Arena FPS Controller

User Guide

Table Of Contents

Quickstart

The Demo Scene

To try out the character controller right away:

1. Import the Arena FPS Controller package into unity.
2. Enter the “AFPSDemo” scene under ArenaFPSController/Demo.
3. Press Play and try out the example character controller.

The scene is arranged as the following:

- **Scene:** Contains all geometry and content organised in different gameobjects.
- **CharacterController:** The character itself which you control.

Controls:

- WASD to move.
- Mouse to look around.
- Space to jump.
- C or Ctrl to crouch.
- N to noclip.

Package Overview

Package structure

- **Core:** This folder contains the core scripts of the character controller. These scripts are the ones you must keep if you use this package.
- **Demo:** This folder contains the optional example component used for the demo. It includes the example player prefab and example controller.

Core scripts

- **AFPSMotor:** This is the physics solver of the controller.
- **AFPSCollisionSolver:** This is the collision solver of the controller.
- **AFPSControllerBase:** The script to inherit from when creating a custom controller
- **AFPSSettings:** The script to generate the settings scriptableobject.

Demo scripts

- **ArenaFPSController:** The example controller used in the demo.
- **PlayerInput:** A script generated by the PlayerInput.InputAction.

Integration Guide

Setting Up Layer Masks

The "**ArenaFPSCollisionSolver**" relies on layer masks to distinguish between walkable surfaces, triggers, and collidable surfaces.

- Go to Edit > Project Settings > Tags and Layers.
- Ensure your environment geometry is assigned to a specific layer (e.g., World, Default, or Environment).

Dropping in the Player Prefab

- Locate the AFPSController prefab in ArenaFPSController/Demo/Prefabs.
- Drag and drop the prefab into your hierarchy.
- In the Inspector for the Collision Solver component, assign your layer mask to the Collision Mask and Ground Mask fields.

Configuring the settings

- All movement settings are controlled via the **ArenaFPSSettings ScriptableObject.**
- The provided player prefab uses the demo settings scriptable object located under ArenaFPSController/Demo/Scripts.
- To create a custom settings Scriptable Object, right click in your project view and select Create > Arena FPS Movement > Arena FPS Settings.

Arena FPS Settings

All movement physics are controlled via the ArenaFPSSettings Scriptable Object config file. You can create multiple presets by right clicking in your project view and selecting **Create > Arena FPS > Movement Settings**.

Speed Settings

- **SprintSpeed:** How fast the player moves when sprinting.
- **WalkSpeed:** How fast the player moves when walking normally.
- **CrouchSpeed:** How fast the player moves while ducking.
- **AirSpeed:** The maximum speed limit you can manually *add* with keyboard inputs while in mid-air. Keeping this low is what allows for sharp, controllable Quake-style air strafing.

Acceleration Settings

- **Acceleration:** How quickly the player builds up speed from a standstill on the ground.
- **AirAcceleration:** How responsive the player's controls are in mid-air. High values allow for tight air turning and Source style ramp surfing.
- **FrictionCoefficient:** The ground friction that naturally slows the player down when they let go of the movement keys.
- **StopSpeed:** A velocity threshold that forces the player's momentum to quickly go to zero, preventing them from feeling like they are sliding on ice when trying to stop.

Jump Settings

- **JumpForce:** The vertical power of the player's jump. Higher values mean higher jumps.
- **JumpDelay:** A brief cooldown required between jumps to prevent players from spamming the jump key.
- **CoyoteTime:** A tiny grace period that lets the player jump even if they just walked off the physical edge of a ledge.
- **JumpBufferTime:** A short window that remembers a jump press right *before* hitting the ground, executing a perfect jump the exact frame you land.

Crouch Settings

- **StandingHeight:** The world height of the player's collision capsule when standing up.

- **CrouchHeight:** The world height of the player's collision capsule when ducking down.

Ladder Settings

- **LadderTag:** The exact Unity Tag name your scripts look for to identify climbable ladder objects in your world.
- **ClimbSprintSpeed:** How fast you climb up or down a ladder while holding the sprint key.
- **ClimbWalkSpeed:** Your standard, controlled climbing speed on a ladder.

Physics Settings

- **Gravity:** The downward pulling force applied to the player whenever they are airborne.
- **MaxSpeed:** The absolute hard speed limit the player can reach from any combination of surfing, or strafe-jumping.
- **TerminalVelocity:** The maximum speed the player can reach when falling straight down through the air.
- **MaxSlopeAngle:** The steepest ramp or hill the player can walk up normally before the physics engine considers them "airborne" and makes them slide off.
- **MaxStepHeight:** The tallest curb, step, or ledge lip the character can automatically step over without needing to press the jump key.
- **NoClipSpeed:** The free-flying movement speed used when cheating or debugging through walls in noclip mode.
- **CeilingRaycastExtraLength:** A tiny safety buffer shot above the player's head before standing up, ensuring they won't clip their head through low ceilings.
- **GroundRaycastExtraLength:** A tiny safety distance cast downward to keep the player smoothly snapped to the floor when sprinting down steep ramps or stairs.

