Master Degree in Computer Science
and Engineering
Musical Acoustics track

Vynthesizer: Sound Analysis Synthesis and Processing course project

Edoardo Daniele Cannas
Registration n. 879063

Academic Year 2017/2018

Introduction

The following report aims at resuming the activity behind the realization of the **Sound Analysis Synthesis** and **Processing** course project, named **Vynthesizer**.
Vynthesizer is a polyphonic web synthesizer, made of three oscillators each of which produces the output of one of the three octaves of a piano keyboard. The keyboard can be either used through the mouse, or by using the actual computer's keyboard, allowing the user to produce custom monophonic sounds or even chords up to three notes (see figure 1).
The user has access to several controls on the sound generated by each oscillator, such as the waveform, the envelope curves, etc..., that will be explained in details in **Chapter 1**.
While producing sounds, the signals generated by the oscillators are analyzed in realtime to extract meaningful information that are in turn used to drive the animations shown in the display (see figure 2).
In **Chapter 2** we will rundown the implementation details and the methods used for the feature extraction and visualization.
This project is based mainly on Tone.js for the sound analysis and synthesis part, and on p5.js for the visualization part. A complete list of the libraries used, along with the source code, is available at the following GitHub repository, while the Vynthesizer can be used and played at the following link https://vynthesizer.herokuapp.com/.

