Baking Neural Radiance Field Video to Immersive Layered Depth Images

Forrest Briggs
Lifecast Incorporated
fbriggs@lifecastvr.com

Lawrence Neal
Lifecast Incorporated
nealla@lwneal.com

Last Updated
May 15, 2024

Abstract

We present a system for reconstructing, compressing, and real-time rendering of NeRF (light field) video, with a more practical format and orders of magnitude faster processing than prior work [5]. Current immersive 3D VR video is stereoscopic, and does not respond to translation in the user's viewpoint, which can cause motion sickness due to a conflict between the user's perception of motion according to their eye and vestibular system. To avoid motion sickness, novel view rendering in real-time with 6 degrees of freedom is required. We accomplish this by baking neural radiance fields into layered depth images in inflated equiangular projection, with inverse depth maps stored via a 12-bit error correcting code in an 8-bit/channel container. The final representation can be compressed using conventional h264, h265, or ProRes, streamed over the internet, edited in existing video tools, and rendered in real-time on mobile VR devices, in a web browser, or in game engines such as Unity and Unreal (we provide open source implementations for all of these platforms). The proposed methods are simple and compatible with any 3D or 4D radiance field. As part of our evaluation, we provide interactive demos of several scenes, some of which are static and captured with a regular phone, and others are video captured with an array of 46 synchronized cameras on a hemispherical dome.

1. Introduction

We are interested in a practical system for capturing, rendering, editing, and deploying "immersive volumetric video." By immersive, we mean that the video covers the entire field of view of mobile VR/AR/MR devices while allowing some motion for the user. In this work we are focused on fields of view close to 180° (as opposed to 360°, which is another flavor of immersive video). We use the term "volumetric" loosely to convey the intuition that it is a type of 3D video which supports photorealistic novel view synthesis from arbitrary poses with 6 degrees of freedom (6DOF). Photorealistic implies handling complex phenomena that appear in the real world, such as thin structures, partially transparent materials, and view-direction dependent effects. Different from most other "volumetric" video, we focus on inside-out capture of hemispherical scenes instead of outside-in capture of a single subject. The problem we address might also be called light field video, or 4D/time-varying neural radiance fields. By video we simply mean we must support capture and playback of time-varying scenes, although the proposed methods work for static capture as well.

We aim to create a system which meets several requirements (prior work has addressed some but not all of these simultaneously):

- Photorealistic 6DOF novel view synthesis with near 180° field of view, within a small viewing volume (about 0.5m radius).
- Video can be compressed using only universally supported features of conventional codecs such as h264.
- Video format should be possible to edit (at least basic cuts) in existing video tools.
- Can be decoded and rendered in real time on a wide variety of platforms with relatively little GPU power.
- The spatial resolution of the video must be high enough to appear sharp in VR displays.
- Processing and encoding cannot be prohibitively slow.
- Static or dynamic scenes can be captured with any camera system a user has at their disposal, ranging from a phone, to a light field array.

Current-generation 3D VR video formats such as VR180 and omnidirectional stereo are immersive, and are often rendered by projecting a texture for the left and right eye on spherical geometry that is far away. This approach results in the user seeing stereoscopic views which respond only to their head rotation, but not translation (as tracked by a head-mounted display). These purely stereoscopic formats do not enable rendering novel views from arbitrary poses with 6DOF. This limitation can cause motion sickness for a user,

because if they move their head, their vestibular system perceives motion, while their eyes will not see a corresponding motion. Even rotating while staying in place causes enough translation to be subtly incorrect without 6DOF rendering. 6DOF is necessary to avoid motion sickness, but it is much more difficult to create, edit, compress, and render 6DOF video. The current state of the VR video industry is that the vast majority of video is not 6DOF, due to the technical challenges with creating it.

In order to accelerate adoption of immersive volumetric video, we aim to devise a format which is easy for creators to work with, and can be deployed across a wide variety of platforms and limited devices. To this end, we seek an encoding scheme which relies on only the most universally supported features of conventional 2D video codecs such as h264, h265, VP9, and ProRes. This means we assume lossy compression, no alpha channels, no more than 8 bits pers channel, no multi-view encoding, 422 chroma subsampling, and avoid the use of additional data streams (which are not trivial to decode or perfectly synchronize in all browsers). An ideal format can be edited (at least basic cuts, if not more advanced effects) in widely used video tools such as Adobe Premiere, Final Cut Pro, or DaVinci Resolve. Prior work [5] made considerable progress on similar goals to ours, but in our view its widespread adoption has been limited by the complexity of their format, prohibitive runtime for processing necessitating cloud infrastructure, and insufficient spatial resolution for VR. Therefore we also aim to estimate a volumetric representation and compress it into a streamable format in reasonable amount of time on a single workstation, and to do this with roughly "8K" resolution which is now expected in VR video (in our experience, users are extremely sensitive to resolution and may prefer higher resolution video even if it is not 6DOF; therefore we believe high resolution is key to adoption). Finally, we aim for a system which can ingest camera data from a wide variety of camera systems, ranging from monoscopic capture with phones to light field camera arrays [19].

Recently, neural radiance fields (NeRF) [16] have emerged as a powerful category of methods for reconstructing a 3D model from one or more images of a scene, and enabling photorealistic novel view synthesis. This is accomplished by using a neural net to represent a radiance field, which maps a 3D point and ray direction to color and density, then training the net by minimizing a loss between predicted pixel colors obtained by differentiable volumetric rendering, and ground truth colors from pixels in real images. Some areas of active research for NeRF include extensions to time-varying scenes, and real-time rendering. Layered depth images (LDI) are another representation for novel view synthesis [20]. LDIs consist of a set of layers, each of which has color and alpha channels, and a depth map which defines its 3D geometry. LDIs and NeRF have different strengths and weaknesses. This work illustrates some connections between the two methods.

We demonstrate a complete system for immersive volumetric video, which simultaneously addresses all of the goals stated above. Our contributions include:

- We show how to bake neural radiance fields into layered depth images with only a few layers, which enables extremely fast rendering of photorealistic novel views within a limited viewing volume (Sec. 3.3).
- We show how to encode LDIs with immersive field of view and high spatial resolution (in a foveated center region) using inflated equiangular projection, and how to encode inverse depth maps with 12-bits of accuracy in a lossy 8-bit/channel container (Sec. 3.1). A sequence of such LDIs can be encoded with universally supported features of conventional video codecs.
- To evaluate the proposed methods, we construct neural radiance fields and bake them into the proposed encoding using two datasets: one consisting of static scenes captured with a phone, and another using a light field video array with 46-cameras on a hemispherical dome [5]. Our proposed methods are several orders of magnitude faster than prior work [5]. We provide interactive demos online which work in a 2D web browser on desktop or mobile, or in VR head-mounted displays including Meta Quest Pro, Quest 3, and Apple Vision Pro, via WebXR.
- We provide open source implementations for baking a radiance field into a layered depth image with our proposed encoding, and for real-time rendering on web, and Unity and Unreal Engine (game engines).[1]

2. Background

We begin with a review of projections for immersive media, and volumetric rendering of radiance fields, building toward our proposed methods.

