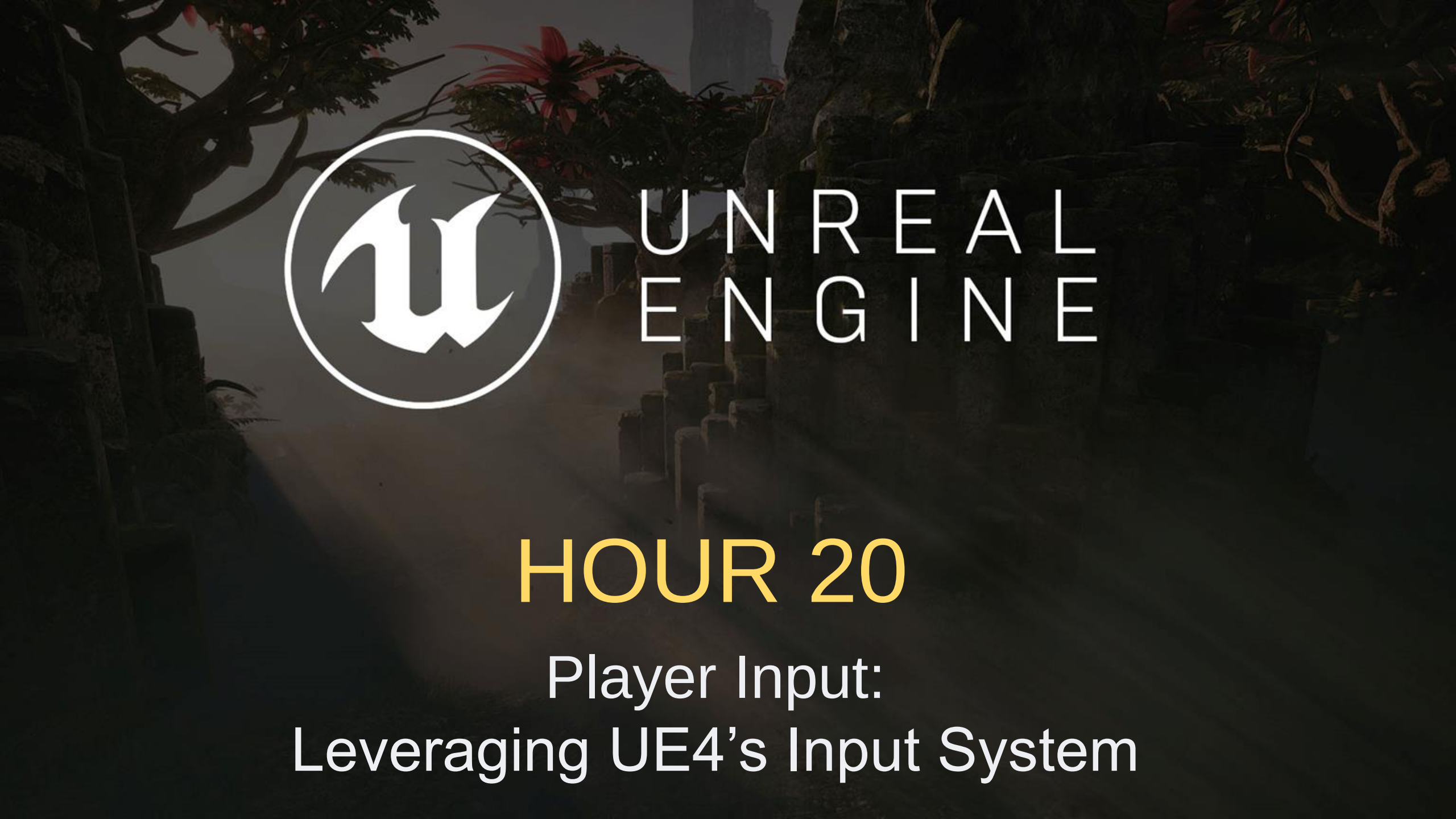


HOUR 20

Player Input:
Leveraging UE4's Input System

INTRODUCTION

Unreal Engine has a robust, flexible input management system.

In this lecture, you will learn how to define inputs using the Project Settings dialog and how to take those inputs and drive gameplay with them in Blueprints.

LECTURE GOALS AND OUTCOMES

Goals

The goals of this lecture are to

- Learn how UE4's input system operates
- Understand the PlayerInput class and the input stack
- Learn about Action and Axis mappings
- Learn how to get VR motion controller input

Outcomes

At the end of this lecture you will be able to

- Create input events in Blueprints
- Create Action and Axis bindings using Project Settings
- Use motion controllers for VR applications

Unreal Engine 4 has a robust, easy-to-use input system.

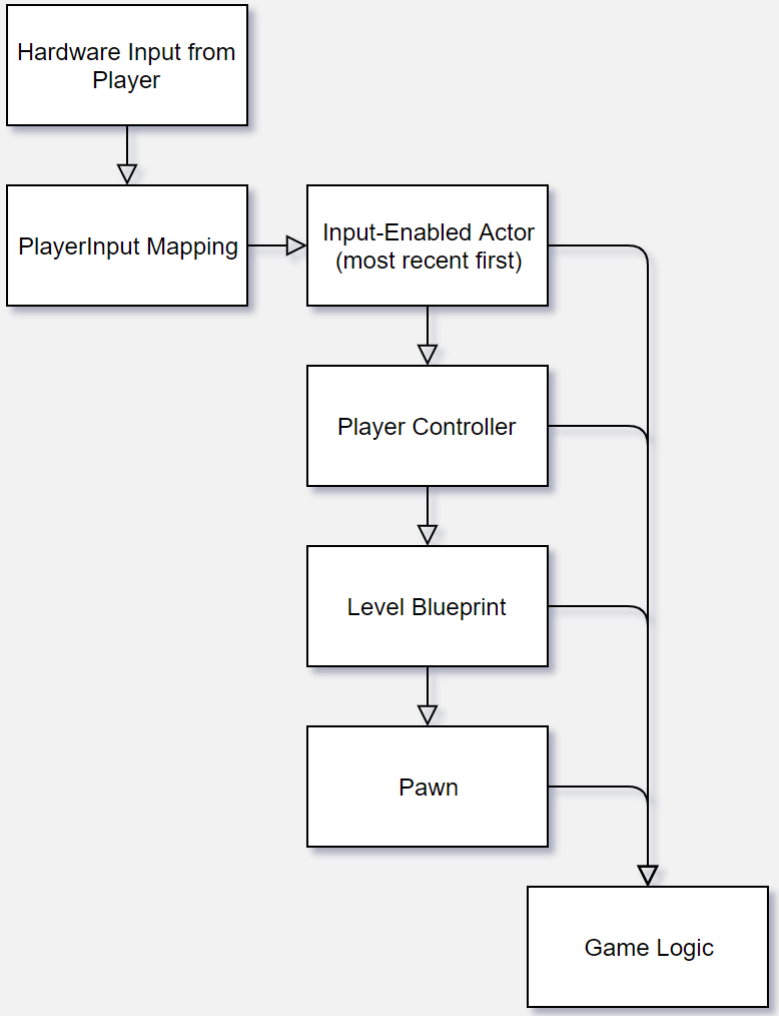
Player input in UE4 is handled via several methods:

- **Action inputs**: Used for onetime events, like hitting or releasing a button
- **Axis inputs**: Used for events whose values can vary over time, like moving an analogue joystick or a mouse
- **Input bindings**: Project-wide abstraction of player input

INPUT OVERVIEW

UE4 handles player input in a cascading manner.

