Gamma: A C++ Sound Synthesis Library Further Abstracting the Unit Generator

Lance Putnam
Department of Architecture, Design and Media Technology
Aalborg University
Aalborg, Denmark
lp@create.aau.dk

ABSTRACT

Gamma is a C++ library for sound synthesis that was created to address some of the limitations of existing sound synthesis libraries. The first limitation is that unit generators cannot easily be organized into separate sampling domains. This makes it difficult to use unit generators with different sample rates and in other domains, namely the frequency domain. The second limitation is that certain internal unit generator algorithms, such as interpolation, cannot be customized. This tends to lead to closed architectures consisting of multiple unit generators with only slight algorithmic differences. Gamma makes explicit two novel abstractions—assignable sampling domains and algorithm Strategies—to help overcome these limitations and extend the application range of its unit generators.

1. INTRODUCTION

There currently exist myriad C++ libraries oriented towards real-time sound synthesis. Each is based on the unit generator abstraction [1] in order to allow construction of a large variety of synthesis instruments and effects. Where the libraries differ, however, is in the more specific kinds of generalizations incorporated into the provided unit generators. Gamma is a C++ sound synthesis library that aims to provide a basic set of lightweight, efficient, and, most importantly, flexible unit generators both in terms of how they can be connected and what types of data they can process. Unlike existing libraries, Gamma utilizes both sampling domain and generic programming abstractions to extend the range of applicability of its unit generators. Not only can unit generators run at different rates, but they can also be used in the frequency domain. In addition, unit generators are type generic, and in certain cases, algorithm generic so that they can easily be customized and extended without having to re-implement certain core functionality.

In this paper, we first introduce related work and then discuss the motivation and design principles of Gamma. The next sections discuss two novel abstractions—assignable sampling domains and algorithm Strategies—that are used to increase the range of application of Gamma's unit generators.

2. BACKGROUND

Existing C++ libraries that are oriented towards real-time sound synthesis include CLAM [2], the CREATE Signal Library (CSL) [3], the ICST DSP library [4], IT++ [1], JamomaDSP [5], Marsyas [6], Maximilian [7], Nsound [2], sig++ [3], SndObj [8], SPKit [9], SPUC [4], the Synthesis Toolkit (STK) [10, 11], and UGen++ [12]. We identify at least three main distinctions between the implementations of unit generators in these libraries: (1) processing granularity (single-sample and/or block-based), (2) support for processing generic types, and (3) ability to run at multiple sample rates.

One distinction between the available libraries is their processing granularity, namely, whether the unit generators operate on blocks of samples or process one sample at a time. The advantage of single-sample processing is that it allows arbitrary routing of signals between unit generators making it trival to implement, for instance, loop filters and feedback FM. Approximately half of the libraries identified above use block-based processing, while the others are based on single-sample processing. The block-based processing libraries typically require unit generators to be connected into a graph structure in order to be used. With single-sample processing, unit generators simply contain a method that returns the next sample which obviates the need for a separate graph structure. sig++ and SPKit are exceptions to this, where unit generators are explicitly connected into a graph.

Another distinction that can be made, given that C++ supports generic types through its template mechanism, is whether the unit generators can process generic types. Kronos [13], a descendent of PWGLSynth [14], serves as a good example of generic-type processing in musical DSP albeit it is not a C++ class library. IT++ uses three different generic types for the input samples, output samples, and coefficients of its filters. SPUC also uses generic types for its filters, but only one type, `Numeric`, for both the input and output samples. IT++ and SPUC, however, are mainly oriented towards filtering and more general signal process-

[1] http://itpp.sourceforge.net/devel/
[2] http://nsound.sourceforge.net/
[3] http://sig.sapp.org/
[4] http://spuc.sourceforge.net/

ing tasks rather than (musical) sound synthesis. Mozzi [5] uses generic sample types for several of its unit generators, however, most are specialized for integer types. While several libraries allow generic sample types, none of them permit generic *algorithms* for customizing the unit generators. What is typically seen are suffixes added to unit generator names to designate different behaviors, such as different interpolation policies. Csound/SndObj and Supercollider/UGen++, for example, take this approach.

