Web Audio Updates in Chrome 49

By Chris Wilson

Chris is a contributor to Web**Fundamentals**

Chrome has been consistently and quietly improving its support for the Web Audio API. In Chrome 49 (Beta as of Feb 2016, and expect to be Stable in March 2016) we've updated several features to track the specification, and also added one new node.

decodeAudioData() now returns a promise.

The decodeAudioData() method on **`AudioContext`** now returns a **`Promise`**, enabling Promise-based asynchronous pattern handling. The **`decodeAudioData()`** method has always taken success and error callback functions as parameters:

```
context.decodeAudioData( arraybufferData, onSuccess, onError);
```

But now you can use standard Promise method to handle the asynchronous nature of decoding audio data instead:

```
context.decodeAudioData( arraybufferData ).then(
     (buffer) => { /* store the buffer */ },
     (reason) => { console.log("decode failed! " + reason) });
```

Although in a single example this looks more verbose, Promises make asynchronous programming easier and more consistent. For compatibility, the Success and Error callback functions are still supported, as per the specification.

OfflineAudioContext now supports suspend() and resume()

At first glance, it might seem strange to have suspend() on an OfflineAudioContext. After all, **`suspend()`** was added to **`AudioContext`** to enable putting the audio hardware into standby mode, which seems pointless in scenarios when you're rendering to a buffer (which is what **`OfflineAudioContext`** is for, of course). However, the point of this feature is to be able to construct only part of a "score" at a time, to minimize the memory usage. You can create more nodes while suspended in the middle of a render.

As an example, Beethoven's Moonlight Sonata contains around 6,500 notes. Each "note" probably deconstructs to at least a couple of audio graph nodes (e.g. an AudioBuffer and a Gain node). If you wanted to render the entire seven-and-a-half minutes into a buffer with **`OfflineAudioContext`**, you probably don't want to create all those nodes at once. Instead, you can create them in chunks of time:

```
var context = new OfflineAudioContext(2, length, sampleRate);
scheduleNextBlock();
context.startRendering().then( (buffer) => { /* store the buffer */ } );

function scheduleNextBlock() {
    // create any notes for the next blockSize number of seconds here
    // ...

    // make sure to tell the context to suspend again after this block;
    context.suspend(context.currentTime + blockSize).then( scheduleNextBlock );

    context.resume();
}
```

This will enable you to minimize the number of nodes that need to be pre-created at the beginning of the rendering, and lessen memory demands.

IIRFilterNode

The spec has added a node for audiophiles who want to create their own precisely-specified infinite-impulse-response: the IIRFilterNode. This filter complements the BiquadFilterNode, but allows complete specification of the filter response parameters (rather than the **`BiquadFilterNode`**'s easy-to-use **`AudioParams`** for type, frequency, Q, and the like). The **`IIRFilterNode`** allows precise specification of filters that couldn't be created before, like single-order filters; however, using the IIRFilterNode requires some deep knowledge of how IIR filters work, and they are also not schedulable like BiquadFilterNodes.

Previous changes

I also want to mention a couple of improvements that have gone in previously: in Chrome 48, **`BiquadFilter`** node automation started running at audio rate. The API didn't change at all to do this, but this means your filter sweeps will sound even smoother. Also in Chrome 48, we added chaining to the **`AudioNode.connect()`** method by returning the node we're connecting to. This makes it simpler to create chains of nodes, as in this example:

```
sourceNode.connect(gainNode).connect(filterNode).connect(context.destinat
```

That's all for now, and keep rocking!

Last updated July 2, 2018.