(draft 16 Mar 2023)

What would a Webchuck Chuck?

Chris Chafe, Jack Atherton, Ge Wang
Center for Computer Research in Music and Acoustics (CCRMA), Stanford University

Abstract

Webchuck is a new platform for real-time web-based music synthesis. Combining expressive music programming power with the ubiquity of web browsers is a game changer, one which we've experienced in recent teaching at Stanford's CCRMA. The article describes how this has fulfilled a long-sought promise for easily accessible music making and simplicity of experimentation. The Chuck music programming language now runs anywhere a browser can and a link is all it takes. Sample-synchronous live coding alongside full-on studio functionality is what a Webchuck can Chuck, thanks to advances in the Web Audio API.

Introduction

This article assumes familiarity with the Web Audio API [1][2] and introduces a real-time live coding environment that runs on top of it. Chuck, the music programming language, was invented by Ge Wang and Perry Cook [3] [4]. Like the Web Audio API (WAAPI) it provides constructs for digital signal processing (DSP) unit generators (UGens) and their manipulation in time.

The differences between Chuck and WAAPI run deep but at the outset we want to stipulate that it's usually possible to code a given project / program / audio task in either of them. Assembling patches of UGens, changing their parameter values and hooking them to digital-to-analog signal outputs (DACs) and analog-to-digital signal inputs (ADCs) is the essence of the kinds of programs that are supported by the two platforms. Chuck has its own syntax which, like WAAPI, is text based, but unlike WAAPI is not written as an extension of another language (namely JavaScript). Chuck reinvents much of what a programming language needs to provide while the WAAPI gains comprehensiveness through its derivation from a full-featured language.

Differences favoring one or the other for a given project are inherent in their designs but foremost it's the ability to easily structure time and musical texture that sets Chuck apart. Chuck syntax provides expressions for temporal manipulation from the sample level up and a primary aspect is its top-level spawning of concurrent processes. Computer music languages like PLA [5] and Common Music [6] are probably closest in how they express musical concurrency. The WAAPI, out-of-the-box, provides lower-level constructs which custom JavaScript code can use in higher-level abstractions. An early attempt by one of the authors at a Chuck-like set of modules for structuring time [7] was an inviting prospect but did not succeed nearly as well as Wechuck. With Webchuck [8], Chuck itself comes assembled as a callable program using web assembly code. A web page's JavaScript connects it to WAAPI facilities for connection to the page's audio. The Web MIDI API [9] is also available to the embedded Chuck environment. Webchuck is identical to native Chuck and brings all the advantages of deployment in browsers.

The following sections present examples of projects built in Webchuck along with some of its history and possibilities for the future.

