Nyquist: A Language for Composition and Sound Synthesis[i]

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Nyquist is an interactive language for music composition and sound synthesis. Features of Nyquist include: (1) a full interactive environment based on Lisp, (2) no distinction between the "score" and the "orchestra," (3) support for behavioral abstraction, (4) the ability to work both in terms of actual and perceptual start and stop times, and (5) a time- and memory-efficient implementation.

Introduction

Signal processing, including synthesis, is an important component of any digital audio system. Often, however, signal processing is offered with little or no additional support. the goal of Nyquist is to provide an open-ended programing langauge that supports high-level compositional tasks in addition to low-level signal processing. One of the key advantages of Nyquist is the integration of signal processing, control, and temporal structure within a general purpose programming language.

The programming language Lisp provides an interactive interface, flexibility in manipulating sounds, and a base for performing other related symbolic processing. Sounds are first-class types in Nyquist, hence they can be assigned to variables, passed as parameters, and stored in data structures. Storage for sounds is dynamically allocated as needed and reclaimed by automatic garbage collection (Schorr and Waite 1967). This feature allows "instruments" to be implemented as ordinary Lisp functions and eliminates the orchestra/score dichotomy found in other systems.

Nyquist semantics include behavioral abstraction as introduced by Arctic (Dannenberg 1984, Dannenberg, McAvinney, and Rubine 1986) and Canon (Dannenberg 1989). The motivation for behavioral abstraction is the idea that one should be able to describe behaviors that respond appropriately to their environment. For example, stretching a sound may mean one thing in the context of granular synthesis and another in the context of sampling. It almost never means to compute a short sound and then resample it to make it longer. Nyquist allows the programmer to describe abstract behaviors that "know" how to stretch, transpose, change loudness, and shift in time. Transformation operators are provided to operate on these abstractions.

---

[i] Published as: Roger B. Dannenberg. 1997. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis," *Computer Music Journal*, 21(3):50-60.

Composition requires that sounds be placed simultaneously, in sequence, and at arbitrary offsets. Because musical sounds often have attack and release portions, we make a distinction between the absolute first and last samples of a sound and the perceptual start and end to which other sounds should be aligned.

Functions that combine sounds include:

`(at t s)`	shifts sound `s` by `t` seconds in time.
`(seq s1 s2 s3 ...)`	arranges each sound sequentially in time.
`(sim s1 s2 s3 ...)`	arranges each sound simultaneously in time.

By making sounds an integral part of the language and including time-based control constructs, a remarkably powerful notation is obtained. In this article, I hope to show that the Nyquist approach leads to a versatile, expressive language for sound analysis, manipulation, and composition. This overview describes the general approach, main features, and applications of the language. Following articles will cover specific aspects in greater detail, including time warps and continuous transformations (Dannenberg, to appear A), the implementation (Dannenberg, to appear B), and techniques for efficient signal processing (Dannenberg and Thompson, to appear).

Related Work

Many software synthesis and compositional systems have been created, each encountering and addressing a slightly different set of problems. To understand Nyquist, it is beneficial to first review some other systems.

Music N, which refers to Music V (Mathews 1969) and it descendants (Music-11, Csound, cmusic, etc.) (Vercoe 1981, 1986, and Moore 1982) takes a programming language approach to sound *generation* in that unit generators can be combined as functions applied to sample streams. The resulting instruments can be applied to parameter lists in the score. However, instruments cannot be applied to other instruments, nor can scores be constructed hierarchically. Furthermore, even though these languages are conceptually quite similar to the functional style of Nyquist, they adopt a Fortran or assembly language syntax and even require the composer to properly order unit generators and allocate buffers so that samples are generated into buffers before they are read by other unit generators. The division between sound manipulators (or generators) and parameter lists (note statements) results in a corresponding separation between the orchestra and the score. Other consequences include a non-interactive environment. An interesting aspect of Music V is the idea that an *instance* of an instrument is created for each note specified in the score.

Kyma (Scaletti and Johnson 1988, Scaletti 1989) and the Sun/Mercury Workstation (Rodet and Eckel 1988) take a different approach, treating sound manipulators and sound generators as objects that can be "patched" together. This results in an intuitive system for synthesis, but there are problems. A level of indirection is required to manipulate graphs of unit generators which in turn manipulate sounds rather than to manipulate sounds directly. Various extensions have become necessary in order to handle graphs that change over time, but this also adds complexity to programs. Symbol processing and working with data

structures are also difficult with these graph-oriented program representations. Both systems run all objects in synchrony, thereby assuming a global sample rate.

