Simple Audio Synthesis on a PC

Christian Oeien

August 27, 2016

Abstract

The purpose of this study is to identify a set of algorithms that synthesize a rich variety of audio. I provide implementations that allow for further experimentation by allowing combination and parameterization. Some measurements are done to investigate efficiency of implementation techniques on a PC, specifically for memory cache utilization.

Several fields of science are involved with sound synthesizing, and I here extract from relevant previous work to conclude with a reduced amount of knowledge needed to implement and configure a few algorithms. Study is performed as required of digital signal processing, the physics of wave phenomena and our hearing ability itself.

1 Introduction

The general motivation was to condense previous theory into working implementation of algorithms under APIs that allow a programmer to compose audio output. I seeked to synthesize waves of sound, using simple computational tools where the origin of each sound sample may be understood by the programmer.

I did not desire to enter into the field of general signal processing, but merely aqcuire the understanding needed to synthesize sound, and apply techniques found in previous work. I should mention the following, however. The Fourier transform [1] enables to appreciate the frequencies present in a signal manifesting along time, just as our ear does. A generalization of this is the z transform [6], where the complex number plane is used to understand and classify filters of signals. This theory is out-of-scope, and I preferred that a user of my API would not need to manage this in order to use the given tools.

The provided API could not be concerned with neither z transforms or complex numbers, but on frequencies, delays and amplitudes.

I do investigate the cues for orientation of a sounds source, as described in [2] which was in the scope of enabling experimentation with our integral auditory sense.

As shown by Karpluss and Strong in [3] it is possible to find algorithms that are both computationally efficient and that can be controlled and tuned without involving any advanced mathematical tools. I exemplify this algorithm

here, because it was the best fit for the tool set I conclude with. This example communicates what I searched for.

The problem of this work was of identifying simple computations and provide an API that enables combination and parameterization. My intent was not to delve deep into the physics and mathematics more than what was needed to understand the computation itself and to arrange for programatic control.

I desired to present techniques using programming languages that are widely used and hardware-efficient, and provide a powerful API model. Using suitable languages I investigated what optimization techniques matters, and how they are related to the PC architecture [4]. When optimizing for data cache utilization the resulting program performs smaller parts of the calculations on an array, that I refer to as *unit*, of several samples at a time, and in turn the next part of the calculation, for a complete unit of values. This also yields smaller active code which may give improved code cache utilization. This latter observation was not studied further for memory size details and effects. That work would be needed to arrive at any conclusion on these techniques.

2 Material and Methods

The strategy I used was to investigate in these broad fields of sciences for algorithms that I could prototype and experiment with.

In agreement with the stated principal problem and scope, I chose to not depend on techniques like Fourier transforms or any calculation using complex numbers. It was difficult to find good material on how to configure digital filters and I skimmed web pages on engineering, to find some calculation notes permitting to adapt a filter with inputs like a desired peak frequency and gain.

I used the GNU C++11 compiler, Python3, on a Linux kernel running 4 CPU cores, whereas I use only one (for simplicity in implementation I do not introduce threads) each as described in /proc/cpuinfo with cpuMHz : 600.000 cache size : 2048 KB TLB size : 1024 4K pages clflush size : 64 alignment : 64 address sizes : 40 bits physical, 48 bits virtual

3 Results

The following presentation of a set of tools and the understanding required to configure them constitutes the result of this research, as well as the produced program designs, implementation and investigation of their runtime leverage of the PC architecture.

The two factors that makes up a sound, are its envelope and its timbre. Envelope is how the loudness of a sound develops over time, from its beginning to the end of the sound. A simple change of envelope can make an impact on how we interpret the sound. A definition of timbre should then be the remaining aspects of the sound, which is what spectrum of frequencies are present in the vibration. Our ears are not providing us information on phases of the waves

except for the orientation cue of left and right ear for appropriate frequencies. Effects may be added that are experienced as extrinsic to the sound.

