

Clean Rinse

Shampoo for your System

I don't know who the Web Audio API is designed for

Posted on [September 13, 2017](#)

WebGL is, all things considered, a pretty decent API. It's not a great API, but that's just because OpenGL is also not a great API. It gives you raw access to the GPU and is pretty low-level. For those intimidated by something so low-level, there are quite a few higher-level engines like [three.js](#) and [Unity](#) which are easier to work with. It's a good API with a tremendous amount of power, and it's the best portable abstraction we have for a good way to work with the GPU on the web.

HTML5 Canvas is, all things considered, a pretty decent API. It has plenty of warts: [lack of colorspace](#), you can't directly draw DOM elements to a canvas without [awkwardly porting it to an SVG](#), blurs are strangely hidden from the user into a "shadows" API, and a few other things. But it's honestly a good abstraction for drawing 2D shapes.

[Web Audio](#), conversely, is an API I do not understand. The scope of Web Audio is hopelessly huge, with features I can't imagine anybody using, core abstractions that are hopelessly expensive, and basic functionality basically missing. To quote the specification itself: "It is a goal of this specification to include the capabilities found in modern game audio engines as well as some of the mixing, processing, and filtering tasks that are found in modern desktop audio production applications."

I can't imagine any game engine or music production app that would want to use *any* of the advanced features of Web Audio. Something like the [DynamicsCompressorNode](#) is practically a joke: basic features from a real compressor are basically missing, and the behavior that is there is underspecified such that I can't even trust it to sound correct between browsers. More than likely, such filters would be written using asm.js or WebAssembly, or ran as Web Workers due to the rather stateless, input/output nature of DSPs. Math and tight loops like this aren't hard, and they aren't rocket science. It's the only way to ensure correct behavior.

For people that do want to do such things: compute our audio samples and then play it back, well, the APIs make it near impossible to do it in any performant way.

For those new to audio programming, with a traditional sound API, you have a buffer full of samples. The hardware speaker runs through these samples. When the API thinks it is about to run out, it goes to the program and asks for more. This is normally done through a data structure called a "ring buffer" where we have the speakers "chase" the samples the app is writing into the buffer. The gap between the "read pointer" and the "write pointer" speakers is important: too small and the speakers will run out if the system is overloaded, causing crackles and other artifacts, and too high and there's a noticeable lag in the audio.

There's also some details like how many of these samples we have per second, or the "sample rate". These days, there are two commonly used sample rates: 48000Hz, in use by most systems these days, and 44100Hz, which, while a bit of a strange number, rose in popularity due to its use in [CD Audio](#) (why 44100Hz for CDDA? Because Sony, one of the organizations involved with the CD, cribbed CDDA from an earlier digital audio project it had lying around, the [U-matic](#) tape). It's common to see the operating system have to convert to a different sample rate, or "resample" audio, at runtime.

Here's an example of a theoretical, non-Web Audio API, to compute and play a 440Hz sine wave.

```
1 const frequency = 440; // 440Hz A note.
2 // 1 channel (mono), 44100Hz sample rate.
3 const stream = window.audio.newStream(1, 44100);
4 stream.onfillsamples = function(samples) {
5   // The stream needs more samples!
6   const startTime = stream.currentTime; // Time in seconds.
7   for (var i = 0; i < samples.length; i++) {
8     const t = startTime + (i / stream.sampleRate);
9     // samples is an Int16Array
10    samples[i] = Math.sin(t * frequency) * 0x7FFF;
11  }
12 };
13 stream.play();
```

The above, however, is nearly impossible in the Web Audio API. Here is the closest equivalent I can make.

```
1 const frequency = 440;
2 const ctx = new AudioContext();
3 // Buffer size of 4096, 0 input channels, 1 output channel.
4 const scriptProcessorNode = ctx.createScriptProcessorNode(4096, 0, 1);
5 scriptProcessorNode.onaudioprocess = function(event) {
6   const startTime = ctx.currentTime;
7   const samples = event.outputBuffer.getChannelData(0);
8   for (var i = 0; i < 4096; i++) {
9     const t = startTime + (i / ctx.sampleRate);
10    // samples is a Float32Array
11    samples[i] = Math.sin(t * frequency);
12  }
13 };
14 // Route it to the main output.
15 scriptProcessorNode.connect(ctx.destination);
```

Seems similar enough, but there are some important distinctions. First, well, this is deprecated. Yep. [ScriptProcessorNode has been deprecated](#) in favor of Audio Workers since 2014. Audio Workers, by the way, don't exist. Before they were ever implemented in any browser, they were replaced by the [AudioWorklet](#) API, which doesn't have any implementation in browsers.

Second, the sample rate is global for the entire context. There is no way to get the browser to resample dynamically generated audio. Despite the browser requiring having fast resample code in C++, this isn't exposed to the user of ScriptProcessorNode. The sample rate of an AudioContext isn't defined to be 44100Hz or 48000Hz either, by the way. It's dependent on not just the browser, but also the operating system and hardware of the device. [Connecting to Bluetooth headphones](#) can cause the sample rate of an AudioContext to change, without warning.

