Instant Neural Representation for Interactive Volume Rendering

Qi Wu, Michael J. Doyle, David Bauer and Kwan-Liu Ma

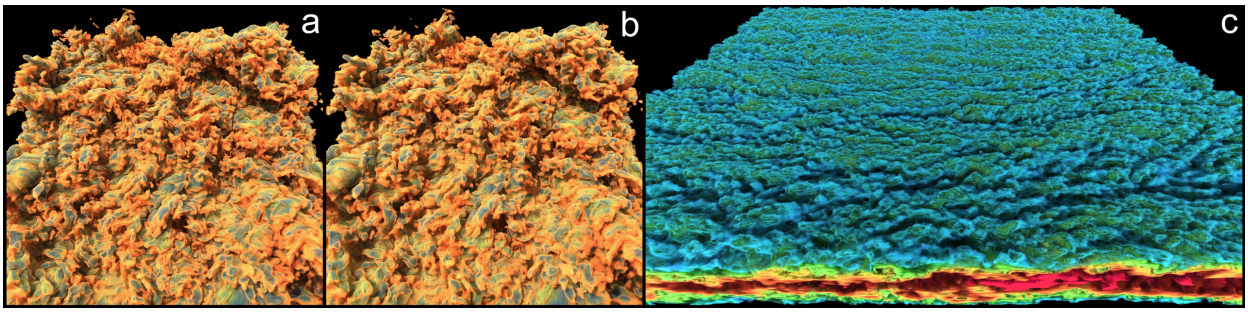

Fig. 1. a) Ground truth rendering of the `1atmHR` data. b) Rendering the same data using our instant volumetric neural representation. c) Out-of-core online-training and rendering of the 0.95TB DNS dataset. All the renderings are done using volumetric path tracing (with maximum 7 bounces per ray, 1 ray per pixel, and 1 directional light in the scene).

Abstract—Neural networks have shown great potential in compressing volumetric data for scientific visualization. However, due to the high cost of training and inference, such volumetric neural representations have thus far only been applied to offline data processing and non-interactive rendering. In this paper, we demonstrate that by simultaneously leveraging modern GPU tensor cores, a native CUDA neural network framework, and online training, we can achieve high-performance and high-fidelity interactive ray tracing using volumetric neural representations. Additionally, our method is fully generalizable and can adapt to time-varying datasets on-the-fly. We present three strategies for online training with each leveraging a different combination of the GPU, the CPU, and out-of-core techniques. We also develop three rendering implementations that allow interactive ray tracing to be coupled with real-time volume decoding, sample streaming, and in-shader neural network inference. We demonstrate that our volumetric neural representations can scale up to terascale for regular-grid volume visualization, and can easily support irregular data structures such as OpenVDB, unstructured, AMR, and particle volume data.

Index Terms—Neural network, Online learning, Interactive visualization, Volume rendering

1 Introduction

Modern simulations and experiments can produce massive amounts of high-fidelity data given continuous advancements in computation and data acquisition technologies. These data can provide critical details for understanding complex physical phenomena and chemical processes but can be challenging to visualize interactively without using purpose-built hardware such as a PC cluster. One approach to better handle such large datasets is *volume compression*.

Although much work on volume compression using more traditional techniques can be found in the existing literature [1,7,11,23,38,42,45], a recent interesting development in this space is the use of novel deep learning-based compression methods. For example, recent work by Lu et al. [27] introduces a promising volume compression method based on implicit neural representations [40], in which a multilayer perceptron (MLP) is constructed to approximate a volumetric function by overfitting it to the corresponding dataset. However, since the proposed neural network architecture is fairly complex[1], the cost of training and inference can be substantial. Although the performance of this method was substantially improved by Weiss et al. [50] (via smaller MLPs, frequency-based input encodings, and a tensor-core-accelerated CUDA inference kernel), the current methods still require pre-training of the order of minutes or hours before high-quality visualization can begin. Since such implicit neural representations can only be trained for a single dataset at a time, these methods are applicable only for offline data processing workflows.

Given these limitations, we argue that the potential of such representations for interactive volume visualization has yet to be fully realized. To close this gap, we extend the capabilities of previous work to overcome these limitations and achieve high-performance, high-fidelity interactive volume rendering based on neural data representations without requiring lengthy pre-training. Similar to Lu et al.'s idea, our neural representation is volumetric, as it constructs the mapping from a spatial coordinate to a data value. Since the mapping is defined over a continuous domain, such a representation can be directly sampled arbitrarily at any spatial location without any interpolation. Moreover, since neural networks can analytically extract high-level features from the dataset, our volumetric neural representation does not necessarily require a more complex network to faithfully represent a volume as the volume resolution increases. This property allows our method to scale up to terascale datasets.

For optimal performance, we build our training and inference infrastructure entirely in a native CUDA machine learning framework [32], which leverages the recent work of [33] to greatly improve training times. Furthermore, these large performance improvements make it practical for us to fully embrace online supervised training methods, which allows us to overcome the aforementioned data generalization challenge via fast adaptation. By starting with an empty representation and optimizing it completely online during rendering, our system can respond quickly to different or even dynamic datasets without any pre-processing, while reaching a high-quality visualization within a few seconds on a desktop machine. To achieve interoperability between the machine learning framework and sophisticated ray-traced renderers, we also produce our hardware-accelerated network inference routine that can work seamlessly on both ends. Additionally, with the help of the

- *Qi Wu, David Bauer and Kwan-Liu Ma are with University of California - Davis, USA.E-mail: {qadwu | davbauer | klma}@ucdavis.edu.*
- *Michael J. Doyle is with Intel Corporation, USA.E-mail: michael1.doyle@intel.com.*

[1] 18 layers with 8 residual blocks, each layer with 128 neurons

multi-purpose volume traversal library, OpenVKL [17], we can easily support non regular grid data.

While online training is a key component of our method, it has previously only been used in offline rendering and in a small number of interactive applications [24, 34–36]. This is in part because performing both network training and rendering simultaneously, while still achieving good framerates as well as fast network convergence, is a non-trivial task. Given that online training is yet to be applied in the interactive volume visualization context, we develop and evaluate three strategies for doing so. Our first method, which we call *GPU-based training*, loads the entire data to GPU memory and performs training fully on the GPU. This yields a fast training speed but is limited to datasets that fit entirely in GPU memory. For larger datasets, we develop a second method which we term *CPU-based training*. For this method, the data only needs to fit in the larger CPU memory. We collect training samples from the dataset on the CPU and send these to the GPU to perform the remainder of the training process. Since the number of training samples required at each frame is typically much lower than the number of samples required by a native CPU volume renderer, this method (coupled with our GPU-based neural representation renderer) can significantly outperform a native CPU-based renderer. Our third method, *Out-of-core training*, is useful for volumetric data that is too large to fit even in CPU memory. This method maintains a continuously changing stochastic subset of the volumetric data which can be sampled on each frame for online training.

We also develop three strategies for rendering our volumetric neural representation. *Volume Decoding* is perhaps the simplest and quickest rendering method, as it fully decodes the volume from the neural representation to a traditional grid representation which can then be visualized with a traditional GPU renderer. If the volume cannot fit into the GPU memory at full resolution, a lower resolution grid will be used. Since we are rendering an ever-improving neural representation due to online training, the decoding process is executed for each frame. However, the cost of decoding the entire volume all at once is typically high, so the decoding process is also performed progressively. This strategy is important for us to maintain highly interactive framerates. To achieve volume rendering with better fidelity for datasets that do not fit in GPU memory, we present a second method, *Sample Streaming*, which directly utilizes the neural representation for rendering. In particular, we implement a mechanism to iteratively interrupt a ray marching algorithm, stream out all the sample coordinates for network inference, and then resume the algorithm. This method overcomes GPU memory limitations, but it is difficult to integrate with a more sophisticated renderer such as a path tracer. For this reason, we also introduce a third method, *In-Shader Inference*, by developing a fully customized network inference routine. We follow the same optimization principle used by the `tiny-cuda-nn` framework; however, our inference routine can be directly invoked inside a rendering shader, and thus can be applied much more easily to a variety of rendering algorithms.

We conclude this paper with an experimental evaluation of our techniques. We first evaluate our neural network training methods by comparing the quality of the reconstructed volume as well as rendered images with ground truth. We then evaluate the rendering performance of our implementation in terms of FPS and memory footprint. Next, we highlight the effectiveness of our approach when applying it to large-scale out-of-core volumes and time-varying volumes. Finally, we compare our implementation with the concurrent work done by Weiss et al. [50].

2 Related Work

The notable success of deep neural networks over the past many years has motivated a lot of recent scientific visualization work to adopt such methods. As we are focusing on creating a compact neural representation for interactive volume rendering, we first give a brief overview of related deep learning methods as applied to volume compression. Since our volume representation is a type of implicit neural representation, we also review existing research in this area. Finally, since our method relies heavily on the use of a multi-resolution hash grid encoding technique for fast feature learning, we conclude this section by providing background on this technique.

Deep Learning for Volume Compression High-quality volume visualization can be challenging, in part because handling large-scale, high-resolution volumetric data is often very difficult. Therefore, many deep learning techniques have been explored to compress volumetric data. Earlier work by Jain et al. [18] presents an encoder-decoder network to compress a high-resolution volume, and then decompress it before rendering via ray casting. Wurster et al. [51] later uses a hierarchy of generative adversarial networks (GANs) to complete the same task. Super-resolution neural networks can directly work with a low-resolution volume, and upscale it when necessary. This technique can be useful when it is too expensive to simply store the data in high resolution [10, 14, 53], or for all timesteps [13, 15], or both [15]. This technique can also be used to accelerate simulations by allowing them to work on lower resolution grids [52]. However, it requires a regular grid in space-time due to the use of convolutional and recurrent networks and requires a whole block to be decompressed before rendering. Lu et al. [27] explore the use of multilayer perceptrons (MLP). By directly using the trained MLP as the output, the network complexity can be reduced while still maintaining a high reconstruction quality. However, a time-consuming training process is needed for every new dataset. Concurrent work performed by Weiss et al. [50] improves this work by employing GPU-accelerated methods to render such networks. Our work also improves Lu et al.'s method, but we focus not only on network inference but also on accelerating the training process significantly. By embracing online training, we can enable interactive and responsive visualization of detailed volumetric data while improving data scalability via compression.

Representing Volumes using Neural Networks Implicit neural representations are neural networks that directly map a spatio-temporal position to the volumetric data value. Unlike encoder-decoder and super-resolution methods, implicit neural representation can be directly sampled without the need to explicitly decode the volume. Thus, they are more flexible and can produce an even smaller memory footprint. Implicit neural representations were first proposed to represent 3D opaque meshes, either as occupancy grids [29, 30] or signed distance fields [3, 6, 25, 46]. These works train networks directly using spatial coordinates and data values. Our work follows this principle. Notably, implicit neural representations can also be trained using rendered or real-world images through a differentiable rendering process. Such methods were first introduced for 3D reconstruction [41], and popularized by NeRF [31]. Further development of this method includes reduction of aliasing artifacts [2], speed improvements by caching [8], or incorporation of lighting effects [43].

Network Input Encoding Although our method does not use the same method as the NeRF work performed by Mildenhall et al. [31], we do adopt their idea to construct an encoding layer in the neural network that first maps inputs to a higher-dimensional space, as it allows MLPs to better capture high-frequency, local details. One-hot encoding [16] and the kernel trick [48] are early examples of techniques which make complex data arrangements linearly separable. In deep learning, input encodings have proven useful for recurrent networks [9] and transformers [49]. In particular, they encode scalar positions as a sequence of sine and cosine functions. Similar encoding methods are used by the NeRF work and many others [2, 34, 35, 47], and are often referred to as *frequency encodings*. To make the encoding more task specific, trainable parameters are added to the encoding layer by creating additional data structures such as dense grids [29], sparse grids [3, 4, 12, 19, 26, 37], octrees [46], or multi-resolution hash tables [33]. These input encoding methods are referred to as *parametric encodings*. By putting additional parameters in the encoding data structure, the size of the neural network can be reduced. Therefore, networks with such encoding methods can typically converge much faster while maintaining approximately the same accuracy. In this work, we adopt the multi-resolution hash grid method proposed by Müller et al. [33] because of its excellent performance in training MLPs.

3 Representing Volumes as Functions

Volumetric data in scientific visualization essentially can be written as a function Φ that maps a spatial location $\mathbf{x} = (x, y, z)$ to a value vector $\mathbf{v}$ which represents the data value at that spatial location:

$$\Phi : \mathbb{R}^3 \to \mathbb{R}^D, \ \mathbf{x} \mapsto \Phi(\mathbf{x}) = \mathbf{v}. \tag{1}$$

Such a volumetric function is typically generated by simulations or measurements and then discretized, sampled, and stored. In this work, we focus on scalar field volumetric functions (i.e., $D = 1$) and employ a neural network as the approximator to model the volumetric function Φ, by optimizing it directly using samples' spatial locations $\mathbf{x}$ and data values $\mathbf{v}$. Since the network is defined over the continuous domain $\mathbb{R}^3$, it can directly calculate data values at an arbitrary spatial resolution, and explicit data interpolation can be avoided. Moreover, since the network processes each input position independently of the others, the volumetric function can also be sampled at arbitrary positions on-demand. Additionally, because the neural network approximates the volumetric data analytically, the number of neurons needed to faithfully represent the volumetric data will not necessarily increase as the data resolution increases, promising greater scalability for this method. Finally, although only univariate scalar field volumes are studied in this work, we believe that there is no reason why the same method can not be trivially extended to multivariate volumes.

In practice, we represent Φ as a multi-layer perceptron (MLP) following the approach used by previous works [27, 50], and thus the spatial resolution of our volumetric neural representation is limited by the capacity of the MLP. However, unlike NeRF [31], which uses the network to directly predict an RGBα for rendering, we focus solely on the mapping from spatial locations to data values before the transfer function classification for three reasons. Firstly, a neural representation that directly predicts volumetric data values can also be used in non-rendering related workloads which access the same data. Secondly, because such a neural representation is functionally exactly equivalent to a conventional discrete volumetric data representation, it can work directly with existing rendering algorithms such as ray marching and volumetric path tracing, with almost no modifications to the rendering algorithm. Thirdly, performing transfer function classification independently of the training process will allow any transfer function to be supported and interactively edited during visualization. Although with our online-training approach, a neural representation incorporating transfer function information can quickly adapt to new transfer functions during an interactive visualization session, we did not find this approach beneficial in our early experiments. Because a neural representation trained to predict RGBαs can sometimes fail to converge when a rather sharp transfer function is used, leading to very poor user experiences. Similar phenomena are also reported by Weiss et al. [50].

