SUPPORT FOR MIR PROTOTYPING AND REAL-TIME APPLICATIONS IN THE CHUCK PROGRAMMING LANGUAGE

Rebecca Fiebrink
Princeton University
fiebrink@princeton.edu

Ge Wang
Stanford University
ge@ccrma.stanford.edu

Perry Cook
Princeton University
prc@cs.princeton.edu

ABSTRACT

In this paper, we discuss our recent additions of audio analysis and machine learning infrastructure to the ChucK music programming language, wherein we provide a complementary system prototyping framework for MIR researchers and lower the barriers to applying many MIR algorithms in live music performance. The new language capabilities preserve ChucK's breadth of control—from high-level control using building block components to sample-level manipulation—and on-the-fly re-programmability, allowing the programmer to experiment with new features, signal processing techniques, and learning algorithms with ease and flexibility. Furthermore, our additions integrate tightly with ChucK's synthesis system, allowing the programmer to apply the results of analysis and learning to drive real-time music creation and interaction within a single framework. In this paper, we motivate and describe our recent additions to the language, outline a ChucK-based approach to rapid MIR prototyping, present three case studies in which we have applied ChucK to audio analysis and MIR tasks, and introduce our new toolkit to facilitate experimentation with analysis and learning in the language.

1. INTRODUCTION

ChucK [13] began as a high-level programming language for music and sound synthesis, whose design goals included offering the musician user a wide breadth of programmable control—from the structural level down to the sample level—using a clear and concise syntax, and employing a set of abstractions and built-in objects to facilitate rapid prototyping and live coding. We have recently expanded the language to provide support for audio analysis and machine learning, with two primary goals: first, to offer real-time and on-the-fly analysis and learning capabilities to computer music composers and performers; and second, to offer music information retrieval (MIR) researchers a new platform for rapid prototyping and for easily porting algorithms to a real-time performance context. We address the former goal in [6]; here, we focus on the latter.

We begin in Section 2 by reviewing prototyping in MIR and motivating the need for additional shared tools between MIR and performance. We describe the ChucK language as it is used for music creation, including prototyping and live coding systems. In Section 3, we describe in some detail how we have incorporated analysis and learning into the language, with attention both to preserving the flexible and powerful control that makes ChucK suited for prototyping and experimentation, and to tightly and naturally integrating the new functionality with ChucK's synthesis framework. Sections 4 and 5 illustrate the new potential of ChucK as an MIR rapid prototyping workbench, introducing an example working pipeline for prototyping an MIR task and presenting three examples of how we have used ChucK to perform and teach music analysis. Finally, in Section 6 we discuss ongoing work to improve ChucK as an MIR tool and announce the release of a toolkit containing examples and supporting code for MIR researchers desiring to experiment with the language.

2. BACKGROUND AND MOTIVATION

2.1. MIR Prototyping

One primary goal in this work is to provide a new rapid prototyping environment for MIR research. Our definition of an MIR prototyping environment includes: the ability to design new signal processing algorithms, audio feature extractors, and learning algorithms; the ability to apply new and existing signal processing, feature extraction, and learning algorithms in new ways; and the ability to do these tasks quickly by taking advantage of high-level building blocks for common tasks, and by controlling the system either via a GUI or through concise and clear code. Furthermore, a prototyping system should encourage experimentation and exploration. There exist several programming environments for MIR that accommodate many of the above requirements, including Matlab, M2K[1], MARSYAS [12], CLAM [1], and Weka [15], and their popularity suggests that they meet the needs of many MIR research tasks. We propose that ChucK also meets all of these requirements and inhabits a complementary place in the palette of tools at the MIR researcher's disposal. In particular, ChucK features real-time support for analysis and synthesis, easy accommodation of concurrency and auditory feedback, and a syntax and running environment

[1] http://www.music-ir.org/evaluation/m2k/index.html

that encourage a development cycle rapid enough for live-coding, in which the performer modifies the code in a live performance [14]. As we discuss below, ChucK is therefore able to facilitate a rapid, feedback-driven, real-time and performance-enabled approach to MIR prototyping that is not a primary concern of other tools.

