TARTU UNIVERSITY VILJANDI CULTURE ACADEMY
AND BALTIC FILM AND MEDIA SCHOOL OF TALLINN
UNIVERSITY
Sound Engineering Arts

Thomas Lieb

ELECTRONIC DANCE MUSIC PRODUCTION HELPER

A TOOL FOR GENERATING RISERS

Master's Thesis Project

Supervisor: Janar Paeglis, M.A. Electronic Music

Allowed for defence .....
(supervisor's signature)

Viljandi 2014

This research was supported by European Social Fund's Doctoral Studies and Internationalisation Programme DoRa

Abstract

In this thesis an audio plug-in was developed. The plug-in is a tool for creating electronic dance music risers. Risers are elements in the production of songs in that genre and connect different parts by transitions. The most common example is a filter sweep applied to noise. A plug-in, which supports the major architectures, was previously unavailable. The producers had to build risers manually or use existing audio samples from sound libraries. The goal was to build a tool where the producer can sonically judge and adjust the riser at any time in the production process.

Risers are built with many small elements based on signal processing algorithms. This software is derived from an object oriented strategy that matches the structure of bundled small objects. The algorithms are based on theories that are commonly used to develop electronic instruments that generate sound. The theories are shortly explained in the thesis and later used during the development of the final product. Risers are sound objects that transform over time: transformation appears in automated control changes of the components. The solution provides an easy way to the manipulation of parameters over time. The result of the thesis is a product that can be used in either professional or home studios.

Acknowledgements

The project was done at AIR Music Technology GmbH (AIR) in Bremen, Germany. I would like to thank the company for supporting me during the thesis. They provided me with a friendly platform of knowledge and inspiration. Special thanks go to Mario Reinsch. Before the thesis I did a traineeship at AIR and during that time he led me to the idea of building this product. He also helped me a lot during the implementation phase and was always an inspiring mentor. Jan Kraneis also earns a note here. He was compassionate and supportive of my work—especially at hard times during the project. Based on his trust I was not only able to do the thesis in cooperation with AIR, but also I got an opportunity to work for the company afterwards. Mark Ovenden is the sound designer at AIR and often helped me to judge the artistic needs for good sounding risers.

Many thanks go to the Viljandi Culture Academy. The Master's program in Sound Engineering Arts was great a joy and opened doors to my future career. Special thanks to Janar Paeglis and Christoph Schulz who supported my traineeship and thesis abroad and the cooperation with the company. Also, I would like to thank Kadri Steinbach and Djuddah Leijen who helped me with academic writing issues.

Appreciation also goes to Stanislav Barvinski. He not only became a good friend during my studies, but also was a good person to talk to about my thesis. We did seemingly endless Skype sessions about our projects and helped each other mentally and productively.

Last but not least thanks to Maarja Raudvee. She greatly helped to organize my studies and always had an open ear for my problems I encountered as a foreign student.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API	application programming interface (pp. 24, 25)
BPM	beats per minute (pp. 9, 22, 32)
CD	Compact Disc (p. 10)
C++	C plus plus (pp. v, 9, 22, 23, 25–32)
DAW	digital audio workstation (pp. 2, 8, 9, 20–22, 24, 32, 35, 36)
EDM	electronic dance music (pp. i, 1–3, 5, 7, 8, 14, 22–24)
GUI	graphical user interface (pp. 22, 24, 33–35)
LFO	low frequency oscillator (pp. 9, 31, 32, 34)
Max	Max/MSP (pp. 22, 23, 25, 26, 34, 35)
MIDI	musical instrument digital interface (pp. v, 8, 24, 26, 27, 33, 35)
VST	Virtual Studio Technology (pp. 24, 25, 35)

Introduction

> The term "electronic dance music" encompasses a broad range of music produced during the last two decades, including genres such as techno, house, drum 'n' bass, and trance. [...] One of the most distinctive characteristics of electronic dance music is the way in which it is produced—namely, through the use of electronic technologies such as synthesizers, drum machines, sequencers, and samplers. (Butler 2006, pp. 32-33)

> Built up from layers, a common strategy in the music's unraveling is to introduce, or drop out, layers one or more parts at a time. At certain sections, the bulk of the material is removed in favor of a breakdown (a moment particularly prevalent in trance or some drum and bass) only to allow the increasing tension of building up once more. (Collins, Schedel, & Wilson 2013, p. 114)

This thesis is about risers and their use in electronic dance music (EDM). Risers are relevant musical elements for creating tension and thus excitement. They fill the gap in breakdowns and connect different parts. This thesis is a thesis project, so the time was spent on creating a product with artistic applicability. The resulting product is a riser plug-in, which is a software tool for creating risers. Today the procedure for creating risers is a series of manual steps performed by EDM producers and considerable time is required to translate their ideas into sound. My goal was to simplify this process and give them a tool that enhances their productivity and creativity.

"The origins of electronic music lie in the creative imagination. The technologies that are used to make electronic music are a realisation of the human urge to originate, record and manipulate sound." (Hugill 2008, p. 7)

At the moment there are several solutions for creating risers available and these will be discussed later in this thesis. Unfortunately they are not satisfying in usability and artistic freedom: they don't fit to the common workflow of a EDM producer. There is no dedicated and easy way to quickly build risers and preserve the creativity of the user. Neither exists a virtual instrument that can be hosted in modern digital audio workstations (DAWs). So there was still a need to build a tool that is different. I wanted to supply the EDM producers with a helper plug-in where they can create risers more effectively.

Exploring possible risers led to a series of different ideas, which were implemented in the final plug-in. I defined the ideas more and more precisely with the help of literature. Some ideas had already been explored historically or as a technical or physical phenomenon. I could use that knowledge to build an interesting and creative plug-in.

The thesis was written in the Sound Engineering Arts program, but it is rather technical. Developing an audio plug-in is beyond the usual scope of an audio engineer. He will most of the time stick to the operation of plug-ins. For this reason I decided to structure my thesis accordingly. In chapter 1, I will discuss theoretical background supporting my decision while building the solution or explaining the technological or physical background needed to understand the process of creating the riser software. In chapter 2, you will find information about the process of building the plug-in and the details of its implementation. In most sections I will refer back to the theory to give a better understanding about my progression.

1. Theoretical background

1.1. Problem description

In every modern EDM production we can listen to transitions. "Transitions are primarily used in dance music to merge two sections together, build tension within a section, or provide subtle changes to a monotonous sound." (David 2012, p. 237) The transitions can be tonal or atonal—they are called risers. Risers usually build up musical tension followed by a drop. Uplifting and downshifting are the corresponding verbs used in the communities. When uplifting a track in musical terms we use a *crescendo*, upward *glissando* or *arpeggio* and sometimes even *accelerando* to create the desired effect; when downshifting a track we use the corresponding antonyms *diminuendo* (*decrescendo*), downward *glissando* or *arpeggio* and *ritardando*. Atonal sweeps cannot really be expressed with the classical music terminology. But these sweeps are popular in electronic music and created by filtering different noises and automating the center frequency over time.

Sometimes creating these effects can be time-consuming and unintuitive. Thus most producers stick to the basics—creating a white noise sample and filtering it with a lowpass. Automating the cut-off frequency from low to high creates a basic uplifting sweep. This sounds like an easy task, but it is a complex theoretical procedure when considering the filter characteristics with variable slopes and resonances. Infinite sweeps of this kind can be created. Producers handle this simple task easily and are creative when setting up the automation for the cut-off, resonance and slope of the filter.

What if you want to create a more complex transition? Then, the process is much more difficult. There are many sample libraries, which provide a collection of various sweeps. The user can manipulate the samples to fit his production by time stretching or pitch

shifting. Here, not only sound quality, but also creativity is lost. The main problem is though that the riser will probably sound good on its own, but not fit into the mix. Producers need a tool to manipulate the sound easily and consider changes in every song production phase. It is necessary to go back and adjust the overall tonality, for example, when new sound layers are introduced. Sometimes there is a need to fit the riser harmonically to the song. Some riser samples have a certain harmony built-in, but it is almost impossible to pitch shift polyphonic sounds easily—especially not if they are masked by glissandos or noises. To offer also this option the risers should have a flexible harmonic element: a chord generator.

Control is the main problem. We need control over the sound of the riser to make our life easier and produce better sound—in the end better music. With a riser plug-in the sound can be merged into a project: we can adjust the sound to our needs without using a big set of tools. The parameters are accessible all the time during the production, so we can make adjustments when new ideas arise. Through a simple interface the workflow is optimized for speed. Evolution of parameters over time can be visually represented, so we can see and imagine what we will hear when playing the song.

1.2. Theory

1.2.1. The different views on musical values

There are different views on musical values. For example, the musicologist or composer defines different volumes with piano to forte; the sound engineer measures volume in decibels; the audio developer uses a normalized floating point number from 0.0-1.0. These values have to be mapped accordingly when developing audio plug-ins. The audio developer has to know all the stages, whereas the musicologist does not have to care about a normalized floating point. The internal processing of a plug-in is most of the time also unknown to sound engineers—tuning the parameters is important. A finished audio plug-in should present a decibel or piano to forte mapping depending on the target user of the plug-in.

In this section I will try to make the transition from the musicologist's to the audio developer's view. In chapter 2, I will use the audio developer's vocabulary to keep the writing precise and short.

1.2.2. Musical tension

To create excitement, composers use the principle of tension and release. "Purposefully denying or delaying listeners what they expect or hope for creates tension." (Cope 2011, p. 64) Tension can be built up in different ways: variations in dynamic level or harmonic, melodic and rhythmical gestures anticipate a relief and calm. "Balancing excitement and quiescence, action and relaxation" (Liebman 1991, p. 13) plays with expectation and surprise of the listener.

> [Equalization (EQ)] can be used as an effect to create sonic tension in builds and breakdowns. Strap a high-pass filter across the bass and slowly raise the frequency during the breakdown so the bass gradually loses its impact in the mix. Then, as the drop comes, bypass the EQ so that the full bass returns to cause maximum dancefloor mayhem. Experiment with resonance too, for added scream-factor, but avoid big boosts when sweeping the lower frequencies as you'll quickly end up with unpleasant volume spikes. (Adamo 2009, p. 36)

A riser is a form of musical tension. In electronic dance music this form of tension is used in almost every track. This is different than in classical music, where the tension lies in the music itself—created by harmonic patterns—but with exceptions as seen in subsection 1.2.3. Controlling the pitch and frequency spectrum of one or more instruments creates tension. Tension can also be described as different levels of energy in a song. With build-ups and releases of this tension the song is contoured. In EDM the tension is usually built by an effect: producers use, for example, a filter sweep.

1.2.3. How Beethoven used uplifters and downshifters

In the classical era the composers created "special effects". Hoffer (2011, p. 125) writes about the "gradual *crescendo* and *decrescendo*" developed during that period. These "changes in dynamic level were considered quite dramatic at the time. They were an important contribution of that period to the development of music." These effects last until today and have ancestrally influenced transitional elements of EDM.

The Mannheim school

In the 18th century, an orchestra was founded called the Mannheim *Hofkapelle* (court orchestra). Mannheim was one of the most flourishing seats of the arts and sciences especially in the second half of the century. The Bohemian musician Johann Stamitz was the founder of the Mannheim school; with him as the concertmaster the Mannheim orchestra became renowned all over Europe for its virtuosity and dynamic performances. Multiple composers influenced each other during that time and developed the Mannheim *Manieren* (figures); Table 1.1 shows a list of the different figures.