Network Implementation We implemented our MLP based volumetric neural representation directly in CUDA and C++ using the `tiny-cuda-nn` machine learning framework [32] because it provides a much faster implementation for training and inference. This is achieved by fitting MLP's weights into the shared memory and registers and efficiently utilizing GPU tensor cores. Additionally, as indicated by Mildenhall et al. [31] and Tancik et al. [47], when the neural representation is optimized directly with spatial coordinates $\mathbf{x}$, the network cannot efficiently capture high-frequency features in the data. To capture these high-frequency details, an input encoder that can convert spatial coordinates into a high-dimensional feature vector $\mathbf{f}$ is needed. In our implementation, we employ the multi-resolution hash grid encoding method recently proposed by Müller et al. [33] and found that this method can also significantly accelerate the training speed even for volumetric neural representations. This feature is key to how our neural representation can support high-performance, interactive volume visualization. Figure 2 compares the effectiveness of different encoding methods.

Network Parameters We use a small MLP of size 64×4 with a multi-resolution hash grid encoding layer to represent volumetric data. Our encoding layer has 16 levels, 4 features per level, a hash table

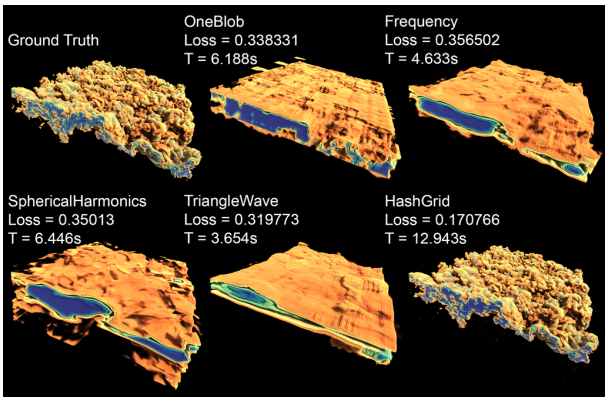

Fig. 2. Comparing the effectiveness of different input encoding methods with the `1atmHR` dataset. We train each neural network for 500, using L1 loss and the Adam optimizer with learning rate = 0.01. Documentation for each encoding method can be found in [32].

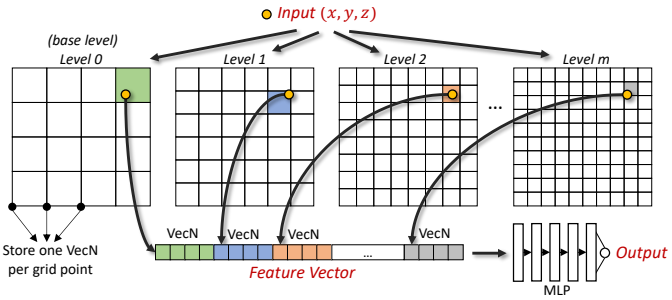

Fig. 3. The architecture of our neural network with the multi-resolution hash grid encoding method of Müller et al. [33].

of 2^{19} entries per level, and a base resolution of 16. We use `ReLU` as the activation function for the MLP's hidden layers and no activation function is applied for the output layer. We found that the removal of the output activation function is crucial for some datasets to converge. To train the network, we use the $L1$ loss function and a regular Adam optimizer [20] with learning rate decay. The learning rate starts at 0.01 and decays by a factor of 0.8 for every 1000 steps. The training batch size is 65536 in all the cases.

4 Online Training

As mentioned before, we choose online training rather than pre-training for better data adaptivity. The general idea is, during each visualization, we start with a randomly initialized neural network. At each frame, a number of (the "batch size") random coordinates are generated within a normalized domain $[0, 1)^3$ as the input, then the training system will use these coordinates to query or reconstruct the corresponding volumetric function values (i.e., the ground truth data) using the target dataset. In this paper, we refer to this step the "sampling" step. Because each volumetric function value is generated independently of others, the sampling step is embarrassingly parallelizable. Then, once the sampling is done, the input and the ground truth data will be loaded into the GPU memory (if they are not already there), and then used to optimize our volumetric neural representation using the `tiny-cuda-nn`'s C++ API. This step is referred to as the "training" step. Finally, after the training step, the current neural representation will be used by the renderer to generate an image frame for displaying. Notably, unlike normal images, volumetric data in scientific visualization can sometimes have a highly dynamic range. Therefore, in the sampling process, we also normalize ground truth values to $[0, 1]$, and then adjust our rendering configuration accordingly. In fact, in our experiment, we found that if unnormalized ground truth values were accidently used for training, the training process can quickly fail, producing a zero representation as a result. In this work, we develop three training strategies based on the target dataset's location.

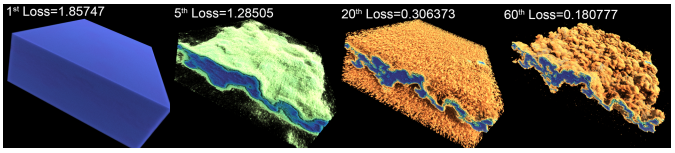

Fig. 4. Screenshots of a online-trained volumetric neural representation at different training steps. The data being trained is `1atmHR`. This sequence is captured from the same experiment done for Figure 2, therefore their loss values are comparable.

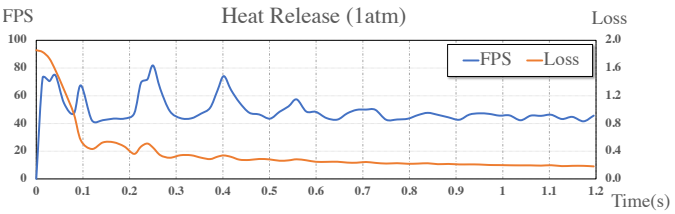

Fig. 5. Real-time loss curve of the same experiment described in Figure 4. Only displaying the loss until the 60^{th} step. It took about 1.2s to complete 60 steps.

4.1 GPU-Based Training

When the target dataset is small enough to be entirely loaded into the GPU memory, we can conduct the sampling and training entirely on GPU. Because the input and the ground truth data does not need to be constantly transferred from CPU to GPU, the I/O overhead is minimized. Because now the entire GPU can be used to power the parallel sampling process, the sampling step can also finish faster. As a result, the overall framerate can be improved.

However, since the target volumetric data already resides in the GPU memory, there is no need to use the neural representation as the media for rendering. It would be easier, faster, and more accurate to directly access and render the target volume using traditional rendering techniques. Therefore, in practice, this training strategy will only be meaningful if the goal of the output volumetric neural representation can be saved to disk as a data archive or snapshot. As a result, our technique is suitable for applications such as *in situ* or *in transit* visualizations, and can be served as an alternative approach for large data data compression and I/O. The overall size of a trained volumetric neural representation is negligible (∼55MB) compared to the target dataset.

4.2 CPU-Based Training

When the target dataset is larger than the GPU memory, executing the sampling and training process entirely on the GPU would no longer be possible. Assuming that the target dataset can still be loaded into the system memory, then we can perform the volume traversal and sampling steps on the CPU side. In this case, multi-threading and SIMD vectorization can be used to accelerate the sampling process. In fact, the training batch size used in our implementation is 65536, which is generally smaller than the number of pixels in a normal image and can be significantly smaller than the amount of samples needed by a regular volumetric ray marcher or volumetric path tracer to complete a frame. Thus, our sampling step is much cheaper than a typical CPU-based rendering process. Therefore, we can combine this strategy with our CUDA machine learning framework to train a neural representation, and then render the representation using GPU. Such a hybrid training and rendering strategy can still outperform a well-optimized CPU-based ray tracer.

Moreover, in our implementation, the sampling step for this strategy is implemented using the OpenVKL volume computation kernel library. This is because OpenVKL already provides a very complete set of volume sampling features, and can support a variety of different volume structures. This also allows us to apply our volumetric neural representations beyond regular-grid volumes. In Table 1, we compared the performance of this training strategy (coupled with the rendering strategy described in Section 5.3) with OpenVKL's example volume renderer. Our method is not only faster for the path tracing algorithm, but can also outperform OpenVKL's well optimized interval traversal implementation (which utilizes empty space skipping and adaptive sampling). We also benchmark our volumetric neural representation using an OpenVDB volume (i.e., the `Cloud` dataset) as shown in Table 1.

Table 1. Comparing the path tracing and ray marching performance in FPS (and speedups) between the OpenVKL example renderer (VKL) and our renderer (VNR). The VNR method took training samples on the CPU using OpenVKL and was rendered on the GPU using the volume decoding method described in Section 5.3. VKL renderings were done completely on the CPU. A Linux PC with a NVIDIA RTX 3090 and an Intel 9900k was used. Rendering settings were matched as closely as possible with rendered images provided in the supplementary materials. All the datasets were trained at full resolutions. However, `PigHeart`, `Instability` and `Cloud` were decoded into smaller 1024^3 grids for VNR rendering. Whereas for VKL, these datasets were rendered at full resolutions.

	Path Tracing		Ray Marching	
Dataset	VNR	VKL	VNR	VKL
1atmTemp	27.1 (×20)	1.36	36.3 (×14)	2.52
Instability	10.0 (×42)	0.24	14.9 (×2.2)	6.92
PigHeart	12.2 (×39)	0.31	15.0 (×1.9)	8.01
Cloud	20.9 (×20)	1.05	27.7 (×12)	2.36

4.3 Out-of-Core Training

The out-of-core training method is designed to handle data that are too big to be loaded into the system's main memory. Generally, there are three methods to render such datasets. The first method is to resample the data into a smaller volume with a lower resolution for rendering. However, small scale high-frequency details that appear in the original data would be lost with this method. The second method is distributed rendering, which requires a powerful computer cluster. The third method is out-of-core rendering, which fetches data from the storage device to the memory on-demand for rendering. Although this method can produce high-quality and high-performance rendering through progressive or LoD-based techniques, it comes with a price that a very sophisticated data cache needs to be maintained. With our volumetric neural representation and online-training technique, we introduce a new method. Unlike the resampling method, our method can potentially capture very high-frequency details and still produce a very stable memory footprint similar to the out-of-core rendering method. However, by using a neural network to "memorize" seen data, we are essentially creating a self-maintained data cache.

Our out-of-core online training method maintains a buffer of R randomly selected blocks of size B (referred to as the random block buffer) in the system's main memory as illustrated by Figure 6. Each block in the buffer is able to hold an equal amount of continuous data from the volumetric data file. At the start of the visualization, the whole random buffer is populated by randomly read R blocks from disk. Then at each frame, for each training sample, we first randomly select a block within the random buffer, and then randomly select a data value within the selected block as the sample's ground truth value. Because each block also remembers the file offset of its first data value, we can also calculate the corresponding spatial coordinates for selected data values. These spatial locations are then normalized and used as the neural network inputs. Once the sampling process is done, a batch of asynchronous I/O requests is created to randomly stream M new blocks, with each request handling exactly one block. These I/O requests can be executed concurrently with the training and the rendering. They are synchronized only at the start of the next frame's sampling step.

In our implementation, we use NVMe SSD as the data storage device because it offers superior random read bandwidth. We use 32KB-worth-of-data-sized block for streaming (B=32KB) to compensate the small overhead introduced by each I/O request. Additionally, we use $M = 512$ to maintain a relatively good interactivity, and a fairly large $R = 131072$ to give a good initial representation of the full resolution volumetric data in disk.

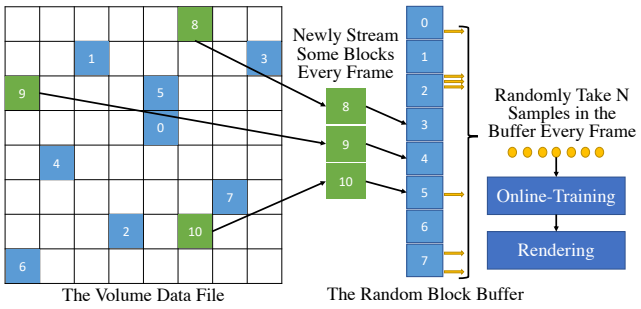

Fig. 6. Our out-of-core-streaming-based online training method is achieved by maintaining a random block buffer in the system memory. This buffer is sampled and asynchronously updated every frame. Sampled results are transferred to GPU for training.

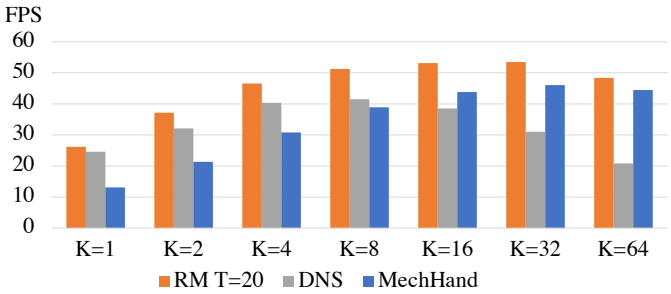

Fig. 7. In our sample-streaming renderer, for each iteration, K samples are generated by each ray. The choice of K can affect the algorithm's performance. We tested our algorithm's performance using three different with different K values. We found that the algorithm performs the best when K is between 8 and 32. In our implementation, we use $K = 16$.

5 Rendering

To visualize the trained neural representation, we provide three distinct rendering techniques. Two of the techniques use the trained network directly to infer sample values for rendering. The remaining technique progressively passes the network through a decoding step in every rendering pass to reconstruct the volumetric field. The reconstructed volume is then rendered with conventional rendering techniques. Each technique takes a unique approach to visualization and overcomes different hurdles. Therefore we do not recommend a single one as the preferred method.

5.1 Sample Streaming

The machine learning framework we use was made to support batch training and batch inference, meaning that a fairly large amount of inputs need to be pre-generated and then the neural network infers all the inputs at the same time. Such an execution model works for training, but it is not optimal for a general rendering task. For example, in a typical front-to-back ray marcher, a ray iteratively samples the volume and calculates a composited pixel color (via alpha blending) along its path within the volume. The ray terminates when the composited color is nearly opaque. Although it is possible to pre-compute all the sample coordinates for each ray, it is, however, impossible to predict when a ray will terminate. Blindly infering all the possible samples at each frame would be very inefficient. Similarly, in a volumetric path tracer, a ray is scattered within the volume, with each scattering position stochastically depending on the sample value of the previous scattering position. In such a scenario, it is difficult to even predict sample coordinates for each ray. Therefore, we need to fundamentally change the rendering algorithm to take advantage of the machine learning framework.