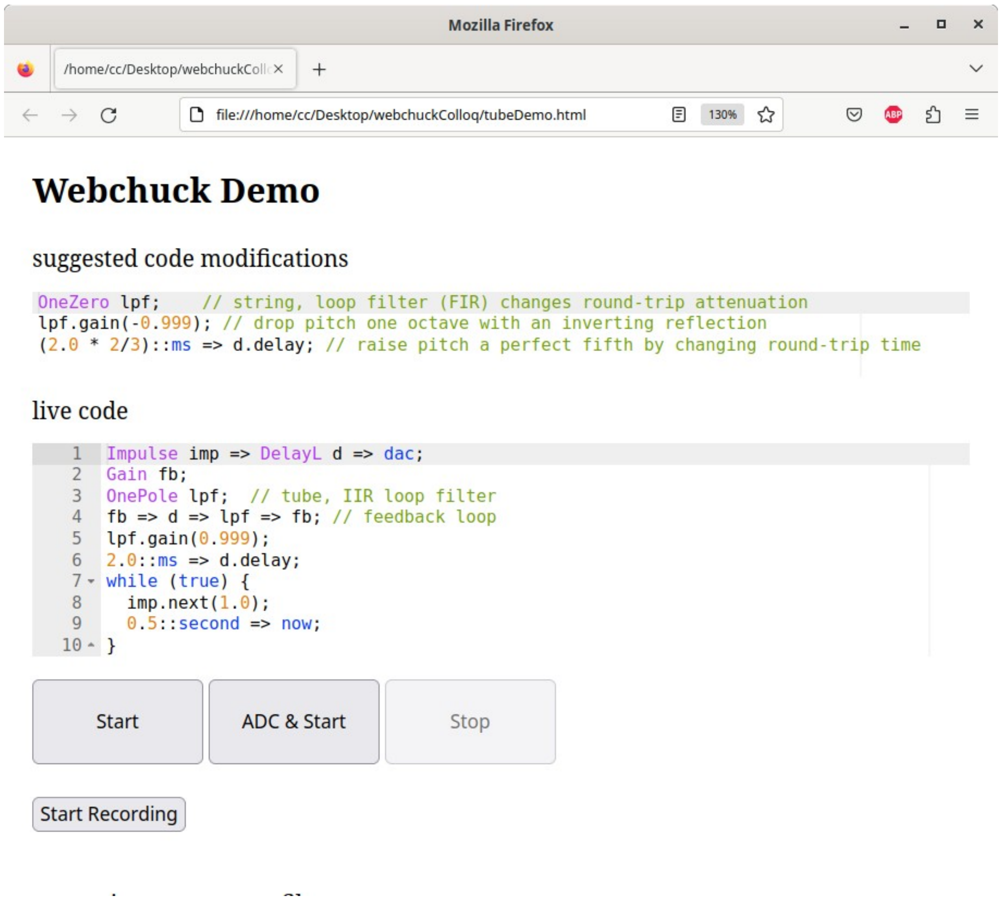
Webchuck example applications

At CCRMA there's been an eternal course on computer-generated sound that's been on the books since the 1970's. The latest incarnation of Music 220a (Fall 2022) embraced Webchuck for the first time. The switch to the new platform was logical in that it followed almost a decade of offering native Chuck teaching examples and WAAPI teaching examples side-by-side. Webchuck allows the same examples to run as live code and all examples have been converted to interactive web pages. Similar to web-based notebooks, for example, Jupyter [10] or IDE’s [11], participants go to a URL where a program is displayed and executes at the push of a button. In the Music 220a examples, Chuck code is embedded in a live code editor block. Running code can be replaced interactively and since it's a web page, basic html controls such as sliders and displays that are running in real time can be included in the program.

TubeDemo

TubeDemo (Fig. 1) is a self-contained example of the kind used in the class. It's a demonstration of a simple waveguide resonator that plays a series of repeated tones after the "start" button is pushed (at which point the button changes its name to "replace"). The sound is like tapping on the end of a PVC pipe. Fig. 2 shows the signal flow in the program. The "suggested modifications" present some possible single line modifications to the program for a student to try out in the live code. These produce two alternative instruments: a closed-end tube and a stretched string. The initial example is a strongly attenuated open-open tube with a OnePole low-pass filter in the feedback loop. By changing to a simulation of a closed-end tube with an inverting reflection, the pitch drops one octave. The stretched string modification uses a less-attenuating and milder OneZero low-pass filter in the loop that causes the sound to ring longer.

Hitting the "Start Recording" button captures audio output to an in-memory blob. When "Stop" is clicked, the blob can be written to a .wav file on the local file system.


Mozilla Firefox

/home/cc/Desktop/webchuckColl

file:///home/cc/Desktop/webchuckColloq/tubeDemo.html

Webchuck Demo

suggested code modifications

```
OneZero lpf;    // string, loop filter (FIR) changes round-trip attenuation
lpf.gain(-0.999); // drop pitch one octave with an inverting reflection
(2.0 * 2/3)::ms => d.delay; // raise pitch a perfect fifth by changing round-trip time
```

live code

```
Impulse imp => DelayL d => dac;
Gain fb;
OnePole lpf;  // tube, IIR loop filter
fb => d => lpf => fb; // feedback loop
lpf.gain(0.999);
2.0::ms => d.delay;
while (true) {
  imp.next(1.0);
  0.5::second => now;
}
```

Figure 1. **TubeDemo.html** web page.

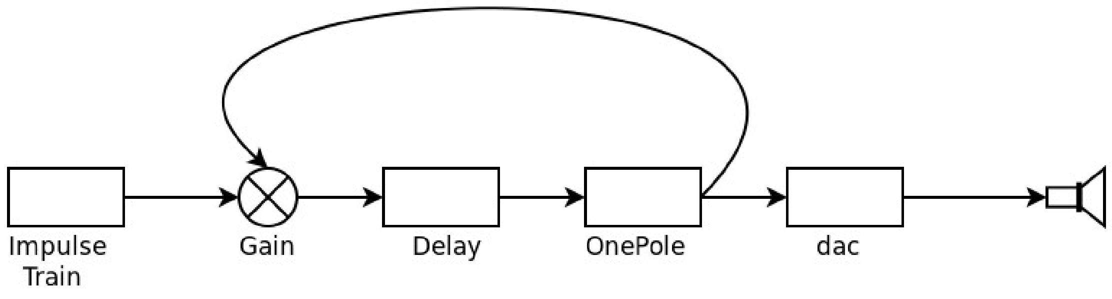


Figure 2. A simple waveguide resonator circuit.