Figure	Description
Mannheim rocket	a swift ascending melody using the arpeggiated triad, apparently inspired by a Roman candle firework
Mannheim *crescendo*	a gradual build-up in volume by the entire orchestra, often followed by an abrupt *piano* (quiet) or a long pause
Mannheim (steam)roller	a gradual *crescendo* through a rising melody over an *ostinato* (repeating) bass line accompanied by *tremolando* figures
Mannheim sigh	a sequential stepwise-resolving *appoggiatura* figuration with an emphasis and elongation on the first note in a pair of slurred notes
Mannheim hammerstroke	a series of three initial tonic chords struck in sequence loudly
coup d'archet (striking of the bow)	a definitive chordal or unison statement at the beginning of a fast movement, sometimes followed thereafter by a scale upward or downward
the grand pause	the orchestra suddenly stops playing only to resume with great verve; this creates suspense and forward momentum in the music
Mannheim bird	imitating bird song to brighten solo passages

Table 1.1.: Mannheim *Manieren*: trademarks of the Mannheim school

The imitation of something outside the musical work is called mimesis (Demers 2010, p. 171). In the 18th century the emulation of natural phenomena was rather rough; with synthesis and sampling the emulation can be achieved more precisely. When you

look at the Mannheim figures you will notice the parallels that exist with EDM risers. These figures were the risers of the 18th century. The ideas developed by the Mannheim orchestra provide working examples for risers and can therefore be a good influence when developing an audio plug-in.

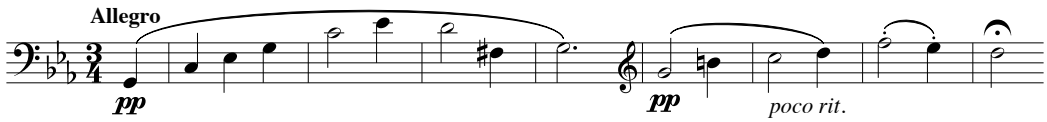

Figure 1.1.: Rocket theme played by cello and bass, Ludwig v. Beethoven, Symphony No. 5, 3rd movement, measures 1–9 (Hoffer 2011, p. 166)

In Figure 1.1, you see an example for a Mannheim rocket in Beethoven's Symphony No. 5. The progression of notes is rising in pitch almost constantly and ending in a *fermata*. This can be seen as a metaphoric figure or auditory simulation of a rocket. Beethoven also uses a *poco ritardando* in measure 7, so we also have a temporal interpretation—this could be due to the excitement of a human spectator in the last moment before the rocket starts, who psychologically perceives time slower in this special situation.

1.2.4. Musical terms converted to electronic music

Instrumentalists, conductors, composers and musicologist describe music with musical terms. They have a vocabulary to express playing style and a notation system to reproduce a musical piece. Sound engineers are also interested in, for example, dynamics and correct pitch, but they often use a different vocabulary. Moreover, they use the terms sound and frequency, which originate from the science of physics.

Sound is created by compression and rarefaction of air molecules: this results in a wave. The wave can be stored electronically and afterwards reproduced with an amplifier and speakers. Sound in its waveform representation entirely stores pitch, intensity and timbre. It can be seen as a combination of pure waves, which are called sine waves. Sine waves are waves with exactly one pitch or better—one frequency. Frequency is defined as oscillations of the wave per second. Sound engineers manipulate the stored sound electronically and digitally to improve the recording and make the listening experience more enjoyable.

These principles apply to the analog domain as to the digital domain. When storing sound digitally, the wave is not saved entirely. We only store points in time periodically.

Characteristic	Parameter	Perceptual Sensation
Frequency	Pitch	High $\longleftrightarrow$ Low
Amplitude	Intensity	Forte $\longleftrightarrow$ Piano
Waveform	Timbre	Sound color

Table 1.2.: Terminological correspondences between sound characteristics, musical parameters and perceived sonority (Cipriani & Giri 2013, p. 8)

How many points (samples) will be preserved is defined by the sampling rate. The amplitude—how big the wave is—also has a resolution, so we have to round the incoming values up or down. So digital signal processing is done sample by sample, where the value of the sample has a defined resolution. After converting the sound back to the analog domain, we can amplify to a degree of volume.

EDM is produced in DAWs. Today most of the time producers don't record any real instruments—instead, they use virtual instruments, samples and synthesizers. A musical *crescendo* is a steady increase of volume over time. In DAWs automation is the solution for manipulating a parameter over time. So a *crescendo* would be simulated by automating the MIDI volume (CC 7): we specify a low value at the beginning and a high value at the end.

$$x_n = x_{n-1}(g_{n-1} + \Delta g) \tag{1.1}$$

Equation 1.1 shows the mathematical formula of a steady rise in volume where x is a sample and g is a normalized gain value. The samples are processed sequentially and stored in a buffer. So $_n$ is the index of the current sample and past samples can be accessed by a negative index: $_{n-1}$ is the previous sample before the current. The output volume is calculated by multiplying the sample with the gain value g. For the current value it is the sum of the gain value of the previous sample (g_{n-1}) and the delta gain value (Δg). The delta gain value indicates how fast the volume will rise—the higher the value, the faster the rise.

Listing 1.1 shows an implementation example of Equation 1.1. The variables `in` and `out` are pointers to the actual sample values. The asterisk (star) on the left side of the sample variable resolves the pointer to its value. The `++` on the right side of the variable is an indicator that the pointer is incremented to point to the next sample, after the current value has been resolved. `currentGain` is increased by `deltaGain`, while iterating

```
1 while (sampleframes--) {
2   currentGain += deltaGain;
3   *out++ = *in++ * (currentGain);
4 }
```

Listing 1.1: Example for a crescendo (C++)

through the audio vector with the size of `sampleframes`. The new value of `currentGain` is multiplied by the input sample (`*in++`) and stored to the output sample (`*out++`).

```
1   *out++ = *in++ * (automationGain[currentTick]);
```

Listing 1.2: Lookup of automation gain values (C++)

In a DAW an automation curve is stored in memory with discrete values at a predefined temporal resolution depending on the projects beats per minute (BPM) and the hosts (DAWs) pulses per quarter. The gain values will not be calculated as in Listing 1.1, line 2—instead a look-up from an array is used (Listing 1.2). The `automationGain` array contains the gain values. The values are read from the array with the index (song position pointer) `currentTick` and multiplied with the sample. With the `currentTick` value we can also calculate the current beat or the elapsed seconds as seen in Listing 1.3.

```
1   currentBeat = currentTick / pulsesPerQuarter;
2   elapsedSeconds = currentBeat / (beatsPerMinute * 60);
```

Listing 1.3: Calculation of current beat and elapsed seconds (C++)

When building the riser plug-in this automation needs to be implemented. The automation is basic: we have a start value and an end value. This generates a straight line. But we don't want to limit the user to only that. The solution is an exponential or logarithmic curve. If we do a crescendo we can, for example, set the curve so that the volume is very low in the first half of the riser and and then increases rapidly to the loudest point in the second half. The possibility of combining multiple sonic elements is an important feature to preserve artistic freedom.

Moreover, we can add modulation to the curve with a low frequency oscillator (LFO) and shape the parameter. With this step the result is already rather complex.

1.2.5. Sound waves emulated digitally

To understand how a riser is created, we need to have a look at the elements. These elements are based on physics. We have to understand how a sound wave is defined. As already discussed in subsection 1.2.4, a sound wave is generated naturally by compression and rarefaction of air molecules. The air molecules act like a spring thus generating a wave. Usually this phenomenon is plotted in a graph by drawing amplitude on the y-axis and time on the x-axis. In Figure 1.2, we see the waveform of a sine wave and how the graph corresponds to the natural phenomenon.

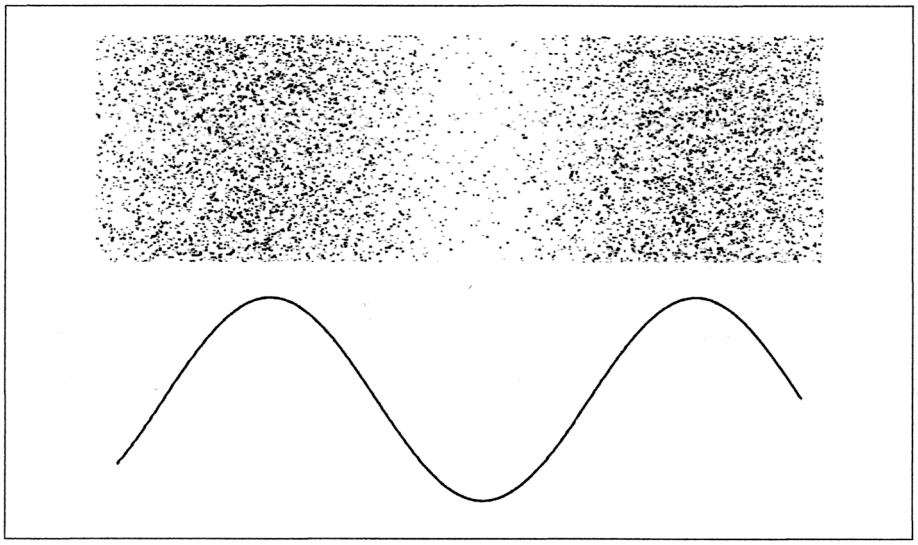

Figure 1.2.: Compression and rarefaction of air molecules (Cipriani & Giri 2013, p. 9)

Sine waves can be generated electronically and are heavily used in electronic music. A sine wave is idealized: it has no harmonics at all—which does not occur in nature. But we can use it to describe the principle theory. Additive synthesis is an example. Here multiple sine waves are generated and added together. If the frequencies of the sine waves have integer ratios to each other, we can add the missing harmonics to a base. The result is a sound that is much more "natural and believable".

If we want to process sound with computers we need to store it digitally. Sound is sampled as seen in Figure 1.3 where values are recorded to a memory at a constant rate. This rate is named the sample rate and is, for example, 44100 Hz on a Compact Disc (CD). So 44100 samples are stored per second. The possible amplitude values represented on the y-axis in Figure 1.3 have to be finite, too. On a CD there are $2^{16} = 65536$ possible values of amplitude. We talk about a bit-depth or resolution

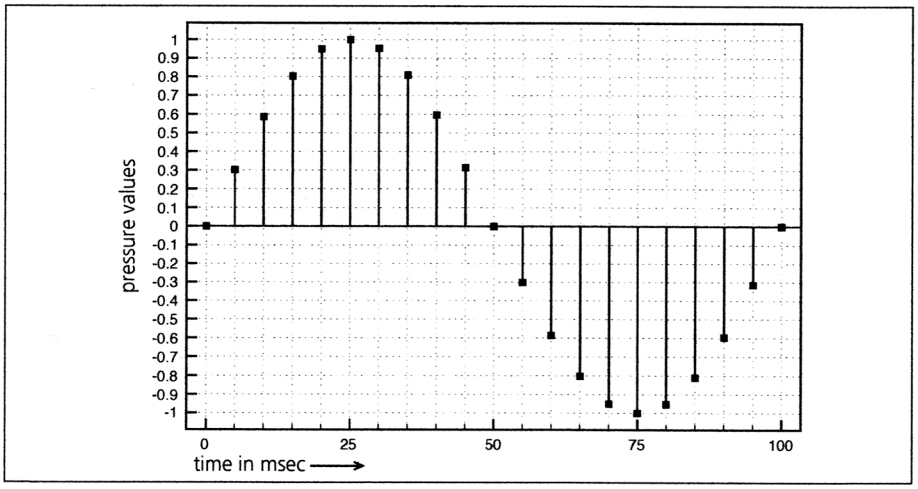

Figure 1.3.: Digital representation of a waveform (Cipriani & Giri 2013, p. 13)

of 16 bits. If we convert an analog signal to a digital signal this process is called quantization.

