usfxr

usfxr is a C# library used to generate and play game-like procedural audio effects inside Unity. With usfxr, one can easily synthesize original sound in real time for actions such as item pickups, jumps, lasers, hits, explosions, and more, without ever leaving the Unity editor.

usfxr comes with code that allows for real-time audio synthesizing in games, and an in-editor interface for creating and testing effects before you use them in your code.

usfxr is a port of Thomas Vian's as3sfxr, which itself is an ActionScript 3 port of Tomas Pettersson's sfxr. And as if the acronym collection is not enough, it also supports additional features first introduced by BFXR such as new waveform types and filters.

This video explains the ideas behind as3sfxr, and the ideas supported by usfxr.

Introduction

First things first: if you're just looking for a few good (but static) 16 bit-style sound effects to use in games, anyone can use sound files generated by the original sfxr or as3sfxr's online version without any changes, since both applications generate ready-to-be-used audio files.

However, by using a runtime library like usfxr, you can generate the same audio samples in real time, or re-synthesize effects generated in any of those tools by using a small list of parameters (encoded as a short string). The advantages of this approach are twofold:

- Audio is generated in real time; there's no storage of audio files as assets necessary, making compiled project sizes smaller
- Easily play variations of every sound you play; adds more flavor to the gameplay experience

I make no claims in regards to the source code or interface, since it was simply adapted from Thomas Vian's own code and (elegant) as3sfxr interface, as well as Stephen Lavelle's BFXR additional features. As such, usfxr contains the same features offered by these two ports, such as caching of generated audio and ability to play sounds with variations. But because the code is adapted to work on a different platform (Unity), it has advantages of its own:

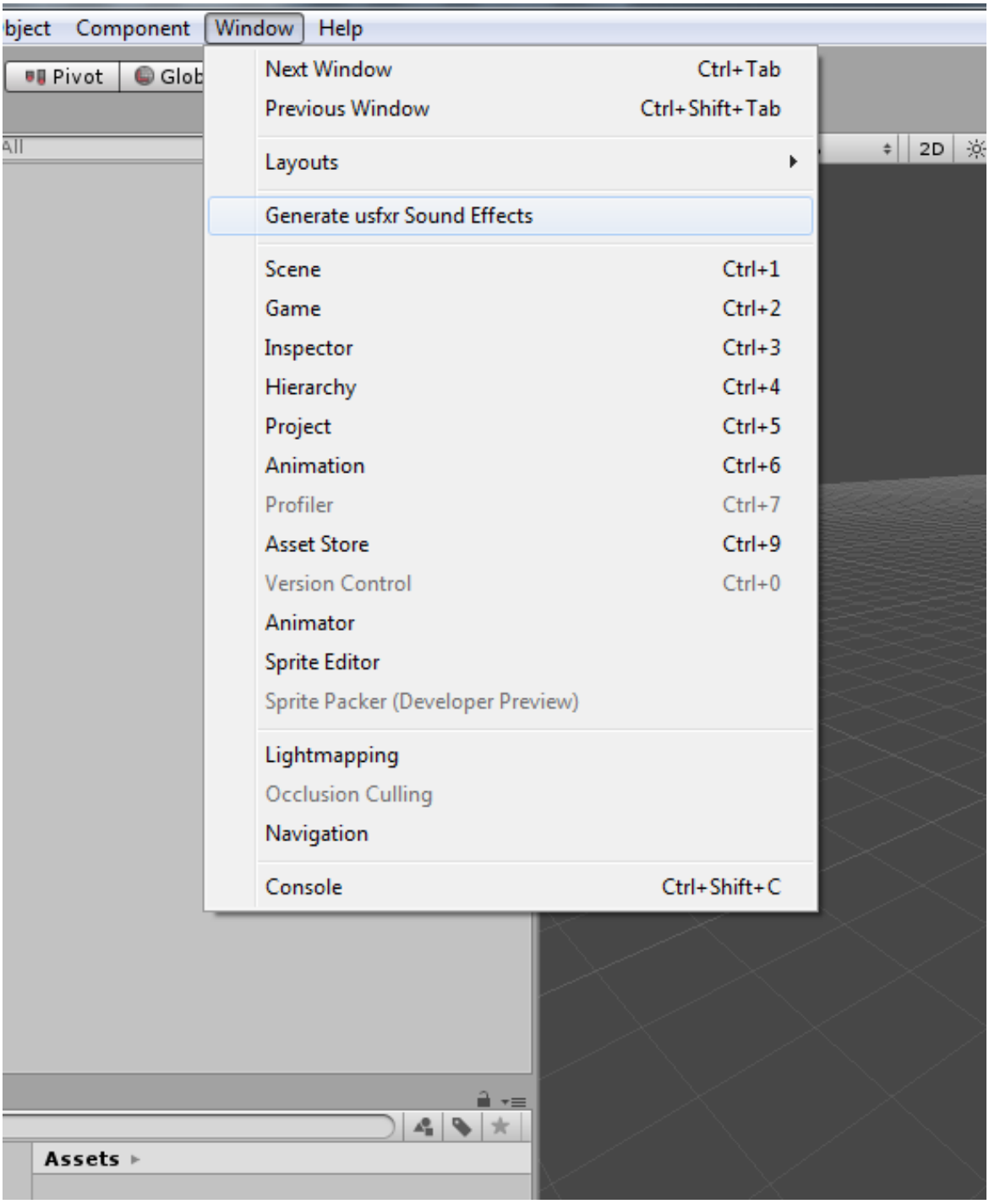
- Fast audio synthesis
- Ability to cache sounds the first time they're played
- Completely asynchronous caching and playback: sound is generated on a separate, non-blocking thread with minimal impact on gameplay
- Minimal setup: as a full code-based solution, no drag-and-drop or additional game object placement is necessary

