Use of Pure Data, Lisp, Python and Supercollider3 for composing music and realtime work with sound.

Kjetil S. Matheussen

24th January 2005

This document is an english translation of the paper I wrote for the course “Sound Technology Project”, which I attended winter/spring 2004 at the Music Department/University of Oslo.

The project consisted of source codes, sound files, a norwegian version of this paper, and a demonstration/concert.

The source codes, the original version of this paper, and sound files can be found online at this location: http://www.notam02.no/~kjetism/mus2860/
Contents

1 Introduction 3

2 Software used in the project 4
   2.1 Pure Data 4
   2.2 SuperCollider3 4
   2.3 Jack 4
   2.4 Guile 4
   2.5 Python 5
   2.6 OSC 5

3 Software developed in conjunction with the project 5
   3.1 Jack in/out extension module for Pure Data 5
   3.2 Guile extension module for Pure Data 6
   3.3 Supercollider3 module for Python 8
   3.4 Object-oriented system for Guile 10

4 Composition 10

5 Literature 12

6 Web references 13
1 Introduction

The purpose of this project was to, in one way or the other, make a system for algorithmic composition and real-time work with sound. If the composer wants to, it is going to be possible to control the progress of the composition based on what happens at the moment the sound is produced.

This is of course something you already have the possibility to do in various ways. The two most used software-systems for the mentioned purpose is Pure Data (and later MAX/MSP which is very similar) and SuperCollider3. Unfortunately, I think both Pure Data and SuperCollider have their weaknesses. The greatest problem regarding Pure Data is its graphical programming environment, which I think is not very dynamic, and is time-consuming to use for larger programming projects (if even possible). SuperCollider3, on the other side, is a complete programming language, based on SmallTalk. I could have done the whole project with SuperCollider3, but wanted to use the possibilities of Pure Data, besides wanting to use Lisp rather than SmallTalk for the algorithmic composing.

The great strength of Lisp, as I see it, is its ability to make itself fit for any task. For example, while Lisp is the second oldest programming lanugae still in use today (Graham, p.1), its concept has always had full support for object-oriented programming, long time ago before Simula1 was made.

The great strength of Pure Data, as I see it, is that it is a nice tool for binding together various components. There’s three reasons for this: 1. Pure Data has a defined system for signal routing. 2. Pure Data has a graphical interface for routing and programming. 3. Pure Data has an open defined application programming interface (API). The API is the reason for Pure Data’s many extension modules 2, and which makes it possible in this project to let lisp, python and OSC communicate.

What this means is that I from a graphical interface made in Pure Data can control SuperCollider3, lisp, python and sound signals on a very high level.

The operating system I have used is Linux. There are many reasons for this. The most important ones are that it has a very nice multimedia support, is fast, has a large amount of software available and is Unix-like. As the only alternative, I could have used MacosX, which has many of the same qualities as Linux. But MacosX is still quite new, and the prices of the apple hardware is, today, many times more expensive than the cheapest hardware for a PC. Another reason is that since the development of the main components in the project mostly is being done on the on the Linux platform, I most probably would have had to use a lot of time to make things work together. (...)

---

1 the first object-oriented programming language
2 known as “externals”
2 Software used in the project

2.1 Pure Data

Pure Data, or PD, is a graphical programming environment for sound and graphical processing in real time. The program is made by Miller Puckette, the same person that made the first version of MAX. When you make programs in PD, you set up graphical boxes with text inside, and draw lines between them. The lines either define a signal path or a data path.

2.2 SuperCollider3

Supercollider3 consists of two parts; scsynth, which works as a server that makes sound in real time by receiving OSC signals, and sclang, which is a programming language communicating with scsynth. sclang is quite similar, and is based on, the programming language Smalltalk.

Supercollider3 is made by James McCartney. Originally, Supercollider3 only worked with MacosX, but when the source code for Supercollider3 was made available to the public two years ago, much work was made, mostly by Stefan Kersten, to make it run in Linux as well. Today, Supercollider3 is a stable program running under Linux. Stefan Kersten has also made numerous objects for the Linux version of Supercollider3 to support graphical interfaces.

