Optimizing 360-Degree Video Streaming based Texture Tile Division

Xingming Zhang, Yudong Wen, Jiasheng Chen

Abstract—There is a big challenge saving bandwidth usage in 360° video streaming. Commonly, users will not see all parts of a 360° video in a period and not pay the same attention to the different areas of a frame. To make use of these features, we propose our method to save bandwidth usage. We stream two sorts of video: one is whole the video in low resolution, others are part of whole the video in high resolution. We use the latter (high-resolution parts) to fill the viewport of the user, and the former (low-resolution parts) to fill the remaining viewport. The latter contains many tiles of the video in different resolutions. On delivering steam, we choose tiles according to the position of the viewport in varying resolution decided by texture complexity of the tile. Because of less bandwidth of low-resolution parts and more bandwidth of high-resolution, we make a trade-off between quality and bandwidth. We evaluate our prototype implement, compared to a baseline requesting all tiles in full resolution among 4 times area of the current viewport. And the results show our method can get less rebuffering time, less average bandwidth, and similar visual quality.

Index Terms—network transmission, 360-degree video, texture, tiling

1 Introduction

360-degree video is becoming more and more popular on many video platforms like YouTube, Facebook, iQIYI, Bilibili. It gives a chance to make users explore the content of the 360-degree video in their way. That is the most attractive feature for users, compared to traditional video.

The 360-degree video provides users with all-around visible content, including the space view surrounded by 360 degrees horizontally and 180 degrees vertically at the location of the observer. 360-degree video is usually made by multiple cameras shooting scenes from multiple angles at the same time and then mapped into the high-resolution spherical video after algorithm splicing [1].

In 2015, YouTube opened the virtual reality channel, and in 2016, Facebook also launched the Facebook 360 channel. In addition, the 360-degree video also attracted close attention from academic circles. In 2018, the Moving Picture Experts Group (MPEG) launched the standard chemical work (MPEG-I) for the immersive media, and the 360-degree video is the video part of the immersive media[2]. The Joint Video Research Group (JVET) has also started to introduce the coherent support of 360-degree video frequency into the next generation video coding standard - High-Efficiency Video Coding(HEVC)[2].

1.1 Challenges

360-degree video brings many challenges to streaming media transmission[1]:

1) High image quality requirements, to make the image quality of 360-degree video FoV clear and the number of visible pixels per degree(PPD) at each angle reach the level of ordinary 2D video, 360-degree video needs extremely high spherical full-angle resolution, which requires at least 4K to reach 8K to meet the basic requirements, and too low FoV picture quality will cause visual fatigue.

2) low interaction delay. To ensure the consistency between the user's turn head and the change of FoV picture, the Motion-to-Photons(MTP) delay should not exceed 20ms, otherwise, the user will feel dizzy.

3) High network bandwidth, high-resolution 360-degree video storage volume is huge, and network transmission needs plenty of bandwidth. To make users' FoV picture achieve the ultimate experience, the video needs a full-view resolution of 24K and FPS of 120, and the bandwidth is as high as 5Gbps.

1.2 Transmission Schemas

The total 360-degree video pipeline is described as follows: 1) Use multiple cameras to shoot scenes and record videos; 2) Merge the above videos to a high-resolution spherical video; 3) Transform the spherical video into a rectangular video; 4) Encode the rectangular video and transmit it into the client player; 5) Decode the video and inversely transform it into spherical video; 6) Render the rectangular content of the video based on the viewport.

There are mainly three streaming schemas as follows [2]:

1) Viewport-independent Streaming(VIS). The Equirectangular Projection(RRP) is used in this way that following the idea of map latitude and longitude projection, the spherical surface is projected to the side of the cylinder and then expanded into a plain rectangle. After such a projection, whole the rectangular 360-degree video is encoded and transmitted as the general videos do.

2) Viewport-dependent Streaming(VDS). The previously designated viewpoints are used in this way to produce as many as the number of viewpoints videos for a certain 360-degree video. Each of the above videos is related to a certain viewpoint and contains high-resolution content around the viewpoint and low-resolution content away from the viewpoint. During the user watching a 360-degree video, the client player will choose the video whose viewpoint is the most closed to the current users' viewpoint among all videos produced by the original 360-degree video.

