Web Audio API performance and debugging notes

This version:
https://padenot.github.io/web-audio-perf

Issue Tracking:
GitHub

Editor:
Paul Adenot <padenot@mozilla.com>

Abstract

These notes present the Web Audio API from a performance and debugging point of view, outlining some differences between implementation.

Table of Contents

1. Introduction

In this tutorial, we will look at two different aspects of working with the Web Audio API.

First, we'll have a look at the different implementations available today, how to inspect their source code, and report problems found while testing application using the Web Audio API with them.

We'll then have a look into the performance characteristics of the different `AudioNode`s available, their performance profile, overall CPU and memory cost.

We'll continue by exploring the different strategies and techniques implementors have used when writing their implementation of the Web Audio API.

We'll then look into ways to make processing lighter, while still retaining the essence of the application, for example to make a "degraded" mode for mobile. We'll use techniques such as substituting rendering methods to trade fidelity against CPU load, pre-baking assets, minimizing resampling.

Finally, we'll touch on tools and techniques useful to debug audio problems, both using the browser developer tools, or JavaScript code designed to inspect static and dynamic audio graphs and related Web Audio API objects.

2. The different implementations

Four complete (if there is such thing, considering the standard is always evolving) Web Audio API implementations are available as of today in browsers:

- The first ever implementation was part of WebKit. At the time, Chrome and Safari were sharing the same code.
- Then, Blink got forked from WebKit, and the two gradually diverged. They share a lot of code, but can be considered separate implementations these days.
- Gecko was the second implementation, mostly from scratch, but borrowing a few files from the Blink fork, for some processing code.
- Edge's source are not available, but is based on an old snapshot of Blink.

The source code from the first three implementations can be read, compiled and modified, here are the relevant locations:

- WebKit's implementation lives at https://trac.webkit.org/browser/trunk/, in at the path `/Source/WebCore/Modules/webaudio`.
- Blink's implementation can be found at https://code.google.com/p/chromium/codesearch#chromium/src/ which is a handy web interface, with cross-referencing of symbols. The Web Audio API implementation lives at `third_party/WebKit/Source/modules/webaudio`, but a number of classes and functions, intended to be shared among different Chromium modules are located at `./third_party/WebKit/Source/platform/audio/`.

- Gecko's implementation can be found at https://dxr.mozilla.org/mozilla-central, which is also a nice web interface with cross-referencing of symbols, and the Web Audio API implementation is located in `dom/media/webaudio`. Some shared components are in `dom/media`.

Issues (about performance or correctness) can be filed in the project's bug tracker:

- WebKit: https://bugs.webkit.org/enter_bug.cgi?product=WebKit&component=Web%20Audio (WebKit bugzilla account needed).
- Blink: https://new.crbug.com/ (Google account needed).
- Gecko: https://bugzilla.mozilla.org/enter_bug.cgi?product=Core&component=Web%20Audio (GitHub, Persona or Mozilla Bugzilla account needed).

When filing issues, a minimal test case reproducing the issue is very welcome, as is a benchmark in case of performance problems. A stand alone HTML file is usually preferred to a jsfiddle, jsbin or similar service, for archival purposes.

3. Performance analysis

3.1. `AudioNode`s characteristics

This section explains the characteristics of each of the `AudioNode` that are available in the Web Audio API, from four angles.

- CPU, that is the temporal complexity of the processing algorithm;
- Memory, whether node needs to keep buffers around, or needs internal memory for processing;
- Latency, whether the processing induces a delay in the processing chain. If this section is not present, the node does not add latency;
- Tail, whether you can have a non-zero output when the input is continuously silent (for example because the audio source has stopped). If this section is not present, the node does not have a tail.

3.1.1. AudioBufferSourceNode

CPU

The `AudioBufferSourceNode` automatically resamples its `buffer` attribute to the sample-rate of the `AudioContext`. Resampling is done differently in different browsers. Edge, Blink and Webkit based browser use linear resampling, that is cheap, has no latency, but has low quality.

