Third-Person Finite State Machine and Movement

Documentation v1.0
Last Updated 11/3/2025

Contents

1. Quick Start Guide

REQUIRES CINEMACHINE 3.0 TO BE INSTALLED!

Navigate to 'DemoScene' in the ThirdPersonMovement folder and hit play. You should have a character that can walk, run, sprint and jump!

If you have already built a world and need to put a character in, simply drag the 'CORE OBJECTS' prefabs into your scene and ensure all references are hooked up correctly (see 'creating a new player character'). Also, delete any existing camera that is in your scene.

The default inputs are:

Camera – Mouse

Move – WASD

Jump – Space

You can begin to create new states by following the more detailed guide below. Keep an eye out for video walkthroughs coming soon!

2.. Asset Overview

This asset uses a professional level finite state-machine system to achieve third-person movement ... a perfect base for platforms, RPGs, third-person shooters and action-adventures!

Included are the basic states Free Look (running around), Jump and Fall. The structure of the state-machine makes it easy to add extra states (which will be available to purchase soon, or make your own!) to enhance and personalise your gameplay, while keeping the code-base clean.

The demo scene is a simple setup, allowing you to test the movement straight away and begin prototyping as fast as possible.

This documentation will help you understand the project hierarchy, flow for creating new states, the various helper methods throughout the state machine and setting up a new character from scratch, including using animations efficiently (no more spider-web hell in the animator!)

3. Creating a New Player Character

You can use the existing prefab character, and just swap out the visual child object (ensuring to change the animator avatar as well) and tweaking the character controller values if you wish. Otherwise, create a new Empty game object and add the following components:

- CharacterController - For movement
- Animator - For animations
- PlayerStateMachine - The state machine script
- InputReader - For input handling
- ForceReceiver - For gravity and external forces

Next, configure the following components:

PlayerStateMachine Inspector:

- Animator: Drag the Animator component
- Controller: Drag the CharacterController component
- ForceReceiver: Drag the ForceReceiver component
- FreeLookCamera: Drag your Cinemachine FreeLook camera
- InputReader: Drag the InputReader component

- MainCameraTransform: Drag in the Main Camera OR Leave empty (auto-assigned at runtime)

Free Look Movement:

- FreeLookMovementSpeed: 5-8 for walking
- FreeLookSprintMovementSpeed: 8-12 for sprinting
- RotationDamping: 10-15 (higher = faster rotation)

Jump Movement:

- JumpForce: 8-12 (depends on gravity scale)

ForceReceiver:

- Controller: Drag the CharacterController component
- Agent: Drag NavMeshAgent if using AI (leave empty for player)
- Drag: 0.3 (how quickly knockback dissipates)

Finally, add your character model as a child of this empty parent object.

4. State Architecture

Every state inherits from PlayerBaseState. They all contain a constructor to store a reference the state machine, and implement three core methods: Enter, Tick and Exit.

e.g.

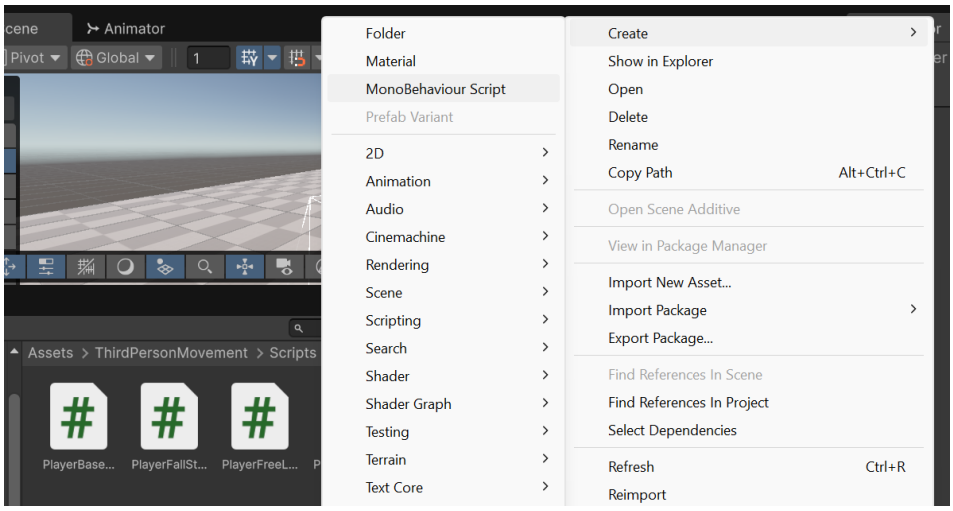
```
// Constructor - stores reference to state machine
public PlayerMyNewState(PlayerStateMachine stateMachine) :
base(stateMachine) { }
```

```
public override void Enter()
{
   // Called once when entering this state
   // Initialize state
   // Subscribe to input events
   // Start animations
}

public override void Tick(float deltaTime)
{
  // Called every frame while in this state
   // Update logic
   // Handle input
   // Check transition conditions
   Etc ...
}

public override void Exit()
{
   // Called once when exiting this state
   // Clean up
   // Unsubscribe from events
}
```