2.2. MIR and Music Performance

Research in MIR has primarily focused on analyzing and understanding recorded audio, static symbolic representations (e.g., MIDI or Humdrum files) and other non-performative musical representations and metadata. There is some exceptional work, notably real-time score-following and accompaniment systems such as [8], [11]. However, many other popular music information retrieval tasks, such as mood and style analysis and instrumentation and harmony identification, are directly pertinent to real-time interactive performance, and the relevance of core MIR work to live music remains under-exploited.

For one thing, a primary focus of MIR is building computer systems that understand musical audio at a semantic level (e.g., genre, rhythm, mood, harmony), so that humans can search through, retrieve, visualize, and otherwise interact with musical data in a meaningful way. Making sense of audio data at this higher level is also essential to *machine musicianship*, wherein the performing computer—like any musically trained human collaborator—is charged with interacting with other performers in musically appropriate ways [10].

Despite the shared need to extract useful features from audio, and to bridge the gap between low-level audio features and higher-level musical properties, there does not exist a shared tool set for accomplishing these tasks in both computer music and MIR. On one hand, most computer music languages do not readily accommodate analysis of audio *in the language*; for example, highly custom spectral analysis and processing tasks must be coded in C++ externals in order to be used in SuperCollider or Max/MSP. Externals' development overhead, enslavement to the audio and control rates and APIs of the associated music language, and unsuitability as recursive building blocks within larger analysis algorithms make them an unattractive choice for MIR researchers (not to mention musicians). On the other hand, most MIR toolkits and frameworks do not support real-time audio processing or synthesis. Those that do (namely MARSYAS [3] and CLAM [1]) do not offer the full-fledged synthesis and interaction support of a computer music *language*, so while popular among MIR researchers, they do not have widespread adoption among computer musicians.

Computer musicians would undoubtedly benefit from lower barriers to adapting state-of-the-art MIR algorithms for pitch tracking, beat tracking, etc. for their real-time performance needs. We additionally posit that MIR researchers can benefit from increased collaboration with musicians and composers, which does not suffer from the common challenges of copyright restrictions on obtaining data or releasing systems to the public, nor the difficulty and expense in obtaining reasonable ground truth to perform evaluations. Additionally, applying MIR research in music performance can offer researchers outside of industry a greater potential to directly impact people's experiences with music.

We realize that no single tool will meet the needs of everyone in computer music and MIR, but we propose to lower barriers to working at the intersection of these fields—in analysis-driven computer music and real-time, potentially performance-oriented MIR—via a full-fledged computer music language that allows for arbitrarily complex signal processing and analysis tasks within a user-extensible, object oriented framework.

2.3. ChucK

ChucK is a cross-platform, open-source computer music programming language [13] whose primary design goals include the precise programmability of time and concurrency, with an emphasis on encouraging concise, readable code. System throughput for real-time audio is an important consideration, but first and foremost, the language was designed to provide maximal control and flexibility for the audio programmer. In particular, key properties of the language are as follows:

- *Flexibility*: The programmer may specify both high-level (e.g., patching an oscillator to a filter, or initiating structural musical events) and low-level, time-based operations (e.g., inspecting and transforming individual samples) in a single unified language mechanism, without any need for externals.
- *Readability*: The language provides a strong correspondence between code structure, time, and audio building blocks; as a result, the language is increasingly being used a teaching tool in computer music programs, including at Princeton, Stanford, Georgia Tech, and CalArts.
- *Modularity*: Like many other languages, ChucK encapsulates behaviors of synthesis building blocks (filters, oscillators, etc.) using the "Unit Generator" model. Its object-oriented nature also supports modular user code.
- *A "do-it-yourself" approach*: By combining the ease of high-level computer music environments with the expressiveness of lower-level languages, ChucK supports high-level musical representations, as well as the prototyping and implementation of low-level, "white-box" signal-processing elements in the same language.
- *Explicit treatment of time*: There is no fixed control rate. It's possible to assert control on any unit generator at any point in time, and at any rate, in a sub-sample-precise manner.

- *Concurrency*: Parallel processes (called shreds) share a notion of time, so one can precisely synchronize and easily reason about parallel code modules according to each's treatment of time. Parallelism is easy and expressive.
- *On-the-fly programming*: Programs can be edited as they run; this functionality is supported and encouraged in the miniAudicle development environment[1].