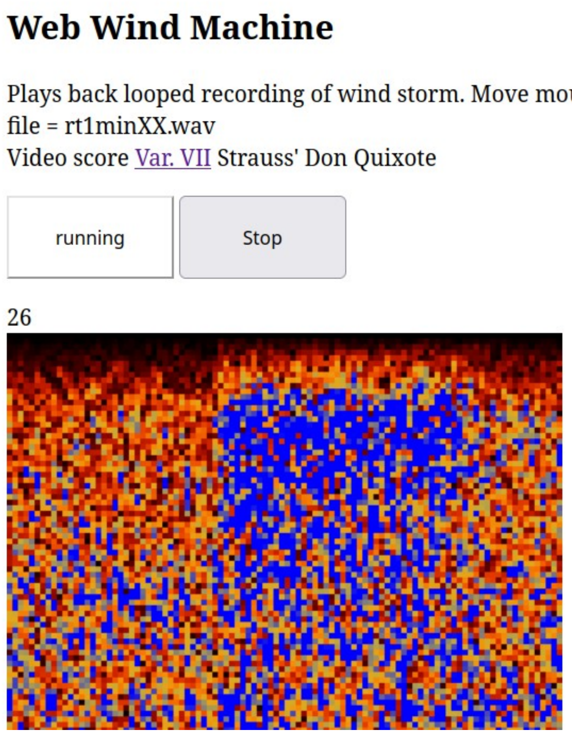
WindMachine

The WindMachine Webchuck application shown in Fig. 3 was developed to simulate the orchestral wind machine called for in Richard Strauss' Don Quixote. There is no live code in this deliverable web app but live coding was used to tweak the program during development. Strauss calls for a hand-cranked instrument that creates wind-like swells alongside the orchestral music of Variation VII. The web app has a scrolling FFT display, right to left, indicating level of spectral energy (from black through orange to blue).

The spectral display also functions as a touch-sensitive input responding to X-Y movement across its region. The performer makes gestures across its surface carving swells in volume and frequency. Playing scary, dynamic wind gusts that match the mechanical wind machine's sound is the intended goal.

Smoothly animating the page's canvas element [12] is tricky. The Chuck program produces FFT frames every 21 msec and the display is redrawn asynchronously by JavaScript D3 code embedded in the **.html** file [13].

Any orchestra, anywhere, can rehearse using this web app which is freely downloadable. It runs equally well on laptop, tablet, or phone with pointing devices like mouse, trackpad, or mobile touch. The web app page includes a handy link to a scrolling video score of Variation VII for practice, Fig. 4. Setting it up so that one browser instance (or tab) is the instrument and another is playing the orchestral recording is convenient for learning the part. Browsers usually mix the audio output of multiple tabs and instances, a feature that's used to advantage in this scenario.


Web Wind Machine

Plays back looped recording of wind storm. Move mouse over graph (X = volume, Y = brightness)
file = rt1minXX.wav
Video score Var. VII Strauss' Don Quixote

Figure 3. **WindMachine.html** web page with link to Video Score (Fig. 4).

