BROWSER-BASED SONIFICATION

Chris Chafe

Center for Computer Research in Music and Acoustics (CCRMA)
Stanford University, USA
cc@ccrma.stanford.edu

ABSTRACT

TimeWorkers is a programming framework for coding sonification projects in JavaScript using the Web Audio API. It is being used for sonification workshops with scientists, doctors, and others to facilitate ease of use and cross-platform deployment. Only a browser and text editor are needed. Using Free and Open-source Software (FOSS) the system can run standalone since No Internet is Required for Development (NIRD). Workshop participants rapidly master principles of sonification through examples and are encouraged to bring their own datasets. All mapping code is contained in a project's .html landing page. A single generator function iterates over the project's data series and provides a fine-grained interface to time-varying sound parameters. This loop and its internals are patterned after similar constructions in the Chuck language used by the author in earlier sonification tutorials.

1. INTRODUCTION

Sonification shares much with other kinds of computer music making including the wide range of programming tools which can be used. Sonification also shares in the kinds of decisions found in photography and soundscape recording. Gathering, selecting, framing and contrast enhancement are a part of working with material from the (outside of music) outside world. On the other hand, another key part of creating a sonfication, mapping, has affinities with algorithmic composition. TimeWorkers is a browser-based software framework described in this paper which, while not limited to sonification, provides in it's initial rollout functional support for decisions specific to such work.

Specialized programming languages have evolved and continue to evolve which are custom-designed to express musical relationships, especially timing and concurrency. I've used several over the course of composing computer music with succeeding generations of hardware platforms, for example, Pla[1], MIDILisp[2], Common Music[3] and Chuck[4], all of which are examples of computer music languages with ways of programmatically expressing organization of sound in time.

TimeWorkers is written in JavaScript and provides a readily available computation environment for my sonification workshops. To give a glimpse of what will be explained later in detail, the name comes from its use of the Web Worker API[5] for composing musical layers or voices which unfold in time. The software uses browsers' existing means for sound generation, in this case the built-in computer music capabilities of the Web Audio API[6]. The added functionality provided by TimeWorkers provides ways to compose higher-level aspects of musical timing and texture.

Stepping back for a moment, it's worth reflecting on how computers and music have been mingling their intimate secrets for over 50 years. These two worlds evolve in tandem and where they intersect they spawn practices that are entirely novel. One of these is sonification, the practice of turning raw data into sounds and sonic streams to discover new relationships within the dataset by listening with a musical ear. This is similar to exploring data visualization with strategies made for the eye to reveal new insights from data using graphs or animations. A key advantage with sonification is sound's ability to present trends and details simultaneously at multiple time scales, allowing us to absorb and integrate this information the same way we listen to music.

Kramer, et al.'s prescient *Sonification Report* [7] (2010) merits quoting here at length and will be revisited in the conclusion section. The paper identified "three major issues in the tool development area that must be tackled to create appropriate synthesis tools developed for use by interdisciplinary sonification researchers." The TimeWorkers framework addresses some (but not all) of the following points.

"Portability: Sonification scale places demands on audio hardware, on signal processing and sound synthesis software, and on computer operating systems. These demands may be more stringent than the requirements for consumer multimedia. Researchers dealing with problems that go beyond the limits of one system should be able to easily move their sonification data and tools onto a more powerful system. Thus, tools must be consistent, reliable, and portable across various computer platforms. Similarly, tools should be capable of moving flexibly between real- time and nonreal-time sound production."

"Flexibility: We need to develop synthesis controls that are specific and sophisticated enough to shape sounds in ways that take advantage of new findings from perceptual research on complex sounds and multimodal displays and that suit the data being sonified. In addition to flexibility of synthesis techniques, simple controls for altering the data-to-sound mappings or other aspects of the sonification design are also necessary. However, there should be simple 'default' methods of sonification that allow novices to sonify their data quick and easily."

"Integrability: Tools are needed that afford easy connections to visualization programs, spreadsheets, laboratory equipment, and so forth. Combined with the need for portability, this requirement suggests that we need a standardized software layer that is integrated with data input, sound synthesis, and mapping software and that facilitates the evaluation of displays from perceptual and human factors standpoints."

