The SysSon Platform

Technical Report TR-2016-11-1
Institute of Electronic Music and Acoustics, Graz
(Status: in progress)

Hanns Holger Rutz

November 2016

1 QBO - Blob Sonification

After some initial experiments with frequency modulation to indicate blob slice height, it was decided to try out other forms of timbre modification such as wave-shaping. SuperCollider provides wave-shaping by means of the `Shaper` UGen, typically with a buffer prepared with Chebychev polynomial functions. However, it seems not possible to continuously fade in a particular timbre by altering the input signal's amplitude, as different lower partials will transitorily be attenuated. Another possibility is through the `VOsc` variable table oscillator. In order to generate the appropriate wave-tables, a graph element `BufferGen` has been added to *Sound Processes*. As `VOsc` depends on a trick of allocating multiple buffers with consecutive identifiers, and this consecutiveness is currently not possible to guarantee in *Sound Processes*, one can simply mix and blend multiple `Osc` instances manually, which has the same effect. The code is shown in Fig. 1, where the `amp` parameter is expected to be in the range from zero to one, and it scans through the different spectra.

Fig. 2 shows the sonogram of a bounce of this sonification model with the QBO blob data. The bounce can be heard at `https://soundcloud.com/syssonproject/blob-shaper161102`. The parameters are:

- time = 2002-01-16 12:00:00Z to 2016-02-15 12:00:00Z
- lon = 75.00 °W; lat = 2.50 °S
- speed = 6 months/sec, mag-max = 3, min-freq = 300 Hz, max-freq = 800 Hz
- spread-mod-depth = 1.5, spread-mod-offset = 0.3

1.1 "Importing" Timbres From a GA Process

We have been recently experimenting with the algorithmic production of sound synthesis structures, by using genetic programming of synth graphs based on a target sound and a fitness function that correlates the spectra and loudness contours of target sound and individuals in the GP populations. While this does not yield sounds close to the target sound in reasonable time and number of iterations, it has proven to be a very useful generator of interesting timbres.

Therefore, it seems useful to try and "import" timbres produced in this evolutionary algorithm to SysSon. There is a simple source code generator for the found sound structures, and the work is then to select a sound, unclutter the source code from "dead" branches (e.g. those that essentially produce silence or components that one wants to remove from the timbre), and finally identify parameters that one wants to control (e.g. fundamental frequency). The GP comes up with interesting solutions for modulating timbre, such as adding signals and then clipping the sum, producing thus co-modulations.

Fig. 3 shows the result of such an imported sound, with the sonogram for applying it in the usual QBO setup shown in Fig. 4. A bounce of this sound can be found at `https://soundcloud.com/syssonproject/blob-shaper-timbre2-161107`.

Fig. 5 shows the result of another such imported sound, with the sonogram for applying it in the usual QBO setup shown in Fig. 6. A bounce of this sound can be found at `https://soundcloud.com/syssonproject/blob-shaper-timbre3-161107`.

Fig. 7 shows the result of another such imported sound, with the sonogram for applying it in the usual QBO setup shown in Fig. 8. A bounce of this sound can be found at `https://soundcloud.com/syssonproject/blob-shaper-timbre4-161107`. Here we fade between three different mixes and with wider overlap, resulting in less discernible "stages", but perhaps also leading to more difficulties in "categorising" the sound.

```
def mkOsc(freq: GE, amt: GE): GE = {
  val oddBase  = 1f
  val evenBase = 0f
  val oddDamp  = 0.7f
  val evenDamp = 0.8f
  val numHarm  = 9
  val numBufs  = 5
  val tableSz  = 1024

  val oscs = (0 until numBufs).map { i =>
    val amps0 = Seq.tabulate(numHarm) { j =>
      val isEven = (j + 1).isEven
      val base   = if (isEven) evenBase else oddBase
      val damp   = if (isEven) evenDamp else oddDamp
      val exp    = (j / 2) * (numBufs - i)
      base * damp.pow(exp)
    }
    // first is forced to be fundamental only
    val amps = if (i == 0) Seq(1f) else amps0
    val buf = BufferGen.sine1(amps, numFrames = tableSz)
    Osc.ar(buf, freq)
  }

  val idx  = amt.linlin(0, 1, 0, numBufs - 1)
  val idxF = idx.floor
  val idxC = idx.ceil
  val wC   = idx % 1.0
  val wF   = 1.0 - wC

  val osc  = Select.ar(idxF, oscs) * wF + Select.ar(idxC, oscs) * wC
  osc
}
```

Figure 1: Generation of oscillator mix implementing blending of partial frequencies.

