Linnéuniversitetet
Kalmar Växjö

Bachelor Thesis in Computer Science

Implementing virtual analog synthesizers with the Web Audio API

An evaluation of the Web Audio API

Författare: Oskar Eriksson
Handledare: Jesper Andersson
Termin: VT13
Kurskod: 2DV00E

Abstract
This thesis in computer science aimed to evaluate the suitability of the Web Audio API to implement virtual analog synthesizers. In order to do so a method for producing a reference architecture for analog synthesizers and a categorization and point system for the evaluation were developed. A reference architecture were made and then implemented with the API and the evaluation were then made based on that implementation. The API were found to cover a lot of the necessary areas, but lacking in a few key components; a noise generator, a native way to automate custom numeric properties and the implementation of the oscillator were deemed too closed to support all use cases.

Table of Contents

1. Introduction

This bachelor thesis in computer science were written at the Linnaeus University. It was written as part of the course 2DV00E, Degree Project at Bachelor level, 15 credits.

1.1 The Web Audio API and Audio Synthesis

In January 2011 the Web Audio API (here on abbreviated as WAAPI) was released in Google Chrome version 10 [1]. The WAAPI has been a W3C working draft since December 15, 2011 [2], and allows sample perfect scheduling of audio and includes a wide range of facilities for generating and processing audio in real time, which is exactly what is needed for interactive musical applications. However, up until Chrome version 20 the API were designed to use pre recorded audio files as the source of audio, and there wasn't much choice but to programmatically synthesize audio waves by hand using JavaScript to achieve greater control over the waveforms. This, obviously, demands much of the developer to create effective implementations and synthesis algorithms is not entirely trivial either.

Chrome version 20 included an implementation of the oscillator node [3] in the WAAPI which opened up the possibilities for creating synthesizer instruments on the web. Instead of writing the waveforms using JavaScript the API now provides an implementation in the underlying C/C++ layer of the browser.

Digital models of analog synthesizers has been around quite a while, but not much has been done in terms of synthesizers in browsers implemented with JavaScript. There is a Google Doodle [4] that model the Mini Moog, built with the WAAPI, however it was made before the native oscillator was introduced, as can be seen in the source code[5]. There are some other attempts [6, 7] though nothing more robust has yet been released since the oscillator node came out. This poses an interesting question - isn't the WAAPI ready yet for more ambitious synthesizer implementations?

1.2 Purpose, goal and target reader

The purpose of this thesis is to evaluate the WAAPI, based on a reference architecture, on its suitability to implement virtual analog synthesizers.

The goal is a well defined reference architecture for VA synthesizers and a playable virtual analog synthesizer built in the browser based on that reference architecture.

The target reader is a developer interested in implementing a VA synthesizer. The reference architecture is language agnostic, but the API evaluation is specifically targeted for JavaScript developers. Some familiarity with musical concepts and a certain technical proficiency is expected of the reader, though mathematics will be kept to a bare minimum.

1.3 Problem formulation

The main problem formulation is *an evaluation of the Web Audio API's suitability for implementing a virtual analog synthesizer for web browsers based on a reference architecture.*

The problem formulation gives birth to the sub question *what does a reference architecture for a virtual analog synthesizer look like*?

The suitability of the API will then be judged by the facilities the WAAPI provide that can be used to implement the reference architecture as well as what it doesn't provide.

1.4 Limitations

The thesis will be strictly focused on modeling analog synthesizers in terms of audio generating architecture. While auditory comparisons might be undertaken, this is not the goal of the thesis. Audio effects, that sometimes are a part of synthesizers, will not be evaluated since these already has been proven to be implementable in tuna.js [8]. The implementation used for the evaluation will be based on the reference architecture that will be synthesized from the examination of both analog synthesizers and existing virtual analog synthesizers.

1.5 Abbreviations and terminology used

ADSR - attack, decay, sustain and release
Analog Synthesizer - a physical synthesizer made with electronic circuits
Automatization - see 'tweening'
Formant filter - a filter that gives the signal a human vocal quality
LFO - low frequency oscillator, used to generate continuous modulation signals
Modulation/control signal - a signal that controls a parameter of a module
Native (browser) - something built into the browser by default
Native (API) - something provided by the API in the API's most basic form
Octave - a relation between two pitches where one is double or half the frequency of the other, or *do* to *do* using the French solfège system
Oscillator - an unit, digital or analog, that generates a cyclic audio signal
Plugin (browser) - a non-native program that extends the browser's functionality
Reference architecture - see 3.3
Tremolo - an audio effect causing the volume of a signal increase and decrease in a periodic form
Tweening - changing a value in small steps over time
(VA) Virtual Analog Synthesizer - a digital synthesizer using modeled analog synthesizer components
W3C - World Wide Web Consortium, an organizations developing web standards
WAAPI - Web Audio Application Programming Interface
Waveform - see 3.1.1

2 Method

This is an inductive thesis since no previous, similar, work could be found. Neither a reference architecture for virtual analog synthesizers, nor an evaluation of the WAAPI could be found, so these will both have to be produced in order to answer the problem formulation. This requires that methods for producing both these items are defined in a reproducible way, and this will be done in this chapter.

In the theory chapter the basic concepts of analog synthesis will be explained through a qualitative literature study.

2.1 Defining a reference architecture

In order to define a reference architecture for a VA synthesizer I'm going to examine six synthesizers, both analog and VA. For maximal data confidence I will primarily use the manufacturers own manuals or other similar references in a qualitative literature study to define three levels of specifics for my reference. I will also, when available, examine the actual instruments to verify my findings in the literature study. I choose to examine both real analog synthesizers and virtual analog synthesizers since this will give a wholesome picture of the architecture domain.

2.1.1 Level one - Top level

On this level the different sections of the examined synthesizers are identified and examined how they interact with each other. This will be modeled by making block diagrams for each synthesizer and then making an overlay where the units common to the synths are merged into one generic block diagram. This provides a birds eye view of what a VA synthesizer architecture could look like.

2.1.2 Level two - Section level

At this level the sections found in the top level block diagram are defined in terms of design and what elements they are made up of. This will also be modeled using block diagrams and provides design suggestions for the different sections of the VA synthesizer.

2.1.3 Level three - Element level

On the lowest level the common properties of the elements found, that make up the individual sections, are defined. This will be done by creating a matrix of properties and then sorting the properties into three categories;

- required - these properties can be found in all six synthesizers (90% <)
- recommended - these properties can be found in four or five of the synthesizers (60% < 90 %)
- optional - these properties can be found in two or three of the synthesizers (30% < 60)

Properties that only appear once will still be included in the matrix, but will not be a part of the reference architecture. By defining these three levels of properties, this level of the architecture provides the developer with pointers to what can be expected of the elements found in level two of the architecture. The elements themselves will be presented as UML class diagrams.

2.2 Evaluating the Web Audio API

The evaluation will be done by making an implementation of the architecture, with help from the API specification, and then evaluating the API based on the three levels of the reference architecture. The evaluation itself will also be done in three necessity levels, each appointed a score of 0 to 2;

- Native - the property is natively implemented in the API, worth 2 points
- Implementable - the API provides the tools necessary to mimic the property with some custom code, worth 1 point
- Non-native - the property cannot be mimicked without resorting to writing considerable amounts of custom code, worth no points

This allows an evaluation of the suitability of the API to implement the different levels of the properties independently. If a property is deemed implementable or non-native, arguments needs to be made why that is so that the result can be reproduced and questioned. The point scores will allow visualizations of the results as well as provide an unweighted, objective, viewpoint on the results where no single entity is more important than another.

2.3 Reliability

By using the manufacturers own documentation and, whenever available, their own diagrams the data on which the reference architecture is based should be accurate. However, synthesizers are complex systems and some simplifications might have to be done in order to produce readable block diagrams and some reasoning has to be done whether two units that are called different things in different synthesizers, but that produces the same or similar results, should be considered the same in the reference architecture.

There is also the unavoidable human element where certain properties of a synth might be missed when looking through the data set, due to the complexity of the systems. The method of deriving the architecture though should make up for these anomalies whenever the process is reproduced by another party. The categorization of elements into categories of necessity is the part most subject to bias since this will be somewhat based on the skill of the author. This should be alleviated by the fact that arguments needs to be made why things are deemed implementable or non-native.

2.4 Constraints

Due to the limited time available for the implementation, I will implement the required properties first and then progress down to the recommended properties and implement as much as possible of those. If there’s still time left I will look at implementing the optional properties too, but that will not be prioritized The optional properties will however at least be discussed if they appear to be a native, implementable or non-native.

3 Theory

This chapter will describe the basic concepts of analog synthesis and the WAAPI, as well as reference architectures. These all need to be understood in order to follow the rest of the examination and evaluation.

3.1 Analog synth architecture

The method of synthesis used by (most) analog synthesizers is called subtractive synthesis [9]. The theory in subtractive synthesis is to start with a harmonically rich waveform and, by using filters, shape the sound.

According to Eric Lyon, "the basic method of analog synthesis is to patch together self-contained modules that generate or modify electrical signals into configurations with particular sound qualities" [10]. There are three such modules that can be considered the very core of analog synthesis; the oscillator, the filter and the amplifier [11]. Moving beyond these three, virtually all synthesizers feature a modulation section.

3.1.1 The Oscillator

The oscillator is a module that generates a repeating periodic signal, called a waveform, that make up the base of an audible sound. Jari Kleimola [11] identifies a basic set of waveforms, as seen in illustration 3.1, that are used in analog synthesis;

- sine waves
- triangle waves
- square waves
- pulse waves with width modulation
- sawtooth waves
- noise

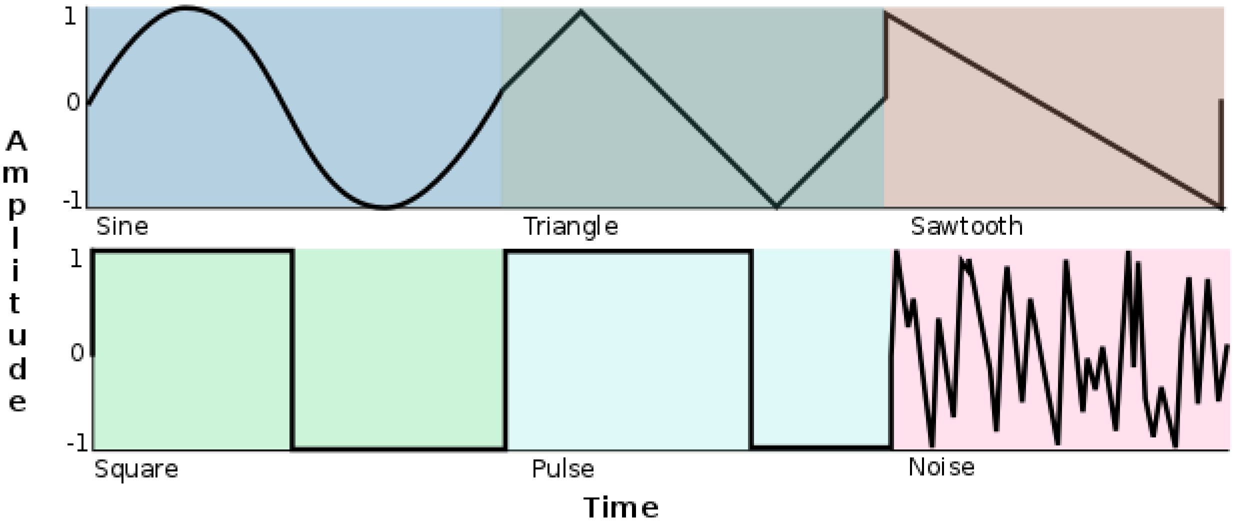


Illustration 3.1: The basic waveforms plotted over time

The sine is the purest form of waveforms which only contains the fundamental frequency of the tone and no harmonics [9]. The triangle wave is a somewhat sharper version of the sine wave with more harmonics, but still a relatively mellow waveform [9].

