Easy Character Movement 2

Rigidbody-based Character Movement System

User Manual

Version 1.0.1

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.

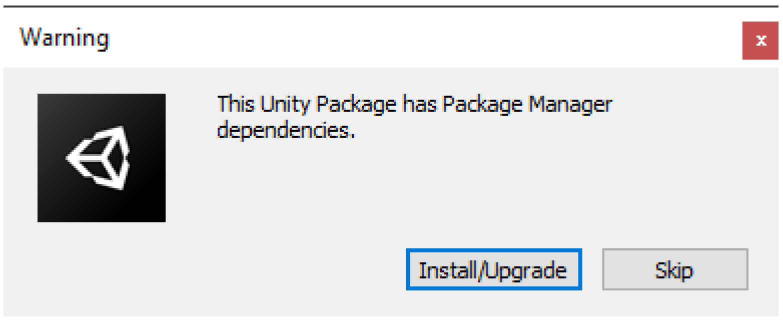
Worth note ECM2 makes full use of **Unity's New Input system**, and as such, it requires it to be installed in your project.

While you can choose to install the input system package during the ECM2 import process, **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, located here.

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:

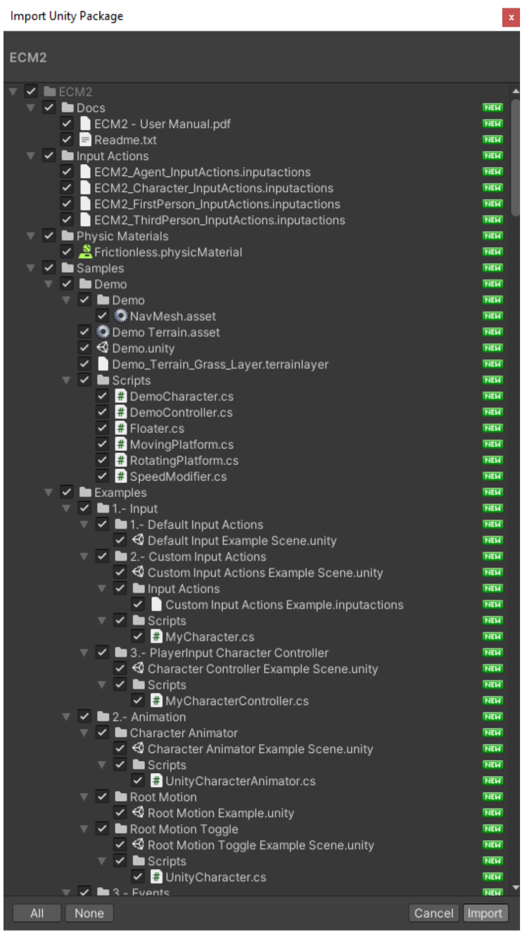


Warning

This Unity Package has Package Manager dependencies.

Press Install/Upgrade button to continue.

The default import unity package dialog will prompt, here you can choose what to import, while you can completely ignore the samples folder, I suggest you to import it as this apart from the demo and many example scenes, includes 4 different template scenes for rapid prototyping.



Import Unity Package

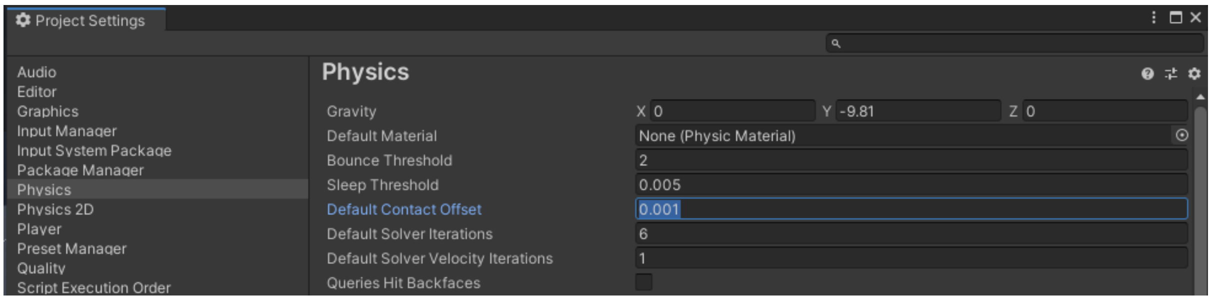
ECM2

```
ECM2
  Docs
    ECM2 - User Manual.pdf
    Readme.txt
  Input Actions
    ECM2_Agent_InputActions.inputactions
    ECM2_Character_InputActions.inputactions
    ECM2_FirstPerson_InputActions.inputactions
    ECM2_ThirdPerson_InputActions.inputactions
  Physic Materials
    Frictionless.physicMaterial
  Samples
    Demo
      Demo
        NavMesh.asset
      Demo Terrain.asset
      Demo.unity
      Demo_Terrain_Grass_Layer.terrainlayer
      Scripts
        DemoCharacter.cs
        DemoController.cs
        Floater.cs
        MovingPlatform.cs
        RotatingPlatform.cs
        SpeedModifier.cs
    Examples
      1.- Input
        1.- Default Input Actions
          Default Input Example Scene.unity
        2.- Custom Input Actions
          Custom Input Actions Example Scene.unity
          Input Actions
            Custom Input Actions Example.inputactions
          Scripts
            MyCharacter.cs
        3.- PlayerInput Character Controller
          Character Controller Example Scene.unity
          Scripts
            MyCharacterController.cs
      2.- Animation
        Character Animator
          Character Animator Example Scene.unity
          Scripts
            UnityCharacterAnimator.cs
        Root Motion
          Root Motion Example.unity
        Root Motion Toggle
          Root Motion Toggle Example Scene.unity
          Scripts
            UnityCharacter.cs
      3.- Events
```

All None Cancel Import

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

Once ECM2 has been successfully imported into your project, while not mandatory, I suggest you modify the default physics contact offset value and set it to 0.001 to keep it in sync with ECM2 default contact offset.



Project Settings

Audio
Editor
Graphics
Input Manager
Input System Package
Package Manager
Physics
Physics 2D
Player
Preset Manager
Quality
Script Execution Order

Physics

Gravity	X 0 Y -9.81 Z 0
Default Material	None (Physic Material)
Bounce Threshold	2
Sleep Threshold	0.005
Default Contact Offset	0.001
Default Solver Iterations	6
Default Solver Velocity Iterations	1
Queries Hit Backfaces	

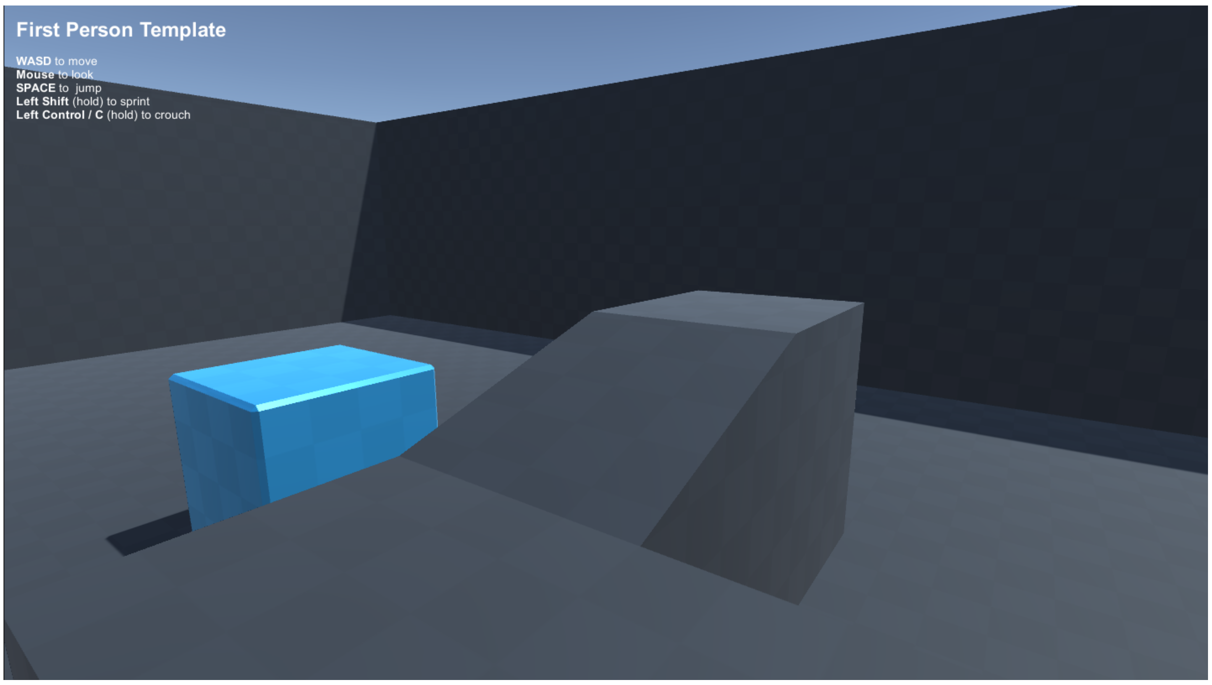
Done! You have successfully imported ECM2 into your project.

Quickstart

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

First Person Template Scene

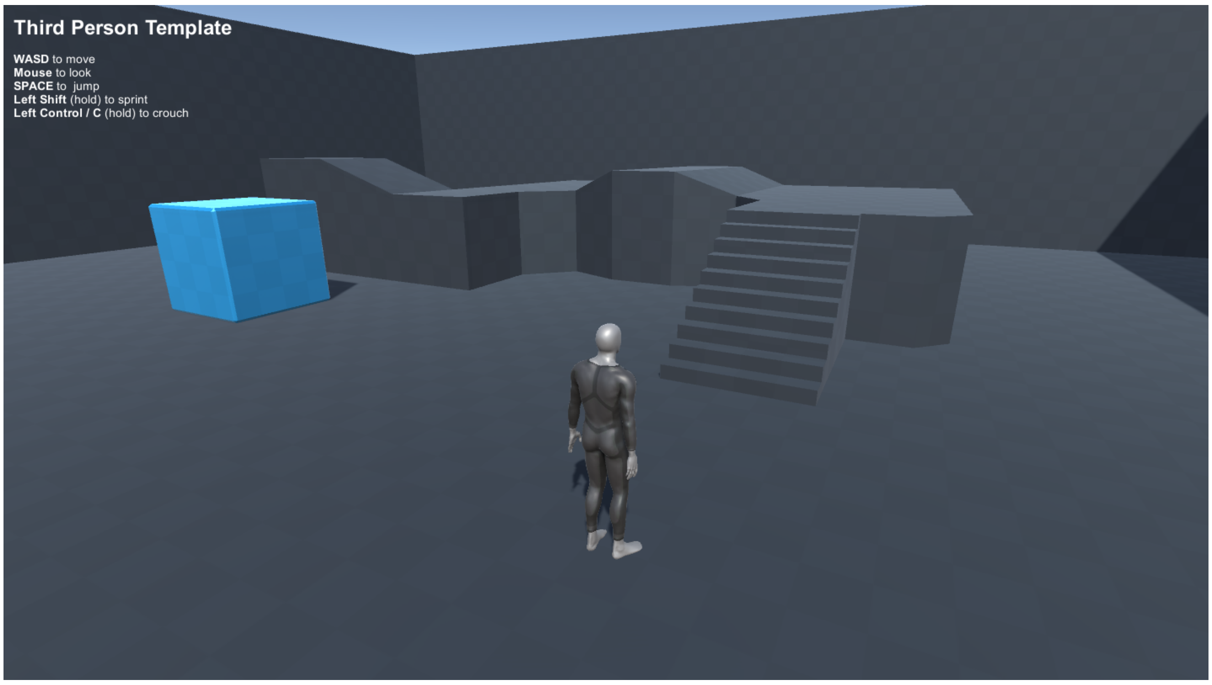
This features a player character which is viewed from first person perspective. The character can be moved around the level using a keyboard or controller. Additionally the player can look around using the mouse or controller.



