CHUCK RACKS: TEXT-BASED MUSIC PROGRAMMING FOR THE DIGITAL AUDIO WORKSTATION

Jordan Hochenbaum
California Institute of the Arts
jhochenbaum@calarts.edu

Spencer Salazar
California Institute of the Arts
salazar@calarts.edu

Rodrigo Sena
California Institute of the Arts
rodrigosena@alum.calarts.edu

ABSTRACT

We present ChucK Racks, a VST/Audio Unit plug-in that brings the ChucK programming language to any digital audio workstation (DAW). ChucK includes many unit generators that can be used to process and generate audio. In developing ChucK Racks, many extensions to the ChucK language were written to facilitate the flow of information between the ChucK virtual machine and the host, including audio, MIDI, automation, transport, and tempo synchronization. This paper describes the extensions to ChucK and how they facilitate meaningful new musical interactions for performers and composers by combining the flexibility of ChucK within their DAW work flow.

1. INTRODUCTION

The maturation and proliferation of digital audio workstation (DAW) software such as Ableton Live and FL Studio has greatly expanded the reach of advanced computer-based music production, extending the digital recording environment from the academic institution or industry studio to the bedroom studio or the garage. Many people now make electronic music using DAW software on a personal computer. Music programming languages are used by a minority of these modern computer musicians; even when users have familiarity with both types of environments, they are seen as independent and worked with separately. Existing ways of integrating text based sound programming and DAWs are complicated and unsatisfactory.

In order to bridge these two fields, we have created a tool called ChucK Racks. ChucK Racks provides an opportunity for electronic musicians who are familiar with the workflow of a DAW to explore the deep possibilities of music coding, and vice versa. ChucK Racks comes in the form of an audio plugin in Steinberg's Virtual Studio Technology (VST) format and Apple's Audio Unit format, intended to be used inside of a DAW. It can be used to make, load, edit, and run programs made in the ChucK programming language inside one of the hosts audio channels. It provides numerous methods to interface ChucK with the DAW host, including receiving timing information, sending and receiving MIDI messages, and automation of internal parameters. In addition to generating sound, it can also be used as an audio effect to process sounds coming from the host, and to programmatically process or generate MIDI control information.

2. RELATED WORK

ChucK Racks draws upon a number of systems for working with the ChucK programming language [1], including the miniAudicle [2], a graphical user interface for editing, executing, and performing with ChucK code. The Faust programming language allows for developed audio processing code in a functional programming language and compiling to audio plugin formats such as VST or Audio Unit, as well as other backend targets [3].

Max for Live is an environment for developing virtual instruments, effects, and generators in the Max programming language, for use in the Ableton Live digital audio workstation. Native Instruments' Reaktor enables the creation of standalone audio processing and synthesis programs using a modular patching interface; these programs can also be imported as plugins for use in a digital audio workstation. [1] Jules' Utility Class Extensions (JUCE) is a C++ framework for writing audio applications supporting diverse platforms such as VST, Audio Unit, stand-alone, and others. [2]

Audacity [4], an audio editing software application, contains an embedded interpreter for the Nyquist programming language [5] for customized processing of audio samples. Cecilia is an audio production environment incorporating processing based on the Csound programming language [6]. Programming systems such as Overtone [7], ixi lang [8], Gibber [9], TidalCycles [10], and Sonic Pi [11] are oriented towards live coding of music during a performance, with explicit support for mainstream and popular genres such as dance or electronic music.

3. MOTIVATION

Digital audio workstations have made computer-based music production techniques accessible to a vast number of music professionals. Using DAWs like Ableton Live or FL Studio and their complements of software plugins, it is possible to produce, mix, and master an entire album. Computer music programming tools such as Max/MSP or ChucK have also expanded the sonic palette available within a single consumer-grade computer, to a more limited audience. These software systems merge the worlds of algo-

[1] https://www.native-instruments.com/en/products/komplete/synths/reaktor-6/

[2] https://www.juce.com/

rithms, process, sound design, and music composition for those who can invest in learning the intricacies of computer programming.

