ADAPTIVE STREAMING AND RENDERING OF STATIC LIGHT FIELDS IN THE WEB BROWSER

Hendrik Lievens[1,2,3], *Maarten Wijnants*[1,2,3], *Brent Zoomers*[1],
Jeroen Put[1,2,3], *Nick Michiels*[1,2,3], *Peter Quax*[1,2,3] *and Wim Lamotte*[1,2]

[1] Hasselt University – tUL
[2] Expertise centre for Digital Media, Wetenschapspark 2, 3590 Diepenbeek, Belgium
[3] Flanders Make

ABSTRACT

Static light fields are an image-based technology that allow for the photorealistic representation of inanimate objects and scenes in virtual environments. As such, static light fields have application opportunities in heterogeneous domains, including education, cultural heritage and entertainment. This paper contributes the design, implementation and performance evaluation of a web-based static light field consumption system. The proposed system allows static light field datasets to be adaptively streamed over the network and then to be visualized in a vanilla web browser. The performance evaluation results prove that real-time consumption of static light fields at AR/VR-compatible framerates of 90 FPS or more is feasible on commercial off-the-shelf hardware. Given the ubiquitous availability of web browsers on modern consumption devices (PCs, smart TVs, Head Mounted Displays, ...), our work is intended to significantly improve the accessibility and exploitation of static light field technology. The JavaScript client code is open-sourced to maximize our work's impact.

***Index Terms*—** JavaScript, IndexedDB, WebVR, HTTP Adaptive Streaming, MPEG-DASH.

1. INTRODUCTION

Manually modeling objects and scenes for inclusion in Augmented Reality (AR) or Virtual Reality (VR) environments - jointly denoted as eXtended Reality (XR) - using tools like Blender or Autodesk Maya can be very time-consuming and thus expensive. Scanning-based solutions like, for example, those based on Time-of-Flight (ToF) sensors [13], light fields [14, 3] or photogrammetry [15], allow for XR-compliant content acquisition directly from the real world with minimal manual modeling effort involved. All these approaches have their respective strengths and weaknesses. In this paper, we focus exclusively on light fields by contributing a Proof-of-Concept implementation (and performance evaluation thereof) that allows users to consume static light fields in contemporary web browsers.

A static light field can conceptually perhaps best be explained by comparing it to traditional 2D images. Whereas a 2D image captures only the *intensity* of the light rays that pass through a physical scene, a light field also captures the *direction* in which those light rays travel. A *static* light field does so at a particular moment in time, giving rise to a static, inanimate virtual scene. *Dynamic* light fields (sometimes also called light field video) attach a temporal component to the light field concept, yet are beyond the scope of this article. Static light fields are a cost-efficient technology to digitalize intricate real-world objects with idiosyncratic visual properties such that these objects can then be rendered in photorealistic visual quality in virtual environments. As such, static light fields have value-adding applications in various domains, including education (e.g., visualize welding seams in manufacturing training), cultural heritage (e.g., visualize fragile archaeological findings) and entertainment.

In this paper, we present the design, implementation and performance evaluation of a web-based static light field consumption system. The proposed system includes both a network streaming component (to dynamically deliver the light field data over the transportation network) and a real-time rendering component (to visually present the static light field to the user). We discuss the challenges that a web browser execution environment poses for handling and rendering static light field data, together with the mitigation actions that we adopted to overcome these challenges. The resulting client source code of our web-based light field consumption client is publicly available [5]. Finally, our performance evaluation results provide insights into potential optimization strategies to maximize rendering performance of static light fields in web browsers.

The remainder of this paper is organized as follows. Section 2 provides the necessary background information and briefly reviews related work pertaining to the capturing, rendering and network streaming of static light fields. Section 3 gives an overview of the proposed web-based static light field consumption system. Section 4 details the JavaScript-based client implementation. Section 5 presents decoding performance results for static light field data and discusses the potential impact of these results on the user's Quality of Experience (QoE). Finally, we draw our conclusions and suggest avenues for follow-up research in Section 6.

