Embedding Pure Data with libpd

Peter Brinkmann
Google Inc.
peter.brinkmann@gmail.com

Peter Kirn
createdigitalmusic.com
peter@createdigitalmedia.net

Richard Lawler
PatternMusic.com
richard@patternmusic.com

Chris McCormick
McCormick IT
chris@mccormick.cx

Martin Roth
Reality Jockey Ltd.
mhroth@gmail.com

Hans-Christoph Steiner
hans@at.or.at

Abstract

We present libpd, a thin wrapper that turns Pure Data into an embeddable audio library. libpd comes with language bindings for Java, Processing, Objective-C, and Python, as well as support for Android and iOS. We discuss the design and structure of the library, its relation to Pd itself, and several use cases including OpenFrameworks and Webkit.

Keywords

Libraries extending Pd, Pd on modile devices, embedding Pd

1 Introduction

Developers and artists today may conceive integrated audiovisual performances and mobile music applications running on everything from phones to wearable suits, or generative sonic and musical structures for games. With such varied applications, a monolithic sound toolkit often isn't enough: you need a toolkit that can operate within a range of contexts. Pure Data's essential design as interactive, digital plumbing for sound and data, represented graphically as a dataflow "patching" environment, is perfectly suited to the task, but it has traditionally been tied to a user interface and audio APIs that assume standalone operation on a desktop platform. libpd is an exercise in subtractive development: by removing audio, user interface, timing, and threading capabilities from Pd, one is able to separate the concerns of the tool, leaving a more flexible, embeddable library. That library can then run in the context of other applications, such as providing an interactive music engine for a game or performance tool, as well as function more easily on mobile platforms like iOS and Android.

Here, we report on the design of libpd, the way in which it may be used by developers, artists, and end users (and those who are a combination of the three categories), and practical applications already shipping for a variety of platforms that use the library. We also consider possibilities for future development, not only in regards to libpd, but also Pure Data itself.

Developers and users of libpd meet at the web forum Pd Everywhere.[1]

2 Overview

The heart and soul of a DSP library is a rendering callback that takes input samples and computes output samples. Samples go in; magic happens; samples come out. That's signal processing in its purest form, and the main purpose of libpd is to extract this functionality from Pure Data and make it available as an audio processing callback. The second purpose of libpd is to allow for straightforward exchange of control and MIDI messages between Pure Data and client code. libpd supports bang, float, and symbol messages, as well as lists and typed messages. (In other words, pointer messages are the only messages that are not supported.)

Essentially, the libpd API consists of variations of the central processing callback for different sample types (short, float, double), a set of functions for sending messages to Pd, and a set of function pointers for receiving messages from Pd. In order to receive messages, client code must implement suitable receiver functions and assign them to the appropriate function pointers of libpd. Bindings for object-oriented languages preserve the general feel of the API but supply an object-oriented mechanism for passing messages.

[1] http://noisepages.com/groups/pd-everywhere/

3 Workflow

libpd provides great separation of concerns. Sound designers, musicians, and composers don't have to know about programming, and programmers don't have to know about sound design. The sound designer can stay within the confines of Pd's graphical dataflow user interface, without needing to work, for instance, with a game coded in C++. The game designer, likewise, can use their tool of choice and need not understand how to use Pd. All they have to do is agree on the number of input and output channels as well as the collection of send and receive symbols through which the client code will interact with the Pd patch. The sound designer can go ahead and build a patch, controlling it with the usual GUI elements.

Building a patch for libpd is no different from building a patch for Pd itself. In order to prepare the patch for deployment, the sound designer only has to assign the appropriate send and receive symbols to the GUI elements. In the context of libpd, client code will communicate with these send and receive symbols programmatically, sending messages from GUI events and sensors to Pd, or updating its own GUI in response to messages from Pd. Now the application programmer can simply load the patch and use it as a black box.

The collection of GUI elements that the sound designer uses when designing and testing a patch is also the conduit through which the deployed patch communicates with the client code, and the embedded copy of Pd simply becomes a part of the application. Pd itself allows for rapid prototyping of audio components. With libpd, the prototype becomes the production code.

