TOWARDS A HIGH EFFICIENT 360° VIDEO PROCESSING AND STREAMING SOLUTION IN A MULTISCREEN ENVIRONMENT

Louay Bassbouss, Stephan Steglich, Sascha Braun
Fraunhofer FOKUS
Berlin, Germany
{firstname}.{lastname}@fokus.fraunhofer.de

ABSTRACT

Immersive video has been around for some time but only recently the technology became more popular since affordable cameras with sufficient resolution as well as stitching software with reasonable quality became available to allow professionals and interested amateurs to create 360° movies. Networks became fast enough to allow end-users to stream 360° video content to their devices. Hardware on TVs, smartphones and tablets is sufficiently powerful to handle the content and react on view changes without noticeable delay. Most efforts in this area, however, have been aimed at the technical challenges creating and viewing of 360° content. As 360° video is starting to reach a wider audience, the need arises to pay attention to the use of such content in a realistic commercial environment. For this, two issues need to be addressed. The efficient distribution of 360° content and the added-value that it can bring content providers. We will address in this paper two main challenges a) the efficient streaming of high quality 360° video content using existing content delivery networks (CDNs) and without the need for additional bandwidth comparing to traditional video streaming and b) the playback of 360° content even on devices with limited processing resources and programmatic capabilities.

Index Terms **—** 360° Video, Streaming, Immersive Media, Multiscreen

1. INTRODUCTION

Most public presentations of 360° innovations are based on immersive experiences and the use of VR headsets. While providing the most engaging experience, the use of such devices, more than any other content consumption device, isolates the user. Some solutions have been tested for VR environments (such as including other viewers as avatars in the scene), but most of them are specific to online use. At the moment scenarios with multiple users that are physically at the same site (such like a family or a group of friends watching a TV program together) are largely unexplored. The solution presented in this paper will address these kind of devices especially TV and streaming devices like HbbTV, Chromecast or Android TV.

Almost all current solutions stream the full 360° content to the end-user device, where only about 10% is actually presented to the viewer, while the other 90% are disregarded, causing a huge waste of bandwidth. All these solutions render the 360° video on the end-user device which requires much more computation and graphical processing capabilities comparing to classical video rendering especially when dealing with high quality content like UHD (4K, 8K, 16K). In this paper we will provide a new solution that streams only the visible field of view (FOV) to the end-user device which reduces the bandwidth requirement to the same level as for classical videos. Another advantage of the solution is that it pre-renders all relevant FOVs in advance and prepare them for streaming over existing CDNs. The client needs only to play a FOV video without performing any geometric transformation locally.

2. STATE OF THE ART

There are investigations in research and standardization as depicted in the subsections below to reduce the bandwidth and processing requirements but all of the existing solutions have limitations. We will compare these state of the art solutions according the following requirements (an ideal 360° video playout fulfills all these requirements).

- ***R1***: Stream only the visible field-of-view to the end-user device
- ***R2***: Adaptive bitrate streaming applicable on a single FOV
- ***R3***: Playback of 360° videos without additional processing on end-user device comparing to classical video
- ***R4***: Streaming of 360° videos without additional processing on server comparing to classical video
- ***R5***: Usage of content delivery networks (CDNs)
- ***R6***: No distortion in the visible FOV
- ***R7***: Photon-to-Motion latency under 20ms (it is the time needed for a user movement to be fully reflected on a display screen)
- ***R8***: No Additional Storage comparing to source 360° video content

2.1. Current solutions

Fraunhofer FOKUS Cloud-based 360° Video Playout for HbbTV [1]: Fraunhofer FOKUS provides a solution that renders 360° videos in the cloud and streams only the requested FOV to the client (HbbTV Application). It is the first solution that provides a 360° video playback in HbbTV but it is not scalable since a new rendering instance on the server is required for each client.

Transcoding and streaming-as-a-service for improved video quality on the web [2]: Bitmovin introduces a 360° video player for HTML5 browser based on MPEG-DASH. It basically applies adaptive streaming on the entire source 360° video and preforms the geometric transformation on the target device. Playing a FOV in good quality still needs to stream large amount of video data for the unseen FOVs.

