Easy Character Movement 2

Kinematic Character Movement System

User Manual

Version 1.3.2

Installation

ECM2 has been developed to be easily imported into existing projects and as such it is now being distributed under the **Asset Store Physics category**, which does not include or override your project settings.

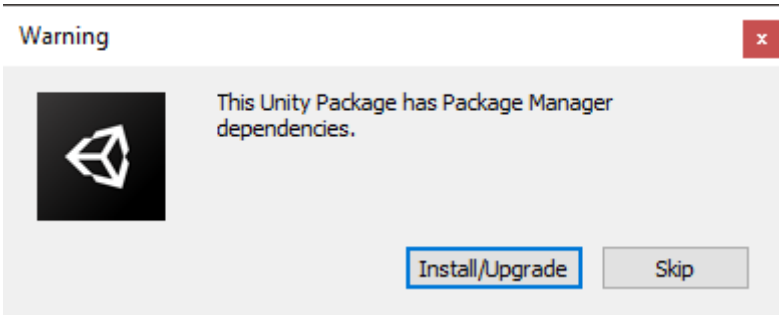
Worth note ECM2 uses **Unity's New Input system** as its default input system and while you are not forced to use it, it's required to be installed in your project.

I strongly suggest you to install the New Input System into your project BEFORE importing the ECM2 package, as this will greatly ease the ECM2 installation.

To install **Unity's new Input System**, please follow the steps from the official documentation.

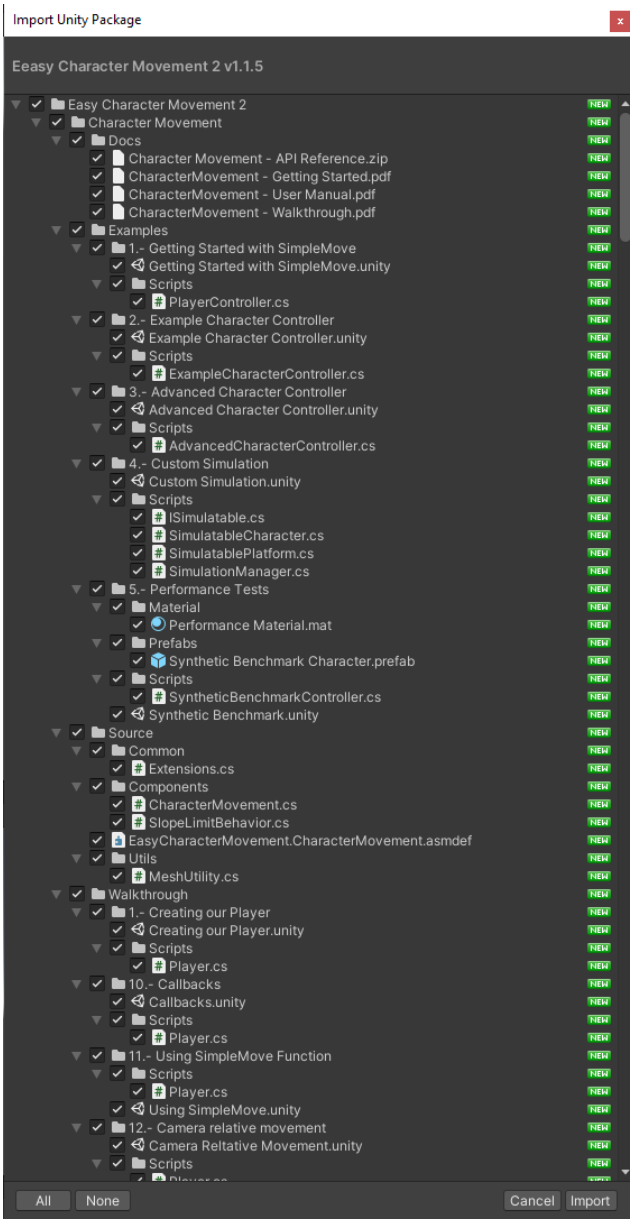
Once you have successfully installed (and enabled) the new input system in your project, proceed with ECM2 installation from the asset store:

When importing ECM2 package and depending on your project settings you may or may not be prompted with the following dialog:



Press Install/Upgrade button to continue.

The default import unity package dialog will prompt. While you can completely ignore the samples folder, I suggest you to import it in order to familiarize with it, additionally it includes 4 different template scenes for rapid prototyping.



Press the Import button to complete ECM2 installation**.**

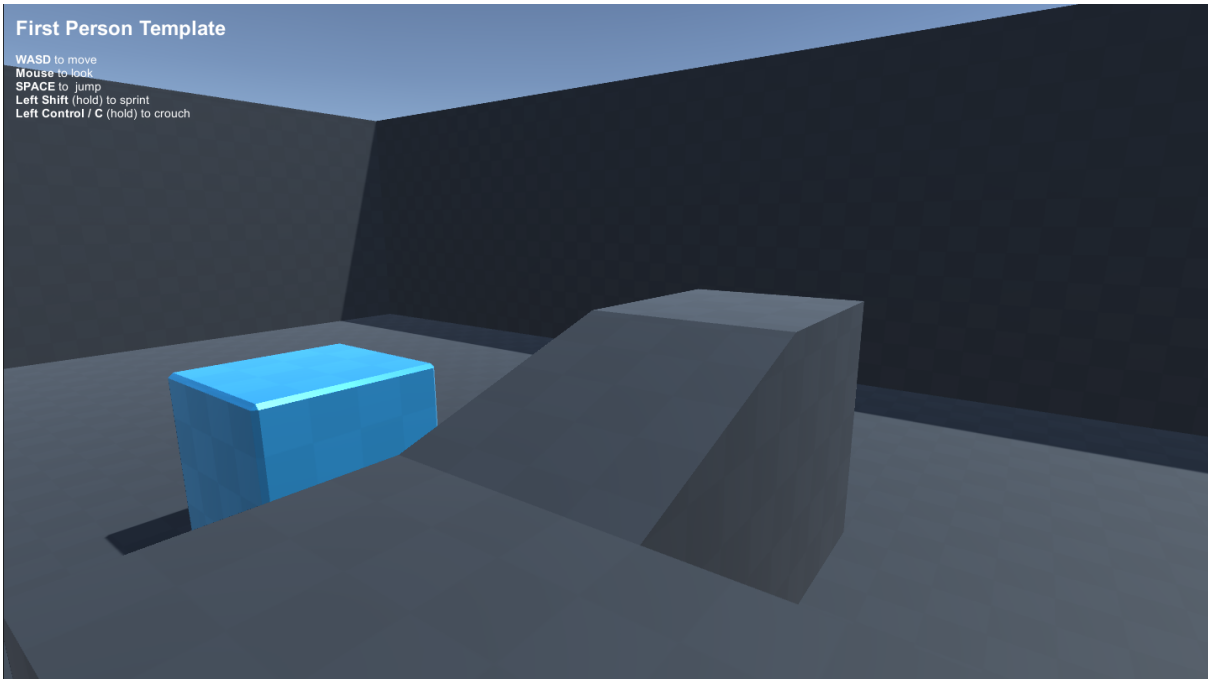
Done! You have successfully imported ECM2 into your project.

Quickstart

The easiest way to start working with ECM2 is using one of its included template scenes (`Easy Character Movement 2\Templates`) as a starting point. Each scene has been designed for faster prototyping and includes a custom `Character` derived class to add your own game mechanics.

First Person Template Scene

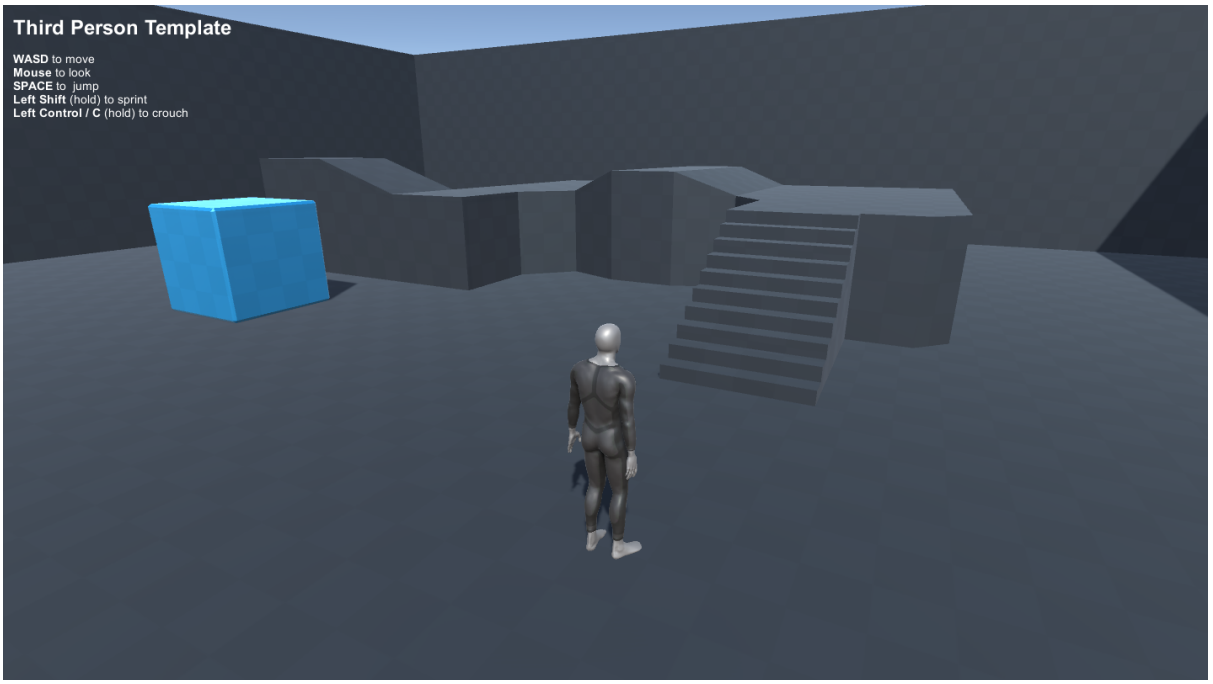
This features a player character which is viewed from first person perspective. The character can be moved around the level using a keyboard or controller. Additionally the player can look around using the mouse or controller.



