A SUPERCOLLIDER CLASS FOR VOWEL SYNTHESIS AND ITS USE FOR SONIFICATION

Florian Grond[1], *Till Bovermann*[2], *Thomas Hermann*[1]

[1] Ambient Intelligence, CITEC, Bielefeld University Germany
`[fgrond,thermann]@techfak.uni-bielefeld.de`
[2] Media Lab Helsinki School of Art and Design Helsinki Finland
`till.bovermann@aalto.fi`

ABSTRACT

In this paper, we present building blocks for the synthesis of vowel sounds in the programming language `SuperCollider`. We discuss the advantages of using vowel based synthesis, and make a review where it has already been used in sonifications. Then, we describe in detail the main class *Vowel* which handles all parameters related to the formants that are typically used for vowel synthesis. In order to simplify the handling of the *Vowel* class, we introduce two auxiliary *pseudo Ugens*: *Formants* for additive synthesis, and *BPFStack* for subtractive synthesis. This introduction of the building blocks is followed by code examples for sound synthesis, which make use of the described classes and their specific features. We finally present sample applications, showing how these building blocks can be used in sonification.

Keywords: vowel synthesis, sonification.

1. INTRODUCTION

The mapping of data features to sound parameters is one of the most widespread methods in sonification. Whilst this is a straight forward approach when the data are aptly preprocessed, the results are often unsatisfying from a sonic perspective when only simple sound parameters are used which are obviously easy to manipulate . Typical examples are pitch and level as two of the most salient sound characteristics. More complex sound features, such as tone color or timbre, that encompass the spectral evolution of a sound give many sounds their emotional appeal and enrich the listening experience. The physical characteristics of sound that mediate the perception of timbre are however more challenging in synthesis and hence difficult to access for sonification.

If the sound design does not rely on the timbre of prerecorded samples, one is only left with physical modeling as it has been used e.g. in the sonification of a rolling ball by Rath et al. [1] [2] and Lagrange et al. [3]. However, not all sonification scenarios have a corresponding physical model. Here model-based sonification opens the door to sonifications with rich spectral content but the underlying simulation processes are generally computationally expensive. In brief, the spectral characteristics which result in a richer and more interesting sonic experience are in turn also more difficult to synthesize and to control.

A fairly accessible type of sound signal with differences in the spectrum that are highly characteristic and hence distinguishable are vowels. This is why recently a growing number of sonification approaches can be found in the literature where the sound synthesis is based on vowel sounds. Examples of these sonification originate from various contexts. Ben-Tal et al. [4] used vowel synthesis in stock-market and oceanographic data. Cassidy et al. [5] used vowel synthesis to improve the diagnosis of colon tissue. Hermann et al. [6] explored intensively vowel based sonification for the diagnostics of EEG signals. Kleiman et al. used vowels in the context of an human motion display [7] for golf movements.

In this work we present classes for the sound synthesis environment `SuperCollider (SC)` which allow for a convenient and yet flexible synthesis and control of vowel sounds. The focus of this work is not on the synthesis of perfectly natural sounding vowels. We rather think of the spectral envelope that constitutes a vowel as a convenient point of entry to get control over certain aspects of timbre space. Using vowels in sonification or more generally speaking complex spectral envelops opens access to the following sound design options:

- By using vowel synthesis, we gain access to a continuous and well controllable dimension in timbre space that is orthogonal to pitch and loudness. The resulting potential to design sounds is usually less accessible and therefore less systematically explored.
- Vocal sounds 'point' to ourselves and to our capacity to make vocal utterances. Therefore a vocal sonification constitutes less a technological artifact from the outside world. It rather refers to the listeners and their embodied knowledge and ability to interpret the particular sound type.
- As it has been mentioned in [6], vowel sounds are easy to mimic. Hence, even people with less analytical listening skills can literally talk about the sonic experience of a vowel based sonification.
- There is no natural sounding object that does not vary - at least slightly - its spectral characteristics with increasing sound level. Hence spectral envelopes that can be systematically manipulated constitute a necessary complement for the one-to-many mapping scheme as proposed by Kramer [8].

1.1. Why do we need a new tool?

One problem in the field of sonification is the difficulty to reuse displays and hence it is difficult to compare the various research results. Frameworks, like for instance the sonification sandbox [1] try

[1] `http://sonify.psych.gatech.edu/research/sonification_sandbox`

to address this problem. However, these functional frameworks, which are often operated through a GUI that allows to configure the mapping, restrict how sound parameters can be mapped and hence limit the sound design space. As a consequence, these frameworks do not allow to implement sonification methods like data-sonograms [9]. Further, a large parameter space as spanned by vowel synthesis cannot be expected to be orthogonal to the perceptual domain. Therefore flexible mapping functions as demonstrated in [10] need to be constructed for an efficient use of the relevant subspaces.