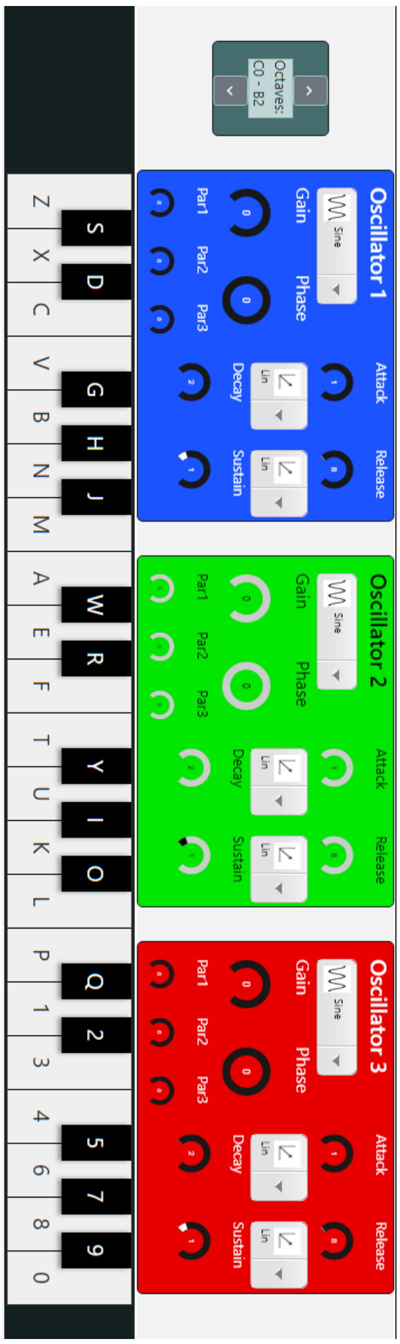

Figure 1: Input keyboard and oscillators' controls

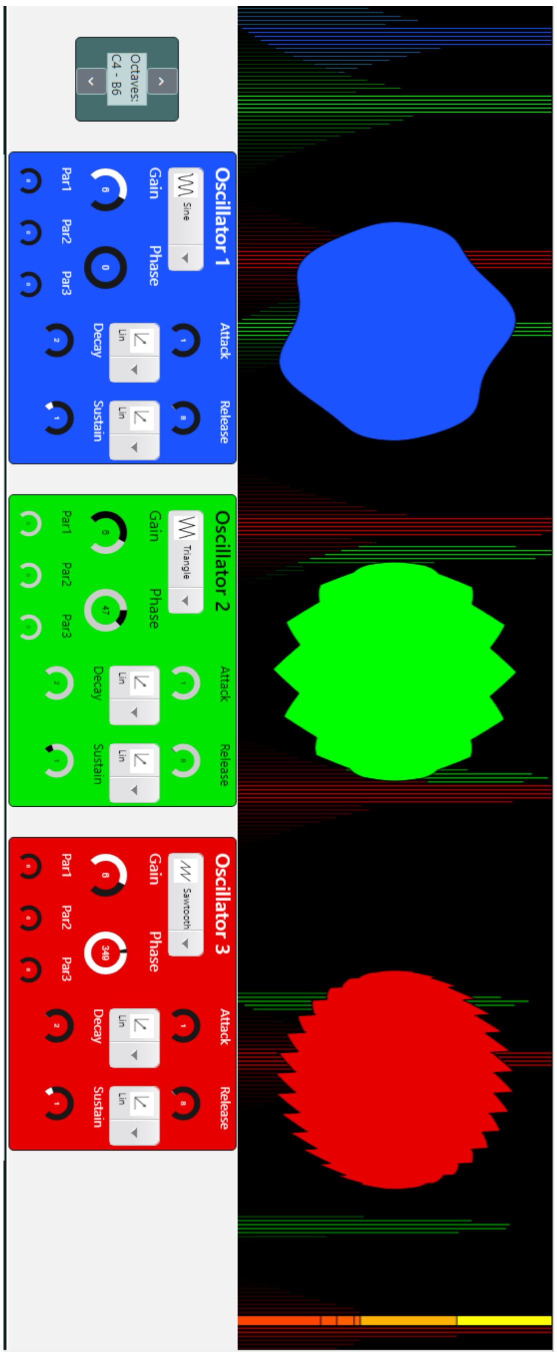

Figure 2: Real-time feature visualization

Chapter 1

Sound synthesis - Web Audio API and sound generation in web applications

1.1 Audio in the Web and the Web Audio API

With the evolution of devices and the spread of the Internet as now maybe the most used mean for communication and service providing, the tools made available to developers for creating web applications became more and more powerful too. In particular, **Javascript**, one of the most used scripting language for web development, has seen a steep increase in its programming capabilities due to the birth of several APIs whose domain of interest vary from Virtual Reality, to gaming, and even **audio applications**.
The **Web Audio API** has been developed for the specific purpouse of handling audio operations on the Web, inside an **audio context** and specifically designed to allow **modular routing**. The basic element in the API is the **AudioNode**, which performs basic audio operations and linked together with other nodes forms an **audio routing graph**. Several sources can be supported within a single context, and the modular design provides enough flexibility to create rather complex audio functions with dynamic effects. Among audio nodes we can list:

- **Source nodes**, which provide arrays of sound intensities (samples) either computed mathematically (in this case we call them **OscillatorNodes**) or taken from sound/video files (in this case we call them **AudioBufferSourceNode** or **MediaElementAudioSourceNode**);

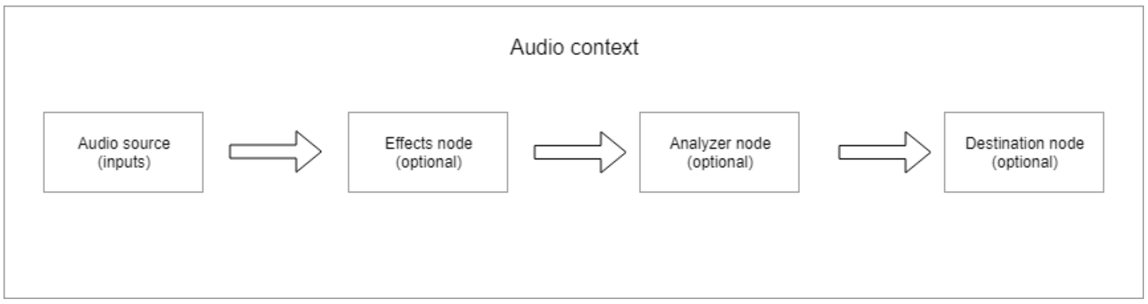

Figure 1.1: Modular routing example in the Web Audio API

- **Effects node**, which elaborate the signal in different ways (for example multiplying each sample by a value to make it louder or quieter, as the **GainNode** does);
- **Analyzer nodes**, which perform basic signal processing operations such as the **Fast Fourier Transform** allowing the user to have a basis for further signal processing analysis;
- **Destination nodes**, which sends the processed signal to the speakers or headphones.