2. BACKGROUND AND RELATED WORK

2.1. Capturing and Compressing Static Light Fields

Static light fields are typically captured as a collection of 2D images (called *source views*), each covering the physical object or scene from a different vantage point. However, storing each source view as a separate image quickly leads to enormous file sizes. Therefore, compression of static light field datasets is an active research domain. While specialized compression schemes for static light fields are being researched (e.g., JPEG Pleno [4], Steered Mixture-of-Experts [22]), another option is to apply commodity video compression to the collection of source views [11, 23]. While this latter approach typically suffers from lower compression efficiency compared to specialized compression schemes, it has the advantage that it can exploit software and hardware that is commonly available in consumer devices (e.g., dedicated video decoding chipsets on GPUs). Given that the perspective change between spatially adjacent source views is typically small, especially for dense datasets, multiview extensions to traditional video codecs (e.g., MVC for H.264/AVC [10], MV-HEVC [1]) are being considered for light field compression in academic research; unfortunately, these multiview extensions lack wide-spread implementation in consumer-grade hardware. As an important goal of our research is to maximize the deployability and accessibility of static light fields, we exploit the widely adopted (non-multiview) H.264/AVC video codec for static light field compression purposes (see later). A full survey on light field coding strategies is provided by Conti et al. [2].

2.2. Rendering Static Light Fields

The visualization of a light field can be achieved by switching between the densely captured 2D images. The change in perspective around the recorded object will determine which 2D image needs to be decoded and rendered in the 3D virtual environment. This approach achieves a high-quality and photorealistic visualization of the object because it relies solely on real recorded images, without the need of underlying geometry and appearance modeling [17]. The 2D image is projected onto a proxy geometry, which is an approximation of the geometry of the 3D object. In its most simple form, the proxy geometry can be approximated with a 3D plane, however improving the geometrical resemblance of the proxy geometry will increase the quality of the visualization. More complex rendering approaches will generate new virtual viewpoints by tracing rays for each pixel and extract and interpolate the correct color values from neighboring captured viewpoints [7, 9, 12, 16]. The quality and the rendering speed of the light field is highly depended on underlying light field representation and the selected coding and compression scheme [2]. Debevec et al. [20] proposed an end-to-end recording and rendering system for omnidirectional static light fields. The result is a real-time immersive viewing experience in VR with 6D of freedom. They achieve this by proposing a custom decoder that only works on high-end consumable hardware. Other techniques focus on improved view interpolation for the virtual viewpoints, based on advanced view synthesis or deep learning techniques [6, 19]. Although those recent advancements achieve high quality, they often require dedicated hardware and custom decoders that only work well on high-end consumable hardware. This is not readily available in a web context. In this paper we return to a more classic approach of image based rendering that relies solely on the 2D recorded images. This maps well to both the widespread available video codecs as well as to the standard graphics pipeline available in the web. The proposed image-based rendering visualization will act as a proof of concept and its quality can be easily improved in the future once web-based implementations of view synthesis techniques become available.

2.3. Adaptive Network Streaming of Static Light Fields

Like any other media type, static light field datasets can be distributed over the transportation network using an opaque bulk download (which might or might not be progressive). This approach however lacks flexibility (e.g., no options to respond to fluctuating network conditions), leads to high start-up latencies (in the case of non-progressive downloads), and might overload storage-constrained consumption clients (e.g., stand-alone Head Mounted Displays like the Oculus Quest have limited storage space which will quickly overflow if integral static light field datasets need to be installed on it). A more flexible solution therefore is to resort to the HTTP Adaptive Streaming (HAS) paradigm (and its standardized MPEG-DASH implementation), which in recent years has become the de facto standard to deliver audiovisual content over the Internet (e.g., Netflix, YouTube, . . .) [21]. In HAS, the to-be-streamed media content is segmented in chunks that can be streamed and consumed independently of each other. These chunks can additionally be provided in multiple qualities, such that a client-side Adaptive BitRate (ABR) controller can decide at run-time (e.g., based on prevailing network conditions) in which quality to fetch individual segments.

Seminal research to map the MPEG-DASH specification to the adaptive streaming of static light fields has been conducted by Wijnants et al. [23]. In this paper, we adopt this mapping and extend it by not only considering the source view essence but also source view metadata (i.c., background masks to allow for efficient and accurate alpha compositing of static light fields in larger 3D scenes, see Figure 6).

3. SYSTEM OVERVIEW

A high-level description of the proposed web-based system for the adaptive streaming and rendering of static light fields is as follows. The light field datasets are stored on a plain HTTP server (e.g., Apache, nginx, ...) and are transferred to the web browser via MPEG-DASH adaptive streaming. To facilitate this process, the composing source view images of a light field dataset are spatially subdivided and then video encoded (using H.264/AVC[1]) in multiple qualities to enable adaptive streaming. This leads to multi-quality MPEG-DASH Media Segments which each hold a predefined number of spatially adjacent source views. The spatial clustering scheme to subdivide source views into MPEG-DASH segments is configurable and can have a substantial impact on both video compression efficiency [23] and on rendering performance (see Section 5). In case background masks are available for a particular dataset, these mask images are encoded as a separate video and treated identically to the actual source view essence in terms of MPEG-DASH-compatible content preparation. This leads to the source view essence and its associated background mask to be represented by separate AdaptationSets in the MPEG-DASH MPD manifest describing the static light field.

