Input handling

- Using InputEvent
- Input examples
- Mouse and input coordinates
- Customizing the mouse cursor
- Controllers, gamepads, and joysticks
- Handling quit requests

Using InputEvent

What is it?

Managing input is usually complex, no matter the OS or platform. To ease this a little, a special built-in type is provided, InputEvent. This datatype can be configured to contain several types of input events. Input events travel through the engine and can be received in multiple locations, depending on the purpose.

Here is a quick example, closing your game if the escape key is hit:

GDScript

```
func _unhandled_input(event):
    if event is InputEventKey:
        if event.pressed and event.keycode == KEY_ESCAPE:
            get_tree().quit()
```

C#

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event is InputEventKey eventKey)
        if (eventKey.Pressed && eventKey.Keycode ==
Key.Escape)
            GetTree().Quit();
}
```

However, it is cleaner and more flexible to use the provided InputMap feature, which allows you to define input actions and assign them different keys. This way, you can define multiple keys for the same action (e.g. the keyboard escape key and the start button on a gamepad). You can then more easily change this mapping in the project settings without updating your code, and even build a key mapping feature on top of it to allow your game to change the key mapping at runtime!

You can set up your InputMap under **Project > Project Settings > Input Map** and then use those actions like this:

GDScript

```
func _process(delta):
    if Input.is_action_pressed("ui_right"):
        # Move right.
```

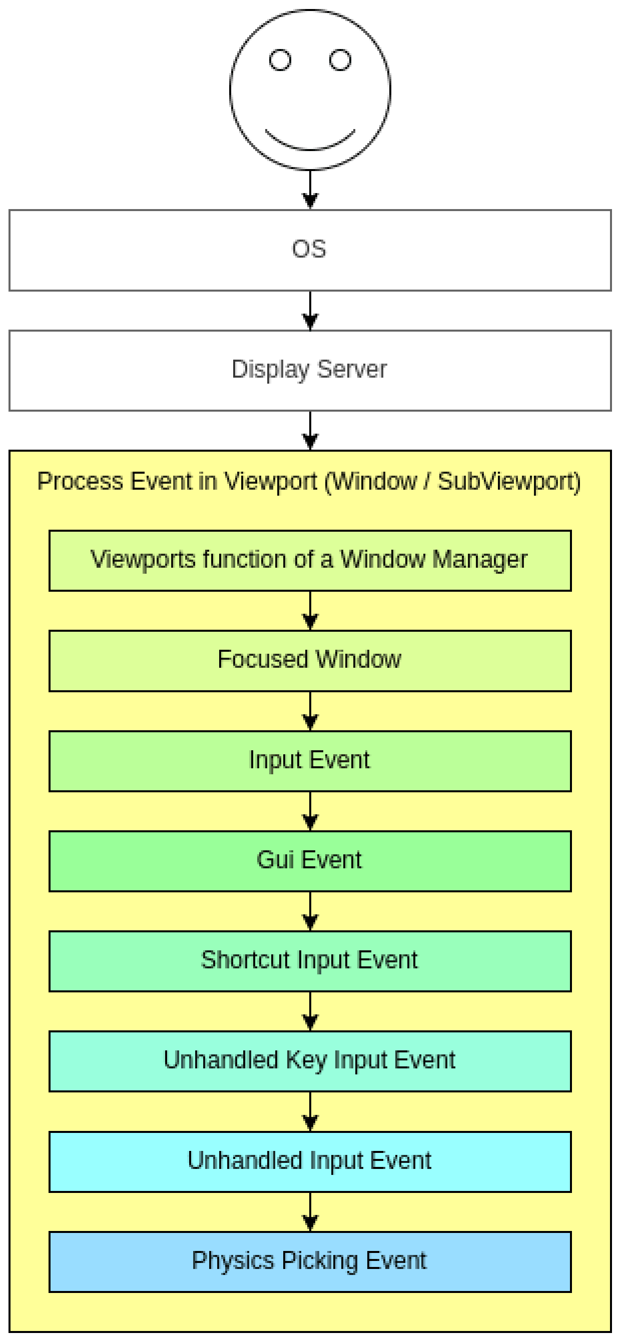
C#

