CS322 - Music Programming
Project Report

Voice Changer App

Abel Hii 12367631
Sam Johnson 12323371

Contents

Languages Used:

The application was designed with HTML5 and CSS3. All of the functionality is written using JavaScript and the WebAudio API in JavaScript.

Ideas/Chosen Idea:

For this project, we were given the freedom to design and build any kind of app that used audio processing, which gave us a lot of room to explore different ideas. We discussed a number of very different possible projects, our first being the Tenori-On. The Tenori-On is a Japanese electronic instrument with a grid of 16x16 switches or buttons which can be turned on in a number of different ways to produce sounds. To turn this instrument into an app seemed like a simple enough idea. There are a number of similar apps already available which gave us somewhere to look for inspiration.

Fig 1: Original Tenori-On

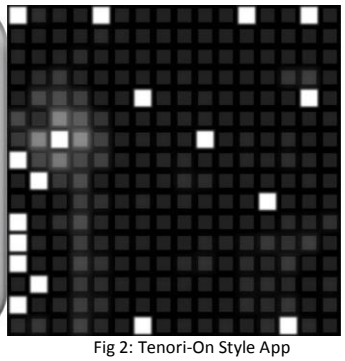

Fig 2: Tenori-On Style App

Another idea that we briefly discussed was a Theremin, of Russian origin, an instrument controlled by proximity sensors that monitor the position of the hands of the player. One hand controls pitch and the other, intensity. We briefly discussed controlling the app with keyboard or mouse input, or with a motion sensor but we did not discuss this idea much further.

Our decision was to make a voice changer. The original idea was to have four presets that we had hoped would mimic the voices of four different robot characters. At first we thought that we would allow the user to record a short message of up to fifteen seconds and then play it back with a robotic filter applied and then give the user the option to discard, save or set the clip as a notification sound on their device. The idea of having the app run on a web client or a mobile client while transforming the input live to a loudspeaker was also discussed. We decided that we would transform the audio input live and play it through the devices speakers with a number of adjustable filters and that the app would also record anything that was spoken into the microphone so that the user could save it to their device. We additionally wanted to add a graphical representation of the audio being played. We considered this to be a bonus feature which we would add if the project moved quickly enough and we had time.

Project Goals:

The project had the following goals which aims to create a fun and interesting application which would allow users to alter their voice in unique ways. To achieve this, we set goals for ourselves to follow:

- Create a simple UI to show the audio being altered which in turn helps with the UX
- Give the user full control over a number of different filters
- Enable live mixing of the user's voice
- Give the user the option to record and save the audio in .wav format

Project Plan:

There were many languages at our disposal which we considered for use in our project. In lectures, we had learned about WebAudio and saw that it was very useful for all kinds of audio manipulation. We had seen that these libraries allow the designer to get very creative. Having seen some demos, we really liked the potential it showed.

WebAudio is a high level API in the JavaScript language for the processing and synthesizing of audio within the web browser environment. It is primarily based on the use of an Audio Context container. The audio nodes in the Audio Context environment allow many different effects to be applied to the audio source before it is played to the output. There is also an Analyser Node in the API which we realised could be used to display a graphical representation of the audio live as it is processed.

The user interface would likely be designed with CSS and HTML5 with additional JavaScript to provide certain functions within the gui itself such as recording, saving and manipulating the filters.

The project was broken into eight incremental and coincidental sub-goals as follows:

1. Read audio input from microphone
2. Prepare blob file of the audio when it stops recording
3. Simultaneously playback microphone audio live to device speakers
4. Create at least three distinct audio filters
5. Design a simple GUI to show how the audio is being transformed
6. Make each filter controllable with a dial or slider in the user interface
7. Apply each filter to microphone audio before it reaches .wav file or speaker
8. Simultaneously record and play live input through device speaker with filters applied

Web Audio/Audio Context:

In the past, it was only possible to have audio within a web environment using a Flash plug-in. The HTML5 <audio> tag was the next step for introducing audio into the web environment. This ended the need for a flash plug-in, although it had its limitations. With WebAudio, it is possible to integrate the capabilities of the kinds of applications used for desktop audio processing, into the web environment without the need for flash.