The computer stores values with a certain data type. If we save a sampled value we usually use a floating point number, which is simply a data type that can hold real numbers. We want to have a maximum amplitude of 1.0—the amplitude is normalized.

Sinusoid waves can be easily generated in computers. Sound generators of this type are called oscillators. The sine oscillator builds the waveform by calculating the mathematical sine function. This function is natively available in almost every modern programming language. To save processing power usually we use wavetables. Wavetables store a single period of a waveform. When the oscillator streams the audio to the sound card output, it just has to read from this memory. This is much more efficient than calculating the values with the sine function for every sample. When the end of the buffer is reached, we start again from the beginning. We can skip samples in the buffer: for example, skipping every other one and preserving the original sample rate will result in an oscillation with the frequency doubled. In Figure 1.4, we see a glissando: when the frequency rises the oscillation is faster.

With the sine wave oscillator in our hands and the knowledge that every wave can be described with layered sine waves, we can build all other "classic" waveforms. The waveforms are illustrated in Figure 1.5; the highlighted area is exactly one period. To construct other waveforms we have to sum multiple sine waves together. The partials—here harmonics generated by multiplying the base wave with integer values—need to have

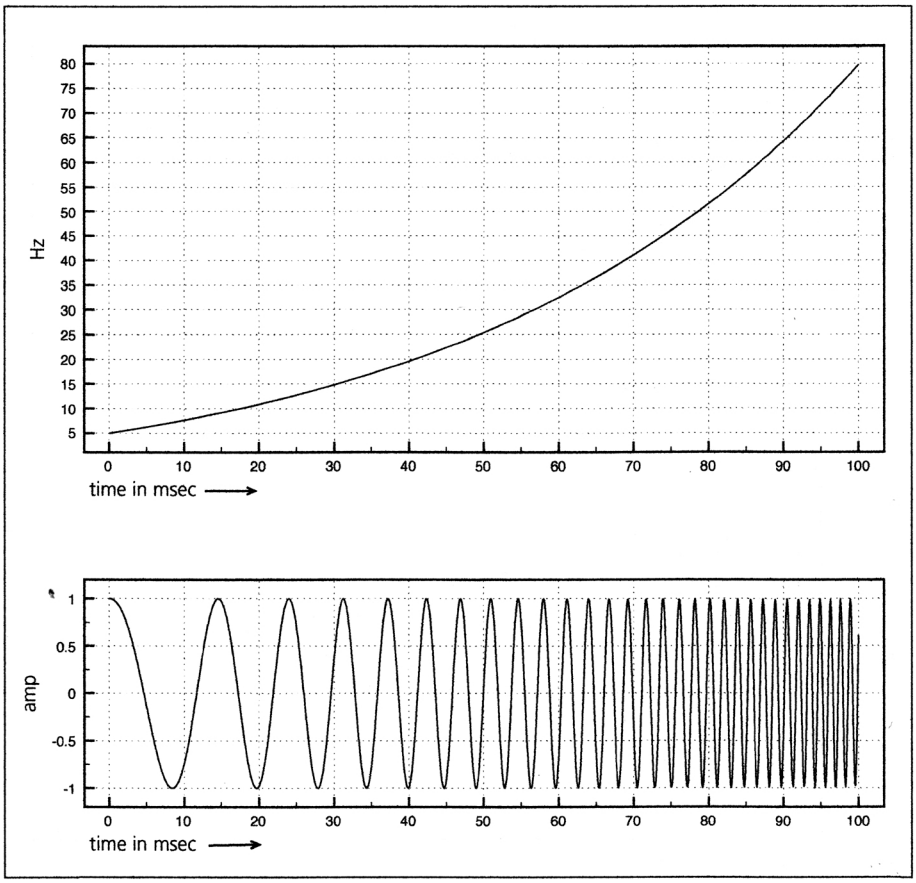

Figure 1.4.: An ascending glissando (Cipriani & Giri 2013, p. 30)

predetermined amplitudes. In Figure 1.6, we see how these amplitudes have to match to create a square, a triangle and a sawtooth wave. In the first row the mathematically perfect graph is displayed. In the middle we see the band-limited approximation and on the right there is a line spectrum with the component strength of every sinusoid. Band-limitation is the term for limiting the amount of sinusoids which are used for construction. It is impossible to add an infinite amount of waveforms together—this would give us the ideal waveform though. In Figure 1.6, the top-right spectrum tells us that a square wave is generated by summing only odd harmonics. The more harmonics we add the steeper the slope of the square wave is. This can be seen in the top-center diagram. If we could add infinite harmonics the slope would have a straight alignment to the vertical axis.

$$squ(x) = sin(x) + \frac{1}{3}sin(3x) + \frac{1}{5}sin(5x) + \frac{1}{7}sin(7x) + \dots \tag{1.2}$$

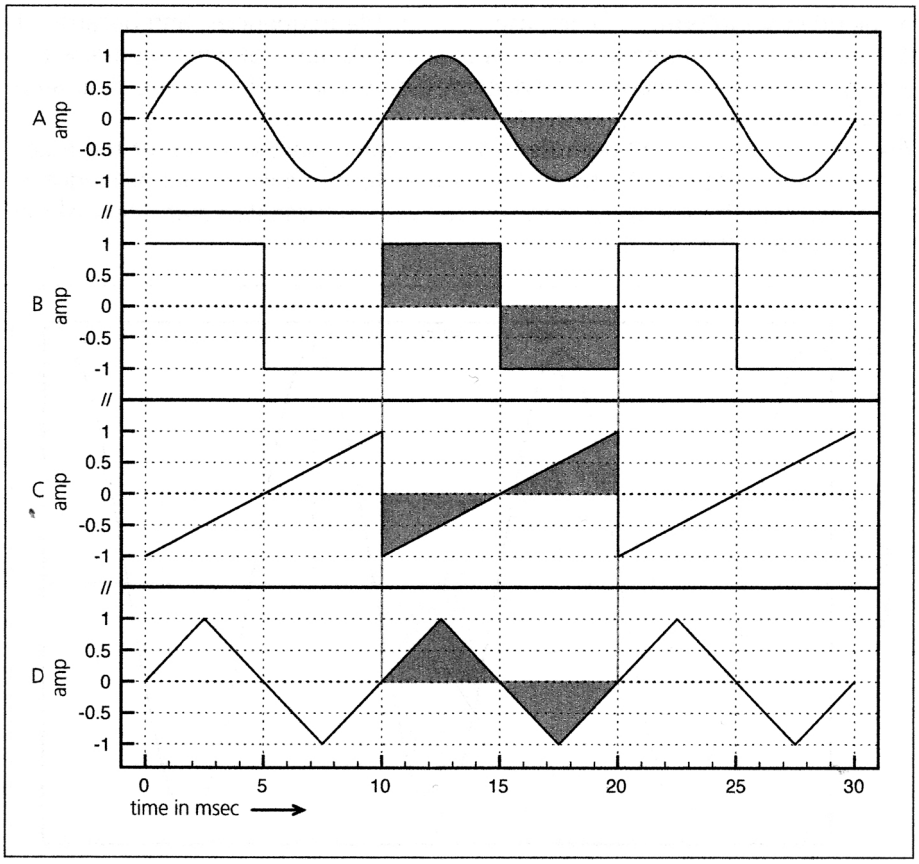

Figure 1.5.: The four "classic" waveforms (Cipriani & Giri 2013, p. 20)

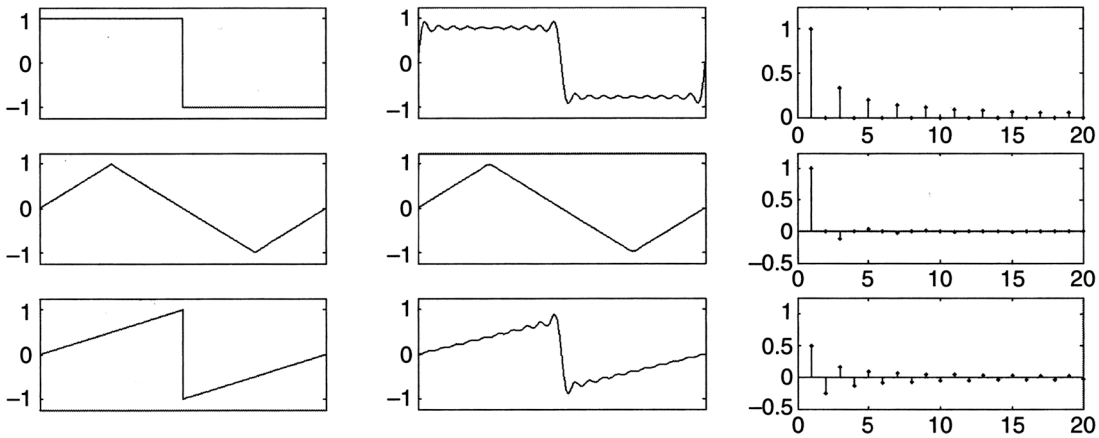

Figure 1.6.: Ideal plot, composite plot and spectra of square, triangle and sawtooth waveforms (Collins 2010, p. 165)

$$tri(x) = sin(x) - \frac{1}{(3^2)}sin(3x) + \frac{1}{(5^2)}sin(5x) - \frac{1}{(7^2)}sin(7x) + \dots \tag{1.3}$$

$$saw(x) = sin(x) - \frac{1}{2}sin(2x) + \frac{1}{3}sin(3x) - \frac{1}{4}sin(4x) + \dots \tag{1.4}$$

Equation 1.2 is the formula defining how a square wave is constructed. The value in front of the sinūs indicates the amplitude. The factor inside the sinus functions is the integer number of the partial. When we look at Equation 1.3 and Equation 1.4, which are the formulas to construct a triangle and sawtooth wave, we notice the alternating signs. Also, for square and triangle waves we discover only odd partials. This is also reflected in the spectra on the right side in Figure 1.6.

1.2.6. Noise

> Periodic sounds, such as the pitched sound made by musical instruments, or the vowel sound of the human voice, are perceived as being equipped with a definite pitch. [...] Non-periodic sounds, such as the sounds made by musical instruments of indeterminate pitch like cymbals gongs, and triangles, or consonants produced by the human voice, are not perceived to have definite pitch. The closest we can come to pitch identification for such sounds is to perceive a frequency band in which the density of components is thick enough to bestow a relevant amplitude. (Cipriani & Giri 2013, p. 208)

Noise is a non-periodic and non-harmonic signal that contains ideally every frequency. Random oscillation produces noise. If the spectrum shows a uniform distribution (flat at 0 dB) of all audible frequencies, the noise is called white noise. It is called white noise because it is similar to combining all colors of light in the human visual range: the result is white light. Figure 1.7 displays computer generated white noise. All frequencies have approximately the same amplitude.

