MDPI

Article

Dimensionality Reduction for the Real-Time Light-Field View Synthesis of Kernel-Based Models

Martijn Courteaux, Hannes Mareen, Bert Ramlot, Peter Lambert and Glenn Van Wallendael *

IDLab, Ghent University—imec, 9052 Ghent, Belgium; martijn.courteaux@ugent.be (M.C.); hannes.mareen@ugent.be (H.M.); bert.ramlot@ugent.be (B.R.); peter.lambert@ugent.be (P.L.)
* Correspondence: glenn.vanwallendael@ugent.be

Abstract: Several frameworks have been proposed for delivering interactive, panoramic, camera-captured, six-degrees-of-freedom video content. However, it remains unclear which framework will meet all requirements the best. In this work, we focus on a Steered Mixture of Experts (SMoE) for 4D planar light fields, which is a kernel-based representation. For SMoE to be viable in interactive light-field experiences, real-time view synthesis is crucial yet unsolved. This paper presents two key contributions: a mathematical derivation of a view-specific, intrinsically 2D model from the original 4D light field model and a GPU graphics pipeline that synthesizes these viewpoints in real time. Configuring the proposed GPU implementation for high accuracy, a frequency of 180 to 290 Hz at a resolution of 2048 × 2048 pixels on an NVIDIA RTX 2080Ti is achieved. Compared to NVIDIA's *instant-ngp* Neural Radiance Fields (NeRFs) with the default configuration, our light field rendering technique is 42 to 597 times faster. Additionally, allowing near-imperceptible artifacts in the reconstruction process can further increase speed by 40%. A first-order Taylor approximation causes imperfect views with peak signal-to-noise ratio (PSNR) scores between 45 dB and 63 dB compared to the reference implementation. In conclusion, we present an efficient algorithm for synthesizing 2D views at arbitrary viewpoints from 4D planar light-field SMoE models, enabling real-time, interactive, and high-quality light-field rendering within the SMoE framework.

Keywords: light field model; rendering; transformation; view synthesis

Citation: Courteaux, M.; Mareen, H.; Ramlot, B.; Lambert, P.; Van Wallendael, G. Dimensionality Reduction for the Real-Time Light-Field View Synthesis of Kernel-Based Models. *Electronics* **2024**, *13*, 4062. https://doi.org/10.3390/electronics13204062

Academic Editors: Mário P. Véstias and Rui Policarpo Duarte

Received: 14 September 2024
Revised: 11 October 2024
Accepted: 12 October 2024
Published: 15 October 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Light fields enable users to move around with six degrees of freedom (6DoF) within camera-captured media. For example, a common way users can experience such freedom is through virtual reality (VR) systems with a head-mounted display. While 6DoF functionality is well established in 3D graphics applications (e.g., 3D video games), it remains challenging when applied to camera-captured content.

A successful system for representing, transmitting, and displaying light fields should have all of the following properties simultaneously:

1. Plenoptic abilities: A unique advantage of light field recordings is that they can capture direction-dependent and complex light phenomena, such as refractions, smoke, fire, and reflections of curved surfaces.
2. Camera-captured material: The system should be able to create representations from real-world camera-captured images or videos. This comes with a few challenges, such as the lack of accurate depth information and dealing with calibration information (e.g., regular sampling grids cannot be assumed).
3. Six degrees of freedom: The representation should be fit for allowing a user to move in three translational and three rotational degrees of freedom, synthesizing views accordingly.
4. Large immersive volumes: The representation should allow the user to explore a large volume containing significant occluders (e.g., entering different rooms).

5. Panoramic: As the user has complete freedom to look around, content needs to be available in an immersive way in all viewing directions.
6. Low latency and low decoding complexity: Due to the interactive 6DoF nature, it is not a priori known how the user will consume the content. E.g., the user can behave unpredictably, wander off, walk far away, make abrupt movements, etc. The representation and system have to be able to keep up with these non-predetermined ways of consuming content.
7. Streamable representation: Ideally, one would like to deliver these interactive experiences remotely. This requires the representation at hand to be streamable while respecting the interactive nature of content consumption.
8. Real-time playback: The system and representation should be fast enough to deliver all of the above aspects in real time and on commodity hardware. For VR applications, the system should ideally achieve rendering at 90 frames per second with dual output.

Although there are a wide variety of light-field-related technologies [1], there is no common consensus on which of them (or combinations thereof) will serve as the basis for such a successful system. Several technological approaches, such as 3D-graphics-based, depth-image-based rendering (DIBR), neural radiance fields (NeRFs), and kernel-based regression methods, are all actively researched. Each technology or system makes a trade-off between the described challenging properties. For example, traditional and stereoscopic 360° images/videos lack the 6DoF aspect. Approaches that rely on reconstructing the physical geometry using conventional 3D triangle meshes usually have all light information baked into the textures, taking away the unique advantages of light fields, while often struggling to represent fine details. DIBR systems tend to achieve high quality and high frame rate but typically have small immersive volumes, have trouble with disocclusions, and additionally need accurate depth information, where small inaccuracies in depth often cause unpleasant artifacts. Recently, NeRF approaches have proven to be promising, although not yet suitable for interactive frame rates on commodity hardware. Kernel-based representations such as SMoE have an appealing combination of properties, such as random access, small, and high streamability, but can be slow to render (see Section 2.2 for details).

This paper proposes a novel method that solves the real-time playback problem of higher-dimensional kernel-based models for arbitrary 6DoF viewpoints. We demonstrate this methodology more specifically on planar 4D light-field SMoE models, which use Gaussian kernels. Our method uses a modern GPU graphics pipeline to first transform the 4D model to a view-specific 2D model in the geometry shader, which emits ellipses that are rasterized with a custom fragment shader. We use OpenGL and geometry shaders for simplicity, but our method can easily replace geometry shaders with instanced rendering and be adapted to a wide variety of hardware and graphics APIs, such as Vulkan, DirectX, and mobile GPUs. For a video overviewing the proposed technique, as well as the public source code, see the data availability statement.

This paper is structured as follows. First, Section 2 gives an overview of related works in light-field technologies and a summary of existing work on SMoE. Then, in Section 3, we present our proposed GPU-based rendering technique, which relies on our novel scheme to obtain a 2D view-specific representation of the original 4D light-field model. Next, in Section 4, we validate our method thoroughly and compare it against the state-of-the-art NeRF representation from 'instant-ngp' [2]. Our conclusions are formulated in Section 5. Finally, formulas and derivations of our scheme are detailed in Appendix A.

2. Related Works

Due to the plenoptic requirements, as discussed in Section 1, we consider the following signature of the plenoptic function:

$$\mathcal{L} : \mathbb{R}^5 \to \mathbb{R}^3 : (x, y, z, \theta, \phi) \mapsto (R, G, B). \tag{1}$$

This variant of the plenoptic function describes the light at every point (x, y, z) in space and incoming from every direction (θ, ϕ) by a color in some RGB color space. For simplicity,

we have left out the time dimension. Taking pictures with a camera in a scene essentially records samples from this plenoptic function, where x, y, z match the position of the camera sensor, and θ and ϕ vary according to the field of view of that camera and lens. When one takes enough pictures of a scene (i.e., gathers a lot of samples from the plenoptic function), reconstructing novel (i.e., unseen) viewpoints based on the captured data becomes a reasonable feat. Creating these unseen viewpoints is called light-field synthesis.

The remainder of this section is divided into two parts: a general overview of light-field-related technologies and a more detailed summary of existing work regarding Steered Mixtures of Experts (SMoEs) for light-field representations.

2.1. Light-Field-Related Works

This section provides an overview of approaches to represent plenoptic data. Not all representations are part of one coherent system that supports interactive rendering, bt they are discussed anyway to provide deeper insight into the specific properties we desire. We consider a few classes of representations, which are briefly discussed next.

2.1.1. Three-Dimensional Graphics

The 3D graphics technique aims to reconstruct the physical environment of a scene using textured 3D meshes, but this introduces noticeable artifacts due to the difficulty of the reconstruction process [3–5]. Textures fail to capture non-Lambertian lighting effects, leading to unnatural results [6]. Notable examples include Microsoft's mixed reality solutions and 8i Studio [7,8].

2.1.2. Depth + Image-Based Techniques

Depth maps or other geometric information can be used to improve the ray sampling strategy. Unstructured Lumigraph Rendering was one of the first to use a real-time approach, using a blending field that determines the contribution of each input image to the synthesized view [9]. Neural Lumigraph Rendering improved upon this concept by optimizing a neural signed distance field and an emissivity function [10]. Another approach is the one by Overbeck et al., which combines image-based techniques and depth maps with video coding technology [11]. Broxton et al. used dynamic layered meshes textured by alpha-enabled video streams to handle reflections [12]. The MPEG-I standards suite and the MPEG Immersive Video (MIV) standard use atlases of texture and depth information, encoded with H.264 AVC, H.265 HEVC, or H.266 VVC [13]. In summary, these techniques are capable of producing real-time results, but they face challenges with disocclusions and ghosting artifacts.

2.1.3. Fourier Disparity Layers

Fourier Disparity Layers (FDLs) represent the light-field data as a stack of Fourier transformed images [14]. Every layer corresponds to a certain depth of visual information. FDL data are then compressed using HEVC to arrive at a compact file format [15]. As a translation in the pixel domain corresponds to a multiplication by a complex number in the Fourier domain, the authors present view interpolation within the plane of original viewpoints (i.e., two degrees of freedom). Additionally, thanks to the fact that every layer is already in the Fourier domain and separated by depth, simulating different apertures is an elegant process. Nonetheless, FDL is limited to two degrees of freedom in view interpolation, which restricts its flexibility for more complex 3D scene navigation.

2.1.4. Neural Radiance Fields (NeRFs)

NeRFs have garnered significant attention due to their impressive modeling and view-synthesis capabilities. The original NeRF implementation utilizes a dense neural network (5D input, position and direction; and a 4D output, density and RGB color) to perform volumetric rendering along marched rays [16]. This neural network is optimized using a differentiable ray-marching procedure by training on the rays corresponding to the source images' pixel data. Since the release of Mildenhall et al.'s seminal paper, numerous improvements and/or

variations have been proposed to the original NeRF implementation [17,18]. Most notably, Mip-NeRF [19] and Mip-NeRF 360 [20] were able to substantially improve upon the original NeRF's quality. Additionally, instant-ngp improved the original NeRF's performance by two orders of magnitude, achieving real-time inference speeds (above 1 Hz for 1080p resolution) [2]. More recently, Tri-MipRF was shown to simultaneously achieve Mip-NeRF 360's quality while retaining instant-ngp's performance [21]. While these frameworks are promising, their inference times are currently still far too high for real-time 6DoF camera-captured VR experiences (on commodity hardware).

2.1.5. Kernel-Based Methods

The work on Steered Mixture-of-Experts (SMoE) by Verhack et al. establishes the fundamentals for the kernel-based modeling of light fields and other multi-dimensional image modalities [22,23]. The core idea is that the plenoptic function underlying to a set of images captured from a scene is sufficiently simple to be able to approximate it in a continuous and geometrically sensible way. SMoE carries this out by composing the 4D light-field function using simple 4D building blocks called kernels. In essence, these are mixture models approximating the underlying plenoptic function. Once an SMoE model is built, the continuous plenoptic function itself is available, and synthesizing new views is conceptually as trivial as reconstructing captured views. The challenge is to make sure that the continuous plenoptic extension of the discretely captured pixel data makes sense such that the synthesis of unseen viewpoints produces reasonable results. Section 2.2 dives deeper into SMoE for light fields. Bochinski et al. and Liu et al. worked on similar concepts but applied them to images and video instead of to light fields [24,25]. Although Verhack et al. described synthesizing *in-plane* views for planar light fields, this synthesis process takes 10 s to 10 min per frame, which makes it inadequate for real-time free-viewpoint playback [23,26]. This synthesis algorithm was in essence a block-accelerated version of the per-pixel SMoE reconstruction process because it only considered nearby kernels for each block. More recently, kernel-based methods have gained more interest since the publication of 3D Gaussian Splatting [27] and 2D Gaussian Splatting [28]. The main difference between SMoE and Gaussian Splatting lies in the representation of direction-dependent light information. For SMoE, kernels live in four dimensions and change their position and appearance based on the requested virtual camera direction. In Gaussian Splatting, each kernel has a fixed position in 3D or 2D space combined with a set of spherical harmonics to code/represent the angle-dependent color emitted from the kernel position.

In summary, kernel-based methods are a promising approach for modeling light fields and synthesizing new views. This geometric and continuous extension allows for a smooth and intuitive reconstruction of unseen viewpoints. Currently, 3D Gaussian splatting kernels present a real-time solution, and SMoE lags behind 3D Gaussian splatting due to its slow view synthesis process, which can take between 10 s to 10 min per frame. This limitation is a key challenge that the current paper seeks to address. The next subsection will delve deeper into SMoE's disadvantages.

2.2. Summary of SMoE for Light Fields

2.2.1. Preliminaries

SMoE is a mixture of experts inspired by Gaussian Mixture Models (GMMs) used to model camera-captured visual data. In the case of planar light fields, these data have four intrinsic dimensions: c_x (camera-x), c_y (camera-y), p_x (pixel-x), p_y (pixel-y). More specifically, when considering a two-dimensional array of cameras on a plane, every camera has a $(c_\mathrm{x}, c_\mathrm{y})$ coordinate in that plane. Each camera takes a picture such that every pixel has a $(p_\mathrm{x}, p_\mathrm{y})$ coordinate within the picture. Thus, every recorded sample (i.e., pixel) has a four-dimensional identifying coordinate $(c_\mathrm{x}, c_\mathrm{y}, p_\mathrm{x}, p_\mathrm{y})$ with a corresponding color value $(c_\mathrm{r}, c_\mathrm{g}, c_\mathrm{b})$. Note that the order of the dimensions implies our coordinate space convention: first camera dimensions, then pixel dimensions.

An SMoE model is like a 4D canvas on which every component (i.e., expert) locally paints in a 4D region of the canvas. The paint color is determined by a component-specific

color approximation function (i.e., regressor). These components paint on top of the canvas in a predetermined order with a certain opacity (also known as alpha), by means of alpha compositing (also known as alpha blending). This way, components work together to paint the objects and, more generally, the entire scene. An object being occluded by another object from a range of viewpoints is modeled by components painting over the region of the canvas that was already painted in by the other components representing the occluded object. This is very similar to how a video game can draw the (translucent) objects in the game world back to front, thereby occluding distant objects with nearby objects.

For a given point in the four-dimensional coordinate space, the alpha (i.e., opacity) and color approximation functions of all components are calculated. The colors are then alpha-composited on top of each other in the predetermined order of the model. When combining enough components, covering the entire region of captured samples, a full, high-quality model can be constructed.

In this work, the alpha functions are clipped density functions of multivariate normal distributions, while the color regressors are linear. The i-th component is characterized by its center $\vec{\mu}_i \in \mathbb{R}^4$, its covariance matrix $R_i \in \mathbb{R}^{4\times4}$, its sharpness parameter $s_i \in \mathbb{R}$, its center color $\vec{\xi}_i \in \mathbb{R}^3$ with the corresponding color gradient matrix $W_i \in \mathbb{R}^{3\times4}$, and an additional alpha-scaling factor $\alpha_i \in \mathbb{R}$. The alpha-function $\gamma_i(\vec{x}) \in \mathbb{R}$ for component i at position $\vec{x} \in \mathbb{R}^4$ is given by

$$\gamma_i(\vec{x}) := \alpha_i \min(1, e^{s_i} \sqrt{(2\pi)^4 |R_i|}\ \mathcal{N}(\vec{x}\ ;\ \vec{\mu}_i, R_i)) \tag{2}$$

$$= \alpha_i \exp\Big(-\frac{1}{2}\max\Big(0, (\vec{x}-\vec{\mu}_i)^\intercal R_i^{-1}(\vec{x}-\vec{\mu}_i) - 2s_i\Big)\Big), \tag{3}$$

where the exponent 4 corresponds to the dimensionality of the model and $\mathcal{N}$ represents the density function of a multivariate normal distribution. The multivariate normal distribution density function $\mathcal{N}$ is purposely rescaled with the factor $\sqrt{(2\pi)^4|R_i|}$. This factor is the inverse of the typical normalization factor found in 4-variate normal distributions, such that the choice of R_i does not influence the peak amplitude of the alpha.

