Functional Programming for Signal Processing:

There's More to Life than Inner Loops

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University

1

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

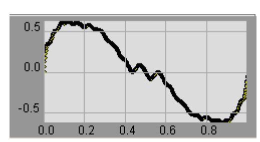
2

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

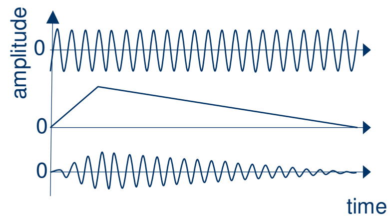
- What to compute?
- Simple example
  - Assume an array $w[0..N\text{-}1]$ with one waveform period
  - $s(t) = w[\lfloor (t \times f \times N) \rfloor \bmod N]$
  - Evaluate at discrete time points, $t = i / r$



Copyright 2005, Roger B. Dannenberg

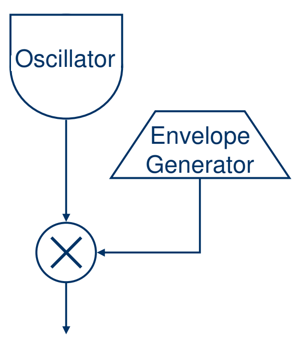
Combining Functions

- Let's multiply by an *envelope* to avoid sudden on/off:
- $e(t) = \text{if } t < 0.1 \text{ then } t / 0.1$
  $\text{elif } t < 1.1 \text{ then } 1.1 - t$
  $\text{else } 0.0$
- $c(t) = s(t)\ e(t)$

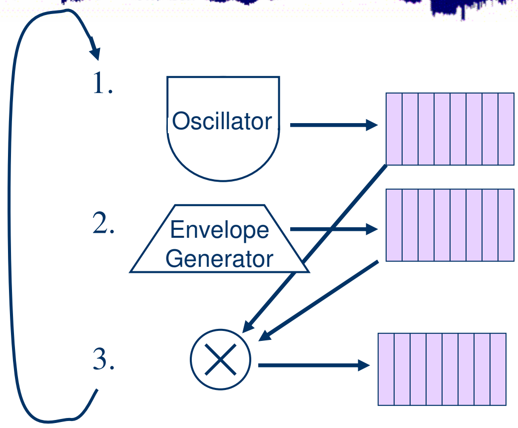


Unit Generators

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

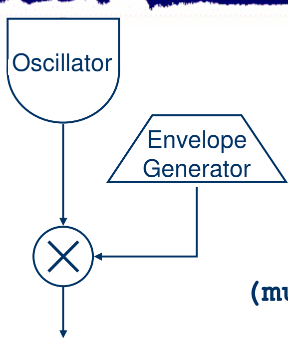


Implementation - Traditional



7

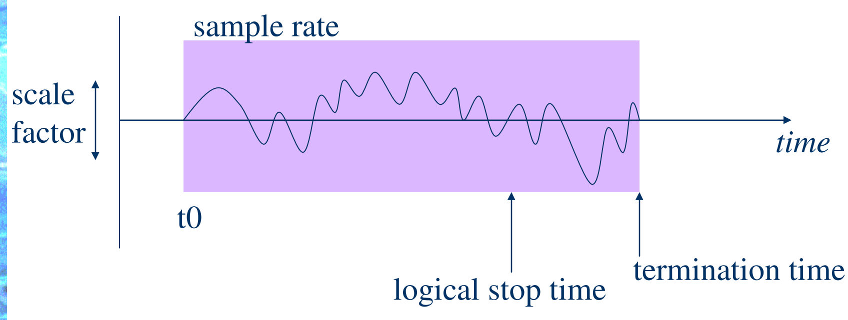
The Model In Functional Terms




```
(multiply (waveform table freq)
          (envelope))
```