The DAW world and the music programming world both afford distinct advantages to the computer musician. DAWs give precise control and visualization of sequential events with respect to time, such as laying out a polyphonic musical progression or rhythm. The commercially-oriented DAW ecosystem has fostered a vibrant marketplace for audio plugins, including models of well-known studio hardware, advanced software synthesizers, and sophisticated effects processors. Music programming, developed largely from the academic and avant-garde spaces, offers advanced capabilities for encoding musical process, allowing sophisticated developments in rhythm, pitch, scales, tuning, and timbre. Music programming also allows for the automated generation of these parameters, wherein a software process chooses control parameters according to an algorithm or a set of rules, or within the constraints of an existing genre.

The workflows of DAW-based and programmatic music making have largely remained segregated, aside from instances mentioned in Section 2. The introduction of Max for Live has bridged this gap to a significant degree, expanding the role of graphical music programming in Ableton Live and enabling new musical possibilities even for non-programmers. ChucK Racks is intended to bring these two worlds together in the context of textual programming in ChucK, and for any DAW. By integrating music programming with ChucK into a DAW, ChucK Racks enables procedural generation and processing of musical control information and the development of novel audio synthesis and processing techniques in the context of computer music production. Furthermore, the ChucK Racks interface is built around a text editor for ChucK code, allowing musicians and programmers to quickly sketch out ideas in code within an existing music production environment. One of the most powerful features of many DAWs is automation, the ability to specify and hand-tune changes in a musical property over time. Introducing these capabilities into a text-based music programming environment allows for controlling time-varying parameters that might be cumbersome or over-complicated to express in code. The ability to easily lay out melodies and rhythmic patterns in a DAW could allow for efficient experimentation with synthesis and sound design in ChucK. As well, combining text programming techniques with commercial-grade audio plugins might yield improvements to overall sound quality of text-based programming compositions.

4. DESIGN AND IMPLEMENTATION

ChucK Racks includes a light-weight integrated development environment (IDE) that makes it possible to create, modify, or otherwise play with ChucK programs on-the-fly. This makes real-time interaction with ChucK possible in a number of musical contexts from initial ideation, experimentation, composition, arrangement, mixing, and live performance. The remainder of this section describes the general layout and design of the ChucK Racks, as well as it's key features and capabilities.

The ChucK Racks plugin user interface is divided into three primary areas: (a) the main toolbar; (b) the editor; and (c) the console (see Figure 1). A fourth area for parameter mapping and automation will be discussed late in section 4.4.

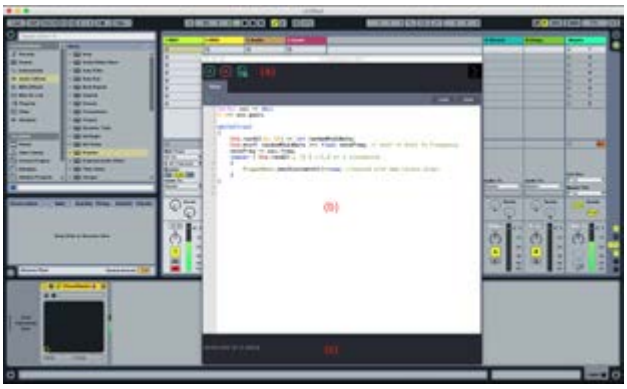

Figure 1. Main ChucK Rack User Interface

4.1 The Main Toolbar

One of the strengths of the ChucK language is the ability to run many concurrent processes called "shreds". Concurrency in ChucK is also 'sample-synchronous', meaning that inter-process audio timing is guaranteed to be precise down to the sample. This enables all code across all code editor tabs to be added or removed at exactly the same time.

The main toolbar (see Figure 1a) includes the basic functionality needed to do just that. The Add All button ('+' icon), simultaneously adds/runs all ChucK code present in the plugin. The Remove All button ('x' icon), removes all shreds running in the VM. The Add All button can be triggered as many times and whenever it is desired, allowing for the same program to be executed concurrently with previous instances. Additionally, through the PluginHost interface described later in section 4.5.1, users can configure their ChucK programs to synchronize execution with the plugin host's tempo and play position. This is useful, for example, to ensure that code is always added on the nearest 1/16th note, the nearest bar, or some other beat-division of the main host tempo. Lastly, the "New Document" icon allows the user to add as many concurrent tabs (ChucK programs) to the project as needed. Being able to add independent tabs allows one to quickly iterate on ideas and variations, and to facilitate this type of sketching and experimentation, tabs can be both duplicated and removed. The content in each tab can be quickly saved and recalled later from standard ChucK .ck files - this is useful not just for writing finished works, but also building blocks that can be re-used, recycled, and re-imagined in future works or performances.

