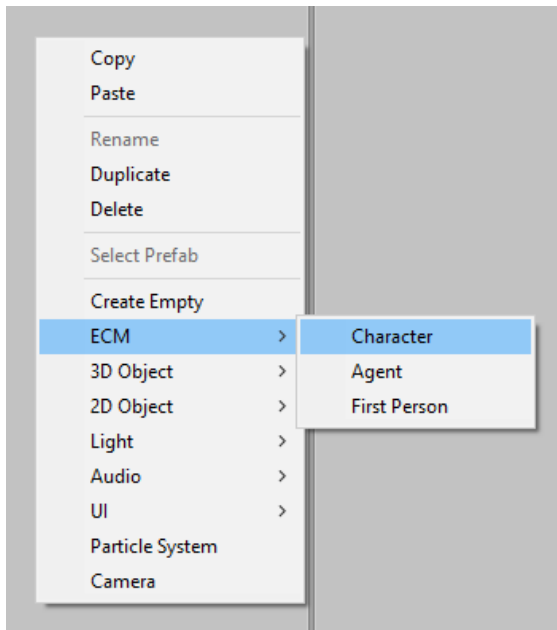
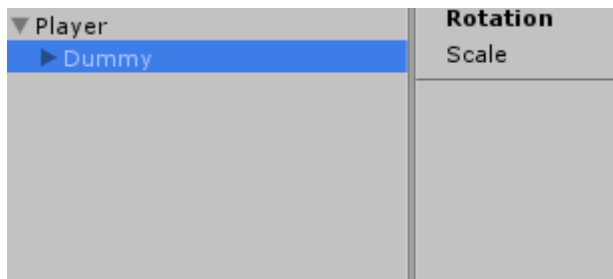
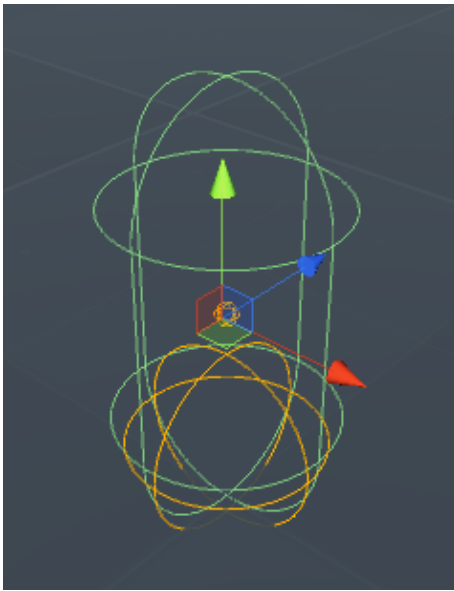


Quick-start guide

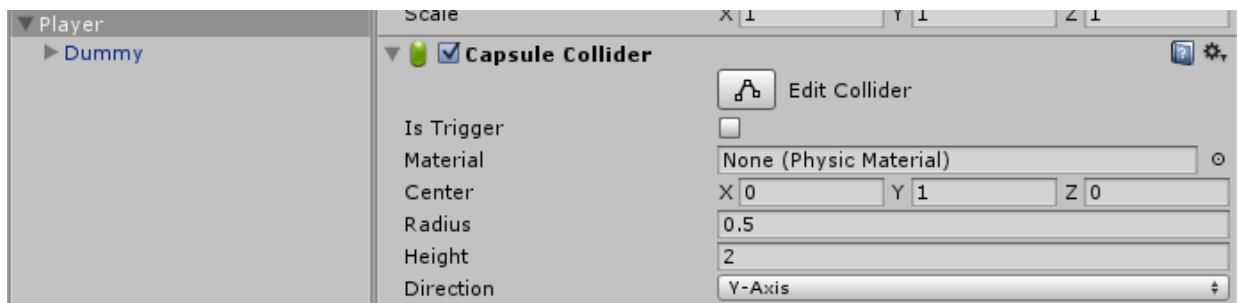
1. Import the Easy Character Movement package into your project.
2. Right-click on the hierarchy window to open the creation dialog and select the ECM option, then choose the type of character you want to create.



- It will create an *empty* (no visual representation) character named **ECM_Character**, **ECM_Agent** or **ECM_FirstPerson** based on your selection. Please make sure its origin is at 0,0,0. This will save troubles when parenting.



- Parent your character model (character's visual representation) to this new game object.



- Adjust the Capsule collider to match your model size.
- Done! You are ready to move around using the keyboard.

Foreword

Thank you for purchasing Easy Character Movement!

I've developed Easy Character Movement as part of my currently in development platformer game. In the end, I was so happy with the results that I decided to share it with the great unity community.

I sincerely hope this help you to make awesome games and have fun while doing it!

Support

I'm a solo developer and your feedback and support is greatly appreciated!

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

To get customer support, please include the invoice number for the purchase of the Asset from the Unity asset store.

Kind Regards,
Oscar

Overview

Easy Character Movement is a feature-rich high performance yet incredible easy to use Rigidbody based character controller.

It can be used for any kind of character, from Players to NPCs to Enemies, and for a wide range of games like, platformer, first person, third person, adventure, point and click, and more!

Easy Character Movement was designed from the ground up to be as efficient and flexible as possible with zero allocations after initialization. As a result, it runs great on all platforms including mobile.

