Budapest University of Technology and Economics

Department of Telecommunications and Media Informatics

Students' Scientific Conference 2015

Lóránt Marcell

Analysis of H.264 encoded video streams embedded in 3GPP containers

Supervisor

Dr. Sándor Molnár (BME-TMIT)

Péter Megyesi (BME-TMIT)

Budapest, 2015

Contents

Abstract

Video streaming has become very important nowadays but sufficient bandwidth for satisfying quality is not always available. During my research work I was removing frames from video streams and observing the quality changes of the bitrate-reduced video streams. I was working with H.264 videos embedded in 3GPP containers which consist of Inter (I) and Predicted (P) frames. With the manipulations of these frames the changing of video playback speed or disposing all the motion, which makes the video look like a slide show, are possible.

In this research I have performed QoE (Quality of Experience) measurements including human testers in order to see that quality degradation of lower bitrate videos are how much sensible to the human eye. In this experiment different amount of frames from different positions of test videos have been removed.

Kivonat

Napjainkban a video streaming szolgáltatás egyre fontosabb szerepet tölt be, ám nem mindig áll rendelkezésre a megfelelő sávszélesség az adatfolyam kielégítő átviteléhez. A tudományos diákköri munkám során azzal foglalkoztam, hogy egy videó adatfolyamból különböző helyekről bizonyos mennyiségű képkockát eltávolítva mennyire romlik a felhasználói élmény, miközben a bitsebesség csökken. A kutatás során 3GPP konténerben tartalmazott H.264 kódolású videókkal dolgoztam, melyben teljes (I) és prediktált (P) képkockák voltak fellelhetők. Ezek manipulálásával érhető el a lejátszási sebesség gyorsítása vagy lassítása, illetve akár a teljes mozgás eltávolítása a videóból, diavetítés jellegű hatást elérve.

A munkám során több tesztalany bevonásával QoE (Quality of Experience) méréseket is végeztem azzal a céllal, hogy bizonyos tesztvideókból különböző mennyiségű és elhelyezkedésű képkockákat eltávolítva az emberi szem mennyire érzékeli a romlást a csökkentett bitrátájú videó minőségében és folyamatosságában.

1. Introduction

Video streaming's popularity is continuously growing and more and more modern technologies appear on the Web. Unfortunately, our Internet connection could be a bottleneck at times, enjoying our video streams in a satisfying quality is not always possible. To get around this problem, reduction of the stream's bandwidth can be achieved by transcoding the video with a smaller resolution or lower bitrate, but in real time these methods are quite resource-intensive. It is much quicker and less expensive to remove some lower priority frames from the motion picture so high image quality is still available with the use of less bandwidth.

But how do we know that which frames could be removed safely without messing up the whole stream? In this research I investigated that how does removing several frames from different positions affect the experience to the viewers and I also did measurements of bandwidth and frame sizes. This task included file format analysis and Android application development.

In Chapter 2 I am going to introduce the relevant parts of the 3GPP container format and the H.264 encoding technology. In Chapter 3 and 4, I will describe the development of a screen recorder and a video stream manipulator application. Chapter 5 will contain my measurement results while Chapter 6 will summarize my research work.

2. The 3GPP container and the H.264 codec

H.264 is an efficient video coding technology and the 3GPP container is a very popular format to store H.264 streams. For example, all Android phones use these formats by default while recording motion pictures. A stream consists of several frames which are of three types: Inter (I), Predicted (P) and Bidirectional (B). An Inter frame is a full static image while Predicted frames only store differential information related to the previous frames. Bidirectional frames do quite the same as the Predicted ones, but they are able to refer to the following frames. Each frame starts with the byte sequence $000001B6_h$, and their type can be determined by the following two bits, these are 00_b, 01_b and 10_b in case of I, P and B frames, respectively [1].

The 3GPP container [2] has several so-called boxes that hold information on the file. A box can be contained within another so they can construct a tree-like structure. A box starts with its length in bytes as a 32-bit big-endian integer (everything inside the container is big-endian). The size of the box is followed by the box tag. The top-level boxes are *ftyp* (file type), *moov* (movie metadata), *free* (padding) and *mdat* (media data), this last one holds the sequence of frames.

In this research the relevant header boxes were *stsz* and *stts*. These are contained within *moov*/*trak/mdia/minf/stbl*. *Stsz* contains the frame sizes, and when I started developing the frame replacer application I was not aware of this. So the program read the *mdat* section from byte to byte, and collected details like sizes and types of the frames this way. Obviously it was really slow; moreover, the modified video became unplayable. Realizing that frame sizes are located in *stsz*, implementing a function that reads sizes from there caused a 60-times performance improvement in parsing. The structure of the *stsz* box can be seen in Table 1. The *stts* box (see Table 2) contains the durations of the frames. They are stored as STTSRECORDs (see Table 3) that contain two numbers. The first one is the sample count – which describes how many consecutive frames have the same duration – and the second one is the sample duration.

Field	Size (byte)	Comment
Tag	4	„stsz”
Version	1	Expected to be 0
Flags	3	No flags defined, set to 0
Constant size	4	Used if all the frames share the same size
Count	4	Number of size entries
Size entries	Count*4	Only exists if constant size is not set

Table 1 – The stsz box structure.

Field	Size (byte)	Comment
Tag	4	„stts”
Version	1	Expected to be 0
Flags	3	No flags defined, set to 0
Count	4	Number of STTSRECORD entries
STTSRECORD entries	Count*8	Array of STTSRECORD entries

Table 2 – The stts box structure.

Field	Size (byte)	Comment
Sample count	4	Number of consecutive samples with the same duration
Sample duration	4	Should be divided by 100 to get milliseconds

Table 3 – The STTSRECORD structure.

3. The screen recorder application

I have developed an Android application that can record the screen using MediaProjectionManager [3] and MediaProjection [4] classes introduced in API Level 21 so the application can only run on Android version 5.1 (Lollipop) or higher.

