EDITING SYSTEM WITH HISTORY FOR COMPRESSED AND NON-COMPRESSED AUDIO / VIDEO STREAMS

СИСТЕМА ЗА РЕДАКТИРАНЕ НА КОМПРЕСИРАНИ И НЕКОМПРЕСИРАНИ АУДИО И ВИДЕО ПОТОЦИ С ИСТОРИЯ НА РЕДАКЦИИТЕ

Angel Radanov Kanchev, Antoaneta Aleksandrova Popova

FKTT, Technical University of Sofia, “Bratya Bukston” bvd 40, Sofia, Bulgaria, phone: +359 898 427123, angel_kanchev@mail.bg

Keywords: video, compression, bitstream, editing, lazy evaluation, history

Abstract – There are specifics in video editing that every GUI editor must take care of. Well written GUI editor can take advantage of these specifics in order to be interactive and with low requirements for free disk space. We propose abstract editing system that can do any type of edit operation over compressed or non-compressed audio, video or data stream. The system is very open because it uses external module for editing (called Editing Tool). In addition it allows multiple undo / redo with command history without the need of large free disk space. The price of using the system is the requirements for the creation of the Editing Tool but they don’t affect the tool’s functionality. The proposed system loads the CPU (Central Processor Unit) more than the other editing systems but the use of lazy evaluation approach makes the system more interactive than the others.

1. INTRODUCTION

Most of today’s video editors are specialized. There are editors for montage of movies ([11], [2]), editors for effects applying ([3], [4]), editors for compression changing ([5], [6]). The most universal editors are combination of frame server ([7], [8], [9]), script interpreter ([8], [9]) and simple GUI editor ([6], [9]). There is no universal GUI-based editor for both compressed and non-compressed streams of any type: audio, video or data. This work describes algorithm that can edit both
compressed and non-compressed streams in any way. The algorithm is abstract because it uses external tool for the editing logic (Editing Tool).

Video editing has one specific thing – the number of operations that are applied to the streams (audio, video or data) is much smaller than the size of the data that is affected. Very often the streams are in different files – using their data via reference (like in MPEG4 standard – [10]) instead of copying the data saves a lot of disk space and time. In addition, applying of edit commands could be done not in the time of applying but when the edited data is necessary. That technique is called “lazy evaluation approach” or “deferred execution model”. There are many studies for evaluation strategies ([11]) or specifically for lazy evaluation ([12]) but few are specialized for video editing. Lazy evaluation approach is used in [13] but the optimization problem is fixed only with caching.

We propose a technique that allows history list for multiple undo / redo along with optimizations according CPU and necessary free space. The basic idea is that the modifications caused by the user will not be applied in the moment of their request but just when they are necessary. Modifications are necessary when the user saves the editing result; or when the user wants to see some part of the stream that is edited. The real slow operation is saving the final result when all operations are performed for real. The preview will load the processor more because some operations will be done while getting the “edited” data. But with current processors that are over 2 GHz the user should not even feel that some of the modifications are executed in the preview time. Anyway, caching technique will be used to lower the CPU loading.

Having system with commands for later execution and the possibility to apply the commands in random time allow creation of history queue that can be used for multiple undo / redo without the need of using large spaces on the hard disk.

Finally – the system allows for the user to watch all original sources and the final result of the editing in parallel.

General description of the algorithm is given in paragraph 2, command interoperation – in paragraph 3 and information on compilation of commands – in paragraph 4.

2. DESCRIPTION OF THE EDITING ALGORITHM

There are two types of streams that can be handled by the algorithm: compressed and non-compressed. In compressed streams the smallest accessible piece of data is one bit and for indicator is used the index of that bit in the stream. In non-compressed stream the smallest piece of data is one frame (audio or video) and indicator is the frame index (audio or video frame index). For non-compressed stream with external data the appropriate index is used (for example subtitles need the video frame index).

On Fig. 1 is shown the block scheme of the editing algorithm. With thick arrows is shown data flow. The data that is “edited” is called virtual data. The reason for that name is that the virtual data is evaluated on-demand – via command execution. It is not data for real.
The operations that the user can perform (User Operations) are split in 3 categories: “Pull Data”, “Edit” and “Undo / Redo”. Operations “Pull Data” and “Edit” can be performed by the user as well as by some Editing Tool.

