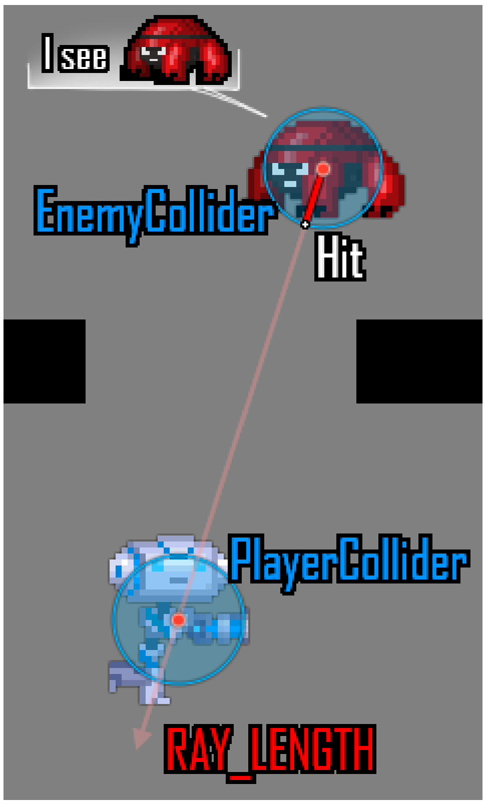
The data is similar in 3D space, using Vector3 coordinates. Note that to enable collisions with Area3D, the boolean parameter `collide_with_areas` must be set to `true`.

GDScript

```
const RAY_LENGTH = 1000

func _physics_process(delta):
    var space_state = get_world_3d().direct_space_state
    var cam = $Camera3D
    var mousepos = get_viewport().get_mouse_position()

    var origin = cam.project_ray_origin(mousepos)
    var end = origin + cam.project_ray_normal(mousepos) *
RAY_LENGTH
    var query = PhysicsRayQueryParameters3D.create(origin,
end)
    query.collide_with_areas = true

    var result = space_state.intersect_ray(query)
```

Collision exceptions

A common use case for ray casting is to enable a character to gather data about the world around it. One problem with this is that the same character has a collider, so the ray will only detect its parent's collider, as shown in the following image:



To avoid self-intersection, the `intersect_ray()` parameters object can take an array of exceptions via its `exclude` property. This is an example of how to use it from a CharacterBody2D or any other collision object node:

GDScript

```
extends CharacterBody2D

func _physics_process(delta):
    var space_state = get_world_2d().direct_space_state
    var query =
PhysicsRayQueryParameters2D.create(global_position,
player_position)
```

```
    query.exclude = [self]
    var result = space_state.intersect_ray(query)
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    public override void _PhysicsProcess(double delta)
    {
        var spaceState = GetWorld2D().DirectSpaceState;
        var query = 
PhysicsRayQueryParameters2D.Create(globalPosition, 
playerPosition);
        query.Exclude = new Godot.Collections.Array<Rid> { 
GetRid() };
        var result = spaceState.IntersectRay(query);
    }
}
```

The exceptions array can contain objects or RIDs.

Collision Mask

While the exceptions method works fine for excluding the parent body, it becomes very inconvenient if you need a large and/or dynamic list of exceptions. In this case, it is much more efficient to use the collision layer/mask system.

The `intersect_ray()` parameters object can also be supplied a collision mask. For example, to use the same mask as the parent body, use the `collision_mask` member variable. The array of exceptions can be supplied as the last argument as well:

GDScript

```
extends CharacterBody2D

func _physics_process(delta):
    var space_state = get_world_2d().direct_space_state
    var query = 
PhysicsRayQueryParameters2D.create(global_position,
target_position,
        collision_mask, [self])
    var result = space_state.intersect_ray(query)
```

C#

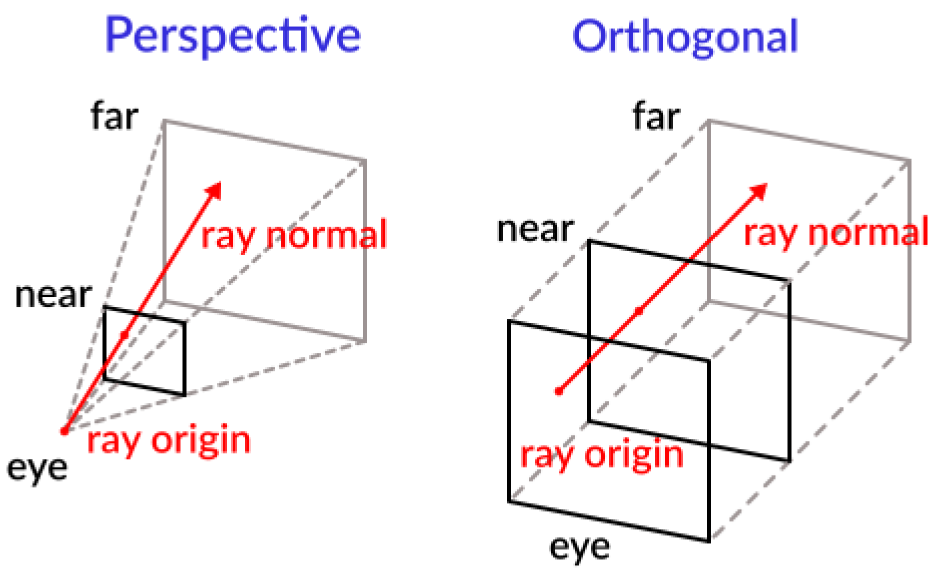
```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    public override void _PhysicsProcess(double delta)
    {
        var spaceState = GetWorld2D().DirectSpaceState;
        var query =
PhysicsRayQueryParameters2D.Create(globalPosition,
targetPosition,
            CollisionMask, new Godot.Collections.Array<Rid>
{ GetRid() });
        var result = spaceState.IntersectRay(query);
    }
}
```

See Code example for details on how to set the collision mask.

3D ray casting from screen

Casting a ray from screen to 3D physics space is useful for object picking. There is not much need to do this because CollisionObject3D has an "input_event" signal that will let you know when it was clicked, but in case there is any desire to do it manually, here's how.

To cast a ray from the screen, you need a Camera3D node. A `Camera3D` can be in two projection modes: perspective and orthogonal. Because of this, both the ray origin and direction must be obtained. This is because `origin` changes in orthogonal mode, while `normal` changes in perspective mode:



To obtain it using a camera, the following code can be used:

GDScript

```
const RAY_LENGTH = 1000.0

func _input(event):
    if event is InputEventMouseButton and event.pressed and
event.button_index == 1:
          var camera3d = $Camera3D
          var from =
camera3d.project_ray_origin(event.position)
          var to = from +
camera3d.project_ray_normal(event.position) * RAY_LENGTH
```

C#

```
private const float RayLength = 1000.0f;

public override void _Input(InputEvent @event)
{
    if (@event is InputEventMouseButton eventMouseButton &&
eventMouseButton.Pressed && eventMouseButton.ButtonIndex ==
MouseButton.Left)
    {
        var camera3D = GetNode<Camera3D>("Camera3D");
        var from =
camera3D.ProjectRayOrigin(eventMouseButton.Position);
        var to = from +
camera3D.ProjectRayNormal(eventMouseButton.Position) *
RayLength;
    }
}
```

Remember that during `_input()`, the space may be locked, so in practice this query should be run in `_physics_process()`.

Ragdoll system

Introduction

Since version 3.1, Godot supports ragdoll physics. Ragdolls rely on physics simulation to create realistic procedural animation. They are used for death animations in many games.

In this tutorial, we will be using the Platformer3D demo to set up a ragdoll.

Note

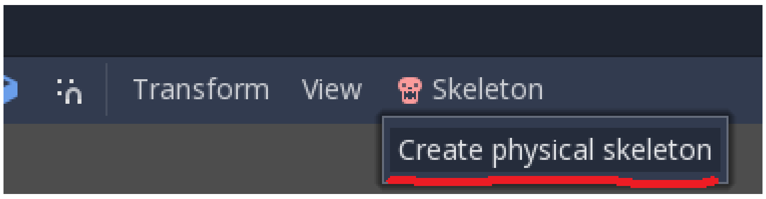
You can download the Platformer3D demo on GitHub [https://github.com/godotengine/godot-demo-projects/tree/master/3d/platformer] or using the Asset Library [https://godotengine.org/asset-library/asset/125].