5. Creating a New State

Making new states is fairly straightforward, even if creating the logic to fill them may be more challenging. Following the steps below will create a 'PlayerLookState' where the player could be using a telescope or something similar to look around, with no movement allowed.

a. Navigate to 'ThirdPersonMovement' → 'Scripts' → 'StateMachine' → 'Player'
b. Right-click and select 'Create' → 'MonoBehaviour Script'
c. Name the script 'PlayerLookState'. Open it in your code editor.

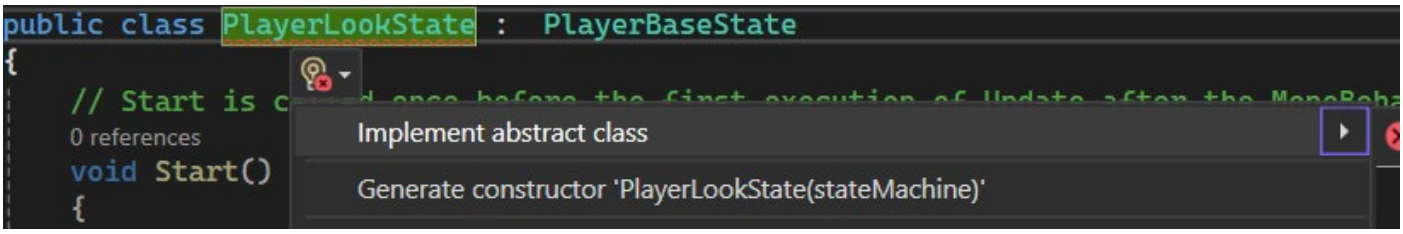


d. Each state needs to inherit from 'PlayerBaseState'. So, we'll replace the 'MonoBehaviour' inheritance. Inheriting from PlayerBaseState also inherits from MonoBehaviour.

```
using UnityEngine;

0 references
public class PlayerLookState : PlayerBaseState
{
```

You'll notice that there are a few errors at this stage. That is because we haven't yet implemented the necessary methods that each state requires, or generated the constructor. In Visual Studio Community, the quickest way to do this is to left-click ONCE on the name of the class (PlayerLookState), then click control + . (full stop / period). Then:

- 'Implement Abstract Class'
- 'Generate Constructor'



e. I like to just clean the script up at this stage, so we can:

- Remove the 'Start' and 'Update' methods (these are instead replaced by our Enter and Tick methods)
- Re-order the other methods to be in order (Enter, Tick, Exit). This is completely optional!

Your new state script should look like the below. Now we can add some logic!

```
using MistInteractive.ThirdPerson.Player;

1 reference
public class PlayerLookState :  PlayerBaseState
{
    0 references
    public PlayerLookState(PlayerStateMachine stateMachine) : base(stateMachine)
    {
    }

    2 references
    public override void Enter()
    {
        throw new System.NotImplementedException();
    }
    2 references
    public override void Tick(float deltaTime)
    {
        throw new System.NotImplementedException();
    }

    2 references
    public override void Exit()
    {
        throw new System.NotImplementedException();
    }

}
```

