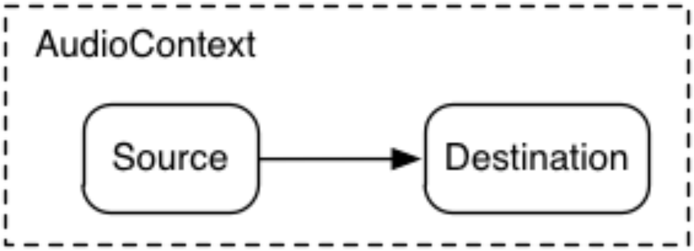
Web Audio Intro

1) Web Audio API

The Web Audio API specification describes a high-level JavaScript API for processing and synthesizing audio in web applications. The top level class of the API is `AudioContext`

To play sounds using web audio, we connect a sound source to a destination.

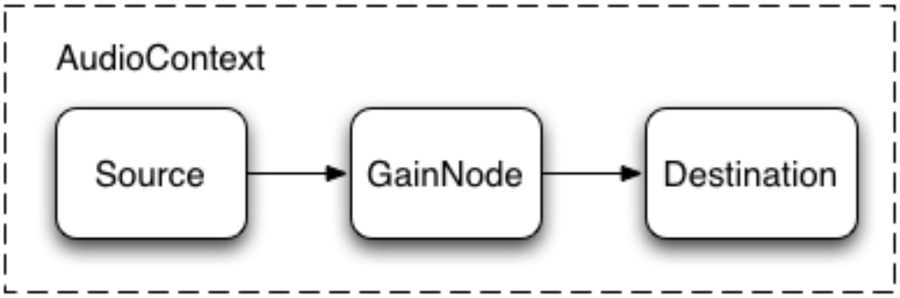


A simple audio graph

The primary paradigm used by **WebAudio** is that of an *audio routing graph*, where `AudioNode` objects are connected together to define the overall audio rendering.

Below we have an example of an AudioContext graph with a `GainNode` (volume) between the source node and the destination node. The `AudioNode` instances you place between the source and the destination allow us to manipulate and analyze the audio stream.

An audio graph for controlling the volume of a sound



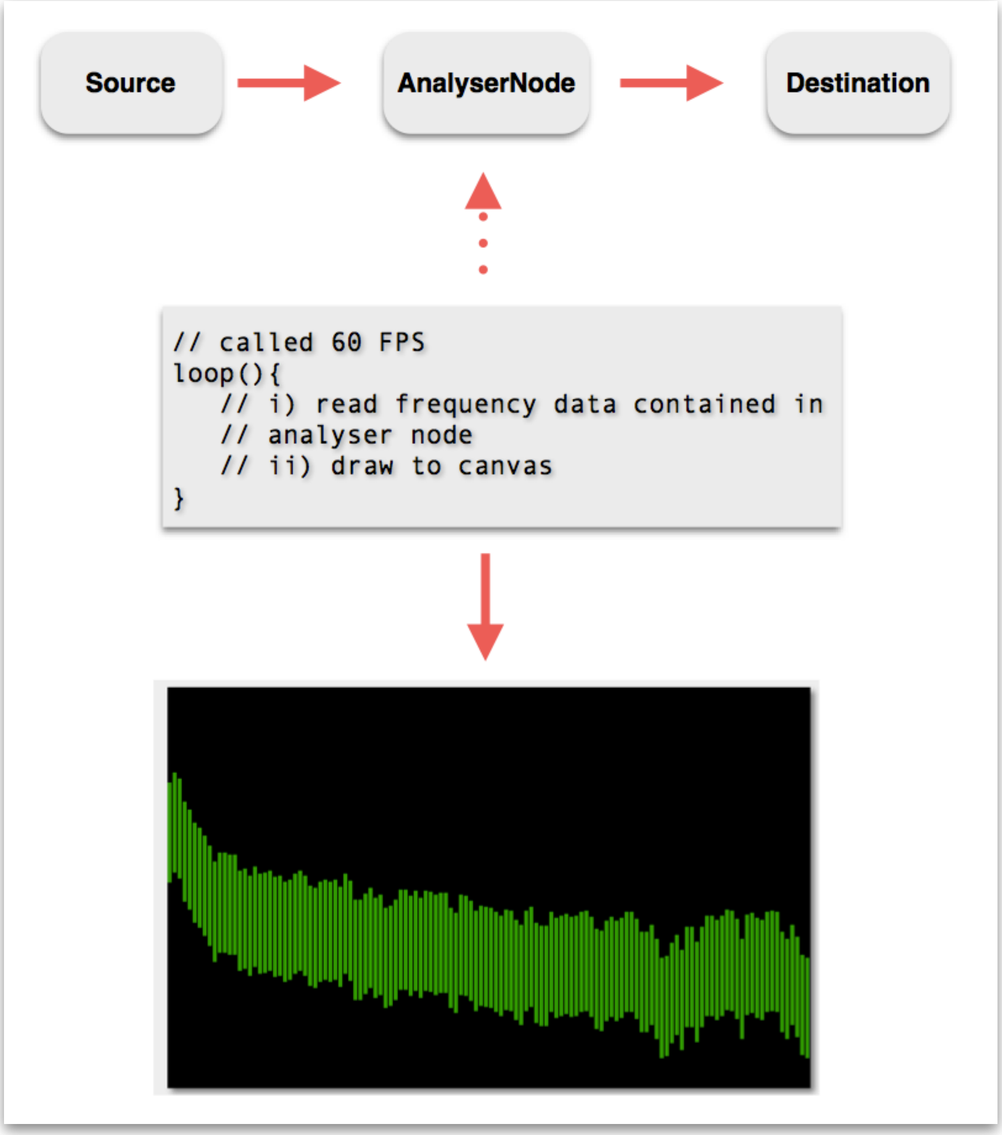
In the ICE, we will be placing an `AnalyserNode` between the source and the destination. This analyzer node will not actually change the sound, but it will allow us to analyze it.