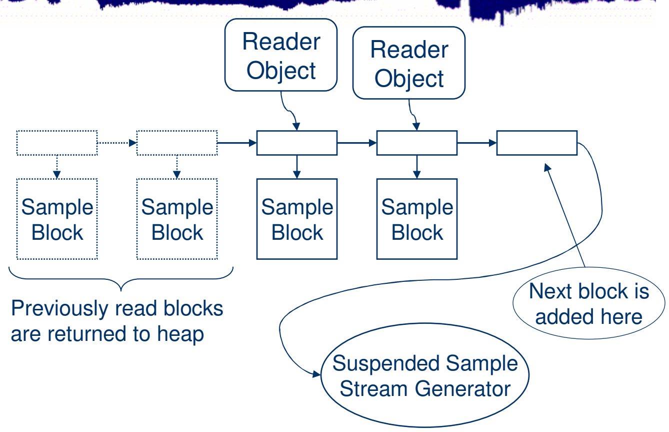
8

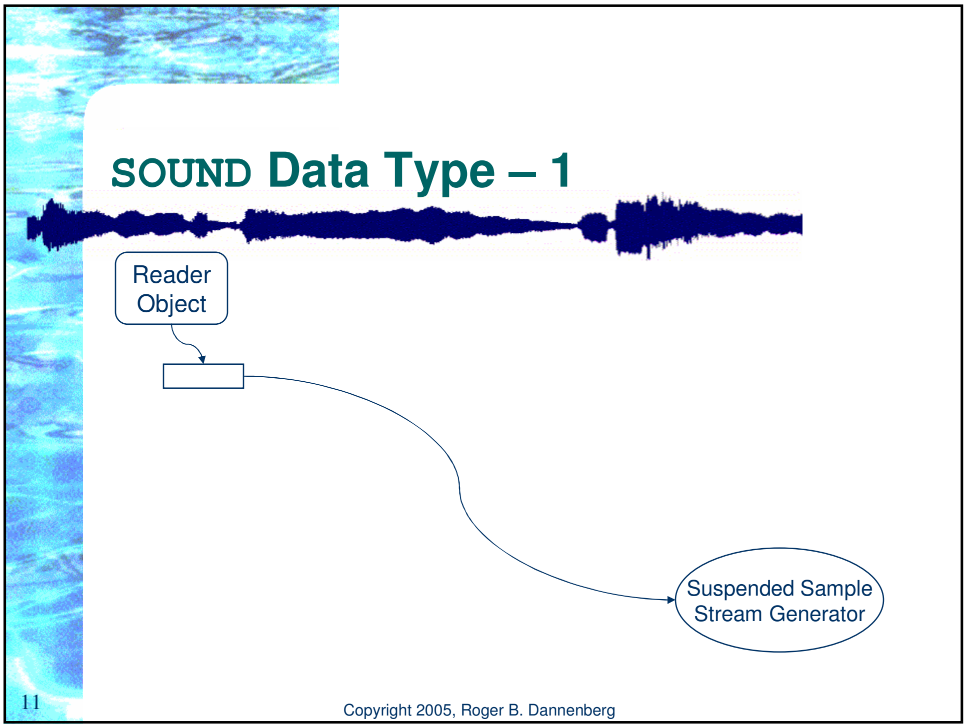
The SOUND Data Type

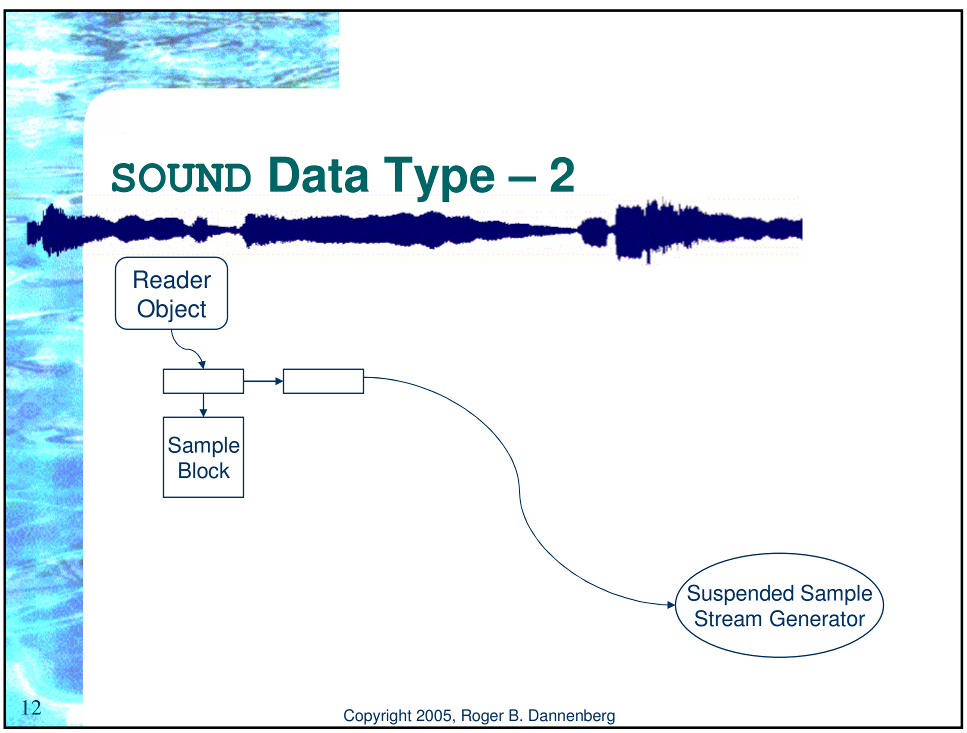


To the user: *just a function of time*.

