FreeVerb

An algorithmic approach to creating reverb

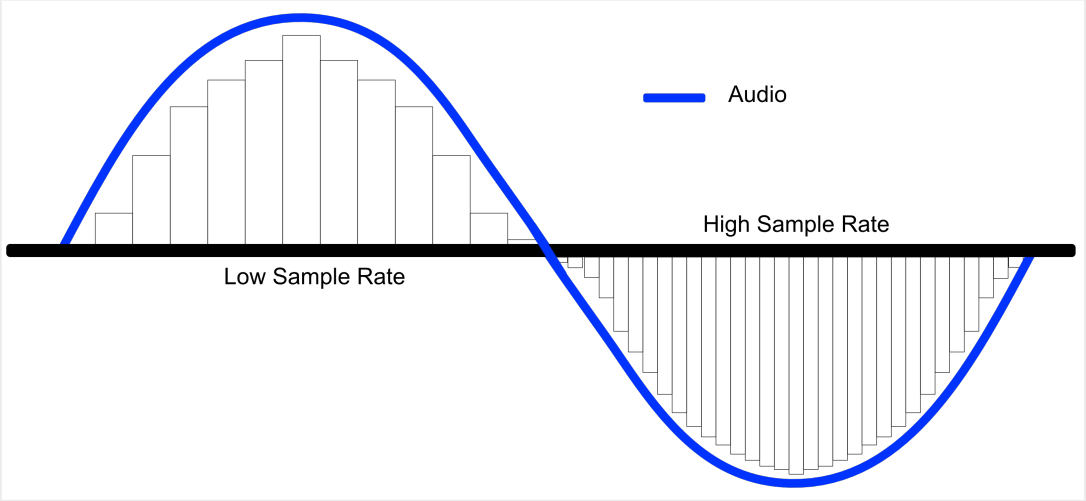
Prerequisites

What is reverb?

Everyone that has ever heard any sound is familiar with reverb, wether they know it or not. Reverb is the natural echo that is produced when sound waves bounce off surfaces. Think of clapping in a large gymnasium or yelling in a parking garage. Below are some examples.

How Do Computers 'Hear'?

Sound occurs as a function of time, *S(t)*, and time is a continuous function. Unfortunately, computers struggle to work with continuous functions. To solve this, we need to convert *S(t)* into a discrete function, one with finite measurements at regular intervals. We do this by taking very small slices, or *samples*, of the sound waves. There are quite a few different sample lengths but the two most common are 44.1 kHz and 48 kHz. This means that the computer is taking 44,100 or 48,000 points of data every second.



Processing Audio

We know that computers take extremely small slices of sound to create a discrete representation, but it would be pretty inefficient to take 1 sample at a time and add the reverb to it. Instead, the samples are stored in an audio buffer, an array of samples. Audio buffers vary in size depending on the work being done and the capability of the CPU. Shorter buffer sizes have less latency but require more from the cpu. To add effects in real time the buffer should be no more than 512 samples, but even that is pushing it. Ideally, the buffer size would be 128 or less.

The computer will fill the audio buffer and send it to the reverb algorithm. The

algorithm will take each sample, do some cool stuff, and return the result to the output buffer.

The Algorithm

First we'll take a look at the entire process, then break it down.

```
void revmodel::processreplace(float *inputL, float *inputR,
  float *outputL, float *outputR, long numsamples, int skip)
{
  float outL,outR,input;
  int i;

  while(numsamples-- > 0)
    {
      outL = outR = 0;
      input = (*inputL + *inputR) * gain;

      // Accumulate comb filters in parallel
      for(i=0; i<numcombs; i++) {
        outL += combL[i].process(input);
        outR += combR[i].process(input);
      }

      // Feed through allpasses in series
      for(i=0; i<numallpasses; i++) {
        outL = allpassL[i].process(outL);
        outR = allpassR[i].process(outR);
      }

      // Calculate output REPLACING anything already there
      *outputL = outL*wet1 + outR*wet2 + *inputL*dry;
      *outputR = outR*wet1 + outL*wet2 + *inputR*dry;

      // Increment sample pointers, allowing for interleave
      // (if any)
      inputL += skip; // For stereo buffers, skip = 2
      inputR += skip;
      outputL += skip;
      outputR += skip;
    }
}
```

BreakDown

If were going to do anything with audio, we need a few things first.