Given these alpha functions, all components are combined into the final color $\vec{f}(\vec{x}) \in \mathbb{R}^3$ by alpha compositing (on a black background)

$$\vec{f}(\vec{x}) := \sum_{k=0}^{K-1} \vec{f}_k(\vec{x})\gamma_k(\vec{x}) \prod_{i=k+1}^{K-1} (1-\gamma_i(\vec{x})), \tag{4}$$

where $\vec{f}_i \colon \mathbb{R}^4 \to \mathbb{R}^3$ denotes the linear approximation function of the i-th component:

$$\vec{f}_i(\vec{x}) := \vec{\xi}_i + W_i(\vec{x}-\vec{\mu}_i). \tag{5}$$

2.2.2. Related Works in SMoE for Light Fields

Equation (4) allows us to evaluate an SMoE model at arbitrary positions if the model parameters are known. The question remains how one uses this formula in practice. The most straightforward approach is to implement the alpha-composited interpretation of this formula as CPU or GPU code and evaluate it for every pixel of the desired view. However, in order to evaluate this formula for one pixel, all components' alphas and color approximations need to be computed using Equations (2) and (5). This approach would result in very poor performance, getting worse with increasing output resolution and an increasing number of model components. The complexity can be denoted as $O(KN)$, where K is the number of components and N is the number of output pixels.

3. Proposed Rendering Method

This section presents a novel method for efficiently synthesizing arbitrary viewpoints (including out-of-plane), by preprocessing the model parameters on a per-component basis (see Section 3.1), which are then efficiently rendered on a modern GPU pipeline (see Section 3.2).

Loosely speaking, the complexity of this rendering method resembles $O(N + K)$, with K as sthe number of components and N as the number of output pixels. This is in contrast to the naive approach described in Section 2.2.2, for which the complexity is denoted as $O(KN)$. As such, our approach yields very fast render times, as evaluated in Section 4.

3.1. Constructing a View-Specific 2D Representation

Suppose that given a viewpoint $\vec{v}$, camera rotation, and projection matrix, we wish to synthesize a view from a 4D SMoE light-field model. This implies that the dimensionality of $\vec{x}$ in Equations (3) and (5) is also four. However, the rendered result for a particular view is naturally 2D (such as the 2D pixel grid of a digital monitor). To obtain a 4D model coordinate, we can cast a ray from the virtual viewpoint to the camera plane. As defined in Section 2.2.1, the four coordinates are camera-x, camera-y, pixel-x, and pixel-y. The first two are the camera-x and camera-y coordinates of the intersection point. The last two are the pixel coordinates obtained after projecting the ray direction using the projection matrix of the original cameras. This procedure is explained in detail in Appendix A.3. We can query the original 4D model by applying the described mapping procedure to every pixel of the virtual viewpoint. The mapping function is denoted as $\vec{q}_{\mathrm{s}}(\vec{s}) \in \mathbb{R}^4, \forall \vec{s} \in \mathbb{R}^2$. Naturally, using this function $\vec{q}_{\mathrm{s}}(\vec{s})$ in Equations (3) and (5) gives 2D screen-space equivalents:

$$\gamma_{\mathrm{s},i}(\vec{s}) = \gamma_i(\vec{q}_{\mathrm{s}}(\vec{s})), \tag{6}$$

$$\vec{f}_{\mathrm{s},i}(\vec{s}) = \vec{f}_i(\vec{q}_{\mathrm{s}}(\vec{s})). \tag{7}$$

These are, respectively, the alpha function and the color regression function of the i-th component in screen-space coordinates $\vec{s}$.

To devise a quick and specialized rendering algorithm on a GPU-based modern graphics pipeline, we need to extract the screen-space center and covariance matrix for every component. However, as Appendix A.2 elaborates on in detail, this mapping function is non-linear, which prevents us from extracting this information easily. To overcome this, we begin by replacing the non-linear mapping function $\vec{q}_{\mathrm{s}}$ by its first-order Taylor approximation. For every model component, we calculate the optimal point around which this Taylor approximation $\vec{q}'_{\mathrm{s}}$ is developed. As such, we introduce the approximated screen-space equivalent functions based on $\vec{q}'_{\mathrm{s}}$ instead:

$$\gamma'_{\mathrm{s},i}(\vec{s}) = \gamma_i(\vec{q}'_{\mathrm{s}}(\vec{s})), \tag{8}$$

$$\vec{f}'_{\mathrm{s},i}(\vec{s}) = \vec{f}_i(\vec{q}'_{\mathrm{s}}(\vec{s})). \tag{9}$$

Simplifying these formulas allows us to extract the per-component screen-space information.

To conclude, in Appendix A, the reader can follow the (non-trivial) derivations to obtain the parameters and formulas of a two-dimensional component. All components' two-dimensional approximations put together, in the same way as in Equation (4), form the visual output image of a virtual camera. This means that the camera position, orientation, and projection matrix can be configured freely.

3.2. GPU Implementation

In Section 3.1, we explained how to obtain a two-dimensional model that approximates the view as seen from a virtual camera. In this section, we present a technique to efficiently render such a 2D model, using the final forms of Equation (8) obtained in Appendix A, at Equations (A42) and (A43).

Our technique relies on the fact that every component's impact becomes negligible at a certain distance from the component center $\vec{\mu}_i$ or equivalently from a certain distance from $\vec{s}_{\mathrm{opt}}$ on the screen. When considering the two-dimensional alpha function in Equation (A42), one can indeed see that as the Mahalanobis distance $M(\vec{s})$ becomes larger, the alpha quickly becomes very small. Equation (A42) can be used to solve the Mahalanobis distance $\bar{M}$ at which the function yields a desired threshold alpha $\bar{\alpha}$. This Mahalanobis distance $\bar{M}$ can

be used as a cut-off distance, beyond which we consider the impact of a component to be negligible. In the case of a monitor with eight-bit color depth, an $\bar{\alpha}$ between 0.125/256 and 2/256 is a reasonable range. As a good trade-off between speed and accuracy, we suggest $\bar{\alpha} = 1/256$. Larger thresholds will cause the cut-off distance to be smaller but might start causing noticeable edges (this effect is evaluated later). Solving Equation (A42) for the screen-space cut-off distance $\bar{M}$ yields

$$\begin{aligned} \bar{\alpha} &= \alpha_i \exp\left(-\frac{1}{2}\max\left(0, \bar{M}^2 + (\Delta\vec{q})^\intercal R_i^{-1}\Delta\vec{q} - 2s_i\right)\right) \\ \iff \bar{M} &= \sqrt{2\log\left(\frac{\alpha_i}{\bar{\alpha}}\right) - (\Delta\vec{q})^\intercal R_i^{-1}\Delta\vec{q} + 2s_i}\,. \end{aligned} \tag{10}$$

The rendering of an SMoE model is a matter of a straightforward alpha compositing of every component, one by one, after transforming them from 4D to 2D using the formulas from Section 3.1. As our implementation was written in OpenGL, the remainder of this section uses OpenGL terminology, but can be it can be carried out in any graphics API supporting geometry shaders. Specifically, we propose the following implementation strategy.

First, store the parameters of the 4D components in a vertex buffer, such that one vertex contains the attributes of one component, and there are as many vertices as components. A single component should fit in nine 4-vectors.

The shader program consists of a vertex, geometry, and fragment shader. The vertex shader passes all vertex attributes as-is to the geometry shader.

The geometry shader is responsible for executing the required calculations described in the previous section to obtain the two-dimensional parameters for the current component. The required information to carry this out, such as the viewpoint $\vec{v}$, the matrices M and P of the virtual camera, and the P_o matrix, are all passed in as uniforms. Given this parameterization, the geometry shader should then emit a triangle strip with vertices at positions $\vec{v}_k$ in the shape of an ellipse around $\vec{s}_{\text{opt}}$. The shape of the ellipse corresponds to the 2×2 screen-space covariance matrix $\hat{R}_i$ and is bound by the computed Mahalanobis-cutoff distance $\bar{M}$. We do this by multiplying 10 coordinates along the unit circle with the Cholesky decomposition of $\hat{R}_i$, followed by a corrective factor.

$$\vec{v}_{k,\text{circle}} := \bar{M} \cdot \begin{bmatrix} \cos((-1)^k 2\pi k/10) \\ \sin((-1)^k 2\pi k/10) \end{bmatrix}, \quad \forall k \in (0..9), \tag{11}$$

$$\vec{v}_{k,\text{ell}} := \text{chol}(\hat{R}_i) \cdot \vec{v}_{k,\text{circle}}, \tag{12}$$

$$\vec{v}_k := \vec{s}_{\text{opt}} + \vec{v}_{k,\text{ell}} \frac{\bar{M}}{\left\| L_i^{-1}\left(\vec{q}_{\text{s}}(\vec{s}_{\text{opt}} + \vec{v}_{k,\text{ell}}) - \vec{q}_{\text{opt}}\right)\right\|}, \tag{13}$$

where,

$$L_i := \text{chol}(R_i) \quad \in \mathbb{R}^{2\times 2}, \tag{14}$$

$$\vec{q}_{\text{s}}(\vec{s}) := D \cdot \vec{d}_{\text{n}}(\vec{s}) + \vec{e} \quad \in \mathbb{R}^4, \tag{15}$$

$$\vec{d}_{\text{n}}(\vec{s}) := -\left(\vec{d}(\vec{s})\right)_{\text{xy}} \Big/ \left(\vec{d}(\vec{s})\right)_{\text{z}} \in \mathbb{R}^2, \tag{16}$$

$$\vec{d}(\vec{s}) := MP^{-1}\langle \vec{s} \,|\, 1\rangle \quad \in \mathbb{R}^3. \tag{17}$$

The Cholesky decomposition of $\hat{R}_i$ is easy to compute in the shader itself:

$$\text{chol}(\hat{R}_i) := \begin{bmatrix} a & 0 \\ b & c \end{bmatrix}, \tag{18}$$

$$a := \sqrt{\hat{R}_{i,11}}, \quad b := \hat{R}_{i,12}/a, \quad c := \sqrt{\hat{R}_{i,22} - b^2}. \tag{19}$$

The emitted vertices are additionally attributed with their corresponding $\vec{v}_{k,\text{circle}}$ position, their color $\vec{f}_{s,i}(\vec{s})$ (as determined by Equation (A43)), the alpha α_i and sharpness s_i, and the constant term $(\Delta\vec{q})^\intercal R_i^{-1}(\Delta\vec{q})$ in the squared Mahalanobis distance from Equation (A42).

The fragment shader then receives all interpolated vertex attributes. The squared Euclidean norm of the interpolated $\vec{v}_{k,\text{circle}}$ serves as M^2 in Equation (A42), which is then evaluated to obtain $\gamma'_{s,i}(\vec{s})$ for the current pixel. The output of the fragment shader is the four-vector $\langle \vec{f}'_{s,i}(\vec{s}) \,|\, \gamma'_{s,i}(\vec{s}) \rangle$ containing the (r, g, b, alpha) values.

We recommend a 16-bit (also known as, half-precision) floating-point frame buffer (`GL_RGB16F`) as a trade-off between speed and precision. As a side note, note that our models happen to use color values outside of the range $(0, 1)$, and as such, the floating point formats are desired for correct reconstructions. Otherwise, an eight-bit color depth represented as integers would suffice (and in fact comes with a noticeable increase in speed). On the other hand, we have observed that 32-bit (also known as single-precision) floating point frame buffers (`GL_RGB32F`) cause a significant slowdown (up to 12 times slower) on several tested machines.

Finally, using the standard alpha-compositing blend mode (`glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`), the full pipeline computes the equivalent of Equation (4).

In conclusion, an overview has been presented of how a GPU-based rendering algorithm should be implemented, using the transformations described in Section 3.1. The necessary adjustments to improve the numerical stability of these computations are also discussed.

4. Evaluation

This section evaluates the speed and quality of the proposed method. More specifically, as the proposed dimensionality reduction method introduces some approximations, we quantified the resulting quality degradations in terms of peak signal-to-noise ratio (PSNR) and Structural Similarity Index Measure (SSIM) by comparing the quality results from the GPU implementation to those of the reference implementation on CPU, as explained in Section 4.1.3. The SSIM is calculated and averaged over R, G, and B color channels of the sRGB image data, using `skimage.metrics.structural_similarity (A, B, data_range=1.0, gaussian_weights=True, use_sample_covariance=False, sigma=1.5, channel_axis=2, multichannel=True)`.

Additionally, as the primary goal of this proposed method of rendering is high performance, we measured the rendering speeds of our GPU implementation. As NeRFs are conceptually comparable to SMoE (i.e., volumetric rendering and the model-based representation of light-field data), we compare our proposed technique with the NeRF implementation of instant-ngp [2], which also focuses on delivering speed, and is the most commonly used implementation of NeRF (in terms of GitHub stars and forks). For extra reference, we report the speed of our block-optimized CPU reference renderer as explained in Section 4.1.3.

For our experiments, we perform the same view-rendering tasks using all three approaches. The dataset, reconstruction tasks, experiment setup, and results are discussed in the following sections.

4.1. Experiment Setup

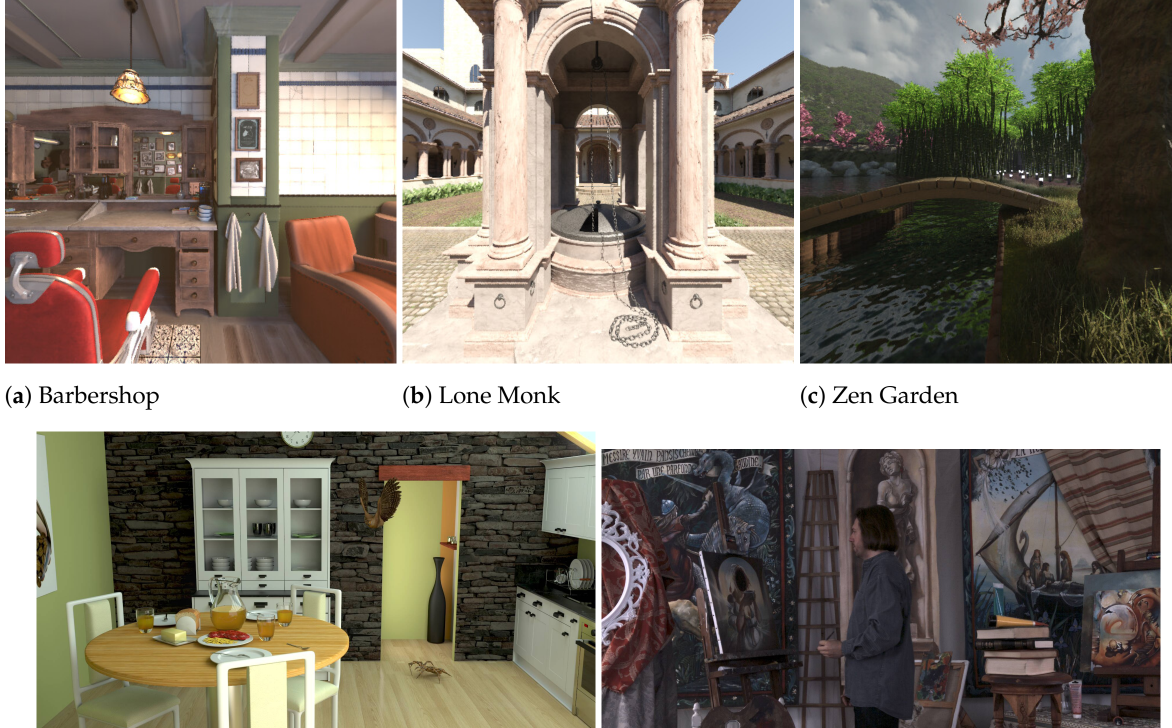
4.1.1. Dataset

For our research and evaluation, we are seeking a dataset that has wide-area coverage for an immersive experience that allows users to walk around a few meters. We also prefer a dataset that is realistic, making it easier to assess the subjective quality. Additionally, we place importance on having good specular lighting and volumetric effects in the dataset as light fields can appear unimpressive without them. Datasets offering all of the above properties simultaneously were rare, so, in previous work, we created SILVR and made it publicly available [29].

