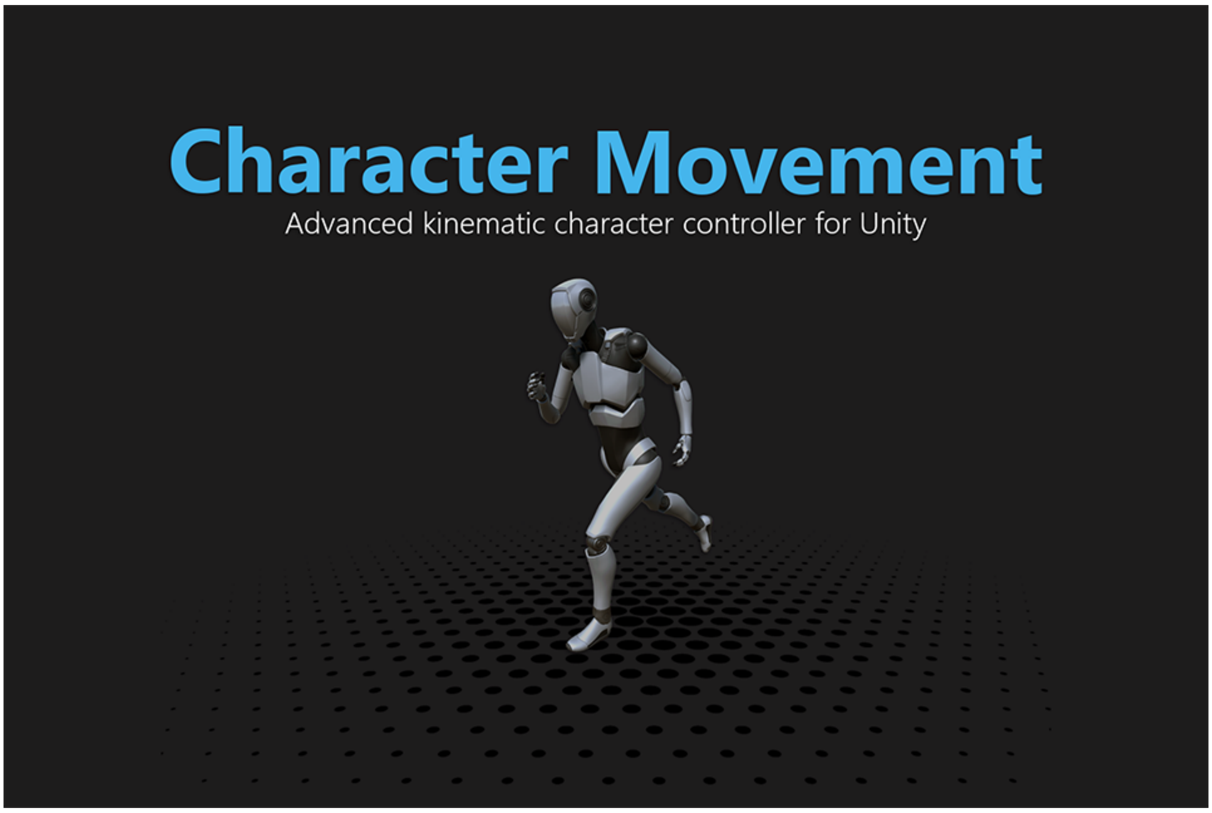
Character Movement

Advanced kinematic character controller for Unity

Walkthrough

Version 1.0.0



Overview

This document will guide you through the steps needed to implement a complete player controlled character (*e.g. Player*) from scratch using the *CharacterMovement* component as our character controller.

This should serve you as a starting point to get you familiar with all the *CharacterMovement* features and let you write your own game movement and mechanics.

Worth note that while the player controlled character created through this guide is fully functional it is not meant to be as a game ready character but developed as an example with learning purposes.

What is a character controller ?

A **Character Controller** allows you to easily do movement constrained by collisions without having to deal with a dynamic rigidbody.

A **Character Controller** is not affected by forces and will only move when you call its *Move* function. It will then carry out the movement but be constrained by collisions.

In the past, games did not use a *'real'* physics engine like the **PhysX SDK** (Unity's underlying physics engine). But they still used a character controller to move a player in a level. These games, such as **Quake** or even **Doom**, had a dedicated, customized piece of code to implement collision detection and response, which was often the only piece of physics in the whole game. It actually had little physics, but a lot of carefully tweaked values to provide a good feeling while controlling the player. The particular behavior it implemented is often called the '**collide and slide**' algorithm, and it has been 'tweaked for more than a decade'.

The *CharacterMovement* component is an implementation of such an algorithm, providing a robust and well-known behavior for character control.

Creating our Player

TIP

It is recommended to follow each section of this guide along with its companion scene and example source code.

1.- Create an empty *GameObject* and rename it as **Player**.

2.- Drag and drop the **ECM2_Capsule** *prefab* (`CharacterMovement\Samples\Shared Assets\Prefabs`) into your **Player** *GameObject* so it becomes a child of it. This will be our character's visual representation.

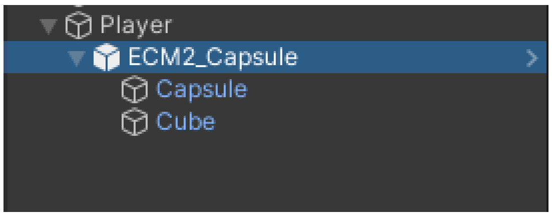


Fig. Player visual representation (our capsule).

3.- Add an empty *GameObject* as a child of our **Player** *GameObject*. Call it **Camera Follow** and position it at 0, 1, 0 (capsule's center). This will be our *Camera* follow target.