The character extends the `FirstPersonCharacter` class.

Third Person Template Scene

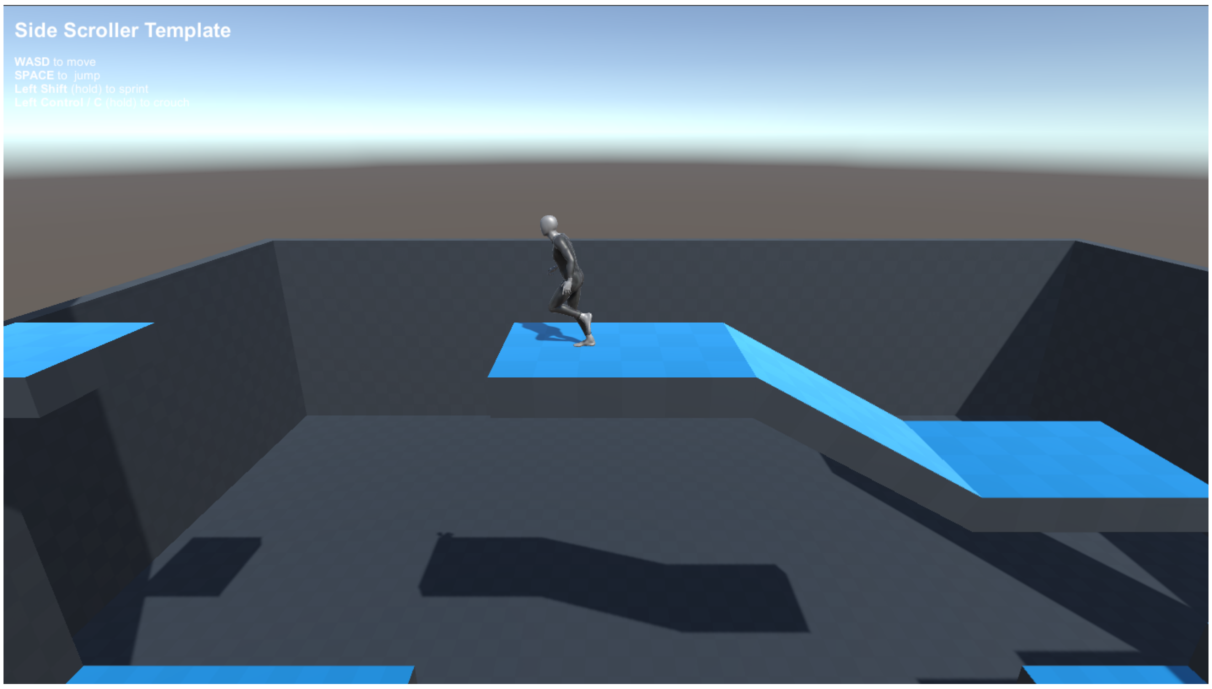
The third-person template features a playable character where the camera is positioned behind and slightly above the character. As the character moves using a mouse, keyboard or controller, the camera follows the character, observing from an over-the-shoulder point of view. This perspective emphasizes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


Third Person Template
WASD to move
Mouse to look
SPACE to jump
Left Shift (hold) to sprint
Left Control / C (hold) to crouch

The character extends the `ThirdPersonCharacter` class.

Side Scroller Template Scene

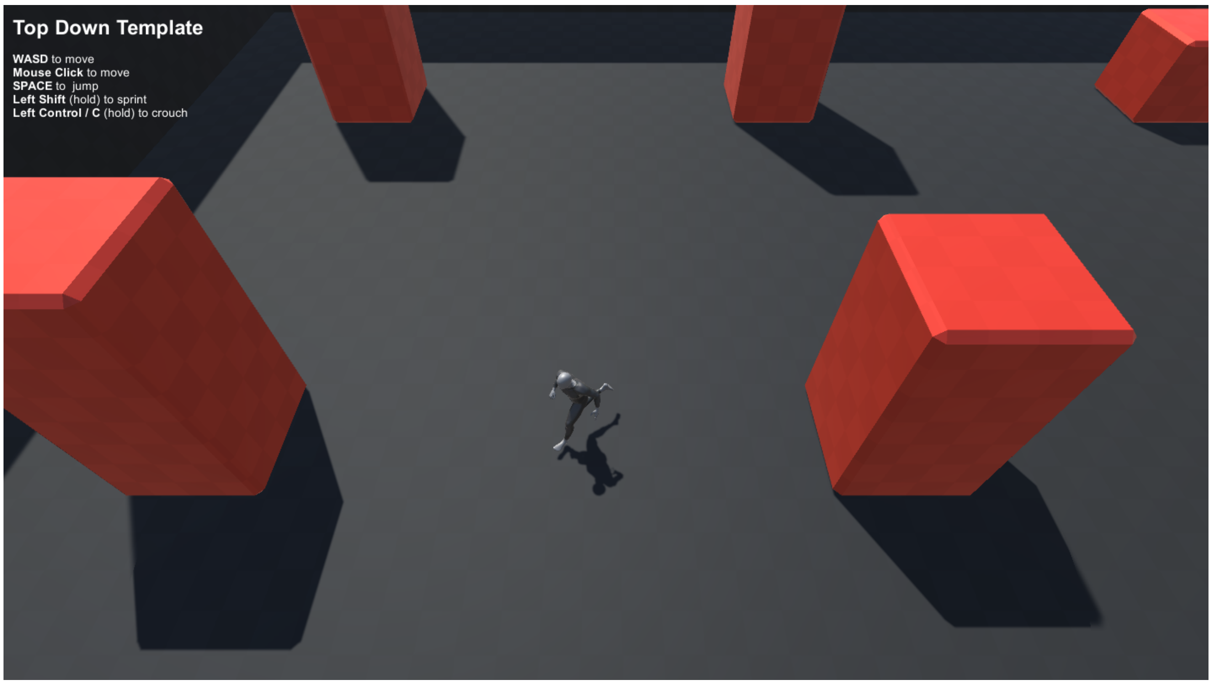
This side scroller template features an animated model with a camera positioned at the character's side. The character can be controlled with a keyboard or controller and constrained to two dimensions: forward, backward and jumping. A number of ledges are featured in the level and the character can jump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


Side Scroller Template
WASD to move
SPACE to jump
Left Shift (hold) to sprint
Left Control / C (hold) to crouch

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 (`ECM2\Samples\Demo`), and over 25 fully commented examples (`ECM2\Samples\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 feature-rich, highly customizable, general purpose (not game-genre tied) rigidbody-based character movement system.

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 thas 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.

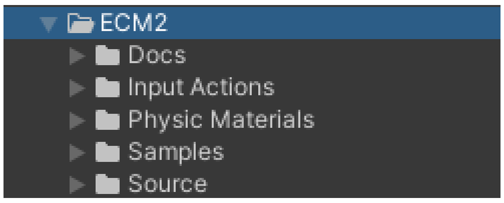
ECM2 includes 25+ examples, ranging from using the new input system custom actions, `PlayerInput` component, first person, third person, target lock, twin-sick movement, dash, slide, ladders, Cinemachine, and many more! Extensive documentation, and fully commented readable source code.

Features

- Rigidbody-based **general purpose** character movement.
- **Capsule-based** character collisions.
- Full integration with **Unity New Input System**.
- **Cinemachine** integration examples.
- **Highly configurable and robust**, ground and collision detection and response system based on the well-known **Collide and Slide** algorithm.
- Ability to **climb high steps** (higher than capsule radius), up to character's height.
- **Different movement modes** and states, like walking, falling, unlimited variable height jump, crouching, sprinting, flying, swimming.
- **First person**, **Third person**, **Agent** "base" characters.
- **Different template scenes** (First person, Third person, Side scroller, Top down) for faster startup.
- **25+ 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 scripts (e.g. Bolt).
- **Physics Volumes**, to easily handle water, falling zones, etc. with configurable properties.
- **Fully configurable friction based movement including water buoyancy**!
- **Root motion** support.
- **Moving platforms** without parenting.
- **Custom gravity and up direction** (eg: mario galaxy like).
- **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 events** like GroundHit, MovementHit, Landed, WillLand, Jumped, ReachedJumpApex, etc.
- Extensive and **configurable collision filtering**.
- Easy **integration into existing projects**.
- **Fully commented C# source code**. Clear, readable and easy to modify.
- **Mobile friendly**.
- **Garbage-Collector friendly**.
- And **much more!**

Package Contents



Docs

It contains the ECM2 user manual.

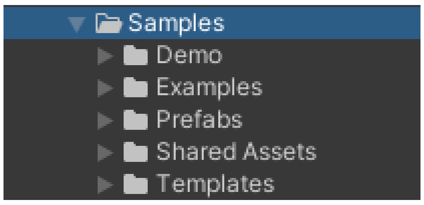
Input Actions

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

Physic Materials

This contains the ECM2 Character frictionless material, this is required in order to slide along collisions.

