CHUGENS, CHUBGRAPHS, CHUGINS: 3 TIERS FOR EXTENDING CHUCK

Spencer Salazar *Ge Wang*

Center for Computer Research in Music and Acoustics
Stanford University
{spencer, ge}@stanford.edu

ABSTRACT

The ChucK programming language lacks straightforward mechanisms for extension beyond its built-in programming and processing facilities. Chugens address this issue by allowing programmers to implement new unit generators in ChucK code in real-time. Chubgraphs also allow new unit generators to be built in ChucK, by defining specific arrangements of existing unit generators. ChuGins allow a wide array of high-performance unit generators and general functionality to be exposed in ChucK by providing a dynamic binding between ChucK and native C/C++-based compiled code. Performance and code analysis shows that the most suitable approach for extending ChucK is situation-dependent.

1. INTRODUCTION

Since its introduction, the ChucK programming language [11] has become a popular tool for computer music composers, educators, and application software developers. However, to date, its catalogue of audio processing unit generators and general programming functionality has been largely limited to those that are built-in when the ChucK binary executable is compiled. Adding new unit generators mandates recompilation of the entirety of ChucK, requiring a level of expertise and motivation reserved for an elite group of power-users. Furthermore, community-based development efforts are hampered by this centralization of functionality, as developers of new unit generators have no easy way to share their work.

The aim of the work described herein is to provide ChucK with multiple levels of extensibility, each essential and appropriate to specific tasks and levels of user expertise. On the one hand, ChucK's pervasive ethos of on-the-fly development creates the desire to design and implement new audio processors in ChucK itself in real-time, working down to the per-sample level if necessary. Furthermore, implementing these components in ChucK allows their use on any operating system ChucK supports with no additional effort from the developer. For these cases we have developed Chugens and Chubgraphs.

On the other hand, real-time performance requirements often mandate the use of compiled native machine code for complex audio-rate processing. There also exists a wealth of C/C++-based software libraries for audio synthesis and effects, such as FluidSynth [3] and Faust [7]. These situations can be straightforwardly handled given portable bindings between ChucK and native compiled code, which is precisely the intent of ChuGins.

2. RELATED WORK

Extensibility is a primary concern of music software of all varieties. The popular audio programming environments Max/MSP [12], Pure Data [8], SuperCollider [6], and Csound [4] all provide mechanisms for developing C/C++-based compiled sound processing functions. Max also allows control-rate functionality to be encapsulated in-situ in the form of Javascript code snippets. Csound allows the execution of Tcl, Lua, and Python code for control-rate and/or audio-rate manipulation and synthesis. A thriving ecosystem revolves around extensions to popular digital audio workstation software, in the form of VST, RTAS, AudioUnits, and LADSPA plugins developed primarily in C and C++. In general-purpose computing enivornments, JNI provides a highly flexible binding between native machine code and the Java virtual machine-based run-time environment [5]. RubyGems is a complete plugin and package management tool for both Ruby-based plugins and C/C++ libraries compiled with Ruby bindings [9].

3. CHUGENS, CHUBGRAPHS, CHUGINS

3.1. Chugens

The goal of Chugens (pronounced "chyoo-jen") is to facilitate rapid prototyping of audio synthesis and processing algorithms. Additionally, Chugens provide a basic framework for extending ChucK's built-in audio processing functionality. Using the Chugen system, a programmer can implement sample-rate audio algorithms within the ChucK development environment, utilizing the full array of programming facilities provided by ChucK. These processing units can be naturally integrated into standard ChucK programs, even in the same script file, providing seamless control of audio-rate processing, control-rate manipulation, and higher-level compositional organization.

A Chugen is created first by subclassing the built-in `Chugen` class. This subclass is required to implement a `tick` function, which accepts a single floating-point argument (the input sample) and returns a single floating-point value (the output sample). For example, this code uses a Chugen to synthesize a sinusoid using the cosine function:

```
class MyCosine extends Chugen
{
   0 => int p;
   440 => float f;
   second/samp => float SRATE;
   float tick(float in)
   {
       return Math.cos(p++*2*pi*f/SRATE);
```

```
    }
}
```

In the case of an audio synthesizer that does not process an existing signal, the input sample may be ignored.

