Projects and Tools in Clojure

Jens Mehler - March 31, 2014

Overtone

Contents

Chapter 1

Introduction

This document shall give a short but brief overview of what is and can be done with Overtone. It shall explain how Overtone works[1] and how you can get a grip on programming music with it. The main focus is on stringed-synth, metronomes and drum-synth[2]. In the end we can code some nice sounds and rythms.

1.1 Overtone

Overtone is written in Clojure. The idea behind it is to use synthesizers to create sampled music and build whole instruments. Since playing music is a real-time application and the JVM may put a big lag into the whole show the project actually uses another tool to emit the audio. This tool is an audio-engine called SuperCollider.

1.2 SuperCollider

The SuperCollider environment uses its own programming language for real time audio synthesis and algorithmic composition. It's basically a client-server application. Every client sends its commands to the server which processes these commands and emits the audio described in them.

1.3 Overtone and SuperCollider collaboration

Overtone acts as such a client. It uses the Java Virtual Machine to connect to the SuperCollider-Server and to send it the commands to produce the sounds, create the instruments and so forth. An overview of how SuperCollider works would blow this documentation out of proportion therefore have a look at the SuperCollider website[5].

[1] At least in general

[2] Still some other topics are covered because that's how you get started - by making noise and working your way up

Chapter 2

Entering the world of music?

After we got some very basic theory covered we can get started - with more theory. As described above, Overtone is nothing more than a client to the SuperCollider-Server - but where is that server? The answer is simple: On your device or somewhere in your network. For this project I used a local installation of SuperCollider. In addition to that Overtone is shipped with a SuperCollider Server inside. There we have it: Two different kind of servers - the internal and the external. As always there is a downside on using either one of them.

2.1 Different kind of servers

Let's have a look at the difference between the internal and the external SuperCollider-Server. Both of them are working nearly flawlessy but each of them has its own pro and cons.

- Internal Server
  - Pros
    - Doesn't have external dependencies
    - Faster retrieval of audio buffers
  - Cons
    - Doesn't work on all OSes
    - Crashes the whole JVM if SuperCollider crashes
- External Server
  - Pros
    - More stable than internal server
    - Works on all OSes
  - Cons

* Slower access to audio buffers

As a rule of thumb you can say that if you want to get started really fast you can use the internal server and code your time away. Be aware that errors might crash the JVM.
If you want a more professional setup that is robust and doesn't crash your whole development environment you have to use the external server. The external server is basically a separate installation of SuperCollider which you have to start before connecting Overtone to the server.

2.2 Getting connected to ScSynth

After starting your Clojure-REPL it will wait for commads.
You start the internal ScSynth by entering:

```
user=>(use 'overtone.live)
```

This will load Overtone and start the internal SuperCollider-Server.

To start the external server do the following:

```
user=>(use 'overtone.core)
user=>(boot-external-server)
```

This will load the Overtone-core and start the external SuperCollider-Server.
After those commands Overtone will welcome you

Figure 2.1: Welcome to Overtone

```
--> Loading Overtone...
--> Booting internal SuperCollider server...
--> Connecting to internal SuperCollider server...
--> Connection established
    _____                    __
   / __  /_  _____  _____/ /_____  ____  ___
  / / / / | / / _ \/ ___/ __/ __ \/ __ \/ _ \
 / /_/ /| |/ /  __/ /  / /_/ /_/ / / / /  __/
 \____/ |___/\___/_/   \__/\____/_/ /_/\___/

   Collaborative Programmable Music. v0.9.1

Hello <username>, <random welcome sentence>.
```

Now we are ready to do some ... music?

Chapter 3

Entering the world of music!

With Overtone loaded and the REPL awaiting commands we are theoretically ready to code music. But what is music[1]? In a very abstract way you can say 'Music is noise that sounds good'. Music needs timing to sound good. You can play random notes or chords which are just some sin-waves but that isn't what music is. Well basically that's music but it doesn't have soul if you do it that way. There is more to it than a scientific approach[2]. Not making this too complicated - let's say we want to make some noise.

A warning to everyone - fooling around with noise can and will impare your hearing! Synthesizers normally don't have a trampoline installed that protects your ears if the noise gets too loud!
Well we are talking Overtone and Sam Aaron implemented this save cushion for your hearning. It might get loud but not too loud to destroy your hearing - anyway be very careful! Don't use a headset if you test this stuff. If you do for heavens sake lower your volume!!