Gecko based browser use a more expensive but higher quality technique, that introduces some latency.

Memory

The `AudioBufferSourceNode` reads sample from an `AudioBuffer` that can be shared between multiple nodes. The resampler used in Gecko uses some memory for the filter, but nothing major.

3.1.2. ScriptProcessorNode

CPU

On Gecko-based browsers, this node uses a message queue to send buffers back and forth between the main thread and the rendering thread. On other browsers, buffer ping-ponging is used. This means that the former is more reliable against dropouts, but can have a higher latency (depending on the main thread event loop load), whereas the latter drops out more easily, but has fixed latency.

Memory

Buffers have to be allocated to move audio back and forth between threads. Since Gecko uses a buffer queue, more memory can be used.

Latency

The latency is specified when creating the node. If Gecko has trouble keeping up, the latency will increase, up to a point where audio will start to drop.

3.1.3. AnalyserNode

CPU

This node can give frequency domain data, using a Fast Fourier Transform algorithm, that is expensive to compute. The higher the buffer size, the more expensive the computing is. `byte` version of the analysis methods are not cheaper than `float` alternative, they are provided for convenience: the `byte` version are computed from the `float` version, using simple quantization to 2^8 values.

Memory

Fast Fourier Transform algorithms use internal memory for processing. Different platforms and browsers have different algorithms, so it's hard to quantify exactly how much memory is going to be used. Additionaly, some memory is going to be used for the `AudioBuffer` passed in to the analysis methods.

Latency

Because of the windowing function there can be some perceived latency in this node, but windowing can be disabled by setting it to 0.

Tail

Because of the windowing function there can be a tail with this node, but windowing can be disabled by setting it to 0.

3.1.4. GainNode

CPU

Gecko-based browsers, the gain is always applied lazily, and folded in before processing that require to touch the samples, or before send the rendered buffer back to the operating system, so `GainNode` with a fixed gain are essentially free. In other engines, the gain is applied to the input buffer as it's received. When automating the gain using `AudioParam` methods, the gain is applied to the buffer in all browsers.

Memory

A `GainNode` is stateless and has therefore no associated memory cost.

3.1.5. DelayNode

CPU

This node essentially copies input data into a buffer, and reads from this buffer at a different location to compute its output buffer.

Memory

The memory cost is a function of the number of input and output channels and the length of the delay line.

Latency

Obviously this node introduces latency, but no more than the latency set by its parameter

Tail

This node is being kept around (not collected) until it has finished reading and has output all of its internal buffer.

3.1.6. BiquadFilterNode

CPU

Biquad filters are relatively cheap (five multiplication and four additions per sample).

Memory

Very cheap, four float for the memory of the filter.

Latency

Exactly two frames of latency, due to how the filter works.

Tail

Variable tail, depending on the filter setting (in particular the resonance).

3.1.7. IIRFilterNode

CPU

Similarly to the biquad filter, they are rather cheap. The complexity depends on the number of coefficients, that is set at construction.

Memory

Again, the memory usage depends on the number of coefficients, but is overall very small (a couple floats per coefficients).

Latency

A frame per coefficient.

Tail

Variable, depending on the value of the coefficients.

3.1.8. WaveShaperNode

CPU

The computational complexity depends on the oversampling. If no oversampling is used, a sample is read in the wave table, using linear interpolation, which is a cheap process in itself. If oversampling is used, a resampler is used. Depending on the browser engine, different resampling techniques can be used (FIR, linear, etc.).

Memory

This node is making a copy of the curve, so it can be quite expensive in terms of memory.

Latency

This node does not add latency if oversampling is not used. If over-sampling is used, and depending on the resampling technique, latency can be added by the processing.

Tail

Similarly, depending on the resampling technique used, and when using over-sampling, a tail can be present.