```
public override void _Process(double delta)
{
    if (Input.IsActionPressed("ui_right"))
    {
        // Move right.
    }
}
```

How does it work?

Every input event is originated from the user/player (though it's possible to generate an InputEvent and feed them back to the engine, which is useful for gestures). The DisplayServer for each platform will read events from the operating system, then feed them to the root Window.

The window's Viewport does quite a lot of stuff with the received input, in order:



1. If the Viewport is embedding Windows, the Viewport tries to interpret the event in its capability as a Window-Manager (e.g. for resizing or moving Windows).
2. Next if an embedded Window is focused, the event is sent to that Window and processed in the Windows Viewport and afterwards treated as handled. If no embedded Window is focused, the event is sent to the nodes of the current viewport in the following order.
3. First of all, the standard Node._input() function will be called in any node that overrides it (and hasn't disabled input

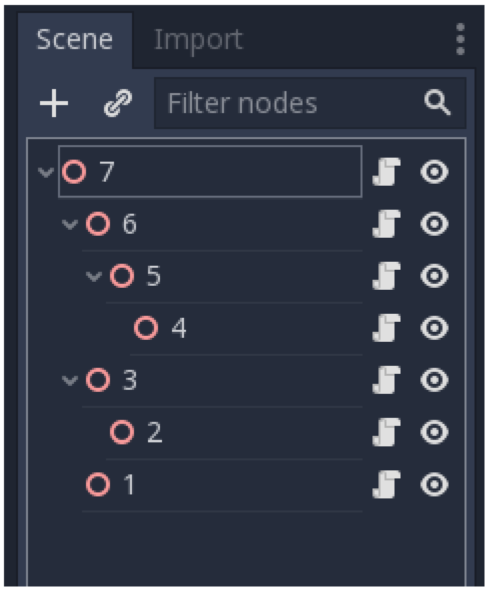
processing with Node.set_process_input()). If any function consumes the event, it can call Viewport.set_input_as_handled(), and the event will not spread any more. This ensures that you can filter all events of interest, even before the GUI. For gameplay input, Node._unhandled_input() is generally a better fit, because it allows the GUI to intercept the events.

4. Second, it will try to feed the input to the GUI, and see if any control can receive it. If so, the Control will be called via the virtual function Control._gui_input() and the signal "gui_input" will be emitted (this function is re-implementable by script by inheriting from it). If the control wants to "consume" the event, it will call Control.accept_event() and the event will not spread any more. Use the Control.mouse_filter property to control whether a Control is notified of mouse events via Control._gui_input() callback, and whether these events are propagated further.
5. If so far no one consumed the event, the Node._shortcut_input() callback will be called if overridden (and not disabled with Node.set_process_shortcut_input()). This happens only for InputEventKey, InputEventShortcut and InputEventJoypadButton. If any function consumes the event, it can call Viewport.set_input_as_handled(), and the event will not spread any more. The shortcut input callback is ideal for treating events that are intended as shortcuts.
6. If so far no one consumed the event, the Node._unhandled_key_input() callback will be called if overridden (and not disabled with Node.set_process_unhandled_key_input()). This happens only if the event is a InputEventKey. If any function consumes the event, it can call Viewport.set_input_as_handled(), and the event will not spread any more. The unhandled key input callback is ideal for key events.
7. If so far no one consumed the event, the Node._unhandled_input() callback will be called if overridden (and not disabled with Node.set_process_unhandled_input()). If any function consumes the event, it can call Viewport.set_input_as_handled(), and the event will not

spread any more. The unhandled input callback is ideal for full-screen gameplay events, so they are not received when a GUI is active.

8. If no one wanted the event so far, and Object Picking is turned on, the event is used for object picking. For the root viewport, this can also be enabled in Project Settings. In the case of a 3D scene if a Camera3D is assigned to the Viewport, a ray to the physics world (in the ray direction from the click) will be cast. If this ray hits an object, it will call the CollisionObject3D._input_event() function in the relevant physics object (bodies receive this callback by default, but areas do not. This can be configured through Area3D properties). In the case of a 2D scene, conceptually the same happens with CollisionObject2D._input_event().

When sending events to its child and descendant nodes, the viewport will do so, as depicted in the following graphic, in a reverse depth-first order, starting with the node at the bottom of the scene tree, and ending at the root node. Excluded from this process are Windows and SubViewports.



This order doesn't apply to Control._gui_input(), which uses a different method based on event location or focused Control.

Since Viewports don't send events to other SubViewports, one of the following methods has to be used:

1. Use a SubViewportContainer, which automatically sends events to its child SubViewports after Node._input() or Control._gui_input().
2. Implement event propagation based on the individual requirements.

GUI events also travel up the scene tree but, since these events target specific Controls, only direct ancestors of the targeted Control node receive the event.

In accordance with Godot's node-based design, this enables specialized child nodes to handle and consume particular events, while their ancestors, and ultimately the scene root, can provide more generalized behavior if needed.

Anatomy of an InputEvent

InputEvent is just a base built-in type, it does not represent anything and only contains some basic information, such as event ID (which is increased for each event), device index, etc.

There are several specialized types of InputEvent, described in the table below:

Event	Description
InputEvent	Empty Input Event.
InputEventKey	Contains a keycode and Unicode value, as well as modifiers.
InputEventMouseButton	Contains click information, such as button, modifiers, etc.

InputEventMouseMotion	Contains motion information, such as relative and absolute positions and speed.
InputEventJoypadMotion	Contains Joystick/Joypad analog axis information.
InputEventJoypadButton	Contains Joystick/Joypad button information.
InputEventScreenTouch	Contains multi-touch press/release information. (only available on mobile devices)
InputEventScreenDrag	Contains multi-touch drag information. (only available on mobile devices)
InputEventMagnifyGesture	Contains a position, a factor as well as modifiers.
InputEventPanGesture	Contains a position, a delta as well as modifiers.

InputEventMIDI	Contains MIDI-related information.
InputEventShortcut	Contains a shortcut.
InputEventAction	Contains a generic action. These events are often generated by the programmer as feedback. (more on this below)

Actions

Actions are a grouping of zero or more InputEvents into a commonly understood title (for example, the default "ui_left" action grouping both joypad-left input and a keyboard's left arrow key). They are not required to represent an InputEvent but are useful because they abstract various inputs when programming the game logic.

This allows for:

- The same code to work on different devices with different inputs (e.g., keyboard on PC, Joypad on console).
- Input to be reconfigured at run-time.
- Actions to be triggered programmatically at run-time.

Actions can be created from the Project Settings menu in the **Input Map** tab and assigned input events.

Any event has the methods InputEvent.is_action(), InputEvent.is_pressed() and InputEvent.

Alternatively, it may be desired to supply the game back with an action from the game code (a good example of this is detecting gestures). The Input singleton has a method for this: Input.parse_input_event(). You would normally use it like this:

GDScript

```
var ev = InputEventAction.new()
# Set as ui_left, pressed.
ev.action = "ui_left"
ev.pressed = true
# Feedback.
Input.parse_input_event(ev)
```

C#

```
var ev = new InputEventAction();
// Set as ui_left, pressed.
ev.SetAction("ui_left");
ev.SetPressed(true);
// Feedback.
Input.ParseInputEvent(ev);
```

InputMap

Customizing and re-mapping input from code is often desired. If your whole workflow depends on actions, the InputMap singleton is ideal for reassigning or creating different actions at run-time. This singleton is not saved (must be modified manually) and its state is run from the project settings (project.godot). So any dynamic system of this type needs to store settings in the way the programmer best sees fit.

Input examples

Introduction

In this tutorial, you'll learn how to use Godot's InputEvent system to capture player input. There are many different types of input your game may use - keyboard, gamepad, mouse, etc. - and many different ways to turn those inputs into actions in your game. This document will show you some of the most common scenarios, which you can use as starting points for your own projects.

Note

For a detailed overview of how Godot's input event system works, see Using InputEvent.

Events versus polling

Sometimes you want your game to respond to a certain input event - pressing the "jump" button, for example. For other situations, you might want something to happen as long as a key is pressed, such as movement. In the first case, you can use the `_input()` function, which will be called whenever an input event occurs. In the second case, Godot provides the Input singleton, which you can use to query the state of an input.

Examples:

GDScript