The programmer has complete control even on the **timing** of the signal, allowing (s)he to write code that responds accurately to events and to target specific samples, and the API also allows to control how audio is spazialized[1].

Actually, several other APIs exist as wrappers of the Web Audio API, allowing for the consumer to develop even more elaborated audio applications. For this project, the choice has fallen upon the **Tone.js** API.

1.2 Vynthesizer: the (S)ynthesis part

As said previously, the sound synthesis module of the Vynthesizer relies on the Tone.js API, which is a wrapper around the native Web Audio API. Tone.js offers a good compromise between the wrapping (hiding to the consumer some tedious operation like the audio context instantiation, for example) and the possibility to the user of accessing and using the primitives of the Web Audio API.

[1]For a introduction and a guide on the Web Audio API, check the Mozilla developer guide on the API

The essence behind the use of Tone.js remains the same: the programmer instantiates audio nodes and connect them designing an audio routing graph. For the Vynthesizer, the synthesis part consists of three **OscillatorNodes**, whose output is connected to three **AmplitudeEnvelopeNodes**, and finally to three **GainNodes**, as depicted in the figure 1.2.
The Oscillators produce 128 samples (32-bit linear floating-point PCM values) per frame, at a sampling rate of 48kHz, emitting a note whose frequency is guided by the key pressed by the user. Each one of the three Oscillators in fact is assigned to one of the three octaves of the keyboard, allowing the user to create even basic chords. The actual range of the keyboard can be chosen starting from C0 (16.35 Hz) - B2 (123.47 Hz) in steps of three octaves up to C6 (1046.50 Hz) - B8 (7902 Hz).
The user has control on the **waveform** of each Oscillator: in fact, by using the **PeriodicWave** interface, programmers can create their own custom waveforms by setting appropriate coefficients on the Fourier series expansion of the desired signal. Letting a and b represent the real and imaginary arrays of coefficients of length L of the Fourier series, the basic time-domain waveform $x(t)$ can be computed using the following equation:

$$x(t) = \sum_{k=1}^{L-1} (a[k] \cos 2\pi kt + b[k] \sin 2\pi kt). \tag{1.1}$$

In the case of Vynthesizer, the user can choose among 4 basic types of waveforms:

- a **sine function**, for which the coefficients of 1.1 are $a[k] = 0$ and $b[k] = \begin{cases} 1 & for \quad k = 1 \\ 0 & otherwise \end{cases}$
- a **square** function, for which the coefficients of 1.1 are $a[k] = 0$ and $b[k] = \frac{2}{k\pi}[1 - (-1)^k]$
- a **triangle** function, for which the coefficients of 1.1 are $a[k] = 0$ and $b[k] = \frac{8 \sin \frac{k\pi}{2}}{(\pi k)^2}$
- a **sawtooth** function, for which the coefficients of 1.1 are $a[k] = 0$ and $b[k] = (-1)^{k+1} \frac{2}{k\pi}$.

For each Oscillator, the user can set also the phase of the principal component, or set manually the values of the first three armonics of the signal through the little knobs at the bottom of the Oscillators' panel: in this case, the waveforms becomes automatically a sinusoid, where the knobs controls