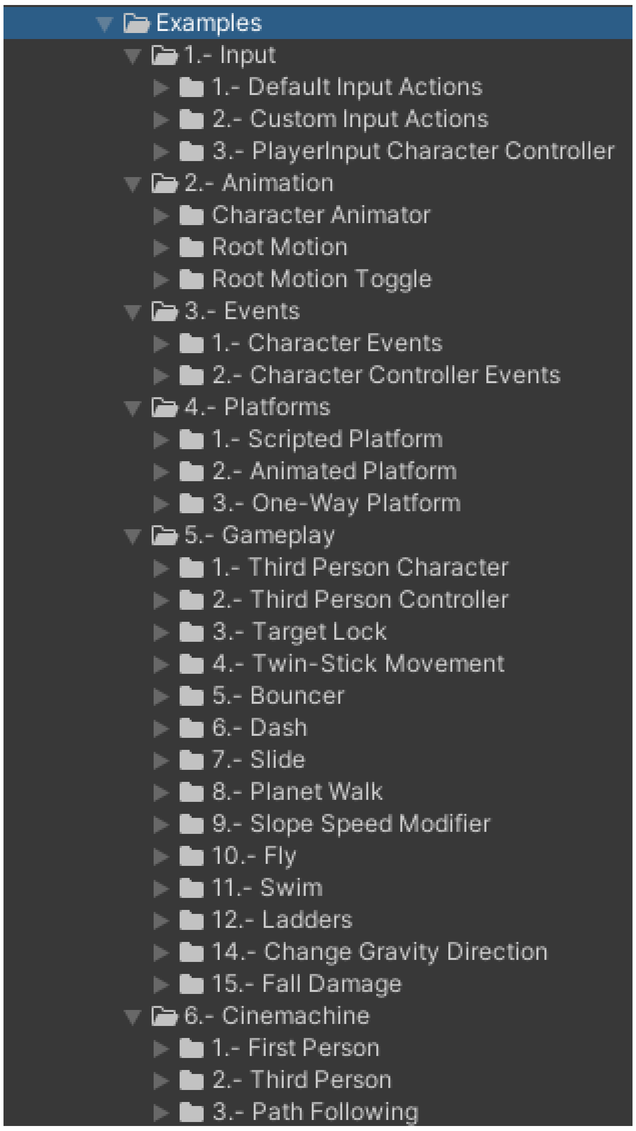
Samples



Demo

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

Examples



Here you can find over 30 fully-commented examples.

Prefabs

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

Shared Assets

Includes all the assets used by the demo, examples, templates and prefabs.

Templates

This contains the different template scenes used for faster startup.

Source

Contains ECM2 full source code.

Introduction

Easy Character Movement 2 is a set of components that provides a movement system with common modes of movement 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!

ECM2 (for short), has been developed in response to the great feedback I received from its awesome user base and in order to improve the very capable ECM with an up to date set of features.

ECM2 is based on a well-known (but not so easy to implement) collision detection and response algorithm called 'Collide and slide' providing a robust and well-known behavior for character control. This algorithm has been around for a long time (e.g. Used by original Quake) and to date, many modern games use it as part of their character's movement.

While I have developed ECM2 to follow the same methodology as ECM, due this new algorithm some minor changes were necessary but in the end IMO it offers a much more flexible, robust and easier to use general purpose character movement system.

Description

Easy Character Movement 2 follows the single responsibility principle where a given class should be, ideally, responsible for just one task. Thus, the Easy Character Movement 2 system is composed of several classes with specific tasks, being the two most importants: `CharacterMovement` component and the `Character` class.

The `CharacterMovement` component is responsible to perform all the geometric related tasks, such as ground detection (e.g. Classify 'ground' into walkable or non-walkable), collision detection and response, handle platform movement, and to physically move the character to its new position.

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 somewhat similar to the `BaseCharacterController` of the previous Easy Character Movement version, however in ECM2 a clear separation between a `Character` and the `Controller` has been made, as now you can use a default `Character` as it its without the requirement of 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 does it work ?

Each `FixedUpdate`, a `Character` will call its `Move` method, which handles all pending forces and impulses accumulated during last frame and will call the `CharacterMovement`

`Move` function (remember it is responsible for all geometric related tasks) to physically move the `Character` to its new position.

The `CharacterMovement Move` method will perform the ground detection, (e.g. classify the 'ground' into walkable or non-walkable) and then call its `OnMove` delegate, to let the `Character` determine how should it be moved accounting its current movement mode (e.g. walking, falling, swimming, etc).

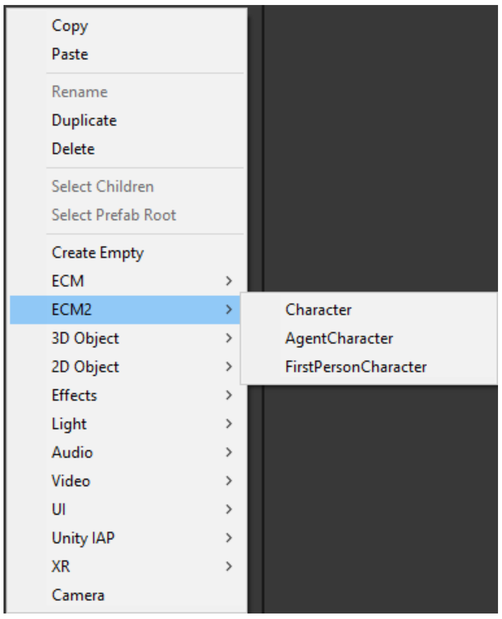
Each movement mode has its own function that is responsible for calculating velocity. For example, `Walking` determines the character's movement physics when moving on the ground, while `Falling` determines how it behaves in the air.

If a movement mode changes during a `FixedUpdate`, such as when a character starts falling or landed, the movement mode function calls `SetMovementMode` function to continue the character's motion in the new movement mode.

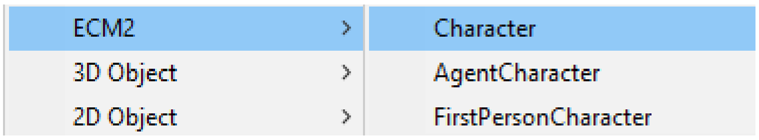
Once the new velocity has been calculated the `CharacterMovement` component will use this to "look ahead" and perform a collision-constrained movement to determine the character's new position and apply the final movement to the owning character.

How do I create a Character?

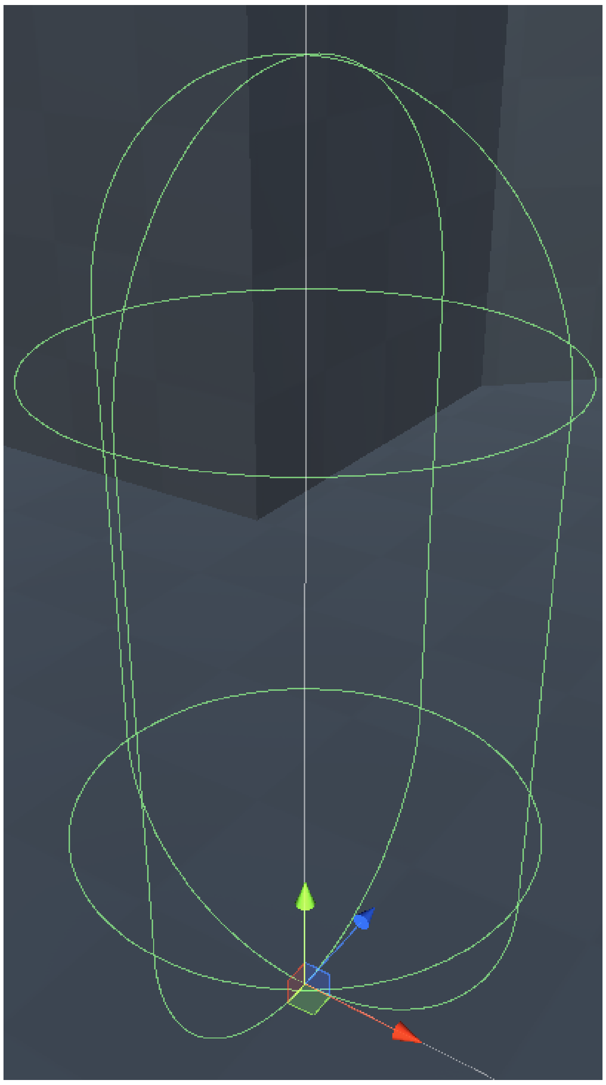
1. Right-click on the hierarchy window to open the creation dialog.



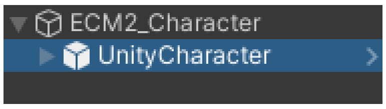
2. Select the type of `Character` you would like to create.



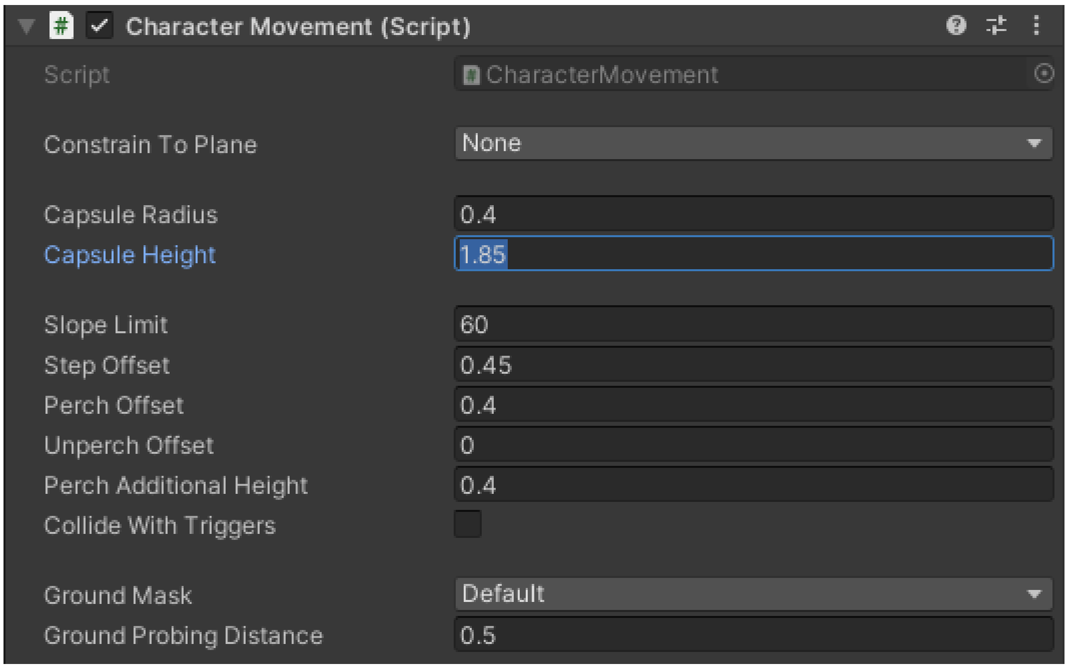
3. It will create an empty character (no visual representation) named `ECM2_Character`, `ECM2_AgentCharacter` or `ECM2_FirstPersonCharacter` based on your selection. Make sure its origin is at 0,0,0, as this will save you troubles when parenting your character's visual representation (e.g. your model).



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



5. 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.



