Demos for a new Tool for Procedural Game Audio

Landahl, D. University of Portsmouth

Overview

This project implements the *Volsung* sound language (https://landahl.tech/volsung) as a native plugin for the Unity game engine. To run audio code, you make calls from C# to a native interface, which allows you to submit a program as a null-terminated string, have it be compiled into an audio processing graph, and then start drawing PCM samples from it while writing synthesis parameters.

Managed Interface and Editor UI

To streamline this process, a Unity C# wrapper script with a custom editor UI is provided. It implements necessary functionality and makes it easy to supply code through a text file and to control real-time parameters without risking crashes due to incorrect use of the API. Through this high-level managed interface, real-time parameters for the synthesis engine become regular Unity parameters in the editor, which can be controlled in the project using animation curves, scripts or any other method. The C# wrapper for the plugin implements a Unity AudioFilter, meaning it can be used to generate audio to be output by an AudioSource and then heard by an AudioListener, with spatial audio capabilities of Unity being available as normal.

Plugin User Code

Listed below is a minimal script which shows how to use the plugin through the Unity managed environment. The instance of a Volsung Unity plugin should be made to refer to a Volsung Unity script component attached to the game object. It will start processing audio once the `Compile` function has been called. To update a synthesis parameter, use the `SetParameter` function and pass a name that is registered through the plugin UI, and a value. That name will be exposed as an audio processing node in the Volsung audio processing graph, and will always output the value that has last been submitted through `SetParameter`. The user may wish to implement parameter smoothing in the audio code. This can be done using the `Smooth~` object (first-order infinite-impulse-response lowpass filter with adjustable

corner frequency), a biquadratic filter such as `Lowpass~` or `Bandpass~`, or any of the available filter design tools such as poles, zeros, and digital delay lines.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Example : MonoBehaviour
{
    Volsung_Unity plugin;
    void Start()
    {
        plugin = GetComponent<Volsung_Unity>();
        plugin.Compile();
    }

    void Update()
    {
        plugin.SetParameter("position", transform.position.x);
    }
}
```

Demos

Described below are the three demo scenes which utilize this wrapper to add sound to various video game-related contexts. They are designed to showcase technical capabilities, highlight creative use cases, and teach the process of implementing the plugin for users. Each demo section also has a “parameter mapping” subsection, which explains what data is moved from the Unity game engine into the Volsung language in the form of real-time parameters, to produce the sound for that scene.

Demo 1 - Spaceship

The first demo allows you to simulate different game states of a theoretical space game, while synthesising a correlated audio signal. It demonstrates how the spaceship engine sound can respond more dynamically to the state of the game than is possible using traditional sample-trigger based techniques, while mitigating

some compromise in artistic sound quality brought by similar synthesis-procedural based methods.

Audio Algorithm Description

The Volsung language is used in this project to specify a two-operator frequency modulation (FM) synthesis algorithm, with some added frequency filtering, sub oscillation, and nonlinear (distortion) processing. The audio processing is a function of the current movement speed of the spaceship, and the physical condition the spaceship is in. Below is a rudimentary implementation of FM in Volsung. Note that the code in the project uses differently-shaped oscillators and has a few extra nuances, but you can get a comparable sci-fi effect with this program.

```
; Specify basic parameters for synthesis
carrier_f: 220
ratio: 0.201

; Calculate modulator frequency from carrier and ratio
modulator_f: carrier_f * ratio

; Connect everything up
Sine_Oscillator~ modulator_f
-> index: Multiply~
-> Add~ carrier_f
-> Sine_Oscillator~
-> *0.5 -> output

