Cristóvão Diniz Trevisan, Viktor Volochtchuk

Guitar Digitizer

Brazil

2017, v-0.0.1

Cristóvão Diniz Trevisan, Viktor Volochtchuk

Guitar Digitizer

Project presented as graduation material for the course of Electronic Engineering at UTFPR

Federal University of Technology - Paraná – UTFPR

Electronic Engineering

Graduation Program

Supervisor: Gustavo Benvenutti Borba

Brazil

2017, v-0.0.1

Cristóvão Diniz Trevisan, Viktor Volochtchuk
Guitar Digitizer/ Cristóvão Diniz Trevisan, Viktor Volochtchuk. – Brazil, 2017, v-0.0.1-
44 p. : il.

Supervisor: Gustavo Benvenutti Borba

Graduation Final Project – Federal University of Technology - Paraná – UTFPR
Electronic Engineering
Graduation Program, 2017, v-0.0.1.

1. Hexaphonic Guitar 2. Digitizer 3. MIDI 4. Pitch Detection I. Guitar Digitizer II. Gustavo Benvenutti BorbaIII. Federal University of Technology - Paraná IV. Electronic Engineering

CDU 02:141:005.7

Cristóvão Diniz Trevisan, Viktor Volochtchuk

Guitar Digitizer

Project presented as graduation material for the course of Electronic Engineering at UTFPR

Project Approved. Brazil, October 19, 2017:

Gustavo Benvenutti Borba
Supervisor

Professor
Invited 1

Professor
Invited 2

Professor
Invited 3

Professor
Invited 4

Brazil
2017, v-0.0.1

This work is dedicated to those who supported our way into through enginnering course,
even more when ourselfs tried to give up.

Acknowledgements

We give our special thanks to Eng. Mikhail Anatholy Koslowski, who gave us both intellectual (since he once used an equipament with the same purposed) and physical support (with compenents importation and equipaments). Also to our advisor who was present even in the moment that the initial idea came to life, in a situation nobody else would have stayed.

Abstract

Guitar are one of most popular instruments today, but there is one big disadvantage to use it: there is no good and affordable way to digitalize it's music. The biggest problem with this is the cost to annotate music, as it needs to be done by manually. This project tries to build one such system, building from passive hardware (hexaphonic pickup) to modern signal processing (pitch detection), attempting to produce a cheap and effective equipment for guitar music annotation by means of generating MIDI format data.

Key-words: guitar. digitizer. MIDI. pitch. detection. hexaphonic.

Resumo

Violões e guitarras estão entre os instrumentos mais populares da atualidade, mas existe uma grande desvantagem em os utilizar: não há um meio barato e eficaz para digitalizar sua música. O grande problema com isso é o alto custo para transcrever partituras, que atualmente é um processo manual. Esse projeto tenta construir um sistema com esse propósito, criando desde sensores passivos (captador hexafônico) até processamento digital de sinais moderno (detecção de nota), visando um produto barato e eficaz para anotação musical através da geração de dados no format MIDI.

Key-words: guitarra. digitalizador. MIDI. nota. detecção. hexafônico.

List of Figures

List of Tables

List of abbreviations and acronyms

MIDI	Musical Instrument Digital Interface
UI	User Interface
GUI	Graphical User Interface
API	Application Programming Interface
USB	Universal Serial Bus
DOM	Document Object Model
n.d.	No Date
fft	Fast Fourier Transform
ifft	Inverse Fast Fourier Transform
npm	Node Package Manager
DOM	Document Object Model
ADC	Analog to Digital Converter
DMA	Direct Memory Access

List of symbols

Ω	Ohm resistance unit

Contents

Introduction

Part I

Hardware

Part II

Firmware

1 Tools Selection

The firmware is basically an analog sampler, all it has to do is sample six analog channels, add a header (to identify the beginning and check continuity) and send it through USB. Lets start by listing the requirements for the hardware.

1.1 Requirements

a) Super cheap

b) 6 analog channels (more precision is better)

c) High sample rate (at least 10 kHz for each channel, but ideally 44 kHz or more)

d) Fast USB support, to send the data with headers

Based on this requirements the minimum transfer speed can be calculated, let's consider that a header will be set for every 252 samples (42 for each channel) and it has 4 bytes (3 of identification - to assure it is the header and not some data - and a counter). Previewing the worst case, each sample is 2 bytes long. The transfer rate given by Equation 1.1.

$$transfer\ rate = \left(\frac{channels * \frac{bytes}{sample} * \frac{samples}{package} + header\ size}{\frac{samples}{package}} \right) * f_s[B/s] \tag{1.1}$$

Considering that f_s has to be somewhere between 10 kHz and 50 kHz the transfer rate the numerical result is given by Equation 1.2.

$$transfer\ rate = \left(\frac{6 * 2 * 252 + 4}{252} \right) * f_s = 12.0159 * f_s = 120.159 - 600.794[kB/s] \tag{1.2}$$

USB transfer speed is usually refered in Mbps, which gives us a range between 961.27 and 4806.34 Mbps. This is too high for serial communication (typical max of 1 Mbps) so we need to add a requirement for raw USB support, which allows bulk transfer that can a have transfer rate up to 12Mbps (USB full-speed standard).
We still need to choose or exact sample rate (f_s), but first let's select which hardware will be used.