In this API, all audio is processed within an Audio Context. The Audio Context is a very powerful tool to have in your arsenal. It seemed reasonable then that WebAudio would be the main tool that we used in our project.

A single Audio Context can support several audio sources with multiple channel layouts. Inside the Audio Context, nodes are created. Each audio node has specific properties and the nodes are linked together to form an audio routing graph. Within the scope of a single Audio Context, we could blend together sources to create new sounds. Each audio node can be used independently or the output of one node could become the input of the next to form a chain. For example, a gain node can be used to modify the intensity of a sound sample. We could simply connect an input to a gain node and then to the output node to amplify a sound, however we also have the ability to connect the gain node to for example, a convolver node to create an amplified reverberating sound.

The API also includes definitions for a channel splitter node and a channel merger node. The splitter node separates an audio source into a set of mono outputs which could be routed through various effector nodes

before being routed to the channel merger node. We spent a little time reading tutorials online to get familiar with the basic functions available.

Getting the Audio Source and Recording:

The first step was to take the live audio input from the user's microphone which was quite tricky as there was the issue with browser compatibility (explained in the "Problems Faced" section). Getting the Audio Source required us to get permissions from the user to use their microphone which is acquired by using the built in Audio Context function "*navigator.mediaDevices.getUserMedia({audio: true, video: false});"*

```
//**************INIT AUDIO**********//
navigator.getUserMedia = navigator.getUserMedia ||
                         navigator.webkitGetUserMedia ||
                         navigator.mediaDevices.getUserMedia
var p = navigator.mediaDevices.getUserMedia({audio: true, video: false});

p.then(function(mediaStream){
    var audio = document.getElementById('liveAudio');
    audio.src = window.URL.createObjectURL(mediaStream);
    audio.onloadedmetadata = function(e){
        //do something
        gotStream(mediaStream);
    }
});

p.catch(function(e) {alert("no mic detected"); console.log(e.name); }); //check for errors
//**************INIT AUDIO************//
```

Fig. 3: Connecting to the User's Audio Source

Once we get the user's permission we can get the stream from the microphone which needs to be initialised. It is initialised by getting the stream and creating an Audio Context for that stream. So "*audioContext.createMediaStreamSource(stream);"* is getting the stream from the microphone and creating a new audioContext for it to hold the audio nodes.

```
    if (!audioContext.createGain)
        audioContext.createGain = audioContext.createGainNode;
    inputPoint = audioContext.createGain();

    // Create an AudioNode from the stream.
    realAudioInput = audioContext.createMediaStreamSource(stream);
    audioInput = realAudioInput;
    audioInput.connect(inputPoint);
    //audioInput = convertToMono( input );
```

Fig. 4: Creating and Connecting the Audio Source

After we got the audio source and permissions working, recording the audio was a little easier to set up because there were a lot of sample code and resources already out there for recording with web audio. The main resource used for recording this was mattdiamond's Recorderjs which primarily uses webaudio to record sound on the web (https://github.com/mattdiamond/Recorderjs).

The general idea of how we set up recording and saving is that when the user clicks the record button it starts recording the sound which is then saved in a blob file when the user stops recording. The blob file with the recorded audio, will then be able to export in .wav format when the user clicks on the save button.

```
function toggleRecording( e ){
    if (e.classList.contains("recording")) {
        // stop recording
        audioRecorder.stop();
        e.classList.remove("recording");
        audioRecorder.exportWAV( doneEncoding );
    } else {
        // start recording
        if (!audioRecorder){
            return;
        }
        e.classList.add("recording");
        //audioRecorder.clear();
        audioRecorder.record();
```

Fig. 5: Code Snippet for Recording

```
function saveAudio() {
    audioRecorder.exportWAV( doneEncoding );
    // could get mono instead by saying
    // audioRecorder.exportMonoWAV( doneEncoding );
}
```

Fig. 6: Class to Save the Recorded Audio

Filters:

Filters in WebAudio are defined as audio nodes and must be created within an Audio Context. The following snippet of code shows an example on how we could create nodes for distortion and gain and connect them between the source and the destination audio:

```
var audioCtx = new (window.AudioContext || window.webkitAudioContext)();

//create the nodes

var distortion = audioCtx.createWaveShaper();
var gainNode = audioCtx.createGain();

// connect the nodes together

source = audioCtx.createMediaStreamSource(stream);
source.connect(distortion);
distortion.connect(gainNode);
gainNode.connect(audioCtx.destination);
```

Fig. 7: Code Snippet for Creating an Audio Node

There are many other filters that are available in the API such as detune, highpass and lowpass, high shelf, low shelf and convolver to name a few. These particular types of filter can be defined using a Biquad fIlter node within Audio Context. Once we have created a number of effector nodes, we can connect them into an audio routing graph. These structures will be extremely useful for our project and we hope that they will allow us to create various effects.