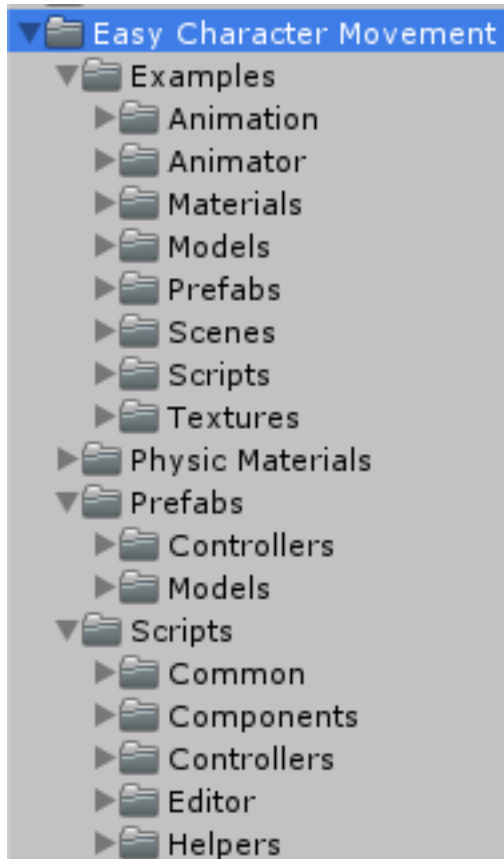
If you are looking for an easy, efficient and flexible character controller for your next project, please let Easy Character Movement be there for you.

Features

- Rigidbody-based character controller.
- Capsule based character colliders.
- Support for steps, the character can walk (if desired) on any surface up to 89 degrees.
- High performance, feature-rich ground detection component, capable of detect, report and query multiple grounding cases.
- Flat-base capsule bottom. This avoids the situation where characters slowly lower off the side of a ledge.
- Configurable ledge offset. This set how close / far a character can stand on a ledge without fall.
- Ground-snap. This help to maintain the character on ground no matter how fast it is running and not launch of ramps.
- Move and Rotate on dynamic platforms.
- Stay still on slopes.
- Keep same speed on straight segments and slopes.
- Slide on steep slopes (if desired).
- Base controller for Characters.
- Base controller for Agents (NavMeshAgent).
- Base controller for First-Person.
- Solid Root Motion support.

- Orient to ground slopes.
- Easy integration into existing projects.
- Fully commented C# source code. Clear, readable and easy to modify.
- Mobile friendly.
- Garbage-Collector friendly.
- and more!

Package Contents



Examples

This contains the assets required for the example / demo scenes. This is not required and should be omitted when importing ECM into your projects.

Physic Materials

Contains the frictionless physical material required for the character's collider.

Prefabs

This contains prefabs for the supplied base character controllers (**BaseCharacterController**, **BaseAgentController** and **BaseFirstPersonController**). This is optional and can safely be omitted when importing ECM into your projects.

Scripts

Contains the C# source code of the Easy Character Movement package.

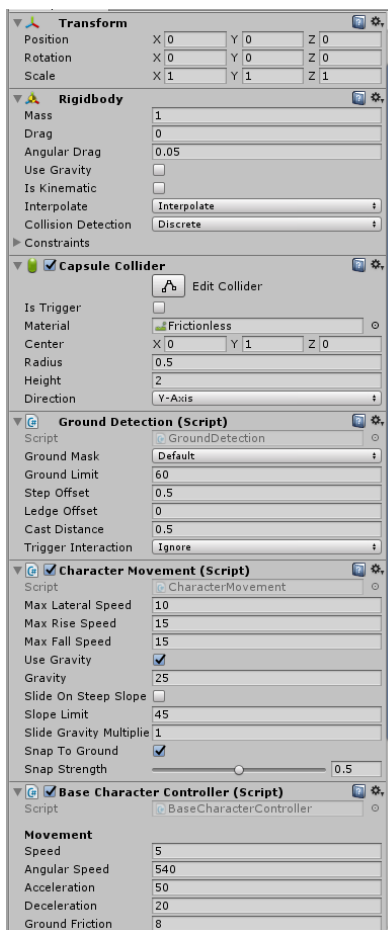
Description

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

The **CharacterMovement** component is the core of the ECM system and is responsible to perform all the heavy work to move a character (a.k.a. Character motor), such as apply forces, impulses, constraints, platforms interaction, etc. This is analogous to the Unity's character controller, but unlike the Unity's character controller, this make use of Rigidbody physics.

ECM has been developed to be easy and flexible to use, however to operate **CharacterMovement** requires a **Rigidbody**, a **CapsuleCollider**, and **GroundDetection** component) to be in the same GameObject where **CharacterMovement** is.

Here is a suggested set of components your characters should follow:



Along the *CharacterMovement* component, is the *GroundDetection*, which performs the ground detection and supply this information to the *CharacterMovement* component.

Above all, is the controller (eg: **BaseCharacterController**) which determines how the Character should be moved, such as in response from user input, AI, animation, etc. and feed this information to the *CharacterMovement* component, which perform the movement.

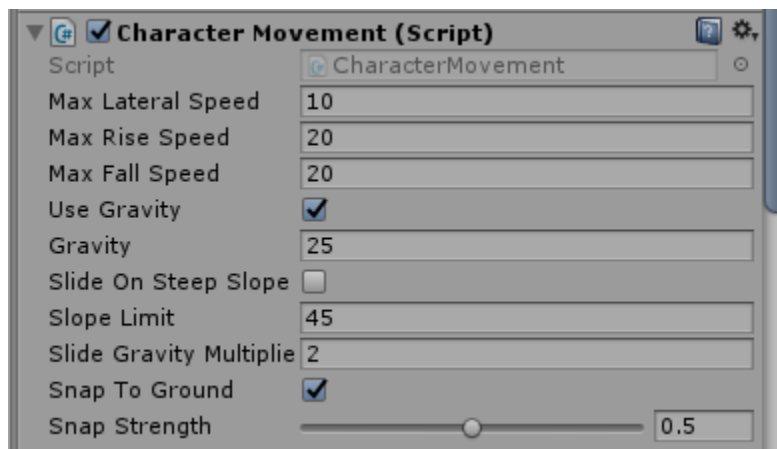
Worth note, that the *CharacterMovement* component by itself will do nothing, you must call its **Move** method to move the character.