The square wave has a hollow, or reedy, quality. As can be seen in the lower left part of illustration 3.1 the square cycle is equally divided between positive and negative values. If this is changed, so that either the positive or negative part of the cycle becomes dominant a pulse wave is achieved. The amount of offset between the positive and negative values are called the pulse width [9].

The sawtooth gives a very bright timbre rich with harmonics and is one of the most common waveforms which can be found in virtually every analog synthesizer, according to Cann [9].

Noise, or more specifically white noise, is an “equal probability distribution of all frequencies” [12] which results in a random waveform without a recognizable visual pattern when plotted in a graph.

3.1.2 The Filter

The filter is the component that affects an analog synthesizer´s main characteristic the most. Depending of which type, or mode, of filter one uses it will add or subtract to the amplitude of certain frequencies of a sound. The filters differentiate in how hard they affect the sound, which is a property measured in decibels per octave and is called filter *slope* [9]. Virtually every filter also has a resonance property that adds some extra color to the sound by boosting certain frequencies. Common to all filters is the cutoff frequency property which is the frequency where the filter operates, whether it’s attenuating that frequency or changing all frequencies around it.

Four basic filter modes can be identified [11];

- Low pass filters
- High pass filters
- Band pass filters
- Band reject filters

The low pass filter allows low frequencies to pass through and stops the frequencies above the filters cutoff frequency. If the cutoff frequency is set to its maximum position all frequencies are let through and no modification to the sound is made [9].

The high pass filter is the opposite of the low pass filter; it lets frequencies higher than the cutoff frequency through and rejects frequencies below [9].

The band pass filter works as a combination of the high and low pass filter where only the frequencies closest to the cutoff frequency is allowed through and the frequencies above and below are rejected [9].

The band reject (more commonly referred to as “notch”) is the opposite of the band pass filter. It rejects the frequencies closest to the cutoff frequency but allows the frequencies above and below to pass [9].

3.1.2.1 Filter slopes

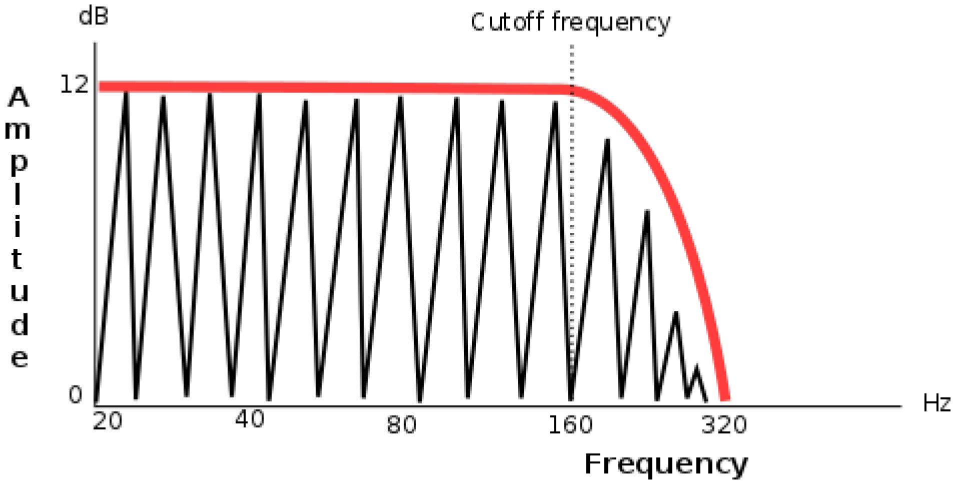


Illustration 3.2: A 12 dB/octave low pass filter

As can be seen in illustration 3.2, the filters affect the frequencies around its cutoff frequency. The amount of frequencies that are affected is decided by the filter slope. The slope is measured in dB per octave and are common in three variations; 6dB, 12dB and 24dB slopes. If the filter is a 6dB filter it will reduce, in the case of a low pass filter, the frequencies above the filter with 6dB per octave [9]. The three variations can in some cases also be referred to as 1-pole, 2-pole and 4-pole filters, respectively.

3.1.2.2 Filter resonance

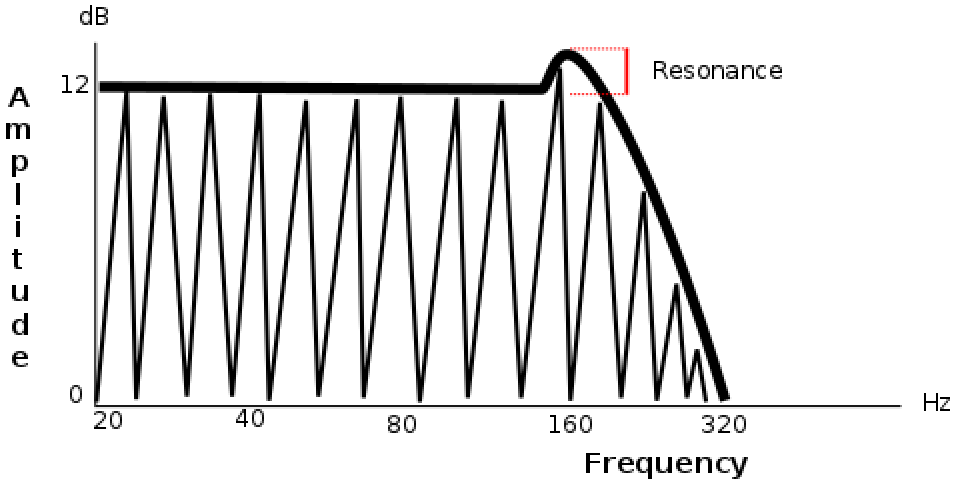


Illustration 3.3: A 12 dB/octave low pass filter with resonance

Resonance is a common feature of filters that adds an intensification at the cutoff frequency, i.e. boosting the frequencies around the cutoff frequency before starting to reduce the frequencies above, in the case of a low pass filter (see illustration 3.3). More resonance results in a bigger intensification of the frequencies around the cutoff [9].

3.1.3 The Amplifier

According to Kleimola [11], in its most basic form the amplifier simply brings up the signal generated by the oscillator (and processed by the filter) to a level that other audio devices can work with. By itself there's not much exciting going on there in terms of auditory qualities. However, that changes as we introduce one fundamental concept in audio synthesis; modulation, and in more specifically envelopes.

3.1.4 Modulation

The basic principle of modulation is that a modulation source affects a parameter called a modulation destination [13]. As an example one can let an oscillator's output (modulation source) control the level of the amplifier module (modulation destination) to create a tremolo. By doing so modulation allows the creation of animation in a sound and can make a static oscillator signal turn into a dynamic voice.

Modulation sources come in two relevant variants for a browser based VA that doesn't use a physical keyboard controller [9];

- Single-shot sources - envelopes
- Continuous sources - LFO's

3.1.4.1 Envelopes

A sound generated with normal acoustic means does rarely maintain a fixed volume throughout the lifespan of the sound; it varies over time. Analog synthesizers mimics this phenomena using envelope generators [14]. An envelope generator produces a curve of voltage that can control modulation destinations, and which can be repeated any number of times and will generate the same exact amount of voltage at any given time during the curve. There are variations, but the classic type of envelope generator is the ADSR envelope [9]. This envelope divides the voltage curve into four parts; the attack portion, the decay portion, the release portion and the sustain portion. The attack, decay and release are given as time units and the sustain is given as a level.

Assuming we're using the ADSR envelope to modulate the volume of a sound, the parts does the following; the attack affects how long it takes the sound to reach it's full volume. The decay portion governs the decrease in the volume before it levels out on the sustain level. When striking a key on a keyboard these three parts all happens consecutively from when the key was struck. So given an attack time of 500 ms, a decay time of 20 ms and a sustain level of 0.7 in a 0 to 1 scale, it takes a total of 520 ms from when the key was struck and the volume was 0 to the point where the level reached 0.7. At 500 ms the volume was whatever the volume of the amplifier was set to. As long as the key is kept pressed down the sustain level is kept, but as soon as the key is released the sustain portion takes over to bring the volume back to 0. If set to a low value the sound will be cut off when the key is released and longer sustain values gives a sort of reverberative effect on the sound, where it fades out slowly.

3.1.4.2 LFO

Just like the envelope generator can affect a modulation destination over time so can the LFO. While the envelope generator has a set time for attack, decay and release and in between keeps a steady signal at the sustain level, the LFO generates a continuous signal at a low frequency [9], generally in the sub audio spectra ($0 < n \leq 20$ Hz, though the author has seen synths that allows frequencies in the audible spectra too.). Since it's basically an oscillator that works with low frequencies it can have the same basic waveforms as ordinary oscillators and sometimes also other waveforms such as sample-and-hold [15]. Sample-and-hold generates a random value at a defined interval of time and outputs that value until it generates a new value, which generates a sort of "controlled" random signal.

Using a LFO, set to a sine waveform, to control the level of an amplifier module would cause the level to increase and decrease at the rate of the LFO's frequency and produce a tremolo effect to the sound, while doing the same with the filter frequency parameter of a filter will produce a sweeping motion, back and forth, in the sound.

3.2 Web Audio API architecture

The WAAPI is a "high level JavaScript API for processing and synthesizing audio in web applications. The primary paradigm is of an audio routing graph, where a number of AudioNode objects are connected together to define the overall audio rendering. The actual processing will primarily take place in the underlying implementation (typically optimized Assembly / C / C++ code), but direct JavaScript processing and synthesis is also supported." [2].

The central object in a Web Audio application is the AudioContext which has a number of methods defined that allows creation of specific AudioNodes. The AudioNodes all affect the audio signal in some way, be it creating signals or modifying the signals. The AudioNode interface, that is implemented by all the nodes in Web Audio, has a connect method that allows the developer to route the nodes together in a audio chain. The final node in the audio chain is the AudioContext's destination node, which in turns outputs the signal to the computer's sound hardware.

3.3 Reference Architectures

According to Governor et al [16], a reference architecture can be compared to a "somewhat abstract blueprint-type view of a system that includes the system's major components, the relationships among them, and the externally visible properties of those components." Paul Reed mention in an article [17] that a reference architecture often comes with supporting artifacts and that they often are harvested from previous projects.

4 Results and Analysis

In this chapter the results of the examinations of the synthesizers, and the reference architecture derived from this examination, will be accounted for. Then the result of the evaluation and the analysis of these results will be presented.

4.1 A reference architecture for virtual analog synthesizers

The block diagrams for each synthesizer that were inspected can be found in appendix 3. These are the diagrams from which the top level and the section level of the reference architecture were derived. The matrix that was used to derive the element level of the architecture can be found in appendix 4. Finally, the result matrix with implementation levels and point scores can be found in appendix 5.

4.1.1 Top Level

The top level architecture boils down to a rather simple model, see illustration 4.1. These sections were found in all examined synthesizers. The oscillator section generates the basic waveforms and the signals are then mixed together in the mixer section. From there the signal is sent to the filter section where the sound is shaped and finally sent to the amplifier which controls the overall level of the sound. The modulation section is a rather complex entity, but on the top level it can be pictured as able to modulate any of the other sections without specifying the specific modulation destinations.

