Extracted from:

Programming Sound with Pure Data

Make Your Apps Come Alive with Dynamic Audio

This PDF file contains pages extracted from *Programming Sound with Pure Data*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2014 The Pragmatic Programmers, LLC.

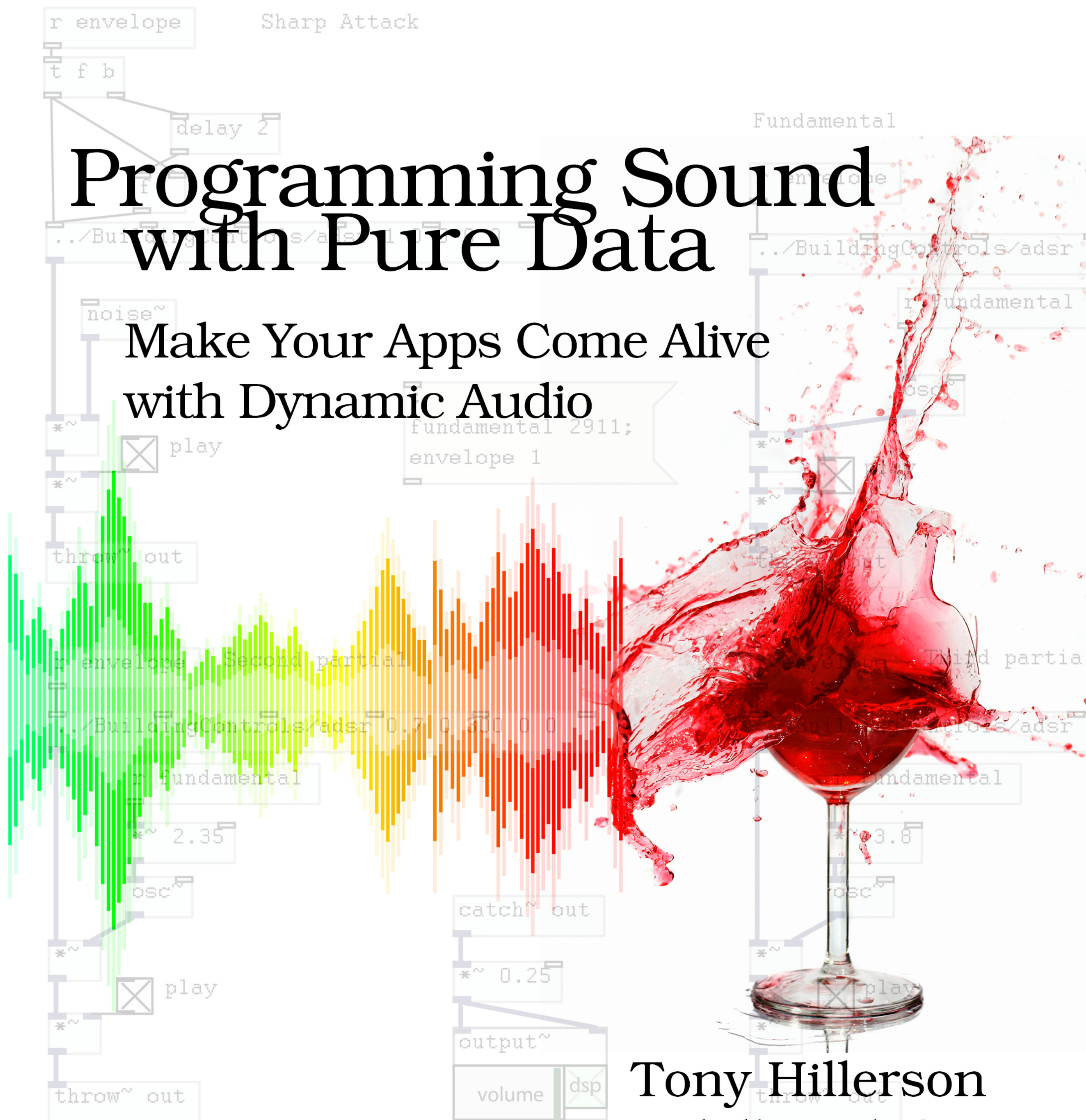
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

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)

Copyright © 2014 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

Designing and Building the Patch

Keeping in mind the goals we have for the app, let's design a patch that will do what we need.

