UNIT GENERATORS

Building blocks for sound synthesis

Overview for the Week

• What’s a Unit Generator?
• What are some unit generators in Nyquist?
• Unit Generator Implementation
• Functional Programming
• Wavetable Synthesis
• Scores in Nyquist
• Score Manipulation
What Is a Unit Generator?

- In the 50’s Max Mathews conceived of sound synthesis by software using networks of modules: “Unit Generators”
- UGs are “primitives” in a sound synthesis system
- They perform sound generation and sound processing

Unit Generator examples

- Oscillator
- Multiplier
- Envelope
Combining Unit Generators

Unit Generators in Nyquist

- Unit Generators are Functions on sounds
Some Basic Unit Generators

- $\text{osc}(c4)$
- $\text{pwl}(0.03, 1, 0.8, 1, 1)$
- $\text{osc}(c4) \times \text{pwl}(0.03, 1, 0.8, 1, 1)$
- $\text{osc}(c4) \times \text{osc}(g4)$

Evaluation

- Normally, SAL expressions evaluate their parameters, then apply the function: $f(a, b)$
- What about sounds?
  - To avoid storing huge values in memory,
  - Nyquist uses *lazy evaluation*
  - Samples are computed only when they are needed
  - Nyquist *Sounds* contain either samples or the potential to deliver samples, or some combination
UNIT GENERATOR IMPLEMENTATION

What’s inside a Unit Generator and how do we access it?

Unit Generator Implementation

- Have to store the intermediate state somewhere
  - e.g. the current phase and frequency of an oscillator UG
  - therefore, Unit Generators are implemented as objects in Nyquist
- Objects are accessed implicitly to provide samples – they are hidden from the user
- Many languages present (expose?) UG’s as an explicit graph of objects.
  - A pass is made over the graph to propagate the next sample (or block of samples) from input to output
Sounds in Nyquist

- In Nyquist, there’s no direct access to the Unit Generators as objects
- Instead, functions make and return objects called SOUNDS
- Instances of Unit Generator objects are contained within sounds and called upon when samples are needed
- We’ll learn more about SOUNDS later
Playing a Sound

- If you write `play sound-expression`
  - A sound is returned
  - Internally, the sound has a graph of unit generators
  - To play the samples, the graph is traversed, generating samples incrementally
  - The samples (in blocks of about 1000) are played in "real time"
- If you write `set var = sound-expression`, the entire sound might be computed, saved, and stored in memory

FUNCTIONAL PROGRAMMING IN NYQUIST

Programs are expressions!
Functional Programming

- Program in terms of functions and values
- NOT VARIABLES
- Compose functions: f(g(x), h(x)) to get complex behaviors
- DO NOT MAKE MANY STEPS AND STATE CHANGES TO GET COMPLEX BEHAVIORS

A Very Stateful Program

```plaintext
variable sum
function init(x) sum = x
function addx(x) sum += x
function multx(x) sum *= x
function mysound()
  begin
    exec init(hzosc(440.0))
    loop for i from 2 to 10
      exec addx(hzosc(440.0 * i) * rrandom()))
    end
    exec multx(env(0.05, 0.2, 0.5, 1, 0.5, 0.2))
  end
exec mysound()
play sum
```
### A Functional Program

```plaintext
function rand-harm(hz) return hzosc(hz) * rrandom()

function harmonics(hz, n)
begin
  if n = 1 then
    return rand-harm(hz)
  else
    return rand-harm(hz * n) + harmonics(hz, n - 1)
  end

function mysound()
  return harmonics(440.0, 10) * env(0.05, 0.2, 0.5, 1, 0.5, 0.2)
play mysound()
```

### Mostly Functional, Local Variables

```plaintext
function harmonics(hz, n)
begin
  with snd = hzosc(hz * n) * rrandom()
  if n > 1 then
    set snd += harmonics(hz, n - 1)
  return snd
end

function mysound()
  return harmonics(440.0, 10) * env(0.05, 0.2, 0.5, 1, 0.5, 0.2)
play mysound()
```
A Better Functional Program

```plaintext
function harmonics(hz, n)
    return simrep(i, n,
        hzosc(hz * (i + 1)) * rrandom())

function mysound()
    return harmonics(440.0, 10) *
        env(0.05, 0.2, 0.5, 1, 0.5, 0.2)

play mysound()
```

ELIMINATING GLOBAL VARIABLES

Use expressions and functions instead!
Keeping Samples out of Memory

- Never assign sounds to global variables:
  - `set gv = osc(c4); BAD`
- Instead,
  - `function gv()`
  - `return osc(c4); GOOD`
- Then, to access: use `gv()`, not `gv`
Building Waveforms

