Virtual electric guitars and associated audio effects in Faust and C++

Julius O. Smith III (jos@ccrma.stanford.edu) , Nick Porcaro, and Nelson Lee
Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music, Stanford University
Stanford, California 94305

Poster 1674 — July 3, Acoustics 2008
Paris, FRANCE

Long-Term Objective:

Free, open-source, reference implementations for virtual musical instruments and associated audio effects

Immediate Objective:

Compare Faust and Synthesis Tool Kit (STK) programming tools

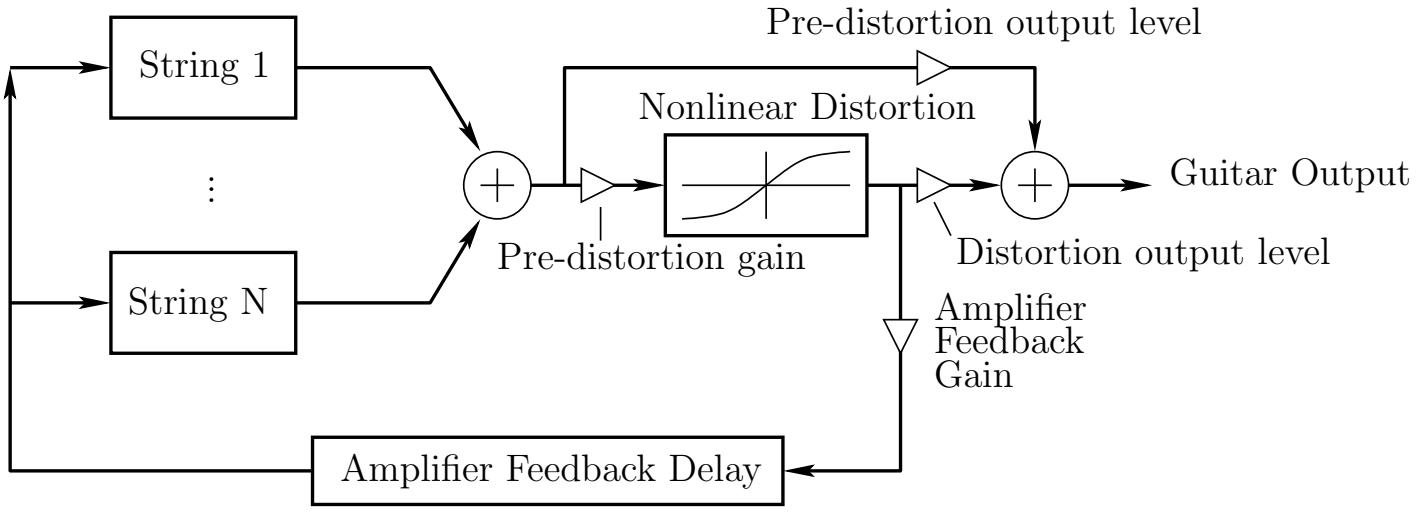
Prior Work

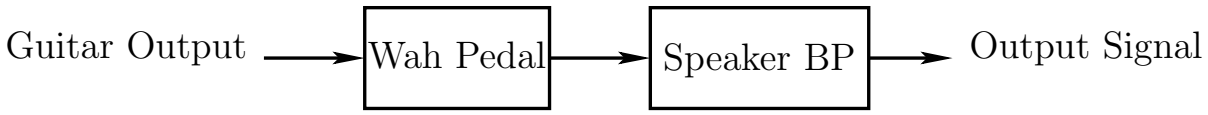
This poster can be considered a follow-on to

> *Virtual Electric Guitars and Effects Using Faust and Octave*, by J.O. Smith, Linux Audio Conference (LAC-2008), Cologne, March 2008.

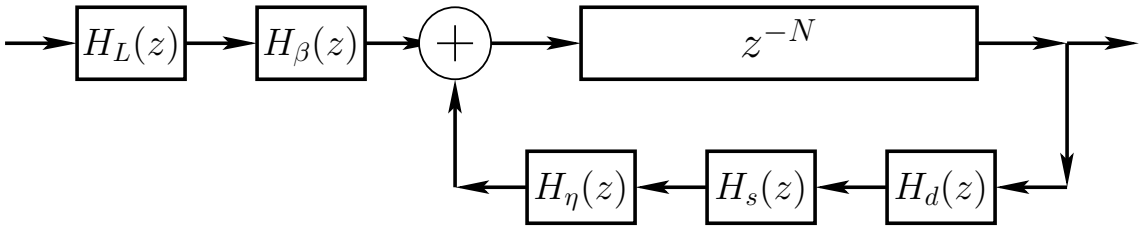
- That paper was focused on implementations in the Faust language by Yann Orlarey
- Resulting new Faust libraries:
  - `osc.lib`: four filter-based sinusoidal oscillators
  - `filter.lib`: variable elementary filter sections, Lagrange (FIR) and Thiran (allpass) interpolation
  - `effect.lib`: brightness filter, cubic distortion, Moog VCF, digitized CryBaby wah pedal, and Rauhala piano-string dispersion filter
- Pointers:
  - LAC-2008 Paper:
    `http://lac.linuxaudio.org/download/papers/22.pdf`
  - LAC-2008 Overheads:
    `http://lac.linuxaudio.org/download/slides/22/`
  - Supporting website:
    `http://ccrma.stanford.edu/realsimple/faust_strings/`

