The "Make Me Better" App

ECE 420 Final Project Report

Joel Schurgin
Electrical & Computer Engineering
University of Illinois
Champaign, Illinois
joelbs3@illinois.edu

Cameron Obrecht
Electrical & Computer Engineering
University of Illinois
Champaign, Illinois
obrecht3@illinois.edu

***Abstract*—This is a report on the "Make Me Better" app, a music synthesis app created as the final project for ECE 420. Its purpose, functionality, potential improvements to it, and a literature source of some of the principles used in creating this app, are discussed in detail.**

***Index Terms*—TD-PSOLA, envelope, pitch, Moog filter.**

I. Introduction

Our final project is an Android app that allows a user to play an instrument with minimal skill. The up-and-coming musician would play or sing a single note which is captured with the microphone on the Android device. The sound can either be recorded or played back immediately. The note is repitched based on programmed melody which is given by text input from the user.

The user can set a tempo, which is measured in beats per minute. The device will play eighth notes meaning that two notes will play for every beat. For example if the tempo is 60 beats per minute, the device will play 120 notes in that minute. In addition, there are parameters that control the texture of the sound so the notes sound like a synthesizer. Each sound has a rise and fall, called attack and decay, which are adjustable through a single parameter, envelope shape. Changing attack and decay can help achieve softer or more aggressive sounds. In addition, the user can change the filter cutoff and resonance. Even if the user does not know what these controls do, it is engaging to play with the knobs and notice the difference in the sound, and try to attain a desired sound.

II. Research Review

The basic principle of operation of the Make Me Better App is the TD-PSOLA algorithm, coupled with a variable low-pass filter implemented similarly to the design put forth in "New Approaches to Digital Subtractive Synthesis," a review on filters and algorithms for digital music synthesis by Vesa Välimäki et al. of the Helsinki University of Technology.[1] Subtractive synthesis entails the use of oscillators or some electronic source to generate a sound frequency signal, and then using a filter and an ADSR envelope, whether on the signal volume or bandwidth or both, to tune the texture of the sound by "subtracting" energy away. Fig. 1 is a block diagram depicting a "classic example of the subtractive synthesis principle," as implemented by Sequential Circuits' *Prophet 5* synthesizer in 1979.[1] The variable low-pass filter used here in the Make Me Better app is a simplified version of the modified Moog filter described in Välimäki's review, which uses several cascaded low-pass filters to achieve a sharper cutoff than would a singular filter. The modified Moog filter in the review, shown here in Fig. 3, features magnitude coefficients on each low-pass filter in the cascade. The app did not incorporate this, instead using a simple on or off for each filter, and essentially limiting the coefficients to 1 or 0. The app also does away with the sum of generative oscillators "Osc1" and "Osc2" in the *Prophet 5* example in Fig. 1, and instead uses the output of a TD-PSOLA algorithm whose inputs are an acoustic audio signal (such as a human voice or a musical instrument) and a desired pitch to which to correct that signal. The ADSR (attack-decay-sustain-release) envelopes are another feature that was simplified, with the sustain and release not explicitly implemented since the sustain is always a fixed length for the musical notes generated as output by the app. The decay stage serves the purpose of the release if no sustain is used, making the release also unnecessary. The low-pass filter element used in the variable cascade mentioned earlier was implemented in the fashion described in Välimäki's review (shown in Fig. 4), without using Fourier transforms.

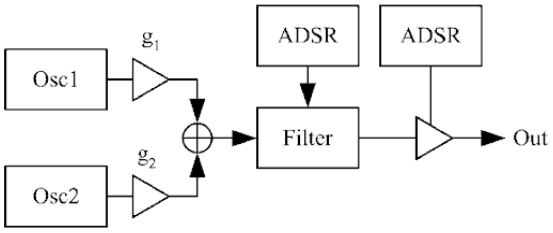

Fig. 1. Block diagram of subtractive synth.

III. Overview of the System

1. Detect voiced/unvoiced signal
2. If voiced, start melodies
3. For each note in the melody: Re-pitch using TD-PSOLA, filter (with cutoff setting proportional to fundamental frequency, to preserve timbre), and apply volume envelope.

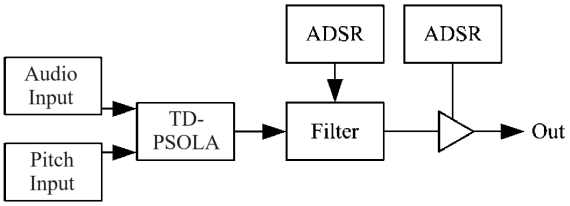

Fig. 2. Block diagram of subtractive synth with TD-PSOLA.

