Welcome to the course! In this lesson, we will take a first look at what the project is about, how it fits into a larger multi-course series, and what topics this first course will cover. The course is led by instructor Daniel Buckley, and the goal is to build a fully functional local multiplayer fighting game inside of Godot, complete with various attacks, abilities, great-looking visuals, audio, and polish.

About the Project

The project we will be developing is a two-player fighting game, similar in style to titles like *Mortal Kombat*. Two players will appear on screen at the same time and face each other in combat, with movement, jumping, attacks, and dodges all available to them.

A key feature of the game is local multiplayer. Both players will share the same keyboard rather than playing on separate devices or over a network. The control scheme is split between the left and right sides of the keyboard:

- Player one uses the WASD keys on the left-hand side of the keyboard.
- Player two uses the arrow keys and other nearby buttons on the right-hand side to perform their movements, attacks, and actions.

A Multi-Course Project

It is important to understand that this course is only the beginning of a much larger project. The full fighting game is being built across four separate courses, each focused on a specific slice of functionality. This allows each topic to be covered in enough depth without overwhelming any single course.

1. **Course 1 (this course):** Developing the basic systems and introducing local multiplayer.
2. **Course 2:** Continuing to build out the core systems, including visuals, audio, and polish needed to complete the base game.
3. **Course 3:** Introducing enemy AI, so that a single player can fight against a computer-controlled fighter with tweakable settings.

4. **Course 4:** Implementing online multiplayer functionality.

What This Course Covers

This first course focuses on creating the first half of the game, with an emphasis on setting up a state-based fighter. A state-based approach means that the fighter's behavior is broken down into discrete states (such as standing, jumping, or attacking), which makes the logic easier to manage and extend as the project grows.

Specifically, by the end of this course, you will have worked through the following topics:

- Implementing fighter abilities, including jumping, light attacks, heavy attacks, and dodging.
- Setting up fighter animations using Godot's AnimationTree node.
- Detecting and managing inputs for local multiplayer on a single shared keyboard.
- Implementing an interesting mechanic known as input buffering, which lets the game register recent inputs over a short window of frames.
- Setting up fighter visuals, including different outfits for Player 1 and Player 2, as well as a hit flash effect that flashes the fighter whenever they take damage.

Course Requirements

Before starting this course, you should have an intermediate understanding of Godot and GDScript. This is not a beginner-friendly course, because we will be diving into more advanced topics, such as state machines and going deeper into Godot's animation system than a first-time tutorial would. A moderate level of comfort using Godot and writing GDScript is recommended before embarking on this course.

About Zenva

Zenva is an online learning academy with over a million students, featuring a wide range of courses for people who are just starting out as well as for those who simply want to learn something new. The courses are designed to be versatile, allowing you to learn in whatever way works best for you. You can choose to watch the online video tutorials, read the included lesson summaries, or follow along with the instructor using the included course files.

With the project scope, course structure, and requirements now clear, it is time to move on and get started with the first lesson.

Which version of Godot should you use for this course?

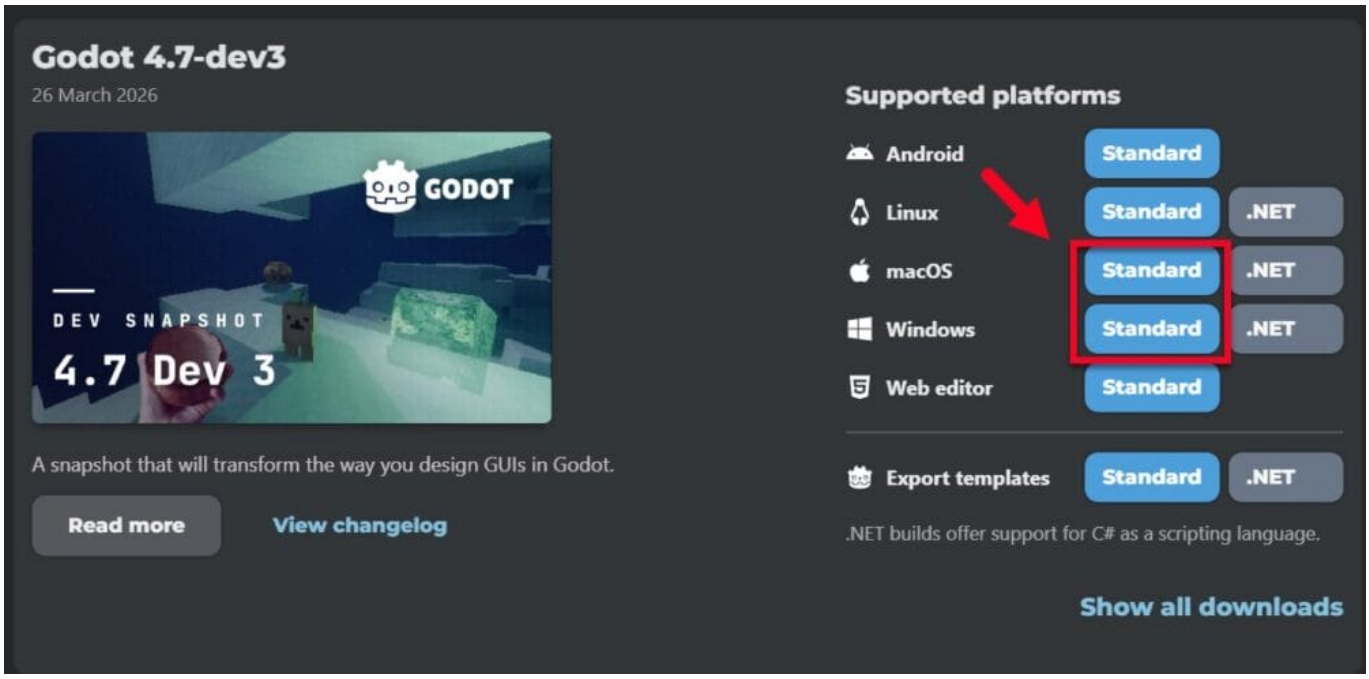
Technology changes rapidly, so to make sure you have an optimal learning experience, we recommend using **Godot 4.7** for this course.

Please make sure NOT to use newer versions of the software than what we recommend, as the course material provided might not work as expected.

How to Install Version 4.7

You can download the development version of Godot 4.7 by heading to the Godot archive download page here: https://godotengine.org/download/archive/4.7-dev3/

Once on the download page, click the Standard option that matches your operating system**.** This will download the Godot engine to your local computer. From there, unzip the file and simply click on the application launcher – no further installation steps are required!



Before jumping into Godot and writing any code, it helps to step back and understand what we are building and how the different pieces will fit together. This lesson walks through the design of the fighting game, the features it will ship with, and the high-level structure of the fighter character that most of the course will revolve around.

Project scope and course series

The project is a two-player fighting game, similar in feel to something like Mortal Kombat: one fighter on the left, another on the right, trading blows until one of them is defeated. This particular course is the first in a planned multi-course series, and each course expands on the previous one.

- **Course 1 (this course):** the main game systems and local multiplayer.
- **Course 2:** enemy AI, adding a computer-controlled opponent.
- **Course 3:** online multiplayer over the network.

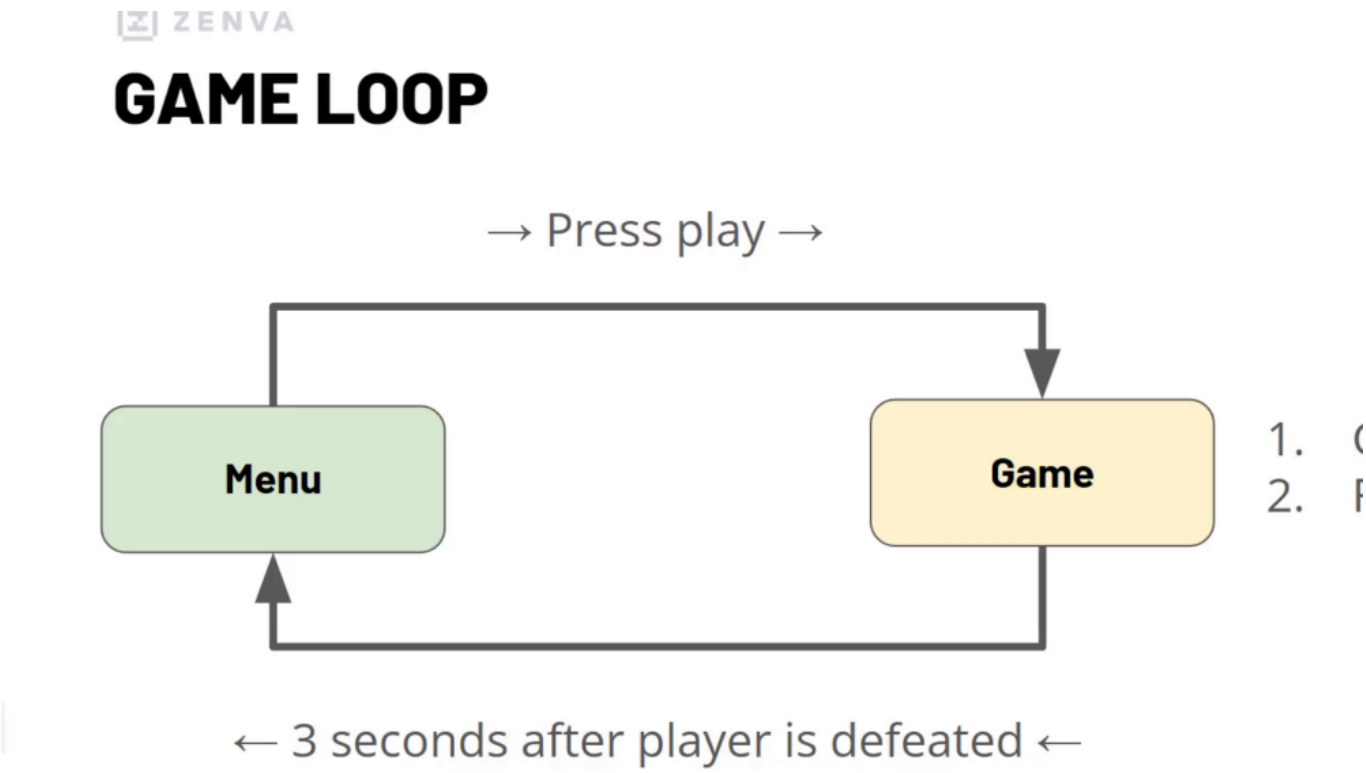
Because each course builds on the last, the focus here is on getting the core fighter, game loop, and local multiplayer right so that later courses have a solid foundation to expand upon.

Local multiplayer controls

Local multiplayer means a 1v1 match where both players share a single keyboard. Player 1 sits on the left-hand side of the keyboard and uses the W, A, S, and D keys along with some additional nearby keys, while Player 2 sits on the right-hand side and uses the arrow keys together with their own set of attack keys.

The game loop

The overall flow of the game is intentionally simple. The player begins in a menu scene, and pressing the play button launches into the game scene. Once inside the game scene, a short countdown plays before the fighters are allowed to move, and then the match continues until one of them is defeated. A few seconds after a defeat, the game automatically returns to the menu, ready to start the loop again.



Fighter actions

Each fighter has the same core set of actions, driven by that player's half of the keyboard. The available moves are:

- **Move left and right** along the arena.
- **Jump** to change vertical position.
- **Dash** quickly in a direction to avoid incoming attacks.
- **Light attack**, which is fast but deals less damage.
- **Heavy attack**, which is slower but deals more damage.

Because the two attack types have different speeds and damage values, there is a design element to tuning those numbers so the game feels fair and fun. Getting those values to feel good is part of the iterative work that goes alongside writing the code.

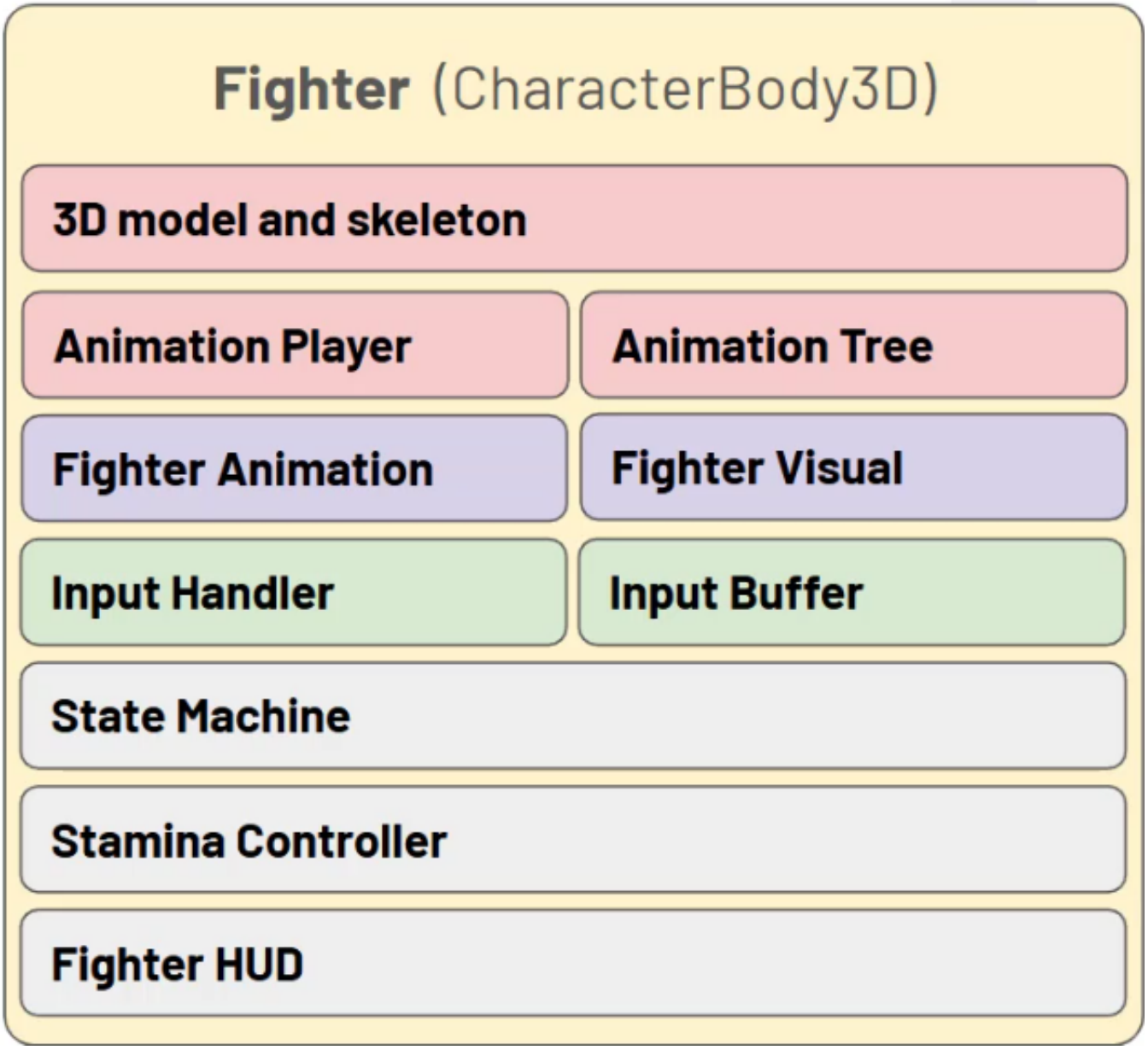
Health and stamina

Every fighter carries two stats that drive the flow of a match: health and stamina. Health depletes whenever the fighter is hit, and once it reaches zero, the fighter is defeated. That defeat triggers an end-of-game event, puts text on screen announcing that either Player 1 or Player 2 has won, and then kicks both players back to the menu.

Stamina works in the opposite direction. It regenerates over time on its own, but is consumed whenever the fighter performs an action that should not be spammed, such as an attack or a dash. If a player keeps throwing attacks nonstop, their stamina will run out, and they will need to back off and let it recover before they can keep pressuring their opponent. Both stats are presented as a stacked pair of UI bars at the top of the screen, with Player 1’s bars in the top-left corner and Player 2’s in the top-right.

Structure of the fighter

Most of this course will be spent building the fighter itself, which is composed of a surprisingly large number of nodes and systems working together. Rather than a single monolithic script, the fighter is broken into small, focused pieces that each handle one responsibility.



At the top is a Fighter node based on CharacterBody3D, and underneath it live components such as the 3D model and skeleton, an Animation Player and Animation Tree, a Fighter Animation helper, a Fighter Visual, an Input Handler, an Input Buffer, a State Machine, a Stamina Controller, and a Fighter HUD. If that list feels overwhelming at this stage, do not worry: a dedicated lesson early in the course walks through what each of these nodes is responsible for, and the implementation is then built up slowly so that each piece is understood in context before the next one is added.

Environment and scene

Alongside the fighter itself, the course sets up an immersive arena that reflects the tone of the game. Pre-made 3D models and textures are used to dress the scene, resulting in a stylized arena with sparse trees, crates, and character models on a tiled platform beneath a dramatic red-orange sky.

The exact look of the environment is flexible and can be tweaked freely, but following along tightly with the course is easiest using the included course files. Those downloadable assets will be introduced in the next lesson.

Other features

On top of the core fighter and environment, the course weaves in a handful of smaller systems that add polish and round the game out. These include:

- An **audio pooling system** for efficiently managing and playing sound effects.
- A **countdown at the start of each match**, with animated text and a countdown voiceover.
- A **menu scene** that players click “Play” from to begin a match and return to once the game ends.
- A **hit flash** on the fighter whenever they take damage, combined with a **screen shake** effect and matching sound effects for impact.
- A **dynamic camera** that tracks the fighters horizontally and zooms in or out as they move, keeping both of them in frame at all times.

The details of how each of these systems work will be explained as the course progresses, with a few more slide-based lessons planned along the way.

With a fresh Godot project in place, the next step is to lay down the foundations for the fighting game. In this lesson we import the provided asset pack, take a quick tour of what is inside each folder, create the main game scene, and set up a handful of additional project folders that will hold scripts, materials, and custom resources later in the course.

Downloading and extracting the asset pack

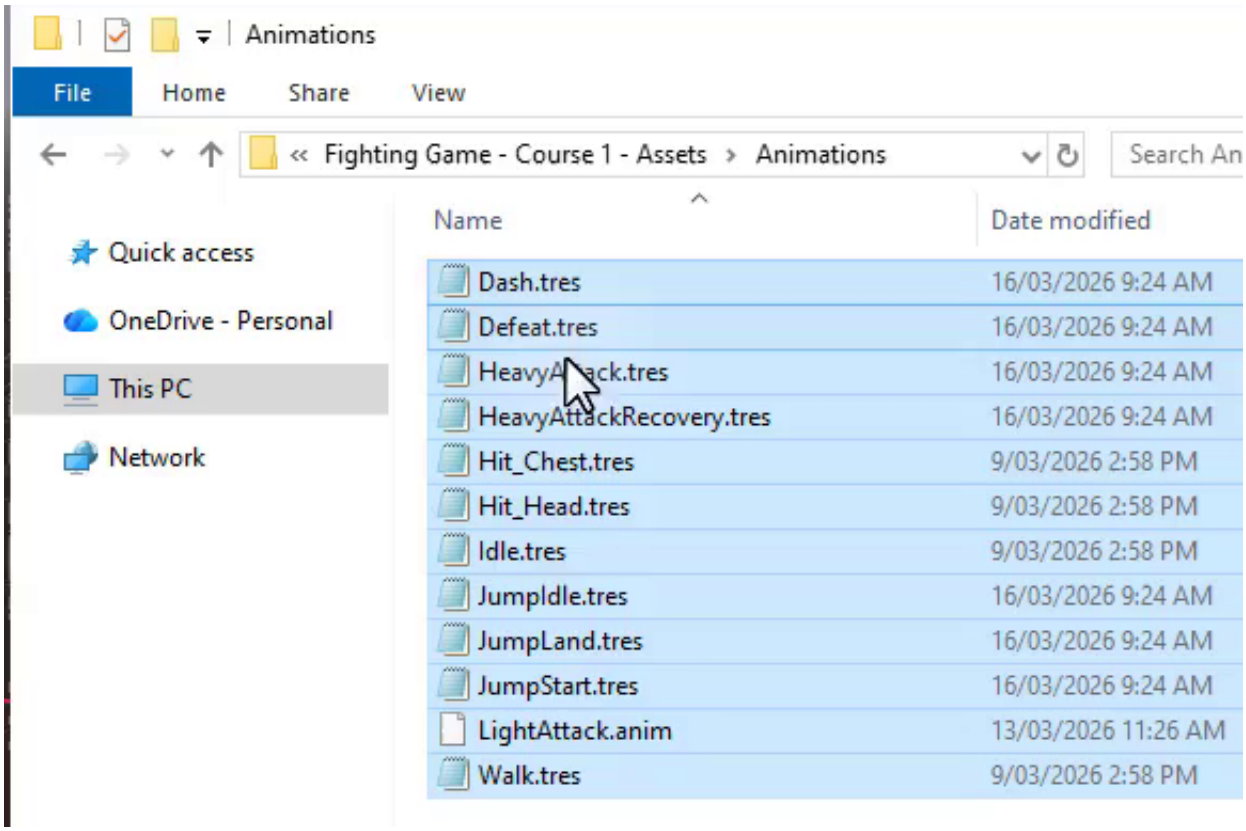
Before opening Godot, head to the **Course Files** tab and download the included assets zip file. Extract it somewhere on your computer. Inside the extracted folder you should see five subfolders that together cover everything the game needs to look, sound, and feel complete:

- **Animations** — character animations for every action in the game.
- **Audio** — combat sounds, music, and voice overs.
- **Fonts** — a custom Google font used throughout the UI.
- **Models** — 3D models for the environment, characters, and props.
- **Textures** — particle and skybox textures.

A quick tour of what is inside

It is worth opening each folder and taking a look at its contents before importing, so you know what is available as the course progresses.

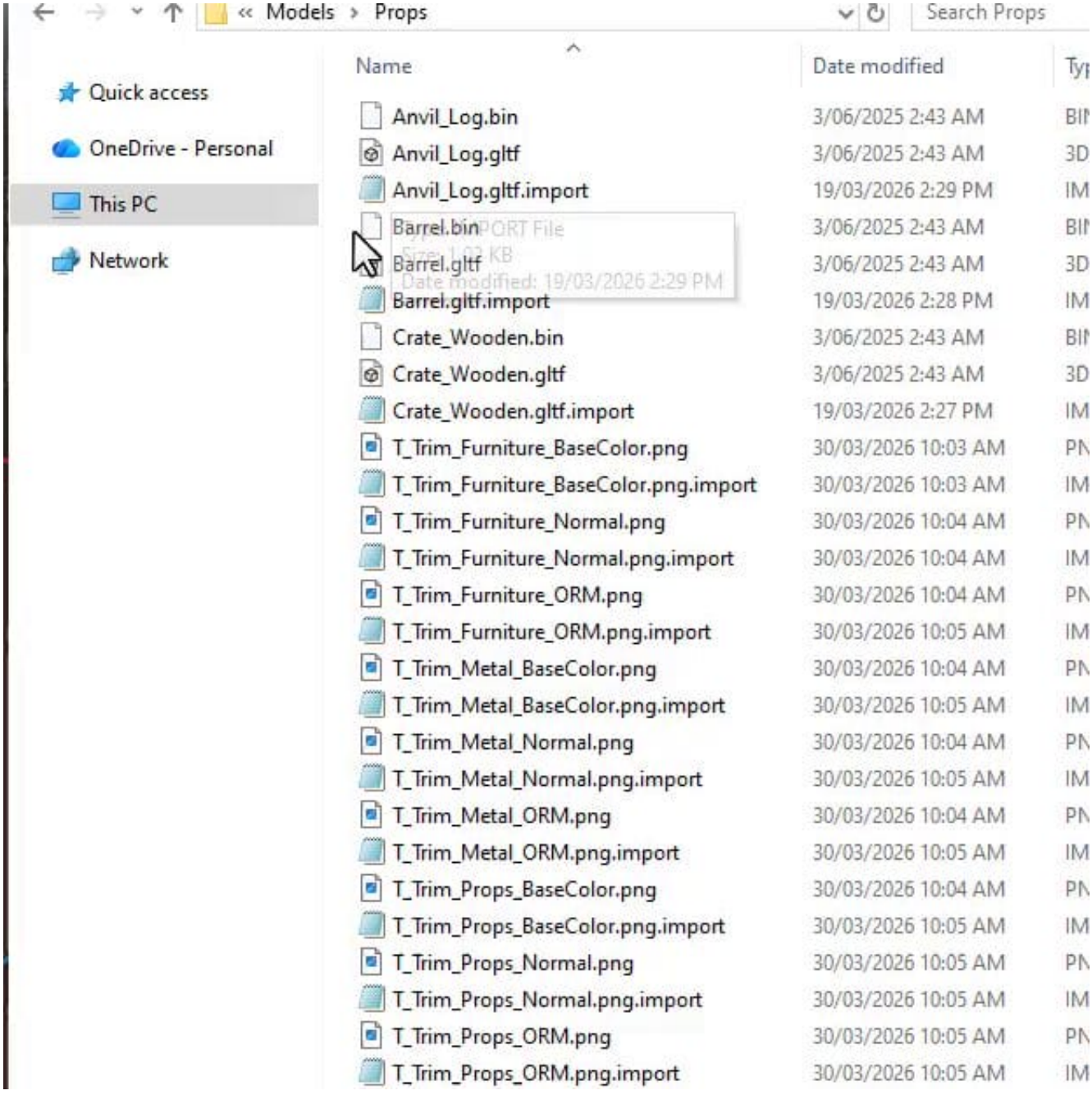
The **Animations** folder contains a large list of animations for the fighter character — idle, walking, jumping, dashing, light and heavy attacks, hit reactions, and the defeat animation. These cover every state the fighter will move through during gameplay.



The **Audio** folder provides combat sounds for attacks and impacts, menu music and battle music, and voice overs used for countdowns and for announcing the winner or loser at the end of a match. The **Fonts** folder adds a custom Google font used for in-game text.

The **Models** folder is organised into several subfolders that together build the game world:

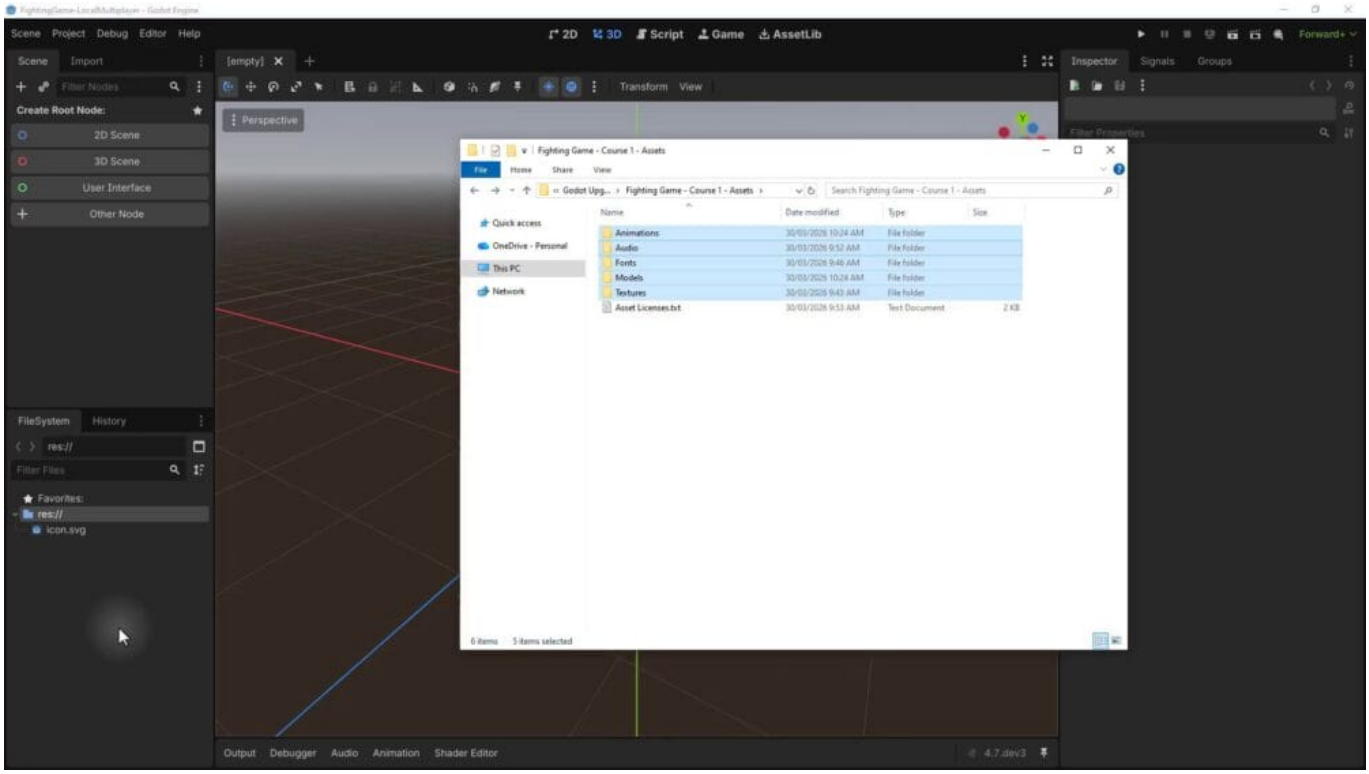
- **Castle** — the environment the players fight on.
- **FighterHead** — the head of the fighter character.
- **Outfits** — the character outfit along with several colour, texture, and material variants.
- **Mountains** — background scenery placed behind the arena.
- **Nature** — natural assets such as trees to dress the scene.
- **Props** — small environmental props including an anvil, a barrel, and a wooden crate.



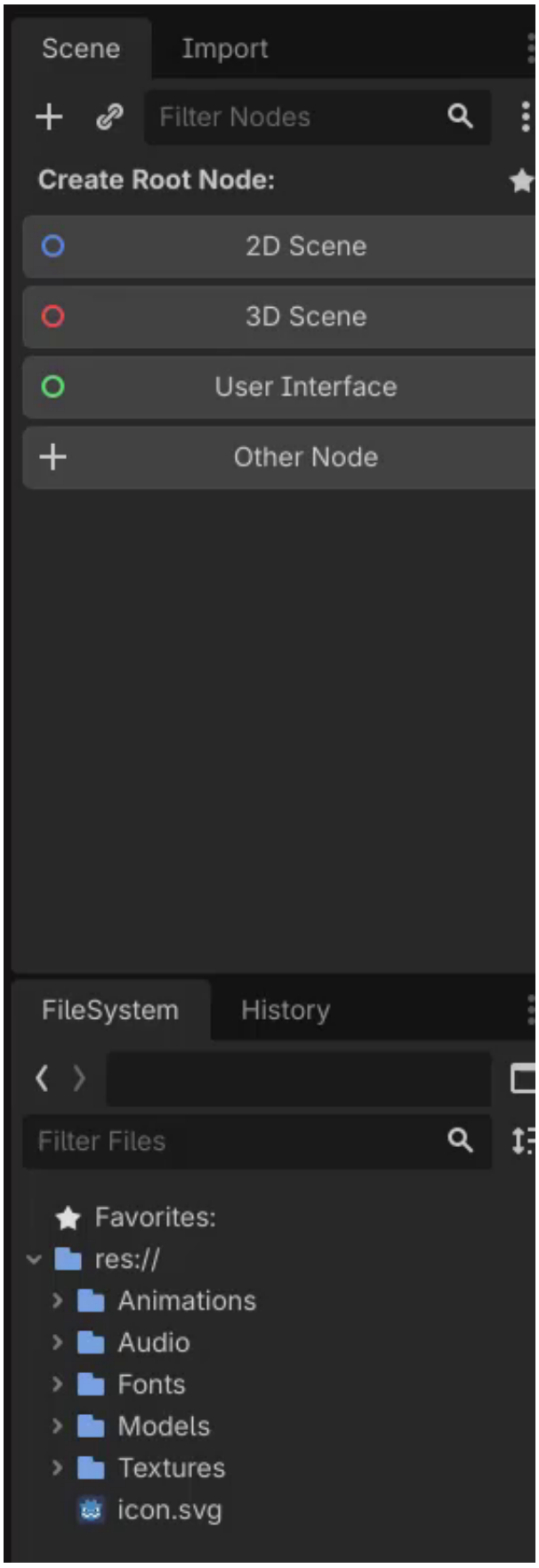
Finally, the **Textures** folder supplies a particle texture used for visual effects and a skybox texture for the background.

Importing the assets into Godot

With Godot open on the new project, importing these assets is straightforward. In the file explorer, select all five folders — Animations, Audio, Fonts, Models, and Textures — and drag them directly onto the Godot **FileSystem** dock. Godot will copy the folders into the project and begin importing each file automatically.



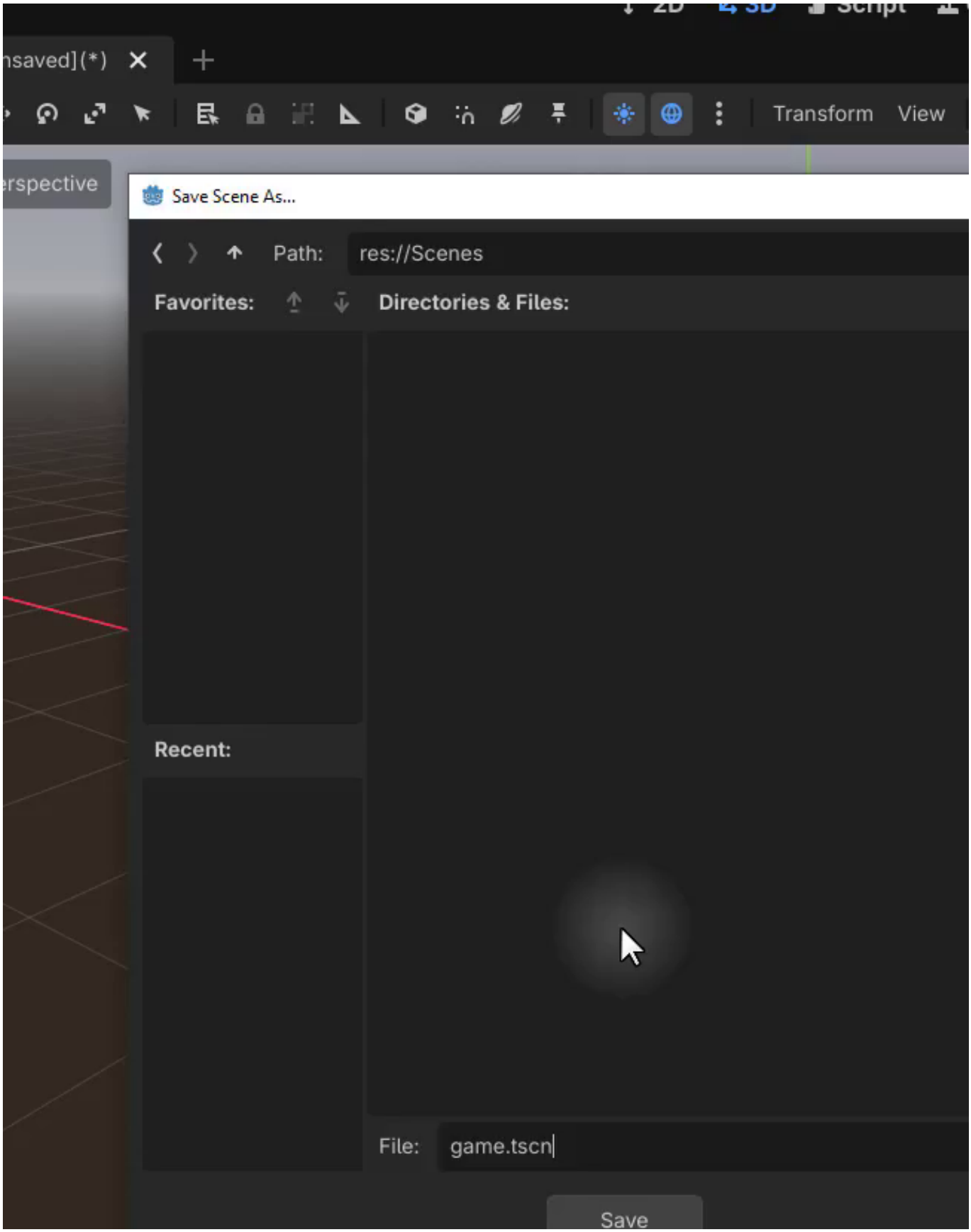
Once the import finishes, glance at the **Output** panel at the bottom of the editor. Plain messages are fine, but any red error lines usually point to a problem with the project setup. A very common cause of import errors is nesting the folders one level too deep, so double-check that *Animations*, *Audio*, *Fonts*, *Models*, and *Textures* sit directly at the project root (res://) and not inside a wrapper folder — a lot of the resources in the pack rely on this exact layout to resolve their references.



Creating the game scene

With the assets in place, the next step is to create the main game scene — the scene the two characters will load into and fight in. Start by creating a new folder in the FileSystem dock called **Scenes**. Inside it, create a brand new 3D scene and rename the root node to **Main**.

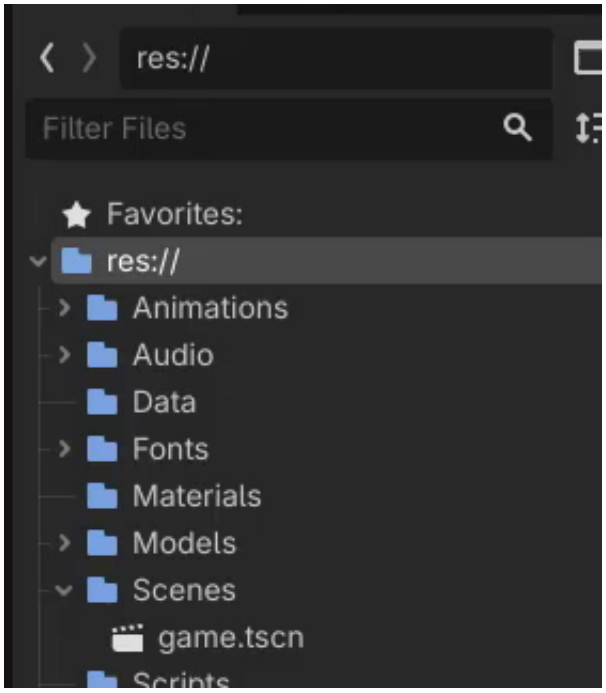
Save the scene inside the Scenes folder as **game.tscn**. This file will act as the entry point for gameplay and will be built out over the following lessons.



Creating the remaining project folders

A few more folders round out the project structure. Each has a clear purpose that will become relevant in later lessons:

- **Scripts** — holds every game script, including the fighter, input handlers, and state machine.
- **Materials** — holds custom materials. Only one will be created in this course: a hit material that flashes the character white for a few frames when they take damage.
- **Data** — holds custom resources. In this project it will contain a single resource that tracks the stamina cost of player actions such as jumping, attacking, and dashing.



What comes next

With the assets imported, the game scene created, and the folder structure in place, the project is ready for development. The next lesson covers setting up the input actions for both player one and player two, since this is a local multiplayer game where player one uses the left side of the keyboard and player two uses the right side. Those inputs will then be used later on when building out the individual fighters.

In this lesson, we set up the input actions that both players will use to control their fighters. Since our project is a local multiplayer game, the two players will share a single keyboard, so we need a clean way to define independent control schemes for each one and distinguish them later in code.

Planning the control scheme

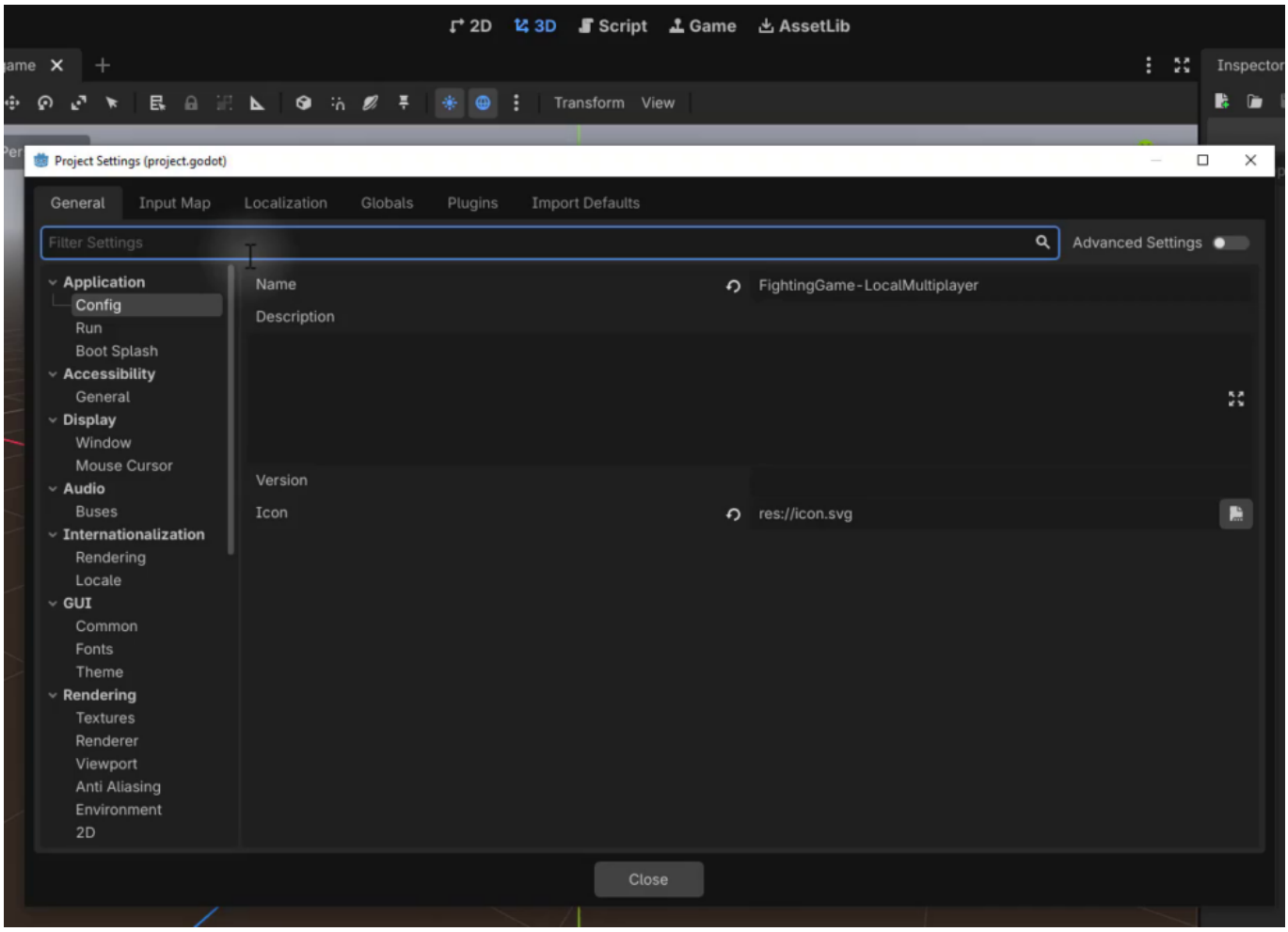
Before touching the editor, it helps to decide how each player will interact with the keyboard. The idea is to split the keyboard in half so that both players can sit comfortably in front of the same machine:

- **Player 1** uses the left-hand side of the keyboard, with A and D for movement and nearby keys for jumping, dashing, and attacking.
- **Player 2** uses the arrow keys for movement, with additional keys on the right-hand side for jumping, dashing, and attacking.

These exact key choices are entirely up to you, but keeping each player's hands in their own zone makes the game far more playable on a shared keyboard.

Opening the Input Map

All input actions in Godot are configured from the Project Settings window. Open the top menu and go to **Project > Project Settings**, then switch to the **Input Map** tab. This is where we will register every action both players can perform.

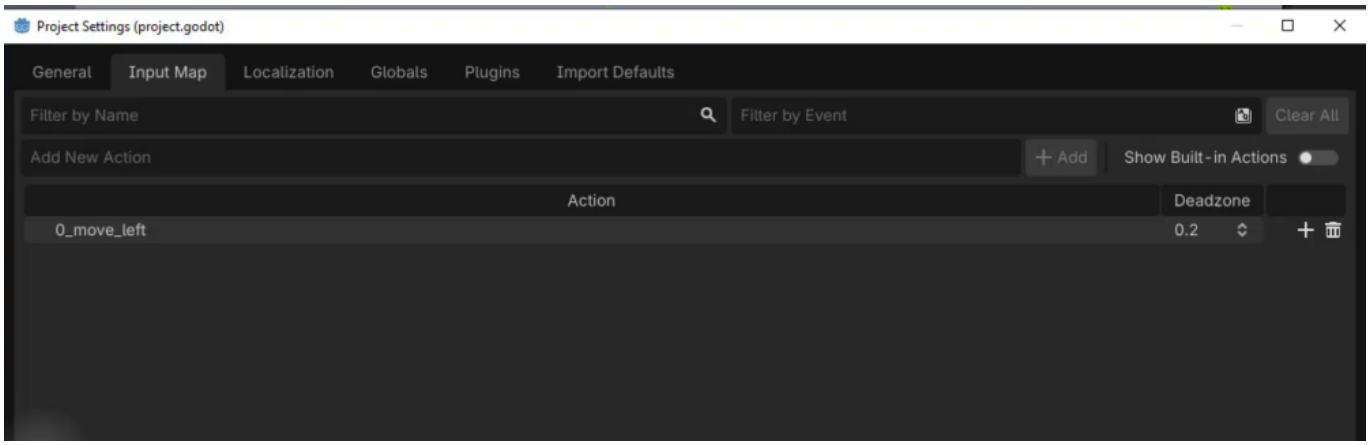


A naming convention for two players

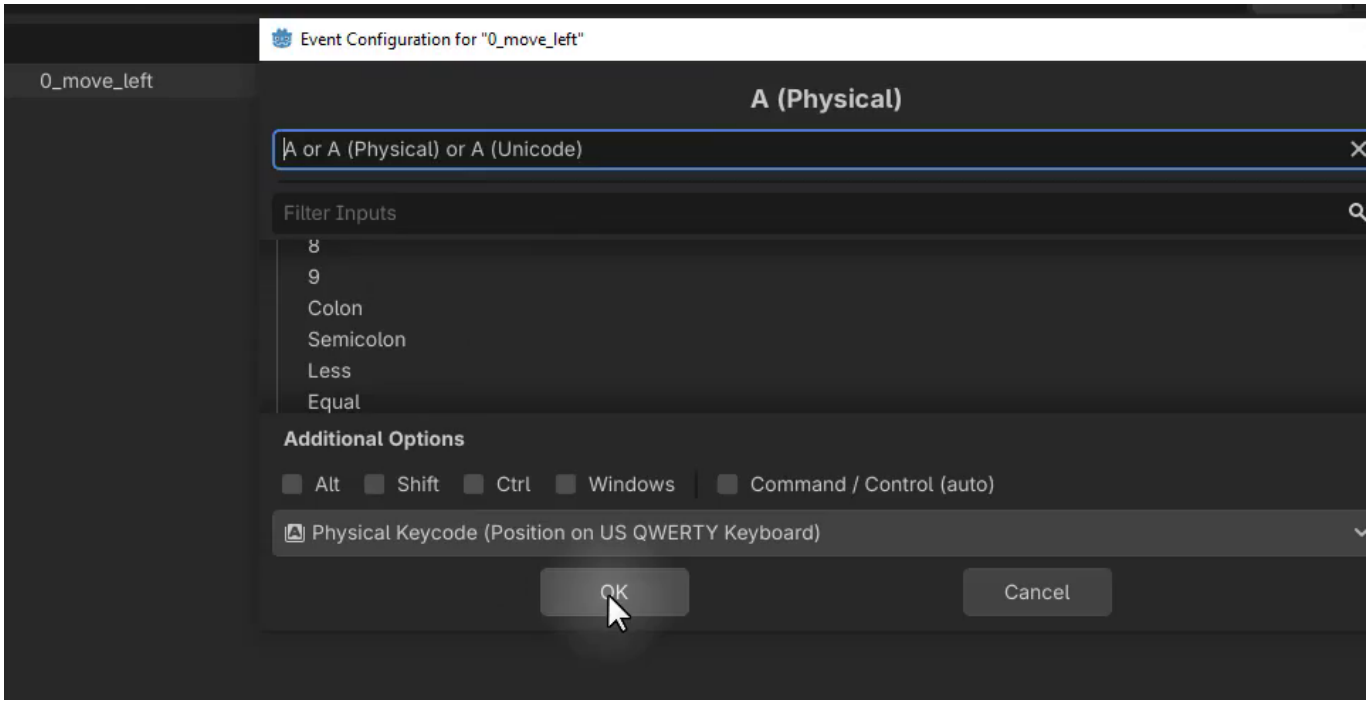
Godot's Input Map does not support grouping actions into categories, so we need another way to tell Player 1's inputs apart from Player 2's. The approach used in this project is to prefix every action name with the player's index: **0_** for Player 1 and **1_** for Player 2. Later on, the code will be able to combine a player's ID with a short action name (for example, *jump*) to build the full action name (*0_jump* or *1_jump*) and look up the correct input for that specific player.

Adding the movement actions

Start by creating the first action. In the *Add New Action* field at the top of the Input Map, type **0_move_left** and press Enter. The new action appears in the list with a default deadzone of 0.2.



Next, bind a physical key to this action. Click the plus icon next to **0_move_left** to open the Event Configuration dialog and press the **A** key. The dialog shows the key as *A (Physical)*, meaning Godot identifies it by its position on a US QWERTY keyboard rather than by the character it produces — this keeps the layout consistent across different keyboard locales.

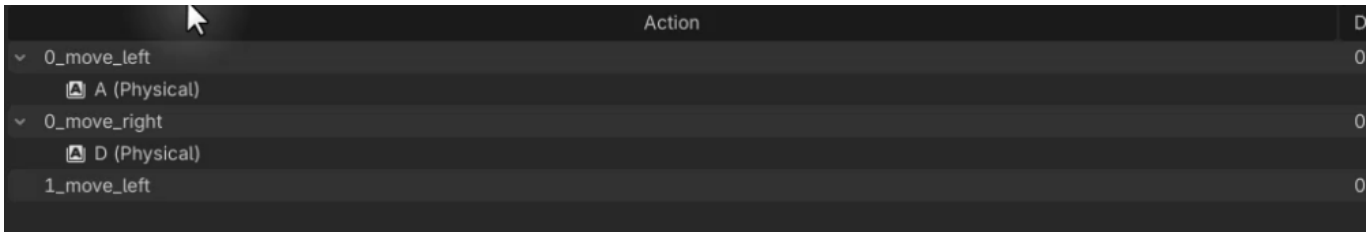


Repeat the same process for the remaining movement actions:

- **0_move_right** bound to the **D** key (Player 1).

- **1_move_left** bound to the **Left Arrow** key (Player 2).
- **1_move_right** bound to the **Right Arrow** key (Player 2).

Once these are in, both players have their own independent move actions living side by side in the same Input Map, differentiated only by their numeric prefix.

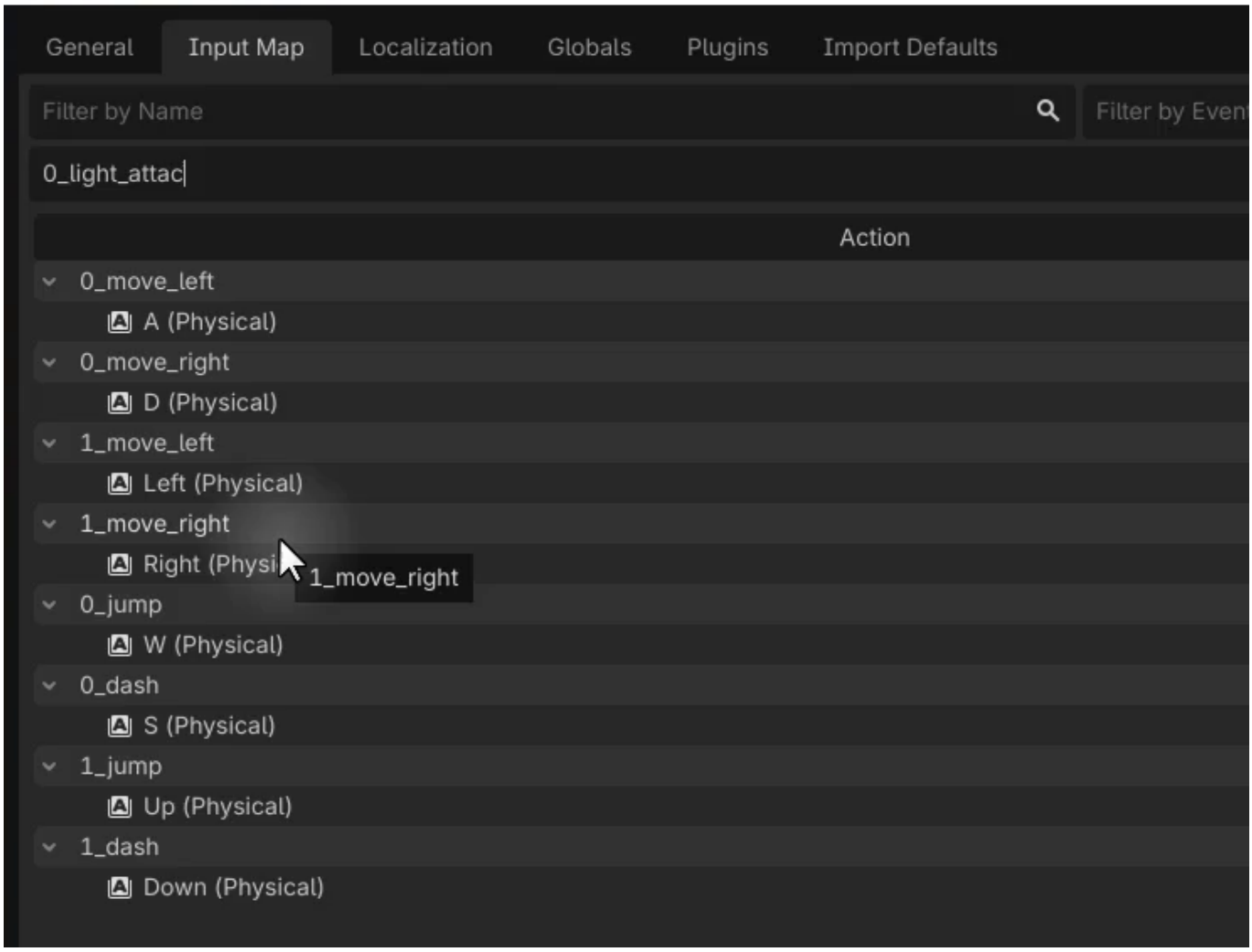


Adding jump and dash

Movement is not the only thing our fighters will do — they also need to jump and dash. Create four more actions following the same convention: **0_jump**, **0_dash**, **1_jump**, and **1_dash**. Then assign keys to each one:

- **0_jump** uses **W**, so Player 1 presses upward to jump.
- **0_dash** uses **S**, so Player 1 presses downward to dash.
- **1_jump** uses the **Up Arrow** for Player 2.
- **1_dash** uses the **Down Arrow** for Player 2.

With these assignments in place, both players now have a full set of movement, jump, and dash actions bound to physical keys on their respective sides of the keyboard.



Adding light and heavy attacks

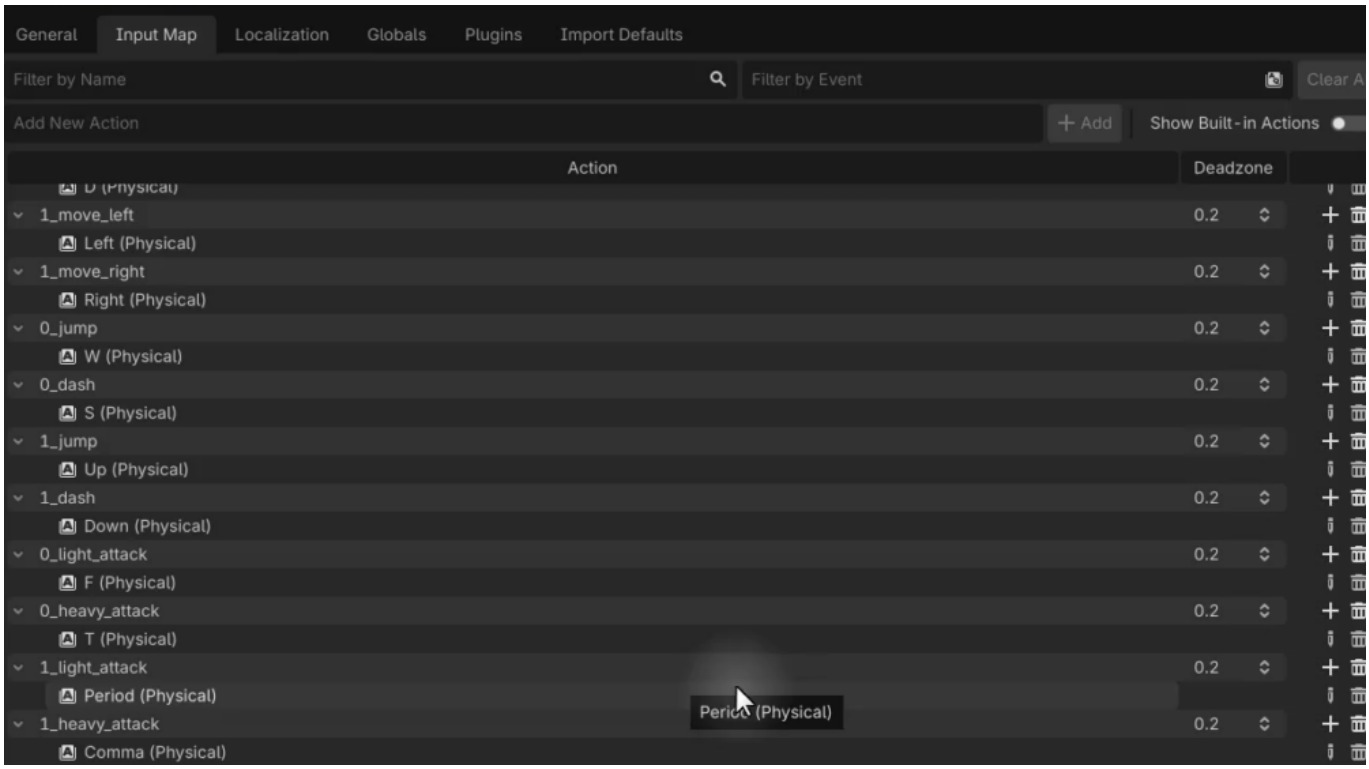
The final pieces of the control scheme are the two attacks each fighter will have: a light attack and a heavy attack. Create four more actions — **0_light_attack**, **0_heavy_attack**, **1_light_attack**, and **1_heavy_attack** — and bind them as follows:

- **0_light_attack** uses **F**, keeping Player 1's attack fingers near their movement keys.
- **0_heavy_attack** uses **T**, just above F for Player 1.
- **1_light_attack** uses the **Period (.)** key for Player 2.
- **1_heavy_attack** uses the **Comma (,)** key for Player 2.

Again, these specific choices are only a suggestion. Feel free to tweak them to whatever feels natural for your keyboard and your testers — the important part is that each player's attack keys are close to their movement keys.

The complete input list

With everything in place, the Input Map now contains twelve actions in total: six for Player 1 and six for Player 2, each with a clear numeric prefix identifying which player it belongs to.



Because we are building a local multiplayer game, there is no need for any engine hacks to share a single control scheme between two players. Instead, we simply maintain two parallel sets of input actions and rely on the numeric prefix to tell them apart in code later on.

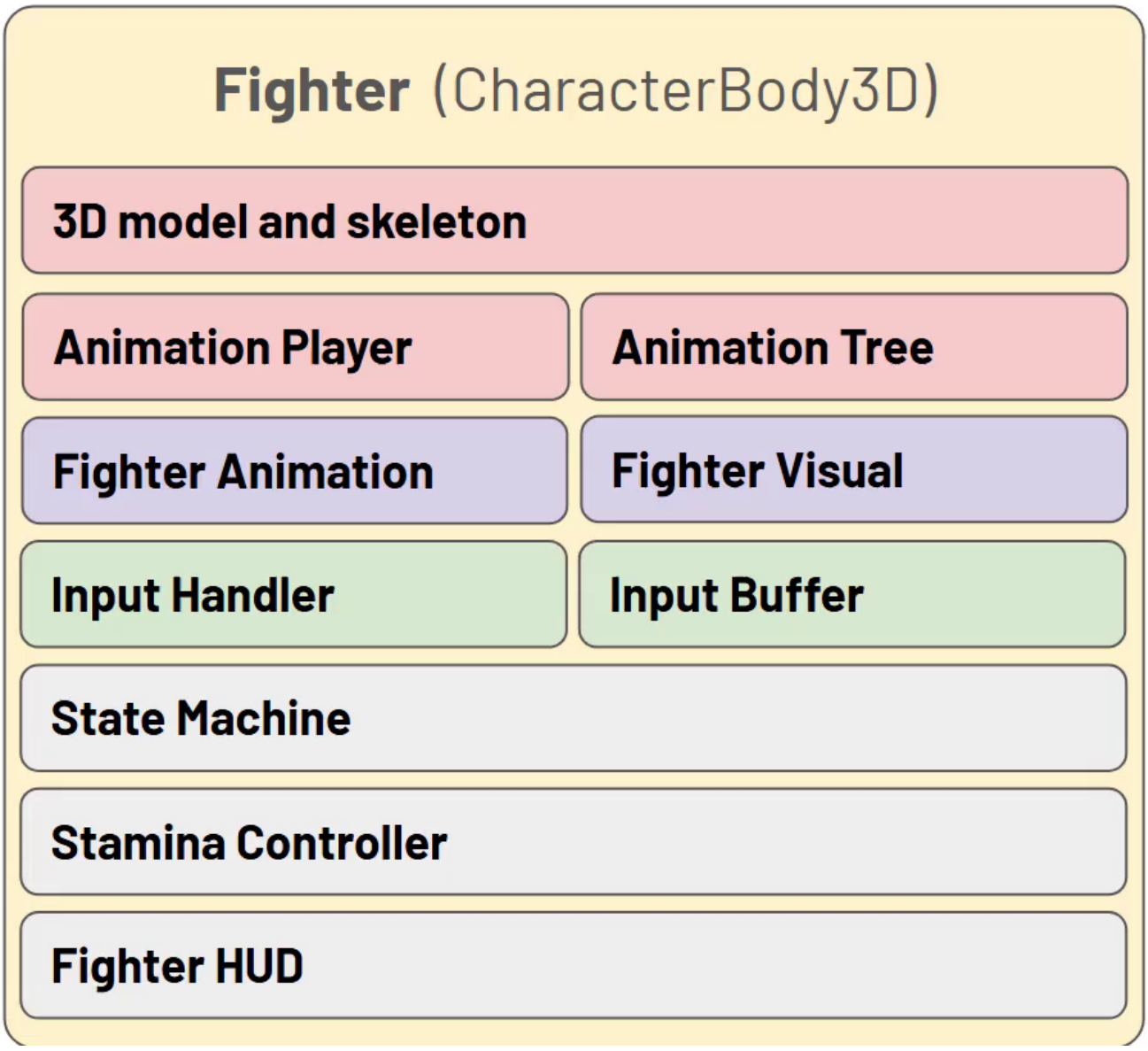
What's next

In the next lesson we will start looking at the structure of our fighting character. Fighters are made up of quite a few nodes, so it is worth getting a solid understanding of how they fit together before we begin building one in the editor.

In this lesson, we step outside of Godot for a moment to look at the fighter character that each player will control in our game. The fighter is the heart of the project: it is a character that can move around, jump, dash, attack, deal damage, and spend stamina to do so. To support all of these behaviors, the fighter scene is made up of a number of specialized nodes, each responsible for a specific part of the character's functionality. Before jumping back into Godot to build it, it is important to understand what each of those nodes does and how they fit together.