SRL (Kopec 1985) is a signal processing language that represents signals as parameterized computations. SRL signals are immutable objects that can be reused. SRL supports lazy evaluation and function caching by retaining a symbolic representation of all signals. SRL lacks many musically useful concepts, such as behavioral abstraction and the starting time and duration of signals. Also the user must explicitly free buffers when they are no longer needed.

Groove (Mathews and Moore 1970) was an early system for creating and manipulating control functions. Its idea that any time-varying input or output can be represented and manipulated as a function is also very important in Nyquist.

Formes (Cointe and Rodet 1984, Rodet and Cointe 1984) takes an object-oriented approach to the computation of functions of time, but Formes was not designed to compute audio directly. Formes was originally designed to compute control information for the Chant (Rodet, *et al.* 1984) synthesis system.

Boynton's Scheme-based music language (Boynton 1992) uses functional programming to generate MIDI data. The semantics are virtually identical to Canon, except that transformations are discrete rather than continuous, and the system runs in real-time. The real-time implementation allows communication and synchronization between sequentially executing processes, a feature that is not found in Canon or Nyquist.

CLM (Common Lisp Music) (Schottstaedt 1994), like Nyquist, is based on Lisp and inherits many ideas from Music V, but there are significant structural differences between CLM and Nyquist. CLM has separate notions of *score* and *instrument*. Instruments write their outputs to a file rather than returning them as values. To pass output to further processing such as reverberation, a separate pass over the output stream is made. (Nyquist's approach is described in the next section.) CLM instruments are composed from unit generators, but within each instrument definition, the composer must first create and initialize unit generator objects, then reference them in a loop that generates samples. Although there may be good reasons to expose these implementation details, this is a lower-level approach than even Music V's.

Nyquist has a close affinity to some systems intended to support expressive control of tempo and other parameters (Anderson and Kuivila 1990, Desain and Honing 1992a, 1993). Since time warping and continuous transformations are the topic of a forthcoming article (Dannenberg, to appear A), we will defer this discussion.

Finally, signals in Nyquist have an interesting similarity to sequences of values in Lucid (Ashcroft and Wadge 1977), another functional language with an implementation based on lazy evaluation. Lucid has no explicit temporal semantics, however.

A Brief Tutorial

Nyquist is based on functional programming. The composer combines expressions to denote sounds. Expressions are written using Lisp (Touretzky 1984) syntax, where parentheses denote applying a function to a set of parameters. For

example, the expression

```
(osc c4)
```

means "apply the function `osc` to the parameter `c4`." Parameters are evaluated first. In this case, `c4` is a global variable (intended to remain constant) containing the value 60.0. The value 60.0 is passed to the `osc` function, and the result is a sinusoid with a frequency corresponding to middle C with a duration of 1 second. In Nyquist, fractional pitch values are used to work with microtones and non-equal temperaments: 60.01 is one cent sharper than 60.0.

Expressions can be combined. The following expression computes a 6Hz sinusoid:

```
(lfo 6).
```

This can be scaled to a different amplitude:

```
(scale 20 (lfo 6))
```

and used to provide frequency modulation to an FM oscillator:

```
(fmosc c4 (scale 20 (lfo 6)))
```

To hear this sound, we use the play command:

```
(play (fmosc c4 (scale 20 (lfo 6))))
```

The `play` command is a special function in the sense that it has the side-effect of storing the sound in a file and invoking a system command to play the sound file. Here, functions are nested 4 deep. The normal order of evaluation is from inner-most expression to outer-most, although there are exceptions.

At some point, nested expressions start to become unwieldy, and it is best to define new functions to factor the complexity into manageable units. Function definition in Nyquist is exactly as in Lisp. Since Nyquist expressions can return sounds, Lisp functions can play the role of instruments. Here is an example of a simple instrument based on an FM oscillator:

```
(defun fminst (pitch depth)
   (mult (env 0.01 0.2 0.1 1 0.3 0.2)
         (fmosc pitch (scale depth (osc pitch)))))
```

The defun (for "define function") operation is followed by the name of the new function "`fminst`", a parameter list "`(pitch depth),`" and the body of the function. The body uses `mult` to multiply two signals sample-by-sample: an envelope generated by `env` and a tone generated by `fmosc`.

As always in Lisp, `defun` is a special function that does not evaluate its parameters (name, parameter list, and body). Also note that the parameter list is just a list, not an invocation of a function.

