Andrea Valle
CIRMA/DAMS, Department of Humanities (StudiUm)
University of Torino
Guestprofessorship "Detmold Residence for Sound, Image & Space Design"
Hochschule für Musik Detmold
andrea.valle@unito.it

Algorithmic spatialization. Interfacing SuperCollider to the Wave field system

Abstract: The document reports back on the experience of linking the IOSONO Wave field synthesis system for virtual sound spatialization and the SuperCollider environment for algorithmic music composition and audio synthesis. Development of the discussed project happened on October 18th-26th 2016 at Erich-Thienhaus-Institut (ETI), Hochschule für Musik Detmold, in the framework of the Guestprofessorship "Detmold Residence for Sound, Image & Space Design", and was presented on October 26th at Konzerthaus Detmold, in the workshop *Algorithmic spatialization. Interfacing SuperCollider to the Wave field system*, with live usage of the IOSONO system.

Contents

1 The IOSONO Wave field synthesis

The IOSONO wave field system for 3D audio[1] implements a methodology to use secondary audio sources to recreate primary wave fields, that is, to create an audio hologram that simulates the original radiation pattern of a certain source in a certain position [1]. Wave field systems feature many loudspeakers that have to contribute to the composition of the wave front, but they are not intended to be accessed directly (as in multichannel setups like e.g. Acousmonium), rather through a dedicated software, the user defining various control parameters, the most important being the position of the virtual source. In this sense, wave field can be compared to Ambisonic [2] or VBAP [3] techniques. The system taken into account in this report is the one in use at the Erich-Thienhaus-Institut (ETI), Hochschule für Musik Detmold. Actually, two systems are available, a large one included into the Konzerthaus, that features real 3D spatialization by including loudspeakers on the ceiling, and a reduced implementation in a dedicated Studio, that allows to control source placement only on the horizontal plane.
The standard hw/sw configuration in IOSONO

[1] `http://www.iosono-sound.com/`

includes:

- audio channels (32 available in the studio, 64 in the Konzerthaus) are addressable, typically by means of a MADI soundcard;
- the soundcard can be interfaced to the IOSONO system running under Linux (direct connection, available in the Konzerthaus) or via a plug-in for Steinberg Nuendo workstation (running under the Windows operating system, required in the studio), that in turn routes the audio to IOSONO (indirect connection);

The Steinberg Nuendo plug-in is the standard editing user interface to the system. It allows to graphically define source placement by linking audio streams to space by drawing trajectories into a graphic space. While the IOSONO system works in real time, Nuendo-based editing is conceptually intrinsically related to non-real-time, as audio streams are thought of as tracks into a multitrack environment, and source position drawing in the plug-in precedes position setting ("draw, then set" mode). In short, the multitrack environment provides some GUI-based facilities by endorsing a typical audiovisual sound design/*musique concrète* interface configuration, as in these operational domains the typical working mode is editing intended as "montage" (see [4] for the term).

2 OSC interface

The IOSONO system features a second control layer that can bypass the Nuendo plug-in to address directly the system by means of OSC messages [5]. Only one type of OSC message has a specific semantics in the IOSONO system (`/iosono/renderer/verticalpan/v1/src`). Every UDP packet transmits 96 bytes of data, that set the many available parameters [6]. In this report only the most relevant ones (source identification and position) will be taken into account.
As

1. audio channels can act as placeholders for real-time audio streams as provided by the sound card;
2. OSC messages can be sent on-the-fly,

the OSC implementation provides a full real-time, interactive control over the IOSONO system. In this way, the Nuendo plug-in can be bypassed, as the system can be controlled by any software application capable of sending OSC-compliant messages.
The OSC message addresses audio channels (accessed by means of direct or indirect connection, see before) with a progressive numerical identifier (starting from 0). To sum up in plain English, a message asks the IOSONO system to create a *sound source event* from a certain audio channel in a certain position, with certain optional parameters. The situation is shown in Figure 1.

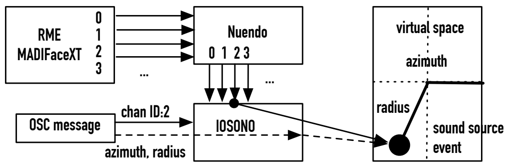


Figure 1 An OSC message in IOSONO.

The OSC message refers to a channel ID that is then streamed to the IOSONO system and placed in the virtual space according to the desired position, expressed in polar coordinates (azimuth angle and radius in meter). The IOSONO system calculates a *sound source event* by computing for a certain audio stream from a channel the opportune set of signals then driven to the loudspeaker set. It can be termed "event" because computation starts upon receiving a message and ends after a certain duration (default value: 1 s), the system applying a steep amplitude fadeout envelope (see Figure 2). Due to this time-based behaviour, in order to simulate a stationary sound source, a stream of OSC messages with e.g. a default rate of 1 msg/s has to be sent for the desired source duration.

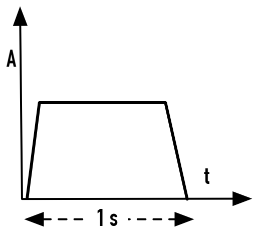


Figure 2 Sound source envelope.

OSC specification is described in the IOSONO Scene Data Protocol V2.3.1. [6]. Previous work has also been taken into account in order to understand typical/useful values for other parameters. In particular, composer Örjan Sandred previously worked on the same topic –OSC control of the IOSONO system– by interfacing Cycling'74 Max/MSP (see [7]).

3 The SuperCollider environment

SuperCollider[2] is an environment for real-time, interactive audio programming and music composition. It also features extended possibilities in GUI creation/manipulation and communication with other software/hardware by means of various options (OSC, ethernet, serial port, MIDI, Unix terminal). SuperCollider features an audio engine (scsynth) that works as a server, natively interfaced to a client application (sclang), that is also the interpreter of an Object-Oriented language. Connection to the IOSONO system is shown in Figure 3.

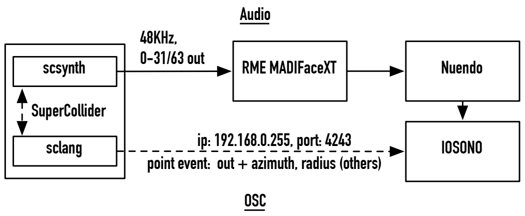


Figure 3 Information flow from SuperCollider to IOSONO.

Two layers of communication are required. Audio communication (audio streams generated in real-time by scsynth upon request by sclang) is operated by directly connecting the software to the RME MADIFaceXT audio card. Thence, streams are routed into Nuendo plug-in and/or to IOSONO. OSC communication is addressed directly by sclang to IOSONO on the local ethernet network: while addresses change between studio and Konzerthaus, the port value is typically set to 4243. Caution must be made in studio system, as there the IOSONO setup requires the client to send to `192.168.0.255`, where `255` indicates broadcast communication. The latter is possible from sclang only if a specific broadcast flag is set as `true` (see the class `NetAddr`). Also, IOSONO strictly adheres to OSC implementation that requires to declare data types for all data to be sent: data types in OSC message from sclang must be set properly in order to have it received and acted upon by IOSONO. As already said, the source ID in the OSC command identifies the audio channel (counting from 0) streamed to the source event in IOSONO.