2.3 Jack

Jack is an acronym for “Jack Audio Connection Kit” and consists of a sound server and a shared library. Programs that use Jack’s library to communicate with the sound server can both send and receive sound from other programs also supporting Jack. Both Pure Data and Supercollider3 support jack. Jack is made by the jack community, leaded by Paul Davis and Jack O’Quin. What makes Jack different from other sound servers is that all clients run synchronously. Each client is guarantied to receive and send the same amount of sound data per channel within the same time frame. This is because jack is made for professional real time work with sound, and it is not as natural for an mp3 player or a movie player to support jack as it is for a multitracker or a sound editor.

2.4 Guile

Guile is a scheme interpreter. Scheme is a clean and pretty lisp dialect, and although the specification itself for scheme is much smaller than the one for Common Lisp, guile still provides a lot
of functionality making guile a usable alternative to a Common Lisp implementation. For example, in addition to the latest scheme specification, r5, Guile supports common lisp-like macros, posix functions, hash tables and a lot of additional list related functions. Guile is especially made to be an extension language. For the project, I have made Guile available as an extension language for Pure Data by making a Guile extension module.

2.5 Python

Python is a simple and powerful programming language. In the project I have used Python to communicate with scsynt (the server part of SuperCollider3) inside Pure Data by using Python as an extension language. Python is made by Guido van Rossum and the Python extension module for Pure Data is made by Thomas Grill. The OSC module for Python which I’ve used, is made by Stefan Kersten.

2.6 OSC

OSC stands for “Open Sound Control” and is an open protocol developed at the University of Berkeley to make multimedia software communicate together. OSC is in the project used to let Python and SuperCollider3 talk together.

3 Software developed in conjunction with the project

3.1 Jack in/out extension module for Pure Data

Pure Data already supports Jack to get sound in and out of the program. But it was not previously possible to route jack sound signals within the Pure Data’s graphical programming environment. And for various technical reasons, it should not be possible either, at least not without buffering data, and make latencies in the signal path. But with the help of some dirty tricks, involving accessing undocumented functions and variables, it was possible to make it work. The result is an extension module called “k_jack~“. The trick used was to change Pure Data’s number of in and out channels when a k_jack~ object is created, automatically route the jack graph, and let the k_jack~ object work exactly like a combined dac~ and/or adc~ object, only with a special syntax.

The syntax for k_jack~ is [k_jack~ <jackname>] . So if you for instance make an object called [k_jack~ SuperCollider], this object will have the same number of signal inputs as the number of output channels for SuperCollider, and the same number of signal outputs as the number of input channels for SuperCollider. The argument for k_jack~ takes regexp strings as well, so one can for
example create an object called [k.jack~], and get access to all jack input and output channels currently in the system.

3.2 Guile extension module for Pure Data

To be able to use lisp for algorithmic composition from within Pure Data, it was necessary to make a lisp extension module for it. There exist many lisp implementation, but the scheme interpreter Guile stands out from the rest since its especially made to be used as an extension language. It should therefore not be necessary to use external processes and other tricks to make it work. An additional, and more important, argument for choosing Guile is that the CCRMA software Common Lisp Music, Common Music and Common Music Notation is available for Guile, and one get access to these software environments from within Pure Data by using Guile.

The extension module I have made is called k\texttt{guile}. It is fairly complete, but loses one important functionality; It should have been possible to run Guile in a separate thread. The reason is that Guile is not constructed for hard real time work, it has a garbage collector thats quite often stops the world for many ms. This is often unacceptable for realtime sound work. As a workaround for the problem, in the project, I run to instances of Pure Data; one that use the k\texttt{guile} object and doesn’t play sound, and another one that doesn’t use k\texttt{guile} that threats data and plays sound. These two instances could have been communicating with each other with the help of the netsend and netcreceive objects if there had been a need for it.

The syntax for k\texttt{guile} is \texttt{[k.guile <filename>]. <filename>} is the name of a scheme source code file. When the object is created, k\texttt{guile} first sets up a new scheme \textit{environment} by putting all the code inside a \texttt{define} block. This means that the code in the scheme file won’t run at the top level. To run code at the \textit{top level}, use the \texttt{(load)} command to load another file, since \texttt{(load)}’ed files are evaluated at the top level.

\texttt{k\texttt{guile} provides the following functions:}

- \texttt{(pd-inlets num-inlets)}
  \texttt{pd-inlets} sets the number of inlets of a Pure Data object. This one must be called before any other \texttt{pd-*} commands.

