FLUENTLY REMIXING MUSICAL OBJECTS WITH HIGHER-ORDER FUNCTIONS

Adam T. Lindsay, David Hutchison

Computing Department,
Lancaster University
Lancaster, United Kingdom
{atl,dh}@comp.lancs.ac.uk

ABSTRACT

Soon after the Echo Nest Remix API was made publicly available and open source, the primary author began aggressively enhancing the Python framework for re-editing music based on perceptually-based musical analyses. The basic principles of this API – integrating content-based metadata with the underlying signal – are described in the paper, then the authors' enhancements are described.

The libraries moved from supporting an imperative coding style to incorporating influences from functional programming and domain specific languages to allow for a much more fluent, terse coding style, allowing users to concentrate on the functions needed to find the portions of the song that were interesting, and modifying them. The paper then goes on to describe enhancements involving mixing multiple sources with one another and enabling user-created and user-modifiable effects that are controlled by direct manipulation of the objects that represent the sound. Revelations that the Remix API does not need to be as integrated as it currently is point to future directions for the API at the end of the paper.

1. INTRODUCTION

The Echo Nest is one of the most visible commercial proponents of modern practice in machine listening and musical information retrieval. On 8 September 2008, the company made its first public, open-source release of its Remix API on the Google Code hosting site[1]. Since then, there have been frequent and vigorous updates, rapidly evolving the framework from a distillation of the code used in internal projects into a domain-specific language capable of radically re-editing music with very concise, music-oriented commands.

The Echo Nest Remix API ('Remix API' hereafter) is a programming framework written in Python that is oriented towards making computational re-edits of existing music very easy. While analytic feature-driven approaches to re-editing material have been proposed [2] and executed [3] before, a large proportion of the API's power is derived from the musically-sensitive analyses made available through the same company's Analysis API. The remote analyses provide hierarchical, content-based segmentation of music, and the Remix API provides many methods and classes for using that segmentation in order to re-edit music seamlessly.

After giving a brief overview in sections two and three of what was made available in the original version of the Remix API, this paper gives a narrative of the various design decisions the author made since the Echo Nest open-sourced the framework, evolving it into a somewhat idiosyncratic, but terse and powerful, domain-specific language for manipulating music, not just in the time dimension, but with multiple parallel tracks and user-defined effects. New and old are otherwise not distinguished in the paper, as all the code exists as part of the same open-source project.

2. THE ECHO NEST ANALYZE API

The Echo Nest Analyze API ('Analysis API') performs efficient, perception-based signal processing, simulating how people hear sounds (accounting for temporal and pitch masking, for example [4]) in order to extract higher-level musical knowledge about a sound [5]. From the point of view of a Remix API user, one of the most useful outputs is a three-level, hierarchical rhythmic analysis of a given song: beats, bars, and tatums, as in Figure 1.

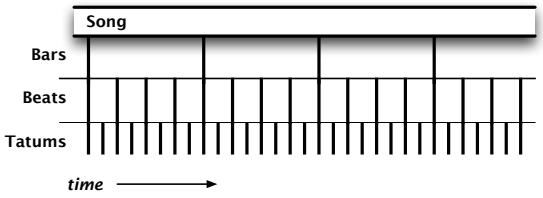


Figure 1: *Bars, beats, and tatums, as analysed in a song.*

The analyses that lead to bar and beat grouping are not perfect, but they work reasonably well in a wide variety of situations. Tatums, as the most fundamental rhythmic unit, as described in [6], are considerably more reliable.

There are two other ways the Analysis API divides a song: into sections and segments.

- *Sections* reflect large scale changes in the song's sound, whether it be in rhythmic pattern, instrumentation, or harmonic structure. Although it is often described as dividing a song into a verse-chorus-bridge structure, reality yields not as clear-cut results as that.
- *Segments* are the most elemental portions of a song. They do not precisely align with beats or tatums, but rather are representative of some sort of 'event' in the song, such as a note or a drum beat.

Since there are likely many instruments playing many notes that overlap, the practical result of this part of the analysis is that segments capture the starts of notes and drum beats. Because segments are elemental, containing one event, they also act as containers for further sonic analysis. They contain information on

the loudness envelope, the overall pitch content (with the relative loudness of each step of the scale), and the timbre of the segment. For the purposes of the Analysis API, timbre is reduced to twelve component time-frequency elements: they are the first twelve principal components of the time-frequency shape of the event's musical surface.

By combining this low-level information with general music knowledge, the Analysis API also offers estimates about global features of the entire song, such as time signature, key signature, and tempo. As with any case of reducing thousands of complex, evolving values with a single number, there exist some edge cases in particularly irregular or complex sonic material where performance is not perfect. Nevertheless, with proper sanity checks, it is very useful information for the purposes of the Remix API.

