Implementing a Polyphonic MIDI Software Synthesizer using Coroutines, Realtime Garbage Collection, Closures, Auto-allocated Variables, Dynamic Scoping, and Continuation Passing Style Programming

Kjetil Matheussen

Norwegian Center for Technology in Music and the Arts (NOTAM)

May 1, 2010
1 Introduction
2 Basic MIDI Soft Synth
3 Realtime Memory Allocation and Garbage Collection
4 ADSR
5 Reverb
6 CPS Sound Generators. (Adding stereo reverb and autopanning)
7 Summary
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample

- Demonstrated by implementing MIDI software synthesizers

- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample
- Demonstrated by implementing MIDI software synthesizers
- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample
- Demonstrated by implementing MIDI software synthesizers
- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample

- Demonstrated by implementing MIDI software synthesizers

- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample

- Demonstrated by implementing MIDI software synthesizers

- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample
- Demonstrated by implementing MIDI software synthesizers
- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample

- Demonstrated by implementing MIDI software synthesizers
- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample
- Demonstrated by implementing MIDI software synthesizers
- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Introduction

- Audio programming techniques
  - CS, not DSP
  - Low-latency (hard realtime)
  - Sample-by-sample

- Demonstrated by implementing MIDI software synthesizers

- Snd-RT
  - CLM
  - Stalin Scheme Compiler
Techniques

- Coroutines
- Garbage collection
- Closures (and functional programming!)
- Auto-Allocated Variables
- Dynamic Scoping
- Continuation Passing Style Programming
Techniques

- Coroutines
- Garbage collection
- Closures (and functional programming!)
- Auto-Allocated Variables
- Dynamic Scoping
- Continuation Passing Style Programming
Techniques

- Coroutines
- Garbage collection
- Closures (and functional programming!)
- Auto-Allocated Variables
- Dynamic Scoping
- Continuation Passing Style Programming
Introduction

Techniques

- Coroutines
- Garbage collection
- Closures (and functional programming!)
- Auto-Allocated Variables
- Dynamic Scoping
- Continuation Passing Style Programming
Techniques

- Coroutines
- Garbage collection
- Closures (and functional programming!)
- Auto-Allocated Variables
- Dynamic Scoping
- Continuation Passing Style Programming
Techniques

- Coroutines
- Garbage collection
- Closures (and functional programming!)
- Auto-Allocated Variables
- Dynamic Scoping
- Continuation Passing Style Programming
Techniques

- Coroutines
- Garbage collection
- Closures (and functional programming!)
- Auto-Allocated Variables
- Dynamic Scoping
- Continuation Passing Style Programming
Why demonstrating a MIDI software synthesizer?

1. Both audio rate and control rate.
2. Variable polyphony.
3. Data allocation.
Why demonstrating a MIDI software synthesizer?

1. Both *audio rate* and *control rate*.
2. Variable polyphony.
3. Data allocation.
Why demonstrating a MIDI software synthesizer?

1. Both *audio rate* and *control rate*.
2. Variable polyphony.
3. Data allocation.
Why demonstrating a MIDI software synthesizer?

1. Both *audio rate* and *control rate*.
2. Variable polyphony.
3. Data allocation.
Why demonstrating a MIDI software synthesizer?

1. Both audio rate and control rate.
2. Variable polyphony.
3. Data allocation.
Implementing what is probably the simplest type of MIDI Soft Synth.
Basic MIDI Soft Synth

(<rt-stalin>
  (range note-num 0 128
   (define phase 0.0)
   (define volume 0.0)

   (sound
    (out (* volume (sin phase)))
    (inc! phase (midi->radians note-num)))

   (spawn
    (while #t
      (wait-midi :command note-on :note note-num
        (set! volume (midi-vol)))
      (wait-midi :command note-off :note note-num
        (set! volume 0.0))))))

Kjetil Matheussen (NOTAM)
Rollendurchmesserzeitsammler
Example

(define-stalin (softsynth)
 (while #t
   (wait-midi :command note-on
     (define phase 0.0)
     (define tone (sound
       (out (* (midi-vol) (sin phase)))
       (inc! phase (midi->radians (note-num)))))
     (spawn
       (wait-midi :command note-off :note (midi-note)
         (stop tone)))))))

(<rt-stalin>
 (softsynth))
Need to remove clicks when starting and stopping tones.
Example