Components

Character Movement

CharacterMovement is the core of the ECM system and is responsible to perform all the heavy work to move a character (a.k.a. Character motor), such as apply forces, impulses, constraints, platforms interaction, etc.

This is analogous to the Unity's character controller, but unlike the Unity's character controller, this make use of Rigidbody physics.



Max Lateral Speed

The maximum lateral speed this character can move, including movement from external forces like sliding, collisions, etc.

Max Rise Speed

The maximum rising speed, including movement from external forces like sliding, collisions, etc. Effective terminal velocity along Y+ axis.

Max Fall Speed

The maximum falling speed, including movement from external forces like sliding, collisions, etc. Effective terminal velocity along Y- axis.

Use Gravity

Enable / disable Character's custom gravity. If enabled, the character will be affected by its custom gravity force.

Gravity

The amount of gravity to be applied to this character. We apply gravity manually for more tuning control.

Slide on Steep Slope

Should the character slide down of a steep slope?

Slope Limit

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

Slide Gravity Multiplier

The amount of gravity to be applied when sliding off a steep slope.

Snap to Ground

When enabled, will force the character to safely follow the walkable 'ground' geometry.

Snap Strength

A tolerance of how close to the 'ground' maintain the character.

0 == no snap at all, 1 == 100% stick to ground.

BaseGroundDetection

This is an abstract class, and is the responsible of perform ground detection, and supply information about the "ground" such as the ground point (where characters touch the ground), the ground normal, and more.

This class can to be extended to perform your custom ground detection method if desired or required.

Included is a robust, feature-rich **GroundDetection** class which extends the abstract **BaseGroundDetection** class, this **GroundDetection** class is capable of detect and report a complete set of grounding information, such as is the character standing on a ledge? How far is the character from the ledge, is the character on a step? How high is the step? Is the character on a slope? Etc.

As you can see it offer a set of valuable data to better sync your character animations to your surrounding ground.

Additional, you can use the new method **ComputeGroundHit** to query the character's distance (plus additional data) at any time and at any position you like.

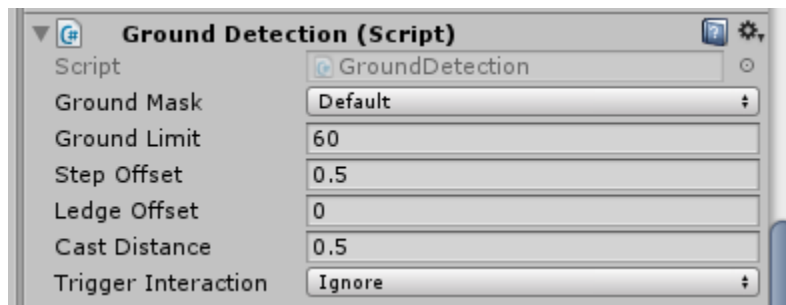
Also, there's a new helper method **SweepTest**, this will test if a character would collide with anything, if it was moved through the scene.

Worth note that you should not query / cache the BaseGroundDetection component, an instead, please use the **CharacterMovement** component exposed methods and properties.

Ground Detection

Extends the abstract **BaseGroundDetection** class and uses the character capsule's bottom sphere to represent the character's "feet".

This is a robust, feature-rich component capable of detect and report a complete set of grounding information, such as is the character standing on a ledge? How far is the character from the ledge, is the character on a step? How high is the step? Is the character on a slope? Etc. This is the responsible of supply this information the **CharacterMovement** component.



Ground Mask

Layers to be considered as ground (walkable).

Ground Limit

The maximum angle (in degrees) that will be accounted as 'ground'. Anything above is treated as a 'wall' and as a such, not walkable.

Step Offset

The maximum height (in meters) for a valid step. The character will step up a stair only if it is closer to the ground than the indicated value.

As rule of thumb, configure it to your character's collider radius.

Ledge Offset

The maximum horizontal distance (in meters) a character can stand on a ledge without slide down.

This allows the character to stand on a ledge (without slide down) a distance up to its capsule's radius.

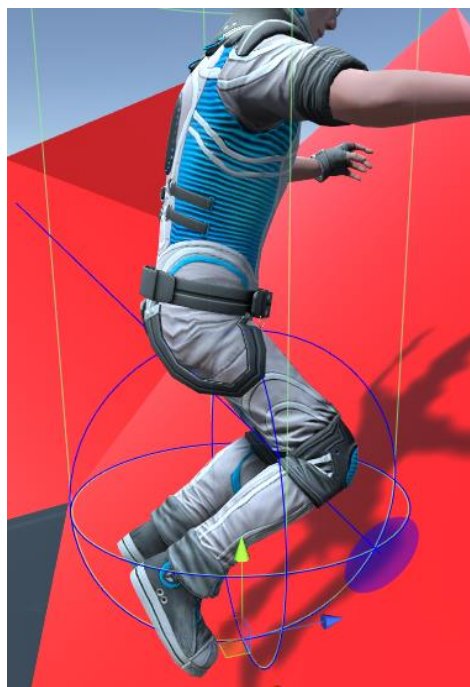
Cast Distance

Determines the maximum length of the cast when the character is on ground. Internally this is treated as a dynamic value based on character's grounding state, however this value will be used when the character is on ground, to help to maintain the ground.

As rule of thumb, configure it to your character's collider radius.

Grounding info

The new ***GroundDetection*** component adds a new property ***groundLimit*** to effectively separate walkable 'ground' from 'walls', thus there are cases where the character's capsule is touching the 'ground' but based on configured parameters that ground is '*invalid*', this 'invalid ground' is shown with a blue sphere at capsule's bottom.



