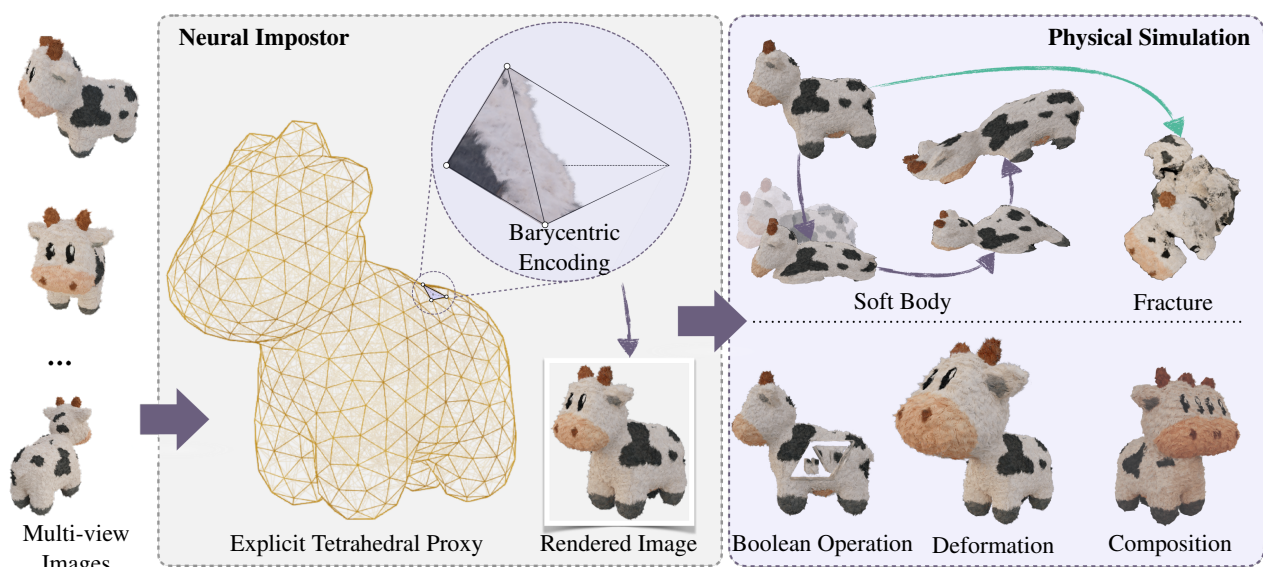


# Neural Impostor: Editing Neural Radiance Fields with Explicit Shape Manipulation

Ruiyang Liu<sup>†1,2</sup> , Jinxu Xiang<sup>†1</sup> , Bowen Zhao<sup>1</sup> , Ran Zhang<sup>\*1</sup> , Jingyi Yu<sup>2</sup> and Changxi Zheng<sup>1</sup> 

<sup>1</sup>Tencent Pixel Lab, <sup>2</sup>ShanghaiTech University, <sup>†</sup>Equally contributed co-first authors



**Figure 1:** The creation and application of Neural Impostor. Neural Impostor takes multi-view captured images as input and build a coarse geometry proxy with local radiance fields encoded within the barycentric coordinate. With the help of the coarse geometry proxy, Neural Impostor empowers novice users to conduct vivid physical simulations and perform versatile editing operations on radiance fields, including Boolean operation, local deformation and composition.

## Abstract

Neural Radiance Fields (NeRF) have significantly advanced the generation of highly realistic and expressive 3D scenes. However, the task of editing NeRF, particularly in terms of geometry modification, poses a significant challenge. This issue has obstructed NeRF's wider adoption across various applications. To tackle the problem of efficiently editing neural implicit fields, we introduce **Neural Impostor**, a hybrid representation incorporating an explicit tetrahedral mesh alongside a multigrid implicit field designated for each tetrahedron within the explicit mesh. Our framework bridges the explicit shape manipulation and the geometric editing of implicit fields by utilizing multigrid barycentric coordinate encoding, thus offering a pragmatic solution to deform, composite, and generate neural implicit fields while maintaining a complex volumetric appearance. Furthermore, we propose a comprehensive pipeline for editing neural implicit fields based on a set of explicit geometric editing operations. We show the robustness and adaptability of our system through diverse examples and experiments, including the editing of both synthetic objects and real captured data. Finally, we demonstrate the authoring process of a hybrid synthetic-captured object utilizing a variety of editing operations, underlining the transformative potential of **Neural Impostor** in the field of 3D content creation and manipulation.

## CCS Concepts

• **Computing methodologies** → **Rendering; Appearance and texture representations; Image-based rendering;**

## 1. Introduction

The Neural Radiance Field (NeRF) serves as a groundbreaking neural implicit representation that revolutionizes novel view synthesis processes. It allows users to rebuild a scene directly from multi-view photographs using a standard camera through an end-to-end training process. This significant advancement has streamlined complex 3D reconstruction processes, signifying a considerable paradigm shift in the field. However, the transformation of 3D scene data into a higher-dimensional feature space implicitly stored within neural networks presents notable challenges, including incompatibility with most existing modeling and editing software. As a result, editing neural radiance fields, especially in terms of geometry modification, remains a considerable challenge.

Recent research in editing neural radiance fields have been split primarily into two categories: implicit and explicit mapping. Implicit methods, though widely adopted, are often limited due to their dependence on high-dimensional feature space interpolation, restricted to accommodate only simple edits. On the other hand, explicit methods that employ dedicated proxies for direct manipulation of the radiance field provide enhanced editing capabilities, albeit with inherent challenges such as modeling limitations and increased computational complexity. For example, while *triangle meshes based methods* [CFHT22, CJH\*22, WXB\*23] favored for their compatibility with existing 3D modeling tools and high-speed rendering capabilities, they struggle with modeling volumetric appearances like furs and hairs. The employment of *point clouds* [XXP\*22, CLW23] enables volumetric modeling but lacks geometry constraints, complicating contiguous authoring. The utilization of *volumetric cages* [GKE\*22, YSL\*22, XH22, PYL\*22, JKK\*23], on the other hand, offers topology constraints for stable authoring and benefits in physical simulation. Despite these advantages, their rendering speed is often hampered by complex point-in-tetrahedron queries.

In response to these challenges, we introduce Neural Impostor, a hybrid model that addresses the primary considerations necessary for creating a fully editable neural radiance field that compatible with common geometric editing pipeline. It is also capable of high fidelity reconstruction and optimized for real-time rendering.

The Neural Impostor segments the modeling space of NeRF with explicit tetrahedral meshes, transitioning from Cartesian space encoding to local barycentric encoding within the tetrahedral proxy. Each tetrahedron is assigned a distinct Multi-grid Neural Radiance Field for detailed local depictions, enhancing scalability in the modeling and rendering process. Our barycentric encoding scheme maintains visual uniformity during physical simulations driven by vertex transformations due to its harmonic nature around  $n + 1$  control points. Furthermore, Neural Impostor's compatibility with existing 3D animation software, such as Houdini, offers distinct advantages in animations and games. Drawing inspiration from space partitioning concepts, our model facilitates high-quality, real-time rendering and is crucial for modeling, rendering, and simulating complex volumetric objects like plush toys.

In essence, the Neural Impostor framework brings together the advantages of explicit mesh editing with the volumetric appearance of implicit fields, thereby offering a powerful and versatile tool for

manipulating Neural Radiance Fields. We highlight our key contributions as follows:

- The design and implementation of *Neural Impostor*, a novel hybrid representation that bolsters the editing capabilities of Neural Radiance Fields by adeptly leveraging the benefits of both explicit meshes and implicit fields.
- The deployment of a *Robust Hash Encoding* method combined with *Sophisticated Spatial Sampling* techniques. By utilizing barycentric coordinates encoding of impostor geometric elements, we ensure high-quality reconstruction and rendering during animations.
- The introduction of a diverse suite of *Geometric Operations* underpinned by the *Neural Impostor*. This approach enables the seamless transfer of local geometry editing operations, such as shape morphing, remeshing and boolean operations, from contemporary 3D modeling software into Neural Radiance Fields through local retraining.
- The creation of an efficient, hardware-accelerated rendering algorithm, which streamlines the traditionally laborious barycentric coordinate mapping process. This algorithm fulfills real-time requirements, achieving an impressive performance of approximately 30 FPS for  $1080 \times 1080$  rendering.

In the forthcoming sections, we will first delve into the modeling of *Neural Impostor* (Section 3)). Alongside this, we detail a set of editing operations for Neural Impostor that accommodate edits at the animation, geometric, and appearance levels (Section 4). We then demonstrate the modeling and editing capabilities of *Neural Impostor* through systematic testing on the *nerf-synthetic* dataset, as well as on a few custom-built *plush toy* models with complex volumetric appearances(Section 5). Additionally, we will present operations such as shape morphing, soft body deformation, fracture, and composition made possible by *Neural Impostor*, thereby effectively underlining the utility of *Neural Impostor* in real-world content creation workflows. This will be exemplified through a practical demonstration of constructing a snowman toy by editing multiple pre-trained Neural Impostors (Section 6).

## 2. Related Work

Neural Impostor builds upon the foundation of Neural Radiance Fields (NeRF) [MST\*20], which presents a method for representing a 3D scene using a scalar density field  $\sigma$  and a view-dependent color field  $\mathbf{c}$ . These fields enable NeRF to generate high-quality renderings from novel viewpoints using volume rendering techniques like [Max95]. The core idea of NeRF is to employ an implicit representation due to the inherent complexity of capturing both 3D geometry and appearance. This representation involves encoding the spatial information, which typically consists of positional parameter  $\mathbf{x}$  and directional parameter  $\mathbf{d}$ , into a learnable function  $f_\theta : (\mathbf{x}, \mathbf{d}) \mapsto (\mathbf{c}, \sigma)$  with adjustable parameters  $\theta$ . With a space sampler  $\mathcal{S}$ , the novel view appearance can be rendered by integrating  $(\mathbf{c}, \sigma)$  along the sampling ray  $\mathbf{r}$ :

$$\mathcal{C}(\mathbf{r}) = \sum_{i=1}^N T_i (1 - \exp(-\sigma_i \delta_i)) c_i, \text{ where } T_i = \exp\left(-\sum_{j=1}^i \sigma_j \delta_j\right) \quad (1)$$

where  $\delta_i$  denotes the sampling intervals along the ray  $\mathbf{r}$  and the transmittance  $T_i$  is the accumulated density along the ray. With multi-view images as supervision, the learnable function  $f_\theta$  can be trained by minimizing the reconstruction loss between the rendered image and the ground truth image.

Recent surveys on NeRF [TTM\*22] and [TTM\*22] presented a comprehensive review of every process in computing NeRF. This paper primarily concentrates on the geometric editing of NeRF among all the stages involved in its construction and manipulation. Rather than providing an exhaustive overview of all existing NeRF research, we specifically focus on reviewing the approach that directly pertains to our topic.

### 2.1. Spatial Encoding in Neural Radiance fields

When it comes to geometric editing of 3D content, the core of the problem is to encode and manipulate the spatial information of the scene. The spatial encoding process of NeRF is to map the low-dimensional smooth inputs such as position and direction to a high-dimensional feature space that can be trained to encode high-frequency details. Initially, researchers use a fully connected MLP with positional encoding [MST\*20] to encode the spatial information of the entire scene, which synthesizes realistic novel views with fine details, but also suffers from high computational cost when querying the large neural network for every sample. To enhance the training efficiency, hybrid spatial representations utilize the sparsity of spatial data, factorize the scene into explicitly stored features and implicit neural functions. Depending on the underlying structure, we categorize spatial encoding methods into grid-based and mesh-based structures. Grid-based encoding schemes, here we refer to pre-constructed regular spatial structures, such as sparse voxel grids [LGL\*20, HSM\*21], octree grids [YLT\*21, YFKT\*21], triplanar structures [CLC\*21], codebooks [TET\*22], tensorial decomposition [CXG\*22] and multi-grid hash encoding [MESK22]. The grid-based approaches provide efficient querying of features stored in the grid, which greatly improves the training and rendering speed. Mesh-based encoding, on the other hand, uses a mesh-like structure with more flexible topologies, such as point cloud [XXP\*22] (point clouds can be seen as unconnected mesh vertices), triangle soup [CFHT22], manifold mesh [CJH\*22, WXB\*23], duplex mesh [WRB\*23] and tetrahedral mesh [GCX\*20]. Mesh-based encoding provides additional flexibility in manipulating implicit fields; however, its irregular structure necessitates a robust adaptive mesh generator. Furthermore, when storing all the features on explicit meshes, mesh-based encoding often lacks volumetric appearances.

In this paper, we aim to achieve both the efficiency of grid-based encoding and the flexibility of mesh-based encoding, therefore enables the geometric editing of NeRF while maintaining the high-quality volumetric rendering. In the proposed method, we utilize a tetrahedral based proxy, and bounds the explicit mesh and implicit field with a generalized barycentric encoding method.