http://webaudio.github.io/web-audio-api/#the-analysernode-interface

The `AnalyserNode` gives us access to the *frequency data* of the sound, as well as the *waveform data* (think "change in values" like an oscilloscope). We will then read and visualize this data by drawing onto our canvas tag.

Note the British spelling of "Analyser" (I thought WE won the revolutionary war!?)

An audio graph for analyzing the frequencies of a sound



```
// called 60 FPS
loop(){
    // i) read frequency data contained in
    // analyser node
    // ii) draw to canvas
}
```

2) What does the audio frequency data look like?

Here’s the raw frequency data (*byte frequency* data) array as seen in the debugger. xWe’ve asked for 64 samples that evenly sample the frequency range of the sound from 0 to 21050 Hz (21.05 kHz)

Array element 0 represents: 0 - 329 Hz
Array element 1 represents: 329 - 658 Hz
Array element 63 represents: 20721 - 21050 Hz

The values in the array elements represent the loudness of each frequency *bin*. They are an average of all of the frequencies in the range of that bin.

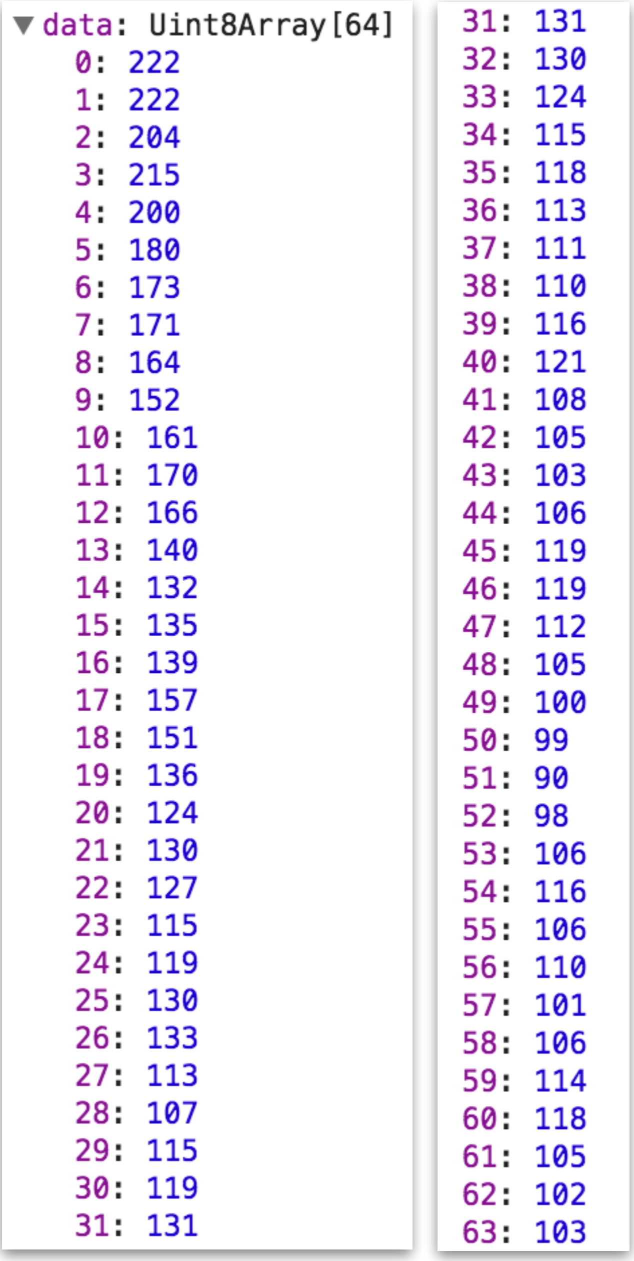
The range of the values is 0-255, where 0 is no loudness, and 255 is the maximum loudness.

(You can also request the byte frequency data as percentages if you wish, from 0-1.0)

Here we are sampling this data 60 frames per second. With most sounds, the contents of this array will therefore change every 1/60th of a second as the `<audio>` player progresses through the sound.

Note:

You can also get *waveform data* that represent the change in the frequency bins, similar to what you might see in an oscilloscope.