4.2 The Editor

ChucK Racks can have multiple tabs where ChucK files are created, opened, or edited. Each tab includes a code editor (see Figure 1b) equipped with syntax highlighting for the entire ChucK and ChucK Racks APIs. In addition to it's primary code editing area, each tab contains an Add button that enables it's code to be added to the VM concurrently, but independently, of the code in other tabs.

4.3 The Console

The console area (Figure 2) can be expanded up from the bottom of the plugin window and enables all ChucK error messages or warnings from the VM to be displayed. The console can also used for debugging, as all print statements encountered in the running programs are displayed.

Figure 2. Debugging in the Console

4.4 Parameter Mapping and Automation

An advantage of working with a DAW and compatible plugins is the possibility for the plugin to expose the values of some internal parameters. The DAW can typically access those parameters and allow the artist to control them by either mapping them to a MIDI controller, or drawing/recording in the parameter's changes in an automation lane.

A novel way of interacting with custom parameters was developed in ChucK Racks, opening up a powerful technique to control variables in the ChucK program(s) in real-time. Since automation follows the DAWs tempo and can be quantized to various beat-divisions of the arrangement, achieving time or beat synced modulations, and other musical automation effects, is extremely easy. Figure 3 shows an example of automation drawn and quantized on a 1/16th note grid.

This functionality can also be used to modify the program over the duration of an entire composition, making it possible to finally integrate ChucK programs in the context of a through-composed composition, or any other context where written and fixed automation changes are needed. There are many other possibilities as well, e.g. performers using DAWs like Ableton Live and Bitwig can store predefined automation curves in MIDI clips, which they can then use to trigger parameter automation gestures in the ChucK programs on-the-fly when performing live.

Figure 3. Automation on a 1/16th grid timeline in Ableton Live

Adding new parameters can be done through the Parameter Panel (Figure 4), which can be expanded/collapsed via an arrow on the right-hand side of the screen. All parameters added to a plugin instance are automatically saved and recalled with the DAW project. New parameters can be added and named, and their values mapped to the host, simply by enabling the host's mapping function, right-clicking on the parameter, and selecting 'map'. Naturally, because most hosts can map plugin parameters from an audio plugin to a MIDI controller, it is thus possible, without writing any additional code, to map a MIDI controller directly to variables within the code, and recall those mappings with the project.

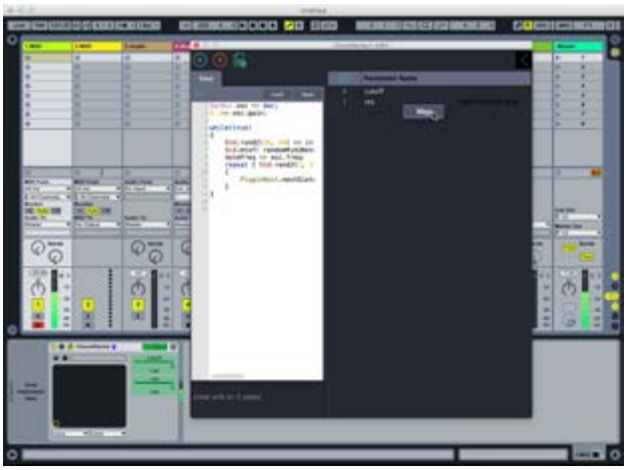

Figure 4. Mapping Automatable Plugin Parameters to the Host

4.4.1 Accesing Parameter Values in Code

A new extension called *PluginParameters* was added to ChucK to access user defined plugin parameters. Calling the *PluginParameter::getValue()* function from anywhere in the ChucK code, retrieves the current value of *param* as a floating point number between 0.0 and 1.0.

```
PluginParameters.getValue("volume") => osc.gain;
```