Advanced Manual Setup

If you prefer not to use the provided **ArenaFPSPlayer** prefab and want to assemble the controller manually onto a custom GameObject, follow these steps:

Component Hierarchy

- Create a new empty GameObject in your scene hierarchy and name it (e.g., CustomPlayer).
- Create a child GameObject named **Model**, and place your player model inside it.
- Create another child GameObject and call it **CameraHolder**.
- Create a camera and place it under CameraHolder.

Attaching the Scripts

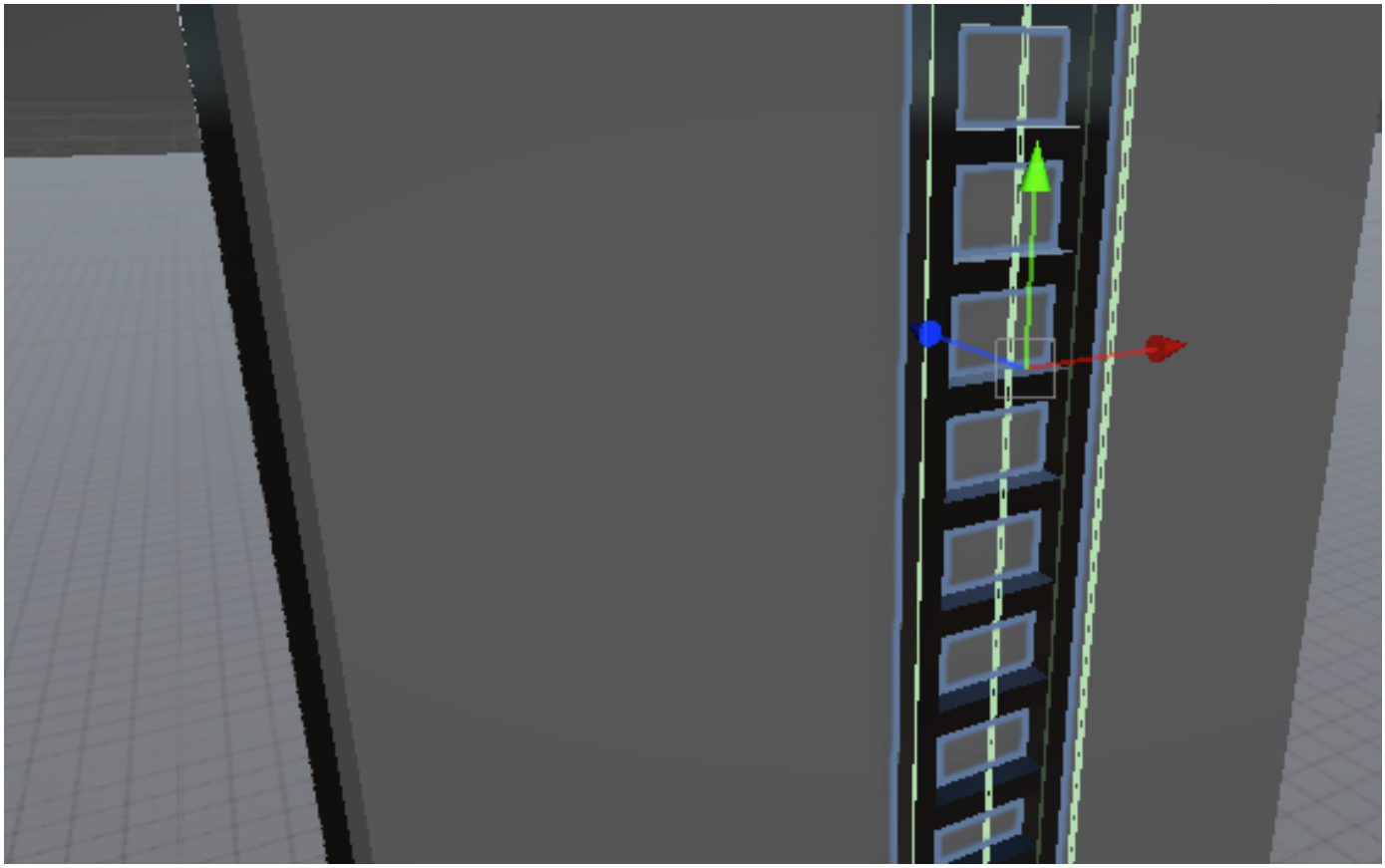
- Attach the AFPSMotor script to the root CustomPlayer object. The AFPSMotor will attach all dependencies on its own
- Create a custom settings Scriptable Object by right clicking in your project view and select Create > Arena FPS Movement > Arena FPS Settings.
- You will also need to create a custom controller script(Creating one). Alternatively, you can use the provided **ArenaFPSController** under ArenaFPSController/Demo/Scripts.