Design Goals

We've decided we'll use musical tones to create the feel we want for the app, including encouraging the user to complete tasks. Now let's consider the type of sound we want. There's nothing scientific here; in other words, we're not trying to make a sound effect to match a real-world sound. As with most musical things, this is largely a matter of taste. However, we want to place a few constraints on the design, given the fact that the target is a mobile device.

- We want a clear, short sound.
- It must cut through background noise well on mobile speakers.
- We want an upbeat, positive feel.

For those reasons, let's use a bell-like tone for the sound, one that has a bit of an attack and is harmonically rich. This will help it cut through better on small speakers, and will make a nice, interesting sound.

Designing the Sound

There are a number of ways to make a bell-like sound with synthesis. We did much the same thing when we modeled the wineglass strike, adding the exact frequencies we wanted using additive synthesis. We could also use filters to cut away or emphasize the frequencies we wanted: subtractive synthesis. Or we could capture the sound we wanted and play back the sample or use it to build a wavetable for wavetable synthesis.

There's one more type of synthesis we'll look at now, though, which perfectly fits the criteria we have. It's called *frequency-modulation synthesis*. It was discovered in the '70s and developed in the '80s as a way to make complex timbres with only a few oscillators, which made it great for the low-powered musical synthesizers of the time. It also is characterized by harmonically rich and interesting bell-like or metallic tones.

FM Synthesis

The idea of frequency-modulation (FM) synthesis is to use one oscillator, called the *carrier*, to produce the desired musical note, and another oscillator, called the *modulator*, to change the carrier's frequency. The result is that the output's frequency has a very complex harmonic character for the processing

• Click HERE to purchase this book now. discuss

cost of just a few oscillators. Note, however, that oscillators are usually called *operators* when speaking about FM.

Designing Our API

Now let's talk about the patch we want to create. We want it to work inside the native mobile apps and have some controls exposed to them.

Our goal is not only to build a patch to create the tones we want, but to add the configurability to allow the patch to be controlled the way we want from inside the mobile apps we create. I find that it helps to think of this as no different from creating a code module or API that we expect to expose to some other software.

We know we want a way to play a distinct musical tone, so we should have a way to specify the note we want the patch to play. Pd can easily receive MIDI messages, which you'll recall are note and control messages passed to and from digital musical instruments. In this case, since we don't plan on using a musical instrument to control Pd, we don't really need to use a MIDI interface. It's still convenient to use MIDI to specify the notes we want to play, however, so we'll have a way for the patch to receive a MIDI note number and play it.

As an extra feature thrown in at the last minute (as software developers we wouldn't feel at home without a few of these), let's also expose a few other settings that allow the native apps to slightly change the characteristics of the tone so the Android app can sound distinct from the iOS app.

The Task-App Patch

Now we'll walk through the patch. Follow along if you like by creating a patch in a working directory. To ship the patch with the app, we'll need to copy the patch into the Android and iOS app directories. For iOS it's enough to include the patches in the main bundle. For Android, the patches need to be zipped up and placed in the `raw` resources directory. More about Android later, when we go over the Android code.

We'll start at the top of the patch, and then work through some abstractions. You can find the final patch in the code download.

The Top-Level Patch

To begin with, look at the top-level patch in Figure 36, *The Top-Level Task-App Patch*, on page 8. There are two "voices," or subpatches, named `bellvoice~` with a number of `receive` objects connected to inlets and the same arguments

sending output into throw~ out objects. The catch~ out object is connected to a dac~.

We have two separate bellvoice~ subpatches so we can play two notes that sound at the same time. In most musical synthesizers this capability is referred to as the number of *voices* it supports. Notice that the top voice has an r midinote1 and the bottom has an r midinote2 connected to the first inlets. We'll send a message to each of these MIDI note receivers when we want the two separate notes to sound. We have here, then, an FM synthesizer with two voices.

The rest of the receive messages make up the rest of the API for the patch. There are two sets of tune and depth messages because each bellvoice~ has two operator pairs, for a little extra character to the sound. The tuning value isn't the frequency we want the modulators to have; it's actually a multiplier of the note frequency sent to the carrier. So, the modulation frequency is always expressed in terms of the carrier frequency in this design.

