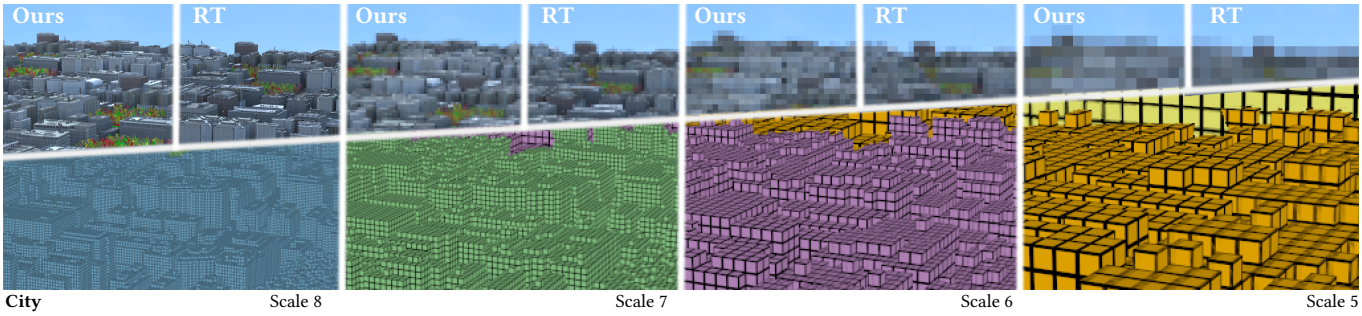
Deep Appearance Prefiltering

STEVE BAKO, University of California, Santa Barbara, USA
PRADEEP SEN, University of California, Santa Barbara, USA
ANTON KAPLANYAN, Facebook Reality Labs, USA



City	Scale 8	Scale 7	Scale 6	Scale 5
Per-pixel memory decrease:	28.1×	44.6×	155×	528×

Fig. 1. We demonstrate the first neural framework for prefiltering the appearance of complex scenes without any access to the original geometry, materials, or textures, whereas a ray tracer (RT) would require the memory cost of the full scene. This allows for a significantly reduced memory footprint across pixels relative to the reference inputs to the ray tracer, as shown here for the **City** scene, which uses the Disney BRDF and consists of over a million voxels in scale 8, the finest resolution. Our method is compared to ray tracing on the top, while the bottom shows the selected voxels colored with the scale being accessed.

Physically based rendering of complex scenes can be prohibitively costly with a potentially unbounded and uneven distribution of complexity across the rendered image. The goal of an ideal level of detail (LoD) method is to make rendering costs *independent* of the 3D scene complexity, while preserving the appearance of the scene. However, current prefiltering LoD methods are limited in the appearances they can support due to their reliance of approximate models and other heuristics. We propose the first comprehensive multi-scale LoD framework for prefiltering 3D environments with complex geometry and materials (e.g., the Disney BRDF), while maintaining the appearance with respect to the ray-traced reference. Using a multi-scale hierarchy of the scene, we perform a data-driven prefiltering step to obtain an appearance phase function and directional coverage mask at each scale. At the heart of our approach is a novel neural representation that encodes this information into a compact latent form that is easy to decode inside a physically based renderer. Once a scene is baked out, our method requires no original geometry, materials, or textures at render time. We demonstrate that our approach compares favorably to state-of-the-art prefiltering methods and achieves considerable savings in memory for complex scenes.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; **Visibility**; **Reflectance modeling**; **Antialiasing**; **Volumetric models**; **Learning latent representations**.

Authors' addresses: Steve Bako, stevebako@ucsb.edu, University of California, Santa Barbara, Santa Barbara, USA; Pradeep Sen, psen@ucsb.edu, University of California, Santa Barbara, Santa Barbara, USA; Anton Kaplanyan, kaplanyan@gmail.com, Facebook Reality Labs, Redmond, USA.

0730-0301/2022/1-ART1
https://doi.org/10.1145/3570327

Additional Key Words and Phrases: level of detail, appearance prefiltering, machine learning, volume rendering, transmittance, beam tracing

ACM Reference Format:
Steve Bako, Pradeep Sen, and Anton Kaplanyan. 2022. Deep Appearance Prefiltering. *ACM Trans. Graph.* 1, 1, Article 1 (January 2022), 22 pages. https://doi.org/10.1145/3570327

1 INTRODUCTION

Photorealistic rendering of complex synthetic and captured 3D environments can have an unbounded rendering cost for scenarios such as those in the augmented and virtual reality (AR/VR), gaming, and film industries. The geometry of these scenes can be prohibitively complex, requiring significant amounts of both storage and computation [Savva et al. 2019], especially on immersive mobile devices where compute is limited and missing the performance target is not an option. Furthermore, both captured real-world environments and physically based synthetic scenes typically have sophisticated appearances due to complex materials and light transport. Thus, it is essential for rendering systems to keep costs within budget and make the rendering complexity more predictable.

One strategy is to perform an offline precomputation of the light transport falling within a given pixel to utilize during subsequent renderings and obtain an estimate of the pixel color with fewer samples than before, as is done in the *prefiltering* framework for surfaces by Belcour et al. [2017]. Another option is to apply level of detail (LoD) techniques that replace complex 3D assets with an ideally indistinguishable multi-scale approximation to determine the appearance of surfaces and volumes more efficiently. Seminal work in this area includes geometric simplification [Hoppe 1996; Xia et al. 1997; Cohen et al. 1997] and texture mip-mapping [Williams 1983], among many others. Recent work takes a more holistic approach by prefiltering multiple scene parameters simultaneously, essentially

aiming at preserving the overall appearance of the scene at different scales [Heitz et al. 2015; Dong et al. 2015; Loubet and Neyret 2018].

However, current approaches have numerous limitations. Firstly, the types of materials that can be accurately prefiltered with existing models are limited. For example, effects like specularities and high-frequency glints are difficult to approximate, as these effects show up as unique shapes or spikes within a narrow solid angle. Moreover, materials with sophisticated BRDFs, such as the Disney BRDF, typically cannot be captured easily since the lobes are often combined in a non-linear way and mapping from arbitrary BRDF parameters to a volumetric representation is non-trivial. Finally, local occlusions are also difficult to fully capture in previous approaches.

Another limitation of existing prefiltering work is that at each particular scale there can be a mixture of both macroscale geometry, more amenable to techniques including edge simplification and microfacet approximations, and aggregate microgeometry, for which a volumetric representation such as microflakes is proven to be more suitable. Ideally, a prefiltering method would not have to choose between different representations. The state-of-the-art hybrid approach by Loubet and Neyret [2017] uses geometric analysis to label regions of a scene as either belonging to macrogeometry or an aggregate microgeometry in order to perform either the geometric or volumetric prefiltering, respectively. Although this provides improvements by enabling a more fine-grained heterogeneous representation, the heuristics used to decide on a label can misclassify regions that do not clearly belong to one or the other representation, which again results in a different appearance. There can be regions within a single asset that straddle both macro and microgeometry and where selecting a single representation could have errors (see Fig. 2). Thus, such approaches can neither represent the continuum of in-between cases nor smoothly transition between them, which is important for the practical rendering of complex scenes.

To overcome the aforementioned issues and to robustly handle a wide range of appearances, we propose a multi-scale hierarchy of local neural representations that can preserve the appearance of scenes containing both complex geometry and materials. We use a standard, multi-scale sparse voxel octree [Laine and Karras 2010], or SVO, as the data structure for our representation. This allows us to render without the original geometry or materials once an initial offline data generation step is completed, during which a ray tracer captures the light transport within each voxel. Our network then learns to represent the rendered appearance within a voxel as a lightweight latent representation, which can be efficiently evaluated within a standard physically based renderer.

To summarize, the precomputation stage of our framework voxelizes a given scene, computes per-voxel reflectance data, and trains a single network to compress this information across all voxels of the specific scene. We note that our network framework is trained per scene on data that takes days to precompute and does not generalize to new scenes (i.e., new scenes require performing data generation and training again). Afterwards, at runtime, a beam tracer is used to find all voxels falling within a pixel's footprint and the corresponding compressed information is efficiently decoded by our network to accurately determine the voxel's contribution and, ultimately, the color of the pixel using only a fraction of the original memory.

Our current system serves as a proof-of-concept that we hope inspires a new, compelling research direction: machine learning for prefiltering. Although this is a promising area to explore, there are numerous challenges to overcome, and, thus, in this initial prototype, we seek to address some of the most important aspects of the problem and leave other issues as limitations to be addressed in future work. First, we focus on rendering the directly visible appearance, which is often the most sensitive to artifacts and, therefore, quite challenging. However, in Sec. 3, we present the framework in the context of general light transport to show how the framework can be extended to arbitrary bounces, not just the first bounce shown in the results. In addition, as is common in prefiltering and LoD methods, we assume that the prefiltered portion of the scene is mostly static and can be sufficiently reused across multiple renderings for users to reap the full benefits of our approach.

Thus, our results have limitations include handling only the primary bounce on scenes using simple homogeneous materials (i.e., all objects use the Disney BRDF, albeit with different artist-set parameters). Although our theoretical framework supports global illumination, multiple bounces are omitted from our implementation and results due to the extra computational overhead required during rendering. Our method captures local shadows inherently during voxel data generation, while global occlusions are handled through shadow maps that we generate. Currently, our unoptimized implementation is bottlenecked by network inference, which can be alleviated by GPU parallelization. In its current form, our approach is only faster than ray tracing at coarser LoD scales.

In summary, our work makes the following contributions:

- We introduce the first neural method for appearance prefiltering of large complex scenes, requiring only data structure traversals and network inference to generate an image without using the original geometry or materials at render time.
- Our data-driven representation can preserve hard-to-render appearances resulting from sophisticated materials, complex geometry, local occlusions, and high-frequency, view-dependent effects such as specularities by capturing the phase function more accurately than current approximate models.
- We apply a learning-based compression of rendered data to enable its efficient utilization by a physically based renderer.
- We preserve the full appearance in a single unified representation rather than using either geometric or volumetric simplification techniques that cannot robustly handle all cases.

2 PREVIOUS WORK

Relevant to our approach are the various appearance models and the sophisticated effects they capture, the prefiltered approximations to these models, and recent applications of machine learning.

2.1 Appearance models

Background. For brevity, we only highlight appearance models most relevant to our discussion. For more in-depth information, we refer readers to texts on the subject [Pharr et al. 2016; Akenine-Moller et al. 2018] and a survey on volumetric rendering [Novák et al. 2018].

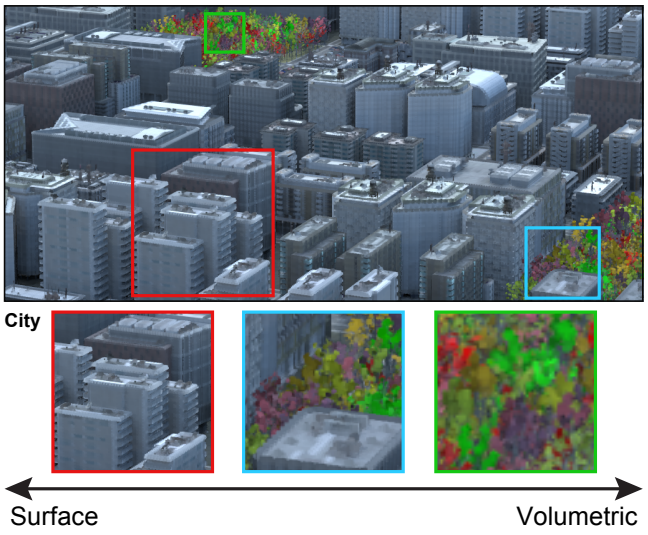


Fig. 2. The regions of a scene often exist along a continuum with macrosurfaces corresponding to watertight meshes on one end and an aggregate microgeometry on the other. For prefiltering level of detail, geometric simplification is applied to large surfaces, while volumetric appearance models, such as microflakes, are used for the latter. Although most methods typically have to decide between one or the other, recent hybrid approaches can use a combination of the two for different portions of the same scene. Still, these methods are susceptible to misclassification of regions and are bounded by the limitations of the approximate models. Our approach avoids heuristics and accurately represents the appearance of the scene, effectively handling all scenarios along the spectrum implicitly.

Microfacet materials. Microfacet BRDF models can be used with explicit geometry to render compelling effects, such as glossy reflections and glints. Yan et al. [2014] propose to use high resolution normal maps to render effects such as glints on highly specular materials, and later improve the efficiency by approximating details with Gaussians [Yan et al. 2016]. Jakob et al. [2014] demonstrate procedurally generated glints that are efficient, but do not scale to larger structures. Zirr and Kaplanyan [2016] procedurally generate materials to enable real-time rendering of glints and brushed materials at multiple scales, while Raymond et al. [2016] use a spatially varying BRDF model to render scratched metals at multiple scales.

Recently, there have been improvements in the microfacet-based BRDF for rendering highly specular materials especially at grazing angles [Chermain et al. 2019a] and also a patch-based extension to better handle glint rendering [Chermain et al. 2019b]. Since our framework bakes out the appearance from a standard ray tracer, it can be used in combination with these sophisticated BRDFs and even multi-lobe models such as the Disney BRDF [Burley and Studios 2012], which we demonstrate. Furthermore, another important advantage is that our neural framework does not require access to the original scene, allowing us considerable savings in memory.

Microflake theory. Jakob et al. [2010] derive the anisotropic radiative transfer equation to allow for physically based rendering of anisotropic participating media. This enables objects to be rendered as scattered particles in a volume rather than with a microfaceted complex geometry, which is more expensive to compute. The model is widely used for items including woven fabric, since it accurately preserves such appearances at fine scales [Zhao et al. 2011, 2012]. However, some appearances are difficult to model such as glints and self-occlusions that are naturally accounted for in our approach.

Correlated transmittance in volumetric light transport. Properly accounting for transmittance correlation in anisotropic volumes induced by surfaces is a non-trivial and open research topic. Bitterli et al. [2018] introduce new transmittance models to account for correlated scattering in non-exponential media. Other work by Jarabo et al. [2018] extend the Generalized Boltzmann Equation to use the Radiative Transfer Equation (RTE) and account for spatially correlated media. Guo et al. [2019] propose a new transmittance model to account for correlation in RTE frameworks by using fractional Gaussian fields. Our approach is orthogonal to these works and, in theory, could utilize these more sophisticated models. However, we decided to use the advantage of a data-driven representation and store a simple, spatial coverage mask to account for correlation, similar to Heitz et al. [2012].

2.2 Prefiltering and level of detail

Background. Until recently, level of detail approaches primarily consisted of either geometric simplification in the case of macrosurfaces or the mapping to a volumetric model for microgeometry. Mixtures of the two have been proposed by recent hybrid approaches. We discuss each in turn, but focus on the most relevant approaches and refer interested readers to other sources for more information [Luebke 2001; Luebke et al. 2003; Bruneton and Neyret 2011].

Geometric simplification. Seminal work in this area [Hoppe 1996; Xia et al. 1997; Cohen et al. 1997, 1998; Cohen 1999; Luebke and Erikson 2006; Yoon et al. 2006] take a mesh as input and reduce their complexity by merging vertices or deleting edges. For assets with large surfaces without a significant amount of sub-resolution detail, these approaches can perform quite well and capture the appearance faithfully. However, it is becoming commonplace that for highly detailed models with complex materials applying a mesh-based simplification will not capture these microscale structures or interesting effects such as glints or sharp specularities.

Cook et al. [2007] use random pruning to simplify scenes with aggregate detail, which are many small, similar objects next to each other that form a complex whole (e.g., sand or foliage), but assume geometric properties are uniformly distributed, an assumption that is frequently violated. Prefiltering occlusions has also been examined [Lacewell et al. 2008], but this approach involves some prior knowledge about the scene and which discrete scales are viable candidates for their occlusion model. Simplification of textured meshes [Williams et al. 2003] has also been explored.

There has also been work in hierarchical voxel representations to approximate geometry [Crassin et al. 2009, 2011; Laine and Karras 2010; Heitz and Neyret 2012; Palmer et al. 2014]. These approaches use a sparse voxel octree (SVO) data structure for prefiltering assets in an appearance-preserving way for LoD by storing surface attributes within each voxel. We use the SVO for storing our neural representations by replacing the simplistic models with our phase functions, base colors, and coverage masks that are precomputed with a ray tracer and compressed/rendered by a neural network.

Moreover, we can simplify scenes that have complex materials and microdetail which are difficult to approximate in such frameworks.

Material reflectance and aggregate detail prefiltering. There has been work on prefiltering in surface space to preserve the appearance of complex materials and assets. Dong et al. [2015] use an ellipsoid normal distribution function as a fast approximation in a microfacet reflectance model for rendering metals. Kaplanyan et al. [2016] filter the normal distributions on curved surfaces to preserve specular highlights during real-time rendering. Other work explores representing the normal distributions of multi-lobe BRDFs with a mix of Gaussian, cosine, or vMF lobes [Tan et al. 2005, 2008; Xu et al. 2017].

Also worth noting is the work by Heitz et al. [2015] that introduces the SGGX microflake distribution to efficiently model anisotropic microflake participating media and prefilter the distribution of visible normals. The recent self-shadow microflake model further builds on SGGX by taking into account occlusions inside a volume for the application of downsampling [Loubet and Neyret 2018; Loubet 2018]. Such microflake models are often applied for prefiltering aggregate detail, but with a focus on a specific subset of cases such as foliage [Max et al. 1997; Loubet and Neyret 2017], fabric [Schroder et al. 2011; Zhao et al. 2011, 2013, 2016], hair [Moon et al. 2008], or granular materials [Meng et al. 2015; Müller et al. 2016].