Synthesis libraries need to have a mechanism for keeping unit generators synchronized with a sampling domain. Synchronization typically occurs according to either: (1) a pull model whereby unit generators simply read a sample rate variable whenever control parameters are updated or (2) a push model whereby unit generators are notified of a change in sample rate. While the pull model is simpler to implement, the push model lends itself better to optimizations involving pre-computing certain intermediate variables, such as phase increment factors. In addition to the push or pull approach, the sample rate is typically either defined globally to be used by all unit generators or defined locally within each unit generator. Defining the sample rate locally permits unit generators to run at multiple sample rates. Maximilian and Ugen++ unit generators read a global sampling rate variable to stay synchronized. This has the advantage of simplicity, but does not allow unit generators to run with multiple sample rates. CSL, NSound, and Marsyas allow the sample rate to be specified locally for each unit generator, thus allowing multiple sample rates. However, the unit generator sample rates must be synchronized manually. In JamomaDSP, sig++, SndObj and STK, the unit generator base classes have a virtual method permitting specific tasks to be executed by unit generators when the sampling rate changes. STK also allows unit generators to ignore notifications of a change in the global sampling rate so they can be used in a multi-rate context.

3. LIBRARY DESIGN

The purpose of this section is to introduce some of the motivation and design decisions underlying Gamma. Since the purpose of this paper is not to introduce the library in detail, it is recommendation that interested readers peruse the available documentation on the Gamma homepage [6].

3.1 Design Motivations

The overall goal of Gamma is to provide an easy-to-use library for constructing complex, yet efficient synthesis instruments and effects that can run on a wide variety of platforms. This goal implies a design that

1. has a standard set of unit generators (oscillators, noise, sample player, envelopes, filters, and variable delays),
2. has a short-time Fourier transform (STFT),
3. performs single-sample processing,
4. supports generic types, and
5. strives for low per-object memory and CPU consumption.

[5] http://sensorium.github.com/Mozzi/
[6] http://www.mat.ucsb.edu/gamma

C++ was desired largely for its zero-overhead rule of "what you don't use, you don't pay for" [15] and for its templates which support generic programming. Generic typing is especially useful for signal processing as many processing algorithms are, at their core, simply algebraic formulations. Single-sample processing was preferred over block-based processing as it makes the least assumptions about how unit generators should be used and keeps control parameter and processing updates separate. Low memory/CPU consumption has obvious performance benefits, but is also seen as an important component of scalability. A well-made library should run efficiently on as many platforms as possible, especially those with limited resources.

At the moment, there are no other sound synthesis libraries satisfying all of these design requirements. The Synthesis Toolkit [10, 11] comes close, but lacks an STFT class and does not support generic types.

3.2 Unit Generators

Unit generators in Gamma are divided into *generators* and *filters*. Generators produce a sequence of samples and filters transform an input sample into an output sample. The basic generators and filters are listed and described in Fig. 1 and Fig. 2, respectively.

Unit generators are implemented as function objects [16]. Function objects are essentially objects with an overloaded function call operator that performs the object's main action. The main action for unit generators is simply to process the next sample. Generators overload the nullary function call operator while filters overload the unary function call operator. For example, the next output of a generator `gen` is obtained by calling `gen()` and the next output of a filter `flt` is obtained by calling `flt(x)` where `x` is the input.

4. PROCESSING ABSTRACTIONS

Gamma provides two primary abstractions that greatly extend the range of application of its provided unit generators. The first of these is the use of generics for unit generator sample and parameter types and processing algorithms. The second abstraction is assignable sampling domains where unit generators can operate under arbitrarily defined one-dimensional sampling domains.

4.1 Generic Types

Generic types are used to increase the versatility of generators and filters without needing to change their underlying algorithm. Gamma uses C++ templates to allow concrete classes to be made according to generic types. The advantage of this approach over, for example, macros or typedefs, is that the library can easily accommodate different sample types in application code without needing to resort to multiple explicit compilations. This makes it easy to define

`Accum`	Phase accumulator/timer
`Osc`	Wavetable oscillator
`LFO`	Non-band-limited oscillator
`Sine`	Sine wave
`SineR(s)`	Sine resonance (bank)
`SineD(s)`	Damped sine resonance (bank)
`CSine`	(Damped) complex sinusoid
`DSF`	Discrete summation formula
`Impulse`	Band-limited impulse train
`Saw`	Band-limited saw wave
`Square`	Band-limited square wave
`SamplePlayer`	Sample/sound file player
`NoiseWhite`	White noise
`NoisePink`	Pink noise
`NoiseBrown`	Brown noise
`Env`	N-segment exponential envelope
`Decay`	Exponential decay
`Seg`	Interpolated segment