We believe that the field of sonification is best served with small but flexible building blocks, this is why we focus on a library that addresses first and foremost vowel sounds. In order to fully explore the sound design flexibility we chose to implement it in SC, because many of the requirements mentioned above can only be met if the sound synthesis can be flexibly scripted through a text based programming language.

SC is considered do be more abstract compared to data flow based sound synthesis environment such as as Pd and Max/MSP, and hence has a steeper learning curve . The outweighing benefits are that SC meets typical needs when dealing with sound design, its strengths for sonification have been described in [11]. The building blocks which we implemented allow for the flexible control of both, the low level synthesis parameters (frequencies, bandwidths, and gains of the frequency components) as well as the control of the gestalt of the vowel. Manipulation on this higher level of perception include the change in brightness that leaves the sonic gestalt intact or the controlled transition from one vowel to another.

In this paper we will first describe the SC class *Vowel*, which handles all formants (typically 5, or more) parameters of one vowel sound. Then we describe in detail two auxiliary *pseudo Ugens* which are both structurally similar with the only differences that the first, *Formants*, is designed for additive synthesis and the second, *BPFStack*, offers the possibility to implement a simple source filter model of vowel sounds. We follow with synthesis examples using synthesis definitions *(SynthDefs)*. Finally we give two example demonstrations: Firstly we demonstrate how vowel sounds can be well combined with the one to many mapping paradigm [8]. Secondly, we show how the model based sonification technique known as data-sonograms can be well combined with conceptual mapping using vowel sounds as markers for categories.

We include code examples for a better integration of theory and practice and recommend to study this paper together with the implementation, The SC code and all sound examples discussed in this paper can be downloaded. [2].

2. THE CLASS *VOWEL*

Sounds that are perceived as vowels have a characteristic spectral structure i.e. their harmonic spectrum exhibits a particular shape with pronounced local maxima which typically decrease in level with increasing frequency. These maxima of the spectral envelope are known as formants and can be described with the parameters *center frequency* (f_c), *bandwidth*(Δf) and *gain*(g). Please confer Fant [12] as a standard textbook for formant definition.

[2]Note that this SC class is available as a *Quark* which can be found here `http://quarks.sourceforge.net/`, but can also be conveniently installed through SC, the sound files can be found here: `http://www.techfak.uni-bielefeld.de/ags/ami/publications/GBH2011-ASC`

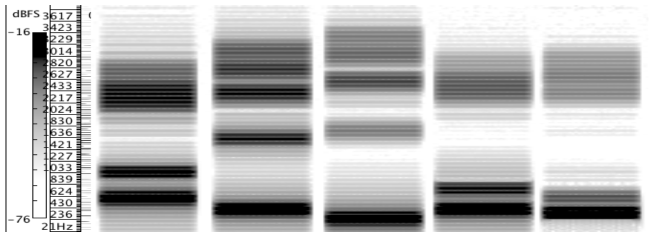


Figure 1: spectrogram for the vowels $[a:]$ $[e:]$ $[i:]$ $[o:]$ $[u:]$ for the register bass at 70 Hz

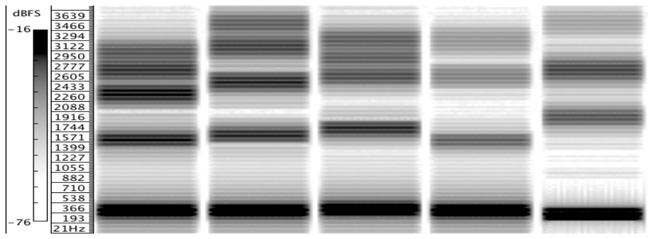


Figure 2: spectrogram for all the registers $bass$, $tenor$, $countertenor$, $alto$, $soprano$ for the vowel $[e:]$ at 70 Hz

Usually two formants suffice to recognize and differentiate the 5 vowels from the chart known in German as *vocal triangle [i:],[e:],[a:],[o:],[u:]* as in *bee, bear, bar, bot, boot.* The resulting synthesized sounds are however rather unnatural and often a set of 5 formants is taken for sufficiently natural sounding vowels. A list of the resulting 15 vowel parameters for these 5 vowels spanning across the registers *bass, tenor, countertenor, alto, soprano* can be found in the online Csound manual [3]. Figures 1 and 2 show spectrograms of generated vowels whose formants are based on the Csound manual. In all spectrograms shown in this paper, the base frequency is kept low (70 *Hz*) in order to make the spectral envelope stand out through densely spaced harmonics.

