Manipulating A/V files with FFmpeg

This page was adapted from a page in the LLILAS Benson Digital Inititaives wiki space, and much of what follows was put together by Ryan Sullivant, Digital Content Curator at AILLA.

FFmpeg

The open-source FFmpeg engine can be used for several purposes. These include, converting between a/v containers and formats, re-encoding media with a different codec to produce smaller derivative access files, and rotating video. FFmpeg can be used as a command line tool or can be integrated into Python scripts using the `ffmpeg-python` package (github), which is useful for batch processes, especially if you are more comfortable using Python tools to identify which files need to be fed into FFmpeg (literal lists, reading from a file, regex filters with `glob`, etc.) than doing the same things with the command line, terminal, or powershell. As FFmpeg has no native MP3 encoder, another library, like LAME, may need to be installed to produce MP3 files.

Windows installation

FFmpeg is often distributed as a compressed folder. Depending on the format, Windows Explorer may not be able to open or extract this folder by default. You can use 7-Zip or WinRAR to open a wide variety of compressed file formats in Windows. Start by extracting the contents of the FFmpeg folder to a local directory, such as C:\FFmpeg.

In order to run FFmpeg directly using the command prompt in Windows, the location of the extracted *bin* folder containing ffmpeg.exe needs to be added to the *Path* environment variable. This process may require system administrator approval, but is relatively fast and easy. Instructions can be found here: http://web.archive.org/web/20210723181111/https://www.thewindowsclub.com/how-to-install-ffmpeg-on-windows-10

Once FFmpeg has been properly mapped to *Path*, it can be run in the Windows command prompt with 'ffmpeg' alone. The command prompt examples below assume FFmpeg has been installed in this way.

Convert audio formats

Sometimes we receive digitized or born-digital audio in formats that are not supported by our Islandora repository. Circa 2019 at AILLA, these have included files in WMA, AIFF and OPUS (a lossly format used by WhatsApp). Staff should convert these files into the supported WAV or MP3 formats, depending on the nature and quality of the unsupported digital files. Format conversion with FFmpeg can be as simple as specifying a different extension for the output file, but additional details, such as those discussed in Creating smaller audio files, may also be specified.

Using the Windows command prompt:

```
ffmpeg -i input_file_path output_file_path
```

Using Python:

```
ffmpeg.input(input_file_path).output(output_file_path).run()
```

This syntax can also be used to extract MP3 audio from an MP4 file, by specifying the MP4 file as the input and the MP3 file as the output.

Batch converting audio formats

If you have a directory full of files that need to be converted from one format to another, you can perform this transformation on each of them in succession using a *for* loop in the command prompt:

```
for /f "tokens=1 delims=." %a in ('dir /b *.wma C:\directory\audio_files') do ffmpeg -i C:
\directory\audio_files\%a.wma C:\directory\audio_files\%a.mp3
```

In the above example, all WMA files found within the C:\directory\audio_files folder will have MP3 copies created. These extensions can be changed to any format supported by FFmpeg to achieve different results

Creating smaller audio files

Some digitization labs will produce uncompressed files (typically WAV format) with much greater resolution than is necessary or useful (for example, 24 /96k), and are often 2 or 4 times are big as files produced in more smaller resolutions. In these cases, we can re-encode the files with one of these lower resolutions. (*NB: these resolutions are ideal for recordings of speech meant for listening and acoustic speech analysis; audio files with different contents and purposes may require higher resolutions.*)

The two resolution options for audio (`ar`) are '`48000`' and '`44100`' (the former is more common with audio associated with video files, the second for standalone audio). Audio file bitrate (`ab`) can be specified as well. Stereo files can be converted to mono by specifying only one audio channel (`ac`). This should be done in cases there the original recording was made with one microphone, which is likely the case for many field recordings originally made on tape. Some digitization workflows will produce two audio channels even from mono recordings. In these cases, slight differences in the digitization settings and condition of the tape heads may result in slightly different left and right channels despite originally having the same input.

