Assignment 1: Sine Synthesis

Overview

In lecture, we learned about creating a tone with a basic sine wave. We created an Audio class as an interface for generating sound, and we instantiated that Audio class in a simple framework that can receive real-time keyboard input to control the Audio class instance.

This assignment will expand the functionality of the audio module as follows:

- Allow multiple simultaneous tones
- Specify pitches (A, B-flat, C, etc...) instead of frequencies
- Specify a duration for each note as well as an amplitude shape (ie, an envelope)
- Allow for notes of differing timbres

You will in essence be creating a very simple synthesizer, which you can then play by mapping keys on your laptop keyboard to “play note” commands.

Once you get the whole thing working, create something fun. For example, you don't have to just have the chromatic scale mapped to your keyboard. What other mappings can you create that are fun to play?

Part 1

Refactor the Audio interface and implementation so that Audio is a more generic player / mixer of audio and the details of specifying what kind of sound should be played is relegated to a class. We define an interface called *Generator*. A class that knows how to generate audio data implements the Generator interface. Generators are responsible for generating a single kind of sound (like a single note!). The Audio class's job is to manage a bunch of generators and combine their output together into a single channel.

Audio should have the following interface:

```
class Audio:
   def __init__(self):
      pass

   def add_generator(self, gen):
      pass

   def set_volume(self, volume):
      pass

   def get_volume(self):
      return 0

   def _callback(self, in_data, frame_count, time_info, status):
```

```
        return (buffer_as_float_array, pyaudio.paContinue)
```

A generator's job is to create audio. It must implement the following interface that the Audio class will call:

```
class Generator:
   def generate(self, frames):
      return (array, continue_flag)
```

A generator must return a *numpy array* of length *frames* and a flag (*True* or *False*) indicating whether the generator is done (*False*), or has more audio to generate next time around (*True*). So, for example, you would write class *NoteGenerator* to create audio in a *numpy array* and continue returning *True* so long as the note is playing. When the note ends, it will return *False.*

Audio._callback(…) should iterate through all generators, mixing (ie, adding) their results into a master buffer that is returned to the pyAudio callback. If a generator returns *False*, it should be removed from the list.

Create a NoteGenerator class that is instantiated with a frequency (Hertz) and duration (Seconds) value. When added to the Audio class, it will play the said frequency for the correct duration.

In MainWidget, have a few key-down messages play notes with specific pitches and durations of your choice.

Part 2

Make your note generator take a pitch value instead of a frequency. The pitch value is an integer, and a continuous set of integers defines the chromatic scale. We will learn about MIDI soon, so we'll stick with the MIDI convention that frequency A440 = pitch value 69. That also means that middle C is 60 (9 semitones lower than the A).

As discussed in class, we will use equal-tempered tuning. To find the frequency one octave higher than a note with frequency F, we multiply by 2. To find the frequency one semitone higher than a note with frequency F, we multiply by the 12th root of 2. See: http://en.wikipedia.org/wiki/Pitch_%28music%29 for more details.

Write your Pitch to Frequency conversion in separate function:

```
def pitch_to_frequency(pitch) :
   ...
   return freq
```

Part 3

Right now, your note generator plays a single volume pitch for N seconds and abruptly turns off (most likely causing a slight pop/click). Let's make that nicer by creating an amplitude envelope.

This envelope lasts the full duration of the note. Look at Section 9.2.1 In Musimathics, Volume 2. You will see how to generate an envelope with a specific attack time and decay time. For simplicity, ignore the attack time (so set a=0). This makes the curve simpler to compute because it is single continuous function. Create the decay envelope as described in Section 9.2.1. You can play around with the parameter n2 to get different types of decay curves.

As an optional part of this assignment, if you would like, implement the attack and decay portion of the envelope curve.

Part 4

A sine wave is boring. Let's make other kinds of sounds. Expand the capabilities of your note generator to add a series of overtone frequencies to the fundamental frequency. You can compactly represent the overtone series as an array of amplitudes such as [a0, a1, a2, a3…], where aN is the amplitude of the N'th overtone. In Musicmathics, Volume 2 Section 9.25 - 9.2.7, you can see how to construct the "Geometric Waveforms": Square Wave, Triangle Wave, and Sawtooth Wave.

Give your note generator an additional argument - a list of overtone amplitudes. Since these amplitudes values drop off fairly quickly, you only need to provide the first 10 or so values. Beyond that, the effect is minimal. Create a few different note timbres (Square Wave, Sawtooth). You can also try other values to see how a note's timbre changes with different strengths of the overtone series.

Part 5

Come up with a creative way to use this system that let's you "perform" something. Create mappings between your keyboard and your synthesizer.
Some ideas:

- Define a sequence of notes (like you saw in the first day of class) so you can trigger a melody.
- Hitting a single key can play more than one note – it can play an entire chord.
- You can trigger staccato (short) notes and legato (long) notes to create different melodic effects, or have different timbres for different parts of your piece.
- You can have a set of "meta keys" that have larger effects and can affect other mappings. For example, if you have 1-9 mapped to a C-major scale, hitting a single other key can change that mapping to a different scale.

Write up a short description of the how to control your system in a README file.
Create a quick / rough / unedited video of your performance (doesn't need to be long: 30-60 seconds is fine), and submit that as well.

Finally...

Remember to add a paragraph about how you collaborated. Please have good comments in your code. When submitting your solution, submit a zip file that has all the necessary files. For

example, if you used other files that I provided (like core.py), re-provide those files back to me in your submission. Upload your zip file to Stellar / Homework.