Laboratório VISGRAF
Instituto de Matemática Pura e Aplicada

Real-Time Rendering of Neural Radiance Fields

Thales Magalhaes, Luiz Velho (supervisor)

Technical Report TR-24-01 Relatório Técnico

January - 2024 - Janeiro

Real-Time Rendering of Neural Radiance Fields (Work-in-Progress)

Thales Magalhães

Abstract

This technical report describes a work-in-progress project which aims to implement real-time rendering of neural radiance fields (NeRF) inside the Unity platform. It is primarily based on the method developed by Yu et al. [17] which works by caching the results of the NeRF network using a sparse voxel octree (SVO) structure which can be efficiently queried using GPU acceleration. With our method, scenes containing a combination of NeRF and mesh-based assets are rendered using a hybrid pipeline developed as an extension of an existing, fully-featured raster renderer. Our implementation of this method is based on widely available graphics hardware functionality.

1 Introduction

Mildenhall et al. [10] introduced the concept of neural radiance fields (NeRF) as a method for synthesizing novel views of complex 3D scenes by extrapolating from relatively small set of reference (RGB) images. The technique produced impressive results while addressing a number limitations of previous methods by being capable of reproducing complex, view-dependent effects at high resolutions. It did this by using a continuous, volumetric representation of scenes based on a relatively simple variety of feedforward neural network known as a multilayer perceptron (MLP). These networks could be iteratively optimized to closely match the reference images for each scene using well-known deep learning techniques.

The release of [10] was quickly followed by a number of publications from other researchers. Some aimed at extending the idea of using NeRFs to other contexts, such as large scale scenes [19], video [4, 8, 16], and animation [11, 13]. Others focused on addressing limitations of the original method by improving training times [3], adding support for relighting [1, 6, 15], etc.

Among these there have been many attempts at improving NeRF's inference time and adapting the method to real-time applications [5, 7, 14, 17]. While these methods have been largely successful at achieving interactive framerates, they were focused on the task of rendering individual NeRF scenes. This came at the expense of their capability to render more complex configurations with multiple NeRF scenes or to combine them with other 3D representation formats (i.e.

triangle meshes). Moreover, their implementations often rely on vendor-specific hardware acceleration technology to achieve their advertised performance characteristics, making them inaccessible to a significant portion of users.

With this project we aim to develop an efficient method of rendering NeRF scenes which is compatible with current industry standard real-time rendering techniques. We base our method on Yu et al. [17]'s strategy of caching the results of a modified NeRF network using a sparse voxel octree (SVO) structure. This network is trained to encode view-dependent effects using spherical harmonic coefficients, reducing the size of it's input space (and the resulting cache). NeRF volumes are then rendered through ray-marching (using hardware acceleration) as part of a hybrid rendering pipeline. This pipeline is developed as an extension to an existing rasterization-based pipeline using widely available, vendor-agnostic GPU functionality. We provide a proof-of-concept implementation of our method using the Unity platform.

2 Related Work

Garbin et al. [5] present a novel method (FastNeRF) for accelerating NeRF inference through caching. Caching the output of a vanilla NeRF network would be unfeasible as it would require enough samples to capture the entire 5D input space of the network. To solve this, the authors propose to split this network in two parts: one MLP which captures position-dependent features and another for reconstructing view-dependent effects. The outputs of these two networks are then combined using the inner product to produce RGB color values. Although this factorization scheme is not based on a physically-based rendering model, the authors note that it is inspired by well established rendering techniques (i.e. spherical harmonic lighting). However, despite achieving a significant reduction relative to the theoretical memory requirements for such a cache, often the resulting caches produced by this method are still too large for the average consumer GPU (see Table 1).

Another method based on reducing the input space of NeRF's network and caching the model's results is presented by Hedman et al. [7]. Here, the authors take an approach inspired by deferred shading techniques in which diffuse and specular colors are computed separately. To do this, they train their network (SNeRG) to predict both a diffuse color and a view-independent feature vector for every point in the scene. During rendering, a separate decoder network is used to predict specular colors based on the accumulated the value of these feature vectors. Crucially, this step is only performed once per ray. The results of this network are then sampled and cached using a custom sparse voxel grid data structure.