SILVR is short for "Synthetic Immersive Large-Volume Ray dataset", and it consists of 180° fisheye-projection images. We reprojected these images to square rectilinear images of

size 512 px with a horizontal and vertical field of view of 102.68° (i.e., the equivalent of a 14 mm lens on a 35 mm by 35 mm sensor).

In addition to the SILVR scenes, we selected one frame from the real-world-captured "painter" [30] and synthetic "kitchen" [31] light-field video sequences, commonly used in the MPEG community. Camera-captured content contains inaccuracies caused by imperfect camera localization, color differences, and so on. These inaccuracies lead to reduced modeling performance, resulting in more components or lower visual quality. The impact on the proposed rendering technique is limited to an increase in the number of components used in the model. Example views can be seen in Figure 1.



(**a**) Barbershop (**b**) Lone Monk (**c**) Zen Garden

(**d**) Kitchen (**e**) Painter

Figure 1. Example viewpoints from the input views for each scene.

In summary, we have the following scenes:

- barbershop (SILVR, cuboid/right face): The "barbershop" scene from the Blender Institute short movie Agent 327: Operation Barbershop [32] (licensed CC-BY). Camera grid: 30×10, spaced 11 cm apart.
- lone monk (SILVR, cuboid/front face): This scene was created by Carlo Bergonzini from Monorender (licensed CC-BY). Camera grid: 21×21, spaced 20 cm apart.
- zen garden (SILVR, cuboid/back face): This scene was created by Julie Artois and Martijn Courteaux from IDLab MEDIA (licensed CC-BY 4.0). Camera grid: 21×21, spaced 10 cm apart.
- kitchen (MPEG, frame 20): This dataset by Orange Labs was software-generated and features a kitchen with a table ready for breakfast and an owl and a spider [31]. This dataset has a rather small area. Camera grid: 5×5. Resolution 1920 px $\times$ 1080 px.
- painter (MPEG, frame 246): This dataset was captured with real cameras and features a man in front of a canvas [30]. This dataset has a rather small area and few specularities. Camera grid: 4×4. Resolution 2048 px $\times$ 1088 px.

We used our still-in-development modeling software to create SMoE models from these datasets with 25,000 components. However, the number of components retained in

the final models varied due to pruning strategies (barbershop: 16,938, lone monk: 15,910, zen garden: 13,668, kitchen: 21,634, painter: 21,647).

4.1.2. Reconstruction Tasks

We define several tasks to evaluate quality and performance. A task consists of a series of viewpoints (i.e., the combination of a position, camera orientation, and field of view). Within the MPEG community, these are called "pose traces". The indicated tasks are the following (to visually examine the tasks, see the data availability statement):

- 'spin' A trajectory where the camera moves from left to right while panning from right to left.
- 'push–pull' A trajectory where the camera moves several meters backward, while the field of view narrows, similar to the cinematic push–pull or dolly zoom effect.
- 'zoom-to-*xyz*' A trajectory where the camera moves both forward and narrows the field of view slightly towards a certain object of interest in the scene.

All experiments were executed on a Linux machine (Super Micro Computer Inc., San Jose, CA, USA) with an Intel i7-5960X (Intel Corporation, Santa Clara, CA, USA) and an NVIDIA RTX 2080Ti (Nvidia Corporation, Santa Clara, CA, USA) .

4.1.3. SMoE CPU Implementation Details

The CPU versions of the SMoE reconstructor are multi-threaded C++ programs using Eigen for the linear algebra operations. This program is compiled with GCC 9.4.0 with `-O3 -mavx2`. There are two variations, CPU-Full and CPU-Block, both implemented in the same codebase but with different execution strategies. The CPU-Full version evaluates all components of an SMoE model for all pixels. Because of that, the number of computations performed by this implementation is independent of the view: both the number of components of the model and the output resolution (i.e., the number of pixels) are the same for every rendered view.

The CPU-block version is a block-optimized interpretation of Equation (4). The view-synthesis task is split into square blocks, where for every block, only relevant components are considered. The block size is manually optimized per scene to find a trade-off between the number of components per the block (more unnecessary computations), and the number of blocks (more time spent selecting components for the blocks). If a component would cause a color contribution of more than a quarter of a quantization level (i.e., $\frac{1}{4}/256$), it is used for reconstructing the block. Blocks are constructed in parallel, on the eight-core test machine, using a queue. Rendering 10% of all views with CPU-Full, the CPU-Block version was validated to be correct, with PSNR scores ranging from 60 to 80 dB and maximal pixel error values of 1 (i.e., rounding error). As such, to reduce experiment time, we used CPU-block results for most of our comparative evaluations.

4.1.4. SMoE GPU Implementation Details

Every view in a task is rendered 20 times, to allow the GPU driver to properly pipeline consecutive frames, allow clock speeds to adjust, and reduce variability, in order to get reliable results. We measure the duration of the entire loop of 20 repetitions followed by a final call to `glFinish()`. This is repeated eight times, such that eight time measurements are performed, of which the fastest result is selected. The reported time is the average time per frame.

4.1.5. NeRF Experiment Details

For every scene, two NeRF models are built with "instant-ngp" [2]: *base* and *small*. The 'base' model is made with the authors' default configuration. The "small" model starts from the base configuration but takes the authors' suggestions on making the model faster for rendering (The "small" configuration uses 2^{11} instead of 2^{19} hash table slots. The slowest element of the instant-ngp NeRF rendering process is the hash table lookups. The neural network computations are in comparison fast and resolve hash collisions. Therefore, the "small" configuration speeds up rendering substantially by having faster hash table

lookups but introduces more hash collisions by doing so, which are then taken care of by the neural network.) Creating the NeRF models, we positioned the camera plane parallel to the XY plane and adjusted it to scale and translation to fit the entire contents in the model bounding box. All reconstructions were generated using five samples per pixel (spp), again for pipelining and reducing overhead. The results are reported (in Section 4.2.3) by rescaling to 1 spp for a fair comparison with SMoE.

4.2. Results and Discussion

4.2.1. Speed Versus Quality Trade-off

The quality and speed aspects of our proposed method are under the great influence of the $\bar{\alpha}$ threshold in Equation (10). This $\bar{\alpha}$ threshold directly determines the component radius $\bar{M}$ and can thus control the number of pixels being evaluated. To this end, we quantified the impact of $\bar{\alpha}$ when varying this threshold between 0.125/256 and 15/256 and comparing the results to the exact CPU reference renders. As can be seen in Figure 2, higher threshold values will cause higher errors but will reduce rasterization workload and hence cause a speedup (approximately 40% faster at $\bar{\alpha} = 2/256$). In practice, using values of $\bar{\alpha}$ up to 3/256 produces results with imperceptible artifacts (PSNR is still around 50 dB). As a demonstration of the type of artifacts to be expected with extreme values for $\bar{\alpha}$, Figure 3 shows the result of the tilted view from Figure 2 for $\bar{\alpha} = 15/256$, which ends our test at 39 dB. As such, we have a way to trade off quality for speed, depending on what is desired. Note that the top and bottom edges of the view in Figure 3 correspond with the edges of the dataset. The modeling process relies on a diverse set of cameras that are not available in these edge regions. As a result, the quality in these areas is lower due to factors beyond our rendering approach.

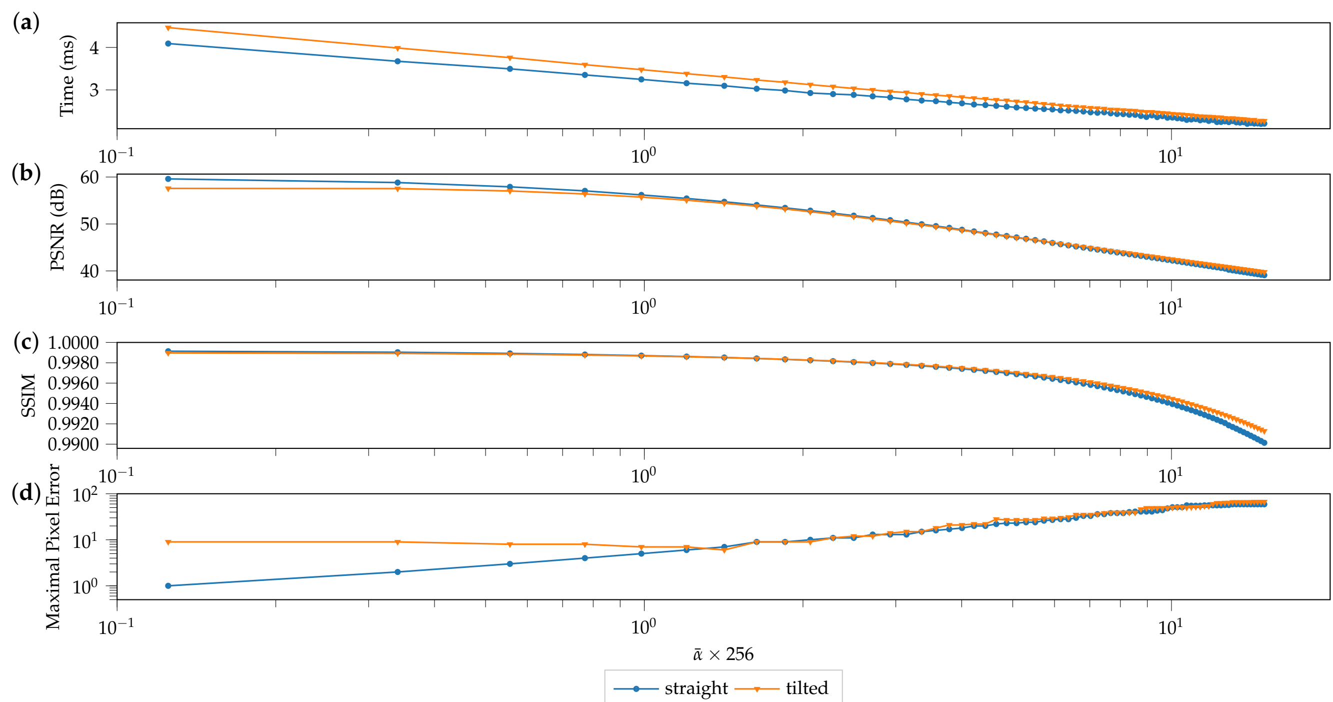


Figure 2. Speed and quality analysis of varying the $\bar{\alpha}$ threshold value on a resolution of 2048 × 2048. This threshold directly influences the size of the area that the component is evaluated on. Two views from the *barbershop* model were used for this test: *straight*, where the optical axis is aligned with the original input views (like in the *push–pull* tasks), and *tilted*, where the virtual camera is rotated instead (like in the *spin* tasks). The horizontal axis is rescaled by factor 256 such that one quantization level of an 8-bit display corresponds to one unit.

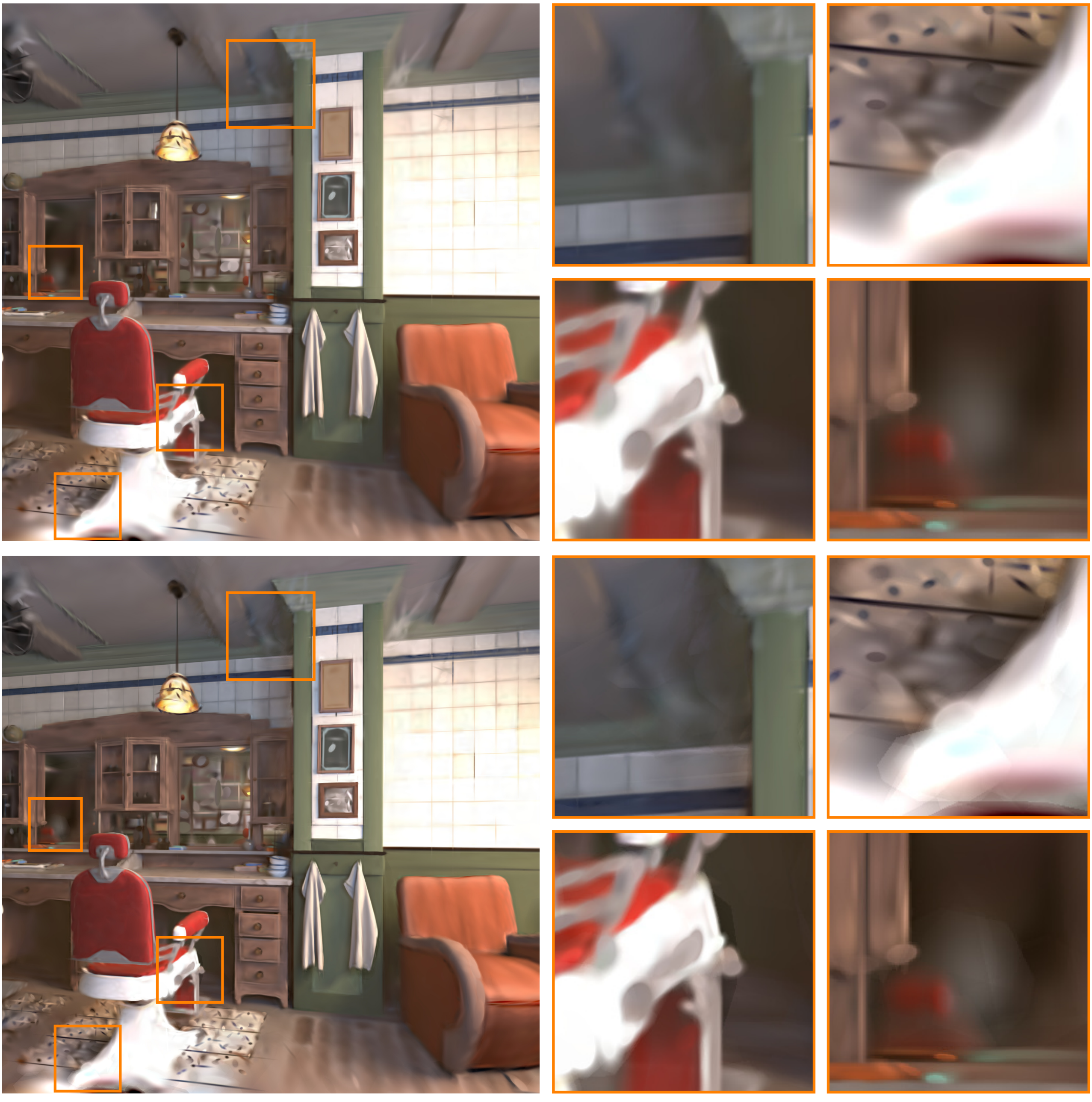


Figure 3. The highest-quality render with $\bar{\alpha} = 0.125/256$ (**top**), compared to the lowest-quality render with $\bar{\alpha} = 15/256$ (**bottom**), of the *tilted* view in the test of Figure 2. These results were produced in 4.1 ms and 2.2 ms, respectively. The quality of the fastest render (i.e., highest $\bar{\alpha}$, **bottom**) is still relatively high (i.e., 39 dB instead of 59 dB). The most noticeable artifacts have been highlighted. Notice how the regions of influence of components being cut off too early cause sharp edges for smooth components (i.e., low s_i). This effect is less noticeable for components that already have a sharp edge.