### 2.2. Proxy-based manipulation of 3D models

Using explicit proxies to manipulate 3D models is a common practice in computer graphics. Proxies usually retain a lower dimen-

sional representation of the original model, which serve as a model reduction process and can be used to accelerate the rendering process or provide a more intuitive interface for users to manipulate the model. For example, in geometric modeling and shape manipulation, proxies reduce the complexity of original models, therefore enable structure preserving shape manipulation [ZFCO\*11], shape collection synthesis [XZCOC12] and image-based geometric modeling [XZZ\*11]. Bounding volumes, which are one of the simplest forms of proxies, are widely used in collision detection [GLM96, KHM\*98], ray tracing [MOB\*21] and shape approximation [CB17]. Proxy-based methods are also used in manipulating NeRF, one of the commonly used proxies is the tetrahedral mesh, which is used to approximate the geometry of the scene. By mapping the samples in the deformed tetrahedral mesh to the canonical space [GKE\*22, XH22, YSL\*22, PYL\*22], one can manipulate the neural implicit field with geometric operations on the explicit tetrahedral mesh. In this paper, we follow the general idea of using tetrahedral mesh as proxy, but instead of bending sampling rays to the canonical space, we propose a generalized barycentric encoding method efficiently store the implicit field in the tetrahedral mesh, which allows more flexible operations, please see Sec. 4. Worth mentioning that there is a special type of proxy called impostor [Jes05], which originally is a 2D image that approximates the appearance of a 3D model from a specific viewpoint by interpolating pre-rendered multi-view appearances. The concept of impostor is later extended to 3D models, such as [JW02, YD08], and is widely used in real-time rendering of large-scale scenes. In this paper, we introduce the term, "Neural Impostor," which is derived from the concept of Impostor. Our proposed method entails the creation of an editable 3D proxy that approximates the volumetric appearance of a 3D model from arbitrary viewpoints, just like traditional impostor did in real-time rendering.

### 2.3. Editing Neural Radiance Fields

Recently, Editing Neural Radiance Fields has seen significant progress in terms of reconstructing and modifying various aspects of appearance, such as relighting, controlling shapes, and altering colors or palettes of objects [LZZ\*21, KLB\*22]. Some techniques [YSL\*22, GKE\*22, JKK\*23] have enabled modification of scene parts and their respective locations, thus providing more control over the rendered scene. However, editing the dynamics of moving objects, especially with topological changes, has been a major challenge. These changes can lead to motion discontinuities, causing noticeable artifacts if not properly modeled [YSL\*22]. [KYK\*21] has tried to mitigate these challenges using manual supervision, but its capabilities are limited to one-dimensional editing per scene part, necessitating user annotations for supervision. In contrast, EditableNeRF [ZcLX22] uses an image sequence from a single camera to train a network, modeling topologically varying dynamics via surface key points. Users can edit the scene by manipulating these key points, enabling more intuitive multi-dimensional editing (up to 3D). NeuralEditor [CLW23] is another novel approach that leverages the explicit point cloud representation underlying NeRFs for shape editing tasks. It employs a unique rendering scheme based on deterministic integration within K-D tree-guided density-adaptive voxels, which results in high-quality rendering and precise point clouds. Despite these advancements, edit-

ing capabilities within a computed NeRF scene remain relatively limited, especially when compared to traditional CGI workflows. Recent progress in pre-trained large-scale models has enabled rapid progress in a brand new instruction or prompt based creation and interaction with digital contents. For instance, Instruct-NeRF2NeRF [HTE\*23] leverages an image-conditioned diffusion model to iteratively edit input images while simultaneously optimizing the underlying radiance field of the 3D scene, resulting in a realistic 3D scene that adheres to the provided editing instructions. [MPE\*23] offering genuine, personalized, temporal consistent 3D avatar edit by combining text-to-image diffusion model with neural radiance field (NeRF) through an innovative sampling strategy to incorporate multiple keyframes representing different camera viewpoints and timestamps into a single diffusion model. The edits are made in a canonical space and propagated to remaining time steps through a pretrained deformation network.

In this study, we revisit the issue of geometry editing using a new localized retraining technique that allows for the flawless integration of geometry manipulation tasks from modern 3D modeling software into Neural Radiance Fields. This improves NeRF's suitability as a representation for User Generated Content (UGC) and AI Generated Content (AIGC).

### 3. Neural Impostor: A Hybrid Explicit-Implicit Neural Field

*Neural Impostor* uses barycentric coordinates defined by the tetrahedron mesh as the encoding space. Compared to the Cartesian coordinate-based encoding in the neural radiance field, barycentric coordinates have inherent advantages. On one hand, barycentric coordinates are symmetrical to the  $n + 1$  vertices of the  $n$ -simplex that defines them. During the vertex-based deformation process, the points within this simplex have a unique barycentric representation. This allows the *Neural Impostor* to maintain stability during the model's physical deformation process and avoids effects like flickering. On the other hand, barycentric coordinates are intrinsically normalized expressions. During the encoding process, no additional normalization is needed, providing good support for scenes of any scale. Simultaneously, barycentric coordinates establish a continuous space subject to linear interpolation. Consequently, *Neural Impostor* transcends resolution limitations, facilitating the high-quality rendering of intricate structures, such as hair. In the following sections, we will reconstruct the five components of the neural radiance field to elaborate on the modeling method of the *Neural Impostor*.

#### 3.1. Tetrahedral Neural Representation

As shown in Figure 1, to construct an editable implicit radiance field, we reintroduce an explicit representation of its geometric form while modeling the radiance field representing the appearance. That is, a coarse tetrahedron mesh that acts as the boundary for shape editing, also known as the "cage" [JSW05, JMD\*07]. Unlike those traditional cage-based shape editing methods, the tetrahedral cage in our NeRF model does *not* have a fine mesh embedded in—after all, the bare bones of a NeRF model lie in its implicit representation of the radiance field. Therefore, within each tetrahedron of the cage, we use MLPs to represent its view-dependent radiance field distribution, maintain a separate radiance field in each

tetrahedron of the cage, and convert the position encoding under the original Cartesian coordinate system into the barycentric coordinate space defined by the cage. At the same time, because the definition of distance and angle under barycentric coordinates is not clear, we retain the original direction encoding method of NeRF under the Cartesian coordinate system, to model the view-dependent effect by introducing the direction of light in the subsequent decoding process (as seen in Figure 2).

Specifically, we use  $M = (V, T)$  to represent the tetrahedron mesh of the Neural Impostor, where  $V = (v_1, v_2, \dots, v_N)$  represents the set of  $N$  vertices of the tetrahedron mesh, and  $T = (t_1, t_2, \dots, t_K)$  represents the  $K$  tetrahedron that make up the tetrahedron mesh. Each tetrahedron  $t_k \in T$  is represented by its four vertices  $V_t = (v_0^k, v_1^k, v_2^k, v_3^k)$ . For each position coordinate  $p$  in Cartesian space, we map it to the corresponding tetrahedron  $t$  and the corresponding barycentric coordinates  $\Lambda_t = (\lambda_0, \lambda_1, \lambda_2, \lambda_3)$  using the barycentric coordinate query function  $Q(\cdot)$  to construct the encoding space, that is:

$$\mathcal{G}_M = \bigcup_{t \in T} \Lambda_t$$

$$\Lambda_t = \left\{ \{\lambda_0^t, \lambda_1^t, \lambda_2^t, \lambda_3^t\} \mid 0 \leq \lambda_i^t \leq 1 \text{ and } \sum_i \lambda_i^t = 1 \right\} \quad (2)$$

$$Q: \mathbb{R}^3 \rightarrow \mathbb{R}^5, \quad Q(p) = (t, \Lambda_t), \quad \text{and } p = \Lambda_t V_t$$

during rendering,  $p$  can be transformed to be represented in a ray coordinate system with the ray origin  $\dot{o}$  as the coordinate origin and the ray direction  $\vec{d}$  as the  $z$ -axis. Considering the good linear interpolation characteristics of barycentric coordinates, for any point  $p$  inside the tetrahedron, we can obtain it by linearly interpolating the barycentric coordinates of the incident and exit points of the ray relative to the tetrahedron, namely:

$$\Lambda_t^p = \alpha \Lambda_t^{p_0} + (1 - \alpha) \Lambda_t^{p_1},$$

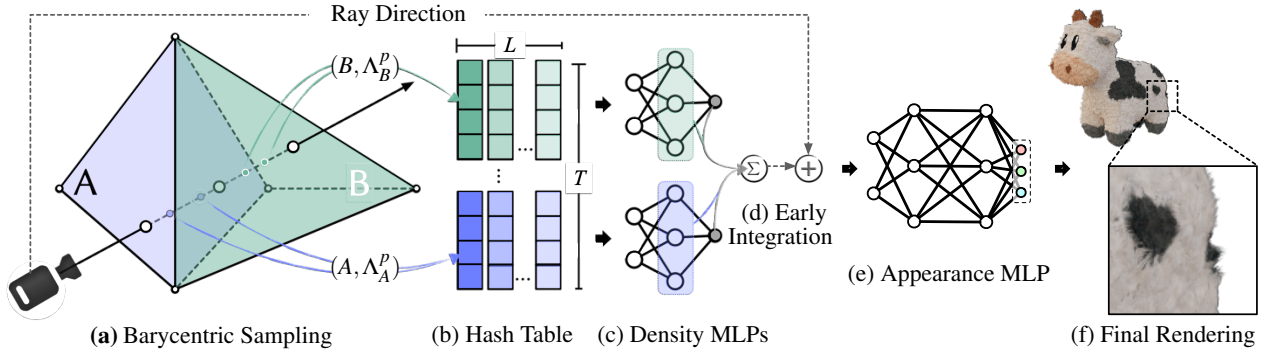
$$\text{where } p_0 = \dot{o} + t_0 \vec{d}, \quad p_1 = \dot{o} + t_1 \vec{d}, \quad \text{and } t_0 < t_1 \quad (3)$$

where  $\alpha \in [0, 1]$  determines the relative distance of  $p$  to the entrance/exit points. Finally, we pack the barycentric coordinates of each sampling point  $\Lambda_t^p$ , the tetrahedron index  $t$ , and the light ray direction in the Cartesian coordinate system  $\vec{d}$  together to represent a sample in the encoded space  $\mathcal{G}$ .

Apart from enabling shape editing (see Section 4.1), the tetrahedral cage can be also used as a proxy for collision processing. It allows the use of full-fledged collision processing algorithms to detect and resolve collisions between a NeRF model and other objects—an essential component for bringing NeRF models in a physical simulation. Last but not least, the tetrahedral cage offers a natural data structure that helps to speed up the image rendering process. Typically, the NeRF rendering algorithm samples a set of points along each camera ray and accumulates a color value. With the cage, this process can be implemented more efficiently.

#### 3.2. Robust Space Sampler

NeRF [MST\*20] treats the target scene as a volume space with a certain density, and the physical quantities of discrete sampling points are approximated by integration using body rendering.



**Figure 2:** Rendering of the Neural Impostor starts by determining intersection points between a ray and a tetrahedron mesh, then samples query points between the in and out intersections, based on the size of the tetrahedron (a). For each sample point, we use its barycentric coordinates and corresponding tetrahedron to query the hash table (b). A series of small multi-layer perceptrons (MLPs) calculate the spatial density for each sampling point from positional features stored in the hash table (c). To improve rendering efficiency, the early integration process aggregates the features of the sampling points, then alpha blends them into a final appearance feature vector for each ray (d). The final step involves calculating the per-ray RGB color with a substantial appearance MLP and summing over all tetrahedrons (e-f).

Within a certain limit of the number of samples, in order to accurately model the scene details, we want the sampling points to be concentrated as much as possible in places that contribute to the pixel color to avoid computational waste, i.e., the sampling density should approximate the real density distribution of the scene as much as possible. For this reason, in NeRF [MST\*20], researchers designed a sampling method based on probability distribution, i.e., first obtain the approximate density distribution in space by coarse-grained uniform sampling, then perform fine-grained sampling according to the probability density of coarse-grained sampling points, and fuse the coarse and fine-grained sampling points for the integration process of body rendering. This process involves querying the spatial density decoder for both coarse- and fine-grained sampling, and therefore greatly reduces the rendering effectiveness.

Instant-NGP [MESK22] accelerates the rendering process with an explicit spatial occupancy grid that roughly marks the empty and non-empty states of space using binary bits. The sampling process then skips over sparse space (i.e., empty space), thereby concentrating the sample points near the surface. This method of sampling based on the spatial occupancy grid can greatly speed up the rendering process in NeRF. However, this approach also requires the spatial occupancy grid to be updated at a certain frequency during the training process, namely re-querying the MLP to obtain the spatial density of the grid vertices. Besides, the misalignment between the voxel grid structure defining the occupancy field and the tetrahedron grid may cause jittering or ghosting artifacts.