Setting Inspector References/Values

- Select the CustomPlayer object and look at the Arena FPS Motor component.
- Drag your ArenaFPSSettings Scriptable Object config file into the Settings slot (optionally set the model to make it change size based on if the player is crouching or not).
- Look at the Arena FPS Collision Solver component:
- Assign your environment layers to the Collision Mask and Ground Mask drop-downs.
- Set the Radius from the AFPSCollisionSolver which will automatically match the capsule collider's radius.
- The skin width is a small buffer to keep the character from getting stuck in walls due to floating-point errors. (default value is fine)
- The hit budget is how many things can be in contact with the controller and still register them instead of stopping. (the default value is a bit generous)

Ladder Usage

To use the ladder function you must have a collider that has Is Trigger enabled, and has the same tag as the ladder tag set in the Arena FPS Settings. The orientation of the ladder matters: The local Z axis must be facing the wall the ladder is on as shown in the picture.



Creating a Custom Controller

The core movement motor does not use Unity's input systems directly. Instead, it reads data from AFPSControllerBase. This decoupled design makes it easy to swap between the Legacy Input Manager, the New Input System package, mobile touch controls, or even AI/Bot scripts without changing a single line of physics code.

To implement your own custom controller or input handler, create a new script that inherits from AFPSControllerBase and feed it data from your preferred input framework.

The Input Architecture

Your custom script simply needs to overwrite these inherited fields every frame:

- MoveInput (Vector2): Raw directional joystick/WASD layout.
- Move (Vector3): Calculated world-space direction vector based on your camera's heading.
- LookRotation (Vector3): Mouse pitch and yaw data for the look system.
- Jump, Crouch, Sprint (bool): Action state flags.

Implementation Example: Unity New Input System

Below is an implementation example demonstrating how to hook up Unity’s modern Input System Package to the AFPSMotor:

```
using UnityEngine;
using ArenaFPSMovement;

public class ExampleNewInputController : AFPSControllerBase
{
    private PlayerInput input; // Generated C# wrapper class
    [Header("Settings")]
    public float Sensitivity = 15f;
    public float MaxVerticalRotation = 90f;

    private void Awake() => input = new PlayerInput();
    private void OnEnable() => input.Enable();
    private void OnDisable() => input.Disable();

    private void Update()
    {
        // Get LookDelta
        Vector2 look =
input.CharacterController.Look.ReadValue<Vector2>();
        // Calculate and clamp current look rotation
        LookRotation += new Vector3(-look.y, look.x, 0f) * Sensitivity *
Time.deltaTime;
        LookRotation.x = Mathf.Clamp(LookRotation.x, -MaxVerticalRotation,
MaxVerticalRotation);

        // Get raw move input
        Vector2 moveRaw =
input.CharacterController.Move.ReadValue<Vector2>();
        MoveInput = moveRaw; //Store Raw Vector for normal movement
        // Calculate look-dependent movement for Ladders and Noclip
     Quaternion horizontalHeading = Quaternion.Euler(0f, LookRotation.y,
0f);
        Move = (horizontalHeading * Vector3.forward * moveRaw.y) +
(horizontalHeading * Vector3.right * moveRaw.x);

        //Movement Action State Mapping
        Crouch = input.CharacterController.Crouch.IsPressed();
        Jump = input.CharacterController.Jump.IsPressed();
        Sprint = input.CharacterController.Sprint.IsPressed();
        if (input.CharacterController.NoClip.WasPressedThisFrame())
```

```
            Noclip = !Noclip;
    }
}
```

How to use your Custom Controller in-scene:

- Attach your newly written **ExampleNewInputController** script to the player as the **AFPSMotor** will throw out an error if an input controller is not found.

Usage with Bots

Because input is abstract, you can create a **BotController** that inherits from **AFPSControllerBase**. Instead of hardware inputs, use a **Unity NavMeshAgent or AI routine to calculate path directions and pass them directly into the MoveInput vector**. The bot will use the physics, step-climbing, and sliding algorithms exactly like a human player.

Motor API References

To hook up your own custom systems like gameplay triggers, camera shakes, audio managers, or UI readouts the **`ArenaFPSMotor`** exposes a clean set of C# events and public methods.

Public C# Events

Subscribe to these to execute code when specific movement events happen in the engine.

- **OnJumped:** Fires the exact frame the player successfully triggers a jump.
- **OnLanded (passes float):** Fires the frame the player capsule collides with a walkable floor after being airborne. It passes a **float** representing the **downward impact velocity**.
- **OnStateChanged (passes MovementState):** Fires whenever the player changes Movement State(e.g., switching from *Walking* to Sprinting, or Crouching).

Public Physics Methods

Call these from external scripts (like weapons, hazards, or level mechanics) to manually alter the player's velocity.

- **SetVelocity(Vector3 newVelocity):** Instantly overwrites the player's current speed and direction with a brand new vector value.
- **AddForce(Vector3 force):** Appends an impulse vector directly on top of the player's existing momentum without resetting it.