The Analysis API begins its work with a user uploading an MP3 file to its servers. The analysis itself happens on the Echo Nest's centralized servers and usually takes far less time to complete than it takes to upload the file from a domestic broadband connection. Once the analysis is complete, simple web (HTTP) requests that identify the file that was uploaded receive responses in XML. For example, an API call to `get_beats` would yield an XML-formatted list of beats with their starting times and confidence values.

3. BASIC REMIX API OPERATION

This paper details two data models: the original model, and therefore the fundamentals of those seen in the Remix API (as seen in Figure 2) and another model that shows various enhancements – the current data model as it stands.

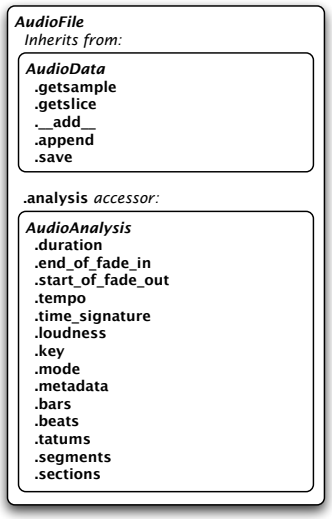


Figure 2: *The basic accessor methods available in the Remix API from the start.*

The foundation of the Remix API is derived from the Echo Nest Analyze API: the same metadata and perceptual analysis that is exposed there forms the foundation of the data structures used in the Remix API. Jehan [7] has described the underlying techniques for the analysis exposed in the Analysis API. The professional offering adds rhythmic groupings, such as beat and bar measurements, to the event-synchronous audio analyses offered in his PhD research.

The Echo Nest Remix API first appeared with three main parts: a library for transparent access to their Analysis API, classes and methods for performing simple audio manipulation, and some functions for gluing these together. In providing transparent access to the Analysis API, the Remix API offered methods for uploading audio to the Echo Nest servers and for accessing the subsequent analyses available through their web API. The methods would parse and transparently cache (in memory) the results of calls to the Analysis API, returning Python-native datatypes and objects. The simple audio manipulation involved loading and saving multiple audio formats, and representing them internally with sample data kept in NumPy [8] n-dimensional arrays and other metadata such as sample rate, number of channels, and a pointer to the location of the final sample.

These two parts, analysis metadata and audio, were glued together with the remainder of the original Remix API. All rhythmic units are represented as `AudioQuantum` objects within the API. An `AudioQuantum` is represented at the minimum with a start (time offset, in seconds, from the beginning of the original file) and a duration (also in seconds). Given an `AudioQuantum` and a source, the samples for the specific time interval of the `AudioQuantum` can be retrieved. With `AudioQuantum`s collected into a generic Python list, a new edit of the original material can be obtained by a function, getpieces().

Much more sophisticated manipulations could be performed on an *ad hoc* basis with additional user classes, such as those shown in the demonstration code for the popular "More Cowbell" site, but none of them had yet been included in the API. The essence of the original Remix API lay in accessing the Echo Nest Analyze API over HTTP, offering access to audio via the NumPy libraries, and the sample-accurate glue between them.

4. RE-EDITING MUSIC WITH FUNCTIONAL PROGRAMMING

With the basic API operation established, the primary author's attention turned to utilising more of the rich data held within the Analysis API. Segments hold rich timbral, loudness envelope, and pitch data but they were unused by the Remix API. There are implicit relationships between tatums, beats, and bars that went un-exploited. At that point, there was nothing in the API that took advantage of the fact that all elements in a list full of `AudioQuantum`s had special properties. The first set of external changes set to work on exposing these properties.

4.1. Introducing `that()`

Although from the start, anyone could manually loop through the rhythmic lists and collect the audio quanta that were of interest, the fact that most operations began with an `AudioQuantumList` and ended with a different one suggested a more functional programming style. Instead of iterating through a loop, it is perfectly plausible to filter the entire loop through a single function. Once filtering is accomplished, then it follows to use the same paradigm for sorting and otherwise applying transformative functions on `AudioQuantumList`s. The change that most informed the developments that followed with the Remix API, giving it its

unique flavour, was the introduction of the `AudioQuantumList` method, '`that()`'.

Not only did `that()` establish the beginnings of a functional programming style, but it also served as the beachhead for a fluent interface. Fluent interfaces are related to natural language APIs and internal (or embedded) domain-specific languages [9]. The goal was to establish a pleasant, natural-language-style portion of the language, where a naive user could use existing libraries and functions and get usable results out while writing a minimum of code.

A guiding principle of this interface paradigm is that each of these natural language methods operates on an `AudioQuantumList` and returns a new `AudioQuantumList`. In this way, filtering, sorting, and otherwise modifying lists of elements can be chained so that an entire programmatic remix could conceivably be expressed in a one-line program. However, since `AudioQuantumLists` are native Python sub-classes of lists, many other familiar methods are available to the user, if what they wish to accomplish lies outside of the filter-sort-and-manipulate paradigm.