We started by focusing on ray marching and split the algorithm into three smaller kernels: the ray-generation kernel, the coordinate-computation kernel, and the shading kernel The ray-generation kernel initializes all the primary rays and conducts an intersection test with the volume. If no intersection is found, the ray is marked as invalid, and a stream compaction is performed to remove invalid rays and return the total number of valid rays. Within the loop, for each ray, the next K samples' coordinates are written into a temporary buffer. This buffer is used for neural network inference. After the inference, sample values calculated by the neural network are retrieved by each ray from a different temporary buffer for shading. Once the shading is done, another stream compaction is performed to remove rays that are terminated (because they have exited the volume or have become fully opaque). Finally, the loop terminates if all the rays are invalid.

When the number of samples generated by each iteration K is greater than 1, the number of iterations needed to complete a frame can be reduced. Because there are exactly 4 CUDA kernel launches within one iteration, the CUDA kernel launch overhead is also reduced. Additionally, because exited rays are removed at the end of each iteration, the number of valid rays within each iteration will decrease as the loop iterates. Because one CUDA thread is launched per ray, the CUDA kernel launch size within each iteration will also decrease and eventually become very small. By increasing K, these small kernel launches can be batched up to further reduce the launch overhead. However, if K is too large, the number of samples being computed per iteration can be more than necessary because many rays might have exited earlier. These unused samples can only be discarded and the amount of time spent to compute these samples are wasted. Therefore, in order to achieve the optimal performance, K needs to be tuned. Figure 7 illustrates the relationship between K and the rendering framerate for different datasets. We can see that the algorithm performs the best when K is between 8 and 32. In this paper we use $K = 16$. The impact of K can become obvious when the rendering framerate is above 20fps.

5.2 In-Shader Inference

Although the sample-streaming method can directly render at an interactive framerate, implementing it can be very tricky. In fact, we only managed to implement it inside a volume ray marcher. For more sophisticated rendering algorithms such as ray marching with global shadows or volumetric path tracing with next event estimation, we could not convert them to the iterative form described previously. Therefore, a more general solution that allows the volumetric neural representation to be directly inferred inside a shader program is needed. We managed to do so by developing a fully customized neural network inference implementation in our machine learning framework.

First, we eliminate all the accesses to the GPU global memory for reading inputs and writing outputs, because these values are directly generated and consumed by each thread locally. Additionally, in our implementation, the input encoding step and the MLP inference are fully fused as a single GPU kernel, whereas in `tiny-cuda-nn`, they are implemented and invoked separately.

Second, as illustrated by Figure 8A, our neural network framework utilizes GPU tensor cores by breaking a MLP layer into 16-neuron-sized groups, then allowing the computation within each group to be completed by a 32-thread warp for 16 consecutive inputs (i.e., 16 encoded feature vectors). Thus, each warp only handles multiplications between two 16×16 matrices, which can be accelerated using GPU tensor cores. When the width of the network W is greater than 16, $\frac{W}{16}$ warps are combined to form a GPU block, and executed in parallel. Additionally, in order to compensate the GPU scheduling overhead, each warp also iterates for N times to infer more inputs. Therefore, $16\times N$ inputs are processed by each block. When it comes to in-shader inference, the number of inputs processed by each block should be exactly the same as the number of threads within a block because in a typical GPU-based ray tracer, each thread handles exactly one ray and only produces one input coordinate at a time. Therefore, we let $N = \frac{W}{8}$. Additionally, our implementation synchronizes the entire block at the beginning and the end of an inference, to re-arrange threads and allow each thread's input and output to be accessible by others through the GPU shared memory.

Third, because a special thread-block arrangement and the GPU shared memory are needed, our volumetric neural representation should also be responsible for all the kernel invocations. To do so, we provide a templated CUDA invocation wrapper function that takes the ray-generation shader of a rendering algorithm as a callback argument, and launches the shader for the rendering algorithm. In this way, a regular

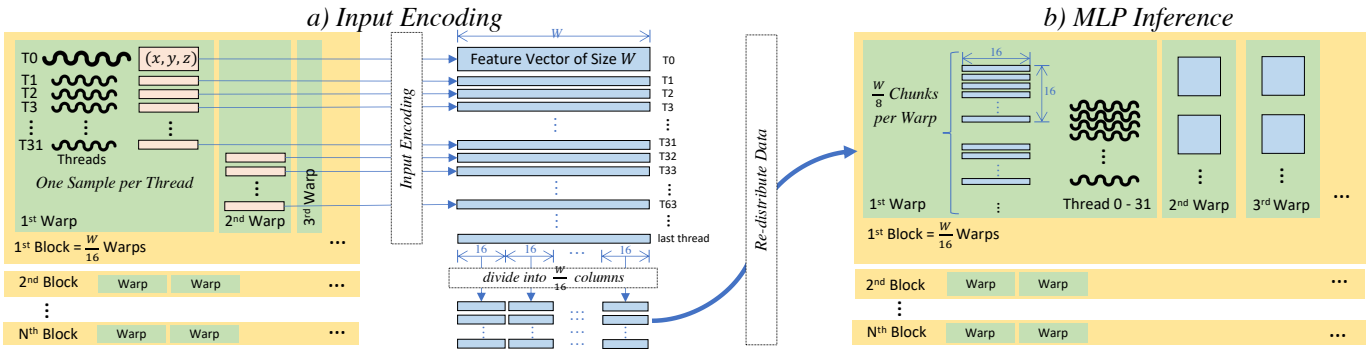

Fig. 8. a) A typical GPU ray tracer produces one sample coordinate (x, y, z) per thread. The input encoding is also computed by each thread independently, each producing a feature vector of size W. These feature vectors are then divided into $\frac{W}{16}$ 16-element-sized smaller vectors, written into the shared memory. b) The MLP inference is computed by re-distributing feature vectors between threads. Particularly, threads in a warp collaboratively compute the matrix multiplication of a 16×16 data chunk each time using a tensor core, and iteratively computes $\frac{W}{8}$ chunks. There are $\frac{W}{16}$ warps in a block, computing $2W$ feature vectors in total.

```
inline __device__ bool
block_any(bool b) { return __syncthreads_or(b); }

template<typename Volume>
__device__ float4
TraceRay(const Volume& vnr, ...) {
  bool intersected = intersectBox(...);
  if (block_any(intersected)) {
    /* compute coordinates */
    float value = vnr.sample(coordinate);
    /* shading */
  }
  return intersected ? float4{r, g, b, alpha} : float4{};
}

template<typename Volume>
__global__ void
RayGen(Volume vnr, int width, int height, ...) {
  int ix = ..., iy = ...;
  bool skipped = (ix >= width) || (iy >= height);
  /* ... */
  float4 color = TraceRay(vnr, ...);
  if (!skipped) { framebuffer[iy][ix] = color; }
}
```

Listing 1. Pseudo-code for a in-shader renderer.

GPU-based volume ray tracing algorithm can be easily converted to support our in-shader rendering method. Additionally, because the use of GPU tensor core requires co-operation from all threads in a warp, we need to ensure that when an in-shader network inference is involved, all the threads within a block should be active. To achieve this, we provide a `block_any` function (as shown in Listing 1) for warp control flows. This function evaluates true for all threads within a block if any of the thread-local-predicates `v` evaluates true. In Listing 1 we highlight all the code changes that need to be done within a rendering algorithm in order to support in-shader rendering of our volumetric neural representation. In our implementation, we port this rendering method to a ray marching algorithm with gradient shading, and a volumetric path tracing algorithm with next event estimation.

5.3 Volume Decoding

Although calculating all the volume samples by directly inferencing the neural representation enables high rendering quality, the amount of computations needed can be massive. This usually leads to a lower framerate. For scientific visualization, it is often acceptable to render a low-quality representation of the volume when the user is tuning the camera angle or editing the transfer function. When the user is no longer actively interacting with the system, high-quality rendering can be restored. Because rendering the low-quality representation is usually much faster, an interactive experience can still be maintained. Thus, we took inspiration from this observation, and developed an alternative rendering mode for maintaining interactivity and improving user experience. This method enables interactive volume rendering by decoding the neural representation at a lower resolution every frame, and then rendering the decoded volume conventionally and interactively. The decoding process effectively reconstructs the volumetric field using a 3D regular grid. However, since our neural representation is being constantly optimized, a decoding process is generally needed for every frame. Unlike images, the number of samples needed to recreate a volume can be a lot. Thus, any performance advantages brought by using a conventional rendering method can quickly vanish in this step. Our solution is to execute the decoding process progressively. More specifically, we still allow the full neural representation to be updated every frame, but we calculate volume decoding using multiple frames. In this way, only a fixed amount of volume slices to be decoded every frame (which is 16 in our implementation), thus lots of computation can be saved.

One immediate impact of this progressive decoding approach is that the decoded volume being rendered is never fully up-to-date with the neural representation of the volume. This is particularly problematic in the early stage of training as the neural representation can change drastically between frames. As a result, "stripe-like" artifacts are often observed. However, we found that because our neural network converges extremely fast (usually a fairly stable representation can be obtained within 5 seconds), such effect can quickly become less noticeable. Additionally, we provide special GUI features such as "pause training", which stops the volumetric neural representation from being updated, "decode fully", which decodes the entire volume once, and "fast-forward training", which pauses rendering and continuously trains the neural network for a fixed number of steps (1000 in our current implementation). By training a user to smartly use these features, the overall user experiences can be generally improved with this rendering method. Moreover, this rendering method is never meant to be used alone. In practice, we expect that a user will combine the volume decoding method with the sample streaming method or the in-shader inference method, and achieve a good balance between interactivity and rendering quality.

6 Evaluation

We evaluate our implementation of neural representation with respect to the training performance, memory footprint, and adaptivity in comparison with the state-of-the-art techniques. All the experiments are conducted on a Linux PC using a single NVIDIA RTX 3090 GPU, an Intel 9900k CPU, and 64GB RAM.

6.1 Datasets

We evaluate our volumetric neural representation with several datasets with a varity of sources, contents, and sizes.

First, to study the model accuracy, we test our implementation with seven smaller datasets that can be fitted into the GPU memory of our RTX 3090. The temperature (`Temp`) and heat release rate (`HR`) fields for turbulent lean ammonia/hydrogen/nitrogen-air flames at pressures of 1

Table 2. Benchmark for training quality, rendering quality, and memory footprint. Green datasets are trained with the GPU-based method, blues are with the CPU-based method, red is with the out-of-core method. The neural network described in Section 3 was used. Our technique does not require pre-training a neural representation. However, for the purpose of this benchmark, we had all datasets **pre-trained** for exactly 10k steps. The training times are also reported. Volume accuracy was measured at full resolutions using the `scikit-learn` library. Because the Python library crashed for some large-scale data, some PSNRs and MSSIMs are unavailable. These data also could not be rendered at full-resolutions on GPU, thus their rendering quality comparisons are unavailable as well. However, decoded volumes were still computed for them (maximum resolution = 1024 on each dimension). Memory footprints reported only measure the memory consumption when VNRs were used. They do not cover runs when ground truth images were rendered.

		Volume Accuracy				Rendering Quality						Memory Footprint			
						RM+S		PT+NEE		RM+G					
Dataset	Dimension	Training Time	Loss	PSNR	MSSIM	PSNR	MSSIM	PSNR	MSSIM	PSNR	MSSIM	VRAM	Trainer	Decoded Volume	VNR
1atmTemp	1152×320×853	43.3s	0.013	29.8	0.996	23.7	0.63	24.7	0.75	22.5	0.52	4.1 GB	20.0%	1.2 GB	54.3 MB
10atmTemp	1152×426×853	43.7s	0.017	29.2	0.991	24.1	0.65	23.5	0.72	22.4	0.53	4.9 GB	17.8%	1.6 GB	54.3 MB
1atmHR	1152×320×853	44.9s	0.017	27.2	0.861	23.8	0.73	22.4	0.71	22.6	0.55	4.1 GB	20.1%	1.2 GB	54.3 MB
10atmHR	1152×426×853	44.9s	0.019	24.8	0.972	23.0	0.61	20.1	0.66	19.9	0.47	4.9 GB	17.8%	1.6 GB	54.3 MB
Chameleon	1024×1024×1080	43.0s	0.003	44.3	0.995	41.1	0.97	39.6	0.97	35.5	0.93	9.2 GB	12.0%	4.2 GB	54.3 MB
MechHand	640×220×229	35.9s	0.004	28.8	0.950	37.8	0.97	41.2	0.99	34.3	0.94	2.3 GB	30.6%	0.1 GB	54.3 MB
SuperNova	432×432×432	34.9s	0.001	41.7	0.991	36.7	0.93	32.5	0.91	34.6	0.89	2.6 GB	27.8%	0.3 GB	54.3 MB
Instability	2048×2048×1920	150s	0.015	25.8	-	-	-	-	-	-	-	8.9 GB	12.4%	4.0 GB	54.3 MB
PigHeart	2048×2048×2612	152s	0.001	34.0	-	-	-	-	-	-	-	8.9 GB	12.4%	4.0 GB	54.3 MB
Cloud	OpenVDB	129s	0.005	-	-	-	-	-	-	-	-	5.0 GB	16.5%	1.8 GB	54.3 MB
DNS (dp)	10240×7680×1536	207s	0.016	-	-	-	-	-	-	-	-	8.9 GB	12.1%	4.0 GB	54.3 MB

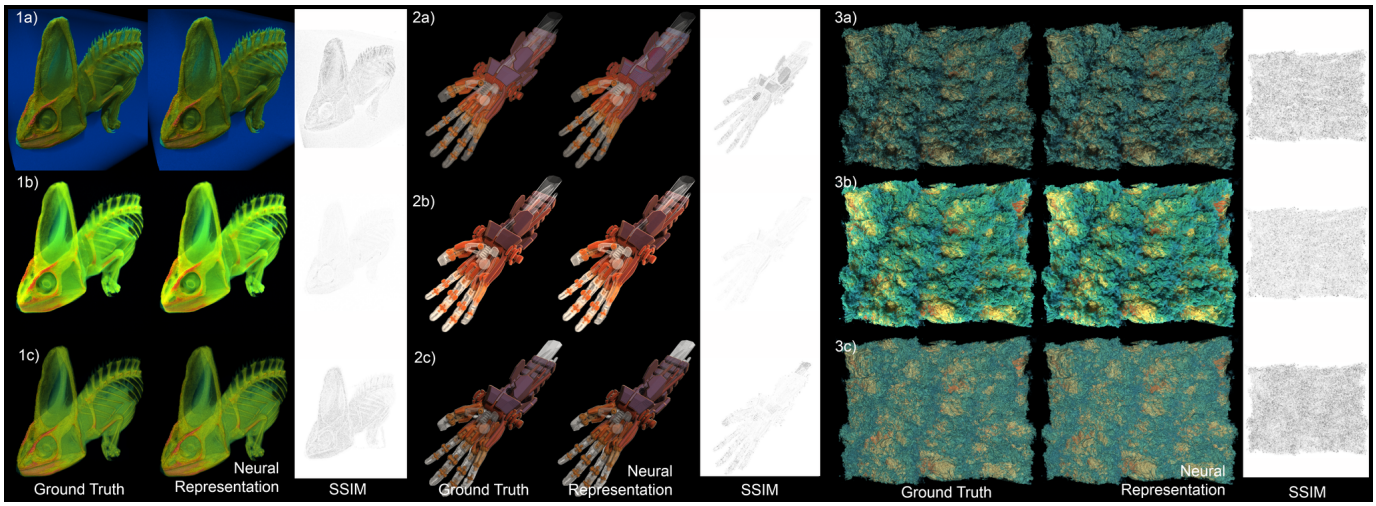

Fig. 9. Rendering comparison between the ground truth and our neural representation. We also display the image SSIM. Renderings are done by the `Chameleon`(1), `MechHand`(2), and `1atmTemp`(3) experiments described in Table 2. Rendering algorithms used are: a) volume-decoding-based ray marching with global shadow, referred to as RM+S, b) volume-decoding-based path tracing with next event estimation, referred to as PT+NEE, c) volume-decoding-based ray marching with gradient shading, referred to as RM+G.

and 10 atm in a temporally-evolving shear layer configuration [39]; A CT scan of a chameleon [28] (`Chameleon`); A mechanical hand converted from surface meshes (`MechHand`); The `SuperNova` is provided by astrophysicists at the Oak Ridge National Laboratory. They are trained using GPU-based method and colored with green in Table 2.

