Catlike Coding

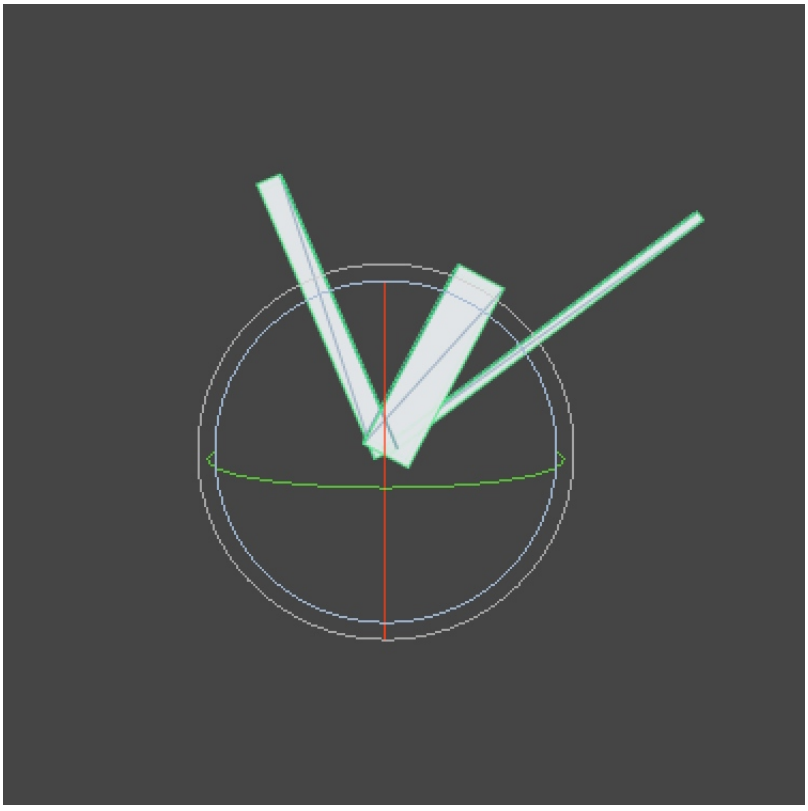
Unity C# Tutorials

Clock A Simple Time Display

Create an object hierarchy.
Write a script and attach it to an object.
Access namespaces.
Update objects through methods.
Rotate things based on time.

In this tutorial we'll write a small C# script to animate the arms of a very simple clock.

You're assumed to already have a basic understanding of Unity's editor. If you've played with it for a few minutes then you're good to go.



You will create this in no time.