Character Movement (Script)

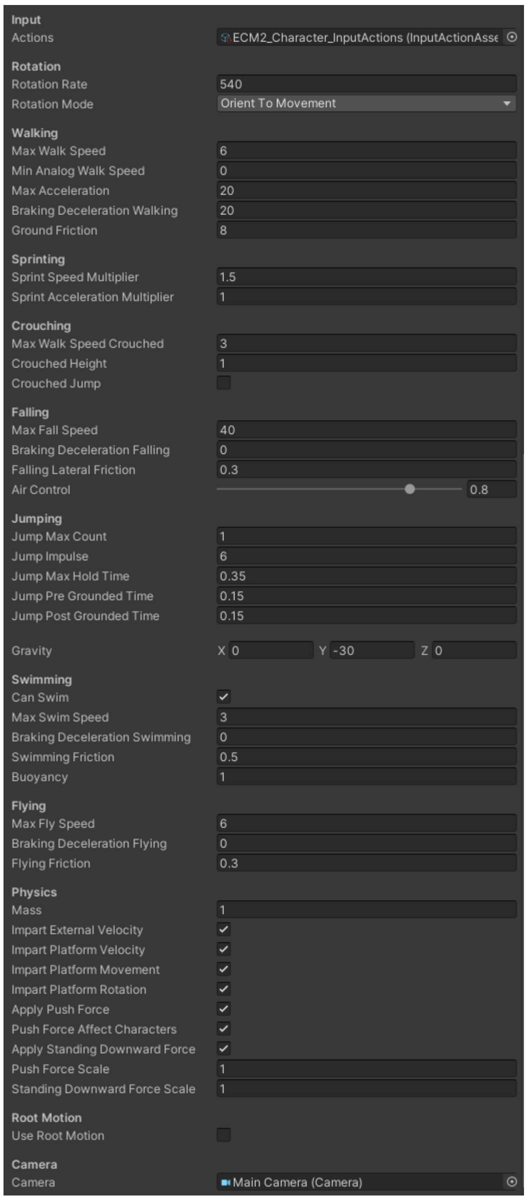
Script	CharacterMovement
Constrain To Plane	None
Capsule Radius	0.4
Capsule Height	1.85
Slope Limit	60
Step Offset	0.45
Perch Offset	0.4
Unperch Offset	0
Perch Additional Height	0.4
Collide With Triggers	☐
Ground Mask	Default
Ground Probing Distance	0.5

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

Base Characters

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`.

`Character` class



Input	
Actions	ECM2_Character_InputActions (InputActionAsse
Rotation	
Rotation Rate	540
Rotation Mode	Orient To Movement
Walking	
Max Walk Speed	6
Min Analog Walk Speed	0
Max Acceleration	20
Braking Deceleration Walking	20
Ground Friction	8
Sprinting	
Sprint Speed Multiplier	1.5
Sprint Acceleration Multiplier	1
Crouching	
Max Walk Speed Crouched	3
Crouched Height	1
Crouched Jump	☐
Falling	
Max Fall Speed	40
Braking Deceleration Falling	0
Falling Lateral Friction	0.3
Air Control	0.8
Jumping	
Jump Max Count	1
Jump Impulse	6
Jump Max Hold Time	0.35
Jump Pre Grounded Time	0.15
Jump Post Grounded Time	0.15
Gravity	X 0 Y -30 Z 0
Swimming	
Can Swim	✓
Max Swim Speed	3
Braking Deceleration Swimming	0
Swimming Friction	0.5
Buoyancy	1
Flying	
Max Fly Speed	6
Braking Deceleration Flying	0
Flying Friction	0.3
Physics	
Mass	1
Impart External Velocity	✓
Impart Platform Velocity	✓
Impart Platform Movement	✓
Impart Platform Rotation	✓
Apply Push Force	✓
Push Force Affect Characters	✓
Apply Standing Downward Force	✓
Push Force Scale	1
Standing Downward Force Scale	1
Root Motion	
Use Root Motion	☐
Camera	
Camera	Main Camera (Camera)

Actions

Input actions asset associated with this character. This is mainly for player controlled characters. If not assigned, the `Character` class will not process input. This allows the character to be externally controlled (e.g. By a `Controller`).

Rotation Rate

Change in rotation per second (degrees per second). If set to zero (0) will perform instant turns. This is particularly useful for 2D / 2.5D games to perform side to side lock rotation.

Rotation Mode

Character's current rotation mode:

- `None`: Disables character's rotation.
- `OrientToMovement`: Rotates the character towards the given input move 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 rotation.

Max Walk Speed

The maximum ground speed when walking. Also determines maximum lateral speed when falling.

Min Analog Walk Speed

The ground speed that we should accelerate up to when walking at minimum analog stick tilt.

Max Acceleration

Max Acceleration (rate of change of velocity).

Braking Deceleration Walking

Deceleration when walking and not applying input acceleration. This is a constant opposing force that directly lowers velocity by a constant value.

Ground Friction

Setting that affects movement control. Higher values allow faster changes in direction. If `useSeparateBrakingFriction` is `false`, it also affects the ability to stop more quickly when braking (whenever input acceleration is zero).

Sprint Speed Multiplier

The max walk speed multiplier while sprinting.

Sprint Acceleration Multiplier

The max acceleration multiplier while sprinting.

Max Walk Speed Crouched

The maximum ground speed when walking and crouched.

Crouched Height

The character's capsule height while crouched.

Crouched Jump

If true, allows to jump while crouched (uncrouch character).

Max Fall Speed

The maximum vertical velocity a character can reach when falling. E.g. Terminal velocity. `Note`: If the character is within a `PhysicsVolume`, this value will be replaced with the `PhysicsVolume` `maxFallSpeed` property value.

Braking Deceleration Falling

Lateral deceleration when falling and not applying input acceleration.

Falling Lateral Friction

Friction to apply to lateral movement when falling. If `useSeparateBrakingFriction` is `false`, it also affects the ability to stop more quickly when braking (whenever input acceleration is zero).

Air Control

When falling, the amount of lateral movement control available to the character. 0 = no control, 1 = full control at max acceleration.

Jump Max Count

The max number of jumps the Character can perform.

Jump Impulse

Initial velocity (instantaneous vertical velocity) when jumping.

Jump Max Hold Time

The maximum time (in seconds) to hold the jump. E.g. Variable height jump.

Jump Pre-Grounded Time

How early before hitting the ground you can trigger a jump (in seconds).

Jump Post-Grounded Time

How long after leaving the ground you can trigger a jump (in seconds). Sometimes referred to as “Coyote time”.

Gravity

The gravity applied to this character.

Can Swim

If True, the character is capable of swimming or moving through fluid volumes.

Max Sim Speed

The maximum swimming speed.

Braking Deceleration Swimming

Deceleration when swimming and not applying input acceleration.

Swimming Friction

Friction to apply to movement when swimming.

Buoyancy

Water buoyancy ratio. 1 = Neutral Buoyancy, 0 = No Buoyancy. Can be greater than 1.

Max Fly Speed

The maximum flying speed.

Braking Deceleration Flying

Deceleration when flying and not applying acceleration.

Flying Friction

Friction to apply to movement when flying.

Mass

This character's mass. Will set the `Rigidbody`’s mass.

Impart External Velocity

If true, impart the velocity caused by external forces during internal physics update.

Impart Platform Velocity

If true, impart the platform's velocity when jumping or falling off it.

Impart Platform Movement

Whether the character moves with the moving platform it is standing on.

Impart Platform Rotation

Whether the character receives the changes in rotation of the platform it is standing on.

Apply Push Force

Should apply a push force to rigidbodies when walking into them?

Push Force Affect Characters

Should apply push force to characters when walking into them ?

Apply Standing Downward Force

Should we apply a downward force to rigidbodies we stand on ?

Push Force Scale

Force applied to rigidbodies when walking into them (due to mass and relative velocity) is scaled by this amount.

Standing Downward Force Scale

Force applied to rigidbodies we stand on (due to mass and gravity) is scaled by this amount.

Use Root Motion

Should animation determine the character's movement ?

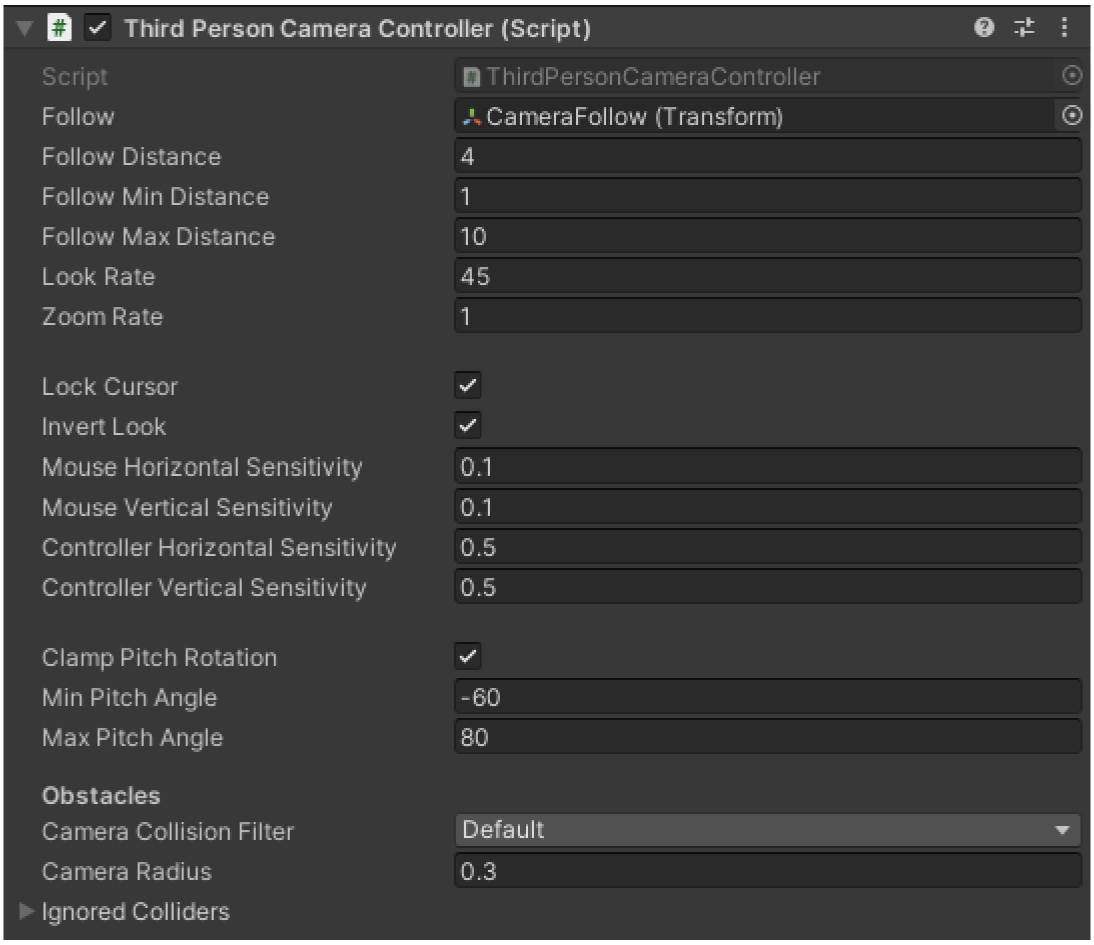
Camera

Reference to the Player's `Camera`. If assigned, the character's movement will be relative to this camera, otherwise movement will be relative to the world axis. Like `actions`, this is mainly for player controlled characters.

`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.