The following example synthesis code illustrates several of these concepts. In it, an impulse train is synthesized, and its frequency is modified in a concurrent parallel function at an arbitrary "control rate."

```
// patch impulse generator to output
Impulse i => dac;

50::samp => dur impulsePeriod; // start at 50 samples
spork ~ sweepPeriod(); // run function in parallel

// generate an impulse every period
while( true ) {
    1.0 => i.next;        // fire impulse
    impulsePeriod => now; // advance time
}

// sweep the period from 50 to 200 samples
// updating every .1 second
fun void sweepPeriod() {
    while( impulsePeriod < 200::samp ) {
        impulsePeriod * 1.01 => impulsePeriod;
        .1::second => now;
    }
}
```

Figure 1: Simple concurrent ChucK code. Note that '… => now' acts to control program execution in time, and '=>' alone acts as a left-to-right assignment operator.

3. ADDITIONS TO CHUCK

3.1. Unit Analyzers

In previous work, we introduced a language-based solution to combining audio analysis and synthesis in the same high-level programming environment of ChucK [15]. The new analysis framework inherited the same sample-synchronous precision and clarity of the existing synthesis framework, while adding analysis-specific mechanisms where appropriate. The solution consisted of three key components. First, we introduced the notion of a Unit Analyzer (UAna), similar to its synthesis counterpart, the Unit Generator (UGen), but augmented with a set of operations and semantics tailored towards analysis. Next, we introduced an augmented dataflow model to express dependencies among unit analyzers (UAnae), and to manage caching and computation accordingly. Third, we ensured that the analysis framework made use of the existing timing, concurrency, and on-the-fly programming mechanisms in ChucK as a way to precisely control analysis processes.

```
// set up analysis network for mic input
adc => FFT fft =^ Centroid c => blackhole;
fft =^ RollOff r;

// (would set FFT window, size, etc. here)
// ...

// launch in parallel
spork ~ computeRolloff();

// compute centroid every 512 samples
while( true ) {
    c.upchuck();
    512::samp => now;
}

fun void computeRolloff() {
    // compute rolloff every 1 second
    while( true ) {
        r.upchuck();
        1::second => now;
    }
}
```

Figure 2: An example analysis patch

For example, it's possible to instantiate an FFT object for spectral analysis, instantiate spectral centroid and spectral rolloff objects that compute using the output of the FFT, and trigger the extraction of each feature at its own rate. Code for this task appears in Figure 2. The programmer has dynamic, sample-precise control over analysis parameters such as FFT/IFFT sizes, analysis windows, hop sizes, feature extraction rates, etc. The primary advantages of using ChucK for analysis are threefold:

- *Conciseness*: ChucK implicitly manages real-time audio and buffering, and the language is tailored for audio, so analysis system implementation time and code length are greatly reduced.
- *Rapid turnaround experimentation*: Through the application of on-the-fly programming and ChucK's concise audio programming syntax, one can quickly prototype systems and sub-systems, changing parameters as well as the underlying system structure, and experience the results almost immediately.
- *Concurrency*: Parallel management of feature extraction, even at multiple rates, is straightforward in ChucK. This sample-synchronous concurrency would be extremely challenging and/or inefficient in C++, Java, or a library built in these languages, due to their support for preemptive, thread-based concurrency, and the consequent need to contend with thread-instantiation, synchronization, bookkeeping, etc.

[1] http://audicle.cs.princeton.edu/mini/

These highly useful flexibilities come with a tradeoff: system performance and throughput. A system implemented in reasonable ChucK code would likely run much less efficiently than an optimized C++ implementation. In this regard, it may be desirable to leverage the flexibility and rapid experimentation abilities of ChucK to prototype systems and components, and if needed, then implement "production" code in a low-level language with an optimizing compiler. But for researchers experimenting with new MIR algorithms, such a prototyping stage can be instrumental in crafting new systems and testing the feasibility of new ideas.