The character extends the `FirstPersonCharacter` class.

Third Person Template Scene

The third-person template features a playable character where the camera is positioned behind and slightly above the character. As the character moves using a mouse, keyboard or controller, the camera follows the character, observing from an over-the-shoulder point of view. This perspective emphasizes the main character and is popular in action and adventure games.

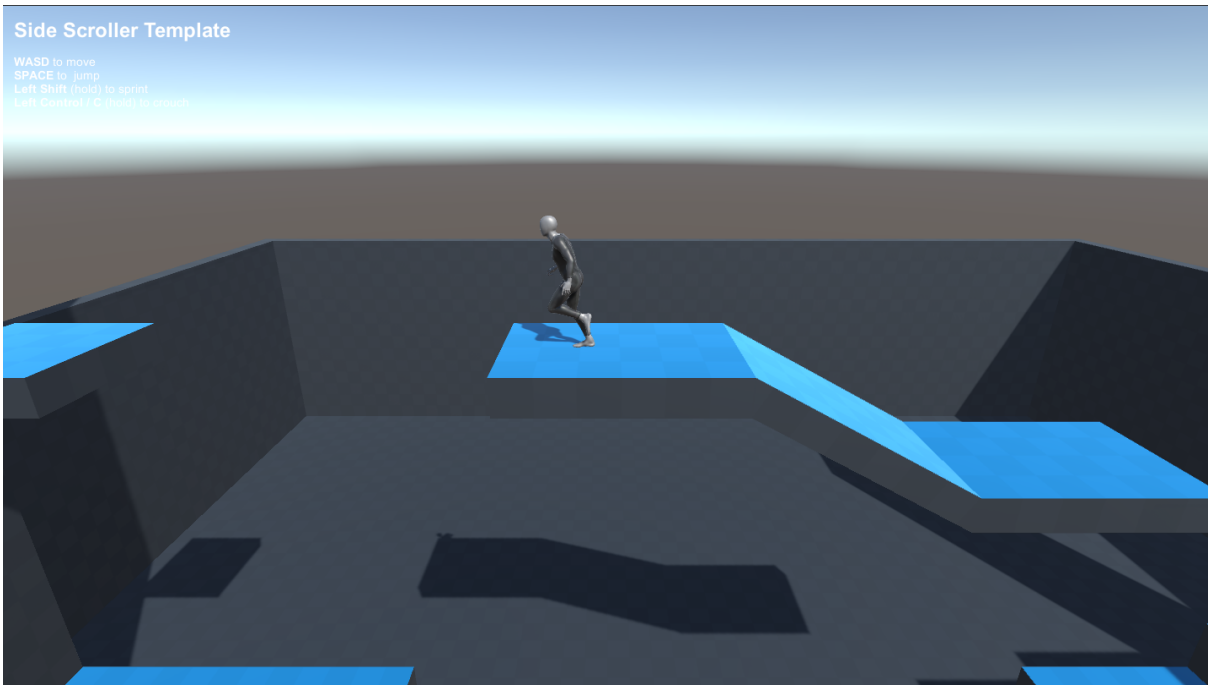


The character extends the `ThirdPersonCharacter` class.

Side Scroller Template Scene

This side scroller template features an animated model with a camera positioned at the character's side. The character can be controlled with a keyboard or controller and constrained to two dimensions: forward, backward and jumping. A number of ledges are featured in the level and the character can jump between them.

This enables the **Use Flat Base For Ground Detection** option in order to prevent the situation where a character slowly lowers off the side of a ledge (as their capsule 'balances' on the edge).



The character extends the `Character` class.

Top Down Template Scene

The top down template scene features a playable character where the camera is positioned behind and above the avatar at greater distance. The player is controlled by keyboard, controller or by using a mouse to click on the required destination and uses the navigation system to facilitate the character movement. This perspective is often used in action role playing games.

The character extends the `AgentCharacter` class.

Additionally ECM2 includes a full featured demo scene (`Easy Character Movement 2\Demo`), and over 40 fully commented examples (`Easy Character Movement 2\Examples`) to help you get the most of it.

Each example source code is fully commented and written for others in mind, in a clear and compressive style.

Preface

First of all I would like to thank you for purchasing ECM2, I sincerely appreciate your support and hope this helps you to make awesome games and projects!

If you have any comments, need some help or have a feature you would like to see added, please don't hesitate to contact me at **ogracian@gmail.com** I'll be happy to help you.

Please include the **invoice number** you received as part of your purchase when requesting support via email. Thanks!

Kind Regards,
Oscar

Overview

Easy Character Movement 2 is a **robust**, **feature-rich**, **highly customizable**, general purpose (not game-genre tied) **kinematic** character controller.

ECM2 can be used for any kind of character (player or AI controlled) and for a wide range of games like, platformer, first person, third person, adventure, point and click, and more!

ECM2 has been developed with extensibility in mind so you can rest assured it will serve you as a robust platform to build your game or add your unique game mechanics on top of it.

ECM2 includes 40+ examples, ranging from using the new input system custom actions, `PlayerInput` component, first person, third person, target lock, twin-sick movement, dash, slide, ladders, **Cinemachine**, and many more! Extensive documentation, and fully commented readable source code.

What's new?

Starting from version 1.1.0, ECM2 now includes the new **Character Movement** component, a **robust**, **high-performance**, and **feature-rich kinematic** character controller.

It directly replaces **Unity's Character Controller** offering a similar workflow (*Move* function) but with many features and advantages over it.

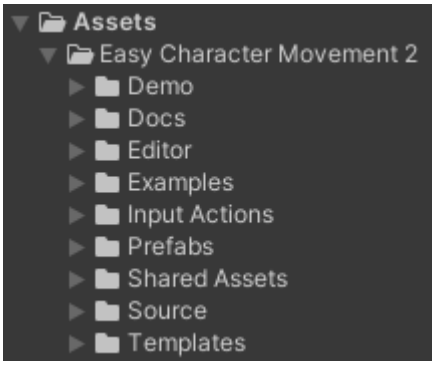
A strong separation has been made between the **CharacterMovement** component and the **Character** class, so you can make use of the **CharacterMovement** component by itself, and include new examples of this working path.

Features

- Fully **Kinematic** character controller.
- Work the Unity's way, simply call its *Move* method, just like Unity' **Character Controller.**
- **Robust ground detection** based on **real** surface normal**.**
- **Slope Limit Override** lets you define walkable areas per-face or even per-object.
- **Configurable walkable area** (*perchOffset*) lets you define the character's 'feet' radius.
- **Capsule**-based with **Flat Base for Ground Checks** option**.** This avoids the situation where characters slowly lower off the side of a ledge.
- **Ground constraint,** prevent the character's being launched off ramps.
- Configurable **Plane Constraint** so movement along the locked axis is not possible.
- **Auto-stepping** preserving character's momentum (e.g. its stride).
- **User-defined gravity** and **up vector** for **mario galaxy like** effects.
- **Physics** interactions.
- **Character vs Character** interactions.

- **First-class, transparent** (without further steps needed by you) **platforms** support, be it animated, scripted, or physics based.
- Collision and Grounding **events**.
- **Network friendly**. Full control over the update and simulation timestep.
- **Flexible** and **customizable behavior** through callbacks.
- **Fully configurable friction based movement including water buoyancy**!
- Uses **Unity New Input System** as its default input (but not limited to).
- **Different movement modes** and states, like *walking*, *falling*, *unlimited variable height jump*, *crouching*, *sprinting*, *flying*, *swimming*, etc.
- **First person**, **Third person**, **Agent** "base" characters.
- **Full control** over how a character is affected (or not) by external forces, platform velocity, platform movement and / or rotation, push other rigidbodies and or characters.
- Character actions **events**.
- Extensive and **configurable collision filtering**.
- **Different template scenes** (*First person*, *Third person*, *Side scroller*, *Top down*) for faster startup.
- **30+ examples** including custom input actions, cinemachine integration, first person, third person, character events, bouncers, fly, swim, dash, slide, ladders, and much more!
- **Developed with extensibility in mind**, can be used as it is, extend one of its "base" characters or simply take control of them using controllers such as other scripts or visual scripting (e.g. *Bolt*).
- **Physics Volumes**, to easily handle water, falling zones, etc. with configurable properties.
- **Root motion** support.
- **Simple** to use but **complete API**.
- Easy **integration into existing projects**.
- **Fully commented C# source code**. Clear, readable and easy to extend.
- Garbage-Collector friendly, no GC allocations.
- And **much more!**

Package Contents



Demo

Contains the ECM2 full-featured demo scene and related assets.

Docs

It contains the ECM2 user manual.

Editor

Unity editor related source code.

Examples

Here you can find over 25 fully-commented examples.

Input Actions

Includes the ECM2 default input actions for Character, First person, Third person and Agent characters.

Prefabs

This contains a set of character's prefabs ready to be used.

Shared Assets

Includes all the assets used by the demo, examples and template scenes.

Source

Contains ECM2 full source code.

Templates

This contains the different template scenes used for faster startup.

Introduction

Easy Character Movement 2 (ECM2 for short) is a set of components that provides a robust movement system with common movement modes for humanoid characters (but not limited to), including walking, falling, swimming, and flying.

The main goal of ECM2 is to make it easy for you to move your game characters in a 3D or 2D (actually 2.5D as it is still 3D based) world and let you build your game mechanics on top of it, because in the end, no one knows your game better than you!