Third Person Camera Controller (Script)

Script	ThirdPersonCameraController
Follow	CameraFollow (Transform)
Follow Distance	4
Follow Min Distance	1
Follow Max Distance	10
Look Rate	45
Zoom Rate	1
Lock Cursor	✓
Invert Look	✓
Mouse Horizontal Sensitivity	0.1
Mouse Vertical Sensitivity	0.1
Controller Horizontal Sensitivity	0.5
Controller Vertical Sensitivity	0.5
Clamp Pitch Rotation	✓
Min Pitch Angle	-60
Max Pitch Angle	80
Obstacles	
Camera Collision Filter	Default
Camera Radius	0.3
Ignored Colliders	

Follow

The object that the camera wants to move with (e.g. Its target).

Follow Distance

The default distance behind the Follow target.

Follow Min Distance

The minimum distance to Follow target.

Follow Max Distance

The maximum distance to Follow target.

Look Rate

The change in rotation per second (deg / sec) when looking around.

Zoom Rate

The change in follow distance per second.

Lock Cursor

Should the cursor be locked?

Invert Look

Determines if the vertical axis should be inverted.

Mouse Horizontal Sensitivity

The horizontal sensitivity while using Mouse input, higher values cause faster movement.

Mouse Vertical Sensitivity

The vertical sensitivity while using Mouse input, higher values cause faster movement.

Controller Horizontal Sensitivity

The horizontal sensitivity while using a Controller, higher values cause faster movement.

Controller Vertical Sensitivity

The vertical sensitivity while using a Controller, higher values cause faster movement.

Clamp Pitch Rotation

Should clamp pitch rotation? (e.g. Up / down).

Min Pitch Angle

If clamp pitch rotation is enabled, determine the minimum pitch angle (in degrees).

Max Pitch Angle

If clamp pitch rotation is enabled, determine the maximum pitch angle (in degrees).

Camera Collision Filter

Camera will avoid obstacles on these layers.

Camera Radius

Specifies how close the camera can get to obstacles.

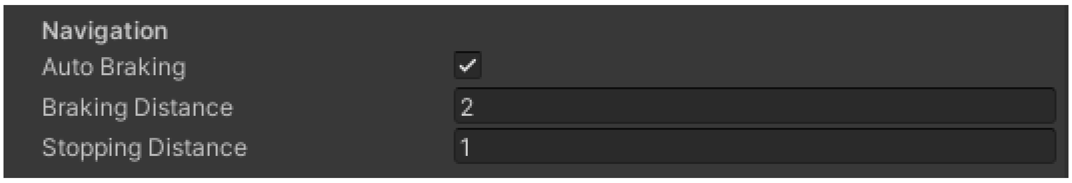
Ignored Colliders

List of colliders to be ignored as obstacles.

`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).

Extends the `Character` class with following properties:



Navigation	
Auto Braking	✓
Braking Distance	2
Stopping Distance	1

Auto Braking

Should the agent brake automatically to avoid overshooting the destination point? If true, the agent will brake automatically as it nears the destination.

Braking Distance

Distance from target position to start braking.

Stopping Distance

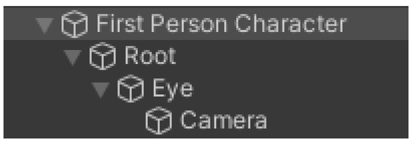
Stop within this distance from the target position.

`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



```
▼ First Person Character
  ▼ Root
    ▼ Eye
        Camera
```

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.

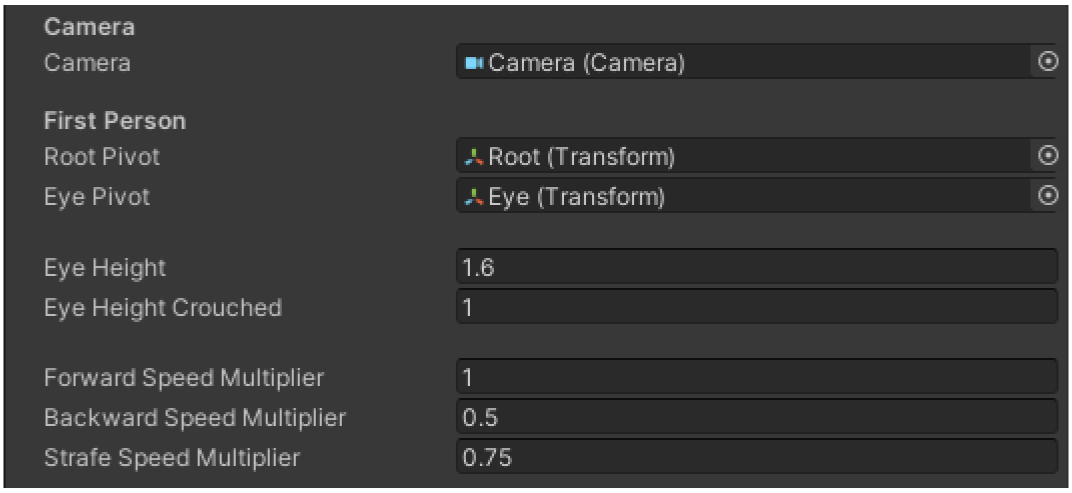
Eye

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` extends the `Character` class with the following properties:



Camera	
Camera	Camera (Camera)
First Person	
Root Pivot	Root (Transform)
Eye Pivot	Eye (Transform)
Eye Height	1.6
Eye Height Crouched	1
Forward Speed Multiplier	1
Backward Speed Multiplier	0.5
Strafe Speed Multiplier	0.75

Root Pivot

The `FirstPersonCharacter` hierarchy Root pivot `GameObject`. TheYaw-Rotation is applied to this.

Eye Pivot

The `FirstPersonCharacter` hierarchy Eye pivot `GameObject`. The Pitch-Rotation is applied to this.

Eye Height

The default Eye height (e.g While Walking). This modifies the Eye pivot's height.

Eye Height Crouched

The Eye height while the character is crouched. This modifies the Eye pivot's height.

Forward Speed Multiplier

The speed multiplier while the character is walking forward.

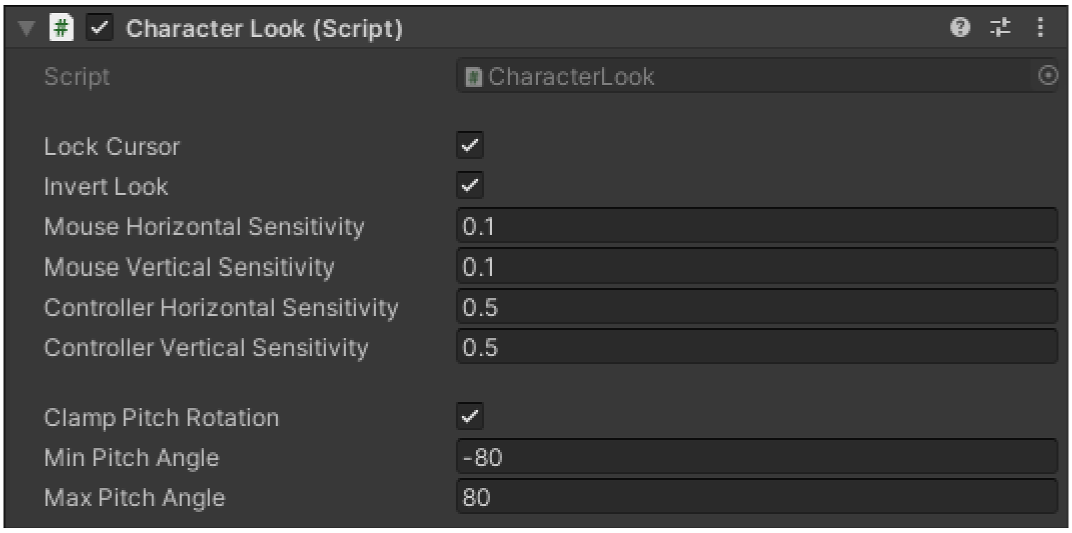
Backward Speed Multiplier

The speed multiplier while the character is walking backward.

Strafe Speed Multiplier

The speed multiplier while the character is walking sideways.

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



Script	CharacterLook
Lock Cursor	✓
Invert Look	✓
Mouse Horizontal Sensitivity	0.1
Mouse Vertical Sensitivity	0.1
Controller Horizontal Sensitivity	0.5
Controller Vertical Sensitivity	0.5
Clamp Pitch Rotation	✓
Min Pitch Angle	-80
Max Pitch Angle	80

Lock Cursor

Determines if the mouse cursor should be locked.

Inver Look

Determines if the vertical axis should be inverted.

Mouse Horizontal Sensitivity

The horizontal sensitivity while using Mouse input, higher values cause faster movement.

Mouse Vertical Sensitivity

The vertical sensitivity while using Mouse input, higher values cause faster movement.

Controller Horizontal Sensitivity

The horizontal sensitivity while using a controller, higher values cause faster movement.

Controller Vertical Sensitivity

The vertical sensitivity while using a controller, higher values cause faster movement.

Clamp Pitch Rotation

Determines if pitch rotation should be clamped between `minPitchAngle` and `maxPitchAngle`.

Min Pitch Angle

If clamp pitch rotation is enabled, determine the minimum pitch angle (in degrees).

Max Pitch Angle

If clamp pitch rotation is enabled, determine the maximum pitch angle (in degrees).

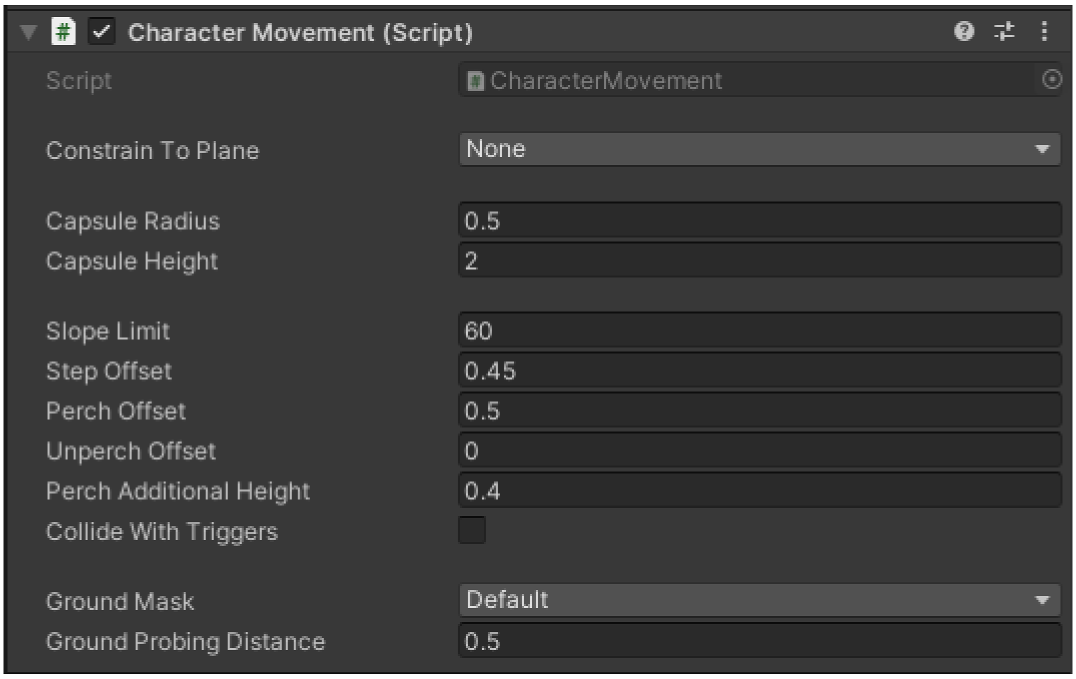
`CharacterMovement` component

The `CharacterMovement` component is responsible to perform all the geometric related tasks, such as ground detection (e.g. Classify 'ground' into walkable or non-walkable), collision detection and response, handle platform movement and rotation, and to physically move the character to its new position.