4 Design decisions

The focus of libpd is on signal processing and message passing. The aspects of Pure Data that concern editing, GUI interactions, and interactions with a desktop system were deliberately removed in the process of making libpd. In particular, libpd has no audio drivers, no MIDI drivers, no user interface, no thread synchronization, and no innate sense of time. This approach is essential to the purpose of libpd: to be embedded in a wide range of applications. It is the next big step for Pd after separating user interaction into `pd-gui` from the DSP process `pd`. When Pd was designed, uniprocessor systems dominated and editing operations were therefore included in the `pd` process [1]. Now the opposite is true, so splitting up operations onto multiple cores makes more sense [2].

While Pd does come with a vast collection of well tested audio and MIDI drivers, discarding this code simplifies the task of embedding Pure Data. The audio drivers that come with Pd itself are rather complex because they need to provide configuration options for the user interface, while integrating libpd into a new audio architecture can be as simple as taking boilerplate driver code and adding the libpd processing callback in the right place. Writing a new driver for libpd is typically much simpler than trying to adapt an existing driver from Pure Data. Additionally, in environments like OpenFrameworks, Python, and Processing, having no drivers means that libpd can use the native audio and MIDI I/O methods for each of those environments, simplifying development and deployment in that context.

As of Pure Data 0.42, the flow of time can be driven via a "plug-in scheduler", allowing regular calls to a function to drive the timing of both message processing and the DSP [2]. This allows libpd to remove all innate timing capabilities, and instead it provides a `process(...)` callback function to trigger the regular computation of messages and DSP ticks.

libpd keeps track of time only in terms of the number of samples requested so far (equivalently, the number of invocations of the processing callback). For example, when running with the default buffer size of 64 frames, at a sample rate of 44,100Hz, each invocation of the processing callback will advance the internal time of libpd by 64/44,100 seconds. Without invocations of the processing callback, time stops as far as libpd is concerned. Since invocations of the processing callback are typically driven by the audio interface, this means that libpd and audio interface will always be synchronized, without requiring further effort from the programmer.

How Pd should support threading has been a long debated and unresolved topic. Pd provides the thread synchronization primitives `sys_lock` and `sys_unlock` for working with threads,[2] but these functions have issues preventing them from becoming widely useful [3]. They were therefore removed from libpd.

libpd itself is not thread-safe because any decisions about thread synchronization are best made at a higher level of abstraction. For instance, the language bindings for Java and Objective-C

[2] http://lists.puredata.info/pipermail/pd-dev/2009-05/013565.html

add a thread-safe, object-oriented layer on top of libpd, while the Python bindings have no need for synchronization because of Python's global interpreter lock. The only drawback of removing `sys_lock` and `sys_unlock` is that multi-threaded externals may not work with libpd, but this seems like a small price to pay for the simplicity and flexibility of our approach.

On platforms that support dynamically loaded libraries (e.g., Android), it suffices to compile an external and place it on the search path, and then libpd will load it as needed. On platforms without dynamic loading (e.g., iOS), externals will have to be statically linked and client code has to explicitly call the setup method of each external, but the additional effort is negligible. Generally speaking, though, we expect that applications based on libpd will require fewer externals than traditional uses of Pure Data because many externals provide functionality that doesn't need to reside within Pd and is better supplied by client code.

5 Language bindings

In addition to the core library, libpd includes language bindings for Java, Objective-C, and Python. Wrappers for other languages, such as C#, may follow. The existing language bindings have a number of features in common. To wit, they make the functionality of libpd available to the target language, they convert between the custom data types of Pure Data and standard data types of the target language, they provide an object-oriented interface for the message passing functions and function pointers, and they are thread-safe.

Up to minor idiomatic adjustments, the Java API of libpd, implemented in PdBase.java, looks much like the C API. The most noticeable difference is that there is no init method because the static class initializer of PdBase implicitly initializes libpd, and the assignment of function pointers for receiving messages from Pd is handled by the JNI code.

In order to receive messages from Pd, client code implements the PdReceiver interface and registers a receiver object with PdBase. On top of this basic functionality, the Java bindings also include a utils package that provides a collection of convenience classes The bindings for Objective-C and Python are similar in spirit and design.

6 Pd for Android

The development of libpd began with a port of Pure Data to Google's Android operating system in the summer of 2010, using an earlier effort by Naim Falandino, Scott Fitzgerald, Peter Kirn, and Hans-Christoph Steiner as a starting point. At the time, the Native Development Kit for Android was already available, but apps still had to be primarily written in Java and there was not yet the ability to connect to audio input and output APIs via code written in C. These limitations turned out to be a stroke of luck, because they required the creation of a Java wrapper for Pure Data that would provide a small, clean API for signal processing and message passing. After much refactoring, the Java wrapper fell into two parts, namely the JNI glue that translates between Java and C and the C library that became libpd.