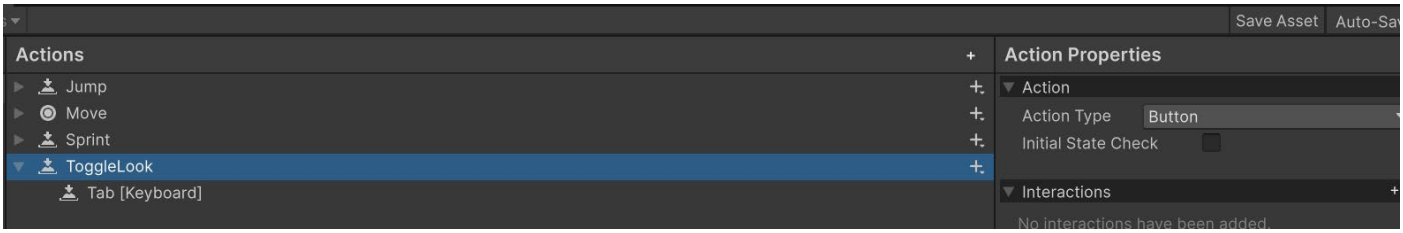
f. In the class, above the constructor, we can set up the animation logic. Like the other states, we set an animator hash value and a crossfade duration:

```
public class PlayerLookState :  PlayerBaseState
{
    private readonly int LookAnimHash = Animator.StringToHash("Look");
    private const float CrossFadeDuration = 0.1f;

    0 references
    public PlayerLookState(PlayerStateMachine stateMachine) : base(stateMachine)
    {
    }
```

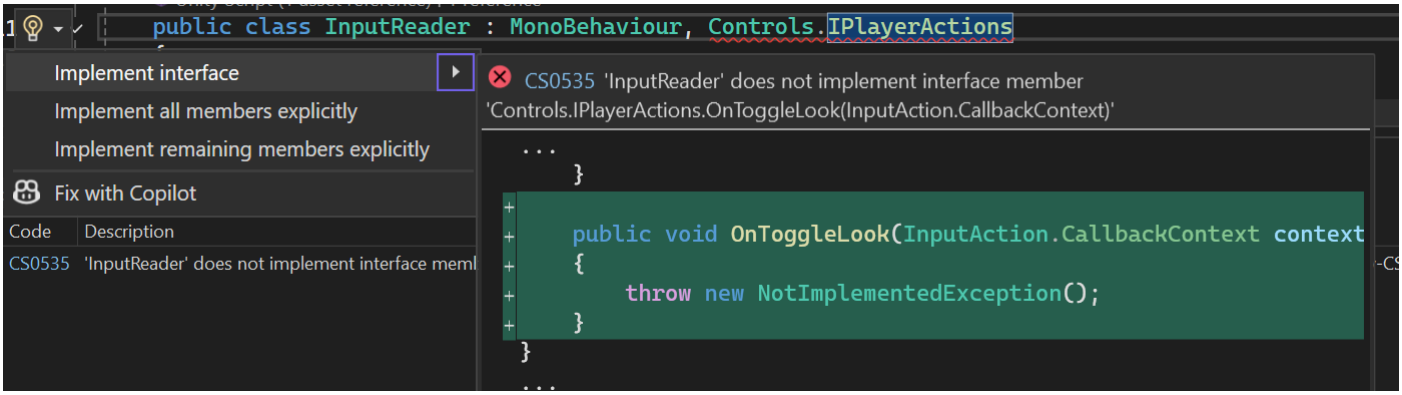
Then, in the Enter() method, we crossfade into that animation state:

```
public PlayerLookState(PlayerStateMachine stateMachine) : base(stateMachine)
{
}
2 references
public override void Enter()
{
    stateMachine.Animator.CrossFadeInFixedTime(LookAnimHash, CrossFadeDuration);
}
```

g. Before we add more code, we need to set up an input in the action map, and an event in the Input Read.
Open the ‘Controls’ action map, and add a new Action called ‘ToggleLook. It should be a button, with a key like ‘Tab’ as the input:



Press save, and then open the InputReader.cs file. Left click once on the Controls.IPlayerActions line that is complaining, followed by Control + . (full stop . period) then ‘Implement Interface’.



This will add a new method at the bottom of the class. Before we can fill it out, add a new event to fire when the ToggleLook input is pressed. Finally, complete the new method.

```
// Player events

public event Action JumpEvent;
public event Action ToggleLookEvent;
```

```
public void OnToggleLook(InputAction.CallbackContext context)
{
    if (!context.performed) return;
    ToggleLookEvent?.Invoke();
}
```

h. Back in the PlayerLookState, we can now subscribe to some input events in Enter(), which will allow the state to do things when it hears the event being fired. We also need to unsubscribe in Exit.

```
public override void Enter()
{
    stateMachine.InputReader.JumpEvent += OnJump;
    stateMachine.InputReader.ToggleLookEvent += OnToggleLook;

    stateMachine.Animator.CrossFadeInFixedTime(LookAnimHash, CrossFadeDuration);
}
```

```
public override void Exit()
{
    stateMachine.InputReader.JumpEvent -= OnJump;
    stateMachine.InputReader.ToggleLookEvent -= OnToggleLook;
}
```