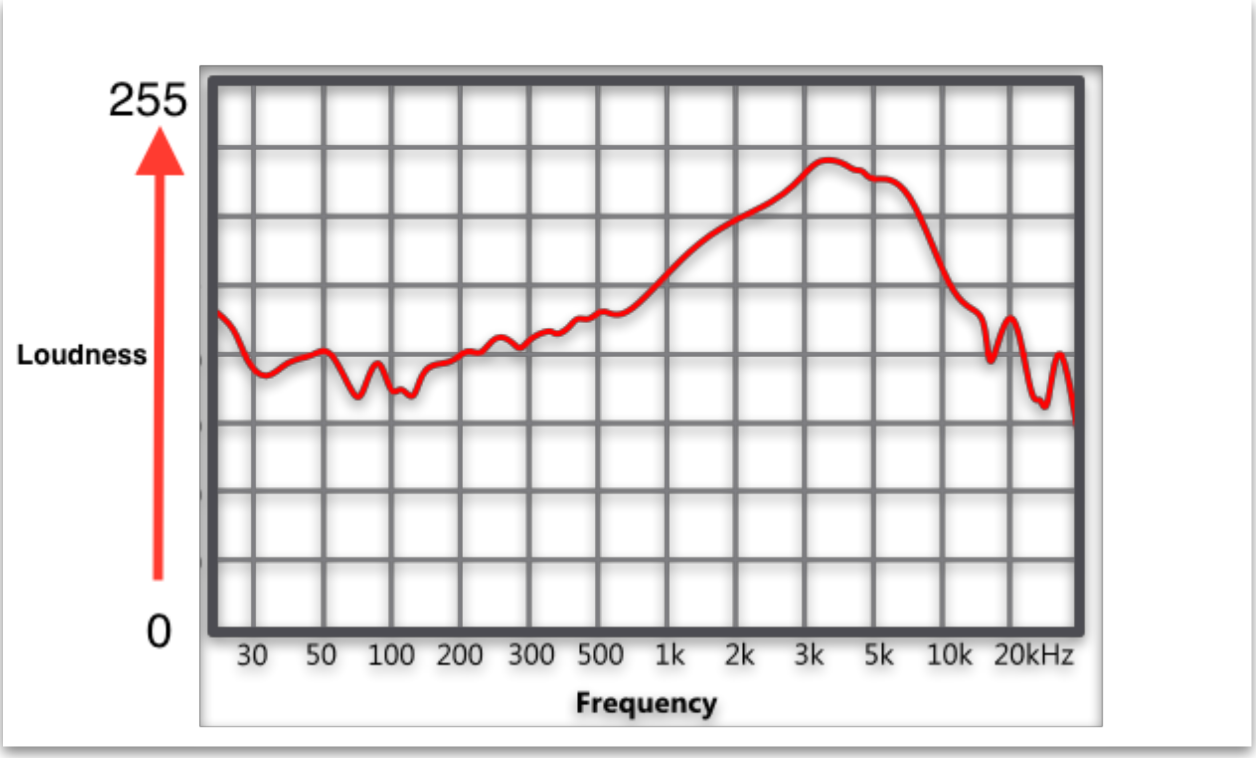
```
▼ data: Uint8Array[64]
    0: 222
    1: 222
    2: 204
    3: 215
    4: 200
    5: 180
    6: 173
    7: 171
    8: 164
    9: 152
    10: 161
    11: 170
    12: 166
    13: 140
    14: 132
    15: 135
    16: 139
    17: 157
    18: 151
    19: 136
    20: 124
    21: 130
    22: 127
    23: 115
    24: 119
    25: 130
    26: 133
    27: 113
    28: 107
    29: 115
    30: 119
    31: 131
```

```
31: 131
32: 130
33: 124
34: 115
35: 118
36: 113
37: 111
38: 110
39: 116
40: 121
41: 108
42: 105
43: 103
44: 106
45: 119
46: 119
47: 112
48: 105
49: 100
50: 99
51: 90
52: 98
53: 106
54: 116
55: 106
56: 110
57: 101
58: 106
59: 114
60: 118
61: 105
62: 102
63: 103
```

To see how to access this waveform data, look for the following line of commented out code in the ICE:

```
analyserNode.getByteTimeDomainData(data); // waveform data
```

Below is an example of what we might get from one of these 1/60th of a second snapshots of the byte frequency data.



If we sample the audio data, draw points to the screen, and update it every 1/60 of a second, we'll get animation.

Common frequency ranges:

Normal Speech falls between 500 Hz to 2 kHz
- Low frequencies are vowels and bass
- High frequencies are consonants

