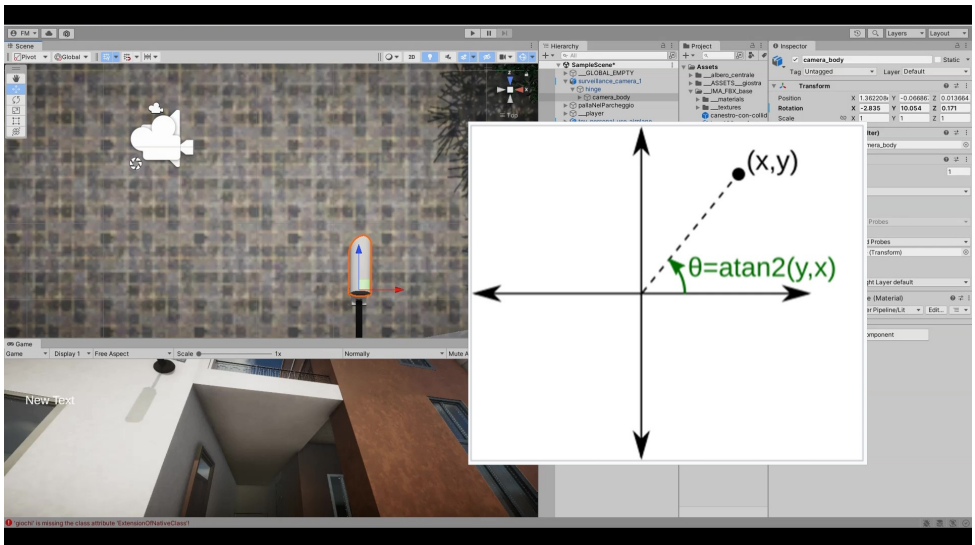


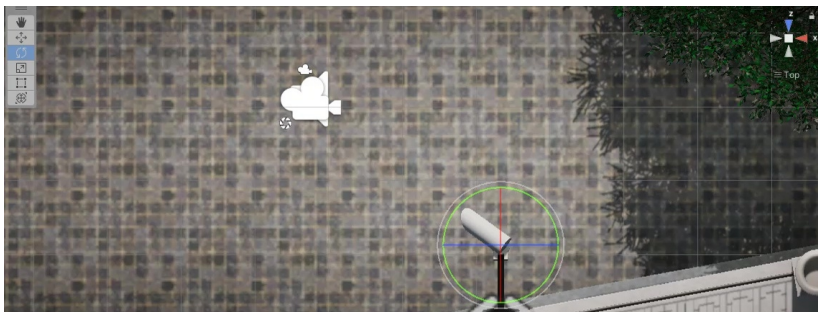
This PDF is available for free; feel free to share it!

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



Hello everyone!

This is the first of two tutorials on orienting an object towards another in Unity; specifically, we will see how to make the 3D model of a surveillance camera automatically orient itself towards the Player, following their movements within the scene.

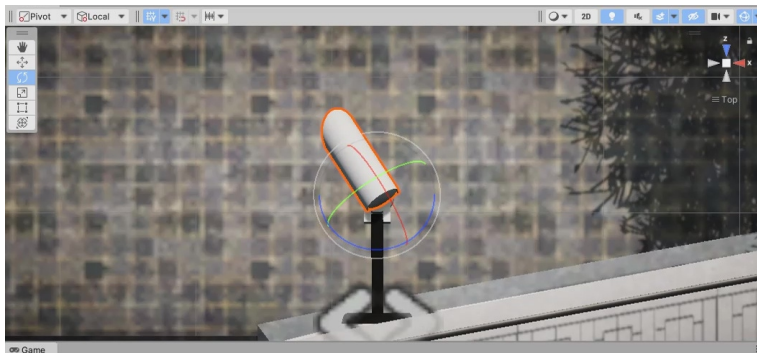


This tutorial was created using Unity 2022 and is divided into two parts.

In the first part, we will analyze the problem and approach it using the **Rotate** method to achieve an instant rotation of the camera.

In the second part, we will use **quaternions** and the **Slerp** method to create smoother and more gradual rotations.

Let's analyze the problem to be solved before moving on to the implementation. The camera can only rotate around its local vertical axis, so **we can reduce the problem to the 2D plane**; to make this aspect more understandable, I'm framing the scene from the TOP viewpoint in Orthogonal mode.



Both the 3D model of the surveillance camera and the Player have, like all game objects, **XYZ Position coordinates** that identify their position in the scene.