; Have the modulation index increase proportional to
; the square of the elapsed time
Timer~ -> ^2 -> *100 -> 1|index
```

Game Design Implications

Setting the game state parameters to simulate a damaged spaceship (the spaceship would be controlled by the player) creates a dissonant and somewhat unpleasant sound. This can help motivate the player to participate in a basic gameplay-loop, such as going out to mine asteroids, but returning to the hub when the ship gets damaged as the player would want to avoid listening to their spaceship clearly

suffering through space. It may also increase the immersiveness of the game, since the player may be enticed to move more slowly when the ship is damaged, since a fast-moving and damaged spaceship compounds the dissonance of the synthesised sound. The spaceship could also be made to sound more modern or pleasant as it gets upgraded with better engines, which would motivate the player to progress through the game. Using traditional audio techniques, linking more of the game state into the sound of a game object results in an exponentially increasing number of required audio files, whereas with this technique it is possible to simply add small sections to the synthesis algorithm.

Parameter Mapping

- **Speed → Modulation index**: a faster moving spaceship quadratically increases the magnitude of the modulating oscillator, introducing more sidebands. When the movement speed is at the lowest setting, there is nearly no FM, and the carrier oscillator is close to a pure tone.
- **Speed → FM oscillator output volume**: a faster moving spaceship linearly increases the volume of the output of the FM process. When the movement speed is at the lowest setting, the FM carrier can barely be heard at all, and mainly the steady sub oscillator is audible.
- **Damage → Frequency filter resonance**: as the damage that has occured to the spaceship increases, a resonant peak at the corner-frequency of the filter creates a ringing tone, as if unfavourable resonances in the construction of the ship are excited by vibrations from the engine - something that would only happen to a damaged vehicle.
- **Damage → Sub oscillator frequency offset**: the sub oscillator becomes dissonant relative to the FM oscillator when the spaceship gets damaged. This introduces a slow modulation. When the spaceship is not damaged, the sub oscillator is at the carrier frequency, creating no modulation.
- **Damage → FM ratio**: the FM ratio (modulator frequency / carrier frequency) deviates from being mathematically simple (½) as damage increases. This makes the sidebands that are generated by the modulation more dissonant.

Demo 2 - Particle System

The second demo shows a particle system in Unity, where each individual particle executes a Volsung program to create sound. The system can support a large number of procedural audio synthesisers being created and destroyed each second in parallel. In order to use multiple procedural audio generators, we just instantiate more than one game object with the Volsung Unity C# script attached. The library can quickly compile the code at run-time and start generating sound.

Particle System Parameters

The user may control parameters of the particle system using sliders on screen. Since the positional data and lifetime of every particle is used to synthesise sound, changing the visuals of the particle system changes the sound. Because the position of the particle on-screen has the greatest impact on the synthesis process, there are several particle system parameters which affect the position. Firstly, there is the size of the range of random starting positions for each of the X and Y coordinates. Then, there are minimum and maximum random movement speeds for each particle along each axis. Thirdly, there is a vibration factor, which can make each particle move along a sinusoidal path on a certain axis with a certain magnitude and frequency.

Audio Algorithm Description

Each particle performs synthesis based on a pitched-noise technique. A stream of white noise is filtered through a bank of narrow bandpass filters wired in series, isolating one pitch at the centre frequency. The volume is controlled by an envelope section, the length of which is related to the lifetime of the particle. The X and Y position of each particle are related to the stereo panorama and frequency respectively. This demo utilises the stereo capabilities of the plugin. By selecting the stereo option on the script's custom editor UI, the audio code will be compiled as a processing graph with two ports on the output node, corresponding to the left and

right audio channels. If the stereo option is not selected, data from the single output port will be copied to both channels.

Parameter Mapping

- **Lifetime → Envelope length**: because a particle stays visible longer on the screen when its lifetime is longer, the sound is made to expire more gradually too. The envelope length is set to about 200 milliseconds for each second that the particle lives.
- **Y Position → Frequency**: particles that are closer to the top of the screen produce a higher pitch. This is easy to follow visually in cases of smaller numbers of particles. By using the sinusoidal motion parameters of the particle system, a vibrato effect with adjustable rate and depth can be created.
- **X Position → Stereo panning**: particles that move towards or are spawned on the left side of the screen produce louder audio in the left audio channel and vice versa.