The class *Vowel* allows for a convenient instantiation of a vowel whose data structure holds an array representing the parameters f_c, Δf, g of all 5 formants from the source above.

2.1. The Formant Library

The class *Vowel* contains as class variable a library of formants that is instantiated only once. This library is initialized with the formants from the online Csound manual. The entries in the library are hierarchically ordered:

`Vowel.formLib.at(\a)` for instance returns the whole set of parameters for the vowel *[a:]* as a multilevel dictionary.

`Vowel.formLib.at(\vowel, \bass)` returns as a dictionary the 15 parameters for a vowel of the chosen register *bass* that holds arrays for the f_c, Δf, and g of all formants.

`Vowel.formLib.at(\register, \vowel, \freq)` finally returns the array of frequencies for a *vowel* of a chosen *register*. The parameters Δf and g can be accessed with the corresponding key `\bw` and `\amp`.

The inclusion of this formant library is meant to help users to focus on the sound design by choosing parameter combinations where they know what to expect in terms of the sonic result. While

[3]`http://www.csounds.com/manual/html/MiscFormants.html`

designing sounds, one often comes across parameter combinations with a particular timbre. In order to be able to reproduce and study these formant combinations, the class provides the methods *save* to save it in a specified file with a specific name and register. These file entries can be added to the library by using the *load* method which allows to reuse them as conveniently as the standard set of formants form the Csound manual.

2.2. The Instance of a *Vowel*

The entries of the formant library are all automatically assigned to the member variables $freqs$, dBs, and $widths$ of a *Vowel* instance. The instantiation without arguments `Vowel()` defaults to the parameters for the vowel *[a:], bass*. Any vowel from any register can be instantiated like this: `Vowel(\vowel, \register)`. Following the multi-channel expansion paradigm from `SC`, arrays of *Vowels* can be conveniently instantiated by either assigning an array of vowels or registers or both to the constructor.

If there is the need to compose an individual *Vowel* that originates from within the parameter space of the library above the user can create an instance with the method compose: `Vowel.compose([`$\backslash vowel_1, ...\backslash vowel_N$`,],` `[`$\backslash register_1, ...\backslash register_N$`,],` `[`$\backslash weight_1, ...\backslash weight_N$`,])`, which returns a linear combination according to the weights of the specified vowels. Note that the weights need to be a normalized sum.

However, if the user does not want to take advantage of the predefined vowels from the library, there is the option to independently define formants by specifying the parameters manually using the following instantiation method: `Vowel.basicNew([`$f_{c1}, ..., f_{cN}$`],` `[`$bw_1, ..., bw_N$`],` `[`$g_1, ..., g_N$`])`.

It is possible to alter any of the formant parameters by directly setting the elements in the arrays of the member variables *freqs*, *dBs*, and *widths*. For internal use and for convenience in combination with other synthesis *Ugens*, these member variables have the complements *midinotes*, *amps*, and *rqs*, which are returned by calling the methods of the same name.

In order to give a vowel additional extra flavor, the user can add to or remove from a *Vowel* either a single or several formants using the *addFormant* and *removeFormant* method. If the parameters are carefully chosen the sonic gestalt of the vowels might resemble those extracted from the library with the additional signature of an individual voice.

2.3. Controlling the Sound of a *Vowel*

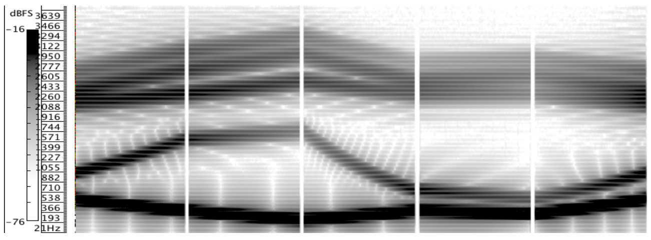


Figure 3: spectrogram for the blending between two vowels $[a: -e:]$, $[e: -i:]$, $[i: -o:]$, $[o: -u:]$, $[u: -a:]$ at the register bass at 70 Hz

A salient mapping dimension in vowel based synthesis is the transition of one vowel to another, which has been extensively used in [6]. We make this mapping dimension accessible with the *blend* method. Given the instance of two vowels *v1* and *v2*, a blending of both is implemented as a linear interpolation between the parameter sets *midinotes*, *dBs*, and *widths*. `v1.blend(v2, frac)` morphs from *v1* where *frac* is 0 to *v2* where *frac* is 1. The same method allows for more control by providing an array of 3 parameters as arguments for the morphing of all three aspects of a formant (*midinotes*, *dBs*, and *widths*) individually. Although the linear interpolation is done in MIDI notes, the corresponding frequencies in *Hz* are the units of the member variables. Figure 3 shows a spectrogram of morphing vowels.