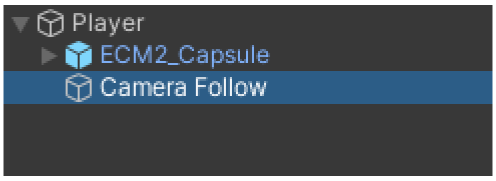
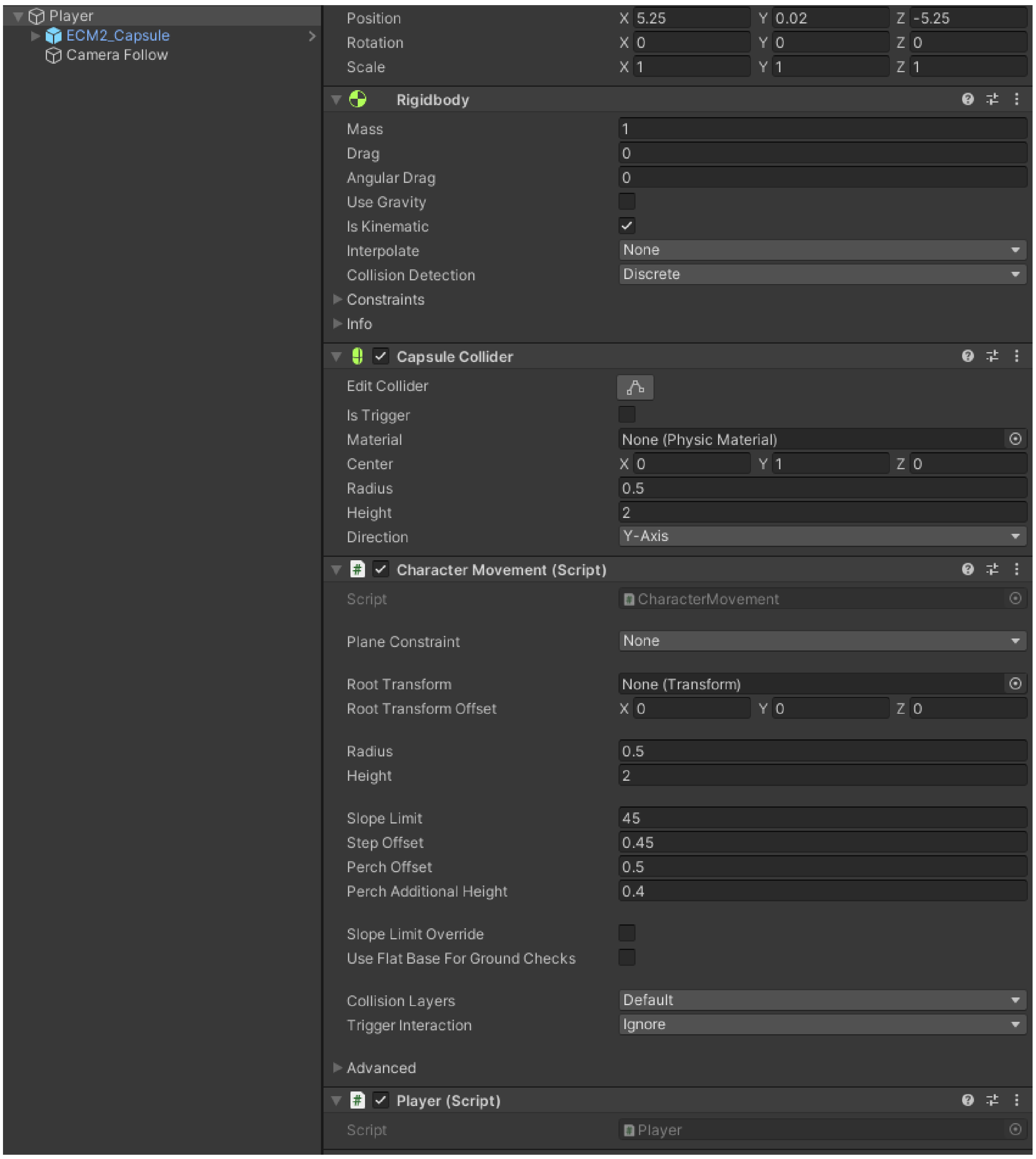


Fig. Camera's follow target

4.- Add the *CharacterMovement* component to our newly created **Player** *GameObject*. It will add its required components for proper operation.

3.- Create a C# Script named **Player**.

4.- Add this newly created **Player** script to your **Player** *GameObject*. At this point we have a complete armature for our player character.



Player
- ECM2_Capsule
- Camera Follow

Position	X 5.25	Y 0.02	Z -5.25
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	None
Collision Detection	Discrete
Constraints	
Info	

Capsule Collider ✓

Edit Collider	
Is Trigger	☐
Material	None (Physic Material)
Center	X 0 Y 1 Z 0
Radius	0.5
Height	2
Direction	Y-Axis

Character Movement (Script) ✓

Script	CharacterMovement
Plane Constraint	None
Root Transform	None (Transform)
Root Transform Offset	X 0 Y 0 Z 0
Radius	0.5
Height	2
Slope Limit	45
Step Offset	0.45
Perch Offset	0.5
Perch Additional Height	0.4
Slope Limit Override	☐
Use Flat Base For Ground Checks	☐
Collision Layers	Default
Trigger Interaction	Ignore
Advanced	

Player (Script) ✓

Script	Player

Fig. Player hierarchy and components.

5.- Lastly, drag and drop the *SimpleCameraController* (`CharacterMovement\Samples\Shared Assets\Scripts`) to the scene *Camera* and assign the **Player's Camera Follow** *GameObject* as the *SimpleCameraController Target*, so the camera follows our **Player** around.

Writing the Player script

Input

In order to be able to move our player around, first we need to know the direction we want to move, for it we read the current *horizontal* and *vertical* input values and convert them into a movement direction vector in *world-space*.

Add a new *HandleInput* method the **Player** class and add the following code into it:

```
// Read Input values

float horizontal = Input.GetAxisRaw($"Horizontal");
float vertical = Input.GetAxisRaw($"Vertical");

// Create a movement direction vector (in world space)

movementDirection = Vector3.zero;

movementDirection += Vector3.forward * vertical;
movementDirection += Vector3.right * horizontal;

// Make Sure it won't move faster diagonally

movementDirection = Vector3.ClampMagnitude(movementDirection, 1.0f);
```

We use the *horizontal* and *vertical* axes to create a *horizontal* (*XZ* plane) movement direction vector. Make sure it won't move faster on diagonals clamping its magnitude to 1.

Worth noting the movement is in *world-space*, where the *horizontal* input axis is mapped to the world’s *X-Axis* and the *vertical* input axis is mapped to the world’s *Z-Axis*. Often it's desired to move the player relative to the camera’s view direction (third person, first person, etc). Well see that later on this guide, but for now this will suffice.