There are other types of noises: they have a different spectral distribution. Some theoretically based noises have other color names. For example, in Figure 1.8 we see the spectrum of pink noise. Here the amplitude of higher frequency drops. The attenuation is 3 dB per octave: the energy is distributed equally in every octave.

In EDM white noise is used for risers. The “classical” riser is a filter sweep that produces a “swoosh” effect. I will call this the basic riser. A noise sweep is an emulation of a natural phenomenon. When a sound source approaches us, our attention is drawn to

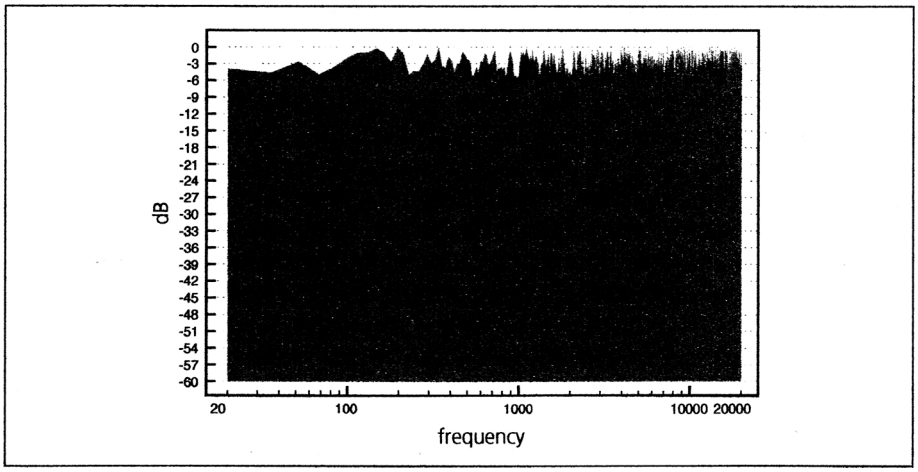

Figure 1.7.: The spectrum of white noise (Cipriani & Giri 2013, p. 300)

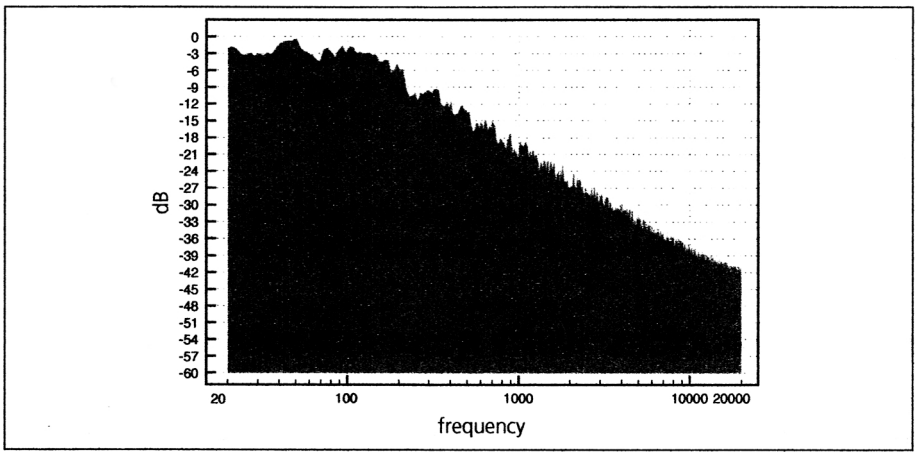

Figure 1.8.: The spectrum of pink noise (Cipriani & Giri 2013, p. 300)

it. This is based on our instinct that makes us aware of enemies. Musical instruments have an envelope that gradually rises in volume, too. So a white noise sweep can also be thought of as an emulation of a real instrument. This would be a big instrument or any object that produces sound. The attack time is the time that is needed to bring the instrument to maximum output: the bigger the object, the higher is the attack time. Big objects draw attention to themselves and if we can simulate this by a riser it can get us immersed into a song.

1.2.7. Filters

Filters are systems that modify the signal by subtracting parts of it unless a filter goes into self-oscillation. The cutoff frequency defines the working range of the filter. It is the point where filtering starts on lowpass or highpass filters or the center frequency on notch and bandpass filters.[1]

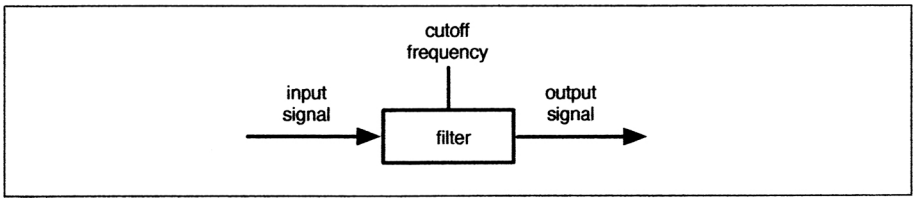

Figure 1.9.: A diagrammatic filter (Cipriani & Giri 2013, p. 304)

Filters are elements of subtractive synthesis. We eliminate parts of the sound. For example, if we use a lowpass filter we attenuate the high frequencies. Figure 1.10 shows a lowpass filter with an attenuation of 48 dB per octave. This is quite a steep slope. The slope of the filter is determined by the order of the filter. The order is proportional to the complexity of the filter and determines the amount of samples needed for the calculation.

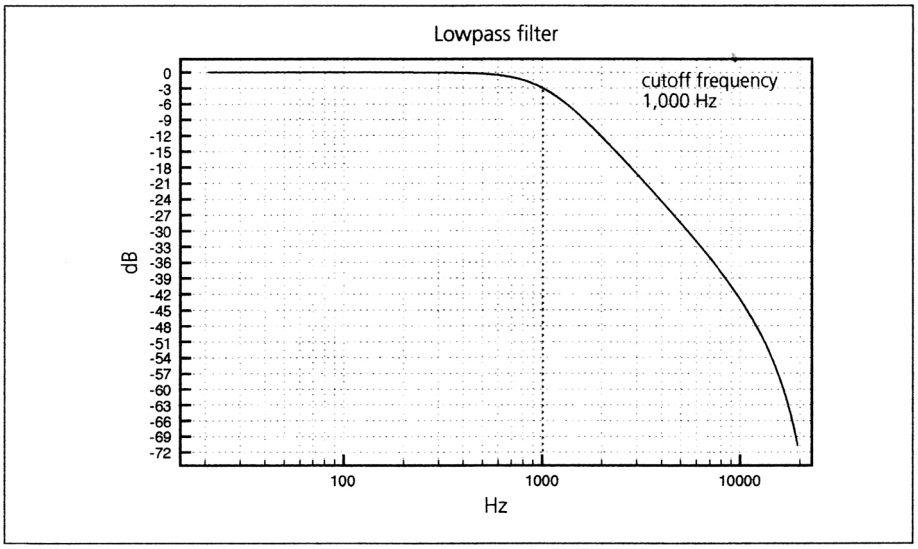

Figure 1.10.: An attenuation curve for a lowpass filter (Cipriani & Giri 2013, p. 305)

[1] Lowpass filters are sometimes called highshelf and similarly are highpass filters called lowshelf filters. Notch filters are also often called bandstop or bandreject filters.

In Figure 1.11, we see how a lowpass affects the partials of a sound. It not only attenuates higher frequencies, but also changes their starting phase. Higher fundamentals are shifted further away from the initial phase. The fundamental frequency (f_1) though is unaffected in amplitude and phase. We can see that the resulting waveform is somehow "smoothed" out. If we listen to a lowpass filtered sound we can hear the smoothness. In comparison to the unfiltered sound the "roughness" is gone –which originated simply from the high frequencies.

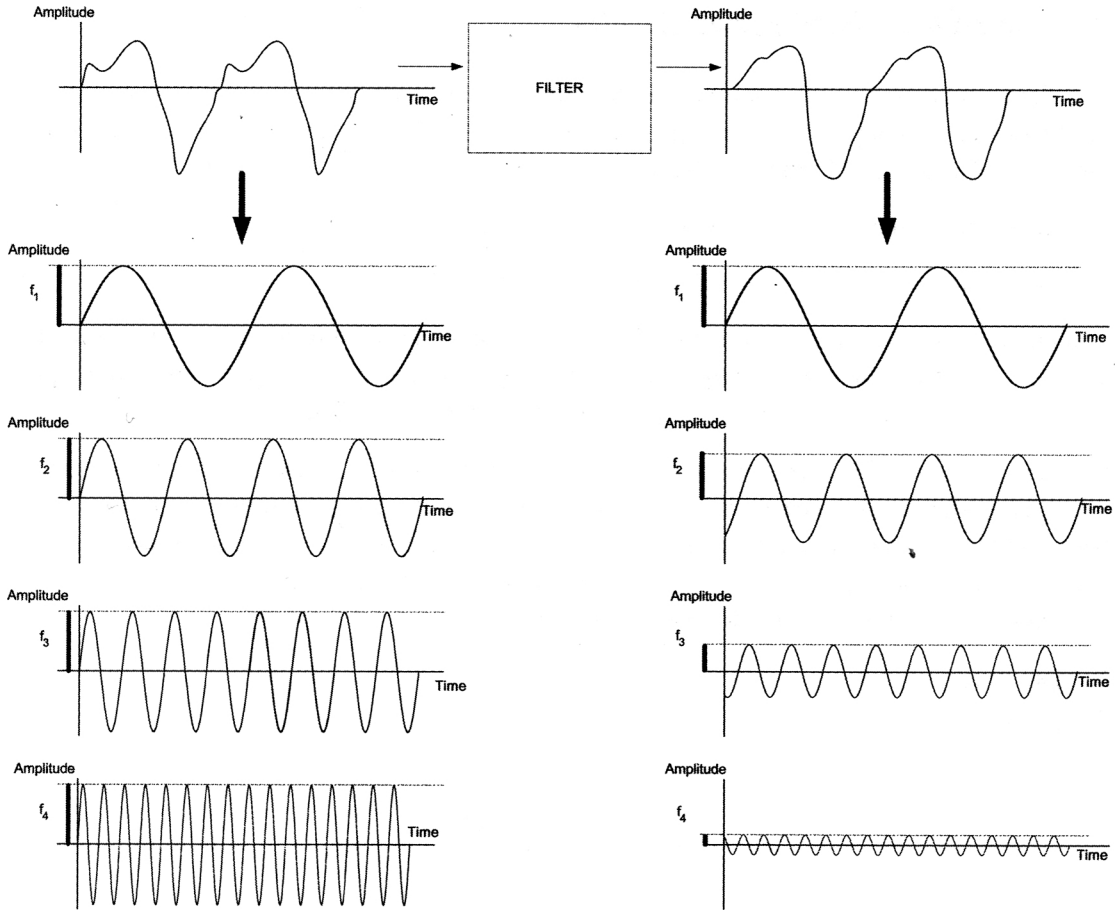

Figure 1.11.: A complex waveform is filtered into a smoothed output. The input and output are decomposed into their Fourier-series components. (Pirkle 2012, p. 73)

So how can we calculate a filter with the computer? In subsection 1.2.5, we saw how sound is stored digitally. The great advantage of digitally stored sound is that we can access values from the past. It is usually a stream written into a buffer with a defined size that wraps around when the end of the buffer is reached. Wrapping is just starting from the beginning. The defined size of the buffer defines how far we can go back. For

simple filters we only need the current and the previous sample for calculation.

