Software Musical Synthesizer

Vladimír Arnošt*
arnost@dcse.fee.vutbr.cz

Abstract: This paper describes main methods used during development of a pure software musical synthesizer named *WTSynth*. The program simulates real synthesizers by using Wave-table synthesis. It shows that software can be an inexpensive alternative to special musical equipment.

Key Words: music, synthesis, MIDI, wave-table synthesis, pitch shifting, interpolation, resampling, filtering

1 Introduction

The average computational power of current computers is constantly growing and the number of possible applications is increasing accordingly. Tasks traditionally devoted to specialized hardware can be delegated to custom software running on mainstream processors.

The aim of this project was to create a program that would make it possible to replace expensive hardware musical synthesizers. This solution would give its users a chance to improve the computer music (MIDI) playback quality at a substantially smaller cost than if they bought equivalent piece of hardware. Software is also more flexible than hardware and can be upgraded or completely replaced very easily.

2 Principles of Operation

A software synthesizer tries to work in a similar fashion as its hardware counterpart. Its operation can be described as a pipeline of functional units working together to get expected results. Unlike hardware, a program cannot execute different pieces of code in parallel if only one processor is available. The biggest difference is the inability to parallelize the work, so different parts of the algorithm must be executed sequentially.

Multiple simultaneous voices must be calculated at once to reach required level of *polyphony*. Values typically range from 16 to 32 voices. In hardware, all these voices could be theoretically assigned independent synthesis units and thus work in parallel. Software must process all the voices in sequence and then combine the results into an output stream.

Hardware synthesizers are usually independently working devices. They can spend 100% of their time on synthesis and other tasks are handled by other devices. Software usually cannot do the same, because other programs might be executing in parallel and thus only a fraction of CPU power is available to the synthesizer.

The above mentioned differences outline part of the problems that must be solved in order to create a usable software synthesizer.

* Ing., Department of Computer Science and Engineering, Božetěchova 2, CZ-612 66 Brno

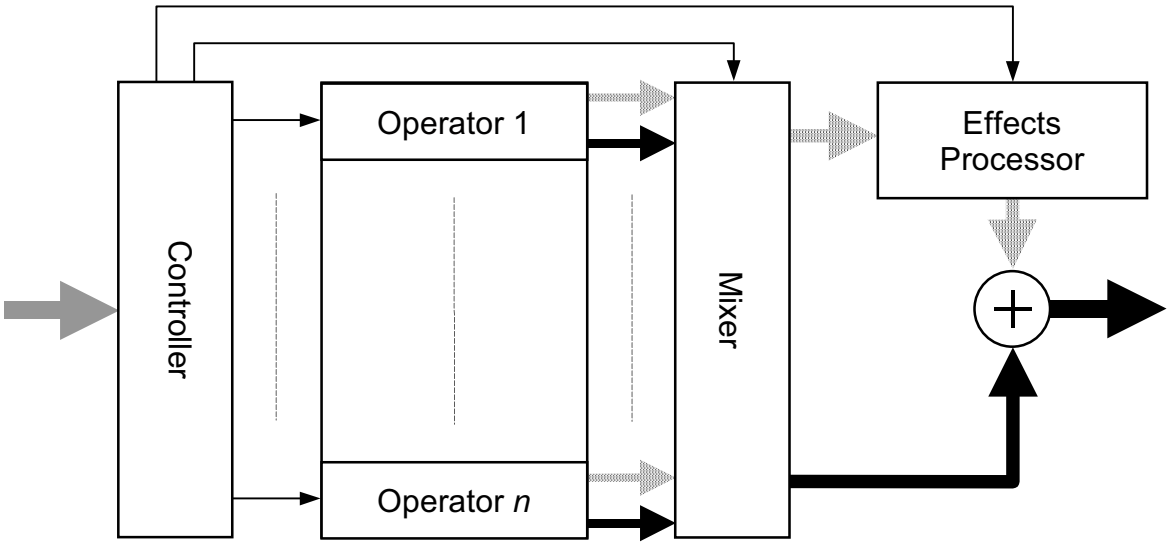

Fig. 1. Schematic diagram of generic synthesizer

2.1 Wave-table Synthesis

Wave-table synthesis (WT synthesis from here on) makes it possible to imitate the sound of real musical instruments with only moderate computational power requirements. The method is based on a table of sound samples recorded from real instruments or artificially calculated. These samples can be played back at various frequencies and the output sound can resemble the original timbre.