Rotating Our Player

We’ll orient our character towards its current movement direction:

Add a new *UpdateRotation* method the **Player** class and add the following code into it:

```
// Rotate towards character's movement direction

characterMovement.RotateTowards(movementDirection, rotationRate *
Time.deltaTime);
```

While it's perfectly fine to directly modify your character’s rotation (*e.g. transform.rotation*) to rotate your character as your game needs, for this example we will use the *CharacterMovement RotateTowards* function for simplicity.

Moving our Player

One important characteristic of a **kinematic character controller**, is that it **will only ever move until its** *Move* **function is called**.

The *Move* function is the core of the *CharacterMovement* component, it's here where the aforementioned '*collide-and-slide*' algorithm takes place (along its other features), or in other words, **where the magic happens!**

The *CharacterMovement* component includes two *Move* functions:

Declaration

```
public CollisionFlags Move(Vector3 newVelocity, float deltaTime = 0.0f);
```

Parameters

newVelocity The updated velocity for the current frame. It is typically a combination of vertical motion due to gravity and horizontal motion when your character is moving.

deltaTime This is an optional parameter **primarily used to perform tick based simulations**. If omitted, it defaults to *Time.deltaTime*.

Returns

CollisionFlags. It indicates the direction of a collision: *None, Sides, Above, and Below.*

Description

Moves the character along the given velocity vector. This performs collision constrained movement resolving any collisions / overlaps found during this movement.

Declaration

```
public CollisionFlags Move(float deltaTime = 0.0f);
```

Parameters

deltaTime This is an optional parameter **primarily used to perform tick based simulations**. If omitted, it defaults to *Time.deltaTime*.

Returns

CollisionFlags. It indicates the direction of a collision: *None, Sides, Above, and Below*.

Description

Moves the character along its current *velocity* vector. This performs collision constrained movement resolving any collisions / overlaps found during this movement.

We'll be using the latter to perform our movement.

Ground Detection

A particularly **important feature** of the *CharacterMovement* component **is its ability to perform proper ground detection**. This, as its name suggests, determines if the character is standing on *ground* (*e.g.* its capsule's bottom hemisphere is colliding with 'something') and if it is able to walk on found *ground*.

This is a vital part for character movement as it determines how it will be moved depending if it is moving on *walkable ground*, is sliding off a *non-walkable* slope or is falling through the air under the effects of gravity.

We will make use of the *Character Movement isGrounded* property to correctly handle our *grounded* and *not-grounded* movement.

Add a new *Move* method to the **Player** class and add the following code into it:

```
// Create our desired velocity using the previously created movement
direction vector

Vector3 desiredVelocity = movementDirection * maxSpeed;

// Update character's velocity based on its grounding status

if (characterMovement.isGrounded)
    GroundedMovement(desiredVelocity);
else
    NotGroundedMovement(desiredVelocity);

// Perform movement using character's current velocity

characterMovement.Move();
```

Grounded Movement

We'll implement a simplistic (but effective) *acceleration* and *deceleration* model to be easy to remove, replace, or iterate upon.

Add a new *GroundedMovement* method to the **Player** class and add the following code into it:

```
characterMovement.velocity = Vector3.Lerp(characterMovement.velocity,
desiredVelocity, 1f - Mathf.Exp(-groundFriction * Time.deltaTime));
```

This *accelerates* or *decelerates* the character's current velocity towards our given desired velocity using the current *groundFriction* value.

Not-Grounded Movement

Similar to the *grounded* movement, we will implement a basic *acceleration* (*desiredVelocity != Vector3.zero*) and *deceleration* (*desiredVelocity == Vector3.zero*) model for our *not-grounded* movement, however in this case, **it's important to separate the horizontal and vertical components of the velocity so the character can freely move side to side while preserving the gravity effects.**

Add a new *NotGroundedMovement* method to the **Player** script and add the following code into it:

```
// Current character's velocity

Vector3 velocity = characterMovement.velocity;

// If moving into non-walkable ground, limit its contribution.
// Allow movement parallel, but not into it because that may push us up.

if (characterMovement.isOnGround &&
    Vector3.Dot(desiredVelocity, characterMovement.groundNormal) < 0.0f)
{
    Vector3 groundNormal = characterMovement.groundNormal;
    Vector3 groundNormal2D = groundNormal.onlyXZ().normalized;

    desiredVelocity = desiredVelocity.projectedOnPlane(groundNormal2D);
}

// If moving...

if (desiredVelocity != Vector3.zero)
{
    // Accelerate horizontal velocity towards desired velocity

    Vector3 horizontalVelocity = Vector3.MoveTowards(velocity.onlyXZ(),
desiredVelocity, maxAcceleration * airControl * Time.deltaTime);

    // Update velocity preserving gravity effects (vertical velocity)

    velocity = horizontalVelocity + velocity.onlyY();
}

// Apply gravity

velocity += gravity * Time.deltaTime;

// Update character's velocity

characterMovement.velocity = velocity;
```

The movement works by accelerating the *horizontal velocity* (*XZ*) towards our *desired horizontal velocity* (*XZ*) only, so we preserve the gravity effects acting along the *vertical velocity* (*Y*).

Lastly, we compute our final velocity (adding both *vertical* and *horizontal* components) and update character's velocity.

We make use of the extension methods *onlyXZ()* and *onlyY()* to separate the velocity components; these are defined in the *Extensions.cs* (`CharacterMovement\Source\Common\`) file and helps to improve code readability.

One particularly important part of the above code is preventing moving into a *non-walkable* surface as this helps to better constrain the character movement to *walkable* areas only.

Crouching

In this section we'll see how to implement a crouching mechanic.

Let's start adding the following code to our **Player** class, so we can know if we want to crouch and if it is already crouching or not:

```
public bool crouch { get; set; }

public bool isCrouching { get; protected set; }
```

Add the following code to the *HandleInput* method:

```
crouch = Input.GetKey(KeyCode.LeftControl) || Input.GetKey(KeyCode.C);
```

Add a new **Crouching** method to the **Player** class and add the following code into it:

```
private void Crouching()
{
    if (crouch)
    {
        if (isCrouching)
            return;

        characterMovement.SetHeight(crouchingHeight);

        isCrouching = true;
    }
    else
    {
        if (!isCrouching)
            return;

        if (!characterMovement.CheckHeight(standingHeight))
        {
            characterMovement.SetHeight(standingHeight);

            isCrouching = false;
        }
    }
}
```

Lastly, call the *Crouching()* method from our *Move()* method.