A common issue with these models is that, although they are fast, they have a relatively simplistic underlying reflectance or phase function that cannot faithfully capture effects such as specularities or glossy reflections. Meanwhile, our approach better preserves the appearance by more accurately representing the true reflectance obtained through standard ray tracing.

Hybrid techniques. Far voxels [Gobbetti and Marton 2005] is among the first hybrid approaches and it combines different representations based on scale, often relying on geometry for finer scales and volumetric representations at coarse scales. The recent approach by Loubet and Neyret [2017] is the first fully hybrid system that can do heterogeneous simplification of an asset at a given scale by performing analysis on a surface mesh to label portions as candidates for either geometric simplification or a voxelized volumetric model.

Our framework avoids the difficult problem of trying to unify the two representations [Dupuy and Heitz 2016] or selecting between them (i.e., either a simplified geometry or a volumetric voxel) and, as a result, we can handle the continuum of in-between cases, as shown in Fig. 2. Noma [1995] also sought to bridge this gap to preserve the appearance of collections of surfaces using volume textures to enable rendering such assets both closeup and far away on multi-resolution displays. Surfaces containing rough detail or holes are difficult to accurately classify with heuristics and cannot be properly represented in an appearance-preserving way by exclusively relying on either prefiltering model. We use only a single representation that bakes out the appearance and bypasses such issues.

Visibility. Accounting for visibility is important for determining occlusions and shadows in level of detail applications. For example, Meyer et al. [2001] utilize a visibility cube map corresponding to each level detail to enable shadowing on trees. Coverage maps [Lokovic and Veach 2000] have been used to track visibility for anti-aliasing [Crassin et al. 2018]. We similarly employ coverage maps to handle the transmittance for the multitude of scenarios that are potentially encountered when rendering a prefiltered asset.

Recent approaches. Some recent approaches target prefiltering for certain materials and effects. For example, Wu et al. [2019] jointly prefilters displacement-mapped surfaces and their BRDFs to preserve their appearance along with shadows and interreflections. Meanwhile, Gamboa et al. [2018] capture high frequency lighting effects for global illumination by using an appearance model that can tractably account for micronormal variation.

2.3 Machine learning

Background. Machine learning using neural networks has exploded in popularity since its demonstration as a practical solution for image-based classification [Krizhevsky et al. 2012]. Since then, they have achieved state-of-the-art results in countless applications within the fields of computer vision and graphics, among others. See recent texts [Goodfellow et al. 2016] for a more detailed background.

For graphics applications. Learning-based approaches have been successfully applied to Monte Carlo (MC) denoising [Kalantari et al. 2015; Bako et al. 2017; Chaitanya et al. 2017; Vogels et al. 2018]. We too apply this concept to denoise our phase functions which are undersampled to save on computation (see Fig. 11). Machine learning approaches are used to capture global illumination for relighting [Ren et al. 2013, 2015], evaluate complex luminaires [Zhu et al. 2021], represent sky models [Satỳlmỳs et al. 2017], render clouds [Kallweit et al. 2017], handle multiple scattering in participating media [Ge et al. 2018] and subsurface scattering [Vicini et al. 2019], synthesize materials [Zsolnai-Fehér et al. 2018], generate glints [Kuznetsov et al. 2019], and to perform the rendering itself [Granskog et al. 2020, 2021]. We also utilize networks to render similar anisotropic view-based effects such as glints, but without relying on approximate appearance models. Takikawa et al. [2021] focus on LoD for the reconstruction of implicit geometry represented through signed distance functions. Kuznetsov et al. [2021] leverage neural networks that are functions of incoming and outgoing directions, but they focus on learning complex, precomputed material responses that can then be mapped onto meshes. On the other hand, our approach learns the entire phase function across voxelized representations of the scene capturing the full appearance including materials, geometry, and occlusions.

Finally, neural networks have been applied for lightfield view synthesis [Kalantari et al. 2016; Wang et al. 2017; Mildenhall et al. 2019] and dense reconstruction of sparse lightfield videos [Bemana et al. 2019]. In particular, the seminal work by Mildenhall et al. [2019] created the NeRF architecture, which poses the view synthesis application as an optimization over a volumetric scene function using sparse RGB inputs. The resulting network can be queried by a 5D coordinate over spatial location and viewing direction. The novel take on view synthesis coupled with its state-of-the-art results inspired a large research field exploring NeRF-like architectures to further improve performance and quality. For example, Liu et al. [2020] use a sparse voxel representation to discard irrelevant parts of the scene and learn simpler local properties. The Mip-NeRF system [Barron et al. 2021] renders anti-aliased conical frustums across scales rather

than individual rays. Our work shares some similarities including the multi-scale, sparse voxel representation that integrates over the pixel footprint, but ours additionally supports relighting. Recent NeRF approaches now support relighting including NeRF-Tex [Baatz et al. 2021], which captures texture information within the networks that can be applied to a mesh. NeRV [Srinivasan et al. 2021] also supports relighting but only focuses on large watertight surfaces and cannot represent small microgeometry. Furthermore, the reconstruction does not have high fidelity to the reference nor does it support LoD.

For embedding and compression. A subset of approaches, including some in graphics, use networks to encode or compress the input data into a latent representation [Hinton and Salakhutdinov 2006]. A decoder network could act on this compressed representation, commonly referred to as a latent feature vector, to reconstruct the data or a subset of the data. For example, Miandji et al. [2013] compress light fields for real-time global illumination, while Chen et al. [2018] and Kang et al. [2018] encode reflectance capture. Furthermore, encoding networks are used for appearance models for face rendering [Lombardi et al. 2018], image-based relighting from optimal sparse samples [Xu et al. 2018], and appearance maps [Maximov et al. 2019]. See a recent survey [Dong 2019] for more details.

Recently, a related approach compresses the bidirectional texture function (BTF) with a neural network [Rainer et al. 2019] utilizing an encoder/decoder architecture. This method similarly queries a latent vector to directly obtain the appearance for specific view and lighting directions. Inspired by such embedding approaches, we compress/encode our voxel data to keep storage sizes tractable and to practically decode the data during runtime for appearance prefiltering. Rainer et al. [2019] use a compression network per BTF, while in our framework we use a single network per scene, each of which has on the order of one million voxels with different properties and effects captured within each. After training, our decoder networks are subsequently utilized by a beam tracer, rather than a typical ray tracer, to efficiently evaluate the contribution of voxels falling within a pixel's footprint.

3 PREFILTERED LIGHT TRANSPORT

We now present the theoretical framework for our prefiltering approach. Since we use volumetric structures, we begin with the Radiative Transfer Equation (RTE) [Chandrasekhar 1960] in its integral form for volumetric rendering:

$$L(\mathbf{x},\omega) = \int_{\mathbf{y}}^{\mathbf{x}} \tau(\hat{\mathbf{x}},\mathbf{x}) \left[\varepsilon(\hat{\mathbf{x}}) + \sigma_s(\hat{\mathbf{x}}) \int_{4\pi} f_p(\hat{\mathbf{x}},\omega,\hat{\omega}) L(\hat{\mathbf{x}},\hat{\omega}) \,\mathrm{d}\hat{\omega} \right] \mathrm{d}\hat{\mathbf{x}} + \tau(\mathbf{y},\mathbf{x}) L(\mathbf{y},\omega), \tag{1}$$

where L is the radiance at a point in a particular direction, $\mathbf{x}$ is a position in the volume, ω is the outgoing direction, $\mathbf{y}$ is a point on the volume boundary, ε is the emission at every point, σ_s is the scattering coefficient, f_p is the phase function, and τ models the transmittance by accounting for absorption and out-scattering.

After expanding the products out, we can see three distinct terms in the equation:

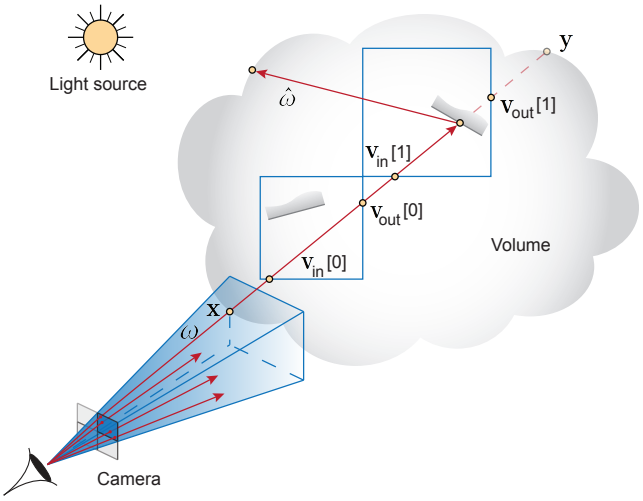


Fig. 3. Our approach prefilters the appearance of an asset by treating it as a voxelized volume through which we trace rays to integrate over a pixel's spatio-angular footprint (denoted by the blue beam). We determine the light reaching the camera sensor according to the volume rendering equation, but with precomputed terms used to calculate the amount of light propagated across each voxel's boundary, indicated by the *in* and *out* labels.

$$\begin{aligned} L(\mathbf{x},\omega) = &\int_{\mathbf{y}}^{\mathbf{x}} \tau(\hat{\mathbf{x}},\mathbf{x})\varepsilon(\hat{\mathbf{x}}) \,\mathrm{d}\hat{\mathbf{x}} \\ &+ \int_{\mathbf{y}}^{\mathbf{x}} \int_{4\pi} \tau(\hat{\mathbf{x}},\mathbf{x})\sigma_s(\hat{\mathbf{x}}) f_p(\hat{\mathbf{x}},\omega,\hat{\omega}) L(\hat{\mathbf{x}},\hat{\omega}) \,\mathrm{d}\hat{\omega} \,\mathrm{d}\hat{\mathbf{x}} \\ &+ \tau(\mathbf{y},\mathbf{x}) L(\mathbf{y},\omega). \end{aligned} \tag{2}$$

The first term is the contribution from emission, the second term is the in-scattered light, and the last term corresponds to the volume's boundary condition, which we call L_b for brevity.

If we discretize the volume into a set of voxels, V, and define the subset $V_{\mathbf{xy}} = \{\mathbf{v} \in V \mid \mathbf{x} \le \mathbf{v}_a \le \mathbf{y}\}$ where a is a point inside the voxel (i.e., the set of all voxels that lie in the interval $[\mathbf{x},\mathbf{y}]$), we can rewrite the previous equation as a sum of integrals:

$$\begin{aligned} L(\mathbf{x},\omega) = &\sum_{\mathbf{v}\in V_{\mathbf{xy}}} \int_{\mathbf{v}_i}^{\mathbf{v}_o} \tau(\hat{\mathbf{x}},\mathbf{x})\varepsilon(\hat{\mathbf{x}}) \,\mathrm{d}\hat{\mathbf{x}} \\ &+ \sum_{\mathbf{v}\in V_{\mathbf{xy}}} \int_{\mathbf{v}_i}^{\mathbf{v}_o} \int_{4\pi} \tau(\hat{\mathbf{x}},\mathbf{x}) f(\hat{\mathbf{x}},\omega,\hat{\omega}) L(\hat{\mathbf{x}},\hat{\omega}) \,\mathrm{d}\hat{\omega} \,\mathrm{d}\hat{\mathbf{x}} \\ &+ L_b(\mathbf{y},\mathbf{x},\omega), \end{aligned} \tag{3}$$

where $f(\hat{\mathbf{x}},\omega,\hat{\omega}) = \sigma_s(\hat{\mathbf{x}}) f_p(\hat{\mathbf{x}},\omega,\hat{\omega})$ is used for conciseness and $\mathbf{v}_i$ and $\mathbf{v}_o$ refer to the *in* and *out* points of voxel $\mathbf{v}$ along direction ω.

To introduce the prefiltering of light transport in path space, we follow Belcour et al. [2017] and introduce the notion of a pixel footprint. However, unlike this prior work, we formulate the expression for volumes, instead of surfaces (see Fig. 3). Thus, we can determine the final flux incident on a sensor's pixel, L_I, by integrating the radiance incident on the sensor over the pixel filter, H_I:

$$L_I = \int_{\mathcal{F}} H_I(\mathbf{x},\omega) L(\mathbf{x},\omega) \,\mathrm{d}\omega \,\mathrm{d}\mathbf{x} = E + S + B. \tag{4}$$

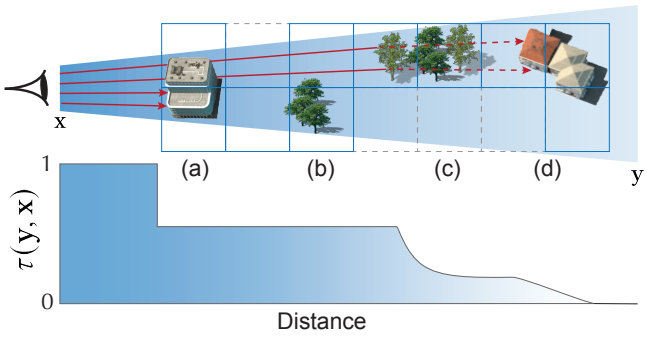


Fig. 4. Motivation for our mask-based transmittance model. The top half of the figure shows a top-down view of a beam intersecting with voxels in a scene (non-empty voxels are shown in blue, and ignored (empty) voxels are shown with dashed gray lines), while the bottom half shows the ideal tracked transmittance versus distance along the beam based on the micro and macrogeometry within each voxel. (a) First, we have a large watertight building that occludes half of the beam and the transmittance decreases to 0.5. (b) Next, the tree in this region is completely covered by the previous building, so there is no change in transmittance. (c) The voxels contain trees with many random, small leaves that in aggregate behave similar to a volume, and, thus, an exponential transmittance model is suitable for this region. (d) Finally, there are non-axis-aligned buildings. These macrosurfaces will linearly influence the transmittance. Therefore, in such a typical scenario, no single traditional transmittance model is entirely appropriate, so we instead utilize a spatial coverage mask to track the occlusions along the beam and determine the radiance contribution of each voxel.

Here $\mathcal{F}$ is the spatio-angular footprint of pixel I, as similarly defined in Belcour et al. [2017], which is introduced to integrate over image positions and visible directions within the extent of the pixel. Using the three terms of Eq. 3 in place of $L(\mathbf{x}, \omega)$ yields the filtered emission, scattering, and boundary terms (defined as E, S, and B, respectively), which we discuss in turn.

Boundary term. The propagated radiance received at the volume's boundary is simply multiplied by the pixel filter and is given by:

$$B = \int_{\mathcal{F}} H_I(\mathbf{x}, \omega) L_b(\mathbf{y}, \mathbf{x}, \omega)\, \mathrm{d}\omega\, \mathrm{d}\mathbf{x}. \tag{5}$$

Therefore, even though our framework prefilters an asset by treating it as a volume, it can be plugged into existing hybrid renderers and will properly transport incoming radiance through the volume. Thus, in practice, the integral in Eq. 5 would be evaluated with distributed ray tracing [Cook et al. 1984].

In-scattering term. We next apply our pixel filter to the in-scattering term and rearrange it to get:

$$S = \sum_{\mathbf{v} \in V_{\mathrm{xy}}} \int_{\mathcal{F}} H_I(\mathbf{x}, \omega) \int_{\mathbf{v}_i}^{\mathbf{v}_o} \int_{4\pi} \tau(\hat{\mathbf{x}}, \mathbf{x}) f(\hat{\mathbf{x}}, \omega, \hat{\omega}) L(\hat{\mathbf{x}}, \hat{\omega})\, \mathrm{d}\hat{\omega}\, \mathrm{d}\hat{\mathbf{x}}\, \mathrm{d}\omega\, \mathrm{d}\mathbf{x}. \tag{6}$$

We then apply the *far-field* assumption commonly used in prefiltering (i.e., we assume the voxel is sufficiently far away and the rays in the beam are near parallel), so the incident radiance $L(\hat{\mathbf{x}}, \hat{\omega})$ becomes constant across voxel $\mathbf{v}$ (i.e., in the range $[\mathbf{v}_o, \mathbf{v}_i]$), and we can parameterize it as $L(\mathbf{v}, \hat{\omega})$, which represents the radiance arriving at the corresponding voxel $\mathbf{v}$. After rearranging the integral, we have:

$$S = \sum_{\mathbf{v} \in V_{\mathrm{xy}}} \int_{4\pi} L(\mathbf{v}, \hat{\omega}) \int_{\mathcal{F}} H_I(\mathbf{x}, \omega) \int_{\mathbf{v}_i}^{\mathbf{v}_o} \tau(\hat{\mathbf{x}}, \mathbf{x}) f(\hat{\mathbf{x}}, \omega, \hat{\omega})\, \mathrm{d}\hat{\mathbf{x}}\, \mathrm{d}\omega\, \mathrm{d}\mathbf{x}\, \mathrm{d}\hat{\omega}. \tag{7}$$

Note that $\tau(\hat{\mathbf{x}}, \mathbf{x})$ is parameterized relative to the point in the volume $\mathbf{x}$, *not* the voxel boundary $\mathbf{v}_o$, so it could be outside the voxel $\mathbf{v}$. This poses a potential obstacle to our prefiltering application, as $\tau(\hat{\mathbf{x}}, \mathbf{x})$ cannot be reasonably precomputed, unless we can factor τ into the portion inside and outside the voxel:

$$\tau(\mathbf{y}, \mathbf{x}) = \tau(\mathbf{y}, \mathbf{v}_o)\tau(\mathbf{v}_o, \mathbf{x}). \tag{8}$$