The *Character Movement* component is a robust and feature-rich fully kinematic character controller and the central part of the ECM2 package.

It has been developed as a direct replacement for Unity's *Character Controller* and follows its same working methodology, where your character will only move when you call its *Move* function. It will then carry out the movement but be constrained by collisions.

Built on top of the *Character Movement* component, it's the *Character* class. The *Character* class is the base class for all avatars that can be controlled by players or AI.

A *Character* is the physical representation of a player or AI entity within the world. This not only means that the *Character* determines what the player or AI entity looks like visually (eg: your model), but also how it interacts with the world in terms of collisions and other physical interactions.

Character Components

A `Character` is comprised of the following components:

- Transform
- Rigidbody
- CapsuleCollider
- `CharacterMovement` component
- `Character` class
- `RootMotionController`. This is optional and only required if you would like to make use of root motion.

A `Character` is similar to the `BaseCharacterController` of the previous Easy Character Movement version, however in ECM2 a clear separation between a `Character` and the *Character Controller* (e.g. `CharacterMovement` component*)* has been made, as now you can use a default `Character` as it is without having to create a custom `Character` derived class.

This has been accomplished thanks to the Unity new input systems, which allows to easily modify its input bindings, and in part to its new design which makes it easier **to take full control** of a `Character`, be it locally (eg: `Character` derived class) or externally (e.g. a `Controller`) through it input commands, such as `SetMovementDirection`, `Jump`, `StopJumping`, `Crouch`, `StopCrouching`, etc.

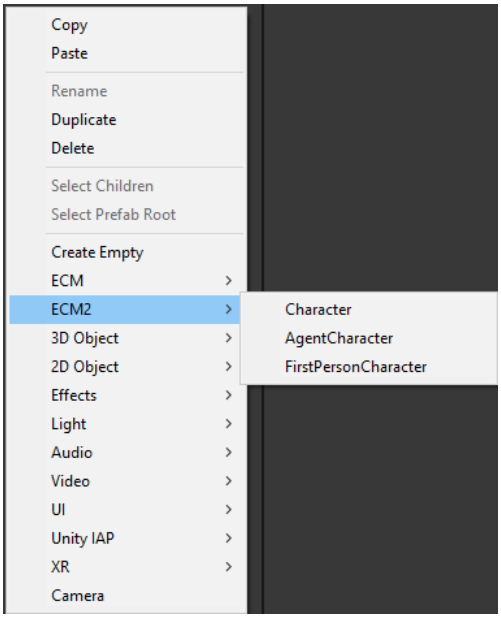
A `Controller` (while not a concrete representation exists in ECM2) is the interface between a character and the human player (or AI) controlling it, be it in the form of a c# script or a visual scripting system like Bolt.

One thing to consider when setting up your `Controller` is what functionality should be in the `Controller`, and what should be in your `Character` derived class . It is possible to handle all input in your `Character`, especially for less complex cases. However, if you have more complex needs, like multiple players on one game client, or the ability to change characters dynamically at runtime, it might be better to handle input in the `Controller`. In this case, the `Controller` decides what to do and then issues commands to the `Character` (e.g. "crouch", "jump").

Also, in some cases, putting input handling or other functionality into the `Controller` is necessary. The `Controller` persists throughout the game, while the `Character` can be transient. For example, in deathmatch style gameplay, you may die and respawn, so you would get a new `Character` but your `Controller` would be the same. In this example, if you kept your score on your `Character`, the score would reset, but if you kept your score on your `Controller`, it would not.

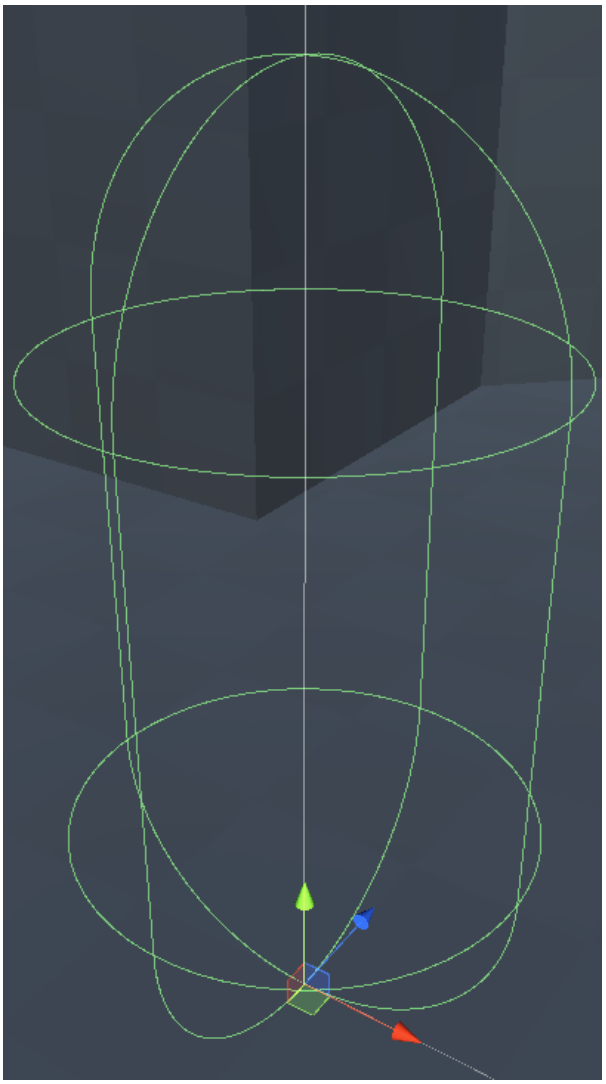
How do I create a `Character`?

1. Right-click on the hierarchy window to open the creation dialog.

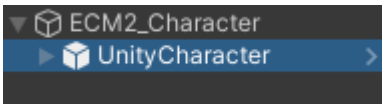


2. Select the type of `Character` you would like to create.

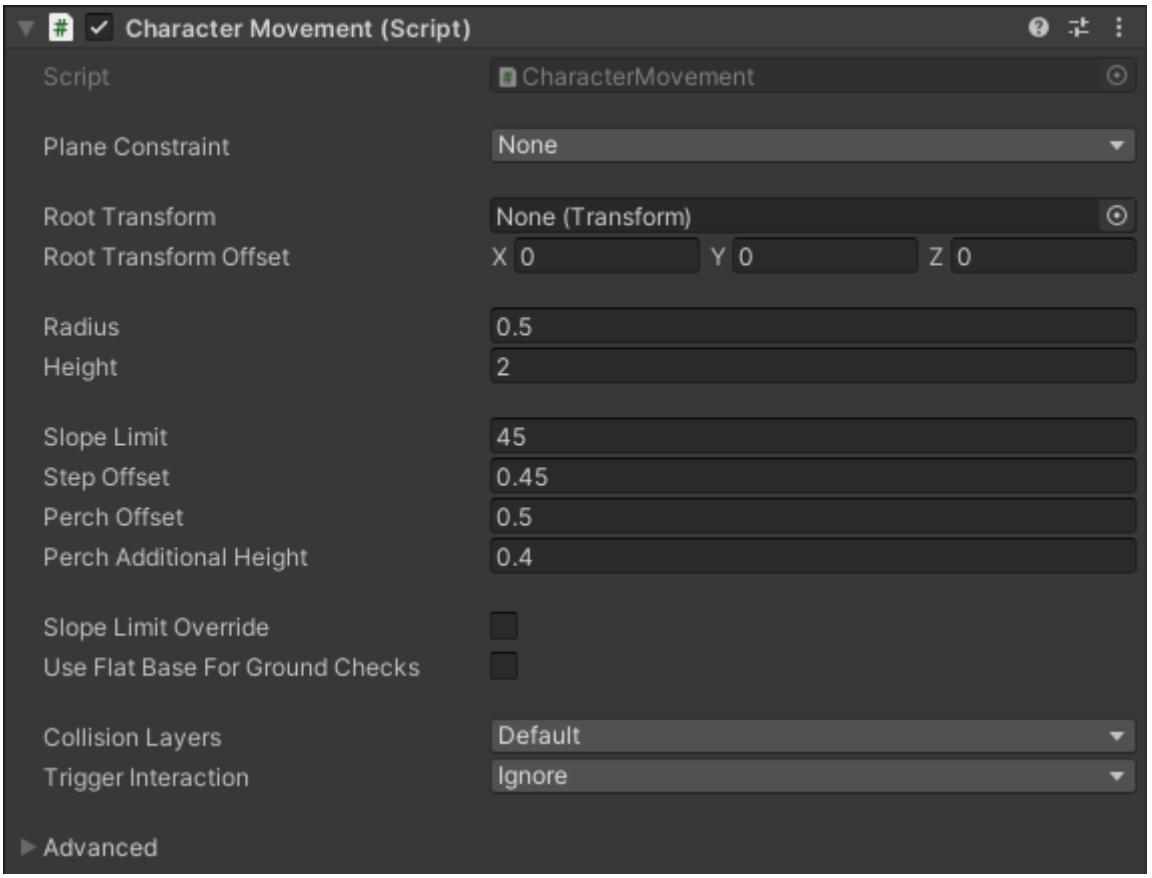
It will create an empty character (no visual representation) named `ECM2_Character`, `ECM2_AgentCharacter` or `ECM2_FirstPersonCharacter` based on your selection. Make sure its origin is at 0,0,0, as this will save you troubles when parenting your character's visual representation (e.g. your model).



3. Parent your character model (character's visual representation) to this newly created game object.



4. In the `CharacterMovement` component, adjust your **Capsule Radius** and the **Capsule Height** values (this will automatically configure your character's capsule collider) to better fit your character's model.