TIP

It's suggested to handle all your character movement and states before calling the CharacterMovement Move function.

When updating the character's volume, it is important to note that volumes are directly modified without any extra tests, and thus it might happen that the resulting volume overlaps some geometry nearby.

For example when resizing the character to leave a crouch pose, *i.e.* when the size of the character is increased, it is important to first check that the character can indeed *'stand up'*: the volume of space above the character must be empty (collision free). It is recommended to use the overlap queries for this purpose, in this case we use the *CharacterMovement CheckHeight* function to make sure our character can stand up.

Jumping

At this point we can freely move our character around, crouch and fall from surfaces, however it's not very funny, we'll fix that by adding a jump!

Add the following code to your *HandleInput* method so we can know when a player wants to jump:

```
jump = Input.GetButton($"Jump");
```

Ground Constraint

The *ground-constraint* helps to keep the player 'glued' to *walk-able* surfaces, prevent it from climbing / accessing *non-walk-able* surfaces, bounce when walking down-hill, or even being launched off ramps when running at high speeds. However, due to this constraint, **it is necessary to manually disable / pause it when a character is allowed to leave *ground***, *e.g. jumping*, *flying*, *climbing*, etc.

You can use the *Character Movement constrainToGround* property to disable and enable the *ground-constraint* or the *PauseGroundConstraint(float unconstrainedTime = 0.1f)* method to temporarily disable the ground constraint, as it will automatically re-enable it when *unconstrainedTime* runs out.

Add a new *Jumping* method to the **Player** class and add the following code into it:

```
if (jump && characterMovement.isGrounded)
{
    // Pause ground constraint so character can jump off walkable-ground

    characterMovement.PauseGroundConstraint();

    // perform the jump

    Vector3 jumpVelocity = Vector3.up * jumpImpulse;

    characterMovement.LaunchCharacter(jumpVelocity, true);
}
```

Here we make use of the *PauseGroundConstraint* to temporarily disable the *ground-constraint* so the character can jump, and make use of the handy *LaunchCharacter* function to perform our jump.

A jump, being an *impulse*, *e.g. a sudden change in velocity*, and as such you can implement it simply modify the character's *velocity* property, however when jumping, it's important to preserve the character's *horizontal* movement, and this is where the *LaunchCharacter* function enters, as it lets you add or override the character's *velocity* component wise in an easy way!

TIP

While not mandatory, it's a good practice to group your character's states (e.g. Jumping, Crouching, Dashing, etc.) into its own methods.

Events

Landing

Now our player is able to freely move around and jump, however it's important to know when it has *landed* so we can react accordly, *e.g.* perform animations, reset its jump counts, etc.

You can use the following *snippet* to know when a character has landed (*e.g.* gains *walkable-ground*) in a easy and consistent way:

```
if (!characterMovement.wasGrounded && characterMovement.isGrounded)
    Debug.Log("landed!");
```

This tells us that the character was on *not-walkable-ground or not-on-ground* during the last *Move* call (*e.g.* previous frame) BUT it is now on *walkable-ground.*

A more elegant and complete solution to know when a character finds *ground* (*e.g.* its capsule's bottom hemisphere collides) *walkable* or *not-walkable*, is by making use of its *FoundGround event*. This, unlike the snippet approach, will provide us valuable information through its *FindGroundResult* object so we can react accordly, *e.g.* play sound effects, apply landing forces, determine where we have landed, etc!

Add a new method called *OnFoundGround* to the **Player** class as follows:

```
private void OnFoundGround(ref FindGroundResult foundGround)
{
    Debug.Log("Found ground...");

    // Determine if the character has landed

    if (!characterMovement.wasOnGround && foundGround.isWalkableGround)
    {
        Debug.Log("Landed!");
    }
}
```

This will be our *FoundGround event-handler*, *e.g.* the method where we will receive *FoudGround event* notifications.

In order to start receiving *FoundGround event* notifications, first we need to *subscribe* to it, also it's a good practice to *unsubscribe* from it once no longer needed, such as when our player is disabled or destroyed.

To *subscribe* add a new *MonoBehaviour OnEnable* method and add the following code into it:

```
private void OnEnable()
{
    // Subscribe to CharacterMovement FoundGround event

    characterMovement.FoundGround += OnFoundGround;
}
```

And to *unsubscribe* from it:

```
private void OnDisable()
{
    // Un-Subscribe from CharacterMovement FoundGround event

    characterMovement.FoundGround -= OnFoundGround;
}
```

Go ahead and try it! Now, each time the character’s bottom hemisphere collides we'll be notified!

Collision Events

An important *event* in order to create interactive applications like games, is to know when our character collides with something or something collides with us. You can use this *event* to react accordly like playing sound effects, instantiate particles, push others, destroy objects, etc!

Just like with the *FoundGround event*, we will make use of the *Collided event* to exactly know when our character collides with something or something collided with us.

Add a new method called *OnCollided* to the **Player** class as follows:

```
private void OnCollided(ref CollisionResult inHit)
{
    Debug.Log($"{name} collided with: {inHit.collider.name}");
}
```

This will be our *Collided event-handler*, *e.g.* the method where we will receive *Collision event* notifications. The *CollisionResult inHit* object is used to retrieve some information about the controller's evolution.

Once again, in order to start receiving *Collided event* notifications, first we need to *subscribe* to it, and *unsubscribe* from it once we no longer need it.

Add the following code to your **Player** *MonoBehaviour OnEnable* method:

```
characterMovement.Collided += OnCollided;
```

And the following code to the **Player** *OnDisable* method to unsubscribe from it:

```
characterMovement.Collided -= OnCollided;
```

In addition to the *Collided event*, we can make use of the *Character Movement GetCollisionCount()* and *GetCollisionResult()* functions to iterate over the collisions found during the last *Move* call if preferred.