A Chugen defined so may be integrated into audio graphs like any standard ChucK ugen. Since the `tick` function is just a standard ChucK class member function, it can be as simple or as elaborate as required. Standard library calls, file I/O, multiprocessing (using `spork`), and other general ChucK programming structures can be integrated into the `tick` function and supporting code. For performance reasons, its important to consider that the `tick` function will be called for every sample of audio, so simple `tick` functions will typically perform better. Moreover, the intrinsic overhead of ChucK's virtual machine architecture will cause Chugens to underperform compared to a native C/C++ implementation. Lastly, since Chugens are fundamentally a specialization of a ChucK class, it may define functions to provide structured access to whichever parameters it wishes to expose to the programmer.

3.2. Chubgraphs

Chubgraphs (pronounced "chub-graph") provide a way to construct new unit generators by composition, arranging multiple existing ugens into a single unit. In this way, common arrangements of existing unit generators can be defined and instantiated. Furthermore, Chubgraph parameters can be exposed in a structured manner via class member functions.

A Chubgraph is defined by extending the `Chubgraph` class. The `Chubgraph` class has member variables named `inlet` and `outlet`; `inlet` is a ugen that represents the input signal to the Chubgraph, and `outlet` is the output signal. The Chubgraph's internal audio processing graph is created by spanning a sequence of ugens between `inlet` and `outlet`. The following Chubgraph implements a basic feedback echo processor:

```
class Feedback extends Chubgraph
{
    inlet => Gain dry => outlet;
    dry => Delay delay => outlet;

    delay => Gain feedback => delay;

    0.8 => feedback.gain;
    1::second => delay.delay;
}
```

(Chubgraphs that do not wish to process an input signal, such as audio synthesizing algorithms, may omit the connection from `inlet`.)

Compared to Chugens, Chubgraphs have obvious performance advantages, as primary audio-rate processing still occurs in the native machine code underlying its component ugens. However Chubgraphs are limited to audio algorithms that can be expressed as combinations of existing unit generators; implementing, for example, intricate mathematical formulae or conditional logic in the form of a ugen graph is possible but, in our experience, fraught with hazard.

3.3. ChuGins

ChuGins (pronounced "chug-in") allow near limitless possibilities for expansion of ChucK's capabilities. A ChuGin is a distributable dynamic library, typically written in C or C++ compiled to native machine code, which ChucK can be instructed to load at runtime. When loaded, the ChuGin defines one or more classes that are subsequently available to ChucK programs. These classes may define new unit generators or provide general programmatic functionality beyond that built in to ChucK. Since these classes are normal ChucK classes implemented with native code, member functions and variables can be used to provide an interface to control parameters.

ChuGins are best suited for audio algorithms that are reasonably well understood and stand to gain from the performance of compiled machine code. The "write-compile-run" development cycle and C/C++-based programming mandated by ChuGins make implementation of audio processors require comparatively more effort than the Chubgraph or Chugen approaches. However for ugens a programmer intends to use over an extended period of time, the effort to implement a ChuGin will quickly pay off in the form of lower CPU usage.

An additional advantage of ChuGins is that they may provide functionality far outside the intrinsic capabilities of ChucK. Complex synthesis C/C++ based synthesis packages can be imported wholesale into ChucK, opening up an abundance of sonic possibilities. For example, ChuGins have been implemented to bring audio processing programs from the Faust programming language into ChucK. Similarly, the SoundFont renderer FluidSynth has been packaged as a ChuGin. This functionality is not limited to audio processing; a serial port input/output ChuGin is under development, as are other general purpose programming libraries.

Development of a ChuGin is somewhat more complex than Chubgraphs or Chugens, and does not lend itself to explicit presentation of code herein. Using a set of convenience macros, a ChuGin developer first defines a `query` function, which ChucK calls upon first loading the ChuGin. ChucK provides the `query` function with routines with which to define unit generators and classes, specify what member variables and functions are associated with these, and indicate a `tick` function in the case of unit generators. These are then defined in C/C++, using predefined macros for interactions with the upper-level ChucK runtime, such as retrieving function arguments, getting and setting member variables, and handling input/output samples. This code is then compiled into a dynamic library using the standard facilities for doing so on the target computing platform (`gcc` for Mac OS X and Linux systems, Visual C++ for Windows systems). Additional C/C++ code and parts of or entire libraries, such as STK, may be compiled into the dynamic library using the mechanisms standard for those operations on the target platform.