It was found that non-harmonic sounds like crashes and percussion instruments are hardest to mimic, with the trivial exception of the samples from a pseudo-random function, which resembles wind or cascading water. Synthesizing percussion instruments require some understanding of the wave-guides and modes, which is briefly introduced in [5]. Techniques that combine a periodic function and series of basic mathematical operations arrive at sounds similar, when engineered to arrive at the frequencies desired. It was found difficult to identify one good tool for percussion. The user of the provided API must combine and experiment.

3.1 Algorithms

The Karpluss-Strong sound generator algorithm is responsible for both envelope and timbre, and a sort of resonance works inside of the operated buffer. A function is repeatedly applied to consecutive sound-sample values in a ring-buffer, initially of random-values samples (white noise), so that the signal is smoothed out in time; slow movements remains while fast ones are dampened; the low-pass filter can be provided as parameter to the algorithm. Any periodicity of a multiple of the buffer-length that happened to be present in the initial random signal will be conserved more than other frequencies. Why this happens can be understood by noting that for waves that meet their tail in the ring-buffer at a very different amplitude, do at that point represent a high-frequency change. Thus, like a plucked guitar string, with time after the excitement the sound gets dominated by integer multiple frequencies corresponding to the base period of the vibrating string. These frequencies are called *overtones*.

Another algorithm identified was *frequency modulation* (FM) which is valuable as a tool in that a vast range of timbres are created with relatively simple calculations. I choose to implement real FM, and not the even less computationally intensive algorithm of phase-shift modulation. Like amplitude modulation (multiplication of two signals), non-harmonic series of frequencies are produced. Even tho the mechanics of the production is not as in natural sources, this is experienced as resembling a rich set of sound sources, like hit wood or a metallic bell, depending on variances in the input parameters. The resemblance is due to non-resonant wave-guides and frequency-dependent dampening of the various modes of vibration in such media. What natural phenomenon is resembling to the output almost seems to vary arbitrarily with input controls. I could speculate this is due to how the dynamics of the frequencies present are used by our auditory sense to qualify the sound.

Vocal formants: Study of the spectra in vocal sounds shows that peaks along the frequency domain is what permits us to distinguish an E from an U, or any two vocals from each other. Additive synthesis may be used, and results can be cross-phaded to create diphthongs. We add overtones to a base frequency to also discover that providing a zero factor on even overtones we get air-instrument timbres, as opposed to strings. This stems from physics of the wave origin,

where a pipe open in only one end permits a half-wave, and series of integer waves plus that, hence even multiples of half the pipe-length.

It was found that the Bi-quad filter can be configured for well working behaviors like pass and notch filters. I also provide configuration-helpers for low- and high- pass and shelf filters and a peaking-equilizer. We need comb and echo to add impression of sounds created in a spatial surrounding, which are both implemented by feeding (in the case of echo adding back the feed as well) from a certain delay in a buffer. Chorus may also be considered an effect, where the same timbre at slightly different pitches are added together. In my approach I add parallel rendering of a periodic signal, of a wave provided as input.

3.2 Implementations

I provided three very different implementations of synthesis systems that all include at least the instruments as mentioned above. The first program is more of a prototype, exemplifying different tools in use to compose a few guitar chords filtered with a wah-wah pedal. The second approach is providing a framework with fixed basic concepts like *ADSR* (attack, decay, sustain, release) envelopes, vibrato, tremolo, and providing simple orientation in its stereo output. The third implementation attempts to optimize CPU time by using memory for periodic functions and, more importantly, calculates samples by chunks (referred to as units) at a time at each partial sample-calculation (referred to as generator). This latter technique, conceived with data-cache utilization as motivation also results in smaller amount of code active at a given point in time and therefore also should ensure fast execution of the instructions. I demonstrated that on a composition sufficiently information-dense to be audibly interesting, the time needed on my system doubles when I reduce the unit-size from 100 to 8 samples. This indicates improved utilization of the data cache. However, there is some added complexity to the program when going away from single-sample calculations. Therefore I have no conclusive results when it comes to what approach is more hardware efficient, and it certainly depends on what generators and composition is being rendered. The GNU profiler was used to find bottlenecks, and debugger to appreciate the size of the code I run on my system.