This is a reasonable assumption which holds true for the widely-used exponential transmittance with the extinction coefficient, σ_t, given by the sum of the absorption and scattering coefficients (i.e., $\sigma_t(\mathbf{x}) = \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x})$):

$$\begin{aligned} \tau(\mathbf{y}, \mathbf{x}) &= \exp\Big(-\int_{\mathbf{y}}^{\mathbf{x}} \sigma_t(\hat{\mathbf{x}})\, \mathrm{d}\hat{\mathbf{x}}\Big) \\ &= \exp\Big(-\int_{\mathbf{y}}^{\mathbf{v}_o} \sigma_t(\hat{\mathbf{x}})\, \mathrm{d}\hat{\mathbf{x}}\Big) \exp\Big(-\int_{\mathbf{v}_o}^{\mathbf{x}} \sigma_t(\hat{\mathbf{x}})\, \mathrm{d}\hat{\mathbf{x}}\Big) = \tau(\mathbf{y}, \mathbf{v}_o)\tau(\mathbf{v}_o, \mathbf{x}). \end{aligned} \tag{9}$$

Thus, using Eq. 8, we can rewrite the scattering term as:

$$\begin{aligned} S &= \sum_{\mathbf{v} \in V_{\mathrm{xy}}} \int_{4\pi} L(\mathbf{v}, \hat{\omega}) \int_{\mathcal{F}} H_I(\mathbf{x}, \omega)\tau(\mathbf{v}_o, \mathbf{x}) \int_{\mathbf{v}_i}^{\mathbf{v}_o} \tau(\hat{\mathbf{x}}, \mathbf{v}_o) f(\hat{\mathbf{x}}, \omega, \hat{\omega})\, \mathrm{d}\hat{\mathbf{x}}\, \mathrm{d}\omega\, \mathrm{d}\mathbf{x}\, \mathrm{d}\hat{\omega} \\ &\approx \sum_{\mathbf{v} \in V_{\mathrm{xy}}} \int_{4\pi} L(\mathbf{v}, \hat{\omega})\; T(\mathbf{v}_o, \mathbf{x})\; F(\mathbf{v}, \omega, \hat{\omega})\, \mathrm{d}\hat{\omega}. \end{aligned} \tag{10}$$

The approximation stems from splitting the integral of the product as the product of the integrals to obtain our *prefiltered transmittance function*, T, and our *prefiltered phase function*,[1] F, given by:

$$T(\mathbf{v}_o, \mathbf{x}) = \int_{\mathcal{F}} H_I(\mathbf{x}, \omega)\tau(\mathbf{v}_o, \mathbf{x})\, \mathrm{d}\omega\, \mathrm{d}\mathbf{x}, \tag{11}$$

$$F(\mathbf{v}, \omega, \hat{\omega}) = \int_{\mathcal{F}} H_I(\mathbf{x}, \omega) \int_{\mathbf{v}_i}^{\mathbf{v}_o} \tau(\hat{\mathbf{x}}, \mathbf{v}_o) f(\hat{\mathbf{x}}, \omega, \hat{\omega})\, \mathrm{d}\hat{\mathbf{x}}\, \mathrm{d}\omega\, \mathrm{d}\mathbf{x}. \tag{12}$$

Since the prefiltered phase function is integrated over the in and out points of the voxel, $\mathbf{v}_i$ and $\mathbf{v}_o$, and the spatial footprint, $\mathbf{x}$, we parameterize it over voxel $\mathbf{v}$ and the incoming and outgoing directions, $\hat{\omega}$ and ω. Note, by splitting the integral product and applying our transmittance separation (Eq. 8), the prefiltered transmittance, Eq. 11, includes the transmittance from $\mathbf{v}_o$ to $\mathbf{x}$, while the prefiltered phase, Eq. 12, accounts for the transmittance through a point in the voxel, $\hat{\mathbf{x}}$, to the corresponding voxel boundary, $\mathbf{v}_o$. Although it is possible to avoid this assumption and preserve the integral of the product of transmittance and the prefiltered phase function, this adds substantial costs to the precomputation, as we would have to store the prefiltered phase function across two extra dimensions corresponding to the pixel's spatial footprint in order to evaluate

[1] This is a slight abuse of notation as it is really the phase function, f_p, multiplied by the transmittance, τ, and scattering coefficient, σ_s, across a voxel, but it can still be thought of as the 4D throughput over incoming outgoing directions.

the exact integral during runtime. Furthermore, we found this approximation worked well in practice (see Fig. 15) and use Eq. 10 in our system. At runtime, we simply sum over all the voxels in a pixel's footprint and attenuate the incoming radiance over the sphere by the scalar voxel-to-voxel tracked transmittance, $T(\mathbf{v}_o, \mathbf{x})$, and the stored prefiltered phase function, $F(\mathbf{v}, \omega, \hat{\omega})$.

Note that this scattering term is a general equation that works recursively to propagate indirect illumination back to the camera sensor, similar to Belcour et al.'s [2017] surface formulation. Although multiple bounces are theoretically supported with our prefiltering framework, we focus on demonstrating results for the first bounce, as that is typically the most susceptible to objectionable artifacts.

Emission term. The pixel filter applied to the emission term yields:

$$\begin{aligned} E &= \sum_{\mathbf{v} \in V_{\mathbf{xy}}} \int_{\mathcal{F}} H_I(\mathbf{x}, \omega) \int_{\mathbf{v}_i}^{\mathbf{v}_o} \tau(\hat{\mathbf{x}}, \mathbf{x}) \varepsilon(\hat{\mathbf{x}}) \, \mathrm{d}\hat{\mathbf{x}} \, \mathrm{d}\omega \, \mathrm{d}\mathbf{x} \\ &\approx \sum_{\mathbf{v} \in V_{\mathbf{xy}}} T(\mathbf{v}_o, \mathbf{x}) \; G(\mathbf{v}, \omega), \end{aligned} \tag{13}$$

where we apply the split in transmittance, τ, as before using Eq. 8. Similarly to our scattering term, we approximate the integral product over the pixel footprint to obtain G, our *prefiltered emission*:

$$G(\mathbf{v}, \omega) = \int_{\mathcal{F}} H_I(\mathbf{x}, \omega) \int_{\mathbf{v}_i}^{\mathbf{v}_o} \tau(\hat{\mathbf{x}}, \mathbf{v}_o) \varepsilon(\hat{\mathbf{x}}) \, \mathrm{d}\hat{\mathbf{x}} \, \mathrm{d}\omega \, \mathrm{d}\mathbf{x}. \tag{14}$$

This means that we can precompute the contribution from internal emitters and then track the voxel-to-voxel transmittance during runtime to determine the final contribution at the pixel sensor. Although our framework supports prefiltered emissions, this is less common and our scenes did not contain any internal emitters, so we omit this term from subsequent discussion.

Transmittance. Since an exponential transmittance is not suitable for watertight surfaces and neither is a simple linear one for aggregates (see Fig. 4), we utilize a more appropriate numerical transmittance model. Specifically, in the spirit of deep shadow maps [Lokovic and Veach 2000] and previous SVO-based prefiltering approaches [Heitz and Neyret 2012], we use a 2D spatial coverage mask to numerically model different transmittance modes along the beam:

$$T(\mathbf{y}, \mathbf{x}) = \int_{\mathcal{F}} H_I(\mathbf{x}, \omega) \tau(\mathbf{v}_o, \mathbf{x}) \, \mathrm{d}\omega \, \mathrm{d}\mathbf{x} \approx \frac{1}{N} \sum_{j=0}^{N-1} \Lambda_j(\mathbf{y}, \mathbf{x}), \tag{15}$$

where N is the number of pixels in our coverage mask and $\Lambda_j(\mathbf{y}, \mathbf{x})$ is the tracked coverage from $\mathbf{x}$ to $\mathbf{y}$ at the j^{th} pixel. The general tracked coverage is then given by:

$$\Lambda_j(\mathbf{y}, \mathbf{x}) = \left[1 - \sum_{\mathbf{v} \in V_{\mathbf{xy}}} \Lambda_j(\mathbf{v}_o, \mathbf{x}) \right] \alpha_j(\mathbf{v}\,[\mathbf{y}], \mathbf{x}), \tag{16}$$

where $V_{\mathbf{xy}}$ is the ordered set (closest to $\mathbf{x}$ first) of all voxels that overlap the interval $\mathbf{x}$ to $\mathbf{y}$ and $\Lambda_j(\mathbf{x}, \mathbf{x}) = 0$. Meanwhile, $\alpha_j(\mathbf{v}\,[\mathbf{y}], \mathbf{x})$ is the jth pixel of the coverage mask corresponding to the voxel at $\mathbf{y}$ in the direction towards $\mathbf{x}$, and which lies in the range from 0 (transparent) to 1 (blocked). Fortunately, we are able to precompute the coverage mask at each voxel, α, during the preprocessing step, so only the tracked coverage, Λ, needs to be computed at runtime. Essentially, we sum over all of the contributions of the voxels in order up until the edge of the current voxel containing $\mathbf{y}$. Since $\sum \Lambda_j(\mathbf{v}_o, \mathbf{x}) \in [0, 1]$, we subtract this from 1 to determine the maximum amount of contribution this voxel can have. Finally, we multiply this weight by the coverage of the current voxel, $\alpha_j(\mathbf{v}\,[\mathbf{y}], \mathbf{x})$, to determine its final contribution.

4 NEURAL PREFILTERING

4.1 Framework overview

This section discusses our implementation based on the theory from Sec. 3. We describe our general pipeline (see Fig. 5) starting with the original scene as an explicit mesh and materials to the intermediate prefiltered representation and, finally, the rendered image.

Spatial representation. We chose a sparse hierarchical voxel representation that directly corresponds to the level of detail (LoD) scales. Specifically, for a given scale, we perform uniform discretization in world space (i.e., x, y, and z coordinates) and then we double the resolution of our voxelization in each dimension for each subsequent, higher scale. In this way, LoD scale i, contains 8^i total voxels (before considering sparsity). During rendering, the filtering kernel (such as the pixel footprint and pixel filter) determines which LoD scale is used so that the relative size of a voxel does not exceed the bandlimiting frequency of the filtering kernel.[2] In other words, similarly to texture filtering, the further away the camera is from the asset, the coarser the LoD scale that is used since a pixel will map to an increased portion of the scene and larger voxels.

As in Heitz et al. [2012], we use a sparse voxel octree (SVO) representation at each scale, which discards empty voxels and allows for efficient voxel lookups. Each voxel contains both the prefiltered phase function, F, and 2D coverage mask, α, introduced in Sec. 3, and an albedo term, which are each described in turn.

Phase function. In general, a phase function is a dimensionless measure of how light is scattered at a point in a volume, typically parameterized by incoming and outgoing directions. Thus, each one of our voxels has a different phase function based on how light propagates through it. We represent the prefiltered phase function, F from Eq. 10, as a high-resolution uniform grid that maps to a 4D parameterization of input and output directions as spherical coordinates: $\omega_i = (\theta_i, \phi_i)$ and $\omega_0 = (\theta_0, \phi_0)$. Each index of this table is a single monochromatic value of the voxel's phase function corresponding to direction pairs that map to that index. This information can be queried during rendering and multiplied with a similarly queried base color, described next, to find the radiance contribution of each voxel based on view and lighting directions.

Albedo. In addition to the monochromatic phase function, we record the albedo, the average RGB base color (i.e., the average diffuse color across the geometry's projected cross-section) of the phase function in a specific outgoing direction. The 2D directional albedo is decoupled from the phase to reduce storage costs of a voxel during precomputation and also as a means of helping the network

[2]Typically, LoD approaches even use the next finer scale to so that the voxels are sub-Nyquist resolution.

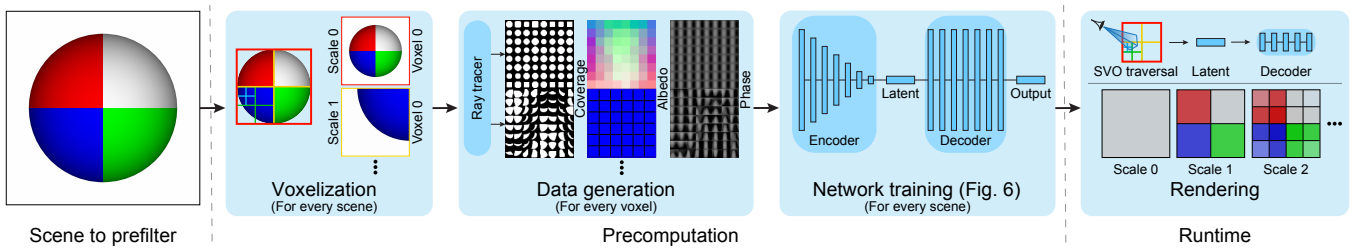


Fig. 5. Overview of our algorithm. We first voxelize the scene at multiple scales and store non-empty voxels in a sparse voxel octree (SVO). Next, in the data-generation step, we process the portion of the scene within each voxel in isolation and sample incident and outgoing directions using a ray tracer to record the phase (i.e., the throughput), coverage, and albedo information across the volume. We then train a single network to efficiently compress this per-voxel data into small latent feature vectors that can be unpacked by a lightweight decoder. During rendering, for each voxel intersected by our beam along a pixel's footprint, we evaluate the phase, albedo, and coverage by decoding the precomputed latent representations for specific incident and outgoing query directions from a light transport algorithm to generate an image. Note that during rendering, we use only SVO traversals and inference of our pretrained networks and have no reliance on the original geometry or materials of the scene.

optimization. By learning a single value per direction for the albedo, the phase sub-network does not have to learn to reproduce the same base color for a given outgoing direction at all the incident directions and instead it just learns to reproduce the more coherent monochromatic data. The albedo sub-network predicts a single RGB color per direction, which then multiplies the phase. Thus, our full phase function with RGB values is given by:

$$F(\mathbf{v}, \omega, \hat{\omega}) = \rho(\mathbf{v}, \omega, \hat{\omega})\gamma(\mathbf{v}, \omega), \tag{17}$$

where ρ and γ are the scalar monochromatic phase and RGB base albedo, respectively. Note that γ is parameterized over the outgoing direction only (2D spherical coordinates), unlike ρ which uses both outgoing and incoming directions (4D spherical coordinates). Moreover, we chose to represent only the diffuse albedo and leave explorations in additionally utilizing a specular color for future work. By using only a single scalar per direction for albedo, we do not account for different contributions of spatially varying textures, as these contributions are averaged out in each outgoing direction (see limitations in Sec. 7). Furthermore, we do not track color correlations from voxel to voxel, but we do spatially track transparency.

Coverage mask to approximate non-exponential transmittance. Our transmittance model is used to determine the contribution of a voxel based on its correlation with preceding voxels along the path to the camera. If two voxels are 100% negatively correlated, it means that neither voxel is blocked by the other and the contribution of each is simply added. If the voxels are 100% positively correlated, it means that one voxel completely blocks another one behind it, and, thus, the second voxel has no contribution. Note, there can be varying degrees of correlation that create a spectrum of cases (see Fig. 4).

We determine each voxel's coverage with a fixed-resolution 2D spatial mask, α, per particular viewing direction and we compute this for the same outgoing directions as our phase function, resulting in a 4D table. In other words, the elements of our table are indexed by a particular view, and the element itself is the voxel's 2D coverage mask. The 2D tracked coverage, Λ, is determined during runtime based on the 2D precomputed, per-voxel coverage mask, α, as shown in Eq. 16. To determine the contribution of a given voxel, we compute the proportion of its coverage mask that remains unoccluded based on the beam's tracked coverage mask so far. We found our approach was sufficient for the scenes we evaluated, but it would be interesting to utilize recent models in our framework [Vicini et al. 2021].

Rendering with a prefiltered representation. We implement a single-bounce renderer (direct lighting only) to demonstrate the directly visible quality of our prefiltered scene representation. We use a beam tracer that traverses our SVO and determines the ordered set of voxels using the intersection distance of a beam originating at each pixel. From the phase and albedo, it determines the RGB throughput in a given incoming/outgoing direction based on the scene's camera and lighting setup (Eq. 17) and evaluates the prefiltered scattering term in Eq. 10. The 2D coverage mask, α, is used to track correlations across voxels to compute the transmittance along the beam with Eq. 15, while Eq. 4 determines the final radiance accumulated along the beam to a given pixel. Note, our current scenes have no emissive surfaces, so the filtered emission term, E, is ignored.

4.2 Prefiltering and data generation

During data generation, we compute the phase function, albedo, and coverage mask for each voxel at every LoD scale. We normalize the scene by the largest dimension and then consider a unit volume bounding box around the scene. We then perform voxelization and construct our SVO before proceeding to compute each voxel's data.

We represent our phase function as a 4D table indexed by spherical coordinates corresponding to a pair of incoming/outgoing directions and where each element in a bin corresponds to a scalar monochromatic throughput. In other words, if viewing the scene from a given outgoing direction, each bin corresponds to how much light is reflected towards the camera for a given incoming direction.

To compute the value at each bin (a 4D index of incoming/outgoing directions) of a given voxel, we measure the throughput by sending many samples with a ray tracer in a specific view direction and recording each sample's eventual exit direction from the voxel's volume, as well as its accumulated radiance. Specifically, we first set up a constant environment map with a value of one everywhere (i.e., a white furnace) and sample the projected cross-section of the scene from various outgoing directions using an orthographic camera, so that all samples are intersecting the voxel with the same direction. The samples bounce around in the voxel until they ultimately exit by hitting the environment map. We use MIS sampling within a voxel and record the corresponding incident radiance and direction for the light sample at each bounce.

