Enter AudioWorklet

By Hongchan Choi
Software Engineer working on Web Audio in Chromium

Note: AudioWorklet is enabled by default in Chrome 66.

Chrome 64 comes with a highly anticipated new feature in Web Audio API - AudioWorklet. This article introduces its concept and usage for those who are eager to create a custom audio processor with JavaScript code. Please take a look at the live demos on GitHub or the instruction on how to use this feature.

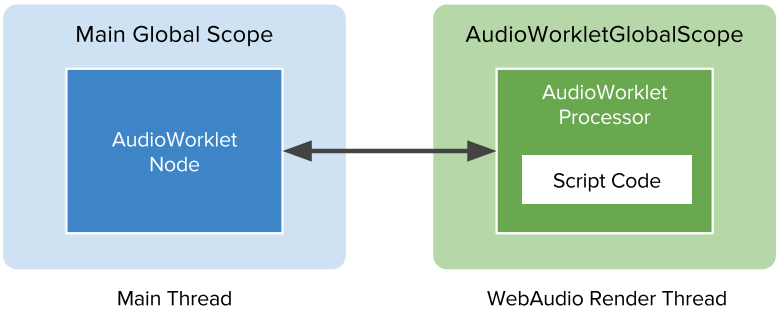
Background: ScriptProcessorNode

Audio processing in Web Audio API runs in a separate thread from the main UI thread, so it runs smoothly. To enable custom audio processing in JavaScript, the Web Audio API proposed a ScriptProcessorNode which used event handlers to invoke user script in the main UI thread.

There are two problems in this design: the event handling is asynchronous by design, and the code execution happens on the main thread. The former induces the latency, and the latter pressures the main thread that is commonly crowded with various UI and DOM-related tasks causing either UI to "jank" or audio to "glitch". Because of this fundamental design flaw, `ScriptProcessorNode` is deprecated from the specification and replaced with AudioWorklet.

Concepts

AudioWorklet nicely keeps the user-supplied JavaScript code all within the audio processing thread — that is, it doesn't have to jump over to the main thread to process audio. This means the user-supplied script code gets to run on the audio rendering thread (AudioWorkletGlobalScope) along with other built-in AudioNodes, which ensures zero additional latency and synchronous rendering.



Registration and Instantiation

Using AudioWorklet consists of two parts: AudioWorkletProcessor and AudioWorkletNode. This is more involved than using ScriptProcessorNode, but it is needed to give developers the low-level capability for custom audio processing. AudioWorkletProcessor represents the actual audio processor written in JavaScript code, and it lives in the AudioWorkletGlobalScope. AudioWorkletNode is the counterpart of AudioWorkletProcessor and takes care of the connection to and from other AudioNodes in the main thread. It is exposed in the main global scope and functions like a regular AudioNode.

Here's a pair of code snippets that demonstrate the registration and the instantiation.

```
// The code in the main global scope.
class MyWorkletNode extends AudioWorkletNode {
  constructor(context) {
    super(context, 'my-worklet-processor');
  }
}

let context = new AudioContext();

context.audioWorklet.addModule('processors.js').then(() => {
  let node = new MyWorkletNode(context);
});
```

Creating an AudioWorkletNode requires at least two things: an AudioContext object and the processor name as a string. A processor definition can be loaded and registered by the new AudioWorklet object's **`addModule()`** call. Worklet APIs including AudioWorklet are only available in a secure context, thus a page using them must be served over HTTPS, although **`http://localhost`** is considered a secure for local testing.