2. USING THE FRAMEWORK

Meant to be very hands-on, my 2-hour workshops ask the participants to bring their own laptop and headphones. I first take them through a simple example which has a been an early "etude" assignment in my course, "*Computer Music Fundamentals* "[8], taught at Stanford's CCRMA. The goal is to get students to start working with their own datasets as soon as possible and get them exploring a range

of sonifications through experimentation.

A dataset to play with can be scouted out by searching the web and copied or exported from a spreadsheet or other format. For starters, it's simply a single column of numbers in plain text. The range of values doesn't matter because it will be automatically rescaled when read by the framework's file input layer. In my own development work, examples and code repository are all linux-based and other operating systems work equally well.

2.1. Basic Sonfication How-to

2.1.1. *What you'll need*

The browser can be a recent version of Firefox, Chromium, Chrome, or Edge. A simple text editor like Gedit is all that's required for developing the code and preparing an ASCII data file.

2.1.2. *Testing the demo*

Open the demo URL `https://ccrma.stanford.edu/~cc/sonify` to see a page that looks like Figure 1. There's a default time series "*tides.dat*" that can be played by clicking on the demo icon (the small globe is a button).

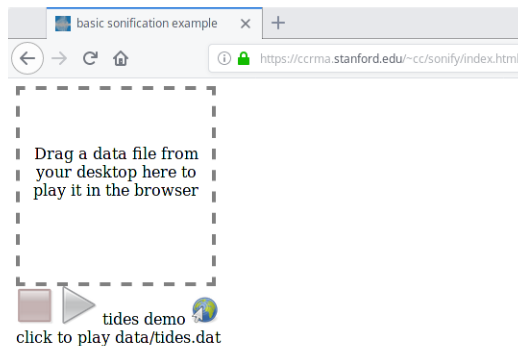

Figure 1: *An example page with options for playing a default time series or dragging in a data file.*

Alternatively, a data file can be dragged from the desktop onto the page to sound it with the same preset sonification parameters.

The demo was created by Chris Hartley, a biologist who participated in the first workshop (in 2016) at the University of British Columbia. In it, "*You can hear the rising and then falling chirp-chirp-chirp of the major high tides, which get highest at the new and full moons, and then the slightly lower trill of two roughly equal high tides per day, which occurs during the quarter moons.*" Hartley's sonfication plays a year's worth of tidal data at a fast rate using a sine tone.

After starting the demo or after loading a data file the stop and play buttons on the web page become activated, Figure 2.

2.1.3. *Modifying the demo*

To practice modifying the demo, a good first goal is to make the rate of running through the data much slower. To accomplish this, we'll make a local copy of the demo, test it and then edit it.

Go to its repository `https://cm-gitlab.stanford.edu/cc/sonify` and download a snapshot. The downloaded .zip file

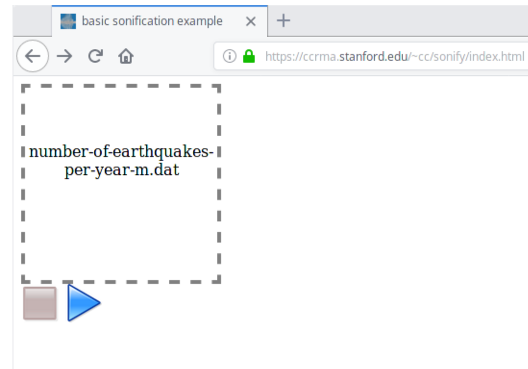

Figure 2: *Stop and play buttons become activated after starting the demo or dragging in a data file.*

will have a long name that depends on the version. Extract the contents of the .zip file and open its index.html file in a browser (use Firefox because it will allow the demo to run as a local file without manual intervention).

This will allow you to test the local copy of the landing page in a browser and make sure it's working identically to the version on the workshop's web server. If it's all good, then the local copy of the landing page can be opened in a text editor. Search for the line

`let dur = 0.005`

and assign a new value, for example:

```
function* sonify(data) {
  let dur = 0.05
// duration between data points in seconds
```

Save the modification in the text editor and then refresh the browser page to load the changed file. The example can then be played as before but the rate will now be $10x$ slower.