4 Software infrastructure development

In order to operatively work with IOSONO from SuperCollider, it has been mandatory to provide a software infrastructure on which a more abstract, higher level approach could be pursued. Three classes have been developed that encapsulate basic functionalities in an efficient way (Figure 4). Class `WFSrc` represents a generic sound source. It includes a position attribute that can be set by the `xy` method. Cartesian coordinates have been chosen as a typically easier way to handle the virtual space (that in this first approach is intended as a surface, elevation still not being considered). A `WFSrc` acts as an audio placeholder that receives an audio stream externally (by other SuperCollider components), to be routed into by the `bus` attribute. From `WF-`

[2] `http://supercollider.github.io`. For general introductions, see [8] and [9].

Src, the audio stream is then routed almost transparently to an out bus (via the out attribute), in turn connected to the sound card. A basic control interface is added (play/pause, mute/unmute) to the streaming component.

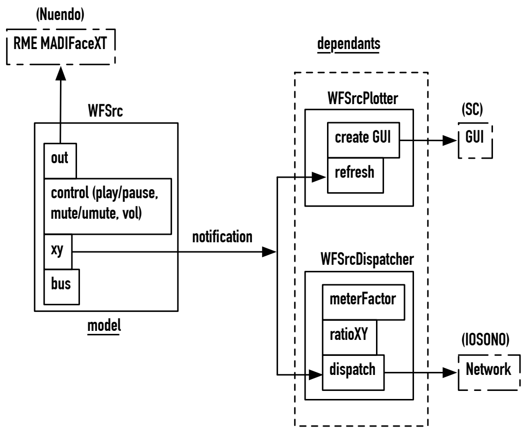


Figure 4 Classes and their relations.

GUI elements are crucial to verify the intended behaviour of sound sources and to monitor them in real-time. Class WFSrcPlotter is a basic visualization that shows sources as circles in a 2D space, and update their positions each time a source position is changed. In order to uncouple GUI (WFSrcPlotter) from data generation (WFSrc), a notification mechanism has been implemented (the so-called "Observer behavioural pattern" in OOP parlance [10], also implemented in the Model-View-Controller schema). In short, the xy method in WFSrc notifies a list of "dependants" at every call (i.e. a source has changed position). At instantiation, the WFSrcPlotter registers as a dependant of all WFSrc instances, and it is then notified about every WFSrc's position change (i.e. at every xy call). Uncoupling happens as a WFSrc may have an empty list of dependants, so its working mode is not affected by the presence of dependants (it is up to the dependants to register to their so-called "model", that is, the WFSrc). An instance of WFSrcPlotter receives a list (technically, an array) of WFSrc and registers to all, mapping their ID by scaling the hue of the color wheel by the total number of WFSrc (Figure 5). If needed ID number can be easily plotted inside the circle marker. A relevant issue in working with source positions emerges in relation to absolute metrics, as the latter is different in Studio and Konzerthaus. As a design decision, source positions have been normalized in the range $[-1.0, 1.0]$ on both axes. This facilitates to understand the positions of sources and, moreover, the latter can be always plotted in the same (squared) GUI. The GUI component is intended exclusively as a data visualizer, not as a graphical controller receving input e.g. from mouse. Although possible, such an approach is outside the scope of this work, that rather focuses on algorithmic spatialization. A third class has been developed, WFSrcDispatcher that manages OSC dispatching and basic geometric operations. While these two functions are logically autonomous (and heterogeneous), as geometric operations are minimal, a single class has been developed for sake of simplicity. The WFSrcDispatcher communicates with WFSrc following the same notification mechanism as WFSrcPlotter, so that its instantiation can be avoided while the IOSONO system is not connected: as an example, while composing, plotting is still needed but audio can be mapped to e.g. mono out for purely sound testing, thus dispatching can be avoided. On instantiation, WFSrcDispatcher receivess a list of WFSrc to which registers (again, reacting to the xy message), and the ip address and the port where to dispatch messages (the IOSONO address) (Figure 4). Basic geometric setting includes a metric factor (meterFactor) that multiplies the normalized space values in order to obtain actual measures in meters, and a ratio of the axes (ratioXY), that becomes necessary as the actual space is not squared. On a call to xy in a WFSrc, the WFSrcDispatcher applies factor and ratio to the received new coordinates, converts them into polar coordinates (as required by the IOSONO system), pack the correct OSC message and dispatch it to the passed address[3].

The following code shows basic usage. First, two `WFSrc` instances are created (variables `~src1` and `~src2`, 1) and their `xy` attribute is set in the normalized range (2, 3). Variable `~pl` is assigned a `WFSrcPlotter` that is given an array with the sources (4). Two other variable store the ip address and the port, and are passed as parameters (together with the same array of sources) to an instance of `WFSrcDispatcher` (`~disp`, 6). After all has been set up, every time the attribute `xy` is set (7), the plotter updates the GUI and the dispatcher sends a message.

```
1 ~src1 = WFSrc(0) ; ~src2 = WFSrc(1) ;
2 ~src1.xy_(0.10, 0.20) ;
3 ~src2.xy_(0.50,0.10) ;
4 ~pl = WFSrcPlotter([~src1, ~src2]) ;
5 ~ip = "192.168.0.255" ; ~port = 4243 ;
6 ~disp = WFSrcDispatcher([~src1, ~src2], ~ip, ~port) ;
7 ~src1.xy_(0.3, -0.5) ;
```

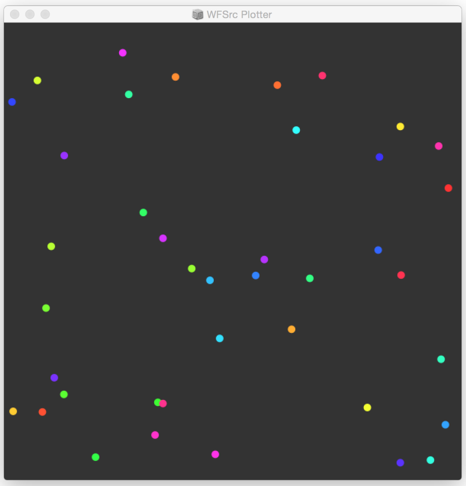


Figure 5 The `WFSrcPlotter` window for 40 sources.

Figure 5 shows a GUI for 40 sources. It can be created by means of the following code:

```
1 ~arr = Array.fill(40, {|i|
2     WFSrc(i)
3     .xy_(rrand(-1.0, 1.0), rrand(-1.0, 1.0))}) ;
4 ~pl = WFSrcPlotter(~arr) ;
```

In the previous example, there is still not audio streamed to the `WFSrc` instances. Audio streams can be routed into the latter by means of the `bus` method, as shown in the following code.

```
 1 SynthDef(\sine , {|out, amp = 1, freq = 440|
 2     Out.ar(out,
 3         SinOsc.ar(freq, mul: amp))
 4 }).add ;

 6 ~arr = Array.fill(10, {|i|
 7     WFSrc(i)})
 8 .collect{|i|
 9     i.xy_(rrand(-1.0,1), rrand(-1.0,1))
10 } ;

12 ~synths = Array.fill(10, {|i|
13     Synth(\sine , [\out , ~arr[i].bus])
14 }) ;

16 ~arr.do{|i| i.mute} ;
17 ~arr.do{|i| i.unmute} ;
18 ~arr.do{|i| i.vol_(-6)} ;
```

Code block 1–4 shows the typical SuperCollider way to encode a definition of an audio generator (a `synth`) by means of a so-called `SynthDef`. Every resulting actual generator built from such a definition simply generates a sinusoidal signal with a default 440 Hz frequency. The block 6–10 creates procedurally an array (`~arr`) of 10 `WFSrc` instances and sets for each a position as a random value in the normalized space. The block 12–14 creates, again procedurally, an array (`~synth`) of 10 audio generators that are routed to the relative `WFSrc` instances by setting the `bus` argument. Finally, lines 16–19 show some controls that mute/unmute and set the volume to -6 dB for all the elements of `~arr`.
Such a simple infrastructure allows to work

[3] A third class –`WFSrcLogger`– is not shown in Figure 4, as it is not relevant for real-time usage. It follows the same mechanism, i.e. registers as a dependant of `WFSrc`. Once instantianted and registered, a `WFSrcLogger` object reacts to each `xy` message from `WFSrc` by writing to a file a time-stamped array of coordinates. Logged data can then be used to inspects positions over time, e.g. for plotting usage, as it happens in the figures of the Applications sections, that have been created automatically by parsing logged data with the Python-based Nodebox package, `http://nodebox.net`.

with SuperCollider interfaced to the IOSONO system from a musical point of view.