On its user interface there are three fields (width, height and bitrate) and a toggle button that starts or stops the recording. For the UI, see Figure 1. Pressing this button causes two boolean variables (start and stop) to change their values. This is important because these variables are observed in a listener running on a separate thread. If they are changed, the listener calls the startScreenSharing or the stopScreenSharing methods. The first one calls the prepareRecorder method and a screen capture Intent is sent through startActivityForResult. At this point the user will see a window asking for screen capture permission. Inside the onActivityResult method, if screen capture is permitted, the media projection object is initialized and the recording is started. When the recording is stopped, null value is assigned to the media recorder and the media projection object. In the prepareRecorder method, encoding parameters are set such as video source, format, frame rate, resolution and bitrate. File name is also set here which is the current date and time. The user cannot specify an individual file name, it is always automatically generated. The maximal duration of the video is set to 1 hour, after this interval the recording restarts itself.

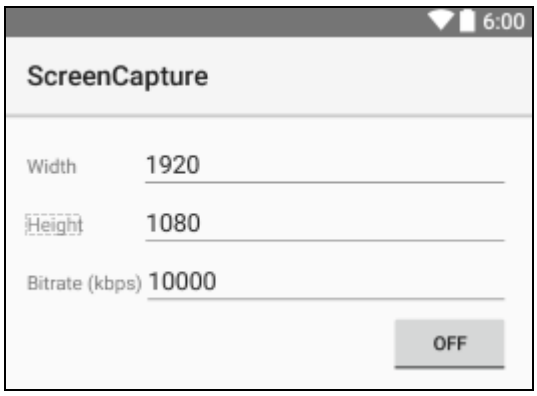

Figure 1 – Screen capture application UI.

4. The H.264 Parser application

In order to perform the measurements I have developed an application that reads the video file and exports statistical data. The program was written in Java with Android Studio [5], so it is a smartphone application.

4.1. Reading data from video files

The H264Parser class reads the data from the video file and stores it in its private members. Frame types, offsets, sizes and durations are stored as arrays. Frame types are stored as boolean values – true if the actual frame is an Inter and false if it is a Predicted frame. Bidirectional frames cannot be found in the streams recorded by the screen capture application, so B frame recognition had not been implemented. Offsets cannot be determined directly from the file, they must be computed. There are six more integers among the private members: frame count and box positions of *tkhd*, *stts*, *stsz*, *stsc* and *mdat*. I made getter functions to be able to read the private members' values from external classes. For example, the frame type getter method has an integer parameter N and returns the type of the N-th frame. The constructor has only one parameter, a string that is the path of the input video file. The file is opened here, then the box positions and frame details are determined by the proper methods, finally the file is closed.

For getting the positions of the boxes, I have chosen not the most optimal solution. There are two methods for this purpose, the first one (getBoxPosition) uses a naive algorithm: it always reads four bytes at a time and goes one byte ahead until the currently read four bytes equal the string in the method's only parameter. It can be seen that it is really time-consuming to find a string in a larger file using this method, but fortunately most of the boxes are located near the beginning of the file. For the *mdat* box, this is not the case. There is a large padding before it so finding the *mdat* with this method would take a lot of time, so a second function had to be written for this purpose (getMdatPosition). I knew that *mdat* is a top-level box so with reading the sizes of the top-level boxes and skipping them we finally get to the *mdat*. The most optimal solution would have been a recursive function that can find the boxes on each level.

The frame details reader method (getFrameDetails) starts with jumping to 16 bytes after the *stsz* box position where the frame count is located. When it is read from the file, the

arrays for storing the frame offsets, sizes, types and durations are initialized. We are already in the right position to start reading the frame sizes, so this and the frame offset calculation are done within a for cycle immediately. Then the file pointer is positioned 12 bytes after the *stts* box start position where the STTSRECORD count can be found. The number is parsed and then a for cycle goes through the entries. Sample count and duration is read in case of each entry and then an inner for cycle fills the frame duration array with the help of an index variable named k that is incremented in every iteration of this inner cycle. When this is done, finally frame types are determined. In a for cycle, file pointer is positioned four bytes after the offset of the actual frame; there are the two bits that determine the frame type. A bitmask of $C0_h$ must be used because a whole byte is read at a time.

4.2. Writing statistical data to file

The StatWriter class is used to write statistical data that had been extracted by H264Parser to a text file. Its constructor has two parameters: a H264Parser object and an output file path string. The method called writeStatistics does all the operations, it generates a CSV file whose content is at first concatenated to a string. A row consists of the frame type, timestamp, duration, offset, length and bandwidth. Timestamp and bandwidth are computed within this method. Finally, the string is written to a file located in the specified path.

4.3. Removing frames

For user experience testing, a class had to be implemented that can remove Predicted frames from different positions. It is important to mention that streams created using the Android H.264 encoder have a constant I/P frame ratio – an Inter frame is always followed by 25 Predicted frames. The frame remover should have three modes: removal of every N-th frame, removal of N frames from the beginning and from the end. Figure 2 shows the frame sequence and the modes in case of N=3.

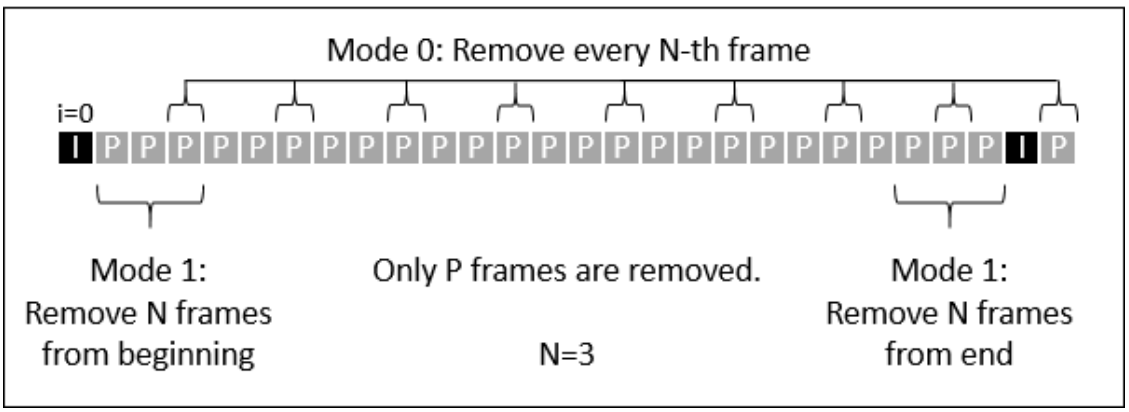

Figure 2 – example of Predicted frame removal.