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (spawn
        (define volume (midi-vol))
        (define phase 0.0)
        (define adsr (make-adsr :a 20:-ms :d 20:-ms :s 0.2 :r
          (define tone (sound :while (-> adsr is-running)
              (out (* volume
                (-> adsr next)
                (sin phase))))
              (inc! phase (midi->radians (midi-note))))
              (wait-midi :command note-off :note (midi-note)
                (-> adsr stop)))))))
1. Dynamic scoping
2. Auto-Allocated variables
1. Dynamic scoping

2. Auto-Allocated variables
1. Dynamic scoping
2. Auto-Allocated variables
(define-stalin (reverb input delay-time)
  (delay :size (* delay-time (mus-srate)))
  (+ (comb :scaler 0.742 :size 9601 allpass-composed)
      (comb :scaler 0.733 :size 10007 allpass-composed)
      (comb :scaler 0.715 :size 10799 allpass-composed)
      (comb :scaler 0.697 :size 11597 allpass-composed)
    :where allpass-composed
      (send input :through
        (all-pass :feedback -0.7 :feedforward 0.7)
        (all-pass :feedback -0.7 :feedforward 0.7)
        (all-pass :feedback -0.7 :feedforward 0.7)
        (all-pass :feedback -0.7 :feedforward 0.7))))
Simple Stereo Reverb

+-- (reverb 0.13) --> out ch 0
/  
(softsynth)--><
\  
+-- (reverb 0.11) --> out ch 1
CPS Sound Generators. (Adding stereo reverb and autopanning)

1. Sound Generators, inspired by Faust / BDA
2. CPS is a programming technique
3. CPS means that no function will ever return
4. CPS provides a simple way to support more than one output
CPS Sound Generators. (Adding stereo reverb and autopanning)

1. Sound Generators, inspired by Faust / BDA
2. CPS is a programming technique
3. CPS means that no function will ever return
4. CPS provides a simple way to support more than one output
CPS Sound Generators. (Adding stereo reverb and autopanning)

1. Sound Generators, inspired by Faust / BDA
2. CPS is a programming technique
3. CPS means that no function will ever return
4. CPS provides a simple way to support more than one output
CPS Sound Generators. (Adding stereo reverb and autopanning)

1. Sound Generators, inspired by Faust / BDA
2. CPS is a programming technique
3. CPS means that no function will ever return
4. CPS provides a simple way to support more than one output
CPS Sound Generators. (Adding stereo reverb and autopanning)

1. Sound Generators, inspired by Faust / BDA
2. CPS is a programming technique
3. CPS means that no function will ever return
4. CPS provides a simple way to support more than one output
Seq Operator

(Seq (between -1.0 1.0)
 (* 0.5))

->

(let ((generator0 (lambda (kont0)
                        (kont0 (between -1.0 1.0)))))
 (generator1 (lambda (arg1 kont1)
                (kont1 (* 0.5 arg1)))))

(lambda (kont2)
 (generator0 (lambda (result0)
                (generator1 result0 kont2)))))
Par Operator

\[
(\text{Par} (* -1) \\
(* 1))
\]

\[\Rightarrow\]

\[
(\text{let} ((\text{generator} 0 (\lambda (\text{arg} 1 \ \text{kont} 0) \\
(\text{kont} 0 (* -1 \ \text{arg} 1))))))
(\text{generator} 1 (\lambda (\text{arg} 1 \ \text{kont} 1) \\
(\text{kont} 1 (* 1 \ \text{arg} 1))))))
(\lambda (\text{input} 2 \ \text{input} 3 \ \text{kont} 1) \\
(\text{generator} 0 \ \text{input} 2 \\
(\lambda (\text{result} 0) \\
(\text{generator} 1 \ \text{input} 3 \\
(\lambda (\text{result} 1) \\
(\text{kont} 1 \ \text{result} 0 \ \text{result} 1)))))))))}
Auto-Allocated variables in CPS Generators

(Seq (all-pass :feedback -0.7 :feedforward 0.7))

->

(let ((generator0 (let ((var0 (make-all-pass :feedback -0.7 :feedforward 0.7)))
                        (lambda (kont input)
                            (kont (all-pass var0 input))))))

    (lambda (input kont)
        (generator0 input kont)))
Other CPS Generators

Split, Merge, Identity, Cut, Sum, Prod, Lambda, Buffer, Counter, Read-table, Sin, Incrementer, Osc and In.
Final version of the MIDI Soft Synth.
Summary

- Garbage collection, Functional Programming, Coroutines, Faust
Questions.
Last slide