1. First, the PlayerInput class receives the raw input from the various input devices on the system: mouse, keyboard, accelerometer, motion controllers, and so on.
   - PlayerInput converts the raw hardware input into easily accessible input events and Axis values that are then passed on to the Gameplay Framework.
2. If there are any Actors in the scene with input enabled, they will receive the input first. If they don't *consume* that input, it is passed down the stack to the Player Controller, the Level Blueprint, and finally on to the currently possessed Pawn.
3. If any one of the objects consumes the input, the input will not go to the Level below it (unless specifically allowed) and will pass to the game logic (Blueprints or C++).



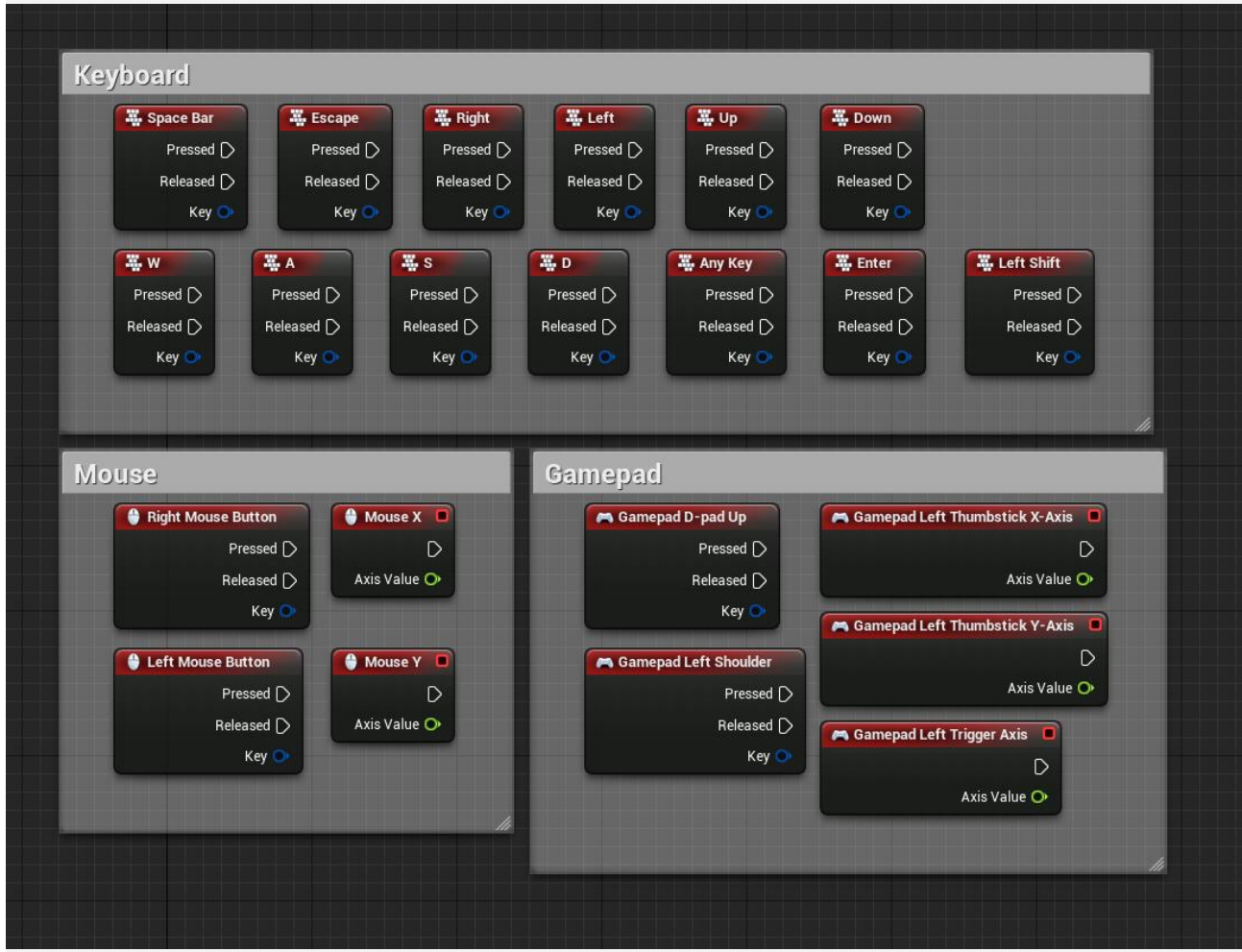
INPUT EVENTS

The most direct way of handling input in UE4 is by using input events.

There's an event for each of the various inputs defined by the PlayerInput class.

There are two types of events: **Action** and **Axis**.

- **Action** events happen once, like a button press; **Axis** events happen continuously, like mouse movement or movement of gamepad thumbsticks and triggers.
- Axis events return an Axis Value float value. These are either in the [0, 1] range or the [–1, 1] range.