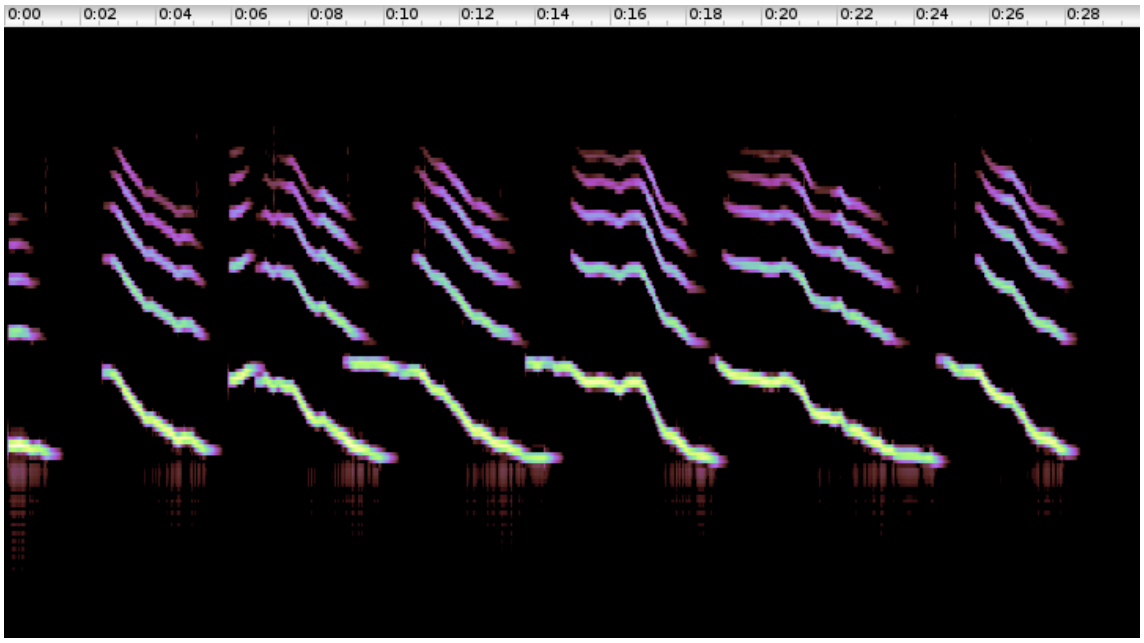

Figure 2: Sonogram of QBO sonification with blob slice height mapped to overtone spectrum

```
// ---- oscillator ----

def mkOsc1(freq: GE): GE = SinOsc.ar(freq) * 0.75

def mkOsc2(freq: GE): GE = {
  val freq_4          = freq

  val amt_1           = 0.1
  val off_1           = 0.7
  val amt_2           = 1.0
  val freq_3          = 0.6 // adds irregularity
  val freq_5          = (freq_4 * 2).min(18000) // lpf

  val lFCub_0         = LFCub.ar(freq = freq_3, iphase = 0.660289)
  val min_7           = lFCub_0.min(0.0)
  val lFDNoise0       = LFDNoise0.ar(freq_3) + off_1
  val gbmanL          = GbmanL.ar(freq = freq_4, xi = 383.95047, yi =
   383.95047)
  val min_33          = gbmanL min 0.36345935
  val blip            = Blip.ar(freq = freq_4, numHarm = 1.0)
  val plus            = blip + 0.1321
  val mix             = Mix(Seq[GE](
      lFDNoise0 * amt_1,
      min_7 * amt_2,
      min_33,
      plus
  ))
  val sig0 = LeakDC.ar(mix.clip2(1)) * 0.75
  val sig  = LPF.ar(sig0, freq_5)
  sig
}

def mkOsc(freq: GE, amt: GE): GE = {
  val numOscs = 2
  val oscs = Seq(mkOsc1(freq), mkOsc2(freq))

  val idx  = amt.linlin(0, 1, 0, numOscs - 1)
  val idxF = idx.floor
  val idxC = idx.ceil
  val wC   = idx % 1.0
  val wF   = 1.0 - wC

  val osc  = Select.ar(idxF, oscs) * wF + Select.ar(idxC, oscs) * wC
  osc
}
```