2.1. Equirectangular and Equiangular Projections

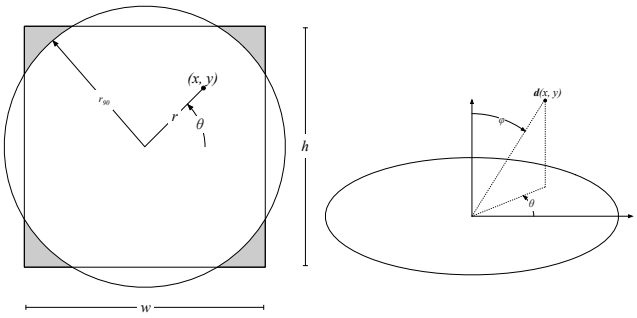
Equirectangular projection (Fig. 1a) is widely used in virtual reality video and photos to wrap a rectangular image around a sphere or half of a sphere. For example, the VR180 format, which has become increasingly popular in recent years, consists of a stereoscopic pair of images (one for each eye) in equirectangular projection, with 180° horizontal and vertical field of view. Equirectangular projection is used for both 360° and 180° content.

Equirectangular projection is arguably not the best choice for 180° content, because it has the lowest pixel density in the forward direction, and higher pixel density in the

[1] https://github.com/fbriggs/lifecast_public

(a) Equirectangular (b) Equiangular (c) Inflated equiangular

Figure 1. Equirectangular, equiangular, and inflated equiangular projections. Photo by Thomas Hübner.



(a) 2D image view of equiangular projection, showing an example where the image circle exceeds the image dimensions.

(b) 3D view of equiangular projection, showing the ray direction corresponding to pixel (x, y).

Figure 2. Equiangular projection defines a correspondence between a pixel coordinates (x, y) and ray directions $\boldsymbol{d}(x, y)$ via polar coordinates of the pixel (r, θ).

peripheral part of the view. This is the opposite of what we want, because we expect users to mostly look forward. An alternative is equiangular projection (Fig. 1b), which is less commonly used to deliver immersive media, and more often used to represent fisheye lenses. Equiangular projection has a desirable property of putting more pixel density in the forward direction compared with equirectangular.

Fig. 2 illustrates how equiangular projection defines a correspondence between pixel coordinates (x, y) in a 2D image with dimensions $w \times h$, and a ray direction in 3D space $\boldsymbol{d}(x, y)$. Equiangular projection can be parameterized in terms of a focal length but we find it more convenient to specify r_{90}, the radius in pixels at which the angle is $90°$ off forward (for $180°$ total FOV). The pixel coordinate (x, y) is rewritten in polar coordinates relative to the image center as (r, θ). In 3D, the corresponding ray direction is written in spherical coordinates using the same angle θ, and the angle off forward ϕ, where

$$\phi = \frac{\pi}{2} r' \quad \text{where} \quad r' = \frac{r}{r_{90}} \tag{1}$$

then the ray direction (using OpenGL coordinate conventions) is

$$\boldsymbol{d}(x, y) = \begin{bmatrix} \cos(\theta)\sin(\phi) \\ \sin(\theta)\sin(\phi) \\ -\cos(\phi) \end{bmatrix} \tag{2}$$

We modify equiangular projection to put even more pixel density in the forward direction in Sec 3.1.2.

3. Proposed Methods

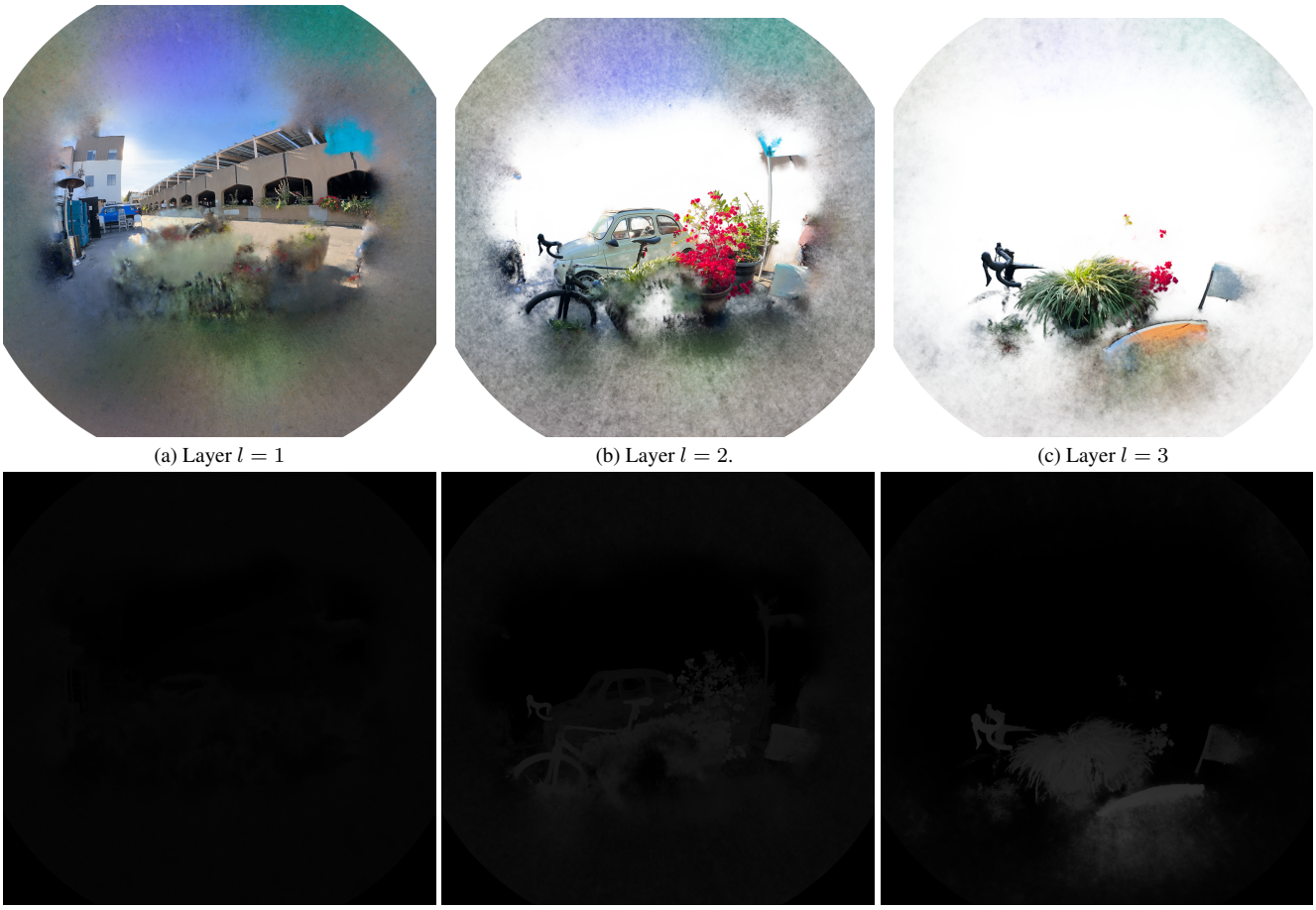
Our proposed methods consist of two main parts: several tricks for immersive LDI video, and a method for baking radiance fields into LDIs.

3.1. Immersive Layered Depth Images and Video

In this section, we describe a few tricks for representing layered depth images that are optimized for video, and for immersive fields of view.

3.1.1 Equiangular with image circle larger than image

With equiangular projection, it is typical to use a square image ($w = h$), and $r_{90} = \frac{w}{2}$, which fits the circle corresponding to $180°$ field of view exactly in the square image. We instead find it useful to use $r_{90} = S\frac{w}{2}$, where S is a scaling parameter we are free to choose. Figure 2a illustrates the case where $S > 1$, and the image circle does not fit inside the image. The horizontal and vertical FOV is less than $180°$; this is a tradeoff that produces higher pixel density at a fixed resolution. In our experiments, we use $S = 1.15$, for slightly less than $180°$ FOV (but still enough to be "immersive"), and slightly higher pixel density.