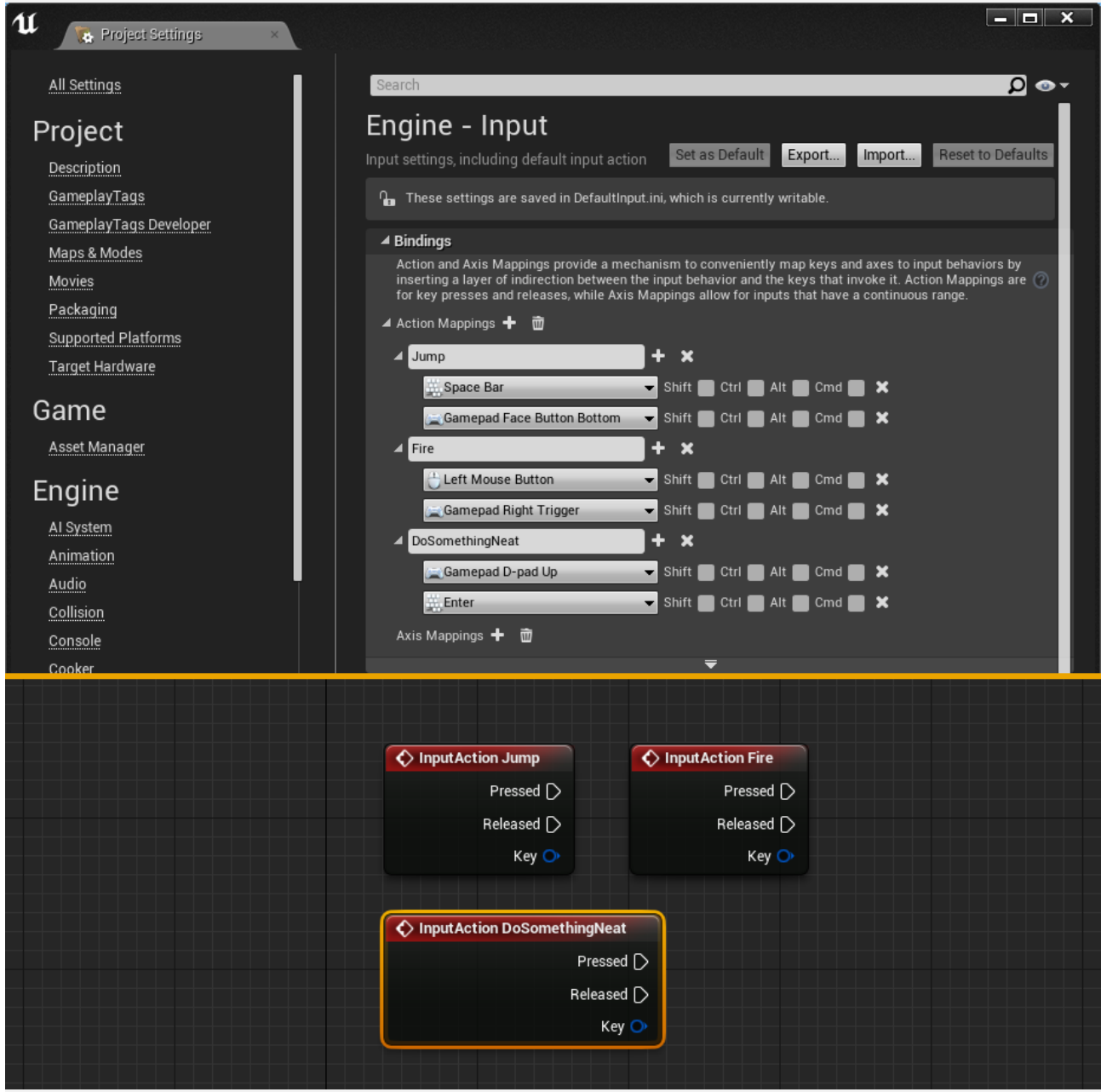


INPUT MAPPING

PlayerInput features the ability to set up input mappings.

Mappings allow you to map discrete inputs to a “friendly name” that can be accessed easily in Blueprints or C++.

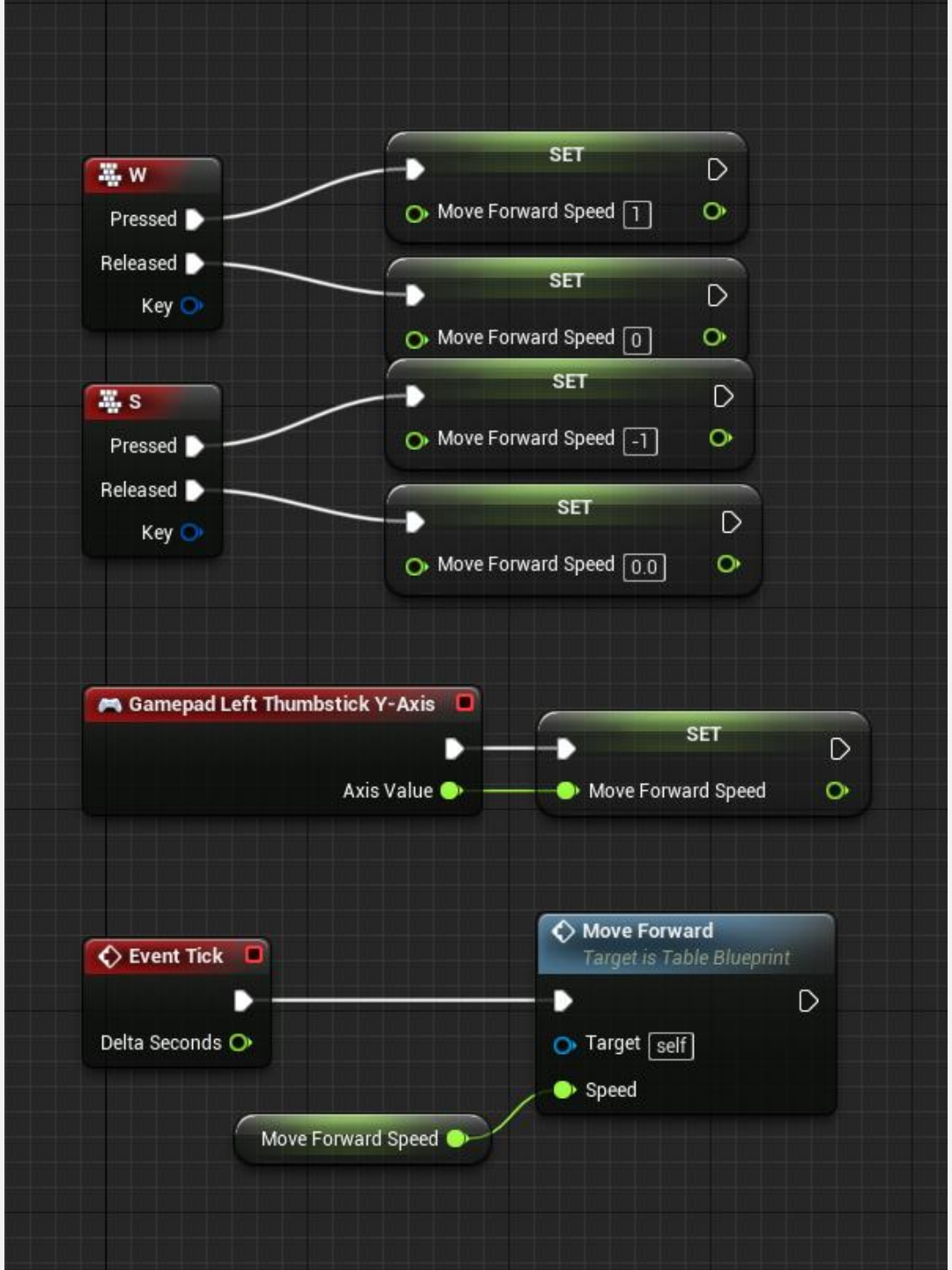
You can combine several hardware inputs in each mapping. This helps organize your code and allows you to modify mappings easily without having to modify your code.



INPUT MAPPING

Using raw input events can make your code very messy and hard to maintain. It can also be difficult to modify inputs globally.

In the example on the right, input events have been used to get the player's input for moving forward. Each is wired to update the Move Forward Speed float variable; then the MoveForward function is called on the Tick event.



