Spawner tutorial

Summary

This tutorial is showing how to use spawners.
NB: If you want to see how to use orx while using C++ for your game, please refer to the localization tutorial.

See previous basic tutorials for more info about basic object creation, clock handling, frames hierarchy, animations, cameras & viewports, sounds & musics, FXs, physics, scrolling and C++ localization.

This tutorial shows how to create and use spawners for particle effects.
It's only a tiny possibility of what can be achieved using them.
For example, they can be also used for generating monsters or firing bullets, etc...

The code here is only used for two tasks:

- creating 1 main object (a scene) and a viewport
- switching from one test to another one by reloading the appropriate config files

Beside that, this tutorial is completely data-driven: the different test settings and the input definitions are stored in the config files along all the spawning/move/display logic.

With this very small amount of lines of code, you can have an infinite number of results: playing with physics, additive/multiply blend, masking, speed/acceleration of objects, etc...
It's all up to you! All you need is changing the config files and you can even test your changes without restarting this tutorial: config files will be entirely reloaded when switching from a test to another.

If there are too many particles displayed for your config, just turn down the amount of particles spawned per wave and/or the frequency of the waves.
To do so, search for the `WaveSize`/`WaveDelay` attributes in the different spawner sections. Have fun!


Details

Let's begin with a quick look to our `main` function.

```
int main(int argc, char **argv)
{
  orx_Execute(argc, argv, Init, Run, Exit);

  return EXIT_SUCCESS;
}
```

Nothing new here, we only execute orx using the helper `orx_Execute` function, providing three

callbacks: `Init()`, `Run()` and `Exit()`.

Let's now have a glimpse to our `Init()` function.

```
orxSTATUS orxFASTCALL Init()
{
  return LoadConfig();
}
```

We simply call our `LoadConfig()` function that will be described below.

```
void orxFASTCALL Exit()
{
}
```

As you can see, our `Exit()` function is even shorter as we will let orx the task of cleaning everything when we quit. 

```
orxSTATUS orxFASTCALL Run()
{
  orxSTATUS eResult = orxSTATUS_SUCCESS;

  if(orxInput_IsActive("NextConfig") && orxInput_HasNewStatus("NextConfig"))
  {
    ss32ConfigID = (ss32ConfigID < orxConfig_GetListCounter("ConfigIDList")
- 1) ? ss32ConfigID + 1 : 0;

    LoadConfig();
  }
  else if(orxInput_IsActive("PreviousConfig") &&
orxInput_HasNewStatus("PreviousConfig"))
  {
    ss32ConfigID = (ss32ConfigID > 0) ? ss32ConfigID - 1 :
orxConfig_GetListCounter("ConfigIDList") - 1;

    LoadConfig();
  }
  else if(orxInput_IsActive("Quit"))
  {
    eResult = orxSTATUS_FAILURE;
  }

  return eResult;
}
```

In our `Run()` function, in addition to quitting when the input `Quit` is active, we update our global `ss32ConfigID` which is an index to our currently selected config, depending on our inputs `PreviousConfig` & `NextConfig` status.
They'll then both call `LoadConfig()` that will reload everything.

Let's now have a closer look to `LoadConfig()`.

```
static orxINLINE orxSTATUS LoadConfig()
{
  orxSTATUS eResult = orxSTATUS_FAILURE;

  if(pstScene)
  {
    orxObject_Delete(pstScene);
    pstScene = orxNULL;
  }
  if(pstViewport)
  {
    orxViewport_Delete(pstViewport);
    pstViewport = orxNULL;
  }

  orxConfig_Clear();
  orxConfig_Load(orxConfig_GetMainFileName());
  orxConfig_SelectSection("Tutorial");

  if(ss32ConfigID < orxConfig_GetListCounter("ConfigList"))
  {
    orxSTRING zConfigFile;
    zConfigFile = orxConfig_GetListString("ConfigList", ss32ConfigID);

    if((eResult = orxConfig_Load(zConfigFile)) != orxSTATUS_FAILURE)
    {
      pstViewport = orxViewport_CreateFromConfig("Viewport");
      pstScene = orxObject_CreateFromConfig("Scene");
    }
  }

  return eResult;
}
```