5 Applications

Composition work can take into account two different levels (Figure 6). A first level concerns synthesis techniques for the generation of audio signals. A second level is directly related to the control of sources in the space, that is, by setting the xy argument in WFSrc. While synthesis techniques are indeed autonomous from their position in the space, they can be coupled to the latter by taking into account as synthesis parameter the same space coordinates related to the sources (as shown in Figure 6 by the arrows from xy).

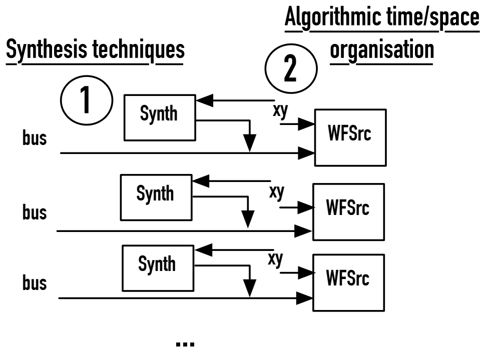


Figure 6 Composing with WFS.

5.1 Brownian motion

The first application was aimed, on one side, to test spatialization, on the other one to exploit a feature introduced before, that is, "parameter linking" [11] between spatialization and spectral elements in synthesis. The aesthetic assumption is that in this way the control space acts both as a controller for synthesis and spatialization, in this way providing a unique and integrated composition environment. The same approach was previously implemented in the GeoGraphy system [12 and 13]. Moreover, such an approach can be thought as a sort of reversible "parameter-based sonification" [14], where spatial data are represented by sound features, and vice versa.

As a first attempt concerning spatialization, Brownian motion has been considered (for applications to musical composition see [11] and [15]). In this case, each source's position changes at a specified rate by a certain interval.

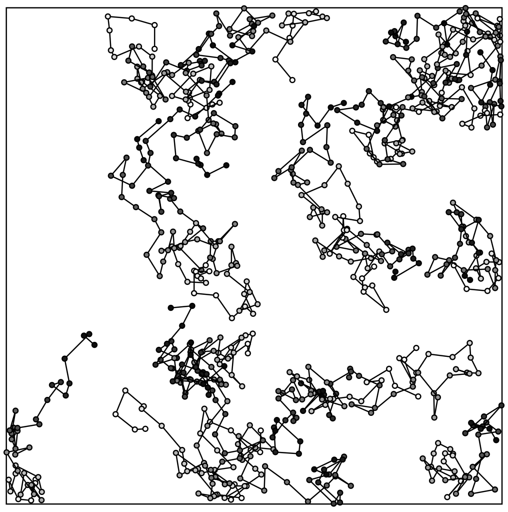


Figure 7 Brownian motion, 20 sources, 39 states from random position.

Figure 7 plots 20 sources during 39 states with a random variation in the range $[-0.1, 0.1]$ for each coordinate. Sequences of points represents paths in the control space of a certain source, its position being connected by a line while point grey value represents time (from white to black). Interactive control allows to vary the update interval to the next state and the variation range. Also, position of all source can be assigned on-the-fly, thus moving all the sources to a certain point or inside a desired region. Figure 8 shows a radiation pattern with the same update date and variation range for 73 states where all sources are initially placed in the origin of the control space. In order to couple spatialization with spectral behaviour, a specific mapping has been defined. The synthesis technique is simply based on sinusoidal generation. A basic SynthDef is shown below, its structure is automatically plotted (via the GraphViz package, [16]) in Figure 20:

```
// simplified version
// only one sinusoid without dephasing
SynthDef(\pulsexy , { arg out, amp = 0.1, freq = 50,
    x =1, y = 1 ;
    var sin =
        SinOsc.ar(
            freq: (freq *
                y.linlin(-1.0,1.0, 1.0, 12).round),
            phase:2pi.rand);
    var sig = EnvGen.kr(Env.perc,
        Impulse.kr(x.linlin(-1,1, 0.5, 4)))*sin;
    Out.ar(out, sig*amp) ;
}).add ;
```

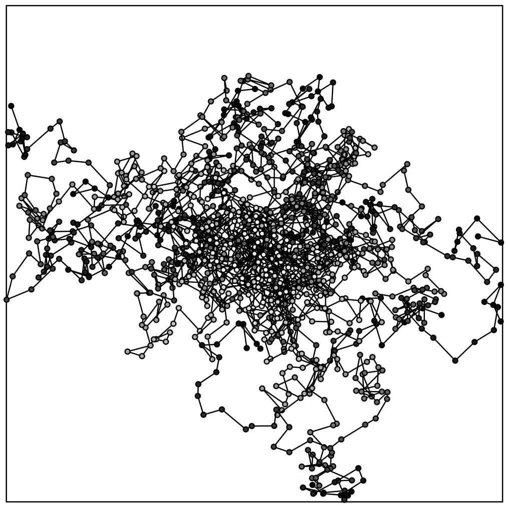


Figure 8 Brownian motion, 20 sources, 73 states from origin.

A sinusoidal generator receives a base frequency as input (default is 50) and it is then enveloped by a unipolar signal with a percussive shape. Control space is taken into account as the frequency is multiplied by the y parameter that maps the normalized space ($[-1.0, 1.0]$) into a multiplier range ($[-1.0, 12.0]$), rounded to integer. In this way frequency is always an integer of the base frequency: vertical position of the source is thus related to a harmonic component of the base frequency. The x parameter (that is, abscissa in the space) is mapped to the envelope retriggering in the range $[0.5, 4]$, to be interpreted as Herz (retriggering per second), thus resulting in a variable temporal density. Input, clipping and mapping of x and y data is shown on top of Figure 20. The technique can be seen as a form of granular synthesis. The actual SynthDef used in the implemented examples is slightly more complicated, as it adds for each sinusoid 20 components with the same frequency but with randomized phases, in order to give a richer microtemporality, as shown below:

```
// freq is a base freq, all synth should have
// the same base freq in order to explore space
SynthDef(\pulsexy , { arg out, amp = 0.1, freq = 50,
    x =1, y = 1 ;
    var sin = Mix.fill(20, {|i|
        SinOsc.ar(
            freq: (freq *
                y.linlin(-1.0,1.0, 1.0, 12).round),
            phase:2pi.rand,
            mul: (-3*(i+1)).dbamp
    )}) ;
    var sig = EnvGen.kr(Env.perc,
        Impulse.kr(x.linlin(-1,1, 0.5, 4)))*sin;
    Out.ar(out, sig*amp) ;
}).add ;
```

When various sources are moving in the space, a relative number of harmonics of the base frequency is excited as a result of the y placement (low/high in relation to back/front), with a density depending for each on their left/right placement (respectively lower/higher density in relation to x).