Therefore, we designed a sampling method based on barycentric coordinates. According to the truncation theorem [MESK22], the sampling step length during light propagation should be proportional to the propagation distance along the ray. This proportion is determined by the predefined cone angle  $\theta$ . Considering that the size of each tetrahedron may change significantly during the deformation process, while the barycentric coordinate distances defined by the incident and outgoing points of the tetrahedron remain rela-

tively stable, we use the interpolation weight  $\alpha$  from Equation (3) to replace the step length in the Cartesian coordinate system, which means:

$$\Delta\alpha_{i+1} = \Delta\alpha_i + \theta \cdot A_i \quad (4)$$

where  $A_i = \sum_j \Delta\alpha_j$  represents the cumulative sampling distance starting from the incident point of the first valid tetrahedron. Consequently, the sampling process can be described as:

$$S: \mathbb{R}^6 \rightarrow \mathbb{R}^{N \times 8}$$

$$S(\hat{o}, \vec{d}) = \left\{ \{t, \Lambda_t, \vec{d}\}_{\times N} \mid t \in T, \Lambda_t \in \mathbb{R}^4, 0 \leq \Lambda_t \leq 1 \text{ and } \|\Lambda_t\|_1 = 1 \right\} \quad (5)$$

where  $\hat{o}, \vec{d}$  represent the origin and direction of the ray,  $N$  stands for the number of sample points for each ray, and  $(t, \Lambda_t)$  respectively represent the index of the tetrahedron to which the sample point belongs and its 4-dimensional barycentric coordinates. After sampling, each sample point is represented as a set composed of tetrahedron index, barycentric coordinates, and ray direction.

### 3.3. Neural Encoding in a Tetrahedron

In every single tetrahedron, we need to store a view-dependent neural radiance field. Furthermore, the radiance field must transform smoothly when the tetrahedron is deformed to avoid flickering artifacts. With this in mind, we represent the positional quantity of the neural radiance field in barycentric coordinates. As the light ray direction of the neural radiance field is a deformation-independent quantity, we encode the light ray direction in the form of spherical harmonic functions in original Cartesian coordinate system. The barycentric coordinate system is a local linear space relative to each tetrahedron. When we change the tetrahedron's vertex positions, points with fixed barycentric coordinates will move correspondingly. Therefore, under the encoding method of the *Neural Impostors* based on the tetrahedron barycentric coordinates, when the tetrahedron acting as a proxy deforms, the neural radiance field

encoded by each tetrahedron will also undergo corresponding shape changes. Represent the tetrahedron lookup function as  $Q$ , local coordinate encoder inside tetrahedron  $t$  as  $E_p^t$  and the global direction encoder as  $E_d$ , then the encoding process of the *Neural Impostors* at sampling point  $p$  can be represented as:

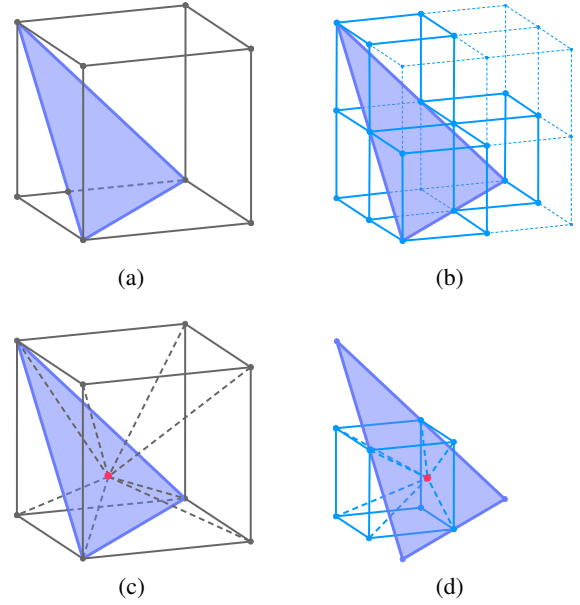
$$\begin{aligned} Q: \mathbb{R}^3 &\rightarrow \mathbb{I}, \quad t = Q(p) \\ E_p^t: \mathbb{R}^4 &\rightarrow \mathbb{R}^{F \times L}, \quad f_p = E_p^t(\Lambda_t^p) = h_t(\Lambda_t^p) \\ E_d: \mathbb{R}^3 &\rightarrow \mathbb{R}^{D^2}, \quad f_d = E_d(\vec{d}) = SH_D(\vec{d}) \end{aligned} \quad (6)$$

where  $\Lambda_t^p$  is the barycentric coordinate of the sampling point  $p$  inside the tetrahedron  $t$ ,  $L$  is the number of levels in the multi-resolution hash,  $F$  is the number of features in each level,  $D$  is the order of the spherical harmonic function encoding,  $h_t$  represents the feature querying processing from tetrahedron  $t$ 's local hash table, and  $SH_D$  is the  $D$ -order spherical harmonic function. After encoding, we can obtain the position feature  $f_p$  and direction feature  $f_d$  corresponding to each sampling point  $p$ .

However, the use of barycentric coordinates complexes the encoding process (i.e.,  $E_p^t$  in Equation (6)). We wish to leverage the multi-resolution hash encoding scheme proposed in [MESK22], as it offers high rendering quality and low memory footprint. This encoding scheme conceptually stores trainable features on a multi-resolution Cartesian grid, and directly transferring this encoding scheme in barycentric coordinates requires the creation of a multi-resolution tetrahedron mesh within a tetrahedron and the storage of feature vectors on the tetrahedron mesh's vertices. Yet, this approach is troublesome. When a tetrahedron is subdivided for finer level mesh creation, it results in four smaller tetrahedron and an octahedron in the center. It is unclear how to split the octahedron into a set of tetrahedron in a consistent way. More importantly, when it comes to image rendering, a recurring step is to find, on each multi-resolution level, the tetrahedron in which a sampled point on a camera ray is located. This, however, is computationally much more expensive than its counterpart in [MESK22] (i.e. finding in an axis-aligned Cartesian grid the voxel in which a sampled point is located).

Instead, we propose to maintain trainable features not on any tetrahedral vertices but on a four-dimensional (4D) grid associated with the tetrahedron. The 4D grid is multi-resolution, constructed in the following way. Consider a point  $p$  inside the tetrahedron  $t$  with barycentric coordinates  $\Lambda_t(p) = \lambda_0^t, \lambda_1^t, \lambda_2^t, \lambda_3^t$ . Although its barycentric coordinates only have three degrees of freedom (i.e., constrained by  $\sum_{i=0}^3 \lambda_i^t = 1$ ), algebraically it can be regarded as a 4D point located inside a 4D voxel. This 4D voxel has 16 vertices (corners), each with a coordinate  $(u_1, u_2, u_3, u_4) \in 0, 1^4$ . This voxel forms the highest-level grid with a resolution of  $1 \times 1 \times 1 \times 1$ . We create the multi-resolution grid by progressively subdividing this highest-level grid, and bind the trainable features characterizing the radiance field of the tetrahedron to the nodes of the multi-resolution grid.

The relation of a tetrahedron to its 4D grid can be interpreted geometrically. On the highest-level grid—which is a single 4D voxel—there are 16 voxel vertices. Four of them have coordinates satisfying  $\sum_{i=1}^4 u_i = 1$ , which are valid barycentric coordinates corresponding to the tetrahedron's four vertices. In light of this, the



**Figure 3:** Multi-grid hash encoding for a triangle, its barycentric coordinates define a three-dimensional encoding space, but the valid encoding space is merely a two-dimensional embedding (purple triangle) within this three-dimensional space. (a) and (c) gives the valid voxels and voxel corners used for interpolation under the first grid level, while (b) and (d) demonstrate the corresponding scenario after subdivision.

tetrahedron can be viewed as a 3D region embedded in a 4D hyperspace. A sampled point  $p$  on a camera ray is always located in the tetrahedron, but in order to compute the feature vector at  $p$ , we interpolate the features stored in the corners of a 4D voxel (see Figure 3). Further, the highest-level grid is subdivided into 16 4D voxels on the second level of the grid. But the 3D tetrahedron is embedded in only 5 of the 16 voxels. The rest of the voxels remains unused when we compute feature vectors of sampled ray points. See Figure 3 for visualization of this interpretation in 3D space.

We note that the unused voxels will not waste memory storage, because following Instant-NGP [MESK22], the feature vectors on grid nodes are stored in a hash table, and the hash table size is set on purpose much smaller than the grid size to “compress” feature vector storage. Meanwhile, considering that we need to maintain separate hash tables for each tetrahedron in the tetrahedron mesh of the *Neural Impostor*, when the number of tetrahedrons increases, the amount of information each tetrahedron actually needs to encode also decreases. Therefore, in practice, we reduce the hash table size of each tetrahedron according to the number of tetrahedrons in the mesh, in order to maintain the hash table size of the entire tetrahedron mesh comparable to that in the original Instant-NGP with a given level. For instance, consider a tetrahedron mesh with  $T$  tetrahedrons and base-2 logarithmic hash table size  $H$ , the corresponding local base-2 logarithmic hash table size in each tetrahedron would be  $H' = H / \lfloor (\log_2 T + 1) \rfloor$ . The rest of the encoding pro-

cess is similar to Instant-NGP. At each multiresolution grid level  $l$ , the feature vector  $f_l(x)$  at the sampling point  $x$  is obtained through four-dimensional linear interpolation, using the feature vectors at the 16 corner points of the voxel containing  $x$ . We concatenate feature vectors  $f_l(x)$  from all levels  $l = 1 \dots L$  to form the feature vector  $f_p$  used for decoding. The decoding process of *Neural Impostor* mirrors that of NeRF [MST\*20]. Given a density decoder, denoted  $D_\sigma$ , and a radiance decoder  $D_r$ , we have:

$$\begin{aligned} D_\sigma: \mathbb{R}^{F \times L} &\rightarrow \mathbb{R}, & \sigma(p) &= D_\sigma(f_p) \\ D_r: \mathbb{R}^{D^2 + F \times L} &\rightarrow \mathbb{R}^3, & c(p) &= D_r(f_p \oplus f_d) \end{aligned} \quad (7)$$

herein,  $p = \dot{o} + t\vec{d}$  denotes a sampled point along the ray,  $f_p$  and  $f_d$  represent the position feature and the direction feature of  $p$  and  $\oplus$  indicates feature concatenation.

### 3.4. Efficient Image Rendering

During rendering, we need to calculate the color value  $C(r)$  along the sampled camera ray  $r(t) = \dot{o} + t \cdot \vec{d}$ , where  $\dot{o}$  is the origin of the ray and  $\vec{d}$  is the ray direction. Similar to the original NeRF model [MST\*20], following the principles of Direct Volume Rendering (DVR), the color of the ray  $C(r)$  can be estimated by integrating over a bounded distance interval  $[t_n, t_f]$ :

$$C(r) = \int_{t_n}^{t_f} T(t) \sigma(r(t)) c(r(t), \vec{d}) dt, \quad (8)$$

here,  $T(t) = \exp\left(-\int_{t_n}^t \sigma(r(s)) ds\right)$  represents the accumulated transmittance from  $t_n$  to  $t$  along the ray direction.  $\sigma(r)$  is the volume density at location  $r(t)$ ;  $c(r(t), \vec{d})$  is the radiance at  $r(t)$  in the direction  $\vec{d}$ .

The fundamental idea of NeRF [MST\*20] is to represent  $\sigma(r(t))$  and  $c(r(t), \vec{d})$  using neural networks. In our tetrahedron neural representation, the camera ray  $r$  intersects with a series of tetrahedron. During the deformation process, the volume of each tetrahedron might undergo significant changes. Under these circumstances, if we keep the Cartesian coordinate-based integral method, the final color and transparency of the ray would directly correlate with the integral distance. Consequently, any changes in the integral distance due to the deformation of the tetrahedron would also affect the output color.

To dissociate the color of the ray from the integral distance, we leverage the sampling method discussed in Section 3.2. We rewrite Equation (8) as a sum of the integrals for each tetrahedron, while converting the integral distance into a distance representation in barycentric coordinates. This ensures that the color of the ray remains unaffected, even when the volume of the tetrahedron experiences significant deformation. As illustrated in Figure 2, we denote the  $K$  tetrahedron that the ray passes through during propagation as  $t_1, t_1, \dots, t_K$ . Each tetrahedron intersects with the ray at entry point  $p_0$  and exit point  $p_1$ , with corresponding barycentric coordinates  $\Lambda_{t_i}^0$  and  $\Lambda_{t_i}^1$ , respectively. We use the Manhattan distance between  $\Lambda_{t_i}^0$  and  $\Lambda_{t_i}^1$  as the integral distance, and recast Equation (8) as a sum of integrals over each tetrahedron:

$$C(r) = \sum_{i=1}^K \int_{\Lambda_{t_i}^0}^{\Lambda_{t_i}^1} T(\Lambda_{t_i}^p) \sigma(\Lambda_{t_i}^p) c(p, \vec{d}) d\Lambda_{t_i}^p \quad (9)$$

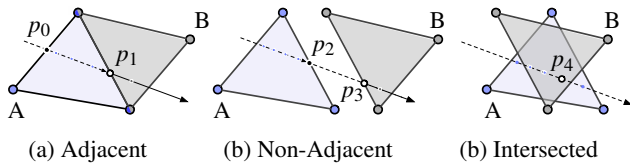
Here,  $\Lambda_{t_i}^p$  represents the barycentric coordinates of the sampling point  $p$  within the tetrahedron  $t_i$ . Due to the linearity of the barycentric coordinates,  $\Lambda_{t_i}^p$  can be determined by the barycentric coordinates of the entry and exit points,  $\Lambda_{t_i}^0$  and  $\Lambda_{t_i}^1$ , respectively, as well as the interpolation weight  $\alpha$  as defined in Equation (3).