We are not interested in the **position.y** coordinate, because the camera can only **rotate around that LOCAL axis**, so we will only consider the **x and z position coordinates**, as if the two objects were on the same plane, with the same value for their y coordinates.

In our case, moreover, the surveillance camera can be considered as the origin of the reference system (what would be the origin in a Cartesian plane, at point 0,0).

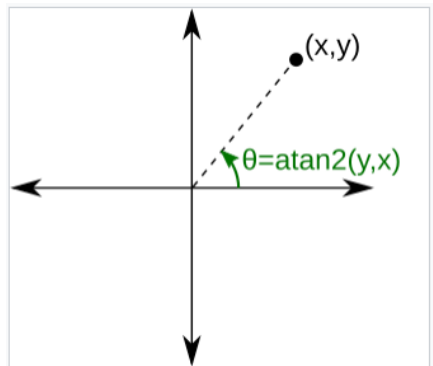
Not only that, but we can also consider **the direction framed by the camera**, which in this case corresponds to the **negative LOCAL Y-axis**, as one of the Cartesian axes.

We can then identify the direction that connects the virtual camera and the Player's position in the scene: this is simply a vector, which in this case, we will also consider only in two dimensions (i.e., GLOBAL X and Z).

It follows, therefore, that **the rotation we want to apply to the Surveillance Camera** is given by the angle that separates **the asset's orientation** (its negative local Y-axis) and the **ideal vector** connecting the Surveillance Camera to the Player: by finding this angle, we will have the rotation to be applied to the 3D model around its local vertical axis.

We are very lucky: there is a **mathematical function** called the **two-argument arctangent**, implemented by the *Atan2* method in *Mathf*, which calculates precisely this angle by taking, as a parameter, the direction vector, given by the difference between the coordinates of the two points.

Specifically, *Atan2* considers the angle between the X-axis of the Cartesian plane and the direction vector between the origin and the target.



In our case, **the origin is given by the X and Z coordinates of the 3D model** of the Surveillance Camera, so the target coordinates (x, y) will be given by the differences in coordinates between the surveillance camera and the player, while **the X-axis coincides with the negative local Y-axis** of the Surveillance Camera.

The function takes the two parameters in reverse order: in the Cartesian plane, in fact, we have *atan2(y,x)*; in our case, considering **Unity's reference system** (where the x-axis is the lateral one, the z-axis is the frontal one, and the y-axis is the vertical one, which we do not consider here), **we will write *atan2(x, z)***.

Unity's online documentation tells us that *Mathf.Atan2* takes the y and x coordinates (which in our case will be x and z) and returns, as a float value, the angle *IN RADIANS* whose tangent is y/x.

Mathf.Atan2

Declaration

```
public static float Atan2(float y, float x);
```

Description

Returns the angle in radians whose **Tan** is y/x .

Return value is the angle between the x-axis and a 2D vector starting at zero and terminating at (x,y).

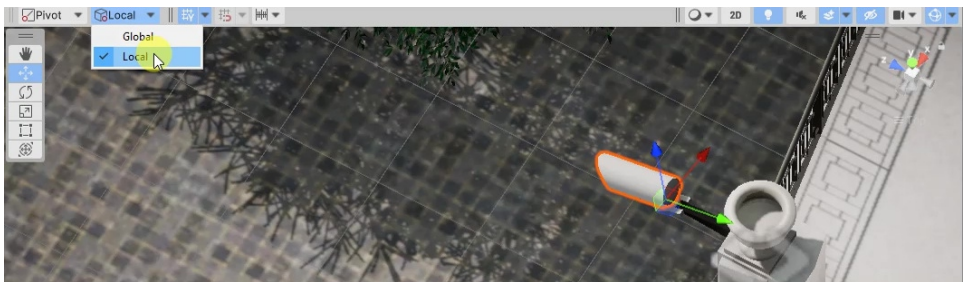
The **Rotate** function in Unity wants a rotation angle *IN DEGREES*, so we will have to take into account this difference in units of measurement (and the necessary conversion) when writing the code.

Now that we have analyzed the problem and its solution, let's move on to the first implementation: instant rotation with **Rotate**.

First, I create a new **C# script** called **LookAtObject** to be associated with the actual surveillance camera; I say "actual surveillance camera" because this asset is actually composed of three elements: a base to be placed on the wall, a pivot element, and the camera.

We are interested in having the camera **rotate around ITS vertical axis**, i.e., around its **LOCAL** vertical axis.