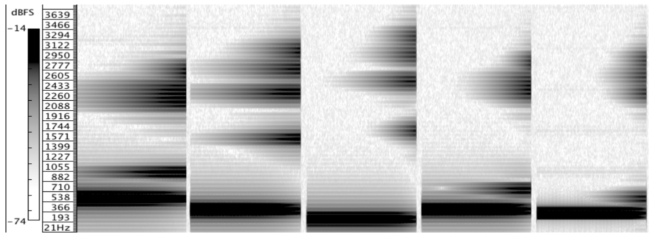


Figure 4: spectrogram for the brighten method applied to all the vowels $[a:]$ $[e:]$ $[i:]$ $[o:]$ $[u:]$ for the register bass at 70 Hz

The class also offers the options to change the brightness of a vowel sound. This is generally achieved by raising the gain of the higher formants for which there are 3 methods:

The fist method *brightenLin(b, ref)* changes the gain of a formant i by adding to the level of this formant a value that is based on the linear equation: $g_{new,i} = g_i + b\ \log(f_i) + N$. Where the term $N = (g_{new,i} - g_{ref})$ allows to compensate the change in gain of all formants by adjusting it to the previous gain of a reference formant with the index ref. This index is by default 0, i.e. the first formant. The range for the parameter b spans from negative to positive real numbers. For positive values higher formants are raised which leads to a brighter sound and for negative values they are lowered. The value $b = 0$ leaves the formants unchanged.

The second method *brightenRel(b, ref)* changes the formant gain according to the following equation (2): $g_{new,i} = b\ g_i + N$. Here all gains are simply multiplied with the factor b and compensated through the term N as in *brightenLin*. The value range of b are all positive real numbers: 1 leaves the formants unchanged, values greater 1 lower the gain of the higher vowels and values smaller 1 brighten the sound. If b is set to 0 all formants are of equal level (0 dB). A spectrogram for this method is shown in Figure 4.

The difficulty with those two methods is that the overall gain can become very big, this is why there is the third method *brightenCAmpSum* , which brightens the sound by manipulating the amplitude of the formants according to the following equation $amp_{new,i} = amp_i^b \cdot N$ where the factor $N = \frac{\sum amp_{new_i}}{\sum amp_i}$ corrects the gain of all formants so that the sum of all amplitudes remains constant, this is only an approximate compensation in order to achieve constant loudness but it yielded satisfying sonic results.

2.4. Methods that Return the Formant Data

In order to get the formant data as arrays there is the convenience methods `v.asArray` which returns `[[`$f_{c1}, ..., f_{cN}$`],[`$bw_1, ..., bw_N$`],[`$g_1, ..., g_N$`]]`. The amplitude

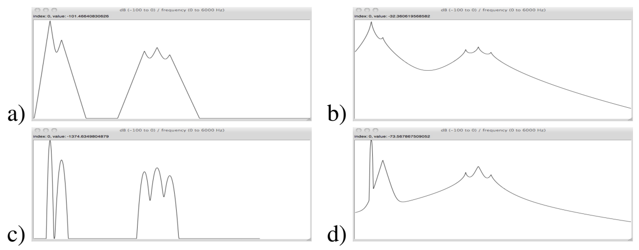


Figure 5: Typical spectral envelops as *frequency / dB* value pairs that can be easily generated through the convenience method *plot* using internally *ampAt*.ampdb.

of any frequency under the spectral envelope can be accessed by the method *ampAt*. This method takes also ranges of frequencies and for each formant the transition steepness can be modeled with an exponent as a function of the distance to the centre frequency of the formant.

The convenience method *plot* renders a visual display of the spectral envelope since it makes use of the *ampAt* exponents argument in order to control transition steepness. A selection of resulting plots is shown in Figure 5. Synthesis options based on *ampAt* are discussed in detail below.