The Fighter Scene at a Glance

The fighter is built as its own scene, using a **CharacterBody3D** as the root node. CharacterBody3D is well suited for characters that need to be moved around on screen with custom movement logic, which is exactly what we need for a fighting game. Inside this root node, the fighter scene contains all of the components needed for movement, jumping, dashing, attacking, animation, hit flashing, outfit changes (so the two players can be visually distinguished), a stamina controller, on-screen UI, and input management.



There are quite a few child nodes in total, so the rest of this lesson walks through each one, grouped roughly by purpose: the visuals and animation layer, the input layer, the state machine and

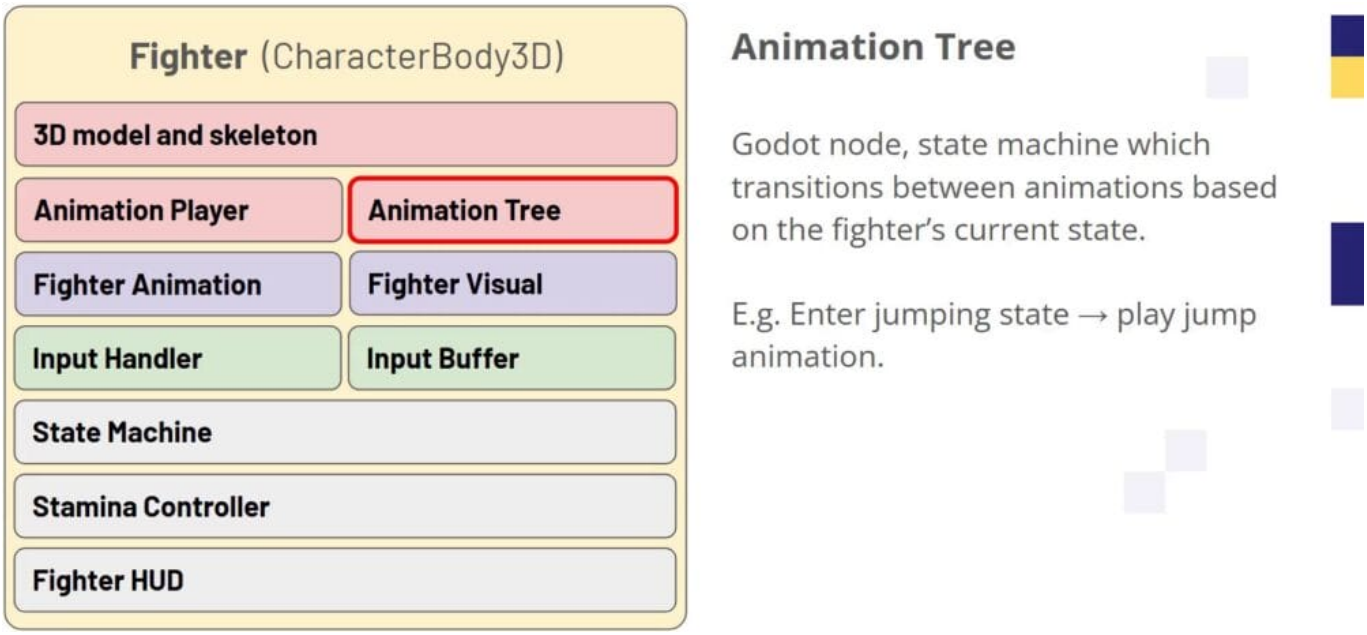
gameplay logic, and the on-screen interface.

3D Model, Skeleton, and Animation Player

Because this is a 3D game, the fighter includes a fully rigged and animated 3D character. The **3D model and skeleton** represent the visible mesh and the underlying bone structure that will be posed as the character walks, throws punches, jumps, and performs other actions. The skeleton's bone rotations are driven by the **Animation Player**, a built-in Godot node that stores every animation the fighter can play — idle, move, jump, attack, and so on.

Animation Tree

The Animation Player holds the animations, but something still has to decide which animation should play at any given moment. That is the job of the **Animation Tree**, another Godot node. The Animation Tree acts as a state machine that transitions between animations based on the fighter's current state. For example, when the character is standing still, the Animation Tree keeps them in the idle state playing the idle animation. When the player presses the jump button and the character jumps into the air, the fighter's state changes, and the Animation Tree transitions over to the jump animation. Once the character lands back on the ground, the Animation Tree transitions back to the standing animation.



Fighter Animation

Sitting between the game's own logic and the Animation Tree is a custom node called **Fighter Animation**. This is not a built-in Godot node — it is something we will create ourselves, acting as a thin layer of glue between the custom state machine (covered below) and the Animation Tree. The Fighter Animation node keeps track of what state the character is currently in and communicates that information to the Animation Tree, so the correct animations play at the correct times. Think of it as the bridge between the character's brain and its visible movement.

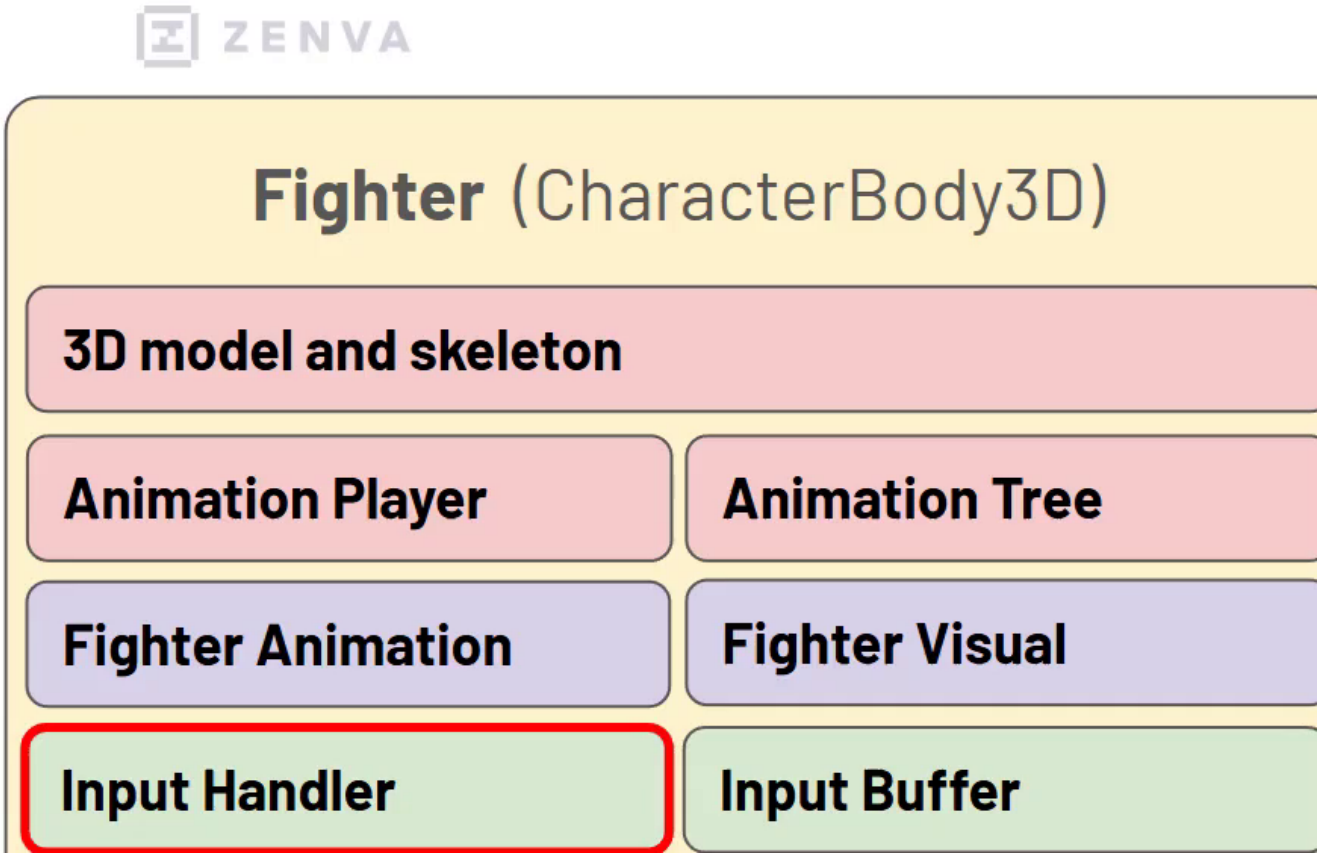
Fighter Visual

The **Fighter Visual** node is responsible for a number of visual details that are not part of the skeletal animation. Two key examples are the outfit material the character is currently wearing — which is how we will differentiate player one from player two — and the brief white flash applied to

the character's model for a few frames whenever they take damage. Anything related to how the character looks beyond their animation is centralized in this node.

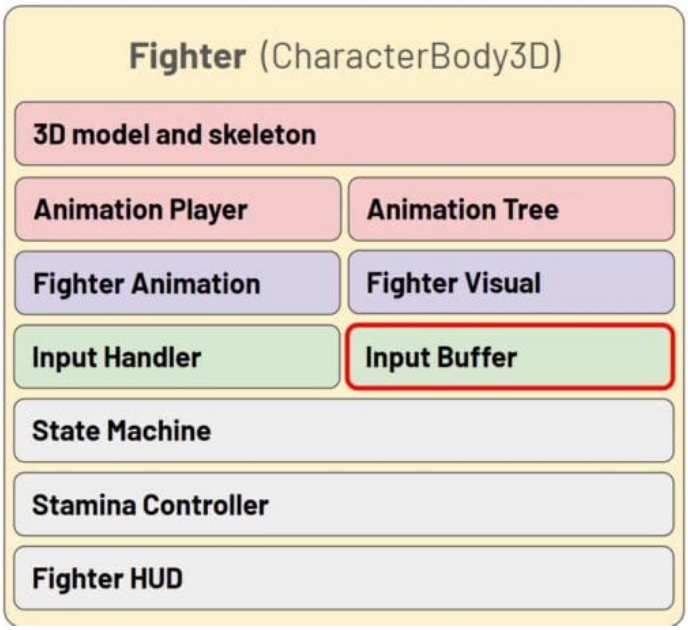
Input Handler

The fighter has two dedicated input-related nodes, and the first is the **Input Handler**. This node is responsible for detecting inputs from the keyboard and wrapping them into an input packet. Every frame, the Input Handler reads which actions are currently pressed for this fighter and produces an input packet that represents the character's intent during that frame. That packet is then passed on to the next node in the chain.



Input Buffer and the Rolling Buffer Concept

The packet produced by the Input Handler is sent over to the **Input Buffer**. The Input Buffer keeps a rolling buffer of input packets, which the state machine can then read from. A rolling buffer is essentially a fixed-size array of input packets — for example, 12 or 20 frames long. Every time a new input packet is added to the front of the buffer, the oldest packet is removed from the end, so the buffer always represents the most recent window of inputs the player has made.



Input Buffer

Keeps a rolling buffer of InputPackets which is then read by the State Machine.

This allows us to press the attack key while an attack is already in progress, then when we exit the attack state, we can read the buffer *n* number of frames in the past.

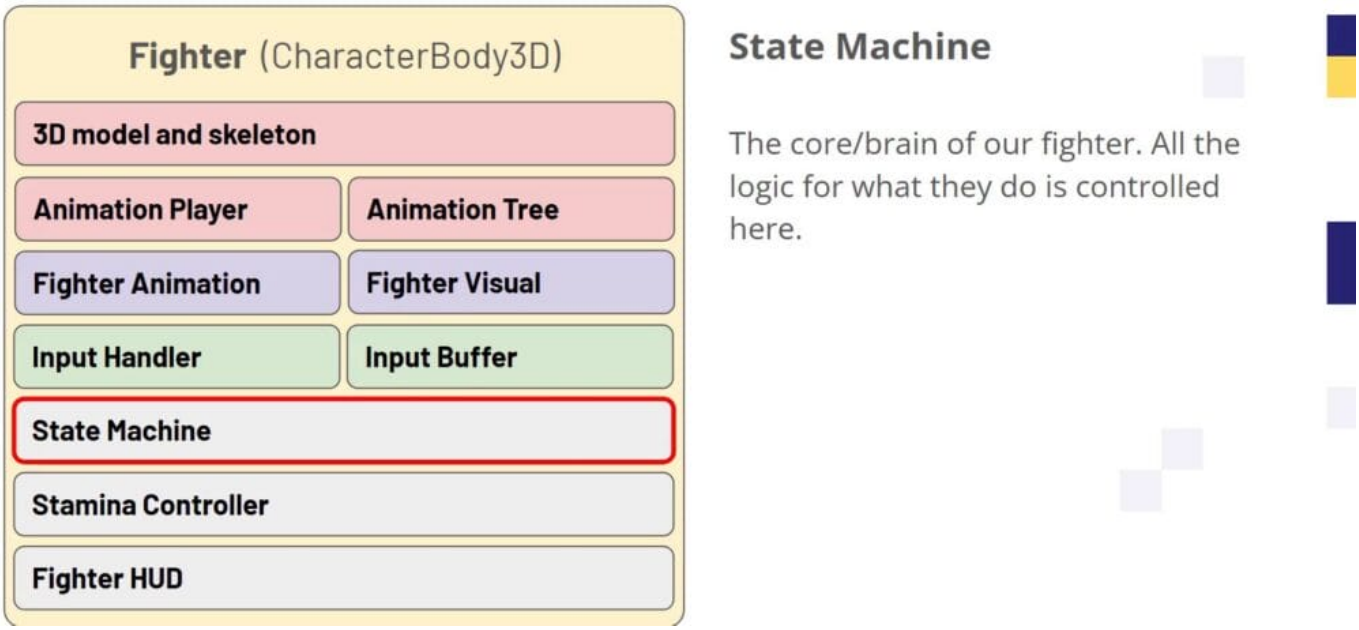
This allows for smoother feeling controls.

This approach is very useful for making controls feel smooth and forgiving. Many games use a technique like this without ever telling the player about it, but it has a real effect: subconsciously, the controls just feel nicer. Consider a common scenario — the character is jumping in the air, and the player presses the attack button while still a couple of frames away from landing. With traditional input detection, the game would register the attack press during a frame in which the character cannot attack (because they are still mid-air), the player would land, and nothing would happen.

A rolling buffer solves this. When the player presses the attack button, the corresponding input packet is added to the buffer, and the buffer continues to shift along as the frames progress. When the character finally lands on the ground, the game can look back over the last five or six frames of the buffer and ask: “Did the player press attack recently?” If so, the attack is triggered immediately on landing. The result is that players do not have to hit their inputs with frame-perfect timing — they can mash buttons naturally, and the game will understand what they intended to do.

State Machine — The Brain of the Fighter

After the input layer comes the **State Machine**. This is the brain of the fighter, and a large portion of the code we write will live inside it. A later lesson will go into much more depth about how this custom state machine works, but at a high level it is responsible for controlling every major behavior the fighter can exhibit.



The fighter will have a dedicated state for each major behavior, including:

- A **standing** state for when the fighter is moving around on the ground
- An **attacking** state for when they are performing an attack
- A **hit** state for when they have been struck by an opponent
- A **jumping** state for when they are in the air
- A **dashing** state for quick directional movement
- A **defeated** state for when they have lost the match

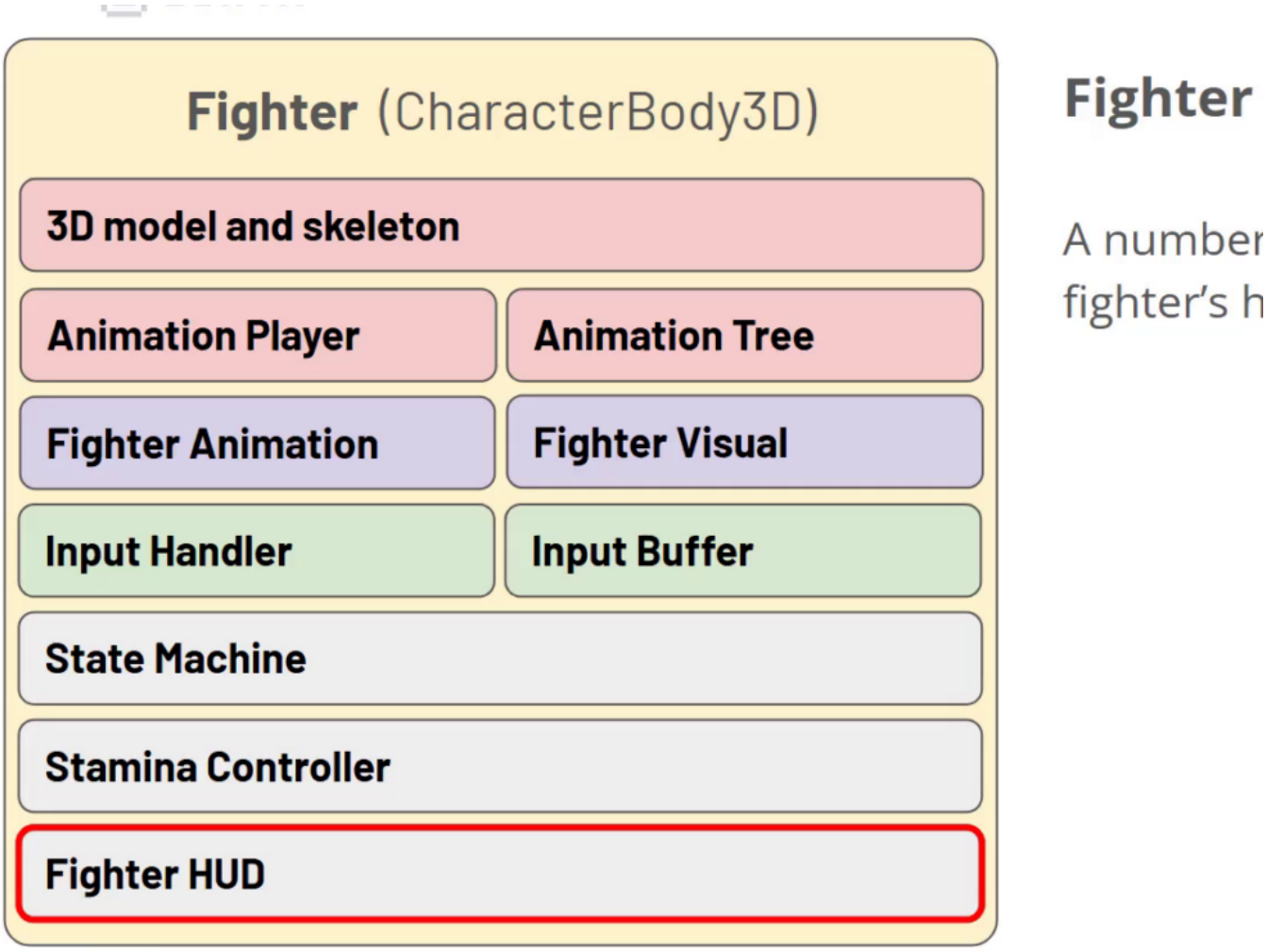
Transitions between these states will be driven by our own custom logic, which reads from the Input Buffer and reacts to what is happening in the game world.

Stamina Controller

The **Stamina Controller** is a dedicated node that keeps track of how much stamina the fighter currently has. It handles regeneration when stamina depletes, and it also exposes functions for subtracting stamina whenever the fighter performs a stamina-costing action such as attacking, dashing, or jumping. Keeping stamina logic in its own node keeps it cleanly separated from the rest of the fighter's behavior.

Fighter HUD

Finally, the fighter scene contains a **Fighter HUD** node. The HUD is a collection of control nodes responsible for rendering the player's health and stamina as little progress bars on the screen. It will automatically position itself on the left or right side of the screen depending on whether this fighter belongs to player one or player two, so both players can see their status at a glance.



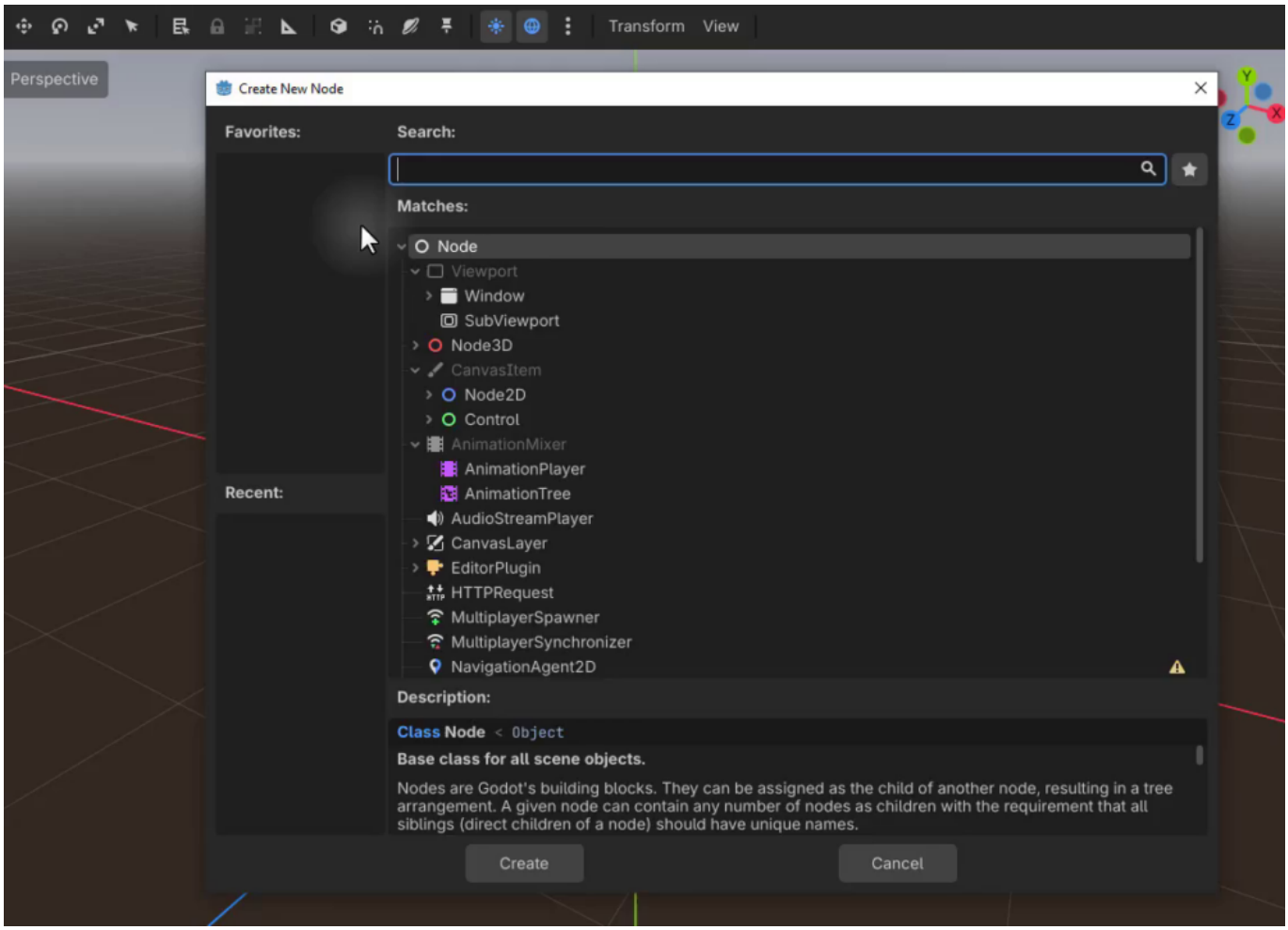
Wrapping Up

As you can see, the fighter is a fairly large collection of nodes, but this is really where the meat of the game lives and where most of our attention will be focused throughout the course. Understanding what each node is responsible for — and how the pieces connect from input all the way through to animation and visuals — will make the upcoming lessons much easier to follow. If any of this feels unclear right now, feel free to watch this lesson again or review it later as a reference. In the next lesson, we will start actually building the fighter inside Godot.

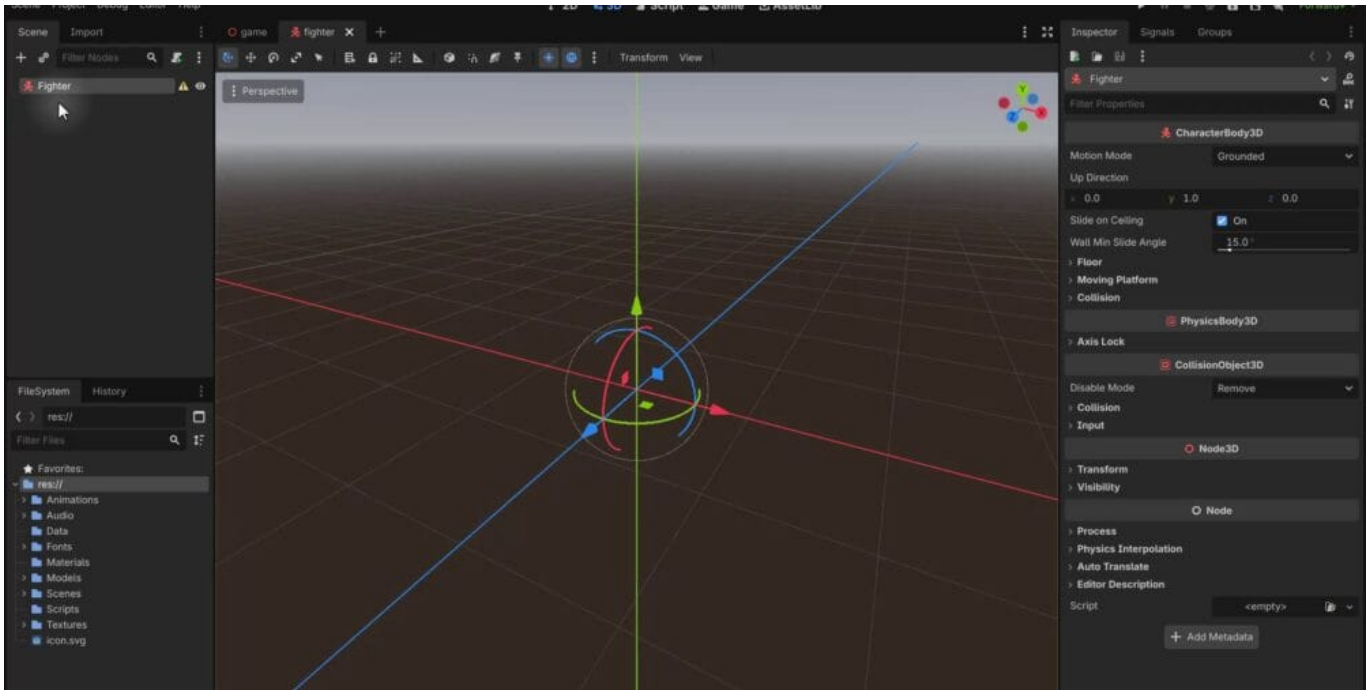
In this lesson we begin building the fighter scene — the character that each player will control in the game. This is the foundation every other system (input, state machine, animation, combat) will plug into, so we start by creating the scene, bringing in the character model, rigging the head so it follows the body, setting up animations, and finally giving the character a basic collision shape.

Creating the Fighter Scene

We begin by creating a brand new scene through the **Scene > New Scene** menu. The root node of the scene needs to be a **CharacterBody3D**, which is the node Godot provides for characters that move through a 3D world and react to the environment. From the root node selection dialog, choose **Other Node** and search for “CharacterBody3D” to create it.

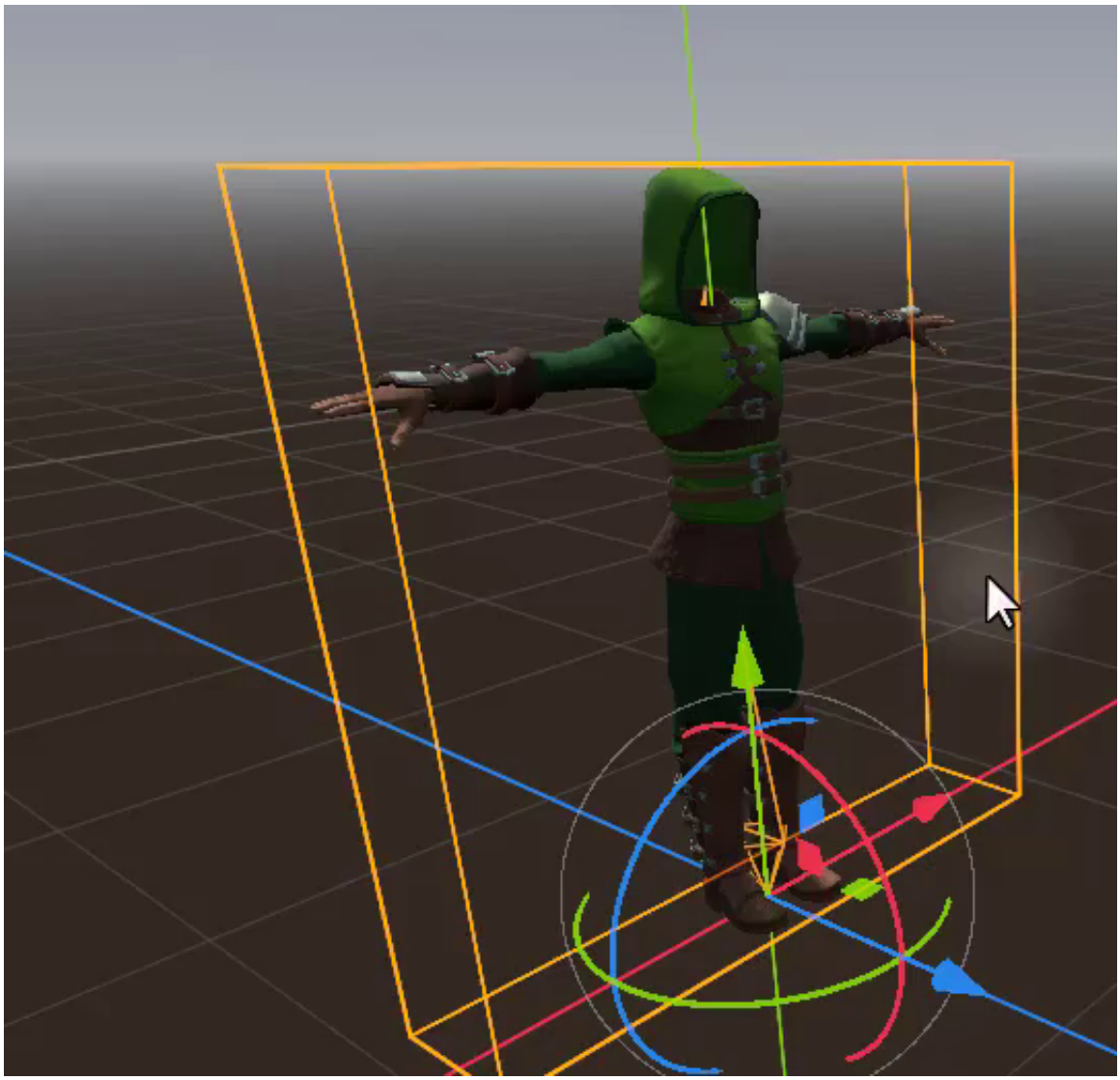


Rename the new root node to **Fighter**, then save the scene. Inside the existing *Scenes* folder, create a new *Fighters* subfolder and save the scene there as **fighter.tscn**. At this point the scene contains a single empty CharacterBody3D node ready to have more components attached to it.



Adding the Character Model

The main visual element attached to the fighter is the character's 3D model. From the FileSystem panel, navigate into *Models*, then into *Outfits*, and drag the **Male Ranger** model into the Fighter scene. Once it is in the scene, set its position and rotation to zero so it is centered at the origin.



You will notice immediately that the character has no face. This is not a bug — the asset pack used in this course deliberately separates faces from outfits so that different heads can be mixed and matched with different outfits. The head therefore has to be attached as a separate step.

Attaching the Head to the Skeleton

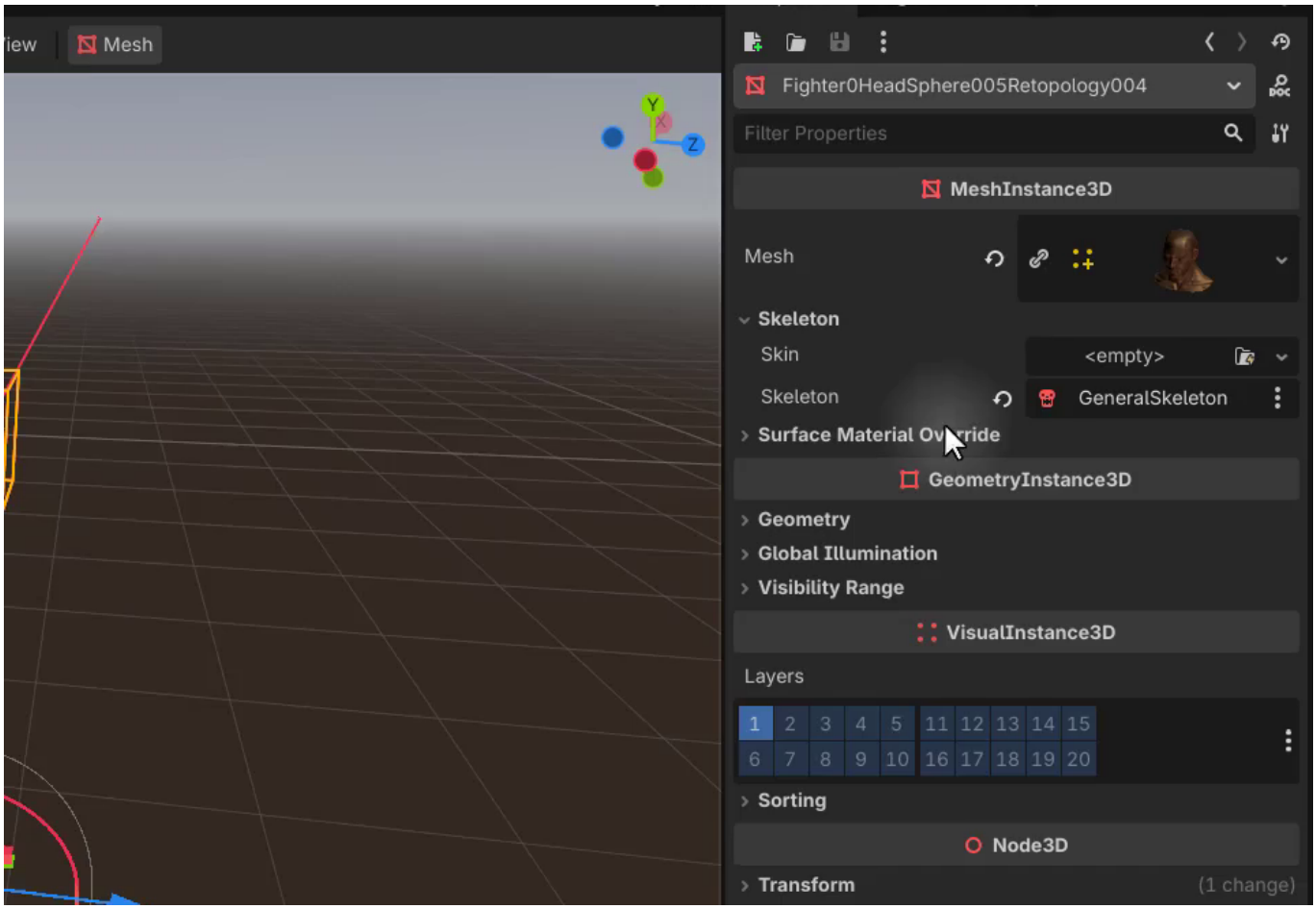
If you look inside the *Models* folder, you will find a *Fighter Head* folder containing three relevant files: a face, a second face (the eyebrows), and a sphere that represents the head itself. Dragging all three of these into the scene places them at the origin, but simply adding them is not enough — they need to follow the body whenever an animation is played.

To make the head follow the body, it needs to be attached to the same skeleton that drives the rest of the character. A **skeleton** in a 3D model is the collection of bones that control how the mesh deforms when animations play. Joints exist at each meaningful point on the body — fingers, neck, arms, shoulders, hips, knees, feet and toes — and when an animation plays, the skeleton moves and the mesh that sits over it follows along.

The skeleton inside the Male Ranger is hidden by default. To access it, right-click the Male Ranger node in the Scene tree and enable **Editable Children**. This reveals the internal structure: an **Armature** node containing a **GeneralSkeleton** with all of the character's bones.

With the skeleton accessible, the head, eyebrows, and eyes each need to be reparented to it and told which skeleton they belong to. For each of the three head meshes:

1. Drag the mesh so it becomes a child of the **GeneralSkeleton** node.
2. Select the mesh and, in the Inspector, find the **Skeleton** property. It will be empty by default — drag the GeneralSkeleton node into that field so the mesh is bound to it.
3. Reset the mesh’s position to **(0, 0, 0)** and its rotation to zero so it sits exactly where the skeleton expects it.



Setting Up the AnimationPlayer

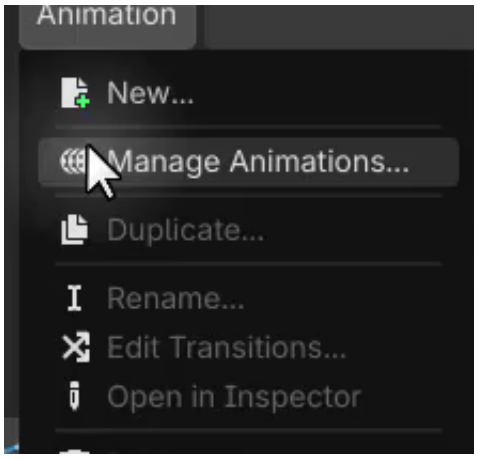
At this point, the head is attached, but without an animation playing, there is no way to see whether everything will track correctly. To fix that, attach an **AnimationPlayer** node to the Fighter scene. The AnimationPlayer is the node responsible for listing and playing animations for a given skeleton.

With the AnimationPlayer selected, the Inspector exposes a **Root Node** property. This needs to point at the Male Ranger model, so the player knows which node hierarchy the animations apply to. Drag the Male Ranger into the Root Node field.

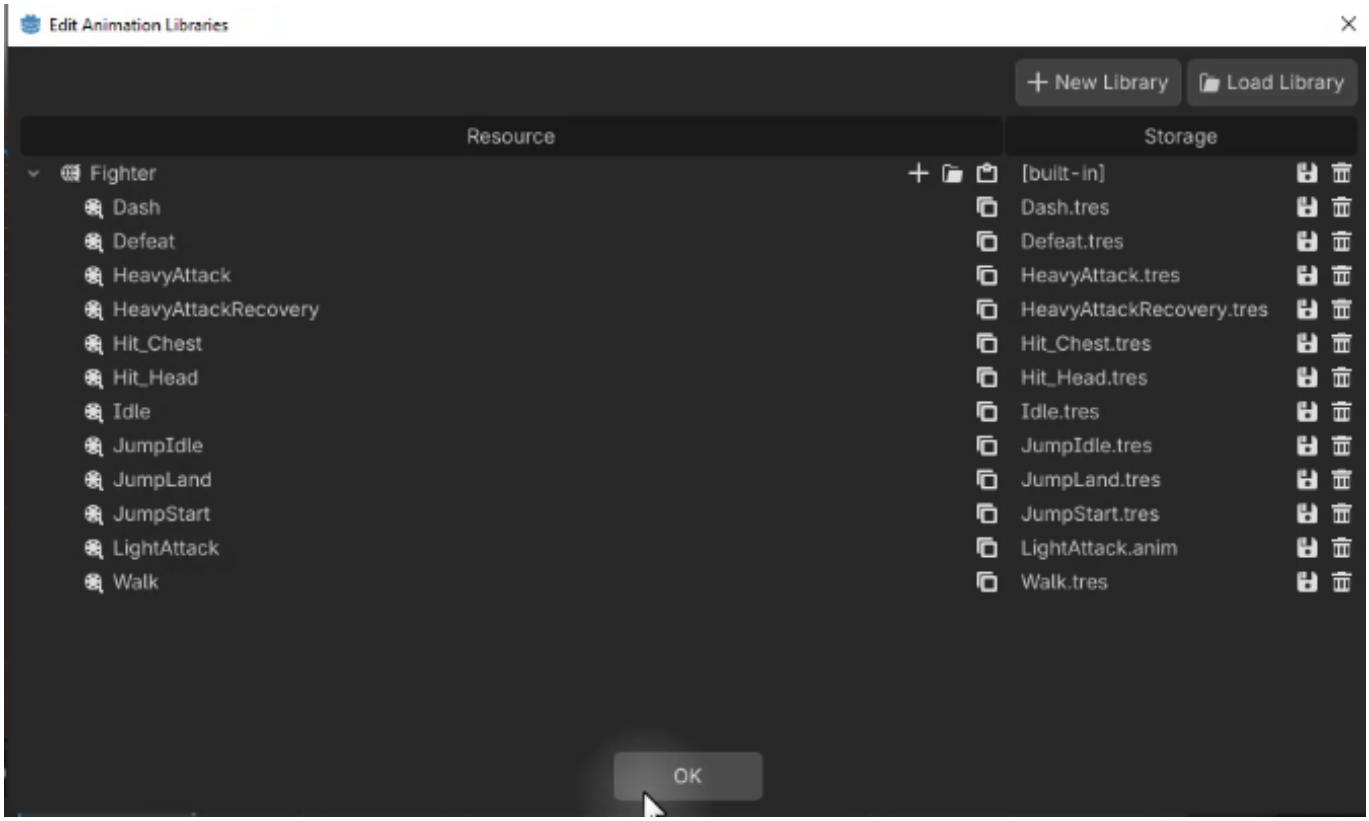
The animation editor appears at the bottom of the screen when the AnimationPlayer is selected. If it does not appear automatically, click the **Animation** button at the bottom of the editor to open it.

Creating an Animation Library

Animations in Godot are grouped into **libraries**, which act like playlists of animations that can be assigned to an AnimationPlayer. From the animation panel, go to **Animation > Manage Animations** to open the library editor.

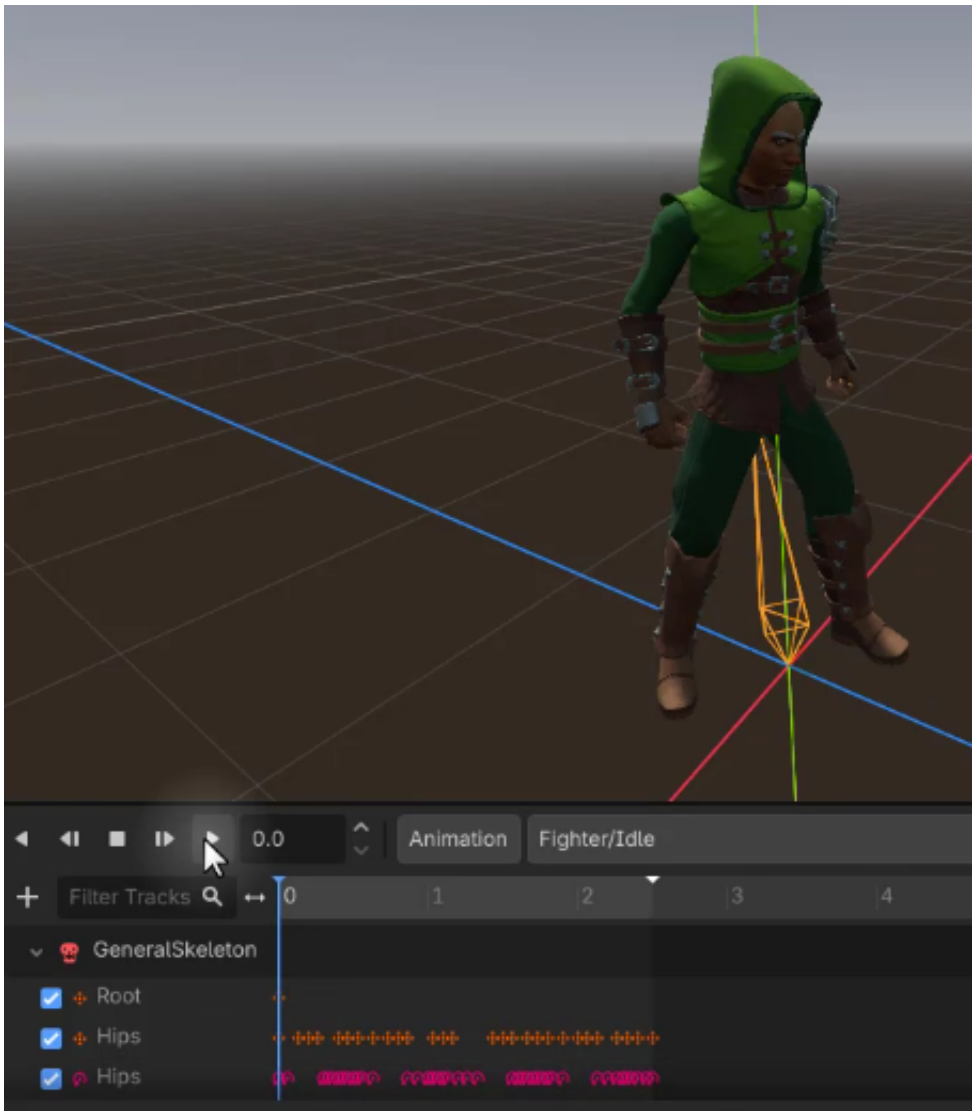


In the Manage Animations window, create a new library and name it **fighter**, then click OK. With the empty library created, click the folder icon labelled **Load animation from file and add to library**. Browse into the *Animations* folder supplied with the course project, select all of the animation files at once, and click Open. Click OK to close the window. The new library is now populated with every animation the fighter will need.



Testing the Animations

With the library loaded, use the animation dropdown in the AnimationPlayer panel to select a specific animation and press Play to preview it. Starting with **Idle** is a good sanity check — the character should stand and breathe naturally.



Switch to the **HeavyAttack** animation and press Play to see the character perform a heavier motion. This is a particularly useful test because it moves the head around dramatically, which makes it obvious whether the head, eyebrows, and eyes are tracking with the body as intended.

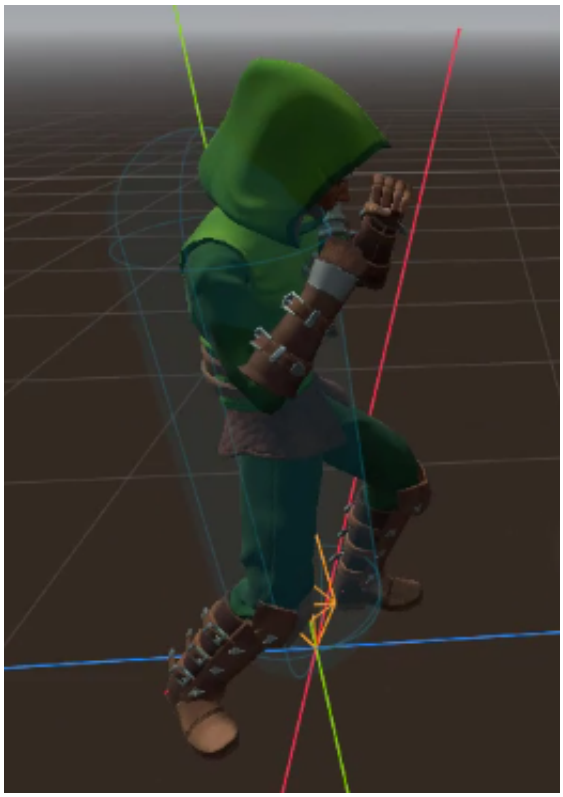
If the head or facial features do not follow the body during the animation, double-check the following for each of the three head meshes:

- The mesh must be a direct child of the **GeneralSkeleton** node in the Scene tree.
- Its **Skeleton** property in the Inspector must be set to the GeneralSkeleton node.
- Its position must be **(0, 0, 0)** and its rotation must be zero, otherwise there will be a visible offset from the rest of the body.

Once all three conditions are satisfied, the character will be fully animated. Keep in mind that being able to play animations manually does not yet mean the animations react to gameplay events — that logic will be wired up later on through an AnimationTree, which is not part of this lesson.

Adding a Collision Shape

With the visuals and animations working, the Fighter node still shows a warning because a CharacterBody3D requires a collision shape in order to interact with the physics world. Attach a new **CollisionShape3D** node to the Fighter and, in the Inspector, set its **Shape** to a **Capsule**. Move and resize the capsule so it roughly matches the character’s body.



A precise fit is not required here. This collision shape is used only for interacting with the environment — walking on the ground and colliding with obstacles to the left or right. Hits between fighters will be handled later through a custom hitbox and hit sender implementation, so the capsule just needs to be a reasonable approximation of the character's body.

Summary and Challenge

At this point, the fighter has everything it needs for a basic visual presence: a CharacterBody3D root, a rigged Male Ranger model with a properly attached head, an AnimationPlayer loaded with a fighter animation library, and a capsule collision shape for interacting with the environment. From here, the remaining work for the fighter will revolve around creating custom nodes for detecting inputs and moving the character around.

Before the next lesson, try the following challenge to prepare the game world for testing:

- Go back to the main game scene and build a simple platform that the fighters can stand on, so that when gravity is applied, the characters do not fall straight through the ground.
- Set up a camera with a side-on view of that platform, matching the perspective of a traditional fighting game.

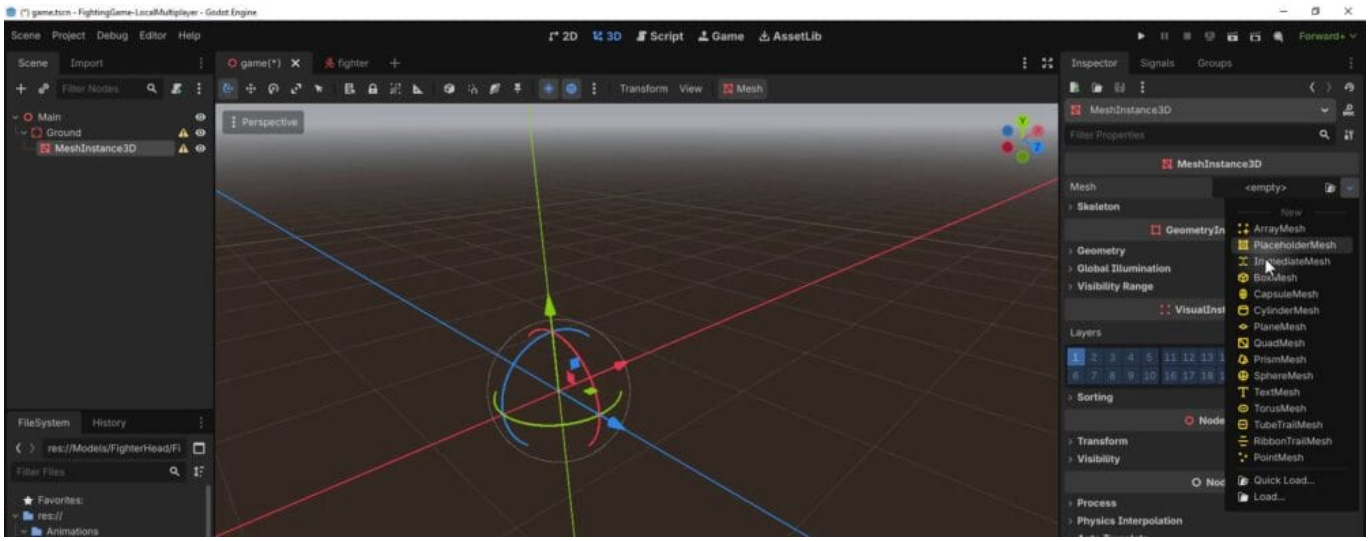
Completing the challenge will make it much easier to test the input and movement systems added in the following lessons.

In this lesson we continue building the fighter test scene that was started in the previous lesson. The goal is to end up with a minimal playable environment containing a ground platform, a camera, lighting, a sky, and two fighter characters — and then to attach a first script that will become the central hub for each fighter.

Building the test platform

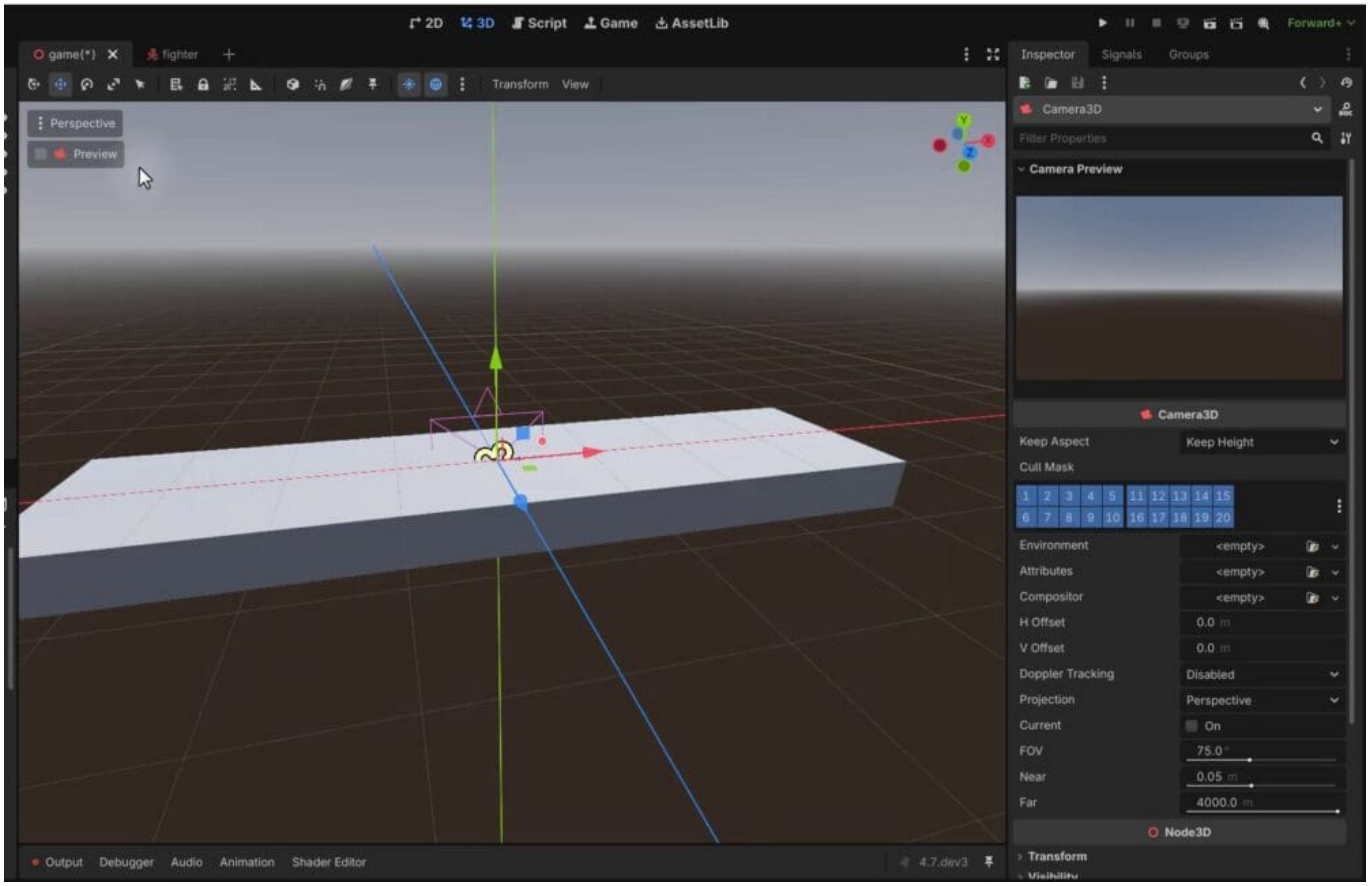
Rather than jumping straight into the final environment, we start with a simple sample platform so that we have a surface to drop the fighters onto. In the scene, create a new **StaticBody3D** and rename it to **Ground**. Attach a **MeshInstance3D** to it as a child — this is going to be the visible surface of the platform.

With the MeshInstance3D selected, open the Mesh property in the Inspector and set it to a **BoxMesh**. This gives the ground its flat, box-shaped geometry.



Next, attach a **CollisionShape3D** as another child of the Ground node. Its shape should also be a box, sized to match the visible mesh so the fighters have something solid to stand on. Finally, add a **Camera3D** to the scene and position it so that the ground is visible in front of it.

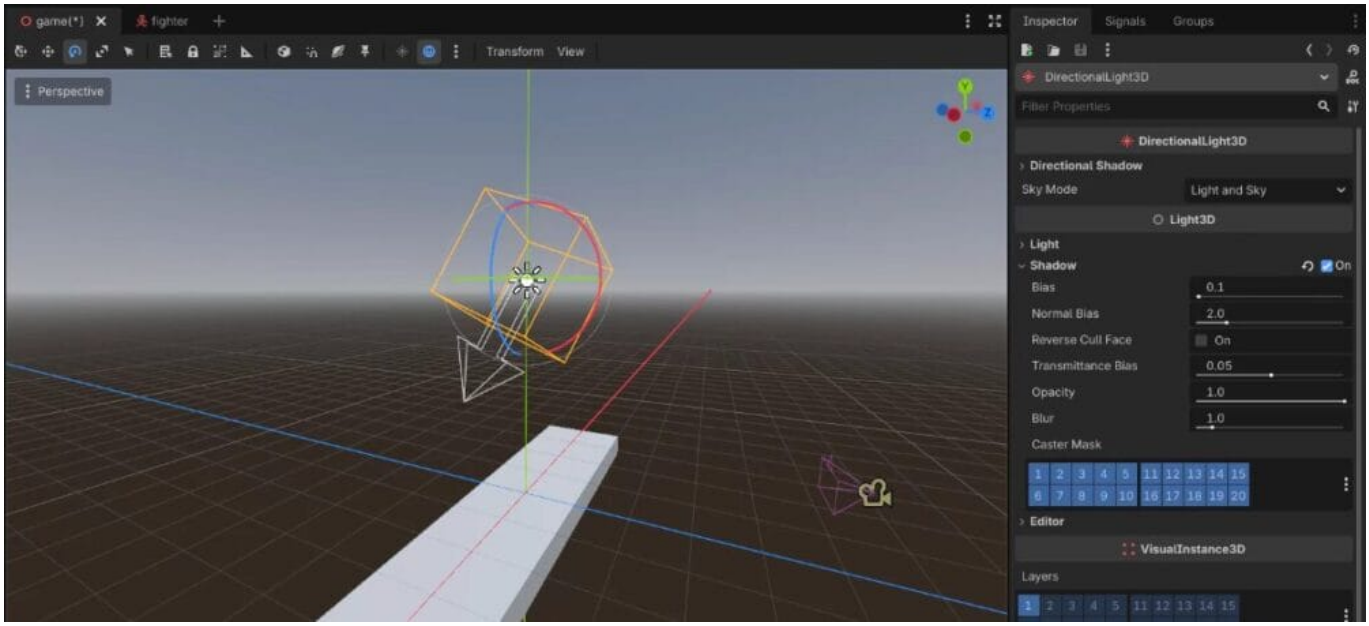
With the camera selected, lower the field of view to **50** to get a slightly narrower angle, and then pull the camera back a bit so that the platform fits comfortably in view.



Adding lighting and a skybox

Before pressing play, select the current game scene as the main scene so Godot knows what to launch. When the game runs you will immediately notice that it looks very dark. This is expected: Godot does not add a default light or environment to new 3D scenes, so we have to do that ourselves.

To fix this, add a **DirectionalLight3D** to the scene. Move it up above the ground and rotate it so that it points downward and slightly at an angle — this gives the scene a natural-looking primary light direction. In the light's properties, enable **shadows** so the fighters will cast them onto the ground.



Next, add a **WorldEnvironment** node. This is what lets us define the overall sky and ambient

environment. In the Inspector, configure it as follows:

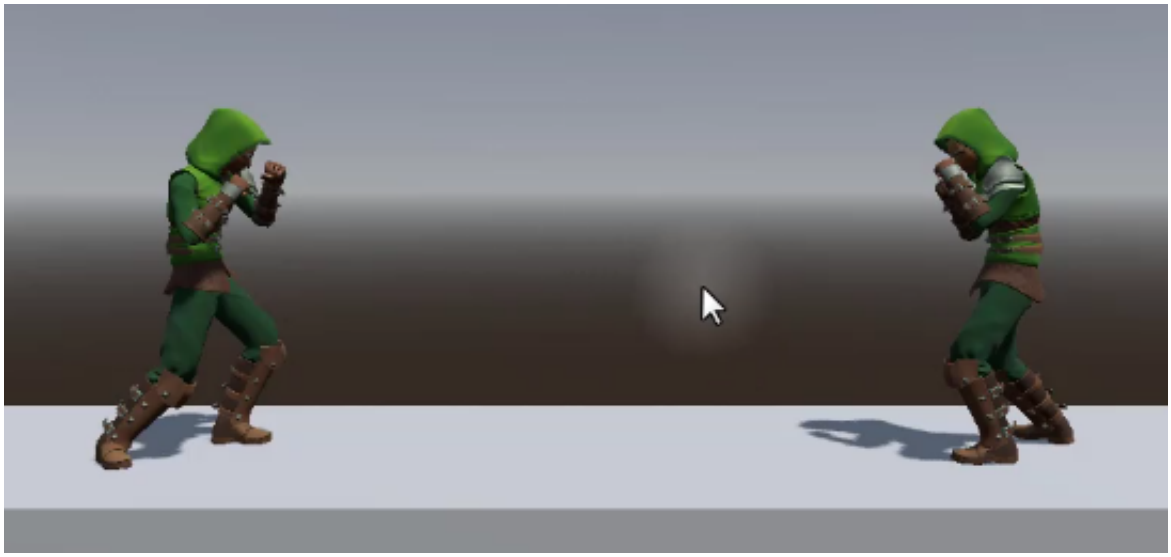
1. Under Environment, create a **New Environment**.
2. Open the environment and go to **Background**, then change the Mode to **Sky**.
3. Scroll down to the **Sky** section and create a **New Sky**, then open it.
4. For the Sky Material, create a **New ProceduralSkyMaterial**.

Save the scene and press play. The running game should now look much closer to what you see in the editor viewport: a lit ground plane with a procedural sky in the background.

Placing the fighters

With the test environment ready, we can bring in the fighter character that was built earlier in the course. In the FileSystem dock, navigate to **Scenes › Fighters** and drag the **Fighter** scene into the viewport. You will see a default animation plane attached to the character, but it does not really change anything at runtime and can be ignored for now.

Rotate this first fighter so that its forward direction faces to the right. Then duplicate the node to get a second fighter, move it over to the opposite side of the platform, and rotate it so that it faces the other way. You now have two fighters standing on the ground, ready to face off.

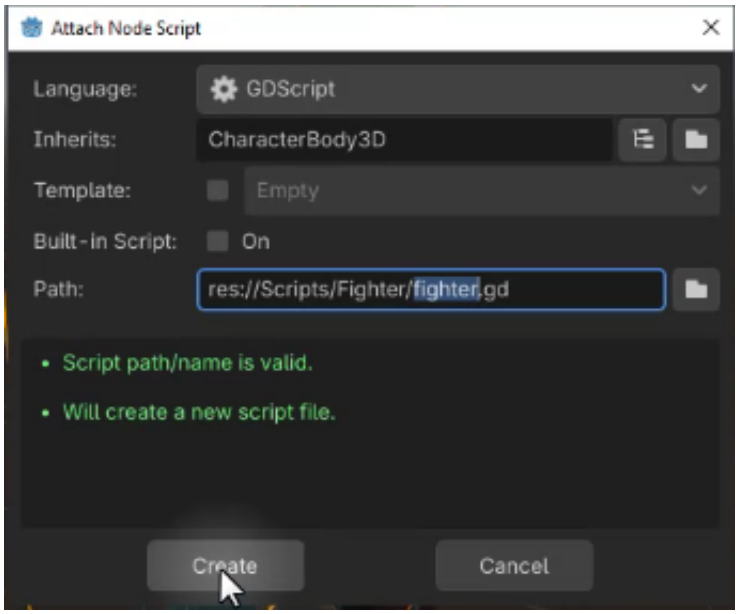


Attaching the Fighter script

With the fighters in the scene, the next job is to start wiring up gameplay. The first step is to attach a new script to the fighter scene that will act as the central hub for everything related to that character — connecting its different components together as we build them up in later lessons.

Open the **Fighter** scene (not the game scene) so that the script is attached directly to the fighter itself. In the FileSystem dock, go to the **Scripts** folder and add a new subfolder inside it called **Fighter**. This is where all fighter-related scripts will live.