The radiance in a scene typically carries a higher frequency of information than the spatial density. Therefore, the radiance decoder in the process of constructing a neural radiance field usually exhibits more complexity than the spatial density decoder. Moreover, the forward process of a MLP is more costly in terms of time efficiency compared to encoding and rendering. To further optimize the rendering efficiency, we first decode the position features of the sample points into density values using a few density decoders  $D_\sigma$ . In contrast to previous rendering schemes that decode before integration for radiance, we propose a rendering scheme that integrates before decoding. Specifically, we first integrate the position features of the sampling points and ray transmittance onto its belonging tetrahedron surfaces, then alpha composite them together into a final appearance feature for each ray. Subsequently, the composited appearance feature are combined with the ray directional feature for radiance decoding, as such we only need to computer the appearance decoder once for each ray:

$$C(r) = D_r \left[ \underbrace{\sum_{i=1}^K \int_{\Lambda_{t_i}^0}^{\Lambda_{t_i}^1} T(\Lambda_{t_i}^p) \sigma(\Lambda_{t_i}^p) E_p^t(\Lambda_{t_i}^p) d\Lambda_{t_i}^p}_{\text{integrated positional feature}} \oplus E_d(\vec{d}) \right] \quad (10)$$

here,  $D_r$  is the radiance decoder, and  $E_p, E_d$  are the encoders for the barycentric coordinates and ray direction, respectively. The density values of the sample points are obtained by decoding the barycentric coordinates according to Equation (7). This "integration-before-decoding" rendering approach effectively prebakes the scalar field of the encoding space onto the surface of each tetrahedron. As each ray can define a unique surface point for each tetrahedron, this "integration-before-decoding" approach allows us to rasterize the tetrahedron in the space layer by layer and project them onto the imaging plane to further accelerate the rendering process during inference.

While the "integration-before-decoding" rendering approach can accelerate the rendering process during inference, in the barycentric coordinate-based encoding method, we also need to determine the tetrahedron to which each sample point belongs during the training process. For a 1080p image, the forward process of training requires conducting a barycentric coordinate test for approximately  $1080 \times 1080 \approx 10^8$  points. Traditional tests based on matrix multiplication would require us to compute the determinant of a  $N_{rays} \times N_{points} \times 4 \times 3$  matrix, which is unfeasible for complex tetrahedron meshes in one forward pass. Building upon the work of Wald et al. (2019) [WUM\*19], we consider the triangles that make up the tetrahedron as the testing units during the rendering process. For non-self-intersecting tetrahedron meshes, the concept of a point being "in front of a triangle" or "behind a triangle" can be uniquely determined by the winding order of the triangle vertices. In their method, determining to which tetrahedron a point belongs is equivalent to casting a ray from that point in a random direction, evaluating the winding order at the first intersection point, and checking on which side of the triangle the sample lies. Since each triangle



**Figure 4:** Three tetrahedron connection scenarios observed during sampling. For adjacent tetrahedrons sharing a triangle face, the entry point  $p_0$  originates the ray, and steps determine the exit point  $p_1$ , which then updates to the new entry point. For separate tetrahedron, we link them at runtime, moving a point at  $p_2$  to  $p_3$  before sampling. With intersecting tetrahedron, an aggressive sampling approach adds a point  $p_4$  that belongs to both tetrahedron A and B to the sampling set.

belongs to at most two tetrahedron, this winding order-based testing approach can yield a unique tetrahedron index for any point in space. When hardware acceleration is available, the triangle-based query operation is also very efficient in terms of memory and computation.

We extend the method of [WUM\*19] to the framework of ray tracing to support the sampling process in Section 3.2. Specifically, in the context of ray tracing, we can perform stepwise sampling along the direction of the ray. Unlike the separate testing for each point in the method of Wald et al. (2019) [WUM\*19], we only need to determine the entry and exit points of the tetrahedron that each ray passes through. Depending on the arrangement of tetrahedron in space, the progression of the ray may encounter three situations shown in Figure 4, namely adjacent/non-adjacent tetrahedron and intersecting tetrahedron. We adopt different stepping strategies for these three situations. As shown in Figure 4, for adjacent tetrahedron that share one triangle faces, we take the entry point as the origin of the ray and step to determine the exit point. After sampling, the entry point is updated to the exit point. For separated tetrahedron, we connect them at runtime. That is, if the sample point is at position  $p_2$  in Figure 4, we first update it to  $p_3$  before sampling. For intersecting tetrahedron, we use an aggressive sampling approach, i.e., for a sample point  $p_4$  that belongs to both tetrahedron A and B, we add its position in both A and B to the sampling set.

**Rendering Acceleration** Besides the raymarching-based rendering algorithm stated in Section 3.4, we provide a tailored algorithm for ultra-fast rendering in large-scale and multi-object scenes. We achieve the ultra-fast rendering by baking the appearance of a tetrahedron onto its faces as view-dependent neural textures. For each image in the input image set, we cast a ray from the camera through each pixel. As shown in Figure 2 (a), we first compute the intersection points between the ray and the tetrahedral mesh envelope, using the NVIDIA Optix library for this computation. Then, along each ray, we sample the implicit field and integrate the feature values of the sampled points onto the tetrahedral surface using an early integration approach. Finally, we obtain the pixel color through opacity blending. During inference, we can bake the volumetric features of each tetrahedron onto the tetrahedral surface and render

the scene using rasterization. Table 1 provides the efficiency comparison between our implementation and the barycentric coordinate query-based rendering approach used in NeRF-Editing [YSL\*22] with KNN neighborhood query. As seen, even without baking, we can achieve real-time volume rendering with the hardware acceleration of Optix (method "Optix Accelerated" in Table 1). After baking, rendering efficiency can be significantly improved by using rasterization-based rendering in a multi-layer fashion (method "Baking+Rasterization" in Table 1).

#### 4. Editing operations of Neural Impostor

A comprehensive explicit geometric editing framework typically includes continuous editing based on spatial transformations, and topology editing which changes the topological structure of the shape. More specifically, we divide editing operations into three categories: continuous shape morphing, explicit remeshing, and boolean operations. Continuous shape morphing changes only the positions of the vertices of the explicit tetrahedron mesh without altering its topological structure. Explicit remeshing regenerates the explicit mesh to enable detailed editing. Boolean operations are used to logically combine two or more shapes to create a new object.

In addition to supporting basic geometric editing operations, the hybrid modeling approach based on *Neural Impostor* allows complex combinations of operations on implicit fields. By utilizing the aforementioned operations, our system achieves a wide range of editing tasks, including physical simulation, mesh reconstruction, and scene composition. In the following, we will explain how we leverage explicit tetrahedron meshes in *Neural Impostor* to realize the aforementioned editing operations for neural implicit field.

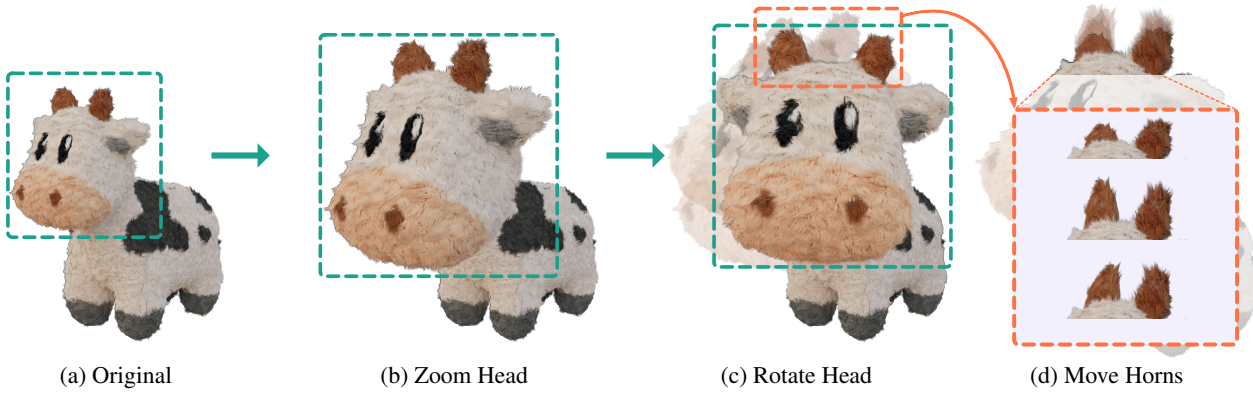
##### 4.1. Shape Transformation and Deformation

Shape transformation and deformation are the most fundamental geometric editing operations. Fundamentally, shape transformation and deformation reveal geometric invariance under continuous mappings, which can naturally be satisfied by the hybrid modeling of *Neural Impostor*. Thanks to the centroid coordinate encoding method described in Section 3.1, in *Neural Impostor* any spatial transformation based on tetrahedron vertices can be naturally transformed into the implicit field. For example, in Figure 5, the input shape morphs from the original shape  $M$  on the left to the head rescaled scaled shape  $M'$  on the right. Recalling equation 2, the sampled points in the deformed tetrahedron mesh  $M'$  are given by:

$$t, \Lambda_t^p \leftarrow S(p, M'), \text{ where } p \in \mathbb{R}^3 \text{ and } t \in M' \quad (11)$$

In this case, both the sampling spaces before and after deformation are defined in the Cartesian space of the scene. When the tetrahedron vertices move, the implicit neural field inside the deformed tetrahedron remain unchanged under the barycentric coordinate. Thus, we can adopt the same sampling, encoding, and rendering methods as the original process without any additional transformations. Moreover, as this process only involves updates to the tetrahedron mesh vertices without calculating the transformations on sampling points, the deformation process in *Neural Impostor* is more efficient than approaches based on bending rays [XH22, PYL\*22, GKE\*22, YSL\*22].





**Figure 5:** Shape morphing based on vertices with local remeshing. In this process, we scale some vertices of (a) the original tetrahedron mesh to achieve (b) local scaling and (c) the rotating motion of the neural radiance field. Further, (d) we remesh the ear region for finer scale shape morphing.

#### 4.2. Subdivision and Remeshing

Compared to continuous shape morphing, topological operations pose more challenges. When the mesh topology changes, the number and connectivity of the original tetrahedron are altered. However, in *Neural Impostor* the radiance field is strictly tied to the tetrahedron topology. This requires an effective mapping scheme to transfer the radiance field baked into the original tetrahedron mesh to the new tetrahedron structure.

To achieve this goal, we propose a local retraining strategy to synthesize neural implicit fields from existing neural implicit fields with different explicit tetrahedral mesh. When remeshing happens, rather than performing a global retraining, we select local tetrahedron where the topology changes for retraining while keeping the other tetrahedron unchanged. Besides, to accelerate convergence, we initialize the updated multigrid hash table based on feature-only supervision shown in Figure 6 (c). Then follow a joint optimization on both the feature hash table and the MLP weights. As shown in Figure 6, during the retraining stage, our input is the pre-trained *Neural Impostor* which includes its tetrahedron mesh and the implicit neural field encoded in each tetrahedron illustrated in Figure 6 (b), as well as the explicit tetrahedron mesh after the topological transformation as shown in Figure 6 (a). The goal of local retraining is to transfer the radiance field defined on the original tetrahedrons to the new radiance field on the remeshed tetrahedral mesh. The retraining process consists of two stages. In the first stage, as shown in Figure 6(c), we randomly sample points inside (solid dots in Figure 6 (a)) and outside (hollow dots in Figure 6 (a)) of the new mesh to optimize the hash table. In the second stage (Figure 6 (d)), we use ray tracing to locate the pixel positions in that region and select effective rays to obtain pixel colors as further supervision according to the rendering process in Equation (10), this improving the accuracy of the rendered results after retraining. Specifically, let  $(M, h, D)$  represent the tetrahedron mesh, hash encoding table, and decoder of the *Neural Impostor* before retraining, and let  $(M', h', D')$  represent the reconstructed tetrahedron mesh and its to-be-trained hash encoding table and decoder. Let  $R$  and  $E_d$  represent the renderer and ray direction encoder used in the second stage

training. The training process can be formalized as follows:

$$\begin{aligned}
 \text{Stage 1: } \mathcal{L}_{hash} &= \|h(\Lambda_p) - h'(\Lambda_p)\|^1, \text{ where } p \in M \cup M'. \\
 \text{Stage 2: } \mathcal{L}_{render} &= \left\| R \circ D[h(\Lambda_p) \oplus E_d(\vec{d})] - R \circ D'[h'(\Lambda_p) \oplus E_d(\vec{d})] \right\|^2, \\
 &\text{ where } p \in \dot{o} + t\vec{d}.
 \end{aligned} \tag{12}$$

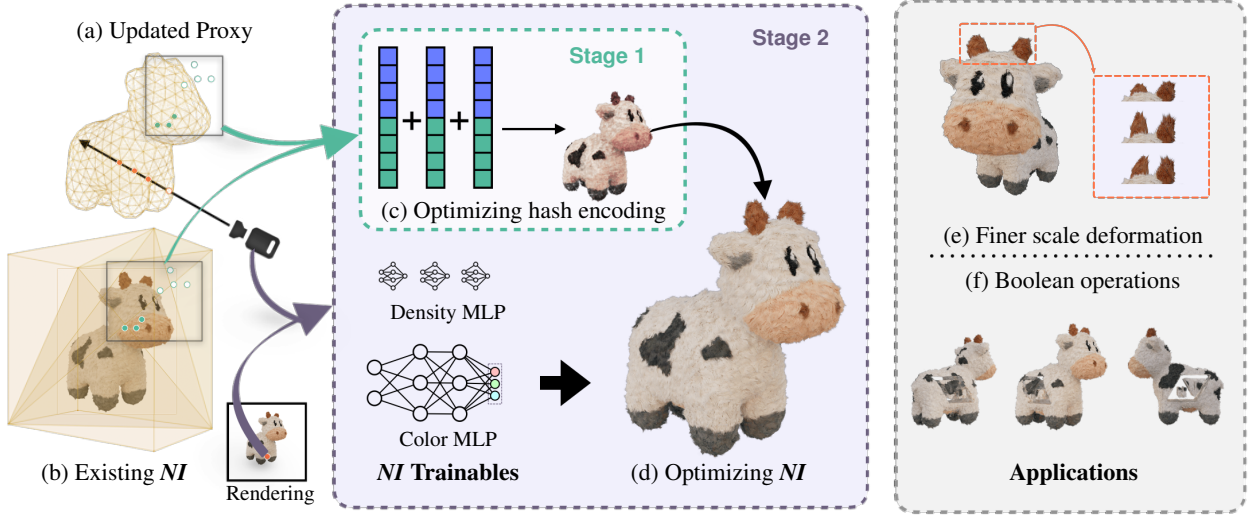
where  $\circ$  denotes function composition and  $\oplus$  represents feature concatenation.