Knowing that a Predicted frame always holds the difference to the previous frame, it can be guessed that removing every N-th frame or deleting them from the beginning can mess up the video until the next Inter frame. Thus better user experience can be achieved if P frames are removed from the end of the sequence. Hence, deleting several frames from the stream generates another question – what should be done with frame timing? Simply removing a frame including its duration entry causes play time to reduce, but we do not want that – the only thing that must be reduced is bandwidth. The solution is time stretching: the total duration of removed frames should be equally divided among non-removed frames, as shown in Figure 3.

Figure 3 – Time stretching.

The FrameRemover class' constructor has the following parameters: input and output file path strings, a H264Parser object, the mode and frame count as integers. Information on the original video stream is accessed through the parser object. New frame size and duration tables have to be built, these operations are written in the makeNewBoxes method. At first, the amount of removed frames and their total duration have to be determined. The final value of these numbers depend on the removal mode. Inter frames are always ignored. See Figure 4.

```
for (int i = 0; i < totalFrameCount; i++) {
    if (!parser.getFrameType(i) && (              // do not remove I frames
            (mode == 0 && i%count == 0)  ||       // remove every N-th frame
            (mode == 1 && i%26 <= count) ||       // N frames from beginning
            (mode == 2 && i%26 >= 26-count))) {   // N frames from end
                framesToRemove++;
                duration += parser.getFrameDuration(i);
    }
}
```

Figure 4 – Determining count and total duration of removed P frames.

Now that if we have these values, new frame offsets, sizes and durations can be computed. This is done in a for cycle similar to the previous one but the inner 'if' condition is inverted so it will be true for the non-removed frames. Time stretching value is added to frame durations.

The next step is rebuilding the *stts* box which includes creating new STTSRECORD entries. If there are consecutive frames with the same duration, only one entry should be created for them. The algorithm is shown in Figure 5.

```
j=0; // helper variable; used to index stts entries
int prevDuration = newFrameDurations[0];        // previous duration set to first value
for (int i=0; i<newFrameCount; i++) {           // go through new frame durations
    if (newFrameDurations[i] != prevDuration) { // if a duration differs from the prev
        sttsDurationValues[j] = prevDuration;   // store the duration
        prevDuration = newFrameDurations[i];    // change previous
        j++; // increment index variable
    }
    sttsDurationCounts[j]++; // increment the sample count for the actual duration
}
```

Figure 5 – Sample duration record generator algorithm.

New *stts* and stsz boxes are generated and stored as arrays of bytes. The writeNewHeader method reads all the data until the *mdat* box into a buffer in which *stts* and *stsz* boxes are simply overwritten. The modified boxes are smaller than the original ones because there are fewer entries after the frame removal, so there was no need for implementing an upper-level box size corrector algorithm. Zero-padding the remaining space was not necessary, entries from the original file filled the space. A number in the *stsc* box also has to be modified. It is the frame count and if it does not correspond the values contained within other boxes, the video will not play. Finally, the new *mdat* section has to be written. This is done using the arrays containing the new frame offsets and sizes. Data is read from the original file into a buffer and it is copied to the new file from there.

4.4. Modifying video playback speed

This class is quite simple but does really spectacular modifications to the video. The constructor has four parameters: input and output file path strings, a H264Parser object and a multiplier as a double. This last one is the number (now referred to as M) that frame durations will be multiplied by, so playback speed will be M times slower is M>1 and it will be 1/M times faster if M<1. If M=1, playback speed will not change. Time modification is done within the manipulateFile method which needs the multiplier and the *tkhd*, *stts* and *mdat* box positions. All data is copied to the new file until the overall video duration entry located in *tkhd*. Then this entry is read, multiplied by M and written to the output file. This is needed because if we had only modified the frame durations, the player would still display the original duration that would make jumping to a specific scene really hard.

After this, the file pointer is positioned to the duration entry count in the *stts* box and until that position all data is copied to the new file. Duration entry count is read and a for cycle goes through the entries in the original file and multiplies them by M, then writes them to the modified file. Finally, the rest of the original file is entirely copied. Thus all the frames and header boxes except for *tkhd* and *stts* stay untouched – only duration entries are modified.

4.5. User interface and Main Activity

The user interface is very simple: it has five text fields, a seek bar and a button. The user has to specify the paths of the input and output video file, the statistical file of the original and the modified video, the amount of removable frames (or the play time multiplier) and the frame manipulation mode. This last one can be set with the seek bar. The UI can be seen in Figure 6. When the user modifies the text in the first field, text in the other fields will be automatically changed according to the input. This is a convenient function because the user does not have to specify four paths, one is enough and the program generates the rest. However, if the user needs custom file names for the generated files, that is also possible.

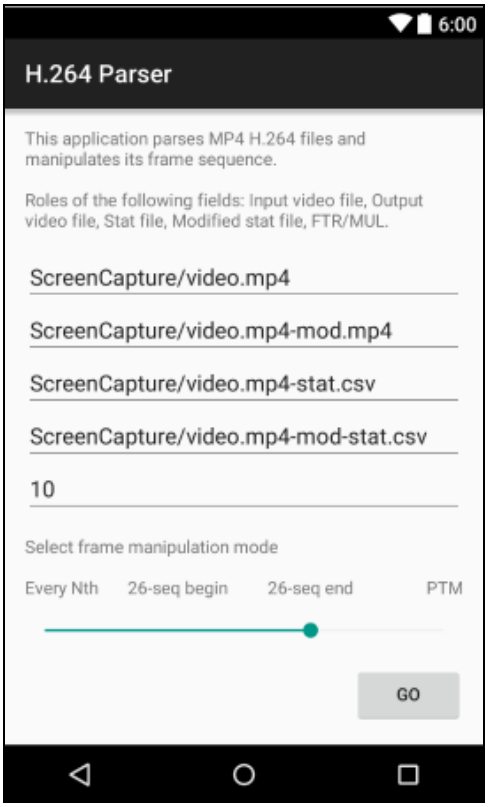

Figure 6 – The H.264 Parser UI.

