Programming Sound with Pure Data

Make Your Apps Come Alive with Dynamic Audio

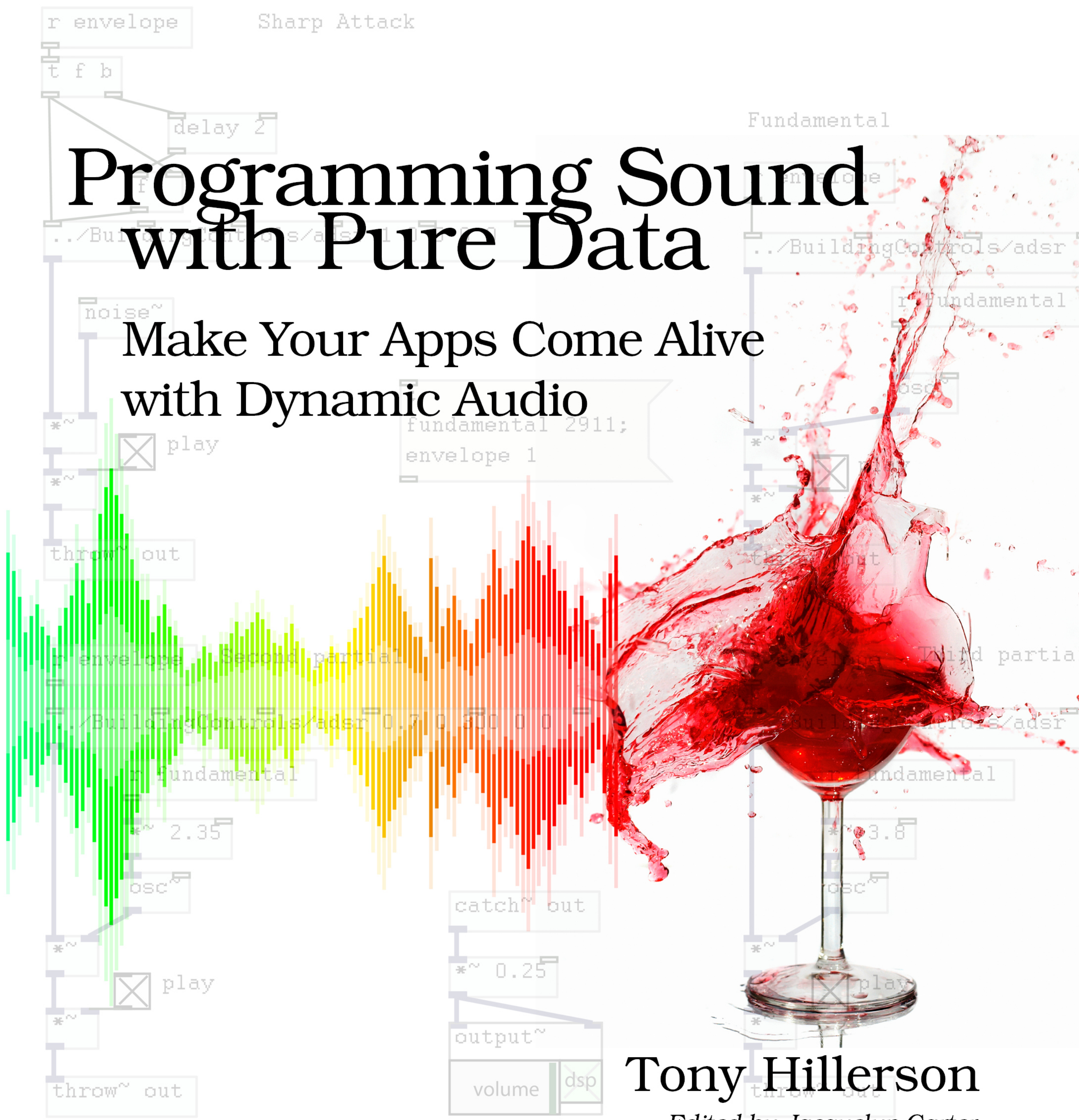
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Sound with Pure Data

Make Your Apps Come Alive with Dynamic Audio



Tony Hillerson

Edited by Jacquelyn Carter

Programming Sound with Pure Data

Make Your Apps Come Alive with Dynamic Audio

Tony Hillerson

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93778-566-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January, 2014

It’s time to have some fun. In this chapter we’ll start to apply what we know about Pd and use the tools we’ve built to create some sound effects. The sounds are relatively simple to create, but are surprisingly realistic for the amount of work it takes to create them.