Audio development for Android poses a number of challenges, including large round-trip input-to-output latency (as much as hundreds of milliseconds, depending on hardware) as well as the need to support a wide range of devices Pd for Android works around this by providing a utility class, AudioWrapper.java, that papers over these problems as much as possible. In particular, the audio wrapper creates a credible illusion of synchronized audio I/O, and it includes a number of workarounds for known, device-specific peculiarities. While it cannot reduce the actual audio latency, which is beyond the control of application code, it protects developers from having to deal with latency issues.

When configuring the audio wrapper, developers only need to specify the number of audio frames per buffer that they would like to use (which implicitly defines their target latency), and then the audio wrapper will configure the audio subsystem for the smallest admissible buffer size that is a multiple of the requested one. Audio apps only operate on buffers of the requested size; if the platform improves and required internal buffer sizes shrink, users of the audio wrapper will automatically reap the rewards, without having to revise their code. We believe that this wrapper will be useful in many Android audio applications, not just when working with Pure Data.

The special structure of Android apps forces developers to make a number of decisions regarding the placement and management of audio components. In a nutshell, Android apps consist of Activities, which have a user interface, and Services,

which run in the background. Moreover, Services may be local, i.e., running in the same process as the current app, or remote, i.e., running in a separate process. Remote Services are more flexible and versatile, but they come at a significant cost in terms of complexity and overhead.

The Android API of libpd supports a number of different configurations. Simple apps consisting of only one Activity, such as the CircleOfFifths demo that is included in Pd for Android, can run the audio components directly, without having to launch a Service. Apps consisting of multiple Activities (such as the PdTest app that uses all major features of Pd for Android) will need to launch a Service that renders audio in the background and does not depend on individual Activities. Early versions of Pd for Android supported remote Services, but we decided that the flexibility of remote Services was not worth the extra effort.

The Android API of libpd has held up well in practice. In particular, it turned out to be powerful enough to build an Android version of the famous RjDj player, supporting all features of RjDj scenes that were publicly documented in the summer of 2010. It also drives other complex projects such as PdWebkitDroid, discussed below, and, in conjunction with OpenFrameworks for Android, Reactable for Android.

7 Pd for iOS

Apple's iOS has seen remarkable success in the area of music creation, largely due to the inclusion of the highly-optimized and mature Core Audio library and driver stack on all iOS devices. By layering libpd on top of Apple's Core Audio, we hide the complexity of Core Audio and provide a robust solution for many music and sound design projects targeting the large family of iOS-based devices. By now, there are dozens of iOS apps using libpd, including some that have been installed on millions of devices, like Reality Jockey's *Inception The App* (based on the *Inception* motion picture soundtrack).

By using libpd on iOS, developers can leverage their sound design and musical instrument logic across any platforms where libpd is supported, developers are insulated from the complexity of iOS Core Audio, and at the same time they can still have the benefits of a platform-specific native app with a fully native user interface and access to the entire toolbox of device capabilities such as the accelerometer, multitouch display and rich, accelerated graphics.

Apple's iOS is written in C (especially low-level libraries such as Core Audio) and Objective-C and thus natively supports C language software development. As libpd is written in C, it can be easily compiled, linked with and called from Objective-C iOS app projects.

libpd on iOS includes a couple of utility classes to provide a high-level Objective-C interface to the library. These utility classes facilitate initialization, patch loading and saving and message processing in a manner consistent with Objective-C-based iOS development. It is also possible for an app to bypass these utility classes and use the libpd C language interface directly.

The iOS platform is developing at an accelerated pace. This dynamic nature of the iOS platform presents ongoing challenges to a library like libpd that aims to support a broad range of applications across the widest range of possible configurations. The iOS family of devices now includes more than a dozen different models with specifications ranging from 128MB to 512MB of dynamic RAM and processing capabilities ranging from a single-core 412MHz armv6 CPU to a 1GHz dual-core armv7 CPU with a vector-processing unit. Also, the iOS operating system capabilities continue to rapidly evolve, with major system releases at least every year. Operating system releases often drop support for discontinued, older devices, yet these devices are still being used by millions of people throughout the world, and many developers using libpd want their software to be compatible across a large cross-section of devices and OS configurations.

