A SuperCollider Implementation of Luigi Nono's Post-Prae-Ludium Per Donau

1. Introduction

The idea of processing audio during a live performance predates commercial computers. Starting with Thaddeus Cahill's Telharmonium in 1897, electronics have been used to create music in front of audiences [1]. Beyond live performances of purely electronic instruments, composers began writing music for conventional acoustic instruments accompanied by live signal processing in the 1960s. Unfortunately, these works are seldom performed due to the difficulty of acquiring the signal processing equipment necessary for performing the works.

In one notable composition, the *Post-Prae-Ludium Per Donau* (1987) by Luigi Nono, the signal processing component of the work requires two sound engineers and hardware effects capable of producing quadraphonic panning in two directions at once as well as filter, delay, phasing, and reverb effects. Nono gives no directions for how to create or obtain these audio effects—a challenge for most who wish to perform the piece. This paper will investigate the benefit of programming the audio effects for a work like Nono's *Post-Prae-Ludium Per Donau* to facilitate its performances.

2. Choice of Software

In order to realize Nono's signal processing directions, an audio programming environment had to be selected. Although Cycling 74's Max/MSP would have been a viable option, SuperCollider was prefered for several reasons. First, SuperCollider is free and distributed under a General Public License (GPL). It works on a variety of operating systems and has few system requirements. SuperCollider is a high level language that is effective for live

signal processing because it runs a virtual machine at interpret level, allows dynamic typing, and has implemented real time garbage collection and an object oriented paradigm [2] and [3].

Running at the interpret level means that SuperCollider can allow audio processing to run at a higher priority than less important system tasks (within the limits of processor speed and memory). Dynamic typing means that data objects can change in size throughout a program's execution. This flexibility is crucial for live signal processing. Garbage collection refers to how a computer allocates and de-allocates memory. While a program runs, the computer assigns memory to objects. When no longer needed, they are considered garbage and the memory can be reclaimed; running out of memory would be a serious issue. Unlike languages such as C or Pascal, which require memory to be reclaimed with an explicit command, garbage collection takes place automatically in SuperCollider. Additionally, SuperCollider was chosen because it is easy to learn and has a strong and active community of developers.

3. Analysis of the *Post-Prae-Ludium Per Donau* and Programming of the Piece

The *Post-Prae-Ludium* was written in the final decade of Nono's life during his tenure working at the Experimental studio of the Heinrich-Strobel Foundation. Nono's first piece calling for live electronics was written nearly thirty years earlier. He collaborated with tubist Giancarlo Schiaffini to compose a piece that takes advantage of the full range of the tuba's sound, in terms of dynamics, pitch, and timbre [4]. He expands and redefines the sound of the tuba though the use of live electronics. In this piece, Nono implements a quadraphonic sound system and a variety of audio effects. The composition is textural and divided into five sections.

Instead of switching between hardware effects, a performance of the *Post-Prae-Ludium* using this SuperCollider score (included in Appendix A) requires a sound operator to simply

follow the musical score and tubist and evaluate individual lines of code that automatically start and stop tasks to control the audio effects.

3.1 Section 1

In the first section (0'00–5'20) the tubist is given four types of non-traditional performance techniques—(1) half valve playing, (2) singing though the instrument, (3) notes played with vibrato, and (4) multiphonics. Nono draws colored lines though the staff system and asks the tubist to vary which musical material is performed. Throughout this section, the directions for the sound engineer are to randomly vary the input and output volume to four delays (5, 7, 10, and 15 seconds), one for each speaker of the quadraphonic sound system. For the original performance, one sound operator (Rudolf Strauss) controlled the input to each delay and a second operator (Hans Peter Haller) controlled the outputs. In order to simplify the operator's job controlling this section, a function is used to automatically generate input and output envelopes. An example of a generated input/output curve is displayed beneath Nono's graphic in the score (see Figure 1).

At the end of the section, the feedback into the delays is slowly turned up until 100 percent of the signal flows back into the delay. At this point, the feedback parameter returnes to zero and the signal that remains in the delay's buffer is allowed to blend with the musical material of the second section.