3) Tile-based Streaming(TBS). The original 360-degree video is divided into many spatial rectangular tiles, and each of the tiles is related to some area of the picture. The client player chooses some tiles to request from a server according to the user's viewport. Because of the independence of tiles, each of the tiles is precessed and transmitted individually. This schema is most popular compared to others.

1.3 Our Method

In this paper, we propose our 360-degree video transmission system.

For saving bandwidth usage, we use tile-based streaming. Our player only requests the tiles corresponding to the user's viewport, which saves bandwidth. But a stall occurs when the user moves their viewport to another area where the tiles are not transmitted.

For addressing the above problem, we introduce a backup stream. The backup stream consists of a succession of short video chunks (all the chunks constitute the whole video). Because the chunks are short and continued, we replace the missing tiles with the part of chunks in time.

For saving bandwidth further, we propose our core idea: use different resolutions based on texture complexity for tiles in the viewport. This idea is based on such an assumption: Human pays more attention to the area containing complex texture and pays less attention to that is less complex. In detail, we define and compute the texture complexity of each tile. Then, we request different-resolution tiles based on their texture complexity. As a request, the tiles in the viewport are different-resolution rather than full-resolution, which saves bandwidth further.

2 Related Works

There are many existed works in the transmission of 360-degree video.

Ideally, it is satisfying for the video player to play partial content in the user's viewport. But user's viewport varies along with time and there is a nonnegligible delay from starting transmission to play. In these cases, a sudden viewport moving leads to a stall when the player only requests partial content in the user's viewport.

To address this problem, **Flare**[3] use a mixture of Linear Regression(LR) and Ridge Regression(RR) to predict the next several viewports according to history viewport trajectory. And the tiles around predicted viewports are picked for transmission. Besides, the author designs a Quality of Experience(QoE) formula favoring high average-tile quality and penalizing quality switches and stalls. And the formula is used to choose the quality of each video tile.

As mentioned in section 1.3, TBS is the most popular streaming schema. But how to decide the proper tiling schema? If the number of tiles is too few, the TBS will degrade into VIS that does not save bandwidth. If the number of tiles is too many, the video compress algorithm for each tile is not effective, which means each tile is too larger to save bandwidth.

To address this problem, **ClusTile**[5] first divide video into small tiles(named by basic-tiles). Then the author constructed an integer optimization for the combination of basic tiles to several big tiles(named by valid tiles). And the target of optimization is minimizing the total bandwidth of all valid tiles. After solving the above problem, the most proper tiling schema for saving bandwidth is ready.

Because of the spatial characteristics of 360-degree video, at a certain moment, users can only see part of the content. Therefore, users often take the initiative to explore, which will lead to users' distraction, resulting in the inconsistency between the video quality felt by users and the video quality sent by the server. **Pano**[4] define the quality-perception-index with the moving speed of viewpoint, the change of brightness, and the difference of depth of field around the viewpoint. As ClusTile does, Pano first divides video into small tiles, then combinates the small tiles of the same quality-perception-index to big tiles. In addition, the author uses a robust MPC[9] algorithm to estimate network bandwidth. And the quality allocation of tiles is defined as the optimization of maximizing the overall video quality-perception-index under limited network bandwidth.

For a certain 360-degree video, some popular areas are drawing more attention among all viewers. **Popularity-Aware**[6] detect these popular areas and use macro-tiles to encode them. In some cases that there are no macro-tiles in the viewport, and then the conventional tiling scheme is used. To support popularity aware 360-degree video streaming, the client selects the right tiles (a macro-tile or a set of conventional tiles) with the right quality, level to maximize the QoE under bandwidth constraint.

Compared to our method, **Non-Linear Sampling**[7] also uses the whole content of the 360-degree video. But for each CoRE frame, the full resolution is applied in the predicted viewport and the resolution is decreased in a non-linear way outside the viewport. So that missing pixels are avoided, irrespective of the view prediction error magnitude. Besides, for each CoRE video chunk(temporal chunk vs. spatial tile), it has the main part at full frame rate and an extension part (a backup of the main part of the next chunk) at a gradually non-linear decreasing frame rate, which avoids stalls while waiting for a delayed transfer.

3 Method and Implement

As mentioned in section 1.3, our key assumption is Human pays more attention to the area containing complex texture and pays less attention to what is less complex. So that we can select the proper resolution for each tile according to its texture complexity. As a request, for these less complex tiles, we select low-resolution version for them. And in this way, we can ensure as similar as high-resolution perception quality and save a bit of bandwidth.