3.4. FaucK

Faust is a functional programming language designed for real-time signal processing and synthesis. Using a concise block-diagram based specification language, Faust can produce audio signal processors and synthesizers for a wide variety of target backends, such as Pure Data, Max/MSP, SuperCollider, VST, and others. The ease with which one can implement intricate

signal processing routines in Faust has led to a plethora of high-quality audio effects and synthesizers implemented using it.

A bridge between Faust and ChucK would both expand the sonic palette of ChucK and motivate ChucK users to develop new unit generators in Faust, enriching the software ecosystems of both. FaucK seeks to provide such a bridge, in the form of a Faust architecture file and an accompanying compilation script. Executing this script with the name of a Faust source file as an argument will produce a ChuGin implementing the processor defined in the Faust file. Using these utilities, a variety of existing algorithm designs can be readily introduced into the ChucK unit generator library.

4. PERFORMANCE CASE STUDIES

Execution speed of audio software is typically a vital metric for computer musicians, as real-time audio synthesis requires timely production of tens of thousands of samples per second. To evaluate the performance of our mechanisms for extending ChucK, we designed and implemented several reference unit generators using each method. For each method, we measured the time required to offline-render 5 minutes of audio using the resulting unit generator.

In many situations not only is CPU time at a premium, but a programmer's time to implement a specified application is also limited. Therefore, often "less complex" programs are more desirable than "more complex" programs because they can be developed faster and be better understood by other programmers. While quantifying code complexity is a nuanced and volatile field of inquiry, we have chosen to represent it in the form of lines of code required to implement the unit generator using each method. In addition to characterizing code complexity, lines of code gives some approximation of how long the extension may have taken to develop.

Finally, to construct an overall picture of performance vs. development effort, we have created a composite statistic formed by the product of these two metrics, named "aptness." A lower aptness indicates that a particular technique is more appropriate for a particular processing application than a technique with a higher aptness.

4.1. CombFilter

For this case study we implemented a basic recursive feedback with moving average comb filter (the "plucked string filter" described in [10]). For reference, the comparatively concise Chubgraph version is as follows:

```
public class CombFilter extends Chubgraph
{
    inlet => Delay d => outlet;
    d => OneZero oz => d;

    -1 => oz.zero;
    100::samp => d.delay;
    0.75 => oz.gain;
}
```

As can be seen in Table 1, the ChuGin variant of CombFilter performs best, but the Chubgraph version is not far behind. The Chugen version performs poorly as a result of requiring execution of ChucK code at audio-rate, and also requires more code than the Chubgraph, as the delay line and moving average filter needed to be implemented from scratch. Given the vastly superior lines of code and aptness metrics, the Chubgraph is probably the best approach to use in this case. Chubgraph excels here because CombFilter is essentially a particular arrangement of existing unit generator primitives, i.e. Delay and OneZero, allowing a concise and efficient implementation. A ChuGin may be more appropriate if speed is the utmost concern, but its inferior code complexity makes it largely undesirable in this class of audio processing algorithm.

CombFilter	Time (s)	Lines of Code	Aptness
Chugen	10.944	18	197
Chubgraph	2.841	9	26
ChuGin	2.001	65	130

Table 1. Performance and complexity measurements for CombFilter.

4.2. Bitcrusher

This audio processor performs sample rate reduction by decimation and destructive sample-width compression to recreate its input at a specific sample rate and sample width. The result typically sounds as if it was synthesized by vintage, low-resolution audio or video game hardware such as the Casiotone family of consumer keyboards or the Nintendo Entertainment System. For reference, the shortest version, using Chugen, is listed here:

```
public class Bitcrusher extends Chugen
{
    8 => int bits;
    4 => int ds;
    float sample;
    int count;
    Math.pow(2,31) $ int => int INT_MAX;
    INT_MAX $ float => float fINT_MAX;

    fun float tick(float in)
    {
        if(count++ % ds == 0)
          Math.min(1,Math.max(-1,in))=> sample;

        (sample * fINT_MAX) $ int => int q32;
        32-bits => int shift;
        ((q32 >> shift) << shift) => q32;

        return q32 / fINT_MAX;
    }
}
```