We’ll start with the simplest patch, simulating waves with white noise, a filter, and a low-frequency oscillator (LFO). Then we’ll build up a bit with a patch simulating wind with noise in both the signal and control domains, and an envelope for more control. Finally, we’ll do our first more technical sound analysis by looking at a recording of a wineglass being tapped, and then reproducing the sound it makes.

When you are done with this chapter, you will

- Understand what noise is, and how it can be used
- Know how to shape sound the way you want
- Know the principles behind two types of synthesis: subtractive and additive
- Have a basic understanding of sound analysis

Each section in this chapter follows the pattern of a general description of the sound we want to make, an analysis of what’s going on in the sound in the real world, the approach we’ll take in Pd to reproduce the sound, and a discussion of the patch. From now on we’ll start with completed patches or subpatches and break them down instead of talking through building the patches step by step.

Let’s get started by making a simple patch that reproduces the sound of waves at a beach.

Waves

The sound of waves at a beach is a nice, calming sound, and is a great one for our first shot at designing environmental ambience. If you’ve ever been to the beach, you can probably recall the waves as an undulating, mellow, hissing sound that repeats over and over. It depends a bit on the makeup of the beach, but the common element is the water.

I remember visiting a beach, French Beach, in British Columbia. The beach there isn’t sand; it’s made up of fist-sized rocks, and as the waves rolled in and out they made this wonderful *tocking* sound multiplied countless times by the sheer number of the rocks that added an uncommon and interesting element to the waves. I’d love to be able to capture and analyze that sound. For our purposes, though, let’s consider a sandy beach—it’s familiar and relatively easy to approximate with some simple tools in Pd, so it’s a great place to start practicing sound design. Let’s begin with a simple analysis.

Analysis

The first thing to note about waves is their periodic nature. That's one of the reasons they have such a lulling effect on you as you sit on the beach—the regularity of the sound of the water building up forward motion, reaching up the beach, and then flowing back down the sand into the body of water over and over again.

The second thing to note is the materials involved. The effect of water flowing around and through itself and over and around the material of the beach is a complex physics problem to describe, involving hydrodynamics and a whole set of interesting theorems. Our model doesn't have to take that deep of an approach, though, because the sonic effect of all this activity can be simulated with *noise*. In sound production, noise has a more specific definition than we use in day-to-day speech; it means some sort of random signal.

Noise is classified into different *colors*, drawing an analogy between sound and light and the distribution of energy across the visible spectrum. *White noise* is a random signal with a uniform distribution across all frequencies. For comparison, there is also *pink noise*, with the frequencies weighted toward the lower end, and *blue noise*, with the frequencies weighted toward the higher end of the spectrum. Because of the way our ears perceive sound, white noise sounds higher-pitched than you might expect from the even frequency distribution. Most times you hear *noise* in this technical sense you're probably hearing about white noise. This is the case in Pd, so we'll stick with "noise" instead of "white noise."

Noise is close to the *spectrum* of water flowing in and around sand while the grains of sand raise and settle, bumping into each other. The spectrum of a signal is the distribution and strength of the frequencies in the signal, and reproducing the spectrum of a sound is the greater part of reproducing the sound.