Back in the Scene tree, select the root node of the Fighter scene and create a new script called **fighter.gd**. When the save dialog opens, navigate to **Scripts/Fighter** and store the file there.

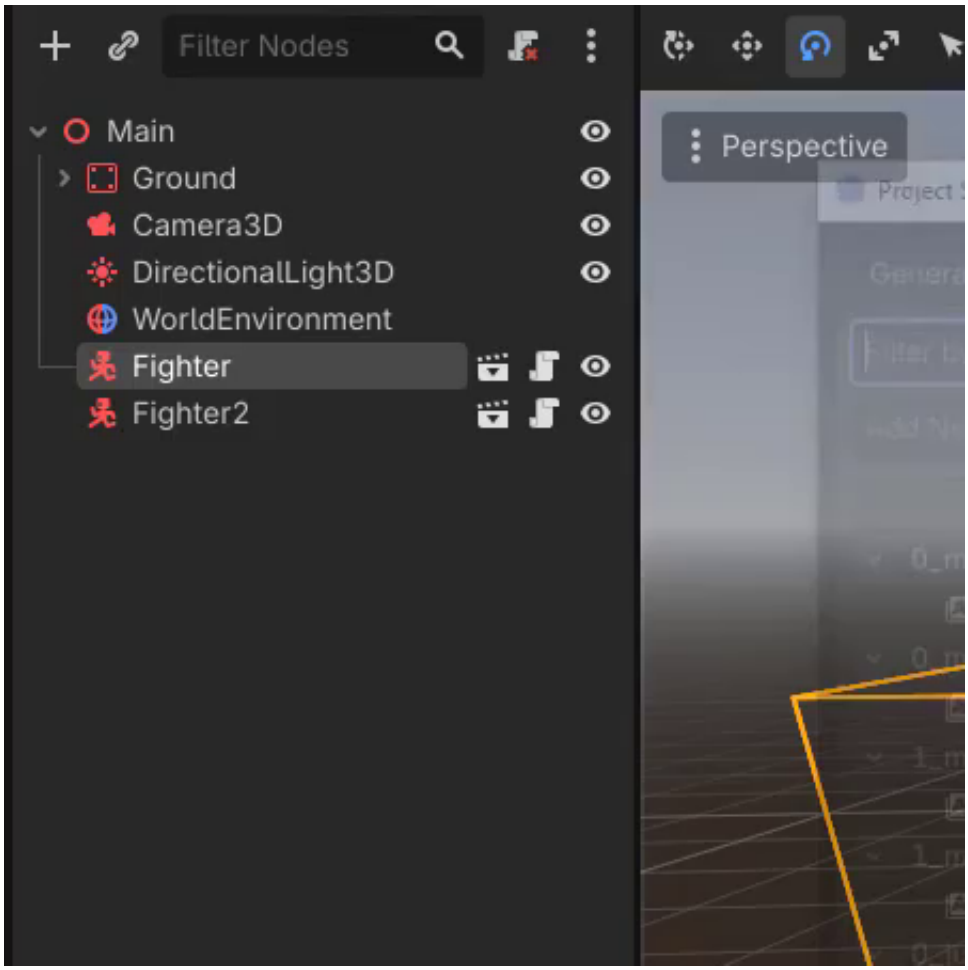


Inside the new script, start by giving the class a name. A class name makes the Fighter type available globally, which will be useful later when other scripts need to type-reference a fighter statically. Because the fighter root node is a CharacterBody3D, the script also extends that class.

Then add a single exported variable — player_id of type int. This value is what tells the rest of the game which player is controlling this particular fighter.

```
class_name Fighter
extends CharacterBody3D

@export var player_id : int
```

Save the script and go back to the game scene. Select the first fighter and, in the Inspector, set its **Player ID** to **0**. Then select the second fighter and set its **Player ID** to **1**. These values line up with the input map defined earlier in the course: player **0** uses the WASD keys on the left, while player **1** uses the arrow keys and the matching action buttons on the right.



What's next

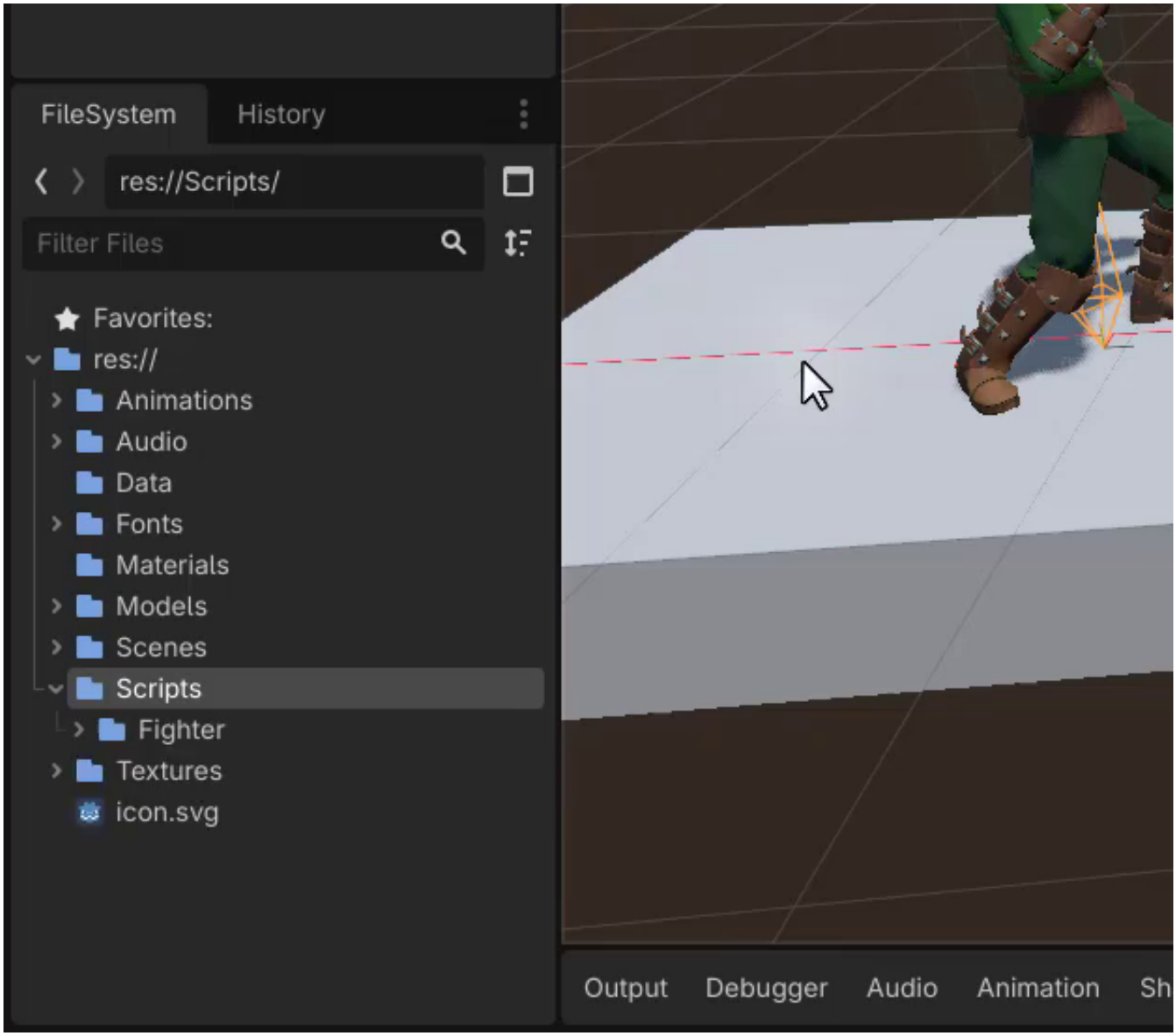
At this point, the fighter test scene is fully set up: there is a ground platform with collision, a camera with a narrower field of view, a directional light, and a procedural skybox for lighting, and two fighters assigned to player IDs 0 and 1. The groundwork is in place, but the fighters still cannot react to input. In the next lesson, we will start detecting keyboard input for each player and feeding it into the Fighter so the characters can actually move around.

In this lesson we begin building the input handling system for our fighting game. The input handler is the component responsible for detecting player inputs from the keyboard, packaging them up into an *input packet*, and sending that packet over to the input buffer. The input buffer will later keep a rolling buffer of the last few frames of inputs, which is what allows tricks like pressing the attack button a couple of frames before landing and still having it register as a valid attack.

This lesson focuses on the first two pieces of that pipeline: the InputPacket data class and the base InputHandler script, plus a first pass at a PlayerInputHandler that reads the keyboard.

Creating the Inputs folder and input_packet script

Inside the **Scripts** folder, create a new folder called **Inputs**. This folder will hold every script related to input handling. Inside it, create a new script called input_packet.gd.



The input packet is going to be a lightweight class that simply holds a dictionary of input states. Because it is never attached to a node — it is just a chunk of data created in memory when needed — we can remove the default extends Node line. Start by giving the script a class name:

```
class_name InputPacket
```


Defining the inputs dictionary

Next, declare a variable called inputs of type Dictionary. A dictionary is similar to an array, except that instead of accessing elements by numeric index, each element is identified by a custom key. For our input packet, the key type will be String and the value type will be bool — each entry answers the question “is this input pressed down this frame?”.

Initialize the dictionary with one entry for every possible input the player can make. We need move_left, move_right, jump, dash, light_attack, and heavy_attack, all defaulting to false:

```
class_name InputPacket

var inputs : Dictionary[String, bool] = {
    "move_left": false,
    "move_right": false,
    "jump": false,
    "dash": false,
    "light_attack": false,
    "heavy_attack": false
}
```

With this structure, each frame we can read an InputPacket and know exactly which inputs were pressed on that frame.

Adding the is_pressed helper

To make reading the dictionary a bit more convenient, we will add a helper function called is_pressed. It takes an input name as a String and returns a bool indicating whether that input is currently pressed.

Inside, we use the dictionary’s built-in get method. The second argument to get is the default value to return when the key is not found — we pass false, so unknown inputs are treated as “not pressed”:

```
func is_pressed (input_string : String) -> bool:
    return inputs.get(input_string, false)
```

So if we want to know whether jump was pressed on a given packet, we can simply call packet.is_pressed("jump") and we’ll get a boolean back. If the key was missing for any reason, we still safely get false instead of an error.

Creating the base InputHandler script

Next, inside the same **Inputs** folder, create another new script called input_handler.gd. Unlike the input packet, this one *will* be attached to a node, so leave the extends Node line in place and give it a class name of InputHandler.

We need a single exported variable called fighter of type Fighter. This reference is needed because the handler needs to know which player ID it is reading inputs for — that information lives on the

fighter node.

We also declare a single function, get_input_packet, which returns an InputPacket. For now, this function simply returns null:

```
class_name InputHandler
extends Node

@export var fighter : Fighter

func get_input_packet () -> InputPacket:
    return null
```

At first glance this script looks unfinished — it does nothing. That is intentional. InputHandler is not the script we will actually attach to a node in the scene. Instead, it acts as a **template** (a base class) that other, more specialized input handlers will inherit from. This is important because the game will eventually need several different flavours of input handlers:

- A local player handler that reads keyboard inputs.
- An enemy AI handler (planned for a future course) that lets an AI latch on and feed inputs through the same interface.
- A networked multiplayer handler for online matches.

By defining get_input_packet here in the base class, every subclass is guaranteed to provide one, and the rest of the fighter code only has to know about the base InputHandler.

Creating the PlayerInputHandler

Now we create the first concrete handler. Inside the **Inputs** folder, add one more script called player_input_handler.gd. This one extends our base handler and gets its own class name:

```
class_name PlayerInputHandler
extends InputHandler
```

To start detecting inputs, we override the get_input_packet function we declared on the base class. Inside, we create a fresh InputPacket, and at the end we simply return it. Everything in between is where we populate the packet with the current frame’s input states:

```
func get_input_packet () -> InputPacket:
    var packet : InputPacket = InputPacket.new()

    return packet
```

As a first attempt at filling the packet, we assign an entry for move_left by calling Godot’s built-in Input.is_action_pressed. For now, we hard-code the action name to "0_move_left" — the 0_ prefix is there because our input map defines separate actions per player (player 0 and player 1):

```
func get_input_packet () -> InputPacket:
    var packet : InputPacket = InputPacket.new()
```



```
    packet.inputs["move_left"] = Input.is_action_pressed("0_move_left")

    return packet
```

This works — but only for player zero. If we left it like this, the second player in the game would not be able to move left at all, because they are bound to the "1_move_left" action instead.

Supporting both players with _full_action_name

To make this script work for both players without duplicating every line, we add a small helper at the top of the class called _full_action_name. It takes a base action name as a String and returns a new String with the fighter's player ID prepended to it:

```
func _full_action_name (action : String) -> String:
    return str(fighter.player_id, "_", action)
```

The str() function simply concatenates all of its arguments into a single string. So if we send in "move_left", the function looks up fighter.player_id — either 0 or 1 — and returns something like "0_move_left" or "1_move_left", exactly matching the action names defined in our input map.

We can now replace the hard-coded "0_move_left" string in our get_input_packet function with a call to _full_action_name("move_left"):

```
func get_input_packet () -> InputPacket:
    var packet : InputPacket = InputPacket.new()

    packet.inputs["move_left"] = Input.is_action_pressed(_full_action_name("move_left
"))

    return packet
```

With this change, the same single line of code works for either player, because the action name is resolved dynamically from the fighter's player ID.

Challenge

Before moving on to the next lesson, take a moment to finish filling out get_input_packet yourself. Using the exact same pattern as the move_left line above, add entries for the remaining inputs — move_right, jump, dash, light_attack, and heavy_attack — each calling Input.is_action_pressed with _full_action_name for the matching action.

The solution, along with the rest of the input handling pipeline, will be covered in the next lesson.

In this lesson, we complete the PlayerInputHandler script by filling in all of the input actions, then integrate it into the Fighter scene. We also begin building the core game loop inside the Fighter's _physics_process function and test that inputs are being captured correctly for both players.

Completing the PlayerInputHandler

In the previous lesson's challenge, the task was to duplicate the move_left input check and update it for the remaining five actions. Each line inside get_input_packet() follows the same pattern: it reads whether a specific action is pressed using the player-specific action name, then stores the result as a boolean in the InputPacket's inputs dictionary.

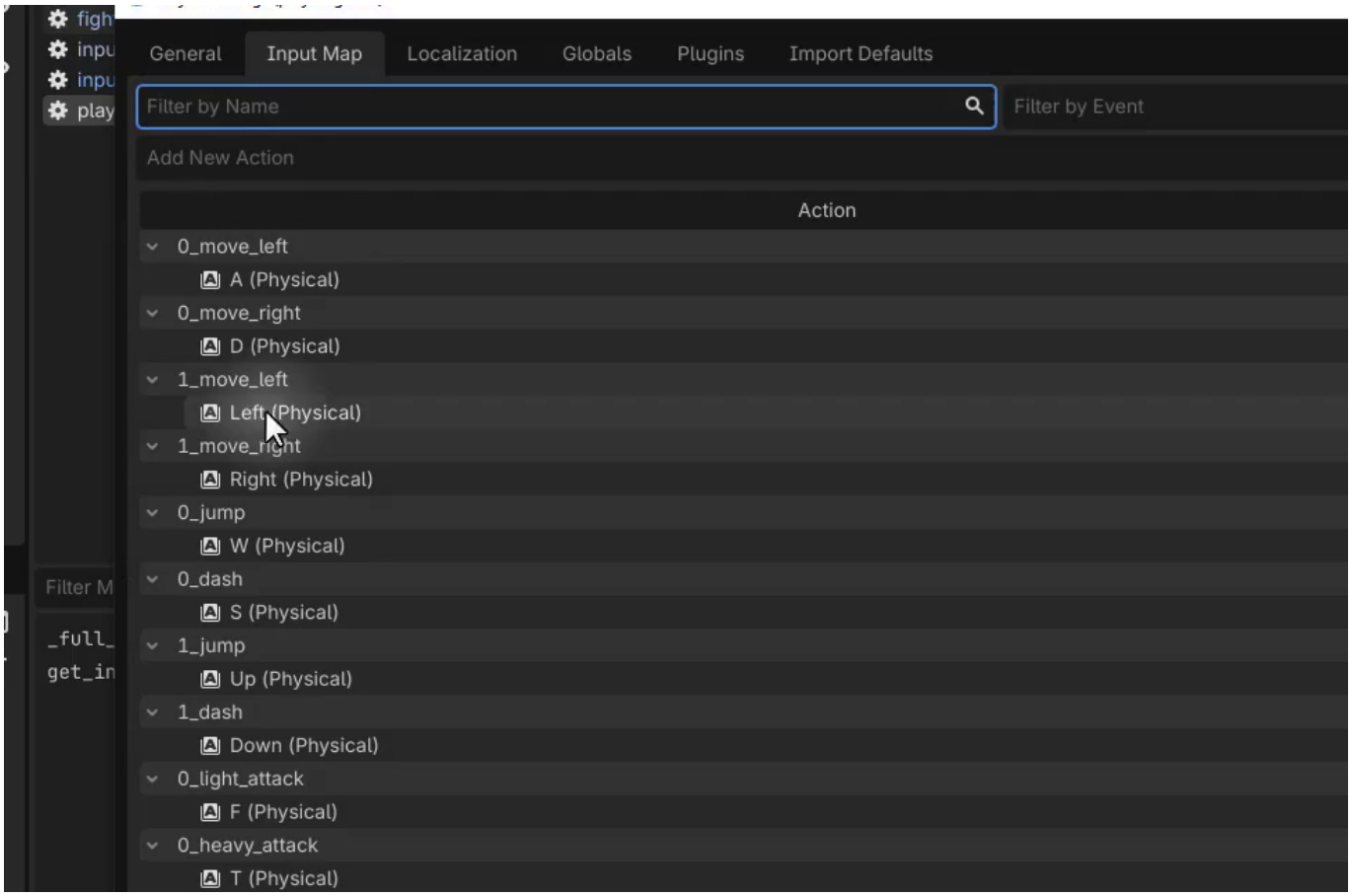
The completed get_input_packet() function looks like this:

```
func get_input_packet () -> InputPacket:
 var packet : InputPacket = InputPacket.new()

 packet.inputs["move_left"] = Input.is_action_pressed(_full_action_name("move_left"))
 packet.inputs["move_right"] = Input.is_action_pressed(_full_action_name("move_right"
))
 packet.inputs["jump"] = Input.is_action_pressed(_full_action_name("jump"))
 packet.inputs["dash"] = Input.is_action_pressed(_full_action_name("dash"))
 packet.inputs["light_attack"] = Input.is_action_pressed(_full_action_name("light_att
ack"))
 packet.inputs["heavy_attack"] = Input.is_action_pressed(_full_action_name("heavy_att
ack"))

 return packet
```

How the Input System Works

Each time get_input_packet() is called, a new InputPacket is created. The InputPacket class contains a dictionary of all possible inputs, each defaulting to false. The function then checks Godot's Input system for each action and updates the dictionary accordingly.

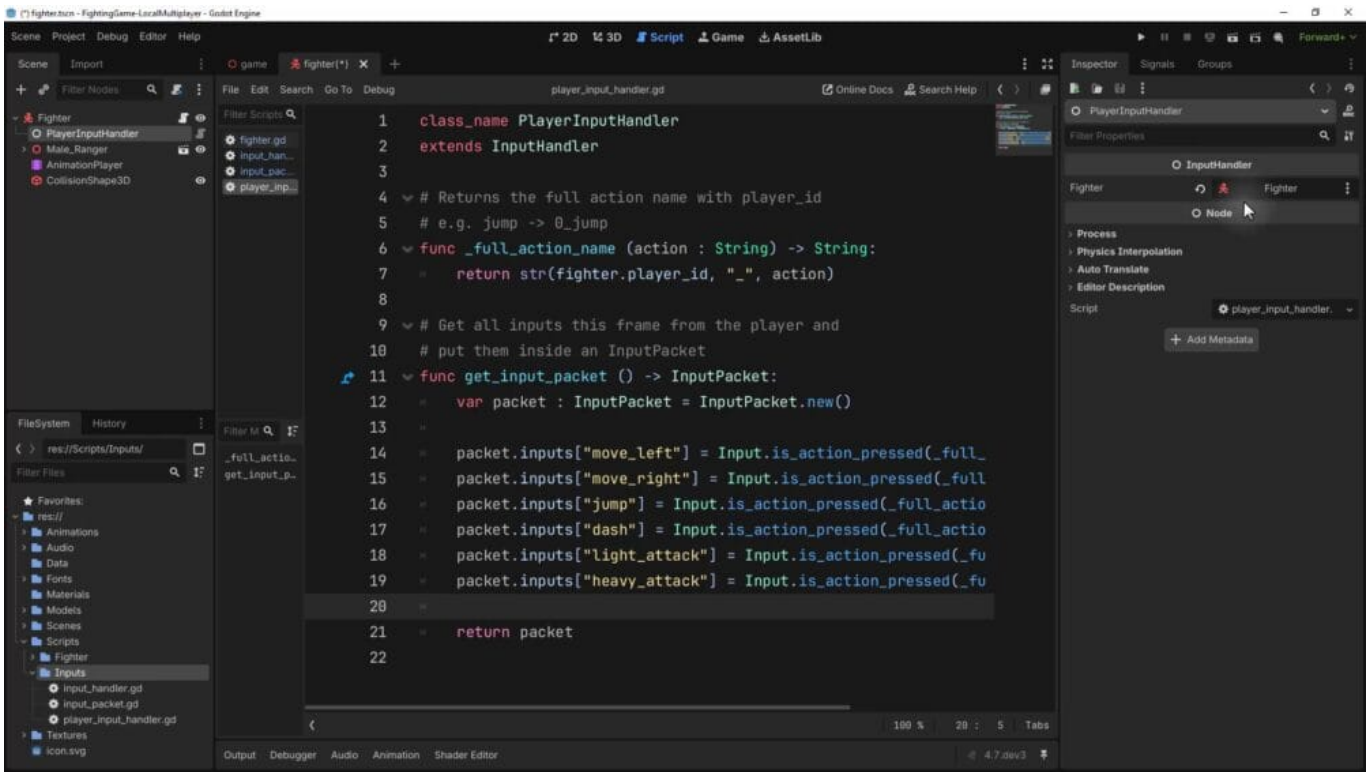
The helper method _full_action_name() prepends the fighter's player_id to each action string. For example, if the fighter's player_id is 0, calling _full_action_name("jump") returns "0_jump". This corresponds directly to the action names configured in Godot's Input Map, where each action is prefixed with the player index.



Player 1 (index 0) uses keys such as A, D, W, S, F, and T, while Player 2 (index 1) uses the arrow keys and other designated keys. This prefix system is what allows the same PlayerInputHandler script to work for both players — only the player_id changes.

Adding the PlayerInputHandler Node

With the script complete, the next step is to add a PlayerInputHandler node to the Fighter scene. Open the Fighter scene, click “Add Child Node,” and search for PlayerInputHandler. After adding it, move it to the top of the node list for organizational clarity. The PlayerInputHandler node has an exported Fighter property that needs to be assigned. Drag the root Fighter node from the Scene tree into this field in the Inspector panel.



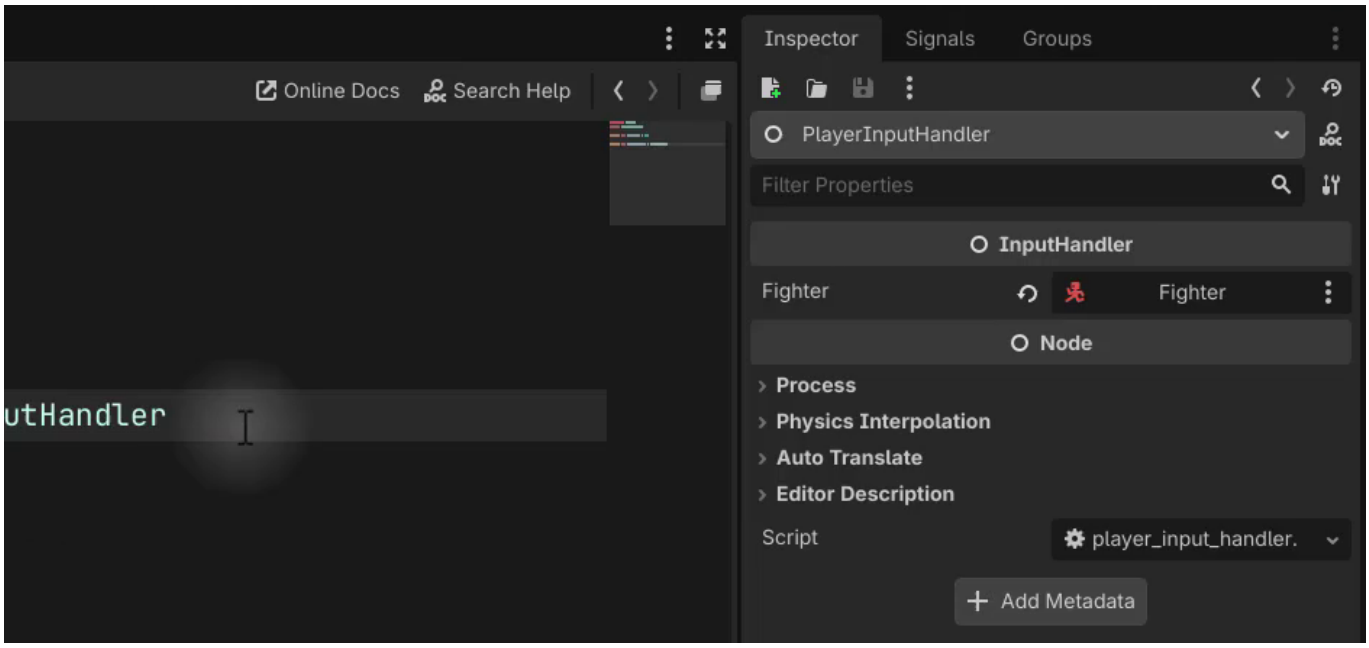
Connecting the Input Handler to the Fighter Script

At this point, the PlayerInputHandler exists as a node in the scene, but nothing calls its get_input_packet() function yet. To fix this, open fighter.gd and add a new exported variable:

```
@export var input_handler : InputHandler
```

The variable type is InputHandler (the base class) rather than PlayerInputHandler specifically. This is an intentional design choice — in the future, AI-controlled fighters or networked fighters will have their own input handler scripts that also extend InputHandler. By using the base type, any input handler can be assigned to a fighter, and it will work automatically.

After saving the script, select the Fighter node in the editor and drag the PlayerInputHandler child node into the newly appeared Input Handler property in the Inspector.



The Physics Process Function

The fighter's game logic runs inside _physics_process rather than _process. There are two reasons for this. First, movement uses Godot's CharacterBody3D, which relies on physics-based movement via move_and_slide(). Second, fighting games typically run on fixed time steps or fixed frame rates, and _physics_process runs at a consistent rate. This makes it possible to define frame-specific durations, such as how long an attack lasts or how long a player remains stunned.

The _physics_process function will eventually handle five steps each frame:

1. Get the input packet from the input handler
2. Send that input packet to the input buffer
3. Process the input through the state machine
4. Update the fighter's facing direction
5. Process movement

These steps will be implemented over the course of several future lessons. For now, only the first step is added along with a temporary print statement for testing:

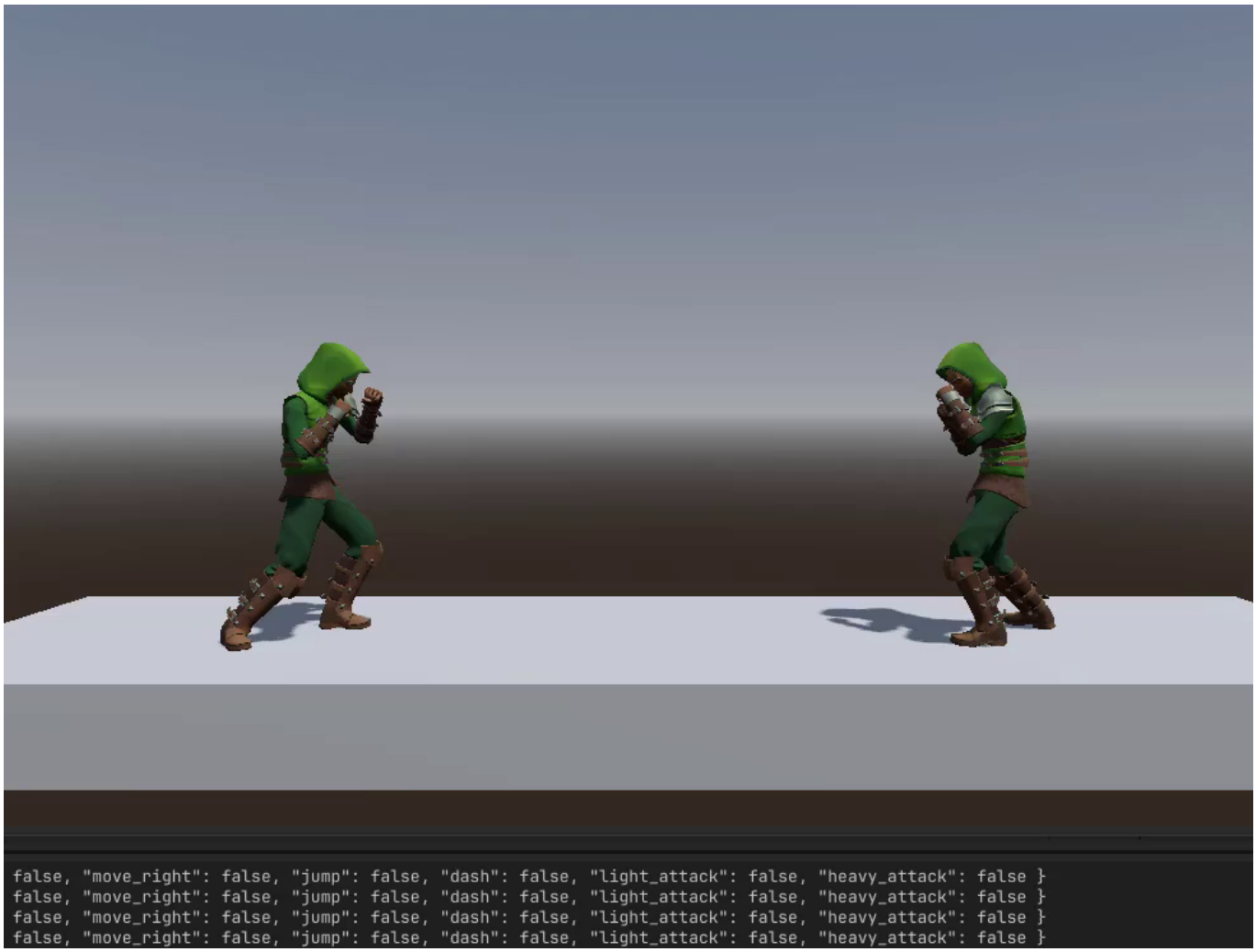
```
func _physics_process (delta : float):
 # 1. Get the input packet
 var input_packet : InputPacket = input_handler.get_input_packet()
 print(input_packet)
```

Testing the Input System

Before running the game, make sure to set the player_id values in the main scene. Select the first fighter and set its Player Id to 0, then select the second fighter and set its Player Id to 1. This ensures each fighter reads from the correct set of input actions.

When the game runs, the output console prints a dictionary of input states for each fighter every frame. Initially, all values are false. Pressing Player 1's movement key (A for left, D for right) sets the corresponding value to true in that fighter's output, while Player 2's values remain unaffected. The

same applies in reverse when using Player 2’s arrow keys.



All six actions — move_left, move_right, jump, dash, light_attack, and heavy_attack — can be verified independently for each player. Pressing both players’ keys simultaneously confirms that both inputs are tracked separately and correctly.

After confirming that everything works, the temporary print line can be removed from _physics_process, as it is no longer needed. In the next lesson, the input buffer system will be built to store these input packets in a rolling buffer for use by the state machine.

In this lesson, we are going to create the **InputBuffer** node for our fighter. Each fighter will own an input buffer, and every time the input handler produces an input packet, that packet will be sent to the buffer to be stored. The buffer keeps a short history of the most recent frames of input, which is what allows the controls of our fighting game to feel smooth and responsive.

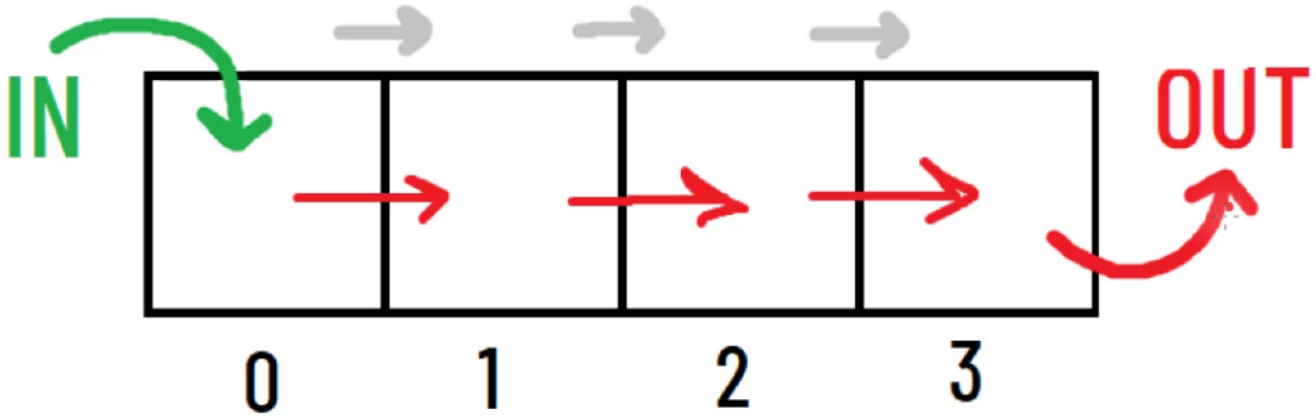
Why do we need an input buffer?

Imagine you are playing the game, throwing a punch. The punch animation plays, and once it ends, the fighter returns to its default state — the state in which inputs are detected and new actions can be triggered. Now suppose that as the punch animation is still pulling back, a couple of frames before the fighter actually resets to that default state, you press the punch button again. Without a buffer, that second press would simply be missed, and no follow-up attack would happen.

That kind of strict, frame-perfect input detection does not feel very fluid. To fix it, many games use a *rolling buffer* where the inputs from the last several frames are remembered. When the fighter returns to a state that can accept new actions, it can look back into the buffer and act on any input that was pressed very recently. The result is a bit more leeway for the player and much smoother-feeling controls.

How the rolling buffer works

Our input buffer is going to be an array. Each element of the array represents one frame of input, and each element holds an InputPacket — the same packet object produced by the input handler. Every frame, we generate a new input packet and push it to the front of the array. Everything else in the array shifts down by one position, and whatever was at the end of the array is dropped.



The diagram above uses only four elements for clarity, but the actual buffer we are going to build will store the last 20 frames of inputs. As the game runs, frame 1's packet enters the buffer, then frame 2's packet pushes it back one slot, and so on, until the buffer fills up. From that point on, each new packet that enters the front evicts the oldest packet from the back.

Creating the InputBuffer script

Inside the **Scripts/Inputs** folder, create a brand new script called input_buffer.gd. We want to be able to reference this type from other scripts, so give it a class name of InputBuffer. It extends Node so that we can drop it into our Fighter scene as a child node.

```
class_name InputBuffer
extends Node
```


Next, we declare the buffer itself as a variable. It is an array of InputPacket objects, initialized to an empty array.

```
var buffer : Array[InputPacket] = []
```

Defining the buffer size as a constant

The maximum size of the buffer is defined using a **constant** variable. A constant is a value that, once you define it, cannot be changed — it is essentially compiled into the script as a static value. We will use 20 as our buffer size, meaning the buffer will always hold the last 20 frames of inputs.

```
const BUFFER_SIZE : int = 20
```

Initializing the buffer in _ready

When the game first starts, we want the buffer to already be full of empty input packets. This way, any code that tries to read from the buffer will always find a valid packet there, and we do not have to worry about null checks when accessing elements. Inside the _ready function, we loop through the buffer size and push a fresh InputPacket to the front each time.

```
func _ready ():
 for i in range(BUFFER_SIZE):
  buffer.push_front(InputPacket.new())
```

Receiving new input packets

The fighter’s input handler will call a function on the buffer every frame, handing it the newly generated input packet. We create a receive_input function that accepts the packet and adds it to the front of the array using push_front. Pushing to the front is the behaviour shown in our diagram: the new packet lands at index 0, and everything already in the buffer shifts one slot toward the back.

After pushing the new packet, we need to check whether the buffer has grown beyond its fixed size. If it has, we remove the element at the back using pop_back. In our model, push_front is the “in” side of the buffer and pop_back is the “out” side.

```
func receive_input (packet : InputPacket):
 buffer.push_front(packet)

 if buffer.size() > BUFFER_SIZE:
  buffer.pop_back()
```

Checking the current frame with is_pressed

With the buffer in place, we can now add some helper functions that other parts of the fighter will call when they want to query the player’s inputs. The first is is_pressed, which takes an input name as a string and returns a boolean indicating whether that input is pressed *this frame*.

To answer that, we only need to look at the most recent packet in the buffer — the element at index 0 — and ask it whether the input is pressed. Remember that the InputPacket class already exposes

an is_pressed function that takes an input string and returns true or false.

```
func is_pressed (input_string : String) -> bool:
 var packet : InputPacket = buffer[0]
 return packet.is_pressed(input_string)
```

Getting the movement direction

Our fighters will move left and right using the move_left and move_right inputs. Rather than forcing the rest of the code to keep calling is_pressed for both actions separately, we provide a convenience function that returns a single direction value: 0 if the fighter is not moving, 1 if moving right, and -1 if moving left.

We start with a local dir variable at zero, subtract one if move_left is pressed, add one if move_right is pressed, and return the result. A nice side effect of this logic is that pressing both directions at the same time cancels out and returns zero, so neither direction gets priority over the other.

```
func move_direction () -> int:
 var dir : int = 0

 if is_pressed("move_left"):
  dir -= 1
 if is_pressed("move_right"):
  dir += 1

 return dir
```

Checking recent frames with was_pressed

Finally, we create the function the entire buffer was built for: was_pressed. Whereas is_pressed only looks at the most recent packet, was_pressed searches through a window of the most recent frames to see if the input was pressed at any point during that window.

The function accepts the input string to look for and a buffer_window integer that defaults to 3, meaning it checks the last three frames. Inside, we loop over the range of frames we want to inspect. Because we cannot be certain that the buffer is larger than the requested window, we take the minimum of buffer_window and buffer.size() to avoid indexing out of bounds. For each packet in that window, if the input is pressed, we return true immediately; otherwise, once the loop ends, we return false.

```
func was_pressed (input_string : String, buffer_window : int = 3) -> bool:
 for i in range(min(buffer_window, buffer.size())):
  if buffer[i].is_pressed(input_string):
   return true

 return false
```

Plugging the buffer into the Fighter

With the script complete, we now need to wire the buffer up inside our fighter. Open fighter.gd and,

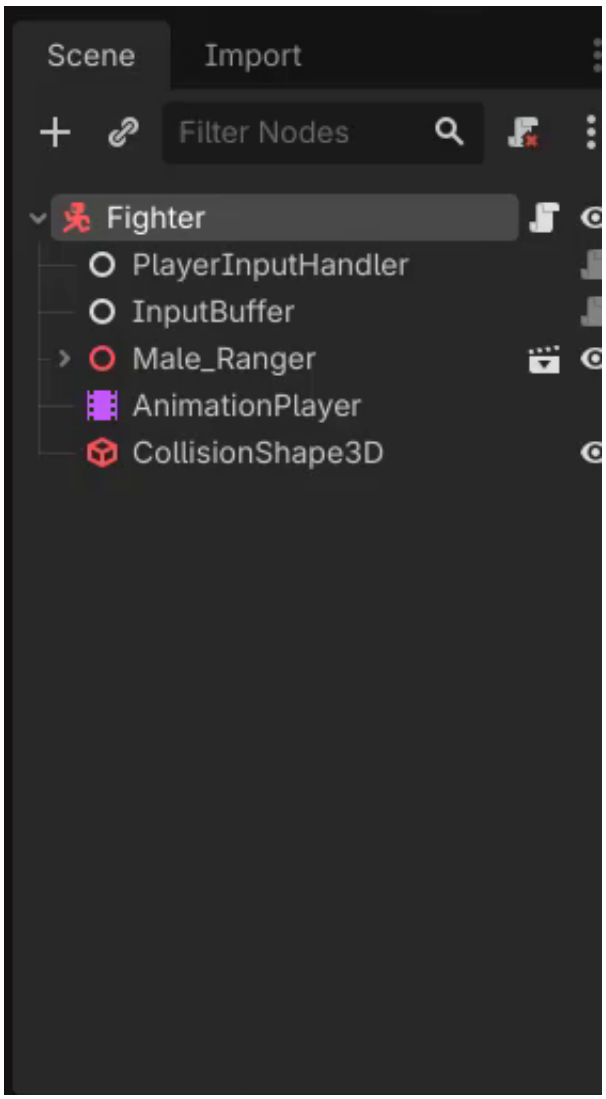
near the top where the input handler is declared, add another exported variable for the input buffer:

```
@export var input_buffer : InputBuffer
```

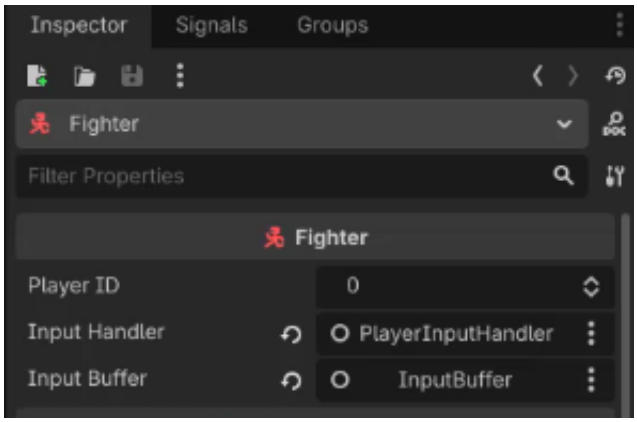
Then, inside _physics_process, the second step of the flow — after getting the input packet from the handler — is to hand that packet off to the buffer so it gets stored:

```
# 2. Receive the input in the buffer
input_buffer.receive_input(input_packet)
```

Back in the Fighter scene, create a new **InputBuffer** node as a child of the Fighter and drag it into place alongside the existing input handler.



Finally, assign that new node to the Input Buffer slot in the Fighter's Inspector so the exported variable is wired up.



Wrapping up

Our input system is now fully set up: every frame, the input handler produces a packet, and the input buffer stores it in a rolling 20-frame history that other systems can query for both instantaneous and recent inputs. We will not really see the benefits of this buffer until we start building the **State Machine**, which will act as the brain of the fighter — deciding what it does and when, based on the inputs we have just made available. In the next lesson, we will begin putting that state machine together.

Before jumping into Godot and writing any code for the fighter's behavior, it is worth slowing down and building a mental model of the system that will drive it: a **state machine**. This lesson is purely conceptual — it explains what a state machine is, what an individual state looks like, how the fighter's states will be organised, and how the whole thing will be laid out inside the Godot editor. The next lesson picks up where this one leaves off and begins implementing it.

What is a state machine?

A state machine is a way of breaking complex behavior down into a collection of distinct *states*. Instead of writing a single script that tries to manage moving, jumping, attacking, dashing, and everything else all at once — which quickly becomes unmanageable — each behavior lives in its own small, focused script. Moving has its own script, jumping has its own, attacking has its own, and so on.

A key rule of a state machine is that only one state can be active at any given time. If the fighter is jumping, they cannot also be attacking. If the fighter is walking around, they cannot also be dashing. What the fighter *can* do is transition from one state to another.

There are two common ways a transition happens:

- **Input-driven transitions** — the player presses a button while in one state, which triggers a move to another. For example, while in the moving state, pressing the attack button transitions the fighter into the attack state.
- **Direct transitions** — state changes that are not tied to input. For example, if the fighter is standing still and gets hit, they automatically enter the hit state. Or once an attack animation finishes playing, the fighter automatically returns to the default standing state.

What a state contains

Each state is a self-contained unit that processes its own behavior and stores its own data. Although it lives in isolation from the other states, it can still communicate with outside systems — the root fighter node for moving the character around, the animation system for playing or changing animations, the visual system for changing appearance, the stamina values for reading or subtracting resources, and the input system for detecting what the player is pressing.

Every state in this project will implement four functions that together form its lifecycle:

- **can_enter** — returns true or false depending on whether the state is currently allowed to be entered. For example, if the player is spamming the attack button, the attack state might use this to enforce a cooldown and refuse entry until that cooldown has elapsed. It can also check things like stamina — if the fighter's stamina is too low, can_enter returns false and the attack simply does not start.
- **enter** — called once when the state becomes active. This is where the one-off setup happens. For the attack state, for example, the first thing enter does is play the attack animation.
- **exit** — called once when the state is being left. Any code or values that need to be reset on the way out go here.
- **update** — essentially the _process function for the state. It runs every frame while the state is active and is where ongoing logic lives, including detecting player inputs that might trigger a new transition.

How the state classes are organised

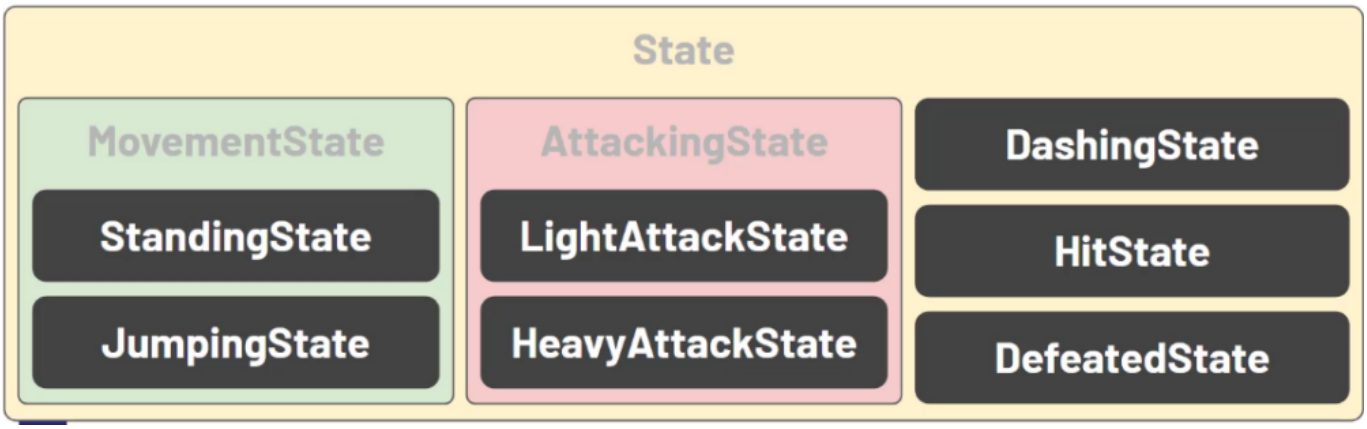
The fighter's states are not all built from scratch — they share a lot of behavior, so the project uses inheritance to avoid repeating code. Everything starts from a base **State** class that defines the four

lifecycle functions above.

From there, a **MovementState** class inherits from State. MovementState is not itself a state the fighter ever transitions into — it acts as a shared parent for both the **StandingState** and the **JumpingState**, since both of them need the same code for moving left and right along the ground.

Similarly, an **AttackingState** parent class holds the logic shared by the **LightAttackState** and **HeavyAttackState**, since the two attacks work in essentially the same way — only their values differ.

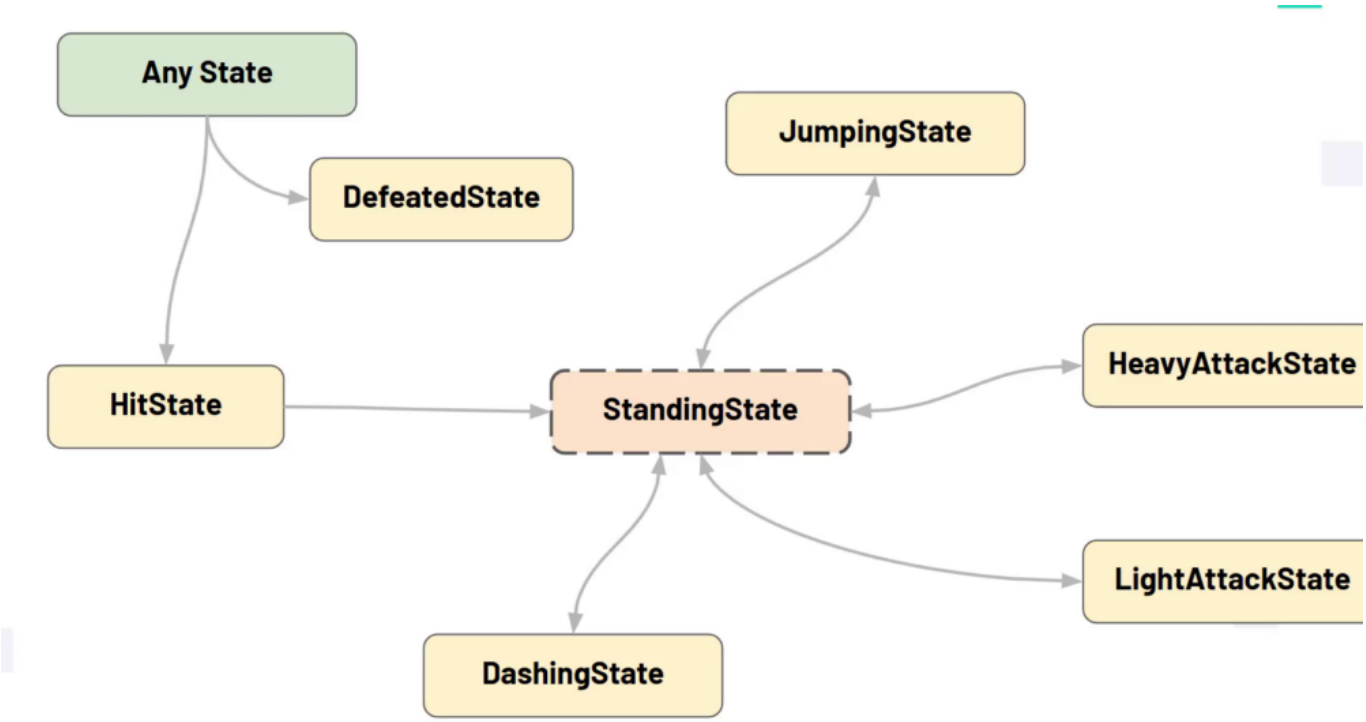
Finally, the **DashingState**, **HitState,** and **DefeatedState** inherit directly from the base State class, since they do not share code with the movement or attacking families.



The state transition diagram

With the class structure in place, the transitions between states can be visualised as a diagram. **StandingState** sits at the centre as the default state and effectively encapsulates all grounded movement and idle behavior. From standing, the fighter can transition into the jumping, dashing, light attack, or heavy attack states whenever the corresponding input is pressed. Each of those states has an arrow pointing back to standing — meaning once the action is complete (the jump lands, the dash finishes, the attack animation ends), the fighter automatically returns to standing.

On the side of the diagram sits a special **Any State** node. Any State is not a real state of its own; it is a shorthand meaning “from any currently active state”. It connects to the **DefeatedState** and the **HitState**, because those two should be enterable from anywhere. It makes sense that if the fighter is standing and gets defeated, they enter the defeated state — but the same is true if they are mid-jump when their health reaches zero. Likewise, for the hit state: it can interrupt any other state. Once the hit state finishes, the fighter transitions back to standing.

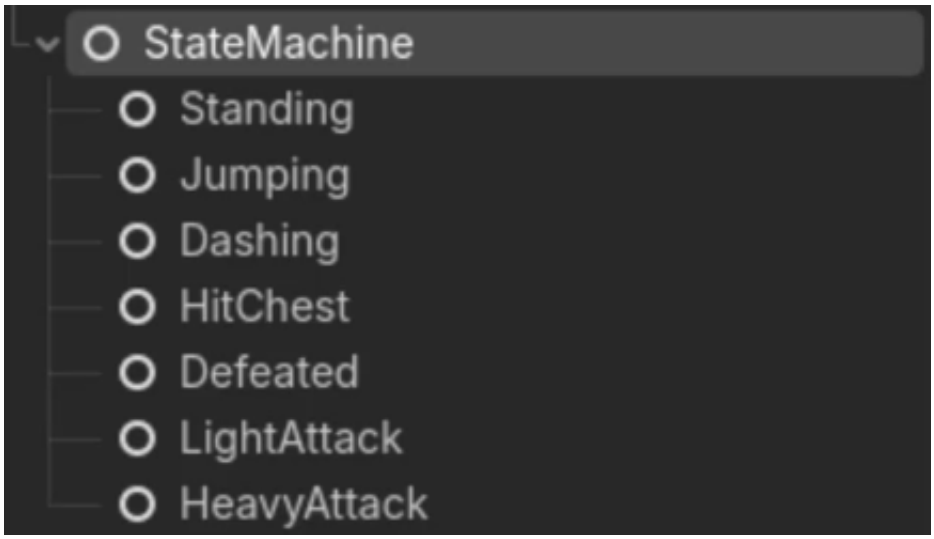


This diagram is worth keeping in mind throughout the rest of the course. Since the state machine will be entirely driven by code, having a clear picture of how each state connects to the others makes it much easier to reason about what should happen at any point during gameplay.

How it will be set up inside Godot

Inside Godot, the state machine maps cleanly onto the scene tree. The plan is to have a single **StateMachine** node that owns the logic for transitioning between states and setting up the default starting state. Each individual state — Standing, Jumping, Dashing, HitChest, Defeated, LightAttack, HeavyAttack — is then attached as its own child node underneath the StateMachine node.

This layout makes the state machine very visual in the editor: clicking on any state node reveals its properties in the inspector (duration, damage, cooldowns, and so on), while the script attached to each node holds the actual logic. Changes can be made either by tweaking inspector values or by jumping into the script, without having to hunt through one monolithic file.



What comes next

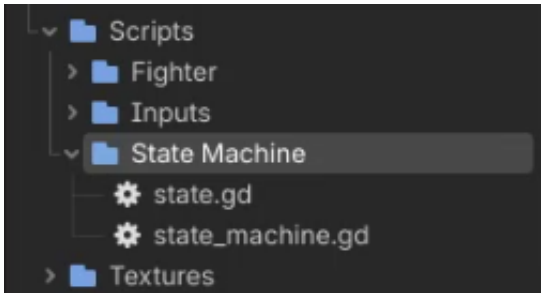
With the concept, the class hierarchy, the transition diagram, and the target Godot layout all covered, the groundwork for the state machine is in place. The next lesson jumps back into Godot and begins actually setting it up — starting from the base State class and building upward towards the full set of fighter states.

In this lesson, we begin implementing the state machine that will drive our fighter in Godot. The goal of this first part is to create the two foundational scripts — a generic **State** template and a **StateMachine** manager — and to wire them up so that the machine can hold a collection of states and transition between them at runtime.

Organising the scripts folder

Before writing any code, we create a dedicated folder for everything state-machine-related. Inside the project's *Scripts* directory, we add a new subfolder called **State Machine** and then create two empty scripts inside it:

- **state.gd** — the base template that every concrete state will inherit from.
- **state_machine.gd** — the manager that keeps track of all the states and controls which one is currently active.



The State base class

A state represents the logic and data for a single behavior that a fighter can be in — standing, jumping, attacking, being defeated, and so on. Because every concrete state will share the same core structure, we start by building a template class that they will all inherit from.

Open **state.gd** and give it a class name so other scripts can reference the type directly. It extends Node so that individual state instances can live in the scene tree as children of the state machine:

```
class_name State
extends Node
```

Next, we add a reference to the state machine that owns this state. This is what allows a state to request a transition — for example, the jumping state will tell the state machine to switch back to standing the moment the fighter touches the ground. For this to compile, the companion **state_machine.gd** script needs a class name as well, so jump over to it briefly and add class_name StateMachine at the top before continuing.

We also add two timing variables: enter_time, which records when the state became active, and local_time, which measures how many seconds have elapsed since then. These will be useful later on for states that run for a fixed duration, such as an attack that should transition back to standing after its animation finishes.

```
class_name State
extends Node

var state_machine : StateMachine
```



```
# Time we entered this state
var enter_time : float

# Time that has elapsed since entering this state
var local_time : float
```

Initialising a state

Each state needs a way to learn about the state machine that owns it. We expose an initialize function for this purpose, which will be called by the state machine when the game starts. Inside the function, we store the passed-in reference on the instance. Because the parameter and the variable share the same name, we use self to disambiguate the assignment:

```
# Called when the fighter is loaded in
func initialize (state_machine : StateMachine):
 self.state_machine = state_machine
```

Lifecycle functions

Every state shares the same set of lifecycle hooks. We provide default implementations here in the base class so concrete states only need to override the ones they care about.

can_enter decides whether it is currently valid to transition into this state. By default, any state can be entered, so it returns true:

```
# Returns true if we can enter this state
func can_enter () -> bool:
 return true
```

enter runs on the frame the state becomes active. The base implementation records the current time so that local_time can be calculated later:

```
# Called when this state first becomes active
func enter ():
 enter_time = Time.get_unix_time_from_system()
```

exit runs when the state is being left. The default does nothing, but concrete states can override it to clean up:

```
# Called when this state is exited
func exit ():
 pass
```


update is our own per-frame tick. We deliberately avoid Godot's built-in _process here because we do not want every state to tick every frame — only the one that is currently active. The state machine will call this function on the current state instead. Inside, we compute local_time as the difference between the current time and the time we entered the state:

```
# Called each physics frame this state is active
func update (delta : float):
 var time : float = Time.get_unix_time_from_system()
 local_time = time - enter_time
```

That completes the base template. Concrete behaviors such as movement, jumping, dashing or attacking will each live in their own script that inherits from this State class and overrides the appropriate lifecycle functions.

The StateMachine manager

With the template in place, we move over to **state_machine.gd** and begin building the manager. The state machine is responsible for discovering all of the state nodes that sit underneath it, holding on to them, and switching between them at runtime.

We start with some exported variables so the machine can be configured from the Inspector. The starting_state determines which state the fighter will be in when the game starts, and we also expose references to the systems that states might need to reach into — the fighter itself and the input buffer — with animation and stamina controller references planned for the future.

```
class_name StateMachine
extends Node

# Exported variables for configuration in the editor
@export var starting_state : State  # The initial state when the state machine starts

@export var fighter : Fighter
@export var input_buffer : InputBuffer
```

Storing the states

Next, we need somewhere to keep the set of available states. A dictionary keyed by the state's name is ideal because transitioning to another state then only requires passing in a string. We also add a variable for the state that is currently active:

```
var states : Dictionary[String, State] = { }  # Dictionary mapping state names to Sta
te objects
var current_state : State                     # Currently active state
```

Discovering child states in _ready

The state machine populates its dictionary automatically at the start of the game. Inside _ready, we loop over the node's children, skip anything that is not a State, and register the rest. Each discovered state is added to the dictionary under its node name and is initialised with a reference back to the state machine:

```
func _ready ():
 # Find all states that are children nodes of this one
 for child in get_children():
  if child is not State:
   continue

  states[child.name] = child
  child.initialize(self)
```

To verify that this discovery logic has something to find, we hop back into the Fighter scene and add a **StateMachine** node as a child. In its Inspector, we drag the Fighter and the InputBuffer into the corresponding properties, then add a couple of placeholder State nodes underneath the state machine — one called **Standing** and another called **Attack**. These are throwaway examples for testing purposes; the real states will replace them in later lessons.

Changing states

We now need a function that other parts of the code can call to request a transition. change_state takes the name of the target state, verifies that it exists in the dictionary, exits the current state if there is one, activates the new state, and prints a debug line so we can see the transition in the Output panel:

```
# Called whenever we want to change our state
func change_state (state_name : String):
 # Return if the state doesn't exist
 if not states.has(state_name):
  printerr("Cannot change state to ", state_name, " as it doesn't exist!")
  return

 # Exit our current state
 if current_state:
  current_state.exit()

 # Enter the new state
```



```
current_state = states[state_name]
current_state.enter()

print("New State: ", state_name)  # Debug output for state transitions
```

The existence check with printerr protects us from typos in the state name, since trying to look up a missing key would otherwise cause a runtime error. The guard around current_state.exit() handles the very first transition, where there is nothing to exit out of yet.

