FX tutorial

Summary

See previous basic tutorials for more info about basic object creation, clock handling, frames hierarchy, animations, cameras & viewports and sounds & musics.

This tutorial shows what FXs are and how to create them.

FXs are based on a combination of curves [1] applied on different parameters such as scale, rotation, position, speed, alpha and color.

FXs are set through config file requiring only one line of code to apply them on an object.
There can be up to 8 curves of any type combined to form a single FX.
Up to 4 FXs can be applied on the same object at the same time.

FXs can use absolute or relative values, depending on the `Absolute` attribute in its config.
Control over curve period, phase, pow and amplification over time is also granted.
For position and speed FXs, the output value can use the object's orientation and/or scale so as to be applied relatively to the object's current state.
This allows the creation of pretty elaborated and nice looking visual FXs.

FX parameters can be tweaked in the config file and reloaded on-the-fly using backspace, unless the FX was specified to be cached in memory (cf. the `KeepInCache` attribute).
For example you won't be able to tweak on the fly the circle FX as it has been defined with this attribute in the default config file.
All the other FXs can be updated while the tutorial run.

As always, random parameters can be used to allow some variety for FXs.
For example, the wobble scale, the flash color and the “attack” move FXs are using limited random values.

We also register to the FX events so as to display when FXs are played and stopped.
As the FX played on the box object is tagged as looping, it'll never stop. Therefore the corresponding event (`orxFX_EVENT_STOP`) will never be sent.

We also show briefly how to add some personal user data to an `orxOBJECT` (here a structure containing a single boolean).
We retrieve it in the event callback to lock the object when an FX starts and unlock it when it stops.
We use this lock to allow only one FX at a time on the soldier.
It's only written here for didactic purpose.

Details

As usual, we begin by creating a viewport, getting the main clock and registering our `Update` function to it and, lastly, by creating our soldier and box objects.

Please refer to the previous tutorials for more details.

We then register to input and FX events.

```
orxEvent_AddHandler(orxEVENT_TYPE_FX, EventHandler);
orxEvent_AddHandler(orxEVENT_TYPE_INPUT, EventHandler);
```

As you can see, we are using the same callback (`EventHandler`) for both types of event.

Now let's have a quick look to our own “object” data structure.

```
typedef struct MyObject
{
  orxBOOL bLock;
} MyObject;
```

And now let's see how we bind it to our soldier using `orxObject_SetUserData()`.

```
MyObject *pstMyObject;

pstMyObject = orxMemory_Allocate(sizeof(MyObject), orxMEMORY_TYPE_MAIN);
pstMyObject->bLock = orxFALSE;

orxObject_SetUserData(pstSoldier, pstMyObject);
```

We now need so see how we apply FXs in our `Update` function.

```
orxSTRING zSelectedFX;

if(orxInput_IsActive("SelectWobble"))
{
  zSelectedFX = "WobbleFX";
}
else if(orxInput_IsActive("SelectCircle"))
{
  zSelectedFX = "CircleFX";
}

[...]

// Soldier not locked?
if(!((MyObject *)orxObject_GetUserData(pstSoldier))->bLock)
{
  if(orxInput_IsActive("ApplyFX") && orxInput_HasNewStatus("ApplyFX"))
  {
    orxObject_AddFX(pstSoldier, zSelectedFX);
```

```
  }
}
```

We can see we get our associated data back using `orxObject_GetUserData()` and that adding a FX to our soldier is done exactly the same way as adding a sound with `orxObject_AddFX()`.

Let's have a look to our `EventHandler` function.
First the input handling part, where we only display which keys have been used for every active input.

```
if(_pstEvent->eType == orxEVENT_TYPE_INPUT)
{
  if(_pstEvent->eID == orxINPUT_EVENT_ON)
  {
    orxINPUT_EVENT_PAYLOAD *pstPayload;

    pstPayload = (orxINPUT_EVENT_PAYLOAD *)_pstEvent->pstPayload;

    if(pstPayload->aeType[1] != orxINPUT_TYPE_NONE)
    {
      orxLOG("[%s] triggered by '%s' + '%s'.", pstPayload->zInputName,
orxInput_GetBindingName(pstPayload->aeType[0], pstPayload->aeID[0]),
orxInput_GetBindingName(pstPayload->aeType[1], pstPayload->aeID[1]));
    }
    else
    {
      orxLOG("[%s] triggered by '%s'.", pstPayload->zInputName,
orxInput_GetBindingName(pstPayload->aeType[0], pstPayload->aeID[0]));
    }
  }
}
```

As you can see, we display a different information depending if it a single key input or a combination. We only use the 2 first input entries as we know we didn't use combinations of more than 2 keys in our config file.
However orx supports up to 4 combined keys for a single input.
`orxInput_GetBindingName()` gives us a string version of an input such as `KEY_UP`, `MOUSE_LEFT` or `JOY_1` for example.
NB: Those are also the names used in the config file to bind keys, mouse or joystick buttons to inputs.

