DYNAMIC LARGE-SCALE GESTURAL CONTROL IN SUPERCOLLIDER SERVER

Joshua Parmenter
Center for Digital Arts and Experimental Media
University of Washington, Seattle

ABSTRACT
Discusses a number of classes in the JoshLib extension library for the SuperCollider real-time synthesis language. ProcMod gives the user the ability to control groups of events with global controls and gestural shaping, as well as real-time performance flexibility. ProcEvents and ProcSink give the composer and performer control over instances and overall structure of ProcMod events.

1.INTRODUCTION
The SuperCollider classes described here have been developed from my composition of a number of pieces for instrument and live-electronics as well my recent research for the composers Richard Karpen and Juan Pampin at the Center for Digital Arts and Experimental Media at the University of Washington is Seattle. In the late 1990s, Karpen began a series of pieces for soloist and real-time processing with the SuperCollider 2 programming language for Mac OS 9. In the first of these, Sotto-Sopra for amplified violin and real-time electronics, a system was created to allow the computer part of the piece to be controlled during a performance through input from a computer operator. One of the great advantages of the structure developed by Karpen was the ability to start processes dynamically and free the resources of the process once they were no longer needed (rather then a more common approach to many live electronics pieces where all processes are started before a piece begins, and are then controlled through live or automated mixing). The structure of Sotto-Sopra required a pre-set ordering of the signal processing functions to be composed then triggered in real-time to coincide with events in the violin part. This required a very precise performance by both violinist and computer operator, while opening up many opportunities in the composition of the electronic part since Karpen able to use the computer’s full power for any part of the piece, rather then having to base his processing decisions on the limitations on the total processing power needed for continuous computation of all signal processing techniques. Karpen’s Solo-Tutti for viola and live-electronics, Juan Pampin’s Oid for piano, live electronics and video and my own Music for Bassoon and Live Electronics and Organon Sustenuto for flute, cello, bassoon, double bass and live electronics challenged and refined this structure even further to allow for more flexible performance situations, the triggering of events in response to the performer’s input in addition to the computer operator, as well as the ability to control processes involving multiple instruments. With the appearance of Mac OS X and the development of SuperCollider Server, I have been able to begin work on a library of functions that encapsulates many aspects of the control structures that have been developed in this style of interactive piece. The library also greatly expands on the earlier possibilities to allow for control of the computer part solely by the performer, visual feedback on the state of a performance, as well as the ability to connect a single SuperCollider language client (sclang) to multiple sound synthesis engines (scsynth) across multiple processors on a single machine or to multiple computers sharing a high speed network. While the language itself provides a number of tools to control small parts of this compositional approach (envelope generators, control buses, synthesis grouping), until now it has been up to the programmer to find ways to synchronize these tools. For example, functions that could control the release of a global envelope with the halting of a loop controlling the creation of a stream of notes had to be created on a piece by piece basis. With ProcMod, there is now a tool available in SuperCollider that allows for easy control and syncing of a number of these parameters, providing a structure through which the control of complex algorithmic process can be treated as a single unit. ProcEvents and ProcSink provide a structure that allows the combination, dynamic creation and control of the gestures created with ProcMod.

2.PROCMOD
ProcMod (short for Process Module) is the most basic object in the system for the management of large scale events (example 1). The creation of an instance of the ProcMod class automatically communicates with the language to set-up a number of basic controls. Each ProcMod will create its own group in the scsynth node tree, as well as register a control bus for sending values from a global amplitude envelope that can be utilized by each event a ProcMod generates. Amplitude can be changed dynamically, and these changes of signal can be smoothed with ProcMod’s lag parameter.

Each instance of ProcMod can have a function to be evaluated when the ProcMod is played, as well as additional functions that can be executed when the ProcMod is released and finished. OSCresponder (for receiving OpenSound Control messages from scsynth or other OSC enabled programs) can also be registered to an instance of ProcMod. The allocation and release of the OSCresponder will be added at play time, and removed when the ProcMod is released. Once a function is started, ProcMod also records its starttime according to the global system time, and makes the time elapsed since the start of a ProcMod available for use to the functions both within
**ProcMod**

**Class Methods**

*new(env, amp, id, group, addAction, target, function, releaseFunc, onReleaseFunc, responder, ...)*

env - an overall amplitude envelope that synths create in a ProcMod function can access. This will run a js env on ProcMod/envs that can be read by other synths in the same process through the creation of a procmodenv synth. There is a max of 20 breakpoints to the env. If the Env has a releaseNode, ProcMod will continue to process events until release is called.

amp - an overall amplitude control for an instance of ProcMod.

id - a symbol or "string" to be used later to identify an instance of ProcMod.

group - a group for an instance of ProcMod to run in. Defaults to nil and creates a new group.

addAction - an addAction for this instance of ProcMod. Defaults to 0.