3. THE AUXILIARY PSEUDO UGENS

The class *Vowel* handles only the formant parameters. It exhibits its full potential together with the two new *pseudo Ugens* `Formants` and `BPFStack` . Whilst Formants is designed for additive synthesis, *BPFStack* can be used for subtractive synthesis of vowel sounds. As *pseudo Ugens* they are implemented in *sclang* and contain the methods *ar* which instantiate and return a collection of *Ugens*. In a nutshell they wrap around each formant the desired unit generators.

3.1. Formants: pseudo UGen for additive Synthesis

Formants is based on the already existing Ugen *Formant* which generates a set of harmonics around a centre frequency at a given fundamental frequency. Formants takes the arguments *baseFreq, vowel, freqMods, ampMods, widthMods* and *unfold*. The formant parameters from the argument vowel (an instance of *Vowel*) are assigned to as many *Formant Ugens*, as the instance of vowel holds formants. If the flag *unfold* is set to the default value false, *Formants* returns a BinaryOpUGen as the sum of all the *Formant Ugens*. Additive synthesis of a *soprano [o:]* vowel sound with a 200 *Hz* fundamental becomes hence as easy as:

`{Formants(200, Vowel(\o,\soprano))}.play.`

In order to make the sound more lively and dynamic, there are the arguments *freqMods, ampMods, widthMods*, which are by default set to 1. Each of these arguments can be either a single modulator such as a *SinOsc.kr* with an appropriate offset, which is then uniformly applied to all 5 formants, or it can be an array of modulators thereby modulating each formant individually.

The last in the argument list is the flag *unfold*, which is as already mentioned by default set to *false*. This flag controls whether the stack of *Formant Ugens* is summed up or returned as an array. In those cases where the programmer wants to keep the vowel as one auditory perceptual unit, it might be preferable to keep it *false*. If set to true each *Formant* occupies its own synthesis channel and further options to manipulate each *Formant* such as individual spatialization are available.

3.2. BPFStack - pseudo UGen for substrative Synthesis

Whilst additive synthesis of a vowel sound leads already to recognizable sounds, even more natural results are usually achieved with an independent source filter model. Here the source is typically modeling the larynx and the filters mimic the resonance of the vocal tract assuming that the centre frequencies of the resonators coincide with the formants. This formant synthesis is described in Klatt [13].

Such a simple synthesis model can be realized with *BPFStack* which is in its structure analog to *Formants*. As the name suggests the basic *Ugen* of *BPFStack* is a band pass filter. The argument list *[in, vowel, freqMods, ampMods, widthMods, unfold]* differs only in the first argument. Unlike *Formants* where the first argument *baseFreq* is a scalar base frequency, here *in* is the sound source which is sent through the band pass filter stack.

BPFStack allows users to synthesize both, voiced and unvoiced vowels, which depends on the sonic characteristic of the *in* signal being either a pitched or unpitched source. An example of a subtractive vowel synthesis with a pronounce fundamental at 200 *Hz* would be:

`{BPFStack(Impulse.ar(200), Vowel(\a,\soprano))}.play.`

Unvoiced vowel sounds can be realized as:

`{BPFStack(WhiteNoise.ar(), Vowel(\a,\soprano))}.play.`

The variation between these two sound features has already been used as a salient parameter mapping dimension in vowel based sonification in [6].

4. FURTHER WAYS TO USE THE SPECTRAL ENVELOPE OF VOWELS

The additive and subtractive synthesis schemes from above are efficient options to generate sounds that are rich in their spectral dynamics. Using the method *ampAt* in combination with the Ugens *Klang / Klank* and their time varying equivalents *DynKlang / DynKlank* gives even more flexible synthesis options, which are however computationally more expensive. In the sequel we will give two examples, one for additive and one for subtractive synthesis.

4.1. ampAt with DynKlang: additive synthesis

DynKlang is a bank of sine oscillators, which is basically a wrapper around *SinOsc UGens*. The parameters of this oscillator-bank can be dynamically set after launching the sound synthesis process. The method *ampAt* allows to extract arrays of amplitudes for arbitrary selections of frequencies. In Figure 6 for instance, we show how two different effects can be dynamically changed: One is the alternation of even and odd harmonics. The second effect is the increasingly lowered transition steepness of the formants. The convenience of using *DynKlang* is that the sum of the gain of all oscillators can be easily limited by applying the method *normalizeSum* onto the amplitude array. The effect of this limitation in gain can be also seen in Figure 6.

