Lossless Editing of Lossy-compressed Audio Data

Joshua Haberman

January 8, 2004

MP3 and other popular compression formats are *lossy*, meaning that the compressed representation cannot reconstruct the original data exactly. Once a file is compressed in a lossy format, it cannot be manipulated directly in the same way that uncompressed data is. For this reason, audio editors generally deal with lossy formats by uncompressing the data on import and manipulating the uncompressed data. If the user chooses to export to a lossy format, the editor must recompress the data which causes another generation of loss. This makes it undesirable to edit data once it has been compressed in a lossy format, since each iteration of import/modify/export will cause the sound quality to degrade.

Though lossy formats cannot be modified sample-by-sample as uncompressed data can be, it is often possible to rearrange the compressed data to the extent that arbitrary cut and paste editing can be accomplished without uncompressing and recompressing. A few special-purpose programs can perform this kind of manipulation on MP3 files, but this capability is rarely found in general purpose audio editors.

In this paper I discuss my endeavor to give the Audacity audio editor lossless editing capabilities. Though most of the concepts and implementation apply to any compressed format, in this project I use the Ogg Vorbis format, a patent-free alternative to MP3.

1 Introducing Audacity

Audacity is a freely available digital audio editor released under the GNU General Public License. It is cross-platform, running on Windows, Mac OS X, and Linux. It can be obtained from <http://audacity.sourceforge.net>.

Audacity has a diverse set of features, but most of them fall into broad categories: reading and writing sound files, performing basic editing such as cut/copy/paste, applying effects, recording and playing with a sound card, and multi-track mixing. Audacity is also known for its intuitive interface: its broad feature set doesn't get in the way of simple operations.

Audacity is a disk-based editor, meaning that the data being edited is always saved to disk and read into memory only as needed. This allows Audacity to work with files that are much larger than the physical memory available. The on-disk data storage scheme is unique to Audacity, and explained in detail in Dominic Mazzoni and Roger Dannenberg's paper *A Fast Data Structure for Disk-Based Audio Editing.* [1] Briefly, every track of audio is broken into blocks of approximately 2MB, each of which is stored in a separate file; these files are referred to as "blocks" or "BlockFiles" (the latter is the name of the class that implements them in the C++ source code). BlockFiles can be referred to by more than one track and are reference counted, so that copying data from one track to another only creates more references to the existing blocks.

Tracks holding sound data are implemented by the `WaveTrack` class. WaveTrack provides methods for high-level editing such as `WaveTrack::Copy()` and `WaveTrack::Paste()` as well as accessors for getting and setting individual samples. There is one intermediate class between WaveTrack and BlockFile

[1] Dominic Mazzoni and Roger B. Dannenberg. "A Fast Data Structure for Disk-Based Audio Editing." Computer Music Journal, Volume 26, No. 2, 2002, pp. 62-76.

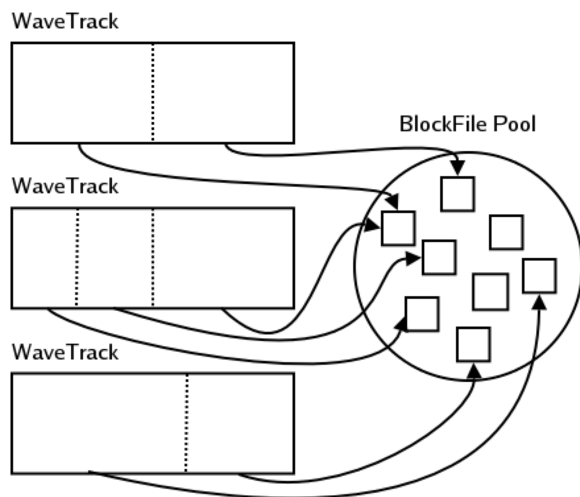

Figure 1: Audacity's data representation scheme

that performs the manipulations on BlockFiles: the `Sequence` class acts as as a vector of samples that provides its storage using BlockFiles. Each WaveTrack contains a Sequence. It is the methods of Sequence that implement copy and paste by constructing and manipulating BlockFiles.