Once "instruments" are defined, they can be used to create sounds. Instruments can be tested interactively by playing a single instance:

```
(play (fminst g4 100))
```

If function definition is analogous to a Music N instrument definition, then function application is analogous to a Music N note statement. With Music N, instruments include in their definitions a computation that sums their samples to a sample stream that is typically written to a file. In Nyquist, however, samples are simply returned from the function as a value of type `SOUND`.

To sum (mix) these sounds, Nyquist provides a number of control constructs. The simplest is sim, short for "simultaneous." The following computation:

```
(sim (fminst g4 100)
     (fminst b4 100))
```

sums two simultaneous notes. Notes can be shifted in time using `at`:

```
(sim (at 0.0 (fminst g4 100))
     (at 0.5 (fminst b4 100)))
```

This is beginning to look something like a Music N score: a list of notes, each with a starting time and a list of parameters. Each note generates a sound and the results are all summed. Note however, that while Music N requires a special syntax for the score, and Music N scores always write samples to a specific sample stream, Nyquist requires no special syntax, and the result of a Nyquist "score" is just a value of type `SOUND`. This value can be written to a file, passed through a filter, summed to other values, used as a wavetable, *etc.*

A common problem with Music N is to add some effect such as reverberation to certain notes in a score. The typical solution is to modify instruments to sum their output to a special global variable, which then serves as input to a reverberator. A reverberator "instrument" must be invoked from the score as if it is a note. In contrast, the solution in Nyquist is to merely apply reverberation to the sound returned by a note or a score:

```
(reverb (sim (fminst g4 100)
             (fminst b4 100)))
```

Furthermore, since "scores" are part of the unified language, one can write functions that perform scores:

```
(defun phrase1 (depth)
   (sim (at 0.0 (fminst g4 depth))
        (at 0.5 (fminst b4 depth))))
```

This definition gives a name, `phrase1`, to the two-note sequence. Notice the depth parameter is passed to each note instance. Now, `phrase1` is a reusable shorthand that can be used to create a "score":

```
(sim (at 0.0 (phrase1 100)
     (at 2.0 (phrase1 200))))
```

The process of constructing phrases and sections hierarchically, using parameters to obtain variations, and applying transformations for further control creates a powerful notation for composition.

Temporal Control Constructs

Lisp has no notion of time, but Nyquist extends Lisp with temporal control constructs. Two common ones are `sim`, which is described above, and `seq`, which combines sound behaviors sequentially. The precise semantics are discussed below. For now, we present a simple example of two phrases in sequence:

```
(seq (phrase1 100) (phrase2 100))
```

There are also constructs that iterate behaviors either in parallel or in sequence. Figure 1 shows a simple instrument that sums sinusoidal partials. Although a small example, this illustrates several strengths of Nyquist. First, Nyquist is dynamic; the number of partials is a parameter to the instrument (see add-inst in Figure 1.) Second, Nyquist is a general-purpose language; when we need to compute the chromatic pitch step corresponding to the n^{th} harmonic, we can simply define a new

function (nth-partial) to compute it. Third, Nyquist supports hierarchical definitions; to make this example simpler and modular, envelope generation is defined as separate function. All of this is accomplished in 8 lines of code (not counting comments).

```
;Function to compute pitch for Nth harmonic
(defun nth-harmonic (n)
   (hz-to-step (* n (step-to-hz n))))

;Function to compute partial envelopes
; Higher harmonics get longer rise times
; (pwl is the "piece-wise linear" generator)
(defun nth-env (n)
   ; at n*5ms, rise to 1, then decay to 0 at time 1
   (pwl (* n 0.005) 1 1))

;Additive synthesis instrument, uses partial which
; takes a pitch and an envelope and returns a
; sinusoidal partial.
(defun add-inst (pitch npartials)
   ;iterates body npartials times and return sum:
   (simrep (i npartials)
      ;i ranges from 0 to npartials - 1, so add 1:
      (partial (nth-harmonic (+ i 1))
               (nth-env (+ i 1)))))
```

Figure 1. An additive synthesis instrument definition with a variable number of partials.

Transformations

Nyquist supports a variety of transformations that make it convenient to modify the time, duration, overlap, loudness, pitch, and other attributes of sounds in a composition. For example, to change the loudness of a passage, we can use the loud transform:

```
(loud 10 (phrase1 50))
```

One might expect the loud function to denote a signal multiplication operation, but in fact the loudness parameter is interpreted by `phrase1`, as we shall see in the next section.

