5 years of using SuperCollider in real-time interactive performances and installations - retrospective analysis of *Schwelle*, *Chronotopia* and *Semblance*.

Marije A.J. BAALMAN
Design and Computation Arts
Concordia University
Montréal, Québec
Canada,
marije@nescivi.nl

Abstract

Collaborative, interactive performances and installations are a challenging coding environment. *SuperCollider* is an especially flexible audio programming language suitable to use in this context; and in this paper I will reflect on 5 years of working with this language in three professional projects, involving dance and interactive environments. I will discuss the needs and context of each project, common problems encountered and the solutions as I have implemented them for each project, as well as the resulting tools that have been published online.

Keywords

SuperCollider, interactive performance, sensing, composition systems, live coding, tool/code development

1 Introduction

Between 2005 and 2010 I have been involved in three real-time interactive projects, in which *SuperCollider* (SC3) was the central core to deal with realtime sensor data, audio analysis, interaction with other programs for data exchange and show control, and sound, vibration and light output. This paper gives an overview of the techniques used within *SC3* and provides a critical analysis of the problems encountered along the way and solutions provided. The work on these three projects has resulted in a number of tools that have been made available to the general public as open source software, and that aid other artists in the realisation of their projects.

The three projects are all collaborations with artist/researcher Christopher Salter[1] and various other artists. Two of the projects are dance performances, one of them is a one-person experiential installation. Furthermore all three projects have an interactive or responsive component based on sensor inputs and dynamic mapping of these inputs to output media. They are also all situated in a collaborative context, where there are several artists collaborating using different output media, as well as different programming environments with which data is to be exchanged. As the development of these three projects has been sequential and taking place over the course of 5 years, certain approaches and methods using *SC3* have emerged, as well as a re-evaluation of methods used in earlier projects. In all of these projects, I have encountered similar problems, however, as my proficiency at *SC3* developed, I have discovered different solutions. In some cases because I found that the new project asked for a different approach, based on the specifics of the problems, or because I wasn't content with the solution used previously.

While working on artistic projects there is always a trade-off between developing "general-purpose" tools that are robust and flexible in use, and quickly putting something together, that is usable and reliable for the project at hand, but may not translate well to other projects. This paper reviews the methods I have used over the years, and identifies the components that could be, or have been, adapted to more general purpose tools for use in future projects.

For those readers unfamiliar with *SuperCollider*, I have added a section detailing some general information regarding *SC3* at the end of this paper. Class names within *SC3* will be put in bold in the following text.

2 Coding in the context of interactive performance

Coding in a professional performance context has different demands than product oriented coding, in the sense that while writing the code, the purpose of the code and its needed functionality is not yet known, but will emerge during the artistic process of discussions, experimentation and rehearsals. This is especially true,

[1] `http://www.chrissalter.com`

when the artistic project involves real-time sensing, where it is not known beforehand what the input data will be, and how it will influence the output media, which are also being shaped in the process of creation.

Within the rehearsal process for all of these projects it is important to have a flexible system which allows for on-the-fly manipulation of audio synthesis processes as well as sensor data mappings. Part of the preparation for the rehearsal process is to create systems that allow for such flexibility, so that many different kinds of interactions can be explored. This is only possible if it is clear in advance what kind of possibilities there are, i.e. what kind of data is to be expected from the sensors, the type of audio processes that will be used (its compositional structure, as well as its sonic quality), and what kind of interactions the collaborators in the project are interested in. Extensive discussions about this with the other collaborators, as well as short exploratory sessions with the performers, and a basic understanding of some of the movement material of the dancers (so that you can e.g. wear an accelerometer and produce some data yourself while writing and testing code) are essential components in this process. Having some skill at livecoding to quickly develop new interactive processes is also vital for a succesfull rehearsal process.

For the eventual showtime in the theater or at an exhibition, it is important to have a robust "show control" system[2] from which the show can be run, while at the same time being flexible to adapt to differences in setup (e.g. audio balance/mix), based on the venue in which the performance takes place. Ideally, you should be able to adapt "cues" during the show, should there be the need. Backup solutions, in case sensing infrastructure breaks down, can also be useful (even just as a reassurance).