target - a target for this instance of ProcMod. Defaults to the ProcMod.

function - a Function, Task or Routine to be evaluated on the playing of this instance of ProcMod.

releaseFunc - a Function, Task or Routine to be evaluated after the ProcMod has released its release message.

onReleaseFunc - an instance of Function, Task or Routine to be evaluated at release time.

responder - an instance of OSCResponder or OSCResponderNode for use by this instance of ProcMod. It is automatically added when the ProcMod starts, and released after the ProcMod mishes its release.

timeScale - applies a scale function to the ProcMod envelope. Defaults to 1. lag applies to changes the val since passed into this instance of ProcMod.

clock - an instance of Clock to run this instance of ProcMod. Defaults to SystemClock.

server - an instance of Server to run this ProcMod on. Useful for remote servers. Defaults to Server.default.

**Instance methods**

play - evaluates this instance of ProcMod. ProcMod.function is evaluated, and ProcMod.responder is set up if they are declared.

release - same as play.

release - releases an instance of ProcMod. If ProcMod.env has a release function, functions and OSC responders will wait until this has executed before releasing the ProcMs functionality.

kill - immediately free the ProcMod, regardless of ProcMod.env.

release - return the control bus id the global envelope is written to for this instance of ProcMod.

env - (EnvNode) - an instance of Env to be sent to the synthdef controlling an instance of ProcMod.s overall amplitude and event control. If a number is passed it represents a realtime for the ProcMod.

function - (func) - an instance of Function, Task or Routine to be evaluated on ProcMod.envbus.

releaseFunc(func) - an instance of Function, Task or Routine to be evaluated after a ProcMod has released.

onReleaseFunc(func) - an instance of Function, Task or Routine to be evaluated after a ProcMod has released.

responder - (OSCResponder) - an instance of OSCResponder or OSCResponderNode for use by this instance of ProcMod.

amp - (val) - If there is an envelope controlling the overall amplitude of events, set the amplitude to val.

lag - (val) - If there is an envelope controlling the overall amplitude of events, set the lag time for changes of amplitude to take effect (with the amp, instance method)

data - (Association) - places the Association into a Dictionary for later access.

**Example 1: The ProcMod Class Description**

a ProcMod as well as in the SuperCollider language itself. If the global envelope is of a fixed duration, the ProcMod, when started, will execute that envelope over the specified duration then free all of the resources associated with that instance of ProcMod. If the envelope has a releaseNode, ProcMod will continue to execute its functions until the release message is sent to the object. The object also contains a generic data slot that can hold information that may need to be accessed outside of a the individual ProcMod instance by any other part of the SuperCollider client, (similar to creating and referencing a structure in C).

The global envelope control and function handling are the two strongest features of the ProcMod class. When the ProcMod is created, it checks the env slot for a number of possibilities. If an envelope of indefinite duration is used, the release duration of the envelope is calculated and will be used to free resources after the envelope is released. If a fixed duration envelope is passed in as the argument, the information about its duration is used to halt the creation of new events in the ProcMod function, as well as freeing the resources that are being used in the group that is created by the ProcMod. The control bus where envelope data is written to can be passed into events created by the ProcMod by querying ProcMod's envbus instance variable, or through an argument to the ProcMod's function. If nothing is passed into the envelope argument, then the process will run indefinitely, and on release all processes will be immediately shut-off. If a number is passed in, it is used as a duration in seconds that indicates the amount of time after a release message is received by the ProcMod before all processes will cease. In both of these instances, no envelope or control bus is created.

The function argument accepts a number of different kinds of objects. Functions can be used most effectively for events that simply need to be started when the ProcMod is played. Tasks and Routines (or a Function that returns a Task or Routine) may be used for events where future scheduling is necessary. Infinite loops and other events where it is impossible to foresee the number of events that will need to be created are best handled in this way. The Task or Routine will create its own processing thread, and once started, the process will continue to execute until the instance of ProcMod is released and the thread is freed. The group, control bus and the instance of server that are used by ProcMod are passed in as arguments to the Function when the ProcMod executes.

Since the functionality of ProcMod is all contained within its own thread and synthesis within its own node group, it is easy to create and run multiple ProcMocs at the same time, each with its own gestural shaping. The entire ProcMod process can also be placed into a specific place in scsynth’s order of execution, to allow sound to be routed from one ProcMod into another. Effects routing and processing is easily handled, and multiple processes can be routed into a single effects process. Finally, ProcMod contains a simple GUI interface (example 2) that allows the starting and stopping of the ProcMod’s function, as well as amplitude control.

**Example 2: The ProcMod Graphic User Interface (GUI)**

The server slot will allow a single sclang process to control multiple scsynth server processes. On multiprocessor systems, separate server instances can be run on different processor cores. In addition, the communication between the SuperCollider language and multiple servers can be broadcast over a TCP or UDP network, so it is possible for a single sclang process to control an almost limitless number of synthesis servers. At this point, the limitations are placed on the calculations necessary to run the single client process, rather then the limitations of the DSP engine.

