本页由 落后的簑羽鹤 翻译自 官方的教程

视口与摄像机

综述

前面的基本教程基础, 对象创建), 时钟 , 框架层次结构 和 动画

此教程显示了如何使用有多个摄像机的多视口技术。教程中将同时创建4个视口。

分别为左上角的Viewport1右下角的Viewport4它们共用一个摄像机(Camera1)实现此功能，只需要在配置文件中配置2个视口的Camera属性，为同一个(也就是Camera1)当我们使用鼠标的左右键旋转摄像机(Camera1),left Control或left Shift键+方向键进行摄像机的缩放操作，关联的两个Viewport1和Viewport4将相应的发生变化。

右上角视口Viewport2是基于另一个摄像机Camrea2此摄像机的视锥较第一个窄，所以显示时比例是其的两倍大。在教程的程序中，我们不能通过任何操作设置此视口。

最后一个视口Viewport3是基于Camera3的，Camera3的配置与Camera1完全一样。

NB当两个视口重叠，较先创建的将显示在顶层。

最后，有一个固定不动的箱子和一个世界坐标随着鼠标实时移动的小兵，也就是说无论如何设置视口的摄像机，无论鼠标在那个视口上移动，小兵在它所属的视口中，相对于鼠标在在屏幕中的位置移动。

在配置文件中使用随机关键字符‘~’, 使的视口和基本对象的颜色和大小可以随机创建。

NB摄像机将它的坐标/缩放尺度/旋转存放在orxFRAME structure中，在

frame教程中我们看到他们是orxFrame hierarchy的一部分。另一方面Object应该置于其Camera所关联的Viewport中。

详细说明

常我们需要首先载入配置文件，创建时钟和注册回调的 Update函数，最后创建主要的Object信息。关于实现的详情，请联系前面的教程。

虽然这次我们创建了4个视口，却没有什么新东西，仅仅是以下4行代码。

```
pstViewport = orxViewport_CreateFromConfig("Viewport1");
orxViewport_CreateFromConfig("Viewport2");
orxViewport_CreateFromConfig("Viewport3");
orxViewport_CreateFromConfig("Viewport4");
```

正如你所看到的，我们只使用了 Viewport1的引用，以便后面进行操作。

让我们直接跳到Update函数的代码。

首先我们通过捕捉鼠标的坐标，设置士兵的位置。我们已经在frame tutorial里实现过了。这里我们做了一样的事情，但在4个视口中工作的都很完美。当鼠标离开视口时，世界坐标的指针，将被orxNull值所代替，也就不会触发士兵的移动了。

```
orxVECTOR vPos;

if(orxRender_GetWorldPosition(orxMouse_GetPosition(&vPos), &vPos) !=
orxNULL)
{
  orxVECTOR vSoldierPos;

  orxObject_GetWorldPosition(pstSoldier, &vSoldierPos);
  vPos.fZ = vSoldierPos.fZ;

  orxObject_SetPosition(pstSoldier, &vPos);
}
```

在操作视口之前，我们先关注下视口所关联的摄像机，我们可以移动，旋转和缩放它。获取摄像机的代码如下所示：

```
pstCamera = orxViewport_GetCamera(pstViewport);
```

非常简单。让我们实现旋转。 [1].

```
if(orxInput_IsActive("CameraRotateLeft"))
{
  orxCamera_SetRotation(pstCamera, orxCamera_GetRotation(pstCamera) +
orx2F(-4.0f) * _pstClockInfo->fDT);
}
```

我们再次看到旋转的角度时间并不依赖于FPS而是时钟的DT我们也可以通过设置System这个配置选项来设置旋转速度，而不是使用硬编码。

实现缩放如下：

```
if(orxInput_IsActive("CameraZoomIn"))
{
  orxCamera_SetZoom(pstCamera, orxCamera_GetZoom(pstCamera) * orx2F(1.02f));
}
```

因为这个代码没有使用时钟信息，所以他将会被时钟频率和帧率所影响。 最后让我们移动摄像机。

```
orxCamera_GetPosition(pstCamera, &vPos);

if(orxInput_IsActive("CameraRight"))
{
  vPos.fX += orx2F(500) * _pstClockInfo->fDT;
}

orxCamera_SetPosition(pstCamera, &vPos);
```

We're now done playing with the camera.
As we'll see a bit later in this tutorial, this same camera is linked to two different viewports.
They'll thus both be affected when we play with it.