Figure 1. Generator classes.

processors having different precision within the same application. For example, single- and double-precision one-pole filters can be declared as:

```
OnePole<float> opf;
OnePole<double> opd;
```

Beyond permitting different precision types, unit generators can also operate on non-scalar types, such as complex numbers and vectors. For example, it is often necessary to apply the same filter to a stereo signal. Ideally, only one set of filter coefficients should be used to save memory and eliminate duplicate effort in computing the coefficients from parametric controls. A one-pole filter that processes a 2-vector using the provided n-vector class, `Vec`, is declared as:

```
OnePole<Vec<2,float> > op2;
```

For convenience, Gamma provides 2-vector `float2` and `double2` types, so the previous example can be written

```
OnePole<float2> op2;
```

4.2 Strategies

One can broaden the scope of generics beyond types to also include algorithms. In the parlance of design patterns, a *Strategy* is an object that represents an algorithm [17]. Strategies are light-weight function objects, typically having little or no data, that conform to an identical interface,

`OnePole`	1-pole filter
`AllPass1`	1st-order allpass
`AllPass2`	2nd-order allpass
`Biquad`	2-pole, 2-zero filter
`Notch`	2-zero notch
`Reson`	2-pole resonator
`BlockDC`	DC blocker
`Integrator`	Leaky integrator
`DelayShift`	Fixed n-sample delay
`Delay`	Variable length delay
`Comb`	Comb filter/feedback delay
`Multitap`	Multitap delay
`Hilbert`	Hilbert transformer

Figure 2. Filter classes.

yet behave differently. Strategies permit certain behaviors of a class to be swapped out or customized without having to define a new class.

In Gamma, Strategies are employed for two main purposes—to reduce the number of base unit generator types and to permit unit generators to be extended more easily than by subclassing. For example, Listing 1 shows how Strategies are used to declare different types of a wavetable oscillator class. The Strategies used in Gamma are compile-time rather than run-time so that they can be efficiently inlined. Two main Strategies are utilized—interpolation and phase increment.

```
// Oscillator with truncating interpolation
Osc<float, ipl::Trunc, phsInc::Loop>

// Oscillator with linear interpolation
Osc<float, ipl::Linear, phsInc::Loop>

// One-shot with linear interpolation
Osc<float, ipl::Linear, phsInc::OneShot>

// Ping-pong oscillator with cubic
    interpolation
Osc<float, ipl::Cubic, phsInc::PingPong>
```

Listing 1. Different oscillator types based on combinations of interpolation and phase increment Strategies.

Interpolation Strategies are used to specify the interpolation method used in delay lines, table-based oscillators, and envelope segments. Two types of interpolation Strategies are present in Gamma: random-access and sequence. Random-access interpolation Strategies are used for interpolating values at arbitrary positions along an array. Sequence interpolation Strategies are for interpolating a stream of sample points.

The currently provided random-access interpolation Strategies are `Trunc`, `Round`, `Linear`, `Cubic`, and `AllPass`. The `Switchable` Strategy allows switching between any of the aforementioned interpolation types at

```
namespace ipl{

// Truncating interpolation strategy
template <class T>
class Trunc{
public:
    T operator()(
        const T * src, int size,
        int iInt, double iFrac) const
    {
        return src[iInt];
    }
};

// Linear interpolation strategy
template <class T>
class Linear{
public:
    T operator()(
        const T * src, int size,
        int iInt, double iFrac) const
    {
        return src[iInt] +
            (src[(iInt+1)%size] -
            src[iInt])*iFrac;
    }
};
}

// Wavetable with interpolation strategy
template <int N, class T, class InterpStrat>
class Wavetable{
public:
    T read(double index) const {
        unsigned i = int(index);
        double f = index - i;
        return mInterpStrat(mTable,N, i,f);
    }

private:
    T mTable[N];
    InterpStrat<T> mInterpStrat;
};

// Declare table w/ linear interpolation
Wavetable<1024,float, ipl::Linear> tableL;

// Declare table w/ truncating interpolation
Wavetable<1024,float, ipl::Trunc> tableT;
```

Listing 2. Example interpolation Strategy class definitions and usage with a Wavetable class.

run-time. Listing 2 gives example class definitions for truncating and linear interpolation Strategies and their usage with a `Wavetable` class. Each interpolation Strategy shares the same function operator prototype to access an array. (In practice, there could be many such function operators for specific types of array access.) The `Wavetable` class takes an interpolation Strategy as a template parameter and then creates a member of that type. In `Wavetable::read`, the interpolation Strategy's overloaded function operator is called to compute the interpolated value.