HEVC Tiles [3]: HEVC provides improved compression rates for video in comparison with H.264/MPEG-4 AVC. It also allows easier access to sub-regions of the video frame, suitable for streaming only tiles that contain elements visible to the user. HEVC coding can only partly address the technical requirements for more efficient 360° video streaming. As 360° content is currently mostly represented as a distortion-mapped Equirectangular format, a view of the 'polar regions' of the virtual image spherical view can still require the transmission of a significant version of the source video, if only rectangular areas can be selected. The format also requires individual processes on the server for every active viewer of the content, reducing scalability.

MPEG-DASH SRD [4]: The Spatial Relationship Description (SRD) feature of the MPEG-DASH standard allows a video player to request spatial subparts of a particular video stream, which might be available in multiple resolutions. The feature extends the Media Presentation Description (MPD) specified in part 1 of MPEG DASH by describing spatial relationships between associated pieces of video content. The spatial relationship is represented by the relative position of the top-left corner and the size of the spatial object. Therefore, SRD is more suitable to describe region of interests (ROI) in panoramic videos but not for full-spherical immersive videos.

D'Acunto et al. use MPEG DASH SRD for zoomable and navigable video [5]: They present a video streaming client implementation that makes use of the Spatial Relationship Description (SRD) feature of the MPEG-DASH standard, to provide a zoomable and navigable video to the end-user. The video streaming client is implemented in JavaScript and extends dash.js, an MPEG DASH reference client implementation.

2.2. Comparison of 360° solutions

Current solutions that use spatial video content to transmit part of the video to reduce required bandwidth are not applicable to full-spherical video without streaming additional content in order to render the requested FOV. Furthermore, the client needs to process the received content on the end-user device. This may have implications on battery lifetime on mobile devices and performance issues on constrained devices. The current solutions that don't require processing on the client are applicable only to panoramic videos and support zoom to specific content. Below is a classification of existing solutions and an evaluation based on the eight requirements addressed in the begin of this section:

- ***S1***: stream the entire video to the client and preform processing on the client. Example: Bitmovin 360° player.
- ***S2***: stream parts of the video to the client and render requested FOV without additional processing. Example: HEVC tiles for panorama videos.
- ***S3***: stream parts of the video to the client and render requested FOV by processing the received content on the client. Examples: HEVC tiles for full-spherical videos with equirectangular projection.
- ***S4***: process the video on the server and send only the visible FOV to the client which needs to play it without any processing in same player as for traditional videos. Example: Fraunhofer FOKUS 360° Cloud Renderer.
- ***S5*** (solution of this paper): pre-render and store different FOVs of a 360° video. During streaming only pre-rendered FOVs stored on the server are transmitted to the client. Therefore, no processing is needed either on the server or on the client. There is also no need for additional bandwidth since only the requested FOV is transmitted to the client.

Table 1 shows a comparison of existing solutions according the requirements addressed above:

	S1	**S2**	**S3**	**S4**	**S5**
R1	--	+	-	++	++
R2	-	+	-	++	++
R3	--	+	+	++	++
R4	++	+	+	--	++
R5	++	++	++	--	++
R6	++	--	++	++	++
R7	++	-	-	-	--
R8	++	+	+	++	--

Table 1 Comparison of existing solutions (--very weak, -weak, +good, ++very good)

3. 360° VIDEO PRE-RENDERING SOLUTION

As we can see from the comparison of the different solutions, we are focusing in our solution on the following aspects:

- Support of best quality for a single FOV. For example, FOV in HD or UHD resolution.
- Use existing streaming infrastructures as for traditional videos and without additional bandwidth. In other words, the required bandwidth for transmitting a FOV is the same as for a classical video with same quality.
- Support of constrained devices or any device that can play a classical video without the need for additional processing on the client.

- Solution that scales without the need for additional processing on the server during streaming comparing to classical videos.