1.2 Microcontroller Selection

There are too many microcontrollers that fit our requirements, but the most popular and cheap one is clearly the ARM from ST called STM32F103C8T6, and that is why it

was selected.
It has eight 12 bits ADC inputs (with 2 parallel channels), DMA for the ADCs and USB full-speed support. It's also relatively fast (72 MHz clock, 32 bits architecture). All this for under 2 U$D in a developing board from China (the actual μC is under 0.2 U$).

1.3 Program

Part III

Software

2 Tools Selection

2.1 Top Level Requirements

The exact implementation of each of the items will be discussed later, but to simply set our requirements a general list of them is:

a) Desktop GUI

b) Efficient signal processing (for pitch detection)

c) Real time graph visualization of the signals (oscilloscope like)

d) Access to *libusb* (LIBUSB, n.d.) API

e) Access to MIDI API

2.2 Language Choice

2.2.1 Java

The first choice was Java, as it meets all requirements. Desktop GUI can be done using Swing, a good library for pitch detection is also available (called TasosDSP (SIX; CORNELIS; LEMAN, 2014)). There is also a binding to libusb called usb4java and native MIDI support.
Following that idea a functional prototype was built, but a few problems came to rise. The first is that usb4java high-level API had bugs and was not working correctly. The solution was to fall back to the lower level API, but that made things much more complicated as threading and synchronization problems had to be dealt with. There was no good library for real time visualization either, which made really hard to both debug and tune the frequency detection algorithm. On top of that Swing is at least non-pleasant compared to more modern UI programming, so a second approach came to be.

2.2.2 JavaScript

In alignment with both current work experience and world programming tendencies JavaScript was taken as a choice. We will see that requirements fit much better now, for the following reasons.
For the desktop GUI, JavaScript has a few nice and mature Desktop GUI frameworks, like Electron and NW.js.
JavaScript is an interpreted language, and for that has a low efficiency when compared

to C++ or Java. That is huge problem, but there is an easy overcome. As this project tries to build a desktop application, Node.js will be used ultimately, and it has support for C++ bindings. That means the JavaScript code can call a compiled C++ library to calculate the pitch, thus solving the problem.
Graph visualization should not be a problem either as there are a lot of libraries for that. The most problematic requirement in Java was *libusb* support. It is available in JavaScript using *node-usb* (NODE-USB, n.d.), and a few simple tests returned good results with a much simpler API. MIDI was also tested and worked just fine.

2.3 Desktop Framework

Now that JavaScript is set as our final selection we need an environment to run it. There are two already listed really mature and popular choices: Electron and NW.js.
At first Electron was used to build a test application, because it is the most popular of the two (in fact even the editor used to write this words is built with it), but the pitch detection call was running slowly. As a mater of fact it was running much faster using pure JS code rather than the C++ library. A deeper research was needed, and the way Electron worked was getting in our way, but first it's necessary to know what Node.js is.

2.3.1 Interpreter

JavaScript is a interpreted language and thus needs an interpreter. The most common one is Google V8, which happens to be the same one used in most web browsers as well as in Node. The difference between browsers and Node is simply the API that comes with them. Web needs firstly to access the UI (html) and ways to modify it, it also needs secure and limited access to hardware and internet calls. Of course that means web JavaScript code cannot use C++ libraries directly.
On the other hand Node is a more pure version of V8, it also gives the possibility to write and call C++ code (feature needed for this project), which ultimately makes it as capable as any desktop program can be. Node also comes with a hand-full set of native resources (like file system and full communication access), but it does **not** provide any kind of GUI. Knowing that it is possible to have a better understanding on how the two desktop environments work.

2.3.2 Electron

Electron by design has at least two processes running (HOW..., 2017), one for the "web" and other for Node access. The answer for how the web process access native resources is also to why the execution of the C++ processing library was slow: it uses inter-process communication (IPC). IPC makes things a lot slower, which ultimately makes impossible to use Electron in this project.