3.1 Noise generation

Overtone has a lot of predefined noise-generators which we can use to output our first sound. Let's do something simple. Let's play a sin-wave at 440Hz[3]. To understand what is going on here, a quick example:

```
1 (demo (sin-osc))
```

The demo functions takes a synthesizer definition and creates the SuperCollider definition for it. It will automatically play this for an already defined time. By default it creates a sin-wave of 440Hz.

```
(demo (sin-osc 440))
```

Compare the outputs: They are equal.
Here we see that we can give the demo-function a definition of synthesizer with an argument. The first argument is the frequency.

```
(demo 5 (sin-osc))
```

This will play the sin-wave for exactly five seconds. The predefined default value is two seconds.

What exactly is this '(sin-osc)'. This is called a UGen (Unit-Generator) and is the most basic block for a SuperCollider-Synth. It is used to process and controll signals within the server. Any

[1] This is not trivial and I will skip most of the theory behind music
[2] Music is not science it's more. Music is a feeling - it has emotions and lives
[3] this is not randomly picked

synthesizer is build upon these UGens. You can control the number of input and output-channels as well. Let' have a look at that.

```
(defsynth mysynth [freq 440]
  (out 0 (sin-osc freq))
  (out 1 (saw freq)))

(mysynth)
;user=> (load-string "(mysynth)")
;#<synth-node[loading]: music.core/mysynth 3161>
(kill mysynth)
(kill 3161)
```

This will create a synthesizer with two output-channels (aka stereo). It will output a sin-wave on the left and a saw-wave on the right.
'kill' is pretty straight forward: As you can hear 'mysynth' is playing forever and believe me it will play until your speaker die or your power goes down. 'kill' takes one argument - this is the synthesizer to be killed. If we start 'mysynth' multiple times different instances will be created and they will all spill some noise. We can use 'kill' to either kill one instance with '(kill ¡id¿) or kill all instances with (kill mysynth)

We can even define different instruments which we can use.

```
(definst foo [freq 220] (saw freq))
(foo)
(mysynth)
(stop) ; stops all sound playback
```

Inst creates the SuperCollider Synthesizer definition and loads it into the SuperCollider server. It returns a function to start this synthesizer. After those commands the 'foo' synthesizer and 'mysynth' are emitting sounds until we either kill them or stop the whole playback.

3.1.1 Amplitude modulation and behavior control

While the synthesizers are playing it is possible to change their behaviour. Let's define an instrument that uses a saw-wave synthesizer which emits a frequency of 440Hz with and amplitude of 0.3.

```
(definst bar [amp 0.3 freq 440] (* amp (saw freq)))
(bar 1)
(ctl bar :amp 0.5) ;modify the amplitute of bar
(ctl bar :freq 220) ; modify the frequency of bar
(kill bar)
```

What will happen is the following: While the synthesizer is running it will emit a saw-wave of 440Hz. This is represented as a stream of floating-point values from -1 to 1. If we multiply that value with another we can control the volume of the signal.
This can be done even when the synthesizer is emmiting sound by using the 'ctl' function.

3.1.2 Chaining UGens

Whenever a UGens takes an argument it can also take another UGen which controls this argument.

```
(definst trem ""
  [freq 420 depth 10 rate 6 length 3]
    (* 0.5
       (line:kr 0 1 length FREE)
       (saw (- freq (* depth (sin-osc:kr rate))))))
```

```
(trem)
```

What happens here is that we feed and sin-wave UGen into the saw-wave UGen. The sin-wave controls the frequency of the saw-wave while the line UGen is used to stop the whole synthesizer after a specified time.

3.2 Notes and chords

After playing around with different ways of emitting sound by producing different frequencies with different UGens we can take a very short look at what notes and Chords are. We will use them later on to program some short pieces of music. For this, visualize a guitar. A guitar has six strings. Each string, when picked emits a different sound. We call those notes - those notes each have a specific frequency. Those notes are E A D G B e.
In short we can say a note is a specific frequency. Futhermore a guitar has frets, putting your finger on a string at a fret makes the frequzency lower or higher. You can play the guitar note by note by just picking on note after the other.
Sometimes you can't express yourself with just notes and want something more. I will skip the whole music-theory part of this and explain in short what a chord is. A chord is a combination of different notes played over multiple strings at the same (nearly the same - the gap is pretty short) time. The frequencies start to overlay each other and produce a frequency which is the sum of all single frequencies played together.
So far for the theory. That's all we need to know

- a note = a frequency
- a chord = multile frequencies which overlay each other

Let's look at this in action. We need something that can give us a sound at a specific frequency.

```
(definst saw-wave [freq 440 attack 0.01 sustain 0.4 release 0.1 vol 0.4]
  (* (env-gen (lin attack sustain release) 1 1 0 1 FREE)
     (saw freq)
     vol))
```