(a) Layer $l = 1$ (b) Layer $l = 2$. (c) Layer $l = 3$

(d) Layer $l = 1$ (e) Layer $l = 3$ (f) Layer $l = 2$

Figure 3. A layered depth image with 3 layers, in equiangular projection. (a-c) show the color and alpha channel $(\boldsymbol{C}^l, a^l)$, and (d-f) show the inverse depth map v^l.

3.1.2 Inflated equiangular projection

(a) Equirectangular, center crop. (b) Inflated equiangular, center crop.

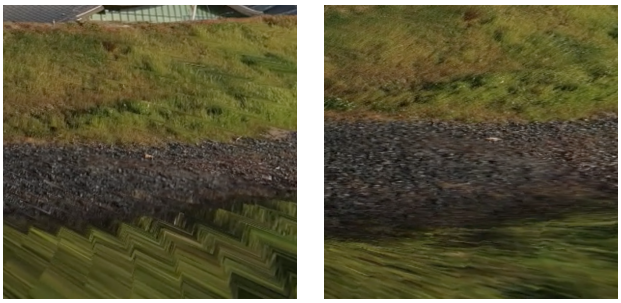
Figure 4. The center 512×512 pixels cropped from a 4400×4400 equirectangular image, compared with the center 512×512 pixels from a 1920×1920 inflated equiangular image.

Equiangular projection puts higher pixel density compared with equirectangular in the forward direction, which is good. We take this idea further by introducing "inflated equiangular" projection, which further magnifies the part of the scene in the forward direction, with a tradeoff of reducing pixel density in the peripheral directions. Equation (1) changes to

$$\phi = \frac{\pi}{2}[\beta r' + (1-\beta)r'^{\gamma}] \tag{3}$$

where $\beta \in (0, 1]$ and $\gamma > 1$ are aesthetic parameters which control the shape of inflation. We use $\beta = 0.5$ and $\gamma = 3$. Figure 1c shows an inflated equiangular image with these parameters, and $S = 1.15$. Figure 4 compares the center 512×512 pixels of an inflated equiangular image with resolution 1920×1920, with the center 512×512 pixels of an equirectangular image with resolution 4400×4400.[2] This comparison shows that the pixel densities are similar (in the

[2] In VR180 format, when one eye's image is roughly 4000×4000, this is called "8K" resolution because the left right eye are stacked horizontally. Hence, the pixel density for an inflated equiangular image at 1920×1920 is comparable to 8K VR180.



(a) 8-bit. (b) 12-bit.

Figure 5. A rendering of the same LDI, with the inverse depth map stored in either 8 or 12 bits. With 8 bits, there are noticeable staircase artifacts.

forward direction), despite the much lower resolution used with inflated equiangular projection.

3.1.3 12-bit inverse depth in an 8-bit container

We are interested in storing immersive LDIs using only the most universally supported features of standard formats such as png, jpg, and mp4, and in working within the constraints of deployment environments including the web, rendering in OpenGL, and in game engines, etc.

Depth values t naturally have values in $[0, \infty)$, which cannot be stored in universal web formats. Therefore it is useful to store depth maps using encoded inverse depth values, defined as

$$v = [K/t]_{01} \tag{4}$$

where $[\cdot]_{01}$ denotes clamping to $[0, 1]$, t is a depth in units of meters, and K is a constant that adjusts the scaling of the inverse depth map (we use $K = 0.3$). The value of v is stored as the intensity of a pixel in an image or video file, and is quantized to the bit-depth of the container. 8-bit/channel is fairly universal, whereas 10-bit (or more) video compression codecs are not supported on all browsers and devices, and even when supported, may come with significant performance tradeoffs.

However, encoding inverse depth with 8-bit accuracy produces unacceptable staircase artifacts (Fig. 5a) when rendering the resulting LDI from novel views. With a 12-bit encoding of v, the staircase artifacts are much less apparent (Fig. 5b). Therefore we consider how to store a 12-bit encoding of v in an 8-bit/channel image or video format. A few further issues complicate this endeavor: first, video compression codecs often use chroma-subsampling, so we avoid trying to pack data into different color channels of the same pixel, and instead only use luminance. Second, the data may be stored with lossy compression, and simply encoding as a texture and then decoding in OpenGL may be a lossy operation in terms of preserving individual bits. Lossy compression for color images may be acceptable, but corruption of the most significant bits of the depth map can produce major artifacts.

Our approach is to store two 8-bit values in different regions of a container image or video, which can be reassembled into a 12-bit value, using an encoding that is robust to some corruption. We store the inverse depth map at half the resolution of the color map, to make space for storing these two copies; in typical use, this doesn't come with any loss of quality when rendering the LDI in realtime, because realtime rendering is done via a triangle mesh that is lower resolution than even the half-resolution depth map (due to limitations of the rendering devices). A higher resolution depth map just gets aliased anyway.

```
void encode12(
  const float v,
  uint8_t& low,
  uint8_t& high)
{
  int iv = v * ((1 << 12) - 1);
  high = iv >> 8;
  low = iv & ((1 << 8) - 1);
  low = (iv & (1 << 8)) == 0 ? low : 255 - low;
  high = high * 16 + 8;
}
```

Listing 1. C++ code for 12-bit encoding

```
float decode12(
  uint8_t low,
  uint8_t high)
{
  high = high / 16;
  low = (high & 1) == 0 ? low : 255 - low;
  int i12 = (low & 255) | ((high & 15) << 8);
  float v = float(i12) / float((1 << 12) - 1);
  return v;
}
```

Listing 2. C++ code for 12-bit decoding

The bit-level logic is inspired by the Pack10 algorithm [7], which produces better compression than simply splitting the bits of a 12-bit value naively, by "folding" bits to reduce high-frequency banding patterns. Listings 1 and 2 give C++ code for encoding and decoding the 12-bit inverse depth values into "high" and "low" 8-bit values.

3.1.4 The ldi3 Format

'ldi3' is a format for immersive layered depth images and video [4] which incorporates the tricks mentioned above, and is optimized for video on devices with limited GPU power, circa 2024 (e.g., Meta Quest 2, Quest 3, Quest Pro, Apple Vision Pro, mobile phones and tablets, etc). Across these VR devices, we identify a baseline set of video decoding capabilities, and optimize the ldi3 format around working within these limitations. The maximum resolution video that can be decoded is 5760×5760 at 30 or 60 frames per second, 8 bit, RGB (not RGBA). Mobile phones may have

