Project Mashbot: Final Report
An Engine for Dynamic Audio Mashups
Michael DuBois
michael@michaeldubois.me

ABSTRACT

In the music world, a mashup is a fusion of two or more pre-recorded songs. The songs are beat-matched, then selectively combined with audio filters and signal addition. Often, the vocals of one song are mixed over the beats, bass, and melody of another. Project Mashbot represents an attempt to automate this process while maintaining the aesthetic appeal found in mash-ups created by skilled human DJs. This automation will allow users and music artists alike to quickly and easily experiment by combining songs from their music collections on the fly. In this report, the author details the mechanism with which a music mix can be executed. The author then expounds on how this mechanism could be extended to generate mixes automatically.

Categories and Subject Descriptors
H.5.5 [Information Systems]: Sound and music computing

General Terms
Design

Keywords
Mashup

1. BACKGROUND

Mixed music has long been an important part of our culture. DJ Francis Grasso was the first to start beatmatching songs and playing a continuous mix of music in 1969. He invented a technique called "slip-cueing," where the DJ holds a vinyl record still and allows the turntable to spin beneath it. This allowed him to immediately start a new track right on the beat. Armed with capability to keep a continuous mix going, Grasso was able to maintain the energy of the dance floor longer than any other DJs at the time. According to the New York Times [4]:

One of his best-known tactics was his simultaneous playing of percussion-heavy songs, like Chicago Transit Authority’s “I’m a Man,” with sexually charged vocals, like Led Zeppelin’s “Whole Lotta Love,” anticipating the rise of disco.

Grasso, thus, began a new cultural tradition, a self-reflective art form that combines and juxtaposes distinct ideas. The tradition has lived on and expanded greatly with the rise of the laptop and cloud-based music platforms like Soundcloud.

The idea for Project Mashbot was first conceived in 2010 when we began to question why popular music players, like iTune, could not, themselves, play continuous music. After all, automatically beat-matching music was definitely not an unsolved problem. Many DJ suites have performed the task quite well for years. Moreover, the demand for a widely available application of this technology would surely be as great as the enthusiasm for Grasso’s innovation in the DJing world.

In the fall of 2013, we finally decided to investigate the plausibility of such an endeavour. We began, as Grasso did, by learning how to beat-match songs. Before long we encountered a project called Dancing Monkeys, started by Karl O’Keeffe during his masters thesis at Imperial College London. The project analyzes song structures to generate unique ”step files” for a clone of the popular game Dance Dance Revolution. O’Keeffe endeavoured to generate dance steps that would both match the rhythmic structure of the song and vary in intensity in parallel to the intensity of the song. To complete his project, he compiled a comprehensive summary of all the research regarding the analysis of song structure that was available in 2005 [3].

The most promising of the techniques described by O’Keeffe was one described as The Beat Spectrum[1]. In this technique, the energy of a song at every sample is mapped in a self-similarity matrix. From this self-similarity mapping, it becomes possible to detect patterns, such as the regular repitition of a beat, or even the repition of the chorus.

The next piece of the puzzle came into place when we discovered Tristan Jehan’s 2005 doctoral thesis at the Massachusetts Institute of Technology[2]. Jehan extended the ideas put forth in The Beat Spectrum to detect the position of a song’s sections, the position of individual sounds or notes (termed "segments"), and the position of beats. Later
that year, Jehan founded The Echonest, a company that now provides all of this data (and more) for a database of 35,573,753 songs. In March 2014, The Echonest was acquired by Spotify.

2. IMPLEMENTATION

2.1 Engine Overview

At its absolute simplest, the Mashbot Engine can be divided into two abstract concepts — the marionette and the manipulator. The marionette constitutes all of the logic required to represent and manipulate the state of a mix, whereas the manipulator constitutes the "intelligent" logic that decides how the marionette’s mechanism should be used to play a pleasing mashup. This report will focus on the marionette, since this component has been the primary focus of our implementation efforts in the past four months.

With that out of the way, the implementation of the Mashbot Engine can be broken down into three areas of concern — those of the model-view-controller (MVC) pattern. We will see that the marionette — because it is the primary mechanism of the engine — must be split across all three of these areas of concern, whereas the manipulator can be wholly contained within the model.