$$y(n) = a_0 x(n) + a_1 x(n-1) \tag{1.5}$$

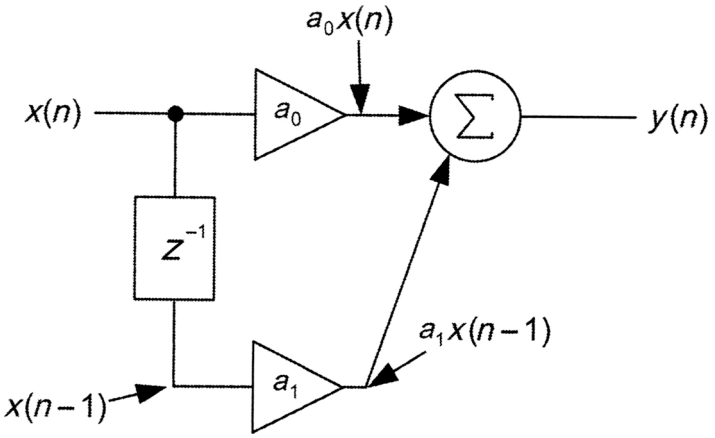

Figure 1.12.: The first-order feed-forward filter block diagram (Pirkle 2012, p. 74)

$$y(n) = a_0 x(n) - b_1 y(n-1) \tag{1.6}$$

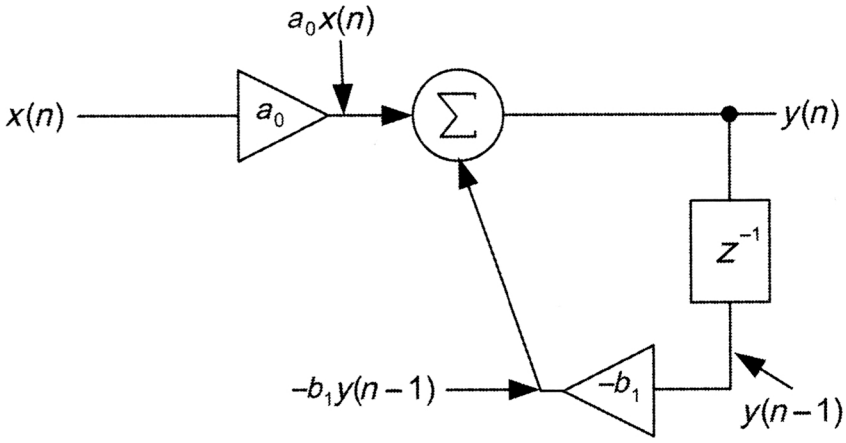

Figure 1.13.: The first-order feed-back filter block diagram (Pirkle 2012, p. 88)

In digital signal processing there are two filter strategies mostly used: filters that work with the delayed input buffer called feedforward filters; filters that work with the delayed output buffer called feedback filters. In Equation 1.5 and Equation 1.6, we see the formulas for both strategies. n is the index of the sample, $x(n)$ the input and $y(n)$ the output buffer. a_0, a_1 and b_1 are the filter coefficients. We can control the filter behavior

with the coefficients, so that we either get a lowpass filter, a highpass filter or anything in between them. Figure 1.12 and Figure 1.13 visualize the formulas in block diagrams. z^{-1} indicates a delay of one sample.

$$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) - b_1y(n-1) - b_2y(n-2) \tag{1.7}$$

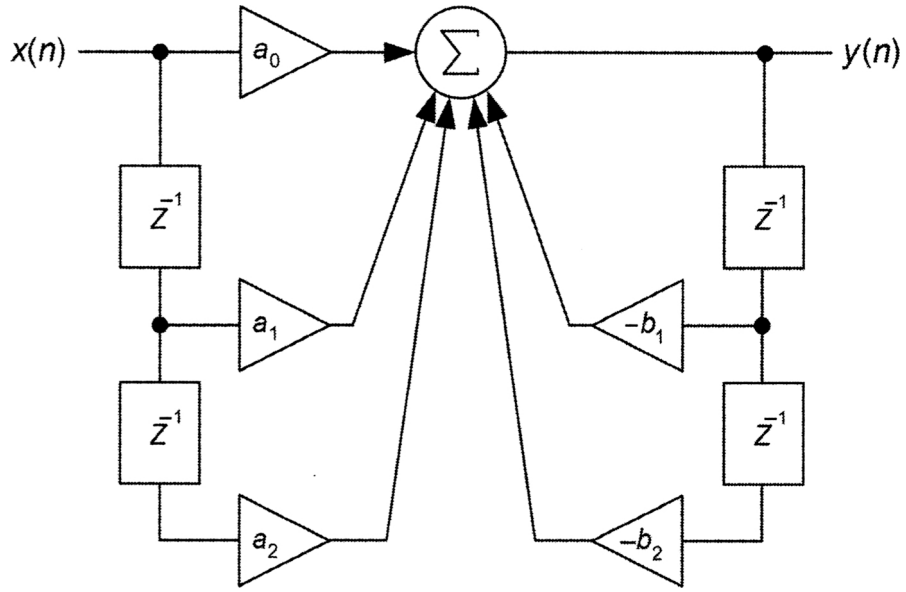

Figure 1.14.: Generic bi-quad structure (Pirkle 2012, p. 182)

These two strategies can be combined into a more complex but flexible approach: the bi-quad filter. This filter is show in Figure 1.14. Equation 1.7 shows its formula. By setting the coefficients right we can emulate analog filters. We also can produce bandpass, bandstop, different filter configurations like *Butterworth* or *Linkwitz-Riley* and allpasses. Allpasses are filters that only change the phase but leave the frequency response untouched.

1.2.8. Spatial sound and sonic dimension

"Spatialization is the art of placing sounds in space, perhaps to evoke and reproduce the original acoustic at the site of a recording, or to perform within a given space by controlling the localization of sounds." (Collins 2010, p. 55)

"A spatially *extended* type of music could offer new musical dimensions. Certainly, an aesthetic input to the design of the compositional environment will probably produce

something that cannot be logically anticipated." (Lennox 2011, p. 272)

Placing the sound in the virtual space created by a stereo listening environment can be done horizontally and vertically. The horizontal placement is rather easy: you can move sound sources left or right using the panorama value in a DAW. Vertical placement is a bit more complicated. Sound appearing further away from us is more reverberant. So the easiest, but unrealistic, option is to put a reverb effect on the desired track. Also, a sound source moving away from us loses high frequencies: molecules need more energy for distributing higher frequencies, so their impulses are lost quicker on they way to our ear. We can play with these physical facts and simulate moving sound sources for an immersive effect.

Fake stereo effects are most of the time constructed with two signals being slightly out of phase. If we have multiple oscillators on the left and the right signal, we can set their phases randomly. The effect is a widened stereo perception. Noises are already constructed with random numbers. In this case we only have to use separated instances of noise generators for each stereo channel.

1.2.9. Endless riser with Shepard tones

> Shepard tones, as they are called, can be created using additive synthesis, though they are not based on sounds that occur in nature, nor are they based on tones produced by artificial musical instruments. Shepard tones exhibit the peculiar property of circular pitch in the sense that if one constructs a musical scale out of them, the pitches eventually repeat themselves when the notes of the scale are played consecutively upward or downward. The illusion is that the notes have either continually increasing or decreasing pitch while simultaneously staying at or near the same place[...]. (Moore 1990, p. 221)

The endless riser is based on an acoustic illusion discovered by Roger Shepard in 1960. The effect created is an infinite *glissando*. The pitch seems rising up forever. The effect is often compared to the ever-ascending staircase by the visual artist *M. C. Escher* displayed in Figure 1.15.

In Figure 1.16, we see how the effect works. We need to construct several waveforms with distances of an octave to each other. The example here shows eight oscillators. The

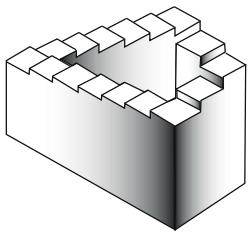

Figure 1.15.: Ever-ascending staircase (Sethares 2007, p. 10)

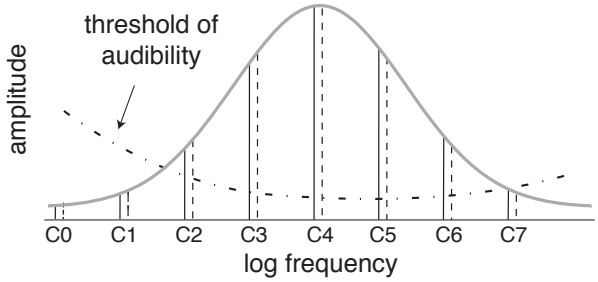

Figure 1.16.: Shepard tone envelope (Sethares 2007, p. 12)

frequencies of the oscillators are constantly rising upwards. The amplitude of each oscillator is determined by the curve overlaid in the graph. The amplitudes are increased in the lower frequencies and decreased in the highs. Figure 1.16 also includes the threshold of audibility. Everything below the intersection of this line and the amplitude curve cannot be heard. So the oscillator with currently the lowest frequencies become perceptible over time as the oscillators with the highest frequencies fade out. When an amplitude drops below audibility at the fade out we can set the frequency to an octave lower than the oscillator with currently the lowest frequency. Repeating this procedure creates the illusory pitch trajectories.

1.3. State of the art

Today producers handcraft risers in DAWs or use existing risers from sample libraries. Handcrafting risers is complex and time-consuming. The prerequisite is a sample that usually will be modified by automation. The sample includes some kind of noise or different oscillators. Noises or basic signals can also be generated directly in the DAW by a signal generator. The signal is then bounced (meaning recorded) to a track and the resulting state will be the same as a sample from a library that had been made without modulation. Afterwards the sample can be modulated via the DAW's automation

possibilities or any plug-in inserted in the effect chain.

At the time this thesis was written, I found software that addresses the need for riser tool. One is *Crescendo* (Norris 2013). *Crescendo* is built in Max/MSP (Max)[2]. When creating a Max patch[3] it can be exported as a standalone application. *Crescendo* also uses the classical riser elements mentioned earlier. With the tool we can create audio files and export them with a certain BPM. The problem occurs when we use these risers in DAWs: we always have to go back to the tool, export and import again to our track. This destroys a solid workflow. The second big problem I discovered in this specific plug-in is the graphical user interface (GUI). There are no intuitive controls for automation—rather we find knobs for start and end values of a parameter and another one for shaping a curve. Furthermore, the tool lacks the graphical representation of the curve and the paradigm "trust your ears" does not really apply here.

The second software I found is the *Point Black Riser* (Herbert 2012). This is a *Max for Live* device. So it can be used realtime in *Ableton Live* as a plug-in. This has a better workflow compared to *Crescendo*, but if you do not own *Ableton Live* you are excluded from using the software. Also, if you are using a different DAW you again have to import and export. The interface is slicker, but still lacks intuitive controls for best usability. Compared to *Crescendo* it is completely free.

Both of the discussed riser tools are implemented with native Max objects. This objects are optimized for fast rendering on their own, but it could be further optimized when packaged into a single C plus plus (C++) class. This is the main reason why this tools are not available in plug-in formats.