On the other hand, our solution brings some disadvantages regarding usability and the requirement for additional storage on the server. Storage is nowadays not that big issue comparing to the benefits it brings to save bandwidth and processing. The concept of storing pre-processed content is not new and is used in other domains for example adaptive streaming (like HLS or MPEG-DASH) which is supported nearly in any modern streaming infrastructure. In case of adaptive streaming, the same video content is available in different qualities. In our case as we will see later, we will pre-render and store FOVs with some overlap. The overlap-factor has impact on the required additional storage and level of granularity to navigate in the video. The usability aspect is the more important aspect we need to consider. The most relevant parameter that has direct impact on the usability is how fast the system reacts to a user movement to be fully reflected on the display screen. It is also known as motion-to-photon latency and is very important to consider in case of head-mounted displays which has a limit of 20ms. In case the entire 360° video is available on the client, the processing must be finished within this limit. This depends from the processing resources available on the client and the quality of the source video and single FOV. In our solution we are not addressing head mounted displays and we are focusing on devices with large screens which means automatically requirement for better quality for a single FOV. More investigation and improvement of the current solution to support head-mounted displays may be addressed in the future. For navigation, we are more focusing on input devices like TV remote control, keyboard or touch screen. Bringing 360° video experience to large screens is what many content providers and especially broadcasters are currently looking for. We already started pilots with some broadcasters using our solution on smart TVs. End-users can use the direction control keys (left, right, up and down) on the remote control to navigate in the video. The content provider can also offer a set of most relevant FOVs for selection which allows the end-user to jump directly to the FOV of interest. The result of the pilots shows that a motion-to-photon delay between up to 500ms is very acceptable when remote control is used as input device and navigation is done in large steps to certain hot spots. Lower motion-to-photon latency is preferred with small steps between neighbor FOVs.

3.1. Architecture

The architecture of our solution as depicted in Figure 1 comprises the four steps “Pre-processing”, “Storage”, “Streaming”, and “Playback”. During pre-processing step, a source 360° video will be pre-rendered with different FOV combination and the output FOVs will be stored somewhere in local storage. In next step, the created FOVs will be analyzed and a manifest file will be created. After the pre-processing is completed, all FOVs and the manifest file will to be made available on a storage server that is accessible by a CDN provider. For streaming the content to the client, the CDN infrastructure of the provider will be used. In the last step “playback”, the player reads and parses the manifest file to get information about available FOVs and how to access them. The player starts playback with the default FOV and listens to requests from input device to navigate to another FOV. Main components of the system are explained below in more details.

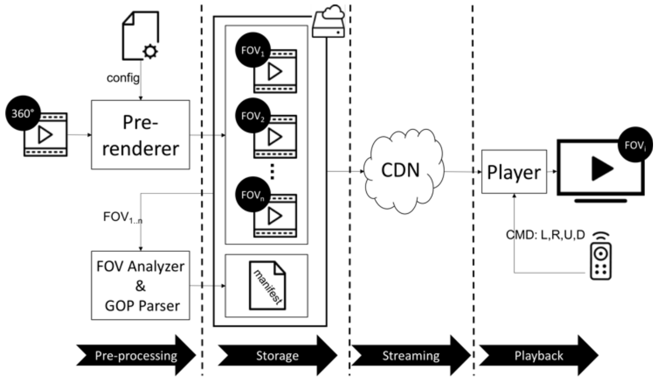

Figure 1 overall architecture

3.1.1. Pre-processing

The pre-renderer operates on the source 360° video and calculates the different FOVs depending from the configuration that is also passed as input. A FOV is defined using the four parameters (ϕ, θ, A_h, A_v):

- ϕ is the centre of the horizontal angle of view. $0 \leq \phi \leq (360° - A_h)$.
- θ is the centre of the vertical angle of view. $(-90° + A_v/2) \leq \theta \leq (90° - A_v/2)$.
- A_h is the width of the horizontal angle of view.
- A_v is the height of the vertical angle of view.

A_h and A_v remain constant during pre-rendering of the same zoom level. In this explanation we will consider a fix zoom level and keep A_h and A_v constant. Experience gained from our experiments shows that (A_h, A_v) = (106.7°, 60°) is good default configuration for displays with 16/9 aspect ratio (106.7° = 60°x16/9). Therefore, we will use in the rest of this paper (ϕ, θ) instead of (ϕ, θ, A_h, A_v) just for simplicity.