Setting up the ragdoll

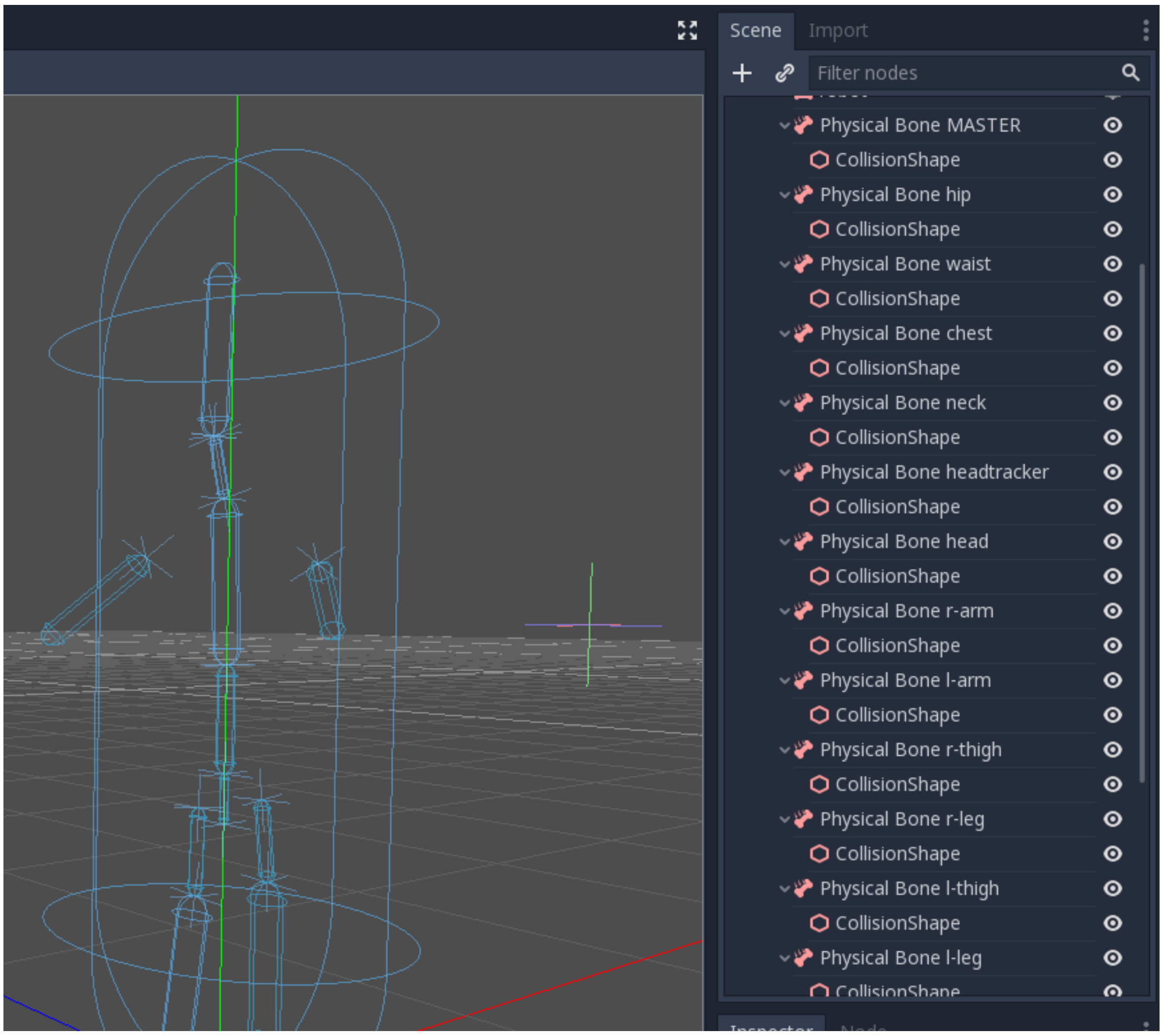
Creating physical bones

Like many other features in the engine, there is a node to set up a ragdoll: the PhysicalBone3D node. To simplify the setup, you can generate `PhysicalBone` nodes with the "Create physical skeleton" feature in the skeleton node.

Open the platformer demo in Godot, and then the Robi scene. Select the `Skeleton` node. A skeleton button appears on the top bar menu:



Click it and select the `Create physical skeleton` option. Godot will generate `PhysicalBone` nodes and collision shapes for each bone in the skeleton and pin joints to connect them together:



Some of the generated bones aren't necessary: the `MASTER` bone for example. So we're going to clean up the skeleton by removing them.

Cleaning up the skeleton

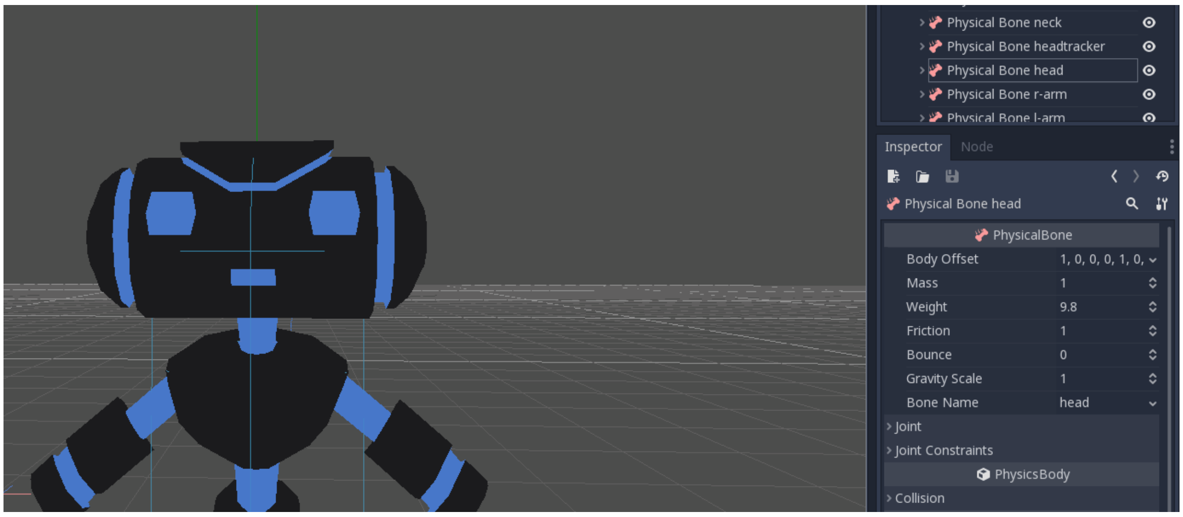
Each `PhysicalBone` the engine needs to simulate has a performance cost, so you want to remove every bone that is too small to make a difference in the simulation, as well as all utility bones.

For example, if we take a humanoid, you do not want to have physical bones for each finger. You can use a single bone for the entire hand instead, or one for the palm, one for the thumb, and a last one for the other four fingers.

Remove these physical bones: `MASTER`, `waist`, `neck`, `headtracker`. This gives us an optimized skeleton and makes it easier to control the ragdoll.

Collision shape adjustment

The next task is adjusting the collision shape and the size of physical bones to match the part of the body that each bone should simulate.

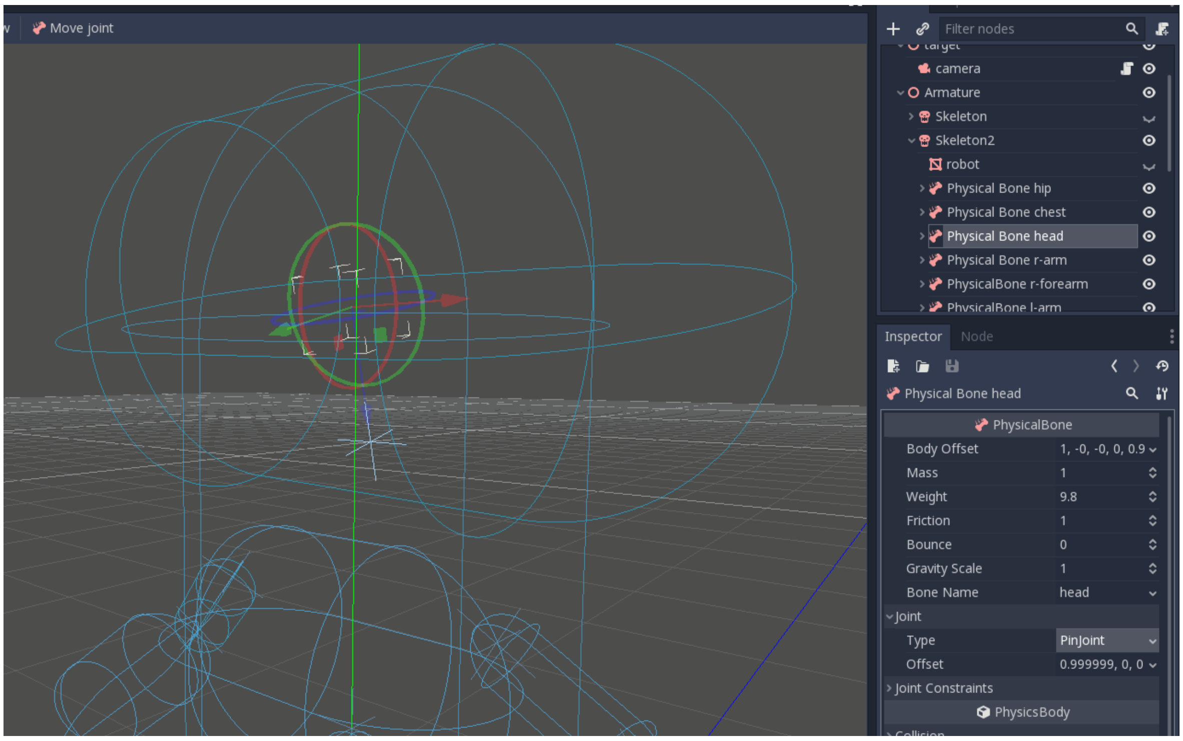


Body Offset	1, 0, 0, 0, 1, 0,
Mass	1
Weight	9.8
Friction	1
Bounce	0
Gravity Scale	1
Bone Name	head

Joints adjustment

Once you adjusted the collision shapes, your ragdoll is almost ready. You just want to adjust the pin joints to get a better simulation. `PhysicalBone` nodes have an unconstrained pin joint assigned to them by default. To change the pin joint, select the

`PhysicalBone` and change the constraint type in the `Joint` section. There, you can change the constraint's orientation and its limits.