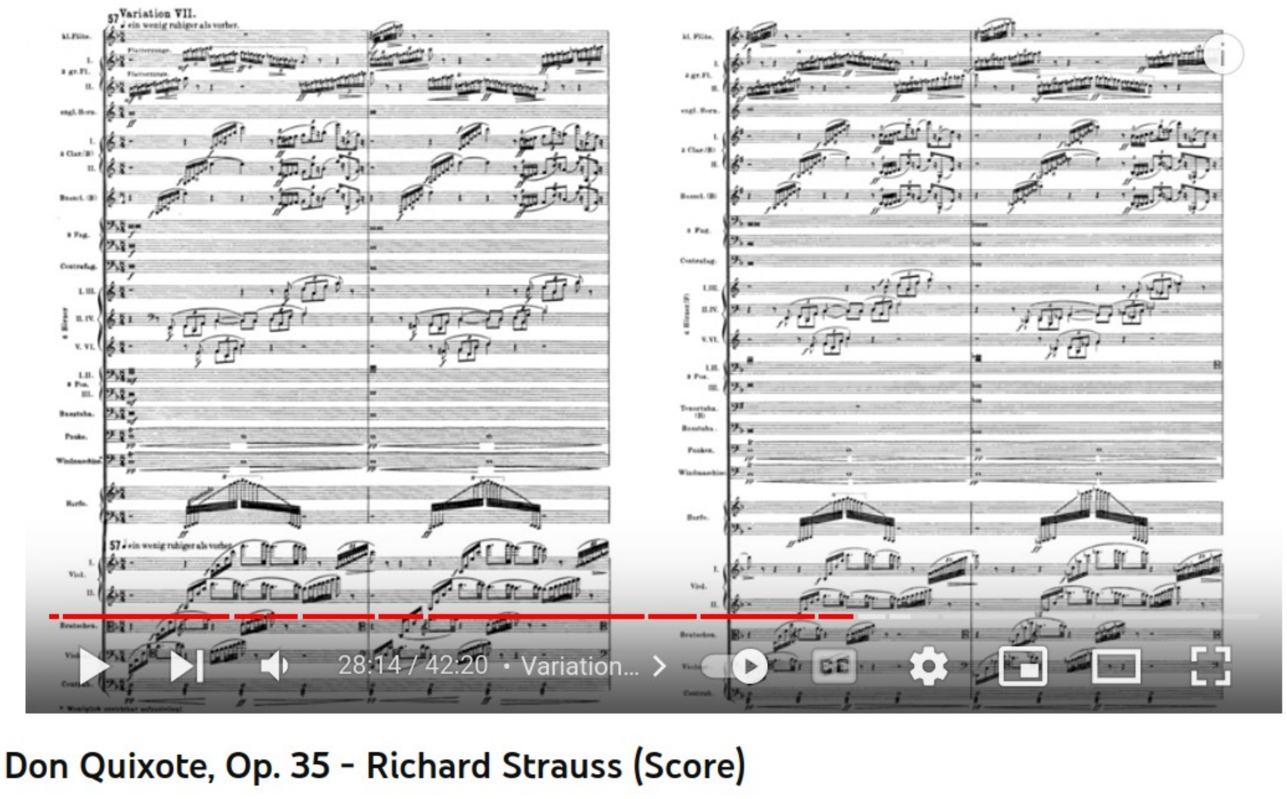

Variation VII.
28:14 / 42:20 • Variation...
Don Quixote, Op. 35 - Richard Strauss (Score)

Figure 4. Video Score of Don Quixote for the WindMachine Webchuck application instrument (Fig. 3) so a performer can learn their part.

Sonification

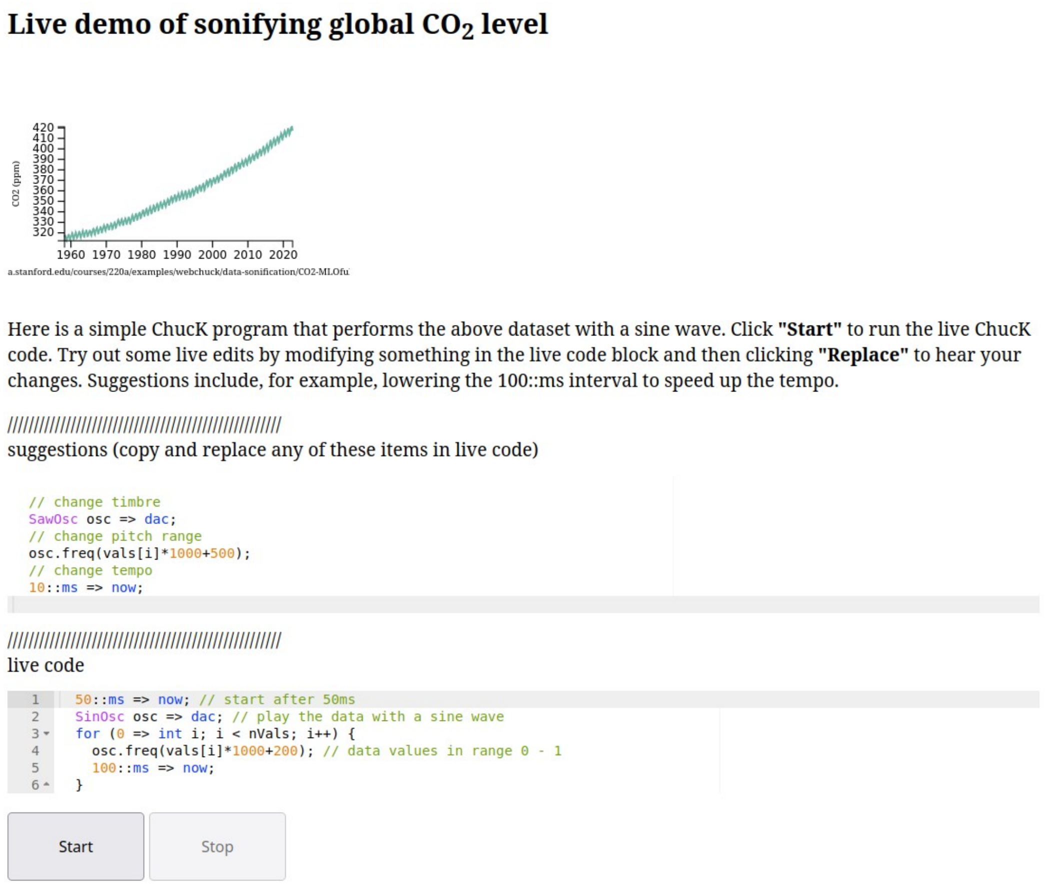
SonifyCO2.html was developed for introducing sonification methods in several ongoing workshops and in Music 220a. In this example, a data set from the Mauna Loa Observatory [14] drives changes to the frequency of a sinusoidal oscillator in real time.