```
func _input(event):
    if event.is_action_pressed("jump"):
        jump()
```

```
func _physics_process(delta):
    if Input.is_action_pressed("move_right"):
        # Move as long as the key/button is pressed.
        position.x += speed * delta
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event.IsActionPressed("jump"))
    {
        Jump();
    }
}

public override void _PhysicsProcess(double delta)
{
    if (Input.IsActionPressed("move_right"))
    {
        // Move as long as the key/button is pressed.
        position.X += speed * (float)delta;
    }
}
```

This gives you the flexibility to mix-and-match the type of input processing you do.

For the remainder of this tutorial, we'll focus on capturing individual events in `_input()`.

Input events

Input events are objects that inherit from InputEvent. Depending on the event type, the object will contain specific properties related to that event. To see what events actually look like, add a Node and attach the following script:

GDScript

```
extends Node

```

```
func _input(event):
    print(event.as_text())
```

C#

```
using Godot;

public partial class Node : Godot.Node
{
    public override void _Input(InputEvent @event)
    {
        GD.Print(@event.AsText());
    }
}
```

As you press keys, move the mouse, and perform other inputs, you'll see each event scroll by in the output window. Here's an example of the output:

```
A
Mouse motion at position ((971, 5)) with velocity ((0, 0))
Right Mouse Button
Mouse motion at position ((870, 243)) with velocity
((0.454937, -0.454937))
Left Mouse Button
Mouse Wheel Up
A
B
Shift
Alt+Shift
Alt
Shift+T
Mouse motion at position ((868, 242)) with velocity
((-2.134768, 2.134768))
```

As you can see, the results are very different for the different types of input. Key events are even printed as their key symbols. For example, let's consider InputEventMouseButton. It inherits from the following classes:

- InputEvent - the base class for all input events
- InputEventWithModifiers - adds the ability to check if modifiers are pressed, such as `Shift` or `Alt`.

- InputEventMouse - adds mouse event properties, such as `position`
- InputEventMouseButton - contains the index of the button that was pressed, whether it was a double-click, etc.

Tip

It's a good idea to keep the class reference open while you're working with events so you can check the event type's available properties and methods.

You can encounter errors if you try to access a property on an input type that doesn't contain it - calling `position` on `InputEventKey` for example. To avoid this, make sure to test the event type first:

GDScript

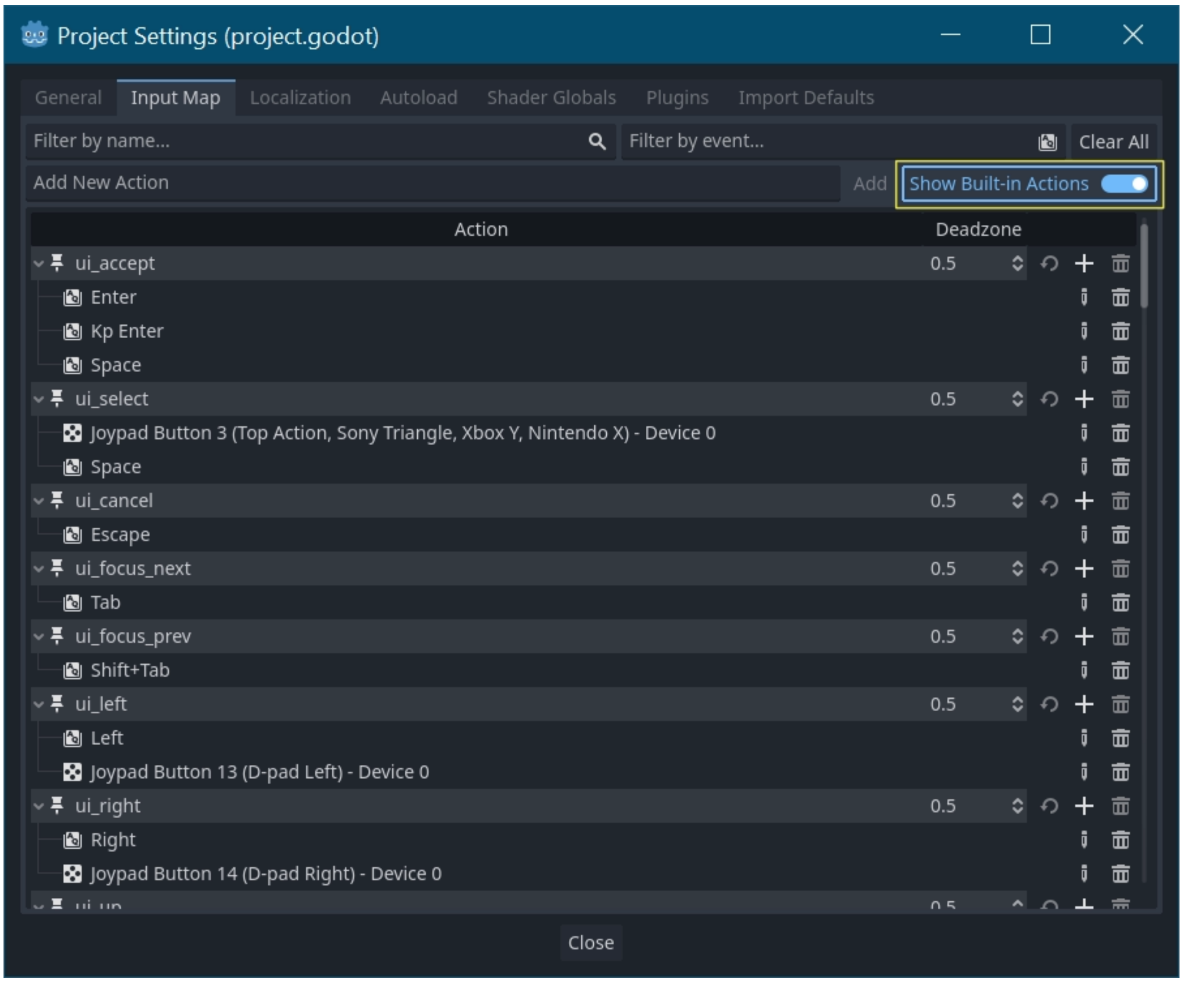
```
func _input(event):
    if event is InputEventMouseButton:
        print("mouse button event at ", event.position)
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event is InputEventMouseButton mouseEvent)
    {
        GD.Print("mouse button event at ",
mouseEvent.Position);
    }
}
```

InputMap

The InputMap is the most flexible way to handle a variety of inputs. You use this by creating named input *actions*, to which you can assign any number of input events, such as keypresses or mouse clicks. To see them, and to add your own, open Project -> Project Settings and select the InputMap tab:



Action	Deadzone
ui_accept	0.5
Enter	
Kp Enter	
Space	
ui_select	0.5
Joypad Button 3 (Top Action, Sony Triangle, Xbox Y, Nintendo X) - Device 0	
Space	
ui_cancel	0.5
Escape	
ui_focus_next	0.5
Tab	
ui_focus_prev	0.5
Shift+Tab	
ui_left	0.5
Left	
Joypad Button 13 (D-pad Left) - Device 0	
ui_right	0.5
Right	
Joypad Button 14 (D-pad Right) - Device 0	
ui_up	0.5

Tip

A new Godot project includes a number of default actions already defined. To see them, turn on `Show Built-in Actions` in the InputMap dialog.

Capturing actions

Once you've defined your actions, you can process them in your scripts using `is_action_pressed()` and `is_action_released()` by passing the name of the action you're looking for:

GDScript