5. Analyzing and manipulating video streams

5.1. Removing Predicted frames

I recorded a sample video [6] from YouTube that contains fast and slow movements. I was curious that how does removing frames from the stream influence the bandwidth and the user experience in case of faster or slower motion. I recorded this video with the resolution of 1920x1080 and the bandwidth of 10 Mbit/s. Then I tried to remove frames from different positions, at first, from the beginning of the P frame sequences. The result was really bad, removing a single P frame causes the whole video to look awful so I discarded to perform user experience measurements using this frame removal mode. The result can be seen in Figure 7.

Figure 7 – Bad image quality caused by removing one Predicted frame.

Also this is the case when removing every N-th Predicted frame. The reason is that P frames store the difference to the previous frame so removing one of them makes the incorrect differential information to accumulate with the further Predicted frames. With the removal of every N-th frame, the result is even worse.

The only method that creates enjoyable modified video is the removal of P frames from the end of their sequence. If a P frame is removed, all the following ones should be deleted too until the next Inter frame so image quality will not suffer. User experience measurements only make sense with this frame removal method. There are no artifacts

observable in the picture, only the motion continuity becomes worse as more and more Predicted frames are removed but thanks to time stretching, playback duration stays the same. In this measurement I removed from 1 to 25 P frames from the aforementioned video. In this case, removing 25 P frames means that the modified file has only I frames – this looks like a slide show without any motion.

Firstly, I looked at how the file size and video bandwidth changes with the removal of frames. In Chart 1 it can be seen that when no P frames were removed the bandwidth was above 10 Mbit/s and when all the P frames was removed it reduced below 2 Mbit/s so removing an extra Predicted frame reduces bandwidth by 384 Kbit/s on average. This was a 1m 33s long video, its original size was 121 MB and by removing all the P frames its size reduced to 13 MB. Removing an extra P frame caused file size to decrease by 4.3 MB on average.

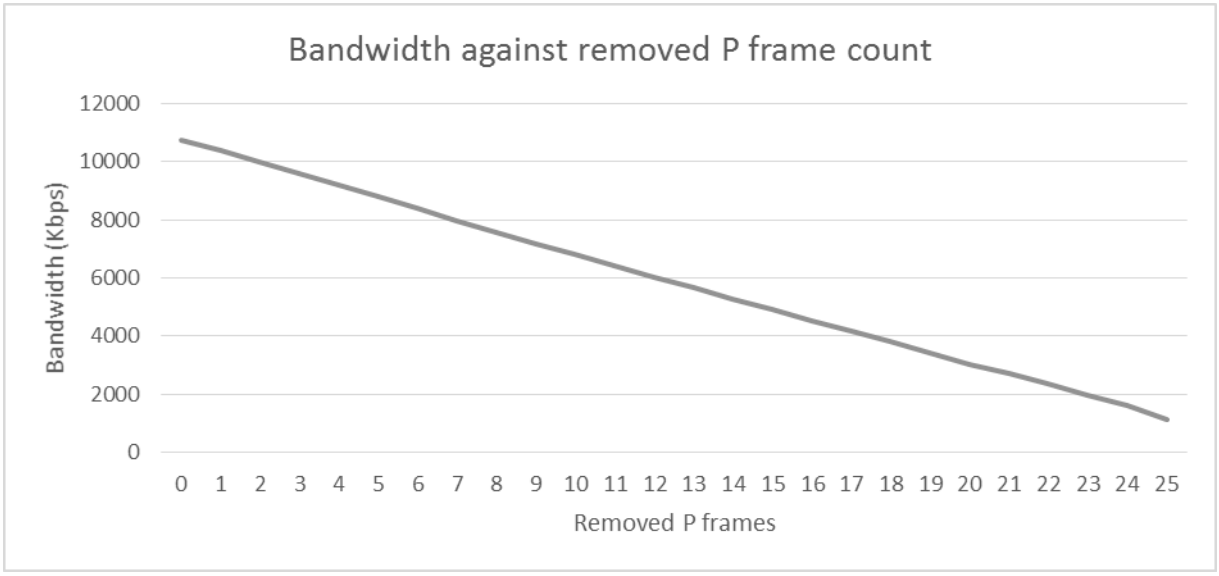

Chart 1 – Bandwidth against removed P frame count.

The next step in this measurement was to observe that how does current bandwidth change with the removal of different amount of P frames. In case of the original video (see Chart 2) current bandwidth varied between 235 and 108425 Kbit/s. The largest value was around 4 seconds, in case of one Inter frame (this one was nearly 500 KB). Its bandwidth was almost two times higher than the second highest value. Looking at the video, at this moment a sudden change of the picture was observable with a very big difference to the previous frame.