By other hand, a character is considered grounded, when is on valid ground (any walkable 'ground' with a ground angle less than specified **groundLimit**) AND is on 'ground' (its capsule's bottom sphere is touching any 'ground').

You can easily query this state using the **CharacterMovement** **isGrounded** property or using its properties **isOnGround** && **isValidGround**.



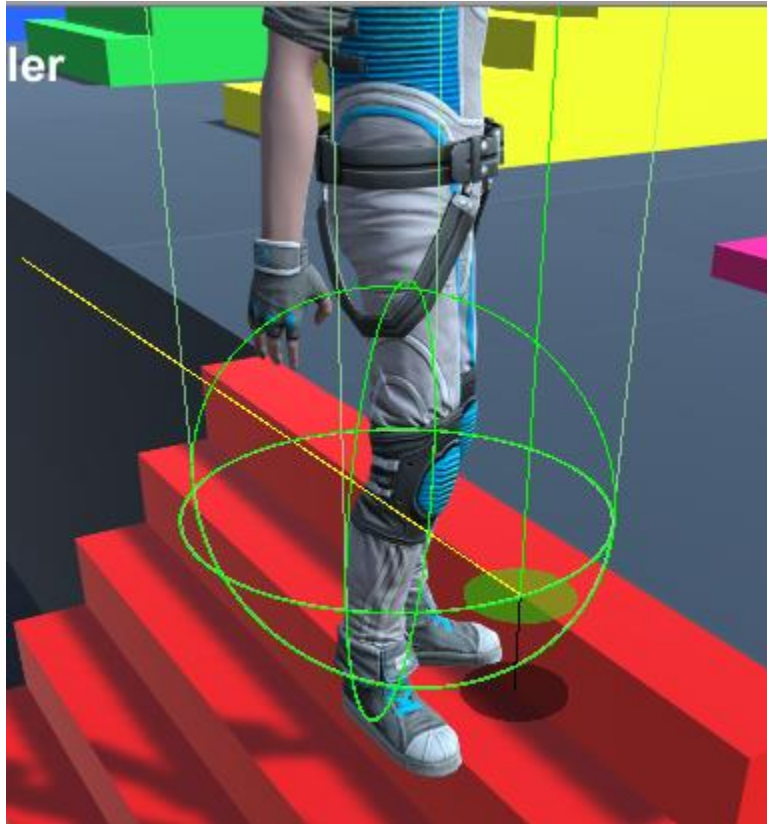
When the character is grounded, it will display a green sphere at capsule's bottom.

Additionally, you can query many of the character 'grounding' state info, using the **CharacterMovement** ground related properties, such as: **isGrounded**, **isOnGround**, **isOnPlatform**, **isOnLedgeSolidSide**, **isOnLedgeEmptySide**, etc.

Steps

ECM v1.6 adds the ability to climb steps, the character can climb a step with a maximum height of its collider radius (***stepOffset*** property).

You can check if a character is on a step, using the ***CharacterMovement*** property ***isOnStep***, it will also report the current step height with the property ***stepHeight***.



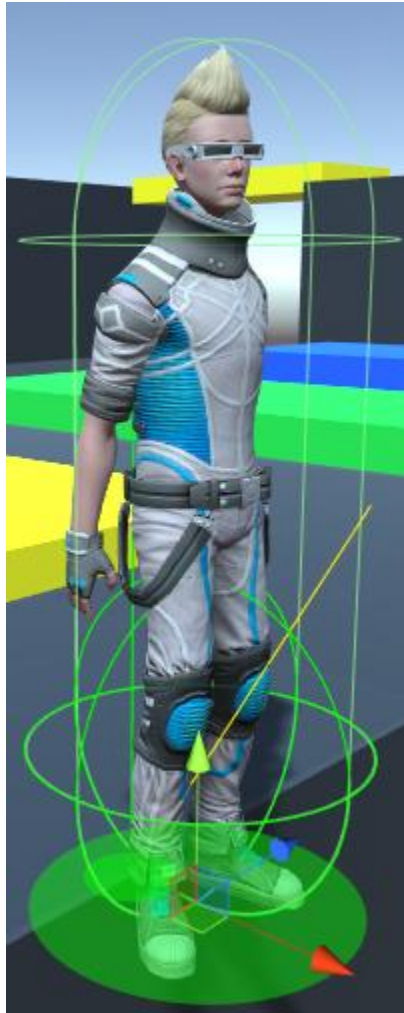
When on a step, it will show the step bottom ground point (black point), plus the step height (black line).

Ledges

One of the new features of ECM is the way it handle ledges, in this implementation, when a ledge is detected, the ***CharacterMovement*** will automatically treat the capsule's bottom as flat, instead of rounded. This avoids the situation where characters slowly lower off the side of a ledge (as their capsule 'balances' on the edge).

Additionally, you can now configure how far a character can stand on a ledge without slide down of it, using the new property **ledgeOffset**.

You can easily check if a character is standing on a ledge (on its 'solid' side), using the **CharacterMovement** property **isOnLedgeSolidSide** for example, you can use it to trigger a balance animation like the old Sonic games did.



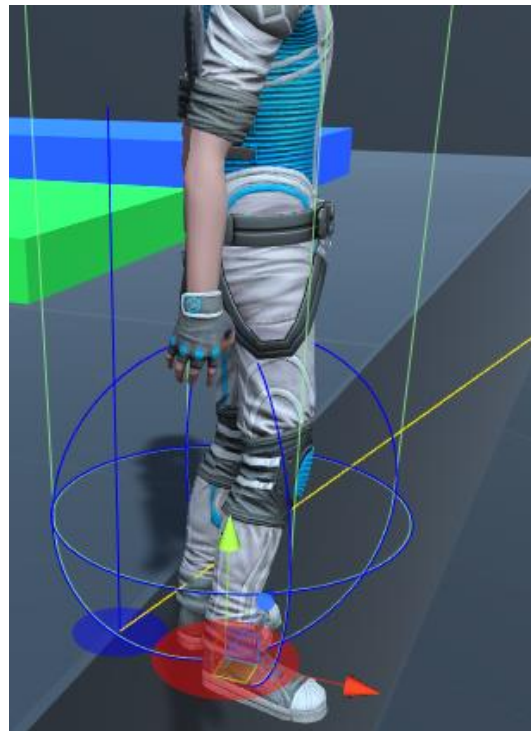
The green circle shows the character is on a ledge 'solid' side.