Creating the clock

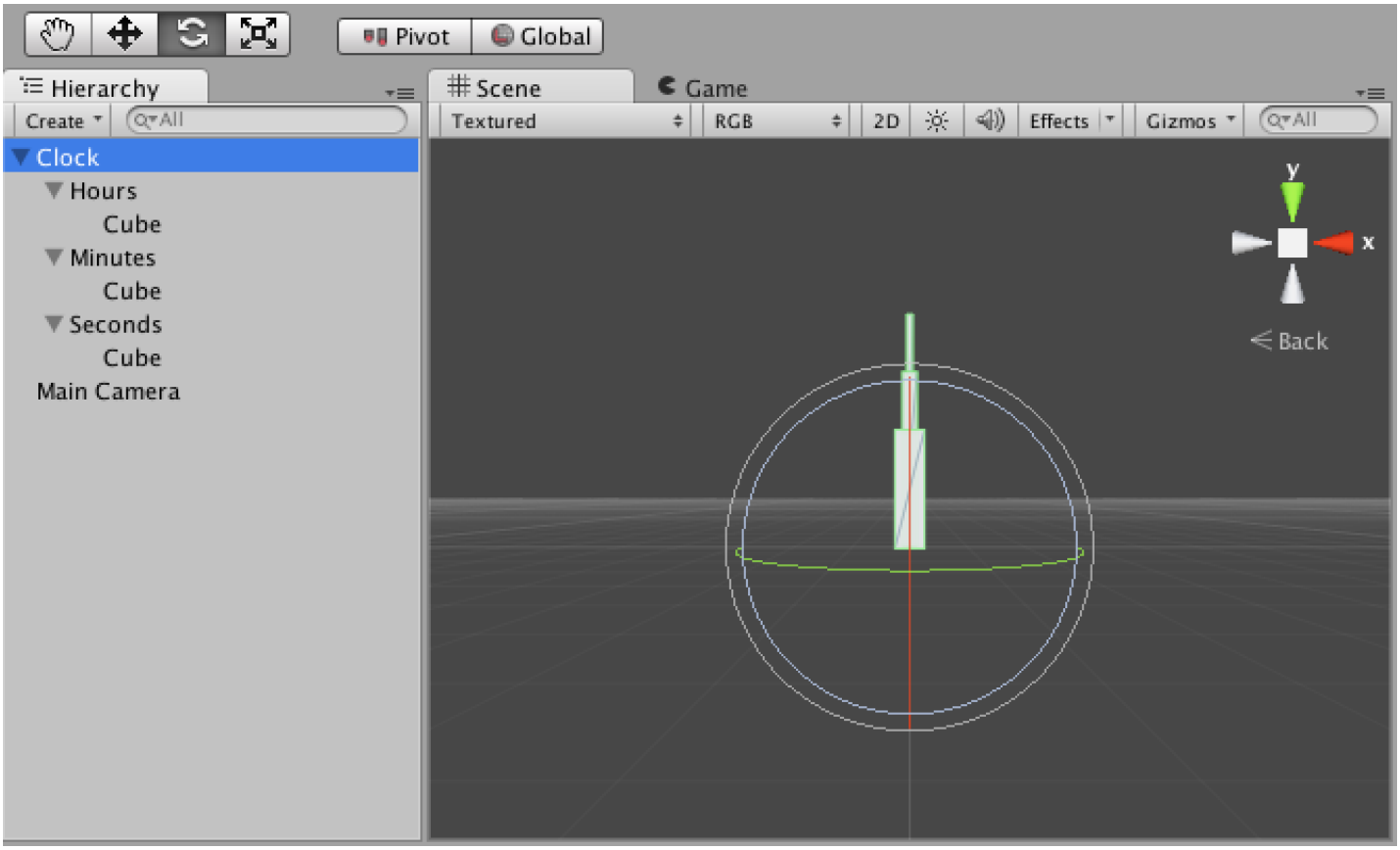
We start by creating a new Unity project without any packages. The default scene contains a camera positioned at **(0, 1, -10)** looking down the Z axis. To get a similar perspective as the camera in the scene view, select the camera and perform *GameObject / Align View to Selected* from the menu.

We need an object structure to represent the clock. Create a new empty `GameObject` via *GameObject / Create Empty*, set its position to **(0, 0, 0)**, and name it *Clock*. Create three empty child objects for it and name them *Hours*, *Minutes*, and *Seconds*. Make sure they are all positioned at **(0, 0, 0)** as well.

> **What's a `GameObject`?**
>
> Basically, everything that's placed in a scene is a `GameObject`. It has a name, a tag, a layer, a `Transform` component, and can be marked as static. By itself it doesn't do anything, it's just an empty container. You can turn it into something useful by attaching components to it and by putting other objects inside it.

> **What's a child object?**
>
> If you put an object inside another (by dragging in the *Hierarchy* view) then it's considered the child of the object that now contains it. The transformation of the parent is inherited by its children and is applied first. So it the child's transform's position is set to **(10, 0, 0)** while the parent's is **(2, 1, 0)**, then the child will end up at **(12, 1, 0)**. But if the parent's transform's rotation is set to **(0, 0, 90)** as well, the child effectively orbits its parent and ends up rotated **(0, 0, 90)** at position **(2, 11, 0)**. Scale is inherited in the same fashion.

We'll use simple boxes to visualize the arms of the clock. Create a child cube for each arm via *GameObject / Create Other / Cube*. Give the cube for *Hours* position **(0, 1, 0)** and scale **(0.5, 2, 0.5)**. For the *minutes* cube it's position **(0, 1.5, 0)** and scale **(0.25, 3, 0.25)**. For *seconds* cube it's **(0, 2, 0)** and **(0.1, 4, 0.1)**.



