Harriet Phillips [redacted] 15th May 2020

COMP2300 Sequencer Part-2 – Harmony of Four Tracks

Overview

Part 2 of Sequencer is an implementation of a harmony comprising of four different tracks using the additive synthesis of triangle waves.

My main goal when choosing to what to implement for part 2 was to make something I would want to show off. This lead to the decision to implement a sequencer that can play multiple notes at one in order to play a song. The choice of four tracks stemmed from the song chosen, the transcript of which I found had an 8-piece harmony [4]. I originally wanted to have every line playing, but couldn't do so with the limited amplitude. Through trial and error, I found the best combination was to have 4 tracks. This resulted in a good emulation of the song that is still audible when only one track is playing.

Each triangle wave that makes up the harmony has an amplitude of 0x1fff. This was chosen to ensure the amplitude of the waves would not exceed 0x7FFF and not go beneath 0x8000 if they happened to peak/trough simultaneously, which would cause distortion to the wave.

I decided to use triangle waves instead of other wave forms was because in doing research I found that triangle waves are a great choice for melody lines [1]. They are also not as buzzy as square or sawtooth waves, which I wanted.

Implementation

Storage of waves and note sequence

The song played by the sequencer is stored in an array of frequency, time * 48000 pairs. To create these arrays, I wrote a small script based on code by Nicolas Bonnet [2] using python that generates them from a midi file [3]. The values required for calculating each wave are also stored in arrays. This makes them easily accessible and modifiable each time a value needs to be accessed, altered and stored. Another advantage of using arrays is that it allows the song to easily by changed by altering the frequency/time arrays.

Main Components:

Song loop and calculations

The song loop is used to check if the sequence is finished by comparing the array counter to the length of the array. Then the values needed to produce the wave for the current note are calculated by calculations.

Calculations loads in the frequency and time for the next note in the sequence. A frequency of 0 means silence, otherwise the number to increase/decrease by to get the correct number of outputs per period for the new frequency. This is calculated by dividing the range of number in the amplitude by the sample by frequency divided by the desired frequency divided by 2.

Song loop and calculations could be joined together into one function, however I found that it made my code neater and more understandable when they were separated.

Sub loop, add loop and blank loop

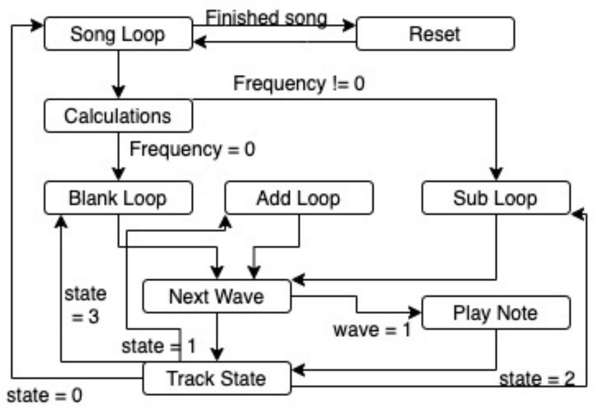
Inside the sub/add loops, the current value in the sequence is added to the total value. The next value in the sequence is then calculated. Checks are done to see if the note has played for the correct amount of time, and if the lower/upper amplitude has been met. In blank loop, all that has to be done is to check if the note has played for the correct amount of time.

While these loops could be combined, using the state of the wave to determine whether to add/subtract/do nothing, it was easier to see what was happening when debugging when they were separate and it also made the code easier to understand.

Next wave and track state

Next wave goes to the track state of the next wave in the sequence, which then handles which loop to go to in order to calculate the correct next value in that wave's sequence. This is determined by looking at the state of the wave.

Control flow

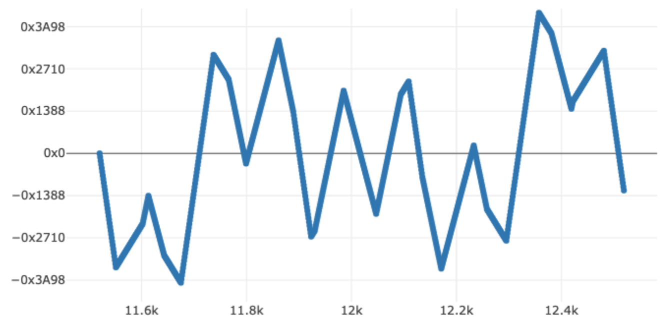


The control flow of the program can be seen in this diagram.

From this it can be seen that the main controllers are next wave and track state. They branch to the loops used to calculate the values in each wave sequence.

Output

Sample plotter:



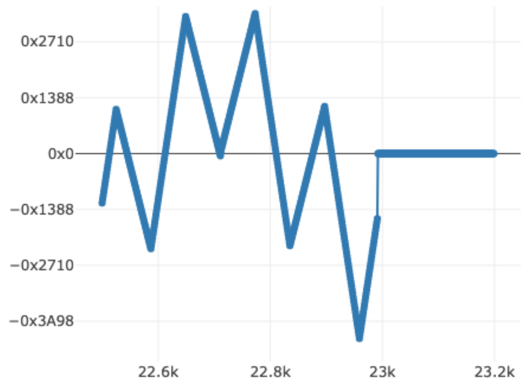
This plot is the plot from sample 11472 – 12472. It shows the additive synthesis of track 2 at a frequency of 98, track 3 at a frequency of 392 and track 4 at a frequency of 147. During this period track 1 is silent.

Reflection

A few improvements could be made to my current implementation.

It could be more efficient with memory usage. The values stored in the arrays are relatively small and so don't require a whole word of space, so if more songs were to be added it would make sense to combine values to decrease memory usage. Some of the values in the wave arrays can also be removed in order to save space. For example, the time length of the wave could be found using the offset and memory address stored in the array. However, as memory was not a big concern here, it was more convenient to have all the values readily available in memory.

In the future, an improvement that I can't yet implement would be to easily change songs by using interrupts. E.g. when the joystick is pressed, the wave arrays are replaced with those of another song.



The biggest difficulty I has was with removing artefacts, and didn't manage to do it. The artefacts are caused by the value of the wave jumping/dropping directly to 0, which can be seen in the picture to the left.

Overall, I would say that my design is a successful and modular implementation of a sequencer able to play a 4-note harmony. I also met my personal goal of wanting to show off my what I implemented. My partner was very excited about the animal crossing song played.

References:

The Difference Between Waveforms and Why It Matters:

[1] https://www.perfectcircuit.com/signal/difference-between-waveforms

[2] Source code for midi parser:

https://github.com/bonnetn/midiparser

[3] Midi Parser:
https://gitlab.cecs.anu.edu.au/u6698981/midi_parser/blob/master/midi_parser.py

[4] Bubblegum KK sheet music: https://musescore.com/user/10340996/scores/2560281/piano-tutorial