In order to make the natural language aspects of the system concrete were a small library of queries and selection filters ready for use with `that()` and similar methods. The majority of the selection filters are higher-order functions that generally take one or two arguments and return a function of a single argument. Since this pattern is so pervasive but is a non-trivial way of writing functions, definitions are made much easier with a Python decorator pattern [10]:

```
def overlap(aq):
    def fun(x):
        if x.end > aq.start and \
                x.start < aq.end:
            return x
    return fun
```

...was simplified to this:

```
@selector
def overlap(x, audio_quantum):
    return x.end > aq.start and \
        x.start < aq.end
```

(Note that in this and most Python examples that follow, backslashes indicate line continuations interrupting a single statement.)

The naming of such functions was essential to establishing the fluent interface, as well. The linguistic pattern that ended up working was to name selection filter functions as verb phrases that agree with a plural noun, typically in a restrictive clause introduced by 'that,' but may also work as the end of an subordinate clause. For example, given a `section1` in a song and the `beats` derived from an analysis:

```
beats.that(overlap(section1)) \
      .that(fall_on_the(2))
```

This code fragment takes the subset of all beats that have some portion of their time intervals overlap with the time interval of `section1`, and takes the further subset of those beats that fall on the second beat of their respective `bars`. Admittedly, the idiom can present some difficulties to get used to writing: while the various function and method calls read as coherent fragments of English, the punctuation – knowing what must be a Python method call separated with a period (`.`), and what should be a pre-defined function as an argument to such a method – takes a non-trivial amount of learning. Nevertheless, we believe it has been beneficial to introduce such linguistic patterns to the API. It provides a friendly introduction and aids a sense of empowerment in the user to understand the available example code.

4.2. Rhythmic context

With conventions for scanning the native `AudioQuantumLists` established, the next development was to use these techniques, plus the reasonable (but not always true) assumption that if an analysis has found `beats`, it also has `tatums` and `bars` available.

By giving each `AudioQuantum` some representation of its context within an `AudioQuantumList`, and giving each `AudioQuantumList` access to the `AudioAnalysis` that contains it, we enabled each component in the rhythmic partitioning of the original song to have a representation of its rhythmic context. The rhythmic context of an `AudioQuantum` allows a user to programmatically access the `AudioQuantum`'s neighbours, forward and backwards in time, of the same type of rhythmic quantum. The context allows one to navigate up the hierarchy (beat to bar, tatum to beat) to its rhythmic 'parent.' Conversely, one can access a rhythmic unit's children (beat to tatum, bar to beat) by going down the hierarchy. The children of an `AudioQuantum`'s parent form the 'group' to which that `AudioQuantum` belongs. These concepts are all illustrated in Figure 3.

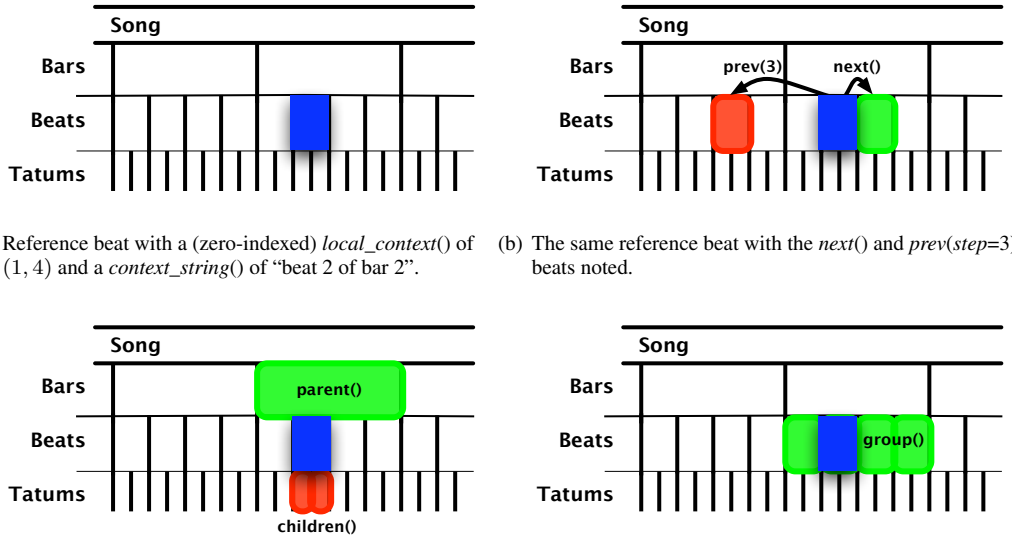
Additionally, a user may also programmatically access relative and absolute indices. The `absolute_context()` returns a Python tuple with the index number of the item in its entire containing `AudioQuantumList` and the length of that entire list. The `local_context()` returns a similar tuple, but only with reference to its `group()`.