#### 4.3. Boolean Operation

Besides the operations on the explicit part in the tetrahedral neural representation, inter-operations between different neural radiance fields are essential to certain cases, such as Boolean operations. To merge or subtract two neural radiance fields, we need to define boolean status on the implicit fields. Luckily, its much easier to define the boolean status on the implicit field (such as NeRF) than the explicit geometry (such as mesh). For explicit geometry, the boolean status is defined by the in/out status of the sampling point, which is not well defined for non-watertight geometry, and usually hard to compute. On the other hand, the implicit field is indexed with positional parameters in a universal encoding space  $G$ , which allows us to define the boolean status based on the scalar values interpreted for each sampling point. Here we use a straightforward filtering algorithm to define the boolean status  $\mathcal{B}(\lambda)$  based on simple thresholds:

$$\mathcal{B}(\lambda) = \begin{cases} 1, & \text{if } \mathcal{D}_\sigma(\lambda) > \epsilon \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

Therefore, the boolean field  $\mathcal{B}$  is a binary function of the density field  $\mathcal{D}$ . Based on the boolean field, we can define the basic boolean



**Figure 6:** Two-Stage Retraining Process. During (c) Stage 1, the hashtable is optimized using randomly sampled points to align with the existing *Neural Impostor*. Only points located within the updated proxy (solid green points) are taken as valid samples, while others (hollow points with green borders) are disregarded. However, solely updating the hash table could potentially result in noisy rendering due to hash collision. To mitigate this issue, a 2nd-stage training (d) is introduced to refine the decoder MLPs and resolve hash collision. Here, rays are randomly traced into the updated proxy, with the accumulated per-ray color further guiding the adjustment of density and color decoders for the new *Neural Impostor*. This efficient local retraining mechanism facilitates (e) fine-scale deformation and (f) local appearance editing.

operations between neural radiance fields  $\mathcal{B}_i$  and  $\mathcal{B}_j$  as follows:

$$\begin{aligned}
 \mathcal{B}_i \cup \mathcal{B}_j &= \mathcal{B}_i + \mathcal{B}_j - \mathcal{B}_i \cdot \mathcal{B}_j \\
 \mathcal{B}_i \cap \mathcal{B}_j &= \mathcal{B}_i \cdot \mathcal{B}_j \\
 \mathcal{B}_i \setminus \mathcal{B}_j &= \mathcal{B}_i - \mathcal{B}_i \cdot \mathcal{B}_j \\
 \mathcal{B}_i \oplus \mathcal{B}_j &= \mathcal{B}_i + \mathcal{B}_j \pmod{2}
 \end{aligned} \tag{14}$$

Then we mask out the corresponding region of the objects and do the local re-training by using the result boolean field as binary weights for the volumetric rendering process.

$$C_k = \text{Dr} \left( \int_{t_0}^{t_1} T_{tet}(\lambda_t) \underbrace{\mathcal{B}(\lambda_t)}_{\text{boolean}} \underbrace{D_\sigma(E_p(\lambda_t))}_{\text{density}} \cdot \underbrace{E_p(\lambda_t)}_{\text{pos feature}} dt \oplus \underbrace{E_d(\vec{d})}_{\text{dir feature}} \right) \tag{15}$$

accumulated feature

As shown in Figure 7, through implicit boolean operations, we can add constructive pattern details to a pre-trained *Neural Impostor* model. Please note that the boolean field determines the presence of the density field, but the color of the model is interpreted based on the selection of the radiance field. By the way, boolean operations can be accelerated by first checking the boolean state of tetrahedron and then performing implicit boolean operations on the selected tetrahedron. By default, boolean operations are binary selective operations on implicit fields. As a natural extension, selective blending can be achieved using the similar strategy. Instead of keep or drop a radiance field, we can blend or perform any mutual operations on two radiance fields. For example, as shown in Figure 7 (c), within the region defined by the pattern, we multiply the color given by the pattern with the density field representing

fur in the original *Neural Impostor* resulting in a fur model with color. For such cases, local retraining is accomplished by using the second-stage blending perspective-related appearance as the target after determining the tetrahedron covered by the pattern:

$$\begin{aligned}
 C_k &= D_r^i \left( \int_{t_0}^{t_1} T_{tet}^i(\lambda_t^i) \mathcal{B}^i(\lambda_t^i) D_\sigma^i(E_p^i(\lambda_t^i)) \cdot E_p^i(\lambda_t^i) dt \oplus E_d^i(\vec{d}^i) \right) \\
 &+ D_r^j \left( \int_{t_0}^{t_1} T_{tet}^j(\lambda_t^j) \mathcal{B}^j(\lambda_t^j) D_\sigma^j(E_p^j(\lambda_t^j)) \cdot E_p^j(\lambda_t^j) dt \oplus E_d^j(\vec{d}^j) \right)
 \end{aligned} \tag{3-16}$$

## 5. Experiments

In this chapter, we first present the data acquisition method and implementation details of *Neural Impostors*. Then, we perform qualitative and quantitative analyses on the modeling capability and rendering efficiency of *Neural Impostor*, using *nerfstudio* [TWN\*23] as a benchmark for comparison. To demonstrate this, we create a dataset specifically for plush toys through rendering and real captures and merge it with the *nerf-synthetic* dataset for quantitative analysis.

### 5.1. Implementation Details

**Data Acquisition** The input for *Neural Impostor* consists of a set of images with corresponding camera parameters and a tetrahedral mesh of the input scene. The input images are similar to those used in traditional 3D reconstruction tasks and serve as input for the *Neural Impostors*. Since the modeling approach of *Neural Impostor* does not require fine geometric details, we have robust sup-



**Figure 7:** Boolean operations on Neural Impostor with given patterns. (b) Geometric difference and union based on the density of the given pattern. (c) Implicit union operation for color editing.

port for generating tetrahedral meshes from both generated models in modeling software and reconstructed meshes from scene captures. Specifically, if the 3D scene is rendered from a virtual model, we can generate a coarse triangular mesh envelope of the original scene by simplifying and decimating the original 3D model. If the 3D scene is captured from real cameras, we run structure-from-motion algorithms to obtain camera poses and leverage efficient reconstruction methods from "NSR" [ZJY\*22] to create a rough surface mesh of the scene for the NeRF model. Then, we compute the triangular mesh envelope based on the surface mesh by using shape modifiers in geometry processing libraries or modeling softwares, such as Libigl and Blender, etc. This process enables us to generate a concise representation of the surface geometry that encapsulates the volumetric characteristics of the object. Finally, we use the *TetWild* [HSW\*20] algorithm to generate the tetrahedral mesh from the triangular mesh envelope.

**Model Structure** As described in Section 3.3, the trainable components in *Neural Impostor* include the multi-tetrahedron hash encoder  $E_p$  and the density decoder  $D_\sigma$  and radiance decoder  $D_r$ , based on multi-layer perceptrons. In most scenarios, we fix the overall size of the hash encoding table to  $2^{19}$ , and the hash table size per tetrahedron varies between  $2^8$  and  $2^{11}$ , depending on the number of tetrahedra. More specifically, inspired by the approach in "Instant-NGP" [MESK22] for handling different resolution levels, we optimize the hash encoding tables of all tetrahedra in each *Neural Impostor* by packing them together based on tetrahedron indices. During querying, we locate the fixed region of the packed hash table using the tetrahedron index. Our density decoder and

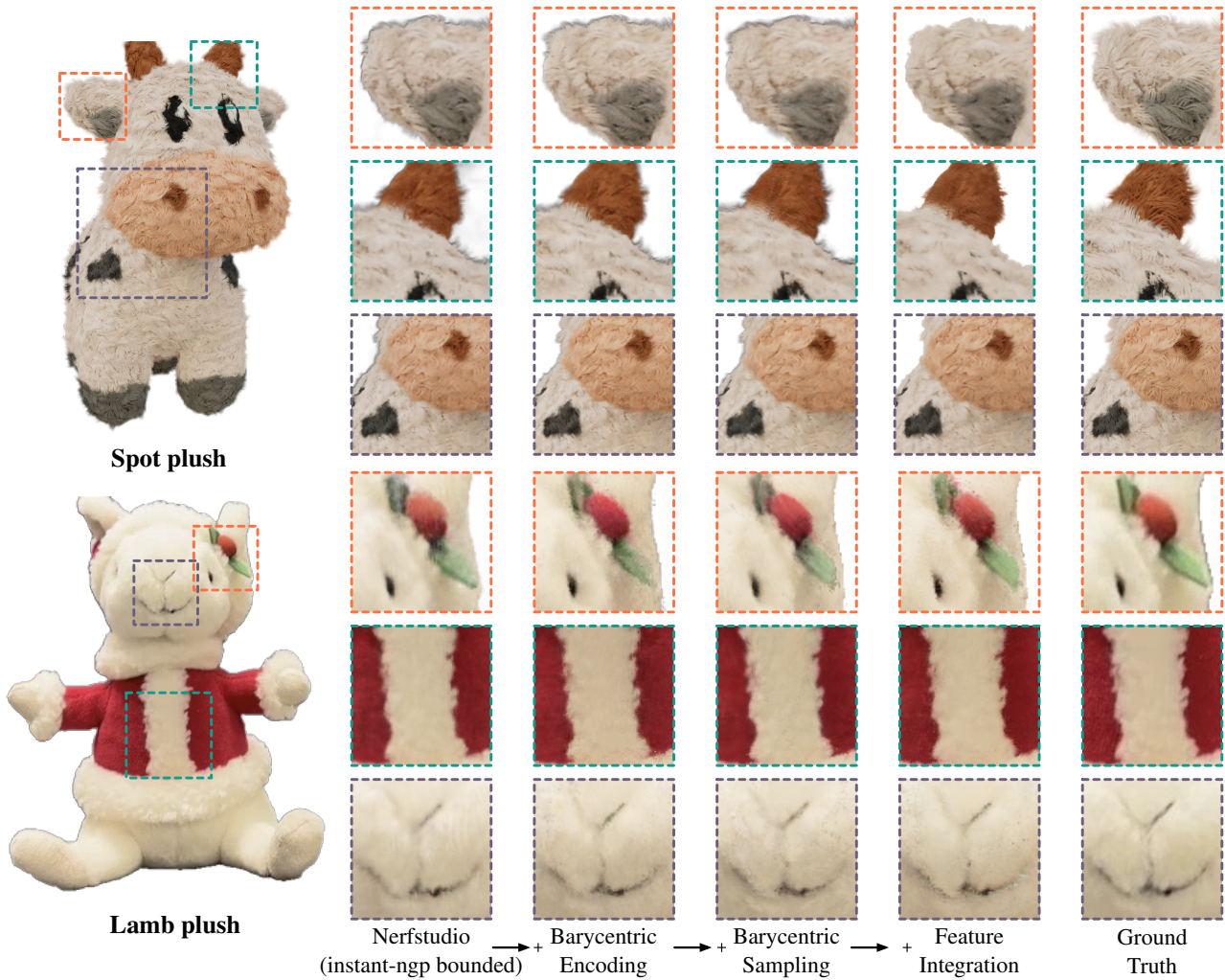
appearance decoder both adopt a multi-layer perceptron structure. The density decoder includes a hidden layer with a width of 16, while the appearance decoder includes two hidden layers each with a width of 64.

**Table 1:** Comparison of Rendering Efficiency (conducted on the *mic* scene in the *nerf-synthetic* dataset, consisting of 294 tetrahedrons and 797 triangles). While *NeRF-Editing* [YSL\*22] takes approximately 7 seconds to render an image with a resolution of  $800 \times 800$ , *Neural Impostor* can achieve real-time rendering at 39.42FPS with *Optix* acceleration. Furthermore, it satisfies real-time gaming requirements after baking features onto the surfaces of tetrahedrons and rendering with a rasterizer.