Raw data from the period of 1960 to 2020 lies in the range of 320 to 420 ppm. Three aspects are immediately clear to the eye from the graph displayed. There's an inexorable increase in CO_2 we know to be happening, there is a curved shape to it that is accelerating, and it is inscribed with seasonal oscillations.

Hitting the “start” button plays 60 years worth of data compressed in time to about 80 seconds. The sound is something like a teapot whistling. The metaphor is inescapable. At 80 seconds, the amount of seasonal "vibrato" it is very prominent to the ear. But what about the exponential curve visible in the data graph? Changing the sound, mappings and timebase may be helpful to bring that out. The workshops this example is used for focus on enhancements that bring out qualities of the data set.

Suggested one line modifications to the live code include changing the oscillator's waveshape (to sawtooth), changing the mapping of data to frequency (by increasing or decreasing the excursion), and speeding up the update rate so that playback happens an order of magnitude faster or more. These hints are a starting point for taking the sonification in directions where nuances in the data are more audible. Ultimately, it's a matter of experimentation.

Making the exponential "hockey stick" curve audible is just barely possible with only the six decades worth of data. But add in the previous one hundred years of CO_2 record from other sources and it becomes an alarming sound.



Live demo of sonifying global CO_2 level

a.stanford.edu/courses/220a/examples/webchuck/data-sonification/CO2-MLOfu

Here is a simple ChucK program that performs the above dataset with a sine wave. Click **"Start"** to run the live ChucK code. Try out some live edits by modifying something in the live code block and then clicking **"Replace"** to hear your changes. Suggestions include, for example, lowering the 100::ms interval to speed up the tempo.

///
suggestions (copy and replace any of these items in live code)

```
// change timbre
SawOsc osc => dac;
// change pitch range
osc.freq(vals[i]*1000+500);
// change tempo
10::ms => now;
```