Yu et al. [17] describe a similar strategy for achieving real-time inference through caching: They make the input to their model view-independent by training a new network (NeRF-SH) which encodes view-dependent effects using a fixed number of spherical harmonic coefficients. This modification reduces the input space of the network such that caching it requires only $O(n^3)$ sam-

ples instead of $O(n^5)$ (where the resolution of the cache is given by n). These samples are then stored using a sparse voxel octree structure. Section 3.1 and Appendix A cover aspects of this method's implementation in more detail.

Reiser et al. [14] (KiloNeRF) take a rather different approach by partitioning the monolithic NeRF network into thousands of smaller MLPs, each of which is responsible for encoding a small part of the scene. By doing this, the number of floating point operations required to compute each sample along the ray is considerably reduced. Despite this seemingly simple modification to the original method, the implementation of Reiser et al.'s method requires batching of matrix multiplications on the GPU to achieve the desired performance. For this purpose the authors rely on third-party libraries (i.e. MAGMA) to dynamically load matrices during inference time in an efficient manner. As a consequence of this, their implementation is dependent on Nvidia's CUDA ecosystem.

It is difficult to draw a direct comparison between each of these methods since there's often a trade-off to be made between image quality, render time, and memory usage. Some authors even provide results for multiple variants of their method, each parameterized to prioritize a different performance characteristic. Furthermore, while some authors provide exact measurements for these properties, others only offer rough estimates (this is particularly true for memory/storage requirements). Keeping these caveats in mind, readers may refer to Table 1 for a summary of the memory/storage requirements of each method described.

Method	**Memory (MB)** $\downarrow$	**File Size (MB)** $\downarrow$
FastNeRF	340–16,200*	—
KiloNeRF	<100	—
PlenOctrees	300–1,900	—
SNeRG	6,900	30–6,900 MB

Table 1: **Comparison of memory/storage requirements.** When multiple variants of the same method exist, the range of reported values is presented. Since an exact value was not provided for the size of FastNeRF's largest cache (1024^3), an optimistic approximation was used instead (*). Hedman et al. (SNeRG) provide a range of possible storage requirements for their cache depending on the file format used for compression (PNG, JPEG, or H.264). Separate values for memory/storage requirements were not provided for any of the other methods.

Similarly, Table 2 compares the performance characteristics of the different methods in terms of render throughput in frames per second (FPS). Timings for each are reproduced as reported by their respective authors (modulo unit conversions). Since the hardware used to obtain these benchmarks was not uniform, performance benchmarks for the vanilla NeRF model were also included for reference.

		FPS ↑		
Method	**GPU**	**Baseline**	**Method**	**Speedup ↑**
FastNeRF	RTX 3090	0.06*	172.4	2873
KiloNeRF	GTX 1080Ti	0.018	38.46	2137
PlenOctrees	Tesla V100	0.023	167.68	7290
SNeRG	MacBook Pro	0.03	84.06	2802

Table 2: **Rendering performance benchmarks.** Column "baseline" represents the frame rate for the vanilla NeRF model using the same hardware. Results measured using the NeRF synthetic dataset [9] at a resolution of 800×800 (with the exception of FastNeRF's baseline, which was measured using only the LEGO scene).

3 Method

In this section we describe our method for extending an existing, rasterization-based rendering pipeline to render NeRF scenes. We developed this method based on the needs of our specific implementation, but we describe the techniques used in general terms whenever possible.

In particular our rendering strategy was designed around Unity's forward rendering pipeline, but the techniques presented here should be general enough to be adaptable to the pipelines of other real-time rendering engines. Since engines like this are primarily designed for working with 3D meshes, we based our method on using these primitives to represent NeRF assets within a scene.

3.1 Dataset

We source our data from [18], the dataset released by Yu et al. along with their paper. This dataset contains the NeRF-SH models used in the paper (which were trained using NeRF's LLFF/synthetic dataset [9]) and their corresponding PlenOctrees.

Yu et al. store their PlenOctrees data using a custom N^3-tree structure (see Appendix A), for which they provide a Python interface. This interface is implemented as a PyTorch CUDA extension and distributed through GitHub. They serialize this structure to disk using the `.npz` format – a custom binary serialization format implemented by the NumPy package. A single `.npz` file consists of multiple `.npy` files (each representing a single NumPy array) compressed together as a ZIP archive. Since the NumPy library only provides official bindings for Python, the serialization/deserialization of this data from other languages can be quite cumbersome.