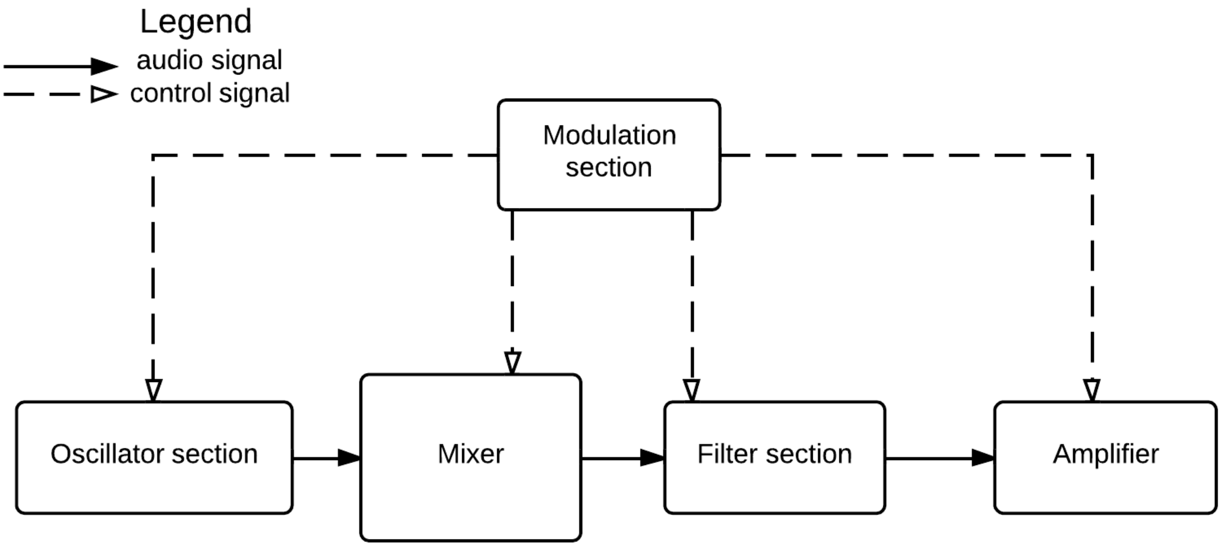


Illustration 4.1: Top level architecture

4.1.2 Section Level

On this level of the architecture the designs of the different sections were examined. The result gave a number of entities that make up each section, and also show that both the mixer and amplifier only contain a single mixer and amplifier entity, respectively, as seen in illustration 4.2.

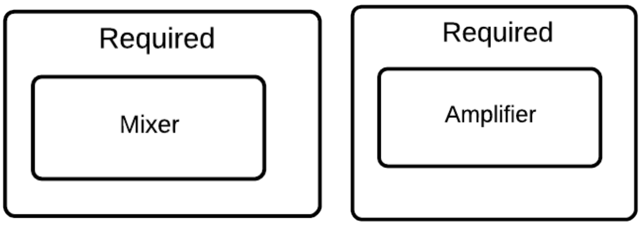


Illustration 4.2: The mixer and amplifier sections

The oscillator section, pictured in illustration 4.3, is always made up of at least two oscillators and one noise generator. Beyond that the designs tend to differ, but adding more oscillators as well as sub oscillators tend to be some of the more common options. It's also quite common to have an audio input that is mixed with the oscillator signals and then processed through the filter.

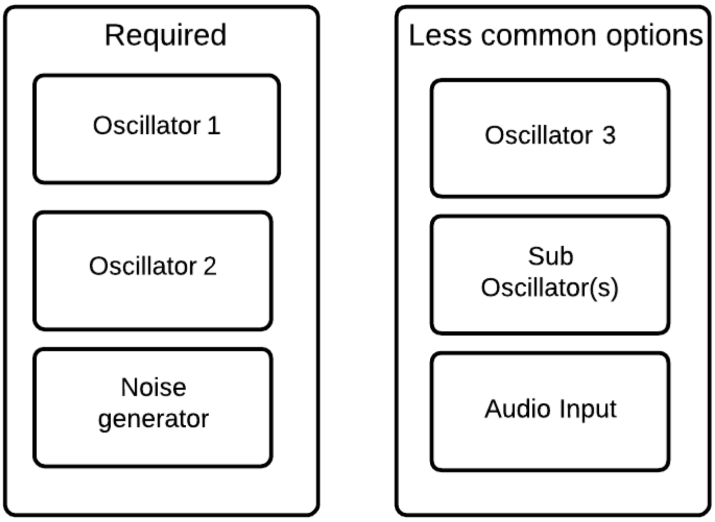


Illustration 4.3: The oscillator section

Often the filter section, seen in illustration 4.4, consists of a single filter element, but in some cases an extra filter is added and different filter configurations are possible. In serial mode, the first filter affects the signal coming from the mixer and then the second filter affects the signal coming from the first filter. In parallel mode, the mixer signal is sent to both filters separately and then the outputs of the filters are mixed together after processing.

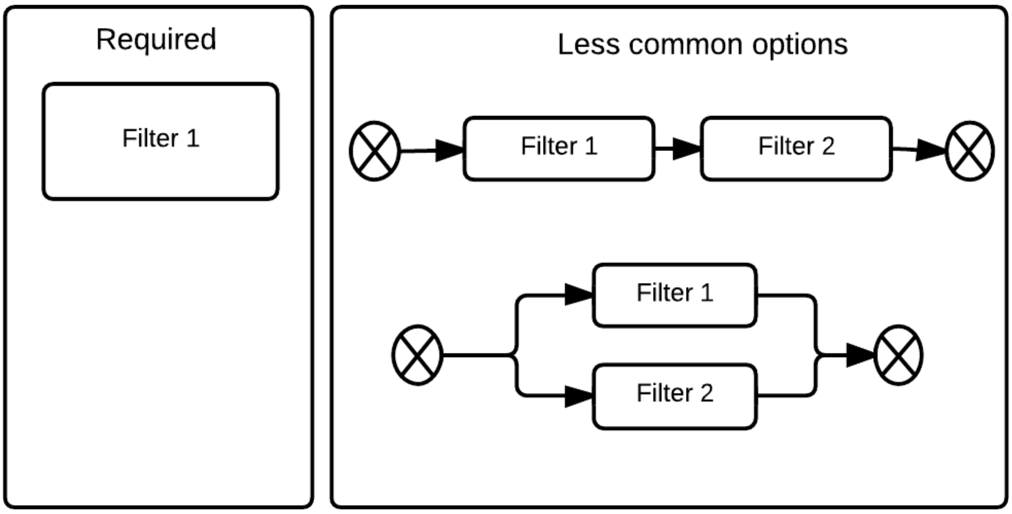


Illustration 4.4: The filter section

In the most basic case the modulation section is made up of an LFO and two envelopes. The envelopes are routed to control the filter cutoff frequency and the amplifier level, respectively. The LFO is often routed to control the pitch of the oscillators, for a vibrato effect. Beyond this the designs start to differ a lot in terms of modulation sources and destinations, and this section is the one that were found to be the most rich in variations between the inspected synthesizers, as can be seen in illustration 4.5 and table 4.1.

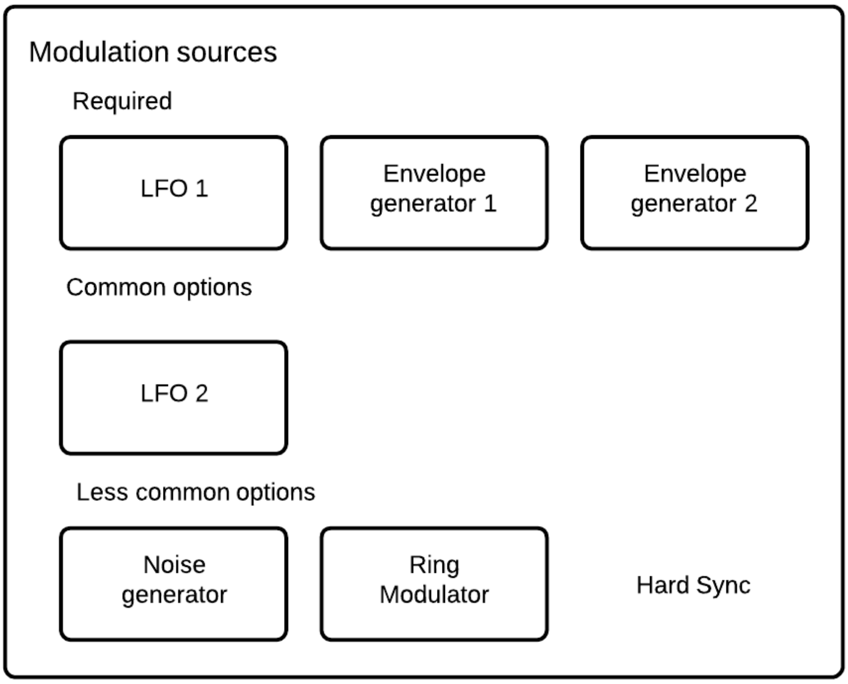


Illustration 4.5: Modulation Sources

In the hard sync mode the waveform of the second oscillator is reset whenever a waveform cycle of the first oscillator is finished, regardless of where in the cycle the second oscillator is.

Table 4.1: Modulation Destinations

Required	amplifier level	oscillator pitch	filter cutoff frequency		
Recommended	pulse width	panning			
Optional	oscillator mix	noise level	filter resonance	LFO frequency	LFO amount
	envelope gen amount	envelope gen attack rate	envelope gen decay rate	envelope gen release rate	

4.1.3 Element Level

The oscillator element has, in it's most basic form, two waveforms; pulse and sawtooth as can be seen in illustration 4.6. It also has properties for controlling the pitch of the signal and a glide property that defines how fast the signal transitions between two frequencies. Common options includes a tune property that gives a constant value offset from the pitch property value when calculating the pitch of the oscillator signal. A pulse width property can also be found that changes the pulse when in square mode (see 2.1.1).

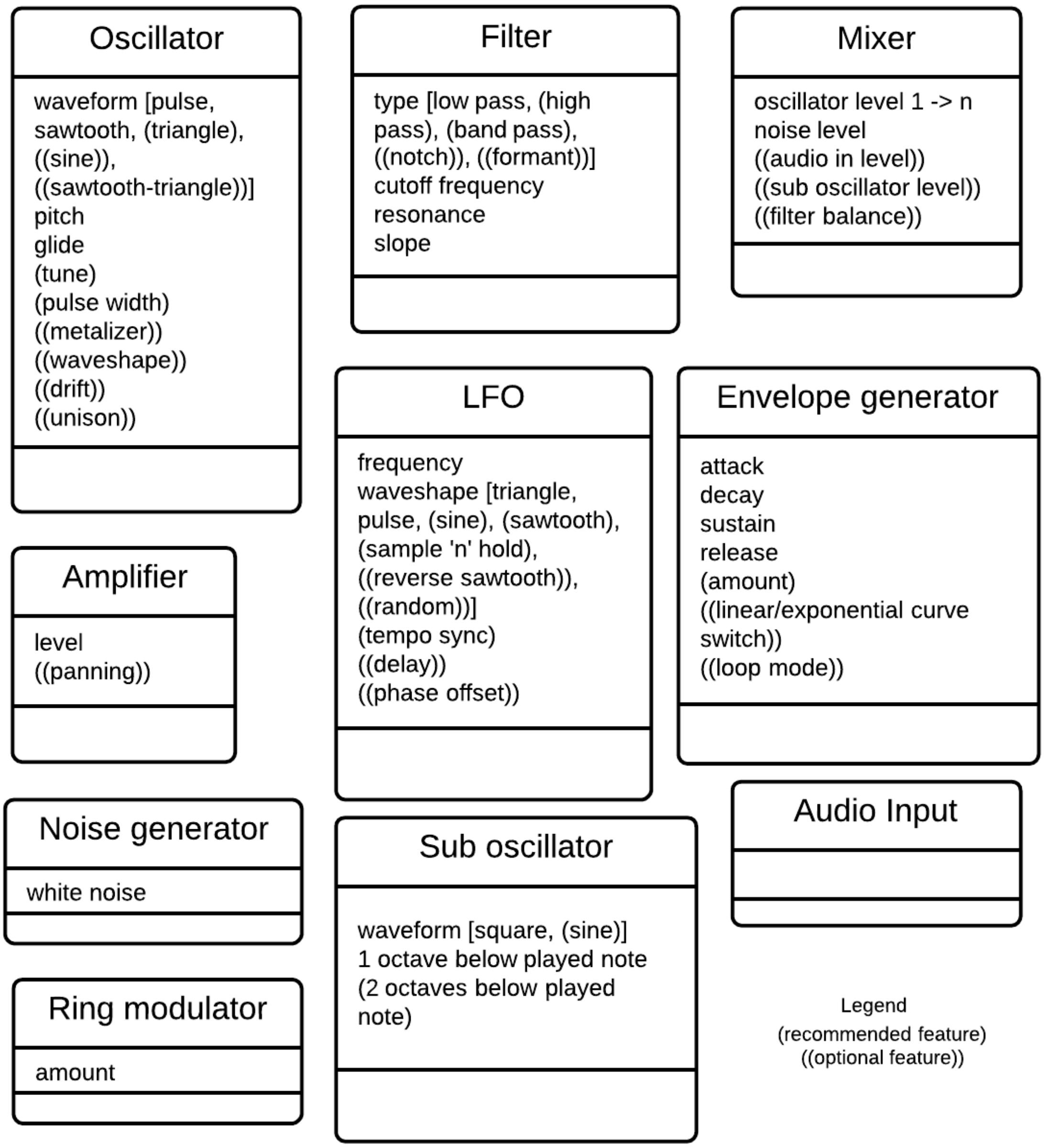


Illustration 4.6: Element class diagrams with feature necessity level

The less common options include a metalizer, wave shaper, drift and unison properties. The metalizer adds more overtones to a triangle wave which results in a metallic character of the sound. The wave shaper does different kinds of shaping of the waves depending on which waveform is selected. Both the metalizer and pulse width can be considered different types of wave shaping. The drift property add yet another offset to the pitch of the oscillator that changes slightly over time. This mimics the instability of an analog oscillator that doesn’t output a perfectly pitched signal due to heat variations etc. that can occur in an analog synthesizer. When an oscillator is set to unison mode, multiple signals are stacked on each other with slight pitch variations that together creates a richer version of the basic waveform.