Clock construction and hierarchy.

Animating the clock

We need a script to animate the clock. Create a new C# script via *Create / C# Script* in the *Project* view and name it *ClockAnimator*. Open the script and empty it so we can start fresh.

First, we indicate that we want to use stuff from the `UnityEngine` namespace. Then we declare the existence of the **`ClockAnimator`**. We describe it as a publicly available **`class`** that inherits from **`MonoBehaviour`**.

```
using UnityEngine;

public class ClockAnimator : MonoBehaviour {
}
```

What's a namespace?

It's like a website domain, but for code stuff. For example, **`MonoBehaviour`** is defined inside the `UnityEngine` namespace, so it can be addressed with `UnityEngine.`**`MonoBehaviour`**.

Just like domains, namespaces can be nested. The big difference is that it's written the other way around. So instead of forum.unity3d.com it would be com.unity3d.forum. For example, the **`ArrayList`** type exists inside the `Collections` namespace, which in turn exists inside the `System` namespace. So you need to write `System.Collections.`**`ArrayList`** to get hold of it.

By declaring that we're using a namespace, we don't need to write it again each time we want to use something from it. So, when **`using`** `UnityEngine`, instead of having to write `UnityEngine.`**`MonoBehaviour`** we can suffice with **`MonoBehaviour`**.

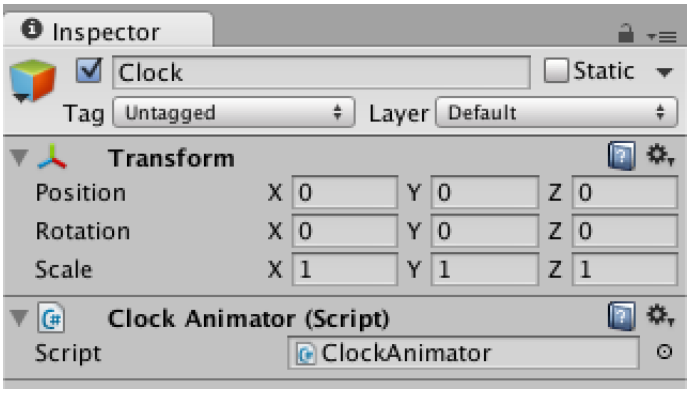
What's a `class`?

A **`class`** is a blueprint that can be used to create objects that reside in your computer's memory. The blueprint defines what data these objects contain and how they behave.

What's a `MonoBehaviour`?

`MonoBehaviour` is a class from the `UnityEngine` namespace. If you create a class that you want to use as a Unity component, then it should inherit from **`MonoBehaviour`**. It contains a lot of useful stuff and makes things like `Update` work.

This gives us a minimal class that can be used to create components. Save it, then attach it to the the *Clock* object by dragging from the *Project* view to the *Hierarchy* view, or via the *Add Component* button.



ClockAnimator attached to Clock.

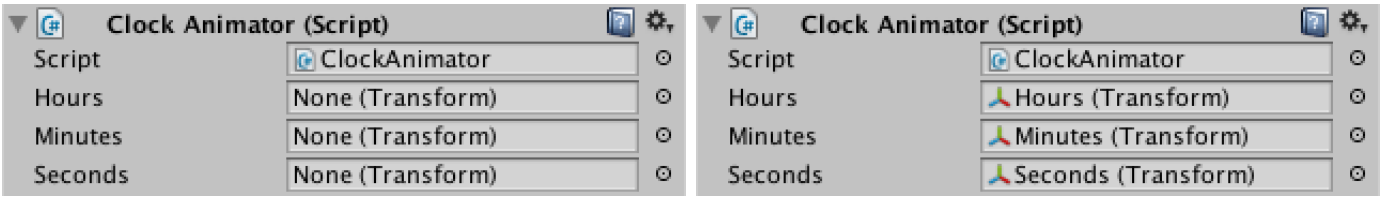
To animate the arms, we need access to their `Transform` components first. Add a public `Transform` variable for each arm to the script, then save it. These public variables will become component properties which you can assign object to in the editor. The editor will then grab the `Transform` components of these objects and assign them to our variables. Select the *Clock* object, then drag the corresponding objects to the new properties.