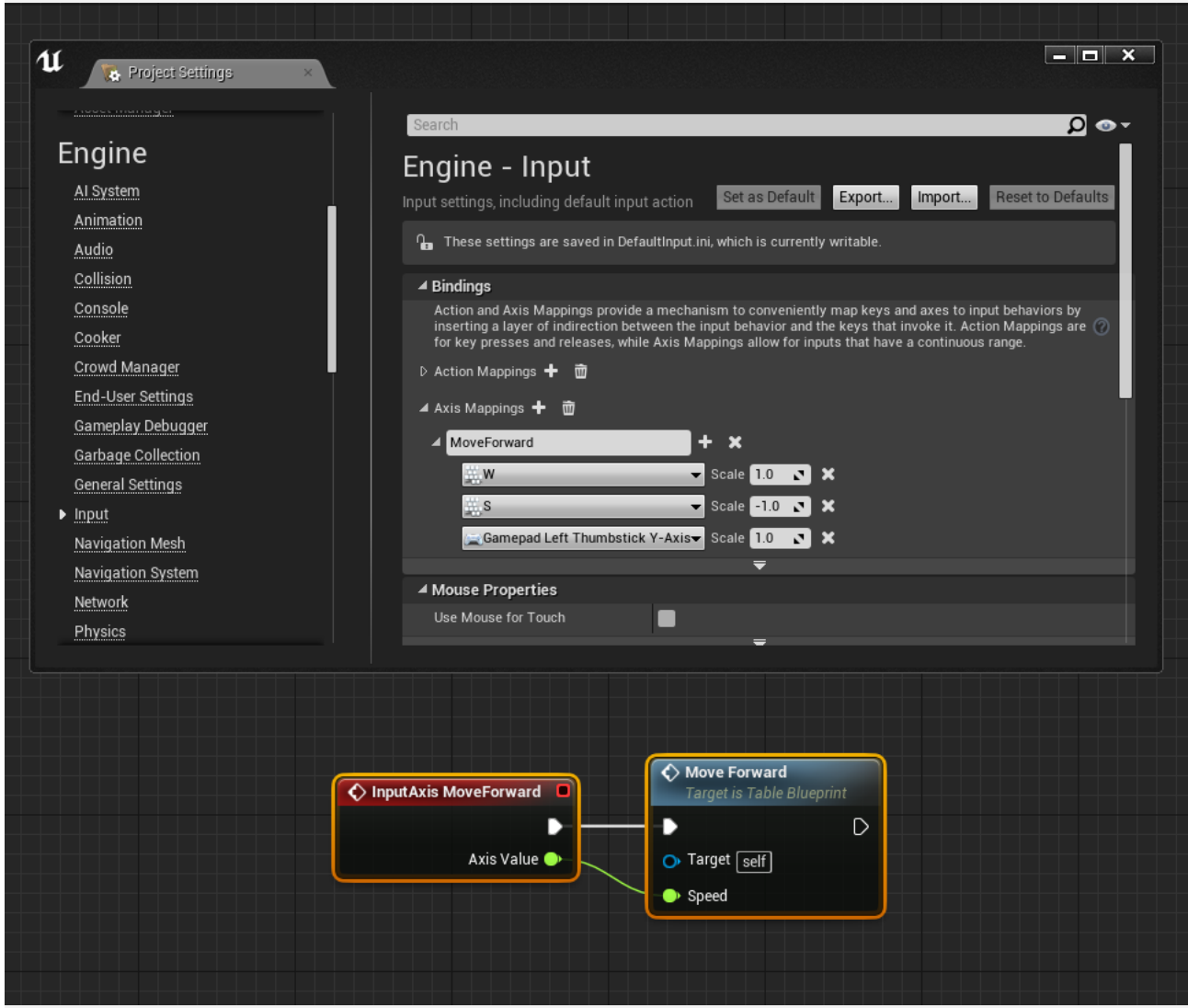
INPUT MAPPING

Here is the same code using input mapping.

Assigning keys to Axis mappings means that any given key will return the assigned Scale value as long as the key is held down.

- Note that the S key is set to return a value of –1.0 when pressed, while the W key will return 1.0.

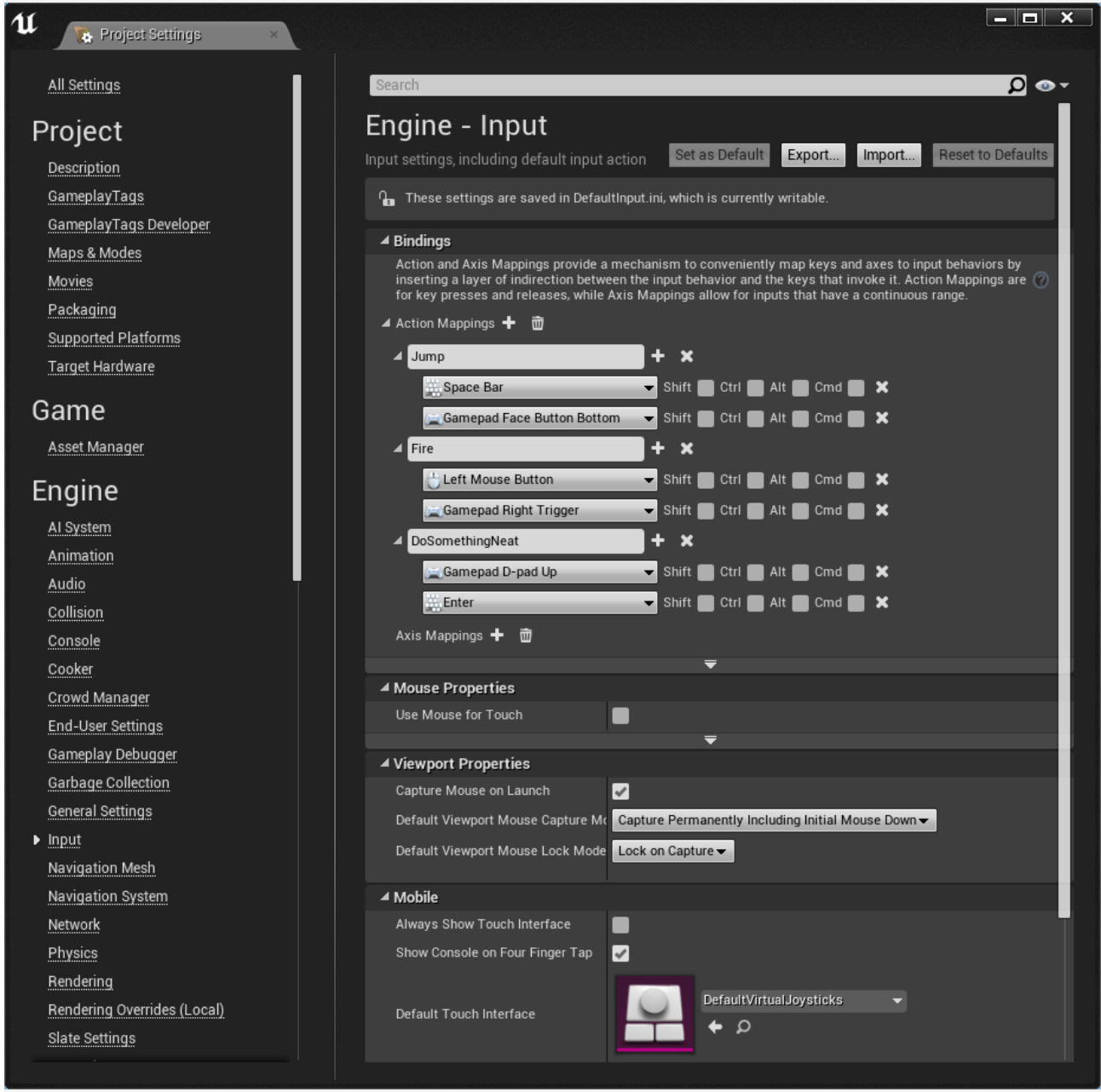
The Gamepad Left Thumbstick Y-Axis will return 1.0 when pushed all the way up, –1.0 when held all the way down, and 0 when released.



INPUT MAPPING

Mappings are stored in DefaultInput.ini in your project's Config folder. You can edit them directly in the .ini file or by using the Editor.