Virtual Electric Guitar Architecture





String Model



where (see third pointer above for full details) . . .

$$
\begin{aligned}
N &= \text{pitch period } (2\times \text{ string length}) \text{ in samples} \\
H_\beta(z) &= 1 - z^{-\beta N} = \text{pick-position comb filter, } \beta \in (0,1) \\
H_d(z) &= \text{string-damping filter (one/two poles/zeros typical)} \\
H_s(z) &= \text{string-stiffness allpass filter (several poles and zeros)} \\
H_\rho(z) &= \text{first-order string-tuning allpass filter} \\
H_L(z) &= \text{dynamic-level lowpass filter}
\end{aligned}
$$

STK Implementation

- Reference implementations at CCRMA are typically written directly in C++ using the Synthesis Tool Kit (STK) originated and maintained by Perry Cook and Gary Scavone
- Since the Faust implementations reported at LAC-2008, STK implementations have been developed (primarily by Nick Porcaro) for substantially the same virtual electric guitars and audio effects
- We wish to compare the relative merits of Faust and the STK as reference-implementation vehicles
- We also wish to *integrate* Faust and STK resources:
  - Incorporate STK objects into Faust programs
  - Create new STK objects from Faust compilations

Faust and STK Compared

Commonalities:

- Both specify C++ implementations
- Both support Linux, Mac OS-X, and Windows

Faust Advantages:

- Higher level language for signal processing and GUI specification
- Reference implementations are compact
- Automatic block diagram generation
- Multiprocessing support (in alpha)
- Strong optimization of generated C++ code
- Platform-independent specification
- Automatic generation of plugins for many platforms:
  - Pure data (Pd)
  - LADSPA
  - VST
  - SuperCollider

  - and others, as well as various stand-alone programs

  To add a new plugin or stand-alone program type, just create a new architecture file from one of the existing examples

Faust Disadvantages:

- Because Faust syntax is designed primarily for traditional left-to-right signal flow with occasional feedback, it can be unwieldy for specifying physical models which require "bidirectional" signal paths between physical blocks
- Some traditional programming constructs can be unwieldy to specify:
  - If-then conditional expressions
  - Table look-ups
- No vector support (yet) — signals are real or int
- Multiple sampling rates not supported (yet)

The Faust future-development road-map includes *vector* and *multirate signal* support. Tables will be facilitated by the planned extensions, as will block-based processing (such as FFTs)

C++/STK Advantages:

- More general language features
  (anything C++ can express)
- Potentially faster compiled code (since hand-written)

C++/STK Disadvantages:

- Lower level programming language
- Program source typically more voluminous
- Program specification is slightly platform-dependent
  (e.g., the `StkFloat` data type can be arbitrarily redefined)
- No built-in multiprocessing support
- GUI specification handled externally (in TCL/Tk) and more laboriously
- No automated plugin generation
  (no "architecture file" counterpart)
- More difficult for making Pd externals. The FLEXT interface by Thomas Grill helps - see, *e.g.*,
  `http://ccrma.stanford.edu/realsimple/stkforpd/`

Another Faust-STK commonality is that Pd externals consisting of a set of modules arranged in a feedback loop must normally be fused into a single external:

- Must use `send~` and `receive~` to form a signal processing loop (cannot simply connect Pd signal objects in a loop)
- Such a loop will have at least a 64-sample delay inserted (the current chunk size in Pd)
- Since module run-order cannot be specified in Pd, an unknown number of additional 64-samples of delay may be inserted as well
- As a result, we placed the entire STK instrument in a single Pd external (as in the Faust case)
- The FLEXT Pd external interface by Thomas Grill was used for the STK case (latest CVS required)
- The Faust `puredata.cpp` architecture file and companion Q script `faust2pd`, both by Albert Graef, make it very easy to generate Pd plugins from Faust programs. To provide such facility in the STK environment, one could define an STK instrument-definition language (analogous to instrument definitions in Music V, CSound, or SAOL), and a companion `stkins2pd` utility, for example.

Faust and STK Integration

We presently incorporate STK objects inside C-function wrappers that are declared as a Faust "foreign functions"

Simple example (white noise generator):

- Complete Faust program:

```
stknoise = ffunction(float StkNoise(int),
 "stkf/stkf.h", "stkf/libstkf.a,libstk.a");

seed = 10; // any int
process = seed : stknoise;
```

- Separately compiled C function:

```
#include <Stk.h>
#include <Noise.h>
Noise *theNoise;

float StkNoise(int seed) {
  if (theNoise == NULL) {
    theNoise = new Noise(seed);
  }
  return theNoise->tick();
  // Future: theNoise->tick(theFrame,0);
}
```

Limitations:

- A Faust `ffunction` must return a single `float`
- Arguments can only be type `float` or `int`
- No explicit state-initialization support
  (work-around: static init variable in C wrapper function, as in above example)
- No explicit instance support
  (work-around: use an integer signal argument to serve as "instance number" — argument `seed` in above example can be viewed in this way)