2.3.3 Node Webkit

Differently from Electron, NW.js (NODE..., n.d.; HOW..., 2017) takes the Node environment and combines it with Chromium into a single process, removing the use of IPC. Initial tests reported that the pitch detection library has fast execution as expected.

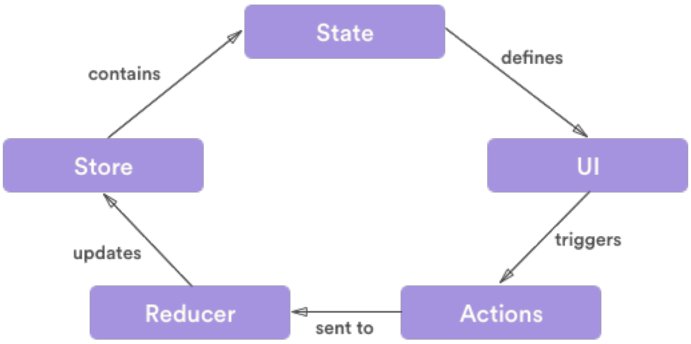
2.4 Architectural Tools

NW.js will go only as far as to give access to both Node and DOM API's. But that is too crude, and not what we wanted by given up on Java Swing. Again based on current work experience and world tendencies the setup chosen is React + Redux.
React (REACT, n.d.) is a library created by Facebook and world wide used for UI applications. It uses a declarative component-based system that makes it easy to build scalable and reusable code.
React only go as far to help building the UI, but we also need to pass the state of the application to the UI components, and that is where Redux (REDUX, n.d.) comes in. It keeps all the application state stored in a single place, described by transition functions. That makes the storage system easy to be tested and used, because all actions (that modify the state) must be well defined and it doesn't rely on the UI (React), making it easy to test.
Figure 1 shows the flow of an application that uses React + Redux. It is obvious to see the simplicity it has, a single path must be followed. This simplicity is what makes it much easier to use against other frameworks like Java Swing.
There is still the choice of the visual library to use, and the chosen one is Semantic UI React (SEMANTIC..., n.d.). It has some nice and robust React components to build a well designed application.

2.5 Fast Signal Processing

Pitch detection is a heavy problem to solve, and good implementations are time consuming, so we need efficiency to run it real-time. The library already said to be used didn't actually existed, the only one available was a pure JavaScript library (PITCHFINDER, n.d.) which is not suitable for this project. The solution was to build our own library based on both the pure JavaScript one and TarsosDSP (SIX; CORNELIS; LEMAN, 2014). Implementation details discussed further on chapter 3.

Figure 1 – React Redux Flow Diagram



Source: Getting... (2016)

2.6 Real Time Visualization

There are lots of charting libraries available for use with web interfaces (and by extension NW.js), unfortunately none was good fit for real time high density signals such as audio. The solution was again to build one, since all other things are looking to run smoothly in JavaScript, implementation details on chapter 4.

3 Pitch Detection

Pitch detection is simply frequency detection with the restriction of note quantization, Table 1 shows the base frequency for each of the 12 existent notes. Multiples of the same frequency are seen as the same note on a different range, known as octave. Even though the quantization makes things simpler it's still a hard task, even more for instruments where there is the presence of harmonic series. Harmonic series notes are multiples of the fundamental frequency (most important note) produced by integer sections of the instrument vibration. Figure 2 shows an visual representation of why there exist. The existence of them as well as the presence of both inter-signal and white noise makes necessary the use of non-trivial algorithms for pitch detection, and two of them will be discussed next.

Table 1 – Notes Frequencies

Note Name	Frequency
A	440.00
A#	466.16
B	493.88
C	523.25
C#	554.37
D	587.33
D#	622.25
E	659.25
F	698.46
F#	739.99
G	783.99
G#	830.61

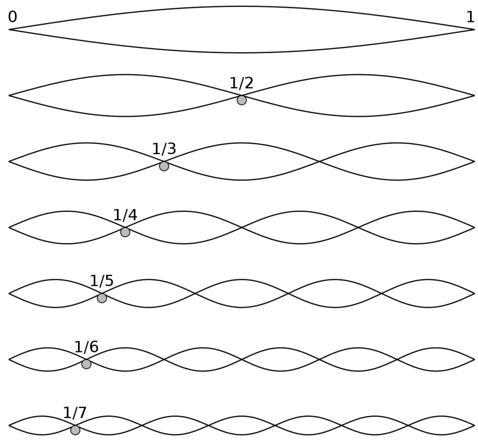
Source: made by authors

3.1 YIN algorithm

Autocorrelation is a well know function to calculate a signal's fundamental frequency, but it gives too much error for this project use case. YIN (CHEVEIGNÉ; KAWAHARA, 2002) is a method that uses a few improvements to the autocorrelation method, achieving a much higher precision. It also can be implemented with logarithmic growth as the autocorrelation can be calculated using the fft and ifft algorithms. The algorithm can be divided in 6 steps, as follows:

1. Autocorrelation
2. Difference

Figure 2 – Harmonic Series



Source: Wikipedia (2017)

3. Cumulative mean normalized difference

4. Absolute threshold

5. Parabolic interpolation

6. Best local estimate

It's important to notice that the absolute threshold is a controlled attempt to regulate the error introduced by the harmonic series (as in Figure 2), it thus gives preference to lower frequencies (below the threshold).

3.2 MacLeod algorithm

MacLeod (MCLEOD; WYVILL, 2005) goes for another approach, using the square difference function. More precisely it uses a special normalized version of it. The best result is then calculated by means of using a parabolic interpolation of the highest peak and its two neighbors, this process also gives a threshold constant that limits the detection of the neighbors, thus the possibility of some tuning. As we will see this gave the best results for our project after some tuning.

3.3 Implementation

Implementation for both algorithms follow the same pattern, taking the pretty Java code of TarsosDSP (SIX; CORNELIS; LEMAN, 2014) and replacing the syntax and data structures with C++ ones (using standard library for containers). There is also a

JavaScript bridge for data type conversion, so we can use the library calls with simple arrays of numbers. All code is available as an open source project at Github and npm (NODE-PITCHFINDER, n.d.).

4 Data Visualization

The objective of data visualization is to live debug and tune both hardware gain and algorithms control constants. The ideal case is to have both real time chart as well as a buffered/triggered one, essentially something like an oscilloscope. That has to be performant for real time audio signals, sampled at more than 47 kHz.
There are a great amount of JavaScript DOM libraries for charting, but most of them are too automatic or have too much details, making them too slow for our need. The solution was to build our own library for that, again available as an open source project at both GitHub and npm (REACT-PLOTTER, n.d.).

4.1 Requirements

a) Be a React Component

b) Automatic calculations for array input

c) Option for triggering

d) Option for buffering

e) Minimum Redraw

f) Fixed Height/Width

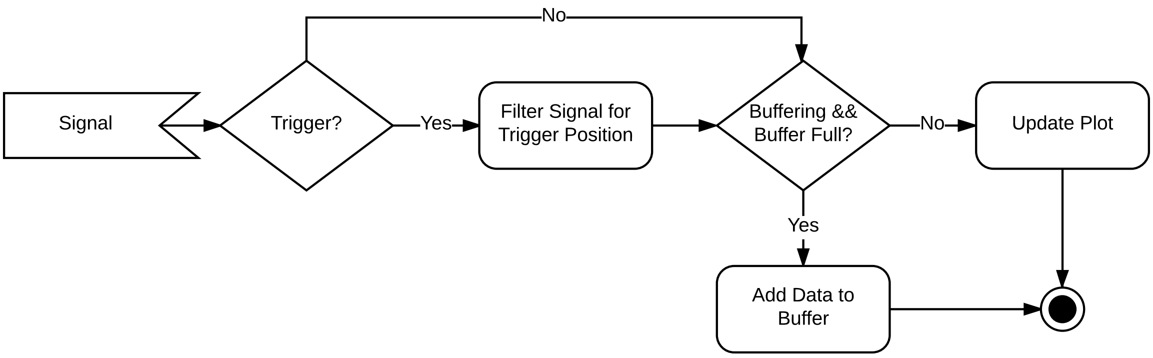
4.2 Algorithm

Triggering and buffering are achived by using a filter that only calls the ploting function when the options are met. This filter is simply the function called to add data, and it is represented by Figure 3.
For the actual drawing a triple buffer technique is used, one for holding the last state (called plotBuffer), on for drawing (called drawingBuffer) and finally the one actually rendered (called canvas). That last one is needed so the arrows don't get saved on the drawing scene.
For linear plot time a translation is established, in a way that only the new points will be drawn, the past ones are only translated to the left. The steps of the algorithm are as follow:

1. Clear drawingBuffer

2. Copy plotBuffer to drawingBuffer translating (removing) extra data

Figure 3 – Add Function Diagram



Source: made by authors

3. Draw new data on drawingBuffer
4. Copy drawingBuffer to plotBuffer
5. Copy plotBuffer to canvas
6. Draw arrows on canvas

4.3 Implementation

Using the listed requirements (section 4.1) a minimum API is built as a React component. Being such all it gives is a set of properties, for which the chart is drawn (when needed), they are listed in Table 2. The style property is a function that is called to

