DESIGNING SYNTHETIC REVERBERATORS IN KRONOS

Vesa Norilo

Sibelius Academy
Centre for Music & Technology, Helsinki, Finland
vnorilo@siba.fi

ABSTRACT

Kronos is a special purpose programming language intended for musical signal processing tasks. The central aim is to provide an approachable development environment that produces industrial grade signal processors.

The system is demonstrated here in the context of designing and building synthetic reverberation algorithms. The classic Schroeder-Moorer algorithm is presented, as well as a feedback delay network, built with the abstraction tools afforded by the language. The resulting signal processors are evaluated both subjectively and in raw performance terms.

1. INTRODUCTION

The Kronos package consists of a language specification as well as an optimizing compiler. The compiler can be paired with several back ends. Currently the main focus is on a just in time compiler for the x86 architecture, while a C-language generator is also planned.

Syntactically Kronos is inspired by high level functional languages[1]. The syntax of functional languages is ideally suited for visualization; this match is demonstrated by another source of inspiration, Faust[6]. Eventually, Kronos aims to combine the ease of use and approachability of graphical environments like Pure Data[7] and PWGL[3] with the abstraction and rigour typical of functional programming languages.

Kronos programs are generic, meaning that signal processing blocks can be written once and used in any number of type configurations. For example, a digital filter could be designed without specifying single or double precision sample resolution or even if the data is numerically real or complex, monophonic or multichannel. The type system used in Kronos is more thoroughly discussed in[5].

When a Kronos patch is connected to a typed source, like an audio input, the patch is *specialized*. The generic algorithm description is type inferred[ref]. Following this, the execution is deterministic, which faciliates drastic compiler optimization[ref]. The system could be summarized as a type-driven code generator that produces highly optimized, statically typed code from high level, functional source code.

The rest of this paper is organized as follows. Section 2, *Implementing a Reverberator*, discusses the implementation of various primitives and algorithms in Kronos. Section 3, *Tuning*, discusses additions and enhancements to the basic algorithms. The results are evaluated in Section 4, *Evaluation*, before the paper's *Conclusion*, Section 5.

2. IMPLEMENTING A REVERBERATOR

Reverberation is a good example case, as the algorithms are straightforward yet complicated enough to stress the development environment and provide opportunities for utilization of several language features. They are also well suited for performance benchmarks.

2.1. Primitives for Reverberation

The example cases start with the primitives that are essential to synthetic reverberation. Delay lines, comb filters, and allpass filters will be examined.

2.1.1. Delay

While the functional programming paradigm intuitively matches the signal flow graph widely used for describing signal processing algorithms, there is an apparent clash. Functional programs do not support program *state*, or memory, a fundamental part of any processor with delay or feedback.

The solution to this problem offered by Kronos is a integrated delay operator. Called *rbuf*, short for a ring buffer, the operator receives three parameters: an initializer function, allowing the user to specify the contents of the ring buffer at the start of processing, the size of the buffer or delay time and finally the signal input. In Listing 1, a simple delay function is presented. This delay line is 10 samples long and is initialized to zero at the beginning.

Listing 1. Simple delay

```
Delay(sig)
{
  Delay = rbuf('0 #10 sig)
}
```

2.1.2. Comb filter

A delay line variant with internal feedback, a comb filter, is also widely used in reverberation algorithms. For this configuration, we must define a recursive connection. The *rbuf* operator allows signal recursion. As in all digital systems, some delay is required for the recursion to be finitely computable. A delay line with feedback is shown in Listing 2. In this example, the symbol *output* is used as the recursion point.

Listing 2. Delay with feedback
```
Delay(sig fb delay)
{
  delayed = rbuf('0 delay sig + fb * delayed)
  Delay = delayed
}
```

2.1.3. *Allpass-Comb*

An allpass comb filter is a specially tuned comb filter that has a flat frequency response. An example implementation is shown in Listing 3, similar to the one described by Schroeder[8].

Listing 3. Allpass Comb filter
```
Allpass-Comb(sig fb delay)
{
  delayed = rbuf('0 delay sig - fb * delayed)
  Allpass-Comb = 0.5 * (sig + delayed + fb * delayed)
}
```

2.2. Multi-tap delay

As a precursor to more sophisticated reverberation algorithms, multi-tap delay offers a good showcase for the power of generic programming.