Transformations may be nested. In the next example of two phrases, the second phrase is transposed by a total of 13 semitones, subjected to a `loud` transform of 10, and implicitly shifted in time to the end of the first instance of `phrase1`:

```
(transpose 12
   (loud 10
      (seq (phrase1 50)
```

```
(transpose 1 (phrase1 100)))))
```

Transformations can specify functions of time as well as scalar values. These are especially useful for smoothly changing parameters such as loudness during a *crescendo*. The semantics of continuous transformations are borrowed from Canon (Dannenberg 1989). Continuous transformations are interesting in Nyquist because the full power of the language can be used to "synthesize" control functions, which are no different from audio and control signals.

In addition to the features illustrated by these few examples, Nyquist includes a library of many signal-processing and generating functions, sound file support for many popular formats, and the ability to use MIDI files as a source of note and control information. Nyquist is compatible with score files from the CMU MIDI Toolkit (Dannenberg 1986, 1993).

Behavioral Abstraction

The Nyquist approach offers a declarative style that is comparable to note lists while offering the power of a full programming language. Note lists of classical score languages are attractive because they can be generated, stored, and manipulated as data. For example, making all the notes in a section louder is easy to do if the notes are represented as data. On the other hand, note lists suffer from the fact that they are not programs. In particular, there comes a time when "loudness" (and every other note-list parameter) must be interpreted to produce or control sound. The point at which interpretation starts defines the boundary between the "score" and the "orchestra."

An alternative is to eliminate note lists and write programs instead. This approach is attractive because it eliminates the distinction between score and orchestra, and releases any constraints on what can be expressed in the score. Higher-level concepts such as "trill" or "glissando" can then be defined through programming. However, the ability to conveniently manipulate scores is often lost, because it is much easier to manipulate data than it is to manipulate programs.

Nyquist addresses this problem with declarative-style programs that "feel" like note lists and by using the same language to define both scores and synthesis procedures. It is possible to alter Nyquist scores by applying various transformations along the dimensions of time, loudness, pitch, articulation, and even sample rate.

A potential liability of these transformations is that they may transform the wrong thing. For example, in stretching a section of music that contains a trill, we do not necessarily want the trill to slow down, and we almost certainly do not want the pitch to drop! Nyquist provides defaults for transformations, but allows the programmer/composer to override the defaults with more appropriate behaviors.

Thus, the programmer/composer defines behaviors that "do the right thing" in the context of a specified set of transformations. A class of behaviors that are realized according to a context is called *behavioral abstraction*. A few examples should clarify how Nyquist works. Consider a sequence of three sounds:

```
(seq (s-read "a.snd") (grains) (osc Bf3)),
```

where `s-read` is a behavior that simply reads a sound from a file, `grains` is a composer-defined function that implements granular synthesis, and `osc` is a

behavior that plays a given pitch. If we wanted to hear the same sequence at a lower amplitude, we could write:

```
(loud -6 (seq (s-read "a.snd") (grains) (osc Bf3))).
```

The loudness transformation requests the sound to be 6dB softer, but what exactly does "softer" mean? In the case of the built-in functions `s-read` and `osc`, the resulting sounds are scaled by 6dB. However, the `grains` behavior is free to implement a more perceptual notion of "softer," for example generating sounds with less high-frequency content.

Now suppose we wish to change the pitch. We could write

```
(transpose 3 (seq (s-read "a.snd") (grains) (osc Bf3)))
```

This would have the effect of transposing the sequence up by 3 semitones. However, the `s-read` abstraction overrides transposition and prevents alteration of samples, the `grains` behavior implements transposition in a manner specified by the composer (perhaps higher-pitched grains have a different timbre), and the `osc` behavior will be transposed in the expected manner.

How is this accomplished? It was noted earlier that the `loud` transform is not simply a multiplication, and the reason should now be clear: if the meaning of a transformation is to be defined internal to the behavioral abstraction, then a simple external operation such as multiplication or resampling is not sufficient. Somehow, the desired transformation must be communicated to the point where samples are generated.

One way to achieve this goal is to pass transformation information as explicit parameters of every operation. This is very unwieldy because there are many transformations. Users would never tolerate having to extend every parameter list with half a dozen transformation parameters.

The solution is to make transformation parameters implicit. *Every* function, in addition to the explicitly declared parameters, accepts a set of implicit parameters called the *environment*, which includes all the transformation parameters. If the programmer does nothing to override the defaults, then the implicit environment parameters are passed unchanged to nested functions. Therefore, when we write

```
(transpose 5 (seq (osc c4) (osc d4)))
```

the transposition amount is passed implicitly to both of the nested `osc` functions.