Table 2 – React Plotter Props

Property	**Type**	**Description**
style	Function	Style function (called to print the data)
[trigger]	number	Use trigger
[onlyFull=true]	bool	When using trigger it tells if the view should wait for a complete dataset before updating
[width=300]	number	
[height=150]	number	
[initialData=[ ]]	number[ ]	
[appendData=[ ]]	number[ ]	
[dataSize=100]	number	
[pixelSkip=1]	number	Pixels between points
[max=100]	number	Maximum Y Value
[min=-100]	number	Minimum Y Value
[useMean=true]	bool	Use mean calculation, otherwise median

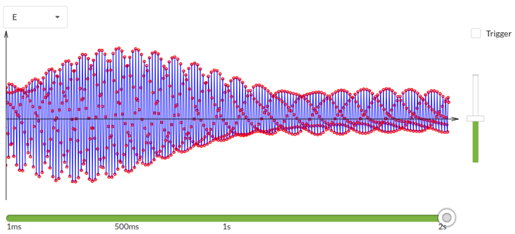
Fonte: react-plotter (n.d.)

render each point. Two styles were built, a line plot (points are connected by a straight line) and a digital plot (digital signal standard chart, not used in the final version of this project).

4.4 Results

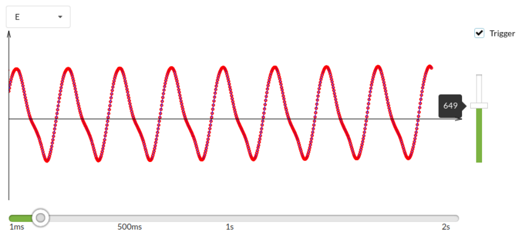
The results are more than satisfactory, tested to be able to run multiple plots of audio speed signals at the same time without much effort. Figure 4 and Figure 5 show how the visualization looks on the project, but full details and working examples are also available at GitHub (REACT-PLOTTER, n.d.).

Figure 4 – Real Time Plot



Source: authors

Figure 5 – Triggered Plot

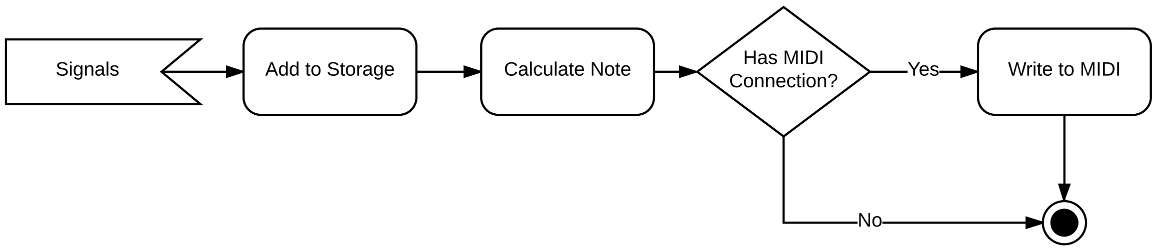


Source: authors

5 Main Program

The general idea is to build an UI with three main divisions: Home (with device selection), midi selection and signal-to-MIDI connections, Plot (with the signal visualization) and Options (with the algorithm tuning options, as well as virtual MIDI creation). When a device is selected it's signals will go through a simple process, as in Figure 6.

Figure 6 – Signals Flow Diagram



Source: made by authors

5.1 GUI

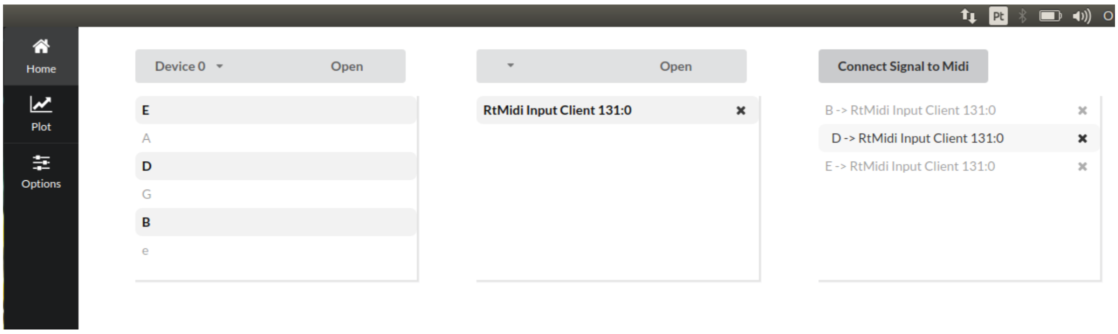
5.1.1 Home

The home page has three horizontally divided sections: Device and signals, MIDI selection and list, and finally connections, as in Figure 7.
The first is used to open the device (there can be only one used at a time). When it is open a list of signals will be displayed (six of them, named as each guitar note). Each signal can be selected so it can be connected to a MIDI device.
The second is to open any given number of existing MIDI devices, which will be listed bellow (can be also closed). The listed devices can also be selected, but only one at a time.
The final section is used to establish the connections, given the selected signals and MIDI, the connections are showed in the box bellow and can be deleted.