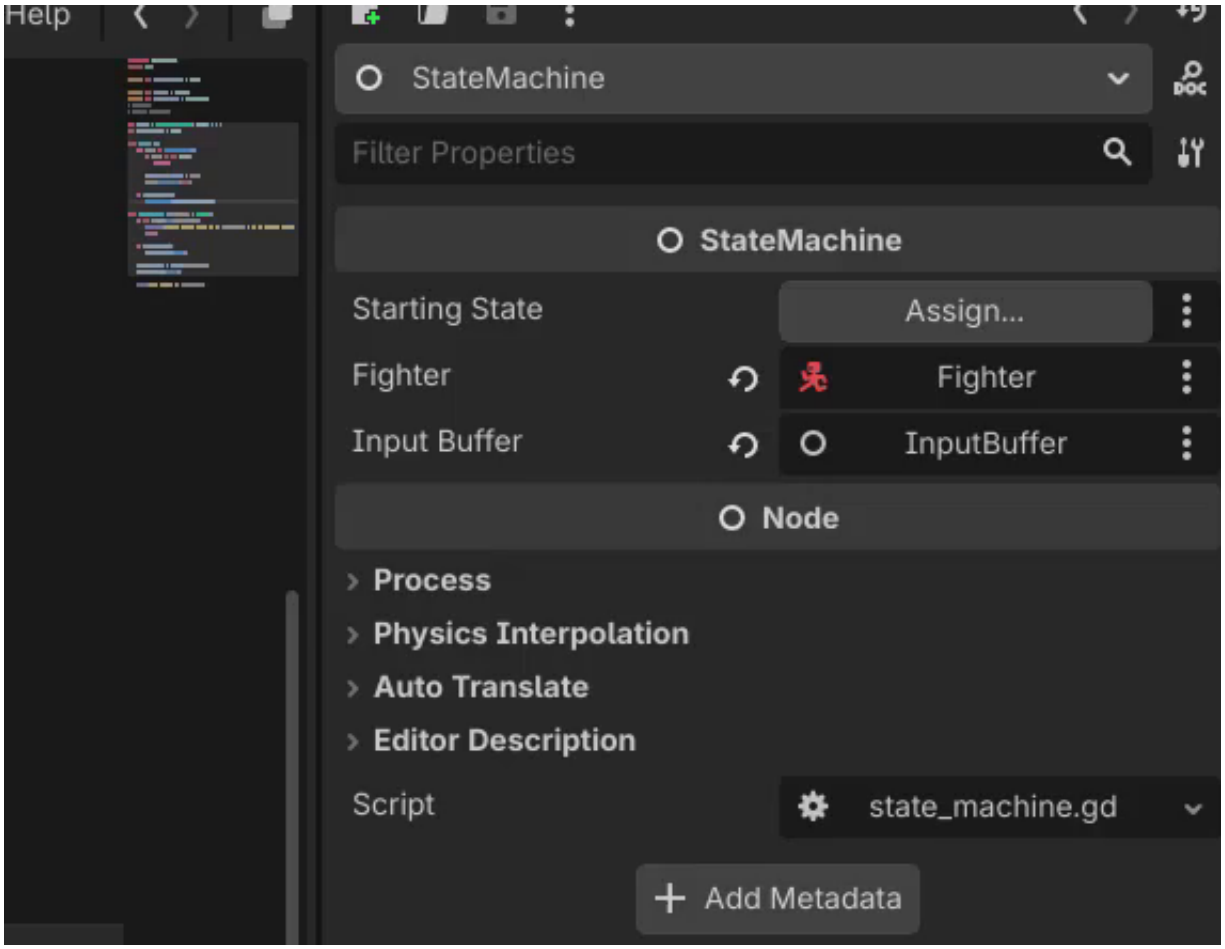
Kicking off the starting state

Finally, we extend _ready so that the state machine automatically transitions into its configured starting state once all its children have been discovered and initialised:

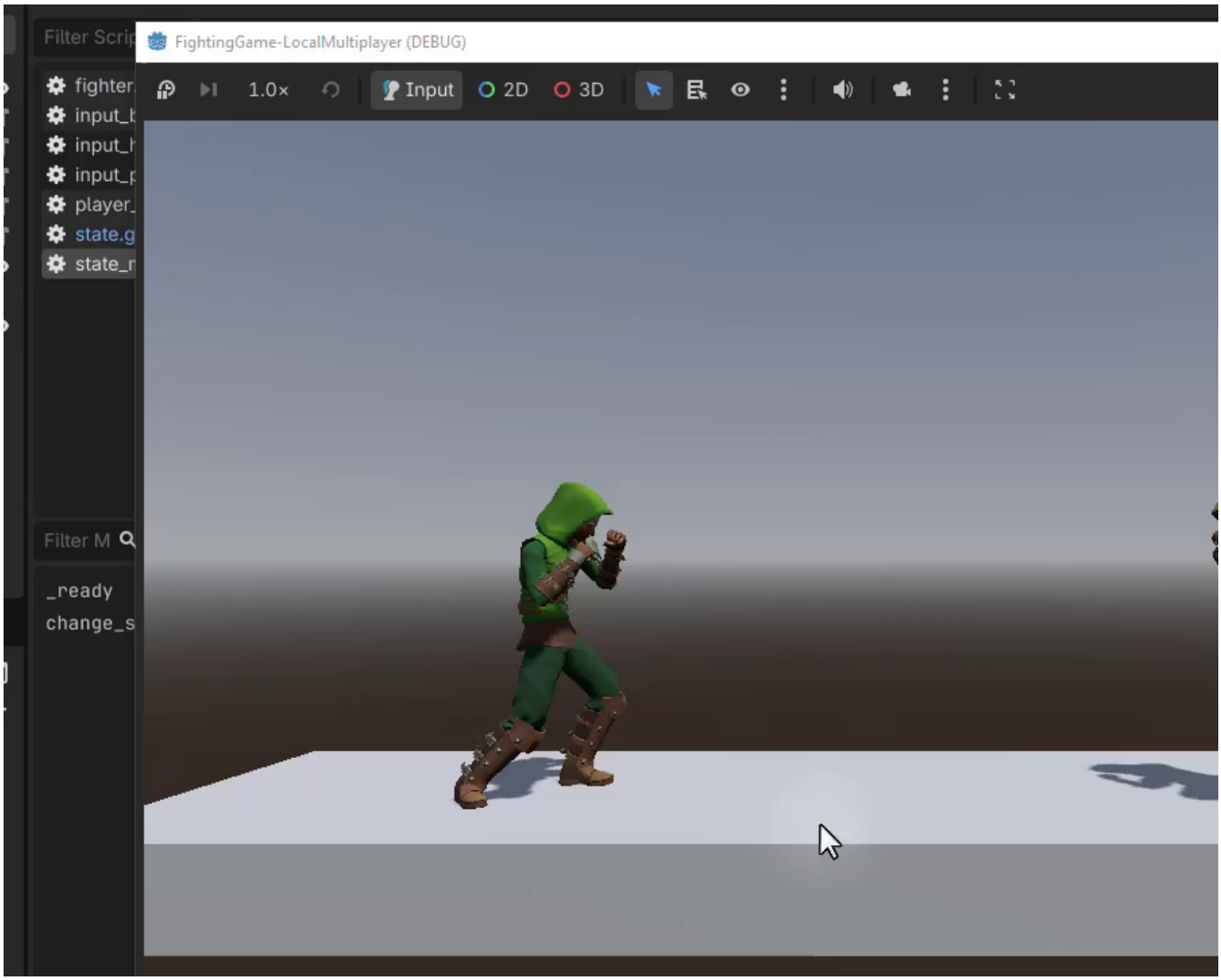
```
# Set the starting state (Standing)
if starting_state:
 change_state(starting_state.name)
```

Testing the transition

Back in the editor, we select the StateMachine node, drag **Standing** into the *Starting State* slot in the Inspector, and press Play. The Output panel prints **New State: Standing**, confirming that the state machine discovered its children, initialised them, and transitioned into the configured starting state.



Swapping the starting state over to **Attack** and pressing Play again prints **New State: Attack** instead, proving that the lookup and transition logic both work correctly.



What's next

At this point, the state machine can hold a collection of states and switch between them on command. In the next lesson, we will continue building out the state machine script and then connect it up to the fighter so that the state transitions actually drive the character's behavior.

In this lesson, we continue building the custom state machine for our fighter. In the previous lesson, we set up the _ready function — which collects all child states and stores them in a dictionary — and the change_state function, which handles transitions from one state to another. We also have a starting state being assigned in the Inspector so that the state machine knows which state to begin in. To finish off the state machine, we still need two more functions: one that updates the current state every frame, and a small helper function for checking which state is currently active. After that, we will test the whole system by switching between two states using a button press, and then do a small refactor to make writing states more convenient.

Adding the update Function

If you look at the State base class from the previous lesson, you will notice it already has an update function, but right now nothing is calling it. We want that function to run every frame so that each state can perform per-frame logic. To do this, we add an update function on the state machine itself.

You might expect us to use Godot's built-in _process or _physics_process callback, but we are not going to use either. Our game runs on a fixed time step, so _process is out. And we are deliberately avoiding _physics_process on the state machine as well, for a reason we will get to in a moment. Instead, we create our own custom function called update that takes a delta value:

```
# Called by the Fighter script every frame - update the current state
func update (delta : float):
 if not current_state:
  return

 current_state.update(delta)
```

The logic is simple: if for some reason we do not currently have an active state, we return immediately and do nothing. Otherwise, we forward the call to current_state.update(delta), letting whatever state is currently active handle its own per-frame behavior.

Why Not Use _physics_process Directly?

The reason we are not simply writing a _physics_process function on the state machine and calling current_state.update there comes down to **order of execution**. If we go back to the fighter.gd script, we can see that its own _physics_process function runs a set of steps in a very specific sequential order every physics frame:

1. Get the input packet from the input handler attached to this fighter.
2. Send that input packet into the input buffer.
3. Process the input through the state machine.
4. Update the facing direction.
5. Process movement.

These steps need to run in order. In particular, the state machine needs to run *after* the input has been fetched and written into the input buffer, so that the state machine reads the most recent input when it decides what to do. If we put a _physics_process on the state machine node directly, Godot would call it independently, and it would not be obvious whether the state machine runs before or after the fighter has processed its input on any given frame. We might end up with the state machine running, then the new input being registered — which is the opposite of what we want.

By exposing our own update function on the state machine and having the fighter call it manually in

its sequential steps, we keep full control over when the state machine runs. Inside the fighter's _physics_process, that call looks like this:

```
state_machine.update(delta)
```

And to make that call possible, we add a new export on the fighter so we can assign the state machine in the Inspector:

```
@export var state_machine : StateMachine
```

Now the fighter fetches an input packet, adds it to the input buffer, and only then updates the state machine — guaranteeing the state machine always reads the latest input.

The is_current_state Helper

Back in the state machine script, we add one more small function. We will not use it right away, but it will be handy later whenever we want to check whether a particular state is currently active. It takes a state name as a string and returns a boolean:

```
# Returns true if the requested state is the current one
func is_current_state (state_name : String) -> bool:
 if not current_state:
  return false

 return current_state.name == state_name
```

If there is no current state for some reason, we return false. Otherwise, we compare the current state's name to the requested name and return the result. This gives us a clean, reusable way to ask "is the fighter currently in the Standing state?" without having to dig into current_state directly every time.

Testing State Transitions

To make sure everything works, we are going to temporarily edit the state.gd script so we can test switching between two states with a button press. These changes are throwaway — we will revert them at the end of the lesson because they are not part of the final design.

First, we add a temporary exported variable on the base State called go_to_state, which will let us pick a target state from the Inspector:

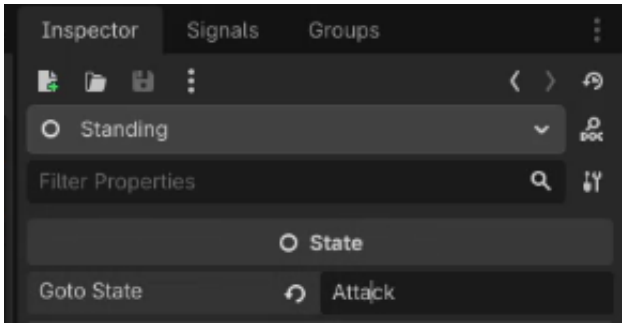
```
@export var go_to_state : String
```

Next, we need the state to be able to read inputs. Right now, the base state has access to the state

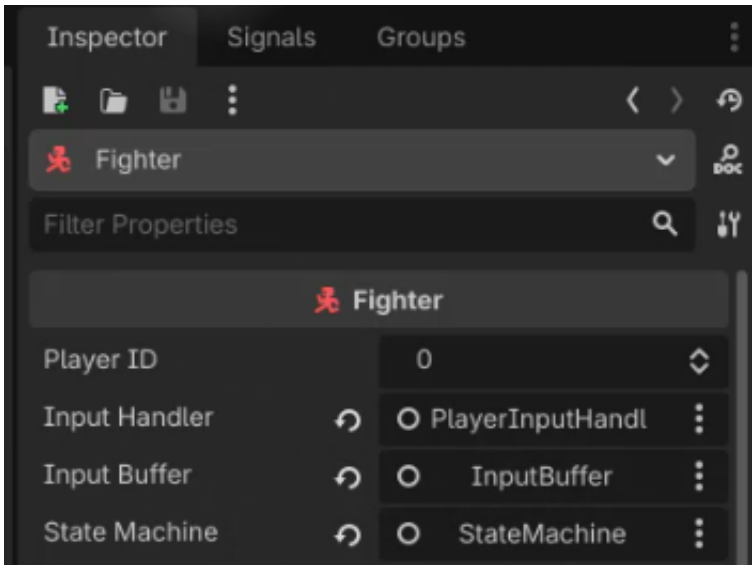
machine, and the state machine has references to both the fighter and the input buffer. That means from inside a state we can reach the input buffer via state_machine.input_buffer. We go to the update function of the state and add a check: if the jump input is pressed on this frame, transition to the state named in go_to_state:

```
func update (delta : float):
 var time : float = Time.get_unix_time_from_system()
 local_time = time - enter_time

 if state_machine.input_buffer.is_pressed("jump"):
  state_machine.change_state(go_to_state)
```

With this in place, we select the Standing state in the scene and set its go_to_state to Attack, then select the Attack state and set its go_to_state back to Standing. The result should be that pressing jump bounces the fighter between the two states.



We also need to make sure the Fighter node has its state_machine property assigned in the Inspector — drag the StateMachine node onto that slot.



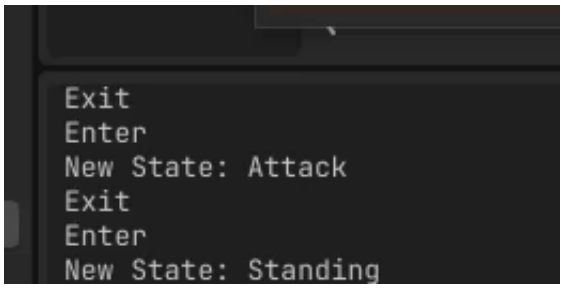
When we press Play and hold the W key (player 1's jump), the state machine rapidly transitions between Attack and Standing. Because the button is held down, the transitions happen many times per second. Doing the same with the Up arrow key triggers the same behavior for player 2.

To confirm that enter and exit are also being called correctly, we temporarily add print statements to

those functions:

```
func enter ():
 enter_time = Time.get_unix_time_from_system()
 print("Enter")

func exit ():
 print("Exit")
```

Running the game again and pressing the jump button shows the expected sequence of Enter, Exit, and “New State: Attack” / “New State: Standing” messages in the debugger output — confirming the full transition lifecycle is firing correctly for both players.



Once we have confirmed the behavior, we clean up: delete the if statement from the update function, remove the go_to_state export, and strip the temporary print calls. These were only for testing.

Cleaning Up Access With Property Getters

Looking at the test code, you can see that checking an input required writing state_machine.input_buffer.is_pressed("jump"). That is a lot of typing, and we are going to need to reach into the input buffer and the fighter from many states throughout the course. To make this cleaner, we can add small helper variables at the top of the State script that look like normal variables but are actually **property getters** — whenever we access them, they transparently return something else.

The syntax in GDScript is to declare a variable with its type, followed by a colon, and then a get block that returns the value we want:

```
var input_buffer : InputBuffer:
 get: return state_machine.input_buffer
```

Now, anywhere inside a state (or any script that inherits from State), we can simply write input_buffer.is_pressed("jump") and it will be exactly the same as writing state_machine.input_buffer.is_pressed("jump"). The getter fetches the value from the state machine behind the scenes every time we access it.

We do the same for the fighter reference, adding another getter property:

```
var fighter : Fighter:
```

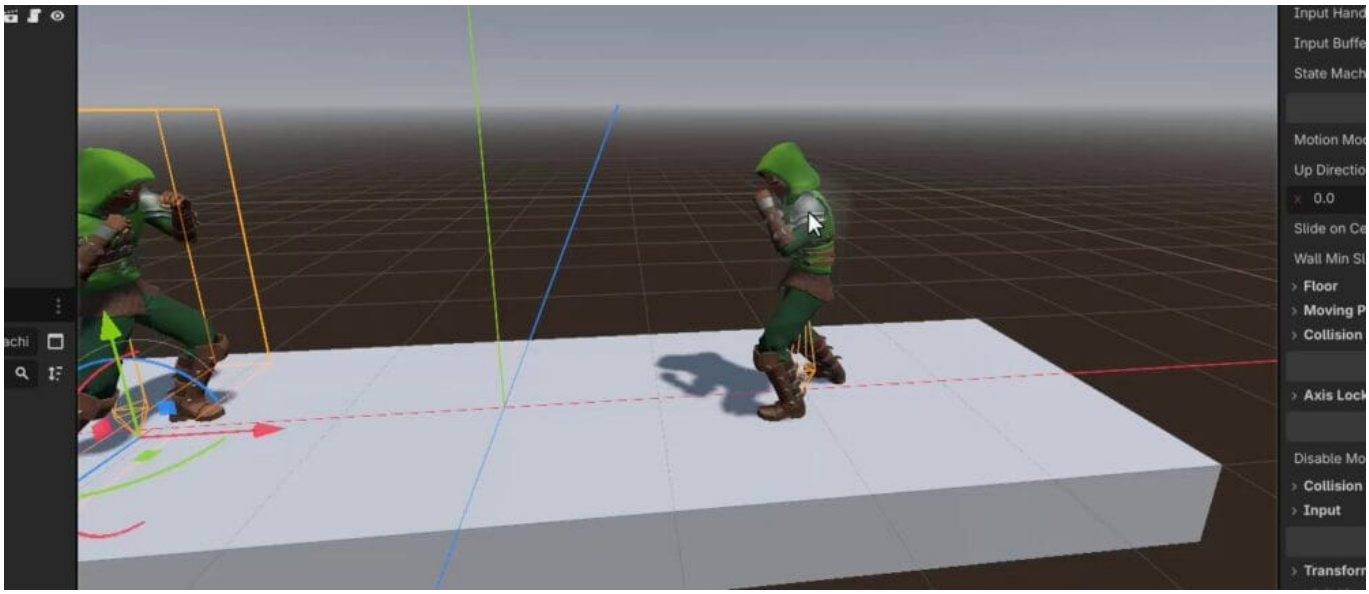


```
get: return state_machine.fighter
```

With these two helpers in place, writing new states will be much more concise — we can refer to fighter and input_buffer directly instead of constantly going through the state machine.

That wraps up our custom state machine. We now have an update function driven in a controlled order from the fighter's physics process, a helper for checking the active state, and clean getter-based access to the fighter and input buffer from any state. In the next lesson, we will start building actual state behaviors on top of this foundation.

With the fighter scene and input systems in place, the next step is to let each character actually move around the arena. Before worrying about running left or right, however, the fighters need a consistent idea of which way they are facing, so that their attacks and animations always point toward their opponent. This lesson adds that facing logic first, then introduces a simple movement state that drives horizontal motion through the existing state machine.



Defining forward direction

The idea of a forward direction is straightforward: the fighter on the left should always be pointing to the right, and the fighter on the right should always be pointing to the left. If, at any point, one fighter runs past the other, their forward direction needs to flip so that they continue to face each other.

Open the fighter.gd script and add a new non-exported variable called forward_direction of type int. A value of 1 represents facing right, while -1 represents facing left:

```
var forward_direction : int
```

For a fighter to know which way it should be facing, it also needs a reference to its opponent. Add an exported variable so that the opponent can be assigned through the Inspector:

```
@export var opponent : Fighter
```

Assigning the opponent in the scene

Back in the game scene, select the first fighter and drag the second fighter into the new *Opponent* slot in the Inspector. Do the same for the second fighter, using the first fighter as its opponent. With both references assigned, each fighter now has a live link to the node it should be tracking.

Updating the facing direction each frame

Next, create a function in fighter.gd that runs every physics frame and updates the fighter's orientation based on its opponent's position. If the fighter's global X position is less than its opponent's, it should face right; otherwise it should face left. The rotation is handled by setting rotation_degrees.y on the underlying CharacterBody3D:

```
func _update_facing_direction ():
    if global_position.x < opponent.global_position.x:
        forward_direction = 1
        rotation_degrees.y = 90
    else:
        forward_direction = -1
        rotation_degrees.y = -90
```

This function needs to be called every frame, after the state machine has been updated. Inside _physics_process, it fits in as the fourth step of the update loop:

```
func _physics_process (delta : float):
    # 1. Get the input packet
    var input_packet : InputPacket = input_handler.get_input_packet()

    # 2. Receive the input in the buffer
    input_buffer.receive_input(input_packet)
```

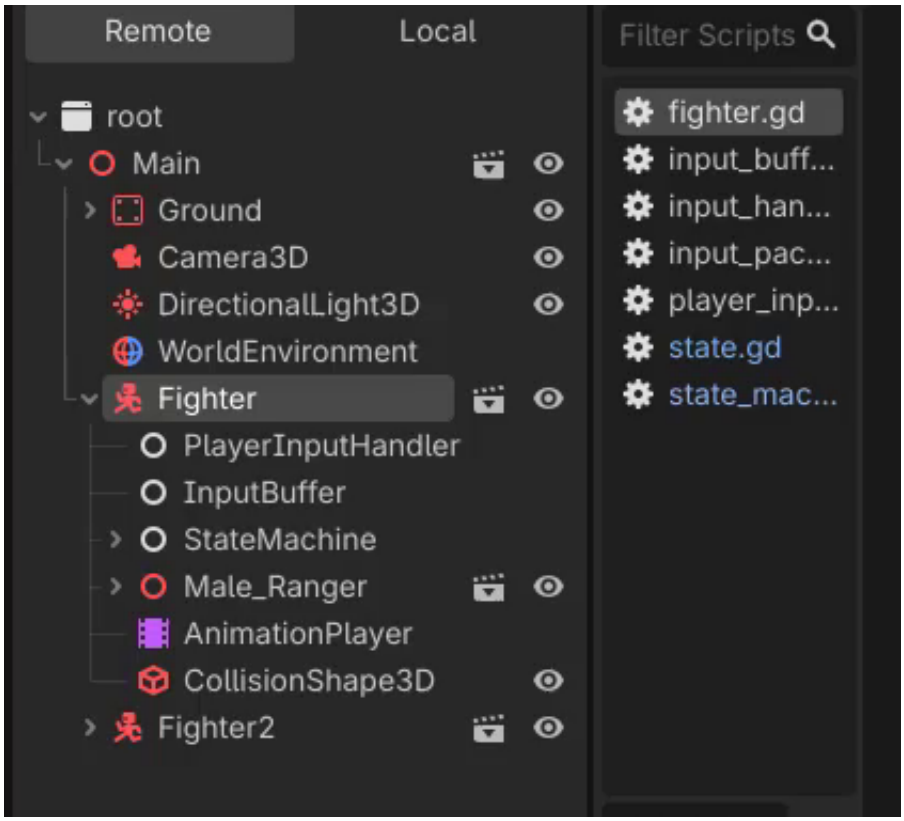


```
    # 3. Process the input in the state machine
    state_machine.update(delta)

    # 4. Update facing direction
    _update_facing_direction()
```

Testing facing direction with the Remote scene tree

Pressing play shows both fighters already correctly oriented toward each other, but to verify that the logic actually reacts to position changes, the fighters need to be moved at runtime. Godot makes this possible through the Scene dock's **Remote** mode.

With the game still running, click over to the Godot editor, open the Scene dock, and switch it from *Local* to *Remote*. This exposes the live scene tree of the running game, so any node can be selected and its properties edited on the fly.



Select a fighter in the remote tree, then use the Inspector to drag its position across to the opposite side of the arena. Once the fighter crosses past its opponent, the _update_facing_direction function flips its forward_direction and rotates it, and the fighters end up facing each other again from their new positions.

Adding move velocity and a movement function

With facing logic in place, it is time to give the fighters actual movement. Rather than writing directly to the CharacterBody3D's velocity from many different places, the fighter owns a dedicated move_velocity vector that states code can modify, and a single _movement function is responsible for applying it.

Back in fighter.gd, declare the new variable:

```
var move_velocity : Vector3
```

Then add a _movement function that copies the X and Y components of move_velocity onto the character body's built-in velocity and calls move_and_slide to apply it:

```
func _movement (delta : float):
    velocity.x = move_velocity.x
    velocity.y = move_velocity.y
    move_and_slide()
```

Finally, this function needs to be called from _physics_process as the fifth and final step in the update loop:


```
func _physics_process (delta : float):
    # 1. Get the input packet
    var input_packet : InputPacket = input_handler.get_input_packet()

    # 2. Receive the input in the buffer
    input_buffer.receive_input(input_packet)

    # 3. Process the input in the state machine
    state_machine.update(delta)

    # 4. Update facing direction
    _update_facing_direction()

    # 5. Process movement
    _movement(delta)
```

At this point, the full update loop is complete: inputs are gathered, buffered, interpreted by the state machine (which is where move_velocity will be set), the facing direction is refreshed, and then the final movement is applied.

Creating the movement state

The state machine still needs a state that actually modifies move_velocity based on player input. Since both grounded movement (standing) and air movement (jumping) will want the ability to move left and right, it makes sense to build a shared **MovementState** base class that both of those states can extend later.

Under the *State Machine* scripts folder, create a new script called movement_state.gd. It should extend the existing State class and expose a single exported move_speed variable controlling how fast the fighter walks horizontally:

```
class_name MovementState
extends State

@export var move_speed : float = 1.5
```

Because this state needs to do its own work every frame, it must override the update function from the base State class. Overriding simply means defining a function with the same name on the child class; whenever the state machine calls update on this state, Godot runs this implementation instead of the base one.

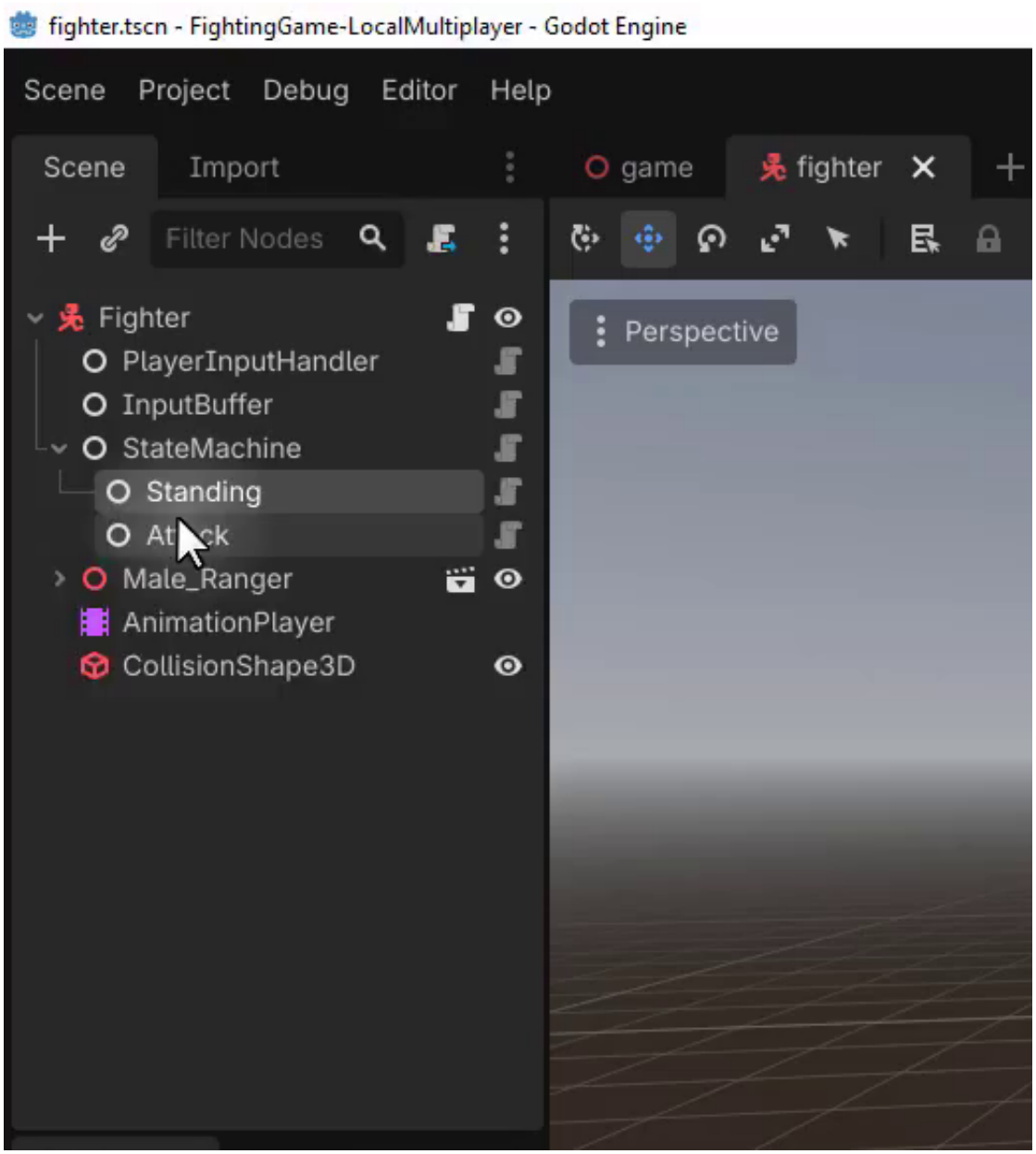
The base class still has important bookkeeping in its own update (such as tracking local_time), so the override needs to call super.update(delta) first to keep that behavior intact. After that, it reads the current move direction from the input buffer and writes the resulting velocity onto the fighter:

```
func update (delta : float):
    super.update(delta)

    # Set movement direction
    var move_dir : int = input_buffer.move_direction()
    fighter.move_velocity.x = move_dir * move_speed
```

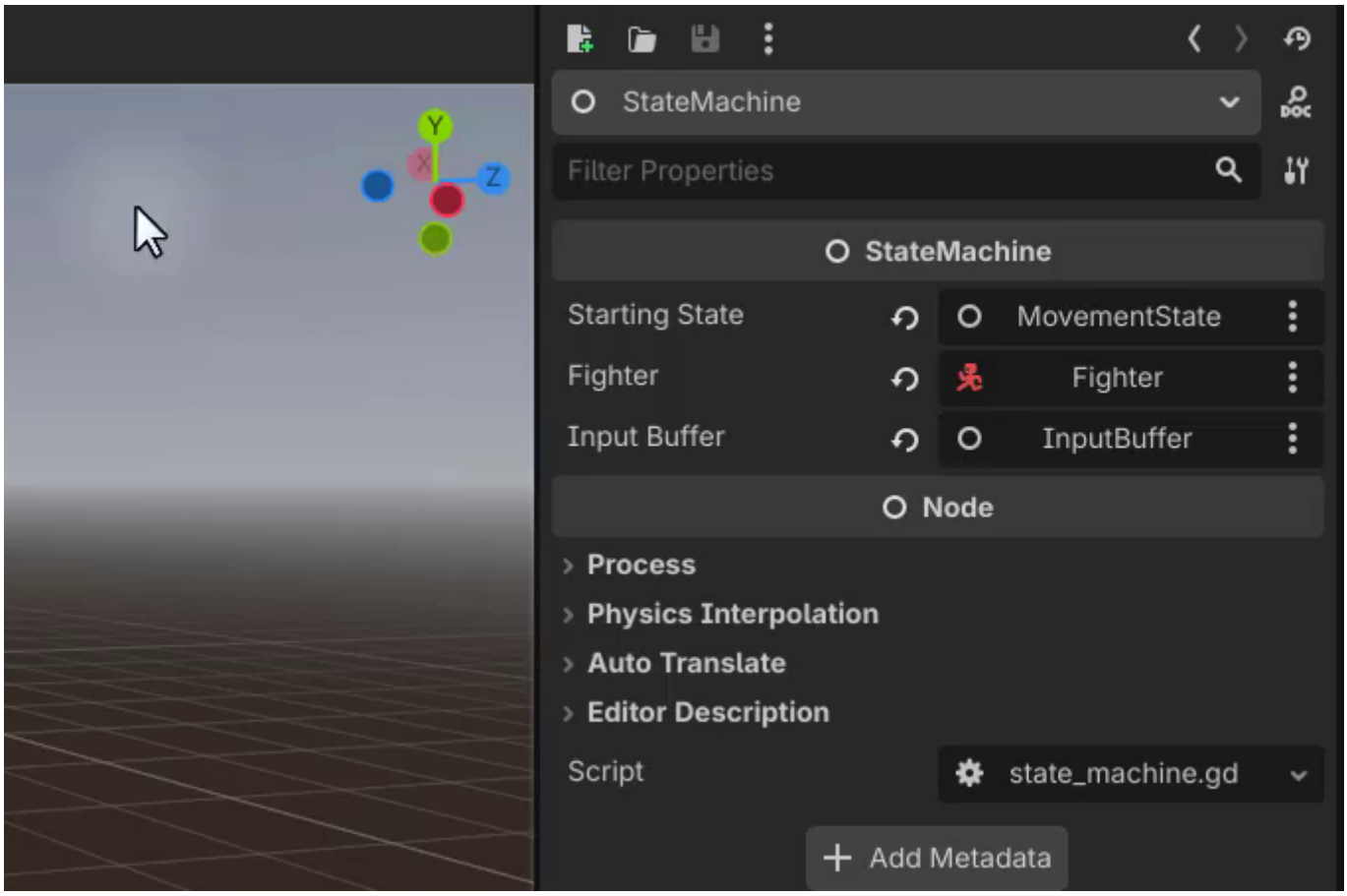

Recall that the base State class exposes fighter and input_buffer as convenience properties that forward to the owning state machine, which is why they can be used directly here without any additional setup.

Hooking the movement state into the fighter

With the script ready, open the fighter.tscn scene and clean up the old placeholder states from the state machine. The old *Standing* and attack state nodes can be removed, since proper versions will be rebuilt on top of the new movement state later in the course.



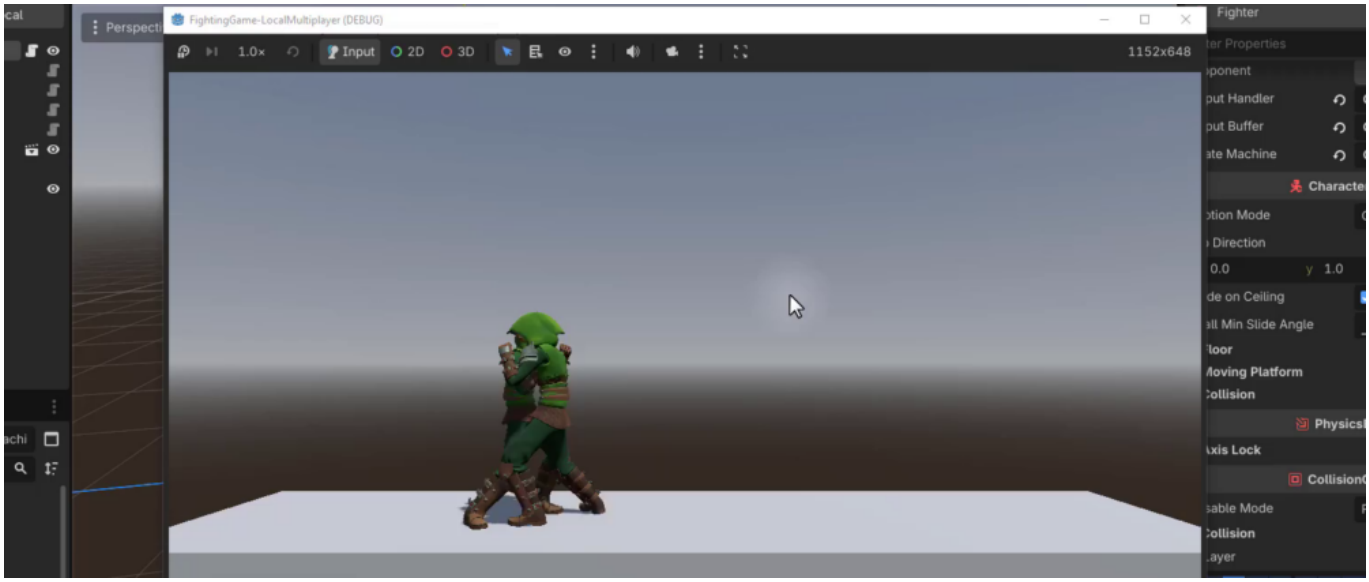
Next, add a new child node under *StateMachine* using the movement_state.gd script, and drag it into the *Starting State* slot on the StateMachine's Inspector so the fighter begins in MovementState as soon as the game runs.



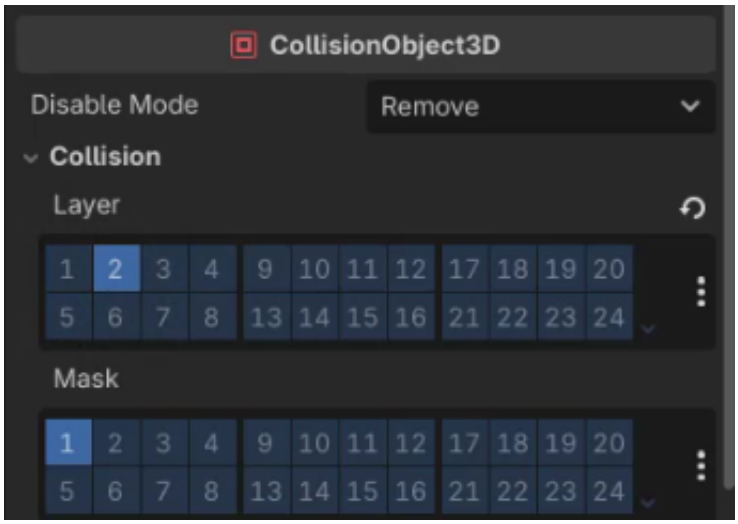
Pressing play now allows Player 1 to move their fighter left and right with the A and D keys, while Player 2 uses the left and right arrow keys.

Fixing collision so fighters can pass through each other

There is one remaining issue: the two fighters currently block each other physically, so they bump and push instead of passing cleanly through. For this project, the fighters should be able to move past one another so that the facing-direction logic can kick in.



The fix is to move both fighters onto their own collision layer. Select a fighter, find its CollisionObject3D collision settings in the Inspector, enable layer 2 (the character layer) and disable layer 1. Apply the same change to the other fighter.



Pressing play once more confirms the new behavior: the fighters now walk straight through each other, and as they swap sides their forward directions automatically flip so that they stay facing one another. Since both fighters currently use the identical model and color, this can look visually similar to them bouncing back and forth, but the rotation logic really is running and will become much more obvious later once they have distinct outfits and attacks.

In this lesson we finally start giving our fighter real behaviour by implementing its two most fundamental states: **Standing** and **Jumping**. The Standing state is the central “hub” of the fighter — from here, we can branch off into attacking, dashing, jumping, and so on. It represents the character not doing any particular action, just standing on the ground with the ability to move left and right. The Jumping state handles the character being in the air and falling back down under gravity.

Both states are going to share a lot of the same horizontal movement logic, which is exactly why we created the MovementState base class earlier — we only have to write that shared code once, and both new states can inherit from it.

Creating the standing state

Inside the **State Machine** scripts folder, create a new script called standing_state.gd. It extends our MovementState base class and gets a class name of StandingState.

For now, all we need is an update function that calls into the base MovementState.update via super.update(delta). We are not going to add the logic that transitions out of Standing just yet — we’ll come back to that once the Jumping state exists.

```
class_name StandingState
extends MovementState

func update (delta : float):
    super.update(delta)
```

Over in the scene tree, on the state machine node, we can delete the generic base movement state that was previously attached and replace it with our new StandingState. Rename the node simply to **Standing** — no “State” suffix — so it matches the name we will use when changing states later.

Creating the jumping state

Back in the **State Machine** folder, create another new script called jumping_state.gd. Like Standing, this one extends MovementState and is given a class name of JumpingState.

Inside the jump state, we need a couple of tunable parameters exposed in the inspector. Declare two exported variables: jump_force, the upward force applied when the jump begins, and gravity, the downward force that pulls the fighter back to the ground over time. Both are typed as float:

```
class_name JumpingState
extends MovementState

@export var jump_force : float = 5.0
@export var gravity : float = 10.0
```

Applying gravity in update

Override the update function. Each frame, we want to drag the fighter’s vertical velocity downwards a bit, which is exactly what applying gravity means. We subtract gravity * delta from the fighter’s vertical velocity — multiplying by delta ensures the effect is frame-rate independent.

We also still want the player to be able to drift left and right while airborne, so we call super.update(delta) to run the base MovementState horizontal movement code as well:

```
func update (delta : float):
    fighter.move_velocity.y -= gravity * delta
    super.update(delta)
```

Kicking off the jump in enter()

Update handles the ongoing gravity pull, but something needs to actually launch the fighter off the ground the moment the state begins. The base State class gives us an enter() method that is called exactly once, whenever the state first becomes active — the perfect place to set the initial jump velocity.

Override enter() and set the fighter's vertical velocity to jump_force:

```
func enter ():
    fighter.move_velocity.y = jump_force
```

Resetting velocity in exit()

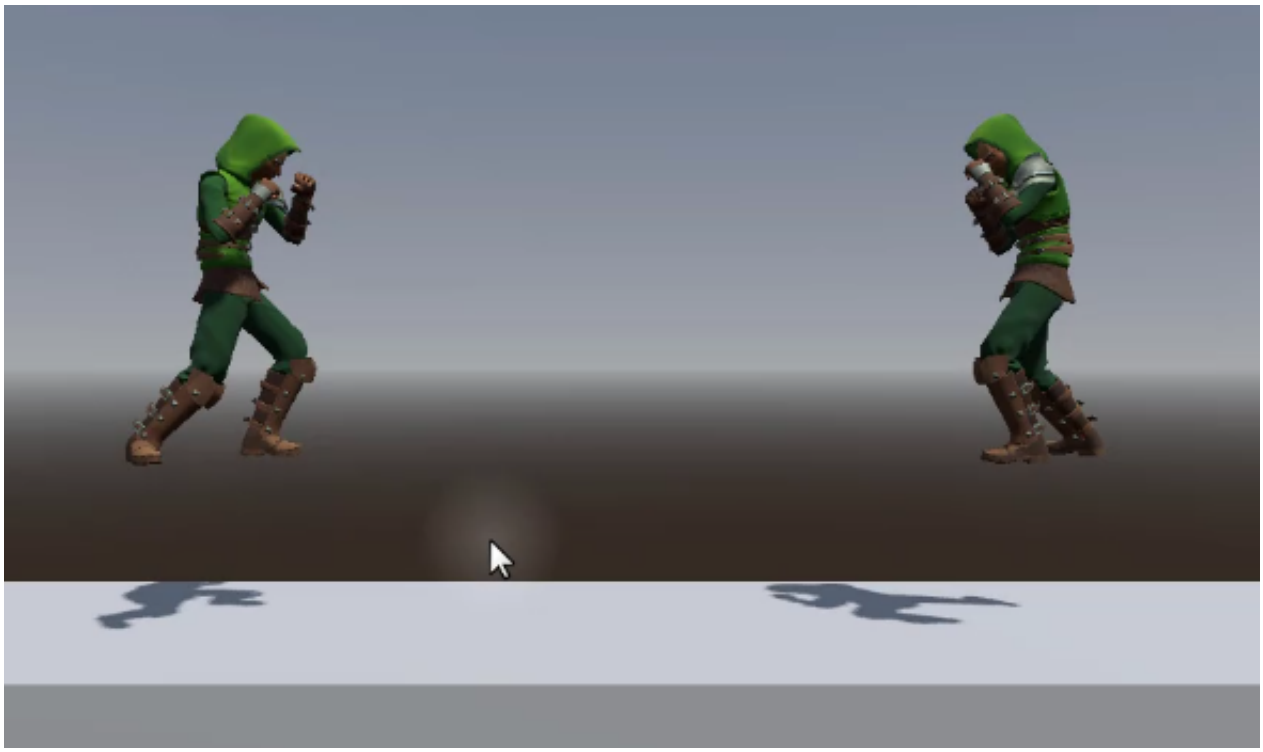
Just as enter() runs on entry, the base State class also exposes an exit() method that fires when the state is swapped out. When we leave the Jumping state, it means we've landed back on the ground, so we should cancel out any remaining vertical velocity so the fighter doesn't carry stray upward or downward momentum into the next state.

```
func exit ():
    fighter.move_velocity.y = 0
```

Testing the jump in isolation

Back in the scene, add a new child node under the state machine for the Jumping state, attach jumping_state.gd to it, and rename it to **Jumping**. To quickly test that the jump itself works, temporarily set the state machine's starting state to **Jumping** instead of Standing and press play.

Both fighters should immediately launch into the air, arc back down under gravity, and land on the platform. The Output console will log New State: Jumping for each fighter on startup.



However, there's a problem: after landing, the fighters never leave the Jumping state. The console keeps reporting New State: Jumping but never transitions back to Standing. We need to add that transition ourselves.

Transitioning from Jumping back to Standing

Back in jumping_state.gd, at the end of the update function, we check whether the fighter is currently touching the floor. Godot's CharacterBody3D provides a built-in is_on_floor() method for exactly this. If the fighter is on the floor, we tell the state machine to change back to the "Standing" state — and remember, the string must match the state node's name exactly:

```
func update (delta : float):
    fighter.move_velocity.y -= gravity * delta
    super.update(delta)

    if fighter.is_on_floor():
        state_machine.change_state("Standing")
```

Press play again. Both fighters will jump, fall, land, and this time the Output log will show them returning to the Standing state as soon as they touch the ground.

```
New State: Jumping
New State: Jumping
New State: Standing
New State: Standing
--- Debugging process stopped ---
```

Transitioning from Standing to Jumping

We now have one direction of the transition working, but we still need a way to trigger a jump in the first place — going from Standing to Jumping when the player presses the jump button.

Head back into standing_state.gd. Inside the update function, after calling super.update(delta) (which is what gives the character the ability to actually move left and right via the base MovementState), we check the input buffer for the jump input. If it's pressed, we ask the state machine to change to the **Jumping** state — again using the exact same name as the node:

```
func update (delta : float):
    super.update(delta)

    if input_buffer.is_pressed("jump"):
        state_machine.change_state("Jumping")
        return
```

You can start to see the rhythm of the state machine here. Each state is entered, its update function runs every physics frame, and it exits based on some condition — Jumping exits when the fighter lands on the floor; Standing exits when the player presses jump.

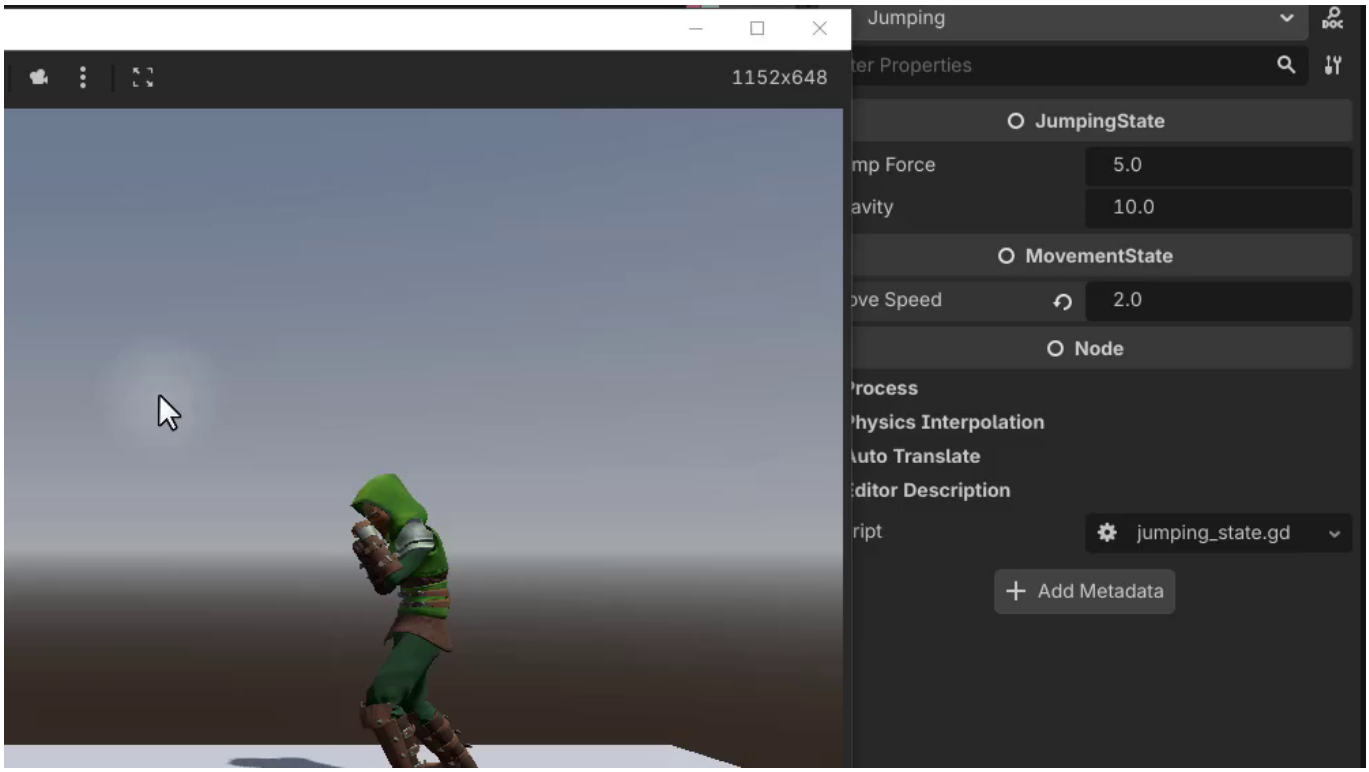
Before testing, remember to flip the state machine's starting state back to **Standing** so the fighters don't launch into the air the moment the game begins.

Now pressing play gives us a real mini game loop: both players can walk left and right independently, and pressing their jump input launches them into the air. The Output log shows New State: Jumping and New State: Standing cycling as they land and jump again.

Giving each state its own move speed

Because both Standing and Jumping inherit from MovementState, each of them gets its *own* independent move_speed value in the inspector. That means we can give the character a different horizontal speed while airborne than while grounded.

Select the **Jumping** node in the state machine and bump its Move Speed up in the inspector — something a bit higher than the Standing state's value. Now when a player jumps, they'll slide sideways through the air a little faster than when they walk on the ground.

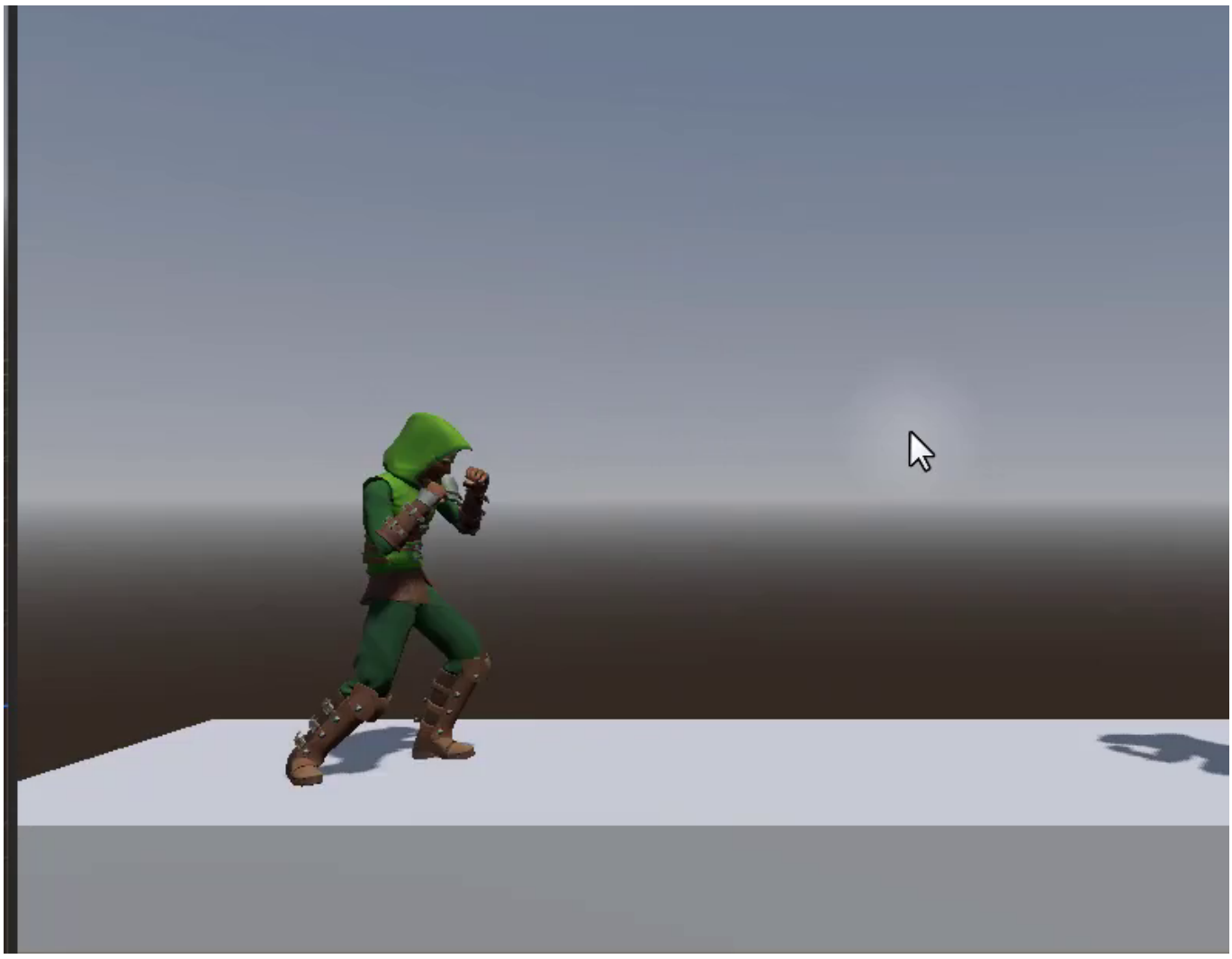


This is actually a meaningful game design choice, not just a cosmetic tweak. Later in the course, we'll make it so that the player cannot attack while in the air. Pairing that restriction with faster air movement gives jumping a real purpose: it becomes a fast evasive tool for dodging attacks at the cost of being unable to strike back. Meanwhile, slower grounded movement keeps the ground a more committed, offensive space. You can tweak these numbers freely to get a feel you like — and of course, everything also works independently for the second fighter.

With Standing and Jumping now in place, the rest of the fighter's behavior is largely a matter of following the same pattern for every additional state we want — dashing, light attacks, heavy attacks, hit reactions, and so on — each one a small script that extends from the appropriate base and transitions in and out of the existing states.

In the previous lessons, we built the fighter scene, the input handling pipeline and the core of the state machine. If we press play right now, the character can move left and right and jump, but everything looks frozen because the fighter never leaves its default pose. In this lesson, we will take the first step toward bringing the character to life by setting up an **AnimationTree**, building a **1D blend space** for the standing state, and creating a small script that will act as the bridge between the state machine and the animation system.

Animations are not just a visual polish detail for this project. Later on, the attack states will depend directly on the animations, since the hit detection windows are tied to specific animation frames. That makes getting the animation system in place a prerequisite for everything that follows.



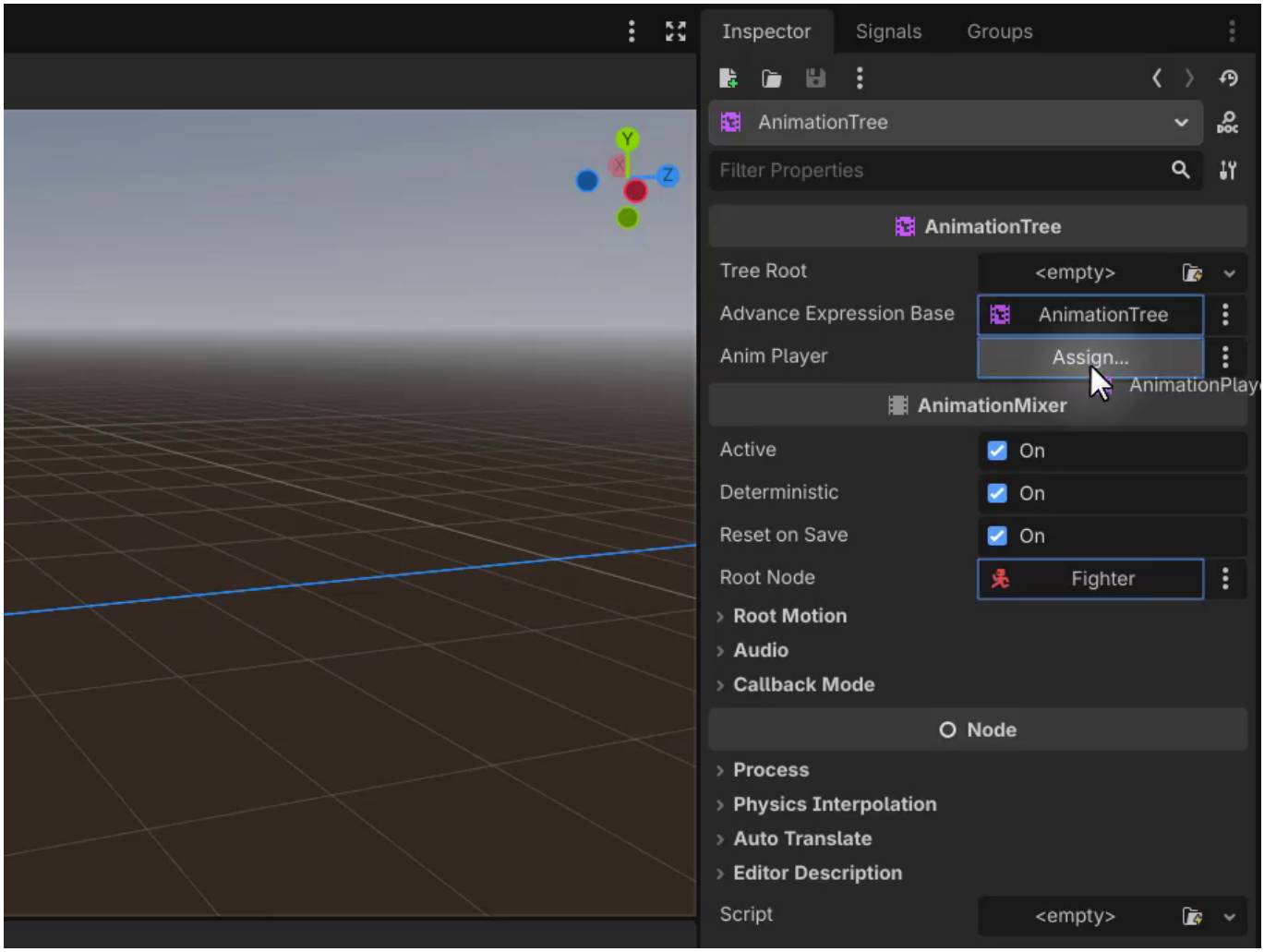
The AnimationPlayer and the AnimationTree

Inside the Fighter scene there is already an AnimationPlayer node that was added in a previous lesson. It contains all of the animations the fighter will ever need to play: idle, walk, jump start, jump idle, jump land, light and heavy attacks, hit reactions, defeat, and so on. What we are missing is a way of deciding *when* each of those animations should play and how to transition between them.

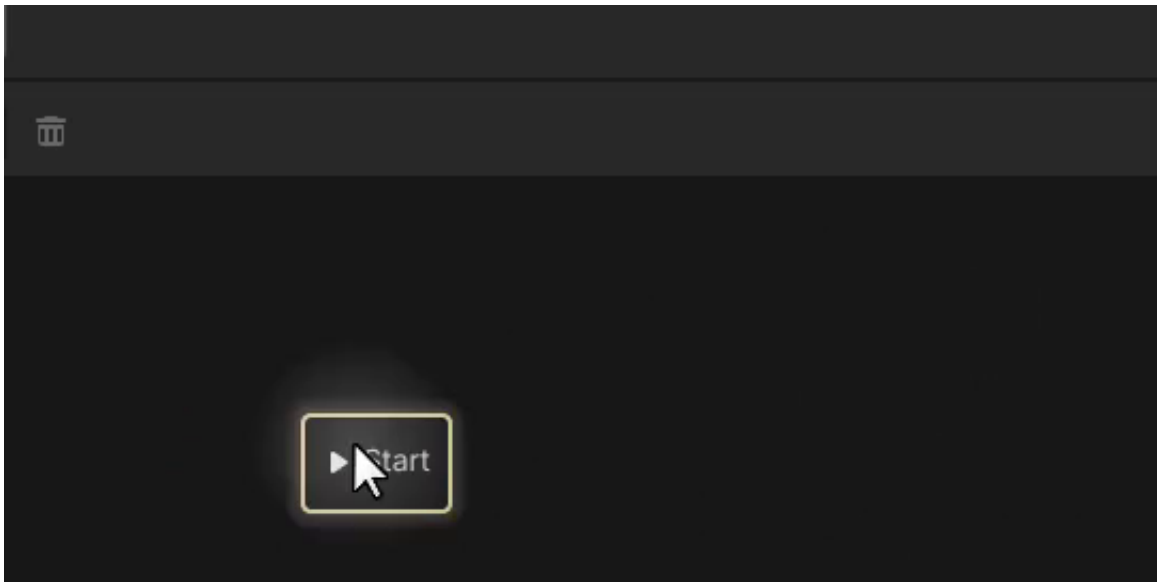
For example, jumping is not a single animation. It is a sequence: JumpStart plays as the fighter leaves the ground, JumpIdle loops while they are in the air, and JumpLand plays on impact. Coordinating that by calling the AnimationPlayer directly from code would quickly become tangled. Instead, Godot provides an **AnimationTree** node that lets us describe animation behavior as a state

machine of its own, very similar in spirit to the gameplay state machine we have been building.

Add a new child of the Fighter called AnimationTree. In the Inspector, drag the existing AnimationPlayer into the *Anim Player* property so the tree knows which animations it can play.



Next, look for the *Tree Root* property. This is where we tell the AnimationTree what kind of tree we want. Godot offers a few options here; the one we want is **AnimationNodeStateMachine**. Once you assign it, an AnimationTree graph editor panel opens at the bottom of the screen, which can be panned by clicking and dragging with the middle mouse button.

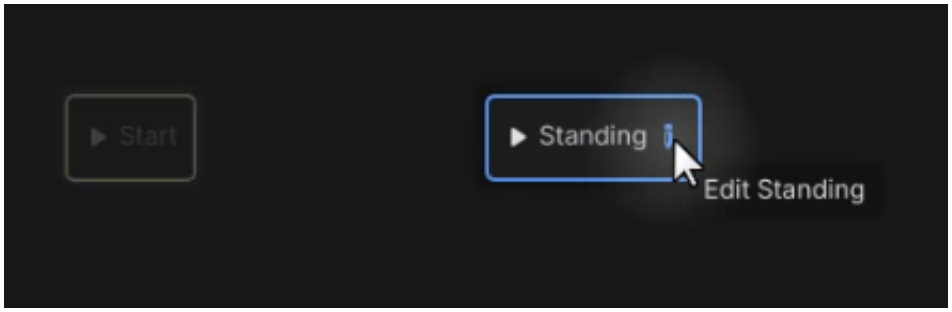


The graph starts with two nodes, Start and End. Conceptually, Start is the entry point: when the AnimationTree is initialized at runtime, whichever node is connected to Start becomes the default animation state. This mirrors the gameplay state machine, which also has a starting state and transitions between states, but here each node represents an animation (or a group of blended animations) rather than a gameplay behavior.

Creating the Standing state as a BlendSpace1D

The first state we will add is **Standing**. This state needs to cover two visuals at once: the fighter standing still (idle) and the fighter walking. Instead of creating two separate states and transitioning between them, we can use a single node that smoothly blends between the two animations based on a value we feed in from code.

Right-click in the graph and, instead of adding an animation directly, choose *Add BlendSpace1D*. Rename the new node to Standing, then click the little pencil icon on the node to open its editor.

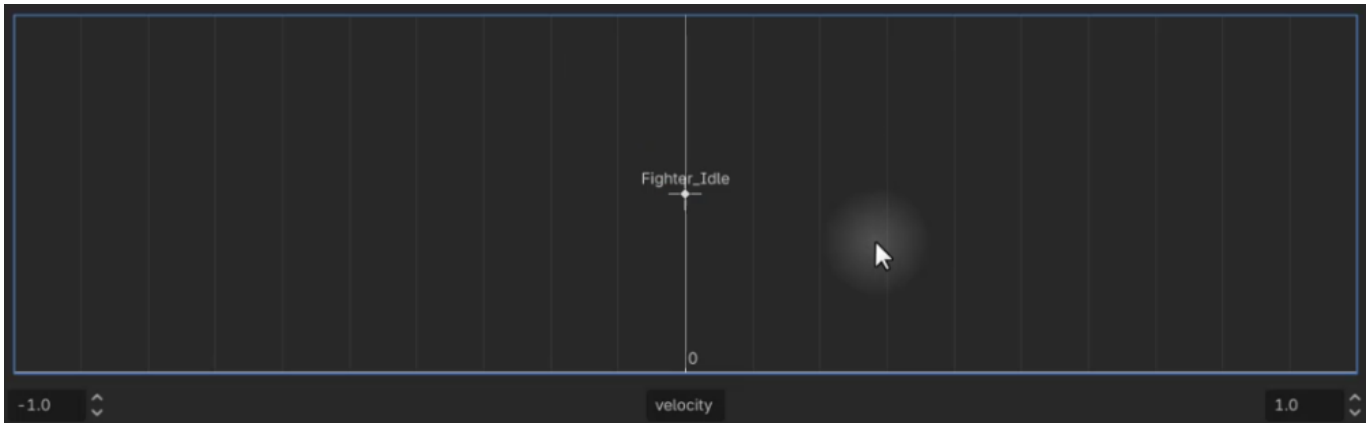


A 1D blend space is essentially a one-dimensional graph. It has an x-axis that by default runs from -1 to 1, and we can place animations anywhere along that axis. At runtime, we will assign a value to the axis from a script, and the blend space will automatically interpolate between the two closest animations.