When targeting a VR headset, 90 Hz is desirable. However, the workload is higher as two separate high-resolution views now need to be synthesized. In order to maintain the required 90 Hz dual-output playback speed on our HTC Vive Pro VR headset, we had to select $\bar{\alpha} = 2/256$ on a NVIDIA RTX 2080Ti. As mentioned earlier, artifacts cannot be seen at this value of $\bar{\alpha}$, and in fact, we found the quality of the image to be satisfying until $\bar{\alpha} = 5/256$ in our VR headset. A video demonstrating this can be found in the data availability statement. To conclude, the proposed method achieves real-time rendering speeds for SMoE.

4.2.2. Correctness

In Figure 4 and Table 1, we show an analysis of the correctness of our proposed method using the highest-quality setting ($\bar{\alpha} = 0.125/256$). We compare the proposed method with the CPU ground truths based on the tasks described earlier. We observe that as long as the virtual camera is aligned with the original cameras of the dataset (as in push–pull), the results are nearly perfect. That is, the PSNR reaches beyond 60 dB, and the maximal pixel color error does not exceed two quantization levels. In fact, the source of the errors is the reduced precision of the 16-bit floating point frame buffer. Repeating the experiment for 32-bit floating point frame buffers yields a ± 2 dB increase along with a maximal pixel color error of only one quantization level.

As soon as the camera tilts out of the original optical axis (as in spin), the first-order Taylor approximation becomes apparent, and we observe a quality decrease to 45 dB in the worst measured case. This observation was consistent across the different scenes. It should be noted that the drop in PSNR due to the Taylor approximation is imperceptible in practical applications. Furthermore, the error introduced by this approximation is minimal compared to the visual artifacts caused by the modeling process, which suffers from a lack of captured data when rendering views from extreme angles, leading to reduced quality.

Table 1. Validation of the proposed GPU implementation for the tested datasets. Note that rendering was performed on a resolution of 2048 × 2048 with a threshold $\bar{\alpha}$ of 0.125/256 to demonstrate exactness. Quality metrics are measured between the renders from the GPU implementation and the CPU-Block implementation as ground truth. Render times are per frame. The indicated numbers represent the mean and standard deviation over the resulting renders for each task.

Scene	Task	PSNR	Time GPU
barbershop	push–pull	59.7 dB	4.5 ms
	spin	57.0 dB	4.6 ms
lone monk	push–pull	58.4 dB	5.3 ms
	spin	54.6 dB	5.2 ms
zen garden	push–pull	62.3 dB	4.1 ms
	spin	59.4 dB	4.5 ms
kitchen	push–pull	59.3 dB	4.3 ms
	spin	59.2 dB	4.2 ms
	zoom-to-sink	58.9 dB	3.5 ms
	zoom-to-table	58.8 dB	4.7 ms
painter	push–pull	61.7 dB	3.6 ms
	spin	61.5 dB	3.5 ms
	zoom-to-painting-1	61.7 dB	3.5 ms
	zoom-to-painting-2	61.0 dB	3.4 ms

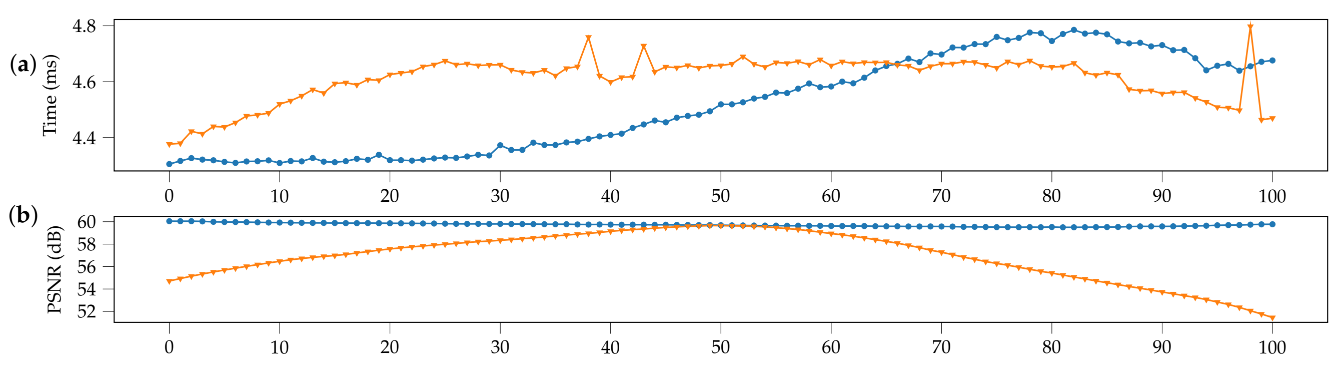


Figure 4. *Cont.*

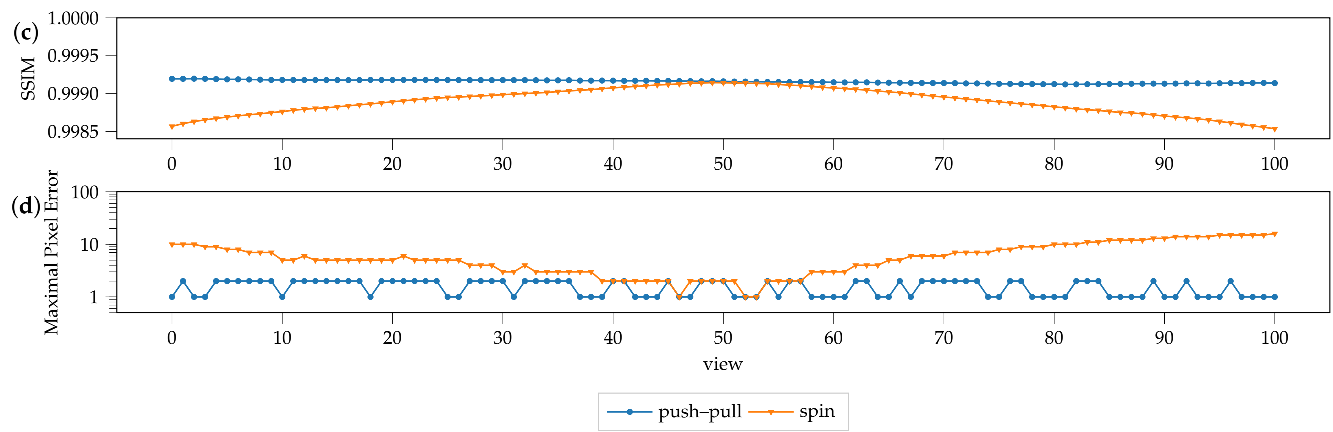


Figure 4. Analysis of the rendering time and quality of the GPU rendering approach, compared to the reference quality from the CPU-block implementation, along with rendering time per frame. Note that rendering was performed with high precision using a threshold $\bar{\alpha}$ of 0.125/256. Figure (**d**) shows the highest absolute error on an individual pixel (MPE) in each frame. The MPE being less than 2 for the *push–pull* task indicates near-perfect reconstruction over the whole image. This analysis is performed with the *barbershop* model on a resolution of 2048×2048.

4.2.3. Comparison with NeRFs

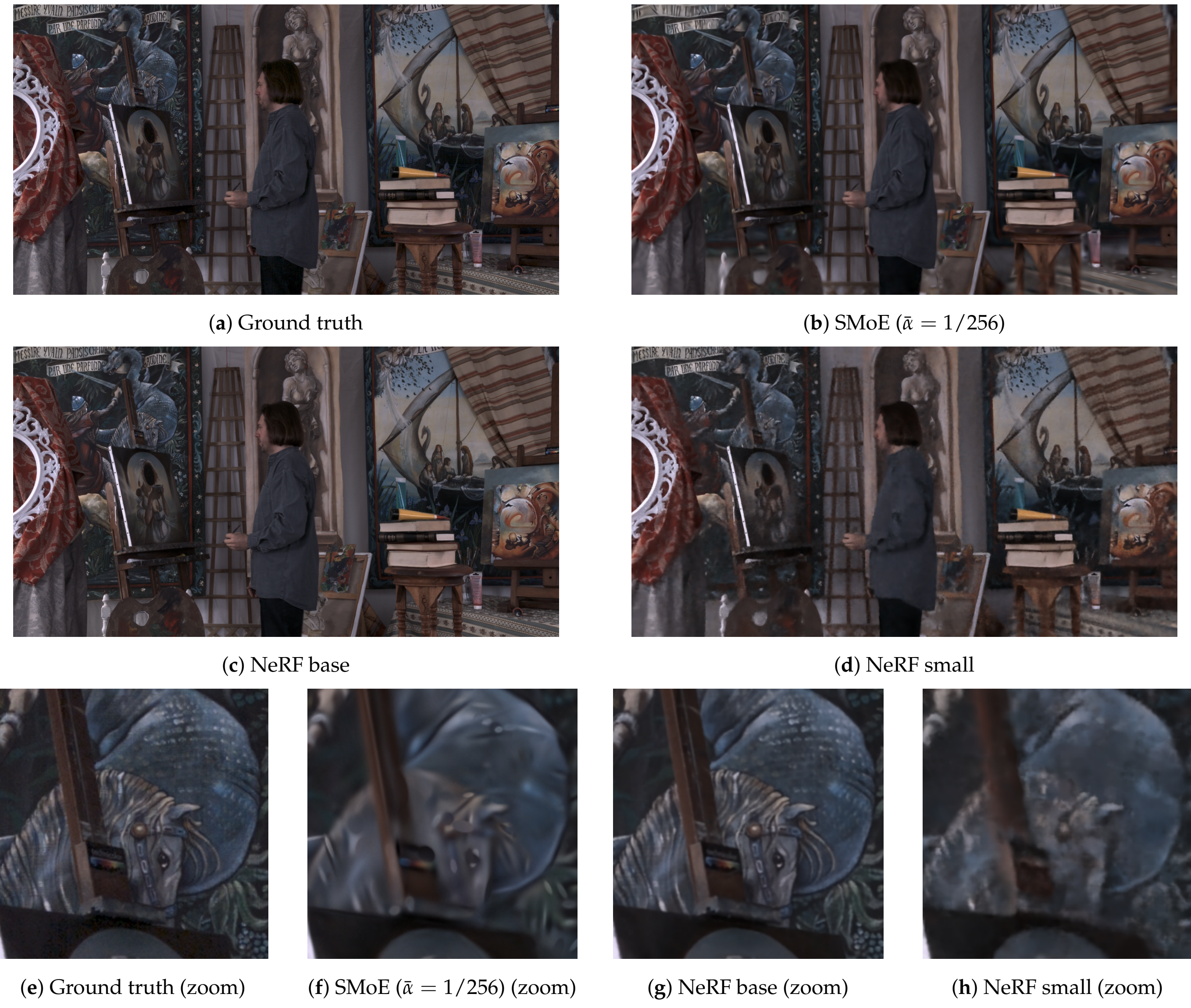
Table 2 compares our proposed SMoE rendering approach (with $\bar{\alpha} = 1/256$) to the performance of NeRFs of "instant-ngp" [2]. These results are obtained by reconstructing the views corresponding to the original images from the dataset and measuring the quality. Observe that instant-ngp's NeRFs with the *small* configuration require on average 229 ms to reconstruct the 1080p resolution views of *kitchen*. As such, these NeRFs are not suitable for real-time interactive frame rates at a 1080p resolution. On the other hand, SMoE reconstructs the same 1080p views in 2.4 ms, reaching 416 Hz (i.e., far above the typical 60 Hz for interactive desktop applications). Additionally, for similar resolutions, instant-ngp has rendering times that vary greatly across scenes (e.g., for the tests at 512×512, it varies from 52 ms for *lone monk* to 112 ms per frame for *zen garden* for the *base* configuration and from 16 ms to 43 ms per frame for the *small* configuration). SMoE has a more predictable performance (i.e., between 1.0 ms and 1.2 ms correspondingly). We hypothesize that the greater variation in NeRF's cross-scene rendering times stems from its use of geometry-dependent optimizations. For example, NeRF uses occupancy grids, whose performance highly depends on the scene's geometry, while our proposed method uses no such geometry-based optimizations. Additionally, our rendering time complexity is practically dominated by the scene-independent number of pixels N and not by the scene-dependent number of components K. More research can be carried out to validate our hypothesis and quantify how scene-dependent elements influence the rendering times.

Table 2. Comparison of the proposed SMoE rendering technique with `instant-ngp` NeRFs in terms of speed and model quality based on the original views. Note that SMoE GPU rendering was performed with a threshold $\bar{\alpha}$ of 1/256. SMoE rendering speed drops when the resolution is small. This happens because resolution-independent work (such as performing the geometry shader computations, and other overhead) takes a more significant portion of the total time. The comparison is therefore more meaningful at higher resolutions (such as with "kitchen" and "painter").

Model	Resolution	SMoE ($\bar{\alpha} = 1/256$)				NeRF Base				NeRF Small			
		PSNR	SSIM	Time	Speed	PSNR	SSIM	Time	Speed	PSNR	SSIM	Time	Speed
barbershop	512×512	26.6 dB	0.85	1.2 ms	222 MP/s	31.1 dB	0.95	65 ms	4.2 MP/s	26.3 dB	0.81	21 ms	12.6 MP/s
lone monk	512×512	24.9 dB	0.78	1.1 ms	234 MP/s	27.9 dB	0.88	52 ms	5.5 MP/s	23.6 dB	0.70	16 ms	16.4 MP/s
zen garden	512×512	29.2 dB	0.78	1.0 ms	259 MP/s	32.0 dB	0.88	112 ms	2.4 MP/s	28.4 dB	0.73	43 ms	6.2 MP/s
kitchen	1920×1080	28.6 dB	0.82	2.4 ms	865 MP/s	36.5 dB	0.95	799 ms	2.6 MP/s	27.0 dB	0.75	229 ms	9.1 MP/s
painter	2048×1088	29.6 dB	0.82	2.4 ms	935 MP/s	36.2 dB	0.93	1425 ms	1.6 MP/s	29.2 dB	0.78	650 ms	3.4 MP/s

Table 3 compares the results from Table 2 in terms of speed and quality. We see that SMoE is up to 273 times faster than the *small* NeRF configuration and achieves comparable quality. When considering the default *base* NeRF configuration, SMoE is up to 597 times faster. While instant-ngp NeRFs do achieve around 3 dB to 8 dB higher PSNR scores, the rendering time for a single frame at 1 sample per pixel is above half a second.

Figure 5 shows a visual comparison between the ground truth, SMoE, NeRF base, and NeRF small for the scene painter. As confirmed by the results from Tables 2 and 3, NeRF base has the highest visual quality, whereas NeRF small has the lowest, and SMoE has a slightly better quality than NeRF small. The lower SMoE model qualities are because our modeling software has difficulties accurately representing content near the edges of the data domain, among a few other things. Improving the SMoE modeling techniques is a subject for future work.



(**a**) Ground truth (**b**) SMoE ($\tilde{\alpha} = 1/256$)

(**c**) NeRF base (**d**) NeRF small

(**e**) Ground truth (zoom) (**f**) SMoE ($\tilde{\alpha} = 1/256$) (zoom) (**g**) NeRF base (zoom) (**h**) NeRF small (zoom)

Figure 5. Visual comparison for the scene painter between SMoE, `instant-ngp` NeRFs, and the ground truth. The NeRF base model exhibits the highest visual quality and NeRF small the lowest, with SMoE falling in between.