4.2. ampAt with DynKlank: substractive synthesis

When using a source filter model for vowel synthesis, the perceived pitch is determined through the base frequency of the ex-

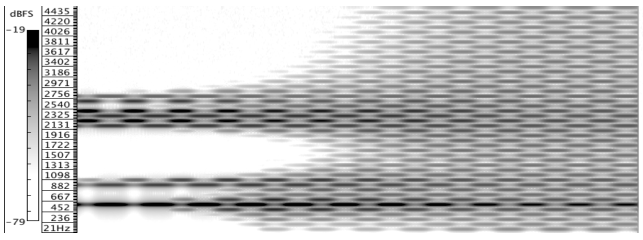


Figure 6: spectrogram showing the alternation for even and odd harmonics in the formants for $[a:]$ $tenor$ at 70 Hz together with a decrease in the transition steepness of the formants

citatory signal. In sonic interaction design a new trend of augmented auditory objects is currently emerging. In this recent approach naturally occurring interaction sounds with objects are used as sound sources and additional information is imprinted into the sound through filtering. A recent example by Bovermann et al. can be found in [14]. As a result the pitch is already fixed through the nature of the interaction sounds. Here the Ugen *DynKlank* comes in handy, which is a resonator bank that can simulate the resonant modes of an object. Similar to its additive equivalent *DynKlang*, *DynKlank* takes an array of frequencies, amplitudes and ring-times in order to configure its resonators. The frequency array gives the possibility to influence the perceived pitch although the excitatory signal has no pronounced pitch itself, such as in attack or friction sounds. The spectrogram of the prototype of an auditory augmentation using surface friction as sound source is shown in Figure 7.

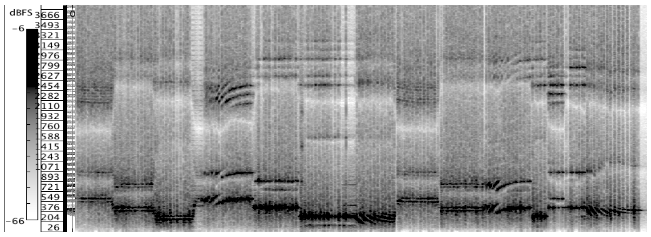


Figure 7: spectrogram for the auditory augmentation of interaction sounds $[a:]bass, [i:]bass, [o:]counter\ tenor$ at $50 - 150$ *Hz*. The slow transition between the frequencies leads to the glissando like structures in the spectrogram

5. USING THE PSEUDO UGENS AS SYNTHESIS NODES ON THE SERVER

Often in sonification applications, the SC server is used without *sclang*. This is of particular interest for the port of the SC server to mobile platforms. Hence it is desirable to be able to include many vowel related parameters and methods to manipulate them in a *SynthDef*. This means that the sound design can be conveniently made in SC, but the actual application only requires the compiled *SynthDef* to be invoked on the server. One or several vowels can for instance be instantiated by the constructor or the *compose* method within a SynthDef without problems. Also the methods for blending *Vowels*, as well as the methods that brighten *Vowels* can be used within *SynthDef*, i.e. they do not contain any flow control statements that would not be properly executed after compilation.

However, if the programmers wishes to take advantage of *sclang* there is the convenient method *addControls* that allows to create control busses within the *SynthDef*. These buses can later be conveniently set by sending to a *Vowel* the *asKeyValuePair* message in order to correctly distribute the data structure of a *Vowel* to the created control busses.

6. SAMPLE SONIFICATION APPLICATIONS

6.1. One to many mapping

The following example shows a prototype of a *one to many* mapping approach [8]. We chose as dataset for sonification the z time series of the Roessler system.[15] This prototype of a nonlinear dynamical system exhibits chaotic behavior i.e. small deviations grow into exponentially deviating trajectories. Hence it is of interest to make even small variations noticeable on the display at any range they occur.

The z variable of this system gives a spiking time-series, which poses a particular challenge for the parameter mapping. One solution is to chose a logarithmic scale. Additionally a *one to many* mapping can be used to make deviations stand out in different amplitude ranges. This saliency over a wide range is particularly important for the given dataset when looking at the distribution of this spiking time series:
10 $\%$ of the data-points are found in the lower 0.015% range of the amplitude.
50 $\%$ are in the lower 0.037% of the amplitude range.
90$\%$ of the data points are within the lower 2.5 $\%$ range of the amplitude.

