Viewport tutorial

Summary

See previous basic tutorials for more info about basic object creation, clock handling, frames hierarchy and animations.

This tutorial shows how to use multiple viewports with multiple cameras.
Four viewports are created here.

The top left corner one (`Viewport1`) and the bottom right corner one (`Viewport4`) shares the same camera (`Camera1`).
To achieve this, we just need to use the same name in the config file for the camera.

Furthermore, when manipulating this camera using left & right mouse buttons to rotate it, arrow keys to move it and left control & left shift to zoom it, the two viewports associated with this camera will both be affected.

The top right viewport (`Viewport2`) is based on another camera (`Camera2`) which frustum is narrower than the first one, resulting in a display twice as big. You can't affect this viewport at runtime in this tutorial.

The last viewport (`Viewport3`) is based on another camera (`Camera3`) which has the exact same settings than the first one.
This viewport will display what you originally had in the `Viewport1` & `Viewport4` before modifying their camera.
You can also interact directly with the first viewport properties, using WASD keys to move it and 'Q' & 'E' to resize it.

NB: When two viewports overlap, the latest one (ie. the one created after the other) will be displayed on top.

Lastly, we have a box that doesn't move at all, and a little soldier whose world position will be determined by the current mouse on-screen position.
In other words, no matter which viewport your mouse is on, and no matter how the camera for this viewport is set, the soldier will always have his feet at the same position than your mouse on screen (provided it's in a viewport).

Viewports and objects are created with random colors and sizes using the character '~' in config file.

NB: Cameras store their position/zoom/rotation in an orxFRAME structure, thus allowing them to be part of the orxFRAME hierarchy as seen in the frame tutorial.
As a result, object auto-following can be achieved by setting the object as the camera's parent.
On the other hand, having a camera as parent of an object will insure that the object will always be displayed at the same place in the corresponding viewport [1].

Details

As usual, we begin by getting the main clock and registering our `Update` function to it and, lastly, by creating our main object.
Please refer to the previous tutorials for more details.

However we create four viewports this time. Nothing really new, so we only need to write this code.

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

As you can see we only keep a reference to one created viewport. We do so as we want to interact with it later on, but we won't touch the three other ones.

Let's jump directly to our `Update` code.
First we snap our soldier guy under the mouse position. We've already seen such a thing in the frame tutorial.
Here we do the exact same thing and we see that it works perfectly with multiple viewports.
When the mouse is not over a viewport, `orxNULL` is returned instead of a pointer to the world coordinate values.

```
orxVECTOR vPos;

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

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

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

Before interacting directly with a viewport, let's play a bit with its associated camera.
We could, for example, move it, rotate it or zoom.

Let's begin by getting our first viewport camera.

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

Ok, that was easy. Let's try to rotate it [2].

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

```
}
```

Again, we see that our rotation won't be affected by the FPS and can be time-stretched as we use the clock's DT.

We could still have used the config system to get the rotation speed instead of hardcoding it! 

Let's zoom, now.

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

As this code doesn't use any clock info, it'll get affected by the clock's frequency and by the framerate.

Lastly, let's move our camera.

```
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 [3] 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 [4] 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)
very useful for making HUD & UI, for example
2)
only part of the code will be shown as for other directions, the logic remains
3)
composed of keywords `top`, `bottom`, `center`, `right` and `left`
4)
the '~' character is used as a random operator between two numeric values