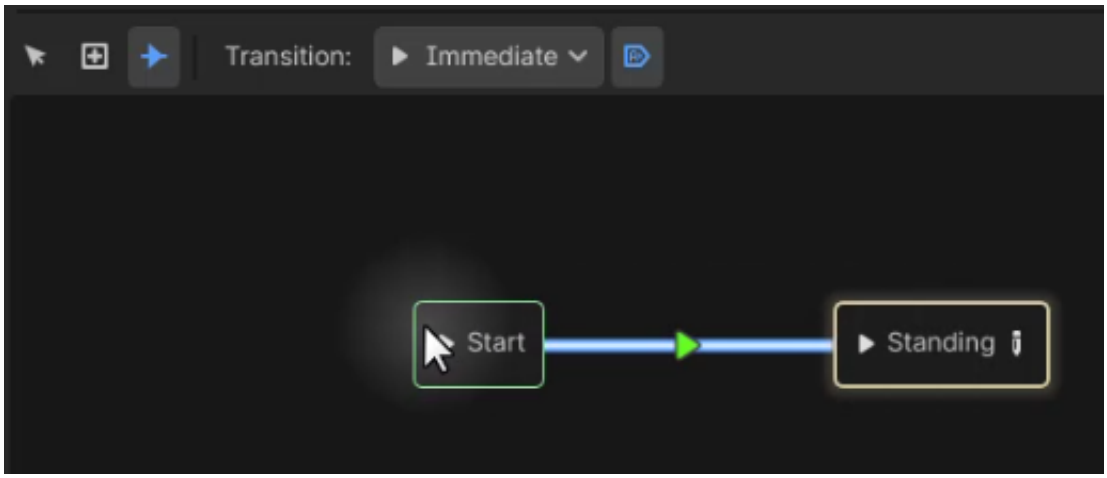
Rename the axis label from its default to velocity. This is the value we will feed in from code: it will represent the direction the player is moving. Standing still corresponds to 0, walking forward corresponds to 1, and walking backward corresponds to -1.

Placing the idle and walk animations

With the blend space open, right-click in the middle of the graph, choose *Add Animation*, and select the idle animation. This places the fighter's idle pose at the centre of the axis, where velocity is 0.

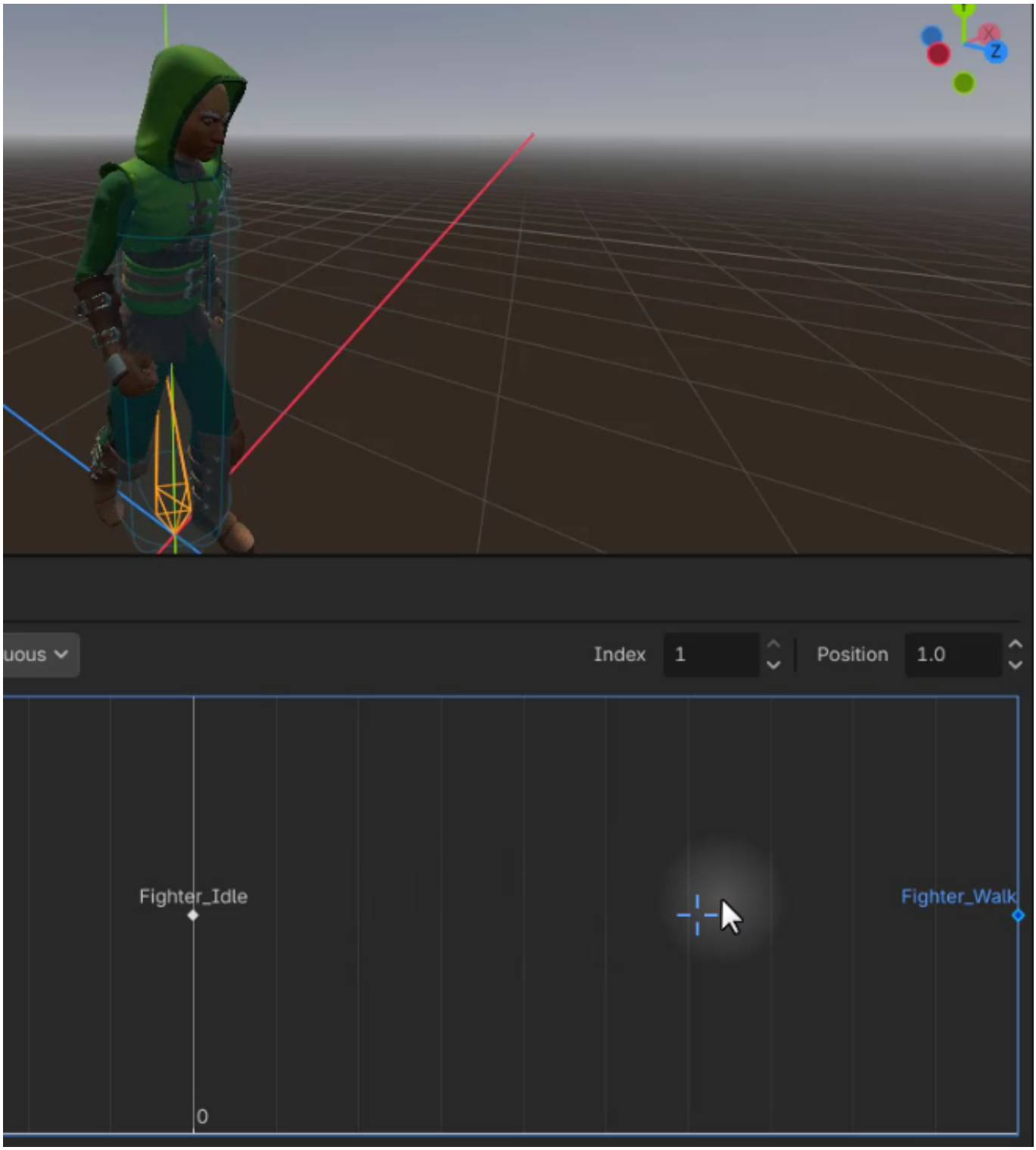


To preview the result inside the editor, we need the Standing node to actually be the active state in the parent graph. Click the *root* button in the top-left corner of the graph to go back to the outer state machine. Then select the *Connect nodes* tool at the top of the graph editor and click-drag from the Start node into the Standing node. Once the connection is in place, the AnimationTree treats Standing as the default state that becomes active when the tree is initialized.



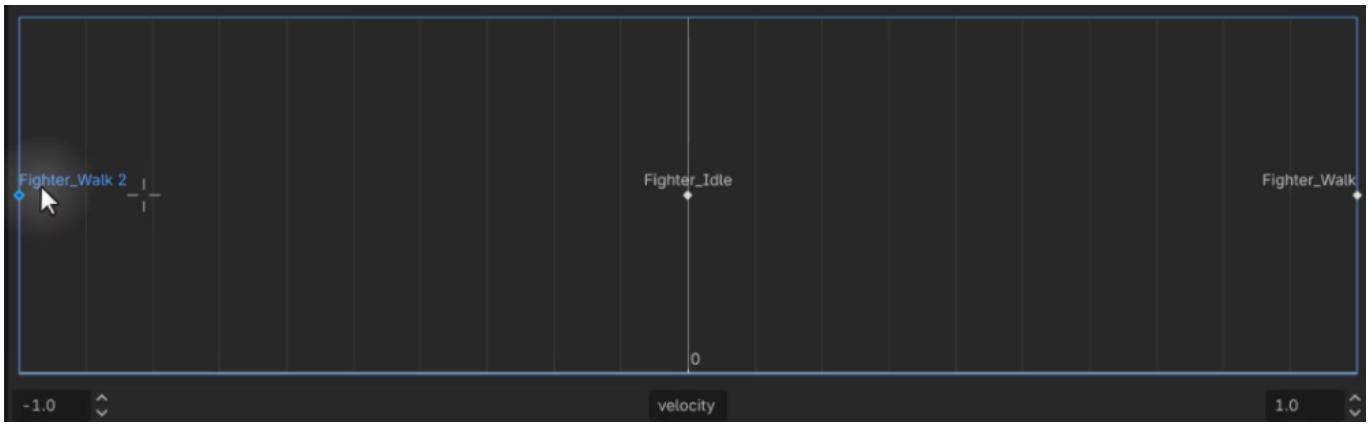
Switch back to the pointer tool and click *Edit* on the Standing node to re-enter the blend space. You should now see the fighter playing their idle animation in the viewport. Right-click on the right-hand side of the axis, add the walk animation, and drag it to the position 1.

There is a small button in the blend space toolbar labelled *Set the blending position within the space*. Enable it and click-drag along the axis. As the indicator moves from 0 toward 1, the fighter gradually transitions from idle into a full walk, and if you stop halfway the two animations are visually blended together. This is exactly what the blend space does for us automatically: we only ever have to set one number, and Godot handles the smooth interpolation between the neighbouring animations.

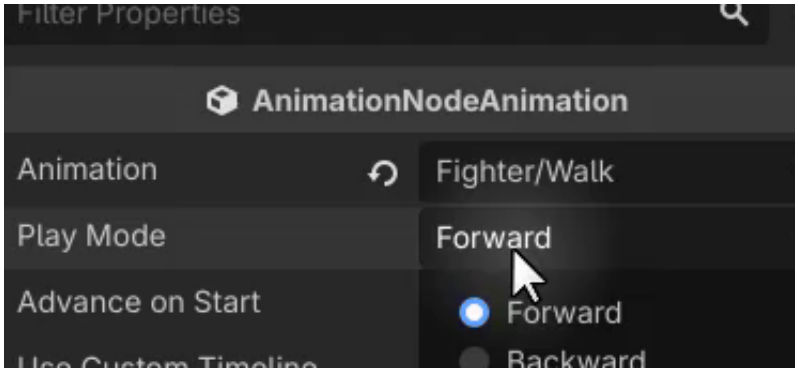


Reusing the walk animation for walking backward

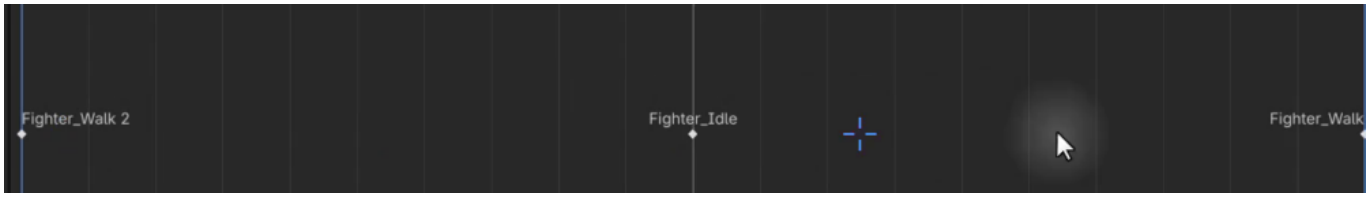
The project does not ship with a dedicated backward walk animation, but we can fake one by playing the existing walk animation in reverse. Back in the pointer tool, right-click on the left side of the axis, add the walk animation again, and drag it to the position -1.



With this new node selected, look in the Inspector for the *Play Mode* property on the AnimationNodeAnimation. Change it from Forward to Backward. That single change tells the blend space to play that particular instance of the walk animation in reverse.



Now when you drag the velocity indicator along the axis you should see the fighter smoothly cycle through walking backward at -1, idle at 0, and walking forward at 1. With three blend points in place, the Standing state can cover the full range of ground movement without ever having to leave this single node.



The FighterAnimation script

The blend space works beautifully in the editor, but in the actual game we need some code that reads the player's movement and feeds the corresponding value into the tree. Rather than having the state machine talk directly to the AnimationTree, we will introduce a small dedicated script that acts as a clean bridge between the two. This keeps the state machine code focused on gameplay logic and gives us a single place to configure how the game drives animations.

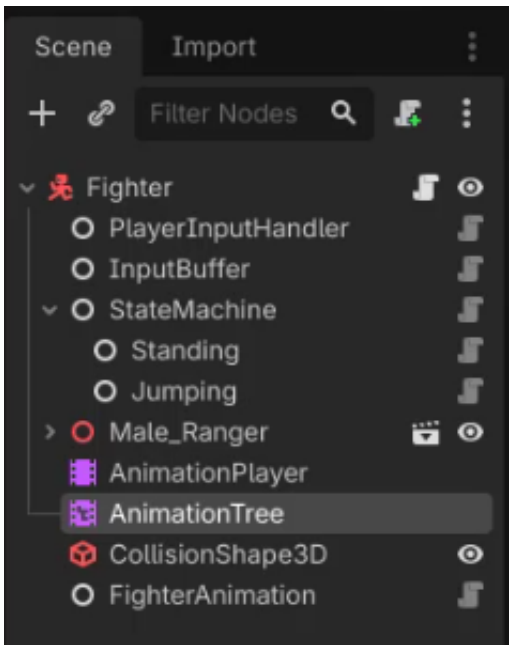
In the FileSystem panel, go to Scripts/Fighter and create a new script called fighter_animation.gd. Give it the class name FighterAnimation and have it extend Node. The script needs a reference to the AnimationTree it will be controlling, so expose one as an exported variable:

```
class_name FighterAnimation
extends Node
```



```
# Reference to the AnimationTree node for controlling animations
@export var anim_tree : AnimationTree
```

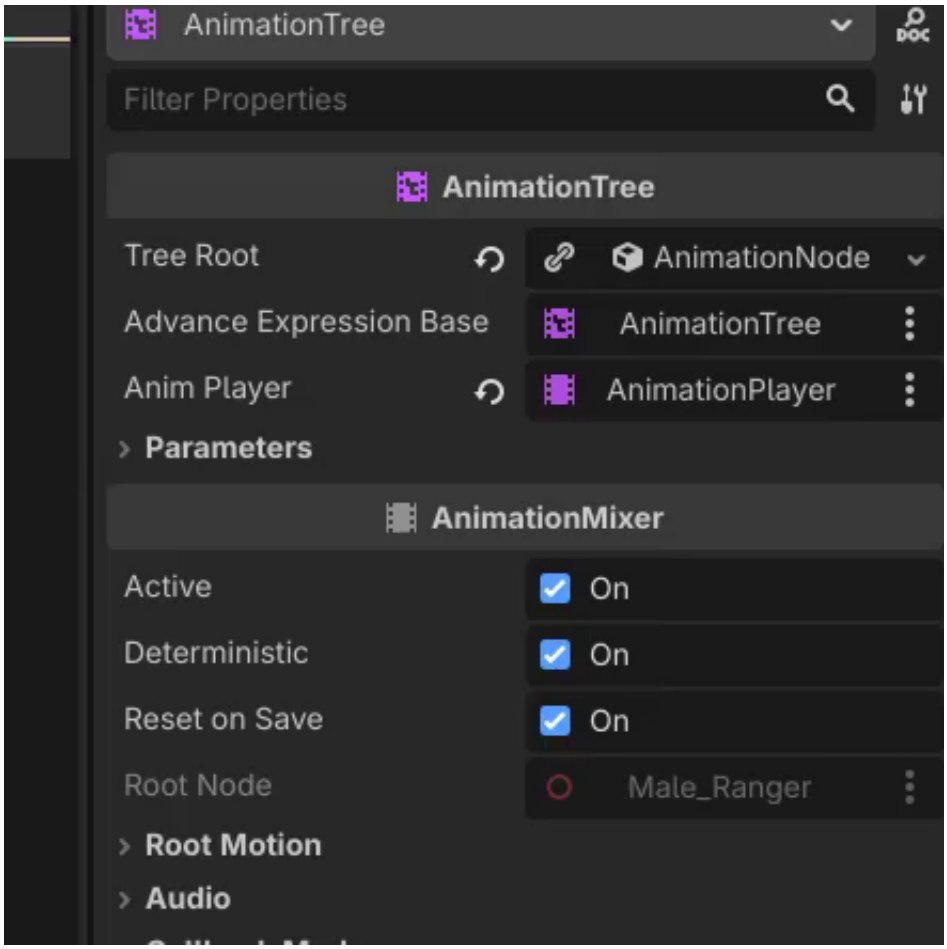
Back in the Fighter scene, right-click the Fighter node and add a new child node. Search for FighterAnimation in the node list and add it. With the new node selected, drag the existing AnimationTree into its *Anim Tree* slot so the reference is wired up.



Setting the blend position from code

The goal of the first function on this script is to change the value that drives the blend space, that is, the velocity parameter on the Standing node. Inside an AnimationTree, every such value is addressable via a string path. To find the exact path we need, select the AnimationTree in the scene, expand *Parameters* in the Inspector, navigate to *Standing* → *blend position*, and hover over the property. Godot shows the full path in the tooltip:

```
parameters/Standing/blend_position
```

That string is what we will feed into the function. Back in fighter_animation.gd, add a function called set_blend_position that takes the path as a String and the new value as a float. The implementation is a single line: index into the AnimationTree with the path and assign the new value.

```
func set_blend_position (path : String, value : float):
 anim_tree[path] = value
```

The square-bracket syntax on the AnimationTree works just like a dictionary lookup: it writes the value into whatever parameter the path points to. Since the path is a generic argument, the same function can be reused later for any other blend space we add to the tree, not just Standing.

Exposing the animation on the state machine

The states themselves are the ones that will eventually call set_blend_position, so they need an easy way to reach the FighterAnimation instance. The cleanest place to hold that reference is the state machine, which is already shared by every state.

Open state_machine.gd and add a new exported variable so the FighterAnimation node can be assigned in the Inspector:

```
@export var animation : FighterAnimation
```


Then open the base state.gd script and add a read-only getter that forwards to the state machine. This mirrors the pattern already used for the fighter and input buffer references, and means individual states can simply write animation.set_blend_position(...) without having to reach into the state machine each time:

```
var animation : FighterAnimation:
 get: return state_machine.animation
```

These wrapper variables are not strictly necessary, but they keep the code inside each state considerably shorter and easier to read.

Wrapping up

At this point the AnimationTree is in place, the Standing blend space blends smoothly between idle, forward walk, and reversed walk, and there is a FighterAnimation script ready to drive the blend value from code. The states can already see that script through the new animation property on the base state class. In the next lesson, we will actually start calling set_blend_position from inside the Standing state so the blend value reacts to the player’s movement input in real time.

In this lesson, we will connect the AnimationTree's blend position to our movement state script so that the fighter plays the correct idle, walk forward, and walk backward animations based on input. We will also smooth out animation transitions using linear interpolation and set up the jump animation states in the AnimationTree.

Adding the Blend Position Parameter to MovementState

Open the movement_state.gd script. This is the base class for both the Standing and Jumping states. We need to add an exported string variable that will hold the path to the AnimationTree's blend position parameter. This path will be assigned in the Inspector for each state that uses it.

```
@export var blend_position_parameter : String
```

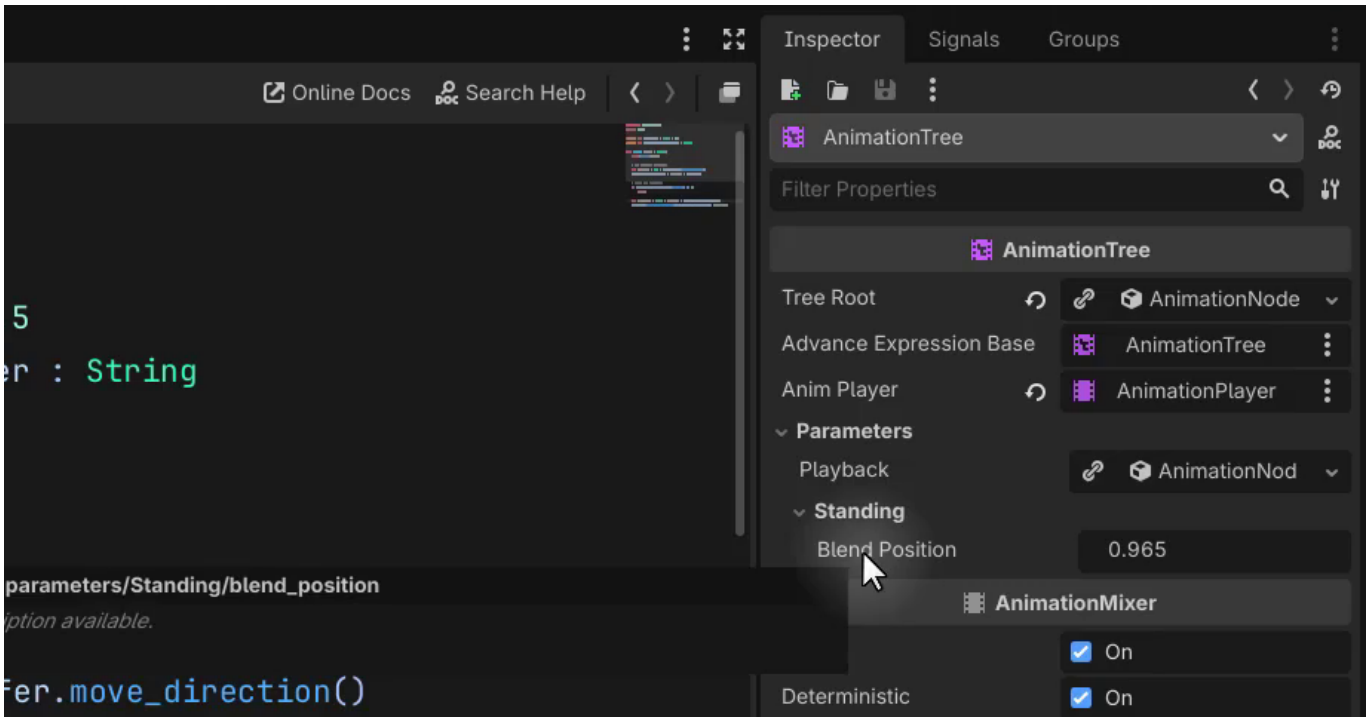
Next, inside the update() function, after the existing code that sets the fighter's horizontal velocity based on input direction, add the animation blending logic. First, check whether a blend position parameter has been assigned. If the string is empty, we skip the blending — this is important because states like Jumping also extend MovementState but do not need a blend position parameter.

```
func update (delta : float):
 super.update(delta)

 # Set movement direction
 var move_dir : int = input_buffer.move_direction()
 fighter.move_velocity.x = move_dir * move_speed

 # Blend the animation
 if blend_position_parameter.length() == 0:
  return

 var blend_pos : float = move_dir * fighter.forward_direction
 animation.set_blend_position(blend_position_parameter, blend_pos)
```

The blend position value is calculated by multiplying the move direction by the fighter's forward direction. This ensures the animation plays relative to the direction the character is facing — so moving toward the opponent always plays the walk-forward animation, regardless of which side of the screen the fighter is on.

Configuring the Standing State in the Inspector

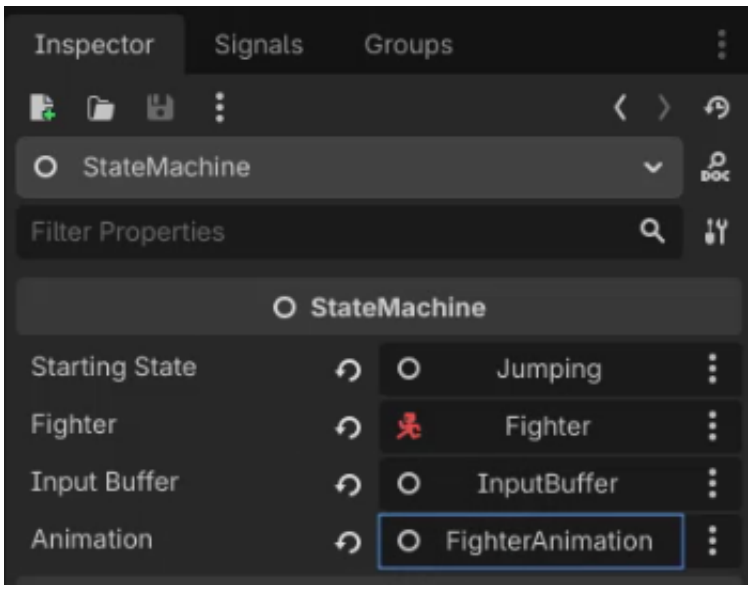
Save the script and select the Standing state node in the scene tree. In the Inspector, you will see the new **Blend Position Parameter** field. To find the correct path, select the AnimationTree node and expand the **Parameters** section in the Inspector. Under **Standing**, locate the **Blend Position** property. Right-click on it and select **Copy Property Path**.



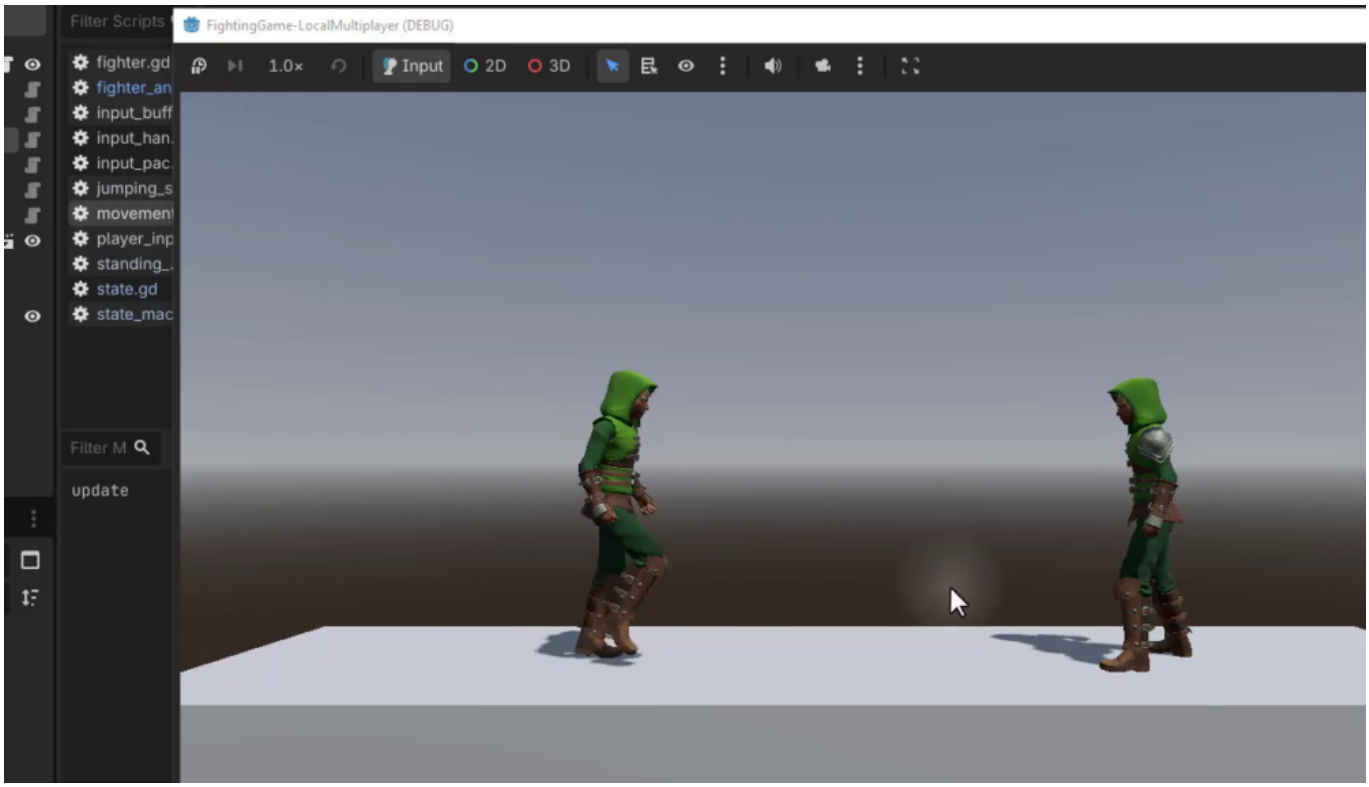
Now select the Standing state node again and paste the copied path into the **Blend Position Parameter** field. This links the script variable to the correct AnimationTree parameter so that it can be controlled at runtime.

Fixing the Setup and Testing Walk Animations

Before testing, make sure the StateMachine node has the FighterAnimation node assigned to its **Animation** property in the Inspector. Drag the FighterAnimation node into that slot if it is not already assigned.



Press play and test the movement. The fighter should now play the idle animation when standing still, the walk-forward animation when moving toward the opponent, and the walk-backward animation when moving away. Both fighters should display the correct animations.



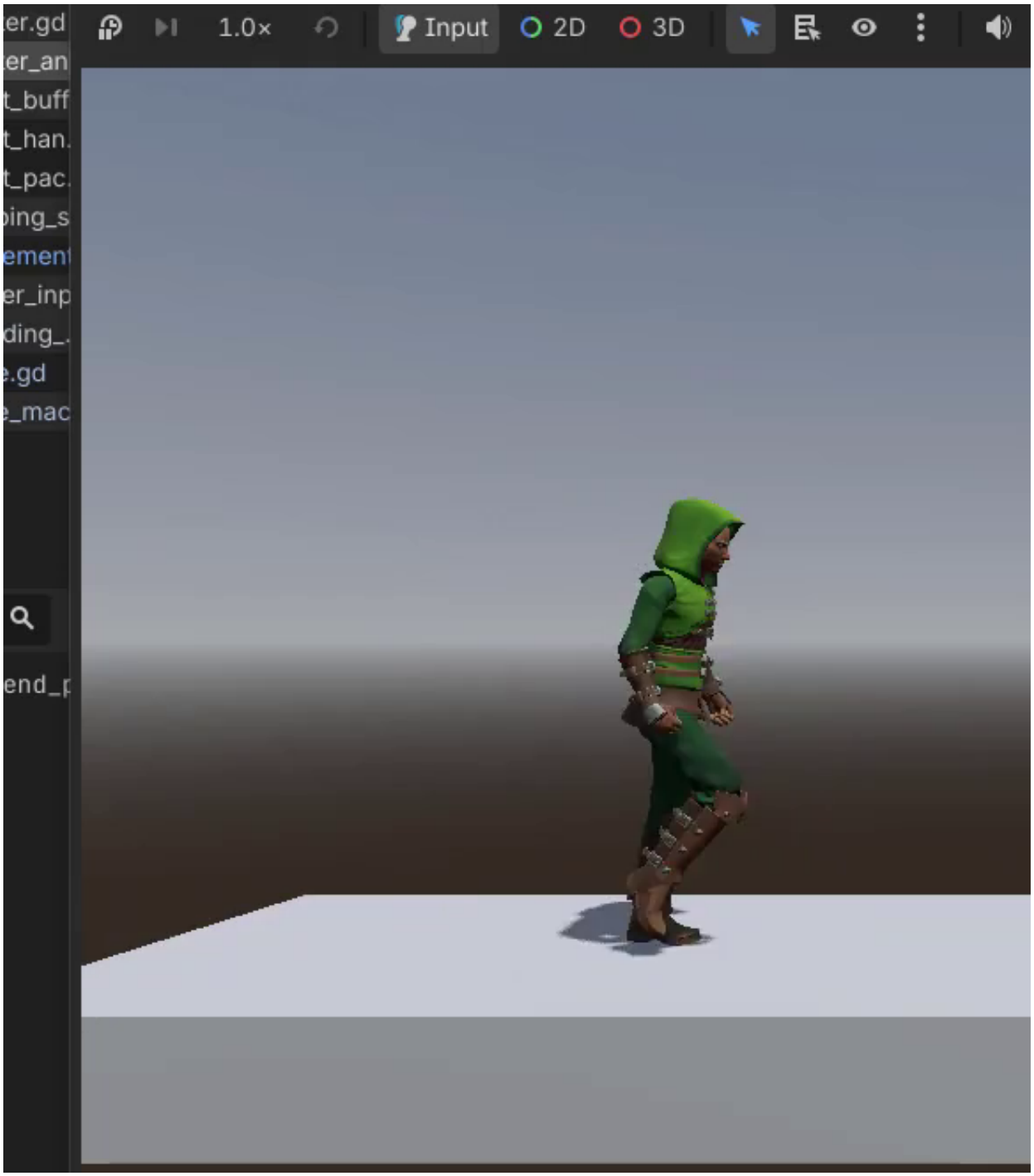
Smoothing Animation Transitions with Lerp

While the walk animations work, you may notice that rapidly switching between left and right movement causes the animation to snap abruptly between states. This happens because the blend position value is being set each frame directly.

To smooth this out, open fighter_animation.gd and modify the set_blend_position() function. Instead of assigning the value directly, use Godot's lerp() function to gradually interpolate from the current value toward the target value over multiple frames.

```
func set_blend_position (path : String, value : float):
 anim_tree[path] = lerp(anim_tree[path], value, 0.3)
```

The lerp() function takes three arguments: the current value (read from the AnimationTree at the given path), the target value, and a weight that controls how quickly the transition occurs. A weight of 0.3 provides a smooth transition that is fast enough for a fighting game while avoiding the jerky snapping of direct assignment.



While a fighting game benefits from responsive controls, having a small amount of blending prevents the animations from looking unnatural when the player rapidly changes direction.

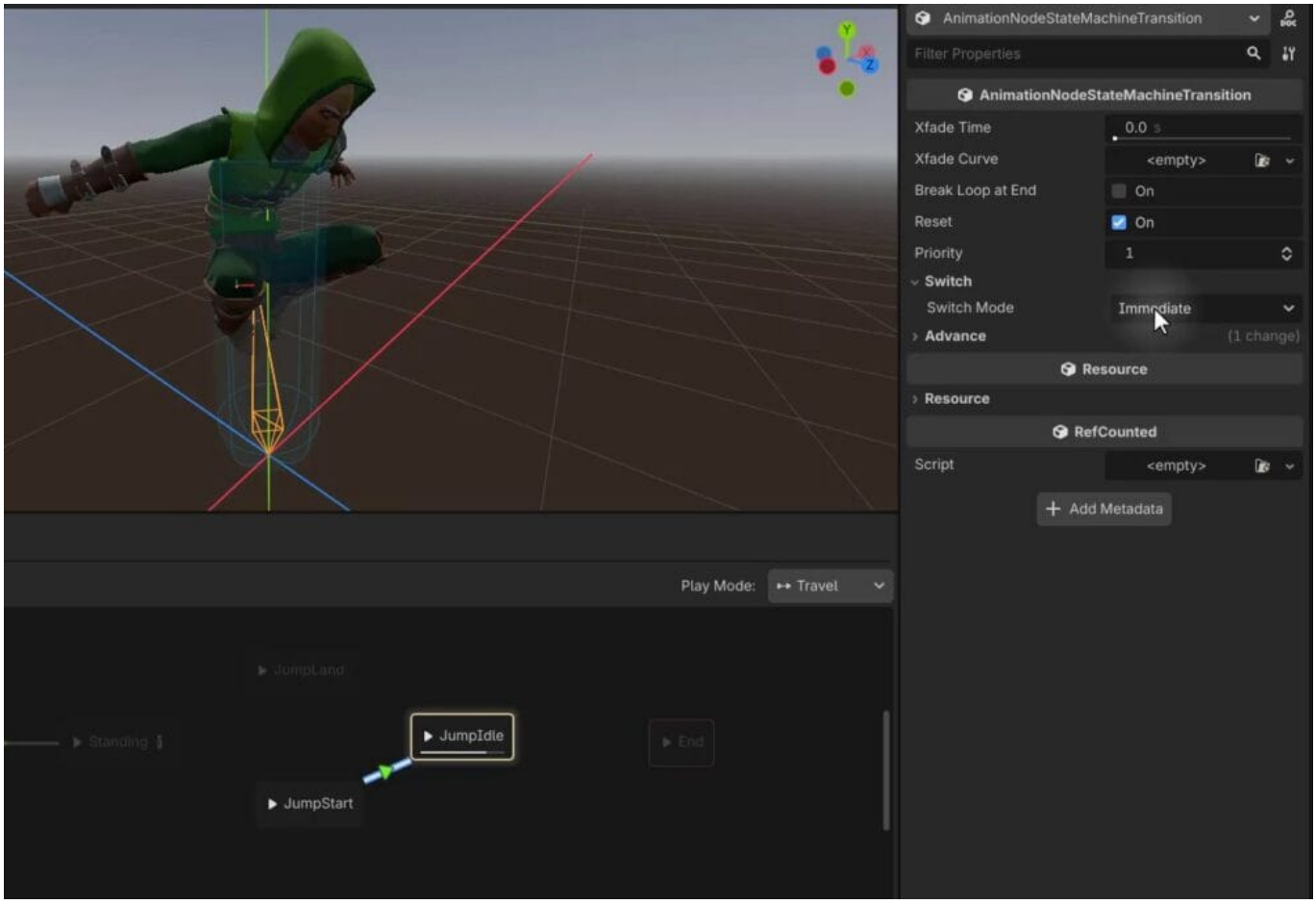
Setting Up Jump Animations in the AnimationTree

With the standing animations working, the next step is to add jump animations. Open the AnimationTree and navigate back to the root of the state machine. Currently, it only contains the Standing state.

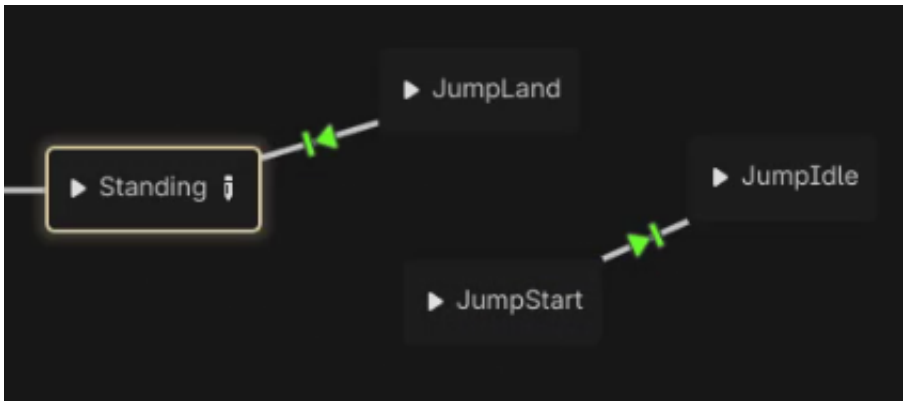
The jump animation is divided into three parts: **JumpStart** (the initial upward motion), **JumpIdle** (the airborne loop), and **JumpLand** (the landing recovery). Each of these needs to be added as a separate animation node in the state machine.

Right-click in the AnimationTree editor and select **Add Animation**. Choose the **JumpStart** animation and rename the node to JumpStart. Repeat this process to add **JumpIdle** and **JumpLand** as separate animation nodes.

Now connect them together. Use the **Connect Nodes** option to draw a transition from JumpStart to JumpIdle. By default, this transition uses **Immediate** switch mode, which means it will transition instantly — before JumpStart has finished playing. Select the transition line and in the Inspector, change the **Switch Mode** from **Immediate** to **At End**. This ensures JumpStart plays completely before transitioning to JumpIdle.



Next, connect JumpLand to Standing with another transition, and set that one to **At End** as well. This way, when the fighter lands, the landing animation plays in full before returning to the standing state.



Creating the set_animation Function

To trigger these animations from code, we need a new function in fighter_animation.gd that can instantly switch the AnimationTree to a specific animation state. Add the following function:

```
func set_animation (animation_name : String):
 anim_tree["parameters/playback"].travel(animation_name)
```

This accesses the AnimationTree's playback parameter and calls the travel() function, which instantly switches the state machine to the specified animation state. The "parameters/playback" path can be found by selecting the AnimationTree node, hovering over the **Playback** property under Parameters, and using **Copy Property Path**.

Triggering Jump Animations from the JumpingState

Open jumping_state.gd and update the enter() and exit() functions to trigger the appropriate jump animations. When the fighter enters the jumping state, play the JumpStart animation. When the fighter exits the jumping state (upon landing), play the JumpLand animation.

```
func enter ():
 fighter.move_velocity.y = jump_force
 animation.set_animation("JumpStart")

func exit ():
 fighter.move_velocity.y = 0
 animation.set_animation("JumpLand")
```

The enter() function applies the upward jump force and triggers JumpStart. The AnimationTree will automatically transition from JumpStart to JumpIdle once the start animation finishes. When the fighter lands and the state machine transitions back to Standing, the exit() function fires and plays JumpLand, which then transitions back to the Standing animation on its own.

Testing the Jump Animation

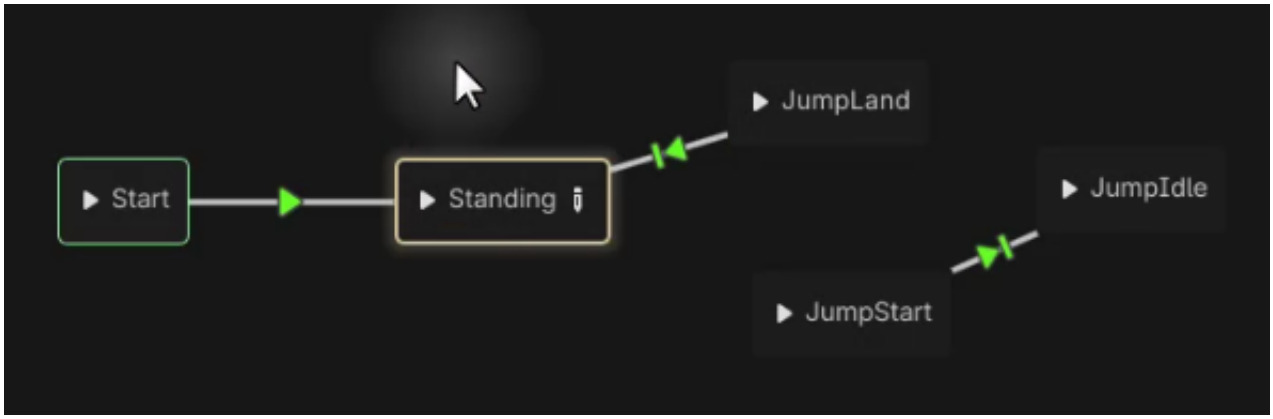
Save all scripts and press play. The fighter should now play the full jump animation sequence: JumpStart when leaving the ground, JumpIdle while airborne, and JumpLand when touching down.

The landing animation may feel slightly slow to recover from. Adjusting the animation playback speed to make it feel more responsive will be covered in the next lesson.

In this lesson, we continue building out the AnimationTree for the fighter character in our Godot fighting game. Although the corresponding game states have not yet been created, we will preconfigure all the animation states and transitions so they are ready to use when the state logic is implemented in upcoming lessons.

Reviewing the Existing Animation Tree

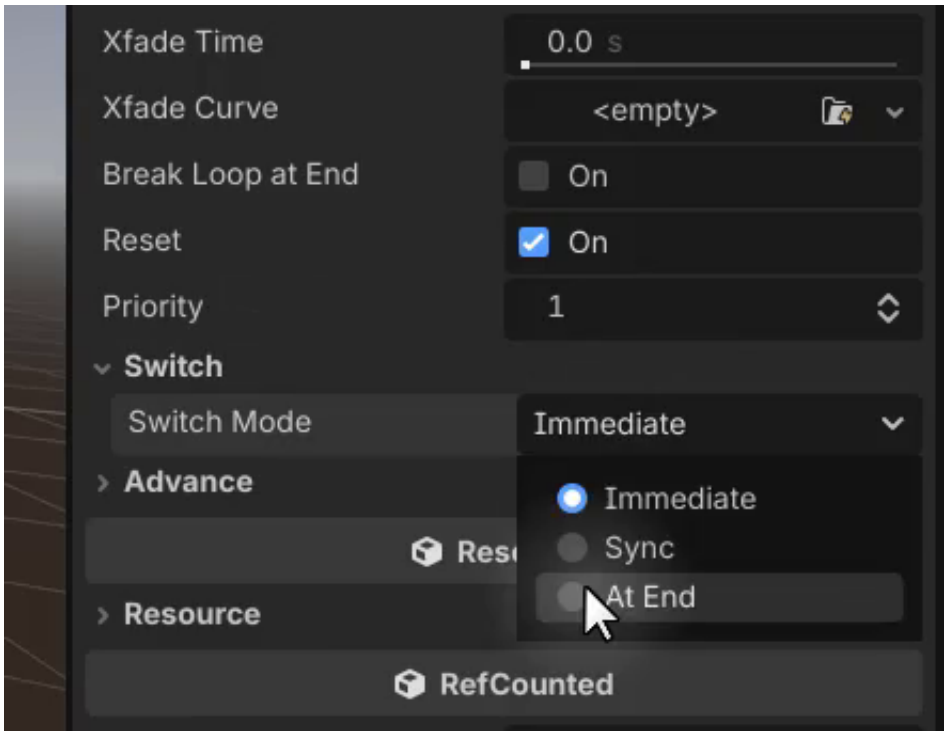
The AnimationTree already contains a few states from previous lessons. The Standing state includes idle and walk animations controlled by a blend position. The jump routine consists of three connected states: JumpStart transitions into JumpIdle, and when the character touches the ground, it transitions to JumpLand. With these in place, we are ready to add the remaining animation states that the fighter will need.



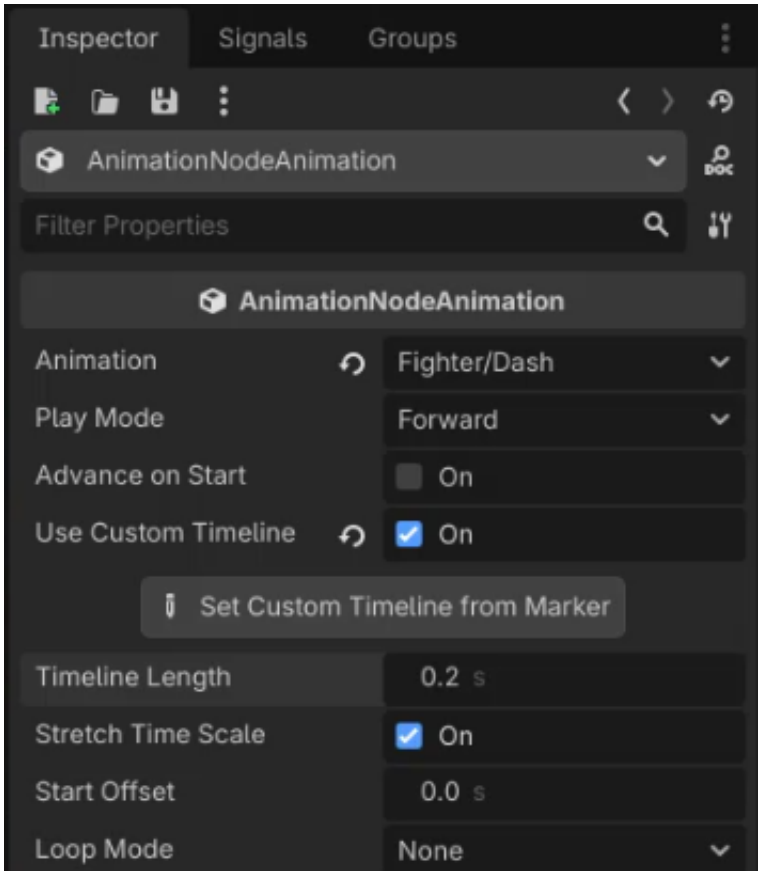
Adding the Dash Animation

The first new animation to add is the dash. In the AnimationTree editor, right-click and add a new animation node, selecting the Fighter Dash animation. This node needs a transition back to Standing, since the fighter should return to the standing state once the dash is complete.

To ensure the transition only occurs after the dash animation finishes playing, select the transition arrow between Dash and Standing, open the Inspector, navigate to the Switch section, and change the Switch Mode from Immediate to At End. Playing the dash animation now confirms that it transitions back to Standing only after completion.

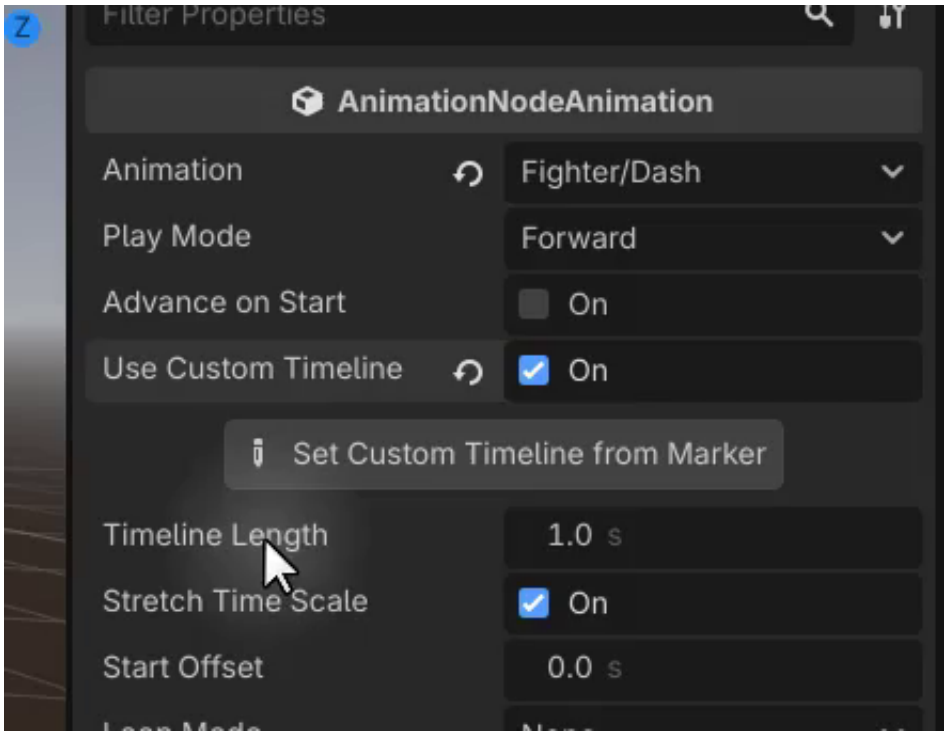


Rename the node from “Fighter_Dash” to simply “Dash” for clarity. Since the dash should be very quick in gameplay, enable Use Custom Timeline on the Dash animation node in the Inspector and reduce the Timeline Length to approximately 0.2 seconds. Playing the animation again shows it executes much faster.



Adjusting Jump Animation Speeds

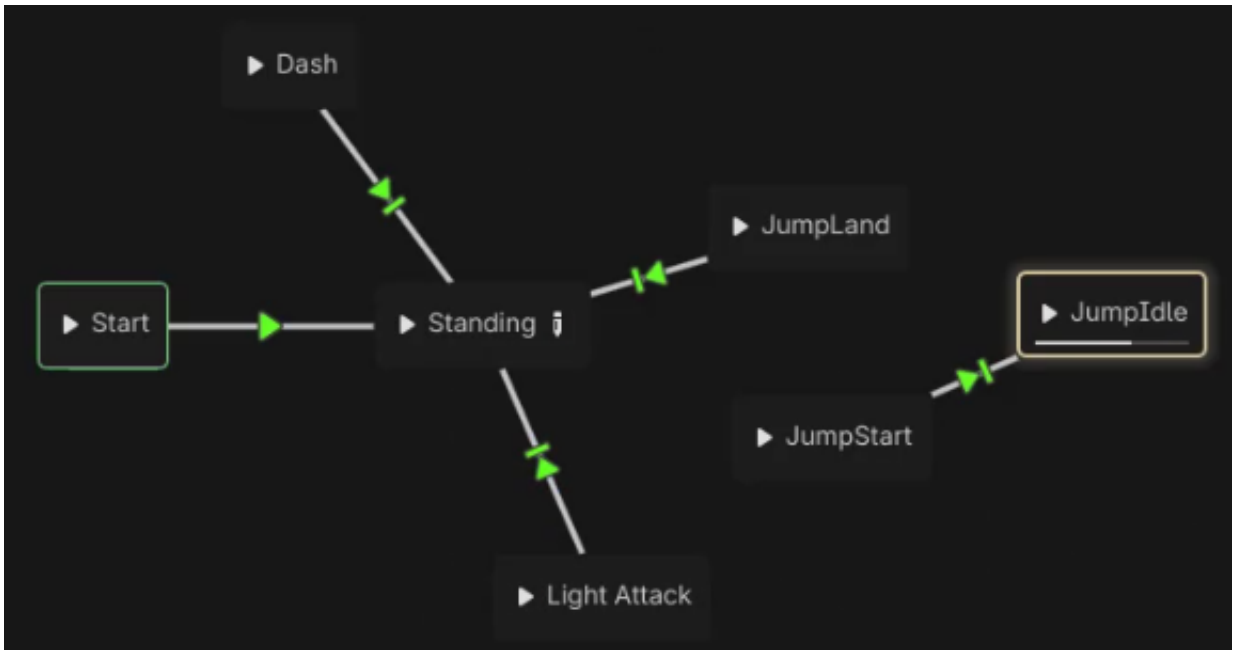
The same technique can be applied to the jump animations to make them feel more responsive. Select the JumpStart animation node, enable Use Custom Timeline, and set its Timeline Length to 0.3 seconds. Testing the game confirms the character now jumps noticeably faster.



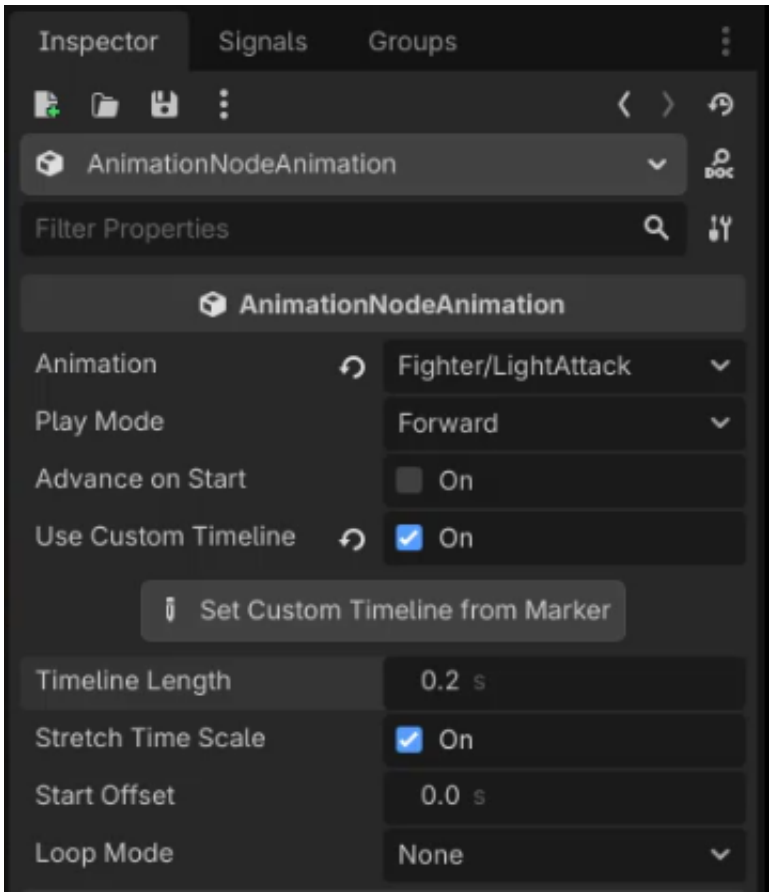
The JumpLand animation also benefits from the same adjustment. Select it in the AnimationTree, enable Use Custom Timeline, and set the Timeline Length to 0.3 seconds as well. Testing in-game shows both the jump and landing animations play much quicker, which feels more appropriate for a fast-paced fighting game.

Adding the Light Attack Animation

With the movement animations adjusted, the next step is to add the attack animations. Right-click in the AnimationTree editor, add a new animation node, and select the light attack animation. Rename it to “Light Attack” and create a transition from Light Attack back to Standing. Set the transition’s Switch Mode to At End so the fighter returns to the standing state only after the attack animation finishes.



The default light attack animation plays too slowly for a quick jab. Enable Use Custom Timeline on the Light Attack node and reduce the Timeline Length to about 0.2 seconds. The resulting animation feels much snappier and more fitting for a light attack in a fighting game.



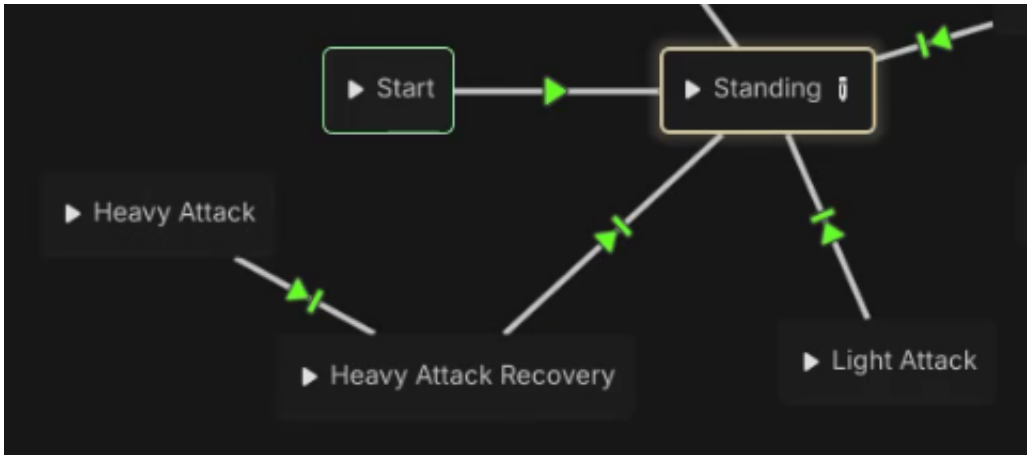
Adding the Heavy Attack Animation

The heavy attack requires two animation nodes because the animation pack splits it into two parts.

The first animation, Heavy Attack, shows the swing motion but freezes at the end. The second animation, Heavy Attack Recovery, returns the character to its default standing pose.

Add both animation nodes to the AnimationTree and rename them to “Heavy Attack” and “Heavy Attack Recovery” respectively. Create transitions in this order:

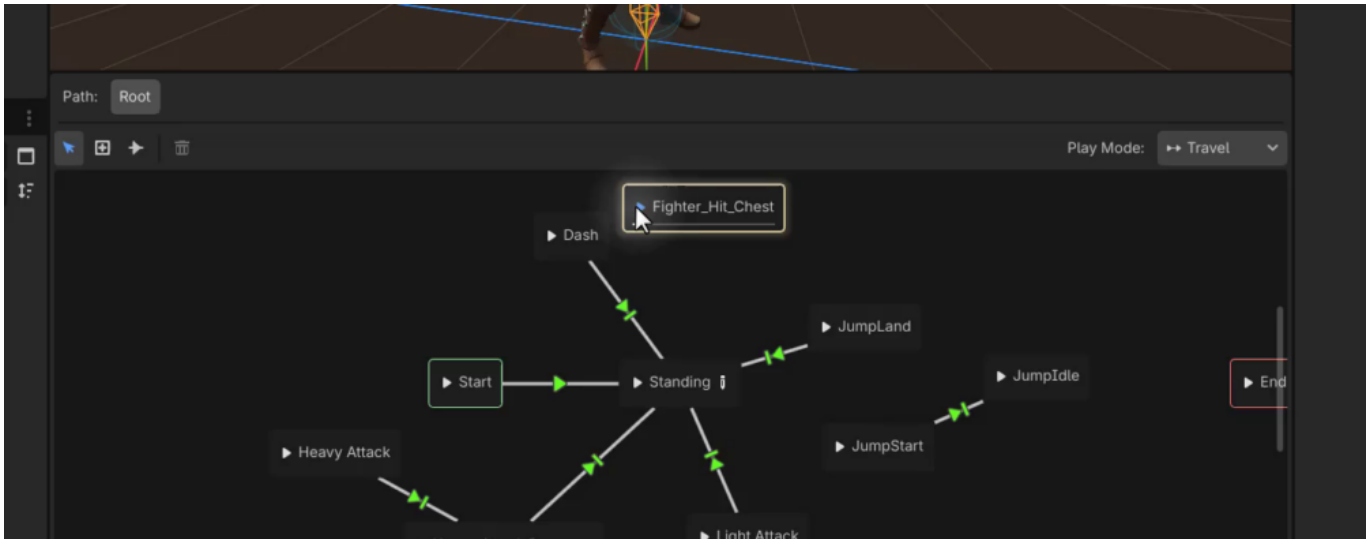
1. From Heavy Attack to Heavy Attack Recovery, with the Switch Mode set to At End.
2. From Heavy Attack Recovery to Standing, also with the Switch Mode set to At End.



With both transitions set to At End, playing the Heavy Attack animation now shows the full sequence: the character swings and then recovers to the standing pose. To speed things up, enable Use Custom Timeline on the Heavy Attack node and set its Timeline Length to 0.4 seconds. Apply the same to Heavy Attack Recovery with a shorter duration. These values can be fine-tuned later to achieve the desired pacing between light and heavy attacks.

Adding the Hit Chest Animation

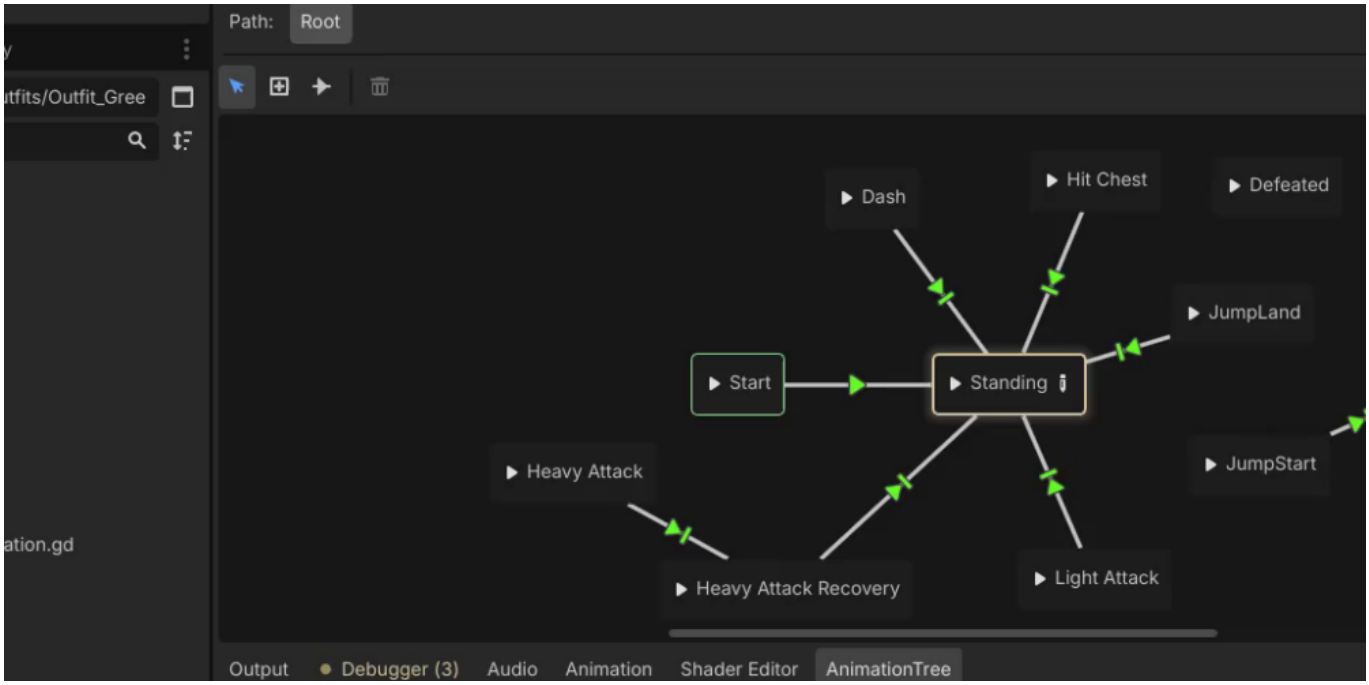
When a fighter takes damage, a hit reaction animation should play. Right-click in the AnimationTree editor, add a new animation node, and select the Hit Chest animation. Rename it to “Hit Chest” and create a transition back to Standing with the Switch Mode set to At End. This ensures the fighter briefly reacts to being struck before returning to the standing state.



The hit animation is already fairly quick by default, so no custom timeline adjustment is needed for now, though it can be tweaked later if necessary.

