Synthesizer Project #1

CS 245, Spring 2018

Due Thursday, February 1

You will create a rudimentary real–time synthesizer that plays notes in response to MIDI messages. This is in preparation for the final project, a functional wave–table synthesizer.

Install PortAudio

Yes, yet another software API. For audio output, we will use PortAudio — a cross–platform API for audio input and output. You can install and use the pre–compiled binaries available on the course web page. Alternatively, you can download the API directly from the website

`http://www.portaudio.com`

In either case, you should compile and run the program `audio_out_pa.cpp` to make sure that you have installed PortAudio correctly.

Project details

You are to implement a class named `SimpleSynth`, whose public portion is given below (the private portion will be left to you)

```
class SimpleSynth : private MidiIn {
  public:
    SimpleSynth(int devno, int R);
    ~SimpleSynth(void);
    float operator()(void);
};
```

(the header file `MidiIn.h` has been included).

`SimpleSynth(devno,R)` — (constructor) creates a simple synthesizer object associated with the MIDI device with number `devno`. Audio samples are generated by the synthesizer at the specified sampling rate R.

`~SimpleSynth()` — (destructor) destroys the simple synthesizer object.

`operator()()` — returns the next output sample of the synthesizer.

I will provide you with a driver for the synthesizer `SimpleSynthDriver.cpp`, which will use PortAudio to send the audio data that your synthesizer generates to the audio card. You are, of course, responsible for generating reasonable audio data. Here are the requirements.

- The synthesizer voice should be a simple waveform. For example, a sawtooth wave or a square wave. A sawtooth wave can be generated using the code.

  ```
  float sawtooth(float t) {
    t -= std::floor(t);
    return 2*t-1;
  }
  ```

  The frequency of this function is 1 Hz (assuming that t is measured in seconds), so that to sound a frequency f, you will need to speed up by a factor of f. In general, if W is a waveform with frequency 1 Hz, then $y(t) = W(ft)$ gives a signal with frequency f.

- Notes must be played at the correct pitch. E.g., if the MIDI device sends the message to turn on note 60, the note that sounds should be middle C (approximately 261.63 Hz).

- A note must start playing when a note–on message is received, and stop playing when a note–off message is received (this is not typical synthesizer behavior, where a note will have a "decay" after the key is released — we will address this issue later).

- The pitch wheel should be implemented. A full range of the pitch wheel should smoothly change the pitch from -200 cents to 200 cents. There should be no audible artifacts as the pitch wheel is moved.

- Notes played should be velocity sensitive: the volume of the note that sounds should be related in a reasonable way to the note velocity value in the corresponding MIDI message.

- Your synthesizer can be monotonic (only a single note at a time can be played). However, you need to handle "note stealing" in a reasonable manner. For example, suppose that a note–on message for note 50 has been received, so you would respond by playing a sound for that note. Suppose further that after this event, a note–on message for note 55 is received, even though a note–off message for note 50 has not yet been received. You can then "steal" from note 50 by playing a sound for note 55. Since note 55 is playing, you should not stop playing this note until a note–of message for note 55 is received — even if note message for note 50 is received before this occurs. Note that note 50 is "stolen" from permanently: if a note–off message for note 55 is received before a note–off message for note 50 is received, no notes would be played.

  For bonus points, you can try your hand at polyphony (multiple notes can be played simultaneously). However, you must limit the polyphony to a maximum number of notes (between 2 and 10) that can be played at one time. This also requires some form of note stealing heuristic. Audible clipping should not occur when the maximum number of notes are played simultaneously.

Your submission for this assignment should consist of two files: (1) the header file `SimpleSynth.h`, and (2) the source file `SimpleSynth.cpp`. You are allowed to include the header files `SimpleSynth.h` and `MidiIn.h`, as well as any *standard* C++ header file.