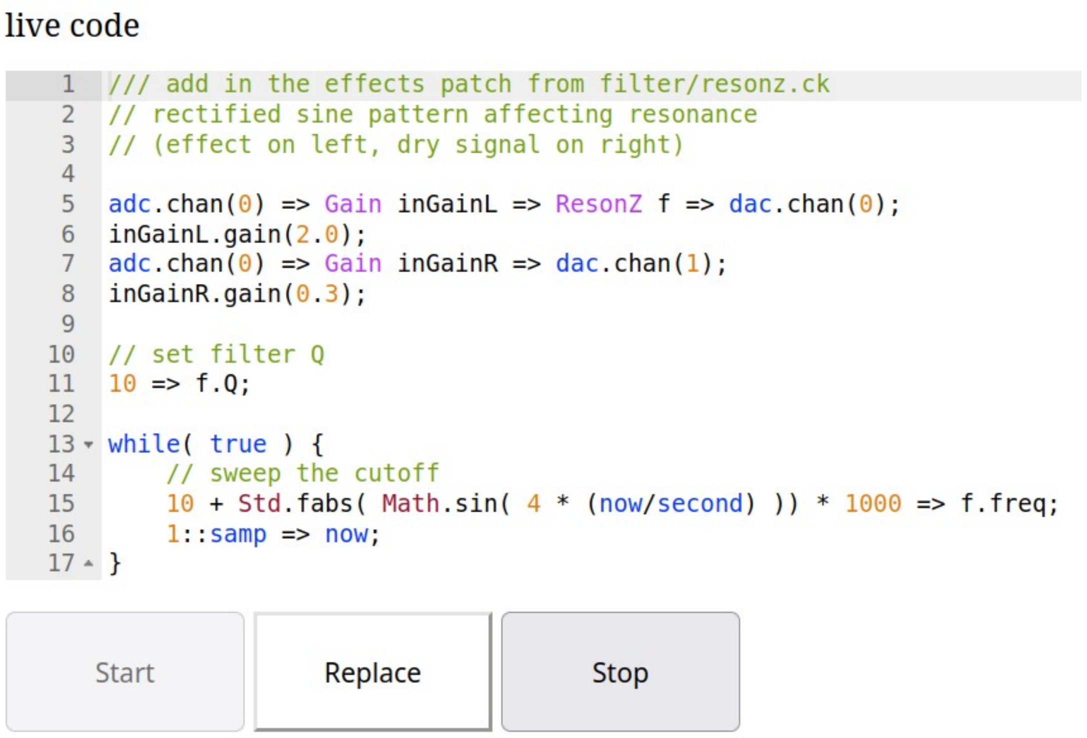
///
live code

```
50::ms => now; // start after 50ms
SinOsc osc => dac; // play the data with a sine wave
for (0 => int i; i < nVals; i++) {
  osc.freq(vals[i]*1000+200); // data values in range 0 - 1
  100::ms => now;
}
```

Figure 5. Web app for sonifying CO2 level.

Audio processing

Connecting Webchuck's adc input to effects processing is shown in Fig.5. On every audio sample, the a resonator filter’s frequency term is set by the output of a slowly changing sine function. The low-frequency oscillation could be rewritten to use the instantaneous value of a SinOsc UGen.


live code

```
/// add in the effects patch from filter/resonz.ck
// rectified sine pattern affecting resonance
// (effect on left, dry signal on right)

adc.chan(0) => Gain inGainL => ResonZ f => dac.chan(0);
inGainL.gain(2.0);
adc.chan(0) => Gain inGainR => dac.chan(1);
inGainR.gain(0.3);

// set filter Q
10 => f.Q;

while( true ) {
    // sweep the cutoff
    10 + Std.fabs( Math.sin( 4 * (now/second) )) * 1000 => f.freq;
    1::samp => now;
}
```

Figure 6. Audio processing with Chuck adc sent through resonance filter.

Theory of Operation

When a Webchuck web page is loaded by a browser, it runs the top-level file **webchuck_host.js** which sets up the system. Two components are needed: a monolithic web assembly file and the JavaScript interface to it. The web page itself currently specifies where the top-level file resides, and whether it's on a server or in a local file path. The top-level file **webchuck_host.js** similarly specifies the location of the other two system files.

The top-level file **webchuck_host.js** establishes an instance of AudioContext, generally by making a call to a function named `startAudioContext`. Basic configurations like choice of sample rate are specified after which it will load **webchuck.wasm**, instantiate the AudioContext and then use the instance's `audioWorklet.addModule` function to bring up the worklet specified by the **webchuck.js** interface. As stated earlier, all of the Chuck language is provided by Webchuck and a global runnable instance of it results from a call to `createAChuck`. The audio output of the returned worklet instance gets connected to the browser's AudioContext and system audio output device in WAAPI fashion:

```
theChuck.connect( audioContext.destination );
```

Likewise, the system audio input device can be wired to the input of `theChuck` using a MediaStream [15].

```
     theADC = audioContext.createMediaStreamSource( stream );
          theADC.connect( theChuck );
```

A couple of provisos related to the above must be mentioned. The complete operation for connecting input involves `navigator.mediaDevices.getUserMedia(constraints)` which will trigger the browser to ask for permission to use the chosen audio input device, for example, a built-in microphone or an audio interface. Browsers by default use three stages of DSP on the input audio signal which typically would want to be turned off in a Webchuck application. These constitute the `constraints` shown below.

```
  const constraints = {'video': false, 'audio': {
        'echoCancellation': false,
        'autoGainControl': false,
        'noiseSuppression': false,
    }};
    var theADC = null;
    var connectAudioInput = function() {
      navigator.mediaDevices.getUserMedia(constraints)
        .then( function( stream ) {
          theADC = audioContext.createMediaStreamSource( stream );
          theADC.connect( theChuck );
        });
    }
```

