**ECS7012 Music and Audio Programming - Lab 1: HelloWorld**

Required resources:

* Code and sound examples contained in the **Lab1-resources.zip** file – available on QM+.
* **JUCE**
* **Visual Studio**
* **Sonic Visualiser** – a free analysis and visualization software package developed at QMUL, used in the last part of the lab.

The basics of the Web Audio API will be introduced from the ground up in this lab. Please note that this lab isn’t being graded as it only serves as an introduction.

The lab is divided into 3 parts;

1. Building a Hello World application- from the ground up, this shows how to code just about the simplest thing with JUCE; generating sound.
2. Tester Plugin- Often, when we are developing a plug-in, its useful to see how it works for various test signals used as input. These test signals are usually ones with very well understood behaviour, like pure noise or sinusoids, or known samples of speech or music. In this part, we build a Tester plugin.
3. Recording audio – Its useful to be able to record audio for the plugins and apps that we build. Often times, we can use the recording functionality of a DAW, but here, we build a simple plugin that records its input audio to a wav file.

The last two parts have some questions as parts of the steps that you should follow. Doing the exercises and typing in the code yourself is a lot better way for you to learn than just reading the assignment and running the code examples.

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. I encourage you to check it out since it is a thorough guide, will reinforce everything being demonstrated here today and will give you a more grounded understanding for future assignments.

Start by reading the introduction to the WAA to learn about concepts like: audio context, modular routing, audio nodes and audio graphs. Check out all the available source nodes, effect filters and audio destinations available from the WAA.

**Part 1. Hello World**

1. Download lab1-resources.zip from QMPlus, unzip it and save it where you want to store your code.

2. Install the Atom text and source code editor

3. Install the atom-live-server add-on: Atom -> Preferences -> Install.

4. Install Chrome and set it as your default browser. At the moment Chrome is the browser with the widest support for the WAA so we will always use it for testing and debugging in this module.

5. Install the Web Audio Inspector extension for Chrome: Window -> Extensions

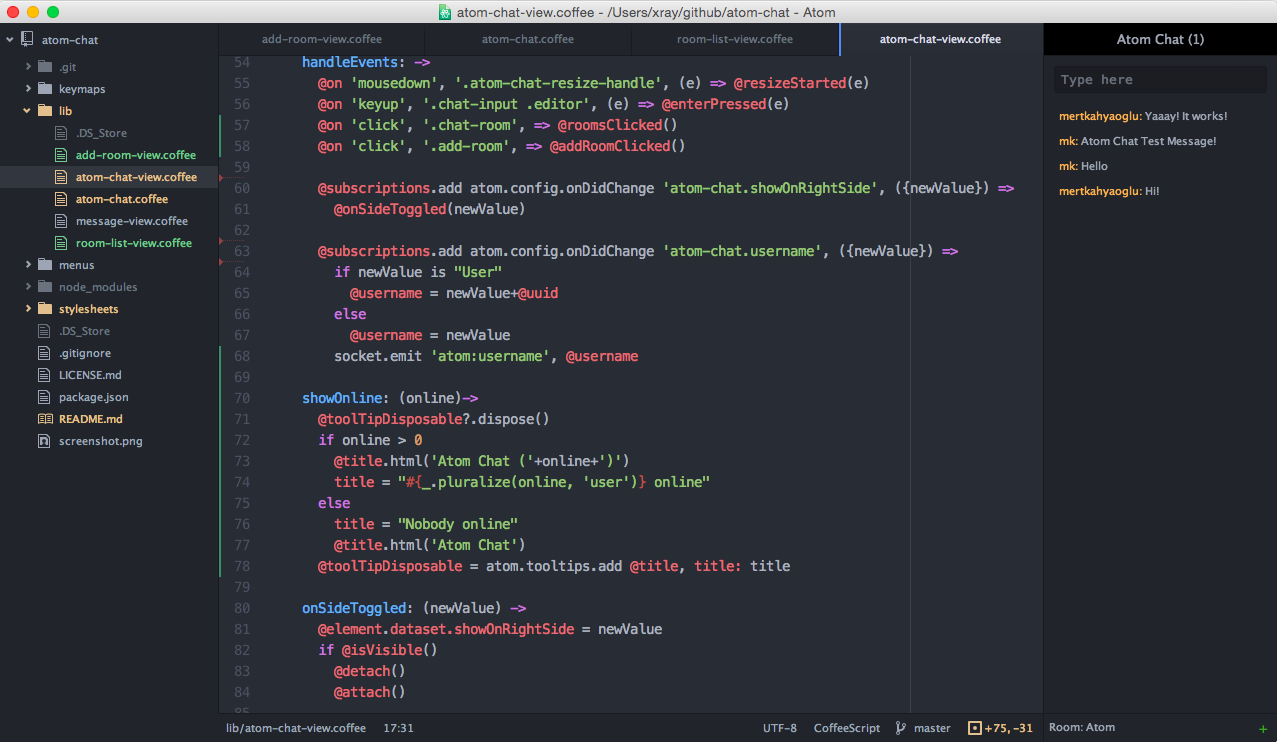
6. Keep the Developer Tools open in Chrome: View -> Developer -> Developer Tools 

Figure 1. Screenshot of Atom in use.

7. In Atom, from the menu go to File -> Open Folder and select the ‘1 Basic Tone’ folder from within your unzipped ‘lab1-resources’

8. From Atom’s menu, navigate to Packages 🡪 atom-live-server 🡪 Start server on the menu. This will open a tab in Chrome to render html within the current project.