Then, to validate the accuracy and the efficiency our CPU-based online-training method, we employ two datasets that are larger than the GPU memory but can still be loaded into the system memory. The `Instability` is the entropy field (timestep 160) of Richtmyer-Meshkov instability simulation [5]; The `PigHeart` is a CT imaging data on an excised, postmortem porcine heart, with the experiment performed by the University of Utah [21]. These datasets are converted into single-precision floating-point regular-grid volumes for our experiments. We color them with blue in Table 2.

Additionally, to varify the effectiveness of our technique with regards to non-regular-grid volume, we also use the high-resolution `Cloud` sparse volume [44], which is in OpenVDB format. In Table 2, we also color this dataset with blue because non-regular-grid volume can only be learned using our CPU-based training method.

Finally, to evaluate our out-of-core training strategy, we use the 950GB `DNS` channel flow simulation dataset [22]. We tested this data in double-precision to stress-test the scalability.

6.2 Memory Footprint

In Table 2, we also highlight the GPU memory footprint of our implementation. Because we used the same neural network for all the datasets, each neural representation produced exactly the same memory footprint. To train the neural network, some additional GPU memory spaces are allocated. However, they are proportional to the training batch size (65536 in our implementation). The remaining memory space is used by the renderer, with a significant amount of them used to store the decoded volume for the volume decoding method.

6.3 Training

Model Accuracy. We measured how well our volumetric neural representation can learn features from a given volumetric data by comparing a reconstructed volume as well as rendered images with the ground truth. In particular, for the purpose of this experiment, we pre-trained each volumetric neural representation for 10k steps (64k samples per training step) and recorded the final $L1$ loss value in Table 2. We also reported all the training times in the same table as "training times". Note that in practice, we usually do not need 10k training steps to obtain a renderable neural representation. Then, we progressively decoded the volume at its original resolution and calculated the peak signal-to-noise ratio (PSNR) as well as the mean structural similarity index (MSSIM) between the reconstructed volume and the original data. These cal-

Table 3. Framerates in FPS of different rendering methods and datasets. The percentages of frametime used for training are also reported.

	Method	1atmTemp	1atmHR	10atmTemp	10atmHR	Chameleon	MechHand	SuperNova	Instability	PigHeart	Cloud	DNS (dp)
Decode	RM	26.4 - 37%	41.5 - 57%	35.0 - 62%	25.4 - 49%	9.28 - 46%	91.7 - 58%	55.8 - 45%	15.1 - 85%	14.3 - 79%	29.7 - 59%	16.0 - 94%
	RM+S	4.42 - 6.3%	7.80 - 12%	9.07 - 14%	4.05 - 6.3%	0.65 - 3.1%	15.6 - 11%	4.63 - 3.9%	1.98 - 10%	2.55 - 13%	2.44 - 4.9%	13.5 - 78%
	PT	30.3 - 43%	43.5 - 56%	36.2 - 66%	36.2 - 59%	8.90 - 43%	87.9 - 59%	53.9 - 43%	11.2 - 62%	13.2 - 69%	21.2 - 43%	15.6 - 93%
	PT+NEE	21.0 - 28%	32.9 - 45%	29.5 - 55%	30.1 - 52%	8.50 - 40%	74.6 - 49%	45.1 - 36%	10.0 - 52%	11.8 - 65%	19.1 - 41%	15.3 - 90%
In-Shader	RM	3.08 - .69%	2.92 - .63%	3.01 - .63%	2.28 - .48%	0.37 - .07%	1.91 - .31%	0.83 - .13%	1.11 - .76%	1.01 - .79%	0.66 - .37%	8.63 - 8.9%
	RM+G	0.75 - .17%	0.77 - .16%	0.78 - .16%	0.59 - .12%	0.09 - .02%	0.48 - .08%	0.21 - .04%	0.28 - .22%	0.26 - .18%	0.17 - .09%	2.37 - 2.4%
	PT	0.28 - .06%	0.26 - .05%	0.28 - .06%	0.22 - .05%	0.10 - .02%	0.46 - .08%	0.34 - .06%	0.12 - .09%	0.19 - .13%	0.18 - .09%	1.01 - 1.0%
	PT+NEE	0.18 - .04%	0.17 - .03%	0.18 - .04%	0.16 - .03%	0.07 - .02%	0.34 - .06%	0.23 - .04%	0.07 - .05%	0.11 - .09%	0.13 - .07%	0.65 - .65%
Sample Streaming		13.2 - 2.7%	13.0 - 2.7%	13.7 - 3.9%	8.62 - 1.8%	1.50 - .30%	8.01 - 1.4%	3.00 - .51%	4.10 - 2.9%	5.0 - 3.6%	2.38 - 1.4%	30.4 - 31%

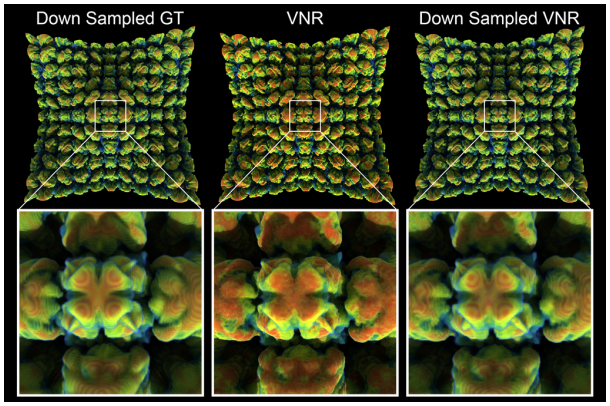

Fig. 10. Rendering quality comparison between a ground truth data and a trained volumetric neural representation (10k training steps). Left: a 8× downsampled `Instability` data rendered using a reference volumetric path tracer. Sampling artifacts can be clearly observed. Middle: direct rendering of our neural representation (VNR) using our in-shader path tracer. No obvious artifacts can be seen. Right: rendering the same neural representation using a volume-decoding-based path tracer. The decoded volume is of the same resolution as the down sampled GT. Again we can observe sampling artifacts.

culated results are reported in the training section (Column 4-6) of Table 2. Next, we rendered the learned neural representation using three algorithms: ray marching with volume decoding, path tracing with volume decoding, and ray marching with sample streaming. We calculated the image-space PSNR and SSIM by comparing rendered images with their corresponding ground truths. Because the ground truth images could only be rendered with volume fully loaded into the GPU, we only calculated PSNR and SSIM for smaller datasets (green in Table 2). From the results, we found that neural representations that are trained using datasets with clean structures (the `Chameleon` and the `MechHand`) generally performed very well (i.e., loss < 0.005, volume SSIM > 0.9, and image SSIM > 0.9). When the data is small enough ($< 500^3$), our neural representation can also capture almost all the features.

Training Speed. We measure the performance of our training strategies by calculating the percentage of frame time spent for training in each frame. The average percentages are shown in Table 3 alongside with the average rendering framerates for each dataset and each rendering method. Clearly, for all datasets, the training time is generally more significant when volume-decoding-based rendering methods are used. This is because volume-decoding-based rendering methods are generally faster while the training time is generally constant. For example, for a given dataset (say the `1atmTemp` dataset), the per-frame training times for all the decoding based methods are nearly identical ($\sim$0.014s for `1atmTemp`). In addition, the volume decoding process can take a noticeable amount of time to complete. For example, if we look at a dataset of a different resolution (e.g., the smaller `MechHand` dataset), the training time would be different ($\sim$0.007s in this case). However, if similar-sized datasets are compared, similar training costs can be found (e.g.,$\sim$0.014s for `1atmHR`). Moreover, for other rendering methods, the volume decoding process is skipped, therefore a comparable training time can be found ($\sim$0.002s) for all the rendering methods and datasets trained with the GPU-based method. When it comes to the CPU-based training method, the training costs are generally higher mostly because of the data transfer overheads ($\sim$0.008s for CPU-based training, $\sim$0.005s for the OpenVDB dataset trained with OpenVKL assistance, and $\sim$0.01s for the out-of-core method). However, despite being a bit slower, they are not the bottleneck for achieving interactive rendering in any cases.

Training Efficiency. Now we know our online-training strategies are fast; then the next question to ask is: Are they equally efficient? We define the term "efficiency" as the ability to optimize the neural network for every training step. To answer this question, we train the same dataset for a fixed amount of steps (10k steps) using different training strategies. In Figure 11, we show the results of this experiment in terms of loss curves. As we can see, the CPU-based method can produce a virtually identical loss curve compared with the GPU-based method, even though the CPU-based method requires 2× more time to complete the same amount of training steps. This is understandable because their training inputs are sampled using exactly the same approach, despite being implemented differently. The out-of-core method, however, is generally less efficient. This is also anticipated because the out-of-core method computes training inputs using a random buffer rather than the whole dataset. Because the random buffer can never be a fully accurate representation of the original dataset, the gradients calculated in the subsequent training process will be less accurate, leading to a less efficient training process. However, such a difference is only noticeable after a significant amount of training steps and can vary between datasets. Thus, we believe that our out-of-core training method is considered efficient for optimizing neural representations, especially in the early stage of the training process.

6.4 Rendering

Rendering Quality. Next, we analyze the quality of images produced by our rendering algorithms. In particular, we fix the rendering implementation and compare the differences between images rendered directly using the full-resolution data and images rendered using a volumetric neural representation trained for 10k steps (shown in Table 2). We also display some of the rendered images in Figure 9 and provide additional ones in the supplementary materials. We found that our rendering methods combined with our volumetric neural representation can produce faithful visualizations (with PSNR generally $>=$ 20dB for all rendering methods). For datasets that are more complicated for our neural representations to learn, we would have lower structural similarity indices, but these differences are still hard to notice visually. When the original data is too large for the GPU memory, our in-shader rendering method becomes useful because it allows the rendering algorithm to access higher quality details compared with a traditionally downsampled volume. In Figure 10, we compare images produced by a regular volumetric path tracer rendering a downsampled ground truth volume (downsampled GT), our in-shader renderer di-

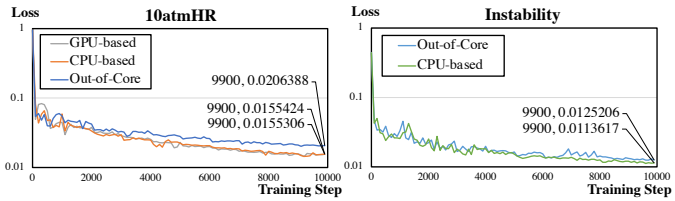

Fig. 11. The loss curves of the `10atmHR` and the `Instability` datasets using different training methods. Each dataset was trained for 10k steps.

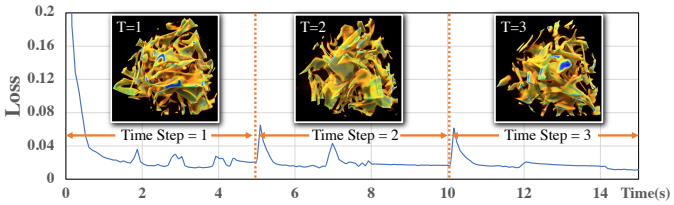

Fig. 12. We can use a single volumetric neural representation to interactively visualize a time-varying dataset. In the experiment, the timestep was automatically incremented every 5 seconds.

rectly rendering a trained volumetric neural representation (VNR), and our volume decoding renderer that decodes the neural representation at the same resolution as the downsampled GT (downsampled VNR). Visually the images produced by downsampled VNR matches the results from downsampled GT, indicating that our neural representation is a faithful representation of the ground truth. In contrast, very obvious artifacts that are caused by the downsampling process can be observed. Whereas our in-shader rendering method is free of these artifacts, it can produce significantly better renderings.

Rendering Performance. In Table 3, we also report the performance of all three rendering techniques we developed. For volume decoding and in-shader rendering, we have the results for regular ray marching (RM), ray marching with global shadows (trace one shadow ray per sample along a ray, RM+S), ray marching with gradient shading (RM+G), volumetric path tracing (PT), and path tracing with next event estimations (trace one shadow ray per scattering, PT+NEE). We did all the renderings with the sampling density = the unit length in the world coordinate. For green datasets, their world space bounding boxes were the same as their volume dimensions. For blue and red datasets, their world space bounding boxes were calculated by scaling down their volume dimensions by a factor of $s = \texttt{reduce\_max}(\texttt{VolumeDimension})/1024$. These settings can greatly affect framerates because they change the amount of samples taken per voxel, which was why the `DNS` dataset performed better than many smaller datasets. We made sure our rendering qualities were good visually and can exhibit very obvious occlusion effects if enabled. All the renderings are done with online-training enabled concurrently. Our results indicate that all of our volume decoding methods can achieve highly interactive volume rendering, with the RM+S being the most costly one. The sample-streaming method performs reasonably well, with interactive rendering can be achieved for almost all the scenes. In-shader rendering methods are currently significantly slower. This is because, compared with 3D textures, sampling the neural representation is much more costly due to the amount of computations involved in the network inference process. For example, the RM+G method computes an extra gradient at every sample location, thus this method takes exactly $4\times$ more samples than the RM method. Our results reflects this. Therefore, we believe, without further optimizations, our in-shader rendering method alone would be useful only for generating high-quality renderings once visual explorations are done, while for interactive visualization, this method should be coupled with our volume decoding or sample streaming based methods.