Adding Forces

Until now, all our character motion has been initiated by us, like *walking*, *jumping*, etc. However no game will be complete without interaction with others and as such the *CharacterMovement* component includes a set of functions you can use to externally influence your character's movement like *AddForce*, *AddExplosionForce* and *LaunchCharacter.*

This can be used to implement bounces, force-fields, wind-zones, grenades and explosions, etc.

Creating a Bouncer

We'll start creating a bouncer, so when our character enters it, it will be launched through the air!

1.- Create a cube (*3D Object->Cube*) and set its scale to 2, 0.1, 2.

2.- Rename it to **Bouncer** and check its "*Is Trigger*" checkbox from its *Box Collider*.

3.- Create a new C# script named **Bouncer** and add it to the **Bouncer** *GameObject*.

4.- Add a *MonoBehaviour OnTriggerEnter* method to the **Bouncer** class and add the following code into it:

```
// Check if the entered collider is using the CharacterMovement component

if (other.TryGetComponent(out CharacterMovement characterMovement))
{
    // If necessary, temporarily disable the character's ground constraint so
it leave the ground

    if (characterMovement.isGrounded)
        characterMovement.PauseGroundConstraint();

    // Launch character!

    characterMovement.LaunchCharacter(transform.up * launchImpulse,
overrideVerticalVelocity, overrideLateralVelocity);
}
```

How does it work?

It first checks if the entered collider is a character using the *CharacterMovement* component, if no, won't do anything.

If yes, it temporarily disables the character's *ground constraint* so it will be able to leave the *ground* and make use of the *LaunchCharacter* function we already know to launch the character off the ground along the **Bouncer** *up-axis*.

Creating a ForceField

In a similar way to the **Bouncer**, we'll use a *trigger* to implement a **ForceField**.

1.- Duplicate the previously created **Bouncer** *GameObject* and name it as **ForceField.**

2.- Set its scale to 4, 0.1, 4.

3.- Set the *Box Collider Center* to 0, 25, 0 and its *Size* to 1, 50, 1. We need it to be a bigger area.

2.- Remove the **Bouncer** script from our **ForceField** *GameObject*.

3.- Create a new C# script named **ForceField** and add it to the **ForceField** *GameObject*.

4.- Add a *MonoBehaviour OnTriggerEnter* method to the **ForceField** class and add the following code into it:

```
private void OnTriggerEnter(Collider other)
{
    characterMovement = other.GetComponent<CharacterMovement>();
}
```

5.- Add a *MonoBehaviour OnTriggerExit* method to the **ForceField** class and add the following code into it:

```
private void OnTriggerExit(Collider other)
{
    // If our cached character leaves the trigger and remove cached
CharacterMovement component

    if (other.TryGetComponent(out CharacterMovement component) &&
        characterMovement.gameObject == component.gameObject)
    {
        characterMovement = null;
    }
}
```

6.- Lastly, add a *MonoBehaviour Update* method to the **ForceField** class and add the following code into it:

```
private void Update()
{
    // If a character is inside the ForceField trigger area, add force!

    if (characterMovement)
    {
        // If the character is grounded, pause ground constraint so it can
leave the ground

        if (characterMovement.isGrounded)
            characterMovement.PauseGroundConstraint();

        // Add continuous force

        characterMovement.AddForce(transform.up * forceMagnitude, forceMode);
    }
}
```

But wait! Why do we have to use the *Update* method here? Why not just add the force in the *OnTriggerEnter* or *OnTriggerStay method*?

Well this is inherently related to how game physics works in general, they like to be updated in a consistent way and even better using a fixed time interval like unity's *FixedUpdate* method (called every *fixed frame-rate frame*)

However, since we are *simulating* our character movement using the *MonoBehavior Update* method (*called every frame*) **for consistency** we need to add forces in the same way using an *Update* method, remember, **that's what game physics likes!**

The same applies to *moving platforms, enemies, etc*. You have to be **consistent** and *simulate* all your game using the same method, *Update* or *FixedUpdate* otherwise this will lead to desyncs, jitter and stutter.

But what if my game needs physics interactions! Don't worry we'll see later how to achieve rock-solid physics integration without having to implement managers, custom simulations or anything crazy!

Game Physics by Glenn Fiedler
https://gafferongames.com/categories/game-physics/

Timesteps and Achieving Smooth Motion in Unity
https://www.kinematicsoup.com/news/2016/8/9/rrypp5tkubynjwxhxjzd42s3o034o8

Physics Interactions

As we have seen, **game physics** like **consistency** and a **fixed time interval**, so in order to make our character to correctly interact with physics objects push / be pushed, add forces, and even use fully simulated physical objects like vehicles, boats, etc. as moving platforms, we need to make a few changes to our **Player** class.

As you know, **Unity** offers a specialized method to add your physics related code, the *FixedUpdate* method. This method is *framerate-independent* and has the frequency of the physics system; it is called every fixed frame-rate frame.

This seems to be the perfect place to *simulate* our character movement!

However, due design decisions and in order to support **Tick-based Simulation, a key factor for high-quality network multiplayer games**, without having to rely on complicated managers, custom interpolation or a strict execution order, I decided to move the character's simulation to a *LateFixedUpdate* method that is executed after the internal physics update, so our character can read the up-to-date world state.

This will offer **rock-solid physics interactions and a seamless integration with Unity's workflow**, so let's get started!

Creating our LateFixedUpdate method

In order to fulfill our *LateFixedUpdate* requirements we'll make use of a *Coroutine* tied to internal physics updates.

What is a coroutine ?

A coroutine allows you to spread tasks across several frames. In Unity, a coroutine is a method that can pause execution and return control to Unity but then continue where it left off on the following frame.