The processing and rendering loop of our system behaves as follows. To be able to render the static light field, a number of files first need to be downloaded from the content server: a file containing info about the camera that was used to capture the source views, a file containing the adaptive streaming metadata (i.e., the MPEG-DASH MPD file), and the setup data including the LFS center and rotation of the object (see Section 4). Once this data is locally available, a plane is shown in the 3D scene on which the light field source views will be projected. At each render pass, this plane will be updated based on the current location and viewing angle of the virtual camera. This potentially involves updating the source view that is shown on the plane, updating the rotation of the plane so that it keeps on pointing towards the virtual camera, and applying the alpha mask that corresponds to the currently shown source view (if available). To decide which source view must be projected on the plane, we determine which source view has the smallest angle to the virtual camera when the center of the static light field is taken as the center of an imaginary sphere. By using the angle between two points instead of the distance between them, we have no issues when the virtual camera distance to the static light field is large or when the source views are not perfectly aligned on a sphere (which could result in one perspective being closer because it is further outside compared to its neighboring perspective).

[1]Although newer video codecs exists, most notably H.265/HEVC and H.266/VVC, these codecs lack pervasive (hardware-accelerated) web browser support. In contrast, H.264/AVC is broadly implemented in hardware and is supported by all major contemporary web browsers.

4. CLIENT IMPLEMENTATION

The Proof-Of-Concept implementation consists of a static light field element, which can be placed in a three.js scene at will. The creation of this element requires different factors which will be discussed in detail throughout this section.

A first step towards achieving a static light field element is getting the required content. As mentioned before, this happens via MPEG-DASH adaptive streaming. In the implementation, fetching these segments can happen in 1 of 2 ways. The first way makes use of a strategy which fetches segments in a specified order while no other segments are required by the user. This strategy can be swapped with any other strategy that conforms to the interface. Currently this interface consists of one function which gets halted when the user requires a specific segment and continues when said segment is fetched. The second way fetching can happen has already been mentioned, being a fetch of a segment that is needed by the user. This fetch strategy can, in the same way as the ABR-strategy, be swapped with another strategy. For this Proof-Of-Concept implementation, we implemented an ABR-strategy which first fetches all segments in the lowest quality available, then fetches all segments in the next higher quality and keeps working its way up until all segments have been fetched in the highest quality available. This strategy is not optimized for realistic usage, but it allows us to test multiple factors within our implementation. To visualize which segments have been fetched or which segments are being fetched, we use small orbs to represent the different source views of the light field dataset (see, for example, Figure 1). The color of these orbs changes based on the highest quality that has been fetched or is being fetched.

Similar to segments being saved in memory, some segments that have been decoded are stored in cache which allows for fast access. But before a segment can be decoded, we fetch the required m4s file from local memory and concatenate it to the corresponding initialization mp4 file which is saved in code. This concatenation is then passed to the video decoder (via the experimental WebCodecs API [18]) which decodes the frames into ImageBitmap objects and places these objects in cache. The contents of this cache can also be visualized using the same orbs as mentioned in the previous paragraph.

The just mentioned WebCodecs API is instrumental to our approach. Without it, there is no way of precisely accessing the individual frames which jointly constitute a video fragment in a web browser context. While frame-precise access is typically not needed for watching traditional video content, it is an absolute necessity in our implementation, as each frame corresponds with a specific light field source view.

With all needed data available in cache, some steps are required to create a realistic feeling to the user. The first step is to render the required texture, which has been saved in cache, to a plane. In this step, several other things occur such as the application of the background mask (if available) and the deprojection of the image. In the future, also interpolation between different source views could be considered (for those scenarios where the user's viewport falls in between multiple source view perspectives). We apply these things in custom shaders which allows for more control of what happens than standard three.js. The deprojection plane is also rotated to face towards the camera and it is possible to apply a small correction to the plane if needed. This correction can fix irregularities in the original dataset which can occur through small errors in the setup used to create the dataset. Next to fixing these small irregularities, it is also possible to provide a light field sphere (LFS) center which specifies the actual center point of the object. Rotating around this center point instead of around the null-point leads to a more accurate representation of reality. Rotation can also be applied to the object to change its standard orientation in a given scene.