I will now describe my three implementations with reference to the source code. The first is a one-file prototype-like program, `digitarp.cpp`. It leverages Karpluss-Strong and randomized comb filters, and also a Wah-pedal like filter configuration. The two other projects of this work is `mrender` - that includes a MIDI parser, and `libsynth` - the implementation that arranges for optimized cache utilization, are both under `HTTP://github.com/biotty/rmg/sound`. The Python3 wrapper on `libsynth` is named `fuge` and demonstrated in `demo.py`. The MIDI file format was found to not be efficient to contain information on gliding controls, and a direct scripted API more suitable for a composer. `fuge` arranges for the possibility for infinite-time musical composition, by allowing to insert top-level scores while the audio-units are generated.

Three output configurations are made available in `fuge` are mono, stereo by pan and by orientation, and stereo by separate compositions, `binaural`.

I found that it is convenient in the unit-generator API to provide boolean function `more`, in addition to `generate`, and define the invariant that only zero-samples will be produced once `more` is false, and that the latter only transitions to false.

The algorithms are implemented as separate unit-generators that are then combined in various ways by the user of the API. One generator takes as input other generators or envelopes. I use `std::unique_ptr` to link together this hierarchy. A generator produces a chunk of samples, a `unit`, per invocation. Envelopes are used for control parameters, including the period-function of a looped signal.

4 Discussion

Even tho the user of the tools API is generally not concerned withsample rate, he should be aware that the Nyquist theorem applies. Continuous functions are recorded sample-by-sample just as in analog-to-digital conversion. When performing additive synthesis one must dampen amplitude towards the Nyquist frequency (not promptly nullify overtones when over a certain frequency). This matters when providing an envelope for the formants (frequency domain peaks in vocal) in the diphthong tool. For the Bi-quad filter tool it is important to know that this is not a FIR filter like comb, but it is an IIR and resonates at the peak frequency. I provide filter setup functions that allow the API user to glide controls of intuitive input parameters like cut-off frequency. The API does thus not require using any pole-zero model or complex numbers to compose the filter parameters. Power-users knowledgeable on z transforms may compose the Bi-quad parameters directly, also thru the same API.

The filters evaluation of the controls is an example of possible optimization when input-control changes are an order of magnitude slower than sound-wave movements. In those cases it could hold values and calculate only at a certain interval, like a hundred samples. In the case of the filter controls this skipping must take random strides between each evaluation. Otherwise, the step-movement may resonate with the filter, and accumulate unexpected energy.

Tools I found to apply simple concepts but that I omitted are the following. Granular synthesis was omitted because it depends on sampled sounds, and therefore not pure synthesis. Band-noise and chorus may be considered extremes of grain-addition. There are techniques to perform digital (inverse) Fourier transform hardware-efficiently. However, this would make the tool set depend on advanced mathematics, and would best depend on an external library which specializes at this task. Both of these goes against the goal of a broadly understandable, simple and self-contained tool set.

The `mrender` program uses the traditional definition of the term `envelope` and very similar semantics as the common ADSR model. The unit-generator framework `libsynth` uses *envelope* generally to mean any function $f : R \longrightarrow R$ where R is the set of all real numbers. The argument is typically time, in

seconds, and an envelope is used to provide both envelopes and wave shapes. I provide bases for a tabular, punctual and functional envelope, where the API user defines the envelope with the respective kind of data.

In the `libsynth` unit-generator framework, some compromise is called for to arrive at a good utilization of both the code cache and the data cache. Each buffer-unit of samples are calculated, most often involving an envelope. An example illustrating this is that in order to shape a timbre with attack and release to become a sound, as done in the `sound` builder, I could generate a unit of the envelope and then multiply that unit with the unit generated with the timbre. Note that this layout leads to an extra copy of the data to arrive at very small active code, and therefore poor exploitation of the code-cache. The copy also thrashes data-cache for no re-use. I therefore calculate the envelope sample-by-sample multiplying with the timbre unit.