5.1.2 Plot

The Plot page is a simply a react-plotter (REACT-PLOTTER, n.d.) component with a few visual controls, being: a dropdown to select which signal is being displayed, a checkbox to enable trigger, a slider to control the time range and another slider to control the trigger value. The full page is as in Figure 8. It was chosen to show only one plot

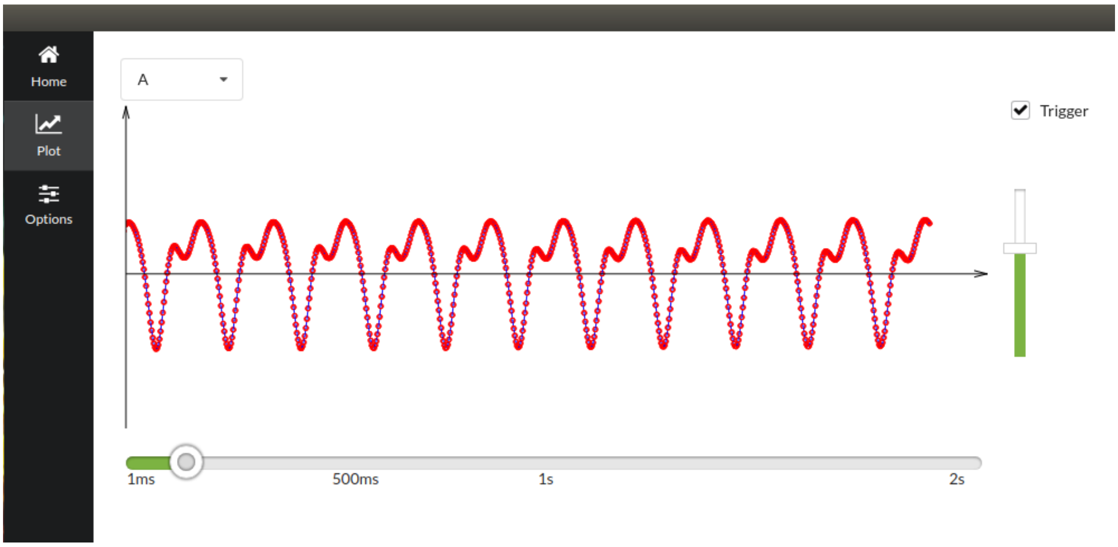
Figure 7 – Home Page



Source: authors

at a time so it's size is bigger, easier to see. This also makes the program a little more performant.

Figure 8 – Plot Page

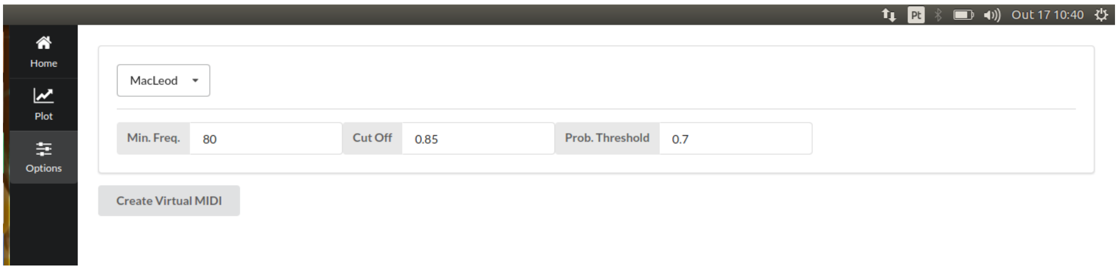


Source: authors

5.1.3 Options Page

The options page is simply an UI to control the used algorithm (from chapter 3) and it's parameters. For both Linux and macOS it's also possible to create virtual MIDI devices, from which our program can write and a synthesizer can read. For Windows this is still possible, but using a hacky solution (since Windows does not give any official API for this) - the easiest option being LoopBe1 (LOOPBE1, n.d.). The page can be seen at Figure 9.