On iOS, we have encountered minor differences in handling of floating point arithmetic across different devices as well as OS-version specific differences in Core Audio behavior. libpd attempts to abstract these differences as much as possible. Developers attempting to use libpd across a wide range of devices and systems must plan for additional configuration testing of their applications. The older iOS devices have significantly less RAM and no floating point unit, and they may not have enough power to run many Pd patches that run fine on a typical PC.

In addition to the challenges presented to libpd by the dynamic nature of the iOS platform, there are many other areas of possible development to better support libpd's objectives on iOS, including the use of multiple cores on high-end de-

vices and optimizations using ARM NEON vector-processing as well as the Accelerate Framework. We are aiming to take advantage of such platform-specific optimizations without fragmenting libpd.

8 Pd for Processing

As a free software environment for sound and music, Pd pairs well with the graphics and interaction capabilities of Processing, originally designed by Ben Fry and Casey Reas. The Processing API[3] employs a simplified syntax and full compatibility with target environments in Java and (via Processing.js) JavaScript, and has become a popular choice among artists, designers, and in education. Like Pd, it has also been traditionally friendly to creative communities who lack a background in computer science, who are new to programming, or who might not consider themselves expert programmers.

Pd for Processing, therefore, becomes an important initiative for pedagogical purposes. It means that in a single, beginner-friendly environment, students and artists can experiment with mixing music and visuals, or interactive scores and sound with games. Previously, Processing users would often connect Processing with Pd by starting both applications, then sending OpenSoundControl messages between them. libpd removes the need for OSC and supporting libraries, by allowing the two tools to interface directly using Pd's messages. With support for the array datatype in Pd, it is even possible, for instance, to load and play an audio file via Pd, while visualizing its spectra in Processing.

Because its capabilities are inherited from the Java platform, and because of the limitations and inconsistencies of JavaSound, Processing has thus far lacked a stable, full-featured, high-performance audio library. A JavaSound and Tritonus-based library called Minim ships with Processing, but because of its dependency on encumbered audio libraries, should not be considered a long-term, cross-platform solution in a future increasingly centered around OpenJDK.

Given the lack of a standard audio API in the Java version of Processing, therefore, Pd for Processing must be bundled with some means of interfacing with audio and MIDI. Processing users expect drag-and-drop operation, not manually writing their own audio I/O code. Initially, the library includes Peter Brinkmann's JackNativeClient, a streamlined adaptation of his Java wrapper for the JACK audio server, which is presently supported on Mac OS X and Linux.

The combination means that a Processing sketch with audio capabilities provided by Pd can act as a sound source or effect processor in conjunction with other audio tools, like Ardour or Ableton Live. With help from the Processing community, future versions of the library will provide additional library support for native audio and MIDI capabilities on Mac OS X, Linux, and Windows. Ideally, at some point, these will be stable enough to ship separately or even be bundled with the Processing distribution. Future possibilities for work with Processing branches include making supporting Processing for Android more seamless via the standard libpd for Android distribution. (For now, it suffices to simply use the existing Processing for Android project alongside libpd for Android, as with any other Android application.)

9 Pd for OpenFrameworks

ofxPd[4] for OpenFrameworks, developed by Dan Wilcox using libpd, demonstrates how Pd can be adapted in a similar form to other uses of libpd. OpenFrameworks provides Processing-like syntax in a C++ environment. Unlike Pd for Processing, ofxPd may safely rely on the user to connect the output of libpd to an audio API, particularly since OpenFrameworks ships with stable audio implementations (currently, based on rtaudio, which may also prove a viable option for Java). In only a couple of months, ofxPd has provided support for OpenFrameworks 0.62 and 0.7x branches, and has been employed in NodeBeat, a shipping, commercial iOS application for iPad, iPhone, and iPod touch. NodeBeat uses ofxPd and OpenFrameworks for iOS rather than the standard iOS libpd. Dan Wilcox is also adapting libpd to a wearable computer suit for live performance.

10 Pd everywhere

Interest in Pd as an audio engine for video games and other applications has been evident since at least as far back as 2003 [4]. Electronic Arts has been using Pd as an internal prototyping tools for at least ten years, and more recently they included Pd as an audio engine in the game "Spore."[5] With libpd, we have made this paradigm

[3] http://processing.org/reference/
[4] https://github.com/danomatika/ofxPd
[5] http://lists.puredata.info/pipermail/pd-list/2007-11/056300.html

more accessible to game developers, application makers, and even synth hardware enthusiasts everywhere.