Figure 3: Oscillator mix by fading between sine and complex timbre.

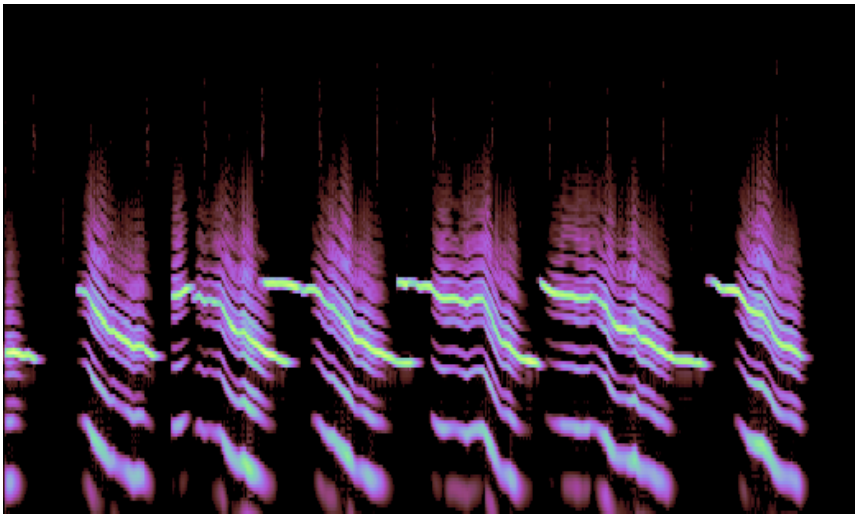

Figure 4: Sonogram of QBO sonification with fading between sine and complex timbre.

```
def mkOsc1(freq: GE): GE = SinOsc.ar(freq) * 0.75

def mkOsc3(freq: GE): GE = {
  val freq1 = freq
  val amt1  = 0.125 // modulation
  val freq2 = 13.0
  val freq3 = (freq1 * 3.0).min(18000)

  val gbmanL_0        = GbmanL.ar(freq = freq1, xi = 383.95047, yi =
   383.95047)
  val blip            = Blip.ar(freq = freq1, numHarm = 1)
  val min_3           = -0.058492452
  val difsqr          = blip difsqr min_3
  val ring2           = gbmanL_0 ring2 difsqr
  val min_4           = 0.660289 min ring2
  val min_5           = min_4 min -0.058492452
  val b               = gbmanL_0.min(0.0)
  val min_6           = b min gbmanL_0
  val a               = min_4 min min_6
  val decayTime_1_X   = QuadN.ar(freq = 0.660289, a = a, b = b, c =
   min_4, xi = min_5)
  val decayTime_1 = decayTime_1_X.clip2(770)
  val in_3            = LeakDC.ar(decayTime_1)
  val min_10          = min_6.min(0.0)
  val in_4            = LeakDC.ar(min_10)
  val combL           = CombL.ar(in_4, maxDelayTime = 0.0,
    delayTime = 0.0, decayTime = 0)
  val min_11          = combL min min_6
  val min_13          = min_11 min min_6
  val min_15          = min_5 min min_13
  val min_16          = min_15.min(0.0)
  val in_6            = LeakDC.ar(difsqr)
  val allpassC        = AllpassC.ar(in_6, maxDelayTime = 0.01,
    delayTime = 0.01, decayTime = decayTime_1)

  val mix0 = min_16 * amt1 + allpassC
  val mix = LPF.ar(mix0, freq3)
  LeakDC.ar(mix.clip2(1)) * 0.75
}

def mkOsc(freq: GE, amt: GE): GE = {
  val numOscs = 2
  val osc1 = mkOsc1(freq)
  val osc3 = mkOsc3(freq)
  val oscs = Seq(osc1, osc1 * 0.5 + osc3)
  val idx  = amt.linlin(0, 1, 0, numOscs - 1)
  val idxF = idx.floor
  val idxC = idx.ceil
  val wC   = idx % 1.0
  val wF   = 1.0 - wC
  val osc  = Select.ar(idxF, oscs) * wF + Select.ar(idxC, oscs) * wC
  osc
}
```

