Object Transformations

Summary

See previous basic tutorials for more info about basic object creation and clock handling.

All objects' positions, scales and rotations are stored in `orx`FRAME structures.
These frames are assembled in a hierarchy graph, meaning that changing a parent frame properties will affect all its children.

In this tutorial, we have four objects that we link to a common parent [1] and a fifth one which has no parent.
The first two children are implicitly created using the object's config property `ChildList` whereas the two others are created and linked in code (for didactic purposes).
The invisible parent object will follow the mouse cursor. Left shift and left control keys will respectively scale up and down the parent object, where as left and right clicks will apply a rotation to it.
All these transformations will affect its four children.

This provides us with an easy way to create complex or grouped objects and transform them (position, scale, rotation, speed, ...) very easily.

Details

As with the previous tutorials, we begin by creating a viewport.

```
orxViewport_CreateFromConfig("Viewport");
```

We then create our parent object.

```
pstParentObject = orxObject_CreateFromConfig("ParentObject");
```

As in the config file, for our `ParentObject` we defined:

```
[ParentObject]
ChildList = Object3 # Object4
```

Thus when we create our parent object, those two children [2] have also been *automagically* created and linked.
We could have done so for all the four children, but, for a learning purpose, we'll create the two remaining children in code and link them manually.

```
orxOBJECT *pstObject;
orxObject_CreateFromConfig("Object0");
pstObject = orxObject_CreateFromConfig("Object1");
orxObject_SetParent(pstObject, pstParentObject);
```

```
pstObject = orxObject_CreateFromConfig("Object2");
orxObject_SetParent(pstObject, pstParentObject);
```

Here, `Object0` is our static object: the only one that won't be linked to our `ParentObject`.

Please note that when we create and link manually objects in code, it's our responsability to delete them. On the contrary, `Object3` and `Object4` will be automatically deleted when `ParentObject` will be deleted.

We then look for the main clock and register our `Update` function to it. This function is where we'll manage the inputs to scale/rotate the `ParentObject` and make sure it'll follow our mouse cursor. [3)]

```
pstClock = orxClock_Get(orxCLOCK_KZ_CORE);

orxClock_Register(pstClock, Update, orxNULL, orxMODULE_ID_MAIN,
orxCLOCK_PRIORITY_NORMAL);
```

Let's now have a look to our `Update` function.
First, we make sure we can find the position in our world space that corresponds to our mouse cursor in the screen space.
We then copy our `ParentObject` Z coordinate (ie. we keep the same depth as before) over it and we finally set it back on our `ParentObject`.

```
if(orxRender_GetWorldPosition(orxMouse_GetPosition(&vPosition), &vPosition))
{
  orxVECTOR vParentPosition;
  orxObject_GetWorldPosition(pstParentObject, &vParentPosition);
  vPosition.fZ = vParentPosition.fZ;
  orxObject_SetPosition(pstParentObject, &vPosition);
}
```

The only thing left to do is to apply scale and rotation according to our inputs.
In our case, we defined the following inputs in 03_Frame.ini: `RotateLeft`, `RotateRight`, `ScaleUp` and `ScaleDown`.

Let's see how we handle them. First, the rotations.

```
if(orxInput_IsActive("RotateLeft"))
{
  orxObject_SetRotation(pstParentObject,
orxObject_GetRotation(pstParentObject) + orx2F(-4.0f) * _pstClockInfo->fDT);
}
if(orxInput_IsActive("RotateRight"))
{
  orxObject_SetRotation(pstParentObject,
orxObject_GetRotation(pstParentObject) + orx2F(4.0f) * _pstClockInfo->fDT);
}
```

And now, the scales.

```
if(orxInput_IsActive("ScaleUp"))
```

```
{
  orxObject_SetScale(pstParentObject, orxVector_Mulf(&vScale,
orxObject_GetScale(pstParentObject, &vScale), orx2F(1.02f)));
}
if(orxInput_IsActive("ScaleDown"))
{
  orxObject_SetScale(pstParentObject, orxVector_Mulf(&vScale,
orxObject_GetScale(pstParentObject, &vScale), orx2F(0.98f)));
}
```

That's all! Our `ParentObject` will be updated and all his children with it.

NB:

- *We could have used config values instead of constants for the rotation and scale values. This way, we could change them without having to recompile and even update them in real time by using the interactive console.*
- *As we use the clock's DT for the rotations, they will benefit from time consistency [4]. Unfortunately, that won't be the case for the scales. (Which is usually something we really don't want!)*

Resources

Source code: 03_Frame.c

Config file: 03_Frame.ini

1)
an empty object, with no visual
2)
`Object3` and `Object4`
3)
Remember to always use the main clock for callbacks that will handle inputs!
4)
they won't depend on frame rate and they will be time-stretchable