To illustrate the use of libpd as the audio engine for a video game, consider one of the Python examples, pygame_fun_test.py, that ships with the libpd source code. This example utilizes the Python bindings for libpd together with the Pygame 2D game library. In this setup, libpd handles the audio rendering and Pygame handles the graphics and event handling, and copies the audio data from libpd to the Pygame sound buffers. An extrapolation of this example to real video games could be based on this simple example. Combinations like this allow rapid prototyping of all aspects of the game engine and the potential for deeper and more complex audio in those games. One could employ the same technique with Blender 3D's game engine via the new Audaspace API, for example, or any other game engine in a language with bindings for libpd.

A further example of libpd as the audio engine for an application is found in the PdWebkit-Droid Android app.[6] It demonstrates communication between JavaScript, running in Android's WebKit-based browser, and Pd patches that produce sound via libpd. This app allows people without high level programming experience to quickly and easily develop interactive, Pd-based applications. Anybody who knows how to make a web page in HTML and script in JavaScript is able to create a graphically rich frontend that communicates and controls their Pd patches running on an Android phone in a self-contained app.

Another area of considerable interest is that of Pd as the embedded audio engine for hardware synthesizers. Whilst this area is relatively new, developers at BUG labs in New York have been experimenting with libpd on embedded hardware.[7] One important feature of libpd is that it does not depend on outside libraries, and so we can easily conceive of dedicated audio devices that allow Pd patches to be uploaded directly into the device, for example. We hope to see more development in this area in the future.

11 Outlook

libpd is a thin layer on top of Pd Vanilla, requiring no changes to Pd itself. So far, libpd has mostly been tracking the development of Pure Data itself, but we expect that new requirements arising from libpd-based applications will drive future developments of Pd. This has already happened on a small scale, with the acceptance of a number of patches that came from libpd developers.

We hope that Pd will be redesigned to admit multiple independent instances. One single instance is enough when running Pd as a stand-alone application, but it is a problem when running Pd as a VST plugin, for example. A related problem is thread safety. The redesign of Pd for multiple instances will bring an opportunity to revisit the matter of thread synchronization and introduce a fine-grained, implicit, and encapsulated approach to thread safety that no longer relies on public locks.

In the long term, we hope to see Pd itself restructured as an audio library with separate modules for audio drivers, MIDI drivers, and user interfaces. In other words, we hope that future versions of Pd will be structured like libpd applications, and that libpd as a separate project will no longer be necessary.

12 Acknowledgments

libpd grew out of Peter Brinkmann's Android port of Pure Data, building on an earlier effort by Naim Falandino, Scott Fitzgerald, Peter Kirn, and Hans-Christoph Steiner. The Objective-C bindings were developed by Peter Brinkmann, Dominik Hierner, and Martin Roth. Richard Lawler contributed sample projects for iOS. Rich Eakin provided an improved way of handling patches. Chris McCormick integrated libpd into Webkit and Pygame. Damian Stewart and Dan Wilcox integrated libpd into OpenFrameworks. Peter Brinkmann and Peter Kirn created the Processing branch of libpd. We are grateful to all contributors to Pd and libpd, to Reality Jockey Ltd. for releasing the Objective-C bindings as open source, and to Miller Puckette for creating and supporting Pd.

[6] http://code.google.com/p/pd-webkit-droid/

[7] http://lists.puredata.info/pipermail/pd-list/2011-04/088285.html

Appendix: Sample code

The following listing shows a simple C program that uses all major features of libpd. Most importantly, it hooks up a receiver function for print messages from Pd, sets audio parameters, loads a patch, and processes ten seconds worth of samples. Note that this program sends the message [; pd dsp 1] to Pd before invoking the process method. Most applications of libpd will enable audio processing at the beginning and then leave it enabled. Audio processing can be paused by simply not invoking the process method.