Once a sample terminates, we accumulate the radiance in the corresponding bins based on the view (outgoing direction) and the exit or light sample directions (incoming direction). After all the samples for this outgoing direction are recorded, we normalize each bin by the total sample count. We then calculate this for all outgoing directions based on our table's resolution. Note we sample outgoing directions uniformly along the sphere rather than uniformly in the grid to avoid bias from bins mapping to different solid angles. To ensure that the entire voxel gets properly sampled at each outgoing direction, we sample the entire projected cross section and discard any samples that do not go through the bounding box of the voxel.

At the same time we generate samples of our phase function, we also construct our 2D coverage mask, α, and directional albedo, γ. When processing each sample and binning them, we can use the cross-section sample location to find the corresponding texel in our fixed-resolution coverage mask (e.g., 16×16 in our implementation) and deposit either a 1 or a 0, based on whether the sample intersected any geometry within the voxel. We average this visibility information at each texel to generate the final supersampled 2D coverage mask corresponding to the processed outgoing direction. To compute the albedo, we average the base color from each phase function sample that intersected voxel geometry (i.e., those samples with a value of 1 for visibility). In other words, we integrate the albedo over the area of the voxel's projected cross-section along a given view direction and record a single RGB value at the 2D index.

4.3 Neural compression

Although baking out the phase function, albedo, and coverage mask into our tables should be sufficient to render an image with high fidelity to the original, there are various considerations that make this straightforward approach impractical. First, since these tables need to have a reasonably high resolution to be able to reproduce the original appearance, each voxel has a significant amount of data associated with it (i.e., on the order of MBs). Moreover, typical pipelines utilize several scales of LoDs for their assets which can amount to millions of voxels at the finer scales, even when considering sparsity (i.e., we can ignore empty voxels), totaling TBs of data for a single scene, which would undermine the savings relative to a brute-force ray tracer. Another issue is that the tables must be computed for directions at a high resolution to minimize approximation errors stemming from interpolating the non-computed views.

To be able to get a high compression rate and to more accurately interpolate non-calculated views, we propose to use a deep neural network solution. Specifically, the network takes in the full generated dataset and encodes it into a latent representation, a common strategy in the machine learning community, that is orders of magnitude smaller than the dataset size. The network includes a decoder that extracts the desired elements from the latent vector at runtime. A network-based solution allows substantial compression as is shown in previous work [Rainer et al. 2019]. Furthermore, we have the added benefit that network can perform more accurate, non-linear interpolation for views that were not precomputed.

After the network is trained, we incorporate it into our pipeline with a preprocessing step that encodes all the voxels in a scene at all the scales and generates their latent feature vectors, which are stored on disk. Prior to rendering, we load all of these vectors into memory, along with the decoder network and its trained weights. During rendering, for every intersected voxel, the beam tracer evaluates the decoder using the voxel's latent vector and a specific query direction based on the outgoing and incident directions to generate the corresponding element. Finally, we combine all the voxel data to synthesize the final image, as described in the previous sections.

We now describe the specifics of our system's neural networks.

5 NETWORK DESIGN AND UTILIZATION

Our networks first must be trained offline on the precomputed data described in Sec. 4. The networks are trained across all the voxels for a specific scene. During prefiltering, compressed representations in the form of latent vectors are precomputed for all the voxels using the converged, frozen weights of the networks. Finally, during rendering, we evaluate a subset of the networks and apply the equations from Sec. 3 to generate the final image. In this section, we describe our network architecture and optimization, followed by its use during runtime, and conclude with implementation details.

5.1 Compression networks

We use four networks in our system, as shown in Fig. 6. In general, all networks act as compressors/decompressors with a "butterfly," or U-Net style, encoder and a lightweight decoder architecture. The networks are trained per scene, *not* per voxel. In other words, these four networks are trained to handle the encoding/decoding of the millions of voxels across all of the scales of a given scene. We now describe their specific uses and subtle differences.

Encoders. The goal of the encoders is to generate a compact latent representation of the various prefiltered input data to be able to efficiently store and access this information in each voxel during rendering. Using the raw prefiltered input can quickly become intractable. Fortunately, the input plenoptic data has plenty of coherence that can be effectively compressed with a network.

The encoders are made up of convolutional neural networks (CNNs) that operate on 2D slices of the original data. The use of convolutional encoders is also present in the architecture of Rainer et al. [2019]. For example, for the phase encoder and a given outgoing direction, we take the throughput for all incoming directions and arrange them into a uniform 2D grid corresponding to their spherical coordinates, $\omega_i = (\theta_i, \phi_i)$. These 2D grids can be viewed as separate images, one for each outgoing direction, that are all sent to the encoder to be processed one at a time. Thus, our input is $N \times N \times k$, where N is the grid resolution of our table in a single dimension and k is the number of outgoing views that are fed in. The encoders separately process the k slices from each viewing direction and then concatenate each of the k resulting latent vectors together at the bottleneck. We found concatenating latent vectors performed significantly better than averaging when using equal vector sizes.

In order to support both single directional queries (e.g., point sampling light sources) and range queries to efficiently handle area light sources (e.g., environment maps), we use two different kinds of phase networks. During rendering, with a single inference, we can determine the throughput for multiple directions simultaneously, which can be used to perform the dot product with a large light source, such as an environment map. Without such a network, we

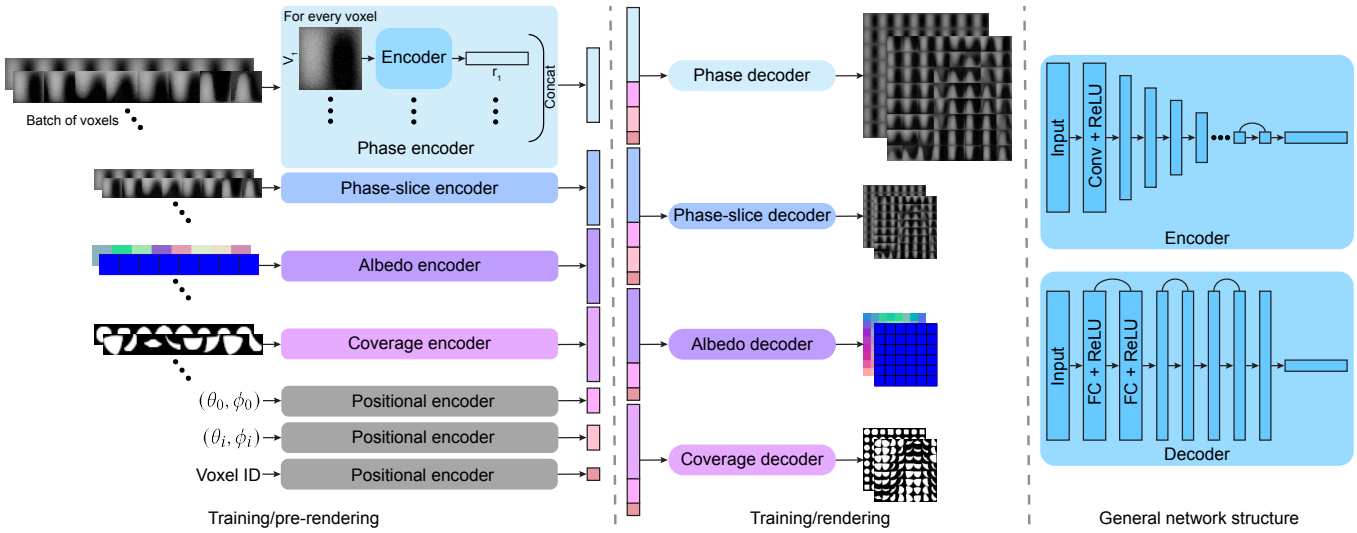


Fig. 6. An overview of our four networks: *phase*, *phase-slice*, *albedo*, and *coverage*. The positional encoder is an explicit function [Mildenhall et al. 2020] and not a learned one. All our learned encoder/decoder nets compress the input data for each voxel into latent representations during a preprocess step. During rendering, a decoder takes the latent feature vectors for active voxels along with directional queries from light transport to evaluate these functions. The phase net returns a scalar throughput at a specific 4D direction, while the phase-slice net returns a 2D image corresponding to the throughput for incident directions conditioned by a particular view direction. The coverage net and albedo net return a 2D coverage mask and RGB base color, respectively, conditioned by outgoing direction. On the right is the general structure of the encoders/decoders used within the different network modules.

would have to point sample a range query, which could result in noise and would require many calls of the point query network.

Decoders. The goal of the decoders is to accurately access elements from the original data during rendering using the latent vectors produced by the encoders, directional queries (both outgoing and incoming), and voxel ID. Of course, these queries can exist anywhere within the plenoptic data, including regions outside the available data, requiring the decoder to be able to interpolate. At render time, none of the original scene data (e.g., geometry and materials) is directly available to the decoder, which requires the encoder to compress the essential information into the latent representation.

The decoders consist of fully connected layers, as in Rainer et al. [2019] and Mildenhall et al. [2019], that operate on flat vectors, rather than 2D images typical of convolutional networks, which can be efficiently evaluated during runtime. It is worth emphasizing that the decoders only return a small subset (e.g., a single value) of the original data corresponding to the query direction, in contrast with autoencoders that reconstruct the full input, allowing the memory footprint to remain minimal when rendering. Thus, the phase and albedo decoders output a scalar and RGB value, respectively. Meanwhile, the phase-slice and coverage decoders both generate 2D images. Still, these images are conditioned by a single outgoing direction query.

Phase networks. The *phase network*, takes both outgoing and incident directions as input to the decoder and provides a single scalar throughput value, which is useful for sampling light sources. The range query network, called the *phase-slice network*, will take in only the outgoing query direction for the decoder to produce a uniform grid containing the throughput for all incident directions at once.

The phase and phase-slice encoders take in the same data, except we use a downsampled version of the original phase data as input to the phase-slice encoder, since reconstructing the original resolution at the decoder backend requires a larger, more expensive network and would be intractable during rendering. Furthermore, we found that using separate encoders was more effective than having the decoders share the latent vector of a single encoder.

Input parameterization. First, we reparametrize all our input data so that the coordinate frame is in a local space of the view direction instead of a global world-space parameterization. This bypasses the additional complexity of having the network learn how to shift phase functions based on the outgoing direction. With this scheme, all the views of a diffuse sphere would have the same orientation for the 2D throughput across incident directions, whereas globally they would be translated versions of each other. To facilitate training, we employ standard practice and normalize the incident and outgoing directions and voxel ID to be between [-1,1]. Next, we apply positional encoding on all three vectors, as in the recent NeRF architecture [Mildenhall et al. 2020], since it helped the network recognize patterns in the data and improved convergence time.

The latent vectors do not undergo any additional processing with the exception that the albedo compressed representation has the original raw albedo that was input to the encoder concatenated to it, as we found this helped interpolation and had a low memory overhead (only k additional RGB values need to be stored or, in other words, one RGB base color for each encoded view). The raw coverage and albedo values are used as provided by the rendering system since they are already low dynamic range (LDR).

However, our phase data is high dynamic range (HDR), which presents two issues: 1) the influence of dark regions is lower than

that of brighter ones when using a typical ℓ_1 or ℓ_2 norm, and 2) high values can produce large gradients that make network updates unstable. To combat this, learning-based methods have applied logarithmic transformations [Bako et al. 2017; Vogels et al. 2018] and gamma compression [Lehtinen et al. 2018] to reduce the gap in range. For our system, using another alternative, the range compressor, performed best. This function was applied in recent works for high dynamic range imaging [Kalantari and Ramamoorthi 2017, 2019] and importance sampling [Bako et al. 2019], and is written as:

$$T(x) = \frac{log(1+\mu x)}{log(1+\mu)}, \tag{18}$$

where $\mu = 2$ controls the degree of compression. We use our range compressor only to preprocess the inputs to the network and leave the reference data for training in the linear domain. In general, having network inputs with large values negatively impacts training and can cause exploding gradients. Indeed, we observed significantly more stable behavior and a lower converged error after applying the range compressor to the inputs. A similar observation was described in Lehtinen et al. [2018], where the authors used tonemapping, albeit using a different function, only on the inputs of their denoising network for the optimization benefits and similarly kept the reference in the linear domain.

5.2 Training

Training is performed end-to-end, with both the encoders and decoders trained simultaneously.

Data. To sample the outgoing directions, we use stratified sampling to better ensure coverage over the entire sphere. This means that, within each stratum, we generate a uniform random sample. In the encoding step, we send k representative views for determining the latent encoding. For generating query samples, we sample from all available views, including those not provided to the encoder, which enables the decoder to generate accurate results for any direction.

We employ a continuous sampling of the space and have the data generator running during training, which will periodically update the data stored at each voxel using different outgoing directions. Without this, we found the network would easily overfit to the static views in the table and would not interpolate well to other non-computed views. Note, generating data on the fly can mitigate storage costs (i.e., the networks can consume new training data as it is produced instead of placing it on disk), but this does not forego our discretized tables for the coverage and phase-slice networks.

Loss function. For our loss, we optimize:

$$\mathcal{L} = \|\hat{\alpha} - \alpha\|_2 + \mathbf{1}(\alpha)\left[\frac{(\hat{\rho}-\rho)^2}{(\hat{\rho}+\epsilon)^2} + \|\hat{\gamma}-\gamma\|_1 + \|\hat{P}-P\|_2\right]. \tag{19}$$

Here α and $\hat{\alpha}$ denote the reference and network estimate of the voxel coverage. Similar definitions follow for phase, ρ, albedo, γ, and phase slice, P. We use $\epsilon = 0.01$ to avoid division by zero. The $\mathbf{1}(\alpha)$ is an indicator function that is on if the reference coverage, α, has any nonzero pixels for the current slice. There are certain views of non-empty voxels where no coverage is detected, such as the case of a triangle viewed along its edge. In these cases, the network estimate of the coverage should still match the all-zero coverage mask of the reference, but we no longer need to force the phase and albedo networks to predict zero. Since a zero-coverage view should not have any contribution, the indicator function allows the network flexibility in not needing to accurately predict the remaining terms.

For the phase term, we use the relative MSE metric as defined in Noise2Noise [Lehtinen et al. 2018], which is more robust in handling HDR data by being less susceptible to outliers compared to an ℓ_2 norm. Note, as described in that work, we use the network prediction, $\hat{\rho}$, instead of the typical reference, ρ, in the denominator to avoid biasing the expected value through a nonlinearity. For this same reason, we avoid using the range compressor (Eq. 18) on the reference data, which would introduce a nonlinearity that can map small errors in the compressed domain to large errors in the linear domain, creating noticeable problems in the final image. Thus, as in Lehtinen et al. [2018], we train in the linear domain. Although both the phase (ρ) and phase slice (P) are HDR, we found that the typical ℓ_2 norm performed better when generating a 2D image and averaging across pixels, as in the phase slice P, whereas the relative metric was more robust for the single point queries of the phase, ρ.

5.3 Prerendering and runtime

After training, we must encode voxel data into latent variables to be utilized during the subsequent runtime rendering.

Data. First, we found it necessary to render out inference data for each voxel, which differs slightly from our training data. We use random views generated with our stratified sampling strategy, but only create a single set of random outgoing directions and use this same set for computing the data at every voxel. Using random views at each voxel in the same manner as our training data caused objectionable noise in the final image due to slight variations in the latent vectors, and hence the output values, at adjacent pixels. We also attempted using data generated with a fixed uniform stride but found this resulted in unacceptable structural artifacts/patterns.

Encoding and prerendering. Afterwards, the encoding happens as a preprocess step before rendering to generate and store the latent features for each voxel, which can be interpreted as compressed representations of the input data. After saving this vector for all voxels, we can discard the encoder. Next, the trained weights of the decoder are frozen and exported to inference within the renderer.

Rendering. Finally, during rendering, we look up the latent vector and append directional information from the current render to query with the decoder-only portion of our network to generate a specific entry from the original tabular data. Note, the original geometry and materials are no longer used at this stage, and all values are computed through SVO lookups and network inference.

5.4 Implementation details

Data generation. Our framework was implemented using our own GPU ray tracer developed with CUDA and OptiX [Parker et al. 2010] and we used a cluster of NVIDIA Volta GPUs (256 in total for computing data and training). For each scene, a full cycle of data generation took ~0.5 to 2 days (depending on the number of voxels).

At any given time, each voxel had 16 slices of data (where a slice is a 2D image of the 4D light field obtained by fixing the outgoing

direction and having each pixel correspond to a different incoming direction), each computed with 256K samples and corresponding to different outgoing directions. Note, we found this sampling rate sufficient for all voxels in the multi-scale hierarchy, even those from the coarse scales despite them accounting for a larger portion of the entire scene and potentially more complex light transport. For each voxel, we also save out the bounding sphere around the actual geometry (4 floats total, 1 float for radius and 3 floats for the offset). This is another form of regularization for the data so that the coverage masks are more uniform across slices and so the network does not have to learn translations. Moreover, it facilitates training by making the coverage masks larger for voxels with tiny geometry allowing for more gradients in the coverage regions and, thus, better optimization. Note that during runtime, we re-scale and translate the network's predicted coverage mask using this information to accurately compute the transmittance.

Our uniform grid sizes are $N = 128$ for the phase encoder, $N = 16$ for the coverage and phase-slice encoders, and $N = 1$ for the albedo encoder (i.e., 128×128 and 16×16 images and a 1×1 scalar, respectively) and they correspond to spherical coordinates for the incoming direction. These dimensions are used for both training and runtime. In all encoders, we use $k = 8$ random views, one from each octant of the sphere, but chosen so that samples uniformly cover the sphere. Each of these corresponds to a different viewing direction that is constantly updated to a new viewing direction by the data generator to allow for a continuous sampling of the space. Note, these views are randomly selected for each iteration and are only used to encode the latent variable, while the point query training samples can come from all the data available on disk for that voxel at any given time (e.g., 16 views per voxel), not only from the $k = 8$ random views fed to the encoder. For example, for the phase tabular data, each voxel would have $16 \times 128 \times 128$ floats saved on disk, which are constantly being updated with random views by the data generator during training. We downsample the phase-slice encoder's input images using OpenCV's resize function with the area filter option.