Other new ledge related properties are:

isOnLedgeEmptySide: Is this character standing on the 'empty' side of a ledge?

ledgeDistance: The horizontal distance from the character's bottom position to the ledge contact point.

By other hand when a character is on a ledge 'empty' side, it will show a red circle (of **ledgeOffset** radius), at character's bottom.



Controllers

The responsibility of the controller, as stated before, is determine how the Character should be moved, such as in response from user input, AI, animation, etc., and pass this information to the **CharacterMovement** component, which in response will perform the movement.

ECM offers 3 different Base (which can be extended) controllers, **BaseCharacterController**, a general-purpose character controller, **BaseAgentController**, a base class for *NavMeshAgent* controlled characters, and **BaseFirstPersonController**, a base class for typical first-person movement.

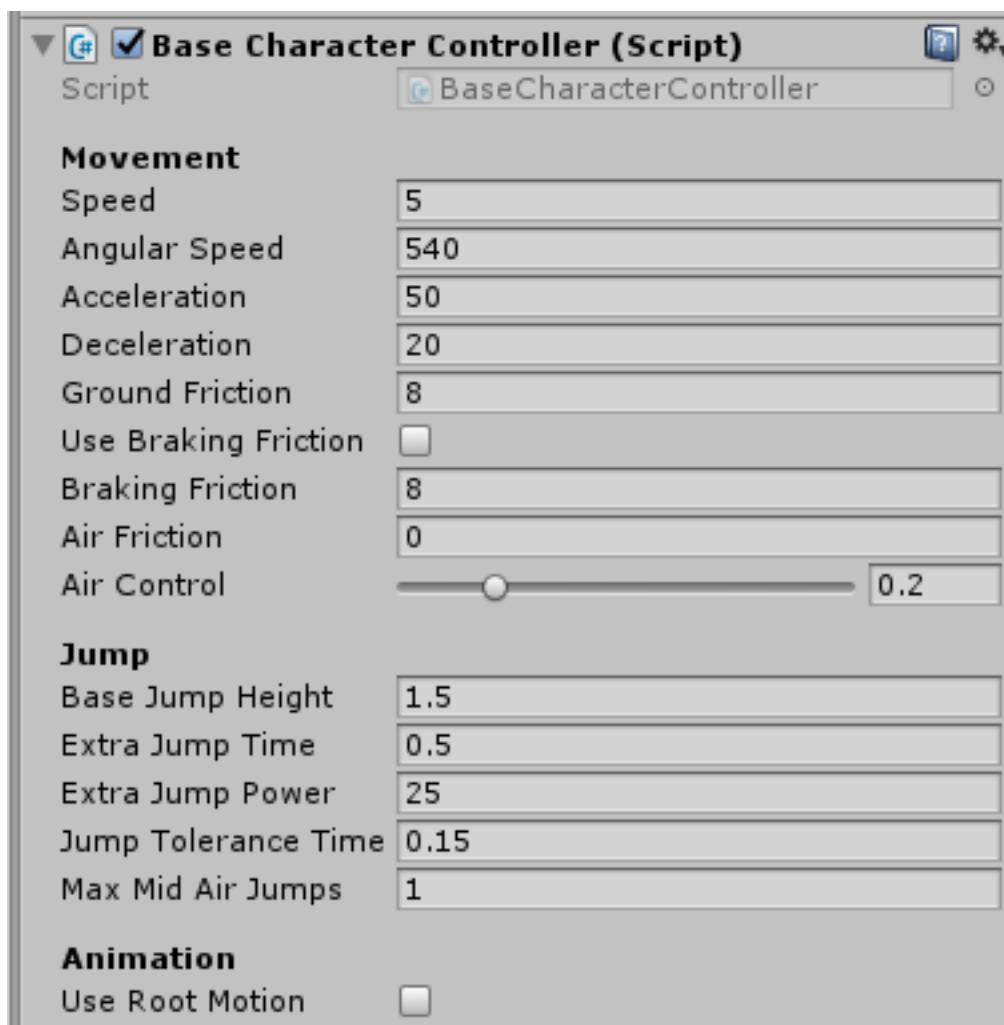
The suggested approach to work with ECM, is extend one of the supplied base controllers (eg: **BaseCharacterController**), creating a custom character controller derived from base controllers and add custom code to match your game needs (*please refer to the included examples*), because ultimately no one knows your game better than you!

Worth note that the use of supplied base controllers is suggested but not mandatory, it is perfectly fine (if preferred) create your own character controller and relay on the **GroundDetection** and the **CharacterMovement** components to perform the character movement for you, however, extending one of the 'Base' controllers will give you many features out of the box and should be preferred.

Base Character Controller

A general-purpose character controller and base of other controllers. It handles keyboard input, and allows for an accelerated friction-based movement, a variable height jump, and unlimited mid-air jumps (if desired), however this default behavior can easily be modified or completely replaced overriding its related methods in a derived class.

This is a robust class which among many features, it offers character *locomotion*, *variable height jump*, *unlimited mid-air jumps*, *root motion support*, etc., and should serve you as a strong foundation to develop your custom controllers on top.



Speed

Maximum movement speed (in m/s).