Method	NeRF-Editing	Optix Accelerated	Baking+Rasterization
FPS $\uparrow$	0.147	39.42	157.83

## 5.2. Reconstruction Quality Evaluation

As mentioned earlier, we analyze the modeling capability of *Neural Impostor* from the perspectives of modeling, rendering efficiency, and continuity in the deformation process. In this section, we focus on evaluating the reconstruction quality. Specifically, using the implementation of *Instant-NGP* in *nerfstudio* [TWN\*23] (method "NGP-Bounded" in Table 2) as a baseline, we first replace the Cartesian coordinate-based encoding part in the original model with barycentric coordinate encoding while retaining the sampling method based on occupancy fields (method "BaryEnc" in Table 2).



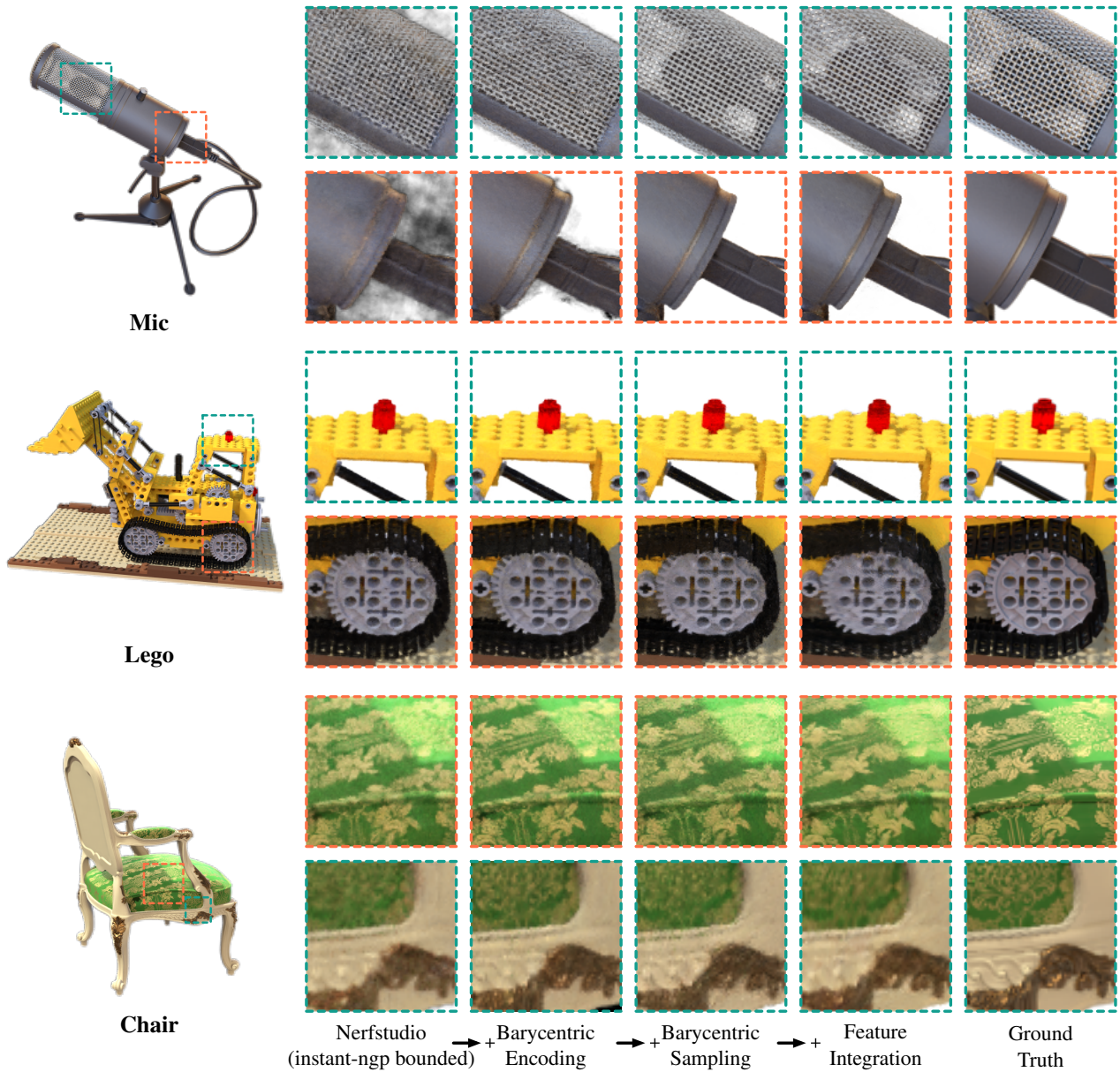
**Figure 8:** Visual comparison on the plush toy dataset. The spot instance above is synthetic data, while the lamb instance below is captured from the real-world with a camera. Neural Impostor accurately reconstructs the hair details. On the instance of the lamb, barycentric encoding achieves better quality on recovering the highly saturated parts (the red part) of the scene than regular space encoding. At the same time, on the instance of the spot, tetrahedral meshes help restrict the sampling space and thus better recover the spatial density field, eliminating the black edge phenomenon.

**Table 2:** Integrated testing result based on nerfstudio

Method	BatchSize	Steps	Nerf-Synthetic (Avg. Impostor Size = 545.57)				Plush Toy (Average Impostor Size = 1754.4)			
			Training Time	Rendering FPS	PSNR $\uparrow$	SSIM $\uparrow$	Training Time	Rendering FPS	PSNR $\uparrow$	SSIM $\uparrow$
NGP-Bounded	8192	30k	40.09m	33.272	<b>36.545</b>	0.981	44.65m	22.28	32.708	0.9474
BaryEnc	8192	30k	36.72m	33.134	36.473	0.9809	32.51m	26.95	32.843	0.947
BarySpl	8192	30k	30.63m	33.566	36.423	<b>0.9813</b>	29.91m	27.51	<b>32.915</b>	<b>0.9477</b>
FeatInt	8192	30k	<b>29.17m</b>	39.42	34.53	0.9764	<b>28.74m</b>	33.11	32.797	0.9393
Baking + Rast	/	/	/	<b>157.83</b>	33.796	0.9695	/	<b>129.32</b>	32.249	0.937

Then, we replace the sampling method based on occupancy fields with barycentric coordinate-based sampling (method "BarySpl" in Table 2). Finally, we integrate the "early integration" rendering approach from Section 3.4 into the model based on the barycentric coordinate sampling method (method "FeatInt" in Table 2). We also

provide the results of rendering after baking onto the tetrahedral surface and using rasterization for rendering (method *Baking+Rast* in Table 2). Among them, we compare the PSNR (Peak Signal-to-Noise Ratio) and SSIM (Structural Similarity) under various rendering methods for 7 instances in the *nerf-synthetic* dataset and 5



**Figure 9:** Visual comparison on the *nerf-synthetic* dataset. Neural Impostor provide comparable or even better quality. More specifically, Neural Impostor performs better on disentangling the density field and the radiance field. For example, in the mic instance, with the help of barycentric sampling, we recover not only the details of the metal case but also the capsule of the microphone.

instances in the *Plush Toy* dataset (including 3 real-captured instances and 2 rendered instances). PSNR measures the quality of an image by computing the mean squared error between the original image and the processed (compressed or reconstructed) image. The higher the PSNR value, the smaller the difference between the processed image and the original image, indicating better quality. SSIM measures the similarity between two images by considering their luminance, contrast, and structural information. It is a more accurate reflection of image quality compared to traditional metrics like PSNR, as it aligns better with human visual perception.

We provide visual comparisons of the reconstruction results on the *nerf-synthetic* and plush toy datasets in Figure 9 and Figure 8, as well as their quantitative results in Table 2. The quantitative analysis on the *nerf-synthetic* dataset shows that, without introducing the *early integration* rendering approach (Section 3.4), using the barycentric coordinate encoding in the modeling process (method "BaryEnc") achieves comparable quality to the original *Instant-NGP* [MESK22], with a PSNR value of 33.134 for barycentric coordinate encoding compared to 33.272 for the original *Instant-NGP* encoding. Furthermore, since barycentric coordinate encoding is

not limited by resolution, it can achieve good reconstruction quality (PSNR = 32.915) in complex scenes such as plush toys when combined with barycentric coordinate-based sampling, surpassing the quality of the original *Instant-NGP* (PSNR = 32.708). When the *early integration* (Section 3.4) approach is introduced, although rendering efficiency improves, there is a slight decrease in modeling quality due to the texture features of each tetrahedron being baked onto the tetrahedral surface. However, after baking, we can use rasterization for rendering, resulting in a significant improvement in rendering efficiency (from 30+ FPS before baking to 100+ FPS after baking). Therefore, in scenarios with high interactivity demands, we can choose to use the *early integration* (Section 3.4) rendering approach to improve interactive performance.

The proposed hybrid representation relies on an explicit tetrahedron mesh as a proxy of the neural implicit field. The quality and density of tetrahedron mesh do impact the encoding of the neural radiance field. However, these factors only have subtle effects on the reconstruction quality. Figure 10 examines the impact of different tetrahedral mesh quality and density on the reconstruction quality of neural impostors in static scene reconstruction. We tests 7 different tetrahedron proxies with similar implicit field sizes (i.e. sum of hash table size of all tetrahedrons) on the fluffy ball example by varying the tetrahedron density and envelope shape. The similar PSNR of all tests indicates that the selection of tetrahedron proxies has an insignificant influence on the quality of the implicit field.

### 5.3. Editing Quality Evaluation

Different from evaluating the reconstruction quality, there is no well-defined metric or reference ground truth for the edited radiance field. Therefore, we use the rendering of the edited Neural Impostor as a qualitative evaluation (please check Section 6.1) and calculate the LPIPS (Learned Perceptual Image Patch Similarity) between the edited Neural Impostor to the reference rendering as the quantitative evaluation. Here, we analyzed the rendering quality during simulation. Specifically, we use the Houdini simulation engine to create dynamic meshes for fracture and deformation effects for the testing scenes. We then perform animation simulations using the ray bending algorithm in NeRF-Editing [YSL\*22] with two-stage sampling (*Ray Bending* in Table 3), the algorithm based on occupancy field sampling combined with barycentric coordinate encoding (*Occupancy Field* in Table 3), and the barycentric-based sampling algorithm in Section 3.2 (*Barycentric Sampling* in Table 3). Since we cannot obtain real multi-view data as a reference during the simulation process, and the deformations generated during the simulation can be negligible with a sufficiently high frame rate, we calculate the LPIPS between each pair of frames as a measure of animation simulation quality. From the comparison of LPIPS between each pair of frames in Table 3, it can be observed that our barycentric sampling method better maintains the appearance stability during the deformation process.

## 6. Application

### 6.1. Physical Simulation








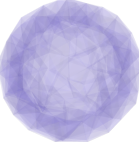

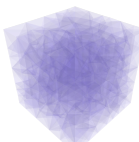
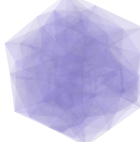
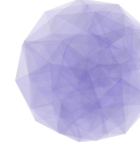
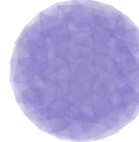
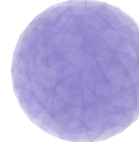
Soft body deformation algorithms are used in computer graphics to simulate the motion and deformation of soft, flexible materials such

**Table 3:** Animation quality comparison. We calculate the LPIPS value between adjacent sampled frames during deformation, and observed that