In the case of installations, the code (and the machine it runs on) may need to be prepared to be started and stopped by gallery personnel who have no knowledge at all about coding, and in some cases even of computer environments. In the ideal case the machine running the code can boot up and start the code automatically, so the computer only needs to be turned on.

[2]In theater/performance the collective control for all events supporting the performer's action on stage (i.e. sonic, light, mechatronics, video) happening on stage is usually referred to as "show control".

3 The artistic projects

3.1 Schwelle

Schwelle is a theatrical performance that takes place between a solo dancer/actor (Michael Schumacher) and a "sensate room". The exerted force of the performer's movement and changing ambient data such as light and sound are captured by wireless sensors located on both the body of a performer as well as within the theater space. The continuously generated data from both the performer and environment is then used to influence an *adaptive audio scenography*, a dynamically evolving sound design that creates the dramatic impression of a living, breathing room for a spectator. We aimed at creating an auditory environment whose sonic behaviour is determined continuously over different time scales, depending on the current input, past input and the internal state of the system generated by performer and environment in partnership with one another.

The data from the sensors is statistically analyzed so the system reacts to changes in the environment, rather than absolute values. The statistical data is then scaled dynamically, before being fed into a dynamical system inspired from J.F. Herbart's theory on the strength of ideas [Herbart, 1969]. The dynamic scaling ensures that when there is little change in the sensor data, the system is more sensitive to it. The output of the Herbart systems is mapped to the density, as well as to the amplitude of various sounds that comprise a "room" compositional structure of more than 16 different layers. An overview of the data flow is given in figure 1.

The mapping, which is indicated between the dynamic scaling and the Herbart Groups, is a matrix which determines the degree to which each sensor influences which sound [Baalman et al., 2007]. Additionally, there is a "state" system, defining different parameter spaces within which the soundscape can move. The system moves between these states, depending on long term development of the input data. The theatrical light also has distinct behaviours based on the state of the room. The information about the current state is transferred to the computer controlling the lights via OpenSoundControl (OSC)[3] [Wright et al., 2003].

The diagram also shows that there is a second data flow path, which constitutes a more classical, instrumental approach of using the

[3]`http://www.opensoundcontrol.org`

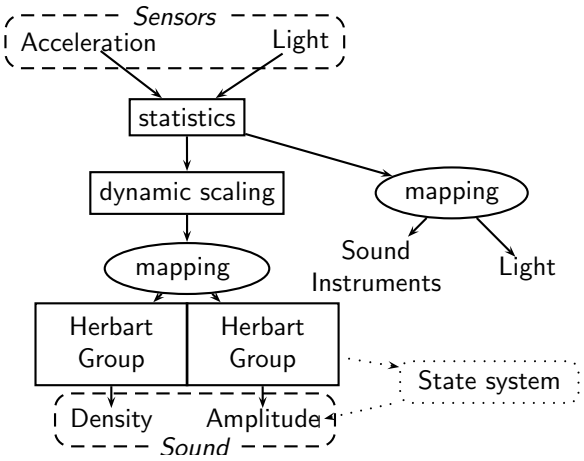

Figure 1: Data flow diagram

sensor data. The mapping between a movement created within the room by the performer or objects in the room and a resulting lighting or sonic event is direct, and recognizable as an action-reaction interaction. This interaction has been carefully tuned to certain dramatic sequences within the piece, and is easily switched on and off, depending on which scene is taking place. In some scenes I change the strength of the interaction at suitable points in the dance, thus creating a duet with the dancer.

Apart from the system mentioned above, there was also a backup system in case the wireless transmission of data from the dancer would break down, so in case of need I could mimic the data using the joysticks of a gamepad device.

The sound was spatialised across the performance space using several methods of amplitude panning between speakers. Additionally, there was an elaborate system for submixing the different audio layers, before sending them to the outputs. In this submix system there were controls, both for direct setting of volumes, and for dynamic volume control mapped to the dynamical system output data.