- In-editor interface to test and generate audio parameters

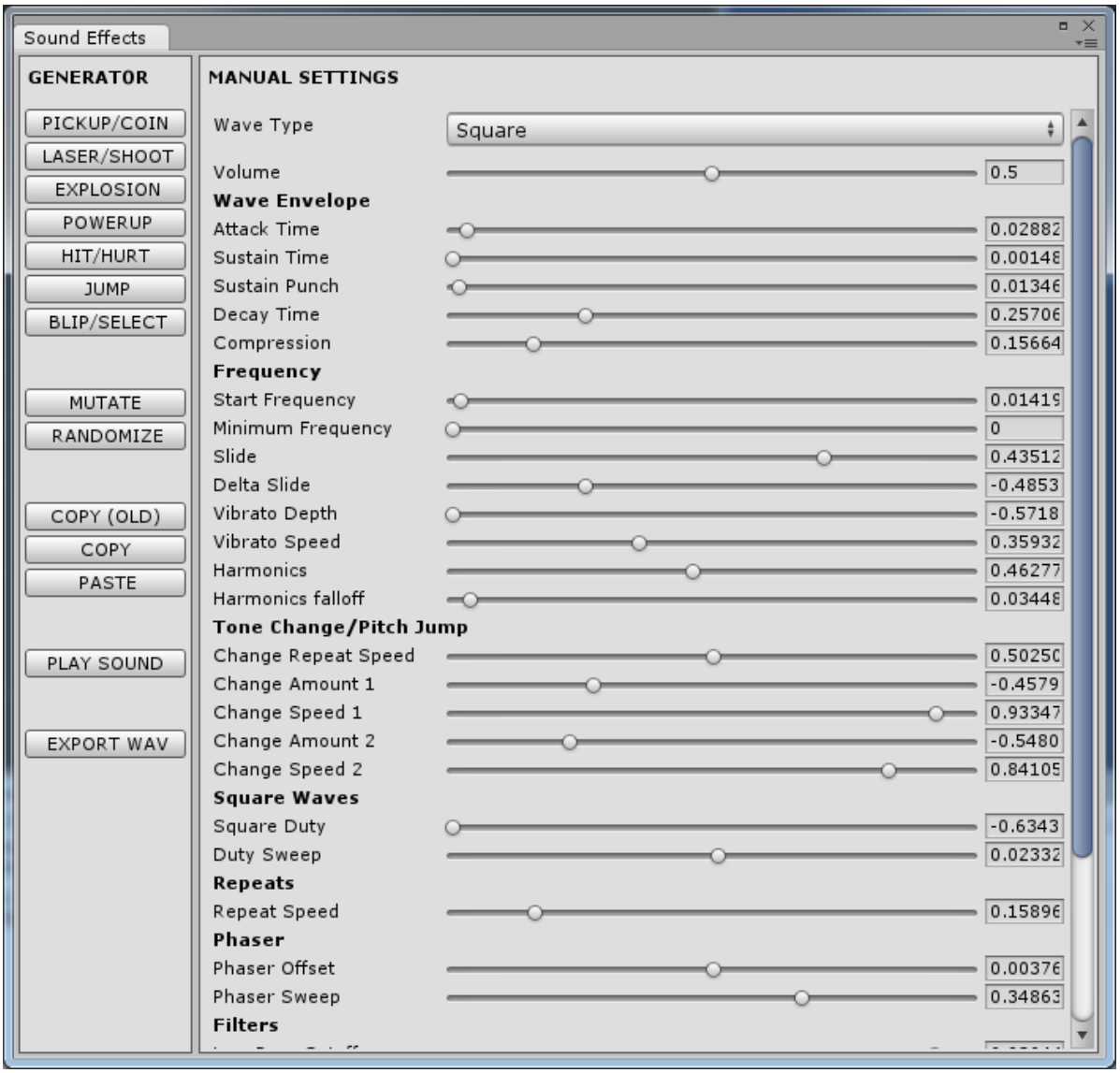
Usage

Typically, the workflow for using usfxr inside a project is as such:

1. Use the menu "Window" > "Generate usfxr Sound Effects" to open the sound effects window:



2. Play around with the sound parameters until you generate a sound effect that you want to use:



3. Click "COPY" to copy the effect parameters to the clipboard (as a string)

4. Write some code to store your sound effect, pasting the correct string:

```
SfxrSynth synth = new SfxrSynth();
synth.parameters.SetSettingsString("0,,0.032,0.4138,0.4365,0.834,,,,,,0.3117,0
.6925,,,,,,1,,,,,0.5");
```

Finally, to play your audio effect, you call its `Play()` method where appropriate:

```
synth.Play();
```

With usfxr, all audio data is generated and cached internally the first time an effect is played, in real time. That way, any potential heavy load in generating audio doesn't have to be

repeated the next time the sound is played, since it will already be in memory.

Because of that, while it's possible to generate new SfxrSynth instances every time they need to be played, it's usually a better idea to keep them around and reuse them as needed.

It's also important to notice that audio data generation does not occur all at once. This is a good thing: the audio data is generated as necessary, in batches, before playback of each 20ms chunk (more or less), so long audio effects won't take a lot of time to be generated. Audio is also generated on a separate thread, so impact on actual game execution should be minimal. Check `OnAudioFilterRead` for more details on how this happens.

Notice that, alternatively, you can also use the online version of as3sfxr to generate the parameter string that can be used when creating SfxrSynth instances. Strings generated by as3sfxr use the same format as usfxr so they're interchangeable.

Advanced usage: caching

Although this is rare, in case of long or numerous audio effects, it may make sense to cache them first, before they are allowed to be played. This is done by calling the cacheSound() method first, as in:

```
SfxrSynth synth = new SfxrSynth();
synth.parameters.SetSettingsString("0,,0.032,0.4138,0.4365,0.834,,,,,,0.3117,0
.6925,,,,,,1,,,,,0.5");
