An Introduction to Sound Filtering

by Jonathan Blow

We're going to talk about the basics of sound filtering — playing with the various frequencies that compose a sound effect — in real time. In the process, we'll take a small step into the world of digital signal processing (DSP). DSP is a topic that intimidates a lot of people. University courses in DSP come with thick textbooks, and many programmers have heard of this thing called the "Fourier Transform" that consumes terrifying amounts of CPU time. Recently, I was shocked to learn just how simple the concepts underlying DSP are, and how sophisticated effects are easily done using small amounts of CPU time.

Motivation

A goal of most (if not all) game developers is to use interactive, realistic sounds in their titles. Many game programmers are familiar with the concept of "spatialization," which is the practice of manipulating sounds to fool listeners into thinking those sounds are coming from somewhere other than the speakers. Vendors of spatialization products would like you to believe that these products will make your game stunningly realistic; the fact is that, though currently available spatialization is a welcome feature, it's only one of many steps that you must take to achieve realistic sound.

Sound reverberates in closed spaces. In wide-open spaces, sound loses energy as it travels; high frequencies fall away faster than low frequencies, changing the character of the sound. In fact, this change is affected by the weather conditions between the source and the listener. When a sound bounces off a sheer rock face, different frequencies are deflected in different ways. When you're strutting down a bilinearly-filtered hallway cradling your BFG-31337, and you see a nice scripted sequence of a monster disemboweling a scientist on the other side of a thick polymer window, the cracking-bone effects should sound as though they're vibrating their way through a solid barrier — very different than the same event unfolding beside you in the open air.

Some game-oriented 3D sound systems, such as QSound or Aureal's A3D, contain some features for distance attenuation, reverberation, and the like; but those features usually aren't very sophisticated, nor do they give the developer enough control. Because I've never heard a sound API that let me say, "This cannon blast is happening on the other side of an echoey ravine, and it's passing through a damp fog bank on the way here, and the weather is very windy, and by the way I'm listening to the sound underwater," I believe that more game developers should learn to do this stuff themselves. (Some of the very latest games do use sophisticated filtering effects. UNREAL uses a filter to create a reverberation effect inside caves, and HALF-LIFE uses a filter to simulate sonic resonance when the player is crawling through air ducts.)

In this article, we'll cover the basic ideas behind achieving environmental effects through the manipulation of

And when we were all fallen to the earth, I heard a voice speaking unto me, and saying in the Hebrew tongue, jon@bolt-action.com, why persecutest thou me? It is hard for thee to kick against the pricks.

frequencies within a sound. This tutorial will be easier if we first accept a couple of ideas about sine waves, without proof. These ideas aren't too hard to swallow; ample references to explain them are given at the end of the article.

Sine Wave Review

Because we're going to talk about sine waves, let's review some of their properties. Sine waves are little curved beasties made by the equation $f(t) = \sin(t)$. The argument t is given in radians; sine waves repeat themselves every 2π radians. We say that this 2π radians is the wave's period of repetition, and we may think of period in the temporal sense: it is the amount of time the wave takes to repeat itself. The sine function can produce values from –1 to 1, so we say that the wave has an amplitude of 1 (Figure 1A).

To amplify a wave, we multiply it by a constant A, so the equation becomes $f(t) = A \sin(t)$. If A is 2, then every point on the wave becomes twice as far away from the 0 line, and now the wave ranges from –2 to 2 (Figure 1B). To shift a sine wave to the side by an amount x, we add x to the time parameter: $f(t) = A \sin(t + x)$. We can change the frequency of the sine wave by accelerating the flow of time as the sine function sees it; to do this we multiply t by a constant: $f(t) = A \sin(kt + x)$ (Figure 1C). When we add sine waves together, the positive and negative values can cancel each other out, just like any other function (Figure 1D).