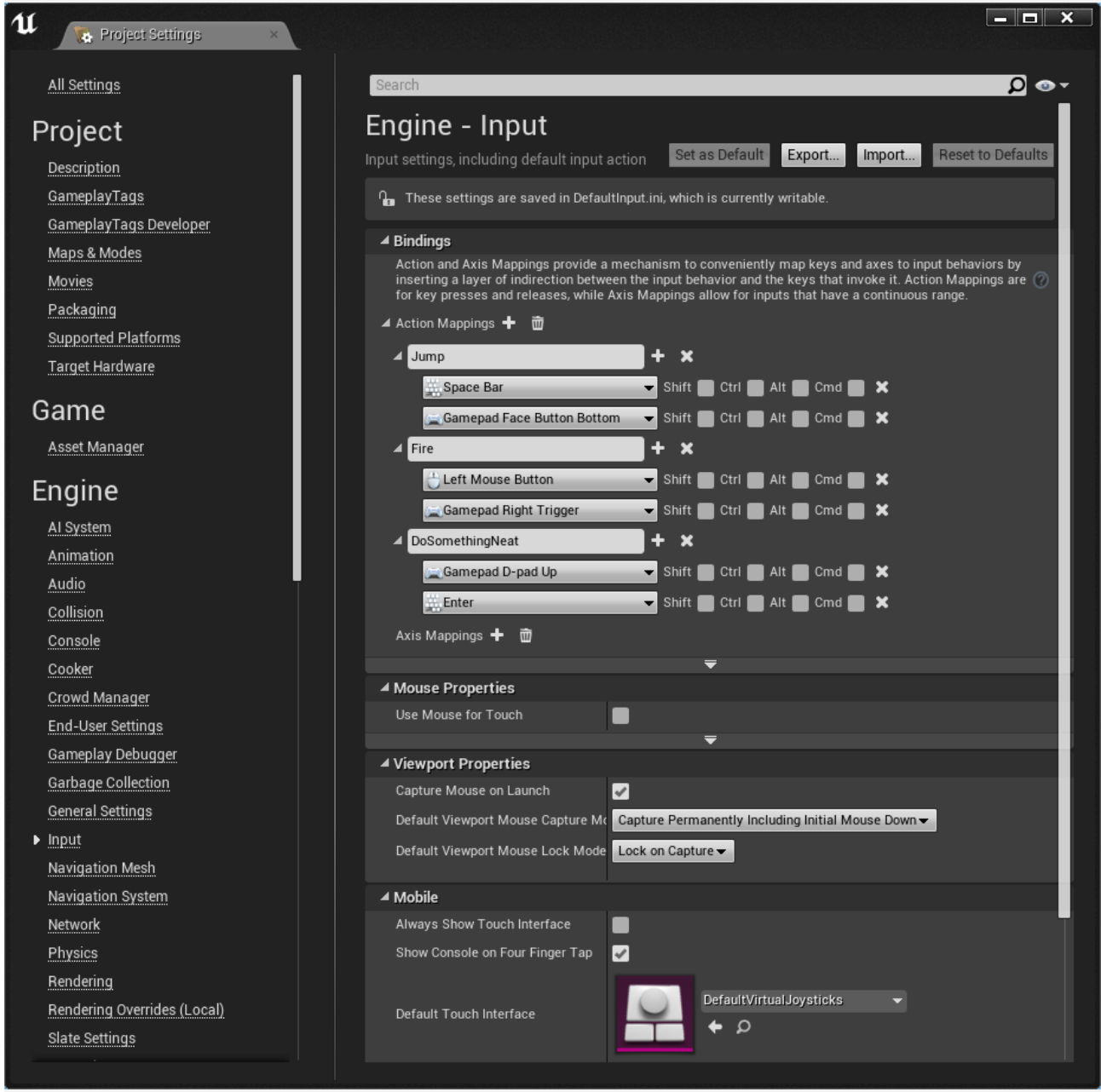
To access the mappings in the Editor, go to Edit > Project Settings > Input.



INPUT MAPPING

Each Action and Axis mapping can have multiple inputs assigned to it. This allows you to write one codebase that handles all input devices uniformly.

Use Action mappings for onetime events, like a button press, and Axis mappings for continuous data, like player movement and rotation.



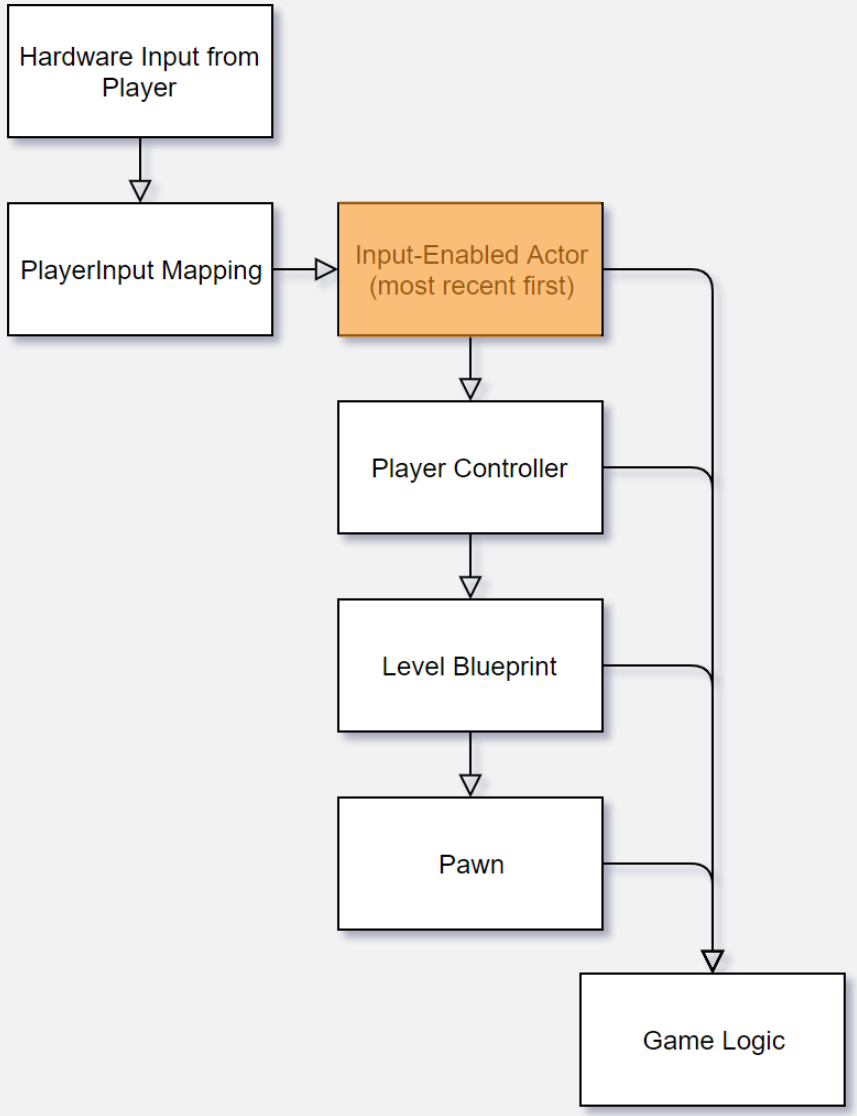
ACTORS

Any Actor in the game can accept input.

Typically in UE4, Actors that receive input are Pawns. If you are intending to control an Actor with player inputs, you should use a Pawn that's being controlled by a Player Controller instead.

An example would be a panel that the player must interact with.

- Because the Actor is at the top of the input stack, it can take over input while the player is using it, then return input to the Player Controller and Pawn.