As for viewport direct interactions, we can alter it's size of position, for example.
We can do it like this, for example.

```
orxFLOAT fWidth, fHeight, fX, fY;

orxViewport_GetRelativeSize(pstViewport, &fWidth, &fHeight);

if(orxInput_IsActive("ViewportScaleUp"))
{
  fWidth *= orx2F(1.02f);
  fHeight*= orx2F(1.02f);
}

orxViewport_SetRelativeSize(pstViewport, fWidth, fHeight);

orxViewport_GetPosition(pstViewport, &fX, &fY);

if(orxInput_IsActive("ViewportRight"))
{
  fX += orx2F(500) * _pstClockInfo->fDT;
}

orxViewport_SetPosition(pstViewport, fX, fY);
```

Nothing really surprising as you can see.

Let's now have a look to the data side of our viewports.

```
[Viewport1]
Camera            = Camera1
RelativeSize      = (0.5, 0.5, 0.0)
RelativePosition  = top left
BackgroundColor   = (0, 100, 0) ~ (0, 255, 0)

[Viewport2]
Camera            = Camera2
RelativeSize      = @Viewport1
RelativePosition  = top right
BackgroundColor   = (100, 0, 0) ~ (255, 0, 0)

[Viewport3]
Camera            = Camera3
RelativeSize      = @Viewport1
RelativePosition  = bottom left
BackgroundColor   = (0, 0, 100) ~ (0, 0, 255)

[Viewport4]
```

```
Camera           = @Viewport1
RelativeSize     = @Viewport1
RelativePosition = bottom right
BackgroundColor  = (255, 255, 0)#(0, 255, 255)#(255, 0, 255)
```

As we can see, nothing really surprising here either.
We have three cameras for 4 viewports as we're using `Camera1` for both `Viewport1` and `Viewport4`.
We can also notice that all our viewports begins with a relative size of `(0.5, 0.5, 0.0)`. This means each viewport will use half the display size vertically and horizontally (the Z coordinate is ignored).
In other words, each viewport covers exactly a quart of our display, whichever sizes we have chosen for it, fullscreen or not.

As you may have noticed, we only gave an explicit value for the `RelativeSize` for our `Viewport1`.
All the other viewports inherits from the `Viewport1` `RelativeSize` as we wrote `@Viewport1`.
That means that this value will be the same than the one from `Viewport1` with the same key (`RelativeSize`).
We did it exactly the same way for `Viewport4`'s `Camera` by using `@Viewport1`.

We then need to place them on screen to prevent them to be all displayed on top of each other.
To do so, we use the property `RelativePosition` that can take either a literal value [2] or a vector in the same way we did for its `RelativeSize`.

Lastly, the first three viewports use different shades for their `BackgroundColor`.
For example,

```
BackgroundColor = (200, 0, 0) ~ (255, 0, 0)
```

means the this viewport will use a random [3] shade of red.

If we want to color more presicely the `BackgroundColor` but still keep a random, we can use a list as in

```
BackgroundColor = (255, 255, 0)#(0, 255, 255)#(255, 0, 255)
```

This gives three possibilities for our random color: yellow, cyan and magenta.

Finally, let's have a look to our cameras.

```
[Camera1]
FrustumWidth  = @Display.ScreenWidth
FrustumHeight = @Display.ScreenHeight
FrustumFar    = 1.0
FrustumNear   = 0.0
Position      = (0.0, 0.0, -1.0)

[Camera2]
FrustumWidth  = 400.0
FrustumHeight = 300.0
FrustumFar    = 1.0
```

```
FrustumNear   = 0.0
Position      = (0.0, 0.0, -1.0)

[Camera3@Camera1]
```

We basically define their frustum (ie. the part of world space that will be *seen* by the camera and rendered on the viewport).
NB: As we're using 2D cameras, the frustum shape is rectangular cuboid.

Note that the `Camera3` inherits from `Camera1` but don't override any property: they have the exact same property.
NB: When inheritance is used for a whole section, it's written this way: [MySection@ParentSection].
Why using two different cameras then? Only so as to have two physical entities: when we alter properties of `Camera1` in our code, the `Camera3` will remain unchanged.

We can also notice that `Camera1`'s `FrustumWidth` and `FrustumHeight` inherits from the `Display`'s screen settings.
NB: When inheritance is used for a value, it's written like this `MyKey = @ParentSection.ParentKey`.
The parent's key can be omitted if it's the same as our key: `SameKey = @ParentSection`.

Lastly we notice that our `Camera2` has a smaller frustum.
This means `Camera2` will see a smaller portion of the world space. Therefore the corresponding viewport display will look like it's zoomed! 😀

Resources

Source code: 05_Viewport.c

Config file: 05_Viewport.ini

1)
其他方向仅仅只有部分代码，但是逻辑是一样的
2)
composed of keywords `top`, `bottom`, `center`, `right` and `left`
3)
the '~' character is used as a random operator between two numeric values

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

Permanent link:
https://www.orx-project.org/wiki/cn/orx/tutorials/viewport?rev=1278420805

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