What's a variable?

A variable is value that can be changed. This can be either a reference to an object or something simple like an integer. A variable must be of a specific type, which is written before its name when defined.

Variables exist only inside the scope that their are defined in. By default, if a variable is defined inside a `class`, every object instance of that `class` has its own version of that variable. However, it can be marked as `static`, in which case it exists once for that `class`, independent of any objects. If a variable is defined inside a method, you can think of it only existing while that method is being invoked.

```
using UnityEngine;

public class ClockAnimator : MonoBehaviour {

    public Transform hours, minutes, seconds;
}
```



ClockAnimator with empty and filled properties.

Next, we'll add an update method to the script. This is a special method that will be called once every frame. We'll use it to set the rotation of the clock arms.

```
using UnityEngine;

public class ClockAnimator : MonoBehaviour {

	public Transform hours, minutes, seconds;

	private void Update () {
		// currently do nothing
	}
}
```

What's a method?

A method is a chunk of behavior, which is defined in a `class`. It can accept input and produce an output. Input is defined and provided after the method name, in parentheses, even if there is none. The method's type is that of its output. If there's no output this is indicated with `void`.

By default, methods define behavior of objects, but it's possible to define behavior that doesn't need an object to function. Such methods are marked as `static`.

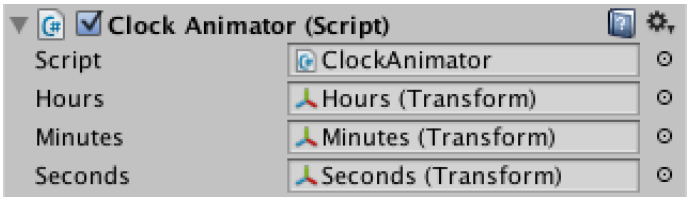
Shouldn't `Update` be `public`?

Update and a collection of other methods are considered special by Unity. It will find them and invoke them when appropriate, no matter how we declare them.

We shouldn't declare Update public, because it's not intended to be invoked by anything other than the Unity engine. The philosophy here is to make everything private that doesn't need to be accessed from outside the class. Private stuff is to not be messed with from – or directly relied upon by – the outside world. This reduces the potential for bugs.

We could omit the `private` statement, because methods and fields are private by default. I prefer to be explicit, so it is obviously intentional instead of a potential oversight.

After saving the script, the editor will notice that our component has an update method and will show a checkbox that allows us to disable it. Of course we keep it enabled.



Updating ClockAnimator adorned with a checkbox.

Each hour, the *Hours* arm has to rotate 360/12 degrees. The *Minutes* arm has to rotate 360/60 degrees per minute. Finally, the *Seconds* arm has to rotate 360/60 degrees every second. Let's define these values as private constant floating-point values for convenience.

```
using UnityEngine;

public class ClockAnimator : MonoBehaviour {

	private const float
		hoursToDegrees = 360f / 12f,
		minutesToDegrees = 360f / 60f,
		secondsToDegrees = 360f / 60f;

	public Transform hours, minutes, seconds;

	private void Update () {
		// currently do nothing
	}
}
```

What's special about `const`?

The `const` keyword indicates that a value will never change and needn't be variable. Its value will be computed during compilation and is directly inserted wherever it's referenced.

By the way, the compiler will pre-compute any constant expression, so writing `(1 + 1)` and simply writing `2` will both result in the same program.

Each update we need to know the current time to get this thing to work. The `System` namespace contains the `DateTime struct`, which is suited for this job. It has a static property named `Now` that always contains the current time. Each update we need to grab it and store it in a temporary variable.

```
using UnityEngine;
using System;

public class ClockAnimator : MonoBehaviour {

	private const float
		hoursToDegrees = 360f / 12f,
		minutesToDegrees = 360f / 60f,
		secondsToDegrees = 360f / 60f;

	public Transform hours, minutes, seconds;

	private void Update () {
		DateTime time = DateTime.Now;
	}
}
```

What's a `struct`?