3. PROCEVENTS

The ProcEvents class can be used to organize a predetermined pattern of ProcMod or function instances. Its main functionality comes through the ability to initiate ordered events and releases. The predominant usage of ProcEvents would be through its gui method that displays current event information, gives control to the computer operator of overall amplitude, and allows for all computation to be released or immediately halted. The
GUI also allows the operator to skip to a specific event if needed. The events array that is passed into an instance of ProcEvents is also very flexible. Each event in the event array is itself an array of two optional objects, the first a ProcMod or function to execute when the event index is referenced, the second a ProcMod’s ‘id’ that should be released. Both of these slots may be yet another array of multiple ProcMods to start or release, encouraging the user to build ProcMods as modularly as possible, then creating complexity through varying combinations of simpler processes.

In addition to simply stepping through events, there are slots in the class to store instances of ProcMod that can be used to for special instances of ProcMod that need to be executed with the first event or for when the program is killed. These ‘init’ and ‘kill’ ProcMods are especially useful for starting synthesis processes that are global (e.g. Ambisonic decoding or dynamics processors) or for allocating and freeing memory.

As with ProcMod, multiple instances of ProcEvents may be run, and again, multiple synthesis servers may be used to distribute heavy processor loads. ProcEvents also contains a number of timers from which the processes under the control of an instance of ProcEvents can call upon. The starttme method of ProcEvents will return the time stamp of the initial event. If the ‘id’ of a ProcMod contained in a ProcEvent’s event array is passed in to the starttme function, ProcEvents will return the starttme of the ProcMod instance in relation to the initial ProcEvents starttme. The new method of ProcEvents will return the current time in relation to the starttme of the instance of ProcEvents, while passing this function an ‘id’ will return how long a specific ProcMod has been running. Finally, it is possible to attach an audio trigger to an instance of ProcEvents to tell the class to move on to the next event. This allows the triggering of events to be completely controlled by the performer. In performances where score following is of great importance, this allows the performer, who is following the score as closely as anyone can, to step through the necessary processes. The GUI also has the option to display a large number display in these instances to provide feedback to the performer (example 3).

Including code for an entire piece is not possible in the space provided, but example 4 shows a short sample of code. initproc reads live input, limits it, and sends it out to a virtual bus for use by any other process. pmod is a function that creates an instance of ProcMod that is programmed to create a granular gesture from filtered versions of the limited input. The pmod function allows each event to have its own global envelope, controls over the windowsize and number of overlapping grains, as well as lower and upper bounds for a random number generator that will be polled with each new grain to control the center frequency of each windows filter.

```{ 
var pevents, pmod, initproc, routebus;
//filters, envelopes and pans the input
SynthDef(\filtgrain, [arg procenv, inbus, dur, filtfreq, pan, 
var filt, grainenv;
grainenv = EnvGen.ar(Env([0, 1, 0], [0.5, 0.5], \wavelm),
timeScale: = dur, doneAction: = 2) * In.kr(procenv);
filt = BPF.ar(In.ar(inbus), filtfreq, 0.01, grainenv);
OffsetOut.ar(0, Pan2.ar(filt, pan))).load(s);

SynthDef(\limit, [arg procenv, outbus, limit = 1; 
var src = AudioIn.ar([(1) * In.kr(procenv)]; 
Out.ar(outbus, Limiter.ar(src, limit))).load(s));

//an audio bus to route sound from limit to all others synths 
routebus = s.audioBusAllocator.alloc(1);

//a function to create new ProcMods with varying parameters
pmod = (arg globenv, id, amp = 1, grainsize, 
hifreq = 1760, lowfreq = 480; 
ProcMod(globenv, amp, id) 
.function({(arg group, envbus, server; 
Task({
var waittme; 
waittme = grainsize / overlaps; 
loop({ 
server.sendMsg({\ns_new, \filtgrain, 
server.nextNodeID, 0, group, 
procenv, envbus, \dur, grainsize, 
\filtfreq, hifreq, \rand(lowfreq), 
\pan, 1.rand2, \inbus, routebus); 
waittme.wait; 
}))});

initproc = ProcMod(Env([0, 1, 0], \sin, 1), 
.addAction: 0, target: 0) .function({(arg group, envbus, server; 
server.sendMsg({\ns_new, \limit, server.nextNodeID, 0, 
group, \outbus, routebus); 
"ProcEvents started".postln })

/*Event Numbers*/
pevents = ProcEvents( 
/*0*/ [pmod.value(Env([0, 1, 0], \sin, 1), \ev1, 
20.dbamp, 0.02, 1, 1760, 880), 
nil],
/*1*/ [pmod.value(Env([0, 1, 0], \lin, 1), \ev2, 
6.dbamp, 0.02, 1, 1760, 880), 
nil],
/*2*/ [pmod.value(Env([0, 1, 0], \lin, 1), \ev3, 
6.dbamp, 0.02, 1, 1760, 880), 
nil],
/*3*/ [nil, \ev3],
/*4*/ [nil, initproc] 
).load, initproc, nil, "Sample Piece";
pevents.perfGUI;
})
```