Declaring our coroutine

In our **Player** class, add the following field to declare our *Coroutine*.

```
private Coroutine _lateFixedUpdateCoroutine;
```

Declare our *LateFixedUpdate* methods as follows:

```
private IEnumerator LateFixedUpdate()
{
    WaitForFixedUpdate waitTime = new WaitForFixedUpdate();

    while (true)
    {
        yield return waitTime;

        OnLateFixedUpdate();
    }
}
```

To start, and to stop our *Coroutine*, we'll use the *OnEnable* and *OnDisable* *MonoBehavior* functions respectively:

```
private void OnEnable()
{
    // Start LateFixedUpdate coroutine

    if (_lateFixedUpdateCoroutine != null)
        StopCoroutine(_lateFixedUpdateCoroutine);

    _lateFixedUpdateCoroutine = StartCoroutine(LateFixedUpdate());
}

private void OnDisable()
{
    // Ends LateFixedUpdate coroutine

    if (_lateFixedUpdateCoroutine != null)
        StopCoroutine(_lateFixedUpdateCoroutine);
}
```

Lastly, create a new *OnLateFixedUpdate* method and move the *UpdateRotation* and *Move* functions from the *MonoBehaviour Update* to this newly created method, as follows:

```
private void OnLateFixedUpdate()
{
    UpdateRotation();
    Move();
}
```

The new ForceField

Now that our character movement is fully in sync with the physics engine, we can simplify the *ForceField* script as follows:

```
/// <summary>
/// The updated ForceField.
/// </summary>

public class NewForceField : MonoBehaviour
{
    public ForceMode forceMode = ForceMode.Force;
    public float forceMagnitude = 15.0f;

    private void OnTriggerStay(Collider other)
    {
        if (other.TryGetComponent(out CharacterMovement characterMovement))
        {
            if (characterMovement.isGrounded)
                characterMovement.PauseGroundConstraint();

            characterMovement.AddForce(transform.up * forceMagnitude,
forceMode);
        }
    }
}
```

Go ahead and try the included example scene.

TIP

Try to push and jump into the ice block!

In conclusion, put all your game logic in either *Update* or *FixedUpdate* (*LateFixedUpdate* in our case). **Do not mix and match timesteps** unless you are willing to bite the bullet and accept some stutter and physics desync. Additionally, **it is strongly worth considering putting all game states in *FixedUpdate***, using *Update* exclusively for user input, visual effects, and interpolation between game states. While this requires a change in how you structure your games, it is a proven design structure with many advantages.

Moving Platforms

The *CharacterMovement* component is able to **transparently support moving platforms**, without you having to write custom code for it, no matter if your platform is scripted, animated, tweened or even fully simulated physics objects like cars, boats, etc!

The included example shows a set of different movement platforms (*scripted*, *animated* and *physics-based*) so you can try it.

Collision Filtering

Many times in a game you will need to ignore collisions against another character, a specific *collider* or even all the colliders attached to a *rigidbody*, for this, the *CharacterMovement* component includes a set of functions to let you easily define your desired interaction.

```
public void IgnoreCollision(Collider otherCollider, bool ignore = true);
```

```
public void IgnoreCollision(Rigidbody otherRigidbody, bool ignore = true)
```

The former specifically ignores collisions against the given *collider*, while the latter will ignore all the *colliders* attached to the given *Rigidbody.*

In a similar way, you can use the *detectCollisions* property to completely enable or disable the collision detection system.

When greater control is needed, you can make use of its *ColliderFilterCallback* to determine if the collided *Collider* should be ignored or not.

Well see *callbacks* in next section, a really powerful option to tailor-fit the *CharacterComponent* to match your game needs!

Creating a One-Way Platform

In this example, we'll see how to make use of the *CharacterMovement IgnoreCollision* function to implement a one-way-platform.

1.- Create a cube (*3D Object->Cube*) named **One-Way Platform** and set its *scale* to 4, 0.1, 4.

2.- Add an empty *GameObject* as a child of our **One-Way Platform** *GameObject* and name it **PlatformTrigger.**

3.- Add a *Box Collider* to the **PlatformTrigger** *GameObject* and check its "*Is Trigger*" checkbox.

4.- Set the **Platform Trigger** *Box Collider center* to 0, -1.5, 0 and its *size* to 1, 4, 1. So it fits below the **One-Way Platform**.

5.- Create a new C# script named **OneWayPlatform** and add it to the **Platform Trigger** *GameObject*.

4.- Add a *MonoBehaviour OnTriggerEnter* method to the **OneWayPlatform** class and add the following code into it:

```
// If the entered collider is using the CharacterMovement component,
// make the character ignore the platform collider

if (other.TryGetComponent(out CharacterMovement characterMovement))
    characterMovement.IgnoreCollision(platform);
```

5.- Add a *MonoBehaviour OnTriggerExit* method to the **OneWayPlatform** class and add the following code into it:

```
// Re-enable collisions against the platform when character leaves the
trigger volume

if (other.TryGetComponent(out CharacterMovement characterMovement))
    characterMovement.IgnoreCollision(platform, false);
```

As you can see the implementation it's pretty straightforward, when the character enters to the trigger below the OneWayPlatform we make the character ignores collisions against the platform so it can pass-through it, once the character leaves the trigger (he's on top of platform) we re-enable collisions against the platform so the character can land on it.

Callbacks

The `CharacterMovement` component has been developed with extensibility in mind, so you can fine tune it to match your game needs.

It offers a set of callback functions to let you determine how a character should proceed when a collision occurs, e.g. Filter (ignore) objects, define the desired *CollisionBehavior* or even determine the collision response impulses, all in a friendly and easy to implement way.

What is a callback?

A callback is **a function that will be called when a process is done executing a specific task**.

The *callback* usage is completely optional, but provides greater flexibility to tailor-fit the *Character Movement* to match your game needs.

Using the *Collider Filter Callback*

In this example we'll see how to make use of the *ColliderFilterCallback* to ignore collisions against other characters using the *Character Movement* component.

Well use the *ColliderFilterCallback* to ignore collisions against other characters using the *Character Movement* component.

The first thing in order to make use of the *ColliderFilterCallback* is create our *callback* function, this is pretty similar to the previously seen *event-handlers* functions as both are implemented using *delegates*.

Add the following code to the **Player** class:

```
private bool ColliderFilterCallback(Collider collider)
{
    // If collider is a character (e.g. using CharacterMovement component)
    // ignore collisions with it (e.g. filter it)

    if (collider.TryGetComponent(out CharacterMovement _))
        return true;

    return false;
}
```

Now, in order to actually start using our *callback*, we first need to register it and unregister when no longer needed, just like we did with our *event-handlers*.