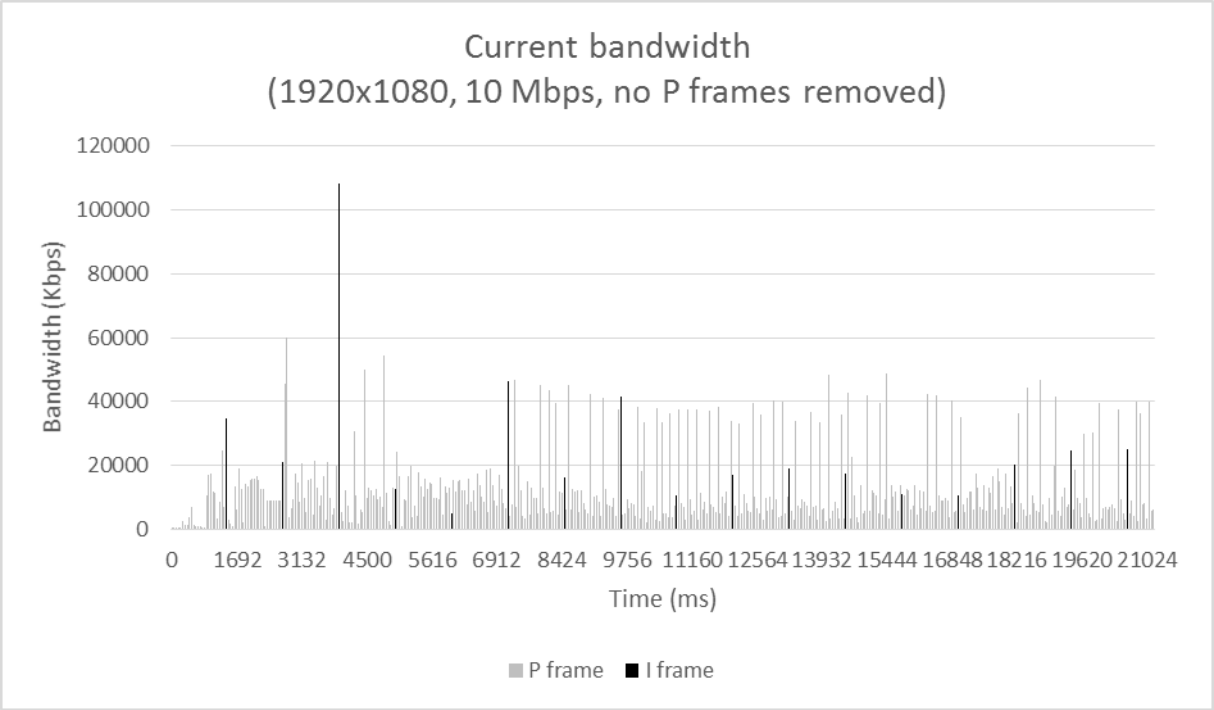

Chart 2 – Current bandwidth values of the original video.

In Chart 3, where 5 P frames were removed per I frame, the bandwidth reduction and lower Predicted frame density is noticeable. The highest current bandwidth peaked at 86740 Kbit/s while the lowest value was 188 Kbit/s. The video was still enjoyable, however, at faster motions some continuity loss was observable.

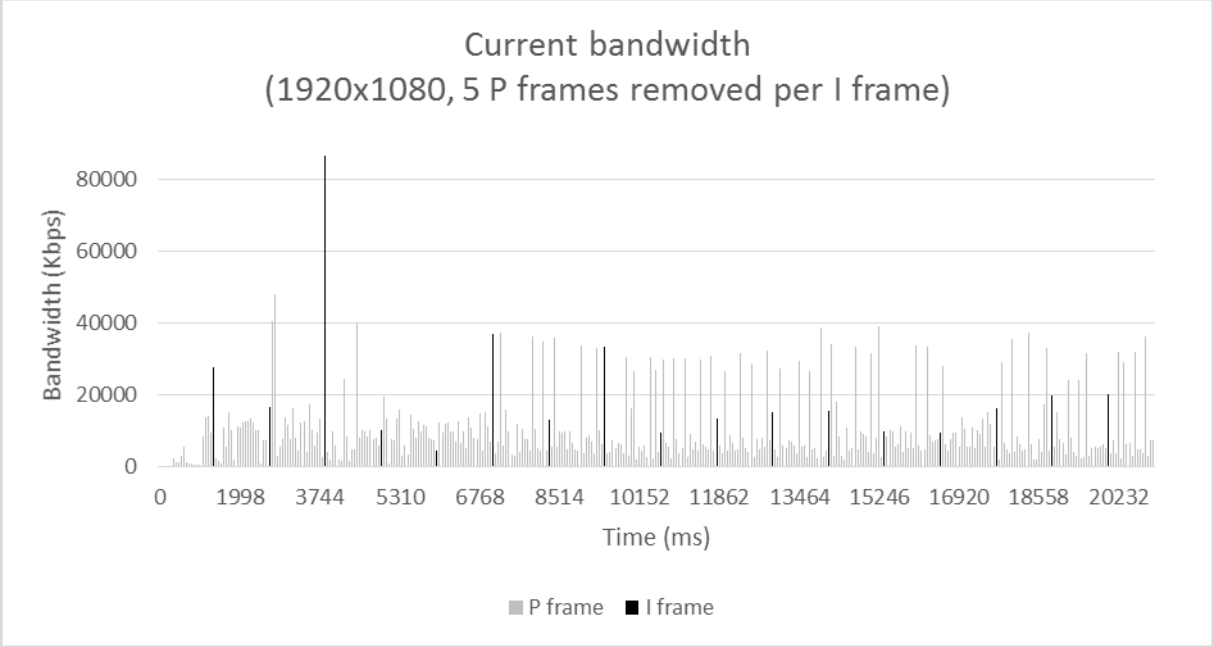

Chart 3 – Current bandwidth values with 5 P frames removed.

In case of the removal of 10 Predicted frames per Inter frame (see Chart 4), bandwidth values were between 139 and 63988 Kbit/s. The playback was noticeably laggy. When 15 P frames were removed (see Chart 5), playback became absolutely teary, and the bandwidth values were between 100 and 42893 Kbit/s. There were only 10 P frames for one I frame.

