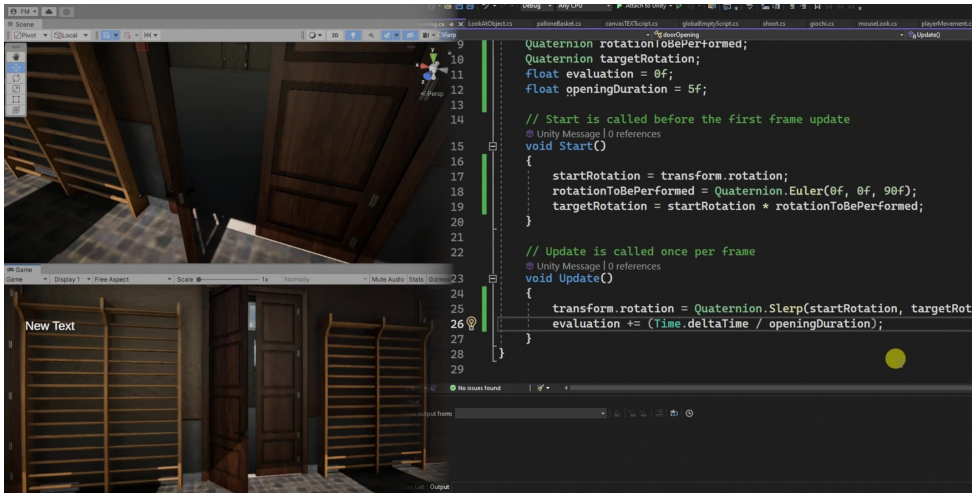


*This PDF is available for free; feel free to share it!*

[Subscribe to my YT channel](#) & turn on notifications to get updates on new videos!



Hello everyone!

In this tutorial, we will study the **Slerp** function of **Quaternion** in Unity, which allows us to transition from an initial orientation to a final one over a specified period of time, performing smooth rotations.

In this tutorial, I will assume that you are already familiar with **Time.deltaTime**, which I have covered in a previously published video tutorial. If you are not familiar with **Time.deltaTime**, I highly recommend watching that video before proceeding with this one.

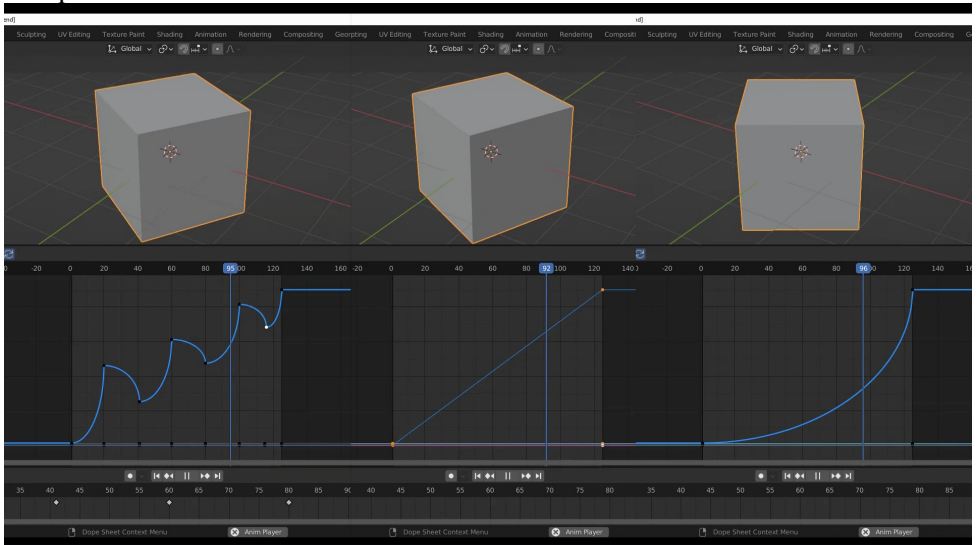
This tutorial was recorded using **Unity 2022**; however, the **Slerp** function has been available in multiple versions and will likely continue to be available in future releases.

A rotation from orientation **A** to orientation **B** (and, more generally, an interpolation between value A and value B over a specified period) can occur in various ways: the values can follow a linear progression, a spherical progression, or other types of progressions, including customized ones.

In the video, I'm showing some examples of interpolation, specifically rotating a cube by 90 degrees around its vertical axis. It is a simple example, but it effectively conveys the concept.

The rotation time is mapped within the **range [0,1]**, where at the instant 0 we have the initial value, and at the instant 1 we have the final value. Intermediate values in the range correspond to the values given by the chosen interpolation.