Listing 1. Mapping Parameter called *volume* to oscilattor gain in ChucK

Listing 1 shows how to map a plugin parameter called *volume* to the gain of an oscillator called *osc*. Further scaling the normalized values provided by PluginParameters to some other desired range is straightforward. Listing 2 shows an example of mapping a parameter called *cutoff* to the cutoff frequency of a lowpass filter called *filter*. The parameter's value is first skewed exponentially, before finally converting it's range to be between 50 and 10000.

```
//values from plugin parameters are between 0 and 1
PluginParameters.getValue( "cutoff" ) => float valueFromParam;

//since it's for frequency we want an exponential response
valueFromParam * valueFromParam => float expValue;

//scale to desired range of the filter (in Hz)
Std.scalef( expValue, 0.0, 1.0, 50.0, 10000.0 ) => filter.freq
```

Listing 2. Scaling a parameter's value to another range

The above examples demonstrate how to query the value of a plugin parameter instantaneously, however, it is often the case that a variable should be updated continuously, outside of the current code block in ChucK. Listing 3 defines a poller function in ChucK called *updateVolume()*, which runs infinitely, and updates an oscillator's gain with the plugin's *volume* parameter every 10ms. This is achieved using *spork*, a ChucK language construct which allows a function call to be dynamically added to the virtual machine and run in its own concurrent shred (see 4.1 for a brief overview of shreds). Note, because ChucK gives the user complete control of time, it is up to the user to decide what rate to update the value at. In this example, the

authors chose to update the value every 10ms, but ChucK supports as low as subsample rates, and other durations such as milliseconds, seconds, minutes, hours, days, and weeks. Furthermore, as ChucK supports the definition of custom durations, it is possible to make the polling rate a function of host's tempo (accessing the host's tempo will be discussed later in section 4.5.1)

```
spork ~ updateVolume();

fun void updateVolume(){
    while(true) {
        PluginParameters.getValue("volume") => osc.gain;
        10::ms => now;
    }
}
```

Listing 3. Updating Plugin Parameters Continuously

Additionally, parameters are created at the plugin level and are thus accessible (and can be mapped) across one or more code files. This allows for typical one-to-one mappings (e.g. a plugin parameter called *cutoff* which is mapped to the cutoff frequency of a filter on a synthesizer) but also one-to-many mappings. These 'macro' parameters can thus control many disparate variables across multiple ChucK programs simultaneously, from a single automation lane or MIDI controller input. Of course, their values can be skewed in the code where they are mapped to make non-linear relationships across the mappings.

4.5 Sharing Information Between ChucK and the Host

Section 4.4 described some of the extensions necessary to be able to map plugin parameters between the host and code running in ChucK Racks. The remainder of this section will describe a number of extensions that facilitate the exchange of information including host information such as transport state and tempo synchronization, as well as audio, and MIDI.

4.5.1 PluginHost API

A challenge users face when trying to integrate ChucK with other software is the sharing of critical timing information, like tempo, between the two. The *PluginHost* API was added to ChucK to make this easier, and to give the user direct access to host information and events.

Firstly, a number of static methods to communicate useful information to/from the host can be called on-demand, from code in ChucK Racks. Table 1 provides an overview of the API. Using these functions, it is possible to get the host's tempo and time signature, to query if the transport is playing or stopped, and if playing, the current play position, the position in the current beat, the position in the current bar, as well as the start position of the last (current) bar. Using these functons, a ChucK programmer has the ability to make most timing calculations they might need to control their musical works. Furthermore, a number of convenience functions were added to obtain common note durations based on the host tempo as ChucK *dur* objects (ChucK's unit of time). Lastly, a function was also created to send MIDI out of ChucK Racks, and will be described later in section 4.5.3.

As detailed in Table 2, a number of ChucK event callbacks have also been added to the PluginHost API to synchronize ChucK to the host. This includes being notified on play and stop events from the host's transport, incoming midi messages from the host's sequencer or midi track input, as well when common beat divisions are passed (e.g. the down beat of the next bar, or the next sixteenth note).