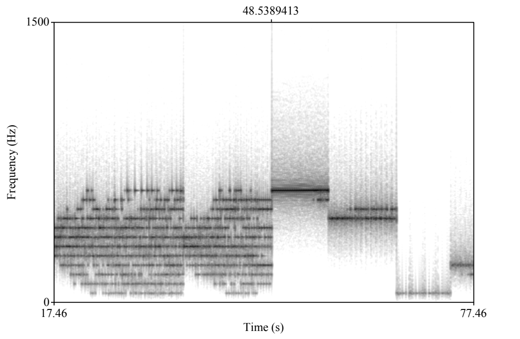


Figure 9 Brownian motion mapped into harmonic explorations: sonogram of the resulting signal.

Figure 9 shows a sonogram of 6 brownian ex-

plorations, all starting from a certain (variable) point in the space. The first two explorations show a greater amount of variation in the brownian parameter: this results in many harmonics to be excited. The last fours (see the temporal marker in Figure 9) show a much slower explosion (almost a spectral freezing) depending on the small amount of variation introduced in the brownian motion (this means that sources are slowly diffusing from a certain point in the space).
Control of spatialization and audio synthesis over time is obtained via SuperCollider's native scheduling constructs (e.g. "routines" and "tasks"). The following code shows how Brownian motion in the control space is implemented and linked both to IOSONO control and sound synthesis.

```
 1 ~num = 20 ; // num of sources
 2 ~arr = Array.fill(~num, {|i|
 3     WFSrc(i).xy_(rrand(-1.0,1), rrand(-1.0,1))
 4 }) ;
 5 ~synths = Array.fill(~num, {|i|
 6     Synth(\pulsexy , [
 7         \out , ~arr[i].bus,
 8         \x , ~arr[i].xy[0], \y , ~arr[i].xy[1]
 9     ])
10 }) ;

12 ~vr = 0.1; ~time = 0.1 ; // variation and rate
13 ~brownian = {
14     inf.do{
15         var x, y;
16         ~arr.do{|i,id|
17             x = (i.xy[0]+rrand(~vr.neg, ~vr)) ;
18             y = (i.xy[1]+rrand(~vr.neg, ~vr)) ;
19             case { x > 1 }{
20                 x = 1-x.frac;
21             }
22             { x < -1}{
23                 x = x.frac.neg;
24             } ;
25             case { y > 1 }{
26                 y = 1-y.frac;
27             }
28             { y < -1}{
29                 y = y.frac.neg;
30             } ;
31             i.xy_(x, y);
32             ~synths[id].set(\x , x, \y , y)
33         };
34         ~time.wait
35     }
36 }.fork(AppClock)
```

It can be seen as a general way to define composition (i.e. a high-level process) within the low-level infrastructure. The variable ~num stores the number of desired sources (1). An array ~arr is assigned a number ~num of WFSrc instances, each with a random position (2-4). The array ~synths is assigned a number ~num of Synth instances, that is, real-time audio generators, intended to be associated each with a WFSrc (5-10). As a consequence, the position parameters for each WFSrc are used to set the related arguments in the associated synth (8). The block 12-36 is the scheduling process. For each WFSrc in ~arr a new position is calculated by choosing a random interval in the [-~vr, ~vr] as a variation of the two coordinates (17-18). Lines 18-30 implements boundary reflection (billiard-like) in case new coordinates are outside the control space. Then, position of WFSrc is set (31) and the arguments of the relative synth are set (32). The process is repeated for an infinite number of times (14) after an interval set by the ~time variable (34). The process can be stopped/reset interactively via code, and all the so-called environment variables (the ones precede by ~) can be set on-the-fly.
Extensions of the technique has included the change of the basic pitch according to certain chordal structures. In this way, the exploration of the virtual space becomes the exploration of the overall harmonic space of a certain chord.

5.2 Trajectories and radial patterns

The harmonic space mapping has been intended as a general test bed for a unified space semantics. The following two applications follow the same schema by changing synthesis technique and space exploration. Brownian motion is a non-deterministic strategy, even if it can be parameterized in a controllable way (e.g. by resetting positions to a specific location). A second mapping address the concept of trajectory.
Figure 10 shows 6 trajectories generated in the control space having the same start and end points, but exploiting SuperCollider integrated envelope geometric constructors that allow to

specify a curvature parameter for a segment. Linear trajectory crosses indeed the centre of the space, as also the sinusoid curvature does. The other four trajectories are created by manipulating the curvature parameter.

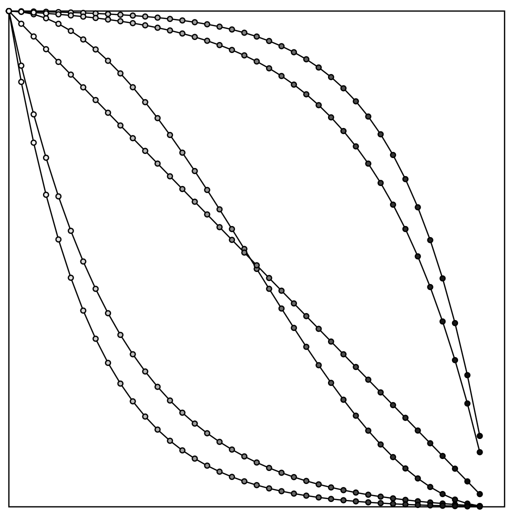


Figure 10 Curvature options for the same trajectory.

In the examples that have been generate the basic synthesis technique makes use of a set of 17 samples, chosen from a set of percussive ones. The basic SynthDef for a sample player is shown below:

```
SynthDef(\play2 , {|out, buf, rate = 1, amp = 1|
    var del = Rand(0.1, 0.7) ;
    var sig = DelayC.ar(
        PlayBuf.ar(1, buf,
            rate: BufRateScale.kr(buf)*rate,
            loop:1),
        del, del) ;
    Out.ar(out,
        sig*amp)
}).add ;
```

The SynthDef uses the basic generator `PlayBuf` (4) that reads a sample previously loaded in RAM (into a `buffer` in the SuperCollider parlance). Apart from the buffer itself, the only parameter passed to the actual synthesizer are `rate` and `amp`, that is, the rate at which the sample is played back (with 2 meaning double speed, 0.5 half speed, and so on) and amplitude (scaled with inverse proportion to distance). The 17 sources are assigned a certain sample that is continuously retriggered when it reaches its end. The SynthDef also includes a randomized delay so that sample playbacks do not start synchronously (3). As before, the position of the sources affects synthesis, simply by varying the `rate` argument. In this case, what is taken into account is not the position but the distance of each source from the origin. The greatest the distance (the more peripheral a source is in the space) the lower the playback rate. Also, amplitude is scaled accordingly to distance. This implements a very crude but perceptually not ineffective Doppler effect (physical simulation was not the aim of the mapping). By this mapping, a substantially clear perception of spatial distribution reverberates in the spectrum domain (and not only, as rate indeed affects sound duration).

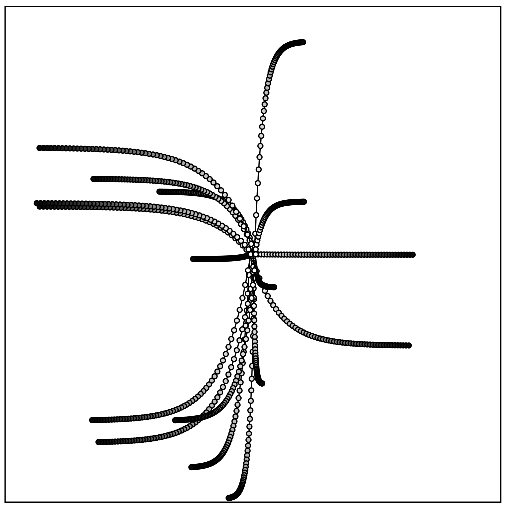


Figure 11 Trajectories in the space for 17 samples, from origin.