The important part of this instrument is the 'env-gen'. It's an envelope generator which modifies the frequenzy that is emitted. It modifies the frequency like this. The attack is the time of the rising flank of the signal, the sustain is the time the signal stays at it amplitude and the release is the time it takes to fall back to zero. This will modify our saw-wave to slighty trail in, stay persistent for a while and slightly fade out gain.
We can now use this to play actual notes - they won't sound like a note from a guitar but that is something else we can take care of later.

```
;; We can play notes using frequency in Hz
(saw-wave 440) ; This is A4
(saw-wave 523.25) ; This is C5
(saw-wave 261.63) ; This is C4
```

Notes cannot only be represented by a specific frequency but also by MIDI values. Simply put - each note has a specific MIDI value as which it is represented. This is also implemented in Overtone[1] and therefore we can use the MIDI-values to play notes.

[1] it is actually possible to connect a MIDI-device to Overtone - if you have one handy try it out.

```
;; We can also play notes using MIDI note values
(saw-wave (midi->hz 69)) ; This is A4
(saw-wave (midi->hz 72)) ; this is C5
(saw-wave (midi->hz 60)) ; This is C4
```

'midi-hz ¡n¿' translates the midi-value to a freqzenzy which is put the input for our saw-wave.
Okay, simple as that but who wants to remember a frequency or a plain number, look at the code above and you can see that each note has a name, a frequency and a midi-value. They all represent the same thing!
Now, wouldn't it be a lot easier to just write '(play :A4)' instead of the stuff above? Turns out that it's possible.

```
;; Let's make it even easier
(defn play [music-note]
    (saw-wave (midi->hz (note music-note))))

;; Great!
(play :A4)
(play :C5)
(play :C4)
```

There we go. What happens is that (note ¡note¿) return the midi-value of a note which is mapped to a frequency which is passed to your saw-wave which plays that frequency.

After notes we can do what I explained above - we can combine notes to play a chord. Don't worry Overtone as some nifty functions to help us create a chord. Actually it's called 'chord'. We just need to take care of one tiny bit that I mentioned above. Playing notes at the same time.

```
(defn play-chord [chord]
  (doseq [note chord] (play note)))
```

This function will play the given notes as a sequence, they will overlay each other and we will hear a chord.

As I said I will skip most os the music theory part just accept that this below is a chord.

```
(chord :C4 :major) ; this creates the sequences of midi-notes to be played
;user=> (load-string "(chord :C4 :major) ; this creates the sequences of midi-notes to
    be played")
;(60 64 67)
;feed it to play-chord
(play-chord (chord :A4 :major))
```

With this we have played our first chord and understood how it is handled. We can go one step higher in the abstraction and use an instrument to make music.

3.3 Stringed Synthesizer aka Guitar

Since I wanted to see what is possible with Overtone I started programming an insturment that should behave like a guitar[1]. After writing several lines of code something in my head went like 'Use the source Luke - use the source'. I rushed over to the overtone repository[3] and started looking if someone already had that idea. It turned out they had, there was already a functioning string synthesizer which

[1] I play electric guitar myself so this was the obvious choice

I could use and therefore could destroy a week of wrecking my brain on how to describe a guitar in Overtone.[1].

3.3.1 Creating a stringed synthesizer

To use a string synthesizer you can define a variable that uses the pedefined guitar from 'overtone.synth.stringed' this instrument named 'guitar' has six strings and isn't freed when all strings go silent.

```
(def g (guitar))
```

From now on you can use 'g' instead of 'guitar'. This instrument has several options that can be set. By default you will get an accoustic guitar without any effects. You can put on many effects for your guitar and make it sound pretty[2]. Explaining all those possible effects would cover more than we need and therefore I picked some to give a small overview:

Table 3.1: Stringed Synthesizer Effects and causes

Effect	Cause
distortion	Causes the guitar to sound distorted
pre-amp	regulates the input to the real amplifier
amp	regulates loudness

In addition we can define some reverbs but that should be expirienced not explained.

```
(ctl g :pre-amp 6.0 :amp 1.0 :distort 0.9)
```

We use the already known function 'ctl' to set those control-parameters for our guitar.

3.3.2 Playing a note

As described a few pages above you can play a note on a guitar by picking one of the strings or putting your finger on a fret and pick that string.

```
(guitar-pick g 1 0) ; A
```

With this we will play the second string on our guitar which is the A-string. Overtone starts numbering the strings from 0 to max-string which is defined by the stringed-synth. Something different to a real guitar is that we have an endless fretboard. Yes, we can play fret 42! The second argument is the fret-number which modifies the frequency of the emitted sound. There is an optional third argument, this is the starting time of the note[3]. This is important for note-progressions which is basically the timed playing of notes in a specific order.

