Introduction

- Computer Music
  - has demanding computational requirements
  - has demanding requirements for expressivity
  - has given rise to many language innovations
    - Dataflow computing
    - Temporal semantics
    - Visual programming for novice programmers
- Nyquist is a functional programming language for sound synthesis and music composition
Overview

- A Model for Computer Music Synthesis:
  - The structure of sound synthesis programs
- The Model in Functional Terms
- Efficient Inner Loops
- Expressiveness and Efficiency
- Conclusion

A Model for Computer Music Synthesis

- What to compute?
- Simple example
  - Assume an array \( w[0..N-1] \) with one waveform period
  - \( s(t) = w\lfloor (t \times f \times N) \rfloor \mod N \)
  - Evaluate at discrete time points, \( t = i / r \)
Combining Functions

- Let's multiply by an envelope to avoid sudden on/off:
  - \( e(t) = \begin{cases} 
  t / 0.1 & \text{if } t < 0.1 \\
  1.1 - t & \text{if } t < 1.1 \\
  0.0 & \text{else}
  \end{cases} \)
  - \( c(t) = s(t) \cdot e(t) \)

Unit Generators

- Since everything is a function of time, we can drop the \( t \).
- We refer to signal generating and signal processing primitives as unit generators.
Implementation - Traditional

1. Oscillator
2. Envelope Generator
3. (multiply (waveform table freq) (envelope))

The Model In Functional Terms

Oscillator
Envelope Generator
(x)
The SOUND Data Type

scale factor

sample rate

t0

logical stop time

termination time

time

To the user: *just a function of time.*

Nyquist Implementation – Data Structure

Previously read blocks are returned to heap

Next block is added here

Copyright 2005, Roger B. Dannenberg
**SOUND Data Type – 3**

- Reader Object
- Sample Block
- Suspended Sample Stream Generator

**SOUND Data Type – 4**

- Reader Object
- Sample Block
- Suspended Sample Stream Generator
SOUND Data Type – 5

Reader Object

Sample Block

Suspended Sample Stream Generator

SOUND Data Type – 6

Reader Object

Sample Block

Suspended Sample Stream Generator
SOUND Data Type – 1b

Suspended Sample Stream Generator

Reader Object

Reader Object

SOUND Data Type – 2b

Sample Block

Suspended Sample Stream Generator
SOUND Data Type – 3b

SOUND Data Type – 4b
SOUND Data Type – 5b

SOUND Data Type – 6b
A SOUND Expression

\[
\text{(multiply (waveform table freq) (envelope))}
\]

Implementation - Traditional

1. Oscillator
2. Envelope Generator
3. Suspension Block
Efficient Inner Loops – 1

(PROD-ALG
  (NAME "prod")
  (ARGUMENTS ("sound_type" "s1")
               ("sound_type" "s2"))
  (START (MAX s1 s2))
  (TERMINATE (MIN s1 s2))
  (COMPUTATIVE (s1 s2))
  (LINEAR s1 s2)
  (INNER-LOOP "output = s1 * s2")
)

Efficient Inner Loops – 2

Get samples from s1 and s2
Calculate n, the number of samples ready to process

if (n) do { /* the inner sample computation loop */
  *out_ptr_reg++ = *s1_ptr_reg++ * *s2_ptr_reg++;
} while (--n); /* inner loop */

Repeat until we have a full output buffer
Then save state in the suspension object and return
Expressions and Efficiency

- Compiled inner loops and large blocks (about 1K) make Nyquist about as efficient as C code.
- Language expressiveness can make Nyquist significantly faster than C code alone.

Mixed Sample Rates

- Control signals:
  - Amplitude envelopes
  - Vibrato
  - Filter coefficients
- … can be computed at low sample rates.
- Nyquist supports mixed sample rates.
- Linear interpolation within inner loops
  - Machine generated code is essential here.
- By calculating everything in terms of time and thinking of sounds as functions of time (rather than vectors), mixed sample rates are not apparent to users.
Sample Accuracy

- In many synthesis systems, time is quantized to boundaries of fixed-length sample blocks.

- This forces blocks to be small.
- Nyquist uses variable-length blocks so all sounds are timed to within 0.5 sample, even with large blocks.
- Allows better amortization of overhead.

Scores and Instruments – 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Dur</th>
<th>Instr</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.5</td>
<td>flute</td>
<td>c#, -5db, …</td>
</tr>
<tr>
<td>1.5</td>
<td>0.5</td>
<td>flute</td>
<td>d, -5db, …</td>
</tr>
<tr>
<td>2.0</td>
<td>1.0</td>
<td>flute</td>
<td>f#, -10db, …</td>
</tr>
<tr>
<td>1.5</td>
<td>1.5</td>
<td>sax</td>
<td>a, -10db, …</td>
</tr>
</tbody>
</table>

“Orchestra”

- Instrument “flute”:
  - How to synthesize a “flute” sound
- Instrument “sax”:
  - How to synthesize a “sax” sound
Scores and Instruments – 2

- Nyquist has temporal semantics, no need for separate score language:
  - `(seq a b ...)` makes sequence of sounds `a, b, ...`
  - `(sim a b ...)` mixes (add) simultaneous sounds
  - `(at t a)` shifts `a` to time `t`
  - `(stretch d a)` stretches `a` by `d`
- So sequential and score-like behavior can be embedded anywhere
  - E.g. `(seq (attack) (sustain) (release))`
- Much more efficient to schedule changes in outer loop than within inner loops
Using Language Features

- Polymorphism
  - (lp (noise) 500)
  - (lp (noise) (sweep 500 1000))

- Closures
  - (seq (crash) (boom) (bang))

- Macros
  - seq, transpose, stretch, at

Some Examples

- Granular Synthesis
  - What are the parameters?
  - Are the details hidden?

- “Tail Iteration”
  - (defun drum-roll ()
    (seq (drum-stroke) (drum-roll)))
Conclusions

- Nyquist is based on XLisp, with a new datatype: SOUND
- SOUND allows programmers to express sound computation using nested expressions
- Lazy evaluation automatically reduces expression trees to an efficient “standard model” of unit generators and buffers
- Further efficiency is obtained through:
  - Automatically translated inner loops
  - Highly expressive language (temporal semantics, mixed sample rates, etc.)
- Nyquist is both highly expressive and highly efficient