We chose to convert this data into a simpler SVO representation (Section 3.2) and serialize it using a more portable format (Section 3.3) so it may be easier to work with in other environments. Because we ran into hardware compatibility issues when trying to use their Python package (as it relies on specific versions

of Nvidia's CUDA driver), we instead use a stripped-down fork of their N^3-tree implementation during the conversion process.

3.2 The Sparse Voxel Octree Structure

In this section we describe the sparse voxel octree (SVO) structure we use to store and query PlenOctree data. This data structure acts as a bridge between our Python scripts, the Unity .NET runtime, and shaders on the GPU; as such, implementations were developed for Python, C#, and HLSL (with varying degrees of functionality).

Our octrees are parameterized with a fixed resolution (W, H, D) upon initialization. Voxels are indexed using integer triplets (x, y, z) where $0 \leq x < W$, $0 \leq y < H$, and $0 \leq z < D$. We adopt the convention of a right-handed XYZ coordinate system. This is unlike Unity's native coordinate system which is left-handed, therefore care must be taken to consistently convert between coordinate systems when working within the engine (particularly in shader code).

Octants are sorted in colexicographic order with $(-, -, -)$ being considered the first octant. This is is equivalent to the memory layout of a C multidimensional array of the form `A[2][2][2]` when indexed using the expression `A[z > 0][y > 0][x > 0]`. This scheme results in the following octant indices:

x	y	z	**Index**
$-$	$-$	$-$	0
$+$	$-$	$-$	1
$-$	$+$	$-$	2
$+$	$+$	$-$	3

x	y	z	**Index**
$-$	$-$	$+$	4
$+$	$-$	$+$	5
$-$	$+$	$+$	6
$+$	$+$	$+$	7

Octree traversal always starts from the root of the structure, moving down the leaves. This method has been performant enough to render the scenes tested in real time, though there are likely more efficient traversal algorithms given the predictable access patterns of the ray marching algorithm (in particular, it might be possible to take advantage of the fact that samples are highly correlated since they're taken along a ray).

Voxels are sampled using the nearest-neighbor method, though we plan on implementing bilinear and trilinear interpolation in the future. As of now it is unclear if the quality benefits of using these more expensive sampling methods are worth their additional runtime cost.

3.3 Serialization

Our main objective when developing this structure was to have good compatibility across different platforms. Ideally, creating new implementations for different languages/systems should be as straightforward as possible. To this end, our serialization format should be well supported in many different environments. After comparing a selection of different serialization protocols (see Appendix B) we landed on Protocol Buffers as the most appropriate format for serializing

```
class SparseVoxelOctree<T>
{
        public readonly int Width;
        public readonly int Height;
        public readonly int Depth;

        private List<int> _nodeChildren;
        private List<T> _nodeData;

        // ...
}
```

Listing 1: **SVO structure in C#.** Implemented as a generic class. The structure of the octree is encoded as a list of $8n$ integers (`_nodeChildren`). Within this list, nodes are represented as blocks of 8 integers containing the indices of their children. A second list (`_nodeData`) with n entries of generic type `T` stores the data for each node.

```
struct SparseVoxelOctree {
    int width;
    int height;
    int depth;
    int dataDim;
    int maxLevel;
    StructuredBuffer<int> nodeChildren;
    StructuredBuffer<float> nodeData;
};
```

Listing 2: **SVO structure in HLSL.** Follows the same memory layout described in Listing 1. Unlike the C# implementation, it is specialized for floating point numbers. Each node stores a fixed number (`dataDim`) of floats representing the SH coefficients for that voxel.

this data structure. As a binary serialization format, it relies on a specification (in the form of a `.proto` file) written in a custom schema description language to parse the contents of the data structure. Listing 3 shows the specification used for our data structure.

```
syntax = "proto3";
package svo.protobuf;
option csharp_namespace = "SVO.Protobuf";

message SparseVoxelOctree {
    string type_url = 1;
    int32 width = 2;
    int32 height = 3;
    int32 depth = 4;
    repeated int32 node_children = 5 [packed=true];
    bytes node_data = 6;
}
```

Listing 3: **SVO structure schema (`.proto` file).** To support arbitrary data types, we store voxel data in binary form as a `bytes` array. The `type_url` field is an URL pointing to a separate `.proto` file describing the data stored per voxel. Because Protobuf assigns fixed URLs to basic types it considers "well-known" (e.g. booleans, integers, floats, etc.), a separate `.proto` is not necessary for those types.