PhysicalBone

Body Offset	1, -0, -0, 0, 0.9
Mass	1
Weight	9.8
Friction	1
Bounce	0
Gravity Scale	1
Bone Name	head

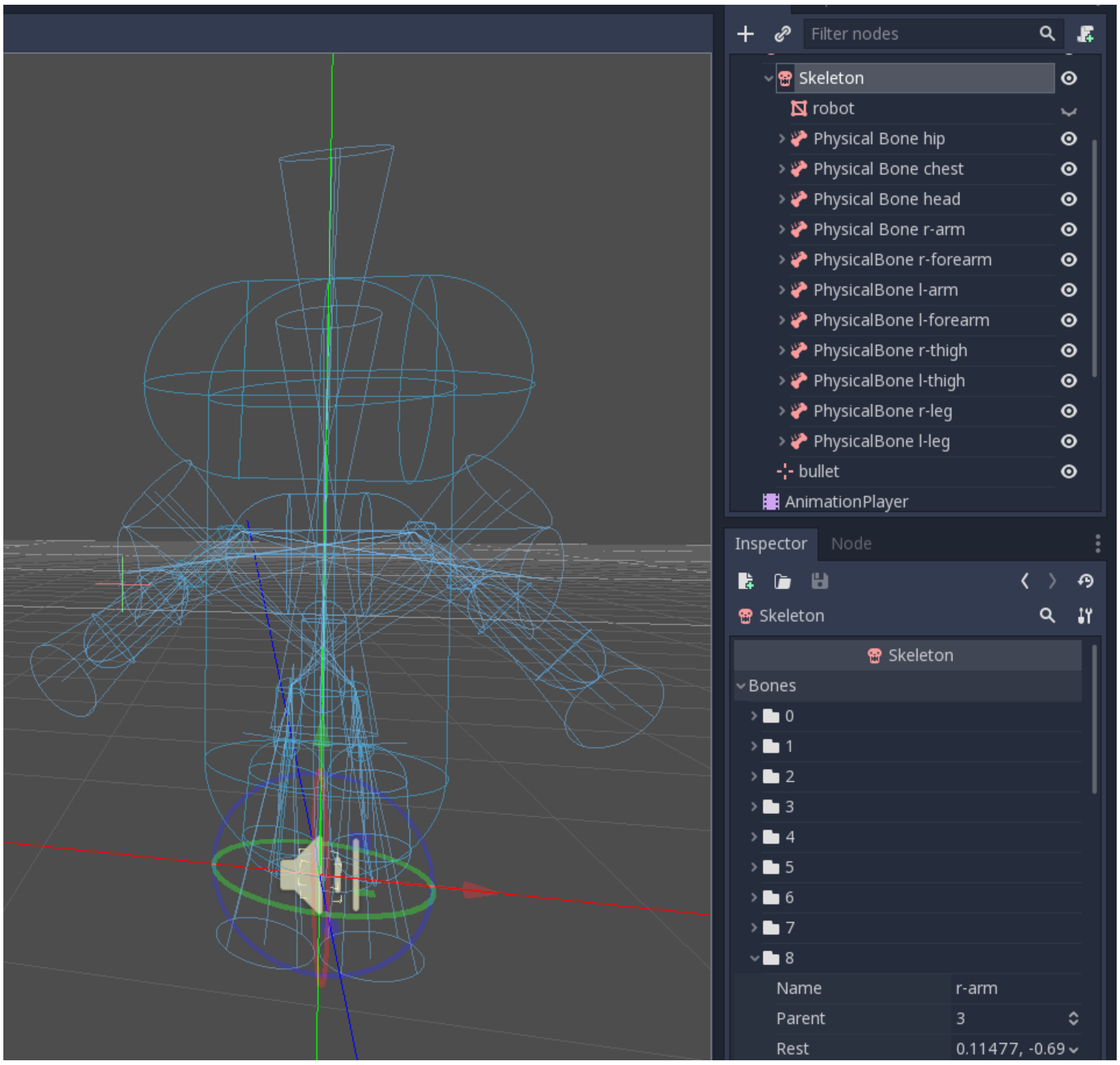
Joint

Type	PinJoint
Offset	0.999999, 0, 0

Joint Constraints

PhysicsBody

This is the final result:



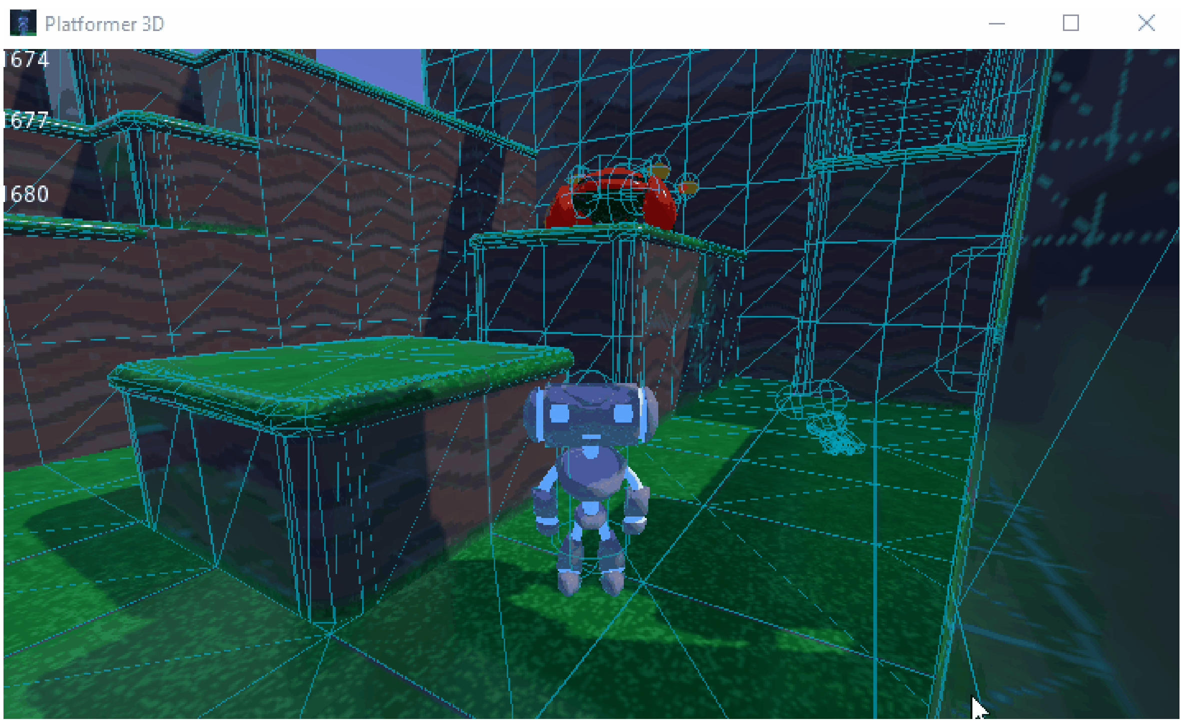
Simulating the ragdoll

The ragdoll is now ready to use. To start the simulation and play the ragdoll animation, you need to call the `physical_bones_start_simulation` method. Attach a script to the skeleton node and call the method in the `_ready` method:

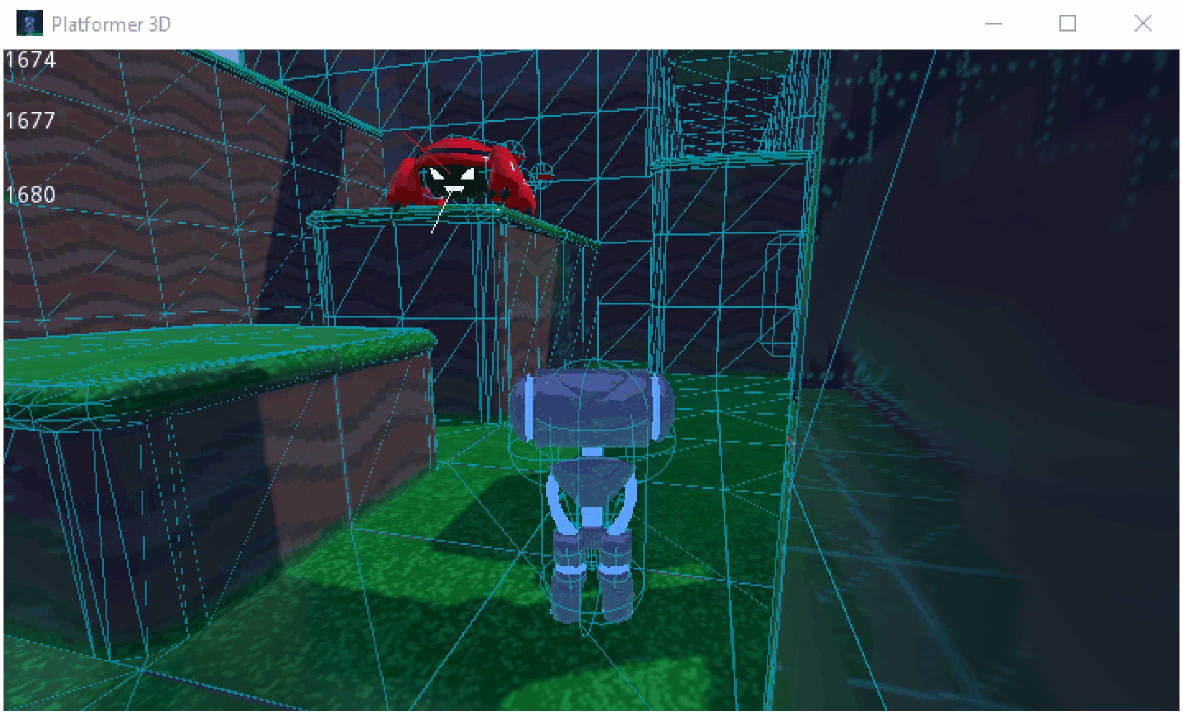
GDScript

```
func _ready():
    physical_bones_start_simulation()
```

To stop the simulation, call the `physical_bones_stop_simulation()` method.