The system as described thus far allows for viewing the light field from one viewpoint at a time (cf. consumption on a 2D monitor). Our implementation however also allows for VR usage. VR does not change the architecture of our light field element tremendously. If we look at VR as 2 different perspectives, we can treat each perspective as a separate one which results in it behaving exactly the same way as without VR. The only difference being that we have to be careful with what gets rendered when. We solved this by making use of a callback provided by three.js which gets called right before an object gets rendered. Since a view as seen from each eye is treated as a different render pass, this allows for per-view rotation and texturing of the plane (e.g., it is often needed to show a different source view for the left versus right eye).

Up to now we have explained how content is acquired and how it is managed client-side. However, to create an interactive experience, we need a way to show correct light field perspectives on screen while allowing movement to the user. In our implementation, we chose to use three.js since it provides out-of-the-box support for XR and it allows for ease-of-use while still being very flexible. The final product is thus a three.js element that allows for placement in a scene at will. The only parameters required from the user are a position in the scene, and a string representation of the dataset that is needed from a server. Next to these mandatory input values, there are other parameters that can be specified by the user. These parameters include the back-end URL, a maximum of decoded segments in cache and debugging parameters. Given these values, the element will setup everything needed to create a static light field in an interactive three.js scene.

Next to the possibility to show a static light field in three.js, the code was implemented to support objects on which a mask can be applied to filter out parts of the frame. The mask frames consist of pixels whose RGB values denote the alpha value that needs to be applied to the corresponding pixel in the frame holding the light field source view (see Figure 6).

5. PERFORMANCE EVALUATION

5.1. Experimental Set-Up

The desktop computer, used for the bulk of the evaluation, is equipped with an intel i9 9900k processor, a NVIDIA RTX 3080 graphics processor, 32GB of DDR4 system memory and has a 1TB NVME SSD. The system runs Windows 10 version 21H1 and Google Chrome version 93.0.4577.82 was used to perform the experimentation.

Even though the implementation fully supports ABR strategies, we will not include these in our tests. The evaluation solely deals with segments that have been downloaded and stored (still as encoded blobs) in the web browser's cache to explicitly evaluate the client-side decoding and rendering performance.

5.2. Decoding Performance Break-Down

To fully be able to understand the system's performance, a deeper dive into the actual render and update loop is required. There are a few factors that greatly impact the application's performance, both implementation wise as well as the chosen dataset to visualize.

When the user is moving, a corresponding view-point will need to be decoded. The system relates this view-point to the MPEG-DASH segment that contains the corresponding source view and requests for that segment to be decoded by the hardware-accelerated video decoder that is exposed to the browser, in this case the chip present on the NVIDIA graphics card. The encoded blob is transferred from host-space to the video decoder. Note that a segment contains multiple, spatially related, view-points and thus a single decode pass results in multiple decoded source views.

The Firetruck dataset (see Figure 1) was used to evaluate the decoding performance. It consists out of 15 concentric rings, each containing 120 views for a total of 1800 views. The dataset is encoded in H.264 in both 3840x2160 with a bit rate of 24Mbps and 1920x1080 with a bit rate of 12Mbps. It is split into segments, each containing 30 spatially related source views, where the first frame is an I-frame (intra), and the 29 subsequent frames are P-frames (predictive). The light field is packaged into rectangles, containing 6 horizontal and 5 vertical viewpoints. The virtual camera follows a pre-defined path horizontally around the light field, triggering numerous decodes. Table 1 contains the average decoding times for a single segment, containing 30 light field source views in this case. Table 2 shows the average frame time for a single render pass as well as the average frames per second for both quality representations.

Of course, this is only one piece of the puzzle. Another factor that impacts the application's performance is the way that these decoded frames are transformed to a WebGL texture. Currently there exists no way for the decoded frame to stay in GPU memory and needs to be transferred back to the

Dataset	Decode time (segment)
Firetruck 1920x1080	107.81 ms
Firetruck 3840x2160	266.28 ms

Table 1. Decoding time for a single MPEG-DASH segment in the Firetruck dataset, expressed in milliseconds. A single segment contains 30 views: 6 horizontally and 5 vertically.

