Fun-thesizer

Richard Shen and Danni Tang

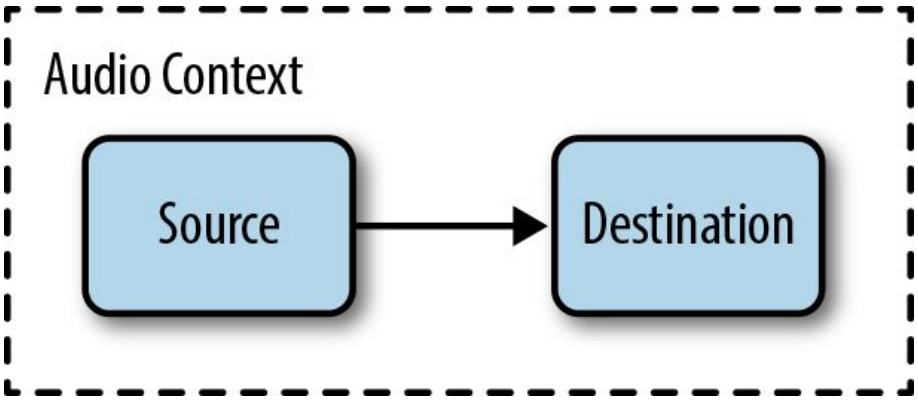
Abstract - **Visualization has existed in order to see data in context, inspire but most importantly, provide a clear platform to analyze and understand connections. Audio effects are normally mathematically intensive and, for many, difficult to comprehend through equations alone. In order to complement understanding, this project explores the capability of web applications in creating a visual representation of popular audio effects found on modern synthesizers. This project leverages the web audio API [1] and tuna.js [2] library in order to achieve this.**

I. Introduction

This audio signal processing project consists of several main components: the Web Audio API, the Qwerty Hancock Keyboard, the Tuna Library, and the Visualizer. All parts of the project were programmed in HTML, CSS, and JavaScript. The main goal of the was to build a musical keyboard that can play accurate pitches on an actual piano when pressing down a key, implement audio effects learned in class, and draw a visualizer which is a Fast-Fourier Transform of the sound being played. The final build of the project is hosted at `souloist.github.io/Fun-thesizer`. The build was mostly successful, with a few minor bugs (such as the visualizer bars not going down when first playing the keyboard), and lacks some user functionalities. However, the audio effects and visualizer were implemented perfectly, which are the core components of the project.

II. Web audio API

The web audio API made real-time processing possible on the web. In order to leverage the API, an audio context must be created. The web audio API utilizes a network of audio nodes in order to achieve audio processing.



The audio context is simply a direct graph of audio nodes. As the audio stream flows through the graph, it can be either modified or analyzed before it reaches the destination node and your speakers. There are four different kinds of audio nodes: source nodes, modification nodes, analyzer nodes, destination nodes. The source node contains sound sources such as audio buffers, built-in oscillators and live audio input. The modification node modifies the audio stream as it pass through. These changes can include filters, convolvers (for creating reverb effects) or panners. The analysis node can retrieve real-time frequency or time information in order to create an oscilloscope or other visualizations. Finally, the destination node is the audio output which is most commonly the speakers.

In order to utilize the web audio API, an audio context class needs to be created. Since the web audio API is not yet compatible with all browsers, initialization is different for each browser.

```
var contextClass = (window.AudioContext ||
  window.webkitAudioContext ||
  window.mozAudioContext ||
  window.oAudioContext ||
  window.msAudioContext);
if (contextClass) {
  // Web Audio API is available.
  var context = new contextClass();
} else {
  // Web Audio API is not available. Ask the user to use a supported browser.
}
```

For example, as seen above, Chrome is `window.AudioContext` while Firefox is `window.mozAudioContext`

After initialization, the audio context class can now create the various audio nodes. In order to connect nodes, it is simply `first_node.connect(second_node)`

```
// Create the source.
var source = context.createBufferSource();
// Create the gain node.
var gain = context.createGain();
// Connect source to filter, filter to destination.
source.connect(gain);
gain.connect(context.destination);
```

III. Qwerty Keyboard

The Qwerty Hancock keyboard [3] is a free musical keyboard written in JavaScript and can be easily embedded in HTML. The source file draws the keyboard, maps frequencies to each key, and programs functions for producing sound when you press a key down or mouse click a key. These functions (`keyboard.keyDown` and `keyboard.keyUp`) can be readily called in other JavaScript files. The Web API nodes are inserted into the `keyboard.keyDown` function in this order: Source, Gain, Effect, Analyser, and Destination. The keyboard can also be customized to start on a certain key, span any number of octaves, and embedded with a specified height and width.

The source node created in the `keyDown` function can be readily selected by users. Four basic periodic waveforms can be called (sine, square, sawtooth, and triangle); while a custom periodic waveform can also be made, we chose not to make one for the sake of simplicity [9]. The 4 basic waveforms as single oscillators are available as source nodes, and we also added the option of double oscillators by combining 2 waveforms and connecting them to one gain node. Making pairs from the 4 waveforms will result in 6 combinations without repetition, so the user has a total of 10 different source nodes to choose from.

