Assignment 2: Wave Files

Overview

In lecture, we learned about audio data stored in WAVE files, how to read them, how to play them in full, and how to play them as short snippets.

Your assignment is to flesh out the basic interfaces that we developed in class with more features. Then, put these to good use by creating a mini performance system.
Note that the assignment code contains *superstition.mp3* but not *superstition.wav* (so the download would not be too large). If you want to play with the same code as we saw in Lecture, you should first convert the mp3 to a wave file using Audacity.

Part 0

Find some audio - a song (or two) that you want to play with - and convert it to a WAV file with Audacity.

Part 1

Add shuttle controls to `WaveGenerator`. Allow the user to:

- Pause and Play the audio playback. A toggle function is also handy to allow toggling between these two states. When paused, the `WaveGenerator` should still generate audio – but that audio should be silence!
- Reset back to the beginning of the file. On your music player, this is usually connected to the "Reset Button."

Test these functions by hooking them up to key presses.

Part 2

Create some regions in your WAVE file using Sonic Visualizer:

- See details on how to do this in *LectureNotes2.pdf*.
- Export the regions layer and call it <name>_regions.txt

Write the code to read this text file so that you can easily create some `WaveBuffers` from the data file:

- Make a class `AudioRegion` that holds the information for one Region. It should have a name, a start frame, and a length in frames (no audio data). This is just a simple data container.
- Make a class `SongRegions` that holds a collection of `AudioRegions` and knows how to create that collection by reading and parsing the _regions.txt file.
- Finally, create the function `make_wave_buffers` that returns a python dictionary of `WaveBuffers` from the `SongRegions`. The dictionary should be of the format:

```
{
   name1: buffer1,
        name2: buffer2 :
    } :
```
where nameN is a string and bufferN is a WaveBuffer instance. Now, you can easily play back any of your buffers by retrieving a buffer by name, creating a WaveGenerator, and handing it off to the *Mixer* class.

Test your code by triggering some WaveGenerators based on key_down events.

Part 3

Add a looping option to WaveGenerator.generate(). If looping is enabled, when the generator gets to the end of the audio data, it will automatically loop back to the beginning. WaveGenerator's constructor should take another argument:

```
    class WaveGenerator(object): :
        def __init__(self, wave_source, loop=False): :
```

But this means that if looping is on, the generator will never stop producing audio, so add a function to stop the generator:

```
        def release(self): :
              ... :
```

This release function should work even if looping is not turned on. The difference between release() and pause() is that release() terminates the generator by causing the continue_flag to be False.

Test release() by hooking into keyboard keys. When a key is pressed, it should start a WaveGenerator. When that same key is released, it should release() that same WaveGenerator. Use on_key_up() and on_key_down().

When you author regions for looping in Sonic Visualizer, you'll need to pay careful attention to the start and end of the region. Use the built-in playback looping ability to test out what the region sounds like when it loops.

Part 4

So far, all of our audio has played back at the intended speed (ie, the speed that it was recorded). Let's mess with that as well! Create a new generator called SpeedModulator that doesn't create its own audio, but knows how to modulate audio from another generator. SpeedModulator looks like this:

```
    class SpeedModulator(object): :
        def __init__(self, generator, speed = 1.0): :
           ... :
        def set_speed(self, speed): :
           ... :
        def generate(self, num_frames, num_channels) : :
           ... :
```

When `SpeedModulator` is asked for audio data, it first asks its own internal generator for audio data, then stretches or squashes that data, and finally returns that modified audio data. You can also adjust the speed variable on-the-fly with `set_speed()`.

Implementing speed change requires resampling the audio using interpolation. We discussed how interpolation works in class, but implementing it is a job for you. You will want to use the function `np.interp()`. See *LectureNotes2.pdf* for more discussion as well.

Careful about interpolation using stereo audio! You must de-interleave, change speeds, and then re-interleave.

Part 5

Now, pull it all together to make a mini-performance system that combines some or all of the systems you have built so far:

- Full song playback
- Short buffer playback
- Looping
- Speed changes
- Note synthesis

Find song / audio content you like, create mappings to trigger that content in different ways, and if you like, add mappings for synth note playback on top. You can build on the work you did in Assignment 1, or start with something else.

Some ideas:

- Drum loops are fun. Find audio of drums and loop those
- Short audio buffers that only contain one note or chord can be triggered as notes. If you carefully alter their speed, you can get a variety of different pitches as well.
- Loop a long buffer (say 4 or 8 beats) and trigger short buffer on top of that. This works well if the music of the long buffer acts as background and the short buffers can act as melody or foreground music.
- Identify the chords / harmonic progression of some areas of a song and make note mappings that match the harmonies and can be played on top of those areas.

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. It doesn't need to be long: 30-60 seconds is fine. You can either submit the video file or (better) upload it to YouTube/Vimeo and provide a link in the README. Remember to also include the audio itself: wave file or mp3 files.

Finally...

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.