A separate class of objects are created for composing of the generators, the `builder`. The user of the API constructs a `std::unique_ptr` linked hierarchy of instances of this type. There are two advantages of having this be a separate type; their `generate()` would be a no-op wasting call activation records on the stack. Secondly, it is a way of providing for lazy-initiation of generator state, which is specifically important for a score (layout in time of a set of other generators output). The `builder` has no further *raison-d'etre*. A generic `builder` requires C++14 to provide rvalue-reference capture in lambdas in order to capture input-builders held by a builder to be built (recursively initiating their generators ready for operation).

Filters in the unit-generator framework take envelopes as input-parameters and are sample-by-sample processors. Some unit-generators, like the `karpluss` generator takes one filter as input, and `strong` is typically provided. I pursued the idea of providing a generator that applies filters in parallel and then mixes the results together. However, both for Bi-quad and delay-network filters (comb and echo), this invites to unexpected audio results and poor hardware-leverage respectively. The former will result in phase-shifts of the filters nullify the signal at certain frequencies, and this is not controlled by a user of the configuration-helper API. The latter will allocate buffers with redundant data when each filter could have tapped into the passed signal at the desired delays of one and the same buffer, allowing data cache utilization.

Providing stereo output uncovers a general problem with a `std::unique_ptr` based data-structure. The ownership contract locked into the designed types does not hold when arranging for generator output limited and interleaved from left and right ear. In pan or orientational stereo, they both originate from one source. But a generator is owned by `std::unique_ptr`, so it cannot have two owners. Even if I used raw pointers, there would be a problem; the need for two copies of the same audio stream, to be processed differently for right and left ear (some delay and filtering as cues to the listener about sound orientation). My solution was to wrap `std::shared_ptr` in a type of generator and use this for the left and right generator paths. The wrapped shared generator is doing the duplication of the audio. I generalized 2 to N in the implementation. Solving the problem, I came to realize that `std::unique_ptr` expresses something sig-

nificant to a design, and invites by its presence to address the problem, namely the need to duplicate the audio units, that generally have one unique user.

`fuge` allows composing music while generating, to allow for real-time performance or by an infinite artificial-creativity script. Initially the output (master) is set up.

```
left = new sum();
right = new sum();
master = new limiter(
    make_unique<inter>(ug_ptr(left), ug_ptr(right)));
```

Rendering is done by repeatedly calling `master->generate(u)` and queuing the samples to the loudspeaker driver, or piping to a tool like `aplay` on a typical Linux-based operating system. The programmer can interleave this, following a musical algorithm or as interrupted by a live musician, insert more to the composition. Given a builder `b`,

```
std::shared_ptr<generator> g =
    std::make_shared<ncopy>(2, se.b->build());

const double r = p * pi * .5;
left->c(make_unique<wrapshared>(g), std::cos(r));
right->c(make_unique<wrapshared>(g), std::sin(r));
```

In the above example the variable `se` would hold a tree of generator-builders as composed by the program. I here add it onto the stereo output being generated, using a power-conserving mix of $r \in [0, 1]$ right-to-left.

References

[1] Ronald N. Bracewell. The fourier transform. In *Scientific American 6*, pages 86–95, 1989.

[2] David A Burgess. Techniques for low cost spatial audio. In *GVU Technical Report GIT-GVU-92-09*, 1992.

[3] Kevin Karplus and Alex Strong. Digital synthesis of plucked string and drum timbres. In *Computer Music Journal 7(2)*. MIT Press, 1983.

[4] Markus Kowarschik and Christian Weiss. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies LNCS 2625*, pages 1–13. Proceedings of the GI-Dagstuhl Forschungseminar, 2003.

[5] Gordon Reid. Synthesizing percussion. In *Sound on Sound 11*, 2001.

[6] J.O Smith. Pole-zero analysis. In *Introduction to Digital Filters with Audio Applications*, 2007.