3.2 Section 2

The second section (5'20–7'00) directs the tubist to play high-pitched, impulse-like sounds. The signal processing effect of this segment is phasing and dual direction,

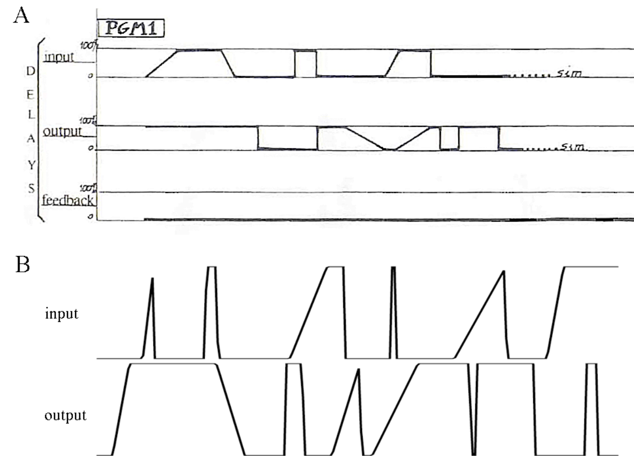

Figure. 1: A, Nono's suggestion for delay input/output levels. B, Sample SuperCollider generated input/output levels.

speed independent, quadraphonic panning. Just like in the first section, this effect would be challenging to perform without either two engineers or the use of a computer.

To manually create this effect using hardware equipment, one would have to send the signal from the tuba through a phaser effect and into two 3D panners, each rotating at a different speed and in a different direction. In the SuperCollider programming of this section, a linear panning algorithm is employed to rotate the sound around the audience. The signal is duplicated so that there can be two simultaneous rotations.

3.3 Sections 3 and 4

In the third section (7'00–7'53), the tubist simply plays an extremely low register note for about 53 seconds. The audio effect through this passage is reverb with an RT60 of 30 seconds sounded through the front two speakers. Overall, the sound quality is low, rumbling, and cavernous. The audio processing of section 3 cross fades into the effect for section four (7'53–11'12), which is a low-pass filtered (566 Hz) version of the tuba part with ten seconds of reverb

played through the rear speakers. The tuba plays a microtonal passages with varied note lengths to excite the reverb.

3.4 Section 5

Section five (10’00–15’10) marks a recapitulation to the sonic world of the previous sections. The tuba continues to play the sonic material of section four until 11’12, but the feedback delay effects of section one start slightly earlier at 10’00. The instructions, “very long, increasingly suspended” suggest that the performer is supposed to blend the tuba’s sound with the delays, which continue until the end of the piece [5].

4. Conclusions and Further Study

The most complicated issue with performing a piece such as Nono's *Post-Prae-Ludium* is to determine how to interpret the composer's idea of how the piece should sound. Nono specifies few details about how the audio effects are supposed to sound. For example, when he asks for phasing and quadraphonic panning in the second section, what effect does he actually want? There is a wide variety of sounds that could be considered phasing and the speeds for the sound rotations is not specified. This problem permeates the entire piece; what does Nono want the reverb to sound like beyond the decay time, how random are the input/output changes to the delays in section one supposed to be, and how is the tubist supposed to interpret some of the nontraditional notation? These issues are not unique to a computer version of the signal processing directions—in the preface to the work the editor writes, “The task of preparing and putting the finishing touches to a composition by Luigi Nono never finished with its first performance. Instead the compositions always ended up as veritable ‘works in progress’

undergoing continuous revision [5].”

Doing the sound manipulation in software rather than hardware greatly increases the practicality of performances of the work. It means that the program can be tailored to each individual performance space, tubist, and sound system. One is not limited to the sounds produced by a single piece of equipment and adjustments to the sound effects can always be implemented. Additionally, the way the sound effects for each section of the piece were programmed allows smooth transitions from section to section. One criticism of this realization of the *Post-Prae-Ludium* is the emotionally detached characteristic of the generated input/output envelopes of the first section. While this is true, it could be argued that without more information from Nono himself, we can never really know what his intentions were for this section.