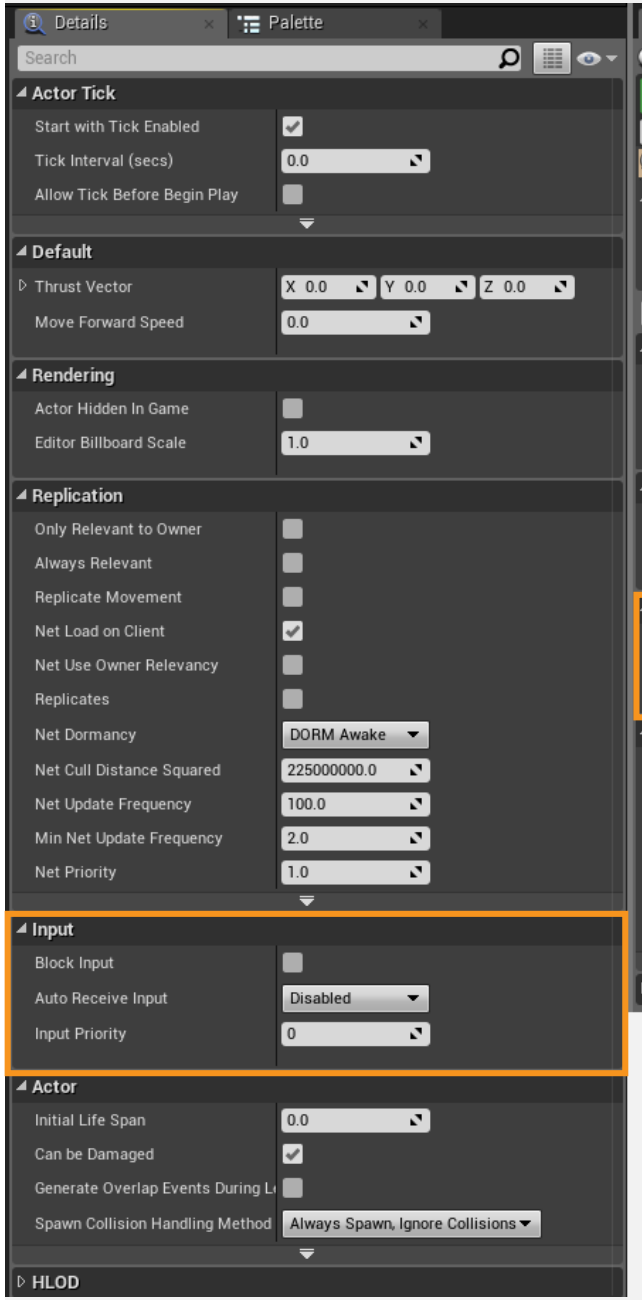


ENABLING INPUT

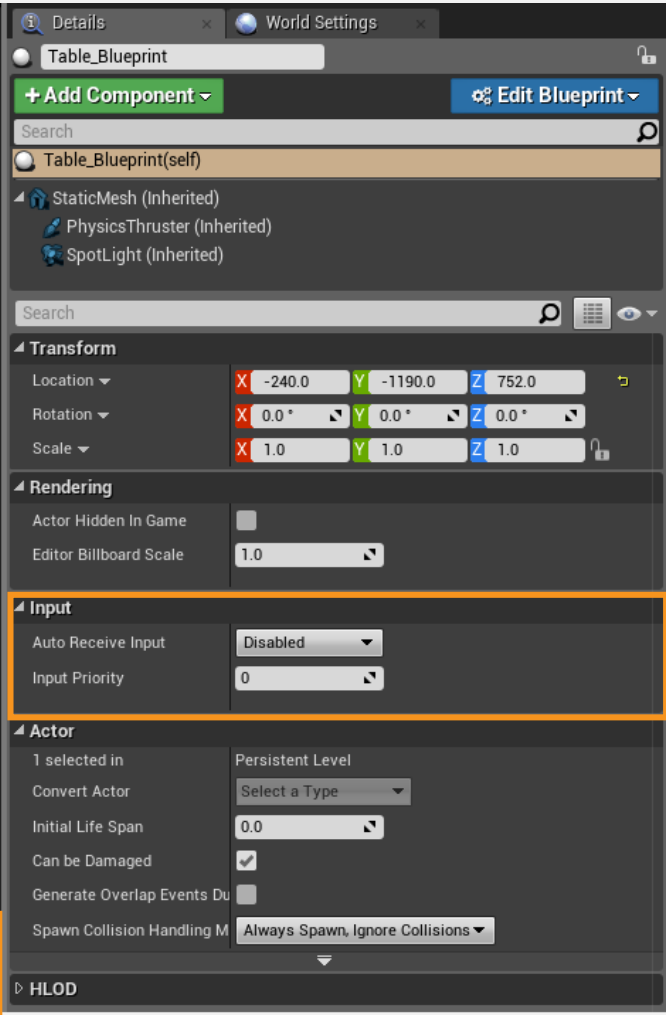
Actors will not receive input unless they have Auto Receive Input set to a specific player. You can set this in the Blueprint's Class Defaults panel.

You can also set it on a per-Actor basis via the Actor's properties in the Details panel. You can toggle Auto Receive Input via code to control the behavior at runtime.

Blueprint class defaults



Actor properties



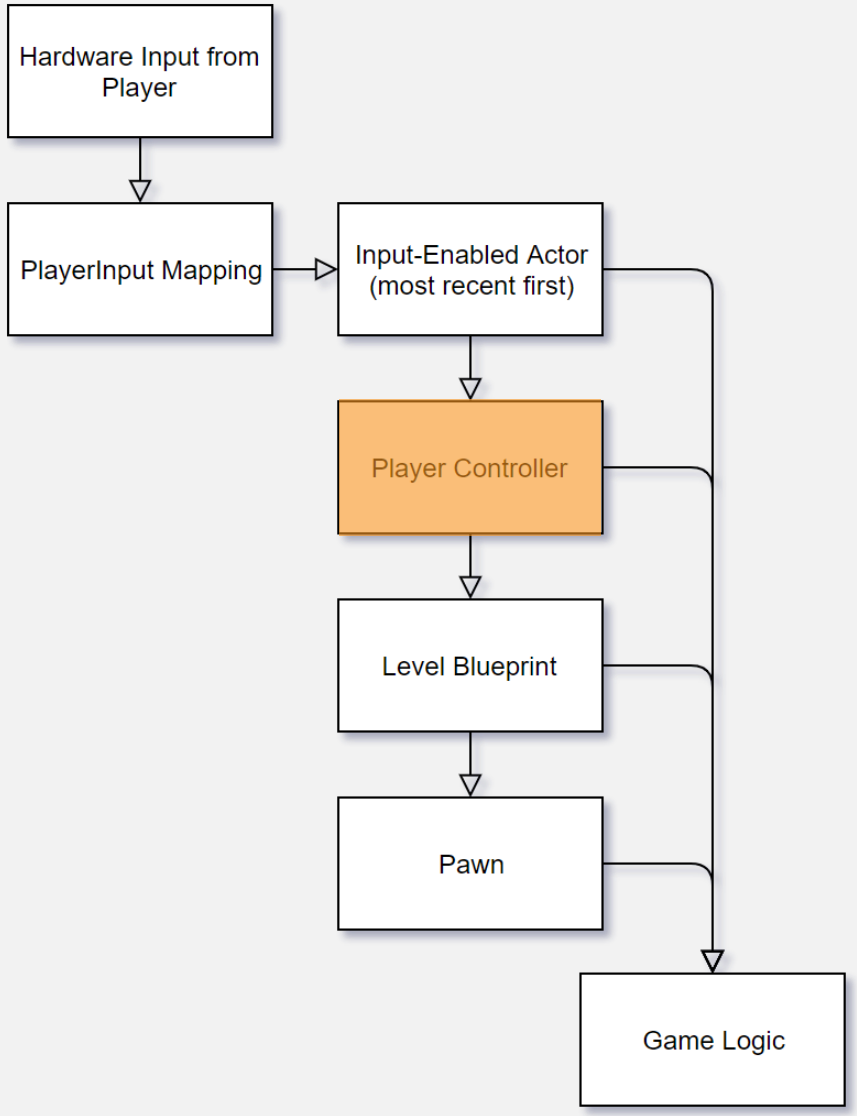
PLAYER CONTROLLER

Your PlayerController class will always accept player inputs passed down from the PlayerInput class.