The MIDI note messages control the frequency of the carrier operators, and the other messages control the characteristics of the modulators. The messages are as follows:

- mod1_tune: tuning for the first modulator, as a ratio of the first carrier frequency
- mod2_tune: tuning for the second modulator, as a ratio of the second carrier frequency
- mod1_depth: the depth of the first modulator
- mod2_depth: the depth of the second modulator

Going back to what we covered about FM, the tuning of the modulators will set their frequency, which is the rate at which the carriers change their frequency. The depth messages will control the modulators' amplitude. One way of looking at this is how much the modulators modify the carrier, so I've used the word *depth*. In FM this is often referred to as the *index* of the modulator, so keep that in mind. The specs for FM are sometimes a little academic sounding.

Those are the messages we'll use as our API for the mobile apps to interact with. Five other arguments are passed to the bellvoice~ subpatch, however, so let's talk about those, and the subpatch itself, next.

• Click HERE to purchase this book now. discuss

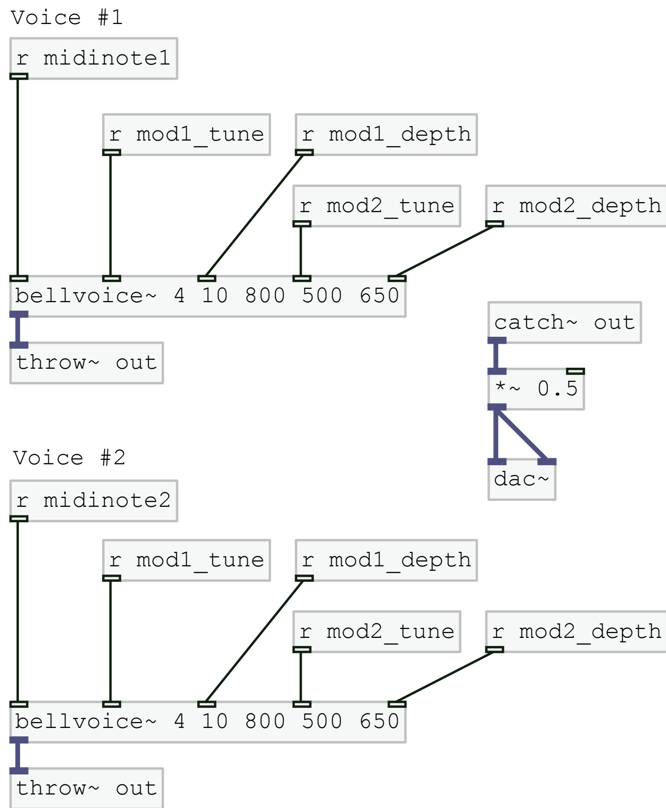
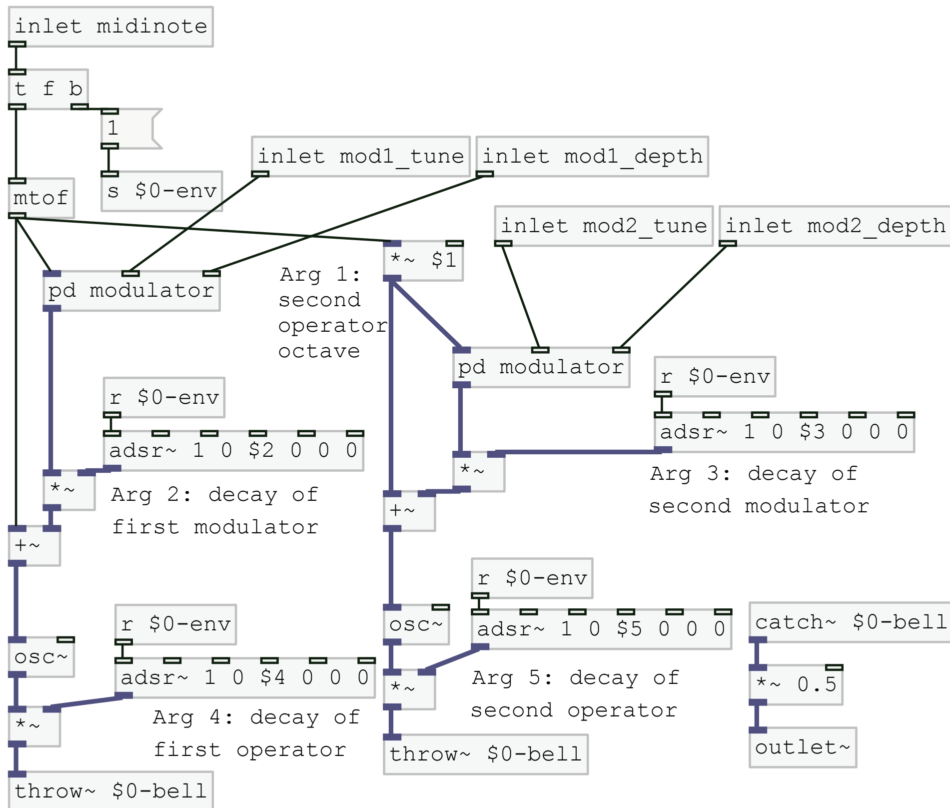