There are two ways to expand on the work presented here. First, writing a user interface and creating a stand-alone application for performing Nono's *Post-Prae-Ludium Per Donau* would make the program more accessible to non-technical performers. Second, other compositions written for live electronics need to be realized with computer programming. This piece is just one example among a very large catalogue of works—many of which merit modern performances. A computer programming of the audio processing component of Nono's work successfully facilitates the work's performance. It reduces the resources necessary to put the work together as it decreases the number of sound engineers needed from two to one and it reduces the processes of gathering the hardware effects required for the piece.

References

[1] N. Collins, “Live Electronic Music” in *The Cambridge Companion to Electronic Music*, 38–54. New York: Cambridge University Press, 2007.

[2] J. McCartney, “SuperCollider: a New Real Time Synthesis Language” in Proceedings of the International Computer Music Conference. Hong Kong,1996: accessed at: http://www.audiosynth.com/icmc96paper.html

[3] J. McCartney, “Rethinking the Computer Music Language: Super Collider,” Computer Music Journal, Vol. 26, No. 4 pp. 61–68 (Winter 2002).

[4] S. Tigner, “A Performance Guide to Luigi Nono’s *Post-Prae-Ludium no. 1 Per Donau*” Diss. University of North Texas, 2009.

[5] L. Nono, “Post-Prae-Ludium Per Donau: Per Tuba [in F] en Live Electronics” Italy: Ricordi, 1987.

Appendix A: SuperCollider Code

A.1 Nono Sound Operator program

```
Server.local.options.memSize = 1048576;
s = Server.local;
s.boot;
s.latency = 0;

// input stage
~inBus = 0; ("~inBus = "++~inBus.postln);
~numChans = 4; ("~numChans = "++~numChans.postln);
~time = 0.1;

// load data, tasks, and synths (computer lab)
thisProcess.interpreter.executeFile("insertPath/nono/nono_synths.rtf");
thisProcess.interpreter.executeFile("insertPath/nono/nono_data.rtf");
thisProcess.interpreter.executeFile("insertPath/nono/nono_tasks.rtf");


//to perform the Nono, run each of the lines of code below at the approximate time listed

//program the piece
~t1.reset; ~t1part2.reset; ~t2.reset; ~t3.reset;                     /*start from a clean slate*/

/*(00:00)*/ ~t1.play;                                                   /*start t1 (pgm1 &part2)*/

/*(04:30)*/ ~t1part2.stop; ~t1.stop; ~t2.play;                       /*start t2 end t1 (&part2)*/

/*(05:20)*/ ~pg2 = Synth(\pgm2);                                      /*start pgm2 end (pgm1 t2)*/

/*(07:00)*/ ~pg2.set(\gate, 0); ~pg3 = Synth(\pgm3, [\inVol, 0.35]);      /*start pgm3 end pgm2*/
                 //prevention from reverb blowing up
                 ~pg3.set(\inVol, 0.2);
                 ~pg3.set(\invol, 0);

/*(07:53)*/ ~pg3.set(\gate, 0); ~pg4 = Synth(\pgm4, [\inVol, 0.4]);       /*start pgm4 end pgm3*/
                 //prevention from reverb blowing up
                 ~pg4.set(\inVol, 0.3);
                 ~pg4.set(\inVol, 0);

/*(10:00)*/ ~t3.play;                                            /*start t3 (pgm0 pgm1) end pgm4*/
```