Sequence interpolation Strategies maintain a small FIFO buffer of samples from which an interpolated value can be computed using a specific interpolation method. At the moment, `Trunc`, `Linear`, `Cubic`, and `Cosine` sequence interpolation Strategies are provided. For example, the `Linear` sequence interpolation Strategy operates as follows:

```
iplSeq::Linear lerp;
lerp.push( 0);
lerp.push(20);  // sample points are 0, 20
lerp(0.5);      // returns 10
lerp(0.1);      // returns 2
lerp.push(40);  // sample points are 20, 40
lerp(0.5);      // returns 30
```

The `Seg` unit generator utilizes a sequence interpolation Strategy to create an envelope between two sample points. The basic operation of `Seg` is to interpolate between two values over some specified length and then hold the end value indefinitely. In this way, it can be used to smooth low sample rate synchronous or asynchronous signals. Another mode of operation allows periodic generation of segments in a process similar to upsampling. This is accomplished through an overloaded function call operator that takes a function object as an argument. Whenever the end of the segment is reached, it requests the passed-in function object to generate its next sample, pushes this onto the segment's internal FIFO buffer, and starts the segment over. This effectively starts a new segment that is piece-wise continuous with the old one. Perhaps one of the most useful applications of this mode of operation is producing low-frequency signals from stochastic, non-linear, or other sequence generators. Listing 3 demonstrates how the `NoisePink` and `Seg` unit generators can be used together to create low-frequency, cubic-interpolated pink noise.

```
// SETUP
// Pink noise generator
NoisePink<> noise;

// Interpolated segment running at 10 Hz
Seg<float, iplSeq::Cubic> seg(1./10);

void audioCallback(...){
    for(int i=0; i<blockSize; i++){
        float s = seg(noise);
    }
}
```

Listing 3. Low-frequency, cubic-interpolated pink noise built from the `NoisePink` and `Seg` unit generators.

In the example above, the noise object (not a noise sample) is passed as an argument to the segment's function operator in the sample loop. The segment's function operator will generate the noise's next sample and update the segment endpoints on the condition that the end of the segment has been reached. Otherwise, the next interpolated sample between the existing endpoints is returned.

Phase increment Strategies include `Loop`, `OneShot`, `NShot`, `PingPong`, and `Rhythm` (Fig. 3). `Loop` repeatedly cycles the phase, like a typical phase accumulator driving an oscillator. `OneShot` cycles the phase once and then holds its end value. In this way, it can be used for one-shot playback, such as with sample playback and table-based envelopes. `PingPong` is a bidirectional loop that alternates the phase forward and backward. `NShot` and `Rhythm` are slightly more complex Strategies that permit specific kinds of phase patterns. `NShot` is like `OneShot`, except cycles the phase a specified number of times. `Rhythm` repeatedly cycles or holds the phase for one period according to a binary pattern of up to 32 bits. If the bit is 1, then the phase wraps. If the bit is 0, then the phase holds its position for one period.

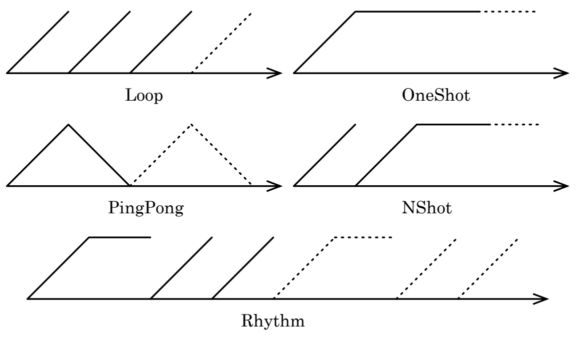

Figure 3. Phase increment Strategies. The dashed lines indicate the Strategy's long-term (repeating) pattern. The `Rhythm` Strategy has the pattern string "`/.//`".

