Controllers

The responsibility of the controller, as stated before, is to 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, 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.

TIP

As part of this guide, it is expected / recommended to follow each of the included examples source code, this is fully commented and complement the explanation found in this document.

Custom Controllers

The suggested approach to work with ECM, is use on 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 creating a new class which extends one of the included base controllers, for example ***BaseCharacterController***, and overrides its desired methods.

This is the most basic custom controller, this will use the default ECM functionality, just adding your character animator related code.

```
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***, as this will inherit all the BaseCharacterController functionality.

Custom Input

ECM handles all its input related code in its **BaseCharacterController HandleInput** method, in your custom controller (derived from one of the included base controllers) you can override its default implementation to add support for custom input solutions, like mobile input, Unity new Input System, etc.

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");

		// Replace the above code with your custom input code
	}
}
```

Movement Relative To Main Camera

A common solution in games is to move the character relative to the Main camera (the one tracking the character's) view direction, however, by default ECM input is handled in world space, its moveDirection vector is just fed with input horizontal axis in X, and input horizontal axis in Z.

Later (in the **BaseCharacterController CalcDesiredVelocity** method) ECM will use this given **moveDirection** vector (in this case fed by input), to compute the character's desired velocity, so in order to move the character relative to camera's view direction we can use the following approach

In your HandleInput method, simply transform the **moveDirection** input command to be relative to **Main camera**, using the included helper extension method:

```
moveDirection = moveDirection.relativeTo(mainCamera.transform);
```

Custom Rotation

By default ECM will rotate the character toward its given **moveDirection** vector, however as many of the ECM default functionality can easily be modified, overriding the correct method, in this case the **UpdateRotation** method.

In the UpdateRotation method, we simply modify the CharacterMovement rotation (its yaw rotation) with the rotation amount, however in order to move like a tank, we also need to modify the desired velocity, because as we have seen, by default it is just moveDirection * speed.

We use the **CalcDesiredVelocity** method to simply move the character towards its current forward direction.

You should use this approach when you need to modify the character's rotation and / or its desired velocity.

Crouching

ECM will handle the capsule collider scaling when a character isCrouching and restore it to its standing height when un-crouched, however it will not modify your character's model as it can be accomplished in different ways. For example using an Animation just like the Ethan character does.

The above statement is not true when using the FirstPersonCharacterController, by default in its (**BaseFirstPersonController AnimateView** method) it will scale the camera rig's pivot transform in order to perform a basic crouch animation, so in order to replace it, you must override its AnimateView and perform a custom camera animation for the crouch state.

The included example, replace the default (scale based) crouch animation by a position based, simply moving up / down the camera's pivot transform.

Additionally this shows how you can follow the same approach to modify the **BaseFirstPersonController** and **BaseAgentController** methods, because in the end all of those are just extending the **BaseCharacterController**!

Fall Damage

At this point you should be familiar enough with what methods should be used in order to modify / extend the default ECM functionality adding your game custom mechanics on top of it.

In this example, we show how to easily add fall-damage mechanics. Here we keep track of the character's last grounded position, and when the character has just landed (eg: **!movement.wasGrounded && movement.isGrounded**) , 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.

An important point to consider in this example, is the modification of the **BaseCharacterController Move** method, this is the method which (as its name suggests) handles all the character's movement, jump, etc. And like the previous methods this can be extended to add your game custom mechanics on top of it, or completely replace it if needed.

Toggle Gravity Direction

One of new features in v1.8 is the ability to modify the gravity direction, this allows to perform unique game mechanics and actually is pretty easy to do.

In this example, we simply will toggle the gravity direction along up and down, this will also rotate the character to match the new gravity direction. When an ECM 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).

To accomplish the rotation, once again we override the **UpdateRotation** method in order to rotate the character against the new gravity direction.

Orient To Ground

This example is related to the previous one, but here we simply orient the character along the ground normal, this way the character models can follow the terrain orientation.

As with the previous example, we use the **UpdateRotation** method to get the ground normal, and rotate the character's to match the ground orientation.

Once again, keep in mind, once the character is rotated, all its movement will be relative to its current up vector, for example if the character is on a 30 degrees plane, once rotated to match this plane's normal, internally the character's will see this as a flat plane as its new up direction is the plane's normal.

Platforms

By default an ECM character will automatically interact with a platform without further interaction on your part, however the platform must be a **kinematic rigidbody**.

The platforms (a Kinematic Rigidbody) can be animated as you prefer, for example, by script using **Rigidbody MovePosition** as I did in the included examples, by **Animator**, in this case the requirement is to set the **Animator** update mode to **Animate Physics** in order to correctly interact with the ECM Character.
The same applies for motion tweened, this library usually includes the option to **Animate Physics**, which should be used in order to correctly interact with an ECM character.