Internally, all of these parent-child relationships are computed on the fly by using the knowledge of the kind of `AudioQuantum` in question, the circular links back to the containing lists, and the same queries that the `that()` method normally works with. What might be seen by some as syntactic sugar yields immediately accessible navigation and queriable context information.

4.3. More functional and fluent additions

After establishing the functional programming pattern and trying it with further use, additional methods were prototyped and then added. First to be added was the `sorted_by()` method for `AudioQuantumLists`. The input to the method is a function of a single argument that returns a scalar value to be used as a sorting key. Many of the input functions are trivial, such as returning the value of a given accessor on an input `AudioQuantum`, but framing the method this way allows for rich comparisons as well, such as ordering segments by the mean-squared distance from a reference segment's timbre vector. The naming convention here is of nouns (or noun phrases) that could conceivably follow "sorted by."

An early version of `that()` was implemented using the python built-in function '`map()`' instead of the more specific '`filter()`' built-in. A curious side-effect was that returning the current `AudioQuantum` being tested was merely a convention: the selection filters could actually return any `AudioQuantum` available to them. This could be used for much more sophisticated remix generation, but in the interest of simplicity and retaining conceptual clarity to the very fundamental `that()` method, a new `AudioQuantum` method was introduced: `beget()`. Any element within an `AudioQuantumList` may `beget()`



(a) Reference beat with a (zero-indexed) *local_context*() of $(1, 4)$ and a *context_string*() of "beat 2 of bar 2".

(b) The same reference beat with the *next*() and *prev*(*step*=3) beats noted.

(c) The same reference beat with its *parent*() bar and *children*() tatums noted.

(d) The same reference beat with its *group*() bar noted. The group is its *parent*()'s *children*(): its siblings.

Figure 3: *Rhythmic context in relation to a beat.*

zero, one, or more `AudioQuantum`s. By convention, multiple `AudioQuantum`s are returned as an `AudioQuantumList`, to aid in further processing chains if needed.

Another method turned out to be particularly helpful for remixing: `changed_by(fn, if_they=cond)`. In this functional programming "`map()`" variant, the function passed to the named `if_they` argument is used as a filter-like test for each element of an `AudioQuantumList`. If the condition is true, the function `fn` is applied to the `AudioQuantum`; otherwise, the `AudioQuantum` passes through unchanged.

Given a hypothetical function, '`adding(filename)`,' that mixes the audio from an external file, a rudimentary, beat-synchronous drum pattern could be added to a song with a single assignment statement:

```
remix = song.analysis.beats          \
    .changed_by(adding('bdrum.wav'),
        if_they=fall_on_the(1))      \
    .changed_by(adding('snare.wav'),
        if_they=fall_on_the(2))      \
    .changed_by(adding('bdrum.wav'),
        if_they=fall_on_the(3))      \
    .changed_by(adding('snare.wav'),
        if_they=fall_on_the(4))
```

5. OBJECT-ORIENTED MUSIC EFFECTS

5.1. Recursion and the rendering chain

The innovations that completed the previous section do raise additional questions of the simple audio glue: what happens now that a list returned by beget may contain other lists? How might we implement the `adding()` mixer function in the `changed_by()` example? By dint of following the functional programming paradigm to its logical conclusions, we have ended up with recursive data structures: `AudioQuantumLists` may contain other `AudioQuantumLists`. This should not disturb us one bit: music's structures are very hierarchical and recursive, and numerous music representations (e.g., SuperCollider [11], JMSL [12], CLM [13]) use such structures extensively.

Once we make this recursive leap, the hypothetical mixer function is fairly obvious in concept: it is an `AudioQuantumList` wherein all the sounds are simultaneous rather than sequential. Thus, we have the two most important concepts of time-based media according to SMIL 1.0: par(allel) and seq(uential) groups of objects.

Walking the chain of audio quanta and lists thereof suggests that each of these objects should respond to a uniform message, `render()`, and return a uniform structure, an `AudioData` object. An `AudioQuantumList` sequentially sends a `render()` message to each of its components, and `append()`s the data to its own. An `AudioQuantum` obtains its sample data by following back references to the containing song, and returns the data bounded by its time interval. The new `Simultaneous` object sequentially sends a `render()` message to each of its components, and *sums* the resulting data with its own. By adopting a protocol for renderable objects where each responds to a `render()` message, and provides accessors to a `source` (always an `AudioData` object) and a `duration` (in seconds), the Remix API moved from being primarily about re-shuffling segments from an single file to inter-mixing and cutting between any number of source files.