6.5 Adaptivity with Time-Varying Data

Our neural representation method can also be applied to time-varying data directly. In Figure 12 we demonstrate the real-time loss curve when interactively visualizing the multi-timestep vortices dataset provided by Deborah Silver at Rutgers University. Because we are continuously

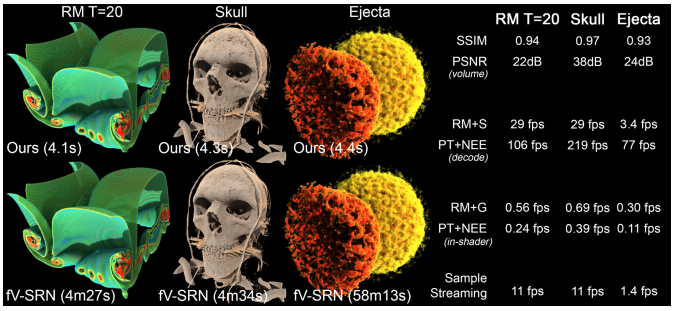

Fig. 13. Comparison between our technique with the fV-SRN [50]. We could not find the full resolution `Ejecta` dataset used by fV-SRN online, so it was trained using a low-resolution version available online. Full resolution images are provided in the supplementary materials.

training the same neural network (without resetting the weights and the step count) despite having a changeable target data, we cannot train the network with learning rate decay. Instead, we used a simple Adam optimizer with learning rate being fixed to 0.01. As shown in Figure 12, we changed the data timestep every 5 seconds. We can see that our neural network can quickly adapt to the new timestep, and automatically adjust what has been learned within around two seconds. This makes our technique very useful for interactive visualizations.

6.6 Comparison with fV-SRN

The concurrent work done by Weiss et al. [50] presents a volumetric neural representation, fV-SRN, very similar to ours, for direct volume rendering, and they create a fast tensor-core-accelerated CUDA implementation to compute MLP inference. However, their networks are trained using PyTorch and they use a frequency-based method for network input encoding. We consider this technique as the state-of-the-art technique for volumetric neural representations. To compare our technique with fV-SRN, we obtained the same data tested in their work, and report the volume reconstruction quality as well as the rendering performance in Figure 13. Because fV-SRN is a pre-trained network, we pre-trained all the data for 1k steps and rendered them without online-training. Additionally, in their work, the `Ejecta` dataset is a 1024^3 regular-grid; however, we could only locate a downsampled version (of size 256^3). Therefore, we rendered the `Ejecta` with roughly $4\times$ more samples per voxel to approximate the total number of rendering samples described in the fV-SRN paper. Since different renderers with different settings can produce totally different performance, we cannot direct compare our rendering time with theirs. However, we found that with our method, we can obtain volumetric neural representations with similar accuracy (SSIM > 0.93) while being nearly $60\times$ faster in terms of training time. We believe that our speedups come from the use of the native CUDA training framework (which reports a $10\times$ speedup compared with pytorch) as well as the multi-resolution hash grid encoding method (which reports another $10\times$ speedup compared with frequency-based encoding).

7 Conclusion

In this work, we present a volumetric neural representation that can be trained nearly instantaneously and rendered highly interactively. In particular, by simultaneously utilizing a fast CUDA machine learning network, modern GPU tensor cores, and our well-optimized online-training and rendering implementations, we achieve a nearly $60\times$ speedup in training time compared with the state-of-the-art. Our online-training strategies are highly scalable, which allows us to handle not only large scale volumes, but also irregular and time-varying volumes. Our rendering strategies are highly flexible, which enables high performance and high fidelity rendering for volumetric neural representations.

There is still room for future work in this direction. For example, even though our in-shader rendering method can provide the best rendering quality, it is significantly slower compared with our volume decoding based rendering method; therefore, both methods should be used simultaneously. However, if a more efficient in-shader rendering

technique can be found, then we can completely skip the volume decoding process, which not only saves rendering time, but also significantly reduces the memory footprint. Next, although we failed to find a general solution, we still believe that the transfer function information can potentially be used to improve the training process. This is because a user-defined transfer function should help the neural network to filter data noises. Such noises can disturb the neural network, making it fail to capture small-scale high-frequency details. Then, our online-training technique would be very suitable for such a transfer function assisted technique because of the fast adaptivity. Finally, while our volumetric neural representation focuses on direct volume rendering, developing a similar technique for other methods such as isosurface rendering should also be possible. Techniques such as the deep signed distance function have already provided some clues about constructing a surface representation using neural networks. Therefore, possible directions would be on how online-training can be used for such tasks and how such tasks can be used for general scientific visualizations.

Acknowledgments

This project is sponsored in part by the U.S. Department of Energy through grant DE-SC0019486. The ammonia/hydrogen/nitrogen-air flames dataset is kindly provided by Dr. Martin Rieth. The channel flow simulation dataset is kindly provided by Dr. Myoungkyu Lee.

References

[1] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola. Tthresh: Tensor compression for multidimensional visual data. *IEEE transactions on visualization and computer graphics*, 26(9):2891–2903, 2019.

[2] J. T. Barron, B. Mildenhall, M. Tancik, P. Hedman, R. Martin-Brualla, and P. P. Srinivasan. Mip-NeRF: A multiscale representation for anti-aliasing neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 5855–5864, 2021.

[3] R. Chabra, J. E. Lenssen, E. Ilg, T. Schmidt, J. Straub, S. Lovegrove, and R. Newcombe. Deep local shapes: Learning local sdf priors for detailed 3d reconstruction. In *European Conference on Computer Vision*, pp. 608–625. Springer, 2020.

[4] J. Chibane, T. Alldieck, and G. Pons-Moll. Implicit functions in feature space for 3d shape reconstruction and completion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6970–6981, 2020.

[5] R. H. Cohen, W. P. Dannevik, A. M. Dimits, D. E. Eliason, A. A. Mirin, Y. Zhou, D. H. Porter, and P. R. Woodward. Three-dimensional simulation of a richtmyer–meshkov instability with a two-scale initial perturbation. *Physics of Fluids*, 14(10):3692–3709, 2002.

[6] T. Davies, D. Nowrouzezahrai, and A. Jacobson. On the effectiveness of weight-encoded neural implicit 3d shapes. *arXiv preprint arXiv:2009.09808*, 2020.

[7] S. Di and F. Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pp. 730–739. IEEE, 2016.

[8] S. J. Garbin, M. Kowalski, M. Johnson, J. Shotton, and J. Valentin. FastNeRF: High-fidelity neural rendering at 200fps. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 14346–14355, 2021.

[9] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional sequence to sequence learning. In *International Conference on Machine Learning*, pp. 1243–1252. PMLR, 2017.

[10] L. Guo, S. Ye, J. Han, H. Zheng, H. Gao, D. Z. Chen, J.-X. Wang, and C. Wang. SSR-VFD: Spatial super-resolution for vector field data analysis and visualization. In *Proceedings of IEEE Pacific Visualization Symposium*, 2020.

[11] S. Guthe and M. Goesele. Variable length coding for gpu-based direct volume rendering. In *Proceedings of the Conference on Vision, Modeling and Visualization*, pp. 77–84, 2016.

[12] S. Hadadan, S. Chen, and M. Zwicker. Neural radiosity. *ACM Transactions on Graphics (TOG)*, 40(6):1–11, 2021.

[13] J. Han and C. Wang. TSR-TVD: Temporal super-resolution for time-varying data analysis and visualization. *IEEE transactions on visualization and computer graphics*, 26(1):205–215, 2019.

[14] J. Han and C. Wang. SSR-TVD: Spatial super-resolution for time-varying data analysis and visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[15] J. Han, H. Zheng, D. Z. Chen, and C. Wang. STNet: An end-to-end generative framework for synthesizing spatiotemporal super-resolution volumes. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):270–280, 2021.

[16] D. Harris and S. L. Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[17] Intel Corporation. Intel open volume kernel library, 2022 (Online). https://www.openvkl.org, version 0.9.0, accessed on 2022-03-09.

[18] S. Jain, W. Griffin, A. Godil, J. W. Bullard, J. Terrill, and A. Varshney. Compressed volume rendering using deep learning. In *Proceedings of the Large Scale Data Analysis and Visualization (LDAV) Symposium. Phoenix, AZ*, 2017.

[19] C. Jiang, A. Sud, A. Makadia, J. Huang, M. Nießner, T. Funkhouser, et al. Local implicit grid representations for 3d scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6001–6010, 2020.

[20] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] P. Klacansky. Open scientific visualization datasets. https://klacansky.com/open-scivis-datasets/.

[22] M. Lee and R. D. Moser. Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$. *Journal of Fluid Mechanics*, 774:395–415, July 2015. doi: 10.1017/jfm.2015.268

[23] M.-C. Lee, R. K. Chan, and D. A. Adjeroh. Fast three-dimensional discrete cosine transform. *SIAM Journal on Scientific Computing*, 30(6):3087–3107, 2008.

[24] J. Lehtinen, J. Munkberg, J. Hasselgren, S. Laine, T. Karras, M. Aittala, and T. Aila. Noise2noise: Learning image restoration without clean data. *arXiv preprint arXiv:1803.04189*, 2018.

[25] J. Lei, K. Jia, and Y. Ma. Learning and meshing from deep implicit surface networks using an efficient implementation of analytic marching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[26] L. Liu, J. Gu, K. Zaw Lin, T.-S. Chua, and C. Theobalt. Neural sparse voxel fields. *Advances in Neural Information Processing Systems*, 33:15651–15663, 2020.

[27] Y. Lu, K. Jiang, J. A. Levine, and M. Berger. Compressive neural representations of volumetric scalar fields. In *Computer Graphics Forum*, vol. 40, pp. 135–146. Wiley Online Library, 2021.

[28] J. Maisano. Chamaeleo calyptratus. Digital Morphology, 2003 (Online). http://digimorph.org/specimens/Chamaeleo_calyptratus/whole.

[29] J. N. Martel, D. B. Lindell, C. Z. Lin, E. R. Chan, M. Monteiro, and G. Wetzstein. Acorn: Adaptive coordinate networks for neural scene representation. *arXiv preprint arXiv:2105.02788*, 2021.

[30] L. Mescheder, M. Oechsle, M. Niemeyer, S. Nowozin, and A. Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4460–4470, 2019.

[31] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*, pp. 405–421. Springer, 2020.

[32] T. Müller. Tiny CUDA neural network framework, 2021 (Online). https://github.com/nvlabs/tiny-cuda-nn.

[33] T. Müller, A. Evans, C. Schied, and A. Keller. Instant neural graphics primitives with a multiresolution hash encoding. *arXiv preprint arXiv:2201.05989*, 2022.

[34] T. Müller, B. McWilliams, F. Rousselle, M. Gross, and J. Novák. Neural importance sampling. *ACM Transactions on Graphics (TOG)*, 38(5):1–19, 2019.

[35] T. Müller, F. Rousselle, A. Keller, and J. Novák. Neural control variates. *ACM Transactions on Graphics (TOG)*, 39(6):1–19, 2020.

[36] T. Müller, F. Rousselle, J. Novák, and A. Keller. Real-time neural radiance caching for path tracing. *arXiv preprint arXiv:2106.12372*, 2021.

[37] S. Peng, M. Niemeyer, L. Mescheder, M. Pollefeys, and A. Geiger. Convolutional occupancy networks. In *European Conference on Computer Vision*, pp. 523–540. Springer, 2020.

[38] T. Peterka, Y. S. Nashed, I. Grindeanu, V. S. Mahadevan, R. Yeh, and X. Tricoche. Foundations of multivariate functional approximation for scientific data. In *2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 61–71. IEEE, 2018.

[39] M. Rieth, A. Gruber, F. A. Williams, and J. H. Chen. Enhanced burning rates in hydrogen-enriched turbulent premixed flames by diffusion of molecular and atomic hydrogen. *Combustion and Flame*, p. 111740, 2021.

[40] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33:7462–7473, 2020.

[41] V. Sitzmann, M. Zollhöfer, and G. Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. *Advances in Neural Information Processing Systems*, 32, 2019.

[42] M. Soler, M. Plainchault, B. Conche, and J. Tierny. Topologically controlled lossy compression. In *2018 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 46–55. IEEE, 2018.

[43] P. P. Srinivasan, B. Deng, X. Zhang, M. Tancik, B. Mildenhall, and J. T. Barron. Nerv: Neural reflectance and visibility fields for relighting and view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 7495–7504, 2021.

[44] W. D. A. Studios. Clouds data set - walt disney animation studios. https://www.disneyanimation.com/data-sets.

[45] S. K. Suter, M. Makhynia, and R. Pajarola. Tamresh–tensor approximation multiresolution hierarchy for interactive volume visualization. In *Computer Graphics Forum*, vol. 32, pp. 151–160. Wiley Online Library, 2013.

[46] T. Takikawa, J. Litalien, K. Yin, K. Kreis, C. Loop, D. Nowrouzezahrai, A. Jacobson, M. McGuire, and S. Fidler. Neural geometric level of detail: Real-time rendering with implicit 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11358–11367, 2021.

[47] M. Tancik, P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. Barron, and R. Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *Advances in Neural Information Processing Systems*, 33:7537–7547, 2020.

[48] S. Theodoridis and K. Koutroumbas. *Pattern recognition*. Elsevier, 2006.

[49] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[50] S. Weiss, P. Hermüller, and R. Westermann. Fast neural representations for direct volume rendering. *arXiv preprint arXiv:2112.01579*, 2021.

[51] S. W. Wurster, H.-W. Shen, H. Guo, T. Peterka, M. Raj, and J. Xu. Deep hierarchical super-resolution for scientific data reduction and visualization. *arXiv preprint arXiv:2107.00462*, 2021.