3.4 Rendering Strategy

Like most real-time 3D rendering engines, Unity's rendering pipeline is based on rasterization using the Z-buffer algorithm. This is because, even with recent advancements in hardware accelerated ray-tracing, out-of-order rendering still provides the best performance out of modern GPU architectures. Because of this we represent our NeRF assets using 3D meshes (e.g. a box, sphere, etc.) which act as a convex hull around the contents of the NeRF.

Each of these meshes implicitly defines a volume. We render these volumes using a custom ray-marching algorithm implemented using programmable pixel shaders. First, the SVO representing the NeRF scene is loaded into GPU memory; We then render the NeRF's hull to determine which parts of the image are occupied by the NeRF: For every camera ray intersecting the NeRF's volume (i.e. for every pixel not culled during the last step), we perform ray-marching by sampling the SVO along the ray using custom vertex and fragment shaders; Finally, the resulting color, transmittance, and depth values are used to generate the final image.

Our method uses a mixture of transmittance and depth to composite NeRF volumes with the rest of the 3D scene. During the ray-marching procedure, we estimate the depth of each ray by computing the expected ray termination distance $E[d]$ and use this value (along with the Z-buffer) to discard fragments that would otherwise be occluded by the environment. We then use the ray's total transmittance value to add it's contribution to the framebuffer using alpha blending.

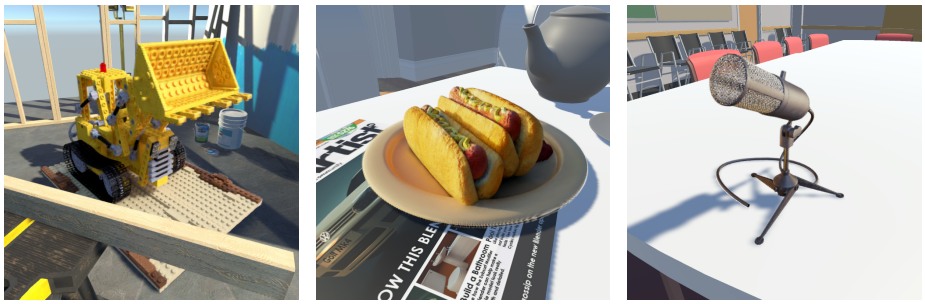
4 Results and Conclusion

Our method is capable of rendering NeRFs in real time as part of complex scenes along with more traditional, mesh-based 3D assets. As Table 3 shows, our method achieves performance that is comparable to state of the art methods while using consumer grade hardware. We do this using widely available hardware acceleration functionality (without relying on vendor-specific features) by basing our method on portable, well-established rendering techniques. We hope this is a step in making NeRFs more accessible to a larger audience by enabling their adoption in contexts where this was previously not practical.

Figure 1 shows the qualitative results of our proof-of-concept implementation of this method using the Unity engine for a selection of sample scenes. The figure showcases some the advantages offered by the hybrid approach used by the method: Notice, for example, how NeRF objects interact with the rest of the scene by casting shadows over the environment despite the fact that these elements use different rendering strategies (rasterization vs. ray-marching).

Scene	**Cache Size**	**FPS** $\uparrow$
Chair	128^3	317.9
LEGO	128^3	205.0
	256^3	153.5
Hot dog	128^3	204.5
	256^3	156.3
Microphone	128^3	280.0
	256^3	214.9

Table 3: **Method performance benchmarks.** Results measured using the Unity Editor's preview mode at a resolution of 800×800. Some variance is expected depending on the virtual camera's angle relative to the NeRF scene. Performance is also roughly proportional to the amount of screen space covered by the NeRF's bounding volume. Hardware used: Ryzen 5 5600 (CPU), RX 6700 XT (GPU).



(a) (b) (c)

Figure 1: **Qualitative results of our method** for the LEGO (a), hot dog (b), and microphone (c) scenes using a cache size of 256^3. Background environments for each example were built using textured 3D meshes.