As you can see we first delete our `Viewport` and our `Scene` object, if needed.
We then clean the whole config content and reload our main config file [1].
We then try to find the current config settings we want from the `ConfigList`.
If we've found it, we then load the corresponding file.
Some config files will add new properties, some will override basic ones (like the `Viewport` background color or the content of the `Scene` object).
Finally we (re-)create our `Viewport` and our `Scene` object.

And we're done for the code! =D

Again, nothing really specific here to spawners or particle handling: all the magic will come from the config files. 

In the same way we did for the C++ localization tutorial, the config file in our subfolder [2] merely

includes the config file from the parent folder.
We'll then skip this one and directly have a look to the parent folder's one.

As you can see by looking at the config file, we provide values for all basics system (`Display`, `Physics`, `Input`, `Viewport`, etc...).
As we've already covered all this in previous tutorials, we won't do it here again.

However, there's an additional section called `Tutorial`. Let's have a closer look.

```
[Tutorial]
ConfigList =
../11_Base.ini#../11_Blend.ini#../11_Mask.ini#../11_Physics.ini#../11_BlendMask.ini#../11_BlendPhysics.ini#../11_FusionFountain.ini#../11_MeltingDots.ini#../11_Shader.ini#../11_BlendShader.ini
```

If you remember, in our `LoadConfig()` function, we were selecting the `Tutorial` section [3] to look at the property called `ConfigList`.
This `ConfigList` provides our program with a list of config files, one file for each particle settings.
We then loaded the corresponding file manually by calling `orxConfig_Load()`.

*NB: All paths defined in config files are relative to the current directory **at runtime**. By default, it's the executable's one, hence the use of "`../`" in front of all our files.*

Let's now look at some of them [4].

We begin with the most basic one: `11_Base.ini`.

```
@../11_ParticleBase.ini@

[Tutorial]
Name = Base
```

The first line tells orx to include `../11_ParticleBase.ini` which will be instantly loaded and processed.
We then extend our `Tutorial` section by adding a property called `Name` that contains the value `Base`.

As we'll see later, we'll use this property to display on screen the current setting's name. 

Let's see another one, such as `11_FusionFountain.ini`.

```
@../11_ParticleBase.ini@
@../11_ParticleBlend.ini@
@../11_ParticleMask.ini@
@../11_ParticleDot.ini@

[Tutorial]
Name = FusionFountain
```

As we can see, it's built the same way as `11_Base.ini`, except that we now load 4 config files instead of a single one and that we provide a different name for `Tutorial.Name`.

NB: The file include order is important as the last included file might override properties defined in the first included one, which will happen in our case as we'll see later.

Let's now see `11_ParticleBase.ini` which is included in all our settings.
We'll only cover the interesting details and skipp all the fx/particle configuration.

```
[Scene]
ChildList = ParticleSpawner
```

Now we know exactly the content of our `Scene` object! 😀
Let's see what this `ParticleSpawner` is made of.

```
[ParticleSpawner]
ChildList = Name
Spawner   = Spawner
Position  = (0, 200, 0)
```

Another child here! This one's called `Name`.
There's also an attribute called `Spawner` which is defined.

Let's have a very quick glance at the `Name` object.

```
[Name]
Graphic   = NameGraphic
Position  = (0, 50, 0)

[NameGraphic]
Pivot = Center
Text  = NameText

[NameText]
String = @Tutorial.Name
```

As we can see, `Name` is simply a text object, however it's content points to `Tutorial.Name`.
As we just saw, `Tutorial.Name` is different for each config setting.
This means that every time we create a `ParticleSpawner` object, we'll display the name of the current setting just next to it, thanks to the relative positioning in a parent/child relation. 😀

Let's get back to our `Spawner`.

```
[Spawner]
Object      = Particle
WaveSize    = 5
WaveDelay   = 0.01
```

Spawners can use far more attributes than the three defined above as we can see in https://github.com/orx/orx/blob/master/tutorial/bin/CreationTemplate.ini, but those are the most important ones.