5. Done! Feel free to tweak the character's properties to match your game needs.

Base Characters

`Character` class

The `Character` is a full-feature highly customizable class, and the base for all your game avatars (player or AI controlled). ECM2 already includes some custom `Character` derived classes, customized for different game-genres, like **First Person**, **Third Person**, **Top Down** (e.g. `NavMeshAgent`), etc. You can use any of these "base" characters as your parent class (e.g. Creating a derived `Character` class) or to be used by your `Controller`.

`ThirdPersonCharacter` class

The `ThirdPersonCharacter` is a customized `Character` derived class, for games where the camera is positioned behind and slightly above the character. As the character moves using a mouse, keyboard or controller, the camera follows the character, observing from an over-the-shoulder point of view. This perspective emphasizes the main character and is popular in action and adventure games.

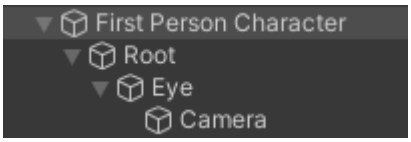
This makes use of the `ThirdPersonCameraController` class.

`FirstPersonCharacter` class

The `FirstPersonCharacter` is a customized `Character` derived class and features a player character which is viewed from first person perspective. The character can be moved around the level using a keyboard or controller. Additionally the player can look around using the mouse or controller.

This requires a predefined hierarchy, or a set of pivots, to handle the character and camera (look around) rotations. This hierarchy is similar to those found in `Cinemachine` cameras and allows great flexibility and smoother rotation.

First Person Character Hierarchy



Root

This (as its name suggests) acts as our first person character's root pivot, this handles the Yaw-Rotation (character's rotation remains unaltered all the time), this is necessary to offer a smoother rotation.

Eye

This is the camera's parent and defines the camera's position WRT character. This handles the Pitch-Rotation.

Camera

The physical `Camera` e.g. The device that captures and displays the world to the player.

The `FirstPersonCharacter` makes use of the `CharacterLook` component. This is mostly a "data-only" component used to configure the "look" behavior.

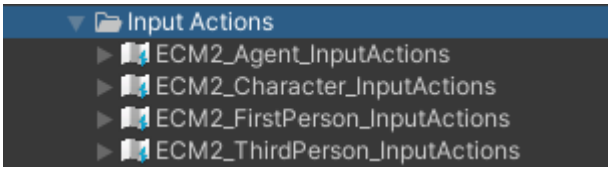
`AgentCharacter` class

The `AgentCharacter` is a customized `Character` derived class adding navigation and pathfinding capabilities (`NavMesh`, `NavMeshAgent`). This allows you to create characters that can intelligently move around the game world, using navigation meshes (player or AI controlled).

`Character` Input

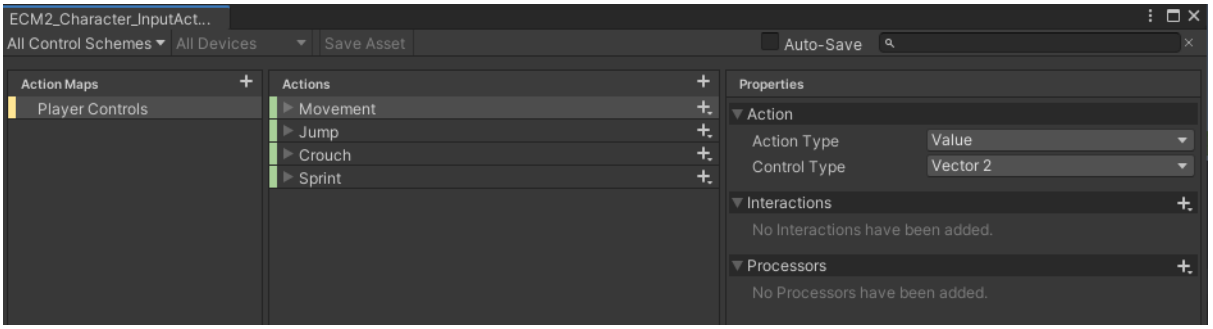
ECM2 by default utilizes Unity's new input system and includes a set of *Input Action Assets* customized for each of its "Base" characters (*Character*, *ThirdPersonCharacter*, *FirstPersonCharacter* and *AgentCharacter*). When an I*nput Action Asset* is assigned to a character's *input actions* property, it will handle the input for you within the `Character` class (i.e. `HandleInput` method), on the other hand, if you leave the character's *input actions* property empty, it won't process any input related code so you are free to externally take control of your character.

Each "Base" character includes its corresponding *Input Action Asset*.



Note: To extend a character's **input actions asset**, it is recommended to duplicate any of included **input action assets**, and use your copy to add your custom `InputAction`, this way you won't lose any modification in case of ECM2 updates.

`Character` Input Actions Asset



The `Character` class includes code to use its default **Input Actions**, this includes a definition for each `InputAction` and its corresponding event handler.

```
protected InputAction movementInputAction { get; set; }

protected InputAction jumpInputAction { get; set; }

protected InputAction crouchInputAction { get; set; }

protected InputAction sprintInputAction { get; set; }
```

And its corresponding input event handlers and functions:

```
protected virtual Vector2 GetMovementInput()
{
	if (movementInputAction != null)
		return movementInputAction.ReadValue<Vector2>();

	return Vector2.zero;
}

protected virtual void OnJump(InputAction.CallbackContext context)
{
	if (context.started || context.performed)
	    Jump();
	else if (context.canceled)
	    StopJumping();
}

protected virtual void OnCrouch(InputAction.CallbackContext context)
{
	if (context.started || context.performed)
	    Crouch();
	else if (context.canceled)
	    StopCrouching();
}

protected virtual void OnSprint(InputAction.CallbackContext context)
{
	if (context.started || context.performed)
	    Sprint();
	else if (context.canceled)
	    StopSprinting();
}
```

Creating a custom `Character`

To create a custom `Character` you need to create a new class (e.g. MyCharacter) and use one of the included "Base" characters (e.g. *Character*, *ThirdPersonCharacter*, *FirstPersonCharacter* and *AgentCharacter*) as your parent class, so you inherit all of its functionality and extend with your own game mechanics.

For example, to create a new `Character` derived class:

```
public class MyCharacter : Character
{
    // Add your game custom code here…
}
```

To use this newly created `MyCharacter` class, replace the `Character` class from your character's `GameObject` (in Unity Editor) with `MyCharacter` class. it will inherit all its functionality plus any new functions you add.

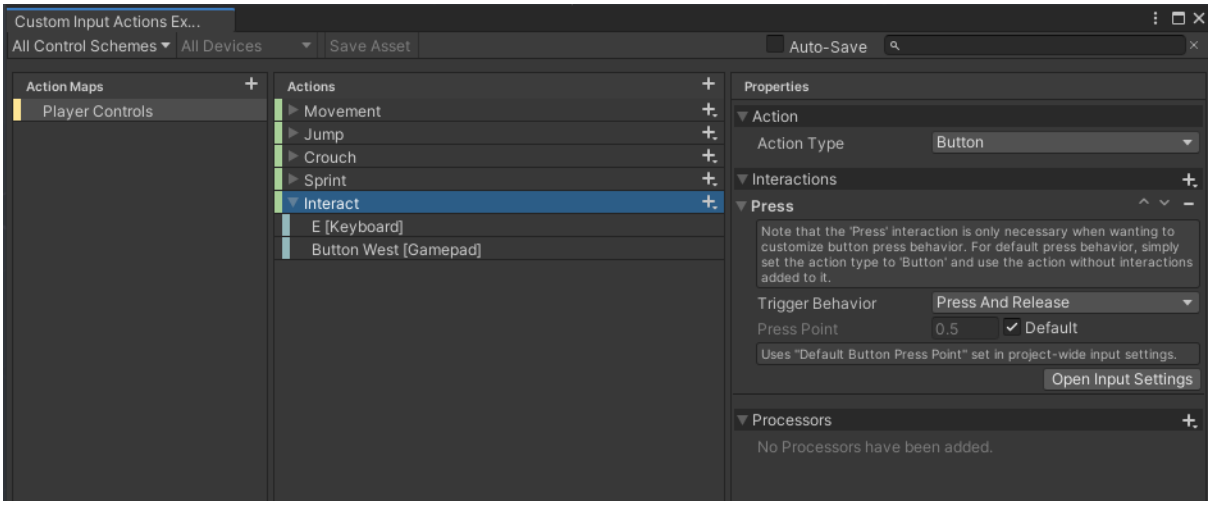
Adding a custom `InputAction` to `MyCharacter`

This shows the guidelines to follow when adding a new `InputAction` to a `Character` based class.

1. Duplicate the `Character` default **Input Actions Asset** (`ECM2_Character_InputActions`) and name the copy "**Custom Input Actions**"

2. Double-click on the **Custom Input Actions Asset** to open the **Action Editor**.

3. Add a new `InputAction` named "**Interact**" to the **Player Controls Actions Map**, as you can see here:



In order to actually be able to read the state of the new **Interact** `InputAction`, we need to make our `OnInteract` input event handler to listen for input events. ECM2 includes a predefined method used to configure your player input, `InitPlayerInput` (previously known as `SetupPlayerInput`) and its corresponding `DeinitPlayerInput` to disable input actions, unsubscribe from events, etc.