In previous ECM versions, the `CharacterMovement` component was also responsible for calculating and updating the character's velocity, however in ECM2 this has now been moved (delegated) to the `Character` class.

This change offers greater flexibility, letting the `Character` (and its derived custom classes) decide how it should be moved. For example, while ECM2 includes a robust and fully-configurable friction model to calculate a character's new velocity based on its set of physical properties (e.g. Acceleration, deceleration, friction, buoyancy, etc.) sometimes you would prefer an instant-velocity change (like in a 2D game Mega Man style), or while following a path or even completely new friction model, etc. Now You'll just need to override (extend) the character's `CalcVelocity` method with your new model.

It exposes the following fields for edit:



Script	CharacterMovement
Constrain To Plane	None
Capsule Radius	0.5
Capsule Height	2
Slope Limit	60
Step Offset	0.45
Perch Offset	0.5
Unperch Offset	0
Perch Additional Height	0.4
Collide With Triggers	☐
Ground Mask	Default
Ground Probing Distance	0.5

Constrain To Plane

Constrain the character so movement along the locked axis is not possible. This is particularly useful when developing a 2D game, so you can prevent the character's movement on the z-axis.

Capsule Radius

The Character's capsule collider radius. This automatically configures the `CapsuleCollider` for you.

Capsule Height

The Character's capsule collider height. This automatically configures the `CapsuleCollider` for you.

Slope Limit

The maximum angle (in degrees) for a walkable slope.

Step Offset

The maximum height (in meters) for a valid step (up to character's height).

Perch Offset

Allow a character to perch on the edge of a surface if the horizontal distance from the character's position to the edge is closer than this. A character will not fall off if they are within `stepOffset` of a walkable surface below.

Unperch Offset

Allow a character to unperch from the edge of a surface if the horizontal distance from the character's position to the edge is greater than this. A character will not fall off if they are within `stepOffset` of a walkable surface below.

Perch Additional Height

When perching on a ledge, add this additional distance to `stepOffset` when determining how high above a walkable floor we can perch. Note that we still enforce `stepOffset` to start the step up, this just allows the character to hang off the edge or step slightly higher off the ground.

Collide With Triggers

Determines if a character should collide with triggers.

Ground Mask

Determines the layers to be considered as walkable ground.

Ground Probing Distance

Determines the maximum length of the cast while performing ground detection. As a rule of thumb, configure it to your character's collider radius or greater.

Ground Detection

The `CharacterMovement` component is responsible to perform the ground detection, e.g. To "parse" the "ground" below the character's capsule collider and decide if it is standing on **walkable** or **non-walkable** ground. Later, the character will use this information to update its velocity accordingly.

The ground detection starts with a capsule cast along the character's down direction up to it's `probingDistance`, if no hit, the function ends as no further tests are necessary. By other hand, if a hit occurs the process will perform additional tests to determines if the collided "ground" is **walkable** or **non-walkable**, at the end of this process, the current hit has been fully evaluated and its results can be queried through the `CharacterMovement` `groundHit` and `lastGroundHit` (last frame ground hit results) properties.

Walkable and Non-Walkable Ground

Each of the `CharacterMovement` parameters affects the way how it will interpret if the ground below our character is **walkable** or **not-walkable**.

The first parameter to test during the ground detection phase, is the `slopeLimit` this (as its name suggests) indicates the maximum slope a character can walk, e.g. The angle between the character's up vector and the hit normal is less than `slopeLimit`. However this by itself, does not provide enough information to accurately take a decision in case we hit a corner or edge of a surface.

To overcome this limitation, we follow this initial capsule sweep with two additional short raycasts in order to determine if our character is standing on a ledge.

What is a ledge ?

Is considered a ledge:

- If one side of the edge has no ground.
- If one side of the edge is non-walkable ground (e.g. Its angle WRT character's up vector is greater than `slopeLimit`).
- If one side of the edge has walkable ground, but this is farther than `stepOffset` plus `perchAdditionalHeight`.

Once a ledge has been detected, we will use its ledge-related (`perchOffset, unperchOffset and perchAdditionalHeight)` parameters to determine if the character can stand on a found ledge.

A character can stand on a ledge (empty side):

- If the horizontal distance from the character's position to the ledge is less than `perchOffset`.

- If the character is walking-off a ledge (e.g. Its horizontal velocity points in the same direction as the ledge forward vector) and its horizontal distance to the ledge is less than `unperchOffset`.

- If the ground below the character is closer than `stepOffset` plus `perchAdditionalHeight`

In case the hit is not walkable (e.g. Hit a wall), we perform additional tests in order to determine if it is a **walkable step** (e.g. Its height is less than character's radius) or a **climbable step** (e.g. Its height is higher than character's radius).

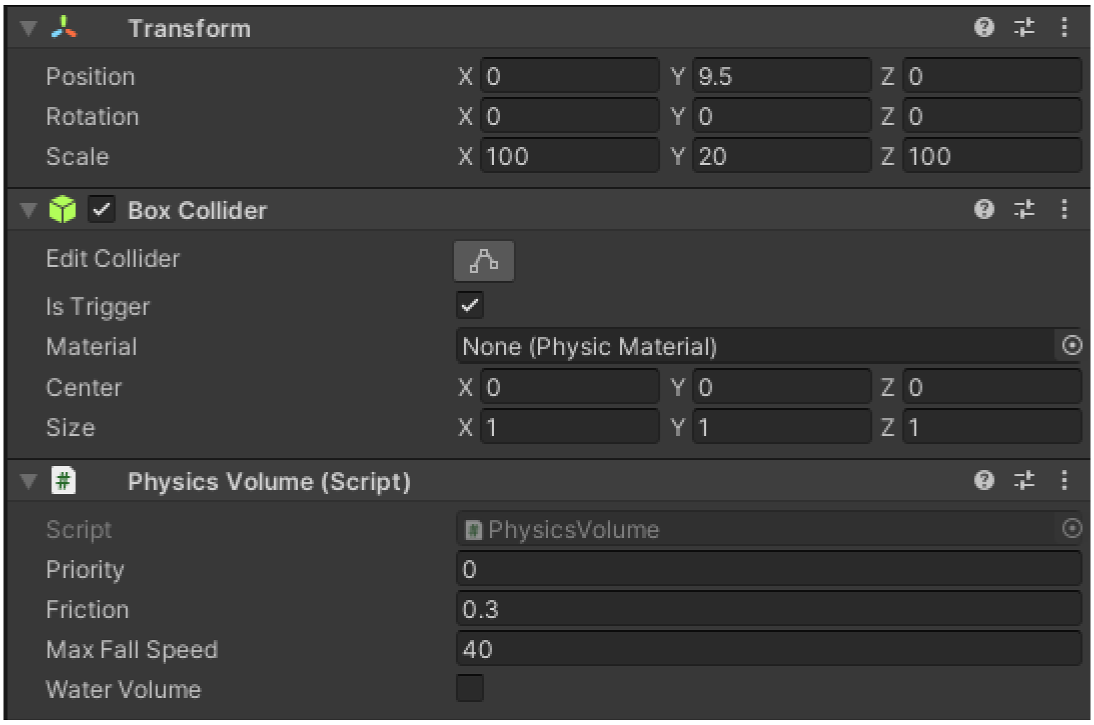
`PhysicsVolume` component

A PhysicsVolume it's a helper component used to define three-dimensional zones to alter the behavior of areas 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. The `CharacterMovementComponent` class uses the current fields to adjust how their owning `Character` moves through the environment.

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

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



Transform

Position	X 0	Y 9.5	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 100	Y 20	Z 100

Box Collider ✓

Edit Collider			
Is Trigger	✓		
Material	None (Physic Material)		
Center	X 0	Y 0	Z 0
Size	X 1	Y 1	Z 1

Physics Volume (Script)

Script	PhysicsVolume
Priority	0
Friction	0.3
Max Fall Speed	40
Water Volume	☐

Priority

Determines which PhysicsVolume takes precedence if they overlap (higher value == higher priority).

Friction

Determines the amount of friction applied by the volume as `Character` using `CharacterMovement` moves through it. The higher this value, the harder it will feel to move through the volume.

Max Fall Speed

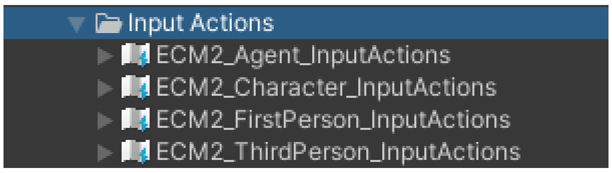
Determines the terminal velocity of Characters using `CharacterMovement` when falling.

Water Volume

Determines if the volume contains a fluid, like water. If enabled, will cause the character to enter its swim movement mode (if the character is allowed to swim).

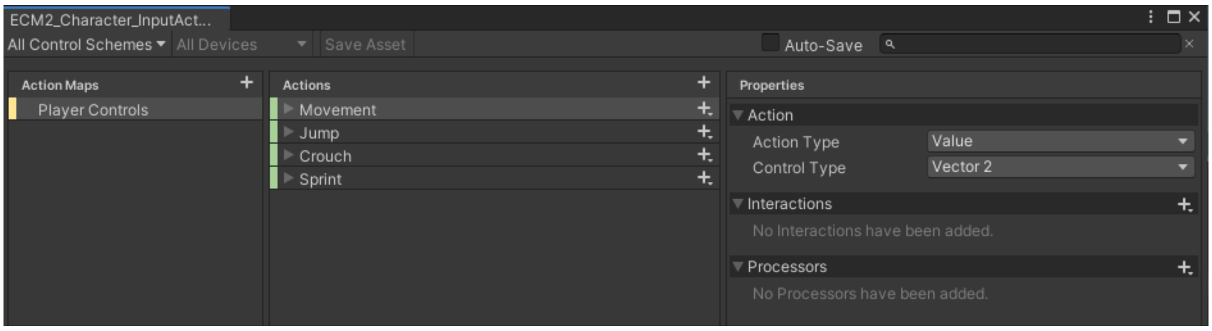
`Character` Input

ECM2 makes full use of 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 **Input Action Asset** is assigned to a `Character`, it will handle the input for you. Each "Base" character includes its corresponding **Input Action Asset**.