Figure 9 – Options Page



Source: authors

5.2 Implementation

5.2.1 Resources

At first let's take a look at our resorces:

a) USB device: only one can be connected

b) MIDI devices: a list of them can be used at any time

c) Signals: fixed for the only connected device

It's easy to see they are all global, so they can be represented as static classes. But JavaScript has good functional programming capabilities, which are very suitable for global resources. JavaScript imported modules are also scoped, by default, this means it works like a C++ namespace, keeping our static resources separated in a nice way. Taking these mentions in account three modules were built for resource easy access, being MIDI, USB and signal processing.

5.2.2 Entry Point

The entry point for our project is both a declarator and connector. It allocates all needed structures (or calls the module that does it), the most significant one being the Redux store (REDUX, n.d.) - store being a short for storage, which is where all of our application visible state is held.
The entry point also connects all callbacks and logic in a declarative way. In this single file all of the program's internal functionalities are declared, so much that if you read it you should also understand the entire program.

5.2.3 Reducers

Redux stored data is not defined by a set a properties, like typical OOP applications, instead it uses a more functional programming paradigm. The storage is defined by transfer

functions, each of them describing actions that can modify the current state by returning a new one. Each of these functions, called reducer, receive two parameters - the current state and the action to be processed - and should return the new state for the action (or the current one if there are no changes).
Our program has nine reducers, but five of them share the same transfer function, as in Table 3.

Table 3 – Reducers

Name	**Data Type**	**Actions**	**Used for**
device	string	set, remove	current selected device
devices	string[ ]	add, remove	list of devices
signals	string[ ]	set, clear	list of signals
signalsData	object[ ]: name: number[ ]	set, clear	list of signals data (name and values)
object	object: any	set, clear, remove	used for: Plot Page options General options MIDI devices Signal to MIDI connections

Fonte: made by authors

5.2.4 Tests

This is for now a prototype of a real world project, so it has a limited amount of tests. The tests fall into one of two categories: unity or timing. Unity tests are made for the signal processing relating modules and also for every reducer, these are all automated and sum up to a total of 31 tests over 9 modules.
The timing tests were used to check if each separated functionality that may cause processing issues can run in real-time. There are timing tests for: signal average value calculation, signal window buffering, raw data conversion, pitch detection and USB polling.

Part IV

Results and Dicussions

Conclusion

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

Sed eleifend, eros sit amet faucibus elementum, urna sapien consectetuer mauris, quis egestas leo justo non risus. Morbi non felis ac libero vulputate fringilla. Mauris libero eros, lacinia non, sodales quis, dapibus porttitor, pede. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi dapibus mauris condimentum nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam sit amet erat. Nulla varius. Etiam tincidunt dui vitae turpis. Donec leo. Morbi vulputate convallis est. Integer aliquet. Pellentesque aliquet sodales urna.

Bibliography

CHEVEIGNÉ, A. de; KAWAHARA, H. YIN, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, p. 111, jan. 2002. Available from Internet: <http://audition.ens.fr/adc/pdf/2002_JASA_YIN.pdf>. Accessed: 30 sep. 2017. Quoted in page 24.

GETTING Started with React, Redux and Immutable: a Test-Driven Tutorial (Part 2). 2016. Theodo. Available from Internet: <https://www.theodo.fr/blog/2016/03/getting-started-with-react-redux-and-immutable-a-test-driven-tutorial-part-2/>. Accessed: 8 oct. 2017. Quoted in page 23.

HOW does Node.js work with NW.js and Electron? 2017. Medium. Available from Internet: <https://medium.com/@paulbjensen/how-does-node-js-work-with-nw-js-and-electron-da9e50e2c3c1>. Accessed: 8 oct. 2017. Quoted 2 times in pages 21 and 22.

LIBUSB. n.d. Cross-platform API for generic USB access. Available from Internet: <http://libusb.info/>. Accessed: 8 oct. 2017. Quoted in page 20.

LOOPBE1. n.d. A Free Virtual MIDI Driver. Available from Internet: <http://www.nerds.de/en/loopbe1.html>. Accessed: 18 oct. 2017. Quoted in page 31.

MCLEOD, P.; WYVILL, G. A Smarter Way to Find Pitch. *Proc. International Computer Music Conference*, Barcelona, Spain, p. 138–141, sep. 2005. Available from Internet: <http://miracle.otago.ac.nz/tartini/papers/A_Smarter_Way_to_Find_Pitch.pdf>. Accessed: 30 sep. 2017. Quoted in page 25.

NODE-PITCHFINDER. n.d. A compilation of pitch detection algorithms Node. Available from Internet: <https://github.com/cristovao-trevisan/node-pitchfinder>. Accessed: 8 oct. 2017. Quoted in page 26.