There are a few internal details that work as conventions more than anything else. It is, after all, an open-source project: if users need further capabilities, the flexible architecture of the software ought to be able to handle further enhancements. The `AudioData.endindex` becomes very important, as it

is the running marker for the next insertion point for a list. The `endindex` is advanced by the `AudioQuantum`'s duration when appending to an `AudioQuantumList`, and not at all when summing to a `Simultaneous` object. The duration of a `Simultaneous` is by default that of its first child's `AudioQuantum`. Although the duration of an object determines the insertion point of the next object in a sequence, an object may have sample data that extends past the 'rhythmic' duration. This way, overlapping tails and effects do not have to be truncated. Pre-roll, however, is not currently supported.

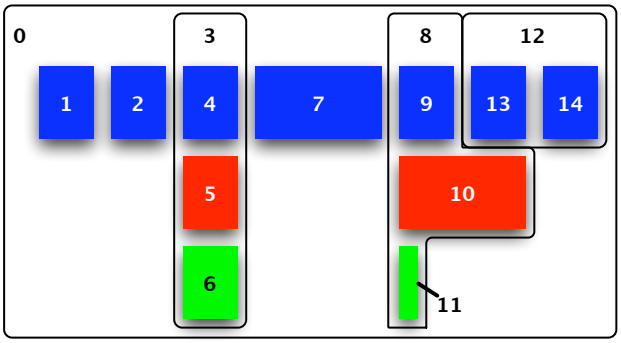


Figure 4: *Given an `AudioQuantumList` (0) that contains six renderable audio objects (three `AudioQuantums`, two `Simultaneous` objects, and one `AudioQuantumList`), the order in which the different objects receive their respective `render()` message is shown. Colours here signify different sources.*

The rendering chain is illustrated in Figure 4, where rendering one item triggers all of its constituent parts to be rendered in turn. Despite segment 10's length, segment 13 directly follows segment 9 in a rhythm consistent throughout that source's meter and tempo.

At this point, it is worthwhile to note one last thing that relates back to the fluent interface. Any renderable audio object can also be `encode()`d. That is, a convenience function common to all renderables is `encode()` without having to call `render()` first. As a result, any object can be written out to an MP3 file or WAVE file, making some remixes literally one-line programs, from input to remixing to output. Given a function that reverses the audio on a given `AudioQuantum`, this variation on example code reverses the fourth beat of each bar with one long statement (split among several lines):

```
audio.AudioFile("input.mp3").analysis \
  .beats                              \
  .changed_by(reversing,              \
    if_they=fall_on_the(4))           \
  .encode("output.mp3")
```

5.2. Modification chain and Plugins

Given the fact that we can now freely manipulate segments across two dimensions (time and track), we now look to a third dimension, in terms of user-defined effects. The paradigm favoured here was to enable direct user manipulation of the `AudioQuantum`'s parameters derived from analysis. The sense here was that – even in code, programmatically – the `AudioQuantum` objects would be malleable and responsive: by manipulating the analytic parameters on the sound segment, the underlying sound can itself be manipulated.

To achieve this, however, required a rethinking of how `AudioQuantums` are manipulated. Until this critical point, `AudioQuantums` were effectively immutable. Although you could conceivably change the start and duration of an `AudioQuantum`, you would change its essence: it would point to a different segment of the original audio. Worse, any other copies of that object retained for remixing would also be affected: only pointers to the original items are passed around. As a result, the `UserAudioQuantum` was introduced. Although an instance appears to be the original `AudioQuantum`, with the same analysis parameters and the same methods, it is actually an entirely different type of object, fairly sparse of methods and native data, but acting as a proxy object for the original `AudioQuantum`. Any unknown methods and accessors in a `UserAudioQuantum` object are passed on to the original `AudioQuantum` for dispatch.

In order to ensure that users manipulate `UserAudioQuantums`, all access methods for `AudioQuantumLists` were re-written. When a user attempts to index an element, a range of elements, or iterate among all the elements in an `AudioQuantumList`, the special methods intervene and transparently return the proxy object.

The relatively hollow insides of a `UserAudioQuantum` means that further methods and accessors can be injected into the `UserAudioQuantum` Class by the user at runtime. The plugin class names an accessor to watch. If a user attempts to set that variable within an `UserAudioQuantum` instance, then the "setter" instance in the plugin class takes over, obtains the original values (from the original `AudioQuantum`) behind the scenes, and then creates new private variables and sets their values in the current instance. If the user attempts to read the variable in the instance, the injected "getter" method takes over, and finds the most appropriate value, either from a private, modified, newly-created variable, or from an appropriate value held in the original `AudioQuantum`.

The third method the plugin class must define is a `modifier()`. It is the effects version of a `render()` method. Given the context of an `AudioQuantum` and the data from the previous effect in the chain, it is the task of the `modifier()` method to create new sample data, presumably according to the accessor it claims to watch.