3.1.9. PannerNode, when `panningModel == "HRTF"`

CPU

Very expensive. This node is constantly doing convolutions between the input data and a set of HRTF impulse, that are characteristic of the elevation and azimuth. Additionaly, when the

position changes, it interpolates (cross-fades) between the old and new position, so that the transition between two HRTF impulses is smooth. This means that for a stereo source, and while moving, there can be four convolvers processing at once. Additionaly, the HRTF panning needs short delay lines.

Memory
The HRTF panner needs to load a set of HRTF impulses around when operating. Gecko loads the HRTF database only if needed, while other engines load it unconditionally. The convolver and delay lines require memory as well, depending on the Fast Fourier Transform implementation used.

Latency
HRTF always adds some amount of delay, but the amount depends on the azimuth and elevation.

Tail
Similarly, depending on the azimuth and elevation, a tail of different duration is present.

3.1.10. PannerNode, when `panningModel == "equalpower"`

CPU
Rather cheap. The processing has two parts:

- First, the azimuth needs to be determined from the Cartesian coordinate of the source and listener, this is a bit of vector maths, and can be cached by the implementation for static sources.
- Then, gain is applied, maybe blending the two channels is the source is stereo.

Memory
The processing being stateless, this has no memory cost.

3.1.11. StereoPannerNode

CPU
Similar to the `"equalpower"` panning, but the azimuth is cheaper to compute since there is no need to do the vector math, we already have the position.

Memory
Stateless processing, no memory cost.

3.1.12. ConvolverNode

CPU

Very expensive, and depending on the duration of the convolution impulse. A background thread is used to offload some of the processing, but computational burst can occur in some browsers. Basically, multiple FFT are computed for each block.

Memory

The node is making a copy of the buffer for internal use, so it's taking a fair bit or memory (depending on the duration of the impulse). Additionaly, some memory can be used for the Fast Fourier Transform implementation, depending on the platform.

Latency

Convolver can be used to create delay-like effect, so latency can certainly be introduced by a `ConvolverNode`.

Tail

Depending on the convolution impulse, there can be a tail.

3.1.13. ChannelSplitterNode / ChannelMergerNode

CPU

This is merely splitting or merging channels, that is copying buffer around.

Memory

No memory implications

3.1.14. DynamicsCompressorNode

CPU

The exact algorithm is not specified yet. In practice, it's the same in all browsers, a peak detecting look-ahead, with a pre-emphasis and post-de-emphasis, not too expensive.

Memory

Not very expensive in terms of memory, just some floats to track the internal state.

Latency

Being a look ahead compressor, it introduces a fixed look-ahead of six milliseconds.

Tail

Because of the emphasis, there is a tail. Also, compression can boost quiet audio, so audible sound can appear to last longer.

3.1.15. OscillatorNode

CPU

The basic wave forms are implemented using multiple wave tables computed using the inverse Fourier transform of a buffer with carefully chosen coefficients (apart from the sine that is computed directly in Gecko). This means that there is an initial cost when changing the wave form, that is cached in Gecko-based browser. After the initial cost, processing is essentially doing linear interpolation between multiple wave tables. When the frequency changes, new tables have to be computed.

Memory

A number of wave tables have to be stored, that can take up some memory. Those are shared in Gecko-based browsers, apart from the sine wave in Gecko, that is directly computed.

3.2. Other noteworthy performance characteristics

3.2.1. Memory model

Web Audio API implementation use two threads. The *control thread* is the thread on which are issued the Web Audio API calls: `createGain`, `setTargetAtTime`, etc. The *rendering thread* is the thread that is responsible for rendering the audio. This can be a normal thread (for example for an `OfflineAudioContext`) or a system provided, high-priority audio thread (for a normal `AudioContext`). Of course, informations have to be communicated between the two threads.

Current Web Audio API implementations have taken two different approaches to implement the specification. Gecko-based browsers use an *message passing* model, whereas all the other implementation use a *shared memory* model. This has a number of implications in practice.