The "view," in our case, is not tied to a graphical user interface, as is customary, but rather to an audio user interface — the mixed audio output streamed to the user’s speakers. By way of the controller interface between the model and the view, we are able to sever the model from the host application’s implementation of this audio output. Our model then contains the most portable, core business logic — in this case, for creating and representing a mix – our controller contains the second-most portable logic — that for executing a mix — and the view contains the unportable, host implementation details that realize the execution. We will now discuss how this is accomplished, beginning with the view.

2.2 The View: Churning the Gears

As in many MVC applications, in the Mashbot Engine, it is the view that instigates state changes. The primary action of the Mashbot Engine is driven by a series of render callbacks from the host application. These render callbacks will usually implement the double buffering pattern so that each callback serves to fill and queue a buffer of audio samples for output by the host system. Hence, each callback advances the mix by BUFFER_SIZE samples.

Due to the nature of double buffering, the system will always be processing one buffer while it requests the next buffer from the engine. Therefore, the engine state will always be processing one buffer while it requests the next buffer ahead of the output state by up to BUFFER_SIZE from the engine. Therefore, the engine state will always be processing one buffer while it requests the next buffer. Due to the nature of double buffering, the system will always be processing one buffer while it requests the next buffer. Hence, each callback advances the mix by BUFFER_SIZE samples.

The "view," in our case, is not tied to a graphical user interface, as is customary, but rather to an audio user interface — the mixed audio output streamed to the user’s speakers. By way of the controller interface between the model and the view, we are able to sever the model from the host application’s implementation of this audio output. Our model then contains the most portable, core business logic — in this case, for creating and representing a mix – our controller contains the second-most portable logic — that for executing a mix — and the view contains the unportable, host implementation details that realize the execution. We will now discuss how this is accomplished, beginning with the view.

Each render callback is an opportunity to take the following actions:

1. Realize commands that reconfigure the audio processing graph
2. Queue command results
3. Process and queue a block of audio samples for the host
4. Queue an updated mix timestamp

2.3 The Model: Representing a Mix

Any host implementation that supports low-latency audio will make render callbacks on a high-priority thread to ensure precise timing. If this high priority thread is forced to wait on a resource lock held by a lower priority thread (like the GUI thread or our main controller thread), its high priority scheduling will count for nothing (priority inversion) and we may end up underrunning the audio output buffer (which results in unpleasant pops and clicks). Therefore, it is essential that we use lockfree mechanisms to communicate with the rendering thread. We have selected the boost::spsc_queue for this purpose. It is a single-producer/single-consumer (SPSC) queue backed by a circular buffer. Moreover, it is essential that we do not allocate memory on the rendering thread, since memory allocation is not guaranteed to be lockfree.

Render callbacks depend on two thread boundaries where we make use of SPSC queues.

The first boundary is the render-decoder boundary, where the render thread acquires decoded audio samples from each active audio stream’s SPSC queue. Once the samples from each stream are acquired, they must be processed individually, combined, and then processed as a composite. This allows us to apply effects (aka FX) to both individual streams and the mixed output. The effects applied at each stage can be as simple as volume control or as complex as reverb. We accomplish this by arranging FX units in an audio processing graph. This graph has one head (or sink) node from which a render callback can request blocks of processed samples. Our graph has a single MixNode that combines input from a variable number of stream subgraphs. These subgraphs, in turn, draw samples from the aforementioned stream SPSC queues. The main processing graph, therefore, represents the primary configuration of our audio output "view."

The second boundary is the render-controller boundary, which is composed of three SPSC queues — one for incoming control commands, one for outgoing results, and one for outgoing mix timestamps. As we will see later, the data passed through these queues is sufficient to keep the model synchronized with the current (realized) state of the view.
The key to understanding the Mashbot Engine’s Model is recognizing that a mix is simply a sequence of state transitions that occur at specific times relative to the start of the mix. For instance, at the beginning of a mix, one song might be added. Then, later, a low pass filter might be added to the song and another song might be added. Each of these events changes the state of the mix and the corresponding state of our “view.” The MixState object, thus, serves to encapsulate the means by which the engine seeks forwards and backwards within this sequence as audio playback proceeds ever onward.