Then, just create the OnJump and OnToggleLook methods that contain the logic you want to happen when those inputs are pressed when in this state.

```
2 references
private void OnToggleLook()
{
    ReturnToLocomotion();
}

2 references
private void OnJump()
{
    //Do some logic here!
}
```

i. The last step is to add an appropriate animation in the Animator window. All it requires for a simple state such as this is to create a new empty state, name it PlayerLook, and assign an animation.

   Because we are handling the animations in code within the states, there is no need for anything else in the animator!

You now have a perfectly functioning new state!

6. Script Overviews

PlayerBaseState

Abstract base class for all player-specific states. Inherits from State and adds player movement functionality.

Every player state you create should inherit from this class.

Key Features

Movement Methods:

- Move(Vector3 motion, float deltaTime) - Moves character with input + physics

- MoveNoInput(float deltaTime) - Moves with only physics (gravity, knockback)

State Transition:

- ReturnToLocomotion() - Returns to free look state

Animation Helper:

- GetNormalizedTime(string tag) - Returns 0-1 progress of tagged animations

PlayerStateMachine

The main controller for the player character. Inherits from StateMachine and stores references to all player components and settings. It is attached to your player GameObject. One per character. Configured in Inspector.

Adding Custom Settings

Add new serialized properties for your custom states:

[field: Header("Dodge Settings")]

[field: SerializeField] public float DodgeDistance { get; private set; } = 5f;

PlayerFreeLookState

The default locomotion state. Handles walking, running, sprinting, and transitioning to jump. Already implemented and working out of the box, you can reference this as an example when creating similar movement states. Most states return here via **ReturnToLocomotion()**

On Enter:

- Subscribes to jump input
- Starts "FreeLookBlendTree" animation (crossfade or instant)

During Tick:

- Reads movement input from InputReader
- Calculates camera-relative movement
- Applies walk or sprint speed
- Rotates character to face movement direction
- Updates animator blend tree parameter

On Exit:

- Unsubscribes from jump input

Constructor Parameter:

- shouldFade = true - Crossfade into animation (smooth)
- shouldFade = false - Instant animation start (when spawning/respawning)

Customisation Points:

```
// Change movement calculation
private Vector3 CalculateMovement()
{
    // Modify for different camera systems, strafing, etc.
}

// Change rotation behavior
```

```
private void FaceMovementDirection(Vector3 movement, float deltaTime)
{
    // Modify for instant rotation, mouse-look, etc.
}
```

InputReader

Bridges Unity's Input System to the state machine. Converts input callbacks into events and cached values. States subscribe to events for button presses and read properties for continuous input (movement, aim)

Input Properties (read by states)

```
Vector2 MovementValue  // WASD/Left Stick input (-1 to 1)
bool IsSprinting      // Is sprint button held?
```

Input Events (subscribe in states)

```
event Action JumpEvent  // Space/A button pressed
```

Usage in States:

```
// Read continuous input
Vector2 moveInput = stateMachine.InputReader.MovementValue;
bool isSprinting = stateMachine.InputReader.IsSprinting;

// Subscribe to button events
public override void Enter()
{
```

```
    stateMachine.InputReader.JumpEvent += OnJump;
}

public override void Exit()
{
    stateMachine.InputReader.JumpEvent -= OnJump;
}

private void OnJump()
{
    stateMachine.SwitchState(new PlayerJumpState(stateMachine));
}
```

Adding New Input Actions:

1. Add to Controls asset:
   - Create action "Dodge" (Button)
2. Add to InputReader:

```
public event Action DodgeEvent;

public void OnDodge(InputAction.CallbackContext context)
{
    if (!context.performed) return;
        DodgeEvent?.Invoke();
}
```

3. Subscribe in state:

stateMachine.InputReader.DodgeEvent += OnDodge;

ForceReceiver

Handles vertical velocity (gravity, jumping) and external horizontal forces (knockback, explosions). Works alongside CharacterController. Automatically handles gravity. Call methods when applying forces or jumping.