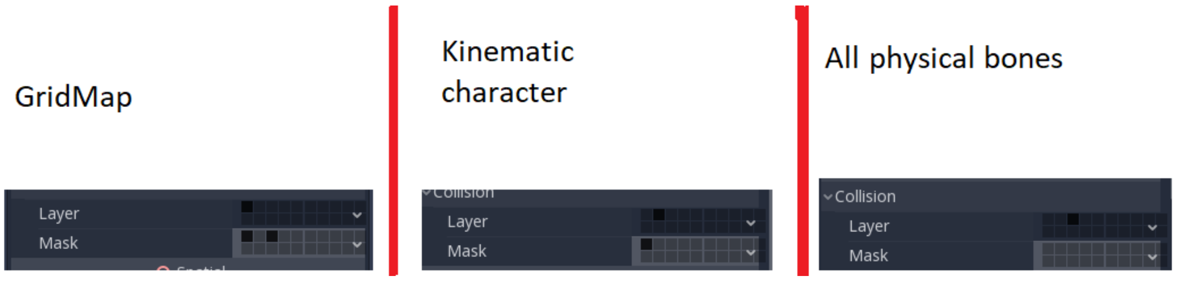


You can also limit the simulation to only a few bones. To do so, pass the bone names as a parameter. Here's an example of partial ragdoll simulation:



Collision layer and mask

Make sure to set up your collision layers and masks properly so the `CharacterBody3D`'s capsule doesn't get in the way of the physics simulation:



For more information, read Collision layers and masks.

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.

Physics process

To manage the logic of a kinematic body or character, it is always advised to use physics process, because it's called before physics step and its execution is in sync with physics server, also it is called the same amount of times per second, always. This makes physics and motion calculation work in a more predictable way than using regular process, which might have spikes or lose precision if the frame rate is too high or too low.

GDScript

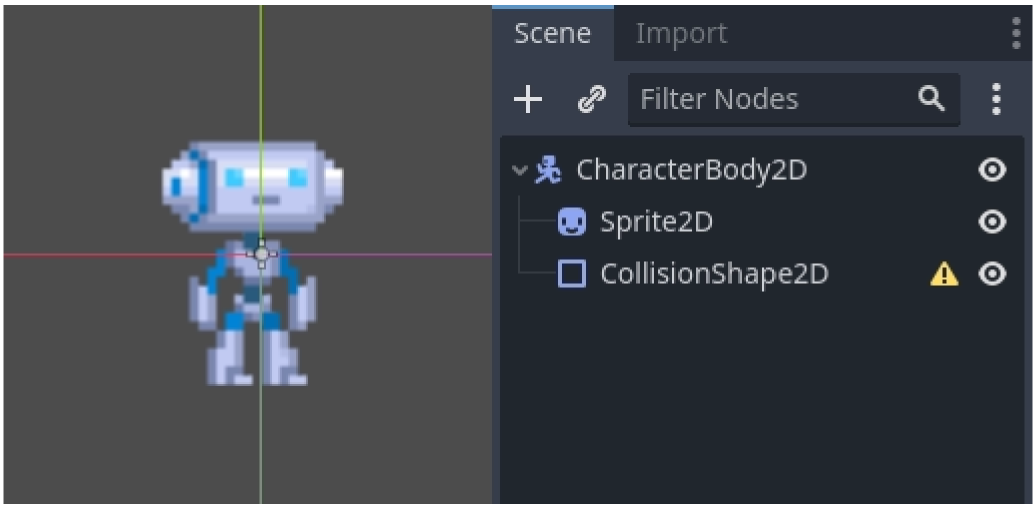
```
extends CharacterBody2D

func _physics_process(delta):
    pass
```

C#

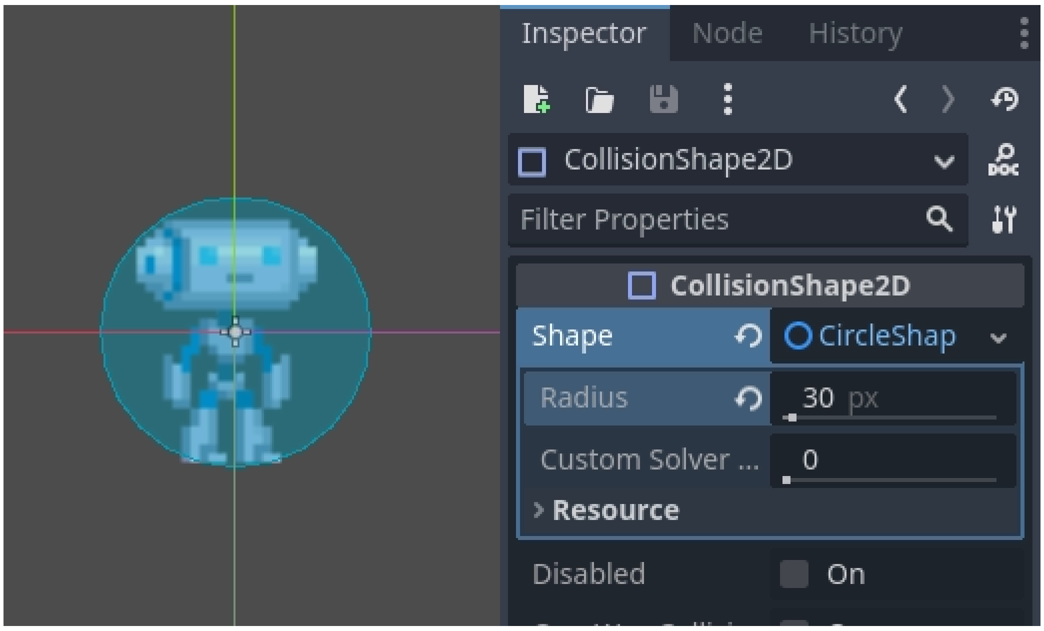
```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    public override void _PhysicsProcess(double delta)
    {
    }
}
```

Scene setup

To have something to test, here's the scene (from the tilemap tutorial): kinematic_character_2d_starter.zip [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/kinematic_character_2d_starter.zip]. We'll be creating a new scene for the character. Use the robot sprite and create a scene like this:



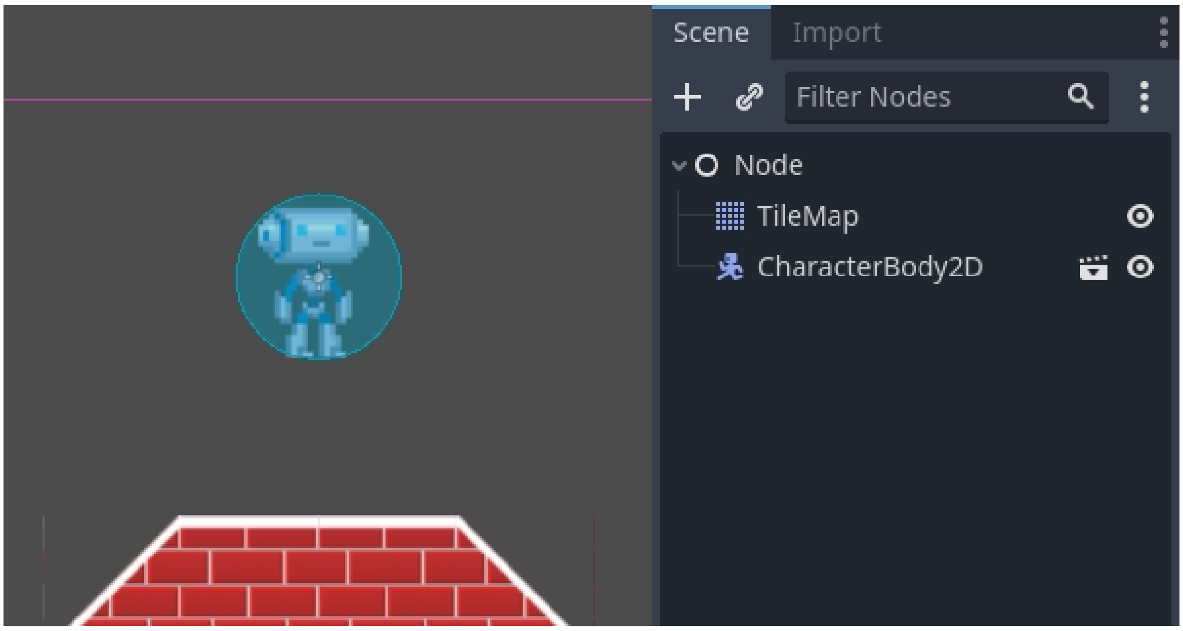
You'll notice that there's a warning icon next to our CollisionShape2D node; that's because we haven't defined a shape for it. Create a new CircleShape2D in the shape property of CollisionShape2D. Click on <CircleShape2D> to go to the options for it, and set the radius to 30:



Note: As mentioned before in the physics tutorial, the physics engine can't handle scale on most types of shapes (only collision polygons, planes and segments work), so always change the parameters (such as radius) of the shape instead of scaling it. The same is also true for the kinematic/rigid/static bodies themselves, as their scale affects the shape scale.