Further modifications are quickly explored with the same work flow of edit-save-refresh-play. For example, in the mapping function

`map(v)`

where, for a given value of v, sound parameters are determined for pitch and loudness (respectively, kn in MIDI key number units and db in a decibel range from -100 to 0). These in turn are used to calculate values which will be applied to the sine tone's frequency (Hz) and amplitude (range 0.0 to 1.0):

```
function map(v) {
  let kn = 60 + v * 40
  let f = mtof(kn)
  let db = -30 + v * 10
  let a = dbtolin(db)
  return {pit: f, amp: a}
}
```

`map(v)` returns pitch frequency and loudness amplitude in an object created by an object initializer. Its argument, v, is expected to lie in the range 0.0 to 1.0. In a hidden step which happens when the data is loaded, the data series has been automatically normalized to this range. `map(v)` is set so that the lowest data value will be sounded at Middle-C (MIDI key number 60) and the highest will be 3 Octaves and a Major Third above. Intermediate values will be linearly interpolated across key number values (using fractional quantities, in other words, not quantized to integer key numbers). Code for

the utility functions mtof and dbtolin, respectively for conversion from MIDI key number to frequency in Hz and dB loudness to amplitude, have been borrowed from Hongchan Choi's Web Audio API Extension (WAAX) project [9].

The sonify generator function sets a new target pitch when processing each new data value and starts a glissando (a smooth frequency ramp) to reach the target pitch in the length of time specified by the data update period, dur. The ramp is a linear function which updates the sine tone's frequency each audio sample. Amplitude is smoothly modulated in the same way.

The complete sonify generator function for this example is listed below and includes a definition of the sound source along with a mechanism for applying updates to its parameters. The new function Sin(timeWorker) instantiates a SinOsc and several methods which start the oscillator, apply parameter updates to it and stop it. After instantiation as a local object s, it is initiated with the first values from the mapping function and a gain of 0. Ramps are set in motion and the process pauses until they reach their targets with yield dur after which the loop continues and cyclically churns through each data point until all have been "performed." The last few lines ramp the oscillator to 0 and then stop and finish.

```
function* sonify(data) {
  let dur = 0.005
  let datum = data.next()
  function map(v) {
    let kn = 60 + v * 40
    let f = mtof(kn)
    let db = -30 + v * 10
    let a = dbtolin(db)
    return {pit: f, amp: a}
  }
  function Sin(timeWorker) {
    let s = new SinOsc(timeWorker)
    s.start()
    this.setPit = function(freq) { s.freq(
        freq ) }
    this.setAmp  = function(gain) { s.gain(
        gain ) }
    this.rampPit = function(freq,dur) { s.
        freqTarget( freq,dur ) }
    this.rampAmp = function(gain,dur) { s.
        gainTarget( gain,dur ) }
    this.stop = function() { s.stop() }
    this.ramps = function (f,a,d) {
      this.rampPit(f,d)
      this.rampAmp(a,d)
    }
  }
  let sin = new Sin(this)
  if (withFFT) postMessage("makeFFT()")
  let params = map(datum.value)
  sin.setPit(params.pit)
  sin.setAmp(0)
  while (!datum.done) {
    sin.ramps(params.pit, params.amp, dur)
    yield dur
    if (withSliderDisplay) postMessage("
        move1D("+datum.value+")")
    if (withChart) postMessage("move2D()")
    datum = data.next()
    params = map(datum.value)
  }
  sin.rampAmp(0,0.1)
  yield 0.1
  sin.stop()
  postMessage("finish()")
}
```

Workshop discussions are mostly focused on customizing the above code and demonstrating extensions described later in this report. What follows in the next section is a discussion of the TimeWorkers framework "under the hood." This can be skipped if one's main interest is in customizing sonifications rather than digging into the underlying system.

3. PROGRAMMING STRUCTURE AND SUPPORTING FUNCTIONS

The framework has no dependencies. It is a lightweight project which is Free Open-source Software (FOSS) and has the additional feature of No Internet Required for Development (NIRD). Workshops and individual work are equally possible online and offline, for example, during field work with no connectivity. A project's .html landing page loads a single associated script file, engine.js, which contains all supporting functions. Files and modules are shown schematically in Figure 3.

