SFX

BigWorld Technology 2.1. Released 2012.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

The FX module implements a data-driven special effects framework. Special effects are defined in an XML document, and loaded by the FX module on cue. The SFX files tie together not just particle effects, but all related components, for example sounds, models, decals, camera shake etc. This is designed to hide a lot of simple Python code that is often duplicated in entity scripts, which can frequently be error-prone.

It is also designed to implement best-practices regarding special effects in BigWorld. For example, it will load resources in the background to avoid causing i/o stalls in the rendering thread. It implements circular buffers of often-used effects to minimise the wait time and manage memory usage. It manages a number of conditions often encountered by special effects code, for example, it keeps track of where effects are attached to the world, so they can be cleaned up properly, even if the entity the effect is attached to leaves the world during the playing of the effect. And it allows customisation of the effect at run-time.

It is designed to be extensible for your game; simply implement your own Events and register them with the class factory and they will be availble for use by the effects system.

Chapter 2. SFX

2.1. How SFX Works

The FX module itself is based on 3 major interfaces; `Actors`, `Joints` and `Events`.

- `Actors` represent the resources used by an effect, for example Particle Systems, Models and Sounds.
- `Joints` determine how the `Actors` are attached in the world, for example on Hard- Points, Nodes, or assigned as Entity Models.
- `Events` describe how the effect will play out, and there are many options to choose from.

There are 4 main API functions the programmer needs to know about. These are:

- `FX.prerequisites()` This method returns a list of all the resources required to be loaded before an effect can be played. It is designed to be passed directly into the `BigWorld.loadResourceListBG()` method. The resulting resource dictionary can then be passed directly back into the FX module to construct effects.
- `FX.OneShot()` This method creates a once-off effect, that will automatically attach itself in the world, play the effect, and detach itself.
- `FX.bufferedOneShot()` This method also plays a one-shot effect, but internally it manages a fixed-size circular buffer and plays the first available one. This is especially important for high-frequency effects such as bullet "pchangs", where you need instantaneous access to an effect, and you need to cap the number that may exist at any one time for memory or performance reasons. The FX system keeps track of how many of each type of buffered effect was asked to play, and can report whether there was a buffer overrun, and how large the buffer should have been in order to satisfy all requests for the effect at any one time.
- `FX.Persistent()` This method creates an effect designed to be attached for a long amount of time, in particular, over the course of many plays of the effect. Additionally it is used for effects that do not have a specified duration, i.e. do not end by themselves.

2.2. Event Timings

All FX are started over 3 frames, not just 1.

Firstly, immediate events are played. There is nothing special to say about these.

Secondly, transform dependent events are played. These events require exact knowledge of where the effect is to be played in the world. Due to the way nodes are accessed in python, there can be one frame between accessing a node, and that node having a valid world transform stored in it. Since many actors are attached to nodes or hard-points, most events need to wait a single frame after attachment before they play. An example is simply spawning a particle. If the particle system's node is not yet valid, the particles will be spawned in the wrong spot, e.g. at the world origin.

Finally, duration dependent events are played. These events need to know the duration of the entire effect, before they are initiated. An example would be a light fade-out event that needs to be timed to finish with the rest of the effect.

Note that the effects system currently does not implement a timeline for the effect, but this would be the next most obvious extension (along with an effects sequencer.) Currently it only supports the concept of a random delay for events.

2.3. Viewing an Effect In-game

Let's make an effect show in-game. This will attach a "matrix swarm" effect to the `PlayerAvatar`'s body. Start up Fantasy Demo in offline mode and open the Python console.

At this point you might prefer to prevent the background from being dark, so that it's easier to see the effect play. Press ~ + `F9` to turn off background darkening.

Type the following to load and play the `person_explosion` on the player:

```
import FX
s = FX.OneShot("sfx/person_explosion.xml", $p.model, $p)
s.go($p, $p)
```

Note

`$p` is an alias for `BigWorld.player()` in the Fantasy Demo Python console (it get's the current player).

You should see the `"person_explosion"` *matrix swarm* on the avatar's body.

2.4. Writing FX Files

Please see the "`<fx>.xml`" for a full description of the FX file grammar.

See the Content Creation Guide chapter on Particles and Special Effects for a description of how to create SFX with an artist.

The example SFX xml spec

```
<!--
  Actors in the effect system.
  These are the resources used by events.
-->
<Actor> muzzleSmoke
  <ParticleSystem>
    sets/global/fx/particles/muzzle_smoke.xml
  </ParticleSystem>
</Actor>

<Actor> muzzleFlash
  <Model>
    sets/global/fx/actors/muzzle_flash.model
  </Model>
</Actor>
```

```
<Actor> healingBeam
  <ParticleSystem>
    sets/global/fx/particles/healing_beam.xml
  </ParticleSystem>
</Actor>

<!--
  Joints in the effect system.
  These attach the actors to the source upon invokation
  of the special effect.
-->
<Joint> muzzleSmoke
  <Node> HP_muzzle </Node>
</Joint>

<Joint> muzzleFlash
  <Hardpoint> muzzle   </Hardpoint>
</Joint>

<Joint> healingBeam
  <Node> HP_muzzle </Node>
</Joint>


<!--
  All events must return a duration.
  Thus the total length is known,
  and the Attach objects can be asked to Detach at the end.
-->
<Event>
 <SwarmTargets> healingBeam
  <Node> biped Neck </Node>
     .
     .
     .
 </SwarmTargets>
</Event>
<Event>
 <ForceParticle> muzzleSmoke </ForceParticle>
</Event>
<Event>
 <PlayAction> muzzleFlash
  <Action> Flash </Action>
 </PlayAction>
</Event>
<Event>
 <ForceParticle> healingBeam </ForceParticle>
</Event>
```

2.5. Using the SFX API

2.5.1. One-Shot SFX

The OneShot sfx will attach, and detach to the model automatically, all you have to do is create it and call `go( source, target, args... )` on the effect.

Creation is always safe (empty SFX can be created) so you never need to check.

eg. firing a gun once.

```
s = FX.OneShotSFX( fileName )
s.go( gun )
```

2.5.2. Buffered SFX

Some things like sparks happen very often, and thus you would not like to allocate particle memory every time.

Also, you'd like to cap the maximum amount of them in effect at any time, to limit any worst-case scenarios (especially with respect to memory).

eg. firing a gun once, but this happens often.

```
FX.bufferedOneShotSFX( fileName, gun )
```

2.5.3. Persistent SFX

Some effects should be run continuosly.

Once constructed, you need to call `attach( source )`, `go( target, args... )` and `detach()` on the effect.

eg. smoke continuously rising from a gun.

```
FX.Persistent( fileName )
s.attach( gun )
```

2.6. Post-Processing Special Effects

When adding post processing effects, you will most likely want some way to make them fade in and out. Use `PPAnimateProperty` on a phase in the chain. You can reverse the keys for fade in and fade out.

To pass in the location of the SFX source to a post processing effect, use `PPTranslationProperty`.