```
func _input(event):
    if event.is_action_pressed("my_action"):
        print("my_action occurred!")
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event.IsActionPressed("my_action"))
    {
        GD.Print("my_action occurred!");
    }
}
```

Keyboard events

Keyboard events are captured in InputEventKey. While it's recommended to use input actions instead, there may be cases where you want to specifically look at key events. For this example, let's check for the `T`:

GDScript

```
func _input(event):
    if event is InputEventKey and event.pressed:
        if event.keycode == KEY_T:
            print("T was pressed")
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event is InputEventKey keyEvent && keyEvent.Pressed)
    {
        if (keyEvent.Keycode == Key.T)
        {
            GD.Print("T was pressed");
        }
    }
}
```

Tip

See @GlobalScope_Key for a list of keycode constants.

Warning

Due to *keyboard ghosting*, not all key inputs may be registered at a given time if you press too many keys at once. Due to their location on the keyboard, certain keys are more prone to ghosting than others. Some keyboards feature antighosting at a hardware level, but this feature is generally not present on low-end keyboards and laptop keyboards.

As a result, it's recommended to use a default keyboard layout that is designed to work well on a keyboard without antighosting. See this Gamedev Stack Exchange question [https://gamedev.stackexchange.com/a/109002] for more information.

Keyboard modifiers

Modifier properties are inherited from InputEventWithModifiers. This allows you to check for modifier combinations using boolean properties. Let's imagine you want one thing to happen when the `T` is pressed, but something different when it's `Shift + T`:

GDScript

```
func _input(event):
    if event is InputEventKey and event.pressed:
        if event.keycode == KEY_T:
            if event.shift_pressed:
                print("Shift+T was pressed")
            else:
                print("T was pressed")
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event is InputEventKey keyEvent && keyEvent.Pressed)
    {
        switch (keyEvent.Keycode)
        {
            case Key.T:
                GD.Print(keyEvent.ShiftPressed ? "Shift+T was
pressed" : "T was pressed");
                break;
        }
    }
}
```

Tip

See @GlobalScope_Key for a list of keycode constants.

Mouse events

Mouse events stem from the InputEventMouse class, and are separated into two types: InputEventMouseButton and InputEventMouseMotion. Note that this means that all mouse events will contain a `position` property.

Mouse buttons

Capturing mouse buttons is very similar to handling key events. @GlobalScope_MouseButton contains a list of `MOUSE_BUTTON_*` constants for each possible button, which will be reported in the event's `button_index` property. Note that the scrollwheel also counts as a button - two buttons, to be precise, with both `MOUSE_BUTTON_WHEEL_UP` and `MOUSE_BUTTON_WHEEL_DOWN` being separate events.

GDScript

```
func _input(event):
    if event is InputEventMouseButton:
        if event.button_index == MOUSE_BUTTON_LEFT and
event.pressed:
            print("Left button was clicked at ",
event.position)
```

```
        if event.button_index == MOUSE_BUTTON_WHEEL_UP and
event.pressed:
            print("Wheel up")
```

C#

```
public override void _Input(InputEvent @event)
{
    if (@event is InputEventMouseButton mouseEvent &&
mouseEvent.Pressed)
    {
        switch (mouseEvent.ButtonIndex)
        {
            case MouseButton.Left:
                GD.Print($"Left button was clicked at
{mouseEvent.Position}");
                break;
            case MouseButton.WheelUp:
                GD.Print("Wheel up");
                break;
        }
    }
}
```

Mouse motion

InputEventMouseMotion events occur whenever the mouse moves. You can find the move's distance with the `relative` property.

Here's an example using mouse events to drag-and-drop a Sprite2D node:

GDScript

```
extends Node


var dragging = false
var click_radius = 32 # Size of the sprite.


func _input(event):
    if event is InputEventMouseButton and event.button_index
== MOUSE_BUTTON_LEFT:
        if (event.position - $Sprite2D.position).length() <
click_radius:
            # Start dragging if the click is on the sprite.
            if not dragging and event.pressed:
                dragging = true
        # Stop dragging if the button is released.
        if dragging and not event.pressed:
            dragging = false

    if event is InputEventMouseMotion and dragging:
        # While dragging, move the sprite with the mouse.
        $Sprite2D.position = event.position
```

C#