Back to your **MyCharacter** class and add the following code:

```
public sealed class MyCharacter : Character
{
    /// <summary>
    /// Interact InputAction.
    /// </summary>

    private InputAction interactInputAction { get; set; }

    /// <summary>
    /// Interact input action handler.
    /// </summary>

    private void OnInteract(InputAction.CallbackContext context)
    {
        if (context.started)
            Interact();
        else if (context.canceled)
            StopInteracting();
    }

    /// <summary>
    /// Start interaction.
    /// </summary>

    public void Interact()
    {
        Debug.Log("Player Pressed Interaction Button");
    }

    /// <summary>
    /// Stops interaction.
    /// </summary>

    public void StopInteracting()
    {
        Debug.Log("Player Released Interaction Button");
    }
```

```
    /// <summary>
    /// Setup player input actions.
    /// </summary>

    protected override void InitPlayerInput()
    {
        // Setup base input actions (eg: Movement, Jump, Sprint, Crouch)

        base.InitPlayerInput();

        // Setup Interact input action handlers

        interactInputAction = inputActions.FindAction("Interact");
        if (interactInputAction != null)
        {
            interactInputAction.started += OnInteract;
            interactInputAction.canceled += OnInteract;

            interactInputAction.Enable();
        }
    }

    /// <summary>
    /// Unsubscribe from input action events and disable input actions.
    /// </summary>

    protected override void DeinitPlayerInput()
    {
        base.DeinitPlayerInput();

        if (interactInputAction != null)
        {
            interactInputAction.started -= OnInteract;
            interactInputAction.canceled -= OnInteract;

            interactInputAction.Disable();
            interactInputAction = null;
        }
    }
}
```

We use the character `InitPlayerInput` method to make our `OnInteract` input event handler to listen for input events and Enable it, otherwise the `InputAction` won't do anything.

In a similar way, we use the `DeinitPlayerInput` to unsubscribe from input actions and disable it. This is the recommended behavior as this fully complies with the **Unity's Play Mode** feature.

You can find a fully commented working example for adding a custom input action as part of the included examples (`Easy Character Movement 2\Examples\1.- Input\1.2.- Custom Input Actions`).

Character Controller

As previously discussed, a `Character` can be "possessed" by a `Controller`, and while no concrete representation exists in ECM2, this serves as the interface between a character and the human player (or AI) controlling it, be it in the form of a c# script or a visual scripting system.

In ECM2 you can find several controller examples, being the most basic example using the `PlayerInput` component.

Please refer to the included **Character Controller Example Scene** (`Easy Character Movement 2\Examples\1.- Input\3.- PlayerInput Character Controller`) for a fully working example.

Character Movement Modes

In ECM2 the `Character` class introduces the concept of **Movement Modes**, e.g. **Walking**, **Falling**, **Flying**, **Swimming** and **Custom**. Each movement mode has a set of predefined rules and properties to determine how the character is moved through the world.

This, while somewhat related to `Character` states (e.g. Logical states like **Jumping**, **Attacking**, **Dying**, etc.), should not be confused, as the **sole purpose of a** `MovementMode` **is to determine how the character should be moved through the world**.

For example, the **Flying Movement Mode**, while it suggests the character is in flying logical state, what the flying movement mode does is determine how is the character being moved now, e.g: **Moving through the air ignoring the effects of gravity, the character is unconstrained to ground and its vertical velocity is preserved.**

The Character `MovementMode`

- **None**: This disables the character's movement. Internally will make the character's rigidbody kinematic, preventing it being affected by any force. This replaces the pause from the previous ECM version.

- **Walking**: Moving along a walkable surface ignoring the effects of gravity, but affected by friction. The character is constrained to ground and the input vertical velocity is ignored.

- **Falling**: This is when the character is falling on air or sliding-off a non-walkable surface affected by gravity. The character is constrained to ground and the input vertical velocity is ignored.

- **Flying**: Moving through the air ignoring the effects of gravity, the character is unconstrained to ground and its vertical velocity is preserved, e.g. Fly up or down.

- **Swimming**: Moving through a fluid volume (e.g. Water), under the effects of gravity and buoyancy, the character is unconstrained to ground and its vertical velocity is preserved e.g. Swim up or down.

- **Custom**: User-defined custom movement mode, including many possible sub-modes.

A `Character` defaults to **Walking** movement mode (can be changed if desired) .

You can change the character's movement mode using its `SetMovementMode` function, this will call `OnMovementModeChanged` function and trigger the `MovementModeChanged` event. `The OnMovementModeChanged` method does special handling for starting certain modes, e.g. Enable / disable ground constraint, StopJump, reset jump count, etc.

Worth note that **Walking** and **Falling** modes are automatically managed, as those respond to the character's grounding status. While a character is constrained to ground, if the character is on walkable ground it will enable the **Walking** movement mode, and if not on ground, or is on non-walkable ground, will change to **Falling** movement mode.

By other hand, for the **Flying** movement mode, you must explicitly enable and disable it as needed. To leave **Flying** state is perfectly safe, to use the **Falling** movement mode to exit, as this will automatically transition to **Walking**. For example:

```
// Enter flying mode

SetMovementMode(MovementMode.Flying);

...

// Exits flying mode

SetMovementMode(MovementMode.Falling);
```

While you can manually enable / disable the **Swimming** movement mode, this is automatically managed when using **Physics Volumes**, it will be enabled when a character enters a `Water PhysicsVolume`, and disabled when exits a `Water PhysicsVolume`.

Worth note, by default **a character is not allowed to jump while in Swimming** movement mode, however as with many of the ECM2 functionality, it can easily be modified to fit your game needs.

The **Demo** scene and **Swimming** example scene shows how to perform a jump while swimming if needed.

`PhysicsVolume` component

A `PhysicsVolume` it's a helper component used to define a three-dimensional zone to alter the character's behavior within levels.

Physics Volumes are zones in which the physical setup affects characters. A common use for them is for the creation of watery environments in which the player needs to swim.

When a `Character` enters a new `PhysicsVolume` (e.g. its center is inside the `PhysicsVolume`) it will call its `OnPhysicsVolumeChanged` method and will trigger the `PhysicsVolumeChanged` event.

You can find a working example of nested *Physics Volumes* in the demo scene.

Moving a Character

To move a character you use its `SetMovementDirection` function to feed a *direction* vector in world space, be it from a derived `Character` class, external script or visual solution.

E.g. To set the desired input direction to move our character:

```
// Poll movement InputAction

var movementInput = GetMovementInput();

// Add movement input relative to world axis

Vector3 movementDirection = Vector3.zero;

movementDirection += Vector3.right * movementInput.x;
movementDirection += Vector3.forward * movementInput.y;

SetMovementDirection(movementDirection);
```

Movement relative to a Camera

By default, when you assign a `Camera` to a `Character` class, the character's movement will be relative to this camera. This is primarily for player controlled characters.

To manually move a character relative to a camera, you could use the include helper `.relativeTo` extension, this will transform the vector to be relative to the given `Transform`.

```
// Poll movement InputAction

var movementInput = GetMovementInput();

// Add movement input relative to world axis

Vector3 movementDirection = Vector3.zero;

movementDirection += Vector3.right * movementInput.x;
movementDirection += Vector3.forward * movementInput.y;

// Make movementDirection vector relative to camera's transform

movementDirection = movementDirection.relativeTo(cameraTransform);

SetMovementDirection(movementDirection);
```

Input Action Commands

In addition to the `SetMovementDirection` function, a `Character` (and its derived classes), includes a set of predefined **input action commands**; these functions **issue the character to perform an action**, for example, **Jump**, **StopJumping**, **Crouch**, **StopCrouching**, etc.

Typically these are used in response to an input event such as on 'down', on 'up', etc.

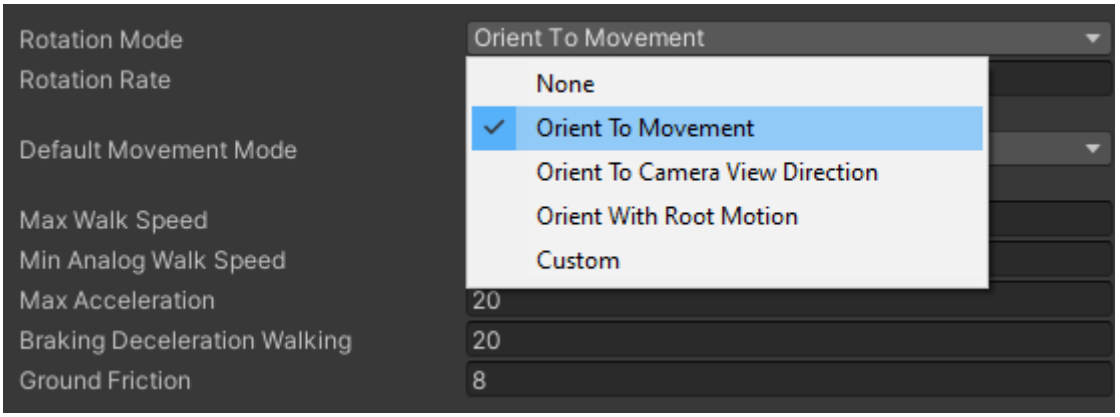
- `Jump`. Start a Jump.
- `StopJumping`. Stops the character from jumping.

- `Sprint`. Request the character to start sprinting.
- `StopSprinting`. Stops the character from sprinting.

- `Crouch`. Request the character to start crouching.
- `StopCrouching`. Stops the character from crouching.

Character Rotation

A `Character`, while it can be rotated as desired (e.g. Modifying its *rotation*), it already includes functionality to handle common rotation / orientation modes.



Character `RotationMode`

The included rotation modes are:

- **None**: Disables character's rotation.

- **OrientToMovement**: Rotates the character towards the given input movement direction vector, using `rotationRate` as the rate of rotation change.

- **OrientToCameraViewDirection**: Rotates the character towards the camera's current view direction, using `rotationRate` as the rate of rotation change.

- **OrientWithRootMotion**: Append root motion rotation to character's current rotation.

- **Custom:** Lets you add a user-defined rotation mode.

You can set your character's default **rotation mode** from the editor or change it at run-time as needed using the `SetRotationMode` function; on the other hand, use the `GetRotationMode()` function to query the character's current **rotation mode**.

Based on its current rotation mode, the character's rotation will be modified in its `UpdateRotation` method. Alternatively, you can extend this method to perform custom rotations or completely replace the default modes.