First, in engines that are using the *shared memory* model, changes to the graph and `AudioParam` can occur at any time. This means that in some scenario, manipulation (from the main thread) of internal Web Audio API data structures can be reflected more quickly in the rendering thread. For example, if the audio thread is current rendering, a modification from the main thread will be reflected immediately on the rendering thread.

A drawback of this approach is that it is necessary to have some synchronization between the control thread and the rendering thread. The rendering thread is often very high priority (usually the highest priority on the system), to guarantee no under-runs (or dropouts), which are considered catastrophic failures, for most audio rendering system. Under-runs usually occur when the audio rendering thread did not make its deadline. For example, it took more than 5 milliseconds of processing to process 5 milliseconds of audio. Non-Gecko based browers are often using *try locks* to ensure smooth operation.

In certain parts of the Web Audio API, there is a need to a be able to access data structure sent to the rendering thread, from the main thread. Gecko-based browsers keep two synchronized copies of the data structure to implement this, this has a cost in memory, that other engines don't have to pay.

3.2.2. AudioParam

The way a web application uses `AudioParam` plays an important role in the performance of the application. `AudioParam` come in two flavours, *a-rate* and *k-rate* parameters. *a-rate* parameters have their value computed for each audio sample, whereas *k-rate* parameters are computed once per 128 frames block.

`AudioParam` methods (`setValueAtTime`, `linearRampToValueAtTime`, etc.) each insert *events* in a list of events that is later accessed by the rendering thread.

Handling `AudioParam`, for an engine, means first finding the right event (or events) to consider for the block of audio to render. Different approaches are taken by different browsers. Gecko prefers to prune all events that are in the past but the one that is right before the current time (certain events require to have a look at the previous event's value). This guarantees amortized *O(1)* complexity (amortized because deallocations can take some time). Other engines do a linear scan in the event list to find the right one.

In practice, both techniques perform well enough so that the difference is not noticeable most of the time. If the application uses *a lot* of `AudioParam` events, non-Gecko based browers can have performance issues, because scanning through the list starts to take a non-trivial amount of time. Strategies can be employed to mitigate this issue, by creating new `AudioNode`, with new `AudioParam`, that start with an empty list of events.

For example, let's take a `GainNode` that is used as the envelope of an `OscillatorNode`-based kick drum, playing each beat at 140 beat per minute. The envelope is often implemented using a `setValueAtTime` call to set the initial volume of the hit, that often depends on the velocity, immediately followed by a `setTargetAtTime` call, at the same time, with a value of 0 to have a curve that decays to silence, and a time constant that depends on the release parameter of the synth. At 140BPM, 280 events will be inserted by minute. To ensure stable performance, and leveraging the fact that a `GainNode` is very cheap to create and connect, it might be worth it to consider swapping the node that is responsible for the envelope regularly.

Gecko-based browers, because of their event-loop based model, have two copies of the event list. One on the main thread, that contains all the events, and one on the rendering thread, that is regularly pruned to only contain relevant events. Other engines simply synchronize the event list using a lock, and only have one copy of the event list. Depending on how the application schedules its `AudioParam` events (the worst case being when all the events are scheduled in advance), having two copies of the event list can take a non-negligible amount of memory.

The second part of *AudioParam* handling is computing the actual value (or value, for an *a-rate* parameter, based on past, present and future events. Gecko-based browsers are not very efficient are

computing those values (and suffer from a number of bugs in the implementation). This code is going to go through a rewrite *real soon (tm)*. Other engines are much more efficient (which, most of the time, offsets their less efficient way of searching for the right events to consider). Blink even has optimized code to compute the automation curves, using SSE intrinsics on x86.

All that said, and while implementations are becoming more and more efficient at computing automation curves, it is event more efficient to not use `AudioParam` if not necessary, implementation often take different code path if they know an `AudioParam` value is going to be constant for a block, and this is easier to detect if the `value` attribute has been directly set.