3.2 Chronotopia

Chronotopia is a dance performance by the Bangalore (India) based Attakkalari Centre for Movement, in collaboration with visual artist Chris Ziegler; the music score is composed and performed by Matthias Duplessy. For this performance we created a responsive light installation: a 6 by 6 matrix of cold cathode fluorescent lights (CCFL), and 3 handheld lights. We controlled these lights with wireless technology, using a combination of an Arduino board and

Figure 2: A snapshot of the view on both the stage and my computer screen visualising the light matrix.

an XBee wireless chip.

For the control of the lights, I used **Synth**s on the server sending their output to control rate busses, which I then polled at a regular interval in order to send it over a serial protocol to the XBee network from within the langauge. This approach allowed me to make use of the various envelope curves that are available within **SynthDef**s, and use the extensive **Pattern** library for sequencing of these **Synth**s. In the prototyping phase for this project, I extensively used *scgraph*[4], which allowed me to model the light matrix and see its behaviour on screen. During the setup and performances in the theater, it allowed me to monitor the behaviours of the system and compare it to the actual output on the stage (see figure 2).

Additionally, I used camera based motion tracking data (from a camera looking down at the stage), to map to the light matrix, as well as beat and pitch tracking data extracted in realtime from the soundtrack. We also exchanged data between the light control and the interactive video, both for synchronisation of cues with the soundtrack (using frametime of the playback), and for connecting the intensity of the lights to the video image (maximum output value of all the lights was used to control the brightness of the video image in specific scenes).

3.3 Semblance

JND/Semblance is an interactive installation that explores the phenomenon of cross modal perception — the ways in which one sense impression affects our perception of another sense. The installation comprises a modular, portable environment, which is outfitted with devices

[4] `http://scgraph.sourceforge.net`

that produce subtle levels of tactile, auditory and visual feedback for the visitors, including a floor of vibrotactile actuators that participants lie on, peripheral levels of light and audio sources, which generate frequencies on the thresholds of seeing, hearing and (tactile) feeling.

In this installation we use wireless sensing devices to gather data from floor pressure sensors. The loudspeaker setup consists of 12 special speakers designed to enhance home theater sound setups with tactile vibrations. These speakers are laid out in a grid of 2 by 6 underneath a platform, or bed, on which the visitor lies down. With the pressure sensors we can detect micro-movements of the body. The light appearing above the visitor is controlled with DMX (see §5.5) from Max/MSP. OSC communication is used to send the desired cues to Max/MSP.

The synthesis of vibrational output was a difficult process. While sound synthesis methods can be used, it is quite different to find vibrations that work — artistically — on a tactile level. While with sound you can sit behind the computer, and code synthesis processes and tweak them while listening to them, to lie down on a vibrational floor and code at the same time is unpractical. Furthermore, it is a medium with which neither of us had any previous experience to draw upon. It was also hard to create vibrations without an acoustical counterpart so that we had to find vibrations that were also sonically interesting.

The sensor data was analysed statistically in realtime so that changes in pressure, rather than the absolute pressure, was used to map to the synthesis processes. Since there were 24 areas of sensing, in a 6 by 4 grid, and 12 speaker outputs, some combining of sensor data was done to map local movements of the body to local vibrations. In certain parts, we used a sum of all the changes in pressure to map to an overall amplitude of the vibration. Furthermore, in one part we did amplitude tracking on the vibration output, and used that to determine the maximum level of brightness of the light.

For spatialisation we employed various methods of panning (in one or more directions), or outputting to one or more speakers at the same time, with either the signals in phase to all speakers, or with a randomized phase.

The composition is made up of three movements and lasts about 13 minutes. Within each movements there are a couple of different parts, with varying output combinations, and various mappings to the sensor data.

4 Common techniques

4.1 Collecting sensor data

The collection and processing of sensor data is an essential part of working on interactive performances. The first step is interfacing with the hardware that actually does the collection of data. In *Schwelle* we used Create USB interfaces[5], which show up as HID devices to the operating systems. In *Chronotopia* motion tracking data is used, which is received from another program (see §5.4) through OSC. In *JND/Semblance* we use wireless XBee based sensors and the data communication takes place over a serial port[6].