So ScriptProcessorNode is a no go. There is, however, an API that lets us provide a differently sampled buffer and have the Web Audio API play it. This, however, isn't a "pull" approach where the browser fetches samples every once in a while, it's instead a "push" approach where we play a new buffer of audio every so often. This is known as BufferSourceNode, and it's what emscripten's SDL port uses to play audio. (they use to use ScriptProcessorNode but then [removed it because it didn't work good, consistently](#))

Let's try using BufferSourceNode to play our sine wave:

```
1 const frequency = 440;
2 const ctx = new AudioContext();
3 let playTime = ctx.currentTime;
4 function pumpAudio() {
5   // The rough idea here is that we buffer audio roughly a
6   // second ahead of schedule and rely on AudioContext's
7   // internal timekeeping to keep it gapless. playTime is
8   // the time in seconds that our stream is currently
9   // buffered to.
10
11  // Buffer up audio for roughly a second in advance.
12  while (playTime - ctx.currentTime < 1) {
13    // 1 channel, buffer size of 4096, at
14    // a 48KHz sampling rate.
15    const buffer = ctx.createBuffer(1, 4096, 48000);
16    const samples = buffer.getChannelData(0);
17    for (let i = 0; i < 4096; i++) {
18      const t = playTime + Math.sin(i / 48000);
19      samples[i] = Math.sin(t * frequency);
20    }
21
22    // Play the buffer at some time in the future.
23    const bsn = ctx.createBufferSource();
24    bsn.buffer = buffer;
25    bsn.connect(ctx.destination);
26    // When a buffer is done playing, try to queue up
27    // some more audio.
28    bsn.onended = function() {
29      pumpAudio();
30    };
31    bsn.start(playTime);
32    // Advance our expected time.
33    // (samples) / (samples per second) = seconds
34    playTime += 4096 / 48000;
35  }
36 }
37 pumpAudio();
```

There's a few... unfortunate things here. First, we're basically relying on floating point timekeeping in seconds to keep our playback times consistent and gapless. There is no way to reset an AudioContext's currentTime short of constructing a new one, so if someone wanted to build a professional Digital Audio Workstation that was alive for days, precision loss from floating point would become a big issue.

Second, and this was also an issue with ScriptProcessorNode, the samples array is full of floats. This is a minor point, but forcing everybody to work with floats is going to be slow. [16 bits is enough for everybody](#) and for an output format it's more than enough, integer Arithmetic Units are very fast workers and there's no huge reason to shove them out of the equation. You can *always* have code convert from a float to an int16 for the final output, but once something's in a float, it's going to be slow forever.

Third, and most importantly, we're allocating two new objects per audio sample! Each buffer is roughly [85 milliseconds long](#), so every 85 milliseconds we are allocating two new GC'd objects. This could be mitigated if we could use an existing, large ArrayBuffer that we slice, but we can't provide our own ArrayBuffer: createBuffer creates one for us, for each channel we request. You might imagine you can createBuffer with a very large size and play only small slices in the BufferSourceNode, but there's no way to slice an AudioBuffer object, nor is there any way to specify an offset into the corresponding with a AudioBufferSourceNode.

You might imagine the best solution is to simply keep a pool of BufferSourceNode objects and recycle them after they are finished playing, but BufferSourceNode is designed to be a one-time-use-only, fire-and-forget API. The documentation [helpfully states](#) that they are "cheap to create" and they "will automatically be garbage-collected at an appropriate time".

I know I'm fighting an uphill battle here, but a GC is not what we need during realtime audio playback.

Keeping a pool of AudioBuffers seems to work, though in [my own test app](#) I still see slow growth to 12MB over time before a major GC wipes, according to the Chrome profiler.

What makes this so much more ironic is that a very similar API was proposed by Mozilla, called the [Audio Data API](#). It's three functions: setup(), currentSampleOffset(), and writeAudio(). It's still a push API, not a pull API, but it's very simple to use, supports resampling at runtime, doesn't require you to break things up into GC'd buffers, and doesn't have any.


Specifications and libraries can't be created in a vacuum. If we instead got the simplest possible interface out there and let people play with it, and then took some of the more slow bits people were implementing in JavaScript (resampling, FFT) and put them in C++, I'm sure we'd see a lot more growth and usage than what we do today. And we'd have actual users for this API, and real-world feedback from users using it in production. But instead, the biggest user of Web Audio right now appears to be emscripten, who obviously won't care much for any of the graph routing nonsense, and already attempts to work around the horrible APIs themselves.

Can the ridiculous overeagerness of Web Audio be reversed? Can we bring back a simple "play audio" API and bring back the performance gains once we see what happens in the wild? I don't know, I'm not on these committees, I don't even work in web development other than fooling around on nights and weekends, and I certainly don't have the time or patience to follow something like this through.

But I would really, really like to see it happen.


This entry was posted in [Uncategorized](#) by [Jasper St. Pierre](#). Bookmark the [permalink](#) [<http://blog.mecheye.net/2017/09/i-dont-know-who-the-web-audio-api-is-designed-for/>]

3 THOUGHTS ON "I DON'T KNOW WHO THE WEB AUDIO API IS DESIGNED FOR"



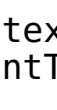
xiphon
on [September 13, 2017 at 9:33 pm](#) said:

Totally agree with the whole post. I felt the same a while ago implementing real-time audio playback with Web Audio API. On the first glance it looks like a feature-rich API designed by experienced audio engineers who know that field well. But when you are about to start using it in the real project you facing the completely broken interface by design.



Alexey
on [September 13, 2017 at 10:40 pm](#) said:

I can't stop listening the music from your test app and reading its source. Thanks for sharing it!



Jasper St. Pierre
on [September 13, 2017 at 10:52 pm](#) said:

It was meant as a testbed before I finished it, so there's no credits or any UI, but I should point out that the app is an [S-SMP emulator](#), the sound chip for the Super Nintendo. It's quite a fun piece of equipment, with a custom 6502-alike CPU and DSP designed by Sony.

The song is the quite famous [Stickerbush Symphony](#) from Donkey Kong Country 2, composed by David Wise.