We applied the following mapping scheme on the logarithmic data in order to control synthesis of a *Vowel* using the *BPFStack*:

- the 0 to 30 percentile is mapped to an Δ gain of 90 dB.
- the 20 to 50 percentile fades between unvoiced and voiced.
- the 40 to 70 percentile blends between the vowels *a* and *i*.
- the 60 to 90 percentile change the pitch from 82 to 116 *Hz*.
- the 80 to 100 percentile brightens the vowel.

All ranges overlap so that the evolution of the different sound parameters build a single sound stream. The spectrogram in Figure 8, shows the mapping strategy: The transition between unvoiced and voiced is visible as repeatedly emerging partials. The transition between the vowels can be best seen in the changing first formant. The pitch variation corresponds to the changing location of the partials and the brightening of the sound lifts the gain of the higher partials.

The realization of this mapping was straight forward using the methods of the *Vowel* class. One interesting aspect of this mapping strategy is that the mapping range of of all parameters in the sound domain can be kept small but the perceived sum of all effects yields a highly differentiable result.

6.2. Vocagram, A sonogram with *Vowels*

The second sonification which we developed based on vowel synthesis is a data-sonogram. Data-sonograms belong to the category of model based sonifications, but apart from the data preparation step it can be implemented as a parameter mapping scheme. This is

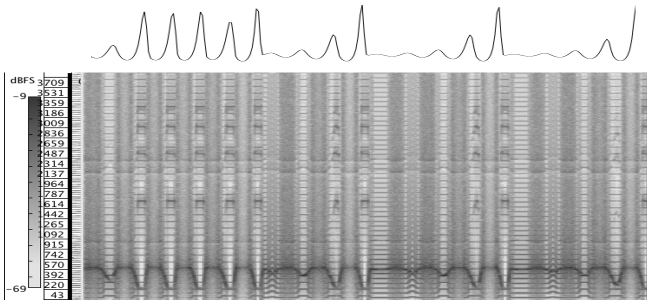
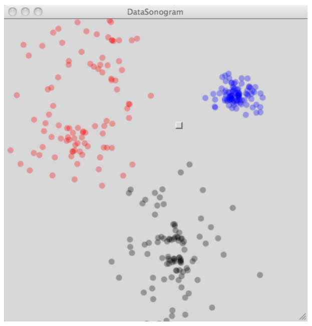


Figure 8: spectrogram showing the described *one to many* mapping, on top there is the logarithmic time series of the z variable scaled between 0 and 1.

why data sonograms offer the possibility to combine model based sonification with perceptual based mapping by using the vowel timbre space. In Figure 9 you find a screenshot of the GUI from the data-sonogram,



On the left you see the data vocagram as implemented with vowel synthesis. A 2D Slider allows to chose any position within the dataset. Triggering the mouse button releases the virtual shock-wave and plays the sounds of each data point, when they are "hit" by the wave. The different colors indicate the categories of the data-points.

Figure 9: GUI from the data-vocagram.

The three categories from the dataset correspond to the vowels $[a:]$ *blue*, $[o:]$ *black*, $[i:]$ *red*. The position of each datapoint relative to the position of the virtual shock-wave was played back in the stereo panorama.

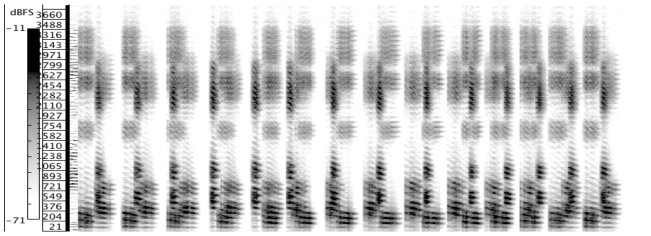


Figure 10: Combined spectrogram of the stereo channels showing the sounds of different triggered shockwaves triggered in the vocagram.

In Figure 10 there is a spectrogram of the sounds of several subsequent explorations of the dataset. The spectrogram shows the different sonic characteristics that correspond to the different data clusters. Distance is additionally mapped to pitch, as the glissandi like movement of the harmonics shows. In most positions within the the dataset where the shockwave is triggered all three data-clusters and their position relative to the epicenter of the shock-wave can be well identified.

7. CONCLUSION

We have presented building blocks for vowel synthesis in the sound synthesis environment SC. As several synthesis examples and two sonification applications have demonstrated, these building blocks allow for an efficient and yet flexible sound design of complex evolutions of timbre by manipulating the spectral envelops. These building-blocks for vowel synthesis are released as SC quarks. We believe that this work provides a flexible interface for the manipulation of the commonly shared sonic experience of a vowel sound and hope it helps to increase interchangeability and collaborative improvements of vowel based sonifications.