The first remarkable fact that we'll accept on faith is this: whenever we add two sine waves of the same frequency, the result is a sine wave of that frequency. This is easy to see if the waves are not shifted: $A \sin(kt) + B \sin(kt) = (A + B) \sin(kt)$, a wave with amplitude $(A + B)$. However, the sum is still a sine wave even when the source waves are shifted: $A \sin(kt + \alpha) + B \sin(kt + \beta) = C \sin(kt + \gamma)$ for some C and γ.

Sine waves of differing frequencies, however, do not sum to produce a simple sine wave. $A \sin(k_1 t) + B \sin(k_2 t) =$ Something_More_Complex(t) in general. But this is a good thing, because sine waves themselves don't make for very interesting sounds. It is by throwing them together and making a mess that we develop true character.

FIGURE 1. *A. Sine wave y=sin(t). B. A sine wave with amplitude 2: y=2sin(t). C. A sine wave with a period of half of wave A.: y=sin(2t). D. When the two purple waves combine, destructive interference produces the green wave.*

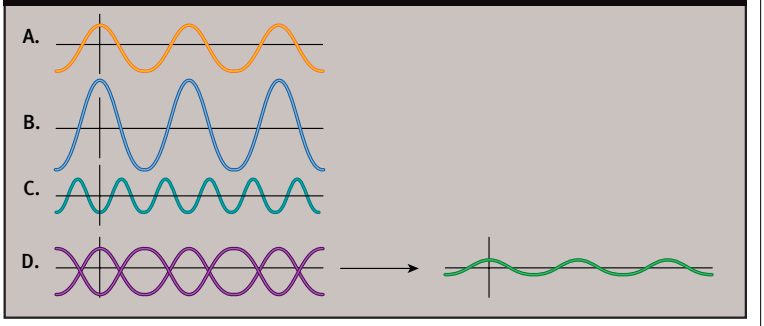
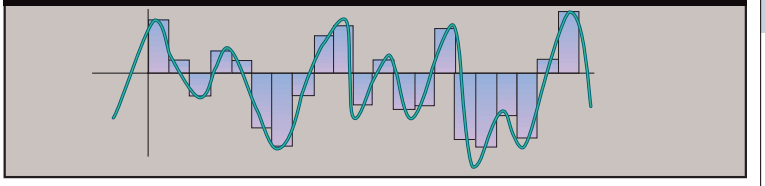


FIGURE 2. *We represent a continuous sound wave with an array of samples.*



Sound Wave Review

These days, we usually represent sounds as arrays of signed 16-bit integers. Each number in the array represents a sample of a sound wave at an instant in time (Figure 2). The sound card converts our samples back into a continuous waveform so that they can be played. We will call this waveform — which is essentially just a function that varies over time — a signal.

The second remarkable fact we'll take on faith is that every signal (over a finite interval) can be represented as a sum of sine waves of differing frequencies. This brilliant realization is called the Fourier Theorem, named after Jean Baptiste Joseph Fourier, who came up with the idea early in the nineteenth century. (There is nothing too magical about sine waves that gives them this capability; the study of wavelets is all about how to decompose signals into varying wave shapes.)

What This Means

Now we are armed with two important facts: all sounds are compositions of sine waves, and two sine waves of the same frequency, added together, produce a new wave of the same frequency. These facts give us the ability to look at sound in an enlightening new way.

Suppose we want to mix two sounds together by adding them; the sounds are represented by functions $f(t)$ and $g(t)$, and we want to add them to produce a new sound $h(t)$: $h(t) = f(t) + g(t)$. The old way of thinking about this is that, for each t, we evaluate $f(t)$ and $g(t)$, add those together, and the result is the value of h at t. According to the DSP way of thinking, we take all the sine waves that make up f, and all those that make up g, then we add them all together to get h.