For *Schwelle* I made an abstraction between a class named **SchwelleSensor** and a class interfacing with the actual HID device (two classes, one for Linux, one for OSX). I then had alternate versions of the **SchwelleSensor** class which were using backends like the WiiMote and a "mix" of several other sensors. In later projects this abstraction was generalized, instead using a general purpose data framework (the SenseWorld DataNetwork) into which any kind of device can input data and further use of the data is agnostic of the way in which the data was initially gathered [Baalman et al., 2009].

4.2 Processing sensor data

In the class **SchwelleSensor**, I also stored data calculated from statistics of the data, which was performed in the class **SensorData**. These calculations all took place in *sclang*. In the later projects I moved the statistical processing to *scsynth*, taking advantage of the efficiency of the DSP algorithms implemented in the UGens. The resulting, "derived" or "cooked" data was made available on the DataNetwork — a central hub for all the control data. This approach makes it flexible to switch between using the direct data and a derived version by simply changing the data source.

4.3 Mapping sensor data

Mapping of the sensor data involves not only remapping the value ranges between the input data and output parameters, but also the merging of data streams, extracting features from

[5] `http://www.create.ucsb.edu/~dano/CUI/` and `http://overtone-labs.com`

[6] See `http://sensestage.hexagram.ca`

data streams, and creating dynamical processes which develop compelling behaviours based on realtime sensor inputs.

Schwelle was by far the most complex system of interaction as described above and shown in figure 1. The interactions between the different stages in the dataflow path is organised with the class **SchwelleSensorSystem**, which reads the input data, calculates the dynamical scaling (in **DynamicScaleSystem**) and maps it to input data for the dynamical system, implemented in the class **SchwelleHerbart**. All the processing of the data is taking place inside *sclang*, and in custom classes with a lot of interconnection between these classes.

In *Chronotopia* and *JND/Semblance* the data processing is centered around data processing units that are integrated with the SenseWorld DataNetwork framework. This approach makes the creation and alteration of dataflows much more flexible than the approach taken in *Schwelle*. However, some of the data processing algorithms used in *Schwelle* still have to be ported as units to the general framework.

For *JND/Semblance*, I started developing a framework for creating presets, combining set parameters for use with specific **Synth**s as well as mapping of parameters to specific data streams taken from the DataNetwork. These presets can then be stored to disk and recalled in future sessions, and instantiated and tweaked in realtime, both through a graphical interface and through code.

4.4 Data exchange with other programs

In each of the three projects, one of the collaborators was using the software *Max/MSP* to control theatrical lighting. In order to exchange data we had to set up OSC-communication protocols to exchange data. While in *Schwelle* this was done on an ad-hoc basis, defining a OSC address pattern each time we needed to exchange some data, for the later projects we developed a general purpose framework, namely an OSC-interface to the previously mentioned SenseWorld DataNetwork. The framework provides for robust methods to allow for quick reconnection upon restarting the code or patch. The DataNetwork provides a very quick way of sharing any data that may be needed by more than one collaborator, and it is used for sharing show control data (timing, cues), as well as sensor data and output data.

4.5 Managing synthesis processes

SC3 has two distinct methods for working with **Synth**s. One is instantiating a **Synth** directly on the server and then changing parameters of a synth either manually or automated in a task or routine. The other method is using the **Pattern** infrastructure, which provides many higher-level mechanisms for creating sequences in time.

In *Chronotopia* I mostly employed the **Pattern** infrastructure, to create spatial sequences across the light matrix. Only in a few instances I found it more convenient to instantiate **Synth**s, which would then be mapped to control busses with data from the motion tracking.