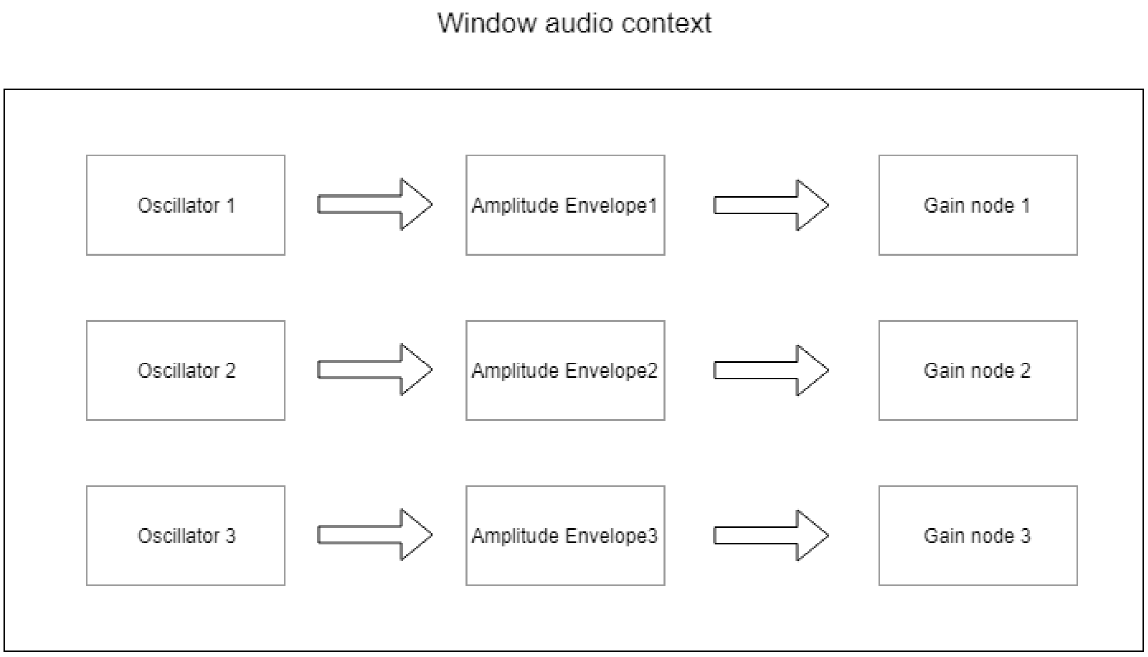

Figure 1.2: Audio routing graph for the synthesis module of the Vynthesizer

the amplitude of the partials up to a value equal to the fundamental component's amplitude.
For what concerns the AmplitudeEnvelope, it is a simple envelope filter which follows the **Attack Decay Sustain Release** model. The user can set the values of the Attack, Decay, and Release time in ms, up to $1000ms$, along with the Sustain value, which is a percentage value of the maximum signal value at which the signal rests before the Release is triggered. Moreover, the user can select the waveforms of the Attack and Release part of the envelope among different options:

- the **sine** and **square** functions explained before, together with a **cosine**;
- **linear** and **exponential** functions;
- a **ripple** and a **bounce** functions.

Finally, the user can set the final gain value for the **GainNodes** of each Oscillator, with the gain defined as the ratio between the incoming and outgoing signal.

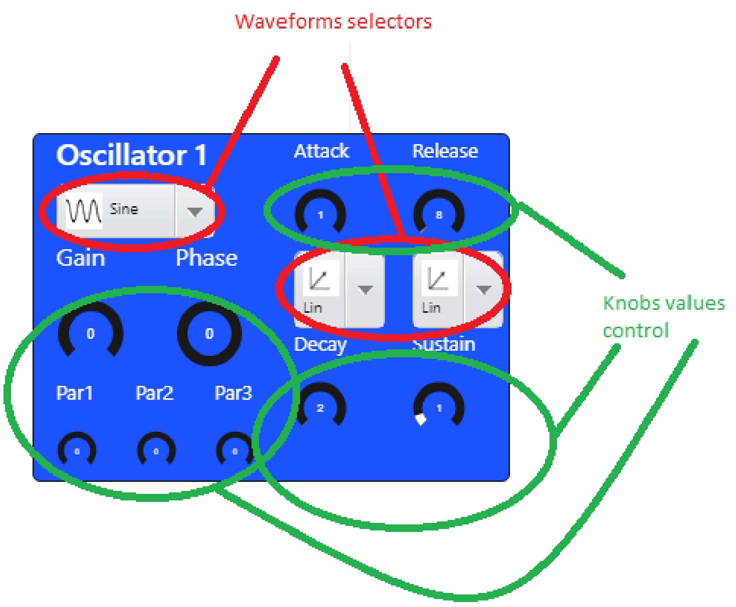

Figure 1.3: Oscillators controls

Chapter 2

Sound analysis and visualization: AnalyzerNodes and p5.js

2.1 "Processing" for the Web

Along with APIs for the Digital Signal Processing, the web development has seen the rise of several libraries for the creation of animations, drawings, etc...
p5.js is a Javascript library started as a parallel project to **Processing**, with the original goal of making coding accessible for artists, designers, educators, etc... reinterpreting for today's web. p5.js provides a full set of drawing functionality, conceiving the web page as a **sketchbook** where the programmer can freely write code to create the visuals (s)he has in mind; recently, p5.js has started offering a wide set of libraries that make it easy to interact with other HTML5 object, as well as text, inputs, videos, webcams and sounds.
The main core of each p5.js scripts consists in two functions:

- a **setup** function, where the programmer creates all the variables and functions needed for the execution of its drawings;
- a **draw** function, which creates the visuals and runs continuosly unless the program is stopped.

A little example is shown below. The output is depicted in figure 2.1, and is taken from here.

Listing 2.1: A simple p5.js example

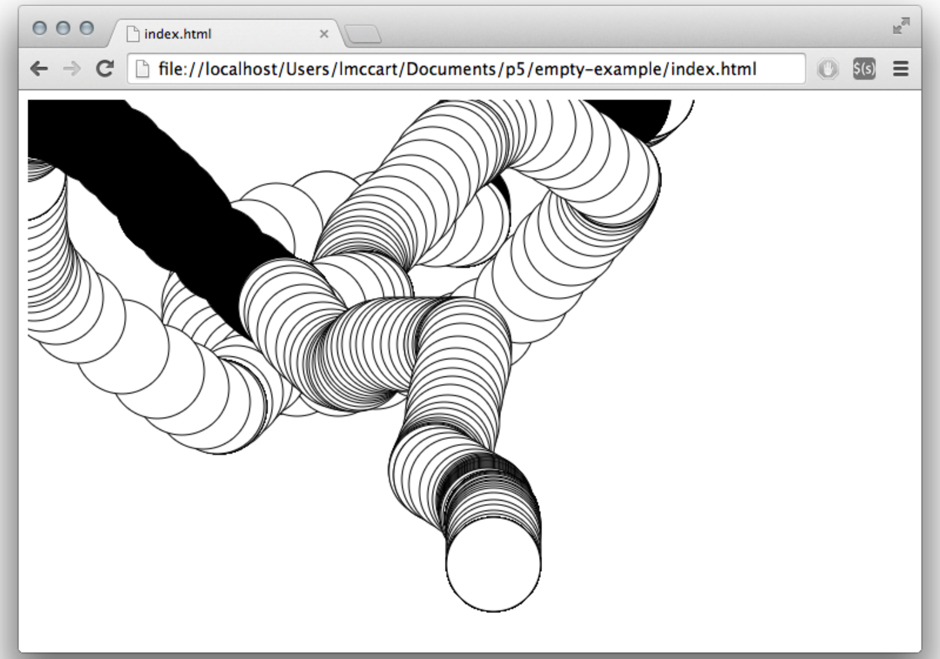
index.html
file://localhost/Users/lmccart/Documents/p5/empty-example/index.html

Figure 2.1: Code output

```
function setup() {
  createCanvas(640, 480);
}

function draw() {
  if (mouseIsPressed) {
    fill(0);
  } else {
    fill(255);
  }
  ellipse(mouseX, mouseY, 80, 80);
}
```

2.2 Vynthesizer: the V(isualization) part

The continous run of the **draw** function in the p5.js scripts, allows for the synchronization and triggering of drawings and animations steer in real time by some control signals. In the case of Vynthesizer, the animations are driven by the sounds generated by the user, and by the spectral information extracted from them. All the information have been extrapolated using the **AnalyzerNodes** of the Web Audio API, creating the audio routing graph depicted in the picture 2.2.