Table 1. Overview of PluginHost Functions

Function	Info
float getTempo ()	Returns the current tempo in Beats-Per-Minute
int timeSigUpper()	Returns the number of beats in a bar (3 in 3/4)
int timeSigLower()	Returns the note value of one beat (the beat unit, 4 in 3/4)
int isPlaying ()	Returns 1 is host is currently playing, 0 if it is stopped
float pos()	Returns play position (in quarter notes)
float posInBeat()	Returns play position in current beat between 0.0 and 0.9999
float posInBar()	Returns play position in current bar between 0 and number of quarter notes in time-signature (e.g 0.0 - 3.999 in 4/4)
float posLastBarStart()	Returns start position of the last bar in quarter notes
dur barDur()	Returns the length of a bar
dur halfDur()	Returns the length of a half note
dur quarterDur()	Returns the length of a quarter note
dur eighthDur()	Returns the length of a eighth note
dur sixteenthDur()	Returns the length of a sixteenth note
void sendMidi (MidiMsg msg)	Sends a midi message *msg*

Table 2. Overview of PluginHost Events

Event	Info
onPlay ()	Triggered when host transport starts playing
onStop ()	Triggered when host transport stops playing
onMidi ()	Triggered when new midi event is available
int PluginHost.recvMidi(MidiMsg msg)	Triggered when midi messages are received from host Returns 1 if message was received.
nextBar()	Triggered on the start of the next bar
nextHalf()	Triggered on the next half note
nextQuarter ()	Triggered on the next quarter note
nextEighth ()	Triggered on the next eighth note
nextSixteenth ()	Triggered on the next sixteenth note

4.5.2 Audio Input, Output, and Processing

Audio can be streamed in and out from ChucK Racks using ChucK's audio signal graph. Typically the *adc* unit generator (UGen) in a ChucK program represents an input device such as a microphone, and the *dac* UGen represents the underlying audio cards output. ChucK Racks takes advantage of this system and uses the adc UGen for the audio coming into the plugin from the host, and the dac UGen is simply the plugin's main output buffer. In doing so, code running in ChucK Racks can serve as either an audio effect or virtual instrument (synthesizer) plugin, and be portable from standard ChucK to ChucK Racks.

```
adc => LPF filter => dac;
```

Listing 4. A Simple Low-pass Filter Audio Effect in ChucK Racks

Listing 4 demonstrates a simple audio effect plugin in ChucK Racks, where incoming audio is processed through ChucK's low-pass filter UGen. It should be noted that ChucK's audio graph allows multiple unit generators to be chained together in between the adc and dac, and so a single instance of ChucK Racks, placed inline as an audio effect on a DAW track, can actually be an arbitrarily complex chain of audio processors. The rest of the program controls how those processors then shape the sound, either programmatically, algorithmically, or through other means like automation and real-time user input. ChucK includes a number of built-in classes for audio synthesis, processing, and analysis, with several mechanisms to create custom sample-rate synthesizers and processors [12].

4.5.3 MIDI Processing

It is essential that ChucK Racks provides bi-directional MIDI message handling with the host. By receiving MIDI note and Control Change (CC) information, custom synthesizers and samplers can be created in ChucK Racks, and used like any other virtual instrument plugin. This enables the user to sequence and arrange their ChucK virtual instrument using the DAW's sequencer, arrangement, and score editors. The ability to send MIDI out of ChucK Racks allows programmers and artists to take full advantage of ChucK's strong timing facilities to build novel sequencers to control other audio plugins. Combining both MIDI input and output, it is also possible to build real-time MIDI effects, like arpeggiators, scale quantizers, chord generators, and more. In this way, ChucK racks can become a modular environment for virtual instruments. The remainder of this section will describe in greater detail how MIDI information is communicated between ChucK Racks and the host.