The last piece of software I found is the *Endless Series* by Oliver Larkin (Larkin 2012). This is a synthesis and effect plugin based on the *Shepard Tone* auditory illusion described in subsection 1.2.9. The plug-in is released in every modern plug-in format, so it could be used in any song production. The artistic use of this plug-in in EDM can be doubted though. So far I do not know any EDM song which uses this theory. For my product I left out the endless riser. One reason was that a solution already existed with the *Endless Series*.

[2]MSP stands for either Max Signal Processing or Miller S. Puckette, the "inventor".
[3]A Max patch is a graphically programmed manipulation or creation of sound.

2. Solution

2.1. Description of approach

A riser plug-in is not based solely on one algorithm. Multiple audio elements are connected and building a riser effect over time. We have oscillators, filters, noises and other effects. For all of these elements different algorithms exist. But the inputs and outputs are mostly similar. Riser effects are used today in almost every EDM track. Producers today combine these elements themselves. The goal is to bundle the elements together and create a simplified user interface.

With this knowledge we can copy the concept of combining multiple simple elements to the development strategy of a riser plug-in. I approached my idea in Max: small objects, which are either direct or signal processing (per sample) operators, can combined to a bigger patch or abstraction. In C++ we have the equivalent classes that pursue the same purpose: modular, reusable objects. All we have to do is to build stable and flexible objects with an easy to use interface. Afterwards we can merge them together to a more complex product.

A riser plug-in can be divided into sound generators, filters, modulators and effects. The generators like oscillators and noise need to be built first. This is not a very innovative task. Rather we can use existing algorithms that are already approved in stability and sound quality. Merging algorithms in a way that the end user can effectively work with it is the key in every audio plug-in that uses, for example, synthesis or filters. The signal flow in an audio plug-in is most important. That is why the next object on the list of implementation is the filter. The same as for generators applies here—we can reuse software and optimize for our needs.

The *classes* we collected provide us with a lot of parameters that have to be controlled. Especially a riser has to provide a proper strategy to avoid confusion. Risers built today by EDM producers use a lot of automation—and there lies the problem: it takes a lot of time and might often develop to a complex and confusing state. My goal was not only to simplify the user experience but also to reflect the simplicity in good and modular code.

2.2. Prototyping and artistic evaluation

2.2.1. Anatomy of an audio plug-in

To build an audio plug-in, the step was to look at the structure of a simple plug-in. For this thesis I will define a simple plug-in as a plug-in with only one algorithm. First we can divide the algorithm from the rest, which are the elements for the wrapper, which is used to talk to a plug-in host (DAW) defined by a certain plug-in format. When programming plug-ins, each format has an application programming interface (API), which is an interface language so the programmer can write code that talks to host.

Let us look at the Virtual Studio Technology (VST) plug-in format as an example. VST plug-ins are member of either the VST effects or the VST instruments category. A riser plug-in here is categorized as a VST instrument, because it does not use input samples and does not manipulate them with digital signal processing. Instead a signal is generated by sending control values to a system that outputs audio blocks. The user can manipulate control values via musical instrument digital interface (MIDI) messages or parameter changes via the GUI.

2.2.2. Measuring plug-ins

When I developed my first plug-in during my internship at AIR before the thesis, I discovered the importance of measurement to avoid mistakes. These mistakes would show up later in the frequency response and phase response. Most of the time the mistakes are not noticed and detected by the sound engineer. He does not care about technical and physical mistakes. His judgement is based on artistic evaluation. These

errors should be fixed already in the development phase of the plug-in. This is what makes a good audio plug-in.

I tested different algorithms with *RackAFX* (Pirkle 2012), which has analyzers for frequency and phase response. Yet this workflow is only possible in the development stage and cannot be applied to the final product. With product I mean the VST plug-in itself. There is a good analyzer for doing quality assurance for the final product. It is called *VST Plugin Analyser* (Budde 2012). Observing the behavior of a plug-in while changing the parameter reveals avoidable artifacts. For example, it is nice to have a knob which has a range from 0-12 dB, but when the plug-in starts to wrap the phase at a value higher than 10 dB, the range should be limited. The plug-in will sound different after this point and not in a good—neither in an expected way. The sound engineer would maybe choose this setting with unexpected and incalculable results.

2.2.3. Max

Max is a modular environment. The concept of a modular environment is to have small abstract objects that perform a certain task. These objects are connected with virtual wires.[4] Digital signal processing algorithms are stored in these objects. So in Max we can avoid actual coding but graphically connecting the objects to a bigger system that modifies an audio signal or simply creates it.[5]

I used Max for effective prototyping in multiple stages to the final plug-in. Max keeps you focused on the goal because you can always see and—more important—listen to the result. When coding problems occur you could get lost and frustrated. To avoid this dilemma we can use Max.

Max objects[6] are usually coded in C and not C++. We need a wrapper *class* to built source code that later can be reused when developing an audio plug-in. For this purpose I found *MaxCPP* (Wakefield 2013). The wrapper allows us to write in C++. In theory we just have to include a header file that wraps the code into C when compiling.

[4] Max has two wire types: one for control data and another for signals.

[5] For prototyping we can use the objects available in Max but their source code is not available. From the theory though we know how an oscillator works and how to rebuild, for example, an object like `cycle~`. The `cycle~` is simply a sine wave oscillator, where we can set frequency and phase.

[6] Max users talk about *externals*. The API can be found online on the *Cycling '74* website (Max/MSP/Jitter Software Development Kit 2013).

While experimenting with the *MaxCPP* wrapper (Wakefield 2013) I had to fix and improve the source code. The *MaxCPP* sources were committed in March 2013, but they did not work with the current *MaxSDK* and also missed essential features for my work. One example is the implementation of the *MaxSDK* `assist()` and `dblclick()` function.

I developed a prototype of the riser in Max which was written in C++. This prototype was first a Max patch constructed with objects available in Max, but soon I switched to an external and implemented the algorithms. The language of the external is C++ and was built with the concept of object orientation. This allowed me to reuse everything later. I also experimented with Max objects to get new ideas and best sound. It was possible to quickly adjust sonic elements to taste in the Max environment and use listening to evaluate the technical construct.

2.2.4. MIDI

The riser plug-in responds to MIDI. MIDI is a protocol to send musical control values between electronic devices. Also, it is used in musical computer software for exchanging data. For my prototype I had to write my own *class* that responds to MIDI data. MIDI uses a serial data stream for communication. To parse the stream we have to know how to decode it. The main block are MIDI messages, which are three bytes long. In Table 2.1, the structure of a note-on message is listed: in this case we have a note-on message of a middle *E* with a velocity of 89 sent on channel five.

	Status Byte	Data Byte 1	Data Byte 2
Description	Status/channel #	Note #	Attack velocity
Binary data	(1001.0100)	(0100.0000)	(0101.1001)
Numeric value	(Note on/CH#5)	(64)	(89)

Table 2.1.: Structure of a MIDI note-on message (Huber 2007, p. 16)

We see that the status byte[7] is divided into two four bit blocks: the first indicates the message type and the second is the midi channel. The following two data bytes are dependent on the message type. I implemented note-on/off, continuous controller and pitch bend in my prototype. To indicate a status byte its first byte is always set to one;

[7] A byte is a binary number of eight bits.

to indicate data bytes the byte has a leading zero. This is why MIDI data has only a resolution of seven bits.

```
1  long   data[3];  ///<  MIDI input stream has to be separated
2                   ///<  into an array of size 3
3
4  int    messageType =  (data[0] & 0xF0) >> 4;
5  int    channel     =  (data[0] & 0x0F) + 1;
6
7  bool   isNoteOn    =   messageType == 0x8 && data[2] > 0;
8  int    note        =   data[1];
9  double frequency   =   440.0 * pow(2.0, (note - 69) / 12.0);
10 int    velocity    =   data[2];
11 double gain        =   velocity / 127.0;
12
13 int    controller  =   data[1];
14 int    value       =   data[2];
15
16 int    pitchbend   = ((data[1] & 0x7F) << 7) +
17                        (data[2] & 0x7F)       - 0x2000;
```

Listing 2.1: MIDI parsing (C++)

$$f(n) = 440(2^{\frac{n-69}{12}}) \tag{2.1}$$

Listing 2.1 shows some examples how to parse incoming midi data that I used in my *class*. In line 7-11 we determine if we have a note-on message. Then we fetch the midi note from the array. In the next step the MIDI note number is converted into frequency. Equation 2.1 is the equivalent formula for calculating a MIDI note to frequency as in Listing 2.1, line 9. To get the volume of the note in the range of 0.0 to 1.0 we only have to divide the incoming velocity by 127.0.

2.3. Implementation

2.3.1. Oscillators

The oscillator class that I used in the riser plug-in is a modified version of an existing class. The class was named `VirtualAnalogOscillator`. It is based on the principles

described in subsection 1.2.5. With its band-limited character it emulates an analog oscillator.[8] This algorithm uses a sampled step function to emulate analog behavior. It is stored in a wavetable and the look-up is performed on runtime.

I chose this oscillator because it can render different sawtooth and square waves including, for example, supersaws. Supersaws are considered "fat" sounding saws. This sound is generated by stacking multiple saws—all with a different initial phase. In the original algorithm each starting phase was hardcoded. This means, there were fixed values in the code. I changed them to dynamic values—in the end the starting phases were calculated randomly. The riser uses two instances of the oscillator: one for left and one for right. The different starting phases create a stereo effect[9]. Up to seven waveforms are added per oscillator instance in the available oscillator modes.

Mode	Description	Shape Control
hard sync	two hard synced oscillators	shift up to six octaves
cross modulation	modulator (triangle) modulates carrier	shift up to six octaves
multi	seven oscillators with phase offset	detune the oscillators
pulse width	square with pulse width modulation	pulse width

Table 2.2.: Oscillator modes and shape control

The oscillator has a shape parameter that controls different values depending on the mode. Table 2.2 list different operations the algorithms can be forced to. Each mode has a base oscillator with either a sawtooth or a square wave. Pulse width modulation is only available with the square as waveform. This results in a selectable amount of seven different oscillator modes.

2.3.2. Noise

As seen in subsection 1.2.6, noise can be created with just random numbers that are uniformly distributed. I found a fast algorithm online (Fast Whitenoise Generator 2006) that is almost uniform and produces white noise without using internal random functions of C++. I surrounded the algorithm with the necessary class environment to make it usable in the object oriented environment.

[8] Analog oscillators cannot create steep slopes. But we could, for example, create a "perfect" square wavetable with ones and zeros when using digital signal processing.

[9] The inter-aural time difference is necessary for our spatial perception. When small time variation on the stereo channels of an audio stream occur, we cannot localize the sound but perceive it as spatial.

2.3.3. Filter

The filter used is also derived from an existing class called *VirtualAnalogFilter*. It is based on the bi-quad concept explained in subsection 1.2.7. The filter here has 23 working modes. The modes are achieved by setting the coefficients in the algorithm accordingly. Table A.1 in Appendix A, lists all the possible filter modes with their corresponding attenuation.

