DATA JOCKEY, A TOOL FOR META-DATA ENHANCED DIGITAL DJING AND ACTIVE LISTENING

Alex Norman and Xavier Amatriain
University of California, Santa Barbara
Media Arts and Technology

ABSTRACT

Data Jockey is a free and open-source *digital DJ* software system that allows users to mix several audio works based on content features, such as average brightness and average harmonicity, while syncing the works to a common clock. This software allows users to search through an audio database using audio content *descriptors* as search criteria.

These features help users explore their ever-expanding digital audio collections in new ways. Providing users with access to content *descriptors* will provoke interesting juxtapositions of different works and the evolution of content features over mixes. Access to these *descriptors* will help users find new meaning in these audio works and their combination.

1. INTRODUCTION

The increasing capacity and decreasing cost of digital storage make it easier than ever to amass large collections of digital audio works. While some people may be satisfied with the passive entertainment their media libraries provide, others seek more involved ways to interact with these collections.

One such option is to use these works as material for new creative expression, mixing works together and juxtaposing them to create something new, just as a disc jockey does with phonograph records. This is the basic function of Data Jockey, a free and open-source software (FOSS) system that allows users to interact with pre-recorded audio works based on their content and mix them while locking the playback tempo of each piece to a common clock. This is commonly known as *digital DJing*. While there are other software tools that allow users to *DJ*, none focus as heavily on search and comparison between audio works using audio *descriptors*.

By providing its users with access to content descriptors, Data Jockey encourages them to search for interesting combinations of audio works that they might not have found otherwise. It also allows users to quickly search in a large collection of audio recordings for works that fit a specific criterion, encouraging exploration and providing ways to create more meaningful *playlists*. We expect that access to content descriptors will help users learn about what it is, specifically, that they like about these works and their combination, and, in turn, create more interesting mixes.

1.1. Related Tools

A popular example of a digital DJ system that has contributed to the inspiration for Data Jockey is Ableton's *Live* [1]. Live is a commercial software product that allows users to mix pre-recorded audio. While Live is a very powerful performance and composition tool, it lacks the rich database central to Data Jockey's feature set. Live analyzes audio works for beat information but does not provide any additional feature analysis data for users; Data Jockey extracts beat information as well as other content feature information like "average brightness" and "average harmonicity," which it then associates with works through content descriptors.

In addition to providing a more powerful content management and retrieval system than Live, Data Jockey's users have more freedom in their work since they can modify the system as much as they please because Data Jockey is a FOSS system.

One example of a FOSS *digital DJ* system is Mixxx. Mixxx provides a virtual disc jockey interface that is much like a traditional disk jockey setup. It has automatic *tempo matching* capabilities but also allows users to *beat match* works manually [7].

There are other systems which attempt to mimic the traditional disc jockey experience through haptic interfaces. One example is D'Groove [4], a system which gives users a physical interface that employs an actual phonograph record, and a software system that helps to *beat match* and mix digital audio files. These systems essentially allow users to replace their record collection with a library of digital music while attempting to retain the historical interface and experience. While Data Jockey would benefit from a haptic interface of some sort, it has a more hands-off approach to audio *beat matching*, allowing users to focus on exploring new territory that would not be possible otherwise.

2. SYSTEM OVERVIEW

Data Jockey consists of a database of audio and meta-data for that audio, scripts to import data into the database, and a graphical performance interface written using a combination of the Ruby programming language and C++. It is being developed using Linux but with cross-platform compatibility in mind.

2.1. Database

Data Jockey's database provides a simple way of organizing and retrieving data from a large collection of audio works and audio meta-data. It also provides the ability to relate included audio works in a number of ways. It organizes works by title, artist, and album and also allows works to be associated with "tags" and "descriptors," which provide data to use in content comparisons as well as additional avenues for content retrieval.

2.1.1. Tags

Tags are *typed* labels that can be applied to works. Some tag types are "genre," "lyrical theme" and "mood". As an example, a work might be tagged with a "genre" tag that has the value "punk." Tag data can be extracted from audio file meta-data, like the ID3 tags in MP3 files, or can be manually added by a user. Users can also add their own tag types. These tags give users ways to associate multiple works and search for content beyond specifying title, artist or album. Extensible tags give users a way to organize their content using whichever label types they find useful.