Furthermore, there are three key problems to explain:

3.1 *How to calculate the texture complexity of a tile?*

To describe the texture complexity C of each tile T in a video chunk, a simple idea is to calculate the gray difference of each pixel of tile T by using the Prewitt operator, which can get two results —— $Diff_x$ in the horizontal direction and $Diff_y$ in vertical direction.

$$\begin{aligned} Diff_x(i,j) = |&T(i-1,j-1)+T(i-1,j)+T(i-1,j+1) \\ &-T(i+1,j-1)-T(i+1,j)-T(i+1,j+1)| \end{aligned} \tag{1}$$

Fig. 1. Original image of a video frame intercepted in 360-degree video(i.e.$Diff$ value).

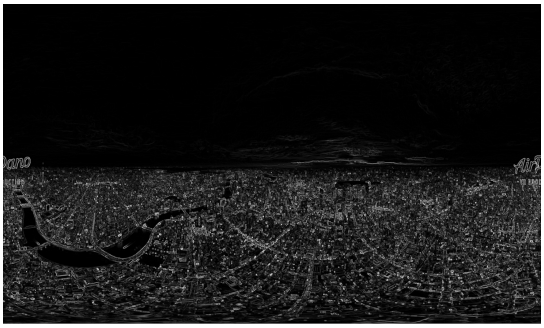

Fig. 2. Image processed by Prewitt operator.

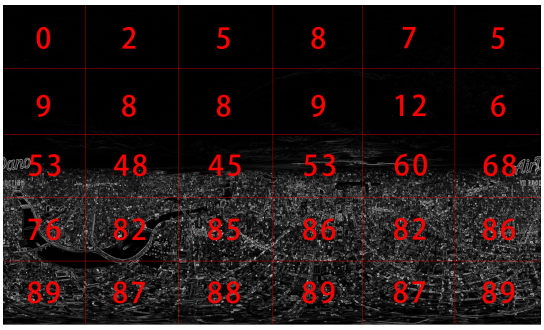

Fig. 3. Complexity statistics of each tile.

$$Diff_y(i,j) = |T(i-1,j+1) + T(i,j+1) + T(i+1,j+1) - T(i-1,j-1) - T(i,j-1) - T(i+1,j-1)| \quad (2)$$

Then, get the gray difference of a pixel $Diff$ by average both, and calculate the proportion of pixels that $Diff$ greater than λ to the total pixels as the representation of texture complexity C.

$$Diff(i,j) = \frac{Diff_x(i,j) + Diff_y(i,j)}{2} \quad (3)$$

$$C = \frac{|\{\, p(i,j) \mid Diff(i,j) > \lambda \ \wedge \ p(i,j) \in T \,\}|}{|\{\, p(i,j) \mid p(i,j) \in T \,\}|} \times 100\% \quad (4)$$

Figure 2 shows the visualization result of the $Diff$ value. It can be seen that this method can effectively mark tiles of high texture complexity, i.e. tiles with more white pixels in the figure. The more white points in the tile, the higher texture complexity it has, which is consistent with the judgment of most viewers.

3.2 *How to pick tiles for transmission?*

Our pick-tile method is reactive. In each render-frame phase, we calculate related tiles according to user's current viewpoint. Then the configuration of these tiles is sent to several Loaders (a tile is related to a Loader). The Loader will automatically choose soon and proper time to download the corresponding tile. Because of the short duration of each tile, this reactive method makes tile change in time. Besides, the continued backup steam makes the player not stall.

3.3 *How to select quality level of each tile?*

For simplicity, the quality level of a tile is only dependent on its texture complexity. To map the continuous value of texture complexity into a discrete quality levels, we design the following function:

$$level = \lfloor \log_2(a \times complexity + 1) \rfloor \quad (5)$$

where $a = (2^{levels} - 1)/100$ ($levels$ is the number of different quality level of tiles), $0 <= complexity < 100$ and $0 <= level < levels$.

In addtion, we have evaluated other function as following in section 4.X:

$$level = \lfloor \frac{complexity \times levels}{100} \rfloor \quad (6)$$

3.4 System Implement

See Fig.4. Our system architecture is similar to [8].

On the server-side, we use FFMPEG to prepare video tiling chunks (spatial and temporal parts).