[52] Y. Xie, E. Franz, M. Chu, and N. Thuerey. TempoGAN: A temporally coherent, volumetric gan for super-resolution fluid flow. *ACM Transactions on Graphics (TOG)*, 37(4):1–15, 2018.

[53] Z. Zhou, Y. Hou, Q. Wang, G. Chen, J. Lu, Y. Tao, and H. Lin. Volume upscaling with convolutional neural networks. In *Proceedings of the Computer Graphics International Conference*, pp. 1–6, 2017.

```
RayGenerationKernel<<<...>>>(...)
size_t num_valid_rays = StreamCompaction(...)
while (num_valid_rays > 0) {
  CoordinateComputationKernel<<<...>>>(...)
  BatchInference(...)
  ShadingKernel<<<...>>>(...)
  num_valid_rays = StreamCompaction(...)
}
```

Listing 2. Pseudo-Code for our sample-streaming renderer.

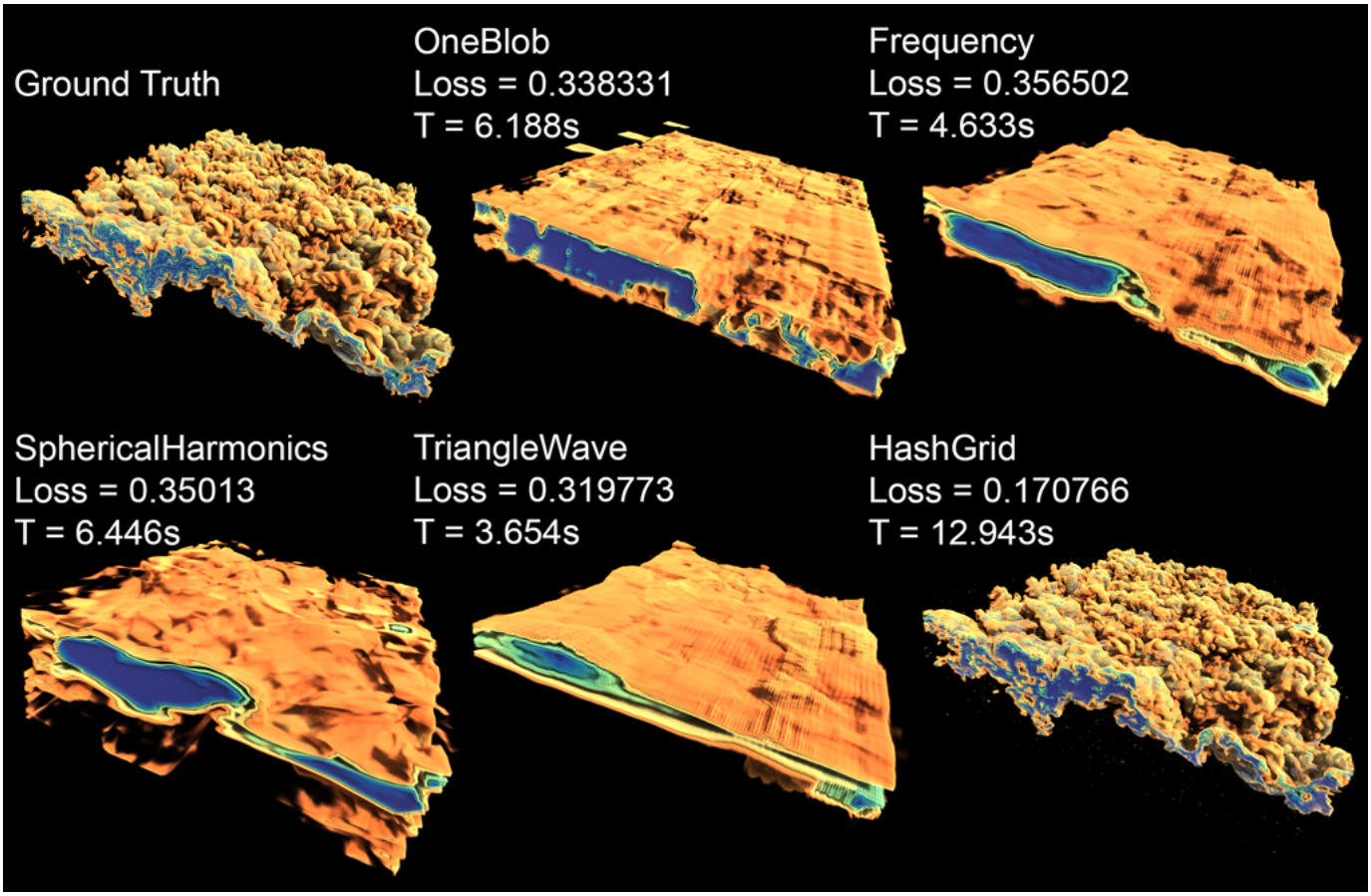

Fig. 14. Compare the effectiveness of different input encoding method. We train each neural network for 500, using L1 loss and the Adam optimizer with learning rate = 0.01. Documentations for each encoding method can be found in [32]. The `1atmHR` data is rendered.

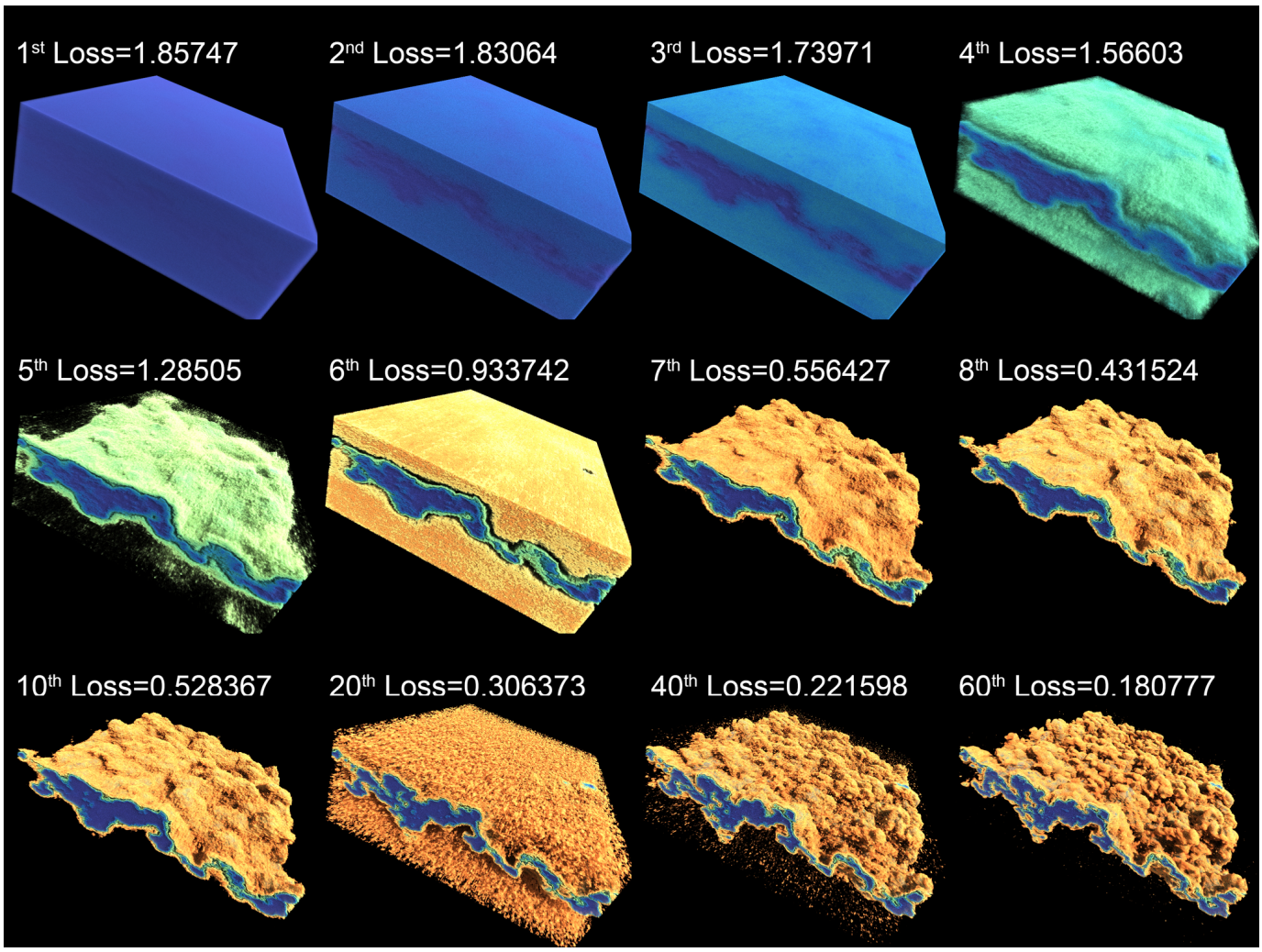

Fig. 15. Screenshots of a online-trained volumetric neural representation at different training steps. The data being trained is `1atmHR`. This sequence is captured from the same experiment done for Figure 2, therefore their loss values are comparable.

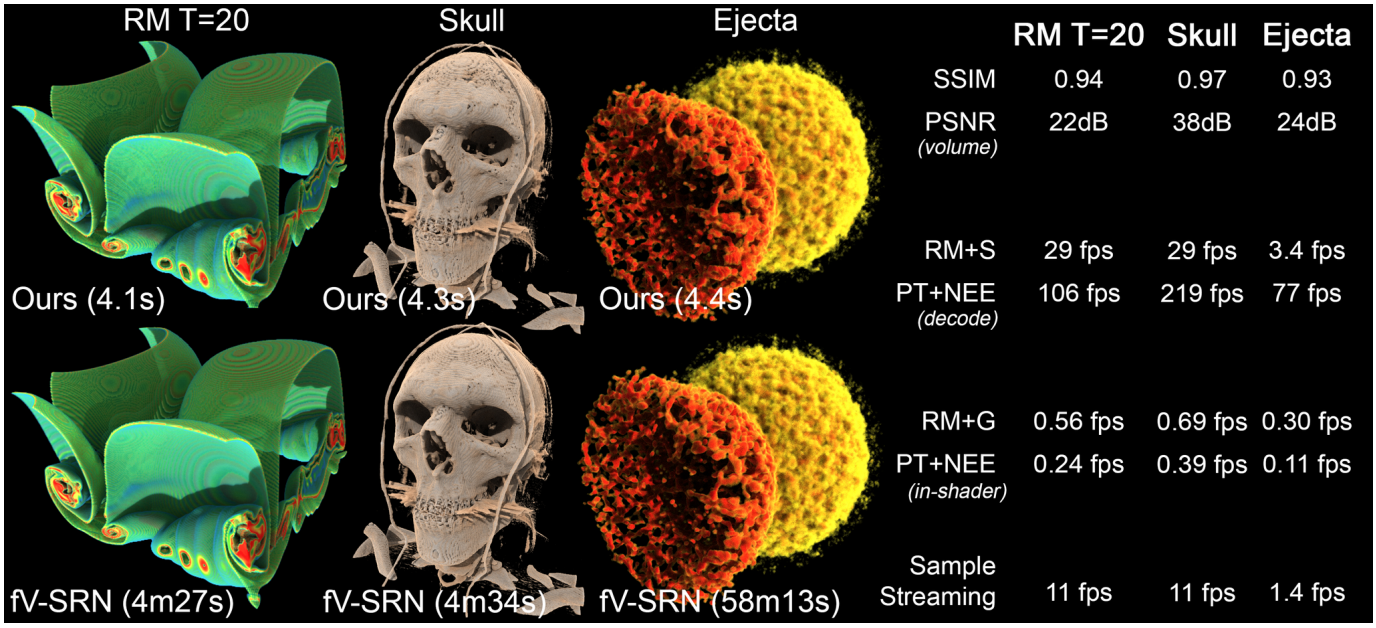

Fig. 16. Comparison between our technique with the fV-SRN [50]. We could not find the full resolution `Ejecta` dataset used by fV-SRN, so it was trained using a low-resolution version available online.

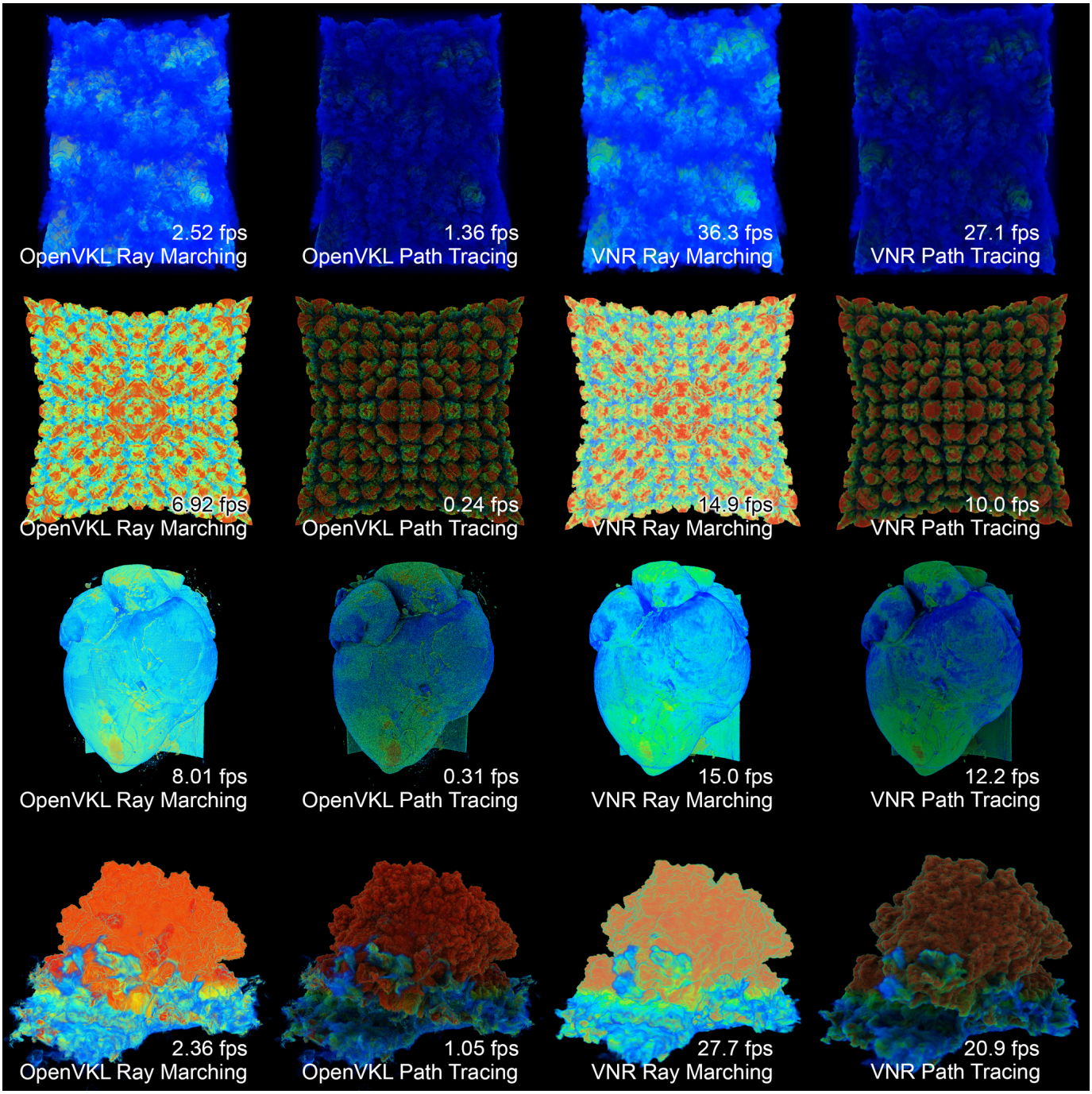

Fig. 17. Performance comparisons between the OpenVKL example renderer and our volume-decoding-based volumetric neural representation online-trained using our GPU-based method. The renderings are produced by the experiment described in Table 1.