Since the environment parameters are implicit, there must be some way to access their values. The parameters are bound to symbols such as `*loud*`, `*warp*`, and `*transpose*`, and it is recommended to access the environment through calls to built-in functions such as `get-loud`, `get-warp`, `get-transpose`, *etc.*. In any case, these are *read-only* variables, and normal Lisp assignment using `setf` is prohibited. (Unfortunately, the Nyquist interpreter does not presently enforce this rule.) Even though assignment to the environment is prohibited, it is essential to have control over the values passed to nested functions, overriding the default values of these implicit parameters. This is the role of the transformation operations. When Nyquist evaluates

```
(transpose 5 (osc c4))
```

the following steps happen:

1. The value of the symbol `*transpose*` is incremented by 5.
2. The expression `(osc c4)` is evaluated. The implementation of the function

`osc` uses `*transpose*` in its calculation of frequency, so the result is transposed 5 semitones from C4 to F4.

3) The previous value of `*transpose*` is restored.

4) The sound computed by `(osc c4)` is returned as a result.

Note that transpose does not follow normal Lisp expression evaluation order, which would first evaluate `5` and `(osc c4)`, then pass the values to transpose. All transformations, including `transpose`, are special functions (Lisp macros to be precise). They operate on unevaluated parameters, allowing the Nyquist implementation to modify the environment *before* evaluating a behavior.

One more example will help emphasize this critical point. Consider the following:

```
(seq (phrase1 100) (phrase2 100))
```

An intuitive but incorrect assumption would be that `phrase1` and `phrase2` are computed to yield sounds, and that `seq` then shifts `phrase2` by the duration of `phrase1` and adds the two sounds together. In fact, `seq` first evaluates `phrase1`, passing it the current environment. Next, the logical stop time of `phrase1` is determined to be 1.5 (based on the definition given earlier.) The environment is modified so that the start time is 1.5, and `phrase2` is evaluated. It is entirely up to `phrase2` when it starts, although any "well-behaved" function will start at the indicated start time. All of the Nyquist built-in functions are "well-behaved" by default.

Before leaving the topic of transformations, I should mention that Canon has been criticized for its use of global variables (Honing 1995). This is, however, just an instance of shallow binding, a well-known implementation technique. All language *implementations* use global variables, so please do not infer properties of the semantics from isolated details of the implementation. The semantics are essentially those of dynamically scoped variables, further restricted by the rule that the variables are only modified through a set of transformation control constructs.

Logical Stop Times

A problem faced in many score languages is how specify the end of a sound such that sequences of sounds can be specified. An obvious definition would be the point where the sound reaches and remains at zero, or one could simply define sounds to have a length of so many samples. The problem is that phrases can end with rests, and sometimes the decay of one note should overlap the attack of the next. Sequences based on sample counts do not support the *musical* concept of duration. In Nyquist, all sounds have a "logical stop time," which can be before or after the last sample of the sound.

The `seq` operator computes samples until the logical stop is reached. It then instantiates the next behavior in the sequence and begins to add remaining samples from the first sound to samples of the new sound. On the other hand, if the logical stop time is after the samples, the `seq` operator will append zero samples until the logical stop time. In this way, overlapping and non-overlapping sequences can be easily specified.

Control and Audio Signals

One of the goals of Nyquist is to provide a very powerful environment in which to control synthesis. In many respects, control is more complex and requires more language support than synthesis. After all, control is at the interface between symbolic score description and continuous signal processing. This is where the two abstractions interact.

In Nyquist there is no difference between a control signal and an audio signal. The full power of the language, including behavioral abstraction, sequential and parallel composition, and a full range of signal generators and filters, can be applied to generate control functions. One use of control functions is to parameterize a signal manipulation process. Examples of this usage include amplitude envelopes, frequency envelopes, and filter coefficients. Another use of control functions is to parameterize a transform. For example, the `loud` and `transpose` introduced in the previous section become time-varying transforms when provided with a control signal. A full description of the semantics and implementation of continuous controls is forthcoming (Dannenberg, to appear A).

Music-11 (Vercoe 1981) and Csound (Vercoe 1986) demonstrate a gain in efficiency by computing control information at a lower sample rate than the audio sample rate. In Nyquist, signals can be computed at any sample rate, but there are default rates for both audio and control signals. When signals of different sample rates are combined, the signal with the low sample rate is automatically up-sampled to match the higher sample rate. In most cases, linear interpolation is used, avoiding "zipper noise" in envelopes. Band-limited sample-rate conversion can be added explicitly if desired. To simplify mixed sample rate computation, all specifications are in terms of seconds rather than sample counts.