Future solution: The Faust road map includes "foreign objects" (`fobject`) and vector-valued signals

Faust and STK Benchmarks

We compared the run-times of Faust and STK implementations of elementary signal-processing modules used in the virtual electric guitar

Methodology

- The Faust architecture file `bench.cpp`, distributed with Faust, was used to measure all run-times

  - One block of 128 signal samples computed
  - Run time = elapsed time in cycles of system clock
  - Each benchmark defined as the minimum over 100 executions of the benchmark program
- STK modules integrated as foreign functions, as described above
  - The type `StkFloat` was changed from `double` to `float` throughout the STK by editing `Stk.h`
  - Since many STK modules support vectorization while `ffunctions` can only return a single `float`, each STK ffunction call was run internally for 128 samples, using the method `tick(frames,chan)` implemented in `Filter.cpp`, etc., and the resulting run-time reported by `bench.cpp` was divided by 128 (the number of samples in `frames`)
- C++ compiler optimization was set to `-O3` for the STK library `libstk.a`, the Faust-generated `<module>-bench.cpp` file, and for the separately compiled `ffunction` wrappers for STK classes
- C++ compiler = `gcc` version 4.1.2 (Fedora 8)
- Faust version 0.9.9.4f was used (CVS July 1, 2008)
- STK version 4.3.0 was used, with Cubicnl.cpp added

Results

Linear Congruential Random Number Generation (White Noise):

	Cycles
STK Module: `Noise.cpp`	12076
Faust Function: `noise` (`music.lib`)	692
Cycles Ratio: 17.45	

Note: The system `rand` function is called each sample in the STK version

Two-Pole, Two-Zero Filter:

	Cycles
STK Module: `BiQuad.cpp`	9151
Faust Function: `TF2` (`music.lib`)	1576
Cycles Ratio: 5.81	

Notes:

- The STK BiQuad class uses the C++ `vector` class from the Standard Template Library (STL) for both filter coefficients and input/output signal history
- The vectorized `tick` method could be reimplemented for greater speed. Its overhead nearly negates the value of vectorized processing
- Coefficients were fixed. Updating them once per signal block would add little additional computation.

Linearly Interpolated Delay Line:

	Cycles
STK Module: `DelayL.cpp`	10373
Faust Function: `fdelay` (`music.lib`)	829
Cycles Ratio: 12.5	

Notes: The cycle count per block for the *non-vectorized* STK module was found to be almost the same (10393 cycles), again presumably due to the use of `virtual StkFrames& tick(StkFrames& frames, unsigned int chan)`.

Cubic Nonlinearity Distortion:

	Cycles
STK Module: `Cubicnl.cpp` (new)	7395
Faust Function: `cubicnl` (`effect.lib`)	3387
Cycles Ratio: 2.18	

Note: Nonlinearity parameters were fixed. Updating them once per signal block would add little additional computation.

Conclusions

Based on experiences to date, we conclude:

- If a software module is reasonably easy to code in Faust, then a Faust implementation is preferable over using the STK library. (Of course, there is still no substitute for architecture-dependent, hand-written assembly language when true optimality is required.) For high-level reference implementations, however, Faust generally wins over the STK in terms of performance and in terms of many useful ancillary features. Even STK applications should consider using library replacement modules generated by Faust. A possible drawback for some, when Faust fits the bill, is having to learn the language (⌣̈)

- The Faust white-noise generator was measured to be $17.5\times$ faster than the STK white-noise generator, even after STK ffunction-call overhead was made negligible (inner loop run for 128 samples)
- The Faust biquad (two-pole, two-zero filter) was measured to be $5.8\times$ faster than the STK BiQuad, again with ffunction-call overhead made negligible
- The Faust linearly interpolated delay line was measured to be $12.5\times$ faster than the STK version
- The Faust cubic nonlinearity distortion was measured to be $2.2\times$ faster than the STK version
- Note that the STK was designed with C++ readability and modularity in mind, not minimized execution time. In particular, each STK module originally computed only a single sample in its `tick()` method, and the STK still supports this. Single-sample ticks give the important advantage of introducing only one sample of pipeline delay in a module loop (since module run-order is usually always specified in an STK program). However, when feedback pipeline delay is not at issue, performance can be increased greatly in principle by computing a *block* of samples in the inner-loop of each module, as

is done in CSound, the Music Kit, Pd, and so on. Such "vectorization" enables much better optimization for modern superscalar processor architectures. The relatively recent `tick(frames, chan)` method in the STK appears to provide such performance-enhancing vectorization, but very little performance increase is realized due to its relatively high-level implementation in C++ as a "virtual function".

Related Future Plans

- Further integration of Faust and C++/STK software
- Score-file support for Pd-resident Faust plugins and/or STK-FLEXT modules (e.g., SKINI $\rightarrow$ MIDI $\rightarrow$ Pd)
- New string excitation models
- More instruments

Acknowledgment

Thanks to Yann Orlarey and Perry Cook for helpful feedback based on an earlier draft of this poster.