`Rhythm` allows complex rhythmic patterns to be produced at both audio and sub-audio rates, as with pulsar synthesis using burst masking [18]. `Rhythm` patterns can be specified using a 32-bit unsigned integer where the most significant bit is the start of the pattern or as a C-style character string. Character strings follow the convention of '`.`' indicating off and '`/`' indicating on in a similar fashion to GROOVE [19]. For example, we can use `Rhythm` to apply a rhythmic envelope to an oscillator (Listing 4).

```
// SETUP
Osc<> src(400);
LFO<phsInc::Rhythm> env(8);
env.phsInc().pattern("/../../.");

void audioCallback(...){
    for(int i=0; i<blockSize; i++){
        float s = src() * env.downU();
    }
}
```

Listing 4. Using the `Rhythm` Strategy to apply a rhythmic envelope to an oscillator source.

4.3 Assignable Sampling Domains

Perhaps the most novel abstraction of Gamma is *assignable sampling domains*, a way to dynamically assign unit generators to a particular sampling domain. The original motivation behind this design was to make it easy to run unit generators in both the time domain and frequency domain. Of course, it also allows unit generators to be configured to run at various rates, such as audio or block rate or some windowed analysis rate.
The abstraction utilizes an Observer pattern [17] so that groups of unit generators can be notified whenever their associated sampling interval changes. There are two main classes involved with assignable sampling domains, `Domain` and `DomainObserver`, which are the subject and observer, respectively, of the Observer pattern. By default, all unit generators inherit from `DomainObserver`. A `DomainObserver` attaches itself to a `Domain` so that it is notified whenever the sampling interval changes. The overloaded << operator is used to attach a `DomainObserver` to a `Domain`. The following illustrates this

```
DomainObserver obs;
Domain dom;
dom << obs;
```

It is possible to instantiate more than one `Domain` so that multiple sampling intervals can be used within a single system. `DomainObserver`s can attach to any `Domain`, but always have exactly one `Domain`.

Most of the time, unit generators will only need to observe a single sampling rate. For convenience, a default `Domain` called `master` is supplied. All `DomainObserver`s are automatically attached to `master` when constructed. The `master` domain is initialized with a sample rate/interval of 1. To set it to a specific sample rate, say 44.1 kHz, one calls

```
Domain::master().spu(44100);
```

where `spu` stands for samples per unit. A slightly more complex situation involves unit generators running at both sample and control rate. For this, an additional control-rate domain can be utilized. Listing 5 illustrates how one could implement a vibrato effect operating at block rate.

```
// SETUP
Domain::master().spu(44100.);
Domain blockDomain(44100./blockSize);
Sine<> mod(5);
Sine<> car;

// Attach modulator to block domain
blockDomain << mod;

void audioCallback(...){

    car.freq( mod()*5 + 440 );

    // SAMPLE LOOP
    for(int i=0; i<blockSize; i++){
        float s = car();
    }
}
```

Listing 5. Control-rate vibrato implemented using a block-rate time domain.

A perhaps more interesting use of assignable domains is configuring unit generators to operate in frequency domain. For example, an oscillator or a break-point envelope can be used as a magnitude envelope. Listing 6 demonstrates how one can create a barber-pole combing effect using an STFT and two sine oscillators.

All unit generators have as their last template parameter a domain class which is inherited by the unit generator. The default domain class is `DomainObserver`. A special type of domain, `Domain1`, can be used for unit generators that function entirely with normalized frequencies in the interval $[0, 1]$. `Domain1` has the advantage that it does not consume memory or need to do unit conversions since both its sampling frequency and sampling interval are fixed at 1. This is also especially useful for composite objects where unit conversions from a particular domain may only need to be done once by the composing object.

```
// SETUP
Domain::master().spu(44100.);

STFT stft;
Sine<> env(1/100.);
Sine<> envPhase(1);

stft.domainFreq() << env;
stft.domainHop() << envPhase;

void audioCallback(...){

    // TIME SAMPLE LOOP
    for(int i=0; i<blockSize; i++){

        float s = ...; // current sample

        // Check if next spectral frame is
            ready...
        if(stft(s)){
            env.phase(envPhase()*0.5 + 0.5);
            int N = stft.numBins();

            // FREQUENCY SAMPLE LOOP
            for(int k=0; k<N; ++k){
                stft.bin(k) *= env();
            }
        }

        // Resynthesis
        s = stft();
    }
}
```

Listing 6. Barber-pole combing effect using hop- and frequency-domain oscillators.