We have decided that it is important that the Engine be able to seek backwards as well as forwards for two reasons. Firstly, there must be some way to reset a mix to its starting point so that a user can play the entire mix again after it has finished. Secondly, we believe users will enjoy the ability to rewind and listen to a particularly good song transition a second, third, or even fourth time. This highlights a particular advantage of our model. It will be trivial for future versions of the Mashbot Engine to provide methods for skipping back and forward between “points of interest” in the mix (i.e. state transitions). If a user does not like a particular song combination, they will be able to skip to the next combination. If they enjoyed the last combination, they can listen to it again.

As we have seen, the rendering thread realizes the mix state transitions by interpreting Command objects. However, we do not want to represent a mix as a series of explicit commands, since this would unnecessarily couple the mix with the implementation. Therefore, to represent a mix, we use a series of Event objects which the controller converts into explicit Command objects that the rendering thread can realize. Event objects, thus, encapsulate only the absolute time and intention of a state transition, whereas Command objects also provide the allocated resources necessary to realize the transition. However, as you will see, we still use the same concepts from the command pattern to manage these Events.

It’s worth noting that a state in the mix is composed of several substates that are not necessarily exited in the same order that they are entered. For instance, audio streams might be added to the mix in the order ABC, then removed from the mix in the order BAC.

2.3.1 Events
Since we have decided to allow the engine to seek both forwards and backwards, we must provide a means for the MixState to recover substates by either their start or their end, depending on the seek direction. As such, Event objects always come in pairs. One mate in the pair opens the substate and one mate in the pair closes the substate.

We have determined that the following Event types are sufficient to represent most rudimentary mixes:

- **STREAM_LOAD / STREAM_UNLOAD**
  - Allocates/deallocates stream object that loads stream data from a local or (potentially) a remote uri.
- **STREAM_ATTACH / STREAM_DETACH**
  - Attaches/detaches a stream to the mixing node so that it is added to the mixed output.
- **STREAM_SEEK*/ / STREAM_UNSEEK**
  - Seeks a stream to a given point, or, if we’re moving backwards in the mix, “unseeks” it from the next STREAM_SEEK pair to this STREAM_SEEK pair’s starting point plus the appropriate offset.
- **FX_ADD / FX_REMOVE**
  - Adds/removes an effect to/from a stream.
- **FX_PRESET_ENABLE*/ / FX_PRESET_DISABLE**
  - Enables/disables an FX preset with an optional easing transition.

*Not yet fully supported.

2.3.2 Scopes
Armed with our Event types, all that remains is to break the MixState down into temporal scopes and provide some logic for how Event’s move from one scope to the next. It is immediately clear that a mix requires at least three scopes: a past, present, and future.

The present scope must contain all the Events that opened the currently active substates. Maintaining this scope allows us to easily reinitialize a MixState that has been serialized between sessions or hibernated for the purposes of releasing memory while playback is paused.

The past and future scopes are conceptually similar to the undo and redo stacks commonly used in the command pattern. They differ, however, in that the Events (and their corresponding Commands) come in pairs that open and close resources. Therefore, the past scope will store pairs of events sorted by their closing event for easy retrieval. When the MixState seeks backwards, it must simply pull pairs out of the past, drop their open Event in the present and move their close Event onward to the future.

However, since the controller and the view must interact asynchronously through the command/result/timestamp queues, the controller must remain ahead of the view so that commands relevant to the current moment are always in the queue when the render callback needs to consume them. As such, we have further divided the future scope into the future and horizon. The Events in the horizon scope are pending realization. That is, they have been issued to the view, but the view has not yet realized them and advanced the mix timestamp beyond them. For the purposes of serializing a MixState object, the horizon scope is considered part of the future — since its Events are yet unrealized.

Initially, we tried to incorporate all of this scope functionality directly in the MixState object. However, this quickly became unwieldy, as the present scope behaves differently
than all the other scopes. As such, we encapsulated this logic into an abstract Scope interface and some associated concrete implementations.

Scope. At its simplest, a Scope is an item container with two sides. Each side has a timestamp corresponding to the most shallow item on that side — or, if the Scope is empty, the last item to pass through that side of the Scope. Each side of the Scope can then be popped to a given timestamp, meaning that all items shallower than the given timestamp will be popped off that side of the Scope. The Scope objects can then be arranged in a bidirectional network so that items popped from the left or right side of one Scope are automatically pushed to the corresponding side of the adjacent Scopes. Since some Scopes may convert items from one side to differently typed items on the opposite side, each Scope is required to implement two interfaces — ScopeLeft and ScopeRight — that handle the corresponding types. This allows the adjacent Scope to only keep a reference to a ScopeRight<PrevType> previous and a ScopeLeft<NextType> next without knowing what items the adjacent Scopes deal in on their far sides.