5 Future Work

5.1 Memory Layout of the SVO Structure

Currently our implementation stores the list of child indices for every single node in the octree. This is unnecessary, however, since we know that the nodes at the last level of the octree can't possibly have any descendants. Not storing these nodes could save a significant amount of space since in practice the overwhelming majority of nodes in the octree are leaf nodes. This would require a pre-processing step to sort the nodes of the octree based on their depth in the graph. As a bonus, performing this transformation should make the nodes of each layer contiguous in memory, which could improve performance since this layout would have better data locality. This layout would also simplify memory management for level-of-detail since reducing LOD level would require simply trimming off the last few entries in the nodes list.

Also, as of now our implementation stores PlenOctree data using the same layout as the original `N3Tree` data structure. This means that, for example, when using the `SH16` format each voxel contains an array of 49 floating-point values corresponding to the 16 SH coefficients for each color channel (plus the density). The particular order for these coefficients within the array is determined entirely by convention. This is unfortunate since it requires developers to be familiar the layout of these coefficients when using the data structure. However, since our implementation is generic on type, we have the capability of storing SH coefficients using a custom data structure – which could make the sampling process considerably less error-prone than it currently is. Although our serialization format of choice makes storing and loading custom data types a bit of a hassle, tweaking it to support this wouldn't be too difficult (more information in Section 3.3).

5.2 Alternative Rendering Strategies

While our current rendering strategy produces accurate results in most situations, it can nevertheless be inaccurate in some edge cases. Since color and transmittance are computed by ray-marching across the entire volume, this can result in physically incorrect results when the volume is occupied by other objects (such as opaque meshes).

This issue could be avoided by rendering NeRFs during the transparency queue of the render pipeline (i.e. after all opaque geometry). In this case, the Z-buffer could be used to determine how far a ray can travel before intersecting with the environment. This step would ensure we compute more physically accurate color and transmittance values. As before, the resulting color would be computed by adding the ray's contribution to the final image using alpha blending.

This strategy, however, suffers from some of the limitations inherent to the rasterization of transparent objects using alpha blending, such as the need to sort primitives based on distance to the camera and inaccuracies when rendering intersecting primitives.

For NeRF assets representing opaque (or nearly opaque) objects, it might be possible to skirt these limitations without much visual impact by avoiding the alpha-blending step altogether. This could be done by discarding fragments based on a transmittance threshold set by the user: rays that that have a transmittance value below the threshold are considered fully opaque, while rays that have a transmittance above that are considered fully transparent. This avoids the requirement to render NeRF primitives during the transparency queue, allowing them to be rendered along with opaque geometry (in practice, this material would be rendered during Unity's "transparent cutout" queue).

References

[1] Mark Boss et al. *NeRD: Neural Reflectance Decomposition from Image Collections*. 2021. arXiv: 2012.03918 [cs.CV].

[2] c80k and contributors. *c80k/capnproto-dotnetcore: A Cap'n Proto implementation for .NET Standard and .NET Core*. URL: https://github.com/c80k/capnproto-dotnetcore (visited on 10/25/2023).

[3] Kangle Deng et al. *Depth-supervised NeRF: Fewer Views and Faster Training for Free*. 2022. arXiv: 2107.02791 [cs.CV].

[4] Yilun Du et al. *Neural Radiance Flow for 4D View Synthesis and Video Processing*. 2021. arXiv: 2012.09790 [cs.CV].

[5] Stephan J. Garbin et al. *FastNeRF: High-Fidelity Neural Rendering at 200FPS*. 2021. arXiv: 2103.10380 [cs.CV].

[6] Michelle Guo et al. "Object-Centric Neural Scene Rendering". In: *arXiv preprint arXiv:2012.08503* (2020).

[7] Peter Hedman et al. *Baking Neural Radiance Fields for Real-Time View Synthesis*. 2021. arXiv: 2103.14645 [cs.CV].

[8] Zhengqi Li et al. *Neural Scene Flow Fields for Space-Time View Synthesis of Dynamic Scenes*. 2021. arXiv: 2011.13084 [cs.CV].

[9] Ben Mildenhall et al. *NeRF synthetic Blender data and LLFF scenes (Google Drive)*. URL: https://drive.google.com/drive/folders/128yBriW1IG_3NJ5Rp7APSTZsJqdJdfc1 (visited on 01/02/2024).

[10] Ben Mildenhall et al. *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis*. 2020. arXiv: 2003.08934 [cs.CV].

[11] Atsuhiro Noguchi et al. *Neural Articulated Radiance Field*. 2021. arXiv: 2104.03110 [cs.CV].