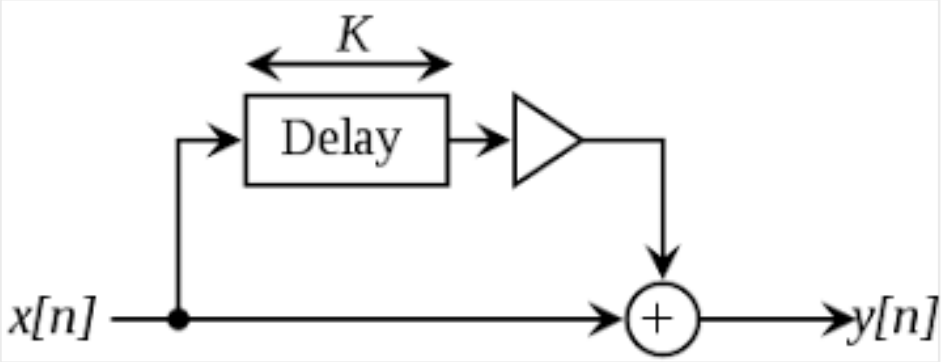
1. Pointer to Left input buffer
2. Pointer to Right input buffer
3. Pointer to Left output
4. Pointer to Right output
5. Number of samples in the buffer
6. How many samples to increment by (1 for mono signal, 2 for stereo)

These are all going to be parameters for the reverb function.

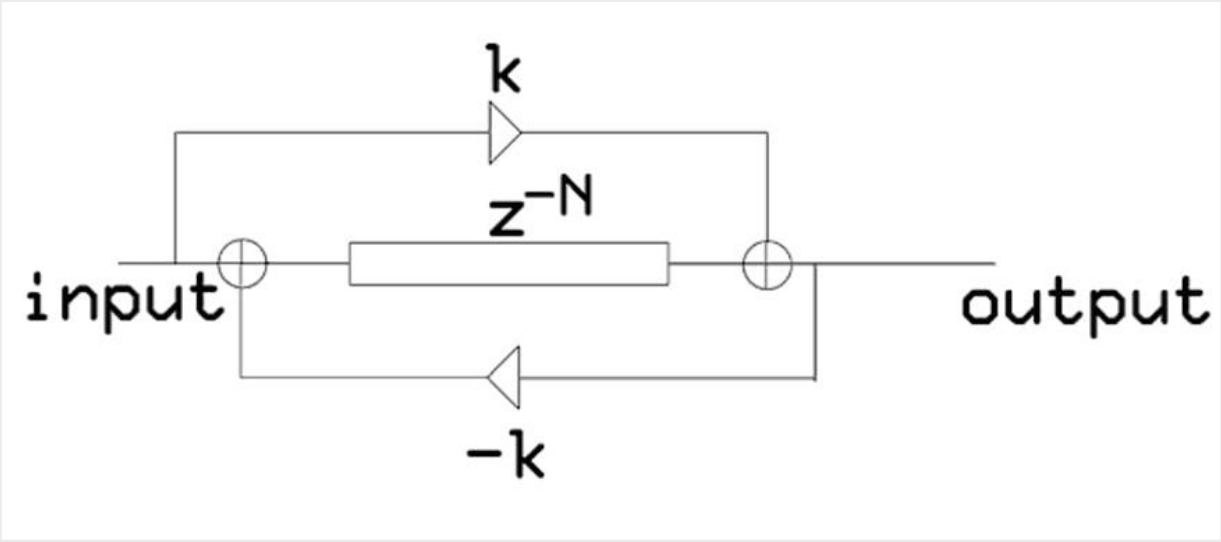
Next, we'll need a loop to go cycle through the samples in the buffer. We need to stop the loop when we've visited all the buffers so we'll set decrement numsamples and terminate the loop when it hits zero.

Set output to zero and lets make a new input that's a sum of the left and right input. Multiply that by the gain to get the correct volume.

And now for the tricky part. In real life, the soundwaves bounce all over the place and interact with with each other, altering the sound in the process. To simulate this we're going to run the sound through several different comb filters at the same time.



Next we need to 'smear' the audio, otherwise it will sound really grainy. The way we do this is by running the sound through allpass filters, one after the other. Allpass filters let the audio through without any modification but it feeds forward and back to introduce phase shifting. A visual representation of an allpass filter can be seen below.



After all the filters are done filtering, we need to give the processed signal to the output, but not JUST the processed signal, we need some of the original signal too or it'll sound weird (example found below). So we add the wet signal with the original signal and place it in the output path.

Finally, we increment the pointers to the next sample to repeat the process until the entire input buffer is reverbed.

Complexity

Because this algorithm utilizes FFTs, its time complexity is O(n*Log(n))