Sound tutorial

Summary

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

This tutorial shows how to play sounds (samples) and musics (streams).
As with other features from previous behavior, it only requires, most of the time, a single line of code, everything being data driven.
This tutorial also demonstrates how to alter sound settings in real time, using the soldier graphic as a visual feedback.

If you press up & down arrows, the music volume will change accordingly. The soldier will be scale in consequence.
By pressing left & right arrows, the music pitch (frequency) will change. The soldier will rotate like a knob.
Left control key will play the music (and activate the soldier) if the music was paused, otherwise it'll pause the music (and deactivate the soldier).
Lastly, enter and space will play a sound effect on the soldier.
Space triggered sound effect is the same as enter except that its volume and pitch are randomly defined in the default config file.

This config-controlled frequency randomness allows to easily generate step or hit sounds with slight variations with no extra line of code.
We randomly change the soldier's color to illustrate this.
The sound effect will only be added and played on an active soldier.
If you want to play a sound effect with no object as support, you can do it the same way we create the music in this tutorial.
However, playing a sound on an object will allow spatial sound positioning (not covered by this tutorial).

Many sound effects can be played at the same time on a single object.

The sound config attribute KeepDataInCache allows to keep the sound sample in memory instead of rereading it from file every time.
This only works for non-streamed data (ie. not for musics).
If it's set to false, the sample will be reloaded from file, unless there's currently another sound effect of the same type being played.

We also register to the sound events to display when sound effects are played and stopped.
These events are only sent for sound effects played on objects.

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 object.

Please refer to the previous tutorials for more details.

We then create a music and play it.

```
orxSOUND *pstMusic;
pstMusic = orxSound_CreateFromConfig("Music");

orxSound_Play(pstMusic);
```

As we can see, music and sound are both of type `orxSOUND`. The main difference being that a music is streamed whereas a sound is completely loaded into memory. As we'll see below, we set this difference in the config file.

Last step of our `Init` function: we subscribe to the sound events.

```
orxEvent_AddHandler(orxEVENT_TYPE_SOUND, EventHandler);
```

As we only logs when sounds are started/stopped, let's have a look at the corresponding code.

```
orxSOUND_EVENT_PAYLOAD *pstPayload;

pstPayload = (orxSOUND_EVENT_PAYLOAD *)_pstEvent->pstPayload;

switch(_pstEvent->eID)
{
  case orxSOUND_EVENT_START:
    orxLOG("Sound <%s>@<%s> has started!", pstPayload->zSoundName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;

  case orxSOUND_EVENT_STOP:
    orxLOG("Sound <%s>@<%s> has stoped!", pstPayload->zSoundName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;
}

return orxSTATUS_SUCCESS;
```

Nothing really new here as you can see.

Let's now have a look to how we add sounds on our soldier.

```
if(orxInput_IsActive("RandomSFX") && orxInput_HasNewStatus("RandomSFX"))
{
  orxObject_AddSound(pstSoldier, "RandomBip");
  orxObject_SetColor(pstSoldier, orxColor_Set(&stColor,
orxConfig_GetVector("RandomColor", &v), orxFLOAT_1));
}

if(orxInput_IsActive("DefaultSFX") && orxInput_HasNewStatus("DefaultSFX"))
```

```
{
  orxObject_AddSound(pstSoldier, "DefaultBip");
  orxObject_SetColor(pstSoldier, orxColor_Set(&stColor, &orxVECTOR_WHITE,
orxFLOAT_1));
}
```

As we can see, adding a sound on an object requires only one line of code and, more important, it's done the same way for our random sound and our fixed-frequency one. As we'll see later, the difference is only expressed in the config file.
When adding a `RandomBip` sound we also change our soldier color with a random one defined in our config file with the key `RandomColor`. When playing a `DefaultBip` on it, we simply change its color back to white.

NB: A sound will be played on the soldier everytime the corresponding input is activated.
So far we only cared if an input was active or not, here we want to do some action only at the precise moment where the input is activated.
To do so, we use the function `orxInput_HasNewStatus()` that will return `orxTRUE` when the input goes from passive to active and, reciproquely, when it goes from active to passive.
Using the result from this function along the one we get from `orxInput_IsActive()` makes sure we only play the sound when the input is going from passive to active!

Now let's play with the music activation.

```
if(orxInput_IsActive("ToggleMusic") && orxInput_HasNewStatus("ToggleMusic"))
{
  if(orxSound_GetStatus(pstMusic) != orxSOUND_STATUS_PLAY)
  {
    orxSound_Play(pstMusic);
    orxObject_Enable(pstSoldier, orxTRUE);
  }
  else
  {
    orxSound_Pause(pstMusic);
    orxObject_Enable(pstSoldier, orxFALSE);
  }
}
```

This simple code, when activating the input `ToggleMusic`, will either start the music and activate our soldier if the music was not playing or, on the contrary, will stop it and deactivate our soldier if the music was playing.

Now, let's change the pitch.

```
if(orxInput_IsActive("PitchUp"))
{
  orxSound_SetPitch(pstMusic, orxSound_GetPitch(pstMusic) + orx2F(0.01f));
  orxObject_SetRotation(pstSoldier, orxObject_GetRotation(pstSoldier) +
orx2F(4.0f) * _pstClockInfo->fDT);
}
```

Nothing really surprising here. We do the same with opposite values for the input `PitchDown`.

Lastly, let's change the volume.

```
if(orxInput_IsActive("VolumeDown"))
{
  orxSound_SetVolume(pstMusic, orxSound_GetVolume(pstMusic) - orx2F(0.05f));
  orxObject_SetScale(pstSoldier, orxVector_Mulf(&v,
orxObject_GetScale(pstSoldier, &v), orx2F(0.98f)));
}
```

Same as for the pitch, nothing unusual.

NB: We can notice that only our object rotation will be time-consistent (cf. clock tutorial).
The music's pitch and volume, as well as the object's scale will be framerate-dependent, which is a bad idea.
To fix this we'd only need to use the clock's DT to ponderate the given values.

We're done with the code. So let's now have a look to the data!

First, let's define our music.

```
[Music]
Music = ../../data/sound/gbloop.ogg
Loop  = true
```

Easy enough! If we hadn't asked it explicitely, the music wouldn't have looped when played.

Let's now see our `DefaultBip`.

```
[DefaultBip]
Sound       = ../../data/sound/bip.wav
KeepInCache = true;
Pitch       = 1.0
Volume      = 1.0
```

As told before, the `KeepInCache` attribute will make sure this sound will never been unloaded automatically from memory.
The pitch and volume are explicitely defined to their default value which isn't actually needed.

Lastly, let's see our `RandomBip`.

```
[RandomBip@DefaultBip]
Pitch   = 0.1 ~ 3.0
Volume  = 0.5 ~ 3.0
```

As we can see, our `RandomBip` inherits from `DefaultBip`. This means that if we change the sample for `DefaultBip`, it'll also be changed for `RandomBip`.
Beside that, we only ask for random values for the `Pitch` (ie. frequency) and the `Volume`.
This means that everytime the `RandomBip` will be played, it'll have a different frequency and volume, but nothing is required codewise to get this behavior.

Resources

Source code: 06_Sound.c

Config file: 06_Sound.ini