2.1.2. Descriptors

Descriptors are much like tags but, instead of text, they have numerical values. Descriptors also expand the search space available to Data Jockey users, helping them explore their music library in ways they would not have otherwise. Descriptor values generally come from analysis of the content which they describe but can also be added manually. Instead of simply finding works by name, artist or album, descriptors allow users to search for works based on features of the works themselves.

The three descriptors currently provided are "average brightness," "median tempo" and "average harmonicity;" these can be input by the user manually or can be computed through audio feature analysis. As is the case with tags, users can add their own descriptor types.

These descriptors can be used in a number of ways. For instance, a user might decide that over time their play-list should get brighter yet decrease in tempo and should prefer tracks that have a relatively high harmonicity value. A search that follows this user-defined *envelope* would trace a path through works that are brighter than the tracks currently playing, that are above a specified harmonicity threshold and are slower in tempo than the current global tempo value.

This envelope could be constrained further by including tag data as well, for instance only allowing "techno" songs to be chosen for the mix. These user-defined constraints define a path to search through the database and allow for high-level envelopes on the features of a mix.

This *path* is similar to those created through *automatic play-list generation* attempts, as described in [3] and [8]. These *play-list generators* generally keep meta-data out of view of users, providing high level controls like percentage of songs using brass instruments, or simply tracking user's habits and generating play-lists based on those habits [3] [8]. In contrast to these *automatic play-list generators*, users of Data Jockey have access to all of the descriptors, which they can aggregate themselves in order to create their own high-level controls.

3. TEMPO TRACKING AND SYNCHRONIZATION

The fundamental feature that Data Jockey requires of its audio data is beat location information. With this beat location information Data Jockey can sync audio works to a clock and, in turn, play works in sync with each other.

3.1. Tempo Synchronous Playback

Synchronizing audio works to a common clock while mixing is often desirable and is a main feature of Data Jockey. Data Jockey syncs songs using their pre-calculated beat location information. Each playback synthesizer has both a buffer of audio data and a buffer of beat location timepoints. The synthesizers read through their beat location buffers at a common rate given by the tempo controller, synchronizing their beat locations to the pulse of the tempo controller and interpolating values in-between pulses using an audio-rate ramp provided by the tempo controller.

The values read from the beat location buffers are used as time pointers for reading data out of the audio buffers, which is eventually sent to the output to be played. While the beat buffer information is read at a common rate, the data that is retrieved from the beat location buffers is unique to its associated audio work. The result is that each audio buffer is played back at a rate which synchronizes its beat locations to the pulses of the tempo controller. If the tempo controller's pulse output rate is not the same as an audio work's original tempo the pitch of that work will be scaled as a side effect of scaling its tempo.

While it would be nice to separate the pitch and playback tempo, the current implementation is quite usable. In fact, coupling playback tempo and pitch might be desirable for some as it is inherent in mixing phonograph records. An approach to this decoupling is discussed in the "Future Work" section.

3.2. Tempo Extraction

Data Jockey uses BeatRoot, a beat tracking software application, to extract the beat location information from audio recordings. BeatRoot, winner of the MIREX 2006 audio beat tracking competition, analyzes audio recordings and returns the beat location information in the form of a text file with beat times indicated in seconds. BeatRoot's inner workings are described in detail in [5].

While the data extracted by BeatRoot has proven to be reliable for most songs, there is some jitter in the distances between beats. This jitter is especially apparent with Data Jockey because Data Jockey's current time stretching technique changes the pitch of audio works, so the jitter results in unnatural pitch changes during playback. In order to combat this jitter, Data Jockey applies a moving average filter to the distances between beats. This filtering operation removes these pitch change side-effects during sections of audio that have stable tempos.

4. PERFORMANCE INTERFACE

The performance interface consists of a graphical user interface (GUI) and an audio playback back-end written in C++. The GUI uses the Gimp Toolkit (Gtk+), a cross platform GUI toolkit.