NODE-USB. n.d. Node bindings to libusb. Available from Internet: <https://github.com/tessel/node-usb>. Accessed: 8 oct. 2017. Quoted in page 21.

NODE Webkit. n.d. Available from Internet: <https://nwjs.io/>. Accessed: 8 oct. 2017. Quoted in page 22.

PITCHFINDER. n.d. A compilation of pitch detection algorithms for Javascript. Available from Internet: <https://github.com/peterkhayes/pitchfinder>. Accessed: 8 oct. 2017. Quoted in page 22.

REACT. n.d. A JavaScript library for building user interfaces. Available from Internet: <https://reactjs.org/>. Accessed: 8 oct. 2017. Quoted in page 22.

REACT-PLOTTER. n.d. Real Time (high speed) Plotter Component for React. Available from Internet: <https://github.com/cristovao-trevisan/react-plotter>. Accessed: 16 oct. 2017. Quoted 4 times in pages 27, 28, 29, and 30.

REDUX. n.d. Predictable state container for JavaScript apps. Available from Internet: <http://redux.js.org/>. Accessed: 8 oct. 2017. Quoted 2 times in pages 22 and 32.

SEMANTIC UI React. n.d. The official Semantic-UI-React integration. Available from Internet: <https://react.semantic-ui.com>. Accessed: 9 oct. 2017. Quoted in page 22.

SIX, J.; CORNELIS, O.; LEMAN, M. TarsosDSP, a Real-Time Audio Processing Framework in Java. In: *Proceedings of the 53rd AES Conference (AES 53rd)*. [S.l.: s.n.], 2014. Quoted 3 times in pages 20, 22, and 25.

WIKIPEDIA. *Harmonic series (music)*. 2017. Available from Internet: <https://en.wikipedia.org/wiki/Harmonic_series_(music)>. Accessed: 9 oct. 2017. Quoted in page 25.

Appendix

APPENDIX A – Quisque libero justo

Quisque facilisis auctor sapien. Pellentesque gravida hendrerit lectus. Mauris rutrum sodales sapien. Fusce hendrerit sem vel lorem. Integer pellentesque massa vel augue. Integer elit tortor, feugiat quis, sagittis et, ornare non, lacus. Vestibulum posuere pellentesque eros. Quisque venenatis ipsum dictum nulla. Aliquam quis quam non metus eleifend interdum. Nam eget sapien ac mauris malesuada adipiscing. Etiam eleifend neque sed quam. Nulla facilisi. Proin a ligula. Sed id dui eu nibh egestas tincidunt. Suspendisse arcu.

APPENDIX B – Nullam elementum urna vel imperdiet sodales elit ipsum pharetra ligula ac pretium ante justo a nulla curabitur tristique arcu eu metus

Nunc velit. Nullam elit sapien, eleifend eu, commodo nec, semper sit amet, elit. Nulla lectus risus, condimentum ut, laoreet eget, viverra nec, odio. Proin lobortis. Curabitur dictum arcu vel wisi. Cras id nulla venenatis tortor congue ultrices. Pellentesque eget pede. Sed eleifend sagittis elit. Nam sed tellus sit amet lectus ullamcorper tristique. Mauris enim sem, tristique eu, accumsan at, scelerisque vulputate, neque. Quisque lacus. Donec et ipsum sit amet elit nonummy aliquet. Sed viverra nisl at sem. Nam diam. Mauris ut dolor. Curabitur ornare tortor cursus velit.

Morbi tincidunt posuere arcu. Cras venenatis est vitae dolor. Vivamus scelerisque semper mi. Donec ipsum arcu, consequat scelerisque, viverra id, dictum at, metus. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut pede sem, tempus ut, porttitor bibendum, molestie eu, elit. Suspendisse potenti. Sed id lectus sit amet purus faucibus vehicula. Praesent sed sem non dui pharetra interdum. Nam viverra ultrices magna.

Aenean laoreet aliquam orci. Nunc interdum elementum urna. Quisque erat. Nullam tempor neque. Maecenas velit nibh, scelerisque a, consequat ut, viverra in, enim. Duis magna. Donec odio neque, tristique et, tincidunt eu, rhoncus ac, nunc. Mauris malesuada malesuada elit. Etiam lacus mauris, pretium vel, blandit in, ultricies id, libero. Phasellus bibendum erat ut diam. In congue imperdiet lectus.

Annex

ANNEX A – Morbi ultrices rutrum lorem.

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

ANNEX B – Cras non urna sed feugiat cum sociis natoque penatibus et magnis dis parturient montes nascetur ridiculus mus

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

ANNEX C – Fusce facilisis lacinia dui

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.