Dataset	Avg. Frame Time	Avg. Frames Per Second
Firetruck 1920x1080	3.59 ms	278.27 FPS
Firetruck 3840x2160	8.87 ms	112.66 FPS

Table 2. Rendering performance of the Firetruck dataset.

dataset	Avg. Bitmap time	Overhead
Firetruck 1920x1080	10.96 ms	10.96 %
Firetruck 3840x2160	17.12 ms	6.43 %

Table 3. The average time it takes per segment to move decoded light field source views from the video decoder to a bitmap in host-space. The overhead represents the percentage of time this takes w.r.t. to the total decoding time, as presented in Table 1.

host, where it is converted to a bitmap, which is then again uploaded to the GPU in a WebGL context to be rendered. The creation of the bitmap and the transfers necessary to facilitate this impose a significant but unavoidable overhead on the system. As can be seen in Table 3, the creation of these bitmaps can result in an overhead of up to 11 %. Note that the 4K dataset has a smaller overhead. This indicates that the time to create bigger bitmaps rises less rapidly compared to the time it takes to decode bigger light field source views.

5.3. Segment Layout Comparison

Not only the application's implementation dictates its performance, also the way datasets are structured and consumed have a (significant) impact on overall application fluidity. To demonstrate this, the aforementioned Firetruck dataset has been packaged in three different ways, as shown in Figure 1. The three packaging strategies are:

3x10 Segments hold 3 views horizontally, 10 views vertically.

10x3 Segments hold 10 views horizontally, 3 views vertically.

6x5 Segments hold 6 views horizontally, 5 views vertically.

All three packaging strategies are evaluated using the exact same set-up. The virtual camera follows a pre-determined path, which is shown in Figure 2. Sideways movement is followed by upwards motion simulating the user's interest to look at the object from above. Then the user moves sideways again and ends the path by moving down to the vertical center to conclude the experience. During this interaction, every

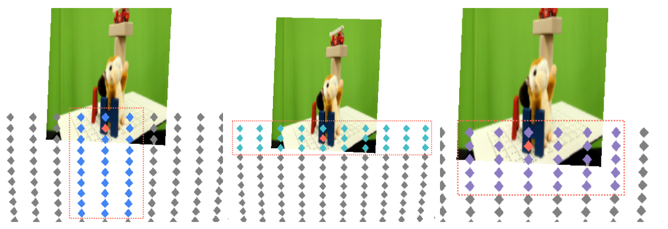

Fig. 1. The three packaging strategies. From left to right: 3x10, 10x3, and 6x5. The orange dot represents the currently rendered view-point.

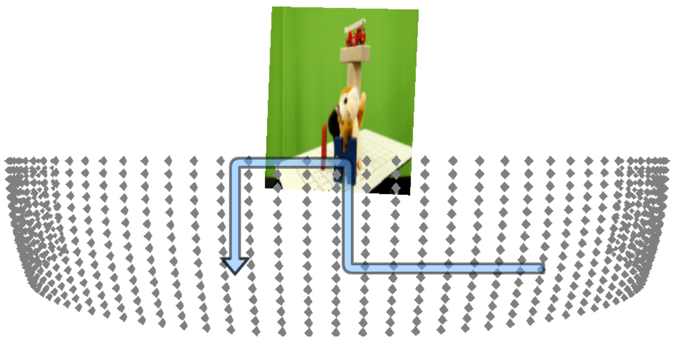

Fig. 2. Segment layout comparison: involved virtual camera path, containing both horizontal and vertical motion.

render pass is timed. The measurement contains the complete render loop, consisting of IndexedDB read operations, video decoding, texture application, and projection. The actual frame times are the average of 5 individual test runs per strategy. The test is concluded after 1000 rendered frames.

Figure 3 shows the average frame times for the **3x10** packaging strategy. The graphs also include a red line at the 11.11 ms mark, representing the 90Hz threshold, which is a typical rendering target for VR consumption. At the start there is a significant spike, attributed to the fact that the application is still starting and is performing some bookkeeping and caching operations. This is followed by a period of jittery behaviour. The explanation for this behaviour is rather simple when looking at Figure 1 and Figure 2 again. The comparatively small width of this packaging strategy in conjunction with the horizontal movement at the beginning of the virtual camera path triggers numerous segment decodes. These are clearly visible in the graph. However, when the first vertical part of the path is reached (around frame 500), the frame times stabilize due to the fact that the entire upwards motion is contained in a single segment, thus eliminating decodes along the way. This pattern is then repeated again for the final horizontal (jitter) and vertical (stable) part of the camera path.