Playing back the samples without any modification rarely satisfies any musician so *modulation* was introduced. WT synthesis differs from playing whole samples in the use of modulation to recreate the characteristic timbres from small sound samples and a set of parameters. The most common modulation methods include:

- pitch shifting, resampling
- amplitude envelopes
- filtering
- periodic changes of pitch and/or volume (vibrato, tremolo)
- sound effects (reverberation, chorusing, etc.)

The project was focused mainly on the problem of *pitch shifting*. For real-time operation the algorithms used to accomplish the task must be as simple as possible. On the other hand their quality and precision must be high enough to avoid audible *artifacts* from appearing on the output. These two requirements are obviously contradictory, so a compromise solution must be found.

2.2 Synthesizer Model

The software synthesizer is based on an abstract model shown in Fig. 1. Here, a block called an *operator* is repeated for every voice based on polyphony level (*n*). All the outputs are routed into a signal mixer, where they are combined together. Part of the combined sound is passed through an optional *effects processor*, which may add artificial echo, chorus and other effects. In the end, outputs of the mixer and the effects processor are added together and sent to the output.

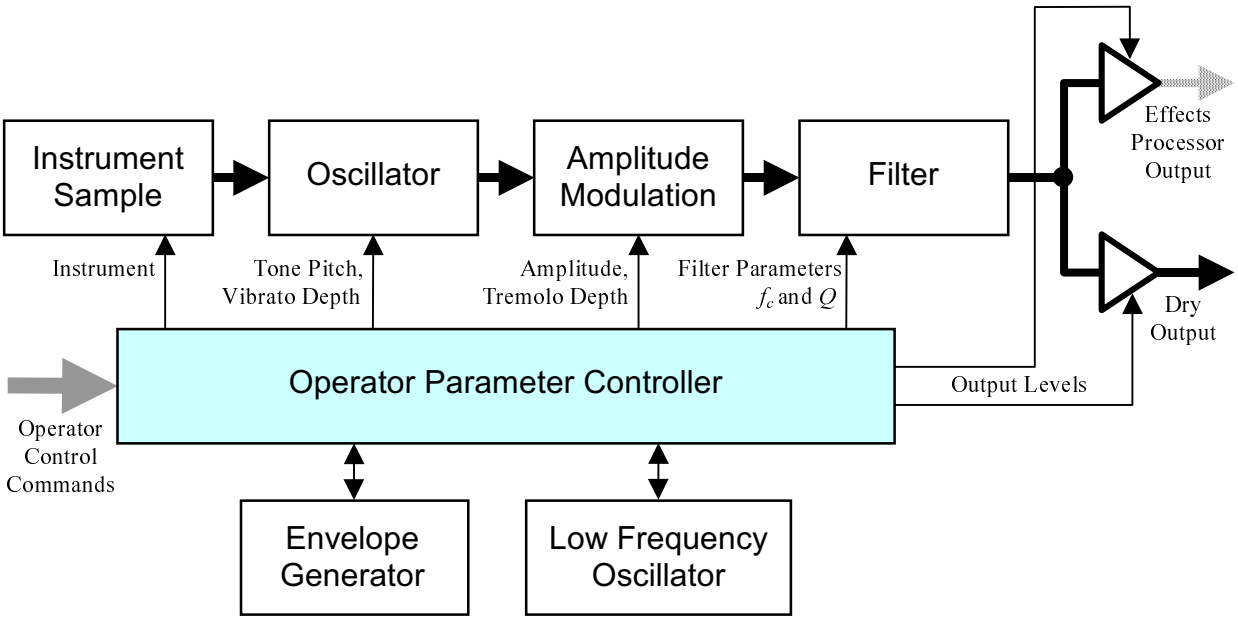

Fig. 2. Schematic diagram of an operator

The whole system is managed by a *controller*. This block is responsible for receiving commands from outside and distributing necessary parameters into the rest of the system. Structure of this block depends both on the format of data coming to its input and also on the implementation of the remaining blocks.