- \texttt{(pd-outlets num-outlets)}
  \texttt{pd-outlets} sets the number of outlets of a Pure Data object. This one must be called before any other \texttt{pd-*} commands.

- \texttt{(pd-inlet inlet-num type func)}
  \texttt{pd-inlet} lets you define what a function that is called when a message from Pure Data is received to one of the inlets of a k\texttt{guile} object. “func” is the function that is called if there comes a message of the type “type” to the inlet number “inlet-num”. “type” myst be a
Pure Data type, and can be for instance “float”, “list” or ’bang. One special type is ’all. If ’all is chosen to be the type, “func” is called no matter what type “type” is, unless a more specific type is specified for the inlet as well.

- (pd-outlet outlet-num arg0 arg1 ...)
  pd-outlet sends arguments to the objects outlet number “outlet-num”.

- (pd-bind symbol func)
  If a message is sent to “symbol” inside Pure Data, “func” will be called with the arguments. A better name for the function would perhaps had been “pd-receive”.

- (pd-unbind symbol)
  Call this function when you don’t any longer want to receive messages.

- (pd-send symbol arg0 arg1 ...)
  Works the same way as the “send” object in Pure Data.

- (pd-get-symbol symbol)
  Both scheme and Pure Data use symbols to bind data. This is a function to receive the Pure Data version of a Scheme symbol. The function “pd-send” understands both symbol types, but runs faster if it gets a Pure Data symbol rather than a scheme symbol.

  There is no difference in speed between running the two following code blocks:

  (pd-send (pd-get-symbol ’gakk) 0)

  and

  (pd-send ’gakk 0).

  ...However,

  (let ((gakk (pd-get-symbol ’gakk)))
    (pd-send gakk 0)
    (pd-send gakk 1))

  ...will run faster than:

  (begin
    (pd-send ’gakk 0)
    (pd-send ’gakk 1)).

  But since the pd symbols are stored in a hash table, I doubt there’s much to gain by using this function.

- (pd-set-destroy-func thunk)
  “thunk” is called before the k_guile object is deleted or receives a “reload” message.
3.3 Supercollider3 module for Python

The server for Supercollider3, “scsynt”, communicate with other programs and computers by using OSC. And by using an OSC implementation for Python made by Steffan Kersten, I have made a Supercollider module for Python which makes it possible to control “scsynth” from Python. And by further using Thomas Grills pyext Python extension module for Pure Data, I can control SuperCollider3 from Pure Data, and then also from lisp by using the k_guile extension module. This does sound unnecessary cumbersome, and it is. It would have been far simpler to make a supercollider module for guile instead. The reason is that I originally planned to use Python for tasks that I later decided to do in lisp instead, and therefore, at that time, it was a lot more natural to let python take care of the supercollider communication as well. Fortunately, it seems to work fine to access supercollider from Pure Data this way, although its probably not the ultimate solution.

The implementation of the syntax is very close to sclang. So a more detailed description of this module can be obtained by reading the documentation of Supercollider3.

The Supercollider3 module for python provides the following classes and methods:

- **Server**

  - def __init__(self,magic,ip=standardip, port=standardport)
    Constructor. Only in very special situation it should be necessary to call this one. You get the standard server by using the variable “localServer” instead.

  - def sendMsg(self,command,*args)
    Sends an OSC message to SuperCollider3.

  - def sendGetMsg(self,command,*args)
    Sends an OSC message to SuperCollider3, and returns an answer. Answering form the SuperCollier3 server is not implemented, so the function works nearly as sendMsg.

  - def boot(self)
    Boots the supercollider server in the same way as when you call server.boot in sclang. (The method currently only sends a /g_freeAll message to the Supercollider3 server.)

  - def dumpOSC(self,code)
    Calls self.sendMsg(“dumpOSC”,code)

  - def nextNodeID(self)
    Returns the next free Node ID.

  - def nextBufferID(self)
    Returns the next free Buffer ID.

  - def loadSynthDef(self,name)
    Calls self.sendMsg(“/d_load”,name)
- def loadSynthDefDir(self, dir)
  Calls self.sendMsg("/d\_loadDir", dir)
- def evalSynth(self, name, body)
  Evaluates a synth. “body” is a sclang code block that is evaluated by sclang.
- def evalSynthFile(self, synthname)
  Calls self.evalSynth with the contents of the file “synthame” as arguments.