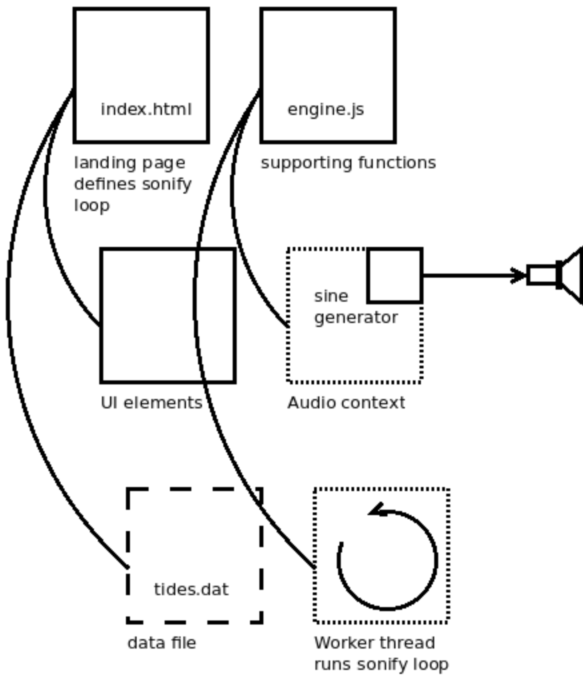

Figure 3: *Structure and modules.*

The project landing page sets up web-related configurations, specifies the user interface (UI), loads the script file, engine.js, and is where the sonification is "composed." Various "hardwired" globals need to be declared which will be communicated to the script file, including a default value for dataFileName. Likwise, the script file expects a "hardwired" generator function with the name sonify (which should be defined using JavaScript's function* syntax [10]).

Table 1: project files

web landing page	supporting script
index.html	**engine.js**

Table 2: **index.html** elements

<head>	<body>	<script>
<meta> specifies metadata	configures UI elements	sets global and local variables
(optional) <script> loads any auxiliary script files	(options to hide or expose)	loads **engine.js**
e.g., graphing library	e.g., drag and drop	must define function* sonify(data)

Table 3: **engine.js** tasks, **classes** (and optional functionality)

set locals	polish UI	specify web worker(s)	set up spork mechanism	define DSP ugens
audio context	check browser capabilities	**WorkerThread**	**TimeIterator**	e.g., **SinOsc**
data source	get UI elements	uses inline definitions	play / stop	e.g., **FM**
timing cushion	set UI element states	(add graphing capability)	nextEventAt	uses setValueAtTime,
worker arrays	(add drag and drop)	(connect real-time UI elements)	uses async / await	linearRampToValueAtTime

This function instantiates any unit generators (ugens) it will be using, for example with

```
new SinOsc(timeWorker)
```

as shown above, and specifies data-to-sound parameter mappings which unfold through time.

For brevity's sake the script file, engine.js, is not reproduced here but can be found in `js/` subdirectory of the project repository[11]. This script provides the TimeWorkers structure through its class definitions, functions and own variable settings. Any special tokens which are referenced by the sonify generator function, e.g. `SinOsc` will be resolved against what is defined or declared in the global scope after engine.js has been loaded.

The script file contains several parts. Setting local variables, polishing the UI and a system for "performing" sonifications composed with the sonify generator function.

A *WorkerThread* interface sets up and runs this time-sensitive apparatus in separate threads. The *TimeIterator* class provides a mechanism which waits between events in the sonify generator's loop and compensates for timing jitter. It uses the `performance.now()` clock to compare real time with expected logical time. Finally, the ugen part of the script file defines any synthesis or DSP patches which are used.

```
var context
```

is declared to hold the *window.AudioContext* which gets instantiated at sound start and closed at sound stop,

```
var workerThreads = []
```

is the array containing the pool of `WorkerThread` instances and

```
var uwta = []
```

is a multi-dimensional array (whose name is shorthand for "*ugenWorkerThreadsArrays*") that contains the set of all ugens in all WorkerThreads.

A programming pattern often used in sonification in the Chuck language [4] has two aspects. The first is the `spork` function which calls a given function in a parallel, separate thread with its own logical timebase. (A child process spawned by a sporked function can also spork its own child processes.) The second construct is a means for looping over data, in Chuck this is usually a `while` loop where event time advances each iteration. The loop executes in its own thread. The present framework supports both features using its WorkerThread and TimeIterator constructs.