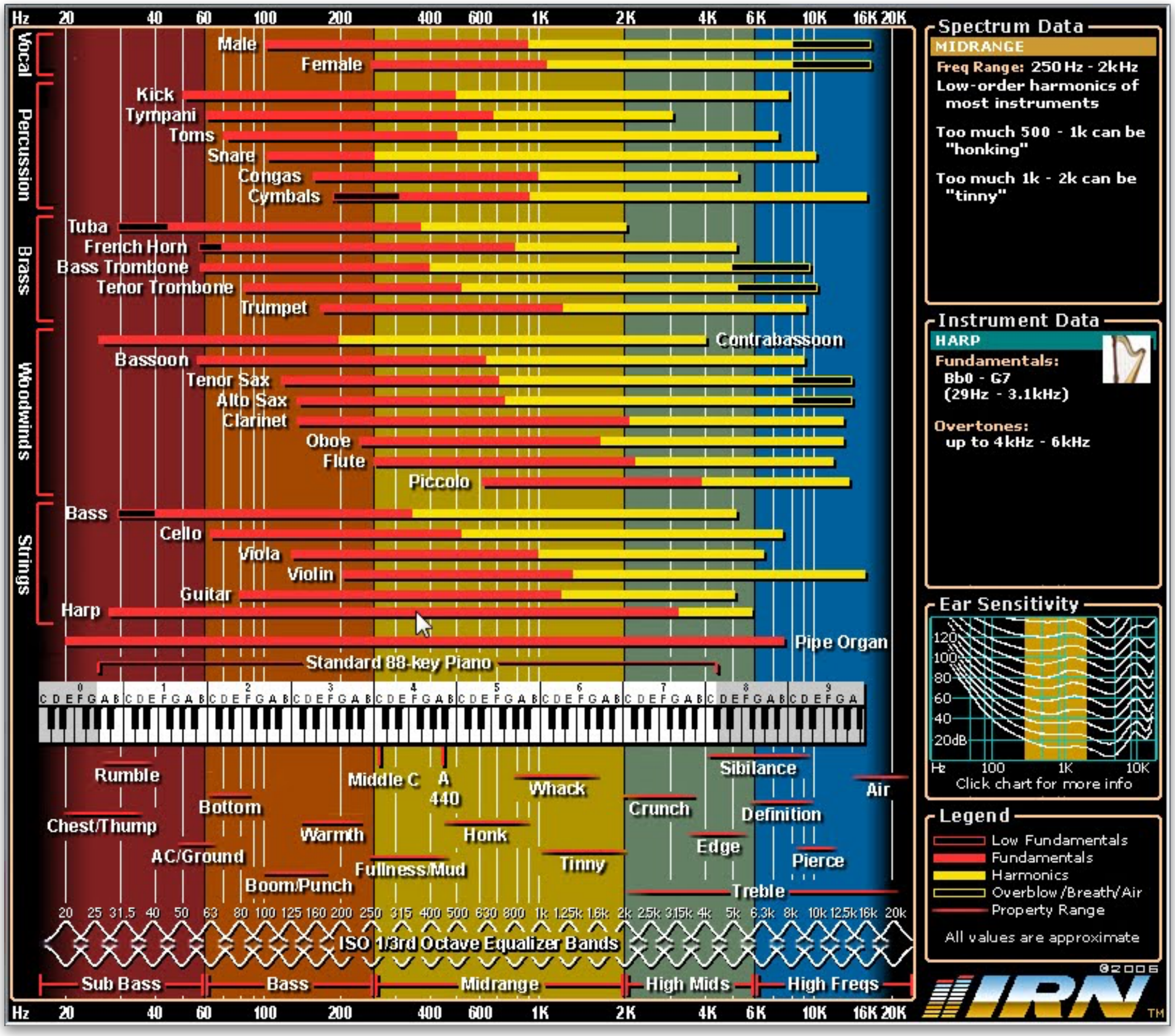
Standard Piano Keyboard: 27.5 Hz to 4186 Hz

Middle C: 261.6 Hz

High-pitched Scream: 3000 Hz

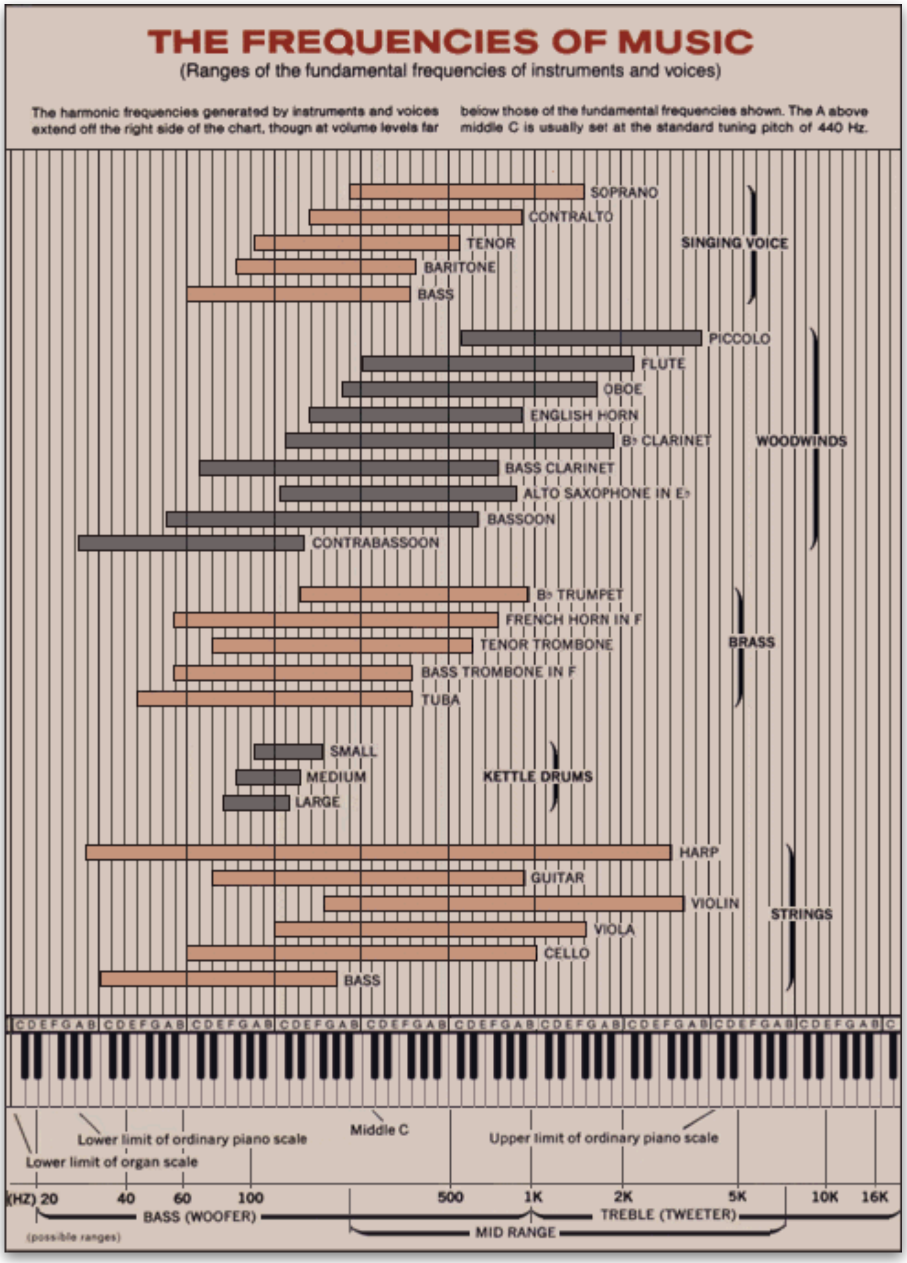
Instrument Chart:

https://www.gearslutz.com/board/electronic-music-instruments-electronic-music-production/817538-instrument-frequency-chart-electronic-music-what-goes-where.html



Another Chart:

https://www.gearslutz.com/board/electronic-music-instruments-electronic-music-production/817538-instrument-frequency-chart-electronic-music-what-goes-where.html



THE FREQUENCIES OF MUSIC

(Ranges of the fundamental frequencies of instruments and voices)

The harmonic frequencies generated by instruments and voices extend off the right side of the chart, thougn at volume levels far below those of the fundamental frequencies shown. The A above middle C is usually set at the standard tuning pitch of 440 Hz.

3) Issues with viewing the visualization

Because of browser security restrictions, when you are running the starter HTML page off of a hard drive (as opposed to a web server) you will get an error in the console:

MediaElementAudioSource outputs zeroes due to CORS access restrictions

CORS stands for “Cross-origin Resource Sharing” - the browser doesn’t want to let the audio element load local files from the hard drive.

Solution #1 - Put all the files up on a web server like gibson and run and edit them there. Due to the low amount of storage we have on gibson, you may have to limit yourself to just one sound file.

Solution #2 - Use an IDE like Brackets - which creates a local web server for you to run your code on

Solution #3 - You can also create a web server using Python on your local machine: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/set_up_a_local_testing_server

Solution #4 - Tell your browser to turn off the access restrictions. Here are some ideas on how to do that: https://blog.nraboy.com/2014/08/bypass-cors-errors-testing-apis-locally/

Web Audio Visualizer ICE

1) A) Test the ICE - it's all ready for you. Be sure to read over the commented code - we'll also walk through it in class.