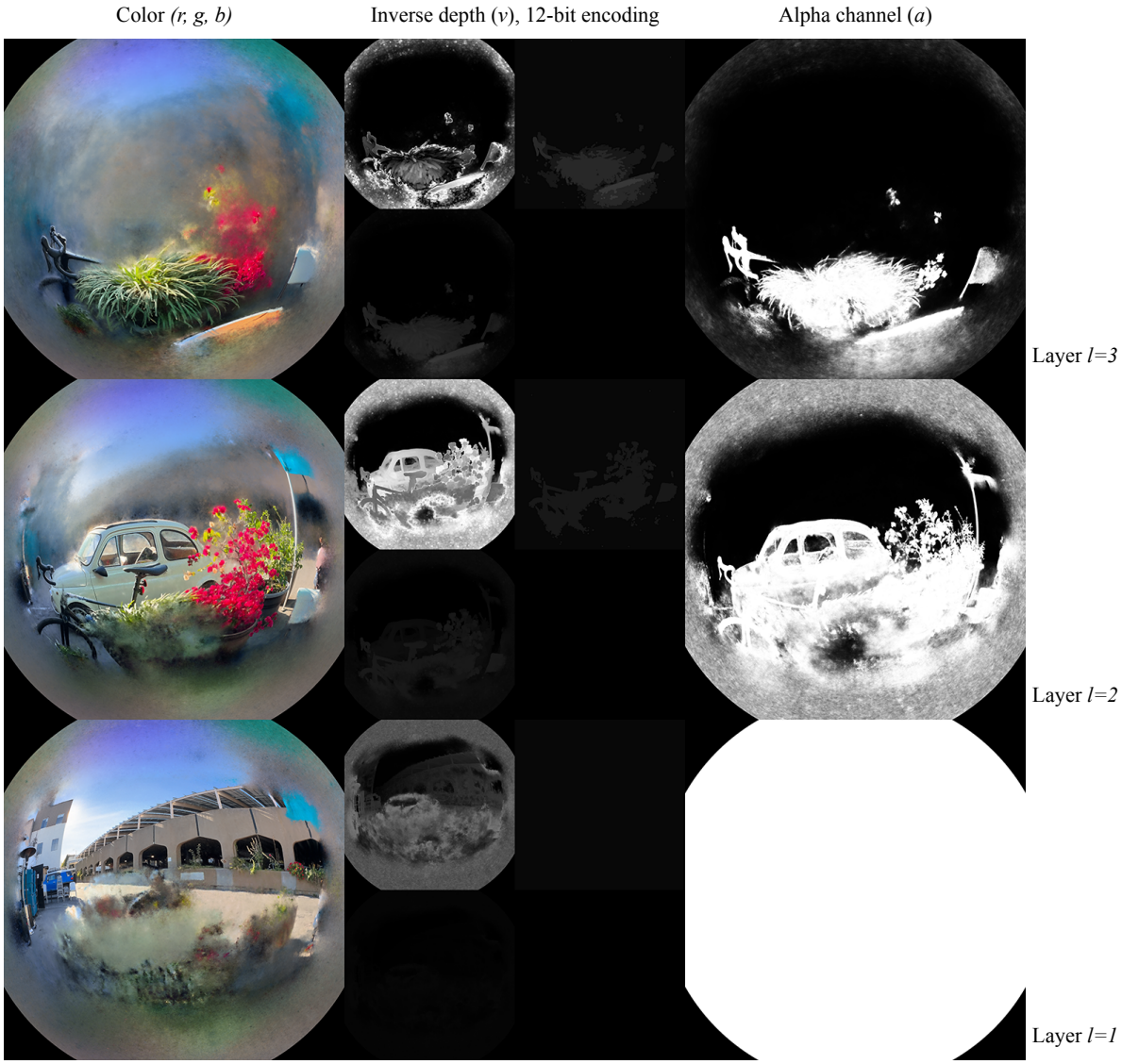


Figure 6. An example in the ldi3 format, which uses inflated equiangular projection, and 3 layers with color, alpha, and 12-bit inverse depth encoded into an 8-bit container.

a lower maximum resolution, e.g., 1920×1920, but we can construct an ldi3 frame at the full resolution, then downscale, and the 12-bit inverse depth encoding does not break due to the resizing operation. The question now is, how can we best utilize these 5760×5760 pixels?

With ldi3, we choose the number of layers $N_l = 3$ because because we can store a 3×3 grid of 1920×1920 cells in 5760×5760, and with inflated equiangular projection, the center of a 1920×1920 image has similar pixel density to "8K" VR180 in equirectangular projection. Increasing the number of layers beyond three would come with a tradeoff in spatial resolution which we think is unacceptable to most users (resolution is one of the most important qualities of immersive video that users notice). Furthermore, our preliminary experiments suggest that WebXR/OpenGL rendering of more than three layers is not fast enough on current mobile VR devices due to alpha blending operations. So it appears that on current hardware, three layers is a sweet spot. More layers could enable representing more geometrically complex scenes, but we show that with careful construction, three layers is sufficient to achieve good novel view synthesis on many real-world scenes.

Figure 6 illustrates the ldi3 format. Each row corresponds to one layer. The left column stores the RGB color for the layer. The right column stores the alpha channel. The middle column stores inverse depth maps in 12-bit en-

coded format. Each 1920×1920 region in the middle column is divided into four cells, each half the size. The upper two of these store the high and low bits in the 12-bit encoding. The lower two are unused; we store an 8-bit version of the inverse depth map in one of these cells just for human readability, and the other is left empty. These cells could be used for something else in future work, e.g., to encode data for view-dependent effects.

3.1.5 Rendering ldi3 in real time

Once we have an image or video in ldi3 format, the main use is to be able to render it from novel views in real time, on a variety of platforms such as WebXR (OpenGL), and game engines such as Unity and Unreal Engine. We provide open source implementations of players for all of these platforms.[3] In this section we give a high-level description of how to render ldi3 in a 3D graphics engine such as OpenGL, which uses vertex and fragment shaders.

The main idea is to construct a triangle mesh for half a unit sphere, store the ldi3 image in a texture (which updates each frame for video), then use a vertex shader to scale each vertex so that instead of being a unit vector, its length is the decoded value of the inverse depth map for that layer. The initial vertex positions are constructed by iterating over a grid in pixel coordinates, and applying Eqn. 2 to get the corresponding ray direction. The vertex shader must implement the bit-level decoding logic seen in Listing 2. The fragment shader is relatively simple; it just gets the RGB and alpha channels from their respective regions of the ldi3 encoding and uses that as the fragment color.

The alpha channel for the farthest layer is optional; if it is unused, the fragment shader can be optimized to not read the alpha channel from the texture, and not perform alpha blending for that layer, and this space in the ldi3 format is also unused and available for other purposes. If the alpha channel of the last layer is used, it enables other applications such as "pass through" video for mixed reality.

3.2. Volumetric Rendering of Radiance Fields

We begin by restating the rendering equations for classic NeRF [16], which we will build upon for baking an LDI. A radiance field is a function which maps a position in 3D space $\boldsymbol{x}$, and a ray direction $\boldsymbol{d}$, to a color $\boldsymbol{c} = (r, g, b)$, and a density σ, i.e.,

$$F(\boldsymbol{x}, \boldsymbol{d}) = (\boldsymbol{c}, \sigma) \tag{5}$$

In practice, F is often represented by a neural net or data structure with some neural components.

To render a pixel (x, y) in an image, we start with a camera model which associates the pixel with a ray direction $\boldsymbol{d}(x, y)$, and a ray origin $\boldsymbol{o}$, then evaluate the radiance field F at N samples along the ray, at depth t_i for $i = 1 \dots N$. At points $\boldsymbol{o} + t_i \boldsymbol{d}$, the sampled radiance values are $(\boldsymbol{c}_i, \sigma_i)$. The density values σ are related to alpha values in traditional alpha compositing; based on the distance between consecutive samples $\delta_i = t_{i+1} - t_i$, the alpha value is

$$\alpha_i = 1 - \exp(-\sigma_i \delta_i) \tag{6}$$

The color for the pixel from volumetric rendering is a linear combination of the colors sampled from the radiance field,

$$\boldsymbol{C}(x, y) = \sum_{i=1}^{N} w_i \boldsymbol{c}_i \tag{7}$$

The weights in the linear combination are

$$w_i = \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j) \tag{8}$$

$$= \alpha_i T_i \quad \text{where} \quad T_i = \exp\left(- \sum_{j=1}^{i-1} \sigma_j \delta_j \right) \tag{9}$$

T_i is referred to as transmittance. Eqn. 9 is more common in papers, while Eqn. 8 more closely resembles most code.