Example code:

```
75    keyboard.keyDown = function(note, frequency) {
76      var osc = context.createOscillator();
77      osc.frequency.value = frequency;

92        osc.detune.value = -10;
93        osc.connect(masterVolume);
```

```
98          masterVolume.connect(analyser);

126         analyser.connect(context.destination);
127
128         oscillators[frequency] = osc;
129         osc.start(context.currentTime);
130
131         Visualize(analyser);
```

IV. Tuna.js Library

The tuna library is an open-source library created by the github user theodus [2] to allow for an easy way to implement audio effects. In order to leverage the Tuna library, a Tuna object must be created. This object will then create the audio effects as nodes which can be connected into the audio context graph. The library hosts a wide variety of audio effects, and we chose a handful of effects that were taught in class to implement in the final project. The effects implemented were: Chorus, Tremolo, Ping-pong, and Phaser. These audio effects can be easily implemented to any source node; switching audio effects is a seamless process in the final build.

Example code:

```
38  // Objects (effects) created with the tuna library
39  var chorus = new tuna.Chorus({
40      rate: 1.5,
41      feedback: 0.2,
42      delay: 0.0045,
43      bypass: 0
44  });
```

V. Visualization

Visualization was made possible by leveraging the <canvas> tag.

```
function Visualize(analyser) {
    var canvas = document.getElementById('canvas'),
```

Once a <canvas> tag is created with specific dimensions, it is possible to draw on the canvas by inputting the id of the tag into the document.getElementById() function.

```
ctx = canvas.getContext('2d'),
gradient = ctx.createLinearGradient(0, 0, 0, 300);
gradient.addColorStop(1, '#9331CB');
gradient.addColorStop(0.5, '#0CD7FD');
gradient.addColorStop(0, '#f00');
```

You can choose for the canvas to be in either 2-dimensional or 3-dimensional space. In this case, it was 2-D. A gradient can also be chosen and changed depending on whichever hex values you prefer.

```
var drawMeter = function() {
    var array = new Uint8Array(analyser.frequencyBinCount);
    analyser.getByteFrequencyData(array);
```

The visualizer took as input the frequencies of the audio stream as an array of 4096 values. Those values are then mapped to 79 bars and plotted. The magnitude of a specific frequency is proportional to the height of the canvas.

```
ctx.fillRect(i * 15, cheight - value, meterWidth, capHeight);
```

Due to the fact that the web audio api is not adopted on all browsers, the performance of the visualizer is dependent on the browser. If there is no `cancelAnimationFrame()` function in that browser, the visualizer will result in a significant memory leak. It is important to realize that in order for visualization to be possible, bars and caps must be drawn on the canvas. Without the `cancelAnimationFrame()`, memory will not be freed after every frame and the program will lag.

VI. Interface Design

Due to our intermediate understanding of web development, the team decided to embed everything in one HTML page. The interface of the final build consists of menus allowing the user to choose source nodes and audio effects at the top, the visualizer in the middle, and the keyboard below the visualizer. This is due to the fact that the visualizer and keyboard are the most aesthetically interesting objects on the webpage and we want to draw visitors' eyes to the center of the page. The visualizer was placed on top of the keyboard because the visualizer bars creates empty space and should go on top. The menus were placed at the very beginning so it is more intuitive for the user to first choose a source node and/or waveform (since the keyboard will not play without a source).

VII. Challenges

The most notable challenge was to understand how to implement the web audio API which was written in javascript. Previously, the group had only used python and the Pyaudio module in order to process real-time signals. Particularly, utilizing closures in javascript was difficult to understand but proved to be a valuable technique. A closure is the local variables of an inner function which has access to the variables of the outer function and global variables. As a result, the local variables can remain accessible even after returning from the function. Another challenge was linking the keyboard and the visualizer.

VIII. Improvements

While the final project makes good use of creating source nodes, implementing audio effects to them, and drawing the FFT visualization, we believe that it lacks user functionalities. For example, the user currently cannot change the volume, change the parameters of the audio effects (such as changing delay time, etc), combine two or more different audio effects, or record and save audio files of them playing the keyboard. These would make the project a more appealing web application if more functionalities were available. Other improvements include changing the source node to a more realistic piano sound; currently it is an oscillator node in mono and does not sound very interesting.

IX. References

[1] Web Audio Api - https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
[2] Tuna Library - https://github.com/Theodeus/tuna
[3] Qwerty Hancock Keyboard - https://github.com/stuartmemo/qwerty-hancock
[4] Visualizer - https://github.com/wayou/HTML5_Audio_Visualizer
[5] Web Audio Guide - https://github.com/mmckegg/web-audio-school
[6] Web Audio API - Boris Smus - http://chimera.labs.oreilly.com/books/1234000001552/index.html
[7] Web Audio Visualization - http://www.smartjava.org/content/exploring-html5-web-audio-visualizing-sound
[8] Advanced Web Audio API usage - https://developer.tizen.org/community/tip-tech/advanced-web-audio-api-usage

[9] Oscillator Node documentation - https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode/type