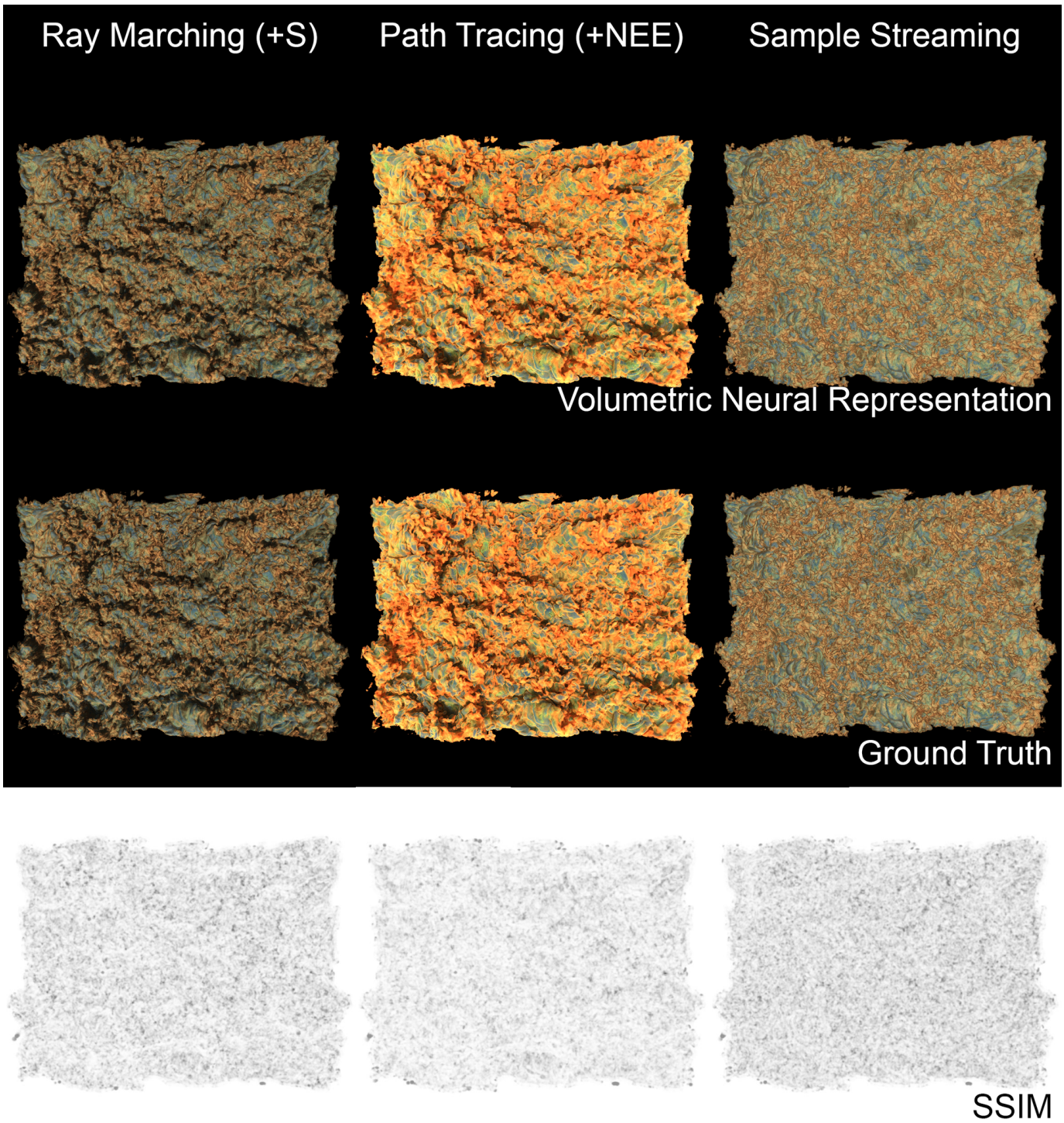

Fig. 18. Rendering of the `1atmHR` dataset produced by the corresponding experiment described in Table 2.

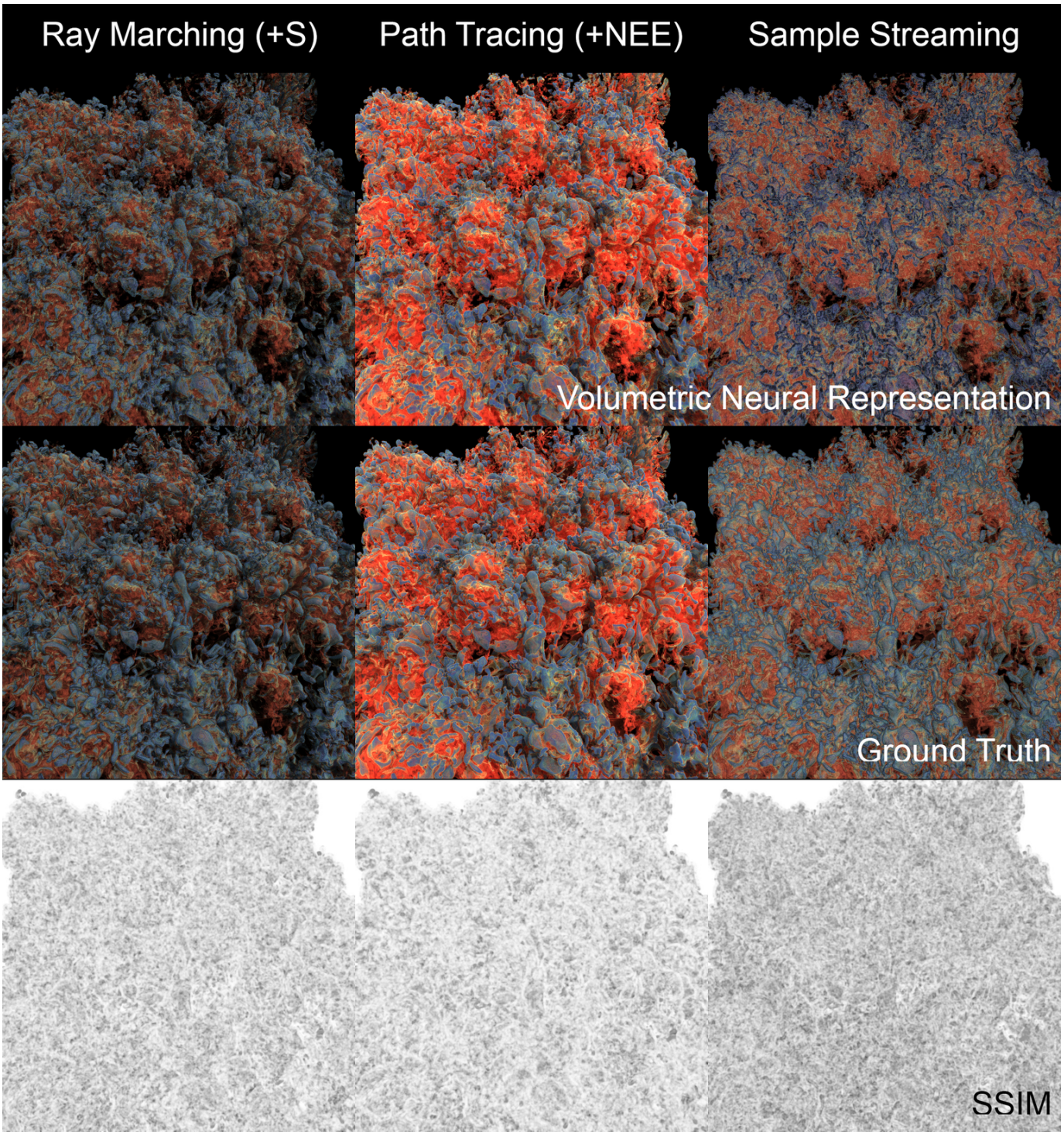

Fig. 19. Rendering of the `10atmHR` dataset produced by the corresponding experiment described in Table 2.

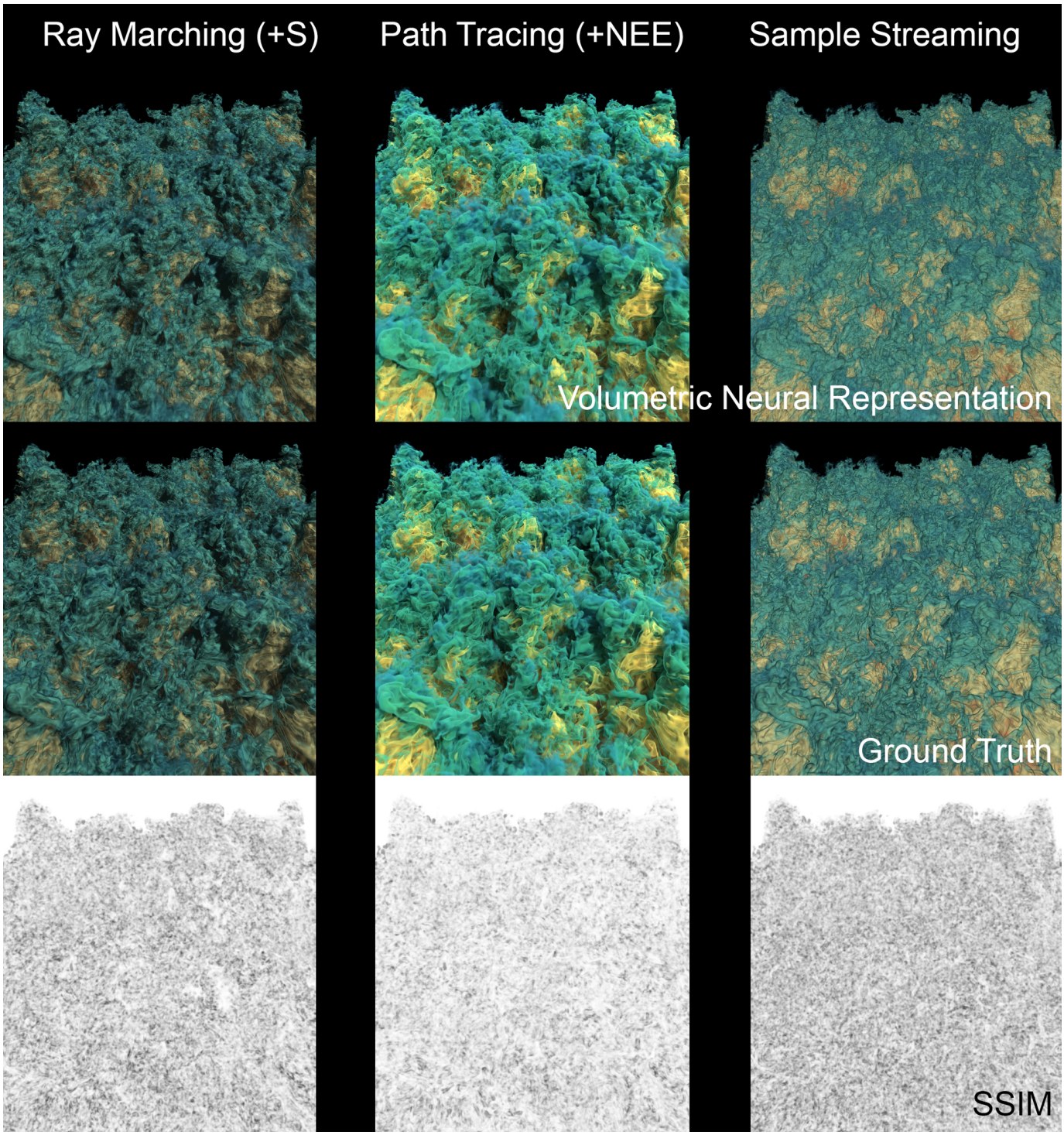

Fig. 20. Rendering of the `1atmTemp` dataset produced by the corresponding experiment described in Table 2.