Multiple Channels

Nyquist signals can have any number of channels. A multi-channel signal is denoted by an array of simple signals. Most functions in Nyquist will accept multi-channel signals as operands. Simple operands are "widened" as necessary. For example, if we try to multiply a stereo signal <A, B> by a simple envelope signal C, the envelope is duplicated and a stereo signal <A ∞ C, B ∞ C> is returned. The following "pan" function takes a monophonic signal and a pan control, returning a stereophonic signal. The Lisp `vector` function assembles an array from two elements:

```
(defun pan (signal control)
  (vector (mult signal (sum 1 (scale -1 control)))
          (mult signal control)))
```

Signals as Values

One of the nice features of Nyquist is that signals are immutable values that can be assigned to variables and passed as parameters. In most computer music languages, signals are transient because they are computed one block at a time, and the previous block is overwritten by the current one. In Nyquist, if a signal is assigned to a variable, the whole sound is available for reuse. This means that

waveform tables, envelopes, grains, and impulses can be saved and reused. With large primary memories, it is convenient to save computed sounds in memory rather than in disk files. For example, sounds stored in memory can be used for comparison or as readily available inputs when evaluating filter parameters.

There is the danger of exceeding memory limits. One megabyte holds about 11 seconds of Nyquist's 32-bit samples at 22KHz. One drawback of Nyquist is that it is up to the user to avoid storing huge sounds in memory. On the other hand, as long as the user avoids assignment, memory usage is very modest. Nyquist *never* computes an entire sound in memory unless the sound is explicitly assigned to a global variable. A future article (Dannenberg, to appear B) will discuss the memory management implementation.

Discussion

Nyquist is the result of an evolution that began with Arctic, where the basic semantics were developed, continued with Canon, where the Lisp syntax and continuous transforms were added, and includes Fugue (Dannenberg, Fraley, and Velikonja 1991, 1992), an earlier implementation. Nyquist is near the "end of the line" in this evolution, and I am making extra efforts to document (Dannenberg 1995) and distribute the results.

Within the current language framework, there is much room for additional work. Additional synthesis techniques, support for spectra and analysis data, and high quality resampling (Smith and Gossett 1984) are all in progress. Ports to other operating systems, especially to Microsoft Windows would help Nyquist reach a broader audience.

In many ways, the evolution of Nyquist goes all the way back to the beginnings of Music N. There are a few core ideas in Music N that are at the foundation of Nyquist, and it is humbling to think how revolutionary these ideas were in the 50's. First, there is the idea of instantiation, that every note "card" in the score creates an instance, a copy, of a computation. In Nyquist, every function application creates an instance of a computation as in Music N. Nyquist diverges from visual systems like Kyma (Scaletti 1989) and Max (Puckette 1991), where icons denote specific resources (Dannenberg, Rubine, and Neuendorffer. 1991) rather than abstractions that can be instantiated.

Second, Music N pioneered the idea of connecting signal-processing operators (unit generators) via streams of samples representing signals. This idea was picked up later in computer science (Dennis 1980, Dennis and Weng 1979). In Music N, one has to order unit generator expressions by hand to get the proper behavior, but the illustrations of Music N patches (Mathews 1969) make it clear that there was a foundation of functional, dataflow semantics underlying the procedural syntax. Nyquist improves on the procedural syntax with an expression syntax. Furthermore, Nyquist introduces standard functional programming facilities using its Lisp base language. Finally, Nyquist retains the idea that a symbol should represent a function of time, making it simple to interconnect and combine operators by writing expressions.

Third, Music N introduced the idea of associating a time and duration with the evaluation of an expression. This idea was also used in the 4CED program (Abbott

1981), which strongly influenced Arctic. Time and duration (actually "stretch factor") were the only "environment" variables in Arctic, and the idea of the implicit parameters came from observing how timing is so elegantly handled in Music N. This notion was later extended to allow transformations of a variety of parameters and to allow continuous transformations.

Fourth, behavioral abstraction is not present in Music N, but there are envelope generators whose output does not bear a linear relationship to duration, e.g. the attack time may be invariant. This preoccupation with envelopes and other behaviors influenced the design of Arctic, which along with Formes (Cointe and Rodet 1984) was an early language to demonstrate that classes of behavior could be encapsulated into abstractions, such that the abstractions could be temporally transformed. The same mechanism, extended to continuous transformations is present in Nyquist.