Depending on the game type you are making, you may have significant input processing in the Player Controller or none.

Put in input code that needs to be used in every part of your game, no matter what Pawn is being controlled, such as

- Pause menu
- Quit button
- Inventory access

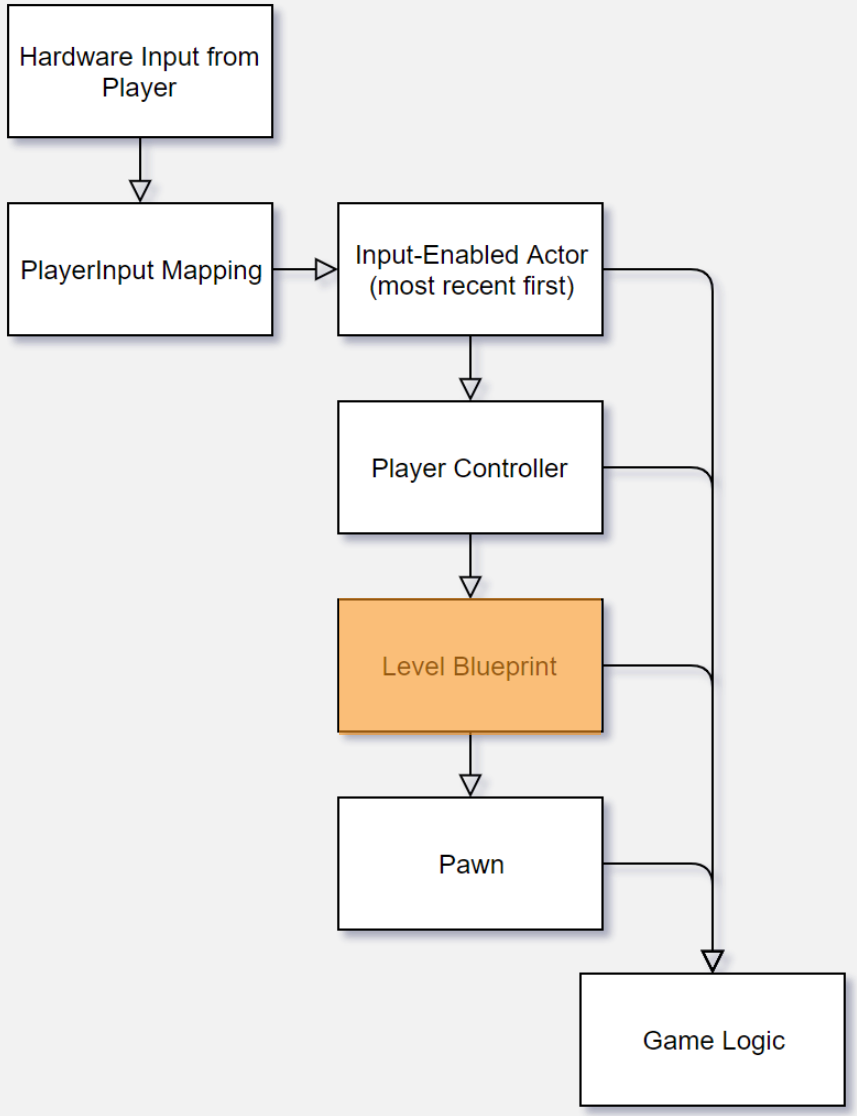


LEVEL BLUEPRINT

The Level Blueprint can accept inputs from the player, which can be useful for testing or creating Level-specific inputs.

Much like Actors receiving inputs, this functionality is intended for non-character input tasks like selecting items in a menu. Avoid putting in input processing that is needed in multiple Levels or is for character movement.

Level Blueprints aren't typically where you'd expect to find input processing. They are usually used only for debugging where you use player input to trigger game events.

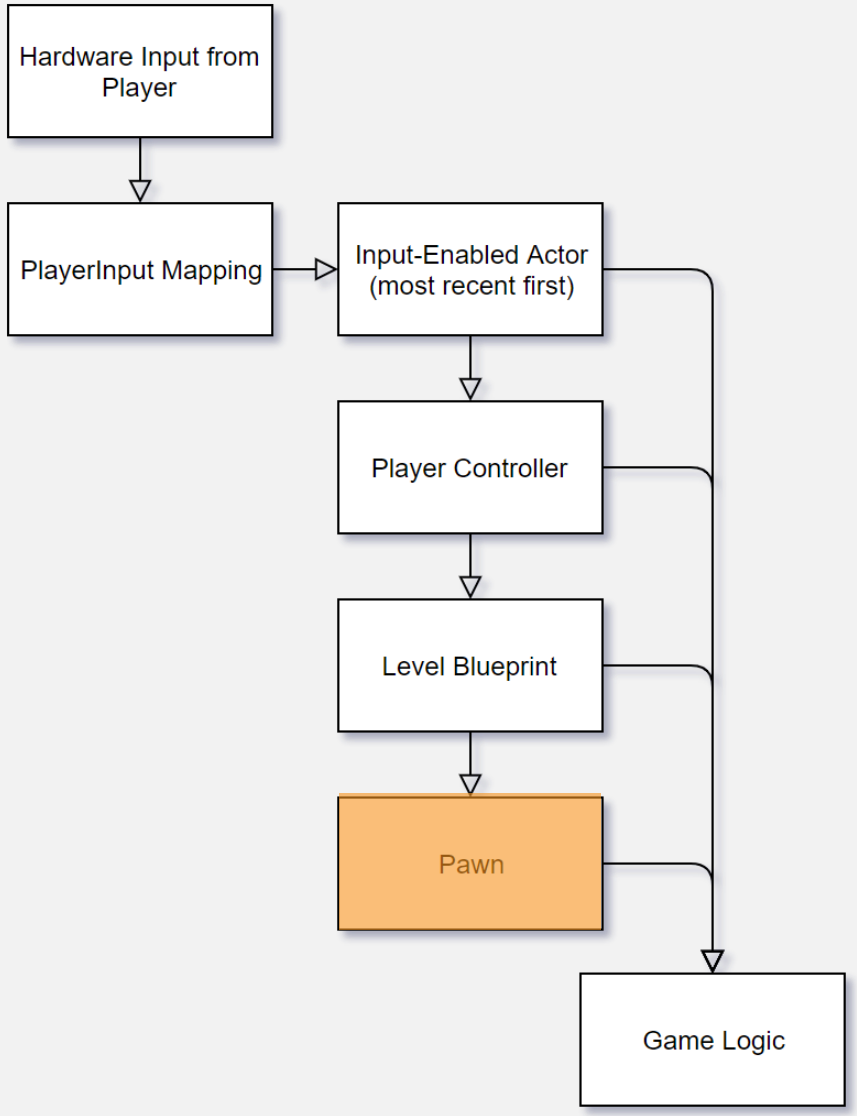


PAWN

The Pawn is where most input that deals with movement is managed.