```
using Godot;

public partial class MyNode2D : Node2D
{
    private bool _dragging = false;
    private int _clickRadius = 32; // Size of the sprite.

    public override void _Input(InputEvent @event)
    {
        Sprite2D sprite = GetNodeOrNull<Sprite2D>
("Sprite2D");
        if (sprite == null)
        {
            return; // No suitable node was found.
        }

        if (@event is InputEventMouseButton mouseEvent &&
mouseEvent.ButtonIndex == MouseButton.Left)
        {
            if ((mouseEvent.Position -
sprite.Position).Length() < _clickRadius)
            {
                // Start dragging if the click is on the
sprite.
                if (!_dragging && mouseEvent.Pressed)
                {
                    _dragging = true;
                }
            }
            // Stop dragging if the button is released.
            if (_dragging && !mouseEvent.Pressed)
            {
                _dragging = false;
            }
        }
        else
        {
            if (@event is InputEventMouseMotion motionEvent && _dragging)
            {
                // While dragging, move the sprite with the mouse.
                sprite.Position = motionEvent.Position;
            }
        }
    }
}
```

Touch events

If you are using a touchscreen device, you can generate touch events. InputEventScreenTouch is equivalent to a mouse click event, and InputEventScreenDrag works much the same as mouse motion.

Tip

To test your touch events on a non-touchscreen device, open Project Settings and go to the "Input Devices/Pointing" section. Enable "Emulate Touch From Mouse" and your project will interpret mouse clicks and motion as touch events.

Mouse and input coordinates

About

The reason for this small tutorial is to clear up many common mistakes about input coordinates, obtaining mouse position and screen resolution, etc.

Hardware display coordinates

Using hardware coordinates makes sense in the case of writing complex UIs meant to run on PC, such as editors, MMOs, tools, etc. However, it does not make as much sense outside of that scope.

Viewport display coordinates

Godot uses viewports to display content, and viewports can be scaled by several options (see Multiple resolutions tutorial). Use, then, the functions in nodes to obtain the mouse coordinates and viewport size, for example:

GDScript

```
func _input(event):
   # Mouse in viewport coordinates.
   if event is InputEventMouseButton:
       print("Mouse Click/Unclick at: ", event.position)
   elif event is InputEventMouseMotion:
       print("Mouse Motion at: ", event.position)

   # Print the size of the viewport.
   print("Viewport Resolution is: ",
get_viewport().get_visible_rect().size)
```

C#

```
public override void _Input(InputEvent @event)
{
    // Mouse in viewport coordinates.
    if (@event is InputEventMouseButton eventMouseButton)
        GD.Print("Mouse Click/Unclick at: ", 
eventMouseButton.Position);
    else if (@event is InputEventMouseMotion 
eventMouseMotion)
        GD.Print("Mouse Motion at: ", 
eventMouseMotion.Position);

    // Print the size of the viewport.
    GD.Print("Viewport Resolution is: ", 
GetViewport().GetVisibleRect().Size);
}
```

Alternatively, it's possible to ask the viewport for the mouse position:

GDScript

```
get_viewport().get_mouse_position()
```

C#

```
GetViewport().GetMousePosition();
```

Note

When the mouse mode is set to `Input.MOUSE_MODE_CAPTURED`, the `event.position` value from `InputEventMouseMotion` is the center of the screen. Use `event.relative` instead of `event.position` and `event.velocity` to process mouse movement and position changes.

Customizing the mouse cursor

You might want to change the appearance of the mouse cursor in your game in order to suit the overall design. There are two ways to customize the mouse cursor:

1. Using project settings
2. Using a script

Using project settings is a simpler (but more limited) way to customize the mouse cursor. The second way is more customizable, but involves scripting:

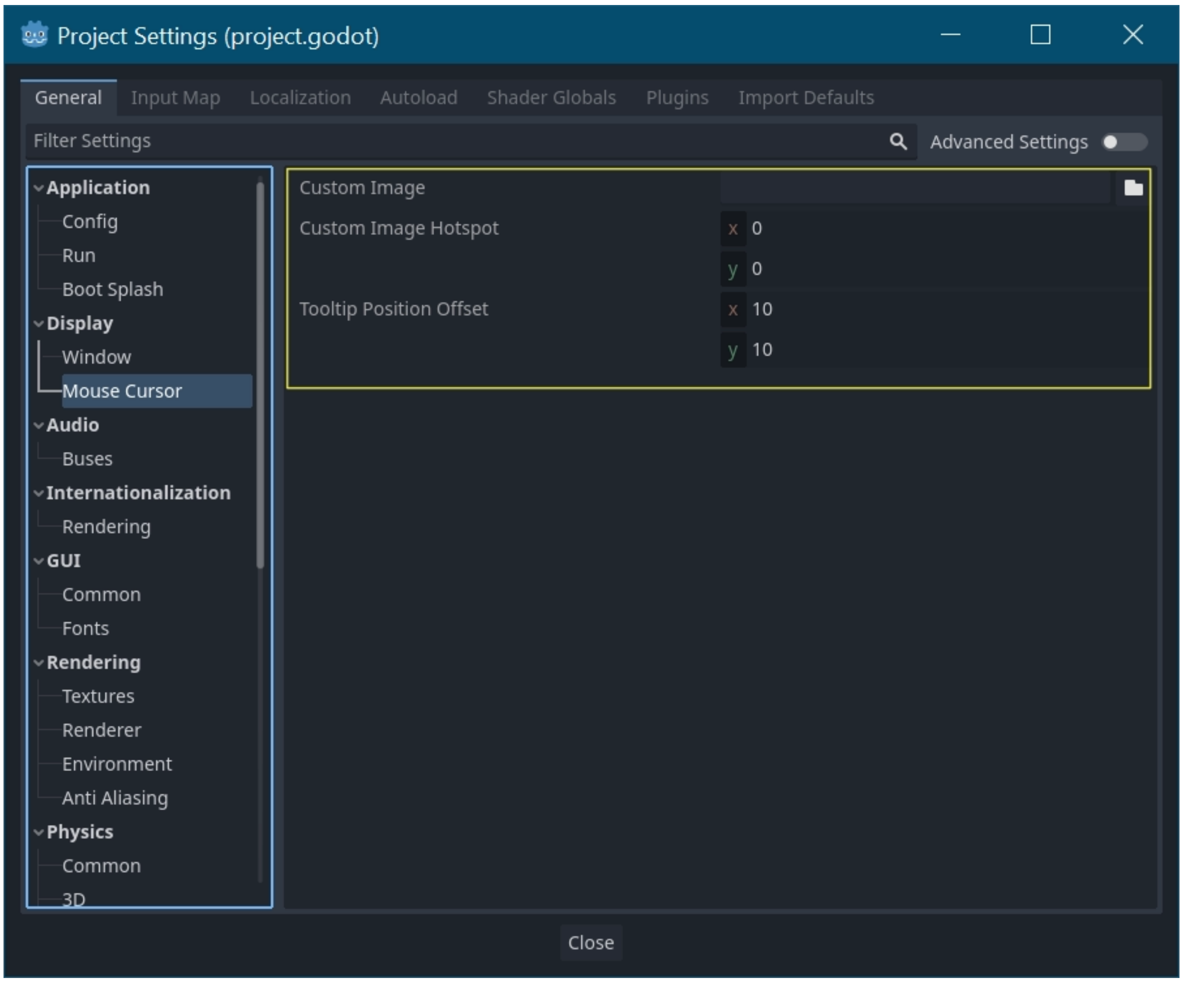
Note

You could display a "software" mouse cursor by hiding the mouse cursor and moving a Sprite2D to the cursor position in a `_process()` method, but this will add at least one frame of latency compared to an "hardware" mouse cursor. Therefore, it's recommended to use the approach described here whenever possible.

If you have to use the "software" approach, consider adding an extrapolation step to better display the actual mouse input.

Using project settings

Open project settings, go to Display>Mouse Cursor. You will see Custom Image, Custom Image Hotspot and Tooltip Position Offset.



Custom Image
Custom Image Hotspot x 0 y 0
Tooltip Position Offset x 10 y 10

Custom Image is the desired image that you would like to set as the mouse cursor. Custom Hotspot is the point in the image that you would like to use as the cursor's detection point.

Warning

The custom image **must** be 256×256 pixels at most. To avoid rendering issues, sizes lower than or equal to 128×128 are recommended.

On the web platform, the maximum allowed cursor image size is 128×128.

Using a script

Create a Node and attach the following script.

GDScript