Listing 4. Multi-tap delay
```
Multi-Tap(sig delays)
{
  Use Algorithm
  Multi-Tap = Reduce(Add Map(Curry(Delay sig) delays))
}
```

The processor described in Listing 4 can specialize to feature any number of delay lines. The well known higher order functions *Map* and *Reduce* define the functional language equivalent to a loop[1]. *Map* applies a caller supplied mapping function to all elements of a list. *Reduce* combines the elements of a list using caller-supplied reduction function.

In this example, another higher order function, *Curry*, is used to construct a new mapping function. *Curry* reduces the two argument *Delay* function into an unary function that always receives *sig* as the first argument. Curry is an elementary operator in combinatory logic.

This curried delay is then used as a mapping function to the list of delay line lengths, resulting in a bank of delay lines, all of them being fed by the same signal source. The outputs of the delay lines are finally summed, using *Reduce(Add ...)*. The remarkably short yet highly useful routine is a good example of the power of functional abstraction in Kronos.

A reader familiar with developing real time signal processing code might well be worried that such high level abstraction will adversely affect the performance of the resulting processor. Fortunately this is not the case, as the language is designed around constraints that allow the compiler to simplify all the complexity extremely well. Detailed performance results are shown in Section 4.

2.3. Schroeder Reverberator

Expanding upon the concepts introduced in Section 2.2, the classic diffuse field reverberator described by Schroeder can be implemented. Listing 5 implements the classic Schroeder reverberation[8]. Please refer to Section 4.1 for sound examples.

Listing 5. Classic Schroeder Reverberator
```
Feedback-for-RT60(rt60 delay)
{
  Feedback-for-RT60 = Crt:pow(#0.001 delay / rt60)
}

Basic(sig rt60)
{
  Use Algorithm
  allpass-params = ((0.7 #221) (0.7 #75))
  delay-times = (#1310 #1636 #1813 #1927)

  feedbacks = Map(
    Curry(Feedback-for-RT60 rt60)
    delay-times)

 comb-section = Reduce(Add
      Zip-With(
        Curry(Delay sig)
        feedbacks
        delay-times))

  Basic = Cascade(Allpass-Comb comb-section allpass-
       params)
}
```

All the tuning parameters are adapted from Schroeder's paper[8]. The allpass parameters are constant regardless of reverberation time, while comb filter feedbacks are calculated according to the specified reverberation time. The comb section is produced similarly to the multi tap delay in Section 2.2. Since the delay function requires an extra feedback parameter, we utilize the *Zip-With* function, which is similar to *Map*, but expects a binary function and two argument lists. The combination of *Curry* and *Zip-With* generates a bank of comb filters, all fed by the same signal, but separately configured by the lists of feedback coefficients and delay times.

The series of allpass filters is realized by the higher order *Cascade* function. This function accepts a parameter cascading function, *Allpass-Comb*, signal input, *sig*, and a list of parameters, *allpass-params*. The signal input is passed to the cascading function along with the first element of the parameter list. The function iterates through the remaining parameters in *allpass-params*, passing the output of the previous cascading function along with the parameter element to each subsequent cascading function. Perhaps more easily grasped than explained, this has the effect of connecting several elements in series.

While the same effect could be produced with two nested calls to *Allpass-Comb*, this formulation allows tuning the allpass section by changing, inserting or removing parameters from the *allpass-params* list, with no further code changes, regardless of how many allpass filters are specified.

2.4. Feedback Delay Network Reverberator

Feedback delay network is a more advanced diffuse field simulator, with the beneficial property of reflection density increasing as a function of time, similar to actual acoustic spaces. The central element of a FDN algorithm is the orthogonal feedback matrix, required for discovering the lossless feedback case and understanding the stability criteria of the network. For a detailed discussion of the theory, the reader is referred to literature[2]f.