The results in Table 2 once again show the ChuGin as the performance leader. Chugen is the most concise, and, unsurprisingly, the poorest performing contender. Chubgraph performs better than Chugen, but much worse than ChuGin, and is far more complex in this case. The built-in unit generators at the disposal of Chubgraphs are ill suited to tasks involving conditional logic and intricate non-linear arithmetic manipulation. (Subjectively, the Chubgraph implementation is so obtuse that a cursory perusal of its code would likely not reveal its function except to an expert ChucK programmer.) Therefore, this algorithm would be ideal to develop and prototype as a Chugen. If a computer musician finds him or herself using this Chugen often, or requires faster execution time, reimplementation as a ChuGin would be appropriate.

Bitcrusher	Time (s)	Lines of Code	Aptness
Chugen	10.600	21	223
Chubgraph	6.544	41	268
ChuGin	1.965	84	165

Table 2. Performance and complexity measurements for Bitcrusher.

5. FUTURE WORK

Further developments we are pursuing include more versatile mechanisms for creating general purpose programming libraries in ChucK. Additionally we are investigating a unified ChuGin repository and distribution system, similar to Debian APT or RubyGems, in order to simplify the task of finding and installing extensions to ChucK.

Our current system of importing ChuGins does not scale well, as ChuGins cannot be loaded on demand. Rather, ChucK will load every ChuGin that is installed on a system, which may take a noticeable amount of time if there are many ChuGins. In the future we wish to more intelligently load ChuGins only when their component ugens and classes are invoked by active ChucK code.

Additional improvements may be required in the form of process and memory safety in ChuGins. Currently, ChuGins execute in the same process space as ChucK itself, which means that buggy ChuGins can crash ChucK outright. Technological approaches can alleviate these problems somewhat, but organizational solutions may also be required, such as having registries of vetted and thoroughly tested ChuGins.

6. ACKNOWLEDGEMENTS

Special thanks to Kassen Oud, Casper Schipper, Jorge Herrera, and Hongchan Choi for their invaluable beta testing, bug finding, and feature requesting. Additional support for this research was made possible by a National Science Foundation Creative IT Grant, No. IIS-0855758.

7. REFERENCES

[1] Apt. http://wiki.debian.org/Apt. Accessed February 10, 2012.

[2] Cook, P., and Scavone, G. 1999. "STK: The Synthesis Toolkit." In *Proceedings of the International Computer Music Conference*. Beijing, China.

[3] FluidSynth. http://www.fluidsynth.org/. Accessed February 11, 2012.

[4] Lazzarini, V. 2005. "Extensions to the Csound Language: from User- Defined to Plugin-Opcodes and Beyond." In *Proceedings of the Linux Audio Conference*. Karlsruhe, Germany.

[5] Liang, S. 1996. *The Java Native Interface: Programmers Guide and Specification*. Addison-Wesley, Reading, MA.

[6] McCartney, J. 2002. "Rethinking the Computer Music Language: SuperCollider," *Computer Music Journal*, 26(4): 61-68.

[7] Orlarey, Y., Fober, D., and Letz, S. 2009. "Faust: an Efficient Functional Approach to DSP Programming," *New Computationals Paradigms for Computer Music*. Edition Delatour, France.

[8] Puckette, M. S. 1997. "Pure data." In *Proceedings of the International Computer Music Conference*. Thessaloniki, Greece.

[9] RubyGems. http://rubygems.org/. Accessed February 10, 2012.

[10] Steiglitz, K. 1996. *Digital Signal Processing Primer*. Addison-Wesley, Reading, MA.

[11] Wang, G. 2008. *The ChucK Audio Programming Language: A Strongly-timed, On-the-fly Environ/mentality*. PhD Thesis. Princeton University.

[12] Zicarelli, D. 1998. "An Extensible Real-Time Signal Processing Environment for MAX." In *Proceedings of the International Computer Music Conference*. Ann Arbor, MI.