When `makeWorkerThread` (Table 1) creates a new instance, the spawned JavaScript Worker [12] is of a special inline type (as opposed to the more common type which is usually created by loading a dedicated script file).

```
var blob = new Blob([script])
var worker = new Worker(URL.createObjectURL(
    blob))
```

The script passed into the new Blob sets up a mechanism for dynamic object definition. It calls `addEventListener` on the new worker and sets how the worker will handle incoming messages. By telling it to handle them with an `eval` (in the global scope), the worker's set of variables and functions is literally "grown" by posting message strings to be evaluated which contain the desired definitions and settings. One of these, for example, is the sonify function defined back in the landing page. Dynamically defining timeWorkers in this way allows the sonify function to also spork processes which will become its own new child workers each of which runs in a separate thread.

The `spork` function itself instantiates a time-sensitive data iterator with `makeTimeIterator`. A TimeIterator will pause a generator for a given duration with its method `nextEventAt()` which is an async function utilizing JavaScript's async / await ([13]) pausing functionality. When sporked, a sonify generator's loop is started with `nextEventAt("start")` that executes its first cycle. A subsequent `yield` in the sonification loop will set the amount of time to pause on the next call to `nextEventAt` (which calls itself recursively) and the loop continues.

In the definition below, `fstar` is the sonify generator defined in the landing page and `args` contains a data iterator with the provided data series (which has had its range normalized).

```
function spork(fstar, ...args) {
  let ti = makeTimeIterator()
  ti.sporkScript = fstar.apply( ti, args )
  ti.nextEventAt("start")
}
```

To reiterate, calling `spork` with both a sonify generator and a TimeIterator containing the data as shown

```
spork(sonify,data)
```

will create a pattern comparable to a Chuck-based sonification which consists of essentially the same parts: spork a new thread which sets up a sound source and mapping strategy, and then loops through a conditioned data series, pausing after each data point.

In Chuck, pausing is written using the syntax

```
dur => now;
```

whereas the TimeWorkers equivalent uses

```
yield dur
```

A `yield` in the sonify generator loop invokes a JavaScript `Promise` in the TimeIterator object whose `setTimeout` is set to the duration to `await`.

3.1. SinOsc ugen example

Custom ugens comprise patch definitions made with the Web Audio API's audio nodes. The `makeSinOsc` example shown here instantiates an oscillator with gain control using the API's createOscillator() and createGain() methods[6].

```
function makeSinOsc()
{
  let o = context.createOscillator()
  let g = context.createGain()
  o.type = "sine"
  o.frequency.value = 440
  g.gain.value = 0.1
  o.connect(g)
  g.connect(context.destination)
  g.connect(dac)
  return { osc:o, gain:g }
}
```

The object gets instantiated in a wrapper called `SinOsc` which when instantiated itself with `new` also includes methods to alter its parameters, for example, by changing its frequency with the following custom `freq()` method:

```
    freq: function (hz) {
        let n = this.dsp
        postMessage(ugens+"["+n+"].osc.
            frequency.setValueAtTime("+hz+",
            "+(myThread.now+cushion)+")")
      }
```

`this.dsp` refers to the ugen itself which is held in the main thread's array `ugens[]`. The message posted to the main thread looks up the osc field of the ugen and changes its frequency using the Web Audio API's `setValueAtTime` (which corresponds to the worker thread's "now" plus a constant offset). A full ugen definition comprises instantaneous setters for all parameters, as well as custom time-varying envelopes, for example made with the Web Audio API's `linearRampToValueAtTime`. Note that the patch code also includes a connection from the patch's summing point to a global summing point called `dac`.

Different sound sources can be made available by expanding the library of ugens defined in engine.js. Each would comprise a "make the patch" portion and a wrapper (with the ugen name) which includes the set of parameter altering methods.

3.2. FM patch

For example, a simple two-oscillator FM patch could look like the following:

```
function makeFM()
{
  let mod = context.createOscillator()
  let modGain = context.createGain()
  mod.type = "sine"
  mod.connect(modGain)
  let car = context.createOscillator()
  let g = context.createGain()
  car.type = "sine"
  modGain.connect(car.frequency)
  car.connect(g)
  g.connect(context.destination)
  g.connect(dac)
  let cFreq = 2200
  let index = 33
  let mRatio = .1
  modGain.gain.value = cFreq * index
  mod.frequency.value = cFreq * mRatio
  car.frequency.value = cFreq
  g.gain.value = 0.1
  return { osc:car, gain:g, mod:mod, modGain:
     modGain }
}
```

All ugens need to be accessible in the timeWorker thread in which the sonify loop is running. A last step, then, in ugen creation is to add the ugen wrapper, for example `FM`, to the list of functions which gets dynamically installed inline when a new WorkerThread is instantiated.