Additionally you can use the following functions, to directly modify the character's rotation as needed:

- `SetRotation.`
- `SetYaw.`
- `RotateTowards.`
- `AddYawInput.`
- `AddPitchInput.`
- `AddRollInput.`

Character Position Right Up and Forward

A `Character` class includes a set of methods used to access the character's position and orientation information, such as `GetPosition()`, `GetRightVector()`, `GetUpVector()` and `GetForwardVector()`.

Adding Forces to a Character

As part of the changes introduced in **ECM2**, being now a fully kinematic character controller, it's not possible to add forces directly to the rigidbody, however the CharacterMovement and Character includes new methods to let you add forces and impulses in a similar way.

The forces added are carried and are applied the next *Character Movement Move* call.

These `AddForce` and `AddExplosionForce` functions behave just like Unity's counterparts.

`LaunchCharacter` function

`LaunchCharacter`, is a useful helper function used to explicitly modify the character's velocity component wise.

```
public virtual void LaunchCharacter(Vector3 launchVelocity, bool
overrideVerticalVelocity = false, bool overrideLateralVelocity = false)
```

This, unlike `AddForce`, allows to directly modify (adding or replacing) the character's velocity components (*horizontal*, and/or *vertical*). This offers great flexibility and should be preferred when adding impulses (e.g. An instant velocity-change) to the character.

This can use it to implement **bouncers**, make the character **dash**, **slides**, **speed up**, etc.

Ground Constraint

In order to prevent a character from being launched off the ground while walking / sprinting at greater speeds, ECM2 (like ECM before) implements a **Constraint to Ground** feature.

This will make sure the character maintains its walkable ground at all times. However, as a result, we should let the system know when the character is allowed to leave the ground, e.g. remove this ground constraint such as when *flying*, *swimming*, *jumping* etc. otherwise the character will be pulled back to the ground.

To accomplish this, we use the `CharacterMovement constrainToGround` property, to explicitly enable or disable this or the `PauseGroundConstraint` to temporarily (N seconds) disable the ground constraint.

For example, to implement a bouncer:

```
// Pause the ground constraint to let the character leave the ground

PauseGroundConstraint();

// Apply a vertical impulse to the character

LaunchCharacter(Vector3.up * 20.0f);

.
.
.

// Move character

characterMovement.Move();
```

Animating your Character

ECM2 does not use animation or requires it to be animated in a special way, so you are free to animate your characters using 'plain Unity code' or your preferred method.

The `Character` class includes an animation related method `Animate`, you can use this method to add your **Animator** related code, **however this is not mandatory and can be safely ignored if desired**.

It is completely normal (**and actually recommended**) to handle your character's animation in an external animation controller (in the way you prefer) and just query the character's state or **subscribe to its many events** to keep your animation perfectly in sync with the character's state, e.g is it grounded? is it falling? Is it jumping? etc.

Querying Character State

The `Character` class offers a wide range of methods, events and delegates you can use to read the character's information, such as: `GetPosition()`, `GetRotation()`, `GetVelocity()`, `IsWalking()`, `isFalling()`, `IsOnWalkableGround()`, `etc` . This provides useful information about the character's current state and can be used to keep your animation fully in sync among other things.

In addition to the information provided by the `Character` class, you can get further information through the `CharacterMovement` component, like ground related information, get and set capsule collider dimension, access to collision detection functions, or even compute a new set of ground related information (eg: is walkable, is a step, distance to ground, etc).

* Please refer to the included *Character Movement* user manual for further information.

Using Root Motion

Root motion means the motion is built right into the animation and it's the animation that determines how far something moves rather than code.

In order to make use of root motion in a `Character` derived class, you must add the `RootMotionController` component to your model's `GameObject`, as this `RootMotionController` is responsible to provide the animation's *velocity*, *rotation*, etc to the `Character`.

Additionally, you must enable the `useRootMotion` character's property in order to enable or disable the use of root motion based movement (you can enable or disable at any time).

Once a character is being moved with root motion, **the animation takes full control over the character's movement**, e.g. this will replace all of our procedural movement. So

properties like **maxWalkSpeed**, **maxFallSpeed**, etc are ignored as the character's is being driven by the animation.

So what are the benefits of using root motion?

It keeps the collision capsule anchored to the model where it should be, it enables you to utilize acceleration and deceleration in your animation, it eliminates foot sliding and other artifacts, and it simplifies the programming work load.

Can I use root motion on my Player-controlled character?

Yes, but instead of the joystick axis controlling how much acceleration to apply, instead it controls which animation to play and how fast to play it. Controlling your character == controlling the animation that is playing, e.g. **Want to walk faster?** Play the animation to walk faster. **Want to turn 90 degrees?** Play an animation that turns 90 degrees. etc.

Does it work with vertical root motion?

Yes, but your character needs to be in **Flying movement mode**.

Remember when I commented that a movement mode should not be confused with a logical state? Well this is a good case to make it clear.

Consider, you might need to create a special ability where a character leaps a specific height into the air, then lands with a powerful attack. Here the character's logical state should be **Attacking** (or whatever better fits your game)**,** while its **movement mode** (in order to allow vertical root motion movement) **must be Flying.**

Character Events and Handlers

The `Character` class offers a wide number of **events** and **event handlers** you can use to respond accordingly locally (e.g. `Character` derived class) or externally (e.g. `Controller`).

Events

```
/// <summary>
/// Event triggered when entering or leaving a PhysicsVolume.
/// </summary>

public event PhysicsVolumeChangedEventHandler PhysicsVolumeChanged;

/// <summary>
/// Event triggered after a MovementMode change.
/// </summary>

public event MovementModeChangedEventHandler MovementModeChanged;

/// <summary>
/// Event triggered when Characters start sprinting.
/// </summary>

public event SprintedEventHandler Sprinted;

/// <summary>
/// Event triggered when Character stops sprinting.
/// </summary>

public event StoppedSprintingEventHandler StoppedSprinting;

/// <summary>
/// Event triggered when Character enters crouching state.
/// </summary>

public event CrouchedEventHandler Crouched;

/// <summary>
/// Event triggered when character exits crouching state.
/// </summary>

public event UnCrouchedEventHandler UnCrouched;

/// <summary>
/// Event triggered when character jumps.
/// </summary>

public event JumpedEventHandler Jumped;
```

```
/// <summary>
/// Triggered when Character reaches jump apex (eg: change in vertical speed
from positive to negative).
/// Only triggered if notifyJumpApex == true.
/// </summary>

public event ReachedJumpApexEventHandler ReachedJumpApex;

/// <summary>
/// Triggered when the Character will hit walkable ground.
/// </summary>

public event WillLandEventHandler WillLand;

/// <summary>
/// Event triggered when character enter isGrounded state (isOnWalkableGround
AND isConstrainedToGround)
/// </summary>

public event LandedEventHandler Landed;
```

Event Handlers

```
protected virtual void OnCollided(ref CollisionResult collisionResult);

protected virtual void OnFoundGround(ref FindGroundResult foundGround);

protected virtual void OnWillLand();

protected virtual void OnLanded();

protected virtual void OnSprinted();

protected virtual void OnStoppedSprinting();

protected virtual void OnCrouched();

protected virtual void OnUnCrouched();

protected virtual void OnJumped();

protected virtual void OnReachedJumpApex();

protected virtual void OnPhysicsVolumeChanged(PhysicsVolume newPhysicsVolume)

protected virtual void OnMovementModeChanged(MovementMode prevMovementMode,
int prevCustomMode)
```

When creating a `Character` derived class (e.g. A custom character), while you can subscribe to its events, **it is recommended to extend its "On" methods** as this is simpler, clearer, and faster. For example:

```
// Extends Character OnLanded method.
// Called when the character hits walkable ground.

protected override void OnLanded()
{
    // Call base method implementation

    base.OnLanded();

    Debug.Log("Landed!");
}
```

However, when extending an "**On**" event handler method, **it is important to always call its method base implementation**, as this is responsible for triggering the events among other tasks.

By other hand, in order to receive the `OnReachedJumpApex` event, **you first must set** `notifyJumpApex` **property to true**, otherwise this event will not be triggered, e.g:

```
protected override void OnJumped()
{
    // Call base method implementation

    base.OnJumped();

    // Enable jump apex event notification

    notifyJumpApex = true;

    Debug.Log("Jump!");
}

protected override void OnReachedJumpApex()
{
    // Call base method implementation

    base.OnReachedJumpApex();

    Debug.Log("Reached jump apex!");
}
```

`ThirdPersonCharacter` class

This extends the `Character` class adding controls for a typical third-person movement. You should use this as your parent class when creating a third-person game.

`ThirdPersonCharacter` makes use of the `ThirdPersonCameraController` class, however ECM2 includes an additional third-person character which uses **Cinemachine** to control the third-person camera if preferred.

`FirstPersonCharacter` class

This extends the `Character` class adding controls for a typical first-person movement. You should use this as your parent class when creating a first-person game.

This adds extra functionality to the `Character` class, such as `GetEyeForwardVector()` (the camera's view direction), `GetEyeRightVector()`, etc.

`AgentCharacter` class

The `AgentCharacter` extends the `Character` class adding navigation and pathfinding capabilities (`NavMesh`, `NavMeshAgent`). This allows you to create characters that can intelligently move around the game world, using navigation meshes (player or AI controlled).

How do I move an `AgentCharacter` ?

For player controlled characters, it can be controlled by keyboard, controller or using a mouse (e.g. Click to move) and as any `Character` derived class, it inherits all the `Character` *input action commands* such as: *SetMovementDirection*, *Jump*, *StopJumping*, etc.

It adds a new function `MoveToLocation`, which receives a *position* (in world space) and will issue the character to intelligently navigate to the given point using its `NavMeshAgent`.