[12] Jason Paryani and Jacob Alexander. *pycapnp — capnp 1.0.0 documentation*. URL: https://capnproto.github.io/pycapnp (visited on 10/25/2023).

[13] Albert Pumarola et al. "D-NeRF: Neural Radiance Fields for Dynamic Scenes". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021.

[14] Christian Reiser et al. *KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs*. 2021. arXiv: 2103.13744 [cs.CV].

[15] Pratul P. Srinivasan et al. *NeRV: Neural Reflectance and Visibility Fields for Relighting and View Synthesis*. 2020. arXiv: 2012.03927 [cs.CV].

[16] Wenqi Xian et al. *Space-time Neural Irradiance Fields for Free-Viewpoint Video*. 2021. arXiv: 2011.12950 [cs.CV].

[17] Alex Yu et al. *PlenOctrees for Real-time Rendering of Neural Radiance Fields*. 2021. arXiv: 2103.14024 [cs.CV].

[18] Alex Yu et al. *Trained NeRF-SH models and converted plenoctrees (Google Drive)*. URL: https://drive.google.com/drive/folders/1J0lRiDn_wOiLVpCraf6jM7vvCwDr9Dmx (visited on 10/25/2023).

[19] Kai Zhang et al. *NeRF++: Analyzing and Improving Neural Radiance Fields*. 2020. arXiv: 2010.07492 [cs.CV].

A The N^3-Tree Structure

An N^3-tree is a spatial partitioning data structure similar to a sparse voxel octree. N^3-trees are parameterized with an integer N which determines the number of subdivisions (per dimension) at every level of the tree. In other words, each internal node in an N^3-tree contains at most N^3 children (as opposed to 8 in an octree). When $N = 2$ the two structures are functionally equivalent.

Yu et al.'s version of the N^3-tree (simply called an `N3Tree`) is implemented as a PyTorch extension and supports hardware acceleration using CUDA. The authors use PyTorch for the fine tuning step of their method, where they further optimize each PlenOctree using differentiable rendering. They wrote a custom CUDA kernel which performs queries in batches on the GPU; This kernel implements a straightforward traversal strategy by always searching the tree from the root down. Sampling uses the nearest-neighbor method (i.e. no interpolation is performed between neighboring voxels).

In this implementation, N^3-trees are also parameterized with an integer (`depth_limit`) which bounds the size of its internal graph structure. This value effectively limits the maximum resolution achievable within the structure. All voxels are considered to be located inside the unit cube, with samples being indexed using floating-point coordinates (x, y, z) (where $x, y, z \in [0, 1]$). Although the structure doesn't enforce a particular coordinate scheme, the scenes from [18] all use a right-handed XYZ coordinate system.

The structure is primarily designed to store floating-point data in the form of vectors. To this end, it is parameterized with an integer `data_dim` determining how many floating-point values are stored at each node. Internally, the structure mostly consists of two NumPy tensors: `self.child` and `self.data`. The first tensor (`self.child`) has a dimension of (m, N, N, N) where m denotes the number of nodes in the octree. It is used to represent the tree's structure: For each node i, `self.child[i]` contains a (N, N, N)-tensor with the indices of each of its children. The second tensor (`self.data`) has a dimension of $(m, N, N, N,$ `data_dim`) and is used to store the data pertaining to each node. In both cases octants within a node are indexed in lexicographical order (as in the Python expression `self.child[i,x,y,z]`).

Note that the structure stores data *per node*, not per voxel. This means it is capable of storing data not only in the leaves, but also in the intermediate nodes. Though the authors don't seem to take advantage of this feature in their rendering algorithm, it could be used for implementing level-of-detail and/or mipmapping.

Each voxel stores a `data_dim`-dimensional vector of floating-point values. Since no particular layout is enforced for these values, care must be taken to establish a consistent order when reading/writing to these vectors. When storing NeRF-SH data, spherical harmonic coefficients are grouped by color. Within each color coefficients are sorted by spherical harmonic order, then by degree. This results in the following sequence of coefficients:

$$f_0^0, f_1^{-1}, f_1^0, f_1^1, f_2^{-2}, f_2^{-1}, f_2^0, f_2^1, f_2^2, \ldots$$

Where f_ℓ^k is the coefficient corresponding to the spherical harmonic Y_ℓ^k of order ℓ and degree k. The vector's last value always represents the density σ.