4. EXTENSIONS

Changing the sound source, sounding multiple time series and adding graphing capabilities are extensions which complement the basic example described above 2.

4.1. Voicing

Changing to a more interesting sound source is possible in the sonify generator itself. This approach relies on combinations of ugens defined in the engine.js script. Where the basic example uses a single `SinOsc` ugen as its instrument, the example here demonstrates additive synthesis built by summing multiple sines which are harmonically tuned. The new instrument `Harmonics` is defined directy within the sonify generator.

```
function Harmonics(nSins,timeWorker) {
  this.sins = new Array
  for (let i = 0; i < nSins; i++) this.sins.
     push( new SinOsc(timeWorker) )
  this.sins.forEach(function(x) { x.start()
     })
  function fi(f,i) { return f*(i+1) }
  function ai(a,i) { let h = (i+1); let odd =
      (h%2) ? a : a*0.1; return odd/h }
  this.setPitch = function(freq) { this.sins.
     forEach(function(x,i) {x.freq( fi(freq,
     i) )}) }
```

```
  this.setGains = function(gain) { this.sins.
      forEach(function(x,i) {x.gain( ai(gain,
      i) )}) }
  this.freqTarget = function(freq,dur) { this
      .sins.forEach(function(x,i) {x.
      freqTarget( fi(freq,i),dur) }) }
  this.gainTarget = function(gain,dur) { this
      .sins.forEach(function(x,i) {x.
      gainTarget( ai(gain,i),dur) }) }
  this.stop = function() { this.sins.forEach(
      function(x) {x.stop()}) }
  this.ramps = function (f,a,d) {
    this.freqTarget(f,d)
    this.gainTarget(a,d)
  }
}
```

One of these instruments is then instantiated in the sonfication loop, for example, with

```
let vox = new Harmonics(8,this)
```

to create an harmonic series of 8 SinOscs. Given a pitch frequency f `function fi(f,i)` sets their tunings. Amplitude relationships in `function ai(a,i)` create a clarinet-like structure favoring odd harmonics. A convenience function `ramps` is provided which applies frequency and amplitude updates to the entire additive synthesis patch.

The following set of extensions are turned on or off with flags in the index.html file. By default, the withDemo flag is set. Only one option is allowed at a time, so remember to set

```
    withDemo = 0
```

before exploring these others.

4.2. Polyphony from multiple data series

Multiple time series are interesting to sonify at the same time, for example, to hear correlations by ear. Data can be input from two or more separate data files as in this example which combines monthly USA gross domestic product (GDP) from 1969 to 2016 and global CO_2 level for the same period. The curves shown in Figure 4 have been normalized to the same range.

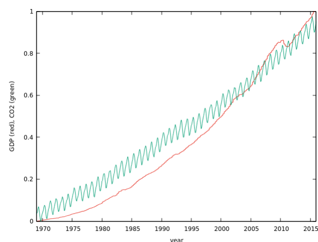

Figure 4: *GDP and CO_2.*

The example landing page, index.html, has a provision for hearing these two playing together, as two independent musical voices. Change the state of `withDemo` and this flag for this to take effect:

```
    withTwoFiles = 1
```

Two data files will now be specified and will spawn two TimeWorker threads both using the single sonify generator as defined. In this example, one can hear details like the 2008 financial downturn and the seasonal flux in global CO_2. Overall, the two quantities follow a coincident rising trend.

4.3. Animated Chart

Similar to the interest in multi-modal data presentation described in [14], sonification in the present framework can be combined with graphing. Chart.js is a FOSS project for interactive plotting in the browser and is integrated into the project by loading a single script file (which can be locally sourced for creating a NIRD environment).