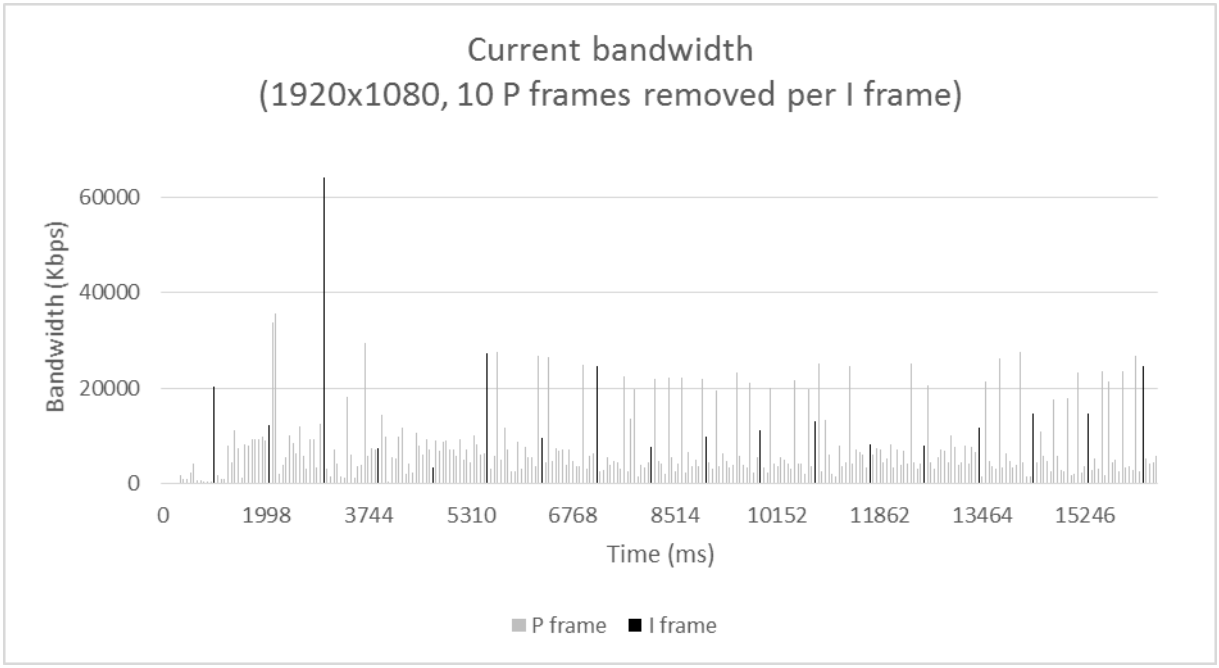

Chart 4 – Current bandwidth values with 10 P frames removed.

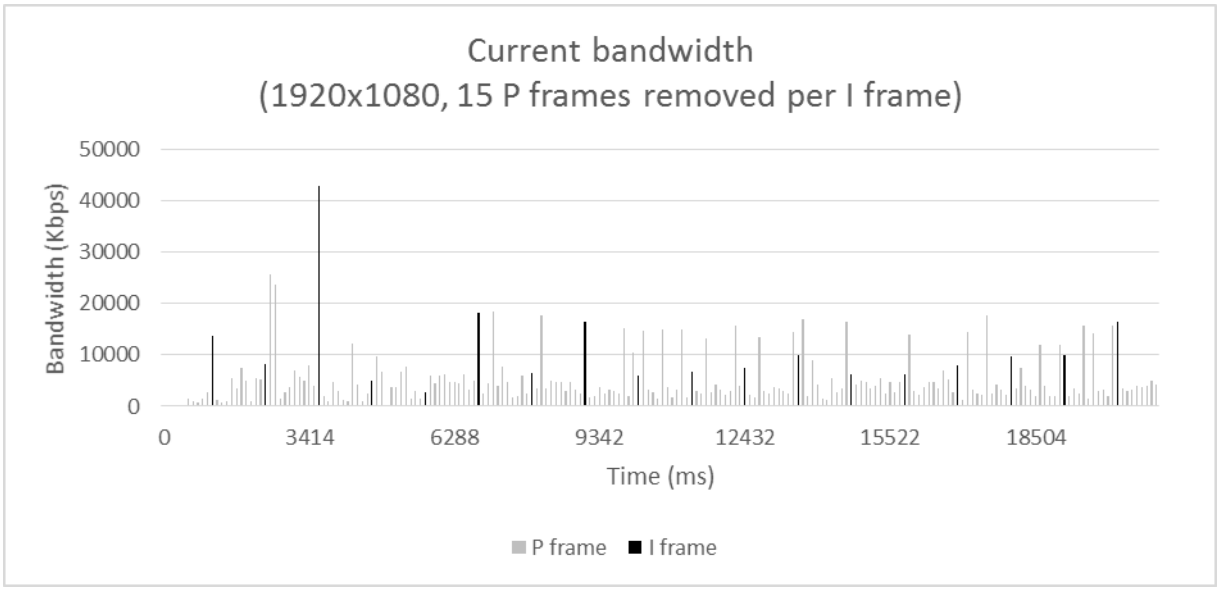

Chart 5 – Current bandwidth values with 15 P frames removed.

Removing 20 Predicted frames caused the video to almost look like a slide show, motion was barely noticeable. The bandwidth was between 58 and 22826 Kbit/s. Low P frame density can also be observed in Chart 6. Removing all the P frames as shown in Chart 7 caused bandwidth to decrease between 247 and 12390 Kbit/s. The minimum value is greater than in the previous case. This is because very small P frames were in the stream, and throwing them out caused the lowest value to be assigned to an Inter frame.