The horizontal angle between two neighbor FOVs $\phi 1$ and $\phi 2$ is defined as $\Delta\phi$ and the vertical angle between two neighbor FOVs $\theta 1$ and $\theta 2$ is defined as $\Delta\theta$. In other words, all neighbor FOVs of (ϕ, θ) along one direction (horizontal or vertical) are: ($\phi-\Delta\phi$, θ), ($\phi+\Delta\phi$, θ), (ϕ, $\theta-\Delta\theta$) and (ϕ, $\theta+\Delta\theta$).

The number of FOVs when θ remain constant and the horizontal angle of view ϕ changes is $N_h = \frac{360°}{\Delta\phi}$ and The number of FOVs when ϕ remain constants and the vertical angle of view θ changes is $N_v = \frac{360°}{\Delta\theta} - 1$. The total number of FOVs is $N = N_h \times N_v$.

The example depicted in Figure 2 has the following values:

- A_h = 106,7°, A_v = 60°.

- $0 \le \phi \le 345°$, $-60° \le \theta \le 60°$.
- $\Delta\phi = 15°$, $\Delta\theta = 30°$.
- $N_h = 24$, $N_v = 5$, $N = 120$.

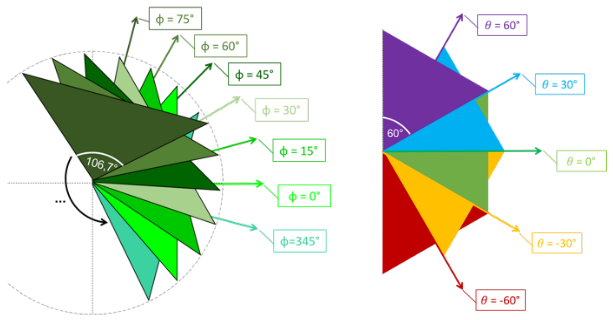

Figure 2: FOVs by changing horizontal angle of view ϕ and vertical angle of view θ

This configuration as shown in the example was used in most of our pre-rendered videos. In many videos where the main spectacle happens for vertical angle of view $\theta = 0°$, $\Delta\theta = 60°$ (with no vertical overlap) can be used instead of $\Delta\theta = 30°$ (vertical overlap = 50%) with no implications on usability. This will reduce total number of FOVs from 120 to 72. The FOVs for $\theta \neq 0°$can be also skipped if the main spectacle happens only for $\theta = 0°$. In this case the total number of FOVs can be reduced to 24. This was also the case for many 360° videos we got from broadcaster.

After pre-rendering all FOVs, they will be stored on the local system and each FOV will be parsed in next step. FOVs are video streams that can be played in any video player that supports the used container format and media codec. Examples for container formats are fragmented MP4 or MPEG-TS (MPEG Transport Stream) and for media codec H264. A FOV consists of a series of video segments or group of pictures (GOPs) that can be played independently from each other. If the player starts playing a GOP and the user requests to switch to another FOV, the player must first finish the current GOP and then switch to the next GOP of the target FOV. This is required because a GOP starts with a main/key frame that contains a fully specified picture and all following frames hold only the changes in the image from the previous frame. This means that the duration of a GOP has impact on the motion-to-photon latency. In average the motion-to-photon latency is increased by $\frac{GOP\ duration}{2}$. A suitable value for number of frames in a GOP is 10. This means a $GOP_{duration} = 333.333$ms for frame rate of 30fps and $GOP_{duration} = 400$ms for a frame rate of 25fps. The number of frames in a GOP has impact on the compression ratio of the used media codec.

During parsing step, the offset in bytes from the beginning of the corresponding FOV will be calculated for each GOP. All offsets will be stored in a manifest file in the same location as for FOV videos. In the player section we will explain in more details how the information from the manifest are used.