Now, create a script for the character, the one used as an example above should work as a base.

Finally, instance that character scene in the tilemap, and make the map scene the main one, so it runs when pressing play.



Moving the kinematic character

Go back to the character scene, and open the script, the magic begins now! Kinematic body will do nothing by default, but it has a useful function called `CharacterBody2D.move_and_collide()`. This function takes a Vector2 as an argument, and tries to apply that motion to the kinematic body. If a collision happens, it stops right at the moment of the collision.

So, let's move our sprite downwards until it hits the floor:

GDScript

```
extends CharacterBody2D

func _physics_process(delta):
    move_and_collide(Vector2(0, 1)) # Move down 1 pixel per
physics frame
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    public override void _PhysicsProcess(double delta)
    {
        // Move down 1 pixel per physics frame
        MoveAndCollide(new Vector2(0, 1));
    }
}
```

The result is that the character will move, but stop right when hitting the floor. Pretty cool, huh?

The next step will be adding gravity to the mix, this way it behaves a little more like a regular game character:

GDScript

```
extends CharacterBody2D

const GRAVITY = 200.0

func _physics_process(delta):
    velocity.y += delta * GRAVITY

    var motion = velocity * delta
    move_and_collide(motion)
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
```

```
{
    private const float Gravity = 200.0f;

    public override void _PhysicsProcess(double delta)
    {
        var velocity = Velocity;
        velocity.Y += (float)delta * Gravity;
        Velocity = velocity;

        var motion = velocity * (float)delta;
        MoveAndCollide(motion);
    }
}
```

Now the character falls smoothly. Let's make it walk to the sides, left and right when touching the directional keys. Remember that the values being used (for speed at least) are pixels/second.

This adds basic support for walking when pressing left and right:

GDScript

```
extends CharacterBody2D

const GRAVITY = 200.0
const WALK_SPEED = 200

func _physics_process(delta):
    velocity.y += delta * GRAVITY

    if Input.is_action_pressed("ui_left"):
        velocity.x = -WALK_SPEED
    elif Input.is_action_pressed("ui_right"):
        velocity.x =  WALK_SPEED
    else:
        velocity.x = 0

    # "move_and_slide" already takes delta time into account.
    move_and_slide()
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
```

```
{
    private const float Gravity = 200.0f;
    private const int WalkSpeed = 200;

    public override void _PhysicsProcess(double delta)
    {
        var velocity = Velocity;

        velocity.Y += (float)delta * Gravity;

        if (Input.IsActionPressed("ui_left"))
        {
            velocity.X = -WalkSpeed;
        }
        else if (Input.IsActionPressed("ui_right"))
        {
            velocity.X = WalkSpeed;
        }
        else
        {
            velocity.X = 0;
        }

        Velocity = velocity;

        // "MoveAndSlide" already takes delta time into
account.
        MoveAndSlide();
    }
}
```

And give it a try.

This is a good starting point for a platformer. A more complete demo can be found in the demo zip distributed with the engine, or in the https://github.com/godotengine/godot-demo-projects/tree/master/2d/kinematic_character.

Using SoftBody

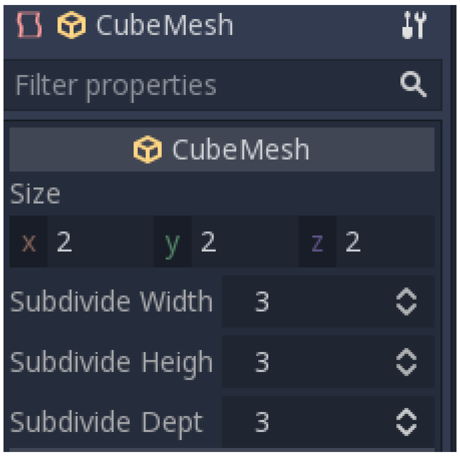
Soft bodies (or *soft-body dynamics*) simulate movement, changing shape and other physical properties of deformable objects. This can for example be used to simulate cloth or to create more realistic characters.

Basic set-up

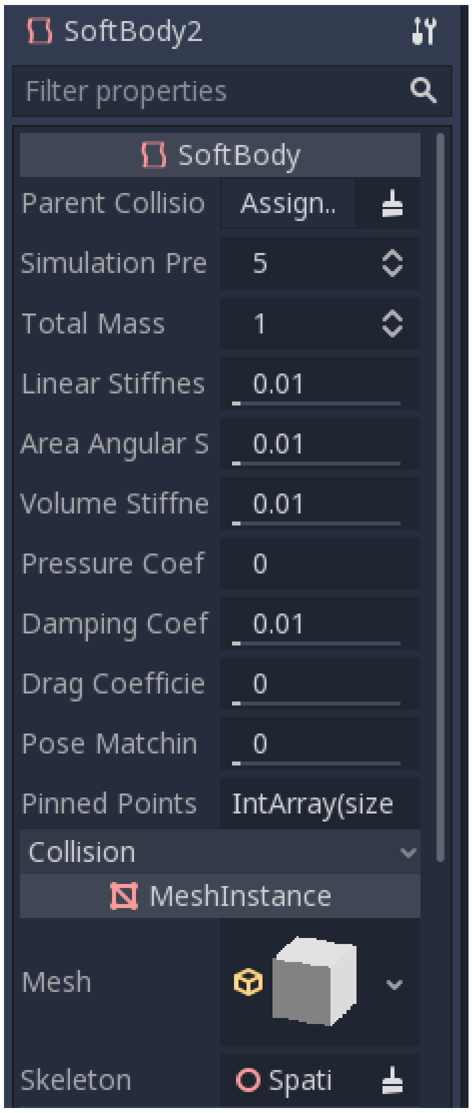
A SoftBody3D node is used for soft body simulations.

We will create a bouncy cube to demonstrate the setup of a soft body.

Create a new scene with a `Node3D` node as root. Then, create a `Softbody` node. Add a `CubeMesh` in the `mesh` property of the node in the inspector and increase the subdivision of the mesh for simulation.


CubeMesh
Filter properties
CubeMesh
Size
x 2 y 2 z 2
Subdivide Width 3
Subdivide Heigh 3
Subdivide Dept 3

Set the parameters to obtain the type of soft body you aim for. Try to keep the `Simulation Precision` above 5, otherwise, the soft body may collapse.



SoftBody2

Filter properties

SoftBody

Parent Collisio Assign..

Simulation Pre 5

Total Mass 1

Linear Stiffnes 0.01

Area Angular S 0.01

Volume Stiffne 0.01

Pressure Coef 0

Damping Coef 0.01

Drag Coefficie 0

Pose Matchin 0

Pinned Points IntArray(size

Collision

MeshInstance

Mesh

Skeleton Spati

Note

Handle some parameters with care, as some value can lead to strange results. For example, if the shape is not completely closed and you set pressure to more than 0, the softbody will fly around like a plastic bag under strong wind.

Play the scene to view the simulation.

Tip

To improve the simulation's result, increase the `Simulation Precision`, this will give significant improvement at the cost of performance.

Cloak simulation

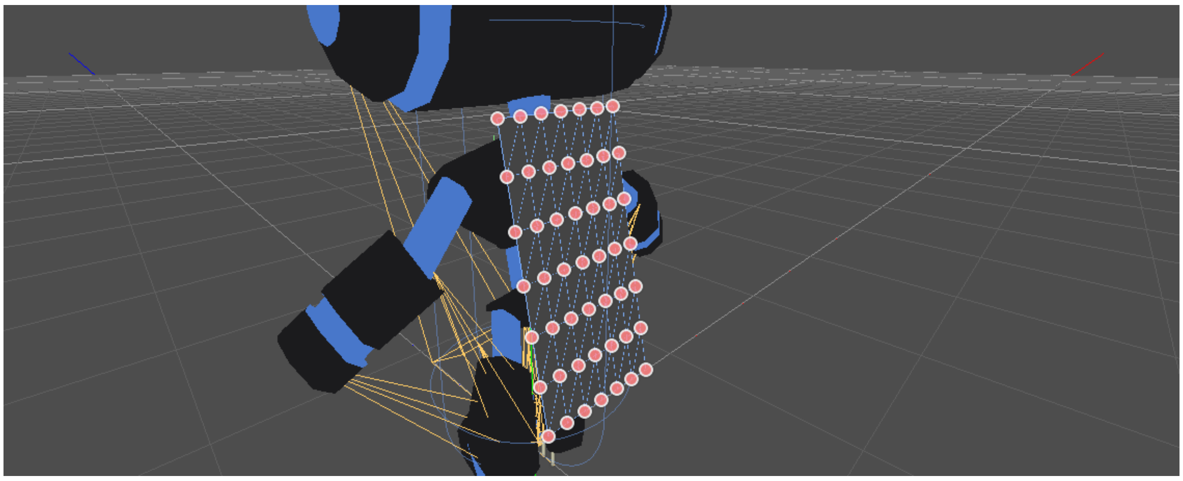
Let's make a cloak in the Platformer3D demo.

Note

You can download the Platformer3D demo on GitHub [https://github.com/godotengine/godot-demo-projects/tree/master/3d/platformer] or the Asset Library [https://godotengine.org/asset-library/asset/125].

Open the `Player` scene, add a `SoftBody` node and assign a `PlaneMesh` to it.