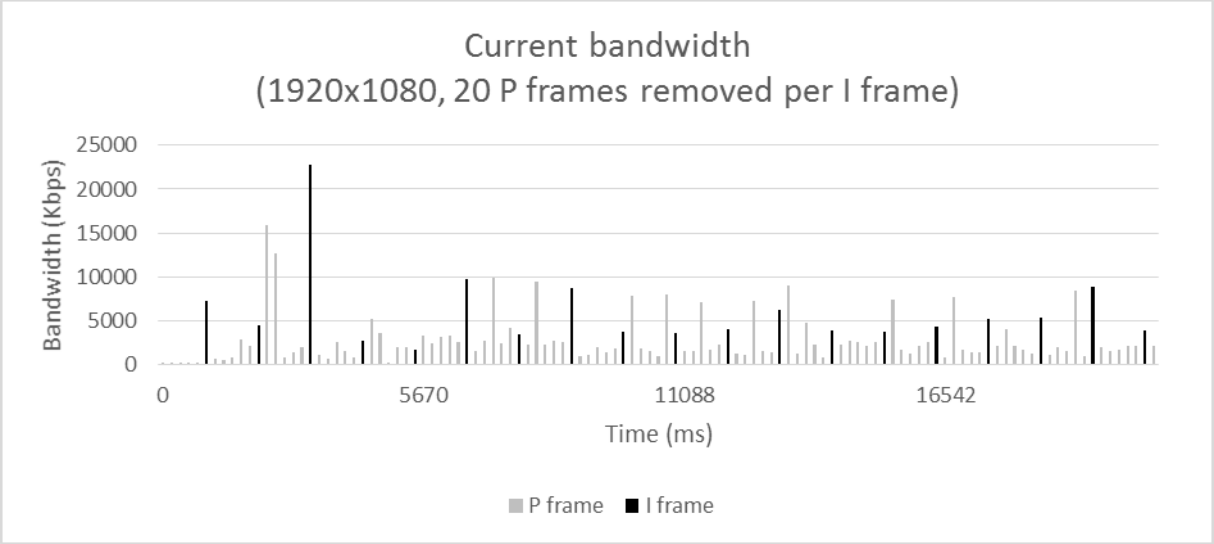

Chart 6 – Current bandwidth with 20 P frames removed

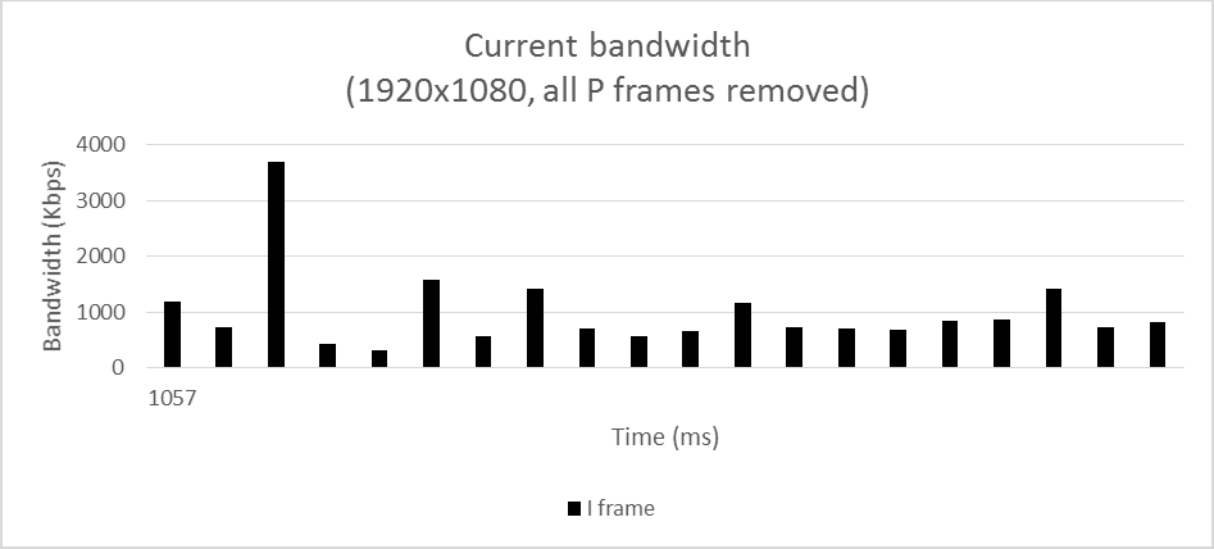

Chart 7 – Current bandwidth values with all P frames removed

I also measured the user experience level. There were 5 human testers whom I showed the original and the modified videos in a random order. They had to rate the continuity of the videos from 1 to 10. 1 meant poor while 10 meant excellent experience. The average scores can be seen in Chart 8, the more frames are removed the more the experience degrades. If we say that an average of 8 points means a fair quality then removing 6 or 7 P frames from the stream is a quite good solution for bandwidth reduction.

Chart 8 – User experience measurement.

5.2. Observing frame sizes during different user activities

The first activity was scrolling in the application launcher with different speeds. I had to observe how the bandwidth varied during this activity. The measurement can be seen in Chart 9. In case of a slower scrolling speed, bitrate was quite low but we can say that low and high values were continuously altering with each other. Higher spikes can be seen in the chart, they came when scrolling speed was really fast so the image changed suddenly causing P frames to be large and increasing the current bandwidth this way.

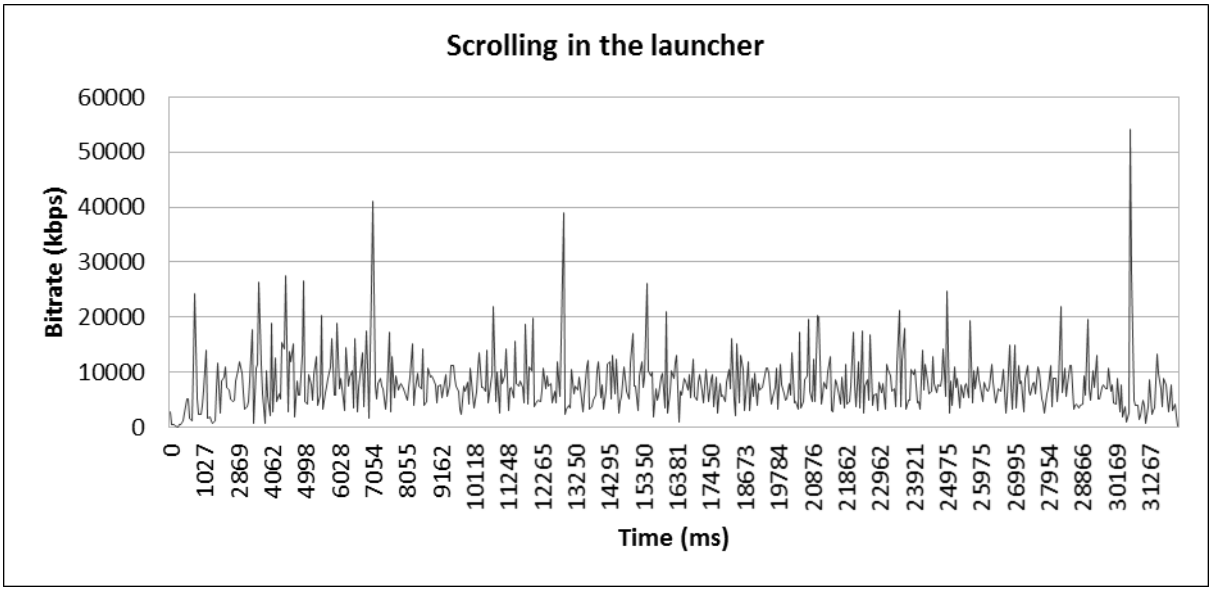

Chart 9 – User activity measurement: scrolling in the app launcher.

During the measurements I noticed that the H.264 encoding was not optimal. I already mentioned that, in case of this Android encoder, no matter how long the P frames are, an I frame is always followed by 25 P frames until the next I frame. This is impractical because if the entire picture suddenly changes, a very large P frame is generated. In this case it would be better to generate a new I frame because the difference from the previous frame is so big that use of a P frame is inefficient. The following measurement, in which I swiped through five static images in the picture viewer application, clearly illustrates this phenomenon (see Chart 10). The images were of 1920x1080 resolutions and each one was displayed for approximately three seconds. It can also be seen that displaying a static image causes frame sizes to be very small, they only increase if the image content is changed.