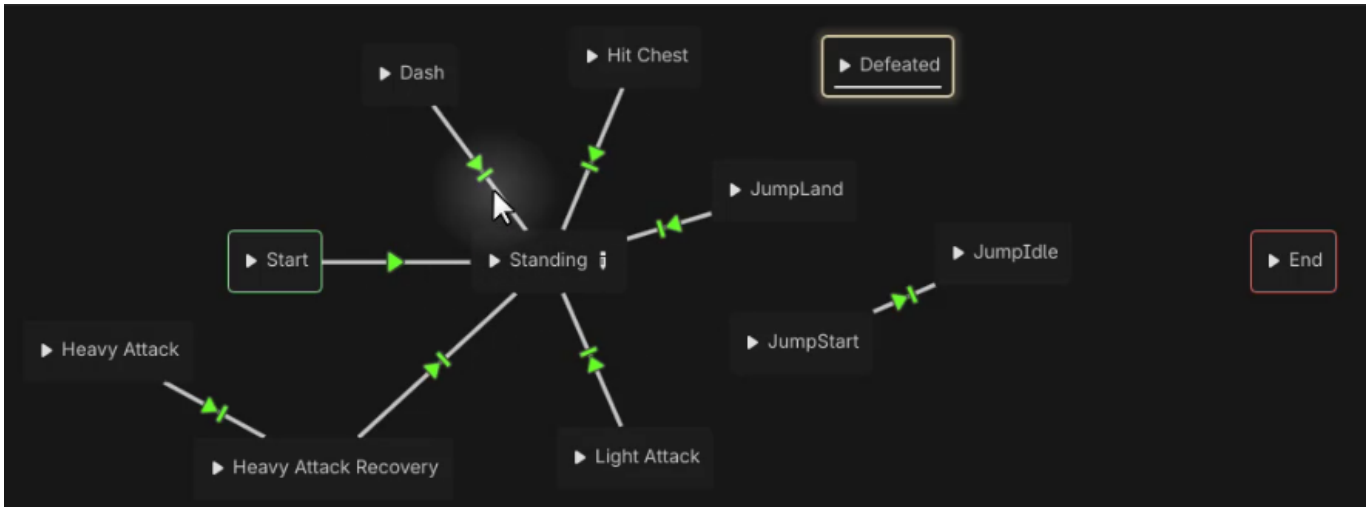
Adding the Defeated State

The final animation state is Defeated. Add a new animation node with the defeated animation and rename it to “Defeated.” Unlike all the other states, this one does not have a transition back to Standing. Once a fighter enters the Defeated state, they remain in that state for the rest of the match until the game returns to the menu. There is no need for outgoing connections since the fighter cannot perform any other actions after being defeated.



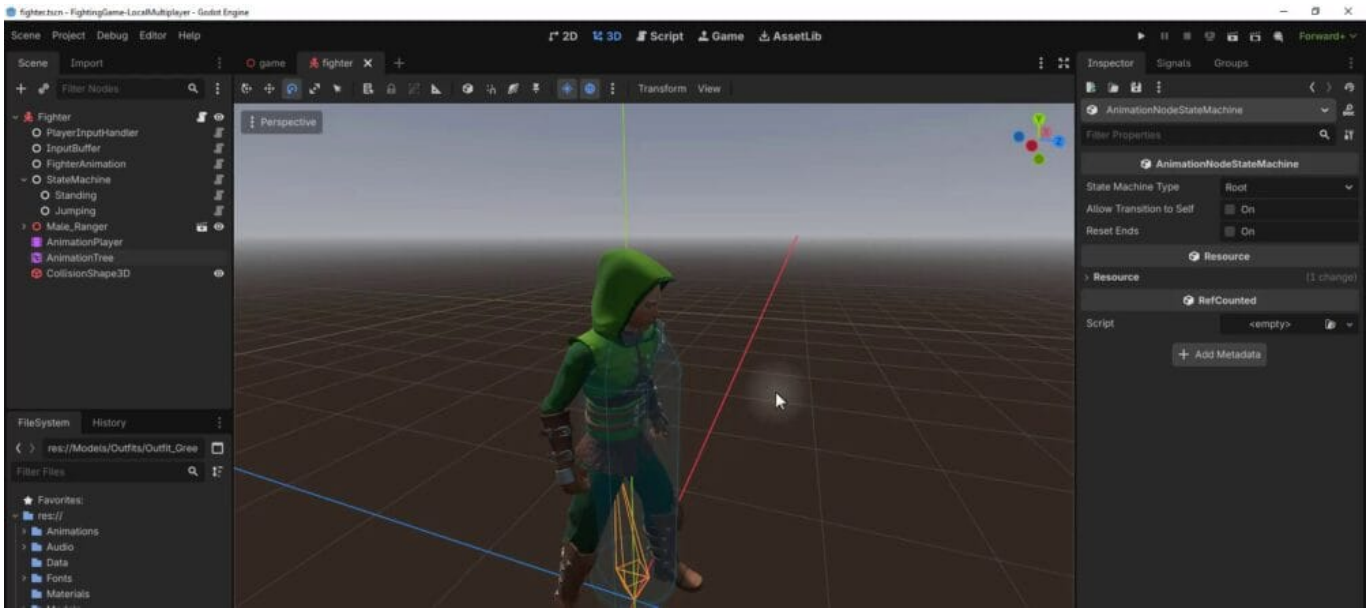
Complete Animation Tree Overview

With all the states in place, the AnimationTree for the fighter is fully configured. The Standing state sits at the center, with transitions leading to and from Dash, Light Attack, Heavy Attack (through Heavy Attack Recovery), Hit Chest, and the jump sequence (JumpStart, JumpIdle, JumpLand). The Defeated state exists separately with no outgoing transitions.



This setup ensures that when the corresponding game states are created in future lessons, the animations will already be properly configured and ready to be triggered through the FighterAnimation class.

In this lesson, we begin implementing the attacking states for our fighting game. The game will feature two types of attacks: a **light attack**, which is quick but deals less damage, and a **heavy attack**, which is slower but deals more. Both attack types share a common base class called AttackingState, which inherits from the base State class. This lesson covers creating the base attacking state, setting up the light attack, triggering it from the standing state, adding a duration-based auto-transition back to standing, and smoothing animation transitions with crossfade.



Creating the AttackingState Base Class

Inside the **Scripts/State Machine** folder, create a new script called attacking_state.gd. This script extends the base State class and will serve as the shared template for both light and heavy attacks. For now, we start with a single exported variable, animation_name, which stores the name of the animation to play from the AnimationTree when the state is entered.

The enter() function calls super.enter() to run any default setup logic from the base state, then triggers the appropriate animation using the animation_name value:

```
class_name AttackingState
extends State

@export var animation_name : String

func enter ():
 super.enter()
 animation.set_animation(animation_name)
```

Creating the Light Attack State Script

With the base class in place, create a new script called light_attack_state.gd in the same folder. This script simply extends AttackingState and declares its own class name. Although the script body is currently empty, having separate scripts for light and heavy attacks is important for future functionality such as stamina costs and enemy AI event tracking.

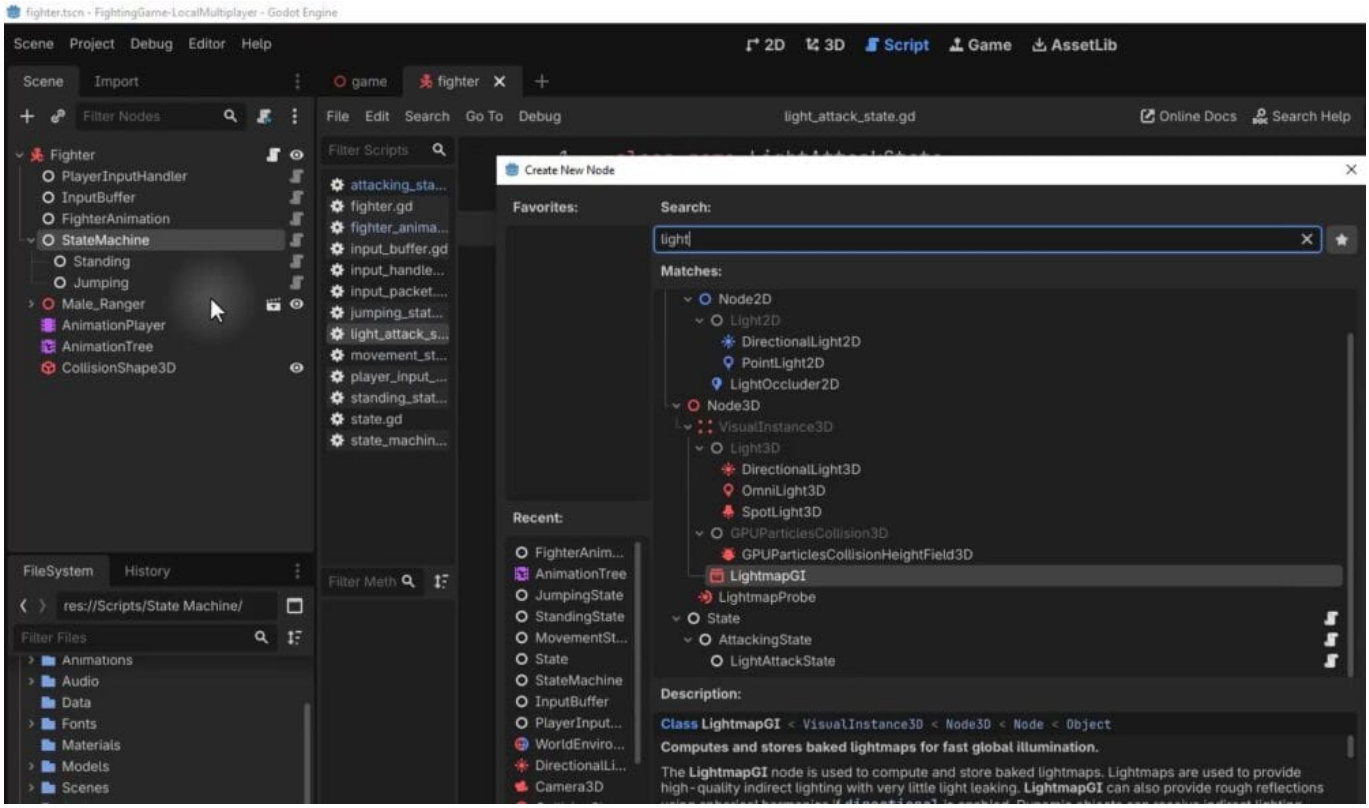
```
class_name LightAttackState
```



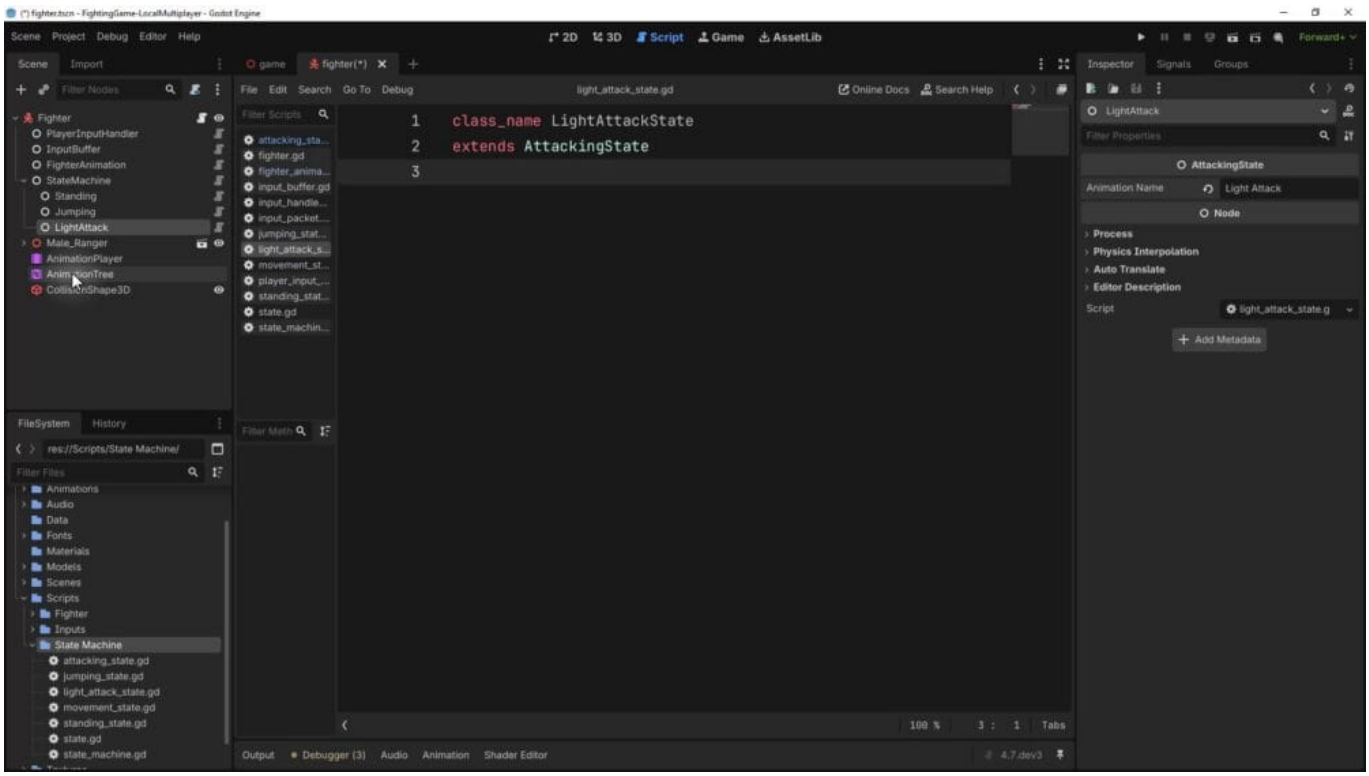
```
extends AttackingState
```

Adding the Light Attack Node to the State Machine

In the Scene tree, right-click on the **StateMachine** node and select **Add Child Node**. Search for and add a LightAttackState node, then rename it to **LightAttack**.



In the Inspector, set the **Animation Name** property to Light Attack. This must match the exact name of the corresponding animation state in the AnimationTree.

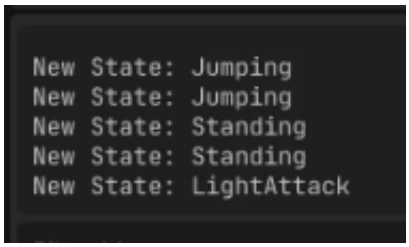


Transitioning to the Light Attack State

The StandingState acts as the central hub from which all other states are reached. To transition into the light attack, open standing_state.gd and add an input check for the light_attack action. When detected, the state machine transitions to the LightAttack state. It is important to add a return statement after each change_state() call to prevent multiple state transitions from occurring in the same frame. For example, if both light attack and jump are pressed simultaneously, without the return the jump check would always win since it runs later in the function.

```
class_name StandingState
extends MovementState

func update (delta : float):
 super.update(delta)

 if input_buffer.is_pressed("light_attack"):
  state_machine.change_state("LightAttack")
  return

 if input_buffer.is_pressed("jump"):
  state_machine.change_state("Jumping")
  return
```

Running the game at this point, the fighter can perform a light attack swing when the attack button is pressed. However, the character gets stuck in the attacking state because there is no logic to transition back to standing.



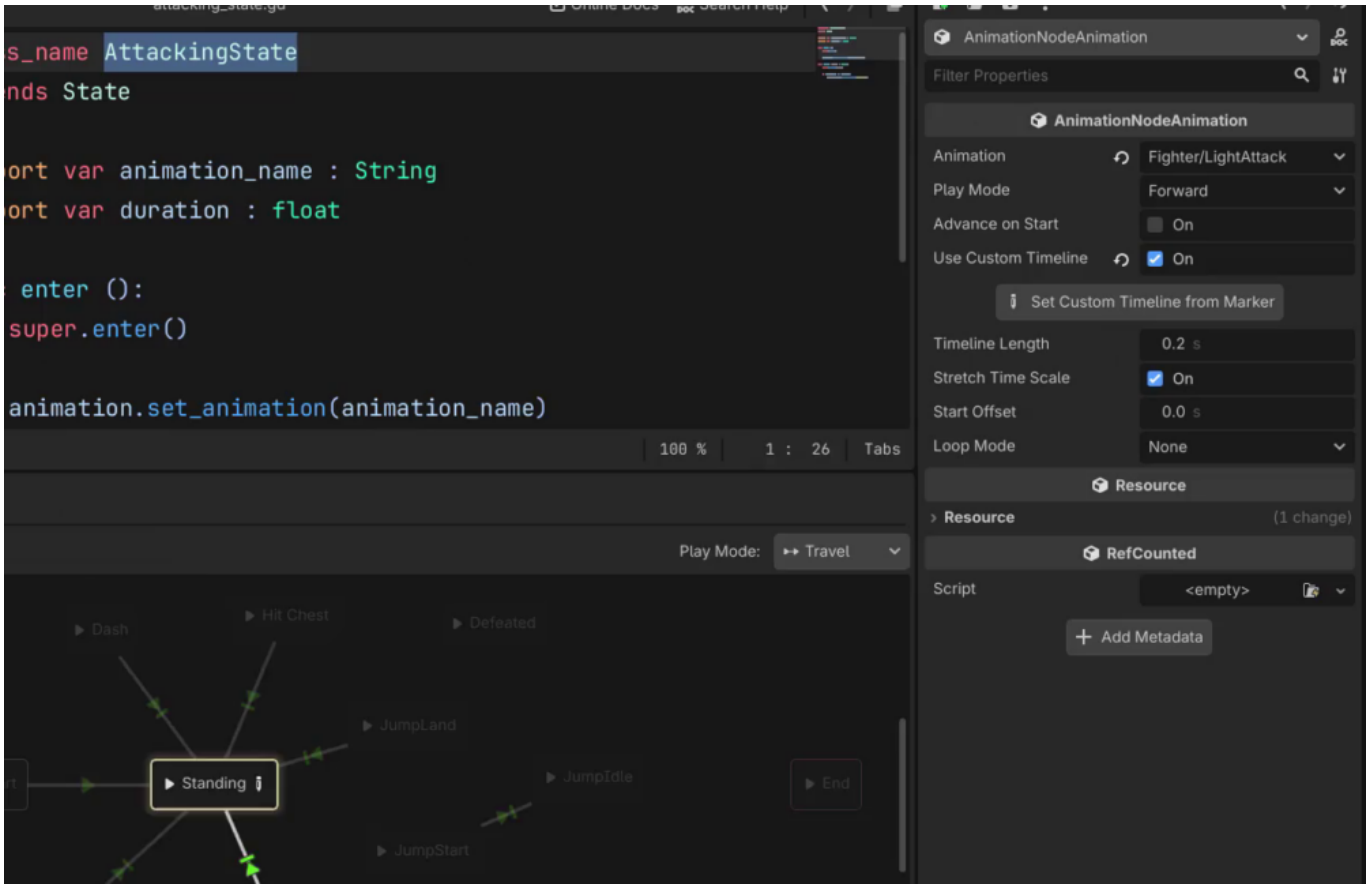
Adding Duration and Auto-Transition

To solve this, the AttackingState needs a duration after which it automatically transitions back to the Standing state. Add an exported duration variable and an update() function that checks the elapsed time. The base State class tracks local_time, which represents the time elapsed since entering the current state. When local_time exceeds the duration, the state machine changes back to Standing:

```
class_name AttackingState
extends State

@export var animation_name : String
@export var duration : float

func enter ():
 super.enter()
 animation.set_animation(animation_name)

func update (delta : float):
 super.update(delta)

 if local_time >= duration:
  state_machine.change_state("Standing")
```

Setting the Duration Value

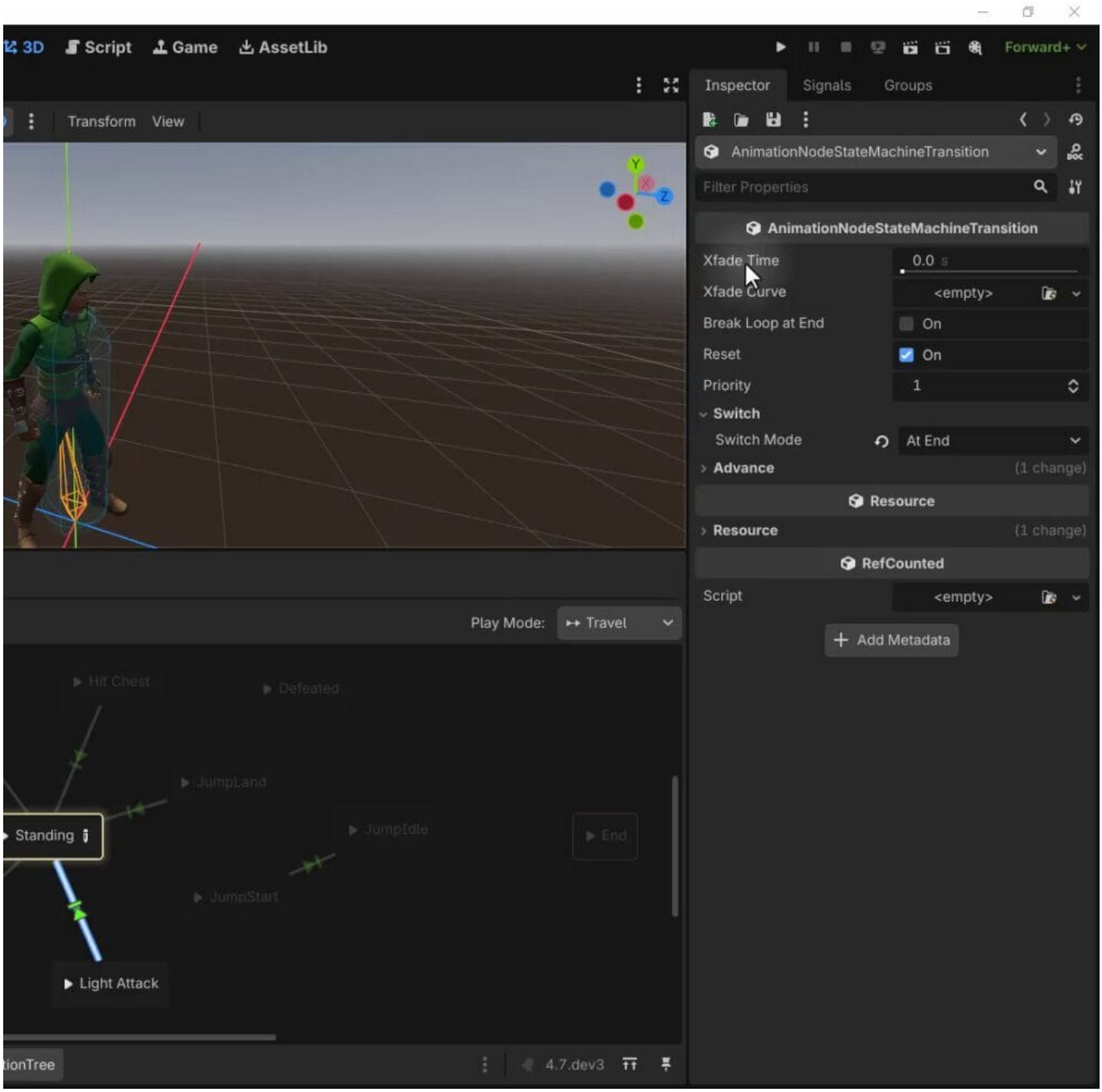
To determine the appropriate duration, select the **Light Attack** animation in the AnimationTree and check its timeline length. The light attack animation is approximately **0.2 seconds** long. Setting the state duration to **0.3 seconds** provides a small amount of leeway beyond the animation length.



With the duration configured, running the game now shows the fighter attacking and correctly reverting back to the standing state. However, the transition between the attack animation and the standing animation appears jittery because the two animation poses do not match at the transition point.

Smoothing Transitions with Xfade Time

To fix the jittery animation, we can add a crossfade between states in the AnimationTree. Select the transition arrow going from **Light Attack** back to **Standing** in the AnimationTree graph. In the Inspector, locate the **Xfade Time** property on the AnimationNodeStateMachineTransition. Setting this to **0.1 seconds** creates a smooth blend between the end of the attack animation and the standing pose, eliminating the visual snap.



Apply the same 0.1-second Xfade Time to other transitions as well, such as **Heavy Attack** to **Standing** and **Dash** to **Standing**. Some transitions like **Hit Chest** to **Standing** and **Jump Land** to **Standing** may not need a crossfade if their end poses already match the standing animation.

After configuring the crossfade values, the attacks transition back to standing smoothly and naturally.

Challenge

As a challenge, try creating the **heavy attack state** on your own. Follow the same process used for the light attack: create a HeavyAttackState script that extends AttackingState, add it as a child node of the StateMachine, configure its animation name and duration in the Inspector, and add the input check in StandingState to transition to it when the heavy attack button is pressed.

In this lesson, we continue building out the attacking states for our fighting game by implementing the heavy attack, adding attack cooldowns, resetting movement velocity during attacks, and introducing a forward force system that pushes the fighter in their facing direction when they swing.

Creating the Heavy Attack State

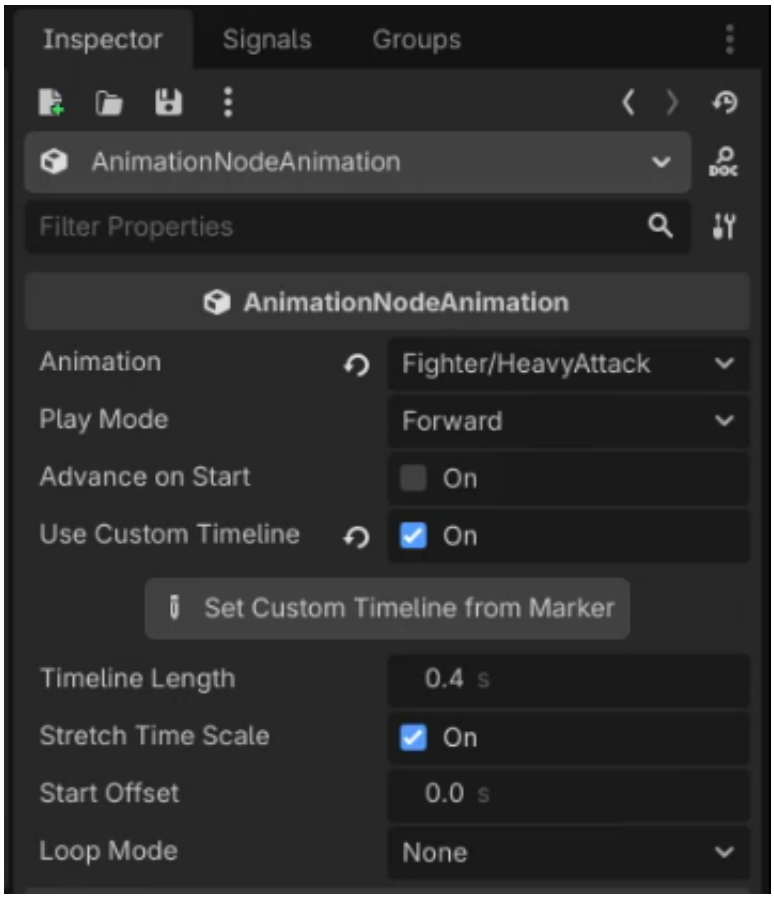
The first step is to create a new script for the heavy attack. This script extends from AttackingState, the same base class used by the light attack. Since all the shared attack logic lives in the base class, the heavy attack state script itself requires no additional code beyond the class declaration.

```
class_name HeavyAttackState
extends AttackingState
```

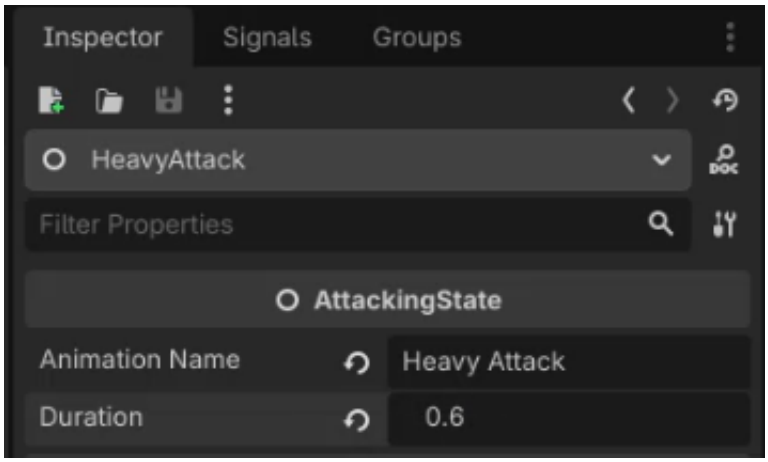
With the script created, add a new child node to the **StateMachine** node in the scene tree. Select **HeavyAttackState** as the node type, then rename it to **HeavyAttack** (removing the “State” suffix to keep naming consistent with the other states).

Configuring the Heavy Attack in the Inspector

Select the **HeavyAttack** node and set its properties in the Inspector. The **Animation Name** should be set to Heavy Attack. To determine the correct **Duration**, check the AnimationTree to see the total length of the heavy attack animation sequence: the main attack animation is 0.4 seconds, the recovery is 0.1 seconds, and the transition is 0.1 seconds.



Adding these together gives a total duration of **0.6 seconds**.

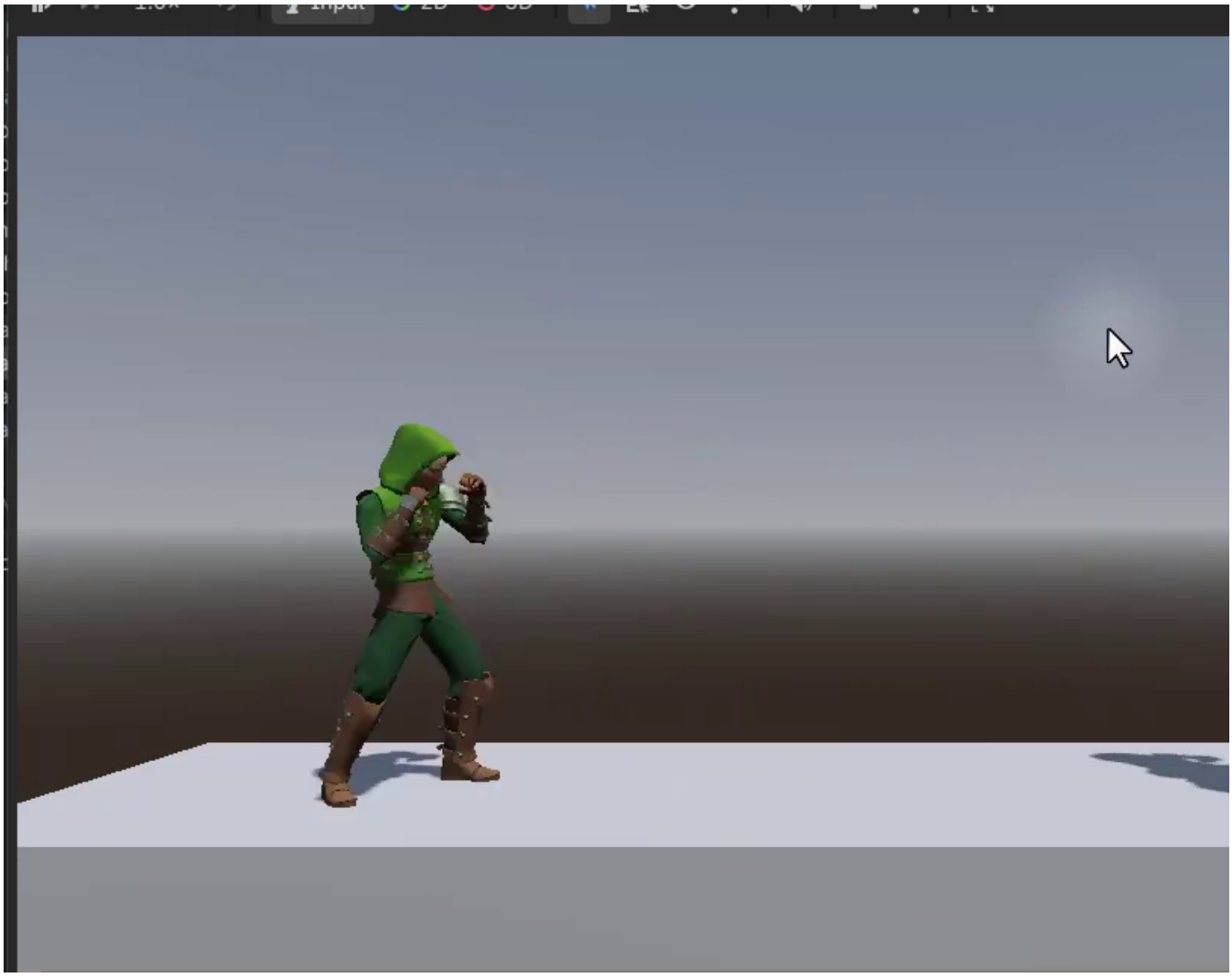


Detecting Heavy Attack Input

To allow the player to trigger the heavy attack, open the standing_state.gd script and add a new input check. Following the same pattern used for the light attack, check whether the heavy_attack button is pressed and transition to the HeavyAttack state if so.

```
if input_buffer.is_pressed("heavy_attack"):
 state_machine.change_state("HeavyAttack")
 return
```

Running the game at this point confirms that both the light attack and heavy attack are now functional. The light attack is a quick punch, while the heavy attack plays a longer, more deliberate animation.



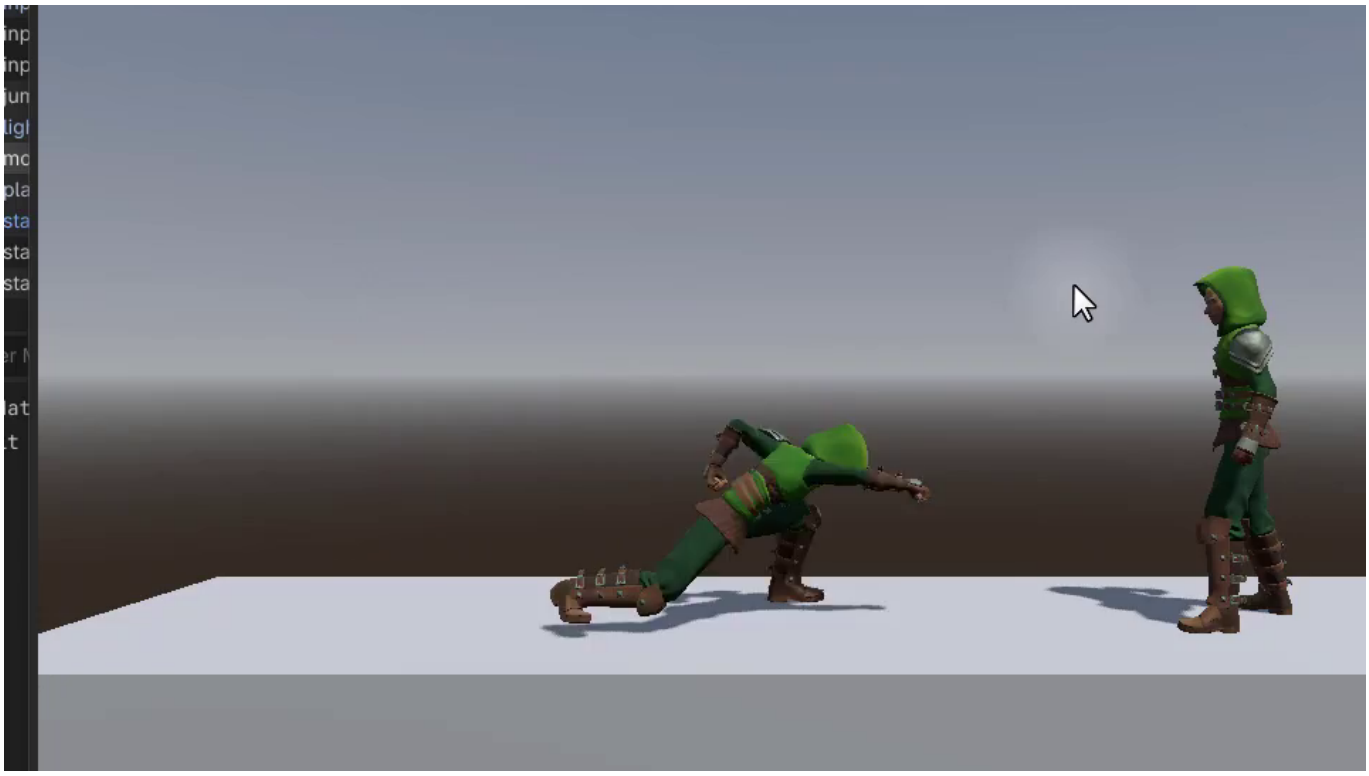
Resetting Velocity on Movement State Exit

One issue becomes apparent when testing: if the player is moving and then attacks, the fighter retains whatever movement velocity they had when they left the standing state. This causes the character to slide during their attack animation, which does not look correct.

To fix this, open the movement_state.gd script and add an exit function. This function resets the fighter's horizontal velocity to zero whenever the player leaves a movement-based state (such as standing or jumping) and transitions into a non-movement state (such as attacking).

```
func exit ():
 super.exit()

 fighter.move_velocity.x = 0
```

With this change, the fighter now stops moving the moment they begin an attack, which feels much more responsive and polished.



Adding Attack Cooldowns

Another issue is that holding down an attack button causes the fighter to attack every single frame as soon as the previous attack ends. To prevent this spamming behavior, a cooldown system is needed.

Open the attacking_state.gd script and add two new variables: an exported cooldown float that can be configured per attack in the Inspector, and a cooldown_end_time float that tracks when the fighter is allowed to re-enter the state.

In the exit function, set the cooldown_end_time to the current system time plus the cooldown duration. This records the earliest point at which the attack can be used again.

```
@export var cooldown : float
var cooldown_end_time : float

func exit ():
 super.exit()
 cooldown_end_time = Time.get_unix_time_from_system() + cooldown
```

Next, override the can_enter function to check whether enough time has elapsed. If the current time is still less than the cooldown_end_time, the function returns false to prevent state entry. Otherwise, it returns true.

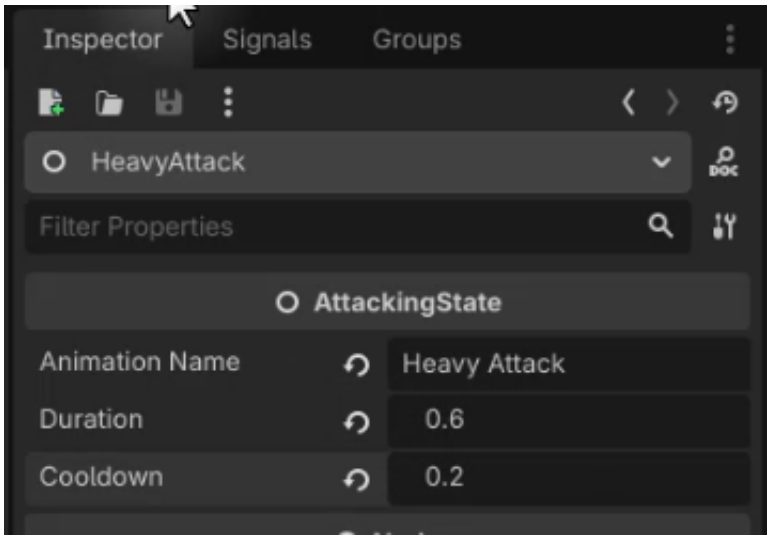
```
func can_enter () -> bool:
 if Time.get_unix_time_from_system() < cooldown_end_time:
  return false

 return true
```


Enforcing the Cooldown in the State Machine

The cooldown logic alone is not enough — the state machine itself must actually respect the can_enter check. Open the state_machine.gd script and find the change_state function. Add a check right before the state transition that calls can_enter on the target state. If it returns false, the function returns early without performing the state change.

```
if not states[state_name].can_enter():
 return
```

With this in place, set the **Cooldown** property on both the **LightAttack** and **HeavyAttack** nodes to **0.2** seconds in the Inspector. This introduces a short delay between consecutive attacks that prevents spamming while still feeling responsive.



Adding Forward Force to Attacks

To make attacks feel more dynamic, the fighter should be pushed slightly forward in the direction they are facing when they swing. This requires building a force system in the fighter.gd script.

Add two new variables: horizontal_force to track the current force applied along the X-axis, and drag to gradually reduce that force over time. The drag value should be between 0 and 1 — a value of 0.8 provides a smooth, gradual falloff.

```
var horizontal_force : float
var drag : float = 0.8
```

In the _movement function, apply drag to the horizontal force each frame by multiplying it by the drag value, then add the horizontal force to the X velocity alongside the regular movement velocity.

```
func _movement (delta : float):
 horizontal_force *= drag

 velocity.x = move_velocity.x + horizontal_force
```



```
velocity.y = move_velocity.y
move_and_slide()
```

Create a public add_force function so that other systems (like the attacking state) can push the fighter.

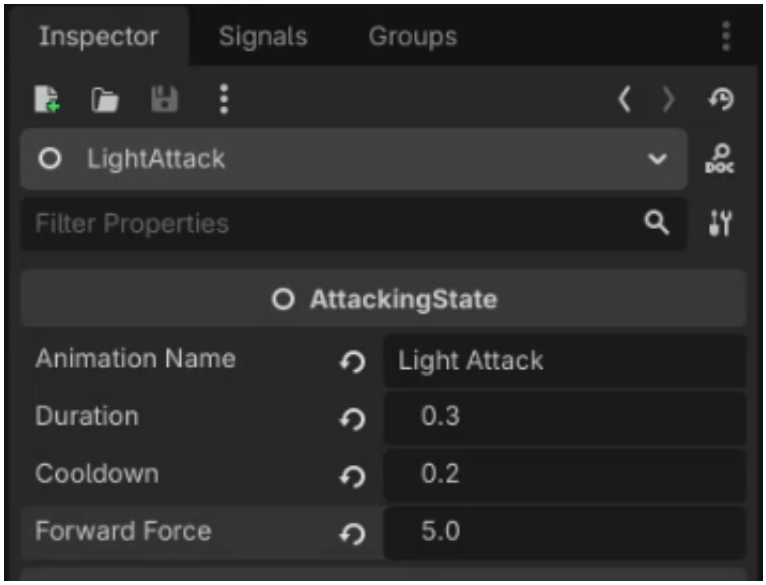
```
func add_force (force : float):
 horizontal_force += force
```

Applying Forward Force in the Attacking State

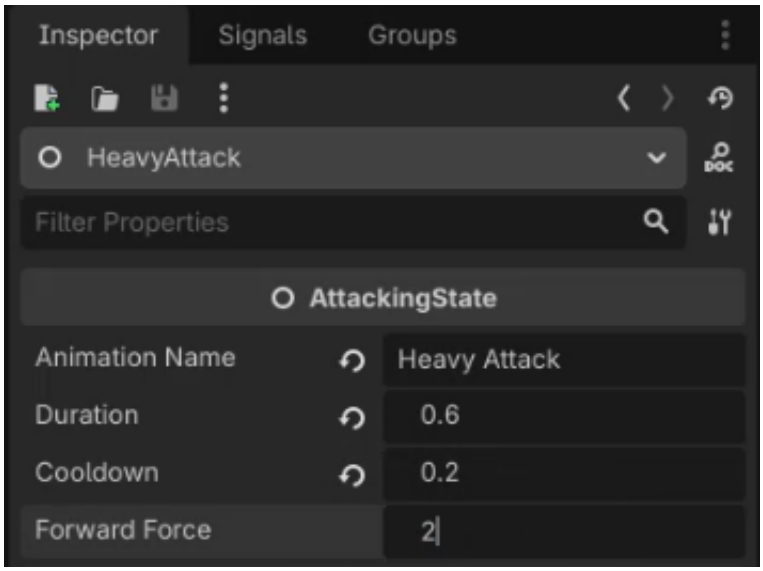
Back in the attacking_state.gd script, add an exported forward_force variable. In the enter function, call fighter.add_force with the forward force multiplied by fighter.forward_direction. The forward_direction property is either 1 or -1 depending on which way the fighter is facing, ensuring the force always pushes the fighter toward their opponent.

```
@export var forward_force : float

func enter ():
 super.enter()

 animation.set_animation(animation_name)
 fighter.add_force(forward_force * fighter.forward_direction)
 has_hit = false
```

In the Inspector, set the **Forward Force** on the **LightAttack** node to a value such as **5.0** — this gives the quick punch a noticeable forward push.



For the **HeavyAttack**, a smaller value like **2.0** works well since the heavy attack is a more grounded, stationary strike that should not move the fighter as far.



Summary

In this lesson, the heavy attack state was created by extending the AttackingState base class, configured in the Inspector with the correct animation name and duration, and wired up through the standing state's input checks. Movement velocity is now reset when leaving movement states, preventing unwanted sliding during attacks. A cooldown system was added to the attacking state with a corresponding can_enter check in the state machine, and a horizontal force system was built into the fighter to push the character forward when attacking.

In the next lesson, hit detection will be implemented to allow fighters to damage each other.

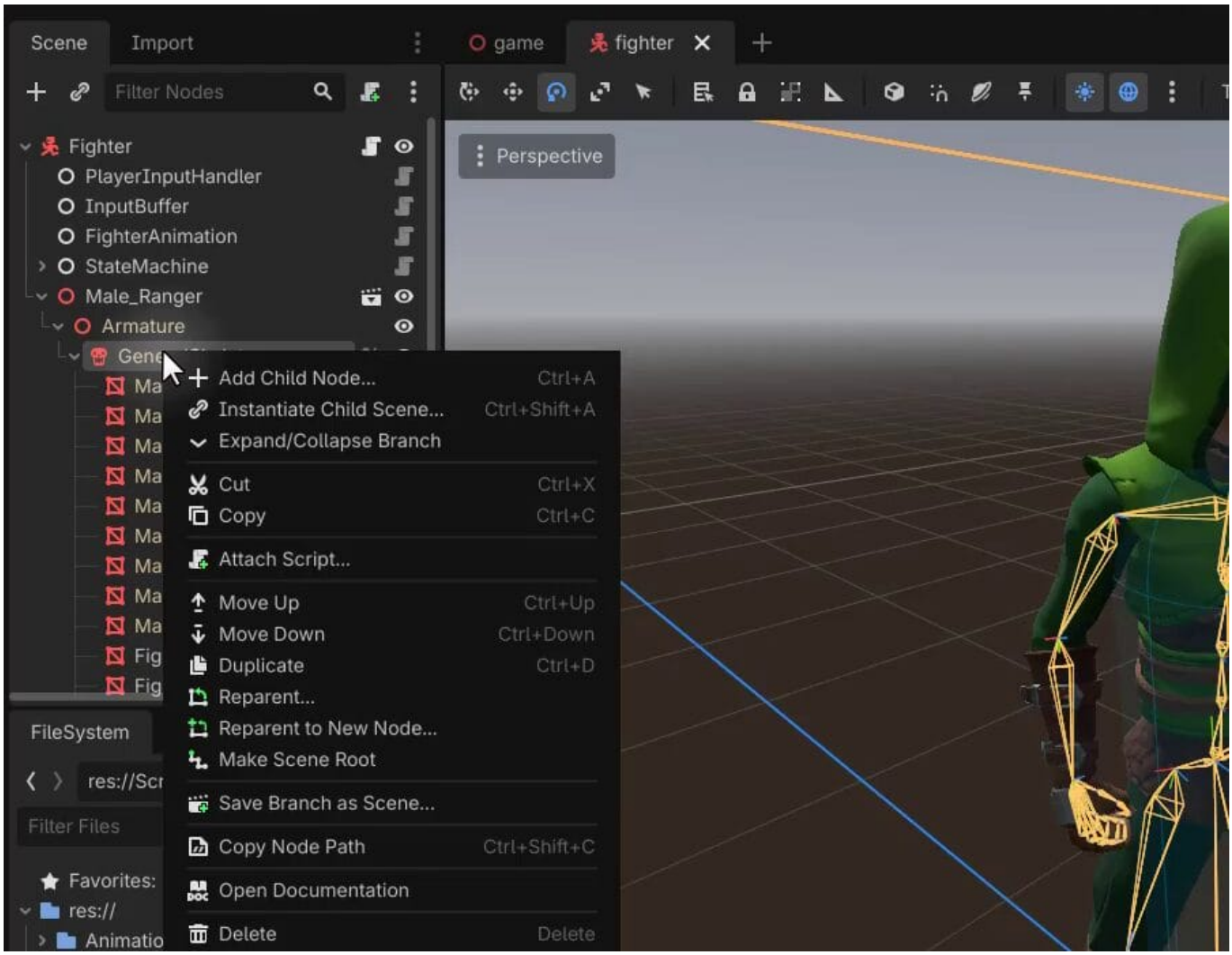
In this lesson, we begin setting up hitboxes for our fighting game characters. The hitbox system is divided into two types: **hit senders** and **hit receivers**. Hit senders are attached to the fighter's hands and detect when they overlap with an opponent's hit receivers, which are placed on areas such as the torso and head. When a hit sender overlaps a hit receiver, damage is dealt to the opponent.

Both hit senders and hit receivers use Area3D nodes, which act as triggers that detect when other collision areas have been overlapped. These Area3D nodes are attached to specific bones on the character's skeleton using a feature called **BoneAttachment3D**.

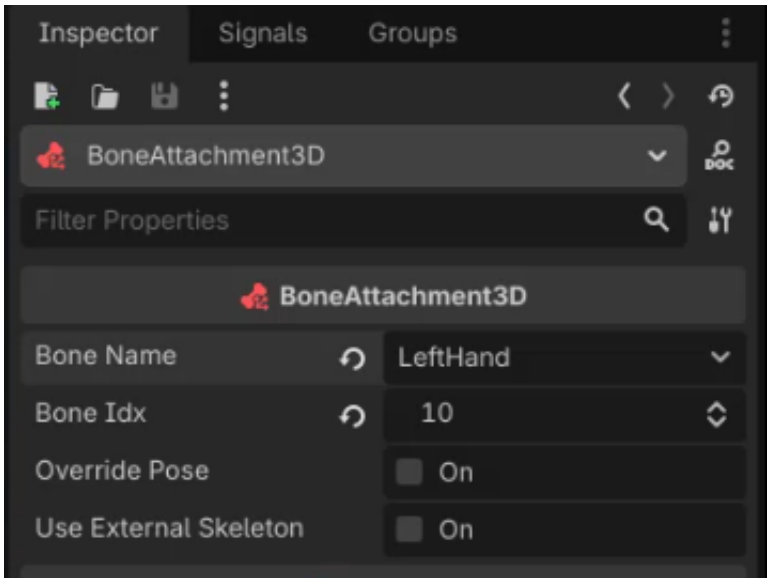
Understanding Bone Attachments

A BoneAttachment3D allows you to attach a node to a specific bone on a skeleton. This means that the attached node will follow the bone's position and rotation as the character animates, keeping the hitbox aligned with the correct body part at all times.

To create a bone attachment, expand the character model's hierarchy in the scene tree. Navigate to the **Armature** node, then to **GeneralSkeleton**. Right-click on the GeneralSkeleton node, select **Add Child Node**, and search for **BoneAttachment3D**.

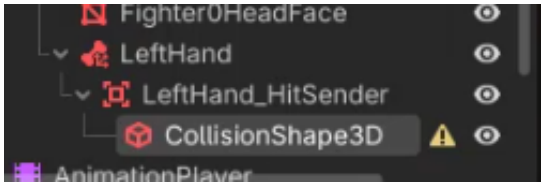


Once the BoneAttachment3D is added, select it and go to the Inspector panel. In the **Bone Name** property, choose the bone you want to attach to. For a hit sender on the left hand, select **LeftHand** from the dropdown. The node will now follow the left hand bone during animations.



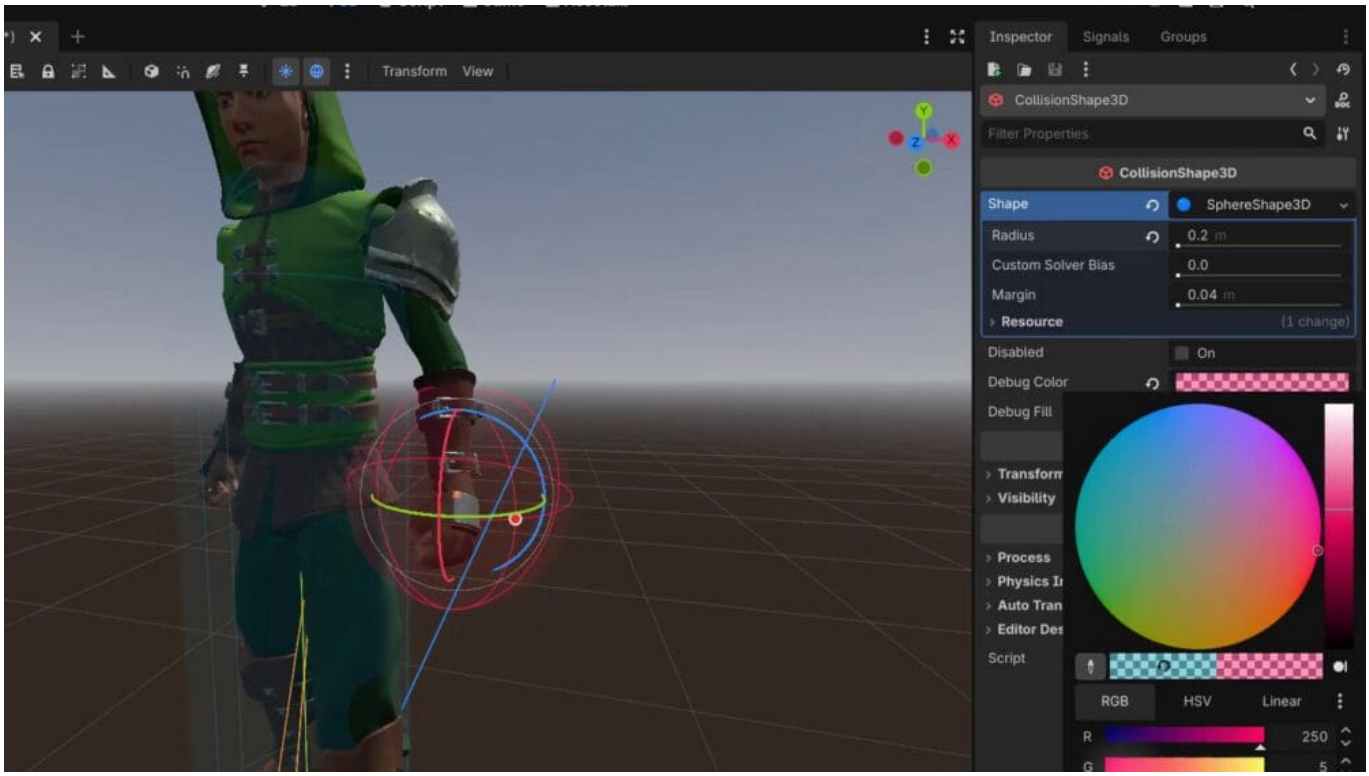
Creating a Hit Sender (Left Hand)

After creating and renaming the BoneAttachment3D to “LeftHand”, the next step is to add the Area3D that will serve as the hit sender. Right-click the LeftHand bone attachment and add a child **Area3D** node. Rename it to **LeftHand_HitSender**.



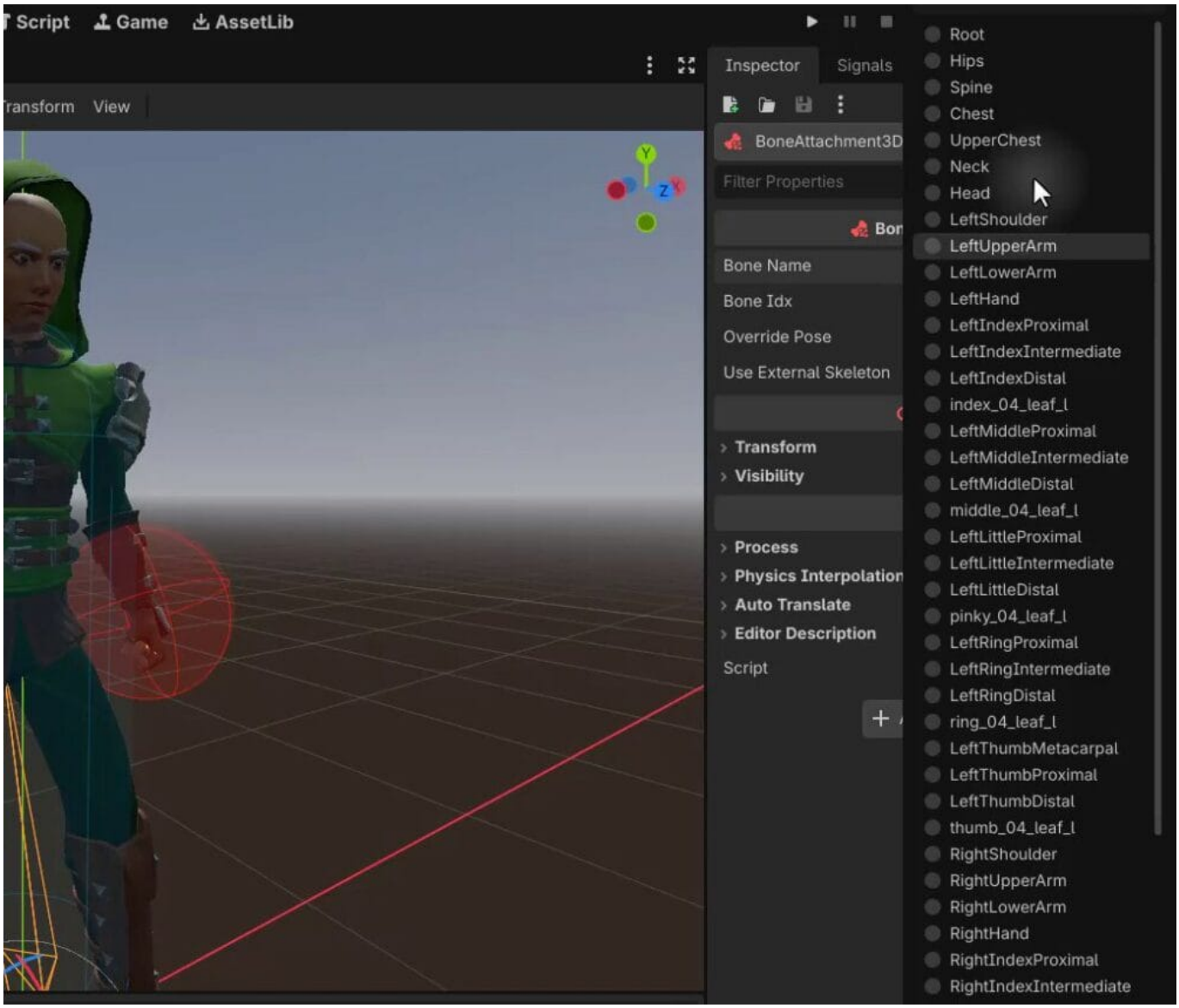
The Area3D needs a collision shape to define the area it covers. Right-click the LeftHand_HitSender node and add a child **CollisionShape3D** node. Set its shape to a **SphereShape3D**, which works well for a hand-sized hitbox. Adjust the radius to approximately **0.2** to keep it proportional to the character’s hand.

To make the collision shape easier to see during development, change the **Debug Color** property in the Inspector from the default blue to a reddish color. This visual distinction helps differentiate hit senders from hit receivers at a glance.

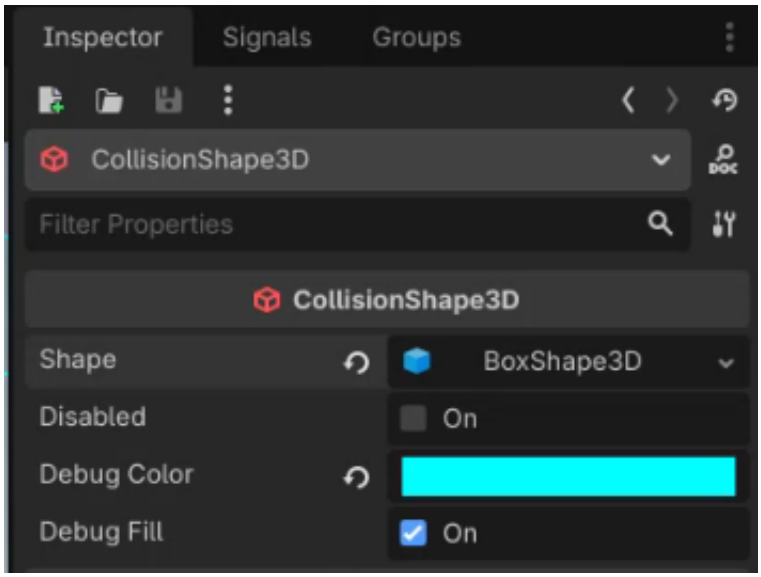


Creating a Hit Receiver (Chest)

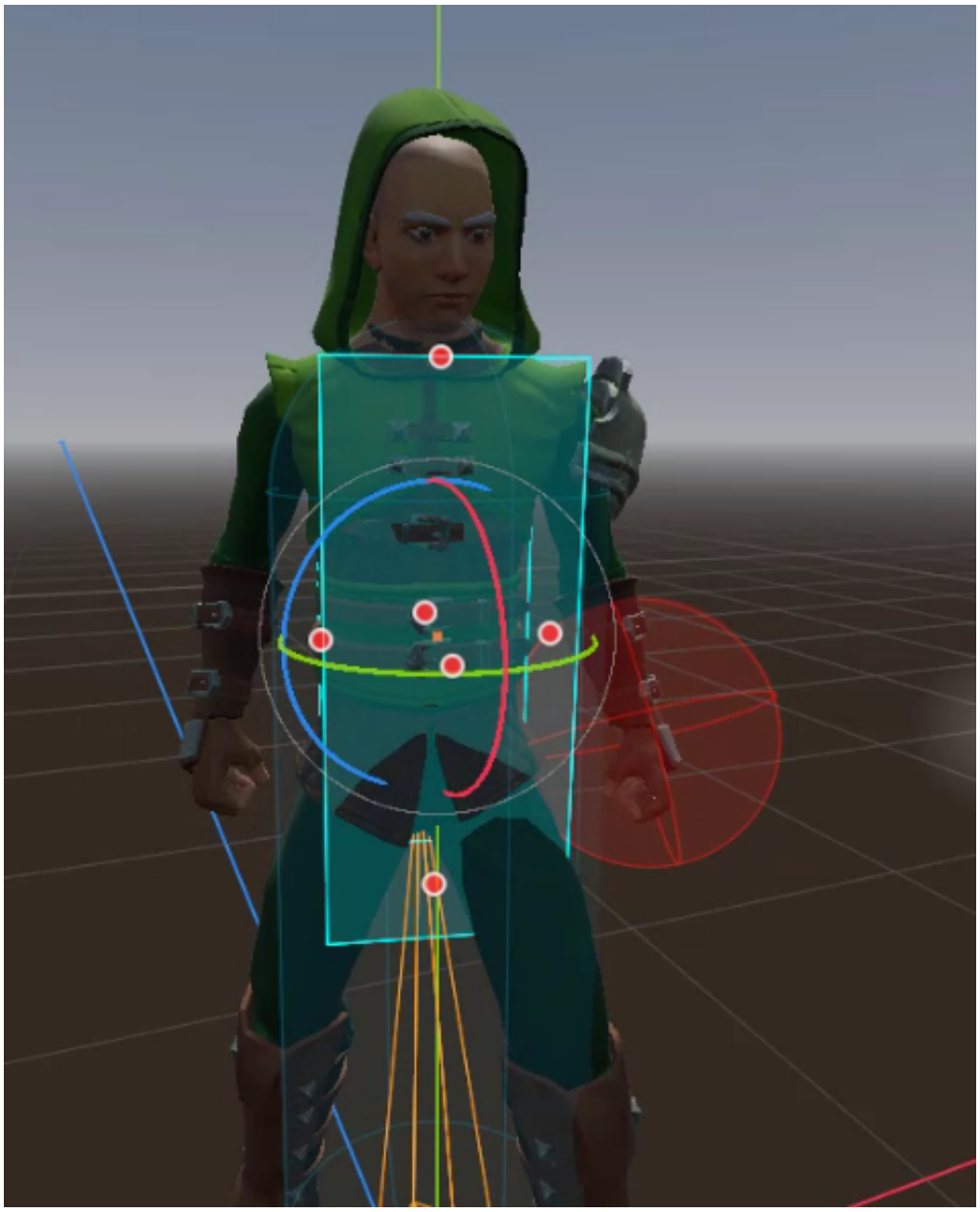
With the hit sender in place, the next step is to create a hit receiver for the character’s torso. This follows the same general process: create a new BoneAttachment3D under the GeneralSkeleton, then set its **Bone Name** to **Chest** in the Inspector. Rename the bone attachment node to “Chest” for clarity.



As a child of the Chest bone attachment, add an **Area3D** node and rename it to **Chest_HitReceiver**. Then add a **CollisionShape3D** as a child of the Area3D. For the torso, use a **BoxShape3D** instead of a sphere, as it better fits the rectangular shape of the character's upper body. Set the **Debug Color** to blue so that hit receivers are visually distinct from the red hit senders.