```
extends Node


# Load the custom images for the mouse cursor.
var arrow = load("res://arrow.png")
var beam = load("res://beam.png")


func _ready():
    # Changes only the arrow shape of the cursor.
    # This is similar to changing it in the project settings.
    Input.set_custom_mouse_cursor(arrow)

    # Changes a specific shape of the cursor (here, the I-
beam shape).
    Input.set_custom_mouse_cursor(beam, Input.CURSOR_IBEAM)
```

C#

```
public override void _Ready()
{
    // Load the custom images for the mouse cursor.
    var arrow = ResourceLoader.Load("res://arrow.png");
    var beam = ResourceLoader.Load("res://beam.png");

    // Changes only the arrow shape of the cursor.
    // This is similar to changing it in the project
settings.
    Input.SetCustomMouseCursor(arrow);

    // Changes a specific shape of the cursor (here, the I-
beam shape).
    Input.SetCustomMouseCursor(beam,
Input.CursorShape.Ibeam);
}
```

See also

Check Input.set_custom_mouse_cursor()'s documentation for more information on usage and platform-specific caveats.

Cursor list

As documented in the Input class (see the **CursorShape** enum), there are multiple mouse cursors you can define. Which ones you want to use depends on your use case.

Controllers, gamepads, and joysticks

Godot supports hundreds of controller models thanks to the community-sourced SDL game controller database [https://github.com/gabomdq/SDL_GameControllerDB].

Controllers are supported on Windows, macOS, Linux, Android, iOS, and HTML5.

Note that more specialized devices such as steering wheels, rudder pedals and HOTAS [https://en.wikipedia.org/wiki/HOTAS] are less tested and may not always work as expected. Overriding force feedback for those devices is also not implemented yet. If you have access to one of those devices, don't hesitate to report bugs on GitHub [https://github.com/godotengine/godot/blob/master/CONTRIBUTING.md#reporting-bugs].

In this guide, you will learn:

- **How to write your input logic to support both keyboard and controller inputs.**
- **How controllers can behave differently from keyboard/mouse input.**
- **Troubleshooting issues with controllers in Godot.**

Supporting universal input

Thanks to Godot's input action system, Godot makes it possible to support both keyboard and controller input without having to write separate code paths. Instead of hardcoding keys or controller buttons in your scripts, you should create *input actions* in the Project Settings which will then refer to specified key and controller inputs.

Input actions are explained in detail on the Using InputEvent page.

Note

Unlike keyboard input, supporting both mouse and controller input for an action (such as looking around in a first-person game) will require different code paths since these have to be handled separately.

Which Input singleton method should I use?

There are 3 ways to get input in an analog-aware way:

- When you have two axes (such as joystick or WASD movement) and want both axes to behave as a single input, use `Input.get_vector()`:

GDScript

```
# `velocity` will be a Vector2 between `Vector2(-1.0, -1.0)`
and `Vector2(1.0, 1.0)`.
# This handles deadzone in a correct way for most use cases.
# The resulting deadzone will have a circular shape as it
generally should.
var velocity = Input.get_vector("move_left", "move_right",
"move_forward", "move_back")

# The line below is similar to `get_vector()`, except that it
handles
# the deadzone in a less optimal way. The resulting deadzone
will have
# a square-ish shape when it should ideally have a circular
shape.
var velocity = Vector2(
        Input.get_action_strength("move_right") -
Input.get_action_strength("move_left"),
        Input.get_action_strength("move_back") -
Input.get_action_strength("move_forward")
).limit_length(1.0)
```

C#