Open the `PlaneMesh` properties and set the size(x: 0.5 y: 1) then set `Subdivide Width` and `Subdivide Depth` to 5. Adjust the `SoftBody`'s position. You should end up with something like this:



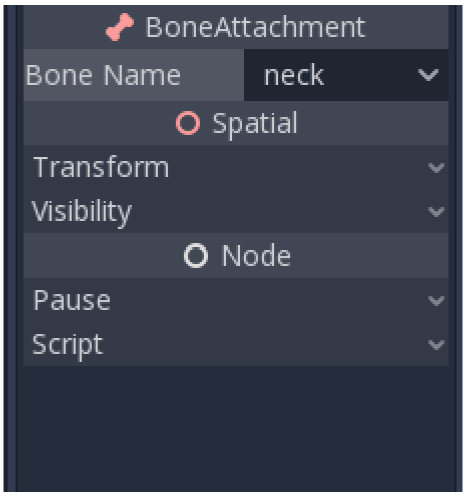
Tip

Subdivision generates a more tessellated mesh for better simulations.

Add a BoneAttachment3D node under the skeleton node and select the Neck bone to attach the cloak to the character skeleton.

Note

`BoneAttachment3D` node is to attach objects to a bone of a armature. The attached object will follow the bone's movement, weapon of a character can be attached this way.



BoneAttachment

Bone Name: neck

Spatial

Transform

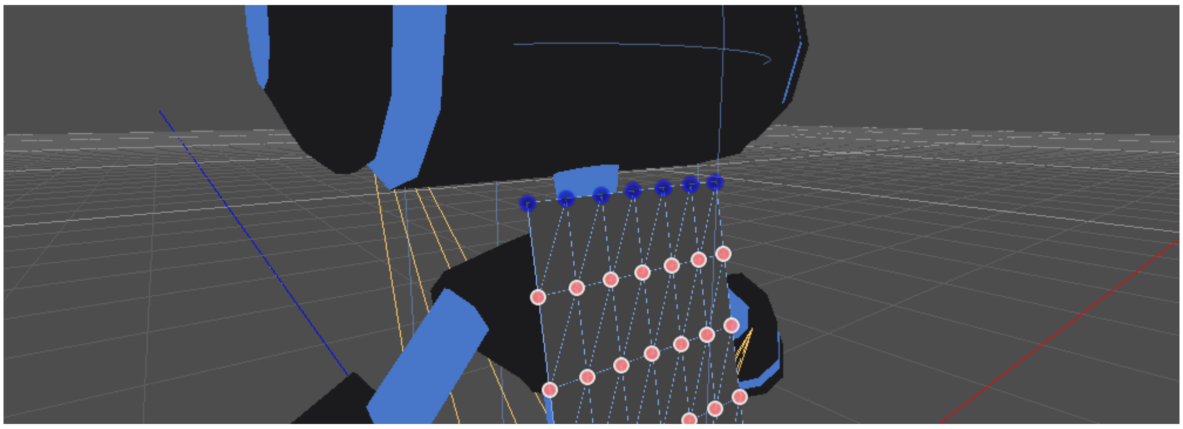
Visibility

Node

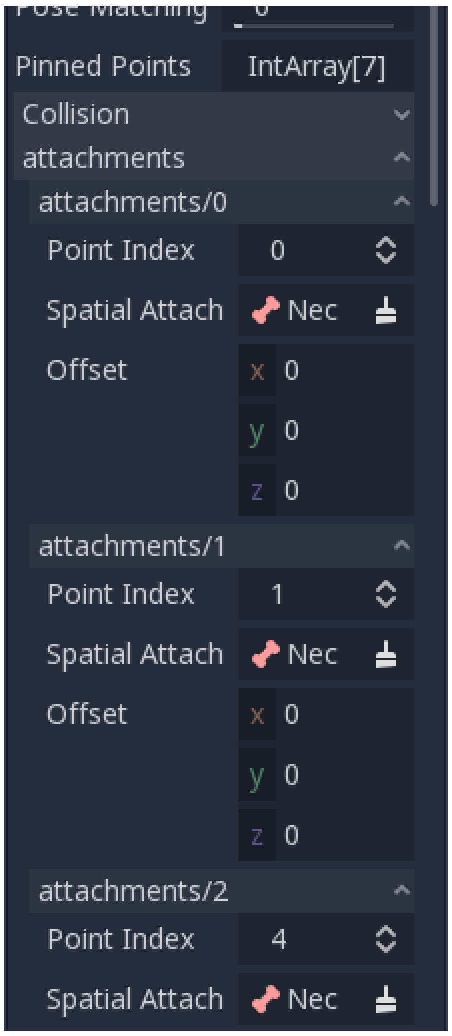
Pause

Script

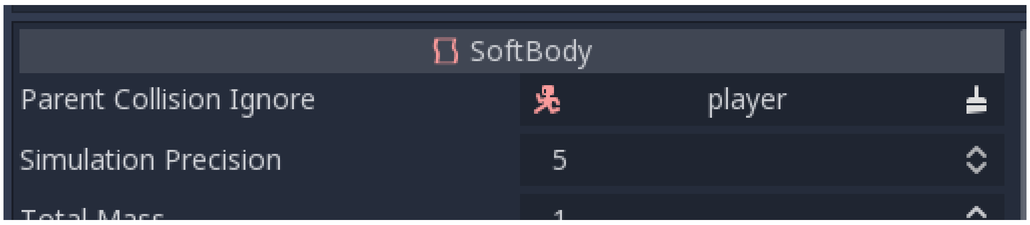
To create pinned joints, select the upper vertices in the `SoftBody` node:



The pinned joints can be found in `SoftBody`'s `Attachments` property, choose the `BoneAttachment` as the `SpatialAttachment` for each pinned joints, the pinned joints are now attached to the neck.


Pinned Points IntArray[7]
Collision
attachments
attachments/0
Point Index 0
Spatial Attach Nec
Offset x 0 y 0 z 0
attachments/1
Point Index 1
Spatial Attach Nec
Offset x 0 y 0 z 0
attachments/2
Point Index 4
Spatial Attach Nec

Last step is to avoid clipping by adding the Kinematic Body *Player* to `Parent Collision Ignore` of the `SoftBody`.


SoftBody
Parent Collision Ignore player
Simulation Precision 5

Play the scene and the cloak should simulate correctly.

This covers the basic settings of softbody, experiment with the parameters to achieve the effect you are aiming for when making your game.

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 child nodes. Note that you must add a Shape2D *resource* to collision shape nodes in the Inspector dock.

Note

When you add multiple collision shapes to a single PhysicsBody2D, you don't have to worry about them overlapping. They won't "collide" with each other.

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.

Convex collision shapes

Warning

Godot currently doesn't offer a built-in way to create 2D convex collision shapes. This section is mainly here for reference purposes.

Convex collision shapes are a compromise between primitive collision shapes and concave collision shapes. They can represent shapes of any complexity, but with an important caveat. As their name implies, an individual shape can only represent a *convex* shape. For instance, a pyramid is *convex*, but a hollow box is *concave*. To define a concave object with a single collision shape, you need to use a concave collision shape.

Depending on the object's complexity, you may get better performance by using multiple convex shapes instead of a concave collision shape. Godot lets you use *convex decomposition* to generate convex shapes that roughly match a hollow object. Note this performance advantage no longer applies after a certain amount of convex shapes. For large and complex objects such as a whole level, we recommend using concave shapes instead.

Concave or trimesh collision shapes

Concave collision shapes, also called trimesh collision shapes, can take any form, from a few triangles to thousands of triangles.

Concave shapes are the slowest option but are also the most accurate in Godot. **You can only use concave shapes within StaticBodies.** They will not work with CharacterBodies or RigidBodies unless the RigidBody's mode is Static.

Note

Even though concave shapes offer the most accurate *collision*, contact reporting can be less precise than primitive shapes.

When not using TileMaps for level design, concave shapes are the best approach for a level's collision.

You can configure the CollisionPolygon2D node's *build mode* in the inspector. If it is set to **Solids** (the default), collisions will include the polygon and its contained area. If it is set to **Segments**, collisions will only include the polygon edges.

You can generate a concave collision shape from the editor by selecting a Sprite2D and using the **Sprite2D** menu at the top of the 2D viewport. The Sprite2D menu dropdown exposes an option called **Create CollisionPolygon2D Sibling**. Once you click it, it displays a menu with 3 settings:

- **Simplification:** Higher values will result in a less detailed shape, which improves performance at the cost of accuracy.
- **Shrink (Pixels):** Higher values will shrink the generated collision polygon relative to the sprite's edges.
- **Grow (Pixels):** Higher values will grow the generated collision polygon relative to the sprite's edges. Note that setting Grow and Shrink to equal values may yield different results than leaving both of them on 0.

Note

If you have an image with many small details, it's recommended to create a simplified version and use it to generate the collision polygon. This can result in better performance and game feel, since the player won't be blocked by small, decorative details.

To use a separate image for collision polygon generation, create another Sprite2D, generate a collision polygon sibling from it then remove the Sprite2D node. This way, you can exclude small details from the generated collision.

Performance caveats

You aren't limited to a single collision shape per PhysicsBody. Still, we recommend keeping the number of shapes as low as possible to improve performance, especially for dynamic objects like RigidBodies and CharacterBodies. On top of that, avoid translating, rotating, or scaling CollisionShapes to benefit from the physics engine's internal optimizations.