What happens if we add a sound $f(t)$ to itself? $f(t)$ consists of a bunch of sine waves, $A_n \sin(k_n t + \alpha_n)$. Working in the DSP way, we add all these sine waves to duplicates of themselves, then put them back together to get the result. $f(t) + f(t)$ equals, for each n, $A_n \sin(k_n t + \alpha_n) + A_n \sin(k_n t + \alpha_n) = 2 A_n \sin(k_n t + \alpha_n)$. Every sine wave has twice the amplitude as before. When we put the waves together, it should come as no surprise that the result has twice the magnitude of the original $f(t)$ at every point. $f(t) + f(t) = 2f(t)$, after all. We've made the sound louder. The DSP way of thinking about sounds hasn't bought us any-

FIGURE 3. *A. A sine wave, added to a shifted version of itself, where the shift is small relative to the period of the wave. The wave mostly reinforces itself. B. A sine wave with a shorter period, shifted by the same amount. It ends up destructively interfering with itself, producing zero.*

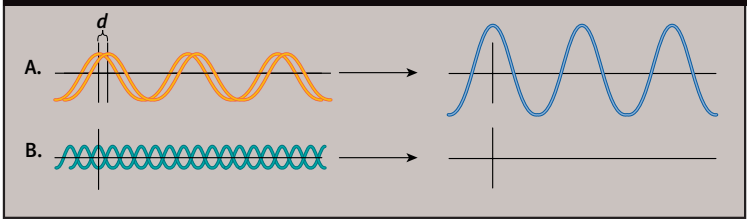
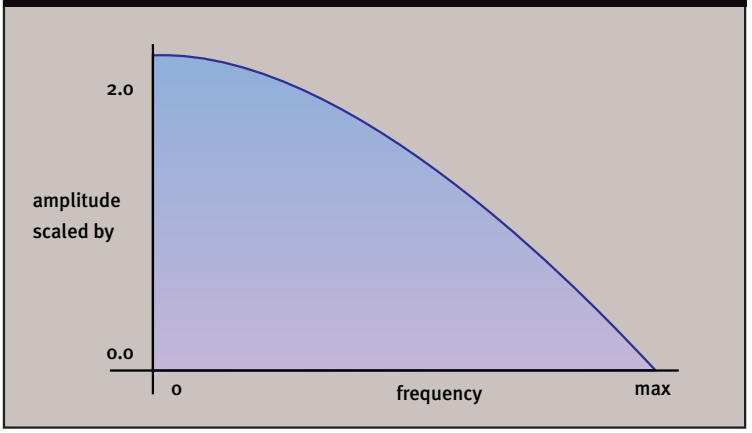


FIGURE 4. *A graph of the frequency response of the low-pass filter in Listing 1. Frequencies are along the x axis; the y axis shows the scale by which each frequency's amplitude will be multiplied.*



thing extra yet — but that was just a way of getting used to this way of thinking and convincing ourselves that it produces consistent results.

Now, here's a revealing thought experiment: what happens when we add the sound f to itself, but before adding, we shift one version of f by a small amount d? That is, if $h(t) = f(t) + f(t + d)$, what does $h(t)$ look like?

Let's look at the individual sine waves in Figure 3. If d is very small compared to the period of a sine wave $A \sin(kt)$, then the wave isn't displaced much with respect to itself. So when we add $A \sin(kt) + A \sin(kt + d)$, the result is very close to $2A \sin(kt)$, as in our last example (Figure 3A).

But what happens when d is longer than the small displacement we just discussed; say, half a period long? We see the result in Figure 3B: $A \sin(kt)$ and $A \sin(kt + d)$ cancel each other out, producing zero.

These cases represent two extremes. The lower the frequency of the wave (the longer its period), the better it will survive the summation process, approaching the ideal of doubling its original value. The higher the frequency (the shorter the period, down to $2d$), the better it will be eliminated by the offset.

It turns out that if d is the width of one sample, the highest frequency contained in the input sound has period $2d$ (so says the Nyquist Theorem, which tells us how much signal information can be stored in a series of samples); this is precisely the frequency that is completely eliminated by the filtering operation. A graph of frequency attenuation is given in Figure 4.

If we take an array of integer sound samples, shift it by one sample, and add it to itself, we are performing a low-pass filter of the sound. If we're representing a sound with 16-bit integers, we want to add the samples using a larger number representation (say, 32-bit integers), then divide them by 2 to ensure that they don't overflow when we pack them back into 16 bits (an issue familiar to anyone who's ever mixed sound). Listing 1 shows code for the low-pass filter that we've just described.