Fig. 8: Simplified Architecture of Voice Changer

The filters we used in the end were the Biquad Filter, Delay filter and the Waveshaper filter. I created the filters by getting the audio context from the main.js file where the initial audio source and authentication request is made "*AudioContext.createBiquadFilter()"*. and connected the filters to the audio source also using a class from the main.js file "*audioInput.connect(filter)*".

```
112  FilterOne.play = function(){
113      //Don't have to create the source because it's being taken live from the user's microphone from main.js
114      // Create the filter.
115      filter = audioContext.createBiquadFilter();
116      //filter.type is defined as string type in the latest API. But this is defined as number type in old API.
117      filter.type = (typeof filter.type === 'string') ? 'lowpass' : 0; // LOWPASS
118      //Set default values for frequency, q and detune values:
119      filter.frequency.value = 3000;
120      filter.Q.value = 30;
121      filter.detune.value = 3000;
122      // Connect source/audioInput to filter, filter to destination.
123      audioInput.connect(filter);
124      filter.connect(audioContext.destination);
125
126      //record the filtered audio
127      audioRecorder = new Recorder(filter);
128
129      // Save audioInput and filterNode for later access.
130      this.audioInput = audioInput;
131      this.filter = filter;
132
133      //updateAnalysers();
134  };
135
```

Fig. 9: Code Snippet to Create the Biquad Filter

Once we figured out how to create and connect the Biquad Filter, connecting the other two filters was easy to do as it was simply a matter of connecting one filter to the next and then to the destination output i.e. the speaker.

The general form of the call to connect a filter, or in fact any audio node within the Audio Context to another node is as follows: "*nodeOne.connect(nodeTwo);"*

For example, in our source code, we connect the microphone input (*nodeOne*) to our Biquad Filter (*nodeTwo*) called "*filter"* as follows: "*audioInput.connect(filter);"*

The other filters also follow this general layout.

Problems Faced:

The first problem we encountered was right at step one. Getting the microphone to work proved to be pretty tricky. The problem we encountered was with the WebAudio built in function *navigator.getUserMedia()* which is used to give access to the user's microphone. After doing some research I found that the reason why we couldn't get access to the microphone was due to the fact that the function was deprecated;

"This feature has been removed from the Web standards. Though some browsers may still support it, it is in the process of being dropped. Do not use it in old or new projects. Pages or Web apps using it may break at any time." (https://developer.mozilla.org/en/docs/Web/API/Navigator/getUserMedia).

I discovered that we should have been using *MediaDevices.getUserMedia()* instead of *navigator.getUserMedia().* The next problem we encountered with *MediaDevices.getUserMedia()* was the fact that it was an experimental technology and still in development. Which meant that it would only work on browsers that supported it such as Firefox. So we chose to carry on and worry about it until later.

"Because this technology's specification has not stabilized, check the compatibility table for the proper prefixes to use in various browsers. Also note that the syntax and behavior of an experimental technology is subject to change in future versions of browsers as the spec changes." (https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia).