Figure 11 shows a diffusion pattern where all sources are placed initially in the middle of the space (thus, as distance is 0, the samples are played at their maximum rate), then reaching random positions by means of various curvatures. As before, point grey value represents time (from white to black).

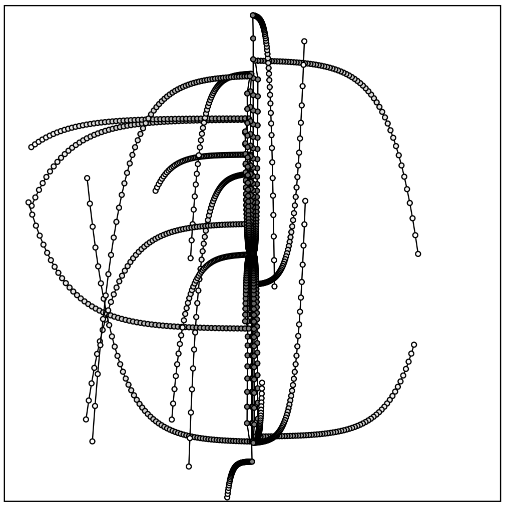


Figure 12 Trajectories in the space for 17 samples, double path.

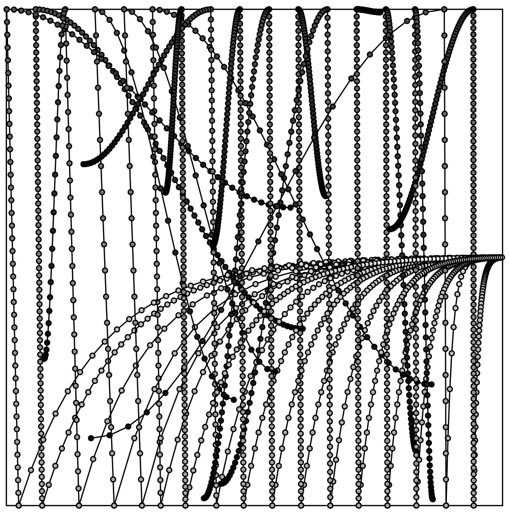


Figure 13 Trajectories in the space for 17 samples, complex movement.

Figure 12 shows a diffusion pattern where all sources are placed initially in the ending positions shown in Figure 11. They are first realigned randomly along the central vertical axis, and then collapsed again int the origin. Figure 13 shows a more complex movement of sources. They all start from position $[1, 0]$, then redistributed along the back line of the space with a specific curvature. Sources then reach linearly the top frontal line, and finally are again redistributed randomly.
The implementation of a "trajectory follower" process is shown in the following block of code:

```
1  ~trajectory = {arg wfsrc, x1, y1, x2, y2, dur, synth,
2      sr = 0.1, curve = 'lin' ;
3      // curve: lin, sine, squared, cubed, floats
4      var xLenght = x2-x1 ;
5      var numPoints = (dur/sr).asInteger ;
6      var xStep = xLenght/numPoints ;
7      var env = Env([y1, y2], [1], curve)
8      .asSignal(numPoints).asArray ;
9      {
10         var x, dist ;
11         env.do{|i,j|
12             x = xStep*j+x1 ;
13             wfsrc.xy_(x, i) ;
14             dist = sqrt(x.squared+i.squared) ;
15             synth.set(
16                 \rate , dist.linlin(0,1.4, 1.5,0.5),
17                 \amp , dist.linlin(0,1.4, 0,-6).dbamp
18             ) ;
19             sr.wait ;
20         };
21     }.fork(AppClock)
22 } ;
```

In this case, a function is passed a `WFSrc` and a `synth`. Start and ending points are represented by `x1, y1, x, y2`, while the argument `curve` allows to specify the type of curve. The argument `dur` is the overall duration. The obtained trajectory is sampled over the duration `dur` at the sampling rate `sr`. Line 7 shows the usage of the `Env` class in SuperCollider, that allows to generate generic segments and includes the curvature specification. The envelope is sampled according to duration and sampling rate, then the temporal process can be started (9–21). For each sample point, the position of `WFSrc` is set, distance from origin is calculated (14) and passed to the synth (15–18), to be used (after scaling) to control rate and amplitude. The sampling rate is the update rate of the process (19).
Geometric construction of position has prompted to explore radial patterns. Interestingly, in this case polar coordinates are indeed more apt to represent positions. The following SuperCollider code example show how to encode radial patterns and their higher-level organization into diffusion processes by means of two functions.

```
 1 ~radial = {|radius, offX = 0, offY = 0|
 2     var point, dist, x, y ;
 3     ~arr.do{|wfsrc, j|
 4         point = Polar(radius, 2pi/~arr.size*j).asPoint;
 5         x = point.x; y = point.y ;
 6         wfsrc.xy_(x+offX, y+offY) ;
 7         dist = sqrt(x.squared+y.squared) ;
 8         ~synths[j].set(
 9             \rate , dist.linlin(0,1.4, 1.5,0.25),
10             \amp , dist.linlin(0,1.4, 0,-6).dbamp
11         ) ;
12     }
13 } ;

15 ~diffusion = {|radius, x, y, dur, rate = 0.1|
16     var howMany = (dur/rate).asInteger ;
17     var step = radius/howMany ;
18     {
19         howMany.do{|i|
20             ~radial.(step*i, x, y) ;
21             rate.wait
22         } ;
23     }.fork(AppClock)
24 } ;
```

The ~radial function is to be passed a radius and the center coordinates (1). For each element of ~arr (containing WFSrc) the position is calculated in polar coordinates: the azimuth is assigned by dividing 2π by the dimension of the array and selecting the angle multiplier in relation to the index of the source (4). Polar coordinates are converted into cartesian ones (5) and assigned to the source (6). The distance from origin is calculated (7) and used to set the source-related synth, after being properly scaled (8-11). The ~diffusion function is simply built upon ~radial. A certain duration of the process is passed, and a diffusion rate (15). The number of iterations of ~radial is then calculated and an internal scheduling process –iterating over ~radial– is set.
Figure 14 shows various radial pattern as basic construction for more complex movements. Radius and centre can be set by passing the opportune parameters to the ~radial function. Figure 15 shows an application of ~diffusion that creates diffusion patterns by progressively expanding the radius for a given centre. Five diffusion patterns are progressively expanding their radius and moving from top left towards the centre of the space. A figurative sound model for the pattern is indeed explosion, that radially expands from a centre, like e.g. in fireworks.

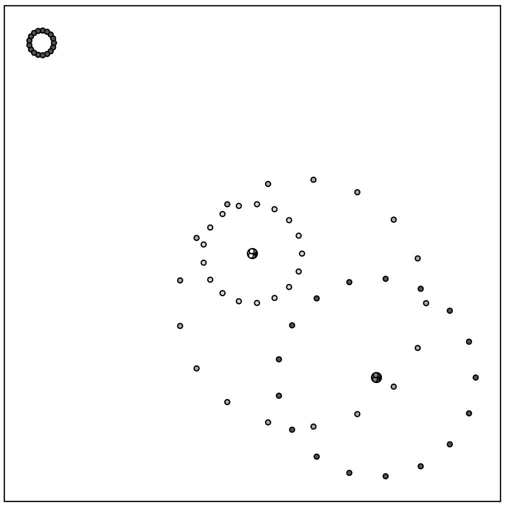


Figure 14 Radial distributions for 17 sources.