The *mixer* simply adds all its inputs and sends the weighted sums into its outputs.

2.3 Operator Structure

An operator (Fig. 2) is responsible for generating the sound of an instrument. It requires several parameters to be supplied from the outside: sound sample, pitch, volume, envelopes, etc. It can have one to more outputs. The main output conveys 'dry' signal directly into the mixer. The other output can lead to the effects processor. The number of outputs doubles if we consider stereo (binaural) signal processing.

Raw samples (waveform) are taken by an *oscillator*, which performs pitch shifting. Sound samples can be either looping or one-shot. Looping samples have 2 additional parameters: loop start and loop end. These two points mark one period of sound that is continuously repeated until the key is released. Pitch shifting is done by advancing current position by a fractional number and interpolating between adjacent samples. Vibrato effect is accomplished by periodically changing the pitch in a certain interval of values.

Next, oscillator output goes into *amplitude modulation* unit. Volume envelope, the overall voice volume and tremolo effect are applied here. Envelopes simulate properties of natural instruments, whose amplitude quickly rises when the key is pressed and then diminishes as the time passes.

The signal can be filtered to remove unwanted frequencies or noise. This is done by a low-pass *filter*, which is linked to amplitude modulation unit output. The filter can have either fixed or variable cut-off frequency f_c and resonance factor Q. The cut-off frequency can even be periodically changing (wah-wah effect).

3 Resampling

Resampling is a process in which the sample rate of a discrete signal is changed. Let f_{s1} be the original sampling frequency and f_{s2} the new one. If $f_{s2} < f_{s1}$, the signal must be band-limited to $f_{s2}/2$ prior to conversion to satisfy *Nyquist sampling theorem* requirements. This should be done by a brick-wall low-pass FIR filter with cut-off frequency of $f_{s2}/2$. Such a filter is very difficult to realize, though, so the requirements must be a bit relaxed. The real filter cut-off frequency is set below $f_{s2}/2$ and the transition band frequency response is expected to be less steep and a bit rippled.

Unfortunately, it is quite difficult to change the sample rate without introducing additional noise and distortion. The best compromise between low processor demands and high quality seems to be simple *linear interpolation* scheme. Despite its far-from-ideal response, it is considerably better than the simplest possible method *sample and hold*. Other methods (e.g. interpolation by higher order curves, oversampling, etc.) seem to be too time consuming to be used in real-time synthesis.

3.1 Sample and Hold

This is the most straightforward method that can be used to resample a signal. It is also used very often in simple systems. The quality of resulting signal is very low and its usage should be limited only to special cases.

Let's suppose the current sample position p_i is expressed as a real number. Every time a new sample is requested, this position is advanced by a real increment Δp:

$$s_i = sample[\lfloor p_i \rfloor] \tag{1}$$

$$p_{i+1} = p_i + \Delta p \tag{2}$$

Equation (1) shows that output sample s_i is chosen simply by chopping off the fractional part of position p_i. Rounding the position to the nearest integer would not improve the algorithm.

The increment Δp used in equation (2) is expressed as a ratio of the original sampling rate f_{s1} and requested rate f_{s2}:

$$\Delta p = \frac{f_{s1}}{f_{s2}}. \tag{3}$$

For example, if the original sampling rate is f_{s1} = 16 kHz and the new rate is f_{s2} = 20 kHz, the increment would be $\Delta p = 16/20 = 0.8$.