One of the goals of Arctic and Nyquist was real-time control, and Nyquist was originally designed to be a real-time system. There are two problems interfering with a real-time version. First, the underlying Lisp interpreter must pause occasionally for garbage collection. The interpreter could be replaced, so this is a solvable problem. The second problem is that we want external events such as MIDI note-on messages to instantiate computation within Nyquist and furthermore to control and update these computations, for example a MIDI note-off message should terminate a sound. Since Nyquist sounds are immutable, it is hard to see how Nyquist can terminate a note once it begins. A solution in keeping with the functional programming style would be to make the note computation a function of MIDI input. This can be done, but it leads (as far as I know) to quite ugly and confusing programs. A more object-oriented solution seems to be the best approach, but this would require a major design change. I prefer to leave Nyquist as is.

In summary, Nyquist is a distillation of some of the most important concepts of language design and composition systems. By integrating symbol processing and signal processing capabilities, it offers a very powerful and flexible language to composers and researchers. In spite of the high-level nature of Nyquist, the low-level routines are highly optimized and the overall performance is quite high. A detailed analysis of software synthesis performance issues will appear in a separate article (Dannenberg and Thompson, to appear).

Conclusion

Nyquist is a high-level language for sound synthesis and composition. Nyquist is unique in that is spans a range of computational tasks from score manipulation to synthesis within a single integrated language. Nyquist already has an efficient implementation running on Unix workstations and Macintosh personal computers.

Nyquist can be used directly through its programming interface. We also see the possibility of using Nyquist as a "sound rendering language" in the sense that Postscript (Adobe 1985) is a graphics rendering language. Thus, Nyquist could operate as a network sound server or a "back end" for any number of graphical composition or data sonification environments.

Companion articles will describe transformation and time warping in Nyquist, the Nyquist run-time system, and optimization techniques for software signal

processing. In the future, we hope Nyquist will become more widely used as a research and composition tool, and we are working to make it more reliable, more complete, and executable on more systems. This is no small task, and I welcome assistance in reaching these goals.

Acknowledgements

Nyquist development has been helped by many people. Chris Fraley wrote most of Fugue, the precursor to Nyquist. Peter Velikonja, an early user, showed us both strengths and weaknesses of Fugue, laying the foundations for Nyquist. Cliff Mercer and Joe Newcomer assisted with the first implementation, and Barry Vercoe's Csound provided some of the inner loops. Eric Dahl has ported Nyquist to the Macintosh, and Alex Sanielevici has contributed many instruments and identified many bugs in an earlier version (all fixed of course!).

References

Abbot, C. 1981. "The 4CED Program." *Computer Music Journal* 5(1) (spring): 13-33.

Adobe Systems, Inc. 1985. *PostScript Language Reference Manual.* Reading, Massachusetts: Addison-Wesley.

Anderson, D. P., and R. Kuivila. 1990. "A System for Computer Music Performance." *ACM Transactions on Computer Systems* 8(1): 56-82, February.

Ashcroft, E. A., and W. W. Wadge. 1977. "Lucid, A Nonprocedural Language with Iteration." *Communications of the ACM* 20(7): 519-526.

Cointe, P. and Rodet, X. 1984. "Formes: an Object & Time Oriented System for Music Composition and Synthesis." *In 1984 Symposium on LISP and Functional Programming*.. New York: ACM Press, pp. 85-95.

Dannenberg, R. B. 1984. "Arctic: A Functional Language for Real-Time Control." In *1984 ACM Symposium on LISP and Functional Programming*. New York: Association for Computing Machinery, pp. 96-103.

Dannenberg, R. B. 1986. "The CMU MIDI Toolkit." In *Proceedings of the 1986 International Computer Music Conference*. San Francisco: International Computer Music Association, pp 53-56.

Dannenberg, R. B. 1989. "The Canon Score Language." *Computer Music Journal*) 13(1):47-56.

Dannenberg, R. B. 1993. *The CMU MIDI Toolkit*. Pittsburgh, Pennsylvania: Carnegie Mellon University.

Dannenberg, R. B. 1995. *Nyquist Reference Manual*. (program documentation).

Dannenberg, R. B. 1997. "Abstract Time Warping of Compound Events and Signals." *Computer Music Journal*, 21(3):61-70.

Dannenberg, R. B. 1997. "The Implementation of Nyquist, A Sound Synthesis Language." *Computer Music Journal*, 21(3):71-82

Dannenberg, R. B., P. McAvinney, and D. Rubine 1986. "Arctic: A Functional Language for Real- Time Systems." *Computer Music Journal* 10(4):67-78.