When using a single non-transformed collision shape in a StaticBody, the engine's *broad phase* algorithm can discard inactive PhysicsBodies. The *narrow phase* will then only have to take into account the active bodies' shapes. If a StaticBody has many collision shapes, the broad phase will fail. The narrow phase, which is slower, must then perform a collision check against each shape.

If you run into performance issues, you may have to make tradeoffs in terms of accuracy. Most games out there don't have a 100% accurate collision. They find creative ways to hide it or otherwise make it unnoticeable during normal gameplay.

Collision shapes (3D)

This guide explains:

- The types of collision shapes available in 3D in Godot.
- Using a convex or a concave mesh as a collision shape.
- Performance considerations regarding 3D collisions.

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

You can define the shape of a PhysicsBody3D by adding one or more CollisionShape3Ds as child nodes. Note that you must add a Shape3D *resource* to collision shape nodes in the Inspector dock.

Note

When you add multiple collision shapes to a single PhysicsBody, you don't have to worry about them overlapping. They won't "collide" with each other.

Primitive collision shapes

Godot provides the following primitive collision shape types:

- BoxShape3D
- SphereShape3D
- CapsuleShape3D
- CylinderShape3D

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.

Convex collision shapes

Convex collision shapes are a compromise between primitive collision shapes and concave collision shapes. They can represent shapes of any complexity, but with an important caveat. As their name implies, an individual shape can only represent a *convex* shape. For instance, a pyramid is *convex*, but a hollow box is *concave*. To define a concave object with a single collision shape, you need to use a concave collision shape.

Depending on the object's complexity, you may get better performance by using multiple convex shapes instead of a concave collision shape. Godot lets you use *convex decomposition* to generate convex shapes that roughly match a hollow object. Note this performance advantage no longer applies after a certain amount of convex shapes. For large and complex objects such as a whole level, we recommend using concave shapes instead.

You can generate one or several convex collision shapes from the editor by selecting a MeshInstance3D and using the **Mesh** menu at the top of the 3D viewport. The editor exposes two generation modes:

- **Create Single Convex Collision Sibling** uses the Quickhull algorithm. It creates one CollisionShape node with an automatically generated convex collision shape. Since it only generates a single shape, it provides good performance and is ideal for small objects.
- **Create Multiple Convex Collision Siblings** uses the V-HACD algorithm. It creates several CollisionShape nodes, each with a convex shape. Since it generates multiple shapes, it is more accurate for concave objects at the cost of performance. For objects with medium complexity, it will likely be faster than using a single concave collision shape.

Concave or trimesh collision shapes

Concave collision shapes, also called trimesh collision shapes, can take any form, from a few triangles to thousands of triangles. Concave shapes are the slowest option but are also the most accurate in Godot. **You can only use concave shapes within StaticBodies.** They will not work with CharacterBodies or RigidBodies unless the RigidBody's mode is Static.

Note

Even though concave shapes offer the most accurate *collision*, contact reporting can be less precise than primitive shapes.

When not using GridMaps for level design, concave shapes are the best approach for a level's collision. That said, if your level has small details, you may want to exclude those from collision for performance and game feel. To do so, you can build a simplified collision mesh in a 3D modeler and have Godot generate a collision shape for it automatically. More on that below

Note that unlike primitive and convex shapes, a concave collision shape doesn't have an actual "volume". You can place objects both *outside* of the shape as well as *inside*.

You can generate a concave collision shape from the editor by selecting a MeshInstance3D and using the **Mesh** menu at the top of the 3D viewport. The editor exposes two options:

- **Create Trimesh Static Body** is a convenient option. It creates a StaticBody containing a concave shape matching the mesh's geometry.
- **Create Trimesh Collision Sibling** creates a CollisionShape node with a concave shape matching the mesh's geometry.

See also

See Importing 3D scenes for information on how to export models for Godot and automatically generate collision shapes on import.

Performance caveats

You aren't limited to a single collision shape per PhysicsBody. Still, we recommend keeping the number of shapes as low as possible to improve performance, especially for dynamic objects like RigidBodies and CharacterBodies. On top of that, avoid translating, rotating, or scaling CollisionShapes to benefit from the physics engine's internal optimizations.

When using a single non-transformed collision shape in a StaticBody, the engine's *broad phase* algorithm can discard inactive PhysicsBodies. The *narrow phase* will then only have to take into account the active bodies' shapes. If a StaticBody has many collision shapes, the broad phase will fail. The narrow phase, which is slower, must then perform a collision check against each shape.

If you run into performance issues, you may have to make tradeoffs in terms of accuracy. Most games out there don't have a 100% accurate collision. They find creative ways to hide it or otherwise make it unnoticeable during normal gameplay.

Large world coordinates

Note

Large world coordinates are mainly useful in 3D projects; they are rarely required in 2D projects. Also, unlike 3D rendering, 2D rendering currently doesn't benefit from increased precision when large world coordinates are enabled.

Why use large world coordinates?

In Godot, physics simulation and rendering both rely on *floating-point* numbers. However, in computing, floating-point numbers have **limited precision and range**. This can be a problem for games with huge worlds, such as space or planetary-scale simulation games.

Precision is the greatest when the value is close to `0.0`. Precision becomes gradually lower as the value increases or decreases away from `0.0`. This occurs every time the floating-point number's *exponent* increases, which happens when the floating-point number surpasses a power of 2 value $(2, 4, 8, 16, \ldots)$. Every time this occurs, the number's minimum step will *increase*, resulting in a loss of precision.

In practice, this means that as the player moves away from the world origin (`Vector2(0, 0)` in 2D games or `Vector3(0, 0, 0)` in 3D games), precision will decrease.

This loss of precision can result in objects appearing to "vibrate" when far away from the world origin, as the model's position will snap to the nearest value that can be represented in a floating-point number. This can also result in physics glitches that only occur when the player is far from the world origin.

The range determines the minimum and maximum values that can be stored in the number. If the player tries to move past this range, they will simply not be able to. However, in practice, floating-point precision almost always becomes a problem before the range does.

The range and precision (minimum step between two exponent intervals) are determined by the floating-point number type. The *theoretical* range allows extremely high values to be stored in single-precision floats, but with very low precision. In practice, a floating-point type that cannot represent all integer values is not very useful. At extreme values, precision becomes so low that the number cannot even distinguish two separate *integer* values from each other.

This is the range where individual integer values can be represented in a floating-point number:

- **Single-precision float range (represent all integers):** Between -16,777,216 and 16,777,216
- **Double-precision float range (represent all integers):** Between -9 quadrillon and 9 quadrillon

Range	Single step	Double step	Comment
[1; 2]	~0.0000001	~1e-15	Precision becomes greater near 0.0 (this table is abbreviated).
[2; 4]	~0.0000002	~1e-15	
[4; 8]	~0.0000005	~1e-15	
[8; 16]	~0.000001	~1e-14	

[16; 32]	~0.000002	~1e-14	
[32; 64]	~0.000004	~1e-14	
[64; 128]	~0.000008	~1e-13	
[128; 256]	~0.000015	~1e-13	
[256; 512]	~0.00003	~1e-13	
[512; 1024]	~0.00006	~1e-12	
[1024; 2048]	~0.0001	~1e-12	
[2048; 4096]	~0.0002	~1e-12	Maximum *recommended* single-precision range for a first-person 3D game without rendering artifacts or physics glitches.
[4096; 8192]	~0.0005	~1e-12	Maximum *recommended* single-precision range for a third-person 3D game without rendering artifacts or physics glitches.

[8192; 16384]	~0.001	~1e-12	
[16384; 32768]	~0.0019	~1e-11	Maximum *recommended* single-precision range for a top-down 3D game without rendering artifacts or physics glitches.
[32768; 65536]	~0.0039	~1e-11	Maximum *recommended* single-precision range for any 3D game. Double precision (large world coordinates) is usually required past this point.
[65536; 131072]	~0.0078	~1e-11	
[131072; 262144]	~0.0156	~1e-10	
> 262144	> ~0.0313	~1e-10 (0.0000000001)	Double-precision remains far more precise than single-precision past this value.

When using single-precision floats, it is possible to go past the suggested ranges, but more visible artifacting will occur and physics glitches will be more common (such as the player not walking straight in certain directions).

See also

See the Demystifying Floating Point Precision [https://blog.demofox.org/2017/11/21/] article for more information.

How large world coordinates work

Large world coordinates (also known as **double-precision physics**) increase the precision level of all floating-point computations within the engine.

By default, float is 64-bit in GDScript, but Vector2, Vector3 and Vector4 are 32-bit. This means that the precision of vector types is much more limited. To resolve this, we can increase the number of bits used to represent a floating-point number in a Vector type. This results in an *exponential* increase in precision, which means the final value is not just twice as precise, but potentially thousands of times more precise at high values. The maximum value that can be represented is also greatly increased by going from a single-precision float to a double-precision float.

To avoid model snapping issues when far away from the world origin, Godot's 3D rendering engine will increase its precision for rendering operations when large world coordinates are enabled. The shaders do not use double-precision floats for performance reasons, but an alternative solution [https://github.com/godotengine/godot/pull/66178] is used to emulate double precision for rendering using single-precision floats.

Note

Enabling large world coordinates comes with a performance and memory usage penalty, especially on 32-bit CPUs. Only enable large world coordinates if you actually need them.

This feature is tailored towards mid-range/high-end desktop platforms. Large world coordinates may not perform well on low-end mobile devices, unless you take steps to reduce CPU usage with other means (such as decreasing the number of physics ticks per second).