3.2.3. Node ordering

Audio nodes have to be processed in a specific order to get correct results. For example, considering a graph with an `AudioBufferSourceNode` connected to a `GainNode`, connected to the `AudioDestinationNode`. The `AudioBufferSourceNode` has to be processed first, it gets the values from the `AudioBuffer` and maybe resamples them. The output of this node is passed to the `GainNode`, that applies its own processing, passing down the computed data to the `AudioDestinationNode`.

Gecko-based browsers and other engines have taken two different approaches for ordering Web Audio API graphs.

Gecko uses an algorithm based on Tim Leslie's iterative implementation [1][2] of Pearce's variant [3] of Tarjan's strongly connected components (SCC) algorithm, which is basically a kind of topological sort that takes cycles into account to properly handle cycles with `DelayNode`.

Other engines simply do a depth first walk of the graph, uses a graph coloring approach to detect cycles, and pull from the `DelayNode` internal buffer in case of cycles. This is way simpler, but differs in rendering with the other approach.

Gecko is only running the algorithm if the topology of the graph has changed (for example if a node has been added, or a connection has been removed), while other engine perform the traversal for each rendering quantum. Depending on if the application has a mostly static or very dynamic graph topology, this can have an impact on performance.

3.2.4. Latency

Audio output latency is very important for the Web Audio API. Usually, implementations use the lowest available audio latency on the system to run the *rendering thread*. Roughly, this mean on the

order of 10ms on Windows (post-Vista, using WASAPI), very very low on OSX and iOS (a few milliseconds), a 30-40ms on Linux/PulseAudio. There are not yet engines that support Jack with low-latency code, although this would certainly have amazing latency numbers. Windows XP does not usually have great output latency, and licensing terms (and available time to write code!) prevent the use of ASIO on Windows, which would also have great performance.

Android devices are a bit of a problem, as the latency can vary between 12.5ms on a Galaxy Nexus (which was the device with the lowest latency) up to 150ms on some very cheap devices. Although there is a way to know the latency using system API (that browsers use for audio/video synchronization), Web App authors don't currently have a way to know the output latency. This is currently being addressed (#12).

Creating `AudioContext` with a higher latency is a feature that is currently under discussion (#348). This allows a number of things, such as battery savings for use-cases that don't require very low latency. For example, an application that only does playback, but wants to display the Fourier transform of the signal, or simply the wave form of the signal itself, have an equalizer on the signal path, apply compression, all would benefit from this feature, because real-time interaction is not required. Of course, the latency that is the most efficient, battery-wise and regardless of audio output latency depends on the operating system and hardware.

By using bigger rendering quantum (by using bigger buffers, caused by increasing the audio latency in most systems) also allows creating and rendering more complex Web Audio API graphs. Indeed, audio systems often pull quite a lot of memory into CPU cache lines to compute each rendering block (be it audio sample data, automation timeline event arrays, the memory for the nodes themselves, etc.). Rendering for a longer period of time means having caches that are more hot and lets the CPU crunching numbers without having to wait for memory faults. Digital Audio Workstation users often increase the latency when they heard "audio crackles" (under-runs), the same logic applies here. Of course, there is a point where having a higher latency is actually not helping the computation. This value depends on the system.

3.2.5. Browser architecture

Firefox (that is based on Gecko) uses only one process for all the tabs and the chrome (which is the user interface of the browser, that is, the tabs, the menus, etc., a very confusing term nowadays), whereas other browsers use multiple processes.

While the rendering of Web Audio API graphs happen on dedicated thread, this has important implications in terms of responsiveness. The event loop is Gecko is shared between all the tabs. When using a Web Application that uses the Web Audio API, if all the other tabs are still doing some

processing, receiving events, running script, etc. in the background, the Web Audio API calls will be delayed, so it is necessary to plan a bit more in advance (see Chris Wilson's Tale of two clocks). This is somewhat being addressed by the electrolysis (e10s) project, but there is still a long way to go to catch up in responsiveness.

Other engines usually use multiple processes, so the event loop load is split among multiple processes, and there is less chances that the script that is issuing Web Audio API calls is delayed, allowing more tight delays for scheduling things (be it an `AudioParam`, the starting an `AudioBufferSourceNode`, etc.).