First, we encode the original 360-degree video to several different-resolution versions of videos. Second, for each version of videos, we segment them into many temporal chunks. Third, for each chunk that is not in lowest-resolution we divide them into many spatial tiles (the lowest-resolution chunks are backup streams). Forth, for each tiling-chunk, we compute their texture complexity as described in section 3.1. Fifth, we collect all the tiling-chunks with their texture complexity. And record this info in a manifest.json file.

On the client-side, our player consists of downloading tile chunks, decoding tile-chunks and rendering frames.

First, the player requests, downloads, and parses the manifest.json file. Second, the player spawns $tiling + 1$ threads, where $tiling$ is the number of tiles for a certain chunk. Each of the threads performs downloading, decoding tiling-chunks under control of Selector and producing frames into an individual buffer. Third, the player retrieves proper frames from buffers and merges these frames into a rendered picture to show.

We start to explain details about components as follows:

1) Selector. The Selector saves texture complexity of all tiling chunks after the player parses the manifest.json file. As mentioned in section 3.3, we only use texture complexity of tiling-chunks to select their quality level. When the Loader is ready for downloading a new tile chunk, it calls Selector with a tiling-chunk id. Then Selector select the proper quality level and return correct URL of next tiling-chunk to the Loader.

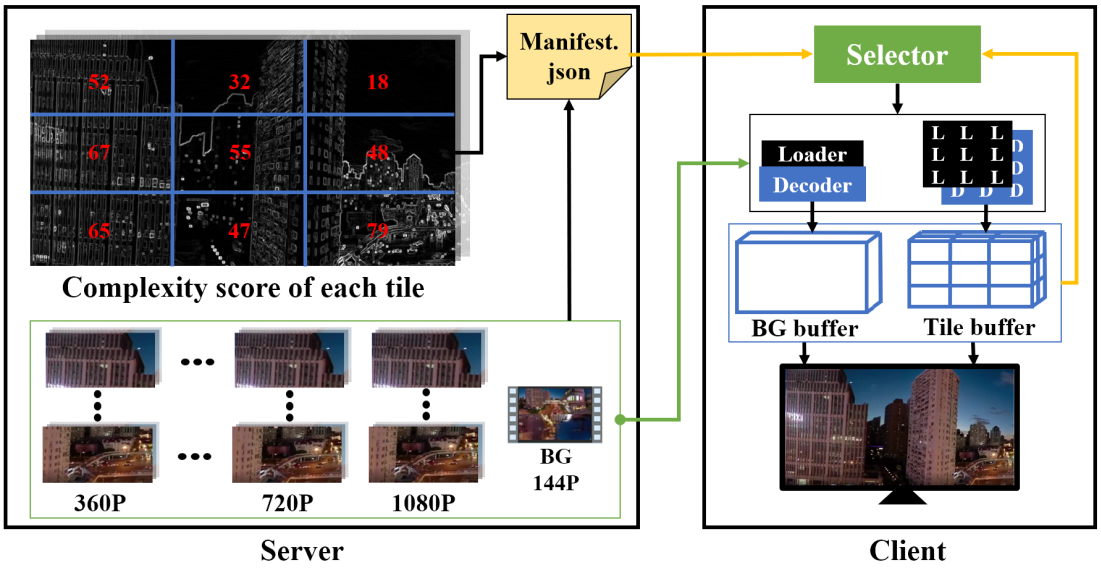

Fig. 4. The architecture of the system.

2) Loader and Decoder. There are two kinds of Loaders and Decoders(Loader and Decoder are binding together). The simpler one performs backup video streaming. It downloads and decodes chunks of a backup stream and produces frames into its buffer till the buffer is full. The more complex one refers to the remaining *tiling* Loaders and Decoders in Fig.4. The player monitors the current viewport to decide Loader working or sleeping. If a tile related to a Loader falls into the viewport, the Loader will be awaked by the player. Otherwise, the Loader will not be asleep until its Decoder finishes decoding a downloaded chunk. Once the Decoder finishes decoding a downloaded chunk and the player decides the Loader to continue working, the Loader will call Selector for downloading the next chunk. The Decoder decodes chunks downloaded by Loader and produces frames into its buffer till the buffer is full.

3) Renderer. It consumes frames in buffers. It is mainly implemented with libraries such as GLUT.

First of all, the Renderer needs to set the camera's position, orientation, and then, the size of the window and the radius of the ball, and other necessary information. After that, the frame buffer and rendering buffer will be initialized.