3.2. Learning Framework

A natural consequence of ChucK's analysis capabilities is that analysis results can be treated as *features* whose relationship to high-level musical concepts can be learned via labeled examples and standard classification algorithms. Classification is a common and powerful technique in MIR, and it has been applied successfully to MIR problems such as genre [2], mood [7], and transcription [5]. Classification has also been widely used in computer music for tasks such as pitch tracking [9], and learning is a key component of the aforementioned accompaniment and score-following systems. ChucK's learning framework was designed with the recognition that a general tool for learning could be useful for accomplishing both MIR analysis tasks and creative compositional tasks, and with acknowledgment that classification of streaming audio is a task that should be natural to perform in real time.

We have built an object-oriented classification framework in ChucK, modeled on the architecture and naming conventions of Weka [16], a popular Java framework for applied machine learning. Classification algorithms such as k-nearest-neighbor and AdaBoost are implemented as ChucK classes which inherit from a parent Classifier class. Each Classifier can be trained on a dataset, represented by an Instances class, and trained Classifiers can be used to assign class predictions to new Instance objects. Because this infrastructure is all written in ChucK, users can easily add their own Classifier child classes in Chuck, as well as modify not just classification parameters but also the underlying algorithms on-the-fly. We have also implemented a suite of standard MIR features, listed in Table 1.

Note that the implementation of ChucK's analysis and learning capabilities preserves its suitability for rapid prototyping, and extends this capability to many MIR tasks. In particular, the user retains sample-level control over audio processing, allowing easy implementation of new feature extraction methods and signal processing algorithms directly in the language. Learning algorithms are also implemented in the language, affording the user arbitrary modification of these algorithms at any time, without the need to recompile. Additionally, the user can choose from standard MIR features and standard classifiers that have already been implemented, and use these out-of-the-box. Furthermore, the ability to code features and algorithms precisely in ChucK, within its object-oriented framework, means that the user can create new "white-box" features and classifiers for immediate usage. In short, ChucK meets the requirements we believe to be important to a prototyping environment.

FFT, IFFT, DCT, IDCT	LPC coefficients
Spectral centroid, spread, rolloff, and flux	Zero crossing rate
	RMS
Mel- and real-frequency cepstral coefficients	Cross- and auto-correlation

Table 1: MIR features in ChucK

4. AN MIR PROTOTYPING PIPELINE

Here we provide an example of how an MIR student or researcher might approach working with the MIR prototyping system of ChucK. Perhaps someone has an idea for an audio analysis task. One can use ChucK to write relevant code immediately: in particular, the programmer can 1) quickly instantiate and connect together essential unit analyzers (FFT's, feature extractors, etc.), then 2) specify any initial parameters, and 3) write control code, potentially in concurrent chuck processes (shreds). If a learning algorithm is part of the task, the programmer will additionally instantiate a classifier and set its parameters, and include control code for passing the extracted feature vectors to the classifier during training and testing stages. The programmer can ignore issues of audio buffering, dependencies among analysis results (e.g., spectral centroid requires an FFT), concurrency management, etc., as these are handled by ChucK, and instead focus on the content of the algorithm.

As soon as a basic skeleton for the algorithm is written, the programmer can immediately run the code and observe the output (which can easily be sonified, if desired). At this point, the programmer can augment or tweak the parameters *and* underlying structures of the system via on-the-fly programming. For example, the programmer can easily change STFT window type, size, zero-padding, and hop size, or add and remove the features used in classification, and immediately observe the consequences of these changes. It is in this focal stage of prototyping, wherein the programmer interactively tunes system performance and logic, that ChucK is most uniquely powerful as a prototyping tool, aided by its live-coding support in miniAudicle.

5. EXAMPLES

We now present three examples of music analysis systems that have been built in ChucK. The first is a trained vowel

identification system with a GUI interface, which illustrates the conciseness of interactive ChucK code and ease with which GUIs can be incorporated. The second is an implementation of a genre and artist classification system described in [2], which illustrates the ease of porting a complex MIR feature extraction and classification system to ChucK and applying it to real-time audio input. The third is a discussion of our experiences employing ChucK's MIR capabilities as an educational tool, illustrating its usefulness to MIR and computer music novices. The code for case studies 1 and 2 is publicly available (see Section 6).