Input Actions
ECM2_Agent_InputActions
ECM2_Character_InputActions
ECM2_FirstPerson_InputActions
ECM2_ThirdPerson_InputActions

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



Action Type: Value
Control Type: Vector 2
No Interactions have been added.
No Processors have been added.

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 it with your 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.

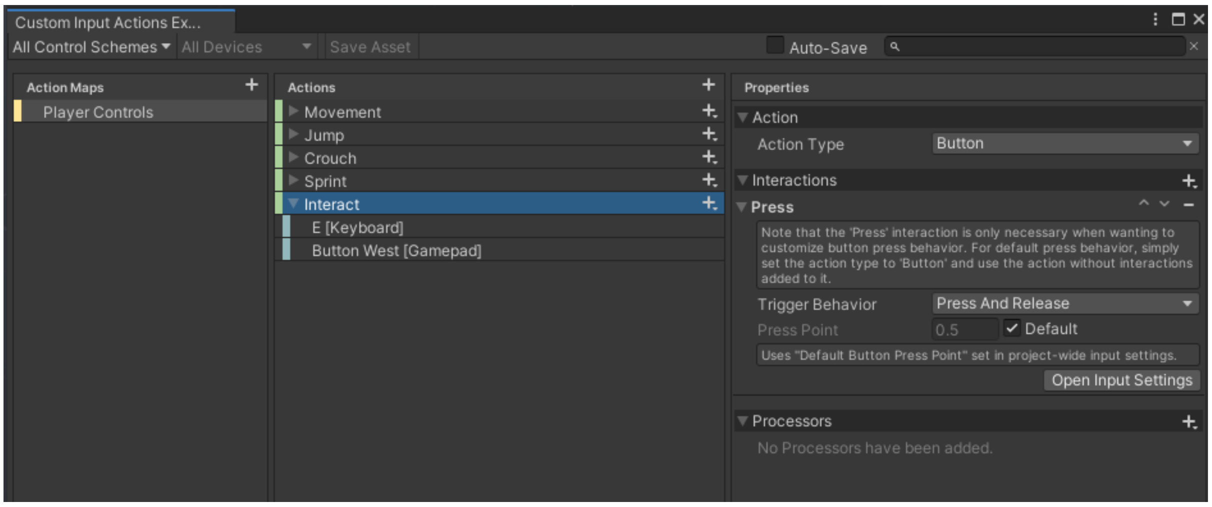
Adding a custom `InputAction` to `MyCharacter`

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:



Action Type: Button

Note that the 'Press' interaction is only necessary when wanting to customize button press behavior. For default press behavior, simply set the action type to 'Button' and use the action without interactions added to it.

Trigger Behavior: Press And Release

Press Point: 0.5 ✓ Default

Uses "Default Button Press Point" set in project-wide input settings.

No Processors have been added.

In order to actually be able to read the state of the new **Interact** `InputAction`, we make our `OnInteract` input event handler to listen for input events. ECM2 includes a predefined method used to configure your player input (if needed), `SetupPlayerInput`.

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

```
public class MyCharacter : Character
{
    // Interact InputAction

    private InputAction interactInputAction { get; set; }

    // Interact InputAction Handler

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

    // Setup player input actions.

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

        base.SetupPlayerInput();

        // Setup Interact input action handlers

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

We use the character `SetupPlayerInput` method to make our `OnInteract` input event handler to listen for input events. However, for an **Action** to do something, you **must first enable it**. You can do this either by individually enabling **Actions**, or by enabling them in bulk through **Action Maps**, here we use the former.

We will extend the Character `OnOnEnable` and `OnOnDisable` methods to enable and disable the Interact Input Action respectively.

OnOnEnable and OnOnDisable

The reason of this "funny" name is because in ECM2 is not allowed to extend the `MonoBehaviour` functions used by our base class (e.g. `OnEnable`, `OnDisable`, `Update`, `FixedUpdate`, etc), instead we define our own set of functions to override following the `Monobehaviour` function name, **but adding the "On" prefix**, so the **MonoBehaviour's** `OnEnable` **function become** `OnOnEnable` **for us.**

```
// Initialize this class.

protected override void OnOnEnable()
{
     // Init Base Class

     base.OnOnEnable();

     // Enable our custom input actions

     interactInputAction?.Enable();
}
```

In the same way we use the `OnOnDisable` `Character` method, to disable our **Input Actions**.

```
// De-Initialize this class.

protected override void OnOnDisable()
{
    // De-Init Base Class

    base.OnOnDisable();

    // Disable our custom input actions

    interactInputAction?.Disable();
 }
```

This summarizes the steps needed to add a custom input to a `Character` derived class.

You can find a fully commented working example for adding a custom input action as part of the included examples (`\ECM2\Samples\Examples\1.- Input\2.- Custom Input Actions`).

Character Controller

As previously seen a `Character` can be "possessed" by a `Controller`, and while not a 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 like Bolt.

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** (`ECM2\Samples\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.

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.

Moving a Character

To move a character you use its **SetMovementInput** function to feed a direction vector in world space. E.g. The desired 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);
```

By other hand, when you assign an **Input Action Asset** to a `Character` (or `Character` derived) class, it will perform the user input and move the character accordly using its `HandleInput` method. When extending a `Character` class, you should use this `HandleInput` method to perform your custom input.

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.

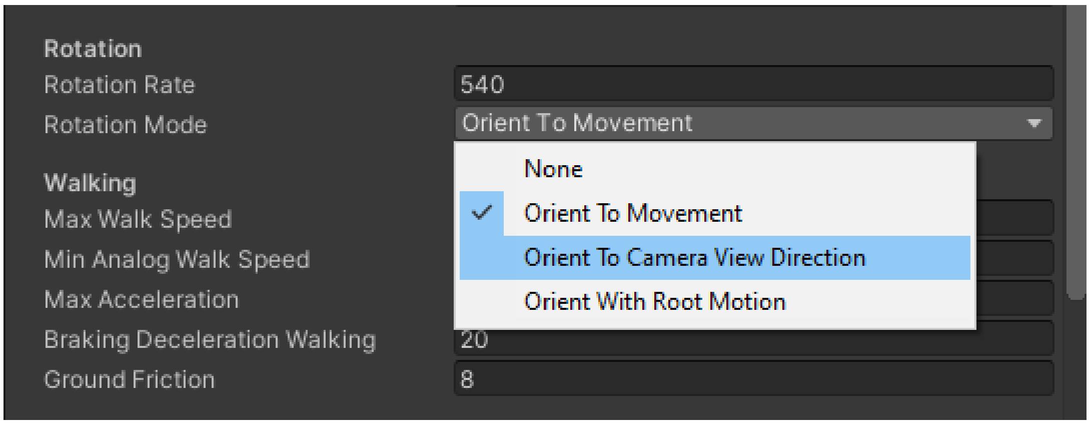
- `Jump`. Start a Jump.
- `StopJumping`. Stops the character from jumping.

- `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 (modifying its rotation), it already includes functionality to handle common rotation / orientation modes.



Rotation

Rotation Rate 540

Rotation Mode Orient To Movement

Walking

Max Walk Speed

Min Analog Walk Speed

Max Acceleration

Braking Deceleration Walking 20

Ground Friction 8

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.

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`.
- `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()`.

These methods should be preferred over transform.position, transform.up, etc, **as this can be different**, being the most obvious case of the `FirstPersonCharacter` where its `transform.forward` vector remains unaltered.

Adding Forces to a Character

As part of the changes introduced in **ECM2**, and while still rigidbody based, **the character's rigidbody should not be directly used**, as it is used as a "**transport**" to physically move the character to our calculated position (**CollideAndSlide**). So it is recommended to use the character's `AddForce` function to apply any force to a character.

This `AddForce` function behaves just like Unity's `AddForce`, but the former correctly complies with our movement algorithm and should be preferred.

`LaunchCharacter` function

`LaunchCharacter`, a newly introduced function, is a useful helper function used to explicitly modify the character's velocity while complying with our movement algorithm.

```
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. while safely complying with our movement algorithm.

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 (previously known as **snap to ground**).

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` function, 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);
```

Animating your `Character`

The `Character` class includes an animation related method `Animate`, you can use this method to handle your **Animator** related code, **however this is not mandatory and can be safely ignored if desired**, as a character does not include or require your character's model be animated in a particular way or ever be animated.

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 in sync with the character's physical state.

Querying Character State

The Character class offers a wide range of methods, events and delegates you can use to read the character's information, e.g: `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 (e.g. `GetCharacterMovement()` method), 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).

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 **handlers** you can use to respond accordly locally (e.g. `Character` derived class) or externally (e.g. `Controller`).

Events

```
// Triggered when the character hits a 'ground' collider.

public event GroundHitEventHandler GroundHit;

// Triggered when the character hits a collider when walking into them.

public event MovementHitEventHandler MovementHit;

// Triggered when the character was not on walkable ground (last frame) and
hit walkable ground.

public event LandedEventHandler Landed;

// Triggered when the character will hit walkable ground.

public event WillLandEventHandler WillLand;

// Triggered when a jump has been successfully triggered and the character is
about to leave ground.

public event JumpedEventHandler Jumped;

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

public event ReachedJumpApexEventHandler ReachedJumpApex;

// Triggered when Character has been launched, e.g. LaunchCharacter…

public event LaunchedEventHandler Launched;

// Triggered when the character crouches.

public event CrouchedEventHandler Crouched;

// Triggered when the character stops crouching.

Public event UncrouchedEventHandler Uncrouched;

// Triggered when the character enters a new physics volume.

public event PhysicsVolumeChangedEventHandler PhysicsVolumeChanged;

// Triggered after MovementMode has changed.

public event MovementModeChangedEventHandler MovementModeChanged;
```