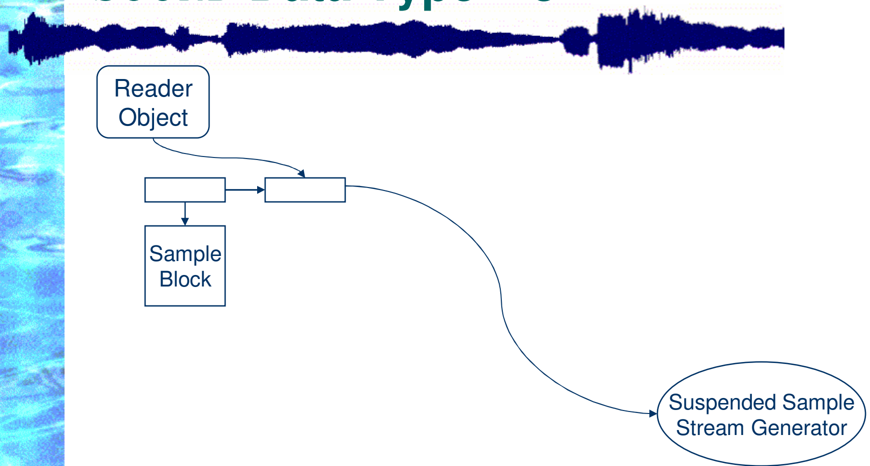
Nyquist Implementation – Data Structure




SOUND Data Type – 1
Reader Object
Suspended Sample Stream Generator
11
Copyright 2005, Roger B. Dannenberg

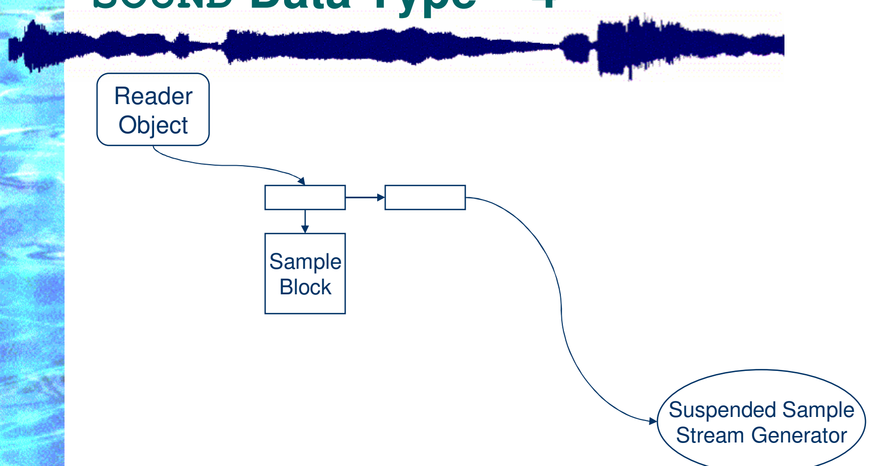

SOUND Data Type – 2
Reader Object
Sample Block
Suspended Sample Stream Generator
12
Copyright 2005, Roger B. Dannenberg