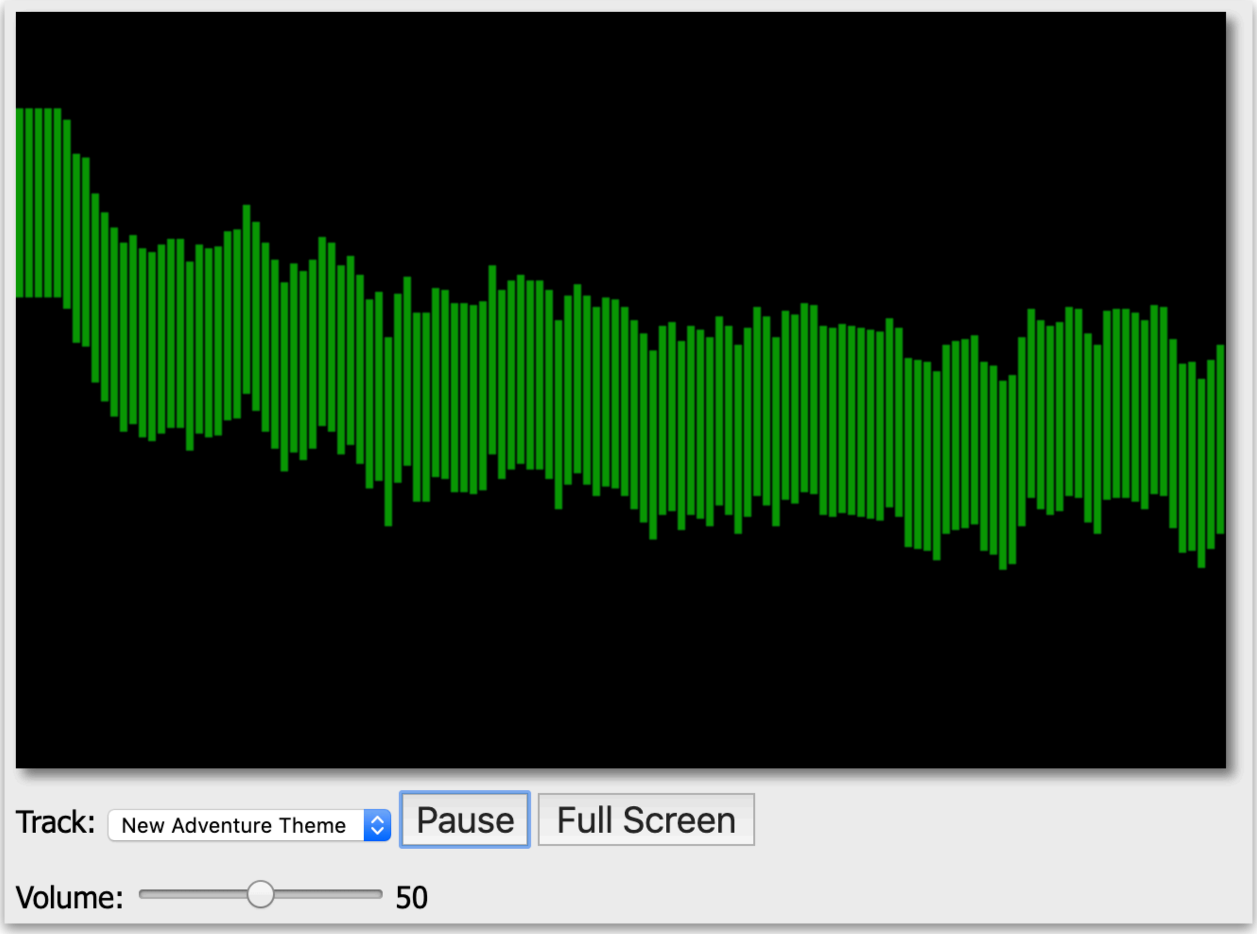
There are **Controls:**

- Audio controls for Play, Pause, and Volume

- A pull down to change what track is playing

- A button that will enable Full Screen mode.

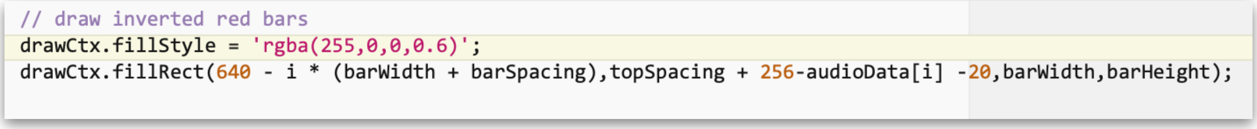
There is a **Visualization:**
 - an array of frequency data (values between 0 - 255) is used to plot and draw bars (rectangles) on a `<canvas>`.