In the examples, the cubes all rotate by 90 degrees in 5 seconds; however, at the fourth second of the animation (which corresponds to the moment 0.8 in the range [0, 1]), the various cubes will have different orientations because their interpolations are different.



When performing rotations, as in our case, **spherical interpolation is preferred over linear interpolation** because the result is more pleasing.

In Unity, spherical interpolation is calculated using the **Slerp** function (the initial 'S' stands for **Spherical**), while linear interpolation is calculated using the **Lerp** function (the initial 'L' stands for **Linear**).

As mentioned earlier, **Slerp is part of the Quaternion class**, which allows specifying orientations and rotations in 3D space using quaternions: a mathematical concept that I will NOT explain in this tutorial.

I will limit myself to saying that they are used to express orientations (and I hope mathematicians will forgive me for this simplification) and to show you how to use the Slerp function.

**Slerp** is used to transition **from orientation A to orientation B**.

It takes three parameters: the initial orientation, the target orientation, and **the moment within the range [0, 1]** at which we want to know the orientation (in other words, the moment at which we want to **evaluate the interpolation**).

**Slerp** returns a quaternion representing the orientation evaluated at the specified moment.

So, when we write:

***Quaternion.Slerp(A, B, 0.2);***

we are asking Unity to retrieve the orientation that the object must have at one-fifth of the rotation needed to transition from orientation A to orientation B, since 0.2 is one-fifth of 1.

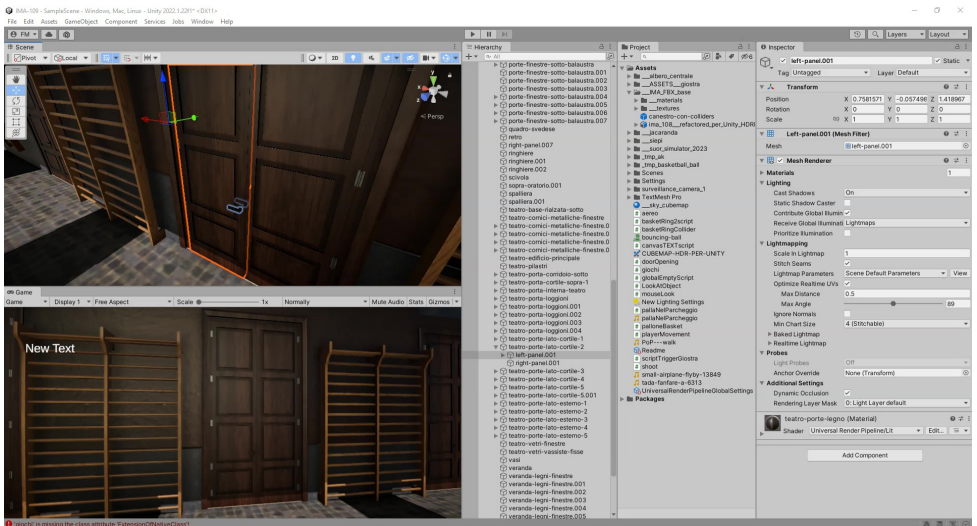
If we want the animation to last for 1 second, we can use **Time.deltaTime** in **Update** because the interpolation range from 0 to 1 can be perfectly converted into a one-second animation using **Time.deltaTime** in **Update**, as I showed in the video tutorial on **Time.deltaTime**.

On the other hand, to perform the animation over an arbitrary number of seconds, we should **divide Time.deltaTime by that amount**. I will discuss this in more detail soon with a practical example.

Most of the time, the initial orientation is the object's orientation, i.e., ***transform.rotation***, since rotation is expressed in quaternions.

Calculating the target orientation, for example, adding an angle in degrees to the initial orientation, can be problematic; however, we will also address this soon in the practical example... well, let's finally move on to a practical example!

In the video, I am showing a scene featuring a 3D model.



Specifically, the model consists of 4 elements: the door frame, the two door panels, and a handle.