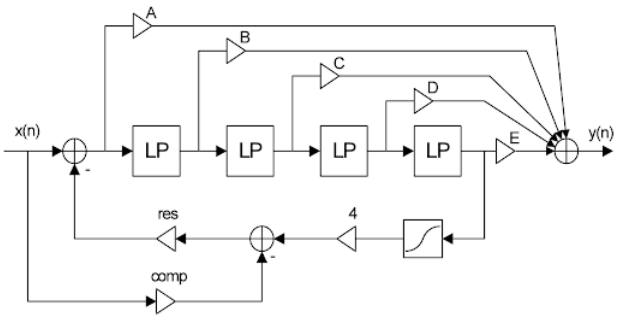

Fig. 3. "Moog" style filter.

IV. Text Input For Melody

The text input will be a string of numbers 1 through 8 which denote the corresponding note in a major scale. So if we are playing the C major scale, the notes in order from 1 to 8 would be C, D, E, F, G, A, B, C. Note 8 is up an octave or is twice the frequency of note 1. It is not important for the user to understand what a major scale is since we will use the starting note that the user plays. If the user plays a C, the melody will be in C major, but if they play a D instead, the melody will be played in D major. If the note is out of tune, its pitch will be snapped to the nearest note.

The choice to limit the note selection guides the user towards creating more listenable melodies with minimal musical knowledge. The user might want the notes to play a rhythm, so we will include the character 'x' to denote a musical rest (silence) to allow for more creative possibilities.

An example input string for "Twinkle Twinkle Little Star" is **1155665x4433221**. The lyrics and the notes line up as shown in Fig. 6.

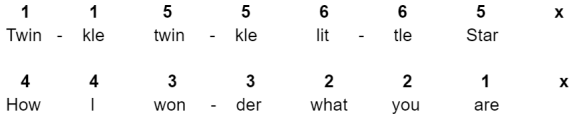

Fig. 6. Twinkle Twink Little Star using custom notation.

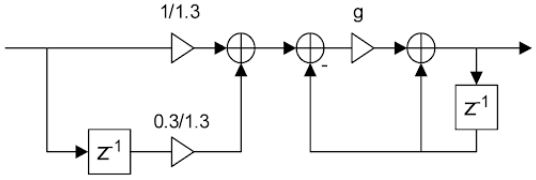

Fig. 4. Single pole filter. (LP block in Fig. 3).

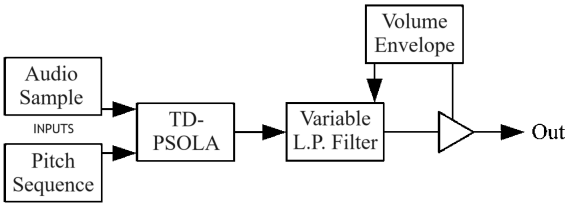

Fig. 5. Block diagram of overall system.

V. Handling Notes

After the text is parsed, an array of "pitch events" is created. A pitch event consists of a frequency and a sample index to indicate when to play that frequency. The indices start at 0 and increase by the number of samples in a note.

The pitch events for each buffer are added to a new array which has the sample index shifted to represent where it should occur in the buffer. The pitch events are used to control the target pitch in TD-PSOLA, the cutoff frequency for the filter, and the start of the volume envelope. The algorithm for handling pitch events is encapsulated in the `PitchEventHandler` class, which updates the pitch event for every sample and outputs the current pitch event for use in the other algorithms.

VI. Re-Pitching

In order to re-pitch the original note, we can use TD-PSOLA and remap the epochs for each note. If there are 3 melodies playing at once and therefore 3 notes playing at once, we can remap the epochs 3 different times at the 3 corresponding pitches.

There are two issues with this approach: the input note might be too long or short for the given melodies, and the note might change volume as it plays. Both problems can be solved by normalizing each impulse in TD-PSOLA. This yields a constant volume, and if the note is too short then TD-PSOLA will repitch noise until the end of the melodies. If the note is too long, we will simply repeat the melody.