Beam tracer. Another challenge when rendering with a beam tracer is the common case where beams are not axis-aligned with the SVO, so multiple voxels can fall within a beam at varying depths and locations and with only partial contributions. To solve this issue, we perform beam marching across sets of voxels ordered by distance, which we call *wavefronts*. A wavefront consists of all the voxels in an interval (e.g., the size of a voxel length) that overlap with the beam. Within a wavefront group, we essentially stitch together the coverage masks from each voxel at a given depth and splat them onto the beam's coverage mask, which we continue to track as we march further along our beam. We weight each voxel's contribution by the proportion of area it adds to the beam. Note, the coverage network robustly interpolates the 2D mask across continuous viewing directions due to our constantly updated training set (described later in this section). This implementation, while not trivial, will be part of our full code release upon publication. Pseudocode of our overall algorithm can be found in Appendix B.

Furthermore, we use discrete LoDs in our system, but this can lead to popping artifacts from abruptly switching between one scale and another. To avoid this issue, we use standard trilinear interpolation between the voxels in the wavefront of the current scale and those in the next coarser scale, as is commonly done for LoD prefiltering [Loubet and Neyret 2017]. Note, we train a single set of networks across all of the voxels in all of the discrete scales of a given scene, so this explicit interpolation helps avoid popping between scales. A more systematic approach that perhaps blends latent vectors of voxels in different scales is left for future research.

Shadow maps. To enable shadows in our current system, we capture intra-voxel and inter-voxel shadowing. For the former, we can save each voxel's precomputed data with self-shadowing accounted for. For example, if two triangles are inside a given voxel, certain lighting and view directions can result in one triangle occluding the other. We incorporate this local shadowing into our data generation step.

On the other hand, to capture global shadowing, we make use of the common strategy of shadow maps [Williams 1978]. Specifically, we first render the depth of the directly visible voxels from the direction of each light source as a pre-process. In the case of environment maps, we compute a shadow map for each direction corresponding to the bins of the low resolution grid used in our phase-slice network (center of the bin was sufficient in practice). All surfaces that are directly seen from the light source can potentially have a radiance contribution, while those surfaces that are not visible will be in shadow. Then during final rendering, while accumulating the contribution of each voxel during beam marching, we use the voxel's world coordinates to project to each light's coordinate system and look up the voxel's depth at the corresponding pixel of the light's shadow map. If the current voxel's depth is greater than the depth in the shadow map then the voxel is occluded and does not have a radiance contribution at the current pixel being rendered.

Networks. We implemented our networks in TensorFlow [Abadi et al. 2015] and used Xavier normal initialization [Glorot and Bengio 2010] and the Adam [Kingma and Ba 2014] optimizer with a learning rate of 3.0×10^{-4} and mini-batches of 1 voxel with 4096 queries. Training took ~2 days, on average. For more efficient inference, we used the TensorRT library within our renderer to evaluate the decoders with the trained weights at the query directions.

For the CNN encoders, we start with a relatively small number of feature channels in the input layer (e.g., 8 for the phase encoder), but then double the layer size after each subsequent reduction of the spatial resolution (clipped to be a maximum of 256 channels). The spatial downsampling was performed with strided convolutions, which performed better than average or max pooling, and ReLU was used for the activation function on all layers except the output layer, which was linear. After concatenating the encodings of all the views, each encoder outputs its final latent vector of 256 floats.

The decoder networks all consisted of three residual blocks (each block has two layers of non-linearity) with ReLU activation functions and differed only in the output layer. The coverage decoder used a sigmoid output activation, whereas the other networks used a linear activation, as is common practice.

All networks employed residual connections where possible (when layer count and spatial dimension matched) to mitigate general optimization pitfalls, such as vanishing gradients, which thereby helped reduce convergence time. Finally, we train the networks from

Cutlery	598,032	**Oak**	12,589,056
Mossy Rock	5,996,334	**Forest**	53,948,550
Parking Lot	24,165,771	**Stormtrooper Army**	175,986
City	87,433,857		

Table 1. Number of triangles for each scene in the paper.

scratch, as we found no noticeable improvements when pretraining an autoencoder architecture to stabilize the encoder weights before using transfer learning with the final decoder architecture.

Regarding our memory requirements, we store voxel ID (1 float), bounding sphere information (4 floats), and a latent vector (792 floats total consisting of 256, 256, and 280 floats from the phase or phase-slice, coverage, and albedo encoders, respectively) at each voxel, making the memory size 797 floats, just over 3 KB per voxel.[3] If we used all four sub-networks (i.e., both the phase and phase-slice networks, not just one), for example when using point light sources together with an environment map, then the vector would be 1048 floats or 4 KB. All of the scenes in the results use only environment maps or point light sources, so our memory estimates are based on using 3 KB per voxel. Note, it could be possible to utilize half-floats (i.e., 16-bit floats) instead of the 32-bit floats at various stages throughout our pipeline for a potential 2× savings, but we leave this optimization for future work.

6 RESULTS

We demonstrate how our method, called Deep Appearance Prefiltering (or DAP), can preserve appearance for level of detail (LoD) of previously unsupported materials, while also robustly handling heterogeneous cases of surfaces and volumes. Our pipeline also scales well to assets with more complex geometric and material properties.

In our comparisons, we report mean squared error (MSE) to quantitatively assess the image quality with respect to the reference, and direct readers to the interactive viewer in the supplemental to see other metrics including the perceptually-based Structural Similarity Index (SSIM) [Wang et al. 2004]. In addition, the supplemental has full color images and error heat maps across all scales.

Our OptiX [Parker et al. 2010] ray tracer is used with a single bounce to generate training data and results for the "Reference" method. The results corresponding to "Ours" are generated with a custom GPU beam tracer that traverses our SVOs and uses TensorRT to inference the trained decoder networks. The beam tracer does not have access to any geometry or materials of the original scene and generates the image simply by evaluating the networks for an ordered set of voxels at each pixel and combining the results using our transmittance model described in Sec. 4.1. For all scenes, we use the Disney BRDF, which we ported to Mitsuba [Jakob 2010] as it is not supported out of the box. We verified that the high-sample-count, ray-traced images generate the same results across renderers. For the state-of-the-art comparisons in Mitsuba, we rendered 256 samples per pixel to ensure there is no visible noise since the approaches operate with rays. Note our approach still only has a single beam per pixel. Our intent is to instead demonstrate the fundamental limitations of the appearance models that our method overcomes.

[3]There is also a fixed cost that we include for the decoder network weights and data structures (e.g., the SVO), but these are relatively small and negligible.

Where applicable, the methods in this section sample according to the Nyquist frequency and utilize voxels from the next finer scale relative to the one determined by the pixel filtering kernel to avoid aliasing and overblurring, as is done in Loubet and Neyret [2017]. We also display difference images with respect to the reference that are color coded from blue to red with increasing error. Finally, for all results and methods, we include a background layer that undergoes standard ray tracing without prefiltering, and which corresponds to our boundary term from Sec. 3. The images in this section are rendered at a resolution of 1024×1024. The complexity in terms of number of triangles for a single instance in each of our scenes can be found in Table 1.

6.1 Appearance models

We begin by comparing against state-of-the-art appearance models including the symmetric GGX microflake distribution (SGGX) [Heitz et al. 2015], the microfacet model with an ellipsoid normal distribution function (EGGX) [Dong et al. 2015], and the recent microflake model with self-shadowing (MMSS) from Loubet and Neyret [2018]. For MMSS, we use the author implementation, which also included comparisons to EGGX and SGGX. Although MMSS was demonstrated for the application of appearance-preserving volumetric downsampling, we compare to it here for its novel microflake model which captures the self shadowing/occlusions that occur within a voxel for improved accuracy relative to SGGX. Since this method only supports a specular lobe, we also modified the original code to include an additional ellipsoid diffuse lobe.

In general, due to their relatively simple phase functions, these approximate models are unable capture a wide range of effects that we highlight in Fig. 7. Note, EGGX still has access to the scene's mesh, while our approach, like SGGX, does not use the original geometry and materials yet is still the most faithful to the reference. Furthermore, since we include a diffuse EGGX lobe for MMSS, it is also not fully volumetric. The comparisons in Fig. 7 are done using a directional light source, so we use our phase network, which outputs the throughput at a single point query.

The **Cutlery** scene shows how the previous approaches struggle with specularities. In particular, they fail to preserve the anisotropic highlights along the spoon which are overblurred. This scenario has a watertight surface, so volumetric representations (SGGX and MMSS) do not perform as well as EGGX, which uses simplified geometric representations. Our approach does not try to fit an approximate model to this difficult scenario. Instead, it learns a significantly more precise phase function that was captured directly from the ray tracer, preserving the highlights. Meanwhile, in **Oak**, SGGX and MMSS fail to capture the glints on the leaves, since sharp specularities are difficult to model with microflakes.

6.2 Hybrid approaches

We compare against the author-provided implementation of the state-of-the-art, hybrid mesh-volume approach (HybridLoD) from Loubet and Neyret [2017] in Fig. 7. This method uses heuristics in a mesh-based analysis to label regions of the asset as either macrosurfaces amenable for geometric simplification and an EGGX microfacet model or sub-resolution microgeometry better represented through

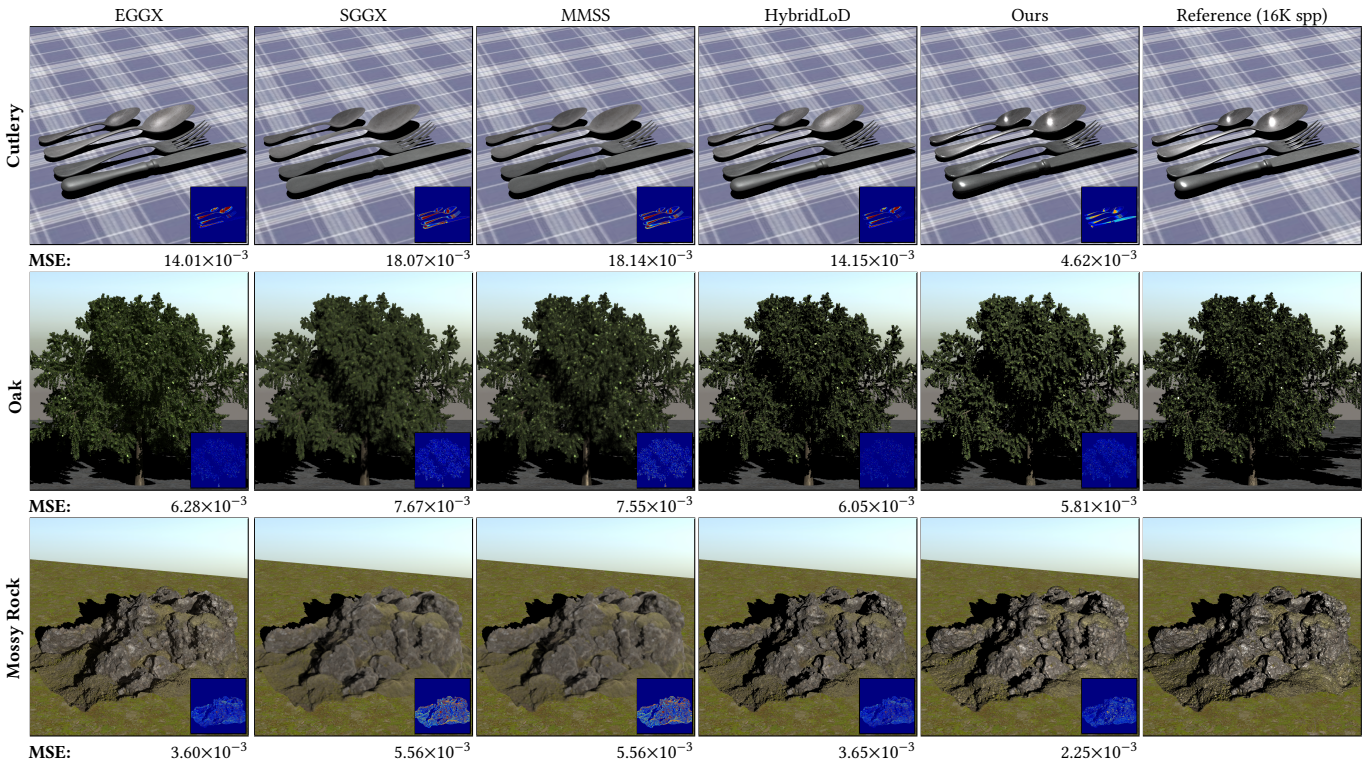


Fig. 7. Comparisons with state-of-the-art approaches EGGX [Dong et al. 2015], SGGX [Heitz et al. 2015], MMSS [Loubet and Neyret 2018], and HybridLoD [Loubet and Neyret 2017]. For all scenes, we demonstrate higher fidelity to the ray traced reference, as can be seen in the difference images (mapped from blue to red with increasing per-pixel squared error) and the lower MSE error. Note, only SGGX and our method do not rely on any explicit geometry or materials for rendering. Full results can be found in the supplemental along with videos.

volumetric rendering with an SGGX microflake distribution. The results of this method for the previous scenes are included in Fig. 7 to demonstrate that, since HybridLoD still uses established microflake models, it will also have the same issues as described earlier with preserving a wide range of complex effects and appearances, due to the limitations of using an approximate model.

The **Mossy Rock** scene shows moss growing on a rock, creating a rough, detailed surface on top of a large watertight mesh. The hybrid approach misclassifies this as sub-resolution geometry that can be rendered volumetrically, resulting in an overblurred appearance. On the other hand, our approach avoids classification altogether and will render the appearance more accurately. Furthermore, for the **Oak** scene, the supplemental shows that for coarser scales (e.g., scale 7 and below), HybridLoD tends to overblur the leaves of the tree as it relies increasingly on a volumetric representation.

In Fig. 9, the plot on the left shows the average MSE loss for all methods for the scenes in Fig. 7 across scales from coarse (scale 4) to fine (scale 8) corresponding to images of resolution 16×16 and 256×256, respectively. Note, we do not plot results for coarser scales to ensure fairness, as previous approaches can output degenerative meshes from their geometric simplification pipeline, which resulted in all black images. Our approach does not rely on explicit meshes and thus always works at all scales. Still, at the scales shown, our method has the lowest average error relative to the other state-of-the-art methods independently of scale.

6.3 Complex scenes

Since our framework can capture a wide range of appearances without having to rely on simplification of explicit geometry or volumetric microflake representations, we are able to demonstrate results on significantly more complex scenes (see Fig. 8). In their current implementations, both the HybridLoD and MMSS code base operates on a single asset (e.g., as a texture-mapped obj) to perform the prefiltering and the environments shown here are extremely difficult to model as a single asset and texture due to their complexity. On the other hand, it is not a trivial extension to extend the algorithms to work on a collection of assets. Thus, we only compare against the reference ray-traced result, which we closely resemble in terms of final image quality, but with a significantly reduced per-pixel memory footprint. Note, the comparisons in Fig. 8 use an environment map, so we use our phase-slice network, which outputs the throughput at a 2D slice for the particular view direction and can be used in a dot product with the environment map to determine the incoming radiance at all solid angles simultaneously.