For *Schwelle* I built up an infrastructure to deal with common methods to handle **Synth**s and their parameters. The class **SchwelleInstrument** handles common methods for starting and stopping **Synth**s, including fading in and out, and providing a submix for the given instrument for individual volume control; various subclasses then implement different variants of instruments, depending on the use of samples (**Buffer**s), audio input from a microphone, mapping of controls to sensor data, as well "clouds" of **Synth**s dependent on input data. Each instrument has a default GUI to see and manipulate the status of the instrument.

For *JND/Semblance* I developed the preset system mentioned above. In addition to this preset system, I developed a central engine, the **JNDEngine** which manages all **JNDSynth**s and their connections to the DataNetwork. **JNDSynth** provides control over settings and mapping of the synthesis processes to data from the DataNetwork. **JNDEngine** also has a graphical interface, with volume controls for all running synths, and buttons to open GUIs for editing individual **JNDSynth**s.

The approach in **JNDSynth** is more general in the sense that it doesn't require many subclasses for special cases of synths, but the submixing approach of **SchwelleInstrument** is absent and it would not yet be able to handle the clouds used in *Schwelle*.

In future work, I anticipate merging the two approaches into a common class.

4.6 Spatialisation methods

All of the works involved spatialisation of some kind. In *Schwelle* there was a setup with two "rings" of speakers, four speakers around the performance area, and then four (or more)

speakers around the audience. Additionally, there were two speakers mounted at the ceiling, one pointed downwards, and one towards the ceiling, so that the audience would only get reflected sound from that speaker. Sounds were then either routed to specific speakers, or panned dynamically between the two rings of speakers. In the class **SchwelleSurround** I created various standard methods for the sound spatialisation, which could then be applied to parts of the soundscape at specific moments during the performance. Thus I was routing the output from one or more **Synth**s through another **Synth**, which did the spatialisation.

In *Chronotopia* I was working with a matrix of outputs, so I needed to pan between these outputs, without wrapping around to the other side. As *SC3* at the time of creation of *Chronotopia* only had a panner that wraps around (**PanAz**), I doubled the amount of output channels used within the panner, and then only sent the channels I actually needed to the output of the **Synth**. But, I also suggested to the *SC3*-community that there should be another panner **UGen** which can pan between multiple channels, but not wrap around. This led to the development of the **UGen PanX**. In order to direct the signals to the right outputs, I either made synths outputting to a specific channel in the matrix (using a **LightGrid** class to select the channel numbers from the one-dimensional array), or I used **SynthDef**s which embedded the **PanX UGen**.

In *JND/Semblance* I was working again with a matrix of outputs, so **PanX** was used extensively. Rather than routing **Synth** outputs to various spatialisation **Synth**s, I developed a system for dynamic creation of **SynthDef**s, where you can define a signal function, which is stored in a library (**JNDSignalLib**), and then create a **JNDSynthDef** with specific types of spatialised outputs. Furthermore all the **JNDSynthDef**s are stored in a separate **SynthDescLib**, which can be browsed with a graphical interface to test each **SynthDef**.

4.7 Show control

In all projects some kind of show control was needed. All works have distinct scenes or movements in which certain things need to happen at certain times (usually referred to as cues in live theater lingua franca).

In *Schwelle* the cues were quite closely tied to the performer's movements on stage, and in certain cases they would prompt the performer. Since there is a fair amount of improvisation in the piece, there was no absolute time at which these cues needed to be executed, although we had defined relative times between events in certain occasions (for the latter, I created a simple **ShowTimer** which showed me a window of how much time had elapsed since a certain moment). In addition, some cues needed specific code to be executed shortly beforehand to prepare processes (allocation of resources), and some others needed code to clean up afterwards (freeing resources). While I started writing a general purpose class for this, I did not end up using this, being unsure about its robustness in showtime (not having tested it extensively during rehearsals). Rather I ended up with a file with code snippets organised according to the timeline of the show, with numerous comments as to when to execute which code. This also allowed me to make quick changes "on the fly" during the performance.

On the other hand in *Chronotopia*, the timing of the piece is strictly tied to the music score and there is no improvisational aspect in the piece. Here, I ended up creating a **CueList** class, which executes functions at specific given frametimes. The cue list is stored in a code file, where functions can be changed, or alternatively you can use the class instance methods to add functions at specific times. The current time was then updated according to the playback of the sound file with the music score.