It is also common to render a depth map along with an image, and this is typically done by linearly combining the depths along the ray, i.e.

$$t(x, y) = \sum_{i=1}^{N} w_i t_i \tag{10}$$

It will be necessary to modify these equations for the purpose of baking an LDI.

3.3. Baking Radiance Fields into LDIs

For each pixel and layer in an LDI, we store a color, alpha channel, and inverse depth value. Layers are indexed $l = 1 \dots N_l$. At pixel (x, y), in layer l, the LDI stores:

$$LDI(l, x, y) = (\boldsymbol{C}^l, a^l, v^l) \tag{11}$$

where $\boldsymbol{C}^l = (r, g, b)$ is a color, a^l is an alpha channel, and v^l is an inverse depth.

Now we show how to construct an $LDI(l, x, y)$ from a radiance field $F(\boldsymbol{x}, \boldsymbol{d})$. For each layer $l = 1 \dots N_l$ and each pixel (x, y) in the LDI, we obtain a ray origin $\boldsymbol{o}$ and ray direction $\boldsymbol{d}(x, y)$,[4] sample the radiance field at depths t_i

[3] https://github.com/fbriggs/lifecast_public

[4] The ray origins and directions for the LDI need not correspond to any camera in the training data used to learn the radiance field. Indeed, they may not even use the same mapping from pixel coordinates to ray directions; we train on images in rectilinear projection, but the LDI uses inflated equiangular projection. This simply means there is a different equation for $\boldsymbol{d}(x, y)$ when we are considering pixels in the LDI or pixels in a training image.

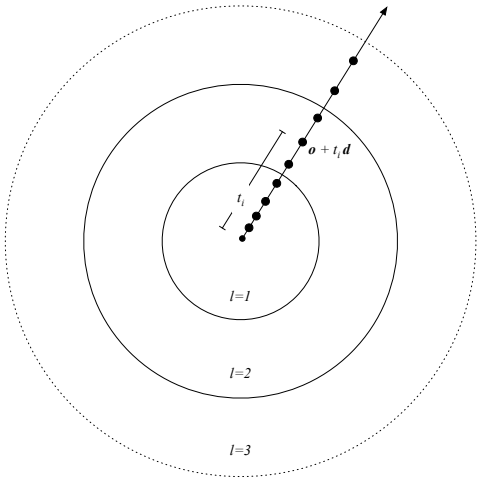


Figure 7. The radiance field is sampled at points along a ray. Each point belongs to one layer of the LDI.

along the ray to obtain $(\boldsymbol{c}_i, \sigma_i)$, and use a modified version of the volumetric rendering equations given in Section 3.2 to calculate $(\boldsymbol{C}^l, a^l, v^l)$.

Our approach to constructing an LDI from a radiance field is based on a simple idea (Fig. 7): for each layer l, define a lower and upper bound $t_{min}(l)$ and $t_{max}(l)$, and limit contributions to volumetric rendering to samples within that range of depths. This changes the α values associated with samples of the radiance field,

$$\hat{\alpha}_i^l = \begin{cases} \alpha_i & \text{if } t_{min}(l) < t_i \leq t_{max}(l) \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

where α_i is defined in Eqn. 6, and $\hat{\alpha}_i^l$ denotes the modified version used to construct layer l of the LDI.

Our goal is to calculate $(\boldsymbol{C}^l, a^l, v^l)$ for each layer in the LDI. We start with the alpha channel a^l (not to be confused with α associated with samples from the radiance field),

$$a^l = \sum_{i=1}^{N} \hat{w}_i^l \quad \text{where} \quad \hat{w}_i^l = \hat{\alpha}_i^l \prod_{j=1}^{i-1} (1 - \hat{\alpha}_j^l) \tag{13}$$

Note that a is not necessarily 1 in every layer of the LDI, because the region of 3D space corresponding to that layer may be empty (near zero σ) or partially transparent. The weights $\hat{w}_i^l$ do not sum to 1 in such cases.

Weights not summing to 1 requires subtle modifications to the equations for color and depth to avoid biases and artifacts. Similar to Eqn. 7, the color is a linear combination of colors from sampled points, but it is necessary to normalize to avoid tinting colors toward black in regions where $a < 1$.

$$\boldsymbol{C}^l = \frac{\sum_{i=1}^{N} \hat{w}_i^l \boldsymbol{c}_i}{a^l + \epsilon} \quad \text{where} \quad \epsilon = 1e^{-10} \tag{14}$$

A naive approach to construct the inverse depth v for a layer is to compute depth according to Eqn. 10, then apply Eqn. 4. The problem is that when $a < 1$, the depth estimated by Eqn. 10 is biased toward 0, which causes the silhouettes of objects, where the LDI transitions from $a = 1$ to $a = 0$, to be closer to the origin than the object. It doesn't matter what v is if $a = 0$ because that part is invisible, but when $0 < a < 1$ it breaks the LDI rendering. To fix this problem, we linearly combine inverse depths and normalize by alpha,

$$v^l = \frac{\sum_{i=1}^{N} \hat{w}_i^l [K/(t_i + \epsilon)]_{01}}{a^l + \epsilon} \tag{15}$$

When baking a radiance field into an LDI, some consideration must be given to the sampling strategy. In the original NeRF paper [16], a "fine" radiance model, and smaller, faster "course" radiance model are both trained on photometric loss. First, N_c samples are taken from the course model using a stratified sampling strategy, then the weights (Eqn. 8) are computed based on these samples, which estimates how much they contribute to the final image. The weights are normalized to form a probability density function, from which N_f further samples are drawn. This can be viewed as importance sampling from a distribution which depends only on transmittance [13]. In MipNerf360 [2], the fine model is replaced with a proposal network, which is trained via a different loss function, but the same general importance sampling framework is present. In original NeRF [16], all $N_c + N_f$ samples are combined, and evaluated in the fine network to compute the final pixel color. In later works [2] it appears to be less common to include the initial N_c samples to compute the final color, and only to use the importance samples.

This is a sensible strategy for rendering a 2D image, but requires some adjustment for baking an LDI. The issue with only using importance samples is that behind any solid object, the weights are zero, so there will be zero importance samples in that region. We need to have some samples in occluded parts of the scene, to create a proper multi-layered effect. In our experiments, we use a similar proposal network sampling strategy to [2] while training the radiance model (only using the importance samples), but then during LDI baking, we use both the initial samples to the proposal network and the importance samples. More advanced ideas are possible, but this suffices to avoid completely undersampling occlusions.

3.3.1 Locally adaptive depth bounds

Figure 7 illustrates a case where the bounds $t_{min}(l)$ and $t_{max}(l)$ are constant for all rays, resulting in each layer corresponding to a spherical shell around the origin. This approach is sufficient for some scenes, and has a desirable property of producing clean alpha channels, which could

be edited with existing video tools. However, every ray can have different bounds, and many heuristics are possible for selecting these bounds. Well chosen bounds can reduce stretched-triangle artifacts while keeping the number of layers N_l small. We use the following heuristic for 3 layers:

- Render an inverse depth map of the scene in inflated equiangular projection.
- Apply an erosion and dilation operation to the inverse depth map at multiple scales, and combine these values to obtain a heuristic score for each pixel to be foreground, middle, or background.
- For each of foreground, middle, and background, solve a least squares optimization problem to obtain an inverse depth map which matches the rendered depth map where the heuristic score is high, and is smooth otherwise.
- The bounds are half-way (in inverse depth) between the foreground and middle, or middle and background.