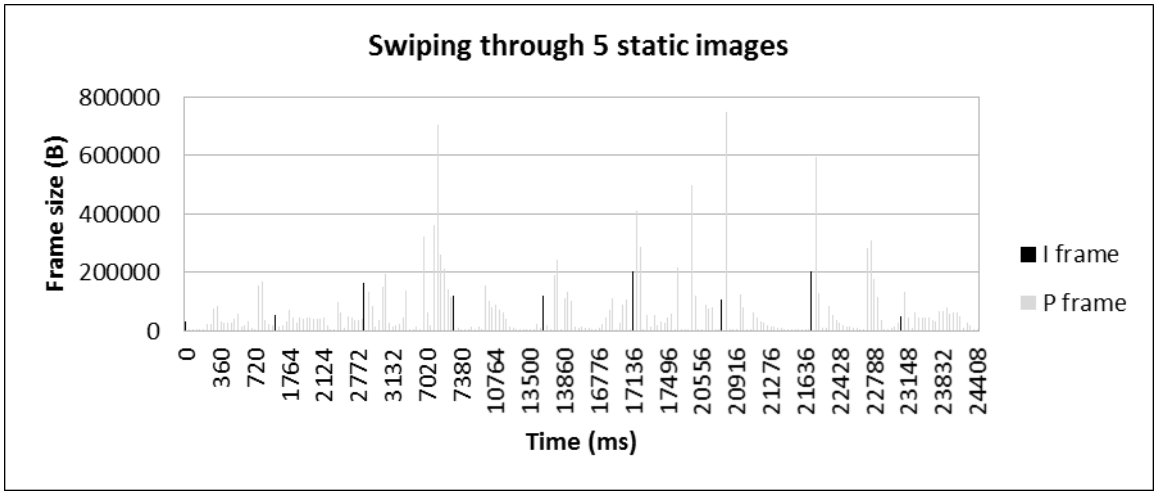

Chart 10 – User activity measurement: Swiping through 5 static images

The next measurement was to record the same YouTube video with different bitrates ranging from 3 to 14 Mbit/s with 1 Mbit/s intervals. I wanted to see that how does average frames size change with the bitrate. A linear relationship could be observed, see Chart 11.

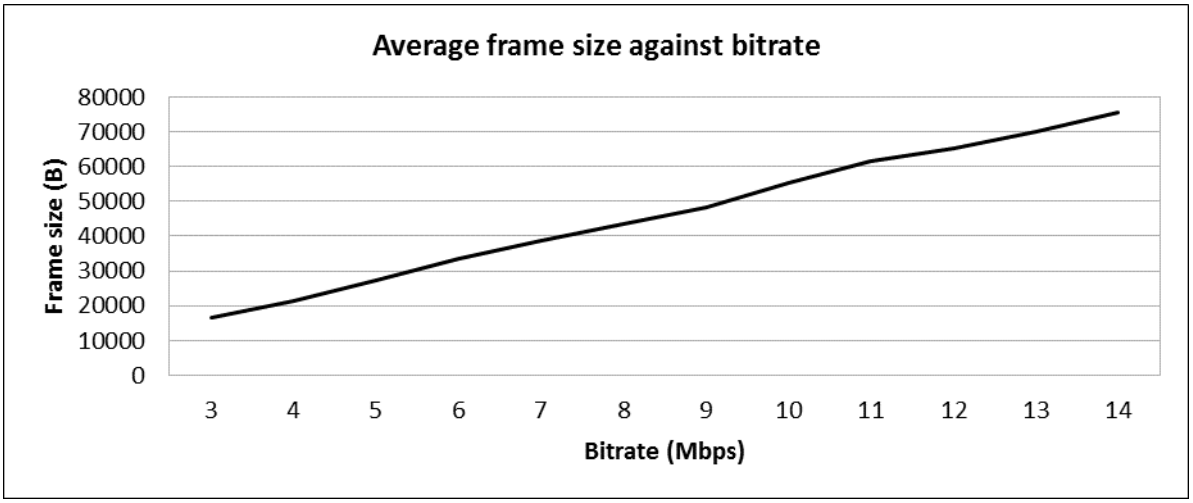

Chart 11 – Average frame size against bitrate.

6. Summary

During this research I investigated video encoding on mobile platform. For this purpose I have developed a screen recorder application using Android's one of the latest APIs, and another program that can collect information on video streams and modify them like removing and altering play speed.

Using these applications I have carried out several measurements related to frame sizes, bandwidth and user experience. I think this last one is the most important because if bandwidth becomes insufficient, we have to know how many frames can be thrown away without dramatically affecting the viewers' experience.

There are plenty of further possibilities in this research. For example, the frame removal algorithm could be implemented in a video streaming server so it could dispose some frames real-time if it experiences that the client has insufficient bandwidth.

7. List of references

[1] Adobe Systems Inc. – Video File Format Specification Version 10

https://www.adobe.com/content/dam/Adobe/en/devnet/flv/pdfs/video_file_format_spec_v10.pdf

[2] Identifying frame types

http://stackoverflow.com/questions/1957427/detect-mpeg4-h264-i-frame-idr-in-rtp-stream

[3] Android's MediaProjectionManager

https://developer.android.com/reference/android/media/projection/MediaProjectionManager.html

[4] Android's MediaProjection

https://developer.android.com/reference/android/media/projection/MediaProjection.html

[5] Android Studio

https://developer.android.com/sdk/index.html

[6] Video recorded from YouTube

https://www.youtube.com/watch?v=mr6x1bBdHiE

8. List of abbreviations

3GPP: Third Generation Partnership Project

I frame: Inter frame; a full image

P frame: Predicted frame; a differential image that can only look backwards

B frame: Bidirectional frame; a differential image that can look backwards and forwards

ftyp box: Contains file type

moov box: Contains movie metadata

stsz box: Contains frame size entries

stts box: Contains frame durations

STTSRECORD: An entry containing count and duration of some consecutive frames

tkhd box: Track header, contains video duration among others

stsc box: Contains frame count

UI: User Interface

9. List of tables, figures and charts