- This is presented more or less as a “formula”:

```
define variable *table* =
    sim(0.5 * build-harmonic(1.0, 2048),
        0.25 * build-harmonic(2.0, 2048),
        0.125 * build-harmonic(3.0, 2048),
        0.062 * build-harmonic(4.0, 2048))
set *table* = list(*table*, hz-to-step(1), #t)
```

Using Waveforms

- *table* is a global – if you set it, OSC will use it:
  - set *table* = ...
  - play osc(c4)
- Or, set another global and pass it to OSC
  - set *mytable* = ...
  - play osc(c4, 1.0, *mytable*)
Piece-wise Linear Functions: PWL

- Common for control functions.
- By default, produces low, control sample rate.
- \( \text{pwl}(t_1, v_1, t_2, v_2, \ldots, t_n) \)

![Graph of PWL function]

Variants of PWL

- \( \text{pwlv}(v_0, t_1, v_1, t_2, v_2, \ldots, t_n, v_n) \)
  - for non-zero starting and ending points
- \( \text{pwe}(t_1, v_1, t_2, l_2, \ldots, t_n) \)
  - exponential interpolation, \( v_i > 0 \)
- \( \text{pwlr}(i_1, v_1, i_2, v_2, \ldots, i_n) \)
  - relative intervals rather than absolute times
- See manual for more variants & combinations
Basic Wavetable Synthesis

• Build a wavetable with the harmonics you want
• Use an oscillator (osc) to generate a tone with these harmonics
• Multiply by an envelope (e.g. pwI) to control the amplitude contour.

• Advantages: simple, efficient, direct control
• Disadvantages: spectrum (strength of harmonics) does not change with pitch or time as in most acoustic instruments.

SCORES
INTRODUCTION

Scores describe sound events organized in time
Terminology – Pitch

- Musical scales are built from two-sizes of intervals: whole steps and half steps
- Whole step = 2 half steps
- “flats” lower by half step, “sharps” raise by half step
- In Nyquist documentation, “step” means half-step
  - step-to-hz, hz-to-step, (osc step)
- Middle C (ISO C₄) arbitrarily represented by 60
  - c₄ = 60, cs₄ = 61, cf₄ = 59,
  - b₃ = 59, bs₃ = 60
- Steps are logarithms of frequency
  - frequency doubles every 12 steps
  - frequency doubling (or halving) is called an interval of an “octave”

Terminology – Harmonics, etc.

- Imagine a periodic function of time
- We hear that as a tone with pitch
- The repetition rate (1/period) is the “fundamental frequency”
  (other frequencies are usually present and are called overtones, partials, or harmonics)
- Any continuous function can be decomposed into a sum of sinusoids. (a finite sum for digital audio)
- Periodic functions can be decomposed into sinusoids with frequencies that are integer multiples of the fundamental frequency (these are called harmonics)
Terminology – Sound Events

- Traditional music has “notes”:
  - Pitch
  - Time
  - Duration
  - Loudness (aka Dynamics)
  - Timbre (= instrument and other qualities)

- New music has “sound events”:
  - May be unpitched
  - Time
  - Duration
  - Loudness (aka Dynamics)
  - Potentially many evolving qualities

LISTS

Scores are made of lists, so let’s learn about lists.
Lists in Nyquist

- Standard singly-linked list
- Dynamic typing
  - arbitrary nesting,
  - you can make any binary tree structure

Notation

- In SAL: \{a b c\}
- These are *literals*
  - No evaluation
  - a, b, and c are *symbols*, not *variables*
- To *construct* list from variables:
  - list(a, b, c)
Literals, Variables, Quoting, Cons

```
set a = 1, b = 2,  
c = 3
print {a b c}
{a b c}
print list(a, b, c)  
{1 2 3}
print list(1, 2, 3)  
{1 2 3}
print list(quote(a),  
quote(b),  
quote(c))
{a b c}
print list(a, {b})  
{1 {b}}
print cons(a, {b})  
{1 b}
```

SCORES

How to make a score
Scores

{ sound-event
  sound-event
  sound-event
  ... }

{ time duration sound }

{ instrument attribute: value
  attribute: value
  attribute: value
  ... }

Score Example

{{0.0 1.0 {note pitch: 60 vel: 100}}
{1.0 1.0 {note pitch: 62 vel: 110}}
{2.0 1.0 {note pitch: 64 vel: 120}}}
Score with `score-begin-end` Pseudo-Event