Event Handlers

```
protected virtual void OnGroundHit(ref GroundHit prevGroundHit, ref GroundHit groundHit)

protected virtual void OnMovementHit(ref MovementHit movementHit)

protected virtual void OnLanded()

protected virtual void OnWillLand()

protected virtual void OnJumped()

protected virtual void OnReachedJumpApex()

protected virtual void OnLaunched(Vector3 launchVelocity, bool overrideVerticalVelocity, bool overrideLateralVelocity)

protected virtual void OnCrouched()

protected virtual void OnUncrouched()

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**" 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. and additional input action commands, such as `AddYawInput()`, `AddEyePitchInput()`, etc. Among other functions.

An important observation here, in order to provide a smoother rotation, the `FirstPersonCharacter` handles the character's rotation in a particular way, this applies the **Yaw-Rotation to its Root Pivot Transform** instead to the character's transform, so as previously commented, it is important to use the `GetForwardVector()`, `GetRightVector()`, `GetUpVector()` methods when querying the character's direction vectors, as this are guaranteed to return the up-to-date values.

`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). Additionally it adds the `MoveToLocation()` function, this receives a position (in world space) and will issue the character to intelligently navigate to the given point using its `NavMeshAgent`.

Platforms

A **Platform** is a special type of "dynamic ground" where the player is moved and / or rotated (if desired) within. While this can be moved procedurally or by an animation, **it must extend the `PlatformMovement` class**, as this is responsible to provide an up-to-date platform position, rotation and velocities to the `CharacterMovement`.
A platform is composed of the following components:

- Transform.
- **Kinematic Rigidbody**.
- Collider.
- `PlatformMovement` derived class.

Procedurally Moving a Platform

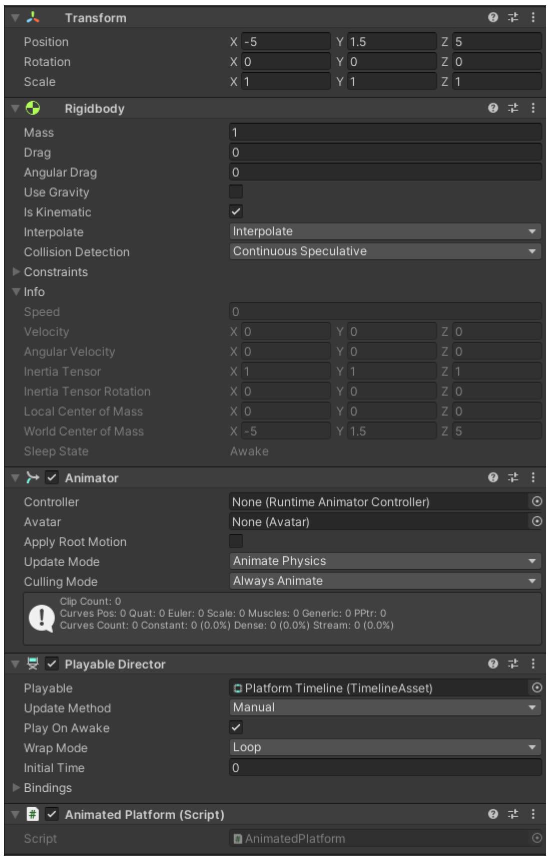
Like a `Character`, to create a moving platform, you'll need to create a new script extending the `PlatformMovement` class and override its `OnMove()` method to perform your movement.

```
public sealed class MyPlatform : PlatformMovement
{
    public static float EaseInOut(float time, float duration)
    {
        return -0.5f * (Mathf.Cos(Mathf.PI * time / duration) - 1.0f);
    }

    protected override void OnMove()
    {
        float t = EaseInOut(Mathf.PingPong(Time.time, moveTime), moveTime);

        position = Vector3.Lerp(_startPosition, _endPosition, t);
    }
}
```

As you can see in the above code, we use the `OnMove()` method, to update the platform's position (and its rotation if needed). Internally the `PlatformMovement` will compute the up-to-date platform's **velocity** and **angularVelocity** so the `CharacterMovement` can use it to move the character.

Moving a Platform with Animation

The procedure to create an animation based is similar to the one above described, where you create a `PlatformMovement` derived class and use its `OnMove` method to update it position (and rotation), however one important difference is the use of a **PlayableDirector to manually play the platform's animation and update our platform's position and rotation accordly**.



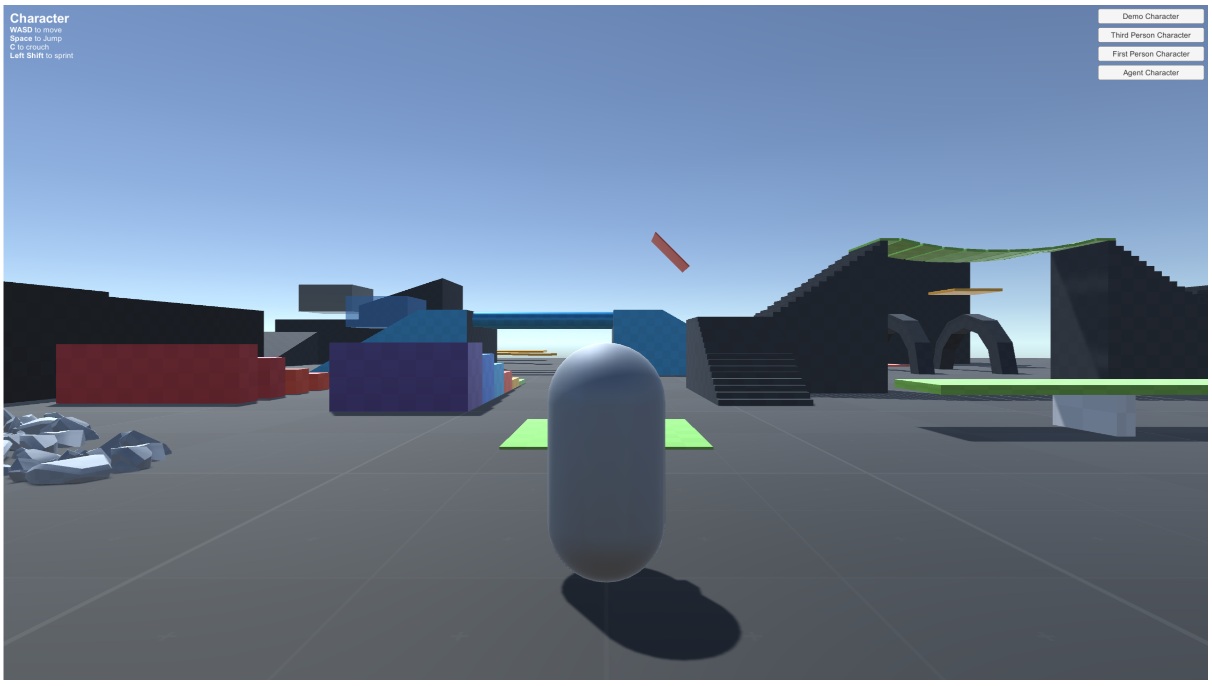
```
Transform
Position                 X -5   Y 1.5  Z 5
Rotation                 X 0    Y 0    Z 0
Scale                    X 1    Y 1    Z 1

Rigidbody
Mass                     1
Drag                     0
Angular Drag             0
Use Gravity              ☐
Is Kinematic             ✓
Interpolate              Interpolate
Collision Detection      Continuous Speculative
Constraints
Info
Speed                    0
Velocity                 X 0    Y 0    Z 0
Angular Velocity         X 0    Y 0    Z 0
Inertia Tensor           X 1    Y 1    Z 1
Inertia Tensor Rotation  X 0    Y 0    Z 0
Local Center of Mass     X 0    Y 0    Z 0
World Center of Mass     X -5   Y 1.5  Z 5
Sleep State              Awake

✓ Animator
Controller               None (Runtime Animator Controller)
Avatar                   None (Avatar)
Apply Root Motion        ☐
Update Mode              Animate Physics
Culling Mode             Always Animate
Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 0 (0.0%)

✓ Playable Director
Playable                 Platform Timeline (TimelineAsset)
Update Method            Manual
Play On Awake            ✓
Wrap Mode                Loop
Initial Time             0
Bindings

✓ Animated Platform (Script)
Script                   AnimatedPlatform
```

Additionally, please make sure your **Animator Update Mode** is set to **Animate Physics**.

```
public sealed class AnimatedPlatform : PlatformMovement
{
    private PlayableDirector _director;

    protected override void OnMove()
    {
        // Manually update animation

        _director.time = Time.time % _director.duration;
        _director.Evaluate();
        // Update our position and rotation with the animated transform
(modified by animation)

        position = transform.position;
        rotation = transform.rotation;
    }

    private void Awake()
    {
        // Cache PlayableDirector component

        _director = GetComponent<PlayableDirector>();
    }
}
```

The `Character` class lets you decide if a character should be affected by a moving platform and how through its properties and **delegates**. **Want to conserve the platform's momentum on leave?** Set `impartPlatformsVelocity` property value to true, **want to prevent the character from being rotated along the platform?** Set `impartPlatformRotation` property value to false, **want to increase the platform's momentum?** Use the `ImpartPlatformVelocity()` method to scale the `platformVelocity`.

Demo



This is a full feature example scene showing 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.

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**.

Unlike the previous ECM version, ECM2 will not automatically enable / disable root motion based on the character's movement mode / or state. As you can see in this example when a character jumps the character's will stop immediately, since the used jump animation does not include root motion support causing the character's velocity to be zeroed.

To overcome this "limitation" you can manually toggle root motion as needed based on the character's movement mode or its logical state or make use of root motion based jump, since now, root motion takes full control when enabled.

Root Motion Toggle

This expands the previous example showing 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 the steps to follow when creating a procedurally (movement controlled by a script) moving platform. This shows how to create a `PlatformMovement` derived class and use its `OnMove()` to perform the platform movement.

The same procedure should be followed while moving a platform using a Tween engine.

Animated Platform

This example shows the steps to follow when creating an animated (movement controlled by animation) moving platform. This shows how to create a `PlatformMovement` derived class and use its `OnMove()` to perform the platform movement.

This makes use of the **PlayableDirector** component to update the animation on demand.

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.

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 component-based the character's velocity (additive or override) complying with our new movement algorithm and should be preferred over setting character's velocity when possible. This does not modify the character's movement mode.

Dash

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

The dashing mechanic implemented here requires to set a dash duration (in seconds) and the desired dash impulse, once dash time is over, the character's returns to default state (e.g. Walking).

Worth note here, a character movement is always affected by its current physical settings (e.g. `MaxAcceleration`, `BrakingDecelerationWaling`, `GroundFriction`, etc.) for its current movement mode, so in order to perform our dash impulse without being affected by the current friction (Walking, Falling, etc.), we will make use of `brakingFriction` property; this, when enabled (`useSeparateBrakingFriction = true`) allows us to bypass the character's current friction with the one assigned to `brakingFriction`.

Additionally we'll make use of `OnMovementHit` to stop the dashing when the character hits a non-walkable surface (e.g. a wall).

Why not make Dashing a movement mode ?

Remember, a movement mode directly modifies the character's velocity and how it is updated. For example, when **Walking**, it ignores any input vertical velocity, reorient the character's velocity along the walkable surface, 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 the dash.

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

Advanced Dash

The dashing mechanic implemented here is a more complete implementation than the previous version, as this takes into consideration additional physical properties (max speed, braking deceleration, etc.), is the character able to dash? Disable character's rotation while dashing, etc.

The dashing ends on collision (e.g. Against a wall), if the dashing button is released or if the character's speed is less than `maxWalkSpeed`.

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`. One particular difference here is the 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.

`OnMove()`

Override the `OnMove` method to add your state handler, Sliding().

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.

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` accordly.

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.

Swimm

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.

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.

This basically works as follows, 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 accordly.

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 struct 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 accordly.

Worth note we make use of the `OnCustomMovementMode` method to handle our custom **Climbing** movement mode, and you should prefer this method to `OnMove` when adding custom movement modes for easiness and clarity.

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.

Change Gravity Direction

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

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.

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, it makes use of the new v1.0.1 TeleportPosition, SetYaw functions.

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

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.

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