To implement the MixState, we require two concrete implementations of a Scope.

SimpleScope. The SimpleScope pushes and pops the same type of object from both the left and the right sides. Thus, it simply wraps a deque with Scope functionality. This Scope type is sufficient for all but the present scope. The past scope reuses this functionality by bundling Events in an EventBundle struct — a type that wraps two events and forwards timestamp requests to the closing event so that ordering can be performed relative to the closing event’s timestamp.

LeftBundlingScope. The LeftBundlingScope is more complicated. As the name suggests, the LeftBundlingScope pushes and pops bundled items from its left facing but pushes and pops individual items from its right facing. This, of course, suits the present scope’s functionality. Internally, the LeftBundlingScope stores close events (temporarily) in a deque and stores open events in a hash map. The map hashes items by their targetId (in our case, a unique stream id), then further hashes them by their type (in our case, the Event types described above). Within these final hashed bins, the items are stored in arrays that are sorted by item timestamp. This organization allows the LeftBundlingScope to quickly find the matching open event for a given close event when the close event is bundled and popped from the LeftBundlingScope’s left facing. However, it also allows the LeftBundlingScope to pop individual events normally from the right facing.

With these scopes, a MixState::seekTo(targetTimestamp, horizonTimestamp) operation becomes as simple as popping the sides of each scope with the correct timestamp:

```c++
// We capture “pending” events popped from the future to the horizon so we can issue them unique_ptr<ConstEventDeque> pending;

// If we’re headed into the future
if(targetTimestamp > mPresentTimestamp) {
  pending = mFutureScope->popLeft(horizonTimestamp, shouldCapturePending);
  mHorizonScope->popLeft(targetTimestamp);
  // present only popLeft’s closing events
  mPresentScope->popLeft(targetTimestamp);
  // If we’re headed into the past
  mPastScope->popRight(targetTimestamp);
  mPresentScope->popRight(targetTimestamp);
  mHorizonScope->popRight(horizonTimestamp);
}
```

2.4 The Controller: Pulling the Strings
All that remains now is to translate the state of the model into the state of the view — that is, we must tie strings between the puppet and the control bar, in our marionette analogy. We have seen that the view consumes commands from SPSC queues and we have seen that the MixState is a sequence of state transitions represented by Event pairs. Recall, however, that the rendering thread cannot allocate memory in a lockfree manner. Therefore, the controller responsibilities are as follows:

1. Play or pause render callbacks at user’s request
2. Consume timestamps from rendering thread
3. Advance the MixState accordingly
4. Allocate resources necessary to realize an Event
5. Package Event and resources in Command object
6. Issue the Command
7. Monitor results for errors.

2.5 Putting the Bot in Mashbot
You may now be wondering how the manipulator fits into all of this. This component maps directly to the Bot seen in Figure A1. The Bot is responsible for generating the future of a mix whenever the playlist is altered. In the current implementation, only the future of a mix can be changed.

The current implementation has only one Bot named DemoBot. DemoBot is a simple Bot that generates a hard-coded mix in which each song plays for two minutes with a 40 second overlap. During this overlap, a Lo-pass filter is applied to the first song and a Hi-pass filter is applied to the second song. This is a naive implementation — a stop-gap given that we have not yet had time to integrate the song structure data provided by the Echonest API.

In the future, we plan to make Bots more intelligent by allowing them to compare song structures. Like all hard problems

5A Lo-pass filter allows frequencies lower than the cutoff frequency to “pass” through. The rest are tapered off to zero silence.
6A Hi-pass filter, naturally, does the opposite.
in computer science, selecting the optimal transition point between two songs is a search problem. It can be broken into a hierarchy, as follows:

\[
\text{Song} \rightarrow \text{Section} \rightarrow \text{Measure} \rightarrow \text{Beat}
\]

The simplest search would simply select the first option available from the pool in each hierarchy. Thus, a simple search would select the first beat of the first measure of the first section of the next song on the playlist. A slightly more complex search might select the most characteristic section (i.e. the chorus). A still more complex search might select a measure that does or does not have vocals.

