Clock tutorial

Summary

See the object tutorial for more info about basic object creation.

Here we register the processing callback on two different clocks for didactic purposes only. Of course, all objects can be updated from the same clock. [1)]

The first clock runs at 0.01s per tick (100 Hz) and the second one at 0.2s per tick (5 Hz).

If you press the arrow keys up, down and right, you can alter the time stretching of the first clock. It'll still be updated at the same rate, but the time information that the clock will pass to the callback will be stretched.

This provides an easy way for adding time distortion and having parts of your logic code updated at different frequencies. One clock can have as many registered callbacks as you want, with an optional context parameter.

For example, the FPS displayed in the top left corner is calculated with a non-stretched clock that runs at 1Hz.

Details

When using orx, we don't have to write a global

```
while(1){}
```

loop to update our logic. Instead we create a clock [2)], specifying its update frequency.

As we can create as many clocks as we want, we can make sure that the most important part of our logic (player, NPCs, ...) will be updated very often, whereas low priority code will be called once in a while (non-interactive/background objects, etc...). For example, physics and render use two different clocks that have different frequencies.

There is also another big advantage in using separate clocks: we can easily achieve time stretching.

In this tutorial, we create two clocks, one that runs at 100Hz (period=0.01s) and the other at 5Hz (period=0.2s).

```
orxCLOCK *pstClock1, *pstClock2;

pstClock1 = orxClock_Create(orx2F(0.01f), orxCLOCK_TYPE_USER);

pstClock2 = orxClock_Create(orx2F(0.2f), orxCLOCK_TYPE_USER);
```

Note that we gave the type `orxCLOCK_TYPE_USER` to be able to retrieve it if we don't store it, from

any place in our game code. Actually, any value higher than this one is valid. The ones lesser than this are reserved for engine internal use.

Now we'll use the same update callback on both clocks. However, we'll given them different context so that the first clock callback registration applies to our first object, and the second one on the other object:

```
orxClock_Register(pstClock1, Update, pstObject1, orxMODULE_ID_MAIN,
orxCLOCK_PRIORITY_NORMAL);

orxClock_Register(pstClock2, Update, pstObject2, orxMODULE_ID_MAIN,
orxCLOCK_PRIORITY_NORMAL);
```

This means our callback will be called 100 times per second with pstObject1 and 5 times per second with pstObject2.

As our update callback just rotates the object it gets from the context parameter, we'll have, as a result, both objects turning as the same speed. However, the rotation of the second one will be far less smooth (5 Hz) than the first one's (100 Hz).

Now let's have a look at the callback code itself.

First thing: we need to get our object from the extra context parameters.
As orx is using OOP in C, we need to cast it using a cast helper that will check for cast validity.

```
pstObject = orxOBJECT(_pstContext);
```

If this returns NULL, either the parameter is incorrect, or it isn't an orxOBJECT.

Our next step will be to apply the rotation to the object.

```
orxObject_SetRotation(pstObject, orxMATH_KF_PI * _pstClockInfo->fTime)
```

We see here that we use the time taken from our clock's information structure.
That's because all our logic code is wrapped in clocks' updates that we can enforce time consistency and allow time stretching.

Of course, there are far better ways of making an object rotate on itself [3].
But let's back to our current matter: clock and time stretching!

In our update callback, we also polls for active inputs. Inputs are merely character strings that are bound, either in config file or by code at runtime, to key presses, mouse buttons or even joystick buttons.

In our case, if the up or down arrow keys are pressed, we'll strecthed the time for the first clock that has been created.
If left or right arrow keys are pressed, we'll remove the stretching and go back to the original frequency.

As we didn't store our first created clock [4], we need to get it back!

```
pstClock = orxClock_FindFirst(orx2F(-1.0f), orxCLOCK_TYPE_USER);
```

Specifying `-1.0f` as desired period means we're not looking for a precise period but for all clocks of the specified type. It'll return the first clock created with the `orxCLOCK_TYPE_USER` type, which is the one updating our first object.

Now, if the "`Faster`" input is active(ie. up arrow key is pressed), we'll speed our clock with a factor 4X.

```
if(orxInput_IsActive("Faster"))
{
  /* Makes this clock go four time faster */
  orxClock_SetModifier(pstClock, orxCLOCK_MOD_TYPE_MULTIPLY, orx2F(4.0f));
}
```

In the same way we make it 4X slower than originally by changing its modifier when "`Slower`" input is active (ie. down arrow pressed).

```
else if(orxInput_IsActive("Slower"))
{
  /* Makes this clock go four time slower */
  orxClock_SetModifier(pstClock, orxCLOCK_MOD_TYPE_MULTIPLY, orx2F(0.25f));
}
```

Lastly, we want to set it back to normal, when the "`Normal`" input is active (ie. left or right arrow key pressed).

```
else if(orxInput_IsActive("Normal"))
{
  /* Removes modifier from this clock */
  orxClock_SetModifier(pstClock, orxCLOCK_MOD_TYPE_NONE, orxFLOAT_0);
}
```

And here we are! 
As you can see, time stretching is achieved with a single line of code. As our logic code to rotate our object will use the clock's modified time, we'll see the rotation of our first object changing based on the clock modifier value.

This can be used in the exact same way to slow down monsters while the player will still move as the same pace, for example. There are other clock modifiers type but they'll be covered later on.

Resources

Source code: 02_Clock.c

Config file: 02_Clock.ini

1)

The given clock context is also used here for demonstration only.
2)

or we register to an existing one, such as the core or the physics clock
3)

by giving it an angular velocity for example, or even by using an `orx`FX
4)

on purpose, so as to show how to retrieve it

From:
https://www.orx-project.org/wiki/ - **Orx Learning**

Permanent link:
https://www.orx-project.org/wiki/es/orx/tutorials/clock?rev=1250268579

Last update: **2025/09/30 17:26 (14 months ago)**