Implementation of Digital Synthesis in Functional Programming

Su Xiaotian, Evan Sitt, Beka Grdzelishvili, Zurab Tsinadze, Xie Zongpu
Hossameldin Abdin, Giorgi Botkoveli, Nicola Cenicj, Tringa Sylaj, Viktória Zsók

{suxiaotian31, sitt.evan, bekagrdzelishvili0, zukatsinadze, szumixie, hossamabdeen17, botko.gio, nicola.cenic, tringasylaj}@gmail.com, zsv@inf.elte.hu

Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
H-1117 Budapest, Pázmány Péter sétány 1/C., Hungary

Introduction

Digital Synthesis is a *digital signal processing* technique for creating musical sounds. In contrast to analog synthesizers, digital synthesis processes discrete bit data to replicate and recreate a continuous waveform. The digital signal processing techniques used are relevant in many disciplines and fields including telecommunications, biomedical engineering, and seismology.

Motivation

We chose to pursue this project in order to extend the possible applications that can be created with the pure lazy functional programming *Clean* language [1]. The language offers very good abstractions tools for sound generation implementation. Digital synthesis is typically accomplished within a *C++* framework, however the nature of synthesis involving multiple tracks and post-processes in parallel lends itself naturally to a functional programming paradigm [4]. While there are disadvantages to processing discrete bit data, such as loss of audio fidelity, digital synthesis offers advantages and synthesis techniques not available to analog systems, such as wavetable lookup synthesis [2].

Wavetable Lookup Synthesis

In implementing the digital synthesis, we utilized a technique called *Wavetable Lookup Synthesis*, in which a certain waveform is stored in a wavetable with constant access, and exploit the relation between frequency and sampling rate to quickly build new waveforms. We used a single cycle (shown in Figure 1) sine wavetable as the basis for our additive and subtractive synthesis, utilizing Fourier series of sine functions to generate all other waveforms.

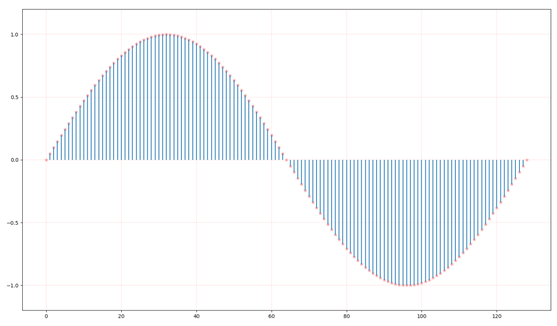

Figure 1: Single Cycle Sine Wavetable

From the simple sine wave, we then do a weighted summation of harmonics to generate more sophisticated waveforms. An example can be seen in Figure 2 where a sawtooth was generated. The sawtooth waveform was then further used with subtractive synthesis to generate the pulse waveform shown in Figure 3.

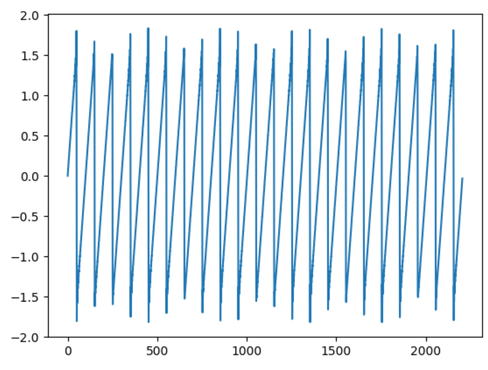

Figure 2: Sawtooth Waveform

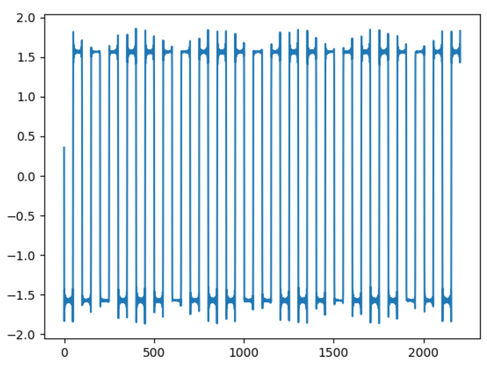

Figure 3: Pulse Waveform

Envelope

After generating the waveforms, we can further post-process the data with effects and filters. The effect we chose to implement was the envelope, starting first with the standard 4-step ADSR envelope, and then later proceeding to a more sophisticated 6-step DAHDSR envelope.
The basic 4-step ADSR envelope is shown below in Figure 4, illustrating the multiplier values. A more sophisticated 6-step DAHDSR envelope applied to a standard sine waveform is shown in Figure 5.