With a simple reversal, we can use this same method to preserve high frequencies and eliminate low frequencies, a process called high-pass filtering. Rather than adding $f(t)$ to itself, what if we subtract it from itself? That is, $h(t) = f(t) - f(t + d)$. This equation is the same as $h(t) = f(t) + (-f(t + d))$; we are negating one of the functions before we add. This has the effect of flipping one of the sine waves about the axis $f(t) = 0$. Now, the low frequencies, because they are hardly shifted at all, negate to cancel themselves out; the high frequencies negate to reinforce themselves (Figure 5).

You can imagine that, by specifying a d that is wider than one sample, one could mute frequencies in the mid-range. For example, if d is three samples wide, we cancel all waves with periods of six samples. But a d of this

LISTING 1. *A subroutine that applies a simple low-pass filter to some samples.*

```
void lowpass_filter(short *samples, int nsamples) {
          static long last_sample = 0;

          int i;
          for (i = 0; i < nsamples; i++) {
                    long current_sample = samples[i];
                    long result = (current_sample + last_sample) / 2;
          last_sample = current_sample;
          samples[i] = result;
          }
}
```

value will also filter waves with a period of two samples (because those waves are shifted by 1.5 times their period, which is the same as shifting them by .5 times their period). This behavior would seem to place some heavy restrictions on the frequencies we'd be able to filter. It turns out, though, that by doing some geometry in the complex plane, we can obtain good control of our frequency response curve, including frequencies that cannot be represented by integer multiples of *d*. See the References section for more information about this. This type of filter is called a feedforward filter because, in producing output, it only uses past values of its input.

Resonance and Feedback

Anyone who's been to an early-90s grunge concert knows what feedback is. This goateed guy holds his guitar too close to the loudspeakers. The guitar pickups (essentially little microphones) catch the loud noise coming out of the speaker and transmit that noise back to the speaker, which tries to make it even louder. Soon, the audio system overloads and we're left with an angry screech.

We can apply this same idea to our audio filter, but in a more restrained and generally soothing way. What happens if we generate our sound $h(t)$ by combining $f(t)$ with previous values of the output, $h(t)$? For example, $h(t) = f(t) + h(t - d)$? For simplicity's sake, we'll measure t in samples, and d will be 1 sample: $h(t) = f(t) + h(t - 1)$.

We can evaluate the first few terms directly: $h(0) = f(0) + h(-1)$. We'll assume that all negative values of h are zero, so $h(0) = f(0)$. That's simple enough. Also, $h(1) = f(1) + h(0) = f(1) + f(0)$. So at the second sample, we're adding f to an offset version of itself, just as before. Now, $h(2) = f(2) + h(1) = f(2) + f(1) + f(0)$. As we repeat this process, we see that the pattern continues forever. We're adding together versions of f that are shifted by successive amounts.

What does this process do? Figure 6 shows the result for waves whose periods are much longer than two samples. As we saw before, adding the wave to a shifted version of itself will cause the wave nearly to double in amplitude. But now, to output the next sample, our feedback loop shifts that result wave again and adds it back to the original. Now, the result is even greater in amplitude — though because it's shifted further, it doesn't reinforce itself quite so strongly. After enough repetitions, the wave will become out of phase with the original version of itself, and the feedback will serve to reduce its amplitude rather than increase it.

