# ECS512 Sound Design - Lab 3: Working with Worklets

Required resources:

• Code and sound examples contained in the **Lab3-resources.zip** file – available on QM+.

• **Google Chrome** browser.

• **Web Audio Inspector** extension for Chrome – to visualize the audio processing chain in the browser.

• **Atom** – a free on-line text and source code editor.

• **atom-live-server package** – a free add-on for Atom, which will launch a development http server.

• **Sonic Visualiser** – a free analysis and visualization software package developed at QMUL.

The web audio API is quite minimal in terms of the number of default nodes. The audio worklet allows developers to write their own custom audio nodes.

The lab is divided into 3 parts;

1. Noise node- a ‘hello world’ worklet
2. Gain node- how to design a worklet with parameters
3. Low pass filter node- initializing and storing internal variables in a worklet

Topics covered in this and other labs are explained in more detail at <https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API> , and links therein. It is a thorough guide, will reinforce everything being demonstrated here and will give you a more grounded understanding for future assignments. I also suggest you look at other background material on audio worklets;

* <https://developers.google.com/web/updates/2017/12/audio-worklet> - an article on the audio worklet, by one of the lead developers
* <https://googlechromelabs.github.io/web-audio-samples/audio-worklet/> - lots of audio worklet examples from the team developing the Web Audio API.
* <https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkletNode> - the main technical document on the audio worklet

## Assignment

Your submission should consist of your source code and recorded audio for each part of the assignment (see sections below), and a short report describing your work and answering the questions below. Where indicated, your report should depict results using figures such as waveforms and spectrograms.

**Questions, Part 1**

1. In milliseconds, how long is a block of samples processed by a worklet? Hint: First find the sample rate by inserting the line console.log(context.sampleRate) in the html file after the context has been declared.
2. An audio worklet node can be configured to have any number of outputs and any number of channels on each output.

See [developer.mozilla.org/en-US/docs/Web/API/AudioWorkletNode/AudioWorkletNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkletNode/AudioWorkletNode) . For instance,

myNode = new AudioWorkletNode(context,'my-generator',{outputChannelCount:[2]})

will create an audio worklet node with one output containing two channels.

Change the code in part 1 so that you have 2 output channels, i.e., generating stereo noise. To check that it is working, set one channel to constant 0 output and the other to noise, then vice versa. You may want to listen over headphones to confirm that noise is only heard on a different side for each case.

**Questions, Part 2**

1. Have gain be in decibels on the interface, and have the gain worklet accept parameter values in decibels and convert them in the process method. To convert from decibels to linear units you can use .
2. Rewrite this so that there is just one worklet which generates noise and applies a gain parameter to that noise.

**Questions, Part 3**

1. Add in a record button (see previous labs). Record the output for various values of the cut-off frequency. Load them in sonic visualizer and plot the spectrum and the spectrogram to show the effect of varying the cut-off frequency.
2. Now try using the Web Audio API’s BiquadFilterNode instead of the lowpass filter node that you just created, see <https://developer.mozilla.org/en-US/docs/Web/API/BiquadFilterNode> .Again, plot the spectrum and spectrogram. How does this compare with your node based on the worklet?
3. Another type of low pass filter is just a simple weighted average of the last two input samples, *y*[*n*]= *c*∙*x*[*n*] + (1-*c*)∙*x*[*n*-1]. Implement this, but note that the parameter is now the coefficient c, not the cut-off frequency. Try playing and recording the output for a few values of c. Load it into sonic visualizer and look at the spectrum to compare it with the other implementations. How does this compare with the BiquadFilterNode's low pass filter and your low pass filter from the previous question?

## Part 1. A noise worklet

1. Download lab3-resources.zip from QMPlus, unzip it and save it where you want to store your code.

2. Make sure you have the Atom text and source code editor, that you’ve installed the atom-live-server add-on, and that you have installed Chrome and set it to be your default browser.

3. In Atom, from the menu go to File -> Open Folder and select the ‘BasicNoise’ folder from within your unzipped ‘lab2-resources’

4. Create a simple html file, noise.html , with the following code;

<html>

<button onclick="StartAudio()">Start</button>

<script>

let context= new AudioContext();

function StartAudio() {context.resume()};

context.audioWorklet.addModule('basicnoise.js').then(() => {

});

</script>

</html>

5. The html file generates an error message because there is no noise.js file. So create an empty file, noise.js , and reload the html. It should not produce an error. But the script does not do anything special. It just does whatever is the function inside the braces { }, which is currently nothing, if it succeeds in adding the worklet module (it does, though there are no worklets there).

Aside: this is a bit of *asynchronous* javascript programming. Javascript is typically synchronous, waiting for each line to complete before moving to the next line. But here, it can continue with the rest of the code while also waiting for addModule to finish. Only once addModule has completed, does it do whatever is inside the braces. If you want to find out more, check out <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous> .

Now we can start creating the worklet.

6. Worklets must be in a separate file from where they are loaded. So the worklet will go in noise.js and be used in noise.html .

registerProcessor('noise-generator',class extends AudioWorkletProcessor {

process(inputs, outputs) {

for (let i=0;i<outputs[0][0].length;++i) outputs[0][0][i]=2\*Math.random()-1;

return true;

}

});

Lets break this down.