To expand in greater detail on the results of Table 2, Figure 6 shows a scatter plot of the quality and reconstruction time for each individual reconstructed view. We can see that all combinations of scenes and approaches yield compact clusters, which is desirable for a predictable and consistent quality of experience.

Table 3. Relative results from Table 2 when comparing SMoE ($\bar{\alpha} = 1/256$) with both NeRF configurations of `instant-ngp`.

Model	SMoE vs. NeRF Base		SMoE vs. NeRF Small	
	ΔPSNR	**Speedup**	**ΔPSNR**	**Speedup**
barbershop	−4.5 dB	×53	+0.3 dB	×18
lone monk	−3.0 dB	×42	+1.3 dB	×14
zen garden	−2.9 dB	×109	+0.8 dB	×42
kitchen	−7.9 dB	×329	+1.6 dB	×96
painter	−6.6 dB	×597	+0.4 dB	×273

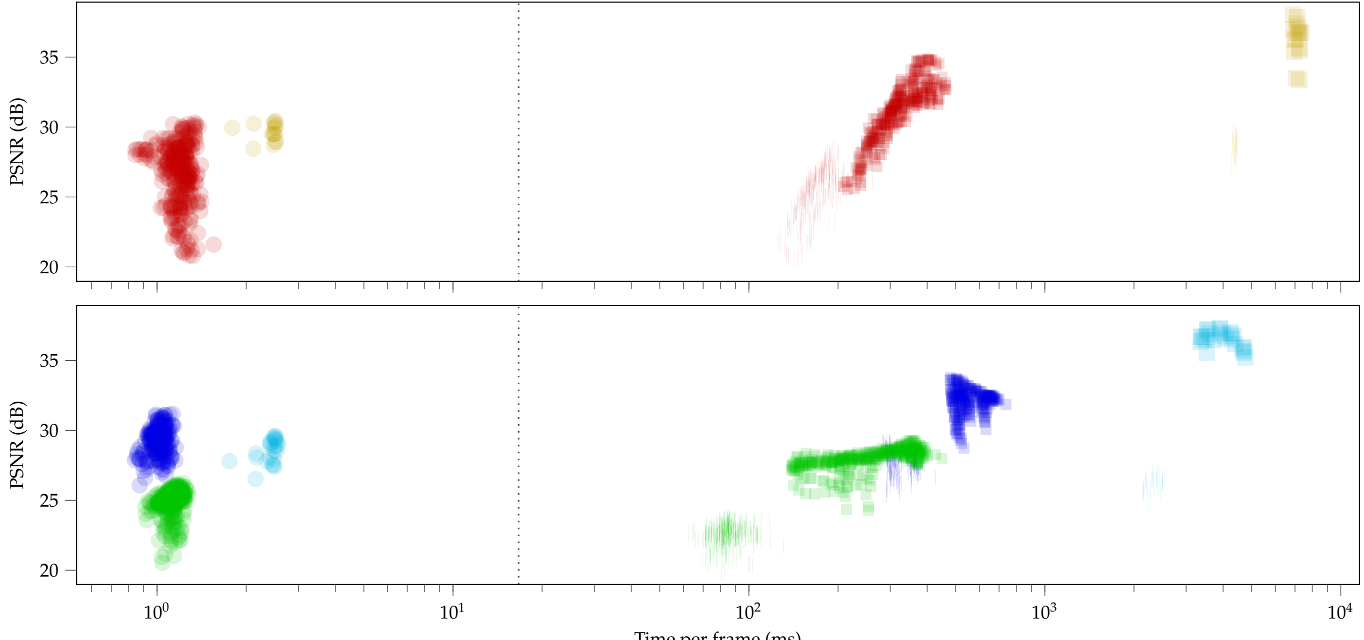


Figure 6. Speed and quality measurements for all original views of all datasets rendered with the three configurations (● SMoE $\bar{\alpha} = 1/256$, ■ NeRF base, | NeRF small). The data are split across two charts to reduce overlapping data points (the axes of both charts are identical). Each color represents a scene (*top chart*: ● barbershop, ● painter – *bottom chart*: ● lone monk, ● zen garden, ● kitchen). The dashed vertical line marks the frame time of 16.6 ms, which is required for 60 Hz playback. Most notably, it can be observed that the SMoE configurations are on the left side of the vertical line (i.e., fast rendering), whereas the NeRF-based configurations are on the right side (i.e., slow rendering).

We do not anticipate that the proposed technique will be memory-constrained as storing the largest scene (Painter) requires less than 5 MiB, which can be further reduced by, for example, leveraging the symmetry of the covariance matrix. For completeness, an indication of the file sizes of each SMoE and NeRF model after ZIP compression is given in Table 4. For both, it is important to stress that this general-purpose ZIP compression is inefficient compared to the compression ratio we expect from custom-made entropy coding. This table is therefore given as a mere order-of-magnitude indication of the size of these model files.

To conclude, when model quality is on par with NeRF, the proposed light-field-rendering technique is 273 times faster, although the maximum quality we are able to reach with the current state of the SMoE modeler is still 3 dB to 8 dB lower. Improving the SMoE models themselves is a subject for future work, but we foresee the impact of having better models (with more components) to minimally impact the rendering performance, and potentially even improve if those models use fewer components per pixel.

Table 4. Comparison of ZIP-compressed file sizes of the model files from each technique. File sizes are expected to be significantly lower using custom-engineered entropy coding.

Scene	SMoE	NeRF Base	NeRF Small
barbershop	3.6 MB	24 MB	2.8 MB
lone monk	3.0 MB	24 MB	1.8 MB
zen garden	3.1 MB	24 MB	2.5 MB
kitchen	3.6 MB	24 MB	1.6 MB
painter	3.6 MB	23 MB	1.1 MB

5. Conclusions

In this paper, we have proposed two important techniques that together enable efficient real-time free-viewpoint 6DoF Steered Mixture of Experts rendering on GPUs. First, the mathematical operations required to transform the 4D planar light-field model into a view-specific 2D model were derived. Next, a description of how to render these 2D models on a modern GPU, using OpenGL employing a geometry shader was presented. The mathematical transformation is lightweight compared to the fragment shading workload. Configuring the proposed GPU implementation for high accuracy, 180 to 290 Hz at a resolution of 2048×2048 pixels on an NVIDIA RTX 2080Ti is achieved. The mathematical formulas derived make use of an approximation, which is partially compensated. This approximation degrades rendering accuracy, which is quantified with PSNR between 47 dB and 63 dB or SSIM values between 0.9985 to 0.9995.

Comparing our light-field-rendering solution to *instant-ngp* neural radiance fields (NeRFs), we achieved 42 to 597 times faster rendering while scoring around 6 dB lower PSNR when reconstructing the original views. Reducing the complexity of the NeRF model and bringing the quality on par with SMoE, we still achieve 14 to 273 times faster rendering. Additionally, when allowing for (near-)imperceptible artifacts in reconstruction quality (not changing the model), our proposed SMoE rendering technique gains an additional 40% speedup. Based on the presented work, it can be stated that SMoE becomes a viable representation and framework for representing interactive light fields.

Future work can study and improve upon the approximations, as well as investigate new techniques for further improving the rendering performance. More broadly, future work should include improving the quality of the models themselves.

Author Contributions: Conceptualization, M.C., P.L. and G.V.W.; methodology, M.C.; software, M.C.; validation, M.C.; formal analysis, M.C., H.M., B.R., P.L. and G.V.W.; investigation, M.C., H.M., B.R., P.L. and G.V.W.; resources, M.C., P.L. and G.V.W.; data curation, M.C.; writing—original draft preparation, M.C.; writing—review and editing, H.M., B.R., P.L. and G.V.W.; visualization, M.C.; supervision, P.L. and G.V.W.; project administration, P.L. and G.V.W.; funding acquisition, M.C., P.L. and G.V.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded in part by the Research Foundation—Flanders (FWO) under Grant 1SA7919N, also by the Flemish Government's Department of Culture, Youth, and Media within the project called Digital Transformation Media, grant number 94186, IDLab (Ghent University—imec), Flanders Innovation & Entrepreneurship (VLAIO), and the European Union.

Institutional Review Board Statement: Not applicable.

Data Availability Statement: Upon acceptance of this paper, we will publish our implementation open-source on GitHub together with the models to allow people to try the SMoE-based light field view synthesis for themselves (in VR if desired). We provide a few videos online: (1) A demonstration of a real-time interactive session; (2) A VR demonstration, showing our setup working at 90 Hz on a HTC Vive Pro; (3) Videos of the reconstructed frames of the pose traces used for comparison are informatively supplied as H.264-compressed videos online. Note that these video files are created using *lossy* compression, meaning the frames of the videos are not perfect copies of the original image files, and that they cannot be used to recompute SSIM and PSNR values. Instead, they are provided for visual inspection. If you would like to obtain access to the lossless PNG sequences, feel free to contact

the authors. You can find these videos here: https://media.idlab.ugent.be/immersive-rendering (accessed on 10 October 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Proposed Reduction Method

This appendix section describes the proposed method for reducing a 4D model to a 2D model corresponding to a specific virtual view:

- Appendix A.1 describes the used coordinate spaces and notational conventions.
- Appendix A.2 gives a high-level overview and the motivation behind the reduction method.
- Appendix A.3 formalizes the concepts introduced in the overview.
- Appendix A.4 derives a new two-dimensional SMoE model which approximates the function derived in Appendix A.3.

For practical purposes, the actual formulas one would need to program to reimplement our proposed approach are marked in bold.

Appendix A.1. Definitions, Conventions and Notations

We assume a right-handed coordinate system, where $+x$ points right and $+y$ points up. Cameras are described by their position vector $\vec{v}$, their rotation matrix $M \in \mathbb{R}^{3\times 3}$, and their projection matrix $P \in \mathbb{R}^{3\times 3}$, which transforms from 3D camera-local coordinates to 2D-homogeneous (i.e., 3D) camera-sensor coordinates. More specifically:

- Cameras look in the $-z$ direction $(0,0,-1)$, such that $M \cdot (0,0,-1)$ corresponds to the world-coordinate-space direction in which the camera looks. Their up direction is the $+y$ direction $(0,1,0)$.
- The camera sensor coordinate space is two-dimensional (after dividing away the third homogeneous component). The sensor x and y coordinates both range from -1 to $+1$.

This means that projecting a 3D point with world-coordinates $\vec{w} \in \mathbb{R}^3$ onto sensor coordinates $\vec{p} \in \mathbb{R}^2$ boils down to

$$\vec{p}\,' := PM^{-1}(\vec{w} - \vec{v}) \quad \in \mathbb{R}^3 \tag{A1}$$

$$\vec{p} := (p'_x/p'_z\,,\; p'_y/p'_z) \in \mathbb{R}^2, \tag{A2}$$

Without loss of generality, we assume the third row of P equals $(0,0,-1)$, such that

$$(P\vec{x})_z = -x_z, \quad \forall \vec{x} \in \mathbb{R}^3, \tag{A3}$$

where $(\bullet)_z$ denotes taking the z-component (i.e., third component) of the vector argument.

Additionally, we introduce a custom notation for constructing submatrices. Given matrix A,

$$A^{(\langle\text{rows}\rangle\ \langle\text{columns}\rangle)} \tag{A4}$$

is made of the rows and columns indicated in the superscript, where $\langle\text{rows}\rangle$ is denoted by letters, and $\langle\text{columns}\rangle$ is denoted by 1-indexed numbers. For example, $A^{(xz24)}$ denotes the result when discarding all rows except for the x and z row, which is the first and third row, and discarding all but the second and fourth column. However, this notation will always be clarified explicitly when used again.

Appendix A.2. Overview and Motivation of the Reduction

Consider a queryable 4D-SMoE model for which the original pixel data originated from cameras located on the plane given by $z = 0$, called *the camera-plane*. Suppose we are given the position $\vec{v}$ of a virtual camera with a field of view corresponding to the desired view to be synthesized. Now, we can pick any pixel coordinate of the field of view,

shoot the corresponding ray with direction $\vec{d}$, and intersect it with the camera plane $z = 0$, such that we obtain a hypothetical original camera position $\vec{\rho}$. The intersection of the ray with the camera plane is visualized in a top-down view in Figure A1. If we consider the optics of the original cameras, we can additionally convert the ray direction $\vec{d}$ to the sensor coordinate of the pixel in this hypothetical camera. The combination of the xy-coordinate of the hypothetical camera and the xy-coordinate of the pixel forms a 4D coordinate and can be used to query the model. This procedure can expressed as a function $\vec{q}_\mathrm{s} \colon \mathbb{R}^2 \to \mathbb{R}^4$, which we call *the mapping function*.

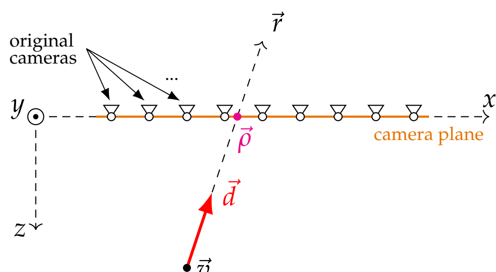


Figure A1. The intersection of a ray $\vec{r}$ with the camera plane. The ray starts in the virtual viewpoint $\vec{v}$ and travels in direction $\vec{d}$ and intersects the camera plane in $\vec{\rho}$.

Evaluating the mapping function $\vec{q}_\mathrm{s}$ for every pixel of the virtual camera yields a corresponding set of *query points*, which are all located on a single two-dimensional surface within the four-dimensional coordinate space of the model. We call this surface *the mapping surface*. (This is similar to the *"viewpoint manifold"* described in "Unstructured Light Fields" [33]). However, in Unstructured Light Fields, the original image data (i.e., pictures) are used instead of a continuous model approximating the image data. As such, Davis et al. [33] came up with a scheme to blend the pixels of the different captured views into the desired view. This pixel-blending scheme based on triangulating and subdividing the viewpoint manifold is a solution to a problem we do not have here. As the SMoE model is continuous, we can also query the model anywhere in between the original views, and the model will yield consistent results. In fact, once an SMoE model is built, all pixel data are forgotten, so intrinsically, there is no distinction between a view corresponding to an original captured viewpoint or an unseen intermediate (in-plane) viewpoint). Constructing the mapping surface in the coordinate space of the model, starting from a virtual camera, is visualized in Figure A2. As can be seen from the visualization in Figure A2, this surface happens to coincide with a plane. However, a crucial observation is that the query points are unevenly spaced, as the ray-plane intersection procedure is not a linear operation. The color values obtained by evaluating the model in these query points form exactly the image as seen by the virtual camera (assuming no volumetric effects disturb the straight-line traveling of light). Put into symbols, the function $\vec{f}_\mathrm{s}$ represents the colors of the virtual view as a function of the pixel coordinate $\vec{s}$:

$$\vec{f}_\mathrm{s}(\vec{s}) \coloneqq \vec{f}(\vec{q}_\mathrm{s}(\vec{s})). \tag{A5}$$

So far, we have merely summarized the light field view synthesis problem, applied to a SMoE model. However, in order to synthesize new views quickly, we wish to minimize the number of computations required. Ideally, we would like to know where within the field of view of the virtual camera, the region of influence of every component is. Suppose we pick a threshold influence value, below which we can neglect the influence of a component. Conceptually, this threshold greatly reduces the number of pixels for which a component is relevant, which would indeed greatly reduce the number of computations required. Selecting such a threshold value $\bar{\alpha}$ gives immediate rise to a region of influence of every component i within the four dimensions of the light field model (as determined by the component center $\vec{\mu}_i$, covariance R_i, sharpness parameter s_i, and the chosen threshold).