Resize the box shape to roughly fit the bounds of the character's torso. This can be adjusted later once the attack system is fully implemented and hits are connecting. The exact dimensions are flexible and can be tweaked to achieve the desired gameplay feel.



Challenge

As a practice exercise, try setting up the following additional hitboxes on your own:

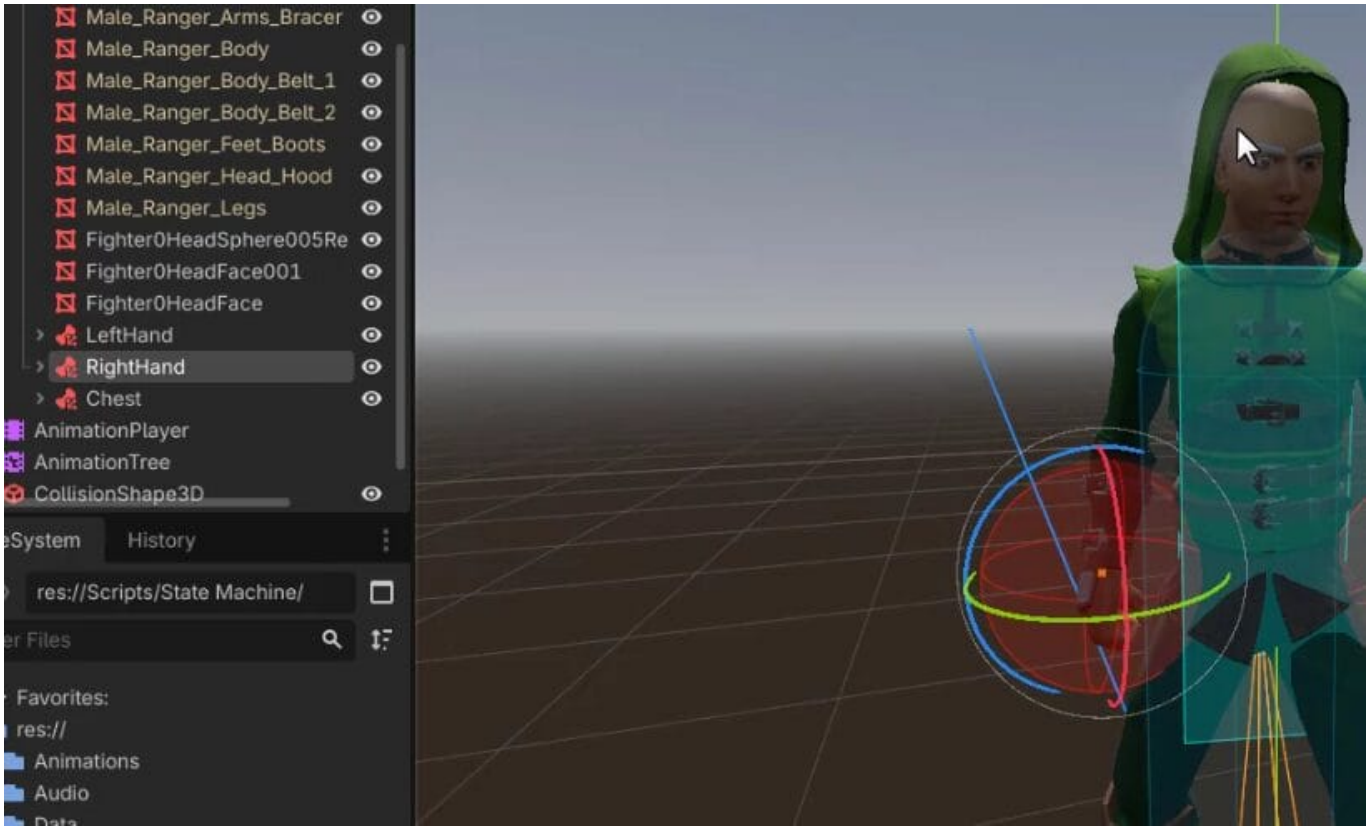
- A **hit receiver for the head** using a SphereShape3D collision shape, attached to the head bone
- A **hit sender for the right hand** following the same setup as the left hand hit sender

The code to make these hitboxes functional will be implemented in the following lessons.

In this lesson, we continue setting up the hitbox system for our fighting game character. We complete the challenge from the previous lesson by creating the remaining hit sender and hit receiver nodes, then write the HitReceiver and HitSender scripts that will power the hit detection logic.

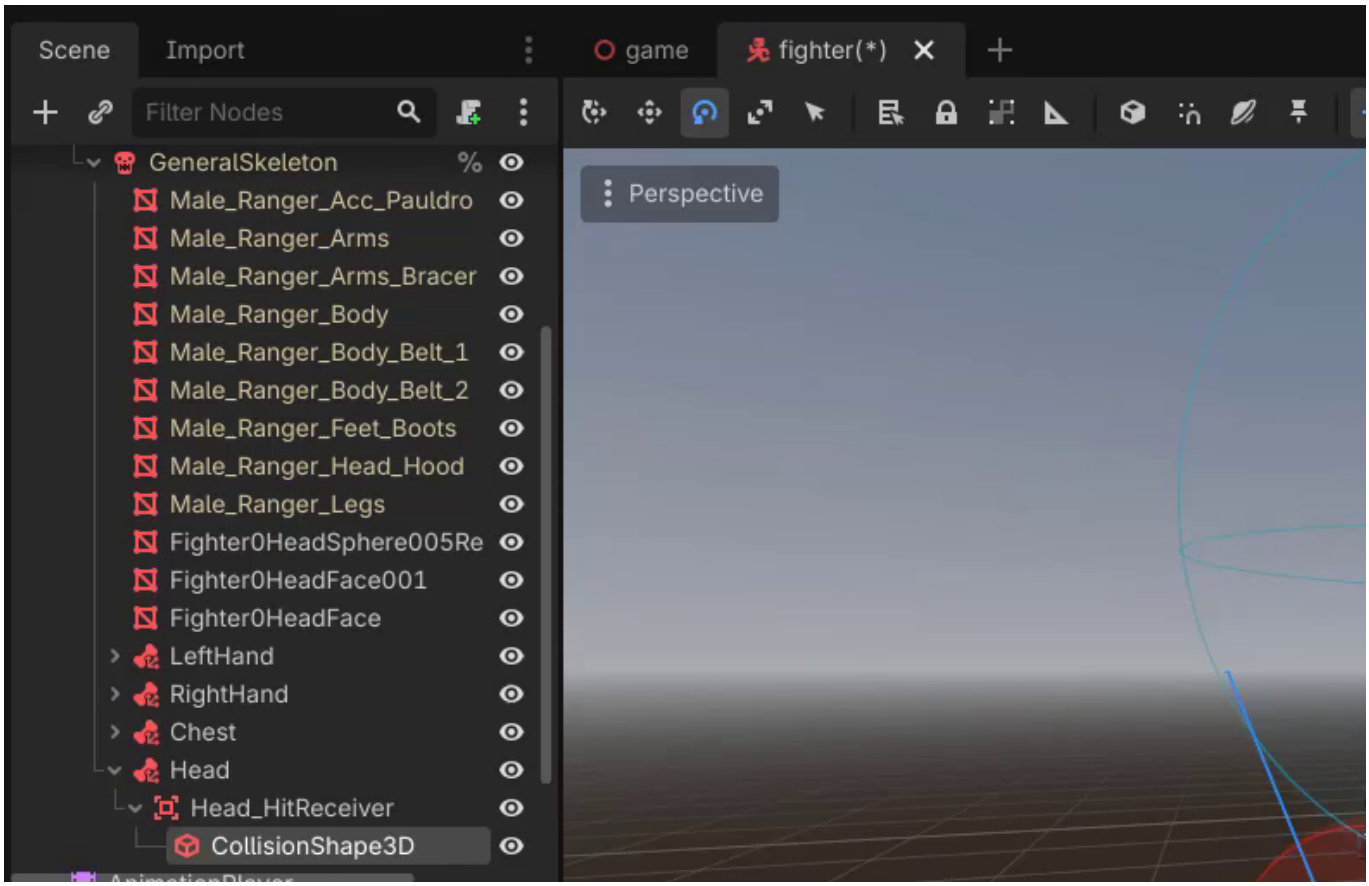
Setting Up the Right Hand Hit Sender

With the left hand hit sender already configured from the previous lesson, the first step is to create a matching hit sender for the right hand. Select the left hand node in the scene tree, duplicate it with **Ctrl+D**, and rename the duplicate to **RightHand**. In the Inspector, change the **Bone Name** property from “LeftHand” to “RightHand” so that the bone attachment targets the correct bone on the skeleton.

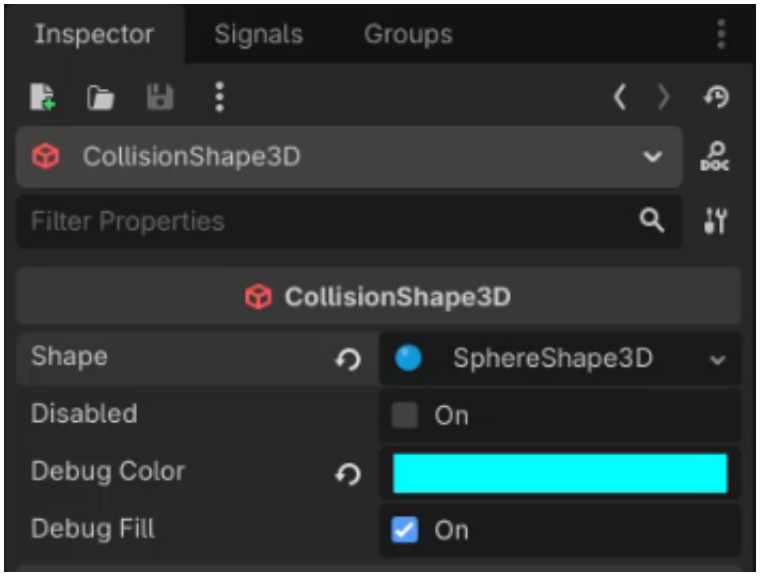


Creating the Head Hit Receiver

Next, we need to add a hit receiver for the character’s head. Unlike the right hand which was duplicated from an existing node, the head hit receiver is built from scratch. Create a new **BoneAttachment3D** node under the skeleton, rename it to **Head**, and set its **Bone Name** to “Head” in the Inspector.



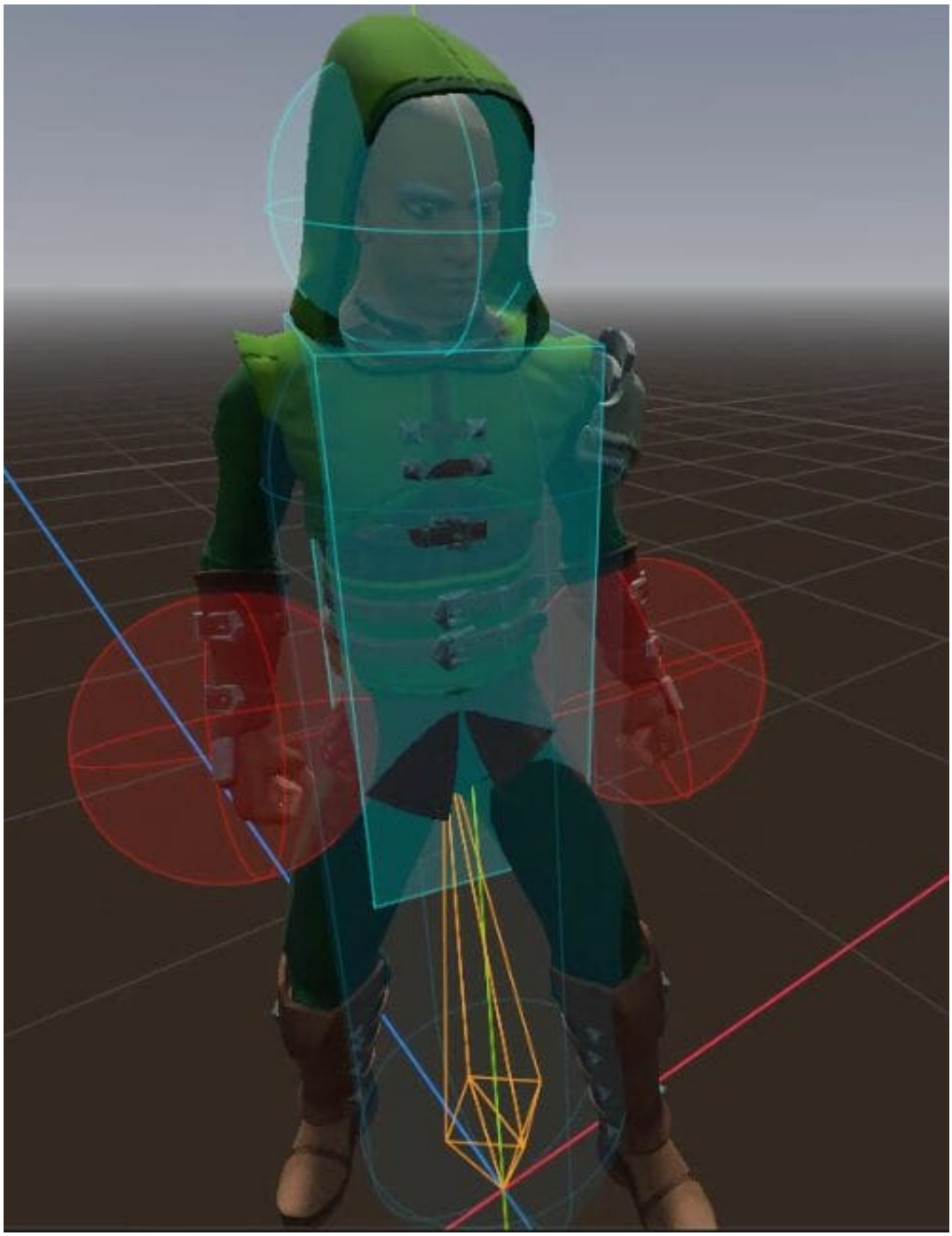
With the bone attachment in place, add an **Area3D** child node and name it **Head_HitReceiver**. Then add a **CollisionShape3D** as a child of the Area3D and set its shape to a **SphereShape3D**. To make the collision shape easier to see while working, change the **Debug Color** to a brighter color, such as cyan.



Scale the sphere down to fit around the character's head and adjust its position upward so it sits correctly.

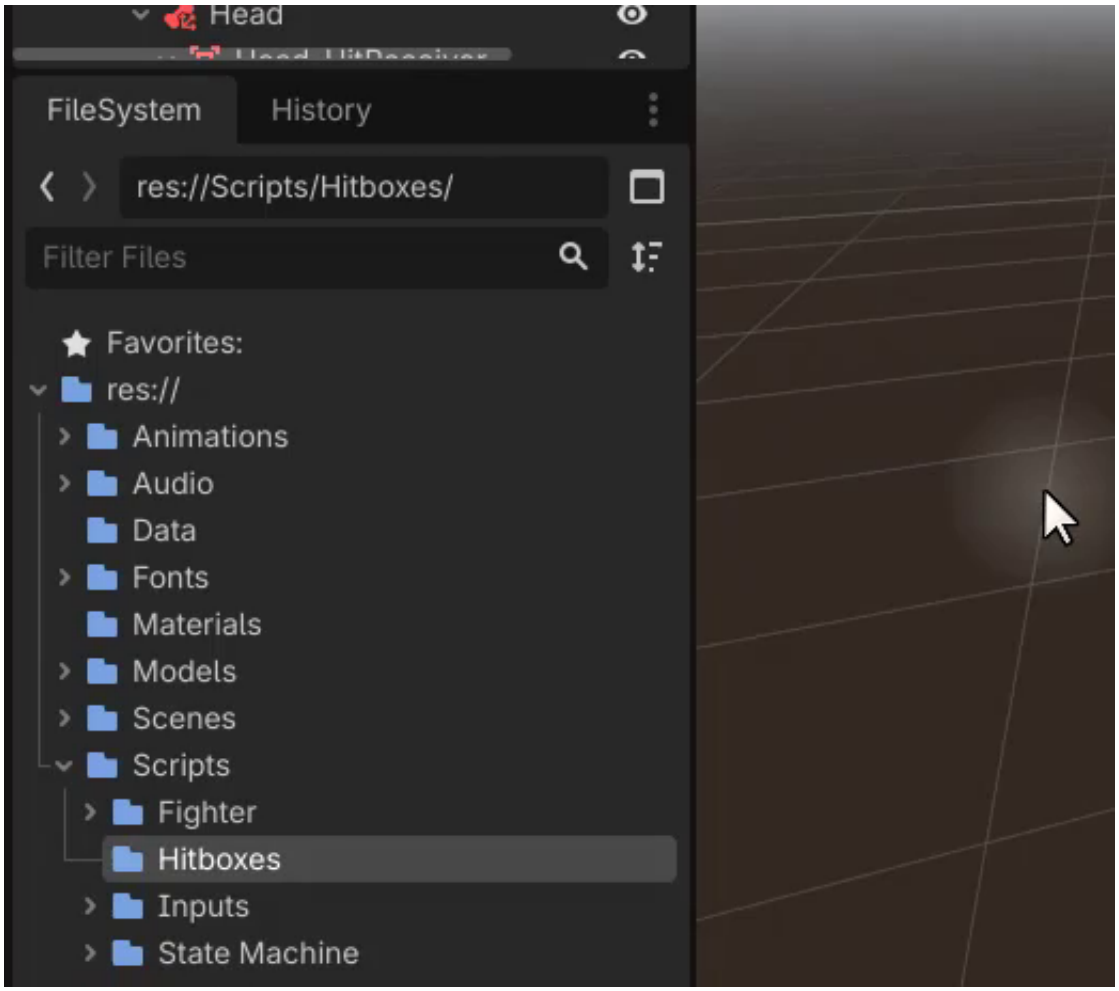
Hitbox Overview

At this point, the character has a complete set of hitboxes. The hit senders (shown as red spheres) are attached to both the left and right hands, and the hit receivers (shown as cyan shapes) cover the chest and head. These collision shapes can be fine-tuned later — for example, widening the chest receiver to detect more hits.



Creating the Hitbox Scripts

With the nodes in place, it is time to create the scripts that give them their functionality. Inside the **Scripts** folder in the FileSystem panel, create a new folder called **Hitboxes**. Within that folder, create two new GDScript files: hit_receiver.gd and hit_sender.gd.



The HitReceiver Script

The HitReceiver script is the simpler of the two. Since hit receivers are Area3D nodes, the script extends Area3D and is given the class name HitReceiver.

```
class_name HitReceiver
extends Area3D
```

The script requires two variables. The first is a reference to the Fighter that owns this hit receiver, which is needed in order to deal damage to the correct character. The second is an exported hit_state_name string, which determines which state the fighter transitions to when hit on this body part. This exported variable allows different hit receivers (head, chest) to trigger different hit reaction states in the future.

```
var fighter : Fighter
@export var hit_state_name : String
```

The script needs two functions. The initialize function receives a reference to the owning Fighter and stores it. Note the use of self.fighter to distinguish the class variable from the parameter, since both share the same name. Inside the function body, writing fighter alone refers to the parameter, while self.fighter refers to the class-level variable.


```
func initialize (fighter : Fighter):
 self.fighter = fighter
```

The hit function accepts a damage value as an integer. For now, the function body is left as a placeholder with pass, since the damage-dealing logic will be implemented in a later lesson.

```
func hit (damage : int):
 pass # deal damage to the fighter
```

The HitSender Script

The HitSender script is responsible for detecting whether its collision area is overlapping an opponent's hit receiver. Like the hit receiver, it extends Area3D and is given the class name HitSender.

```
class_name HitSender
extends Area3D
```

It also holds a reference to its owning Fighter, which is needed to access the fighter's player_id for distinguishing between the player's own hit receivers and the opponent's. The initialize function works the same way as in the hit receiver.

```
var fighter : Fighter

func initialize (fighter : Fighter):
 self.fighter = fighter
```

The key function in this script is detect_hit, which returns a HitReceiver if a valid hit is detected, or null otherwise. Since HitSender extends Area3D, it has access to the get_overlapping_areas() method, which returns an array of all Area3D nodes currently overlapping with this node.

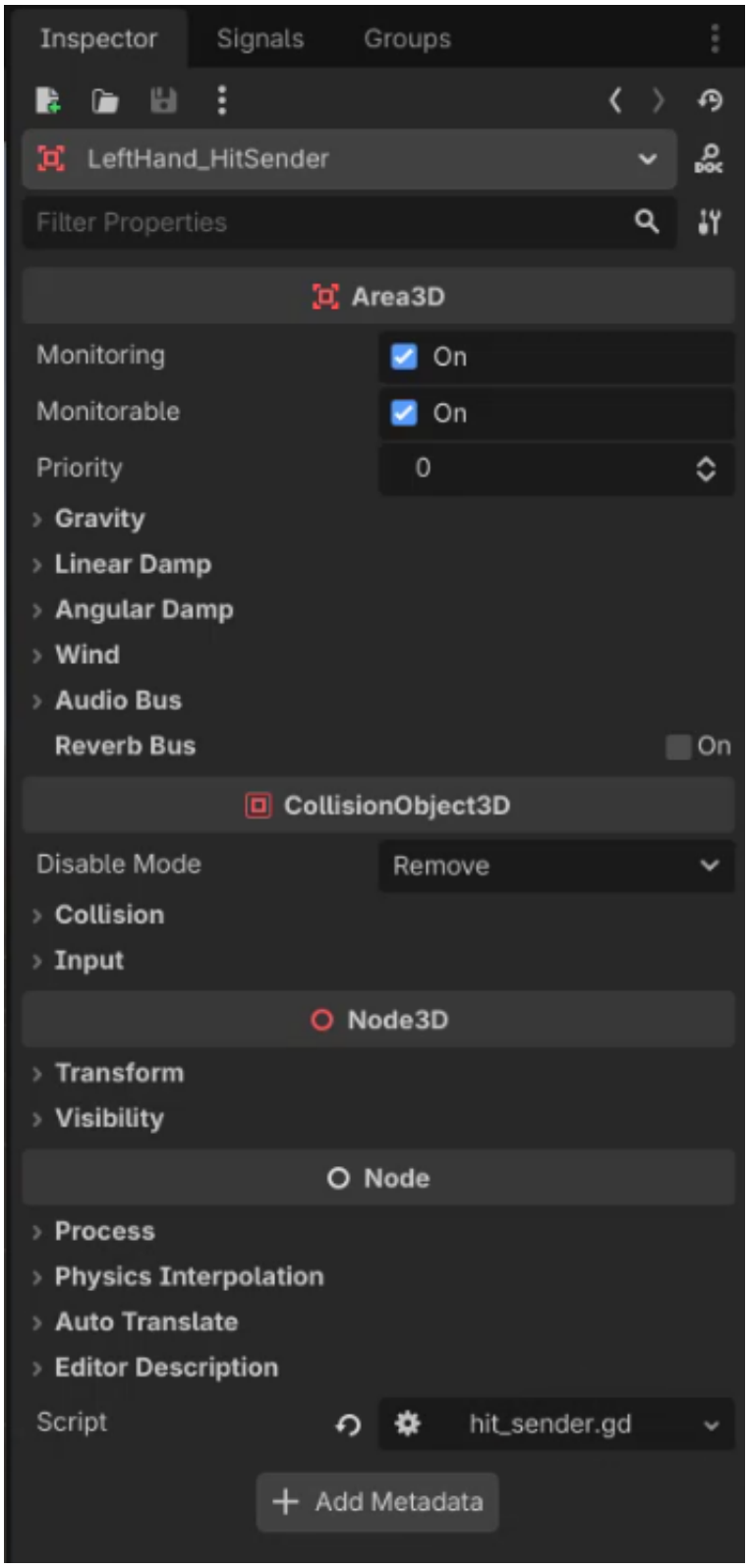
```
func detect_hit () -> HitReceiver:
 var areas : Array[Area3D] = get_overlapping_areas()

 for area in areas:
  if area is not HitReceiver:
   continue

  if area.fighter.player_id == fighter.player_id:
   continue

  return area

 return null
```


The function iterates through each overlapping area and applies two filters. First, it checks whether the area is an instance of HitReceiver — if not, it skips to the next iteration with continue. Second, it compares the player_id of the hit receiver's fighter against the hit sender's fighter. If they match, it means the hit sender is overlapping one of its own hit receivers, which should be ignored to prevent self-damage. If both checks pass, the area is a valid opponent hit receiver and is returned.

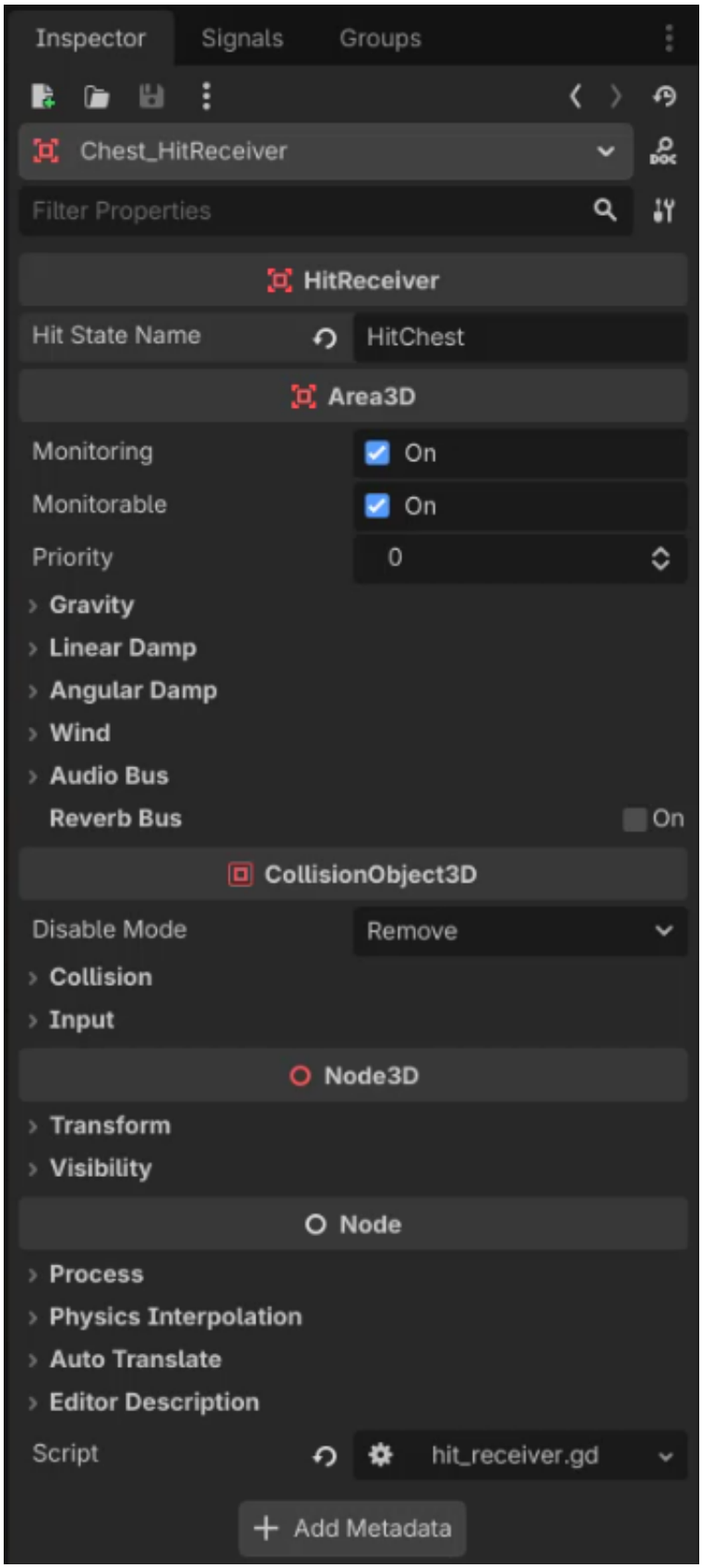
If no valid hit receiver is found after checking all overlapping areas, the function returns null.

Attaching Scripts to Nodes

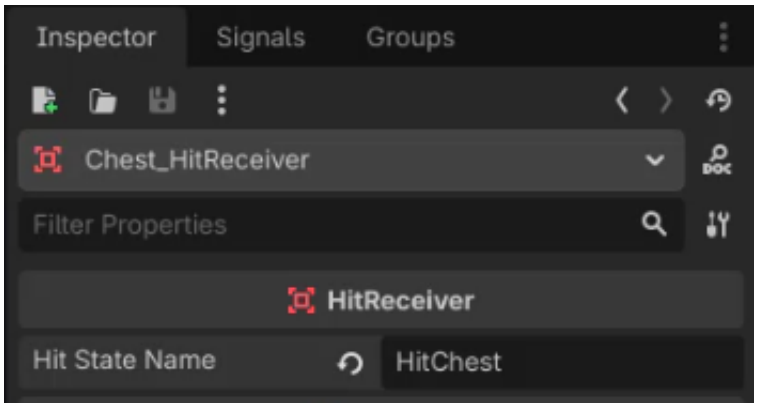
With both scripts written, the next step is to attach them to the appropriate nodes. Drag the hit_sender.gd script onto both the **LeftHand** and **RightHand** hit sender nodes.



Then drag the hit_receiver.gd script onto both the **Chest** and **Head** hit receiver nodes.



Finally, for both the head and chest hit receivers, set the **Hit State Name** property in the Inspector to HitChest. For now, both body parts use the same hit state, but this can be changed later once separate hit reaction states are created.



Summary

In this lesson, we completed the hitbox setup for the fighter character by adding a right hand hit sender and a head hit receiver. We then created the HitReceiver and HitSender scripts, which handle the core hit detection logic. The HitSender uses get_overlapping_areas() to find overlapping Area3D nodes and filters them to identify valid opponent hit receivers. In the next lesson, we will integrate these scripts with the attacking state to detect hits during attack animations.

In this lesson, we integrate the hit detection system into the AttackingState script so that attacks can actually register hits against an opponent. We add new variables for referencing hit senders and defining a time window for hit detection, implement the detection logic in the update() function, and initialize the hitbox components from the Fighter script. Finally, we configure the attack properties in the Inspector and test the system in-game.

Adding Hit Detection Variables to AttackingState

The hitbox system created in the previous lessons included HitSender and HitReceiver nodes, but they are not yet connected to the attacking state. To wire them up, we need several new variables in attacking_state.gd. First, an exported reference to the HitSender that this attack uses, which represents the fist or limb performing the attack. Then a private boolean has_hit to ensure only one hit registers per attack. Finally, two exported float values define the time window during which the attack can register hits.

```
@export var damage : int

@export var hit_sender : HitSender
var has_hit : bool = false
@export var hit_detect_start_time : float
@export var hit_detect_end_time : float
```

The hit_detect_start_time and hit_detect_end_time values refer to the local_time of the state, which tracks how long the state has been active. This allows each attack to have a precise window during its animation where hits count.

Resetting Hit Detection on Enter

Each time the fighter enters the attacking state, the has_hit flag needs to be reset so the attack can register a new hit. This is added to the existing enter() function:

```
func enter ():
 super.enter()

 animation.set_animation(animation_name)
 fighter.add_force(forward_force * fighter.forward_direction)
 has_hit = false
```

Implementing Hit Detection in Update

The core hit detection logic goes inside the update() function. The code checks whether the current local_time falls within the detection window defined by hit_detect_start_time and hit_detect_end_time. If it does, the HitSender attempts to detect a hit by checking for overlapping HitReceiver areas belonging to the opponent.

When a valid hit is detected and has_hit is still false, it sets has_hit to true and calls the hit() function on the HitReceiver, passing the damage amount.

```
func update (delta : float):
  super.update(delta) # Only detect hits during the specified time window
```



```
  if local_time >= hit_detect_start_time and local_time <= hit_detect_end_time:
    var hit : HitReceiver = hit_sender.detect_hit()

    if hit and has_hit == false:
      has_hit = true # Mark that a hit has occurred
      hit.hit(damage) # Return to Standing after duration has elapsed

  if local_time >= duration:
    state_machine.change_state("Standing")
```

The damage variable is declared as an int, which is important because the hit() function on HitReceiver expects an integer parameter.

```
@export var damage : int
```

Adding a Temporary Debug Print to HitReceiver

Before building out the full damage system, a temporary print() statement is added to the hit() function in hit_receiver.gd so we can verify that hits are being detected correctly:

```
func hit (damage : int):
 print("Has been hit")
```

This provides a quick way to confirm the system works by checking the Output panel during gameplay.

Initializing Hit Components in the Fighter Script

Both the HitSender and HitReceiver nodes require a reference to their parent Fighter in order to function properly. The initialize() function on each component stores this reference. To call it for every hitbox component, a _ready() function is added to fighter.gd.

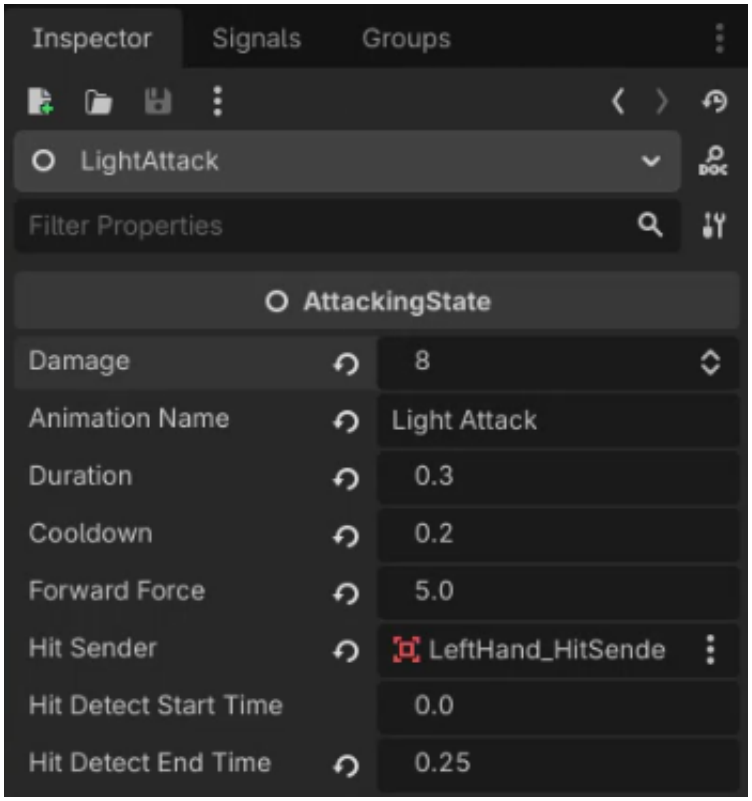
The find_children() function searches all descendant nodes matching a given type. The first argument is the name pattern ("*" for all), the second is the type to filter by, the third enables recursive searching through children of children, and the fourth controls whether to search only owned nodes (set to false to find all child nodes).

```
func _ready ():
 for child in find_children("*", "HitReceiver", true, false):
  child.initialize(self)

 for child in find_children("*", "HitSender", true, false):
  child.initialize(self)
```

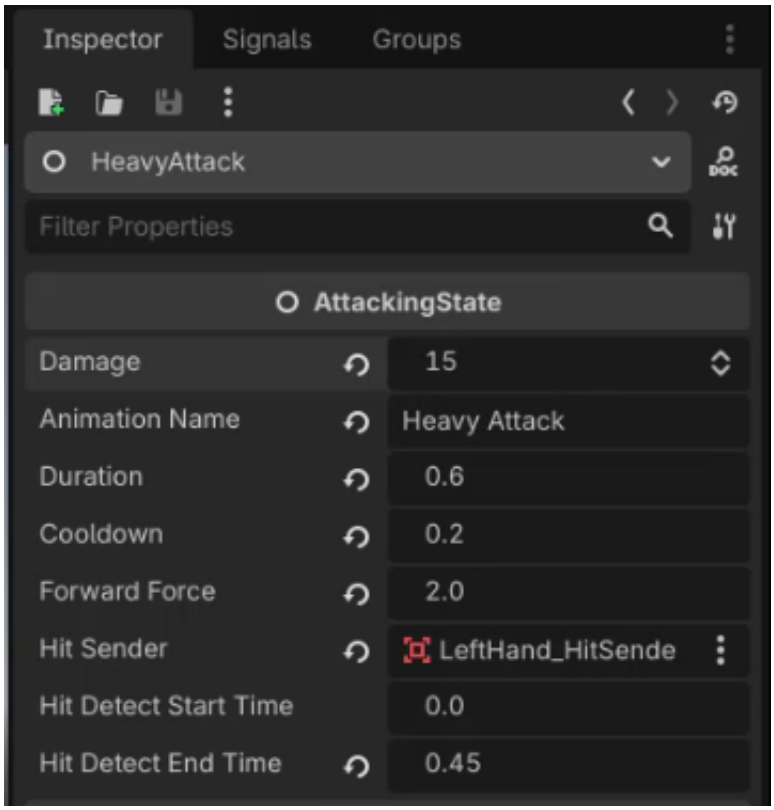
Configuring Attack Properties in the Inspector

With the code in place, the attack state nodes need their exported properties configured in the Godot Inspector. For the LightAttack state, the Hit Sender is set to the right hand’s HitSender node

by dragging it from the scene tree. The hit detection window is set with a start time of 0.0 and an end time of 0.25 seconds, and the damage is set to 8.



The HeavyAttack state is configured similarly, also using the right hand HitSender. The hit detection window is wider with a start time of 0.0 and an end time of 0.45 seconds, reflecting the longer animation. The damage is set to 15.



Testing Hit Detection

Running the game, swinging at empty space produces no output, which is expected since there is no opponent HitReceiver in range. Moving close to the opponent and attacking causes “Has been hit” to appear in the Output panel, confirming that the HitSender is correctly detecting overlapping HitReceiver areas and triggering the hit() function. Both light and heavy attacks register hits successfully.

```
New State: LightAttack
Has been hit
New State: Standing
New State: LightAttack
Has been hit
New State: Standing
```

At this stage, the hit detection is functional, but there is no visual feedback beyond the console output. In the next lesson, the system will be expanded so that when a fighter takes damage, they transition to a hit state that stuns them briefly and plays a reaction animation.

In this lesson, we will implement a hit state for our fighter characters. When a fighter is struck by an opponent's attack, they will transition into a temporary stun state that plays a hit reaction animation and prevents them from performing any actions for a short duration. We will also wire up the damage pipeline so that hit receivers communicate with the fighter script, and add a knockback force to push fighters backward when they take damage.

Creating the HitState Script

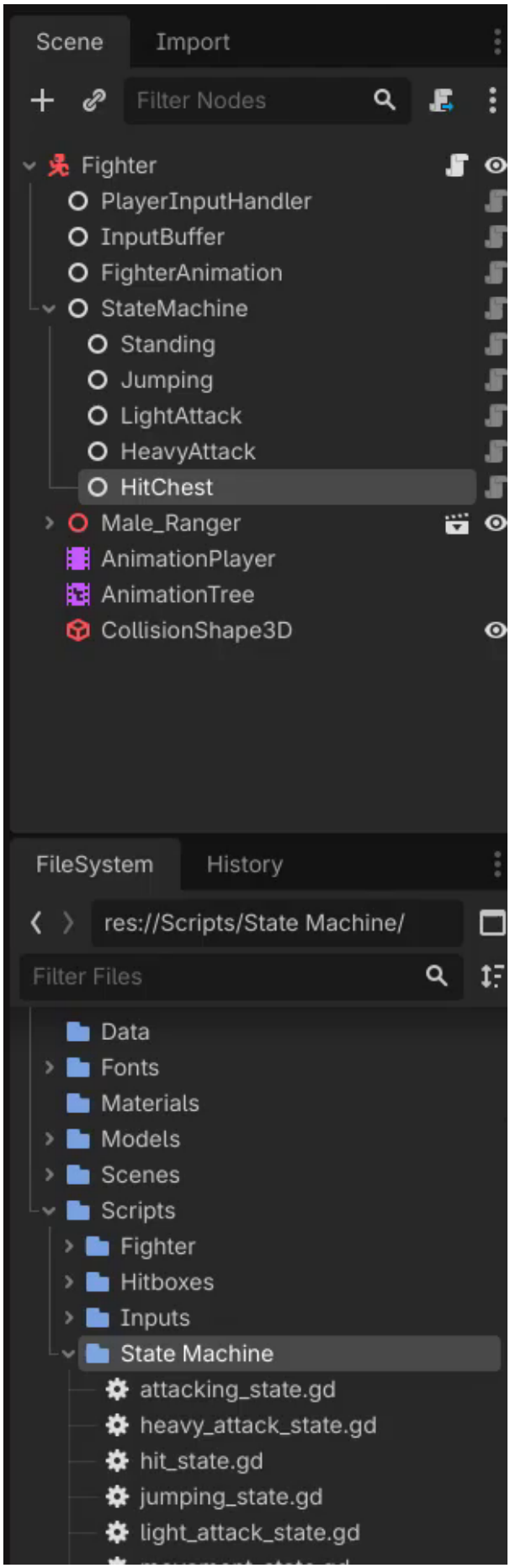
Inside the **Scripts/State Machine** folder, create a new script called hit_state.gd. This script extends the base State class and defines two exported variables: duration controls how long the hit stun lasts, and animation_name specifies which animation to play when the fighter enters this state.

```
class_name HitState
extends State

@export var duration : float
@export var animation_name : String

func enter ():
 super.enter()
 animation.set_animation(animation_name)

func update (delta : float):
 super.update(delta)

 if local_time >= duration:
  state_machine.change_state("Standing")
```

The enter() function calls super.enter() to run the base state's initialization logic, then sets the animation to the specified hit reaction animation. The update() function tracks elapsed time using local_time (inherited from the base State class) and transitions back to the Standing state once the stun duration has elapsed.

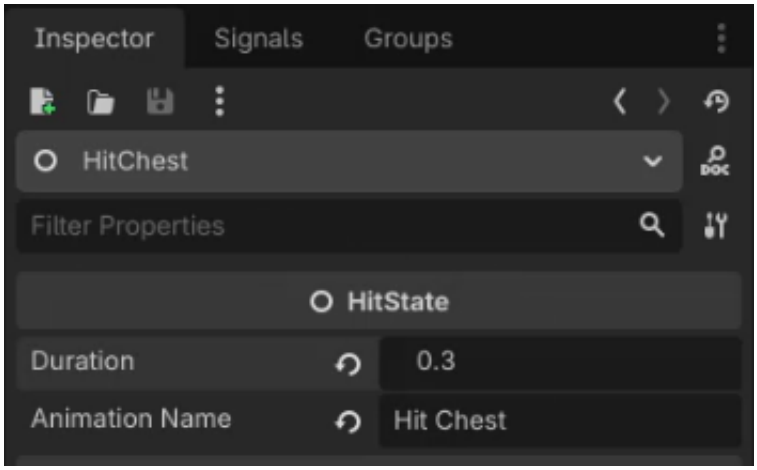
Adding the HitChest Node to the State Machine

With the script created, add a new child node to the **StateMachine** node in the Fighter scene. Attach the hit_state.gd script to it and rename the node to **HitChest**. This name must match the hit_state_name property set on the hit receivers, since the state machine uses the node name to look up which state to transition to.



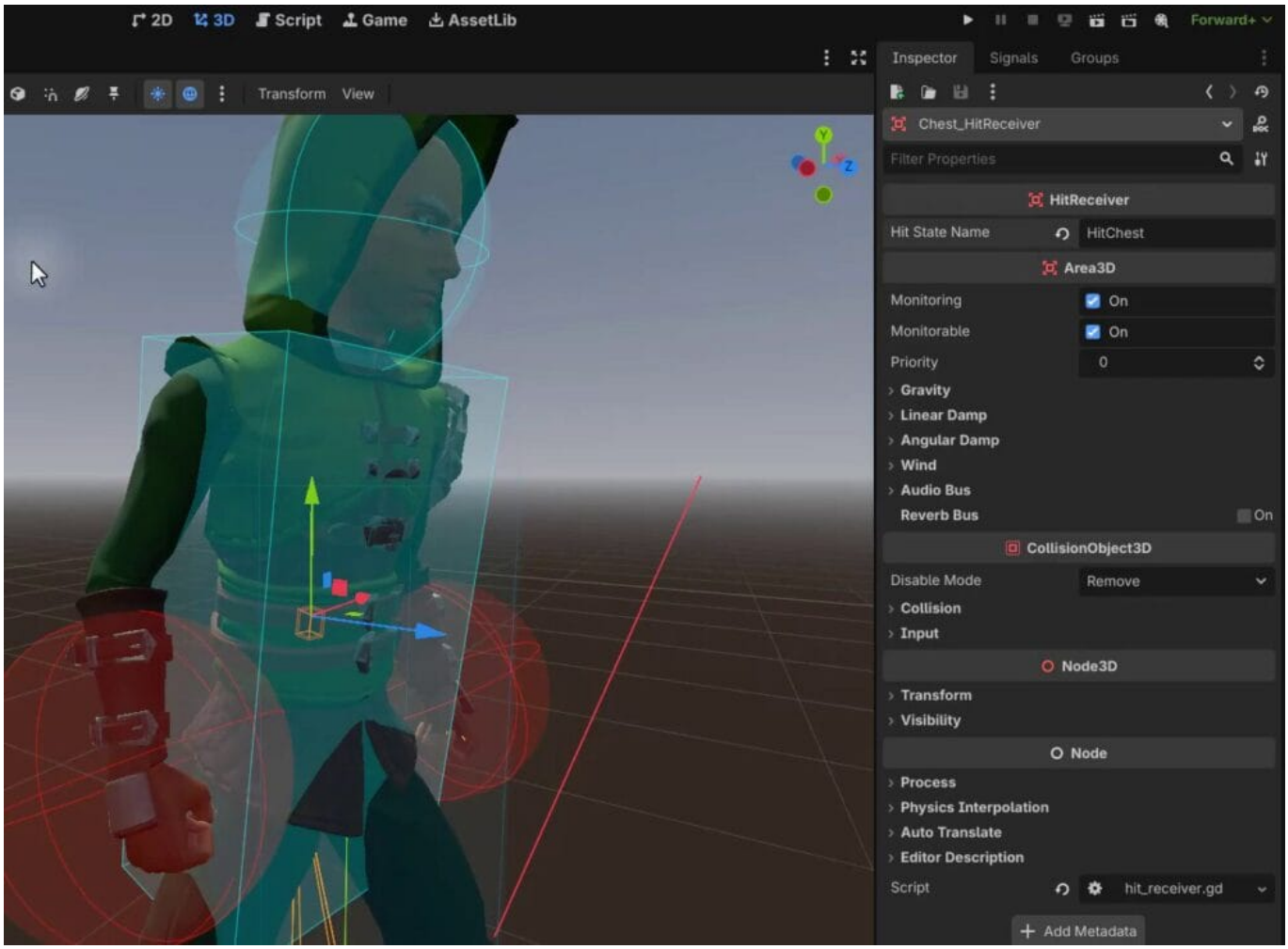
In the Inspector, configure the following properties on the HitChest node:

- **Animation Name**: Set to the name of the hit chest animation in the AnimationTree (e.g. “Hit Chest”)
- **Duration**: Set to 0.3 seconds, meaning the fighter will be unable to move or attack for that period



Verifying Hit Receiver Configuration

Before testing, select each of the hit receiver nodes on the fighter (such as **Chest_HitReceiver**) and confirm that the **Hit State Name** property in the Inspector matches the exact node name of the state that should be triggered. In this case, both hit receivers should have their Hit State Name set to HitChest.



Implementing the take_damage Function

At this point, saving and running the game reveals that hitting the opponent does not trigger the hit state. The hit receivers detect collisions, but there is no function on the fighter to process the damage and perform the state transition. To fix this, open fighter.gd and add a take_damage function at the bottom of the script.

```
func take_damage (hit_receiver : HitReceiver, damage_amount : float):
 state_machine.change_state(hit_receiver.hit_state_name)
```

This function receives a reference to the HitReceiver that was hit along with the damage amount. It uses the receiver's hit_state_name property to determine which state to transition to, allowing different hit receivers to trigger different hit reaction states if needed.

Next, update the hit() function in hit_receiver.gd so that it calls take_damage on the fighter instead of simply printing a debug message. Pass self as the hit receiver reference and forward the damage value:

```
func hit (damage : int):
 fighter.take_damage(self, damage)
```


Testing Hit Reactions

After saving and pressing play, attacking the opponent now triggers the hit chest animation. The struck fighter enters the HitChest state for 0.3 seconds, during which they cannot move or attack. Once the duration elapses, they automatically transition back to the Standing state and regain control.

Adding Knockback Force

To make hits feel more impactful, we can apply a backward force to the fighter when they take damage. In the take_damage function in fighter.gd, add a call to add_force() after the state change. Multiplying forward_direction by -10 pushes the fighter in the opposite direction they are facing, creating a knockback effect:

```
func take_damage (hit_receiver : HitReceiver, damage_amount : float):
 state_machine.change_state(hit_receiver.hit_state_name)
 add_force(forward_direction * -10)
```

Running the game again confirms the knockback is working. Each time a fighter is hit, they are pushed backward slightly while playing the hit reaction animation, giving clear visual feedback that the attack connected.

In the next lesson, we'll create a defeated state so that damaging characters is more meaningful.

In this lesson, we implement the ability to defeat a fighter in our Godot fighting game. This involves adding health tracking variables to the fighter, modifying the damage function to check for defeat, creating a new defeat() function, and building a DefeatedState script that locks the character out of further actions.

Adding Health Variables

Open the fighter.gd script and add three new variables to track health and defeat status. The current_health and max_health exported variables allow you to configure health values in the editor, while is_defeated is an internal flag that prevents a defeated fighter from taking further damage.

```
@export var current_health : float = 100.0
@export var max_health : float = 100.0

var is_defeated : bool = false
```

Modifying the take_damage Function

With health variables in place, the take_damage function needs to be updated to handle the full damage and defeat flow. The function first checks whether the fighter is already defeated and returns early if so, preventing any additional damage processing. It then subtracts the incoming damage from current_health and checks whether health has dropped to zero or below. If the fighter's health is depleted, current_health is clamped to zero to avoid negative values, and the defeat() function is called. Otherwise, the fighter transitions to the hit reaction state and receives a knockback force.

```
func take_damage (hit_receiver : HitReceiver, damage_amount : float):
 if is_defeated:
  return

 current_health -= damage_amount

 if current_health <= 0:
  current_health = 0
  defeat()
 else:
  state_machine.change_state(hit_receiver.hit_state_name)
  add_force(forward_direction * -10)
```

Creating the defeat Function

The defeat() function handles the transition into the defeated state. It sets the is_defeated flag to true so the fighter can no longer take damage, then tells the state machine to switch to the Defeated state.

```
func defeat ():
 is_defeated = true
 state_machine.change_state("Defeated")
```

Creating the DefeatedState Script

Inside the **Scripts/State Machine** folder, create a new script called defeated_state.gd. This is the simplest state in the project. It extends the base State class and overrides the enter() method to play the Defeated animation. Because no update() or transition logic is defined, the fighter remains locked in this state permanently once it is entered.

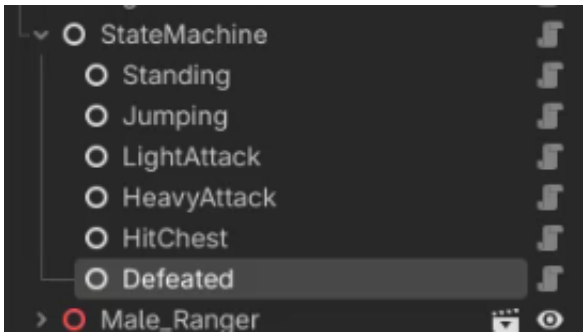
```
class_name DefeatedState
extends State

func enter ():
 super.enter()
 animation.set_animation("Defeated")
```

Make sure the string "Defeated" matches the exact name of the defeated animation state in your AnimationTree.

Adding the Defeated State to the Scene Tree

With the script created, add it to the fighter's state machine:

1. In the Scene tree, right-click on the **StateMachine** node.
2. Select **Add Child Node** and choose **DefeatedState**.
3. Rename the new node to **Defeated** so it matches the state name used in state_machine.change_state("Defeated").



Testing the Defeated State

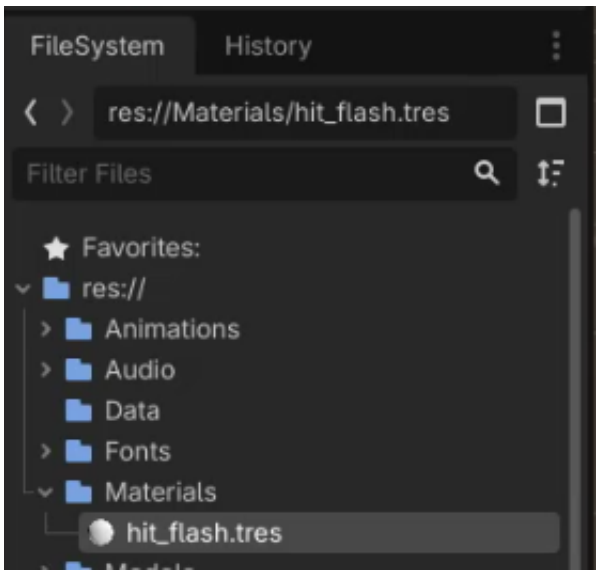
Run the game and walk one fighter up to the opponent. Attack them repeatedly and observe the health values decreasing in the Output panel. Each hit reduces current_health by the attack's damage amount. Once health reaches zero, the fighter enters the defeated state, plays the defeated animation, and can no longer be controlled. This is the point where you would trigger an end-game event in a full implementation.

In the next lesson, we'll take a break from states and focus on improving the feedback from our damage system.

In this lesson, we will implement a hit flash effect for our fighter characters. When a fighter takes damage, all of their mesh instances will briefly flash white to provide clear visual feedback. To accomplish this, we will create a new material, write a FighterVisual script, and set up a global event bus so any part of the game can react to damage and defeat events.

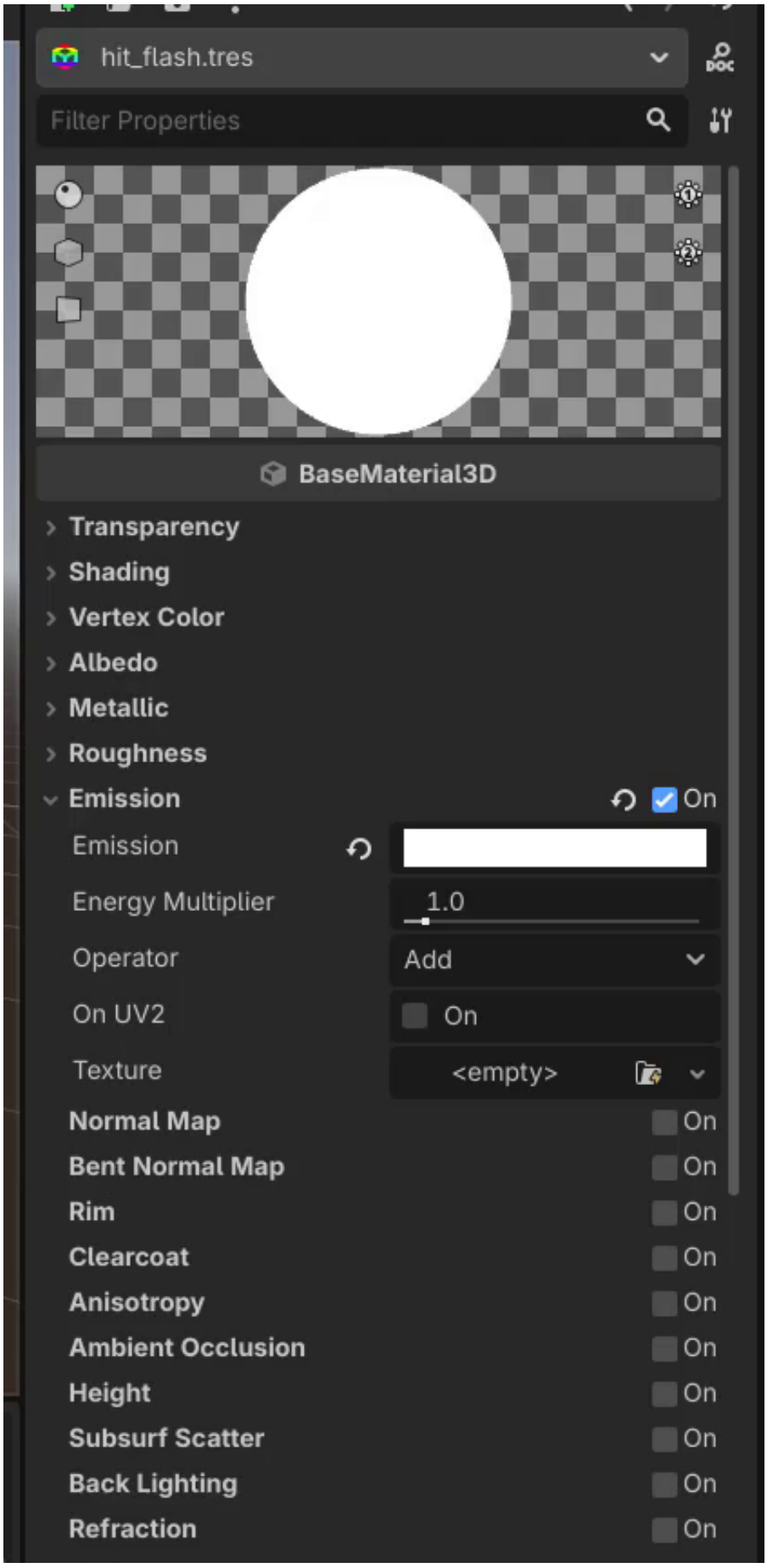
Creating the Hit Flash Material

The hit flash effect works by temporarily applying a bright white material overlay to every mesh on the fighter model. To set this up, right-click in the **Materials** folder in the FileSystem dock, select **Create New > Resource**, search for **StandardMaterial3D**, and save it as hit_flash.tres.

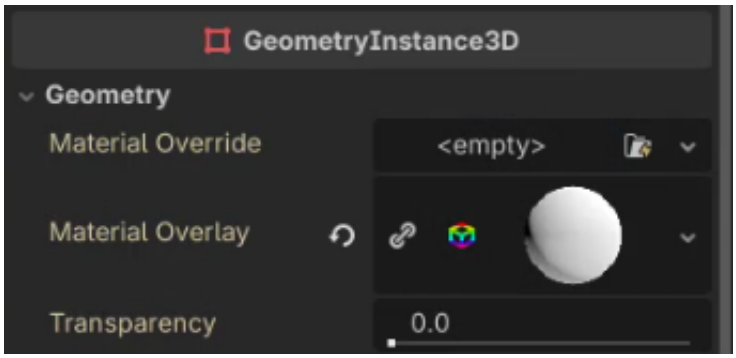


With the material selected, configure it in the Inspector as follows:

- Under **Albedo**, set the color to white.
- Enable **Emission** and set the emission color to white as well. This gives the material a slight glow so the flash is clearly visible regardless of the scene lighting.
- The **Energy Multiplier** can be left at 1.0, though you can increase it if you want a stronger glow.



This material will be applied using the **Material Overlay** property on each MeshInstance3D. The overlay renders on top of the existing material, so there is no need to cache or restore the original material — simply set the overlay to the hit flash material and later set it back to null.



Creating the FighterVisual Script

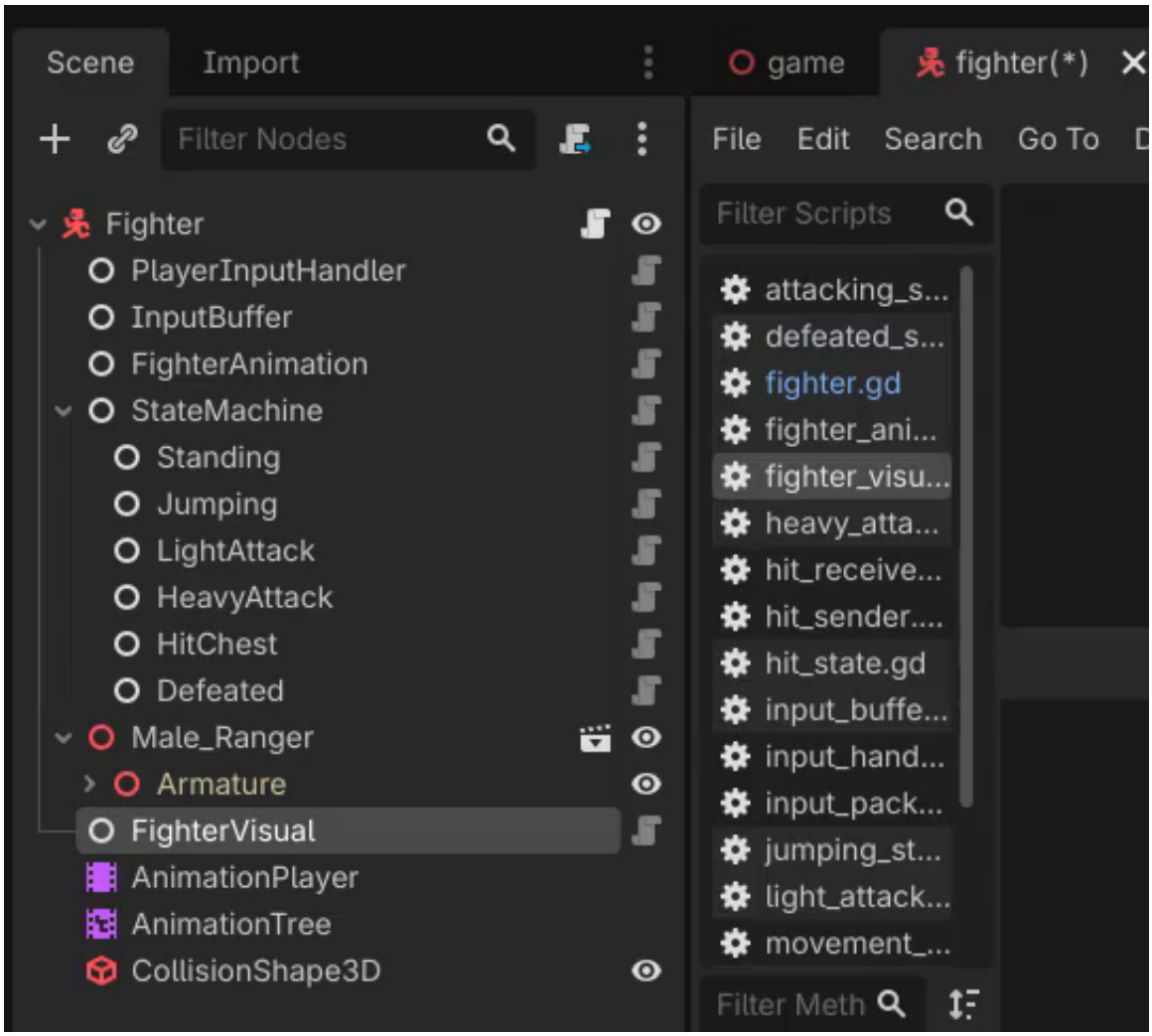
Create a new script in the Scripts/Fighter folder called fighter_visual.gd. This script will manage the visual feedback for a single fighter. Start by defining the class and its variables:

```
class_name FighterVisual
extends Node

var models : Array[MeshInstance3D]

@export var hit_material : StandardMaterial3D
@export var model_root : Node3D

@onready var fighter : Fighter = $".."
```

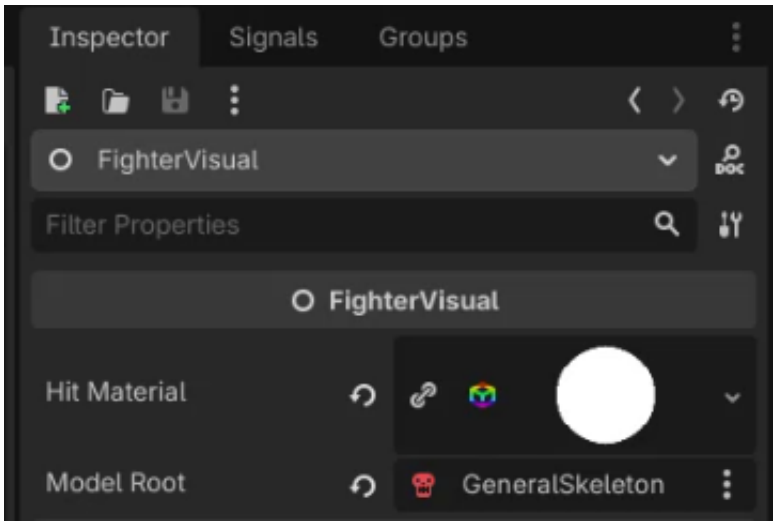
- models — an array that will hold every MeshInstance3D that makes up the fighter character. This is populated at runtime.
- hit_material — the StandardMaterial3D resource (hit_flash.tres) that will be applied as an overlay when the fighter is hit.
- model_root — the root node whose children will be searched for mesh instances. This will be set to the skeleton node in the Inspector.
- fighter — an @onready reference to the parent Fighter node, used to determine whether this particular fighter was the one damaged.

