CHUCKSOUND

A CHUGIN FOR RUNNING CSOUND INSIDE OF CHUCK

Paul Batchelor
thisispaulbatchelor@gmail.com

ChuckSound is a plugin for ChucK (otherwise known as a "chugin") that allows Csound to be run inside of ChucK. Prior to ChuckSound, a typical setup for getting Csound + Chuck working together would be to start ChucK and Csound as separate applications, and to use OSC and/or JACK to communicate. With ChuckSound, Csound is spawned inside of ChucK's audio engine via the Csound API. This approach allows Csound to work seamlessly with ChucK objects without any sort of latency that OSC would produce. ChuckSound has the ability to evaluate Csound orchestra code inside of ChucK as well as send score events.

1 Installation and Setup

The latest version of ChuckSound and installation instructions can be found on github at https://www.github.com/PaulBatchelor/ChuckSound.git.

2 Chuck and Csound

A brief summary of ChucK for Csounders

ChucK is aptly described as a "Strongly-Timed, Concurrent, On-The-Fly Music programming language" [1]. Each of these components makes for a very strong counterpart to Csound.

Firstly, ChucK is *strongly-timed*. Time and timing is a very important feature to ChucK. In fact, time and duration are primitive types in ChucK. [2] Chuck supports many human-readable units of time: samples, milliseconds, seconds, minutes, hours, days, and weeks [3]. The concept of a "control rate" is non-existent in ChucK; most ChucK patches are built up of while loops that pause for an arbitrary period of time using the "time => now" paradigm. Csound users should be encouraged to explore time in Chuck, as the language has a very expressive syntax for this domain.

ChucK has a strong emphasis on *concurrency*, or running processes that occur at the same time. In Chuck, a single program is known as a "shred", and shreds can be "sporked" together to be played simultaneously. While Csound can run simultaneous instrument instances together to achieve things like instrument polyphony, ChucK is able to run multiple files together that are unrelated by running something like "chuck foo.ck bar.ck".

ChucK is designed to write code *on-the-fly*. "On-the-fly" or "live" coding is an important design feature in Chuck. When using ChucK, coding is expected to be part of a performance. Shreds in ChucK can be added and recompiled during a performance without having to stop Chuck from running. While Csound evolved into having real-time capabilities, ChucK has been designed with modern hardware and real-time performance from the beginning. It is still easier to do offline rendering in Csound.

Due to it's resemblance to C-like languages, ChucK could be certainly be classified as a programming language. ChucK supports C/C++ types like floats, ints, and strings. There are also similar control structures in ChucK like for and while loops and if statements. There is support for OOP, with classes, methods, and single inheritance. Writing ChucK code looks and feels like writing a program, whereas Csound looks and feels more like making a patch on a modular synthesizer.

ChucK differs from C-like languages in the way assignment and operators are handled. While C-like languages handle assignment right-to-left, ChucK handles variable assignment left-to-right using the "=>" operator (e.g: "int x = 3" in C would be "3 => int x" in ChucK. For arrays, the "@=>" operator is used (e.g: "[1, 2, 3] @=> int foo"). For audio domain programming, this decision makes sense; more often than not, left-to-right is how signal flow is depicted in diagrams. Nevertheless, this particular syntax can take some adjustment.

Intended Use Cases

ChuckSound is a wrapper for Csound, and while the Csound API is used under the hood, it is not a wrapper for the API. The design of ChuckSound is the author's best attempt to merge the best parts of both languages. Csound in this instance is approached as an event-based signal processor, using a modular synthesizer paradigm for sound and instrument design. ChucK's time granularity and concurrency is used to precisely control Csound events.

3 Usage

Before Csound can run inside of ChucK, ChuckSound must compile a CSD. In order for the CSD to sound properly, it must have the following attributes:

– Realtime audio must be enabled, but any audio drivers should be disabled so that the main audio callback is being handled by ChucK. This can be accomplished with the flags "-onull -+rtaudio=null"

– The buffer size "-b" must match ksmps

– The Csound samplerate matches the samplerate in ChucK (this is typically set system-wide)

– The Csound file is mono (nchan = 1)

While requiring a CSD file is a clumsy implementation in some cases, there are several advantages to this approach. For one, it leverages the several CsOptions flags that can allow for features like sending code over OSC, buffer size tweaks, and MIDI. It is also conceivably easier to integrate existing Csound projects into ChucK for live remixing and performing.

Listed below are several ChuckSound examples, included with this paper.