A.2 SynthDefs (audio effects)

```
//each SynthDef defines a set of audio effects
(
SynthDef(\pgm0, {
	arg bus = 0, gate = 1, inVol = 1;
	var env;
	env = EnvGen.ar(Env.new([1, 1, 0], [3, 3], 'linear', 1), gate, doneAction:2);
	Out.ar(bus, (SoundIn.ar(~inBus, mul:[1, 1, 1, 1, 1, 1, 1, 1]) * env * inVol));
}).load(s);

SynthDef(\pgm1, {
	arg bus = 0, inVol1 = 1, inVol2 = 1, inVol3 = 1, inVol4 = 0, outVol1 = 1, outVol2 = 1,
outVol3 = 1, outVol4 = 1, feedBack1 = 0, feedBack2 = 0, feedBack3 = 0, feedBack4 = 0, gate = 1;
	var in, out, env;
	env = EnvGen.ar(Env.new([1, 1, 0], [3, 3], 'linear', 1), gate, doneAction:2);
	in = SoundIn.ar(~inBus);
	in = FreeVerb.ar(in, 0.5, 0.5, 0.5);
	out = [CombN.ar((in * inVol1), 5, 5, feedBack1, outVol1),
	      CombN.ar((in * inVol2), 7, 7, feedBack2, outVol2),
	      CombN.ar((in * inVol3), 10, 10, feedBack3, outVol3),
	      CombN.ar((in * inVol4), 15, 15, feedBack4, outVol4)] * env;
	Out.ar(bus, out);
	Out.ar((bus + 4), out);
}).load(s);

SynthDef(\pgm2, {
	arg bus = 0, gate = 1;
	var in, out, env;
	env = EnvGen.ar(Env.new([1, 1, 0], [3, 6], 'linear', 1), gate, doneAction:2);
	in = SoundIn.ar(~inBus);
	in = in + CombN.ar(in, 10, SinOsc.ar(3, add:1.1), 3);
	out = PanAz.ar(4, in, LFSaw.kr(5), 0.5, 2.0, 0.5) + PanAz.ar(4, in, (LFSaw.kr(7) * -1),
0.5, 2.0, 0.5);
	Out.ar(bus, (out * env));
	Out.ar((bus + 4), (out * env));
}).load(s);

SynthDef(\pgm3, {
	arg bus = 0, gate = 1, inVol = 1;
	var in, out, env;
	env = EnvGen.ar(Env.new([1, 1, 0], [3, 20], 'linear', 1), gate, doneAction:2);
	in = SoundIn.ar(~inBus);
	out = GVerb.ar((in * inVol), 20, [30, 30], 0.5);
	Out.ar(bus, (out * env));
	Out.ar((bus + 4), (out * env));
}).load(s);

SynthDef(\pgm4, {
	arg bus = 0, gate = 1, inVol = 1;
	var in, out, env;
	env = EnvGen.ar(Env.new([1, 1, 0], [3, 10], 'linear', 1), gate, doneAction:2);
	in = SoundIn.ar(~inBus);
	in = LPF.ar(in, 566);
	out = GVerb.ar((in * inVol), 20, [10, 10], 0.5);
	Out.ar((bus + 2), (out * env));
	Out.ar((bus + 6), (out * env));
}).load(s);
);
```