VII. Rise and Fall: Envelope Generator

Each note will have a rise and fall in volume, also called attack and decay, and should fit within its given space. In other words, if our tempo is 60 beats per minute, there will be 120 notes played in that minute and each note would be 8.33 milliseconds long. The note should complete its full rise and fall in that 8.33 milliseconds.

Since the attack (A) and decay (D) parameters are relative to the tempo, they will be numbers between 0 and 1 where

$$D = 1 - A. \tag{1}$$

The user will only have access to the attack parameter which is renamed "shape."

$$A_{samples} = A * N \tag{2}$$

$$D_{samples} = D * N \tag{3}$$

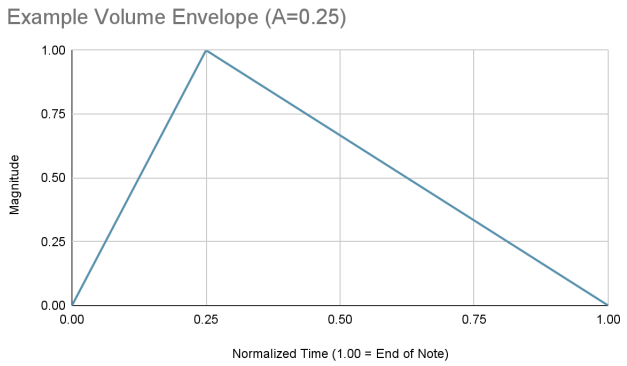

Fig. 7. Block diagram of subtractive synth.

The envelope generator uses a state machine consisting of three states: attack, decay, and off. When a note is triggered, the attack state is set and a float representing the current value of the envelope is set to 0. For each sample, we add the attack increment (the slope of the attack line) until that value reaches 1. Then we go to the decay state and repeat with the decay increment until the value reaches 0. After that, we wait in the off state until a new note is triggered.

VIII. Filter

The filter is an approximation of a Moog filter. The original Moog filter consists of low-pass filters cascaded together, one into the next, and each with its own coefficient of strength in the output. There also is an inverse-tangent feedback component, as well as compensation and resonance parameters. This project only implements the cascade of low-pass filters for the sake of simplicity and the greater preservation of the timbre of the input sample, and additionally allows the user to vary the length of the cascade for greater control over the output sound, since a longer cascade of this type of low-pass filter creates a sharper cutoff. In essence the Moog coefficients are still present, but only allowed to take the values 1 or 0. The app will feature a text box to set the number of low-pass filters in the cascade. The cutoff frequency will be set by a slider as a multiple of the fundamental frequency of the input, rather than as a quantity in hertz, so that the timbre of the original input sound is preserved to the same degree no matter the pitch. Fig. 3 above shows the block diagram.

The implementation of each filter element is such that no Fourier transformation is necessary. The design is adapted from Välimäki et al.[1] Fig. 4 above shows the block diagram. Variation of the cutoff frequency is achieved by adjusting the g parameter, where

$$g = 1 - e^{\frac{-2\pi f_c}{f_s}} \tag{4}$$

and the output of the filter is represented by

$$y[n] = g(\frac{x[n]}{1.3} + \frac{0.3x[n-1]}{1.3} - y[n-1]) + y[n-1] \tag{5}$$

IX. Software Documentation

The software used to create the Make Me Better app was originally based on Lab 5 done in class, which explored pitch detection. Attached at the end of this report is the documentation, in the form of a formatted list of of all of the new classes and data structures declared in the software, whether in C++ or Java, and their component functions and variables.

X. Results

The problem we tackled first was converting the text to an array of pitch events. While it was straightforward to convert the text input to the full array of pitch events, since there is a one-to-one mapping, the main challenge was selecting the right pitch event for each buffer. This was verified by writing "Twinkle Twinkle Little Star" (but just the first half **1155665x**). We used the debugger to verify that the events were parsed correctly.

The next challenge was modifying TD-PSOLA to change pitch part way through a buffer. We used the `PitchEventHandler` to detect when there is a pitch as each epoch is processed. If there is a new event, we recalculate the target period (P1) in accordance with the new frequency. We started with inputting one note to check that the pitch events set the pitch properly, then we added two notes. One problem we encountered when adding the second note is that it would not play. This occurred because we were checking if the melody was done playing by keeping track of our current sample index and checking if that value was greater than the position of the last pitch event. Therefore, the melody was actually playing everything except the last pitch event, so we added one extra event at the end.