Again, the example landing page, index.html, has a provision for demonstrating this extension by changing `withDemo` and this flag:

```
    withChart = 1
```

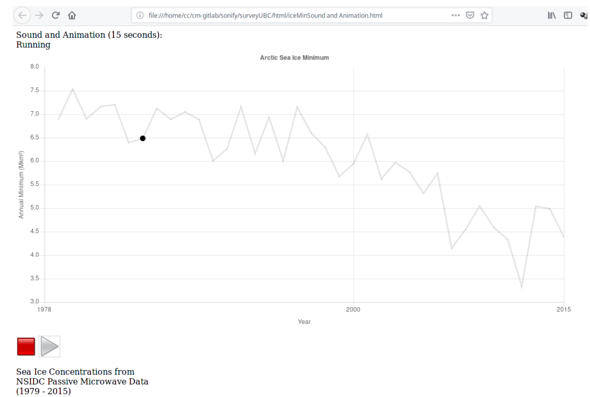

Figure 5: *Simultaneous sound and graph of Arctic Sea Ice Minimum per year.*

Playing the sonification in Figure 5 animates the black dot on the curve. Syncronized sound and animation is accomplished with `postMessage("moveGraph()")` inside the loop in the sonify generator. Each successive call advances the black dot to the next data point in an array of 2D data points that was input from a multi-column data file (columns are year and value).

4.4. Real-time FFT display

Likewise, change `withDemo` and the following flag in the example landing page, index.html, and the sonification's audio output will be displayed as a time-varying spectrum.

```
    withFFT = 1
```

An FFT analyzer computes the spectrum of the global summing point `dac` in real time.

5. CONCLUSIONS

A 40+ year tradition has evolved a well-known pattern for sequencing scores and real-time synthesis in languages like Pla[1], Common Music[3], Chuck[4] and others. The sonify generator's loop is a descendant written in JavaScript. Running in the browser, it allows flexible programming using the full power of the language and can be rapidly experimented with on any browser-equipped system.

Table 4: TimeWorkers framework in terms of goals suggested by *Sonification Report* [7]

attribute	goal	now	soon	never
Portability	consistent		x	
Portability	reliable	x		
Portability	portable across various computer platforms	x		
Portability	moving between real-time and non-real-time sound production		x	
Flexibility	simple controls for altering the data-to-sound mappings	x		
Flexibility	simple "default" methods of sonification that allow novices to sonify their data quick and easily	x		
Integrability	easy connections to visualization	x		
Integrability	easy connections to visualization programs, spreadsheets, laboratory equipment			?
Integrability	standardized software layer			?

Sonifications created using the framwork run equally well on mobile and other smaller systems.

Pla's *voices* are analogous to sonify generator loops because they constitute groups of time-ordered events which can themselves be voices (recall that spork-ed child threads can spork their own children). Other pertinent features of Pla also have bearing on the present framework (these are distilled a 1983 description): "Higher levels of musical control are implemented as *voices* and *sections* ..." "...notes that somehow belong together are grouped under the rubric of a voice." "Arbitrarily large groups of voices can be organized into a *section,* which then becomes nearly equivalent to a voice." "Another kind of grouping is based on voices... voices can create other voices to any level of nesting."

Common Music's similar features involve multiple types: "*Thread – A collection that represents sequential aggregation.* A single timeline of events is produced by processing substructure in sequential, depth-first order." "*Merge – A collection that represents parallel aggregation, or multiple timelines.* A single timeline of events is producted by processing substructure in a scheduling queue." "*Algorithm – A collection that represents programmatic description.* Instead of maintaining explicit substructure, a single timeline of events is produced by calling a user-specified program to create new events."

The TimeWorkers framework described here offers a way to construct the above relationships in browser-based platforms and offers solutions for some, but not all of the goals cited in *Sonification Report* [7]. Table 4 lists the boxes it checks off.

In the future, faster-than-sound soundfile writing will be directly supported though for now, file output is only by browser sound capture plug-ins (which run in real time). Faster-than-sound is a highly-desirable feature and is something that's been supported in both Common Music and Chuck. Regarding the former, "Realization in Common Music can occur in one of two possible modes: run time and real time. In run-time mode, realized events receive their proper 'performance time stamp,' but the performance clock runs as fast as possible. In real-time mode, realized events are stamped at their appropriate real-world clock time." For the latter, Chuck's "silent mode" is the equivalent.

The recently standardized AudioWorklet [15][1] will be integrated into the framework in the coming months. Of particular interest is another recently proposed enhancement to Web Audio to support multi-channel output.