4. Experiments

In this section, we present the details of our NeRF implementation and training, for completeness (but we emphasize that the method for baking is independent of the NeRF implementation and could be be interchanged for any other radiance model). Then we describe two datasets used for evaluation, one from a lightfield video camera array [5], and the other consisting of static scenes captured with short videos on an iPhone 14.

We are primarily interested in the qualitative result-does this approach produce 6-degree of freedom immersive videos and photos that can be viewed on conventional VR devices (or any other device), with some freedom of motion for the user, and acceptable visual quality? To this end, we provide web demos that can be viewed in 2D or in WebXR, showing several challenging video and photo scenes. The demos are available at https://lifecast.ai.

4.1. NeRF Training

Our NeRF implementation is inspired by Nerfacto [21], Instant-NGP [17], MipNerf360 [2], Infonerf [12], FreeNeRF [23], and Nerfacc [13]. Here we specify the model computing $F(\boldsymbol{x}, \boldsymbol{d}) = (\boldsymbol{c}, \sigma)$.

Similar to Instant-NGP [17], we represent the radiance field model with a neural multi-resolution hash map and a small multi-layer perceptron. The hash map encodes an input point $\boldsymbol{x}$ to a feature $\boldsymbol{h} \in \mathbb{R}^m$. The resolution at the lowest level is 16, and it increases by a factor of 1.382 for 16 levels, with 2 features per level (hence $m = 32$). The hash map has 2^{19} entries per level. Because our focus is on unbounded scenes, we use the contraction operator from [2] to map points in 3D space to a unit cube before encoding via the NGP hash map. The hash feature goes through a 2-layer MLP with relu activation in the first layer, and no activation in the second layer to form the "geometry" feature $\boldsymbol{g} \in \mathbb{R}^{64}$.

For view-dependent effects, we encode the ray direction $\boldsymbol{d}$ using a 4th degree spherical harmonic encoder, which gives a feature $\boldsymbol{s} \in \mathbb{R}^{25}$. Similar to [15, 21], we jointly optimize a per-image latent code $\boldsymbol{q}_j \in \mathbb{R}^{16}$ where j indexes images.

The color head of the model outputs $\boldsymbol{c}$. The geometry feature, the spherical harmonic direction feature, and the per-image latent code are concatenated to form the input $[\boldsymbol{g}, \boldsymbol{s}, \boldsymbol{q}_j]$ to the color head, which is a 3-layer MLP with relu activation in the first two layers, and $\mathrm{sigmoid}$ in the last layer. All of the MLP layers use 64 hidden neurons (including the layers for the geometry feature). The density component of the model output is $\sigma = \mathrm{QuadExp}(\boldsymbol{g}_0)$ where $\boldsymbol{g}_0$ is the first element of the geometry feature.

Nerfacto [21] uses the $\mathrm{TruncExp}$ activation function for density, and observes this is better than relu because it enables high post-activation density outputs with smaller internal parameters. In our preliminary experiments, we confirmed that $\mathrm{TruncExp}$ converges faster than relu, but also found that it sometimes causes NaNs during training (which is more likely when training many NeRFs, i.e., one per frame of video). Therefore we introduce the $\mathrm{QuadExp}$ activation function, which is more numerically stable than $\mathrm{TruncExp}$ but also benefits from non-linearity to output high densities,

$$\mathrm{QuadExp}(x) = \begin{cases} \exp(x) & \text{if } x < 0 \\ (x+1)^2 & \text{otherwise} \end{cases} \tag{16}$$

Similar to [2, 13, 21] we use a smaller density proposal network, where the hash map has a course scale of 16, increasing by a factor of 2, over 5 levels, with 2 features per level, and 2^{17} entries per level. The proposal network maps an input point $\boldsymbol{x}$ to density σ, by first applying the contraction operator, then the hashmap, then a 2-layer MLP with relu followed by $\mathrm{QuadExp}$ activation and 32 hidden neurons.

Our method for training the proposal network is different from [2]. Instead of using their histogram loss function, every $k_{update} = 10$ regular batches of optimization, we sample an additional batch of the same size, and evaluate σ from both the full radiance model and the proposal model at each point of the batch's rays. Then we use a smooth L1 loss between the full and proposal σ's. Only the proposal network parameters are updated during this step, the full radiance model is fixed. The motivation for this approach is simple: the purpose of the proposal model is to estimate transmittance to enable importance sampling, and the quality of the estimation is determined by how closely the proposal den-

sities match the full model densities, therefore we just minimize this difference.

We use importance sampling during training as in [2]. Our initial $N_c = 128$ samples from the proposal network are split into two groups of $N/2$. The first half are sampled with uniform stratified sampling between $Z_{near} = 0.2$ and $Z_{mid} = 10.0$, and the second half are stratified samples from inverse depth values between $\frac{1}{Z_{mid}}$ and $\frac{1}{Z_{far}}$, where $Z_{far} = 1000$. After these initial samples, we draw a further $N_f = 64$ samples using importance sampling from the weight distribution. While baking the LDI, we increase these paremeters to $N_c = 256$ and $N_f = 128$.

Our model uses a suite of regularization and loss terms to improve generalization from sparse input views, and to impart other desirable properties in the radiance field.

- We use distortion loss $\mathcal{L}_{dist}$ from MipNerf360 [2], which pushes the weights in the linear combination of colors for each ray to be clustered near a small number of surfaces.
- We use occlusion regularization $\mathcal{L}_{occ}$ from FreeNerf [23], which penalizes density within a distance of t_{occ} units of a camera. This prevents a failure mode for training where each image is represented by a small cloud directly in front of the camera.
- We find it aesthetically pleasing for the radiance field to be empty (instead of filled with random noise) in regions of space which are not visible to any camera in the dataset. To encourage this property, we sample $N_v = 1024$ additional random points in each batch within a ball of radius 100, then check if each point is in any camera in the dataset's frustum. Let the sampled points be $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{N_v}$, and define visibility indicator variables $y_1, \ldots, y_{N_v}$. The radiance model evaluates to densities $\sigma(\boldsymbol{x})$ and colors $\bar{\boldsymbol{c}}(\boldsymbol{x})$ (averaged from RGB down to 1 channel). The loss is

$$\mathcal{L}_{vis} = \frac{1}{N} \sum_{i=1}^{n} y_i \big(\sigma(\boldsymbol{x}_i) + \bar{\boldsymbol{c}}(\boldsymbol{x}_i)\big) \tag{17}$$

- To further encourage unobserved regions to be empty, we include a small penalty on all density,

$$\mathcal{L}_{\sigma} = \frac{1}{N} \sum_{i=1}^{n} \sigma_i \tag{18}$$

- To improve generalization to novel views when training on only a few images, we include a regularization term similar to ray entropy loss in InfoNerf [12], which begins by normalizing the α_i values in each ray to form a probability density function,

$$\hat{\alpha}_i = \frac{\alpha_i}{\sum_{i=1}^{N} \alpha_i + \epsilon} \tag{19}$$

  Instead of computing the entropy of this distribution as in [12], we use the Gini index, which produces slightly better results in our preliminary experiments (we expect it is more stable due to not using $\log$):

$$\mathcal{L}_{gini} = 1 - \sum_{i=1}^{n} \hat{\alpha}_i^2 \tag{20}$$

- As usual, there is a photometric loss term which compares predicted colors from the NeRF with ground truth pixels in the training data, which we denote $\mathcal{L}_{rgb}$. We use a smooth L1 loss here instead of quadratic loss.
- After the first frame of video, we include a loss term to encourage temporal stability, $\mathcal{L}_{prev}$ which is the smooth L1 loss between σ according to the current frame's model and the previous frame's model. Using only importance samples here would not regularize parts of the scene which are behind the first surface, so we instead compute this term on another set of 32 samples according to the initial sampling strategy.