To test the full melody, we listened to the output to check if we heard "Twinkle Twinkle Little Star." Initially, we were unable to hear the repeated notes such as the 11 or 55 in the melody. They sounded like one long note. In order to test, we added an extra pitch event for every note at 0.8 times the note length with a frequency of 0 Hz to signify that the note should be off. This added space between the notes but sounded abrupt as each note stopped and started.

To reduce the abruptness, we implemented the envelope generator. Our first attempt was to generate a float array that represents the envelope for one note. We struggled to sync it with the pitch events, so we switched to the algorithm described above that uses a single value and outputs a float array that is the length of the buffer instead. For this, we screen recorded the app and viewed the wave form.

Implementing the low-pass filter resulted in no problems. We programmed it according to the review by Välimäki et al.,

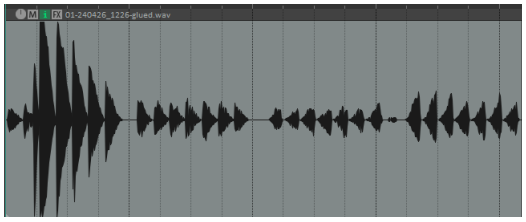

Fig. 8. First half is a fast attack and the second half is a fast decay.

and checked by ear. To know if the filter was moving with the note frequencies, we listened for variation in the low and high notes. If the cutoff was fixed, we would have heard the high notes as significantly quieter than the low notes. We also screen recorded and checked with a spectrogram to verify that the filter was sweeping up and down according to the envelope.

We attempted to implement multiple melodies. Each part of the system had its own pitch event handler, so we encountered synchronization issues between the melodies. We ran out of time to fix this.

XI. Suggestions for Extensions and Modifications

In hindsight, we could have generated full notes as soon as the pitch event occurs. Then we could write the samples into the buffer only generating a new note when we need to. The key difference is we are only calculating the samples needed for the current buffer. This might solve the sync issues when playing multiple melodies, but would require a major restructuring of the code.

One shortcoming of our current volume envelope is that every note has the same length. For instance, the words “star” and “are” in “Twinkle Twinkle Little Star” are actually twice the length of the preceding words, but it would be cut short. To solve this problem, we would add a new character “-” to the text input to signify that we want to continue playing the previous note for an additional note time period. The old “Twinkle” input would be **1155665x4433221**, while the new one would be **1155665-4433221-**. If the preceding character is an “x” such as with the input **1x-**, we would play it as if the input was **1x1**.

Two more features to add are distortion and reverb. Adding more options for tone of the output allows for the non-musician to sound more professional. Distortion would be added before the filter that is applied to each note, while reverb would be added at the end of the processing chain.

Another improvement to the app would be to add a feature that saves the input sound. The user could pluck a guitar string, and instead of hearing the pitch-corrected melody or harmonies played from the device while the guitar string is still vibrating, the plucked string sound could be recorded and saved as a sample, and later played back in its pitch-corrected form. This removes the real-time element from the system, but it may be more practical for budding musicians to pre-record the singular note from their instrument, input the melody, set the filter, and adjust the attack and decay parameters to their liking, and then press the button on the screen to start playing the melody or harmony synthesized from the instrument.

References

[1] Huovilainen, A., & Välimäki, V. (2005, September). New approaches to digital subtractive synthesis.

Software Documentation

The software used to create the Make Me Better app was originally based on Lab 5 done in class, which explored pitch detection. Below is a list of all of the new classes and data structures declared in the software, whether in C++ or Java, and their functions and variables.

void ece420ProcessFrame(sample_buf *dataBuf)
If record mode is OFF, synthesize using incoming data stream. If record mode is ON, either record the incoming signal, or output all 0's until the play button is hit. If playing, synthesize using recorded sound.

void synthesize(const float* inputData, float *outputData)
Implements block diagram in Figure 5.

struct PitchEvent: Represents each musical note that will be played as output.
unsigned long position
float frequency

class TextParser: Converts text input into PitchEvents.
TextParser(int _bufferSize, int _sampleRate)
Constructor

~TextParser()
Destructor

void parse(std::string input)
Converts text input into std::vector<int> where -1 represents a rest and 0 - 12 represents how many semitones above the tonic to play.

void calcPitchEvents(float userFreq)
Converts the above array of ints to PitchEvents based on the userFreq.

std::vector<PitchEvents> getPitchEventsForNextBuffer()
Gets any pitch events for the next buffer. Will return an empty vector if no events are occurring.