Angular Speed

Maximum turning speed (in deg/s).

Acceleration

The rate of change of velocity.

Deceleration

The rate at which the characters slows down.

Ground Friction

Setting that affects movement control. Higher values allow faster changes in direction. If *useBrakingFriction* is false, this also affects the ability to stop more quickly when braking.

Use Braking Friction

Should *brakingFriction* be used to slow the character? If false, *groundFriction* will be used.

Braking Friction

Friction coefficient applied when braking (when there is no input acceleration). Only used if *useBrakingFriction* is true, otherwise *groundFriction* is used.

Air Friction

Friction coefficient applied when 'not grounded'. This is analogous to *groundFriction*.

Air Control

When not grounded, the amount of lateral movement control available to the character. 0 == no control, 1 == full control. Defaults to 0.2.

Base Jump Height

The initial jump height (in meters).

Extra Jump Time

The extra jump time (E.g., holding jump button) in seconds.

Extra Jump Power

Acceleration while jump button is held down, given in meters / sec².

Jump Tolerance Time

How early before hitting the ground you can press jump, and still perform the jump. Typical values go from 0.05f to 0.5f.

Max Mid-Air Jumps

Maximum mid-air jumps. 0 disables mid-air jump.

Set this to Infinity for unlimited mid-air jumps!

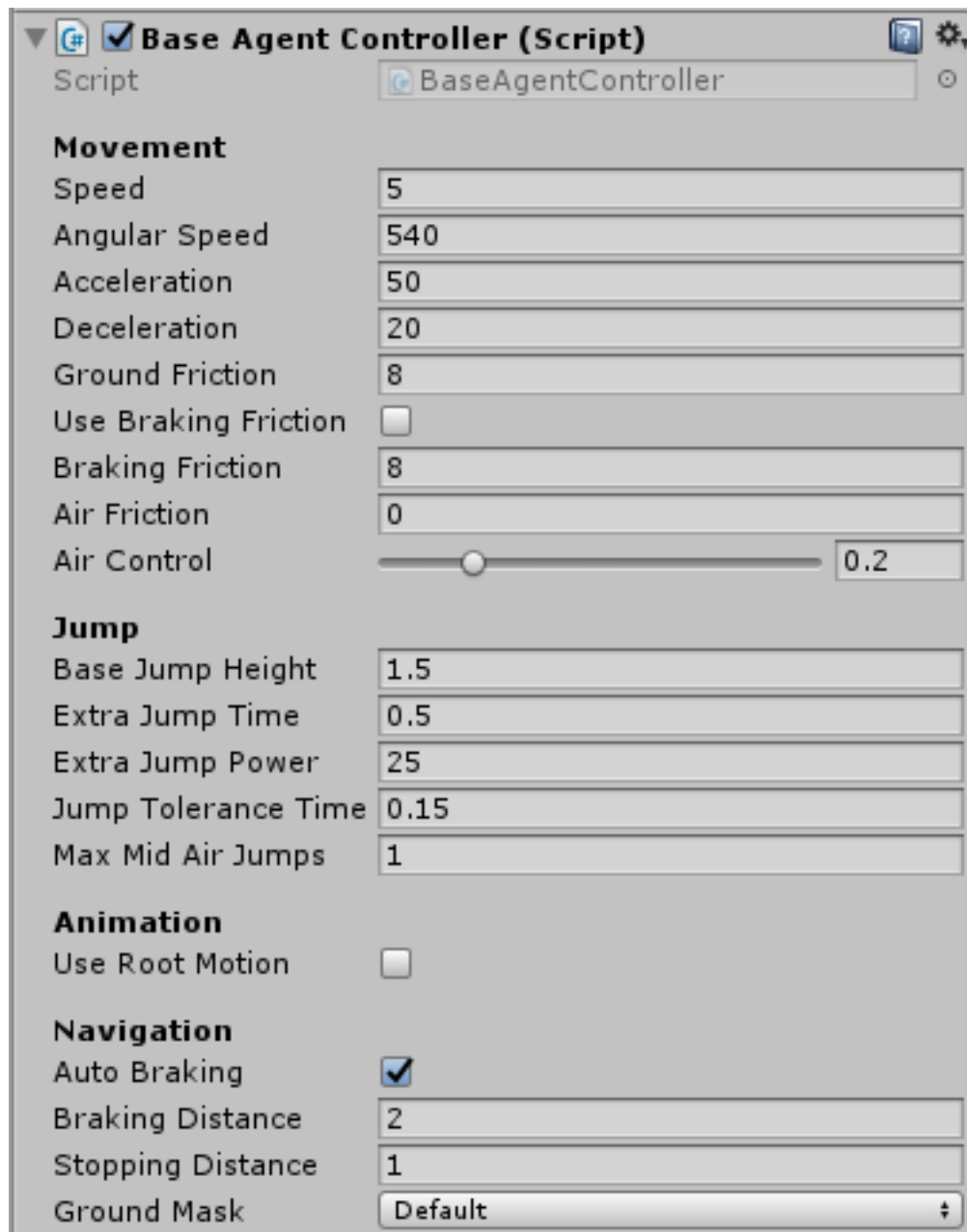
Use Root Motion

Should use root motion? If true, the animation velocity will override movement velocity. This requires a **RootMotionController** attached to the *Animator* game object.

Base Agent Controller

A base class for *NavMeshAgent* controlled characters. It inherits from **BaseCharacterController** and extends it to control a *NavMeshAgent* and intelligently move in response to mouse click (click to move).

Like the base character controller, this default behavior can easily be modified or completely replaced in a derived class.