Listing 6. Basic Feedback Delay Network reverberator

```
Use Algorithm

Feedback-Mtx(input)
{
  Feedback-Mtx = input

  (even odd) = Split(input)
  even-mtx = Recur(even)
  odd-mtx = Recur(odd)

  Feedback-Mtx = Append(Zip-With(Add even-mtx odd-mtx)
          Zip-With(Sub even-mtx odd-mtx))
}

Basic(sig rt60)
{
  delay-times = (#1310 #1636 #1813 #1927)
  normalize-coef = -1. / Sqrt(Count(delay-times))
  loss-coefs = Map(Curry(Mul normalize-coef)
          Map(Curry(Feedback-for-RT60 rt60) delay-
               times))

  feedback-vector = z-1('(0 0 0 0) Zip-With(Mul loss-
       coefs Feedback-Mtx(delay-vector)))

  delay-vector = Zip-With(Delay Map(Curry(Add sig)
       feedback-vector) delay-times)

  Basic = Reduce(Add Rest(delay-vector))
}
```

In Listing 6, functional recursion is utilized to generate a highly optimized orthogonal feedback matrix, the Householder feedback matrix. The function *Feedback-Mtx* recursively calls itself, splitting the signal vector in two, computing element-wise sums and differences. This results in an optimal number of operations required to compute the Householder matrix multiplication[9].

Note that *Feedback-Mtx* has two return values; one of them simply returning the argument *input*. This is a case of parametric polymorphism[5], where the second, specialized form is used for arguments that can be split in two.

The feedback paths in this example are outside the bank of four delay lines. Instead, a simple unit delay recursion is used to pass the four-channel output of the delay lines through the feedback matrix and back into the delay line inputs. Because all the delay lines are fed back into all the others, the feedback must be handled externally.

The final output is produced by summing the outputs of all the delay lines except the first one, hence *Rest(delay-vector)*. The first delay line is skipped due to very prominent modes resulting from the characteristics of the Householder feedback.

3. TUNING

The implementations in Section 2 do not sound very impressive; they are written for clarity. Further tuning and a greater number of delay lines are required for a modern reverberator. The basic principles of tuning these two algorithms are presented in the following Sections 3.1 and 3.2. The full code and sound examples can be accessed on the related web page[4].

3.1. Tuning the Schroeder-Moorer Reverberator

A multichannel reverberator can be created by combining several monophonic elements in parallel with slightly different tuning parameters. Care must be taken to maintain channel balance, as precedence effect may cause the reverberation to be off-balance if the delays on one side are clearly shorter. Reflection density can be improved by increasing the number of comb filters and allpass filters while maintaining the basic parallel-serial composition. Frequency-dependant decay can be modeled by utilizing loss filters on the comb filter feedback path, and overrall reverberation tone can be altered by filtering and equalization.

The tuned example[4] built for this paper features 16 comb filters and 4 allpass filters for both left and right audio channel. Onepole lowpass filters are applied to the comb filter feedback paths and further to statically adjust the tonal color of reverberation.

3.2. Tuning the Feedback Delay Network Reverberator

Likewise, the number of delay lines connected in the feedback network can be increased. Frequency dependent decay is modelled similarily to the Schroeder-Moorer reverberator. Since a single network produces one decorrelated output channel for each delay line in the network, multichannel sound can be derived by constructing several different sums from the network outputs. Allpass filters can be used to further increase sound diffusion.

The tuned example[4] features 16 delay lines connected in a Householder feedback matrix. Each delay line has a lowpass damping filter as well as an allpass filter in the feedback path to improve the overrall sound. A static tone adjustment is performed on the input side of the delay network.

4. EVALUATION

Firstly, the results of implementing synthetic reverberators in Kronos is evaluated. Evaluation is attempted according to three criteria; how good the resulting reverberator sounds, how well suited was the Kronos language to program it and finally, the real time CPU performance characteristics of the resulting processor. The following abbreviations, in Table 1 are used to refer to the various processors described in this paper.