The pre-processing algorithm for pre-rendering and analyzing all FOVs is described below. The function ***render*** takes a 360° source video and angle of view as input and returns the video stream of the corresponding FOV as output. The ***save*** function saves a FOV or manifest file. The ***parse*** function takes the video stream of a FOV as input and returns the GOP offsets as output.

```
preprocess(V, $A_h$, $A_v$, $\Delta\phi$, $\Delta\theta$)
        $\phi_{min}$ := 0;
        $\phi_{max}$ := 360 - $A_h$;
        $\theta_{min}$ := -90 + $A_v/2$;
        $\theta_{max}$ := 90 - $A_v/2$;
        $\phi$ := $\phi_{min}$;
        manifest := Angle x Angle x $GOP_{index}$ $\rightarrow$ $GOP_{offset}$;
        i := 1;
        while $\phi \le \phi_{max}$ do
                $\theta$ := $\theta_{min}$;
                while $\theta \le \theta_{max}$ do
                        $FOV_i$ := render(V, $\phi$, $\theta$, $A_h$, $A_v$);
                        save($FOV_i$);
                        manifest[$\phi$,$\theta$] := parse($FOV_i$);
                        i := i+1;
                        $\theta$ := $\theta + \Delta\theta$;
                end
                $\phi$ := $\phi + \Delta\phi$;
        end
        save(manifest);
end
```

3.1.2. Storage and Streaming

The output FOVs and manifest of pre-processing step need to be available on a streaming server that is accessible from the CDN provider. In general, it is a static http server with range support. It is supported by most storage providers. In our experiments we used Amazon S3 as a storage service. Range requests are important in our case to access specific GOPs of a FOV instead of downloading the whole FOV. The HTTP header ***Range*** needs to be set and contains the from and to offsets in bytes. As alternative for Range requests each GOP can be stored in a separate file. After all FOVs and the manifest are available on a storage server, a CDN provider can be used for streaming. We used in our experiment Amazon CloudFront as a CDN. CloudFront can be easily configured to operate on Amazon S3.

To support adaptive bitrates, the generated FOVs can be converted to different qualities with different bitrates using the same tools that are used for traditional videos.

3.1.3. Player

The player constructs the final video stream to display from the FOV videos. There is no need to process the received video content before playback. The client platform needs only to provide an API that allows applications to send video segments (in our case are the GOPs) to the video decoder. In our implementation, we focused on web technologies and used the W3C Media Source Extension API (MSE) [6]. MSE API allows application to control the source buffer of a

HTML5 video object by appending, removing or replacing segments. There are three modules in the player:

- **Manifest parser**: The URL of the manifest file is the only input required in the player. As described above, the manifest contains all metadata of the video as well as all information of the available FOVs and the GOP offsets for each FOV. We use currently our own simple JSON format for the manifest but in the future it could be replaced by an extension of existing formats like Media Presentation Description (MPD) of MPEG-DASH. The Spatial Relationship Description (SRD) of MPEG-DASH is one of the candidates if it is extended in the future to also support spherical coordinates. A Web player can request the manifest file using a simple HTTP GET request.
- **Player and Buffer Control**: After the manifest is parsed, the player will be initialized with the default angle of view (ϕ_0, θ_0). Initial $GOP_{index} = 1$ if the user starts playback form beginning. GOP_{index} holds the index of the current GOP being played. In each step, the player requests a GOP from the server using a HTTP range request. From the $GOP_{index,}$ the player can read the ***from*** and ***to*** offset of the next GOPs of the current FOV from the manifest. To reduce the number of HTTP requests, the player may request multiple GOPs at same time. All GOPs will be added to the source buffer. Different caching algorithms can be applied and the user interaction behaviour can be considered. Until this step and before any user interaction, the player logic is the same as for classical videos.
- **User Input Control**: This module is responsible for navigating in the video. It receives a request from the input device for example TV remote control or keyboard and changes the FOV. The player updates its internal state with the coordinates of the new FOV and sends immediately a new HTTP request to get the next GOPs of the new FOV. Once the new GOPs are received, the GOPs for the old FOV will be automatically replaced.

Figure 3 shows an example with n FOVs each of them with m GOPs and with 4 user interactions.