The BlockFile interface is abstracted in a way that allows for multiple BlockFile implementations. This has created tremendous opportunity for flexibility and optimization. The simplest BlockFile class, named SimpleBlockFile, stores the sound data in an uncompressed disk file. One example of a more specialized implementation is SilentBlockFile, a class that represents silence by storing only the length of the silence. This class can easily return a buffer of all zeros when it is asked for data, and since it does not store any data on-disk it is inexpensive to create. As a result, a user can perform "insert silence" on a track for almost no cost, regardless of the length of the silence inserted. Another example is PCMAliasBlockFile, a class that does not store any data of its own, but instead references or "aliases" data in an existing uncompressed sound file. This makes it possible to import a WAV file without copying the source data.

2 Motivations

Ideally, one would compress to a lossy format only as the last step of editing. If this were always possible, it would never be necessary to edit compressed files and deal with the question of how to avoid losing quality. However, in some cases the original data may not be available; it may not have been preserved because the cost of doing so is high (uncompressed data is commonly ten times larger than compressed) or it may not be available to the person performing the edits.

A particular domain where the ability to perform arbitrary cuts in a compressed file could be useful is speech processing. A person may want to cut a small part of a large speech or conversation that has been archived in a lossy format.

As discussed in the next section, there are several specialized programs that perform lossless editing. In this paper we take a contrary approach of integrating this capability into a general-purpose editor. This gives users lossless editing within the context of a feature-rich editing environment as opposed to specialized programs which may lack even such basic features as sound playback.

3 Previous work

There are a handful of specialized programs that can losslessly edit MP3 files. Data Becker's *MP3 Editor* advertises its ability to "modify MP3 files in their original format quickly and easily." The *mp3Trim* program[2] can make cuts in mp3 files as well as amplifying/attenuating the sound data. Other programs that support this capability are MPEG Audio Scissors and Mayah EditPro.

For Ogg Vorbis files, concatenation is supported through simply concatenating two smaller files. There is a program included in the main distribution of the standard Ogg Vorbis tools package called "vcut" that can split an Ogg Vorbis file into two on an arbitrary sample. It does so using the same features of the Ogg Vorbis format that I use to add this

[2] http://www.logiccell.com/ mp3trim/

capability to Audacity. Vcut is labeled as "experimental" since it has not been rigorously tested, but the basic algorithm is sound.

All of these programs are very specialized applications that exist primarily to provide lossless editing. They generally support only one format and their feature sets are narrow compared to general-purpose audio editors. To the author's knowledge there are no general-purpose editors that integrate this feature.

4 Constraints

The main constraint for this work to be suitable for Audacity is that it be able to correctly handle data that has been manipulated using any of Audacity's capabilities. It is important that users be able to trust that the file they export will match the work they have done in all cases, from envelope editing to time-shifting to applying effects. It must be able to handle a mix of data that came from Vorbis files and data that did not. When the exporter cannot create the output file losslessly, it must gracefully degrade so that the validity of the output is maintained. Much of this work is ongoing, and it will be a while before a stable version of Audacity is released where this feature is enabled.

We must also work within the existing Ogg Vorbis format, following the official specification.[3] We must use features of this format to support lossless editing operations, and any output we generate must conform to this format.

5 Implementation

The biggest challenge was devising a way to support lossless editing in the context of Audacity's flexible feature set. Once a Vorbis file is imported, it can be manipulated by many features such as effects, envelope editing, cut and paste between projects and mixing with other tracks. Some of these operations can be performed losslessly and some cannot. How can Audacity know, when the user decides to export an Ogg Vorbis file, what can be losslessly copied from the input files and what cannot? This challenge highlights the contrary approach we take compared with the specialized programs mentioned previously. A specialized program need only ask the user what edits to perform, after which it can perform them directly. In our case, we must provide this capability within the context of an existing editing framework which includes features that cannot be implemented losslessly.

Our general strategy, then, is to defer all of the lossless operations until export. Only then can Audacity determine to what extent the edits that have been performed can be re-performed using lossless algorithms. For the exporter to perform this analysis, it must have information about what compressed files the data to be exported originated from. For example, if the Ogg Vorbis exporter sees that the track it is exporting is an excerpt from an existing Ogg Vorbis file, it can copy the data from that file instead of recompressing to Ogg Vorbis. The first task was to find a way to preserve this information until the export step, through the basic edits of cut, copy, and paste.