We have to be careful with this kind of situation. Figure 6B shows what happens when we feed back a wave with a delay time that is a multiple of its period. The amplitude of the wave climbs toward infinity, which leads to total chaos. For this reason, we must multiply feedback signals by a constant of

FIGURE 5. *A. We perform the kinds of summations that took place in Figure 4, but first we negate the orange waves. The dashed light blue curves represent the position of the waves before negation. Now, the low frequencies, as in A, are killed; the high frequencies, as in B, are doubled.*

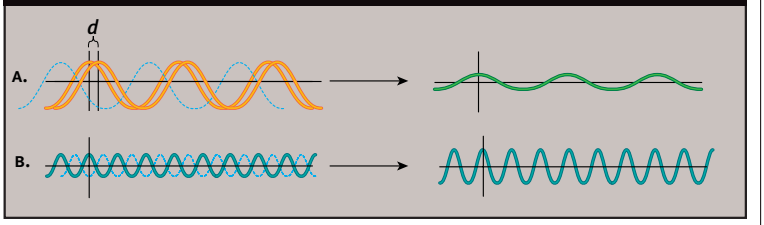


FIGURE 6. *A. The effect of feedback on a wave where* d *is misaligned with the wave's period. As we add it to shifted versions of itself, the result will wax and wane in amplitude, never growing beyond a certain point. B. If* d *is aligned with the wave's period, the wave will continue to grow with each round of feedback.*

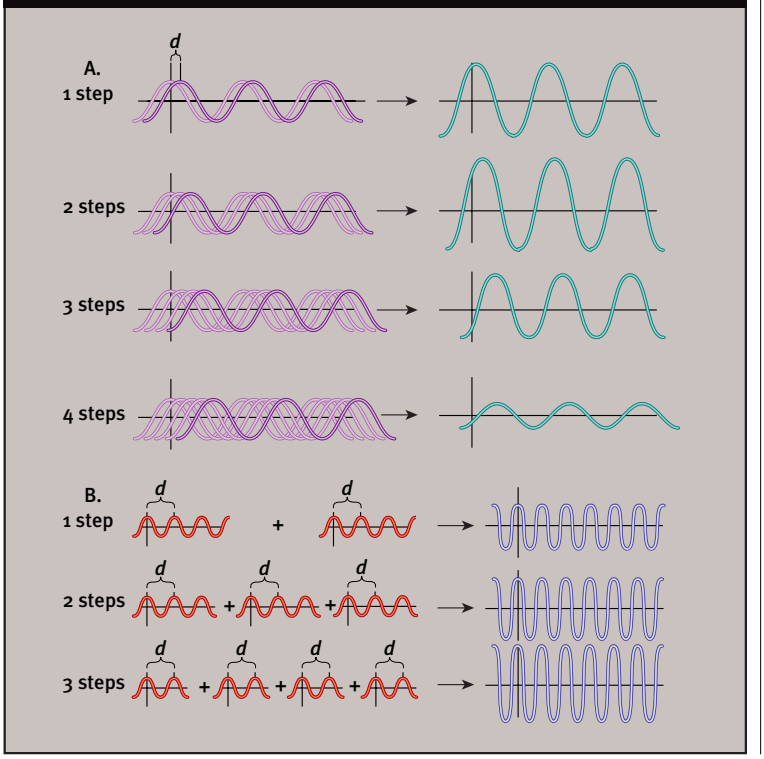
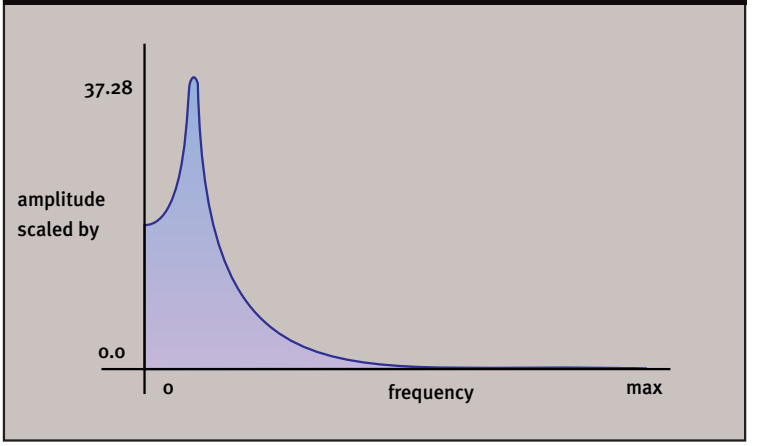


FIGURE 7. *The frequency response from Listing 2.*



magnitude less than 1, to damp them. Thus, $h(t) = f(t) + k\,h(t-1)$, $-1 < k < 1$. Then the feedback filter is stable.