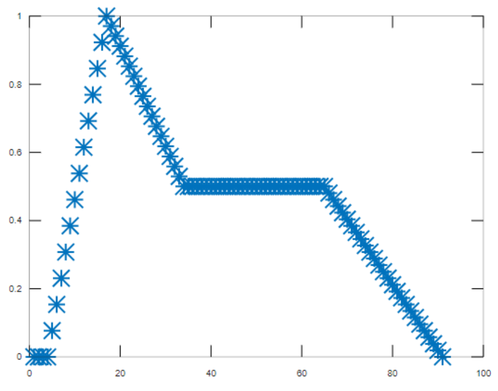

Figure 4: 4-Step ADSR Envelope

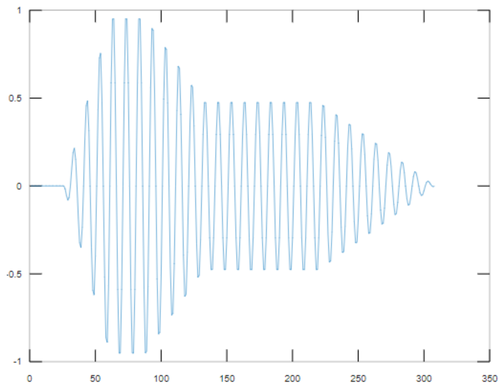

Figure 5: 6-Step DAHDSR Envelope

This code illustrates how list comprehensions are used extensively to render the MIDI event messages into processed waveforms.

```
render :: [NoteChunk] → [Real]
render chunkList = normalized
where
  totalSamples = maxList [numberOfSamples x (x.note.initialTime+x.note.duration) \\x ← chunkList]
  silenceTrack = generateSilence totalSamples
  renderedTrack = [(generateSilence (numberOfSamples x x.note.initialTime))++(renderNoteChunk x)++
                   (generateSilence (totalSamples - (numberOfSamples x (x.note.initialTime +
                   x.note.duration)))) \\ x ← chunkList]
  renderedTrackArr = [listToArr ls \\ ls ← renderedTrack]
  noteSum = [(sum [arr.[ind] \\ arr ← renderedTrackArr]) \\ ind ← [0,1..(totalSamples-1)]]
  normalized = normalizeList noteSum
```

MIDI Input

MIDI is short for Musical Instrument Digital Interface which related audio devices for playing, editing and recording music.MIDI carries event messages, data that specify the instructions for music, including note on, note off events, velocity,etc.

structure / **event type**	status byte	byte2	byte3	byte4
midi events	0x8n - 0xEn	data	(data)	–
sysex events	0xF0 and 0xF7	length	data	–
meta events	0xFF	type	length	data

Table 1: Three types of MIDI Events

The general structure of the MIDI import functions.

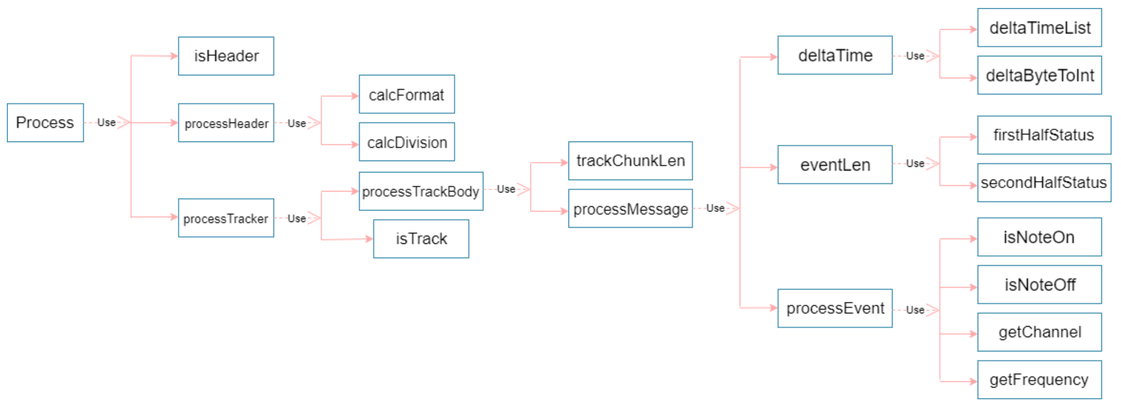

Figure 6: Relations of the functions

This code calculates the cumulative time from the beginning of track to the current event. It utilizes the power of pattern matching in functional programming to handle event messages.

```
findDeltaTime :: Real TrackInfo → Int
findDeltaTime _ [] = 0
findDeltaTime f l
    ♯! {deltaTime,event} = hd l
    ♯! fre = case event of
      NoteOff a b c → b
      _ → -1.0
    |f == fre = deltaTime
    = deltaTime + findDeltaTime f (tl l)
```

WAV Output

The Waveform Audio File Format (WAV) follows the RIFF specification, thus the file consists of a header followed by "subchunks". A typical WAV file has two subchunks, the first one contains file metadata, the second one consists of the sound data in samples, where each sample is split into channels.

chunk descriptor											
52 R	49 I	46 F	46 F	24 chunk size	08	00	00	57 W	41 A	56 V	45 E
fmt subchunk											
66 f	6d m	74 t	20	10 subchunk size	00	00	00	01 format	00	02 channels	00
fmt subchunk											
22	56	00	00	88	58	01	00	04	00	10	00
sample rate				byte rate				block		bits	
data subchunk											
64 d	61 a	74 t	61 a	00	08	00	00	00	00	00	00
subchunk size				sample 1							
data subchunk											
24	17	1e	f3	3c	13	3c	14	...			
sample 2				sample 3							

Table 2: Wave File Specification

Conclusion

The project so far has been successful in meeting the milestones established. We have successfully generated all of the basic waveforms including sine, square, triangle, pulse, sawtooth, and noise. Additionally, we have been able to successfully read event messages from a MIDI file and render out a rendition of the song into a .wav file.

Acknowledgement

This work was supported by the European Union, co-financed by the European Social Fund, grant. no **EFOP-3.6.3-VEKOP-16-2017-00002.**

References

[1] Clean Language Report, https://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf.

[2] Hudak, P., Quick, D.: Haskell School of Music – From Signals to Symphonies, *Cambridge University Press*, 2018.

[3] Thompson, S.: The Haskell: The Craft of Functional Programming, *Addison-Wesley Professional*, 3rd edition, 2011.

[4] Zuurbier, E.: Organ Music in Just Intonation, https://www.ji5.nl/.