ChucK Racks works alongside ChucK's existing MIDI messaging and event classes, and adds new methods to PluginHost: *onMidi()*, *sendMidi()*, and *recvMidi()*. Listing 5 responds to MIDI messages from the host, by waiting on the *PluginHost.onMidi()* event, and then unpacking the incoming MIDI message into a regular ChucK MidiMsg object.

```
MidiMsg msg;

while(true) {
    PluginHost.onMidi() => now;
    while( PluginHost.recvMidi(msg) ) {
        <<< msg.data1, msg.data2, msg.data3 >>>;
    }
}
```

Listing 5. Receiving and Unpacking MIDI from Host

Sending MIDI out from ChucK Racks is also straightforward. Listing 6 sends MIDI out of ChucK Racks using ChucK's standard MidiMsg object.

```
while(true) {
    MidiMsg msg;
    0x90 => msg.data1;
    60 => msg.data2;
    127 => msg.data3;
    PluginHost.sendMidi(msg);
    1::second => now;
}
```

Listing 6. Sending MIDI to Host

4.6 libchuck

ChucK Racks integrates a ChucK compiler and virtual machine in the form of libchuck, a C++ library version of ChucK designed to be embedded into larger applications. [3] libchuck provides functionality for compiling code, reporting code errors, and running or removing code from the virtual machine. Through libchuck, ChucK's vm can be executed in conjunction with an existing real-time audio engine, such as that of a DAW, by requesting the desired number of samples from libchuck. libchuck also allows a host application to load customized chugins[12] to extend the default functionality of ChucK with new unit generators and classes that can interact with the host in sophisticated ways. For instance, this allows ChucK Racks to set up the PluginHost and PluginParameters classes for interfacing with the host DAW environment.

[3] Available at `https://github.com/spencersalazar/libchuck`

5. EXAMPLES

5.1 Quantized Musical Phrase Launching

This example (Listing 7) shows how to quantize a musical section using the PluginHost API event callbacks and durations. Each bar, the function *measure* is executed in it's own shred. *measure()* generates a random note value within a 2-octave range and assigns the result to the frequency of sine oscillator *s*. The note plays for exactly 1 quarter note using *PluginHost.quarterDur()*, and this is repeated four times to complete the measure.

```
SinOsc s => dac;

fun void measure() {
    for (0 => int i; i < 4; i++) {
        Std.mtof(Math.random2(60,84)) => s.freq;
        PluginHost.quarterDur() => now;
    }
}

while (true) {
    PluginHost.nextBar() => now;
    spork ~ measure();
}
```

Listing 7. Quantized Musical Phrase Launching

5.2 Sequenced Low-pass Filter

This audio effect processes the DAW's audio through ChucK's low-pass filter. A 16-step sequence of filter "cutoff" values is randomly generated, and the sequence is applied sequentially to the filter, in sync with 1/16th notes from the host. The cutoff value is multipled by a master cutoff PluginParameter, which the user can automate in the host, or map to a midi controller to scale the values of the cutoff modulation in real-time.

```
adc => LPF lpf => dac;

16 => int numberOfSteps;
float sequence[numberOfSteps];

for (int i; i<numberOfSteps; i++) {
    Math.randomf() => sequence[i];
}

while(true) {
    for(int i; i<numberOfSteps; i++) {
        sequence[i] => float value;
        PluginParameters.getValue("cutoff") *=> value;
        value*value => value;
        Std.scalef(value, 0.0, 1.0, 50.0, 10000.0) => lpf.freq;

        PluginHost.nextSixteenth() => now;
    }
}
```

Listing 8. Sequenced Low-pass Filter

5.3 Quantized Cellular Automata

ChucK Racks bridges the gap between algorithmic composition in ChucK and through-composed pieces in a DAW. In Listing 9, a simple celluar automata based sequencer generates pentatonic notes which are sent out of the plugin to another software instrument or sampler. The events are quantized to sixteenth notes in the DAW, to keep the events in time with the rest of the composition. It is also possible to extend this example, by automating parameters of the algorithmic system, allowing the composer to influence the system over time.

```
// Cellular Automata rule
110 => int rule;
12 => int rhythmLength;

// Cellular Automata binary input
1 => int input;
[0, 2, 4, 7, 9, 12, 14, 16, 19, 21, 24, 26] @=> int
      pentatonicScale[];

while (true) {
    int output, lookup, state;
    for (0 => int i; i < rhythmLength; i++) {
        // Cellular Automata bit math logic
        if (i == 0)
            (input >> rhythmLength - 1) | ((input & 3) << 1) =>
     lookup;
        else if (i == (rhythmLength - 1))
            ((input >> i) & 3) | ((input & 1) << 2) => lookup;
        else
            (input >> (i - 1)) & 7 => lookup;

        // Cellular Automata bit math results
        (rule >> lookup) & 1 => state;
        (state << i) | output => output;

        if (state == 1) {
            MidiMsg msg;
            0x90 => msg.data1;
            48 + pentatonicScale[i] => msg.data2;
            Math.random2(80, 127) => msg.data3;
            PluginHost.sendMidi(msg);
         }

        PluginHost.nextSixteenth() => now;
    }

    output => input;
}
```