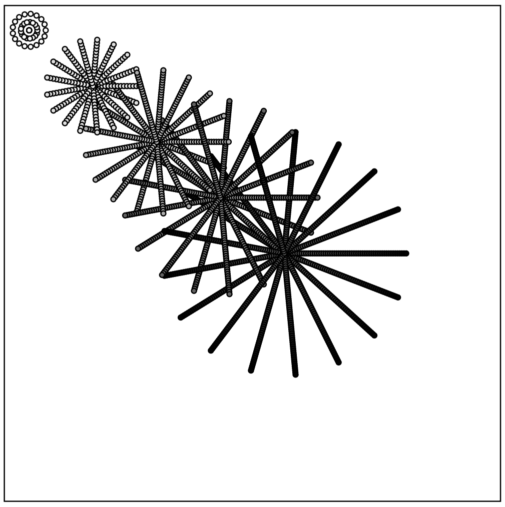


Figure 15 Diffusion patterns for 17 sources, varying radius and centre.

5.3 A Game of Life

The third application has been inspired by two wrong assumptions concerning the IOSONO system while drafting the code:

1. available channels are 192, that is, the number of channels provided by the RME MADIFace XT sound card. Instead, available channels are a maximum of 64 in the Konzerthaus;
2. sources, once placed in the virtual space, keeps on delivering audio. Instead, a source is properly a source event with a default duration of 1 second (Figure 2).

These assumptions have prompted the idea of conceiving spatialization not in terms of moving sources (as in the previous applications), rather as a set of fixed diffusion points in the space where at certain conditions an audio stream may be routed. Hence the idea of considering the control space as a grid of sources to which Conway's notorious *Game of Life* can act as a regulating mechanism[4]. In the Game of Life –a specification of cellular automata– a possibly infinite grid space is filled by live cells. Three rules have been designed in order to determine if in the next state of the system the cells are still live or they die, and if new cells can becomes live. Rules are applied by taking into account the number of live/dead in the 8 neighbour cells of a given cell. Life's rule set has been carefully to deals with both over- and underpopulation, and has sprouted a vast literature that has researched specific cell configurations with an interesting behaviour, e.g. leading to an immutable state during the generation process ("still life") or to the periodic reappearance of the same starting configurations ("oscillators"). The chosen grid for Wave field application has a dimension 13×13, so that each grid element (for a total of 169) can be mapped to a source in the space (based on the wrong assumption of 192 available channels). Such a dimension is still limited for the development of Life's cycles, thus –again with a classic solution in literature– the space has been folded on both its sides, resulting in a lattice. This means that positions on the boundaries that would have their neighbours outside the space always have neighbours on the opposite side.

As the rule set is very simple, a complete SuperCollider implementation of Conway's Game of Life on a generic $m \times n$ lattice is shown below[5]:

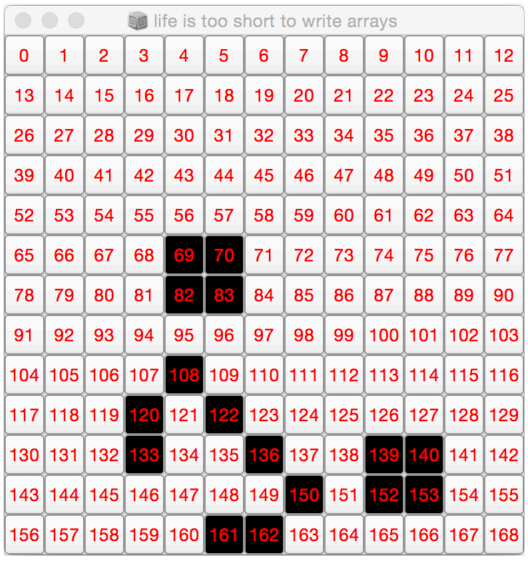
```
// A simulation for Conway's Game of Life
// over a generic m x n lattice

// Get the neighbours of a cell
~getNeighbours = {|a, x,y|
    var row, col, nw, n, ne, w, e, se, s, sw ;
    row = a.size ;
    col = a[0].size ;
    nw = a[(y-1)%row][(x-1)%col] ;
    n = a[(y-1)%row][(x)%col] ;
    ne = a[(y-1)%row][(x+1)%col] ;
    w = a[(y)%row][(x-1)%col] ;
    e = a[(y)%row][(x+1)%col] ;
    se = a[(y+1)%row][(x+1)%col] ;
    s = a[(y+1)%row][(x)%col] ;
    sw = a[(y+1)%row][(x-1)%col] ;
    [nw, n, ne, w, e, se, s, sw]
} ;

// live or dead? Return next state for a cell
~evaluateCell = {|a, x, y|
    var cell = a[y][x] ;
    var state = ~getNeighbours.(a,x,y).sum ;
    var next = if (cell == 1) { // live
        case { state < 2 }{0}
        { [2,3].includes(state) }{1}
        { state > 3 }{0}
    }{ // dead
        if( state == 3){1}{0}
    } ;
    next ;
} ;

// Compute next state, input acutal, returns a state
~nextState = {|current|
    var next = Array.fill2D(current.size,
        current[0].size, {|n,i| nil}) ;
    current.do{|row, i|
        row.do{|col, j|
            next[i][j] = ~evaluateCell.(current, j, i)
        }
    } ;
    next
} ;
```

While many applications of Life to music have been proposed, in this case the basic musical assumption is to directly correlate the grid space to positions in the IOSONO virtual space. Cell configurations are thus activations

[4] The most up-to-date resource is `http://www.conwaylife.com/`. See [17] for an introduction.

[5] As a side note, it is interesting to observe that the implementation, by means of three functions, is almost purely functional, as it uses a substantially stateless mechanism.

of sources in the discretized sound space.
Due to the specific set of rules implemented in Life, random distribution of live cells does not lead to significant processes, as all the cells typically become rapidly dead. As already said, many specific configurations, on the other side, have been discovered that trigger complex behaviours. Hand coding of such configurations is particularly time-consuming and thus prone to error. In order to facilitate the exploration of specific configurations, a GUI tool –LifeDesigner– has been developed that allows to interactively encode configurations on the 13×13 space, save them to disk and load them back when needed. Figure 16 shows the LifeDesigner GUI with a configuration known as the “Unix oscillator”.


life is too short to write arrays

0	1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38
39	40	41	42	43	44	45	46	47	48	49	50	51
52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100	101	102	103
104	105	106	107	108	109	110	111	112	113	114	115	116
117	118	119	120	121	122	123	124	125	126	127	128	129
130	131	132	133	134	135	136	137	138	139	140	141	142
143	144	145	146	147	148	149	150	151	152	153	154	155
156	157	158	159	160	161	162	163	164	165	166	167	168

Figure 16 LifeDesigner GUI interface.

Figure 17 shows a rendering of the first 65 states of a very long process, that is initialized by placing some know patterns, a “toad” (bottom left), a “beacon” (the two four-element blocks) and a “pi-heptomino” (the arc-like structure). Toads and beacons are oscillators, but are disturbed by the interaction with the pi-heptomino. Complex interactions depend indeed also on the lattice structure.