3.2 Linear Interpolation

Linear interpolation means a major improvement over the previous method. Intermediate values not found in supplied samples are calculated to get rid of jagged appearance and audible distortion. These new values are just estimates of the original waveform (before it was sampled and quantized) based on two neighbouring points s_A and s_B. The algorithm assumes linear transition from point A to B, which introduces an error, because the original waveform could have almost any shape between these two points.

$$s_A = sample[\lfloor p_i \rfloor], \quad s_B = sample[\lfloor p_i \rfloor + 1] \tag{4}$$

$$s_i = s_A + (s_B - s_A) \cdot (p_i - \lfloor p_i \rfloor) \tag{5}$$

Expression $(p_i - \lfloor p_i \rfloor)$ is the fractional part of p_i. It is used as a parameter for smoothing the resulting curve.

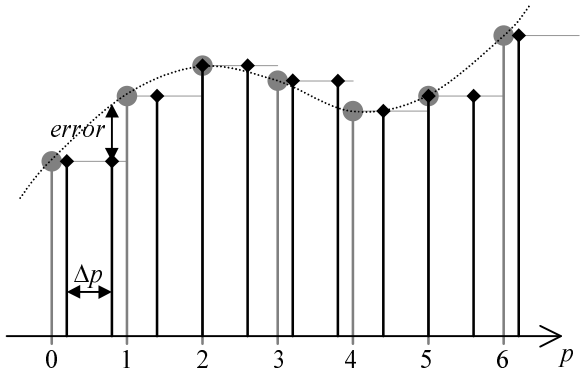

Fig. 3a. Sample and Hold Algorithm

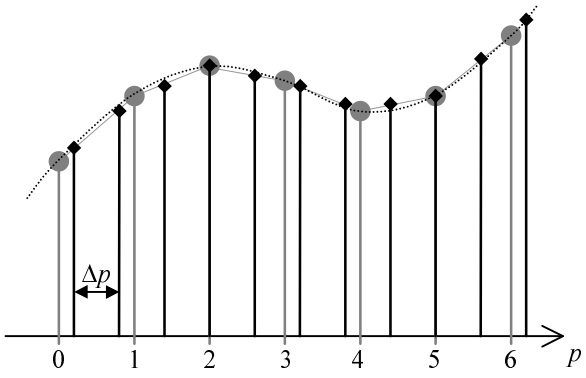

Fig. 3b. Linear Interpolation Algorithm

Fig. 3 shows an example sequence of samples resampled by a) Sample and Hold and b) Linear Interpolation algorithm. The difference between the original curve and new samples (marked as *error*) is many times smaller if linear interpolation is used. The increment Δp is smaller than 1, which is equivalent to $f_{s2} > f_{s1}$ and thus no low-pass filtering is necessary.

4 Program Description

The program *WTSynth* is divided into two distinct parts:

- synthesizer module
- user interface

Currently, only *Windows 95* version exists (initial version was available only for *MS-DOS*). It should be possible to port the program into different environments (like *Unix/X-Window*) by rewriting the user interface code and sound system interface. The synthesizer module code requires a 32-bit processor. All calculations are performed in integers only. Future versions might use floating point operations, because they are performing almost as fast as integer math, but provide more precision.

The program uses musical instrument definition files found on the Internet. These files were originally developed for *Gravis Ultrasound* series of cards, but can be used in many applications. They are called *Patch* files and their filename extension is `.PAT`. Instrument patches can be arranged into *Instrument Banks* for simpler use, e.g. all percussive instruments can be put into a bank called ‘drums’ and then loaded into the synthesizer at once.

Music score is read from *Standard MIDI Files* (SMF). Both single- and multiple-track files are supported. Recent extensions like the *Yamaha XG* format are not fully supported yet. Other formats might be added in the future if needed.