On low-end platforms, an *origin shifting* approach can be used instead to allow for large worlds without using double-precision physics and rendering. Origin shifting works with single-precision floats, but it introduces more complexity to game logic, especially in multiplayer games. Therefore, origin shifting is not detailed on this page.

Who are large world coordinates for?

Large world coordinates are typically required for 3D space or planetary-scale simulation games. This extends to games that require supporting *very* fast movement speeds, but also very slow *and* precise movements at times.

On the other hand, it's important to only use large world coordinates when actually required (for performance reasons). Large world coordinates are usually **not** required for:

- 2D games, as precision issues are usually less noticeable.
- Games with small-scale or medium-scale worlds.
- Games with large worlds, but split into different levels with loading sequences in between. You can center each level portion around the world origin to avoid precision issues without a performance penalty.
- Open world games with a *playable on-foot area* not exceeding 8192×8192 meters (centered around the world origin). As shown in the above table, the level of precision remains acceptable within that range, even for a first-person game.

If in doubt, you probably don't need to use large world coordinates in your project. For reference, most modern AAA open world titles don't use a large world coordinates system and still rely on single-precision floats for both rendering and physics.

Enabling large world coordinates

This process requires recompiling the editor and all export template binaries you intend to use. If you only intend to export your project in release mode, you can skip the compilation of debug export templates. In any case, you'll need to compile an editor build so you can test your large precision world without having to export the project every time.

See the Compiling section for compiling instructions for each target platform. You will need to add the `precision=double` SCons option when compiling the editor and export templates.

The resulting binaries will be named with a `.double` suffix to distinguish them from single-precision binaries (which lack any precision suffix). You can then specify the binaries as custom export templates in your project's export presets in the Export dialog.

Compatibility between single-precision and double-precision builds

When saving a *binary* resource using the ResourceSaver singleton, a special flag is stored in the file if the resource was saved using a build that uses double-precision numbers. As a result, all binary resources will change on disk when you switch to a double-precision build and save over them.

Both single-precision and double-precision builds support using the ResourceLoader singleton on resources that use this special flag. This means single-precision builds can load resources saved using double-precision builds and vice versa. Text-based resources don't store a double-precision flag, as they don't require such a flag for correct reading.

Known incompatibilities

- In a networked multiplayer game, the server and all clients should be using the same build type to ensure precision remains consistent across clients. Using different build types *may* work, but various issues can occur.
- The GDExtension API changes in an incompatible way in double-precision builds. This means extensions **must** be rebuilt to work with double-precision builds. On the extension developer's end, the `REAL_T_IS_DOUBLE` define is enabled when building a GDExtension with `precision=double`. `real_t` can be used as an alias for `float` in single-precision builds, and `double` in double-precision builds.

Limitations

Since 3D rendering shaders don't actually use double-precision floats, there are some limitations when it comes to 3D rendering precision:

- Shaders using the `skip_vertex_transform` or `world_vertex_coords` don't benefit from increased precision.
- Triplanar mapping doesn't benefit from increased precision. Materials using triplanar mapping will exhibit visible jittering when far away from the world origin.

2D rendering currently doesn't benefit from increased precision when large world coordinates are enabled. This can cause visible model snapping to occur when far away from the world origin (starting from a few million pixels at typical zoom levels). 2D physics calculations will still benefit from increased precision though.

Troubleshooting physics issues

When working with a physics engine, you may encounter unexpected results.

While many of these issues can be resolved through configuration, some of them are the result of engine bugs. For known issues related to the physics engine, see open physics-related issues on GitHub [https://github.com/godotengine/godot/issues?q=is%3Aopen+is%3Aissue+label%3Atopic%3Aphysics]. Looking through closed issues [https://github.com/godotengine/godot/issues?q=+is%3Aclosed+is%3Aissue+label%3Atopic%3Aphysics] can also help answer questions related to physics engine behavior.

Objects are passing through each other at high speeds

This is known as *tunneling*. Enabling **Continuous CD** in the RigidBody properties can sometimes resolve this issue. If this does not help, there are other solutions you can try:

- Make your static collision shapes thicker. For example, if you have a thin floor that the player can't get below in some way, you can make the collider thicker than the floor's visual representation.
- Modify your fast-moving object's collision shape depending on its movement speed. The faster the object moves, the larger the collision shape should extend outside of the object to ensure it can collide with thin walls more reliably.
- Increase **Physics Ticks Per Second** in the advanced Project Settings. While this has other benefits (such as more stable simulation and reduced input lag), this increases CPU utilization and may not be viable for mobile/web platforms.

> Multipliers of the default value of `60` (such as `120`, `180` or `240`) should be preferred for a smooth appearance on most displays.

Stacked objects are unstable and wobbly

Despite seeming like a simple problem, stable RigidBody simulation with stacked objects is difficult to implement in a physics engine. This is caused by integrating forces going against each other. The more stacked objects are present, the stronger the forces will be against each other. This eventually causes the simulation to become wobbly, making the objects unable to rest on top of each other without moving.

Increasing the physics simulation rate can help alleviate this issue. To do so, increase **Physics Ticks Per Second** in the advanced Project Settings. Note that increases CPU utilization and may not be viable for mobile/web platforms. Multipliers of the default value of `60` (such as `120`, `180` or `240`) should be preferred for a smooth appearance on most displays.

Scaled physics bodies or collision shapes do not collide correctly

Godot does not currently support scaling of physics bodies or collision shapes. As a workaround, change the collision shape's extents instead of changing its scale. If you want the visual representation's scale to change as well, change the scale of the underlying visual representation (Sprite2D, MeshInstance3D, …) and change the collision shape's extents separately. Make sure the collision shape is not a child of the visual representation in this case.

Since resources are shared by default, you'll have to make the collision shape resource unique if you don't want the change to be applied to all nodes using the same collision shape resource in the scene. This can be done by calling `duplicate()` in a script on the collision shape resource *before* changing its size.

Thin objects are wobbly when resting on the floor

This can be due to one of two causes:

- The floor's collision shape is too thin.
- The RigidBody's collision shape is too thin.

In the first case, this can be alleviated by making the floor's collision shape thicker. For example, if you have a thin floor that the player can't get below in some way, you can make the collider thicker than the floor's visual representation.

In the second case, this can usually only be resolved by increasing the physics simulation rate (as making the shape thicker would cause a disconnect between the RigidBody's visual representation and its collision).

In both cases, increasing the physics simulation rate can also help alleviate this issue. To do so, increase **Physics Ticks Per Second** in the advanced Project Settings. Note that this increases CPU utilization and may not be viable for mobile/web platforms. Multipliers of the default value of `60` (such as `120`, `180` or `240`) should be preferred for a smooth appearance on most displays.

Cylinder collision shapes are unstable

During the transition from Bullet to GodotPhysics in Godot 4, cylinder collision shapes had to be reimplemented from scratch.

However, cylinder collision shapes are one of the most difficult shapes to support, which is why many other physics engines don't provide any support for them. There are several known bugs with cylinder collision shapes currently.

We recommend using box or capsule collision shapes for characters for now. Boxes generally provide the best reliability, but have the downside of making the character take more space diagonally. Capsule collision shapes do not have this downside, but their shape can make precision platforming more difficult.

VehicleBody simulation is unstable, especially at high speeds

When a physics body moves at a high speed, it travels a large distance between each physics step. For instance, when using the 1 unit = 1 meter convention in 3D, a vehicle moving at 360 km/h will travel 100 units per second. With the default physics simulation rate of 60 Hz, the vehicle moves by ~1.67 units each physics tick. This means that small objects may be ignored entirely by the vehicle (due to tunneling), but also that the simulation has little data to work with in general at such a high speed.

Fast-moving vehicles can benefit a lot from an increased physics simulation rate. To do so, increase **Physics Ticks Per Second** in the advanced Project Settings. Note that this increases CPU utilization and may not be viable for mobile/web platforms. Multipliers of the default value of `60` (such as `120`, `180` or `240`) should be preferred for a smooth appearance on most displays.

Collision results in bumps when an object moves across tiles

This is a known issue in the physics engine caused by the object bumping on a shape's edges, even though that edge is covered by another shape. This can occur in both 2D and 3D.

The best way to work around this issue is to create a "composite" collider. This means that instead of individual tiles having their collision, you create a single collision shape representing the collision for a group of tiles. Typically, you should split composite colliders on a per-island basis (which means each group of touching tiles gets its own collider).

Using a composite collider can also improve physics simulation performance in certain cases. However, since the composite collision shape is much more complex, this may not be a net performance win in all cases.

Framerate drops when an object touches another object

This is likely due to one of the objects using a collision shape that is too complex. Convex collision shapes should use a number of shapes as low as possible for performance reasons. When relying on Godot's automatic generation, it's possible that you ended up with dozens if not hundreds of shapes created for a single convex shape collision resource.

In some cases, replacing a convex collider with a couple of primitive collision shapes (box, sphere, or capsule) can deliver better performance.

This issue can also occur with StaticBodies that use very detailed trimesh (concave) collisions. In this case, use a simplified representation of the level geometry as a collider. Not only this will improve physics simulation performance significantly, but this can also improve stability by letting you remove small fixtures and crevices from being considered by collision.

Physics simulation is unreliable when far away from the world origin

This is caused by floating-point precision errors, which become more pronounced as the physics simulation occurs further away from the world origin. This issue also affects rendering, which results in wobbly camera movement when far away from the world origin. See Large world coordinates for more information.