Listing 1: Using libpd directly

```
// Bare-bones example of libpd
// in action

#include "z_libpd.h"

void pdprint(const char *s) {
  printf("%s", s);
}

int main(int argc, char **argv) {

  // Init pd and assign print hook
  int srate = 44100;
  libpd_printhook =
      (t_libpd_printhook) pdprint;
  libpd_init();

  // Configure audio:
  //   one input channel
  //   two output channels
  //   44.1kHz sample rate
  //   one Pd tick (64 frames) per buffer
  libpd_init_audio(1, 2, srate, 1);
  float inbuf[64], outbuf[128];

  // Compute audio
  // [; pd dsp 1(
  libpd_start_message();
  libpd_add_float(1.0f);
  libpd_finish_message("pd", "dsp");

  // Open patch
  void *patch = libpd_openfile(
         argv[1], argv[2]);

  // Run pd for ten seconds
  int i;
  int nTicks = 10 * srate / 64;
  for (i = 0; i < nTicks; i++) {
    // fill inbuf here
    libpd_process_float(inbuf, outbuf);
    // use outbuf here
  }

  // Close patch
  libpd_closefile(patch);
  return 0;
}
```

Our next listing shows some crucial pieces of an iOS app that uses libpd. It illustrates how the Objective-C bindings of libpd on iOS provide an object-oriented callback mechanism for handling messages and data sent from Pd. This mechanism uses the informal protocol PdReceiverDelegate. We implement the receivePrint of the protocol and then register the receiver with the setDelegate method of the base class PdBase.

Listing 2: Using libpd with iOS

```
// Excerpts from an iOS app using libpd

// Receiver method for print messages
// from Pd
- (void)receivePrint:(NSString *)message {
  NSLog(message);
}

// Initialize libpd
[PdBase initialize];
[PdBase setDelegate:self];

// Initialize audio
[PdBase computeAudio:YES];
PdAudio *pdAudio = [[PdAudio alloc]
        initWithSampleRate:44100.0
        andTicksPerBuffer:64
        andNumberOfInputChannels:2
        andNumberOfOutputChannels:2];

// Open a patch and play
PdFile *patch =
    [PdFile openFileNamed:kPatchName
            path:bundlePath];
[pdAudio play];

// Send a message to Pd
[PdBase sendBangToReceiver: @"voicebox"];
```

The next listing shows a sample sketch in Processing. Note that much of the setup is hidden from the developer. For example, if the programmer simply overrides the pdPrint method, the Processing library will implicitly register this new method as a receiver for print messages from Pd.

Listing 3: Using PureDataP5

```
import processing.opengl.*;

PureDataP5 pd;

void setup() {
 size(400,300,OPENGL);
   pd = new PureDataP5(this, 0, 2);
   pd.openPatch("testtone.pd");
   // Will auto-connect to system outputs.
   pd.start("system", "system");
}
```

```
void draw() {
 // Move the mouse to change the frequency
 float frequency =
    map(mouseY, height,0,60,1200);

 // Send a float to Pd.
 // It'll be received at the symbol freq.
 pd.sendFloat("freq", frequency);

 // Move the mouse left or right
 // to change the mix
 float left = map(mouseX,0,width,0.0,0.5);
 float right = map(mouseX,width,0,0.0,0.5);

 pd.sendFloat("left", left);
 pd.sendFloat("right", right);
}

void pdPrint(String s) {
 println("print: " + s);
}
```

Finally, we show a small example for PdWebkitDroid. In this case, almost all the setup is implicit. PdWebkitDroid will automatically load a patch called _main.pd, configure audio drivers, and connect receivers. The HTML file only contains the code that is needed to respond to messages from Pd, and to connect HTML widgets to Pd.

Listing 4: Using PdWebkitDroid

```
<html>
<head>
  <title>PdWebKit Sample</title>
  <script>
    function PdReceive(msg) {
      alert("From Pd: " + msg);
    }
    function start() {
      Pd.send("started", "bang");
    }
  </script>
</head>
<body onload="start();">
  <input type='checkbox'
       onchange='Pd.send("pressed",
                this.checked * 1.0);'
       name='right' checked >
</body>
</html>
```

References

[1] Puckette, M. 1996. "Pure Data: another integrated computer music environment." *Proceedings, International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 224-227. http://crca.ucsd.edu/~msp/Publications/icmc96.ps

[2] Puckette, M. 2009. "Multiprocessing for Pd." *Proceedings, Pure Data Convention.* São Paolo, Brazil. http://puredata.info/community/projects/convention09/puckette.pdf

[3] Grill, T., Köcher, H., Blechmann, T. 2007. "Enhancements to the pd developer branch initiated by the vibrez project." *Proceedings, Pure Data Convention.* Montréal, Québec, Canada. http://artengine.ca/~catalogue-pd/30-Grill.pdf

[4] Paul, L. 2003. "Audio Prototyping with Pure Data" *Gamasutra,* May 30. http://www.gamasutra.com/view/feature/2849/audio_prototyping_with_pure_data.php