21M.385 Lecture Notes

Week 4

MIDI

- See Attached MIDI slides
- Demonstration of FluidSynth – a free General MIDI synthesizer.
    - See common/fluidsynth.py and common/synth.py to see the synth API:
        - After creating Synth(), you can set the a program `Synth.program(self, ch, bank, preset).` Bank is usually set to 0, but can be set to 128 for percussion sounds. See FluidR3_GM_programs.txt for bank/patch information
        - The main functions to use on Synth() are:
            - `noteon()`
            - `noteoff()`
            - `cc()`
            - `pitch_bend()`

Wavetable Synthesizer

- The standard wavetable synthesizer is a fancier version of the `NoteGenerator`, `WaveBuffer`, `Envelope`, `SpeedModulator`, and `Mixer` components we have built so far.
- A Sample generator creates looping audio waveform (could be a sine wave or a looping wave buffer)
- Frequency is controlled using speed modulation
- An Envelope generator shapes the note profile, most commonly using ADSR (Attack, Decay, Sustain Release)
- A Mixer combines all sounds together.
- Additional components of a synthesizer are typically:
    - LFO modulators of parameters (to do pitch bend, or tremolo)
    - Filters
    - Wave table key mappings and velocity mappings
    - Audio FX: Resonant Filters, Chorus, Flange, etc…

Clock and Tempo

- `Clock` is a simple time keeping class. It can be paused / unpaused.
- Tempo is a mapping of time vs ticks. It can be graphed with time on the X-axis and ticks on the Y-axis.
- Ticks are musical units of duration: 480 ticks = 1 quarter note.
- We will sometimes use quarter note and beat interchangeably.
- `SimpleTempoMap` is a class that keeps track of tempo and converts between time and ticks. To change the bpm, you must be careful to avoid a discontinuity in ticks.

Scheduler

- `Scheduler` is a class that manages a list of `Commands` (functions) to be executed in the future.
- Our scheduler keeps track of these commands sorted by tick.
- The Scheduler knows when to execute commands because it has a `Clock` – for keeping track of time, and a `TempoMap` for converting time to ticks.
- The Scheduler API is:
    - `cmd = post_at_tick(tick, func, args)`: This will cause `func` to get called at `tick` (which should be in the future). The function `func` will be called as: `func(tick, args).`

- The `cmd` returned by `post_at_tick` is a reference to the command. You can hold on to this `cmd` and use that to cancel the command before it gets executed.
- Use `Scheduler.remove(cmd)` to remove a pending function. You can also use `cmd.execute()` to execute the command yourself. This can be useful if you cancel a future note-off command but want that command to still happen immediately.
- Scheduler also has `get_time()` and `get_tick()` for convenience.

Example - Metronome

- The Metronome is a simple example of a class that uses the Scheduler and Synth to play a steady beat.
- `_noteon` and `_noteoff` functions are posted to occur at the right times (the right ticks!). Note the importance of having a `_noteoff` for each `_noteon.`
- Changing the bpm of `SimpleTempoMap` is a good way to see the importance of keeping tick continuity.
- See metro.py to see how this class works. It is important to pick a channel for this class – that is the channel that will be used for playing notes. And it is important to set the correct program for the instrument type you want. In this case, we use a percussion bank (128, 0).

Precise Scheduling

- Unfortunatley, the simple Scheduler we built has a weakness – it is only as accurate as the calls to `on_update().` These have a fairly course resolution (~16ms) which can cause jitter in fast musical passages.
- It is possible to improve this behavior by scheduling functions to execute at precise locations within an audio buffer. We can use the audio system's sample-processing mechanism (ie, calls to `generate()`) to measure the passage of time by counting samples.
- Audio generation then happens in sub-chunks. Instead of generating an entire buffer of audio in one shot, that buffer can be processed in a few steps:
  - A sub-chunk of audio is generated that represents audio happening prior to the scheduled event.
  - Then the event is executed (causing, say, a note-on to be generated)
  - The rest of the audio buffer is generated, now containing the audio related to the note-on

Example – NoteSequencer

- We can make a class that sequences a linear set of pitches. We pick a really simple data structure – a list of `(duration, pitch)` pairs.
- For each note, there are three steps:
  - Stop the previous note (if there is one)
  - Play the current note.
  - Post the next note to play `duration` ticks from now.
- Looping can be achieved by setting the play index to 0 after the end of the sequence
- Resets can be achieved by defining `pitch==0` as a rest.

Responsive Interfaces

- Example: Harmonix's first product - The Axe
- Automatic rhythm generation
  - Pros: most people have bad rhythm. Automating rhythmic activity can lead to better sounding interactive music systems.
  - Cons: Automatic rhythm system can sometimes be less immediately responsive.
- Automatic pitch selection
  - Pros: all notes can "sound good"
  - Cons: takes control away from the user.

- The challenge is to make a system that is responsive, and gives as much control as possible, while avoiding “bad sounding notes”