Approach

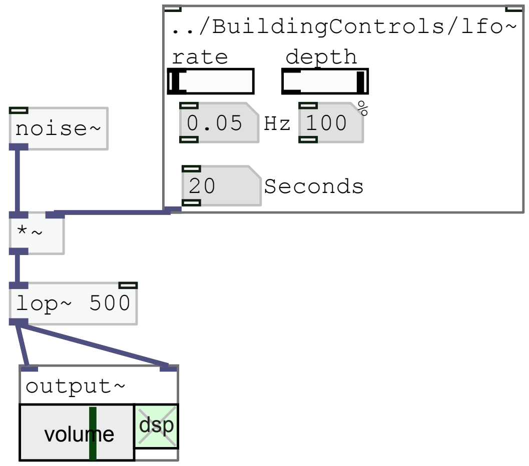
We'll model the sound of waves on a beach by doing the following:

- Using an LFO to model the timing
- Using a noise generator to model the spectrum

Pd supplies us with an easy way to make white noise. Unprocessed white noise sounds unnatural, so we'll tune it to our taste using a filter, which removes frequencies from a signal. Since we don't have any more specific requirements for reproducing the sound of waves, we're free to apply a filter and then adjust it until it sounds right.

The Patch

Create a new patch named waves.pd and save it. Reproduce the patch shown in this figure.



First, note that the sample code has this patch in a sibling directory to the directory containing the patches from the last chapter, including the lfo~.pd subpatch. Pd will load subpatches from relative paths, and in this case the path to the subpatch is ../BuildingControls/lfo~.pd. You can either use the relative path to the LFO subpatch from your system or copy the subpatch into the same directory as this patch and refer to it as lfo~. The LFO is then connected to a *~ to regulate the amplitude of the noise.

The aptly named noise~ object connected to the other side of the *~ generates white noise. The lop~ between the *~ and the output~ is a filter to tune the noise a bit. "Lop" stands for *low-pass filter*, which is a filter that cuts off frequencies above a given frequency, which it takes as a numeric argument, and allows lower frequencies to pass through and remain in the signal. In the figure the lop~ has a relatively low cutoff frequency, but try different ones out and see what you think.

The period of the waves can be controlled with the rate of the LFO, but notice how rates of less than 10 seconds start to sound unnatural. The depth of the LFO is also subtly important to the effect; a depth of 100% sounds unnatural because waves rarely become completely silent. Try something between 75% and 80%.

Experiment and play around: imagine the sound of seagulls and boats in the background, or perhaps children laughing and playing, and you can see how amazing it is that some basic building blocks can produce realistic sounds. Next we'll explore using noise a bit more as we reproduce the sound of wind.

Wind

Wind, too, is a great ambient effect that's relatively easy to reproduce using synthesis. Let's consider what kind of wind we want to reproduce. To get the most out of the effect let's make it a fairly windy day in something like a grassy field with a few trees, so we'll hear a breeze with light gusts every now and again, but sometimes experience stronger, more steady gusts. We'll make it so that the stronger gusts are controllable, which could be useful to tie to some event taking place on a game screen, for instance.

Analysis

As with the waves patch, this analysis is intuitive rather than scientific. The method of reproducing wind is similar to waves in that white noise can reproduce the spectra created by the chaotic flow of air around materials of different shapes.

One big difference is that wind is nonperiodic, unlike waves on a beach. Wind blows harder, then softer, with gusts at unpredictable times. One last thing to note is that wind is a subtler sound than waves, so we'll want to use a more drastic filter to pick out the frequencies we want to be in the signal.

Approach

Let's consider the structure of the patch in both the signal and control domains.

Signal Domain

As with waves, the signal domain will be based on noise. To get the spectrum we want we'll use a `bp~`, Pd's *band-pass filter*. Whereas a *low-pass filter* allows only frequencies below a certain frequency to pass through, a *band-pass filter* allows only frequencies a certain distance *around* a given frequency, which are called the *passed band*. Band-pass filters usually have a parameter called *Q*, which controls the width of the passed bands in an inverse relationship, so higher Q values pass a narrow band and lower values pass a wider band.

Control Domain

We have two goals in the control domain for this patch:

- A steady, quiet wind with small, random gusts
- A way to control and produce louder gusts for a period of time

We'll control starting and stopping the wind and producing the louder gusts with an ADSR (attack, decay, sustain, and release) envelope. That part is much like the test patch we made last chapter. The small, random gusts are a little more tricky. We need some way to introduce randomness into the control domain.