As expected, when changing the packaging strategy to contain more horizontal bias, one can clearly see the more stable behaviour in the beginning of the graph in Figure 4. Note that there is a tiny amount of peaking behaviour around

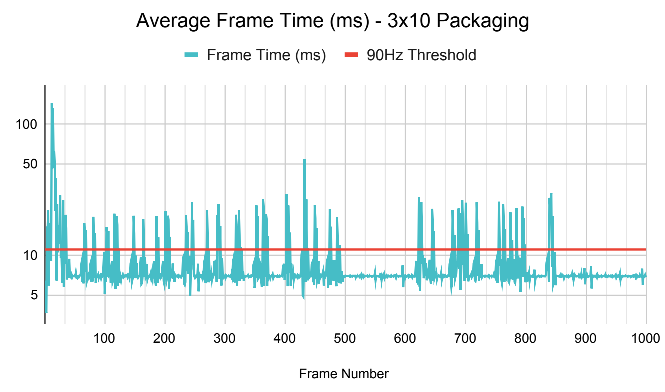

Fig. 3. Average Frame Times for the **3x10** packaging strategy.

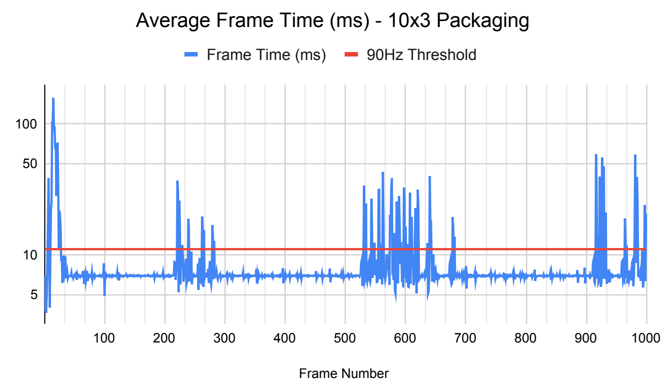

Fig. 4. Average Frame Times for the **10x3** packaging strategy.

rendered frame 220-270 due to the camera crossing a horizontal segment boundary. The initial upwards motion causes, as expected, a lot of stuttering because of the severe lack of vertical view-points present in the segments. This pattern also repeats itself again for the final horizontal (stable) and vertical (jitter) part of the camera path.

The third strategy, **6x5**, contains a rather balanced amount of horizontal and vertical view-points. Figure 5 no longer shows very distinct periods of jittery behaviour, however spikes are still visible because decoding operations remain required when crossing horizontal or vertical boundaries. The main benefit of this strategy is that the stuttering is spread out more uniformly,

The key take-away from this evaluation is that packaging strategies and the consumption path are tightly bound. Horizontal motion benefits from wider, smaller segments, while vertical motion benefits from narrower, taller segments. If the consumption pattern is not known, a segment that contains a decent balance of width and height ensures an overall more uniform rendering performance.

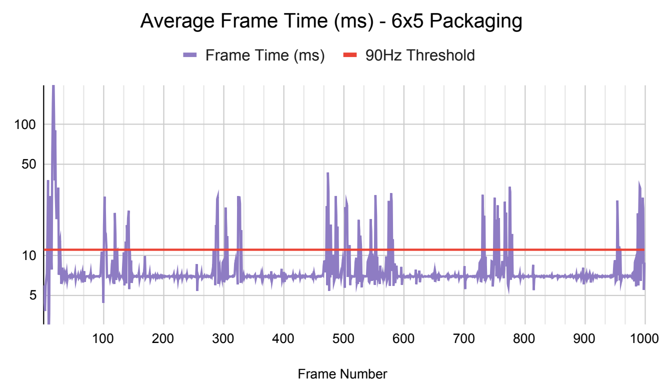

Fig. 5. Average Frame Times for the **6x5** packaging strategy.

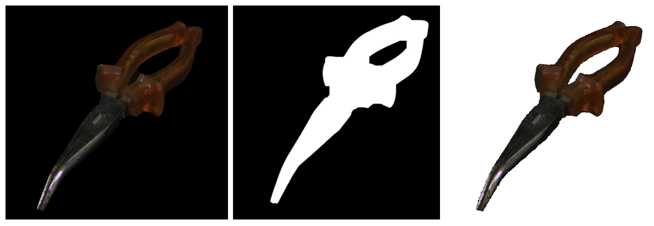

Fig. 6. The left view is the captured light field data. The center view is a binary mask to overlay on the light field data. Unnecessary data is blended, based on the transparency of the pixels in the mask, while the black pixels are cropped away. The right view shows the result of the masking operation.