CSD Player

The simplest usage case is to compile an existing CSD file and to let it run without interruption. Using ChuckSound, this is how it would be accomplished:

File: trapped.ck

```
Csound c => dac;
c.compileCsd("trapped.csd");
283::second => now;
```

Note Launcher

With ChuckSound, one has the ability to send score events. One could leverage ChucK's strong sense of timing and C-like control structures to build very complex sequencers and event generators this way. Also featured in the example below is ChuckSound's ability to evaluate orchestra code on the fly. This is possible thanks to the new improvements to Csound 6 and the Csound 6 API:

File: pluck.ck

```
Csound c => dac;

c.compileCsd("tmp.csd");

"
instr 1
aout = pluck(0.1, p4, p4, 0, 1) * linseg(1, p3, 0)
out aout
endin
"
=> string orc;

c.eval(orc);

string message;
float freq;

while(1) {
    "i1 0 3 " => message;
    Std.rand2(80, 800) => freq;
    freq +=> message;
```

```
    c.inputMessage(message);
    0.5::second => now;
}
```

Evaluating orchestra code inside of ChucK is ideal because it allows multiple ChucK files to use a single template CSD instead of needing to rewriting a new CSD over and over again. The examples from here on will use a single file called “tmp.csd”:

File: tmp.csd

```
<CsoundSynthesizer>
<CsOptions>
;disable audio output and let ChucK handle it all
-d -onull -+rtaudio=null
-b 100
</CsOptions>
<CsInstruments>

sr = 44100
ksmps     = 100
nchnls = 1
0dbfs     = 1

</CsInstruments>
<CsScore>
f 0 $INF
</CsScore>
</CsoundSynthesizer>
```

ChucK audio inside of Csound

ChuckSound is able to process ChucK audio with Csound opcodes. Any audio routed to the Chugin gets sent to an audio-rate channel called “Chuck_Out”. Here in this example a Chuck SawOsc object is being processed by Csound's waveset opcode.

File: waveset.ck

```
SawOsc s => LPF l => Csound c => dac;

c.compileCsd("tmp.csd");
l.set(1000, 0.1);

"
alwayson 2
instr 2
a1 chnget \"Chuck_Out\"
out waveset(a1, 5) * 0.5
endin
"
=> string orc;

c.eval(orc);

float freq;

while(1) {
    Std.rand2(50, 1000) => s.freq;
    500::ms => now;
}
```

Many exciting concepts can arise from this: all of ChucK can be processed through any of Csound's hundreds of opcodes!

Csound across multiple shreds

Much of ChucK's power is leveraged through running and recompiling several shreds. It is not practical to have an instance of Csound on every shred. A better solution would be to utilize public classes and static variables to generate a single instance of Csound that can be accessed across multiple shreds. Such a class could look like this:

File: csEngine.ck

```
public class CSEngine
{
    static Csound @ c;

    fun void compile(string filename)
    {
        c.compileCsd(filename);
    }

    fun void eval(string orc)
    {
        c.eval(orc);
    }

    fun void message(string message)
    {

        c.inputMessage(message);
    }

}

Csound c => Gain g => dac;

CSEngine cs;

c @=> cs.c;
cs.compile("tmp.csd");

/* Avoid clicks */
0 => g.gain;
1::ms => now;
1 => g.gain;

while(1){
    500::ms => now;
}
```

Here is how this class would be used:

File: launcher1.ck

```
CSEngine cs;
```

```
"
instr 1
aout = pluck(0.1, p4, p4, 0, 1) * linseg(1, p3, 0)
out aout
endin
"
=> string orc;

cs.eval(orc);

string message;
float freq;

while(1) {
    "i1 0 3 " => message;
    Std.rand2(80, 300) => freq;
    freq +=> message;
    cs.message(message);
    0.5::second => now;
}
```

Here is another file that can run on another shred:

File: launcher2.ck

```
CSEngine cs;

"
instr 2
aout = moogvcf(vco2(0.1, p4) * linseg(1, p3, 0), 1000, 0.1)
out aout
endin
"
=> string orc;

cs.eval(orc);

string message;
float freq;

while(1) {
    "i2 0 3 " => message;
    Std.rand2(300, 1000) => freq;
    freq +=> message;
    cs.message(message);
    0.9::second => now;
}
```

To see this in action, one could simply run “chuck csEngine.ck launcher1.ck launcher2.ck” from the supplemental file directory. Note that the file “csEngine.ck” must go before “launcher1.ck” and “launcher2.ck” in order to work.

Future Plans

ChuckSound is still very early in development. Current plans for ChuckSound include an easier installation process, better cross-platform support, as well as control-rate and (more) audio-rate channels.

Acknowledgements

Special thanks goes out to Alexander Tape, Ni Cai, and Nick Arner for testing out ChuckSound.

References

[1] "ChucK: Strongly-timed, Concurrent, and On-the-fly Music Programming Language" [Online] Available: http://chuck.cs.princeton.edu/ [Accessed July 30th, 2015].

[2] "Chuck: Language Specification" [Online] [http://chuck.cs.princeton.edu/doc/language/] [Accessed July 30th, 2015].

[3] Floss Manual, "Chapter 21: Time and timing" [Online] [http://en.flossmanuals.net/chuck/ch021_time-and-timing/] [Accessed July 30th, 2015].