Synthesis is performed in *blocks* of fixed size. Once a full block is calculated, it is sent down to sound device driver queue to be played. This is done because it would be very ineffective to call the driver after every single sound sample is calculated. The block approach has another advantage: different parts of complex algorithm can perform certain operations on the whole block, while the processor executes still the same instructions. These instructions are fetched into the cache in the first iteration and can execute much faster in the following iterations. The buffer size should be chosen according to the processor cache size, so that the most often used parts of the algorithm can fully fit in the cache. Too big blocks might cause cached instructions to be replaced by data. This problem does not apply to processors with split instruction and data caches (e.g. Pentium). It is an issue of second-level caches which are unified (code + data together), but they are several times larger than processor caches.

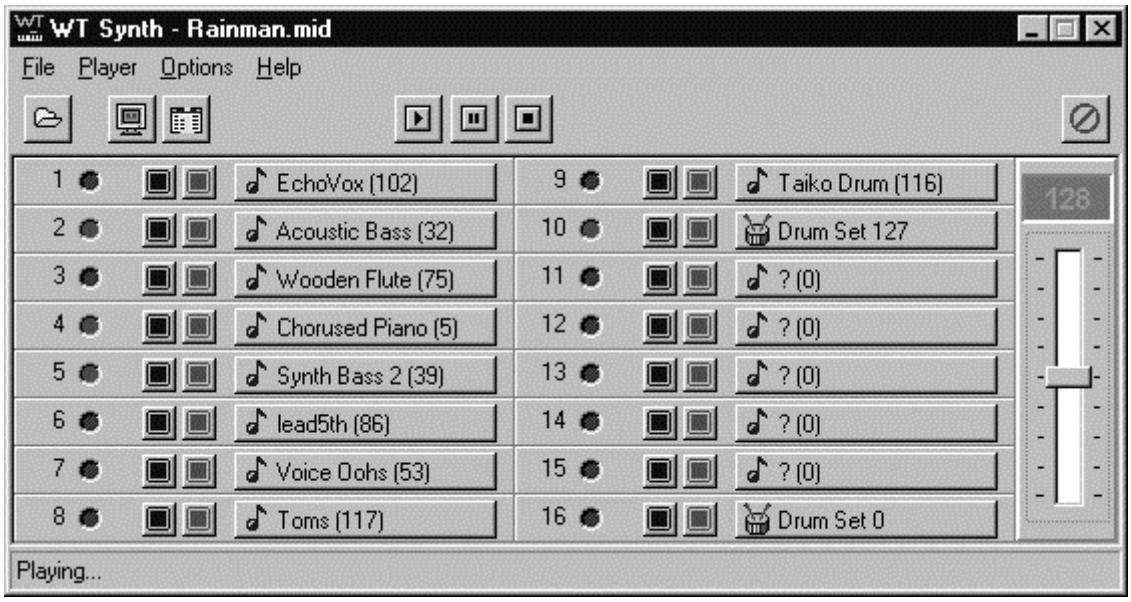

Fig. 4. Synthesizer Control Panel

5 Conclusions

The program *WTSynth* has shown that creating pure software synthesizer software is a perspective way which can lead to very good results. The program still exhibits some problems typical to early research and development phase, but they are being continually solved. It will be necessary to concentrate on the algorithms that form the program core and try to improve them further.

As an average computer power rises, it will be viable to use more sophisticated algorithms for sample interpolation and filtering soon. It seems to be absolutely necessary to have enough experience and working algorithms in order to be prepared for the time when almost any computer sold will be able to bear the load of such software.

References

1. Arnošt, V.: *Software Musical Synthesizer*. [diploma thesis], FEECS Technical University of Brno, 1998 (in Czech)
2. Heckroth, J.: *A Tutorial on MIDI and Wavetable Music Synthesis*. Application Note AN27REV4, Crystal Semiconductor Corporation, Austin, Texas, 1993
3. Rothacher, F. M.: *Sample-Rate Conversion: Algorithms and VLSI Implementation*. [Ph.D. thesis], Swiss Federal Institute of Technology, Zurich, 1995
4. Smith, J. O.: *Bandlimited Interpolation: Introduction and Algorithm*. Stanford University, Stanford, California, 1993