3.3.3 Playing a chord

Already explained is that a chord is nothing else than several notes played in a fast progression. The string synth makes this even easier for us because it comes with a handy function.

[1] Always look at the source and see if there is something you can use
[2] I recomment distortion
[3] my presentation material covers this

```
;(guitar-strum the-guitar the-chord move timing)
(guitar-strum g [-1 3 2 0 1 0] :down 0.01) ; C
(guitar-strum g :C :up 0.01)
```

This function either takes a vector which describes the fret-positon of the corresponting string or a already predifined short-name for that vector. Furthermore it makes a slight difference whether you play the chord as an up-stroke or as a down-stroke. The last argument is the time that it waits before it plays the next note from the vector.
Like 'guitar-pick' this function has another optional argument which represetns the starting time of the chord(note-progression).

3.4 Metronomes

While programming a short piece of music I didn't know about the metronomes. Turns out that a whole day was wasted because of getting the right timing. And that's what metronomes are for. They tick in beats endlessly in the background. There is nothing more to them. They just tick at a given speed - the 'speed' is actually called 'beats-per-minute' (bpm).
Let's define an metronome in overtone.

```
(def one-twenty-bpm (metronome 120))
```

Now in theory we should get a tick every half second. But it doesn't! Take the following code

```
(def kick (sample (freesound-path 2086)))

; this function will play our sound at whatever tempo we've set our metronome to
(defn looper [nome sound]
    (let [beat (nome)]
        (at (nome beat) (sound))
        (apply-at (nome (inc beat)) looper nome sound [])))

; turn on the metronome
(looper one-twenty-bpm kick)
```

You will hear a sound every second[1]

3.4.1 Working with metronomes

We have to compensate the error from above from now on. Let's have a look at what the metronome does when when we call it.

```
(def metro (metronome 120))
(metro) ; returns the current tick - do this multiple times
(metro 600) ; returns the time the beat comes up
```

3.5 Slide guitar

Since I wanted to create something on my own I decided to try to implement a function that realises sliding notes on a guitar. What is that anyway? Well think about it this way: You play a note on your guitar and move the finger up or down the fretbroad. It sounds great and it's fun to play, still

[1] I have informed the Overtone community about this problem and am currently awaiting an answer about this matter.

there is no function in the stringed-synth which allows this behaviour. What we can do with the string synth is - picking a note or chord and muting it. In theory all that's needed to be done is to allow the stringed-synth to move the notes on the fretboard up and down play playing the note. A down-move is i.e. from fret 17 to fret 10. A up-move is i.e. from fret 10 to fret 17.

A quick look at the definition of stringed-synth shows that something like what we need already happens in function 'pick-string'. This function sets the note on a specific string, it can even mute it. Let's build up on that.

```
(defn- fret-to-note
  "given a fret-offset, add to the base note index with special
  handling for -1"
  [base-note offset]
  (if (>= offset 0)
    (+ base-note offset)
    offset))

(defn- mkarg
  "useful for making arguments for the instruments strings"
  [s i]
  (keyword (format "%s-%d" s i)))

(defn set-fret [the-inst string-index fret]
  "Sets fret for the-inst on string-index"
  (let [the-note (fret-to-note (nth guitar-string-notes string-index) fret)]
    (if (= the-note -1) ;mute it
      (ctl the-inst (mkarg "gate" string-index) 0)
    )
    (if (>= the-note 0)
      (ctl the-inst (mkarg "note" string-index) the-note) ; set other note on string
    index
    )
  )
)
```

The 'fret-to-note' and 'mkarg' are non-public functions from overtone.synth.stringed and to make them available in my namespace I simple had to copy them over.
Let's have a look at the 'set-fret'-function. It takes three arguments, the instrument to be modified, the string-index of that instrument and the fret that should be applied to the string-index.