A bare Webchuck application has only three dependencies and these are the three files already mentioned: **webchuck_host.js**, **webchuck.wasm** and **webchuck.js** (approximately 7 MB in total). According to browser convention, these three auxiliary files (and any other auxiliary files, such as **.wav** files or data sets) should be fetched from a server. In practice, a server can be a remote, institutionally provided server (as in the case of CCRMA's Music 220a class) or via a local proxy server system such as the Python SimpleHTTPServer [16]. The latter is convenient for local testing, especially if there is no network available.

Keeping the system as simple as possible has advantages for deployment. A Webchuck application is just a URL pointing to an **.html** file. The file itself launches with a click on the URL from anywhere, for example, from another web page, local file browser, Zoom chat, email, etc. Depending on the need it will be on a server and / or on the local file system. For the latter, the file will not need a proxy server to run as long as its dependencies are accessible from a server. Except for the dependencies, the application created by the **.html** file.is fully self-contained. The file itself is a bit of a Frankenstein with a mix of languages: **.css** and **.html** for building the page, **.js** for loading the dependencies and WAAPI parts, and Chuck for the Chuck program that will be run. All of the parts are applied in conventional ways and easily debugged using built-in browser debugger and inspector tools.

Webchuck in Musical Performance and in an IDE

Demonstrating the capabilities of Webchuck for musical performance is still in its early stages at the time of this writing. A quick way to get a taste of the potential is to load up several browser tabs with different Chuck examples and perform them as independent voices in sequence and in polyphony. One such demonstration is shown at the beginning of a video lecture on Webchuck [17]. The examples shown are Unclap.ck, THX.ck and StifKarp.ck taken from the Chuck repository [18].

A browser-based IDE for Webchuck is under development which is becoming a live performance studio. A “shred” (process thread) is a program which can be sporked (forked) at any time and multiple instances and shreds with different behaviors underlie Webchuck musical performance. The IDE is an open sketchpad for populating shreds and provides a ready menu of code snippets from Chuck's repository of examples. The IDE adds features for saving and restoring edits and a viewer showing currently running shreds, as well as signal visualization and graphical user interface (GUI) objects. It combines the capabilities of a traditional language-specific IDE with Chuck's own miniAudicle application [19] that was designed with shreds in mind. Having an editor to manipulate and spork shred code enables instant access to the full power of the language and turns it into a stage for live performance.

Features for the Future

Browser concurrency capabilities allow multiple Webchuck instances to run in separate tabs and inter-shred communication is a part of the Chuck language. Putting the two together suggests the potentially attractive feature of implementing Chuck's inter-shred signaling between browser tabs. One possible approach borrows from a method for transmitting uncompressed audio through WebRTC [20]. An argument in favor of considering such a mechanism is that it will allow better use of multi-core hardware. Chuck itself is a single-threaded application and that puts an upper bound on processor resources available to any one instance. The limitation can be countered by running multiple Webchuck instances across the browser tabs. The new feature would at first allow them to share audio signals, and then ultimately share Chuck events and states.

Native Chuck incorporates loadable plugin modules called Chugins [21]. These extensions get loaded at run time and are convenient for adding functionality beyond what's already built in to the language. In particular, new UGens appear in Chugin form. As a potential feature for Webchuck this would bring to it a large number of existing contributions to the language.

Native Chuck can run faster than real time if it is started from the command line with the "silent" option. WAAPI's `OfflineAudioContext` mode makes it possible to eventually implement the same feature in Webchuck.

A native Chuck feature that is unlikely to be replicated in the browser is scripting of arbitrary system calls. When run from the command line with the "caution-to-the-wind" flag set, Chuck can be used to accomplish scripting the same way many programmers often use the bash scripting language.
A recently added feature allows capturing of arbitrary data to an internal memory blob. For example, JavaScript can query Chuck for the contents of an array and then write the blob to a local file. The mechanism works similarly to the way audio capture to file has been implemented.

One, single, massive **.html** file with everything baked is an interesting prospect not yet pursued. To make it work, the contents of all system files will need to be present in the single **.html** file making it extremely clumsy to edit but not necessarily impractical for download (7 MB is not that much these days). One way it may work is with a "deploy" step that grafts on the larger portions after edits are saved, possibly from the new IDE, another is possibly with a "smart" editor which knows which chunks are customizable.