void setTempo(float _tempo)
Sets new tempo and calls calcPitchEvents

bool melodyDone()

Returns true if the melody is done playing. Also restarts the melody if it's done.

double getSamplesPerNote()
Returns number of samples per note.

int getNearestNote(float freq) const
Returns the nearest note number in order to set the key of the melody.

class Tuner: Implements TD-PSOLA

Tuner(int _frameSize, int _sampleRate, int maxNumMelodies)
Constructor

~Tuner()
Destructor

void writeInputSamples(const float *data)
Shifts current buffer into bufferIn.

int detectBufferPeriod()
Uses autocorrelation to find buffer period.

void processBlock(std::vector<std::vector<float>>& data, std::vector<std::vector<PitchEvent>> pitchEventsList, int periodLen)
Applies a high pass filter set at the detected frequency, then re-pitches the input and shifts bufferOut.

void pitchShift(std::vector<PitchEvent> pitchEvents, int periodLen, int melodyIdx)
Implements TD-PSOLA with modifications to normalize the impulses and change the target period based on pitchEvents.

void findEpochLocations(std::vector<int> &epochLocations, float *buffer, int periodLen)
Finds epochs for use with TD-PSOLA

void overlapAddArray(float *dest, float *src, int startIdx, int len)
Adds len samples from src to dest beginning at startIdx.

class Recorder: Records input note and plays it back

Recorder(int _sampleRate, int _frameSize, double maxRecordLength)
Constructor

~Recorder()
Destructor

void start()
Clears previous recording and starts new one

void stop()
Used to stop both recording and playing, calls applyFades if recording

void play()
Play back recording

bool isRecording()
Returns true if recording

bool isPlaying()
Returns true if playing back

void writeData(float *data)
Records current buffer

float* getNextBuffer()
Returns next buffer from recording

void applyFades()
Adds fade in and fade out to the recording

class PitchEventHandler: Helper class for detecting new pitch events during processing

PitchEventHandler()
Constructor

~PitchEventHandler()
Destructor

void prepareForNextBuffer()
Sets an internal index variable to 0. This variable is used to keep track of the last event that was searched since events are sorted.

bool setCurrPitchEvent(const int bufferPos, const std::vector<PitchEvent>& events)
Returns true if event was changed

PitchEvent getCurrPitchEvent()
Returns current PitchEvent

class SinglePoleLPF:
SinglePoleLPF(double _sampleRate)
Constructor

~SinglePoleLPF()
Destructor

void reset()
Sets prevInputSample and prevOutputSample to 0.

float calcG(float cutoff)
Finds g parameter to set filter cutoff and returns it.

void setG(float newG)
Useful for setting multiple filters at the same cutoff.

float processSample(float sample)
Returns the filtered sample.

class SinglePoleHPF: Inherits from SinglePoleLPF
SinglePoleHPF(double _sampleRate)
Constructor

float processSample(float sample)
Returns the filtered sample.

class Filter: Combines multiple SinglePoleLPF filters to create a more robust lowpass filter
Filter(double _sampleRate, int _bufferSize, int nCascades, double _res = 0.707, double _maxFrequencyInOctaves = 2.0)
Constructor

~Filter()

Destructor

void processBlock(float *data, const float *envelope, std::vector<PitchEvent> pitchEvents)
Processes incoming buffer and sets filter cutoffs based on current note and envelope.

void reset()
Resets SinglePoleLPFs

void setMaxFrequencyInOctaves(double newFreq)
Sets parameter used in FrequencyCurve.

double FrequencyCurve(double x, double freq) const
Applied to envelope so frequencies sweep logarithmically and are scaled appropriately.

class EnvelopeGenerator: Creates the volume/frequency envelope that is multiplied with each musical note to produce output signal.

EnvelopeGenerator(int _blockSize)
Constructor

~EnvelopeGenerator()
Destructor

enum State { *ATTACK*, *DECAY*, *OFF* };

const float* getNextBlock(std::vector<PitchEvent> pitchEvents)
Finds the next pitch event and its samples, to maintain continuity of the signal.

void setShape(double newShape, double samplesPerNote)
Sets the shape of the volume/frequency envelope according to the tempo and envelope slider position in the app.

void setSamplesPerNote(double samplesPerNote)
Sets the shape of the envelope using the number of samples per note, based on the musical tempo.

void startNote()
Sets state to ATTACK

float getNextSample()

Implements a state machine in order to calculate the next sample in the envelope.