Dannenberg, R. B., D. Rubine, and T. Neuendorffer. 1991. "The Resource-Instance Model of Music Representation." In *Proceedings of the 1991 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 428-432.

Dannenberg, R. B., C. L. Fraley, and P. Velikonja. 1991. "Fugue: A Functional Language for Sound Synthesis." *Computer* 24(7) (July): 36-42.

Dannenberg, R. B., C. L. Fraley, and P. Velikonja. 1992. "A Functional Language for Sound Synthesis with Behavioral Abstraction and Lazy Evaluation." In Denis Baggi, ed. *Readings in Computer-Generated Music.* Los Alamitos, California: IEEE Computer Society Press, pp. 25-40.

Dannenberg, R. B., and N. Thompson (to appear). "Real-Time Software Synthesis on Superscalar Architectures." *Computer Music Journal*.

Dennis, J. B. 1980. "Data Flow Supercomputers." *Computer* 13(11) (November): 48-56.

Dennis, J. B., and K. K.-S. Weng. 1979. "An Abstract Implementation For Concurrent Computation With Streams." In *Proceedings of the 1979 International Conference on Parallel Processing*. New York: IEEE Computer Society, pp. 35-45.

Desain, P., and H. Honing. 1992a. "Time Functions Function Best as Functions of Multiple Times." *Computer Music Journal*, 16(2): 17-34 (Summer). Reprinted in Desain and Honing 1992b.

Desain, P., and H. Honing. 1992b. *Music, Mind and Machine, Studies in Computer Music, Music Cognition and Artificial Intelligence*. Amsterdam: Thesis Publishers.

Desain, P., and H. Honing. 1993. "On Continuous Musical Control of Discrete Musical Objects." In *Proceedings of the 1993 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 218-221.

Honing, H. 1995. "The Vibrato Problem, Comparing Two Solutions." *Computer Music Journal,* 19(3) (fall): 32-49.

Kopec, G. E. 1985. "The Signal Representation Language SRL." *IEEE Transactions Acoustics, Speech and Signal Processing*. 33(4):921-932.

Mathews, M. V. 1969. *The Technology of Computer Music*.. Cambridge, Massachusetts: MIT Press.

Mathews, M. V. and F. R. Moore. 1970. "A Program to Compose, Store, and Edit Functions of Time." *Communications of the ACM* 13(12):715-721.

Moore, F. R. 1982. "The Computer Audio Research Laboratory at UCSD." *Computer Music Journal* (6)1:18-29.

Puckette, M. 1991 "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer Music Journal* 15(3): 68-77.

Rodet, X., and P. Cointe. 1984. "FORMES: Composition and Scheduling of Processes." *Computer Music Journal* 8(3): 32-50.

Rodet, X. and G. Eckel. 1988. "Dynamic Patches: Implementation and Control in the Sun-Mercury Workstation." In *Proceedings of the 1988 International Computer Music Conference.* San Francisco: International Computer Music Association. pp. 82-89.

Rodet, X., Y. Potard, and J.-B. Barriere. 1984. "The CHANT Project: From Synthesis of the Singing Voice to Synthesis in General." *Computer Music Journal* 8(3): 15-31.

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13(2):23-38.

Scaletti, C. and E. Johnson. 1988. "An Interactive Graphic Environment for Object-Oriented Music Composition and Sound Synthesis." In *Proceedings of the 1988 Conference on Object-Oriented Languages and Systems.*. New York: Association for

Computing Machinery. pp. 18-26.
Schorr, H. and Waite, W. 1967. "An Efficient and Machine Independent Procedure for Garbage Collection in Various List Structures." *Communications of the ACM* 10(8):501-506.
Schottstaedt, B. 1994. "Machine Tongues XVII: CLM: Music V Meets Common Lisp." *Computer Music Journal* 18(2): 30-37.
Smith, J. O., and P. Gossett. 1984. "A Flexible Sampling Rate Conversion Method." In *Proceedings of ICASSP, Vol. 2.* New York: IEEE. pp. 19.4.1-19.4.4.
Touretzky, D. S. 1984. *LISP: A Gentle Introduction to Symbolic Computation*. New York: Harper and Row.
Vercoe, B. 1981. *Reference Manual for the MUSIC 11 Sound Synthesis Language*. MIT Experimental Music Studio.
Vercoe, B. 1986. *CSOUND: A Manual for the Audio Processing System and Supporting Programs*. MIT Media Lab.