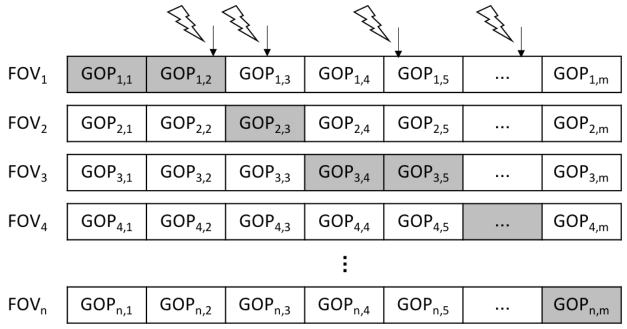
FOV1
GOP1,1 GOP1,2 GOP1,3 GOP1,4 GOP1,5 ... GOP1,m
FOV2
GOP2,1 GOP2,2 GOP2,3 GOP2,4 GOP2,5 ... GOP2,m
FOV3
GOP3,1 GOP3,2 GOP3,3 GOP3,4 GOP3,5 ... GOP3,m
FOV4
GOP4,1 GOP4,2 GOP4,3 GOP4,4 GOP4,5 ... GOP4,m
⋮
FOVn
GOPn,1 GOPn,2 GOPn,3 GOPn,4 GOPn,5 ... GOPn,m

Figure 3: Example with n FOVs with m GOPs

4. EVALUATION

We evaluated our solution (cloud pre-rendering) together with "cloud live rendering" and "client rendering" (e.g. YouTube and Facebook players) solutions according bandwidth, total costs, battery consumption and motion-to-photon latency.

Bandwidth: Figure 4 shows the required bandwidth for 8 different videos. The blue chart shows the bandwidth for steaming 4k source 360° video which applies to "client rendering" solutions and the orange chart shows the bandwidth required to stream a single FOV in HD quality for calculated from the corresponding video.

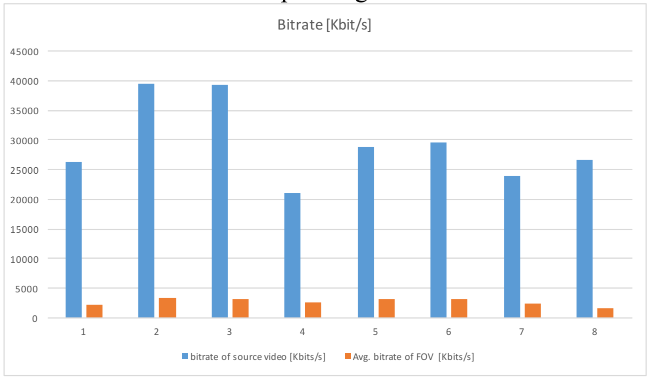

Figure 4 Bandwidth comparison

The required bandwidth is the same for cloud live-rendering and cloud pre-rendering since in both solutions stream only the requested FOV and not the whole 360° source video. With our solution we can deliver 360° video in the same quality by saving 10x bandwidth.

Total costs: Figure 5 shows a comparison of the three solution regarding total costs which include costs for storage, rendering and streaming. We considered in this evaluation the AWS prices (S3 for storage, EC2 for rendering and CloudFront as CDN).

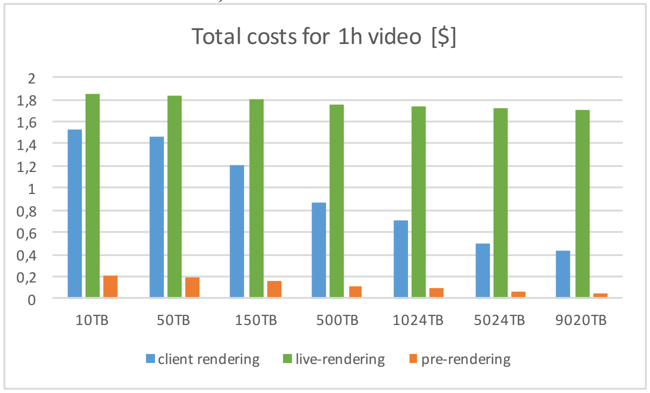

Figure 5 total costs

The costs are in USD per 1h video for different price levels. The blue chart shows the total costs for client rendering which are mainly the costs for streaming 4k content. The green chart shows the costs for cloud live-rendering. Most of the costs are for rendering the FOV in the cloud since for each client a new rendering instance with GPU is needed. The orange line shows the costs for our solution cloud pre-rendering. It includes the costs for pre-rendering the video which are one-time costs per video, the storage costs of all FOVs which is higher than the storage costs of the source 360° video but these are fix costs per month independent of the number of client and the streaming costs which are much lower than the costs for streaming the source 4k videos.