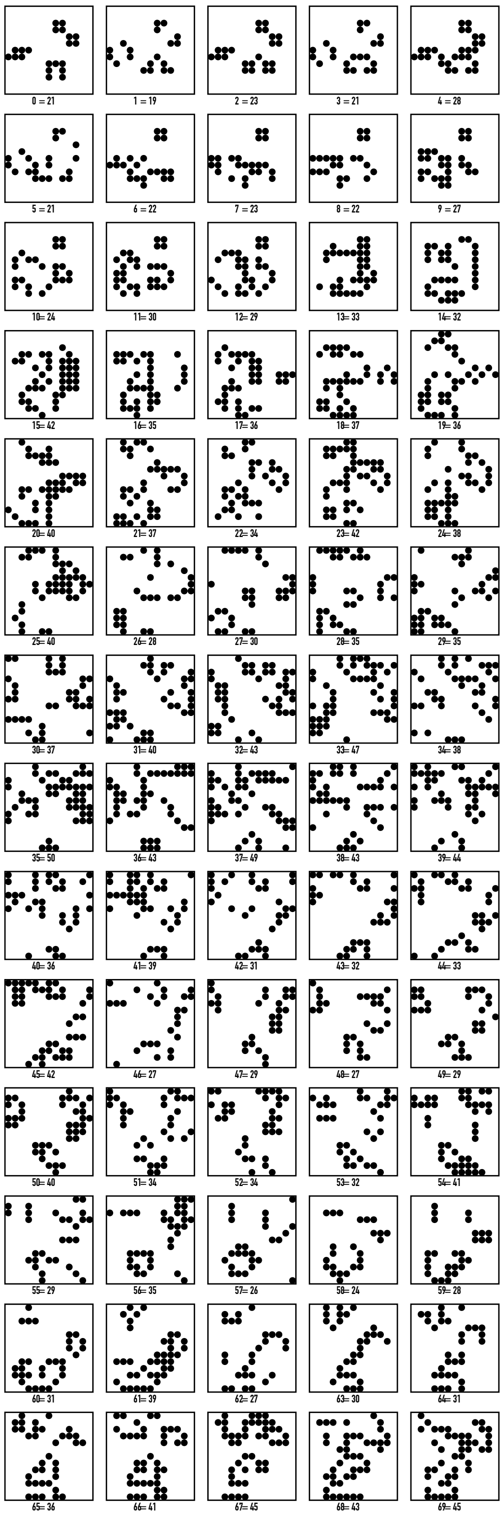


Figure 17 A very long process.

Under each state, the state index (starting from 0 as in Life's convention) and the number of live cell are listed. It can be seen that live cells are typically under the 64 threshold. Figure 18 shows 25 states of the Unix oscillator.

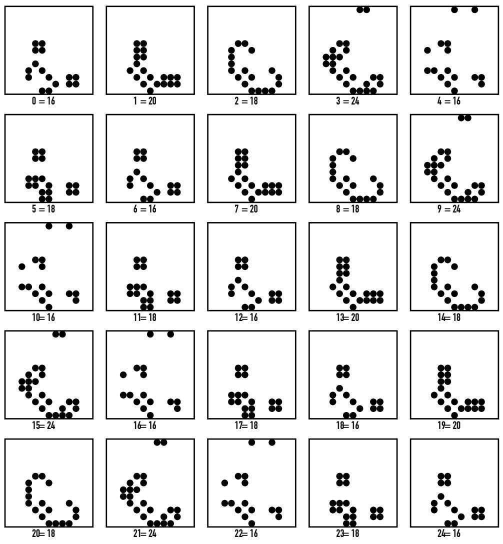


Figure 18 Iterations of the Unix Oscillator.

Periodic states can be easily observed by looking at the diagonal of the states, as the period is 6 generations. The lattice folding becomes evident by looking at states 3 and 4, where bottom cells are folded on top. State plotting is realised in Nodebox by directly exporting the data obtained from the functional SuperCollider implementation shown above (this means that states are those passed to `WFSrc` instances during audio generation).

The two starting wrong assumptions have proven significant in prompting a solution to their implementation that traced a different path from previous examples. Previously, sources were moved in the space by setting the `xy` method, while each `WFSrc` instance retained a specific audio stream routed into.

Moreover, the number of `WFSrc` instances was directly correlated to the number of addressed audio channels, with each `WFSrc` instance being "locked" to a certain channel through its relative synth. In the implemented Life application, empirical observation showed that in general the number of active cells on the chosen grid is inferior to 64, the number of available channels in IOSONO. Thus, an effective dynamic mechanism has been implemented. For each active cell in the current state the next available `WFSrc` is selected, and its position is set according to the position of the cell in the grid. Then, the audio stream relative to the same cell in the grid is routed into the `WFSrc` instance. In this sense, differently from previous applications, `WFSrc` instances are uncoupled from audio streams. As a safety mechanism, the selected `WFSrc` is obtained by iterating on the cell index modulo 64. The worst case scenario is thus that there is not an available `WFSrc` for a cell, and thus the first one is overwritten. But this has never occurred empirically, as 64 channels typically provide room for allocating all the active cells in the selected grid.

Audio mapping has been conceived following two different designs.

In the first case, a long sample is loaded, that is, a 20-minute live improvisation[6]. The file is read by moving the pointer in 169 (13×13, the number of cells on the chosen grid) equi-spaced locations, so that each cell is assigned a file chunk. Once selected, the chunk is enveloped and given a variable duration, that can be set to provide room for the emergence of recognizable sound material or shortened to obtain strong granulation effects. The long-sample strategy has been designed with the idea of providing a fast way to obtain a variety of sounds without loading many different samples (169). Moreover, the percussive and rich spectral material of the selected sample was functional to the rhythmic organization of the resulting generation process. In fact, a typical issue emerges from the use of Life as a music model. There is literally nothing between consequent states (that is, a "generation" or "tick"), and a state is updated instanta-

[6] Institute for the Very Very Nervous (A. Valle, electromechanical setup, D. Sanfilippo, laptop feedback system, A. Secchia, percussions, G. Pagano, prepared guitar), "A hopeful monster", `https://soundcloud.com/ivvn/live_in_milan`.

neously. This necessarily results in a clear basic homogenous time organization that depend on the duration value asssigned to the tick. Longer durations yield a clear rhythmic structure (the "meso-temporal" level to speak with Roads [18]), while very short durations result in a clear granular organization ("micro-temporal" level). Longer durations provides a clearer localization of sources in the space, while grain effects produce sources emerging and suddenly disappearing in the space. It should be noted that the duration range for events is independent from the one referring to generation. This means that is possible both to couple and uncouple slow/fast generation and shorter/longer sounds. So the grain effect depends on the relation between the two parameters, empirically when two conditions are satisfied:

1. sound durations are grain-like (< 100 ms);
2. sound duration $<$ tick duration.

Sound generation shows a variable set of results, from clearly moving percussive/rhythmic patterns to swarm effects.
A second sound mapping has been tested (even if not shown during the workshop) with the idea of exploiting the resolutely discrete nature of the process and at the same to challenge it for obtaining a continuous audio result. Audio code is shown below:

```
 1 SynthDef(\grainy , {arg out, buf, freq=440,
 2     gain = 1,
 3     dur = 0.1, amp = 1 ;
 4     Out.ar(out,
 5         MoogFF.ar(WhiteNoise.ar, freq, gain)
 6         *
 7         EnvGen.kr(Env.perc,
 8             timeScale:dur,
 9             doneAction:2)*amp)
10 }).add ;

12 // usage in Life routine
13 Synth(\grainy , [\out , ~arr[available].bus,
14     \dur , ~tick*~scale,
15     \freq , // according to i: cell index
16     i.linlin(0, ~state.flat.size, 36, 90).midicps,
17     \gain , // according to rows, i.e. back/front
18     (i % ~state.size).linlin(0, ~state.size, 1, 6),
19     \amp , 0.15
20 ]) ;
```

The SynthDef (1-10) simply describes a generator that outputs a grain with a percussive envelope by filtering a white noise by means of an emulation of a Moog VCF filter with adjustable frequency (`freq`) and gain (`gain`). As a side note, it might be noted that the resulting generator deallocates itself after the envelope has finished (for the duration `dur`). This results from passing the `doneAction` argument a 2 value. In this sense, the synthesizer behaves more like a sound event than as an instrument. Lines 13-20 show an excerpt from usage inside a scheduling routine, where `~state` is the 2D matrix encoding a Life state. Frequency is set accordingly to the overall index of the cell ($[0, 168]$), that is scaling in the MIDI domain ($[36, 90]$), and then converted into Hz (16). Gain is instead scaled according to row index ($[0 - 12]$), in this way providing lower gain in the back of the space and higher gain on its front.