The full objective with default hyperparameters is

$$\begin{aligned} \mathcal{L} = \mathcal{L}_{rgb} + 3e^{-2}\mathcal{L}_{dist} + 1e^{-3}\mathcal{L}_{occ} + 1e^{-4}\mathcal{L}_{vis} \\ +1e^{-6}\mathcal{L}_{\sigma} + 1e^{-5}\mathcal{L}_{gini} + 1e^{-4}\mathcal{L}_{prev} \end{aligned} \tag{21}$$

We use three instances of the Adam optimizer, one for the radiance model, one for per-image latent codes, and one for the proposal model, with initial learning rates of $1e^{-2}$, $1e^{-4}$, and $1e^{-2}$ respectively, and weight decay strength $1e^{-8}$, $1e^{-4}$, and $1e^{-6}$. The optimizer for the radiance model applies weight decay only to the MLP parameters, not the hash map parameters. We train for 5000 batches with 4096 rays per batch. The learning rates are adjusted according to the following schedule: during the first 100 "warmup" iterations, the learning rate ramps linearly from $0.01\times$ the initial value up to the initial value. Then at milestones of $\frac{1}{2}$, $\frac{3}{4}$, and $\frac{9}{10}$ of the total iterations, the learning rates decay by a factor of 0.333.

4.2. Datasets

We evaluate the proposed methods on our own dataset of static scenes captured with an iPhone 14, and a few of the light field video datasets from [5].

4.2.1 Light field video array [5]

Broxton et al. [5] captured several videos with a custom camera array consisting of 46 time-synchronized Yi4k action sports cameras distributed on the surface of a 92cm diameter acrylic dome. These datasets illustrate several challenging phenomena for novel view synthesis, including volumetric effects (sparks and flames), thin structures, and reflections. We show results on a shamelessly selected subset of these videos which are most favorable to our method.

The dataset includes a pose and intrinsic parameters for each of the cameras, with some distortion parameters. As a preprocessing step, we rectify the images to remove lens distortion before using them in our training pipeline. To render a video in ldi3 format, we treat each frame of video independently, and train a NeRF on the images for that frame, then bake the frame into an ldi3 image. Finally, we encode the sequence of ldi3 images using conventional h264 or h265 compression (or ProRes if further editing is intended).

Broxton et al. [5] used only 7 of the 46 cameras because "using 46 input images during training would be prohibitively expensive, even at low resolution". The runtime for training with our method does not depend on the number of images (batches sample pixels uniformly from all images), so it is possible to use all 46 cameras, but we use only 20 because the additional cameras beyond this cover a lot of area that is outside the 180° field of view, and it wastes samples on areas of the field that are never used, resulting in slower convergence in the forward direction.

After rendering an LDI for each frame, we do some minor manual cleanup to improve temporal stability, by roughly drawing a soft mask over parts of the image that are not changing, then blending the RGBA and inverse depth values from one frame onto all of the other frames in the masked region. This process takes a few minutes per video. The need for this manual step illustrates a limitation in our current NeRF video engine (solving for NeRFs independently on each frame is not inherently temporally stable). However, the fact that it is easy to fix manually with basic image editing operations is one of the strengths of the ldi3 format.

4.2.2 Casually captured iPhone videos

We captured several static scenes using videos ranging from 4s to 20s in duration, using an iPhone 14, with 4K resolution and 30 fps. The phone camera was held at arm's length, and moved in a square or circle shape to obtain variety camera poses. The camera is on its widest field of view setting, and has auto-exposure on. Before further processing, we down-scale the videos to a maximum width or height of 2048 (some are portrait, some are landscape). We use the Lucas-Kanade method [3] to track key-points between frames of video, then solve a bundle adjustment problem using Ceres [1] to estimate the camera pose in each frame, jointly with the focal length (because we are working with videos from an iPhone, we can assume they are pre-rectified, so we are able to achieve sub-pixel reprojection error while only estimating focal length and no other intrinsic camera parameters). We use an iteratively re-weighted non-linear least squares objective to reduce the effect of outliers from bad keypoint tracking, or from moving objects in the scene such as foliage in wind, water ripples, or people. After solving for camera poses in each frame of video, we reduce the dataset to $n = 30$ images chosen to have maximal diversity in position using a farthest first traversal (this prevents overfitting when the camera dwells in one place longer than others during a video).

5. Comparison to prior work

Broxton et al. [5] presented one of the first complete systems aimed at capturing, processing, and streaming light field video in a practical manner. Their method first estimates a multi-sphere image (MSI) representation of the scene (MSI is a generalization of multi-plane images to spherical shells). They use 160 layers for the MSI, then condense these into 16 "layer groups", each of which is represented by RGBA and depth in equiangular projection (similar to ours, but not inflated). From the layered depth image representation, they encode a texture-atlas and layered mesh representation, with mesh vertex data stored in a separate data stream. While [5] made significant progress toward a practical light field video system, it has some issues which limit widespread adoption:

- Processing time of 28.5 CPU hours per frame of video is prohibitive and necessitated cloud processing.
- The layered mesh representation includes a separate data stream which must be decoded and synchronized with the h264 or h265 videos stream, which is not trivial on all platforms, and makes it more complicated to decode the format across a wide variety of devices.[5] The additional data stream is also not compatible by default with standard video editing software. The result is that it is not straight forward to do basic editing operations like cutting several clips together with industry standard tools such as Adobe Premiere.

We compare runtimes with [5], but it is important to note several caveats. With an Nvidia RTX 4090 GPU, our proposed methods takes about 347 seconds per frame to train the NeRF, and about 83 seconds to bake into ldi3 format (at an output resolution of 5760×5760). This is a roughly 238x faster than [5], but we are running on a GPU instead of CPU. Furthermore, we are not quantitively comparing the visual fidelity of the two methods in this document. With our method, results can be improved at the expense of increasing runtime by increasing the number of samples per ray to $N_c = 256$ and $N_f = 128$, the number of batches to 10000, and the number of rays per batch to 8192. We are actively improving the code, and further optimizations or other changes which affect runtime may be in the open source release after this report is published. Setting these

[5] For example, some browsers intentionally prohibit finding out which frame a video is on for security reasons, which makes it difficult to synchronize video textures with a separate stream of data for vertex geometry.

details aside, the overall result is that vast cloud resources are no longer required, and this is now possible to render lightfield video on a single workstation.

The ldi3 format does not require an additional data stream for vertex geometry because it uses a 12-bit depth map encoding, so it can be directly edited in any video tool, and it is simple to decode on myriad platforms. Instead of constructing an MSI and baking it to a texture-atlas and layered mesh, we construct a NeRF, then bake it into an LDI. In recent years, NeRF has seen an explosion in popularity due to its excellent results for photorealistic novel view synthesis; a strength of our method is that it can be applied with any new method of parameterizing and estimating a radiance field. It can also be applied to a 4D radiance field constructed from sparse input views.