5.4. Masking Overhead Evaluation

Our implementation supports static light field datasets that contain alpha mask metadata (Figure 6). When applying such alpha masks, the irrelevant information in the source view image gets cropped away by performing a blending operation in a pixel shader. The end result is thus a light field that only shows the object of interest and none of the background that might be captured in the source view images. This is especially useful for bringing real-life objects in the virtual world without expensive geometry scanning techniques.

To assess whether this technique, involving an additional decoding step and a blending operation, deteriorates rendering performance, a comparative evaluation was performed. In this set-up a 1920x1080 hemispheric dataset, Pliers, is consumed by a virtual camera following a fixed path. This path again consists out of 1000 total rendered frames, and the results are averaged over 5 runs. In the first scenario, the system reads the segment containing the corresponding masks from IndexdDB, decodes it and applies the masks, while in the second scenario, these masking segments are discarded.

The results, as seen in Table 4, show a fairly small difference in the attained frames per second. A possible explanation is that the segments, containing the masks, are rather

Dataset	Without Masking	With Masking
Pliers 1920x1080	138.63 FPS	131.90 FPS

Table 4. Rendering performance comparison between masking and non-masking (in average Frames Per Second).

efficient to decode as most frames only contain black and white pixels resulting in a lower bitrate. Another candidate that is imposing a cost on the application's performance is the creation of the aforementioned bitmaps. When masking, a bitmap needed for both the light field source view and its mask, thus effectively doubling the time spent on bitmap operations. Still, the cost of the additional mask bitmapping seems small. All in all, the frame rate of the visualization of the Pliers dataset with masking is more than suitable for VR consumption.

6. DISCUSSION AND CONCLUSIONS

The presented system is a proof of concept, containing only the bare necessities to adaptively stream and render a static light field in the web browser. As such, the performance results presented in Section 5 represent the worst-case scenario (from a system optimization perspective). That being said, the attained performance is still surprisingly decent. This is especially true when considering the bulk of the evaluation was performed with highly demanding 4K datasets in a web-based environment, utilizing an experimental video decoding API.

As it stands, the implementation does not take the user's consumption profile into account. Segments are only scheduled to be decoded when the user's viewport has already entered the spatial area that is covered by this segment. There is a lot of value in predicting the user's future movement by extrapolating past movement. Doing so would allow the system to pre-fetch and pre-decode segments that will be needed in the (near) future. This, in theory, should allow for even more stable behaviour when interacting with the light field. To make matters even more interesting, historical consumption data of the light field, averaged over multiple users, could reveal prototypical consumption patterns and provide clues on how to most efficiently pack view-points into segments.

Other performance benefits could be found when eliminating the unnecessary transfers between host-space and graphics device-space. The proposed WebGPU standard [8] boasts support for so called *GPUExternalTextures*, which keep the decoded video data on the graphics device. As prior evaluation has shown, this rather simple change can improve the decoding and rendering speed with up to 11% (see Table 3).

The performance evaluation presented in this paper was carried out on contemporary desktop hardware, which has shown to possess enough horsepower to run this system at acceptable frame-rates. The nature of the target platform, i.e. the web browser, however intrinsically opens the door to heterogeneous consumption devices, such as mobile devices or stand-alone AR or VR head-mounted displays. The explicit evaluation of the system on these devices is not in scope for this paper, yet it has been technically validated that out-of-the-box deployment is a given on these types of devices if they support a version of Google Chrome with the Experimental Web Platform features flag enabled.

Last, but not least, the modularity of the implementation also has clear benefits. One can include a light field visualization on a web page, e.g. a product showcase, but also include it in other 3D (web) applications, e.g. a scene consisting out of both classical 3D geometry and multiple light fields.

All in all, we have shown that a web-based static light field renderer is not only technically feasible, but also performs well in different circumstances. We envision that further development, experimentation and evaluation of this technology can lead the way to more accessible exploitation of static light field technology, especially on the web.

7. REFERENCES

[1] Waqas Ahmad, Roger Olsson, and Marten Sjöström. Interpreting Plenoptic Images as Multi-view Sequences for Improved Compression. In *Proceedings of the IEEE International Conference on Image Processing (ICIP 2017)*, pages 4557–4561, Sept 2017.