With this in mind, we expect a Bot to be a collection of Strategy algorithms. Each Strategy would perform a search that optimizes different features to determine the “best” transition point between two (or more) songs. We expect each Strategy assigned to a Bot would have a different priority (and perhaps different parameters) during each section of the mix (Intro/Beginning/Middle/End/Outro). Our hope is that this would allow longer mixes to begin slowly, build in energy throughout, and then end with a denouement. However, as previously stated, this remains an area for future work.

3. EVALUATION

Without beat-matching, the mixes produced by this engine are hardly satisfactory. However, the mechanism underlying the mixes lays a solid foundation for future development.

Our implementation sports the following advantages:

1. MixState supports seeking both backwards and forwards between “points of interest”
2. MixState allows the future of the mix to remain flexible until commands are issued to the rendering thread.
   (a) As such, users can manipulate the playlist or change the bot on the fly.
3. Implement beatmatching
4. Basic Strategies for Bots

And the following disadvantages:

1. Current implementation lacks beat-matching
2. Bots are largely unimplemented

For a demonstration of the Engine’s command system in action, see Appendix B.1.

4. DISCUSSION

To our knowledge, no other project has modelled a mix in such a flexible manner. All of the “auto-djing” apps currently available on the Android and iOS app stores are limited to beat-matching and mixing the ends of songs. It seems likely that they do not utilize song structure data, like that provided by The Echonest API. Moreover, many DJing apps are not even geared towards full automation of the DJing process. Instead, they provide automated beat matching and allow the user to create playlists and add effects. While Project Mashbot may, one-day, allow users to directly adjust the mixes generated by the Mashbot Engine, our system has the potential to allow for both automated and manual manipulation. As a result, the end product promises to appeal to a wider audience — listeners in addition to aspiring DJs. Furthermore, our MixState allows for a general and conveniently serializable representation of mixes, that — when combined with common music libraries like Spotify’s — can be easily shared. While some expensive desktop DJ suites undoubtedly model and save mixes built from local data, no one has yet leveraged this capability in the highly social sphere of mobile and cloud-based applications. It is true that the current iteration of the Mashbot Engine cannot yet beat-match and mix songs as other can. However, it’s potential is grand and the current Engine presents a solid foundation for future work.

5. FUTURE WORK

The prototype described here is but the first step on a long journey towards the fully automated mashup engine.

The following is a roadmap of features that would complete a rudimentary proof of concept:

1. Setup unit tests and a better debugging environment
2. Integrate Echonest API for song data
3. Implement beatmatching
4. Basic Strategies for Bots

And the following is a roadmap of extensions that would complete our full vision for the project:

1. Implement more filters or port to iOS
2. Advanced Strategies for Bots
   (a) These would require more sophisticated analysis of song structure and even song lyrics
3. Integrate with streaming APIs (Spotify/Beats/Rdio/etc)
4. User accounts and mix sharing
5. Playlist generation

Naturally, the pursuit of mashup Strategies is an endless pursuit with many opportunities for novel artificial intelligence research. Similarly, since The Echonest was recently purchased by Spotify, it is possible that a future version of Project Mashbot will require in-house song analysis. However, it seems clear that the next step for Project Mashbot should be to complete a rudimentary proof-of-concept with the Echonest data to better attract the attention of investors and/or software developers interested in our vision. We believe this prototype is more than half-way to a prototype that can be pitched.
6. CONCLUSIONS
The dream of automated mashup engine is not an impossible one. It is, however a complicated undertaking. With concerted effort, we expect we will be able to accomplish a working prototype in the near future. In the meantime, we have conquered many difficult design decisions and produced a flexible, serializable representation of a mix.

7. REFERENCES

APPENDIX
A. TABLES & FIGURES
A.1 The Mashbot Engine
B. CODE LISTINGS
B.1 Log Output From a DemoBot Mix
Figure 1: An overview of the Mashbot Engine Design.
05-15 23:33:51.120: D/Commands(7908): + STREAM_LOAD at (8792064, 8828928) : /mnt/sdcard/Music/02 Aerodynamic.mp3