Each “Edit” operation consists of one or more commands (Create Command(s)). There are 3 command types: “replace” (modify), “add” and “delete”. All commands have size and position in the stream where should be applied. Note that the size is number of indicator indexes, not bytes. Analogically position in the stream is indexed position (could be bit position or frame position). Commands “add” and “replace” have buffer with data or reference to real or virtual data. The data in these commands could be bit sequence, frame sequence or (in case of “replace” command) parameters for algorithm that modifies the virtual data stream.

On Fig. 1 is shown that after a command is created, it is added to the History List and compilation with the already applied commands is done in order to fill the Execution List. If somebody needs “edited” data, a “Pull Data” operation is used. This operation executes the commands in Execution List in order to evaluate the “edited”, i.e. the virtual data. The Execution List is made in a way to be easy to find and execute the appropriate commands for a virtual index. The process of compilation from History List to Execution one will be explained in details later.

History List contains commands in the order of their applying (in time). For preliminary information – Execution List contains commands, ordered by their indicator index. The module “Execute” in Fig. 1 searches for command on the wanted index and if there is such – it is executed. More information about that can be found in paragraph 4.
It was mentioned above that the operations “Pull Data” and “Edit” could be caused directly by the user as well as by some Editing Tool. The user can cause “Pull Data” operation by playing the stream or by saving it to a file. However most of the editing is not done directly by the user but via some Editing Tool. The tool caches necessary virtual data and allows the user to manipulate it using some limited preview. When the user is satisfied an “Edit” operation is performed (the Editing Tool applies one or more commands). In addition, the tool could need data from some virtual stream, so it could cause “Pull Data” operation.

3. COMMAND INTEROPERATION AND REPRESENTATION

The virtual data stream could be represented as in Fig. 2:

Fig. 2 Command “add” before applying in the virtual stream

Each point is bit or frame at some index in the stream. Before applying command “add” only its place does matter. After command “add” is applied it affects the indexes after its place and gives data for each index in between. On Fig. 3 is shown how data for virtual index 2 became data for virtual index 8 and what data is for virtual index 2 (the first index in the “add” command). Also note that command that is to be applied is shown below the streamline while the already applied command is above it (see Fig. 2 and Fig. 3).

Fig. 3 Command “add” after applying

Analogically “delete” and “replace” commands are shown on Fig. 4 and Fig. 5.

Fig. 4 Command “delete” before and after applying

Fig. 5 Command “replace” before and after applying
Note that command “replace” in contrast to “add” and “delete” does not change the indexes after its position of applying...

All possible interactions between two successive and overlapping commands are given in Fig. 6. One of the commands is already applied and the other is to be applied. The combinations are 9 and for some of them there are different interactions.

When two successive commands overlap in one of the cases in Fig. 6 a compilation procedure is started. It ensures that the commands don’t overlap. Actually Execution List contains such non-overlapping commands that are produced by the compilation. The compilation procedure is done for each new command and for all commands in the Execution List that are overlapped by it.

![Fig. 6 All possible interactions between two successive commands that overlap](image)

### 4. COMPILATION EXAMPLE

Here is shown example with 4 overlapping commands. On Fig. 7 is shown a History List and how the commands in it form the Execution List. For more compact representation only the first letters of the command names are used (a – “add”, r – “replace”, d – “delete”).

You can see that “replace” commands are saved one-above-another. This is called execution stack. There are two types of “replace” commands:

1. Absolute – it does not depend on previous data so it is used directly for virtual data
2. Relative – manipulates previous data so these commands are put in a stack. The output of upper command is input to the command below it in the stack.
6. CONCLUSIONS

The proposed system allows good interactivity, multiple undo / redo (with history) and low requirements for free disk space. The system can be realized with the framework described in [14] and could use the plug-in system to interface with external Editing Tools.

7. REFERENCES

[6] TMPEGEnc (http://wwwTMPGEnc.net/)
[8] Avisynth (http://www.avisynth.org/)
[9] VirtualDub (http://www.virtualdub.org/)