Alternative approaches to "baking" neural radiance fields into a format for real-time rendering exist, e.g., sparse-neural radiance grids (SNeRG) [10], but this and other approaches are not applicable to dynamic scenes or 4D radiance fields. In contrast, our proposed baking method can represent static scenes using typical image formats such as jpg or png, and videos using conventional h264 and h265 compression and no extra data streams.

3D Gaussian splatting [11] is a fast algorithm for 3D scene representation and novel view synthesis. Several NeRF algorithms are comparably fast to train and render in real time [8, 22]. Our implementation trains in minutes, and once baked into an ldi3 it renders in realtime on limited devices (e.g., an iPhone 7). Prior work has extended NeRF and Gaussian splatting to 4D / dynamic scenes [6, 9, 14], but these approaches do not encode the video in a standard format, and therefore do not benefit from dedicated video decompression hardware or enable editing with existing tools.

6. Limitations & Future Work

Here we discuss some limitations of the proposed methods, and ideas for improvement.

- If NeRF training does not produce a good result, the resulting LDI will also not be good. For example, this happens on the "Car" dataset from [5]. We suspect the high-frequency repetitive pattern of the road causes problems for the NeRF 3D reconstruction. This could be mitigated by introducing further regularization terms, e.g., the patch depth smoothness of RegNerf [18].

- Our approach to estimating a time-varying NeRF is fairly simple (just estimate a 3D NeRF for each frame). A 4D formulation could potentially improve temporal stability, processing time, and/or generalization from sparse inputs.

- ldi3 is optimized for compatibility with current devices, but with only 3 layers, our heuristic still sometimes produces stretched triangle artifacts. Increasing the number of layers makes it easier to avoid such artifacts, but is primarily limited by video decoding capabilities. Multi-view codecs are likely to be useful in encoding the RGBA and inverse depth maps for more layers.

- We use inflated equiangular projection for near 180° field of view, but other applications call for a full 360° field of view. It is straightforward to apply the same methods for baking LDIs in any other projection (e.g., equirectangular or cube map for 360° content), with any number of layers, etc. The only issue is that all of the RGBA and inverse depth maps must be arranged within the available space of the container format.

- The proposed methods generally produce photorealistic novel views close to the origin, but rendering artifacts increase farther away (e.g., stretched triangle artifacts).

- The proposed methods to not explicitly model view-direction dependent effects. In some cases, effects like reflections and specular high lights are represented by multiple transparent layers, but the spherical harmonic data in the NeRF is collapsed to a single color. There is unused space in the ldi3 format which could be used to store additional data for view-dependent effects in future work.

7. Conclusion

We present the one of the first practical systems for immersive light field (a.k.a. 6DOF, or volumetric) video with photorealistic novel view synthesis (within a limited viewing volume), high spatial resolution, accessible compute requirements, real-time rendering on a wide variety of platforms, and a compression format that is compatible with existing video tools. Prior work has solved some, but not all of these issues simultaneously. We do this by showing how to bake neural radiance fields into immersive layered depth images.

References

[1] Sameer Agarwal, Keir Mierle, et al. Ceres solver: Tutorial & reference. *Google Inc*, 2(72):8, 2012. 11

[2] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5470–5479, 2022. 8, 9, 10

[3] Jean-Yves Bouguet et al. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel corporation*, 5(1-10):4, 2001. 11

[4] Forrest Briggs. Practical immersive volumetric video for vr and virtual production, with layered depth images and stable diffusion, 2023. URL: https://medium.com/@fbriggs/practical-immersive-volumetric-video-for-vr-and-virtual-production-with-layered-depth-images-and-b842f0d346f9. 5

[5] Michael Broxton, John Flynn, Ryan Overbeck, Daniel Erickson, Peter Hedman, Matthew Duvall, Jason Dourgarian, Jay Busch, Matt Whalen, and Paul Debevec. Immersive light field video with a layered mesh representation. *ACM Transactions on Graphics (TOG)*, 39(4):86–1, 2020. 1, 2, 9, 10, 11, 12

[6] Ang Cao and Justin Johnson. Hexplane: A fast representation for dynamic scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 130–141, 2023. 12

[7] Catid. Pack10. URL: https://catid.io/posts/pack10/. Accessed: March 8, 2024. 5

[8] Daniel Duckworth, Peter Hedman, Christian Reiser, Peter Zhizhin, Jean-François Thibert, Mario Lučić, Richard Szeliski, and Jonathan T Barron. Smerf: Streamable memory efficient radiance fields for real-time large-scene exploration. *arXiv preprint arXiv:2312.07541*, 2023. 12

[9] Sara Fridovich-Keil, Giacomo Meanti, Frederik Rahbæk Warburg, Benjamin Recht, and Angjoo Kanazawa. K-planes: Explicit radiance fields in space, time, and appearance. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12479–12488, 2023. 12

[10] Peter Hedman, Pratul P Srinivasan, Ben Mildenhall, Jonathan T Barron, and Paul Debevec. Baking neural radiance fields for real-time view synthesis. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5875–5884, 2021. 12

[11] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4):1–14, 2023. 12

[12] Mijeong Kim, Seonguk Seo, and Bohyung Han. Infonerf: Ray entropy minimization for few-shot neural volume rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12912–12921, 2022. 9, 10

[13] Ruilong Li, Hang Gao, Matthew Tancik, and Angjoo Kanazawa. Nerfacc: Efficient sampling accelerates nerfs. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 18537–18546, 2023. 8, 9

[14] Tianye Li, Mira Slavcheva, Michael Zollhoefer, Simon Green, Christoph Lassner, Changil Kim, Tanner Schmidt, Steven Lovegrove, Michael Goesele, Richard Newcombe, et al. Neural 3d video synthesis from multi-view video. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5521–5531, 2022. 12

[15] Ricardo Martin-Brualla, Noha Radwan, Mehdi SM Sajjadi, Jonathan T Barron, Alexey Dosovitskiy, and Daniel Duckworth. Nerf in the wild: Neural radiance fields for unconstrained photo collections. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7210–7219, 2021. 9

[16] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021. 2, 7, 8

[17] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics (ToG)*, 41(4):1–15, 2022. 9

[18] Michael Niemeyer, Jonathan T Barron, Ben Mildenhall, Mehdi SM Sajjadi, Andreas Geiger, and Noha Radwan. Regnerf: Regularizing neural radiance fields for view synthesis from sparse inputs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5480–5490, 2022. 12

[19] Albert Parra Pozo, Michael Toksvig, Terry Filiba Schrager, Joyce Hsu, Uday Mathur, Alexander Sorkine-Hornung, Rick Szeliski, and Brian Cabral. An integrated 6dof video camera and system design. *ACM Transactions on Graphics (TOG)*, 38(6):1–16, 2019. 2

[20] Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. Layered depth images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, 1998. 2

[21] Matthew Tancik, Ethan Weber, Evonne Ng, Ruilong Li, Brent Yi, Terrance Wang, Alexander Kristoffersen, Jake Austin, Kamyar Salahi, Abhik Ahuja, et al. Nerfstudio: A modular framework for neural radiance field development. In *ACM SIGGRAPH 2023 Conference Proceedings*, pages 1–12, 2023. 9

[22] Zian Wang, Tianchang Shen, Merlin Nimier-David, Nicholas Sharp, Jun Gao, Alexander Keller, Sanja Fidler, Thomas Müller, and Zan Gojcic. Adaptive shells for efficient neural radiance field rendering. *arXiv preprint arXiv:2311.10091*, 2023. 12

[23] Jiawei Yang, Marco Pavone, and Yue Wang. Freenerf: Improving few-shot neural rendering with free frequency regularization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8254–8263, 2023. 9, 10