* The registerProcessor() method takes a string for the name of the processor to be registered, and the class definition.
* The processing happens within the processor method.
* The processor method must return true to keep the processor alive.
* The processor may have multiple inputs and outputs. Here, there are no inputs, like OscillatorNode and ConstantSourceNode, and only one output, so outputs.length = 1 .
* Each input or output may have multiple channels. Here, there is only one output with only one channel, so outputs[0].length = 1 .
* Audio ranges between -1 and +1, but the Math.random() function returns a number between 0 and -1, so the output of this function needs to be scaled.
* Worklets and nodes in WAA process blocks of length 128 samples, i.e. 128 samples / 48,000 samples per second = 2.666… milliseconds. So outputs[0][0].length = 128.

## Part 2. A gain worklet

The noise worklet was quite minimal.

1. Create a new folder with a gain.html file and a worklets.js file, and open this folder in atom.

2. In the worklets file, copy the text of noise.js, and in the html file, copy the text of noise.html .

3. Now edit gain.html to include the following

…

<p>Gain</p>

<input type="range" min="0" max="1" value="0.1" step="0.01" id="Gain">

<span id="Gain label"></span>

…

var GainLabel = document.getElementById("Gain label");

GainLabel.innerHTML = Gain.value;

…

let myNoise = new AudioWorkletNode(context,'noise-generator');

let myGain = new AudioWorkletNode(context,'gain-processor',{parameterData:{gain:0.1}});

Gain.oninput = function() {

myGain.parameters.get('gain').value=this.value;

GainLabel.innerHTML = this.value ;

…

I haven’t shown exactly where these lines go, but it should be clear if you understand the code.

4. Replace

myNoise.connect(context.destination);

with

myNoise.connect(myGain);

myGain.connect(context.destination);

5. Now add the gain worklet in the worklets.js file. You can add it before or after the noise worklet.

registerProcessor('gain-processor',class extends AudioWorkletProcessor {

// Custom AudioParams can be defined with this static getter.

static get parameterDescriptors() { return [{name:'gain',defaultValue:0.1}] }

process(inputs, outputs, parameters) {

const input = inputs[0],output = outputs[0];

for (let channel=0;channel<inputs[0].length;++channel)

for (let i=0;i<input[channel].length;++i) output[channel][i] = input[channel][i] \* parameters.gain[0];

return true;

}

})

See <https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkletProcessor/parameterDescriptors> for more on the static getter parameterDescriptors .

6. Try this out. You should now have a gain control for the noise. The parameterDescriptors is where one defines any audio parameters for the worklet. In this case, we just have a gain parameter, which multiplies the input signal. In the process method, one can access parameters as an array. This array might have a single element if the parameter is unchanging over a block of samples, or 128 elements if it might change. So one often writes the worklet with an if-then-else to handle the two cases. Here though, we just used the first value in the array.

## Part 3. A simple low pass filter worklet

There’s a little bit more with worklets that we want to show. So far, they can have audio parameters just like any other node. But they need to be able to do things like store values while in operation. As a simple example, suppose you want to create a node, using a worklet, that simply delays a signal by one sample. You could try making output of sample *n* equal to input of sample *n*-1, but what do you do for the first sample in a block of samples passed to the process method?

Here, we will create a low pass filter that attenuates frequencies beyond a certain cut-off frequency. A simple approximation for doing this is

*y*[*n*]= *c*∙*x*[*n*] + (1-*c*)∙*y*[*n*-1],

where *c*=2*fc*/*fs*, x is the input signal, y is the output signal, n is the sample number, *fc* is the cut-off frequency and *fs*is the sampling rate. Note that this is an approximation, but suitable for our purposes.

1. Create a new folder with a filter.html file and a worklets.js file, and open this folder in atom.

2. In the worklets file, copy the text of gain.js, and in the html file, copy the text of gain.html .

3. Now edit filter.html to replace the gain control with a frequency control (basically, just changing the words ‘Gain’ or ‘gain’ to ‘Frequency’ or ‘frequency’), and give it a maximum value of 10,0000 and default value of 1,000. Be sure to change the initial setting of the worklet as well as the default value of the control.

4. Now add the filter worklet in the worklets.js file.

registerProcessor('lowpass-filter', class extends AudioWorkletProcessor {

static get parameterDescriptors() { return [{name:'frequency',defaultValue:1000,minValue:0}]; }

constructor() {

super();

this.lastOut = 0;

}

process(inputs, outputs, parameters) {

let input = inputs[0],output = outputs[0],coeff;

let frequency = parameters.frequency;

for (let channel = 0; channel < output.length; ++channel) {

let inputChannel = input[channel],outputChannel = output[channel];

coeff = 2 \* Math.PI \* frequency[0] / sampleRate;

for (let i = 0; i < outputChannel.length; ++i) {

outputChannel[i]=inputChannel[i] \* coeff +(1-coeff)\*this.lastOut;

this.lastOut=outputChannel[i];

}

}

return true;

}

});

See <https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkletProcessor/parameterDescriptors> for more on the static getter parameterDescriptors .

Here, we’ve introduced the constructor, which is a special method for creating and initializing the object (the object in this case is the audio worklet processor). We did not need it in previous examples because there was nothing to initialize.

The super keyword is used to access and call functions on an object's parent. We need it so that we can use ‘this’. And now we can always store the last output sample, and give it an initial value of 0.

6. Try this out. You should now have a noise source where you can filter out high frequency components.