Figure 5: Oscillator mix – timbre 3.

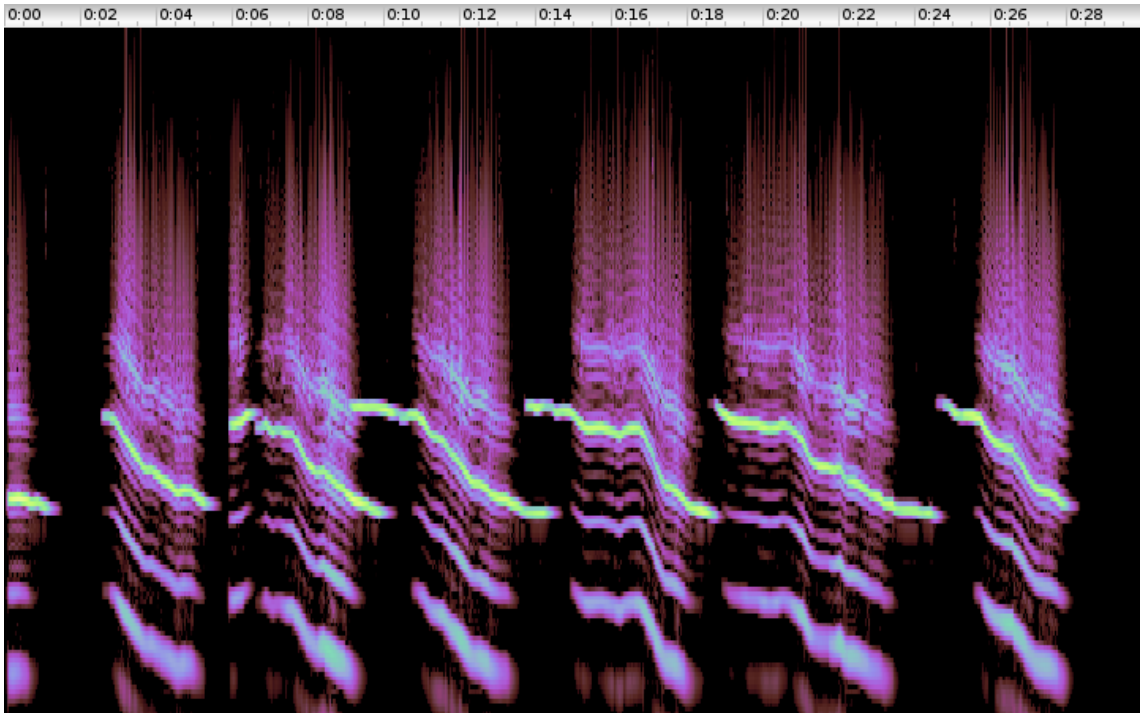

Figure 6: Sonogram of QBO sonification with fading between sine and complex timbre.

```
def mkOsc4(freq: GE): GE = {
  val freq3 = freq
  val freq2 = 3615.845
  val freq1 = 10.285505 // modultion freq
  val amt1  = 0.2 // modulation depth

  val saw_0           = Saw.ar(freq1) * amt1
  val a_2             = BrownNoise.ar.linlin(-1, 1, 0.3, 0.85)
  val latoocarfianL   = LatoocarfianL.ar(freq = freq2, a = a_2, b =
   1.5, c = 0.5, d = 0.0,
    xi = -1.8525897, yi = saw_0)
  val ratio = freq3 / (2.0 * 349.2)
  val shift            = PitchShift.ar(latoocarfianL, pitchRatio =
   ratio) * 1.5
  val mix = Resonz.ar(shift, freq3 * 1.1) * 1.41
  mix
}