For each one of the Oscillators' signals, the following features (according to the MPEG-7 standard) are extracted:

- the audio **waveform**;
- the audio **spectrum**;

while from the master output is computed the **spectral centroid**.
The audio waveform is used together with the envelope's amplitude to drive dynamically the aspect of three circles, one for each oscillator. The radius of the circles in fact is driven by the amplitude envelope's signal, therefore leading to the circles enlarging with the curves of the ADSR imposed by the user for each one of the oscillators. The circumferences of the circles instead, vary taking the form of the oscillator's waveform, as illustrated in figure 2 and figure 2.3.
The audio spectrum is extracted through a **Fast Fourier Transform**, with the operations performed by the AnalyzerNode listed in the following:

1. get the current time domain data;
2. apply a **Blackmann window** $w[n] = a_0 - a_1 \cos\frac{2\pi n}{N} + a_2 \cos\frac{4\pi n}{N}, \quad n = 0, \cdots, N-1$, with $a_0 = \frac{1-\alpha}{2}$, $a_1 = \frac{1}{2}$ and $a_2 = \frac{\alpha}{2}$, with $\alpha = 0.16$;
3. apply the Fast Fourier Transform $X[k] = \frac{1}{N}\sum_{n=0}^{N-1}\hat{x}[n]e^{\frac{-2\pi jkn}{N}}$ to get both the imaginary and real frequency samples, with $\hat{x}[n]$ being the windowed signal and with $k = 0, \cdots, \frac{N}{2} - 1$;
4. smooth over time the module of the FFT according to the following formula $\hat{X}[k] = \tau\hat{X}_{-1}[k] + (1-\tau)|X[k]|$, where $\hat{X}_{-1}$ is the smoothed version of the FFT of the previous windowed signal, and τ is the **smoothing time constant** used to average the FFT across different windows ($0 < \tau < 1$, in this case $\tau = 0.8$);
5. convert the FFT samples to dB $Y[k] = 20\log_{10}\hat{X}[k]$.