Battery lifetime: We will again compare the three solutions client rendering, cloud live-rendering and cloud pre-rendering. Since both cloud live-rendering and pre-rendering use the same client we will compare these two solutions to client rendering. The target device is a Samsung Galaxy Tab S2 Tablet, running Android version 6.0.1 and with battery capacity of 5870 mAh. As depicted in Figure 6 we can see that using our solution the battery level dropped from 100% to 85% after 1h10m37s and under the same conditions with client rendering it was 42m18s. This means with our solution the battery lifetime is about 39% than for client rendering.

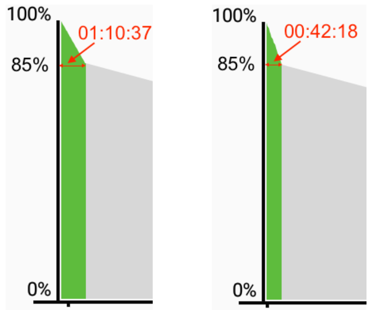

Figure 6 Battery lifetime a) cloud pre-rendering b) client rendering

Motion-to-Photon Latency: There are many factors that influence the motion to photon latency. In case of client rendering the motion to photon latency can be calculated from the decoding time of a single video frame, the time needed to capture user inputs (motions), the time needed to process a frame and calculate the projection and the time to switch the pixels on the display. We will consider a latency under 20ms as given for client rendering. For cloud live-rendering the network latency needs to be considered in addition to the motion to photon latency of client rendering. For pre-rendering solution the duration of a GOP needs to be also considered. For a GOP size of 10 frames and 30fps video, the GOP duration is 333,33ms. This means, in average the additional latency of pre-rendering solution is 166,67ms comparing to cloud live-rendering.

5. CONCLUSION

We presented in this paper a new solution for 360° videos based on pre-processing and streaming of individual FOVs. Our target devices are large displays where users expect best quality for a single FOV. The proposed solution in this paper was evaluated in context of different pilots together with broadcasters and content providers. The current implementation of the pre-processing part is available as Node.js program which generates the FOVs and the manifest and uploads all data to Amazon S3. A player implementation is also available using pure web technologies. Browser support of W3C MSE API is required. The web player was tested on following platforms: Chromecast (1st and 2nd generations), desktop browsers Chrome, Safari, Firefox and Edge, Chrome browser for Android, Amazon Fire TV and Android TV (in WebView), new series of LG and Samsung TVs that support MSE. We will continue working on improving the solution especially for reducing the motion-to-photon latency and to make smooth transition between the FOVs.

5. REFERENCES

[1] Fraunhofer FOKUS Cloud-based 360° Video Playout for HbbTV. Retrieved March 30, 2017 from https://www.fokus.fraunhofer.de/go/360

[2] Christian Timmerer, Daniel Weinberger, Martin Smole, Reinhard Grandl, Christopher Müller, and Stefan Lederer. 2016. Transcoding and streaming-as-a-service for improved video quality on the web. In Proceedings of the 7th International Conference on Multimedia Systems (MMSys '16). ACM, New York, NY, USA, , Article 37 , 3 pages. DOI: http://dx.doi.org/10.1145/2910017.2910637

[3] K. Misra, A. Segall, M. Horowitz, S. Xu, A. Fuldseth and M. Zhou, "An Overview of Tiles in HEVC," in IEEE Journal of Selected Topics in Signal Processing, vol. 7, no. 6, pp. 969-977, Dec. 2013.

[4] Niamut, Omar A., et al. "MPEG DASH SRD: spatial relationship description." Proceedings of the 7th International Conference on Multimedia Systems. ACM, 2016.

[5] Lucia D'Acunto, Jorrit van den Berg, Emmanuel Thomas, and Omar Niamut. 2016. Using MPEG DASH SRD for zoomable and navigable video. In Proceedings of the 7th International Conference on Multimedia Systems (MMSys '16). ACM, New York, NY, USA, , Article 34 , 4 pages.

[6] "W3C Media Source Extensions", Retrieved 10 November 2016, from https://www.w3.org/TR/media-source/.