def mkOsc(freq: GE, amt: GE): GE = {
  val numOscs = 3
  val osc1 = mkOsc1(freq)
  val osc2 = mkOsc2(freq)
  val osc3 = mkOsc3(freq)
  val osc4 = mkOsc4(freq)

  val mix1 = osc1
  val mix2 = osc1 * 0.5 + osc2 * 0.5
  val mix3 = osc4 + osc3 * 0.33

  val idx  = amt.linlin(0, 1, 0, numOscs - 1)
  val mix1w = mix1 * (1.4 - idx.absdif(0).clip(0, 1.4).sqrt)
  val mix2w = mix2 * (1.4 - idx.absdif(1).clip(0, 1.4).sqrt)
  val mix3w = mix3 * (1.4 - idx.absdif(2).clip(0, 1.4).sqrt)
  val osc =  mix1w + mix2w + mix3w
  osc * 0.7
}
```

Figure 7: Oscillator mix – timbre 4 (functions 'mkOsc1', 'mkOsc2', 'mkOsc3' as before).

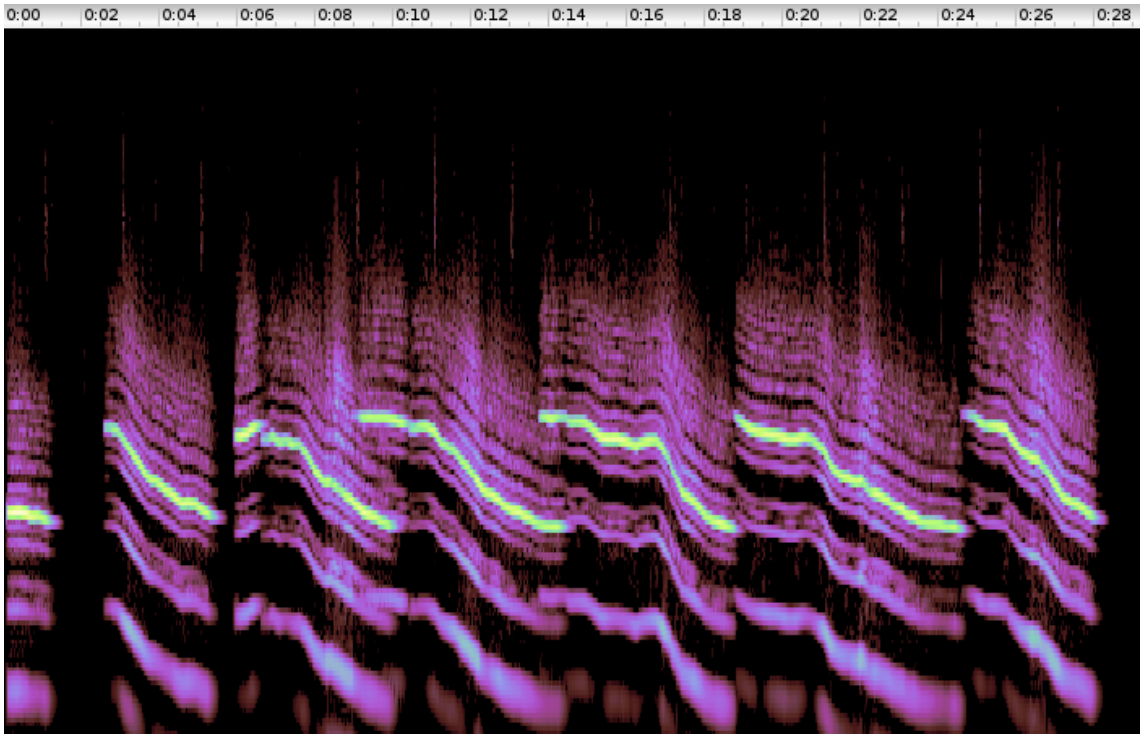

Figure 8: Sonogram of QBO sonification with fading between sine and complex timbre.