The canonical use case is installing an effect that performs its magic on the duration accessor. There are actually two implementations of duration watchers as of this writing: one unsophisticated, poor sound-quality time stretcher effect, and a cheap truncator that truncates the end of the sound in the case of a shorter duration, and inserts extra silence in the case of the user setting the duration to be longer than the original `AudioQuantum`. (One of the benefits of working in open source is making this sort of compromise in public, and having multiple parties raise their hands to attempt to do it better.) A user decides to use a specific effect, and registers it with the `UserAudioQuantum` class:

```
audio.UserAudioQuantum.register(AutoStretch)
```

...and may then modify the `duration` parameter in any `UserAudioQuantum` they encounter. There is an optional parameter in `register()` that allows a user to specify the order within the effects chain with an integer. A single plugin type may exist at multiple points in a chain: an effect (for example,

a reverberation) may have a 'reversed' version by inserting a reverse effect before and after the effect. When the reverse effect is triggered for a given `AudioQuantum`, the sound is forward, but there is reversed reverb. When it comes to `render()`ing the `AudioQuantum`, each `modifier()` that has been registered is then triggered in order with the output of the previous one in the chain. The result of the final effect is returned as the `render()`ed output of the `UserAudioQuantum`.

It is not a particularly safe computational model nor does it make for the most flexible and complete audio network graphs, but the prototype implementation is astoundingly compact: the plugin registration is less than ten lines of code. Effects implementations can be similarly compact. By convention, a `modifier()` method does its own checking of its watched variables. If the variables of interest remain unchanged, then it does nothing and passes the input to the output. For example, changing the level of a `UserAudioQuantum` by entering a positive or negative offset in decibels is simply implemented with:

```
class LevelDB(audio.RenderableAudioObject):
  propertyname = "level_change"

  def __get__(self, instance, owner):
    if hasattr(instance, '_level_change'):
      return instance._level_change
    else:
      return 0.

  def __set__(self, instance, value):
    instance._level_change = value
    instance._modified.add('level_change')

  @staticmethod
  def modifier(aq):
    if aq.level_change == 0.:
      return aq.source
    source = audio.AudioData(
      ndarray = aq.source.data,
      numChannels = aq.source.numChannels,
      sampleRate = aq.source.sampleRate)
    source.data *= pow(10.,
        aq.level_change/20.)
    return source
```

By closing the loop from sound to analysis to modified analytic parameters to novel sounds, we believe the Remix API has access to a powerful sound manipulation paradigm. Although the sound objects are rooted in their original context, users are not limited to their original sounds: effects may radically change their content with minimal effort, but can be targeted down to the briefest tatum, allowing for novel synchronised rhythmic effects.

5.3. XML Output

Somewhat surprisingly, what began as an attempt to have a standard debugging output led to some minor revelations suggesting a life for the Remix API out of the bounds of a single client machine. Just as the analysis is independent of remixing, remixing and audio rendering can be decoupled. This was established with a method added as an experiment, `to_xml()`. As with the `render()` method it mirrors, `AudioQuantum`s are visited in sequence from within an `AudioQuantumList`. The `AudioQuantumList` gets output as a `<sequence>` XML node, with the list contents returned inside it. Complementarily, a `Simultaneous` object is output as a `<parallel>` XML node, making the analogue with SMIL's `<par>` and `<seq>` [14] much more explicit. `AudioQuantum`s are output as XML nodes with start and duration attributes, along with a reference to its source, if it differs from that of its container. Finally, if a user has modified a `UserAudioQuantum`, it is output with its changed parameters and the source `AudioQuantum` contained inside. The order of the nodes in the XML output is exactly the same as the numerical order shown in Figure 4.

A sample XML output from Figure 4 would resemble:

```
<sequence source="blue" duration="10">
  <beat start="10" duration="1"/>
  <beat start="15" duration="1"/>
  <parallel duration="1">
    <beat start="16" duration="1"/>
    <beat start="3" duration="1"
      source="red"/>
    <beat start="42" duration="1"
      source="green"/>
  </parallel>
  <bar start="17" duration="4"/>
  <parallel duration="1">
    <beat start="21" duration="1"/>
    <bar start="4" duration="3"
      source="red"/>
    <tatum start="43" duration="0.25"
      source="green"/>
  </parallel>
  <sequence duration="2">
    <user_modified_audio_quantum
      level_change="-3">
      <beat start="22" duration="1"/>
    </user_modified_audio_quantum>
    <beat start="27" duration="1"/>
  </sequence>
</sequence>
```

In this example, the sources are simplified to colours in the diagram. In practice, they would be set to identifiers for the source audio files. Similarly, for the purpose of the narrow column format for this manuscript, the times are mostly simulated with integers. In actuality, they are floats with enough precision to achieve sample accuracy.

We discuss the ramifications of the independence between remixing and rendering in Subsection 6.1 on future work, below.

5.4. Current architecture

The current class diagram is summarised in Figure 5. Not all classes, methods, or attributes are shown, but it does give a sense of the inheritance, some shared patterns, and how some classes are contained within each others' instances. What had been independent classes are shown to be united by a common superclass, which ends up being a powerful key to giving a uniform interface to any object a user is likely to interact with.