Figure 36—The Top-Level Task-App Patch

A Single Voice

The following figure shows the image for a single voice subpatch.



This subpatch is saved in the same directory as the top-level patch and named bellvoice~.pd. The patch is fairly dense, but conceptually there are two operator pairs. Each operator pair is composed of a carrier (which is simply an osc~) and a modulator (which is a subwindow, pd modulator).

The modulators each accept the tune and depth inlets and are connected to +~ objects. In this way they send out a signal of a certain frequency and amplitude, depending on their settings, which is summed with another frequency, thus modulating it. That frequency is sent to the osc~ objects, which send their signals to the outlet using a throw-and-catch pattern.

The original frequency is derived from the inlet midinote fed into the mtof object, which turns a MIDI note number into a frequency float value. This is how we specify the tone of the note each voice plays.

Finally, both the carriers and modulators are multiplied by asdr envelopes so that they can each be given some timing control. Giving a modulator a longer or shorter decay time than its carrier will change the character of the sound you hear while the note plays. These envelopes are each labeled for clarity

• Click HERE to purchase this book now. discuss

because the decay values are all dollar-sign variables, taken from the arguments to the bellvoice~ subpatch on the parent patch.

Notice that the adsr~ subpatch is not, as it was in other examples, inside of the BuildingControls directory, but rather in the same directory as this set of patches. This is because we must copy all the patches we need into the native app project to build them into the app.

The last thing to note is that the second operator pair on the right of the patch has its signal multiplied by the value in $1 first. This allows us to layer the second pair some octaves above or below the first, which produces a more interesting sound. This isn't a necessary ingredient of the patch; I just thought it sounded better. While you design patches, don't forget to play and experiment!

A Modulator Operator

Finally, the subwindow content of each pd modulator looks like the following image—first, an inlet~ for the frequency being sent to the carrier; second, the tuning multiplier inlet; and then the depth inlet.

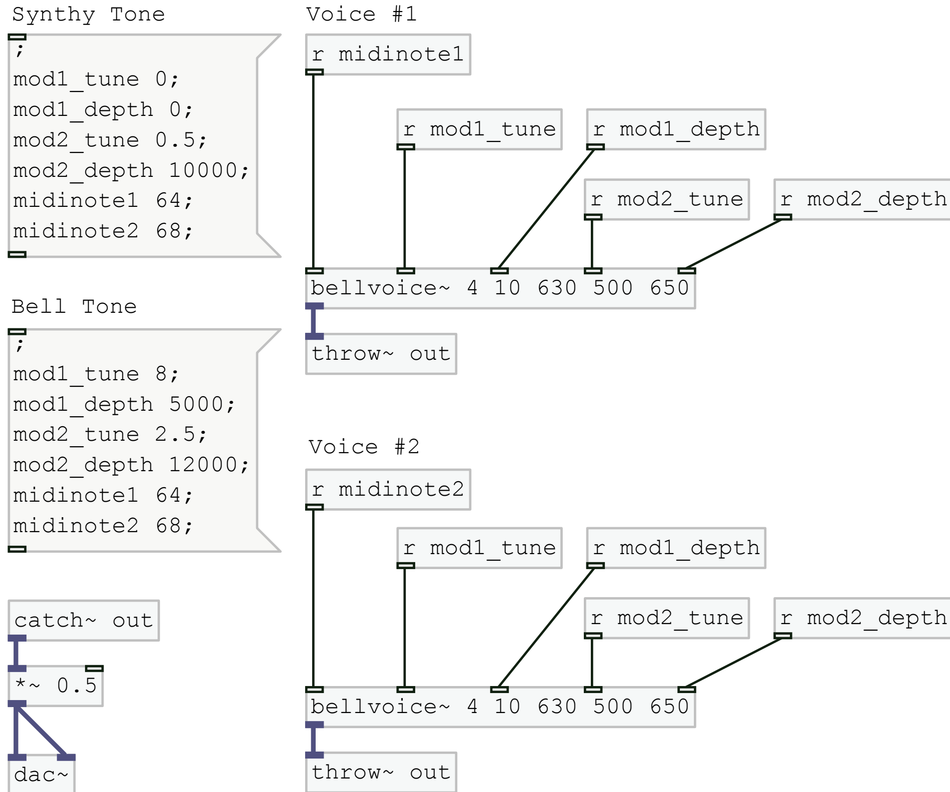
```
inlet~   inlet multiplier   inlet depth
*~
osc~
*~
outlet~
```