Platforms

ECM2 v1.1.0 improves platform management considerably, as now platform movement is handled *transparently* without any additional steps required by you.

It supports out-of-the-box any kind of animated platform be it scripted, tweened, animated or even fully dynamic rigidbody 'platforms' like vehicles, boats, airplanes, etc.

The only requirement for a platform is to be updated in the *FixedUpdate* or if animated, set its *Animator Update Mode* to *Animate Physics*.

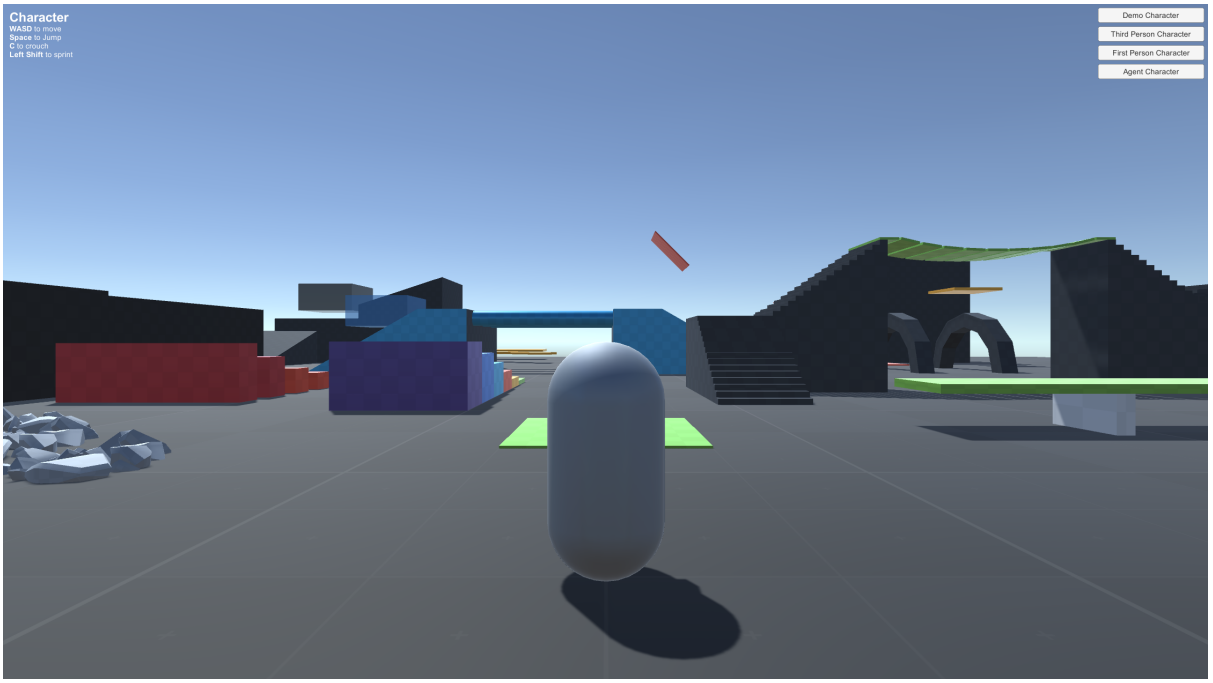
A new function to the *Character Movement* component, `SetPlatform`, lets you 'attach' the `Character` to a 'parent' platform so it moves with it independently if it is in contact with it or not.

Additionally you can use the *inpartPlatformMovement*, *inpartPlatformRotation* and *inpartPlatformVelocity* properties to determine if a `Character` is affected by a platform and how.

Where more control is needed, you can make use of the new *Character Movement* `CollisionBehaviorCallback` function to completely control a character collision.

* Please refer to the included *Character Movement* user manual for further information.

Demos



This includes 2 demo scenes featuring most of the ECM2 functionality in action.

Examples

To get familiar with ECM2 and to make the most of it, it is strongly suggested to read this document along with the corresponding examples source code. Each example includes a clean, clear and fully commented source code to help you better understand how ECM2 works and how to extend it to match your game needs.

Input

Default Input Actions

This example features a default `Character` ready to be tweaked, no custom code is used.

Custom Input Actions

This example shows the procedure to add a new **InputAction** for use with a `Character` derived class.

PlayerInput Character Controller

This example shows how to "Possess" a `Character` derived class, this is the most basic example of a `Controller`, e.g. Externally take control of a character.

This features an externally animated character (`UnityCharacterAnimator`) and makes use of the Unitys **PlayerInput** component to handle the player's **input actions** within our controller (`MyCharacterController`).

While this takes control of a `Character`, the same procedure applies for any of its derived classes like `ThirdPersonCharacter`, `AgentCharacter`, etc.

Old Input Examples

This includes several examples, to show how you can ignore Unity's new input system, and replace it with the old input system. You can follow the same approach to use any other input system as needed.

Custom Deadzone

This example shows how to implement different kinds of deadzone.

Animation

Character Animator

This example shows how to externally animate a `Character` reading its current state in order to feed the character's **Animator**.

While the `Character` class (and its derived classes) includes an animation related method `Animate` which can be used to handle your animation related code, I suggest the use of an external animation controller as this offers greater flexibility and well worth for bigger projects.

Root Motion

This features an animated character moved with **Root Motion** and orientated with **Root Motion**.

Root Motion Toggle

This shows how to toggle (enable / disable) the use of root motion based movement. In this particular example the character is moved with walking root motion only.

It makes use of the `Character` `OnMovementModeChanged` method to toggle root motion, it will enable it when the character is in walking movement mode, otherwise disables it.

`OnMovementModeChanged` its called every time a character change its current movement mode (e.g. `SetMovementMode`) this method handles movement dependent code, like enable / disable ground constraint, cancel jumps, reset jump counter, etc. and it's a great place to add your movement mode dependent code (e.g. enter / exit a movement mode).

Events

Character Events

This example shows how to create a `Character` (e.g. A custom character) derived class and extends its methods to respond to the many `Character` events.

Character Controller Events

This example shows how a character controller can make use of its controlled character events, letting the controller respond to character's events.

Platforms

Scripted Platform

This example shows a basic procedurally animated platform.

Animated Platform

This example shows a basic animated platform.

One-Way Platform

This example shows how to create a one-way platform where the platform is "invisible" to the player when jumped from below.

This shows how to make use of the character's `IgnoreCollision()` method to make the character ignore a specific collider.

Physics Platform

This example shows fully physics based objects used as platforms.

Gameplay

Third Person Controller

This example shows how to implement a third person controller. The controller "possesses" a `Character` and makes use of a `ThirdPersonCamera`.

Target Lock

This example shows extending a `Character` to implement a target lock mechanic. While locking, the character will automatically look at its current target.

Twin-Tick Movement

This example shows extending a `Character` to implement a twin-stick movement as found in many top-view shooter games. Here the character's movement and its rotation (aiming) are decoupled allowing it to move and fire in different directions.

Bouncer

This example shows how to make use of the `Character` `LaunchCharacter` function to implement a bouncer.

The `LaunchCharacter` offers an easy and convenient way to directly modify character's velocity (additive or override) component-wise (horizontal / vertical).

Dash

This example extends a `Character` class adding a **dashing state**. This is a complete example showing how to add a new "action" to a character, in this case a dashing action. The guidelines used here should help you when adding extra functionality to an ECM2 character.

When adding extra functionality to a `Character`, you should use its `Move` method (override it), and use this to handle your new state (e.g. Dashing). Additionally, while you can use the same `Move` method to add a **new custom mode** (is perfectly fine), The `Character` class includes a dedicated method to handle custom movement modes `OnCustomMovementMode`.

Why not make Dashing a movement mode ?

A movement mode directly modifies the character's velocity and how it is updated. For example, when **Walking**, it ignores any vertical velocity, automatically changes to *falling* movement mode, etc. As you can see if we implement **Dashing** as a movement mode, we would need to replicate many of the already implemented functionality just to make a dash.

Ideally a movement mode should be an action not possible using one of the available movements modes.

Slide

This example shows how to extend a `Character` to implement a slide mechanic similar to those found in FPS games. This works as follows:
To slide, a character must be sprinting and while sprinting, press the crouch input. By other hand, a sliding will end if:

- `Character` is *Falling*.
- On a *non-walkable* hit (e.g. A wall).
- The character's *velocity* is less than `maxWalkSpeedCrouched`.
- The character un-crouches.

As you'll see, this example is really close to the **Dashing** implementation as these examples follow the same guidelines used in the `Character` class and while not mandatory, **I suggest you follow it when adding extra functionality** to a `Character`.

In this particular example, we make use of the *Crouch / Uncrouch* event handlers to enter / exit the **Sliding** state, so no additional input is necessary.

These guidelines, while not mandatory, will help to make your code readability easier, and code cleaner. For example, to add a *Sliding* state, I'll use the following:

`IsSliding()`

Is the character currently sliding ?

`CanSlide()`

Determines if the character is able to slide.

`Slide()`

Starts the sliding, typically called in response to an input event (e.g. On button down).

`StopSliding()`

Stops the character from sliding, typically called in response to an input event (e.g. On button up).

`Sliding()`

Handle sliding state.

`Move()`

Override the `Move` method to add your state handler, `Sliding`.

Fly

While the `Character` class by default includes a **Flying movement mode**, it is the responsibility of the user to decide how to move the character while flying; remember, a movement mode only sets the rules to follow when moving a character. This example implements a flying mechanic making use of the character's **flying movement mode**, when flying, the character can move along its view direction and vertically up.

Swim

While the `Character` class by default includes a **Swimming movement mode**, it is the responsibility of the user to decide how to move the character while swimming (e.g. swim up, swim down, etc); remember, a movement mode only sets the rules to follow when moving a character. This example implements a swimming mechanic making use of the character's **flying movement mode** allowing the character to swim up and or along its view direction.