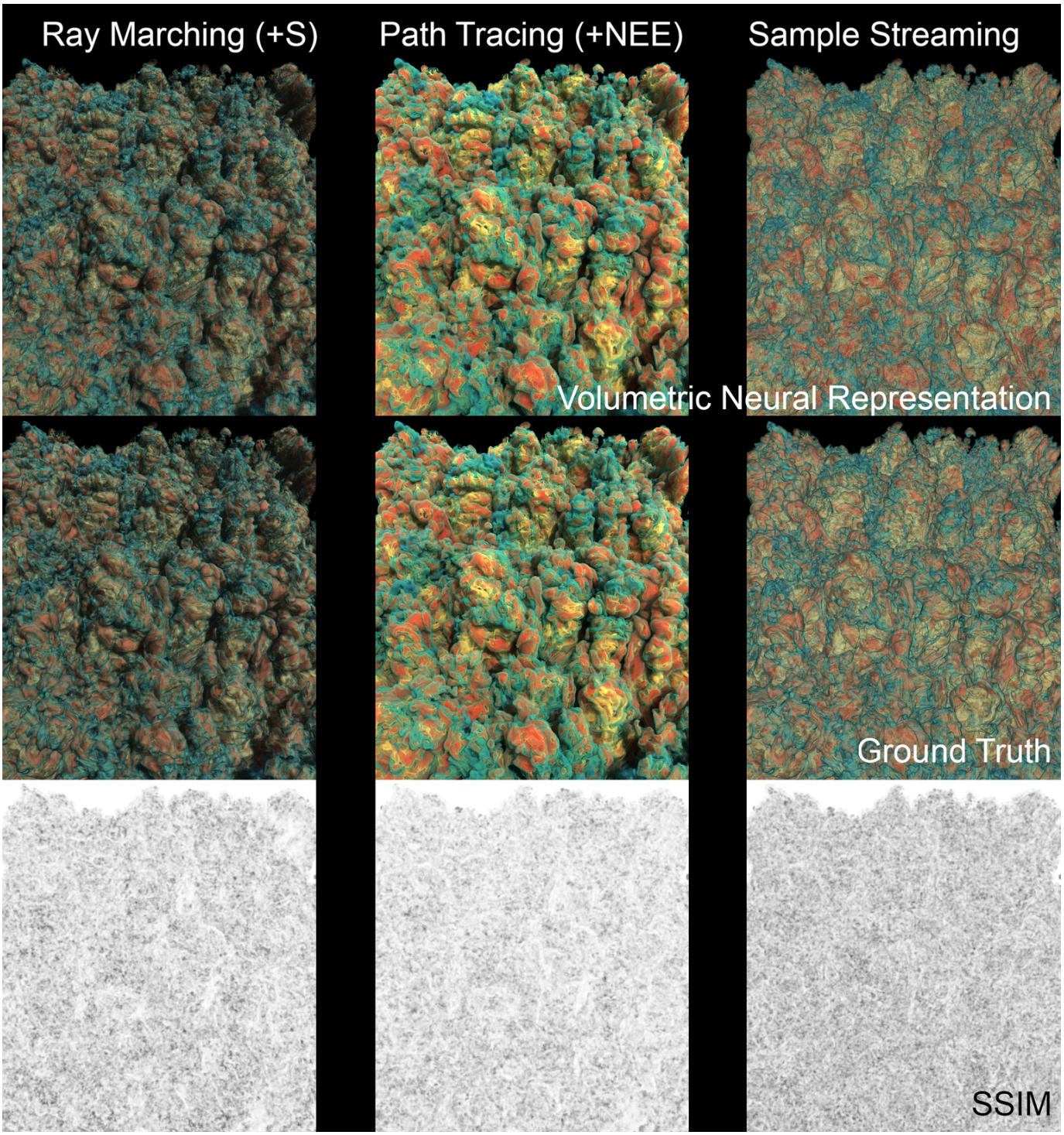

Fig. 21. Rendering of the `10atmTemp` dataset produced by the corresponding experiment described in Table 2.

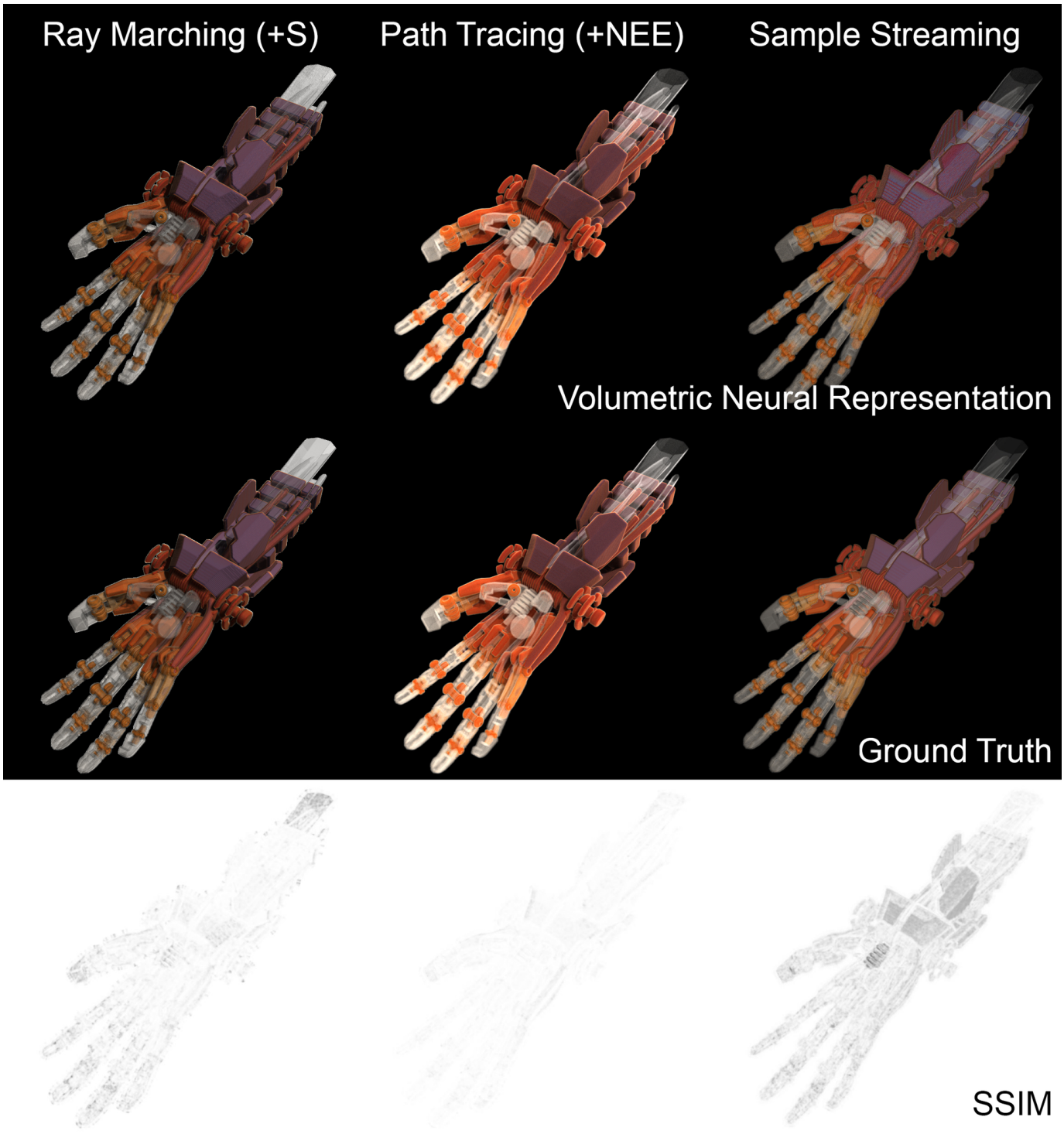

Fig. 22. Rendering of the `MechHand` dataset produced by the corresponding experiment described in Table 2.

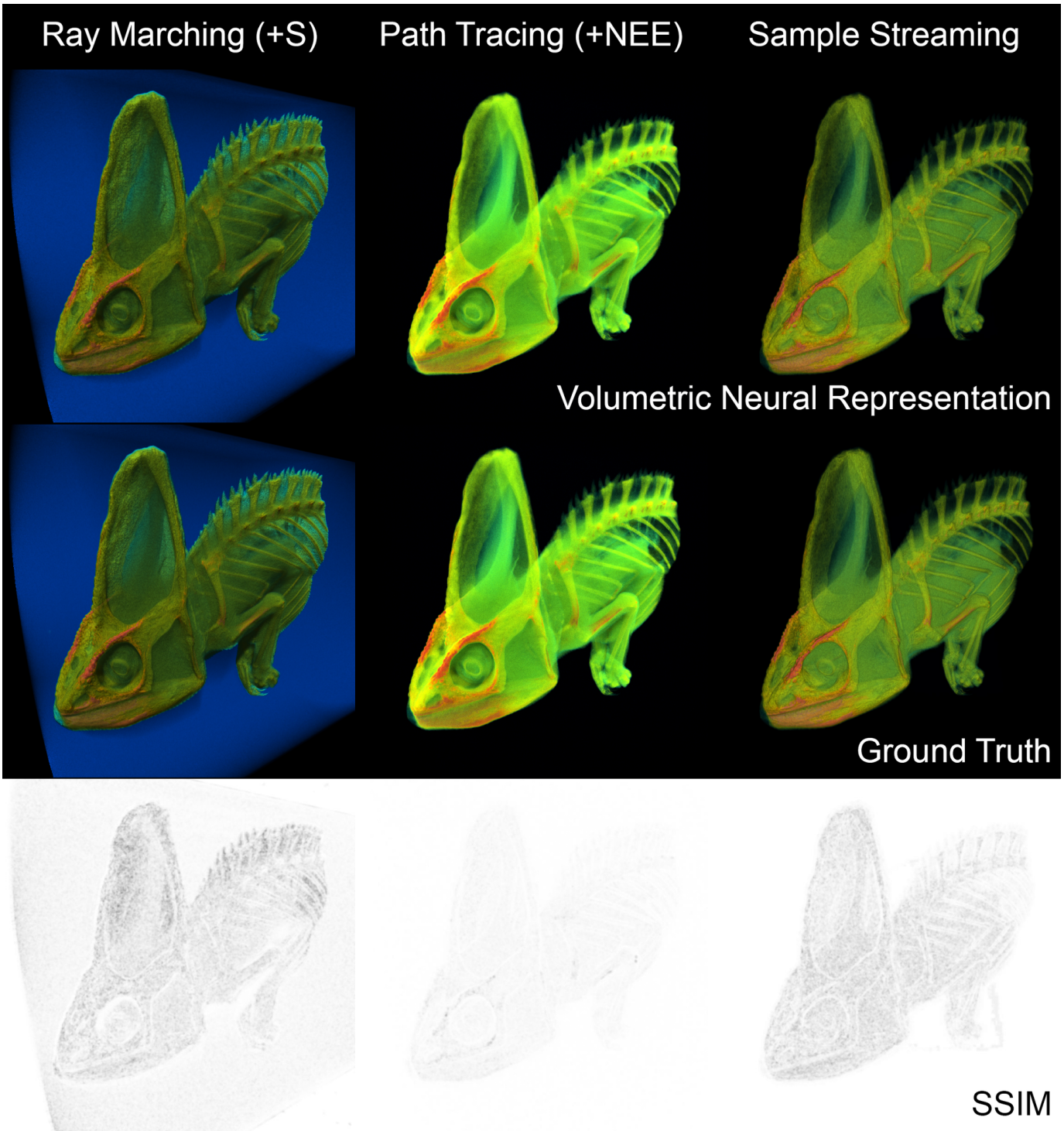

Fig. 23. Rendering of the `Chameleon` dataset produced by the corresponding experiment described in Table 2.

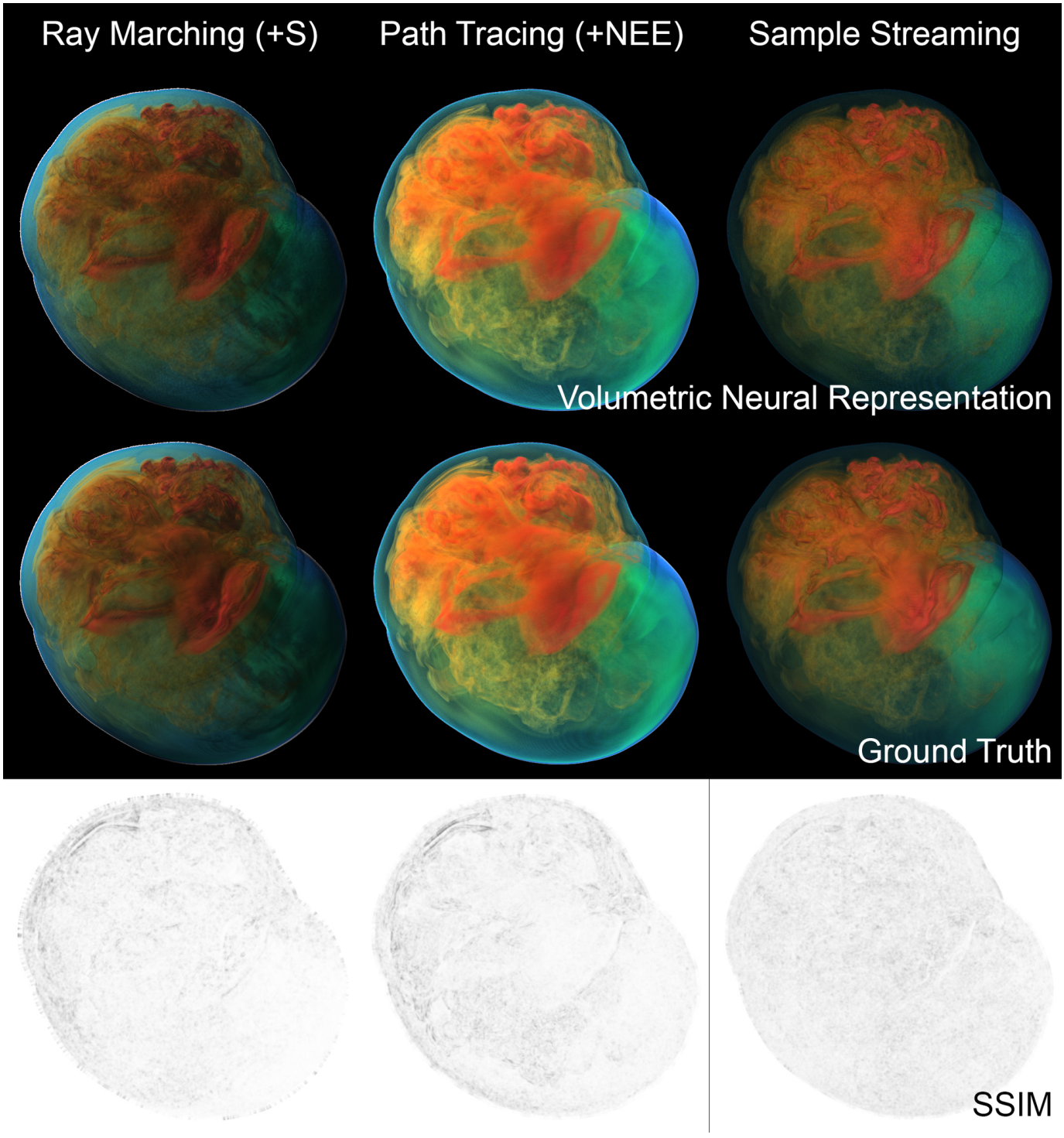

Fig. 24. Rendering of the SuperNova dataset produced by the corresponding experiment described in Table 2.