TONES KeyboardPitch Model, KeyboardNote, and KeyboardNoteGroup

anematode

January 3, 2019

1 Who is this intended for?

It's intended for me. I might read this in a few months because I didn't know what the hell I was doing. Thus, it's written in a somewhat understandable way, but it won't try to explain ideas that aren't a product of this project.

2 KeyboardPitch

A keyboard pitch refers to a unique note "on the keyboard." It does not refer to a specific pitch; this is important because then alternative/ microtonal tuning systems would be hard to implement and that would suck. Instead, it simply refers to something like "A4," "F4," etc.

We model a keyboard pitch as an integer from 0 to 144 inclusive, corresponding with the notes C-1 to C11. In the old TONES it was a special class, but this is such a simple egg that I don't think it deserves another class.

When we convert a note to a "name" like C1 or Db5, we can use the following formula:

noteToName(note) := $octaves\_note \mod 12 + stringify(\lfloor note/12 \rfloor - 1)$

where octaves is ["C"', "C#"', "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"] and zero-indexed. For example, 69 gives us index 9 and $\lfloor 5.75 \rfloor - 1 = 4$, or A4.

The inverse of this function is the nameToNote function, which is a bit more complicated to deal with flats (b), double flats (bb), sharps (# or s), and double sharps (## or ss). We first apply the regex

$\hat{}([ABCDEFG])(\#|\#\#|B|BB|S|SS)?(-)?([0-9]+)\$$

to an uppercased version of the input. The first (zero-indexed) group (letter name) we reference Dict 1 to get a semitone offset from the octave (l). The second group (accidental) we reference Dict 2 to get a semitone offset of the accidental (a). The third group we reference to get whether the octave number is negative as a true or false (b). Finally, the fourth group is parsed to an integer as the octave number itself (o). Then the note's value is

nameToNote(name) := $l + a + (b\ ?\ -12 : 12) \cdot o + 12$.

The relevant dicts:

Dict 1: letter_nums = {"C" : 0, "D" : 2, "E" : 4, "F" : 5, "G" : 7, "A" : 9, "B" : 11}; Dict 2: accidental_offsets = {"#" : 1, "##" : 2, "B" : -1, "BB" : -2, "b" : -1, "bb" : -2, "s" : 1, "ss" : 2, "S" : 1, "SS" : 2};

For utility and pitch estimation, there's the noteTo12TET function which converts a KeyboardPitch to its 12-TET frequency:

noteTo12TET(n, a4 = 440) := $a4 \cdot 2^{(n-69)/12}$.

There's also the concept of a KeyboardInterval, which is just a distance between two KeyboardPitches. Obviously, the KeyboardInterval between a and b is just $|a-b|$. For working with this there are the functions nameToInterval and intervalToName, but I don't expect them to be used and they are just an artifact from the old TONES, so I won't explain how they work here. Their algorithm is lengthy but straightforward. To get the value in cents of a KeyboardInterval **in 12 TET**, multiply by 100.

3 KeyboardNote

A KeyboardNote is a KeyboardPitch played at a certain time, with a certain length, and with certain parameters. Rather simple.

Properties:

pitch: KeyboardPitch of the note

start: time (probably in beats) to start

length: length (probably in beats) to hold the note

vel: velocity (0 to 1 is standard range)

pan: pan (-1 to 1 is standard)

fine: small pitch adjustments in cents

custom: object you can pass for custom properties

get/set end: start + length

Methods:

constructor(params = $\{pitch : A4, start : 0, length : 1, vel : 1, pan : 0, fine : 0, custom : \{\}, [end]\}$)

translateX(x): shift it over by x units

scaleX(x): scale it by a factor of x, rejecting x ≤ 0

transpose(semitones): transpose it by some number of semitones

clone(): return a cloned KeyboardNote, also copying (but not deep-copying) the custom property

toJSON(): convert to JSON format. This isn't too complicated; we basically just return what "params" should be when constructing the note but with abbreviated names for compactness:

pitch $\rightarrow$ p, start $\rightarrow$ s, length $\rightarrow$ l, vel $\rightarrow$ v, pan $\rightarrow$ n, fine $\rightarrow$ f, custom $\rightarrow$ c

static fromJSON(json): static function building a note from the aforementioned js object (in this case, it's basically the same as the constructor with different names)