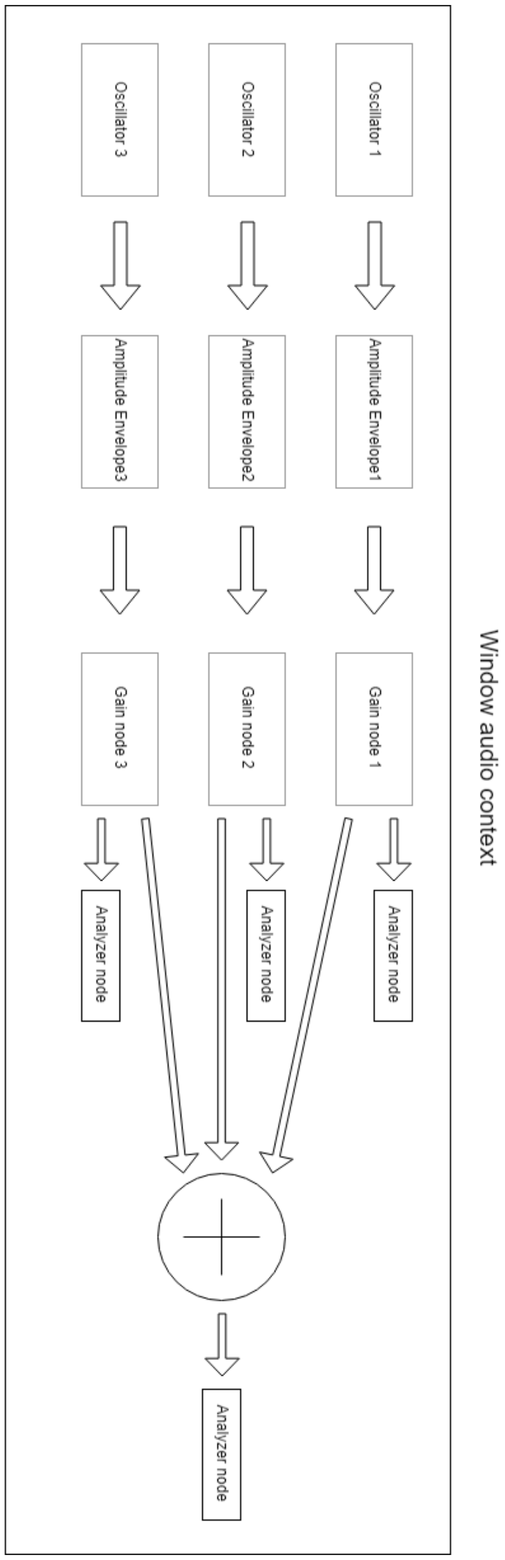

Figure 2.2: Analysis routing graph for the Vynthesizer

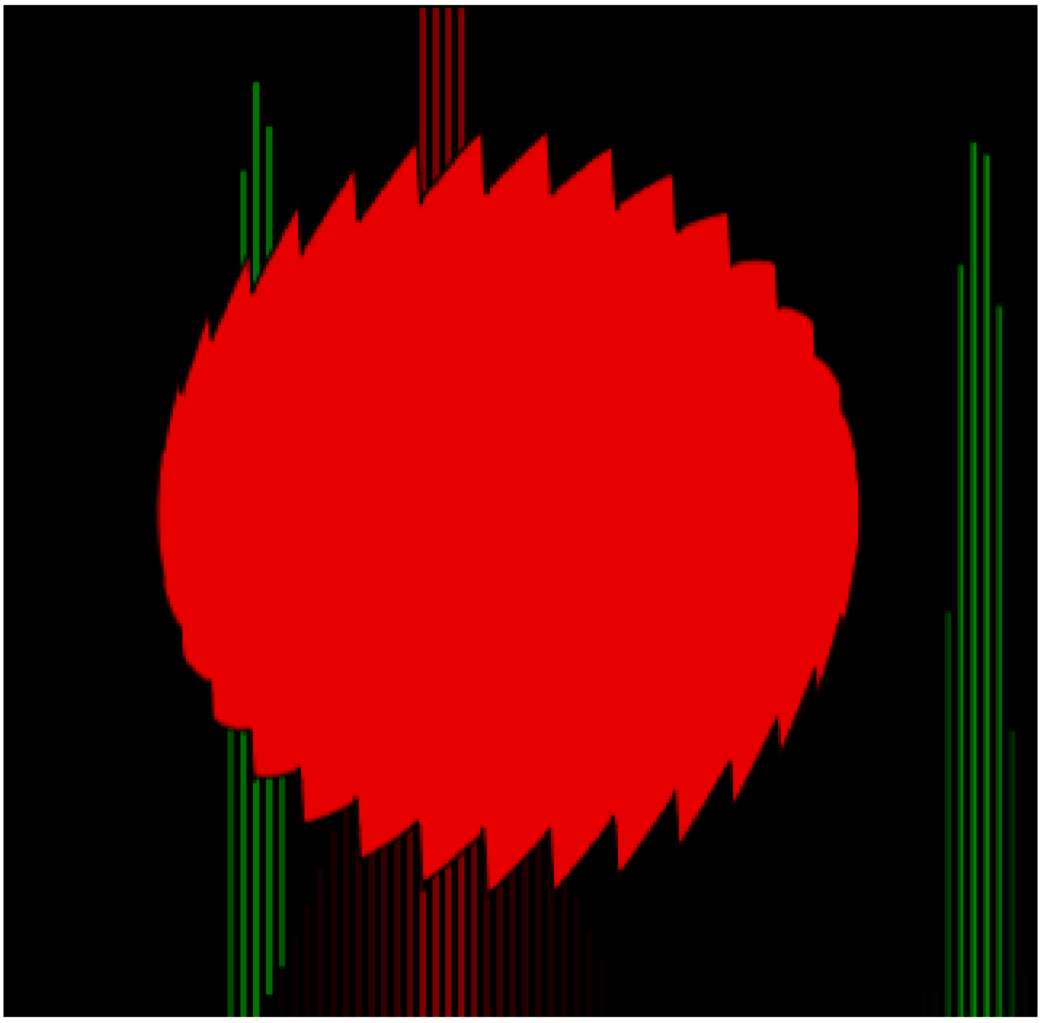

Figure 2.3: Circle corresponding to Oscillator 3: as we can see, the circumference takes the shape of the Oscillators' waveform (a sawtooth in this case)