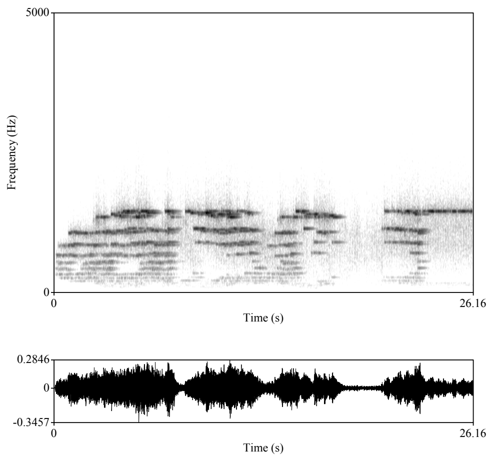


Figure 19 Sonogram and waveform of the Life continuous process.

Figure 19 shows the sonogram and the related waveform from sound synthesis triggerd by the process of Figure 17, where tick has been set to 0.05 seconds. Fusion effects are apparent.

6 Conclusion

Working with the IOSONO system is indeed a rewarding experience as it grants a very complex and fine-tuned experience of sound spatialization. The OSC interface allows to escape the limit of DAW editing and to experiment with more complex behaviours, that, moreover, can be set and controlled in real-time. The interfacing of IOSONO with SuperCollider has shown how complex spatialization patterns can be implemented from a high-level (that is, composition-based) perspective. In this sense, the IOSONO-SuperCollider environment results in a tightly integrated system that brings together state-of-the-art algorithmic composition, audio synthesis and sound spatialization.

7 Acknowledgements

This work has been possible thanks to the Guestprofessorship "Detmold Residence for Sound, Image & Space Design" at Hochschule für Musik Detmold, Winter semester 2016-2017. Prof. Malte Kob has provided substantial help with the Wave field systems. Thanks is due also to Malte Heins for technical support and Stefano Catena for assisting in code testing.

8 References

1 Berkhout, A. J. (1988). A holographic approach to acoustic control. *J. Audio Eng. Soc, 36*(12), 977–995.

2 Malham, D. and Myatt, A. (1995). 3-d sound spatialization using ambisonic techniques. *Computer Music Journal, 19*, 58-70.

3 Pulkki, V. (1997). Virtual sound source positioning using vector base amplitude panning. *JAES, 45*(6), 456–466.

4 Murch, W. (2001). *In the blink of an eye*. 2nd edition Los Angeles: Silman-James Press.

5 Wright, M., Freed, A. and Momeni, A. (2003). Opensound control: State of the art 2003. In Thibault, F., editor, *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)* .

6 Barco Audio Technologies (2015). Iosono scene data protocol v2.3.1 version 22.09.2015. Technical Report Barco Audio Technologies.

7 Sandred, Ö. (2016). Experience of working with the IOSONO Wave Field Synthesis system at the Konzerthaus in Detmold, Germany. Technical Report Hochschule für Musik Detmold.

8 Wilson, S., Cottle, D. and Collins, N., editors (2011). *The SuperCollider Book*. Cambridge, Mass.: The MIT Press.

9 Valle, A. (2016). *Introduction to SuperCollider*. Berlin: Logos.

10 Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc..

11 Roads, C. (1996). *The Computer Music Tutorial*. Cambridge, MA, USA: MIT Press.

12 Valle, A. and Lombardo, V. (2003). A two-level method to control granular synthesis. In Bernardini, N., Giomi, F. and Giosmin, N., editors, *XIV CIM 2003. Computer Music: Past and Future. Firenze 8-10 May 2003. Proceedings*, pages 136-140. Firenze.

13 Valle, A. (2008b). Geography: a real-time, graph-based composition environment. In *NIME 208: Proceedings*, pages 253–256.

14 Hermann, T. (2002). *Sonification for Exploratory Data Analysis*. PhD thesis, Bielefeld: Universität Bielefeld.

15 Miranda, E. R. (2001). *Composing music with Computers*. Burlington, MA: Focal Press.

16 Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper., 30*(11), 1203–1233.

17 Poundstone, W. (1984). *The Recursive Universe: Cosmic Complexity and the Limits of Scientific Knowledge*. New York: William Morrow.

18 Roads, C. (2001). *Microsound*. Cambridge, Mass. and London: The MIT Press.

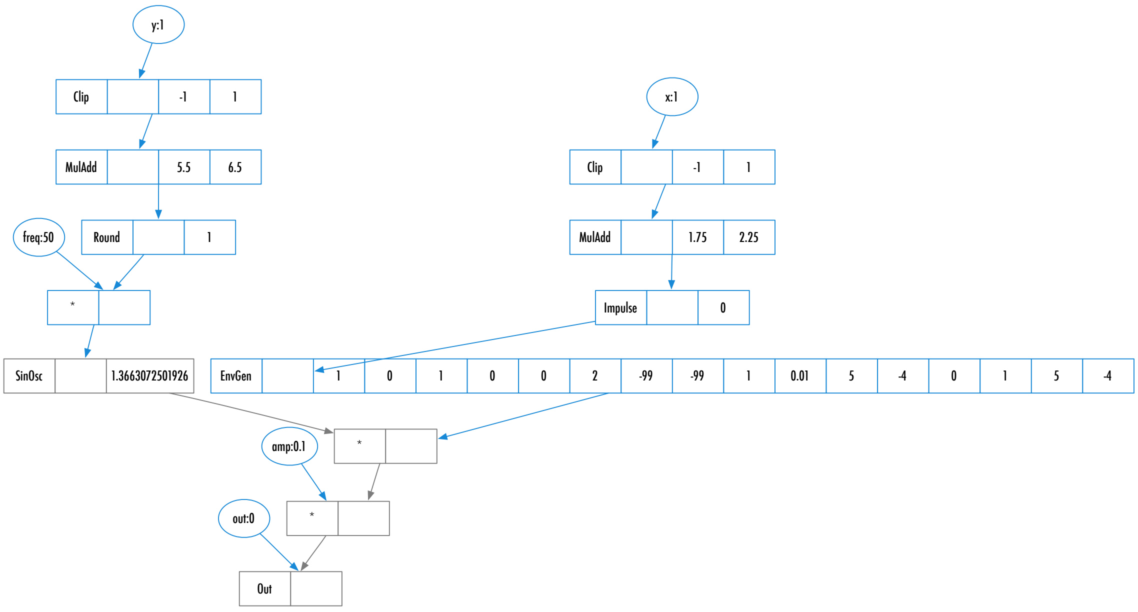


Figure 20 Automated plotting of the SynthDef `\pulsexy`.