Fig. 8 shows our results for four scenes at a coarse and fine scale along with the reference renders. As shown in the color-coded difference images (closer to red is larger error), we accurately preserve the appearance of the ray traced result. However, we do so with only a fraction of the memory cost. Below each result, we report the per-pixel memory footprint. For the reference, this is computed by

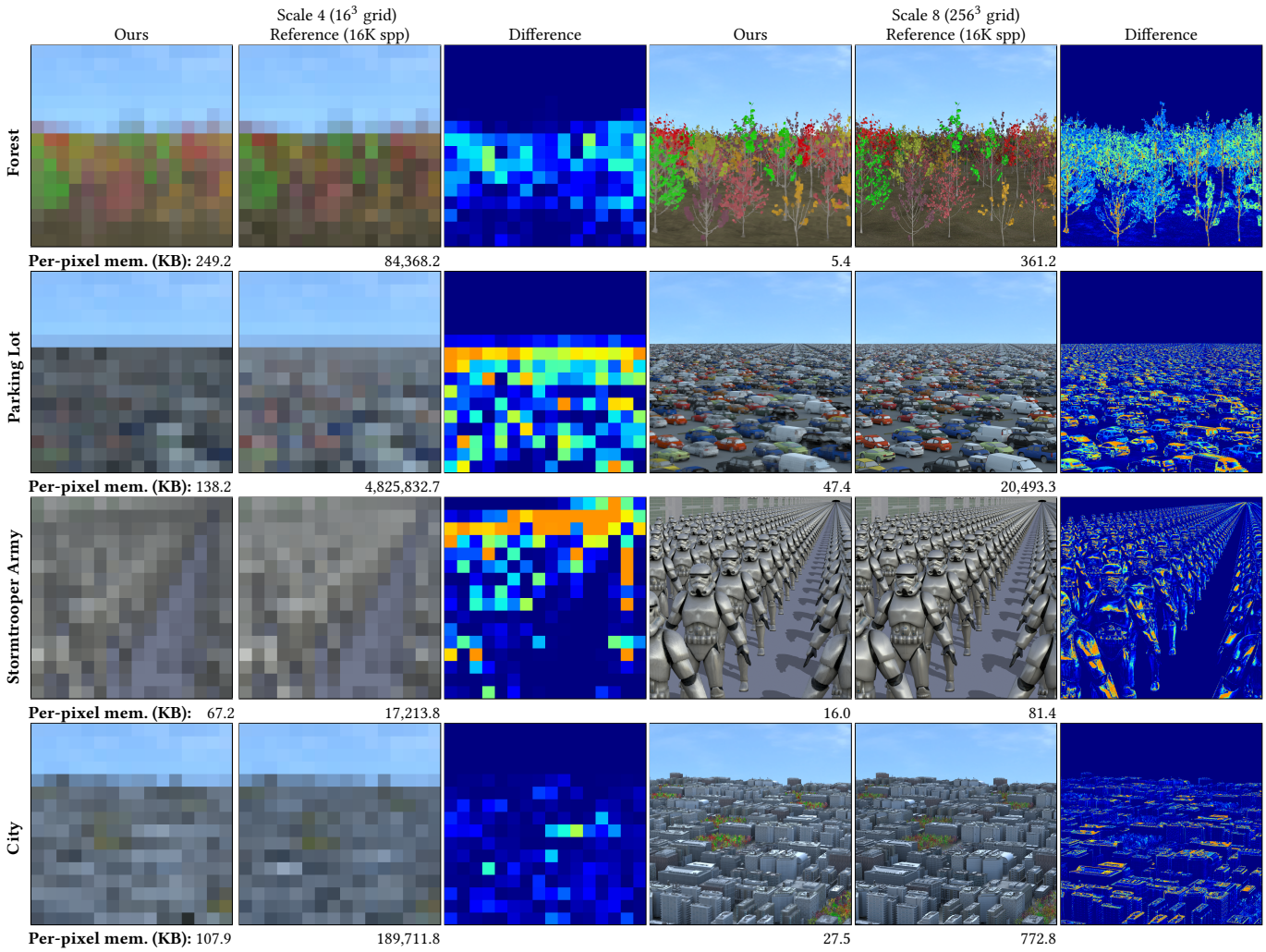


Fig. 8. We compare to standard ray tracing on large scenes with significant geometric and material complexity for a coarse scale (scale 4 on the left) and a finer scale (scale 8 on the right). For example, at scale 4 the voxels divide up the scene as $16 \times 16 \times 16$ or coarser and scale 8 divides up the scene as $256 \times 256 \times 256$ or coarser. Our method's output closely matches the reference, demonstrated by blue (lower error) to red (higher error) MSE difference images, at only a fraction of the memory cost, despite not using any of the scene's original geometry and materials. The memory consumption is reported per-pixel and is computed for ray tracing by dividing the size of the scene (e.g., the size of the mesh and materials) by the number of active pixels for a given scale. Similarly, for our method, we divide the total size of all of the active voxels' latent variables by the number of active voxels (the latter is typically larger than the number of active pixels since multiple voxels often map to a given pixel). The scenes here highlight hard-to-capture scenarios including anisotropic specularities using the multi-lobe Disney BRDF as well as mixtures of macrosurfaces and microgeometry illuminated with an environment map. See the supplemental for the full images, as well as videos of these scenes demonstrating our network's ability to render temporally-smooth, anti-aliased sequences.

taking the memory size of a scene and dividing it by the number of pixels that the scene maps to (i.e., the projection of the asset to the image plane) when the image resolution matches the corresponding LoD scale. To illustrate, for an axis-aligned view, the bounding box of a prefiltered asset at LoD scale 4 would cover 16×16 pixels, while such a view at scale 8 would be 256×256. Setting a certain scale means that the voxels would be no finer than those found at that scale. For our method, the per-pixel memory footprint is computed by taking the number of touched voxels, multiplying by the memory size of the voxel data, and then dividing by the number of pixels for the scale. Note, we show our results here with instancing to highlight the compelling cases our method could be used. In a production setting, the tiles would correspond to different assets that are prefiltered to make up a scene, but the savings in our per-pixel memory footprint would persist since our approach would evaluate roughly the same number of voxels regardless of scene complexity.

As with all LoD methods of this nature, there is an inflection point where the benefit of prefiltering over standard ray tracing no longer exceeds the relative cost. We can see this in the center plot of Fig. 9, which shows the per-pixel footprint for the **City** scene across scales. If the **City** were to fall entirely within one pixel, as in scale 0, a typical ray tracer would need to load the full scene (over 8GB)

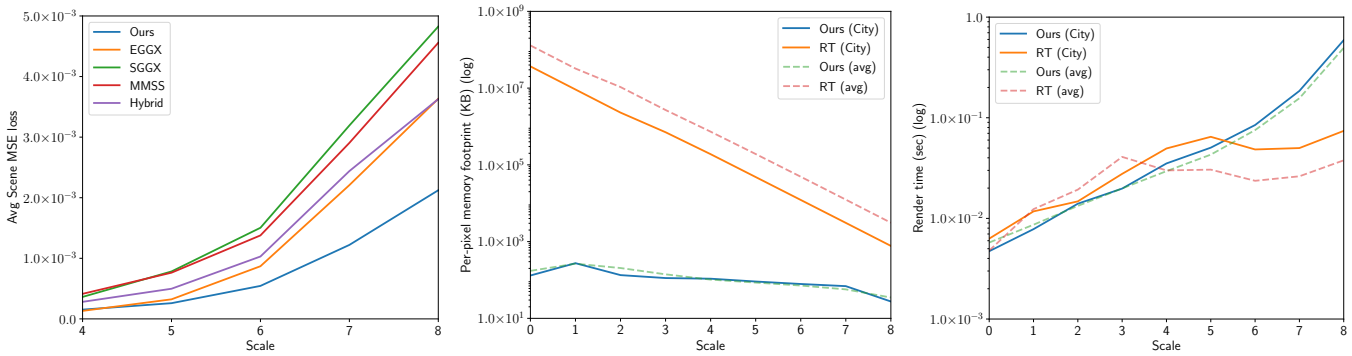


Fig. 9. The plot on the left shows the average MSE across scales for the scenes and methods from Fig. 7. Our approach performs favorably relative to previous state-of-the-art approaches at all scales. Note, we only plot results for scale 4 and above due to degenerative simplified meshes from these prefiltering approaches at coarser scales. The log plot in the middle shows the per-pixel memory footprint (in KB) of our approach compared to standard ray tracing for all scales of the **City** scene and the average across all 7 scenes from Figs. 7 and 8. At the coarsest resolution (scale 0), the entire scene falls under a pixel footprint and we require 5 orders of magnitude less memory to render it. As the image resolution increases, ray tracing memory costs are amortized across the pixels and hence decrease. On the other hand, our footprint remains more-or-less constant across scales, but still yields an order of magnitude improvement at our finest resolution (scale 8), which corresponds to a 256×256 image. The rightmost plot shows our corresponding render times for these scenes as compared to an equal-quality ray traced baseline. At lower scales, our unoptimized prototype tends to be faster than ray tracing up until an inflection point at approximately scale 5, where the incurred cost of network inferencing over multiple batches exceeds the baseline.

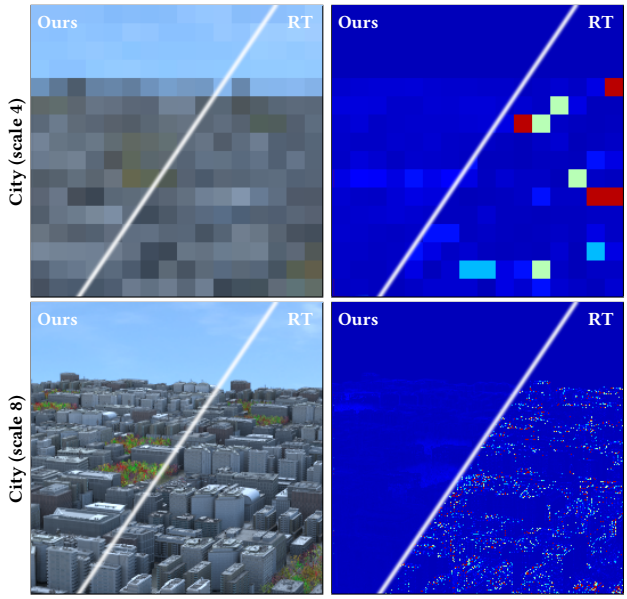


Fig. 10. The two images show complexity as a heatmap of per-pixel timings (blue to red with increasing cost) for our approach and equal-quality ray tracing across scales on the **City** scene. Unlike ray tracing, our method accesses roughly a constant number of voxels at each pixel independently of scene complexity and scale.

and send many samples to properly integrate over the large pixel footprint. This cost is amortized as the footprint covering the **City** grows to 256 pixels in scale 8. However, in our approach each voxel requires only 3KB of memory as mentioned in Sec. 5.4, so evaluating all the voxels along a pixel's footprint requires only about 100 KB of memory on average and is more-or-less constant across scales. This culminates in a savings of ~280,000× for scale 0 and 28× at scale 8 relative to ray tracing. Eventually as the scales get finer, it is more advantageous to switch to ray tracing as one can extrapolate from the plot, but such scales are less common in production. For the typical LoD scales shown here, there is a clear memory savings and this behavior holds true when considering the average memory footprint across all 7 scenes from Figs. 7 and 8.

6.3.1 Timings and complexity. As expected for LoD methods, there is an inflection point where ray tracing becomes faster. For example, at the finest image scale (scale 8), an equal-quality Monte Carlo render (EQMC) requires 128 spp, which takes 74ms on the standard GPU ray tracer, while our GPU beam tracer needs 591ms. However, when we start minifying details (e.g., at scale 5 and lower), our method starts to outperform ray tracing, as illustrated in the rightmost plot of Fig. 9. For a single instance of **City** at scale 4, EQMC would require 128 spp, which takes 50ms on the ray tracer and 35ms on our beam tracer using our framework. The inflection point depends on the implementation efficiency of both algorithms and, with an unoptimized implementation of our algorithm, we observed the inflection point to be around scale 5 across all 7 scenes. One implementation inefficiency occurs at higher image resolutions, where the total number of pixels and inference calls increases and we are technically limited to running a maximum batch of 4096 elements through our pipeline at a time, resulting in low GPU utilization.

The heatmaps in Fig. 10 correspond to the per-pixel complexity as a function of time for our method and EQMC across scales. Specifically, each pixel in ours represents the number of voxels evaluated at that pixel multiplied by the cost of evaluating each voxel. Meanwhile, for EQMC, each pixel shows the number of samples required to reach the same level of MSE error as our approach times the cost of each sample. Note, the per-pixel timings shown here do not capture the overhead of swapping network batches, as discussed in the previous paragraph, since the image could simply be evaluated as tiles in parallel across different GPUs to avoid this cost. Since our

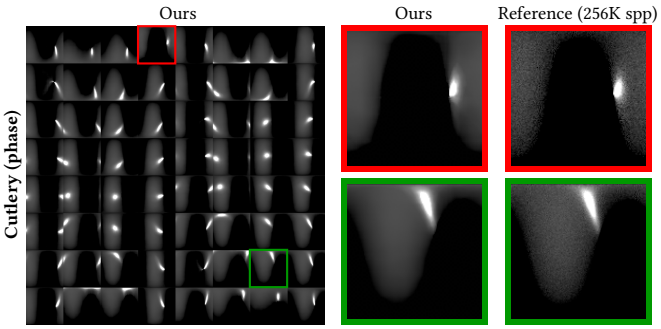


Fig. 11. The full image on the left is a visualization of different view directions along a uniform stride reconstructed using our phase point query network. Each tile is a slice showing the incident throughput (single channel) corresponding to a given outgoing direction. The insets demonstrate how our network is able to fit to the underlying data despite the noise in the reference. Across the two insets, the network faithfully denoises the phase along various supports, while preserving anisotropic specularities.

framework needs to process roughly the same number of voxels everywhere and with the same evaluation cost, we demonstrate constant timings across pixels and scales relative to EQMC, which needs to evaluate more samples based on the complexity of the appearance falling within a pixel's footprint.

6.4 Temporal stability

In the supplemental, we include video fly-throughs for the scenes in Fig. 7 and Fig. 8. Our results are temporally coherent when switching across different voxel scales without any noticeable flickering or popping even though such consistency is not enforced in the training loss for the networks. Moreover, in scenes such as **Stormtrooper Army**, the network produces anti-aliased results for distant stormtroopers along the horizon despite using only a single beam per pixel, while the ray-traced result requires significant samples in this region to reduce flickering. The network is also able to smoothly interpolate across both view and light directions, as a result of the stochastic data generation that continuously updated the voxel data and allowed arbitrary queries.

Note, we did not retrain or use separate networks across scales. By having our network train on voxels from all scales simultaneously, it is able to smoothly interpolate across scales with voxels of varying degrees of geometric and material complexity. In the future, it would be interesting to investigate how interpolation in the latent space of voxels could produce non-linear appearance changes for regions in between LoD scales. This could replace the traditional linear combinations used by previous LoD methods and in our approach.

7 DISCUSSION, LIMITATIONS, AND FUTURE WORK

It is interesting to note that in our system the networks actually do more than just compression; they actually perform both denoising and interpolation. To keep the computation in our data generation step tractable, we limit the number of samples sent out for each outgoing direction which resulted in some noise in the 4D tabular data (see Fig. 11). Here each 2D tile corresponds to a different outgoing direction and each pixel within the tile corresponds to a different incident direction, and each tile can be thought of as a slice of the 4D light field. However, as was observed in the recent Noise2Noise work [Lehtinen et al. 2018], using reference data with slight noise did not cause optimization issues and instead allowed the network to denoise the data and smoothly fit the underlying function.

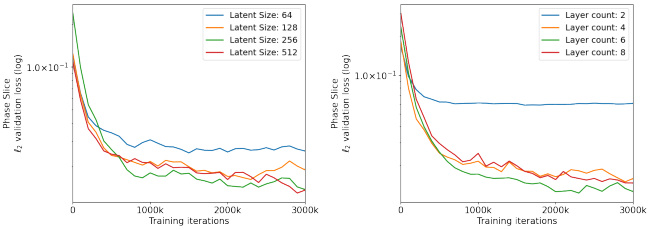


Fig. 12. Ablation studies plotting validation errors (ℓ_2) for the phase-slice network across training iterations using various configurations in the **Cutlery** scene. Increasing the latent size results in better performance, yet gains become marginal for larger sizes (left plot). Increasing the number of network layers also improves performance up to a point (right plot). In our implementation, we use 256 floats for the latent vector and 6 layers in the decoder. See additional ablation analysis in Appendix A.

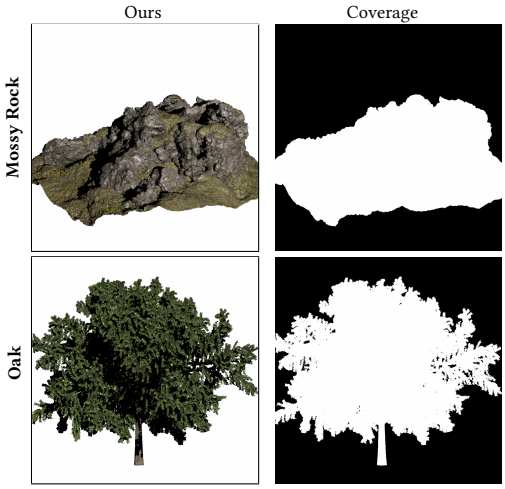


Fig. 13. Total coverage for the **Mossy Rock** scene (top), where transmittance behaves as a watertight surface, and the **Oak** scene (bottom), which behaves volumetrically. Note, the white background does not bleed through the asset.

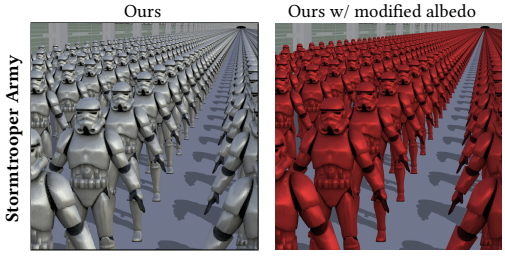


Fig. 14. Our framework's design offers flexibility with its ability to expose certain parameters such as the diffuse albedo to easily modify the scene appearance. Without any additional training or data generation, we can easily change the stormtroopers from the classic white to a modern red.

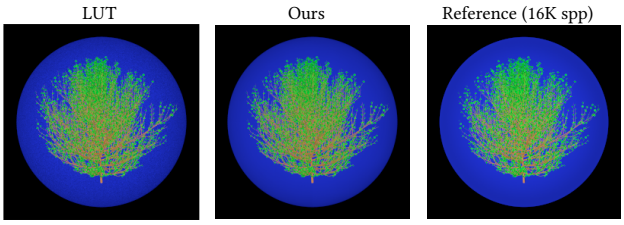


Fig. 15. Results on a toy example of a scene with a bush in front of a blue sphere to demonstrate how well geometric and material correlations are tracked. On the left, we use a lookup table (LUT) to generate the image directly. In the center, we use our network-based compression of the tabular data. Both LUT and ours closely match the ray-traced reference (right).

Fig. 16. Results using a hybrid system with our approach for the minified regions and standard ray tracing for the foreground. Our original approach has blocky artifacts in the foreground since the size of the voxels at the maximum LoD scale precomputed by our framework is significantly larger than the beam at each pixel. The inset straddles a region where the foreground and background layers meet and are blended in a hybrid system.