We see from the developments in the library a kind of audio rendering in three dimensions: sequentially in time, in parallel with tracks, and with modifications applied in series to individual `AudioQuantum`s. The prevailing reduction operator for sequential elements is `append()`, while parallel audio elements are

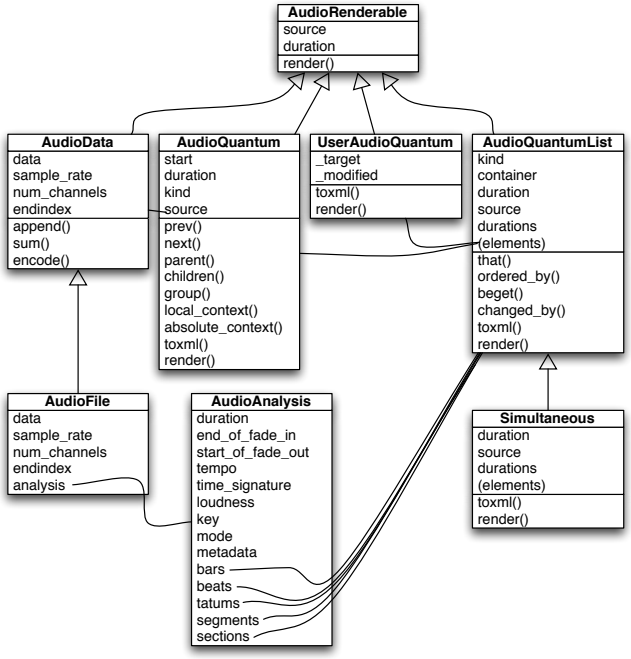


Figure 5: *A partial class diagram showing* essential *attributes, methods, class inheritance and some containment relationships.*

combined with `sum()`. Modifications through successive effects are combined through function composition. Nearly any mixing operation can be modeled by some combination of these fundamental operators.

6. CONCLUSIONS

This paper represents just a snapshot of an *experimental branch* of the Remix API as of mid-April 2009. While it is expected that significant portions of the experimental branch is integrated into the mainline, it is by no means guaranteed. Additionally, while the code examples capture the flavour of the embedded DSL, we cannot warrant that they be up-to-date. The internal implementation code shown is intended to be merely representative of the small scale of code that goes on behind the scenes.

With all of those disclaimers out of the way, we can say that the Remix API is now a highly capable music manipulation *language*. While the API does not accomplish anything profoundly new that could not be achieved with another music programming language – or even Matlab – the art of language design lies in the levels of abstraction afforded to the programmer. The Remix API, in connecting the Echo Nest's Analyze API with sampled audio, offered a powerful concept: automated beat-synchronous editing based on perceptual audio analyses. In taking the concept further through the ideas shown in this paper, the Remix API has become a remixing language where users can interactively, exploratively, or programmatically integrate songs from various sources, expressively filtering through reams of data for the most suitable matches, all while staying locked to the rhythmic clock of their choice.

There are a few users of the Remix API, and it is clear that more have come aboard since the fundamental model has evolved. Nearly all users use at least some of the new methods and classes – particularly the navigation and context ones and the ones involved with the rendering chain, but relatively few have been public about using the terse functional programming-style method chaining. This is no formal means of evaluation, but such user monitoring does offer some feedback on what changes have been generally useful.

6.1. Future work

The Remix API has a broad potential. The Echo Nest has already, from the start, spoken of integrating video manipulation with the existing audio manipulation. Both the Echo Nest programmers and other API users have discussed the possibility of integrating high-quality time stretching external libraries with the Remix API. We hope that the plugin mechanism we describe provides a foundation for integrating those and potentially many other libraries.

Among the issues explored in this paper, there are many areas for further development. The functional programming style discussed throughout uses a lot of iteration. There is room for much improved memory usage on the content-based manipulation side, in which `AudioQuantumLists` are continually and repeatedly scanned. By moving such access to use Python iterators and generators, the functional programming style will be further justified in lowering the performance and memory impact of such operations. Item search within the `AudioQuantumLists` obtained from the analyses currently uses naive scanning through the entire list. Since we know the analysis to be ordered partitions, temporal search (such as that used by all of the rhythmic hierarchy navigation methods) could be vastly improved by considering the assumptions that `AudioQuantums` are ordered and fairly uniformly spaced: a binary or interpolation search would go far.

One of the reasons why the above memory and search time is-

sues have not yet been addressed is that the resource demands they make are generally dwarfed by the demands of manipulating audio data. While moving to a multiple-source rendering paradigm, the audio code was moved from 16-bit integers to a 32-bit format. This, combined with early users' tendencies to want to push the system to its limits, means that inter-cutting a dozen songs can lead to a gigabyte of sample data held in memory before any output is created.