Demo 3 - Physical Modeling

The third demo physically models the sonic behaviour of vibrating strings. The physical model is implemented by a Karplus-Strong based approach, with elements from the Extended Karplus Strong (EKS), algorithm (Smith, 2011). EKS elements include a pick position filter, and a dynamic-level lowpass filter. The 3D scene contains some pillars with strings connecting them. The player may press the left mouse-button to fire a projectile at the strings and excite vibration. A Unity callback procedure catches the collision event, and forwards some useful data to the synthesis algorithm, such as the collision magnitude and the normalised collision position, which is the distance of the collision along the length of the string from the middle of the string, in proportion to its entire length.

Audio Algorithm Description

The Volsung program defines an exciter subgraph and a resonator subgraph. An exciter generates an envelope for a short burst of noise, which gets moved into a resonator, which repeats a short section of audio and simulates a loss of energy on each cycle. A simple multiplication simulates unity gain loss of energy, and is connected in series with a pole, which reintroduces some of the energy in the low frequencies to simulate higher frequency energy components dissipating more quickly. The effect is often achieved using a zero at nyquist, but several zeros would be required to achieve a filtering effect with the same strength, and that would not be as computationally effective.

Potential for Further Development

This project could be developed further by having a damping mechanism when spheres or other objects rest on the strings. Unity has a callback called `OnCollisionStay`, which is called each frame if a collider is resting on another. This could be used to send more information to the string damping filters, and have the strings create a muted sound when they are excited while an object is resting on them. The amount of damping should again be proportional to the normalised collision position, since an object interacting near a vibrational node would cause less energy to be transmitted onto it than one interacting with the string near a vibrational anti-node.

Parameter Mapping

- **Magnitude → Volume**: a collision involving more force creates a louder sound, by generating an excitation signal of greater magnitude.
- **Magnitude → Dynamic level corner frequency**: since objects that are excited with greater force tend to generate a brighter sound, we let the collision magnitude correlate with the magnitude of higher frequency components.
- **Position → Position filter resonance**: we decrease the resonance as the position of the collision moves out, further from the middle of the string. A lower resonance means a shallower transition band between the passband

into the stopband, resulting in greater attenuation of low frequencies by the highpass filter. This is reflective of how real acoustic strings produce brighter sounds when excited near the nut or bridge (vibrational nodes) than when they get excited near the middle (anti-node).

- **String geometry → Frequency**: the frequency is determined based on the geometry of the string mesh in the 3D world. A longer string will be computed to have a proportionally lower frequency, which is information that gets provided to the synthesis algorithm at initialisation time. The model assumes that all strings have equal tension.

Note on performance

Some of the projects, especially the particle system simulation, suffer from some performance problems under high load (large numbers of particles being spawned and they live a long time). This is due to the currently unoptimised synthesis engine of Volsung. There are run-time exceptions and small dynamic allocations and deallocations happening in the audio thread, for purposes of debugging and prototyping language features. In the future, performance will be improved by preallocating all memory, and by not generating code to handle exceptions during audio processing.

Procedural audio can be expensive in general, but game designs which are identified as benefiting greatly from more interest in the audio domain will wish to make particular trade-offs. It's unreasonable to say that procedural graphics are too expensive, so games should just use static pre-rendered images. The reason this seems so unreasonable is because game designers understand the costs and benefits of popular procedural graphics techniques, and will factor a subconscious cost-benefit analysis of these into their ideas for games. It may be hard to see that this is happening, but no one starts thinking about making a game and doesn't understand what sort of graphics are reasonable to do. They would understand that it is not unreasonable to want to render complex graphics in real-time, but that the technology is not at a stage where convincing human faces, distance foliage, or fluid

simulations can be done easily. A game designer or technology lead who understands procedural audio would never say that it is too expensive without the context of a specific procedural audio technique and a specific game design.

References

Smith, J. (2011). *Physical audio signal processing*. [Lexington]: W3K Publishing.

Attributions

Spaceship image is by Vectorstock

Tripfive pixel font is by Pixelmush

Starfield skybox is by Pulsar Bytes

Standard assets by Unity Technologies