5.1. Simple Visualizations and Classifier

In order to visualize spectral properties of a sound in real-time, a filter-based sub-band spectral analyzer was implemented in ChucK using MAUI, miniAudicle's framework for graphical widgets. The GUI appears in Figure 3(a); sliders update to indicate the power in each octave sub-band in real-time. The entire code for this task is 25 lines, 15 of which manage the display.

With a simple modification, this code and display can be used to train a classifier on any task for which sub-band power is a useful feature, then perform classification (and useful visual feedback) on new inputs. For example, code that uses 1/3-octave sub-bands from 200Hz to 4kHz to train a nearest-neighbor classifier to recognize each of 5 vowels spoken by the user (and silence), to classify the vowel of new inputs, and to construct the training and feedback graphical interface shown in Figure 3(b), requires under 100 lines of code. This includes the nearest-neighbor classifier coded from scratch (i.e., the KNN ChucK class was not used here). Clearly, ChucK allows for concise specification of signal processing analysis tasks and for real-time GUI display and input.

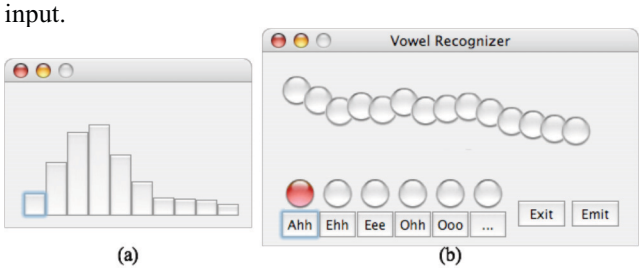

Figure 3: GUIs for (a) octave-band spectral analysis and (b) trained vowel recognition

5.2. Artist Classification

Of obvious concern to MIR researchers interested in ChucK is whether more complex systems are possible. To address this, we have implemented one of the genre and artist classification systems of Bergstra et al. described in [2]. It uses eight types of standard audio features, including FFT coefficients, real cepstral coefficients, mel-frequency cepstral coefficients, zero-crossing rate, spectral spread, spectral centroid, spectral rolloff, and LPC coefficients (all are available in ChucK). Means and variances are computed for each feature over a segment of consecutive frames, then classified by an AdaBoost classifier using decision stumps as the weak learners. Bergstra's system performed in the top two submissions for both genre and artist classification at MIREX 2005 [4].

We have embedded this classification approach in a simple interface for real-time, on-the-fly training and classification of audio. The user can specify via a keyboard or graphical interface that the incoming audio provides examples of a particular class (thereby initiating the construction of labeled Instances from features extracted from this audio), initiate the training rounds of AdaBoost using all available Instances, and then direct the trained classifier to classify the incoming audio and output the class predictions to the screen.

Our experimental on-the-fly application of this system reveals the impressive strength of the features and classifier; in fact, training on only 6 seconds of audio from each of two artists, the system is able to classify new songs' artists with over 80% accuracy. Furthermore, it is easy to use the keyboard interface to experiment with applying this system to new tasks, for example speaker identification and instrument identification, without modification to the algorithm or features.

5.3. ChucK as Introductory MIR Teaching Tool

We have used ChucK to teach audio analysis and basic MIR in introductory computer music courses at Princeton and Stanford. By leveraging the clarity and flexibility of representing analysis and MIR concepts in the language, as well as ChucK's rapid prototyping and on-the-fly programming capabilities, we were able to teach the following topics to students familiar with the basics of the language:

- Practical real-time, short-time Fourier analysis on audio signals (both recorded and live), with on-the-fly demonstration of the effects of window type, window size, FFT size, and hop size
- Standard spectral- and time-domain audio features, their extraction algorithms, and real-time exploration of relationships between audio signals and feature values
- Classification, including applications to speaker identification, vowel/consonant analysis, and physical gesture recognition
- Basic pitch and beat tracking using spectral processing

Additionally, at Stanford, these concepts were taught in conjunction with a large class project to design a computer-mediated performance. Although the use of analysis and MIR algorithms was not a project requirement, more than one-third of the class integrated the extraction of some high-level musical information as a

component of their final projects. These ranged from real-time algorithmic processes that employed extracted features to create compelling accompaniments to a live flutist, to real-time speech analysis that was transformed into gestures controlling a Disklavier, to amplitude and pitch event-triggered generative "sonic clouds." While these projects used very basic MIR components, it was encouraging to witness the students (most of whom had not worked with MIR or audio analysis before) eagerly and efficiently experiment in a 2–3 week period and craft successful musical performances from these investigations.