Key Methods:

Jump(float force)

// In PlayerJumpState.Enter():

stateMachine.ForceReceiver.Jump(stateMachine.JumpForce);

Applies instant upward force. Character rises and gravity brings them down.

AddForce(Vector3 force)

// When hit by explosion:

forceReceiver.AddForce(explosionDirection * explosionPower);

Applies horizontal force (knockback, wind, etc.). Automatically disables NavMeshAgent during knockback.

Reset()

// When respawning or teleporting:

forceReceiver.Reset();

Clears all forces and vertical velocity. Use when you need character to stop all momentum.

Common Use Cases

Jumping:

stateMachine.ForceReceiver.Jump(12f);

Hit by enemy:

Vector3 knockback = (player.position - enemy.position).normalized;

stateMachine.ForceReceiver.AddForce(knockback * 5f);

Explosion:

Vector3 direction = (player.position - explosionCenter).normalized;

float distance = Vector3.Distance(player.position, explosionCenter);

float force = 10f / distance; // Weaker further away

stateMachine.ForceReceiver.AddForce(direction * force);

stateMachine.ForceReceiver.Reset(); // Clear momentum

Typical Inspector Values:

- Drag: 0.3 (standard)
- Drag: 0.1 (slidey/icy feel)
- Drag: 0.5 (quick knockback recovery)

7. Public API Reference Lookup

PlayerBaseState (Inherited Methods i.e. Methods available in all player states)

// Movement

protected void Move(Vector3 motion, float deltaTime)

// Moves character with input + physics forces (gravity, knockback)

protected void MoveNoInput(float deltaTime)

// Moves character with only physics forces (no player input)

// State Transitions

protected void ReturnToLocomotion()

// Returns to PlayerFreeLookState (default locomotion)

// Animation

protected float GetNormalizedTime(string tagToCheck = "Attack")

// Returns 0-1 progress of animations with specified tag

StateMachine (Base Class)

public void SwitchState(State newState)

// Exits current state and enters new state

public State CurrentState { get; }

// Read-only access to current active state

PlayerStateMachine

Access via stateMachine.PropertyName in any state (e.g. stateMachine.InputReader):

// Components

public Animator Animator

public CharacterController Controller

public ForceReceiver ForceReceiver

public InputReader InputReader

public Transform MainCameraTransform

public GameObject FreeLookCamera

// Movement Settings

public float FreeLookMovementSpeed

public float FreeLookSprintMovementSpeed

public float RotationDamping

// Jump Settings

public float JumpForce

InputReader

Subscribe to events in Enter(), unsubscribe in Exit():

// Events (subscribe/unsubscribe)

public event Action JumpEvent

// Fired when jump button pressed

// Properties (read anytime)

public Vector2 MovementValue { get; }

// WASD/Left Stick input (-1 to 1 on each axis)

public bool IsSprinting { get; }

// True when sprint button held

// Control Management

public void DisableControls()

// Disable player input (for UI/cutscenes)

public void EnableControls()

// Re-enable player input

ForceReceiver

public void Jump(float jumpForce)

// Applies instant upward force

public void AddForce(Vector3 force)

// Applies horizontal force (knockback, explosions)

public void Reset()

// Clears all forces and vertical velocity

public Vector3 Movement { get; }

// Read-only: Combined force vector (used internally by Move methods)

CharacterController (via stateMachine.Controller)

void Move(Vector3 motion)

// Moves character by motion vector (don't call directly, use PlayerBaseState.Move)

bool isGrounded { get; }

// True if character is on ground

Common Usage Examples

// Transition to new state

stateMachine.SwitchState(new PlayerJumpState(stateMachine));

// Subscribe to input

stateMachine.InputReader.JumpEvent += OnJump;

// Apply jump

stateMachine.ForceReceiver.Jump(stateMachine.JumpForce);

```
// Play animation
stateMachine.Animator.CrossFadeInFixedTime(AttackHash, 0.1f);

// Check grounded
if (stateMachine.Controller.isGrounded)

// Read movement input
Vector2 input = stateMachine.InputReader.MovementValue;

// Move character
Move(movement * stateMachine.FreeLookMovementSpeed, deltaTime);

// Camera-relative direction
Vector3 forward = stateMachine.MainCameraTransform.forward;
```

8. Credits

Jammo character model from Mix and Jam

https://assetstore.unity.com/publishers/45412