It can be used like this.

```
(def sg (guitar))

(ctl sg :pre-amp 5.0 :distort 0.96
     :lp-freq 5000 :lp-rq 0.25
     :rvb-mix 0.5 :rvb-room 0.7 :rvb-damp 0.4)

(guitar-pick sg 0 1)
(set-fret sg 0 2) ; one slide up
(set-fret sg 0 3) ; another slide up
(set-fret sg 0 -1) ; mute
```

Now we are able to program slides in a more or less very uncomfortable way. There is no timing there and you need more than one line to do a slide! Let's solve that problem

```
(defn slide-string
  "slides the-string of the-inst from fret start-fret to fret end-fret.
  Every note inbetween sounds for duration time.
  if keep is set the last note will be fret end-fret
  otherwise the string gets muted"
  [the-inst the-string start-fret end-fret start duration keep-note]
```

```
  (at start (guitar-pick the-inst the-string start-fret))
  (let [i (atom 1)] ; used to calculate the offset between the sub-slides
    (doseq
      [fret (if (< end-fret start-fret)
              (reverse (range end-fret (inc start-fret)))
              (range start-fret (inc end-fret))
            )
      ]
      (at (+ (* @i duration) start) (set-fret the-inst the-string fret))
      (if (and (= end-fret fret) (zero? keep-note) )
          (at (+ (* (inc @i)  duration) start) (set-fret the-inst the-string -1))
      )
      (swap! i inc)
    )
  )
)
```

The function 'slide-string' takes seven arguments. First off the instrument to which it shall apply the changes. Second the string-index of that instrument. Third and fourth are the start and end of the slide. Futhermore it needs a starting time and duration value for the notes between the slides and last but not least we can keep the last note alive. If the start-fret is bigger than the end-fret the function automatically assumes that the user wants to do a down-slide.

```
(slide-string sg 2 3 9 (now) 50 0)
```

With this we do a slide from fret three to fret nine now. Each note will be played for a duration of 50ms the last note will not be kept alive. Being able to keep the last note alive results some better sounds when playing with slides.

```
(slide-string sg 2 9 3 (now) 50 1)
```

And the down-slide from fret nine to fret three with keeping that last note alive.

Chapter 4

Recording

This is something I find very interesting and should definetly be mentioned. It's possible to save your music in a .wav file on your system. There are two different ways to reach this goal. The most simple one is

```
(recording-start "~/Desktop/foo.wav")
;;make some noise...
(demo (pan2 (sin-osc)))
(recording-stop)
```

You can put everything in between those two commands - everything that creates a sound will be saved[1]. Still if you don't play sound it will just fill the file with no-sound at all.

To work around that you can use an audio-buffer.

```
(def b (buffer 44100))
(defsynth bong [note 60 velocity 0.5 attack 0.01 decay 1]
  (let [freq (midicps note)
        src (+ (sin-osc freq)
               (* 0.5 (sin-osc (* 2.1 freq)))
               (* 0.4 (sin-osc (* 4.9 freq)))
               (* 0.3 (sin-osc (* 7.1 freq)))
               (* 0.2 (sin-osc (* 8.9 freq)))
               (* 0.1 (square (* 1.3 freq)))
               (* 0.1 (square (* 4.2 freq))))
        env (env-gen (perc attack decay) :action FREE)]
    (record-buf (* velocity src env) b :action FREE :loop 0)))
(bong)
(buffer-save b "~/Desktop/bong.wav")
```

[1] I will record my example for the stringed synth during the presentation

Chapter 5

Conclusion

Even though entering the world of Overtone is quite a big step for a beginner[1], still the fun you have while coding and listening to what you are coding totally makes up for this. There are many more things you can do with overtone. Pay the wiki[2] a visit and have fun.

[1] You should now some Clojure - makes it a lot easier

Chapter 6

Special Thanks

Special thanks go to the following people

Cissi Kain Nielsen	for proof reading this document
The Overtone Community	for creating this great piece of Software

Chapter 7

Obtaining the source

All code and this documentation can be downloaded from:

```
https://github.com/jensmehler/clojure-tools-overtone.git
```

Glossary

Clojure A functional LISP-like Programming Language running in the Java Virtual Machine. 3, 18

JVM Java Virtual Machine. 3, 5, 18

MIDI Musical Instrument Digital Interface: Protocol for music instrument communication. 8, 18

Overtone Framework written in Clojure for programmig music. 3, 18

REPL Read-eval-print loop. 5, 18

ScSynth SuperCollider Synthesizer - The SuperCollider Sound-Server. 5, 18

UGen Unit Generator, a basic building block of a synthesizer. 6, 18

List of Figures

List of Tables

Bibliography

[1] Overtone Cheat Sheet:
Overtone Maintainer
https://sourceware.org/newlib/

[2] Overtone Wiki:
Overtone Maintainer
https://github.com/overtone/overtone/wiki

[3] Overtone Repository
Overtone Maintainer
https://github.com/overtone/overtone/tree/master/src/overtone

[4] Overtone Examples:
Overtone Maintainer
https://github.com/overtone/overtone/tree/master/src/overtone/examples

[5] SuperCollider Website:
http://supercollider.sourceforge.net