The sub oscillator has a square waveform, and often a sine waveform, that is added one octave below the played note, or optionally two octaves below.

The noise generator is a simple unit that only outputs a signal called white noise. It has no other properties. The ring modulator is a unit that modulates the signal of one oscillator by multiplying the amplitude of that signal with the signal from another oscillator. It has a amount property that scales the amplitude of the second oscillator before the modulation is done.

Common to all filters in the examination is that they can all be run in low pass mode and can have a cutoff frequency set as well as a resonance. All filters also has a slope property, though it's not always possible to change this setting. Beyond this the variation of a filter element lies in its mode of operation. Common modes include high pass and band pass, while less common options include notch and formant modes.

The mixer element enables the user to change the balance between the oscillators and the noise generator. If any other signals are added in the oscillator sections, these are also available in the mixer section.

The amplifier simply sets the output level of the synthesizer and occasionally has a panning option to position the sound to either left or right in the stereo spectrum.

The LFO element is similar to the oscillator element in that it has a set of waveforms available and a frequency (pitch). Some of these waveforms are different though, more specifically the sample'n'hold option and random mode. Both of these adds a level of randomness, but while the random generates totally random values all the time the sample'n'hold mode randomizes a value at a set interval and holds that value until it's time to generate a new value.

In some cases the LFO can be set to a tempo sync mode where the frequency of the LFO is normalized to values that generates signals at the rate of subdivisions of the given tempo (eighth notes, quarter notes and so on). It's also possible to add a delay before the LFO starts modulating and to add a phase offset to the LFO waveform. The delay governs how long it will take before the LFO starts modulating its destinations from the time when a new note is played. The phase offset allows the LFO cycle to start at another location than the zero divide, meaning that one can start at the highest level of modulation or the lowest level (or anything in between) instead of the middle as would be the case without the phase offset.

The envelope generator has an attack, decay, sustain and release property as described in 2.1.4.1. Sometimes it also features an amount property that decides how big an effect the modulation should have on the destinations. This can sometimes also be set to a negative value which, for example, could make the cutoff frequency change to a lower value and then back up to the starting value. Less common options include a delay that works in the same ways as the delay for the LFO, an option to switch between exponential and linear curves between points in the envelope curve and a loop mode where the envelope curve is repeated until the note is stopped.

The optional audio input has no properties per se. Its level is controlled via the mixer element.

4.2 API evaluation results

In this section the point results will be presented as well as the level of implementation for each entity and property. The results will be discussed in chapter 5. A total score of 105 out of 164 possible were achieved by the API and the distribution of the points can be seen in illustration 4.7.

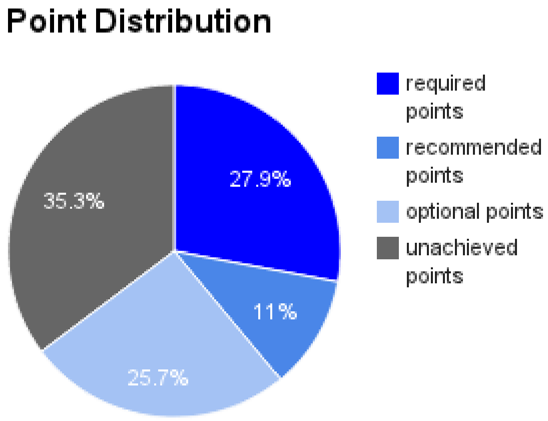


Illustration 4.7: Point distribution between architecture levels

4.2.1 Top Level

At this level the API scored 5 out of 10 possible points. All sections were deemed implementable but none were native to the API. Since the specification defines the API as "a high-level JavaScript API for processing and synthesizing audio" [2], and not a library for actually modeling synthesizer architecture, this result could be expected.

4.2.2 Section Level

At this level the API scored 13 out of 18 possible points. Where it fell short was the noise generator in the oscillator section. There is no native way of generating noise in the API without reverting to writing the noise algorithm from scratch.

Three elements were deemed implementable; the mixer, the amplifier and the envelope generator. The first two of these can be built rather easily using GainNodes from the API. According to the specification, the GainNode "is one of the building blocks for creating mixers" [2] and there is even an example of how a mixer can be implemented in Web Audio. This hints once again that the API doesn't aim at providing a complete suite of high level entities for synthesizer architectures, but rather just to provide the parts needed to implement them.

The envelope generator can be constructed using the tweening methods of the AudioParam interface. The specification provides extensive documentation and examples on how to use these methods. Since these methods are defined on the AudioParam interface it means that any property in an AudioNode that implements this interface can be assigned to an envelope generator which opens up for the more complex modulation routings found in the reference architecture.

All the other elements can be found natively in the API, even if they might not follow the same naming conventions. For example, the OscillatorNode can act as both an ordinary oscillator and an LFO in Web Audio.

4.2.3 Element Level

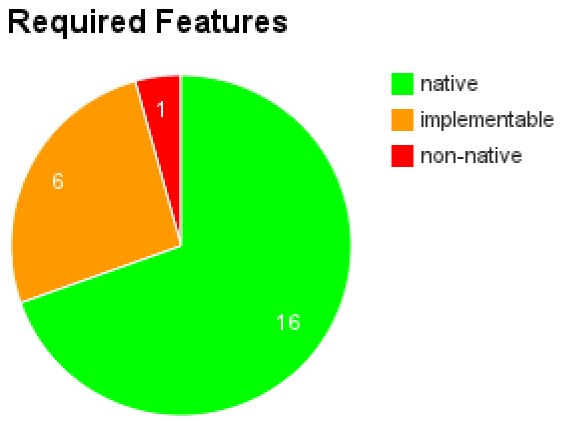


Illustration 4.8: Required features implementation levels

On the element level a total score of 88 out of 136 were achieved. Divided into the three categories of necessity the points looks like follow; the required level scored 38 out of 46, the recommended level scored 15 out of 28 and the optional level scored 35 out of 62.

As can be seen in illustration 4.8, there was only one required feature that was non-native which were the white noise of the noise generator. As previously stated, it’s not possible produce stable white noise without writing a custom noise algorithm.

There were six behaviors that were deemed implementable on the required level; oscillator glide, LFO amount and envelope generator attack, decay, sustain and release. All of these, except LFO amount, can be implemented using the tweening methods of the AudioParam since implementing the glide involves tweening the frequency property of an oscillator between notes and the envelope curve governs AudioParam’s such as the gain property on GainNodes or similar. The LFO amount can be implemented by connecting the LFO to a GainNode whose gain property is set to the amount value, and the output from the LFO is then scaled accordingly.

All other properties and behaviors on the required level were found to be native to the API.

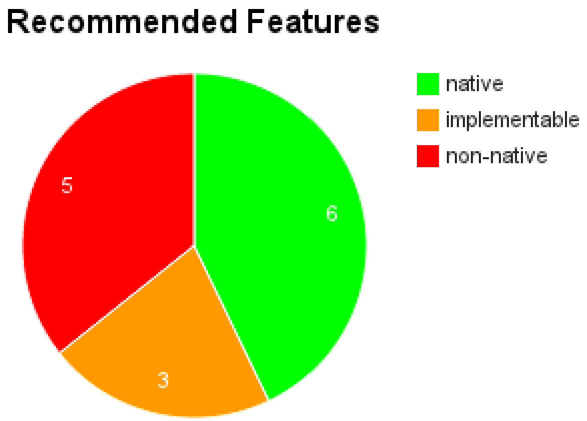


Illustration 4.9: Recommended features implementation levels

The recommended level scored ~54% and the. The main reason why that is is because the OscillatorNode would need some more options to be able to cover all the behaviors found on this level. Out of the five features deemed non-native, as seen in illustration 4.9, four are related to the OscillatorNode. Pulse width (of the square/pulse wave), sample'n'hold, hard sync and pulse width as a modulation destination all fell short since this can't be done with the native oscillators, meaning that the oscillator would need to be coded from the ground up to allow access to the phase of the waveform as well as manipulation of the waveforms themselves.

The current oscillator implementation only exposes a frequency and detune property. It does however allow the use of custom waveforms, and thus enables the use of the pulse width property through the setWaveTable method. This requires though that the developer calculates the waveform in real time whenever the pulse width property is changed and this were deemed too big a custom job to give the credit to the API itself. The sample'n'hold behavior could be mimicked using the setWaveTable method too, though this wouldn't be "true" sample'n'hold since the randomization pattern would eventually loop and no true randomness would be achieved. Hard sync would require access to the phase of the waveform in real time since it would have to be reset on the slave oscillator whenever the master oscillator reached a full wave cycle, and this is not possible in the API.

The panning in the amplifier was deemed implementable by splitting the signal that arrived at the amplifier using a ChannelSplitterNode and then connecting the split signal to two separate right and left GainNode's which allows setting the level of both stereo channels. These two signals are then mixed together again using a ChannelMergerNode. However, since it's not possible to control the phase of an LFO (which is the modulation source we most likely would like to use when modulating panning), it was not deemed possible to implement panning as a modulation destination since the gain of the GainNode's would need to move in opposite directions to get the correct effect. Also worth mentioning is that there is a native PannerNode in Web Audio, though this performs panning in a 3D-space and fails to produce pure stereo panning which makes it unusable for the purpose of panning in a synthesizer.

Beyond that everything is native to the API except the option to tempo sync the LFO and the envelope generator amount. The latter can be implemented in the same way as the amount for the LFO and the tempo sync is some rudimentary maths to calculate the proper frequency for the LFO to use, so it was considered implementable too.

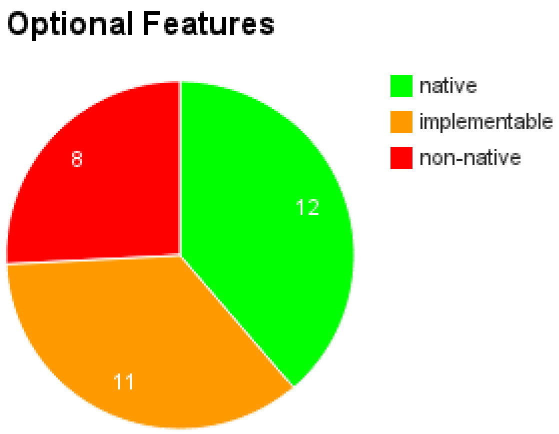


Illustration 4.10: Optional features implementation levels

Finally, the optional features scored ~56% and 8 features were deemed non-native, as seen in illustration 4.10. The main reason why that is can again be found in the closed implementation of the oscillators in Web Audio. There was a random waveform missing in the oscillators for use in the LFO. The LFO also called for a phase offset setting that isn’t native, as stated above. Since noise isn’t native to the API this was also gave 0 points as it was needed as a modulation source in the reference architecture. Also found missing were a formant filter mode.