Listing 9. Cellular Automata Quantized to Sixteenth Notes

5.4 Automatable Wavefolder Distortion

This DSP audio effect applies a custom wavefold distortion Chugen to the incoming audio. The wavefolder's threshold can be automated through a PluginParameter.

```
class Wavefolder extends Chugen {
    0.1 => float threshold;

    fun float tick(float in) {
        if (in > threshold)
            threshold - (in - threshold) => in;
        else if (in < -threshold)
            -threshold + ( -threshold - in) => in;
        return in;
    }
}

adc => Wavefolder myWavefolder => dac;

while(true) {
    PluginParameters.getValue("foldingThreshold") => float val;
    Std.dbtolin(Std.scalef(val, 0, 1, -80, 0)) => float amnt;
    amnt => myWavefolder.threshold;
    10::ms=>now;
}
```

Listing 10. Automatable Wavefolder Distortion Effect

6. CONCLUSIONS

While many artists have familiarity working with DAWs, music programming languages, or both, they are often seen as independent due to the challenges in combining the two effectively. Yet both have unique affordances that musicians and composers take advantage of. We have developed a VST/Audio Unit plugin, ChucK Racks, that leverages the strengths of both, by combining the flexibility of computer music software programming with the workflow of a DAW. ChucK Racks contains a number of unique features that make it effective in a wide range of musical contexts, fulfilling our goals of unifying the workflows of music coding and DAWs and shortening the path between iterating on musical ideas and iterating on code.

ChucK Racks is currently available in source code form and will be made available as a binary release at: `http://mtiid.calarts.edu/projects/software/chuck-racks/`

Acknowledgments

The authors wish to thanks Eric Heep and Jake Penn for their code contributions to the ChucK Racks project.

7. REFERENCES

[1] G. Wang, P. R. Cook, and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal*, 2016.

[2] S. Salazar, G. Wang, and P. Cook, "miniAudicle and ChucK Shell: New interfaces for ChucK development and performance," in *Proceedings of the International Computer Music Conference*, 2006, pp. 63–66.

[3] Y. Orlarey, D. Fober, and S. Letz, "FAUST: an efficient functional approach to DSP programming," *New Computational Paradigms for Computer Music*, 2009.

[4] D. Mazzoni and R. B. Dannenberg, "A fast data structure for disk-based audio editing," *Computer Music Journal*, vol. 26, no. 2, pp. 62–76, 2002.

[5] R. Dannenberg, "The Nyquist Composition Environment: Supporting Textual Programming with a Task Oriented User Interface," in *Proceedings of the International Computer Music Conference*, 2008.

[6] J. Piché and A. Burton, "Cecilia: A production interface to Csound," *Computer Music Journal*, vol. 22, no. 2, pp. 52–55, 1998.

[7] S. Aaron and A. F. Blackwell, "From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages," in *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*. ACM, 2013, pp. 35–46.

[8] T. Magnusson, "ixi lang: a SuperCollider parasite for live coding," in *Proceedings of International Computer Music Conference*. University of Huddersfield, 2011, pp. 503–506.

[9] C. Roberts and J. Kuchera-Morin, "Gibber: Live coding audio in the browser," in *Proceedings of the International Computer Music Conference*, 2012.

[10] A. McLean, "Making programming languages to dance to: live coding with Tidal," in *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. ACM, 2014, pp. 63–70.

[11] S. Aaron, "Sonic Pi–performance in education, technology and art," *International Journal of Performance Arts and Digital Media*, vol. 12, no. 2, pp. 171–178, 2016.

[12] S. Salazar and G. Wang, "Chugens, Chubgraphs, Chugins: 3 Tiers for Extending Chuck," in *Proceedings of the 38th International Computer Music Conference*, 2012.