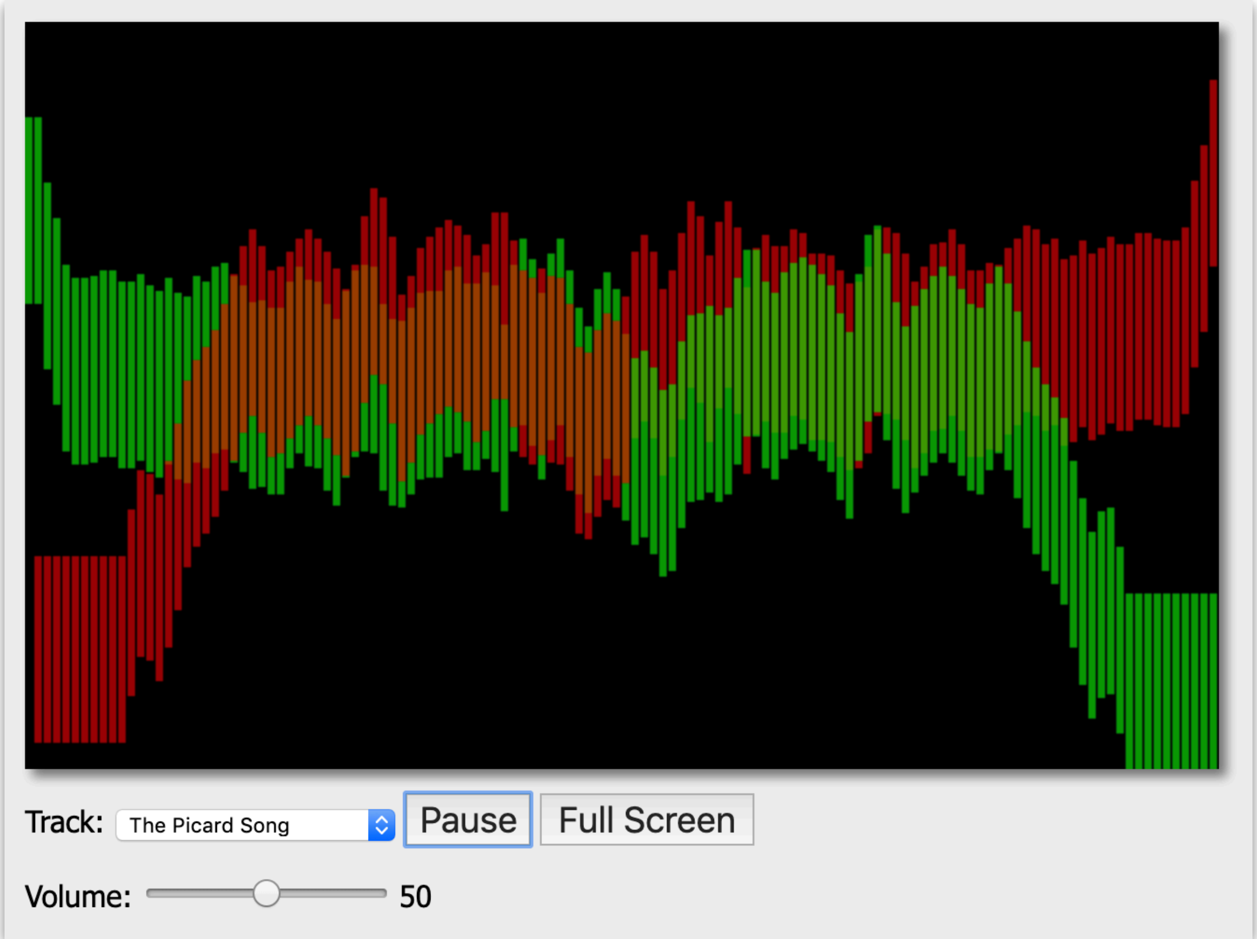
Track: New Adventure Theme

Volume: 50

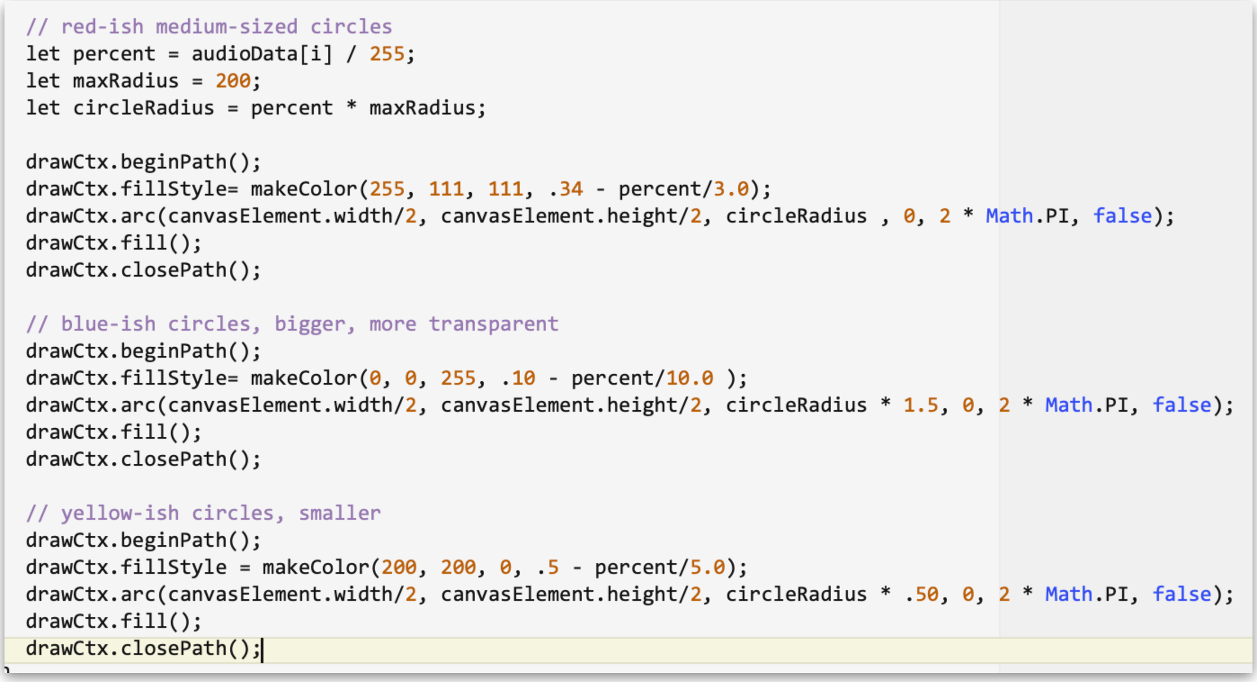
B) To add an inverted bar graph to the canvas, add the following code to the loop:



```
// draw inverted red bars
drawCtx.fillStyle = 'rgba(255,0,0,0.6)';
drawCtx.fillRect(640 - i * (barWidth + barSpacing),topSpacing + 256-audioData[i] -20,barWidth,barHeight);
```



C) To add the circle effects below, add the following code to the loop:

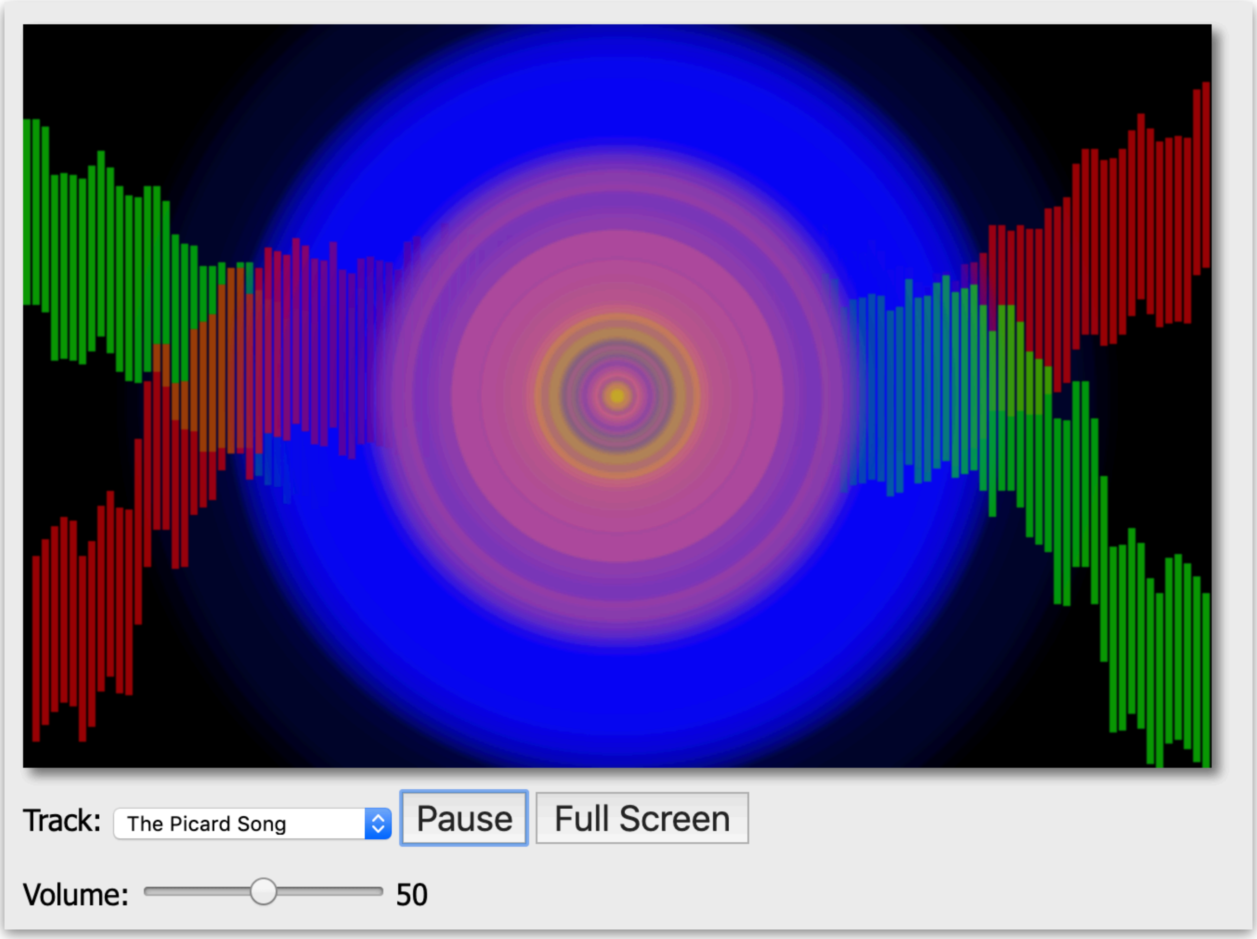


```
// red-ish medium-sized circles
let percent = audioData[i] / 255;
let maxRadius = 200;
let circleRadius = percent * maxRadius;

drawCtx.beginPath();
drawCtx.fillStyle= makeColor(255, 111, 111, .34 - percent/3.0);
drawCtx.arc(canvasElement.width/2, canvasElement.height/2, circleRadius , 0, 2 * Math.PI, false);
drawCtx.fill();
drawCtx.closePath();

// blue-ish circles, bigger, more transparent
drawCtx.beginPath();
drawCtx.fillStyle= makeColor(0, 0, 255, .10 - percent/10.0 );
drawCtx.arc(canvasElement.width/2, canvasElement.height/2, circleRadius * 1.5, 0, 2 * Math.PI, false);
drawCtx.fill();
drawCtx.closePath();

// yellow-ish circles, smaller
drawCtx.beginPath();
drawCtx.fillStyle = makeColor(200, 200, 0, .5 - percent/5.0);
drawCtx.arc(canvasElement.width/2, canvasElement.height/2, circleRadius * .50, 0, 2 * Math.PI, false);
drawCtx.fill();
drawCtx.closePath();
```



Track: The Picard Song

Volume: 50

D) Assignment (out of 10 points)

i) Add a slider and have it change the maximum radius of the circles. (5 points)
Hints:

- there is a local `maxRadius` variable. Declare it as a *script-scoped variable* instead (where `canvasElement` and `drawCtx` are in section I.). Then the slider can change the `maxRadius` value.

ii) Comment out both of the blocks of the rectangle code, and instead draw something else - lines, curves, ovals, circles, …? (5 points)