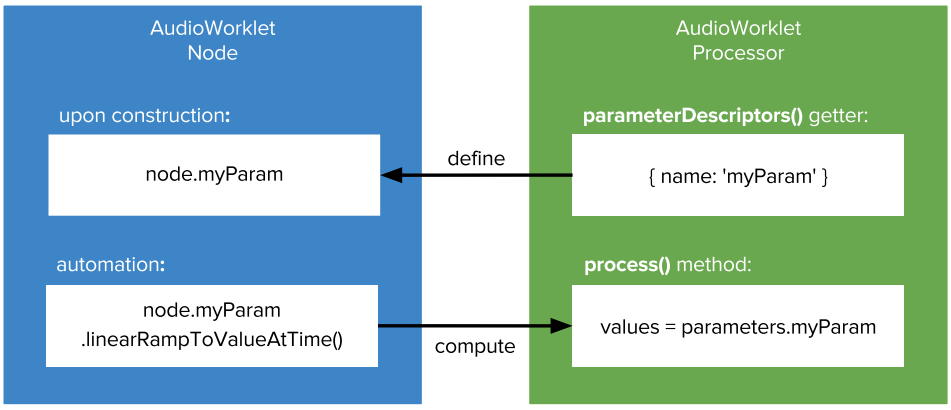
It is also worth noting that you can subclass AudioWorkletNode to define a custom node backed by the processor running on the worklet.

```
// This is "processor.js" file, evaluated in AudioWorkletGlobalScope upon
// audioWorklet.addModule() call in the main global scope.
class MyWorkletProcessor extends AudioWorkletProcessor {
  constructor() {
    super();
  }

  process(inputs, outputs, parameters) {
    // audio processing code here.
  }
}

registerProcessor('my-worklet-processor', MyWorkletProcessor);
```

The **`registerProcessor()`** method in the AudioWorkletGlobalScope takes a string for the name of processor to be registered and the class definition. After the completion of script code evaluation in the global scope, the promise from **`AudioWorklet.addModule()`** will be resolved notifying users that the class definition is ready to be used in the main global scope.

Custom AudioParam

One of the useful things about AudioNodes is schedulable parameter automation with AudioParams. AudioWorkletNodes can use these to get exposed parameters that can be controlled at the audio rate automatically.



User-defined AudioParams can be declared in an AudioWorkletProcessor class definition by setting up a set of AudioParamDescriptors. The underlying WebAudio engine will pick up this information upon the construction of an AudioWorkletNode, and will then create and link AudioParam objects to the node accordingly.

```
/* A separate script file, like "my-worklet-processor.js" */
class MyWorkletProcessor extends AudioWorkletProcessor {

  // Static getter to define AudioParam objects in this custom processor.
  static get parameterDescriptors() {
    return [{
      name: 'myParam',
      defaultValue: 0.707
    }];
  }

  constructor() { super(); }

  process(inputs, outputs, parameters) {
    // |myParamValues| is a Float32Array of 128 audio samples calculated
    // by WebAudio engine from regular AudioParam operations. (automation
    // methods, setter) By default this array would be all values of 0.707
    let myParamValues = parameters.myParam;
  }
}
```

`AudioWorkletProcessor.process()` method

The actual audio processing happens in the **`process()`** callback method in the AudioWorkletProcessor and it must be implemented by user in the class definition. The WebAudio engine will invoke this function in an isochronous fashion to feed **inputs** and parameters and fetch **outputs**.

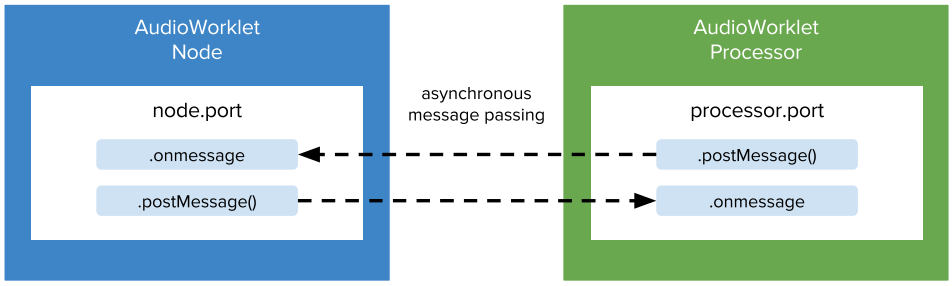
```
/* AudioWorkletProcessor.process() method */
process(inputs, outputs, parameters) {
  // The processor may have multiple inputs and outputs. Get the first input and
  // output.
  let input = inputs[0];
  let output = outputs[0];

  // Each input or output may have multiple channels. Get the first channel.
  let inputChannel0 = input[0];
  let outputChannel0 = output[0];

  // Get the parameter value array.
  let myParamValues = parameters.myParam;

  // Simple gain (multiplication) processing over a render quantum (128 samples).
  // This processor only supports the mono channel.
  for (let i = 0; i < inputChannel0.length; ++i) {
    outputChannel0[i] = inputChannel0[i] * myParamValues[i];
```

```
  }

  // To keep this processor alive.
  return true;
}
```

Additionally, the return value of the **`process()`** method can be used to control the lifetime of AudioWorkletNode so that developers can manage the memory footprint. Returning **`false`** from **`process()`** method will mark the processor inactive and the WebAudio engine will not invoke the method anymore. To keep the processor alive, the method must return **`true`**. Otherwise, the node/processor pair will be garbage collected by the system eventually.