A `struct` is a blueprint, just like a `class`. The difference is that whatever it creates is treated as a simple value, like an integer, instead of an object. It doesn't have the same sense of identity as an object.

What's a property?

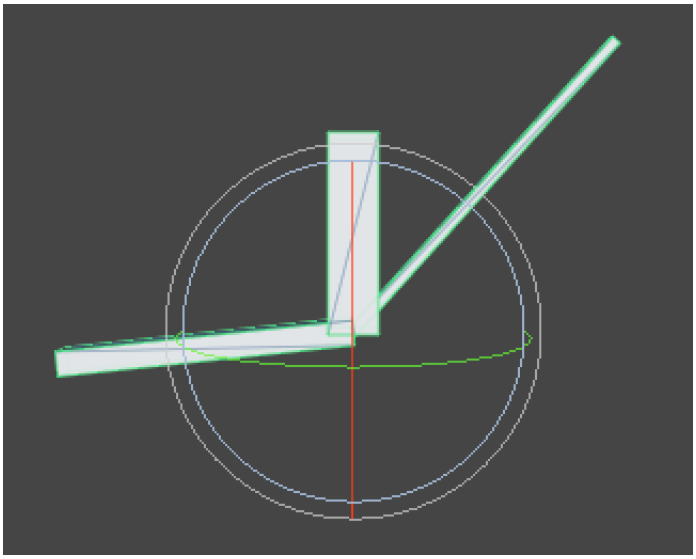
A property is a method that pretends to be a variable. It might be read-only or write-only.

Note that variables exposed by the Unity editor are also commonly called properties, but it's not the same thing.

To get the arms to rotate, we need to change their local rotation. We do this by directly setting the `localRotation` of the arms, using quaternions. `Quaternion` has a nice method we can use to define an arbitrary rotation.

Because we're looking down the Z axis and Unity uses a left-handed coordinate system, the rotation must be negative around the Z axis.

```
using UnityEngine;
using System;

public class ClockAnimator : MonoBehaviour {

	private const float
		hoursToDegrees = 360f / 12f,
		minutesToDegrees = 360f / 60f,
		secondsToDegrees = 360f / 60f;

	public Transform hours, minutes, seconds;

	private void Update () {
		DateTime time = DateTime.Now;
		hours.localRotation =
			Quaternion.Euler(0f, 0f, time.Hour * -hoursToDegrees);
		minutes.localRotation =
			Quaternion.Euler(0f, 0f, time.Minute * -minutesToDegrees);
		seconds.localRotation =
			Quaternion.Euler(0f, 0f, time.Second * -secondsToDegrees);
	}
}
```



Clock showing it's 12:44.

What's a quaternion?

Quaternions are based on complex numbers and are used to represent 3D rotations. While harder to understand than simple 3D vectors, they have some useful characteristics. The `UnityEngine` namespace contains the **`Quaternion struct`**.

Why not use `rotation`?

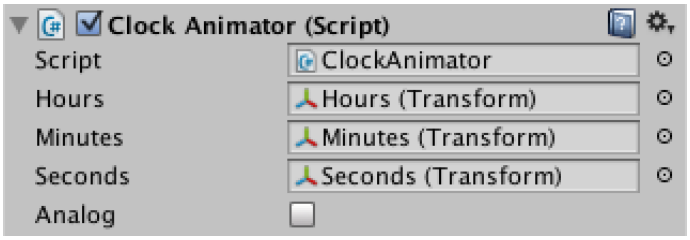
`localRotation` refers to the actual rotation of a **`Transform`**, independent of the rotation of its parent. So if we were to rotate *Clock*, its arms would rotate along with it, as expected.

`rotation` refers to the final rotation of a **`Transform`** as it is observed, taking the rotation of its parent into account. The arms would not adjust when we rotate *Clock*, as its rotation will be compensated for.