The image shows the Unity Inspector window for the **Base Agent Controller (Script)**. The script is attached to the **BaseAgentController** component. The settings are organized into several sections:

- Movement**
 - Speed: 5
 - Angular Speed: 540
 - Acceleration: 50
 - Deceleration: 20
 - Ground Friction: 8
 - Use Braking Friction:
 - Braking Friction: 8
 - Air Friction: 0
 - Air Control: 0.2 (slider)
- Jump**
 - Base Jump Height: 1.5
 - Extra Jump Time: 0.5
 - Extra Jump Power: 25
 - Jump Tolerance Time: 0.15
 - Max Mid Air Jumps: 1
- Animation**
 - Use Root Motion:
- Navigation**
 - Auto Braking:
 - Braking Distance: 2
 - Stopping Distance: 1
 - Ground Mask: Default

Auto Braking

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

Braking Distance

Distance from target position to start braking, e.g. slowing area.

Stopping Distance

Stop within this distance from the target position.

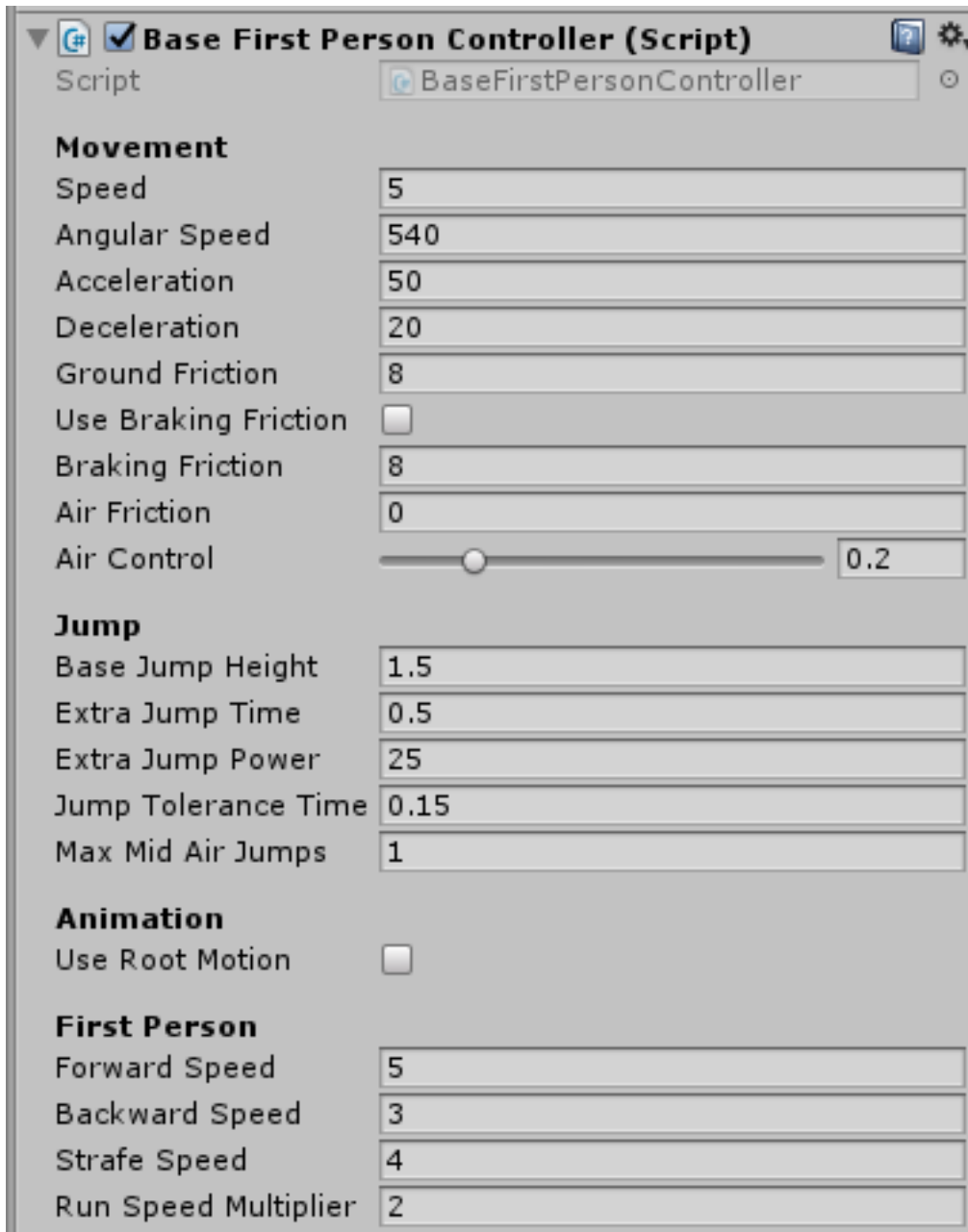
Ground Mask

Layers to be considered as ground (walkable). Used by ground click detection.

Base First Person Controller

A base class for a first-person controller. It inherits from **BaseCharacterController** and extends it to perform classic FPS movement.

Like the base character controller, this default behavior can easily be modified or completely replaced in a derived class.



Forward Speed

Speed when moving forward.

Backward Speed

Speed when moving backwards.

Strafe Speed

Speed when moving sideways.

Run Speed Multiplier

Speed multiplier while running.

Custom Controllers

As stated before, the suggested approach to work with ECM, is use one of the included base controllers and extend it to add game specific features, this way you can use the supplied features as a strong foundation to build your game on top, or if desired, modify it or even completely replace it without directly modify the ECM source code.

One of the many advantages of this approach is the separation of your game code from asset code, this way, when you need to update ECM, your game code will remain unaffected.

To create a custom controller is as simple as create a new class which extends one of the included base controllers, for example ***BaseCharacterController***, and override its desired methods.

```
public sealed class MyCharacterController : BaseCharacterController
{
    protected override void Animate()
    {
        // Add animator related code here...
    }
}
```

To use this newly created custom controller (***MyCharacterController***), all you must do is replace the ***BaseCharacterController*** component from your character `GameObject` with ***MyCharacterController***.