As a result of exploring the XML output discussed previously, we noted that – as the Remix API stood – a complete (and relatively compact) representation of the remixing intent was possible independent of the audio sources. No sample data needs to be preloaded. With regards to alleviating memory usage, the rendering step could load one file, remix the relevant `AudioQuantum`s, and clear its memory before proceeding to another source file. In fact, XML output could conceivably be generated and passed on to another process – or machine – that renders the output and returns a file.

If rendering *is* performed by another process or on another machine, then new manipulation interaction patterns could form. Graphical user interfaces – even on computationally low-powered devices – could concentrate on letting the user directly manipulate the `AudioQuantum`s in the interface, directly reordering beats and tatums at will, and then communicate with a rendering process. Remix libraries could blossom in many different computer languages, being as idiomatic as they wish. (How would a music remixing library look in a LISP dialect, compared with Ruby, or compared with Erlang?) The only requirement would be that they input the XML from the Echo Nest Analyze API, and export rendering directives in an XML dialect resembling that discussed in Subsection 5.3.

These independent rendering processes need not even execute on the same computing resource: they could be distributed. Since audio is transparent, combining the products of `render()`s from independent source files is as trivial as summing across all of the independent renders and normalising upon output. Video has its own techniques for composition, but, as time-based multimedia, it shares much in common with the sequntial/parallel paradigm from audio. This suggests a framework for collaborative, distributed audio-visual remixing, where the basic components of interaction are relatively light XML documents containing metadata or remix directives. Users – or their computational clients – need not handle any multimedia data. All multimedia handling can happen remotely.

6.2. Summary

In open-sourcing the Remix API, the Echo Nest provided a framework for event-synchronous music re-editing backed by their web-based music analysis service. Since then, the re-editing has developed into a content- and context-aware audio remixing environment, capable of not only supporting an imperative coding style, but a terse, expressive, functional and fluent interface. The native API tools for audio manipulation have evolved from being one-dimensional – selection and re-ordering along the time axis – to being three-dimensional, stacking multiple, simultaneous tracks atop one another and then additional, parameter-driven, user-programmable audio effects atop those. The user controls these effects by directly modifying analysis and other parameters on the objects representing `AudioQuantum`s.

Many of the ideas here have been around for a long time, but we hope that in the presentation, their combination and synergies point to something new.

7. ACKNOWLEDGEMENTS

The authors would like to thank the Echo Nest founders, Tristan Jehan and Brian Whitman, and all of the people in their Nest, for their hard work and progressive thinking that led to the open sourcing and active development of the Remix API. We would like to additionally thank them for having the patience and courage to allow some very idiosyncratic changes on a public code repository bearing their company name.

8. REFERENCES

[1] (2009) The Echo Nest Remix API. [Online]. Available: http://code.google.com/p/echo-nest-remix/

[2] A. T. Lindsay, A. P. Parkes, and R. Fitzgerald, "Description-driven context-sensitive effects," in *Proceedings of the Sixth International Conference on Digital Audio Effects (DAFX-03)*, M. Davies, Ed., London, September 2003, pp. 350–353.

[3] N. Collins, "BBCut2: Incorporating beat tracking and on-the-fly event analysis," *Journal of New Music Research*, vol. 35, no. 1, pp. 63–70, 2006.

[4] T. Jehan, "Creating music by listening," PhD Thesis in Media Arts and Sciences, Massachusetts Institute of Technology, Cambridge, MA, September 2005.

[5] The Echo Nest. Analyze API. [Online]. Available: http://the.echonest.com/analyze/

[6] J. A. Bilmes, "Timing is of the essence," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1993.

[7] T. Jehan, "Event-synchronous music analysis/synthesis," in *Proceedings of the 7th International Conference on Digital Audio Effects (DAFx'04)*, Naples, Italy, October 2004.

[8] T. E. Oliphant. (2006, December) Guide to NumPy.

[9] M. Fowler, *Domain Specific Languages*, September 2007, work in progress Implementing an Internal DSL. [Online]. Available: http://martinfowler.com/dslwip/InternalOverview.html

[10] K. D. Smith, J. J. Jewett, S. Montanaro, and A. Baxter, "Decorators for functions and methods," The Python Foundation, Python Enhancement Proposal, June 2003. [Online]. Available: http://www.python.org/dev/peps/pep-0318/

[11] J. McCartney. (2009) SuperCollider. [Online]. Available: http://supercollider.sourceforge.net/

[12] N. Didkovsky and P. Burk. Java music specification language. [Online]. Available: http://www.algomusic.com/jmsl/

[13] B. Schottstaedt. (2009) CLM. [Online]. Available: http://ccrma.stanford.edu/software/clm/

[14] Synchronized Multimedia Working Group, "Synchronized multimedia integration language (SMIL) 1.0 specification," World Wide Web Consortium, W3C Recommendation, June 1998. [Online]. Available: http://www.w3.org/TR/REC-smil/