Improving the clock

This works! When in play mode, our clock shows the current time. However, it behaves much like a digital clock as it only shows discrete steps. Let's include an option to show analog time as well. Add a public boolean variable *analog* to the script and use it to determine what to do in the update method. We can toggle this value in the editor, even when in play mode.

```
using UnityEngine;
using System;

public class ClockAnimator : MonoBehaviour {

	private const float
		hoursToDegrees = 360f / 12f,
		minutesToDegrees = 360f / 60f,
		secondsToDegrees = 360f / 60f;

	public Transform hours, minutes, seconds;

	public bool analog;

	private void Update () {
		if (analog) {
			// currently do nothing
		}
		else {
			DateTime time = DateTime.Now;
			hours.localRotation =
				Quaternion.Euler(0f, 0f, time.Hour * -hoursToDegrees);
			minutes.localRotation =
				Quaternion.Euler(0f, 0f, time.Minute * -minutesToDegrees);
			seconds.localRotation =
				Quaternion.Euler(0f, 0f, time.Second * -secondsToDegrees);
		}
	}
}
```



ClockAnimator allowing analog mode.

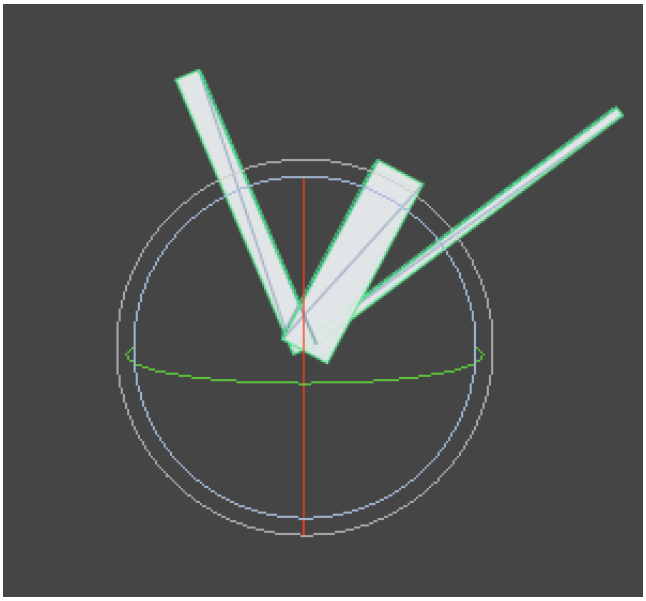
For the analog option we need a slightly different approach. Instead of `DateTime.Now` we'll use `DateTime.Now.TimeOfDay`, which is a `TimeSpan`. This allows us easy access to the fractional elapsed hours, minutes, and seconds. Because these values are provided as doubles – double precision floating-point values – we need to cast them to floats.

What's casting?

Casting changes the type of a value. You indicate this by writing the type within parentheses before the value. Casting simple values results in a conversion, like converting a floating-point value to an integer by discarding the fractional part.

Casting an object reference to another type doesn't convert the object, it only changes how that reference is treated. You'd only do this when you're sure about the object's type.

```
using UnityEngine;
using System;

public class ClockAnimator : MonoBehaviour {

	private const float
		hoursToDegrees = 360f / 12f,
		minutesToDegrees = 360f / 60f,
		secondsToDegrees = 360f / 60f;

	public Transform hours, minutes, seconds;
	public bool analog;

	private void Update () {
		if (analog) {
			TimeSpan timespan = DateTime.Now.TimeOfDay;
			hours.localRotation = Quaternion.Euler(
				0f, 0f, (float)timespan.TotalHours * -hoursToDegrees);
			minutes.localRotation = Quaternion.Euler(
				0f, 0f, (float)timespan.TotalMinutes * -minutesToDegrees);
			seconds.localRotation = Quaternion.Euler(
				0f, 0f, (float)timespan.TotalSeconds * -secondsToDegrees);
		}
		else {
			DateTime time = DateTime.Now;
			hours.localRotation =
				Quaternion.Euler(0f, 0f, time.Hour * -hoursToDegrees);
			minutes.localRotation =
				Quaternion.Euler(0f, 0f, time.Minute * -minutesToDegrees);
			seconds.localRotation =
				Quaternion.Euler(0f, 0f, time.Second * -secondsToDegrees);
		}
	}
}
```



Clock in analog mode showing it's 12:56.

Now our clock works analog too!

Enjoying the tutorials? Are they useful?

Please support me on Patreon!

made by Jasper Flick