For *JND/Semblance*, I used three tasks (**Tdef**s) for the three movements of the piece and used a master task, starting these three tasks at the right time. While this allowed us to try out each movement separately, while preparing the piece, this approach was not yet completely satisfactory in its use, mainly because I had to recalculate the total durations of each movement (for proper execution of the master task) everytime we changed the timing of the piece during the preparations.

During rehearsals of the two theatrical pieces that we often had to go back and forth between specific scenes; this is actually the main challenge for coming up with a general purpose cue system, since skipping back and forth means taking care of the proper allocation and freeing of resources, depending on where we are in the show. Also certain cues may have set events in motion, which run during a number of scenes, so there has to be a check which events should be

turned on at a specific time. Finally, of course, during rehearsals the shape of the piece may change, so a quick editing of cues should be possible too.

4.8 Summary

From the above we can see that the different projects have led to the exploration of multiple ways of working on similar problems. The experiences made with capturing, manipulating and sharing sensor data has resulted in a consolidation into the SenseWorld DataNetwork.

The work done in *JND/Semblance* is moving towards a complete composition system that makes it easy to create signals with different spatialised outputs, and creating presets for different instruments and playing and managing the running synths. This system integrates with the DataNetwork. On the other hand there are still some approaches deployed in *Schwelle*, which would be useful to integrate with the JND system to create a complete system.

And finally, the different approaches for show control could still be consolidated into a generalised system that integrates with the composition system and the DataNetwork.

5 Software tools made public

My artistic work has resulted in several extensions to *SuperCollider* (available as "Quarks"[7]), additions to the standard capabilities of *SC3*, as well as standalone programs (supplied with *SC3* classes to interact with them). In this section I will briefly discuss the various tools that have been released and are publicly available.

5.1 SenseWorld and SenseWorld DataNetwork

The SenseWorld Quark is a collection of classes used for dealing with sensor data at a higher level, and some convenience methods. It contains some language side methods to calculate statistics of incoming data streams, as well as a number of PseudoUGens[8] to do the same.

The SenseWorld DataNetwork is a set of classes dealing with sharing data between collaborators using various programming environments. This framework is discussed at length in [Baalman et al., 2009].

5.2 GeneralHID

Moving back and forth between running code on Linux and OSX developed the need for a general approach for interfacing with HID devices. Working from various parallel, platform specific implementations used in *Schwelle*, and also some previous projects, I developed the abstraction **GeneralHID**, which provides a common interface to both **HIDDeviceService** (the OSX HID implementation in *SC3*), and **LID** (Linux input device). The **GeneralHID** abstraction is part of the standard distribution of *SC3* since May 2007.

5.3 WiiOSC and SC3 WII implementation

The use of the WiiMote in *Schwelle* has resulted in the publicly availabe Linux based program *wiiosc*[9], which captures the data from a WiiMote and sends it to a desired client using the OSC-protocol using *liblo*[10]. It has also resulted in a native implementation in the *SC3* language for direct access to the WiiMote, which has been part of the distribution since June 2007.

5.4 MotionTrackOSC

To use videotracking natively on Linux, I created MotionTrackOSC[11], based on the OpenCV library[12] and *liblo*, a simple adaptation of the motion tracking example of the OpenCV library, expanded with OSC control of parameters, and sending the data to a specific client. Furthermore, this program is integrated with *SC3*, through some classes implementing the OSC-communication. As the output of MotionTrackOSC is "raw", namely there is no consistent numbering of the motion tracking points, I implemented an algorithm to keep track of the positions of previously detected moving points and matching these to the new ones, so that the identifiers are consistent. Additionally, I implemented some algorithms to filter out "short-lived" tracking points, which can occur when the light conditions of the tracked area change — this typically happens a lot in a theatrical context.

[7] `http://quarks.sourceforge.net`

[8] PseudoUGens are implemented as small code blocks in *sclang* consisting of other **UGen**s, which can be used just like any other **UGen**s in a **SynthDef**.