First, the `Object` attribute tells the spawner which kind of object it's going to spawn. In this case it's an object which is called `Particle` and that we won't describe here [5].

We can also learn from this spawner that it will spawn 5 of these `Particle` every 0.01 seconds by looking at the `WaveSize` and `WaveDelay` attributes.

So that's about it, our `Spawner` will spew 500 `Particle` every second as long as it's active.
Gladly the `Particle` object has a `LifeTime` attribute of 2.0 seconds which means we won't have more than 1000 `Particle` existing at the same time.

There are a lot of other attributes to have a better control over our `Spawner`, such as limiting the total number of object spawned, or the maximum number of existing objects at the same time, etc...

As we saw for the `FusionFountain` setting, we also load other config files.
Let's then have a look to them.
We'll begin with `11_ParticleBlend.ini`

```
[ParticleGraphic]
Texture = ../../data/object/particle2.png

[FadeOut]
Type        = color
StartValue  = (0, 0, 0) ~ (-50, -50, -50)
EndValue    = (-255, -255, -255)

[Particle]
Graphic   = ParticleGraphic
BlendMode = add
Color     = (255, 255, 0) # (0, 255, 0) # (255, 0, 255) # (0, 255, 255)
Position  = (-40, 0, 0) ~ (40, 0, 0)

[Spawner]
WaveSize = 8

[ParticleFX]
SlotList = FadeOut#Gravity
```

All those defined properties are actually overriding the ones defined in `11_ParticleBase.ini`.

We can see that we want to spawn more `Particle` objects in every `Spawner`'s wave [6].
We can also see that we change the `Texture`, the `BlendMode` and even the FX of our `Particle`, etc...

Let's now look at `11_ParticleMask.ini`

```
[Scene]
ChildList = ParticleSpawner # Mask

[MaskGraphic]
Texture   = ../../data/object/mask.png
Pivot     = center
```

```
BlendMode = multiply

[Mask]
Graphic   = MaskGraphic
Position  = {0, 0, -0.1}

[Particle]
Color = (255, 255, 255)
```

The most important information we can see here is that our Scene object now has two children instead of one.
This means that, in addition to our ParticleSpawner object, we also create an object called Mask which will use a multiply BlendMode and will displayed on top of the spawned Particle objects [7].
We also change the color of our Particle so that it will always be white.

Finally, let's see 11_ParticleDot.ini.

First we can see that we change our Viewport background color.

```
[Viewport]
BackgroundColor = (200, 200, 200)
```

As it's now a light grey, we also change the Name color to black.

```
[Name]
Color = (0, 0, 0)
```

And we make sure we spawn only 5 Particle objects per wave.

```
[Spawner]
WaveSize = 5
```

We also add a child to our Particle

```
[Particle]
ChildList = ParticleDot
```

That means that every Particle will have its own ParticleDot child that will follow it everywhere it goes! No more lonely Particle! 😬
Let's have a closer look to this ParticleDot.

```
[ParticleDotGraphic]
Texture = ../../data/object/particle.png
Pivot   = center

[ParticleDot]
Graphic   = ParticleDotGraphic
Position  = (0, 0, -0.001)
Alpha     = 1.0
```

```
Scale      = 0.9
```

We can see that this dot will be placed in front of each `Particle` [8] and that it will be a bit smaller than `Particle` as a relative scale of 0.9.
As they both use the same `Texture`, even if we scale `Particle`, `ParticleDot` will always have 0.9 times the size of its parent `Particle`.