Luckily, we don't have to look far, because that's exactly what noise is: a random signal. So we'll use `noise~` objects in both the signal and control domains, one for the wind sound and one to produce gusts, which are just small amplitude jumps to the signal. We'll also use a series of filters to regulate how much effect the control noise has.

We'll split the patch into the wind-making mechanism and the controllable part by using a *subwindow*, or internal subpatch: a subpatch that's not stored in a separate file, but rather in the patch itself.

The Patch

First let's work on the main patch. Create a patch in the same directory as `waves.pd` and call it `wind.pd`. Then create an `adsr~` either by copying the one we made to the same directory or by referring to it with a relative path. Connect the `adsr~` to a `*~` and both sides of the `*~` to an `output~`. You should have something like in Figure 7, *The Main Patch for Wind*, on page 10.

The left side of the `*~` has a new type of object connected to it, called `pd wind`. Create this now. Notice that once you click away from editing the object, Pd automatically opens a new window for you. This syntax creates a subwindow.

Subpatches and subwindows are both great ways to build some abstraction into your patches and focus on the various responsibilities of the different parts of a patch. In this way, Pd is like any other programming language. Subwindows are just like subpatches, but you create subwindows when the function you want to perform is specific to the current patch, and subpatches in separate files when you want to build a reusable component.

The wind subwindow looks like Figure 8, *The Wind Subwindow*, on page 10.

The left side of the window under the comment "Wind signal" starts off with a `noise~` object connected to a `bp~`. This is the band-pass filter discussed earlier. The arguments `800 1` correspond to the center frequency of the filter, 800 Hz, and the Q of the filter, 1. This is a low Q value, which means the band is fairly wide. Here again, 800 Hz sounds fairly good as the center band, but it's

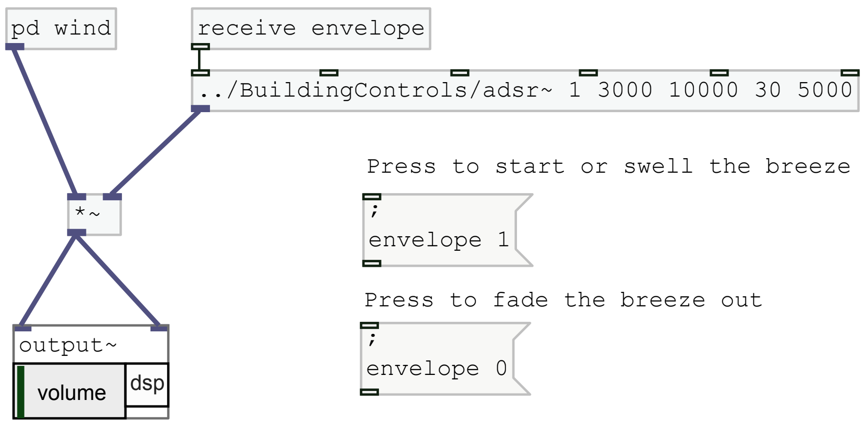

pd wind
receive envelope
../BuildingControls/adsr~ 1 3000 10000 30 5000
Press to start or swell the breeze
; envelope 1
Press to fade the breeze out
; envelope 0
*~
output~
volume
dsp