Listing 2 shows code for a simple feedback filter. The frequency response of this filter is shown in Figure 7. Note the relative sharpness of the peak — feedback filters allow us to alter frequency response more dramatically than the feedforward filters we talked about earlier. Note also that it amplifies some frequencies by large amounts. To keep our number representation from overflowing, we compute the maximum amplification when we build the filter; later, as we output samples from the filter, we divide them by that factor to normalize the sound, thus preventing overflow.

As Figure 7 shows, feedback allows us to pick out certain frequencies and amplify them much more than others. It is said that the filter resonates at those frequencies. Interestingly, this is the same effect that occurs with sounds made in enclosed spaces. They resonate according to the shape and dimensions of the cavity, whether it's a yawning cave or a Stradivarius violin. Many physics books describe the reflection of waves between two surfaces, and in such descriptions, one can easily see the mathematical equivalence.

Implementation

In our feedback filter, we ended up multiplying samples by floating-point values to damp them. For most feedforward filters, we want to use floating-point math as well. Generally, we'll convert our 16-bit source samples to floating point and operate on the floating-point samples. For cases in which we're algorithmically generating a sound, instead of reading that sound from a file, our original samples will probably be in floating point.

Interestingly enough, new instruction sets such as AMD's 3DNow and Intel's new Katmai instructions contain stuff that's great for processing sound in floating point. Unfortunately, almost every sound API (3D or not) wants samples fed to it as integers. Therefore, we must convert our floats back to integers, consuming a little extra CPU time and reducing accuracy. These extra calculations are especially irritating because the sound library usually will convert the samples right back to floating point (for example, to scale the volume of the sounds so that they can be mixed into a single channel). Let me offer this plea to the makers of sound APIs: give us an interface that takes samples in floating point!

The example code that accompanies this article will demonstrate the effects discussed here, along with some others, such as pitch shifting and reverberation. (Reverberation works just like the filtering examples we've discussed already; the main difference is that the filter delay d is much longer, to the point where the delay is audible.) The code is available on the *Game Developer* web site.

Now You Try It

We've seen some low-level ways to manipulate the frequency content of sound. We haven't said much about how to put sounds together, though. If an explosion happens inside a sealed vault, you probably want to low-pass filter that sound for listeners on the outside; but exactly what frequency response to use, and how to make a filter to give you that response, could be the subject of an entire book.

We've tried to present sound filtering in an introductory way that is different from the approach taken by most textbooks. That way, upon further reading, you'll be exposed to ideas from a different direction, which can provide fresh insight. We certainly have not been rigorous here. In fact, we've tried to use as few mathematical ideas as possible; some of the References present filters in an elegant way that requires more background. In any case, further reading is required to accomplish a sophisticated understanding of filters. ■

REFERENCES

- Everybody in the world should read *A Digital Signal Processing Primer, with Applications to Digital Audio and Computer Music* (Addison-Wesley, 1996) by Ken Steiglitz. Starting with some simple mathematical exposition, it charges straight through DSP theory in the most accessible way I've seen to date.
- After that, read section 14.10 of *Computer Graphics: Principles and Practice* (Addison-Wesley, 1990) by Foley, van Dam, et al. This section of the book talks about signal processing with respect to graphics, and provides some good visualizations of what it means to filter a complex signal.
- *The Science of Sound* (Addison-Wesley, 1990) by Thomas D. Rossing is a great book that discusses all aspects of sound emission and reflection, with a focus on musical instruments and the human voice.
- *Principles of Digital Audio, 3rd ed.*, (McGraw-Hill, 1995) by Ken C. Pohlmann, contains some good chapters about the Nyquist Theorem, signal-to-noise ratios, and signal dithering (a topic that will be of interest to those who wish to generate sounds algorithmically).