A pair of waveforms, reverse sawtooth and sawtooth-triangle, were missing from the oscillators, but these can be implemented using the setWaveTable method and were given one point each since these only needs to be calculated once and never change. A metalizer effect can easily be achieved by using a DelayNode and setting it to low delay time values and then feeding it back into the oscillator signal path. Oscillator drift can be achieved by changing the detune property of an oscillator with an LFO by small amounts and an oscillator unison mode can be created by simply creating multiple oscillators for each note and slightly detune them from each other using the detune property of the oscillators.

Filter balance of the mixer could also be implemented by setting up a routing with GainNode’s that allows the user to decide how much of each signal source should be sent to each filter, provided that a filter section is used with more than one filter running in parallel mode.

A LFO delay should be implementable by adding a timeout in the LFO object before starting the LFO when a new note arrive, alternatively turning the amount down to 0 and then turn it up to the set value after the timeout. Switching between linear and exponential curves in the envelope generator should also be implementable since there are tween functions for both types of curves. Ring modulation should be implementable by connecting the first oscillator to a GainNode and then having the second oscillator modulating the gain of than GainNode, which means that the amplitude of the two oscillators are multiplied and ring modulation is achieved.

Oscillator mix as a modulation target should be implementable since the level of the oscillators are governed by GainNode's in the mixer section, and these can be the modulation destinations for both the LFO and envelope generators. Envelope generator amount as a modulation destination should be implementable for the same reason since it can be implemented with a GainNode. However, envelope generator attack, decay and release as modulation destinations were found non-native since these are custom properties in a custom implementation, or in other words; they do not implement the AudioParam interface and thus cannot be the target of either the LFO or another envelope generator. This will be discussed further in chapter 5.

The remaining properties of this level were all found to be native to the API.

4.3 Miscellaneous findings

During the implementation a few unexpected behaviors were found in the API that didn't affect the outcome of the evaluation but are worth noting. These will be presented here.

If the two oscillators are set to the same waveform and the exact same frequency it does happen that one of the oscillators goes out of phase with the other oscillator which results in a thinner sound and an audible movement in the sound as they repeatedly align and misalign. This seems to happen only when the pitch is changed.

To remedy the fact that properties that aren't AudioParams can't be modulated a test were done to see if it was possible to build a custom modulation destination but still use an OscillatorNode as an LFO. The idea was to connect the output of the oscillator to control a GainNode's gain property and continuously update the modulation destination value based on the gain property value of that GainNode. It turns out that modulating parameters via the native modulation routings doesn't update the exposed property though, so whenever the gain property of the GainNode was read, it always reported a value of 1 which resulted in the modulation destination never being updated.

5 Discussion

5.1 Implementation

Both in the score results and in the implementation the modulation section has brought the general impression down. Implementation wise, this is not entirely the API's own fault though. The modulation section is a large and complex entity which poses some challenging design choices; who is in charge of deciding what destination should be modulated by what source and how much do the sections and elements in the synthesizer know about each other? Having a set number of modulation routings hard coded alleviates this issue, but as soon as some customization options are introduced this becomes a real problem. If the API had been able to modulate other properties than AudioParams, like floats and integers, it would have been more simple to implement this but since this is not possible one would have to rewrite the LFO and envelope generator to be able to modulate custom destinations. This obviously makes things even more complex.

The noise generator had a big impact on the scores as well. In reality, this isn't that big an issue since it's actually rather trivial to implement a noise generator. It's just a matter of filling a buffer with random numbers in a JavaScriptProcessingNode. There's even a small third-party library [18] that can do this and some other useful things, though the library was not used in my implementation.

Beyond these two issues I think that the API truly helped making much of the implementation a trivial task. The most important thing, I think, is to have the necessary domain knowledge to be able to put the API to best use. While some things requires more work than other, I'd say that pretty much anything is possible, even if it's considered non-native in the evaluation. This is a strong point of the API; it provides the option to write totally custom code via the JavaScriptProcessorNode and then insert it into the Web Audio chain of nodes.

One thing I did find troubling, however, is the oscillators going out of phase occasionally, as described in 4.5. One possible source of error could be that I'm only using a single OscillatorNode for each waveform per oscillator element, throughout the lifetime of the application, while API is designed to create a new OscillatorNode per note played. The reason why I took this approach was to modulate the analog synthesizers where the oscillators are running all the time and each section works independently of each other. This design allows me to do volume envelopes on the amplifier section without having the oscillators knowing when it should stop. It's possible that the phase offset happens when the oscillator frequency property is changed as a new note is played and thus could be fixed by creating new OscillatorNode's for each note, though this would require a different code structure entirely. Another likely solution would be to implement the drift behavior on the oscillators which would probably render the issue obsolete since the frequencies will never quite match.

One big caveat is that this implementation of the API currently only is available in Google Chrome. No other browser, even Safari that has a Web Audio API implementation, have the same up-to-date implementation as Chrome.

5.2 Results

A total score of 105 out of 164, or ~64% for the API seems reasonable to me. Had everything been deemed implementable the score would have been 50%, so this score hints that the API can handle a lot of the behaviors needed to implement the reference architecture. On the required level there was only a single thing that wasn't native or implementable, and that was the noise generator which, as stated before, isn't a big deal since it's so easy to implement with custom code. Beyond this it was only pulse width, a recommended option, and a formant filter, an optional feature, that was missing for the API to support a full implementation of the oscillator, mixer, filter and amplifier sections. The pulse width would have been useful, but all in all, I think that the API did well so far.

The big letdown is made clear when looking at the scores for the modulation section. Here the closed implementation of the OscillatorNode starts to be a problem. In many cases direct access to the phase of the waveforms, and the ability to affect it, would have been enough to score more points. The ability to natively modulate properties that aren't AudioParam's would also have done great things for the score, and definitely made it easier to come up with a design for the modulation routing that is scalable and reusable.

5.3 Method

The method I used for synthesizing the reference architecture in itself is a good one, in my opinion. The obvious flaw is the human element where someone needs to look through user manuals and documentations to extract properties and behaviors The strength of the method, though, is that the data set will be improved each time someone repeats the process and thus the reference architecture will become increasingly accurate the more synthesizers that examined and already examined ones are processed a second or third time.

Categorizing the found properties and behaviors in native, implementable and non-native categories also makes sense to me since that gives the evaluation objective onset, but it also allows for subjective interpretations of the result. It clearly states what one would have to put most work into when doing an implementation. Giving each category a point value opens up for some graphical visualizations of the result that can be interesting, but at the same time I felt that it also could be misleading. In my implementation I felt like I had more aid from the API than the numbers let on. This could potentially be because I had a 0 to 2 range. I think the numbers might reflect my experience better if I'd given both the native and implementable categories a 1 point value each. If I'd done that the result would have been 82% of the total possible points versus the 64% that the 0 to 2 range gave.

I also feel that the rules for how the behaviors and properties should be categorized can be further defined for an even more objective categorization This would have been hard to do without first doing the work behind this thesis, and see what situations that actually presented themselves though, and there will always be a need of a certain amount of judgment whenever ambiguous situations appear.

6 Conclusion

The problem formulation of this thesis was *an evaluation of the Web Audio API's suitability for implementing a virtual analog synthesizer for web browsers based on a reference architecture*, and it brought with it the sub question *what does a reference architecture for a virtual analog synthesizer look like*? To answer these questions a method for synthesizing a reference architecture were developed and a categorization and scoring system were used to evaluate the WAAPI. A reference architecture were then produced using the proposed method and an implementation of the architecture were then done using the WAAPI and evaluated thereafter.

6.1 Summary

The API were found to provide much of the functionality requested in the reference architecture, with the exception of a few key elements. The evaluation highlights a need for a native noise generator, a more open oscillator implementation with access to the raw waveforms to enable more modulation options and a native way to automate custom numeric properties to help creating generalized modulation sections. The reference architecture with accompanying comments can be found in 4.3.

6.2 Further studies

I was surprised to find that there wasn't a standardized method for producing reference architectures. I don't know if this is because the method would be different depending on architecture domain areas or if there is some other reason. I would definitely like to read a paper or similar on the subject, since I've found reference architectures to be a powerful instrument for communicating ideas and a great foundation for implementations, though I think they might have limited use if the process of generating these architectures isn't standardized. It's possible that things that aren't obvious to the reader is left out because the author thinks it's so, without such a standardized method.

I would also like to read about a further examination of modulation sections and how these can be designed. This was the biggest challenge for me in my implementation, and my reference architecture doesn't provide any hands on suggestions on how things can be structured, only what elements can be included.

Appendix

1. Online sources

[1] Rogers. C, "Web Audio API is now available in Chrome", Pub: January 31, 2011 [Online]
http://lists.w3.org/Archives/Public/public-xg-audio/2011Feb/0000.html April 11, 2013

[2] Rogers. C, "Web Audio API", Pub: December 13, 2012 [Online]
http://www.w3.org/TR/webaudio/ April 11, 2013

[3] Beverloo. B, "Sub-pixel layout, Inspecting Web Socket Frames and Seamless Iframes", Pub: May 9, 2012 [Online]
http://peter.sh/2012/05/sub-pixel-layout-inspecting-web-socket-frames-and-seamless-iframes/
April 11, 2013

[4] Hurst. J, "Robert Moog's 78th Birthday", Pub: May 23, 2012 [Online]
http://www.google.com/doodles/robert-moogs-78th-birthday May 22, 2013

[5] "oscillator.js – bob-moog-google-doodle", Pub: June 29, 2012 [Online]
https://code.google.com/p/bob-moog-google-doodle/source/browse/oscillator.js May 22, 2013

[6] Wilson. C, "cwilson/midi-synth – Github", *GitHub,* Pub: August, 2012 [Online]
https://github.com/cwilso/midi-synth May 22, 2013

[7] McClure. R, "secretfeature.com/mono-synth", Pub: May, 2013 [Online]
http://secretfeature.com/mono-synth/part4/ May 22, 2013

[8] Eriksson. O, Coniglio. C, Saccoia. A, "Dinahmoe/tuna – GitHub", *GitHub,* Pub: November, 2012 [Online]
https://github.com/Dinahmoe/tuna May 22, 2013

[13] Rolando. T, "MakeNoise Modular Synthesis Workshop, Pt. 1 : Tony Rolando, Intro to Modular Synthesis on Vimeo", *Vimeo,* Pub: April, 2013 [Video]
http://vimeo.com/63407413 April 15, 2013

[17] Reed. P, "Reference Architecture: The best of best practices", *IBM developerWorks,* Pub: September 15, 2002
http://www.ibm.com/developerworks/rational/library/2774.html April 19, 2013

[18] Diamond. M, "mattdiamond/synthjs – GitHub", *GitHub,* Pub: September, 2012
https://github.com/mattdiamond/synthjs May 23, 2013

2. Published sources

[9] Cann. S, *How to Make a Noise: Analog Synthesis*, 3rd. ed. New Malden: Coombe Hill Publishing, 2011, E-book

[10] Lyon. E, Editors: Boulanger. R, Lazzarini. V, *Audio Programming Book*, Cambridge: The MIT Press, 2011, p. 629

[11] Kleimola J, *Design and Implementation of a Software Sound Synthesizer*, Helsinki University of Technology, Master's Thesis, 2005

[12] Mitchell D, *Basicsynth*, *creating a music synthesizer in software*, 2008, E-book