Slope Speed Modifier

This example shows how to modify the character's speed based on its current slope angle, so it slows down when going up a slope and speeds up when going down slope.

To accomplish this, we make use of an *AnimationCurve* to define our slope speed modifier, here the curve's x-axis defines the slope angle, while the y-axis defines the speed modifier. A negative angle means going down a slope.

The character will use its signed slope angle to look up into the slope speed curve and get its current speed modifier and update its `maxWalkSpeed` accordingly.

Fall Damage

This example shows how to make use of the character's event handlers to keep track of its last grounded position, later when it lands, we simply compute the fallen distance (the distance between the last grounded position and the character's current position. You can use this fallen distance in order to see if your character should suffer damage or not, based on your game rules.

Teleporter

This example shows how to correctly teleport a character.

Change Gravity Direction

This example shows how to extend a `Character` to change its gravity direction at run-time and update its rotation accordingly.

Here, we will toggle the gravity direction along up and down, this will also rotate the character to match the new gravity direction. When a character is rotated, all of its movement is relative to the character's up vector, this allows the character to walk on walls, ceilings etc.

In order to toggle the gravity direction, press the E key when the character is jumping or falling (eg: not grounded). We override the `UpdateRotation` method in order to rotate the character against the new gravity direction.

Planet Walk

This example extends a `ThirdPersonCharacter` and `ThirdPersonCamera` to implement a "Planet Walk" movement similar to Mario Galaxy, so the character can walk all around the planet.

This will update the character and camera rotation so it follows the planet curvature.

Ladders

This is an **important example** as this **shows how to implement a custom movement mode along with sub-states**.

This example extends a `Character` to add ladder mechanics, in this case we will implement this as a custom movement mode called **Climbing**, this movement mode includes a set of **Climbing** states (e.g. *None*, *Grabbing*, *Grabbed*, *Releasing*).

Worth note, that like previous examples, this follows the same guidelines previously defined when adding new movement modes and or states.

When a player presses the interact key (E) and if it is within the ladder trigger, the **Climb** command is executed to issue a climbing mechanic, this will check if the character is able to climb and react accordingly.

Once a climb has been allowed, the `Climb` method will change the current movement mode to our new custom movement mode as follows:

```
SetMovementMode(MovementMode.Custom, (int) CustomMovementMode.Climbing);
```

Worth note `SetMovementMode` receives an *int* parameter as the current custom state id, so a *enum* to *int* cast is necessary.

We make use of `OnMovementModeChanged` method, to handle our new *movement mode* enter / exit and set any additional configuration required for this, e.g. *Cancel jumps*, *disable / enable ground constraint*, etc.

Once the character is in **Climbing** movement mode, the **Climbing** method is responsible for moving the character along the ladder path and updating the climbing state accordingly.

The character exits a **Climbing** movement move on demand (release ladder) or if reaches one of the ladder enter/exit points. To leave the **Climbing** movement mode, we change to **Falling** movement mode as this is a somewhat "intelligent" state which will automatically switch to **Walking** if necessary (e.g. `Character` is on walkable ground), and can be used as your default exit movement mode.

Prevent Standing On Other `Character`

This shows a way to force a character to slide off another character applying a slide force.

Cinemachine

Here you will find examples showing how to modify a `Character` to take advantage of the **Cinemachine** features. Worth note these examples are included as a **package** in order to prevent **Cinemachine** package dependencies.

Please make sure to install the Cinemachine package into your project before importing these examples.

First Person

This example shows how to modify the `FirstPersonCharacter` to make use of a **Cinemachine** based camera so you can take advantage of it.

This example replaces the `FirstPersonCharacter` default camera with a CM one, and replaces the procedural crouch animation with a **Cinemachine** based.

Third Person

This example shows how to extend a `Character` to implement a basic 3rd person movement using **Cinemachine**. This makes use of the **3rd Person Follow behavior** introduced in **Cinemachine** 2.6 and should serve you as a starting template to build a complete **Cinemachine** based third person character.

Path Following

This example shows how to extend a `Character` to implement a path following behavior, this makes use of a **Cinemachine Path** to drive the character's movement.

When following a path, this basically computes a movement direction vector from the character's current position to its target position in path, if distance to target position is "close enough" update's position in path (our next target position) based on character's velocity making the character's move along the path.

Worth note this implements a **path following behavior** and its not a hard-lock path following, as this offers a more fluid and natural looking path following better for AI, patrolling, etc.

Visual Scripting (Bolt)

Here you will find examples of how to make use of a `Character` with **Bolt**. Worth note these examples are included as a **package** in order to prevent **Bolt** package dependencies.

It includes updated examples for Visual Scripting (previously Bolt).

Please make sure to install the Bolt package into your project before importing these examples.

Networking Examples

This update includes networking examples for major Unity networking libraries, such as **Netcode for GameObjects**, **Mirror**, **Fusion** and **FishNet** for both client-authority and server-authority with client-side prediction and reconciliation.

The examples are developed based on corresponding networking libraries guidelines and examples and show the steps needed to use a **Character** in a networking environment.

The example assumes the user is familiar with the desired networking library and at least has successfully completed the corresponding tutorials and has the corresponding networking library already installed and running.

The examples include client-authority (**NetCode**, **Mirror** and **FishNet**) and server-authority with client side prediction for both **FishNet** and **Fusion** since these are the only which support it out-of-the-box without additional plugins or addons.

Client-Authority Examples

No matter which networking library is used, implementing client-authority is pretty straightforward, it starts by creating a controller (i.e: externally controlling a **Character**) inheriting from **NetworkBehaviour**. This controller is responsible for handling the owner (local player) input.

When a controller is used, it's important to prevent the **Character** class from handling the user input (enabled by default) setting its **handleInput** property to false, or in case of using the new input system, leave its **actions** property empty (null), since this will be externally managed by our controller.

Netcode for GameObjects

As commented above, it just requires a controller inheriting from `NetworkBehaviour` and using this to handle the local input and issue corresponding actions to the **Character** using its input action commands (`StartJump`, `StopJump`, `Crouch`, etc) in response to corresponding user input.

Mirror

This example is pretty much identical to the **Netcode** example, with just a few minor changes. In our controller (once again, extending the **Mirror NetworkBehaviour**), we need to disable the **Character** auto-simulation routine (i.e: its `LateFixedUpdate`) and its **Rigidbody** interpolation for non-owner (ie: ! isLocalPlayer) so only the local player can stimulate its own character. It will be replicated by the **NetworkTransform** to other players.

FishNet

This example uses **FishNet** to implement client-authority movement, the procedure is basically the same as with the other networking libraries using client-authority, however some additional settings are required to correctly work.

First, we need to disable the **Character** auto simulation method (i.e: its `LateFixedUpdate`) for non owner, so only the owner client can simulate its character, additionally we also need to disable the **Rigidbody** interpolation since this will be managed by the **NetworkTransform**.

```
public override void OnStartNetwork()
{
    base.OnStartNetwork();

    // If not is Owner...

    if (!base.Owner.IsLocalClient)
    {
        // Disable Character auto simulation (ie: its LateFixedUpdate).
        // Only the owner can simulate it.

        character.enableLateFixedUpdate = false;

        // Disable rigidbody interpolation. It is handled by FishNet.

        CharacterMovement characterMovement =
character.GetCharacterMovement();
        characterMovement.interpolation = RigidbodyInterpolation.None;
    }
}
```

IMPORTANT NOTE:

Worth noting **FishNet** disables Unity **Auto Simulation** setting, disabling this will cause jitter on ECM2 examples since it disables the **Rigidbody interpolation**.

Server-Authority Examples

These are examples for **FishNet** and **Photon Fusion** network libraries, since they are the only ones with built-in out-of-the-box client side prediction and reconciliation, a must when implementing server-authority movement in order to provide responsive actions and snappy feedback on the client.

FishNet

This example shows how to implement a **Character** server-authority movement using **FishNet** networking library with full client-side prediction and reconciliation.

When implementing CSP (Client-Side Prediction), it is important to replicate relevant data required to ensure simulation continuity between the server and the client, this "**character state"** is the data required to perform actions that both server and client must be aware of.

For this we create a custom **Character** class (i.e: **PlayerCharacter**) extending the default **Character** class, this allows us to expose a state and add the required data our game could need in a compact way.

The **PlayerController** class extends a **NetworkBehaviour** and is responsible for running the character simulation, this includes handling user input, simulating the character and performing client side prediction and reconciliation.

Like with other examples, it's important to disable the **Character** input handling and its auto-simulation, since in a networking environment, it's the controller's responsibility to perform those tasks.

Photon Fusion

This example shows how to implement a **Character** server-authority movement using **Photon Fusion** networking library with full client-side prediction and reconciliation.

The implementation is broken into several classes, each with a particular task as this approach will make it clear each class responsibility.

PlayerInput

This class is responsible for collecting the user input, polling the local client and populating the defined input struct (`INetworkInput`).

PlayerChracter

This class extends the default **Character** class and exposes the character state; this is the data required to ensure simulation continuity between the server and the client.

NetworkChracter

This class extends the **NetworkTransform** class and it's responsible for performing the **Character** state synchronization across the network. This is implemented in its `CopyFromBufferToEngine` and `CopyFromEngineToBuffer` methods.

PlayerController

This extends a **NetworkBehaviour** and it's responsible for actually controlling our character. This will check for any input received, issue character actions and perform character simulation.

Disclaimer

The presented examples show a way to make use of a **Character** in a networking environment in an easy and practical way. It's not intended to fully replicate and implement all the **Character** class features such as moving platforms, as this in particular is a very advanced topic and requires special case handling and tradeoffs based on game needs.

I strongly suggest this great in-depth tutorial from Photon guys, explaining moving platforms in multiplayer games.