There is one filter that can be applied to the individual generators in the latest version of the plug-in. The output signal of the generators with the filter active are added together and then filtered. Generators with filter bypassed are also summed up and then mixed with the filtered output. The design of the code is greatly abstracted, so it would be easy to add more filters to gain more flexibility later. At this point I wanted to keep the interface simple and avoid overloading the user.[10]

2.3.4. Parameters

The core of the riser plug-in is automation. My solution is a *struct*, which is a data structure in C++. A struct can contain several values and functions. In Listing 2.2, I simplified the original code to demonstrate my solution. From top to bottom you will find the constructor, the member variables and the `render()` function. The constructor takes the address of the `riser` control variable from the `Riser` *class* as an argument.[11]

The *struct* listed here provides a only linear automation for demonstration purpose. In the final code it is greatly extended: for example, I coded different `render()` functions to apply curving. The `riser` value runs from 0.0 to 1.0. When the `render()` function is called Equation 2.2 is applied. x corresponds to the `riser`, a is the start and b is the end value. It is obvious that if the start value is higher than the end value we have a fall rather than a riser: $b - a < 0$. Figure 2.1 illustrates an example with $a = 0.7$ and $b = 0.3$.

[10] This by the way is a common strategy nowadays in software development. Additional filters are a good selling argument for version 2.0 of a riser product where the effort for the implementation it is very low.

[11] As we can see in Listing 2.3 the `Automated` *struct* is a member of the `Riser` *class*. The *struct* does not know its parent. So we have to hand over the address of the riser variable via the constructor of the `Riser`. The variable is in the *public* scope of the `Riser` and can therefore be accessed from within the *struct*.

```
struct Automated
{
  Automated(double &riser)
  :
    riser(riser)
  {}

  double &riser;     ///<  The riser value of the Riser instance.
  double begin;      ///<  Start value for the automation.
  double end;        ///<  End value for the automation.

  inline double render()
  {
    return (end-begin) * riser + begin;
  }
}
```

Listing 2.2: `Automated` data *struct* (C++)

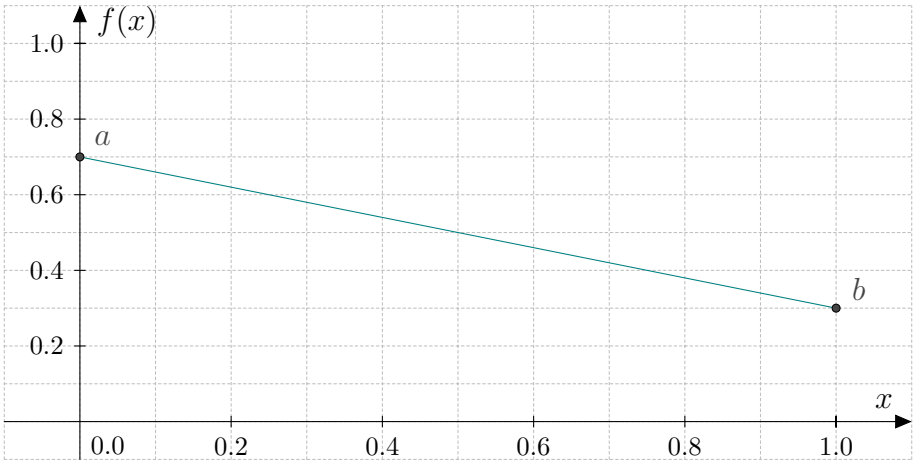

Figure 2.1.: Fall automation of a parameter

$$f(x) = (b - a)x + a \tag{2.2}$$

All the parameters can be automated easily with this *struct*. We declare multiple parameters with the data type `Automated` as shown in Listing 2.3. The member variables of a *struct* are always in the public scope, so we can access them from the outside. For example, when building the user interface the variables can be accessed through the *Riser* instance. An instance of a `Riser` *class* is in our case an audio channel. The plug-in

```
class Riser
{
public:
  double riser;

  Riser()
  :
    filterFrequency(riser),
    filterResonance(riser)
  {}

  struct Automated
  {
    /* insert code here */
  }

  Automated filterFrequency;    ///<  Filter cutoff frequency
  Automated filterResonance;    ///<  Filter resonance
};
```

Listing 2.3: `Automated` used in `Riser` *class* (C++)

```
Riser riserLeft;
riserLeft.filterResonance.begin = 0.7;
riserLeft.filterResonance.end = 0.3;
```

Listing 2.4: Accessing the `Automated` *struct* (C++)

outputs a stereo stream so there exist two *Riser* instances in memory. Listing 2.4 is an example of how to access the start and end values.

2.3.5. LFO modulation

LFOs are low frequency oscillators. The frequency needs to be low so we can perceive the modulation effect—if the frequency is too high we perceive another timbre—like in frequency modulation—rather than a modulation of a parameter. For the riser I decided to build an LFO that is able to produce the "classic" waveforms. My *class* is called `CheapLFO`. It is based on an idea I found online (Another Cheap Sinusoidal LFO 2004).

The initial code can create sinusoidal and triangle waveforms. I optimized them for my needs and added sawtooth and square. Also, I programmed a smoothing parameter where the slope can be adjusted. The LFO concept is different than that of a sound generating oscillator: we don't need to emulate analog behavior. An LFO is just an element modulating another element that in our case already tries to emulate analog sound. For that reason I decided not to use the oscillator I described in subsection 2.3.1. Furthermore, the implementation of `CheapLFO` is a lot faster than that of `VirtualAnalogOscillator` because it does not use the wavetable step function.

```
1 inline double render()
2 {
3   return (depth * lfo) * ((end-begin) * riser + begin);
4 }
```

Listing 2.5: `Automated render()` function extended (C++)

The LFO can be integrated in the `Automated` *struct* that is described in subsection 2.3.4. The `render()` function of `CheapLFO` just returns a *double* value between -1.0 and 1.0. This value is stored in a member variable `lfo` in the `Riser` *class*. As we have seen with the `riser` variable, we also have to announce the address of this variable to every `Automated` *struct* in the constructor of the `Riser` *class*. The `Automated` *struct* is extended by a `depth` value that defines the amount of modulation applied to the parameter. The new render function looks as printed in Listing 2.5

2.3.6. Synchronization to the host and MIDI control

Every DAW has an option to set the song tempo. Usually this is in BPM. The host provides this tempo to its plug-ins. In Listing 2.6, the variable `samplesPerBeat` includes the BPM indirectly. With this value we can calculate the duration of the riser in samples. `riserCounter` is a control variable that is incremented with the sample rate provided by the host and reseted to zero every time it is equal to `durationSamples`.

```
1 durationSamples = int(durationBeats * samplesPerBeat);
2 riserLeft.riser = (double)riserCounter / (double)durationSamples;
```

Listing 2.6: Calculation of the `riser` variable (C++)

`durationTime` has to be set by the user somehow. It was implemented so the user can set it via the MIDI note received from an external keyboard controller or MIDI from the host. I made this choice to give the plug-in a playability and also the possibility to exactly control the riser via the MIDI timeline of the host. For example, when MIDI note 60 is played, the duration of the riser is two bars. Transposition of the same key an octave down extends the riser as transposition upwards shortens the riser duration.

All the keys in an octave on the keyboard are reserved for presets. Presets were implemented in the prototype but not yet in the plug-in version by the end of the thesis.

2.3.7. GUI design

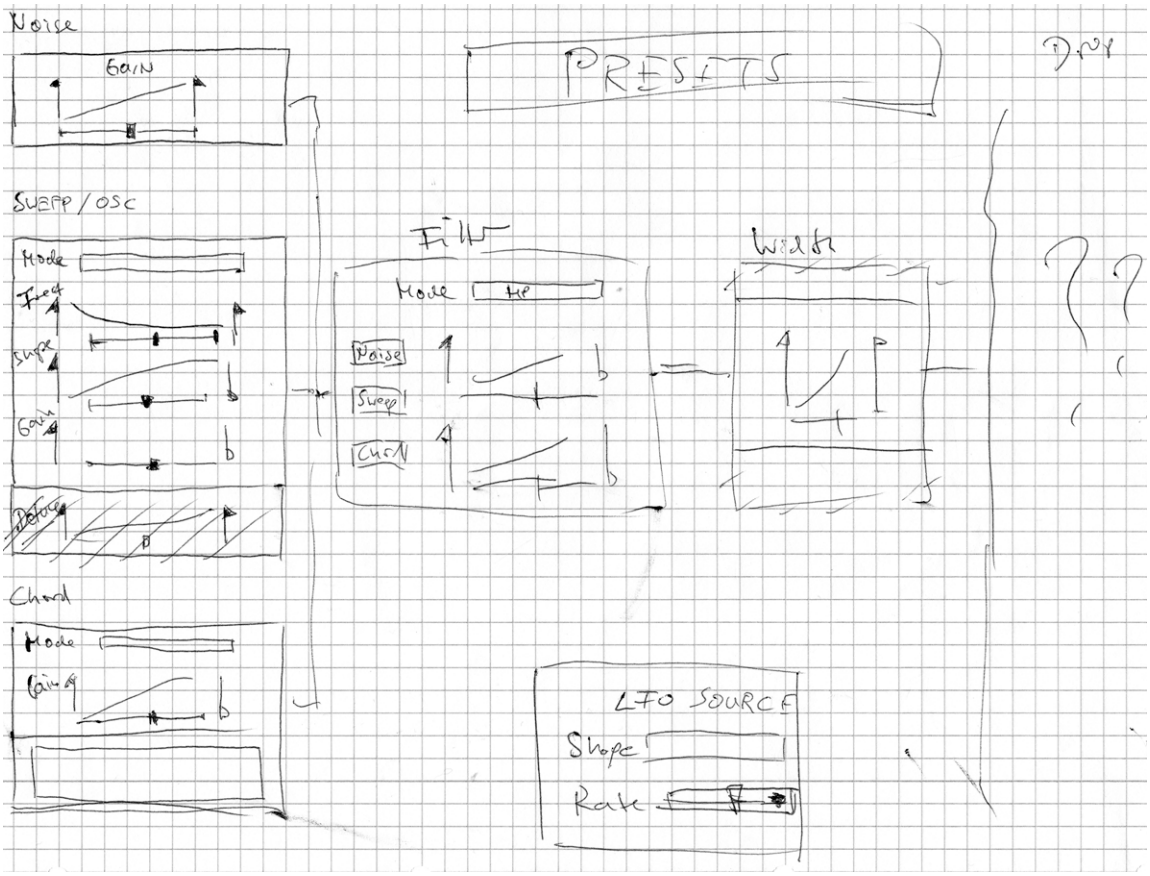

Figure 2.2.: GUI sketch

The biggest challenge in building the GUI was to find a suitable interface to control all automations without having too many knobs: my goal was to have as few as possible. This led me to the automation controller I finally built. If you look back at Figure 2.1

you see the basis for the controller. I prototyped the idea in Max and implemented it later as follows: Imagine the two points a and b as vertical sliders where you can set the start and end points. If you grab the straight line from a to b with the mouse and drag up or down the line can be curved. The next addition was a bipolar horizontal slider below the element. Moving to the right it adds positive modulation to the assigned parameter; moving it to the left negative.

The structure of the GUI follows the signal flow from left to right. Generators where put on the left. The natural signal flow is then through the filter with its controls and finally the stereo width control. The LFO which is separated in the signal flow was put aside and placed to free space that was left. By the end of the thesis there was a GUI that has all the controls for using the riser plug-in. You can look at a development sketch in Figure 2.2.