4.1. GUI

The GUI is written in Ruby and is connected to an audio playback back-end written in C++ through a SWIG interface. The SWIG interface provides the "glue" needed to connect these two components. SWIG is a tool that automatically generates interfaces between C/C++ programs and a number of other, generally higher level, programming languages [9]. The use of two programming languages, while increasing the complexity of the design, allows Data Jockey to take advantage of both the simplicity of database and GUI programming with Ruby and C++'s speed, which is needed for "real-time" audio programming. The GUI is made up of a *mixer view* and a *database view*.

The mixer view is similar to a standard hardware audio mixer, consisting of several *mixer channels* and one master volume control. It also includes controls for manipulating the global playback tempo. Each mixer channel has controls to manipulate the volume, equalization, output destination and playback position of an associated audio clip (usually a full song). Text above each mixer channel indicates which audio clip the channel is currently associated with (see Figure 1).

The database view provides users with a view into their library of audio and audio meta-data. This view also provides users with the ability to select works to load into a mixer channel, including them in the current mix. Users can use this view to browse through their library; they can sort by artist, album, title or by one of the descriptor values (see Figure 1). The more complicated search constraints discussed in the database section are achieved here through an "advanced search" dialog box.

4.2. Audio back-end

The audio playback back-end is written in C++ because Ruby is not suited for "real-time" audio programming.

The audio back-end uses JACK for audio output. JACK is an audio application programming interface (API) which provides applications with low-latency audio input and output, and also gives users the ability to easily route audio between applications themselves [6]. By making Data Jockey a JACK client, we allow users to use other applications to affect or record the results of their performance.

The playback system consists of three main components: a tempo controller, a collection of audio playback synthesizers, and a master volume control. The tempo controller sends out *triggers* at regular intervals (an impulse train) and an audio rate *ramp* that the playback synthesizers use to synchronize their audio playback.

Each playback synthesizer has an associated audio buffer as well as a buffer that includes the locations of the beats. The synthesizers use their beat location buffer in order to synchronize the playback of their associated audio buffers with the tempo controller. The synthesizers accept commands which allow the user to scale the volume of the audio output, reset the playback position, select which output (master or cue) to send the audio to and set the equalization of the audio. If the audio is sent to the master output it is combined with all the other synthesizers' master outputs and scaled by a master volume, then it is finally sent to the sound-card. Otherwise, the audio is summed with all of the synthesizers' cue outputs and sent to a different audio output which is intended to be used to preview the result of adding elements to a mix without sending them to the main output.

In addition to synchronizing audio within Data Jockey, data from the tempo controller is sent to a JACK audio output. Other applications can use this data to sync themselves with Data Jockey.

A nice side effect of using SWIG is that the audio back-end could be used with other programming languages like Python or Ocaml. Users might decide to use the audio back-end exclusively within a non-graphical scripting interface, or with a custom graphical interface of their own.

5. CONCLUSIONS AND FUTURE WORK

Already in its early stages of development, Data Jockey is quite useful. Its use of content "tags" and "descriptors" sets it apart from existing "digital DJ" applications and allows users to explore their music library in new ways. We envision access to this content meta-data encouraging users to approach their own work in new ways as well, comparing and contrasting source content and using the results of these comparisons to create mixes that are dynamic in structure. We also hope that through working with this meta-data, users will be able to qualify what it is about a collection of works that makes them *mixable*.

Many additions are planned for Data Jockey including: audio and audio meta-data visualization and manipulation capabilities, performance recording, improved search capabilities, audio segment looping, time stretching decoupled from pitch scaling and additional audio feature extraction analysis.

There are many feature extraction techniques that may be useful for exploration within Data Jockey. First and foremost, phase vocoder analysis accompanied by onset