That's it for particles, let's now have a quick look to `11_ParticleShader.ini`.

```
[Viewport]
ShaderList = Decompose
```

Here we add a shader called `Colorize` to our `Viewport`.

```
[Decompose]
Code = "void main()
{
  float fRed, fGreen, fBlue;

  // Computes positions with offsets
  vec2 vRedPos    = vec2(gl_TexCoord[0].x + offset.x, gl_TexCoord[0].y +
offset.y);
  vec2 vGreenPos  = vec2(gl_TexCoord[0].x, gl_TexCoord[0].y);
  vec2 vBluePos   = vec2(gl_TexCoord[0].x - offset.x, gl_TexCoord[0].y -
offset.y);

  // Red pixel inside texture?
  if((vRedPos.x >= 0.0) && (vRedPos.x <= 1.0) && (vRedPos.y >= 0.0) &&
(vRedPos.y <= 1.0))
  {
    // Gets its value
    fRed = texture2D(texture, vRedPos).r;
  }

  // Green  pixel inside texture?
  if((vGreenPos.x >= 0.0) && (vGreenPos.x <= 1.0) && (vGreenPos.y >= 0.0) &&
(vGreenPos.y <= 1.0))
  {
    // Gets its value
    fGreen = texture2D(texture, vGreenPos).g;
  }

  // Blue pixel inside texture?
  if((vBluePos.x >= 0.0) && (vBluePos.x <= 1.0) && (vBluePos.y >= 0.0) &&
(vBluePos.y <= 1.0))
  {
    // Gets its value
    fBlue = texture2D(texture, vBluePos).b;
  }
```

```
  // Outputs the final decomposed pixel
  gl_FragColor = vec4(fRed, fGreen, fBlue, 1.0);
}"
ParamList = texture#offset
offset    = (-0.05, -0.05, 0.0) ~ (0.05, 0.05, 0.0); <= Let's take some
random offset
```

As you can see, we directly write our shader's code in the `Code` attribute.
In our case it's a trivial code that will decompose the color of a pixel into 3 channels and mix them with its neighbors.

After defining the code [9], we provide a parameter list called `ParamList`.
Here we define 2 parameters: `texture` and `offset`.
As you can see we give random values for the `offset` parameter. The random values will be picked when the shader `Decompose` is created.

We can also see that we didn't define our `texture` paremeter. This means its content will default to the current parent's texture: here it's our `Viewport` texture [10].

In our case all the parameters are code-independent, ie. they won't change at runtime.
However we can add runtime parameters that we can change on-the-fly, but this won't be covered in detail by this tutorial.

In order to do so, we need to add the attribute `UseCustomParam` with a value set to `true`.
If we do that, every time our shader will be executed, orx will send an `orxEVENT_TYPE_SHADER` event [11].
This event's payload will contain the parameter name [12], its type and its config default value.
You can then change its value on-the-fly to suit your needs.
There's a simple sine wave distortion example that you can see in the bounce demo which is included in orx's source package, source and config.

Resources

Source code: 11_Spawner.c

Config file: 11_Spawner.ini

1)
so as to start on a fresh base that will be modified depending on which files will be loaded next
2)
which also contains the executable
3)
calling `orxConfig_SelectSection()`
4)
they are all built the same way
5)
it's mainly a graphical object with varying color, shape and physical properties that we use as a...

particle! 😀
6)
the value previously defined was 5

7)

the `Particles` are spawned with a default Z=0.0 whereas `Mask` has a Z=-0.1, which is closer to the camera

8)

due to the relative Z=-0.001 positionning

9)

which has to be a valid GLSL code

10)

if the shader was attached on an object instead of on a viewport, `texture` will be the object's texture instead

11)

with the `eID == orxSHADER_EVENT_SET_PARAM`

12)

here it'd be `texture` or `offset`

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

Permanent link:
https://www.orx-project.org/wiki/en/tutorials/spawners/spawner?rev=1598876888

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