Conclusion

If its first 18 months are any indication, Webchuck is an exciting development for music making of all kinds. Teaching just became a lot more fun for CCRMA's Music 220a course and it's already proving its value for teaching in remote and / or asynchronous settings. The idea of putting the student in a hands-on, exploratory environment for experimentation with musical constructions, improvisation algorithms, sound synthesis and processing is something that's reshaping approaches. The "suggested modifications" editor blocks shown above are a start but we imagine something even more effective, where a student's imagination crafts the examples directly instead of from pre-made templates.

That's what we do when creating new applications and new pieces and performances. Webchuck helps the imagination flow.

References

[1] W3C Audio Working Group "Web Audio API W3C Recommendation," 17 Jun 2021 URL: https://www.w3.org/TR/webaudio/ (2021)

[2] H. Choi, "AudioWorklet: The Future of Web Audio," *Proc. Intl. Computer Music Conf.* (2018)

[3] G. Wang and P. R. Cook "Chuck: A Concurrent, On-the-fly Audio Programming Language," *Proc. of the Intl. Computer Music Conf.* (2003)

[4] G. Wang, P. R. Cook, and S. Salazar "ChucK: A Strongly-timed Computer Music Language," *Computer Music J.* 39(4):10-29 (2015)

[5] B. Schottstaedt "Pla: A Composer's Idea of a Language," *Computer Music J.* vol.7, no. 1, pp. 11-20 (1983)

[6] H. Taube "An Introduction to Common Music," *Computer Music J.* vol. 21, no. 1, pp. 29-34 (1997)

[7] C. Chafe "Browser-based Sonification," *Proc. of the 17th Linux Audio Conf.* CCRMA, Stanford University (2019)

[8] CCRMA "What is WebChucK?" 15 Mar 2023 URL: https://github.com/ccrma/webchuck (2023)
[9] A. Barate' and L. A. Ludovico, "Web MIDI API: State of the Art and Future Perspectives," *J. Audio Eng. Soc.* Vol. 70, no. 11, pp. 918–925 (2022)

[10] T. Kluyver, et al. "Jupyter Notebooks—a Publishing Format for Reproducible Computational Workflows," *Proc. of the 20th Intl. Conf. on Electronic Publishing (*2016)

[11] A. Walker "What Is an IDE? How It Helps Developers Code Faster," 3 Dec 2021 URL: https://www.g2.com/articles/ide (2021)

[12] Rik Cabanier, et al. “HTML Canvas 2D Context,” 28 Jan 2021 URL: https://www.w3.org/TR/2dcontext/ (2021)

[13] P. Beshai "Smoothly Animate Thousands of Points with HTML5 Canvas and D3," 16 Mar 2017 URL: https://bocoup.com/blog/smoothly-animate-thousands-of-points-with-html5-canvas-and-d3 (2017)

[14] NOAA Global Monitoring Laboratory "Trends in Atmospheric Carbon Dioxide," 15 Mar 2023 URL: https://gml.noaa.gov/ccgg/trends/ (2023)

[15] M. Casas-Sanchez, J. Barnett, and T. Leithead "MediaStream Recording," 16 Feb 2021 URL: https://www.w3.org/TR/mediastream-recording/ (2021)

[16] M. Hisamuddin "Tech Tip: Really Simple HTTP Server with Python," *Linux J.* 22 Sep 2009 URL: https://www.linuxjournal.com/content/tech-tip-really-simple-http-server-python (009)

[17] C. Chafe "Webchuck Examples," 4 Dec 2022 URL: "https://ccrma.stanford.edu/~cc/220a/webchuckChrisChafe.mp4" (2022)

[18] G. Wang, et al. "Chuck Examples," 15 Mar 2023 URL: https://chuck.stanford.edu/doc/examples/ (2023)

[19] S. Salazar, G. Wang, and P. R. Cook, Perry "miniAudicle and ChucK Shell: New Interfaces for ChucK Development and Performance," *Proc. Intl. Computer Music Conf. (*2006)

[20] M. Sacchetto, A. Servetti, and C. Chafe “JackTrip-WebRTC: Networked Music Experiments with PCM Stereo Audio in a Web Browser,” *Proc. of the Intl. Web Audio Conf.* (2021)

[21] S. Salazar and G. Wang "Chugins, Chuggraphs, and Chugens: 3 Tiers for Extending ChucK.," *Proc. of the Intl. Computer Music Conf.* (2012)