Key	Explanation
S4	Classic Schroeder reverberator, Section 2.3
S16	Tuned Schroeder reverberator [4]
FDN4	4-dimensional Feedback Delay Network, Section 2.4
FDN16	16-dimensional tuned FDN [4]

Table 1. Keys used for the processors

Key	Delays	Allpass	LPFs	Fmt	Time	CPU
S4	4	2	0	mono	6.9ms	0.12%
S16	32	8	33	stereo	89ms	1.5%
FDN4	4	0	0	mono	7.1ms	0.12%
FDN16	16	20	17	stereo	84ms	1.4%

Table 2. Features and performance of the processors

4.1. Sound

Sound quality is highly subjective measure; therefore, the reader is referred to the actual sound examples[ref]. Some observations by the author are listed here.

Unsuprisingly, *S4* is showing its age; the reverberation is rather sparse and exhibits periodicity. The modes of the four comb filters are also spread out enough that they are perceptible as resonances. However, the sound remains respectable for the computational resources it consumes.

FDN4 is also clearly not sufficient by itself. The diffuse tail is quite an improvement over *s-cl*, although some periodicity is still perceived. The main problem is the lack of diffusion in the early tail. This is audible as a sound resembling flutter echo in the very beginning of the reverberation.

S16 and *FDN16* both sound quite satisfying with the added diffusion, mode density and frequency dependant decay. *FDN16* is preferred by the author, as the mid-tail evolution of the diffuse field sounds more convincing and realistic, probably due to the increasing reflection density.

4.2. Language

It is our contention that all reverberation algorithms can be clearly and concisely represented by Kronos. Abstraction is used to avoid manual repetition such as creating all the delay lines one by one. The delay operator inherent in the language allows the use of higher order functions to create banks and arrays of delay lines and filters. In the case of the feedback delay network, a highly effective recursive definition of the Householder feedback matrix could be used.

4.3. Performance

The code produced by Kronos exhibits excellent performance characteristics. Some key features of the reverberators are listed in Table 2, along with the time it took to process the test audio.

A realtime CPU stress is computed by dividing the processing time with the play time of the audio, 5833 milliseconds in this test case. The processor used for the benchmark is an Intel Core i7 running at 2.8GHz. The CPU load caused by the algorithms presented ranges from 1.2 permil to 1.5 percent.

5. CONCLUSION

This paper presented a novel signal processing language and implementations of synthetic reverberation algorithms in it. The algorithms were then tuned and evaluated by both sound quality and performance criteria.

The presented algorithms could be implemented on a high level, utilizing abstractions of functional programming. Nevertheless, the resulting audio processors exhibit excellent performance characteristics.

Kronos is still in development into a versatile tool, to allow real time processing as well as export to languages such as C. Graphical user interface is forthcoming. Kronos is also going to be used as the next-generation synthesizer for the PWGL[3] environment. Interested parties are invited to contact the author should they be interested in implementing their signal processing algorithms in Kronos.

6. REFERENCES

[1] P. Hudak, “Conception, evolution, and application of functional programming languages,” *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, 1989.

[2] A. Jot Jean-Marc; Chaigne, “Digital Delay Networks for Designing Artificial Reverberators,” in *Audio Engineering Society Convention 90*, 1991.

[3] M. Laurson, M. Kuuskankare, and V. Norilo, “An Overview of PWGL, a Visual Programming Environment for Music,” *Computer Music Journal*, vol. 33, no. 1, pp. 19–31, 2009.

[4] V. Norilo, “ICMC2011 Examples,” 2011. [Online]. Available: http://www.vesanorilo.com/kronos/icmc2011

[5] V. Norilo and M. Laurson, “A Method of Generic Programming for High Performance DSP,” in *DAFx-10 Proceedings*, Graz, Austria, 2010, pp. 65–68.

[6] Y. Orlarey, D. Fober, and S. Letz, “Syntactical and semantical aspects of Faust,” *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[7] M. Puckette, “Pure data: another integrated computer music environment,” in *Proceedings of the 1996 International Computer Music Conference*, 1996, pp. 269–272.

[8] M. R. Schroeder, “Digital Simulation of Sound Transmission in Reverberant Spaces,” *Journal of the Acoustical Society of America*, vol. 45, no. 1, p. 303, 1969.

[9] J. O. Smith, “A New Approach to Digital Reverberation Using Closed Waveguide Networks,” 1985, pp. 47–53.