Python:

```
ffmpeg.input(input_file_path).output(output_file_path, ab=16, ac=1, ar=48000).run()
```

Command prompt:

```
ffmpeg -i input_file_path -ab 16 -ac 1 -ar 48000 output_file_path
```

Note that different commands may be used to alter the sizes of compressed files (like MP3).

Rotating video files

Some video files specify their orientation in their technical metadata. Sometimes, a file is incorrectly presented as portrait when it should be in landscape orientation and vice-versa. It is possible to transpose all pixels of the video, but a simple solution is to edit the file's technical metadata (while copying all other metadata) to inform video players of the proper orientation. This is done with the `-map_metadata`, `-metadata:s:v` and `-codec copy` flags as below. Note that rotate specified the degrees of rotation in a counterclockwise orientation; '`90`' rotates counterclockwise and '`270`' rotates counterclockwise. It is unclear how this command (in particular the -metadata:s:v flag) can be implemented with ffmpeg-python, so this function can only be done via the command line currently.

Command prompt:

```
ffmpeg -i input_file_path -map_metadata 0 -metadata:s:v rotate="270" -codec copy output_file_path
```

Creating smaller video files

Some high-definition cameras will use codecs, like XAVC S, that produce very large files, in part because each frame contains many many pixels. Many of these files may be too large to ingest directly into a repository. One way to produce a smaller file that is still suitable for online viewing is to re-encode the file with a different codec, one very widely supported codec to try is H.264.

Python:

```
ffmpeg.input(input_file_path).output(output_file_path,vcodec='h264').run()
```

Cmd:

```
ffmpeg -i input_file_path -vcodec h264 output_file_path
```

If your video files have a high frame rate (that is, a multiple of 24, 25, or 30), then a lower frame rate can be specified with a command like `-r 24` (cmd) or `r='24'` (python).

Commands exist for altering the size of video frames in the output, but these have not yet been explored.

Converting MP3s to AAC MP4s

The UT Libraries Collections Portal will only stream an audio file if it has been ingested into the DAMS as an MP4 file with AAC audio. While it's normally used for video files, the MP4 extension is really just a "wrapper" to combine audio and video streams, and there' s no requirement that there even be a video stream. That means you can convert from MP3 audio directly to MP4, and as long as you specify that the output will use AAC audio, it will work with the Collections Portal. Using the command prompt:

```
ffmpeg -i C:\directory\audio_file.mp3 -acodec aac C:\directory\output.mp4
```

If you have a directory full of MP3 files you would like to convert to AAC MP4, you can batch the process using a simple for loop on the command line:

```
for /f "tokens=1 delims=." %a in ('dir /b *.mp3 C:\directory\audio_files') do ffmpeg -i C:
\directory\audio_files\%a.mp3 -acodec aac C:\directory\audio_files\%a.mp4
```

If no other options are specified, the AAC audio will match the sampling rate of the MP3 input audio.

Note that this will not delete the original MP3 file, but will create a new MP4 file alongside it. Because AAC is a better compression algorithm overall than MP3, the output files will be about 80% smaller than the input files, while achieving the same audio quality.

Compressed inputs like MP3s can be converted relatively quickly, but larger files like WAVs may take more time. You can verify that the output MP4 uses the AAC audio codec by using a tool like File Information Tool Set (FITS) to extract technical metadata. Information about the audio codec is found within the "<track type="audio"...> tags:

```
<track type="audio" id="1" toolname="MediaInfo" toolversion="0.7.75" status="SINGLE_RESULT">
  <audioDataEncoding>AAC</audioDataEncoding>
  <codecId>40</codecId>
  <codecFamily>AAC</codecFamily>
  <compression>Lossy</compression>
  <bitRate>128301</bitRate>
  <bitRateMode>Constant</bitRateMode>
  <duration>1932589</duration>
  <trackSize>30994332</trackSize>
  <soundField>Front: L R</soundField>
  <samplingRate>44100</samplingRate>
  <numSamples>85227175</numSamples>
  <channels>2</channels>
```