3.2.6. `decodeAudioData`

Gecko based browsers use a thread pool (with multiple threads) when handling multiple `decodeAudioData` calls, whereas other browsers serialize the decoding on a thread.

Additionaly, audio is resampled differently, and different audio decoders are used, with different optimizations on different OS and architectures, leading to a wide variety of performance profiles.

Authors should take advantage of (nowadays ubiquitous) multi-core machines (even phones often have four or more core these days), and try to saturate the CPUs with decoding operations. Since the rendering thread is higher priority than the decoding threads, no audio under-runs are to be expected.

3.2.7. Micro-optimizations

Audio processing is composed of a lot of very tight loops, repeating a particular operation over and over on audio buffers. Implementations take advantage of SIMD (Single Instruction Multiple Data) available in CPUs, and optimize certain very common operation after profiling, to make processing go faster, in order to render have bigger graphs, allow lower audio latency.

Gecko has NEON (on ARM) and SSE (on x86) function implementing all the very common functions (Applying a gain in place, copying a buffer while applying a gain value, panning a mono buffer to a stereo position, etc.), as well as optimized FFT code (taken from the FFMPEG project on x86, and OpenMax DL routines on ARM). Of course, these optimizations are only used if the CPU on which the browser is running has the necessary extensions, falling back to normal scalar code if not. Depending on the function, those optimizations have proven to be between three and sixteen times faster than their scalar counterpart.

Blink has SSE code for some `AudioParam` event handling, and has the same FFT code as Gecko (both on x86 and ARM).

WebKit can use different FFT implementation, depending on the platform, for example the FFT from the *Accelerate* framework on OSX.

This results in a variety of performance profiles for FFT-based processing (HRTF panner, `ConvolverNode`, `AnalyserNode`, etc.).

Additionaly, very often, the decoding code used by the implementation of `decodeAudioData` are very well optimized using SIMD and a variety of other techniques.

4. Using lighter processing

Audio processing is a hard real-time process. If the audio has no been computed when the system is about to output it, drop outs will occur, in the form of clicks, silence, noise and other unpleasant sounds.

Web Browsers run a variety of platform and devices, some of them being very powerful, and some of them with limited resources. To ensure a smooth experiences for all users and depending on the application itself, it can be necessary to use techniques that does not sound as good as the normal technique, but still allow to have a convincing experience, retaining the essence of the interaction.

4.1. Custom processing

Sometimes, the Web Audio API falls short in what it has to offer, to solve particular problems. It can be necessary to use an `AudioWorklet` or a `ScriptProcessorNode` to implement custom processing.

JavaScript is a very fast language is written properly, but there are a number of rules to obey to achieve this result.

- Using typed array is a must. They are very fast compared to the normal array. In the Web Audio API, the `Float32Array` is the most common. Re-using arrays is important, there is no need to spend a lot of time in the allocator code.
- Keeping the working set small is important, not using a lot of arrays and always using the same data is faster.
- No DOM manipulation or fiddling with the prototype of object during processing as this invalidates the JITed code.
- Trying to stay as mono-morphic as possible and always using the same code path yields more optimized JITed code. Suddenly calling a function with a normal array instead of a

`Float32Array` will invalidate a lot of code and take a slow path.

- Compiling C or C++ to JavaScript (or soon Web Assembly) yields the best performance. *emscripten* or other tools can be used to compile libraries or custom code into a typed subset of JavaScript that does not allocate and is very fast at number crunching.
- Experimental extensions, such as `SIMD.js` or `SharedArrayBuffer` can make things run faster, but they are not available in all browsers.

4.2. Worker-based custom processing

All of the above techniques can be used with workers, to offload heavy processing. While the worker does not have access to the Web Audio API, buffers can be transfered without copy, and the audio can then be sent back to the main thread for use with the Web Audio API.

This has latency implications, and care must be taken so that the worker always has finished processing on time, generally using a FIFO.