5. CONCLUSION

Gamma attempts to maximize the flexibility of its supplied unit generators by utilizing single-sample processing, generic types and algorithms, and assignable sampling domains. Single-sample processing has proven to be very flexible and efficient if one is satisfied with static unit generator graphs. Generic types and algorithms add more complexity to the library, but it seems to be a reasonable trade-off as they bring a whole new dimension of code reuse and extensibility, which are generally considered good. Assignable sampling domains make it easy to manage unit generators running at different rates. By allowing standard unit generators such as oscillators and envelopes to operate in the frequency domain many possibilities for new and exotic effects emerge. It remains to be seen if LCCD filters, such as biquads, have any meaningful applications in the frequency domain. One unique attribute of the frequency-domain is that it is non-causal, unlike the time-domain, and thus IIR filters can be made linear phase through bidirectional filtering. Instead of filtering across frequency, one could filter the temporal trajectories of individual bin magnitudes to produce spectral blurring and other effects. This would require filters to efficiently handle arrays as sample types, something not handled in Gamma at the moment.

6. REFERENCES

[1] M. Mathews, *The Technology of Computer Music*. Boston: The M.I.T. Press, 1969.

[2] X. Amatriain, "An object-oriented metamodel for digital signal processing with a focus on audio and music," Ph.D. dissertation, Universitat Pompeu Fabra, Barcelona, Spain, 2005.

[3] S. Pope, X. Amatriain, L. Putnam, J. Castellanos, and R. Avery, "Metamodels and design patterns in CSL4," in *Proceedings of the 2006 International Computer Music Conference*, 2006.

[4] S. Papetti, "The icst dsp library: A versatile and efficient toolset for audio processing and analysis applications," in *Proceedings of the 9th Sound and Music Computing Conference*, 2012, pp. 535–540.

[5] T. Place, T. Lossius, and N. Peters, "A flexible and dynamic c++ framework and library for digital audio signal processing," in *Proceedings of the 2010 International Computer Music Conference*, 2010, pp. 157–164.

[6] S. Bray and G. Tzanetakis, "Implicit patching for dataflow-based audio analysis and synthesis," in *Proceedings of the 2005 International Computer Music Conference*, 2005.

[7] M. Grierson, "Maximilian: A cross platform c++ audio synthesis library for artists learning to program," in *Proceedings of the International Computer Music Conference*, New York, 2010.

[8] V. Lazzarini, "Sound processing with the SndObj library: An overview," in *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, 2001.

[9] K. Lassfolk, "Sound processing kit: An object-oriented signal processing framework," in *Proceedings of the 1999 International Computer Music Conference*, 1999, pp. 422–424.

[10] P. R. Cook and G. P. Scavone, "The synthesis toolkit (STK)," in *Proceedings of the 1999 International Computer Music Conference*, 1999.

[11] G. P. Scavone and P. R. Cook, "RtMidi, RtAudio, and a Synthesis Toolkit (STK) update," in *Proceedings of the 2005 International Computer Music Conference*, 2005.

[12] M. Robinson, "Ugen++ —an audio library: Teaching game audio design and programming," in *Proceedings of the AES 41st Conference Audio for Games*, London, UK, 2011.

[13] V. Norilo, "Introducing kronos: A novel approach to signal processing languages," in *Proceedings of the Linux Audio Conference*, 2011, pp. 9–16.

[14] M. Laurson, M. Kuuskankare, and V. Norilo, "An overview of pwgl, a visual programming environment for music," *Computer Music Journal*, vol. 33, no. 1, 2009.

[15] B. Stroustrup, "Evolving a language in and for the real world: C++ 1991-2006," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, 2007.

[16] A. Freed and A. Chaudhary, “Music programming with the new features of standard C++,” in *Proceedings of the 1998 International Computer Music Conference*, 1998, pp. 244–247.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[18] C. Roads, *Microsound*. MIT Press, 2001.

[19] M. Mathews, “GROOVE—a program to compose, store, and edit functions of time,” *Communications of the ACM*, vol. 13, no. 12, 1970.