Then it will read the background image and set the filtering mode. Because the resolution of the background is low, there will be obvious jaggedness in the place where the texture is complex when playing in the panorama mode. Choosing the appropriate filtering method is essential. And the texture tiles are read similarly.

After reading the background and all texture tiles, Renderer starts writing to the frame buffer.

A ball will be drawn at the time of the first rendering, and then the content in the frame buffer will be used as a texture to paste on the ball, and then only need to update the texture of the ball regularly, the camera is located at the center of the ball, and it can achieve the effect of playing a panoramic video.

The Figure 5 shows the effect of the ball seen from the outside.

Fig. 5. The ball for rendering panoramic video.

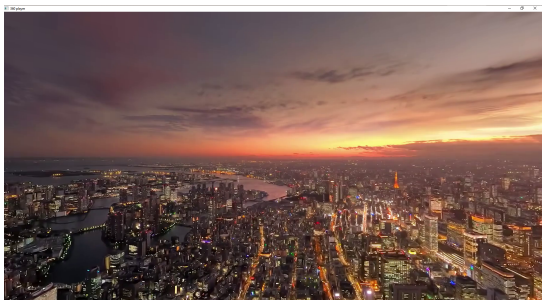

Fig. 6. A mixture of different resolutions. A quarter of the area uses the lowest resolution background image.

The Renderer will also always detect the user's mouse and keyboard input information, and constantly update the current field of view.

The final effect is shown in Figure 6. A quarter of the area uses the lowest resolution background image, but it is difficult for users to feel the difference.

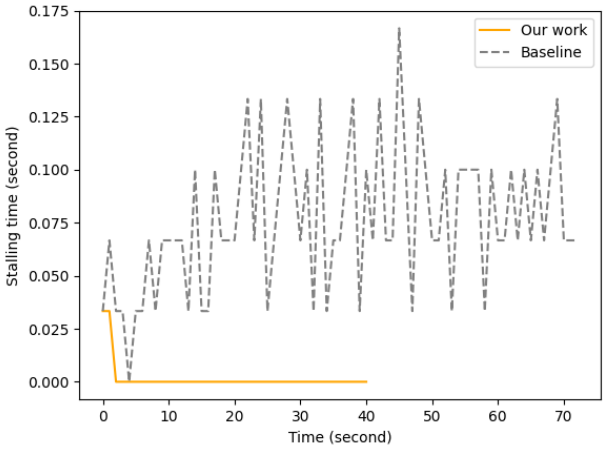

Fig. 7. Stalling time. The baseline lasts too long because of its delays

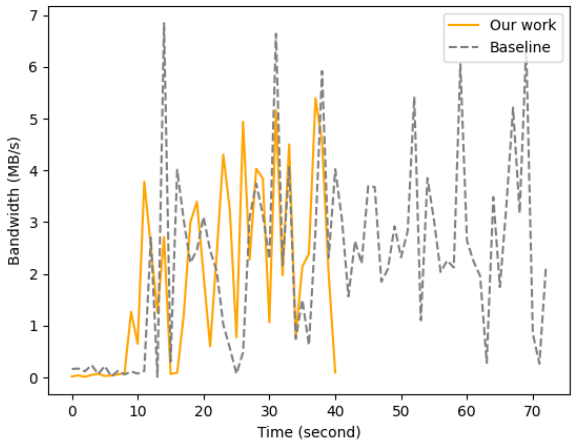

Fig. 8. Bandwidth records. Our system makes a little improvement than baseline and saves a lot of network traffic.

4 Evaluation

To evaluate performance of our work, we design a baseline and compare rebuffering time, average bandwidth, and PSPNR[10] between our work and the baseline.

The baseline also uses TBS schema for video streaming. Compared to our method, it requests tiling-chunks falling into the area of 4 times viewport(double width and double height).

We segment the 360-degree video into 1-second chunks and divide all chunks into 6*4 tiles. The baseline only uses the original 4k-resolution video. And the used resolutions of our system cross 120p, 480p, 960p, 2k and 4k(120p-resolution video as backup stream).

As shown in Figure 7, the baseline is put off because of its stalling delay. Compared to the baseline, our system only causes delay at the beginning of playing video. Furthermore, our system decreases the delay immediately but the baseline keeps delay, which leads to accumulated delays.