- Can a score be a sound event?
- If so, when does it start? How long is it?

```plaintext
{{0 0 {score-begin-end 0 5}}
 {0.0 1.0 {note pitch: 60 vel: 100}}
 {1.0 1.0 {note pitch: 62 vel: 110}}
 {2.0 1.0 {note pitch: 64 vel: 120}}}
```

Instruments

- An “instrument” is a SAL (or XLISP) function
- How do we get from
  ```plaintext
  {note pitch: 60 vel: 100}
  ```
to a function call?
- STEP 1: List representation of function calls
- STEP 2: Keyword parameters
List Representation of Function Calls (Lisp Syntax)

- A function call in Lisp is represented by:
  - Function *symbol* followed by …
  - … parameter expressions
    - (pluck ef4 3.0)
- Expression can be
  - Number: evaluates to self
  - Symbol: evaluated as a variable
  - List: nested function call

Keyword Parameters

```
function note(pitch: 60, vel: 100)
begin
    return pluck(pitch) * vel * 0.01
end
```

- Now, we can call it:
  - play note(pitch: 72)
  - play note(vel: 50, pitch: g3) ~ 2
Putting It Together: Lisp Syntax + Keyword Parameters

- Example of an expression from a score:
  \{note pitch: 48 vel: 95\}
- Equivalent to this SAL function call:
  \texttt{note(pitch: 48, vel: 95)}
- Whole sound event might look like:
  \{3.0 1.5 \{note pitch: 48 vel: 95\}\}
- Equivalent to this SAL expression:
  \texttt{(note(pitch: 48, vel: 95) \sim 1.5) @ 3.0}

CHORDS

A short-hand notation for scores
Why Keyword Parameters?
Why Lisp?

- Scores are *data*
  - Score manipulation: transpose, stretch, select, …
  - Score generation: algorithmic composition, …

- Scores are *programs*
  - Well-defined semantics
  - Extensible through attributes and function definition

**Special Case: Chords!**

- Example of an event from a score:
  {3.0 0.7 \{note pitch: \{48 55 64\} \vel: 95\}}
- Lists of pitches are “expanded” to individual events, i.e. chords
- Equivalent to these events:
  {3.0 0.7 \{note pitch: 48 \vel: 95\}}
  {3.0 0.7 \{note pitch: 55 \vel: 95\}}
  {3.0 0.7 \{note pitch: 64 \vel: 95\}}
- Note that timing and all non-pitch parameters are duplicated for each note in the chord. (This only works for \texttt{pitch:})
Scores Rendering

- `play timed-seq(my-score)`
  - Use `timed-seq` to turn a score into a SOUND
  - Further processing, e.g. reverb, is possible

- `exec score-play(my-score)`
  - Simple function to play a score
  - Does not return a SOUND value

SCORE PROCESSING

Lots of functions to manipulate scores
Score Processing Functions

- score-shift
- score-transpose
- score-sustain
- score-voice
- score-merge
- score-append
- score-select
- score-filter-length
- score-stretch-to-length
- score-filter-overlap
- score-adjacent-events
- score-sort
- score-repeat
- score-index-of
- score-last-index-of
- score-randomize-start
- score-repeat
- score-index-of
- score-last-index-of
- score-randomize-start
- score-read-smf
- score-write-smf

Score events must be sorted in order of increasing start times

exec score-play(score-sort(
  {{0.0 0.5 {plucked-string pitch: 67 vel: 90 cutoff: 4000}}
  {0.5 0.5 {plucked-string pitch: 69 vel: 95 cutoff: 5000}}
  {1.0 0.5 {plucked-string pitch: 71 vel: 100 cutoff: 6000}}
  {1.5 0.5 {plucked-string pitch: 72 vel: 105 cutoff: 7000}}
  {2.0 0.5 {plucked-string pitch: 71 vel: 100 cutoff: 6000}}
  {2.5 0.5 {plucked-string pitch: 69 vel: 95 cutoff: 5000}}
  {3.0 1.0 {plucked-string pitch: 67 vel: 90 cutoff: 4000}}
  {0.0 1.0 {note pitch: 59 vel: 100}}
  {1.0 1.0 {note pitch: 55 vel: 100}}
  {2.0 1.0 {note pitch: 55 vel: 100}}
  {3.0 1.0 {note pitch: 59 vel: 100}}))))
score-shift