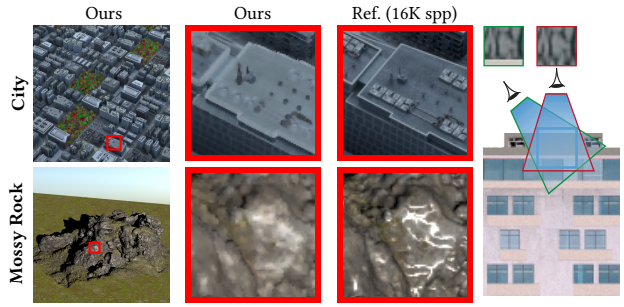


Fig. 17. Limitations of our approach. The first row shows a roof in the **City** scene for which our results are not as dark as the reference. This artifact stems when a voxel straddles occluded geometry with a different base color than the unoccluded region. Specifically, the roof has a room directly below it that has a light colored floor and which is included in the voxels containing the roof. When calculating the average RGB albedo for the directly overhead view (red beam) the albedo contains only the roof texture, however grazing views (green beam) can include the floor's albedo causing the lighter color we see. The albedo of the projected cross-section of the voxel is shown for these corresponding views. Another limitation can be found in the bottom row for the **Mossy Rock** scene. Sharp delta highlights that only occur over a very small solid angle are not always preserved perfectly across all voxels and can get slightly overblurred.

In Fig. 12, we show ablation plots for the **Cutlery** scene. The first plot shows the ℓ_2 error of the phase-slice validation data (a subset of the voxels in the training data since we are overfitting to a given scene) with varying sizes of latent vectors. We see that as the size increases from 64 to 512 floats, the converged error decreases due to the network's ability to store more information within these latent features and its additional coefficients. However, the data suggests there are only marginal returns once the vector is sufficiently large (e.g., increasing from 256 to 512 floats has comparable performance despite requiring double the memory). This tradeoff was not beneficial, so we kept the vector size at 256 floats.

The second plot of Fig. 12 evaluates the impact of decoder size on validation error. As before, we observed better performance as the number of layers increased from 2 to 6 layers. At 8 layers, we found that performance slightly decreased relative to 6 layers (used in our final implementation), in particular for the phase-slice network. This could be due to difficulty optimizing the larger network. In the other decoders, 6 and 8 layers performed comparably, so we decided to use the more lightweight 6-layer architecture consistently across decoders. Appendix A has additional analysis on plots corresponding to the remaining three decoders for the **Cutlery** scene.

Fig. 13 serves as a sanity check of our transmittance model by placing the assets in front of a white background. The top row shows the **Mossy Rock** asset which is a large macrogeometry with a rough, yet watertight, surface. Meanwhile, the bottom row shows the **Oak** asset which contains many small leaves in random locations, where this sub-resolution microgeometry behaves relatively more like a volume. In both cases, the white background does not bleed through showing that our beam has achieved full coverage and our transmittance model was sufficiently accurate.

Fig. 14 demonstrates another application of our approach. Once an asset is prefiltered in our framework, it can still be modified with trivial changes without any computation of additional data or further training. For example, in the **Stormtrooper Army** scene, we can modify each voxel's diffuse albedo by scaling up the red channel of the albedo information in their latent vectors to switch from white stormtroopers to red ones. Moreover, a simple extension to our current approach would be to additionally save out the specular albedo to have control over that as well. Although we show modification in the diffuse albedo across all the voxels in this scene, selecting only a subset of voxels to modify, perhaps through a GUI, is readily realizable. Finally, it would be interesting to explore how to expose additional material parameters, such as roughness, for manipulation within our latent data to control the glossiness of the full asset or portions of it.

In Fig. 15, we set up a toy example with a bush in front of a blue sphere in order to demonstrate how our framework tracks material and geometric correlations and synthesizes an image. If we use the saved out tabular data as a look up table (LUT) we can generate an image that closely resembles the high-sample-count reference rendered with a ray tracer showing that our assumptions and approximations hold up sufficiently. Moreover, our full network-based approach successfully compresses the data without a noticeable loss in quality. In fact, our full approach is able to remove the small noise found in the LUT result since the network denoises the tabular data as shown in Fig. 11.

Some of the scenes in Fig. 8 contain blocky artifacts in the foreground regions of scale 8. This is from the pixel footprint being significantly smaller than the size of the discrete voxels found at our finest scale. For example, in the **Parking Lot** scene shown in Fig. 16 we see that our precomputed voxels are too coarse to accurately represent the foreground objects. However, we can use a hybrid system, a more pragmatic solution for a production pipeline, where we blend between the foreground and the simplified background based on the size of the pixel footprint. Specifically, for pixels where

the voxels are too coarse, we simply use standard rendering and use our approach on pixels where the voxels are a pixel or smaller. With this strategy, we can accurately represent larger objects that are close to the viewer, while still saving on costs by using our prefiltering approach on the background layer. Note, with additional computation finer scales can be precomputed in order to render a larger region of the foreground with our framework.

However, our method has some limitations that are the subject of future work. As noted previously, our approach requires precomputing the voxel data and training on a per-scene basis, which can be a significant computational cost for some pipelines. Constant improvements in real-time and GPU rendering enable faster data generation to alleviate the former issue, while leveraging faster training methods [Müller et al. 2022] could help with the latter.

Another limitation is shown in the top row of Fig. 17 depicting the roof of a building in the **City** scene that is lighter than the ray traced reference. The voxels on the roof straddle a room whose interior is occluded normally. Thus, the albedo of the room's floor (which is lighter than the roof) is included in the average base color we store. To properly handle such a case, we could instead use an RGBA mask (instead of just the alpha channel as with the coverage mask), a 4× increase in correlation-related data. By having the albedo spatially represented, we could determine which regions are occluded and omit them from the average base color.

Extending the phase function to have a spatial component instead of only a scalar can be similarly justified. However, this would increase the data by $M \times M$, where M is the desired spatial resolution of a single slice (e.g., the element of a fixed outgoing direction would be a 4D LUT of size $N \times N \times M \times M$, where $N = 128$ in our implementation). Note, these estimates of increased data do not take into account the likely necessity of increasing the size of the latent vectors for each voxel as well as the networks in order to more accurately capture the additional data. Thus, in this case, there is a high potential for both the memory footprint and the runtime during rendering to increase. Moreover, the single RGB base color per view direction and single throughput per incoming and outgoing direction was sufficient for the scenes shown here. Finally, the current framework demonstrated results with a diffuse base color, but it would be interesting to extend the framework to additionally handle a specular albedo, perhaps in a similar fashion with an additional latent vector and sub-network or to combine the latent vectors with a single albedo decoder.

We used a relatively simplistic transmission model for combining adjacent voxels. In particular, we chose a coverage mask to determine occlusions while tracing beams through our SVO. This approach is limited by the resolution of our masks and there could be inaccuracies from subpixel occlusions. Properly accounting for correlation in volumetric rendering is active research [Bitterli et al. 2018; Jarabo et al. 2018; Kettunen et al. 2021; Vicini et al. 2021] and more sophisticated formulations could be applied in our framework.

In the second row of Fig. 17, we show another limitation present in the **Mossy Rock** scene. If the highlights of a voxel are present in an extremely small solid angle of only a few bins, it poses a difficult scenario for our networks to capture exactly, especially when training on the order of a million voxels. Thus, such sharp highlights in some cases can be slightly overblurred by our approach.

Here, we focused on properly handling primary beams, which tend to be among the most difficult aspects since their contribution, and any artifacts, would be directly visible. It would be interesting to apply our framework to render effects such as global illumination by casting multiple beams. This could be done by spawning new beams at every intersected voxel and accumulating the radiance contribution of that sub-beam. Nothing changes from a theoretical perspective, as our algorithm is presented as a general framework with an arbitrary number of bounces (see Sec. 3). Instead, the main challenge would be to efficiently evaluate multiple sub-beams without accumulating costs beyond that of ray tracing. Furthermore, to avoid inefficient, brute-force sampling for generating additional beams or to enable common sampling strategies such as multiple importance sampling (MIS) in our beam tracer, we would need a mechanism for importance sampling the phase function of each voxel. The most straightforward way to do this in our current framework would be to normalize our decoder's phase-slice estimate to convert it to a PDF and then create an inverse CDF for importance sampling. Ideally, however, a learned solution would be more lightweight and flexible. One prospect would be to train a network to take in a set of random numbers and a latent representation of the phase (or its PDF) and output the next sample direction and its probability. Ensuring unbiasedness with such a strategy would be a compelling research direction.

8 CONCLUSION

We present the first deep learning framework for prefiltering complex 3D environments into multi-scale LoDs. Unlike some previous work, we do not rely on surface-only or volumetric-only models to simplify assets. Instead, we voxelize the scene and convert it to an SVO representation. The full appearance of each voxel is then captured through a ray-traced, data-generation step that saves the phase function, albedo, and coverage information in tabular form. By saving the true appearance in this manner, we avoid heuristics for classifying regions of assets, as in hybrid approaches, and can capture effects including sharp specularities that volumetric approaches cannot. Our novel, learning-based compression scheme compresses the rendered data into small latent vectors that can be easily stored and used by a beamtracer to render the final image by utilizing lightweight decoder networks, and without requiring any access to the original geometry or materials. To facilitate community involvement, our code and supplementary material are available online.[4] Finally, we compare favorably to state-of-the-art prefiltering approaches and demonstrate significant memory savings relative to ray tracing on a variety of complex scenes.

9 ACKNOWLEDGMENTS

We thank Delio Vicini, Anjul Patney, Zhao Dong, and Warren Hunt for helpful discussions. Special thanks to Anton Sochenov for his tremendous help including prototyping a real-time demo. We are very appreciative to Guillaume Loubet for releasing code for comparisons of both of his papers. We are grateful for Matt Chapman's help in generating many of the scenes in the paper. We thank the following artists/sources for the models we used in our scenes:

[4] https://doi.org/10.7919/F4NK3C21

GetWreckedDJ (stormtrooper), avi9526 (hangar), Quixel (mossy rock asset and textures), and Ndakasha (cutlery cloth). We thank Facebook Reality Labs for their unwavering support. This work was partially funded by National Science Foundation grants #IIS-1619376 and #IIS-1911230.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, et al. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. 2018. *Real-time rendering*. AK Peters/CRC Press.

Hendrik Baatz, Jonathan Granskog, Marios Papas, Fabrice Rousselle, and Jan Novák. 2021. NeRF-Tex: Neural reflectance field textures. In *Eurographics Symposium on Rendering*. The Eurographics Association.

Steve Bako, Mark Meyer, Tony DeRose, and Pradeep Sen. 2019. Offline deep importance sampling for Monte Carlo path tracing. *Comp. Graph. Forum* (2019).

Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony DeRose, and Fabrice Rousselle. 2017. Kernel-predicting convolutional networks for denoising Monte Carlo renderings. *ACM Trans. Graph.* 36, 4 (July 2017).

Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. 2021. Mip-NeRF: A multiscale representation for anti-aliasing neural radiance fields. *ICCV* (2021).

Laurent Belcour, Ling-Qi Yan, Ravi Ramamoorthi, and Derek Nowrouzezahrai. 2017. Antialiasing complex global illumination effects in path-space. *ACM Trans. Graph.* 36, 1 (2017).

Mojtaba Bemana, Karol Myszkowski, Hans-Peter Seidel, and Tobias Ritschel. 2019. Neural view-interpolation for sparse lightfield video. *arXiv preprint arXiv:1910.13921* (2019).

Benedikt Bitterli, Srinath Ravichandran, Thomas Müller, Magnus Wrenninge, Jan Novák, Steve Marschner, and Wojciech Jarosz. 2018. A radiative transfer framework for non-exponential media. *ACM Trans. Graph.* 37, 6, Article 225 (Dec. 2018), 17 pages.

Eric Bruneton and Fabrice Neyret. 2011. A survey of non-linear pre-filtering methods for efficient and accurate surface shading. *IEEE Trans. Vis. Comp. Graph.* (2011).

Brent Burley and Walt Disney Animation Studios. 2012. Physically-based shading at Disney. In *ACM SIGGRAPH 2012 Talks*.

Chakravarty R. A. Chaitanya, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive reconstruction of noisy Monte Carlo image sequences using a recurrent autoencoder. *ACM Trans. Graph.* (July 2017).

Subrahmanyan Chandrasekhar. 1960. *Radiative transfer*. Dover publications, New York.

Anpei Chen, Minye Wu, Yingliang Zhang, Nianyi Li, Jie Lu, Shenghua Gao, and Jingyi Yu. 2018. Deep surface light fields. *Proc. ACM Comp. Graph. Interactive Techniques* 1, 1 (2018).

Xavier Chermain, Frédéric Claux, and Stéphane Mérillou. 2019a. A microfacet-based BRDF for the accurate and efficient rendering of high-definition specular normal maps. *The Visual Computer* (2019).

Xavier Chermain, Frédéric Claux, and Stéphane Mérillou. 2019b. Glint rendering based on a multiple-scattering patch BRDF. *Comp. Graph. Forum* 38, 4 (2019).

Jonathan Cohen, Dinesh Manocha, and Marc Olano. 1997. Simplifying polygonal models using successive mappings. In *Proceedings. Visualization'97 (Cat. No. 97CB36155)*.

Jonathan Cohen, Marc Olano, and Dinesh Manocha. 1998. Appearance-preserving simplification. In *Proc. ACM Comp. Graph. Interactive Techniques*.

Jonathan David Cohen. 1999. Appearance-preserving simplification of polygonal models. (1999).

Robert L Cook, John Halstead, Maxwell Planck, and David Ryu. 2007. Stochastic simplification of aggregate detail. In *ACM Trans. Graph.*, Vol. 26.

Robert L Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed ray tracing. In *Proc. ACM Comp. Graph. Interactive Techniques*.

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. 2011. Interactive indirect illumination using voxel cone tracing. *Comp. Graph. Forum* 30, 7.

Cyril Crassin, Chris Wyman, Morgan McGuire, and Aaron Lefohn. 2018. Correlation-aware semi-analytic visibility for antialiased rendering. In *Proceedings of the Conference on High-Performance Graphics*.

Yue Dong. 2019. Deep appearance modeling: A survey. *Visual Informatics* (2019).

Zhao Dong, Bruce Walter, Steve Marschner, and Donald P Greenberg. 2015. Predicting appearance from measured microgeometry of metal surfaces. *ACM Trans. Graph.* 35, 1 (2015).

Jonathan Dupuy and Eric Heitz. 2016. Additional progress towards the unification of microfacet and microflake theories.

Luis E. Gamboa, Jean-Philippe Guertin, and Derek Nowrouzezahrai. 2018. Scalable appearance filtering for complex lighting effects. *ACM Trans. Graph.* 37, 6 (Dec. 2018).

Liangsheng Ge, Beibei Wang, Lu Wang, and Nicolas Holzschuch. 2018. A compact representation for multiple scattering in participating media using neural networks. In *ACM SIGGRAPH 2018 Talks*. Vancouver, Canada.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Int. Conf. Art. Intelligence and Statistics*.

Enrico Gobbetti and Fabio Marton. 2005. Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Trans. Graph.* 24, 3 (2005).

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. The MIT Press.

Jonathan Granskog, Fabrice Rousselle, Marios Papas, and Jan Novák. 2020. Compositional neural scene representations for shading inference. *ACM Trans. Graph. (Proc. SIGGRAPH)* 39, 4 (July 2020).

Jonathan Granskog, Till N. Schnabel, Fabrice Rousselle, and Jan Novák. 2021. Neural scene graph rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 40, 4 (Aug. 2021).

Jie Guo, Yanjun Chen, Bingyang Hu, Ling-Qi Yan, Yanwen Guo, and Yuntao Liu. 2019. Fractional Gaussian fields for modeling and rendering of spatially-correlated media. *ACM Trans. Graph. (Proc. SIGGRAPH)* 38, 4 (2019).

Eric Heitz, Jonathan Dupuy, Cyril Crassin, and Carsten Dachsbacher. 2015. The SGGX microflake distribution. *ACM Trans. Graph.* 34, 4 (2015).

Eric Heitz and Fabrice Neyret. 2012. Representing appearance and pre-filtering subpixel data in sparse voxel octrees. In *Proc. High Perf. Graph.*

Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science* 313, 5786 (2006).

Hugues Hoppe. 1996. Progressive meshes. In *Proc. ACM Comp. Graph. Interactive Techniques*.

Wenzel Jakob. 2010. Mitsuba renderer. http://www.mitsuba-renderer.org.

Wenzel Jakob, Adam Arbree, Jonathan T. Moon, Kavita Bala, and Steve Marschner. 2010. A radiative transfer framework for rendering materials with anisotropic structure. *ACM Trans. Graph. (Proc. SIGGRAPH)*, Article 53, 13 pages.

Wenzel Jakob, Miloš Hašan, Ling-Qi Yan, Jason Lawrence, Ravi Ramamoorthi, and Steve Marschner. 2014. Discrete stochastic microfacet models. *ACM Trans. Graph.* 33, 4 (2014).

Adrian Jarabo, Carlos Aliaga, and Diego Gutierrez. 2018. A radiative transfer framework for spatially-correlated materials. *ACM Trans. Graph.* 37, 4 (2018).

Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. 2015. A machine learning approach for filtering Monte Carlo noise. *ACM Trans. Graph.* 34, 4 (July 2015).

Nima Khademi Kalantari and Ravi Ramamoorthi. 2017. Deep high dynamic range imaging of dynamic scenes. *ACM Trans. Graph.* 36, 4 (2017).

Nima Khademi Kalantari and Ravi Ramamoorthi. 2019. Deep HDR video from sequences with alternating exposures. In *Comp. Graph. Forum*, Vol. 38.

Nima Khademi Kalantari, Ting-Chun Wang, and Ravi Ramamoorthi. 2016. Learning-based view synthesis for light field cameras. *ACM Trans. Graph.* 35, 6, Article 193 (Nov. 2016), 10 pages.