8. ACKNOWLEDGMENT

Special thanks to Alberto DeCampo and Julian Rohrhuber for stimulating discussions and ideas.

9. REFERENCES

[1] M. Rath and D. Rocchesso, "Continuous sonic feedback from a rolling ball," *IEEE Interactive Sonification*, 2005.

[2] M. Rath and R. Schleicher, "On the relevance of auditory feedback for quality of control in a balancing task," *ACTA ACUSTICA UNITED WITH ACUSTICA*, vol. 94, pp. 12 – 20, 2008.

[3] M. Lagrange, G. Scavone, and P. Depalle, "Time-domain analysis / synthesis of the excitation signal in a source / filter model of contact sounds," in *Proceedings of the 14th International Conference on Auditory Display*, Paris, France, 2008, inproceedings. [Online]. Available: http://www.icad.org/Proceedings/2008/LagrangeScavone2008.pdf

[4] O. Ben-Tal, J. Berger, B. Cook, M. Daniels, and G. Scavone, "Sonart: The sonification application research toolbox," in *Proceedings of the 8th International Conference on Auditory Display (ICAD2002)*, R. Nakatsu and H. Kawahara, Eds. Kyoto, Japan: Advanced Telecommunications Research Institute (ATR), Kyoto, Japan, July 2-5 2002. [Online]. Available: http://www.icad.org/Proceedings/2002/BenTalBerger2002.pdf

[5] R. J. Cassidy, J. Berger, K. Lee, M. Maggioni, and R. R. Coifman, "Auditory display of hyperspectral colon tissue images using vocal synthesis models," in *Proceedings of the 10th International Conference on Auditory Display (ICAD2004)*, S. Barrass and P. Vickers, Eds., Sydney, Australia, 2004. [Online]. Available: http://www.icad.org/Proceedings/2004/CassidyBerger2004.pdf

[6] T. Hermann, G. Baier, U. Stephani, and H. Ritter, "Vocal sonification of pathologic EEG features," in *Proceedings of the 12th International Conference on Auditory Display*, T. Stockman, Ed., International Community for Auditory Display (ICAD). London, UK: Department of Computer Science, Queen Mary, University of London UK, 06 2006, pp. 158–163.

[7] M. Kleiman-Weiner and J. Berger, “The sound of one arm swinging: A model for multidimensional auditory display of physical motion,” in *Proceedings of the 12th International Conference on Auditory Display (ICAD2006)*, T. Stockman, L. Valgerur Nickerson, C. Frauenberger, A. D. N. Edwards, and D. Brock, Eds. London, UK: Department of Computer Science, Queen Mary, University of London, UK, 2006, pp. 278–280. [Online]. Available: http://www.icad.org/Proceedings/2006/KleimanWeinerBerger2006.pdf

[8] G. Kramer, *Auditory Display: Sonification, Audification, and Auditory Interfaces*. Perseus Publishing, 1993.

[9] T. Hermann and H. Ritter, “Listen to your data: Model-based sonification for data analysis,” in *Advances in intelligent computing and multimedia systems*, G. E. Lasker, Ed. Baden-Baden, Germany: Int. Inst. for Advanced Studies in System research and cybernetics, 08 1999, pp. 189–194.

[10] T. Hermann, G. Baier, U. Stephani, and H. Ritter, “Kernel regression mapping for vocal eeg sonification,” in *Proceedings of the 14th International Conference on Auditory Display*. Paris, France: International Conference on Auditory Display, 2008, inproceedings. [Online]. Available: http://www.icad.org/Proceedings/2008/HermannBaier2008.pdf

[11] A. deCampo, C. Frauenberger, and R. Höldrich, “Designing a generalized sonification environment,” in *Proceedings of 10th Meeting of the International Conference on Auditory Display*. ICAD, 2004.

[12] G. Fant, *Acoustic Theory of Speech Production*. Mouton: The Hague, 1960.

[13] D. Klatt, “Software for a cascade/parallel formant synthesizer,” *Journal of the Acoustical Society of America*, vol. 67, no. 3, pp. 971–995, March 1980.

[14] T. Bovermann, R. Tünnermann, and T. Hermann, “Auditory augmentation,” *International Journal of Ambient Computing and Intelligence (IJACI)*, vol. 2, no. 2, pp. 27–41, 2010.

[15] O. E. Roessler, “An equation for continuous chaos,” *Physics Letters A*, vol. 57, no. 5, pp. 397–398, 1976.