In Figure 8, the area under the curve is the total network traffic. Our system saves the traffic by requesting non-high-resolution chunks. It seems that our system only saves a little bandwidth, because the texture complexity of these tiling chunks is super high. On the other hand, it's proved that our system can reach the performance of a high-resolution baseline.

As mentioned in section 3.3, it is obvious the schema about *how to select a quality level of each tile* has crucial impact on the above evaluation metrics. So we test different behaviors of two functions (5) and (6). The result is shown in Figure 9.

Commonly, bigger PSNR means better quality of a picture. The picture whose PSNR is higher than 40dB is very similar to its original photo. The PSNR between 30dB and 40dB means people can tolerate such a bit of distortion. As shown in Figure 9, our system preserves the visual quality of the original 360-degree video. And different schema plays a vital role in visual quality. Besides, the trade-off between quality and bandwidth is directly impacted by the schema.

5 Conclusion

In this paper, we propose a new method to save bandwidth of 360-degree video streaming. Our main contributions are defining the texture complexity and designing the formula to select a proper resolution for each tiling-chunk.

We evaluate our prototype implement. And the results show our method ensure as similar as high-resolution perception quality, save a bit of bandwidth and decreasing the rebuffering time.

But there is a lot of spare space to improve performance. For example, we can introduce some network estimating algorithms like MPC[9] and Pensieve[11] for optimizing the formula for selecting a quality level of chunks in the future.

Acknowledgments

Most parts of this paper are written by Xingming Zhang. Section 3.1 is written by YuDong Wen. The part about Renderer in section 3.4 is written by JiaSheng Chen.

References

[1] Xianda Chen, Tianxiang Tan, and Guohong Cao. Popularity-aware 360-degree video streaming. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[2] Yu Guan, Chengyuan Zheng, Xinggong Zhang, Zongming Guo, and Junchen Jiang. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 394–407. 2019.

[3] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.

[4] Mijanur Palash, Voicu Popescu, Amit Sheoran, and Sonia Fahmy. Robust 360° video streaming via non-linear sampling. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[5] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 99–114, 2018.

[6] Yule Sun, Ang Lu, and Lu Yu. Weighted-to-spherically-uniform quality evaluation for omnidirectional video. *IEEE signal processing letters*, 24(9):1408–1412, 2017.

[7] Tian Wang. Research on panoramic video streaming over dash. 2019.

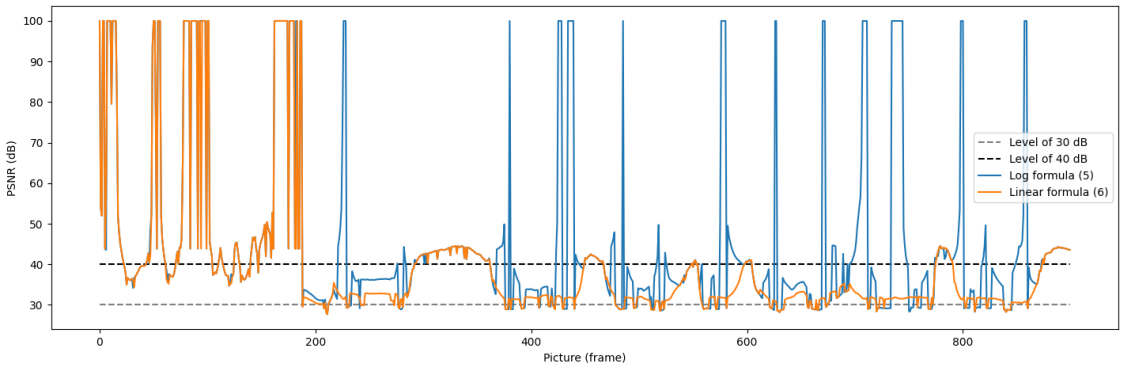

Fig. 9. PSNR records. PSNR of each frame is calculated according to the original 360-degree video frame.

[8] Jingxing Xu. Technical research and system implementation of adaptive streaming transmission for panoramic video. 2019.

[9] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[10] Wenjie Zhang. Research on panoramic videos adaptive transmission technology based on tile. 2019.

[11] Chao Zhou, Mengbai Xiao, and Yao Liu. Clustile: Toward minimizing bandwidth in 360-degree video streaming. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 962–970. IEEE, 2018.

[8] [7] [5] [2] [11] [1] [4] [10] [9] [6] [3]