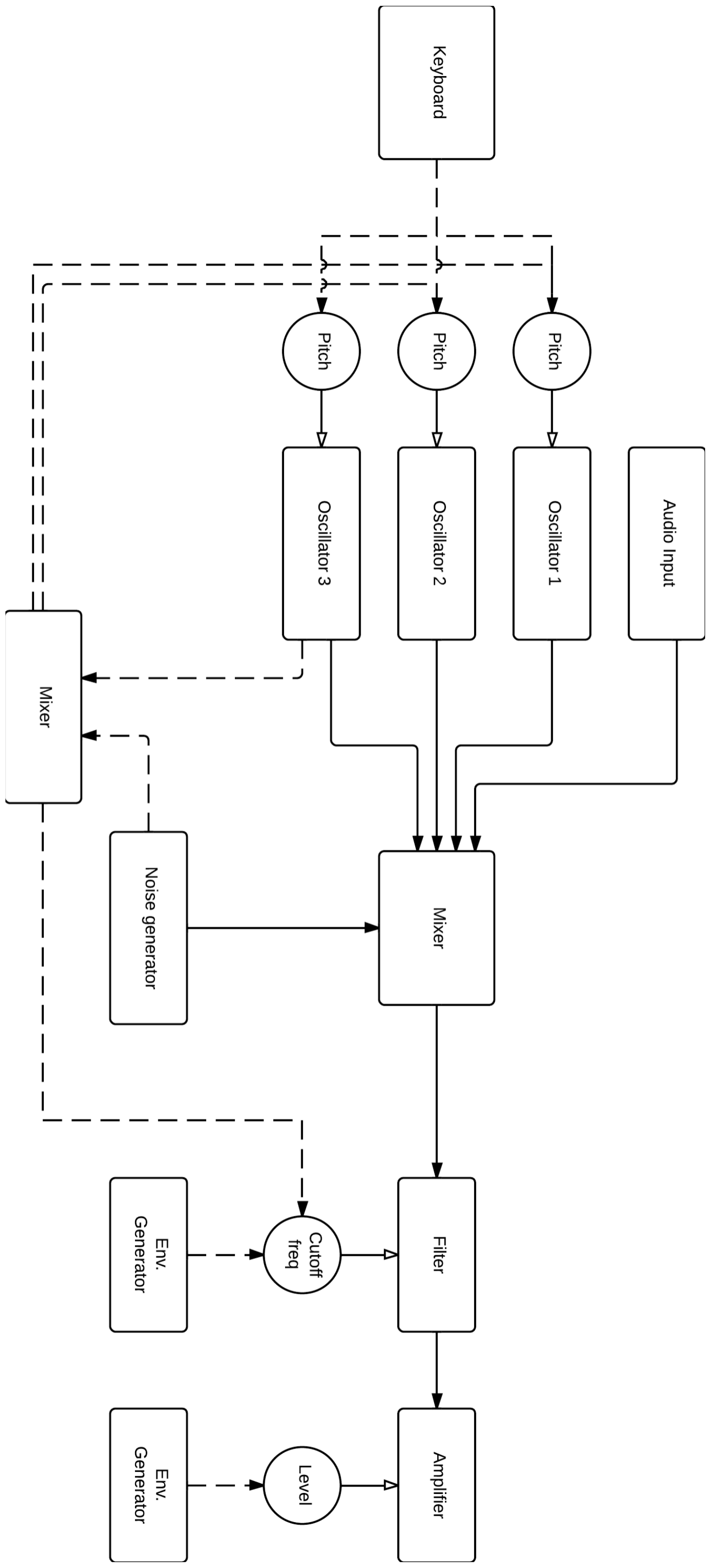
[14] Zimmer. H, Editor: Martin G, *Musikmakarnas handbok* (*org. Making music*), swedish edition, Stockholm: P. A. Nordstedt & Söners Förlag, 1985, page 118

[15] Ekström. L, *Stora Midiboken*, Stockholm: Warner/Chapell Music Scandinavia AB, 1996

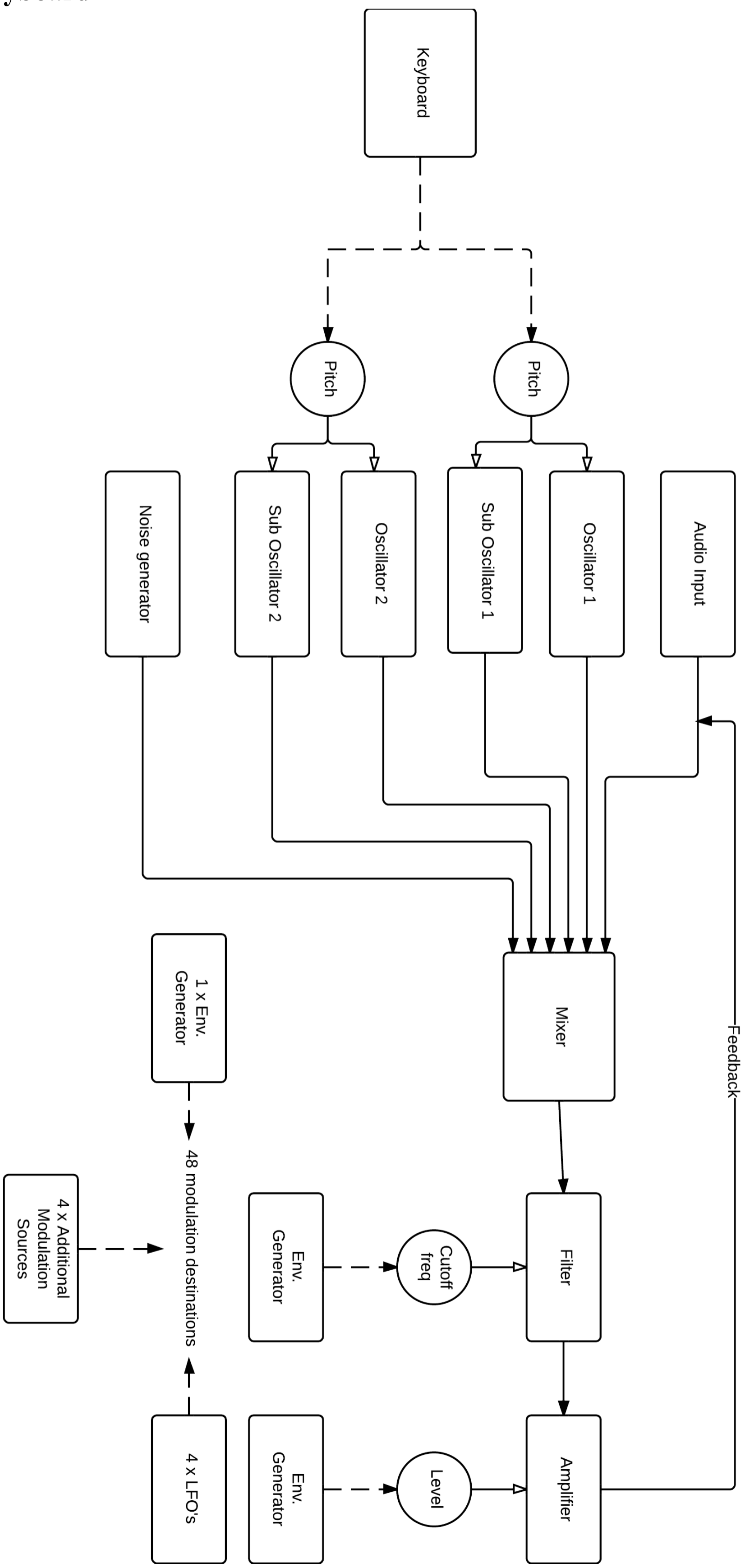
[16] Governor J, Hinchcliffe D, Nickull D, *Web 2.0 Architectures*, Sebastopol: O'Reilly Media Inc., 2009