[9] available at `http://www.nescivi.nl` since August 2007.

[10] `http://liblo.sourceforge.net`

[11] available at `http://www.nescivi.nl` since January 2009.

[12] `http://opencv.willowgarage.com/`

5.5 DMX

DMX is "the MIDI of theater lighting control", a serial protocol consisting of 8bit control values for up to 512 light channels within one DMX "universe". Most common theater light devices can be controlled via DMX, such as dimmer packs, stroboscopes and motorized lights with many individual control channels. Although there exists a *dmx4linux*-project[13], that project has a very lowlevel approach and there are hardly any programs that integrate with interactive software. As a result of the projects discussed in this paper, a DMX extension has been made for *SC3*, which can currently communicate with the EntTec DMX USB Pro[14]. This extension will eliminate the dependency on Max/MSP for DMX control and simplify some of the setups in the future.

5.6 PanX

The need to spatialise sound on a grid of outputs (speakers) rather than a circle led to the development of the **PanX UGen**, which allows for panning similar to the **PanAz UGen**, but does not wrap around. This **UGen** was implemented by Josh Parmenter and is available from the *sc3-plugins* project[15].

6 Conclusions

Interactive live performance is a challenging and exciting context for coding, and *SuperCollider* is certainly a suitable choice of language for this purpose. Creating tools for solving problems as they are encountered (or invented) may lead to ad-hoc solutions for one performance, but result in more solid tools in subsequent works as problems reoccur. While most tools were written based on an immediate need, the publication of these tools has helped and hopefully will help many other artists working in similar areas.

This retrospective analysis of my coding strategies in these three projects will hopefully give other artists and researchers some insight in the creative process of working with code in artistic projects, and the specific challenges in this context.

7 Acknowledgements

Thanks to Chris Salter and Harry Smoak for the many years of collaboration; also to Alberto de Campo for a number of pleasant and insightful coding sessions. Thanks to Josh Parmenter for implementing the **PanX UGen**.

[13] `http://llg.cubic.org/dmx4linux/`

[14] `http://www.enttec.com/`

[15] `http://sc3-plugins.sourceforge.net`

SuperCollider in brief

SuperCollider consists of two components: an audio programming language, called *sclang*, and a audio synthesis engine, called *scsynth*; these two components communicate with each other via a set of OSC messages. *sclang* is an object oriented audio programming language with dynamic type casting and garbage collection. As a reference for some *SC3* nomenclature I have been using throughout this paper:

UGen unit generator, or its representation in *sclang*.

SynthDef "blueprint" for a Synth, like an "instrument", consisting of a set of interconnected **UGens**.

Synth a running synthesis node on *scsynth*, created from a **SynthDef**; like a "voice".

Quark "packaged" set of *sclang* classes to extend the default class library of *SC3*.

SuperCollider can be found at `http://supercollider.sourceforge.net`.

References

Marije A.J. Baalman, Daniel Moody-Grigsby, and Christopher L. Salter. 2007. Schwelle: Sensor augmented, adaptive sound design for live theater performance. In *Proceedings of NIME 2007 New Interfaces for Musical Expression, New York, NY, USA*.

Marije A.J. Baalman, Harry C. Smoak, Christopher L. Salter, Joseph Malloch, and Marcelo Wanderley. 2009. Sharing data in collaborative, interactive performances: the SenseWorld DataNetwork. In *Proceedings of NIME 2009 New Interfaces for Musical Expression, Pittsburgh, PA, USA*.

Johann Friedrich Herbart, 1969. *Kleinere Abhandlungen*, chapter "De Attentionis Mensura causisque primariis" (orig. published 1822). E.J. Bonset, Amsterdam.

M. Wright, A. Freed, and A. Momeni. 2003. OpenSoundControl: State of the art 2003. In *2003 International Conference on New Interfaces for Musical Expression, McGill University, Montreal, Canada 22-24 May 2003, Proceedings*, pages 153–160.