However, the view we wish to synthesize is 2D. This means we would need the corresponding 2D region of influence (within the virtual camera's field of view itself) instead. This 2D region of influence can be denoted as the set of screen-space pixel-coordinates $\vec{s}$ for which the component has an opacity above the chosen threshold: $\{\vec{s}\colon \gamma_i(\vec{q}_s(\vec{s})) > \bar{\alpha}\}$. This 2D region of influence of a component can conceptually be visualized as the intersection of the mapping surface and the component's 4D region of influence within the model. This intersection of the mapping surface and the component's influence is visualized in the bottom diagram of Figure A2.

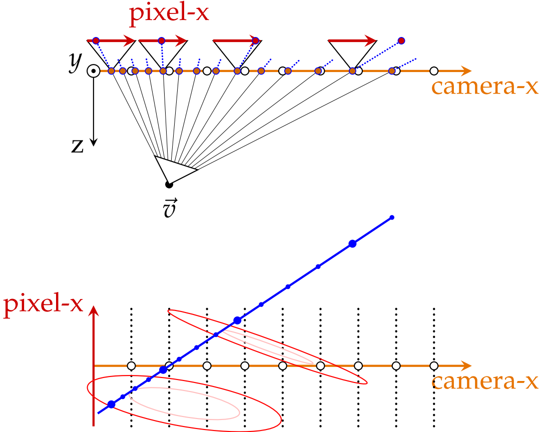


Figure A2. Schematic simulation of obtaining the query points by intersecting the rays coming out of each pixel of the virtual camera in $\vec{v}$ with the plane of original cameras. The top diagram shows a top-down view in real-world, physical dimensions. The bottom diagram visualizes two of the four model dimensions, of which the horizontal one is the same real-world physical x-axis, and the vertical one is the pixel-x coordinate within each camera. The hollow circles represent the positions of the nine original cameras (by means of their xz-coordinate in the top diagram and by their x-coordinate in the bottom diagram). In the top diagram, the rays intersect with the plane of the original cameras at the marked orange dots, which yield the camera-xy coordinates of the query points. Additionally, for four of those intersections, we have drawn the hypothetical cameras to show how a ray can be linked to xy-pixel coordinates of the query points. Those intersections additionally each have a red dot drawn on their pixel-x axis in the top diagram, connected with a blue dotted line to the corresponding orange dot. This connection between the orange and the red dot represents the query point. In the bottom diagram, the blue line is the mapping surface through the model, and the blue dots on it represent the query points. The four intersections for which the hypothetical cameras are drawn have query points that are indicated with bigger blue dots. The dots from the black dotted lines represent the pixels of the pictures the original cameras have taken. The two red ellipses represent example components from the model (by means of their region of influence, which corresponds to a cutoff Mahalanobis distance from their center point). The two fainter ellipses are drawn such that they touch the mapping surface (i.e., they represent the locus of points at the minimal Mahalanobis distance from any point on the mapping surface to the center of the component).

This paper essentially obtains this 2D region of influence, which we will use to evaluate every component only for the relevant pixels. Finding this region accurately and efficiently is important. The proposed technique first finds the center point of it by solving a minimization problem. Then, as a second part, we derive an approximate 2×2 covariance matrix that describes the elliptical shape of this region within the field of view around the center point. Theoretically, we can now use a GPU graphics pipeline to rasterize these regions of influence one component at a time and let the fragment shader calculate Equations (3) and (5) to finally accumulate the results by means of alpha compositing. However, we derive 2D screen-space counterparts of Equations (3) and (5) and rasterize the regions of influence using those instead. Doing so further simplifies the computations

during rasterization, as we reduce the dimensionality of the matrices and vectors involved from 4D to 2D, and we no longer have to evaluate the mapping function for every pixel.

Appendix A.3. Intersecting the Camera Plane

Consider a point $\vec{v} := (v_x, v_y, v_z) \in \mathbb{R}^3$, for which we wish to synthesize a virtual camera viewpoint. Throughout the rest of this paper, the viewpoint is treated as constant, so for notational brevity, it is not always explicitly included as a function argument. Consider the ray $\vec{r}(\lambda)$ originating from this viewpoint $\vec{v}$ going into direction $\vec{d} \in \mathbb{R}^3$:

$$\vec{r}(\lambda) := \vec{v} + \lambda\vec{d} \in \mathbb{R}^3, \qquad \forall \lambda \in \mathbb{R}_{>0}. \tag{A6}$$

Solving the intersection of ray $\vec{r}(\lambda)$ with the plane of original cameras $z = 0$ for λ yields

$$\hat{\lambda}(\vec{d}) = -v_z / d_z \in \mathbb{R}. \tag{A7}$$

Evaluating this particular value of λ in Equation (A6) yields the position of the intersection $\vec{\rho}$:

$$\vec{\rho} := \vec{r}(\hat{\lambda}(\vec{d})) \in \mathbb{R}^3, \tag{A8}$$

which can be interpreted as the 3D coordinate corresponding to a hypothetical camera in the plane of original cameras which would have captured the light ray we are considering. We also need the sensor coordinate $\vec{s}_o$ within the hypothetical camera corresponding to this ray. This is achieved by projecting the ray direction $\vec{d}$ onto that camera using its P_o and M_o matrices (denoted with a subscript o for *original*) (note that the sensor coordinate $\vec{s}_o$ might range from 0 to $\langle$image size$\rangle$, depending on how one chooses to define the pixel coordinates (p_x, p_y) of the original pictures taken by the camera; this is fine, as all these details should be captured in the projection matrix P_o)::

$$\vec{s}_o' := P_o M_o^{-1} \langle \vec{d} \,|\, 0 \rangle \qquad \in \mathbb{R}^3 \tag{A9}$$

$$\vec{s}_o := (s_{o,x}' / s_{o,z}' \,,\, s_{o,y}' / s_{o,z}') \in \mathbb{R}^2. \tag{A10}$$

We can combine the expressions for $\vec{\rho}$ and $\vec{s}_o$ into one vector-function denoted $\vec{q}(\vec{v}, \vec{d})$ ('q' for *query point*):

$$\vec{q}(\vec{v}, \vec{d}) := (\rho_x, \rho_y, s_{o,x}, s_{o,y}) \in \mathbb{R}^4. \tag{A11}$$

We can use the mapping from Equation (A11) to evaluate (or *query*) the model for every ray direction in the field of view of our virtual camera viewpoint and by doing so synthesize the virtual viewpoint.

Next, we can introduce the auxiliary function $\vec{d}(\vec{s})$, which constructs the ray direction that corresponds to a sensor coordinate $\vec{s}$ in our virtual camera:

$$\vec{d}(\vec{s}) := M^{(xyz123)} P^{-1} \langle \vec{s} \,|\, 1 \rangle. \tag{A12}$$

This auxiliary function allows us to define $\vec{q}_s(\vec{s})$ as a shorthand notation, called *the mapping function* (as explained in the Appendix A.2):

$$\vec{q}_s(\vec{s}) := \vec{q}(\vec{v}, \vec{d}(\vec{s})), \quad \forall \vec{s} \in [-1, +1]^2, \tag{A13}$$

In conclusion, using this new function, we can define the function $\vec{f}_s$ describing the color of the incoming light for every sensor coordinate of the virtual camera:

$$\vec{f}_s(\vec{s}) := \vec{f}(\vec{q}_s(\vec{s})), \quad \forall \vec{s} \in [-1, +1]^2. \tag{A14}$$

Appendix A.4. Reducing to a 2D Model

As described in the overview section, we wish to obtain screen-space counterparts of Equations (3) and (5), which essentially describe a single component i:

$$\gamma_{s,i}(\vec{s}) := \gamma_i(\vec{q}_s(\vec{s})), \quad \forall \vec{s} \in [-1,+1]^2, \tag{A15}$$

$$\vec{f}_{s,i}(\vec{s}) := \vec{f}_i(\vec{q}_s(\vec{s})), \quad \forall \vec{s} \in [-1,+1]^2. \tag{A16}$$

The mapping function $\vec{q}(\bullet)$ is non-linear and would make such derivation hard. Therefore, we propose to instead work out these derivations using a linear approximation of the mapping function. Later, it will be clear that when using this linear approximation, formulas can indeed be simplified drastically and yield the desired 2D description of a component.

As a linear approximation of the mapping function, we will derive the first-order Taylor expansion. This means we need to select a vector $\vec{s}_{\text{opt}}$ around which we can develop this Taylor expansion:

$$\vec{q}_s'(\vec{s}) := \vec{q}_{\text{opt}} + G(\vec{s} - \vec{s}_{\text{opt}}) \quad \approx \vec{q}_s(\vec{s}), \tag{A17}$$

$$\text{with} \quad \vec{q}_{\text{opt}} := \vec{q}_s(\vec{s}_{\text{opt}}) \quad \in \mathbb{R}^4, \tag{A18}$$

$$\text{and} \quad G := \left.\frac{\partial \vec{q}_s}{\partial \vec{s}}\right|_{\vec{s}=\vec{s}_{\text{opt}}} \quad \in \mathbb{R}^{4\times 2}. \tag{A19}$$

This linear approximation of the mapping function essentially constructs a tangent plane to the mapping surface around the corresponding query point $\vec{q}_s(\vec{s}_{\text{opt}})$. Note that this procedure is repeated for every component, as $\vec{s}_{\text{opt}}$ can be chosen optimally per component. This can be seen in Figure A2, due to the uneven spacing of the query points, i.e., due to the non-linear parameterization of the mapping function. We propose to select $\vec{s}_{\text{opt}}$ such that the corresponding query point on the mapping surface, *cf.* Equation (A13), minimizes the Mahalanobis distance to the component:

$$\vec{s}_{\text{opt}} := \arg\min_{\vec{s}} \, (\vec{q}_s(\vec{s}) - \vec{\mu}_i)^{\intercal} R_i^{-1} (\vec{q}_s(\vec{s}) - \vec{\mu}_i). \tag{A20}$$

See Figure A3 to visualize how the minimization of the Mahalanobis distance to the mapping surface yields $\vec{q}_{\text{opt}}$.

The next two paragraphs derive closed-form expressions for $\vec{s}_{\text{opt}}$ and G by solving Equation (A20) and evaluating Equation (A19). The last paragraph finally substitutes the Taylor expansion of the mapping surface into the four-dimensional definition of a SMoE component and thereby reduces the dimensionality from four to two.

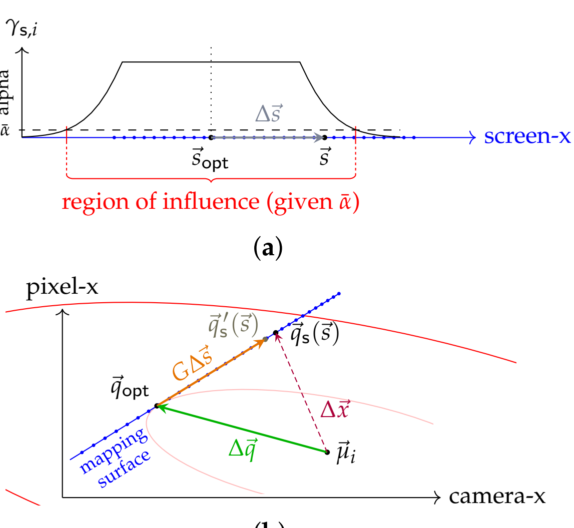


Figure A3. Auxiliary lower-dimensional visualization of the approach taken in this proposed work to calculate the Mahalanobis distance from a certain point to the center of a given component. Note

that subfigure (**a**) represents the intersection of subfigure (**b**) along the mapping surface (indicated by the corresponding shared blue color), where the blue dots correspond on both figures. (**a**) The alpha-function $\gamma_{\mathsf{s},i}$ of a single component in the virtual view. Pixel coordinates are marked with blue dots. The point $\vec{s}$ on the screen represents a point for which we want to reconstruct the color of the SMoE model. Note that this diagram is a 1D simplification, as, in reality, the screen is 2D. The point $\vec{s}_{\mathsf{opt}}$ on the screen corresponds to the point where the Mahalanobis distance is minimal (and thus where the alpha function yields its highest value). The vector $\Delta\vec{s}$ is the difference vector between the $\vec{s}$ and $\vec{s}_{\mathsf{opt}}$ vectors; (**b**) A visualization of how a single component, with center $\vec{\mu}_i$ and covariance matrix R_i, yields its corresponding 1st-order Taylor approximation of $\vec{q}_{\mathsf{s}}$, denoted $\vec{q}_{\mathsf{s}}'$. The mapping surface visualizes the query points corresponding to the screen-space pixel coordinates of the virtual view. The gradient matrix $G \in \mathbb{R}^{4\times 2}$ is the linear factor between $\vec{s} \in \mathbb{R}^2$ and $\vec{q}_{\mathsf{s}}(\vec{s}) \in \mathbb{R}^4$ around $\vec{s}_{\mathsf{opt}}$. To evaluate the alpha-function of the SMoE component for every pixel of the virtual view, the Mahalanobis length of the vector $\Delta\vec{x} := \vec{q}_{\mathsf{s}}(\vec{s}) - \vec{\mu}_i$ is required for every $\vec{s}$. However, the proposed method will actually compute $\vec{q}_{\mathsf{s}}'(\vec{s})$ instead of $\vec{q}_{\mathsf{s}}(\vec{s})$. Per component, the proposed method essentially breaks down the vector $\Delta\vec{x}$ into the constant vector $\Delta\vec{q}$ and a variable vector $G\Delta\vec{s}$ aligned with the mapping surface, where $\Delta\vec{s} := \vec{s} - \vec{s}_{\mathsf{opt}}$. The spacing of the query points is actually non-uniform, and yet the linear Taylor approximation $\vec{q}_{\mathsf{s}}'$ models the spacing of the query points as uniform. As such, a slight discrepancy can be observed between $\vec{q}_{\mathsf{s}}'(\vec{s})$ and $\vec{q}_{\mathsf{s}}(\vec{s})$. Similarly, the computed Mahalanobis cut-off distance of the region of influence will be slightly off. As such, the actual region of influence will be underestimated towards the center of the view and overestimated towards the edges of the view. This diagram also helps to illustrate why $\Delta\vec{q}$ and $G\Delta\vec{s}$ are orthogonal with respect to the covariance R_i. By mentally transforming the whole diagram linearly such that the drawn ellipses end up being perfect circles and the covariance becomes the identity matrix, one can imagine these two vectors are indeed orthogonal in the classical Euclidean way. This is precisely because the inner ellipse touches the mapping surface. This was achieved by finding the point $\vec{q}_{\mathsf{opt}}$ on the mapping surface that minimized Mahalanobis distance to the component.