```
// `velocity` will be a Vector2 between `Vector2(-1.0, -1.0)`
and `Vector2(1.0, 1.0)`.
// This handles deadzone in a correct way for most use cases.
// The resulting deadzone will have a circular shape as it
generally should.
Vector2 velocity = Input.GetVector("move_left", "move_right",
"move_forward", "move_back");

// The line below is similar to `get_vector()`, except that
it handles
// the deadzone in a less optimal way. The resulting deadzone
will have
// a square-ish shape when it should ideally have a circular
shape.
Vector2 velocity = new Vector2(
        Input.GetActionStrength("move_right") -
Input.GetActionStrength("move_left"),
        Input.GetActionStrength("move_back") -
Input.GetActionStrength("move_forward")
).LimitLength(1.0);
```

- When you have one axis that can go both ways (such as a throttle on a flight stick), or when you want to handle separate axes individually, use `Input.get_axis()`:

GDScript

```
# `walk` will be a floating-point number between `-1.0` and
`1.0`.
var walk = Input.get_axis("move_left", "move_right")

# The line above is a shorter form of:
var walk = Input.get_action_strength("move_right") -
Input.get_action_strength("move_left")
```

C#

```
// `walk` will be a floating-point number between `-1.0` and
`1.0`.
float walk = Input.GetAxis("move_left", "move_right");

// The line above is a shorter form of:
float walk = Input.GetActionStrength("move_right") -
Input.GetActionStrength("move_left");
```

- For other types of analog input, such as handling a trigger or handling one direction at a time, use `Input.get_action_strength()`:

GDScript

```
# `strength` will be a floating-point number between `0.0`
and `1.0`.
var strength = Input.get_action_strength("accelerate")
```

C#

```
// `strength` will be a floating-point number between `0.0`
and `1.0`.
float strength = Input.GetActionStrength("accelerate");
```

For non-analog digital/boolean input (only "pressed" or "not pressed" values), such as controller buttons, mouse buttons or keyboard keys, use `Input.is_action_pressed()`:

GDScript

```
# `jumping` will be a boolean with a value of `true` or
`false`.
var jumping = Input.is_action_pressed("jump")
```

C#

```
// `jumping` will be a boolean with a value of `true` or
`false`.
bool jumping = Input.IsActionPressed("jump");
```

Note

If you need to know whether an input was *just* pressed in the previous frame, use `Input.is_action_just_pressed()` instead of `Input.is_action_pressed()`. Unlike `Input.is_action_pressed()` which returns `true` as long as the input is held, `Input.is_action_just_pressed()` will only return `true` for one frame after the button has been pressed.

In Godot versions before 3.4, such as 3.3, `Input.get_vector()` and `Input.get_axis()` aren't available. Only `Input.get_action_strength()` and `Input.is_action_pressed()` are available in Godot 3.3.

Vibration

Vibration (also called *haptic feedback*) can be used to enhance the feel of a game. For instance, in a racing game, you can convey the surface the car is currently driving on through vibration, or create a sudden vibration on a crash.

Use the Input singleton's start_joy_vibration method to start vibrating a gamepad. Use stop_joy_vibration to stop vibration early (useful if no duration was specified when starting).

On mobile devices, you can also use vibrate_handheld to vibrate the device itself (independently from the gamepad). On Android, this requires the `VIBRATE` permission to be enabled in the Android export preset before exporting the project.

Note

Vibration can be uncomfortable for certain players. Make sure to provide an in-game slider to disable vibration or reduce its intensity.

Differences between keyboard/mouse and controller input

If you're used to handling keyboard and mouse input, you may be surprised by how controllers handle specific situations.

Dead zone

Unlike keyboards and mice, controllers offer axes with *analog* inputs. The upside of analog inputs is that they offer additional flexibility for actions. Unlike digital inputs which can only provide strengths of `0.0` and `1.0`, an analog input can provide *any* strength between `0.0` and `1.0`. The downside is that without a deadzone system, an analog axis' strength will never be equal to `0.0` due to how the controller is physically built. Instead, it will linger at a low value such as `0.062`. This phenomenon is known as *drifting* and can be more noticeable on old or faulty controllers.

Let's take a racing game as a real-world example. Thanks to analog inputs, we can steer the car slowly in one direction or another. However, without a deadzone system, the car would slowly steer by itself even if the player isn't touching the joystick. This is because the directional axis strength won't be equal to `0.0` when we expect it to. Since we don't want our car to steer by itself in this case, we define a "dead zone" value of `0.2` which will ignore all input whose strength is lower than `0.2`. An ideal dead zone value is high enough to ignore the input caused by joystick drifting, but is low enough to not ignore actual input from the player.

Godot features a built-in deadzone system to tackle this problem. The default value is `0.5`, but you can adjust it on a per-action basis in the Project Settings' Input Map tab. For `Input.get_vector()`, the deadzone can be specified as an optional 5th parameter. If not specified, it will calculate the average deadzone value from all of the actions in the vector.

"Echo" events

Unlike keyboard input, holding down a controller button such as a D-pad direction will **not** generate repeated input events at fixed intervals (also known as "echo" events). This is because the operating system never sends "echo" events for controller input in the first place.

If you want controller buttons to send echo events, you will have to generate InputEvent objects by code and parse them using

Input.parse_input_event() at regular intervals. This can be accomplished with the help of a Timer node.

Window focus

Unlike keyboard input, controller inputs can be seen by **all** windows on the operating system, including unfocused windows.

While this is useful for third-party split screen functionality [https://nucleus-coop.github.io/], it can also have adverse effects. Players may accidentally send controller inputs to the running project while interacting with another window.

If you wish to ignore events when the project window isn't focused, you will need to create an autoload called `Focus` with the following script and use it to check all your inputs:

```
# Focus.gd
extends Node

var focused := true

func _notification(what: int) -> void:
    match what:
        NOTIFICATION_APPLICATION_FOCUS_OUT:
            focused = false
        NOTIFICATION_APPLICATION_FOCUS_IN:
            focused = true


func input_is_action_pressed(action: StringName) -> bool:
    if focused:
        return Input.is_action_pressed(action)

    return false


func event_is_action_pressed(event: InputEvent, action:
StringName) -> bool:
    if focused:
        return event.is_action_pressed(action)

    return false
```

Then, instead of using `Input.is_action_pressed(action)`, use `Focus.input_is_action_pressed(action)` where `action` is the name of the input action. Also, instead of using `event.is_action_pressed(action)`, use `Focus.event_is_action_pressed(event, action)` where `event` is an InputEvent reference and `action` is the name of the input action.

Power saving prevention

Unlike keyboard and mouse input, controller inputs do **not** inhibit sleep and power saving measures (such as turning off the screen after a certain amount of time has passed).

To combat this, Godot enables power saving prevention by default when a project is running. If you notice the system is turning off its display when playing with a gamepad, check the value of **Display > Window > Energy Saving > Keep Screen On** in the Project Settings.

On Linux, power saving prevention requires the engine to be able to use D-Bus. Check whether D-Bus is installed and reachable if running the project within a Flatpak, as sandboxing restrictions may make this impossible by default.

Troubleshooting

See also

You can view a list of known issues with controller support [https://github.com/godotengine/godot/issues?q=is%3Aopen+is%3Aissue+label%3Atopic%3Ainput+gamepad] on GitHub.

My controller isn't recognized by Godot.

First, check that your controller is recognized by other applications. You can use the Gamepad Tester [https://gamepad-tester.com/] website to confirm that your controller is recognized.

My controller has incorrectly mapped buttons or axes.

First, if your controller provides some kind of firmware update utility, make sure to run it to get the latest fixes from the manufacturer. For instance, Xbox One and Xbox Series controllers can have their firmware updated using the Xbox Accessories app [https://www.microsoft.com/en-us/p/xbox-accessories/9nblggh30xj3]. (This application only runs on Windows, so you have to use a Windows machine or a Windows virtual machine with USB support to update the controller's firmware.) After updating the controller's firmware, unpair the controller and pair it again with your PC if you are using the controller in wireless mode.

If buttons are incorrectly mapped, this may be due to an erroneous mapping from the SDL game controller database [https://github.com/gabomdq/SDL_GameControllerDB]. You can contribute an updated mapping to be included in the next Godot version by opening a pull request on the linked repository.

There are many ways to create mappings. One option is to use the mapping wizard in the official Joypads demo [https://godotengine.org/asset-library/asset/140]. Once you have a working mapping for your controller, you can test it by defining the `SDL_GAMECONTROLLERCONFIG` environment variable before running Godot:

Linux/macOS

```
export SDL_GAMECONTROLLERCONFIG="your:mapping:here"
./path/to/godot.x86_64
```

Windows (cmd)

```
set SDL_GAMECONTROLLERCONFIG=your:mapping:here
path\to\godot.exe
```

Windows (PowerShell)

```
$env:SDL_GAMECONTROLLERCONFIG="your:mapping:here"
path\to\godot.exe
```

To test mappings on non-desktop platforms or to distribute your project with additional controller mappings, you can add them by calling Input.add_joy_mapping() as early as possible in a script's `_ready()` function.

My controller works on a given platform, but not on another platform.

Linux

If you're using a self-compiled engine binary, make sure it was compiled with udev support. This is enabled by default, but it is possible to disable udev support by specifying `udev=no` on the SCons command line. If you're using an engine binary supplied by a Linux distribution, double-check whether it was compiled with udev support.

Controllers can still work without udev support, but it is less reliable as regular polling must be used to check for controllers being connected or disconnected during gameplay (hotplugging).

HTML5

HTML5 controller support is often less reliable compared to "native" platforms. The quality of controller support tends to vary wildly across browsers. As a result, you may have to instruct your players to use a different browser if they can't get their controller to work.

Handling quit requests

Quitting

Most platforms have the option to request the application to quit. On desktops, this is usually done with the "x" icon on the window title bar. On Android, the back button is used to quit when on the main screen (and to go back otherwise).

Handling the notification

On desktop and web platforms, Node receives a special `NOTIFICATION_WM_CLOSE_REQUEST` notification when quitting is requested from the window manager.

On Android, `NOTIFICATION_WM_GO_BACK_REQUEST` is sent instead. Pressing the Back button will exit the application if **Application > Config > Quit On Go Back** is checked in the Project Settings (which is the default).

Note

`NOTIFICATION_WM_GO_BACK_REQUEST` isn't supported on iOS, as iOS devices don't have a physical Back button.

Handling the notification is done as follows (on any node):

GDScript

```
func _notification(what):
    if what == NOTIFICATION_WM_CLOSE_REQUEST:
        get_tree().quit() # default behavior
```

C#

```
public override void _Notification(int what)
{
    if (what == NotificationWMCloseRequest)
        GetTree().Quit(); // default behavior
}
```

When developing mobile apps, quitting is not desired unless the user is on the main screen, so the behavior can be changed.

It is important to note that by default, Godot apps have the built-in behavior to quit when quit is requested from the window manager. This can be changed, so that the user can take care of the complete quitting procedure:

GDScript

```
get_tree().set_auto_accept_quit(false)
```

C#

```
GetTree().AutoAcceptQuit = false;
```

Sending your own quit notification

While forcing the application to close can be done by calling SceneTree.quit, doing so will not send the `NOTIFICATION_WM_CLOSE_REQUEST` to the nodes in the scene tree. Quitting by calling SceneTree.quit will not allow custom actions to complete (such as saving, confirming the quit, or debugging), even if you try to delay the line that forces the quit.

Instead, if you want to notify the nodes in the scene tree about the upcoming program termination, you should send the notification yourself:

GDScript

```
get_tree().root.propagate_notification(NOTIFICATION_WM_CLOSE_
REQUEST)
```

C#

```
GetTree().Root.PropagateNotification((int)NotificationWMClose
Request);
```

Sending this notification will inform all nodes about the program termination, but will not terminate the program itself *unlike in 3.X*. In order to achieve the previous behavior, SceneTree.quit should be called after the notification.