An initial approach was to create a new BlockFile implementation that aliased a segment of an Ogg Vorbis file, similar to the PCMAliasBlockFile previously mentioned. It would decode the Vorbis file on the fly whenever data was read from it, as happens with playback or effects processing. The Ogg Vorbis importer would then create these specialized block files instead of SimpleBlockFiles on import. At export time, the Ogg Vorbis exporter would look to see which block files were of this special kind, and use that information to take data directly from the original Ogg Vorbis file without decoding and re-encoding.

There were two main problems with this approach. First, decoding on the fly introduced significant overhead that would reduce the number of tracks that Audacity could play at once. Second, the scheme was difficult to implement well, especially in the case where an edit would split an existing block; in this case we want to create two smaller blocks that are both of this specialized Ogg Vorbis type, but the way the Sequence class is implemented made the logistics of this difficult.

A second attempt was much more successful. It

[3] http://www.xiph.org/ogg/vorbis/docs.html

used the SimpleBlockFile class to store uncompressed PCM data, but added meta-data to this class to describe what files the data came from. It was represented as an array of "alias regions," where an alias region is an assertion of the form: "samples x–y of this block came from samples u–v (in channel z) of Ogg Vorbis file f." It is easier to preserve this information through block splits and concatenations, because (1) the entire block does not have to be aliased, (2) discontiguous regions can exist within one block, and (3) the Sequence class can continue to construct SimpleBlockFiles in the course of editing as it always has, instead of having to know that in certain situations it should construct a specialty class instead. Implementing this scheme consisted of adding a list of alias regions to the SimpleBlockFile class and modifying the methods of Sequence to copy this information whenever it creates new BlockFiles from parts of old ones. For example, `Sequence::Delete()` creates a new SimpleBlockFile consisting of the part of the boundary block that wasn't deleted. To preserve the alias region information, it should be copied from the original boundary block into the new block.

Once Audacity was successfully preserving information about these alias regions, attention could turn to the Ogg Vorbis exporting routine. The exporter's job is to decide how to split and recombine the source material into an output file in a way that exactly mimics the edits the user performed. To explain how the exporter does this an overview of the Ogg Vorbis format is in order.

"Ogg Vorbis" is so named because it consists of the Vorbis codec framed inside an Ogg bitstream. Ogg provides framing and synchronization information for Vorbis packets. By themselves, Vorbis packets are hunks of binary data with no information about where one packet ends and the next begins. Ogg encapsulates Vorbis packets inside Ogg pages, where Ogg pages are the physical unit of transport. One Ogg page can have one packet, many packets, or only a partial packet that spans more than one page.

A sequence of Ogg pages constitutes a logical Ogg bitstream. One or more logical bitstreams can be concatenated together to form a physical bitstream. In this way, concatenation of multiple files can be achieved by simply concatenating the files verbatim.

Each Vorbis packet with the exception of the first is combined with the previous packet to produce a certain number of samples. For example, the first packet does not produce any samples on its own, but is combined with the second packet to produce a certain number of samples, as is the second with the third. This characteristic of combining two packets to produce samples is known as the "overlap/add" property of Vorbis packets. The stream of Vorbis packets can be truncated at any point, yielding a valid Vorbis stream that ends with the samples that are produced from the last two remaining packets. Likewise, packets can be removed from the beginning of an existing stream, which will effectively remove uncompressed samples from the beginning of the stream. Between these two operations, samples can be cut from the beginning and/or end of an existing Vorbis stream, though only on packet boundaries.

Through a feature of the Ogg Vorbis format, this cutting capability can be extended to provide sample-accurate cut. Every Ogg page contains a field called "granulepos" that represents the total number of decoded samples in the stream after the packets in this page are decoded. For example, if the first five Vorbis packets produce a total of 1000 samples and they were placed together on an Ogg page, the granulepos of that page would be 1000. However, if the granulepos of the first page is less than the number of samples contained in that page, the decoder will discard samples from the beginning so that the decoder returns only the number of samples indicated by the granulepos. The compressed data is not altered in this case, the decoder just discards the extra samples. Likewise, the granulepos can be used to end the stream on an arbitrary sample. If the granulepos of the *last* page is smaller than the total length of the stream, the decoder will discard samples from the end of the stream so that the total length of the stream matches the granulepos field. These two features make it possible to perform arbitrary cuts on existing compressed data.