4 KeyboardNoteGroup

A KeyboardNoteGroup is simply a sorted array of KeyboardNotes that represents a string of notes to play. It permits notes to intersect, and even notes to be on top of each other; dealing with that during playback will be a problem for the scheduler. Nevertheless, it will include a removeIntersections function for convenience (and for my algorithmic testing!).

Properties:

notes: array of KeyboardNotes, which will usually be sorted unless you mess with it directly, after which you should call sort() _sorted: internal checker for whether the array is sorted

Methods:

constructor(notes=[]): sets notes to given notes

sort(): internally sort the keyboard notes in order of start time

addNote(note, clone=false): add a note, asking whether to clone it

get noteCount: number of notes

deleteNote(note): remove a note; this will also remove duplicate notes. Returns true if it removed any notes, false if not

deleteNoteIf(func): removes any notes which satisfy func; returns true if it removed any notes, false if not

translateX(x): shift it over by x units

scaleX(x): scale it by a factor of x, rejecting x ¡= 0

reverse(): reverse the notegroup

transpose(semitones): transpose it by some number of semitones

apply(func): call the function on all notes

some(func): check whether the function is satisfied for any notes

all(func): check whether the function is satisfied for all notes

select(func): returns array of notes that satisfy func

minX(): minimum note x value

maxX(): maximum note x value

length(): diff between maxX and minX

minPitch(): minimum note pitch (lowest note reached)

maxPitch(): maximum note pitch (highest note reached)

* generateNotes(start_x = -Infinity, end_x = Infinity): a generator yielding notes to play from start_x to end_x. Note that if start_x is in the middle of a note, the start of that note will be adjusted in the returned value to start at start_x instead. Similarly, end_x will tell the note to end at end_x if it cuts through the note. Notes ending on start_x will not be played; notes starting on end_x will also not be played. The original note objects may be returned, but only if they don't have to be adjusted; this function will not change the notes array at all.

snip(start_x = -Infinity, end_x = Infinity): return a new KeyboardNoteGroup, copying all KeyboardNotes, from start_x to end_x and with the trimmings described in the previous function. Modifications of the original group will not change the snipped group after it's been snipped. Note that snip() serves as a clone function.

join(group, clone=true, offsetX = maxX()): join group to this group starting at offsetX, modifying this group. If clone, then clone each copied note; if not, then just use the original objects (and thus the joined group will get modified)

clone(): clone this group

add(group, offsetX = maxX()): same as join, but must be cloned and return a new group

repeat(times, repeatX = maxX()): repeat the notes, modifying the original group, with spacing of repeatX. Furthermore, an array with length $times - 1$ can be passed to repeatX to change each repeat length

removeIntersections(): Remove all intersections between notes as specified below, modifying the original group. This won't be used during actual playback, so we don't need to optimize this. This is how it should work:

Note -1. notes on diff. pitches should never occlude each other (obviously)

Note 0. equal end and start times are not an occlusion

Step 1. if two (or more) notes start at the same exact time, remove all notes except the longest one by setting them to null; if they are all the same length, remove all except any one note, which one doesn't matter. Also, keep track of non-null notes in a dictionary mapping pitches to arrays of indices of notes with that pitch.

Step 2. For each pitch, go through each note for that pitch in start order using the previously discussed dictionary. Every time a note starts before the previous one ends (strict inequality, equal values shouldn't trigger it), set the end of the previous note to the start of this note.

Step 3. Remove all "null"s because they're just artifacts from Step 1 kept there to preserve the validity of the indexes generated there as well.

toJSON(): convert to a JSON format, basically just $\{n : notes, s : \_sorted\}$

static fromJSON(json): construct a NoteGroup from JSON