Since the lowest note corresponds to a C0 (16.35 Hz), and the sampling frequency is set to 48kHz, using the **peak localization error theorem** a FFT of 2048 samples has been used.
The obtained frequency samples have then been folded and doubled in amplitude, and finally normalized according to the following formula:

$$b[k] = \left\lfloor \frac{255}{dB_{max} - dB_{min}} (Y[k] - dB_{min}) \right\rfloor$$

where dB_{min} and dB_{max} are the minimum and maximum dB value of the FFT. This operation has been done in order to have the values lying in a range between 0 and 255, in order to have the frequency bin amplitude corresponding to an RGB value for the realization of a "pseudo" spectrogram visualization. A picture of the "pseudo" spectrogram of a Cmaj is reported below in figure 2.4, where the spectrum of each oscillator is rendered with a different colour. Please note that since for a 2048 FFT the bin corresponding to the highest note available for the Vynthesizer (B8, 7902 Hz) is the 340th, for the sake of a more clear drawing and representation the spectrogram reports only the first 341 elements of the FFT.
Finally, as we can see from figure 2.4, a yellow bar indicates the position of the **spectral centroid** of the generated sound signal, which is computed simply as

$$ASC = \frac{\sum_{k=0}^{N/2} f[k]Y[k]}{\sum_{k=0}^{N/2} Y[k]},$$

where $f[k]$ is a vector containing the frequencies corresponding to each FFT bin, and $Y[k]$ the normalized FFT module.

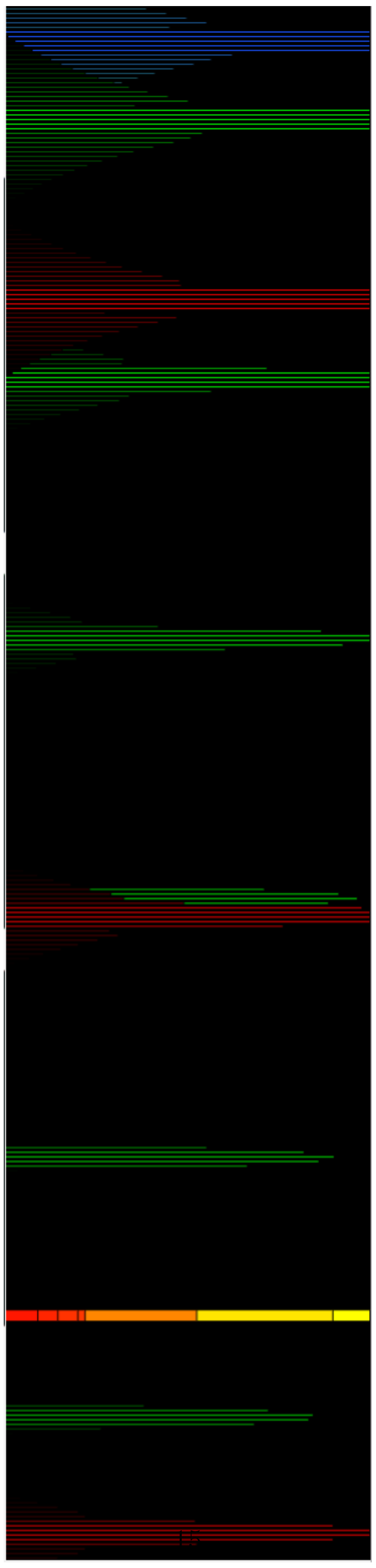

Figure 2.4: Vynthesizer spectrogram: the yellow bar corresponds to the location of the spectral centroid

Chapter 3

Conclusions

The present work aimed at exploring the possibilities open by Web technologies for the use of DSP tools and techniques in creating new content for web applications. Besides the didactic nature of this activity, which has only scratched the surface on the real capabilities of the technologies used, the work done so far has shed some light on the role that web applications are assuming in the landscape of today's informatics. In a continous shift of applications towards distribution, using platforms as cloud services, even the audio engineering is following this route, with some really remarkable results (just to give an example, the new learning site realized by Ableton for beginners in music production, is realized with Tone.js). The drawbacks however are the typical ones related to web technologies, which in turn are the typical ones related with the implementations of standars: for example, Tone.js has some open issues (which can be viewed in Vynthesizer too) when the application is accessed by some Internet browsers (like Mozilla Firefox, for instance). Neverthless, the premises are really encouraging, and without any doubt we will see DSP techniques more and more integrated in web applications in the near future.