[2] Caroline Conti, Luís Ducla Soares, and Paulo Nunes. Dense light field coding: A survey. *IEEE Access*, 8:49244–49284, 2020.

[3] Abe Davis, Marc Levoy, and Fredo Durand. Unstructured Light Fields. *Computer Graphics Forum*, 31(2pt1):305–314, May 2012.

[4] Gustavo De Oliveira Alves, Murilo Bresciani De Carvalho, Carla L. Pagliari, Pedro Garcia Freitas, Ismael Seidel, Marcio Pinto Pereira, Carla Florentino Schueler Vieira, Vanessa Testoni, Fernando Pereira, and Eduardo A. B. Da Silva. The JPEG Pleno Light Field Coding Standard 4D-Transform Mode: How to Design an Efficient 4D-Native Codec. *IEEE Access*, 8:170807–170829, 2020.

[5] EDM-Research. SLF4Web v0.1.0. `https://zenodo.org/badge/latestdoi/432214902`, 2021.

[6] John Flynn, Michael Broxton, Paul Debevec, Matthew DuVall, Graham Fyffe, Ryan Overbeck, Noah Snavely, and Richard Tucker. Deepview: View synthesis with learned gradient descent. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[7] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 43–54, New York, NY, USA, 1996. Association for Computing Machinery.

[8] GPU for the Web Community Group. WebGPU specification. `https://gpuweb.github.io/gpuweb/`, 2021.

[9] Aaron Isaksen, Leonard McMillan, and Steven J. Gortler. Dynamically reparameterized light fields. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 297–306, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[10] Péter Tamás Kovács, Zsolt Nagy, Attila Barsi, Vamsi Kiran Adhikarla, and Robert Bregović. Overview of the applicability of H.264/MVC for real-time light-field applications. In *2014 3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*, pages 1–4, 2014.

[11] Péter Tamás Kovács, Alireza Zare, Tibor Balogh, Robert Bregović, and Atanas Gotchev. Architectures and Codecs for Real-Time Light Field Streaming. *Journal of Imaging Science and Technology*, 61(1):10403–1–10403–13, Jan 2017.

[12] Akira Kubota, Aljoscha Smolic, Marcus Magnor, Masayuki Tanimoto, Tsuhan Chen, and Cha Zhang. Multiview imaging and 3DTV. *IEEE Signal Processing Magazine*, 24(6):10–21, 2007.

[13] Elise Lachat, Helene Macher, Marie-Anne Mittet, Tiemo Landes, and Pierre Grussenmeyer. First Experiences With Kinect v2 Sensor for Close Range 3D Modelling. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XL-5/W4:93–100, 2015.

[14] Marc Levoy and Pat Hanrahan. Light Field Rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 31–42, New York, NY, USA, 1996. ACM.

[15] Shenhong Li, Xiongwu Xiao, Bingxuan Guo, and Lin Zhang. A Novel OpenMVS-Based Texture Reconstruction Method Based on the Fully Automatic Plane Segmentation for 3D Mesh Models. *Remote Sensing*, 12(23), 2020.

[16] Steven Maesen, Patrik Goorts, and Philippe Bekaert. Omnidirectional free viewpoint video using panoramic light fields. In *2016 3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*, pages 1–4, 2016.

[17] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, page 39–46, New York, NY, USA, 1995. Association for Computing Machinery.

[18] Media Working Group. WebCodecs W3C Working Draft. `https://www.w3.org/TR/webcodecs/`, 2021.

[19] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *Proceedings of the 16th European Conference on Computer Vision*, ECCV '20, 2020.

[20] Ryan S Overbeck, Daniel Erickson, Daniel Evangelakos, Matt Pharr, and Paul Debevec. A system for acquiring, processing, and rendering panoramic light field stills for virtual reality. *ACM Transactions on Graphics (TOG)*, 37(6):1–15, 2018.

[21] Iraj Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE MultiMedia*, 18(4):62–67, April 2011.

[22] Ruben Verhack, Thomas Sikora, Glenn Van Wallendael, and Peter Lambert. Steered Mixture-of-Experts for Light Field Images and Video: Representation and Coding. *IEEE Transactions on Multimedia*, 22(3):579–593, 2020.

[23] Maarten Wijnants, Hendrik Lievens, Nick Michiels, Jeroen Put, Peter Quax, and Wim Lamotte. Standards-Compliant HTTP Adaptive Streaming of Static Light Fields. In *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*, VRST '18, 2018.