- add 3 seconds to all start times
  \[
  \text{print } \text{score-shift}(\text{my-score}, 3.0)
  \]
- insert 3s rest at time 10
  \[
  \text{print } \text{score-shift}(\text{my-score}, 3.0, \text{from-time: 10})
  \]

score-transpose

- Transpose pitch up one octave:
  \[
  \text{print } \text{score-transpose}(\text{my-score}, \text{keyword(pitch)}, 12)
  \]
- Increase cutoff freq. by 1000:
  \[
  \text{print } \text{score-transpose}(\text{my-score}, \text{keyword(cutoff)}, 1000)
  \]
- Wrong:
  \[
  \text{print } \text{score-transpose}(\text{my-score}, \text{pitch:}, 12)
  \]
  \[
  \text{print } \text{score-transpose}(\text{my-score}, \text{quote(pitch:)}, 12)
  \]
- OK: print score-transpose(my-score, :pitch, 12)
**score-sustain**

- Increase durations by 25% in the time interval from 1 to 3 seconds
  
  ```
  print score-sustain(my-score, 1.25, 
  from-time: 1, to-time: 3)
  ```

**score-voice**

- Turn plucked-string into note and note into plucked-string
  
  ```
  print score-voice(my-score, 
  {{note plucked-string} 
  {plucked-string note}})
  ```
score-merge

- Double every note an octave higher
  
  \[
  \text{print score-merge(my-score,}
  
  \text{score-transpose(my-score,}
  
  \text{keyword(pitch), 12))}
  \]

- Make my-score with 2 echoes
  
  \[
  \text{print score-merge(my-score,}
  
  \text{score-shift(my-score, 0.1),}
  
  \text{score-shift(my-score, 0.2))}
  \]

score-append

- Play my-score as is, then transposed up 1 step, then up another step
  
  \[
  \text{print score-append(my-score,}
  
  \text{score-transpose(my-score,}
  
  \text{keyword(pitch), 2),}
  
  \text{score-transpose(my-score,}
  
  \text{keyword(pitch), 4))}
  \]
score-select

- A predicate that returns true when pitch is less than 70

```lisp
define function not-very-high(time, dur, expr)
  return expr-get-attr(expr, keyword(pitch), 100) < 70
```

- Select all notes with pitch < 70 and time >= 2

```lisp
print score-select(my-score, quote(not-very-high), from-time: 2)
```

score-filter-length,

score-stretch-to-length

- score-filter-length: remove any note that *ends* after some time.
- Result will not extend beyond 2.4s:

```lisp
print score-filter-length(my-score, 2.4)
```

- score-stretch-to-length: adjust score to have a given length.
- Last event in score will end at 5s:

```lisp
print score-stretch-to-length(my-score, 5.0)
```
score-filter-overlap

- Reduce score to a monophonic texture
  - No overlapping notes/events
  - Removes any event with a start time less than the previous event’s end time

print score-filter-overlap(my-score)

score-apply

- Transform each event using a function
define function add-accents(time, dur, expr)
begin
  ; if the pitch: attrib. of the expr is greater than 70 …
  ; … then modify expr to have :accent 100
  if expr-get-attr(expr, keyword(pitch), 70) > 70 then
    set expr = expr-set-attr(expr, keyword(accent), 100)
  ; whether or not expr was changed, form a new note
  ; by combining time, dur, and expr into a list
  return list(time, dur, expr)
end

; now apply the function to a score
print score-apply(my-score, quote(add-accents))
score-adjacent-events

; a predicate that returns true when pitch is less than 72
define function not-very-high(expression)
    return expr-get-attr(expression, :pitch, 100) < 72

; a function of 3 notes – extend duration of current
; note to the starting time of the next note
define function adjust-durations(prev, cur, next)
begin
    if not-very-high(event-expression(cur)) & next then
        return event-set-dur(cur, event-time(next) –
        event-time(cur))
    else return cur
end

exec score-play(score-adjacent-events(my-score, quote(adjust-durations)))

Composition: Some Guidelines

• Vocabulary
  • Rhythm
  • Melody
  • Harmony
  • Timbre
  • Texture

• Organization
  • Structures
  • Elaboration
  • Ornamentation
  • Contrasting elements
  • Gestures
**Gesture Example**

- Consider this “gesture”:
  - Rhythm: Increasing tempo
  - Melody: Upward melodic contour
  - Harmony: Increasing dissonance
  - Timbre: Progression toward “thinner” sound
  - Texture: Shorter, lighter, busier

- So, organization (structure) transcends vocabulary (the space of variation)

---

**Putting This Into Practice**

- Find an interesting manipulation
- Create a manipulated sound
- Consider repeating it: repetition builds suspense and tension (Xenakis)
- Intensify or vary the manipulation.
- Introduce something new before things get too obvious.
- Variation and development also build tension. Returning to earlier material brings closure.