synth.CacheSound();
...
synth.Play();
```

This caches the audio synchronously, that is, code execution is interrupted until `synth.CacheSound()` is completed. However, it's also possible to execute caching asynchronously, if you provide a callback function. Like so:

```
SfxrSynth synth = new SfxrSynth();
synth.parameters.SetSettingsString("0,,0.032,0.4138,0.4365,0.834,,,,,,0.3117,0
.6925,,,,,,1,,,,,0.5");
synth.CacheSound(() => synth.Play());
```

In the above case, the `CacheSound()` method will immediately return, and audio start to be generated in parallel with other code (as a coroutine). When the audio is cached, the callback function (`Play()`, in this case) will be called.

As a reference, it typically takes around 7ms-70ms for an audio effect to be cached on a desktop, depending on its length and complexity. Therefore, sometimes it's better to let the game cache audio as it's played, or to stack the caching of all audio in the beginning of the gameplay, such as before a level starts.

An important notice when comparing to as3sfxr: the ActionScript 3 virtual machine doesn't normally support multi-threading, or parallel execution of code. Because of this, the

asynchronous caching methods of as3sfxr are somewhat different from what is adopted with usfxr, since Unity does execute code in parallel (through Coroutines or in a separate thread entirely (through `OnAudioFilterRead`). As such, your caching strategy on Unity normally won't have to be as robust as an ActionScript 3 project; in the vast majority of the cases, it's better to ignore caching altogether and let the library handle it itself by using `Play()` with no custom caching calls.

Advanced usage: setting the audio position

By default, all audio is attached to the first main `Camera` available (that is, `Camera.main`). If you want to attach your audio playback to a different game object - and thus produce positional audio - you use `SetParentTransform`, as in:

```
synth.SetParentTransform(gameObject.transform);
```

This attaches the audio to a specific game object.

Help and contact

For help and information, contact me at zeh@zehfernando.com. For the source code and more information about the project, including changelog, visit the project page on GitHub at https://github.com/zeh/usfxr.