SOUND Data Type – 3



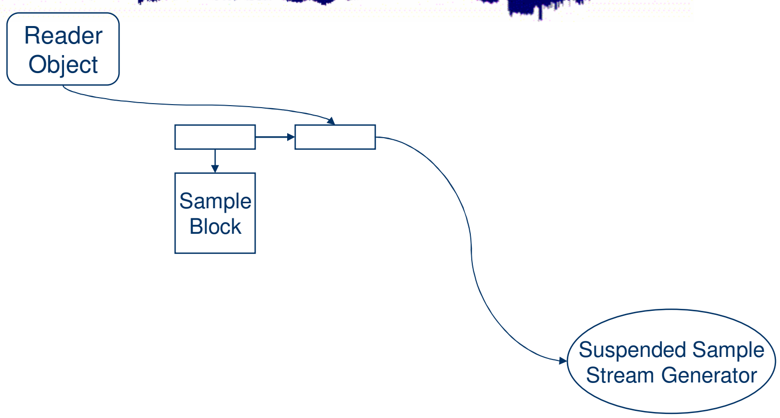
13

SOUND Data Type – 4



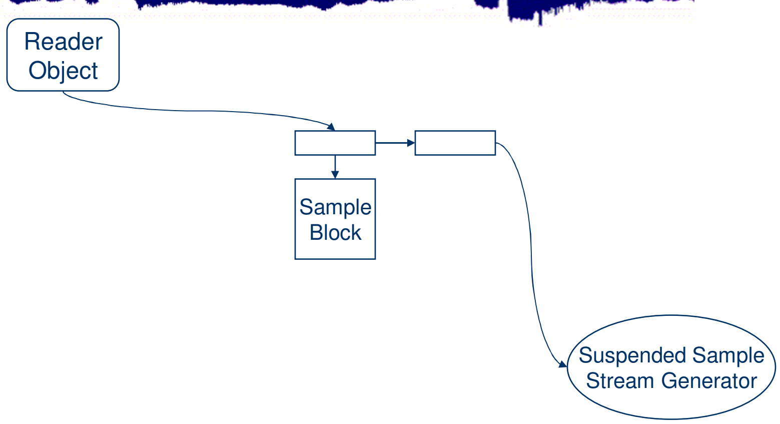
14

SOUND Data Type – 5



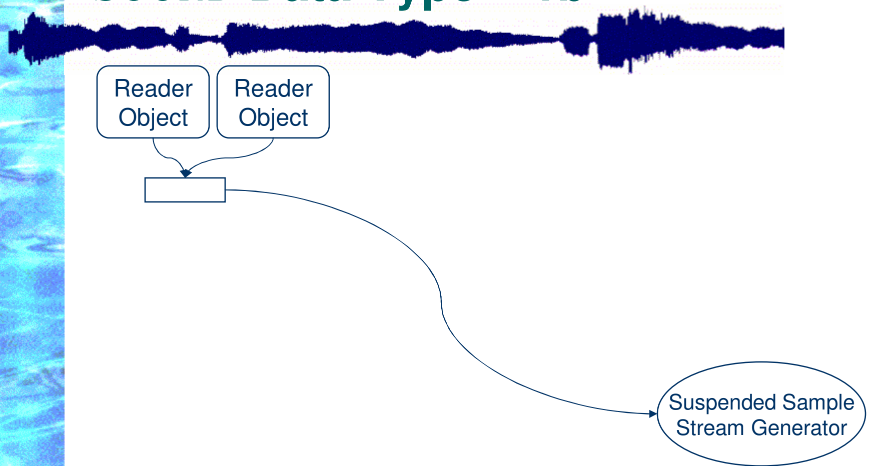
15

SOUND Data Type – 6



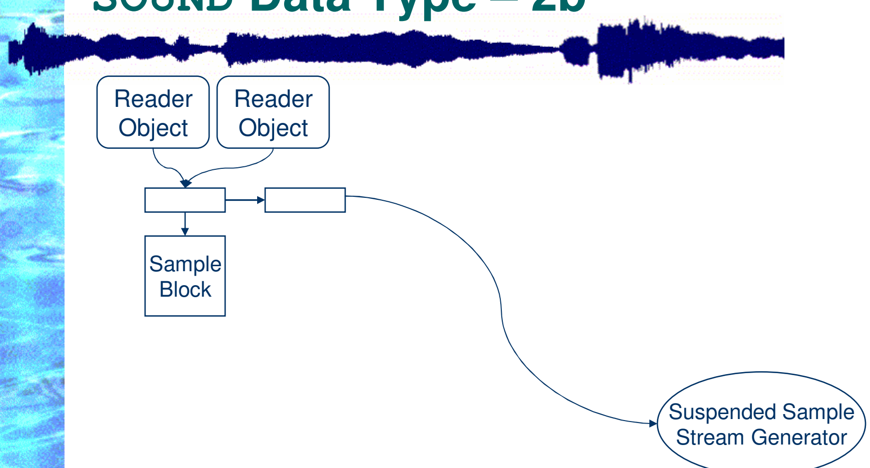
16

SOUND Data Type – 1b



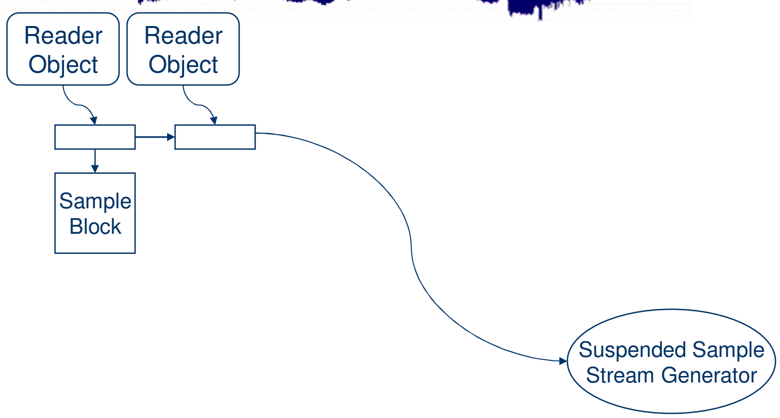
17

SOUND Data Type – 2b



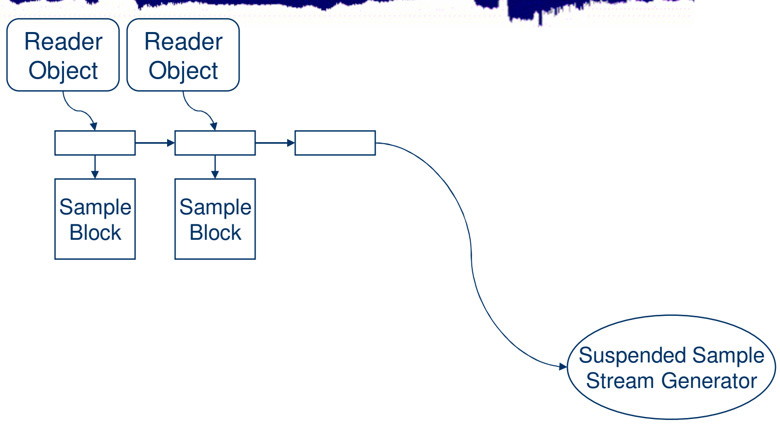
18

SOUND Data Type – 3b



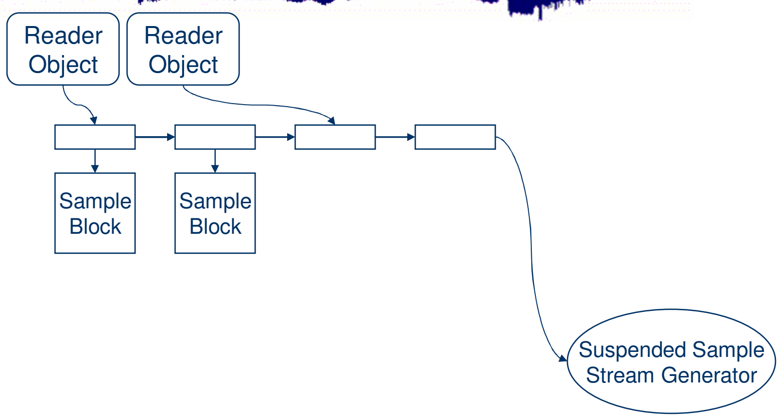
19

SOUND Data Type – 4b



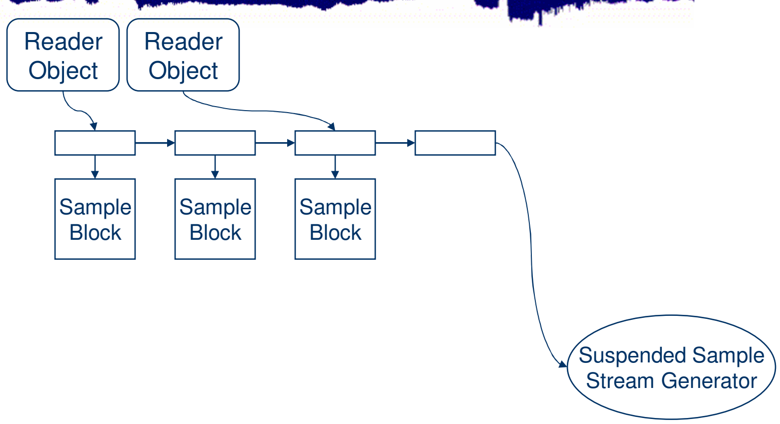
20

`SOUND` Data Type – 5b



21

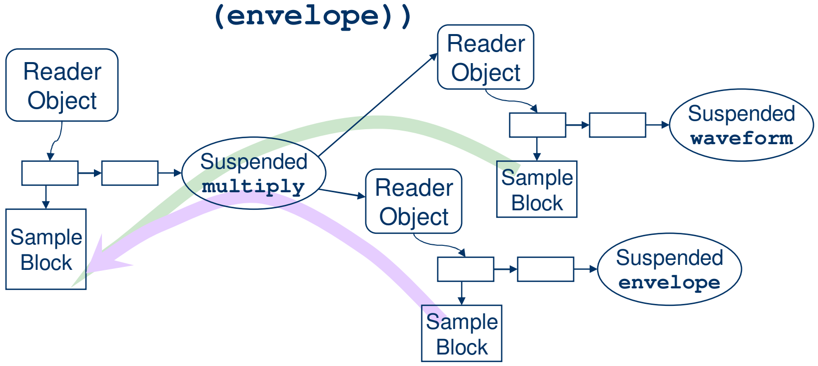
`SOUND` Data Type – 6b



22

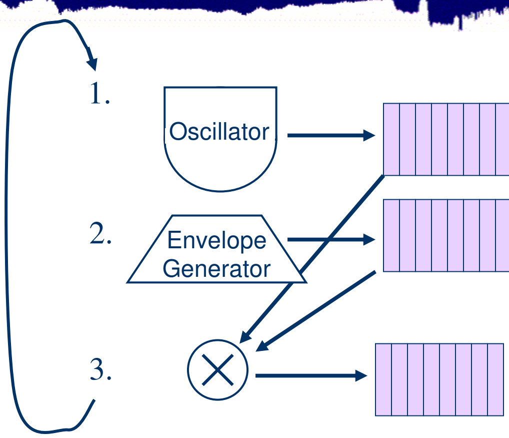
A SOUND Expression

```
(multiply (waveform table freq)