Usually if we were to make the app compatible with every browser this piece of code would work but because *MediaDevices.getUserMedia()* is still a new technology not every browser has it implemented yet.

```
navigator.getUserMedia = navigator.getUserMedia ||
                         navigator.webkitGetUserMedia ||
                         navigator.mediaDevices.getUserMedia
```

Fig. 10: Three Different Function Calls for Connecting to User's Microphone

Once the microphone was working, saving the audio to a file was relatively easy and so playing it was also straightforward. When the user presses the record button, the app begins to process the audio through a buffer. Upon pressing the Stop button, a blob file of the audio is created and stored in cache memory. If the user chooses to save the file, it is downloaded from this cache file. The audio is also played back from this location. The next big problem arose with creating and connecting the filters. We did a lot of reading on Middle Ear Media and Mozilla Developer Network. Both of these websites provided comprehensive tutorials on WebAudio and audio processing however we still had a lot of difficulty getting multiple filters to work together. We were able to have multiple filters and use one at a time, but putting them together did not seem to have any further effect on the audio. It was as if the first filter was changing the audio and then the others did not have any effect. This really held up the project. We spent a lot of time reading about creating filters in webAudio but we struggled with this quite a lot.

The graphical picture created by the analyser node only seemed to register only on the low frequencies. By increasing the sample rate we were able to improve it slightly.

App Features:

Before you start using the app it is recommended to have headphones plugged in and an external mic for the best experience.

There are three main effects for the user to play with; Biquad Filter, Delay Filter and a Waveshaper Filter.

- The Biquad filter has the most functions applied to it where you have the option to alter the frequency, quality, detune and gain of the audio.
- The Delay Filter will echo the audio to make you sound like you're in a cave or a large room and it gives you the option to change the time of the echo.
- The Waveshaper Filter will make your voice sound unrecognisable if you ever want to disguise your voice and it gives you the option to control the range of the disguise.

We also gave the user the option to record their live audio input and save that as a wav file.

In addition, to make the app more aesthetically pleasing, we added the graphical representation of the sound wave which was powered by the Analyser node which we have described above. It simply passes the audio, unchanged to the next node while reading the values of the audio samples. For the best user experience, a good quality microphone is required to allow the analyser node to create the most colourful picture and also allow the filter nodes to have a greater effect.

Below is a screenshot of what the final version of the app for this project looks like:

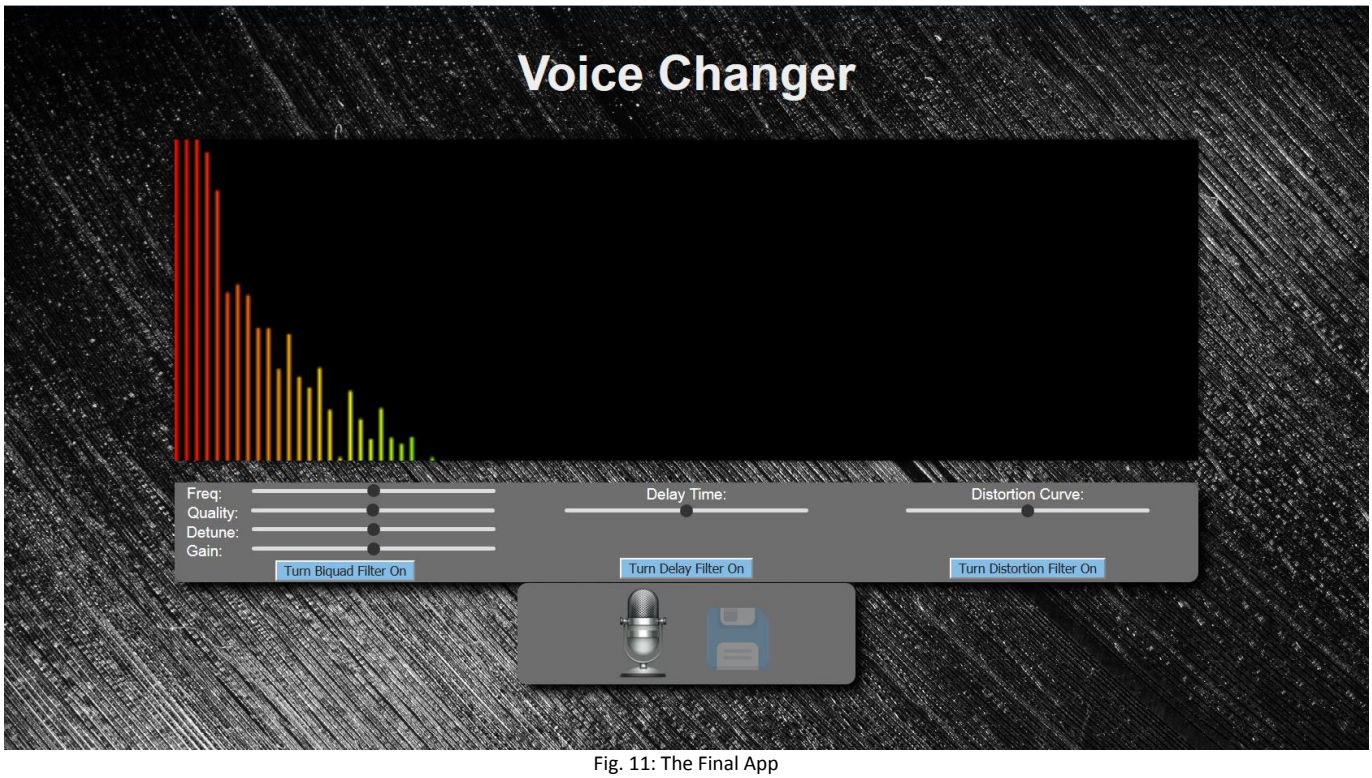

Fig. 11: The Final App

Possible Improvements:

There are a number of matters we could address in order to improve our app. The first and most obvious one is how it looks. We need to make it prettier and more interesting to look at.

We had discussed the idea of having preset values for each filter to create voices of various robotic characters. These could be represented by buttons displaying the character's face or similar. A mockup for such a version of the application is shown below:

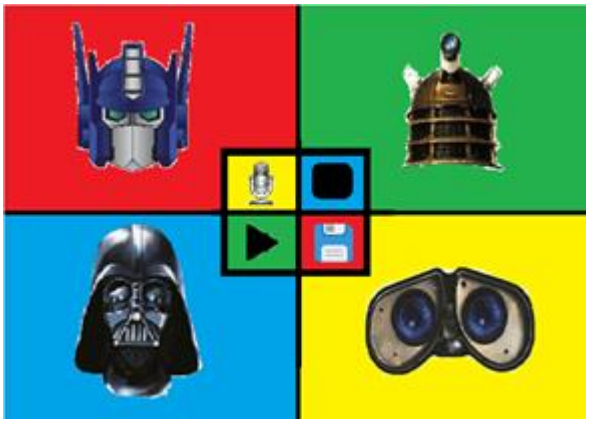

Fig. 12: Mock Up of Potential User Interface

Clicking one of these preset configurations would cause the sliders to snap to the relevant positions but also allow the user to manually move the sliders by swiping to a second screen. This is something we can try to do later once we have mastered using the sliders and filters.

Conclusion:

Since taking part in this application development, we have become more aware of the power of JavaScript. The WebAudio API is a very powerful tool if you can get your head around it. We feel that we have much more to learn about the power and capabilities of this API and JavaScript in general.

The Voice Changer app was thought to be a simple enough project that would help us learn the basics of audio filtering and manipulation. Although there are a couple of problems with the app, it seems to work well to a degree. We feel as though it was a good first attempt at using WebAudio.

We were able to get a graphical display of the audio working on the app. It shows the audio as it is heard on a visual analyser graph in full colour which makes the app a little bit more interesting to look at. This was created using the Analyser node which the audio passes through unchanged while the Analyser takes information about the signal. Of the goals we set out to achieve we have successfully achieved the following:

- Audio recording from microphone
- Playback audio to the user
- Allow user to save their audio recording
- Allow user to control filters
- Apply filters in real-time

Additional Comments:

Upon testing, we have discovered that the app may not work in some browsers. Although the WebAudio API is supported by the five most common browsers, the app has proven temperamental when we have tested it on different browsers. It has been built around Mozilla Firefox as it was our browser of choice.

It could be worthwhile to continue working on the app to ultimately release it as an android and web app. On the Google Play Store alone, the top five free voice changer apps have collectively over thirty-one million downloads.

With such popularity, there may be an opportunity to have an advertisement banner somewhere within the app to generate income with Pay per View or Pay per Click ads.

Sources of Information:

https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

http://middleearmedia.com/web-audio-api-basics/

https://github.com/mattdiamond/Recorderjs

http://webaudio.github.io/web-audio-api/

https://webaudiodemos.appspot.com/AudioRecorder/index.html