To *register* our *callback* function with the *Character Movement* component add a new *MonoBehaviour OnEnable* method to the **Player** class and add the following code into it:

```
characterMovement.colliderFilterCallback += ColliderFilterCallback;
```

And to *unregister* add a new *MonoBehaviour OnDisable* method and add the following code into it:

```
characterMovement.colliderFilterCallback -= ColliderFilterCallback;
```

Done! Now each time our **Player** collides with *'something'* our *callback* function will be called, so we can determine if it should ignore collisions against the given collider (return *true*) or allow collisions against it (return *false*).

Using the *Collision Behavior Callback*

This is a particularly important feature as the *CollisionBehaviorCallback* lets you customize the character's behavior after colliding.

At the time of writing the following returned flags are supported:

- *Walkable* Defines if the collided object is walkable regardless of the *slopeLimit* value.

- *NotWalkable* Defines if the collided object is not-walkable regardless of the *slopeLimit* value.

- *CanPerchOn* Defines if the character can perch on the collider.

- *CanNotPerchOn* Indicates the character can not perch on the collided collider regardless of its perchOffset value. This is equivalent to using a perchOffest value of 0.

- *CanStepOn* Defines if the character is able to step up on the other collider. Note that we still enforce stepOffset to start the step up.

- *CanNotStepOn* Indicates the character is not able to step up on the other collider regardless of its stepOffset value.

- *CanRideOn* Defines if the character can effectively travel with the object it is standing on.

- *CanNotRideOn* Indicates the character is not able to travel with the object it is standing on.

We can use this to control if a character standing on a dynamic bridge should follow the motion of the collider it is standing on. But it should not be the case if the character stands on, *i.e.* a bottle rolling on the ground, some debris etc.

To use the *CollisionBehaviorCallback* we first need to create our *callback* function.

Add the following code to the **Player** class:

```
private CollisionBehavior CollisionBehaviorCallback(Collider collider)
{
    // Assumes default behavior

    CollisionBehavior collisionBehavior = CollisionBehavior.Default;

    // If the collider is a dynamic Rigidbody, prevent using it as a moving
platform (ride on) and disable step up on it (climbable step)

    Rigidbody rb = collider.attachedRigidbody;
    if (rb && !rb.isKinematic)
        collisionBehavior = CollisionBehavior.CanNotRideOn |
CollisionBehavior.CanNotStepOn;

    return collisionBehavior;
}
```

To start using our *callback* function, we need to *register* and *unregister* it when no longer needed.

To *register* our *callback* function with the *Character Movement* component add a new *MonoBehaviour OnEnable* method to the **Player** class and add the following code into it:

```
characterMovement.collisionBehaviorCallback += CollisionBehaviorCallback;
```

And to *unregister,* add a new *MonoBehaviour OnDisable* method and add the following code into it:

```
characterMovement.collisionBehaviorCallback -= CollisionBehaviorCallback;
```

In this example we used the *CollisionBehaviorCallback* to prevent the character being able to use as dynamic platforms any dynamic rigidbody and prevent the character to be able to climb on those.

Another simple example is to return a *NotWalkable* value when the character collides with another character, so our **Player** can not stand on top of another **Player** or **NPC**!

Using the `SimpleMove` Function

At this point you should be familiar with all the *CharacterMovement* features and capable of writing your own character movement using it, however we have not covered its *SimpleMove* function.

The *SimpleMove* function is a higher level function that implements a highly configurable friction-based movement on top of the *Move* function, and those familiar with the original **Easy Character Movement** package will feel at home!

Declaration

```
public CollisionFlags SimpleMove(Vector3 desiredVelocity, float maxSpeed,
float acceleration, float deceleration, float friction, float
brakingFriction, Vector3 gravity = default, bool onlyHorizontal = true, float
deltaTime = 0.0f);
```

Parameters

desiredVelocity The target velocity.

maxSpeed The maximum speed when *grounded*. Also determines maximum horizontal speed when *falling* (*i.e. not-grounded*).

acceleration Maximum acceleration (rate of change of velocity).

deceleration The rate at which the character slows down when braking (i.e. not accelerating or if character is exceeding max speed). This is a constant opposing force that directly lowers velocity by a constant value.

friction Setting that affects movement control. Higher values allow faster changes in direction.

brakingFriction Friction (drag) coefficient applied when braking (whenever *desiredVelocity* = *Vector3.Zero*, or if character is exceeding *maxSpeed*).

gravity The current gravity force. Defaults to zero.

onlyHorizontal Determines if the vertical velocity component should be ignored when *falling* (*i.e. not-grounded*) preserving gravity effects. Defaults to true.

deltaTime The simulation *deltaTime*. Defaults to *Time.deltaTime*.

Returns

CollisionFlags. It indicates the direction of a collision: *None, Sides, Above, and Below*.

Description

Update the character's velocity using a friction-based physical model and move the character along its updated velocity. This performs collision constrained movement resolving any collisions / overlaps found during this movement.

As we can see this let us specify the desired values to **tailor-fit the character's current state movement** as needed.

A common approach to use it is creating a set of methods like *GetMaxSpeed(), GetMaxAcceleration(), GetBrakingDeceleration(),* etc. And use these methods to return the actual value for the character's current state, for example:

```
private float GetMaxSpeed()
{
    if (characterMovement.isGrounded)
        return isCrouching ? maxSpeed * crouchingSpeedModifier : maxSpeed;

    return maxSpeed;
}
```

As you can, we return the *maxSpeed* for the *character's current state* (*crouching* or *not-crouching*).

The same applies for all additional parameters, like:

```
private float GetMaxAcceleration()
{
    return characterMovement.isGrounded ? maxAcceleration : maxAcceleration * 
airControl;
}
```

Lastly, we modify our **Player** *Move* function as follows:

```
private void Move()
{
    // Handle jumping state

    Jumping();

    // Handle crouching state

    Crouching();

    // Perform friction-based movement using the CharacterMovement SimpleMove
method

    float actualMaxSpeed = GetMaxSpeed();
    float actualAcceleration = GetMaxAcceleration();
    float actualBrakingDeceleration = GetBrakingDeceleration();
    float actualFriction = GetFriction();
    float actualBrakingFriction = useSeparateBrakingFriction ?
brakingFriction : GetFriction();

    Vector3 desiredVelocity = movementDirection * actualMaxSpeed;

    characterMovement.SimpleMove(desiredVelocity, actualMaxSpeed,
actualAcceleration, actualBrakingDeceleration, actualFriction,
actualBrakingFriction, gravity);
}
```

That's all! We no longer need to handle our *grounded not-grounded* movement as the *SimpleMove* function will do for us.