Appendix A.4.1. Solving the Minimization to Obtain $\vec{s}_{\mathsf{opt}}$

This paragraph solves the minimization problem described in Equation (A20). The minimization happens with respect to $\vec{s}$. However, when calculating the derivative of the expression and solving for equality to zero (i.e., the typical way to solve optimizations), the expressions involved would grow very complex due to the many non-linearities and repeated application of the chain rule. To overcome this, we will rewrite this minimization problem as an equivalent one. Introduced earlier, Equation (A12) defines a mapping between $\vec{d}$ and $\vec{s}$. Remember how $\vec{d}$ is the ray direction, and it is defined up to a *positive* scaling factor. We can remove this freedom by rescaling it such that we obtain -1 for the z-component (as the camera looks at the $-z$ direction). However, if this factor turns out to be negative, our ray direction would invert. This is not indented, as this means the virtual camera would *look backward*, i.e., towards the opposite direction when compared to the original cameras. When implementing an actual implementation, we must check for this phenomenon, and discard the result in that case. For more details, see Section 3.2. However, if this issue does not occur, we can define a 1-to-1 mapping between the *normalized* ray direction $\vec{d}_{\mathsf{n}} \in \mathbb{R}^2$ and the sensor coordinate $\vec{s} \in \mathbb{R}^2$:

$$\vec{d}_{\mathsf{n}}(\vec{s}) := -\big(\vec{d}(\vec{s})\big)_{\mathrm{xy}} \Big/ \big(\vec{d}(\vec{s})\big)_{\mathrm{z}} \quad \in \mathbb{R}^2, \tag{A21}$$

$$\vec{s}\,'(\vec{d}_{\mathsf{n}}) := PM^{-1}\langle \vec{d}_{\mathsf{n}} \,|\, -1 \,|\, 0 \rangle \quad \in \mathbb{R}^3, \tag{A22}$$

$$\vec{s}(\vec{d}_{\mathsf{n}}) := (\vec{s}\,'(\vec{d}_{\mathsf{n}}))_{\mathrm{xy}} \Big/ (\vec{s}\,'(\vec{d}_{\mathsf{n}}))_{\mathrm{z}} \in \mathbb{R}^2. \tag{A23}$$

This means Equation (A20) can be optimized with respect to $\vec{d}_{\mathsf{n}}$ instead.

Next, all the original cameras are assumed to be organized on the $z = 0$ plane with identical orientation looking into the world-space $-z$ direction. More precisely, this means

that the upper left 3×3 submatrix of their M_o is the identity matrix, while the fourth column contains the translational component (*i.e.*, the camera position). However, as can be seen in Equation (A9), the fourth component equals zero, which causes the translational component to be dropped, and Equation (A9) to simplify to

$$\vec{s}_\text{o}' = P_\text{o}\langle \vec{d} \,|\, 0\rangle \quad \in \mathbb{R}^3, \tag{A24}$$

in which we can choose $\vec{d} = \langle \vec{d}_\text{n} \,|\, {-1}\rangle$, which, using the property in Equation (A3), leads to

$$\vec{s}_\text{o}' = P_\text{o}\langle \vec{d}_\text{n} \,|\, {-1} \,|\, 0\rangle, \tag{A25}$$

$$\text{with} \quad s_{\text{o,z}}' = 1. \tag{A26}$$

Finally, we have all ingredients to express $\vec{q}_\text{s}$ from Equation (A13) as a function of $\vec{d}_\text{n}$ by choosing $\vec{d} = \langle \vec{d}_\text{n} \,|\, {-1}\rangle$, which gives

$$\begin{aligned}\vec{q}_\text{s}(\vec{s}) = \begin{bmatrix} \rho_\text{x} \\ \rho_\text{y} \\ s_\text{o,x} \\ s_\text{o,y} \end{bmatrix} &= \begin{bmatrix} v_\text{x} - (v_\text{z}/d_\text{z})d_\text{x} \\ v_\text{y} - (v_\text{z}/d_\text{z})d_\text{y} \\ s_{\text{o,x}}'/s_{\text{o,z}}' \\ s_{\text{o,y}}'/s_{\text{o,z}}' \end{bmatrix} = \begin{bmatrix} v_\text{x} + v_\text{z}\vec{d}_\text{n,x} \\ v_\text{y} + v_\text{z}\vec{d}_\text{n,y} \\ s_{\text{o,x}}' \\ s_{\text{o,y}}' \end{bmatrix} \\ &= \begin{bmatrix} v_\text{x} + v_\text{z}\vec{d}_\text{n,x} \\ v_\text{y} + v_\text{z}\vec{d}_\text{n,y} \\ P_\text{o}^{(x123)}\langle \vec{d}_\text{n} \,|\, {-1}\rangle \\ P_\text{o}^{(y123)}\langle \vec{d}_\text{n} \,|\, {-1}\rangle \end{bmatrix} = D \cdot \vec{d}_\text{n}(\vec{s}) + \vec{e},\end{aligned} \tag{A27}$$

with:

$$D := \begin{bmatrix} v_\text{z} & 0 \\ 0 & v_\text{z} \\ P_\text{o}^{(x1)} & P_\text{o}^{(x2)} \\ P_\text{o}^{(y1)} & P_\text{o}^{(y2)} \end{bmatrix} \quad \text{and} \quad \vec{e} := \begin{bmatrix} v_\text{x} \\ v_\text{y} \\ -P_\text{o}^{(x3)} \\ -P_\text{o}^{(y3)} \end{bmatrix}. \tag{A28}$$

Here, the notation $A^{(x123)} \in \mathbb{R}^{1\times 3}$ is used to denote the 1×3 matrix containing the elements of columns 1, 2, 3 at the first (x) row of A. Likewise, $A^{(y3)} \in \mathbb{R}$ is the element of A at the second row, third column, etc.

Going back to our minimization objective of Equation (A20), we can now instead minimize over $\vec{d}_\text{n}$, while introducing an auxiliary function $\vec{u}(\vec{d}_\text{n})$:

$$\vec{d}_\text{n,opt} = \underset{\vec{d}_\text{n}}{\arg\min}\ \big(\vec{u}(\vec{d}_\text{n})\big)^\intercal R_i^{-1}\big(\vec{u}(\vec{d}_\text{n})\big), \tag{A29}$$

$$\text{with} \quad \vec{u}(\vec{d}_\text{n}) := D \cdot \vec{d}_\text{n}(\vec{s}) + \vec{e} - \vec{\mu}_i. \tag{A30}$$

Next, we will calculate the derivative of the minimization objective in Equation (A29) and set it equal to zero, while using the fact that R_i and R_i^{-1} are symmetrical. Doing this using a numerator layout gives

$$\begin{aligned} \vec{0}^\intercal &= 2\Big(\vec{u}(\vec{d}_\text{n})\Big)^\intercal R_i^{-1}\,\frac{\partial \vec{u}(\vec{d}_\text{n})}{\partial \vec{d}_\text{n}} \\ \iff \quad \vec{0}^\intercal &= \Big(D\vec{d}_\text{n} + (\vec{e} - \vec{\mu}_i)\Big)^\intercal R_i^{-1} D \\ \iff -D^\intercal R_i^{-1} D\vec{d}_\text{n} &= D^\intercal R_i^{-1}(\vec{e} - \vec{\mu}_i) \\ \iff \quad \vec{d}_\text{n,opt} &= -(D^\intercal R_i^{-1} D)^{-1} D^\intercal R_i^{-1}(\vec{e} - \vec{\mu}_i). \end{aligned} \tag{A31}$$

Note that the second equivalence transposes both sides of the equation.

Finally, we can obtain the optimal $\vec{s}_{\text{opt}}$ by evaluating Equation (A23) with $\vec{d}_{\text{n,opt}}$. The corresponding optimal query point $\vec{q}_{\text{opt}}$ could be found by applying the definition in Equation (A18), but we now obtained a shortcut via Equation (A28) as we now have $\vec{d}_{\text{n,opt}}$:

$$\vec{q}_{\text{opt}} = D\vec{d}_{\text{n,opt}} + \vec{e}. \tag{A32}$$

Appendix A.4.2. Calculating the Derivative G

Given that we found the tangent point $\vec{q}_{\text{opt}}$ in Equation (A32) of Appendix A.4.1, we are seeking the orientation of the plane. As discussed earlier, we will use the first-degree Taylor polynomial of the mapping function $\vec{q}_{\text{s}}(\bullet)$ from Equation (A27) around the optimal sensor coordinate $\vec{s}_{\text{opt}}$ corresponding to the tangent point $\vec{q}_{\text{opt}}$. Therefore, we need the derivative $G \in \mathbb{R}^{4\times 2}$ of $\vec{q}_{\text{s}}(\vec{s})$ with respect to $\vec{s}$ by applying the chain rule on Equations (A21) and (A27):

$$\begin{aligned} G &:= \left.\frac{\partial \vec{q}_{\text{s}}}{\partial \vec{s}}\right|_{\vec{s}=\vec{s}_{\text{opt}}} = \left.\frac{\partial \vec{q}_{\text{s}}}{\partial \vec{d}_{\text{n}}}\frac{\partial \vec{d}_{\text{n}}}{\partial \vec{s}}\right|_{\vec{s}=\vec{s}_{\text{opt}}} \\ &= D\left(\frac{-1}{d_{\text{z}}(\vec{s}_{\text{opt}})}\frac{\partial \vec{d}_{\text{xy}}}{\partial \vec{s}} + \vec{d}_{\text{xy}}\frac{1}{(d_{\text{z}}(\vec{s}_{\text{opt}}))^2}\frac{\partial d_{\text{z}}}{\partial \vec{s}}\right), \end{aligned} \tag{A33}$$

where d_{z} is the z-component of $\vec{d}(\vec{s})$, as given by Equation (A12).

In order to derive $\frac{\partial \vec{d}_{\text{xy}}}{\partial \vec{s}}$ and $\frac{\partial d_{\text{z}}}{\partial \vec{s}}$, let us first rewrite $\vec{d}_{\text{xy}}$ and d_{z} in matrix form, using Equation (A12):

$$\vec{d}_{\text{xy}}(\vec{s}) = M^{(xy123)}P^{-1}(I_{32}\vec{s} + \vec{i}_3), \tag{A34}$$

$$d_{\text{z}}(\vec{s}) = M^{(z123)}P^{-1}\ (I_{32}\vec{s} + \vec{i}_3), \tag{A35}$$

where $I_{32} := \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$ and $\vec{i}_3 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$. With this matrix notation, the derivates are easier to express using matrix calculus:

$$\frac{\partial \vec{d}_{\text{xy}}}{\partial \vec{s}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} MP^{-1}I_{32} \quad \in \mathbb{R}^{2\times 2}, \tag{A36}$$

$$\frac{\partial d_{\text{z}}}{\partial \vec{s}} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} MP^{-1}I_{32} \quad \in \mathbb{R}^{1\times 2}. \tag{A37}$$

In conclusion, we found the tangent point $\vec{q}_{\text{opt}}$, and the gradient matrix G. These two allow us to define the linear approximation of $\vec{q}_{\text{s}}$, which we denote by $\vec{q}_{\text{s}}'$:

$$\vec{q}_{\text{s}}'(\vec{s}) := \vec{q}_{\text{opt}} + G(\vec{s} - \vec{s}_{\text{opt}}) \quad \approx \vec{q}_{\text{s}}(\vec{s}). \tag{A38}$$

Appendix A.4.3. Transforming a Component

We obtained a per-component linear approximation for the $\vec{q}_{\text{s}}$ function in Equation (A38). Plugging this approximation in Equation (3), we obtain

$$\gamma_{\text{s},i}'(\vec{s}) := \gamma_i(\vec{q}_{\text{s}}'(\vec{s})) \tag{A39}$$

For the sake of notational simplicity, we will first simplify the expression for the squared Mahalanobis distance (denoted M^2 from now) before considering the complete formula:

$$\begin{aligned} M^2 &:= (\vec{q}_s'(\vec{s}) - \vec{\mu}_i)^\intercal R_i^{-1}(\vec{q}_s'(\vec{s}) - \vec{\mu}_i) \\ &= (\vec{q}_{\text{opt}} + G(\vec{s} - \vec{s}_{\text{opt}}) - \vec{\mu}_i)^\intercal R_i^{-1}(\vec{q}_{\text{opt}} + G(\vec{s} - \vec{s}_{\text{opt}}) - \vec{\mu}_i) \\ &\Updownarrow \quad \Delta\vec{q} := \vec{q}_{\text{opt}} - \vec{\mu}_i \quad \text{and} \quad \Delta\vec{s} := \vec{s} - \vec{s}_{\text{opt}} \\ &= (\Delta\vec{q} + G\Delta\vec{s})^\intercal R_i^{-1}(\Delta\vec{q} + G\Delta\vec{s}) \\ &= (\Delta\vec{q})^\intercal R_i^{-1}\Delta\vec{q} + (\Delta\vec{q})^\intercal R_i^{-1}G\Delta\vec{s} + (G\Delta\vec{s})^\intercal R_i^{-1}\Delta\vec{q} + (G\Delta\vec{s})^\intercal R_i^{-1}G\Delta\vec{s} \\ &= (\Delta\vec{q})^\intercal R_i^{-1}\Delta\vec{q} + 2(\Delta\vec{q})^\intercal R_i^{-1}G\Delta\vec{s} + (\Delta\vec{s})^\intercal(G^\intercal R_i^{-1}G)\Delta\vec{s} \\ &\Updownarrow \quad R_i^{-1}\Delta\vec{q} \text{ and } G\Delta\vec{s} \text{ are orthogonal} \\ &= (\Delta\vec{q})^\intercal R_i^{-1}\Delta\vec{q} \; + \; (\Delta\vec{s})^\intercal(G^\intercal R_i^{-1}G)\Delta\vec{s} \end{aligned} \tag{A40, A41}$$

The intuitive explanation for why $R_i^{-1}\Delta\vec{q}$ and $G\Delta\vec{s}$ are orthogonal is visualized and explained in Figure A3. Continuing on Equation (A39) using Equation (A41) yields

$$\gamma_{s,i}'(\vec{s}) = \alpha_i \exp\left(-\frac{1}{2}\max\left(0, M^2 - 2s_i\right)\right). \tag{A42}$$

We conclude that the two-dimensional region on the screen is centered around $\vec{s}_{\text{opt}}$ and has covariance $\hat{R}_i := (G^\intercal R_i^{-1}G)^{-1}$.

Finally, we apply the same evaluation replacing $\vec{q}_s$ with the $\vec{q}_s'$ approximation in the $\vec{f}_{s,i}$ of Equation (5) to obtain the two-dimensional color-approximation function of the component:

$$\begin{aligned} \vec{f}_{s,i}'(\vec{s}) &:= \vec{\xi}_i + W_i(\vec{q}_s'(\vec{s}) - \vec{\mu}_i) \\ &= \vec{\xi}_i + W_i(\vec{q}_{\text{opt}} + G(\vec{s} - \vec{s}_{\text{opt}}) - \vec{\mu}_i) \\ &= \vec{\xi}_i + W_i(\vec{q}_{\text{opt}} - \vec{\mu}_i) + W_iG(\vec{s} - \vec{s}_{\text{opt}}). \end{aligned} \tag{A43}$$

In conclusion, we derived the parameters and formulas of a two-dimensional component, which approximates how the original four-dimensional component impacts the view-specific render in Equations (A42) and (A43). This two-dimensional model represents the visual output image of a virtual camera. This means we are free to configure the camera position, orientation, and projection matrix. Using the above mathematics, we can derive the corresponding 2D model, which we can then efficiently render on the screen, with the technique described in the next section.

Appendix A.5. Numerical Stability

The covariance matrix plays a central role throughout these calculations but has a major practical issue associated with it. The numerical values can be quite extreme, as the eigenvalues are proportional to the squared size of the component's main axes (just like the variance is the square of the standard deviation). For an actual implementation, the combination of both extremely small and large values in the covariance matrix is problematic for numerical precision in a computer. To overcome this, the Cholesky decomposition of the covariance matrix is preferred and used where possible. Let us introduce the lower triangular matrix $L_i \in \mathbb{R}^{4\times4}$ as the Cholesky decomposition of R_i, satisfying $R_i = L_iL_i^\intercal$. Now, several important expressions can be rewritten. Equation (A31) can be rewritten as:

$$\begin{aligned} \vec{d}_{n,\text{opt}} &= -(D^\intercal R_i^{-1}D)^{-1}D^\intercal R_i^{-1}(\vec{e} - \vec{\mu}_i) \\ &= -(D^\intercal (L_i^\intercal)^{-1}L_i^{-1}D)^{-1}D^\intercal (L_i^\intercal)^{-1}L_i^{-1}(\vec{e} - \vec{\mu}_i) \\ &= -((L_i^{-1}D)^\intercal(L_i^{-1}D))^{-1}(L_i^{-1}D)^\intercal L_i^{-1}(\vec{e} - \vec{\mu}_i). \end{aligned} \tag{A44}$$

Note that this can still be programmed using only two matrix inversions and four matrix–matrix multiplications. Additionally, L_i^{-1} can be precomputed and stored in the vertex buffer instead of R_i, saving the at-runtime computation of the Cholesky decomposition and the matrix inversion. Equation (A41) can be rewritten as

$$M^2 = \left\| L_i^{-1} \Delta\vec{q} \right\|^2 + \left\| (L_i^{-1} G) \Delta\vec{s} \right\|^2, \tag{A45}$$

although this formula is nowhere explicitly implemented in our implementation. Despite our efforts to improve numerical stability, we have found it beneficial to additionally implement most of the geometry shader using double-precision floating-point operations.