Bi-directional Communication with MessagePort

Sometimes custom AudioWorkletNodes will want to expose controls that do not map to AudioParam. For example, a string-based **`type`** attribute could be used to control a custom filter. For this purpose and beyond, AudioWorkletNode and AudioWorkletProcessor are equipped with a MessagePort for bi-directional communication. Any kind of custom data can be exchanged through this channel.



MessagePort can be accessed via **`.port`** attribute on both the node and the processor. The node's **`port.postMessage()`** method sends a message to the associated processor's **`port.onmessage`** handler and vice versa.

```
/* The code in the main global scope. */
context.audioWorklet.addModule('processors.js').then(() => {
  let node = new AudioWorkletNode(context, 'port-processor');
  node.port.onmessage = (event) => {
    // Handling data from the processor.
    console.log(event.data);
  };

  node.port.postMessage('Hello!');
});
```

```
/* "processor.js" file. */
class PortProcessor extends AudioWorkletProcessor {
  constructor() {
    super();
    this.port.onmessage = (event) => {
      // Handling data from the node.
      console.log(event.data);
    };

    this.port.postMessage('Hi!');
  }

  process(inputs, outputs, parameters) {
    // Do nothing, producing silent output.
    return true;
  }
}

registerProcessor('port-processor', PortProcessor);
```

Also note that MessagePort supports Transferable, which allows you to transfer data storage or a WASM module over the thread boundary. This opens up countless possibility on how the AudioWorklet system can be utilized.

Walkthrough: building a GainNode

Putting everything together, here's a complete example of GainNode built on top of AudioWorkletNode and AudioWorkletProcessor.

Index.html

```
<!doctype html>
<html>
<script>
  const context = new AudioContext();

  // Loads module script via AudioWorklet.
  context.audioWorklet.addModule('gain-processor.js').then(() => {
    let oscillator = new OscillatorNode(context);

    // After the resolution of module loading, an AudioWorkletNode can be
    // constructed.
    let gainWorkletNode = new AudioWorkletNode(context, 'gain-processor');

    // AudioWorkletNode can be interoperable with other native AudioNodes.
```

```
    oscillator.connect(gainWorkletNode).connect(context.destination);
    oscillator.start();
  });
</script>
</html>
```

gain-processor.js

```
class GainProcessor extends AudioWorkletProcessor {

  // Custom AudioParams can be defined with this static getter.
  static get parameterDescriptors() {
    return [{ name: 'gain', defaultValue: 1 }];
  }

  constructor() {
    // The super constructor call is required.
    super();
  }

  process(inputs, outputs, parameters) {
    let input = inputs[0];
    let output = outputs[0];
    let gain = parameters.gain;
    for (let channel = 0; channel < input.length; ++channel) {
      let inputChannel = input[channel];
      let outputChannel = output[channel];
      for (let i = 0; i < inputChannel.length; ++i)
        outputChannel[i] = inputChannel[i] * gain[i];
    }

    return true;
  }
}

registerProcessor('gain-processor', GainProcessor);
```

This covers the fundamental of AudioWorklet system. Live demos are available at Chrome WebAudio team's GitHub repository.

Feature Transition: Experimental to Stable

AudioWorklet is enabled by default for Chrome 66 or later. In Chrome 64 and 65, the feature is behind the experimental flag. You can activate the feature with the following command line option:

```
--enable-blink-features=AudioWorklet
```

Along with the experimental release, we have added this feature in Chrome 64 and 65 as an Origin Trial for all platforms. With the Origin Trial, You can deploy code using AudioWorklet to users and get feedback from them. To participate in this trial please use the signup form. This experimental program is expired on 4/10/2018 and the participating developers need to prepare the site for the stable launch of the AudioWorklet in Chrome 66.

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code samples are licensed under the Apache 2.0 License. For details, see our Site Policies. Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.