The incoming frequency is multiplied by the tuning value, fed into the osc~, and then multiplied by the depth setting, controlling the amplitude, or index of the modulator in FM terms.

Testing the Patch

This is a fairly dense patch, and after all this talk about how FM works, I'm sure you're eager to hear how it sounds. A nice technique, which we've already used in previous patches, is to add some message boxes to send the values to each receive object. In this way we can fine-tune values we expect to be coming from the mobile apps directly in the patch, and then use those values to configure our native code.

• Click HERE to purchase this book now. discuss



The preceding image shows the final top-level patch with two message boxes that hold some settings I've chosen to produce two interesting sounds from the patch we've built. Both use two MIDI notes, 64 and 68, to create a major third, sort of a happy sound.

Two Different Tones

Beyond that, the settings are slightly different. The message box labeled Bell Tone sounds quite like a bell, with a harder initial attack and a metallic quality. Synthy Tone sounds slightly bell-like, but has a more synthesizer-like quality. I experimented with the values and landed on these as the tones I wanted to use in each app—the synthy tone in the Android app and the bell tone in the iOS app.

The Hard-Coded Settings

Let's go over the settings in the message boxes and what they're doing. But first, to understand them we need to understand the non-configurable settings in the arguments to bellvoice~:

- Second operator octave multiplier: 4
- Modulator 1 decay: 10ms
- Modulator 2 decay: 630ms
- Carrier 1 decay: 500ms
- Carrier 2 decay: 650ms

These settings mean the second carrier group is set to be four octaves above the first, adding some depth to the sound. The first modulator affects only the first carrier for a quick burst of 10ms. This is great for making the initial bell attack sound, simulating metal being struck by some object, like a bell-striker.

The second modulator is set to be a little faster to decay than the second carrier, so the sound changes character subtly over time. The second carrier, which is the one four octaves higher than the first, is set to decay 150ms after the first carrier, so that the higher-pitched sound hangs around just a little longer than the lower, which mimics the effect of a bell.

The Bell-Tone Settings

Now have a look at the settings in the message boxes. Notice that the bell sound has a tune and a depth set for the first modulator. It doesn't matter much what the tuning is, and the depth affects the sound only a little because the decay for the first modulator is a very fast 10ms to simulate the attack portion of striking a bell.

The tuning of the second modulator is 2.5, which produces an interesting metallic quality in the sound from the second carrier because it's not an even multiple of the frequency. The depth of this modulator is set rather high as well, to heighten the effect.

The Synth-Tone Settings

For the synth tone, we set the fast-attacking first modulator to 0, which removes the initial percussive sound. We also turn down the second modulator's tuning, so it's still not an even multiple, but it's closer in pitch than the bell tone. Its depth is also slightly lower. These settings were derived mostly by experimenting with the bell settings once I had what I liked there.

Now we have a solid FM synthesizer patch with configurable settings that we can control from our native apps. Let's dive right into seeing how we can accomplish that.

Things to Think About

Play around with some different settings, and explore how FM synthesis works and sounds.

This should be an easy one: Why are each of the `catch~` objects in the top level and submodule patches preceded by a `*~ 0.5`?

What if we wanted to expose more settings to the native apps, such as the decay rates of the carriers? How would we do that?

• Click HERE to purchase this book now. discuss