- Node

  - def __init__(self)
    Constructor.
  - def set(self,*args)
    Calls server.sendMsg with /n\_set as the command name, the id number for the node as the first argument, and args as the rest arguments.
  - def dontFree(self)
    If this method is called, a message to the supercollider3 synth will be sent telling it that it will be released if the object is deleted.
  - def __del__(self)
    Destructor. If dontFree has not been called previously, a delete message is sent to the Supercollider3 synth.

- Synth
  ...Is a Node subclass.

  - def __init__(self,name,args=[],position=sc\_tail,server=localServer,id=-1)
    Constructor. Creates a Supercollider3 synth.

- BufferSuper

  - def __init__(self,server,numFrames=0,numChannels=1,filename="",startFrame=0,id=-1)

  - def record(self,channel=0,length=-1)

  - def __del__(self)

- Buffer
  ...Is a BufferSuper subclass.
def _init_(self,numFrames,numChannels=1,server=localServer,id=-1)

• BufferRead
   ...Is a BufferSuper subclass.
   
   def __init__(self,filename,startFrame=0,numFrames=0,server=localServer,id=-1)

• BufferRecord
   ...Is a BufferSuper subclass.
   
   def __init__(self,channel=0,length=1,server=localServer)

3.4 Object-oriented system for Guile

Guile already has a very nice system for object-oriented programming called “Goops”. “Goops” is quite equal to Common Lisps object-oriented system called “Clos”. Unfortunately, I don’t like these systems, because you have to write so much: You have to, for instance, specify the class type for arguments for methods. Therefore, I have made a smaller message based object-oriented system instead.

The implementation is placed in the file “oo.scm”, which also contains some examples of use. The same code is also present in the sound editor “SND” in the file called “gui.scm”.

In the project, this system is used everywhere to make the code more easy to read. At least I think so. Other people like more functional programming styles.

4 Composition

The composition is defined in many scheme files; “supercollider.scm” is the code-file that receives messages and time signals from Pure Data; “stuff.scm” contains various general function to handle communication with supercollider (via Python/OSC), handle what to do with messages coming from the fiddle “object that is connected to the k_guile object, and scheduling; “instrument.scm” contains some classes to make sound; And “generate.scm” contains the structure of the music.

The sound material consists of 5 sounds; a stretched sound I can’t remember the origin for, an explosion, a breaking glass, a violin playing a somewhat steady tone, and yet another stretched
sound I can’t remmeber the origin for. Each of these sounds are again split into 5 new sounds that is used as a part of a granulating process that happens in the file “instrument.scm”.

I have to admit that I haven’t been using much time on the composition. Its first and foremost made to show that the system works, and to show some of the posibillities the system offers. I think for example the way to feed lambda-functions to the scheduler is extremely elegant.
5 Literature


6 Web references


*Common Music*, http://ccrma.stanford.edu/software/cm/doc/cm.html

*Common Music Notation*, http://ccrma.stanford.edu/software/cm/doc/cm.html

*Flext*, http://www.parasitaere-kapazitaeten.net/ext/flext/


*k_guile*, http://cvs.sourceforge.net/viewcvs.py/pure-data/externals/k_guile/

*k_jack*, http://cvs.sourceforge.net/viewcvs.py/pure-data/externals/k_jack/

*MAX/MSP*, http://www.cycling74.com

*morph*, http://fixme


*OSC*, http://cnmat.cnmat.berkeley.edu/OpenSoundControl/

*OSC modul for Python*, http://user.cs.tu-berlin.de/kerstens/pub/OSC.py

*OSC for SuperCollider*, http://cnmat.cnmat.berkeley.edu/OpenSoundControl/SuperCollider/

*Pure Data*, http://pure-data.sourceforge.net/

*PyExt*, http://www.parasitaere-kapazitaeten.net/ext/py/

*Python*, http://www.python.org

*SND*, http://ccrma.stanford.edu/software/snd/

*SndOBJ*, http://www.may.ie/academic/music/musictec/SndObj/main.html


*SuperCollider3*, http://supercollider.sourceforge.net/