Between the capabilities of concatenation and sample-accurate cut, we can losslessly support all cut, copy, and paste operations with a single exception: the granulepos feature of the Ogg Vorbis format does not support discarding samples from both the begin-

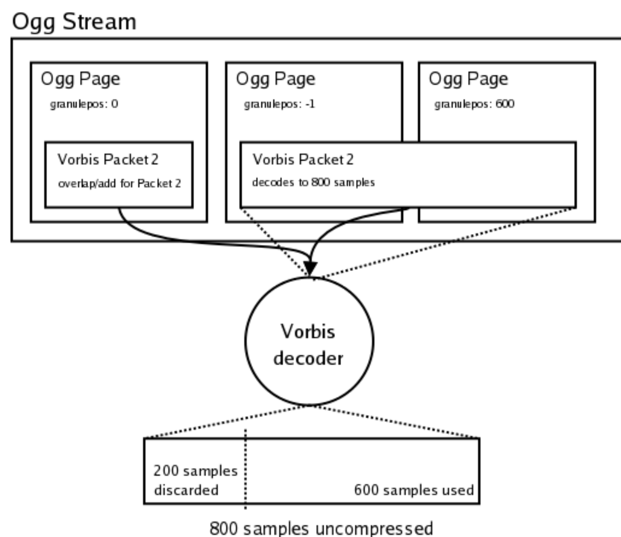

Figure 2: Using the granulepos feature to discard samples from the beginning of a stream

ning and the end of a single Vorbis packet. If a user were to copy and paste a region that was completely within a Vorbis packet, that region could not be losslessly copied to a new file. Taking into account the small size of Vorbis packets which is on the order of hundreds of samples, this is not a significant limitation.

When the user chooses to export a track from the project as Ogg Vorbis, the program passes that WaveTrack off to the exporter along with all of the information about the track's alias regions. With that information, writing the algorithms to create new Ogg Vorbis files while preserving the theoretical maximum amount of the original data was straightforward, though the details require special care.

The exporter works in two basic passes. The first pass deals with the complexity introduced by multichannel export such as stereo or more complex channel configurations. Since channels of a Vorbis stream are inseparable, the first pass ensures that the alias regions of all the output channels line up. It takes the alias regions for each track in the list and consolidates it into a list of alias regions that pertain to all tracks together. Since channels of Vorbis files are inseparable, an alias region can only be used if we are taking all the channels in that region. For example, if the user imports a stereo Vorbis file and tries to export only one channel of it, the exporter cannot use the original Ogg Vorbis material because it cannot separate the unused channel from the used one. An outline of this first pass is:

```
for region in WaveTrack[0].aliasRegions:
  for track in WaveTrack[1..n]:
    region =: INTERSECTION(region, track.aliasRegio

  if not EMPTY(region):
    commonRegionList.push(region)
```

The second pass takes this list of alias regions common to all the channels and uses it to copy compressed data directly from the source files to the output file. For each alias region the exporter creates a new logical bitstream in the output file, with the granulepos fields set to put the cuts on the exact samples.

6 Results

From a purely technical standpoint, the outcome is clear: it is possible to integrate lossless editing into a general-purpose audio editor by maintaining metadata about where the original compressed source data for every audio clip can be found. This defers all of the recombining to the export step.

A useful consequence of the implementation is that it is general enough that adding support for other formats will not require any changes to the alias region system, all that is necessary is writing a new exporter.

Though the technical ramifications are relatively well-understood, some more work is called for concerning how lossless editing can be well-integrated into the user interface. The user will want some feedback about what data can be reconstructed losslessly during the editing process and how their edits affect that. One method for communicating this information that I used in the development of this project is to draw lines around the sections of each track that have alias regions associated with them. It is likely that such sections of the tracks can be exported to

Ogg Vorbis losslessly.

One possible interface enhancement that could yield better results for formats that do not support cuts in the middle of a compressed frame or packet would be an option to restrict the user interface such that cuts always happen on packet boundaries.

One issue I did not address is how to handle cases where some of the data being exported can be copied from a compressed file but some cannot. This case would arise, for example, if a user applied an effect to the middle of a compressed track; the regions that were unaltered could be copied from the source file verbatim, but the region that had the effect applied would need to be recompressed. It is unclear whether it would be beneficial to recompress only part of the data, or how this could be done such that the "seams" between recompressed and copied data are not noticeable.