          (envelope))
```



23

Implementation - Traditional



24

Efficient Inner Loops – 1

```
(PROD-ALG
    (NAME "prod")
    (ARGUMENTS ("sound_type" "s1")
               ("sound_type" "s2"))
    (START (MAX s1 s2))
    (TERMINATE (MIN s1 s2))
    (COMMUTATIVE (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

Expressiveness 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.*

27

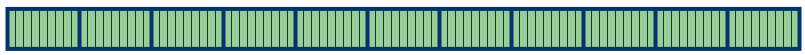
Mixed Sample Rates

- 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.

28

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

29

Scores and Instruments – 1

"Score"

Time	Dur	Instr	Parameters
0	1.5	flute	c#, -5db, …
1.5	0.5	flute	d, -5db, …
2.0	1.0	flute	f#, -10db, …
1.5	1.5	sax	a, -10db, …
…			

"Orchestra"

Instrument "flute":

How to synthesize a "flute" sound

Instrument "sax":

How to synthesize a "sax" sound

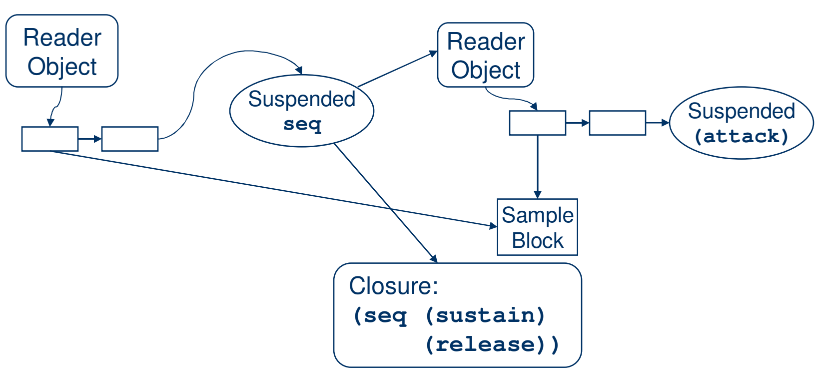
30

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

31

`(seq (attack) (sustain) (release))`



32

Using Language Features

- Polymorphism
  - `(lp (noise) 500)`
  - `(lp (noise) (sweep 500 1000))`
- Closures
  - `(seq (crash) (boom) (bang))`
- Macros
  - `seq, transpose, stretch, at`

33

Some Examples

- Granular Synthesis
  - What are the parameters?
  - Are the details hidden?
- "Tail Iteration"
  - ```
    (defun drum-roll ()
      (seq (drum-stroke) (drum-roll)))
    ```

34
    ```

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

35