3. Results

As we saw in chapter 2, a riser plug-in was created that is available in formats usable in modern DAWs. The plug-in consists of the necessary elements for creative use to develop basic riser ideas and realize them. A riser is defined by a static skeleton with fixed connections between the modules. Moreover, the user is provided with as much freedom as possible to pursue his own urges. This freedom is represented in the GUI and especially in its elements.

On the other hand a working strategy and process was developed to effectively prototype audio plug-ins in Max. The transformation effort into an audio plug-in was greatly optimized. Solutions were found for the technical problems that were encountered during the process of matching the two technologies: Max and audio plug-ins. The workflow I designed was the basis to build a good riser plug-in and will be reused for further projects at AIR Music Technology.

A prototype is now available as a standalone Windows and Mac application. The audio plug-in can be compiled as VST, AudioUnits (AU), Real-Time AudioSuite (RTAS) and Avid Audio eXtension (AAX). The prototype and the plug-in include MIDI support: you can play the plug-in via a keyboard controller. The parameters of the audio plug-in can be automated and thus it is possible to control them from an external hardware controller via MIDI learn. The duration parameter of the riser is synced to the tempo of the host.

Conclusion

The construction of the riser was not a static progression. It was a dynamic process with a lot of creativity involved. Switching back and forth between programming and listening is the main work that has to be done to develop an audio plug-in. The thesis report gives an insight which tools are needed to build the riser software and describes the theoretical background. The strategies described were essential for the success of the project. It is recommended to use an effective prototyping environment where audio algorithms can be tested and evaluated for their sonic quality.

A basic riser plug-in was the result of this work. Still, the plug-in has a lot of potential for updates. The plug-in is easy to extend because of the object oriented approach that was used. For example, after a riser ends, there should be a drop that fades out the riser. Also, different noises can be added to prepare for even more exotic sounds. Different presets have to be created to give the user an insight into what is possible with the plug-in. Effects like reverb can be added in the signal chain of a DAW, but maybe it is easier to have them included, so they are part of a preset. The product that was created will be released as software in the future. The implementation of a drop, presets and different noises is already in progress.

Schedel (2008, p. 29) writes: “Sound technologies are more than just tools for the creation of music; they are social artifacts.” Maybe the riser plug-in could be one.

Bibliography

Adamo, M. 2009. *The secrets of house music production* (2nd ed.). Sample Magic.

Allain, A. 2013. *Jumping into c++*. Cprogramming.com. http://www.cprogramming.com.

Another Cheap Sinusoidal LFO. 2004. http://www.musicdsp.org/showArchiveComment.php?ArchiveID=167.

Bianchini, R. & Cipriani, A. 2008. *Virtual sound: sound synthesis and signal processing - theory and practice with csound*. Contemponet.

Boer, B. H. V. 2012. *Historical dictionary of music of the classical period*. Scarecrow Press.

Boulanger, R. (Ed.). 2000. *The csound book: perspectives in software synthesis, sound design, signal processing,and programming*. The MIT Press. http://www.csounds.com/book/.

Boulanger, R. & Lazzarini, V. (Eds.). 2010. *The audio programming book*. The MIT Press.

Budde, C.-W. 2012. Vst plugin analyser. http://www.savioursofsoul.de/Christian/programs/measurement-programs/.

Butler, M. J. 2006. *Unlocking the groove: rhythm, meter, and musical design in electronic dance music*. Indiana University Press.

Carew, D. 2008. *The companion to the mechanical muse: the piano, pianism and piano music*. Ashgate.

Case, A. 2007. *Sound fx: unlocking the creative potential of recording studio effects*. Focal Press.

Chamberlin, H. 1985. *Musical applications of microprocessors* (2nd ed.). Hayden Books.

Cipriani, A. & Giri, M. 2013. *Electronic music and sound design: theory and practice with max and msp* (2nd ed.). Contemponet. http://www.virtual-sound.com/cms.

Cipriani, A. & Giri, M. 2014. *Electronic music and sound design: theory and practice with max and msp*. Contemponet. http://www.virtual-sound.com/mat2.

Collins, N. 2010. *Introduction to computer music*. Wiley.

Collins, N., Schedel, M., & Wilson, S. 2013. *Electronic music*. Cambridge University Press.

Cook, P. R. 2002. *Real sound synthesis for interactive applications*. A K Peters/CRC Press. http://www.cs.princeton.edu/~prc/AKPetersBook.htm.

Cope, D. 2011. *Righting wrongs in writing songs*. Course Technology PTR.

David, E. 2012. *Lmms: a complete guide to dance music production*. Packt Publishing.

Demers, J. 2010. *Listening through the noise: the aesthetics of experimental electronic music*. Oxford University Press.

Farnell, A. 2010. *Designing sound*. The MIT Press. http://mitpress.mit.edu/books/designing-sound.

Fast Whitenoise Generator. 2006. http://www.musicdsp.org/showArchiveComment.php?ArchiveID=216.

Herbert, D. 2012. Point black riser. http://www.pointblankonline.net/free-stuff/free-plugins.php.

Hoffer, C. 2011. *Music listening today* (4th ed.). Cengage Learning.

Huber, D. M. 2007. *The midi manual: a practical guide to midi in the project studio* (3rd ed.). Focal Press.

Hugill, A. 2008. The origins of electronic music. In N. Collins & J. d'Escrivan (Eds.), *The cambridge companion to electronic music* (pp. 7–23). Cambridge University Press.

Kamenov, A. 2013. *Digital signal processing for audio applications*. CreateSpace Independent Publishing Platform.

Larkin, O. 2012. Endless series. http://www.olilarkin.co.uk.

Lennox, P. 2011. Spatialization and computer music. In R. T. Dean (Ed.), *The oxford handbook of computer music* (pp. 258–273). Oxford University Press.

Liebman, D. 1991. *A chromatic approach to jazz harmony and melody*. Advance Music.

Loy, G. 2006. *Musimathics volume 1: the mathematical foundations of music*. The MIT Press. http://www.musimathics.com.

Lyon, E. 2012. *Designing audio objects for max/msp and pd*. A-R Editions.

Manning, P. 2011. Sound synthesis using computers. In R. T. Dean (Ed.), *The oxford handbook of computer music* (pp. 85–105). Oxford University Press.

Manzo, V. J. 2011. *Max/msp/jitter for music: a practical guide to developing interactive music systems for education and more*. Oxford University Press, USA. http://www.oup.com/us/maxmspjitter.

Max/MSP/Jitter Software Development Kit. 2013. http://cycling74.com/downloads/sdk/.

Moore, F. R. 1990. *Elements of computer music*. Prentice Hall.

Neukom, M. 2013. *Signals, systems and sound synthesis*. Peter Lang Pub Inc.

Norris, A. 2013. Crescendo. http://digitaldjtools.net/software/crescendo/.

Park, T. H. 2009. *Introduction to digital signal processing: computer musically speaking*. World Scientific Publishing Company.

Pejrolo, A. 2011. *Creative sequencing techniques for music production: a practical guide to pro tools, logic, digital performer, and cubase* (2nd ed.). Focal Press. http://www.creativesequencingtechniques.com.

Pirkle, W. 2012. *Designing audio effect plug-ins in c++: with digital audio signal processing theory*. Focal Press. http://www.willpirkle.com.

Puckette, M. 2007. *The theory and technique of electronic music*. World Scientific Publishing Company. http://crca.ucsd.edu/~msp/techniques.htm.

Roads, C. 1996. *The computer music tutorial*. The MIT Press.

Rummenhöller, P. 1983. *Die musikalische vorklassik: kulturhistorische und musikgeschichtliche grundrisse zur musik im 18. jahrhundert zwischen barock und klassik (german edition)*. Barenreiter.

Russ, M. 2008. *Sound synthesis and sampling* (3rd ed.). Focal Press. http://www.martinruss.com/sss.html.

Schedel, M. 2008. Electronic music and the studio. In N. Collins & J. d'Escrivan (Eds.), *The cambridge companion to electronic music* (pp. 24–37). Cambridge University Press.

Sethares, W. A. 2007. *Rhythm and transforms*. Springer. http://sethares.engr.wisc.edu/RT.html.

Snoman, R. 2013. *Dance music manual: tools, toys, and techniques* (3rd ed.). Focal Press. http://www.dancemusicproduction.com/the-dance-music-manual-resources/.

Starr, E. 2009. *The everything music composition book: a step-by-step guide to writing music*. Adams Media.

Steiglitz, K. 1996. *A digital signal processing primer: with applications to digital audio and computer music*. Prentice Hall.

Summers, D. & O'Rourke-Jones, R. (Eds.). 2013. *Music: the definitive visual history.* DK.

Tech Tips Volume 1: Creating Sweeps and Risers. 2009. http://www.youtube.com/watch?v=rdd9nNg0YAQ.

Wakefield, G. 2013. Maxcpp. https://github.com/grrrwaaa/maxcpp.

Wolf, E. K. 2013. Mannheim style. http://www.oxfordmusiconline.com/subscriber/article/grove/music/17661.

Zölzer, U. 2008. *Digital audio signal processing* (2nd ed.). Wiley.

Zölzer, U. (Ed.). 2011. *Dafx: digital audio effects* (2nd ed.). Wiley.

A. Filter modes

Mode	Filter Type	Slope
LP4	lowpass, four poles	24 dB/oct
LP3	lowpass, three poles	18 dB/oct
LP2	lowpass, two poles	12 dB/oct
LP1	lowpass, one pole	6 dB/oct
HP2_LP1	asymmetric bandpass, three poles	12 dB/oct high, 6dB/oct low
HP3_LP1	asymmetric bandpass, four poles	18 dB/oct high, 6dB/oct low
HP4	highpass, four poles	24 dB/oct
HP3	highpass, three poles	18 dB/oct
HP2	highpass, two poles	12 dB/oct
HP1	highpass, one pole	6 dB/oct
BR2	bandreject, two poles	2×6 dB/oct notch
BR4	bandreject, four poles	2×12 dB/oct notch
BR2_LP1	assymetric bandreject, three poles	2×6 dB/oct n, 6 dB/oct low
BR2_LP2	assymetric bandreject, four poles	2×6 dB/oct n, 12 dB/oct low
HP1_BR2	assymetric bandreject, three poles	6 dB/oct high, 2×6 dB/oct n
BP2_BR2	tooth, four poles	2×6 dB/oct pk, 2×6 dB/oct n
HP1_LP2	assymetric bandpass, three poles	6 dB/oct high, 12 dB/oct low
HP1_LP3	assymetric bandpass, four poles	6 dB/oct high, 18 dB/oct low
AP3	phase shift, three poles	undefined
AP3_LP1	phase shift and lowpass, four poles	6 dB/oct low
HP1_AP3	phase shift and highpass, four poles	6 dB/oct high

Table A.1.: Filter modes with attenuation

Non-exclusive license to reproduce the thesis

I, THOMAS LIEB,

herewith grant the University of Tartu a free permit (non-exclusive license) to

1. reproduce, for the purpose of preservation, including for the purpose of preservation in the DSpace digital archives until expiry of the term of validity of the copyright,

   ELECTRONIC DANCE MUSIC PRODUCTION HELPER:
   A TOOL FOR GENERATING RISERS,

   supervised by Janar Paeglis.

2. Making the thesis available to the public is not allowed.

3. I am aware of the fact that the author retains the right referred to in point 1.

4. This is to certify that granting the non-exclusive license does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Viljandi, 20.05.2014