Go ahead and try the included example.

TIP

Try tweaking its values and see how each affects the character movement!

Teleporting

In this example we'll see how to create a teleporter!

Creating a Teleporter

1.- Create a cube (*3D Object->Cube*) and set its scale to 2, 0.1, 2.

2.- Rename it to **Teleporter 01** and check the *"Is Trigger"* checkbox from its *Box Collider*.

3.- Create a new C# script named **Teleporter** and add it to the **Teleporter 01** *GameObject*.

4.- Add a *MonoBehaviour OnTriggerEnter* method to the **Teleporter** class and add the following code into it:

```
// If no destination or this teleporter is disabled, return

if (destination == null || !isTeleporterEnabled)
    return;

if (other.TryGetComponent(out _characterMovement))
{
    // If entered collider is using a CharacterMovement component,
    // disable interpolation and update character position

    _characterMovement.interpolation = RigidbodyInterpolation.None;
    _characterMovement.SetPosition(destination.transform.position, true);

    // Disable destination teleporter until teleported character left it,
    // otherwise will be teleported back in an infinite loop!

    destination.isTeleporterEnabled = false;
}
```

4.- Add a *MonoBehaviour OnTriggerExit* method to the **Teleporter** class and add the following code into it:

```
// On left, make sure teleporter is re-enabled

isTeleporterEnabled = true;

if (other.TryGetComponent(out CharacterMovement characterMovement) &&
    characterMovement == _characterMovement)
{
    // Character has been teleported from this teleporter,
    // Re-enable interpolation setting and clear cached component

    _characterMovement.interpolation = RigidbodyInterpolation.Interpolate;
    _characterMovement = null;
}
```

5.- Duplicate the **Teleporter 01** *GameObject* and rename it to **Teleporter 02**.

6.- Assign your **Teleporter 02** as the **Teleporter 01** destination.

7.- Lastly, assign your **Teleporter 01** as the **Teleporter 02** destination so both teleporters are linked together.

It is important to understand the difference between *Move* and *SetPosition*. The *Move* function is the core of the `CharacterMovement` component. This is where the aforementioned '*collide-and-slide*' algorithm takes place (and its other features). So the function will start from the character's current position, and use sweep tests to attempt to move in the required direction. If obstacles are found, it may make the character slide smoothly against them. Or the character can get blocked against a wall: the result of the move call depends on the surrounding geometry. On the contrary, *SetPosition* is a simple '**teleport**' function that will move the character to desired position no matter what, regardless of where the character starts from, regardless of surrounding geometry, and even if the required position is in the middle of another object.

Camera Relative Movement

Until now, our character movement has been in *world-space*, however often it's desired to move the player relative to the camera's view direction (third person, first person, etc).

Here we will use the included *.relativeTo* helper extension to transform our movement direction vector so it be relative to the camera's view direction.

Add the following code to the Player HandleInput method:

```
// Read Input values

float horizontal = Input.GetAxisRaw($"Horizontal");
float vertical = Input.GetAxisRaw($"Vertical");

// Create a movement direction vector (in world space)

movementDirection = Vector3.zero;

movementDirection += Vector3.forward * vertical;
movementDirection += Vector3.right * horizontal;

// Make movementDirection vector relative to camera view direction

movementDirection = movementDirection.relativeTo(playerCamera.transform);

// Make Sure it won't move faster diagonally

movementDirection = Vector3.ClampMagnitude(movementDirection, 1.0f);
```

Cinemachine

For our last 2 examples, we'll be making use of the **Cinemachine** package to implement typical First Person and Third Person controllers, so let's get started!

Note: In order to try these examples, please make sure to install the **Cinemachine** package into your project!

What is Cinemachine?

Cinemachine is **a modular suite of camera tools for Unity** which give AAA game quality controls for every camera in your project.

First Person Controller

In this example we'll see how to modify our **Player** class to implement a typical *first-person* movement with **Cinemachine**.

As this is a more extensive example than previous ones, Please take a look at the included example scene and the *FirstPersonController script* for details. I'll explain how it works here.

This example expands our previous **Player** class to implement a *first-person* movement.

The first change is to modify our character's rotation so it controls the *yaw-rotation* only (*i.e.* left and right) while the camera pivot *Transform* now controls the *pitch-rotation* (*i.e* up and down).

Lastly, we need to make our movement relative to us, so the *horizontal* movement is mapped along our *right* vector, while *forward/backward* movement is mapped along our *forward* vector. This let us implement the typical strafe movement. We use the previously seen *.relateTo* helper function for this.

A nice feature of **Cinemachine** is it lets us implement the *crouching* camera animation really easily! For this, we create two predefined **Cinemachine** virtual cameras, one configured for the player standing height, and the second one configured for the character crouching height. Now in our code we just need to toggle these cameras depending if the character is *crouching* or not and **Cinemachine** will perform the animated transition for us!

You can make use of the *CinemachineBrain* Default Blend (in our main camera) to define how the blend transitions and its time, or even implement custom blends for your camera animations.

Third Person Controller

In a similar way to our previous example, we'll see how to easily implement a third person movement using **Cinemachine**.

As with the previous **Cinemachine** example, please take a look at the included example scene and its companion *ThirdPersonController* script for further details.

As previously seen, to make our character movement relative to a camera's view direction, is transforming our movement direction vector so it's relative to the camera. We accomplished it using the *.relativeTo()* helper function.

To implement a *third-person* movement, we just need to add camera controls, as our character's movement code remains the same.

To control the **Cinemachine** camera, we rotate our *camera-follow Transform* and since the *Cinemachine virtual camera* is configured to *follow* this *Transform* it will carry the rotations for us!

Worth note the use of *LateUpdate* method to handle the camera rotation, **this is the recommended place when dealing with camera related code** as it guarantees the tracked object is already updated and in its current state.