Figure 7—The Main Patch for Wind

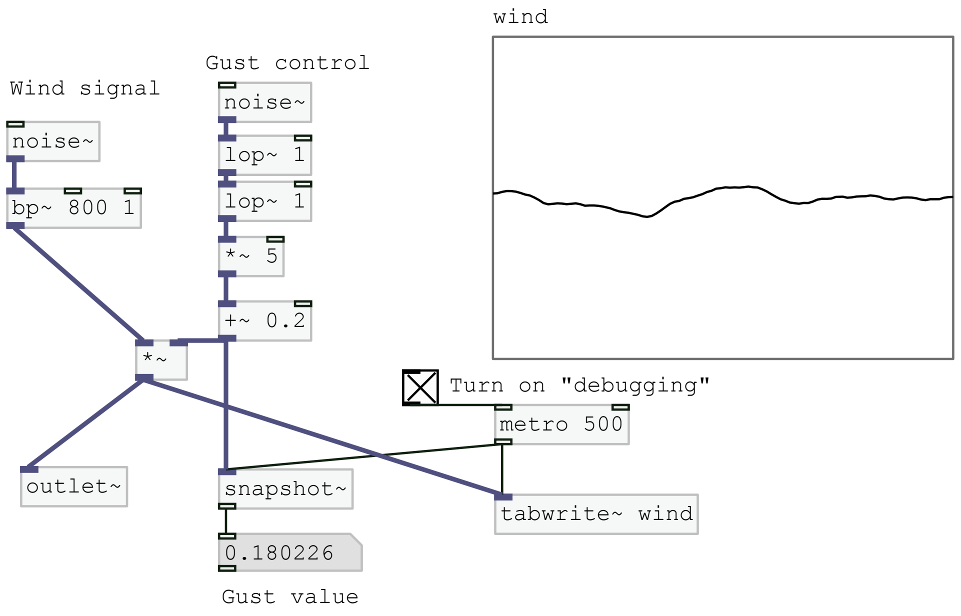


Figure 8—The Wind Subwindow

not scientifically chosen. Play around with different frequencies and Q values to get an idea of how they sound.

The signal comes out of the `bp~` and into a `*~` so we can vary it with the control signal simulating random gusts. That signal chain also starts out with a noise~, but then we pass the signal through a series of very drastic lop~ filters set to 1 Hz. Filters don't lop the sound off (pun intended) directly at the filter frequency, but rather they have a curve to them, so using two filters in series like this makes the cut-off steeper than using one. These filters keep out almost all of the signal from the noise~ object so that only a very subtle signal gets through. It's so subtle that we have to increase the magnitude by 5 with a *~ to get it in a range where the effect will be noticeable.

We don't ever want the control signal to reach 0, because that would stop the sound of the wind. To avoid that we add 0.2 to the control signal with a +~. The control signal is connected to the right side of the *~ to regulate the signal from the left side.

So that we can see what's going on, there is also a graph of an array called wind, a snapshot~ of the control signal fed to a number, and a metro 500 controlled by a check box, which you can turn on to debug the patch. When the patch is running, watch the gust value and notice the small amount of amplitude regulation.

The randomly regulated signal is sent through the outlet~ and back to the main patch. There, the message boxes containing envelope 1 and envelope 0 require a little explanation. This syntax where the message contains a semi-colon (;) followed by a new line is a way of broadcasting a named value. To understand this better, consider the receive envelope object connected to the adsr~. When you press the message containing envelope 1, Pd sets a global variable named envelope to the value 1, and any receive objects with an argument envelope will be triggered, sending the value of envelope to their outlets. The same goes for the message containing envelope 0.

The 1 and 0 values sent to the receive connected to the adsr~ serve to trigger and stop the envelope, which starts and stops the wind. Note the arguments to the adsr~, in order:

- 1—An attack value of 1 means that the attack portion of the envelope will reach 1.
- 3000—This attack rate in milliseconds means the attack portion of the envelope will take 3 seconds.
- 10000—The envelope decay will take 10 seconds.
- 30—The sustain value will be 30% of the attack value, or 0.3.
- 5000—The envelope will stay at 0.3 indefinitely until it receives a 0, when it will take 5 seconds to reach 0.

And that's our wind patch! Try it out; click the `envelope 1` message and listen to how the wind swells and then drops. Listen to how there are small gusts blowing through randomly. Then when you want a larger sustained gust press the `envelope 1` message again. Remember that this could be controlled inside of a game to coincide with wind control that also affects what the player sees on the screen. When you want the wind to fade out, press the `envelope 0` message.

Next we'll take a little bit more time to analyze a real-world sound and reproduce it. But before we do, let's take a moment to talk about a little theory behind what we're doing here.