Video Summarization

CSCI 576 Final Project

Christopher Mangus

Louis Schwartz

Group 17

Presentation Date: 8 Dec. 2011

Background

In this project, we designed and implemented a video summarization tool that returns a truncated version of the source file by a user-defined percentage. The tool analyzes the source video to determine shot breaks and uses audio and motion metrics to assign a weighting to each shot and scene. Consequently, only the most important percentage of shots and scenes are output, resulting in a logical and efficient summary. This is useful if the user wanted to quickly browse a long video in a short period of time.

The Audio/Video Player

Driver: *videoPlayback.java*
Audio Player: *PlaySound.java*
PlayWaveFileException.java
Video Player: *imageReader.java*
imageReaderComponent.java

The media player instantiates two separate threads, one for playing audio and the other for playing video, in order to synchronize the playback.

The audio player is loosely based on the skeleton code provided by the TA. It works by opening an AudioInputStream using a BufferedInputStream from the wav file. Each call to *audioInputStream.read()* returns either a full buffer of bytes or whatever is left over at the end of the audio file. As each chunk of data is stored into the buffer, it is written to the DataLine and output to the system speakers. The audio plays back at a predetermined sample rate given in the header of the wav file.

Similarly, the video is read, frame-by-frame, into a BufferedImage from a FileInputStream. This process is executed in the *readBytes()* method of *imageReader.java.* The input file is assumed to be a raw RGB file where each frame is a sequence of the red pixels, followed by the green and blue pixels. To play back each frame at the correct frame rate, the FPS (Frames Per Second) is a hard coded constant (=24 fps for our videos). The *PlaySound* object is passed into the imageReader class in order to pull the current position of the audio playback. If the video playback is ahead of the audio, imageReader will do nothing until the audio catches up. Similarly, if the video is behind the audio, *imageReader* will fast forward through the frames until synchronization is achieved. Finally, the image is added to the JFrame, which is subsequently repainted.

Input Files

The three input files that are used as sources for testing and implementing the summary program are given in Table 1 below.

Sample	Video File Name	Audio File Name	Length
1	terminator.576v	terminator.wav	10 min.
2	terminator3.576v	terminator3.wav	30 min.
3	sports1.576v	sports1.wav	30 min.

Table 1 – List of Given Input Files

The frame rate for all three video files is 24 frames/sec. Each video frame is 320x240 pixels (width x height). All of the audio files are 16-bit, mono, wav files sampled at 22050 Hz.

Note: The given audio file for sample1 included one extra second of audio samples at the beginning of the file. Thus, we simply cropped the first second from terminator.wav so that the audio and video will be synchronized.

Audio Level Analysis

Audio Analyzer: *audioAnalyze.java*

Once the shot breaks have been determined, the list of breaks is read into the ArrayList *breaks*. The average amplitude of the audio levels throughout each shot is computed. This is done by taking the mean of the Most Significant Bytes of the samples within each shot. Since, for our wav files, there are two bytes per sample, we take the second byte only into consideration, since the byte order is Little-Endian. We are assuming that each break signifies the first frame of each shot. Thus, the end of each shot would be one frame prior to the next break point. This analysis given a very accurate hierarchy of loud shots down to soft shots, even though the average values for each shot are very similar.

Audio and Video Writers

The audio and video writers are methods contained within *audioAnalyze.java*.

The video writer, *writeVideo()*, is essentially identical to the video player, except we write the output to a FileOutputStream, whose destination is "videoOutput.rgb" instead of the JFrame. Additionally, only the frames dictated by the *finalShots* ArrayList are written to the OutputStream.

The audio writer, *writeAudio()*, is essentially identical to the audio play, except we write the output to a FileOutputStream whose destination is "audioOutput.wav" instead of the dataLine. The The frame numbers from *finalShots* are converted to byte numbers, and only the bytes dictated by the *finalShots* ArrayList are written to the OutputStream.

One caveat of writing the audio file is that we cannot simply just write the raw bytes of audio information to the wav file; we must write the header first. To do this, we used http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html as a reference. *buildHeader()* is the method that writes, byte by byte, the header to the beginning of the wav file.

Video Segmentation

Video Segmentation Creator: videoSegment.*java*

To begin separating the video into pieces that can be analyzed, shots had to be made from the given video feed. The first thing that was needed to create shot breaks was a color histogram. Every 3rd frame in the video stream had a color histogram calculated. Using only the 2 most significant bits, the color histogram was created with 64 bins according to where the color was on the scale from 0-255 in each R, G, and B. After this, the average color intensity was calculated for each histogram. The absolute difference between frames was calculated and compared with a threshold. If the difference was greater than the threshold cutoff value, a shot break was detected. This break was then added to an ArrayList of shot breaks, and there was a block on adding new shot breaks for 5 seconds. The block for adding new shot breaks was added due to flickering lighting and fast pans. These effects created unnecessary shot breaks when there was not a very good logical reason to have a shot break.

Motion Analysis

Motion Analyzer: motionAnalyzer.*java*

First for motion analysis, we attempted some methodologies gathered from the research papers. These were deemed to be too complex and time consuming for the summarization to occur fast enough. In replacement, an investigation on the image brightness was made. Using the Y component of the YUV transformation, the brightness for a frame was gathered. Every 2nd frame a calculation was performed. In the first instance, the average intensity value was calculated for the entire frame. This was then compared with the next frame's average intensity versus some threshold. The percentage of threshold crossing values was used as a metric for how much motion was occurring on a frame-by-frame basis. Second, each frame was split into blocks of size 16x16 pixels. This was to gain a finer estimate than the first calculation. These 16x16 blocks were compared in each frame versus a threshold just as before. In this calculation, the metric was the percentage of threshold crossing block changes in every 2nd frame.

Summarization

Video Summarizer: videoSummarize.*java*

In culmination of all of the analysis, weights are given to all of the partitioned shots. The weight is a delicate balance between the audio level analysis, the full-screen brightness analysis, and the block brightness analysis, whose results have all been normalized between 0 and 1. The parameters were tuned after multiple iterations of testing the project.

After the combined weight of each shot is computed, the shot number and corresponding weight are stored, in decreasing order by weight, in the LinkedList *weights*. Finally, the fraction (1 – *percentage*) of shots are copied in the ArrayList *finalShotNums*. Once we have nearly the correct percentage of shots retained, the shots are reordered into correct temporal order into the ArrayList *finalShots*. This array contains a sequence of first and last frame numbers for each shot to be written to the final summary output.