What does this project do for you?

This project provides one module per sample instrument. Each module provides a function, of the same name as the instrument, which returns a ugen suitable for use with overtone. A simple way to describe a ugen might be a composable, commutable unit of sound specification; for more information see. Each module also provides a function which accepts the same form of input, but which affects sounds rather than returning a ugen. The 'definst' infrastructure is not used because these sampled instruments have many variants, e.g. vibrato or non-vibrato, of the same note.

How to use the sampled instrument functions

Let's use the example of a cello, because all the samples instruments are similar to this.

loading samples

(use 'sampled-cello)

assuming philharmonia-samples/src is in your classpath.

(:require [sampled-cello :refer [cello celloi cello-inst]])

all the sampled instrument files have similar information in them, sharing the same name, so you should probably 'require' rather than 'use' it, and you likely don't care about those other names, so you could just refer the ugen producing function (cello) and the instrument-like function (celloi).

You may want to only refer celloi if you are exclusively live-coding (for sake of less typing), or you may want to refer cello-inst if you are writing a program you intend to re-use (for sake of a semantic, rememberable name).

Loading all the samples associated with an instrument can produce a var taking up as little as 21MB, in the case of the mandolin, and as much as 169 MB, in the case of the violin. For this reason, the samples are grouped into separate files by instrument.

calling samples

```
(cello)
```

produces a ugen playing the default sample.

```
(celloi)
```

will make your speakers play the default sample

`(cello-inst)`

equivalent to above

`(demo (cello))`

equivalent to above

`(cello {:note "As3" :loudness "piano" :style "vibrato" :duration "025"])`

you can specify all parameters in a map.

`(cello {:loudness "forte" :style "vibrato" :duration "025" :note "G4"])`

order does not matter in the map, so this will also produce a sound.

`(cello {:note "As3"})`

you can specify only the parameters you wish, and the default value for each unspecified parameter gets filled in.

`(cello :note "As3" :style "vibrato")`

or, since overtone is meant for live coding, you can save some precious keystrokes by passing keys and values instead of an explicit map.

`(cello {:note 60})`

passing an integer will make the value interpreted as a midi note, rather than a note name.

`(cello 60)`

'note' is arguably the most significant feature, so a single scalar argument will be interpreted as a note.

`(cello` ["As3" "piano" "vibrato"])

when passing a vector, you must specify all features.

`(celloi :note 58)`

if your use case lends itself to functions that produce sounds (rather than ugens) when called, you can use function with 'i' as a suffix, which is otherwise identical in how it's used.

Because `(cello)` produces a ugen, it could be used like so:

```
(defcgen [note {:default 60}]
    (:ar
       (let [freq (midicps note)]
         (+ (sin-osc note)
            (cello note)
            (white-noise)))))
```

This simply overlaps the sounds of a sine wave playing a frequency, a cello playing a note which the human ear predominantly associates with that same frequency, and some white noise.

error handling

If you specify a parameter that does not exist, that information is discarded and the default value for the parameter you meant is used. If you specify a value that a parameter cannot take, autocorrection is attempted. If no suitable correction, the default value is used. If you specify a combination of allowed and parameters that has no corresponding sample, an error message is printed and the expression evaluates to false.

It may make sense to make the decision of whether to the use of default values be a configurable thing. Rather than simply printing an error message when there is no corresponding sample for what is requested, it may make more sense to try to play a sample most similar to the requested sample, but that's a lot of effort and this project is not currently in heavy enough use for that to pay off.

customizations

- You can scale up or scale back how eagerly misspellings are autocorrected for each parameter. The `distance-maxes` var in each sampled instrument file has keys corresponding to the parameters, and values corresponding to the maximum string distance between allowed values and values that will be corrected. Note, these values can be set to '0' if you do not want any autocorrection.
- You can change what the default parameters for an instrument are by changing the `defaults` var.

How to get things going:

Overtone requires ".wav" files, but the philharmonia website provides mp3s. I've gone through the work of converting them, and you can download the samples from my google drive

The library expects a directory structure with naming conventions like what you can download at that link. The sample's root directory can be anywhere on your filesystem, and the samples can be used if the `sampleroot` var in `sample_utils.clj` file points to it.

Welcome Changes

- The 1.0.0 version of this project might involve more precise names for the instrument features. For example, I chose “loudness” as the name for the class of features like “piano” and “forte”, and “style” as the name for features like “arco-normal”, but this might not be the terminology musicians want to work with.
- Of course, more samples and a better multi-gigabye public storage solution would be a plus.
- A mechanism to produce a true overtone instrument rather than the current ‘cello-inst’ system might make more sense, but I found it difficult to get parameters into instruments and the current solution works okay.