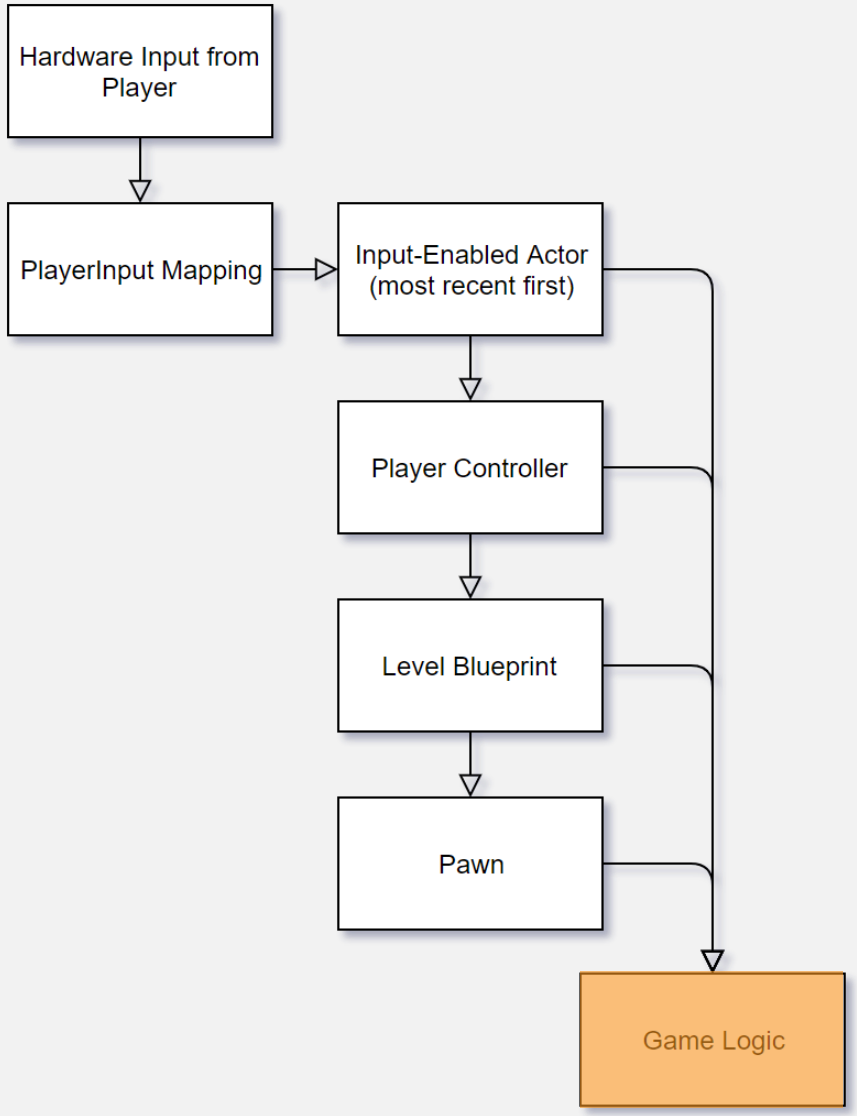
Pawns begin to take player input when they are *possessed* and stop when they are *unpossessed*.

By putting movement code into the Pawn, you can define vastly different movement systems for different Pawn classes.



GAME LOGIC

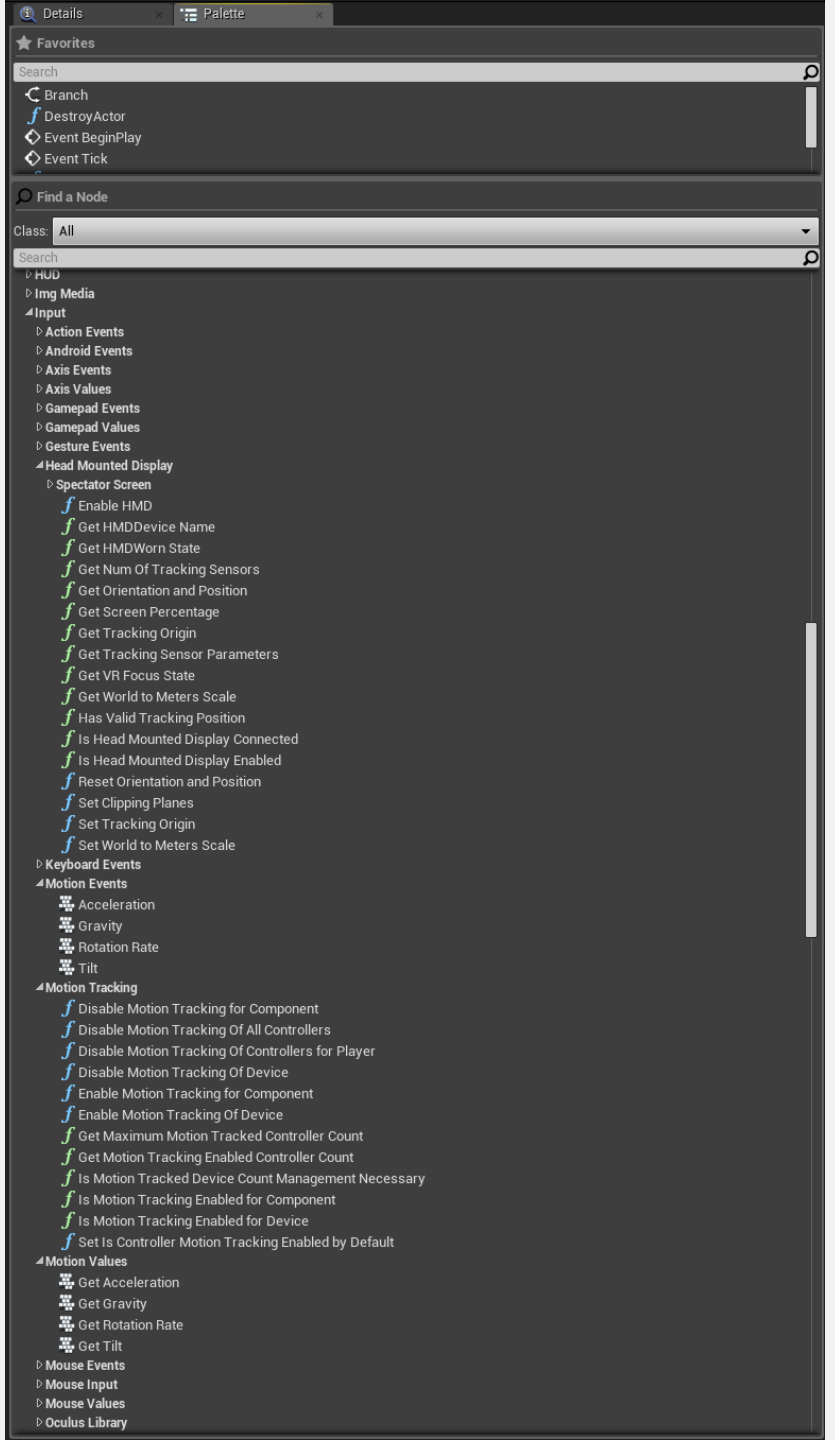
Game logic is simply the code that you write that responds to the various inputs passed by the PlayerInput class into the input stack.



VR INPUT

VR input is slightly different. You have axis and event handlers for the buttons and touch pads on the various VR motion controllers and headsets.

However, to access the spatial data from the motion controllers, you need to use Motion Controller Components.



MOTION CONTROLLER COMPONENTS

Motion Controller Components act as objects in 3D space that you can attach additional components to.

You can set what hand to track using the Hand parameter in the Motion Controller Component's Details panel.

Motion Controller Components use a low-latency update system that updates the controller's position immediately before rendering.

Components attached to the Motion Controller Components will track using the low-latency technique as well.