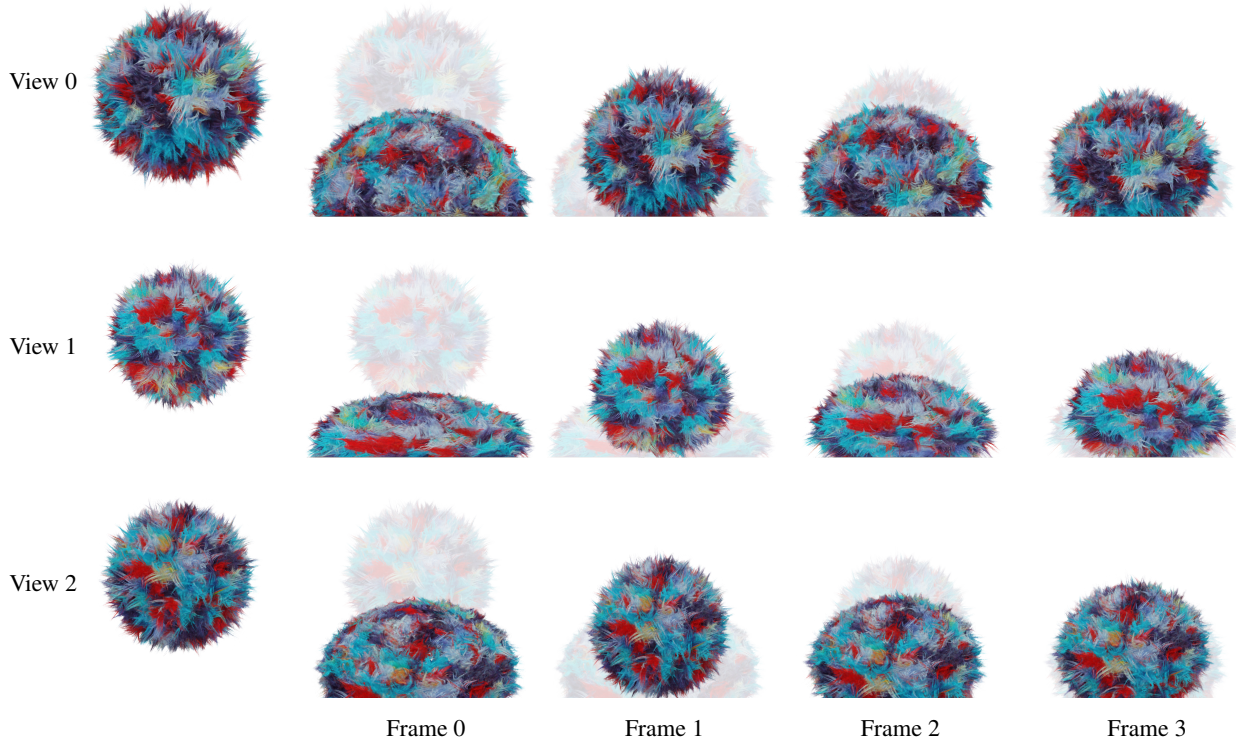
Method	Deform LPIPS	Reference Frame PSNR	Reference Frame SSIM
Ray Bending	0.0999	27.87	0.7572
Occupancy Field	0.0876	<b>37.01</b>	<b>0.9233</b>
Barycentric Sampling (Ours)	<b>0.0669</b>	36.97	0.9175

as cloth, rubber, and muscles. These algorithms combine physics simulation with numerical methods to calculate the motion and deformation of soft bodies under external forces. The basic idea of soft body deformation is to discretize the object into many small elements, and solve for the displacement for each node in the elements. The motion and deformation of each element are calculated based on the forces acting on it, such as gravity, collision forces, tension, and compression forces. One common method of soft body deformation is using finite element analysis (FEA), which divides the body into small triangles or tetrahedral elements. Each element is assigned a set of physical properties, such as stiffness and density, which are used to calculate the motion and deformation of the element under external forces. Other soft body deformation algorithms include particle-spring systems, which model the body as an interconnected network of springs, and position-based dynamics, which use simplified equations to simulate the motion and deformation of the body. By using the same discretized tetrahedron mesh, we can achieve physical simulation directly on *Neural Impostors*. Figure 11 shows an example of soft body simulation based on *Neural Impostor*. In this example, we use a furball with a complex appearance but relatively simple geometry (consisting of 204 vertices and 600 tetrahedra) as the object of simulation. Three significantly different viewpoints and five dynamic frames are extracted and displayed. From the figure, it can be observed that the *Neural Impostor* can achieve continuous simulation with large deformations while maintaining the quality of the volumetric appearance. Furthermore, in Figure 12, we demonstrate continuous soft body transformations based on rendered data and real-shot data. *Plush Spot* represents the rendered data, while *Plush Bear* represents the real-shot data. For more examples, please refer to the supplementary video material.

In addition to soft body simulation, plastic fracture simulation is a technique used in computer graphics to simulate the destruction of buildings, vehicles, and other structures. In plastic fracture, each part of the object is treated as a separate plastic element that can be influenced by external forces such as explosions or collisions. These plastic elements are simulated using physics-based algorithms to calculate their motion and deformation over time. One common method for simulating plastic fracture is to use Voronoi splitting to break the object into smaller fragments. These fragments are then simulated using a physics engine to calculate their motion in the scene and collisions with other objects. Other techniques used for plastic fracture include finite element analysis (FEA) and dynamic fracture modeling, which can produce more detailed and realistic simulations of object destruction. Figure 13 illustrates two examples of plastic fracture simulation on *Neural Impostors*. Similar to soft body simulation, we utilize the Houdini

PSNR	33.01	32.58	32.89	32.96	33.18	33.24	32.80
Rendering							
Proxy							
Shape	Shell	Bounding Box	Cube	Icosphere	1-Subdivided Icosphere	2-Subdivided Icosphere	3-Subdivided Icosphere
Tets #	600	6	2258	544	436	1372	1197

**Figure 10:** The impact of tetrahedron proxy quality. The fluffy sphere is reconstructed with different tetrahedron proxies. The visual quality and PSNR both have subtle differences, indicating the insignificant impact of the proxy quality.



**Figure 11:** Real-time Soft Body Deformation Utilizing Neural Impostor: The Neural Impostor representation enables us to sustain consistent, high-quality rendering amidst deformation. This is applicable even for objects exhibiting intricate volumetric appearances.

engine with the original tetrahedral mesh as a proxy for the plastic fracture simulation.

## 6.2. Content Authoring with Neural Impostors

In addition to its robust support for vertex-based animation, as shown in Figure 7, the *Neural Impostor* algorithm further enables Boolean appearance editing (e.g., baking pattern colors onto an existing *Neural Impostor* model as shown in Figure 7(d)) and geom-

**Spot Plush** (Synthetic data)**Bear Plush** (Captured data)

**Figure 12: Neural Impostor in Action - Real-Time Deformation of Plush Toys:** Our Neural Impostor technique excels at performing seamless deformation for plush toys, which are notoriously difficult to model using traditional triangle mesh representation due to their fluffy appearances. This advancement enables high-quality, consistent rendering regardless of the object's complexity..

etry editing (e.g., sculpting the original *Neural Impostor* as shown in Figure 7(c)) through local retraining operations described in Section 4. By using explicit tetrahedral meshes as proxies, we can effortlessly combine multiple *Neural Impostor* models to create new scenes, offering the possibility of using *Neural Impostor* as a novel modeling primitive. Figure 14 illustrates the process of creating a new *Neural Impostor* asset from material spheres and three existing *Neural Impostor* models as shown in Figure 14(a) through appearance mapping (snowman body), composition (hat, nose, eyes, and other decorations), and geometry editing (snowman arms). Specifically, starting from material spheres representing the appearance of a snowball and wooden arms, we transfer the appearance information represented by their materials to the snowman body and arms, represented by a simple spherical geometry, through local retraining. Additionally, we elongate the arm vertices along the axis to assemble the snowman body. By combining the constructed body parts with *Neural Impostor* representing the hat and nose, we create a complete snowman model as shown in Figure 14(b). It is worth noting that, apart from the retraining step involved in appearance mapping, this modeling process aligns completely with traditional 3D modeling software workflows. After modeling, the snowman remains a new *Neural Impostor* model, allowing seamless integration with subsequent animation simulations. Figure 14(c) illustrates the process of soft body simulation applied to the snowman model.

The above example is just one of many possibilities. In practical applications, the *Neural Impostor* technology can be used to create various 3D models, including animals, buildings, vehicles, and more. Additionally, inspired by the construction of neural radiance fields from multi-view images, artists can start appearance modeling directly from real captured images, simplifying the material design process in traditional modeling. In a nutshell, the *Neural Impostor* technology provides a brand new workflow for authoring complex models with high quality volumetric appearance.

## 7. Conclusion

In this article, we have presented *Neural Impostor*, a hybrid neural representation. *Neural Impostor* addresses the challenges of editability in neural radiance fields, providing a novel approach to modeling and editing of NeRF. Unlike previous methods, *Neural Impostor* employing an explicit tetrahedral mesh and locally encoded radiance fields to achieve high-quality modeling and rendering. The barycentric-based sampling method ensures rendering quality and efficiency, while the proposed local retraining method allows for fine-grained simulations and editing. We also demonstrated its applications in various content authoring processes, including physical simulation, geometric editing and model composition. Meanwhile, there are some aspects that can be improved. Currently, *Neural Impostor* only supports geometric editing opera-



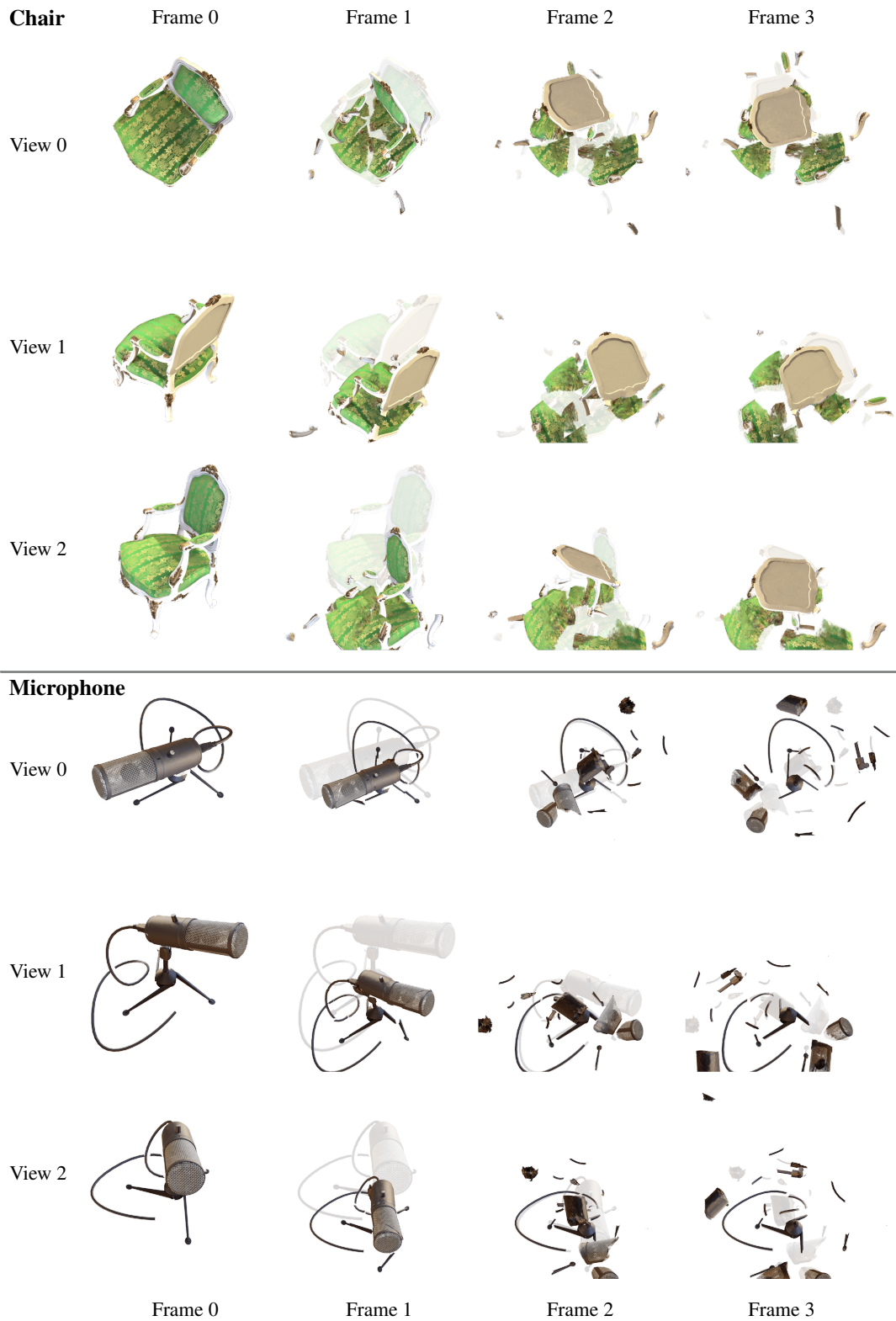
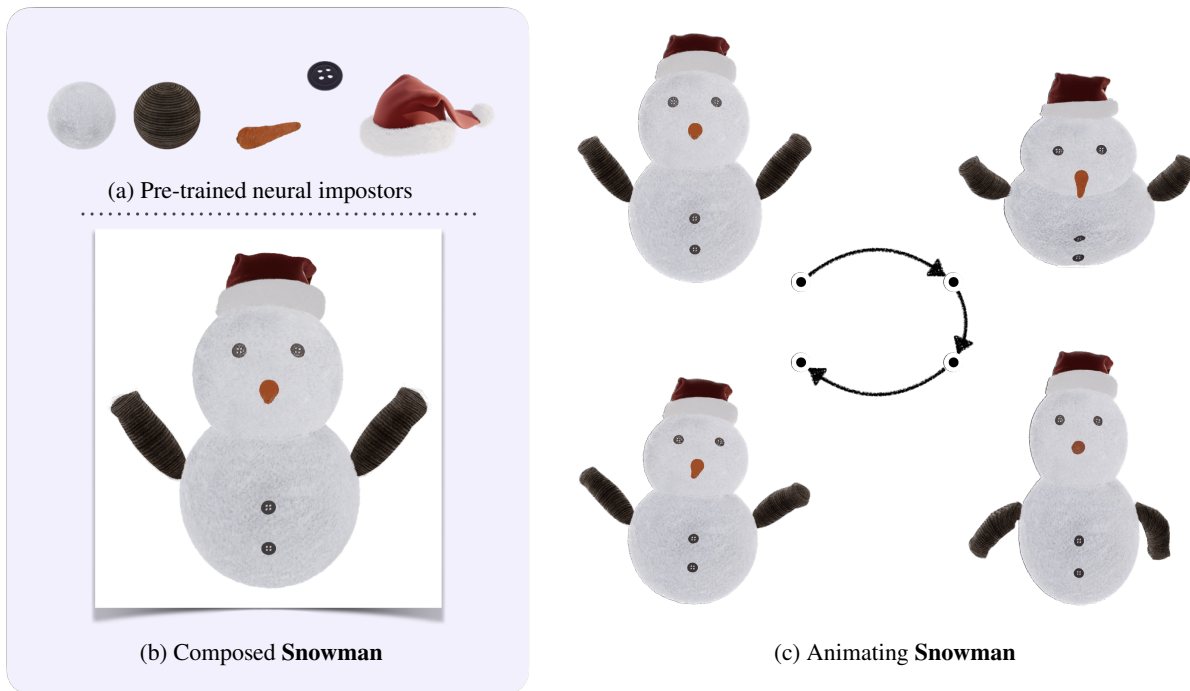


Figure 13: Plastic fracture with Neural Impostor.