3. Synthesizer block diagrams

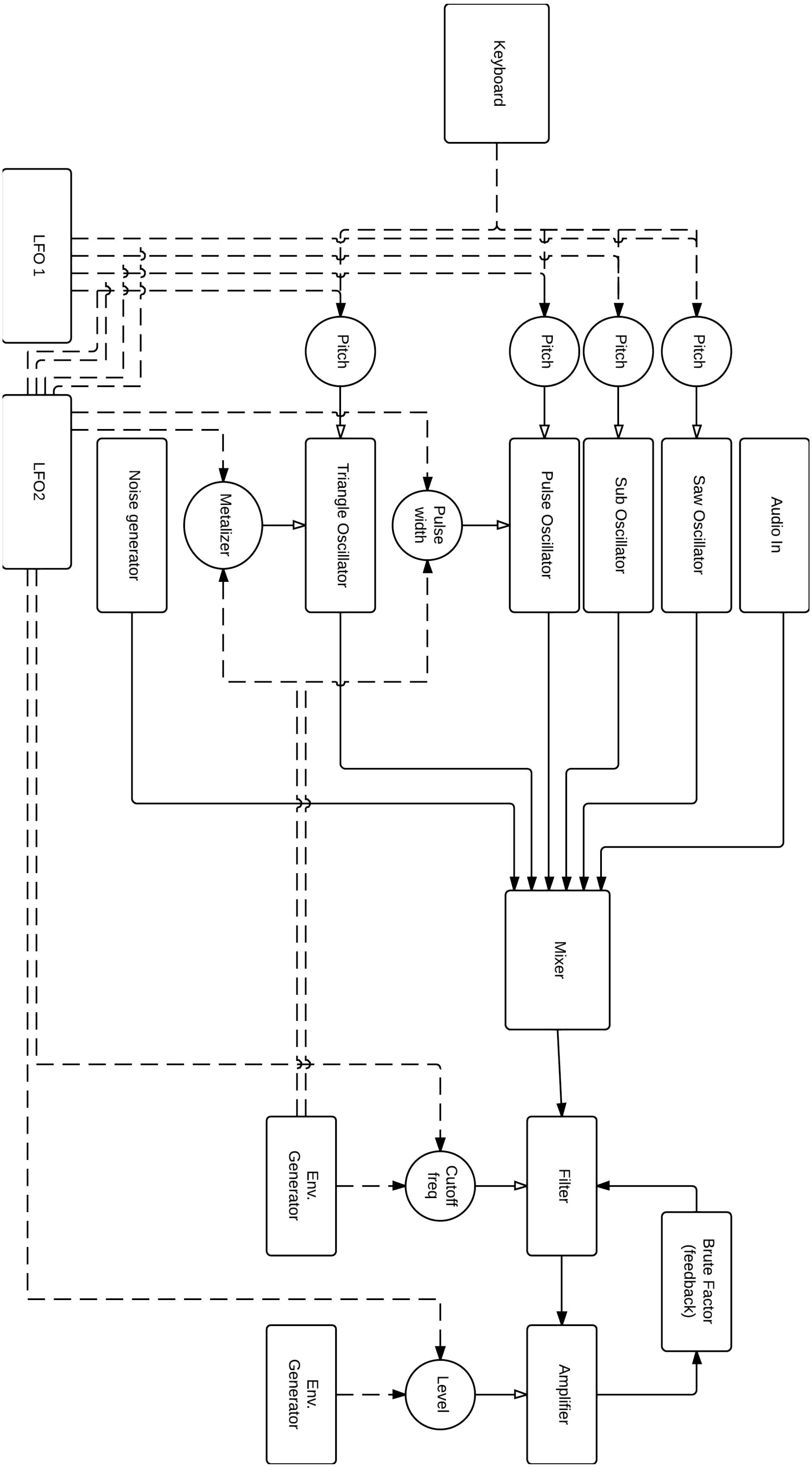
3.1 Mini Moog



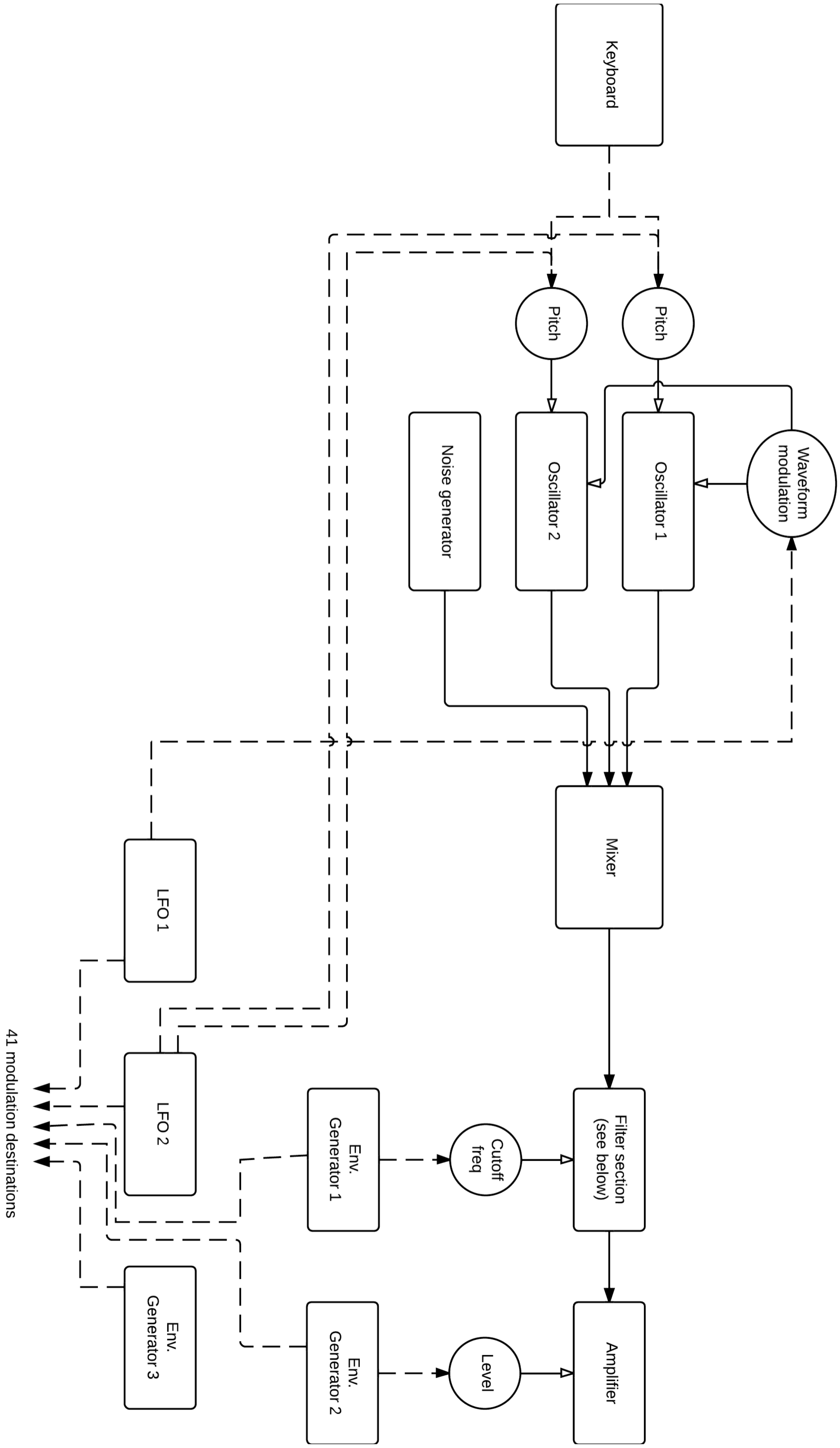
3.2 Mopho Keyboard



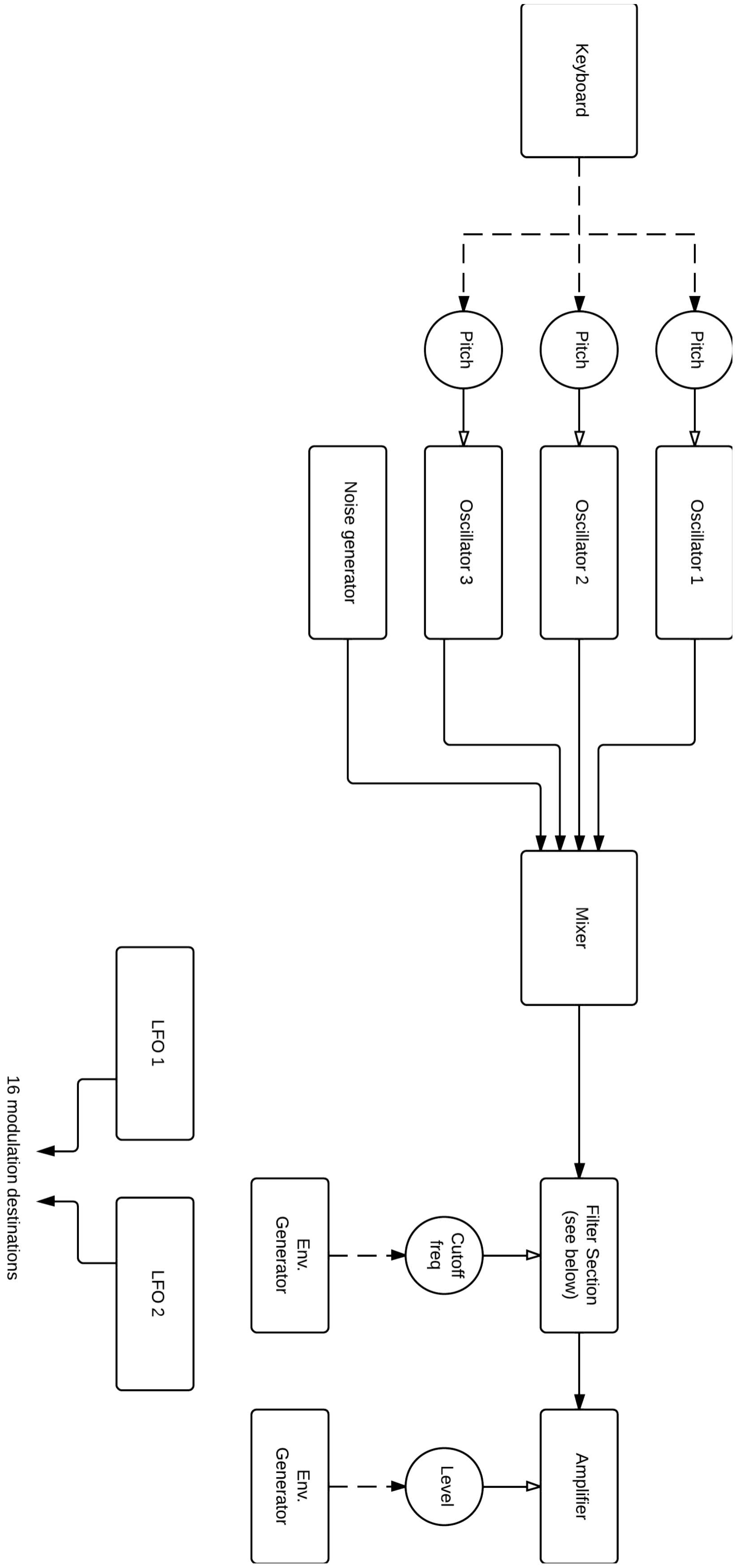
3.3 MiniBrute



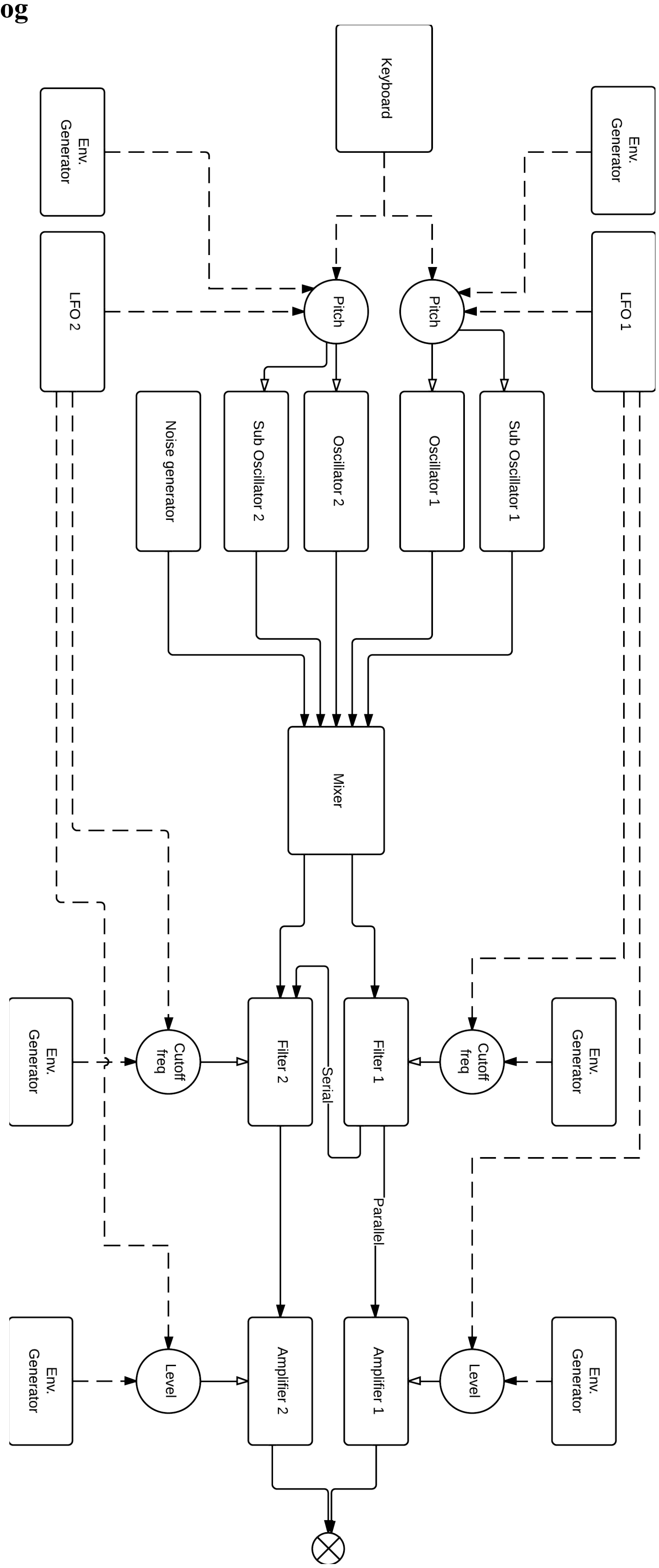
3.4 MicroKORG XL



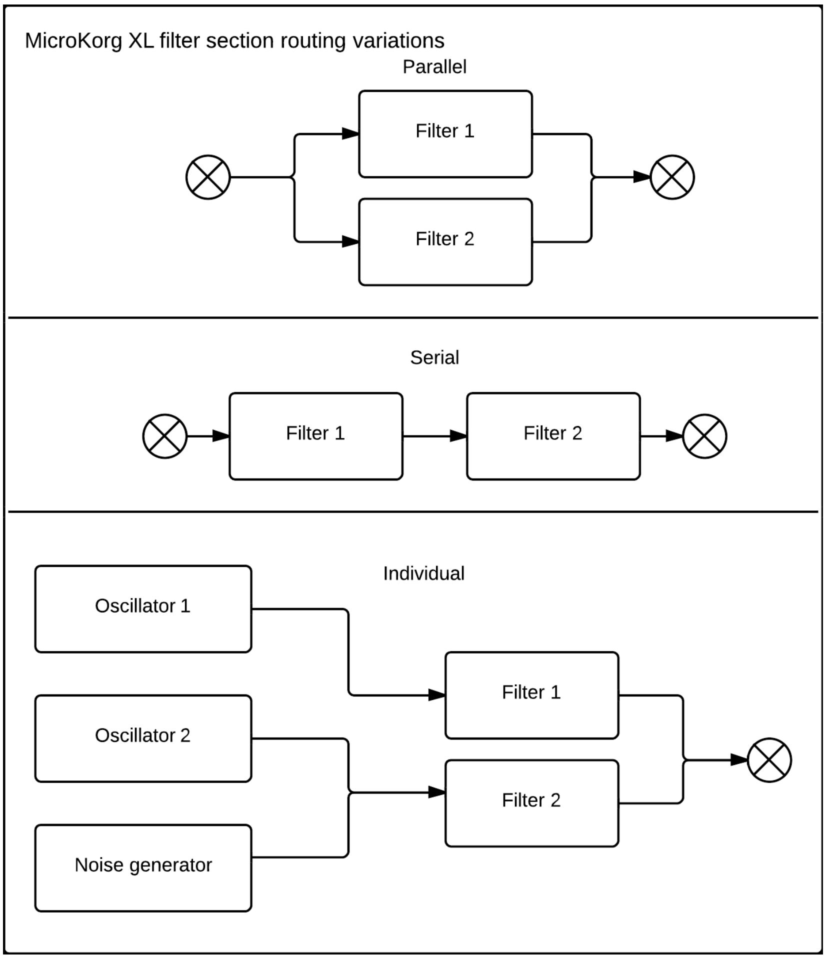
3.5 Magellan


Keyboard
Pitch
Pitch
Pitch
Oscillator 1
Oscillator 2
Oscillator 3
Noise generator
Mixer
Filter Section
(see below)
Env.
Generator
Cutoff
freq
Amplifier
Env.
Generator
Level
LFO 1
LFO 2
16 modulation destinations