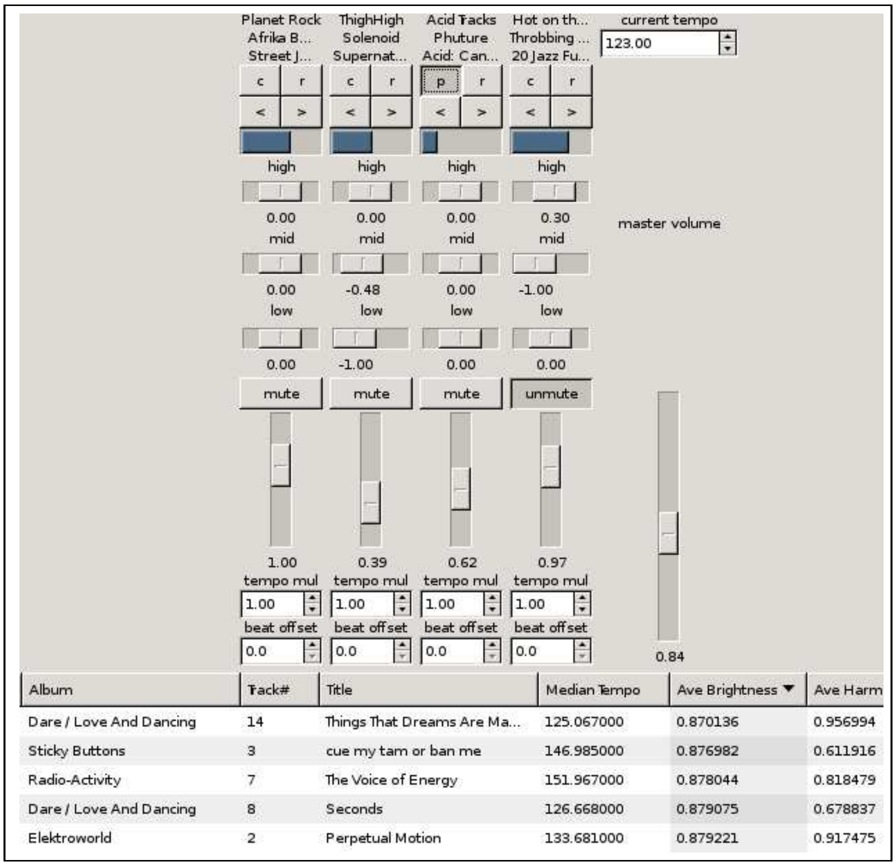

Figure 1. Data Jockey Performance Interface

detection would allow for time stretching to be decoupled from pitch scaling during playback as discussed in [2]. Pitch information combined with this separation of pitch and tempo would allow users to tune works to a common *key*, thus *key matching* works in addition to *beat matching* them. Spectral information would allow users to search for works which could fill the spectral gaps present in their current mix, to make the output fit a desired spectral envelope. Analyzing the space that the combination of these extracted features creates could help segment the audio into sections, like introduction, verse, etc., as well as providing new search criteria. Finally this combined information, as well as information about a user's habits in mixing, derived through performance recording, could help expose what about audio works makes them mix well.

6. REFERENCES

[1] Ableton Homepage. *http://www.ableton.com/*, 1 May 2007.

[2] X. Amatriain, J. Bonada, A.Loscos, and X. Serra, "Spectral Processing", Udo Zlzer (Ed.), *DAFX: Digital Audio Effects*, John Wiley & Sons, Inc., New York, NY USA, 2002, pp 429-435.

[3] A. Andric, G. Haus, "Automatic playlist generation based on tracking user's listening habits", *Multimedia Tools and Applications*, Volume 29, Issue 2, Kluwer Academic Publishers, Hingham, MA USA, 2006, pp. 127-151.

[4] T. Beamish, K. Maclean, S. Fels, "Manipulating music: multimodal interaction for DJs", *Proceedings of the SIGCHI conference on Human factors in computing systems*, Vienna, Austria, 2004, pp. 327-334.

[5] S. Dixon, "An Interactive Beat Tracking and Visualisation System", *Proceedings of the International Computer Music Conference*, Havana, Cuba, 2001, pp 215-218.

[6] JACK Homepage *http://jackaudio.org/*, 29 June 2007.

[7] Mixxx Homepage *http://mixxx.sourceforge.net*, 29 June 2007.

[8] F. Pachet, P. Roy, D. Cazaly, "A Combinatorial Approach to Content-Based Music Selection", *IEEE MultiMedia*, Volume 7 , Issue 1, IEEE Computer Society Press, Los Alamitos, CA USA, 2000, pp. 44-51.

[9] Swig Homepage. *http://www.swig.org/*, 29 June 2007.