Simon Kallweit, Thomas Müller, Brian McWilliams, Markus Gross, and Jan Novák. 2017. Deep scattering: Rendering atmospheric clouds with radiance-predicting neural networks. *ACM Trans. Graph.* 36, 6, Article 231 (Nov. 2017), 11 pages.

Kaizhang Kang, Zimin Chen, Jiaping Wang, Kun Zhou, and Hongzhi Wu. 2018. Efficient reflectance capture using an autoencoder. (2018).

Anton S Kaplanyan, Stephen Hill, Anjul Patney, and Aaron E Lefohn. 2016. Filtering distributions of normals for shading antialiasing. *Proc. High Perf. Graph.*

Markus Kettunen, Eugene d'Eon, Jacopo Pantaleoni, and Jan Novák. 2021. An unbiased ray-marching transmittance estimator. *ACM Trans. Graph. (Proc. SIGGRAPH)* 40, 4, Article 137 (Aug. 2021).

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR* abs/1412.6980 (2014).

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

Alexandr Kuznetsov, Miloš Hašan, Zexiang Xu, Ling-Qi Yan, Bruce Walter, Nima Khademi Kalantari, Steve Marschner, and Ravi Ramamoorthi. 2019. Learning generative models for rendering specular microgeometry. *ACM Trans. Graph.* 38, 6, Article 225 (Nov. 2019), 14 pages.

Alexandr Kuznetsov, Krishna Mullia, Zexiang Xu, Miloš Hašan, and Ravi Ramamoorthi. 2021. NeuMIP: Multi-resolution neural materials. *ACM Trans. Graph. (Proc. SIGGRAPH)* 40, 4, Article 175 (July 2021), 13 pages.

Dylan Lacewell, Brent Burley, Solomon Boulos, and Peter Shirley. 2008. Raytracing prefiltered occlusion for aggregate geometry. In *2008 IEEE Symposium on Interactive Ray Tracing*.

Samuli Laine and Tero Karras. 2010. Efficient sparse voxel octrees. *IEEE Trans. Vis. Comp. Graph.* 17, 8 (2010).

Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. 2018. Noise2Noise: Learning image restoration without

clean data. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 80. PMLR.
Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural sparse voxel fields. *NeurIPS* (2020).
Tom Lokovic and Eric Veach. 2000. Deep shadow maps. In *ACM Trans. Graph. (Proc. SIGGRAPH)*.
Stephen Lombardi, Jason Saragih, Tomas Simon, and Yaser Sheikh. 2018. Deep appearance models for face rendering. *ACM Trans. Graph.* 37, 4 (2018).
Guillaume Loubet. 2018. *Efficient models for representing sub-pixel appearances.* Université Grenoble Alpes PhD thesis.
Guillaume Loubet and Fabrice Neyret. 2017. Hybrid mesh-volume LoDs for all-scale pre-filtering of complex 3D assets. In *Comp. Graph. Forum*, Vol. 36.
Guillaume Loubet and Fabrice Neyret. 2018. A new microflake model with microscopic self-shadowing for accurate volume downsampling. In *Comp. Graph. Forum*, Vol. 37.
David Luebke and Carl Erikson. 2006. *View-dependent simplification of arbitrary polygonal environments.* Technical Report. North Carolina Univ at Chapel Hill.
David Luebke, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. 2003. *Level of detail for 3D graphics.* Morgan Kaufmann.
David P Luebke. 2001. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications* 21, 3 (2001).
Nelson Max, Brett Keating, Curtis Mobley, and En-Hua Wu. 1997. Plane-parallel radiance transport for global illumination in vegetation. In *Rendering Techniques 97*. Springer.
Maxim Maximov, Laura Leal-Taixe, Mario Fritz, and Tobias Ritschel. 2019. Deep appearance maps. In *Proceedings of the IEEE International Conference on Computer Vision*.
Johannes Meng, Marios Papas, Ralf Habel, Carsten Dachsbacher, Steve Marschner, Markus Gross, and Wojciech Jarosz. 2015. Multi-scale modeling and rendering of granular materials. *ACM Trans. Graph. (Proc. SIGGRAPH)* 34, 4 (July 2015).
Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. 2001. Interactive rendering of trees with shading and shadows. In *12th Eurographics Workshop on Rendering Techniques*. Springer.
Ehsan Miandji, Joel Kronander, and Jonas Unger. 2013. Learning based compression of surface light fields for real-time rendering of global illumination scenes. In *SIGGRAPH Asia 2013 Technical Briefs*.
Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. 2019. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Trans. Graph.* (2019).
Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing scenes as neural radiance fields for view synthesis. In *ECCV*.
Jonathan T Moon, Bruce Walter, and Steve Marschner. 2008. Efficient multiple scattering in hair using spherical harmonics. In *ACM Trans. Graph.*, Vol. 27.
Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.* 41, 4, Article 102 (July 2022), 15 pages.
Thomas Müller, Marios Papas, Markus Gross, Wojciech Jarosz, and Jan Novák. 2016. Efficient rendering of heterogeneous polydisperse granular media. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 35, 6 (Dec. 2016).
Tsukasa Noma. 1995. Bridging between surface rendering and volume rendering for multi-resolution display. In *Eurographics Workshop on Rendering Techniques*. Springer.
Jan Novák, Iliyan Georgiev, Johannes Hanika, and Wojciech Jarosz. 2018. Monte Carlo methods for volumetric light transport simulation. *Computer Graphics Forum (Proceedings of Eurographics - State of the Art Reports)* 37, 2 (May 2018).
Sean Palmer, Eric Maurer, and Mark Adams. 2014. Using sparse voxel octrees in a level-of-detail pipeline for Rio 2. In *ACM SIGGRAPH 2014 Talks*.
Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. 2010. OptiX: A general purpose ray tracing engine. In *ACM Trans. Graph.*, Vol. 29.
Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation.* Morgan Kaufmann.
Gilles Rainer, Wenzel Jakob, Abhijeet Ghosh, and Tim Weyrich. 2019. Neural BTF compression and interpolation. In *Comp. Graph. Forum*, Vol. 38.
Boris Raymond, Gael Guennebaud, and Pascal Barla. 2016. Multi-scale rendering of scratched materials using a structured SV-BRDF model. *ACM Trans. Graph.* 35, 4 (2016).
Peiran Ren, Yue Dong, Stephen Lin, Xin Tong, and Baining Guo. 2015. Image based relighting using neural networks. *ACM Trans. Graph.* 34, 4 (2015).
Peiran Ren, Jinpeng Wang, Minmin Gong, Stephen Lin, Xin Tong, and Baining Guo. 2013. Global illumination with radiance regression functions. *ACM Trans. Graph.* 32 (July 2013).
Pỳnar Satỳlmỳs, Thomas Bashford-Rogers, Alan Chalmers, and Kurt Debattista. 2017. A machine-learning-driven sky model. *IEEE computer graphics and applications* 37, 1 (2017).
Manolis Savva, Jitendra Malik, Devi Parikh, Dhruv Batra, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, and Vladlen Koltun. 2019. Habitat: A platform for embodied AI research. In *IEEE/CVF International Conference on Computer Vision*.
K Schroder, Reinhard Klein, and Arno Zinke. 2011. A volumetric approach to predictive rendering of fabrics. In *Comp. Graph. Forum*, Vol. 30.
Pratul P. Srinivasan, Boyang Deng, Xiuming Zhang, Matthew Tancik, Ben Mildenhall, and Jonathan T. Barron. 2021. NeRV: Neural reflectance and visibility fields for relighting and view synthesis. In *CVPR*.
Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. 2021. Neural geometric level of detail: Real-time rendering with implicit 3D shapes. (2021).
Ping Tan, Stephen Lin, Long Quan, Baining Guo, and Harry Shum. 2008. Filtering and rendering of resolution-dependent reflectance models. *IEEE Trans. Vis. Comp. Graph.* 14, 2 (2008).
Ping Tan, Stephen Lin, Long Quan, Baining Guo, and Heung-Yeung Shum. 2005. Multiresolution reflectance filtering.
Delio Vicini, Wenzel Jakob, and Anton Kaplanyan. 2021. A non-exponential transmittance model for volumetric scene representations. *Transactions on Graphics (Proceedings of SIGGRAPH)* 40, 4 (Aug. 2021).
Delio Vicini, Vladlen Koltun, and Wenzel Jakob. 2019. A learned shape-adaptive subsurface scattering model. *ACM Trans. Graph. (Proc. SIGGRAPH)* 38, 4 (July 2019).
Thijs Vogels, Fabrice Rousselle, Brian McWilliams, Gerhard Röthlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák. 2018. Denoising with kernel prediction and asymmetric loss functions. *ACM Trans. Graph.* 37, 4, Article 124 (2018).
Ting-Chun Wang, Jun-Yan Zhu, Nima Khademi Kalantari, Alexei A. Efros, and Ravi Ramamoorthi. 2017. Light field video capture using a learning-based hybrid imaging system. *ACM Trans. Graph. (Proc. SIGGRAPH)* 36, 4 (2017).
Zhou Wang, Alan C Bovik, Hamid R Sheikh, Eero P Simoncelli, et al. 2004. Image quality assessment: From error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004).
Lance Williams. 1978. Casting curved shadows on curved surfaces. In *ACM Trans. Graph. (Proc. SIGGRAPH)*. 5.
Lance Williams. 1983. Pyramidal parametrics. In *Proc. ACM Comp. Graph. Interactive Techniques*.
Nathaniel Williams, David Luebke, Jonathan D Cohen, Michael Kelley, and Brenden Schubert. 2003. Perceptually guided simplification of lit, textured meshes. In *Proceedings of Symposium on Interactive 3D Graphics*.
Lifan Wu, Shuang Zhao, Ling-Qi Yan, and Ravi Ramamoorthi. 2019. Accurate appearance preserving prefiltering for rendering displacement-mapped surfaces. *ACM Trans. Graph. (Proc. SIGGRAPH)* 38, 4 (2019).
Julie C. Xia, Jihad El-Sana, and Amitabh Varshney. 1997. Adaptive real-time level-of-detail based rendering for polygonal models. *IEEE Trans. Vis. Comp. Graph.* 3, 2 (1997).
Chao Xu, Rui Wang, Shuang Zhao, and Hujun Bao. 2017. Real-time linear BRDF mip-mapping. In *Comp. Graph. Forum*, Vol. 36.
Zexiang Xu, Kalyan Sunkavalli, Sunil Hadap, and Ravi Ramamoorthi. 2018. Deep image-based relighting from optimal sparse samples. *ACM Trans. Graph.* 37, 4 (2018).
Ling-Qi Yan, Miloš Hašan, Wenzel Jakob, Jason Lawrence, Steve Marschner, and Ravi Ramamoorthi. 2014. Rendering glints on high-resolution normal-mapped specular surfaces. *ACM Trans. Graph.* 33, 4 (2014).
Ling-Qi Yan, Miloš Hašan, Steve Marschner, and Ravi Ramamoorthi. 2016. Position-normal distributions for efficient rendering of specular microstructure. *ACM Trans. Graph.* 35, 4 (2016).
Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. 2006. R-LODs: Fast LOD-based ray tracing of massive models. *The Visual Computer* 22, 9-11 (2006).
Shuang Zhao, Miloš Hašan, Ravi Ramamoorthi, and Kavita Bala. 2013. Modular flux transfer: Efficient rendering of high-resolution volumes with repeated structures. *ACM Trans. Graph.* 32, 4 (2013).
Shuang Zhao, Wenzel Jakob, Steve Marschner, and Kavita Bala. 2011. Building volumetric appearance models of fabric using micro CT imaging. In *ACM Trans. Graph.*, Vol. 30.
Shuang Zhao, Wenzel Jakob, Steve Marschner, and Kavita Bala. 2012. Structure-aware synthesis for predictive woven fabric appearance. *ACM Trans. Graph.* 31, 4 (2012).
Shuang Zhao, Lifan Wu, Frédo Durand, and Ravi Ramamoorthi. 2016. Downsampling scattering parameters for rendering anisotropic media. *ACM Trans. Graph.* 35, 6 (2016).
Junqiu Zhu, Yaoyi Bai, Zilin Xu, Steve Bako, Edgar Velázquez-Armendáriz, Lu Wang, Pradeep Sen, Milos Hasan, and Ling-Qi Yan. 2021. Neural complex luminaires: Representation and rendering.
Tobias Zirr and Anton S. Kaplanyan. 2016. Real-time rendering of procedural multiscale materials. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '16)*. ACM, New York, NY, USA, 10.
Károly Zsolnai-Fehér, Peter Wonka, and Michael Wimmer. 2018. Gaussian material synthesis. *ACM Trans. Graph.* 37, 4 (2018).

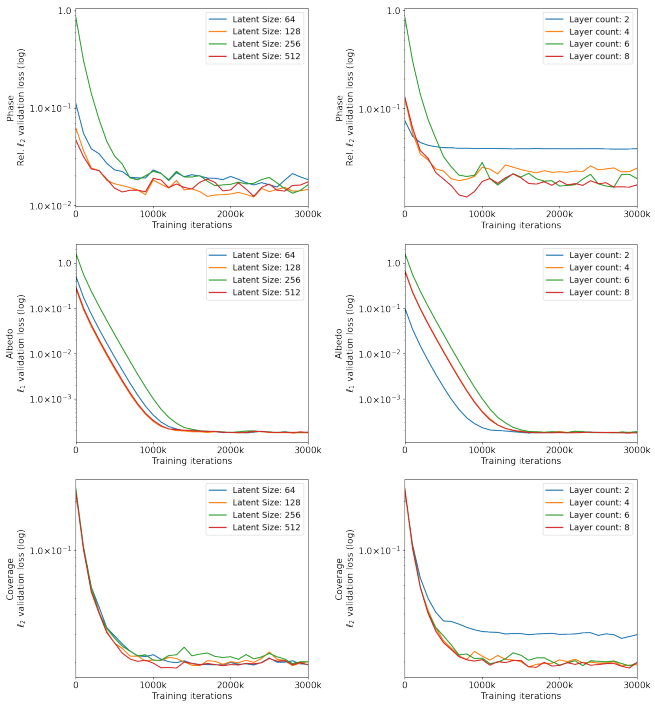


Fig. 18. Additional ablation results plotting validation errors for the other decoders not discussed in the main text. The behavior of the phase network (first row) is similar to the phase-slice network (see Fig. 12) where increasing the latent size results in slightly better performance, yet gains become marginal for larger sizes (left plot). Moreover, increasing the number of network layers also improves performance up to a point (right plot). Meanwhile, the coverage and albedo sub-networks have a relatively lower threshold for latent sizes and layer counts as they do not improve their performance after reaching a certain size. To ensure all cases are handled robustly with a consistent architecture, we use 256 floats for the latent vector and 6 layers in the decoder, as this performs well in all cases and for all decoders.

A ADDITIONAL ABLATION ANALYSIS

Fig. 18 shows additional results from the two ablation experiments for the three remaining decoders (the phase-slice network plots were shown in Fig. 12): phase, albedo, and coverage. The first row shows the results of these two ablations for the phase network. We see a similar behavior to that of the phase-slice results in that there are only marginal gains in performance for the converged networks with latent size of 256 floats and 6 layers, respectively. Next, the coverage network (second row) has roughly the same converged result for all latent sizes and for networks with 4 or more layers. Finally, the albedo network in the last row, performs well at all latent and network sizes. For simplicity and a consistent architecture for all sub-networks, we use 256 floats for each latent component and 6 layers for each decoder. Note, these experiments suggest there is potential to further minimize memory footprint by using albedo and coverage latent sizes of 64 or 128 floats in future implementations.

B PSEUDOCODE

In Algorithm 1, we present pseudocode for various components of our system to facilitate comprehension. Our full code is available at our paper page at to allow for comparisons and future work.

ALGORITHM 1: Deep Appearance Prefiltering (DAP) Algorithm

Input: Scene to prefilter, level of detail (LoD) scale s, scene for final rendering (e.g., camera and lighting parameters)
Output: Rendered image, trained network weights $\mathcal{W}$, sparse voxel octree V, latent voxel encodings $\mathcal{L}$

```
/* SVO creation (Sec. 4.1) */
V = Voxelize(s)

/* Data generation (Sec. 4.2) and training (Sec. 5.2) */
forall training iterations do
    do in parallel
        D = GenerateVoxelData(V) // Includes brute-force rendering
        W = TrainNetwork(D, W)
    end
end

/* Prerendering (Sec. 5.3) */
L = GenerateLatentVectors(D, W) // Save each voxel's latent vector
S = GenerateShadowMaps(V, L, W)
LoadFinalSceneParameters( )

/* Runtime (Sec. 5.3) */
forall pixels I in final image L do

    /* Return list of voxels the beam touches in front to back order */
    V_I = TraceBeam(I, V)

    Λ = InitializeBeamCoverage( )
    L_I = 0
    T = ComputeTransmittance(Λ)
    while T < 1 and Count(V_I) > 0 do
        ω0, ωi = GetOutgoingAndIncomingDirections( )   // Sec. 4.1
        r = EncodeQueryAndConcatenate(v, ω0, ωi)
        ρ, γ, α = EvaluateDecoders(r)
        F = ComputePhase(ρ, γ)                         // Eq. 17
        Λ = UpdateWavefrontAndCoverage(α)              // Eq. 16
        T = ComputeTransmittance(Λ)                    // Eq. 15
        S = ComputeScatteringTerm(F, T, S)             // Eq. 10
        G = ComputePrefilteredEmission( )              // Eq. 14
        E = ComputeEmissionTerm(G, T)                  // Eq. 13
        B = ComputeBoundaryTerm( )                     // Eq. 5
        L_I = UpdateColor(S, E, B)                     // Eq. 4
    end
end
return L
```