B Comparison of Data-Serialization Formats

JSON: Easily the most popular and widely supported serialization format analysed. It has been designed to resemble JavaScript syntax, making it human-readable and somewhat intuitive (at least to those with some programming experience). Unfortunately, this makes it very inefficient in terms of storage and encoding/decoding performance.

BSON: A binary interchange format based on JSON designed to be more efficient in terms of both storage and encoding/decoding performance (particularly scan speed). However, our experiments show that BSON files have an even larger storage footprint than JSON when used for storing our SVO structure.

Binary Formatter: The C# standard library features its own binary serialization format called `BinaryFormatter`. Although it is widely supported within the .NET ecosystem, it is not cross-platform and thus difficult to work with in other contexts.

Protocol Buffers: Also known as Protobuf. A binary serialization format developed by Google. Designed to be efficient, cross-platform, and easy to use. Unlike JSON, Protobuf is designed for structured data – that means encoding/decoding rely on pre-defined data schemas supplied by the programmer in the form of `.proto` files. These files are written in a custom schema specification language.

Flatbuffers: Yet another binary format for structured data, also developed by Google. It is meant to address some of the shortcomings of Protobuf in terms of performance, particularly when working with larger collections of data or when partial decoding of records is required. Our experiments show that it is more efficient than Protobuf in terms of storage size, but encoding speed suffers considerably (however this could be an issue specific to the Python library used; further experiments might be necessary).

Cap'n Proto: A serialization protocol created by the former maintainer of the Protocol Buffers library. It was designed to improve on some of Protocol Buffers' shortcomings by including, for example, deeper support for generic and dynamic types. Unofficial community implementations of the protocol are available for a number of languages; however these implementations tend to be less mature than those of Protocol Buffers and Flatbuffers due to the fact that Cap'n Proto has a considerably smaller community.

Format	Binary Encoding	Python Support	C# Support	Packed Arrays	Generic Types
JSON	·	✓	✓	·	✓
BSON	✓	✓	✓	·	✓
BinaryFormatter	✓	·	✓	✓	✓
Protobuf	✓	✓	✓	✓	✓[1]
FlatBuffers	✓	✓	✓	✓	·
Cap'n Proto	✓	·[2]	·[2]	✓	✓

Table 4: **Serialization format features.** Note that while the Protobuf specification defines standardized representations for generic types, the feature is not well supported across implementations (1). Also, despite the fact that Cap'n Proto doesn't provide official implementations for Python or C#, unofficial support is available to some degree through community-maintained packages (2).

With the exception of C#'s `BinaryFormatter`, all of the the data serialization formats surveyed are language-agnostic. However, their authors often only provide official implementations for a small selection of languages. This is true for Cap'n Proto, which provides no official implementation for Python or C#. However, unofficial support is available through the community-maintained packages `pycapnproto` [12] and `capnproto-dotnetcore` [2].

Most formats surveyed support the serialization of generic data types to some degree. In JSON, for example, containers (i.e. lists or dictionaries) may contain arbitrary data since the format is schema-less. As a consequence of this, the onus lies on the developer (rather than the library) to validate that the data received is correct. On the other hand, formats which enforce a fixed schema must rely on other mechanisms to provide the same functionality.

In the case of Protocol Buffers arbitrary data can be serialized using the `Any` type, which tags binary blobs of data using a separate URL field. Unfortunately, this feature is not well supported across implementations. Additionally, using `Any` requires list entries to be individually tagged, defeating the purpose of packed lists. A workaround for this issue is to represent lists of generic values using the `bytes` type and store an URL describing the type as a separate field of the message. This is what we do in practice, although it is not ideal.

Format	Encoding Time (s) ↓	File Size (MB) ↓
JSON	35.5	244.8
BSON	18.0	421.0
BinaryFormatter	—	—
Protobuf	4.4	120.0
FlatBuffers	60.0	96.0
Cap'n Proto	6.8	96.0

Table 5: **Serialization format benchmarks.** Results measured in Google Colab using the free-tier Python 3 Google Compute Engine backend. Timings were taken using the LEGO scene from [18]. This scene was converted from its original `N3Tree` representation into our custom SVO format by resampling it at a resolution of 128^3 voxels. Results for the JSON format were measured using Python's built-in `json` package.