In the following example we create a custom controller and override its default implementation to make the character's movement relative to main camera instead of world-space relative.

As before, we extend the *BaseCharacterController* and override its methods, in this case, HandleInput method.

```
public sealed class MyCharacterController : BaseCharacterController
{
    public Transform playerCamera;

    protected override void Animate()
    {
        // Add animator related code here...
    }

    protected override void HandleInput()
    {
        // Handle your custom input here...

        moveDirection = new Vector3
        {
            x = Input.GetAxisRaw("Horizontal"),
            y = 0.0f,
            z = Input.GetAxisRaw("Vertical")
        };

        walk = Input.GetButton("Fire3");
        jump = Input.GetButton("Jump");

        // Transform moveDirection vector to be relative to camera view direction

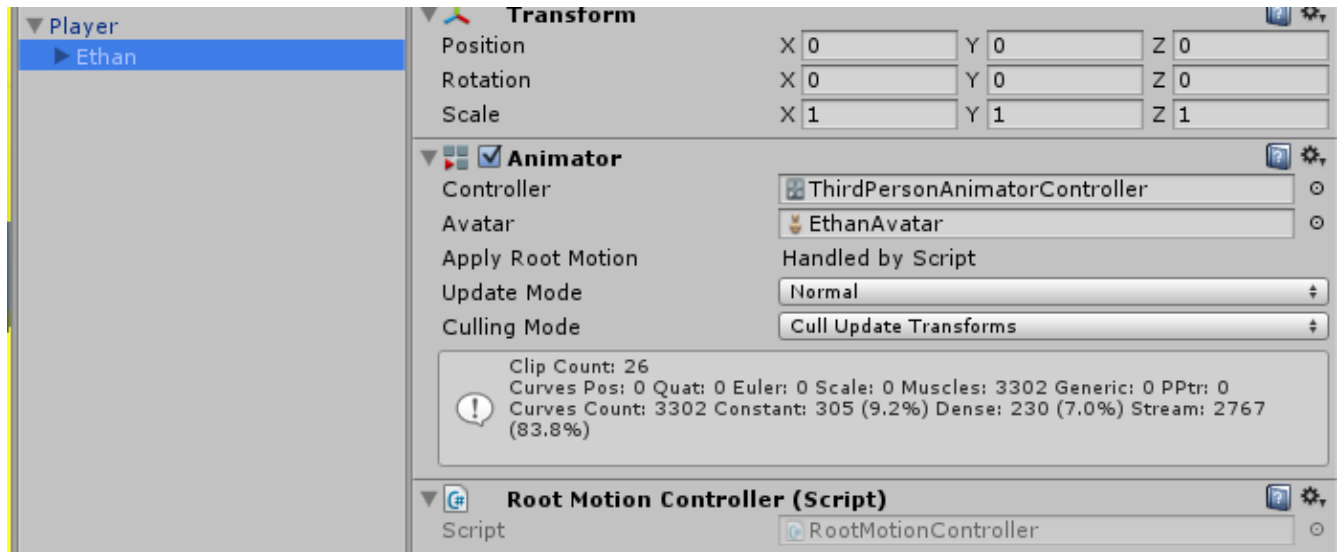
        moveDirection = moveDirection.relativeTo(playerCamera);
    }
}
```

For a more in-depth examples, please refer to the included custom controllers examples (*Easy Character Movement\Examples\Scripts\Controllers*) those are fully commented and should help you get started with ECM.

Helpers

Root Motion Controller

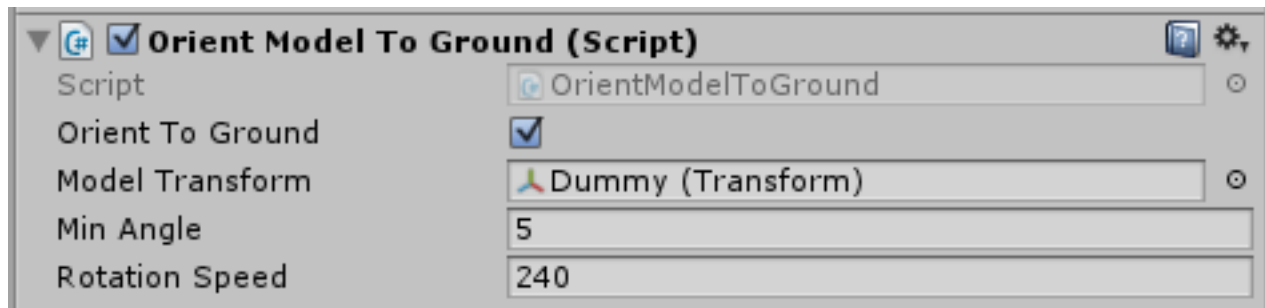
Helper component used to supply **Animator** root-motion velocity vector (**animVelocity**). It must be attached to the game object with the **Animator** component.



Orient Model to Ground (Optional)

Helper component used to orient a model to ground. This must be attached to the game object with **CharacterMovement** component, and the model to orient must be a child of this.

Please refer to prefabs (Examples/prefabs) if you are in doubt about the hierarchy your characters should follow.



Orient to Ground

Determines if the model transform will change its up vector to match the ground normal.

Model Transform

The transform to be aligned to ground. E.g. your child character model transform.

Min Angle

Minimum slope angle (in degrees) to cause an orientation change.

Rotation Speed

Maximum turning speed (in deg/s).

Prefabs

You can find a set of prefabs (one for each Controller) which you should use as a starting point or help if you are in doubt about the hierarchy your characters should follow.

The prefabs live in the Prefabs directory.

Code Reference

Please refer to the included source code to review the full code reference.