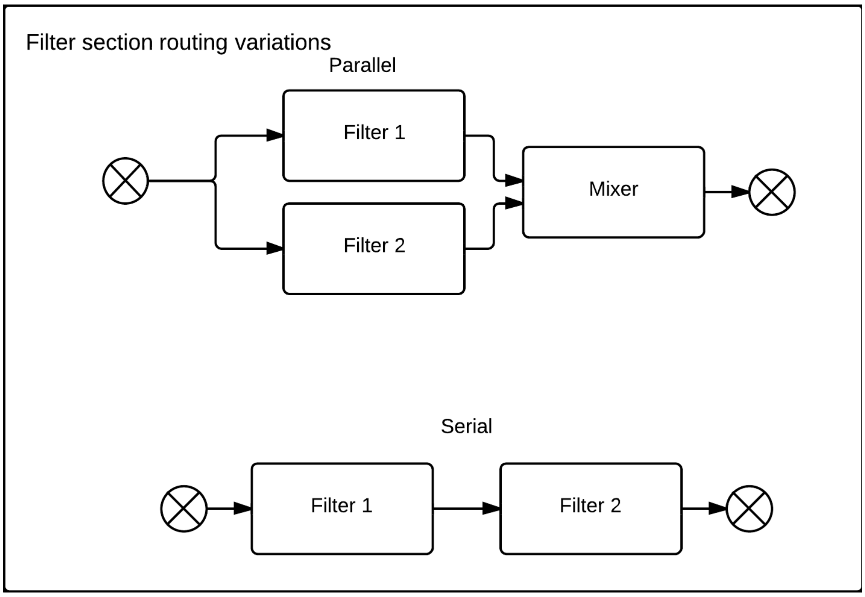
3.6 Ableton Analog



3.7 MicroKORG XL filter section routings



3.8 Magellan filter section routings



4. Feature matrix

Name:	Mini Moog	MiniBrute	Mopho Keyboard	microKORG XL	Magellan	Ableton Analog
Oscillator props:						
	pitch	pitch	pitch	pitch	pitch	pitch
	tune (octave & cent)		tune (semi & cent)	tune (semi & cent)	tune (octave, semi & cent)	tune (octave, semi & cent)
		PWM (in square mode)	PWM (in square mode)	PWM (in square mode)	PWM (in square mode)	PWM (in square mode)
					phase shift	
				waveshape	waveshape	
				unison	unison	unison
	glide	glide	glide	glide	glide	glide
			drift (slight pitch variation)	drift		drift
		ultrasaw amount (in sawtooth mode)				
		ultrasaw rate (in sawtooth mode)				
		metalizer (in triangle mode)		VPM (add metallic overtones, all waves)		
Oscillator Waveshapes:						
	triangle	triangle	triangle	triangle	triangle	
	sawtooth-triangular (osc 1 & 2 only)		sawtooth-triangle		sawtooth-triangle	
	sawtooth	sawtooth	sawtooth	sawtooth	sawtooth	sawtooth
	square	square	square		square	
	wide & narrow pulse			pulse	pulse (3rd oscillator only)	pulse
				sine		sine
	reverse sawtooth (osc 3 only)					
	noise	noise	noise	noise	noise	noise
		sub square				
		sub sine				
				formant		
Mixer props:						
		sub level	sub level			sub level
					filter balance (send)	filter balance (send)
	oscillator level (1, 2 & 2)	oscillator level (triangle, saw, square)	oscillator mix (1 & 2)	oscillator level (1 & 2)	oscillator level (1 to 3)	oscillator level (1 & 2)
	noise level	noise level	noise level	noise level	noise level	noise level
	audio in level	audio in level				
				punch level		
			feedback level			
Filter props:						
	cutoff frequency	cutoff frequency	cutoff frequency	cutoff frequency	cutoff frequency	cutoff frequency
	resonance	resonance	resonance	resonance	resonance (magellan & victor)	resonance
			12 or 24 dB/octave switch			
			audio mod (osc 1 modulates cutoff)			
					Q (voyvoda, comb, all pass, notch)	
						drive
				filter slope amount		
Filter types:						
	low pass 24 dB/octave	low pass 12 dB/octave	low pass 12 or 24 dB/octave	low pass 12 or 24 dB/octave	low pass (magellan, victor, voyvoda)	low pass 12 or 24 dB/octave
		high pass 12 dB/octave		high pass 12 dB/octave	high pass (victor, voyvoda)	high pass 12 or 24 dB/octave
		band pass 6 dB/octave		band pass 6 dB/octave	band pass (victor, voyvoda)	band pass 6 or 12 dB/octave
		notch 6 dB/octave			notch	notch 12 or 24 dB/octave
					all pass	
					comb	
					formant	Formant 6 or 12 dB/octave

Name:	Mini Moog	MiniBrute	Mopho Keyboard	microKORG XL	Magellan	Ableton Analog
Amplifier props:						
	level	level	level	level	level	level
		brute factor (feedback)				
				drive		
			panning	panning	panning	panning
Modulation sources:						
				cross mod (o2 modifies pitch of o1)		
			hard sync	hard sync	hard sync	hard sync
				ring mod	ring mod	
	LFO	LFO x 2	LFO x 4	LFO x 2	LFO x 2	LFO x 2
			audio in envelope follower			
			audio in peak hold			
	noise		noise			
	envelope generator x 2	envelope generator x 2	envelope generator x 3	envelope generator x 3	envelope generator x 3	envelope generator x 6
Modulator destinations:						
	oscillator pitch	oscillator pitch	oscillator pitch	oscillator pitch	oscillator pitch	oscillator pitch
	filter cutoff frequency	filter cutoff frequency	filter cutoff frequency	filter cutoff frequency	filter cutoff frequency	filter cutoff frequency
	amplifier level	amplifier level	amplifier level	amplifier level	amplifier level	amplifier level
		PWM width	PWM width	PWM width	PWM width	
		metalizer amount				
			oscillator mix	oscillator mix	oscillator mix	
			noise level	noise level	noise level	
			filter resonance	filter resonance		filter resonance
			audio in modulation amount			
			LFO frequency	LFO frequency	LFO frequency	
			LFO amount	LFO amount		
			envelope gen amount	envelope gen amount		envelope gen amount
			envelope gen attack rate	envelope gen attack rate		
			envelope gen decay rate	envelope gen decay rate		
			envelope gen release rate	envelope gen release rate		
			sub oscillator level			
			feedback level			
			oscillator mod amount			
					ring modulation depth	
				glide		
				waveshape depth		
				filter balance		
				formant width		
				formant shift		
				waveshape		
				osc detune (in unison)		
				osc phase (in unison)		
				cross mod depth		
				mod harmonics (in VPM)		
				virtual patch intensity (1 to 6)		
			panning	panning	panning	panning
Envelope generator properties:						
standard type:	ADR (filter), ADSR (amp)	ADSR	DADSR	ADSR	ADSR	ADSR
						trigger mode
			loop mode			loop mode
						AD-R mode
						ADR-R mode
						ADS-R mode
						legato mode
					linear/exponential switch	linear/exponential switch

Name:	Mini Moog	MiniBrute	Mopho Keyboard	microKORG XL	Magellan	Ableton Analog
LFO properties:						
	frequency	frequency	frequency	frequency	frequency	frequency
		tempo sync	tempo sync	tempo sync	tempo sync	tempo sync
			key sync			
						PWM (square or triangle)
					delay	delay
						retrig
					phase offset	phase offset
	amount	amount	amount	amount	amount	amount
LFO waveshapes:						
		sine		sine	sine	sine
	triangle	triangle	triangle	triangle	triangle	triangle
	pulse	pulse	pulse	pulse	pulse	pulse
				positive pulse		
						noise
	sawtooth	sawtooth	sawtooth	sawtooth	sawtooth	
	reverse sawtooth		reverse sawtooth		reverse sawtooth	
		sample 'n' hold	sample 'n' hold	sample 'n' hold	sample 'n' hold	
		random		random	random	
Noise gen properties						
	white noise	white noise	white noise	white noise		white noise
					noise color	
						filter balance (send)
						low pass 6 db/octave
	pink noise					
Sub oscillator properties						
		1 or 2 octaves below	1 or 2 octaves below			1 octave below
Sub oscillator waveforms						
		square	square			square
		sine				sine
Audio input properties						
Ring mod properties						
				amount	amount	

5. Categorization and point matrix

Section	Element	Feature	Impl. Level	Point	Top level	Sections	Required features	Recommended features	Optional features
Oscillator section			implementable	1	1				
	Oscillator		native	2		2			
		pulse/square	native	2			2		
		sawtooth	native	2			2		
		glide	implementable	1			1		
		pitch	native	2			2		
		triangle	native	2				2	
		tune	native	2				2	
		pulse width	non-native	0				0	
		sawtooth-triangle	implementable	1					1
		sine	native	2					2
		metalizer	implementable	1					1
		waveshape	native	2					2
		drift	implementable	1					1
		unison	implementable	1					1
	Noise generator		non-native	0		0			
		white noise	non-native	0			0		
	Sub Oscillator		native	2		2			
		square	native	2					2
		sine	native	2					2
		1 octave below	native	2					2
		2 octaves below	native	2					2
	Audio input		native	2		2			
Filter section			implementable	1	1				
	Filter		native	2		2			
		low pass	native	2			2		
		cutoff frequency	native	2			2		
		resonance	native	2			2		
		slope	native	2			2		
		high pass	native	2				2	
		band pass	native	2				2	
		notch	native	2					2
		formant	non-native	0					0
Mixer section			implementable	1	1				
	Mixer		implementable	1		1			
		oscillator level	native	2			2		
		noise level	native	2			2		
		audio in level	native	2					2
		sub oscillator level	native	2					2
		filter balance	implementable	1					1
Amplifier section			implementable	1	1				
	Amplifier		implementable	1		1			
		level	native	2			2		

Section	Element	Feature	Impl. Level	Point	Top level	Sections	Required features	Recommended features	Optional features
Modulation section			implementable	1	1				
	LFO		native	2		2			
		frequency	native	2			2		
		triangle	native	2			2		
		pulse/square	native	2			2		
		amount	implementable	1			1		
		sine	native	2				2	
		sawtooth	native	2				2	
		sample'n'hold	non-native	0				0	
		tempo sync	implementable	1				1	
		reverse sawtooth	implementable	1					1
		random	non-native	0					0
		delay	implementable	1					1
		phase offset	non-native	0					0
	Env. Generator		implementable	1		1			
		attack	implementable	1			1		
		decay	implementable	1			1		
		sustain	implementable	1			1		
		release	implementable	1			1		
		amount	implementable	1				1	
		linear/exp switch	implementable	1					1
	Other mod sources		-						
		Hard sync	non-native	0				0	
		Ring modulation	implementable	1					1
		Noise	non-native	0					0
	Mod destinations		-						
		Oscillator pitch	native	2			2		
		Filter cutoff frequency	native	2			2		
		Amplifier level	native	2			2		
		Pulse width	non-native	0				0	
		Panning	non-native	0				0	
		Oscillator mix	implementable	1					1
		Noise level	native	2					2
		Filter resonance	native	2					2
		LFO frequency	native	2					2
		LFO amount	implementable	1					1
		env gen amount	implementable	0					0
		env gen attack rate	implementable	0					0
		env gen decay rate	implementable	0					0
		env gen release rate	implementable	0					0
	Result			105	5	13	38	15	35
	Total			164	10	18	46	28	62