According to Wikipedia, versions of FFmpeg released before February 2016 used an AAC encoder that produced generally low-quality output, so it is a good idea to use more up to date releases: https://en.wikipedia.org/wiki/Advanced_Audio_Coding#FFmpeg_and_Libav

Getting runtime (or other information) for a list of files

In addition to converting and manipulating objects, FFmpeg can also be used to extract technical information about A/V files. Specific technical metadata fields can be filtered in the Linux command line using the *grep* command. Using a Linux command line such as Ubuntu on the Windows Subsystem for Linux, the following script will produce a text list of filenames & runtimes:

```
for file in `ls /path_to_file_directory/` ; do ffmpeg -i /path_to_file_directory/$file 2>&1 | {
grep "Duration" & echo $file; } | { tr '\n' ' ' ; echo ' '; } >> /path_to_output_directory
/runtimes.txt ; done
```

This script lists the contents of *path_to_file_directory*, then feeds each file in the list to ffmpeg as input. The *grep* command filters out all output lines except those containing "Duration", which is how ffmpeg displays runtime. The filename is echoed, so each runtime can be paired with the corresponding file. The *t r* command replaces the newline characters separating the runtime and filename with a blank space (thereby outputting them on the same line), then adds a blank space after the output to break the output for each file up. The processed output is saved as a text file in *path_to_output_directory*. The output can be further refined to remove additional unwanted information, but as it is written above the script will produce an output that is easy to manipulate and extract data from using Notepad++ or a spreadsheet program. See below for an example output:

```
tav00001_vid1.mp4   Duration: 00:58:27.27, start: 0.000000, bitrate: 627 kb/s
tav00001_vid2.mp4   Duration: 00:59:59.00, start: 0.000000, bitrate: 780 kb/s
tav00001_vid3.mp4   Duration: 00:23:14.39, start: 0.000000, bitrate: 772 kb/s
tav00002_vid1.mp4   Duration: 01:02:18.04, start: 0.000000, bitrate: 495 kb/s
tav00002_vid2.mp4   Duration: 00:49:49.52, start: 0.000000, bitrate: 496 kb/s
tav00003.mp4   Duration: 01:00:25.09, start: 0.000000, bitrate: 498 kb/s
tav00004.mp4   Duration: 00:46:09.90, start: 0.000000, bitrate: 517 kb/s
tav00005_vid1.mp4   Duration: 01:01:03.73, start: 0.000000, bitrate: 776 kb/s
tav00005_vid2.mp4   Duration: 01:01:23.98, start: 0.000000, bitrate: 791 kb/s
tav00005_vid3.mp4   Duration: 00:57:16.87, start: 0.000000, bitrate: 806 kb/s
tav00005_vid4.mp4   Duration: 00:12:59.31, start: 0.000000, bitrate: 781 kb/s
tav00006_vid1.mp4   Duration: 00:57:09.43, start: 0.000000, bitrate: 386 kb/s
tav00006_vid2.mp4   Duration: 00:24:49.79, start: 0.000000, bitrate: 318 kb/s
tav00007.mp4   Duration: 00:44:46.82, start: 0.000000, bitrate: 522 kb/s
tav00008.mp4   Duration: 01:03:51.00, start: 0.000000, bitrate: 516 kb/s
tav00009_vid1.mp4   Duration: 00:59:28.40, start: 0.000000, bitrate: 787 kb/s
tav00009_vid2.mp4   Duration: 00:59:15.99, start: 0.000000, bitrate: 497 kb/s
tav00009_vid3.mp4   Duration: 00:12:06.03, start: 0.000000, bitrate: 500 kb/s
tav00010_vid1.mp4   Duration: 00:38:25.74, start: 0.000000, bitrate: 672 kb/s
tav00010_vid2.mp4   Duration: 00:56:10.24, start: 0.000000, bitrate: 2584 kb/s
tav00011_vid1.mp4   Duration: 00:58:54.66, start: 0.000000, bitrate: 669 kb/s
```