9. In Chrome -> Developers Tools select the Web Audio tab to observe the audio graph. Once it’s defined,

you can click on the nodes to check their properties.

10. Select BasicTone-helloworld.html. This file is only a few lines of code, including the html. In the current version of Chrome, a web page usually requires some user interaction before it can generate sound. So the first line creates a button which, when clicked, ‘turns on’ the Web Audio API if it isn’t on already. The next few lines generates a sine wave of frequency 440 Hz, which is the default value. But it’s really annoying;

a. the sound appears automatically, without the user asking for it.

b. there’s no way to turn it off, other than turning the volume down or sound off on the computer (or laptop, tablet, phone…), leaving the web page or closing the browser.

c. the code is a single file with both javascript and html. This is ok for now, but gets really messy if there’s a lot of code.

So lets add a little bit more to make it better.

**Part 2. User Interaction**

1, First, lets separate the code from Part 1 into two separate files, one for the html and one for the javascript (JS). When the html file loads, it will embed the code from the JS file. Note that I didn’t add <html>. Since it’s a .html file, the browser knows that already, and renders it as html.

<button onclick="context.resume()">Start</button>

<script src="tone.js"></script>

**tone.html**

var context= new AudioContext()

var Tone = context.createOscillator()

var Amplitude = context.createGain()

Amplitude.gain.value = 0.2 // set oscillator's volume

Tone.start() // start generating signal

Tone.connect(Amplitude) // connect the oscillator to the gain node

Amplitude.connect(context.destination) // connect gain node to output

**tone.js**

So now when the html file loads, it will embed the code from the javascript file.

Note that I did not use semicolons at the end of each line. Javascript has automatic semi-colon replacement, so it doesn't matter if you write Tone.start(); or Tone.start() .

2. Now lets add some elements on the interface. We will make the button call a function to turn the sound on and off, and a slider to adjust the tone’s pitch where we will display its value.

<button onclick="StartStop()">Start/Stop</button>

<p>Frequency</p>

<input type="range" min=0 max=1000 step=1 value=440 id="Frequency">

<span id="FrequencyLabel"></span>

**tone.html**

3. Add another slider called Volume, to adjust the volume. Set its value to -0.2, its minimum to 0 and maximum to 1. This should be straight forward, so I won’t show it to you.

In javascript, we will access these elements directly by their IDs, which are made global variables by Chrome. So we have variables for all the relevant elements.

4. Now we need callbacks from the user interface elements. Lets put the following at the end of the javascript file.

StartStop.onclick = function() { }

Freq.oninput = function() { }

Volume.oninput = function() { }

So changing the sliders or clicking the button will call the relevant functions. At the moment though, the functions won’t do anything.

5. We want the Start button to turn the sound on and off. But the Web Audio API does not allow oscillators to be started more than once (I don’t know why, and certainly calling start() and stop() is the intuitive way to turn a sound source on and off). So there are a few ways around this. One can pause the whole audio context with context.suspend(), but here we’d like just to stop one node. One could create a new oscillator each time, which could be done by creating the oscillator, setting its values and connecting it all inside the StartStop function. But this involves a little extra code and thought. And it’s inelegant since all other nodes are created consistently elsewhere. Another solution is to connect or disconnect the oscillator rather than start or stop it. That’s what we will do.

The relevant code in the middle of tone.js is as follows.

var context = new AudioContext()

var Tone = new OscillatorNode(context)

var Amplitude = new GainNode(context,{gain:0.2})

Tone.start()

var Connected = 0 //Oscillator is not connected in the beginning, so silence

Amplitude.connect(context.destination) // connect the Gain node to the output

// Connects/Disconnects the oscillator to the graph

function StartStop() {

context.resume()

if (Connected == false) {

Tone.connect(Amplitude)

Connected = true

} else {

Connected = false

Tone.disconnect(Amplitude)

}

}

And finally, have the slider callbacks update the audio nodes and update the labels. Here it is for the frequency slider.

Frequency.oninput = function() {

Tone.frequency.value=this.value

FrequencyLabel.innerHTML = this.value

}

You should do the same for the Volume slider, remembering that the relevant parameter for a gain node is ‘gain’, not ‘frequency’.

6. Use an html <select> element to give the user the 4 options for the Oscillator type; sine (the usual default), square, sawtooth and triangle wave. See

<https://www.w3schools.com/tags/tag_select.asp> and <https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode/type>

7. Check your implementation against the source code in the User Interaction folder. You may format things slightly differently, but your code should look and behave similarly to the example provided.

8. Play around with this.

1. Change the oscillator properties. Make it a square, sawtooth or triangle wave, rather than a sine wave, and adjust the frequency and volume. Listen to how this changes the sound. This is also a way to check your implementation.
2. Display the gain in decibels, rather than linear units, so that the gain callback converts the value from db to linear. To convert *X* decibels to *Y* linear units, assuming 1 on the linear scale is the 0 dB reference, use *Y* = 10^(*X*/20).
3. Create two oscillators, and connect both of them to the gain control. First do this with both set to the same frequency. How has the sound changed. Then try varying their frequencies. What sort of sounds are produced for different frequencies (try it when the two frequencies are close together and try it when they are far apart)?
4. Now change the connections so that one oscillator is connected to the second oscillator, then the second oscillator connected to the gain control. What sort of sounds are produced for different frequencies? This is a simple form of amplitude modulation.
5. Finally, change the connections so that the first oscillator is connected to the frequency parameter of the second oscillator, with a line of code that might look like Tone1.connect(Tone2.frequency) . Then connect the second oscillator to the gain control. What sort of sounds are produced for different frequencies? This is a simple form of frequency modulation.