4.3. Copying audio data

Reusing `AudioBuffer` internal data can be performed more efficiently than with calling `getChannelData()`. `copyFromChannel()` and `copyToChannel()` allow the engine to to optimize away some copies and allocations.

4.4. Built-in resampling

The `OfflineAudioContext` can be use to perform off-main-thread and off-rendering-thread re-resampling of audio buffer. This can have two different uses:

- By resampling an `AudioBuffer` to the sample-rate of the `AudioContext`, no time is spent resampling in the `AudioBufferSourceNode`.
- By resampling an `AudioBuffer` to a low sample rate, memory can be saved. This has sound quality and CPU implications, but not all sounds have partials that are high enough to require a full-band rate.

```
function resample(input, target_rate) {
  return new Promise((resolve, reject) => {
    if (typeof input != "AudioBuffer")  {
      reject()
    }
    if (typeof target_rate != "number" && target_rate <= 0) {
      reject()
    }
    var resampling_ratio = input.sampleRate / target_rate;
    var final_length = input.length * resampling_ratio;
    var off = new OfflineAudioContext(input.numberOfChannels,
                                      final_length, target_rate);

    var source = off.createBufferSource();
    source.buffer = input;
    source.connect(off.destination);
    source.start(0);

    off.startRendering().then(resolve)).catch(reject);
  });
}
```

4.5. Track or asset freezing

The `OfflineAudioContext` can be use to apply, ahead of playback, complex processing to audio buffer, to avoid having to recompute the processing each time. This is called *baking* or *freezing*. This can be employed for a full sound-track, as well as, for example, the individual sound effects of a video game.

4.6. Cheaper reverb

When using the Web Audio API, the simplest way of adding a reverberation effect to a sound is to connect it into a `ConvolverNode`, setting its `buffer` attribute to a reverb impulse, often a decaying curve of some sort, synthesized using a program, or recorded and processed from a real location. While this setup has a very good sound quality (of course, depending on the impulse chosen), it is very expensive to compute. While computers and modern phones are very powerful, longer reverb tails, or running the Web App on a cheap mobile devices might not work.

Cheaper reverb can be created using delay lines, all-pass and low-pass filters, to achieve a very convincing effect. This also has the advantage of having parameters you can change, instead of an impulse buffer.

4.7. Cheaper panning

HRTF panning is based on convolution, sounds good, but is very expensive. It can be replaced, for example on mobile, by a very short reverb and an equal-power panner, with a distance function and Cartesian positions adjusted properly. This has the advantage of not requiring very very expensive computation when continuously changing the positions of the listener and source, something that is often the case when basing those positions on sensor data, such as accelerometer and gyroscopes.

5. Debugging Web Audio API applications

5.1. Node Wrapping

A good strategy to debug Web Audio API application is to wrap `AudioNode`s using ES6's `Proxy` or normal prototype override, to keep track of state, and being able to tap in an re-route audio, insert analysers and so on.

5.2. Firefox' Web Audio API debugger

Firefox has a custom developer tool panel, that shows the graph, with the following features:

- Ability to see the topology of the graph, garbage collection of nodes and sub-graph portion.
- Setting and getting the value of `AudioNode` attributes.
- Bypass nodes

There are some up and coming feature coming up as well:

- Memory consumption of `AudioNode`s and `AudioBuffers`.
- Tapping into nodes, inserting analysers.
- Inspecting `AudioParam` time-lines
- CPU profiling of real-time budget

5.3. Memory profiling

It is pretty easy to determine the size in bytes of the buffer space used by `Audiobuffer`:

```
function AudioBuffer_to_bytes(buffer) {
  if (!(buffer instanceof AudioBuffer)) {
    throw "not an AudioBuffer.";
  }
  const sizeoffloat32 = 4;
  return buffer.length * buffer.numberOfChannels * sizeoffloat32;
}
```

Other memory figures can be obtained in Firefox by going to `about:memory`, clicking "Measure", and looking to the tab containing your page.

References