A.3 Task Data

```
//~t1 data: arrays are filled with data points correspond to input, output, and feedback volume
(
	//input begining
	j=Array.fill(20, 0);
	a=Array.series(75, 0, 0.1).normalize(0, 1);
	b=Array.fill(60, 1);
	c=Array.series(50, 0, 0.1).normalize(0, 1).reverse;
	d=Array.fill(80, 0);
	e=Array.fill(30, 1);
	f=Array.fill(80, 0);
	g=Array.series(25, 0, 0.1).normalize(0, 1);
	h=Array.fill(50, 1);
	i=Array.fill(20, 0);
	k=j++a++b++c++d++e++f++g++h++i;
	~t1BegIn=k;
);
(
	//output begining
	l=Array.fill(80, 1);
	m=Array.fill(70, 0);
	n=Array.fill(40, 1);
	o=Array.series(75, 0, 0.1).normalize(0, 1).reverse;
	p=Array.fill(25, 0);
	q=Array.series(75, 0, 0.1).normalize(0, 1);
	u=Array.fill(10, 1);
	r=Array.fill(40, 0);
	t=Array.fill(75, 1);
	z=l++m++n++o++p++q++u++r++t;
	~t1BegOut=z;
);
(
	//envelope bits
	~a1 = Array.fill(50, 1) ++ Array.series(50, 0, 0.1).normalize(0, 1).reverse;
	~a2 = Array.fill(50, 0) ++ Array.series(50, 0, 0.1).normalize(0, 1);
	~a3 = Array.fill(50, 0) ++ Array.fill(50, 1);
	~a4 = Array.fill(50, 1) ++ Array.fill(50, 0);
	~a5 = Array.fill(20, 1) ++ Array.series(80, 0, 0.1).normalize(0, 1).reverse;
	~a6 = Array.fill(10, 1) ++ Array.series(90, 0, 0.1).normalize(0, 1).reverse;
	~a7 = Array.fill(70, 1) ++ Array.series(30, 0, 0.1).normalize(0, 1).reverse;
	~a8 = Array.fill(80, 1) ++ Array.series(20, 0, 0.1).normalize(0, 1).reverse;
	~a9 = Array.fill(35, 1) ++ Array.series(65, 0, 0.1).normalize(0, 1).reverse;
	~a10 = Array.fill(20, 0) ++ Array.series(80, 0, 0.1).normalize(0, 1);
	~a11 = Array.fill(10, 0) ++ Array.series(90, 0, 0.1).normalize(0, 1);
	~a12 = Array.fill(70, 0) ++ Array.series(30, 0, 0.1).normalize(0, 1);
	~a13 = Array.fill(80, 0) ++ Array.series(20, 0, 0.1).normalize(0, 1);
	~a14 = Array.fill(35, 0) ++ Array.series(65, 0, 0.1).normalize(0, 1);
	~a15 = Array.fill(10, 0) ++ Array.fill(90, 1);
	~a16 = Array.fill(30, 0) ++ Array.fill(80, 1);
	~a17 = Array.fill(70, 0) ++ Array.fill(30, 1);
	~a18 = Array.fill(90, 0) ++ Array.fill(10, 1);
	~a19 = Array.fill(10, 1) ++ Array.fill(90, 0);
	~a20 = Array.fill(30, 1) ++ Array.fill(80, 0);
	~a21 = Array.fill(70, 1) ++ Array.fill(30, 0);
	~a22 = Array.fill(90, 1) ++ Array.fill(10, 0);
	~a23 = Array.fill(100, 0); ~a24 = Array.fill(100, 1);
	~a25 = Array.series(100, 0, 0.1).normalize(0, 1);
	~a26 = Array.series(100, 0, 0.1).normalize(0, 1).reverse;
);
```

```
//~t2 data
(
	//input
	a=Array.series(110, 0, 0.1).normalize(0, 1);
	b=Array.fill(340, 1);
	c=Array.fill(1050, 0);
	~inT2=a++b++c;
);
(
	//output
	a=Array.series(110, 0, 0.1).normalize(0, 1);
	b=Array.fill(390, 1);
	c=Array.series(1000, 1, 0.1).normalize(0, 1).reverse;
	~outT2=a++b++c;
);
(
	//feedback
	a=Array.series(450, 0, 0.1).normalize(0, 1000);
	b=Array.fill(50, 1000);
	c=Array.series(1000, 0, 0.1).normalize(0, 1000).reverse;
	~feedBackT2=a++b++c;
);

//~t3 data
(
	//input pgm0
	a=Array.series(500, 0, 0.1).normalize(0, 1);
	b=Array.fill(90, 1);
	c=Array.series(850, 0, 0.1).normalize(0, 1).reverse;
	~inT3P0=a++b++c;
);
(
	//input
	a=Array.series(250, 0, 0.1).normalize(0, 0.9);
	b=Array.series(250, 0, 0.1).normalize(0.9, 1);
	c=Array.fill(90, 1);
	d=Array.fill(850, 0);
	~inT3=a++b++c++d;
);
(
	//output
	a=Array.series(260, 0, 0.1).normalize(0, 0.9);
	b=Array.series(280, 0, 0.1).normalize(0.9, 1);
	c=Array.series(700, 1, 0.1).normalize(0, 1).reverse;
	d=Array.fill(200, 0);
	~outT3=a++b++c++d;
);
(
	//feedback
	a=Array.series(200, 0, 0.1).normalize(0, 900);
	b=Array.series(340, 0, 0.1).normalize(900, 1000);
	c=Array.series(700, 0, 0.1).normalize(0, 1000).reverse;
	~feedBackT3=a++b++c++d;
);
```