**Figure 14:** Authoring and simulation based on Neural Impostors. (a) Pre-trained Neural Impostors as building blocks of a (b) snowman. (c) Animated frames of the snowman model by rendering the composed Neural Impostor.

tions. Extending the range of compatible operations, including re-lighting and material properties, would be some challenging open problems. Overall, *Neural Impostor* introduces new possibilities in neural representation, enhancing the modeling, rendering, and editing capabilities of neural primitives in various applications.

## References

- [CB17] CALDERON S., BOUBEKEUR T.: Bounding proxies for shape approximation. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)* 36, 5 (july 2017). 3
- [CFHT22] CHEN Z., FUNKHOUSER T. A., HEDMAN P., TAGLIASACCHI A.: Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures. *ArXiv abs/2208.00277* (2022). 2, 3
- [CJH\*22] CHONG BAO AND BANGBANG YANG, JUNYI Z., HUJUN B., YINDA Z., ZHAOPENG C., GUOFENG Z.: Neumesh: Learning disentangled neural mesh-based implicit field for geometry and texture editing. In *European Conference on Computer Vision (ECCV)* (2022). 2, 3
- [CLC\*21] CHAN E. R., LIN C. Z., CHAN M. A., NAGANO K., PAN B., MELLO S. D., GALLO O., GUIBAS L., TREMBLAY J., KHAMIS S., KARRAS T., WETZSTEIN G.: Efficient geometry-aware 3D generative adversarial networks. In *arXiv* (2021). 3
- [CLW23] CHEN J., LYU J., WANG Y.-X.: Neuraleditor: Editing neural radiance fields via manipulating point clouds. *ArXiv abs/2305.03049* (2023). 2, 3
- [CXG\*22] CHEN A., XU Z., GEIGER A., YU J., SU H.: Tensorf: Tensorial radiance fields. In *European Conference on Computer Vision* (2022). 3
- [GCX\*20] GAO J., CHEN W., XIANG T., TSANG C. F., JACOBSON A., MCGUIRE M., FIDLER S.: Learning deformable tetrahedral meshes for 3d reconstruction. In *Advances In Neural Information Processing Systems* (2020). 3
- [GKE\*22] GARBIN S. J., KOWALSKI M., ESTELLERS V., SZYMANOWICZ S., REZAEIFAR S., SHEN J., JOHNSON M., VALENTIN J.: Volte-morph: Realtime, controllable and generalisable animation of volumetric representations, 2022. URL: <https://arxiv.org/abs/2208.00949>, doi:10.48550/ARXIV.2208.00949. 2, 3, 8
- [GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: Obbtrees: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, Association for Computing Machinery, p. 171–180. URL: <https://doi.org/10.1145/237170.237244>, doi:10.1145/237170.237244. 3
- [HSM\*21] HEDMAN P., SRINIVASAN P. P., MILDENHALL B., BARRON J. T., DEBEVEC P. E.: Baking neural radiance fields for real-time view synthesis. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), 5855–5864. 3
- [HSW\*20] HU Y., SCHNEIDER T., WANG B., ZORIN D., PANOZZO D.: Fast tetrahedral meshing in the wild. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 117–1. 11
- [HTE\*23] HAQUE A., TANCIK M., EFROS A. A., HOLYSKI A., KANAZAWA A.: Instruct-nerf2nerf: Editing 3d scenes with instructions. *ArXiv abs/2303.12789* (2023). 4
- [Jes05] JESCHKE S.: *Accelerating the Rendering Process Using Impostors*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, Mar. 2005. URL: <https://www.cg.tuwien.ac.at/research/publications/2005/jeschke-05-ARI/>. 3
- [JJK\*23] JAMBON C., KERBL B., KOPANAS G., DIOLATZIS S., LEIMKÜHLER T., DRETTAKIS G.: Nerfshop. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 6* (2023), 1 – 21. 2, 3

- [JMD\*07] JOSHI P., MEYER M., DEROSE T., GREEN B., SANOCKI T.: Harmonic coordinates for character articulation. *ACM Transactions on Graphics (TOG)* 26, 3 (2007), 71–es. 4
- [JSW05] JU T., SCHAEFER S., WARREN J.: Mean value coordinates for closed triangular meshes. In *ACM Siggraph 2005 Papers*. ACM New York, NY, USA, 2005, pp. 561–566. 4
- [JW02] JESCHKE S., WIMMER M.: Textured depth meshes for real-time rendering of arbitrary scenes. In *Rendering Techniques* (2002), pp. 181–190. 3
- [KHM\*98] KLOSOWSKI J., HELD M., MITCHELL J., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics* 4, 1 (1998), 21–36. doi:10.1109/2945.675649. 3
- [KLB\*22] KUANG Z., LUAN F., BI S., SHU Z., WETZSTEIN G., SUNKAVALLI K.: Palettnerf: Palette-based appearance editing of neural radiance fields. *ArXiv abs/2212.10699* (2022). 3
- [KYK\*21] KANIA K., YI K. M., KOWALSKI M., TRZCIŃSKI T., TAGLIASACCHI A.: Conerf: Controllable neural radiance fields. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021), 18602–18611. 3
- [LGL\*20] LIU L., GU J., LIN K. Z., CHUA T.-S., THEOBALT C.: Neural sparse voxel fields. *NeurIPS* (2020). 3
- [LZZ\*21] LIU S., ZHANG X., ZHANG Z., ZHANG R., ZHU J.-Y., RUSSELL B. C.: Editing conditional radiance fields. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), 5753–5763. 3
- [Max95] MAX N. L.: Optical models for direct volume rendering. *IEEE Trans. Vis. Comput. Graph.* 1 (1995), 99–108. 2
- [MESK22] MÜLLER T., EVANS A., SCHIED C., KELLER A.: Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics (TOG)* 41 (2022), 1–15. 3, 5, 6, 11, 13
- [MOB\*21] MEISTER D., OGAKI S., BENTHIN C., DOYLE M. J., GUTHE M., BITTNER J.: A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum* (2021), vol. 40, Wiley Online Library, pp. 683–712. 3
- [MPE\*23] MENDIRATTA M., PAN X., ELGHARIB M. A., TEOTIA K., MALLIKARIJUNB. R., TEWARI A. K., GOLYANIK V., KORTYLEWSKI A., THEOBALT C.: Avatarstudio: Text-driven editing of 3d dynamic human head avatars. 4
- [MST\*20] MILDENHALL B., SRINIVASAN P. P., TANCİK M., BARRON J. T., RAMAMOORTHY R., NG R.: Nerf: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision* (2020). 2, 3, 4, 5, 7
- [PYL\*22] PENG Y., YAN Y., LIU S., CHENG Y., GUAN S., PAN B., ZHAI G., YANG X.: Cagenerf: Cage-based neural radiance field for generalized 3d deformation and animation. In *Advances in Neural Information Processing Systems* (2022), Koyejo S., Mohamed S., Agarwal A., Belgrave D., Cho K., Oh A., (Eds.), vol. 35, Curran Associates, Inc., pp. 31402–31415. URL: [https://proceedings.neurips.cc/paper\\_files/paper/2022/file/cb78e6b5246b03e0b82b4acc8b11cc21-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2022/file/cb78e6b5246b03e0b82b4acc8b11cc21-Paper-Conference.pdf). 2, 3, 8
- [TET\*22] TAKIKAWA T., EVANS A., TREMBLAY J., MÜLLER T., MCGUIRE M., JACOBSON A., FIDLER S.: Variable bitrate neural fields. In *ACM SIGGRAPH 2022 Conference Proceedings* (New York, NY, USA, 2022), SIGGRAPH '22, Association for Computing Machinery. URL: <https://doi.org/10.1145/3528233.3530727>, doi:10.1145/3528233.3530727. 3
- [TTM\*22] TEWARI A., THIES J., MILDENHALL B., SRINIVASAN P., TRETSCHEK E., YIFAN W., LASSNER C., SITZMANN V., MARTIN-BRUALLA R., LOMBARDI S., SIMON T., THEOBALT C., NIESSNER M., BARRON J. T., WETZSTEIN G., ZOLLHÖFER M., GOLYANIK V.: Advances in neural rendering. *Computer Graphics Forum* 41, 2 (2022), 703–735. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14507>, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14507, doi:https://doi.org/10.1111/cgf.14507. 3
- [TWN\*23] TANCİK M., WEBER E., NG E., LI R., YI B., KERR J., WANG T., KRISTOFFERSEN A., AUSTIN J., SALAHİ K., AHUJA A., MCALLISTER D., KANAZAWA A.: Nerfstudio: A modular framework for neural radiance field development. *arXiv preprint arXiv:2302.04264* (2023). 10, 11
- [WRB\*23] WAN Z., RICHARDT C., BOŽIĆ A., LI C., RENGARAJAN V., NAM S., XIANG X., LI T., ZHU B., RANJAN R., LIAO J.: Learning neural duplex radiance fields for real-time view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2023), pp. 8307–8316. 3
- [WUM\*19] WALD I., USHER W., MORRICAL N., LEDIAEV L. M., PASCUCCI V.: Rtx beyond ray tracing: exploring the use of hardware ray tracing cores for tet-mesh point location. *Proceedings of the Conference on High-Performance Graphics* (2019). 7, 8
- [WXB\*23] WEI X., XIANG F., BI S., CHEN A., SUNKAVALLI K., XU Z., SU H.: Neumanifold: Neural watertight manifold reconstruction with efficient and high-quality rendering support, 2023. arXiv:2305.17134. 2, 3
- [XH22] XU T., HARADA T.: Deforming radiance fields with cages. In *ECCV* (2022). 2, 3, 8
- [XXP\*22] XU Q., XU Z., PHILIP J., BI S., SHU Z., SUNKAVALLI K., NEUMANN U.: Point-nerf: Point-based neural radiance fields. *arXiv preprint arXiv:2201.08845* (2022). 2, 3
- [XZCOC12] XU K., ZHANG H., COHEN-OR D., CHEN B.: Fit and diverse: Set evolution for inspiring 3d shape galleries. *ACM Trans. Graph.* 31, 4 (jul 2012). URL: <https://doi.org/10.1145/2185520.2185553>, doi:10.1145/2185520.2185553. 3
- [XZZ\*11] XU K., ZHENG H., ZHANG H., COHEN-OR D., LIU L., XIANG Y.: Photo-inspired model-driven 3d object modeling. *ACM Trans. Graph.* 30, 4 (jul 2011). URL: <https://doi.org/10.1145/2010324.1964975>, doi:10.1145/2010324.1964975. 3
- [YD08] YEE J., DAVIS J.: Crowd rendering with non-planar 3d impostors. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games* (2008). 3
- [YFKT\*21] YU A., FRIDOVICH-KEIL S., TANCİK M., CHEN Q., RECHT B., KANAZAWA A.: Plenoxels: Radiance fields without neural networks. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021), 5491–5500. 3
- [YLT\*21] YU A., LI R., TANCİK M., LI H., NG R., KANAZAWA A.: Plenotrees for real-time rendering of neural radiance fields. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), 5732–5741. 3
- [YSL\*22] YUAN Y.-J., SUN Y.-T., LAI Y.-K., MA Y., JIA R., GAO L.: Nerf-editing: geometry editing of neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2022), pp. 18353–18364. 2, 3, 8, 11, 14
- [ZcLX22] ZHENG C., CHANG LIN W., XU F.: Editablenerf: Editing topologically varying neural radiance fields by key points. *ArXiv abs/2212.04247* (2022). 3
- [ZFCO\*11] ZHENG Y., FU H., COHEN-OR D., AU O. K.-C., TAI C.-L.: Component-wise controllers for structure-preserving shape manipulation. *Computer Graphics Forum* 30, 2 (2011), 563–572. doi:https://doi.org/10.1111/j.1467-8659.2011.01880.x. 3
- [ZJY\*22] ZHAO F., JIANG Y., YAO K., ZHANG J., WANG L., DAI H., ZHONG Y., ZHANG Y., WU M., XU L., ET AL.: Human performance modeling and rendering via neural animated mesh. *arXiv preprint arXiv:2209.08468* (2022). 11