====== 时钟教程(Clock tutorial) ======[1]

综述

了解创建对象的基础知识请看 object tutorial

这里我们在两个不同的时钟上注册处理回调(callback)这样做只是为了教学方便。当然，所有的对象都能从同一个时钟上被更新。 [2]

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

第一个时钟以每0.01s计时一次[3](100Hz) 第二个时钟每0.2s计时一次(5 Hz)

如果你按下向上，向下或者向右的方向键，可以修改第一个时钟的频率。它还是以相同的频率刷新，但是时钟传递给callback的时间信息将被增加或减少。

这提供了一种简单的方式来添加时间上的变化或者让部分代码以不同的频率更新。 一个时钟可以有任意你想要的数目的带有一个可选上下文参数(context parameter)的回调。

例如，左上角的FPS是使用一个没有伸缩的以1Hz的频率运行的时钟计算的。

细节

使用orx我们不需要编写一个全局的

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

循环去更新我们的逻辑(logic) 取而代之的是我们创建一个时钟(clock) [4])，并指定其更新频率。

因为我们可以创建任意多的时钟(clock)我们可以确保逻辑(logic)中最重要的角色（玩家NPCs...) 将被很频繁地更新，同时低优先级别的代码将被间或地调用（不活动的对象，背景对象，等等）。

使用分离的时钟还有一个很大的优点：我们可以很容易地实现时间伸缩(stretching)

在本教程中，我们创建两个时钟，一个以100Hz(周期=0.01s)的频率运行，另一个以5Hz(周期=0.2s)的频率运行。

```
orxCLOCK *pstClock1, *pstClock2;

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

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

注意我们给出的类型orxCLOCK_TYPE_USER，在我们没有存储(store)它的情况下，它可以从我们的游戏代码中的任何地方取到。 实际上，任何大于orxCLOCK_TYPE_USER的值都是有效的。小于orxCLOCK_TYPE_USER的值保留给引擎内部使用。

现在我们将在两个时钟上使用相同的更新回调(callback)但是，我们将给它们不同的上下文(context)于是第一个时钟的回调注册应用到第一个对象上，第二个回调应用到其他对象上。

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

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

这意味这我们的pstObject1对象上的回调每秒将被调用100次，pstObject2上的回调每秒将被调用5次。

因为我们的更新回调只是旋转它从上下文参数(context parameter)中获得的对象(object)，所以两个对象都将以相同的速度转动。但是第二个对象的旋转(5 Hz)将远远没有第一个对象的旋转(100 Hz)平滑。

现在我们来看看回调代码自身。

第一件事：我们需要从外部的上下文参数(context parameters)中获得我们的对象。
因为orx使用了C的OOP（面向对象技术），我们需要使用一个会检查转型有效性的转型助手(caster helper)将它转型。[5)]

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

如果返回 NULL，要么是参数不正确，要么它不是一个orxOBJECT。

下一步我们给对象加上旋转。

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

我们看到这里我们使用从时钟信息结构中的获得的时间。
那是因为所有的逻辑代码都封装在时钟的更新中，我们能够迫使时间一致，并允许时间伸展(stretching)。

当然，还有更好的方式实现对象围绕自身旋转[6)]。
但是让我们回到我们当前的问题：时钟和时间伸缩(time stretching)。

在我们的更新回调中，我们也检查活动输入(active inputs)。输入只是一些在配置文件中或者通过运行时代码绑定到按键、鼠标按钮甚至游戏手柄的字符串。

在我们的例子中，当向上或者向下键按下，我们将伸缩第一个时钟的时间。如果向左或者向右键被按下，将删除伸缩回到原来的频率。

因为我们没有保存第一个创建的的时钟[7)]，我们需要将它取回来。

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

指定-1.0f作为预期的周期，意味这我们不是查找精确（匹配）周期的时钟，而是查找所有指定类型的时钟。
它（指这行代码）将返回第一个以orxCLOCK_TYPE_USER类型创建的时钟，也就是更新我们的第一个对象的时钟。

现在，如果“Faster”输入是激活的(例如向上键被按下)， 我们将时钟加速到原来的四倍。

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

```
}
```

类似地，当“Slower”输入激活时（例如向下键被按下），我们通过修改modifier(变速器)使时钟慢到原来的四分之一。

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

最后，我们想要在“Normal（正常）”输入激活时（例如按下向左或者向右键），将它的速度设回到正常。

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

现在我们到了！
就像你所看到的，通过一行代码就实现了时间伸缩(time stretching)，因为我们的逻辑代码去旋转我们的对象时，将使用时钟的修改过的时间，我们将看到第一个对象的旋转基于clock modifier（时钟变速器）的值变化。

这可以以相同的方式用于其他目的。例如，当玩家仍将以相同的节奏移动时减缓monsters（怪物）的速度。还有其他的clock modifier（时钟变速器）类型，它们将在后面的教程中讲到。

资源

源码：02_Clock.c

配置文件：02_Clock.ini

1)
译者注：其实这里的clock就相当于一个计时器(timer)
2)
给定的时钟情境(clock context)被用在这里也只是为了演示
3)
译者注：即调用callback函数一次
4)
或者注册到一个已经存在的时钟上，例如核心时钟(the core clock
5)
译者注:即下面用到的orxOBJECT宏
6)
例如可以通过给顶一个角速度，甚至通过使用orxFX(orx特效)
7)
有意地，为了演示如何获得它

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

Permanent link:
https://orx-project.org/wiki/cn/orx/tutorials/clock?rev=1278641352

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