Also for the future, direct real-time sonification from live sensor data can be contemplated. This important feature opens up applications such as bio-feedback [16] or other kinds of feedback such as providing real-time "cracking" sounds to operators of fracking pumps (where presently feedback is provided after the fact and one can imagine the problems resulting from the over-stimulation of shale gas wells). It has become vital in medical applications, even making inroads on traditional treatment practices in cases where listening to data provides equal or better sensitivity and specificity compared to visual means. The brain stethoscope, for example, allows rapid detection of non-convulsive seizures by non-specialists. [17]

Interest in sonification is burgeoning as sensors and data collections become an increasingly ubiquitous part of daily life. Employing well-known sound generation techniques from computer music, sonification can play a role in the work of domain experts and students in sciences and arts, as well as for general communication.

[1] As of this writing, only the Chromium browser family supports AudioWorklet. It is expected soon in Firefox at which point the integration work will commence.

6. REFERENCES

[1] Bill Schottstaedt, "Pla: A composer's idea of a language," *Computer Music Journal*, vol. 7, no. 1, pp. 11–20, 1983.

[2] David Wessel, Pierre Lavoie, Lee Boynton, and Yann Orlarey, "Midi-lisp: A lisp-based programming environment for midi on the macintosh," in *Audio Engineering Society Conference: 5th International Conference: Music and Digital Technology*, May 1987 (accessed February 2, 2019), http://www.aes.org/e-lib/browse.cfm?elib=4659.

[3] Heinrich Taube, "An introduction to common music," *Computer Music Journal*, vol. 21, no. 1, pp. 29–34, 1997.

[4] Ge Wang, Perry R. Cook, and Spencer Salazar, "Chuck: A strongly timed computer music language," *Computer Music Journal*, vol. 39, no. 4, pp. 10–29, 2015.

[5] Moz://a MDN web docs, *Using Web Workers*, 2019 (accessed February 6, 2019), https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

[6] Moz://a MDN web docs, *Web Audio API*, 2018 (accessed December 16, 2018), https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API.

[7] Bruce Walker Terri Bonebright Perry Cook Kramer, C. and John H. Flowers, *Sonification Report: Status of the Field and Research Agenda*, 2018 (accessed December 16, 2018), http://digitalcommons.unl.edu/psychfacpub?utm_source=digitalcommons.unl.edu%2Fpsychfacpub%2F444&utm_medium=PDF&utm_campaign=PDFCoverPages.

[8] Chris Chafe, *Music 220A*, 2019 (accessed February 6, 2019), https://ccrma.stanford.edu/courses/220a/.

[9] Honchan Choi, *Web Audio API eXtension*, 2019 (accessed January 28, 2019), http://hoch.github.io/WAAX/.

[10] Moz://a MDN web docs, *Iterators and generators*, 2018 (accessed December 16, 2018), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators.

[11] Chris Chafe, *project software repository*, 2018 (accessed December 16, 2018), https://cm-gitlab.stanford.edu/cc/sonify.

[12] Moz://a MDN web docs, *Worklet*, 2018 (accessed December 16, 2018), https://developer.mozilla.org/en-US/docs/Web/API/Worklet.

[13] Moz://a MDN web docs, *async function*, 2018 (accessed December 16, 2018), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.

[14] Tianchu (Alex); Tomlinson Brianna; Walker Bruce N. Kondak, Zachary; Liang, "Web sonification sandbox - an easy-to-use web application for sonifying data and equations," *Proceedings of 3rd Web Audio Conference*, 2017.

[15] Hongchan Choi, *Audio Worklet Design Pattern*, 2018 (accessed December 16, 2018), https://developers.google.com/web/updates/2018/06/audio-worklet-design-pattern.

[16] Jan-Torsten Milde Baumann, Christian and Johanna Friederike Baarlink, *Body Movement Sonification using the Web Audio API*, 2018 (accessed December 16, 2018), https://webaudioconf.com/demos-and-posters/body-movement-sonification-using-the-web-audio-api/.

[17] Josef Parvizi, Kapil Gururangan, Babak Razavi, and Chris Chafe, "Detecting silent seizures by their sound," *Epilepsia*, vol. 59, no. 4, pp. 877–884, 2018.