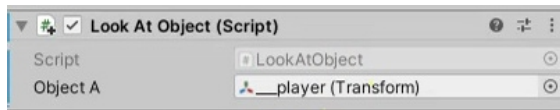
In Unity's virtual universe, the **GLOBAL** vertical axis is the y-axis, but each object can have its own **LOCAL** reference system; in the case of the 3D model of the surveillance camera, this is the **LOCAL Z-axis**, as we can see by **switching to LOCAL display mode** in the Unity editor.



This distinction introduces a slight complexity into our project: we will need to **calculate the rotation angle on the global plane** of the scene, using the X and Z coordinates of the objects, and then we will need to **perform the rotation around the local vertical axis** of the Asset.

First, let's create a reference to the object to follow, by creating a **Transform** variable called **objectA**, which we associate with the **Player** in the **Inspector**.

[SerializeField] private Transform objectA;



Having done this, we can move directly to **Update** to implement the solution discussed earlier; I remind you that the **LOCAL vertical axis** of the camera is its **Z-axis**.

We need just **three lines of code**:

- in the first, we will **calculate the angle** that separates the camera's current orientation from the one it should have (to point at the object);
- in the second, we will **calculate the amount of rotation** to be performed around the camera's local Z-axis to achieve the desired orientation;
- in the third, we will **perform the actual rotation** with the Rotate method, to which we will need to give two parameters: the local vertical vector and the amount of rotation to be performed around that vector.

Let's start with **calculating the angle** between the Surveillance Camera's orientation and the one it should have, using **Mathf's Atan2 function**, which returns a value in **radians** that we will convert to **degrees**.

We have seen that, in the 2D plane, **Atan2** takes two parameters: the difference between the Y coordinates and the difference between the X coordinates of the two points (target and observer, in this order).

In our scenario, with Unity's global coordinate system (where X and Z are on the 2D plane, while GLOBAL Y identifies the vertical axis of the scene), we will need to write:

```
float theta = Mathf.Atan2((objectA.transform.position.x -  
transform.position.x), (objectA.transform.position.z - transform.position.z))
```

but we're not done yet: in the same instruction, add at the end:

** Mathf.Rad2Deg;*

to convert the result to degrees.

Having done that, we can move on to the **second instruction: calculating the rotation to be performed**, in degrees, which can be easily done with:

float deltaAngle = Mathf.DeltaAngle(transform.eulerAngles.z, theta);

where ***transform.eulerAngles.z*** represents the **current orientation of the surveillance camera**, while ***theta*** represents the orientation, calculated in the previous step, **that it must assume**.

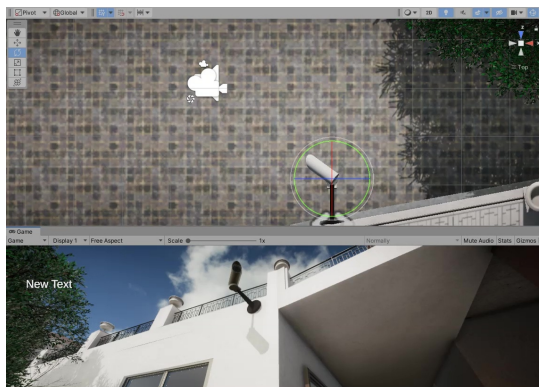
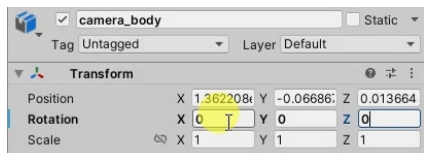
The third and final instruction allows us to **rotate the Surveillance Camera**:

transform.Rotate(Vector3.forward, deltaAngle);

Note that I used ***Vector3.forward***, which identifies the **LOCAL Z-axis** of the object, as the vector around which to perform the rotation; this is due to the fact that, in the specific case of this 3D model, the LOCAL Z-axis is not the one that indicates the front and back but the LOCAL vertical axis of the object.

Save the script, switch to the Unity editor, and press **Play** to evaluate the final result.

NOTE - If you notice that the Surveillance Camera is not pointing towards the target, **make sure that the initial XYZ Rotation values of the Surveillance Camera is set to 0** in the Inspector and try again.



So, to recap: in this first part, we examined the problem to be solved and saw how to consider the two objects on the same plane, how to calculate the direction vector, and most importantly, how to apply it to the specific case of the Surveillance Camera (taking into account its local reference system).

We then applied an **instant rotation** to the object, making it point at the Player at every frame of the game, using the ***Rotate*** function.

In the next part, we will see how to calculate the rotation **quaternions** and use the ***Slerp*** function to rotate the object more slowly.

See you soon!