Example 4: A short piece using the ProcMod and ProcEvents structure

The ProcEvents structure has proven to be very flexible and robust. Pieces for soloist with electronics are easily managed with the structure, while pieces for ensemble with electronics work well. Each ProcEvents instance stores all of the ProcMods it is responsible for in the instance variable ‘eventDict’, allowing other processes or instances of ProcEvents to access the event structure. As a result, data between processes can be shared through ProcMod’s ‘data’ structure, and events in one performer’s ProcEvents instance can release events in another. Finally, all of ProcEvent’s events are set-up when the code is initially interpreted, avoiding computational overhead during a performance.

4.PROCSINK AND FUTURE PLANS

ProcSink is a recent effort to expand the performance possibilities of the ProcMod object. Where ProcEvents works well for pre-designed electronic parts, there is little flexibility
Example 5: An algorithmically created instance of ProcSink with GUI

for situations where a non-linear approach to an electronic part is wanted. ProcSink is designed to have ProcMods added to its structure on the fly, and allows for each instance of ProcMod to have its own on/off toggle and amplitude control. Once again, a flexible GUI is available and will automatically update itself as new processes are created (example 5). In addition to ProcSink’s ability to expand its performance possibilities in real-time, the structure also lends itself very well to bringing the creation and altering of an electronic part’s processes into a rehearsal situation. The state of ProcSink can be saved at any time to a file, and that state can then be recreated later without having to reload any code. While the ProcSink class can be ideal for improvisational or rehearsal work, I also see it as a tool that can be developed to eventually produce fixed event structures. One goal I have for ProcSink is the ability to record performance information so that it can be recreated later, perhaps through a function that captures a ProcSink performance and saves it as a ProcEvents type of structure. Both ProcEvents and ProcSink would also be enhanced if some typical features of digital audio workstation automation controls (as well as the visual manipulation of envelopes) could be created. It would also be ideal for there to be a way for ProcSink to generate multiple instances of a single ProcMod that could then be played simultaneously with any number of layers.

```javascript
// the function ‘a’ returns a new ProcMod. Takes the upper // and lower bounds of possible frequencies as an argument a = {arg high, low; var proc; // create a new ProcMod proc = ProcMod.new(Env((0, 1, 0), [1, 1], \sin(1), -12.dbamp); proc.function(({arg group, envbus, server; Task(); var minsize, overlaps; overlaps = 8; loop({ s.sendMsg(\s_new, \singrain, server.nextNodeID, 0, group, \freq, high.rrand(low), \amp, 1, \dur, minsize, \envbus, envbus); (minsize / overlaps).wait));)})); proc; // the new ProcMod is returned from the function. ;}

// create new instances of ProcMod... store them to the // variables ‘b’, ‘c’ and ‘d’ b = a.value(5000, 10000); c = a.value(880, 440); d = a.value(8000, 12000); // play all three {b, c, d}.do({arg proc; proc.play}); b.release; // stop them one at a time c.release; d.release;
```

Example 6: Functional creation of ProcMod instances from a basic prototype function.

This kind of prototyping functionality still needs to be implemented to ProcMod. Currently, SuperCollider’s Function object can be used to algorithmically generate multiple instances of a basic ProcMod, where the instance that is created takes the arguments to the function into account. A simple example of this can be seen in example 6 where the basic ProcMod functionality is the creation of a random granular cloud texture with note frequencies randomly chosen between an upper and lower bounds. By evaluating the Function ‘a’, a new instance of ProcMod is created to reflect the arguments to the function. This functionality should be set up either as a method to the ProcMod object itself, or as an additional class whose sole function is the generation of ProcMod instances with flexible parameter control. If this can be achieved, I see no reason why the GUI for ProcSink shouldn’t be controlled through the prototyping object, with control arguments also appearing in the interface for the user. Finally, it would be ideal if there were methods to all of these classes that would allow these structures to be easily saved as stand-alone applications.

5. CONCLUSION

The separation of the synthesis server and language client in SuperCollider 3 provided the overall environment with greater stability and more flexibility, yet cost the user the tighter integration between large scale signal based controls that existed in SuperCollider 2. ProcMod and its companion classes provides a language based solution to this loss of larger scale control through its management of a combination of smaller tools available to the user in the standard SuperCollider library of classes and objects, allowing the user to create complex gestures, while controlling those gestures as a single entity.

6. REFERENCES