6. ONGOING WORK AND CONCLUSIONS

We have made available the ChucK learning infrastructure described in Section 3.2, code for the first and second examples described in Section 5, as well as code for several other example tasks, as part of the Small Music Information Retrieval toolKit (SMIRK)[1]. SMIRK will provide a permanent and growing open-source repository for music information retrieval infrastructure and examples using ChucK, targeted at researchers, educators, and students.

Several improvements to ChucK's MIR capabilities are currently underway. First, support for asynchronous file I/O will allow the reading and writing of large datasets in a way that does not interfere with audio production. Second, we are adding more classifiers and support for modeling, beginning with hidden Markov models.

Our recent additions to ChucK have greatly expanded its capabilities beyond the realm of sound synthesis and music performance. Through the integration of analysis and learning tools, we hope to lower the barriers to applying MIR algorithms in real-time settings and suggest a new prototyping paradigm for signal processing, feature extraction, and learning components of MIR systems. We are excited by the nascent potential for new creative and technical work by researchers, musicians, educators, and students using this new framework.

7. ACKNOWLEDGEMENTS

We thank our students for bravely and creatively experimenting with ChucK. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

8. REFERENCES

[1] Amatriain, X., P. Arumi, and D. Garcia, "CLAM: A framework for efficient and rapid development of cross-platform audio applications," *Proceedings of ACM Multimedia*, Santa Barbara, CA, 2006.

[2] Bergstra, J., N. Casagrande, D. Erhan, D. Eck, and B. Kégl, "Aggregate features and AdaBoost for music classification," *Machine Learning*, vol. 65, pp. 473–84, 2006.

[3] Bray, S., and G. Tzanetakis, "Implicit patching for dataflow-based audio analysis and synthesis," *Proc. ISMIR*, 2005.

[4] Downie, J. S., K. West, A. Ehmann, and E. Vincent, "The 2005 Music Information Retrieval Evaluation exchange (MIREX2005): Preliminary overview," *Proc. ISMIR*, 2005, pp. 320–3.

[5] Ellis, D. P. W., and G. E. Poliner, "Classification based melody transcription," *Machine Learning*, vol. 65, 2006, pp. 439–56.

[6] Fiebrink, R., G. Wang, and P. R. Cook, "Foundations for on-the-fly learning in the ChucK programming language," *Proc. ICMC*, 2008.

[7] Liu, D., L. Lu, and H.-J. Zhang, "Automatic mood detection from acoustic music data," *Proc. ISMIR*, 2003.

[8] Raphael, C., "A Bayesian network for real-time musical accompaniment," *Proceedings of Neural Information Processing Systems*, Vancouver, Canada, 2001.

[9] Rodet, X., "What would we like to see our music machines capable of doing?" *Computer Music Journal*, vol. 15, no. 4, Winter 1991, pp. 51–4.

[10] Rowe, R., *Machine musicianship*. Cambridge, MA: The MIT Press, 2001.

[11] Schwarz, D., A. Cont, and N. Schnell, "From Boulez to ballads: Training IRCAM's score follower," *Proc. ICMC*, 2005.

[12] Tzanetakis, G., and P. R. Cook, "MARSYAS: A framework for audio analysis," *Organized Sound*, vol. 4, no. 3, 2000.

[13] Wang, G., and P. R. Cook, "ChucK: A concurrent, on-the-fly audio programming language," *Proc. ICMC*, 2003.

[14] Wang, G. and P. R. Cook, "On-the-fly programming: Using code as an expressive musical instrument," *Proceedings of the 2004 International Conference on New Interfaces for Musical Expression (NIME),* Hamamatsu, Japan, June 2004.

[15] Wang, G., R. Fiebrink, and P. R. Cook, "Combining analysis and synthesis in the ChucK programming language," *Proc. ICMC 2007.*

[16] Witten, I. H., and E. Frank, *Data mining: Practical machine learning tools and techniques*, 2nd ed. San Francisco: Morgan Kaufmann, 2005.

[1] http://smirk.cs.princeton.edu