Adding the FighterVisual Node

In the Fighter scene, right-click the **Fighter** node and add a new child node of type **Node**. Rename it to **FighterVisual** and attach the fighter_visual.gd script.



In the Inspector, assign the **Hit Material** property to the hit_flash.tres resource and set **Model Root** to the GeneralSkeleton node, which contains all of the fighter's mesh instances.

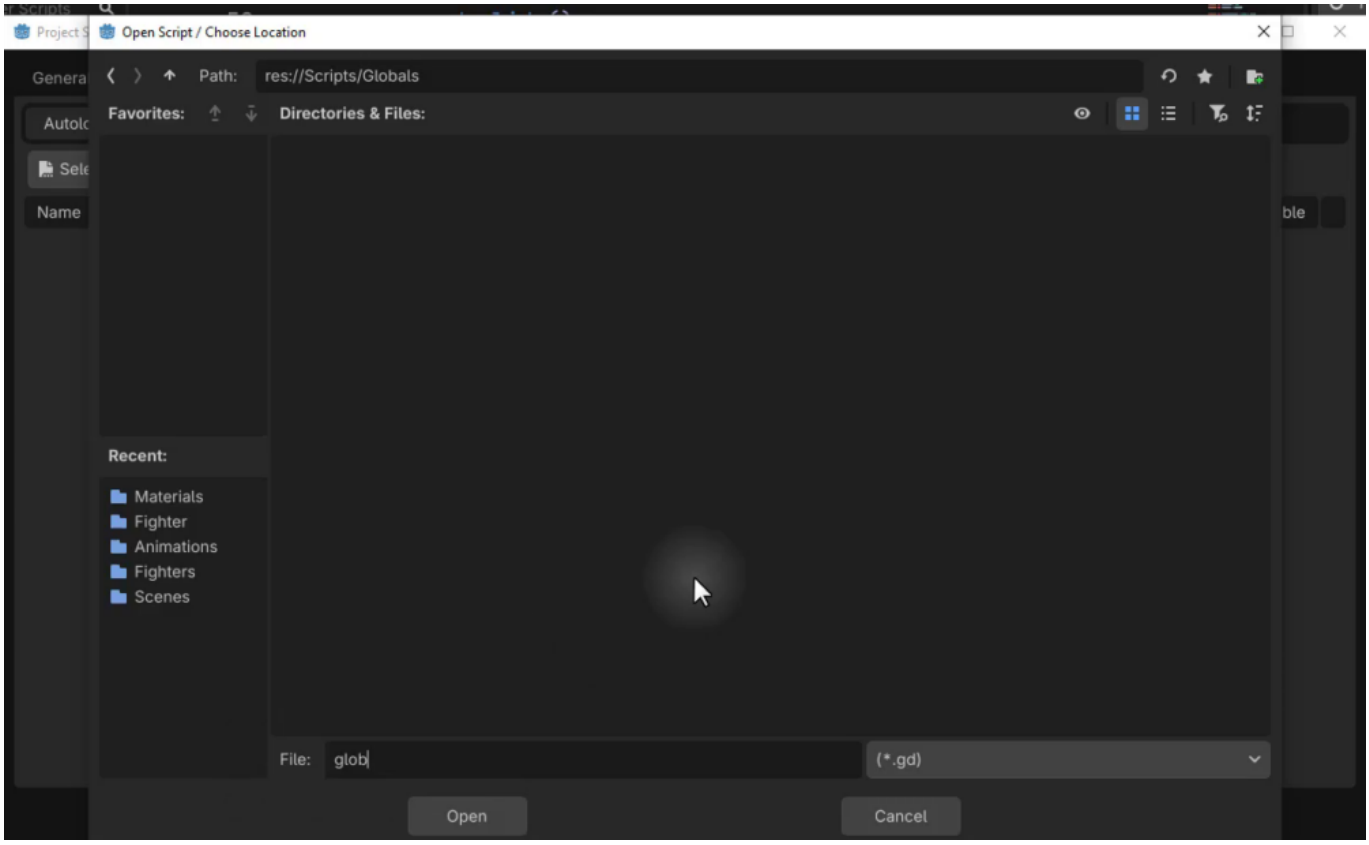


Creating the Global Events Singleton

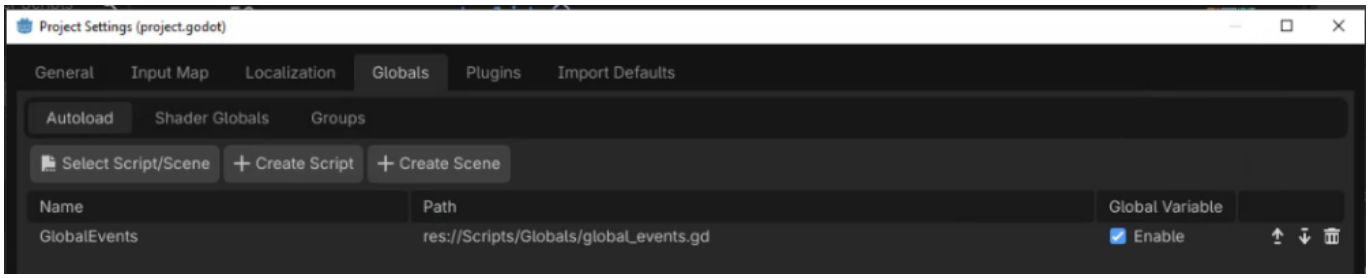
To trigger the hit flash from the fighter script without needing a direct reference to the FighterVisual node, we use a global event bus. This is a script registered as an AutoLoad (singleton) that contains

signals any node can emit or connect to.

Create a new folder called **Globals** inside the **Scripts** folder, then create a new script called global_events.gd inside it.



Open **Project > Project Settings > Globals** (the AutoLoad tab) and add this script so it is available globally.



Inside global_events.gd, define the two signals:

```
extends Node

signal FighterDamaged (fighter : Fighter)
signal FighterDefeated (fighter : Fighter)
```

The FighterDamaged signal is emitted whenever a fighter takes damage, passing the affected fighter as a parameter. The FighterDefeated signal works the same way but fires when a fighter’s health reaches zero. Because GlobalEvents is an AutoLoad, any script can access it directly by name

without needing a node reference.

Populating the Models Array

In the _ready function, search through all children of the model_root and add every MeshInstance3D to the models array:

```
func _ready ():
 for child in model_root.find_children("*", "MeshInstance3D", true):
  models.append(child)

 GlobalEvents.FighterDamaged.connect(_hit_flash)
```

The find_children method searches recursively for all nodes of type MeshInstance3D under the skeleton, which covers every part of the fighter model. The last line connects the global FighterDamaged signal to the _hit_flash function.

Implementing the Hit Flash Function

The _hit_flash function receives the damaged fighter as a parameter from the signal. It first checks whether the damaged fighter matches this node's fighter — if not, it returns early. Otherwise, it applies the hit material overlay to every mesh, waits briefly, and then removes it:

```
func _hit_flash (fighter : Fighter):
 if self.fighter != fighter:
  return

 for model : MeshInstance3D in models:
  model.material_overlay = hit_material

 await get_tree().create_timer(0.05).timeout

 for model : MeshInstance3D in models:
  model.material_overlay = null
```

The await keyword pauses the function until the timer's timeout signal fires. During that 0.05-second window, the white overlay is visible on the fighter, creating the flash effect. After the timer elapses, the overlay is set back to null, restoring the fighter's normal appearance.

Emitting Signals from the Fighter Script

Open fighter.gd and update the take_damage function to emit the FighterDamaged signal after reducing the fighter's health:

```
func take_damage (hit_receiver : HitReceiver, damage_amount : float):
 if is_defeated:
  return

 current_health -= damage_amount
 GlobalEvents.FighterDamaged.emit(self)
```



```
if current_health <= 0:
 current_health = 0
 defeat()
else:
 state_machine.change_state(hit_receiver.hit_state_name)
 add_force(forward_direction * -10)
```

The GlobalEvents.FighterDamaged.emit(self) call broadcasts the signal to all connected listeners, passing the current fighter instance. This is what triggers the _hit_flash function in FighterVisual.

Similarly, update the defeat function to emit the FighterDefeated signal:

```
func defeat ():
 is_defeated = true
 state_machine.change_state("Defeated")
 GlobalEvents.FighterDefeated.emit(self)
```

Testing the Hit Flash

With all the pieces in place — the hit flash material, the FighterVisual script connected to the FighterDamaged signal, and the fighter emitting that signal on damage — pressing play and attacking the opponent will cause the damaged fighter to flash white for a brief moment before returning to their normal appearance.

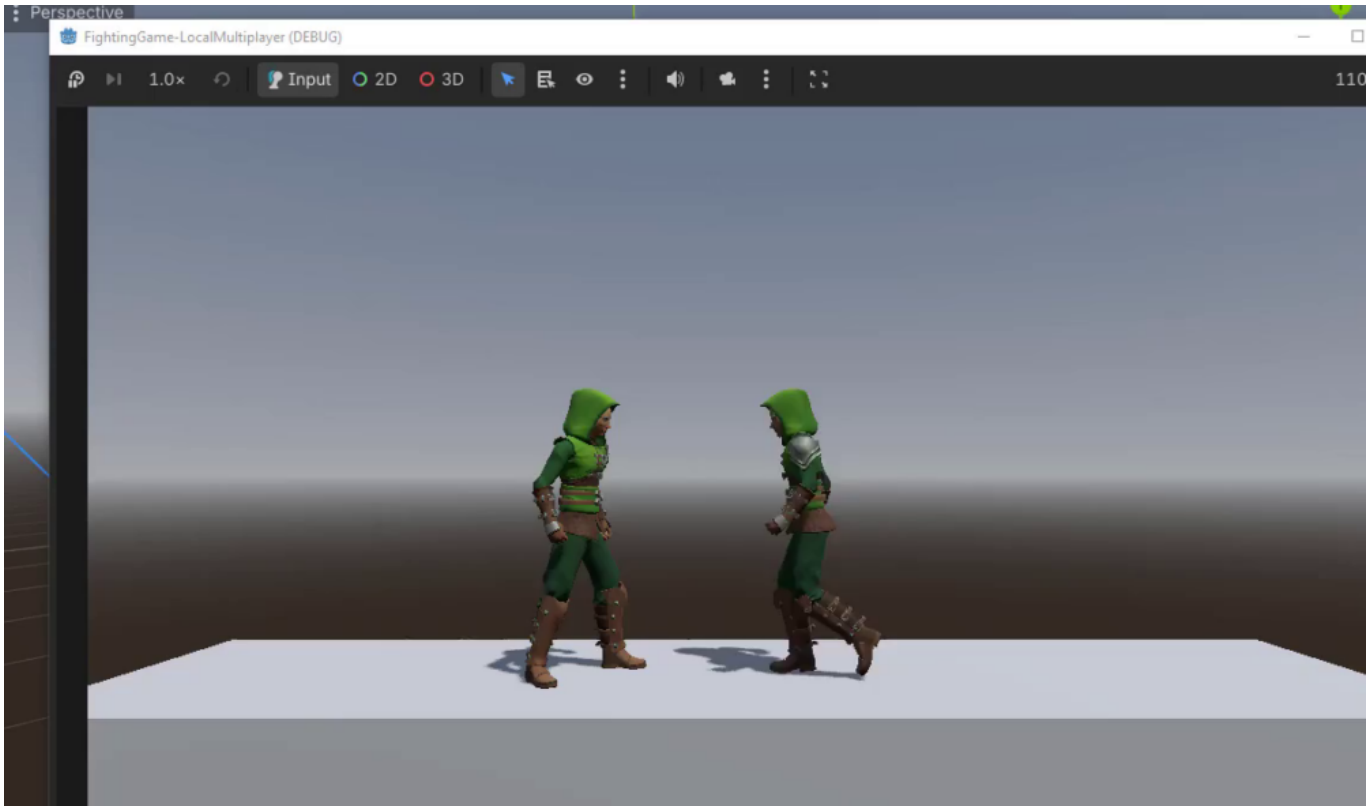
The complete fighter_visual.gd script for this lesson is as follows:

```
class_name FighterVisual
extends Node

var models : Array[MeshInstance3D]
```



```
@export var hit_material : StandardMaterial3D
@export var model_root : Node3D

@onready var fighter : Fighter = $".."

func _ready ():
 for child in model_root.find_children("*", "MeshInstance3D", true):
  models.append(child)

 GlobalEvents.FighterDamaged.connect(_hit_flash)

func _hit_flash (fighter : Fighter):
 if self.fighter != fighter:
  return

 for model : MeshInstance3D in models:
  model.material_overlay = hit_material

 await get_tree().create_timer(0.05).timeout

 for model : MeshInstance3D in models:
  model.material_overlay = null
```

In this lesson, we will implement an outfit system that assigns different colors to each fighter based on their player ID. Currently, both fighters share the same default appearance, making it difficult to tell them apart — especially when they cross paths and swap sides. By the end of this lesson, each fighter will display a distinct outfit color, and we will also fix the starting state so that fighters begin the game standing rather than jumping.

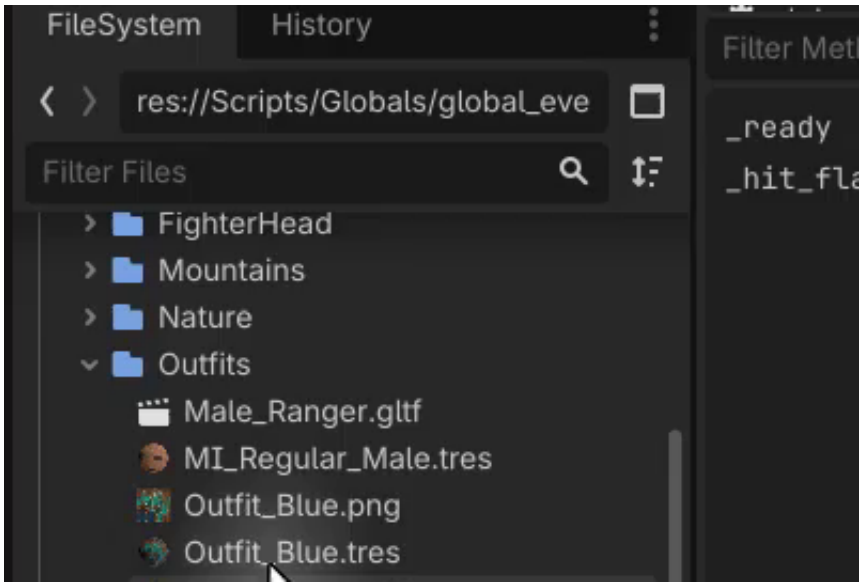


Adding the Outfit Materials Variable

The first step is to open the fighter_visual.gd script, which was created in the previous lesson to handle the hit flash effect. At the top of the script, below the existing variables, we need to add a new exported variable called outfit_materials. This will be an array of StandardMaterial3D resources representing the different outfit colors available to fighters.

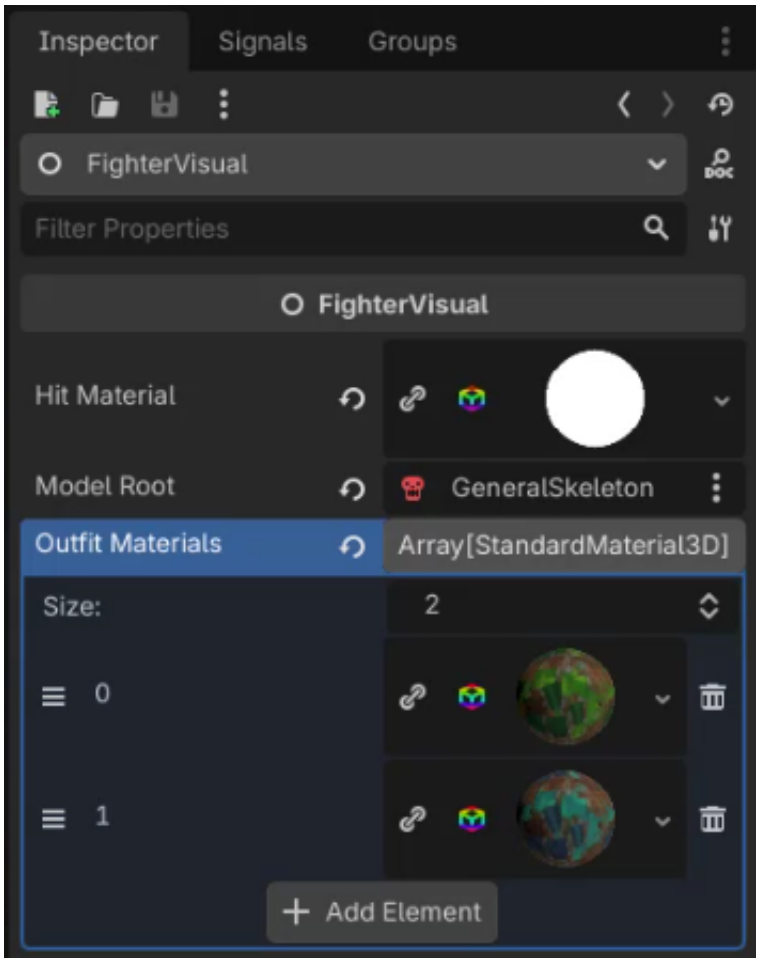
```
@export var outfit_materials : Array[StandardMaterial3D]
```

The project already includes three outfit material resources located in the **Models > Outfits** folder within the FileSystem dock: Outfit_Blue.tres, Outfit_Green.tres, and Outfit_Red.tres. Each of these materials defines a different color that can be applied to the fighter model.



Assigning Outfit Materials in the Inspector

With the variable declared, select the **FighterVisual** node in the Fighter scene. In the Inspector, open the **Outfit Materials** array and set its size to **2**, since there are two players in the game. For the first element (index 0, representing Player 1), drag in Outfit_Green. For the second element (index 1, representing Player 2), drag in Outfit_Blue.



Adding the Outfit Models Variable

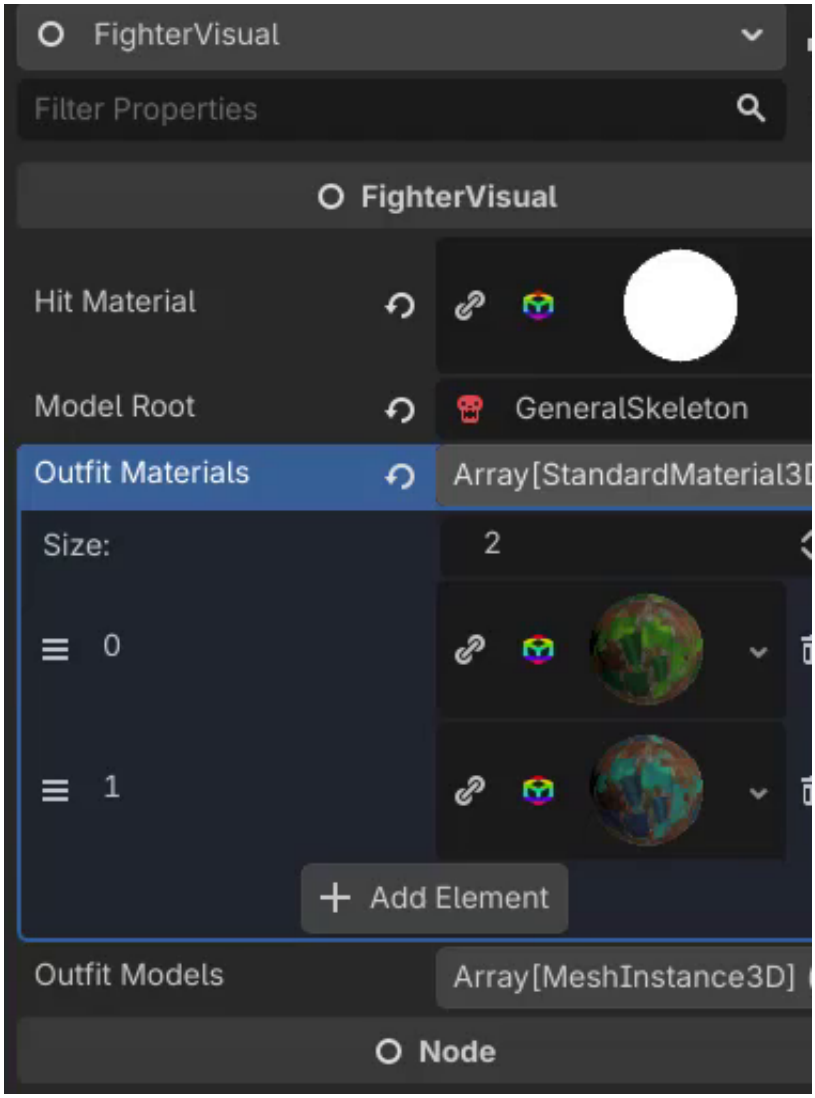
We cannot simply apply the outfit material to every mesh that makes up the fighter character. Doing so would change the color of parts like the face, head, and eyebrows, which should retain their original materials. Instead, we need a separate array that contains only the specific mesh instances that represent the outfit (clothing, armor, accessories).

Back in the fighter_visual.gd script, add another exported variable below outfit_materials:

```
@export var outfit_models : Array[MeshInstance3D]
```

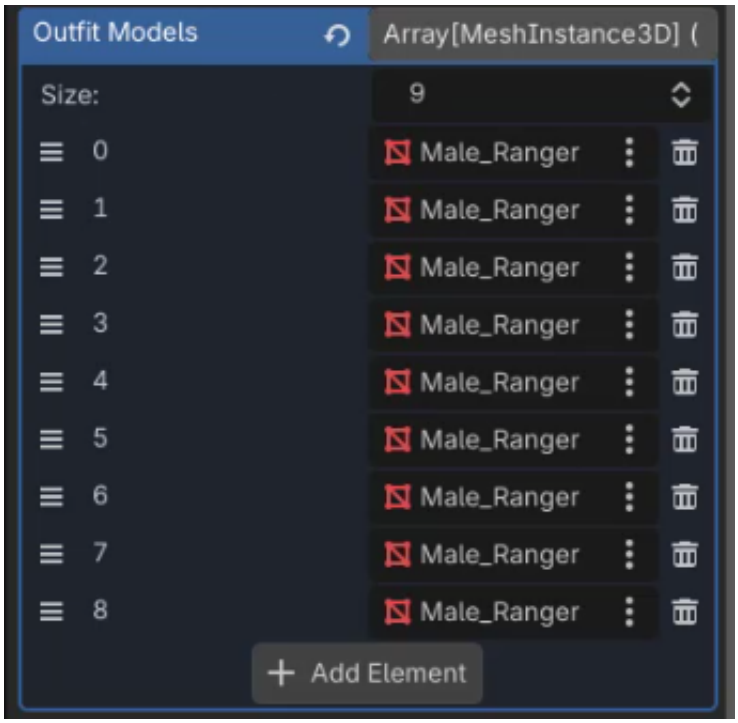
Assigning Outfit Models in the Inspector

Return to the Fighter scene and select the **FighterVisual** node. The new **Outfit Models** array property is now visible in the Inspector. The reason this array is populated manually through the editor rather than programmatically is that it would be difficult to automatically determine which meshes belong to the outfit versus parts like the face or eyebrows.



Open the **Outfit Models** array and begin adding elements. Inside the fighter's skeleton hierarchy, the mesh nodes that belong to the outfit are highlighted in yellow.

Drag each of these nodes one by one into the array. There should be approximately nine outfit-related mesh instances in total.



Writing the _set_outfit Function

Now that both arrays are configured, return to the fighter_visual.gd script to write the function that applies the correct material to each outfit mesh. Create a new function called _set_outfit that accepts an outfit_index parameter of type int:

```
func _set_outfit (outfit_index : int):
 for model : MeshInstance3D in outfit_models:
```



```
  model.set_surface_override_material(0, outfit_materials[outfit_index])
```

The function iterates over each mesh in the outfit_models array and calls set_surface_override_material to replace the material at surface index 0. This approach uses a surface override rather than the material_overlay property used for the hit flash effect. The distinction is important: the overlay is a temporary layer applied on top of the base material, while the surface override replaces the base material entirely. This means that when the hit flash clears the overlay, the fighter reverts to the outfit color set by this function rather than the original default material.

Calling _set_outfit in _ready

The final step in the script is to call _set_outfit inside the _ready function so that each fighter receives their outfit when the game starts. Each fighter has a unique player_id — Player 1 has an ID of 0, and Player 2 has an ID of 1. These IDs correspond directly to the indices in the outfit_materials array (0 for green, 1 for blue).

Add the following line at the end of the _ready function:

```
_set_outfit(fighter.player_id)
```

The complete _ready function now looks like this:

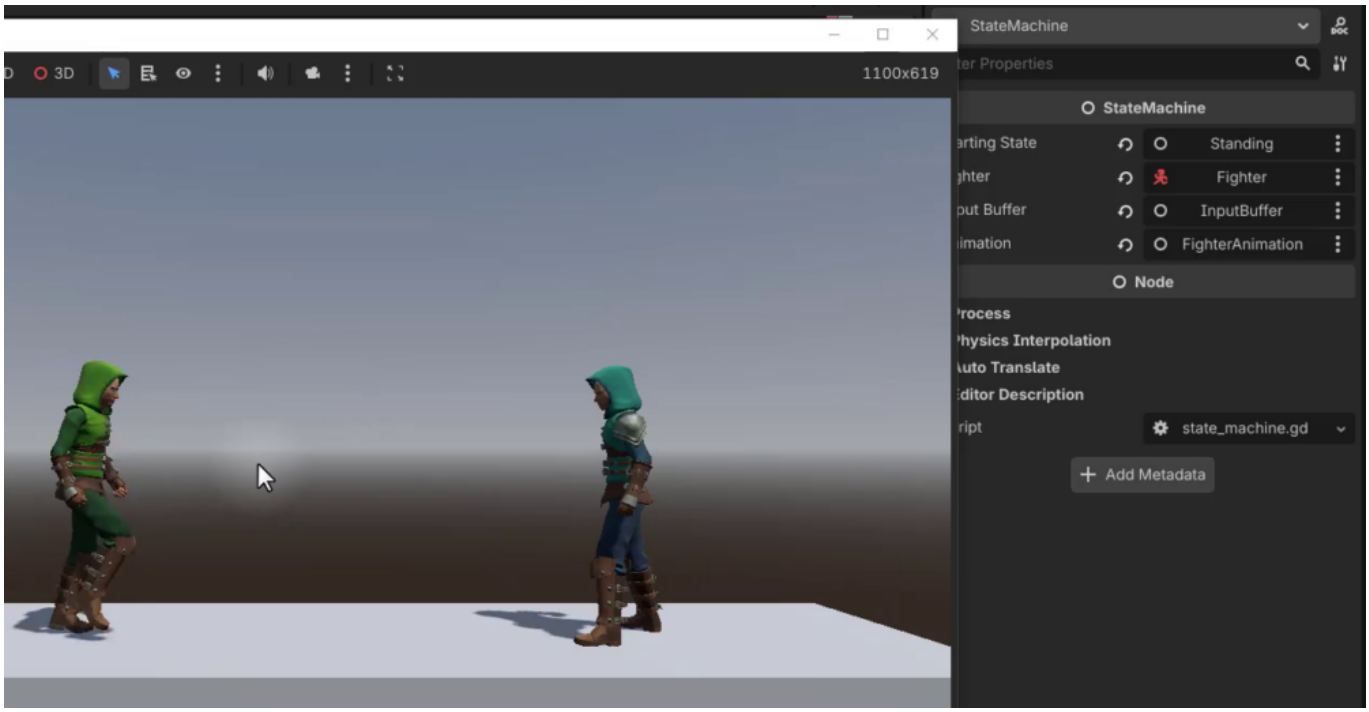
```
func _ready ():
 for child in model_root.find_children("*", "MeshInstance3D", true):
  models.append(child)

 GlobalEvents.FighterDamaged.connect(_hit_flash)

 _set_outfit(fighter.player_id)
```

Testing the Outfit System

After saving the script and running the game, both fighters now display their assigned outfit colors. Player 1 appears in green, and Player 2 appears in blue. When the fighters cross each other, it is now much easier to identify which character belongs to which player.

Setting the Starting State to Standing

One additional adjustment to make is changing the fighters' default state. Currently, both fighters begin the game in a jumping state, which causes them to jump immediately when the scene loads. To fix this, select the **StateMachine** node in the Fighter scene and change the **Starting State** property from its current value to **Standing**. Running the game again confirms that both fighters now start in their idle standing pose with their correct outfit colors applied.



Summary

In this lesson, we implemented a dynamic outfit system for our fighting game characters. The key

concepts covered include:

- Using exported arrays of StandardMaterial3D to store outfit color variants
- Maintaining a separate array of MeshInstance3D nodes to target only outfit-related meshes, leaving parts like the face unchanged
- Applying materials at runtime using set_surface_override_material rather than material_overlay, ensuring compatibility with the hit flash effect
- Assigning outfits based on each fighter's player_id during _ready
- Setting the **StateMachine** starting state to **Standing** for a proper game start

In the next lesson, we're going to add our last state: the dashing state.

In this lesson, we will implement the final state for our fighter character: the dashing state. Dashing allows the player to quickly move in a direction while becoming temporarily invulnerable, providing a way to evade enemy attacks. The dash can only be triggered while the player is already moving, and it includes a cooldown to prevent spamming.

Creating the Dashing State Script

Inside the Scripts/State Machine folder, create a new script called dashing_state.gd. Like all other states in the project, this script extends the State class.

Start by defining the class and the variables the dashing state requires:

```
class_name DashingState
extends State

@export var distance : float
@export var duration : float
@export var cooldown : float
var cooldown_end_time : float
var move_dir : int
```

The distance variable controls how far the fighter travels during the dash, while duration determines how long the dash lasts in seconds. The cooldown variable sets how long the player must wait before dashing again, and cooldown_end_time tracks when the cooldown expires. Finally, move_dir captures the direction the player is moving when the dash begins. This value is -1 for left, 0 for standing still, and 1 for right. By caching the direction at the start, the fighter continues dashing in the original direction even if the player changes their input mid-dash.

The can_enter Function

The can_enter function determines whether the fighter is allowed to enter the dashing state. Two conditions must be met: the player must be moving in a direction, and the cooldown must have elapsed.

```
func can_enter () -> bool:
 if input_buffer.move_direction() == 0:
  return false

 if Time.get_unix_time_from_system() < cooldown_end_time:
  return false

 return true
```

If input_buffer.move_direction() returns 0, the player is standing still and the dash is not allowed. If the current system time has not yet passed cooldown_end_time, the dash is still on cooldown and the function returns false. Otherwise, the state can be entered.

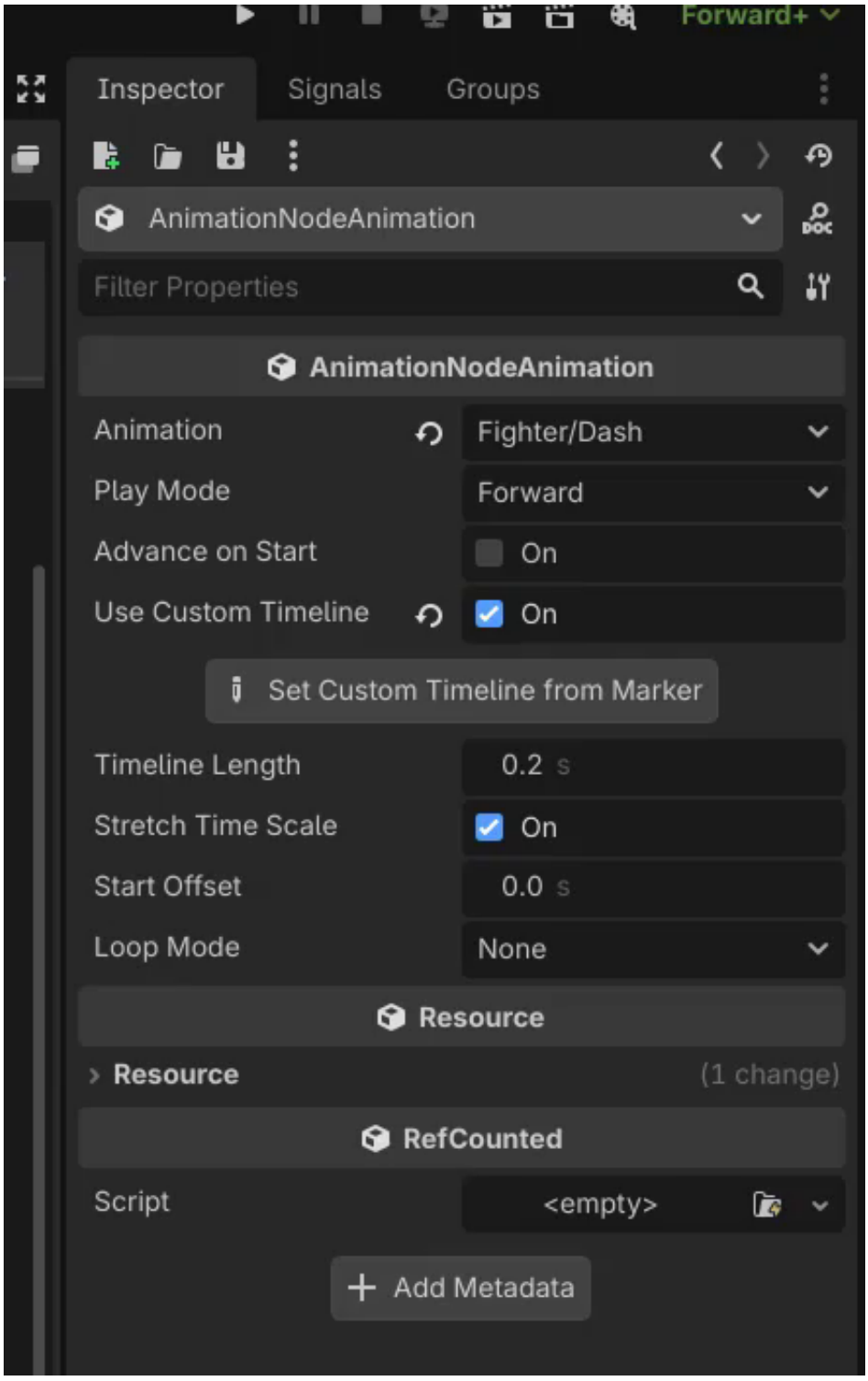
Entering the Dashing State

When the fighter enters the dashing state, several things happen: the fighter’s velocity is reset, invulnerability is enabled (to be implemented in the next course), the movement direction is cached,

and the dash animation is triggered.

```
func enter ():
 super.enter()

 fighter.move_velocity = Vector3.ZERO
 # Implement later: make the fighter invulnerable

 move_dir = input_buffer.move_direction()
 animation.set_animation("Dash")
```

Setting fighter.move_velocity to Vector3.ZERO stops any existing movement before the dash begins. The move_dir variable stores the current input direction, and the "Dash" animation is played through the animation tree.



The Update Function

The update function handles the actual movement during the dash and checks whether the dash duration has been exceeded.

```
func update (delta : float):
 super.update(delta)

 if local_time >= duration:
```



```
  state_machine.change_state("Standing")
  return

 var speed : float = distance / duration
 fighter.move_velocity.x = speed * move_dir
```

Each frame, the function first checks whether the time spent in this state (local_time) has exceeded the duration. If so, the state machine transitions back to the "Standing" state and the function returns early. Otherwise, the dash speed is calculated by dividing distance by duration, giving the velocity needed to cover the specified distance within the allotted time. This speed is then multiplied by move_dir to apply it in the correct horizontal direction.

Exiting the Dashing State

When the dash ends and the state is exited, the fighter's velocity is reset, invulnerability is removed (implemented in the next course), and the cooldown timer is set.

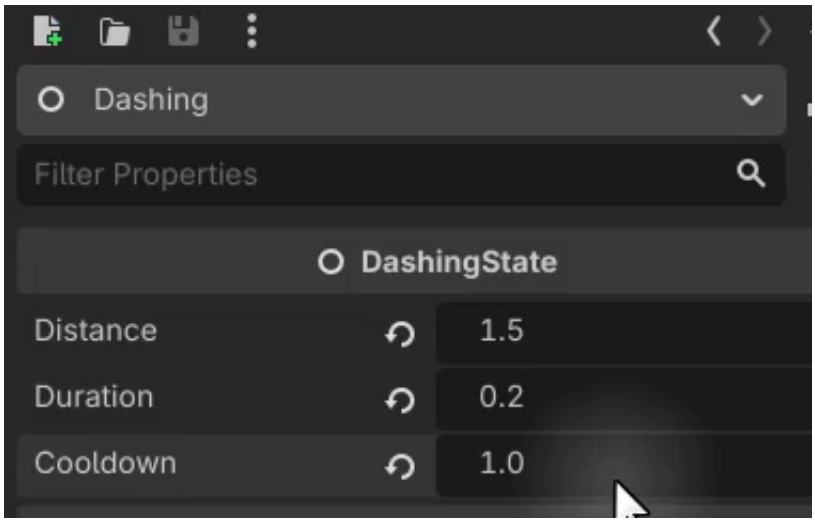
```
func exit ():
 super.exit()

 fighter.move_velocity = Vector3.ZERO
 # make the fighter vulnerable
 cooldown_end_time = Time.get_unix_time_from_system() + cooldown
```

Setting move_velocity back to Vector3.ZERO ensures the fighter does not continue sliding after the dash. The cooldown_end_time is set to the current time plus the cooldown value, preventing the player from dashing again until that time has passed. This approach is the same pattern used by the attacking states.

Adding the Dashing Node

With the script complete, the next step is to add a DashingState node to the state machine. In the Scene tree, right-click the StateMachine node, select **Add Child Node**, and choose DashingState. Rename the node to Dashing.

In the Inspector, configure the exported properties for the Dashing node. Set Distance to 1.5, Duration to 0.2, and Cooldown to 1.0. With these values, the fighter will travel 1.5 units in 0.2 seconds, making for a very quick dash, and must wait at least one second before dashing again.



Transitioning to the Dashing State

To allow the fighter to enter the dashing state, the StandingState script needs to be updated. In the update function of standing_state.gd, add a new input check below the existing jump check:

```
if input_buffer.is_pressed("dash"):
 state_machine.change_state("Dashing")
 return
```

When the player presses the dash input while in the standing state, the state machine attempts to change to the "Dashing" state. The can_enter function of the dashing state will verify that the player is moving and that the cooldown has elapsed before allowing the transition.

Testing the Dash

With everything in place, the dashing mechanic can be tested. For Player 1, pressing the S key (mapped to the dash input) while moving left or right triggers the dash. Standing still and pressing the dash button does nothing, since the can_enter function requires directional movement. Player 2 uses the down arrow key to dash in the same way.

The dash is fast enough to evade attacks if timed correctly. When an opponent swings, dashing in the opposite direction allows the fighter to quickly move out of range. However, the timing must be precise, as the dash duration is very short. The one-second cooldown also means the player cannot rely on spamming dashes to avoid all incoming damage.

With the dashing state implemented, all of the fighter’s states are now complete. The character can stand, move, jump, perform light and heavy attacks, take hits, be defeated, and now dash to evade incoming attacks.

Congratulations on completing the first course in this multi-course series on building a two-player local multiplayer fighting game in Godot. This lesson provides a recap of everything covered so far and a preview of what is coming next.

What Was Covered in This Course

This first course focused on setting up the foundational systems for the fighting game. The key topics covered include:

- **Creating a state-based fighter** — building the core Fighter character with a state machine to manage different behaviors such as standing, jumping, attacking, dashing, and being hit or defeated.
- **Animating the fighter** — setting up the AnimationTree with a state machine that handles transitions between animations like idle, movement, attacks, jumps, and reactions.
- **Detecting inputs for local multiplayer** — implementing an input system with player-specific action mapping, an input handler, and an input buffer that stores recent frames of input data for responsive controls.
- **Fighter visuals** — adding visual polish such as outfit changes based on player ID and a hit flash effect that briefly overlays a material when a fighter takes damage.

Current State of the Project

The project is not yet a complete game at this stage. Running the project will show that not everything is fully in place, as several key systems still need to be implemented. The core fighter mechanics, animations, and input handling are functional, but the game lacks essential elements like a user interface, game flow management, and audio.

What Is Coming in Part 2

The second course in the series will build on the foundation established here and complete the game. The upcoming topics include:

- **Finishing the visuals** — completing the visual presentation of the game, including level design.
- **Designing the HUD** — creating the heads-up display with health bars and stamina bars for each fighter.
- **Stamina system** — implementing a stamina mechanic to add strategic depth to the combat.
- **Game loop** — setting up the full game flow, including a main menu, a countdown before matches begin, and returning to the menu when a player is defeated.
- **Audio** — adding sound effects and music to bring the game to life.

By the end of the next course, the project will be a fully fledged local multiplayer fighting game.

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

fighter.gd

Found in folder: **/Scripts/Fighter**

This code defines a Fighter character for a 3D fighting game. It handles the fighter's movement, health, damage reception, and state management through physics updates. The fighter automatically faces its opponent, processes input to control actions, and can be defeated when health reaches zero.

```
class_name Fighter
extends CharacterBody3D

# Player identifier for multiplayer or tracking
@export var player_id : int
# Reference to the opposing fighter for facing direction and targeting
@export var opponent : Fighter

@export var current_health : float = 100.0
@export var max_health : float = 100.0

# External systems for input processing and state control
@export var input_handler : InputHandler
@export var input_buffer : InputBuffer
@export var state_machine : StateMachine

var forward_direction : int
var move_velocity : Vector3
var horizontal_force : float
var drag : float = 0.8
var is_defeated : bool = false

func _ready ():
 # Initialize all child components that can receive or send hits
 for child in find_children("*", "HitReceiver", true, false):
  child.initialize(self)

 for child in find_children("*", "HitSender", true, false):
  child.initialize(self)

func _physics_process (delta : float):
 # 1. Get the input packet
 var input_packet : InputPacket = input_handler.get_input_packet()

 # 2. Receive the input in the buffer
 input_buffer.receive_input(input_packet)

 # 3. Process the input in the state machine
 state_machine.update(delta)
```



```
 # 4. Update facing direction
 _update_facing_direction()

 # 5. Process movement
 _movement(delta)

# Rotate the fighter to always face their opponent
func _update_facing_direction ():
 if global_position.x < opponent.global_position.x:
  forward_direction = 1
  rotation_degrees.y = 90
 else:
  forward_direction = -1
  rotation_degrees.y = -90

# Push the player along the X axis
func add_force (force : float):
 horizontal_force += force

# Apply the move velocity and force to the fighter
func _movement (delta : float):
 horizontal_force *= drag

 velocity.x = move_velocity.x + horizontal_force
 velocity.y = move_velocity.y
 move_and_slide()

# Called when the opponent damages us
func take_damage (hit_receiver : HitReceiver, damage_amount : float):
 if is_defeated:
  return

 current_health -= damage_amount
 GlobalEvents.FighterDamaged.emit(self)

 if current_health <= 0:
  current_health = 0
  defeat()
 else:
  # Transition to hit reaction state and apply knockback
  state_machine.change_state(hit_receiver.hit_state_name)
  add_force(forward_direction * -10)

# Called when health reaches 0
func defeat ():
 is_defeated = true
 state_machine.change_state("Defeated")
 GlobalEvents.FighterDefeated.emit(self)
```

fighter_animation.gd

Found in folder: **/Scripts/Fighter**

This code provides a simplified interface for controlling an AnimationTree, allowing smooth transitions between blend values and switching animation states. It uses linear interpolation to gradually change blend positions and direct travel calls to change animations.

```
class_name FighterAnimation
extends Node

# Reference to the AnimationTree node for controlling animations
@export var anim_tree : AnimationTree

# Smoothly transitions a blend position towards a target value using linear interpola
tion
func set_blend_position (path : String, value : float):
 anim_tree[path] = lerp(anim_tree[path], value, 0.3)

# Directly switches to a new animation state in the state machine
func set_animation (animation_name : String):
 anim_tree["parameters/playback"].travel(animation_name)
```

fighter_visual.gd

Found in folder: **/Scripts/Fighter**

This code handles visual effects for a fighter character, specifically flashing the model with a hit material when damaged and applying an outfit material based on player ID. It connects to a global event to trigger the hit flash effect and sets the initial outfit when ready. The flash effect temporarily applies an overlay material, then removes it after a short delay.

```
# Manages the fighter's hit flash and outfit

class_name FighterVisual
extends Node

var models : Array[MeshInstance3D]  # Stores all MeshInstance3D nodes for the fighter
 model

@export var hit_material : StandardMaterial3D  # Material applied when fighter takes
damage
@export var model_root : Node3D  # Root node containing all fighter mesh instances

@onready var fighter : Fighter = $".."  # Reference to parent Fighter node

@export var outfit_materials : Array[StandardMaterial3D]  # Materials for different p
layer outfits
@export var outfit_models : Array[MeshInstance3D]  # Specific models that receive out
fit materials

func _ready ():
 # Collect all MeshInstance3D children under model_root
 for child in model_root.find_children("*", "MeshInstance3D", true):
  models.append(child)

 # Connect to global damage event to trigger visual feedback
```



```
 GlobalEvents.FighterDamaged.connect(_hit_flash)

 # Set initial outfit based on player ID
 _set_outfit(fighter.player_id)

func _hit_flash (fighter : Fighter):
 # Only respond if the damaged fighter matches this visual's fighter
 if self.fighter != fighter:
  return

 # Apply hit material overlay to all fighter meshes
 for model : MeshInstance3D in models:
  model.material_overlay = hit_material

 # Wait briefly before removing the overlay
 await get_tree().create_timer(0.05).timeout

 # Remove hit material overlay
 for model : MeshInstance3D in models:
  model.material_overlay = null

func _set_outfit (outfit_index : int):
 # Apply outfit material to specific models based on player ID
 for model : MeshInstance3D in outfit_models:
  model.set_surface_override_material(0, outfit_materials[outfit_index])
```

global_events.gd

Found in folder: **/Scripts/Globals**

This code defines a global signal bus using Godot's AutoLoad feature, allowing other parts of the game to emit or listen to signals without direct node references. It declares two custom signals: `FighterDamaged` and `FighterDefeated`, both passing a `Fighter` object as a parameter. These signals can be used to communicate events related to fighter characters throughout the game.

```
# A collection of signals which can be emitted and connected
# from anywhere in the code without needing to reference a node
# as it is registered as an AutoLoad (singleton)

extends Node

# Emitted when a fighter takes damage, providing the affected fighter instance
signal FighterDamaged (fighter : Fighter)
# Emitted when a fighter is defeated, providing the defeated fighter instance
signal FighterDefeated (fighter : Fighter)
```

hit_receiver.gd

Found in folder: **/Scripts/Hitboxes**

This code defines a 3D area that receives hits for a fighter character. When hit with damage, it calls

the fighter's damage-taking function. The hit state name can be configured externally to determine how the fighter reacts.

```
# Area3D which represents a fighter's hit box

class_name HitReceiver
extends Area3D

var fighter : Fighter
@export var hit_state_name : String  # Name of the state to enter when hit

func initialize (fighter : Fighter):
 self.fighter = fighter

func hit (damage : int):
 fighter.take_damage(self, damage)  # Forward damage to the parent fighter
```

hit_sender.gd

Found in folder: **/Scripts/Hitboxes**

This code defines a HitSender class that detects overlapping HitReceiver areas belonging to opposing fighters. It checks all overlapping areas and returns the first HitReceiver from a different player. If no valid opponent receiver is found, it returns null.

```
# Area3D which detects when overlapping an opponent HitReceiver

class_name HitSender
extends Area3D

var fighter : Fighter  # Reference to the fighter that owns this sender

func initialize (fighter : Fighter):
 self.fighter = fighter

func detect_hit () -> HitReceiver:
 var areas : Array[Area3D] = get_overlapping_areas()

 for area in areas:
  if area is not HitReceiver:
   continue  # Skip non-HitReceiver areas

  if area.fighter.player_id == fighter.player_id:
   continue  # Skip receivers belonging to same player

  return area  # Return first valid opponent receiver

 return null  # No valid opponent receiver found
```

input_buffer.gd

Found in folder: **/Scripts/Inputs**

This code stores recent input events in a fixed-size array, allowing the game to check both current and past inputs. It provides methods to determine if an input is currently active, calculate a movement direction, or detect if an input occurred within a recent window of frames. The buffer automatically discards old inputs when it exceeds its maximum size.

```
# Receives input packets and stores them temporarily in a rolling buffer
# By default, the last 20 frames of inputs are stored
# This allows us to detect inputs x number of frames previous

class_name InputBuffer
extends Node

const BUFFER_SIZE : int = 20

var buffer : Array[InputPacket] = []

func _ready ():
 # Initialize buffer with empty packets to avoid null checks
 for i in range(BUFFER_SIZE):
  buffer.push_front(InputPacket.new())

func receive_input (packet : InputPacket):
 buffer.push_front(packet)

 # Maintain fixed buffer size by removing oldest entry
 if buffer.size() > BUFFER_SIZE:
  buffer.pop_back()

func is_pressed (input_string : String) -> bool:
 # Check only the most recent input packet
 var packet : InputPacket = buffer[0]
 return packet.is_pressed(input_string)

func move_direction () -> int:
 var dir : int = 0

 if is_pressed("move_left"):
  dir -= 1
 if is_pressed("move_right"):
  dir += 1

 return dir

func was_pressed (input_string : String, buffer_window : int = 3) -> bool:
 # Scan recent packets within the specified window
 for i in range(min(buffer_window, buffer.size())):
  if buffer[i].is_pressed(input_string):
   return true

 return false
```


input_handler.gd

Found in folder: **/Scripts/Inputs**

This code defines a base class for input handling in a game. It provides a template for different input sources like players or AI by requiring them to implement a method that returns input data. The class references a Fighter object that will receive the input.

```
# Base class for all input handlers - this is not the node we create
# but rather the template for different handlers (player, enemy AI, networked player)

class_name InputHandler
extends Node

# Reference to the Fighter that will receive and process the input
@export var fighter : Fighter

# This function will be overridden by all who extend from this class
# Returns an InputPacket containing movement/action data from the specific handler
func get_input_packet () -> InputPacket:
 return null
```

input_packet.gd

Found in folder: **/Scripts/Inputs**

This code defines a class that stores a snapshot of player input states in a dictionary. It provides a method to check if a specific input action is currently pressed. The class is designed to be instantiated dynamically rather than attached to a scene node.

```
# This script is never attached to a node, rather
# it is an object we will create when needed in code

class_name InputPacket

# A dictionary containing all possible inputs the player can make
# as well as whether or not they are pressed this frame
var inputs : Dictionary[String, bool] = {
 "move_left": false,
 "move_right": false,
 "jump": false,
 "dash": false,
 "light_attack": false,
 "heavy_attack": false
}

# Returns true if the requested input is true
func is_pressed (input_string : String) -> bool:
 # Safely retrieve input state, defaulting to false if key doesn't exist
 return inputs.get(input_string, false)
```


player_input_handler.gd

Found in folder: **/Scripts/Inputs**

This code handles player-specific input by mapping actions to a player ID and creating a packet of current input states. It extends a base input handler to check if each action (like move or jump) is currently pressed for a specific player. The packet collects these boolean states for use elsewhere in the game.

```
class_name PlayerInputHandler
extends InputHandler

# Returns the full action name with player_id
# e.g. jump -> 0_jump
func _full_action_name (action : String) -> String:
 return str(fighter.player_id, "_", action)

# Get all inputs this frame from the player and
# put them inside an InputPacket
func get_input_packet () -> InputPacket:
 var packet : InputPacket = InputPacket.new()

 # Check each player-specific action and store its pressed state
 packet.inputs["move_left"] = Input.is_action_pressed(_full_action_name("move_left"))
 packet.inputs["move_right"] = Input.is_action_pressed(_full_action_name("move_right"
))
 packet.inputs["jump"] = Input.is_action_pressed(_full_action_name("jump"))
 packet.inputs["dash"] = Input.is_action_pressed(_full_action_name("dash"))
 packet.inputs["light_attack"] = Input.is_action_pressed(_full_action_name("light_att
ack"))
 packet.inputs["heavy_attack"] = Input.is_action_pressed(_full_action_name("heavy_att
ack"))

 return packet
```

attacking_state.gd

Found in folder: **/Scripts/State Machine**

This code defines a base state for attacks that handles cooldown management, animation playback, forward movement, and hit detection during a specific time window. It transitions back to a “Standing” state after the attack duration ends and sets a cooldown timer upon exit to prevent immediate reuse.

```
# Base state for all attacking states
# This state is not created as a node, but acts as a template for light and heavy att
ack
# Makes it so we don't have to repeat code for those 2 states

class_name AttackingState
extends State

# Attack properties
```



```
@export var damage : int
@export var animation_name : String
@export var duration : float

@export var cooldown : float
var cooldown_end_time : float

@export var forward_force : float  # Force applied during attack

@export var hit_sender : HitSender  # Component responsible for detecting hits
var has_hit : bool = false  # Prevents multiple hits per attack
@export var hit_detect_start_time : float  # Start of hit detection window
@export var hit_detect_end_time : float    # End of hit detection window

# Wait for cooldown to end before entering state
func can_enter () -> bool:
 if Time.get_unix_time_from_system() < cooldown_end_time:
  return false

 return true

func enter ():
 super.enter()

 # Play animation and push forward
 animation.set_animation(animation_name)
 fighter.add_force(forward_force * fighter.forward_direction)
 has_hit = false # Reset hit flag for new attack

func update (delta : float):
 super.update(delta) # Only detect hits during the specified time window
 if local_time >= hit_detect_start_time and local_time <= hit_detect_end_time:
  var hit : HitReceiver = hit_sender.detect_hit()

  if hit and has_hit == false:
   has_hit = true # Mark that a hit has occurred
   hit.hit(damage) # Return to Standing after duration has elapsed

 if local_time >= duration:
  state_machine.change_state("Standing")

func exit ():
 super.exit()
 cooldown_end_time = Time.get_unix_time_from_system() + cooldown  # Set cooldown time
r
```

dashing_state.gd

Found in folder: **/Scripts/State Machine**

This code defines a dashing state for a fighter character, allowing them to quickly dash in a direction with invulnerability. It checks if dashing is allowed based on input direction and cooldown timing, then applies movement velocity during the dash duration. After the dash ends, it resets velocity,

removes invulnerability, and sets a cooldown period.

```
# State for when the dash button is pressed while moving
# Quickly speed them in that direction, avoiding any possible damage

class_name DashingState
extends State

@export var distance : float  # Total distance to travel during dash
@export var duration : float  # How long the dash lasts in seconds
@export var cooldown : float  # Time before dash can be used again
var cooldown_end_time : float  # Timestamp when dash becomes available
var move_dir : int  # Direction multiplier (-1 for left, 1 for right)

func can_enter () -> bool:
 # Check if there's valid movement input
 if input_buffer.move_direction() == 0:
  return false

 # Check if dash is still on cooldown
 if Time.get_unix_time_from_system() < cooldown_end_time:
  return false return true

func enter ():
 super.enter()

 # Reset velocity and enable invulnerability for dash
 fighter.move_velocity = Vector3.ZERO
 # make the fighter invulnerable

 # Store direction and play dash animation
 move_dir = input_buffer.move_direction()
 animation.set_animation("Dash")

func update (delta : float):
 super.update(delta) # Transition to standing state when dash duration ends

 if local_time >= duration:
  state_machine.change_state("Standing")
  return

 # Calculate and apply dash velocity
 var speed : float = distance / duration
 fighter.move_velocity.x = speed * move_dir

func exit ():
 super.exit()

 # Clean up dash effects and start cooldown timer
 fighter.move_velocity = Vector3.ZERO
 # make the fighter vulnerable
 cooldown_end_time = Time.get_unix_time_from_system() + cooldown
```

defeated_state.gd

Found in folder: **/Scripts/State Machine**

This code defines a DefeatedState class that triggers when a fighter's health reaches zero, playing a "Defeated" animation and locking the fighter from further actions until returning to the menu. It inherits from a State class and overrides the enter method to set the specific animation upon entering this state.

```
# State which triggers upon health reaching 0
# Locks the fighter out of doing anything else until returning to the menu

class_name DefeatedState
extends State

func enter ():
 super.enter()  # Call parent class enter method to handle any base state setup
 animation.set_animation("Defeated")  # Play the defeated animation when this state i
s entered
```

heavy_attack_state.gd

Found in folder: **/Scripts/State Machine**

This code defines a new state called HeavyAttackState that inherits from AttackingState. It sets up the class for use within the Godot engine's scene system.

```
# Declares a new class named HeavyAttackState
class_name HeavyAttackState
# Inherits from the AttackingState class, extending its functionality
extends AttackingState
```

hit_state.gd

Found in folder: **/Scripts/State Machine**

This code defines a state where the fighter is temporarily stunned after taking damage, playing a specific animation for a set duration before transitioning back to a standing state.

```
# State for when the fighter gets damaged
# Temporarily stun them

class_name HitState
extends State

@export var duration : float  # How long the stun lasts in seconds
@export var animation_name : String  # Name of the animation to play during hit state

func enter ():
 super.enter()
 animation.set_animation(animation_name)  # Start playing the hit animation
```



```
func update (delta : float):
 super.update(delta)

 if local_time >= duration:  # Check if stun duration has elapsed
  state_machine.change_state("Standing")  # Transition back to standing state
```

jumping_state.gd

Found in folder: **/Scripts/State Machine**

This code handles a character's jumping behavior. It applies an initial upward force when entering the jump state, then applies gravity during the jump. When the character lands on the floor, it transitions back to the standing state.

```
class_name JumpingState
extends MovementState

# Configuration variables for jump strength and gravity effect
@export var jump_force : float = 5.0
@export var gravity : float = 10.0

func enter ():
 # Apply initial upward force when entering jump state
 fighter.move_velocity.y = jump_force
 animation.set_animation("JumpStart")

func update (delta : float):
 # Apply gravity to gradually reduce upward velocity
 fighter.move_velocity.y -= gravity * delta

 super.update(delta)

 # Transition to standing state when landing on floor
 if fighter.is_on_floor():
  state_machine.change_state("Standing")

func exit ():
 # Reset vertical velocity and play landing animation when exiting jump state
 fighter.move_velocity.y = 0
 animation.set_animation("JumpLand")
```

light_attack_state.gd

Found in folder: **/Scripts/State Machine**

This code defines a new state called LightAttackState that inherits from AttackingState. It sets up a state class for handling light attack behavior within a state machine.

```
class_name LightAttackState
extends AttackingState
```



```
# State for handling light attack animations and logic.
# Inherits base attacking functionality from AttackingState.
```

movement_state.gd

Found in folder: **/Scripts/State Machine**

This code defines a base movement state that handles horizontal movement and animation blending for derived states like standing or jumping. It updates the fighter's velocity based on input direction and sets animation blend positions accordingly. When exiting this state, it resets horizontal velocity to zero.

```
# Base class for all movement related states (Standing & Jumping)
# This is not an actual state no we use, but rather a template
# It makes it so we don't have to repeat code for both standing and jumping

class_name MovementState
extends State

@export var move_speed : float = 1.5
@export var blend_position_parameter : String

func update (delta : float):
 super.update(delta)

 # Set movement direction
 var move_dir : int = input_buffer.move_direction()
 fighter.move_velocity.x = move_dir * move_speed

 # Blend the animation
 if blend_position_parameter.length() == 0:
  return

 var blend_pos : float = move_dir * fighter.forward_direction
 animation.set_blend_position(blend_position_parameter, blend_pos)

func exit ():
 super.exit()

 fighter.move_velocity.x = 0  # Stop horizontal movement when leaving state
```

standing_state.gd

Found in folder: **/Scripts/State Machine**

This code defines a standing state for a character, checking for player inputs each frame. If specific actions like attacks, jump, or dash are detected, it transitions to the corresponding states. It inherits from a base movement state to maintain common update logic.

```
# The default/core state
```



```
# This is where we detect inputs to transition to other states

class_name StandingState
extends MovementState

func update (delta : float):
 super.update(delta)  # Process any base movement state logic first

 # Check for various inputs and transition to appropriate states
 if input_buffer.is_pressed("light_attack"):
  state_machine.change_state("LightAttack")
  return  # Exit early to prevent multiple state changes

 if input_buffer.is_pressed("heavy_attack"):
  state_machine.change_state("HeavyAttack")
  return

 if input_buffer.is_pressed("jump"):
  state_machine.change_state("Jumping")
  return

 if input_buffer.is_pressed("dash"):
  state_machine.change_state("Dashing")
  return
```

state.gd

Found in folder: **/Scripts/State Machine**

This code defines a base State class that serves as a template for specific states in a state machine. It provides common properties like timing variables and references to fighter systems, along with lifecycle methods such as enter, exit, and update. The class is designed to be inherited and its methods overridden by concrete state implementations.

```
# Base class for all states
# This is not a state, but rather the template
# It includes functions which we can override and helpful variables/properties

class_name State
extends Node

var state_machine : StateMachine

# Time we entered this state
var enter_time : float

# Time that has elapsed since entering this state
var local_time : float

# A collection of property wrappers which return different fighter systems
# These are get only variables, meaning we cannot set them, as their value is
# dependent on another value. e.g. fighter links to our fighter node
var fighter : Fighter:
```



```
 get: return state_machine.fighter

var input_buffer : InputBuffer:
 get: return state_machine.input_buffer

var animation : FighterAnimation:
 get: return state_machine.animation

# Called when the fighter is loaded in
func initialize (state_machine : StateMachine):
 self.state_machine = state_machine

# Returns true if we can enter this state
func can_enter () -> bool:
 return true

# Called when this state first becomes active
func enter ():
 enter_time = Time.get_unix_time_from_system()

# Called when this state is exited
func exit ():
 pass

# Called each physics frame this state is active
func update (delta : float):
 var time : float = Time.get_unix_time_from_system()
 local_time = time - enter_time
```

state_machine.gd

Found in folder: **/Scripts/State Machine**

This code implements a finite state machine for managing character states. It is initialized by collecting child nodes that are State objects and setting an initial state. The state machine allows changing states based on conditions and updates the current state each frame.

```
class_name StateMachine
extends Node

# Exported variables for configuration in the editor
@export var starting_state : State  # The initial state when the state machine starts

@export var fighter : Fighter        # Reference to the fighter entity this state mac
hine controls
@export var input_buffer : InputBuffer  # Buffer for handling input queuing
@export var animation : FighterAnimation  # Animation controller for the fighter

var states : Dictionary[String, State] = { }  # Dictionary mapping state names to Sta
te objects
var current_state : State                     # Currently active state

func _ready ():
```



```
 # Find all states that are children nodes of this one
 for child in get_children():
  if child is not State:
   continue

  states[child.name] = child
  child.initialize(self)  # Pass state machine reference to each state for setup

 # Set the starting state (Standing)
 if starting_state:
  change_state(starting_state.name)

# Called whenever we want to change our state
func change_state (state_name : String):
 # Return if the state doesn't exist
 if not states.has(state_name):
  printerr("Cannot change state to ", state_name, " as it doesn't exist!")
  return

 # Return if we can not yet enter the state
 if not states[state_name].can_enter():
  return

 # Exit our current state
 if current_state:
  current_state.exit()

 # Enter the new state
 current_state = states[state_name]
 current_state.enter()

 print("New State: ", state_name)  # Debug output for state transitions

# Called by the Fighter script every frame - update the current state
func update (delta : float):
 if not current_state:
  return

 current_state.update(delta)

# Returns true if the requested state is the current one
func is_current_state (state_name : String) -> bool:
 if not current_state:
  return false

 return current_state.name == state_name
```