Wall Grab and Wall Jump

For this example we rely on OnCollisionXXX events to detect when the character collides with a grabbable wall. When a grabbable wall is detected and the character can grab it, the character will enter a 'grabbed state'.

While the character is in a grabbed state, we limit the character's max fall speed in order to simulate a grabb drag and allow him to perform a wall jump. Wall jump is a custom jump implementation.

Additionally to the wall grab, wall jump mechanics, this example shows how to implement a one-way platform.

The one-way platform uses a trigger area below the actual platform collider, when the character enters this trigger below the platform, the one-way platform script will disable the platform collisions allowing the character to jump to the platform from below it, by other hand, when the player leaves this trigger area (eg: player on top of platform), it will enable the platform collider allowing the character to step on top of it.

Flying

This example shows an important aspect of ECM, by default an ECM character will treat all its movement as planar movement, this means it discards the vertical component of its given desired velocity vector. In order to allow any vertical movement (eg: climb, fly, swim, etc), you'll need to explicitly tell ECM you want to use vertical movement using its **BaseCharacterController allowVerticalMovement.**
When you set **allowVerticalMovement** to **true**, internally it will disable the character's gravity, allowing the character to move freely in the air (or water, in case of swimming).

To enter in flying state, simply press the jump key (space bar by default) when the character is in the air (like a double jump), this will enable the **allowVerticalMovement** and add vertical movement to the **moveDirection** vector.

When the character is **flying**, you can use the **E key** to move down, while pressing the **jump key** will move the character up. The character will automatically leave the flying state once it becomes grounded.

Swimming

This example is really similar to the previous one (flying) as flying and swimming actually are really similar in terms of implementation, the main difference is how enter / exit swimming state.

In order to enter a swimming state, the example will detect if the whole capsule volume is inside our current water zone, in this case we'll check if the point in top of the capsule is inside the water zone, if it is, the character enter its swimming state, and if not, the character exits the swimming state.

The example will make use of a new (v1.8) feature **OverlapCapsule** method, which makes easy to fast overlaps checks without needing to rely on OnTrigger events, so it's easier to integrate into your code.

Additionally the example makes use of the new (v1.8) **DisableGroundDetection** and **EnableGroundDetection**, in order for the character to touch the water zone ground without being snapped back to it. Worth note this function should be used in pairs, because unlike DisableGrounding, which **temporarily** disables grounding phase, **DisableGroundDetection** will permanently disable it, until you explicitly call **EnableGroundDetection.**

Over The Shoulder Camera Movement

In this example we perform a movement similar to those in Resident Evil 4, where the camera is positioned behind the character, closely following all its movement. The character is rotated along its yaw axis (its up axis) using the mouse lateral motion, while the camera controls the pitch rotation (up / down movement) using the mouse vertical motion.

This example includes a basic over the shoulder camera controller, however worth note this is mostly as a pure example and not expected to be used in production.

In order to perform this over the shoulder camera movement, first the input **moveDirection** vector is transformed to be relative to the character's view direction (**HandleInput** method) , this allows the character to perform a strafe movement, while moving the character forward and backwards along its current view direction.

We replace the default ECM rotation implementation (**UpdateRotation** method) , rotating the character along its yaw axis using the mouse lateral movement.

Lastly, in the camera controller, we handle the camera's pitch rotation (up / down).

As with previous examples, please refer to the example source code, as it is fully commented and should be followed along this guide.

Mesh Analysis

While this is not an ECM example, it's a helper utility which allows to visualize the walkable ground.

ECM will limit its movement based on the angle (wrt its up axis) of the surface the character is standing on or moving to, while the character's shows a grounding gizmo in order to make it clear where can and can not move.

It's not always easy to catch a potential issue in irregular meshes, because sometimes these meshes include tiny triangles or zones of 'not walkables' ground causing issues with ECM, this is where the included Mesh Analysis helper enters.

This Mesh Analysis helper receives a mesh collider and will show the not-walkable (higher than given threshold angle) in a red color, additionally using the script context menu, you can completely paint the mesh (using its vertex colors, and require a shader which supports it). Worth note this mesh (real mesh, not mesh collider) colorization only

will help it. It's the same mesh used by your mesh collider, otherwise it will show incorrect results.

To use the Mesh Analysis simply add it to the game object with your MeshCollider / Mesh and set the desired max walkable ground threshold angle (the character's max walkable ground angle), and tick the Draw Gizmos option, this will show the normals of the MeshCollider geometry.

To colorize the mesh, use the script context menu, and select the PaintMesh, this will use the mesh vertex color to display all the walkable to non-walkable ground in a color gradient.