Let's now see how we handle the FX event.

```
if(_pstEvent->eType == orxEVENT_TYPE_FX)
{
  orxFX_EVENT_PAYLOAD *pstPayload;
  orxOBJECT           *pstObject;

  pstPayload = _pstEvent->pstPayload;
  pstObject  = orxOBJECT(_pstEvent->hRecipient);
```

```
  switch(_pstEvent->eID)
  {
    case orxFX_EVENT_START:
      orxLOG("FX <%s>@<%s> has started!", pstPayload->zFXName,
orxObject_GetName(pstObject));

      if(pstObject == pstSoldier)
      {
        // Locks it
        ((MyObject *)orxObject_GetUserData(pstObject))->bLock = orxTRUE;
      }
      break;

    case orxFX_EVENT_STOP:
      orxLOG("FX <%s>@<%s> has stoped!", pstPayload->zFXName,
orxObject_GetName(pstObject));

      if(pstObject == pstSoldier)
      {
        // Unlocks it
        ((MyObject *)orxObject_GetUserData(pstObject))->bLock = orxFALSE;
      }
      break;
  }
}
```

If a FX starts on our soldier we simply lock it using our own data structure. Reciprocally, we unlock it when an FX stops on it.

As we've covered the code part, let's see how we define FXs config-wise.
First let's have a look at a simple FX: the rotation one that loops on the box.

```
[RotateLoopFX]
SlotList  = Rotate
Loop      = true

[Rotate]
Type        = rotation
StartTime   = 0.0
EndTime     = 2.0
Curve       = sine
Pow         = 2.0
StartValue  = 0
EndValue    = 360

[Box]
FXList = RotateLoopFX
```

We can see above that we apply the FX (`RotateLoopFX`) on the box directly at its creation and not in code.
`RotateLoopFX` contains only one slot (`Rotate`), and is looping (attribute `Loop`).
We then define the `Rotate` slot. All times are expressed in seconds and angles in degrees.
We are basically defining a rotation that uses a square sine-shaped curve that will go from 0° to 360° over a period of 2 seconds.

Let's now look at the our wobble FX.

```
[WobbleFX]
SlotList = Wobble

[Wobble]
Type          = scale
StartTime     = 0.0
EndTime       = 1.0
Period        = 0.2
Curve         = sine
Amplification = 0.0
StartValue    = (1.0, 1.0, 1.0)
EndValue      = (2.0, 2.0, 1.0) ~ (6.0, 6.0, 1.0)
```

As you can see, we're now modifying the scale property. We're giving it a `StartValue` and a `EndValue`.
They are both expressed as vectors. They could have been expressed as floats if we didn't want any anisotropic value.
Even if it looks like we're using isotropic values [2)], the `EndValue` is a random one, ie. it's X and Y components might have different random values!
Beside that, the curve used is a simple sine over a period of 0.2 seconds, and playing it for one second.
As you can see, we also use an `Amplification` of 0. That means that at the end of our defined time (here it's when time reaches 1 second), the applied factor will be 0, which means the curve will decrease over time to finally reach an amplitude of 0 at second 1.
NB: The default `Amplification` is 1, which means the curve stays steady over time. Using a value < 1 will decrease it's amplitude and a value > 1 will increase it.

Let's now have a look at our circle motion.

```
[CircleFX]
SlotList        = CircleX#CircleY
KeepInCache     = true

[CircleX]
Type            = position
StartTime       = 0.0
EndTime         = 1.0
Curve           = sine
StartValue      = (0.0, 0.0, 0.0)
EndValue        = (-50.0, 0.0, 0.0)
```

```
UseOrientation  = true
UseScale        = true

[CircleY@CircleX]
Phase       = 0.25
StartValue  = (0.0, -25.0, 0.0)
EndValue    = (0.0, 25.0, 0.0)
```

Here we need to use 2 slots that affects the position so as to be able to have a circle motion.
The first slot, `CircleX`, will apply a sine curve on the X component of our object's position.
The second slot, `CircleY`, will apply the same curve (with a different amplitude) on its Y component.

This wouldn't result in a circle movement if we didn't alter the `Phase` of `CircleY`.
If we have to describe a sine curve, at phase 0, it's at its starting value (`StartValue`), ready to ramp up.
At phase 0.25, it's at its middle point, fully ramping up.
At phase 0.5, it's at its peek value (`EndValue`), about to ramp down.
At phase 0.75, it's now at its middle value again, fully ramping down.
At phase 1.0, it's exactly the same as at phase 0: `StartValue`, about to ramp up.
NB: This description works for a sine curve and also for a triangle one, but not for a linear one for example.

We'll now skip most of the other FXs as they don't have things we couldn't figure out by ourself.
However we'll have a quick look to the flipping one as it'll show us how we can flip an object, Paper Mario Wii-style.

```
[FlipFX]
SlotList = Flip

[Flip@Wobble]
EndTime       = 0.5
Period        = 1.0
Amplification = 1.0
EndValue      = (-1.0, 1.0, 1.0)
```

As you can see, we simply achieve it by using negative numbers! 
We can also note that we give an explicit value for the `Period`.
As we chose a `Period` twice as long as the length of our define curve, that means we'll only use half of the curve, ie. only the ramp up part of it.
We also chose to revert the `Amplification` back to 1 (as `Wobble` was setting it to 0).

Resources

Source code: 07_FX.c

Config file: 07_FX.ini

1)

with sine, triangle, square or linear shape

2)

the Z component doesn't affect 2D objects