Appendix A.6. Approximation Compensation

We introduce a small improvement to the reconstruction quality of the proposed technique. We attempt to recover from the Taylor approximation of the mapping function. Upon obtaining the vertex positions of the ellipse that will be rasterized on the screen, we have calculated them, expecting them to have a fixed Mahalanobis distance $\bar{M}$ to the center $\vec{s}_{\mathsf{opt}}$. The proposed improvement takes this screen-space vertex position $\vec{s}$ and recomputes the exact query point $\vec{q}_{\mathsf{s}}(\vec{s})$. The Mahalanobis distance from $\vec{q}_{\mathsf{s}}(\vec{s})$ to $\vec{q}_{\mathsf{opt}}$ is compared to the expected value $\bar{M}$. Based on the proportional difference between the expected and actual length, the vertex position $\vec{s}$ is rescaled to $\vec{s}'$, with the center of the ellipse $\vec{s}_{\mathsf{opt}}$ as the origin for scaling:

$$\vec{s}' := \vec{s}_{\mathsf{opt}} + (\vec{s} - \vec{s}_{\mathsf{opt}}) \frac{\bar{M}}{\left\| L_i^{-1} (\vec{q}(\vec{s}) - \vec{q}_{\mathsf{opt}}) \right\|}. \tag{A46}$$

The actual rasterized ellipse uses vertex position $\vec{s}'$ instead. Note that it is beneficial to have a vertex at the center of the ellipse and make the triangle fan around this center point to still have a piecewise linear interpolation of the mapping function (This is, however, not trivial given that OpenGL geometry shaders do not support *triangle fans* as output primitive. In our implementation, we do not have a center vertex, and instead emit a triangle strip that zigzags to the vertices from side to side of the ellipse). Studying Figure A3b is recommended to understand the idea of this proposed improvement.

Appendix A.7. Obtaining z-Depth

We can obtain an approximate depth map, as shown in Figure A4 for the current viewpoint of the user using the same component-based SMoE model. This can be carried out by replacing the color approximation function of Equation (5) with the z-depth for every component (or a color representing the depth). Obtaining the z-depth of a component can be carried out through the following steps.

We calculate the parallax matrix V_i:

$$V_i := R_{\mathsf{sc},i} \, R_{\mathsf{cc},i}^{-1} \quad \in \mathbb{R}^{2\times 2}, \tag{A47}$$

where the subscripts c and s stand for "camera" and "screen". To be precise, $R_{\mathsf{sc},i}$ refers to the 2×2 lower left submatrix of R_i. Likewise, $R_{\mathsf{cc},i}^{-1}$ is the matrix inverse of $R_{\mathsf{cc},i}$, which is the 2×2 upper left submatrix of R_i. This parallax matrix V_i describes the xy-movement of the component on the sensor given an xy-movement of the camera in the plane. This relationship can be used to obtain the estimated signed distance $\bar{z}_i$ from the plane of cameras to the represented object by combining it with the field of view of the original cameras:

$$\bar{z}_i := -\frac{1}{2} \left(\frac{P_{\mathsf{o}_{1,1}}}{V_{i_{1,1}}} + \frac{P_{\mathsf{o}_{2,2}}}{V_{i_{2,2}}} \right) \quad \in \mathbb{R}. \tag{A48}$$

As this signed distance is measured from the xy-plane, it corresponds to the z-component of the reconstructed point $\vec{r}_i$ in 3D. The signed distance can be used as the multiplier for the normalized ray direction, starting from the intersection of the ray with the camera plane:

$$\vec{r}_i := \langle \vec{q}_{\mathsf{opt,c}} \,|\, 0 \rangle + \bar{z}_i \vec{d} \quad \in \mathbb{R}^3. \tag{A49}$$

Now we have the virtual position of the component as if it lived in 3D-space instead of on the plane of cameras. The last thing to do is transform this point with the projection and model matrix and take the z-component, which is the signed z-depth relative to the user's viewpoint:

$$\bar{z}_i' := (PM^{-1}\vec{r}_i)_{\mathsf{z}} \quad \in \mathbb{R}. \tag{A50}$$

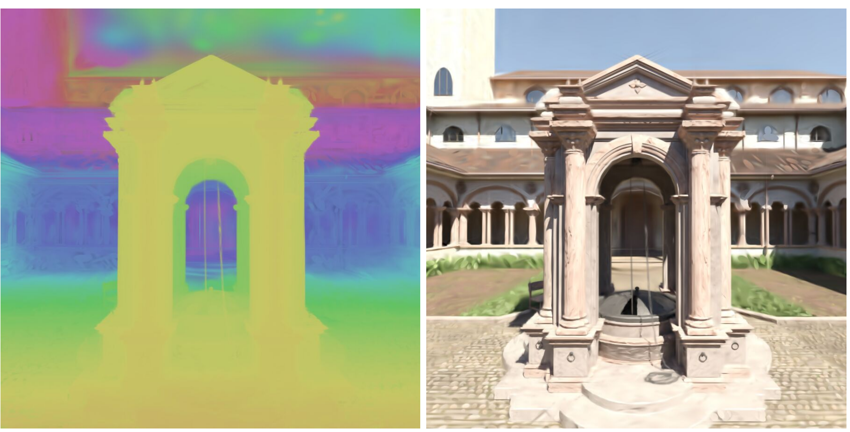


Figure A4. Depth map rendered for the *lone monk* scene, along with the corresponding color rendition. Depth values are mapped onto a rotating color hue. As a side note, observe that the sky has incorrect depth, but this is a modeling artifact.

Appendix A.8. Dropping Components

Given that the user can navigate freely, it is possible that the user can get 'behind' an object. This would result in the system producing a point-mirrored projection of the object that is behind them. This can be mitigated by not rendering the corresponding components. These components are identified by a negative depth according to Equation (A50). One should be cautious when considering this trick, as it might discard components that were needed. For example, a light field capture of a scene featuring a glass lens can cause the refracted light to have opposite parallax. This would yield the reconstructed 3D position $\vec{r}_i$ from Equation (A49) to end up behind the user. This would cause the component to be discarded, whereas it was still visible and modeled a complex light-refracting phenomenon. To mitigate this, one can decide to only discard components if the reconstructed depth has the opposite sign of the depth $\bar{z}_i$ from Equation (A48), as we can assume that everything is visible in the plane, as that is exactly where the light was captured.

References

1. Gao, R.; Qi, Y. A Brief Review on Differentiable Rendering: Recent Advances and Challenges. *Electronics* **2024**, *13*, 3546. [CrossRef]
2. Müller, T.; Evans, A.; Schied, C.; Keller, A. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *ACM Trans. Graph.* **2022**, *41*, 102:1–102:15. [CrossRef]
3. Wen, C.; Zhang, Y.; Li, Z.; Fu, Y. Pixel2Mesh++: Multi-View 3D Mesh Generation via Deformation. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), Seoul, Republic of Korea, 27 October–2 November 2019.
4. Lin, C.H.; Wang, O.; Russell, B.C.; Shechtman, E.; Kim, V.G.; Fisher, M.; Lucey, S. Photometric Mesh Optimization for Video-Aligned 3D Object Reconstruction. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, 15–20 June 2019.
5. Rogge, S.; Schiopu, I.; Munteanu, A. Depth Estimation for Light-Field Images Using Stereo Matching and Convolutional Neural Networks. *Sensors* **2020**, *20*, 6188. [CrossRef] [PubMed]

6. Zerman, E.; Ozcinar, C.; Gao, P.; Smolic, A. Textured Mesh vs Coloured Point Cloud: A Subjective Study for Volumetric Video Compression. In Proceedings of the 2020 12th International Conference on Quality of Multimedia Experience, QoMEX 2020, Athlone, Ireland, 26–28 May 2020. [CrossRef]
7. Microsoft. Microsoft Mixed Reality Capture Studio. 2022. Available online: https://news.microsoft.com/source/features/work-life/microsoft-mixed-reality-capture-studios-create-holograms-to-educate-and-entertain/ (accessed on 14 October 2024).
8. 8i. 8i Studio. 2022. Available online: https://8i.com (accessed on 14 October 2024).
9. Buehler, C.; Bosse, M.; McMillan, L.; Gortler, S.; Cohen, M. Unstructured Lumigraph Rendering. In Proceedings of the Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '01, New York, NY, USA, 12–17 August 2001; pp. 425–432. [CrossRef]
10. Kellnhofer, P.; Jebe, L.; Jones, A.; Spicer, R.; Pulli, K.; Wetzstein, G. Neural Lumigraph Rendering. In Proceedings of the CVPR, Nashville, TN, USA, 20–25 June 2021.
11. Overbeck, R.S.; Erickson, D.; Evangelakos, D.; Pharr, M.; Debevec, P. A system for acquiring, processing, and rendering panoramic light field stills for virtual reality. In Proceedings of the SIGGRAPH Asia 2018 Technical Papers, SIGGRAPH Asia 2018, Tokyo, Japan, 4–7 December 2018 ; Volume 37, p. 15. [CrossRef]
12. Broxton, M.; Flynn, J.; Overbeck, R.; Erickson, D.; Hedman, P.; Duvall, M.; Dourgarian, J.; Busch, J.; Whalen, M.; Debevec, P. Immersive light field video with a layered mesh representation. *ACM Trans. Graph.* **2020**, *39*, 15. [CrossRef]
13. Boyce, J.M.; Dore, R.; Dziembowski, A.; Fleureau, J.; Jung, J.; Kroon, B.; Salahieh, B.; Vadakital, V.K.M.; Yu, L. MPEG Immersive Video Coding Standard. *Proc. IEEE* **2021**, *109*, 1521–1536. [CrossRef]
14. Le Pendu, M.; Guillemot, C.; Smolic, A. A Fourier Disparity Layer Representation for Light Fields. *IEEE Trans. Image Process.* **2019**, *28*, 5740–5753. [CrossRef] [PubMed]
15. Dib, E.; Pendu, M.L.; Guillemot, C. Light Field Compression Using Fourier Disparity Layers. In Proceedings of the Proceedings—International Conference on Image Processing, ICIP, Taipei, Taiwan, 22–25 September 2019. [CrossRef]
16. Mildenhall, B.; Srinivasan, P.P.; Tancik, M.; Barron, J.T.; Ramamoorthi, R.; Ng, R. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In Proceedings of the Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Glasgow, UK, 23–28 August 2020; Volume 12346 LNCS, pp. 405–421. [CrossRef]
17. Qin, S.; Xiao, J.; Ge, J. Dip-NeRF: Depth-Based Anti-Aliased Neural Radiance Fields. *Electronics* **2024**, *13*, 1527. [CrossRef]
18. Dong, B.; Chen, K.; Wang, Z.; Yan, M.; Gu, J.; Sun, X. MM-NeRF: Large-Scale Scene Representation with Multi-Resolution Hash Grid and Multi-View Priors Features. *Electronics* **2024**, *13*, 844. [CrossRef]
19. Barron, J.T.; Mildenhall, B.; Tancik, M.; Hedman, P.; Martin-Brualla, R.; Srinivasan, P.P. Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields. *arXiv* **2021**, arXiv:2103.13415.
20. Barron, J.T.; Mildenhall, B.; Verbin, D.; Srinivasan, P.P.; Hedman, P. Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields. *arXiv* **2022**, arXiv:2111.12077.
21. Hu, W.; Wang, Y.; Ma, L.; Yang, B.; Gao, L.; Liu, X.; Ma, Y. Tri-MipRF: Tri-Mip Representation for Efficient Anti-Aliasing Neural Radiance Fields. In Proceedings of the ICCV, Paris, France, 1–6 October 2023.
22. Verhack, R.; Sikora, T.; Lange, L.; Jongebloed, R.; Van Wallendael, G.; Lambert, P. Steered mixture-of-experts for light field coding, depth estimation, and processing. In Proceedings of the Proceedings—IEEE International Conference on Multimedia and Expo, Hong Kong, China, 10–14 July 2017; pp. 1183–1188. [CrossRef]
23. Verhack, R.; Sikora, T.; Van Wallendael, G.; Lambert, P. Steered Mixture-of-Experts for Light Field Images and Video: Representation and Coding. *IEEE Trans. Multimed.* **2020**, *22*, 579–593. [CrossRef]
24. Bochinski, E.; Jongebloed, R.; Tok, M.; Sikora, T. Regularized gradient descent training of steered mixture of experts for sparse image representation. In Proceedings of the 2018 25th IEEE International Conference on Image Processing (ICIP), Athens, Greece, 7–10 October 2018; pp. 3873–3877. [CrossRef]
25. Liu, B.; Zhao, Y.; Jiang, X.; Wang, S. Three-dimensional Epanechnikov mixture regression in image coding. *Signal Process.* **2021**, *185*, 108090. [CrossRef]
26. Verhack, R.; Sikora, T.; Lange, L.; Van Wallendael, G.; Lambert, P. A universal image coding approach using sparse steered Mixture-of-Experts regression. In Proceedings of the Proceedings—International Conference on Image Processing, ICIP, Anchorage, AK, USA, 25–28 September 2016; pp. 2142–2146. [CrossRef]
27. Kerbl, B.; Kopanas, G.; Leimkühler, T.; Drettakis, G. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Trans. Graph.* **2023**, *42*, 139. [CrossRef]
28. Huang, B.; Yu, Z.; Chen, A.; Geiger, A.; Gao, S. 2D Gaussian Splatting for Geometrically Accurate Radiance Fields. In Proceedings of the SIGGRAPH 2024 Conference Papers, Denver, CO, USA, 28 July–1 August 2024. [CrossRef]
29. Courteaux, M.; Artois, J.; De Pauw, S.; Lambert, P.; Van Wallendael, G. SILVR: A Synthetic Immersive Large-Volume Plenoptic Dataset. In Proceedings of the 13th ACM Multimedia Systems Conference (MMSys '22), New York, NY, USA, 14–17 June 2022. [CrossRef]
30. Doyden, D.; Boisson, G.; Gendrot, R. *[MPEG-I Visual] New Version of the Pseudo-Rectified TechnicolorPainter Content*; Document ISO/IEC JTC1/SC29/WG11 MPEG/M43366; Technicolor-Armand Langlois: Ljublana, Slovenia , 2018.
31. Jung, J.; Boissonade, P. *[MPEG-I Visual] Proposition of New Sequences for Windowed-6DoF Experiments on Compression, Synthesis, and Depth Estimation*; Standard ISO/IEC JTC1/SC29/WG11 MPEG/M43318; Orange Labs: Anaheim, CA, USA, 2018.

32. Blender Institute. Agent 327: Operation Barbershop. 2017. Available online: https://studio.blender.org/films/agent-327/ (accessed on 14 October 2024).
33. Davis, A.; Levoy, M.; Durand, F. Unstructured Light Fields. *Comput. Graph. Forum* **2012**, *31*, 305–314. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.