A.4 Tasks and Functions

```
//~up function creates input and output volume envelopes
(
~up = {
      z = [~a1, ~a2, ~a3, ~a4, ~a5, ~a6, ~a7, ~a8, ~a9, ~a10,
            ~a11, ~a12, ~a13, ~a14, ~a15, ~a16, ~a17, ~a18, ~a19, ~a20,
            ~a21, ~a22, ~a23, ~a24, ~a25, ~a26];
      y = z.choose;
      y;
      };
);

//each task facilitates starting and stopping the audio effects
//~t1
(
~t1 = Task({
      ~pg1 = Synth(\pgm1, [\inVol1, 0, \inVol2, 0, \inVol3, 0, \inVol4, 0]);
      "t1 started".postln;
      ~t1BegIn.size.do({
            arg i;
            ~pg1.set(\inVol1, ~t1BegIn.at(i), \inVol2, ~t1BegIn.at(i),
            \inVol3, ~t1BegIn.at(i), \inVol4, ~t1BegIn.at(i),
            \outVol1, ~t1BegOut.at(i), \outVol2, ~t1BegOut.at(i),
            \outVol3, ~t1BegOut.at(i), \outVol4, ~t1BegOut.at(i));
            ~time.wait;
      });
      ~t1part2.play;
});
);

//~t1part2
(
~t1part2 = Task({
      var ampIn1, ampIn2, ampIn3, ampIn4, ampOut1, ampOut2, ampOut3, ampOut4;
      "t1part2 started".postln;
      inf.do({
            ampIn1 = ~up.value();
            ampIn2 = ~up.value();
            ampIn3 = ~up.value();
            ampIn4 = ~up.value();
            ampOut1 = ~up.value();
            ampOut2 = ~up.value();
            ampOut3 = ~up.value();
            ampOut4 = ~up.value();

            100.do({
                  arg i;
                  ~pg1.set(\inVol1, ampIn1.wrapAt(i), \inVol2, ampIn2.wrapAt(i+20),
                  \inVol3, ampIn3.wrapAt(i+50), \inVol4, ampIn1.wrapAt(i+80),
                  \outVol1, ampOut1.wrapAt(i), \outVol2, ampOut2.wrapAt(i+30),
                  \outVol3, ampOut3.wrapAt(i+50), \outVol4, ampOut4.wrapAt(i+70));
                  ~time.wait;
            });
      });
});
);
```

```
//~t2
(
~t2 = Task({
      "t2 started".postln;
      1500.do({
            arg i;
            ~pg1.set(\inVol1, ~inT2.at(i), \inVol2, ~inT2.at(i),
                     \inVol3, ~inT2.at(i), \inVol4, ~inT2.at(i),
                     \outVol1, ~outT2.at(i), \outVol2, ~outT2.at(i),
                     \outVol3, ~outT2.at(i), \outVol4, ~outT2.at(i),
                     \feedBack1, ~feedBackT2.at(i), \feedBack2, ~feedBackT2.at(i),
                     \feedBack3, ~feedBackT2.at(i), \feedBack4, ~feedBackT2.at(i));
            ~time.wait;
      });
      ~pg1.set(\gate, 0); ~t2.stop;
});
);


//~t3
(
~t3 = Task({
      ~pg4.set(\gate, 0);
      ~pg0 = Synth(\pgm0, [\inVol, 0]);
      ~pg1 = Synth(\pgm1, [\inVol1, 0, \inVol2, 0, \inVol3, 0, \inVol4, 0,
                           \outVol1, 0, \outVol2, 0, \outVol3, 0, \outVol4, 0]);
      "t3 started".postln;
      1440.do({
            arg i;
            ~pg1.set(\inVol1, ~inT3.at(i), \inVol2, ~inT3.at(i),
                     \inVol3, ~inT3.at(i), \inVol4, ~inT3.at(i),
                     \outVol1, ~outT3.at(i), \outVol2, ~outT3.at(i),
                     \outVol3, ~outT3.at(i), \outVol4, ~outT3.at(i),
                     \feedBack1, ~feedBackT3.at(i), \feedBack2, ~feedBackT3.at(i),
                     \feedBack3, ~feedBackT3.at(i), \feedBack4, ~feedBackT3.at(i));
            ~pg0.set(\inVol, ~inT3P0.at(i));
            ~time.wait;
      });
      ~pg1.set(\gate, 0); ~pg0.set(\gate, 0); ~t3.stop;
});
);
```