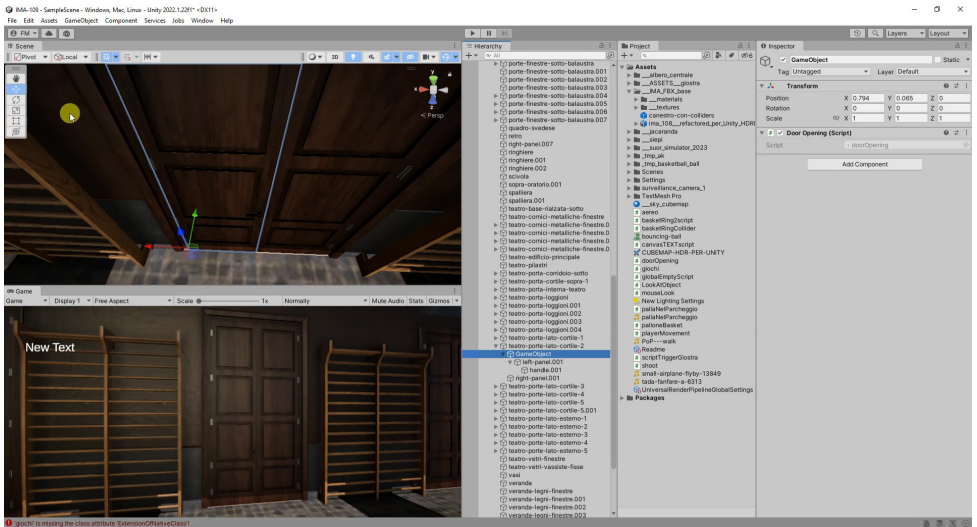
Although the pivot point of the door panel with the handle was initially located at the hinges in my original model (allowing the rotation to happen correctly), for some reason Unity has moved it a bit forward.

To solve this issue, I add an **Empty** object as a child of the frame and position it at one of the hinges of the door panel I want to animate.

Then, I make the **door panel** a child of the **Empty**.

This new **Empty** object will need a **rotation script**, which we will create immediately, naming it "**doorOpening**" for example.

In particular, we want the door panel to open on its own at the beginning of the game, rotating 90° around its **LOCAL** vertical axis (which in this case is the Z axis).



The initial orientation is given by *transform.rotation*.

For convenience (and to make the code more flexible), I also declare a Quaternion variable called "*startRotation*" and initialize it in *Start* with:

*startRotation = transform.rotation;*

The quaternion representing the final orientation must be equal to the initial orientation plus a 90° rotation around the local Z axis; but how do we define this rotation in Quaternions?

The ***Quaternion.Euler*** function comes to our aid, which takes three parameters: the rotation angles in degrees around the three XYZ axes.

So, we define the quaternion of the rotation to be performed as:

***Quaternion rotationToBePerformed;***

and initialize it in *Start* with:

***rotationToBePerformed = Quaternion.Euler(0f, 0f, 90f);***

This quaternion represents the operation to be performed, but it is not yet the final orientation.

We need to calculate that by adding this value to the initial orientation...

...and here comes the tricky part: **to add the rotation to be performed to the initial orientation, the two quaternions must be MULTIPLIED together.**

So we define the final quaternion as:

*Quaternion targetRotation;*

and initialize it in Start with:

*targetRotation = startRotation \* rotationToBePerformed;*

Let's move to the **Update** function (which, as a reminder, is **invoked at each game frame**) and start writing:

*transform.rotation = Quaternion.Slerp(startRotation, targetRotation*

and we stop here because we have not yet defined a variable for the evaluation time of the interpolation!

This variable must start at 0, so we can define and initialize it outside of **Update** with:

*float evaluation = 0f;*

then we can also write it as the third parameter for the **Quaternion.Slerp** function.

Additionally, since its value must increase at each frame, we write in **Update**:

*evaluation += Time.deltaTime;*

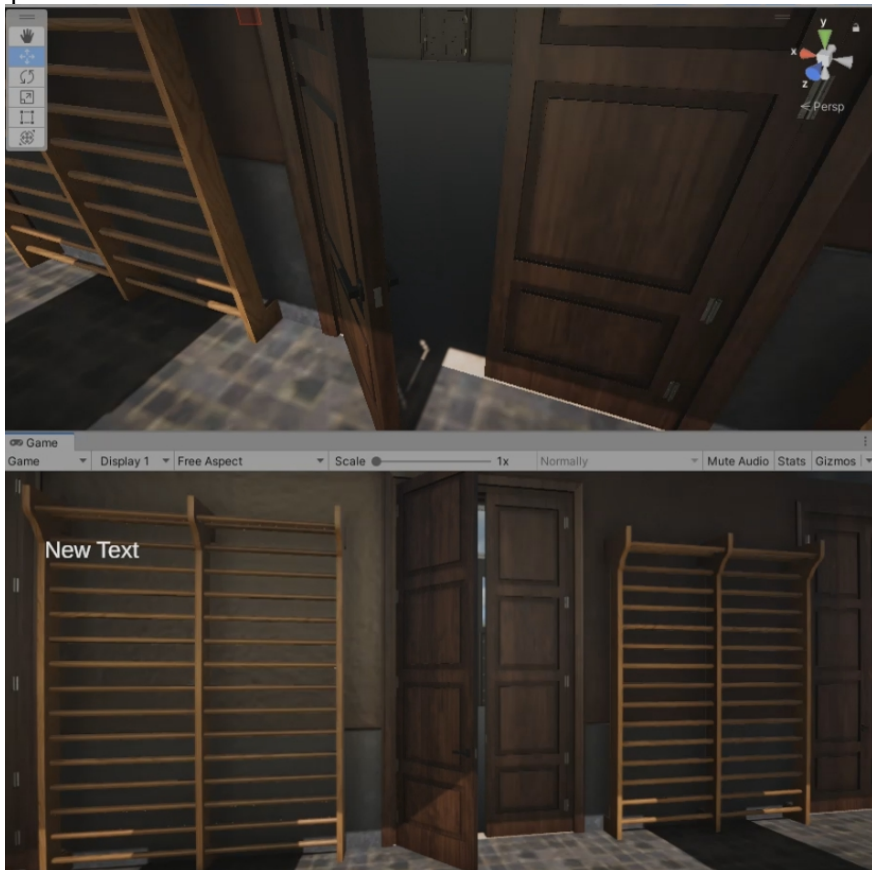
```
6 public class doorOpening : MonoBehaviour
7 {
8     Quaternion startRotation;
9     Quaternion rotationToBePerformed;
10    Quaternion targetRotation;
11    float evaluation = 0f;
12
13    // Start is called before the first frame update
14    [Unity Message | 0 references]
15    void Start()
16    {
17        startRotation = transform.rotation;
18        rotationToBePerformed = Quaternion.Euler(0f, 0f, 90f);
19        targetRotation = startRotation * rotationToBePerformed;
20    }
21
22    // Update is called once per frame
23    [Unity Message | 0 references]
24    void Update()
25    {
26        transform.rotation = Quaternion.Slerp(startRotation, targetRotation, evaluation);
27        evaluation += Time.deltaTime;
28    }
29 }
```

We know that the final evaluation time of the rotation will be 1.0, so this animation will last exactly one second, as we are incrementing evaluation by **Time.deltaTime** at each frame.

Now, when starting the game, we will notice that the Empty rotates around its vertical axis, but **the door panel does not open**.

This is because I initially set this object as **Static**, as I did not expect to animate it and had Unity calculate the global illumination; however, now I want to animate it, so **I deselect the Static checkbox** in its Inspector and confirm the operation for the child objects (i.e., the handle).

Starting the game again, this time we will see the door opening with smooth interpolation in one second.



How can we now define a **different animation duration**?

The answer is quite simple, actually: the increment of evaluation at each frame must equal **Time.deltaTime divided by the animation duration**.

With only `Time.deltaTime`, the animation lasts one second, so to make the animation last, for example, 5 seconds, we will need to increase the evaluation by one-fifth of `Time.deltaTime` at each frame.

So, let's define a variable:

*`float openingDuration = 5f;`*

because we want the animation to last 5 seconds. Then, update the instruction regarding evaluation in the Update method as follows:

*`evaluation += (Time.deltaTime / openingDuration);`*

```
6 public class DoorOpening : MonoBehaviour
7 {
8     Quaternion startRotation;
9     Quaternion rotationToBePerformed;
10    Quaternion targetRotation;
11    float evaluation = 0f;
12    float openingDuration = 5f;
13
14    // Start is called before the first frame update
15    void Start()
16    {
17        startRotation = transform.rotation;
18        rotationToBePerformed = Quaternion.Euler(0f, 0f, 90f);
19        targetRotation = startRotation * rotationToBePerformed;
20    }
21
22    // Update is called once per frame
23    void Update()
24    {
25        transform.rotation = Quaternion.Slerp(startRotation, targetRotation, evaluation);
26        evaluation += (Time.deltaTime / openingDuration);
27    }
28 }
```

Save the script, return to the Unity editor, and start the game to observe the result. This time, the animation will be performed in 5 seconds.

With this practical example, you can see how to use **Quaternion.Slerp** to smoothly interpolate between two orientations in a specific duration. You can adapt this script to different scenarios or objects to achieve smooth rotations and animations in your Unity projects.

So, to recap: in this tutorial we covered:

- how to express the rotation to be performed in **quaternions**;
- how to obtain the **target quaternion** (given by the starting quaternion **multiplied by** the rotation to be performed);

- how to use the **Slerp** function to perform rotations with spherical interpolation, with variable durations and constant execution speed, regardless of the frame rate, thanks to **Time.deltaTime**.

I hope this tutorial has been helpful to you!  
See you soon!

---

---